Private Public Partnership Project (PPP)

Large-scale Integrated Project (IP)

D.3.1.1b: FI-WARE GE Open Specification

Project acronym: FI-WARE

Project full title: Future Internet Core Platform

Contract No.: 285248

Strategic Objective: FI.ICT-2011.1.7 Technology foundation: Future Internet Core Platform

Project Document Number: ICT-2011-FI-285248-WP3-D.3.1.1b

Project Document Date: 2012-10-15

Deliverable Type and Security: Public

Author: FI-WARE Consortium

Contributors: FI-WARE Consortium

Table of Contents

1 Introduction

1.1 Executive Summary

This document describes the Generic Enablers in the Apps and Services Ecosystem chapter, their basic functionality and their interaction. These Generic Enablers form the core business framework of the FI-WARE platform by supporting the business functionality for commercializing services.

The functionality of the frame work is illustrated with several abstract use case diagrams, which show how the individual GE can be used to construct a domain-specific application environment and system architecture. Each GE Open Specification is first described on a generic level, describing the functional and non-functional properties and is supplemented by a number of specifications according to the interface protocols, API and data formats.

1.2 About This Document

FI-WARE GE Open Specifications describe the open specifications linked to Generic Enablers GEs of the FI-WARE project (and their corresponding components) being developed in one particular chapter.

GE Open Specifications contain relevant information for users of FI-WARE to consume related GE implementations and/or to build compliant products which can work as alternative implementations of GEs developed in FI-WARE. The later may even replace a GE implementation developed in FI-WARE within a particular FI-WARE instance. GE Open Specifications typically include, but not necessarily are limited to, information such as:

- Description of the scope, behavior and intended use of the GE

- Terminology, definitions and abbreviations to clarify the meanings of the specification

- Signature and behavior of operations linked to APIs (Application Programming Interfaces) that the GE should export. Signature may be specified in a particular language binding or through a RESTful interface.

- Description of protocols that support interoperability with other GE or third party products

- Description of non-functional features

1.3 Intended Audience

The document targets interested parties in architecture and API design, implementation and usage of FI-WARE Generic Enablers from the FI-WARE project.

1.4 Chapter Context

The Generic Enablers for the Apps Chapter together can be used to build the core infrastructure for enabling a sustainable ecosystem of applications and services of future internet application domains, which foster innovation as well as cross-fertilization. In particular the Apps Generic Enablers supports unified description and publishing of services, matching demand and offering via marketplace capabilities, creating composed value added services and service networks, and monetization and revenue sharing, all in a complementary and harmonized business framework.

The concept of the Generic Enabler implies that there can be several possible implementations. These implementations might even differ in their implementation approach. There are various degrees of flexibility in the non-functional properties or functional profile of the Generic Enabler description. For example the Composition Editor GE has 4 different implementations, relying on four different concepts and mechanisms for composition (event-based, data-driven, mash-ups, business process composition). The reason is that there is no one-fits-all solution for all kinds of composition. Although the generic functionality is to create composite services and applications out of existing services by composing or connecting the parts, resulting in a new service, the mechanisms and tools for composition are quite different. Which concrete enabler implementation is used, depends on the context, the requirements of the use cases and the availability of interfaces in the target infrastructure. So a GE defines the generic functionality and gives a number of specifications that are to be used to be compliant.

Not every GE has a RESTful Web interface. Especially the composition editors expose their functionality mainly through a User Interface. This case requires the interface to be described in an abstract way (e.g. what a user can do) and illustrated by screenshots of specific enabler implementations.

A couple of basic enablers are important to realize the vision of such a service business framework which enables new business models in an agile and flexible way:

- **Repository** - defines a standard way of publishing service description in the Web in a scalable way.

- **Registry** - serves as a common database layer for run-time configuration and defines a common model and access interface.

- **Marketplace** - defines a standard way to access market places in order to find and compare offerings from different stores and provides further functionality to foster the market for future internet applications and services in a specific domain.

- **Revenue Sharing System** - provides a common scheme and protocols for the calculation and distribution of revenues according to the agreed business models.

- **Service Composition** - to compose existing services to value added composite services and applications, which can be monetized in the Business Framework.

- **Mediator** - enables the interoperability between future internet services and applications and also allow to interface to existing enterprise systems.

This set of self-contained enablers, described in this document, represents only an initial starting point for a future business framework. It is expected that supplemental enablers (e.g. for contracting, quotation ...) will be developed outside the FI-WARE projects.

The Business Framework has been designed to inter operate with each other relying on Linked USDL as common uniform description format for services, which does not only focus on technical aspects of service but also covers business aspects as well as functional and non-functional service attributes. Linked USDL itself is not a Generic Enabler, since it is a data format and vocabulary specification. Nevertheless, it will be introduced as an Open Specification, which is used by different enablers in their provided and consumed APIs.

The Applications and Services Generic Enablers are named according to their main functionality. While the role names, introduced in the FI-WARE Vision (Aggregator, Broker, Gateway ...), are used to describe the stakeholders of the service ecosystem in an abstract way, the enablers names now are referring to concrete software components.

The following diagram gives an example of how the Generic Enablers can be combined to form a concrete architecture for a Service Business Framework.
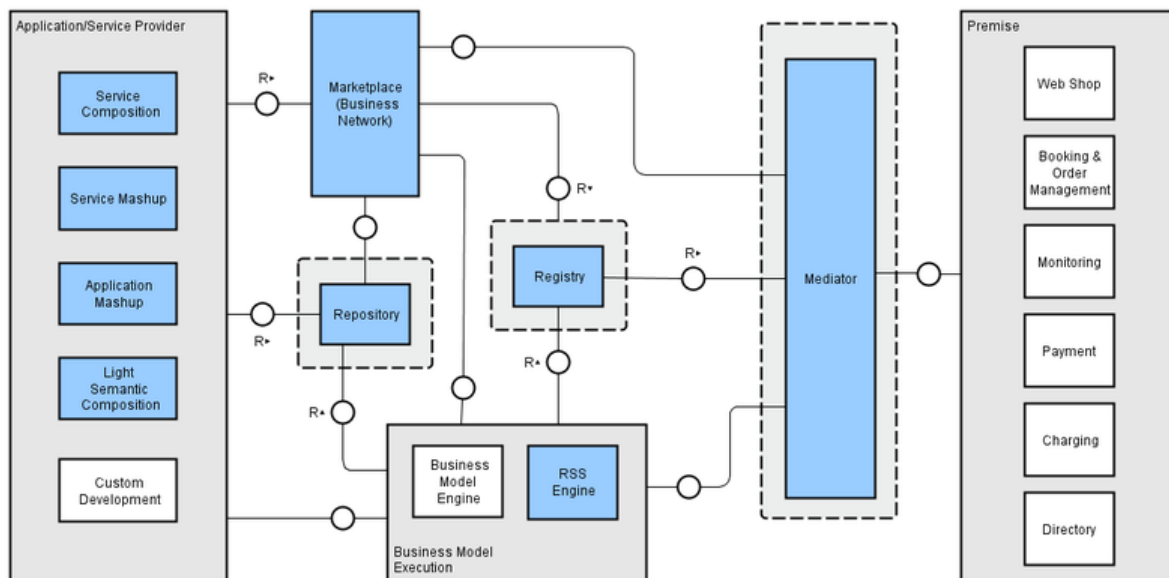
More information about the Apps Chapter and FI-WARE in general can be found within the following pages:

- http://wiki.fi-ware.eu
- Architecture_of_Applications_and_Services_Ecosystem_and_Delivery_Framework
- Materializing_Applications/Services_Ecosystem_and_Delivery_Framework_in_FI-WARE

1.5   Structure of this Document

The document is generated out of a set of documents provided in the public FI-WARE wiki. For the current version of the documents, please visit the public wiki at http://wiki.fi-ware.eu/

The following resources were used to generate this document:

D.3.1.1b FI-WARE GE Open Specifications front page

FIWARE.OpenSpecification.Apps.ServiceMashup

FIWARE.OpenSpecification.Apps.ApplicationMashup

FIWARE.OpenSpecification.Apps.WidgetAPI

FIWARE.OpenSpecification.Apps.ApplicationMashupAPI

FIWARE.OpenSpecification.Apps.ServiceComposition

FIWARE.OpenSpecification.Apps.ServiceCompositionREST

FIWARE.OpenSpecification.Apps.LightSemanticComposition

FIWARE.OpenSpecification.Apps.Marketplace

FIWARE.OpenSpecification.Apps.MarketplaceSearchREST

FIWARE.OpenSpecification.Apps.MarketplaceOfferingsREST

FIWARE.OpenSpecification.Apps.MarketplaceRegistrationREST

FIWARE.OpenSpecification.Apps.Repository

FIWARE.OpenSpecification.Apps.RepositoryREST

FIWARE.OpenSpecification.Apps.Mediator

FIWARE.OpenSpecification.Apps.MediatorREST

FIWARE.OpenSpecification.Apps.Registry

FIWARE.OpenSpecification.Apps.RegistryREST

FI-WARE Open Specifications Legal Notice

Open Specifications Interim Legal Notice

1.6 Typographical Conventions

Starting with October 2012 the FI-WARE project improved the quality and streamlined the submission process for deliverables, generated out of the public and private FI-WARE wiki. The project is currently working on the migration of as many deliverables as possible towards the new system.

This document is rendered with semi-automatic scripts out of a MediaWiki system operated by the FI-WARE consortium.

1.6.1 Links within this document

The links within this document point towards the wiki where the content was rendered from. You can browse these links in order to find the "current" status of the particular content.

Due to technical reasons not all pages that are part of this document can be linked document-local within the final document. For example, if an open specification references and "links" an API specification within the page text, you will find this link firstly pointing to the wiki, although the same content is usually integrated within the same submission as well.

1.6.2 Figures

Figures are mainly inserted within the wiki as the following one:

```
[[Image:....|size|alignment|Caption]]
```

Only if the wiki-page uses this format, the related caption is applied on the printed document. As currently this format is not used consistently within the wiki, please understand that the rendered pages have different caption layouts and different caption formats in general. Due to technical reasons the caption can't be numbered automatically.

1.6.3 Sample software code

Sample API-calls may be inserted like the following one.

```
http://[SERVER_URL]?filter=name:Simth*&index=20&limit=10
```

1.7 Acknowledgements

The following partners contributed to this deliverable: SAP, UPM, EAB, TI, DT, ATOS.

1.8 Keyword list

FI-WARE, PPP, Architecture Board, Steering Board, Roadmap, Reference Architecture, Generic Enabler, Open Specifications, I2ND, Cloud, IoT, Data/Context Management, Applications/Services Ecosystem, Delivery Framework , Security, Developers Community and Tools , ICT, es.Internet, Latin American Platforms, Cloud Edge, Cloud Proxy.

1.9 Changes History

Release	Major changes description	Date	Editor
v1	First draft of deliverable submission	2012-10-09	SAP
v2	Version for final review	2012-11-07	SAP
v3	Final version for deliverable submission	2012-11-07	SAP

2   FIWARE OpenSpecification Apps ServiceMashup

You can find the content of this chapter as well in the wiki of fi-ware.

Name	FIWARE.OpenSpecification.Apps.ServiceMashup
Chapter	Apps,
Catalogue-Link to Implementation	Mashup Factory
Owner	DT, Horst Stein

2.1   Preface

Within this document you find a self-contained open specification of a FI-WARE generic enabler, please consult as well the FI-WARE_Product_Vision, the website on http://www.fi-ware.eu and similar pages in order to understand the complete context of the FI-WARE project.

2.2   Copyright

* Copyright © 2012 by DT

2.3   Legal Notice

Please check the following Legal Notice to understand the rights to use these specifications.

2.4   Overview

Mashups are web applications which combine content and services from various sources in a value-adding manner. This composition of services and content can become a crucial business enabler in combination with the Internet of Services.

Mashups often focus on a very specific situational need and typically use web technologies (e.g. SOAP or RESTful services, or RSS feeds). For the mashup development, expert programming know-how and development environments have been required until now.

Web, telco, media services and content often are accessable via APIs (Application Programming Interface). APIs cover a set of functions that one computer program makes available to other programs so they can talk to the APIs directly. www.programmableweb.de lists more than 4200 APIs. Very popular APIs are google maps, twitter microblogging service, flickr photo sharing service, youtube video sharing and Google search.

The Service Mashup GE is implemented as a tool called Mashup Factory. Mashup Factory is an experimental toolset of DT, which allows end users without programming know-how to compose their own services for their immediate needs in communication, organisation and information.

The Mashup Factory toolset supports a graphical composition style, which allows the combination of services and content from several areas (e.g. communication, multimedia, geolocation). It supports integrated APIs from developergarden.com (e.g. SMS, Conference Call) by Deutsche Telekom and external sources (e.g. Google geo data, translator, weather forecast).

2.5   Basic Concepts

Mashup Factory is an experimental web-based application which supports a user to compose and execute mashups in an intuitive workflow. During the composition of the mashup, user activities like design, creation, configuration and simulation are enabled. After the mashup is designed, it can be activated, executed and monitored by the user. The mashup can be exposed to different consumer environments (e.g. portal, social network) where authentication and invocation can be performed. Technically speaking, the communication between the separated tasks are realised by dedicated service descriptions and service lists.

2.5.1   Build-in Service Repository

During the composition and design process the user can discover and use services from a repository.

Also we envision a large number of repositories containing service descriptions, which also might refer to descriptions in other repositories. Repositories can be hosted by the provider or a provider may use repository services of platform providers. The latter might be an alternative for small sized providers, which don't want to provide an own infrastructure. The figure below shows a screenshot of an example mashup which allows a user to initiate a phone conference with friends, whose phone numbers are provided from a data store. The service composition consists of basic services for conference call, data retrieval and a web dialog for the selection of conference participants from a user list.
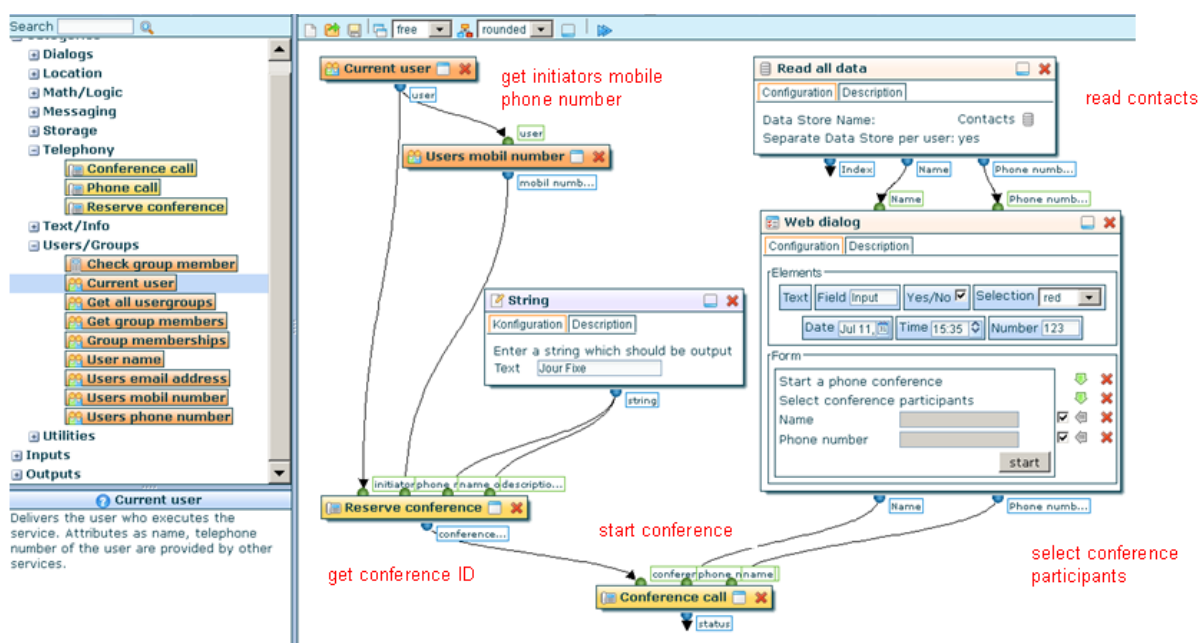
Composition editor of Mashup Factory

The services can be catagorized into three categories: application logic (e.g. sending SMS), data (e.g. storing data) and user interface (e.g. creating web dialog). Each service has a dedicated interface, i.e. input and output parameters of particular types. The repository can be extended by other services.

For experimentation purposes we have included services of different areas in the repository:

- a click-to-call (1-1) and a click-to-conference call for telephony

- email and SMS for messaging

- store, retrieve, modify and remove data for structured data storage

- a design-time based editor to configure user interaction structures with input and output parameters

- content services: weather, bible verse, and translation services

- geo position, geo distance, google places and displaying geo positions within Google maps

- several services to support supplementary functions (e.g. text concatenation), logical functions (e.g. filter), user data functions (e.g. phone number).

2.5.2   Composition Editor

The services are graphically represented as boxes; input and output parameters are represented as ports. The user designs connections between services by linking equivalent input and output ports, represented by links. This allows the service creator to develop the mashup in a dataflow oriented composition style by combining services.
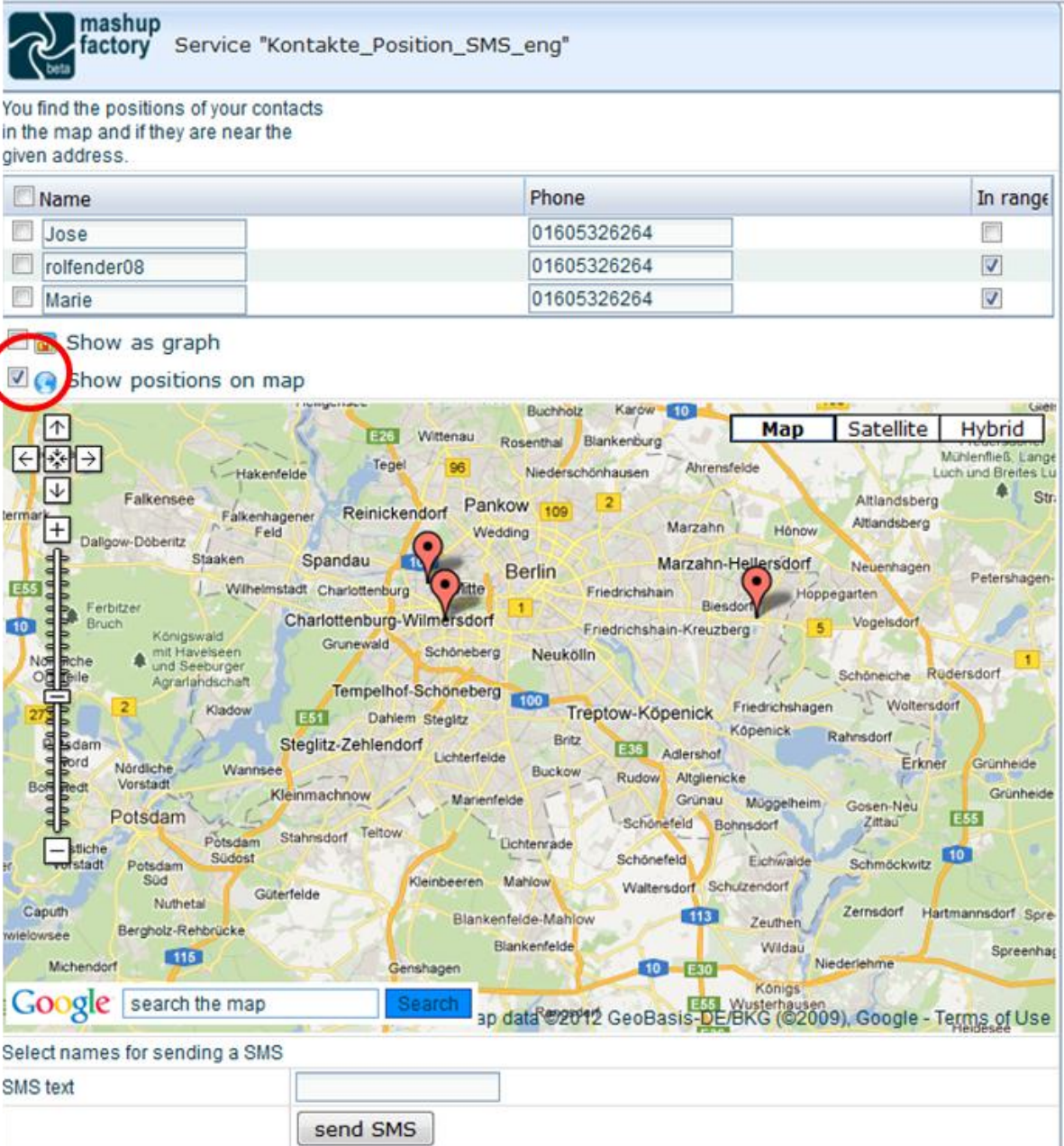
The user interface combines drag-and-drop features for services and simple textual editing functions for configurational purposes. Simple control constructs (e.g. filters, logical operations) are also available. Services are added via drag & drop from the repository to the work surface. A short service description is given in the left corner. Services have input and output ports, services are connected via linking input and output ports. There is a type checker which allows to connect only correct data types, e.g. date with date. Certain services are parameterized with configuration data, e.g. the web dialog. This service adds support for communication with a end-user in a web browser and transfer the user data to other services.

The composition determines which services are integrated in the mashup, and which output is produced. The output (see figure below) of a mashup composition can be

1. a web user interface (e.g. a digital diary)

2. an executed application (e.g. a telephone conference) or

3. a mixture of both (e.g. a web-based poll distributed by SMS).

When a service is composed (i.e. all input ports are connected with an output port), it can be saved as a composition by giving it a name. Service can be debugged by going step by step through the service watching the parameters transferred between the services.

The figure belows shows a screenshot of a service output with the geoposition of users displayed in google maps, with the option to send an SMS.

Output of a mashup

2.5.3   Service Execution

During the execution of a mashup the services are orchestrated in a data flow style, i.e. a service is executed when all input parameters are available.

The lifecycle of service execution can be managed by activating, stopping, renaming or removing services. The status of a service can be monitored and the period of activation defined. Dedicated users and user groups can be given permission to consume the services.

The execution is performed by a (standard) BPEL (Business Process Execution Language) engine. The services are called via SOAP requests and executed under the responsibility of the service provider. Thus, QoS of the services and the mashups cannot be guaranteed by Mashup Factory.

2.5.4   Users and Groups

Mashup Factory allows the management of users and groups. Some attributes (e.g. email address, phone number, login name) of the users can be edited and they can be organised to groups. User attributes can be retrieved by a service composition at run time. Every service composition can be given permission which users and groups (and their respective users) are allowed to execute it.

Caution: The users of Mashup Factory are responsible for the data of test users, keep legal constraints concerning data security.

2.5.5   Example Scenarios

In the following section the composition process and some illustrative application scenarios will be described. Imagine a person who frequently organises (trekking) tours of a group of friends. He wants to reduce his effort to contact his friends for finding adequate dates and notify the group with relevant information. He builds a mashup by Mashup Factory which uses a weather service, SMS and a web based survey.

Other possible scenarios for supporting immmediate needs in communication, organisation and information are

- providing a digital diary for monitoring health data with weight, blood pressure data where an SMS will be sent to a supervising person in case of exceeding a threshold

- conducting a web based survey for a (sports) club

- sending SMS to customers if they are close (related to a geo position) to a defined place (e.g. special offers in shop)

- inviting club members via voice mail to a day of an open door

- distributing particular content (e.g. bible verses, pollen information) to an interested audience

- performing video transmissions to selected persons, e.g. for health training, education, team meetings

For realising the scenarios, appropriate services must be integrated and perform their services with an adequate level of quality.

2.6   ServiceMashup Architecture
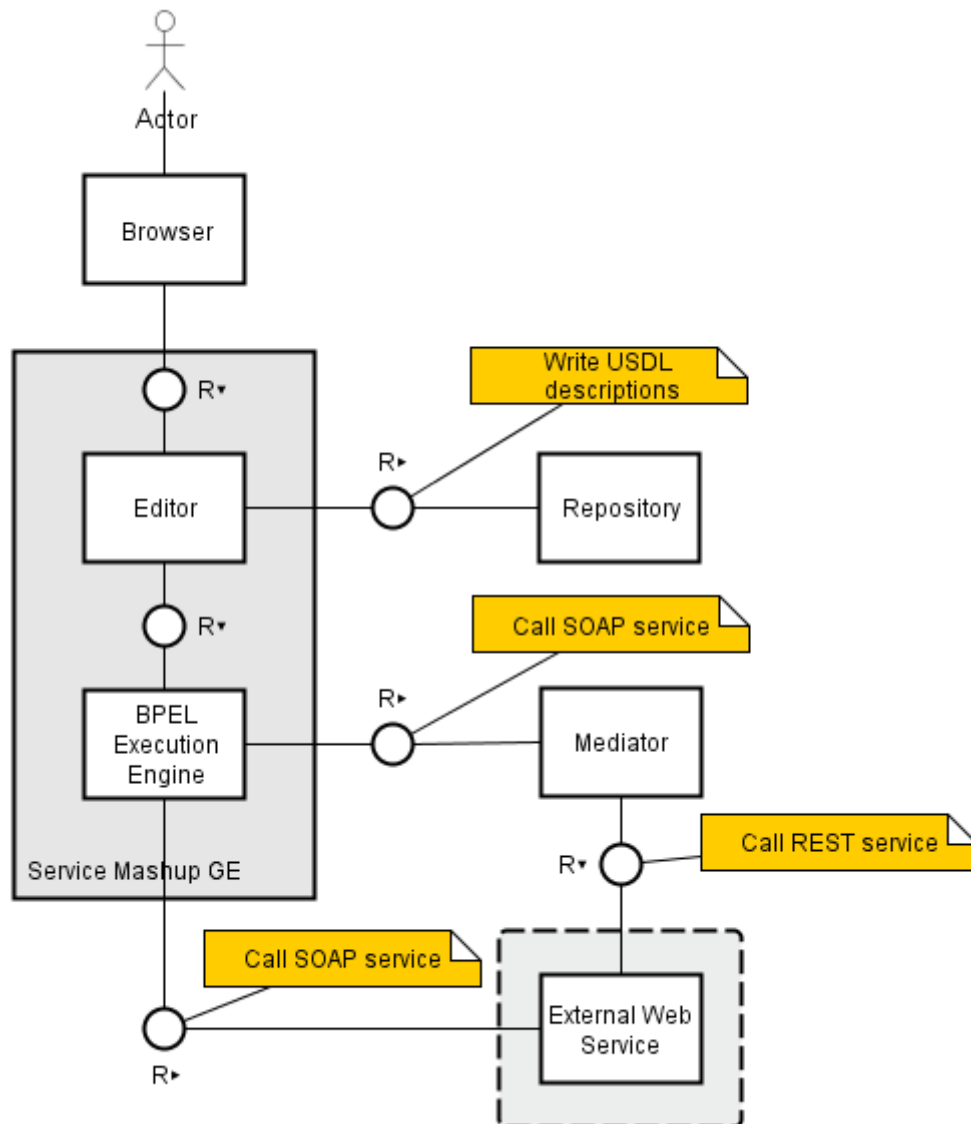
Service Mashup Architecture

The Mashup Factory follows the architecture of a Rich Internet Application. All user interaction takes place via Web browser as user agent, all application logic is implemented on a Web server. There are no software components to be installed on the client side except for the Web browser.

2.6.1   Technical Interfaces

The Web browser communicates with the Mashup Factory server using HTML and Ajax technologies. The Web server needs to comply to the Java EE servlet container specification. The Editor communicates to the BPEL Execution Engine and the Repository GE via REST interfaces, the Mediator GE and all other external Web services are called from the BPEL Execution Engine via SOAP protocol. For the purpose of storing the BPEL service

descriptions and other internal data an XML database is used that is also hosted in the servlet container.

2.7　Main Operations

To give an idea how to use and interact with the Mashup Factory to create Service Mashups this section provides a step-by-step introduction to compose examplary services.

2.7.1　Send SMS Service

A "send SMS" service composition depicts the easy composition process with Mashup Factory and illustrates the functions and components. Purpose of the service is a web dialog to send a SMS, which can be used by an authorized user by entering the message text and the phone number. In the Compose section you browse the repository for "Send SMS". Drag this box with the mouse and drop it on the work surface.
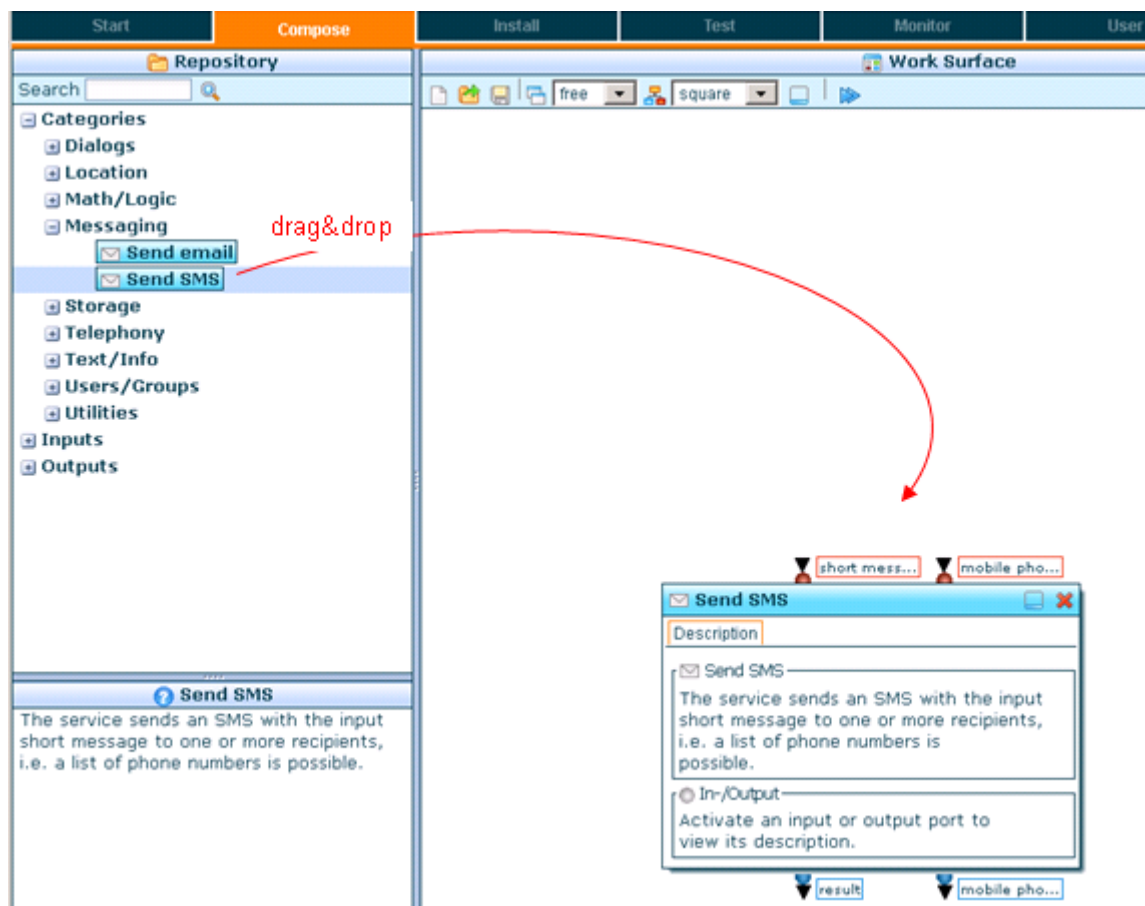
Send SMS Service - Step 1

The "Send SMS" service has two input ports which have to be connected with other services. Thus, fetch the Web Dialog from the repository and drag it on the work surface.
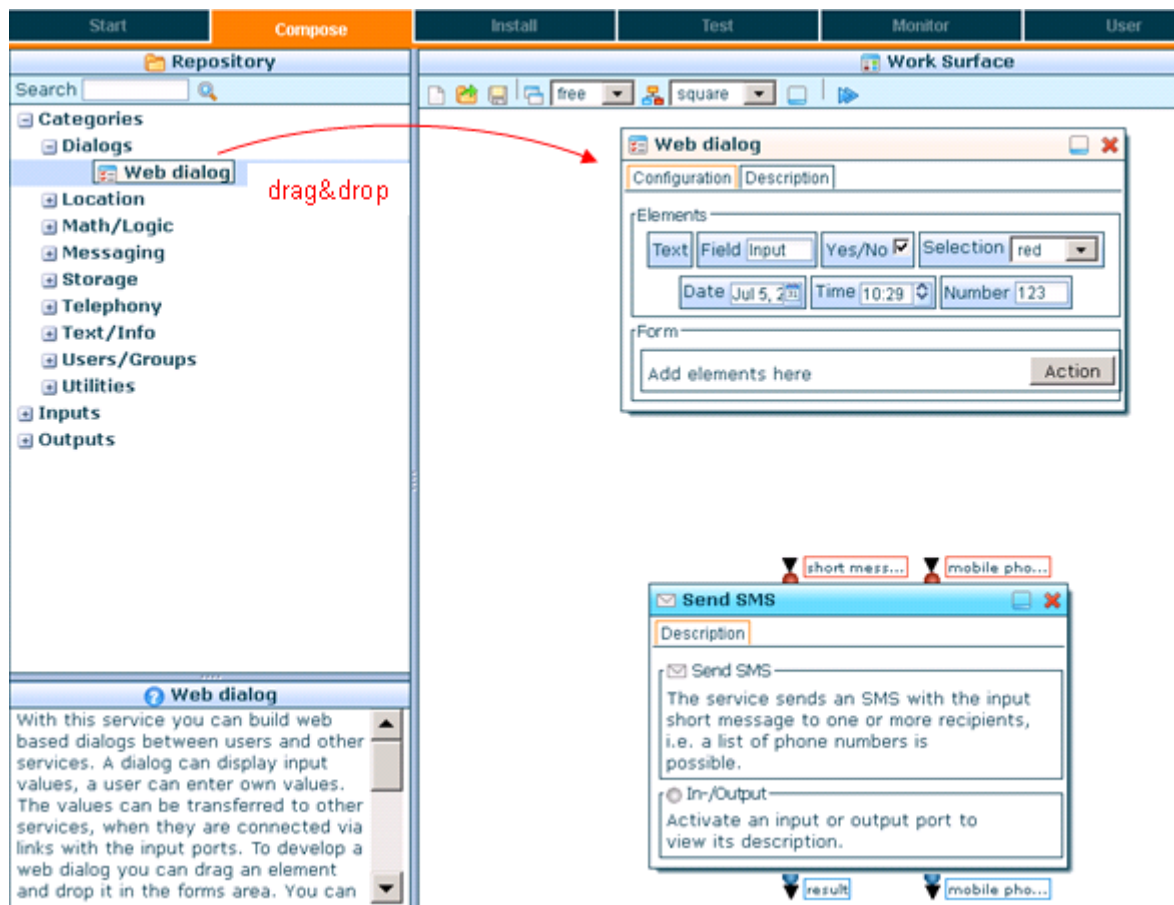
Send SMS Service - Step 2

In the Web dialog you can edit the user interface which later is displayed in the web browser. Relevant data fields of the user who consumes the service, are added herein. To create the appropriate dialog you can drag an element from the Elements area of the Web dialog and drop it in the Form space. Start with dragging a text element, position it, and edit it by double clicking on the Text field. Enter "Send SMS" and press return.
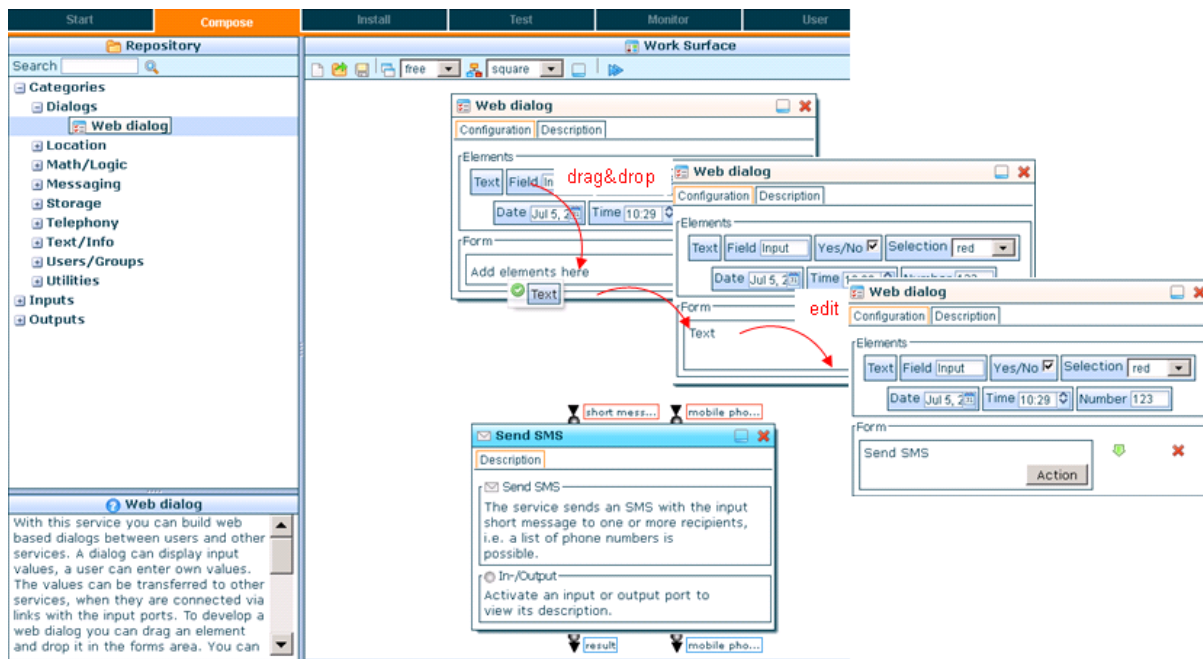
Send SMS Service - Step 3

Next, enter the message text and the phone number by using the field Element. You find two new output ports below the Web dialog box Connect the output ports with the corresponding input ports of the Send SMS.
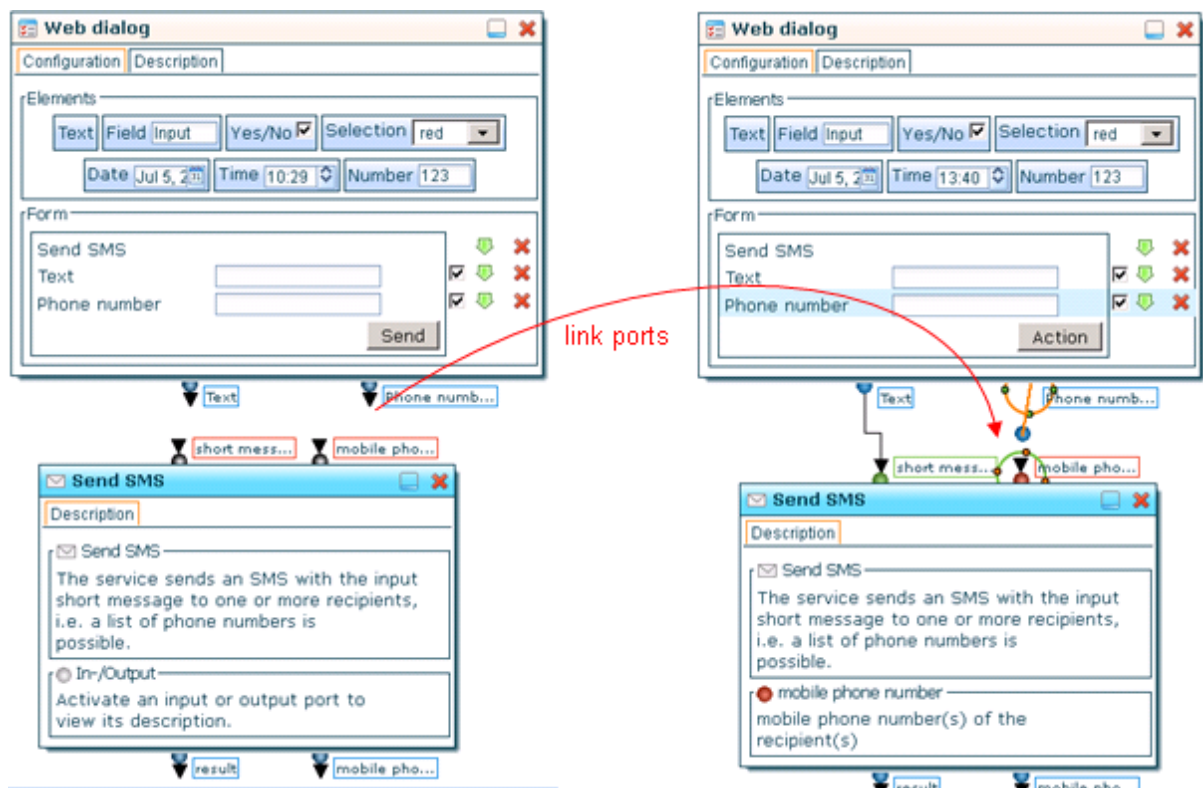
Send SMS Service - Step 4

That's it! Your first service composition is created. Now you save it by giving it a name. You get a result message in case of storage.
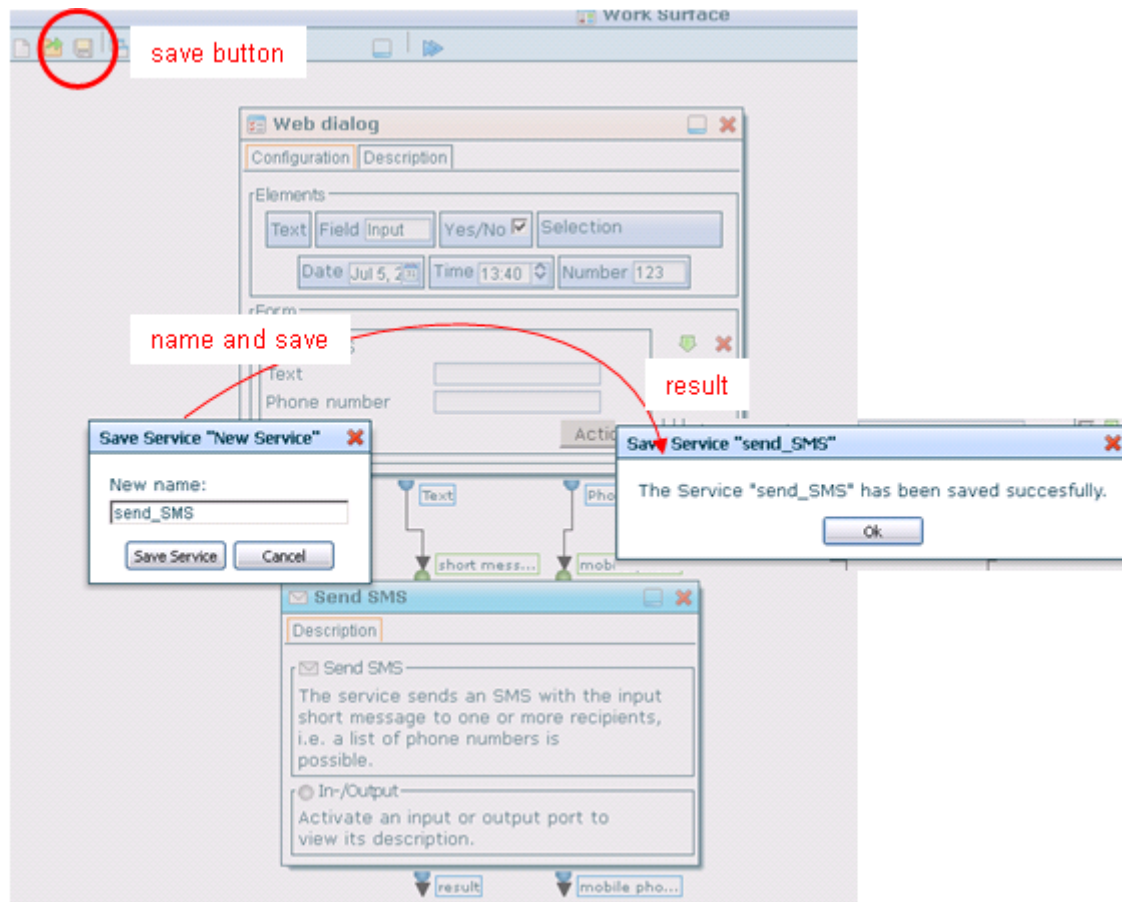
Send SMS Service - Step 5

Install the new service by pushing the execution button. If the status turns to green, the service can be executed.
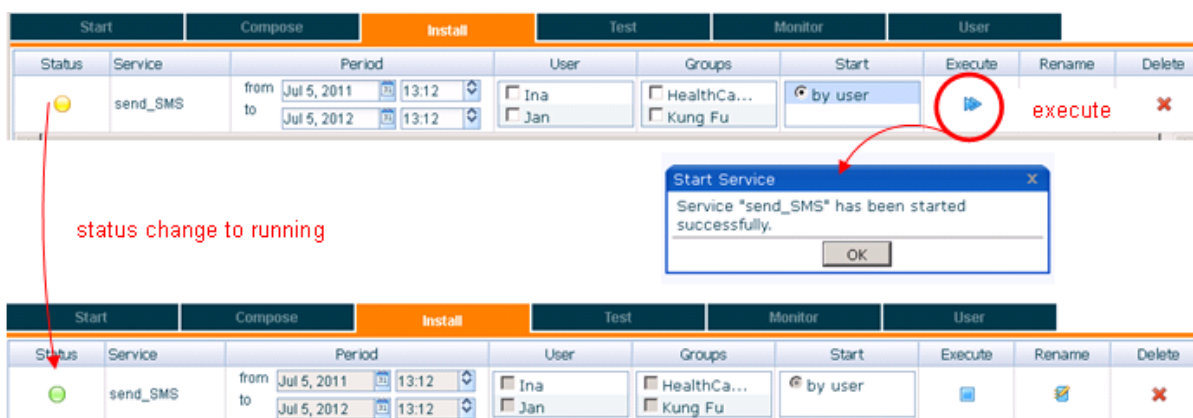
Send SMS Service - Step 6

Test the new service and login under your user name as a developer. You find the new service as link and start it by pushing the link. A new tab is opened with the web dialog you have designed. Fill in data, activated the action button and the SMS is sent!
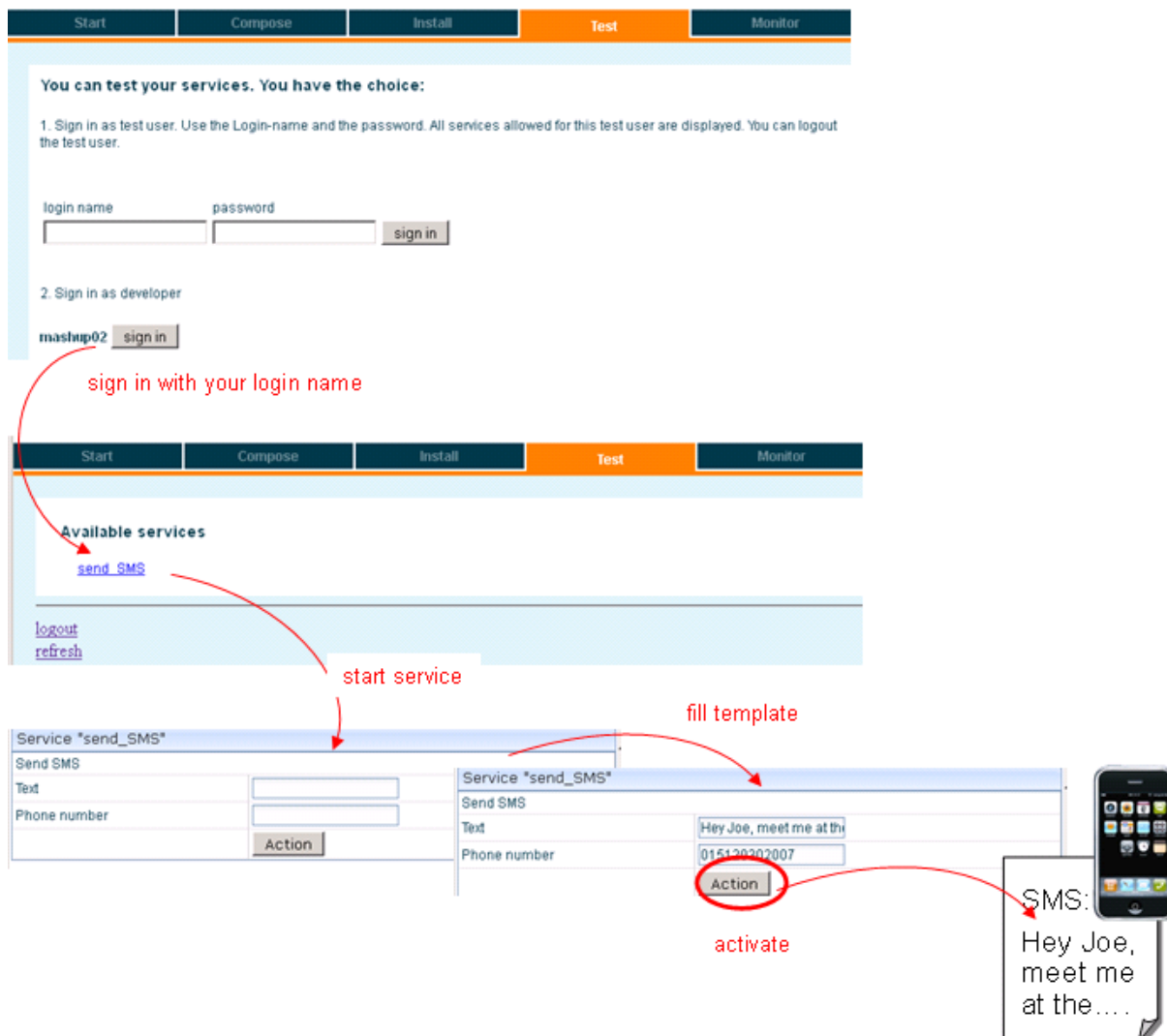
Send SMS Service - Step 7

2.7.2   Using Data Stores

A data store can be used to store and retrieve data for your service composition, e.g. diary data, addresses, locations etc. Once you have created a data store you can add, modify, read or remove the data. A simple addressbook with the two services (add a new contact and read my contacts) illustrates the handling. For adding a new contact you drag the 'New data store' from the repository. The input port is connected with one or more output ports from other services. Here a 'web dialog' with a Field Element Name and Field Element Phone number are used in order to allow the user to enter the data
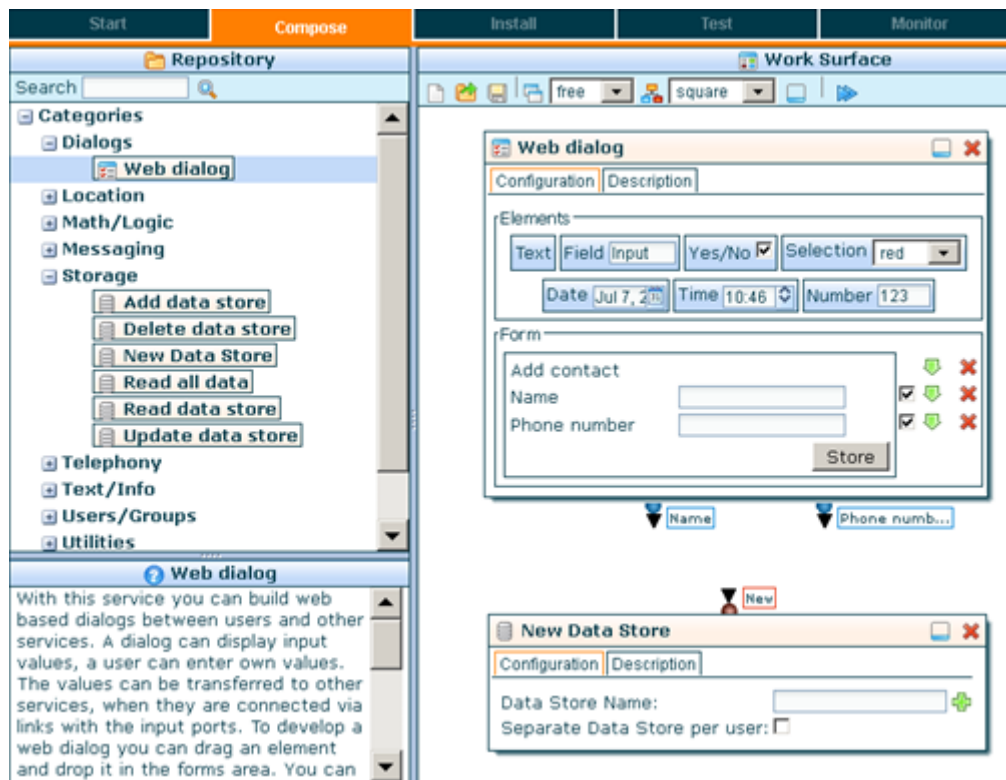
Using Data Stores - Step 1

After connecting of an output port (e.g. Name of 'Web Dialog') to the input port New (of 'New Data Store'), the port is added to 'New data store'. When all input data are defined for the data store, denominate it with a name (e.g. Contacts) for the data store. You can decide, whether the data store is user-specific (e.g. diary) or common (e.g. survey) by setting a flag.
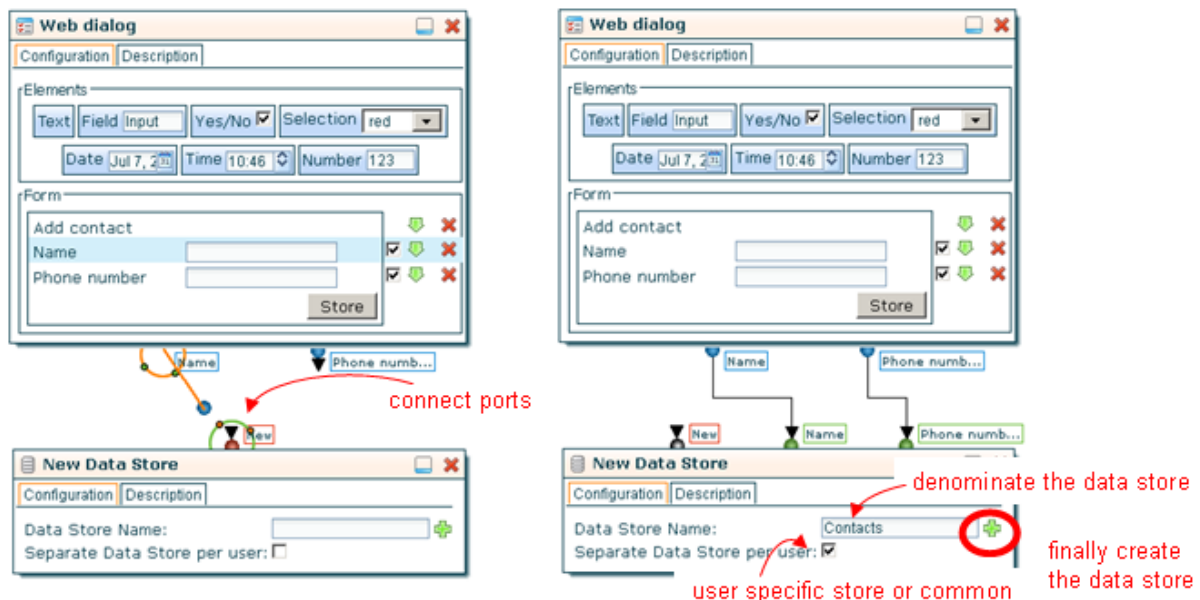
Using Data Stores - Step 2

Click the + to save the store. After storage the name of the service is changed to 'Add data store'. Then save the new composition by giving it a name (e.g. addContact).
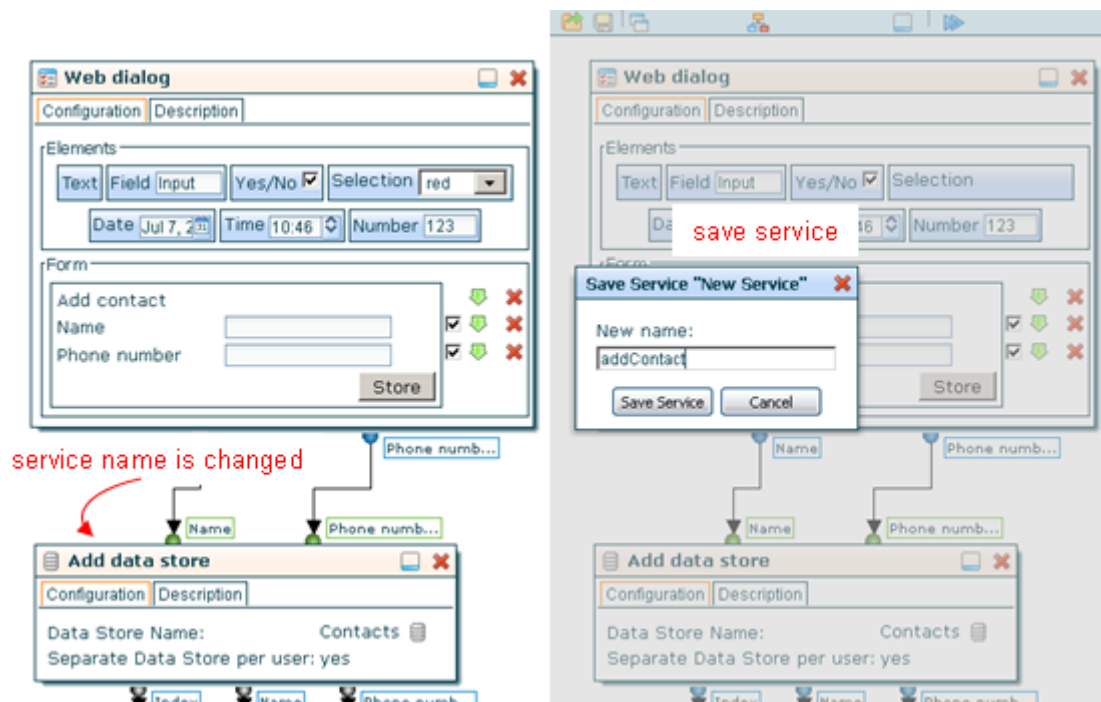
Using Data Stores - Step 3

You install it and test it by clicking on the link. Enter data in the dialog and repeat it with some contacts.
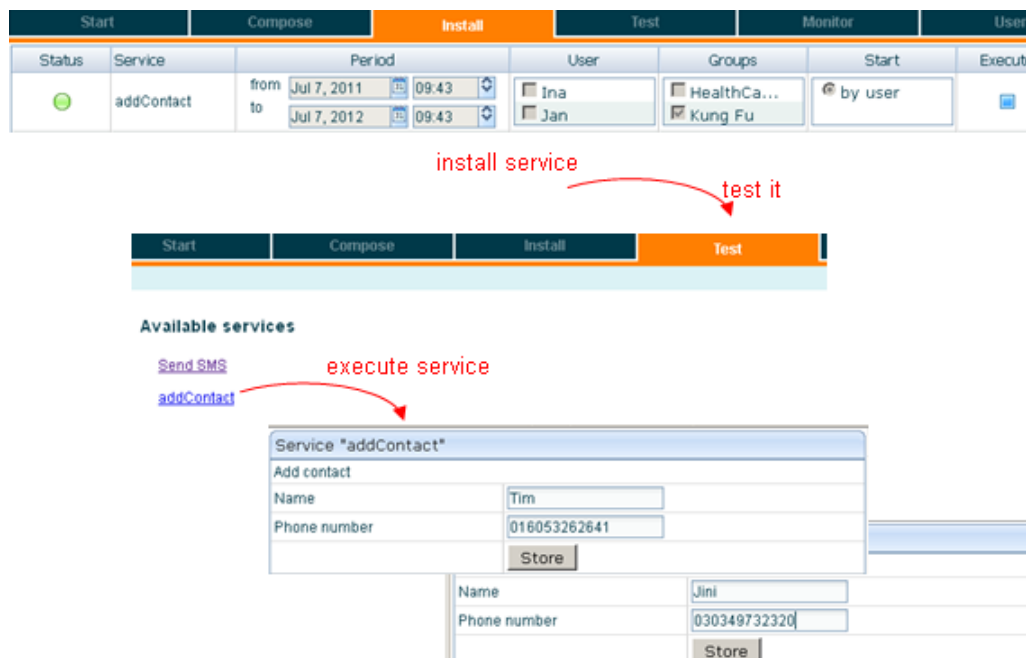
Using Data Stores - Step 4

For displaying the data of your contacts, create a new service for reading the contacts. Drop the 'Read all data' service and choose Contacts from the Data Stores. The service has the

output ports to read the data of Contacts (and in additional port Index, which can be used to modify the data).

Using Data Stores - Step 5

To display the data you can use a 'Web dialog'. Configure the dialog by dragging and dropping the Elements. Mark the fields Name and Phone number as input ports by pressing the green arrow. Connect the corresponding ports and save the composition with the name getContacts.
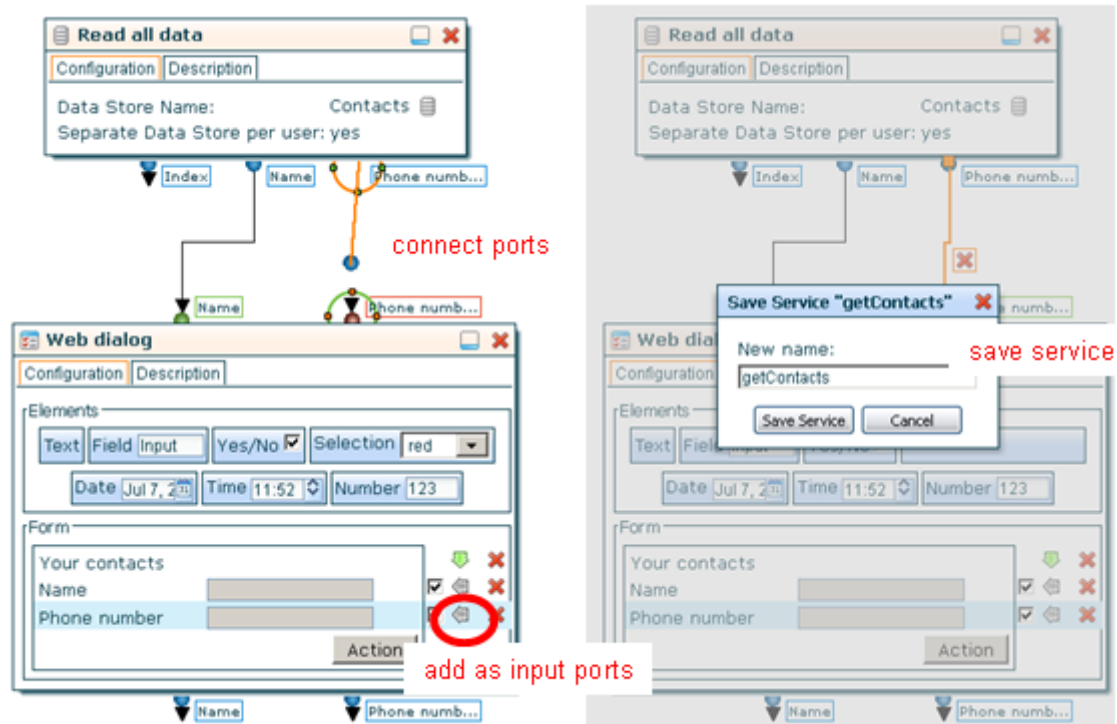
Using Data Stores - Step 6

Install and test getContacts. You get a list of contacts, which you have added to your contact store.
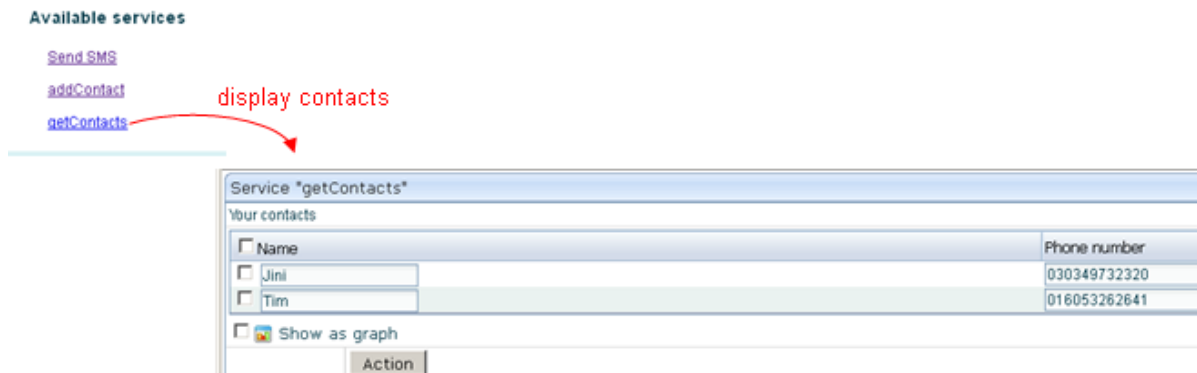
Using Data Stores - Step 7

2.8 Basic Design Principles

The design goal of Mashup Factory was to implement a special purpose BPEL editor for non-experts using a graphical approach to express Service Orchestration. Any technical aspects of the BPEL specification should be hidden from the user experience by using a data flow oriented service composition approach. The underlying BPEL structures should be fully transparent to the end user.

2.8.1 Data flow paradigm

Web services are invoked by providing input data as parameters and retrieving the return values as result of the Web service call. This concept was remodelled to a data flow approach where the input data for a graphical representation of a Web service "flows" into input ports and the result data returned from Web service invocation is presented at output ports. These ports can be connected by virtually wiring the ports in the graphical editor as if the output data of one Web service flow into the input ports of other Web services. By wiring the different Web services the end-user defines the sequence in which the services will be called at runtime by the BPEL execution engine resulting in an Service Orchestration.

2.8.2 Typed ports

Web service parameters are typed and can even handle complex types. In order to keep this typed data handling in the graphical composition approach the input and output ports of the Web service building blocks are also typed. The wiring logic allows only to connect ports of the same type (since it isn't of much sense to "feed" a user name into a port which expects geo coordinates). The data typing uses a two level hierarchical design. The lower level denotes a primitive type like Integer or String the higher level abstracts from the primitive type by defining a semantically meaningful type like "username", "date" or "geo position". Only the semantically meaningful types are presented at the user interface of the service editor. The only exception are generic service components like the dialog service, that allows end-users to build customized forms which will be visually displayed at service execution time. For the sake of clearness only form components/fields with primitive types may be used in the form editor (as there are too many high level types). The wiring logic takes the underlying primitive types into account when a port of the form service is connected to an ordinary port (e.g. a form port of type String is allowed to connect to a port of type "phone number" because the underlying type is also String).

2.9 Detailed Specifications

2.9.1 Open API Specifications

The Service Mashup GE Mashup Factory is not exposed as a service but as a Rich Internet Application, accessed throught the end user's Web browser. Although some GE components are exposed as services, they only expose an API for internal consumption (within the GE), but it is not foreseen that they will be integrated by other GEs.

The Mashup Factory will access the Mediator and Repository REST APIs:

- FIWARE.OpenSpecification.Apps.MediatorREST
- FIWARE.OpenSpecification.Apps.RepositoryREST

2.10 Re-utilised Technologies/Specifications

The Service Mashup GE requires both authentication and authorization in order to safeguard the different compositions from its users.

- It is recommended to use the OAuth2 protocol: http://tools.ietf.org/html/draft-ietf-oauth-v2-30

On the other hand, Service Mashup GE relies on the Mediator and Repository GEs, so it must support the following technologies and specifications:

- RESTful web services
- HTTP/1.1
- JSON and XML data serialization formats
- Linked USDL

2.11 Terms and definitions

This section comprises a summary of terms and definitions introduced during the previous sections. It intends to establish a vocabulary that will be help to carry out discussions internally and with third parties (e.g., Use Case projects in the EU FP7 Future Internet PPP). For a summary of terms and definitions managed at overall FI-WARE level, please refer to FIWARE Global Terms and Definitions

- **Ajax:** an acronym for Asynchronous JavaScript and XML is a group of interrelated web development techniques used on the client-side to create asynchronous web applications. With Ajax, web applications can send data to, and retrieve data from, a server asynchronously (in the background) without interfering with the display and behavior of the existing page. Ajax is not a single technology, but a group of technologies.

- **Application:** Applications in FI-Ware are composite services that have a IT supported interaction interface (user interface). In most cases consumers do not buy the application they rather buy the right to use the application (user license).

- **BPEL:** Business Process Execution Language (BPEL), short for Web Services Business Process Execution Language (WS-BPEL) is an OASIS standard executable language for specifying actions within business processes with web services. Processes in BPEL export and import information by using web service interfaces

exclusively.

