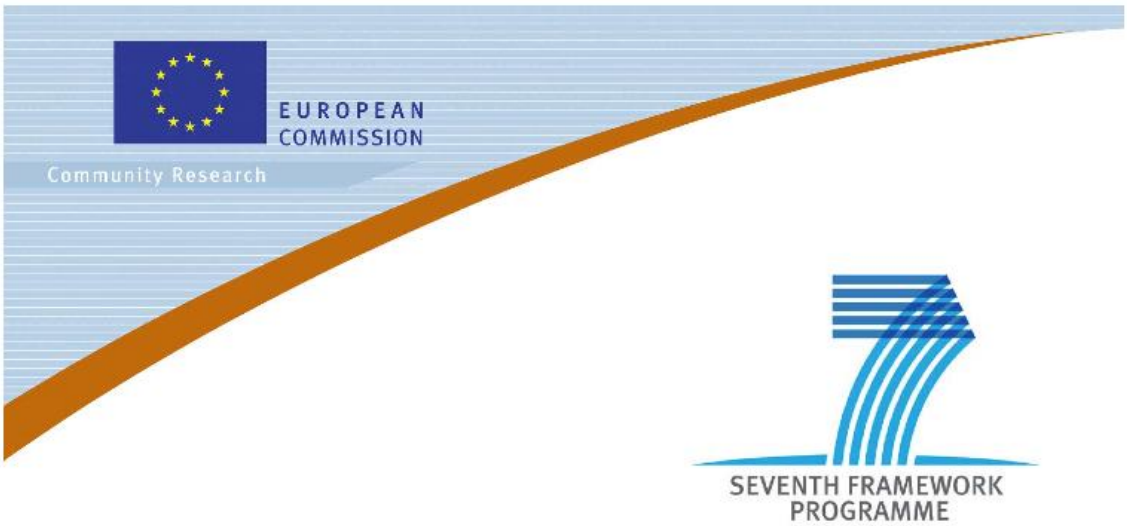


Private Public Partnership Project (PPP)

Large-scale Integrated Project (IP)

D.13.1.3: 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-WP13-D.13.1.3

Project Document Date: 2014-08-06

Deliverable Type and Security: Public

Author: FI-WARE Consortium

Contributors: FI-WARE Consortium

1.1 Executive Summary

This document describes the Generic Enablers in the Advanced Middleware and Web-based User Interfaces chapter, their basic functionality and their interaction. 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, behaviour and intended use of the GE
- Terminology, definitions and abbreviations to clarify the meanings of the specification
- Signature and behaviour 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 Advanced Middleware and Web User Interface (UI) Architecture chapter (A.K.A MiWi) of FI-WARE offers Generic Enablers from two different but related areas:

Advanced Middleware

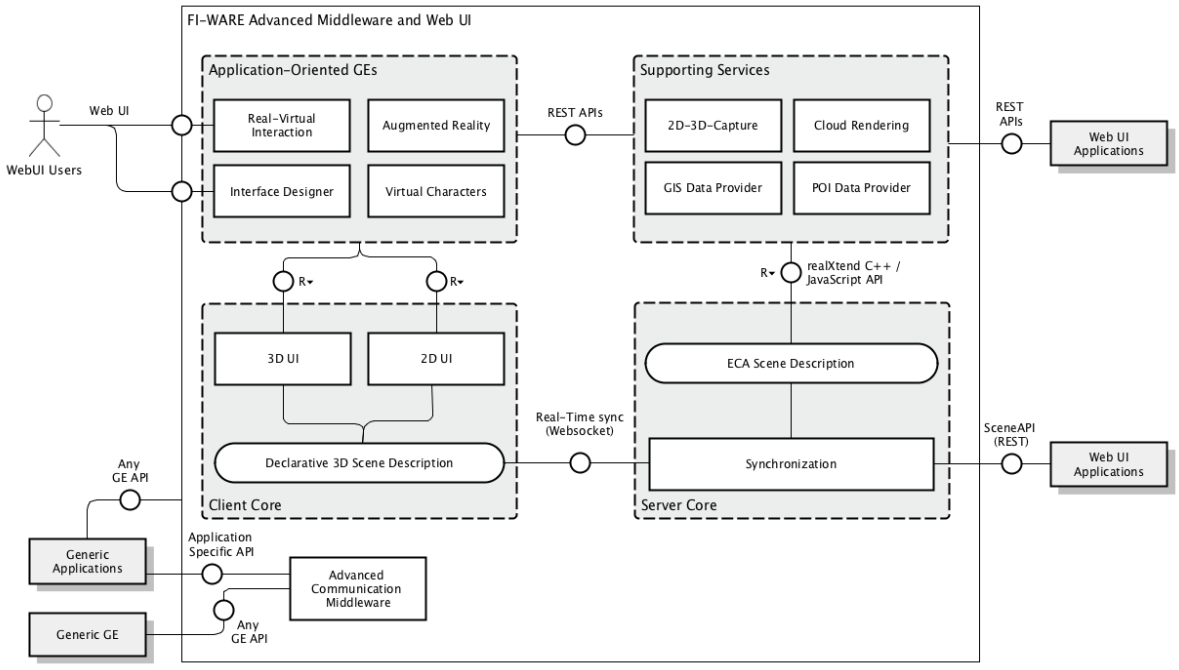
The high-performance middleware that is backward compatible with traditional Web services (e.g. REST) but offers advanced features and performance and dynamically adapts to the communication partners and its environment. A novel API separates WHAT data needs to be communicated from WHERE the data come from within the native data structures of the application, and HOW the data should be transmitted to the target. Additionally, the middleware offers "Security by Design" through a declarative API, where the application defines the security requirements and policies that apply to its data, which are then automatically enforced by the middleware. The middleware uses of the security functionality offered by the Security chapter of FI-WARE.

Advanced Web-based User Interface (Web UI)

In order to become widely visible and adopted by end users, the FI-WARE Future Internet platform must not only offer server functionality but must also offer much improved user experiences. The objective is to significantly improve the user experience for the Future Internet by adding new user input and interaction capabilities, such as interactive 3D graphics, immersive interaction with the real and virtual world (Augmented Reality), virtualizing and thus separating the display from the (mobile) computing device for ubiquitous operations, and many more. The technology is based on the Web technology stack, as the Web is quickly becoming THE user interface technology supported on essentially any (mobile) device while already offering advanced rich media capabilities (e.g. well-formatted text, images, video). First devices are becoming available that use Web technology even as the ONLY user interface technology. The Web design and programming environment is well-known to millions of developers that allow quick uptake of new technology, while offering a proven model for continuous and open innovation and improvement.

These two areas have been combined because highly interactive (2D/3D) user interfaces making use of service oriented architectures have strong latency, bandwidth, and performance requirements regarding the middleware implementations. An example is the synchronization service for real-time shared virtual worlds or the machine control on a factory floor that must use the underlying network and computing hardware as efficiently as possible. Generic Enables are provided to make optimal use of the underlying hardware via Software Defined Networking in the Middleware (SDN, using the GEs of the Interface to Networks and Devices (I2ND) Architecture chapter) and the Hardware Support in the 3D-UI GE (see FIWARE.ArchitectureDescription.MiWi.3D-UI) that provides access to GPU and other parallel compute functionality that may be available.

The following diagram shows the main components (Generic Enablers) that comprise FI-WARE Advanced Middleware and Web-based User Interfaces chapter architecture.



Advanced Middleware and Web UI Architecture Overview

More information about the Advanced Middleware and Web-based UI Chapter and FI-WARE in general can be found within the following pages:

http://wiki.fi-ware.org

Advanced Middleware and Web UI Architecture

Materializing Advanced Middleware and Web User Interfaces 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.org/

The following resources were used to generate this document:

D.13.1.3 FI-WARE GE Open Specifications front page

FIWARE.OpenSpecification.MiWi.Middleware

Middleware Basic Open API Specification

Middleware Advanced Open API Specification

FIWARE.OpenSpecification.MiWi.2D-UI

2D-UI Open API Specification

FIWARE.OpenSpecification.MiWi.3D-UI

XML3D Open API Specification

XFlow Open API Specification

FIWARE.OpenSpecification.MiWi.Synchronization

Synchronization Open API Specification

SceneAPI RESTful API Specification

FIWARE.OpenSpecification.MiWi.CloudRendering

Cloud Rendering Open API Specification

FIWARE.OpenSpecification.MiWi.GISDataProvider

GIS Data Provider Open API Specification

FIWARE.OpenSpecification.MiWi.POIDataProvider

POI Data Provider Open API Specification

FIWARE.OpenSpecification.MiWi.2D-3D-Capture

2D/3D Capture Open API Specification

FIWARE.OpenSpecification.MiWi.AugmentedReality

Augmented Reality Open API Specification

FIWARE.OpenSpecification.MiWi.RealVirtualInteraction

Real Virtual Open API Specification

FIWARE.OpenSpecification.MiWi.VirtualCharacters

Virtual Characters Open API Specification

FIWARE.OpenSpecification.MiWi.InterfaceDesigner

Interface Designer API Specification

FI-WARE Open Specification Legal Notice (essential patents license)

FI-WARE Open Specification Legal Notice (implicit patents license)

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: ADMINO, CYBER, UOULU, LUDOCRAFT, PLAYSIGN, ZHAW, EPROS, USAAR-CISPA, DFKI.

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, Middleware, 3D User Interfaces.

1.9 Changes History

Release	Major changes description	Date	Editor
V1	First draft of deliverable	2014-03-19	ZHAW
V2	Second draft of deliverable ready for review	2014-03-25	ZHAW
V3	Version for delivery	2014-05-28	ZHAW
V4	Version for delivery	2014-06-09	ZHAW
V5	Definite version for delivery after incorporating several formal changes	2014-08-06	ZHAW

1.10 Table of Content

2 FIWARE OpenSpecification MiWi Middleware

Name	FIWARE.OpenSpecification.MiWi.Middleware
Chapter	Advanced Middleware and Web-based UI
Catalogue-Link to Implementation	Middleware KIARA
Owner	EPROS, ZHAW, DFKI, USAAR-CISPA, Christof Marti

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.org and similar pages in order to understand the complete context of the FI-WARE project.

2.2 Copyright

- Copyright © 2013 by eProsima, ZHAW, DFKI, USAAR-CISPA

2.3 Legal Notice

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

2.4 Overview

This specification describes the Advanced Middleware GE, which enables flexible, efficient, scalable, and secure communication between distributed applications and to/between FI-WARE GEs.

Middleware in general provides a software layer between the application and the communication network and allows application to abstract from the intricacies of how to send a piece of data to a service offered by a another application and possibly return results. The middleware offers functionality to find and establish a connection to a service, negotiate the best wire and transport protocols, access the applications native data structures and encode the necessary data in a format suitable for the chosen protocol, and finally send that data and possibly receive results in return. In a similar way an application can use the middleware also to offer services to other applications by registering suitable service functionality and interfaces, which can then be used as targets of communication.

In contrast to other GEs, the Advanced Middleware GE is not a standalone service running in the network, but a set of compile-/runtime tools and a communication library to be integrated with the application.

The Advanced Middleware (AMi) architecture presented here offers a number of key advantages over other available middleware implementations:

- **High-Level Service Architecture:** AMi offers applications a high-level architecture that can shield them from the complexities and dangers of network programming. When applications declare services and data structures they can annotate them with the QoS, security, and other requirements while AMi automatically implement them. Thus applications can exclusively focus on the application functionality.

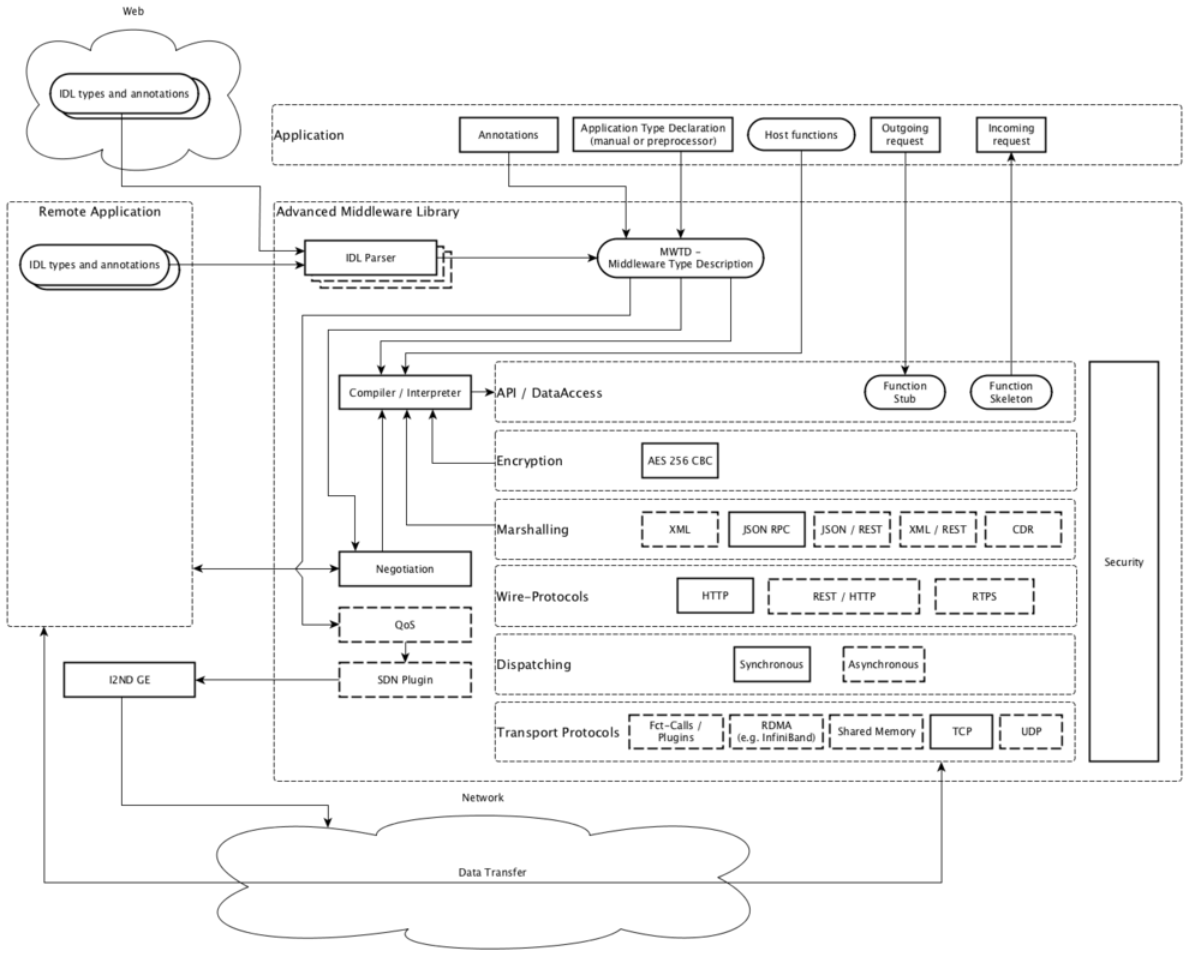
- **Security:** The network is the main security threat to most applications today but existing middleware has offered only limited security functionality that has often been added as an afterthought and requires the application developer (who are often not security experts) to configure the security functionality. Instead, AMi offers *Security by Design* where security has been a designed into the architecture from the start. Applications can simply declare their security needs in the form of security policies (security rules) and apply them to data structures and service at development time or even later during deployment definitions. AMi then makes sure that these requirements are met before any communication takes place and applies any suitable security measures (e.g. encryption, signatures, etc.) during the communication.

- **High-performance:** The AMi API has been designed as to allow for the highest possible communication performance. Besides the common networking technologies like TCP/IP, this also includes the option of doing Remote DMA (RDMA) directly between in-memory data structures thereby completely eliminate the OS and network stack from the data communication on some modern networks (e.g. Infiniband), the use of shared memory on the same machine, or even the use of direct function calls for services within the same address space (e.g. via plug-ins).

- **Dynamic Multi-Protocol support:** The AMi architecture can select at run-time the best way to communicate with a remote service. Thus, an AMi application can simultaneously talk with legacy services via their predefined protocols (e.g. DDS, REST) while able to take advantage of higher performance functionality when talking to other AMi services. It also supports various communication patterns, like Publish-/Subscribe (PubSub), Point-To-Point, or Request-/Reply (RPC).

- **QoS and Software Defined networking:** Where possible the QoS annotations are also used to configure the network using modern Software Defined networking functionality, e.g. to reserve bandwidth.

The following layer diagram shows the main components of the Advanced Communication Middleware GE.



Advanced Middleware Architecture Overview

In the above diagram the principle communication flow goes from top to bottom for sending data, respectively and from bottom to top for receiving data. As in a typical layer diagram each layer is responsible for specific features and builds on top of the layers below. Some modules are cross cutting and go therefore over several layers (e.g. Security).

Here are some of the highlights of the AMi architecture shown in above diagram:

- AMi clearly separates the definition of WHAT data must be communicated (the communication contract via one of many interface definition languages (IDLs)) from WHERE that data comes from in the application and from HOW that data is transmitted. This *separation of concerns* is critical to support some advanced functionality and be portable to a wide range of services and their communication mechanisms.

- AMi offers a declarative API where the applications declare their native data structures (explicitly or implicitly, depending on the language used). Instead of enforcing its own type definitions onto applications, AMi allows applications to declare their native data types to the middleware which are then used as the sources and targets for sending data. This specifically means that AMi operates on an end to end basis between the memory spaces of two communicating applications

- AMi supports multiple IDLs to define what data needs to be communicated. On establishing the connection the interface definition of a service are obtained (explicitly or implicitly).
- AMi offers annotations for QoS, security, or other features that can be added to the data declared by the application, to the IDL, as well as later during deployment. They are are used by the middleware to automatically implement its functionality by requesting QoS functionality from the network layer or automatically enforcing security measures.
- As the connection to a service is established, both sides choose a common mechanism and protocol (negotiation) to best communicate with each other. Besides the traditional network protocols like TCP/IP, AMi also supports advanced communication mechanisms like RDMA, shared memory, or simply function calls for plugins. AMi has been designed to also support Software Defined Networking in order to configure QOS parameters in the network.
- AMi uses an embedded compiler to combine the information from the IDL and the data declarations and generates the most efficient code to map the native types to the network buffers and back. AMi can generate native code for best performance through JIT compilation, use an internal interpreter, or optionally also generate static stubs for fixed configurations. The generated code combines several of the layers including data access, encryption, marshaling, wire protocol encoding, as well as using some transport protocol for sending the data.
- AMi offers an efficient dispatching mechanism for scheduling incoming request to the correct service implementation.

Below we give a short description of the different layers and components.

2.4.1 API & Data Access

The application accesses the communication middleware using a set of defined function calls provided by the API-layer. They may vary depending on the communication pattern (see below).

The main functionality of the Data Access Layer is to provide the mapping of data types and Function Stubs/Skeletons (request/response pattern) or DataReaders/-Writers (publish/subscribe or point-to-point pattern).

The Advanced Middleware GE provides two variants of this functionality:

- A **basic static compile-time Data-Mapping and generation of Function Stubs/Skeletons or DataReaders/-Writers**, created by a compile time IDL-Parser/Compiler from the remote service description, which is provided in an *Interface Definition Language (IDL)* syntax based on the Object Management Group (OMG) IDL (see below) or, in case of WebService compatibility in *Web Application Definition Language (WADL)* syntax, which is submitted as a W3C draft.
- An **advanced dynamic runtime Data- and Function-Mapping** based on a *declarative description of the internal data-structures and functions* provided by the application and the IDL description of the remote service with an embedded **Runtime Compiler/Interpreter**

Quality of Service (QoS) parameters and Security Policies may be provided through the API and/or IDL-Annotations. This information will be used by the QoS and Security modules to ensure the requested garantees.

Depending on the communication pattern, different communication mechanisms will be used.

- For **publish/subscribe** and **point-to-point** scenarios, the DDS services and operations will be provided. When opening connections, a **DataWriter** for publishers/sender and a **DataReader** for subscribers/receivers will be created, which can be used by the application to send or receive DDS messages.
- For **request/reply** scenarios the **Function Stubs/Skeletons** created at compile- or runtime can be used to send or receive requests/replies.

2.4.2 Marshaling

Depending on configuration, communication pattern and type of end-points the data will be serialized to the required transmission format when sending and deserialized to the application data structures when receiving.

- **Common Data Representation (CDR)** an OMG specification used for all DDS/RTPS and high-speed communication.
- **Extensible Markup Language (XML)** for WebService compatibility.
- **JavaScript Object Notation (JSON)** for WebService compatibility.

2.4.3 Wire Protocols

Depending on configuration, communication pattern and type of end-points the matching Wire-Protocol will be chosen.

- For **publish/subscribe** and **point-to-point** patterns the **Real Time Publish Subscribe (RTPS)** Protocol is used.
- For **request/reply** pattern with WebService compatibility the **REST/HTTP** Protocol is used.
- For **request/reply** pattern between DDS end-points the **Real Time Publish Subscribe (RTPS)** Protocol is used.
- For for high-performance communication the wire protocol might be skipped entirely and set up directly on lower layer communication mechanisms and protocols.

2.4.4 Dispatching

The dispatching module is supporting various threading models and scheduling mechanisms. The module is providing single-threaded, multi-threaded and thread-pool operation and allows synchronous and asynchronous operation. Priority or time constraint scheduling mechanisms can be specified through QoS parameters.

2.4.5 Transport Mechanisms

Based on the QoS parameters and the runtime-environment the **QoS module** will decide which transport mechanisms and protocols to choose for data transmission.

In Software Defined Networking (SDN) environments, the **SDN plugin** will be used to get additional network information (e.g. from the I2ND GE) or even provision the network to provide the requested quality of service or privacy.

2.4.6 Transport Protocols

All standard transport protocols (TCP, UDP) as well as encrypted tunnels (TLS,DTLS) are supported. For high-performance communication in specific environments optional optimized protocols will be provided (Memory Mapping, Backplane/Fabric,...).

2.4.7 Security

The security module is responsible for authentication of communication partners and will ensure in the whole middleware stack, the requested data security and privacy. The required information can be provided with Security Annotations in the IDL and by providing a security policy via the API.

2.4.8 Negotiation

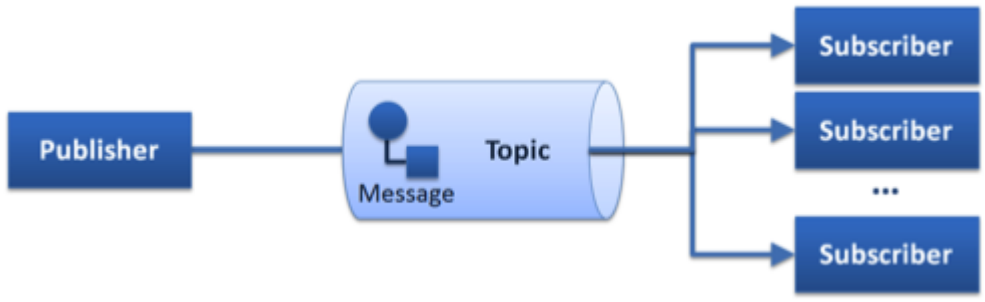
The negotiation module provides mechanisms to discover or negotiate the optimal transmission format and protocols when peers are connecting. It discovers automatically the participants in the distributed system, searching through the different transports available (shared memory and UDP by default, TCP for WebService compatibility) and evaluates the communication paradigms and and the corresponding associated QoS parameters and security policies.

2.5 Basic Concepts

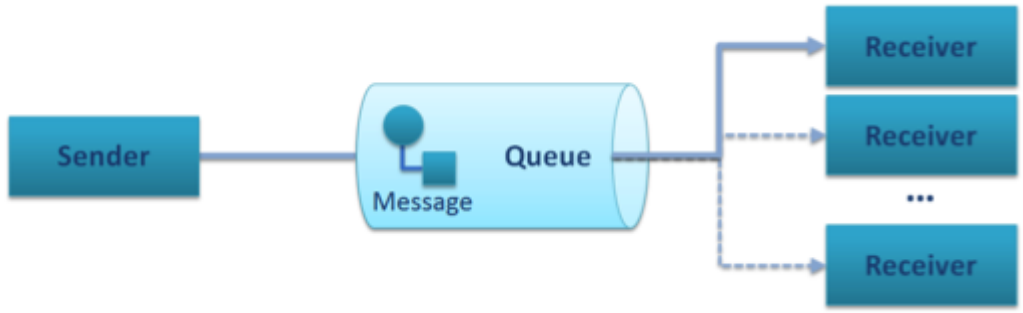
In this section several basic concepts of the Advanced Communication Middleware are explained. We assume that the reader is familiar with the basic functionality of communication middleware like CORBA or WebServices.

2.5.1 Communication Patterns

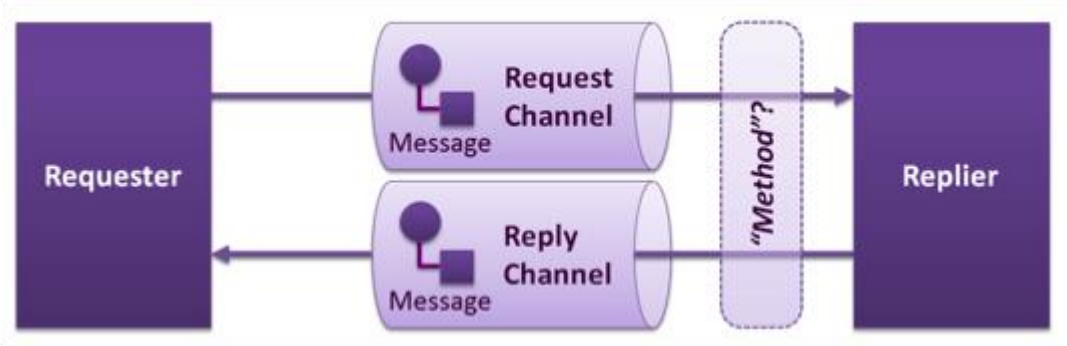
We can distinguish between three main different messaging patterns, Publish/Subscribe, Point-to-Point, and Request/Reply, shown schematically bellow:



Publish/Subscribe Pattern



Point-To-Point Pattern



Request/Reply Pattern

All middleware technologies available implement one or many of these messaging patterns and may incorporate more advanced patterns on top of them. Most of RPC middleware is based on the Request/Reply pattern and more recently, extents towards support of Publish/Subscribe and/or the Point-To-Point pattern.

W3C Web Service standards define a Request/Reply and a Publish/Subscribe pattern which is built on top on that (WS-Notification). CORBA, in a similar way, build its Publish/Subscribe pattern (Notification Service) on top of a Request/Reply infrastructure. In either case the adopted architecture is largely ruled by historical artifacts instead of performance or functional efficiency. The adopted approach is to emulate the Publish/Subscribe pattern on top of the more complex pattern thus inevitably leading to poor performance and complex implementations.

The approach of the Advanced Middleware takes the other direction. It provides native Publish/Subscribe and implements the Request/Reply pattern on top of this infrastructure. Excellent results can be achieved since the Publish/Subscribe is a meta-pattern, in other words a pattern generator for Point-To-Point and Request/Reply and potential alternatives.

2.5.2 Interface Definition Language (IDL)

The Advanced Middleware GE supports a novel IDL to describe the Data Types and Operations. Following is a list of the main features it supports:

- **IDL, Dynamic Types & Application Types**: It support the usual schema of IDL compilation to generate support code for the data types, but also, dynamic runtime type creation, allowing the applications to use its own data structures without forcing to use the IDL compiler generated types. See the data-access feature bellow for a complete description.
- **IDL Grammar**: An OMG-like grammar for the IDL as in DDS, Thrift, ZeroC ICE, CORBA, etc.
- **Types**: Support of simple set of basic types, structs, and various high level types such as lists, sets, and dictionaries (maps).
- **Type inheritance, Extensible Types, Versioning**: Advanced data types, extensions, and inheritance, and other advanced features will be supported.
- **Annotation Language**: The IDL is extended with an annotation language to add properties to the data types and operations. These will, for example, allows adding security policies and QoS requirements.
- **Security**: The IDL allows for annotating operations and data types though the annotation feature of our IDL, allowing setting up security even at the field level.

For compatibility with REST-based WebServices the Middleware also supports the W3C draft submission Application Definition Language (WADL).

2.5.3 Data Access Layer

The Advanced Middleware supports an advanced set of data types:

- **Static Data Types**: Types generated via the IDL compiler in compliance with traditional approaches to warrant backward compatibility.
- **Dynamic Middleware Data Types**: Data types generated by the middleware at runtime.
- **Application Native Data Types (new technique)**: To use application native data types, where the application provides type marshaling and data management using a declarative and/or procedural approach. To this end the Advanced Middleware GE provides two different mechanisms:
    - Letting the application developer provide his own data type plug-in using calls to low-level routine in the middleware that then perform the required marshaling and other operations. Some basis support for this is already provided by RTI-DDS (and also OpenDDS).
    - Exposing an API to describe the application data type and generating the required marshaling and management operations at run-time by:
    - **Interpretation**: Generating an intermediate byte-code to implement the operations and interpret this byte-code by a small “virtual machine”, or

- **Compilation**: Generating an intermediate representation but compiling this data access code to native code with a JIT compiler (e.g. by an embedded LLVM-based compiler). This includes integrating and optimizing (e.g. inlining) the code for performing the chosen data marshaling and submission to the transport mechanism.

2.6 Main Interactions

As explained above, the middleware can be used in different communication scenarios. Depending on the scenario, the interaction mechanisms and the set of API-functions for application developers may vary.

2.6.1 API versions

There will be two versions of APIs provided:

- **Basic API**
  - Static compile-time parsing of IDL and generation of Stub-/Skeletons and DataReader/DataWriter
  - Compatible to RPC-DDS and DDS applications
- **Advanced API**
  - Dynamic runtime parsing of IDL and generation of Stub-/Skeletons
  - Mapping of application datatypes and functions
  - Advanced security policy and QoS parameters
  - Support for high-performance transport mechanisms and protocols
  - REST Webservice support

2.6.2 Classification of functions

The API-Functions can be classified in the following groups:

- **Preparation:** statically at compile-time (Basic API) or dynamically at run-time (Advanced API)
  - Declare the local applications datatypes/functions (Advanced API only)
  - Parsing the Interface Definition of the remote side (IDL-Parser)
  - Building the data-/function mapping (Advanced API only)
  - Generate Stubs-/Skeletons, DataReader-/Writer (Compiler-/Interpreter)
  - Build your application against the Stubs-/Skeletons, DataReader-/Writer (Basic API only)
- **Initialization:**
  - Set up the environment (global QoS/Transport/Security policy,...)

    - Open connection (provide connection specific parameters: QoS/Transport/Security policy, Authentication, Tunnel encryption, Threading policy,...)

- **Communication**
    - Send Message/Request/Response (sync/async, enforce security)
    - Receive Message/Request/Response (sync/async, enforce security)
    - Exception Handling
- **Shutdown**
    - Close connection (cleanup topics, subscribers, publishers)
    - Free resources

Detailed description of the APIs and tools can be found in the User and Programmers guide, which will be updated for every release of the Advanced Middleware GE.

2.7 Basic Design Principles

Implementations of the Advanced Middleware GE have to comply to the following basic design principles:

- All modules have to provide defined and documented APIs.
- Modules may only be accessed through these documented APIs and not use any internal undocumented functions of other modules.
- Modules in the above layer model may only depend on APIs of lower level modules and never access APIs of higher level modules.
- All information required by lower level modules has to be provided by the higher levels modules through the API or from a common configuration.
- If a module provides variants of internal functionalities (e.g. Protocols, Authentication Mechanisms, ...) these should be encapsulated as Plugins with a defined interface.

2.8 Detailed Specifications

Following is a list of Open Specifications linked to this Generic Enabler. Specifications labelled as "PRELIMINARY" are considered stable but subject to minor changes derived from lessons learned during last interactions of the development of a first reference implementation planned for the current Major Release of FI-WARE. Specifications labelled as "DRAFT" are planned for future Major Releases of FI-WARE but they are provided for the sake of future users.

2.8.1 Open API Specifications

- Middleware Basic Open API Specification

- Middleware Advanced Open API Specification

2.9 Re-utilised Technologies/Specifications

The Advanced Middleware GE is a set of communication libraries and tools to be delivered with applications/services. It is not a RESTful service running as a standalone component, but in the final advanced version it however can be used to provide or consume RESTful web services.

The technologies and specifications used in basic version of this GE are:

- DDS - Data Distribution Services (OMG Standard V1.2)
- RPC-DDS – RPC over DDS (OMG proposed Standard)
- RTPS - Realtime Publish Subscribe Wire Protocol (OMG Standard V2.1)

The Advanced Version will use and support additional technologies:

- RESTful web services
- HTTP/1.1 (RFC2616)
- JSON and XML data serialization formats.

2.10 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

Annotations

Annotations refer to non-functional descriptions that are added to declaration of native types, to IDL interface definition, or through global annotations at deployment time. The can be used to express security requirements (e.g. "this string is a password and should be handled according the security policy defined for password"), QoS parameters (e.g. max. latency), or others.