- **Business Process:** Set of related and structured activities producing a specific service or product, thereby achieving one or more business objectives. An operational business process clearly defines the roles and tasks of all involved parties inside an organization to achieve one specific goal.

- **Click-to-call** also known as click-to-talk, click-to-chat and click-to-text, is a form of Web-based communication in which a person clicks an object (e.g., button, image or text) to request an immediate connection with another person in real-time either by phone call, Voice-over-Internet-Protocol (VoIP), or text. Click to talk requests are most commonly made on websites but can also be initiated by hyperlinks placed in email, blogs, wikis, flash animations or video, and other Internet-based object or user interfaces.

- **Composite Service (composition):** Executable composition of business back-end MACs. Common composite services are either orchestrated or choreographed. Orchestrated compositions are defined by a centralized control flow managed by a unique process that orchestrates all the interactions (according to the control flow) between the external services that participate in the composition. Choreographed compositions do not have a centralized process, thus the services participating in the composition autonomously coordinate each other according to some specified coordination rules. Backend compositions are executed in dedicated process execution engines. Target users of tools for creating Composites Services are technical users with algorithmic and process management skills.

- **Data flow:** A data flow is a graphical representation of the "flow" of data through an information system, modeling its process aspects. A Data flow shows what kinds of information will be input to and output from the system, where the data will come from and go to, and where the data will be stored.

- **Java EE:** Java Platform, Enterprise Edition or Java EE is Oracle's enterprise Java computing platform. The platform provides an API and runtime environment for developing and running enterprise software, including network and web services, and other large-scale, multi-tiered, scalable, reliable, and secure network applications. Java EE extends the Java Platform, Standard Edition (Java SE/J2SE), providing an API for object-relational mapping, distributed and multi-tier architectures, and web services. The platform incorporates a design based largely on modular components running on an application server.

- **Mashup:** Executable composition of front-end MACs. There are several kinds of mashups, depending on the technique of composition (spatial rearrangement, wiring, piping, etc.) and the MACs used. They are called application mashups when applications are composed to build new applications and services/data mash-ups if services are composed to generate new services. While composite service is a common term in backend services implementing business processes, the term 'mashup' is widely adopted when referring to Web resources (data, services and applications). Front-end compositions heavily depend on the available device environment (including the chosen presentation channels). Target users of mashup platforms are typically users without technical or programming expertise.

- **Mashable Application Component (MAC):** Functional entity able to be consumed executed or combined. Usually this applies to components that will offer not only their main behaviour but also the necessary functionality to allow further compositions with other components. It is envisioned that MACs will offer access, through applications and/or services, to any available FI-WARE resource or functionality, including

gadgets, services, data sources, content, and things. Alternatively, it can be denoted as 'service component' or 'application component'.

- **Mediator:** A mediator can facilitate proper communication and interaction amongst components whenever a composed service or application is utilized. There are three major mediation area: Data Mediation (adapting syntactic and/or semantic data formats), Protocol Mediation (adapting the communication protocol), and Process Mediation (adapting the process implementing the business logic of a composed service).

- **Portal:** A web portal is a web site that brings information from diverse sources in a unified way. Usually, each information source gets its dedicated area on the page for displaying information (a portlet); often, the user can configure which ones to display.

- **Provider:** Actor who publishes and offers (provides) certain business functionality on the Web through a service/application endpoint. This role also takes care of maintaining this business functionality.

- **QoS:** Quality of service is the ability to provide different priority to different applications, users, or data flows, or to guarantee a certain level of performance to a data flow.

- **Registry and Repository:** Generic enablers that able to store models and configuration information along with all the necessary meta-information to enable searching, social search, recommendation and browsing, so end users as well as services are able to easily find what they need.

- **REST:** REpresentational State Transfer is a style of software architecture for distributed systems such as the World Wide Web. REST has emerged as a predominant Web service design model. REST facilitates the transaction between web servers by allowing loose coupling between different services. REST is less strongly typed than its counterpart, SOAP. The REST language uses nouns and verbs, and has an emphasis on readability. Unlike SOAP, REST does not require XML parsing and does not require a message header to and from a service provider. This ultimately uses less bandwidth. REST error-handling also differs from that used by SOAP.

- **Rich Internet Application (RIA)** is a Web application designed to deliver the same features and functions normally associated with deskop applications. RIAs generally split the processing across the Internet/network divide by locating the user interface and related activity and capability on the client side, and the data manipulation and operation on the application server side. An RIA normally runs inside a Web browser.

- **RSS:** Rich Site Summary (originally RDF Site Summary, often dubbed Really Simple Syndication) is a family of web feed formats used to publish frequently updated works—such as blog entries, news headlines, audio, and video—in a standardized format. An RSS document (which is called a "feed", "web feed", or "channel") includes full or summarized text, plus metadata such as publishing dates and authorship.

- **Service:** We use the term service in a very general sense. A service is a means of delivering value to customers by facilitating outcomes customers want to achieve without the ownership of specific costs and risks. Services could be supported by IT. In this case we say that the interaction with the service provider is through a technical interface (for instance a mobile app user interface or a Web service). Applications could be seen as such IT supported Services that often ar also composite services.

- **Service Composition:** in SOA domain, a service composition is an added value service created by aggregation of existing third party services according to some predefined work and data flow. Aggregated services provide specialized business functionality on which the service composition functionality has been split down.

- **Service Orchestration:** in SOA domain, a service orchestration is a particular architectural choice for service composition where a central orchestrated process manages the service composition work and data flow invocations the external third party services in the order determined by the work flow. Service orchestrations are specified by suitable orchestration languages and deployed in execution engines who interpret these specifications.

- **Servlet Container:** A Servlet is a Java-based server-side web technology. A software developer may use a servlet to add dynamic content to a web server using the Java platform. The generated content is commonly HTML, but may be other data such as XML. To deploy and run a Servlet, a web container must be used. A web container (also known as a Servlet container) is essentially the component of a web server that interacts with the Servlets.

- **SMS:** Short Message Service (SMS) is a text messaging service component of phone, web, or mobile communication systems, using standardized communications protocols that allow the exchange of short text messages between fixed line or mobile phone devices.

- **SOAP:** originally defined as Simple Object Access Protocol, is a protocol specification for exchanging structured information in the implementation of Web Services in computer networks. It relies on Extensible Markup Language (XML) for its message format, and usually relies on other Application Layer protocols, most notably Hypertext Transfer Protocol (HTTP) and Simple Mail Transfer Protocol (SMTP), for message negotiation and transmission.

- **Unified Service Description Language (USDL):** USDL is a platform-neutral language for describing services, covering a variety of service types, such as purely human services, transactional services, informational services, software components, digital media, platform services and infrastructure services. The core set of language modules offers the specification of functional and technical service properties, legal and financial aspects, service levels, interaction information and corresponding participants. USDL is offering extension points for the derivation of domain-specific service description languages by extending or changing the available language modules.

- **User Interface:** In computer science and human–computer interaction, the user interface (of a computer program) refers to the graphical, textual and auditory information the program presents to the user, and the control sequences (such as keystrokes with the computer keyboard, movements of the computer mouse, and selections with the touchscreen) the user employs to control the program.

- **XML database** is a data persistence software system that allows data to be stored in XML format. These data can then be queried, exported and serialized into the desired format. XML databases are usually associated with document-oriented databases. The internal model of such databases depends on XML and uses XML documents as the fundamental unit of storage, which are, however, not necessarily stored in the form of text files.

3 FIWARE OpenSpecification Apps Application Mashup

You can find the content of this chapter as well in the wiki of fi-ware.

Name	FIWARE.OpenSpecification.Apps.ApplicationMashup
Chapter	Apps,
Catalogue-Link to Implementation	Wirecloud
Owner	UPM, Javier Soriano

3.1 Preface

Within this document you find a self-contained open specification of a FI-WARE generic enabler, please consult as well the FI-WARE_Product_Vision, the website on http://www.fi-ware.eu and similar pages in order to understand the complete context of the FI-WARE project.

3.2 Copyright

* Copyright © 2012 by UPM

3.3 Legal Notice

Please check the following Legal Notice to understand the rights to use these specifications.

3.4 Overview

Web application mashups integrate heterogeneous data, application logic, and UI components (widgets/gadgets) sourced from the Web to create new coherent and value-adding composite applications. They are targeted at leveraging the "long tail" of the Web of Services (e.g. the so-called Web APIs, which have proliferated during recent years and have doubled in number during 2012. See programmableweb.com) by exploiting rapid development, the Do-It-Yourself (DIY) metaphor, and shareability. They typically serve a specific situational (i.e. immediate, short-lived, customized, specific) need, frequently with high potential for reuse. It is this "situationality", which prevents them from being offered as 'off-the-self' functionality by solution providers.

Web application mashups can be manually developed using conventional web programming technologies, But this fails to take full advantage of the approach. Application mashup tools and platforms such as the one being specified by the FIWARE's *Application Mashup GE* are aimed at development paradigms that do not require programming skills and, hence, target end users (being them business staff, customers or citizens). They also help to leverage innovation through experimentation and rapid prototyping by enabling their users (a) to discover the best suited mashable components (widgets, operators and off-the-self

mashuplets) for their devised mashup from a vast, ever-growing distributed catalogue, and (b) to visually mash them up to compose the application.

Key features of the *Application Mashup GE* to be covered by its open specification are:

- Support for a Platform-independent Mashup Definition Language (MDL) and a Widget Definition Language (WDL), which are needed to describe the application mashup and its building blocks so that any platform implementing the GE's open specifications will be able to instantiate and execute them. A mashup's MDL links to the WDL descriptions of its constituent widgets.

- Support for XML/RDF template schemas for both the WDL and the MDL languages, containing all widget- and mashup-related contextual information, plus all the preferences, state properties, and wiring/piping, context and rendering information (i.e. elements that manage the platform-widget interaction) required to support application mashup persistence and to instantiate and run the application mashup on a platform that conforms to these open specifications.

- USDL extensions to the WDL and MDL: WDL-RDF / MDL-RDF vocabularies for representing WDL/MDL data as part of a USDL offering.

- Support for a zipped file format (WGT) that allows mashable components to be conveniently stored and distributed.

- Support for *wiring*: a mechanism empowering end users to easily connect widgets in a mashup to create a fully-fledged event-driven dashboard/cockpit with RIA functionality. Different *wiring* editors are allowed provided that they give a MDL description of the resulting mashup.

- Support for *piping*: a mechanism empowering end users to easily connect widgets to back-end services or data sources through an extendable set of operators, including filters, aggregators, adapters, etc. Different *piping* editors (commonly offered as part of a *wiring* editor) are allowed provided that they give a MDL description of the resulting mashup.

- Support for visual rendering of the widgets in the application mashup UI. Different editors are allowed as long as they provide a description of the resulting mashup in MDL.

- MAC (widget, operator and mashup) life-cycle management support.

- An application mashup execution engine model capable of deploying and running a mashup from a MDL file. It provides support for managing mashup state persistence, and for managing the wiring and piping mechanisms.

- Support for interaction with a *catalogue* of mashable components: the catalogue empowers end users to store and share their newly created application mashups with other colleagues and users by communicating with the Store GE.

The Open Specifications of the FI-WARE's Application Mashup GE follows. These GE Open Specifications contain all the information required in order to build compliant products that can work as alternative implementations of the Application Mashup GE and therefore may replace any implementation developed in FI-WARE within a particular FI-WARE Instance.

3.4.1 Target usage

FI-WARE strives to exploit the composability of the application and services technologies in order to support cross-selling and achieve the derived network scaling effects in multiple ways. The platform enables composition either from the front-end perspective --application

mash-ups- or the back-end perspective --composite services. Specifically, the Application Mashup GE targets composition from the front-end perspective and is expected to leverage the creation and the execution of value-added applications not only by application providers but also by intermediaries and end users acting as composers, a.k.a. prosumers. Prosumer are a consumer-side end users who cannot find an application that fits their needs and therefore modify/create an application mashup in an ad-hoc manner for their own consumption. As the capabilities and skills of the target users being considered are expected to be very diverse, all kinds of usability issues, conceptual simplification, recommendation and guidance, etc. are taken into consideration.

3.5 Basic Concepts

3.5.1 Key concepts and ideas

The Application Mashup GE describes a Web platform that helps users to easily and visually create and run their own Web application mashups. Its functionality can be divided into a client-side part running on the user web browser and a server-side part running on a web server.

The Application Mashup GE is based on the composition model presented in [1] (please refer to that publication for a detailed description of the underlying composition model of this enabler), which has been specifically designed to empower end users with few or no programming skills to create and share their own web composite applications in a fully visual fashion:

[1] Lizcano, D., Alonso, F., Soriano, J., and López, G. (2011) A new end user composition model to empower knowledge workers to develop rich Internet applications. In Journal of Web Engineering, Vol.10 No. 3, pp. 197-233, Rinton Press Inc. Sept. 2011.

Widgets are the key elements of the composition model that the Application Mashup GE must support. Together with connectors and mashups (mashups are considered as building blocks for other application mashups), they make up the complete set of Mashable Application Components (MAC, see Terms and Definitions) that the Application Mashup GE must support. A widget is a lightweight Web application that runs on the user's web browser, in the context of an Application Mashup GE implementation. Widgets are usually developed using current Web technologies (HTML(5), CSS, Javascript, ...) and they are bound to heterogeneous data coming from the Web (e.g. Web APIs). They can be regarded as the service front-end, because they offer users a graphical user interface (UI), so that they can easily get a visual representation of the service data and functionality to which the widget is bound.

The figure below shows an example of what iGoogle's widgets (one of the first products to implement this idea) look like:
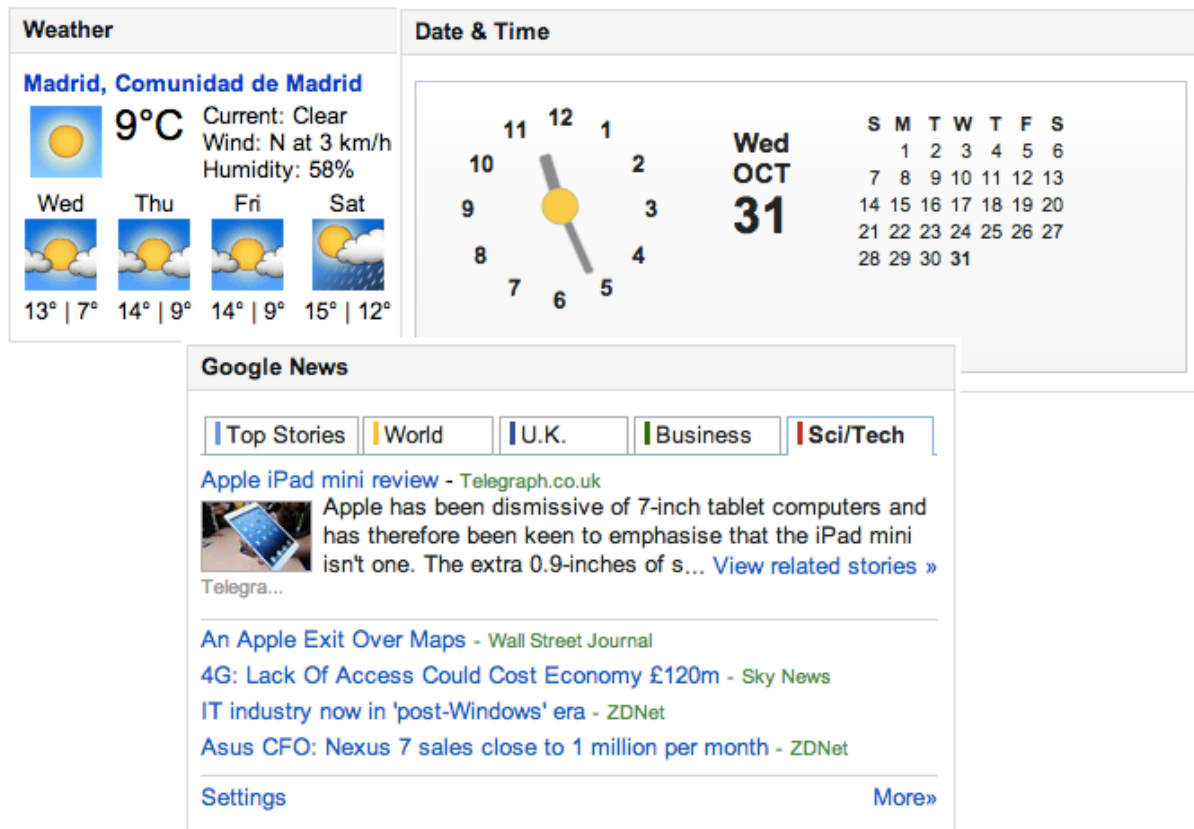
An example of some isolated widgets coexisting in the same desktop

These *early* widgets *per se* are isolated applications that do not interact with each other. However, the Application Mashup GE aims at providing a mechanism to visually compose a fully-fledged web application from different widgets that can now interact with each other via events and data sharing. This mechanism is what the composition model calls **wiring**. The idea behind wiring is easy: widgets expose (data/event) inputs and (data/event) outputs, so that an output from one widget can be linked to other widgets' inputs following a composition technique based on pre- and post-condition mechanisms. This way, the Application Mashup GE manages the data/event flow between widgets. The mechanism allows for the use of event-driven *programming* features, e.g. a widget can send and event through one of its outputs on an event trigger. The figure below shows an example of the wiring metaphor:

An abstract representation of how the Application Mashup GE could support the Wiring mechanism
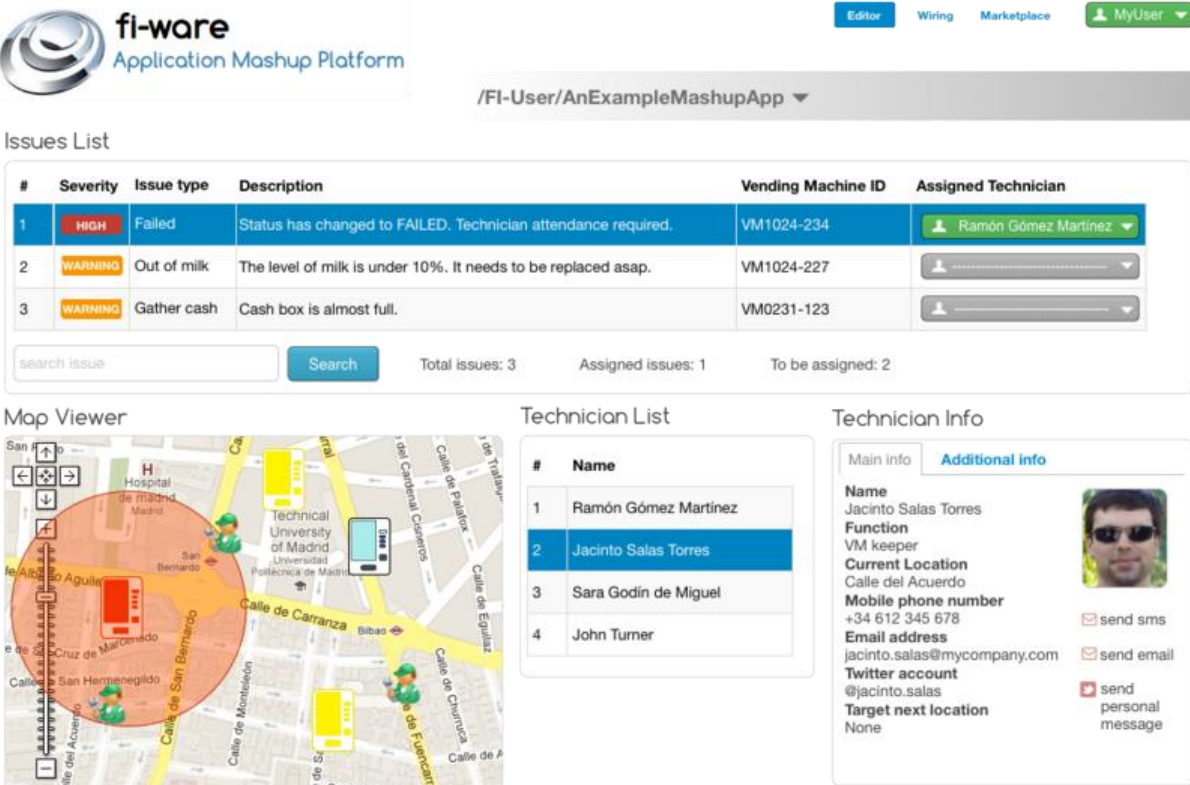
Widgets supported by the Application Mashup GE must be able to access their data from services in at least the following two ways: *programmatically* or by means of *operators* and through a visual technique called *piping* that establishes how these operators can be combined to form a *pipe*.

The Application Mashup GE supports the invocation of services *programmatically* from the widget's code: through what we call *WidgetAPI* (see the Architecture and Open API Specifications sections). But, following the ideas from the composition model, it also supports operator use and the *piping* mechanism. This targets end users (i.e. users with few or no programming skills): an *operator* does not offer a GUI but, like widgets, has an abstract representation with both inputs and outputs and can thus be wired to widgets allowing the data flow between them. Operators are usually bound to some kind of data source (SOAP service, Web API, etc.). In other words, operators are configured out-of-the-box to get access to a backend service, but they can also be made to subscribe to and get events from a publish/subscribe system. They can also act as filters, aggregators, mediators, etc. when used in the *piping* technique to build a *pipe*.

To sum up, an implementation of the Application Mashup GE must support the process of visually creating a composite web application by composing different widgets using the wiring mechanism, which interconnects those widgets, and the piping mechanism, that makes use of operators to get access to new data, perform an operation on that data, and finally pass it to the widgets through their inputs.

The figure below shows and example of what a web application mashup looks like.

An implementation of the Application Mashup GE showing how the mashup developers can arrange widgets to create their own Web mashup

3.5.2   Example scenario

To illustrate what is expected from the Application Mashup Generic Enabler, we have borrowed the following example scenario from the FInest Use Case Project (http://www.finest-ppp.eu/). The scenario is part of its *Fish transport from Ålesund to Europe* use case:

"A fish producer needs to ship frozen/dried fish from Norway to a customer overseas. The scenario covers the feedering phase, i.e. the shipping from Ålesund to Northern Europe. The fish cargo is first delivered at the Port of Ålesund (ÅRH) and stored and stuffed in container at the terminal (Tyrholm & Farstad: TF). The shipping line NCL covers the North Sea voyage (feedering) from Ålesund to Hamburg/Rotterdam, and further shipped overseas by a deep-sea container shipping line (e.g. APL). The process involves customs and food health declarations. The transport set-up is mostly fixed."

As is: The Port updates the website with information on the port's services, capacity, resources, and weather (in practice, port calls info updated systematically). This serves as information source for customers (ship agents, terminal operators) and all other stakeholders.

Challenges: Much manual info registration, and a lot of work duplication.

For improvement in the future, the port envisions the following improvements:

- A marketing portal, like a resource hub accessible from the website, enabling online management of bookings, resources and services as well as communication and coordination with third party service provider systems.

- Automatic update of Webpages ("ship calling", "at port", "departure", etc.) based on information from SafeSeaNet and actual data from AIS.

- Online registration of booking directly by the ship / ship agent.

In order to make these improvements, FInest demands from FI-WARE the following EPIC that will be covered by the Application Mashup GE:

"*FInest.Epic.IoS.WidgetPlatformInfrastructure:* A visual portal website is needed where each user can add, remove and use widgets. Therefore, also a widget repository is also needed from which a user can select widgets from. An infrastructure should be provided to deploy new widgets in the portal. It should be easy for end users to use."

There follows a the description of how the Application Mashup GE can be used to help to deal with the envisioned improvements:

- The functionality and information sources are split into a set of widgets. There is one widget for each resource that will be made accessible from the website: management of bookings and registrations, management of resources, management of services. Widgets from a third party service provider capable of communicating and coordinating with their systems, event-driven widgets connected to SafeSeaNet and actual data from AIS, e.g. "ship calling", "at port", "departure", etc. are also added to the catalogue of available widgets.

- These widgets are shared and offered through a repository (or store, or marketplace) which the project stakeholders and customers (ship agents, terminal operators, etc.) can search to select and retrieve the offerings of their interest.

- Each customer and stakeholder involved in this scenario, regardless of their level of technical or programming skills can leverage the application mashup editor to visually build a customized cockpit with the most valuable data and operations for their work by adding, removing and using available widgets and mashuplets (off-the-shelf

mashups that can be customized by adding and removing widgets to/from them). Moreover, they even can share the resulting application mashup for future use by other customers or stake holders (for further customization).

- A widget platform (or application mashup container) will serve as the envisioned visual portal website where these customers and stakeholders can easily deploy and use the widgets that make up the application mashup (i.e. the customized cockpit or information/operations dashboard that best fit their interests).

3.5.3 Architecture

This section describes the Application Mashup GE architecture. The diagrams use FMC (Fundamental Modelling Concepts) notation to facilitate the communication not only between technical experts but also between technical experts and business or domain experts. The Application Mashup GE provides the functionality necessary for developing and executing mashups. As the figure below shows, the core of the Application Mashup has three main components: the **Composition Editor**, the **Mashup Execution Engine**, and the **Local Catalogue**.
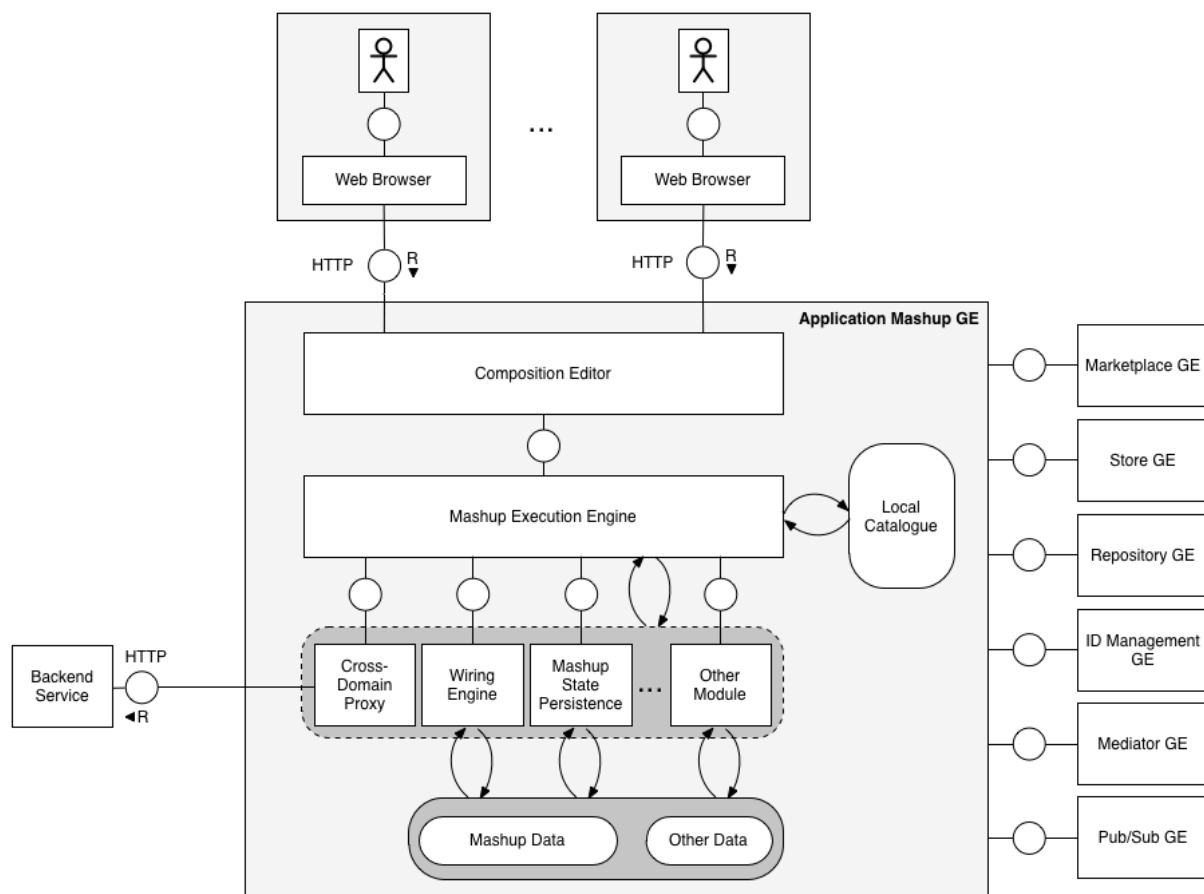
The Application Mashup GE Architecture

The **Composition Editor** component is the web-based tool with which end users interact via a web browser in order to create their own mashup applications. This component must, at

least, offer end users a kind of *workspace* where they can spatially place or arrange widgets, plus an extra view of the wiring mechanism to set the interconnection between the arranged widgets. Because this component is a visual editor, this open specification does not set the visual appearance that this tool must have. It is up to the GE's implementation developer to create their own look & feel for this tool.

The **Mashup Execution Engine** component is probably the most important part to be developed. It coordinates widget execution and controls the data flow between widgets. It can access the Local Catalogue to deploy and execute stored widgets. The functionality of this component can be connected to and extended by a number of plug-in modules as shown in the Application Mashup GE Architecture figure. Module functionality is exposed to the widgets by means of the WidgetAPI (see Open API Specifications). Some of these plug-in modules must always be there:

- The **Cross-Domain Proxy** module: this component will provide widgets with a proxy to overcome the Javascript cross-domain problem.

- The **Wiring Engine** Module: this component manages the wiring mechanism.

- The **Mashup State Persistence** Module: this module is in charge of guaranteeing the persistence of the MACs under execution. This includes not only to storing the widgets and operators involved in the mashup and their state, but also their position in the editor view, their interconnections (wiring and piping) and so on.

It must be possible to enhance widget functionality by adding new modules to the Mashup Execution Engine. For example, a Publish/Subscribe module could be added to provide widgets with the ability to receive and publish data in a pub/sub fashion using the new added module.

All plug-in modules must be able to make use of internal storage (i.e. a database) for their specific persistence needs.

The third main component is the **Local Catalogue**. This component is where MACs, either purchased from the FI-WARE Store GE or installed (uploaded) by the end-user, are stored, configured, and set ready for deployment and execution. This component should be a kind of showcase for the logged user of the Application Mashup GE.

The Application Mashup GE is closely related to other FI-WARE Generic Enablers, especially related to the business infrastructure and the provision of data sources. The figure below depicts some of these relationships:
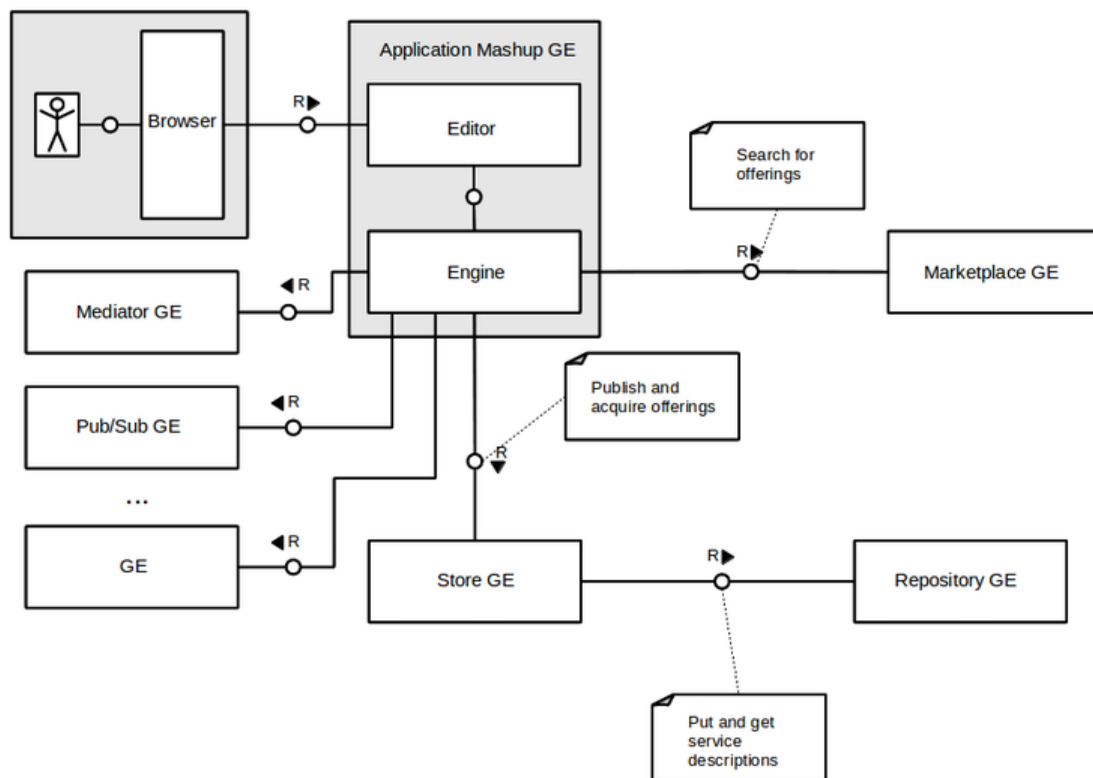
How the Application Mashup GE relates to the other Generic Enablers

The following sections describe the languages needed to support widgets, operators and mashups, including the USDL extensions for all business-related GEs to process any MAC as an offering, and the specific file formats used to store and distribute widgets. By implementing these artefacts, a concrete implementation of the Application Mashup GE will support an internal representation of MACs (widgets and mashups), which is necessary to interact with them.

3.5.4   Mashup and Widgets Definition Languages (MDL and WDL)

The Application Mashup GE should be implemented as a mashup platform that empowers users to create their own application mashups. Application mashups are made of a set of widgets interconnected with each other (that is, wired). To fully support widgets (and mashups) instantiation, the implementation of the Application Mashup GE must support a platform-independent widget and mashup languages.

The *Mashup Definition Language* (MDL) and the *Widget Definition Language* (WDL) are the chosen languages. They define all the inner information (metadata) regarding both a mashup and their widgets and their relationships/interconnections. This includes typical metadata, such as widget's name, vendor, version, last updated date, but also graphical information such as the location of the widgets on the editor canvas, widget width and height, etc.

The following sections shows the concept of "template". The template represents the definition of both languages and must be supported by the Application Mashup GE.

3.5.4.1 *Mashup and Widget Template*

To internally represent and deal with both mashups and widgets, the Application Mashup GE must support their "templates". A template is an XML file that contains all mashup and widget-related contextual information needed by the Application Mashup plus a set of preferences, state properties, and wiring, context and rendering information. Both MDL and WDL templates have their associated XML Schema. The latest version of the XML Schemas described in the following sections are available at:

- MDL XML Template Schema
  (https://github.com/Wirecloud/wirecloud/blob/develop/docs/source/mashups/mashup_template_xml.xsd)

- WDL XML Template Schema
  (https://github.com/Wirecloud/wirecloud/blob/develop/docs/source/widgets/template/xml_template.xsd)

WDL Template description as a XML Schema Definition

First we describe a description of the widget template XML Schema, i.e. what WDL looks like. It uses the *http://morfeo-project.org/2007/Template* namespace for the root element, called **Template**. The figure below shows the `Template` element and the sequence of subelements that it contains.
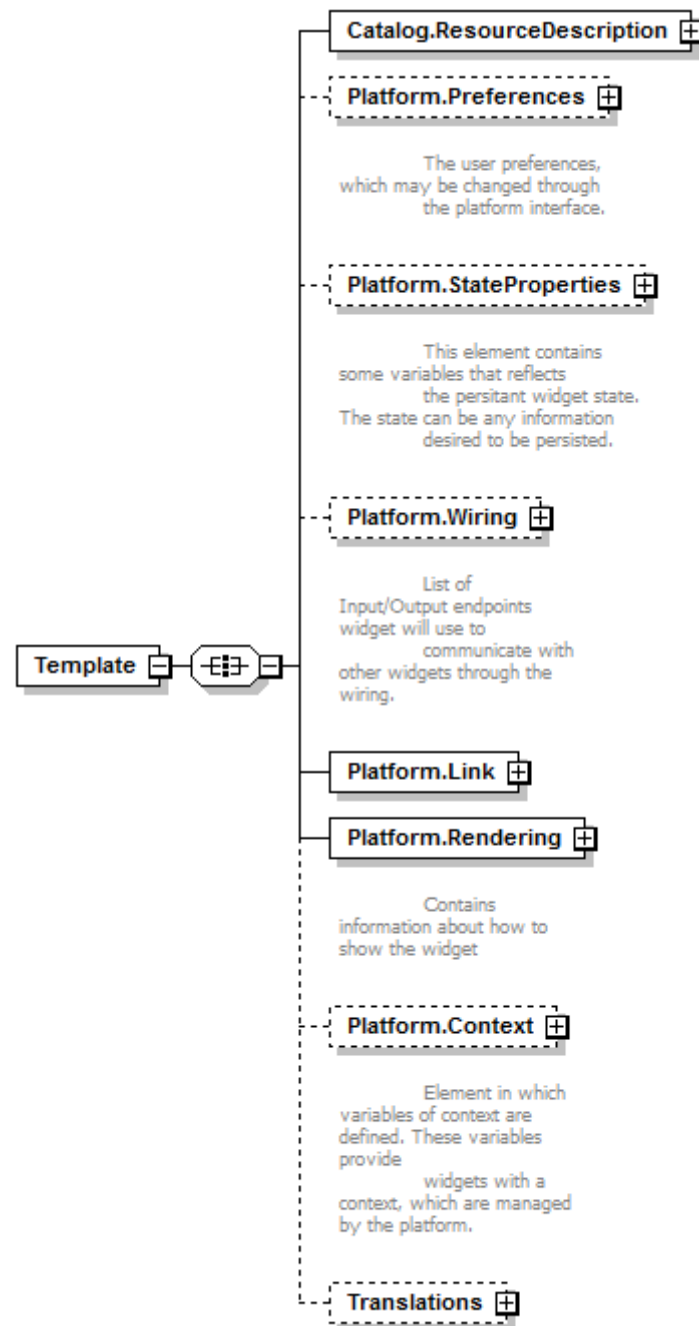
The "Template" root element

The `Template` element defines all the widget-related contextual information in an XML element called **Catalog.ResourceDescription**. This is a mandatory element of the XML document. The figure below depicts what it looks like:

Vendor

The distributor of the widget.

Name

Name of the widget.

Version

Current version of the widget. It must define starting sequences of numbers separated by dots. Moreover, zeros can only be used alone (e.g. 0.1 is valid but 03.2 is not).

DisplayName

Name used in the user interface for the widget. This field can be translated, therefore this field is not used to uniquely identify the widget.

Author

Developer of the widget.

Mail

E-mail address to get in touch with the developer(s).

Description

A brief textual description of the widget.

ImageURI

Absolute or template-relative URL of the widget image for the catalog.

iPhoneImageURI

Image to be used in iPhone and other smartphones.

WikiURI

Absolute or template-relative URL of the widget documentation.

Requirements — **Feature** — attributes — name

0..∞

Catalog.ResourceDescription

The "Catalog.ResourceDescription" element

This core element it is made up of the following attributes:

- **Vendor:** Company that distributes the widget. It cannot contain the character "/".

- **Name:** Name of the widget. It cannot contain the character "/".

- **Version:** Current widget version number. It must define starting sequences of numbers separated by dots. Zeros can only be used alone (e.g. 0.1 is valid but 03.2 is not).

- **DisplayName:** Name shown in the user interface of the widget. This field can be translated; therefore this field does not identify the widget.

- **Author:** Widget developers.

- **Mail:** Developer's e-mail address.

- **Description**: Full widget description to be shown in the catalogue.

- **ImageURI:** Absolute or template-relative URL of the image shown in the catalogue.

- **iPhoneImageURI:** Image to be used in iPhones and other smartphones.

- **WikiURI:** Absolute or template-relative URL of the widget documentation.

The *vendor*, *name* and *version* fields are the the widget's ID. Therefore, no such identifier can appear more than once in any collection of the Application Mashup GE stored resources (this includes widgets, mashups, operators, etc.).

XML elements that manage the Platform-Widget interaction

To guarantee the platform-widget interaction, templates also define a set of variables that widgets use to get connected to the environment and set different platform options. Likewise, it also defines some other interface elements, such as the initial widget size. They are all managed by the platform, which will ensure their persistence.

Let us go through all these elements:

The Platform.Preferences element

The first platform-related element is the `Platform.Preferences` one:
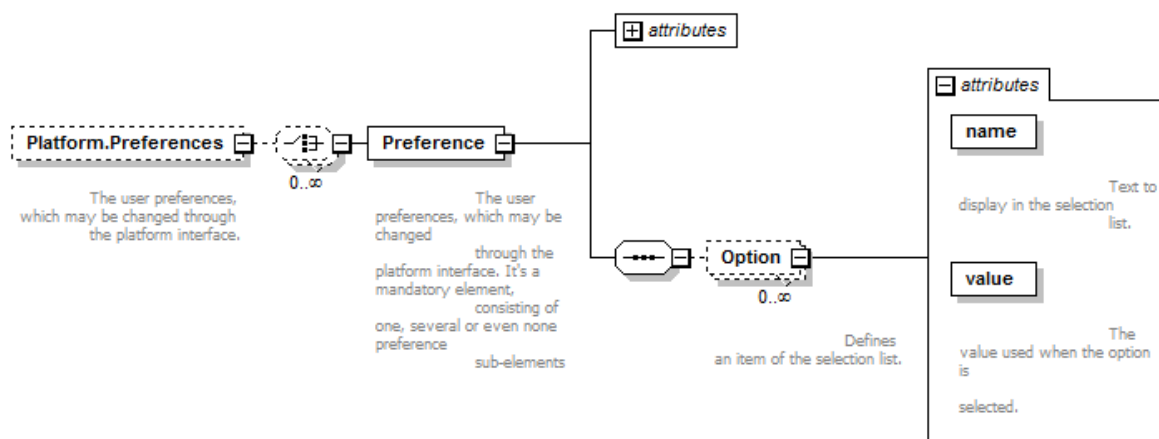
The "Platform.Preferences" element

It defines user preferences, which may be changed through the platform interface. It is a mandatory element that is made up of one, many or none `Preference` sub-elements. This defines the actual user preference. It requires the following attributes:

- **name:** name of the preference to be referenced in the source code.
- **type:** preference data type: text (string), number, Boolean, password and list.
- **description:** text that describes the preference.
- **label:** text that the preference will show in the user interface.
- **default:** preference default value.

If the *type* attribute is set to "list", the different choices are defined by means of the **Option** element. It has the following attributes:

- **name:** text to be displayed in the selection list.
- **value:** value to be used when the option is selected.

The Platform.StateProperties element

The next XML element is the **Platform.StateProperties** element. Its main purpose is to define a set of properties to store the state of the widget while it is executing, in order to have it available for future executions. Its structure is shown in the figure below:
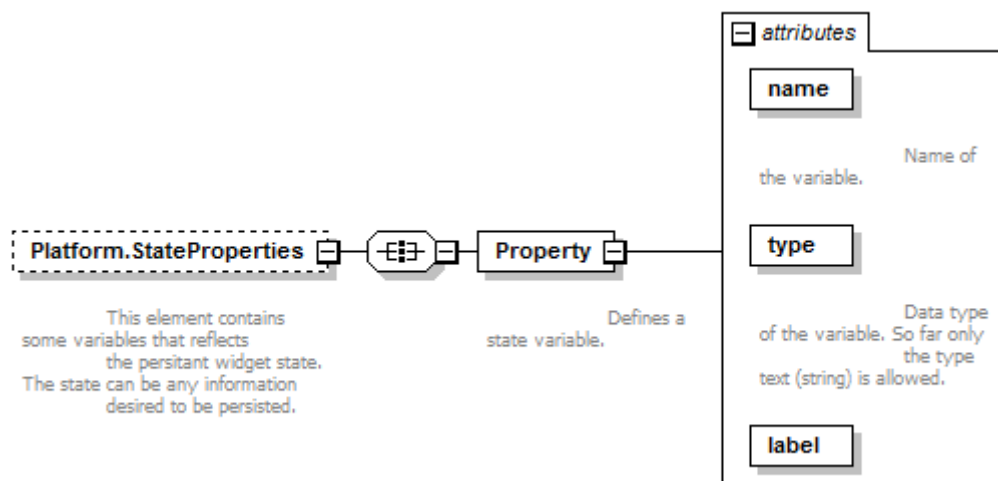
The Platform.StateProperties element

This element is required. It is made up of a list of **Property** elements and requires the following attributes:

- **name:** property name.
- **type:** property data type: only "text" (string) datatype does make sense in here.
- **label:** text to be displayed in the user interface.

The Platform.Wiring element

This is probably one of the most important widget template elements. It defines both the widget inputs and outputs needed to intercommunicate with other widgets. The Application

Mashup GE implementation must take this information into account to manage and control the wiring mechanism and its internal data flow.

The figure below depicts the **Platform.Wiring** element:
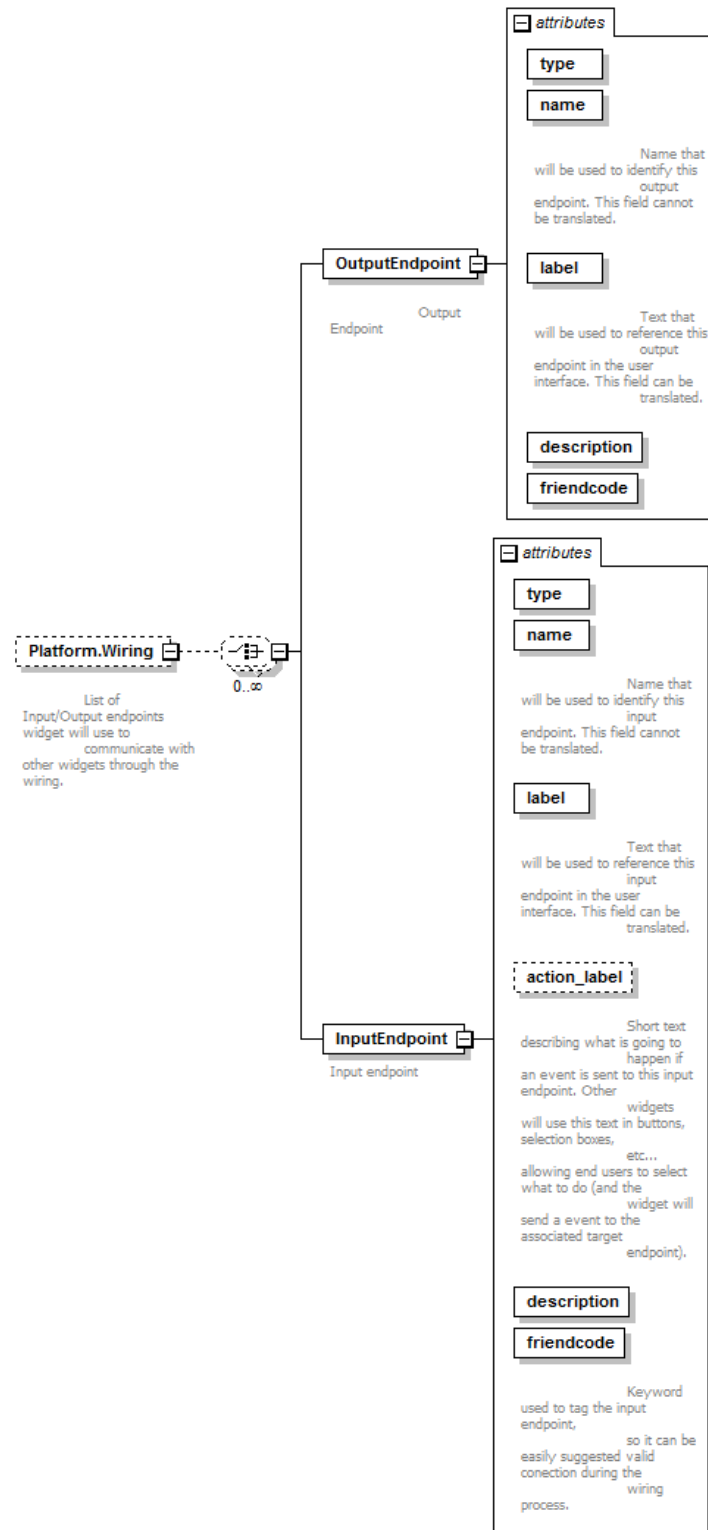
The Platform.Wiring element.

This element may contain any number of **InputEndpoint** and **OutputEndpoint** elements:

Widgets may send data (events) through an output endpoint. To do so, they must declare the endpoint using the `OutputEndpoint` element. These elements have the following attributes:

- **name:** output endpoint name.
- **type:** output endpoint data type: only "text" (string) datatype does make sense in here.
- **label:** text to be displayed in the user interface.
- **description:** text that describes the output.
- **friendcode:** keyword used as an output endpoint tag: it will help the platform to make suggestions in the wiring process.

On the other hand, widgets can receive asynchronous data through the input endpoints. These endpoints are meant to be used by the widget for receiving data (events) coming from other widgets. The required `InputEndpoint` elements requires the following attributes:

- **name:** input endpoint name.
- **type:** input endpoint data type: only "text" (string) datatype does make sense in here.
- **label:** text to be displayed in the user interface.
- **actionlabel:** short text that describes what is going to happen if an event is sent to this input endpoint. Widgets could use this text in buttons, selection boxes, etc... allowing end users to select what to do (and the widget will send an event to the associated target endpoint)
- **description:** text that describes the input.
- **friendcode:** keyword used as an input endpoint tag: it will help the platform to make suggestions in the wiring process.

The Platform.Context element

Widgets can have associated context information (i.e. usernames, current height and width...). The `Platform.Context` element defines which data the widget will be able to access and be notified if changed. The structure of this element is depicted in the figure below:
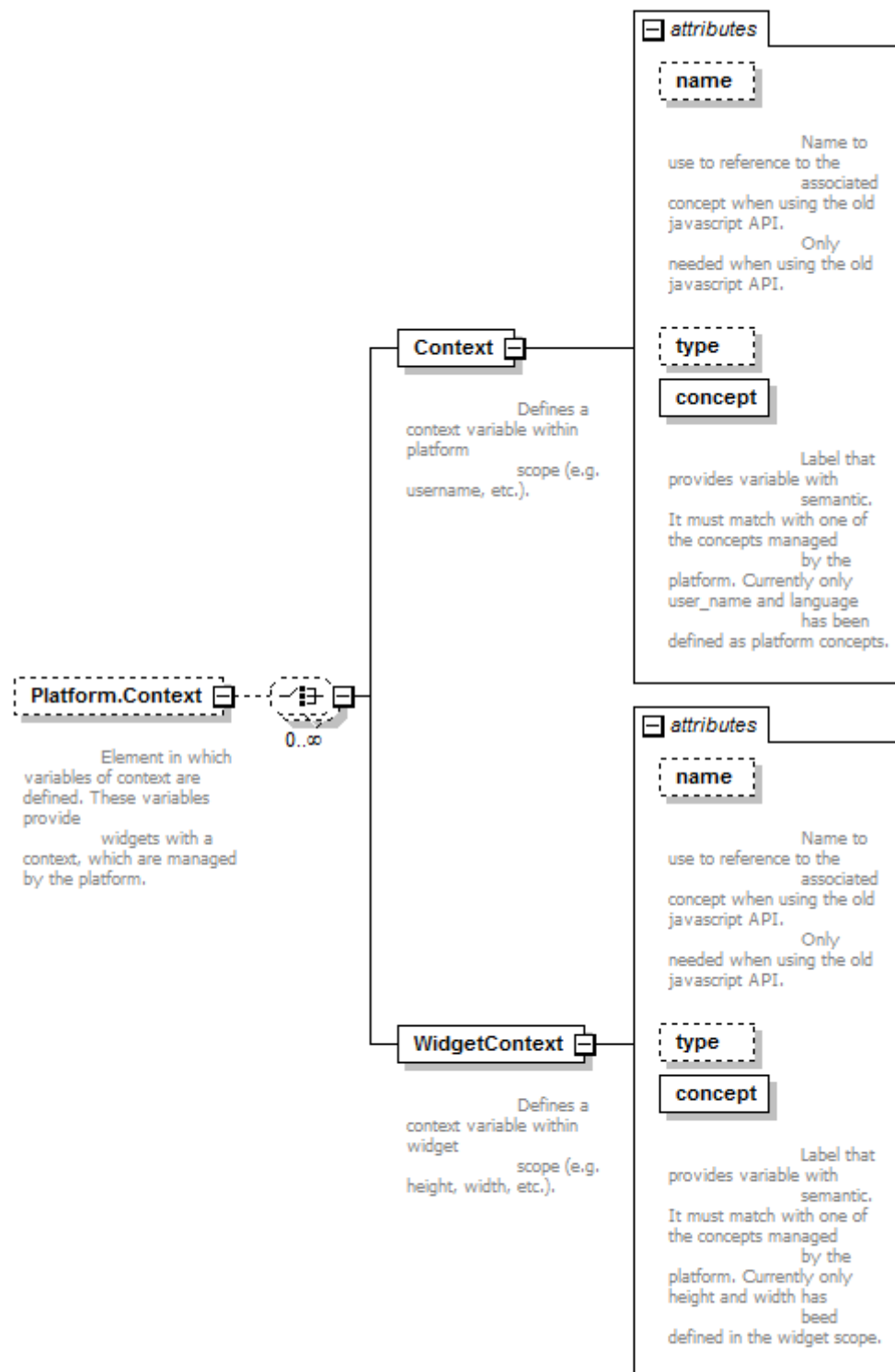
The Platform.Context element

This mandatory element can be followed by any number of these two child elements: **Context** and **WidgetContext**. The **Context** defines a platform-related context variable (i.e. username), whereas the **WidgetContext** defines a widget-related context variable (i.e. height). Both of them must have the following attributes:

- **name:** variable name.

- **type:** data type of the variable. Only "text" (string) datatype does make sense in here.

- **concept:** text that gives the variable meaning by annotating it with semantics. It must match with one of the concepts managed by the platform. Currently only *user_name* and *language* have been defined as platform concepts, and *height* and *width* in the widget scope.

The Platform.Link element

The actual source code of the widget must be linked to this template. To do this, the `Platform.Link` element is needed.
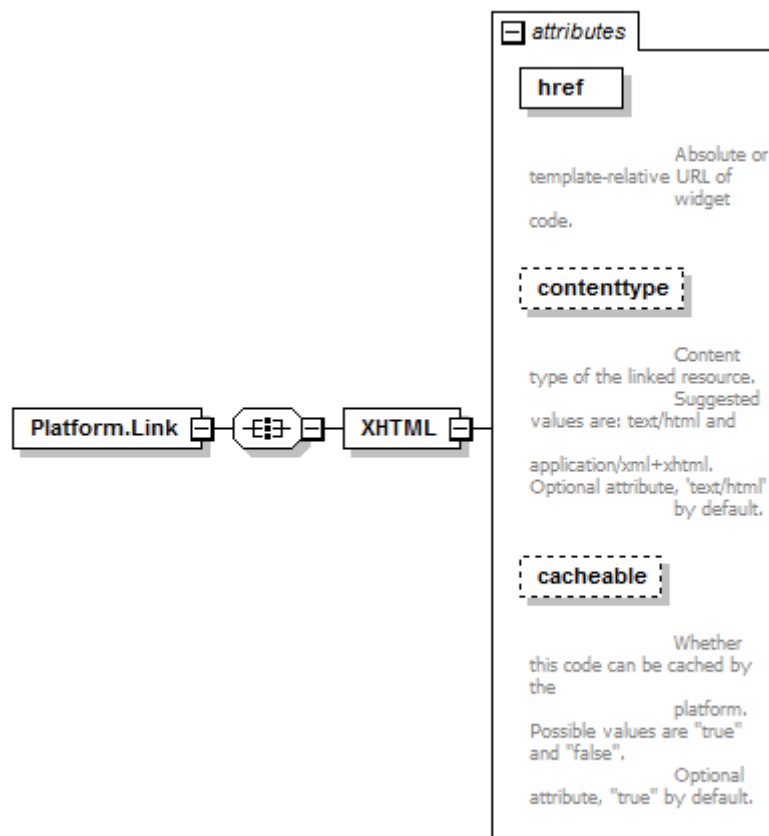
The Platform.Link element binds the template with the actual widget source code

It is made up of the **XHTML** element, which has the following attributes:

- **href:** absolute or template-relative URL of widget code.

- **contenttype:** linked resource content type: suggested values are: text/html and application/xml+xhtml. This is an optional attribute, with 'text/html' by default.

- **cacheable:** sets if the linked code can be cached by the platform: possible values are "true" and "false". This is an optional attribute, "true" by default.

The Platform.Rendering element

The last template XML element is `Platform.Rendering`. It specifies the default width and height of the widget once it is deployed in the user workspace.
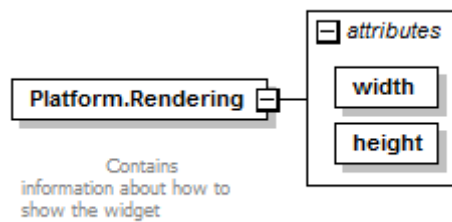
The Platform.Rendering element

Width and **height** are its only subelements. They represent the initial width and height of the widget.

MDL Template description as a XML Schema Definition

The MDL XML template schema is quite similar to the WDL template and is used to describe a mashup composed of widgets. The figure below shows the Mashup `Template` element and its sequence of subelements.
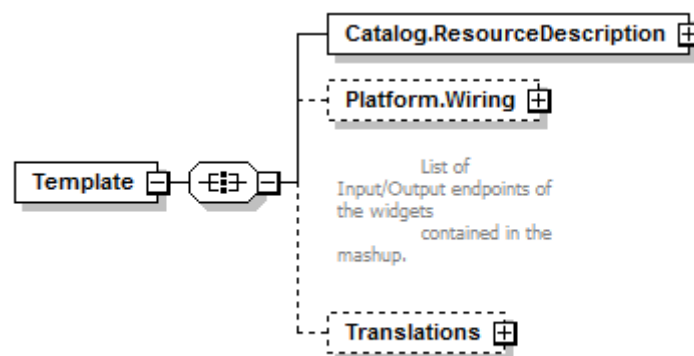
The Mashup "Template" root element

The **Catalog.ResourceDescription** element has the same fields as in the widget template with an extra field called **IncludedResources** that is used to describe widgets within the mashup. The figure below depicts what it looks like:
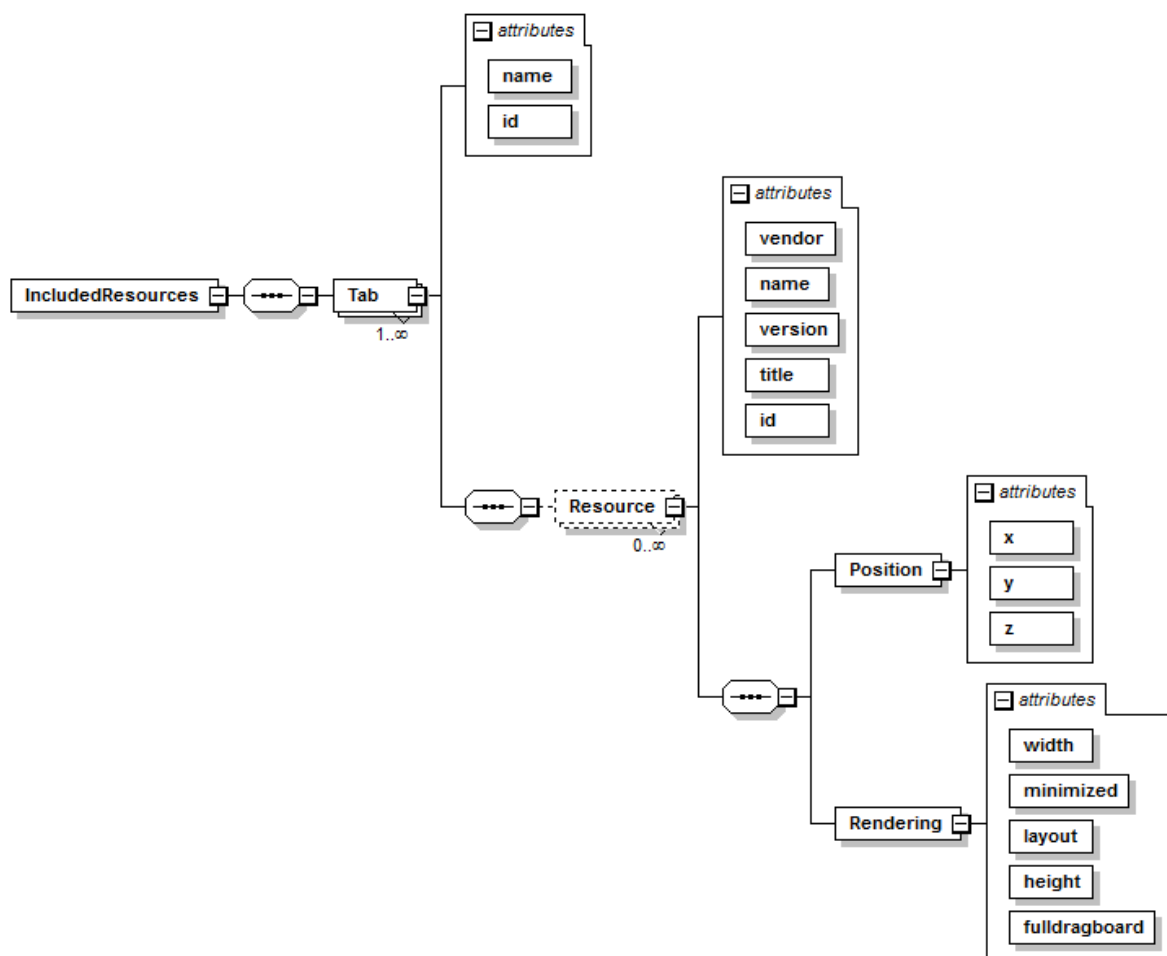
The "IncludedResources" element

This element contains at least one **Tab** element that represents tabs in Application Mashup GE dashboard. It has the following attributes.

- **name:** the name of the tab

- **id:** the identification of the tab; this id is internal to the template.

The **Tab** element may contain any number of **Resource** elements which represent widget instances used in the mashup. It has the following attributes.

- **vendor** the widget distributor; it cannot contain the character "/".

- **name** name of the widget; it cannot contain the character "/".

- **version** current version of the widget; it must define starting sequences of numbers separated by dots, where zeros can only be used alone (e.g. 0.1 is valid but 03.2 is not).

- **title** name to be displayed in the widget dashboard.

- **id** the widget identification; this id is internal to the mashup template.

The **Resource** element is made up of a **Position** element and a **Rendering** element. The **Position** element describes the widget position into the dashboard. It has the following attributes.

- **X:** the widget's x coordinate.

- **Y:** the widget's Y coordinate.

- **Z:** the widget's Z coordinate.

The **Rendering** element describes some characteristics of the widget representation. It has the following attributes.

- **width:** widget width in the dashboard.

- **minimized:** Boolean attribute that defines whether the widget is minimized in the dashboard

- **layout:** widget layout in the dashboard

- **height:** widget height in the dashboard

- **fulldragboard:** Boolean attribute that describes whether the widget is using all the dashboard.

The Platform.Wiring element

This element describes how widgets in the mashup are connected using their output and input endpoints.
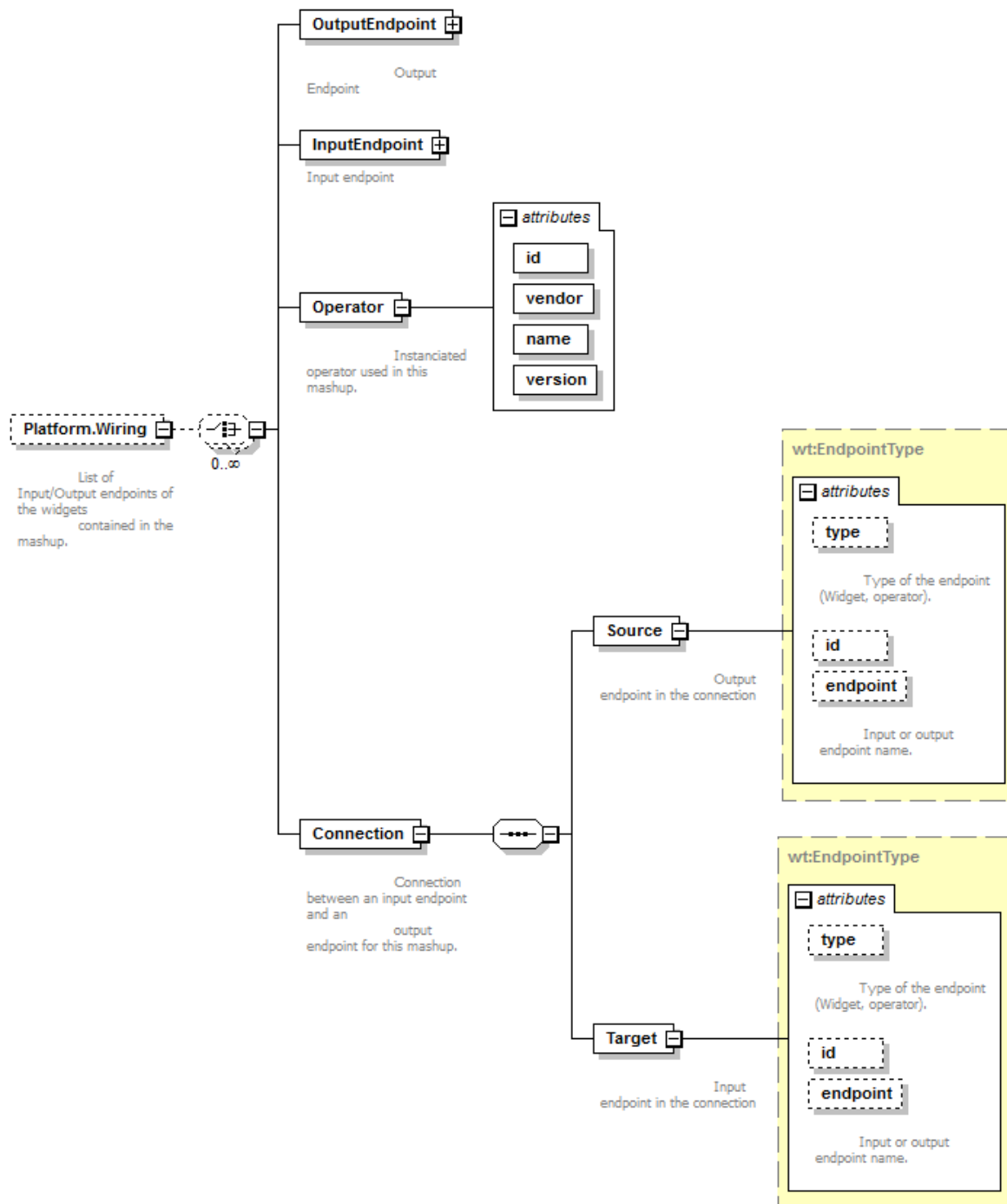
The "Platform.Wiring" element

The **InputEndpoint** and **OutputEndpoint** elements define the same information as in the WDL template. The **Platform.Wiring** element contains all the input and output endpoints of all the widgets and operators in the mashup. The **Platform.Wiring** element may contain any number of **Operator** elements. An **Operator** element defines an operator that is used in the wiring. It has the following attributes.

- **id:** identification of the operator; this id is internal to the mashup template.

- **vendor:** the distributor of the operator; it cannot contain the character "/".

- **name:** operator name; it cannot contain the character "/".

- **version:** current operator version; it must define starting sequences of numbers separated by dots where zeros can only be used alone (e.g. 0.1 is valid but 03.2 is not).

The **Platform.Wiring** element may contain any number of **Connection** elements. These elements describe which output endpoints are connected with which input endpoints. The **Connection** elements are composed of a **Source** element and a **Target** element. The **Source** element defines the output endpoint of the connection. It has the following attributes.

- **type:** type of the element that has the output endpoint; this attribute could has the values "widget" or "operator".

- **id:** id of the element that has the output endpoint; this id is the same as the id defined in the **Resource** element if the element is a widget, whereas this id is the same as the id defined in the **Operator** element if the element is an operator.

- **endpoint:** the name of the output endpoint. This name is the same as the defined in the **OutputEnpoint** element.

The **Target** element defines the input endpoint of the connection. It has the following attributes.

- **type:** type of element that has the input endpoint; the possible values of this attribute are "widget" or "operator".

- **id:** id of the element that has the input endpoint; this id is the same as the id defined in the **Resource** element if the element is a widget, whereas this id is the same as the id defined in the **Operator** element if the element is an operator.

- **endpoint** the name of the input endpoint; this name is the same as the defined in the **InputEnpoint** element.

3.5.4.2   *Mashup and Widgets as a USDL offering: USDL extension*

In order for both mashups and widgets to be offered as a USDL offering in the FI-WARE Store GE so that both widgets and mashups can be part of the data managed by the FI-WARE's Marketplace, Store and Repository Generic Enablers, the Application Mashup GE must make use of the following RDF(S) vocabularies, built upon Linked Data principles as Linked USDL extensions.

The first specification, WDL-RDF:

(https://github.com/Wirecloud/wirecloud/blob/develop/docs/source/widgets/template/rdf_template.rdf),

deals with the definition of the information that the Application Mashup GE must use to instantiate a widget, including its user preferences, state properties, wiring information, and so on.

The second specification, MDL-RDF:

(https://github.com/Wirecloud/wirecloud/blob/develop/docs/source/mashups/mashup_template_rdf.rdf),

defines the mashup-related information needed to create an instance of the user workspace, including platform-specific information such as widget instances, wiring and piping between widgets, etc.

The following sections show both vocabularies to be used within the USDL offerings.

WDL-RDF

The diagram below shows the WDL-RDF vocabulary.

The WDL-RDF extension for USDL

The Application Mashup GE must support this vocabulary to provide a way to represent WDL information as part of a USDL offering.

Classes

Class: `wire:Widget`

This class represents a widget. This is the main class of the vocabulary.

URI: http://wirecloud.conwet.fi.upm.es/ns/widget#Widget

Properties include:

```
dcterms:title, dcterms:description, dcterms:creator, usdl:hasProvider,
usdl:utilizedResource, foaf:page, wire:hasPlatformPreference,
wire:hasContext, wire:hasPlatformWiring, wire:hasPlatformRendering,
wire:hasPlatformStateProperty, usdl:versionInfo, wire:hasImageUri,
wire.hasiPhoneImageUri, wire:displayName, vcard:addr
```

Subclassof: `usdl-core:Service`

Class: `wire:Operator`

This class represents an operator.

URI: http://wirecloud.conwet.fi.upm.es/ns/widget#Operator

Properties include:

```
dcterms:title, dcterms:description, dcterms:creator, usdl:hasProvider,
usdl:utilizedResource, foaf: page, wire:hasPlatformPreference,
wire:hasContext, wire:hasPlatformWiring, wire:hasPlatformRendering,
wire:hasPlatformStateProperty, usdl:versionInfo, wire:hasImageUri,
wire.hasiPhoneImageUri, wire:displayName, vcard:addr
```

Subclassof: `usdl-core:Service`

Class: `wire:PlatformPreference`

This class represents a user preference in the Application Mashup GE, that is, data users can see and configure. The Enabler must make this value persistent and provide users with tools to edit and validate this data.

URI: http://wirecloud.conwet.fi.upm.es/ns/widget#PlatformPreference

Properties include:

```
wire:hasOption, dcterms:title, dcterms:description, rdfs:label, wire:type,
wire:default, wire:secure
```

Used with: `wire:hasPlatformPreference`

Class: `wire:Context`

This class represents the context of the widget.

URI: http://wirecloud.conwet.fi.upm.es/ns/widget#Context

Properties include:

```
wire:hasPlatformContext, wire:haswidgetContext
```

Used with: `wire:hasContext`

Class: `wire:PlatformWiring`

This class represents the wiring status of a widget.

URI: http://wirecloud.conwet.fi.upm.es/ns/widget#PlatformWiring

Properties include:

```
wire:hasOutputEndpoint, wire:hasInputEnpoint
```

Used with: `wire:hasPlatformWiring`

Class: `wire:PlatformRendering`

This class represents the widget size when it is instantiated.

URI: http://wirecloud.conwet.fi.upm.es/ns/widget#PlatformRendering

Properties include:

```
wire:renderingWidth, wire.renderingHeight
```

Used with: `wire:hasPlatformRendering`

Class: wire:PlatformStateProperty

This class represents a widget state variable that the platform needs to know in order to make it persistent.

URI: http://wirecloud.conwet.fi.upm.es/ns/widget#PlatformStateProperty

Properties include:

`dcterms:title, dcterms:description, wire:type, rdfs:label, wire:default, wire:secure`

Used with: `wire:hasPlatformStateProperty`

Class: `wire:Option`

This class represents an option that a user preference could have.

URI: http://wirecloud.conwet.fi.upm.es/ns/widget#Option

Properties include:

`dcterms:title, wire:value`

Used with: `wire:hasOption`

Class: `wire:PlatformContext`

This class represents a platform context variable.

URI: http://wirecloud.conwet.fi.upm.es/ns/widget#PlatformContext

Properties include:

`dcterms:title, wire:type, wire:concept`

Used with: `wire:hasPlatformContext`

Class: `wire:widgetContext`

This class represents a widget context variable.

URI: http://wirecloud.conwet.fi.upm.es/ns/widget#widgetContext

Properties include:

`dcterms:title, wire:type, wire:concept`

Used with: `wire:haswidgetContext`

Class: `wire:OutputEndpoint`

This class represents an output endpoint.

URI: http://wirecloud.conwet.fi.upm.es/ns/widget#OutputEndpoint

Properties include:

`dcterms:title, dcterms:description, rdfs:label, wire:type, wire:outputFriendcode`

Used with: `wire:hasOutputEndpoint`

Class: `wire:InputEndpoint`

This class represents an input endpoint.

URI: http://wirecloud.conwet.fi.upm.es/ns/widget#InputEndpoint

Properties include:

`dcterms:title, dcterms:description, rdfs:label, wire:type,`
`wire:inputFriendcode, wire:actionLabel`

Used with: `wire:hasInputEndpoint`

Properties

Property: `wire:hasPlatformPreference`

This property states a user widget preference.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#hasPlatformPreference

Domain: `wire:Widget`

Range: `wire:PlatformPreference`

Property: `wire:hasContext`

This property states the widget context.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#hasContext

Domain: `wire:Widget`

Range: `wire:Context`

Property: `wire:hasPlatformWiring`

This property states the widget wiring status.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#hasPlatformWiring

Domain: `wire:Widget`

Range: `wire:PlatformWiring`

Property: `wire:hasPlatformRendering`

This property states how the widget must be rendered.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#hasPlatformRendering

Domain: `wire:Widget`

Range: `wire:PlatformRendering`

Property: `wire:hasPlatformStateProperty`

This property states a widget state variable.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#hasPlatformStateProperty

Domain: `wire:Widget`

Range: `wire:PlatformStateProperty`

Property: `wire:hasOption`

This property states a user preference option.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#hasOption

Domain: `wire:PlatformPreference`

Range: `wire:Option`

Property: `wire:hasPlatformContext`

This property states a platform-context variable of the context.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#hasPlatformContext

Domain: `wire:Context`

Range: `wire:PlatformContext`

Property: `wire:hasWidgetContext`

This property states a widget-context variable of the context.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#hasWidgetContext

Domain: `wire:Context`

Range: `wire:WidgetContext`

Property: `wire:hasOutputEndpoint`

This property states a widget wiring output endpoint.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#hasOutputEndpoint

Domain: `wire:PlatformWiring`

Range: `wire:OutputEndpoint`

Property: `wire:hasInputEndpoint`

This property states a widget wiring input endpoint.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#hasInputEndpoint

Domain: `wire:PlatformWiring`

Range: `wire:InputEndpoint`

Property: `wire:platformContextConcept`

This property states the platform-context variable concept.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#platformContextConcept

Domain: `wire:PlatformContext`

Range: `rdfs:Literal`

Property: `wire:WidgetContextConcept`

This property states the widget-context variable concept.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#platformWidgetConcept

Domain: `wire:WidgetContext`

Range: `rdfs:Literal`

Property: `wire:outputFriendcode`

This property states an output's *friendcode.*

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#outputFriendcode

Domain: `wire:OutputEndpoint`

Range: `rdfs:Literal`

Property: `wire:inputFriendcode`

This property states an input's *friendcode.*

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#inputFriendcode

Domain: `wire:InputEndpoint`

Range: `rdfs:Literal`

Property: `wire:actionLabel`

This property states an input's action label.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#actionLabel

Domain: `wire:InputEndpoint`

Range: `rdfs:Literal`

Property: `wire:hasImageUri`

This property states the URI of the image associated with the widget.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#hasImageUri

Domain: `wire:Widget`

Range: `foaf:Image`

Property: `wire:hasiPhoneImageUri`

This property states the URI of the image associated with the Widget if the platform is running on an iPhone.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#hasiPhoneImageUri

Domain: `wire:Widget`

Range: `foaf:Image`

Property: `wire:displayName`

This property states the widget name to be displayed.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#displayName

Domain: `wire:Widget`

Range: `rdfs:Literal`

Property: `wire:value`

This property states the widget configuration element value.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#value

Range: `rdfs:Literal`

Property: `wire:type`

This property states the widget configuration element type.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#type

Range: `rdfs:Literal`

Property: `wire:default`

This property states the widget configuration element default value.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#default

Range: `rdfs:Literal`

Property: `wire:secure`

This property states whether or not a widget configuration element is secure.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#value

Range: `rdfs:Literal`

Property: `wire:index`

This property states the logical order of elements of the same type.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#value

Range: `rdfs:Literal`

Property: `wire:codeContentType`

This property states the widget code MIME type. The widget code URI is represented using `usdl-core:Resource`

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#codeContentType

Domain: `usdl-core:Resource`

Range: `rdfs:Literal`

Property: `wire:codeCacheable`

This property states whether or not the widget code is cacheable.

URI: http://wirecloud.conwet.fi.upm.es/ns/Widget#codeCacheable

Domain: `usdl-core:Resource`

Range: `rdfs:Literal`

MDL-RDF

The diagram below shows the MDL-RDF vocabulary.

The MDL-RDF extension for USDL

Like WDL-RDF, this vocabulary must be supported by the Application Mashup GE to provide a way to represent MDL information as part of a USDL offering.

Classes

Class: `wire-m:Mashup`

This class represents a mashup. It will be implemented as a workspace.

URI :http://wirecloud.conwet.fi.upm.es/ns/mashup#Mashup

Properties include:

`wire-m:hasMashupPreference, wire-m:hasMashupParam, wire-m:hasTab, wire-m:hasMashupWiring, wire:hasImageUri, wire:hasiPhoneImageUri, wire:version`

subClassOf: `usdl:CompositeService`

Class: `wire-m:Tab`

This class represents a workspace tab.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#Tab

Properties include:

`wire-m:hasiWidget, wire-m:hasTabPreference, dcterms:title`

Used with:

`wire-m:hasTab`

Class: `wire-m:iWidget`

This class represents a widget instance.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#iWidget

Properties include:

`wire-m:hasPosition, wire-m:hasiWidgetRendering, wire-m:hasiWidgetPreference, wire-m:hasiWidgetProperty`

Used with:

`wire-m:hasiWidget`

subClassOf: `wire:Widget`

Class: `wire-m:MashupPreference`

This class represents a mashup preference.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#MashupPreference

Properties include:

`dcterms:title, wire:value`

Used with:

`wire-m:hasMashupPreference`

Class: `wire-m:MashupParam`

This class represents a mashup parameter.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#MashupParam

Properties include:

`dcterms:title, wire:value`

Used with:

`wire-m:hasMashupParam`

Class: `wire-m:Position`

This class represents the position of a widget instance in the tab.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#Position

Properties include:

`wire-m:x, wire-m:y, wire-m:z`

Used with:

`wire-m:hasPosition`

Class: `wire-m:iWidgetPreference`

This class represents a widget instance preference.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#iWidgetPreference

Properties include:

`dcterms:title, wire:value, wire-m:readonly, wire-m:hidden`

Used with:

`wire-m:hasiWidgetPreference`

Class: `wire-m:iWidgetRendering`

This class represents a widget instance rendering.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#iWidgetRendering

Properties include:

`wire-m:fullDragboard, wire-m:layout, wire-m:minimized,`
`wire:renderingHeight, wire:renderingWidth`

Used with:

`wire-m:hasiWidgetRendering`

Class: `wire-m:iWidgetProperty`

This class represents a widget instance property.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#iWidgetProperty

Properties include:

`wire-m:readonly, wire:value`

Used with:

`wire-m:hasiWidgetProperty`

Class: `wire-m:TabPreference`

This class represents a tab preference.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#TabPreference

Properties include:

`dcterms:title, wire:value`

Used with:

`wire-m:hasTabPreference`

Class: `wire-m:Connection`

This class represents a wiring connection between two widget instances or operator instances.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#Connection

Properties include:

`wire-m:hasSource, wire-m:hasTarget, dcterms:title, wire-m:readonly`

Used with:

`wire-m:hasConnection`

Class: `wire-m:Source`

This class represents a widget instance or operator instance that is the source of a connection.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#Source

Properties include:

`wire-m:sourceId, wire-m:endpoint, wire:type`

Used with:

`wire-m:hasSource`

Class: `wire-m:Target`

This class represents a widget instance or operator instance that is the target of a connection.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#Target

Properties include:

`wire-m:targetId, wire-m:endpoint, wire:type`

Used with:

`wire-m:hasTarget`

Class: `wire-m:iOperator`

This class represents an operator instance.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#iOperator

Properties include:

`wire-m:iOperatorId, dcterms:title`

Used with:

`wire-m:hasiOperator`

Properties

Property: `wire-m:hasMashupPreference`

This property states a mashup preference.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#hasMashupPreference

Domain: `wire-m:Mashup`

Range: `wire-m:MashupPreference`

Property: `wire-m:hasMashupParam`

This property states a mashup parameter.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#hasMashupParam

Domain: `wire-m:Mashup`

Range: `wire-m:MashupParam`

Property: `wire-m:hasTab`

This property states that a given tab is part of a workspace.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#hasTab

Domain: `wire-m:Mashup`

Range: `wire-m:Tab`

Property: `wire-m:hasiWidget`

This property states that a given widget instance is instantiated in a tab.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#hasiWidget

Domain: `wire-m:Tab`

Range: `wire-m:iWidget`

Property: `wire-m:hasTabPreference`

This property states a tab preference.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#hasTabPreference

Domain: `wire-m:Tab`

Range: `wire-m:TabPreference`

Property: `wire-m:hasPosition`

This property states the position of an widget instance in a tab.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#hasPosition

Domain: `wire-m:iWidget`

Range: `wire-m:Position`

Property: `wire-m:hasiWidgetPreference`

This property states a widget instance preference.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#hasiWidgetPreference

Domain: `wire-m:iWidget`

Range: `wire-m:iWidgetPreference`

Property: `wire-m:hasiWidgetProperty`

This property states a widget instance property.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#hasiWidgetProperty

Domain: `wire-m:iWidget`

Range: `wire-m:iWidgetProperty`

Property: `wire-m:hasiWidgetRendering`

This property states the rendering of a widget instance.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#hasiWidgetRendering

Domain: `wire-m:iWidget`

Range: `wire-m:iWidgetRendering`

Property: `wire-m:hasConnection`

This property states a wiring connection.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#hasConnection

Domain: `wire:PlatformWiring`

Range: `wire-m:Connection`

Property: `wire-m:hasSource`

This property states the source of a connection.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#hasSource

Domain: `wire-m:Connection`

Range: `wire-m:Source`

Property: `wire-m:hasTarget`

This property states the target of a connection.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#hasTarget

Domain: `wire-m:Connection`

Range: `wire-m:Target`

Property: `wire-m:targetId`

This property states the ID of a target.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#targetId

Domain: `wire-m:Target`

Range: `rdfs:Literal`

Property: `wire-m:sourceId`

This property states the ID of a source.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#sourceId

Domain: `wire-m:Source`

Range: `rdfs:Literal`

Property: `wire-m:endpoint`

This property states the ID of the widget instance or operator instance that is the source or target of a connection.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#endpoint

Range: `rdfs:Literal`

Property: `wire-m:hasiOperator`

This property states the wiring of an operator's instance.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#hasiOperator

Domain: `wire:PlatformWiring`

Range: `wire-m:iOperator`

Property: `wire-m:x`

This property states the x coordinate of a widget instance position.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#x

Domain: `wire-m:Position`

Range: `rdfs:Literal`

Property: `wire-m:y`

This property states the y coordinate of a widget instance position.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#y

Domain: `wire-m:Position`

Range: `rdfs:Literal`

Property: `wire-m:z`

This property states the z coordinate of a widget instance position.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#z

Domain: `wire-m:Position`

Range: `rdfs:Literal`

Property: `wire-m:fullDragboard`

This property states whether a widget instance occupies the whole space in the tab.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#fullDragboard

Domain: `wire-m:iWidgetRendering`

Range: `rdfs:Literal`

Property: `wire-m:layout`

This property states the layout of a widget instance.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#layout

Domain: `wire-m:iWidgetRendering`

Range: `rdfs:Literal`

Property: `wire-m:minimized`

This property states whether a widget instance is minimized in its tab.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#minimized

Domain: `wire-m:iWidgetRendering`

Range: `rdfs:Literal`

Property: `wire-m:hidden`

This property states whether a widget instance is hidden in its tab.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#hidden

Domain: `wire-m:iWidgetPreference`

Range: `rdfs:Literal`

Property: `wire-m:readonly`

This property states whether a mashup configuration element is read only.

URI: http://wirecloud.conwet.fi.upm.es/ns/mashup#readonly

Range: `rdfs:Literal`

3.5.4.3 *WGT zipped file format*

The Mashup Application GE relies on PKWare's Zip specification as the archive format for the self-packaged version of the Mashable Application Components. The packaging format acts as a container for files used by a MAC whereas the only initial requirement is to have a configuration document declaring metadata and configuration parameters for the MAC. This configuration file must use one of the metadata description languages supported by the Mashup Application GE (WDL and MDL in either of its flavours: XML and RDF). This configuration file must be present at the root of the Zip container and the name must be `config.xml` or `config.rdf`. Any relative path/URL included in the configuration document will use the root of the zip file as the base path/URL.

The Mashup Application GE should prohibit relative paths for accessing files outside the container. This is especially important as the Mashup Application GE may extract these files to the file system.

3.6 Main Interactions

This section describes in detail all the interactions that the Application Mashup GE must support both with users and with other FI-WARE GEs.

3.6.1 Life-cycle of a Mashable Application Component (mashup, widget and operators)

Note that Web mashups are aimed at leveraging the "long tail" of the Internet of Services by exploiting rapid development, the "Do-It-Yourself" (DIY) metaphor, and shareability. They typically serve a specific situational (i.e. immediate, short-lived, customized, specific) need, often with a high reuse potential. This need for sharing means that the Application Mashup GE should be fully compliant with the FI-WARE's Marketplace, Store and Repository Generic Enablers. The fact that a MAC can be offered in a Store before being used in the Application Mashup GE results in the definition of the following MAC life-cycle:
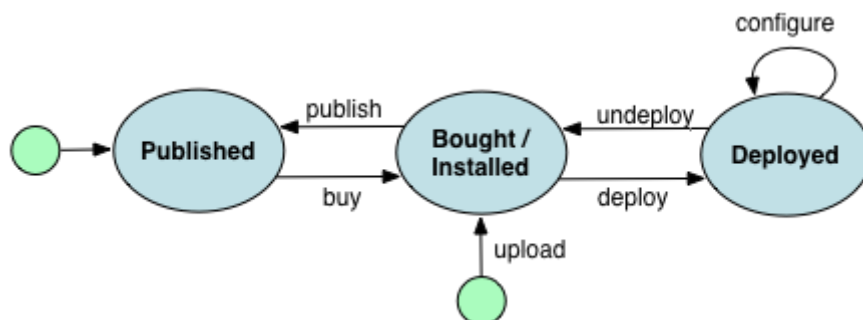
Lifecycle of a MAC (mashup, widget and operator)

Mashable Application Components (i.e, mashups, widgets and operators) must pass through the following states:

- **Published**
- **Bought/installed**
- **Deployed**

The init state for a MAC means that the MAC is neither published in the store, bought, nor installed in the user local repository. A MAC is **published** when it is made available to Store

customers. Users that are interested in using a **published** MAC, can buy the MAC, thus transferring it to a *bought* state. Once *bought*, the MAC is automatically **installed** in the local catalogue of the Application Mashup GE. An alternative is to *upload* a MAC that the users have developed and which tehy do not have to buy. This is why the state is named **bought/installed**. Once a users have *uploaded* the MAC to the local catalogue, they can proceed to *publish*the MAC. Once the MAC is installed, it can be **deployed** in the user workspace. *Bought* and *installed* MACs must be **deployed** in the user workspace before they can be *configured*.

3.6.2   Interaction diagrams

The Application Mashup Generic Enabler must be designed and developed to enable, at least, the interactions shown in this section. They cover the user-platform interactions needed to visually create a web application mashup, plus the main interactions between the platform and other generic enablers.

3.6.2.1   *User-Platform Interactions*

The interactions that the Application Mashup GE must support with regard to its users are as follows.

Upload a Mashable Application Component to the Local Catalogue

MAC developers must be able to upload their own developed resources to the local catalogue of the Application Mashup GE. End users must also be able to upload the MACs they already have stored in their local hard disk to the local catalogue. The implementation of the GE must enable users to select their new *.wgt packaged MAC and upload it to the Local Catalogue.
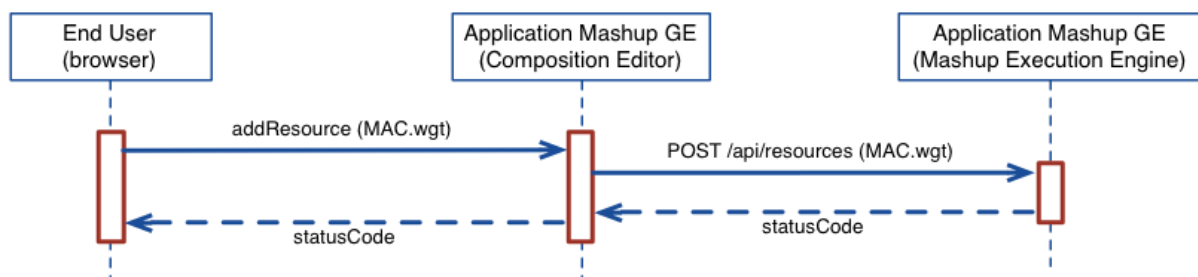
Uploading a packaged MAC to the Local Catalogue

It should also be possible to upload the XML template of the MAC, where the Application Mashup GE is in charge of getting and storing the linked source code of the MAC in the Local Catalogue.
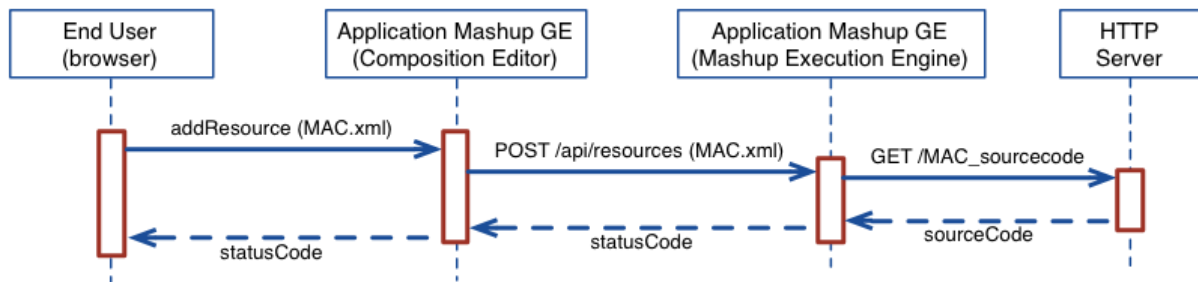
Uploading a MAC from its XML template to the Local Catalogue

Regardless of the upload method, this interaction must result in the uploaded MAC being stored in the local catalogue, ready for configuration and/or deployment. In other words, the MAC will be at the **Bought/Installed state** of its lifecycle. A `HTTP/1.1 201 Created` Status Code response will be received if the interaction went well.

Note that there are two possible scenarios if the uploaded MAC is a mashup:

- If the widgets within the mashup are currently installed in the local catalogue, the uploaded mashup will reference the widgets and will run out-of-the-box.

- If some of the widgets have a commercial license, and the license has to be bought for the widget to be used, the uploaded mashup will be installed, but the GE must warn and notify users that they have to buy the licensed widgets.

Export a MAC from the local catalogue

Platform users must be able to export (download) any of the MACs they have installed in the local catalogue. Users should select the MACs they want to export and click on an export button for the Application Mashup GE implementation to generate a packaged version of the selected MACs, enabling users to download MACs using their Web browser.
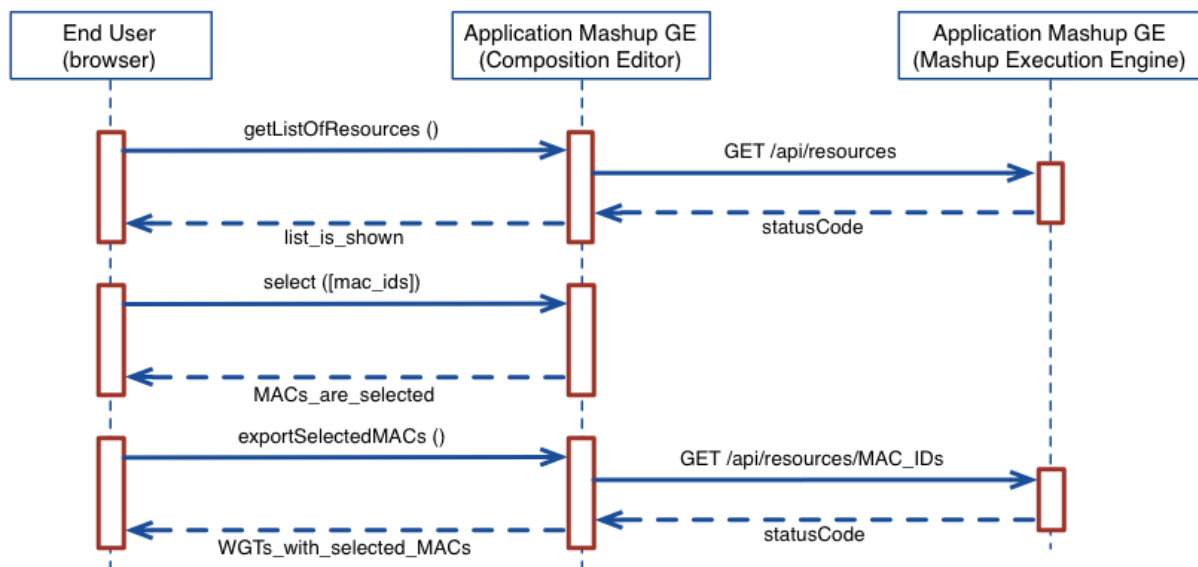
Exporting (downloading) a packaged MAC from the Local Catalogue

Deploy a MAC to get a new runtime instance

The **Deploy MAC** functionality instantiates a mashup or widget, that is available in the local catalogue in the Mashup Execution Engine. This operation is invoked by the mashup developer from the Composition Editor. The call should include the following information:

- *macID*: id of the mashup for instantiation.
- *wsID*: id of the active user workspace.
- *tabID*: id of the current tab within the active user workspace.

As a result of this invocation, the engine will get the MAC template from the local catalogue and will execute the MAC. It will also be available in the Mashup Editor mashup developer workspace. Its state will switch to **deployed**.
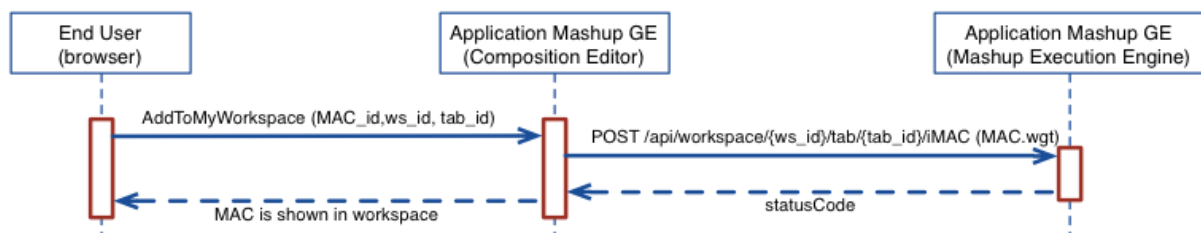
Interaction with the Mashup Execution Engine to deploy both mashups and widgets

Undeploy a MAC. Remove an instance from execution

The operation **Undeploy MAC** removes a mashup application or a widget from execution. Mashup developers invoke this operation from the mashup composition editor when they want to delete the whole mashup application or just a single widget.

The mashupID is required to undeploy a mashup:
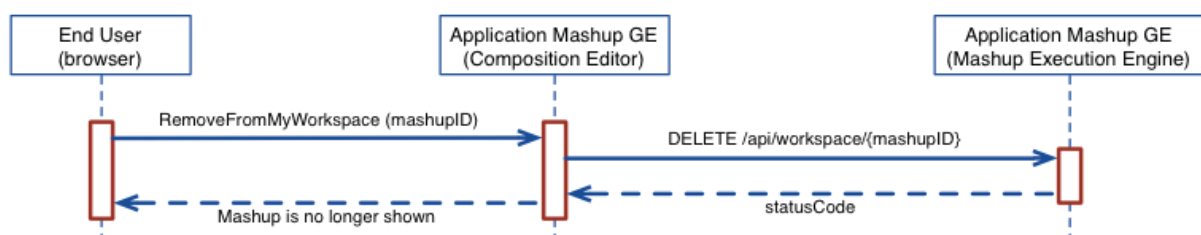
Interaction needed to undeploy a mashup

The widgetID, workspaceID and tabID are required to undeploy a widget.
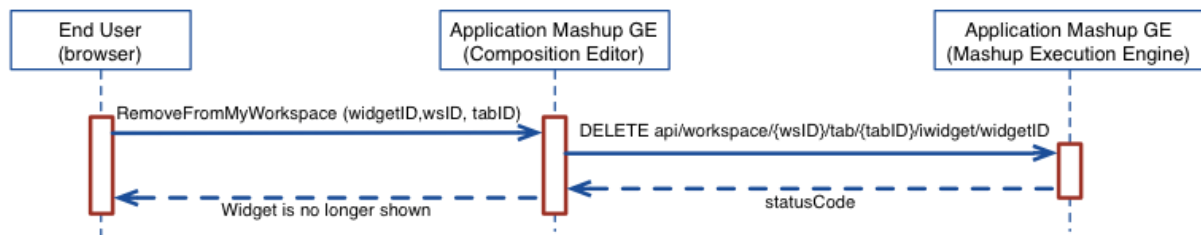
Interaction needed to undeploy a widget

As a result of this operation, the mashup or widget will stop executing, and thus it will reach the **bought/installed** state in its lifecycle.

Interconnect widgets/operators using wiring

Wiring functionality enables users to connect one or more widgets/operators to one or more other widgets/operators by means of a channel. The Composition Editor will help users to connect one or more of the possible outputs of a widget/operator with the input of a another widget/operator. This way, data flows between MACs allowing the mashup application to act as an information and process dashboard.

Besides, piping deals with how operators can bind to a specific backend service to gain access to data provided by the service. Then, users can wire the operator outputs either to other operators in order to perform some kind of data filtering/adaptation processes or to other widgets to consume the data.

The figure below shows how users set up channels between widgets/operators for wiring/piping.
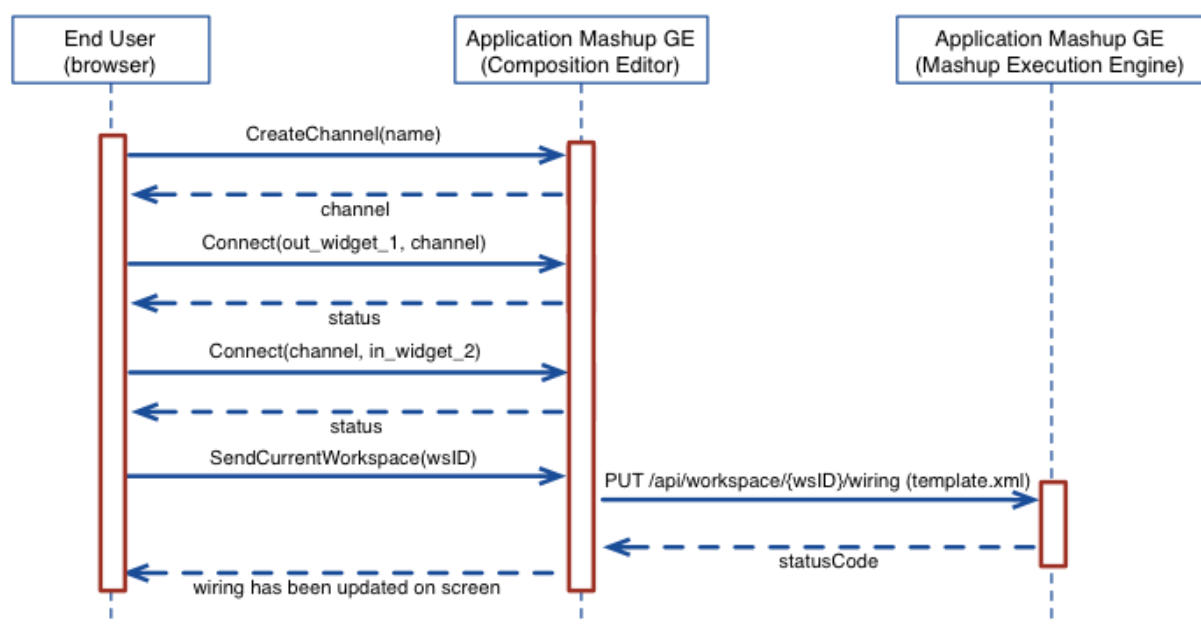
Widget Interconnection Management process

3.6.2.2   *Interactions with other FI-WARE Generic Enablers*

The Application Mashup GE is closely related with those Business-related FI-WARE GEs. The main interactions are as follows.

Buy MACs from a Store

Buying both mashup applications, widgets and operators from an external FI-WARE Store is one of the topics to be taken into account by the Application Mashup GE. Once users have used the Marketplace GE to search and select the MAC that they want to buy, the Application Mashup GE must provide the Store with a URL (through the Application Mashup RESTful API) and request the uploading of the bought MAC. The Store GE will be the actor that uploads the MAC to the local catalogue. Once the MAC has been uploaded, it is automatically installed in the local catalogue. From this point onwards, it will be possible for the MAC to be instantiated and thus executed.
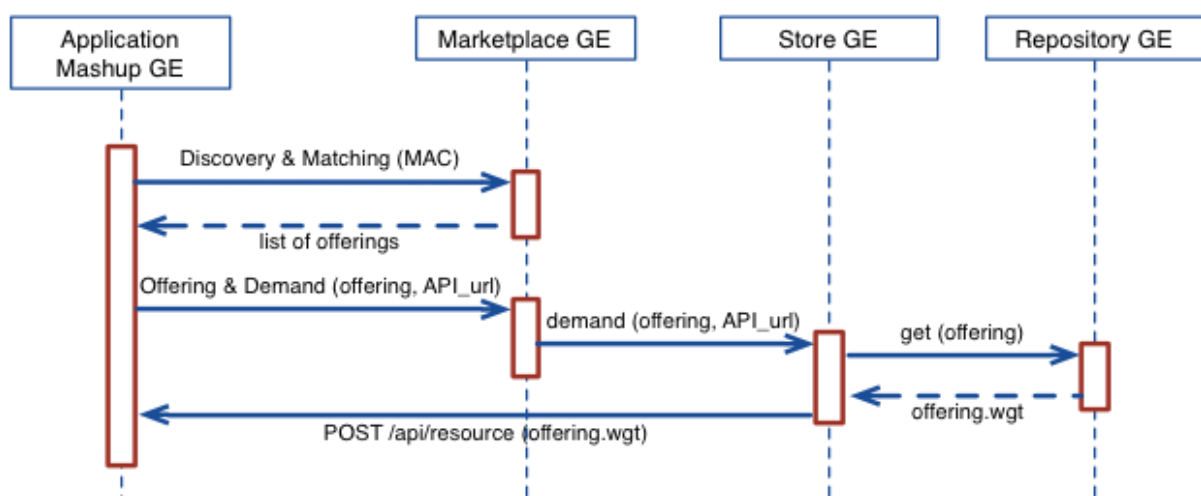
Buying a new MAC from a Store

Publish a MAC to a Store

Once users have either developed and uploaded a new widgets to the local catalogue or created their own mashup application, the Application Mashup GE must enable users to contact the Store GE to publish their brand new MAC.

The Application Mashup GE must enable users to notify the Store GE that they intend to publish the MAC. The Store GE is responsible for the entire transactions process (i.e. sending of pricing data, card number, etc.), which is out of the scope of the Application Mashup GE. However, at some point of the interaction, the Application Mashup must send the packaged MAC to the Store/Repository for storage when it is set to **published**.
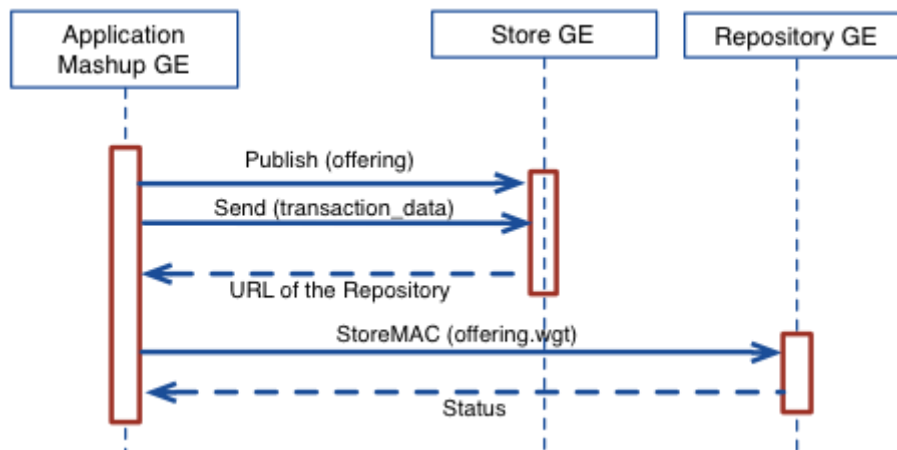
Publishing a new MAC to a given Store

Add Marketplace

The Application Mashup GE must be able to include new Marketplaces to search for new MACs. To do this, Marketplaces must be accessible from the Editor and new ones could be added from time to time. This functionality adds the Marketplace URI to the Composition Editor's internal list of marketplaces.

Adding the URL of a new Marketplace

3.7 Basic Design Principles

- **API Technology Independence**

  The API abstracts from the specific implementation technology. Implementations using more than one type of platform and framework should be possible.

- **Web Browsers should not limit the functionalities of the Application Mashup GE**

  HTML5, CSS and JavaScript must be used to fully exploit the brand new Web applications capabilities.

- **User-matched interaction abstraction level**

  Editors could cater for different user expertise (from technical experts skilled in the composition language to domain experts without technical expertise or even simple end users with no programming or technical skills) and roles (from composed service creators, to resellers and finally prosumers) by hiding complexity behind different types of building blocks, trading off flexibility for simplicity.

3.8   Detailed Specifications

3.8.1   Open API Specifications

The Application Mashup GE offers two separate APIs that cannot be combined because of they are of different types: the WidgetAPI is a JavaScript API, whereas the ApplicationMashupAPI is a RESTful API:

- FIWARE.OpenSpecification.Apps.WidgetAPI

- FIWARE.OpenSpecification.Apps.ApplicationMashupAPI

The Application Mashup GE will access the Repository, Registry and Marketplace via their REST API:

- Repository Open RESTful API Specification (PRELIMINARY)

- FIWARE.OpenSpecification.Apps.RegistryREST

- FIWARE.OpenSpecification.Apps.MarketplaceRegistrationREST

- FIWARE.OpenSpecification.Apps.MarketplaceOfferingsREST

- FIWARE.OpenSpecification.Apps.MarketplaceSearchREST

3.8.2   Other Open Specifications

The Application Mashup GE will use information retrieved from the Repository and the Marketplace using the Linked USDL specifications:

- Linked USDL Core Vocabulary

- Linked USDL Pricing Vocabulary

- Linked USDL Service Level Agreements Vocabulary

- Linked USDL Security Vocabulary

3.9   Re-utilized Technologies/Specifications

The Application Mashup GE requires both authentication and authorization in order to safeguard the different compositions from its users.

- Use of the OAuth2 protocol is recommended: http://tools.ietf.org/html/draft-ietf-oauth-v2-30

On the other hand, it relies on the Marketplace, Store and Repository GEs, and it must support the following technologies and specifications:

- RESTful web services

- HTTP/1.1

- JSON and XML data serialization formats
- Linked USDL

There are a number of widget- and mashup-related specifications that should be considered and/or contributed to, including:

3.9.1   OpenSocial

OpenSocial is a public specification that defines a component hosting environment (container) and a set of common application programming interfaces (APIs) for web-based applications.

3.9.2   Widget Packaging and XML Configuration

W3C has published a set of specifications describing their view about widgets. "Widget Packaging and XML Configuration" is one of the most prominent specifications in this set.

3.9.3   OpenAjax Hub 2.0 Specification

The OpenAjax Hub is a standard JavaScript functionality set defined by the OpenAjax Alliance that addresses key interoperability and security issues that arise when multiple Ajax libraries and/or components are used within the same web page. The OpenAjax Hub represents one of the key technical contributions of the OpenAjax Alliance to the Ajax community consistent with the Alliance's mission. See http://www.openajax.org for information on the OpenAjax Alliance.

The key feature of OpenAjax Hub 2.0 is its publish/subscribe engine that includes a "Managed Hub" mechanism that enables a host application to isolate untrustworthy components into secure sandboxes.

3.9.4   OMA Enterprise Mashup Markup Language

The Open Mashup Alliance (OMA) is a consortium of individuals and organizations dedicated to the successful use of Enterprise Mashup technologies and adoption of an open language that promotes Enterprise Mashup interoperability and portability.

OMA has developed a free-to-use Enterprise Mashup Markup Language (EMML) to promote the development, interoperability and compatibility of Enterprise Mashup offerings. The OMA also provides a reference runtime implementation that processes mashup scripts written in EMML.

3.10   Terms and definitions

This section comprises a summary of terms and definitions introduced during the previous sections. It intends to establish a vocabulary that will be help to carry out discussions internally and with third parties (e.g., Use Case projects in the EU FP7 Future Internet PPP). For a summary of terms and definitions managed at overall FI-WARE level, please refer to FIWARE Global Terms and Definitions.

- **Aggregator (Role):** Supports domain specialists and third-parties in aggregating services and apps for new and unforeseen opportunities and needs. It does so by

providing the dedicated tooling for aggregating services at different levels: UI, service operation, business process or business object levels.

- **Application:** Applications in FI-Ware are composite services that have a IT supported interaction interface (user interface). In most cases consumers do not buy the application they rather buy the right to use the application (user license).

- **Broker (Role):** The business network's central point of service access, being used to expose services from providers that are to be delivered through the Broker's service delivery functionality. The broker is the central instance for enabling monetization.

- **Business Element:** Core element of a business model, such as pricing models, revenue sharing models, promotions, SLAs, etc.

- **Business Framework:** Set of concepts and assets responsible for supporting the implementation of innovative business models in a flexible way.

- **Business Model:** Strategy and approach that defines how a particular service/application is supposed to generate revenue and profit. Therefore, a Business Model can be implemented as a set of business elements which can be combined and customized in a flexible way and in accordance to business and market requirements and other characteristics.

- **Business Process:** Set of related and structured activities producing a specific service or product, thereby achieving one or more business objectives. An operational business process clearly defines the roles and tasks of all involved parties inside an organization to achieve one specific goal.

- **Business Role:** Set of responsibilities and tasks that can be assigned to concrete business role owners, such as a human person or a software component.

- ''''*Channel: Resources through which services are accessed by end users. Examples for well-known channels are Web sites/portals, web-based brokers (like iTunes, eBay and Amazon), social networks (like Facebook, LinkedIn and MySpace), mobile channels (Android, iOS) and work centres. The mode of access to these channels is governed by technical channels like the Web, mobile devices and voice response, where each of these channels requires its own specific workflow.*

- **Channel Maker (Role):** Supports parties in creating outlets (the Channels) through which services are consumed, i.e. Web sites, social networks or mobile platforms. The Channel Maker interacts with the Broker for discovery of services during the process of creating or updating channel specifications as well as for storing channel specifications and channeled service constraints back in the Broker.

- **Composite Service (composition):** Executable composition of business back-end MACs. Common composite services are either orchestrated or choreographed. Orchestrated compositions are defined by a centralized control flow managed by a unique process that orchestrates all the interactions (according to the control flow) between the external services that participate in the composition. Choreographed compositions do not have a centralized process, thus the services participating in the composition autonomously coordinate each other according to some specified coordination rules. Backend compositions are executed in dedicated process execution engines. Target users of tools for creating Composites Services are technical users with algorithmic and process management skills.

- **Consumer (Role):** Actor who searches for and consumes particular business functionality exposed on the Web as a service/application that satisfies her own

needs.

- **Desktop Environment:** Multi-channel client platform enabling users to access and use their applications and services.

- **Event-driven Composition:** Components concerned with the composition of business logic which is driven by asynchronous events. This implies run-time selection of MACs and the creation/modification of orchestration workflows based on composition logic defined at design-time and adapted to context and the state of the communication at run-time.

- **Front-end/Back-end Composition:** Front-end compositions define a front-end application as an aggregation of visual mashable application pieces (named as widgets, gadgets, portlets, etc.) and back-end services. Front-end compositions interact with end-users, in the sense that front-end compositions consume data provided by the end-users and provide data to them. Thus the frontend composition (or mashup) will have direct influence on the application look and feel; every component will add a new user interaction feature.

- Back-end compositions define a back-end business service (also known as process) as an aggregation of backend services as defined for service composition term, the end-user being oblivious to the composition process. While back-end components represent atomization of business logic and information processing, front-end components represent atomization of information presentation and user interaction.

- **Gateway (Role):** The Gateway role enables linking between separate systems and services, allowing them to exchange information in a controlled way despite different technologies and authoritative realms. A Gateway provides interoperability solutions for other applications, including data mapping as well as run-time data store-forward and message translation. Gateway services are advertised through the Broker, allowing providers and aggregators to search for candidate gateway services for interface adaptation to particular message standards. The Mediation is the central generic enabler. Other important functionalities are eventing, dispatching, security, connectors and integration adaptors, configuration, and change propagation.

- **Hoster (Role):** Allows the various infrastructure services in cloud environments to be leveraged as part of provisioning an application in a business network. A service can be deployed onto a specific cloud using the Hoster's interface. This enables service providers to re-host services and applications from their on-premise environments to cloud-based, on-demand environments to attract new users at much lower cost.

- **Marketplace:** Part of the business framework providing means for service providers to publish their service offerings and service consumers to compare and select a specific service implementation. A marketplace can offer services from different stores and thus different service providers. The actual buying of a specific service is handled by the related service store.

- **Mashup:** Executable composition of front-end MACs. There are several kinds of mashups, depending on the technique of composition (spatial rearrangement, wiring, piping, etc.) and the MACs used. They are called application mashups when applications are composed to build new applications and services/data mash-ups if services are composed to generate new services. While composite service is a common term in backend services implementing business processes, the term 'mashup' is widely adopted when referring to Web resources (data, services and applications). Front-end compositions heavily depend on the available device environment (including the chosen presentation channels). Target users of mashup

platforms are typically users without technical or programming expertise.

- **Mashable Application Component (MAC):** Functional entity able to be consumed executed or combined. Usually this applies to components that will offer not only their main behaviour but also the necessary functionality to allow further compositions with other components. It is envisioned that MACs will offer access, through applications and/or services, to any available FI-WARE resource or functionality, including gadgets, services, data sources, content, and things. Alternatively, it can be denoted as 'service component' or 'application component'.

- **Mediator:** A mediator can facilitate proper communication and interaction amongst components whenever a composed service or application is utilized. There are three major mediation area: Data Mediation (adapting syntactic and/or semantic data formats), Protocol Mediation (adapting the communication protocol), and Process Mediation (adapting the process implementing the business logic of a composed service).

- **Monetization:** Process or activity to provide a product (in this context: a service) in exchange for money. The Provider publishes certain functionality and makes it available through the Broker. The service access by the Consumer is being accounted according to the underlying business model and the resulting revenue is shared across the involved service providers.

- **Premise (Role):** On-Premise operators provide in-house or on-site solutions, which are used within a company (such as ERP) or are offered to business partners under specific terms and conditions. These systems and services are to be regarded as external and legacy to the FI-Ware platform because they do not conform to the architecture and API specifications of FI-Ware. They will only be accessible to FI-Ware services and applications through the Gateway.

- **Prosumer:** A user role able to produce, share and consume their own products and modify/adapt products made by others.

- **Provider (Role):** Actor who publishes and offers (provides) certain business functionality on the Web through a service/application endpoint. This role also takes care of maintaining this business functionality.

- **Registry and Repository:** Generic enablers that able to store models and configuration information along with all the necessary meta-information to enable searching, social search, recommendation and browsing, so end users as well as services are able to easily find what they need.

- **Revenue Settlement:** Process of transferring the actual charges for specific service consumption from the consumer to the service provider.

- **Revenue Sharing:** Process of splitting the charges of particular service consumption between the parties providing the specific service (composition) according to a specified revenue sharing model.

- **Service:** We use the term service in a very general sense. A service is a means of delivering value to customers by facilitating outcomes customers want to achieve without the ownership of specific costs and risks. Services could be supported by IT. In this case we say that the interaction with the service provider is through a technical interface (for instance a mobile app user interface or a Web service). Applications could be seen as such IT supported Services that often ar also composite services.

- **Service Composition:** in SOA domain, a service composition is an added value service created by aggregation of existing third party services according to some

predefined work and data flow. Aggregated services provide specialized business functionality on which the service composition functionality has been split down.

- **Service Delivery Framework:** Service Delivery Framework (or Service Delivery Platform (SDP)) refers to a set of components that provide service delivery functionality (such as service creation, session control & protocols) for a type of service. In the context of FI-WARE, it is defined as a set of functional building blocks and tools to (1) manage the lifecycle of software services, (2) creating new services by creating service compositions and mashups, (3) providing means for publishing services through different channels on different platforms, (4) offering marketplaces and stores for monetizing available services and (5) sharing the service revenues between the involved service providers.

- **Service Level Agreement (SLA):** A service level agreement is a legally binding and formally defined service contract between a service provider and a service consumer, specifying the contracted qualitative aspects of a specific service (e.g. performance, security, privacy, availability or redundancy). In other words, SLAs not only specify that the provider will just deliver some service, but that this service will also be delivered on time, at a given price, and with money back if the pledge is broken.

- **Service Orchestration:** in SOA domain, a service orchestration is a particular architectural choice for service composition where a central orchestrated process manages the service composition work and data flow invocations the external third party services in the order determined by the work flow. Service orchestrations are specified by suitable orchestration languages and deployed in execution engines who interpret these specifications.

- **Store:** An external component integrated with the business framework offering a set of services that are published to a selected set of marketplaces. The store thereby holds the service portfolio of a specific service provider. In case a specific service is purchased on a service marketplace, the service store handles the actual buying of a specific service (as a financial business transaction).

- **Unified Service Description Language (USDL):** USDL is a platform-neutral language for describing services, covering a variety of service types, such as purely human services, transactional services, informational services, software components, digital media, platform services and infrastructure services. The core set of language modules offers the specification of functional and technical service properties, legal and financial aspects, service levels, interaction information and corresponding participants. USDL is offering extension points for the derivation of domain-specific service description languages by extending or changing the available language modules.

4   FIWARE OpenSpecification Apps WidgetAPI

You can find the content of this chapter as well in the wiki of fi-ware.

4.1   Introduction to the *Widget* API

Please check the FI-WARE Open Specifications Legal Notice to understand the rights to use FI-WARE Open Specifications.

4.1.1   Widget API Core

The Application Mashup GE offers two separate APIs that cannot be combined because of their different nature: The Widget API (the subject of this entry) is a JavaScript API, while the ApplicationMashupAPI is a RESTful one. You can find the Application Mashup Open RESTful API in this separate entry:

- FIWARE.OpenSpecification.Apps.ApplicationMashupAPI

The Widget API is a JavaScript API that allows deployed widgets in a Mashup Execution Engine to gain access to its functionalities. It does not make sense to expose it as a RESTful API since it needs to be consumed by a widget in its own local execution environment. Amongst other functionalities, this API allows the widgets to gain access to remote resources. For example, in order gain access to a remote REST API or to resolve cross-domain problems, a widget needs to use a proxy through the Widget API.

4.1.2   Intended Audience

This specification is intended for service front-end (aka gadget/widget) developers. This document provides a full specification of how to make widgets interoperate with the Mashup Execution Engine. To use this information, the reader should firstly have a general understanding of the Generic Enablers for Composition and Mashup.

You should also be familiar with:

- JavaScript

4.1.3   API Change History

Revision Date	Changes Summary
Apr 27, 2012	• Initial version

4.1.4   How to Read This Document

It is assumed that reader is familiarized with JavaScript. Along the document, some special notations are applied to differentiate some special words or concepts. The following list summarizes these special notations.

- A bold font is used to represent method names.
- Function parameters are represented in italic font.

For a description of some terms used along this document, see Composition Execution engine .

4.2   *Widget* API

4.2.1   MashupPlatform.http

4.2.1.1   *request options*

Generic options:

- *contentType* (String; default *application/x-www-form-urlencoded*): The Content-type header for your request. Change this header if you want to send data in another format (like XML).

- *encoding* (String; default *UTF-8*): The encoding for the contents of your request. It is best left as-is, but should weird encoding issues arise, you may have to tweak this.

- *method* (String; default *POST*): The HTTP method to use for the request.

- *parameters* (Object): The parameters for the request, which will be encoded into the URL for a get method, or into the request body for the other methods.

- *postBody* (String): Specific contents for the request body on a post method. If it is not provided, the contents of the parameters option will be used instead.

- *requestHeaders* (Object): A set of key-value pairs, with properties representing header names.

- *forceProxy* (Boolean; default *false*): Sends the request through the proxy regardless of the other options passed.

- *context* (Object; default *null*) is the value to be passed as the *this* parameter to the callbacks.

Callback options:

- *onSuccess*: Invoked when a request completes and its status code belongs in the 2xy family. This is skipped if a code-specific callback is defined (e.g., on200), and happens before *onComplete*.

- *onFailure*: Invoked when a request completes and its status code exists but is not in the 2xy family. This is skipped if a code-specific callback is defined (e.g. on403), and happens before *onComplete*.

- *onXYZ* (with XYZ representing any HTTP status code): Invoked just after the response is complete if the status code is the exact code used in the callback name. Prevents execution of *onSuccess* and *onFailure*. Happens before *onComplete*.

- *onComplete*: Triggered at the very end of a request's life-cycle, after the request completes, status-specific callbacks are called, and possible automatic behaviors are processed. Guaranteed to run regardless of what happened during the request.

4.2.1.2   *Methods*

MashupPlatform.http.buildProxyURL(url, options)

Builds a URL suitable for working around the cross-domain problem. This usually is handled using the Mashup Execution Engine proxy but it also can be handled using the access control request headers if the browser has support for them. If all the needed requirements are meet, this function will return a URL without using the proxy.

- *url* - Target URL.

- *options* - Optional object with request options (see the request options section for more details).

MashupPlatform.http.makeRequest(url, options)

Sends a HTTP request.

- *url* - Target URL of the request.

- *options* - Optional object with request options (see the request options section for more details).

4.2.2   MashupPlatform.wiring

4.2.2.1   *Methods*

MashupPlatform.wiring.pushEvent(outputName, data)

Sends an event through the wiring.

- *outputName* - Name of the output endpoint as defined in the GDL.

- *data* - Event content.

MashupPlatform.wiring.registerCallback(inputName, callback)

Registers a callback for a given input endpoint. If the given endpoint already has registered a callback, it will be replaced by the new one.

- *inputName* - Name of the input endpoint as defined in the GDL.

- *callback* - Callback function to use when an event reaches the given input endpoint.

4.2.3   MashupPlatform.prefs

Widgets may use the methods defined in this module to retrieve and to be notified of changes in the values of their preferences.

4.2.3.1   *Methods*

MashupPlatform.prefs.get(key)

Retrieves the value of a preference.

- *key* - Name of the preference to fetch.

MashupPlatform.prefs.registerCallback(callback)

Registers a callback for listening preference changes.

- *callback* - Callback function that will be called when widget's preferences are changed.

5 FIWARE OpenSpecification Apps ApplicationMashupAPI

You can find the content of this chapter as well in the wiki of fi-ware.

5.1 Introduction to the *Application Mashup* API

The FI-WARE Generic Enabler Specification are owned by different partners. Therefore, different Legal Notices might apply. Please check for each FI-WARE Generic Enabler Specification the Legal Notice attached. For this FI-WARE Generic Enabler Specification, this Legal Notice applies.

5.1.1 Application Mashup Core

The Application Mashup GE offers two separate APIs that cannot be combined because of their different nature: The Widget API is a JavaScript API, while the Application Mashup API (the subject of this entry) is a RESTful one. You can find the Widget API in this separate entry:

- FIWARE.OpenSpecification.Apps.WidgetAPI

The Application Mashup API is a RESTful, resource-oriented API accessed via HTTP that uses various representations for information interchange.

5.1.2 Intended Audience

This specification is intended for both software developers and reimplementers of this API. For the former, this document provides a full specification of how to interoperate with products that implement the Application Mashup API. For the latter, this specification indicates the interface to be implemented and provided to clients.

To use this information, the reader should firstly have a general understanding of the Generic Enabler service Application Mashup. You should also be familiar with:

- RESTful web services
- HTTP/1.1
- JSON and/or XML data serialization formats.

5.1.3 API Change History

This version of the Application Mashup API Guide replaces and obsoletes all previous versions. The most recent changes are described in the table below:

Revision Date	Changes Summary
Nov 2, 2012	• Initial version

5.1.4 How to Read This Document

It is assumed that the reader is familiar with the REST architecture style. Within the document, some special notations are applied to differentiate some special words or concepts. The following list summarizes these special notations.

- A bold, mono-spaced font is used to represent code or logical entities, e.g., HTTP method (`GET, PUT, POST, DELETE`).

- An italic font is used to represent document titles or some other kind of special text, e.g., *URI*.

- Variables are represented between brackets, e.g. {id} and in italic font. The reader can replace the id with an appropriate value.

For a description of some terms used along this document, see ApplicationMashup.

5.1.5   Additional Resources

You can download the most current version of this document from the FI-WARE API specification website at Application Mashup API. For more details about the Application Mashup GE that this API is based upon, please refer to High Level Description. Related documents, including an Architectural Description, are available at the same site.

5.2   General *Mashup Application* API Information

5.2.1   Resources Summary
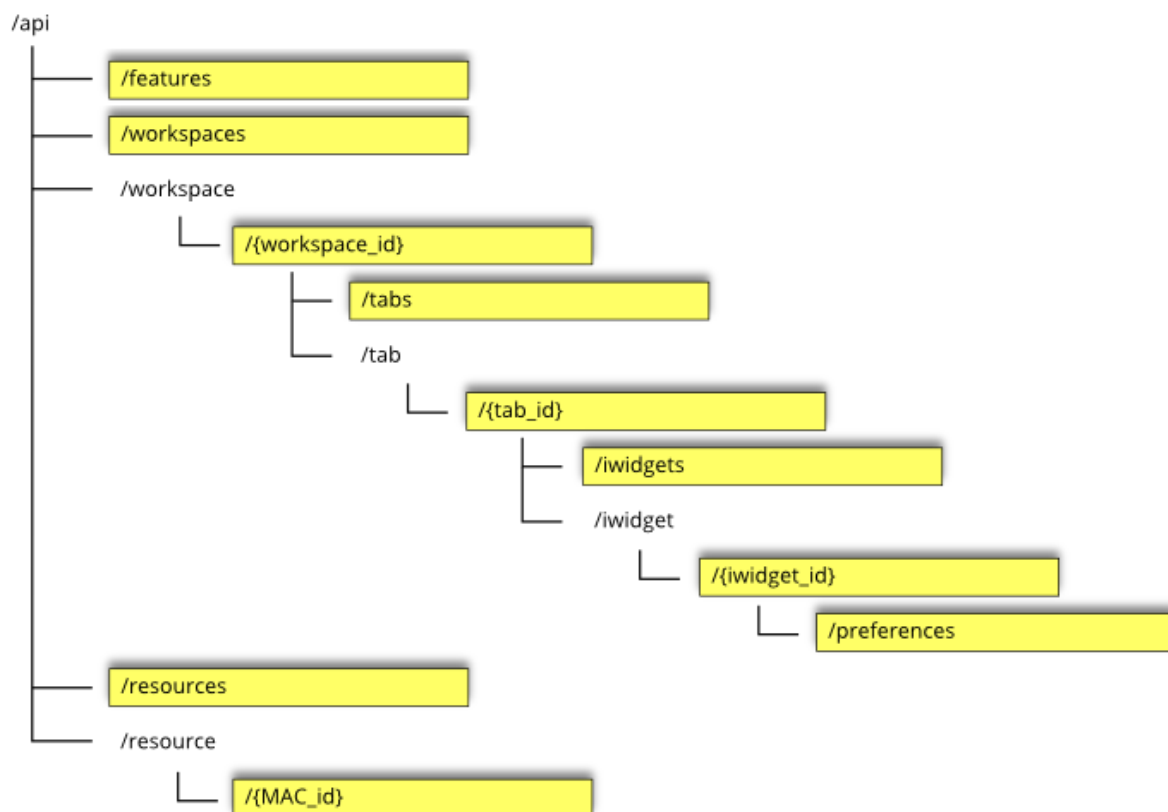
5.2.2   Authentication

Each HTTP request against the *Application Mashup API* requires the inclusion of specific authentication credentials. The specific implementation of this API may support multiple

authentication schemes (OAuth, Basic Auth, Token) and will be determined by the specific provider that implements the GE. Please contact with it to determine the best way to authenticate against this API. Remember that some authentication schemes may require that the API operate using SSL over HTTP (HTTPS).

5.2.3   Representation Format

The *Application Mashup* API supports at least JSON for delivering any kind of resources, it may also support simple text, XML and HTML output format. The request format is specified using the Content-Type header and is required for operations that have a request body. The response format can be specified in requests using the Accept header. Note that it is possible for a response to be serialized using a format different from the request.

The interfaces should support data exchange through multiple formats:

- *text/plain* - A linefeed separated list of elements for easy mashup and scripting.

- *text/html* - An human-readable HTML rendering of the results of the operation as output format.

- *application/json* - A JSON representation of the input and output for mashups or JavaScript-based Web Apps

- *application/xml* - A XML description of the input and output.

- *application/x-www-form-urlencoded* - May be used for submitting using HTML forms.

- *multipart/form-data* - Should be used for submitting HTML forms containing files.

- *application/octet-stream* - Used for uploading/downloading packaged Mashable Application Components.

5.2.4   Representation Transport

Resource representation is transmitted between client and server by using HTTP 1.1 protocol, as defined by IETF RFC-2616. Each time an HTTP request contains payload, a Content-Type header shall be used to specify the MIME type of wrapped representation. In addition, both client and server may use as many HTTP headers as they consider necessary.

5.2.5   Resource Identification

The resource identification for HTTP transport is made using the mechanisms described by HTTP protocol specification as defined by IETF RFC-2616.

5.2.6   Links and References

The Application Mashup API is relying on Web principles:

- consistent URI structure based on REST style protocol

- HTTP content negotiation to allow the client to choose the appropriate data format supporting XML, JSON, ...

5.2.7   Limits

We can manage the capacity of the system in order to prevent the abuse of the system through some limitations. These limitations will be configured by the operator and may differ from one implementation to other of the GE implementation.

5.2.7.1   *Rate Limits*

These limits are specified both in human readable wild-card and in regular expressions and will indicate for each HTTP verb which will be the maximum number of operations per time unit that a user can request. After each unit time the counter is initialized again.

In the event a request exceeds the thresholds established for your account, a 413 HTTP response will be returned with a Retry-After header to notify the client when they can attempt to try again.

5.2.8   Versions

The Mashup Application API is considered to be an extension of itself, so you can query current version of the Mashup Application API using the resource described in the next section. The core extension defining the Mashup Application API is known as "ApplicationMashup".

5.2.9   Extensions

The Application Mashup GE could be extended. The Application Mashup GE provides the following resource. This allow the introduction of new features in the API without requiring an update of the version, for instance, or to allow the introduction of vendor specific functionality.

Verb	URI	Description
GET	/api/features	List of all available extensions

Example request:

```
GET /api/features HTTP/1.1

Accept: application/json
```

Example response:

```
200 OK

Content-Type: application/json

{

    "ApplicationMashup": "1.0"

}
```

5.2.10   Faults

5.2.10.1   *Synchronous Faults*

Error responses will be encoded using the most appropriated content-type in base to the Accept header of the request. In any case, the response will provide an human-readable message for displaying to end users.

XML Example:

```
<error>Resource already exists</error>
```

JSON Example:

```
{

    "error": "Resource already exists"

}
```

Fault Element	Associated Error Codes	Expected in All Requests?
Bad Request	400	YES
Unauthorized	403	YES
Not Found	404	YES
Request Entity Too Large	413	YES
Internal Server error	50X	YES

5.3   API Operations

5.3.1   Managing Workspaces

Here we start with the description of the operation following the next table:

Verb	URI	Description
GET	/api/workspaces	Get a list of all workspaces owned by the user
POST	/api/workspaces	Creates a new workspace
GET	/api/workspace/{workspace_id}	Get info about a specific workspace
DELETE	/api/workspace/{workspace_id}	Delete a workspace
PUT	/api/workspace/{workspace_id}/wiring	Updates workspace wiring configuration
POST	/api/workspace/{workspace_id}/tabs	Creates a new workspace tab
DELETE	/api/workspace/{workspace_id}/tab/{tab_id}	Delete a workspace tab

POST	/api/workspace/{workspace_id}/tab/{tab_id}/iwidgets	Add a new instance of a widget into the tab
POST	/api/workspace/{workspace_id}/tab/{tab_id}/iwidget/{iwidget_id}	Update iwidget information
POST	/api/workspace/{workspace_id}/tab/{tab_id}/iwidget/{iwidget_id}/preferences	Update iwidget preferences
DELETE	/api/workspace/{workspace_id}/tab/{tab_id}/iwidget/{iwidget_id}	Removes an iwidget from a tab

5.3.1.1 *Getting Workspaces*

Example request:

```
GET /api/workspaces HTTP/1.1

Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Vary: Cookie
{
  {
      "name":"tourist_app",
      "creator":"sptel",
      "owned":false,
      "removable":false,
      "active":true,
      "shared":true,
      "id":20
  }
}
```

5.3.1.2 *Creating Workspaces*

Example request:

```
POST /api/workspaces HTTP/1.1
Content-Type: application/json
Accept: application/json

{
  "name": "test"
}
```

Example response:

```
HTTP/1.1 201 Created
Content-Type: application/json
{
    "name":"test",
    "creator":"admin",
    "wiring":"{\"operators\": {}, \"connections\": []}",
    "empty_params":[],
    "active":false,
    "shared":false,
    "tabs":[
        {
            "visible":true,
            "iwidgets":[],
            "id":84,
            "name":"Tab",
            "preferences":{}
        }
    ],
    "id":81,
    "extra_prefs":{},
    "preferences":{}
}
```

5.3.1.3 *Creating Workspaces from Mashup*
Example request:

```
POST /api/workspaces HTTP/1.1
Content-Type: application/json
Accept: application/json

{
  "mashup": "UPM/Mashup/1.0"
}
```

Example response:

```
HTTP/1.1 201 Created
Content-Type: application/json

{
    "name":"example",
    "creator":"admin",
    "wiring":"{\"operators\": {}, \"connections\": []}",
    "empty_params":[],
    "active":false,
    "shared":false,
    "tabs":[
```

```
    {
        "visible":true,
        "iwidgets":[
                {
                    "widget":"UPM/Widget1/0.1",
                    "layout":0,
                    "name":"Widget1",
                    "icon_top":-1,
                    "variables":{},
                    "minimized":false,
                    "id":311,
                    "height":28,
                    "zIndex":0,
                    "width":6,
                    "readOnly":false,
                    "icon_left":-1,
                    "tab":3,
                    "transparency":false,
                    "refused_version":null,
                    "top":0,
                    "fulldragboard":false,
                    "left":0
                },
                {
                    "widget":"UPM/Widget2/0.1",
                    "layout":0,
                    "name":"Widget2",
                    "icon_top":-1,
                    "variables":{},
                    "minimized":false,
                    "id":312,
                    "height":28,
                    "zIndex":0,
                    "width":6,
                    "readOnly":false,
                    "icon_left":-1,
                    "tab":3,
                    "transparency":false,
                    "refused_version":null,
                    "top":0,
                    "fulldragboard":false,
                    "left":7
                }
        ],
        "id":100,
        "name":"Tab",
        "preferences":{}
    }
    ],
    "id":50,
    "extra_prefs":{},
    "preferences":{}
}
```

Notes:

'UPM/Mashup/1.0' is a mashup available on the local catalogue of the user.