AR

AR → Augmented Reality

Augmented Reality (AR)

Augmented Reality (AR) refers to the real-time enhancement of images of the real world with additional information. This can reach from the rough placement of 2D labels in the image to the perfectly registered display of virtual objects in a scene that are photo-realistically rendered in the context of the real scene (e.g. with respect to lighting and camera noise).

IDL

IDL → Interface Definition Language

Interface Definition Language

Interface Definition Language refers to the specification of interfaces or services. They contain the description of types and function interfaces that use these types for input and output parameters as well as return types. Different types of IDL are being used including CORBA IDL, Thrift IDL, Web Service Description Language (WSDL, for Web Services using SOAP), Web Application Description Language (WADL, for RESTful services), and others.

Middleware

Middleware is a software library that (ideally) handles all network related functionality for an application. This includes the setup of connection between peers, transformation of user data into a common network format and back, handling of security and QoS requirements.

PoI

PoI → Point of Interest

Point of Interest (PoI)

Point of Interest refers to the description of a certain point or 2D/3D region in space. It defines its location, attaches meta data to it, and defines a coordinate system relative to which additional coordinate systems, AR marker, or 3D objects can be placed.

Quality of Service (QoS)

Quality of Service refers to property of a communication channel that are non-functional, such a robustness, guaranteed bandwidth, maximum latency, jitter, and many more.

Real-Virtual Interaction

Real-Virtual Interaction refers to Augmented Reality setup that additionally allow users to interact with real-world objects through virtual proxies in the scene that monitor and visualize the state in the real-world and that can use services to change the state of the real world (e.g. switch lights on an off via a virtual button the 3D scene).

Scene

A *Scene* refers to a collection of objects, which are be identified by type (eg. a 3D mesh object, a physics simulation rigid body, or a script object.) These objects contain typed and named data values (composed of basic types such as integers, floating point numbers and strings) which are referred to as attributes. Scene objects can form a hierarchic (parent-child) structure. A HTML DOM document is one way to represent and store a scene.

Security

Security is a property of an IT system that ensures confidentiality, integrity, and availability of data within the system or during communication over networks. In the context of middleware it refers to the ability of the middleware to guarantee such properties for the communication channel according to suitably expressed requirements needed and guarantees offer by an application.

Security Policy

Security Policy refers to rules that need to be fulfilled before a network connection is established or for data to be transferred. It can for example express statements about the identity of communication partners, properties assigned to them, the confidentiality measures to be applied to data elements of a communication channel, and others.

Synchronization

Synchronization is the act of transmitting over a network protocol the changes in a scene to participants so that they share a common, real-time perception of the scene. This is crucial to eg. implementing multi-user virtual worlds.

Type Description

Type Description in the context of the AMi middleware refers to the internal description of native data types or the interfaces described by an IDL. It contains data such as the name of a veriable, its data type, the hierarchical relations between types (e.g. structs and arrays), its memory offset and alignment within another data type, and others. Type Description are used to generate the mapping of native data types to the data that needs to be transmitted by the middleware.

Virtual Character

Virtual Character is a 3D object, typically composed of triangle mesh geometry, that can be moved and animated and can represent an user's presence (avatar) in a virtual world. Typically supported forms of animation include skeletal animation (where a hierarchy of "bones" or "joints" controls the deformation of the triangle mesh object) and vertex morph animation, where the vertices of the triangle mesh are directly manipulated. Virtual character systems may support composing the character from several mesh parts, for example separate upper body, lower body and head parts, to allow better customization possibilities.

3 Middleware Basic Open API Specification

3.1 Introduction

FI-WARE Middleware GE Basic API, is based on the Data Distribution Service (DDS) specifications, an OMG Standard defining the API and Protocol for high performance publish-subscribe middleware, and eProsima RPC over DDS, an Remote Procedure Call framework using DDS as the transport and based on the ongoing OMG RPC over DDS standard. For the data serialization the Common Data Serialization (CDR) is used, another standard from the OMG.

Introduction articles for these technologies are provided in the following links:

- Introduction to DDS
- Introduction to RPC over DDS

3.1.1 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 use DDS and RPC for DDS (Doxygen API documentation). For the latter, this specification provides a full specification of how to comply with the corresponding OMG Standards.

3.2 Middleware GE Basic Open API Specifications

The Middleware Basic API are based on the Open Specifications of the OMG APIs and underlying protocols:

- OMG DDS Specification (API)
- OMG DDS Set of Specs
- OMG RPC over DDS Specification (in RFP Phase)
- CDR Specification (see page 4 of the PDF)

Note: OMG RPC over DDS Standard is a work in progress and several companies have submitted their proposals, including eProsima, one of the members of the middleware GE. The API of eProsima RPC over DDS is aligned with the eProsima proposed standard for RPC over DDS.

3.2.1 RPC over REST

The middleware GE is not a RESTful service, but a set of tools and libraries to interchange data between the different nodes of a distributed system. But, one of the FI-WARE middleware requirements is to offer backwards compatibility with Web Services, specifically RESTful Web Services.

RPC over REST enables the creation and the invocation of RESTful Web Services through the common Basic API used in RPC over DDS, and the future RPC over TCP, both part of the FI-WARE middleware suite.

This API ease the integration or migration of existing web services with the FI-WARE Middleware GE. RPC over REST supports WADL (Web Application Definition Language) as the IDL to define RESTful Web Services.

eProsima RPC over REST Reference (Rev 0.3.0) - Doxygen (HTML) (PDF)

3.3 API Doxygen Documentation (C/C++)

The APIs for DDS and RPC over DDS are offered in many programming languages depending on the implementation. Middleware Basic API is defined C/C++ at this point and it will support other languages in the near future.

The corresponding Doxygen documentation of the APIs is available in the following links:

3.3.1 RPC over DDS

eProsima RPC over DDS API Reference (Rev 0.3.0) - Doxygen (HTML) (PDF)

3.3.2 DDS

The Middleware Basic API supports two different implementations of DDS, RTI DDS and openDDS:

- RTI DDS API Reference
- OpenDDS API Reference

3.3.3 CDR - FastBuffers

Fast Buffers is an implementation of the CDR serialization format. It generates serialization code from an IDL file. The API of the generated code is described in the following link:

eProsima Fast Buffers API Reference (Rev 0.3.0) - Doxygen (HTML) (PDF)

3.3.4 RPC over REST

RPC over REST allows to the creation and the invocation of RESTful Web Services through the common Basic API used in RPC over DDS, but using REST as the transport. The RPC over REST API is described in the following link:

eProsima RPC over REST Reference (Rev 0.3.0) - Doxygen (HTML) (PDF)

4 Middleware Advanced Open API Specification

4.1 Introduction

The Middleware GE framework realizes a novel approach for connecting application code with a framework. The current version supports remote procedure calls (RPC) but support for publish/subscribe communication is planned as well.

Instead of forcing an application to use data types predefined by the middleware (as e.g. in Thrift, CORBA, etc.) applications describes its own native data types to the Middleware GE framework. The middleware then generates the necessary code to access those data structures at run-time using an embedded compiler. Due to the negotiation of the optimal protocol for the targeted server, Middleware GE can generate the optimal code to serialize the native data structures for the chosen protocol.

This approach allows to combine Middleware GE with arbitrary applications without rewriting major parts of the application itself or adding redundant copy operations between the native and middleware generated data structures, as required by other middleware.

This specifications describes C and C++ language support of the Middleware GE.

4.1.1.1.1 *Name prefixes*

Because C has no namespace system, all Middleware GE specific functions, macros, constants and commands need a dedicated prefix to distinguish them from the application functions, macros, constants and commands. In this specification the prefix **mw** or **MW** is used for all of the middleware components. The concrete Middleware GE implementations will then have to use their own specific prefix instead of this generic one.

4.2 Supported types and language constructs

Following an overview of the supported types and language constructs.

4.2.1 IDL Types

The interface definition language (IDL) is used to describe interfaces of the services provided at the remote endpoint, where remote endpoint not necessarily means a different host. The service could even exist as a plug-in within the same application. IDL uses data types and structures, but is platform-, architecture-, and application-agnostic.

The Middleware GE IDL types are based on the Thrift types: http://thrift.apache.org/docs/types/. These types are distinct from native application types and used solely to describe abstract service interfaces. For the detail specification of the Middleware GE IDL see chapter Interface Definition Language.

4.2.2 Primitive types

The Middleware GE provides following primitive types:

- boolean: A boolean value (true or false)

- i8: An 8-bit signed integer
- u8: An 8-bit unsigned integer
- i16: A 16-bit signed integer
- u16: A 16-bit unsigned integer
- i32: A 32-bit signed integer
- u32: A 32-bit unsigned integer
- i64: A 64-bit signed integer
- u64: A 64-bit unsigned integer
- float: A 32-bit floating point number
- double: A 64-bit floating point number
- string: A text string encoded using UTF-8 encoding

In the following table the Middleware GE types are compared with types used in other systems:

Middleware GE	CORBA	.NET	Thrift	SOAP	WebIDL	Range
boolean	boolean	Boolean	bool	boolean	boolean	{true, false}
i8		SByte	byte	byte	byte	[-128, 127]
u8	octet	Byte		unsignedByte	octet	[0, 255]
i16	short	Int16	i16	short	short	[-32768, 32767]
u16	unsigned short	UInt16		unsignedShort	unsigned short	[0, 65535]
i32	long	Int32	i32	int	long	[-2147483648, 2147483647]
u32	unsigned long	UInt32		unsignedInt	unsigned long	[0, 4294967295]
i64	long long	Int64	i64	long	long long	[-9223372036854775808, 9223372036854775807]
u64	unsigned long long	UInt64		unsignedLong	unsigned long long	[0, 18446744073709551615]

float	float	Single	float	float	float	http://en.wikipedia.org/wiki/Single_precision
double	double	Double	double	double	double	http://en.wikipedia.org/wiki/Double-precision_floating-point_format
string	string	String	string	string	DOMString	

4.2.3 Structs

The Middleware GE structs define a derived type that represent a collection of members similar to C/C++'s struct. A struct has a set of strongly typed fields, each with a unique name identifier. Fields may have various annotations (numeric field IDs, optional default values, etc.) that are described in the Middleware GE IDL.

4.2.4 Containers

The Middleware GE supports following container types:

- `array< Type, Size >` - fixed length arrays (e.g. `array< i32, 4 >`)
- `array< Type >` - dynamic arrays

Multidimensional arrays can be constructed by combining multiple `array`'s: 4x4 float array - `array< array< float, 4>, 4>`

4.2.5 Exceptions

Exceptions are functionally equivalent to structs and are thrown when service functions fails.

4.2.6 Services

Services are similar to structs, but contain only functions.

4.2.7 Functions

Functions can only be members of services. Each function has return type, list of named arguments, and list of named exceptions. Function, its arguments and exceptions can be annotated. Functions can have `void` type as a return type.

4.2.8 Annotations

Annotations are functionally equivalent to structs but are only used to represent additional information to the IDL.

4.3 Describing Application Data Types, Functions and Services

In order to connect remote services, abstract IDL types described above need to be mapped to the application-specific data types. The Middleware GE API provides a novel method to perform this task in a

non-intrusive way. Instead of forcing to use predefined data types of the middleware (e.g. Thrift, CORBA, etc.) applications describes their own data types. This allows to combine the Middleware GE with arbitrary applications without rewriting major parts of the application itself.

Application data types of C/C++ application are described using macros which create static in memory representation of the data types. Macros are used as they provide a type-safe way of connecting the type descriptions to the data structures they describe such that changes will automatically detected. These macros mostly declare static data structures than are later references when interface functions and services are declared. Macros can be written by hand, which can be somewhat tedious especially for C language, or generated automatically by a preprocessor tool.

4.3.1 Using the preprocessor tool

An `mw-preprocessor` tool should be able to create the applications type descriptions to be used with the Middleware GE framework. Using the application source code and a number of simple annotations that describe which types should be used by the Middleware GE, the tool should create the required macros. These can be added directly into the source code or provided as a separate header file.

In the following sections we describe how these annotations and macros are used to create the application type and function description.

4.3.1.1 *Describing primitive data types*

In our example we will use the following IDL definition:

```
namespace * calc
service calc {
    i32 add(i32 a, i32 b)
    float addf(float a, float b)
    i32 stringToInt32(string s)
    string int32ToString(i32 i)
}
```

For the client it is required to generate functions that will perform a remote call and connect them with the remote methods of the service:

```
mw_declare_func(Calc_Add, int & result_value result, int a, int b)
mw_declare_func(Calc_Add_Float, float & result_value result, float a, float b)
mw_declare_func(Calc_String_To_Int32, int & result_value result, const char *s)
mw_declare_func(Calc_Int32_To_String, char ** result_value result, int i)
```

mw_declare_func(*type_name*, ...) macro declares remote function that can be called by the client. All arguments after the function name are argument types and names to the function. **result_value** macro

can be used between type and name argument for describing arguments that are received as a result of the call (output arguments). Note that **result_value** can only be applied to a pointer or reference type, which can receive a result value.

For the server it is required to generate service functions that will be called upon remote call:

```
mw_declare_service(Calc_Add_Impl, int & result_value result, int a, int b)

mw_declare_service(Calc_Add_Float_Impl, float & result_value result, float a, float b)

mw_declare_service(Calc_String_To_Int32_Impl, int & result_value result, const char *s)

mw_declare_service(Calc_Int32_To_String_Impl, char ** result_value result, int i)
```

mw_declare_service(*type_name*, ...) macro is similar to the **mw_declare_func** and declares service function that can be called by the client. All arguments after the function name are argument types and names to the function.

Running the `mw-preprocessor` tool on the source code with these macros will generate Middleware GE API macros that will describe not just the client and server function but also generates macros for all involved data types.

4.3.1.2 *Describing complex data types*

The above described approach works also for C/C++ structs as long as they contain combination of structs and primitive data types like int, float, etc. More complex data types require additional annotation since the preprocessor cannot extract the necessary application semantics automatically (C/C++ are low-level languages that do not provide all the needed semantics implicitly). Such types can be described by providing custom user functions which will access the internals of a type. We call such types *opaque*.

For example C++ `std::string` can be mapped to IDL `string` object, but in order to do this the Middleware GE need to know how to get and set internal character sequence:

```
int std_string_SetCString(MW_UserType *ustr, const char *cstr)
{
    if (cstr)
        ((std::string*)ustr)->assign(cstr);
    else
        ((std::string*)ustr)->clear();
    return 0;
}


int std_string_GetCString(MW_UserType *ustr, const char **cstr)
```

```
{
    *cstr = ((std::string*)ustr)->c_str();
    return 0;
}


mw_declare_opaque_object(std::string,
                             mw_user_api(SetCString,
std_string_SetCString),
                             mw_user_api(GetCString,
std_string_GetCString))
```

mw_declare_opaque_object(type_name, user_api_entries) macro associates multiple custom user functions with API known to Middleware GE. In the above case it registers `std_string_GetCString` as a getter and `std_string_SetCString` as a setter functions which return and set std::string contents as a C's null-terminated string respectively. `SetCString` and `GetCString` are reserved Middleware GE keywords that describe an API functions that the user implements.

With this additional declaration it is possible to use `std::string` when calling and implementing service methods `stringToInt32` and `int32ToString`:

```
// for the client
mw_declare_func(Calc_StdString_To_Int32,  int  &  result_value  result,
const std::string & s)
mw_declare_func(Calc_Int32_To_StdString,  std::string  &  result_value
result, int i)


// for the server
mw_declare_service(Calc_StdString_To_Int32_Impl,  int  &  result_value
result, const std::string & s)
mw_declare_service(Calc_Int32_To_StdString_Impl,      std::string      &
result_value result, int i)
```

Besides manipulating contents of opaque types the Middleware GE needs to properly allocate and deallocate them:

```
MW_UserType * std_string_Allocate(void)
{
    return (MW_UserType *)new std::string;
}


void std_string_Deallocate(MW_UserType *value)
```

```
{
    delete (std::string*)value;
}


mw_declare_opaque_object(std::string,
                             mw_user_api(SetCString,
std_string_SetCString),
                             mw_user_api(GetCString,
std_string_GetCString),
                             mw_user_api(AllocateType,
std_string_Allocate),
                             mw_user_api(DeallocateType,
std_string_Deallocate))
```

`AllocateType` and `DeallocateType` represent APIs for allocating and deallocating type respectively.

4.3.1.3 *Array members in structure declaration*

Besides opaque types it may be required to describe more complex structure members like C-arrays. They are often defined in C and sometimes in C++ structs as a combination of an integer member representing array size and a pointer member pointing to the first element of the array. For example:

```
typedef struct IntArray
{
    int size;
    int *array;
} IntArray;
```

Such member combination can be described with the preprocessor macro **mw_struct_array_member**:

```
/* Middleware GE declaration of IntArray */
mw_declare_struct(IntArray,
    mw_struct_array_member(array, size))
```

The **mw_declare_struct(type_name, ...)** macro allows to override a type declaration which usually would otherwise be automatically generated.

4.3.2 Static type construction

The preprocessor described above generates source code containing Middleware GE API macros that need to be included in the application part that uses the Middleware GE API for calling services. These macros can be also written manually or maybe even generated by another tool. In this section we will describe these macros in more detail.

4.3.2.1 *Primitive types*

For declaring C-type for Middleware GE it must have a name, for example by defining it with a typedef. The Middleware GE predefines names for all primitive C types (*Remark: Concrete implementations will replace the **MW_** prefix with their own*):

Middleware GE macro	C type
MW_CHAR	char
MW_WCHAR_T	wchar_t
MW_SCHAR	signed char
MW_UCHAR	unsigned char
MW_SHORT	short
MW_USHORT	unsigned short
MW_INT	int
MW_UINT	unsigned int
MW_LONG	long
MW_ULONG	unsigned long
MW_LONGLONG	long long
MW_ULONGLONG	unsigned long long
MW_SIZE_T	size_t
MW_SSIZE_T	ssize_t
MW_VOID	void
MW_FLOAT	float
MW_DOUBLE	double
MW_LONGDOUBLE	long double
MW_CHAR_PTR	char *
MW_VOID_PTR	void *
MW_INT8_T	int8_t
MW_UINT8_T	uint8_t

MW_INT16_T	int16_t
MW_UINT16_T	uint16_t
MW_INT32_T	int32_t
MW_UINT32_T	uint32_t
MW_INT64_T	int64_t
MW_UINT64_T	uint64_t

Note: this list contains all primitive types that can be described with Middleware GE, not all of them can be directly mapped to the IDL types. See Calling remote functions section for supported mappings.

4.3.2.2 *Pointer declaration*

The Middleware GE supports pointer declaration with the **MW_DECL_PTR** macro.

```
/* Note: MW_INT and MW_FLOAT are predefined, float* is not */
typedef float * FloatPtr;

/* int pointer */
MW_DECL_PTR(IntPtr, MW_INT)

/* float pointer */
MW_DECL_PTR(FloatPtr, MW_FLOAT)

/* FloatPtr* */
MW_DECL_PTR(FloatPtrPtr, FloatPtr)
```

MW_DECL_PTR(*ptr_type_name*, *element_type_name*) macro declare a pointer type for the existing named C-type specified as the second argument. Const pointers are declared with **MW_DECL_CONST_PTR** macro:

```
/* const int pointer */
MW_DECL_CONST_PTR(ConstIntPtr, MW_INT)
```

In C++ pointer declaration is not needed.

4.3.2.3 *Structure declaration*

Structures are declared with the **MW_DECL_STRUCT** macro:

```
/* User-defined C struct */
```

```
typedef struct {
    float x;
    float y;
} Vec2f;

/* Middleware GE declaration of Vec2f */

MW_DECL_STRUCT(Vec2f,
  MW_STRUCT_MEMBER(MW_FLOAT, x)
  MW_STRUCT_MEMBER(MW_FLOAT, y)
)

/* User-defined C struct */

typedef struct {
    Vec2f a;
    Vec2f b;
} Linef;

/* Middleware GE declaration of Linef */

MW_DECL_STRUCT(Linef,
  MW_STRUCT_MEMBER(Vec2f, a)
  MW_STRUCT_MEMBER(Vec2f, b)
)
```

MW_DECL_STRUCT(*type_name*, *members*) macro expects two arguments: name of a C struct type and a sequence of **MW_STRUCT_MEMBER** macros. All Middleware GE macros require that passed type names are typedef names, not tag names. **MW_STRUCT_MEMBER(*member_type_name*, *member_name*)** macro expects also two arguments describing struct member: name of a member type and name of a member. Note that sequence is **not** separated by comma. Also it is allowed to omit members in the described structure.

The type name passed to **MW_STRUCT_MEMBER** macro must either be defined previously by one of the Middleware GE declaration macros or be one of predefined for primitive C types.

When using C++ alternative macros **MW_CXX_DECL_STRUCT** and **MW_CXX_STRUCT_MEMBER** can be used instead:

```
/* Middleware GE declaration of Vec2f */

MW_CXX_DECL_STRUCT(Vec2f,
  MW_CXX_STRUCT_MEMBER(x)
  MW_CXX_STRUCT_MEMBER(y)
)

/* Middleware GE declaration of Linef */

MW_CXX_DECL_STRUCT(Linef,
  MW_CXX_STRUCT_MEMBER(a)
  MW_CXX_STRUCT_MEMBER(b)
)
```

Using C++ templates the macro **MW_CXX_DECL_STRUCT** should deduce types of the struct members automatically.

4.3.2.4 *Array members in structure declaration*

Arrays are often defined in C and sometimes in C++ structs as a combination of an integer member representing array size and a pointer member pointing to the first element of the array. For example:

```
typedef struct IntArray
{
    int size;
    int *array;
} IntArray;
```

Such member combination can be described with the Middleware GE macro **MW_STRUCT_ARRAY_MEMBER**:

```
/* Middleware GE declaration of IntArray */

MW_DECL_PTR(IntPtr, MW_INT)

MW_DECL_STRUCT(IntArray,
  MW_STRUCT_ARRAY_MEMBER(IntPtr, array, MW_INT, size)
)
```

MW_STRUCT_ARRAY_MEMBER(*ptr_member_type_name*, *ptr_member_name*, *size_member_type_name*, *size_member_name*) macro expects also four arguments describing array

member: name of a pointer type, name of a pointer member, name of a size type, and name of a size member.

The type name passed to **MW_STRUCT_ARRAY_MEMBER** macro must either be defined previously by one of the Middleware GE declaration macros or be one of predefined for primitive C types.

Arrays are allocated by default with **malloc** and deallocated with **free** C function calls.

When using C++ the alternative macro **MW_CXX_STRUCT_ARRAY_MEMBER** can be used instead:

```
/* Middleware GE declaration of IntArray */


MW_CXX_DECL_STRUCT(IntArray,
  MW_CXX_STRUCT_ARRAY_MEMBER(array, size)
)
```

Using C++ templates the macro **MW_CXX_STRUCT_ARRAY_MEMBER** can deduce types of the struct members automatically.

4.3.2.5 *Function declaration*

Signatures of functions that are generated by Middleware GE are declared with the **MW_DECL_FUNC** macro:

```
/* Middleware GE declaration of a float* type */
MW_DECL_PTR(FloatPtr, MW_FLOAT)


/* Middleware GE declaration of a function with signature:


   typedef int (*Func)(float *result, int a, float b, double c);
 */


MW_DECL_FUNC(Func,
  MW_FUNC_RESULT(FloatPtr, result)
  MW_FUNC_ARG(MW_INT, a)
  MW_FUNC_ARG(MW_FLOAT, b)
  MW_FUNC_ARG(MW_DOUBLE, c)
)
```

MW_DECL_FUNC(*type_name*, *func_args*) macro is similar to **MW_DECL_STRUCT** and expects two arguments: name of a C function type and sequence of `MW_FUNC_ARG` and `MW_FUNC_RESULT` macros. **MW_FUNC_ARG(*type_name*, *member_name*)** macro expects two arguments describing function argument: name of an argument type and a name of an argument. **MW_FUNC_RESULT(*type_name*,**

***member_name*)** does the same as **MW_FUNC_ARG**, but additionally marks the function parameter as a result of a function call. The return type of the declared function signature is always int and is used to report errors. Note that sequence is **not** separated by comma. The type name passed to `MW_FUNC_ARG` and `MW_FUNC_RESULT` macros must either be defined previously by one of the Middleware GE declaration macros or be one of the predefined for primitive C types (see above).

Similar to structs functions can be declared in C++ with **MW_CXX_DECL_FUNC**, **MW_CXX_FUNC_ARG**, and **MW_CXX_FUNC_RESULT**:

```
/* Middleware GE declaration of Func */

MW_CXX_DECL_FUNC(Func,
  MW_CXX_FUNC_RESULT(float *, result)
  MW_CXX_FUNC_ARG(int, a)
  MW_FUNC_ARG(float, b)
  MW_FUNC_ARG(double, c)
)
```

In C++ derived types of function parameters like pointers and references do not need to be explicitly declared.

4.3.2.6 *Array declaration*

Arrays are declared with **MW_DECL_ARRAY**, **MW_DECL_FIXED_ARRAY**, and **MW_DECL_FIXED_ARRAY_2D** macros.

```
typedef int IntArray[];
typedef float FloatArray4[4];
typedef double DoubleMatrix4[4][4];

/* Unbounded array declaration */

MW_DECL_ARRAY(IntArray, MW_INT)

/* Fixed size 1D array declaration */

MW_DECL_FIXED_ARRAY(FloatArray4, MW_FLOAT, 4)

/* Fixed size 2D array declaration */

MW_DECL_FIXED_ARRAY_2D(mat44, MW_FLOAT, 4, 4)
```

MW_DECL_ARRAY(*array_type_name*, *element_type_name*) macro declares one-dimensional unbounded arrays. **MW_DECL_FIXED_ARRAY(*array_type_name*, *element_type_name*, *size*)** macro declares one-dimensional arrays with fixed length. **MW_DECL_FIXED_ARRAY_2D(*array_type_name*, *element_type_name*, *num_rows*, *num_cols*)** macro declares two-dimensional arrays with fixed length.

All macros expect the name of the declared type as the first argument, previously declared name of the array element as the second, and finally dimension values.

In C++ array types do not need to be explicitly declared.

4.3.2.7 *Forward declaration*

The Middleware GE supports forward declaration of types which is required for declaring cyclic structures. The Middleware GE type is forward declared using the **MW_FORWARD_DECL** macro.

```
typedef struct IntList {
    struct IntList *next;
    int data;
} IntList;

/* Forward declaration of a named type */
MW_FORWARD_DECL(IntList)

/* Pointer declaration */
MW_DECL_PTR(IntListPtr, IntList)

MW_DECL_STRUCT(IntList,
  MW_STRUCT_MEMBER(IntListPtr, next)
  MW_STRUCT_MEMBER(MW_INT, data)
)
```

MW_FORWARD_DECL(*type_name*) macro performs a forward declaration of a specified type.

In C++ forward declaration is not needed.

4.3.2.8 *Opaque types*

Often data structures are part of the implementation and are not defined in a public API. Such data structures are called abstract data types (ADT). In C and C++ such data structures are represented by a declaration of a struct type without public definition. Such struct types are called opaque, and the only way to access their data is to use a public API provided along with the opaque type.

4.3.2.8.1 *Data access*

In the Middleware GE opaque type are defined with **MW_DECL_OPAQUE_TYPE** macro. In order to access its contents the Middleware GE needs to know how to get and set its internal data. The access is performed by getter and setter API functions which signature is known to the Middleware GE. These functions are registered with the opaque type by using the **MW_USER_API** macro. The Middleware GE recursively processes an opaque type by calling API function until it reaches a primitive type supported by default.

The following example shows how the `kr_dstring_t` abstract data type implements dynamic strings in C:

```
int dstring_SetCString(MW_UserType *ustr, const char *cstr)
{
    int result = kr_dstring_assign_str((kr_dstring_t*)ustr, cstr);
    return result ? 0 : 1;
}

int dstring_GetCString(MW_UserType *ustr, const char **cstr)
{
    *cstr = kr_dstring_str((kr_dstring_t*)ustr);
    return 0;
}

MW_DECL_OPAQUE_TYPE(kr_dstring_t,
       MW_USER_API(SetCString, dstring_SetCString)
       MW_USER_API(GetCString, dstring_GetCString))
```

In the above example contents of a `kr_dstring_t` type can be returned as a C-string and set to a C-string. Thus it implements and registers two API access functions named `SetCString` and `GetCString` for setting and getting C-strings from a `kr_dstring_t` type respectively.

With C++ API it is possible to describe the `std::string` type in a similar way:

```
static int std_string_SetCString(MW_UserType *ustr, const char *cstr)
{
    if (cstr)
        ((std::string*)ustr)->assign(cstr);
    else
        ((std::string*)ustr)->clear();
    return 0;
```

```
}

static int std_string_GetCString(MW_UserType *ustr, const char **cstr)
{
    *cstr = ((std::string*)ustr)->c_str();
    return 0;
}

MW_CXX_DECL_OPAQUE_TYPE(std::string,
       MW_CXX_USER_API(SetCString, std_string_SetCString)
       MW_CXX_USER_API(GetCString, std_string_GetCString))
```

4.3.2.9 *Memory management*

Besides accessing contents of an opaque type the Middleware GE needs to know how to allocate and deallocate a type. By default the Middleware GE allocates a type by calling C's `malloc` function. However, when type require additional initialization like in the case with `std::string` and `kr_dstring_t`, we need to register custom allocation and deallocation functions with the opaque type.

```
static MW_UserType * std_string_Allocate(void)
{
    return (MW_UserType *)new std::string;
}

void std_string_Deallocate(MW_UserType *value)
{
    delete (std::string*)value;
}

MW_CXX_DECL_OPAQUE_TYPE(std::string,
       MW_CXX_USER_API(SetCString, std_string_SetCString)
       MW_CXX_USER_API(GetCString, std_string_GetCString)
       MW_CXX_USER_API(AllocateType, std_string_Allocate)
       MW_CXX_USER_API(DeallocateType, std_string_Deallocate))
```

4.3.2.10 *Structs with API*

Not only opaque types, but usual structures may require custom access or allocation behavior. For example when members of a structure require additional initialization after allocation. For this case API functions can be registered for structs:

```
typedef struct Data
{
    int ival;
    kr_dstring_t sval;
} Data;

void initData(Data *data)
{
    data->ival = 0;
    kr_dstring_init(&data->sval);
}

void destroyData(Data *data)
{
    kr_dstring_destroy(&data->sval);
}

MW_UserType * Data_Allocate(void)
{
    Data *data = malloc(sizeof(Data));
    initData(data);
    return (MW_UserType *)data;
}

void Data_Deallocate(MW_UserType *value)
{
    if (value)
    {
        destroyData((Data*)value);
        free(value);
    }
}
```

```
MW_DECL_STRUCT_WITH_API(Data,
  MW_STRUCT_MEMBER(MW_INT, ival)
  MW_STRUCT_MEMBER(kr_dstring_t, sval),
  MW_USER_API(AllocateType, Data_Allocate)
  MW_USER_API(DeallocateType, Data_Deallocate)
)
```

The `Data` structure need to be initialized after allocation and uninitialized after deallocation. Thus it requires custom allocation and deallocation functions.

4.3.2.11 *Service declaration*

Similarly to functions generated by Middleware GE for calling remote services, a remote service function can be declared as well.

```
MW_DECL_SERVICE(Add,
    MW_SERVICE_RESULT(IntPtr, result)
    MW_SERVICE_ARG(MW_INT, a)
    MW_SERVICE_ARG(MW_INT, b))
{
    *result = a + b;
    return MW_SUCCESS;
}
```

MW_DECL_SERVICE(*function_name*, *func_args*) macro declares a function which can be called remotely, similarly to **MW_DECL_FUNC** macro.

4.4 Applications

The following descriptions shows how the Middleware GE will be used by an application.

4.4.1 Initialization and finalization

Before the Middleware GE library can be used it needs to be initialized with the call to the `mwInit` function. After usage all allocated resources should be freed by calling the `mwFinalize` function.

```
int main(int argc, char **argv)
{
    mwInit(&argc, argv);

    mwFinalize();
}
```

4.4.2 Contexts

The Middleware GE require that each thread has a separate context that manages all Middleware GE data structures. Having separate context per thread allows to avoid explicit locking by the user. Context is created with `mwNewContext` function and destroyed with `mwFreeContext` function.

```
int main(int argc, char **argv)
{
    MW_Context *ctx;

    /* Initialize Middleware GE */
    mwInit(&argc, argv);

    /* Create context */
    ctx = mwNewContext();

    /* Destroy context */
    mwFreeContext(ctx);

    /* Finalize Middleware GE */
    mwFinalize();
}
```

Most of Middleware GE API functions require this context as an argument.

4.4.3 Establishing connections

Before Middleware GE can call remote functions a connection to the remote Middleware GE node needs to be established with the call to `mwOpenConnection` function.

```
  MW_Connection *conn;
  MW_Context *ctx;

  /* Create context */
  ctx = mwNewContext();
  /* Open connection to the service */
  conn = mwOpenConnection(ctx, "http://myhost:80/service");
```

The URL used as the argument should refer to the service description which describes resources available at the endpoint of the connection. The service description has following structure:

```
{
    // Description of the server, optional
    info : "test server",

    // absolute or relative URL of the Middleware GE IDL
    idlURL : "/idl/calc.mw",

    // Middleware GE IDL contents as string (either idlURL or
idlContents should be present)
    idlContents : "text",

    // List of servers providing services
    servers : [
        {
            // names of services that served with this configuration,
            // or "*" for all services
            services : "pattern",
            // specification of used marshalling protocol
            protocol : {
                name : "jsonrpc" // name of the protocol
            },
            // specification of used transport protocol
            transport : {
                // name of the transport protocol
                name : "http",
                // URL for URL-based transport
                url : "/rpc/calc"
            }
        },
        ...
    ]
}
```

In the process of establishing a connection the Middleware GE negotiates compatible protocols and transport methods and selects the best one supported by both Middleware GE nodes. Finally descriptions of services available on the remote side are fetched and added to the internal Middleware GE Type Description managed by Middleware GE.