5.3.1.4   *Getting Workspace details*
Example request:

```
GET /api/workspace/81 HTTP/1.1

Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Vary: Cookie

{
    "name":"test",
    "creator":"admin",
    "wiring":"{\"operators\": {}, \"connections\": []}",
    "empty_params":[],
    "active":false,
    "shared":false,
    "tabs":[
        {
            "visible":true,
            "iwidgets":[],
            "id":84,
            "name":"Tab",
            "preferences":{}
        }
    ],
    "id":81,
    "extra_prefs":{},
    "preferences":{}
}
```

5.3.1.5   *Deleting Workspaces*
Example request:

```
DELETE /api/workspace/81 HTTP/1.1

Accept: application/json
```

Example response:

```
HTTP/1.1 204 No Content
```

5.3.1.6   *Updating Workspace Wiring Configuration*
Example request:

```
PUT /api/workspace/81/wiring HTTP/1.1
Content-Type: application/json
Accept: application/json

{
    "operators":{
        "0":{
            "name":"UPM/Operator/0.1",
            "id":"0"
        }
    },
    "connections":[
        {
            "source":{
                "type":"iwidget",
                "id":311,
                "endpoint":"location_info_event"
            },
            "target":{
                "type":"iwidget",
                "id":312,
                "endpoint":"search_text_slot"
            }
        },
        {
            "source":{
                "type":"iwidget",
                "id":311,
                "endpoint":"location_info_event"
            },
            "target":{
                "type":"ioperator",
                "id":0,
                "endpoint":"message"
            }
        }
    ]
}
```

Example response:

```
HTTP/1.1 204 No Content
```

Notes:

'UPM/Operator/1.0' is a operator available on the local catalogue of the user.

5.3.1.7   *Creating Workspace Tabs*

Example request:

```
POST /api/workspace/81/tabs HTTP/1.1
Content-Type: application/json
```

```
Accept: application/json

{
  "name": "Tab 2"
}
```

Example response:

```
HTTP/1.1 201 Created
Content-Type: application/json

{
    "id": 3,
    "name":"Tab2"
}
```

5.3.1.8 *Deleting Workspace Tabs*
Example request:

```
DELETE /api/workspace/81/tab/3 HTTP/1.1
Accept: application/json
```

Example response:

```
HTTP/1.1 204 No Content
```

5.3.1.9 *Adding a New Instance of Widget into a Workspace Tab*

```
POST /api/workspace/81/tab/3/iwidgets HTTP/1.1
Content-Type: application/json
Accept: application/json

{
    "widget": "UPM/Widget/1.0"
}
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
{
   "widget":"UPM/Widget/1.0",
   "left":12,
   "top":0,
   "icon_left":-1,
   "icon_top":-1,
   "zIndex":2,
   "width":6,
   "height":28,
   "name":"Widget",
```

```
    "layout":0
}
```

Notes:

'UPM/Widget/1.0' is a widget available on the local catalogue of the user.

5.3.1.10 *Update IWidget Status*
Example request:

```
POST /api/workspace/81/tab/3/iwidget/5 HTTP/1.1

Content-Type: application/json

Accept: application/json


{

    "name": "new name"

}
```

Example response:

```
HTTP/1.1 204 No Content
```

5.3.1.11 *Update IWidget Preferences*
Example request:

```
POST /api/workspace/81/tab/3/iwidget/5/preferences HTTP/1.1
Content-Type: application/json
Accept: application/json

{
    "pref": "value"
}
```

Example response:

```
HTTP/1.1 204 No Content
```

5.3.1.12 *Remove IWidget From Workspace Tabs*
Example request:

```
DELETE /api/workspace/81/tab/3/iwidget/5 HTTP/1.1

Accept: application/json
```

Example response:

```
HTTP/1.1 204 No Content
```

5.3.2   Managing Local Catalogue

Verb	URI	Description
GET	/api/resources	Get a list of all resources (widgets, operators, etc..) available to the user
POST	/api/resources	Add a resource (widget, operator, etc..) to the local catalogue of the user
GET	/api/resource/{MAC_id}	Download a resource (widget, operator, etc..) from the local catalogue of the user
DELETE	/api/resource/{MAC_id}	Uninstall a resource (widget, operator, etc..) from the local catalogue of the user

5.3.2.1   *Obtaining the list of Mashable Application Components*
Example request:

```
GET /api/resources HTTP/1.1
Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
    {
      "type": "widget",
      "vendor":"CoNWeT",
      "name":"Multimedia Viewer"
      "version":"0.5",
      "description":"This widget allows watch youtube videos, flickr
images and another images.",
      "variables":{
        "urlEvent":{
           ....
        },
        "uriEvent":{
           ....
        },
        "apikeyPref":{
           ....
        },
        "uriSlot":{
           ....
        },
        ....
      },
```

```
    ...
    },
    {
        "type": "operator",
        "vendor":"UPM",
        "name":"Operator"
        "version":"1.0",
        ...
    }
]
```

5.3.2.2 *Uploading Mashable Application Components*
Example request:

```
POST /api/resources HTTP/1.1
Content-Type: multipart/form-data; boundary=----
WebKitFormBoundaryHPwaOXLATyUcGQp8
Accept: application/json

------WebKitFormBoundaryHPwaOXLATyUcGQp8
Content-Disposition: form-data; name="file"; filename="widget.wgt"
Content-Type: application/octet-stream

    ...

------WebKitFormBoundaryHPwaOXLATyUcGQp8--
```

Example response:

```
HTTP/1.1 201 Created
Content-Type: application/json

 {
   "type": "operator",
   "vendor":"UPM",
   "name":"Widget"
   "version":"1.0",
   ...
}
```

5.3.2.3 *Exporting Mashable Application Components*
Example request:

```
GET /api/resource/UPM/Widget/1.0 HTTP/1.1
Accept: application/octet-stream
```

Example response:

```
HTTP/1.1 200 OK
```

```
Content-type: application/octet-stream

Vary: Accept, Cookie

   ...
```

5.3.2.4   *Uninstalling Mashable Application Components*
Example request:

```
GET /api/resource/UPM/Widget/1.0 HTTP/1.1

Accept: application/json
```

Example response:

```
HTTP/1.1 204 No Content
```

5.3.3   Status Codes

200 OK

> The request was handled successfully and transmitted in response message.

201 Created

> The request has been fulfilled and resulted in a new resource being created.

204 No Content

> The server successfully processed the request, but is not returning any content.

304 Not Modified

> Indicates the resource has not been modified since last requested. Typically, the HTTP client provides a header like the If-Modified-Since header to provide a time against which to compare. Using this saves bandwidth and reprocessing on both the server and client, as only the header data must be sent and received in comparison to the entirety of the page being re-processed by the server, then sent again using more bandwidth of the server and client.

400 Bad Request

> The request cannot be fulfilled due to bad syntax.

404 Not Found

> The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible.

500 Internal Server Error

> A generic error message, given when no more specific message is suitable.

6 FIWARE OpenSpecification Apps ServiceComposition

You can find the content of this chapter as well in the wiki of fi-ware.

Name	FIWARE.OpenSpecification.Apps.ServiceComposition
Chapter	Apps,
Catalogue-Link to Implementation	[Ericsson Composition]
Owner	EAB, Calin Curescu

Disclaimer: The sustainability of this Open Specification cannot be guaranteed due to internal changes in the project consortium.

6.1 Preface

Within this document you find a self-contained open specification of a FI-WARE generic enabler, please consult as well the FI-WARE_Product_Vision, the website on http://www.fi-ware.eu and similar pages in order to understand the complete context of the FI-WARE project.

6.2 Copyright

- Copyright © 2012 by EAB

6.3 Legal Notice

Please check the following Legal Notice to understand the rights to use these specifications.

6.4 Overview

The Service Composition is a core enabler of the FI-WARE Platform. It allows users to create, manage and execute composed services. It consists of to main parts, the editor and the execution environment. The editor provides users with a graphical environment that allows them to create composed services in a more convenient way, providing graphical constructs for flow control and component service templates (and hiding away some of the service communication and data representation details). These composed service representations (i.e. skeletons) specify the main parts of the business logic of the composed services. During the run-time the composition engine dynamically decides about what services to invoke or which data source to use based on constraints evaluated at that particular time. Essentially the composition engine is creating the workflow step-by-step during runtime, and different composition decisions can be taken depending on external constraints or on the return values of previously executed services.

As an intermediary step, the composed service specification can be stored/fetched from the Repository GE. Moreover the Service Composition GE can search (based on different

criteria) the Marketplace GE for services to either be used in a new composition or to be executed by the execution environment.

6.4.1   Target usage

The Service Composition GE helps the service provider to create composed services. Editors should provide an environment to combine and configure applications and services in graphical way. The editor could cater for different user expertise (from technical experts to domain experts without technical expertise or even simple end-users with no programming or technical skills) and roles (from composed service creators, to resellers and finally to prosumers) by hiding complexity behind different types of construction blocs, trading off flexibility for simplicity. The Service Composition GE should allow testing, debugging, installing, executing, controlling and post execution analysis of the composed applications.

Composition descriptions and technical service descriptions should be stored/fetched to/from the Repository GE. The Service Composition could be connected to a user and identity management service for controlling access to the applications.

When creating compositions/mashups editors might connect o the business infrastructure:

- Marketplace to search for services

- Shops to purchase component services them for testing /deployment, and to expose composed services for purchase

- USDL Registry to browse business information related to services

6.5 Composition Provider Architecture
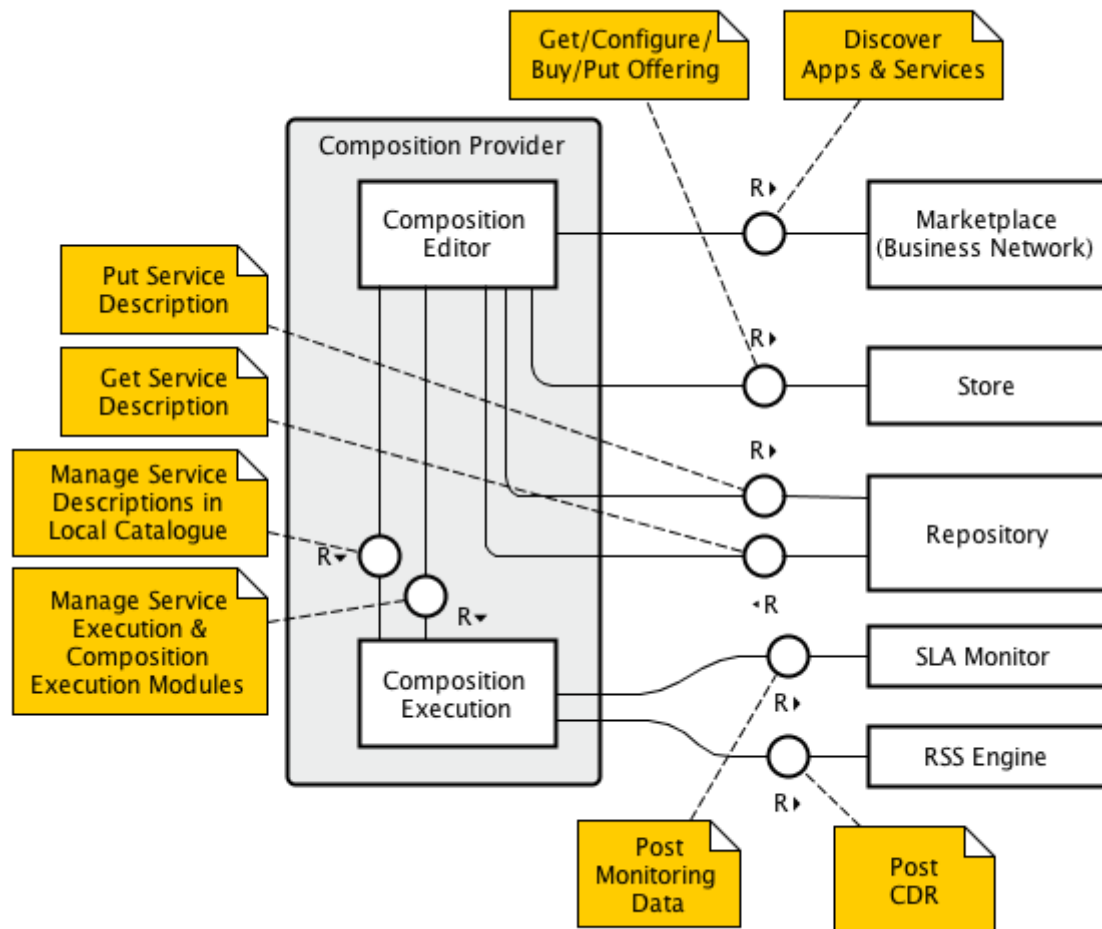
Composition Provider Architecture

6.6 Basic Concepts

The Service Composition GE offers two main functions: composition creation and composition execution.

6.6.1 Composition Creation
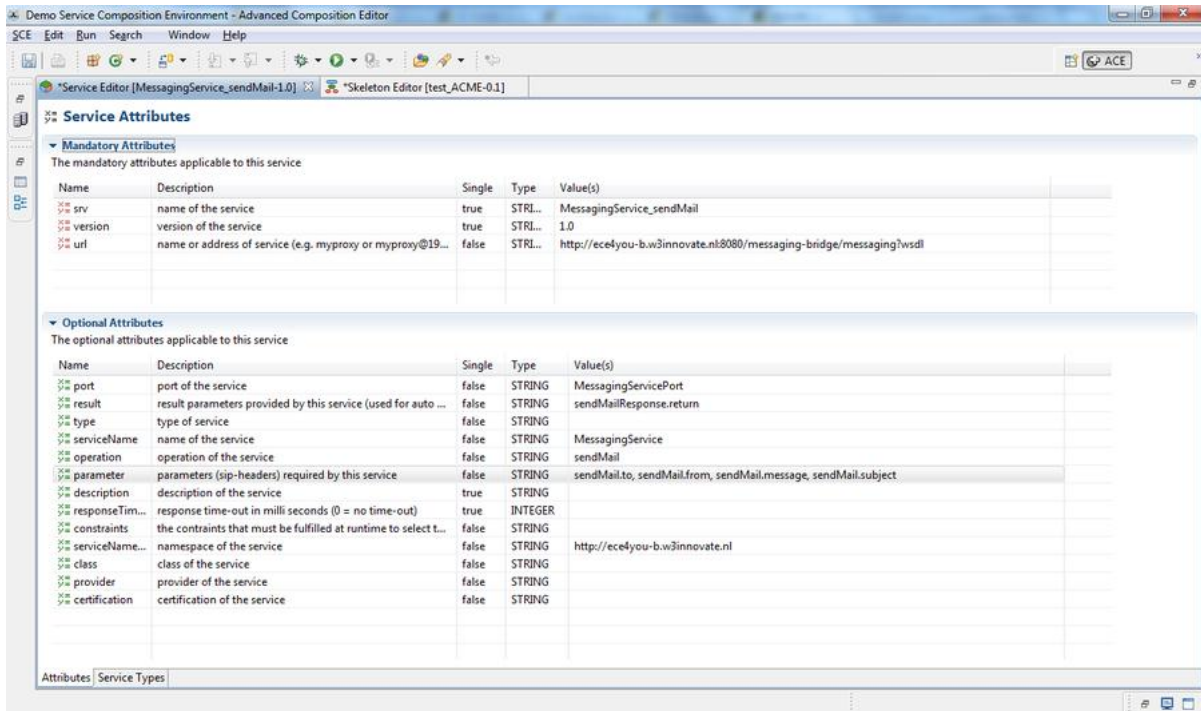
The editor allows the creation of composed services (also denoted as skeletons). The following figure shows the main graphical user interface exposed by the editor. There the user can define service descriptions (to describe how the component services can be invoked) and create compositions by adding skeleton elements and interconnecting them.
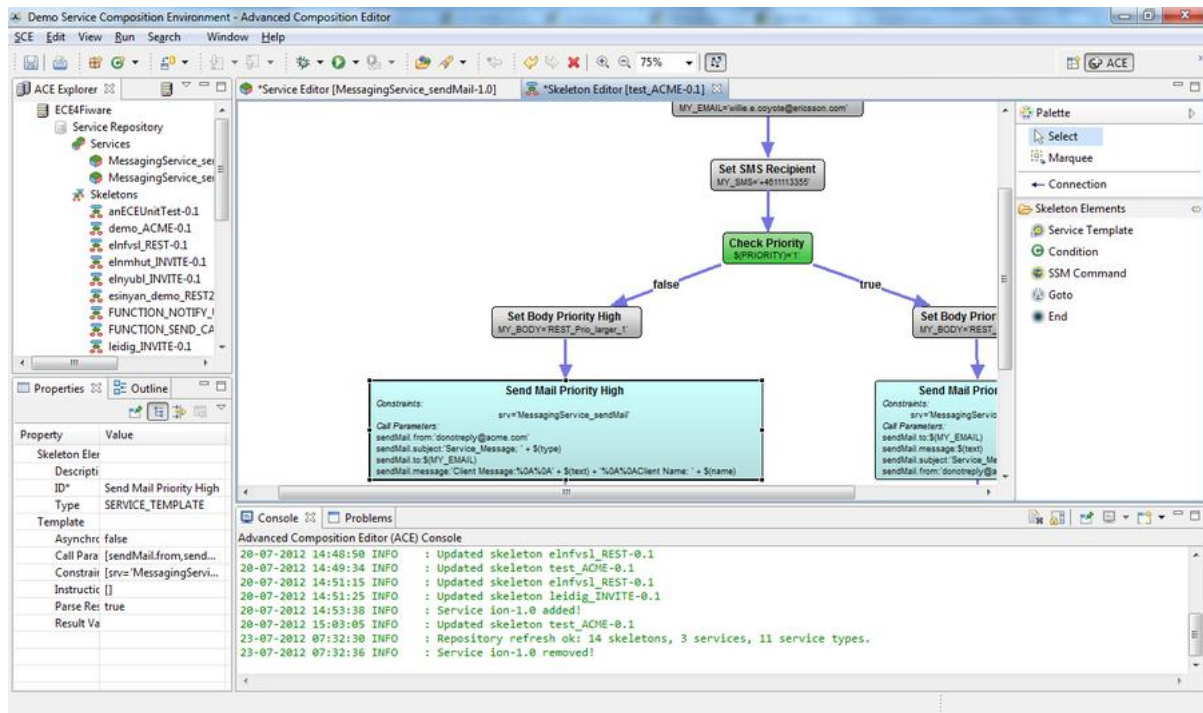
Composition Editor GUI

In order to create compositions, we need to specify in the editor how to access the services that represent the components that are invoked when executing the composition. The execution engine needs to know what are the API and the protocol used by these services, and how to set the parameters at invocation time. Thus we need to create a service description for all the services used in a composition.

When creating a service description first we need to specify what service type(s) this service is (e.g. SOAP, REST, SIP, etc). Then we need to specify values for the attributes that will be used when invoking the service (e.g. in case of a SOAP service the namespace, the port, the operation, the parameters passed to the operation, etc). The attributes to be specified could be either fixed or variable.

Composition Editor - Service Description Example

The skeletons provide the business logic, the data and control flow, and service templates (placeholders that are bound to specific service implementation during each invocation). Several services might be suitable to implement a service template. These services can have different input and output parameters and the editor needs to offer the possibility to correctly map the different dataflow types, while providing an easy way to use the unified interface. These elements are placed via drag&drop in the composition editing area and connected by linking their input and output ports. The elements can then be configured using static data or by using the relevant variables. Each skeleton consists of several interconnected elements of the following types:

- Service Template. The service template element is used during execution of the skeleton in a service composition to decide what service shall be invoked. The service description available in the local descriptions storage of the Composition Editor that fulfills the specified constraints at runtime will be selected. Only one service can be specified per service template element. Call parameters, service selection constraints, invocation semantics (synchronous or asynchronous) and the result variable (the name of the variable in which to save the response of the service) can be specified. Service selection constraints will be evaluated when the execution step arrives at this service template, thus providing late-binding for services.

- Condition. The condition element provides the possibility to branch within the skeleton upon certain conditions evaluated during runtime of the service composition execution. Different outgoing branches are supported where one outgoing branch can be connected to an unspecified default condition value which is chosen in case none of the other branches condition matches.

- SSM Command. The SSM Command element provides the option to set or remove variables in the memory space used during service composition execution. The expression to be assigned in the setVariable clause can be of a static value or a condition to be evaluated at runtime.

- Goto. The goto element provides the option to perform a jump to another skeleton element during skeleton execution. The specified jump target can either be a skeleton element from the same skeleton and thereby offers the possibility to implement a loop construct or can be any skeleton element from a different skeleton present in the advanced composition repository.

- End. Each branch of a skeleton must close with an end element.

A skeleton is created iteratively using the GUI by using a selection tool for selecting skeleton elements and interconnecting them using a connection tool. When making a connection between a Condition Element and another skeleton element, branching can be realized.
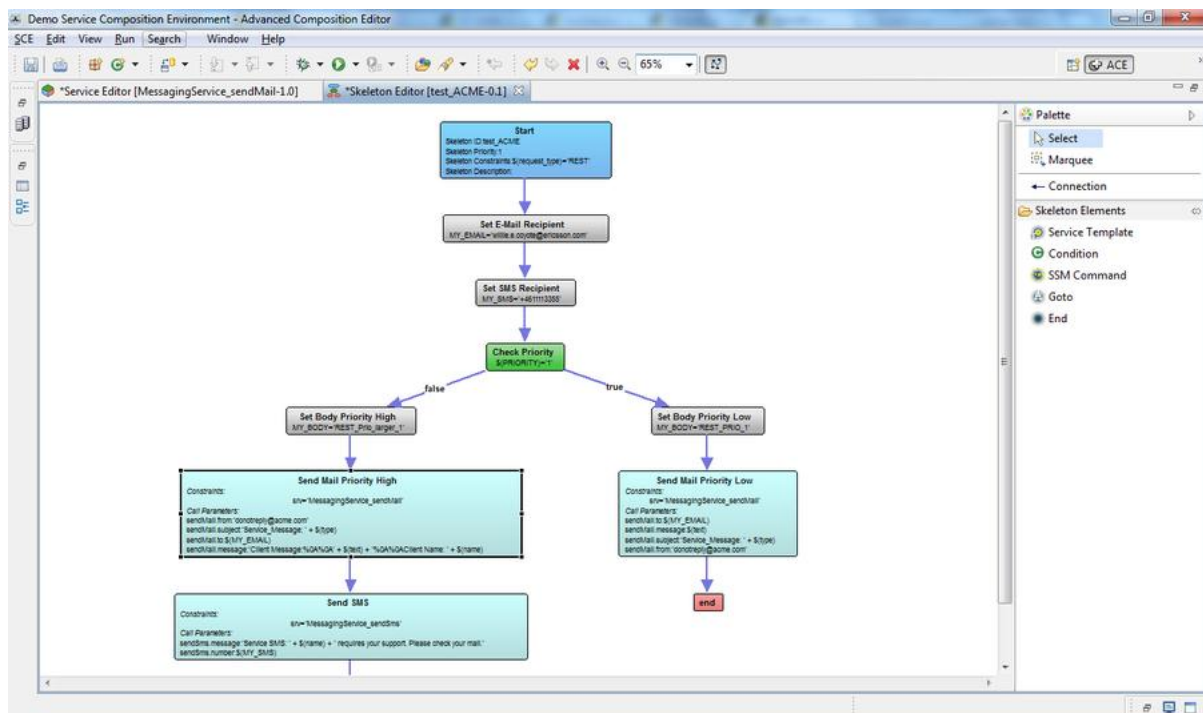
Composition Editor - Service Composition (skeleton) Example

Specification of global and local constraints are used to decide runtime service selection and event filtering. A global constraint can be specified in the skeleton start element, and is valid in the context of the composite service, for example all services used in that particular skeleton must be SIP services (syntax example: $(srv.type)='SIP'). A local constraint is to be specified within a service template. A constraint matches a service attribute against any SSM variable or literal value. Local constraints are evaluated every time the control flow of the skeleton arrives at the evaluation of a service template and invocation of a component service (syntax example: srv='ServiceName'+$(variable_name)). Note that global constraints are defining restrictions applicable to all components of the composite service, while local constraints are applied only for choosing the particular service specified by that particular template element.

Many communication-type services depend heavily on events, and first class support for events needs to be provided. External and internal events may start actions, events can be filtered, events can be triggered. Basically it is assumed that at execution all the services within a session have access to a shared state via a shared state manager (SSM). Any change in the shared state produces a high-level event related to it, e.g. change of the variable's value can generate a state change event depending on variable name, old value,

new value. These events are the only way of communication between components using different technologies. The SSM employs a subscribe/notify model.

6.6.2   Execution
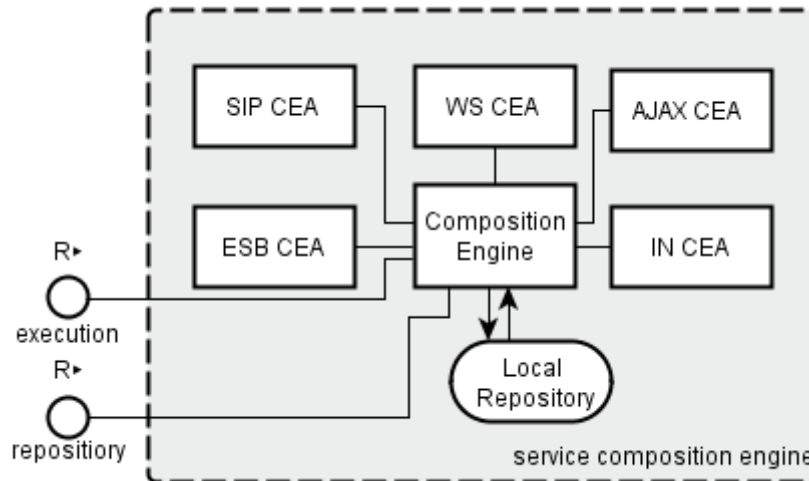
Execution Environment Architecture

Composite applications descriptions - the skeletons - are retrieved end executed by the engine. Protocol-level details related to the interaction with modules are left to the Composition Execution Agents (CEAs), which are responsible for enforcing composition decisions in the corresponding platform in a technology and protocol specific way. A shared state is used as means of mediating information between the application skeleton and the CEAs, thus coordinating the service execution. A variety of CEAs has been developed. The process is triggered by a composition execution agent (CEA) that receives a triggering event and requests the next step from the composition engine. Based on what the triggering events was, the composition engine selects the matching skeleton and creates a new session. Then, at each step it selects a suitable service that matches all the global and local constraints and serves it to the agent to execute. Execution results from the previous steps together with potential external events can influence the constraint-based decision process selecting the service for the new step. If several services are suitable to implement a certain step one of them is chosen. If a component service fails during execution, the next compatible one might be executed instead. An essential feature is the use of formal technical service descriptions for all constituent services. This service description is important for runtime service discovery, selection, and invocation. It is comprised of information about the service API and additional information used in service binding.

The local descriptions storage keeps skeletons and service descriptions used by the engine (previously careated in the editor or obtained from the Repository GE). Further the user can enable/disable the service and skeletons and access logging and tracing information.

The execution environment of the Service Composition exposes a basic life-cycle functionality for service and skeleton descriptions including import/export and enabling/disabling them to be triggered for execution.

6.7 Main Interactions

6.7.1.1 *Create Composite Services*

This functionality allows the end user to create a new composition skeleton using graphical representation for data and control flow and for service placeholders. This is the main function and will be detailed further. Note that this functionality is not available as an API to be used from other architecture components, but functionality exposed to the end user via a GUI.

- **Create/Edit Service Description**

  This function offers users the possibility to create and edit component services. When creating a service description the user specifies what service type(s) this service is (e.g. SOAP, REST, SIP, etc). Then the user specifies values for the attributes that will be used when invoking the service (e.g. in case of a SOAP service the namespace, the port, the operation, the parameters passed to the operation, etc). The attributes to be specified could be either fixed or variable. Once a service is created it is stored in the local descriptions storage. Subsequently the user can browse and edit the description of component services that can be used in the composition.

Create New Service Description
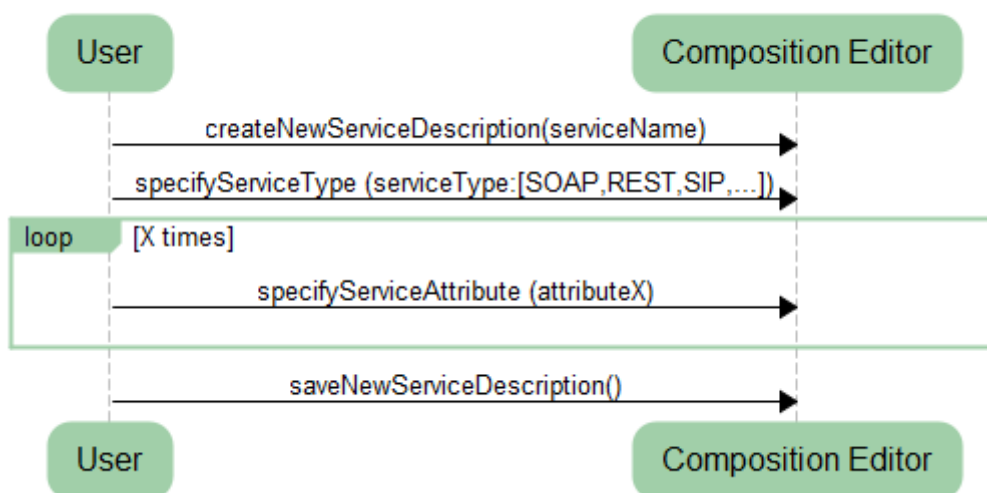
Create New Service Description

- **Create/Edit Skeletons**

  In building the composition several building blocks are added iteratively, using the GUI. The building blocks expose data flow, control, and service invocation functionality (e.g. StartElement ServiceTemplate, Condition, StateManager Command, Goto, End). Connections between skeleton elements denoting result scope and partial order may be also edited. Two types of constraints are present in the context of skeletons, the skeleton constraint and the service constraints. The service constraints are being used for selecting the appropriate service during skeleton execution upon runtime. The service constraints are mandatory in the service template element and the skeleton constraint is optional in the skeleton start element. Once a skeleton is created it is stored in the local descriptions storage. Subsequently the user can browse and edit available skeletons.

Create New Skeleton

Create New Skeleton
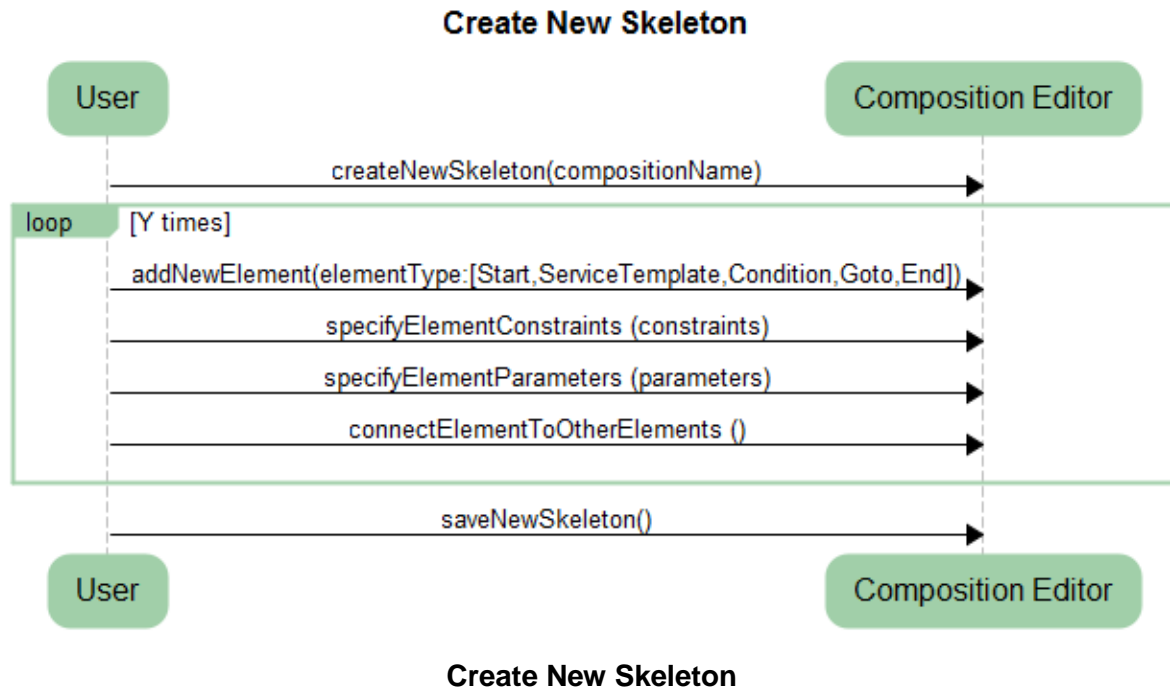
6.7.1.2   *Import Service Descriptions and Skeletons*

This operation imports (composed) service descriptions from a Repository GE to the local descriptions storage. Out of the list of available service descriptions and compositions only a subset may be selected. Note that from the perspective of the Repository and the USDL description part there is no differentiation between compositions (i.e. skeletons) and the other service descriptions. The Composition Editor however will make the difference and can edit and deploy the composed services in an execution engine to be run, while simple service descriptions can be used by the editor only as a component service in a skeleton and needs to be already up and running when the composition is triggered.

Import from Repository
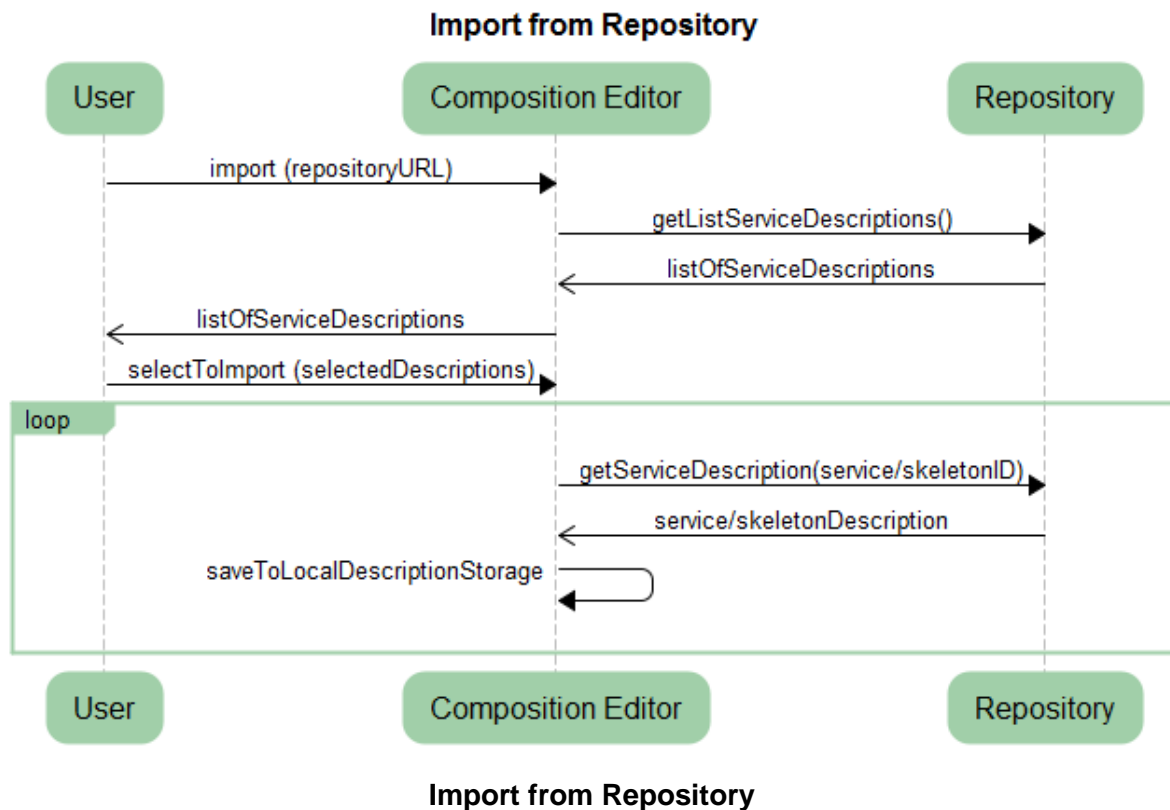
6.7.1.3   *Export Service Descriptions and Skeletons*

This operation exports (composed) service descriptions from the local descriptions storage to a Repository GE. Out of the list of available service descriptions and compositions only a subset may be selected. The description of a (composed) service may contain a USDL description providing a high level business description in addition to the technical description for the use of an execution engine. The latter may provide both the description of the API technology used for exposing/using this service and the composition skeleton that describes the runtime execution of the service in a formal composition language (if applicable).

Export to Repository

Export To Repository
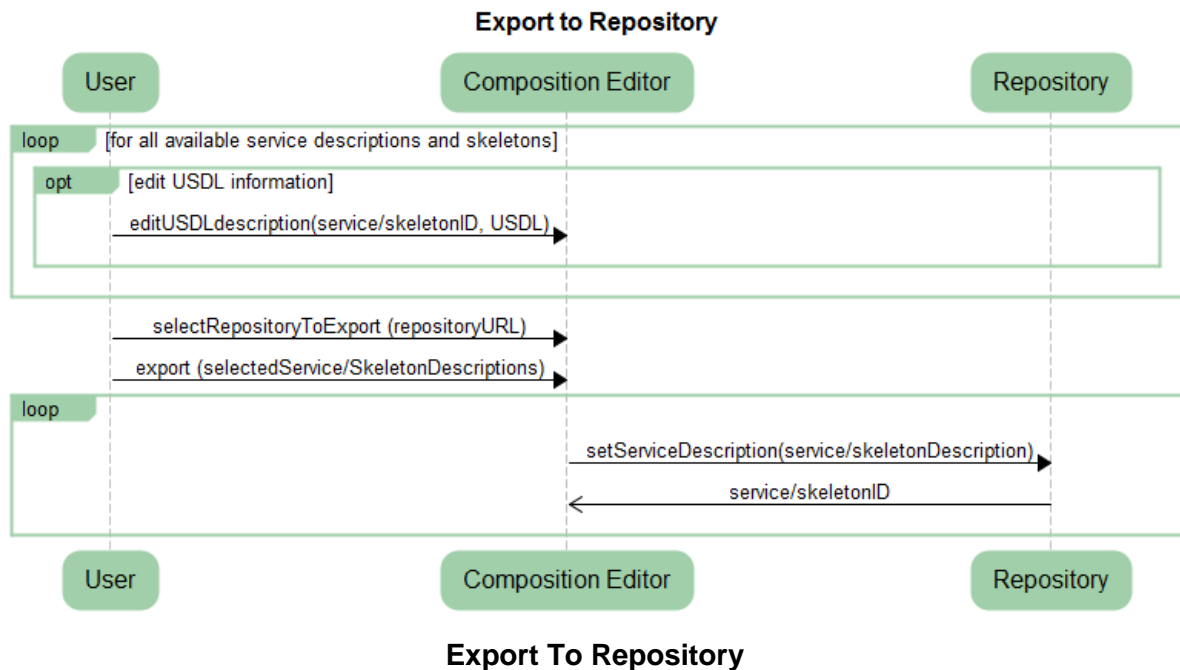
6.7.1.4 *Search Marketplace for Services*

The Composition Editor may allow end users to search for the service they need. To do so, Marketplaces need to be queried, and the editor may allow detailed query construction based on constraints on USDL and other technical service description parameters.

Search Marketplace for Services

Search in Marketplace
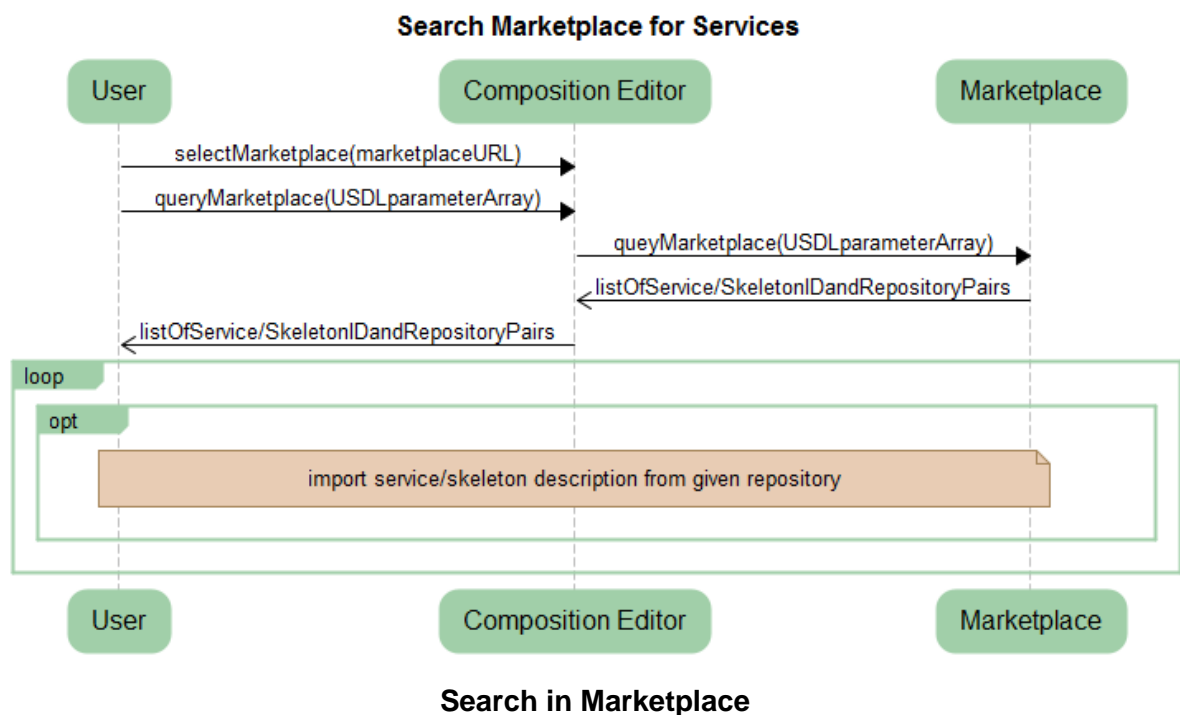
6.7.1.5 *Manage Services for Execution*

Through this UI the user may control how the service is deployed on the execution engine and specify execution parameters such as logging and tracing. Moreover the UI may be used to visualize results and additional information associated with previous runs.
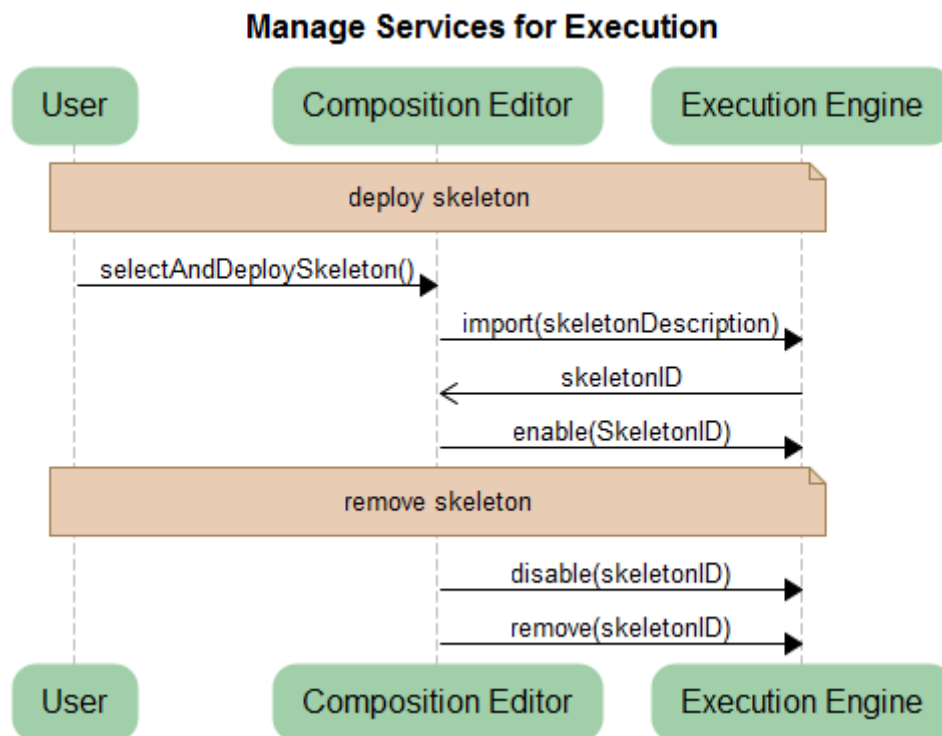
Manage Execution Example

The interactions described below are basic functionality provided by the execution environment to an external controller. The editor is using these functions and exposes them to the end user for controlling the execution of the composed services.

- **Import**

  This operation imports a service or a skeleton description (e.g. from the Repository GE) into the local descriptions storage of the execution environment.

- **Export**

  This operation exports a service or a skeleton description from the local descriptions storage (e.g. to the Repository GE).

- **Remove**

  This operation removes a service or a skeleton description from the local descriptions storage. Only disabled skeletons and services can be removed.

- **Enable**

This operation prepares a service or a skeleton description from the local descriptions storage for execution. In case of a skeleton it instantiates the composed application skeleton and makes it ready to be triggered for execution. Only enabled services will be considered during the skeleton execution.

- **Disable**

This operation removes a service or a skeleton description from the executable list.

6.8 Basic Design Principles

- **API Technology Independence**

The API abstracts from the concrete implementation technology. Implementations using various kinds of platforms and frameworks should be possible.

- **Web Browsers do not have to limit the functionality of the editor**

Modern web browsers as alternative to other GUI frameworks can and should be used fully implement the editor's capabilities.

- **User-matched interaction abstraction level**

Editors could cater for different user expertise (from technical experts with skilled in the composition language to domain experts without technical expertise or even simple end-users with no programming or technical skills) and roles (from composed service creators, to resellers and finally to prosumers) by hiding complexity behind different types of construction blocs, trading off flexibility for simplicity.

- **The specification of the Service Composition GE is not tied to a particular technology for storing (composite) service data**

The specific technology used for storing the inventory of services and their respective associations is not tied to any type of storage solutions, being opened to final implementations through SQL (MySQL, Oracle, ...) or No-SQL systems (MondoDB, Casandra, ...).

- **Service execution isolation**

Service execution does not have to interfere in the execution of other widgets.

- **Composite service exposure via different API technology**

Depending on what CEAs are available, a composed application can be exposed to the outside world via different API technologies

- **Execution engine deployment**

An execution engine implementation might be highly distributed and scalable, if intended to be used by many parties on a global scale.

6.9 Detailed Specifications

6.9.1 Open API Specifications

- FIWARE.OpenSpecification.Apps.ServiceCompositionREST

6.9.2 Other Relevant Specifications

6.10   Re-utilised Technologies/Specifications

The Composition Editor GE requires both authentication and authorization in order to safeguard the different compositions from its users.

- It is recommended to use the OAuth2 protocol: http://tools.ietf.org/html/draft-ietf-oauth-v2-30

On the other hand, Composition Editor relies on the Marketplace, Store and Repository GEs, so it must support the following technologies and specifications:

- RESTful web services

- HTTP/1.1

- JSON and XML data serialization formats

- Linked USDL

6.11   Terms and definitions

This section comprises a summary of terms and definitions introduced during the previous sections. It intends to establish a vocabulary that will be help to carry out discussions internally and with third parties (e.g., Use Case projects in the EU FP7 Future Internet PPP). For a summary of terms and definitions managed at overall FI-WARE level, please refer to FIWARE Global Terms and Definitions.

- **Aggregator (Role):** Supports domain specialists and third-parties in aggregating services and apps for new and unforeseen opportunities and needs. It does so by providing the dedicated tooling for aggregating services at different levels: UI, service operation, business process or business object levels.

- **Application:** Applications in FI-Ware are composite services that have a IT supported interaction interface (user interface). In most cases consumers do not buy the application they rather buy the right to use the application (user license).

- **Broker (Role):** The business network's central point of service access, being used to expose services from providers that are to be delivered through the Broker's service delivery functionality. The broker is the central instance for enabling monetization.

- **Business Element:** Core element of a business model, such as pricing models, revenue sharing models, promotions, SLAs, etc.

- **Business Framework:** Set of concepts and assets responsible for supporting the implementation of innovative business models in a flexible way.

- **Business Model:** Strategy and approach that defines how a particular service/application is supposed to generate revenue and profit. Therefore, a Business Model can be implemented as a set of business elements which can be combined and customized in a flexible way and in accordance to business and market requirements and other characteristics.

- **Business Process:** Set of related and structured activities producing a specific service or product, thereby achieving one or more business objectives. An operational business process clearly defines the roles and tasks of all involved parties inside an organization to achieve one specific goal.

- **Business Role:** Set of responsibilities and tasks that can be assigned to concrete

business role owners, such as a human person or a software component.

- ""*Channel: Resources through which services are accessed by end users. Examples for well-known channels are Web sites/portals, web-based brokers (like iTunes, eBay and Amazon), social networks (like Facebook, LinkedIn and MySpace), mobile channels (Android, iOS) and work centres. The mode of access to these channels is governed by technical channels like the Web, mobile devices and voice response, where each of these channels requires its own specific workflow.*

- **Channel Maker (Role):** Supports parties in creating outlets (the Channels) through which services are consumed, i.e. Web sites, social networks or mobile platforms. The Channel Maker interacts with the Broker for discovery of services during the process of creating or updating channel specifications as well as for storing channel specifications and channeled service constraints back in the Broker.

- **Composite Service (composition):** Executable composition of business back-end MACs. Common composite services are either orchestrated or choreographed. Orchestrated compositions are defined by a centralized control flow managed by a unique process that orchestrates all the interactions (according to the control flow) between the external services that participate in the composition. Choreographed compositions do not have a centralized process, thus the services participating in the composition autonomously coordinate each other according to some specified coordination rules. Backend compositions are executed in dedicated process execution engines. Target users of tools for creating Composites Services are technical users with algorithmic and process management skills.

- **Consumer (Role):** Actor who searches for and consumes particular business functionality exposed on the Web as a service/application that satisfies her own needs.

- **Desktop Environment:** Multi-channel client platform enabling users to access and use their applications and services.

- **Event-driven Composition:** Components concerned with the composition of business logic which is driven by asynchronous events. This implies run-time selection of MACs and the creation/modification of orchestration workflows based on composition logic defined at design-time and adapted to context and the state of the communication at run-time.

- **Front-end/Back-end Composition:** Front-end compositions define a front-end application as an aggregation of visual mashable application pieces (named as widgets, gadgets, portlets, etc.) and back-end services. Front-end compositions interact with end-users, in the sense that front-end compositions consume data provided by the end-users and provide data to them. Thus the frontend composition (or mashup) will have direct influence on the application look and feel; every component will add a new user interaction feature.

- Back-end compositions define a back-end business service (also known as process) as an aggregation of backend services as defined for service composition term, the end-user being oblivious to the composition process. While back-end components represent atomization of business logic and information processing, front-end components represent atomization of information presentation and user interaction.

- **Gateway (Role):** The Gateway role enables linking between separate systems and services, allowing them to exchange information in a controlled way despite different technologies and authoritative realms. A Gateway provides interoperability solutions for other applications, including data mapping as well as run-time data store-forward

and message translation. Gateway services are advertised through the Broker, allowing providers and aggregators to search for candidate gateway services for interface adaptation to particular message standards. The Mediation is the central generic enabler. Other important functionalities are eventing, dispatching, security, connectors and integration adaptors, configuration, and change propagation.

- **Hoster (Role):** Allows the various infrastructure services in cloud environments to be leveraged as part of provisioning an application in a business network. A service can be deployed onto a specific cloud using the Hoster's interface. This enables service providers to re-host services and applications from their on-premise environments to cloud-based, on-demand environments to attract new users at much lower cost.

- **Marketplace:** Part of the business framework providing means for service providers to publish their service offerings and service consumers to compare and select a specific service implementation. A marketplace can offer services from different stores and thus different service providers. The actual buying of a specific service is handled by the related service store.

- **Mashup:** Executable composition of front-end MACs. There are several kinds of mashups, depending on the technique of composition (spatial rearrangement, wiring, piping, etc.) and the MACs used. They are called application mashups when applications are composed to build new applications and services/data mash-ups if services are composed to generate new services. While composite service is a common term in backend services implementing business processes, the term 'mashup' is widely adopted when referring to Web resources (data, services and applications). Front-end compositions heavily depend on the available device environment (including the chosen presentation channels). Target users of mashup platforms are typically users without technical or programming expertise.

- **Mashable Application Component (MAC):** Functional entity able to be consumed executed or combined. Usually this applies to components that will offer not only their main behaviour but also the necessary functionality to allow further compositions with other components. It is envisioned that MACs will offer access, through applications and/or services, to any available FI-WARE resource or functionality, including gadgets, services, data sources, content, and things. Alternatively, it can be denoted as 'service component' or 'application component'.

- **Mediator:** A mediator can facilitate proper communication and interaction amongst components whenever a composed service or application is utilized. There are three major mediation area: Data Mediation (adapting syntactic and/or semantic data formats), Protocol Mediation (adapting the communication protocol), and Process Mediation (adapting the process implementing the business logic of a composed service).

- **Monetization:** Process or activity to provide a product (in this context: a service) in exchange for money. The Provider publishes certain functionality and makes it available through the Broker. The service access by the Consumer is being accounted according to the underlying business model and the resulting revenue is shared across the involved service providers.

- **Premise (Role):** On-Premise operators provide in-house or on-site solutions, which are used within a company (such as ERP) or are offered to business partners under specific terms and conditions. These systems and services are to be regarded as external and legacy to the FI-Ware platform because they do not conform to the architecture and API specifications of FI-Ware. They will only be accessible to FI-

Ware services and applications through the Gateway.

- **Prosumer:** A user role able to produce, share and consume their own products and modify/adapt products made by others.

- **Provider (Role):** Actor who publishes and offers (provides) certain business functionality on the Web through a service/application endpoint. This role also takes care of maintaining this business functionality.

- **Registry and Repository:** Generic enablers that able to store models and configuration information along with all the necessary meta-information to enable searching, social search, recommendation and browsing, so end users as well as services are able to easily find what they need.

- **Revenue Settlement:** Process of transferring the actual charges for specific service consumption from the consumer to the service provider.

- **Revenue Sharing:** Process of splitting the charges of particular service consumption between the parties providing the specific service (composition) according to a specified revenue sharing model.

- **Service:** We use the term service in a very general sense. A service is a means of delivering value to customers by facilitating outcomes customers want to achieve without the ownership of specific costs and risks. Services could be supported by IT. In this case we say that the interaction with the service provider is through a technical interface (for instance a mobile app user interface or a Web service). Applications could be seen as such IT supported Services that often ar also composite services.

- **Service Composition:** in SOA domain, a service composition is an added value service created by aggregation of existing third party services according to some predefined work and data flow. Aggregated services provide specialized business functionality on which the service composition functionality has been split down.

- **Service Delivery Framework:** Service Delivery Framework (or Service Delivery Platform (SDP)) refers to a set of components that provide service delivery functionality (such as service creation, session control & protocols) for a type of service. In the context of FI-WARE, it is defined as a set of functional building blocks and tools to (1) manage the lifecycle of software services, (2) creating new services by creating service compositions and mashups, (3) providing means for publishing services through different channels on different platforms, (4) offering marketplaces and stores for monetizing available services and (5) sharing the service revenues between the involved service providers.

- **Service Level Agreement (SLA):** A service level agreement is a legally binding and formally defined service contract between a service provider and a service consumer, specifying the contracted qualitative aspects of a specific service (e.g. performance, security, privacy, availability or redundancy). In other words, SLAs not only specify that the provider will just deliver some service, but that this service will also be delivered on time, at a given price, and with money back if the pledge is broken.

- **Service Orchestration:** in SOA domain, a service orchestration is a particular architectural choice for service composition where a central orchestrated process manages the service composition work and data flow invocations the external third party services in the order determined by the work flow. Service orchestrations are specified by suitable orchestration languages and deployed in execution engines who interpret these specifications.

- **Store:** An external component integrated with the business framework offering a set

of services that are published to a selected set of marketplaces. The store thereby holds the service portfolio of a specific service provider. In case a specific service is purchased on a service marketplace, the service store handles the actual buying of a specific service (as a financial business transaction).

- **Unified Service Description Language (USDL):** USDL is a platform-neutral language for describing services, covering a variety of service types, such as purely human services, transactional services, informational services, software components, digital media, platform services and infrastructure services. The core set of language modules offers the specification of functional and technical service properties, legal and financial aspects, service levels, interaction information and corresponding participants. USDL is offering extension points for the derivation of domain-specific service description languages by extending or changing the available language modules.

7 FIWARE OpenSpecification Apps ServiceCompositionREST

You can find the content of this chapter as well in the wiki of fi-ware.

Disclaimer: The sustainability of this Open API Specification cannot be guaranteed due to internal changes in the project consortium.

7.1 Introduction to the *Service Composition* API

Please check the FI-WARE Open Specifications Legal Notice to understand the rights to use FI-WARE Open Specifications.

7.1.1 Service Composition API Core

The execution environment of the Service Composition exposes a basic life-cycle functionality for (composed) service descriptions including import/export and enabling/disabling them to be triggered for execution. The API is a RESTful, resource-oriented API accessed via HTTP. The end user can use the editor GUI to access this functionality in conjunction with the created or managed composition specifications, however this functionality can be used also from e.g. a Store GE to automatically deploy and enable a composite service after contracting.

The editor part of the Service Composition is a tool with a graphical user interface that is used to construct new composite services. Thus it does not expose an API per se, it allows the end user to construct the composite service descriptions using the GUI, and the data structure representing the skeleton and component services is also presented.

7.1.2 Intended Audience

This specification is intended for developers of the Service Composition or external components using it. Also, users creating compositions can understand the scope of the skeleton specification. To use this information, the reader should firstly have a general understanding of the Generic Enablers for Composition and Mashup. You should also be familiar with:

- RESTful web services
- HTTP/1.1
- JSON and/or XML data serialization formats.

7.1.3 API Change History

Revision Date	Changes Summary
May 3, 2012	• Initial version
October 29, 2012	• Revised version

7.1.4   How to Read This Document

It is assumed that the reader is familiar with the REST architecture style. Within the document, some special notations are applied to differentiate some special words or concepts. The following list summarizes these special notations.

- A bold, mono-spaced font is used to represent code or logical entities, e.g., HTTP method (`GET, PUT, POST, DELETE`).

- An italic font is used to represent document titles or some other kind of special text, e.g., *URI.*

- Variables are represented between brackets, e.g. {id} and in italic font. The reader can replace the id with an appropriate value.

For a description of some terms used along this document, see [Service Composition architecture](#).

7.2   General *Service Composition* API Information

7.2.1   Resources Summary

The resources represent different skeleton and service descriptions. These can be deployed, retrieved, modified, and deleted from the execution environment. Moreover these can be enabled or disabled. Enabling a skeleton means that the service described by the skeleton can be triggered for execution. Enabling a service description means that this service can be considered as component service in the skeleton (i.e. when evaluating the service template constraints).

7.2.2   Data structure

The structure of the skeleton and services follows the model presented in the next figure. Note that constraints are specified by regular expression defining conditions under which the skeleton shall be executed during service composition. They are evaluated when the control flow reaches the
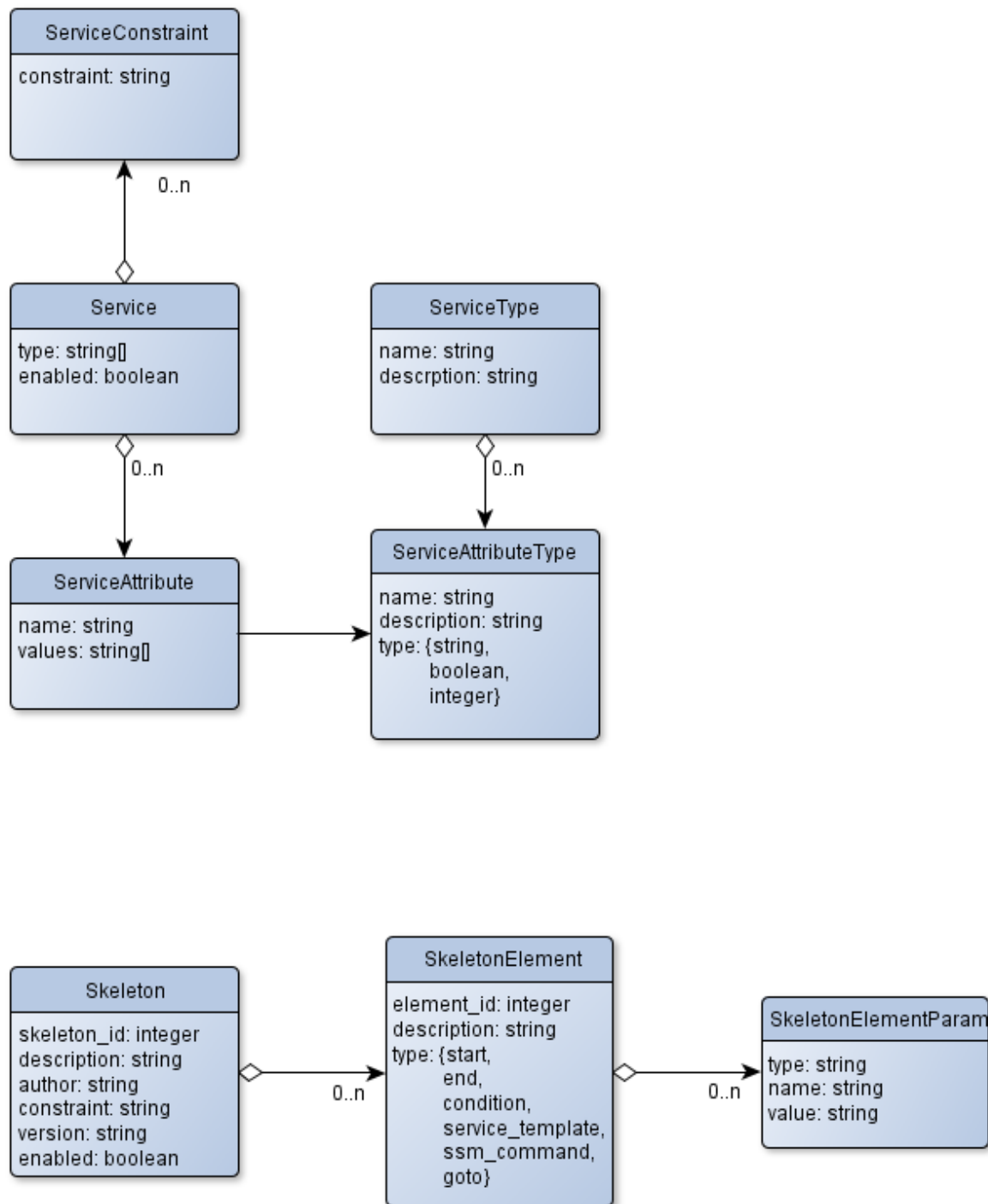
Service Composition Data Structure

Each skeleton element contains its own kind of skeleton parameters as described in the next table.

Type	Name	Value	Used In	Multiplicity

"call_parameter"	<name>	<value>	service template	0..n
"result_var"	-	<value>	service template	0..1
"constraint"	-	<value>	service template	0..n
"condition"	-	<value>	condition	0..1
"condition_case"	<case>	<next element>	condition	0..n
"goto"	-	<element reference>	goto	0..1
"ssmcommand_set"	<name>	<value>	ssm command	0..1
"ssmcommand_remove"	<name>	<value>	ssm command	0..1
"next"	-	<next element>	all (except end)	-

7.2.3   Authentication

Each HTTP request against the *Service Composition GE* requires the inclusion of specific authentication credentials. The specific implementation of this API may support multiple authentication schemes (OAuth, Basic Auth, Token) and will be determined by the specific provider that implements the GE. Some authentication schemes may require that the API operate using SSL over HTTP (HTTPS).

7.2.4   Authorization

It is assumed that access to the *Service Composition GE* is controlled by a authorization mechanisms in order to ensure that only authorized clients can read/modify/write specific information. The specification of a concrete authorization mechanism is out of scope for this document. Within the FI-WARE testbed, the authorization methods of the Security Chapter enablers will be supported by the Registry implementation.

7.2.5   Representation Format

The *Service Composition GE* API supports XML or JSON, for delivering information about services and skeletons. The request format is specified using the Content-Type header and is required for operations that have a request body. The response format can be specified in requests using the Accept header. Note that it is possible for a response to be serialized using a format different from the request (see example below).

If no Content-Type is specified, the content is delivered in the format that was chosen to upload the resource.

The interfaces should support data exchange through multiple formats:

- *text/html* - An human-readable HTML rendering of the results of the operation as output format.

- *application/json* - A JSON representation of the input and output

- *application/xml* - A XML description of the input and output.

7.2.6   Representation Transport

Resource representation is transmitted between client and server by using HTTP 1.1 protocol, as defined by IETF RFC-2616. Each time an HTTP request contains payload, a Content-Type header shall be used to specify the MIME type of wrapped representation. In addition, both client and server may use as many HTTP headers as they consider necessary.

7.2.7   Resource Identification

The skeleton descriptions are identified by the the following URI: /skeletons/{skeletonName}/{version} while the service descriptions are identified with the following URI: /services/{skeletonName}/{version}

7.2.8   Limits

We can manage the capacity of the system in order to prevent the abuse of the system through some limitations. These limitations will be configured by the operator and may differ from one implementation to other of the GE implementation.

7.2.8.1   *Rate Limits*

These limits are specified both in human readable wild-card and in regular expressions and will indicate for each HTTP verb which will be the maximum number of operations per time unit that a user can request. After each unit time the counter is initialized again. In the event a request exceeds the thresholds established for your account, a 413 HTTP response will be returned with a Retry-After header to notify the client when they can attempt to try again.

7.2.9   Extensions

The Registry could be extended in the future. At the moment, we foresee the following resource to indicate a method that will be used in order to allow the extensibility of the API. This allow the introduction of new features in the API without requiring an update of the version, for instance, or to allow the introduction of vendor specific functionality.

Verb	URI	Description
GET	/extensions	List of all available extensions

7.2.10   Faults

7.2.10.1   *Synchronous Faults*

Error codes are returned in the body of the response. The description section returns a human-readable message for displaying to end users.

Fault Element	Associated Error Codes	Expected in All Requests?
Unauthorized	403	YES
Not Found	404	YES

Limit Fault	413	YES
Internal Server error	50X	YES

7.3　API Operations

7.3.1　Importing descriptions

Imports skeletons and service descriptions into the local description storage of the execution environment. If a skeleton/service with the same name and version is available in the local description storage its description will be updated. Note that we can use the function to add an entire set of resources.

7.3.1.1　*Creating and Updating*

Verb	URI	Description
PUT	/{EntityType}/{Name}/{Version}	Create or update one resource
PUT	/{EntityType}/{Name}	Crate or update a set of resources with the same name
PUT	/{EntityType}	Create or update a set of resources

Parameters

{EntityType} can be either "services" or "skeletons"

{Name} is the name of the service or skeleton

{Version} the version of the service or skeleton

Request Body

When creating/updating a resource the request body of a PUT operation should contain the set of attributes of the entry. E.g. if content-type was "application/json":

```
{
  "{attrName1}": "attrValue1",
  "{attrName2}": "attrValue2",
  ...
}
```

When creating/updating a set of resources the request body of a PUT operation contains an array of object attributes (if we update several versions of the object) or a nested array if we update several objects with different names and versions:

```
[
  {"acme_test_skel1": [
    {"v3.7": {
      "{attrName1}": "attrValue1",
      "{attrName2}": "attrValue2",
      ...
      }
    },
    {"v4.2": {
      ...
      }
    },
    ...
    ]
  },
  {"acme_test_skel2": [
    ...
    ]
  },
  ...
]
```

Status Codes

201 Created

> The request has been fulfilled and resulted in a new resource being created.

204 No Content

> The server successfully processed the request, but is not returning any content.

400 Bad Request

> The request cannot be fulfilled due to bad syntax.

404 Not Found

> The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible.

409 Conflict

> Indicates that the request could not be processed because of conflict in the request, such as an edit conflict.

500 Internal Server Error

> A generic error message, given when no more specific message is suitable.

7.3.2   Exporting descriptions

Exports skeletons and service descriptions from the local description storage of the execution environment. Note that we can export also sets of resources, even distinguished by specific attribute name-value pairs.

7.3.2.1   *Reading*

Verb	URI	Description
GET	/{EntityType}/{Name}/{Version}	Read one resource
GET	/{EntityType}/{Name}	Read a set of resources with the same name
GET	/{EntityType}	Read a set of either service or skeleton descriptions
GET	/{EntityType}?{attrName}={attrValue}	Read a set of resources with specific attribute name-value pairs

Parameters

{EntityType} can be either "services" or "skeletons"

{Name} is the name of the service or skeleton

{Version} the version of the service or skeleton

Examples

GET /skeletons/ACME_test_skel1/v3.7

Returns the description of a specific skeleton.

GET /skeletons

Returns the description of all skeletons.

GET /services?serviceType=WSDL

Returns the descriptions of all services where serviceType is WSDL.

Result Format

The result format is similar to the request body examples given when importing descriptions.

Status Codes

200 OK

  The request was handled successfully and transmitted in response message.

400 Bad Request

  The request cannot be fulfilled due to bad syntax.

404 Not Found

  The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible.

500 Internal Server Error

  A generic error message, given when no more specific message is suitable.

7.3.3   Removing descriptions

Removes skeletons and service descriptions from the local description storage of the execution environment. Note that we can remove also sets of resources, even distinguished by specific attribute name-value pairs. Only services and skeletons that are not enabled can be removed.

7.3.3.1 *Deleting*

Verb	URI	Description
DELETE	/{EntityType}/{Name}/{Version}	Delete a specific resource
DELETE	/{EntityType}/{Name}	Delete a set of resources with the same name
DELETE	/{EntityType}	Delete a set of either service or skeleton descriptions
DELETE	/{EntityType}?{attrName}={attrValue}	Delete a set of resources with specific attribute name-value pairs

Parameters

{EntityType} can be either "services" or "skeletons"

{Name} is the name of the service or skeleton

{Version} the version of the service or skeleton

Status Codes

200 OK

> The request was handled successfully and transmitted in response message.

400 Bad Request

> The request cannot be fulfilled due to bad syntax.

404 Not Found

> The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible.

500 Internal Server Error

> A generic error message, given when no more specific message is suitable.

7.3.4   Enabling, disabling and checking enabled status

If a skeleton is set to enabled, it instantiates the composed application skeleton, exposes its interface and makes it ready to be triggered for execution. Only enabled skeletons will be considered during the skeleton execution. Note that by enabling a certain skeletons for execution, the GE automatically exposes the API of these compositions towards the external world, and any access to this API will trigger the execution of the respective skeleton. This is an API exposed by the execution environment, however it is created at runtime in the form specified by the composed service.

If a service is set to enabled it is considered as a potential candidate to match a "service template" when executing the skeleton.

By default all imported services and skeleton are considered disabled, and have to be enabled for use.

When disabling a skeleton the result may not be immediate if the composed service it represents is in use. Thus we need to check the enablement status before trying to remove a

7.3.4.1   *Operations*

Verb	URI	Description
PUT	/{EntityType}/{Name}/{Version}/enabled	Enable a certain service or skeleton
DELETE	/{EntityType}/{Name}/{Version}/enabled	Disable a certain service or skeleton
GET	/{EntityType}/{Name}/{Version}/enabled	Check if a certain service or skeleton is enabled

Parameters

{EntityType} can be either "services" or "skeletons"

{Name} is the name of the service or skeleton

{Version} the version of the service or skeleton

Request/Response Body

The request ore response body should be empty.

Status Codes

200 OK

> The GET/DELETE request has been fulfilled. In case of GET it signals that the object is enabled.

201 Created

> The PUT has been fulfilled and object is enabled.

202 Accepted

> The request has been accepted for processing, but the processing has not been completed. Can appear when disabling a skeleton in use.

400 Bad Request

> The request cannot be fulfilled due to bad syntax.

404 Not Found

> The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible. In case of get it signals that the object is disabled.

409 Conflict

> Indicates that the request could not be processed because of conflict in the request, such as an edit conflict.

500 Internal Server Error

> A generic error message, given when no more specific message is suitable.

8 FIWARE OpenSpecification Apps LightSemanticComposition

You can find the content of this chapter as well in the wiki of fi-ware.

Name	FIWARE.OpenSpecification.Apps.Light-weighted Semantic-enabled Composition
Chapter	Apps,
Catalogue-Link to Implementation	Light-weighted Semantic-enabled Composition - COMPEL
Owner	ATOS, Jesús Gorroñogoitia

8.1 Preface

Within this document you find a self-contained open specification of a FI-WARE generic enabler, please consult as well the FI-WARE_Product_Vision, the website on http://www.fi-ware.eu and similar pages in order to understand the complete context of the FI-WARE project.

8.2 Copyright

- Copyright © 2012 by ATOS

8.3 Legal Notice

Please check the following Legal Notice to understand the rights to use these specifications. Atos Spain strives to make the specifications of this Generic Enabler available under IPR rules that allow for a exploitation and sustainable usage both in Open Source as well as proprietary, closed source products to maximize adoption.

8.4 Overview

Light-weighted Semantic-enabled Composition is a tooling suite that aims at simplifying the development of domain specific business process as service compositions by exploiting the full potential of semantic technologies. Development of business process in BPM realm requires multi-disciplinary teams that cope with the domain specific knowledge about processes, entities, roles, etc and the ICT technology required to implement them. In particular the number and complexity of SOA-related technologies hamper the implementation of business processes as aggregations of open and Internet accessible services.

In this sense, it is desirable that we simplify the access to these composition technologies to the domain specific experts, as unskilled end users. This can be achieved when the

technology itself is capable of producing service compositions without requiring complex information structures, which otherwise can be obtained from end users in a more human readable format.

8.4.1   Target usage

Light-weighted Semantic-enabled Composition is addressing situations where domain specific business processes require to be implemented as service compositions within a certain organization. Business processes are decomposed into different tasks, each one executed by an external service (some provided within the same organization, but others provided by third parties). Commonly, designing and implementing a business process as a service composition requires multidisciplinary teams, ranging from domain business experts (as modelers) to service engineers and integrators. The complexity of these teams, in terms of expertise is required by the modeling activity itself, since modeling a business process as a service composition requires of:

- Business analysts with knowledge on the concrete domain concepts, entities, agents, procedures, etc.

- Business modeling experts, with knowledge on BPM languages, methodologies and practices.

- Service engineers acting either as service providers and/or consumers, who provide or consume services on the domain, and exploit the technological infrastructure that make them possible.

- Service Integrators, who orchestrate services into compositions that offers more complex functionality by aggregating composable functionality.

The involvement of those multidisciplinary teams, their associated procedures and methodologies and required service composition technologies make service composition a complex, time consuming and prone to error activity.

A typical scenario of service composition to reify a business process is as follows. The business analyst describes the domain specific scenario, the required business process and related concepts, entities, roles, etc. The business modeler creates a business process model which is iteratively refined by interacting with the business analyst. Ideally this activity can be performed by the same role. The business process model is implemented as service composition by the service integrator, who aggregates services provisioned by service providers. Commonly, iteratively interactions between the business modeler and the service integrator are conducted to refine and amend the service composition implementation of the business process.

This GE aims at simplifying the implementation of business processes as service compositions, by enabling business modelers to create directly service compositions without requiring the involvement of service integrators. In this way, the business modelers describes the service composition by using domain specific languages (DSL) or vocabularies, in particular ontologies to describe the semantics of the tasks that the composition is decomposed on.

The technical activities conducted by the service integrator are semi-automatically performed by the GE. That is, the GE does not prompt the business modeler to provide technical information required to implement the executable service composition, but the GE obtains that information by exploiting the semantic descriptions attached to the service composition by the modeler from the domain specific ontologies. In this way, a full executable service composition is created without requiring a technical expertise on BPM techniques.

8.4.1.1 *User roles*

For the purpose of this description, two main separate roles are identified:

- Domain business modelers, who has knowledge about the domain within where the business process as service composition will be executed. Therefore, they have the required domain specific knowledge to model the business process as service composition by using domain specific vocabularies (or ontologies)

- Service providers, who will deploy and host the service composition modeler by the business modeler, within the execution environment. They will also take care of composition management at runtime.

8.5 Basic Concepts

This section introduces the most relevant functional concepts considered in the Light-weighted Semantic-enabled Composition GE, which are related to the functional components described in the next section about GE architecture.

Light-weighted Semantic-enabled Composition GE is a tooling suite supporting the implementation of domain business processes as service compositions, executed in backend execution environments as other SOA services. Conceptually, this approach can be decomposed in the following main concepts:

8.5.1 BPMN Composition Edition

A domain business process is decomposed into single working units or tasks, connected logically by a work flow and a data flow, according to the BPMN specification . Each task is performed by an external service, provided by either the same composition provider (same organization) or by an external third party, accessible through open Web standards.

8.5.2 Light-weighted Semantic-enabled Composition

This approach for the modeling of a service composition describes each composition task using semantic descriptions according to some standardized semantic schema (WSMO/OWL-S) and in particular to some light versions (WSMOLite /MicroWSMO). A practical procedure is to annotate each task with concepts selected from a domain specific ontology. These annotations constitute the semantic goal description of the task, that is, the intended purpose of the task. Task goal is matched against the available semantic service descriptions within a semantic knowledge base repository and the matching services are return ranked so the business modeler selects one as task binding or leaves the system to select the best scored.

8.5.3 Semi-automatic Execution Composition Generation

A semantically annotated service composition with tasks bound cannot be executed. A complete service composition including the technical bindings and the data flow mappings needs to be generated out of the semantically annotated composition

8.5.4   Composition Deployment and Execution

The composition is ready to be consumed as soon as the service integrator takes the generated executable service composition, selects a target execution environment and deploys the composition into the target.

8.6   Light-weighted   Semantic-enabled   Composition Architecture

Light-weighted Semantic-enabled Composition GE architecture is depicted in next figure. This architecture is split into two main functional layers: design and execution.
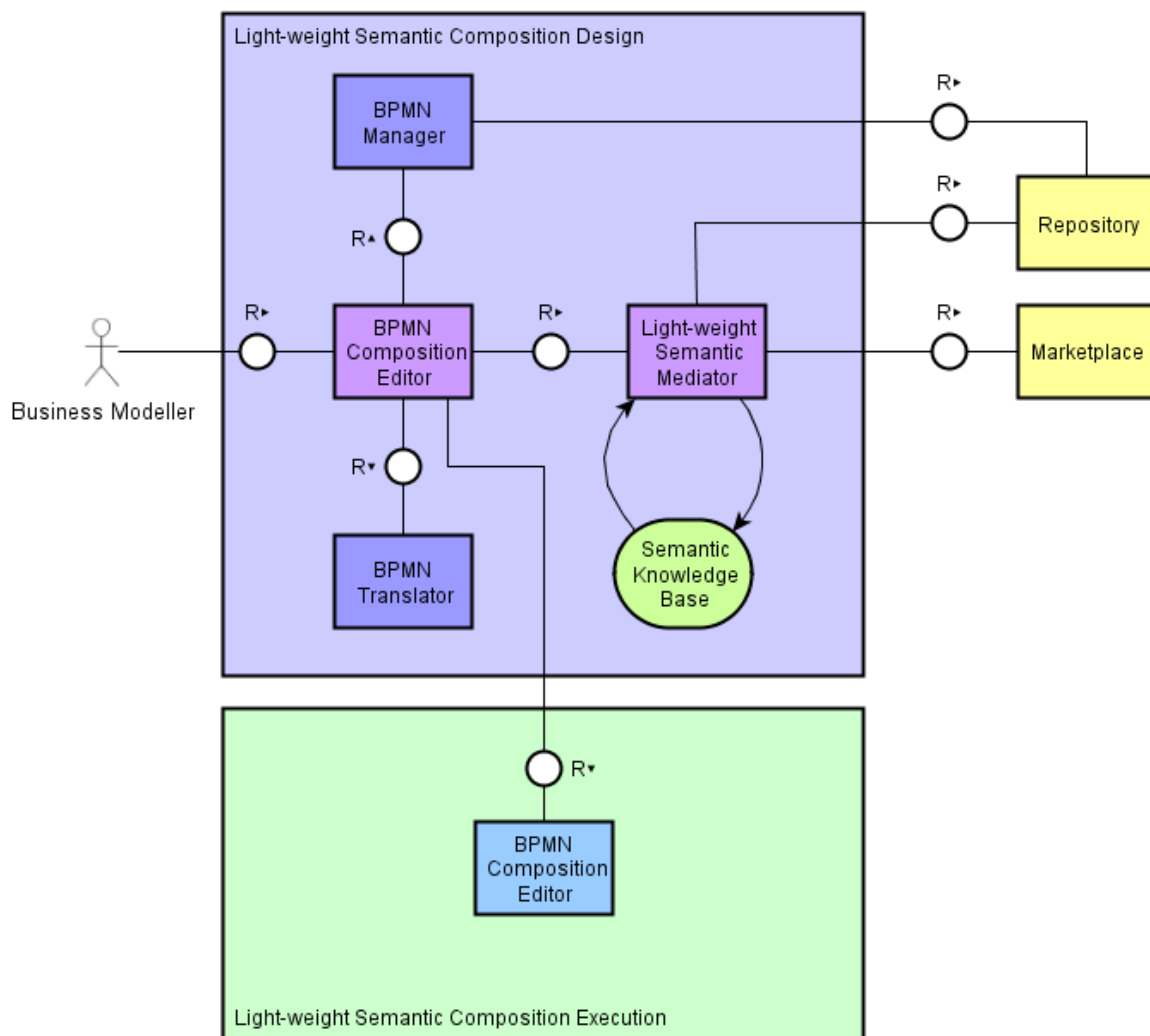
Light-weighted Semantic-enabled Composition GE Architecture

The design layer provides functional support for the modeling of service compositions. This GE constrains service composition modeling to use BPMN 2.0 for both the graphical (modeling) and execution semantics. That is, this GE assumes that service compositions models are instances of the BPMN 2.0 standard metamodel.

The following functional components are part of the design layer of this GE:

- BPMN Composition Editor enables business modelers to create service compositions using the BPMN 2.0 Graphical notation and execution semantics. It provides typical GUI to create, edit and manage service compositions, requiring the business modeler to have some background in BPM modeling.

- Light-weighted Semantic Mediator complements the BPMN Composition Editor by adding additional semantic-enabled modeling aids that simplifies the modeling process. These assisting features allow business modelers to describe the composition tasks, bind matching services, generate the data flow mapping, and so on.

- Semantic Knowledge Base complements the Light-weighted Semantic Mediator with a semantic repository of semantic service descriptions, domain specific vocabularies (DSL, ontologies), service composition annotations (tasks, compositions themselves), etc. This component provides content access features, including querying and reasoning.

- BPMN Manager provides BPMN model management, including features to validate and complete BPMN models with required executable information (service bindings, data flow mappings, etc)

- BPMN Translator provides translation capabilities to other executable composition formats, such as BPEL 1.2/2.0

- Composition Deployer acts as a proxy between the Light-weighted Semantic-enabled Composition layer and the Composition Execution layer. It manages the service composition deployment process into the selected Composition Execution layer.

Components in the design layer interact with some other GE components such as the Marketplace and Repository:

- Marketplace is used to query and retrieve USDL service descriptions for those services matched to composition tasks through semantic matchmaking, thanks to links contained within their semantic descriptions.

- Repository contains the technical descriptions (i.e. WSDL , WADL , etc) of services aggregated in the composition and the composition themselves: the composition models (BPMN) and their technical descriptions (WSDL, WADL, etc)

The Composition Execution layer contains the Composition Execution component, which deploys, enables and executes the service compositions, upon remote invocation by a service consumer.

Main interactions performed by the components that comprise this GE are described in next section, grouped by a functional classification

8.7   Main Operations

A functional classification of Light-weighted Semantic-enabled Composition GE main features is depicted in the next UML use case diagram, and described in more detail in next paragraphs.

Light-weighted Semantic-enabled Composition GE functional classification

8.7.1 Model Composition

This GE provides support for service composition modeling, assuming BPMN 2.0 as graphical notation and execution semantics. Through the BPMN Composition Editor, business modelers can create or open composition models, modify them (edit) and manage them (save, delete). Composition models are stored within a Repository.

Composition edition also includes support for create/update/delete features for composition elements, such as service tasks, gateways (exclusive, parallel), flows and events (start, end).

Composition editor allows to select the composition itself or concrete composition elements.

8.7.2 Prepare DSL/Semantics

In this GE approach, the business modelers describe composition models and their elements by annotating them with concepts taken from concrete domain specific languages DSL (or vocabularies) which provide concrete semantics. From operational point of view, it is common to use ontologies as DSL or vocabularies. The Light-weighted Semantic Mediator enables the business modeler to:

- Register new DSL/Ontologies within it

- Select a concrete DSL/Ontology for a given domain modeling context. Select a concrete DSL/Ontology concept within the domain ontology. These concepts are used to annotate and describe a composition model and their elements.

8.7.3 Describe Model Composition/Task using DSL/Semantics (Business modelers)

The Light-weighted Semantic Mediator enables the business modeler to describe the composition model and its elements using semantic annotations.

In the scope of a composition task, the annotations constitute a description of the task. In other words, they describe the goal of the task. This goal will be used in the service matchmaking process to look for services whose semantic description will match it. A semantic task description is constituted by several annotations of certain type according to the semantic schema used to represent the goal (i.e MSM)

In the scope of the composition itself, the annotations constitute a description of the global composition requirements, preferences and contextual information.

8.7.4 Describe Service using DSL/Semantics (Service Providers)

Light-weighted Semantic-enabled Service Composition GE approach assumes that composable services are described using light semantics. Those semantic service descriptions are available within the Semantic Knowledge Base, and are provided by service providers. A service composition created by applying this GE approach is a service by its own, whereby the business modeler, acting as service provider, is required to provide this semantic description. Same applies to any other third party service intented to be composed by others.

The Light-weighted Semantic Mediator enables service providers to create semantic descriptions compliant to the semantic schema used by the complete GE solution. The concrete schema is left for the implementation, but it should be consistent along with all components that use it.

The schema includes links to the business oriented description stored in the Marketplace, and the technical description stored in the Repository.

8.7.5   Task binding

One of the main jobs in service composition modeling is to bind every task the composition is divided out to a matching service that performs the task. A business modeler can conduct this task binding per task or for the whole composition. The Light-weighted Semantic Mediator enables the modeler to discover matching services based on task goal criteria, rank them according to preferences or non-functional requirements (NFR) and select one service, which is bound to the task. Those activities are typically performed by querying the Semantic Knowledge Base.

8.7.6   Validate, generate, translate executable BPMN composition model

Next step in service composition modeling consists on filling the missing information that the composition model requires before being shipped for deployment and execution. Examples of missing information are:

- Task binding technical description. For each BPMN 2.0 service task, a concrete task binding information has to be included, by inspecting the technical description (i.e. WSDL)

- Data flow mapping, including IO mappings at task and composition level

Once the service composition model has been completed with missing required executable information, the composition model is validated (BPMN 2.0 compliance validation) and serialized (for storage and deployment).

Optionally, the composition model can be translated from its original BPMN 2.0 format to another mappable format, such as BPEL 1.2/2.0 . This is required when the select target environment for execution is not BPMN 2.0 compatible.

8.7.7   Deploy composition model

Full executable validated composition models can be deployed into the selected target Composition Execution environment, using the Composition Deployer. Once deployed, the service composition is enabled, being ready to received incoming requests from service consumers.

Similarly, deployed service compositions can be undeployed anytime.

8.7.8   Composition Execution

During the execution time, deployed services can be enabled or disabled any time through the Composition Execution UI. Besides, running compositions (enabled) can be continuously monitored and monitoring data can be collected for given time frames.

Next paragraphs detail main operations using UML sequence diagrams for the most relevant scenarios concerning the light-weighted semantic modeling of a service composition.

8.7.9   Modeling a BPMN Composition

A business modeler, through the BPMN Composition Editor, can either:

- Create a new BPMN composition model

or

- Open an existing one from the Repository, using a unique model Id. The unique model id is provided by the Repository during the model saving.

The business modeler can work on this composition model, editing the model and its elements. This activity includes create/edit/delete operations on model elements such as tasks, gateways, flows, events, etc. Each model element is uniquely identify by a unique identifier (within the model) provided by the editor upon creation.

Anytime, during the modeling of the composition, the business modeler can either:

- Save the composition model into the Repository. This process requires to serialize the BPMN composition model according to the BPMN 2.0 serialization standard (XSD/XMI)

- Delete the model from the Editor and Repository (in case the model was previously saved). Models are identified within the repository by a unique identifier.
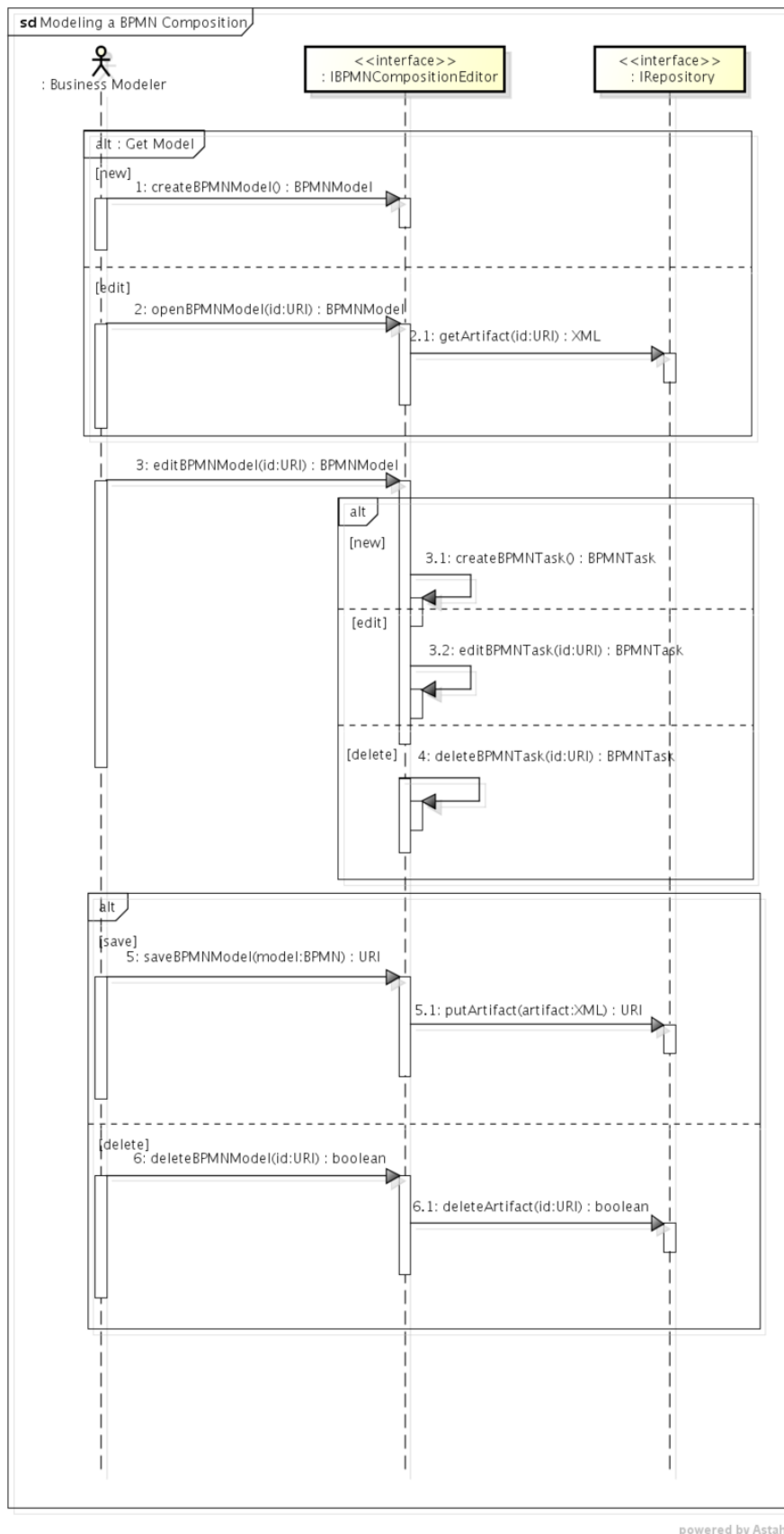
Modeling a BPMN Composition

8.7.10   Modeling a composition using light-weighted semantics

A composition work-flow and its task decomposition can be modeled by using the BPMN Composition Editor itself, but once each task has to be bound to concrete services or the data flow mapping has to be designed, is when the light-weighted semantic composition approach comes up.

This GE encourage modelers to describe composition task by attaching light-semantic annotations, according to some pre-established task goal schema (for instance WSMOLight/MicroWSMO) which are taken from domain specific ontologies (or any other domain specific language (DSL) or vocabulary). Based on this semantic task goal, the semantic matchmaking activity determines the best matching service and bounds the task to it.

Next UML sequence diagram describes the overall process in detailed, decomposed by operation and involved component.
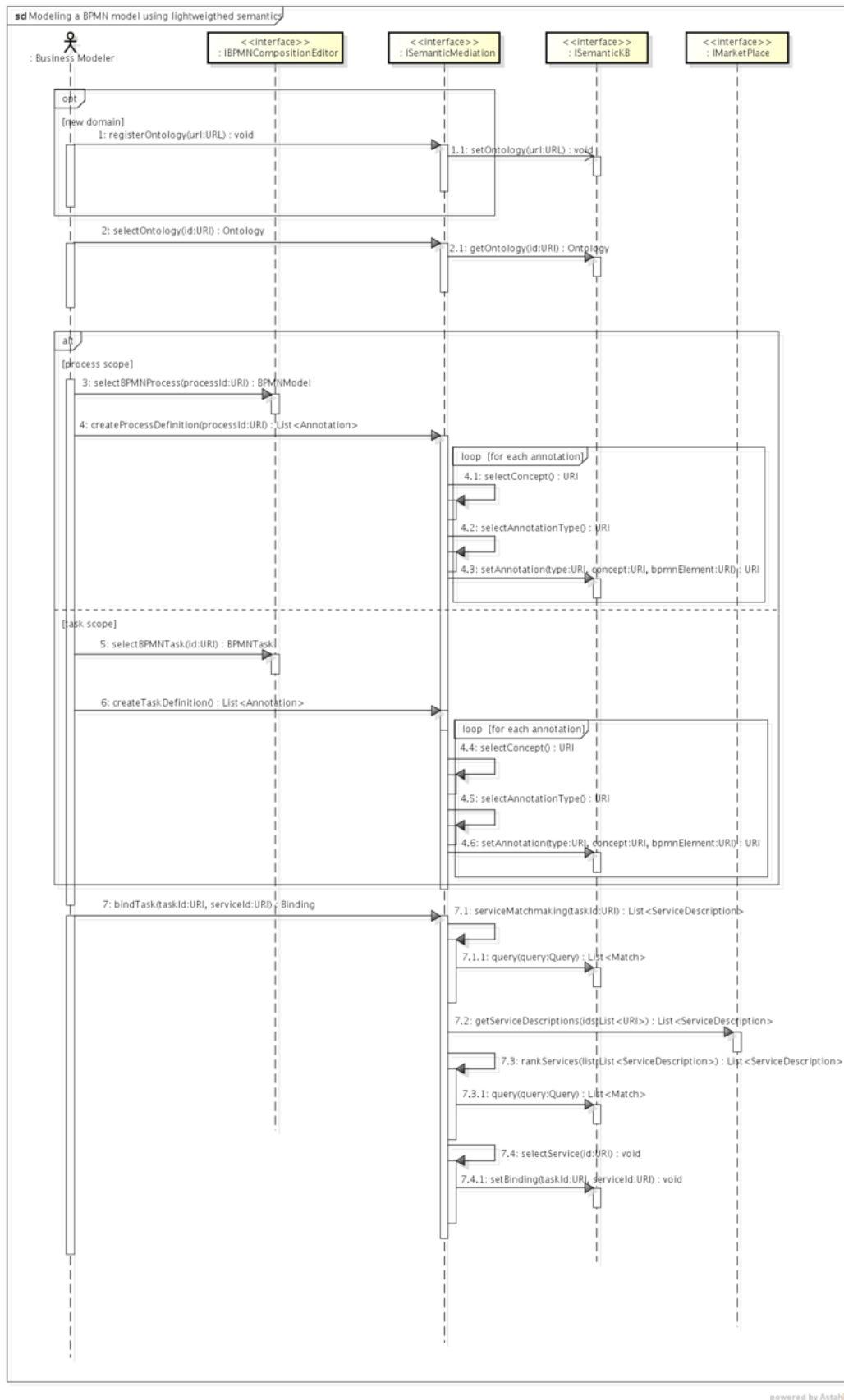
Modeling a composition using light-weighted semantics

8.7.10.1 *Domain Ontologies preparation phase*

A business modeler selects the suitable domain specific ontology that will be used to describe the composition tasks, depending on the concrete context the composition will be applied.

If the domain ontology has not being previously registered within the Light-weighted Semantic Mediator, the modeler registers it by giving the ontology URL. The ontology is downloaded from that URL and stored within the Semantic Knowledge Base. The ontology has to be accessible in a compatible format with the Semantic Knowledge Base (i.e. serialization format such as RDF/XML , N3 , etc

Any time during the composition modeling the business modeler can switch from one domain ontology to another by selecting them in the Light-weighted Semantic Mediator. The selected ontology is loaded from the Semantic Knowledge Base. Ontologies are uniquely identified within the GE implementation by an URI.

8.7.10.2 *Lightweighted semantic composition modeling*

A business modeler can start this activity either by annotating the composition itself (global annotations) or concrete composition elements (tasks in particular).

Global annotations describe the composition global requirements, preferences and contextual constrains, as stated by the selected semantic annotation schema (i.e. WSMOLite/MSM). Each annotation (concretised by an selected ontology concept, given its URI and its type) is stored within the Semantic Knowledge Base, given the annotation type, annotation concept and process URI.

Tasks are light-weighted sematically annotated to describe them, according to the selected semantic annotation schema (i.e. WSMOLite/MSM) as well. Each annotation is stored within the Semantic Knowledge Base, given the annotation type, annotation concept and task URI.

Once the business modeler has described all the tasks within the model using light-weighted sematic annotations, he proceeds to bind each task to a concrete external service. This can be done automatically for the whole composition, whereby all tasks are automatically bound the the best ranked compatible service matched by the matchmaking process, or semiautomatically, task by task where selection is conducted by the business modeler. Nonetheless, this process either manual or automatic is similar.

The Light-weighted Semantic Mediator is invoked to search for services whose semantic descriptions are matching the task goal description. Using this local task goal description, a semantic query (i.e. SPARQL) is prepared and sent to the Semantic Knowledge Base, which returns a list of unranked matches (candidate service descriptions). Base on global annotations (requirements, preferences and context constrains), candidate services are filtered out (according to requirements and context contraints) and ranked (based on preferences), by querying and reasoning the Semantic Knowledge Base.

Finally the best ranked candidate service is automatically selected or the business modeler selects one by inspectind the service candidate list, using descriptions obtained from the Marketplace. Selected service is bound to the task in the Semantic Knowledge Base.

8.7.11 Process a complete executable BPMN composition model

A semantically fully processed composition model, once its tasks have been bound to semantically annotated services, contains all the information required to create an executable composition model.

This procedure is initiated by the service provider through the Light-weighted Semantic Composition Editor, by invoking to process the BPMN model. This procedure conducts two main jobs for each composition task:

- Task binding is included in the BPMN composition model, according to the BPMN specification, by getting the technical required information from the technical description stored within the Repository

- Task data flow mapping (IO mapping) is included in the BPMN composition model, according to the BPMN specification as well. Data flow mapping considers all data objects available before reaching this task following the work flow.

Once the BPMN executable composition model has been created, it is validated against the BPMN specification.

Optionally, the BPMN executable composition model can be converted into another executable model, compliant to another composition language, such as BPEL 1.2/2.0. This could be required by the target Composition Execution Environment.

Finally, the executable composition model is serialized into XML or other standardized serialization schema for interchange as determined by the selected target execution language (BPMN XSD/XMI, BPEL XSD, etc)
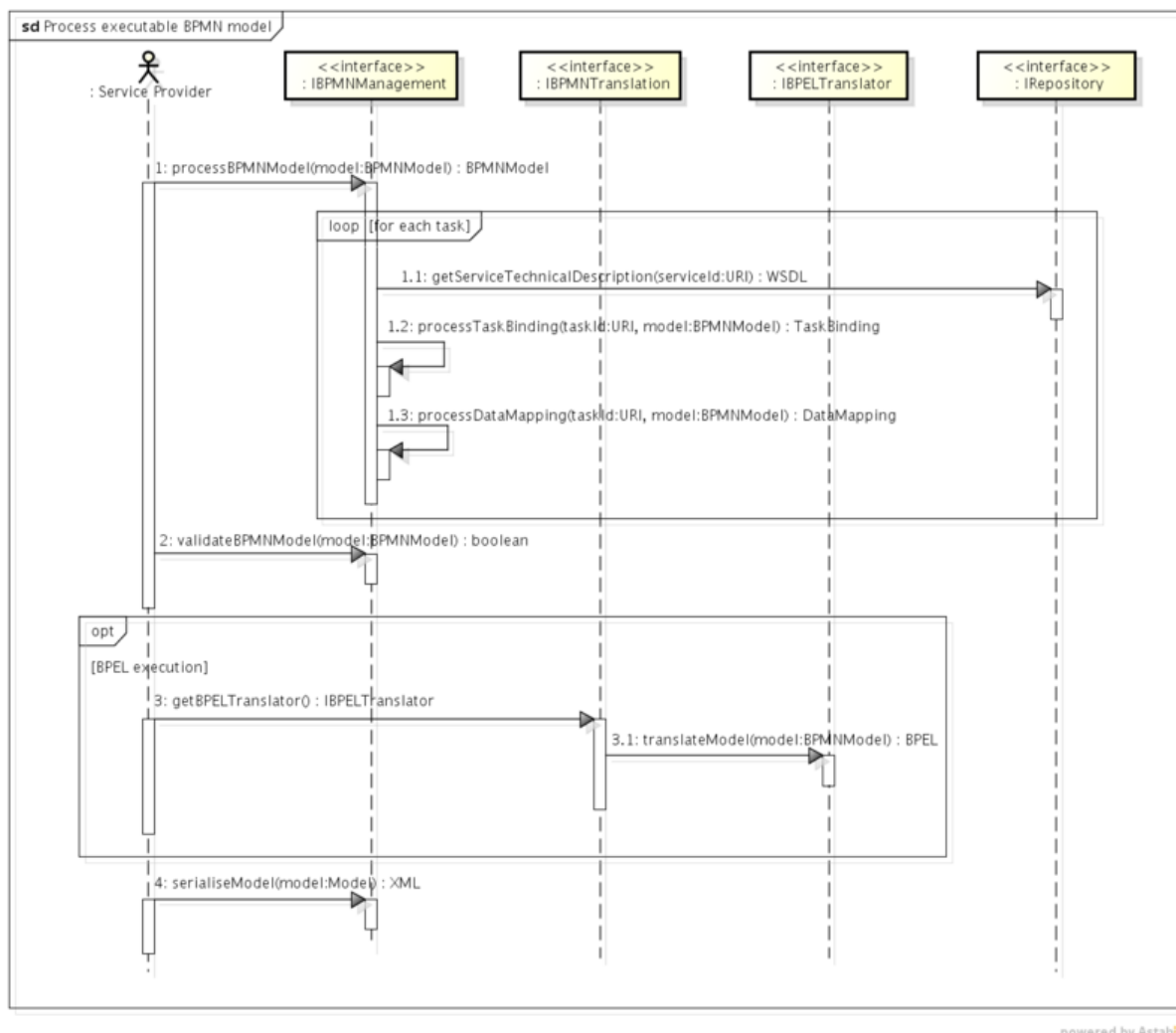
Process a complete executable BPMN composition model

8.7.12 Deploy a service composition model

A service composition model can be deployed within a selected target. A service provider selects the target execution environment and deploys the current composition model edited in the Light-weighted Semantic Composition Editor. The Deployer returns the URL where the deployed composition is listening as service or any other required information to invoke it.

Anytime after deployment, the service provider can enable or disable the composition (depending on its status) through the Composition Execution UI. Moreover, through this UI, the service provider can monitor a composition within a specified time frame.
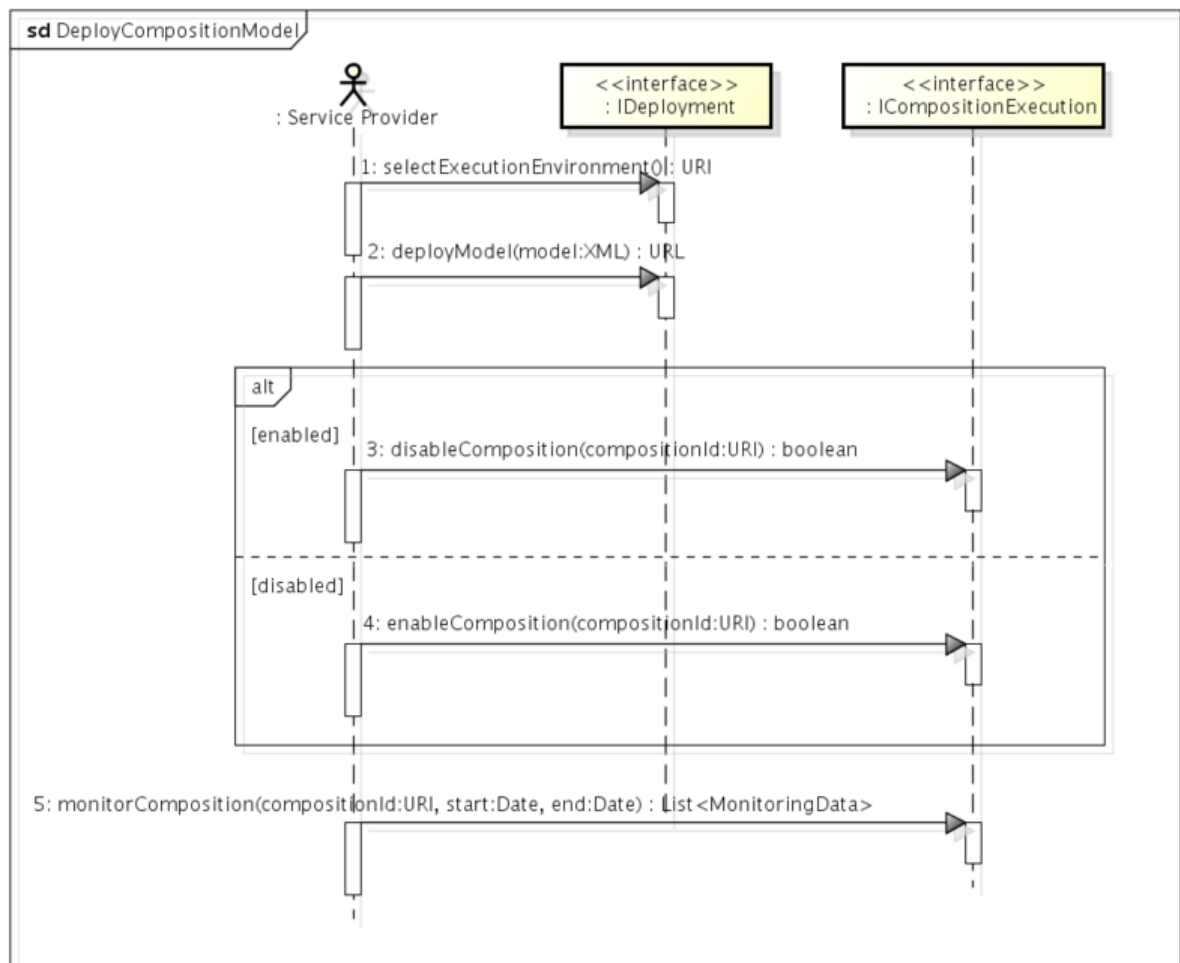
Deploy a service composition model

8.8 Design Principles

Sources:

- [SOA4All D1.1.1 Design Principles for a Service Web](#)
- [Evaluation of Service Construction](#)

This GE follows the following design principles:

- **Machine and human based computation principle**

  This GE allows a semi-automatic (human and machine) service composition modeling

- **Template-based composition**

  Recently created process models (and some fragments) can be reused in future modeling tasks

- **Reusable**

  This GE is completely domain independent concerning the knowledge intensive reusability feature.

- **Composability principle**

  The GE design allows it to split a complex process-modeling problem into several smaller ones that the GE resolves separately by its specific agents.

- **Openness principle**

  This GE can be easily extended either by coding/replacing architectural components

- **Ontology based principle**

  This GE extensively uses ontology based knowledge.

8.9 Detailed Specifications

8.9.1 Open API Specifications

The Light-weighted Semantic-enabled Composition GE is not exposed as a service but as a Webtop application (GUI), accessed throught the end user Web browser. Although some GE components are exposed as services, they only expose an API for internal consumption (within the GE), but it is not foreseen they will be integrated by other GEs.

The Light-weighted Semantic-enabled Composition GE will access the Marketplace and Repository REST APIs:

- FIWARE.OpenSpecification.Apps.MarketplaceSearchREST
- FIWARE.OpenSpecification.Apps.RepositoryREST

8.10 Re-utilised Technologies/Specifications

The Light-weighted Semantic-enabled Composition GE relies on following technical specifications:

- BPMN 2.0
- WSMOLite
- MicroWSMO
- BPEL 2.0

- [USDL](#)
- [WSDL 2.0](#)
- [WSDL](#)
- [Minimum Service Model (MSM)](#)
- [BPMN to BPEL](#)
- [BPMN XSD/XMI serialization formats](#)
- [RDF/XML format](#)
- [N3 format](#)
- [SPARQL 1.1](#)

8.11   Terms and definitions

This section comprises a summary of terms and definitions introduced during the previous sections. It intends to establish a vocabulary that will be help to carry out discussions internally and with third parties (e.g., Use Case projects in the EU FP7 Future Internet PPP). For a summary of terms and definitions managed at overall FI-WARE level, please refer to [FIWARE Global Terms and Definitions](#)

- **Aggregator (Role):** Supports domain specialists and third-parties in aggregating services and apps for new and unforeseen opportunities and needs. It does so by providing the dedicated tooling for aggregating services at different levels: UI, service operation, business process or business object levels.

- **Application:** Applications in FI-Ware are composite services that have a IT supported interaction interface (user interface). In most cases consumers do not buy the application they rather buy the right to use the application (user license).

- **Broker (Role):** The business network's central point of service access, being used to expose services from providers that are to be delivered through the Broker's service delivery functionality. The broker is the central instance for enabling monetization.

- **Business Element:** Core element of a business model, such as pricing models, revenue sharing models, promotions, SLAs, etc.

- **Business Framework:** Set of concepts and assets responsible for supporting the implementation of innovative business models in a flexible way.

- **Business Model:** Strategy and approach that defines how a particular service/application is supposed to generate revenue and profit. Therefore, a Business Model can be implemented as a set of business elements which can be combined and customized in a flexible way and in accordance to business and market requirements and other characteristics.

- **Business Process:** Set of related and structured activities producing a specific service or product, thereby achieving one or more business objectives. An operational business process clearly defines the roles and tasks of all involved parties inside an organization to achieve one specific goal.

- **Business Role:** Set of responsibilities and tasks that can be assigned to concrete

business role owners, such as a human person or a software component.

- ''''*Channel: Resources through which services are accessed by end users. Examples for well-known channels are Web sites/portals, web-based brokers (like iTunes, eBay and Amazon), social networks (like Facebook, LinkedIn and MySpace), mobile channels (Android, iOS) and work centres. The mode of access to these channels is governed by technical channels like the Web, mobile devices and voice response, where each of these channels requires its own specific workflow.*

- **Channel Maker (Role):** Supports parties in creating outlets (the Channels) through which services are consumed, i.e. Web sites, social networks or mobile platforms. The Channel Maker interacts with the Broker for discovery of services during the process of creating or updating channel specifications as well as for storing channel specifications and channeled service constraints back in the Broker.

- **Composite Service (composition):** Executable composition of business back-end MACs. Common composite services are either orchestrated or choreographed. Orchestrated compositions are defined by a centralized control flow managed by a unique process that orchestrates all the interactions (according to the control flow) between the external services that participate in the composition. Choreographed compositions do not have a centralized process, thus the services participating in the composition autonomously coordinate each other according to some specified coordination rules. Backend compositions are executed in dedicated process execution engines. Target users of tools for creating Composites Services are technical users with algorithmic and process management skills.

- **Consumer (Role):** Actor who searches for and consumes particular business functionality exposed on the Web as a service/application that satisfies her own needs.

- **Desktop Environment:** Multi-channel client platform enabling users to access and use their applications and services.

- **Event-driven Composition:** Components concerned with the composition of business logic which is driven by asynchronous events. This implies run-time selection of MACs and the creation/modification of orchestration workflows based on composition logic defined at design-time and adapted to context and the state of the communication at run-time.

- **Front-end/Back-end Composition:** Front-end compositions define a front-end application as an aggregation of visual mashable application pieces (named as widgets, gadgets, portlets, etc.) and back-end services. Front-end compositions interact with end-users, in the sense that front-end compositions consume data provided by the end-users and provide data to them. Thus the frontend composition (or mashup) will have direct influence on the application look and feel; every component will add a new user interaction feature.

- Back-end compositions define a back-end business service (also known as process) as an aggregation of backend services as defined for service composition term, the end-user being oblivious to the composition process. While back-end components represent atomization of business logic and information processing, front-end components represent atomization of information presentation and user interaction.

- **Gateway (Role):** The Gateway role enables linking between separate systems and services, allowing them to exchange information in a controlled way despite different technologies and authoritative realms. A Gateway provides interoperability solutions for other applications, including data mapping as well as run-time data store-forward

and message translation. Gateway services are advertised through the Broker, allowing providers and aggregators to search for candidate gateway services for interface adaptation to particular message standards. The Mediation is the central generic enabler. Other important functionalities are eventing, dispatching, security, connectors and integration adaptors, configuration, and change propagation.

- **Hoster (Role):** Allows the various infrastructure services in cloud environments to be leveraged as part of provisioning an application in a business network. A service can be deployed onto a specific cloud using the Hoster's interface. This enables service providers to re-host services and applications from their on-premise environments to cloud-based, on-demand environments to attract new users at much lower cost.

- **Marketplace:** Part of the business framework providing means for service providers to publish their service offerings and service consumers to compare and select a specific service implementation. A marketplace can offer services from different stores and thus different service providers. The actual buying of a specific service is handled by the related service store.

- **Mashup:** Executable composition of front-end MACs. There are several kinds of mashups, depending on the technique of composition (spatial rearrangement, wiring, piping, etc.) and the MACs used. They are called application mashups when applications are composed to build new applications and services/data mash-ups if services are composed to generate new services. While composite service is a common term in backend services implementing business processes, the term 'mashup' is widely adopted when referring to Web resources (data, services and applications). Front-end compositions heavily depend on the available device environment (including the chosen presentation channels). Target users of mashup platforms are typically users without technical or programming expertise.

- **Mashable Application Component (MAC):** Functional entity able to be consumed executed or combined. Usually this applies to components that will offer not only their main behaviour but also the necessary functionality to allow further compositions with other components. It is envisioned that MACs will offer access, through applications and/or services, to any available FI-WARE resource or functionality, including gadgets, services, data sources, content, and things. Alternatively, it can be denoted as 'service component' or 'application component'.

- **Mediator:** A mediator can facilitate proper communication and interaction amongst components whenever a composed service or application is utilized. There are three major mediation area: Data Mediation (adapting syntactic and/or semantic data formats), Protocol Mediation (adapting the communication protocol), and Process Mediation (adapting the process implementing the business logic of a composed service).

- **Monetization:** Process or activity to provide a product (in this context: a service) in exchange for money. The Provider publishes certain functionality and makes it available through the Broker. The service access by the Consumer is being accounted according to the underlying business model and the resulting revenue is shared across the involved service providers.

- **Premise (Role):** On-Premise operators provide in-house or on-site solutions, which are used within a company (such as ERP) or are offered to business partners under specific terms and conditions. These systems and services are to be regarded as external and legacy to the FI-Ware platform because they do not conform to the architecture and API specifications of FI-Ware. They will only be accessible to FI-

Ware services and applications through the Gateway.

- **Prosumer:** A user role able to produce, share and consume their own products and modify/adapt products made by others.

- **Provider (Role):** Actor who publishes and offers (provides) certain business functionality on the Web through a service/application endpoint. This role also takes care of maintaining this business functionality.

- **Registry and Repository:** Generic enablers that able to store models and configuration information along with all the necessary meta-information to enable searching, social search, recommendation and browsing, so end users as well as services are able to easily find what they need.

- **Revenue Settlement:** Process of transferring the actual charges for specific service consumption from the consumer to the service provider.

- **Revenue Sharing:** Process of splitting the charges of particular service consumption between the parties providing the specific service (composition) according to a specified revenue sharing model.

- **Service:** We use the term service in a very general sense. A service is a means of delivering value to customers by facilitating outcomes customers want to achieve without the ownership of specific costs and risks. Services could be supported by IT. In this case we say that the interaction with the service provider is through a technical interface (for instance a mobile app user interface or a Web service). Applications could be seen as such IT supported Services that often ar also composite services.

- **Service Composition:** in SOA domain, a service composition is an added value service created by aggregation of existing third party services according to some predefined work and data flow. Aggregated services provide specialized business functionality on which the service composition functionality has been split down.

- **Service Delivery Framework:** Service Delivery Framework (or Service Delivery Platform (SDP)) refers to a set of components that provide service delivery functionality (such as service creation, session control & protocols) for a type of service. In the context of FI-WARE, it is defined as a set of functional building blocks and tools to (1) manage the lifecycle of software services, (2) creating new services by creating service compositions and mashups, (3) providing means for publishing services through different channels on different platforms, (4) offering marketplaces and stores for monetizing available services and (5) sharing the service revenues between the involved service providers.

- **Service Level Agreement (SLA):** A service level agreement is a legally binding and formally defined service contract between a service provider and a service consumer, specifying the contracted qualitative aspects of a specific service (e.g. performance, security, privacy, availability or redundancy). In other words, SLAs not only specify that the provider will just deliver some service, but that this service will also be delivered on time, at a given price, and with money back if the pledge is broken.

- **Service Orchestration:** in SOA domain, a service orchestration is a particular architectural choice for service composition where a central orchestrated process manages the service composition work and data flow invocations the external third party services in the order determined by the work flow. Service orchestrations are specified by suitable orchestration languages and deployed in execution engines who interpret these specifications.

- **Store:** An external component integrated with the business framework offering a set

of services that are published to a selected set of marketplaces. The store thereby holds the service portfolio of a specific service provider. In case a specific service is purchased on a service marketplace, the service store handles the actual buying of a specific service (as a financial business transaction).

- **Unified Service Description Language (USDL):** USDL is a platform-neutral language for describing services, covering a variety of service types, such as purely human services, transactional services, informational services, software components, digital media, platform services and infrastructure services. The core set of language modules offers the specification of functional and technical service properties, legal and financial aspects, service levels, interaction information and corresponding participants. USDL is offering extension points for the derivation of domain-specific service description languages by extending or changing the available language modules.

9 FIWARE OpenSpecification Apps Marketplace

You can find the content of this chapter as well in the wiki of fi-ware.

Name	FIWARE.OpenSpecification.Apps.Marketplace
Chapter	Apps,
Catalogue-Link to Implementation	Marketplace
Owner	SAP AG, Andreas Klein

9.1 Preface

Within this document you find a self-contained open specification of a FI-WARE generic enabler, please consult as well the FI-WARE_Product_Vision, the website on http://www.fi-ware.eu and similar pages in order to understand the complete context of the FI-WARE project.

9.2 Copyright

- Copyright © 2012 by SAP

9.3 Legal Notice

Please check the following Legal Notice to understand the rights to use these specifications.

SAP strives to make the specifications of this Generic Enabler available under IPR rules that allow for a exploitation and sustainable usage both in Open Source as well as proprietary, closed source products to maximize adoption.

9.4 Overview

In general a marketplace is an instrument to facilitate commerce by bringing together vendors and buyers, or offers and demand, or producers and consumers. A marketplace can support a variety of mechanisms to achieve this. This specification describes the FI-WARE Marketplace Generic Enabler, which is part of the Applications/Services Ecosystem. Any offering in context of the application and service business can be supported by marketplace functionality. A marketplace implementation can offer many different kinds of services to the participants.

The core functionality of the Marketplace is to provide a uniform service interface to discover and match application and service offerings from providers and sources (e.g. published by different stores) with demand of consumers. This core functionality provides a basis for extended services depending on the domain and nature of the target markets.

9.4.1   Target usage

Internet based business networks require at least one marketplace and stores, where people can offer and deal with services like goods and finally combine them to value added services. On the marketplace one can quickly find and compare services, which enable you to attend an industry-ecosystem better than before. Services become tradable goods, which can be offered and acquired on internet based marketplaces. Beside automated internet services this also applies for services that are provided by individuals. Partner companies can combine existing services to new services whereby new business models will be incurred and the value added chain is extended.

Given the multitude of apps and services that will be available on the Future Internet, providing efficient and seamless capabilities to locate those services and their providers will become crucial to establish service and app stores. Besides well-known existing commercial application stores like Apple App Store, Google Android Market, and Nokia Ovi, there are first efforts to establish open service and app marketplaces, e.g. in the Amazon Web Service Marketplace, the U.S. Government's Apps.Gov repository and Computer Associates' Cloud Commons Marketplace. While these marketplaces already contain a considerable number of services, they are currently, at a premature stage, offering little more than a directory service. FI-WARE will fill this gap by defining generic enablers for marketplaces and providing reference implementations for them.

9.4.1.1   *User roles*

- Service provider will place offers on the marketplace or in a service/app store.

- Consumer can search, browse and compare offers

- Repository will be used to get services descriptions

- Registry acts as a universal directory of information used for the maintenance, administration, deployment and retrieval of services. It will be used to store information necessary for service run-time execution.

- Service store will participate on a marketplace and publishes offerings.

- Channel Maker will consumers give access to the marketplace

9.5   Basic Concepts

The marketplace is structured into five core components. These components are *Registry & Directory*, *Offering & Demand*, *Discovery & Matching*, *Recommendation*, and the *Review and Rating* component.

9.5.1   Registry and Directory

The Registry and Directory component holds information of registered stores, participants and their role(vendors, buyers, resellers, ...) and takes care of registering, updating, and deleting information about market relevant entities.

9.5.2   Offering & Demand

A service offering consists of a link to a concrete USDL description, a pricing model and the classification of the service. The Offering component is responsible for exchanging service offerings with stores and version handling/archiving of out-dated offerings.

Symmetrically to offerings also the demand side of the market need to be represented. A service demand according to expected functionality, pricing and service levels might be expressed, classified and published to the marketplace.

9.5.3   Discovery & Matching

This component is about discovering and matching offering and demand, either explicitly expressed by concrete offerings or implicitly contained in the search criteria for the inquiry process.

9.5.4   Recommendation

This component provides the user service recommendations based on the users' 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).

9.5.5   Review & Rating

The Review and Rating component allows users of the marketplace to give textual and star-rating feedback for services and stores along predefined categories. Reviews of users and their overall rating about applications and services can be used to improve the quality of the recommendation.

9.6   Marketplace Architecture

The Marketplace GE is used by other GE within the FI-WARE platform. Namely the Composition Editor and the Service IDE. The Marketplace itself relies on functionality provided by the Identity Management Service GE and the Repository.
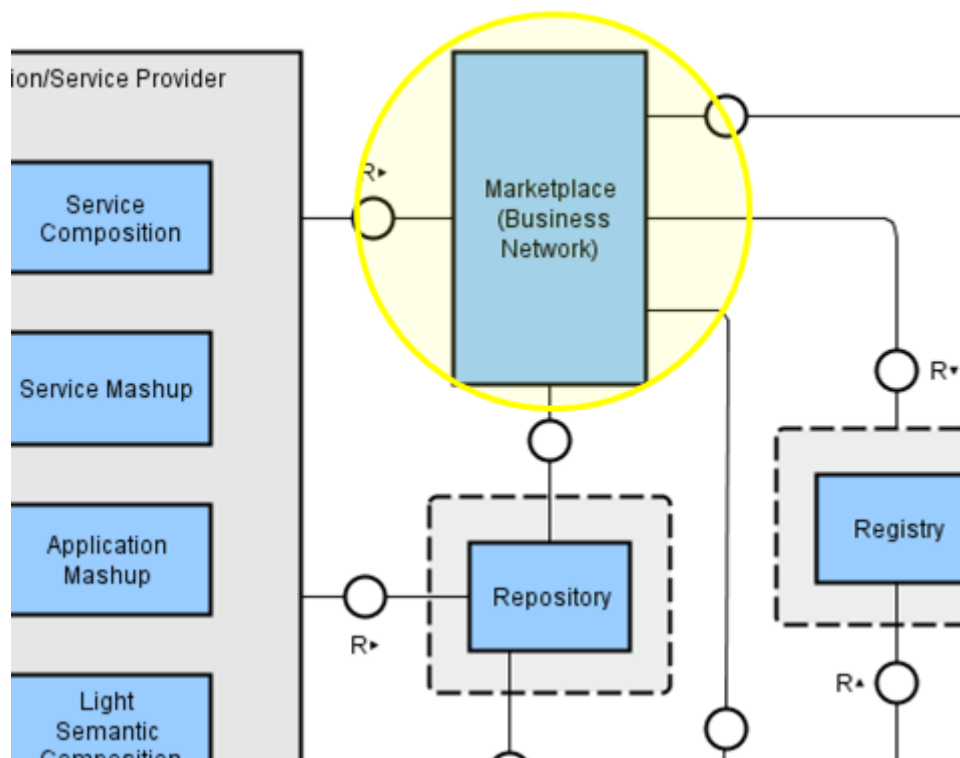
Marketplace in the context of the Business Framework

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, search and discover offerings according to specific consumer requirements as well as lateral functions like review, rating and recommendation. FI-Ware will focus on the core functions but aims to provide a framework to offer also additional services. Besides the core functions, a marketplace may offer value because of its "knowledge" about the market in terms of market intelligence services, pricing support, advertising, information subscription and more.

Furthermore, the marketplace can be accessible by a HTML-based user interface for end users (Marketplace Portal), namely service consumers and service providers (stores) or a programmatic API, which allows to embed market place functionality into existing applications and environments. So for example can the marketplace API be used in order to provide marketplace access directly embedded into the composition environment (in-app shopping). A service/application store can register at the marketplace in order to get access to an API which allows creating new service offerings on the marketplace. The actual buying process always takes place at a store. So for ordering a specific service, the buyer gets returned to the stores ordering management service.

The functional components of the marketplace implementation are outlined the following diagram. This functional architecture picture is to be seen as a blueprint for implementers of the marketplace GE. It does not presume the use of a certain technology. The general idea is that all functions will operate on a huge (virtual) database of entities relevant for the business framework, such as people, organizations, products, services, offerings, ratings, etc. In practice this data might be in one uniform distributed database or a set of databases, which only virtually build the marketplace database. In fact for scaling reasons, in huge marketplaces the database is rather realized by many systems, distributed globally. Linked Data can be a good foundation for a marketplace database abstraction layer (the web is the

database). In this case, which FI-WARE is focussing on, the marketplace database is more like a (semantic) index of all marketplace information bits and pieces stored elsewhere. As a query and update mechanism, we are relying on Semantic Web Repositories, Query Systems and a Query Language.
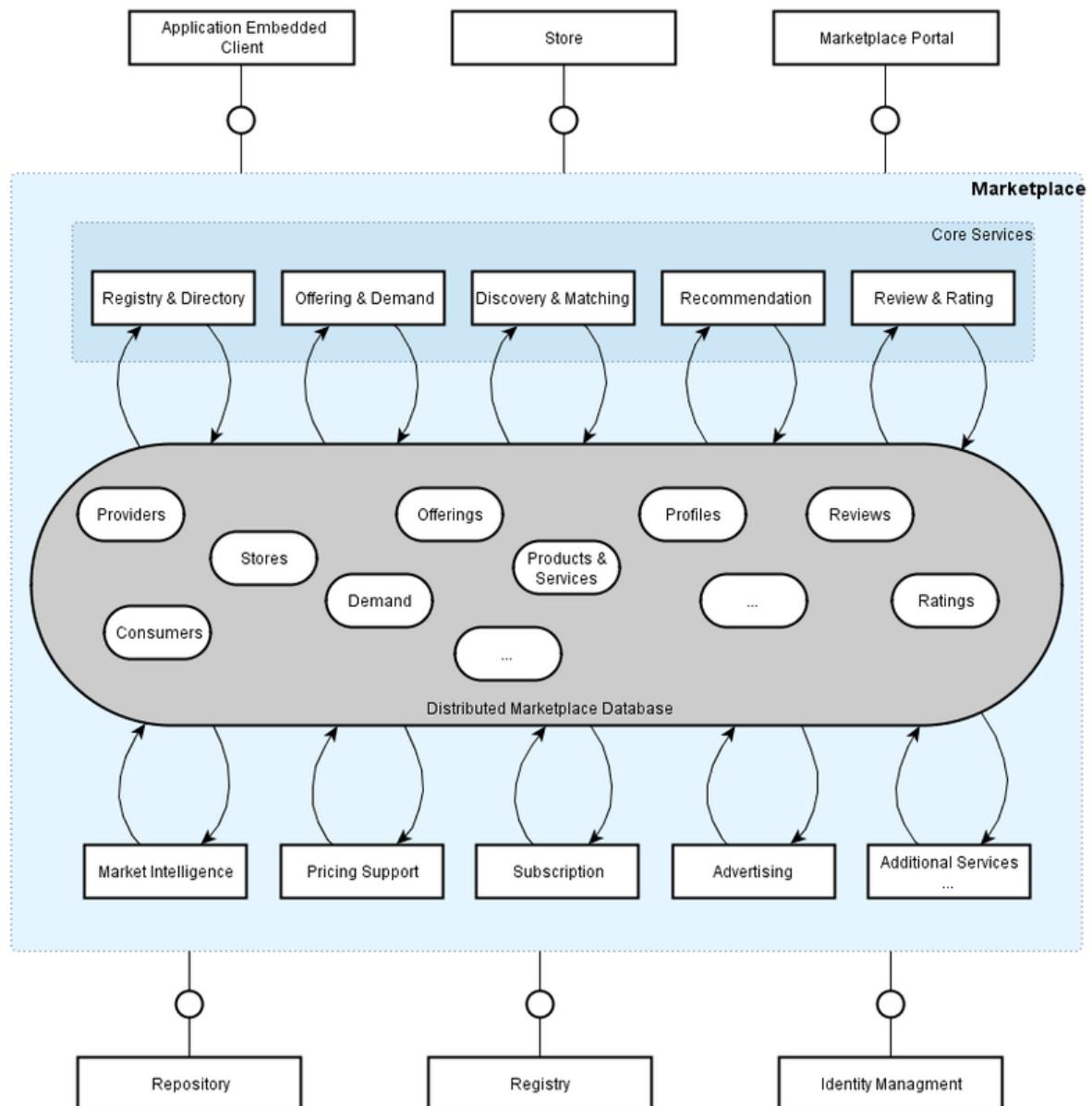
Example architecture of a marketplace implementation

9.7 Main Operations

Each functional block of the Marketplace can be considered as a GE as well. Consequently it has an own abstract operation protocol specification attached.

9.7.1 Registration and Directory

It should be possible to register a number of marketplace relevant entity types such as stores, market participants and their roles in certain business aspects (provider, consumer, reseller ...). The diagram below depicts an exemplary sequence of operations with the Registration and Directory component. A user application refers to an application which has marketplace functionally embedded.
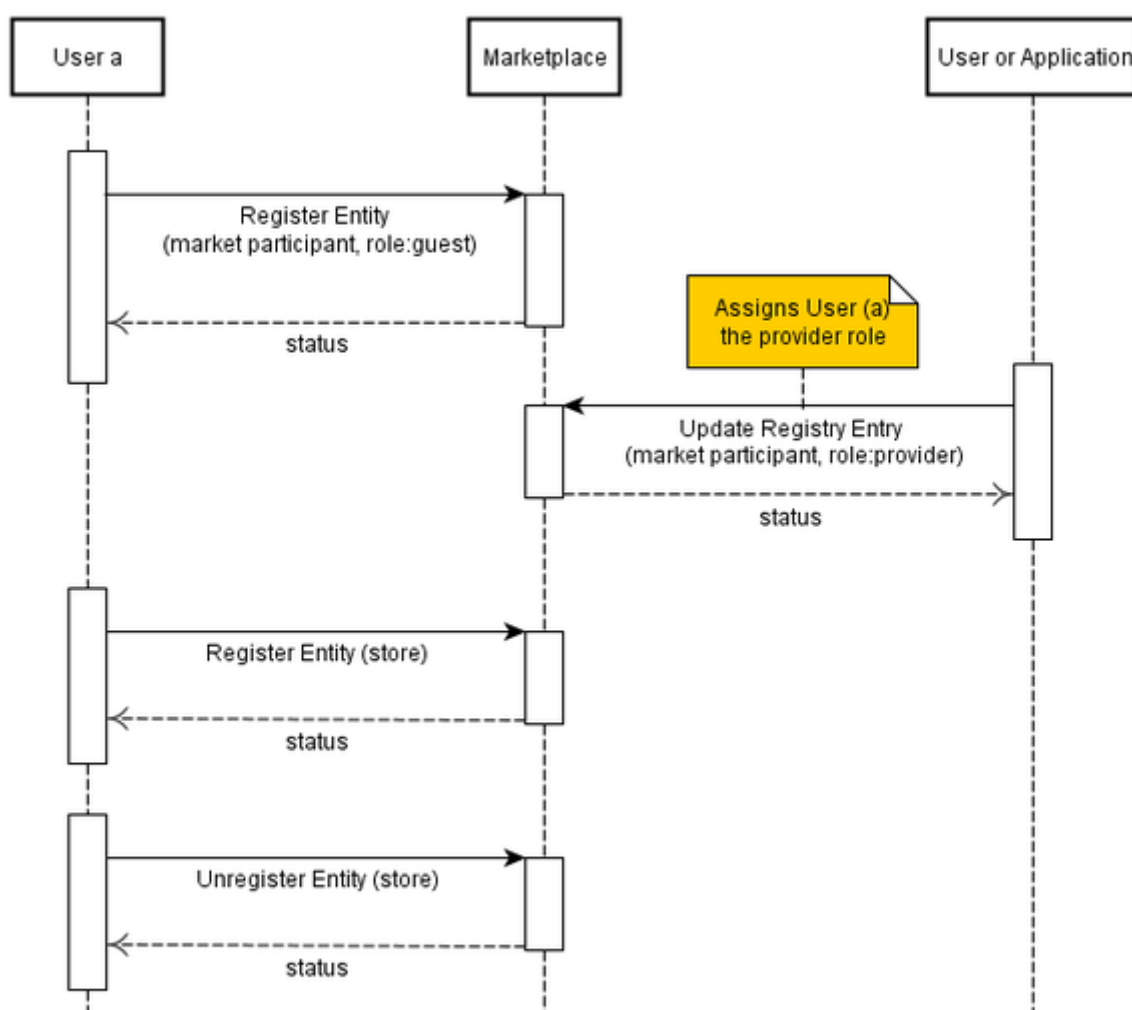
Example sequence of registration operations

The Marketplace provides a *Register Entity* operation for registering market participants and related business elements. Register a new Store on the marketplace. Parameters:

- *entity* - Description of the entity to be created.

The following core entity types are supported:

- *Store* - The stores which are covered by the marketplace. There will be further operations of other marketplace components, which will need to communicate with the stores, in order to get the stores offerings and invoke operations on the stores.

- *Market Participant* - The participants (users) of the marketplace. A market participant must have at least one of the following roles:

  o **Guest** - Guest users are only allowed to the discovery capabilities of the marketplace.

  o **Consumer** - The actual buyers. The organizations or persons who can buy services and applications on the marketplace.

  o **Provider** - Providers of services and applications available on the marketplace. Information about providers is needed in different components on the marketplace.

  o **Reseller** - Reseller of services and applications. The reseller has no own service portfolio and sells services from other providers.

Existing registration information on the marketplace can be modified or updated with the **Update Registry Entry** operation. The parameters are similar to the Register Entity operation:

- *entity* - Description of the entity to be updated including the entity identifier.

For unregistering entities such as stores or market participants from the marketplace, the **Unregister Entry** operation will be used. Unregistering an entity from the Marketplace means that this entity gets flagged as unregistered and not further regarded for the marketplace operation. So, complete deletion is not possible due to consistency and history reasons. The following parameter is required:

- *entity* - Entity to be unregistered from the marketplace registry.