The connection is closed by calling `mwCloseConnection`.

```
  mwCloseConnection(conn);
```

4.4.4 Calling remote functions

In order to call a remote function the Middleware GE generates a client stub function that accepts native application datatypes, serializes them to the format used by the current protocol, performs a call, and deserializes received response. For generating a client stub function **MW_GENERATE_CLIENT_FUNC(*connection*, *idl_method_name*, *func_type_name*, *mapping*)** macro is used. The arguments have the following meaning:

connection - valid MW_Connection handle opened with **mwOpenConnection**.

idl_method_name - name of the remote service method specified in the IDL.

func_type_name - name of the function object type declared with the **MW_FUNC_OBJ(func_type_name)** macro.

mapping - optional mapping of the IDL types to the application types. By default 1:1 mapping by names and types is used. Note: mapping is not implemented yet.

Consider following simple IDL:

```
namespace * calc


service calc {
    i32 add(i32 a, i32 b)
}
```

The IDL method `calc.add` can be mapped to the following function prototype:

```
MW_DECL_FUNC(Calc_Add,
  MW_FUNC_RESULT(IntPtr, result)
  MW_FUNC_ARG(MW_INT, a)
  MW_FUNC_ARG(MW_INT, b))
```

Basic types are mapped accordingly to the following table:

Middleware GE IDL Type	Default native type (C99 types)	Linux 32-bit C-Type
boolean	int32_t	int
i8	int8_t	char
u8	uint8_t	unsigned char
i16	int16_t	short
u16	uint16_t	unsigned short
i32	int32_t	int
u32	uint32_t	unsigned int
i64	int64_t	long long
u64	uint64_t	unsigned long long
float	float	float
double	double	double
string	char *	char *

Native types that are not in the table are mapped by converting them automatically to the required type.

The function instance performing call to the remote side is created in the following way:

```
MW_FUNC_OBJ(Calc_Add) add;


add = MW_GENERATE_CLIENT_FUNC(conn, "calc.add", Calc_Add, "");
```

The macro **MW_FUNC_OBJ(*type_name*)** defines function object type that can store stub instances of a given type generated by the **MW_GENERATE_CLIENT_FUNC** macro. The function object can be executed with the **MW_CALL** macro:

```
int result, errorCode;
```

```
errorCode = MW_CALL(add, &result, 21, 32);
```

Returned value is **MW_SUCCESS** on success and error code otherwise. Error code description can be retrieved by a call to the **mwGetConnectionError** function.

```
if (errorCode != MW_SUCCESS)
  fprintf(stderr, "Error: call failed: %s\n",
mwGetConnectionError(conn));
else
  printf("calc.add: result = %i\n", result);
```

In C++ a less complex syntax for defining and calling functions can be used.

```
MW_CXX_DECL_FUNC(Calc_Add,
  MW_CXX_FUNC_RESULT(int &, result)
  MW_CXX_FUNC_ARG(int, a)
  MW_CXX_FUNC_ARG(int, b))
```

Since C++ provides references it is not required to pass address of a result variable when calling `Calc_Add` function:

```
int result;
Calc_Add add = MW_GENERATE_CLIENT_FUNC(conn, "calc.add", Calc_Add, "");
int errorCode = add(result, 21, 32);
```

4.4.5 Defining remote services

On the server side it is required to define service functions that can be called. First, a service type is defined similarly to the function declaration:

```
MW_DECL_SERVICE(Calc_Add,
    MW_SERVICE_RESULT(IntPtr, result)
    MW_SERVICE_ARG(MW_INT, a)
    MW_SERVICE_ARG(MW_INT, b))
```

This declares only a prototype of the service function. Implementation of the service function must have the same number and type for all arguments. Additionally first argument must be of type **MW_ServiceFuncObj *** and provides information about connection. Result type must be **MW_Result**.

```
MW_Result calc_add_impl(MW_ServiceFuncObj *mw_funcobj, int *result, int
a, int b)
{
    *result = a + b;
    return MW_SUCCESS;
```

```
}
```

When there is only a single implementation of the service function, both declarations can be combined:

```
MW_DECL_SERVICE_IMPL(Calc_Add,
    MW_SERVICE_RESULT(IntPtr, result)
    MW_SERVICE_ARG(MW_INT, a)
    MW_SERVICE_ARG(MW_INT, b))
{
    *result = a + b;
    return MW_SUCCESS;
}
```

When calling service functions the Middleware GE will automatically allocate memory, deserialize input, and serialize output parameters.

After defining service functions we need to create services, load IDL, and register service functions with the IDL methods:

```
MW_Context *ctx;
MW_Connection *conn;
MW_Service *service;
MW_Result result;

/* Create context */
ctx = mwNewContext();

/* Create service */
service = mwNewService(ctx);

/* Load and register IDL with the service */
result = mwLoadServiceIDLFromString(service,
    "MW",
    "namespace * calc "
    "service calc { "
    "    i32 add(i32 a, i32 b) "
    "} ");

/* Register calc.add IDL method with the Calc_Add service function */
```

```
result = MW_REGISTER_SERVICE_FUNC(service, "calc.add", Calc_Add, "",
calc_add_impl);
```

`mwLoadServiceIDLFromString` function parses IDL from string and registers it with the service, alternatively IDL can be loaded from file with `mwLoadServiceIDL` function. The first argument to `mwLoadServiceIDLFromString` is the service handle, then name of the IDL language, and finally string with the IDL.

MW_REGISTER_SERVICE_FUNC macro is similar to **MW_GENERATE_CLIENT_FUNC** macro, but just registers service function with the IDL method, specified by its full name. Internally Middleware GE will generate code that deserializes arguments, calls registered function, and finally serializes result and send it back to the caller. When combined declaration is used **MW_REGISTER_SERVICE_IMPL** needs to be used instead of **MW_REGISTER_SERVICE_FUNC**.

Finally it is required to create server waiting for incoming connections, add all services that server should provide and run it.

```
server = mwNewServer(ctx, "0.0.0.0", "8080", "/service");

mwAddService(server, "/rpc/calc", "jsonrpc", service);

mwRunServer(server);

mwFreeServer(server);

mwFreeService(service);
```

`mwNewServer` function accepts host and port where connections will be accepted. Additionally an URL path with the location of the server configuration document is passed.

After server is created arbitrary number of services that should be served can be added with `mwAddService` function. Finally server is started with the `mwRunServer` function.

4.5 Examples

4.5.1 IDL

The following IDL is used for the examples:

```
namespace * aostest

struct Vec3f {
 float x,
 float y,
```

```
 float z
}

struct Quatf {
 float r,
 Vec3f v
}

struct Location {
 Vec3f position,
 Quatf rotation
}

struct LocationList {
 array<Location> locations
}

service aostest {
  void setLocations(LocationList locations);
  LocationList getLocations();
}
```

4.5.2 Common Source Code

aostest_types.h - common data structures independent of Middleware GE framework

```
/*
 * This file contains application code and data structures independent
of the Middleware GE framework
 */

#ifndef AOSTEST_TYPES_H_INCLUDED
#define AOSTEST_TYPES_H_INCLUDED

#include <MW/mw.h>
#include <string.h>
```

```
#ifdef __cplusplus
extern "C" {
#endif

/* Location */

typedef struct Vec3f {
    float x;
    float y;
    float z;
} Vec3f;

typedef struct Quatf {
    float r; /* real part */
    Vec3f v; /* imaginary vector */
} Quatf;

typedef struct Location {
    Vec3f position;
    Quatf rotation;
} Location;

/* Data */

typedef struct LocationList
{
    int num_locations;
    Location *locations;
} LocationList;

static void initLocationList(LocationList *loclist, size_t size)
{
    loclist->num_locations = size;
```

```
    if (size == 0)
    {
        loclist->locations = NULL;
    }
    else
    {
        loclist->locations = malloc(sizeof(loclist->locations[0])*size);
    }
}

static void destroyLocationList(LocationList *loclist)
{
    loclist->num_locations = 0;
    free(loclist->locations);
    loclist->locations = NULL;
}

static void copyLocationList(LocationList *dest, const LocationList *src)
{
    if (dest->num_locations != src->num_locations)
    {
        destroyLocationList(dest);
        initLocationList(dest, src->num_locations);
    }
    memcpy(dest->locations, src->locations, sizeof(src->locations[0]) * src->num_locations);
}

static MW_UserType * LocationList_Allocate(void)
{
    LocationList *loclist = malloc(sizeof(LocationList));
    initLocationList(loclist, 0);
    return (MW_UserType *)loclist;
```

```
}

static void LocationList_Deallocate(MW_UserType *value)
{
    if (value)
    {
        destroyLocationList((LocationList*)value);
        free(value);
    }
}

#ifdef __cplusplus
}
#endif

#endif
```

4.5.3 Client example

aostest.c - client implementation of aostest (Array-Of-Structures Test) example

```
/*
 * This file contains client implementation of aostest (Array-Of-Structures Test) example
 * implemented in C with Middleware GE framework.
 */

#include <MW/mw.h>
#include <MW/mw_macros.h>
#include <MW/mw_pp_annotation.h>

#include "aostest_types.h"
#include "aostest_mw_client.h"

#include <stdio.h>
#include <string.h>
```

```
#include "c99fmt.h"


/*
 *  Declare application structures and client functions of the
application.
 * Following macros are processed by mw-preprocessor tool and result
 * is output to the aostest_mw_client.h file.
 */


/* mw_declare_struct_with_api(type_name, ...)
 *
 * Declares non-trivial structure type type_name, which require custom
behavior via
 * user specified API functions.
 *
 * mw_struct_array_member(ptr_member_name, size_member_name)
 *
 * Declares member in a structure that represents a C-array composed
from
 * pointer to the array field and integer array size field.
 *
 * mw_user_api(api_name, user_function_name)
 *
 *  Registers user-defined API function user_function_name with a
predefined Middleware GE API api_name
 *
 * In our example we need to declare LocationList structure explicitly
because it
 * contains C-array composed from locations pointer and num_locations
integer and
 *  because LocationList requires custom allocation and deallocation
functions.
 * Both can't be deduced automatically from the source code.
 */
mw_declare_struct_with_api(LocationList,
    mw_struct_array_member(locations, num_locations),
```

```
    mw_user_api(AllocateType, LocationList_Allocate),
    mw_user_api(DeallocateType, LocationList_Deallocate))

/* mw_declare_func(func_name, ...)
 *
 * Declares remote function that can be called by the client.
 * All arguments after the function name are argument types and names
 * to the function.
 *
 * Usually types used as arguments do not need to be explicitly declared.
 * Only when types require custom handling an explicit declaration is required.
 */
mw_declare_func(AOSTest_SetLocations, const LocationList * locations)
mw_declare_func(AOSTest_GetLocations, LocationList * result_value locations)

/*
 * Middleware GE context and connection variables.
 *
 * MW_Context is used for all Middleware GE operations issued from the single thread.
 * Each separate thread require a separate MW_Context instance.
 *
 * MW_Connection is a handle to the remote endpoint
 * over which remote calls are performed.
 */
MW_Context *ctx;
MW_Connection *conn;

/*
 * set_locations and get_locations are handles to the function objects
 * that perform remote call.
 * They are dynamically generated by the MW_GENERATE_CLIENT_FUNC macro.
```

```
 */
MW_FUNC_OBJ(AOSTest_SetLocations) set_locations;
MW_FUNC_OBJ(AOSTest_GetLocations) get_locations;

/*
 * Initialization of the connection
 */
void initConn(const char *url)
{
    /* Create new context */

    ctx = mwNewContext();

    /* Open connection to the service */

    printf("Opening connection to %s...\n", url);
    conn = mwOpenConnection(ctx, url);

    if (!conn)
    {
        fprintf(stderr, "Error: Could not open connection : %s\n",
mwGetContextError(ctx));
        exit(1);
    }

    /*
     *       MW_GENERATE_CLIENT_FUNC(connection,       idl_method_name,
func_type_name, mapping)
     *
     * Generates function that will perform a remote call.
     *
     * connection       - opened and valid MW_Connection handle.
     * idl_method_name   - name of the remote service method specified
in the IDL.
     * func_type_name   - name of the function object type declared
```

```
     *                    with the MW_FUNC_OBJ(func_type_name) macro.
     * mapping           - mapping of the IDL types to the application
types.
     *                     By default 1:1 mapping by names and types is
used.
     *                    Note: mapping is not implemented yet.
     *
     *  Note:  The  IDL  of  the  server  application  is  embedded  in
aostest_server.c.
     */


    set_locations            =            MW_GENERATE_CLIENT_FUNC(conn,
"aostest.setLocations", AOSTest_SetLocations, "");
    if (!set_locations)
        fprintf(stderr,   "Error:   code   generation   failed: %s\n",
mwGetConnectionError(conn));


    get_locations            =            MW_GENERATE_CLIENT_FUNC(conn,
"aostest.getLocations", AOSTest_GetLocations, "");
    if (!get_locations)
        fprintf(stderr,   "Error:   code   generation   failed: %s\n",
mwGetConnectionError(conn));
}


/*
 * Close connection and finalize Middleware GE framework
 */
void finalizeConn()
{
    mwCloseConnection(conn);
    mwFreeContext(ctx);
    mwFinalize();
}


int main(int argc, char **argv)
{
```

```
    const char *url = NULL;
    int errorCode;

    /* Initialize Middleware GE */
    mwInit(&argc, argv);

    if (argc > 1)
        url = argv[1];
    else
        url = "http://localhost:8080/service";

    /* Initialize connection and generate functions */
    initConn(url);

    /* Call remote functions */

    /* Send 10 locations to the server, where they will be stored */
    {
        size_t num, i;
        LocationList loclist;
        num = 10;
        initLocationList(&loclist, num);
        for (i = 0; i < num; ++i)
        {
            loclist.locations[i].position.x = i;
            loclist.locations[i].position.y = i;
            loclist.locations[i].position.z = i;

            loclist.locations[i].rotation.r = 0.707107f;
            loclist.locations[i].rotation.v.x = 0.0f;
            loclist.locations[i].rotation.v.y = 0.0f;
            loclist.locations[i].rotation.v.z = 0.70710701f;
        }
```

```
        /*
         * MW_CALL(funcobj, ...)
         *
         * Will call remote function via function object generated by
MW_GENERATE_CLIENT_FUNC macro.
         * All arguments after function objects are input/output
arguments to the remote function.
         * MW_CALL returns integer value of type MW_Result that
represent an error code.
         */


        errorCode = MW_CALL(set_locations, &loclist);
        if (errorCode != MW_SUCCESS)
            fprintf(stderr,      "Error:      call      failed: %s\n",
mwGetConnectionError(conn));
        else
            printf("aostest.setLocations: DONE\n");
        destroyLocationList(&loclist);
    }

    /* Receive locations stored on the server, and print them */
    {
        size_t num, i;
        LocationList loclist;

        initLocationList(&loclist, 0);

        errorCode = MW_CALL(get_locations, &loclist);
        if (errorCode != MW_SUCCESS)
            fprintf(stderr,      "Error:      call      failed: %s\n",
mwGetConnectionError(conn));
        else
        {
            printf("aostest.getLocations: LocationList {\n");
            printf("  locations: [\n");
            for (i = 0; i < loclist.num_locations; ++i)
```

```
            {
                printf("    position %f %f %f rotation %f %f %f %f\n",
                    loclist.locations[i].position.x,
                    loclist.locations[i].position.y,
                    loclist.locations[i].position.z,
                    loclist.locations[i].rotation.r,
                    loclist.locations[i].rotation.v.x,
                    loclist.locations[i].rotation.v.y,
                    loclist.locations[i].rotation.v.z);
            }
            printf("  ]\n");
            printf("}\n");
        }

        destroyLocationList(&loclist);
    }

    /* Finalize */

    finalizeConn();

    return 0;
}
```

aostest_mw_client.h - code generated by mw-preprocessor from aostest.c

```
#ifndef MW_PP_0A0ZNAOU2TTQRV0FVWC5_H
#define MW_PP_0A0ZNAOU2TTQRV0FVWC5_H

#include <MW/mw.h>
#include <MW/mw_macros.h>

/* This file was generated by mw-preprocessor tool */

#include "aostest_types.h"
```

```
MW_DECL_STRUCT(Vec3f,
  MW_STRUCT_MEMBER(MW_FLOAT, x)
  MW_STRUCT_MEMBER(MW_FLOAT, y)
  MW_STRUCT_MEMBER(MW_FLOAT, z)
)
MW_DECL_STRUCT(Quatf,
  MW_STRUCT_MEMBER(MW_FLOAT, r)
  MW_STRUCT_MEMBER(Vec3f, v)
)
MW_DECL_STRUCT(Location,
  MW_STRUCT_MEMBER(Vec3f, position)
  MW_STRUCT_MEMBER(Quatf, rotation)
)
MW_DECL_PTR(Location_ptr, Location)
MW_DECL_STRUCT_WITH_API(LocationList,
  MW_STRUCT_ARRAY_MEMBER(Location_ptr,        locations,        MW_INT,
num_locations)
  ,
  MW_USER_API(AllocateType, LocationList_Allocate)
  MW_USER_API(DeallocateType, LocationList_Deallocate)
)
MW_DECL_CONST_PTR(const_LocationList_ptr, LocationList)
MW_DECL_FUNC(AOSTest_SetLocations,
  MW_FUNC_ARG(const_LocationList_ptr, locations)
)
MW_DECL_PTR(LocationList_ptr, LocationList)
MW_DECL_FUNC(AOSTest_GetLocations,
  MW_FUNC_RESULT(LocationList_ptr, locations)
)

#endif
```

4.5.4 Server example

aostest_server.c - Middleware GE server implementation of aostest (Array-Of-Structures Test) example

```
/*
 * This file contains server implementation of aostest (Array-Of-Structures Test) example
 * implemented in C with Middleware GE framework.
 */

#include <MW/mw.h>
#include <MW/mw_macros.h>
#include <MW/mw_pp_annotation.h>

#include "aostest_types.h"
#include "aostest_mw_server.h"

#include <stdio.h>
#include <string.h>
#include "c99fmt.h"

/*
 * Declare application structures and service functions of the application.
 * Following macros are processed by mw-preprocessor tool and result
 * is output to the aostest_mw_server.h file.
 */

/* mw_declare_struct_with_api(type_name, ...)
 *
 * Declares non-trivial structure type type_name, which require custom behavior via
 * user specified API functions.
 *
 * mw_struct_array_member(ptr_member_name, size_member_name)
 *
 * Declares member in a structure that represents a C-array composed from
 * pointer to the array field and integer array size field.
```

```
 *
 * mw_user_api(api_name, user_function_name)
 *
 *  Registers user-defined API function user_function_name with a
predefined Middleware GE API api_name
 *
 * In our example we need to declare LocationList structure explicitly
because it
 * contains C-array composed from locations pointer and num_locations
integer and
 *  because LocationList requires custom allocation and deallocation
functions.
 * Both can't be deduced automatically from the source code.
 */
mw_declare_struct_with_api(LocationList,
    mw_struct_array_member(locations, num_locations),
    mw_user_api(AllocateType, LocationList_Allocate),
    mw_user_api(DeallocateType, LocationList_Deallocate))

/* mw_declare_service(service_name, ...)
 *
 * Declares service function type that can be called by the client.
 * All arguments after the function name are argument types and names
 * to the function.
 *
 *  Usually types used as arguments do not need to be explicitly
declared.
 * Only when types require custom handling an explicit declaration is
required.
 */
mw_declare_service(AOSTest_GetLocationsImpl,       LocationList       *
result_value locations)
mw_declare_service(AOSTest_SetLocationsImpl,   const   LocationList   *
locations)

/** Service implementation */
```

```
/* In objectLocations list are stored locations sent by the client */
LocationList objectLocations = {0, NULL};

/* Receive location list sent by the client and store it to the
objectLocations variable */
MW_Result   aostest_set_locations_impl(MW_ServiceFuncObj   *mw_funcobj,
const LocationList *locations)
{
    size_t i;
    size_t num = locations->num_locations;
    for (i = 0; i < num; ++i)
    {

printf("Location.position %f %f %f\nLocation.rotation %f %f %f %f\n",
                locations->locations[i].position.x,
                locations->locations[i].position.y,
                locations->locations[i].position.z,
                locations->locations[i].rotation.r,
                locations->locations[i].rotation.v.x,
                locations->locations[i].rotation.v.y,
                locations->locations[i].rotation.v.z);
    }

    copyLocationList(&objectLocations, locations);

    return MW_SUCCESS;
}

/* Return location list stored in the objectLocations variable back to
the client */
MW_Result   aostest_get_locations_impl(MW_ServiceFuncObj   *mw_funcobj,
LocationList *locations)
{
    copyLocationList(locations, &objectLocations);

    return MW_SUCCESS;
```

```
}


int main(int argc, char **argv)
{
    /*
     * Middleware GE context and connection variables.
     *
     * MW_Context is used for all Middleware GE operations issued from the single thread.
     * Each separate thread require a separate MW_Context instance.
     *
     *  MW_Service is a handle to the service which provides implementation
     * of service methods specified in the IDL.
     *
     * MW_Server is a handle to the server which can host multiple services.
     */

    MW_Context *ctx;
    MW_Service *service;
    MW_Server *server;
    MW_Result result;
    const char *port = NULL;
    const char *protocol = NULL;

    /* Initialize Middleware GE */
    mwInit(&argc, argv);

    if (argc > 1)
        port = argv[1];
    else
        port = "8080";

    if (argc > 2)
```

```
        protocol = argv[2];
    else
        protocol = "jsonrpc";

    printf("Server port: %s\n", port);
    printf("Protocol: %s\n", protocol);

    /* Create new context */

    ctx = mwNewContext();

    /* Create a new service */

    service = mwNewService(ctx);

    /* Add IDL to the service */

    result = mwLoadServiceIDLFromString(service,
            "MW",
            "namespace * aostest "
            "struct Vec3f {"
            " float x, "
            " float y, "
            " float z "
            "} "
            "struct Quatf {"
            " float r, "
            " Vec3f v  "
            "} "
            "struct Location {"
            " Vec3f position, "
            " Quatf rotation  "
            "} "
            "struct LocationList { "
```

```
            " array<Location> locations "
            "} "
            "service aostest { "
            "  void setLocations(LocationList locations); "
            "  LocationList getLocations(); "
            "} ");
    if (result != MW_SUCCESS)
    {
        fprintf(stderr, "Error: could not parse IDL: %s: %s\n",
                mwGetErrorName(result), mwGetServiceError(service));
        exit(1);
    }

    printf("Register aostest.setLocations ...\n");

    /*
     * MW_REGISTER_SERVICE_FUNC(service, idl_method_name,
     *                                          service_type_name, mapping,
service_func_impl)
     *
     * Registers service function implementation with a specified IDL
service method.
     *
     * service           - valid MW_Service handle.
     * idl_method_name    - name of the remote service method specified
in the IDL.
     * service_type_name - name of the service type declared with the
MW_DECL_SERVICE macro.
     * mapping            - mapping of the IDL types to the application
types.
     *                      By default 1:1 mapping by names and types is
used.
     *                     Note: mapping is not implemented yet.
     *  service_func_impl  -  user  function  that  implements  service
method.
     */
```

```
    result = MW_REGISTER_SERVICE_FUNC(service, "aostest.setLocations",
AOSTest_SetLocationsImpl, "", aostest_set_locations_impl);
    if (result != MW_SUCCESS)
    {
        fprintf(stderr, "Error: registration failed: %s: %s\n",
                mwGetErrorName(result), mwGetServiceError(service));
        exit(1);
    }

    printf("Register aostest.getLocations ...\n");

    result = MW_REGISTER_SERVICE_FUNC(service, "aostest.getLocations",
AOSTest_GetLocationsImpl, "", aostest_get_locations_impl);
    if (result != MW_SUCCESS)
    {
        fprintf(stderr, "Error: registration failed: %s: %s\n",
                mwGetErrorName(result), mwGetServiceError(service));
        exit(1);
    }

    /*
     * Create new server and register service
     */

    server = mwNewServer(ctx, "0.0.0.0", port, "/service");

    mwAddService(server, "/rpc/aostest", protocol, service);

    printf("Starting server...\n");

    /* Run server */

    result = mwRunServer(server);
    if (result != MW_SUCCESS)
```

```
        fprintf(stderr, "Error: could not start server: %s: %s\n",
                mwGetErrorName(result), mwGetServerError(server));

    /* Free everything */

    mwFreeServer(server);
    mwFreeService(service);
    mwFreeContext(ctx);
    mwFinalize();

    /* Free temporary copy of location list */
    destroyLocationList(&objectLocations);

    return 0;
}
```

aostest_mw_server.h - code generated by mw-preprocessor from aostest_server.c

```
#ifndef MW_PP_U759IC9SH7PSLPQOSZB7_H
#define MW_PP_U759IC9SH7PSLPQOSZB7_H

#include <MW/mw.h>
#include <MW/mw_macros.h>

/* This file was generated by mw-preprocessor tool */

#include "aostest_types.h"

MW_DECL_STRUCT(Vec3f,
  MW_STRUCT_MEMBER(MW_FLOAT, x)
  MW_STRUCT_MEMBER(MW_FLOAT, y)
  MW_STRUCT_MEMBER(MW_FLOAT, z)
)
MW_DECL_STRUCT(Quatf,
  MW_STRUCT_MEMBER(MW_FLOAT, r)
```

```
  MW_STRUCT_MEMBER(Vec3f, v)
)
MW_DECL_STRUCT(Location,
  MW_STRUCT_MEMBER(Vec3f, position)
  MW_STRUCT_MEMBER(Quatf, rotation)
)
MW_DECL_PTR(Location_ptr, Location)
MW_DECL_STRUCT_WITH_API(LocationList,
  MW_STRUCT_ARRAY_MEMBER(Location_ptr,        locations,        MW_INT,
num_locations)
  ,
  MW_USER_API(AllocateType, LocationList_Allocate)
  MW_USER_API(DeallocateType, LocationList_Deallocate)
)
MW_DECL_PTR(LocationList_ptr, LocationList)
MW_DECL_SERVICE(AOSTest_GetLocationsImpl,
  MW_SERVICE_RESULT(LocationList_ptr, locations)
)
MW_DECL_CONST_PTR(const_LocationList_ptr, LocationList)
MW_DECL_SERVICE(AOSTest_SetLocationsImpl,
  MW_SERVICE_ARG(const_LocationList_ptr, locations)
)

#endif
```

4.6 Interface Definition Language

Current Middleware GE IDL is based on Thirft syntax described here: http://thrift.apache.org/docs/idl/. All modifications to the original syntax are documented below. Major extension is the support of annotations based on Web IDL syntax: http://www.w3.org/TR/WebIDL/ and generic types. A number of Thrift features will be removed in a future and replaced by annotations.

4.6.1 Document

```
[1]  Document        ::=  Header* Definition*
```

4.6.2 Header

```
[2]  Header          ::=  Include | CppInclude | Namespace
```

4.6.2.1 *Middleware GE Include*

```
[3]  Include         ::=  'include' Literal
```

4.6.2.2 *C++ Include*

```
[4]  CppInclude      ::=  'cpp_include' Literal
```

4.6.2.3 *Namespace*

Thrift uses `namespace` for mapping to different scripting languages. This should be performed by annotations. We plan to make namespace similar to CORBA's `module`, so we can annotate all contents of a single IDL file.

```
[5]  Namespace       ::=  ( 'namespace' ( NamespaceScope Identifier ) |
                                        (           'smalltalk.category'
STIdentifier ) |
                                        ( 'smalltalk.prefix' Identifier
) )

[6]  NamespaceScope  ::=  '*' | 'cpp' | 'java' | 'py' | 'perl' | 'rb' |
'cocoa' | 'csharp'
```

4.6.3 Definition

Added syntax for defining new annotations.

```
[7]  Definition      ::=  Const | Typedef | Enum | Senum | Struct |
Exception | Service | AnnotationDef
```

4.6.3.1 *Const*

```
[8]  Const           ::=  'const' FieldType Identifier '=' ConstValue
ListSeparator?
```

4.6.3.2 *Typedef*

```
```

```
[9]  Typedef          ::=  'typedef' DefinitionType Identifier
```

4.6.3.3 *Enum*

```
[10] Enum                  ::=   'enum' Identifier '{' (Identifier ('='
IntConstant)? ListSeparator?)* '}'
```

4.6.3.4 *Senum*

```
[11]  Senum                    ::=    'senum'  Identifier  '{'  (Literal
ListSeparator?)* '}'
```

4.6.3.5 *Struct*

Added syntax for annotating structs.

```
[12]  Struct                ::=    AnnotationList?  'struct'  Identifier
'xsd_all'? '{' Field* '}'
```

4.6.3.6 *Exception*

Added syntax for annotating exceptions.

```
[13] Exception          ::=  AnnotationList? 'exception' Identifier '{'
Field* '}'
```

4.6.3.7 *Service*

Added syntax for annotating services.

```
[14] Service              ::=   AnnotationList? 'service' Identifier (
'extends' Identifier )? '{' Function* '}'
```

4.6.4 Field

Added syntax for annotating fields.

```
[15] Field            ::=  FieldID? AnnotationList? FieldReq? FieldType
Identifier ('= ConstValue)? XsdFieldOptions ListSeparator?
```

4.6.4.1 *Field ID*

```
[16] FieldID          ::=  IntConstant ':'
```

4.6.4.2 *Field Requiredness*

```
[17] FieldReq         ::=  'required' | 'optional'
```

4.6.4.3 *XSD Options*

TODO This should be removed or replaced by an annotation.

```
[18] XsdFieldOptions ::=  'xsd_optional'? 'xsd_nillable'? XsdAttrs?

[19] XsdAttrs        ::=  'xsd_attrs' '{' Field* '}'
```

4.6.5 Functions

`oneway` attribute was removed from the original Thrift syntax. Use `Oneway` annotation and void return type for representing functions that never return a value. Support for annotating function and its return type was added.

```
[20] Function         ::=  AnnotationList? FunctionType AnnotationList?
Identifier '(' Field* ')' Throws? ListSeparator?

[21] FunctionType     ::=  FieldType | 'void'

[22] Throws           ::=  'throws' '(' Field* ')'
```

The first annotation list annotates function as a whole, e.g.:

```
[Bar(22), Baz] void foobar(i32 i, float f);
```

The annotation list after function return type annotates only return type, e.g.:

```
void [Bar(22), Baz] foobar(i32 i, float f);
```

Each function parameter can be annotated separately, e.g.:

```
void foobar([Bar(22)] i32 i, [Baz] float f);
```

Finally, all these cases can be combined together:

```
[Bar(22), Baz] void [Foo(22), Bar] foobar([Bar(22)] i32 i, [Baz] float
f);
```

4.6.6 Types

Thrift's `bool` type is renamed to `boolean`, `byte` type was renamed to `i8`. Unsigned integer types `u8`, `u16`, `u32`, and `u64` were added. Instead of using predefined `list`, `map` and `set` container types

Middleware GE syntax allows to use arbitrary C++/CORBA like generic types: `list< i32>`, `array< i32, 2>`, `array< i32, array< i32, 4> >`.

```
[23] FieldType       ::=  Identifier | BaseType | GenericType

[24] DefinitionType  ::=  BaseType | GenericType

[25] BaseType          ::=  'boolean' | 'i8' | 'i16' | 'i32' | 'i64' |
'u8' | 'u16' | 'u32' | 'u64' | 'double' | 'string' | 'binary' | 'slist'

[26]  GenericType          ::=    Identifier  '<'  GenericTypeArg  (','
GenericTypeArg)* '>'

[27]  GenericTypeArg   ::=    Identifier  |  BaseType  |  GenericType  |
IntConstant | DoubleConstant | Literal | ConstList | ConstMap
```

4.6.7 Constant Values

```
[28] ConstValue        ::=  IntConstant | DoubleConstant | Literal |
Identifier | ConstList | ConstMap

[29] IntConstant     ::=  ('+' | '-')? Digit+

[30] DoubleConstant  ::=  ('+' | '-')? Digit* ('.' Digit+)? ( ('E' |
'e') IntConstant )?

[31] ConstList       ::=  '[' (ConstValue ListSeparator?)* ']'

[32]  ConstMap               ::=    '{'  (ConstValue  ':'  ConstValue
ListSeparator?)* '}'
```

4.6.8 Basic Definitions

4.6.8.1 *Literal*

```
[33] Literal         ::=  ('"' [^"]* '"') | ("'" [^']* "'")
```

4.6.8.2 *Identifier*

```
[34] Identifier      ::=  ( Letter | '_' ) ( Letter | Digit | '.' | '_' )*

[35] STIdentifier    ::=  ( Letter | '_' ) ( Letter | Digit | '.' | '_' | '-' )*
```

4.6.8.3 *List separator*

```
[36] ListSeparator   ::=  ',' | ';'
```

4.6.8.4 *Letters and Digits*

```
[37] Letter          ::=  ['A'-'Z'] | ['a'-'z']

[38] Digit           ::=  ['0'-'9']
```

4.6.9 Annotations

This part is specific to Middleware GE IDL.

```
[39] AnnotationList  ::= '[' Annotation (',' Annotation)* ']'

[40] Annotation      ::= Identifier AnnotationArgs?

[41] AnnotationArgs  ::=  '('  Identifier  ('='  ConstValue)?  (',' Identifier ('=' ConstValue)? )* ')'
```

User can define new annotations using following syntax:

```
[42] AnnotationDef   ::= AnnotationList? 'annotation' Identifier '{' Field* '}'
```

4.6.10 IDL Examples

```
namespace * calc

exception DivisionByZero {
```

```
}

// service annotation
[HTTPPort(8080)]
service Calculator {

  // function annotation
  [Oneway]
  void ping()

  float add(float a, float b)
  float sub(float a, float b)
  float div(float a, float b) throws (DivisionByZero err)
  float mul(float a, float b)

  array<i32> addArray(array<i32> a, array<i32> b)
}
```

5 FIWARE OpenSpecification MiWi 2D-UI

Name	FIWARE.OpenSpecification.MiWi.3D-UI
Chapter	Advanced Middleware and Web-based UI,
Catalogue-Link to Implementation	2D-UI
Owner	Adminotech Oy, Antti Kokko

5.1 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.org and similar pages in order to understand the complete context of the FI-WARE project.