9.7.2 Offering & Demand

Offerings are retrieved from various sources (actually mainly stores, but other sources would be possible). So the operation is rather to ask a registered store for actual offerings. The following diagram shows an example sequence of offering management on a marketplace.
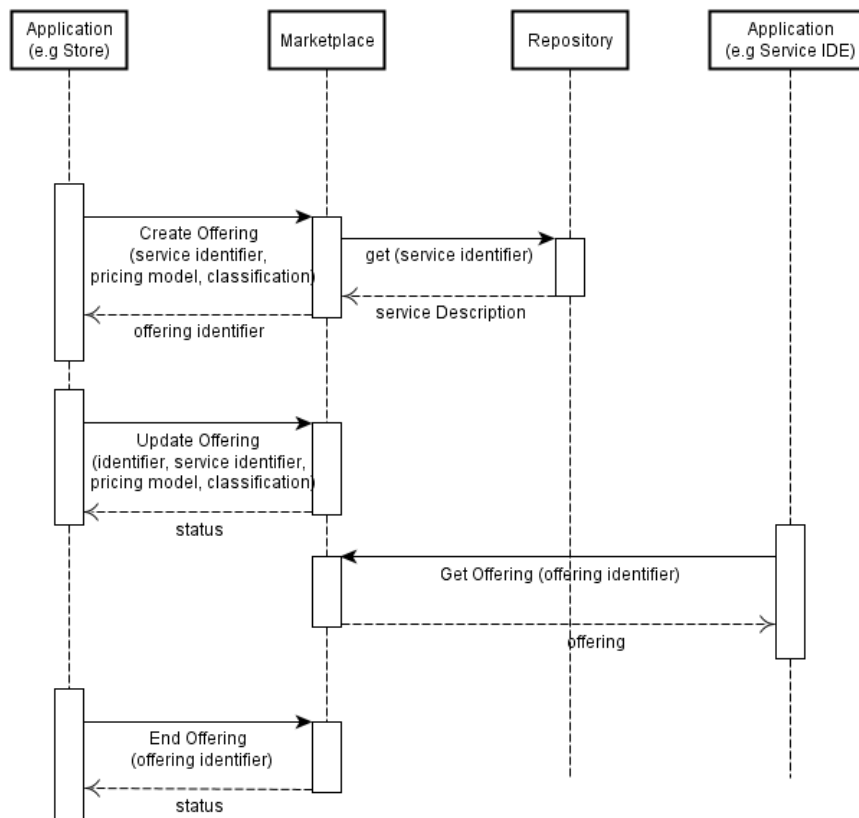
Example sequence of offering management operations

The ***Create Offering*** operation can be used to actively push an offering from a registered store into a marketplace. This operation returns the identifier of the published offering. The following parameters need to be exchanged:

- *service identifier* - Identifier of the service to be offered / link to a USDL service description.

- *pricing model* - Pricing model for the service/app.

- *classification* - Optional - Classification of the offering, e.g. name of a category the offering belongs to

The ***Update Offering*** operation can be used to update an already published offering on the marketplace. The old offering information gets versioned on the markeplace. This operation returns an OK status result on success or a parameter allowing to identify the reason for fault for further exception handling. The following parameters need to be exchanged:

- *offering identifier* - Identifier of the offering.

- *service identifier* - Identifier of the service to be offered / link to a USDL service description.

- *pricing model* - New or updated pricing model for the service/app.

- *classification* - Optional - Classification of the offering.

In order to withdraw or end a concrete offering from the marketplace the ***End Offering***

operation is used. A complete deletion is not possible due to dependency and history reasons. The following parameters are exchanged:

- *offering identifier* - Identifier of the offering.

This operation returns an OK status result on success or a parameter allowing to identify the reason for fault for further exception handling.

The **Get Offering** operation can be used to retrieve a offering from the marketplace. This operation delivers the actual offering data, such as pricing information, service description, associated store, and the offering classification. The following parameters need to be exchanged:

- *offering identifier* - Identifier of the offering.

- *version* - Optional parameter. If not set, the latest version of the requested offering is returned

The **Get Offering History** operation can be used to retrieve the history of an offering from the marketplace. This operation delivers a list of all versions of an offering. The following parameters need to be exchanged:

- *offering identifier* - Identifier of the offering.

The **List Offerings for a Store** operation can be used to retrieve all offerings from the marketplace from a specific store. For practical reasons it might be useful to restrict the list of offerings by specifying the number of results and the starting offset. The following parameters need to be exchanged:

- *store Identifier* - Identifier of the store.

- *filter* - Optional filter expression to reduce the number of delivered results.

- *index* - Index of the first offering to be returned.

- *limit* - Maximal number of results to be returned.

9.7.3   Discovery & Matching

The Discovery and Matching component supports primarily customers finding offerings and stores matching their needs. The following diagram shows an example sequence of a marketplace user (a) searching and comparing offerings as well as a second user (b) searching for stores.
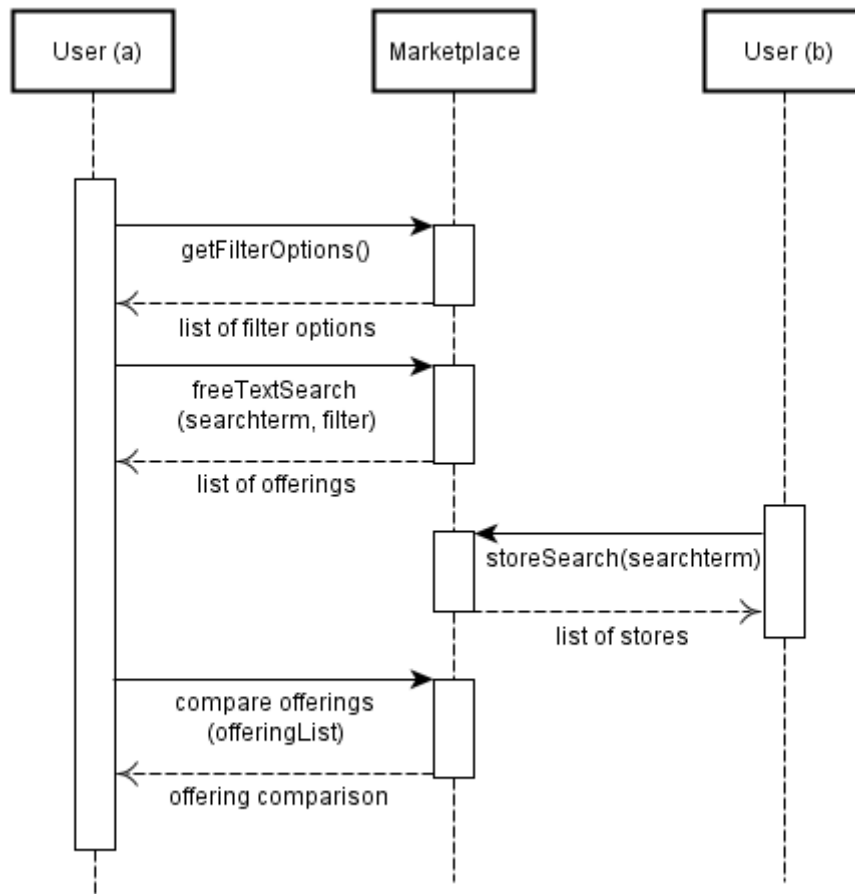
Example sequence of discovery and matching operations

The **Free Text Search** operation can be used to search the marketplace for offerings using a search string with wildcards and filters. This operation delivers a list of all offerings which matches the specified search term and filter constraints. The following parameters can be involved in the operation:

- *search string* - Search string, potentially with wildcard operators.
- *filter* - Optional - Filter expression to reduce the number of delivered results.
- *index* - Optional - Index of the first offering to be returned.
- *limit* - Optional - Maximal number of results to be returned.
- *page size* - Optional - Number of results per page.
- *sort by* - Optional - List of sortoptions, sorted by application order.

The **Get Filter Options** operation can be used to get the possible filter options for a free text search. There might exist classification category specific filter options. If the classification input parameters is set, a list of these specific filter parameters gets returned.

- *classification* - Optional parameter to get filter options for offerings associated with the specified classification.

Sort options are used to define the order of an offering result list. **The Get Sort Options** operation returns a list of possible sort options for a list of offerings. If a classification category is specified in the request, category specific sort options get returned. The optional parameter for category specific sort options is:

- *classification category* - Optional parameter to get filter options for offerings associated with the specified classification.

To get a comparison of pricing models and USDL service descriptions between offerings the **Compare Offerings** operation is used. An optional filter expression can be used to reduce the number of delivered results. The parameters for this operation are:

- *offering list* - List of offerings

- *filter* - Optional - Filter expression to reduce the number of delivered results.

- *index* - Optional - Index of the first offering to be returned.

- *limit* - Optional - number of results to be returned.

The **Store Search** operation enables a client to search a marketplace for registered stores using a search string with wildcards and filters. A list of stores matching the search string as well as the specified filter criteria is returned.

- *search string* - Search string, potentially with wildcard operators

- *filter* - Optional - Filter expression to reduce the number of delivered results.

- *index* - Optional - Index of the first store to be returned.

- *limit* - Optional - Maximal number of results to be returned.

9.7.4   Review and Rating

The Review and Rating component allows users of the marketplace to retrieve and create textual and star-rating feedback for rateable entities such as stores and services. An example sequence of marketplace users creating and retrieving ratings is pictured below.
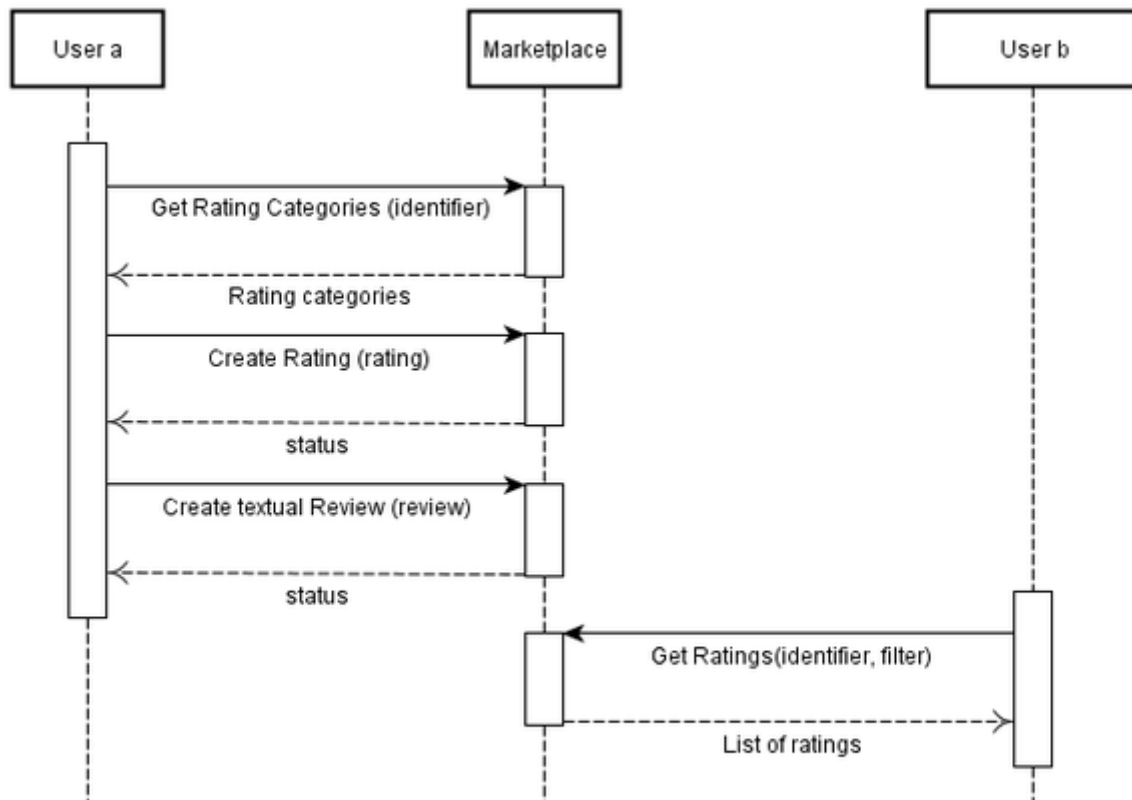
Review and Rating Example sequence

To get the average rating for a store, a service or any other rateable entity on the marketplace the **Get Rating** operation is used. This operation delivers the average ratings for a rateable entity by means of different rating categories as well as an overall average rating. The following parameters need to be exchanged:

- *identifier* - Identifier of a store or a user or any other rateable entity

The **Get Ratings** operation delivers get the details of ratings for a rateable entity. Filter expressions are supported to reduce the number of results. If no filter expression is given then all ratings for the specified entity instance are returned.

- *identifier* - Identifier of a store or a service or any other rateable entity

- *filter* - Optional filter expression to reduce the number of delivered results.

- *index* - Index of the first rating to be returned.

- *limit* - Maximal number of results to be returned.

In order to retrieve a list of textual reviews for a rateable entity the **Get textual Reviews** operation is used. It also supports filter expressions as well as limits and offsets. The following parameters are available for this operation:

- *identifier* - Identifier of a store or a user.

- *filter* - Optional filter expression to reduce the number of delivered results.

- *index* - Index of the first reviews to be returned.

- *limit* - Maximal number of results to be returned.

The **Create Rating** operation is used to persist a new rating for a entity instance in the marketplace. This operation returns the identifier of the new rating as result. To get the available rating categories for a rateable entity use the *Get Rating Categories* operation beforehand. The following parameter needs to be exchanged:

- *rating* - Rating entry which includes a rating value for each mandatory rating category as well as a link to the rateable entity instance

To get the available rating categories for a rateable entity the **Get Rating Categories** is used. It returns a list of available rating categories for the specified service, store, or any other rateable entity instance. The following parameter has to be provided:

- *identifier* - Identifier of a rateable entity instance.

The **Create textual Review** creates a textual review for a entity instance that is flagged as rateable. It returns the identifier of the created review as result. As input parameter, the textual review is sufficient:

- *review* - Review entry which includes the textual review and an the identifier of the rateable entity instance.

9.7.5   Recommendation

The Recommendation component supports users in finding services matching their needs based on user specific data. The following diagram outlines an example sequence of a user retrieving recommendations.
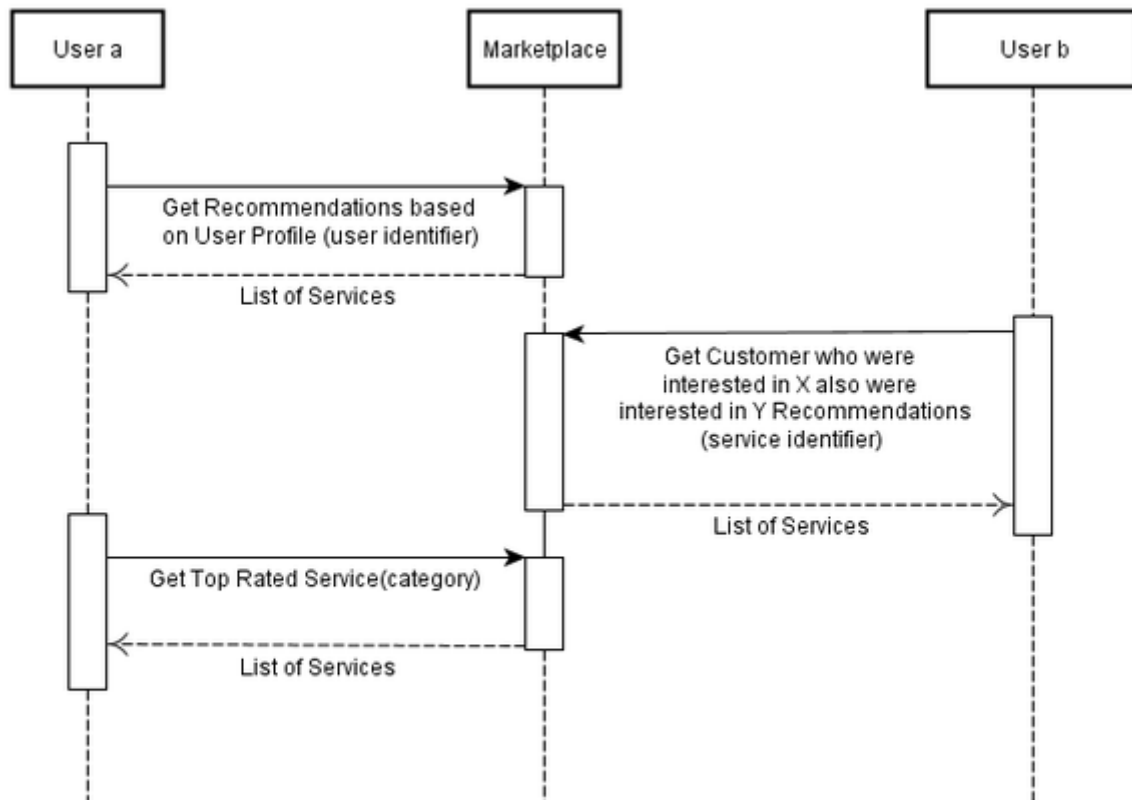
Example sequence of recommendation operations

To retrieve a list of recommendations for a user the **Get Recommendations based on User Profile** operation is used. Recommendations might be based on the users' profile, browsing behaviour, order history and other user specific data. This operation returns a list of recommended services till the specified limit. The following parameters need to be provided:

- *user identifier* - Identifier of a user

- *limit* - Optional - Maximal number of results to be returned.

The **Get Customer who were interested in X also were interested in Y Recommendations** returns a list of services that were often bought together with the a given service. The supported parameters for this operation are:

- *user identifier* - Identifier of a service

- *limit* - Optional - Maximal number of results to be returned.

To get the top rated services on the marketplace as a whole or the top rated services for a certain classification category the **Get Top Rated Services** is used. The following parameters are supported:

- *classification category* - Optional parameter to get the top rated services for a certain classification category.

- *limit* - Optional - Maximal number of results to be returned.

9.8　Design Principles

- **API Technology Independence**

The Marketplace API is independent from implementation technology.

- **Interoperability and flexible use through HTTP content negotiation**

As described from the architecture, the Marketplace offers different interfaces in order to satisfy users' need to discover, compare, create, and update offerings. The Marketplace determines the right representation from information provided in the users' header data, so a client can receive the best representation for its abilities.

- **Mutiple store support**

The Marketplace GE is not limited to interact with one specific store. It supports multiple decoupled stores as long as they implement the marketplace interfaces for Registry and Offering & Demand.

9.9　Detailed Specifications

9.9.1　Open API Specifications

- [FIWARE.OpenSpecification.Apps.MarketplaceRegistrationREST](#)
- [FIWARE.OpenSpecification.Apps.MarketplaceOfferingsREST](#)
- [FIWARE.OpenSpecification.Apps.MarketplaceSearchREST](#)

The Marketplace GE will access the Repository its REST API:

- [FIWARE.OpenSpecification.Apps.RepositoryREST](#)

9.9.2　Other Open Specifications

The data formats for the API rely on the Linked USDL specifications:

- [Linked USDL Core Vocabulary](#)
- [Linked USDL Pricing Vocabulary](#)
- [Linked USDL Service Level Agreements Vocabulary](#)
- [Linked USDL Security Vocabulary](#)

9.10　Re-utilised Technologies/Specifications

The Marketplace GE is based on RESTful Design Principles. The technologies and specifications used in this GE are:

- RESTful web services
- HTTP/1.1
- JSON and XML data serialization formats
- Linked USDL

9.11   Terms and definitions

This section comprises a summary of terms and definitions introduced during the previous sections. It intends to establish a vocabulary that will be help to carry out discussions internally and with third parties (e.g., Use Case projects in the EU FP7 Future Internet PPP). For a summary of terms and definitions managed at overall FI-WARE level, please refer to FIWARE Global Terms and Definitions

- **Aggregator (Role):** Supports domain specialists and third-parties in aggregating services and apps for new and unforeseen opportunities and needs. It does so by providing the dedicated tooling for aggregating services at different levels: UI, service operation, business process or business object levels.

- **Application:** Applications in FI-Ware are composite services that have a IT supported interaction interface (user interface). In most cases consumers do not buy the application they rather buy the right to use the application (user license).

- **Broker (Role):** The business network's central point of service access, being used to expose services from providers that are to be delivered through the Broker's service delivery functionality. The broker is the central instance for enabling monetization.

- **Business Element:** Core element of a business model, such as pricing models, revenue sharing models, promotions, SLAs, etc.

- **Business Framework:** Set of concepts and assets responsible for supporting the implementation of innovative business models in a flexible way.

- **Business Model:** Strategy and approach that defines how a particular service/application is supposed to generate revenue and profit. Therefore, a Business Model can be implemented as a set of business elements which can be combined and customized in a flexible way and in accordance to business and market requirements and other characteristics.

- **Business Process:** Set of related and structured activities producing a specific service or product, thereby achieving one or more business objectives. An operational business process clearly defines the roles and tasks of all involved parties inside an organization to achieve one specific goal.

- **Business Role:** Set of responsibilities and tasks that can be assigned to concrete business role owners, such as a human person or a software component.

- ''''*Channel: Resources through which services are accessed by end users. Examples for well-known channels are Web sites/portals, web-based brokers (like iTunes, eBay and Amazon), social networks (like Facebook, LinkedIn and MySpace), mobile channels (Android, iOS) and work centres. The mode of access to these channels is governed by technical channels like the Web, mobile devices and voice response, where each of these channels requires its own specific workflow.*

- **Channel Maker (Role):** Supports parties in creating outlets (the Channels) through which services are consumed, i.e. Web sites, social networks or mobile platforms. The Channel Maker interacts with the Broker for discovery of services during the process of creating or updating channel specifications as well as for storing channel specifications and channeled service constraints back in the Broker.

- **Composite Service (composition):** Executable composition of business back-end MACs. Common composite services are either orchestrated or choreographed. Orchestrated compositions are defined by a centralized control flow managed by a

unique process that orchestrates all the interactions (according to the control flow) between the external services that participate in the composition. Choreographed compositions do not have a centralized process, thus the services participating in the composition autonomously coordinate each other according to some specified coordination rules. Backend compositions are executed in dedicated process execution engines. Target users of tools for creating Composites Services are technical users with algorithmic and process management skills.

- **Consumer (Role):** Actor who searches for and consumes particular business functionality exposed on the Web as a service/application that satisfies her own needs.

- **Desktop Environment:** Multi-channel client platform enabling users to access and use their applications and services.

- **Event-driven Composition:** Components concerned with the composition of business logic which is driven by asynchronous events. This implies run-time selection of MACs and the creation/modification of orchestration workflows based on composition logic defined at design-time and adapted to context and the state of the communication at run-time.

- **Front-end/Back-end Composition:** Front-end compositions define a front-end application as an aggregation of visual mashable application pieces (named as widgets, gadgets, portlets, etc.) and back-end services. Front-end compositions interact with end-users, in the sense that front-end compositions consume data provided by the end-users and provide data to them. Thus the frontend composition (or mashup) will have direct influence on the application look and feel; every component will add a new user interaction feature. Back-end compositions define a back-end business service (also known as process) as an aggregation of backend services as defined for service composition term, the end-user being oblivious to the composition process. While back-end components represent atomization of business logic and information processing, front-end components represent atomization of information presentation and user interaction.

- **Gateway (Role):** The Gateway role enables linking between separate systems and services, allowing them to exchange information in a controlled way despite different technologies and authoritative realms. A Gateway provides interoperability solutions for other applications, including data mapping as well as run-time data store-forward and message translation. Gateway services are advertised through the Broker, allowing providers and aggregators to search for candidate gateway services for interface adaptation to particular message standards. The Mediation is the central generic enabler. Other important functionalities are eventing, dispatching, security, connectors and integration adaptors, configuration, and change propagation.

- **Hoster (Role):** Allows the various infrastructure services in cloud environments to be leveraged as part of provisioning an application in a business network. A service can be deployed onto a specific cloud using the Hoster's interface. This enables service providers to re-host services and applications from their on-premise environments to cloud-based, on-demand environments to attract new users at much lower cost.

- **Marketplace:** Part of the business framework providing means for service providers to publish their service offerings and service consumers to compare and select a specific service implementation. A marketplace can offer services from different stores and thus different service providers. The actual buying of a specific service is handled by the related service store.

- **Mashup:** Executable composition of front-end MACs. There are several kinds of mashups, depending on the technique of composition (spatial rearrangement, wiring, piping, etc.) and the MACs used. They are called application mashups when applications are composed to build new applications and services/data mash-ups if services are composed to generate new services. While composite service is a common term in backend services implementing business processes, the term 'mashup' is widely adopted when referring to Web resources (data, services and applications). Front-end compositions heavily depend on the available device environment (including the chosen presentation channels). Target users of mashup platforms are typically users without technical or programming expertise.

- **Mashable Application Component (MAC):** Functional entity able to be consumed executed or combined. Usually this applies to components that will offer not only their main behaviour but also the necessary functionality to allow further compositions with other components. It is envisioned that MACs will offer access, through applications and/or services, to any available FI-WARE resource or functionality, including gadgets, services, data sources, content, and things. Alternatively, it can be denoted as 'service component' or 'application component'.

- **Mediator:** A mediator can facilitate proper communication and interaction amongst components whenever a composed service or application is utilized. There are three major mediation area: Data Mediation (adapting syntactic and/or semantic data formats), Protocol Mediation (adapting the communication protocol), and Process Mediation (adapting the process implementing the business logic of a composed service).

- **Monetization:** Process or activity to provide a product (in this context: a service) in exchange for money. The Provider publishes certain functionality and makes it available through the Broker. The service access by the Consumer is being accounted according to the underlying business model and the resulting revenue is shared across the involved service providers.

- **Premise (Role):** On-Premise operators provide in-house or on-site solutions, which are used within a company (such as ERP) or are offered to business partners under specific terms and conditions. These systems and services are to be regarded as external and legacy to the FI-Ware platform because they do not conform to the architecture and API specifications of FI-Ware. They will only be accessible to FI-Ware services and applications through the Gateway.

- **Prosumer:** A user role able to produce, share and consume their own products and modify/adapt products made by others.

- **Provider (Role):** Actor who publishes and offers (provides) certain business functionality on the Web through a service/application endpoint. This role also takes care of maintaining this business functionality.

- **Registry and Repository:** Generic enablers that able to store models and configuration information along with all the necessary meta-information to enable searching, social search, recommendation and browsing, so end users as well as services are able to easily find what they need.

- **Revenue Settlement:** Process of transferring the actual charges for specific service consumption from the consumer to the service provider.

- **Revenue Sharing:** Process of splitting the charges of particular service consumption between the parties providing the specific service (composition) according to a

specified revenue sharing model.

- **Service:** We use the term service in a very general sense. A service is a means of delivering value to customers by facilitating outcomes customers want to achieve without the ownership of specific costs and risks. Services could be supported by IT. In this case we say that the interaction with the service provider is through a technical interface (for instance a mobile app user interface or a Web service). Applications could be seen as such IT supported Services that often ar also composite services.

- **Service Composition:** in SOA domain, a service composition is an added value service created by aggregation of existing third party services according to some predefined work and data flow. Aggregated services provide specialized business functionality on which the service composition functionality has been split down.

- **Service Delivery Framework:** Service Delivery Framework (or Service Delivery Platform (SDP)) refers to a set of components that provide service delivery functionality (such as service creation, session control & protocols) for a type of service. In the context of FI-WARE, it is defined as a set of functional building blocks and tools to (1) manage the lifecycle of software services, (2) creating new services by creating service compositions and mashups, (3) providing means for publishing services through different channels on different platforms, (4) offering marketplaces and stores for monetizing available services and (5) sharing the service revenues between the involved service providers.

- **Service Level Agreement (SLA):** A service level agreement is a legally binding and formally defined service contract between a service provider and a service consumer, specifying the contracted qualitative aspects of a specific service (e.g. performance, security, privacy, availability or redundancy). In other words, SLAs not only specify that the provider will just deliver some service, but that this service will also be delivered on time, at a given price, and with money back if the pledge is broken.

- **Service Orchestration:** in SOA domain, a service orchestration is a particular architectural choice for service composition where a central orchestrated process manages the service composition work and data flow invocations the external third party services in the order determined by the work flow. Service orchestrations are specified by suitable orchestration languages and deployed in execution engines who interpret these specifications.

- **Store:** An external component integrated with the business framework offering a set of services that are published to a selected set of marketplaces. The store thereby holds the service portfolio of a specific service provider. In case a specific service is purchased on a service marketplace, the service store handles the actual buying of a specific service (as a financial business transaction).

- **Unified Service Description Language (USDL):** USDL is a platform-neutral language for describing services, covering a variety of service types, such as purely human services, transactional services, informational services, software components, digital media, platform services and infrastructure services. The core set of language modules offers the specification of functional and technical service properties, legal and financial aspects, service levels, interaction information and corresponding participants. USDL is offering extension points for the derivation of domain-specific service description languages by extending or changing the available language modules.

10 FIWARE OpenSpecification Apps MarketplaceSearchREST

You can find the content of this chapter as well in the wiki of fi-ware.

10.1 Introduction to the *Marketplace Search* API

The FI-WARE Generic Enabler Specification are owned by different partners. Therefore, different Legal Notices might apply. Please check for each FI-WARE Generic Enabler Specification the Legal Notice attached. For this FI-WARE Generic Enabler Specification, this Legal Notice applies.

SAP strives to make the specifications of this Generic Enabler available under IPR rules that allow for a exploitation and sustainable usage both in Open Source as well as proprietary, closed source products to maximize adoption.

10.1.1 Marketplace Search Core

The Marketplace Search API is a RESTful, resource-oriented API accessed via HTTP that uses various representations for information interchange. The Marketplace Search Component provides functionality to search the marketplace for concrete offerings.

10.1.2 Intended Audience

This specification is intended for both software developers and reimplementers of this API. For the former, this document provides a full specification of how to interoperate with products that implement the Repository API. For the latter, this specification indicates the interface to be implemented and provided to clients.

To use RESTful information, the reader should firstly have a general understanding of the Generic Enabler service Marketplace. You should also be familiar with:

- ReSTful web services
- HTTP/1.1
- JSON and/or XML data serialization formats.
- RDF, TURTLE and Atom

10.1.3 API Change History

This version of the Marketplace Search API Guide replaces and obsoletes all previous versions. The most recent changes are described in the table below:

Revision Date	Changes Summary
Apr 25, 2012	- Initial version

10.1.4 How to Read This Document

It is assumed that the reader is familiar with the REST architecture style. Within the document, some special notations are applied to differentiate some special words or concepts. The following list summarizes these special notations.

- A bold, mono-spaced font is used to represent code or logical entities, e.g., HTTP method (`GET, PUT, POST, DELETE`).

- An italic font is used to represent document titles or some other kind of special text, e.g., *URI*.

- Variables are represented between brackets, e.g. {id} and in italic font. The reader can replace the id with an appropriate value.

For a description of some terms used along this document, see Marketplace.

10.1.5 Additional Resources

You can download the most current version of this document from the FI-WARE API specification website at Marketplace Search API . For more details about the Marketplace GE that this API is based upon, please refer to High Level Description. Related documents, including an Architectural Description, are available at the same site.

10.2 General *Marketplace Search* API Information

The Marketplace GE is structured into five core components. These components are Registry & Directory, Offering & Demand, Discovery & Matching, Recommendation, and the Review and Rating component. The API for the Discovery & Matching component is described in this document.

10.2.1 Resources Summary

The Discovery and Matching component supports primarily customers finding offerings and stores matching their needs. In both cases the *Marketplace Search* API supports freetext search as well as attributed search.
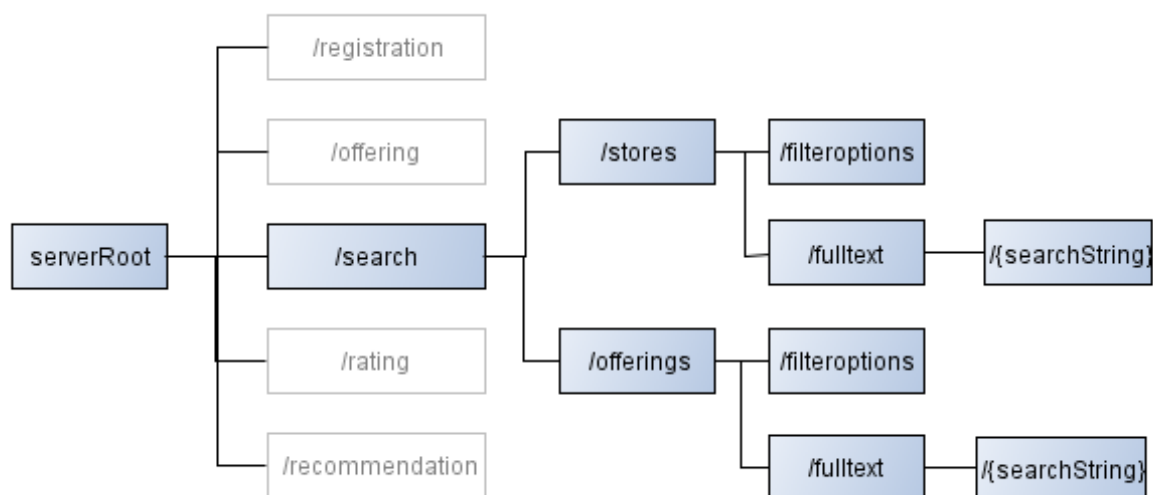
10.2.2 Authentication

Each HTTP request against the *Marketplace Search API* requires the inclusion of specific authentication credentials. The specific implementation of this API may support multiple

authentication schemes (OAuth, Basic Auth, Token) and will be determined by the specific provider that implements the GE. Please contact with it to determine the best way to authenticate against this API. Remember that some authentication schemes may require that the API operate using SSL over HTTP (HTTPS).

10.2.3   Representation Format

The *Marketplace Offering* API supports at least XML and JSON for delivering any kind of resources, it may also support simple text and HTML output format. The request format is specified using the Content-Type header and is required for operations that have a request body. The response format can be specified in requests using the Accept header. Note that it is possible for a response to be serialized using a format different from the request (see example below).

If no Content-Type is specified, the content is delivered in the format that was choosen to upload the resource.

The interfaces should support data exchange through multiple formats:

- *text/plain* - A linefeed separated list of elements for easy mashup and scripting.
- *text/html* - An human-readable HTML rendering of the results of the operation as output format.
- *application/json* - A JSON representation of the input and output for mashups or JavaScript-based Web Apps
- *application/xml* - A XML description of the input and output.

In a concrete implementation of this GE other formats like RSS, Atom, etc. may also be possible.

10.2.4   Representation Transport

Resource representation is transmitted between client and server by using HTTP 1.1 protocol, as defined by IETF RFC-2616. Each time an HTTP request contains payload, a Content-Type header shall be used to specify the MIME type of wrapped representation. In addition, both client and server may use as many HTTP headers as they consider necessary.

10.2.5   Resource Identification

The resource identification for HTTP transport is made using the mechanisms described by HTTP protocol specification as defined by IETF RFC-2616.

10.2.6   Links and References

10.2.6.1   *Web citizen*

The Marketplace Search is relying on Web principles:

- URI to identify resources
- consistent URI structure based on REST style protocol

- HTTP content negotiation to allow the client to choose the appropriate data format supporting HTML, RDF, XML, RSS, JSON, Turtle, ...

- Human readable output format using HTML rendering ('text/html' accept header) including hyperlinked representation

- Use of HTTP response codes including ETags (proper caching)

- Linked Data enablement supporting RDF input and output types

10.2.6.2 *Linked Open Data*

Publishing data as linked data requires every resource to be directly resolvable given their URL. The basic idea of Linked Data is simple. Tim Berners-Lee's note on Linked Data describes four rules for publishing data on the Web:

- Use URIs as names for things

- Use HTTP URIs so that people can look up those names.

- When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL)

- Include links to other URIs, so that they can discover more things.

This can actually achieved by different approaches. One is the use of a special resolver similar to URL shorteners.

So the authoring environment has to ensure that every URI (actually IRI - Internationalized Resource Identifiers - RFC 3987) can be resolved by a HTTP `GET` request. For example: If a resource is maintained in a Marketplace Search under the URL http://marketplaceSearch.acme.com/service/xyz but the IRI used in service descriptions is actually http://fi-ware.org/service/xyz, we need a resolver at this location which redirects the request to the actual Marketplace Search.

Setting up resolvers is more complex task. Therefore we try to follow a simpler approach for the USDLMarketplace SearchRest. The API is designed to be directly used for Linked Data publishing without the need for a resolver.

10.2.7 Paginated Result Lists

In order to reduce the load on the service, we can decide to limit the number of elements to return when it is too big. This section explain how to do that using for example a limit parameter (optional) and a last parameter (optional) to express which is the maximum number of element to return and which was the last element to see. These operations will have to cope with the possibility to have over limit fault (413) or item not found fault (404).

10.2.8 Limits

We can manage the capacity of the system in order to prevent the abuse of the system through some limitations. These limitations will be configured by the operator and may differ from one implementation to other of the GE implementation.

10.2.8.1 *Rate Limits*

These limits are specified both in human readable wild-card and in regular expressions and will indicate for each HTTP verb which will be the maximum number of operations per time

unit that a user can request. After each unit time the counter is initialized again. In the event a request exceeds the thresholds established for your account, a 413 HTTP response will be returned with a Retry-After header to notify the client when they can attempt to try again.

10.2.9   ETag Handling

For standard caching an ETag HTTP header is provided for `GET` and `PUT` requests. If a `GET` requests has a "If-None-Match" header, than the content is only delivered if the stored ETag of the object matches the requested ETag. HTTP status code 304 (not changed) is responded otherwise.

For `PUT` requests the ETag header can be used to ensure integrity of the repository. The `PUT` operation will only be executed if the "If-Match" header matches the stored ETag of the resource in the repository. If no "If-Match" header is given for an existing resource or the "If-Match" header does not match the existing ETag of the resource, status code 409 (Conflict)will be returned. If the resource was changed, then a new ETag header will be returned in the response header.

10.2.10   Extensions

The Marketplace could be extended in the future. At the moment, we foresee the following resource to indicate a method that will be used in order to allow the extensibility of the API. This allow the introduction of new features in the API without requiring an update of the version, for instance, or to allow the introduction of vendor specific functionality.

Verb	URI	Description
GET	/extensions	List of all available extensions

10.2.11   Faults

10.2.11.1 *Synchronous Faults*

Error codes are returned in the body of the response. The description section returns a human-readable message for displaing end users.

Example:

```
<exception>
      <description>Resource Not found</description>
      <errorCode>404</errorCode>
      <reasonPhrase>Not Found</reasonPhrase>
</exception>
```

Fault Element	Associated Error Codes	Expected in All Requests?

Unauthorized	403	YES
Not Found	404	YES
Limit Fault	413	YES
Internal Server error	50X	YES

10.3   API Operations

10.3.1   Search for Offerings

Here we start with the description of the operation following the next table:

Verb	URI	Additional Path Parameters	Description
GET	/search/offerings/fulltext/{searchString}	filter, index, limit, sortBy, order, minScore	search for offerings where the services description matches the specified search string
GET	/search/offerings/filteroptions		returns a list of possible filter options for offering search

A Filter expression, a limit and a starting index are supported for the fulltext search operation to reduce the number of results. Each search result entry has a score value, a minimum threshold score value can be defined to reduce the number of results.

- *filter* - Optional filter expression to reduce the number of delivered results.

- *index* - Index of the first rating to be returned.

- *limit* - Maximal number of results to be returned.

- *order* - Either ascending (asc) or descending (desc).

- *sortBy* - Comma separated list of sort options, sorted by application order. The sort options for this operation are *score*, *storeName*, *serviceName* or *date*.

- *minScore* - Minimum threshold score value of results to be returned.

Example:

```
http://[SERVER_URL]?filter=name:Simth*&index=20&limit=10&order=asc&s
ortBy=name,storeName&minScore=0.75
```

10.3.2   Search for Stores

Here we start with the description of the operation following the next table:

Verb	URI	Additional Path	Description

		Parameters	
GET	/search/stores/fulltext/{searchString}	filter, index, limit, sortBy, order, minScore	search for stores where the store description matches the specified search string
GET	/search/stores/filteroptions		returns a list of possible filter options for store search

A Filter expression, a limit and a starting index are supported for the fulltext search operation to reduce the number of results. Each search result entry has a score value, a minimum threshold score value can be defined to reduce the number of results.

- *filter* - Optional filter expression to reduce the number of delivered results.

- *index* - Index of the first rating to be returned.

- *limit* - Maximal number of results to be returned.

- *order* - Either ascending (asc) or descending (desc).

- *sortBy* - Comma separated list of sort options, sorted by application order. The sort options for this operation are *score*, *storeName* or *registrationDate*.

- *minScore* - Minimum threshold score value of results to be returned.

Example:

```
http://[SERVER_URL]?filter=name:Simth*&index=20&limit=10&order=asc&sortBy=name,storeName&minScore=0.75
```

10.3.3   Status Codes
200 OK

The request was handled successfully and transmitted in response message.

201 Created

The request has been fulfilled and resulted in a new resource being created.

204 No Content

The server successfully processed the request, but is not returning any content.

304 Not Modified

Indicates the resource has not been modified since last requested. Typically, the HTTP client provides a header like the If-Modified-Since header to provide a time against which to compare. Using this saves bandwidth and reprocessing on both the server and client, as only the header data must be sent and received in comparison to the entirety of the page being re-processed by the server, then sent again using more bandwidth of the server and client.

400 Bad Request

The request cannot be fulfilled due to bad syntax.

404 Not Found

The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible.

409 Conflict

Indicates that the request could not be processed because of conflict in the request, such as an edit conflict.

500 Internal Server Error

A generic error message, given when no more specific message is suitable.

11 FIWARE OpenSpecification Apps MarketplaceOfferingsREST

You can find the content of this chapter as well in the wiki of fi-ware.

11.1 Introduction to the *Marketplace Offering* API

The FI-WARE Generic Enabler Specification are owned by different partners. Therefore, different Legal Notices might apply. Please check for each FI-WARE Generic Enabler Specification the Legal Notice attached. For this FI-WARE Generic Enabler Specification, this Legal Notice applies.

SAP strives to make the specifications of this Generic Enabler available under IPR rules that allow for a exploitation and sustainable usage both in Open Source as well as proprietary, closed source products to maximize adoption.

11.1.1 Marketplace Offering Core

The Marketplace Offering API is a RESTful, resource-oriented API accessed via HTTP that uses various representations for information interchange. The Marketplace Offerings Component is responsible for exchanging service offerings with stores and making them public to potential customers.

11.1.2 Intended Audience

This specification is intended for both software developers and reimplementers of this API. For the former, this document provides a full specification of how to interoperate with products that implement the Repository API. For the latter, this specification indicates the interface to be implemented and provided to clients.

To use this information, the reader should firstly have a general understanding of the Generic Enabler service Marketplace. You should also be familiar with:

- RESTful web services
- HTTP/1.1
- JSON and/or XML data serialization formats.
- RDF, TURTLE and Atom

11.1.3 API Change History

This version of the Marketplace Offering API Guide replaces and obsoletes all previous versions. The most recent changes are described in the table below:

Revision Date	Changes Summary
Apr 25, 2012	• Initial version

11.1.4   How to Read This Document

It is assumed that the reader is familiar with the REST architecture style. Within the document, some special notations are applied to differentiate some special words or concepts. The following list summarizes these special notations.

- A bold, mono-spaced font is used to represent code or logical entities, e.g., HTTP method (`GET, PUT, POST, DELETE`).

- An italic font is used to represent document titles or some other kind of special text, e.g., *URI.*

- Variables are represented between brackets, e.g. {id} and in italic font. The reader can replace the id with an appropriate value.

For a description of some terms used along this document, see Marketplace.

11.1.5   Additional Resources

You can download the most current version of this document from the FI-WARE API specification website at Marketplace Offerings API . For more details about the Marketplace GE that this API is based upon, please refer to High Level Description. Related documents, including an Architectural Description, are available at the same site.

11.2   General *Marketplace Offering* API Information

The Marketplace GE is structured into five core components. These components are Registry & Directory, Offering & Demand, Discovery & Matching, Recommendation, and the Review and Rating component. The API for the Offering & Demand component is described in this document.

11.2.1   Resources Summary

A service offering consists of a link to a concrete USDL description, a pricing model and the classification of the service. The Offering component is responsible for exchanging service offerings with stores and version handling/archiving of out-dated offerings. Symmetrically to offerings also the demand side of the marktet need to be represented. A service demand according to expected functionality, pricing and service levels might be expressed, classified and published to the marketplace.
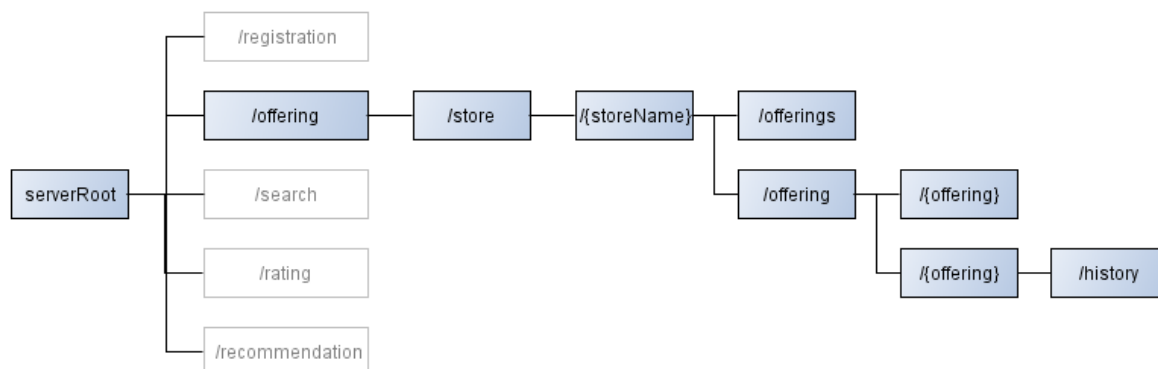
11.2.2 Authentication

Each HTTP request against the *Marketplace Offering API* requires the inclusion of specific authentication credentials. The specific implementation of this API may support multiple authentication schemes (OAuth, Basic Auth, Token) and will be determined by the specific provider that implements the GE. Please contact with it to determine the best way to authenticate against this API. Remember that some authentication schemes may require that the API operate using SSL over HTTP (HTTPS).

11.2.3 Representation Format

The *Marketplace Offering* API supports at least XML and JSON for delivering any kind of resources, it may also support simple text and HTML output format. The request format is specified using the Content-Type header and is required for operations that have a request body. The response format can be specified in requests using the Accept header. Note that it is possible for a response to be serialized using a format different from the request (see example below).

If no Content-Type is specified, the content is delivered in the format that was choosen to upload the resource.

The interfaces should support data exchange through multiple formats:

- *text/plain* - A linefeed separated list of elements for easy mashup and scripting.
- *text/html* - An human-readable HTML rendering of the results of the operation as output format.
- *application/json* - A JSON representation of the input and output for mashups or JavaScript-based Web Apps
- *application/xml* - A XML description of the input and output.

In a concrete implementation of this GE other formats like RSS, Atom, etc. may also be possible.

11.2.4 Representation Transport

Resource representation is transmitted between client and server by using HTTP 1.1 protocol, as defined by IETF RFC-2616. Each time an HTTP request contains payload, a Content-Type header shall be used to specify the MIME type of wrapped representation. In addition, both client and server may use as many HTTP headers as they consider necessary.

11.2.5 Resource Identification

The resource identification for HTTP transport is made using the mechanisms described by HTTP protocol specification as defined by IETF RFC-2616.

11.2.6 Links and References

11.2.6.1 *Web citizen*

The Marketplace Offering is relying on Web principles:

- URI to identify resources

- consistent URI structure based on REST style protocol

- HTTP content negotiation to allow the client to choose the appropriate data format supporting HTML, RDF, XML, RSS, JSON, Turtle, ...

- Human readable output format using HTML rendering ('text/html' accept header) including hyperlinked representation

- Use of HTTP response codes including ETags (proper caching)

- Linked Data enablement supporting RDF input and output types

11.2.6.2 *Linked Open Data*

Publishing data as linked data requires every resource to be directly resolvable given their URL. The basic idea of Linked Data is simple. Tim Berners-Lee's note on Linked Data describes four rules for publishing data on the Web:

- Use URIs as names for things

- Use HTTP URIs so that people can look up those names.

- When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL)

- Include links to other URIs, so that they can discover more things.

This can actually achieved by different approaches. One is the use of a special resolver similar to URL shorteners.

So the authoring environment has to ensure that every URI (actually IRI - Internationalized Resource Identifiers - RFC 3987) can be resolved by a HTTP `GET` request. For example: If a resource is maintained in a Marketplace Offering under the URL http://marketplaceOffering.acme.com/service/xyz but the IRI used in service descriptions is actually http://fi-ware.org/service/xyz, we need a resolver at this location which redirects the request to the actual Marketplace Offering.

Setting up resolvers is more complex task. Therefore we try to follow a simpler approach for the USDLMarketplace OfferingRest. The API is designed to be directly used for Linked Data publishing without the need for a resolver.

11.2.7 Paginated Result Lists

In order to reduce the load on the service, we can decide to limit the number of elements to return when it is too big. This section explain how to do that using for example a limit parameter (optional) and a last parameter (optional) to express which is the maximum number of element to return and which was the last element to see.

These operations will have to cope with the possibility to have over limit fault (413) or item not found fault (404).

11.2.8 Limits

We can manage the capacity of the system in order to prevent the abuse of the system through some limitations. These limitations will be configured by the operator and may differ from one implementation to other of the GE implementation.

11.2.8.1 *Rate Limits*

These limits are specified both in human readable wild-card and in regular expressions and will indicate for each HTTP verb which will be the maximum number of operations per time unit that a user can request. After each unit time the counter is initialized again. In the event a request exceeds the thresholds established for your account, a 413 HTTP response will be returned with a Retry-After header to notify the client when they can attempt to try again.

11.2.9 ETag Handling

For standard caching an ETag HTTP header is provided for `GET` and `PUT` requests. If a `GET` requests has a "If-None-Match" header, than the content is only delivered if the stored ETag of the object matches the requested ETag. HTTP status code 304 (not changed) is responded otherwise.

For `PUT` requests the ETag header can be used to ensure integrity of the repository. The `PUT` operation will only be executed if the "If-Match" header matches the stored ETag of the resource in the repository. If no "If-Match" header is given for an existing resource or the "If-Match" header does not match the existing ETag of the resource, status code 409 (Conflict)will be returned. If the resource was changed, then a new ETag header will be returned in the response header.

11.2.10 Extensions

The Marketplace could be extended in the future. At the moment, we foresee the following resource to indicate a method that will be used in order to allow the extensibility of the API. This allow the introduction of new features in the API without requiring an update of the version, for instance, or to allow the introduction of vendor specific functionality.

Verb	URI	Description
GET	/extensions	List of all available extensions

11.2.11 Faults

11.2.11.1 *Synchronous Faults*

Error codes are returned in the body of the response. The description section returns a human-readable message for displaing end users.

Example:

```
<exception>
      <description>Resource Not found</description>
      <errorCode>404</errorCode>
      <reasonPhrase>Not Found</reasonPhrase>
</exception>
```

Fault Element	Associated Error Codes	Expected in All Requests?

Unauthorized	403	YES
Not Found	404	YES
Limit Fault	413	YES
Internal Server error	50X	YES

11.3   API Operations

11.3.1   Managing Offerings

Here we start with the description of the operation following the next table:

Verb	URI	Additional Path Parameters	Description
GET	/offering/	filter, index, limit	Get a list of all offerings
GET	/offering/store/{storeName}/offerings	filter, index, limit	Get a list of all offerings from a specific store
GET	/offering/store/{storeName}/offering/{offering}	-	Get a specific offering
GET	/offering/store/{storeName}/offering/{offering}/history	-	Get the history of a specific offering
PUT	/offering/store/{storeName}/offering/{offering}	-	Create a new offering
POST	/offering/store/{storeName}/offering/{offering}	-	Update offering information, creates a new version
DELETE	/offering/store/{storeName}/offering/{offering}	-	disables an offering

A Filter expression, a limit and a starting index are supported for the */offerings/* and the */offerings/store/{storeName}/offerings* operation to reduce the number of results. If no filter expression is given then all users are returned.

- *filter* - Optional filter expression to reduce the number of delivered results.
- *index* - Index of the first rating to be returned.
- *limit* - Maximal number of results to be returned.

Example:

```
http://[SERVER_URL]?filter=name:Simth*&index=20&limit=10
```

11.3.2   Status Codes

200 OK

The request was handled successfully and transmitted in response message.

201 Created

The request has been fulfilled and resulted in a new resource being created.

204 No Content

The server successfully processed the request, but is not returning any content.

304 Not Modified

Indicates the resource has not been modified since last requested. Typically, the HTTP client provides a header like the If-Modified-Since header to provide a time against which to compare. Using this saves bandwidth and reprocessing on both the server and client, as only the header data must be sent and received in comparison to the entirety of the page being re-processed by the server, then sent again using more bandwidth of the server and client.

400 Bad Request

The request cannot be fulfilled due to bad syntax.

404 Not Found

The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible.

409 Conflict

Indicates that the request could not be processed because of conflict in the request, such as an edit conflict.

500 Internal Server Error

A generic error message, given when no more specific message is suitable.

12 FIWARE OpenSpecification Apps MarketplaceRegistrationREST

You can find the content of this chapter as well in the wiki of fi-ware.

12.1 Introduction to the *Marketplace Registration* API

The FI-WARE Generic Enabler Specification are owned by different partners. Therefore, different Legal Notices might apply. Please check for each FI-WARE Generic Enabler Specification the Legal Notice attached. For this FI-WARE Generic Enabler Specification, this Legal Notice applies.

SAP strives to make the specifications of this Generic Enabler available under IPR rules that allow for a exploitation and sustainable usage both in Open Source as well as proprietary, closed source products to maximize adoption.

12.1.1 Marketplace Registration Core

The Marketplace Registration is a RESTful, resource-oriented API accessed via HTTP that uses various representations for information interchange. The Marketplace Registration Component takes care of registering, updating, and deleting information about market relevant entities such as stores and marketplace participants.

12.1.2 Intended Audience

This specification is intended for both software developers and reimplementers of this API. For the former, this document provides a full specification of how to interoperate with products that implement the Repository API. For the latter, this specification indicates the interface to be implemented and provided to clients.

To use this information, the reader should firstly have a general understanding of the Generic Enabler service Marketplace. You should also be familiar with:

- RESTful web services
- HTTP/1.1
- JSON and/or XML data serialization formats.
- RDF, TURTLE and Atom

12.1.3 API Change History

This version of the Marketplace Registration API Guide replaces and obsoletes all previous versions. The most recent changes are described in the table below:

Revision Date	Changes Summary
Apr 25, 2012	• Initial version

12.1.4   How to Read This Document

It is assumed that the reader is familiar with the REST architecture style. Within the document, some special notations are applied to differentiate some special words or concepts. The following list summarizes these special notations.

- A bold, mono-spaced font is used to represent code or logical entities, e.g., HTTP method (`GET, PUT, POST, DELETE`).

- An italic font is used to represent document titles or some other kind of special text, e.g., *URI.*

- Variables are represented between brackets, e.g. {id} and in italic font. The reader can replace the id with an appropriate value.

For a description of some terms used along this document, see [Marketplace](#).

12.1.5   Additional Resources

You can download the most current version of this document from the FI-WARE API specification website at [Marketplace Registration API](#). For more details about the Marketplace GE that this API is based upon, please refer to [High Level Description](#). Related documents, including an Architectural Description, are available at the same site.

12.2   General *Marketplace Registration* API Information

The Marketplace GE is structured into five core components. These components are Registry & Directory, Offering & Demand, Discovery & Matching, Recommendation, and the Review and Rating component. The API for the Registry & Directory component is described in this document.

12.2.1   Resources Summary

The Registry and Directory component holds information of registered stores, participants and their role(vendors, buyers, resellers, ...) and takes care of registering, updating, and deleting information about market relevant entities.
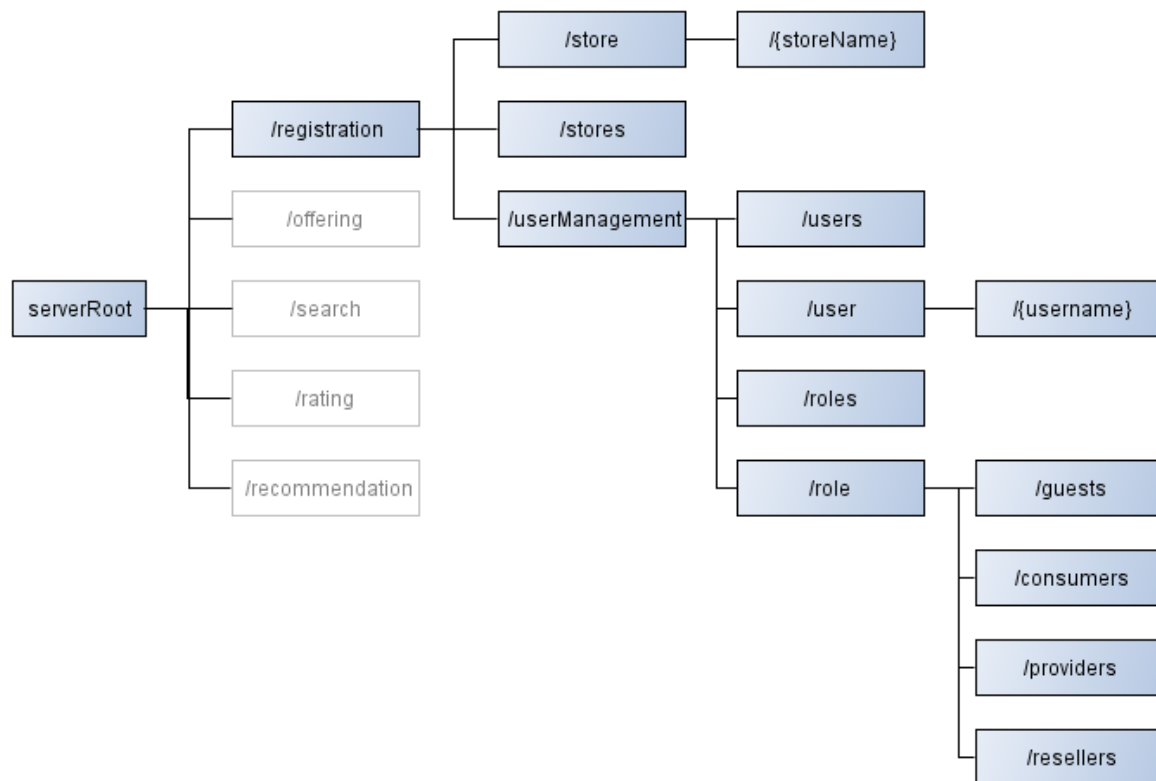
12.2.2 Authentication

Each HTTP request against the *Marketplace Registration API* requires the inclusion of specific authentication credentials. The specific implementation of this API may support multiple authentication schemes (OAuth, Basic Auth, Token) and will be determined by the specific provider that implements the GE. Please contact with it to determine the best way to authenticate against this API. Remember that some authentication schemes may require that the API operate using SSL over HTTP (HTTPS).

12.2.3 Representation Format

The *Marketplace Offering* API supports at least XML and JSON for delivering any kind of resources, it may also support simple text and HTML output format. The request format is specified using the Content-Type header and is required for operations that have a request body. The response format can be specified in requests using the Accept header. Note that it is possible for a response to be serialized using a format different from the request (see example below).

If no Content-Type is specified, the content is delivered in the format that was choosen to upload the resource.

The interfaces should support data exchange through multiple formats:

- *text/plain* - A linefeed separated list of elements for easy mashup and scripting.

- *text/html* - An human-readable HTML rendering of the results of the operation as output format.

- *application/json* - A JSON representation of the input and output for mashups or JavaScript-based Web Apps

- *application/xml* - A XML description of the input and output.

In a concrete implementation of this GE other formats like RSS, Atom, etc. may also be possible.

12.2.4   Representation Transport

Resource representation is transmitted between client and server by using HTTP 1.1 protocol, as defined by IETF RFC-2616. Each time an HTTP request contains payload, a Content-Type header shall be used to specify the MIME type of wrapped representation. In addition, both client and server may use as many HTTP headers as they consider necessary.

12.2.5   Resource Identification

The resource identification for HTTP transport is made using the mechanisms described by HTTP protocol specification as defined by IETF RFC-2616.

12.2.6   Links and References

12.2.6.1   *Web citizen*

The Marketplace Registration is relying on Web principles:

- URI to identify resources

- consistent URI structure based on REST style protocol

- HTTP content negotiation to allow the client to choose the appropriate data format supporting HTML, RDF, XML, RSS, JSON, Turtle, ...

- Human readable output format using HTML rendering ('text/html' accept header) including hyperlinked representation

- Use of HTTP response codes including ETags (proper caching)

- Linked Data enablement supporting RDF input and output types

12.2.6.2   *Linked Open Data*

Publishing data as linked data requires every resource to be directly resolvable given their URL. The basic idea of Linked Data is simple. Tim Berners-Lee's note on Linked Data describes four rules for publishing data on the Web:

- Use URIs as names for things

- Use HTTP URIs so that people can look up those names.

- When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL)

- Include links to other URIs, so that they can discover more things.

This can actually achieved by different approaches. One is the use of a special resolver similar to URL shorteners.

So the authoring environment has to ensure that every URI (actually IRI - Internationalized Resource Identifiers - RFC 3987) can be resolved by a HTTP `GET` request. For example: If a resource is maintained in a Marketplace Registration under the URL http://marketplaceRegistration.acme.com/service/xyz but the IRI used in service descriptions is actually http://fi-ware.org/service/xyz, we need a resolver at this location which redirects the request to the actual Marketplace Registration.

Setting up resolvers is more complex task. Therefore we try to follow a simpler approach for the USDLMarketplace RegistrationRest. The API is designed to be directly used for Linked Data publishing without the need for a resolver.

12.2.7 Paginated Result Lists

In order to reduce the load on the service, we can decide to limit the number of elements to return when it is too big. This section explain how to do that using for example a limit parameter (optional) and a last parameter (optional) to express which is the maximum number of element to return and which was the last element to see.

These operations will have to cope with the possibility to have over limit fault (413) or item not found fault (404).

12.2.8 Limits

We can manage the capacity of the system in order to prevent the abuse of the system through some limitations. These limitations will be configured by the operator and may differ from one implementation to other of the GE implementation.

12.2.8.1 *Rate Limits*

These limits are specified both in human readable wild-card and in regular expressions and will indicate for each HTTP verb which will be the maximum number of operations per time unit that a user can request. After each unit time the counter is initialized again. In the event a request exceeds the thresholds established for your account, a 413 HTTP response will be returned with a Retry-After header to notify the client when they can attempt to try again.

12.2.9 ETag Handling

For standard caching an ETag HTTP header is provided for `GET` and `PUT` requests. If a `GET` requests has a "If-None-Match" header, than the content is only delivered if the stored ETag of the object matches the requested ETag. HTTP status code 304 (not changed) is responded otherwise.

For `PUT` requests the ETag header can be used to ensure integrity of the repository. The `PUT` operation will only be executed if the "If-Match" header matches the stored ETag of the resource in the repository. If no "If-Match" header is given for an existing resource or the "If-Match" header does not match the existing ETag of the resource, status code 409 (Conflict)will be returned. If the resource was changed, then a new ETag header will be returned in the response header.

12.2.10 Extensions

The Marketplace could be extended in the future. At the moment, we foresee the following resource to indicate a method that will be used in order to allow the extensibility of the API.

This allow the introduction of new features in the API without requiring an update of the version, for instance, or to allow the introduction of vendor specific functionality.

Verb	URI	Description
GET	/extensions	List of all available extensions

12.2.11 Faults

12.2.11.1 *Synchronous Faults*

Error codes are returned in the body of the response. The description section returns a human-readable message for displaing end users.

Example:

```
<exception>
        <description>Resource Not found</description>
        <errorCode>404</errorCode>
        <reasonPhrase>Not Found</reasonPhrase>
</exception>
```

Fault Element	Associated Error Codes	Expected in All Requests?
Unauthorized	403	YES
Not Found	404	YES
Limit Fault	413	YES
Internal Server error	50X	YES

12.3   API Operations

12.3.1   Managing Stores

Here we start with the description of the operation following the next table:

Verb	URI	Additional Path Parameters	Description
GET	/registration/stores/	filter, index, limit	Get a list of all registered stores
GET	/registration/store/{StoreName}	-	Get a specific store
PUT	/registration/store/{StoreName}	-	Create a store
POST	/registration/store/{StoreName}	-	Update store information
DELETE	/registration/store/{StoreName}	-	Unregister a store

A Filter expression, a limit and a starting index are supported for the */registration/stores/* operation to reduce the number of results. If no filter expression is given then all users are returned.

- *filter* - Optional filter expression to reduce the number of delivered results.
- *index* - Index of the first rating to be returned.
- *limit* - Maximal number of results to be returned.

Example:

```
http://[SERVER_URL]?filter=name:Simth*&index=20&limit=10
```

12.3.2   Managing Marketplace Participants

12.3.2.1   *Marketplace Participant*

Verb	URI	Additional Path Parameters	Description
GET	/registration/users/	filter, index, limit	Get a list of all registered marketplace participants
GET	/registration/user/{username}	-	Get a specific marketplace participant
PUT	/registration/user/{username}	-	Create a marketplace participant
POST	/registration/user/{username}	-	Update a marketplace participant
DELETE	/registration/user/{username}	-	Unregister a marketplace participant

A Filter expression, a limit and a starting index are supported for the */userManagement/users/* operation to reduce the number of results. If no filter expression is given then all users are returned.

- *filter* - Optional filter expression to reduce the number of delivered results.
- *index* - Index of the first rating to be returned.
- *limit* - Maximal number of results to be returned.

Example:

```
http://[SERVER_URL]?filter=name:Simth*&index=20&limit=10
```

12.3.2.2   *Marketplace Participants by Role*

Verb	URI	Additional Path	Description

		Parameters	
GET	/registration/roles/	filter, index, limit	Get a list of all marketplace participant roles
GET	/registration/roles?role=guest	filter, index, limit	Get all marketplace participants with the role guest
GET	/registration/roles?role=consumer	filter, index, limit	Get all marketplace participants with the role consumer
GET	/registration/roles?role=provider	filter, index, limit	Get all marketplace participants with the role provider
GET	/registration/roles?role=reseller	filter, index, limit	Get all marketplace participants with the role reseller

A Filter expression, a limit and a starting index are supported for all *Marketplace Participants by Role* operations to reduce the number of results. If no filter expression is given then all users are returned.

- *filter* - Optional filter expression to reduce the number of delivered results.

- *index* - Index of the first rating to be returned.

- *limit* - Maximal number of results to be returned.

Example:

```
http://[SERVER_URL]?filter=name:Simth*&index=20&limit=10
```

12.3.3   Status Codes

200 OK

The request was handled successfully and transmitted in response message.

201 Created

The request has been fulfilled and resulted in a new resource being created.

204 No Content

The server successfully processed the request, but is not returning any content.

304 Not Modified

Indicates the resource has not been modified since last requested. Typically, the HTTP client provides a header like the If-Modified-Since header to provide a time against which to compare. Using this saves bandwidth and reprocessing on both the server and client, as only the header data must be sent and received in comparison to the entirety of the page being re-processed by the server, then sent again using more bandwidth of the server and client.

400 Bad Request

The request cannot be fulfilled due to bad syntax.

404 Not Found

The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible.

409 Conflict

Indicates that the request could not be processed because of conflict in the request, such as an edit conflict.

500 Internal Server Error

A generic error message, given when no more specific message is suitable.

13 FIWARE OpenSpecification Apps Repository

You can find the content of this chapter as well in the wiki of fi-ware.

Name	FIWARE.OpenSpecification.Apps.Repository
Chapter	Apps,
Catalogue-Link to Implementation	Service Description Repository
Owner	SAP AG, Torsten Leidig

13.1 Preface

Within this document you find a self-contained open specification of a FI-WARE generic enabler, please consult as well the FI-WARE_Product_Vision, the website on http://www.fi-ware.eu and similar pages in order to understand the complete context of the FI-WARE project.

13.2 Copyright

- Copyright © 2012 by SAP

13.3 Legal Notice

Please check the following Legal Notice to understand the rights to use these specifications.

13.4 Overview

Together with the Registry and the Marketplace, the Repository is a core enabler of the FI-Ware Business Framework. The repository provides a consistent uniform API to USDL service descriptions and associated media files for applications of the business framework. A service provider can use the Repository to publish the description of various aspects of the service according to a unified description language. Wheras the Repository is used to publish service descriptions (service models), the Registry is used for storing runtime information about concrete instances and their configuration settings.