5.2 Copyright

- Copyright © 2013 by Adminotech Oy

5.3 Legal Notice

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

5.4 Overview

We will be researching the current state of possible input devices in a modern web browser environment. These input devices range from traditional keyboard and mouse to touch, gamepads and sensors. One of the key parts in this GE is to implement a Input API for the 3D client and scripts utilizing it, to register events and get notified when user interaction happens on the 3D scene and its objects. The research will include APIs that may not be included in browser standards yet but can be found in nightly builds or such beta releases.

The second part of this GE, the UI, we apply the same research for modern technologies and example implementation that will make creating and managing common widgets easier for your 3D application. The research will heavily include the use of new/upcoming web standard Web Components.

5.5 Basic Concepts

5.5.1 Input API

Enables registering/unregistering to receive input events fired by e.g. touch pad, gamepad, keyboard and mouse. It also enables registering new input events runtime to extend the functionality. These events must be possible to send to used event system so applications/scripts using the event system can listen the events.

5.5.2 Input Abstraction

Enables registering/unregistering or updating existing input context/state which defines keyboard and mouse conditions that fire an event. E.g. pressing 'w' is forward or pressing 'w' and 'mouse left' is jump. End user is free to decide and name own input states. Within the input state end user can define following bindings or conditions:

- Name
- 0-n - keyboard bindings
- Mouse - left down, middle down, right down
- Time slot - within all conditions must be true
- Priority - in which order the registered input states are handled
- Multiplier - how many times either keyboard or mouse conditions must be repeated within the given time slot.

5.5.3 Web component

Web Components is a new browser technology that is being polyfilled with a library called Polymer. Web Components bring a new way to encapsulate your UI, look and feel and your JavaScript functionality into a single component. These components are easy to include and add to any web page without including scripts and dealing with the usual initialization steps for different libraries. Additionally you are interacting with a DOM element that the Web Component defines, not a JavaScript library directly, the Web Component defined its DOM interface and implements the needed JavaScript behind it.

The component model for the Web aka Web component can be understood as a reusable widget/HTML module on screen.

It consists of 5 pieces:

- Templates - which define chunks of markup that are inert but can be activated for use later.
- Decorators - which apply templates based on CSS selectors to affect rich visual and behavioral changes to documents.
- Custom Elements - which let authors define their own elements, with new tag names and new script interfaces.

- Shadow DOM - which encapsulates a DOM subtree for more reliable composition of user interface elements.
- Imports - which defines how templates, decorators and custom elements are packaged and loaded as a resource.

Each of these pieces is useful individually. When used in combination, Web Components enable Web application authors to define widgets with a level of visual richness and interactivity not possible with CSS alone, and ease of composition and reuse not possible with script libraries today.

http://www.w3.org/TR/2013/WD-components-intro-20130606/.

5.5.4 Shadow DOM

Shadow DOM encapsulates DOM tree from the main document DOM tree. In other words a widget or web component can be encapsulated from the rest of the page and all e.g. javascript library updates or style changes won´t affect the Shadow DOM tree.

https://dvcs.w3.org/hg/webcomponents/raw-file/tip/spec/shadow/index.html.

5.5.5 Polymer

Polymer is a new type of library for the web, built on top of Web Components, and designed to leverage the evolving web platform on modern browsers.

http://www.polymer-project.org/

5.5.6 Gamepad API

Gamepad API enables using JavaScript to read the state of any gamepad controller attached to your computer The Gamepad specification

5.6 Generic Architecture

This GE will be implemented as Javascript library and example Polymer web component.

- InputAPI
    - Keyboard
    - Mouse
    - Touch
    - Gamebad (XBOX, Playstation etc.)
    - Kinect
    - Input abstraction
- Web component

- Polymer

5.7 Main Interactions

Research and implement simple and easy way to use Input API and Web Components in browser environment. InputAPI should be extendable by 3rd party developers. Once new input devices or UI libraries are exposed to the browser environment we should provide a clear way of extending the InputAPI. To start with the InputAPI we will provide modules/components described in Generic Architecture chapter. To start with Web components we will provide example implementation of a web component.

- As a test case for InputAPI there will be a simple html page that includes these libraries and print the user input on the screen.
- As a test case for Input Abstraction there will be a simple html page where user can create own input state/context and see how InputAPI reacts to the created state/context printing the user input on the screen.
- As a test case for Web Component there will be html pages with different implementation to place web component on the screen.

5.8 Basic Design Principles

- Clear Javascript libraries with samples and documentation
- Clear html and Javascript files with documentation

5.9 References

- Touch: http://www.w3.org/TR/touch-events/, Javascript libraries: http://www.queness.com/post/11755/11-multi-touch-and-touch-events-javascript-libraries
- Kinect: http://kineticjs.com/, http://jswipekinetic.codeplex.com/
- Gamepad API: http://www.w3.org/TR/gamepad/, https://dvcs.w3.org/hg/gamepad/raw-file/default/gamepad.html
- WebComponents: http://www.w3.org/TR/2013/WD-components-intro-20130606/, Polymer: http://www.polymer-project.org/, AngularJS: http://angularjs.org/
- HTML5: http://www.w3.org/TR/html5/
- Event handling: http://millermedeiros.github.io/js-signals/

5.10 Detailed Specifications

Following is a list of Open Specifications linked to this Generic Enabler. Specifications labeled as "PRELIMINARY" are considered stable but subject to minor changes derived from lessons learned during last interactions of the development of a first reference implementation planned for the current Major Release of FI-WARE. Specifications labeled as "DRAFT" are planned for future Major Releases of FI-WARE but they are provided for the sake of future users.

5.10.1 Open API Specifications

- 2D-UI Open API Specification

5.11 Re-utilised Technologies/Specifications

- FIWARE.OpenSpecification.MiWi.3D-UI

5.12 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

Annotations

Annotations refer to non-functional descriptions that are added to declaration of native types, to IDL interface definition, or through global annotations at deployment time. The can be used to express security requirements (e.g. "this string is a password and should be handled according the security policy defined for password"), QoS parameters (e.g. max. latency), or others.

AR

AR → Augmented Reality

Augmented Reality (AR)

Augmented Reality (AR) refers to the real-time enhancement of images of the real world with additional information. This can reach from the rough placement of 2D labels in the image to the perfectly registered display of virtual objects in a scene that are photo-realistically rendered in the context of the real scene (e.g. with respect to lighting and camera noise).

IDL

IDL → Interface Definition Language

Interface Definition Language

Interface Definition Language refers to the specification of interfaces or services. They contain the description of types and function interfaces that use these types for input and output parameters as well as return types. Different types of IDL are being used including CORBA IDL, Thrift IDL, Web Service Description Language (WSDL, for Web Services using SOAP), Web Application Description

Language (WADL, for RESTful services), and others.

Middleware

Middleware is a software library that (ideally) handles all network related functionality for an application. This includes the setup of connection between peers, transformation of user data into a common network format and back, handling of security and QoS requirements.

PoI

PoI → Point of Interest

Point of Interest (PoI)

Point of Interest refers to the description of a certain point or 2D/3D region in space. It defines its location, attaches meta data to it, and defines a coordinate system relative to which additional coordinate systems, AR marker, or 3D objects can be placed.

Quality of Service (QoS)

Quality of Service refers to property of a communication channel that are non-functional, such a robustness, guaranteed bandwidth, maximum latency, jitter, and many more.

Real-Virtual Interaction

Real-Virtual Interaction refers to Augmented Reality setup that additionally allow users to interact with real-world objects through virtual proxies in the scene that monitor and visualize the state in the real-world and that can use services to change the state of the real world (e.g. switch lights on an off via a virtual button the 3D scene).

Scene

A *Scene* refers to a collection of objects, which are be identified by type (eg. a 3D mesh object, a physics simulation rigid body, or a script object.) These objects contain typed and named data values (composed of basic types such as integers, floating point numbers and strings) which are referred to as attributes. Scene objects can form a hierarchic (parent-child) structure. A HTML DOM document is one way to represent and store a scene.

Security

Security is a property of an IT system that ensures confidentiality, integrity, and availability of data within the system or during communication over networks. In the context of middleware it refers to the ability of the middleware to guarantee such properties for the communication channel according to suitably expressed requirements needed and guarantees offer by an application.

Security Policy

Security Policy refers to rules that need to be fulfilled before a network connection is established or for data to be transferred. It can for example express statements about the identity of communication partners, properties assigned to them, the confidentiality measures to be applied to data elements of a communication channel, and others.

Synchronization

Synchronization is the act of transmitting over a network protocol the changes in a scene to

participants so that they share a common, real-time perception of the scene. This is crucial to eg. implementing multi-user virtual worlds.

Type Description

Type Description in the context of the AMi middleware refers to the internal description of native data types or the interfaces described by an IDL. It contains data such as the name of a veriable, its data type, the hierarchical relations between types (e.g. structs and arrays), its memory offset and alignment within another data type, and others. Type Description are used to generate the mapping of native data types to the data that needs to be transmitted by the middleware.

Virtual Character

Virtual Character is a 3D object, typically composed of triangle mesh geometry, that can be moved and animated and can represent an user's presence (avatar) in a virtual world. Typically supported forms of animation include skeletal animation (where a hierarchy of "bones" or "joints" controls the deformation of the triangle mesh object) and vertex morph animation, where the vertices of the triangle mesh are directly manipulated. Virtual character systems may support composing the character from several mesh parts, for example separate upper body, lower body and head parts, to allow better customization possibilities.

6 2D-UI Open API Specification

6.1 InputAPI

Javascript library that enables registering/unregistering to receive input events fired by e.g. touch pad, gamepad, keyboard and mouse. API is extendable via plugin system and input abstraction. As default API provides mouse and keyboard event handling. By default plugins for touch pad and gamepad are also supported.

- Instantiate InputAPI

```
var inputAPI = new InputAPI({
	//Give container for event registering. Not mandatory. Register can be done manually.
	container: "#my-container"
});
```

- Create hooks for signals

```
function onTouchEvent (obj, event)
{
	console.log("Touch event "+event);
}

function onKeyEvent(event)
{
	console.log("Keyboard event "+event);
}

function onMouseEvent(event)
{
	console.log("Mouse event "+event);
}

function onGamepadEvent(event)
{
```

```
        console.log("Gamepad event "+event);
}

function onGamepadStatusEvent(event)
{
        console.log("GamepadStatus event "+event);
}
```

- Hook to mouse signals

```
inputAPI.mouseEvent.add(onMouseEvent);
```

- Hook to keyboard signals

```
inputAPI.keyEvent.add(onKeyEvent);
```

- Hook to touch plugin

```
var touch = inputAPI.getPlugin("Touch");
if (touch)
{
        touch.touchEvent.add(onTouchEvent);
}
```

- Hook to gamepad plugin

```
var gamepad = inputAPI.getPlugin("Gamepad");
if (gamepad)
{
        //Ask if gamepad is supported
        if (gamepad.isBrowserSupported())
        {
                console.log("Gamepad supported");
        }
        else
        {
                console.log("Gamepad not supported");
```

```
        }

        gamepad.gamepadEvent.add(onGamepadEvent);
        gamepad.gamepadStatusEvent.add(onGamepadStatusEvent);
}
```

6.1.1 IInputPlugin

Extend IInputPlugin -class and create your own InputMyPlugin.js -file:

```
var IInputPlugin = Class.$extend(
{
    __init__ : function(name)
    {
        if (name === undefined)
        {
            console.error("[IInputPlugin]: Constructor called without a
plugin name!");
            name = "Unknown";
        }
        this.name = name;
        this.running = false;
    },

    __classvars__ :
    {
        register : function()
        {
            var plugin = new this();
            InputAPI.registerPlugin(plugin);
        }
    },

    _start : function(container)
    {
```

```
        this.start(container);
        this.running = true;
    },

    start : function()
    {
        console.log("[IInputPlugin]: Plugin '" + name + "' has not
implemented start()");
    },

    _stop : function()
    {
        this.stop();
        this.running = false;
    },

    stop : function()
    {
        console.log("[IInputPlugin]: Plugin '" + name + "' has not
implemented stop()");
    },

    reset : function()
    {
    }
});
```

- Register IInputPlugin

Place created plugin file reference after InputAPI.js -reference on your html -page:

```
<!-- Input SDK -->
<script src="../src/InputAPI.js"></script>
<!-- Touch plugin-->
<script src="../src/InputTouchPlugin.js"></script>
<!-- Gamepad plugin -->
```

```
<script src="../src/InputGamepadPlugin.js"></script>
```

6.1.2 InputAbstraction

- Include InputState.js to your html -page

```
<!-- InputState -->
<script src="../src/InputState.js"></script>
```

- Create input state

```
//InputState 1
var inputState = new InputState ({
        name : "Forward",
        keyBindings : ["w"],
        mouseDown : null,
        timeslot : 0
});
```

- Register InputState

```
var inputStateSignal = inputAPI.registerInputState(inputState);
```

- Create hook for signal returned

```
function onInputSignal(event)
{
        console.log("InputState "+inputState.name+" fired!");
}
```

- Hook signal returned from registering

```
inputStateSignal.add(onInputSignal);
```

- Update InputState

```
inputState.setName("new name");
inputState.setKeyBindings(["s"]);
```

```
inputState.setMouseDown(1);
inputState.setTimeslot(0);
inputState.setPriority(100);
inputState.setMultiplier(0)
inputAPI.updateInputState(inputState);
```

7 FIWARE OpenSpecification MiWi 3D-UI

Name	FIWARE.OpenSpecification.MiWi.3D-UI
Chapter	Advanced Middleware and Web-based UI,
Catalogue-Link to Implementation	3D-UI
Owner	DFKI, Philipp Slusallek

7.1 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.org and similar pages in order to understand the complete context of the FI-WARE project.

7.2 Copyright

- Copyright © 2013 by DFKI

7.3 Legal Notice

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

7.4 Overview

With the advent of WebGL, most web browsers got native 3D graphics support. WebGL gives low-level access to the graphics hardware suitable for graphics experts. But there is no way to describe interactive 3D graphics on a higher abstraction level, using web technologies such as DOM, CSS and Events. XML3D is a proposal for an extension to HTML5 to fill this gap and to provide web developers an easy way to create interactive 3D web applications.

7.5 Basic Concepts

XML3D is an extension to HTML5 for declarative 3D content represented as a scene graph like structure inside the DOM. All nodes within this graph are also nodes in the web sites DOM tree representation and can be accessed and changed via JavaScript like any other common DOM elements as well. On these DOM nodes, HTML events can be registered similar to known HTML elements. Resources for mesh data can be stored externally in any kind of external format (e.g. JSON, XML or binary) and referenced by URL. XML3D is designed to work efficiently with modern GPUs and Graphics API (such as OpenGL/WebGL) but still tries to stay independent of the rendering algorithm.

In addition to XML3D, Xflow allows to combine the scene graph with dataflows. Xflow is a declarative data flow representation that was designed for complex computations on XML3D elements. These computations include for example skinned meshes and key frame animations. In these cases, the current key frame takes the role of a data source in the graph, whereas mesh transformations are sinks in the dataflow. By this, changing the value of a key frame leads to a change in the posture of a mesh, and thus a continuous change of the key frame over time results in an animated mesh.

Both XML3D and Xflow are available as PolyFill implementation, i.e. all functionality to interpret XML3D nodes in a website and create a 3D view out of them is provided entirely by JavaScript implementations. This makes using XML3D with Xflow as easy as including the respective script files into the web application.

7.5.1 XML3D Element

The xml3d element is the root of a XML3D scene and also describes the rendering area this scene is displayed in. It can be placed in an arbitrary point inside the body element of an HTML page. In addition, it can be used inside XML document for external resources connected to the HTML page. Inside XML documents, the standard XML namespace rules apply to indicate which elements describe XML3D content.

- Rendering area: The xml3d element defines the dimension and the background appearance. Dimensions can either be defined by the attributes height and width, or - if those attributes are not given - the element can be sized arbitrarily by stype properties via the inline style attribute. The background of the rendering area can be defined using the CSS2 Background properties. The initial value of the background is transparent. If the xml3d element is embedded in some other rendering (like HTML) the background of the parent box's shines through, otherwise the background is black.
- Scene graph and transformation: The xml3d is the root node of the scene's graph. It defines the world coordinate system of the scene. All transformations defined by group child nodes are local coordinate systems relative to the world coordinate system. XML3D uses a Cartesian, right-handed, three-dimensional coordinate system.
- Views: The initial view to the scene is defined by the reference defined by the activeView attribute. If no activeView is defined, the renderer should set the activeView reference to the first applicable child node of the xml3d element. If the reference is not valid, or if there is no applicable view node as child node of the xml3d element, the renderer will not render the scene. In this case the rendering area should be filled with an error image. A script can set and change the activeView reference during runtime. If the activeView reference changes or gets deleted, the rules above are applied again.

7.5.2 Data Nodes

Data nodes combine multiple named value elements and can be referred and contained by data containers.

The elements data, mesh, shader, and lightshader are data containers that combine all contained value elements (int, float, float2, float3, float4, float4x4, bool, and texture) into a data table - a map with the name attribute of the value element as a unique key and the content of the value element as value. Value elements can be direct children of the data container or part of another data element that is either a child of the data container or referred via the src attribute.

In case multiple value elements with the same name are part of a data container, only one key-value-pair is included into the resulting named data table, according to the following rules:

- If the data container referres a data element via src, all child elements are ignored and the data table of the referred data element is reused directly
- A name-value pair of a child value element overrides a name-value pair with the same name of a child data element
- A name-value pair of a later child value element overrides a name-value pair with the same name of a former child value element
- A name-value pair of a later child data element overrides a name-value pair with the same name of a former child data element

7.5.3 Group Elements

Grouping node with transformation capabilities and surface shader assignment.

- Transformation: The group element defines a coordinate system for its children that is relative to the coordinate systems of its ancestors. The elements local coordinate system is defined as a 4x4 matrix. This matrix is determined from the CSS 'transform' property as defined in CSS 3D Transforms Module Level 3 and the reference to an element that can provide a transformation via the 'transform' attribute. The local matrix of a group node is calculated as:

```
Mcss * Mreference

   Where

Mcss

       is the matrix that is the result of the elements 'transform'
style property or the identity matrix if the style is not defined.

Mreference

       is the matrix that is provided from the element referenced via
the 'transform' attribute or the identity matrix if the reference is
not defined or not valid.
```

- Surface shader: The shader attribute of a group element defines the surface shading of all its children. The shading defined by parent nodes is overridden, while following group nodes can override the shading state again.

7.5.4 Mesh Elements

Geometry node that describes the shape of a polyhedral object in 3D.

This is very generic description of a 3D mesh. It clusters a number of data fields and binds them to a certain name. The interpretation of these data fields is job of the currently active shader. Only connectivity information is required to build the primitives defined by the type attribute:

- Triangles: A float3 element with name index is required. The data type of the bound element has to be evaluable to unsigned int. Every three entries in this field compose one triangle. The number of field entries should be an even multiple of 3. If not, the last entry or the last two entries are ignored. All other fields should have at least as many tuples as the largest value in the index field.

7.5.5 Transform Elements

General geometric transformation element, that allows to define a transformation matrix using five well understandable entities.

The center attribute specifies a translation offset from the origin of the local coordinate system (0,0,0). The rotation attribute specifies a rotation of the coordinate system. The scale field specifies a non-uniform scale of the coordinate system. Scale values may have any value: positive, negative (indicating a reflection), or zero. A value of zero indicates that any child geometry shall not be displayed. The scaleOrientation specifies a rotation of the coordinate system before the scale (to specify scales in arbitrary orientations). The scaleOrientation applies only to the scale operation. The translation field specifies a translation to the coordinate system.

The resulting matrix M that represents the element's coordinate system is calculated by a series of intermediate transformations. In matrix transformation notation, where C (center), SR (scaleOrientation), T (translation), R (rotation), and S (scale) are the equivalent transformation matrices, the resulting matrix is calculated as:

```
M = T * C * R * SR * S * -SR * -C
```

7.5.6 Shader Elements

The shader element describes a surface shader for a geometry.

The shader element connects arbitrary shader attributes with some shader code. The shader code is referenced with the script reference. The shader attributes are bound to the shader using the bind mechanism.

The URI syntax is used to define the shader script. This can be either a URL pointing to a script location in- or outside the current resource or a URN pointing to a XML3D standard shader. Following XML3D fixed-function shaders are defined:

- Matte: urn:xml3d:shader:matte
- Diffuse: urn:xml3d:shader:diffuse
- Phong: urn:xml3d:shader:phong

Example:

```
<shader id="red" script="urn:xml3d:shader:phong">
    <float3 name="diffuseColor">1 0 0</float3>
</shader>
```

7.5.7 Lights and Lightshaders

The light element defines a light in the scene graph.

The light source location and orientation is influenced by the scene graph transformation hierarchy. The radiation characteristics of the light source is defined by the referenced lightshader (s. shader attribute). The light can be dimmed using the intensity attribute and can be switched on/off using the visible attribute. If global is set to 'false', the light source will only light the objects that is contained in its parent group or xml3d element. Otherwise it will illuminate all the objects in its scene graph.

The light shader element describes a light source. The lightshader element connects arbitrary light shader attributes with a light shader code. The light shader code is referenced via the script reference. The shader attributes are bound to the shader using the data mechanism.

The URI syntax is used to define the light shader script. This can be either a URL pointing to a script location in- or outside the current resource or a URN pointing to a XML3D standard light shader. Following XML3D fixed-function light shaders are defined:

- Pointlight: urn:xml3d:lightshader:point
- Spotlight: urn:xml3d:lightshader:spot
- Directional Light: urn:xml3d:lightshader:directional

Example:

```
<lightshader id="myLight" script="urn:xml3d:lightshader:point">
    <float3 name="color">1 1 0.8</float3>
    <float3 name="attenuation">1 0 0</float3>
</lightshader>
```

7.5.8 Texture Elements

Set states on how to sample a texture from an image and to apply to a shape.

The texture source and its dimensions are defined by the texture element's children. The states how to apply the texture is set via the texture element's attributes. Use the attributes to influence

- the dimensions of the texture (type)
- how the texture is applied, if texture coordinates fall outside 0.0 and 1.0 (wrapS, wrapT, wrapU)
- how to apply the texture if the area to be textured has more or fewer pixels than the texture (filterMin, filterMag)
- how to create minified versions of the texture (filterMip)
- what border color to use, if one of the wrapping states is set to 'border'

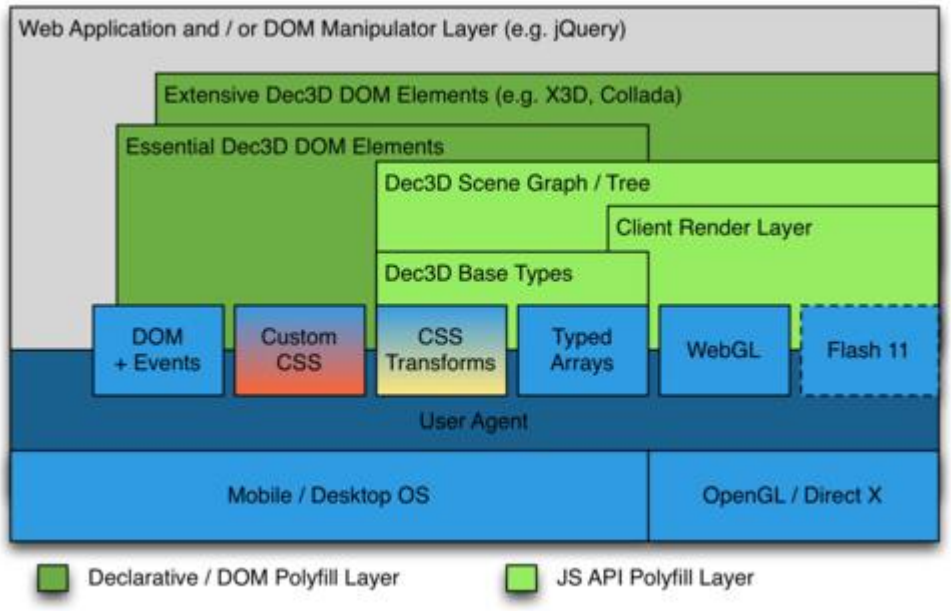
See the attribute documentation for more details.

```
Note: As per the OpenGL ES spec, a texture will be rendered black if:
*The width and height of the texture are not power-of-two and
*The texture wrap mode is not CLAMP_TO_EDGE or
*filterMin is neither NEAREST nor LINEAR
```

7.5.9 View Elements

The view node interface represents a camera in 3D world coordinates.

7.6 XML3D Generic Architecture



Apart from ease-of-use, an important goal of XML3D is to provide powerful and modern 3D graphics. Over time, 3D graphics APIs, such as OpenGL, evolved from fixed-function shading to a flexible rendering pipeline, based on programmable shaders combined with arbitrary buffers for data structures. Modern

3D graphics has relied on this flexibility ever since, achieving many visual effects based on custom shader code and data structures.

Many rendering systems support these capabilities via a fixed set of predefined features (such as character animations based one specific certain skeleton format). If users of these systems want to use other features not supported by default, they will have to work with low-level technologies, such as custom shader code.

In XML3D we provide the flexibility of modern graphics hardware through a generic, yet high-level architecture. Our goal is to support as many features of modern 3D graphics as possible while avoiding a large number of predefined features as well as reliance on low-level technologies.

A Web application that provides a 3D scene usually already contains a DOM manipulator layer. This layer can be realized by common JavaScript libraries, as for example jQuery, which provide a number of functions to query parts of the DOM and modify their parameters efficiently. We extend the DOM layer by sets of elements that describe the 3D scene. These declarative 3D DOM elements form a scene graph structure, consisting of a number of base types (as for example vectors) to describe the scene graph. As for other DOM elements as well, these elements can be accessed and modified both by existing DOM manipulation libraries and CSS properties.

The client renderer layer accesses existing renderer techniques of a user's web browser, for example WebGL, to interactively display the 3D scene in the client.

7.7 Main Interactions

7.7.1 Defining an XML3D scene in the web page

XML3D objects can directly be defined in the website's source using respective tags to define meshes, mesh groups, transformations, shaders (and more). Once XML3D.js is linked to the website, the browser can interpret all tags introduced by XML3D. Just adding an 'xml3d'-tag to the webpage will create a rendering canvas to display the scene. Meshes and groups of meshes are directly declared within the XML3D tag. Mesh resources and textures can be referenced as known for example from images in conventional web pages. By this, 3D content can be easily added to any webpage, without deeper knowledge of programming 3D applications.

The following example shows how to include a 3D model into a webpage. Data defining the mesh (like for example vertex data and texture coordinates) is stored in an external file and referenced by the 'src' -attribute of the mesh tag. It's transformation is described by CSS-style-transformations, whereas the used shader is declared by another XML3D element on the same webpage.

```
<xml3d xmlns="http://www.xml3d.org/2009/xml3d" >
  <shader id="orange" script="urn:xml3d:shader:matte">
    <float3 name="diffuseColor" >1 0.5 0</float3>
  </shader>
```

```
  <view position="0 0 100" />
  <group   shader="#orange"   style="transform:   translate3d(0px,-20px,
0px)" >
    <mesh src="resource/teapot.xml#mesh" />
  </group>
</xml3d>
```

7.7.2 Create XML3D elements in JavaScript

Whereas describing a static scene in advance in XHTML is a convenient way to quickly create 3D content, it is often not sufficient for complex 3D application. For these, objects are likely to appear or disappear during the runtime of applications, content has to be created with respect to certain user input

Like for other HTML elements as well, also XML3D elements can be generated dynamically by JavaScript. A newly created element is rendered as soon as it is added to the XML3D scene graph. Shaders and transformations can directly be applied once they are included in the website. The XML3D API provides functions to conveniently create new elements in JavaScript, so that they are handled and rendered correctly by the browser.

Existing elements are retrieved from the website's DOM tree by the common JavaScript functions. Not only adding elements, but also changing existing elements' attributes triggers a rendering of the scene, so that newly created elements as well as modifications to existing once are directly visible.

The following example shows how to add a new mesh to an existing XML3D scene: First, a new mesh element is created using the respective XML3D function to create new Elements. Using common JavaScript functions, the source file of the mesh vertex data is specified. Creating a group element that will contain the mesh, and adding the group element to the XML3D scene, will immediately render the new mesh.

```
// Create a new mesh element
var newMesh = XML3D.createElement("mesh");
newMesh.setAttribute("src", "teapot.xml#meshData");

// Create a new group element
var newGroup = XML3D.createElement("group");

// Append a mesh element to the group
newGroup.appendChild(newMesh)
```

```
// Append the new group element to an existing group
document.getElementById("MySceneRoot").appendChild(newGroup)
```

7.7.3 HTML event handler

HTML events are a common technique to interact with website content. To provide an easy way to bring interactivity also to XML3D scenes, event handlers like onclick or onmouseover can be registered on XML3D elements.

HTML:

```
  <group       id="teapot"       shader="#orange"       style="transform:
translate3d(0px,-20px, 0px)" onclick = "changeShader();">
    <mesh src="resource/teapot.xml#mesh" />
  </group>
```

JavaScript:

```
function changeShader() {
    document.getElementById("teapot").setAttribute("shader", "#green");
}
```

7.7.4 Using Camera Controllers

Users usually don't want to be provided with just one look onto a 3D scene, but like to inspect objects from different directions, or, for more complex scenes, navigate through the displayed world. If a standard navigation mode is sufficient for your web application, you can include the camera controller that comes with xml3d.js:

```
  <script src="http://www.xml3d.org/xml3d/script/xml3d.js"></script>
  <script
src="http://www.xml3d.org/xml3d/script/tools/camera.js"></script>
```

7.7.5 Processing Generic Data with Xflow

XML3D uses Xflow to process any generic data block inside the document. These capabilities are used to efficiently model any kind of expensive computation, e.g. for character animations, image processing and so on. The dataflow is declared in a functional way which allows for implicit parallelization e.g. by integrating the processing into the vertex shader.

Computations of a dataflow can consist of a number of computation steps, performed with predefined xflow operators to for example add or subtract values, morph or interpolate them. Intermediate results of chain of computations can be directly be used in future computation steps.

A dataflow that performs a morphing operation can look like this:

```
<data compute="position = xflow.morph(position , posAdd2 , weight2)" >
  <data compute="position = xflow.morph(position , posAdd1 , weight1)"
>
    <float3 name="position" >1.0 0.04 -0.5 ...</float3 >
    <float3 name="posAdd1" >0.0 1.0 2.0 ...</float3 >
    <float3 name="posAdd2" >1.0 0.0 0.0 ...</float3 >
    <float name="weight1" >0.35 </float >
    <float name="weight2" >0.6</float >
  </data >
</data >
```

With the <proto> element, we also allow to extract whole dataflow into external documents to make them reusable.

Prototype declaration:

```
<proto id=" doubleMorph" compute="pos = xflow.morph(pos , posAdd2 ,
w2)" >
  <data compute="pos = xflow.morph(pos , posAdd1 , w1)" >
    <int name="index" >0 1 2 ...</int>
    <float3 name="pos" >1.0 0.04 -0.5 ...</float3 >
    <float3 name="posAdd1" >0.0 1.0 2.0 ...</float3 >
    <float3 name="posAdd2" >0.0 1.0 2.0 ...</float3 >
    <float param="true" name="w1" ></float >
    <float param="true" name="w2" ></float >
  </data >
</proto >
```

Prototype instantiation:

```
<data id="instanceA" proto="# doubleMorph" >
```

```
  <float name="w1" >0</float >
  <float name="w2" >0.2</float >
</data >
<data id="instanceB" proto="# doubleMorph" >
  <float name="w1" >0.5</float >
  <float name="w2" >0</float >
</data >
```

Multiplatform support:

Xflow enables possibility to register operators with a same name to multiple platforms. It's possible to set "platform" attribute can be used to force a data sequence or a data flow to utilize a specific platform (js, gl or cl). If no "platform" attribute is defined, a default Xflow Graph platform will be used instead.

There is also a fallback feature which means for example if WebCL is not available Xflow uses Javascript automatically instead of WebCL acceleration. Currently all nodes in a dataflow chain is affected by fallback meaning that all nodes in dataflow will be either Javascript or WebCL nodes.

7.7.6 Hardware Accelerated Parallel Processing with Xflow

Xflow provides a possibility for effective parallel data processing on CPU or GPU device by utilising WebCL. This is especially useful if developers are using Xflow for processing big datasets. For smaller datasets the benefits are not so clearly visible.

Declaring a WebCL based Dataflow does not differ from ordinary Xflow declaration. If WebCL is available on users computer (by utilising Nokia's WebCL plugin on FireFox) WebCL-based data processing is automatically utilised by Xflow. Developers can also force the dataflow to utilise the WebCL platform by setting the optional "platform" attribute value to "cl".

Below is an example of how to declare a WebCL based dataflow and how to force the processing platform to be WebCL.

HTML:

```
<dataflow id="blurImage" platform="cl">
    <compute>
        blur =  xflow.blurImage(image, 9);
    </compute>
</dataflow>
```

However, in order to make the WebCL based data processing to work, a WebCL Xflow operator needs to be registered in a separate JavaScript script.

Below is an example of registering a WebCL Xflow operator. This operator applies a blur effect on the input "image" texture parameter and outputs the processed "result" texture.

Javascript:

```
Xflow.registerOperator("xflow.blurImage", {
    outputs: [
        {type: 'texture', name: 'result', sizeof: 'image'}
    ],
    params: [
        {type: 'texture', source: 'image'},
        {type: 'int', source: 'blurSize'}
    ],
    platform: Xflow.PLATFORM.CL,
    evaluate: [
        "const float m[9] = {0.05f, 0.09f, 0.12f, 0.15f, 0.16f, 0.15f, 0.12f, 0.09f, 0.05f};",
        "float3 sum = {0.0f, 0.0f, 0.0f};",
        "uchar3 resultSum;",
        "int currentCoord;",
        "for(int j = 0; j < 9; j++) {",
        "currentCoord = convert_int(image_i - (4-j)*blurSize);",
        "if(currentCoord >= 0 || currentCoord <= image_width * image_height) {",
        "sum.x += convert_float_rte(image[currentCoord].x) * m[j];",
        "sum.y += convert_float_rte(image[currentCoord].y) * m[j];",
        "sum.z += convert_float_rte(image[currentCoord].z) * m[j];",
        "}",
        "}",
        "resultSum = convert_uchar3_rte(sum);",
        "result[image_i] = (uchar4)(resultSum.x, resultSum.y, resultSum.z, 255);",
    ]});
```

The WebCL Xflow operator is designed in a way that allows a developer to focus purely on the core WebCL kernel programming logic. Developers can write their WebCL kernel code in the "evaluate" attribute of the operator, like shown in the example above. The code that can be written there is based in C language and the methods defined in the WebCL specification can be freely utilised. However, no kernel

function headers or input/output parameters need to be defined as they are created automatically by the underlying Xflow architecture.

Xflow processes "outputs" and "params" of the Xflow operator and allows them to be directly used in the WebCL kernel code. As seen in the example above, the input parameter "image" can be directly used in the code. An iterator for the first input parameter is also automatically generated and it can be safely used in the code. For the "image" param the iterator variable is named as "image_i". Also, some helper variables such as "image_height" and "image_width" are generated and likewise, they can be used in the evaluate code. Only the texture type parameters have height and width helper variable because textures or images are a special case; they are two-dimensional data stored in one-dimensional buffer. All other input parameter types have a "length" helper variable e.g. "parameterName_length" that determines the length of the input buffer.

Additionally, all WebCL application code needed for executing the WebCL kernel code (such as passing WebCL kernel arguments to the WebCL program and defining proper WebCl workgroup sizes) is generated automatically. Thus, developers need no deep knowledge of the WebCL programming and basic programming skills are enough to produce kernel code for simple WebCL Xflow operators.

Below is an example of a very simple WebCL Xflow operator. This operator is used for grayscaling an input texture. Only three lines of kernel code is required.

```
Xflow.registerOperator("xflow.desaturateImage", {
        outputs: [
            {type: 'texture', name: 'result', sizeof: 'image'}
        ],
        params: [
            {type: 'texture', source: 'image'}
        ],
        platform: Xflow.PLATFORM.CL,
        evaluate: [
            "uchar4 color = image[image_i];",
            "uchar lum = (uchar)(0.30f * color.x + 0.59f * color.y +
0.11f * color.z);",
            "result[image_i] = (uchar4)(lum, lum, lum, 255);"
        ]
    });
```

7.7.7 Mapping Synchronization GE data to XML3D objects

To integrate 3D-UI with network synchronization an abstract scene model is used in the client core. It is implemented as a Javascript library and also provides the JS API for application developers to create and

manipulate the scene. It uses the Entity-Component model (EC model for short), which is also used in the network protocol and on the server when implementing networked multi-user applications.

Objects in the EC model can be mapped directly to XML3D objects which allows an easy integration of 3D UI GE with the Synchronization GE. The API to create, remove and listen for changes in the scene is documented in the Synchronization GE docs.

- EC model *entities* are mapped to XML3D **<group>**

```
entity:
  <group>
```

- Scene objects like meshes or lights are mapped directly from the corresponding EC model component to an XML3D object, e.g.:

```
entity.mesh:
  <mesh>
entity.light:
  <light>
```

- A single XML3D element without an encapsulating **<group>** is also a EC model entity with the corresponding component. That is, **<mesh>** is same as **<group><mesh/></group>**: *entity.mesh*.

- Transformations that are described in a reX EC Placeable component are mapped to XML3D transforms and referenced by the corresponding entity:

```
entity.placeable.transform:
  <transform id="t">
  <group transform ="#t">
```

- Camera components are mapped to XML3D **<view>** elements:

```
entity.camera:
  <view>
```

- Hierarchies of entities in EC model are mapped to hierarchical **<group>** trees (nested tags) in XML3D

```
entity.placeable.parent:
  <group>
   <group>
          ...
   </group>
  </group>
```

- XML3D element attributes, for example **<mesh src=x>**, are represented in the scene model by the corresponding EC model attributes: *entity.mesh.meshRef.x*:

```
entity.mesh.meshRef = 'my.mesh':
  <mesh src="my.mesh" />
```

Harmonizing the attribute names to have them the same everywhere is under consideration.

- XML element attributes that are unknown to the EC model vocabulary are mapped directly:

```
entity.myattr = 1:
  <group myattr="1">
```

7.8 Basic Design Principles

Data definitions that are not subject to change during the application runtime, such as static objects, or local transformations within the scene graph of a complex object, should be stored externally and be referenced using external references

The performance of declarative 3D scenes is mainly influenced by the size of the DOM tree that is used to represent the scene. To improve performance, geometry should be grouped in a way that allows to keep the DOM tree as small as possible. An efficient way of grouping geometry is for example to group all meshes that share the same shader.

7.9 References

- XML3D Web Page
- XML3D GitHub Project
- XML3D – Interactive 3D Graphics for the Web
- xml3d.js: Architecture of a Polyfill Implementation of XML3D
- Xflow - Declarative Data Processing for the Web

7.10 Detailed Specifications

Following is a list of Open Specifications linked to this Generic Enabler. Specifications labeled as "PRELIMINARY" are considered stable but subject to minor changes derived from lessons learned during last interactions of the development of a first reference implementation planned for the current Major Release of FI-WARE. Specifications labeled as "DRAFT" are planned for future Major Releases of FI-WARE but they are provided for the sake of future users.

7.10.1 Open API Specifications

- XML3D Open API Specification
- XFlow Open API Specification

7.11 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

Annotations

Annotations refer to non-functional descriptions that are added to declaration of native types, to IDL interface definition, or through global annotations at deployment time. The can be used to express security requirements (e.g. "this string is a password and should be handled according the security policy defined for password"), QoS parameters (e.g. max. latency), or others.

AR

AR → Augmented Reality

Augmented Reality (AR)

Augmented Reality (AR) refers to the real-time enhancement of images of the real world with additional information. This can reach from the rough placement of 2D labels in the image to the perfectly registered display of virtual objects in a scene that are photo-realistically rendered in the context of the real scene (e.g. with respect to lighting and camera noise).

IDL

IDL → Interface Definition Language

Interface Definition Language

Interface Definition Language refers to the specification of interfaces or services. They contain the description of types and function interfaces that use these types for input and output parameters as well as return types. Different types of IDL are being used including CORBA IDL, Thrift IDL, Web Service Description Language (WSDL, for Web Services using SOAP), Web Application Description Language (WADL, for RESTful services), and others.

Middleware

Middleware is a software library that (ideally) handles all network related functionality for an application. This includes the setup of connection between peers, transformation of user data into a common network format and back, handling of security and QoS requirements.

PoI

PoI → Point of Interest

Point of Interest (PoI)

Point of Interest refers to the description of a certain point or 2D/3D region in space. It defines its location, attaches meta data to it, and defines a coordinate system relative to which additional coordinate systems, AR marker, or 3D objects can be placed.

Quality of Service (QoS)

Quality of Service refers to property of a communication channel that are non-functional, such a robustness, guaranteed bandwidth, maximum latency, jitter, and many more.

Real-Virtual Interaction

Real-Virtual Interaction refers to Augmented Reality setup that additionally allow users to interact with real-world objects through virtual proxies in the scene that monitor and visualize the state in

the real-world and that can use services to change the state of the real world (e.g. switch lights on an off via a virtual button the 3D scene).

Scene

A *Scene* refers to a collection of objects, which are be identified by type (eg. a 3D mesh object, a physics simulation rigid body, or a script object.) These objects contain typed and named data values (composed of basic types such as integers, floating point numbers and strings) which are referred to as attributes. Scene objects can form a hierarchic (parent-child) structure. A HTML DOM document is one way to represent and store a scene.

Security

Security is a property of an IT system that ensures confidentiality, integrity, and availability of data within the system or during communication over networks. In the context of middleware it refers to the ability of the middleware to guarantee such properties for the communication channel according to suitably expressed requirements needed and guarantees offer by an application.

Security Policy

Security Policy refers to rules that need to be fulfilled before a network connection is established or for data to be transferred. It can for example express statements about the identity of communication partners, properties assigned to them, the confidentiality measures to be applied to data elements of a communication channel, and others.

Synchronization

Synchronization is the act of transmitting over a network protocol the changes in a scene to participants so that they share a common, real-time perception of the scene. This is crucial to eg. implementing multi-user virtual worlds.

Type Description

Type Description in the context of the AMi middleware refers to the internal description of native data types or the interfaces described by an IDL. It contains data such as the name of a veriable, its data type, the hierarchical relations between types (e.g. structs and arrays), its memory offset and alignment within another data type, and others. Type Description are used to generate the mapping of native data types to the data that needs to be transmitted by the middleware.

Virtual Character

Virtual Character is a 3D object, typically composed of triangle mesh geometry, that can be moved and animated and can represent an user's presence (avatar) in a virtual world. Typically supported forms of animation include skeletal animation (where a hierarchy of "bones" or "joints" controls the deformation of the triangle mesh object) and vertex morph animation, where the vertices of the triangle mesh are directly manipulated. Virtual character systems may support composing the character from several mesh parts, for example separate upper body, lower body and head parts, to allow better customization possibilities.

8 XML3D Open API Specification

8.1 XML3D Elements

3D scenes that are described in XML3D are assembled from a number of DOM nodes that extend the standard HTML5 DOM nodes. Each node can carry one or more attributes to specify the respective element further. Operations on node attributes result on immediate changes in the visible scene.

8.1.1 xml3d

Root node of the scene graph, defines size and appearance of the rendering area

- Attributes:
    - height: height of the canvas drawing the 3D scene in pixels
    - width: width of the canvas drawing the 3D scene in pixels
    - activeView: id reference to the view element that is used to render the scene
- Example:

```
<xml3d width = "800" height = "600" activeView = "#camera1">
```

8.1.2 view

The view node interface represents a camera in 3D world coordinates

- Attributes:
    - position: Position of the camera in the szene
    - orientantion: orientation of the camera (as axis-angle)
    - fieldOfView: field of view of the camera
- Example:

```
<view id="camera1" orientation="0 0 1 0" position="0 -20 0"
fieldOfView="0.7">
```

8.1.3 data

Data node which combines multiple named value elements and can be referred and contained by data containers. Can be used as input or output for Xflow dataflow graphs.

- Attributes:
    - compute: Xflow expression that defines a computation for this data element

- src: Reference to another data element or data file as URI. Can be a location (URL) to a data file or an xml3d data element. If src is defined, all child elements are ignored. Thus, the data table defined by the referred content is reused directly.
- proto: Reference to another data node that should be treated as a template.
- filter: Defines a filter to specify which elements of the data node should be exposed as output of an Xflow computation

- Example:

```
<data id="myData">
   <float3 name="position">0 1 0</float3>
</data>
```

8.1.4 group

Grouping node with transformation capabilities and surface shader assignment

- Attributes:
  - visible: If "false", the element and all its children are not taken into account during rendering. This flag does not affect children referenced from other parts of the scene graph.
  - transform: Reference to an element that can provide a 3D transformation (i.e. transform)
  - shader: Reference to an element that can provide a surface shader (i.e. shader)

- Example:

```
<group id="meshGroup" transform="#transform1" shader="#shader1">
```

8.1.5 transform

General geometric transformation element, that allows to define a transformation matrix using five well understandable entities.

- Attributes:
  - translation: The translation part of the transformation.
  - scale: The scaling part of the transformation.
  - rotation: The rotation part of the transformation. (as axis-angle)
  - center: Origin for scale and rotation
  - scaleOrientation: Rotational orientation for scale

- Example:

```
 <transform id="transform1" translation="0 10 10" rotation="0 0 1 1.57"
scale="1 1 1.5">
```

8.1.6 shader

The shader element describes a surface shader for a geometry.

- Attributes:
  - script: Reference to the shader script as URI. Can be a location (URL) or one of the pre-defined shaders as URN. TODO: Should be of type "AnyURI"
  - src: Reference to another data element or data file as URI. Can be a location (URL) to a data file or an xml3d data element. If src is defined, all child elements are ignored. Thus, the data table defined by the referred content is reused directly.
  - proto: Reference to another data node that should be treated as a template. Children of this node will replaced input nodes that are marked with respective replaceby attribute.
- Example:

```
<shader id="shader1" script="urn:phong">
  <float3 name="diffuseColor">1 0 0</float3>
</shader>
```

8.1.7 mesh

- Attributes:
  - visible: If "false", the element and all it's children are not taken into account during rendering. This flag does not affect children referenced from other parts of the scene graph.
  - type: The type of geometric primitive described by this mesh element.
  - src: Reference to another data element or data file as URI. Can be a location (URL) to a data file or an xml3d data element. If src is defined, all child elements are ignored. Thus, the data table defined by the referred content is reused directly.

8.2 XML3D Scripting

XML3D scenes can be scripted entirely using the standard DOM scripting API. Changing attribute values of XML3D nodes leads to an immediate change of the scene.

9 XFlow Open API Specification

A description of default operators available in Xflow.

9.1 Basic Operations

9.1.1 xflow.add

Adds two float3 with each other, component wise.

result = value1 + value2

Signature

```
result = xflow.add(value1, value2)
@param float3      value1       first float3 value
@param float3      value2       second float3 value
@return float3     result       float3 of value: value1 + value2
```

9.1.2 xflow.sub

Substract two float3 with each other, component wise.

result = value1 - value2

Signature

```
result = xflow.sub(value1, value2)
@param float3      value1       first float3 value
@param float3      value2       second float3 value
@return float3     result       float3 of value: value1 - value2
```

9.1.3 xflow.normalize

Normalize a float3 value.

Signature

```
result = xflow.normalize(value)
@param float3      value        float3 value
@return float3     result       normalized float3 value
```

9.1.4 xflow.mul (Matrix)

Multiply two Transformations (float4x4). When using the resulting transformation on a vector, value1 is applied FIRST and value2 SECOND.

Signature

```
result = xflow.mul(value1, value2)
@param  float4x4 value1  first transformation
@param  float4x4 value2  second transformation
@return float4x4 result  multiplied transformation
```

9.2 Transformations

9.2.1 xflow.createTransform

Compute transformation matrix from basic transformation steps.

Signature

```
result = xflow.createTransform(translation, rotation, scale, center,
scaleOrientation)
@param float3        translation         translation of transformation
(optional)
@param float4         rotation              rotation of transformation
(quarternion)(optional)
@param float3      scale             scale of transformation (optional)
@param float3        center               origin of rotation and scale
(optional)
@param  float4            scaleOrientation    orientation of scaling
(quarternion) (optional)
@return float4x4[] result            final transformation (optional)
```

9.2.2 xflow.createTransformInv

Compute inverse transformation matrix from basic transformation steps.

Signature

```
result = xflow.createTransformInv(translation, rotation, scale, center,
scaleOrientation)
@param float3        translation         translation of transformation
(optional)
@param float4         rotation              rotation of transformation
(quarternion) (optional)
@param float3      scale             scale of transformation (optional)
@param float3        center               origin of rotation and scale
(optional)
```

```
@param  float4            scaleOrientation    orientation  of  scaling
(quarternion) (optional)
@return  float4x4[]  result                  final  inverse  transformation
(optional)
```

9.3 Morphing

9.3.1 xflow.morph

morph a float3 value, by adding a weighted offset value.

result = value + valueAdd * weight

Signature

```
result = xflow.morph(value, valueAdd, weight)
@param  float3 value     The base float3 value
@param  float3 valueAdd  A delta value that is added to the base value
@param  float  weight    weight that is multiplied with valueAdd before
being added
@return float3 result    result = value + weight * valueAdd
```

9.4 Sequences

9.4.1 xflow.lerpSeq

Linear interpolation of float3 values Output is a linear interpolation of two float3 of the sequence depending on the input key.

Signature

```
result = xflow.lerpSeq(sequence, key)
@param  float3 sequence  The sequence values
@param  float  key       determines two values of the sequence that are
interpolated
@return float3 result    interpolated position of the sequence
```

9.4.2 xflow.slerpSeq

Spherical Linear interpolation (slerp) of rotation values. Output is a linear interpolation of two rotations of the sequence depending on the input key.

Signature

```
result = xflow.slerpSeq(sequence, key)
```

```
@param  float4 sequence The base position value - this is expected to
be a sequence
@param  float  key      determines two values of the sequence that are
interpolated
@return float4 result   interpolated rotation of the sequence
```

9.5 Skinning

9.5.1 xflow.skinPosition

Signature

```
result = xflow.skinPosition(pos, boneIndices, boneWeights, boneXform)
@param  float3      pos         position to be skinned
@param    int4                boneIndices  4  indicies  referring  bone
transformations in boneXfm - indices must be >= 0
@param  float4      boneWeights 4 weights for each bone transformation
- sum of all weights has to be one
@param  float4x4[]  boneXform   an array of transformations, one for
each bone
@return float3      result      the skinned position
```

9.5.2 xflow.skinDirection

Signature

```
result = xflow.skinDirection(dir, boneIndices, boneWeights, boneXform)
@param  float3      dir         direction to be skinned
@param    int4                boneIndices  4  indicies  referring  bone
transformations in boneXfm - indices must be >= 0
@param  float4      boneWeights 4 weights for each bone transformation
- sum of all weights has to be one
@param  float4x4[]  boneXform   an array of transformations, one for
each bone
@return float3      result      the skinned position
```

9.5.3 xflow.forwardKinematics

This converts a relative transformation hierarchy into an array of absolute transformations. As input the script takes an array of transformations that represent a flattened hierarchy with the field transform. The field parent provides the necessary information which joint is the parent of a given joint. The result of the

script is the flattened transformation array where each transformation is the accumulation of all transformation from the bone to the root.

Signature

```
result = xflow.forwardKinematics(parent, xform)
@param int[]       parent  described the parent of each bone by index
@param float4x4[]  xform   transformation of each bone (local)
@return   float4x4[]   result      accumulated   transformations   (with
transformations of ancestors)
```

9.5.4 xflow.forwardKinematicsInv

Same as xflow.flattenBoneTransform, only produces the inverse transformation, by accumulating the transformations in the inverse order.

Signature

```
result = xflow.forwardKinematicsInv(parent, xform)
@param int[]       parent  described the parent of each bone by index
@param float4x4[]   xform     transformation of each bone (local and
inverse)
@return float4x4[] result   accumulated inverse transformations (with
transformations of ancestors)
```

10 FIWARE OpenSpecification MiWi Synchronization

Name	FIWARE.OpenSpecification.MiWi.Synchronization
Chapter	Advanced Middleware and Web-based UI,
Catalogue-Link to Implementation	Synchronization
Owner	LudoCraft Ltd., Lasse Öörni

10.1 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.org and similar pages in order to understand the complete context of the FI-WARE project.

10.2 Copyright

- Copyright © 2013-2014 by LudoCraft Ltd.

10.3 Legal Notice

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

10.4 Overview

10.4.1 Introduction

Interactive, multi-user Web applications such as virtual worlds often have the common requirements of storing the "world" or scene on a server, and communicating changes in it to participating clients. Depending on the application, the kinds of objects stored in the scene and the operations performed on them may differ. To allow for arbitrary use-cases a generic scene model is required, as well as open communication protocols. A scene should at the same time be easily accessible and editable from various external applications, but also allow high-performance synchronization for clients which are observing the scene and the changes in its content (such as objects moving due to a physics simulation taking place) in real-time.

In large-scale (3D) applications the scene data will need to be distributed across several server instances for example based on spatial partitioning.

This GE presents a lightweight server architecture and two communication protocols to interact with a synchronized dynamic scene hosted on a server: SceneAPI, a RESTful HTTP API for non-realtime querying

and modification of the scene, and a WebSocket-based bidirectional protocol for connected Web clients to receive continuous real-time scene updates, and to post their real-time changes to the scene.

10.5 Basic Concepts

This GE is centered on the concept of a scene, which is defined as generically as possible. It is not only limited to 3D virtual worlds, but could just as well represent for example a 2D spreadsheet, or an even more abstract collection of data.

10.5.1 Scene

A hierarchical collection of objects with arbitrary attribute values, which can be modified. The scene can be represented in one of the two following ways:

- A HTML DOM document, which is augmented with the elements from the XML3D [1] extension for 3D objects.
- The Entity-Component-Attribute structure as described in Scene and EC model. Entities are empty containers identified by unique integer IDs, with no functionality on their own, to which typed Components (for example a 3D Transform or a 3D Mesh) can be created. Entities can also contain child entities. The actual data of the Components exists in form of named and typed Attributes.

The mapping between the DOM and Entity-Component-Attribute scene representations is defined by the 3D-UI GE, which is concerned with actually rendering the scene in 3D. The synchronization mechanism itself does not need to make distinction between renderable and non-renderable scene data, or understand the mapping.

The basic data types supported for Attributes are:

- boolean
- integer
- float
- string

Compound and specialized attributes can be formed from these basic types, for example a 3D position vector from 3 float values, or a quaternion representing a rotation (or an RGBA color value) from 4 float values.

10.5.2 Server

A program hosting a synchronized scene, and which implements the communication protocols described below.

10.5.3 Client

A program which wishes to either observe or modify the scene hosted by a server. The client can use either, or both of the communication protocols defined below. A typical example would be a JavaScript-based client running in a Web browser.

10.5.4 SceneAPI

A RESTful HTTP API provided by the server, which allows for query and modify operations into the scene.

In addition to the scene query and modify operations, SceneAPI describes server-to-server communication operations with which a distributed, spatially partitioned large scene can be setup. The idea is that a server which hosts a part of a larger scene will "advertise" itself to neighbor servers (for example in a 2D grid formation).

10.5.5 Real-time Sync Protocol

A real-time, connection-oriented protocol that clients use to connect (login) into an interactive scene to observe the scene content, to observe changes, and to push their own changes, as well as to send and receive Entity Actions, a form of Remote Procedure Call. It is implemented using the WebSocket protocol.

Because of realtime processing performance and bandwidth demands, it is estimated that it is more efficient to operate using the Entity-Component-Attribute data model, than using a DOM scene representation. This allows to effectively encode attribute values in binary according to their type (for example integer values can be variable-length encoded according to their magnitude). From this it follows that the internal scene model of the server should be ECA-based.

10.5.6 Entity Action

A real-time command sent either by the client or the server to the other end, which does not directly modify the scene, but is instead interpreted by a script or plugin running on the client or the server. It can include a number of parameters, similar to a Component's Attributes. An example would be to transmit keyboard and mouse controls from the client to steer a moving object which is being simulated on the server.

10.5.7 Access control

The server may choose to limit access to scenes for both observing (read access) and modifying (write access). The actual authentication mechanisms are beyond the scope of this GE, but both the SceneAPI and the Real-time Sync Protocol provide for sending arbitrary credentials or access tokens: the SceneAPI in each operation, and the Sync Protocol during login to scene. The server implementation should provide hooks for authentication and access control.

10.5.8 Local objects

An important concept for the scene objects is to be able to opt-out of the synchronization, ie. be either server-local or client-local objects. This avoids wasting bandwidth for objects which are for example used as special effects only (ie. particle effects in a game.) These can be defined and queried with the SceneAPI, but will not be synchronized with the Real-time Sync Protocol.

10.5.9 Distributed scene

A scene that is distributed into smaller scene regions hosted on separate server instances. The servers need to be aware of neighbor servers to relay a client to the appropriate neighbor region(s) when the client is approaching the border of the server's own region.

10.5.10 Application-specific data

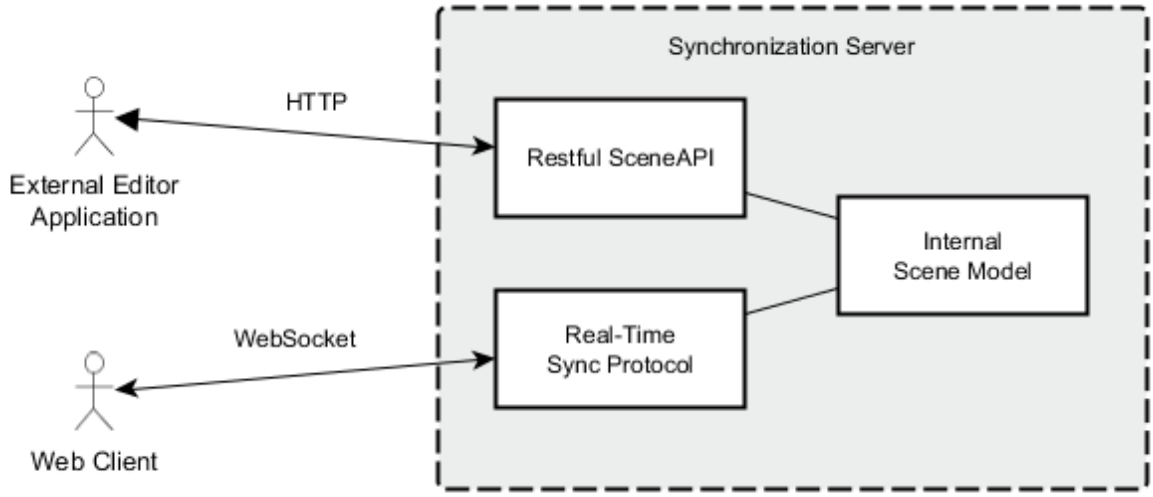
To contain arbitrary application-specific data, new combinations of Attributes can be dynamically introduced during runtime by either using the DynamicComponent mechanism, which allows adding Attributes to a component on the fly, or by registering a new static-structured custom component type by giving its type name and a list of the Attributes it should contain.

10.6 Generic Architecture

The server needs to hold an internal model of the scene, which is manipulated both by the SceneAPI and the Real-Time Sync Protocol. Both protocols can be implemented as plugins, which access the internal scene model.

In addition of a reference server implementation, a JavaScript library for making connections using the Real-Time Sync Protocol will be provided.

The diagram below describes the overall server architecture and examples of usage: a Web client making a real-time connection, and an external editor application performing REST calls to modify the scene.



10.7 Main Interactions

Interactions with the SceneAPI protocol and the Real-Time Sync Protocol are described separately. The SceneAPI operations are stateless, while a real-time server connection requires state to be maintained on both the server and the client.

10.7.1 SceneAPI

The SceneAPI operations can be divided into scene queries, scene modification, and distributed scene setup.

A scene query specifies the criteria for objects and how to return the results, ie. just an object list, or the full attribute content of the objects. Query criteria is either:

- The value of some attribute or a combination of them. For string attributes wildcards and regexes can be supported, to enable for example searching for objects named in a certain manner.
- A spatial query, for example all 3D objects contained within a sphere of certain center and radius.

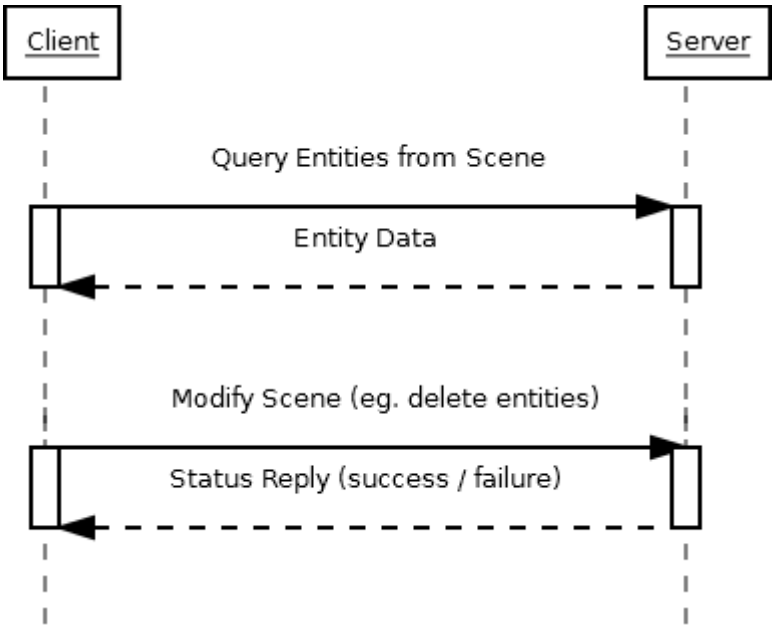
Scene modification consists of creating, modifying and removing objects. An object can either be created empty or with its full content already be specified on creation. Object hierarchies ("scene fragments") can also be created in one operation. The place in the hierarchy can be specified (eg. "create as child of this object")

In terms of the Entity-Component-Attribute model, object modification is the addition and removal of Components, and modifying Attribute values inside the Components.

The whole scene can also be created or queried at once.

Distributed scene setup is server-to-server communication that servers, which each host a piece of a larger scene, use to "advertise" themselves to neighbor servers. The operations consist of adding and removing a neighbor server (the spatial bounds of the partial scene must be known), as well as querying for already known neighbor servers.

The following sequence diagram illustrates basic interactions (query and modification) between the client and the server.