USDL is used in its Linked Data version "Linked USDL". Documentation can be found at <http://linked-usdl.org/> . Information about the FI-Ware Platform is available at https://forge.fi-ware.eu/plugins/mediawiki/wiki/fiware/index.php/Main_Page . USDL describes services on a metadata level and can refer to supplemental resources of any media type. Therefore the repository must be able to store resources in arbitrary formats. The RDF datamodel of USDL allows to refer to entities of the service description via the resource URL. Therefore Linked-USDL is already well prepared to allow the distribution of service descriptions all over the Internet.

13.4.1   Target usage

The Repository is a place to store service models, especially USDL descriptions but also other models required by components of the overall delivery framework (e.g. technical models for service composition and mashup). The repository provides a common location for storage (centrally or distributed and replicated), reference and/or safety.

The use of a repository is required in order to appear at the marketplace or other tools referring to a number of central repositories for information relevant for interoperation of the enablers and roles within the FI-Ware platform. The repository contains published descriptions which can be utilized by any component in respect to privacy and authorization constraints imposed by the business models. Usually a repository is under control of an authority and usually is keeping track of versions, authenticity and publication dates.

13.4.1.1   *User roles*

- The Provider creates services and has an original description describing basic service information as well as technical information. He needs to upload and publish service descriptions on the repository in order to make them available to other components of the platform, such as the Shops/Stores, Aggregators, etc.

- The Aggregator will use for example technical and service-level information of existing services in the repository in order to do a compositon of a value added service or application. So for example, in order to give a valid statement about the availability of the new composed service the availability of each contained service needs to be regarded. The Aggregator also needs information about the technical interfaces of a service in order to develop code to call them correctly . Service descriptions for the newly created composite service can be uploaded and published to the repository again.

- The Broker needs all kind of business relevant descriptions of services, such as general descriptions, business partners, service-levels, and pricing, to be presented in the shop/store. Also technical information can be required, on a level to be able to do comparisons between services for the consumer.

- The Channel Maker needs detailed information about the channel to ensure the proper channel creation or selection. Further a channel may require embedding or wrapping the service so it can be accessed by the user through the specific channel. Various channels and devices such as Web (browser), Android, iOS but also global as well as local social networking and community platforms such as Facebook, LinkedIn, MySpace, Xing, KWICK! might be supported.

- The Hoster requires information on service-level descriptions, deployment and hosting platform requirements to provide the necessary infrastructure in a reliable and scalable way.

- The Gateway will use information about technical interfaces to provide data, protocol and process mediation services. The gateway also provides services for mediation towards premise systems outside of the FI-Ware platform.

13.5   Basic Concepts

13.5.1   Web Citizen

The repository is relying on Web principles:

- URI to identify resources

- consistent URI structure based on REST style protocol

- HTTP content negotiation to allow the client to choose the appropriate data format supporting HTML, RDF, XML, RSS, JSON, Turtle, ...

- Human readable output format using HTML rendering ('text/html' accept header) including hyperlinked representation

- Use of HTTP response codes including ETags (proper caching)

- Linked Data enablement supporting RDF input and output types

13.5.2   Open Distributed Architecture

The Repository Open Specification has to be seen as a specification of the repository abstract functionality. There can be many technologies used to implement the functionality. Often the repository protocol is implemented on top of a Web content management system.

Also we envision a large number of repositories containing service descriptions, which also might to refer to descriptions in other repositories. Repositories can be hosted by the provider or a provider may use repository services of platform providers. The latter might be an alternative for small sized providers, which don't want to provide an own infrastructure.

The service descriptions in a repository are typically used by different other components of the platform, such as service stores or marketplaces, by extracting information needed for the specific functionality.

13.5.3   Data Model

The repository is structured into core objects, which are *resources* and *collections*. These objects constitute also the granularity of access control. Collections are containers for storing resources, which are typically used to maintain all resources belonging to a certain service description in one place.

13.5.3.1   *Resources*

The resources are mainly the USDL service descriptions themselves as well as complementary media files that are used within the service descriptions.

13.5.3.2   *Collections*

A collection is a container for collecting resources. Multiple collections can be used on the repository for various purposes. Collections can be nested and may provide versioning of the resources. Collactions are used to keep all content that is locally referred from the service descriptions together in one place. For example a service description often has additional documentation, depictions and other collateral information, which can be bundled together in one collection.

13.5.3.3 *Recipes*

Recipes are virtual containers selecting resources from different collections.

13.5.4   Content Negotiation

For optimal interoperability and flexible use, the repository should be able to deliver the results of an operation in multiple formats. HTTP content negotiations should be used to let the client choose an appropriate content type. Basic content types (mime-type) are

- HTML - to deliver the results in hyper-linked HTML that can be rendered directly in a Web Browser

- RDF - various RDF serializations for processing in applications

- JSON - Javascript Object Notation for easy processing in a mashup environment.

13.6   Repository Architecture

The repository GE is used by various other GE within the FI-Ware platform. Namely Marketplace, Store, Composition Environment as well as SLA monitoring and Revenue Sharing can access repositories to retrieve detailed information about a service or application. The composition environment for example can retrieve available service offerings for composition from the marketplace. In order to get detailed information about the respective services, the repository API is used. Finished composite services or applications in turn can be described in Linked USDL and published in a repository. New offerings for the service can be posted to the marketplace. Similarly the mediation GE can get details about a service to be mediated from the repository and push back mediator proxy services for a complex mediation type, to be reused by many applications. The repository is also used to store business models according to composite services and applications, which will be used by the business model execution environment and revenue sharing system.
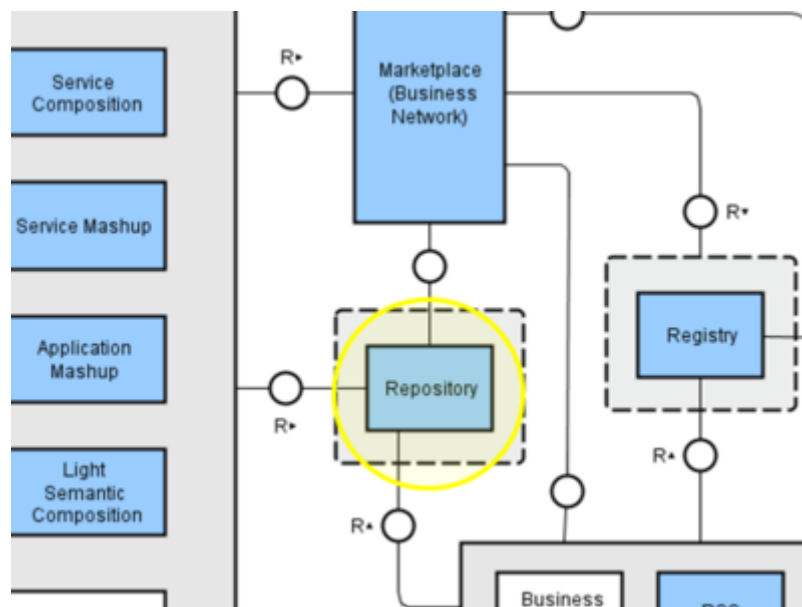
Repository in the context of the Business Framework

Besides the FI-Ware platform also Future Internet applications or composite services on top of the FI-Ware platform can use the repository as a service for their own purpose. An example of the inner architecture of the Repository is shown in the following diagram.
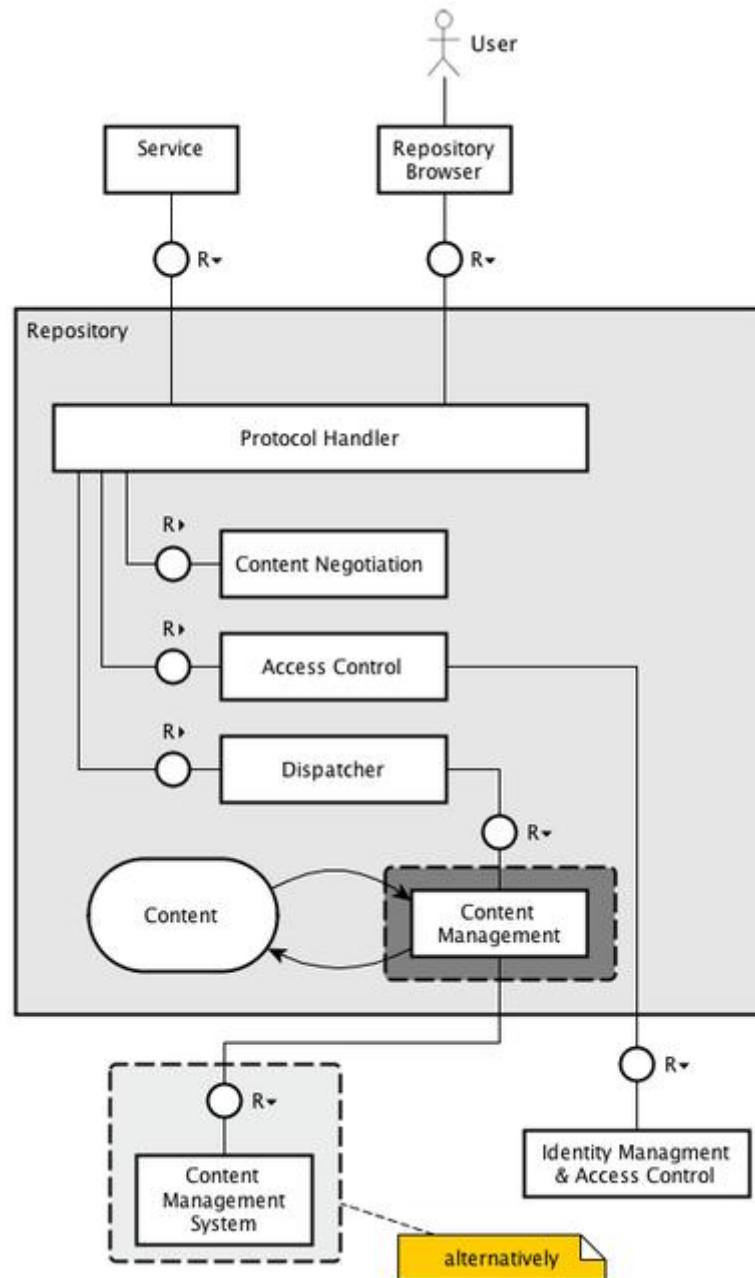
Example high-level architecture of a repository implementation

The architecture shown here is only a blueprint for possible implementations of the repository and show the functional components, necessary to realize this functionality. There are many technology options for a concrete implementation, depending on the context and application domain and its nonfunctional requirements. Since the requirements according to repository size, workload and other parameters can be quite different, there is no obvious all-encompassing implementation solution. The implementations can span very simple ones, which provide only few extensions to a standards Web service to very sophisticated ones that utilizes enterprise content management systems (e.g. based on the "CMIS - Content Management Interoperability Services" standard).

The repository only stores and provides access to serice descriptions. Since there is no common standard for versioning, and the requirement according to versioning may vary

depending on the use case scenario. We do not require version control from a repository implementation, although a real implementation can provide versioning models and mechanism (e.g. using the capabilities of the underlying CMS system).

Also there is no requirement regarding consistency checking of the service descriptions in the repository. The applications themselves have to ensure that the descriptions are consistent. All clients of a repository have cope with incomplete and inconsistent information by default. This reflects the architecture of the Web, where also no consistency commitment of the pages on different Web servers can be made. To ensure integrity additional measures have to be taken.

13.6.1   Technical Interfaces

- FIWARE.Interface.Apps.USDLRepositoryRest - A very simple REST based protocol based on plain HTTP.

13.7   Main Operations

The Repository operation protocol is kept very simple. It basically provides operations to get and put resources, such as service descriptions and media content. Additional operations are used to structure the repository into collections of resources.

13.7.1   Managing Resources

The core functionality of a repository is to store resources and retrieve them when necessary. Further resources sometimes need to be updated and eventually deleted. The following diagram shows an example sequence of resource management operations of a repository.
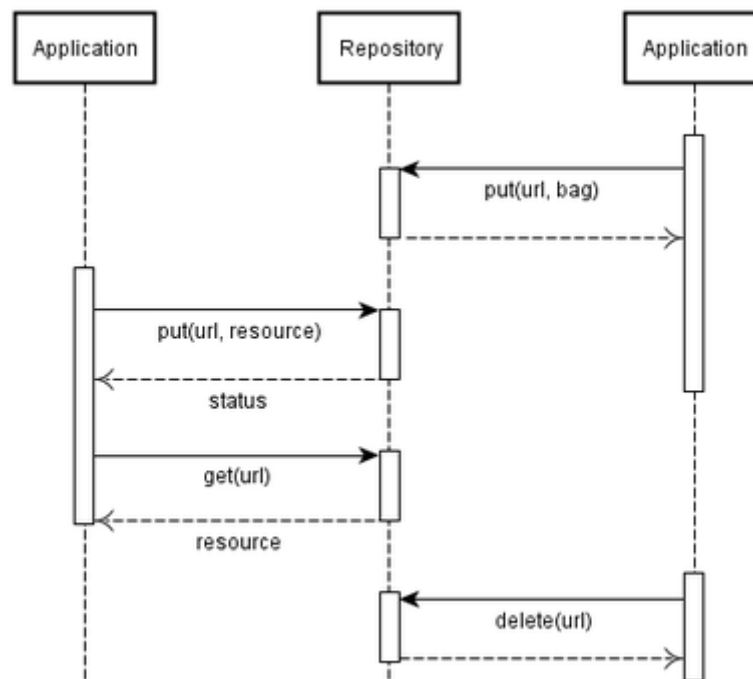
Example sequence of resource management operations

The **Get Resource** operation can be used to retrieve a resource from the repository. This operation delivers the actual content of the resource and/or metadata about the resource, such as the media type, creator, or modification date, depending on the used technical interface. The following parameters need to be exchanged:

- *resource identifier* - Resource identifier of the resource to be returned.

If only information about collections is requested, the collection identifier is used instead of the resource identifier.

- *collection identifier* - Identifier for the collection, which contains the resource.

- *resource* - Resource which will be returned

- *media type* - Media type of the resource which will be returned.

The **Put Resource** operation is used to store a new resource into the repository or update an existing resource with the same resource identifier. The repository should take precautions to provide inconsistent changes due to concurrent access. The following parameters need to be exchanged:

- *resource identifier* - Identifier which contains the resource.

- *collection identifier* - Identifier which denotes the collection into which the resource will be put. The collection can be a part of the resource identifier, if for example URL paths are used to identify a resource.

- *resource* - the content of the resource to be stored into the repository

In order to delete a resource irrevocably from the repository the **Delete Resource** operation is used. The following parameters are exchanged:

- *resource identifier* - Resource identifier of the resource to be deleted.

13.7.2   Managing Collections

Collections are used to put a structure into the repository. In order to easily access parts of the repository, it allows clients to get information about the contents of individual collections.

The **Create Collection** operation creates a new collection in the repository, containing the necessary details such as owner, policies, and other metadata attributes. It requires the parameter:

- *description* - Description of the collection to be created, which contains the location path within the repository and administrative data such as creator and access policies.

To get the details and contents of a collection the **Get Collection** operation is used. The collection information contains information such as owner, policies, textual descriptions, dates, versions, number of resources, and more. The level of detail of the description may depend on the authorization level of the requester. The following parameters can be involved in the operation:

- *collection identifier* - Collection identifier for which a description is to be returned.

- *filter* - Optional filter expression to select the properties to be filtered.

- *description* - Returned collection description containing information according to the filter expression.

A collection in the repository can be deleted with the **Delete Collection** operation. This operation can only be successful for a requester that has the appropriate authorization. The delete operation requires the identifier of the collection as input. After this operation the collection is no longer accessible for clients. Only one parameter is necessary:

- *collection* - Collection identifier for the collection to be deleted

13.7.3   Listing Content

The **List** operation lists collections and/or resources contained in the repository, which are accessible by the user. This operation usually is needed for a repository browser and maintenance tool as well as an editor tool in order to select the resource to be maintained.

The operations using the following parameters: No input parameters are required. However, for practical reasons it might be useful to restrict the list of collections and resources by specifying the number of results and the starting offset.

- *collection* - Collection for which the list is restricted to.

- *index* - Index of the first element of the result set to be returned.

- *limit* - Maximal number of results to be returned.

- *filter* - A repository implementation might also offer the possibility to filter the list of collections according to specific criteria. An optional filter expression can be used to reduce the number of delivered results. A repository may support different criteria to filter the output

- *result list* - The operation results in a list of collections/resources that is returned to the client and contains resource descriptions according to the collection, filter, index, and limit expressions.

13.7.4   List the additional services

Besides the operations described above, a repository might provide additional services, such as search, backup, etc. A repository should list and describe the additional to the clients when the **List Services** operation is invoked. If additional services are offered only for specific collections of the Repository, the collection identifier can be used to list the actual available services for this collection. The required parameters for this operation are:

- *collection* - Collection identifier for which the services will be listed.

- *result list* - List of service descriptions for available services returned to the client.

13.7.5   Searching the Repository

A Repository might provide searching service, to search service descriptions according to the occurrence search terms in properties of the description and media content. It is desireable that in compliance to OpenSearch the repository provides an OpenSearch description to the search API.

13.7.6   Querying the Repository

A repository might also provide a more complex querying service in a specifc query language. As an example a query service based on SPARQL [REP2]would allow to execute complex queries on the Linked Data RDF model [REP1].

13.8   Basic Design Principles

13.8.1   Rationale

There are many proprietary solutions implementing repository functionality and also many standards for various types of repositories. Within FI-Ware we try to abstract this functionality into a Generic Enabler.

13.8.2   Implementation agnostic

The API abstracts from the concrete implementation technology. Implementations using various kinds of databases should be possible. Although the main goal is to store services descriptions in a distributed environment, any implementation of a repository can be used as long as the technical interfaces comply with the GE operation protocol and can be mapped (mediated) to the FI-Ware preferred REST-based reference implementation.

13.9   Detailed Specifications

13.9.1   Open API Specifications

- Repository Open RESTful API Specification (PRELIMINARY)

13.9.2   Other Open Specifications

The data formats for the API rely on the Linked USDL specifications:

- Linked USDL Core Vocabulary

- Linked USDL Pricing Vocabulary

- Linked USDL Service Level Agreements Vocabulary

- Linked USDL Security Vocabulary

13.10   References

REP1	RDF Primer W3C Recommendation 10 February 2004 (http://www.w3.org/TR/2004/R

	EC-rdf-primer-20040210/)
REP2	SPARQL 1.1 Overview, W3C Working Draft 01 May 2012 (http://www.w3.org/TR/2012/WD-sparql11-overview-20120501/9

13.11   Re-utilised Technologies/Specifications

The Repository GE is based on RESTful Design Principles. The technologies and specifications used in this GE are:

- RESTful web services

- HTTP/1.1

- JSON and XML data serialization formats

13.12   Terms and definitions

This section comprises a summary of terms and definitions introduced during the previous sections. It intends to establish a vocabulary that will be help to carry out discussions internally and with third parties (e.g., Use Case projects in the EU FP7 Future Internet PPP). For a summary of terms and definitions managed at overall FI-WARE level, please refer to FIWARE Global Terms and Definitions

- **Aggregator (Role):** Supports domain specialists and third-parties in aggregating services and apps for new and unforeseen opportunities and needs. It does so by providing the dedicated tooling for aggregating services at different levels: UI, service operation, business process or business object levels.

- **Application:** Applications in FI-Ware are composite services that have a IT supported interaction interface (user interface). In most cases consumers do not buy the application they rather buy the right to use the application (user license).

- **Broker (Role):** The business network's central point of service access, being used to expose services from providers that are to be delivered through the Broker's service delivery functionality. The broker is the central instance for enabling monetization.

- **Business Element:** Core element of a business model, such as pricing models, revenue sharing models, promotions, SLAs, etc.

- **Business Framework:** Set of concepts and assets responsible for supporting the implementation of innovative business models in a flexible way.

- **Business Model:** Strategy and approach that defines how a particular service/application is supposed to generate revenue and profit. Therefore, a Business Model can be implemented as a set of business elements which can be combined and customized in a flexible way and in accordance to business and market requirements and other characteristics.

- **Business Process:** Set of related and structured activities producing a specific service or product, thereby achieving one or more business objectives. An operational business process clearly defines the roles and tasks of all involved parties

inside an organization to achieve one specific goal.

- **Business Role:** Set of responsibilities and tasks that can be assigned to concrete business role owners, such as a human person or a software component.

- ""*Channel: Resources through which services are accessed by end users. Examples for well-known channels are Web sites/portals, web-based brokers (like iTunes, eBay and Amazon), social networks (like Facebook, LinkedIn and MySpace), mobile channels (Android, iOS) and work centres. The mode of access to these channels is governed by technical channels like the Web, mobile devices and voice response, where each of these channels requires its own specific workflow.*

- **Channel Maker (Role):** Supports parties in creating outlets (the Channels) through which services are consumed, i.e. Web sites, social networks or mobile platforms. The Channel Maker interacts with the Broker for discovery of services during the process of creating or updating channel specifications as well as for storing channel specifications and channeled service constraints back in the Broker.

- **Composite Service (composition):** Executable composition of business back-end MACs. Common composite services are either orchestrated or choreographed. Orchestrated compositions are defined by a centralized control flow managed by a unique process that orchestrates all the interactions (according to the control flow) between the external services that participate in the composition. Choreographed compositions do not have a centralized process, thus the services participating in the composition autonomously coordinate each other according to some specified coordination rules. Backend compositions are executed in dedicated process execution engines. Target users of tools for creating Composites Services are technical users with algorithmic and process management skills.

- **Consumer (Role):** Actor who searches for and consumes particular business functionality exposed on the Web as a service/application that satisfies her own needs.

- **Desktop Environment:** Multi-channel client platform enabling users to access and use their applications and services.

- **Event-driven Composition:** Components concerned with the composition of business logic which is driven by asynchronous events. This implies run-time selection of MACs and the creation/modification of orchestration workflows based on composition logic defined at design-time and adapted to context and the state of the communication at run-time.

- **Front-end/Back-end Composition:** Front-end compositions define a front-end application as an aggregation of visual mashable application pieces (named as widgets, gadgets, portlets, etc.) and back-end services. Front-end compositions interact with end-users, in the sense that front-end compositions consume data provided by the end-users and provide data to them. Thus the frontend composition (or mashup) will have direct influence on the application look and feel; every component will add a new user interaction feature.

- Back-end compositions define a back-end business service (also known as process) as an aggregation of backend services as defined for service composition term, the end-user being oblivious to the composition process. While back-end components represent atomization of business logic and information processing, front-end components represent atomization of information presentation and user interaction.

- **Gateway (Role):** The Gateway role enables linking between separate systems and services, allowing them to exchange information in a controlled way despite different

technologies and authoritative realms. A Gateway provides interoperability solutions for other applications, including data mapping as well as run-time data store-forward and message translation. Gateway services are advertised through the Broker, allowing providers and aggregators to search for candidate gateway services for interface adaptation to particular message standards. The Mediation is the central generic enabler. Other important functionalities are eventing, dispatching, security, connectors and integration adaptors, configuration, and change propagation.

- **Hoster (Role):** Allows the various infrastructure services in cloud environments to be leveraged as part of provisioning an application in a business network. A service can be deployed onto a specific cloud using the Hoster's interface. This enables service providers to re-host services and applications from their on-premise environments to cloud-based, on-demand environments to attract new users at much lower cost.

- **Marketplace:** Part of the business framework providing means for service providers to publish their service offerings and service consumers to compare and select a specific service implementation. A marketplace can offer services from different stores and thus different service providers. The actual buying of a specific service is handled by the related service store.

- **Mashup:** Executable composition of front-end MACs. There are several kinds of mashups, depending on the technique of composition (spatial rearrangement, wiring, piping, etc.) and the MACs used. They are called application mashups when applications are composed to build new applications and services/data mash-ups if services are composed to generate new services. While composite service is a common term in backend services implementing business processes, the term 'mashup' is widely adopted when referring to Web resources (data, services and applications). Front-end compositions heavily depend on the available device environment (including the chosen presentation channels). Target users of mashup platforms are typically users without technical or programming expertise.

- **Mashable Application Component (MAC):** Functional entity able to be consumed executed or combined. Usually this applies to components that will offer not only their main behaviour but also the necessary functionality to allow further compositions with other components. It is envisioned that MACs will offer access, through applications and/or services, to any available FI-WARE resource or functionality, including gadgets, services, data sources, content, and things. Alternatively, it can be denoted as 'service component' or 'application component'.

- **Mediator:** A mediator can facilitate proper communication and interaction amongst components whenever a composed service or application is utilized. There are three major mediation area: Data Mediation (adapting syntactic and/or semantic data formats), Protocol Mediation (adapting the communication protocol), and Process Mediation (adapting the process implementing the business logic of a composed service).

- **Monetization:** Process or activity to provide a product (in this context: a service) in exchange for money. The Provider publishes certain functionality and makes it available through the Broker. The service access by the Consumer is being accounted according to the underlying business model and the resulting revenue is shared across the involved service providers.

- **Premise (Role):** On-Premise operators provide in-house or on-site solutions, which are used within a company (such as ERP) or are offered to business partners under specific terms and conditions. These systems and services are to be regarded as external and legacy to the FI-Ware platform because they do not conform to the

architecture and API specifications of FI-Ware. They will only be accessible to FI-Ware services and applications through the Gateway.

- **Prosumer:** A user role able to produce, share and consume their own products and modify/adapt products made by others.

- **Provider (Role):** Actor who publishes and offers (provides) certain business functionality on the Web through a service/application endpoint. This role also takes care of maintaining this business functionality.

- **Registry and Repository:** Generic enablers that able to store models and configuration information along with all the necessary meta-information to enable searching, social search, recommendation and browsing, so end users as well as services are able to easily find what they need.

- **Revenue Settlement:** Process of transferring the actual charges for specific service consumption from the consumer to the service provider.

- **Revenue Sharing:** Process of splitting the charges of particular service consumption between the parties providing the specific service (composition) according to a specified revenue sharing model.

- **Service:** We use the term service in a very general sense. A service is a means of delivering value to customers by facilitating outcomes customers want to achieve without the ownership of specific costs and risks. Services could be supported by IT. In this case we say that the interaction with the service provider is through a technical interface (for instance a mobile app user interface or a Web service). Applications could be seen as such IT supported Services that often ar also composite services.

- **Service Composition:** in SOA domain, a service composition is an added value service created by aggregation of existing third party services according to some predefined work and data flow. Aggregated services provide specialized business functionality on which the service composition functionality has been split down.

- **Service Delivery Framework:** Service Delivery Framework (or Service Delivery Platform (SDP)) refers to a set of components that provide service delivery functionality (such as service creation, session control & protocols) for a type of service. In the context of FI-WARE, it is defined as a set of functional building blocks and tools to (1) manage the lifecycle of software services, (2) creating new services by creating service compositions and mashups, (3) providing means for publishing services through different channels on different platforms, (4) offering marketplaces and stores for monetizing available services and (5) sharing the service revenues between the involved service providers.

- **Service Level Agreement (SLA):** A service level agreement is a legally binding and formally defined service contract between a service provider and a service consumer, specifying the contracted qualitative aspects of a specific service (e.g. performance, security, privacy, availability or redundancy). In other words, SLAs not only specify that the provider will just deliver some service, but that this service will also be delivered on time, at a given price, and with money back if the pledge is broken.

- **Service Orchestration:** in SOA domain, a service orchestration is a particular architectural choice for service composition where a central orchestrated process manages the service composition work and data flow invocations the external third party services in the order determined by the work flow. Service orchestrations are specified by suitable orchestration languages and deployed in execution engines who interpret these specifications.

- **Store:** An external component integrated with the business framework offering a set of services that are published to a selected set of marketplaces. The store thereby holds the service portfolio of a specific service provider. In case a specific service is purchased on a service marketplace, the service store handles the actual buying of a specific service (as a financial business transaction).

- **Unified Service Description Language (USDL):** USDL is a platform-neutral language for describing services, covering a variety of service types, such as purely human services, transactional services, informational services, software components, digital media, platform services and infrastructure services. The core set of language modules offers the specification of functional and technical service properties, legal and financial aspects, service levels, interaction information and corresponding participants. USDL is offering extension points for the derivation of domain-specific service description languages by extending or changing the available language modules.

14   FIWARE OpenSpecification Apps RepositoryREST

You can find the content of this chapter as well in the wiki of fi-ware.

14.1   Introduction to the *Repository* API

The FI-WARE Generic Enabler Specification are owned by different partners. Therefore, different Legal Notices might apply. Please check for each FI-WARE Generic Enabler Specification the Legal Notice attached. For this FI-WARE Generic Enabler Specification, this Legal Notice applies.

SAP strives to make the specifications of this Generic Enabler available under IPR rules that allow for a exploitation and sustainable usage both in Open Source as well as proprietary, closed source products to maximize adoption.

14.1.1   Repository API Core

The Repository API is a RESTful, resource-oriented API accessed via HTTP that uses various representations for information interchange. The Repository provides a consistent uniform API to USDL service descriptions and associated media files.

14.1.2   Intended Audience

This specification is intended for both software developers and reimplementers of this API. For the former, this document provides a full specification of how to interoperate with products that implement the Repository API. For the latter, this specification indicates the interface to be implemented and provided to clients.

To use this information, the reader should firstly have a general understanding of the Generic Enabler service Repository. You should also be familiar with:

- RESTful web services
- HTTP/1.1
- JSON and/or XML data serialization formats.

14.1.3   API Change History

This version of the Repository API Guide replaces and obsoletes all previous versions. The most recent changes are described in the table below:

Revision Date	Changes Summary
Apr 2, 2012	• Initial version

14.1.4   How to Read This Document

It is assumed that the reader is familiar with the REST architecture style. Within the document, some special notations are applied to differentiate some special words or concepts. The following list summarizes these special notations.

- A bold, mono-spaced font is used to represent code or logical entities, e.g., HTTP method (`GET, PUT, POST, DELETE`).

- An italic font is used to represent document titles or some other kind of special text, e.g., *URI*.

- Variables are represented between brackets, e.g. {id} and in italic font. The reader can replace the id with an appropriate value.

For a description of some terms used along this document, see Repository.

14.1.5   Additional Resources

You can download the most current version of this document from the FI-WARE API specification website at Repository API . For more details about the Repository GE that this API is based upon, please refer to High Level Description. Related documents, including an Architectural Description, are available at the same site.

14.2   General *Repository* API Information

14.2.1   Resources Summary

The repository is structured into core objects, which are *resources*, and *collections*. These objects constitute also the granularity of access control. Collections are containers for storing resources. Any object can provide services such as search, query, transform, maintain, depending on the type of the resource.

14.2.1.1   *Resources*

The resources are mainly the USDL service descriptions themselves as well as complementary media files that are used within the service descriptions.

14.2.1.2   *Collections*

A collection is a container for collecting resources and other collections. Multiple collections can be used on the repository for various purposes. Collections provide versioning of the resources.

Graphical diagram in which the different URIs that can be used in the API is shown here.
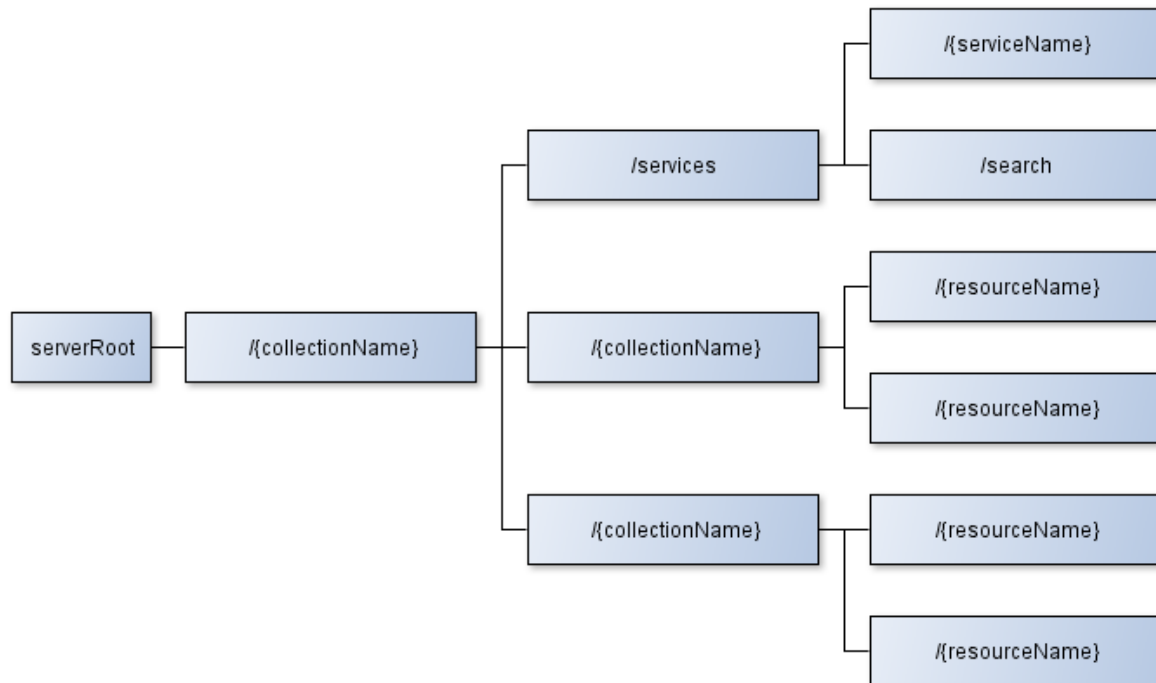
Schematic resoure tree of the Repository

14.2.2   Authentication

Each HTTP request against the *Repository GE* requires the inclusion of specific authentication credentials. The specific implementation of this API may support multiple authentication schemes (OAuth, Basic Auth, Token). It is left open how authentication is realized. In practice ith will be determined by the specific environment and the provider implementing the GE. Some authentication schemes may require that the API operate using SSL over HTTP (HTTPS).

14.2.3   Representation Format

The *Repository* API supports XML and JSON for delivering metadata resources and any kind of media type for media resources, it may also support RDF, Turtle (http://www.w3.org/TeamSubmission/turtle/) and Atom (http://tools.ietf.org/html/rfc4287) HTML for delivering metadata. The request format is specified using the Content-Type header and is required for operations that have a request body. The response format can be specified in requests using the Accept header. Note that it is possible for a response to be serialized using a format different from the request (see example below).

If no Content-Type is specified, the content is delivered in the format that was choosen to upload the resource.

The interfaces should support data exchange through multiple formats:

- *text/plain* - A linefeed separated list of elements for easy mashup and scripting.

- *text/html* - An human-readable HTML rendering of the results of the operation as output format.

- *application/json* - A JSON representation of the input and output for mashups or JavaScript-based Web Apps

- *application/rdf+xml* - A RDF description of the input and output.

In a concrete implementation of this GE other formats like RSS, Atom, etc. may also be possible.

14.2.4   Representation Transport

Resource representation is transmitted between client and server by using HTTP 1.1 protocol, as defined by IETF RFC-2616. Each time an HTTP request contains payload, a Content-Type header shall be used to specify the MIME type of wrapped representation. In addition, both client and server may use as many HTTP headers as they consider necessary.

14.2.5   Resource Identification

The resource identification for HTTP transport is made using the mechanisms described by HTTP protocol specification as defined by IETF RFC-2616.

14.2.6   Links and References

14.2.6.1   *Web citizen*

The repository is relying on Web principles:

- URI to identify resources

- consistent URI structure based on REST style protocol

- HTTP content negotiation to allow the client to choose the appropriate data format supporting HTML, RDF, XML, RSS, JSON, Turtle, ...

- Human readable output format using HTML rendering ('text/html' accept header) including hyperlinked representation

- Use of HTTP response codes including ETags (proper caching)

- Linked Data enablement supporting RDF input and output types

14.2.6.2   *Linked Open Data*

Publishing data as linked data requires every resource to be directly resolvable given their URL. The basic idea of Linked Data is simple. Tim Berners-Lee's note on Linked Data (http://www.w3.org/DesignIssues/LinkedData) describes four rules for publishing data on the Web:

- Use URIs as names for things

- Use HTTP URIs so that people can look up those names.

- When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL)

- Include links to other URIs, so that they can discover more things.

This can actually achieved by different approaches. One is the use of a special resolver similar to URL shorteners.

So the authoring environment has to ensure that every URI (actually IRI - Internationalized Resource Identifiers - RFC 3987) can be resolved by a HTTP `GET` request. For example: If a resource is maintained in a repository under the URL http://repository.acme.com/service/xyz but the IRI used in service descriptions is actually http://fi-ware.org/service/xyz, we need a resolver at this location which redirects the request to the actual repository.

Setting up resolvers is more complex task. Therefore we try to follow a simpler approach for the USDLRepositoryRest. The API is designed to be directly used for Linked Data publishing without the need for a resolver.

14.2.7   Paginated Collections

In order to reduce the load on the service, we can decide to limit the number of elements to return when it is too big. This section explain how to do that using for example a limit parameter (optional) and a last parameter (optional) to express which is the maximum number of element to return and which was the last element to see.

These operations will have to cope with the possibility to have over limit fault (413) or item not found fault (404).

14.2.8   Limits

We can manage the capacity of the system in order to prevent the abuse of the system through some limitations. These limitations will be configured by the operator and may differ from one implementation to other of the GE implementation.

14.2.8.1   *Rate Limits*

These limits are specified both in human readable wild-card and in regular expressions and will indicate for each HTTP verb which will be the maximum number of operations per time unit that a user can request. After each unit time the counter is initialized again. In the event a request exceeds the thresholds established for your account, a 413 HTTP response will be returned with a Retry-After header to notify the client when they can attempt to try again.

14.2.9   ETag Handling

For standard caching an ETag HTTP header is provided for `GET` and `PUT` requests. If a `GET` requests has a "If-None-Match" header, than the content is only delivered if the stored ETag of the object matches the requested ETag. HTTP status code 304 (not changed) is responded otherwise.

For `PUT` requests the ETag header can be used to ensure integrity of the repository. The `PUT` operation will only be executed if the "If-Match" header matches the stored ETag of the resource in the repository. If no "If-Match" header is given for an existing resource or the "If-Match" header does not match the existing ETag of the resource, status code 409 (Conflict)will be returned. If the resource was changed, then a new ETag header will be returned in the response header.

14.2.10   Extensions

The Repository could be extended in the future. At the moment, we foresee the following resource to indicate a method that will be used in order to allow the extensibility of the API.

This allow the introduction of new features in the API without requiring an update of the version, for instance, or to allow the introduction of vendor specific functionality.

Verb	URI	Description
GET	/extensions	List of all available extensions

14.2.11 Faults

14.2.11.1 *Synchronous Faults*

Error codes are returned in the body of the response. The description section returns a human-readable message for displaing end users.

Example:

```
<exception>
      <description>Resource Not found</description>
      <errorCode>404</errorCode>
      <reasonPhrase>Not Found</reasonPhrase>
</exception>
```

Fault Element	Associated Error Codes	Expected in All Requests?
Unauthorized	403	YES
Not Found	404	YES
Limit Fault	413	YES
Internal Server error	50X	YES

14.3   API Operations

14.3.1   Managing Collections

Verb	URI	Description
GET	/{CollectionPath}	Get a collection
PUT	/{CollectionPath}	Create or update a collection
DELETE	/{CollectionPath}	Delete a collection

14.3.1.1 *Status Codes*
200 OK

The request was handled successfully and transmitted in response message.

201 Created

The request has been fulfilled and resulted in a new resource being created.

204 No Content

The server successfully processed the request, but is not returning any content.

304 Not Modified

Indicates the resource has not been modified since last requested. Typically, the HTTP client provides a header like the If-Modified-Since header to provide a time against which to compare. Using this saves bandwidth and reprocessing on both the server and client, as only the header data must be sent and received in comparison to the entirety of the page being re-processed by the server, then sent again using more bandwidth of the server and client.

400 Bad Request

The request cannot be fulfilled due to bad syntax.

404 Not Found

The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible.

409 Conflict

Indicates that the request could not be processed because of conflict in the request, such as an edit conflict.

500 Internal Server Error

A generic error message, given when no more specific message is suitable.

14.3.2   Managing Resources

Verb	URI	Description
GET	/{CollectionPath}/{ResourceID}	Get a resource
PUT	/{CollectionPath}/{ResourceID}	Create or update a resource
DELETE	/{CollectionPath}/{ResourceID}	Delete a resource

14.3.2.1   *Status Codes*

200 OK

The request was handled successfully and transmitted in response message.

201 Created

The request has been fulfilled and resulted in a new resource being created.

204 No Content

The server successfully processed the request, but is not returning any content.

304 Not Modified

Indicates the resource has not been modified since last requested. Typically, the HTTP client provides a header like the If-Modified-Since header to provide a time against which to compare. Using this saves bandwidth and reprocessing on both the

server and client, as only the header data must be sent and received in comparison to the entirety of the page being re-processed by the server, then sent again using more bandwidth of the server and client.

400 Bad Request

The request cannot be fulfilled due to bad syntax.

404 Not Found

The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible.

409 Conflict

Indicates that the request could not be processed because of conflict in the request, such as an edit conflict.

500 Internal Server Error

A generic error message, given when no more specific message is suitable.

14.3.3   Additional Services

Verb	URI	Description
GET	/{collectionPath}/services	Get a list of additional services

14.3.3.1   *Status Codes*
200 OK

The request was handled successfully and transmitted in response message.

500 Internal Server Error

A generic error message, given when no more specific message is suitable.

14.3.4   Searching the Repository

Verb	URI	Description
GET	/{CollectionPath}/search?q={queryString}	Search a collection

14.3.4.1   *Status Codes*
200 OK

The request was handled successfully and transmitted in response message.

400 Bad Request

The request cannot be fulfilled due to bad syntax.

404 Not Found

The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible.

500 Internal Server Error

A generic error message, given when no more specific message is suitable.

15   FIWARE OpenSpecification Apps Mediator

You can find the content of this chapter as well in the wiki of fi-ware.

Name	FIWARE.OpenSpecification.Apps.Mediator
Chapter	Apps,
Catalogue-Link to Implementation	Mediator
Owner	Telecom Italia, THALES, Marco Ughetti, Pierre Chatel

15.1   Preface

Within this document you find a self-contained open specification of a FI-WARE generic enabler, please consult as well the FI-WARE_Product_Vision, the website on http://www.fi-ware.eu and similar pages in order to understand the complete context of the FI-WARE project.

15.2   Copyright

- Copyright © 2012 by TELECOM ITALIA, THALES

15.3   Legal Notice

Please check the following Legal Notice to understand the rights to use these specifications.

15.4   Overview

Providing interoperability solutions is the main functionality of the Mediator. The heterogeneity that exists among the ways to represent data (i.e. to represent syntactically and semantically the information items that are requested or provided by an application or a service), and to represent the communication pattern and the protocol or the public process needed to request a functionality (executing a composition in a different execution environment or implementing dynamic run-time changes might require a process mediation function), are problems that arise in FIWARE. Acknowledging the necessity to deal with these heterogeneities, mediation solutions are required in FIWARE.

The mediator is basically a middleware application responsible for providing interoperability among different communication protocols and among different data models. For example it can convert ASCII delimited message payloads from older protocols such as FTP into an XML message payload submitted to a web service (both soap over http or rest over http). Thus the main capabilities of the mediator are protocol and data transformations. An other example of data transformation is the transformation of an XML payload into an other XML payload through XSLT or XQuery.

The Mediator provides data mediation and protocol mediation capabilities to enable clients that play the role of Mediation Service Creators to compose different kinds of target service,

and enable clients that play the role of Mediation Service Clients to invoke the mediated services   (see   picture). The Composition Engine GEs plays both roles of Mediation Service Creator and Mediation Service Client.
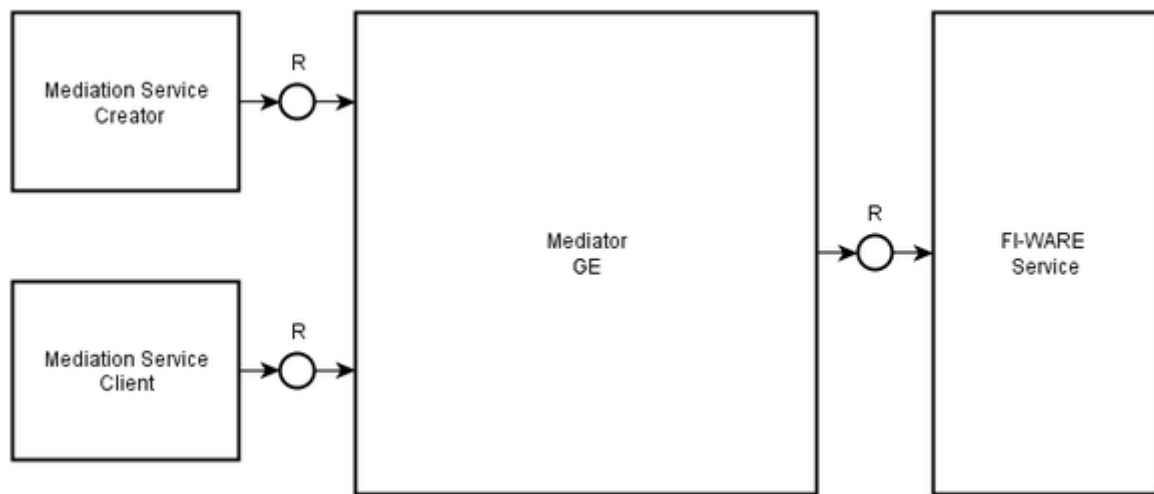
Figure 1.1 Mediator Role in FI-WARE

The mediator provides an Administrator GUI and APIs in order to allow mediation services to be constructed given a target service to be used in a service composition.

15.5   Basic Concepts

FI-WARE platform should be able to support services exposed through different protocols and technologies and enable the creation of new composed services. The mediator shall provide the "glue" between the service layer and the composition layer in order to enhance the composition capabilities of the composing GEs.

Within FI-WARE we abstract this functionality into a Generic Enabler called Mediator. The API abstracts the concrete implementation technology. Implementations using various kinds of platforms and frameworks should be possible. The main goal of the mediator is to provide a virtual proxy of the target service to be used by the Compositon Engine GE instead of the target service. The Virtual Proxy is configured with *Mediation Tasks* and *Dynamic Mediation Tasks* that provide data mediation and protocol mediation capabilities in order to make the target service suitable for composition. The first release of this GE will provide an Administration GUI for the configuration of such Virtual Proxies. The final release will provide remote generic APIs to allow the configuration of the Mediator directly by the other GEs

15.5.1   Data Model

The mediator offers a set of available *mediation tasks* and *dynamic mediation tasks*: the set of mediation capabilities that can be used via the mediator. The mediator allows users to create and manage their *mediation services*: a *mediation service* is a virtual proxy towards a web service that executes a chain of *mediation tasks* and/or *dynamic mediation tasks* between the caller and the target service. The *mediation tasks* and *dynamic mediation tasks* must be choosen from the set of available task types and the concrete implementation of the mediation tasks to be chained are potentially provided by different mediator implementations.

Each mediator implementation (asset) will provide its own set of addressable *mediation tasks* and/or *dynamic mediation tasks*. How to build a concrete *mediation task* or *dynamic mediation task* depends on the specific mediator implementation.

15.5.1.1 *Mediation Task*

Mediation tasks are the mediation capabilities that can be used via the mediator. The mediator maintains a set of the available mediation tasks.

The concrete implementation of a mediation task is provided by a specific mediator implementation (asset).

Examples of mediation tasks provided by the TI Mediation asset include:

- SOAP2REST: allows a REST Service to be called from the SOAP protocol
- SOAP2POX: allows a service that is expecting a POX Payload (Plain Old XML) to be called from the SOAP protocol
- TCP2HTTP: allows a service exposed via HTTP to be called using TCP transport

The mediation tasks exposed by Mediation GE that can be implemented using the TI Mediation asset, are a chain of the built-in low level mediation capabilities provided by WSO2 ESB and Apache Camel

A short list of these **mediation capabilites**:

Name	Description
Send	Send a message out
Log	Logs a message
Property	Set or remove properties associated with the message
Sequence	Refer a sequence
Event	Send event notifications to an event source
Drop	Drops a message
Enrich	Enriches a message
Enqueue	Create an enqueue mediator
Filter	Filter a message using Xpath (if else logic)
Out	Inbuilt filter for choosing messages in ESB out path
In	Inbuilt filter for choosing messages in ESB path
Switch	Filter a message using Xpath (switch logic)
Router	Route messages based on XPath filtering
Conditional Router	Route messages based on 'Condition'
Validate	Schema validation for messages
XSLT	XSLT Transformations

XQuery	XQuery Transformations
Header	Set or remove SOAP Headers
Fault	Create or remove SOAP Faults
Rewrite	Create a rewrite mediator

We provide some example of the configuration of these mediation tasks using the TI Mediation asset.

Example 1: WS-Security

Virtual proxy configuration that adds WS-Security to the unsecured target service "ServiceExample"

```
<proxy xmlns="http://ws.apache.org/ns/synapse"
name="SecuredServiceExampleProxy" transports="https"
startOnLoad="true">
 <target>
    <inSequence>
       <header xmlns:wsse="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
name="wsse:Security" action="remove" />
    </inSequence>
    <endpoint>
       <address uri="http://<ServiceExample URL>" />
    </endpoint>
    <outSequence>
       <send />
    </outSequence>
 </target>
 <enableSec />
 <policy key="conf:/repository/axis2/service-
groups/SecuredServiceExampleProxy/services/SecuredServiceExampleProx
y/policies/UTOverTransport" />
</proxy>
```

Example 2: Protocol Transformation TCP2HTTP

```
<proxy xmlns="http://ws.apache.org/ns/synapse"
name="TCPServiceExample" transports="tcp" startOnLoad="true">
  <target>
    <endpoint>
       <address uri="http://<ServiceExample URL>" />
    </endpoint>
    <outSequence>
       <send />
    </outSequence>
  </target>
</proxy>
```

Additional Mediation tasks will be added to the Mediator GE on the basis of FI-WARE needs.

15.5.1.2 *Dynamic Mediation Task*

In the current vision of the mediator GE, this enabler allows users to create and manage mediation services "offline", at design time. A mediation service is a chain of *mediation tasks* and *dynamic mediation tasks* between a service producer and consumer that can be accessed through a Web service interface. This chain deals with all the mediation problems that may arise between these two protagonists.

To cope with some limitations in this current pragmatic vision of how mediation is made – limitations related to potential missing information at design time – the mediator GE offers specific *mediation tasks* called *"Dynamic mediation tasks"*. These tasks may be needed because, at design time (when the chain of tasks is defined), all the needed data and/or information is not necessarily available to be able to solve mediation issues between the caller request (from "service consumer") and the target service ("service provider").

We postulate that this data and information will became available at runtime and that the *dynamic mediation tasks* will then dynamically solve the remaining mediation issues. This approach is the first step toward a fully dynamic mediation.

The concrete implementation of a *dynamic mediation task* is provided through features provided by an existing THALES asset called *SETHA2* that deals with data, protocol and process mediation in a SOA context.

We identify multiple *dynamic mediation tasks* types that are detailed hereafter:

- Data *dynamic mediation tasks*

  The data dynamic mediation tasks can be used to solve the following issues:

  - Consumer knows the target service and the operation to invoke but not its parameters (e.g. order of parameters, exact type of parameter to use, …)

  - Consumer knows the target service but not exactly the operation to invoke (e.g. the precise name of the target operation is not known)

- Protocol *dynamic mediation tasks*

  If the target service protocol is not known at design time, the protocol dynamic mediation task will be used to bridge the protocol gap between service consumer and producer (e.g. SOAP/HTTP service consumer and DDS service).

- Process *dynamic mediation tasks*

  To be used if there is a potential process mismatch at runtime between the consumer and the producer; in the case where one of the processes (either from consumer or producer) is not known at design time.

A second step toward fully dynamic mediation in FI-WARE would be to rely directly on the THALES asset to deal with extreme cases where only consumer's side requirements are known at design time and all mediation must be automatically defined and invoked at runtime.

15.5.1.3 *Mediation Service*

The mediation service represent the final mediation capability exposed by Mediator GE to the external word. A mediation service can be composed by a single mediation task (the simplest case) or by a chain of mediation tasks that can be provided by different Mediator implementations The mediation service is configured with a chain of *mediation tasks* and/or *dynamic mediation tasks*: they are the mediation tasks that the mediation service will execute between the caller and the service.

The *Mediation service URL* is the URL that allows the invocation of the target service with the mediation logic included in the chain of mediation tasks/dynamic mediation tasks configured.

To configure the mediation service the *Target service endpoint* must be specified: it is the URL of the target web service that will be invoked via the mediation service.

15.6 Mediator Architecture

The Mediator GE is used mainly by the Compositon Engine GEs within the FI-WARE platform. It provides a layer of virtual proxies to be used by the composer instead of the target services in order to allow the composer support various kinds of target services. Besides the FI-WARE platform, Future Internet applications or composed services on top of the FI-WARE platform can use the mediator as a service for their own purpose.
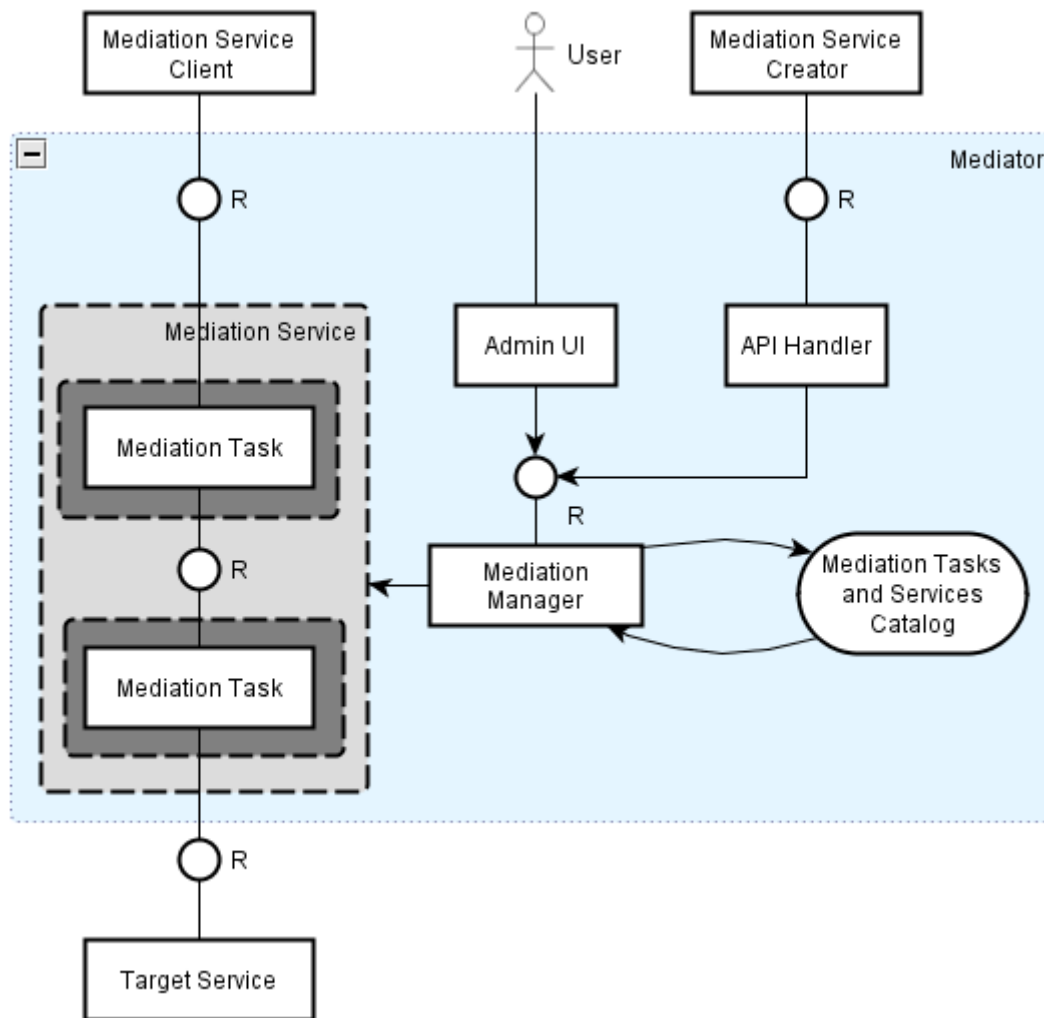
Figure 3.1: Mediator Architecture

15.7   Main Interactions

There are two main interactions provided by the mediator:

- at design time there are interactions in order to create and handle virtual proxies
- at execution time the mediator provides the virtual proxy, whoose URL has to be invoked in order to mediate the target service

The design time interaction occurs between a client that plays the role of the Mediation Service Creator and the Mediator.

The execution time interaction occurs beetween the Mediation Service Clients and the Mediation Services exposed by the Mediator. The mediated services are invoked by the Mediation Service Clients just like any other service.

As regards the current release, all design time interaction needed to manage mediation tasks and services is performed through the Web GUI of the various Mediator Implementations (assets). Refer to the User guide of the specific asset.

The main interaction at execution time will be:

- invocation of the mediation service, exposed by the mediator

The main interactions at design time will be (preliminary description of the remote APIs that will be provided by the final release of the mediator):

- create a mediation service
- delete a mediation service
- get a specific mediation service configuration
- get available mediation tasks
- get available dynamic mediation tasks

15.7.1   Invocation of the Mediation Service

At execution time the mediation capabilities are provided through services that can be invoked by the client GE using the mediation service URL. The mediation services can be exposed using various technologies, for example through soap web services and rest web services.

15.7.2   Mediation Service Management

The design time API will be designed for future releases of the FIWARE platform taking into account the available implementations (assets) of the Mediator GE

15.8   Basic Design Principles

- **API Technology independence**

  The API abstracts the concrete implementation technology. Implementations using various kinds of platforms and frameworks are possible.

- **Modularity**

  Mediation Tasks can be composed, creating Mediation Task chains that realize complex mediation logic.

15.9   Concrete Implementation Documentation

15.9.1   TI Mediation Asset

In order to learn how to create mediation tasks with the TI implementation of the FIWARE GE enabler refers to the documentation of the Architect week in Zurich TI-Mediator-doc and to understand in deep detail the concept of Virtual proxy and how they are implemented in the TI Mediation asset see apache-synapse project mediation catalog:

 (http://synapse.apache.org/userguide/mediators.htm).

In order to have an understanding of the Administration GUI of the TE-Mediator see the User guide of Wso2 ESB Wso2-ESB-UserGuide

15.9.2   THALES Mediation Asset

Thales brings the *setha2* asset into FI-WARE in order to offer a set of dynamic mediation tasks. A brief description of *setha2* is available at SETHA2_DESCRIPTION.

Examples of mediation tasks provided by the THALES Mediation asset ("SETHA2"):

- Data *dynamic mediation tasks*
- Protocol *dynamic mediation tasks*
- Process *dynamic mediation tasks*

15.10   Detailed Specifications

15.10.1   Open API Specifications

- FIWARE.OpenSpecification.Apps.MediatorREST

15.10.2   Other Relevant Specifications

None

15.11   Re-utilised Technologies/Specifications

- RESTful web services
- HTTP/1.1
- W3C WS-*
- XML data serialization format

15.12   Terms and definitions

This section comprises a summary of terms and definitions introduced during the previous sections. It intends to establish a vocabulary that will be help to carry out discussions internally and with third parties (e.g., Use Case projects in the EU FP7 Future Internet PPP). For a summary of terms and definitions managed at overall FI-WARE level, please refer to FIWARE Global Terms and Definitions.

- **Aggregator (Role):** Supports domain specialists and third-parties in aggregating services and apps for new and unforeseen opportunities and needs. It does so by providing the dedicated tooling for aggregating services at different levels: UI, service operation, business process or business object levels.

- **Application:** Applications in FI-Ware are composite services that have a IT supported interaction interface (user interface). In most cases consumers do not buy the application they rather buy the right to use the application (user license).

- **Broker (Role):** The business network's central point of service access, being used to expose services from providers that are to be delivered through the Broker's service

delivery functionality. The broker is the central instance for enabling monetization.

- **Business Element:** Core element of a business model, such as pricing models, revenue sharing models, promotions, SLAs, etc.

- **Business Framework:** Set of concepts and assets responsible for supporting the implementation of innovative business models in a flexible way.

- **Business Model:** Strategy and approach that defines how a particular service/application is supposed to generate revenue and profit. Therefore, a Business Model can be implemented as a set of business elements which can be combined and customized in a flexible way and in accordance to business and market requirements and other characteristics.

- **Business Process:** Set of related and structured activities producing a specific service or product, thereby achieving one or more business objectives. An operational business process clearly defines the roles and tasks of all involved parties inside an organization to achieve one specific goal.

- **Business Role:** Set of responsibilities and tasks that can be assigned to concrete business role owners, such as a human person or a software component.

- ""*Channel: Resources through which services are accessed by end users. Examples for well-known channels are Web sites/portals, web-based brokers (like iTunes, eBay and Amazon), social networks (like Facebook, LinkedIn and MySpace), mobile channels (Android, iOS) and work centres. The mode of access to these channels is governed by technical channels like the Web, mobile devices and voice response, where each of these channels requires its own specific workflow.*

- **Channel Maker (Role):** Supports parties in creating outlets (the Channels) through which services are consumed, i.e. Web sites, social networks or mobile platforms. The Channel Maker interacts with the Broker for discovery of services during the process of creating or updating channel specifications as well as for storing channel specifications and channeled service constraints back in the Broker.

- **Composite Service (composition):** Executable composition of business back-end MACs. Common composite services are either orchestrated or choreographed. Orchestrated compositions are defined by a centralized control flow managed by a unique process that orchestrates all the interactions (according to the control flow) between the external services that participate in the composition. Choreographed compositions do not have a centralized process, thus the services participating in the composition autonomously coordinate each other according to some specified coordination rules. Backend compositions are executed in dedicated process execution engines. Target users of tools for creating Composites Services are technical users with algorithmic and process management skills.

- **Consumer (Role):** Actor who searches for and consumes particular business functionality exposed on the Web as a service/application that satisfies her own needs.

- **Desktop Environment:** Multi-channel client platform enabling users to access and use their applications and services.

- **Event-driven Composition:** Components concerned with the composition of business logic which is driven by asynchronous events. This implies run-time selection of MACs and the creation/modification of orchestration workflows based on composition logic defined at design-time and adapted to context and the state of the

communication at run-time.

- **Front-end/Back-end Composition:** Front-end compositions define a front-end application as an aggregation of visual mashable application pieces (named as widgets, gadgets, portlets, etc.) and back-end services. Front-end compositions interact with end-users, in the sense that front-end compositions consume data provided by the end-users and provide data to them. Thus the frontend composition (or mashup) will have direct influence on the application look and feel; every component will add a new user interaction feature.

- Back-end compositions define a back-end business service (also known as process) as an aggregation of backend services as defined for service composition term, the end-user being oblivious to the composition process. While back-end components represent atomization of business logic and information processing, front-end components represent atomization of information presentation and user interaction.

- **Gateway (Role):** The Gateway role enables linking between separate systems and services, allowing them to exchange information in a controlled way despite different technologies and authoritative realms. A Gateway provides interoperability solutions for other applications, including data mapping as well as run-time data store-forward and message translation. Gateway services are advertised through the Broker, allowing providers and aggregators to search for candidate gateway services for interface adaptation to particular message standards. The Mediation is the central generic enabler. Other important functionalities are eventing, dispatching, security, connectors and integration adaptors, configuration, and change propagation.

- **Hoster (Role):** Allows the various infrastructure services in cloud environments to be leveraged as part of provisioning an application in a business network. A service can be deployed onto a specific cloud using the Hoster's interface. This enables service providers to re-host services and applications from their on-premise environments to cloud-based, on-demand environments to attract new users at much lower cost.

- **Marketplace:** Part of the business framework providing means for service providers to publish their service offerings and service consumers to compare and select a specific service implementation. A marketplace can offer services from different stores and thus different service providers. The actual buying of a specific service is handled by the related service store.

- **Mashup:** Executable composition of front-end MACs. There are several kinds of mashups, depending on the technique of composition (spatial rearrangement, wiring, piping, etc.) and the MACs used. They are called application mashups when applications are composed to build new applications and services/data mash-ups if services are composed to generate new services. While composite service is a common term in backend services implementing business processes, the term 'mashup' is widely adopted when referring to Web resources (data, services and applications). Front-end compositions heavily depend on the available device environment (including the chosen presentation channels). Target users of mashup platforms are typically users without technical or programming expertise.

- **Mashable Application Component (MAC):** Functional entity able to be consumed executed or combined. Usually this applies to components that will offer not only their main behaviour but also the necessary functionality to allow further compositions with other components. It is envisioned that MACs will offer access, through applications and/or services, to any available FI-WARE resource or functionality, including gadgets, services, data sources, content, and things. Alternatively, it can be denoted as 'service component' or 'application component'.

- **Mediator:** A mediator can facilitate proper communication and interaction amongst components whenever a composed service or application is utilized. There are three major mediation area: Data Mediation (adapting syntactic and/or semantic data formats), Protocol Mediation (adapting the communication protocol), and Process Mediation (adapting the process implementing the business logic of a composed service).

- **Monetization:** Process or activity to provide a product (in this context: a service) in exchange for money. The Provider publishes certain functionality and makes it available through the Broker. The service access by the Consumer is being accounted according to the underlying business model and the resulting revenue is shared across the involved service providers.

- **Premise (Role):** On-Premise operators provide in-house or on-site solutions, which are used within a company (such as ERP) or are offered to business partners under specific terms and conditions. These systems and services are to be regarded as external and legacy to the FI-Ware platform because they do not conform to the architecture and API specifications of FI-Ware. They will only be accessible to FI-Ware services and applications through the Gateway.

- **Prosumer:** A user role able to produce, share and consume their own products and modify/adapt products made by others.

- **Provider (Role):** Actor who publishes and offers (provides) certain business functionality on the Web through a service/application endpoint. This role also takes care of maintaining this business functionality.

- **Registry and Repository:** Generic enablers that able to store models and configuration information along with all the necessary meta-information to enable searching, social search, recommendation and browsing, so end users as well as services are able to easily find what they need.

- **Revenue Settlement:** Process of transferring the actual charges for specific service consumption from the consumer to the service provider.

- **Revenue Sharing:** Process of splitting the charges of particular service consumption between the parties providing the specific service (composition) according to a specified revenue sharing model.

- **Service:** We use the term service in a very general sense. A service is a means of delivering value to customers by facilitating outcomes customers want to achieve without the ownership of specific costs and risks. Services could be supported by IT. In this case we say that the interaction with the service provider is through a technical interface (for instance a mobile app user interface or a Web service). Applications could be seen as such IT supported Services that often ar also composite services.

- **Service Composition:** in SOA domain, a service composition is an added value service created by aggregation of existing third party services according to some predefined work and data flow. Aggregated services provide specialized business functionality on which the service composition functionality has been split down.

- **Service Delivery Framework:** Service Delivery Framework (or Service Delivery Platform (SDP)) refers to a set of components that provide service delivery functionality (such as service creation, session control & protocols) for a type of service. In the context of FI-WARE, it is defined as a set of functional building blocks and tools to (1) manage the lifecycle of software services, (2) creating new services by creating service compositions and mashups, (3) providing means for publishing

services through different channels on different platforms, (4) offering marketplaces and stores for monetizing available services and (5) sharing the service revenues between the involved service providers.

- **Service Level Agreement (SLA):** A service level agreement is a legally binding and formally defined service contract between a service provider and a service consumer, specifying the contracted qualitative aspects of a specific service (e.g. performance, security, privacy, availability or redundancy). In other words, SLAs not only specify that the provider will just deliver some service, but that this service will also be delivered on time, at a given price, and with money back if the pledge is broken.

- **Service Orchestration:** in SOA domain, a service orchestration is a particular architectural choice for service composition where a central orchestrated process manages the service composition work and data flow invocations the external third party services in the order determined by the work flow. Service orchestrations are specified by suitable orchestration languages and deployed in execution engines who interpret these specifications.

- **Store:** An external component integrated with the business framework offering a set of services that are published to a selected set of marketplaces. The store thereby holds the service portfolio of a specific service provider. In case a specific service is purchased on a service marketplace, the service store handles the actual buying of a specific service (as a financial business transaction).

- **Unified Service Description Language (USDL):** USDL is a platform-neutral language for describing services, covering a variety of service types, such as purely human services, transactional services, informational services, software components, digital media, platform services and infrastructure services. The core set of language modules offers the specification of functional and technical service properties, legal and financial aspects, service levels, interaction information and corresponding participants. USDL is offering extension points for the derivation of domain-specific service description languages by extending or changing the available language modules.