10.7.2 Real-Time Sync Protocol

The real-time connection into a server scene operates in phases:

- Login into the scene, including access credentials
- Server sends initial scene state
- Server sends additional scene updates, and the client may push its modifications to the scene, which the server will distribute to other connected clients. The server and client can also exchange RPC-like Entity Action commands
- Disconnection (leaving the scene)

After login, the scene updates consist of the following operations, which for the most part mirror the SceneAPI modification operations:

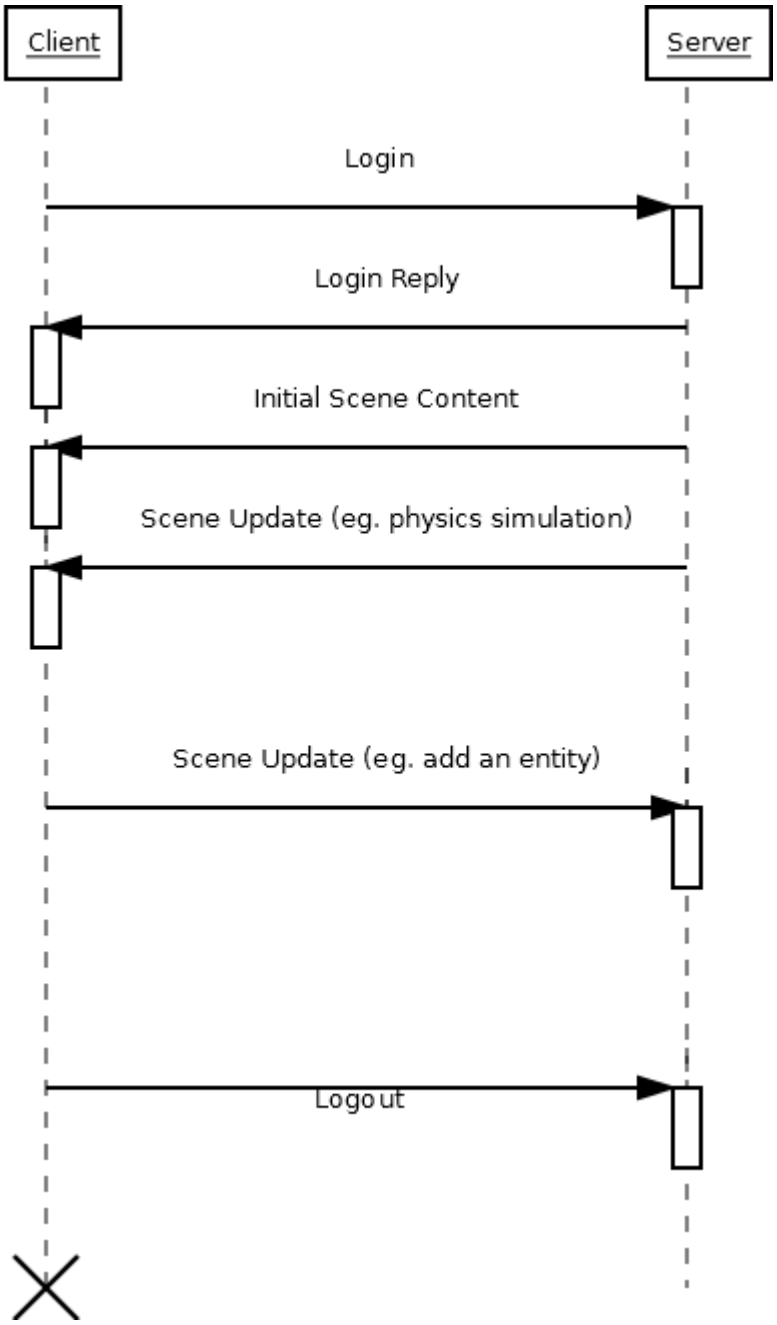
- Create an Entity (can be empty or include full Component / Attribute content)
- Create a Component into an Entity
- Modify an Attribute value in a Component
- Delete a Component from an Entity
- Delete an Entity
- Send an Entity Action
- Introduce a new static-structured custom component type: send its type name and the list of Attributes it should contain

The scene modification operations sent by a client may fail if the client does not have permission: in this case an error reply will be returned and the client should roll back its local scene state.

In its basic form the synchronization protocol sends all updates from the scene to all clients without regard for optimization. Several optimization strategies can be devised to reduce the bandwidth use and CPU load of both the server and the client: these are commonly called interest management. For example a metric based on object bounding box size and distance from observer can be used to determine whether an object is important enough to be synchronized to the client. This kind of interest management requires the client to stream its observer position to the server with sufficient frequency. In non-spatial scenes other interest management strategies would be required.

When a client is observing a spatially distributed scene, and is crossing from the boundary of one server to another, it will need to hold active connections to both of the servers at the same time.

An example session between a single client and server is depicted in the following sequence diagram:



10.8 Basic Design Principles

- The two communication protocols are independent, but manipulate the same internal scene data.
- The possibility for distributed scenes should not induce performance overhead when operating in a non-distributed (single-server) mode.

10.9 Detailed Specifications

Following is a list of Open Specifications linked to this Generic Enabler. Specifications labeled as "PRELIMINARY" are considered stable but subject to minor changes derived from lessons learned during

last interactions of the development of a first reference implementation planned for the current Major Release of FI-WARE. Specifications labeled as "DRAFT" are planned for future Major Releases of FI-WARE but they are provided for the sake of future users.

10.9.1 Open API Specifications

- SceneAPI RESTful API Specification
- Synchronization Open API Specification

10.10 Re-utilised Technologies/Specifications

- XML3D declarative HTML extension for 3D scenes
- Entity-Component-Attribute scene model

10.11 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

Annotations

Annotations refer to non-functional descriptions that are added to declaration of native types, to IDL interface definition, or through global annotations at deployment time. The can be used to express security requirements (e.g. "this string is a password and should be handled according the security policy defined for password"), QoS parameters (e.g. max. latency), or others.

AR

AR → Augmented Reality

Augmented Reality (AR)

Augmented Reality (AR) refers to the real-time enhancement of images of the real world with additional information. This can reach from the rough placement of 2D labels in the image to the perfectly registered display of virtual objects in a scene that are photo-realistically rendered in the context of the real scene (e.g. with respect to lighting and camera noise).

IDL

IDL → Interface Definition Language

Interface Definition Language

Interface Definition Language refers to the specification of interfaces or services. They contain the

description of types and function interfaces that use these types for input and output parameters as well as return types. Different types of IDL are being used including CORBA IDL, Thrift IDL, Web Service Description Language (WSDL, for Web Services using SOAP), Web Application Description Language (WADL, for RESTful services), and others.

Middleware

Middleware is a software library that (ideally) handles all network related functionality for an application. This includes the setup of connection between peers, transformation of user data into a common network format and back, handling of security and QoS requirements.

PoI

PoI → Point of Interest

Point of Interest (PoI)

Point of Interest refers to the description of a certain point or 2D/3D region in space. It defines its location, attaches meta data to it, and defines a coordinate system relative to which additional coordinate systems, AR marker, or 3D objects can be placed.

Quality of Service (QoS)

Quality of Service refers to property of a communication channel that are non-functional, such a robustness, guaranteed bandwidth, maximum latency, jitter, and many more.

Real-Virtual Interaction

Real-Virtual Interaction refers to Augmented Reality setup that additionally allow users to interact with real-world objects through virtual proxies in the scene that monitor and visualize the state in the real-world and that can use services to change the state of the real world (e.g. switch lights on an off via a virtual button the 3D scene).

Scene

A *Scene* refers to a collection of objects, which are be identified by type (eg. a 3D mesh object, a physics simulation rigid body, or a script object.) These objects contain typed and named data values (composed of basic types such as integers, floating point numbers and strings) which are referred to as attributes. Scene objects can form a hierarchic (parent-child) structure. A HTML DOM document is one way to represent and store a scene.

Security

Security is a property of an IT system that ensures confidentiality, integrity, and availability of data within the system or during communication over networks. In the context of middleware it refers to the ability of the middleware to guarantee such properties for the communication channel according to suitably expressed requirements needed and guarantees offer by an application.

Security Policy

Security Policy refers to rules that need to be fulfilled before a network connection is established or for data to be transferred. It can for example express statements about the identity of communication partners, properties assigned to them, the confidentiality measures to be applied

to data elements of a communication channel, and others.

Synchronization

Synchronization is the act of transmitting over a network protocol the changes in a scene to participants so that they share a common, real-time perception of the scene. This is crucial to eg. implementing multi-user virtual worlds.

Type Description

Type Description in the context of the AMi middleware refers to the internal description of native data types or the interfaces described by an IDL. It contains data such as the name of a veriable, its data type, the hierarchical relations between types (e.g. structs and arrays), its memory offset and alignment within another data type, and others. Type Description are used to generate the mapping of native data types to the data that needs to be transmitted by the middleware.

Virtual Character

Virtual Character is a 3D object, typically composed of triangle mesh geometry, that can be moved and animated and can represent an user's presence (avatar) in a virtual world. Typically supported forms of animation include skeletal animation (where a hierarchy of "bones" or "joints" controls the deformation of the triangle mesh object) and vertex morph animation, where the vertices of the triangle mesh are directly manipulated. Virtual character systems may support composing the character from several mesh parts, for example separate upper body, lower body and head parts, to allow better customization possibilities.

11 Synchronization Open API Specification

11.1 SceneAPI

The SceneAPI REST interface is described in the separate document SceneAPI RESTful API Specification.

11.2 Real-Time Sync Protocol

This API description is based on the currently implemented features and is subject to change once more features are implemented.

To maintain low coupling and eg. allow the scene model to be reused without networking, the JavaScript client API is divided into classes with different responsibilities:

- **WebSocketClient**: WebSocket connection management
- **Scene**, **Entity**, **Component** and **Attribute**: scene model implementation
- **SyncManager**: bidirectional synchronization of the scene with the server

11.2.1 Connection management

Initiating a WebSocket connection to a server happens through the WebSocketClient object, which exposes the following functions:

```
webSocketClient.connect(hostname, port, loginData)
webSocketClient.disconnect()
```

The loginData is optional JSON data that specifies user name, authentication information etc.

The WebSocketClient provides signals for the connect succeeding or failing, and provides access to sending and receiving raw binary messages which is utilized by the SyncManager. After a successful connection to the server the client is assigned and sent back an integer user ID and a reply data which is also JSON, and these can be read via the WebSocketClient's "userID" and "loginReplyData" properties.

11.2.2 Scene model

The Scene, Entity, Component and Attribute classes implement access to the Entity-Component-Attribute -based scene and are what the SyncManager manipulates to replicate scene modifications coming from the server.

These classes provide also signals for scene modification operations that happen on them (such as "entityCreated", "entityRemoved") and the SyncManager hooks into these to be able to send the client's scene modifications to the server.

All scene modification operations (creation, modification and removal of entities, components and attributes) are accompanied by a "change type" or signaling mode. It is possible to make changes to the

scene without sending them to the network, or even without signaling them internally at all. The default change mode should be used in most cases, and is also used if the change type parameter is omitted.

change type	Description
Default	Use the default signaling mode which makes most sense, which is Replicated for replicated entities/components, or LocalOnly for local entities / components
Replicate	If the entity or component in question is replicated, send the change as a network message
LocalOnly	Signal the change locally, but do not send a network message
Disconnected	Do not signal locally and do not send a network message

The entities in the scene and the components in an entity are primarily identified with their integer ID's. The ID's are split into different ranges based on their purpose:

ID Range	Purpose
0x00000001 0x3fffffff	Replicated entities / components. These are synchronized between the server and the client
0x40000001 0x7fffffff	Unacked entities / components. When a client creates an entity or component that should appear on the server, it allocates an "unacked" ID for it. This will be transformed by the server to an authoritatively allocated replicated ID. A reply message is sent to the client so that it knows also to change the ID of the entity or component it created.
0x80000001 0xffffffff	Local entities / components. These are created on the client only (or the server only) and willnot be synchronized
0x00000001 0x3fffffff	Replicated entities / components. These are synchronized between the server and the client

Next follows a brief description of the most important functions in the Scene, Entity and Component classes needed by the network synchronization.

11.2.2.1 *Scene*

```
scene.createEntity(id, changeType)
```

Create an empty entity into the scene. ID 0 will assign the next available. The change type decides whether to create a replicated (Replicate) or local (LocalOnly) entity.

```
scene.removeEntity(id, changeType)
```

Remove an entity by ID.

```
scene.entityById(id)
```

Return an entity by ID, or no entity if not found.

```
scene.entityByName(name)
```

Return an entity by string name. The name is contained in its Name component.

11.2.2.2 *Entity*

```
entity.createComponent(id, typeId, name, changeType)
```

Create a component to the entity. ID 0 will assign the next available. The typeId can be a string (such as "Mesh" or "Placeable") or an integer type value that is used internally in the network protocol for optimization. Components can be optionally given a string name.

```
entity.removeComponent(id, changeType)
```

Remove a component from the entity by ID.

```
entity.removeAllComponents(changeType)
```

Remove all components from the entity.

```
entity.triggerAction(name, params, execType)
```

Signal an RPC-like Entity Action. The action is identified with its string name, and the parameters are an array of strings. The execution type is a bit combination of the following constants: cExecTypeLocal, cExecTypeServer, cExecTypePeers, which means to either execute the action only locally, send it to the server, or to send it to all connected clients via the server.

The network synchronization of the action is also implemented through a signal, "actionTriggered", that the SyncManager hooks into.

```
entity.componentById(id)
```

Return a component from the entity by ID, or no component if not found.

```
entity.componentByType(typeId, name)
```

Return a component from the entity by its type, which can be a string or an integer type number. Optionally limit the query to components with the specified name.

11.2.2.3 *Component*

```
component.createAttribute(index, typeId, name, value, changeType)
```

Create a dynamic attribute to the component. Index starts from zero and should be allocated sequentially. Currently only the DynamicComponent supports dynamic attributes. Type id is a string name identifying the type (such as "string", "float3" or "bool"), or an integer type value that is used internally in the network protocol for optimization.

```
component.removeAttribute(index, changeType)
```

Remove a dynamic attribute by its zero-based index.

```
component.attributeById(id)
```

Returns an attribute by its string ID. The ID is a short identifier starting with a lowercase letter which is also the same as the property name of the attribute for shorthand access (ie. component.attributeName)

```
component.attributeByName(name)
```

Returns an attribute by its string name. This is separate from the ID and should be a longer descriptive name that is shown in eg. editor tools. Note that for dynamic attributes the ID and the name can not be separately specified, but are set to the same string value.

```
registerCustomComponent(typeName, blueprintComponent, changeType)
```

Registers a custom static-structured component. Its attribute structure (described by a "blueprint" component that should have been created beforehand) will be sent to the server the next time syncManager.sendChanges() is called. After that components of the new type can be created both on the client and the server.

11.2.3 Scene synchronization

To begin synchronizing scene content with the server, the Scene and SyncManager objects need to be created. On construction, the SyncManager needs to be given a WebSocketClient object that is connected to a server and a Scene, preferably empty of entities.

```
var syncManager = new SyncManager(webSocketClient, scene);
```

After this, the SyncManager automatically receives scene modification messages from the server and applies the changes to the Scene. To send pending changes made on the client side back to the server, the following function on the SyncManager needs to be called:

```
syncManager.sendChanges()
```

11.2.4 Binary protocol description

The binary messages sent between the client and server are described in detail in following document: Tundra protocol

In a WebSocket implementation, each message is sent as one binary WebSocket frame, with the message ID encoded as an unsigned little-endian 16-bit value in the beginning.

11.2.5 Model of operation in pseudocode

Implementing a synchronization client is easier than a server, as it does not need to handle several connections. A client needs only to listen to the binary protocol messages as they arrive from the server, and perform them on its local scene. It also needs to listen to the change signals from the local scene, and send those that have the change type Replicated back to the server.

The server, on the other hand, should for efficiency operate on network "ticks" or "frames" that happen for example 20 or 30 times per second. It collects all the scene changes up to the next tick, then processes them in a batched manner. If for example the position in an entity's Transform component changes several times before the next tick, only the latest data should be sent to conserve bandwidth.

The server needs to maintain a structure, called here the SyncState, for each client connections that contains the list of "dirty" (contains changes that need to be sent) entities, components and attributes. When it is time to perform the next network tick, the default (naive) operation mode is to go through this structure for all client connections, and send all pending changes. More intelligent per-client operations such as interest management (throttling the update rate or excluding updates completely for far away entities) or overall bandwidth throttling can be performed. In pseudocode, the algorithm for going through the SyncState of a client is the following:

```
Go through any newly registered component types. Send them to the client

Go through all dirty entities in the SyncState

 If entity was removed, send RemoveEntity message

 If entity is new for the client, send CreateEntity message with full component and attribute data

 Otherwise go through its dirty components

  If component was removed, collect it into a RemoveComponents message

  If component is new for the client, collect it into a CreateComponents message with full attribute data

  Otherwise go through its dirty attributes

   If attribute is dynamic and was removed, collect it into a RemoveAttributes message

   If attribute is dynamic and is new for the client, collect it into a CreateAttributes message

   Otherwise (the attribute is modified) collect it into an EditAttributes message

  Send RemoveComponents, CreateComponents, RemoveAttributes, CreateAttributes, EditAttributes messages that have data in them

  Clear the dirty status for the entity and all its components & attributes
```

12 SceneAPI RESTful API Specification

12.1 Introduction to the Scene API

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

Interactive, multi-user Web applications such as virtual worlds often have the common requirements of storing the "world" or scene on a server, and communicating changes in it to participating clients. In some applications, such as external agents manipulating the scene infrequently, it is not necessary to use a real-time, live connection (for example WebSocket) for synchronizing the scene data, but instead providing a RESTful API for queries and modification gives the advantages of less continuous processing load on both the server and the client, and using a standard communication method instead of eg. a custom binary network protocol.

12.1.1 Scene API Core

The Scene API is a RESTful, resource-oriented API accessed via HTTP that uses XML-based representations for information interchange. It is used to query and manipulate the data contained in scene objects, as well as to manipulate the structure of the scene itself (creation and deletion of scene objects.) The scene data is organized according to Entity-Component-Attribute structure described in Scene and EC model: the individual scene objects are called entities, which contain components, which contain the actual data payload as attributes.

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 using the supported operations. For the latter, this specification provides a full specification of how to implement those operations.

In order to use this specification, the reader should firstly have a general understanding of the Synchronization Generic Enabler, that implements this API.

12.1.3 API Change History

The Scene API version history is described in the table below:

Revision Date	Changes Summary
Mar 14, 2014	• Initial version

12.1.4 How to Read This Document

All FI-WARE RESTful API specifications will follow the same list of conventions and will support certain common aspects. Please check Common aspects in FI-WARE Open Restful API Specifications.

For a description of some terms used along this document, see the Terns and Definitions.

12.1.5 Additional Resources

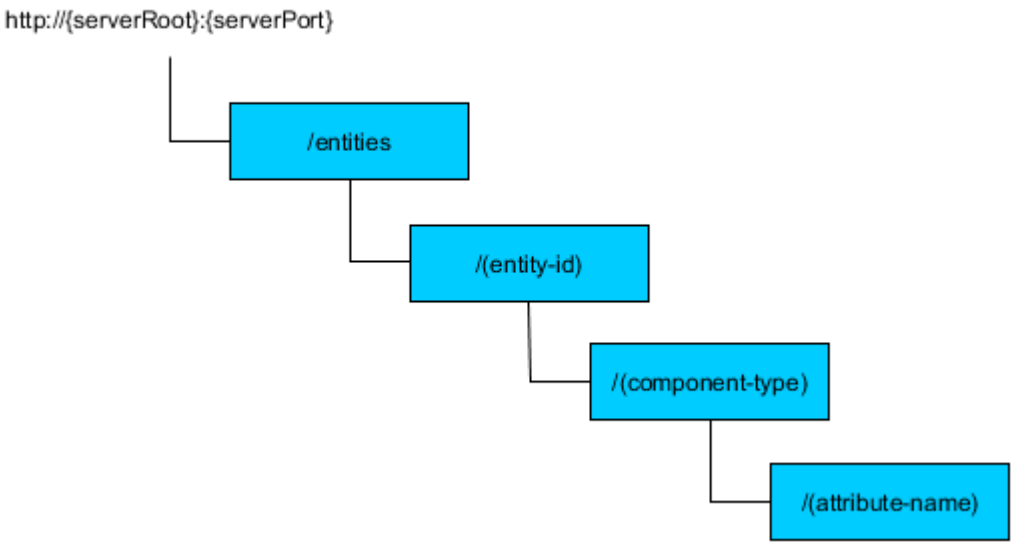
Here you can describe any other information or resources that you could need to understand this document. It could include also the link <url> in which to can obtain this specification and the link to obtain the schemas that we are using. E.g.

> "You can download the most current version of this document from the FIWARE API specification website at <link to the url>. For more details about the <name of the GE service> that this API is based upon, please refer to <link to the High Level Description>. Related documents, including an Architectural Description, are available at the same site."

12.2 General Scene API Information

12.2.1 Resources Summary

The following diagram describes the resources that can be accessed with the Scene API, starting from the server base URL.



Scene API resource summary

12.2.2 Representation Format

The Scene API supports data in XML format.

12.2.3 Resource Identification

All operations (GET, PUT, POST, DELETE) to the Scene API require the target resource to be identified. For HTTP transport, this is made using the mechanisms described by HTTP protocol specification as defined by IETF RFC-2616.

An operation with only the /entities identifier refers to the whole scene. This can be narrowed down first to an entity to be operated on by appending the entity's integer ID, for example /entities/123. Furthermore, it can be narrowed down to a specific component type, and finally to a specific attribute name, eg. /entities/123/placeable/transform.

12.2.4 Links and References

N/A.

12.2.5 Limits

N/A.

12.2.6 Versions

N/A.

12.2.7 Extensions

N/A.

12.2.8 Faults

12.2.8.1 *Synchronous Faults*

Fault Element	Associated Error Codes	Expected in All Requests?
notFound	404	The resource was not found for a query operation.
badRequest	400	The resource identifier was invalid for a modification operation, or XML data submitted in a POST operation was invalid.

When a fault happens, it is also transmitted in the reply body as text for convenience.

12.2.8.2 *Asynchronous Faults*

All current SceneAPI operations are synchronous in nature; therefore there are no asynchronous faults.

12.3 API Operations

12.3.1 Scene queries

12.3.1.1 *Retrieve data of all entities*

Verb	URI	Description
GET	/entities	Retrieve data of all scene entities

Normal Response Code(s): 200

Error Response Code(s): 404

Example response:

```
<scene>
 <entity temporary="false" id="1" sync="true">
  <component temporary="false" typeId="5" type="Script" sync="true">
   <attribute value="avatarmenu.js" type="AssetReferenceList" id="scriptRef" name="Script ref"/>
   <attribute value="true" type="bool" id="runOnLoad" name="Run on load"/>
   <attribute value="0" type="int" id="runMode" name="Run mode"/>
   <attribute value="" type="string" id="applicationName" name="Script application name"/>
   <attribute value="" type="string" id="className" name="Script class name"/>
  </component>
  <component temporary="false" typeId="26" type="Name" sync="true">
   <attribute value="AvatarMenu" type="string" id="name" name="Name"/>
   <attribute value="" type="string" id="description" name="Description"/>
   <attribute value="" type="string" id="group" name="Group"/>
  </component>
 </entity>
 <entity temporary="false" id="2" sync="true">
  <component temporary="false" typeId="5" type="Script" sync="true">
   <attribute value="avatarapplication.js;simpleavatar.js;exampleavataraddon.js" type="AssetReferenceList" id="scriptRef" name="Script ref"/>
   <attribute value="true" type="bool" id="runOnLoad" name="Run on load"/>
   <attribute value="0" type="int" id="runMode" name="Run mode"/>
   <attribute value="AvatarApp" type="string" id="applicationName" name="Script application name"/>
   <attribute value="" type="string" id="className" name="Script class name"/>
```

```
  </component>
  <component temporary="false" typeId="26" type="Name" sync="true">
   <attribute value="AvatarApp" type="string" id="name" name="Name"/>
   <attribute value="" type="string" id="description" name="Description"/>
   <attribute value="" type="string" id="group" name="Group"/>
  </component>
 </entity>
</scene>
```

12.3.1.2 *Retrieve data of an entity*

Verb	URI	Description
GET	/entities/(entity-id)	Retrieve data of a single scene entity

Normal Response Code(s): 200

Error Response Code(s): 404

Example response:

```
<entity temporary="false" id="1" sync="true">
 <component temporary="false" typeId="5" type="Script" sync="true">
  <attribute value="avatarmenu.js" type="AssetReferenceList" id="scriptRef" name="Script ref"/>
  <attribute value="true" type="bool" id="runOnLoad" name="Run on load"/>
  <attribute value="0" type="int" id="runMode" name="Run mode"/>
  <attribute value="" type="string" id="applicationName" name="Script application name"/>
  <attribute value="" type="string" id="className" name="Script class name"/>
 </component>
 <component temporary="false" typeId="26" type="Name" sync="true">
  <attribute value="AvatarMenu" type="string" id="name" name="Name"/>
  <attribute value="" type="string" id="description" name="Description"/>
  <attribute value="" type="string" id="group" name="Group"/>
 </component>
```

```
</entity>
```

12.3.1.3 *Retrieve data of a component within an entity*

Verb	URI	Description
GET	/entities/(entity-id)/(component-type)	Retrieve data of a single component within a scene entity

Normal Response Code(s): 200

Error Response Code(s): 404

Example response:

```
<component temporary="false" typeId="5" type="Script" sync="true">
 <attribute value="avatarmenu.js" type="AssetReferenceList" id="scriptRef" name="Script ref"/>
 <attribute value="true" type="bool" id="runOnLoad" name="Run on load"/>
 <attribute value="0" type="int" id="runMode" name="Run mode"/>
 <attribute value="" type="string" id="applicationName" name="Script application name"/>
 <attribute value="" type="string" id="className" name="Script class name"/>
</component>
```

12.3.1.4 *Retrieve the value of an attribute within a component*

Verb	URI	Description
GET	/entities/(entity-id)/(component-type)/(attribute-name)	Retrieve value of an attribute within a component

Normal Response Code(s): 200

Error Response Code(s): 404

The returned data is the attribute's value as text.

12.3.2 Creation of new scene objects

12.3.2.1 *Create a new entity with a server-assigned ID*

Verb	URI	Description

POST	/entities	Create a new entity into the scene, with server assigned ID.

Normal Response Code(s): 200

Error Response Code(s): 400

The POST request body may contain XML-serialized data to be put inside the entity (components & their attribute values). If omitted, an empty entity is created. The server will assign an entity ID to the new entity and return the entity's data as XML. An example response:

```
<entity temporary="false" id="1" sync="true"/>\n
```

12.3.2.2 *Create a new entity with a specified ID*

Verb	URI	Description
POST	/entities/(entity-id)	Create a new entity into the scene, with the specified ID.

Normal Response Code(s): 200

Error Response Code(s): 400

The POST request body may contain XML data to be put inside the entity (components & their attribute values). If omitted, an empty entity is created. The server will return the new entity's data as XML. If there is an ID conflict with an existing entity, the server will choose another ID. An example response:

```
<entity temporary="false" id="10" sync="true"/>\n
```

12.3.2.3 *Create a new component into an entity*

Verb	URI	Description
POST	/entities/(entity-id)/(component-type)	Create a new component into the scene entity.

Normal Response Code(s): 200

Error Response Code(s): 400

The POST request body may contain initial attribute values as XML to be put into the created component. If omitted, the attributes will have default values. The server will return the new component's data as XML. An example response of creating a component of type Placeable (scene node transformation) with default values:

```
<component temporary="false" typeId="20" type="Placeable" sync="true">
 <attribute
value="0.000000,0.000000,0.000000,0.000000,0.000000,0.000000,1.000000,1
.000000,1.000000" type="Transform" id="transform" name="Transform"/>
```

```
 <attribute value="false" type="bool" id="drawDebug" name="Show
bounding box"/>
 <attribute value="true" type="bool" id="visible" name="Visible"/>
 <attribute value="1" type="int" id="selectionLayer" name="Selection
layer"/>
 <attribute value="" type="EntityReference" id="parentRef" name="Parent
entity ref"/>
 <attribute value="" type="string" id="parentBone" name="Parent bone
name"/>
</component>
```

12.3.3 Modification of scene objects

12.3.3.1 *Set new attribute value inside a component*

Verb	URI	Description
PUT	/entities/(entity-id)/(component-type)?(attribute-name)=(new-attribute-value)	Set new value of an attribute inside a component.

Normal Response Code(s): 200

Error Response Code(s): 400

The server will return the component's new attribute values as XML.

12.3.3.2 *Set new attribute values inside a component*

Verb	URI	Description
PUT	/entities/(entity-id)/(component-type)	Set new attribute values inside a component.

Normal Response Code(s): 200

Error Response Code(s): 400

The attribute values are to be contained inside the PUT request body as XML. The server will return the component's new attribute values as XML.

12.3.3.3 *Set entity's component data*

Verb	URI	Description
PUT	/entities/(entity-id)	Set the scene entity's component data.

Normal Response Code(s): 200

Error Response Code(s): 400

Existing components in the entity will be deleted. The new component(s) and their attribute values are to be contained inside the PUT request body as XML. The server will return the entity's new data as XML.

12.3.4 Deletion of scene objects

12.3.4.1 *Delete an entity*

Verb	URI	Description
DELETE	/entities/(entity-id)	Delete an entity from the scene.

Normal Response Code(s): 200

Error Response Code(s): 400

12.3.4.2 *Delete a component from an entity*

Verb	URI	Description
DELETE	/entities/(entity-id)/(component-type)	Delete a component from the specified scene entity.

Normal Response Code(s): 200

Error Response Code(s): 400

13 FIWARE OpenSpecification MiWi CloudRendering

Name	FIWARE.OpenSpecification.MiWi.CloudRendering
Chapter	Advanced Middleware and Web-based UI,
Catalogue-Link to Implementation	Cloud Rendering
Owner	Adminotech Oy, Jonne Nauha

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.org and similar pages in order to understand the complete context of the FI-WARE project.

13.2 Copyright

- Copyright © 2013 by Adminotech Oy

13.3 Legal Notice

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

13.4 Overview

In some cases it may be impossible, inconvenient, or not allowed to transmit and render the user interface content on a client device. Performance may be inadequate for achieving certain user experience goals, it may be more appropriate to save battery power, or for IP reasons, a designer may decide that the content of the user interface not be delivered to a client machine. In such cases, it should be possible that the UI is rendered on a server in the cloud, forwarding the display to and receiving input from client in a remote location.

The goal of this GE is to provide a generic way to request, receive and control a video stream of a remote 3D application. The complexity and usual heavy performance requirements for a 3D application can be offloaded to a server, from a low end device that could not handle the rendering otherwise.

13.5 Basic Concepts

13.5.1 Web Service

This web service receives requests from clients for video streams. This service should have a generic API for requesting, controlling and closing video streams.

The service logic behind the API depends on the application. In a usual case this will require starting a process to handle the rendering and to actually serve the video stream to the end user. The web service needs to communicate connection information (host, port) for the end user to continue communications with the actual streaming server.

13.5.2 Renderer

Depending on the application this process may serve one or multiple end users with a video stream. Once this server starts it will communicate with the **Web Service** a WebRTC port that gets communicated back to the end user.

WebRTC will be utilized for live video/audio streaming and additionally its data channel can be used to send custom communications, e.g. input events from the client.

13.5.3 Web Client

The client will be an example for the web browser and JavaScript side on how to communicate with the **Web Service** and continue to receive the video stream from the **Streaming Server**.

13.6 Generic Architecture

Architecturally Cloud Rendering splits into three main component. The web service, web client(s) and renderer(s). Web service is the top level service that acts as the WebSocket server and helps clients connect to a renderer. Once the WebRTC connection is established between the renderer and the client, video streaming and additional input events are communicated via the WebRTC connection.

The WebSocket protocol is defined in the Cloud Rendering Open API Specification page.

13.7 Main Interactions

Here is a simplified look at the interactions between the three parts in this GE.

```
Sender           Message                              Receiver(s)          Notes
-------------------------------------------------------------------------------


// A new renderer is started
Renderer      -> Registration                   -> Web Service
```

```
// A new client registers to the service
Client         -> Registration                    -> Web Service          Clients
peerId = "1"

// Web service assigns a renderer and the client to the same room
Web Service    -> RoomAssigned                    -> Renderer & Client
                    "roomId" : "room one",
                    "error"  : 0

// Client wants to start the video stream
Client         -> Offer                           -> Web Service
Web Service    -> Offer                           -> Renderer

Renderer       -> Answer                          -> Web Service
                    "receiverId" : "1"
Web Service    -> Answer                          -> Client

// Client and server start a peer to peer video stream
Renderer      <-> WebRTC                         <-> Client
```

13.8 Basic Design Principles

- The protocol should **NOT** force any application logic. It should be a generic service that registers renderers, clients and joins them together for p2p communications.
- The protocol **should** allow any kind of application level messaging to be implemented within the spec. The "Application" channels messages is meant for this, the data in these messages is free for the GE implementation to exploit. If you implement all three components you can do custom messages from any component to the renderer and the service. However, if you only wish to do client-to-client custom messaging you can use the reference implementation of the service and renderer.
- Similarly any application specific input handling should be done at the application level messaging. Input is too complex to make a generic works-in-all-apps mold, it is best left for the implementation to handle. We provide structured application level messaging directly in the protocol.
- The protocol needs to support delayed/lazy startup of renderer processes. We cannot assume all renderers are always running, should be able to be started on demand.

- The system needs to scale across multiple physical machines on the network. You can have load balancing in from of multiple Cloud Rendering **Services** to have sufficient amount of active/open WebSocket connections.

13.9 Detailed Specifications

Following is a list of Open Specifications linked to this Generic Enabler. Specifications labeled as "PRELIMINARY" are considered stable but subject to minor changes derived from lessons learned during last interactions of the development of a first reference implementation planned for the current Major Release of FI-WARE. Specifications labeled as "DRAFT" are planned for future Major Releases of FI-WARE but they are provided for the sake of future users.

13.9.1 Open API Specifications

- Cloud Rendering Open API Specification

13.10 Re-utilised Technologies/Specifications

- WebRTC
    - Library will be a main component in the specification, WebRTC is used to establish a peer to peer connection and streaming the rendering results to the clients from the renderer. Web client will use existing deployed web browser support for WebRTC.
    - W3C Working Draft for WebRTC 1.0
    - Project homepage
- realXtend Tundra
    - Tundra will be used to implement the reference solution server plugins that implements the video streaming to the end user. The work will be done and published as open source.
    - Project homepage

13.11 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

Annotations

Annotations refer to non-functional descriptions that are added to declaration of native types, to IDL interface definition, or through global annotations at deployment time. The can be used to

express security requirements (e.g. "this string is a password and should be handled according the security policy defined for password"), QoS parameters (e.g. max. latency), or others.

AR

AR → Augmented Reality

Augmented Reality (AR)

Augmented Reality (AR) refers to the real-time enhancement of images of the real world with additional information. This can reach from the rough placement of 2D labels in the image to the perfectly registered display of virtual objects in a scene that are photo-realistically rendered in the context of the real scene (e.g. with respect to lighting and camera noise).

IDL

IDL → Interface Definition Language

Interface Definition Language

Interface Definition Language refers to the specification of interfaces or services. They contain the description of types and function interfaces that use these types for input and output parameters as well as return types. Different types of IDL are being used including CORBA IDL, Thrift IDL, Web Service Description Language (WSDL, for Web Services using SOAP), Web Application Description Language (WADL, for RESTful services), and others.

Middleware

Middleware is a software library that (ideally) handles all network related functionality for an application. This includes the setup of connection between peers, transformation of user data into a common network format and back, handling of security and QoS requirements.

PoI

PoI → Point of Interest

Point of Interest (PoI)

Point of Interest refers to the description of a certain point or 2D/3D region in space. It defines its location, attaches meta data to it, and defines a coordinate system relative to which additional coordinate systems, AR marker, or 3D objects can be placed.

Quality of Service (QoS)

Quality of Service refers to property of a communication channel that are non-functional, such a robustness, guaranteed bandwidth, maximum latency, jitter, and many more.

Real-Virtual Interaction

Real-Virtual Interaction refers to Augmented Reality setup that additionally allow users to interact with real-world objects through virtual proxies in the scene that monitor and visualize the state in the real-world and that can use services to change the state of the real world (e.g. switch lights on an off via a virtual button the 3D scene).

Scene

A *Scene* refers to a collection of objects, which are be identified by type (eg. a 3D mesh object, a physics simulation rigid body, or a script object.) These objects contain typed and named data values (composed of basic types such as integers, floating point numbers and strings) which are referred to as attributes. Scene objects can form a hierarchic (parent-child) structure. A HTML DOM document is one way to represent and store a scene.

Security

Security is a property of an IT system that ensures confidentiality, integrity, and availability of data within the system or during communication over networks. In the context of middleware it refers to the ability of the middleware to guarantee such properties for the communication channel according to suitably expressed requirements needed and guarantees offer by an application.

Security Policy

Security Policy refers to rules that need to be fulfilled before a network connection is established or for data to be transferred. It can for example express statements about the identity of communication partners, properties assigned to them, the confidentiality measures to be applied to data elements of a communication channel, and others.

Synchronization

Synchronization is the act of transmitting over a network protocol the changes in a scene to participants so that they share a common, real-time perception of the scene. This is crucial to eg. implementing multi-user virtual worlds.

Type Description

Type Description in the context of the AMi middleware refers to the internal description of native data types or the interfaces described by an IDL. It contains data such as the name of a veriable, its data type, the hierarchical relations between types (e.g. structs and arrays), its memory offset and alignment within another data type, and others. Type Description are used to generate the mapping of native data types to the data that needs to be transmitted by the middleware.

Virtual Character

Virtual Character is a 3D object, typically composed of triangle mesh geometry, that can be moved and animated and can represent an user's presence (avatar) in a virtual world. Typically supported forms of animation include skeletal animation (where a hierarchy of "bones" or "joints" controls the deformation of the triangle mesh object) and vertex morph animation, where the vertices of the triangle mesh are directly manipulated. Virtual character systems may support composing the character from several mesh parts, for example separate upper body, lower body and head parts, to allow better customization possibilities.

14 Cloud Rendering Open API Specification

14.1 Protocol

The **Renderer** and the **Web Client** will communicate with the **Web Service** with a WebSocket connection. The service will act as a WebSocket server, this host information needs to be communicated or known by the renderer and client.

14.1.1 Messages

Basic Structure

All of the protocol messages are sent and received as JSON. The basic message structure is the following:

```
{
    "channel" : <message-channel-as-string>,
    "message" :
    {
        "type" : <message-type-as-string>,
        "data" :
        {
            <message-data>
        }
    }
}
```

Channels

Each message in the protocol will hold a top level "channel" type. The message will have the channel as string, the number can be used internally by the GE implementation.

The message will always have a unique "type", so the channel type is not meant for message identification. It is intended for message flow control on the GE implementation, as you will most likely have separate logic handling each of the channel types.

```
{
    "Signaling"   : 1,
    "Room"        : 2,
    "State"       : 3,
    "Application" : 4
```

```
}
```

Types

The type is mean for message identification. Each message type have their own data layout/structure. The message will have the type as string, the number can be used internally by the GE implementation.

```
{
    // "State" channel
    "Registration"          : 1,
    "RendererStateChange"   : 2,

    // "Signaling" channel
    "Offer"                 : 10,
    "Answer"                : 11,
    "IceCandidates"         : 12,

    // "Room" channel
    "RoomAssigned"          : 20,
    "RoomUserJoined"        : 21,
    "RoomUserLeft"          : 22,

    // "Application" channel
    "RoomCustomMessage"     : 30,
    "PeerCustomMessage"     : 31
}
```

14.1.1.1 *State*

14.1.1.1.1 *Registration*

A renderer and a client needs to register itself to the Cloud Rendering Service, this is the first message that MUST be sent once the WebSocket connection has been established.

Renderer Registration

For the renderer registration implies that it is online and can be assigned to any room without a renderer by the service. The renderer will receive a RoomAssignedMessage response to a registration message.

The renderer may also choose to create a new private room. The intention here is that a renderer registers itself and wants to create a room, not be assigned as the renderer to any client room request. This allows the renderer to immediately get a new empty room and send the room information forward to clients it wants to invite to its rendering. This option is especially needed when a renderer (eg. web

browser renderer implementation) wants to share its web camera or desktop feed with specific individual clients. It is then needed for the renderer to notify this intent to the service so the scenario can be supported.

Renderers creating their own rooms is very important for certain type of application that do not want to provide renderers just for anyone to use, but for an approach when the implementation is more in control who gets the room information and can join into a specific renderers room.

The "createPrivateRoom" boolean property is defaulted to false if not defined. The renderer MUST set this property to true if it wants a private room assignment after the registration.

Client Registration

For a client it is a request for a new room or if roomId is a non-empty string a the client will be joined to an existing room. The response for the registration message is a RoomAssignedMessage. This message will have the error code 2 (DoesNotExist) set if the requested room did not exist.

Any custom registration properties needs to be set to message.data. These custom properties can be used by the Cloud Rendering GE implementation for example for authentication logic.

```
// Web Client -> Cloud Rendering Service
{
    "channel" : "State",
    "message" :
    {
        "type" : "Registration",
        "data" :
        {
            "registrant" : "client",
            "roomId"     : <optional-room-id-to-join-as-string>,

            // The remaining structure of the data is decided by the Cloud
Rendering GE
            // implementation. For example auth tokens to authenticate the
registrant.
            <application-specific-data>
        }
    }
}

// Renderer -> Cloud Rendering Service
{
```

```
    "channel" : "State",
    "message" :
    {
        "type" : "Registration",
        "data" :
        {
            "registrant"        : "renderer",
            "createPrivateRoom" : <boolean>,

            // The remaining structure of the data is decided by the Cloud
Rendering GE
            // implementation. For example auth tokens to authenticate the
registrant.
            <application-specific-data>
        }
    }
}
```

14.1.1.1.2 *RendererStateChange*

This message is sent from renderer to the cloud rendering web service to inform about state changes.

```
// Valid renderer states
{
    "Offline" : 1, // Renderer is offline, new clients should not be redirected
here.
    "Online"  : 2, // Renderer is online and ready to server new clients.
    "Full"    : 3  // Renderers client limit has been reached, no new clients
should be redirected here.
}

{
    "channel" : "State",
    "message" :
    {
        "type" : "RendererStateChange",
        "data" :
        {
            "state" : <state-id-as-number>
```

```
        }
    }
}
```

14.1.1.2 *Signaling*

14.1.1.2.1 *Offer*

The offer message is the first step in signaling a peer to peer WebRTC connection. It includes the offer senders SDP information and optionally ICE candidate(s) for the WebRTC connection setup. The appropriate response for this message is an Answer message.

ICE candidates may be empty, in that case they will be sent with separate IceCandidates message(s).

```
// Web Client and Renderer -> Cloud Rendering Service
{
    "channel" : "Signaling",
    "message" :
    {
        "type" : "Offer",
        "data" :
        {
            // If not defined when sender is a web client
            // "renderer" is assumed by the web service.
            "receiverId" : <receiver-peer-id-as-string>,

            "sdp" :
            {
                "type" : <sdp-type-as-string>,
                "sdp"  : <sdp-as-string>
            },
            "iceCandidates" :
            [
                {
                    "sdpMLineIndex" : <ice-m-line-index-as-number>,
                    "sdpMid"        : <ice-sdp-mid-as-string>,
                    "candidate"     : <ice-candidate-as-string>
                },
                ...
```

```
            ]
        }
    }
}

// Cloud Rendering Service -> Web Client and Renderer
{
    "channel" : "Signaling",
    "message" :
    {
        "type" : "Offer",
        "data" :
        {
            "receiverId" : <receiver-peer-id-as-string>,
            "senderId"   : <sender-peer-id-as-string>,

            "sdp" :
            {
                "type" : <sdp-type-as-string>,
                "sdp"  : <sdp-as-string>
            },
            "iceCandidates" :
            [
                {
                    "sdpMLineIndex" : <ice-m-line-index-as-number>,
                    "sdpMid"        : <ice-sdp-mid-as-string>,
                    "candidate"     : <ice-candidate-as-string>
                },
                ...
            ]
        }
    }
}
```

14.1.1.2.2 *Answer*

The answer message is the second step in signaling a peer to peer WebRTC connection. It includes the answer senders SDP information and optionally ICE candidate(s) for the WebRTC connection setup.

ICE candidates may be empty, in that case they will be sent with separate IceCandidates message(s).

```
// Web Client and Renderer -> Cloud Rendering Service
{
    "channel" : "Signaling",
    "message" :
    {
        "type" : "Answer",
        "data" :
        {
            // If not defined when sender is a web client
            // "renderer" is assumed by the web service.
            "receiverId" : <receiver-peer-id-as-string>,

            "sdp" :
            {
                "type" : <sdp-type-as-string>,
                "sdp"  : <sdp-as-string>
            },
            "iceCandidates" :
            [
                {
                    "sdpMLineIndex" : <ice-m-line-index-as-number>,
                    "sdpMid"        : <ice-sdp-mid-as-string>,
                    "candidate"     : <ice-candidate-as-string>
                },
                ...
            ]
        }
    }
}

// Cloud Rendering Service -> Web Client and Renderer
{
    "channel" : "Signaling",
    "message" :
```

```
    {
        "type" : "Answer",
        "data" :
        {
            "receiverId" : <receiver-peer-id-as-string>,
            "senderId"   : <sender-peer-id-as-string>,

            "sdp" :
            {
                "type" : <sdp-type-as-string>,
                "sdp"  : <sdp-as-string>
            },
            "iceCandidates" :
            [
                {
                    "sdpMLineIndex" : <ice-m-line-index-as-number>,
                    "sdpMid"        : <ice-sdp-mid-as-string>,
                    "candidate"     : <ice-candidate-as-string>
                },
                ...
            ]
        }
    }
}
```

14.1.1.2.3 *IceCandidates*

This message can be sent from peer to peer to inform the other party of ICE candidates for establishing the WebRTC connection

```
// Web Client and Renderer -> Cloud Rendering Service
{
    "channel" : "Signaling",
    "message" :
    {
        "type"   : "IceCandidates",
        "data"   :
        {
```

```
            "receiverId" : <receiver-peer-id-as-string>,

            "iceCandidates" :
            [
                {
                    "sdpMLineIndex" : <ice-m-line-index-as-number>,
                    "sdpMid"        : <ice-sdp-mid-as-string>,
                    "candidate"     : <ice-candidate-as-string>
                },
                ...
            ]
        }
    }
}

// Cloud Rendering Service -> Web Client and Renderer
{
    "channel" : "Signaling",
    "message" :
    {
        "type" : "IceCandidates",
        "data" :
        {
            "receiverId" : <receiver-peer-id-as-string>,
            "senderId"   : <sender-peer-id-as-string>,

            "iceCandidates" :
            [
                {
                    "sdpMLineIndex" : <ice-m-line-index-as-number>,
                    "sdpMid"        : <ice-sdp-mid-as-string>,
                    "candidate"     : <ice-candidate-as-string>
                },
                ...
            ]
        }
    }
```

```
}
```

14.1.1.3 *Room*

14.1.1.3.1 *RoomAssigned*

This message is sent as a response to Registration message.

The data contains the room id you have been assigned to and your unique peer id inside the room. If the error code is set to anything but 0 (NoError) the roomId and peerId properties SHOULD be omitted by the sender and SHOULD be ignored by the receiver if set to a value.

The error code will tell you why you could not be assigned to a room.

This message is also sent to the renderer once it is assigned to a room. The peer id CAN be omitted when the message is sent to a renderer.

This message is followed by a RoomUserJoined message that will have the full list of peer id:s currently in the channel. This set will also includes your own peer id that has been sent to you in this message.

```
// Valid room assignment errors
{
    "NoError"       : 0, // No errors, you have been assigned to "roomId".
    "ServiceError"  : 1, // Room could not be server, internal service error.
    "DoesNotExist"  : 2, // Requested room id does not exist. Send new request
without room id to create a new room.
    "Full"          : 3  // Requested room is full, try again later.
}


{
    "channel" : "Room",
    "message" :
    {
        "type" : "RoomAssigned",
        "data" :
        {
            "error"  : <error-as-number>,


            "roomId" : <room-id-as-string>,
            "peerId" : <your-peer-id-in-the-room-as-number>
        }
    }
```

```
}
```

14.1.1.3.2 *RoomUserJoined*

This message is sent when new user(s) join the current room you are assigned to.

This message MUST NOT be sent to any client or a renderer before they have been assigned to a room with a RoomAssigned message.

```
{
    "channel" : "Room",
    "message" :
    {
        "type" : "RoomUserJoined",
        "data" :
        {
            "peerIds" :
            [
                <peer-id-1-as-string>,
                <peer-id-2-as-string>,
                ...
            ]
        }
    }
}
```

14.1.1.3.3 *RoomUserLeft*

This message is sent when user(s) leave the current room you are assigned to.

This message MUST NOT be sent to any client or a renderer before they have been assigned to a room with a RoomAssigned message.

```
{
    "channel" : "Room",
    "message" :
    {
        "type" : "RoomUserLeft",
        "data" :
        {
            "peerIds" :
```

```
            [
                <peer-id-1-as-string>,
                <peer-id-2-as-string>,
                ...
            ]
        }
    }
}
```

14.1.1.4 *Application*

14.1.1.4.1 *RoomCustomMessage*

This message can be used by the application to send custom message between clients, the Cloud Rendering Service and the Renderer in the same room.

The Service and Renderer do not have a peer id assigned to them, when you want to send messages to these two components you will use reserved special meaning string identifiers instead. The reserver string identifiers are "service" and "renderer". These identifiers can be encountered in the 'sender' property and can be used in the 'receivers' property.

You can send a 1-to-1 message by assigning the other peers id to peerIds, or 1-to-many by assigning multiple ids.

For 1-to-all the peerIds list can be empty, if empty the message will be sent to all other peers and the Renderer in the room. You don't need to fill the peerIds list with all the peer ids for each client in the room, this is done automatically by the service if the list is empty.

Any custom message properties MAY be set to message.data.payload.

Custom messages MUST NOT be sent before you have been assigned to a room with a RoomAssigned message.

The Cloud Rendering Service MUST inject the 'sender' property into the message before passing it forward to the receivers. Meaning that the sender SHOULD NOT set the 'sender' property to and if it will the service MUST override it.

```
{
    "channel" : "Application",
    "message" :
    {
        "type" : "RoomCustomMessage",
        "data" :
        {
```

```
            // Service will inject this, does not needed to be filled
            // on the client or the renderer when sending the message.
            "sender" : <sender-peer-or-reserved-id-as-string>,

            "receivers" :
            [
                <reserved-string-identifier-1>,
                ...
                <receiver-peer-1-id-as-string>,
                <receiver-peer-2-id-as-string>,
                ...
            ],

            "payload" :
            {
                // The structure of the data is decided by the application and
the web client implementation.
                <application-specific-data>
            }
        }
    }
}

// Example 1: Client sends to all peers and the renderer in the room.
{
    "channel" : "Application",
    "message" :
    {
        "type" : "RoomCustomMessage",
        "data" :
        {
            "payload" : { "nick" : "John Doe" }
        }
    }
}

// Example 2: Client sends to message to a single peer.
```

```
{
    "channel" : "Application",
    "message" :
    {
        "type" : "RoomCustomMessage",
        "data" :
        {
            "receivers"  : [ "2" ]
            "payload"    :
            {
                "type"    : "PrivateMessage",
                "message" : "Hey, long time no see. Whats up?"
            }
        }
    }
}

// Example 3: Client sends message to peers 3, 5 and the renderer
{
    "channel" : "Application",
    "message" :
    {
        "type" : "RoomCustomMessage",
        "data" :
        {
            "receivers"  : [ "renderer", "3", "5" ],
            "payload"    :
            {
                "message" : "Hello my name is John",
                "nick"    : "John Doe"
            }
        }
    }
}

// Example 4: Renderer sending a message to the Service
{
```

```
    "channel" : "Application",
    "message" :
    {
        "type" : "RoomCustomMessage",
        "data" :
        {
            "receivers"  : [ "service" ],
            "payload"    :
            {
                "type"   : "KickUser",
                "peerId" : 2
            }
        }
    }
}
```

14.1.1.4.2 *PeerCustomMessage*

This message can be used by the application to send custom message between any two peers via a WebRTC data channel. This message does not contain sender or receiver id information because it is never relayed via the web service. The needed sender information is known from the WebRTC protocol by the application handling the messages.

These data channel messages should be used when there is no need for one-to-many communication and when you require real time/very frequent communication. The data channel skips the overhead and latency that is added by relaying the message via the web service with WebSocket.

Any custom message properties MAY be set to message.data.payload.

```
{
    "channel" : "Application",
    "message" :
    {
        "type" : "PeerCustomMessage",
        "data" :
        {
            "payload" :
            {
```

```
                // The structure of the data is decided by the application and
the web client implementation.
                <application-specific-data>
            }
        }
    }
}
```

15 FIWARE OpenSpecification MiWi GISDataProvider

Name	FIWARE.OpenSpecification.MiWi.GISDataProvider
Chapter	Advanced Middleware and Web-based UI,
Catalogue-Link to Implementation	GIS data provider
Owner	Cyberlightning Ltd., Sami Jylkkä

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.org and similar pages in order to understand the complete context of the FI-WARE project.

15.2 Copyright

- Copyright © 2013 by Cyberlightning Ltd.

15.3 Legal Notice

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

15.4 Overview

"3D GIS (Geographic Information System) data provider" is a GE, which is able to serve both outdoor and indoor 3D data. The data shall be served roughly speaking in two fashions:

1. For single user apps, which can make queries to GIS backend, to download specific a single object, or a set of geolocated 3D objects (i.e. building and terrain model of a named city centre)

2. For multiuser apps, which communicate via collaborative client platform (realXTend) and in which case the realXtend is providing the 3D models for the client application.

The GIS GE itself will be build of a server, which is able to process geolocated queries, an extension to it which is able to manage 3D data and a backend database where all geolocated data is saved. Because the GIS GE needs to serve both single user client apps (browsers) and multiuser enabled virtual spaces, an API will be defined to serve fluently both of the cases. In this API is a RESTful one, and allows both the browser app as well as realXtend virtual world platform to use it for downloading geolocated data.

15.4.1 The components used in this GE are:

- PostgreSQL & PostGIS
- GeoServer & W3DS community module for GeoServer
- GeoTools
- XML3D

15.4.1.1 *PostgreSQL & PostGIS*

Postgresql is central database for storing GIS related data. PostGIS is a spatial database extender for PostgreSQL object-relational database. It adds support for geographic objects allowing location queries to be run in SQL.

15.4.1.2 *GeoServer & W3DS community module*

GeoServer is an open source software server written in Java that allows users to share and edit geospatial data. GeoServer is the reference implementation of the Open Geospatial Consortium (OGC) Web Feature Service (WFS) and Web Coverage Service (WCS) standards, as well as a high performance certified compliant Web Map Service (WMS). GeoServer forms a core component of the Geospatial Web.

W3DS (Web 3D Service) community module is extension for GeoServer. Module purpose is to provide W3DS functionality defined in "OpenGIS® Web 3D Service Interface Standard 0.4.0" -document. W3DS module functionality has been extended to support XML3D and octet-stream output formats.

GeoServer uses GeoTools for accessing GIS database.

15.4.1.3 *GeoTools*

GeoTools handles reading GIS data from PostGIS database. There is already support for reading 3D GIS content data from database.

15.4.1.4 *XML3D*

XML3D is used for rendering 3D objects in web browser.

15.4.2 The project develops

- Documentation how to set up 3D GIS data provision environment
    - Guidance how to import 3D GIS data to PostGIS
    - How to properly configure GeoServer
    - How client application connects to GeoServer and can acquire 3D GIS data
- Modified GeoServer W3DS module with XML3D and octet-stream output formats are verified
- Implementation of the example web client which acquires 3D GIS data.

15.4.3 Target Usage

Target audience is entities who want to present their own 3D GIS data in web browser. Entities can be companies or individual persons. This GE presents whole visual pipeline how this can be achieved. Guides are provided how 3D GIS data can be created & stored to database and how visual pipeline is configured so that individual services runs properly.

15.4.4 Example Scenarios

The following **single-user** scenario describes a workflow where 3D GIS data is shown in the web browser.

- The client (web browser) sends a RESTful query to the GeoServer (GIS Data Provider GE) to get 3D GIS data to be shown in the web browser. Needed parameters are sent, e.g. bounding box is defined to specify what geocoordinates the data client wants to display.
- The GeoServer handles the client query and fetches 3D objects from the database (PostGIS) based on the bounding box area. There is 2 options to get a 3D object:
    - 3D object vertex definitions can be stored directly in the database or
    - The database contains external references to where the actual 3D model is stored.
- Either the 3D object definition or the reference to the definition file is returned to the GeoServer.
- When the database query is done, the GeoServer will create an XML3D scene or a single XML3D object based on the information it got from PostGIS.
    - Required coordinate transformations are done in the server and the generated XML3D scene or object is sent back to the client (web browser).
- The client receives the XML3D scene/object and displays it using XML3D rendering.
    - In case a single XML3D object is received, it can be directly injected into an existing web page.

In the **multi-user** scenario multiple users can be located in the same virtual world via avatars. A users's avatar location is sent to the frontend server (Synchronization GE), where it is synchronized to other clients. Following an example workflow of such a multi-user scenario:

- A user opens the virtual-world client (3D UI GE), which then connects to the GIS database to get 3D models.
    - The virtual-world client (3D UI GE) gets the end users location in the 3D world.
    - The user,s location is sent to the frontend server (Synchronization GE), which is responsible to keep all persons locations in sync in their clients.
- The virtual world server creates a 3D virtual world with data received from the GIS database.

- When the user moves in the virtual world, the users avatar location is updated in the frontend server, which then passes changed location to the other users clients.

15.4.5 Connections to Results of Other MiWi Projects

- FIWARE.OpenSpecification.MiWi.POIDataProvider
    - POI data provider is capable to store 3D object reference information. GIS client can utilize this information to acquire externally stored 3D models and attach them to correct location in 3D GIS presentation. Also other relevant metadata can be read from POI and inserted to 3D GIS presentation.

15.5 Basic Concepts

- 3D object rendering to web browser with XML3D engine.
- Injection of XML3D object to existing web page.
- Storing 3D data definition to PostGIS and linking it with spatial information. This data can be included to 3D presentation with correct spatial information.
- Using references to external locations to get 3D model definition file and using it.
- Acquiring 3D GIS data only need basis in order to save network bandwidth. When GIS data is queried from database bounding box is given as one parameter. This defines area where stored GIS objects searching is narrowed.

15.5.1 Generic Architecture

Figure GDP-1 presents generic architecture for GIS Data Provider GE. Main interactions between components are displayed in the picture. *Frontend server* is not part of the GIS GE delivery. *Frontend server* can be optionally used when multiple different GIS data sources are used in order to achieve simplified service API for the client.

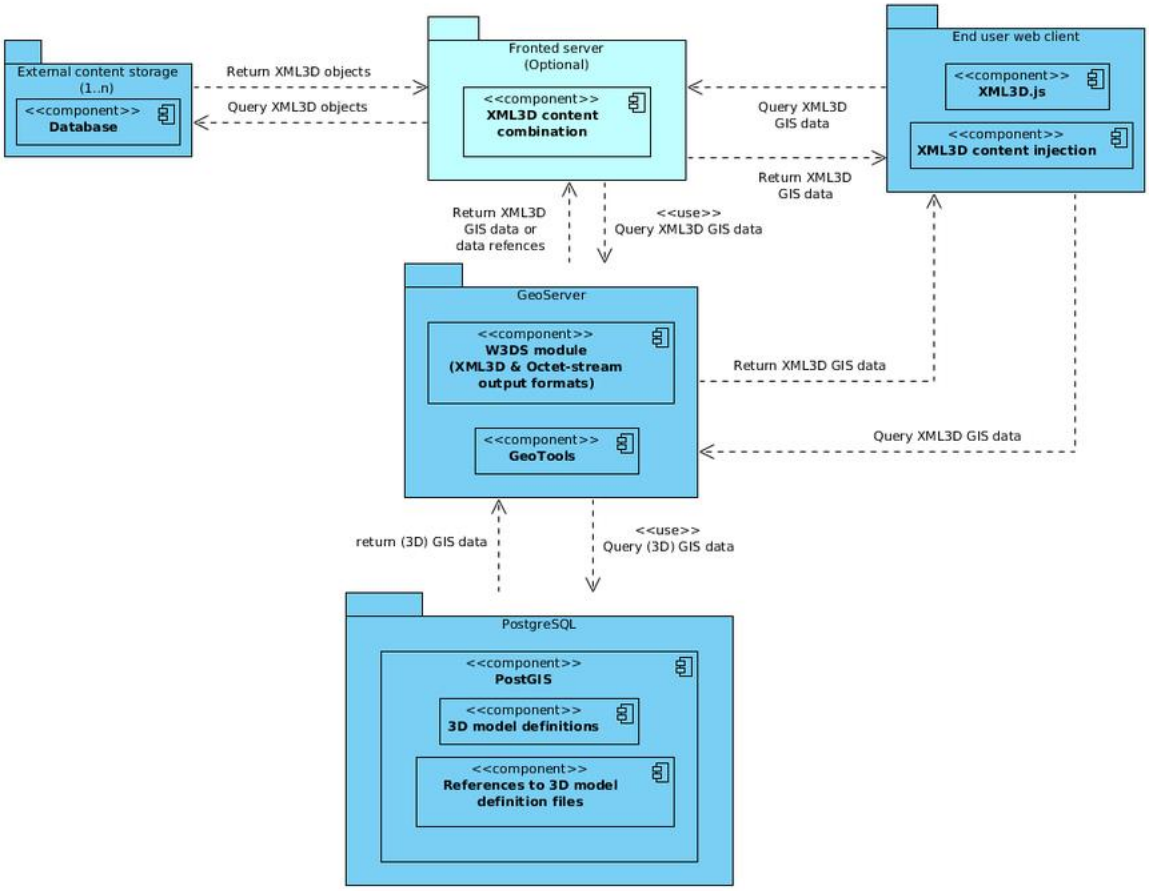


Figure GDP-1: Generic architecture of GIS Data provider GE

15.6 Main Interactions

15.6.1 Modules and Interfaces

Figure GDP-2 presents basic flow how client queries 3D GIS data and displays it in the browser. Picture shows case when 3D object vertices is directly read from database. Communication entities are:

- GIS client
- XML3D
- GeoServer
- W3DS module
- GeoTools
- PostgreSQL

GIS client is responsible to create html web page which contains XML3D content. Flow starts from the client (end user) web browser, which sends query for displaying 3D GIS data inside defined bounding box. GeoServer receives query and responds with GeoServer W3DS capabilities. GIS client parses W3DS

capability information and creates GetScene query based on the information. GetScene query is sent to GeoServer. Query is handled by W3DS module, which fetches 3D GIS data from PostgreSQL database and creates XML3D content based on the fetched data. End result is generated XML3D scene which is sent back to the GIS client where XML3D engine handles rendering of the XML3D content.

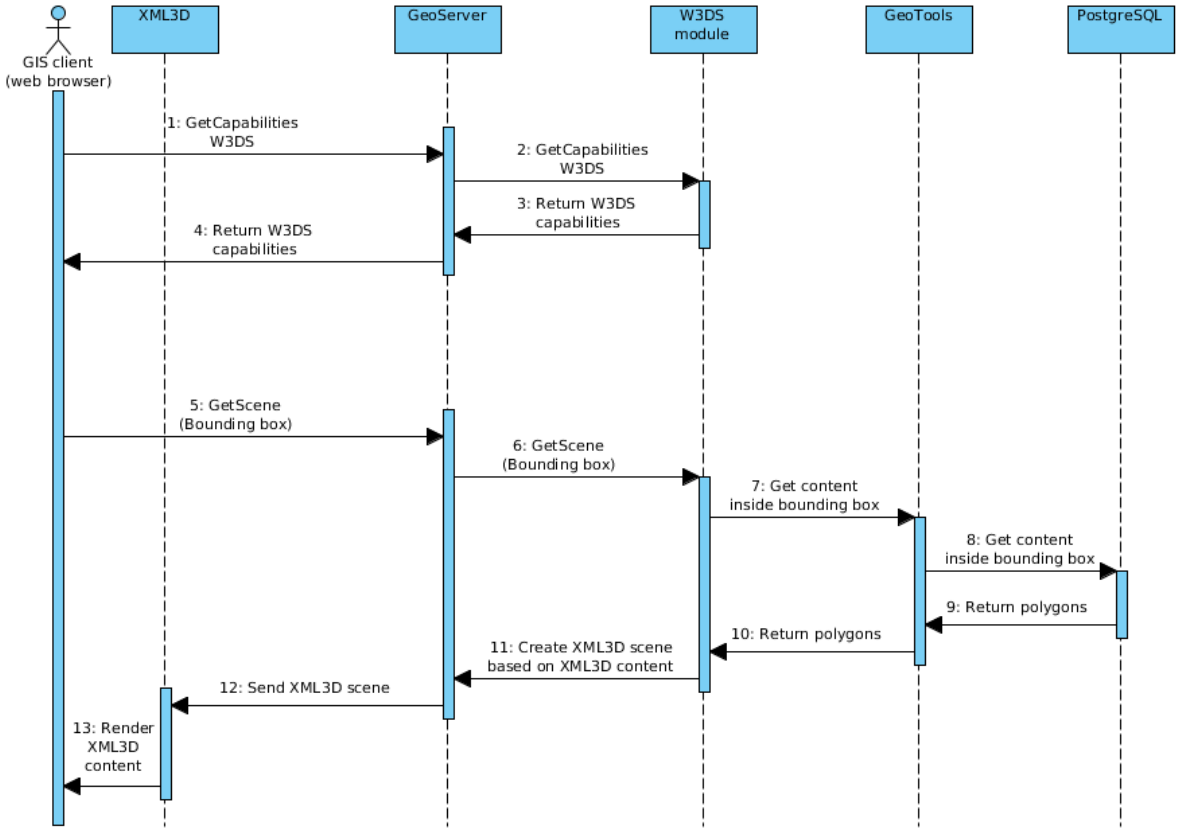


Figure GDP-2 - Sequence diagram for XML3D data fetching

15.7 Basic Design Principles

Basic design principle is to keep existing components untouched as much it is possible. Existing basic functionality in GeoServer and PostgreSQL is in the state that it can be used as basis for development. Target is to leave these components functionality as it currently is.

15.7.1 XML3D & octet-stream output formats support in W3DS module

What needs to be done is support for XML3D output format in W3DS module in GeoServer. Upon this client server needs to be implemented which handles communication with GeoServer and combines references from different data sources. e.g. XML3D content definition files can be located in different storage's, server will pull these and combine them as final output.

Alternative format to request 3D GIS data from the GeoServer is octet-stream. Data in the octet-stream is stored to the byte array, only elevation data is included to the well defined grid form.

15.7.2 DOM manipulation

Demonstration level DOM manipulation needs to be implemented for sake of able to create working demo of 3D GIS data provider. This enables dynamic web view updating, so that new XML3D objects can be displayed based on the location and camera angle information. DOM manipulation is done in client web browser, data queries are send to client web server which is responsible to do actual 3D GIS data fetching from data storages and deliver XML3D objects to client web browser.

15.8 Detailed Specifications

Following is a list of Open Specifications linked to this Generic Enabler. Specifications labeled as "PRELIMINARY" are considered stable but subject to minor changes derived from lessons learned during last interactions of the development of a first reference implementation planned for the current Major Release of FI-WARE. Specifications labeled as "DRAFT" are planned for future Major Releases of FI-WARE but they are provided for the sake of future users.

15.8.1 Open API Specifications

The API Specification is still under development and will be released close to the first software release. The following is an early draft of the APIs, subject to change.