- **Mediation Task:** Mediation tasks are the mediation capabilities that can be used via the mediator. The mediator maintains a set of the available mediation tasks.

- **Dynamic Mediation Task**: Mediation Tasks that can handle scenarios where all the needed data and/or information is not necessarily available at design time, but will became available at runtime.

- **Mediation Service**: The mediation service represent the final mediation capability exposed by Mediator GE to the external word. A mediation service can be composed by a single mediation task (the simplest case) or by a chain of mediation tasks that can be provided by different Mediator implementations.

16 FIWARE OpenSpecification Apps MediatorREST

You can find the content of this chapter as well in the wiki of fi-ware.

16.1 Introduction to the *Mediator* API

Please check the FI-WARE Open Specifications Legal Notice to understand the rights to use FI-WARE Open Specifications.

16.1.1 Mediator API Core

The Mediator API is a RESTful, resource-oriented API accessed via HTTP that uses XML representation for information interchange.

Mediator APIs allow to create and handle mediationServices i.e. services that can be invoked to call a target service adding specific mediation logics to the existing target service basic interface and behaviour. For more details about the Mediator GE, please refer to:

 Mediator GE Open Specification.

The mediator APIs are meant to expose a common interface for different mediator implementations. These APIS provide resources and operations that allow to create mediation services that are able to use the specific mediation tasks provided by the mediator implementations. These APIs don't fully specify the mediator tasks themselves to avoid any constraint into possible mediator implementation capabilities. Mediator APIs are not provided in the current version of the Mediator GE. A reference implementation of these APIs , will be provided in the final release of the Mediator GE.</br/>

16.1.2 Intended Audience

This specification is intended for both software developers and reimplementers of the Mediator GE. For the former, this document provides a full specification of how to interoperate with products that implement the Mediator APIs. For the latter, this specification indicates the interface to be implemented and provided to clients.

To use this information, the reader should firstly have a general understanding of the Mediator Generic Enabler. You should also be familiar with:

- RESTful web services
- HTTP/1.1
- W3C WS-*
- XML data serialization format

16.1.3 API Change History

This version of the Mediator API Guide replaces and obsoletes all previous versions. The most recent changes are described in the table below:

Revision Date	Changes Summary	Users
Apr 30, 2012	• Initial version	TI
Jul 05 05, 2012	• Dynamic mediation tasks	THALES

Sept 05, 2012	• Update Template	TI
Oct 29, 2012	• Updated Data Model Specification	TI THALES

16.1.4   How to Read This Document

It is assumed that the reader is familiar with the REST architecture style. Within the document, some special notations are applied to differentiate some special words or concepts. The following list summarizes these special notations.

- A bold, mono-spaced font is used to represent code or logical entities, e.g.

HTTP method (GET, PUT, POST, DELETE)

- An italic font is used to represent document titles or some other kind of special text, e.g.

URI

- Variables are represented between brackets and in italic font, e.g.

{id}

The reader can replace the *{id}* with an appropriate value.

For a description of some terms used within this document, refer to the Mediator GE Open Specification.

16.1.5   Additional Resources

For more details about the Mediator GE that this API is based upon, please refer to Mediator GE Open Specification.

16.2   General *Mediator* API Information

16.2.1   Resources Summary

The mediator is structured into mediationServices and mediationTasks (mediationTasks type can also be *dynamic mediation tasks*).
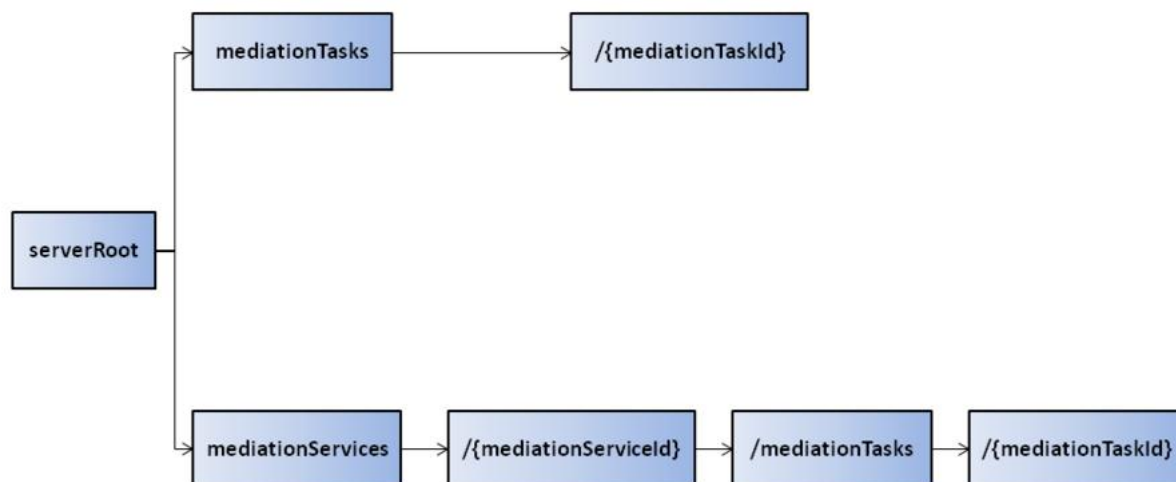
16.2.2   Authentication

Each HTTP request against the *Mediator GE* requires the inclusion of specific authentication credentials. The specific implementation of this API may support multiple authentication schemes (OAuth, Basic Auth, Token) and will be determined by the specific provider that implements the GE. Please contact them to determine the best way to authenticate against this API. Remember that some authentication schemes may require that the API operate using SSL over HTTP (HTTPS).

16.2.3   Representation Format

The *Mediator* API supports XML for delivering metadata resources. The request format is specified using the Content-Type header and is required for operations that have a request body. The response format can be specified in requests using the Accept header (application/xml).

The interfaces should support data exchange through XML format:

- *application/xml* - A XML description of the input and output.

16.2.4   Representation Transport

Resource representation is transmitted between client and server by using the HTTP 1.1 protocol, as defined by IETF RFC-2616. Each time an HTTP request contains payload, a Content-Type header shall be used to specify the MIME type of wrapped representation. In addition, both client and server may use as many HTTP headers as they consider necessary.

16.2.5   Resource Identification

In order to identify unambiguously the resources for HTTP transport is used the mechanisms described by HTTP protocol specification as defined by IETF RFC-2616.

16.2.6   Links and References

16.2.6.1   *Web citizen*

The mediator is relying on Web principles:

- URI to identify resources

- consistent URI structure based on REST style protocol

16.2.7   Paginated Collections

Mediator API will not limit the number of elements to return, because mediation service creation and handling doesn't require the exchange of very big data.

16.3   API Operations

16.3.1   Managing MediationServices

16.3.1.1   *MediationServices Operations*

Operations on the MediationServices resource:

Verb	URI	Name	Description
GET	/MediationServices	getMediationServices	Get the list of available mediationServices
GET	/MediationServices/{mediationServiceID}	getMediationService	Get the mediationService by ID
PUT	/MediationServices/{mediationServiceID}	createOrUpdateMediationService	Create or update a mediationService
DELETE	/MediationServices/{mediationServiceID}	deleteMediationService	Delete the mediationService

16.3.1.2   *Error Codes*

200 OK

The request was handled successfully and transmitted in response message.

201 Created

The request has been fulfilled and resulted in a new resource being created.

204 No Content

The server successfully processed the request, but is not returning any content.

304 Not Modified

Indicates the resource has not been modified since last requested. Typically, the HTTP client provides a header like the If-Modified-Since header to provide a time against which to compare. Using this saves bandwidth and reprocessing on both the

server and client, as only the header data must be sent and received in comparison to the entirety of the page being re-processed by the server, then sent again using more bandwidth of the server and client.

400 Bad Request

The request cannot be fulfilled due to bad syntax.

404 Not Found

The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible.

409 Conflict

Indicates that the request could not be processed because of conflict in the request, such as an edit conflict.

500 Internal Server Error

A generic error message, given when no more specific message is suitable.

16.3.2   Managing MediationTasks and DynamicMediationTasks

16.3.2.1   *MediationTasks and DynamicMediationTasks Operations*

Verb	URI	Name	Description
GET	/MediationTasks/	getMediationTasks	Get the list of available mediationTasks
GET	/DynamicMediationTasks/	getDynamicMediation Tasks	Get the list of available DynamicMediation
GET	/MediationTasks/{mediationTaskID}	getMediationTask	Get the mediationTask by ID
GET	/DynamicMediationTasks/{dynamic MediationTaskID}	getDynamicMediation Task	Get the DynamicMediationTask
GET	/MediationServices/{mediationServic eID}/MediationTasks	getMediationTasksFor MediationService	Get the list of mediationTasks and DynamicMediationTasks set on the media

16.3.2.2   *Error Codes*
200 OK

The request was handled successfully and transmitted in response message.

204 No Content

The server successfully processed the request, but is not returning any content.

400 Bad Request

The request cannot be fulfilled due to bad syntax.

404 Not Found

The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible.

500 Internal Server Error

A generic error message, given when no more specific message is suitable.

16.3.3   Resources and Operations Details

16.3.3.1   *Resources*

The mediator GE offers a set of available mediation tasks: the set of mediation capabilities provided by the specific implementations.

The mediator GE allows users to create and manage their mediation services: a mediation service is a virtual proxy towards a web service that executes a chain of mediation tasks between the caller and the target service. The mediation task must be choosen from the set of available mediation tasks.

Mediation Task

Every available mediation task is identified by a Mediation task id. This identifier is used to specify this mediation task in every usage; it will be used in the creation of a mediation service to configure this mediation task in the mediation task chain that will be executed by the mediation service.

Every mediation task may have task-specific properties.

Mediation task capabilities will be added to the Mediator GE on the basis of FI-WARE needs.

Mediation Service

A mediation service is identified by a Mediation service name.

Every mediation service is configured with a chain of Mediation tasks: it is the mediation task chain that the mediation service will execute between the caller and the service.

The Mediation service URL is the URL that allows the invocation of the target service with the mediation of the mediation task configured to be executed by the mediation service.

To configure the mediation service the Target service endpoint must be specified: it is the URL of the target web service that will be invoked via the mediation service.

Mediation service additional settings can be configured too (e.g. Timeout).

16.3.3.2   *Operations*
createOrUpdateMediationService

This interaction allows to create a new mediation service (virtual proxy) to a web service or to update an existing mediation service, configuring the chain of mediation tasks that will be executed when the mediation service is invoked. The chain of mediation tasks will be executed between the caller and the target service.

Returns the URL that allows the invocation of the target service with the specified mediation, i.e. the mediation service URL.

The APIs allow to obtain the list of the available mediation tasks, in order to choose the mediation tasks that will be executed by the mediation service (see the ***getMediationTasks*** operation). An example of mediation task is the SOAP_TO_REST task that converts SOAP invocations into REST invocation.

Input Parameters

- *Mediation service name* - Identification name for the mediation service. If no mediation service with the specified name exists, creates a new mediation service, identified by this name. If the specified name identify an existing mediation service then the invocation updates that mediation service according to the input data.

- *Target service endpoint* - Target service URL i.e. URL of the service that will be invoked via the mediation service.

- *Mediation service additional parameters* - mediation service additional settings (e.g. Timeout).

- *Mediation task list* - The mediation task list that the mediation service will execute between the caller and the service. For every mediation task the following parameters must be specified:

  -

    o *Mediation task id* - The task identifier (e.g. SOAP_TO_REST is the id of the mediation task that converts SOAP invocations into REST invocation). The list of available tasks, and the id of each of them, can be retrieved via the **getMediationTasks** operation.

    o *Task-specific parameters* - The parameter list of the specific task. For every parameter must be specified the identification name of the task parameter and the value of the parameter in the mediation service configuration.

Output Parameters

- *Mediation service URL* - The URL that will be returned: it is the URL that allows service invocation with the specified mediation, i.e. the mediation service URL.

deleteMediationService

This interaction allows to delete a mediation service.

Input Parameters

- *Mediation service name* - Identification name of the mediation service to delete.

getMediationService

This interaction allows to obtain the configuration of an existing mediation service.

Input Parameters

- *Mediation service name* - Identification name of the mediation service whose configuration we are asking for.

Output Parameters

- *Target service endpoint* - Target service URL i.e. URL of the service that will be invoked via the mediation service.

- *Mediation service additional parameters* - mediation service additional settings (e.g. Timeout).

- *Mediation task list* - The mediation task list that the mediation service will execute between the caller and the service. For every mediation task the following parameters are specified:

  -
    - o *Mediation task id* - The task identifier (e.g. SOAP_TO_REST is the id of the mediation task that converts SOAP invocations into REST invocation). The list of available tasks, and the id of each of them, can be retrieved via the **getMediationTasks** operation.
    - o *Mediation task description* - The description of the Mediation task.
    - o *Task-specific parameters* - The parameter list of the specific task. For every parameter is specified the identification name of the task parameter and the value of the parameter in the mediation service configuration.

- *Mediation service URL* - the URL that allows service invocation with the specified mediation, i.e. the mediation service URL.

getMediationServices

This interaction allows to obtain the configured mediation services, as a list of their identification names.

Input Parameters
None

Output Parameters

- *Mediation service list* - The list of mediation services configured on the Mediator. For every configured mediation service is returned its *Mediation service name*: the identification name of the mediation service, and the URL of the resource: the URL of the **getMediationService** operation for the specific mediation service

getMediationTasks

Returns the list of available mediation tasks that can be used to configure a mediation service. The mediation task is a basic mediation capability. When the mediation service is invoked, a list of mediation task will be executed (by the mediation service) between the caller and the target service.

Input Parameters
None

Output Parameters

- *Mediation task list* - The list of available mediation task that is returned. For every available mediation task is returned its *Mediation task id*, a *Mediation task description* and the URL of the resource: the URL of the **getMediationTask** operation for the specific mediation task. The *Mediation task id* is the identifier that will be used to specify this mediation task in every usage (e.g. it will be used in the

createOrUpdateMediationService operation to configure this mediation task as the one that will be executed by the mediation service)

getMediationTasksForMediationService

Returns the list of mediationTasks and DynamicMediationTasks set on the specified mediationService

Input Parameters
The *Mediation service name*, the identification name of the mediation service whose tasks we want to obtain.

Output Parameters

- *Mediation task list* - The list of mediationTasks and DynamicMediationTasks set on the specified mediationService. For every mediation task is returned its *Mediation task id*, a *Mediation task description* and the URL of the resource: the URL of the **getMediationTask** operation for the specific mediation task..

getMediationTask

Get the configuration of the mediationTask identified by the specified *Mediation task id*

Input Parameters

- *Mediation task id* - The id of the mediation task whose configuration we want to obtain.

The *Mediation task id* is the identifier that will be used to specify this mediation task in every usage (e.g. it will be used in the **createOrUpdateMediationService** operation to configure this mediation task as the one that will be executed by the mediation service)

Output Parameters

- *Mediation task id* - The task identifier.

- *Mediation task description* - The description of the Mediation task.

- *Task-specific parameters* - The parameter list of the specific task. For every parameter is specified the identification name of the parameter and the default value of the task parameter, if the parameter has a default value

16.3.4   Dynamic mediation tasks details

The **getDynamicMediationTasks** and getDynamicMediationTask provide the same information as geMediationTasks and **getMediationTasks**. Here we define more specifically the RESTfull API for these dynamic operations.

User who wants to use these operations must send http request towards their url resources :

Example of HTTP request
GET /DynamicMediationTasks HTTP/1.1
Host: example.org

OR

fi-ware

GET /DynamicMediationTasks/{dynamicMediationTaskID} HTTP/1.1
Host: example.org

16.3.4.1 *getDynamicMediationTasks*

The mediatior HTTP response for this operation follows the XSD file reproduced here:

```
<?xml version="1.0" encoding="UTF-8"?><br \>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.fiware.thalesgroup.com/DynamicMediationT
asks"
xmlns:tns="http://www.fiware.thalesgroup.com/DynamicMediationTasks"
elementFormDefault="unqualified"><br \>
    <element name="DynamicMediationTasks">
        <complexType>
            <sequence>
                <element name="linkrel" type="string" minOccurs="1"
maxOccurs="1">
                    <annotation>
                        <documentation>Relative location of the
operation url that retrieve dynamic mediation tasks.</documentation>
                    </annotation>
                </element>
                <element name="linkhref" type="string" minOccurs="1"
maxOccurs="1">
                    <annotation>
                        <documentation>Location in full http form of
the operation url that retrieve dynamic mediation
tasks.</documentation>
                    </annotation>
                </element>
                <element name="listDynamicMediationTasks"
minOccurs="1" maxOccurs="1">
                    <annotation>
                        <documentation>Listing of all dynamic
mediation tasks.</documentation>
                    </annotation>
                    <complexType>
                        <sequence>
                            <element name="DynamicMediationTask"
minOccurs="0" maxOccurs="unbounded">
                                <complexType>
                                    <sequence>
                                        <element name="id"
type="string" minOccurs="1" maxOccurs="1">
                                            <annotation>
                                                <documentation>Id of
the dynamic mediation tasks :
{dynamicMediationTaskID}</documentation>
                                            </annotation>
                                        </element>
```

```
                                           <element name="linkrel"
type="string" minOccurs="1" maxOccurs="1">
                                                  <annotation>

<documentation>relative location of this dynamic mediation task
resource</documentation>
                                                  </annotation>
                                           </element>
                                           <element name="linkhref"
type="string" minOccurs="1" maxOccurs="1">
                                                  <annotation>

<documentation>Location in full http form of this dynamic mediation
task</documentation>
                                                  </annotation>
                                           </element>
                                           <element name="description"
type="string" minOccurs="1" maxOccurs="1">
                                                  <annotation>

<documentation>Description of the dynamic mediation
task.</documentation>
                                                  </annotation>
                                           </element>
                                    </sequence>
                                 </complexType>
                            </element>
                       </sequence>
                    </complexType>
               </element>
          </sequence>
      </complexType>
   </element>
</schema>
```

Example of HTTP response
HTTP/1.1 200 OK
Date: ...
Content-Type: application/xml; charset=utf-8
Content-Length: nnn
Last-Modified: Sat, 12 Aug 2006 13:40:03 GMT

```
<?xml version="1.0" encoding="UTF-8"?><br \>
<tns:DynamicMediationTasks

xmlns:tns="http://www.fiware.thalesgroup.com/DynamicMediationTasks"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.fiware.thalesgroup.com/DynamicMediati
onTasks DynamicMediationTasks.xsd "><br \>
    <linkrel>/DynamicMediationTasks</linkrel>
    <linkhref>http://example.org/DynamicMediationTasks</linkhref>
    <listDynamicMediationTasks>
```

```
            <DynamicMediationTask>
                <id>{dynamicMediationTaskID}</id>

<linkrel>/DynamicMediationTasks/{dynamicMediationTaskID}</linkrel>

<linkhref>"http://example.org/DynamicMediationTasks/{dynamicMediatio
nTaskID}</linkhref>
                <description>description</description>
            </DynamicMediationTask>
        </listDynamicMediationTasks>
</tns:DynamicMediationTasks>
```

16.3.4.2 *getDynamicMediationTask*

The mediatior HTTP response for this operation follows the XSD file reproduced here:

```
<?xml version="1.0" encoding="UTF-8"?><br \>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.fiware.thalesgroup.com/DynamicMediationT
asks"
xmlns:tns="http://www.fiware.thalesgroup.com/DynamicMediationTasks"
elementFormDefault="unqualified"><br \>
    <element name="DynamicMediationTask">
        <complexType>
            <sequence>
                <element name="id" type="string" minOccurs="1"
maxOccurs="1">
                    <annotation>
                        <documentation>Id of the dynamic mediation
tasks : {dynamicMediationTaskID}</documentation>
                    </annotation>
                </element>
                <element name="linkrel" type="string" minOccurs="1"
maxOccurs="1">
                    <annotation>
                        <documentation>relative location of this
dynamic mediation task resource</documentation>
                    </annotation>
                </element>
                <element name="linkhref" type="string" minOccurs="1"
maxOccurs="1">
                    <annotation>
                        <documentation>Location in full http form of
this dynamic mediation task</documentation>
                    </annotation>
                </element>
                <element name="description" type="string"
minOccurs="1" maxOccurs="1">
                    <annotation>
```

```
                                   <documentation>Description of the dynamic
mediation task.</documentation>
                        </annotation>
                    </element>
                    <element name="specficParameters" minOccurs="1"
maxOccurs="1">
                        <annotation>
                        <documentation>The parameter list of the
specific task. For every parameter is specified the identification
name<br \> of the parameter and the default value of the task
parameter, if the parameter has a default value.</documentation>
                        </annotation>
                        <complexType>
                            <sequence>
                                <element name="parameter" minOccurs="0"
maxOccurs="unbounded">
                                    <complexType>
                                        <sequence>
                                            <element name="name"
type="string">
                                                <annotation>
                                                    <documentation>Name
of the parameter.</documentation>
                                                </annotation>
                                            </element>
                                            <element name="description"
type="string">
                                                <annotation>

<documentation>Desciprtion of the parameter.</documentation>
                                                </annotation>
                                            </element>
                                            <element name="defaultValue"
type="string" minOccurs="0" maxOccurs="1">
                                                <annotation>
                                                    <documentation>The
parameter can have a default value.</documentation>
                                                </annotation>
                                            </element>
                                            <element name="cardinality"
minOccurs="0" maxOccurs="1">
                                                <annotation>

<documentation>Cardinality specifies the number of repetitions of
this parameter (defines array).<br \>OPTIONAL signifies the
parameter can be omittedl. UNBOUNDED signifies that this parameter
can be repeated infinitely.</documentation>
                                                </annotation>
                                                <simpleType>
                                                    <restriction
base="string">
                                                        <enumeration
value="UNBOUNDED">
```

```
                                                    </enumeration>
                                                    <enumeration
value="OPTIONAL">
                                                    </enumeration>
                                                </restriction>
                                            </simpleType>
                                        </element>
                                    </sequence>
                                </complexType>
                            </element>
                        </sequence>
                    </complexType>
                </element>
            </sequence>
        </complexType>
    </element>
</schema>
```

Example of HTTP response
HTTP/1.1 200 OK
Date: ...
Content-Type: application/xml; charset=utf-8
Content-Length: nnn
Last-Modified: Sat, 12 Aug 2006 13:40:03 GMT

```
<?xml version="1.0" encoding="UTF-8"?><br \>
<tns:DynamicMediationTask

xmlns:tns="http://www.fiware.thalesgroup.com/DynamicMediationTasks"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.fiware.thalesgroup.com/DynamicMediati
onTasks DynamicMediationTasks.xsd "><br \>
    <id>{dynamicMediationTaskID}</id>

<linkrel>/DynamicMediationTasks/{dynamicMediationTaskID}</linkrel>

<linkhref>http://example.org/DynamicMediationTasks/{dynamicMediation
TaskID}</linkhref>
    <description>description</description>
    <specficParameters>
        <parameter>
            <name>name</name>
            <description>description</description>
            <defaultValue>defaultValue</defaultValue>
            <cardinality>UNBOUNDED</cardinality>
        </parameter>
    </specficParameters>
</tns:DynamicMediationTask>
```

16.3.4.3 *Listing of all dynamic mediation tasks*

Verb	URI
GET	/DynamicMediationTasks/BUILD_PAYLOAD_NOOP_WITH_PARAMS
GET	/DynamicMediationTasks/BUILD_PAYLOAD_NOOP_WITH_SAXSD
GET	/DynamicMediationTasks/DATA_MEDIATION_AND_INVOCATION_NOOP_WITH_PARAMS
GET	/DynamicMediationTasks/DATA_MEDIATION_AND_INVOCATION_NOOP_WITH_SAXSD

ID:BUILD_PAYLOAD_NOOP_WITH_PARAMS

```xml
<?xml version="1.0" encoding="UTF-8"?><br \>
<tns:DynamicMediationTask
xmlns:tns="http://www.fiware.thalesgroup.com/DynamicMediationTasks"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.fiware.thalesgroup.com/DynamicMediati
onTasks DynamicMediationTasks.xsd "><br \>
    <id>BUILD_PAYLOAD_NOOP_WITH_PARAMS</id>

<linkrel>/DynamicMediationTasks/BUILD_PAYLOAD_NOOP_WITH_PARAMS</link
rel>

<linkhref>http://example.org/DynamicMediationTasks/BUILD_PAYLOAD_NOO
P_WITH_PARAMS</linkhref>
    <description>This method allows to build dynamically the payload
to send to an identified service. taken as input a list a value
and<br \> the semantic operation that the client provide. The data
given will be transformed into the service payload.</description>
    <specficParameters>
        <parameter>
            <name>usdUri</name>
            <description>The target service URL of the file
descriptor in USDL format with references towards a sawsdl
semantic<br \> definition</description>
            <cardinality>UNBOUNDED</cardinality>
        </parameter>
        <parameter>
            <name>usdUri</name>
            <description>The service id inside the USDL file. Need
this id if the USDL reference more than one service.</description>
        </parameter>
        <parameter>
            <name>operationConcept</name>
            <description>The client need to access a certain
operation of the service. In the case the client does not know the
service<br \> operation syntactic name, he provides the semantic
version of this operation.</description>
        </parameter>
        <parameter>
```

```
            <name>semanticInput</name>
            <description>The client has to give the list of
parameter he wants to send to the service. The payload will be
created <br \>based on these information.The parameter value must be
given with the semantic description of the parameter.</description>
            <cardinality>UNBOUNDED</cardinality>
        </parameter>
    </specficParameters>
</tns:DynamicMediationTask>
```

ID:BUILD_PAYLOAD_NOOP_WITH_SAXSD

```
<?xml version="1.0" encoding="UTF-8"?><br \>
<tns:DynamicMediationTask
xmlns:tns="http://www.fiware.thalesgroup.com/DynamicMediationTasks"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.fiware.thalesgroup.com/DynamicMediati
onTasks DynamicMediationTasks.xsd "><br \>
    <id>BUILD_PAYLOAD_NOOP_WITH_SAXSD</id>

<linkrel>/DynamicMediationTasks/BUILD_PAYLOAD_NOOP_WITH_SAXSD</linkr
el>

<linkhref>http://example.org/DynamicMediationTasks/BUILD_PAYLOAD_NOO
P_WITH_SAXSD</linkhref>
    <description>This method allows to build dynamically the payload
to send to an identified service. taken as input the client
payload<br \> to transform with itssemantic
description.</description>
    <specficParameters>
        <parameter>
            <name>usdUri</name>
            <description>The target service URL of the file
descriptor in USDL format with references towards a sawsdl
semantic<br \> definition</description>
            <cardinality>UNBOUNDED</cardinality>
        </parameter>
        <parameter>
            <name>usdUri</name>
            <description>The service id inside the USDL file. Need
this id if the USDL reference more than one service.</description>
        </parameter>
        <parameter>
            <name>operationConcept</name>
            <description>The client need to access a certain
operation of the service. In the case the client does not know the
service<br \> operation syntactic name,he provides the semantic
version of this operation.</description>
        </parameter>
        <parameter>
            <name>saxsd_content</name>
```

```
                    <description>The client must give the SAXSD description
of the data (payload) he wants to transforms (content or url). <br
\>He gives here the value of the SAXSD.</description>
        </parameter>
        <parameter>
            <name>saxsd_url</name>
            <description>The client must give the SAXSD description
of the data he wants to transforms (content or url).He gives here
<br \>the url towards the accessible SAXSD resource.</description>
        </parameter>
        <parameter>
            <name>payload</name>
            <description>The client must gives the data he wants to
transform. He gives here the value of the payload he
sends.</description>
        </parameter>
    </specficParameters>
</tns:DynamicMediationTask>
```

ID:DATA_MEDIATION_AND_INVOCATION_NOOP_WITH_PARAMS

```
<?xml version="1.0" encoding="UTF-8"?><br \>
<tns:DynamicMediationTask
xmlns:tns="http://www.fiware.thalesgroup.com/DynamicMediationTasks"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.fiware.thalesgroup.com/DynamicMediati
onTasks DynamicMediationTasks.xsd "><br \>
    <id>DATA_MEDIATION_AND_INVOCATION_NOOP_WITH_PARAMS</id>

<linkrel>/DynamicMediationTasks/DATA_MEDIATION_AND_INVOCATION_NOOP_W
ITH_PARAMS</linkrel>

<linkhref>http://example.org/DynamicMediationTasks/DATA_MEDIATION_AN
D_INVOCATION_NOOP_WITH_PARAMS</linkhref>
    <description>This method allows to search and invoke a service
capability with the given operation concept. After the invocation
<br \>the result is adapted to he given wanted output.</description>
    <specficParameters>
        <parameter>
            <name>usdUri</name>
            <description>The target service URL of the file
descriptor in USDL format with references towards a sawsdl <br
\>semantic definition</description>
            <cardinality>UNBOUNDED</cardinality>
        </parameter>
        <parameter>
            <name>usdUri</name>
            <description>The service id inside the USDL file. Need
this id if the USDL reference more than one service.</description>
        </parameter>
        <parameter>
            <name>operationConcept</name>
            <description>The client need to access a certain
operation of the service. In the case the client does not know
```

D.3.1.1b FI-WARE GE Open Specification   Page 251

```
the<br \> service operation syntactic name, he provides the semantic
version of this operation.</description>
        </parameter>
        <parameter>
            <name>semanticInput</name>
            <description>The client has to give the list of
parameter he wants to send to the service. The payload will be
created based<br \> on these information. The parameter value must
be given with the semantic description of the
parameter.</description>
            <cardinality>UNBOUNDED</cardinality>
        </parameter>
        <parameter>
            <name>semanticOuput</name>
            <description>The client has to give the list of
parameter he wants to receive from the service. The parameter value
must<br \> be given with the semantic description of the
parameter.</description>
            <cardinality>UNBOUNDED</cardinality>
        </parameter>
    </specficParameters>
</tns:DynamicMediationTask>
```

ID:DATA_MEDIATION_AND_INVOCATION_NOOP_WITH_SAXSD

```
<?xml version="1.0" encoding="UTF-8"?><br \>
<tns:DynamicMediationTask
xmlns:tns="http://www.fiware.thalesgroup.com/DynamicMediationTasks"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.fiware.thalesgroup.com/DynamicMediati
onTasks DynamicMediationTasks.xsd "><br \>
    <id>DATA_MEDIATION_AND_INVOCATION_NOOP_WITH_SAXSD</id>

<linkrel>/DynamicMediationTasks/DATA_MEDIATION_AND_INVOCATION_NOOP_W
ITH_SAXSD</linkrel>

<linkhref>http://example.org/DynamicMediationTasks/DATA_MEDIATION_AN
D_INVOCATION_NOOP_WITH_SAXSD</linkhref>
    <description>This method allows to search and invoke a service
capability with the given operation concept. After the <br
\>invocation the result is adapted to the given wanted
output.</description>
    <specficParameters>
        <parameter>
            <name>usdUri</name>
            <description>The target service URL of the file
descriptor in USDL format with references towards a sawsdl <br
\>semantic definition</description>
            <cardinality>UNBOUNDED</cardinality>
        </parameter>
        <parameter>
            <name>usdUri</name>
```

```
                <description>The service id inside the USDL file. Need
this id if the USDL reference more than one service.</description>
        </parameter>
        <parameter>
            <name>operationConcept</name>
            <description>The client need to access a certain
operation of the service. In the case the client does not know<br \>
the service operation syntactic name, he provides the semantic
version of this operation.</description>
        </parameter>
        <parameter>
            <name>saxsd_content</name>
            <description>The client must give the SAXSD description
of the data (payload) he wants to transforms (content or url)<br \>
and the awaited response format.He gives here the value of the
SAXSD.</description>
        </parameter>
        <parameter>
            <name>saxsd_url</name>
            <description>The client must give the SAXSD description
of the data he wants to transforms (content or url) and <br \>the
awaited response format. He gives here the url towards the
accessible SAXSD resource.</description>
        </parameter>
        <parameter>
            <name>payload</name>
            <description>The client must gives the data he wants to
transform. He gives here the value of the payload <br \>he
sends.</description>
        </parameter>
        <parameter>
            <name>wanted_response_element</name>
            <description>The client gives here the element reference
inside the SAXSD description resource that contains <br \>the
description of the wanted response format.</description>
        </parameter>
    </specficParameters>
</tns:DynamicMediationTask>
```

17 FIWARE OpenSpecification Apps Registry

You can find the content of this chapter as well in the wiki of fi-ware.

Name	FIWARE.OpenSpecification.Apps.Registry
Chapter	Apps,
Catalogue-Link to Implementation	FI-WARE Registry
Owner	SAP, Torsten Leidig

17.1 Preface

Within this document you find a self-contained open specification of a FI-WARE generic enabler, please consult as well the FI-WARE_Product_Vision, the website on http://www.fi-ware.eu and similar pages in order to understand the complete context of the FI-WARE project.

17.2 Copyright

- Copyright © 2012 by SAP

17.3 Legal Notice

Please check the following Legal Notice to understand the rights to use these specifications.

SAP strives to make the specifications of this Generic Enabler available under IPR rules that allow for a exploitation and sustainable usage both in Open Source as well as proprietary, closed source products to maximize adoption.

17.4 Overview

While the Repository Enabler is used to store complete service descriptions, which are more or less static and change only rarely, the Registry Enabler is used to store information on service instances necessary for run-time execution. Discovering entities and their description in an open distributed system often is achieved via registries, which have a well-known address. The registry serves as a kind of directory and for example can store detailed settings for concrete infrastructure components as well as information about human or computing agents. The information can range from stable to extremely volatile and is needed to make specific settings for and adjustments to other components in the platform. For example, the Registry can be used by the Marketplace in order to register stores, providers, persons, infrastructure components and more. The functionality and purpose of the Registry GE is comparable to the Microsoft Windows Registry [REG3] or the Light-weight Directory Access Protocol LDAP [REG1] or the UDDI Business Registry [REG4]. The main difference is that the Registry GE is fully relying on standard Web protocols and has a simpler data model.

17.4.1   Target usage

The Registry acts as a universal directory of information used for the maintenance, administration, deployment and retrieval of services. Existing (running) service endpoints as well as information to create an actual service instance and endpoint are registered. This GE will be used by potentially all GE in the Apps Chapter in order to build a common database of run-time configuration options and properties. It can also be used by GE of other chapters, such as the Cloud, Security, Data or IoT to announce their instance specific information to the rest of the platform components. There could be multiple instances of the Registry for different purposes and usage domains.

17.4.2   Rationale

The Registry has different requirements according to scalability and performance in comparison to the Repository. Highly volatile information can cause a lot of workload for a repository and needs to be dealt with quickly. On the other hand, the number of clients for a repository is usually much smaller.

17.4.3   Background

Registries are quite a common pattern in software architectures. With in the internet protocol stack defined by the IETF RFC, LDAP (Light-weight Directory Access Protocol) [REG1] is a common technical realization of the registry functionality. Other examples of registry implementations are UDDI [REG4]and the Windows Registry database [REG3].

17.5   Basic Concepts

17.5.1   Register and Deregister Entries

This component is mainly used by resource providers to register and deregister their resources as entries in the Registry.

17.5.2   Retrieving Registry Entries

The Retrieving Registry Entries component is responsible for retrieving registry entries. A client service can ask for specific settings and options of the operating environment.

17.5.3   Data Model

The basic data elements of the registry are the **Registry Entry** containing the actual information and the **Registry Key** to access the data entries in the Registry. A registry entry can be a single atomic piece of data or a structured data such as a record of properties. The exact data model and its encoding will be defined in the interface specification. The schema (the exact names of properties and their value encoding) is the matter of the application developer or the community of developers in a respective application domain.

The Registry Key is used for accessing individual entries or a collection of entries is often organized as path into an underlying registry internal organization such as a tree.

17.6   Architecture

The Registry is a searchable index of the Repository GE.
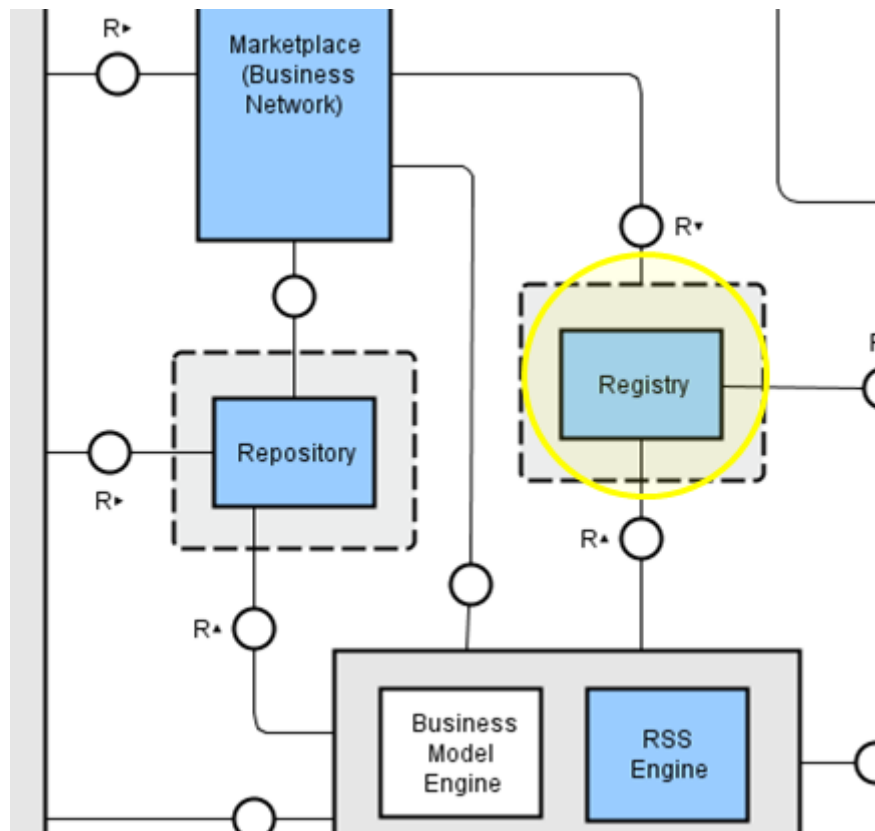
Registry / Repository in the context of the Business Framework

17.7   Main Operations

The following diagram shows an example sequence how a user or other GEs can register, retrieve, and deregister a registry entry.
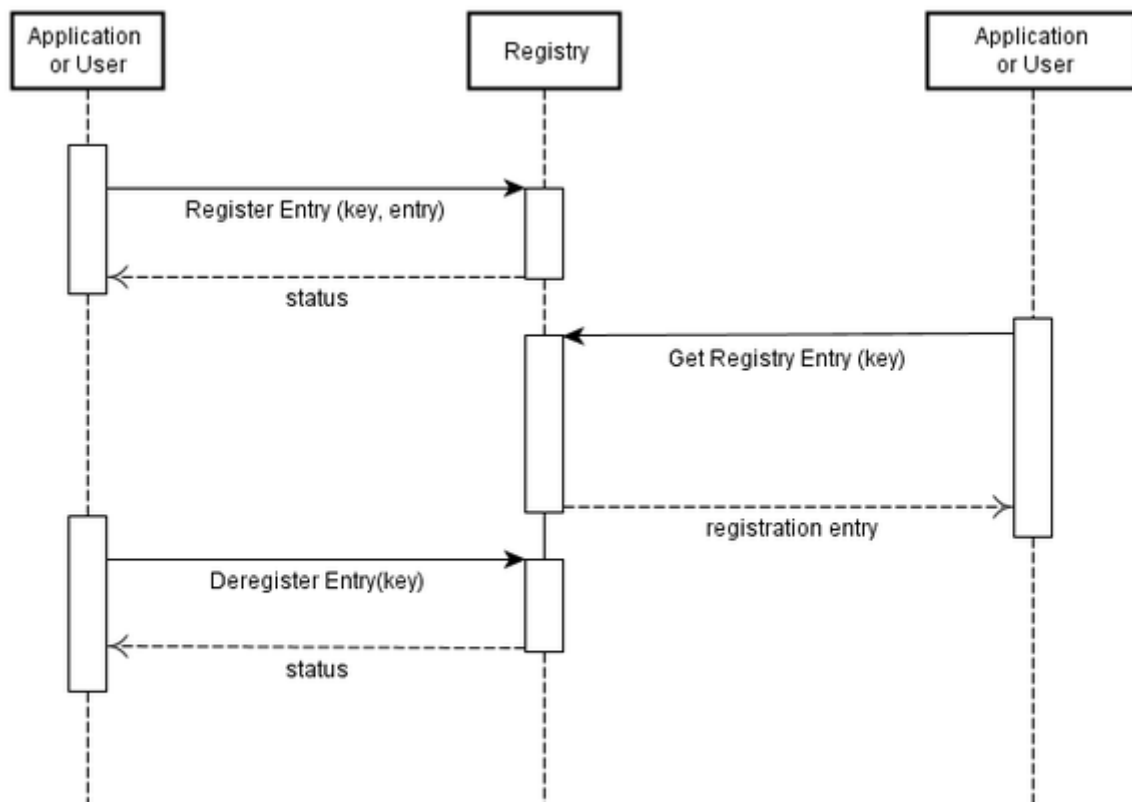
Example sequence of Registry operations

17.7.1   Register and Deregister Entries

The **Register Entry** operation is used to write or update a register information entry into the Registry. Two parameters are essentially needed:

- *key* - The Registry Key of the entry to be registered. A key can exhibit an organizational structure such as a tree.

- *entry* - The Register Entry to be registered. The entry is usually a list of name/value pairs.

For the **Deregister Entry** operation only the Registry Key of the Registry Entry is needed:

- *key* - Registry Key of the entry to be de-registered

17.7.2   Retrieving Registry Entries

It must be possible to directly retrieve a Registry Entry using a unique Key using the **Get Registry Entry** operation. In this case one parameter is sufficient:

- *key* - Registry Key of the entry to be retrieved

The **Query Registry Entries** operation allows the retrieval of Registry Entries matching a filter expression:

- *filter* - A filter expression to select entries to be retrieved.

17.7.3   Basic Design Principles

An implementation for the Registry might take different design decisions and technological approaches, depending on the non-functional requirements of the repository. A registry implementation might be highly distributed and scalable, if it is used by many parties on a global scale. Also the database schema might be of different complexity depending on the data requirements of the use case. Prominent example technologies which have a distributed nature are LDAP, UDDI, or distributed key/value stores for large amounts of records such as MongoDB, CouchDB, or Cassandra. The Registry specification follows the separation of concerns principle. So it is possible to supplement it with an authentication and authorization system such as OAuth [REG2].

17.8   References

REG1	LDAP protocol specifications(RFCs 4510,4512,4514,4516,4517)
REG2	OAuth2 protocol (http://tools.ietf.org/html/draft-ietf-oauth-v2-30)
REG3	Microsoft Windows Registry (http://msdn.microsoft.com/msdnmag/issues/1100/Registry/)
REG4	UDDI Business Registry (http://en.wikipedia.org/wiki/Universal_Description_Discovery_and_Integration)

17.9   Detailed Specifications

17.9.1   Open API Specifications

- FIWARE.OpenSpecification.Apps.RegistryREST

17.9.2   Other Relevant Specifications

17.10   Re-utilised Technologies/Specifications

Because the registry can be used to provide central consistent information base of run-time information it might require authentication and authorization in order to safeguard the

content. For this reasons it need to be combined with a pluggable authenticaton/authentication provider.

FI-Ware for example offers an Identity GE providing API authorization with bearer tokens of the OAuth2 protocol (http://tools.ietf.org/html/draft-ietf-oauth-v2-30).

17.11   Terms and definitions

This section comprises a summary of terms and definitions introduced during the previous sections. It intends to establish a vocabulary that will be help to carry out discussions internally and with third parties (e.g., Use Case projects in the EU FP7 Future Internet PPP). For a summary of terms and definitions managed at overall FI-WARE level, please refer to [FIWARE Global Terms and Definitions](#)

- **Aggregator (Role):** Supports domain specialists and third-parties in aggregating services and apps for new and unforeseen opportunities and needs. It does so by providing the dedicated tooling for aggregating services at different levels: UI, service operation, business process or business object levels.

- **Application:** Applications in FI-Ware are composite services that have a IT supported interaction interface (user interface). In most cases consumers do not buy the application they rather buy the right to use the application (user license).

- **Broker (Role):** The business network's central point of service access, being used to expose services from providers that are to be delivered through the Broker's service delivery functionality. The broker is the central instance for enabling monetization.

- **Business Element:** Core element of a business model, such as pricing models, revenue sharing models, promotions, SLAs, etc.

- **Business Framework:** Set of concepts and assets responsible for supporting the implementation of innovative business models in a flexible way.

- **Business Model:** Strategy and approach that defines how a particular service/application is supposed to generate revenue and profit. Therefore, a Business Model can be implemented as a set of business elements which can be combined and customized in a flexible way and in accordance to business and market requirements and other characteristics.

- **Business Process:** Set of related and structured activities producing a specific service or product, thereby achieving one or more business objectives. An operational business process clearly defines the roles and tasks of all involved parties inside an organization to achieve one specific goal.

- **Business Role:** Set of responsibilities and tasks that can be assigned to concrete business role owners, such as a human person or a software component.

- ''''*Channel: Resources through which services are accessed by end users. Examples for well-known channels are Web sites/portals, web-based brokers (like iTunes, eBay and Amazon), social networks (like Facebook, LinkedIn and MySpace), mobile channels (Android, iOS) and work centres. The mode of access to these channels is governed by technical channels like the Web, mobile devices and voice response, where each of these channels requires its own specific workflow.*

- **Channel Maker (Role):** Supports parties in creating outlets (the Channels) through which services are consumed, i.e. Web sites, social networks or mobile platforms. The Channel Maker interacts with the Broker for discovery of services during the process of creating or updating channel specifications as well as for storing channel

specifications and channeled service constraints back in the Broker.

- **Composite Service (composition):** Executable composition of business back-end MACs. Common composite services are either orchestrated or choreographed. Orchestrated compositions are defined by a centralized control flow managed by a unique process that orchestrates all the interactions (according to the control flow) between the external services that participate in the composition. Choreographed compositions do not have a centralized process, thus the services participating in the composition autonomously coordinate each other according to some specified coordination rules. Backend compositions are executed in dedicated process execution engines. Target users of tools for creating Composites Services are technical users with algorithmic and process management skills.

- **Consumer (Role):** Actor who searches for and consumes particular business functionality exposed on the Web as a service/application that satisfies her own needs.

- **Desktop Environment:** Multi-channel client platform enabling users to access and use their applications and services.

- **Event-driven Composition:** Components concerned with the composition of business logic which is driven by asynchronous events. This implies run-time selection of MACs and the creation/modification of orchestration workflows based on composition logic defined at design-time and adapted to context and the state of the communication at run-time.

- **Front-end/Back-end Composition:** Front-end compositions define a front-end application as an aggregation of visual mashable application pieces (named as widgets, gadgets, portlets, etc.) and back-end services. Front-end compositions interact with end-users, in the sense that front-end compositions consume data provided by the end-users and provide data to them. Thus the frontend composition (or mashup) will have direct influence on the application look and feel; every component will add a new user interaction feature.

- Back-end compositions define a back-end business service (also known as process) as an aggregation of backend services as defined for service composition term, the end-user being oblivious to the composition process. While back-end components represent atomization of business logic and information processing, front-end components represent atomization of information presentation and user interaction.

- **Gateway (Role):** The Gateway role enables linking between separate systems and services, allowing them to exchange information in a controlled way despite different technologies and authoritative realms. A Gateway provides interoperability solutions for other applications, including data mapping as well as run-time data store-forward and message translation. Gateway services are advertised through the Broker, allowing providers and aggregators to search for candidate gateway services for interface adaptation to particular message standards. The Mediation is the central generic enabler. Other important functionalities are eventing, dispatching, security, connectors and integration adaptors, configuration, and change propagation.

- **Hoster (Role):** Allows the various infrastructure services in cloud environments to be leveraged as part of provisioning an application in a business network. A service can be deployed onto a specific cloud using the Hoster's interface. This enables service providers to re-host services and applications from their on-premise environments to cloud-based, on-demand environments to attract new users at much lower cost.

- **Marketplace:** Part of the business framework providing means for service providers

to publish their service offerings and service consumers to compare and select a specific service implementation. A marketplace can offer services from different stores and thus different service providers. The actual buying of a specific service is handled by the related service store.

- **Mashup:** Executable composition of front-end MACs. There are several kinds of mashups, depending on the technique of composition (spatial rearrangement, wiring, piping, etc.) and the MACs used. They are called application mashups when applications are composed to build new applications and services/data mash-ups if services are composed to generate new services. While composite service is a common term in backend services implementing business processes, the term 'mashup' is widely adopted when referring to Web resources (data, services and applications). Front-end compositions heavily depend on the available device environment (including the chosen presentation channels). Target users of mashup platforms are typically users without technical or programming expertise.

- **Mashable Application Component (MAC):** Functional entity able to be consumed executed or combined. Usually this applies to components that will offer not only their main behaviour but also the necessary functionality to allow further compositions with other components. It is envisioned that MACs will offer access, through applications and/or services, to any available FI-WARE resource or functionality, including gadgets, services, data sources, content, and things. Alternatively, it can be denoted as 'service component' or 'application component'.

- **Mediator:** A mediator can facilitate proper communication and interaction amongst components whenever a composed service or application is utilized. There are three major mediation area: Data Mediation (adapting syntactic and/or semantic data formats), Protocol Mediation (adapting the communication protocol), and Process Mediation (adapting the process implementing the business logic of a composed service).

- **Monetization:** Process or activity to provide a product (in this context: a service) in exchange for money. The Provider publishes certain functionality and makes it available through the Broker. The service access by the Consumer is being accounted according to the underlying business model and the resulting revenue is shared across the involved service providers.

- **Premise (Role):** On-Premise operators provide in-house or on-site solutions, which are used within a company (such as ERP) or are offered to business partners under specific terms and conditions. These systems and services are to be regarded as external and legacy to the FI-Ware platform because they do not conform to the architecture and API specifications of FI-Ware. They will only be accessible to FI-Ware services and applications through the Gateway.

- **Prosumer:** A user role able to produce, share and consume their own products and modify/adapt products made by others.

- **Provider (Role):** Actor who publishes and offers (provides) certain business functionality on the Web through a service/application endpoint. This role also takes care of maintaining this business functionality.

- **Registry and Repository:** Generic enablers that able to store models and configuration information along with all the necessary meta-information to enable searching, social search, recommendation and browsing, so end users as well as services are able to easily find what they need.

- **Revenue Settlement:** Process of transferring the actual charges for specific service

consumption from the consumer to the service provider.

- **Revenue Sharing:** Process of splitting the charges of particular service consumption between the parties providing the specific service (composition) according to a specified revenue sharing model.

- **Service:** We use the term service in a very general sense. A service is a means of delivering value to customers by facilitating outcomes customers want to achieve without the ownership of specific costs and risks. Services could be supported by IT. In this case we say that the interaction with the service provider is through a technical interface (for instance a mobile app user interface or a Web service). Applications could be seen as such IT supported Services that often ar also composite services.

- **Service Composition:** in SOA domain, a service composition is an added value service created by aggregation of existing third party services according to some predefined work and data flow. Aggregated services provide specialized business functionality on which the service composition functionality has been split down.

- **Service Delivery Framework:** Service Delivery Framework (or Service Delivery Platform (SDP)) refers to a set of components that provide service delivery functionality (such as service creation, session control & protocols) for a type of service. In the context of FI-WARE, it is defined as a set of functional building blocks and tools to (1) manage the lifecycle of software services, (2) creating new services by creating service compositions and mashups, (3) providing means for publishing services through different channels on different platforms, (4) offering marketplaces and stores for monetizing available services and (5) sharing the service revenues between the involved service providers.

- **Service Level Agreement (SLA):** A service level agreement is a legally binding and formally defined service contract between a service provider and a service consumer, specifying the contracted qualitative aspects of a specific service (e.g. performance, security, privacy, availability or redundancy). In other words, SLAs not only specify that the provider will just deliver some service, but that this service will also be delivered on time, at a given price, and with money back if the pledge is broken.

- **Service Orchestration:** in SOA domain, a service orchestration is a particular architectural choice for service composition where a central orchestrated process manages the service composition work and data flow invocations the external third party services in the order determined by the work flow. Service orchestrations are specified by suitable orchestration languages and deployed in execution engines who interpret these specifications.

- **Store:** An external component integrated with the business framework offering a set of services that are published to a selected set of marketplaces. The store thereby holds the service portfolio of a specific service provider. In case a specific service is purchased on a service marketplace, the service store handles the actual buying of a specific service (as a financial business transaction).

- **Unified Service Description Language (USDL):** USDL is a platform-neutral language for describing services, covering a variety of service types, such as purely human services, transactional services, informational services, software components, digital media, platform services and infrastructure services. The core set of language modules offers the specification of functional and technical service properties, legal and financial aspects, service levels, interaction information and corresponding participants. USDL is offering extension points for the derivation of domain-specific service description languages by extending or changing the available language

modules.

- **Registry Key:** A unique identifier for accessing entries in the Registry. The Registry is often given as a hierarchical naming schema or path.

- **Registry Entry:** Data that is stored for a specific Registry Key. This data is usually a record of key, value pairs defining a property and its value.

18   FIWARE OpenSpecification Apps RegistryREST

You can find the content of this chapter as well in the wiki of fi-ware.

18.1   Introduction to the *Registry* API

The FI-WARE Generic Enabler Specification are owned by different partners. Therefore, different Legal Notices might apply. Please check for each FI-WARE Generic Enabler Specification the Legal Notice attached. For this FI-WARE Generic Enabler Specification, this Legal Notice applies.

SAP strives to make the specifications of this Generic Enabler available under IPR rules that allow for a exploitation and sustainable usage both in Open Source as well as proprietary, closed source products to maximize adoption.

18.1.1   Registry API Core

The Registry API is a RESTful, resource-oriented API accessed via HTTP that uses various representations for information interchange. The Registry Enabler is used to store information on service instances necessary for run-time execution.

18.1.2   Intended Audience

This specification is intended for both software developers and implementers of the FI-WARE Business Framework. For the former, this document provides a full specification of how to interoperate with products that implement the Repository API. For the latter, this specification indicates the interface to be implemented and provided to clients. Software developers intending to build applications on top of FI-WARE Enablers will implement a client of the interface specification. Implementers of the GE will implement a service of the interface specification.

To use this information, the reader should firstly have a general understanding of the Generic Enabler service Registry Enabler. You should also be familiar with:

- RESTful web services
- HTTP/1.1
- JSON and/or XML data serialization formats.
- LDAP

18.1.3   API Change History

This version of the Registry API Guide replaces and obsoletes all previous versions. The most recent changes are described in the table below:

Revision Date	Changes Summary
Apr 20, 2012	• Initial version

18.1.4   How to Read This Document

It is assumed that the reader is familiar with the REST architecture style. Within the document, some special notations are applied to differentiate some special words or concepts. The following list summarizes these special notations.

- A bold, mono-spaced font is used to represent code or logical entities, e.g., HTTP method (`GET, PUT, POST, DELETE`).

- An italic font is used to represent document titles or some other kind of special text, e.g., *URI.*

- Variables are represented between brackets, e.g. {id} and in italic font. The reader can replace the id with an appropriate value.

For a description of some terms used along this document, see Registry Enabler.

18.1.5   Additional Resources

You can download the most current version of this document from the FI-WARE API specification website at Registry API . For more details about the Registry GE that this API is based upon, please refer to the High Level Description. Related documents, including an Architectural Description, are available at the same site.

18.2   General *Registry* API Information

18.2.1   Resources Summary

The registry is structured into core objects, which are called *registry entries.* These objects constitute also the granularity of access control. A registry entry, uniquely identified by its *distinguished name*, can hold a number of *attributes.*

18.2.2   Authentication

Each HTTP request against the *Registry GE* requires the inclusion of specific authentication credentials. The specific implementation of this API may support multiple authentication schemes (OAuth, Basic Auth, Token) and will be determined by the specific provider that implements the GE. Please contact with it to determine the best way to authenticate against this API. Some authentication schemes may require that the API operate using SSL over HTTP (HTTPS).

18.2.3   Authorization

It is assumed that access to the registry is controlled by a authorization mechanisms in order to ensure that only authorized clients can read/modify/write specific information. The specification of a concrete authorization mechanism is out of scope for this document. Within the FI-WARE testbed, the authorization methods of the Security Chapter enablers will be supported by the Registry implementation.

18.2.4   Representation Format

The *Registry* API supports XML/RDF, Turtle, JSON, Atom HTML for delivering information for registry entries. The request format is specified using the Content-Type header and is

required for operations that have a request body. The response format can be specified in requests using the Accept header. Note that it is possible for a response to be serialized using a format different from the request (see example below).

If no Content-Type is specified, the content is delivered in the format that was chosen to upload the resource.

The interfaces should support data exchange through multiple formats:

- *text/plain* - A linefeed separated list of elements for easy mashup and scripting.

- *text/html* - An human-readable HTML rendering of the results of the operation as output format.

- *application/json* - A JSON representation of the input and output for mashups or JavaScript-based Web Apps

- *application/rdf+xml* - A RDF description of the input and output.

In a concrete implementation of this GE other formats like RSS, Atom, etc. may also be possible.

18.2.5   Representation Transport

Resource representation is transmitted between client and server by using HTTP 1.1 protocol, as defined by IETF RFC-2616. Each time an HTTP request contains payload, a Content-Type header shall be used to specify the MIME type of wrapped representation. In addition, both client and server may use as many HTTP headers as they consider necessary.

18.2.6   Resource Identification

The Distinguished Entry Name (DEN) is used to unambiguously identify registry entries. In analogy to the LDAP protocol (RFCs 4510,4512,4514,4516,4517) we assume distinguished entry names can be expressed in a hierarchical way.

Example:

/c=de/o=University%20of%20Michigan - is a DEN similar to an LDAP DN

/de/University%20of%20Michigan - is an alternative representation of the DEN assuming that there is a default hierarchy

18.2.7   Links and References

18.2.7.1   *Web citizen*

The registry is relying on Web principles:

- URI to identify resources

- consistent URI structure based on REST style protocol

- HTTP content negotiation to allow the client to choose the appropriate data format supporting HTML, RDF, XML, RSS, JSON, Turtle, ...

- Human readable output format using HTML rendering ('text/html' accept header) including hyperlinked representation

- Use of HTTP response codes including ETags (proper caching)

- Linked Data enablement supporting RDF input and output types

18.2.8   Paginated Collections

In order to reduce the load on the service, we can decide to limit the number of elements to return when it is too big. This section explain how to do that using for example a limit parameter (optional) and a last parameter (optional) to express which is the maximum number of element to return and which was the last element to see.

These operations will have to cope with the possibility to have over limit fault (413) or item not found fault (404).

18.2.9   Limits

We can manage the capacity of the system in order to prevent the abuse of the system through some limitations. These limitations will be configured by the operator and may differ from one implementation to other of the GE implementation.

18.2.9.1   *Rate Limits*

These limits are specified both in human readable wild-card and in regular expressions and will indicate for each HTTP verb which will be the maximum number of operations per time unit that a user can request. After each unit time the counter is initialized again. In the event a request exceeds the thresholds established for your account, a 413 HTTP response will be returned with a Retry-After header to notify the client when they can attempt to try again.

18.2.10   Extensions

The Registry could be extended in the future. At the moment, we foresee the following resource to indicate a method that will be used in order to allow the extensibility of the API. This allow the introduction of new features in the API without requiring an update of the version, for instance, or to allow the introduction of vendor specific functionality.

Verb	URI	Description
GET	/extensions	List of all available extensions

18.2.11   Faults

18.2.11.1 *Synchronous Faults*

Error codes are returned in the body of the response. The description section returns a human-readable message for displaying end users.

Example:

```
<exception>
      <description>Resource Not found</description>
```

```
      <errorCode>404</errorCode>

      <reasonPhrase>Not Found</reasonPhrase>

</exception>
```

Fault Element	Associated Error Codes	Expected in All Requests?
Unauthorized	403	YES
Not Found	404	YES
Limit Fault	413	YES
Internal Server error	50X	YES

18.3   API Operations

18.3.1   Retrieving Registry Information

18.3.1.1   *Read registry entry*

Here we start with the description of the operation following the next table:

Verb	URI	Description
GET	/{DistinguishedEntryName}? filter={FilterExpression}&attributes={AttributeList}&scope={Scope}	Get registry entry information:

Parameters

FilterExpression - Expression for filtering registered entries under a relative distinguished entry name.

AttributeList - Comma-separated list of attribute names which should be returned.

Scope - Return entries of the DEN only or in the whole sub-tree (one of "base" / "one" / "sub")

Example

GET /de/service/stores/?attributes=Name,serviceResource,endpoint

Returns the name, service description URL, and service endpoint URL for all services registered under "/de/service/stores".

Result Format

Accept: application/json:

```
[ { DEN: "/de/service/stores/store1",
    Name: "Store1 Name",
    service: "http://fiware.org/usdl/servicestorexyz",
```

```
     endpoint:                              "http://fiware-
platform.org/service/store1/instance4711"
  },
  ...
]
```

Status Codes

200 OK

> The request was handled successfully and transmitted in response message.

400 Bad Request

> The request cannot be fulfilled due to bad syntax.

404 Not Found

> The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible.

500 Internal Server Error

> A generic error message, given when no more specific message is suitable.

18.3.2   Modifying Entries

18.3.2.1   *Creating and Updating*

Verb	URI	Description
PUT	/{DistinuguishedEntryName}	Create or update a resource or a number of resources

Request Body

The request body of a PUT operation should contain the set of attributes of the entry. E.g. if Content-type was "application/json":

```
{
  "{attributeName1}": "AttributeValue1",
  "{attributeName2}": "AttributeValue2",
  ...
}
```

or for a number of resources

```
[{
  "RDN": "{RelativeDistinguishedName",
  "{attributeName1}": "AttributeValue1",
  "{attributeName2}": "AttributeValue2",
  ...
 },
 ...
]
```

respectively.

Status Codes

201 Created

The request has been fulfilled and resulted in a new resource being created.

204 No Content

The server successfully processed the request, but is not returning any content.

400 Bad Request

The request cannot be fulfilled due to bad syntax.

404 Not Found

The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible.

409 Conflict

Indicates that the request could not be processed because of conflict in the request, such as an edit conflict.

500 Internal Server Error

A generic error message, given when no more specific message is suitable.

Adding Information

Verb	URI	Description
POST	/{DistinguishedEntryName}	Add attributes to a registry entry

Request Body

The request body contains the attributes to be added to an entry:

```
{
  "{attributeName1}": "AttributeValue1",
  "{attributeName2}": "AttributeValue2",
  ...
}
```

Status Codes

201 Created

The request has been fulfilled and resulted in a new resource being created.

204 No Content

The server successfully processed the request, but is not returning any content.

400 Bad Request

The request cannot be fulfilled due to bad syntax.

404 Not Found

The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible.

409 Conflict

Indicates that the request could not be processed because of conflict in the request, such as an edit conflict.

500 Internal Server Error

A generic error message, given when no more specific message is suitable.

18.3.3   Deleting Registry Information

18.3.3.1   *Deleting Entries*

Verb	URI	Description
DELETE	/{DistinuguishedEntryName}	Delete a registry entry

Status Codes

200 OK

The request was handled successfully and transmitted in response message.

400 Bad Request

The request cannot be fulfilled due to bad syntax.

404 Not Found

The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible.

500 Internal Server Error

A generic error message, given when no more specific message is suitable.

18.3.3.2   *Delete Attributes of an Entry*

Verb	URI	Description
DELETE	/{DistinguishedEntryName}?attributes={AttributeNames}	Delete attributes of a registry entry

Parameters

{AttributeNames} contains a comma-separated list of attribute names to be deleted from the entry.

Status Codes

200 OK

The request was handled successfully and transmitted in response message.

400 Bad Request

The request cannot be fulfilled due to bad syntax.

404 Not Found

The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible.

500 Internal Server Error

A generic error message, given when no more specific message is suitable.

19 FI-WARE Open Specifications Legal Notice

You can find the content of this chapter as well in the wiki of fi-ware.

19.1.1.1 *General Information*

"FI-WARE Partners" refer to Parties of the FI-WARE Project in accordance with the terms of the FI-WARE Consortium Agreement"

19.1.1.2 *Use Of Specification - Terms, Conditions & Notices*

The material in this specification details a FI-WARE Generic Enabler Specification (hereinafter "Specification") in accordance with the terms, conditions and notices set forth below. This Specification does not represent a commitment to implement any portion of this Specification in any company's products. The information contained in this Specification is subject to change without notice.

19.1.1.3 *Copyright License*

Subject to all of the terms and conditions below, the copyright holders in this Specification hereby grant you, the individual or legal entity exercising permissions granted by this License, a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license, royalty free (without the right to sublicense) under its respective copyrights incorporated in the Specification, to copy and modify this Specification and to distribute copies of the modified version, and to use this Specification, to create and distribute special purpose specifications and software that are an implementation of this Specification.

19.1.1.4 *Patent Information*

The FI-WARE Project Partners shall not be responsible for identifying patents for which a license may be required by any FI-WARE Specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. FI-WARE specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

19.1.1.5 *General Use Restrictions*

Any unauthorized use of this Specification may violate copyright laws, trademark laws, and communications regulations and statutes. This Specification contains information which is protected by copyright. All Rights Reserved. This Specification shall not be used in any form or for any other purpose different from those herein authorized, without the permission of the respective copyright owners.

This Specification shall not be used in any form or for any other purpose different from those herein authorized, without the permission of the respective copyright owners.

For avoidance of doubt, the rights granted are only those expressly stated in this Section herein. No other rights of any kind are granted by implication, estoppel, waiver or otherwise

19.1.1.6 *Disclaimer Of Warranty*

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE FI-WARE PARTNERS MAKE NO

19.1.1.7   *Trademarks*

19.1.1.8   *Issue Reporting*

This Specification is subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Procedure described on the web page http://www.fi-ware.eu.

20 Open Specifications Interim Legal Notice

You can find the content of this chapter as well in the wiki of fi-ware.

20.1.1.1 *General Information*

FI-WARE Project Partners refers to Parties of the FI-WARE Project in accordance with the terms of the FI-WARE Consortium Agreement.

20.1.1.2 *Use Of Specification - Terms, Conditions & Notices*

The material in this specification details a FI-WARE Generic Enabler Specification (hereinafter "Specification") in accordance with the terms, conditions and notices set forth below. This Specification does not represent a commitment to implement any portion of this Specification in any company's products. The information contained in this Specification is subject to change without notice.

20.1.1.3 *Copyright License*

Subject to all of the terms and conditions below, the copyright holders in this Specification hereby grant you, the individual or legal entity exercising permissions granted by this License, a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense) under its respective copyrights incorporated in the Specification, to copy and modify this Specification and to distribute copies of the modified version, and to use this Specification, to create and distribute special purpose specifications and software that are an implementation of this Specification, and to use, copy, and distribute this Specification as provided under applicable law.

20.1.1.4 *Patent License*

"Specification Essential Patents" shall mean patents and patent applications, which are necessarily infringed by an implementation of the Specification and which are owned by any of the FI-WARE Project Partners. "Necessarily infringed" shall mean that no commercially reasonable alternative exists to avoid infringement.

Each of the FI-WARE Project Partners, jointly or solely, hereby agrees to grant you, on royalty-free and otherwise fair, reasonable and non-discriminatory terms, a personal, nonexclusive, non-transferable, non-sub-licensable, royalty-free, paid up, worldwide license, under their respective Specification Essential Patents, to make, use sell, offer to sell, and import software implementations utilizing the Specification.

The FI-WARE Project Partners shall not be responsible for identifying patents for which a license may be required by any FI-WARE Specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. FI-WARE specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

20.1.1.5 *General Use Restrictions*

Any unauthorized use of this Specification may violate copyright laws, trademark laws, and communications regulations and statutes. This Specification contains information which is protected by copyright. All Rights Reserved. This Specification shall not be used in any form

or for any other purpose different from those herein authorized, without the permission of the respective copyright owners.

For avoidance of doubt, the rights granted are only those expressly stated in this Section herein. No other rights of any kind are granted by implication, estoppel, waiver or otherwise

20.1.1.6 *Disclaimer Of Warranty*

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE FI-WARE PARTNERS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, <u>WARRANTY OF NON INFRINGEMENT OF THIRD PARTY RIGHTS</u>, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE.

IN NO EVENT SHALL THE FI-WARE PARTNERS BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. The entire risk as to the quality and performance of software developed using this Specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this Specification.

20.1.1.7 *Trademarks*

You shall not use any trademark, marks or trade names (collectively, "Marks") of the FI-WARE Project Partners or the FI-WARE project without prior written consent.

20.1.1.8 *Issue Reporting*

This Specification is subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Procedure described on the web page http://www.fi-ware.eu.