- GIS Data Provider Open API Specification

15.9 Re-utilised Technologies/Specifications

1. W3DS specification draft The draft candidate specification for OpenGIS® Web 3D Service
2. PostgreSQL PostgreSQL is an open source object-relational database system.
3. PostGIS Spatial and Geographic objects for PostgreSQL.
4. GeoServer GeoServer is an open source software server written in Java that allows users to share and edit geospatial data.
5. GeoTools GeoTools is an open source Java library that provides tools for geospatial data.

15.10 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

Annotations

Annotations refer to non-functional descriptions that are added to declaration of native types, to IDL interface definition, or through global annotations at deployment time. The can be used to express security requirements (e.g. "this string is a password and should be handled according the

security policy defined for password"), QoS parameters (e.g. max. latency), or others.

AR

AR → Augmented Reality

Augmented Reality (AR)

Augmented Reality (AR) refers to the real-time enhancement of images of the real world with additional information. This can reach from the rough placement of 2D labels in the image to the perfectly registered display of virtual objects in a scene that are photo-realistically rendered in the context of the real scene (e.g. with respect to lighting and camera noise).

IDL

IDL → Interface Definition Language

Interface Definition Language

Interface Definition Language refers to the specification of interfaces or services. They contain the description of types and function interfaces that use these types for input and output parameters as well as return types. Different types of IDL are being used including CORBA IDL, Thrift IDL, Web Service Description Language (WSDL, for Web Services using SOAP), Web Application Description Language (WADL, for RESTful services), and others.

Middleware

Middleware is a software library that (ideally) handles all network related functionality for an application. This includes the setup of connection between peers, transformation of user data into a common network format and back, handling of security and QoS requirements.

PoI

PoI → Point of Interest

Point of Interest (PoI)

Point of Interest refers to the description of a certain point or 2D/3D region in space. It defines its location, attaches meta data to it, and defines a coordinate system relative to which additional coordinate systems, AR marker, or 3D objects can be placed.

Quality of Service (QoS)

Quality of Service refers to property of a communication channel that are non-functional, such a robustness, guaranteed bandwidth, maximum latency, jitter, and many more.

Real-Virtual Interaction

Real-Virtual Interaction refers to Augmented Reality setup that additionally allow users to interact with real-world objects through virtual proxies in the scene that monitor and visualize the state in the real-world and that can use services to change the state of the real world (e.g. switch lights on an off via a virtual button the 3D scene).

Scene

A *Scene* refers to a collection of objects, which are be identified by type (eg. a 3D mesh object, a

physics simulation rigid body, or a script object.) These objects contain typed and named data values (composed of basic types such as integers, floating point numbers and strings) which are referred to as attributes. Scene objects can form a hierarchic (parent-child) structure. A HTML DOM document is one way to represent and store a scene.

Security

Security is a property of an IT system that ensures confidentiality, integrity, and availability of data within the system or during communication over networks. In the context of middleware it refers to the ability of the middleware to guarantee such properties for the communication channel according to suitably expressed requirements needed and guarantees offer by an application.

Security Policy

Security Policy refers to rules that need to be fulfilled before a network connection is established or for data to be transferred. It can for example express statements about the identity of communication partners, properties assigned to them, the confidentiality measures to be applied to data elements of a communication channel, and others.

Synchronization

Synchronization is the act of transmitting over a network protocol the changes in a scene to participants so that they share a common, real-time perception of the scene. This is crucial to eg. implementing multi-user virtual worlds.

Type Description

Type Description in the context of the AMi middleware refers to the internal description of native data types or the interfaces described by an IDL. It contains data such as the name of a veriable, its data type, the hierarchical relations between types (e.g. structs and arrays), its memory offset and alignment within another data type, and others. Type Description are used to generate the mapping of native data types to the data that needs to be transmitted by the middleware.

Virtual Character

Virtual Character is a 3D object, typically composed of triangle mesh geometry, that can be moved and animated and can represent an user's presence (avatar) in a virtual world. Typically supported forms of animation include skeletal animation (where a hierarchy of "bones" or "joints" controls the deformation of the triangle mesh object) and vertex morph animation, where the vertices of the triangle mesh are directly manipulated. Virtual character systems may support composing the character from several mesh parts, for example separate upper body, lower body and head parts, to allow better customization possibilities.

16 GIS Data Provider Open API Specification

16.1 Introduction to the GIS Data Provider API

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

Interactive, multi-user Web applications such as virtual worlds often have the common requirements of retriveving environmental components for the graphical reproduction. Often this data is related to the geographical information, such as terrain soil, terrain texturing, water areas and such. GIS Data Provider acts as a service which is capable in serving such components via restful API.

16.1.1 GIS Data Provider API Core

The GIS Data Provider API is a RESTful, resource-oriented API accessed via HTTP that uses XML and octet-stream -based representations for information interchange. It is used to query the data contained in GIS Data Provider. The GIS data is organized according to its geographical coordinates which are used to sort the data spatially for the client queries.

16.1.2 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 using the supported operations. For the latter, this specification provides a full specification of how to implement those operations.

16.1.3 API Change History

The GIS Data Provider API version history is described in the table below:

Revision Date	Changes Summary
Feb 13, 2014	• Initial version

16.1.4 How to Read This Document

All FI-WARE RESTful API specifications will follow the same list of conventions and will support certain common aspects. Please check Common aspects in FI-WARE Open Restful API Specifications.

For a description of some terms used along this document, see the Terns and Definitions.

16.1.5 Additional Resources

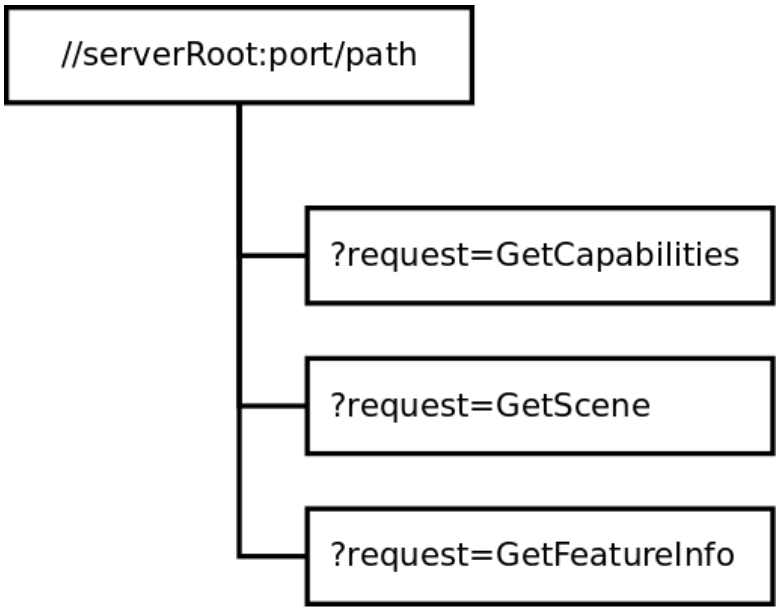
Here you can describe any other information or resources that you could need to understand this document. It could include also the link <url> in which to can obtain this specification and the link to obtain the schemas that we are using. E.g.

"You can download the most current version of this document from the FIWARE API specification website at <link to the url>. For more details about the <name of the GE service> that this API is based upon, please refer to <link to the High Level Description>. Related documents, including an Architectural Description, are available at the same site."

16.2 General GIS Data Provider API Information

16.2.1 Resources Summary

The following diagram describes the resources that can be accessed with the GIS Data Provider API, starting from the server base URL.



GIS Data Provider API resource summary

16.2.2 Representation Format

The GIS Data Provider API supports data in XML3D and octet-stream formats by giving format argument in the request.

- XML3D
    - Request returns group of XML3D 3d-models and those can directly be added to DOM tree
    - All coordinate values inside XML3D object are relative values to requested bounding box.

```
An example of XML3D output format request

http://hostname:port/path?SERVICE=W3DS&REQUEST=GetScene&VERSION=0.4.0&C
```

RS=EPSG:26916&FORMAT=model/xml3d+xml&BoundingBox=202759.0,3310170.0,213200.0,3320896.0&LAYERS=Terrain

- Octet-stream
    - Request returns byte array in the following format:
        - First value in the array is 32bit integer with information how many points are in X-axis
        - Second value in the array is 32bit integer with information how many points are in Y-axis
        - Third value in the array is 32bit float with information about average distance between points in X-axis
        - Fourth value in the array is 32bit float with information about average distance between points in Y-axis
    - After those initialization values there are *First* * *Second* number of 32bit floats containing elevation information for each point in that grid.
        - First elevation value is from top right corner and last value is bottom left corner.
        - Values are read from a grid one row at a time, reading of each row is started from right to left.

```
An example of decoded octet-stream output:

   5 5 9.8 9.8 123.1 115.1 116.7 119.0 123.1 115.1 116.7 119.8 123.1
120.1 115.1 118.4 119.0 123.1 123.1 123.1 123.1 123.1 123.1 123.1 123.1
123.1 123.1 123.1 123.1

An example of octet-stream output format request


http://hostname:port/path?SERVICE=W3DS&REQUEST=GetScene&VERSION=0.4.0&CRS=EPSG:26916&FORMAT=application/octet-stream&BoundingBox=202759.0,3310170.0,213200.0,3320896.0&LAYERS=Terrain
```

16.2.3 Resource Identification

All operations (GET, PUT, POST, DELETE) to the GIS DataProvider API require the target resource to be identified. For HTTP transport, this is made using the mechanisms described by HTTP protocol specification as defined by IETF RFC-2616.

16.2.4 Links and References

N/A.

16.2.5 Limits

N/A.

16.2.6 Versions

N/A.

16.2.7 Extensions

N/A.

16.2.8 Faults

16.2.8.1 *Synchronous Faults*

Fault Element	Associated Error Codes	Expected in All Requests?
notFound	404	The resource was not found for a query operation.
badRequest	400	The resource identifier was invalid for a modification operation.

When a fault happens, it is also transmitted in the reply body as text for convenience.

16.2.8.2 *Asynchronous Faults*

All current GIS Data Provider API operations are synchronous in nature; therefore there are no asynchronous faults.

16.3 API Operations

16.3.1 Capability Query

Verb	URI	Description
GET	/path?SERVICE=W3DS&ACCEPTVERSIONS=0.3.0,0.4.0&request=GetCapabilities	GetCapabilities operation allows clients to retrieve service metadata from a server. The response to a GetCapabilities request shall be an XML document containing service metadata about the server, including specific information about layer properties and how to access data from the server.

Normal Response Code(s): 200

Error Response Code(s): 404

16.3.2 Scene Query

Verb	URI	Description
GET	/path?SERVICE=W3DS&REQUEST=GetScene&VERSION=0.4.0&CRS=EPSG:26916&FORMAT=application/xml3d&BoundingBox=202759.0,3310170.0,213200.0,3320896.0&LAY	The GetScene operation returns a 3D scene representing a subset of the natural or man made structures on the earth surface. Upon receiving a GetScene request, a W3DS shall either satisfy the request or issue a service exception. The required parameters for retrieving a 3D scene from a W3DS comprise

	ERS=Terrain&STYLES=aerial_view_RGB	spatial, thematic, and other constraints. The minimum set of parameters include, in addition to the mandatory parameters service, request and version, which are part of every W3DS operation, the spatial extent of the scene given as bounding box, the CRS in which the scene shall be provided, the format, and the list of layers. The bounding box defines a rectangular region with edges perpendicular to the selected CRS.

Normal Response Code(s): 200

Error Response Code(s): 404

16.3.3 Feature Info Query

Verb	URI	Description
GET	/path?SERVICE=W3DS&REQUEST=GetFeatureInfo&VERSION=0.4.0&CRS=EPSG:26916&FORMAT=text/xml&LAYERS=bldgs&FEATURECOUNT=5&COORDINATE=202042.233,3310094.983,334.0&EXCEPTIONS=text/html	The GetFeatureInfo operation is designed to provide clients with additional attribute information about features within a scene that is currently displayed. Example use case for GetFeatureInfo is that a user explores the response of a GetScene request and points at an object within the scene for which to obtain more information. The concept of this operation is that the client determines a location in 3D space by clicking on an object and calculates either the intersection point of the object geometry with the picking ray or the center point of the object and submits this location together with additional parameters to the server.

Normal Response Code(s): 200

Error Response Code(s): 404

17 FIWARE OpenSpecification MiWi POIDataProvider

Name	FIWARE.OpenSpecification.MiWi.POIDataProvider
Chapter	Advanced Middleware and Web-based UI,
Catalogue-Link to Implementation	POI Data Provider
Owner	University of Oulu, Ari Okkonen

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.org and similar pages in order to understand the complete context of the FI-WARE project.

17.2 Copyright

- Copyright © 2013, 2014 by University of Oulu

17.3 Legal Notice

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

17.4 Overview

This project targets to define Points of Interest Data Provider (POI-DP) and demonstrate its feasibility with a proof-of-concept implementation.

POI-DP is a generic enabler, which provides spatial search services and data on Points of Interest via RESTful web service API. The data model for representing POIs is modular and based on the entity-component model. In the proposed model, each POI is an entity, which is identified by a universally unique identifier (UUID). The data related to an entity is stored in different data components, which are linked to the entity using the UUID. These data components are independent of each other and every POI entity can have a different set of components. This enables the concept to serve a wide range of interest groups, as the data model can easily be extended by application specific data components. POI data can be distributed and different data components can be administered by different organizations using different policies.

The project develops

- Specification of POI-DP API

- A proof-of-concept implementation of distributed POI-DP network,
- An example of application area specific POI-DP,
- A proof-of-concept web application for POI functionality demonstration with 3D content,
- guidelines to design and implement application area specific POI-DPs, and
- guidelines for specifications of application area specific POI-DP APIs.

Applicable parts or entirety of specifications will be contributed to standardization e.g. by OGC.

17.4.1 Target Usage

POI data can is useful in wide variety of applications. The most apparent applications are connected to tourism and finding something interesting in new places in general. However, generic POI data associated with application area specific data can be very useful in different applications. Examples:

- Detailed harbor information in yachting
- Detailed description of underground piping and cables to help avoiding them in construction work
- Information of geographically distributed devices requiring maintenance
- Hotels, festivals, fast food places, emergency hospitals, whatever information a tourist needs

17.4.2 Example Scenario

Use case: A skipper of a yacht is approaching Kemi. He wants to find detailed information of the guest harbor, find a dining place and a hotel near to harbor.
The following subsystems take part in this scenario:

1. a mobile device to query the data
2. a nautical application in the mobile device - The application can show nautically interesting POIs on a nautical chart.
3. a tourist application in the mobile device - The application can show tourism related POIs on an ordinary map.
4. a nautical server - The server does queries to a generic POI server for basic POI data, filters it for nautically interesting POIs, and appends POI information with nautically interesting details: e.g. depth of the harbor and special harbor services.
5. a tourist server - The server does queries to a generic POI server for basic POI data and appends POI information with data relevant to tourists: e.g. classification (stars) of the hotels, class of dining places (Chaîne des Rôtisseurs, Michelin stars, etc.)...
6. generic POI servers - A generic POI server can contain a huge amount of POIs. It is specialized in spatial searches.

Different data components of a POI may be stored in different databases in different servers managed by different organizations. These data components are associated to the POI by an UUID that identifies the POI.

Flow of operation:

- First, the skipper uses a nautical application that shows nautical POIs on the navigation chart.
- The nautical application send a query to a nautical POI server that in turn sends a query to a generic POI server.
- The nautical POI server uses UUIDs of POIs to search its nautical database for extra information, and responds to the application with POIs augmented with nautical data.
- The skipper selects the POI corresponding to the harbor, reads the extra information and drags and drops the POI to a tourist application.
- The tourist application gets the UUID of the POI and focuses to the area of interest - harbor of Kemi. The tourist application sends a query to a tourism related server that in turn queries a generic POI server for nearby hotels and dining places.
- The tourism server adds proper tourist information from its database to the response to tourism application.
- The skipper can now browse more detailed data about the interesting services.

17.4.3 Connections to Results of Other MiWi Projects

Augmented Reality

POI-DP acts as the main data source for the Augmented Reality (AR) GE reference implementation. Different information about POIs can be visualized in the AR view, such as basic information about a POI or a 3D model representing the POI.

Real Virtual Interaction

For those applications that interact with "Internet of Things", a real-virtual extension is provided. A POI identified by its UUID is associated to a real device either by the real-virtual POI extension server or by the real-virtual interaction server. The servers cooperate to provide a connection from the application to the device. The details of cooperation will be negotiated during the project. This feature facilitates control of actuators e.g. directing a camera or adjusting a thermostat, and reading sensors e.g. a thermometer or a sea level gauge.

17.5 Basic Concepts

- Modular POI data model (based on the principles of the entity-component model)
    - Easy to extend by introducing new data components
    - Easy to distribute as data components are independent

- POIs relative to another e.g. restaurants within a cruise ship
- cases where lat/long coordinates are not feasible
- 3D data associated to POI
- Hierarchy of POIs

- RESTful web service API for querying and modifying POI data
  - Spatially limited searches
  - Searches constrained to specific categories e.g. dining, golf course, cemetery, dentist, casino, museum, etc.
  - Temporal constraints, e.g. for finding restaurants that are currently open
  - POI data modification functionality (add, update, delete)

17.5.1 POI Data Network Generic Architecture

A RESTful service architecture is defined for hosting and provisioning the POI data. The services provide different functionality for querying, accessing and modifying the POI data. The services are invoked via RESTful APIs using the standard HTTP protocol. As the POI data can be easily distributed due to its modular structure, the natural outcome is that the service architecture can be distributed as well. Different POI service backends can host a different set of POI data components.

The distributed service architecture also enables implementing service composition, i.e. service backends that do not necessarily host any POI data themselves, but instead fetch the POI data from a set of other backends

The POI data can be searched and accessed via RESTful web service APIs. The POI access API enables accessing specific data components for POIs based on the UUIDs of the POI entities. The POI search API provides different spatial search functions for a client to find relevant POIs in a certain location.

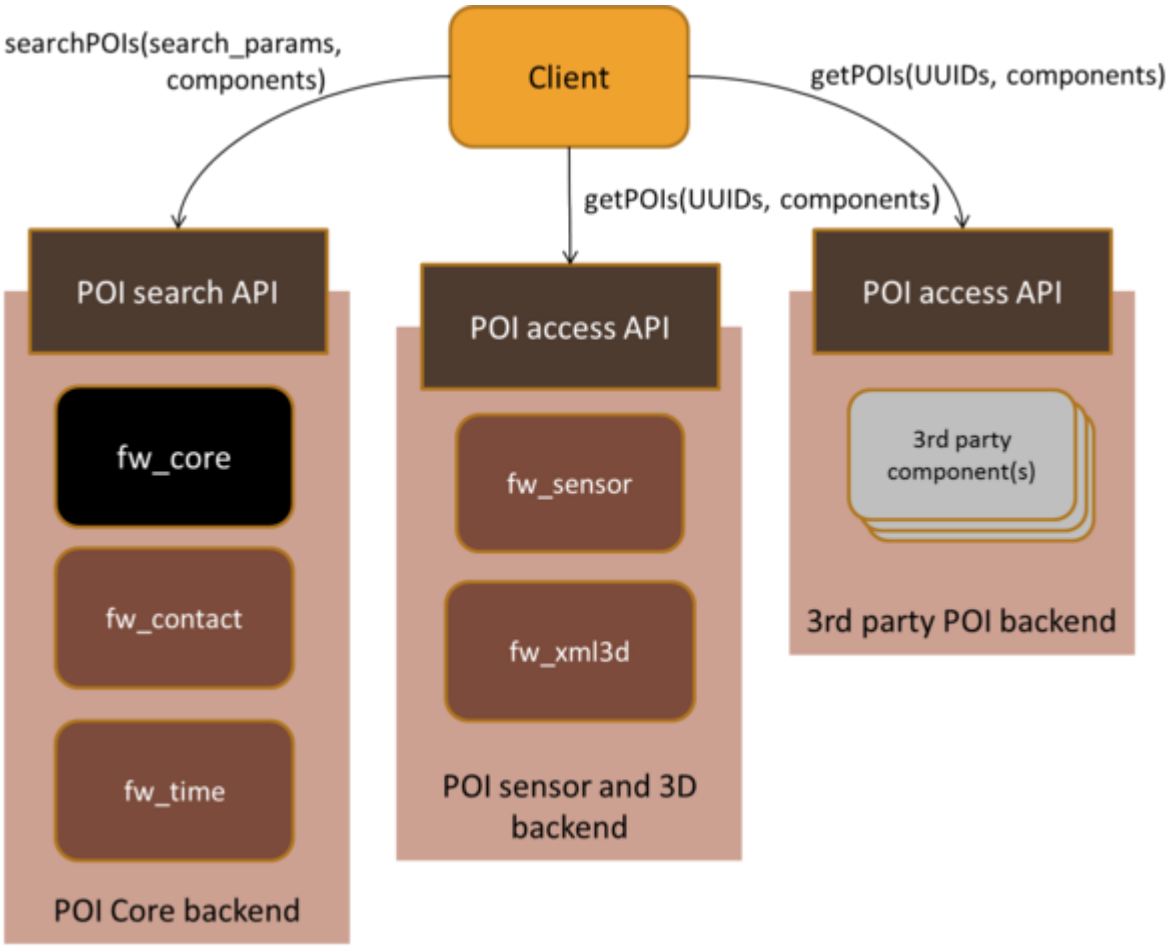


Figure POI-DP-1: Overview of the POI-DP service architecture

17.6 Main Interactions

The POI data provider is deployed as a RESTful web service. It has an easy to use API that can be accessed with regular HTTP requests.

17.6.1 Querying POIs

POI data can be queried in different ways. The goal is to support a variety of different use cases utilizing the POI data. The query operations are accessed with standard HTTP GET requests.

17.6.1.1 *Radial search*

The most common use case is to search for POIs near a certain location. For this, the service provides a spatial search functionality, namely radial search. It can be used to find POIs around a certain location within a given distance, i.e. circle radius. The location is given as latitude and longitude in degrees, and the radius in meters.

17.6.1.2 *Bounding Box search*

Bounding Box search enables searching POIs inside a given bounding box, i.e. a rectangular geographical area.

17.6.1.3 *Access POIs by UUID*

In another use case, the client already knows the UUIDs of the POIs it is interested in. In this case, the POI data can be accessed directly with the POI UUIDs. Several UUIDs can be given in a single query.

17.6.2 Modifying POIs

17.6.2.1 *Add POI*

New POIs can be added to the database. The POI content is sent by the client using HTTP POST request.

17.6.2.2 *Update POI*

Existing POI data can be updated. The updated POI content is sent by the client with HTTP POST request.

17.6.2.3 *Delete POI*

POIs can also be removed from the database. The client can remove existing POIs using HTTP DELETE request.

17.7 Basic Design Principles

- The core POI data must be compact, so that it does not contain much such information that is not needed in most applications. This is to keep data bases small and fast, and to speed up responses to queries.
- It must be easy both technically and organizationally for an application area or an organization to start utilizing the POI data and to append needed extensions to data. Generic software base, extension templates, and well documented extension guides must be provided for this end.
- An organization must be able to administer data in its interests. E.g. a hotel chain must be able to maintain descriptions of its hotels without cooperation with other organizations.
- Cross usage between generic and application area specific clients and servers must be possible without error conditions. Of course some data will be lost or missing in transactions.
- Empty data fields are not transmitted.
- The client can limit the query:
    - maximum POIs returned
    - list of wanted data fields
    - language selection for applicable text fields
    - temporally: e.g. return only restaurants that are currently open

17.8 Detailed Specifications

Following is a list of Open Specifications linked to this Generic Enabler. Specifications labeled as "PRELIMINARY" are considered stable but subject to minor changes derived from lessons learned during last interactions of the development of a first reference implementation planned for the current Major Release of FI-WARE. Specifications labeled as "DRAFT" are planned for future Major Releases of FI-WARE but they are provided for the sake of future users.

17.8.1 Open API Specifications

- POI Data Provider Open API Specification

17.9 Re-utilised Technologies/Specifications

1. PostgreSQL PostgreSQL is an open source object-relational database system.
2. PostGIS Spatial and Geographic objects for PostgreSQL.
3. MongoDB MongoDB is an open-source document database with JSON-style documents.
4. W3C POI Core Points of Interest Core, W3C Working Draft.
5. OGC POI OGC Points of Interest Encoding Specification.

17.10 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

Annotations

Annotations refer to non-functional descriptions that are added to declaration of native types, to IDL interface definition, or through global annotations at deployment time. The can be used to express security requirements (e.g. "this string is a password and should be handled according the security policy defined for password"), QoS parameters (e.g. max. latency), or others.

AR

AR → Augmented Reality

Augmented Reality (AR)

Augmented Reality (AR) refers to the real-time enhancement of images of the real world with additional information. This can reach from the rough placement of 2D labels in the image to the perfectly registered display of virtual objects in a scene that are photo-realistically rendered in the context of the real scene (e.g. with respect to lighting and camera noise).

IDL

IDL → Interface Definition Language

Interface Definition Language

Interface Definition Language refers to the specification of interfaces or services. They contain the description of types and function interfaces that use these types for input and output parameters as well as return types. Different types of IDL are being used including CORBA IDL, Thrift IDL, Web Service Description Language (WSDL, for Web Services using SOAP), Web Application Description Language (WADL, for RESTful services), and others.

Middleware

Middleware is a software library that (ideally) handles all network related functionality for an application. This includes the setup of connection between peers, transformation of user data into a common network format and back, handling of security and QoS requirements.

PoI

PoI → Point of Interest

Point of Interest (PoI)

Point of Interest refers to the description of a certain point or 2D/3D region in space. It defines its location, attaches meta data to it, and defines a coordinate system relative to which additional coordinate systems, AR marker, or 3D objects can be placed.

Quality of Service (QoS)

Quality of Service refers to property of a communication channel that are non-functional, such a robustness, guaranteed bandwidth, maximum latency, jitter, and many more.

Real-Virtual Interaction

Real-Virtual Interaction refers to Augmented Reality setup that additionally allow users to interact with real-world objects through virtual proxies in the scene that monitor and visualize the state in the real-world and that can use services to change the state of the real world (e.g. switch lights on an off via a virtual button the 3D scene).

Scene

A *Scene* refers to a collection of objects, which are be identified by type (eg. a 3D mesh object, a physics simulation rigid body, or a script object.) These objects contain typed and named data values (composed of basic types such as integers, floating point numbers and strings) which are referred to as attributes. Scene objects can form a hierarchic (parent-child) structure. A HTML DOM document is one way to represent and store a scene.

Security

Security is a property of an IT system that ensures confidentiality, integrity, and availability of data within the system or during communication over networks. In the context of middleware it refers to the ability of the middleware to guarantee such properties for the communication channel according to suitably expressed requirements needed and guarantees offer by an application.

Security Policy

Security Policy refers to rules that need to be fulfilled before a network connection is established or for data to be transferred. It can for example express statements about the identity of communication partners, properties assigned to them, the confidentiality measures to be applied to data elements of a communication channel, and others.

Synchronization

Synchronization is the act of transmitting over a network protocol the changes in a scene to participants so that they share a common, real-time perception of the scene. This is crucial to eg. implementing multi-user virtual worlds.

Type Description

Type Description in the context of the AMi middleware refers to the internal description of native data types or the interfaces described by an IDL. It contains data such as the name of a veriable, its data type, the hierarchical relations between types (e.g. structs and arrays), its memory offset and alignment within another data type, and others. Type Description are used to generate the mapping of native data types to the data that needs to be transmitted by the middleware.

Virtual Character

Virtual Character is a 3D object, typically composed of triangle mesh geometry, that can be moved and animated and can represent an user's presence (avatar) in a virtual world. Typically supported forms of animation include skeletal animation (where a hierarchy of "bones" or "joints" controls the deformation of the triangle mesh object) and vertex morph animation, where the vertices of the triangle mesh are directly manipulated. Virtual character systems may support composing the character from several mesh parts, for example separate upper body, lower body and head parts, to allow better customization possibilities.

18 POI Data Provider Open API Specification

18.1 Introduction

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

18.1.1 POI Data Provider RESTful API

The POI Data Provider generic enabler provides a RESTful API for accessing and modifying POI data. The API is accessed via HTTP and it uses JSON based representations for information interchange.

18.1.2 Intended Audience

This specification is intended to be used by data explorers, software developers and systems administrators. To use this information, the reader should firstly have a general understanding of the FIWARE.OpenSpecification.MiWi.POIDataProvider. You should also be familiar with:

- RESTful web services
- HTTP/1.1
- JSON data serialization format.
- JSON Schema v4

18.1.3 API Change History

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

Revision Date	Changes Summary
2013-10-16	• Initial Version
2013-12-31	3.2 - Added: bounding box search, contact information, media, grouping. Improved: location data. Modified: query syntax [Version 3.2]
2014-02-13	Language support, support for availability times.
2014-03-13	First draft of Release 3.3

18.1.4 How to Read This Document

It is assumed that the reader is familiar with RESTful web services architectural style. The document uses the following notation:

- 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.
- The variables are represented between brackets, e.g. {id} and in italic font. When the reader find it, can change it by any value.

For a description of some terms used along this document, see the FIWARE.ArchitectureDescription.MiWi.POIDataProvider

18.1.5 Additional Resources

For more details about the POI GE that this API is based upon, please refer to FIWARE.OpenSpecification.MiWi.POIDataProvider. Related documents, including an Architectural Description, are available at the same site.

18.2 API Overview

Needs associated to a POI provider are very different in different applications. POI system intended for tourists is supposed to offer different kinds of data than various systems for different professional uses. Different organizations may provide different kind of POI information or information about different POIs. It would be difficult and bulky to satisfy all needs by a single unified data description.

In this approach the diversity of needs is addressed by modular data structure supporting independent components. The POI Data Provider GE provides a core component containing basic information about pois that enables spatial search. Additional data components are specified extending the POI data model, including 3D content, multimedia items and sensor information.

18.2.1 Modular POI Data Model

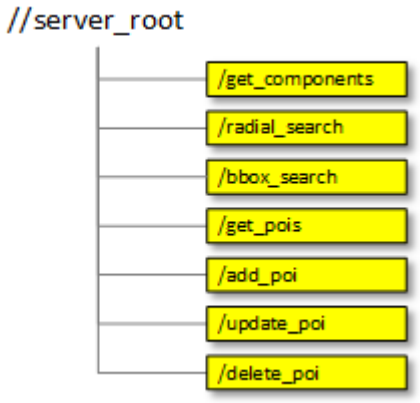
The design of the POI data model is based on the principles of the entity-component model, which allows partitioning the POI data into separate components that are linked to a POI entity. A single POI is an entity, which is identified by a universally unique identifier (UUID). The data related to an entity is stored in different data components, which are linked to the entity using the UUID. These data components are independent of each other and every POI entity can have a different set of components. Every POI must, however, have a 'fw_core' component that contains the very minimal core data describing the POI.

Recommendation: A data component name consists of a short organization identification and a short descriptive type name separated by an underscore "_". The name must be an acceptable JavaScript variable name in lower case only. Fi-Ware project will reserve "`fw`" as its organization identification.

The structure of a POI data item in JSON format:

```
"UUID of the POI": {
    "fw_core": { /* Basic POI data */ },
    "xcomp_data1": { /* Data1 defined by Xcomp organization */ },
    /* More components */
}
```

18.2.2 Resources Summary



18.2.3 Authentication

Authentication is out of this specification's scope. For example the standard authentication mechanisms provided by the web server used in the implementation can be utilized.

18.2.4 Representation Format

The *POI Data Provider RESTFul API* supports JSON data format, where the data is structured as key-value pairs.

POI data is modular consisting components, which are represented as JSON objects. The key of each JSON object identifies the type of POI data component it represents.

The JSON schema and an example JSON structure of the POI data provided by the API is shown below. The root level JSON object is named "pois", which contains all the POIs corresponding to the query. Each POI is represented as a JSON object, where the UUID of the POI is the key of the object. Each POI object contains an individual set of data components.

JSON schema of query response main structure

```
 {
   "title": "POIS Query Response",
   "description": "Generic POIS response.",
   "type": "object",
```

```
   "properties": {
     "pois": {
       "description": "Contains one object per a point of interest. The
key of an object is the UUID of the POI.",
       "type": "object",
       "additionalProperties": {
         "title": "POI data, key is the UUID of the POI"
         "description": "The POI data consists of data components that
are identified by their keys.",
         "type": "object",
         "additionalProperties": {
           "title": "POI data component, key defines the structure"
         }
       }
     }
   }
 }
```

Example of query response main structure - details hidden

```
 {
   "pois": {
     "8e57d2e6-f98f-4404-b075-112049e72346": {
       "fw_core": {
         "category": "library",
         "location": {
           "wgs84": {
             "latitude": 65.0612507,
             "longitude": 25.4667681
           }
         },
         "name": {
           "":"Tiedekirjasto Pegasus"
         },
         /* more core data */
       }
     },
     "30ddf703-59f5-4448-8918-0f625a7e1122": {
       /* POI data */
     }
```

```
    }
  }
```

18.2.5 Support for multilingual content

Many attributes in the POI data, such as strings and URLs, may have multiple language variants. A POI data attribute containing a multilingual value is a JSON object containing key-value pairs, where the key is the ISO 639-1 language code of the language in which the value is represented.

The language variants returned by the response depend on the `Accept-Languages` HTTP header of the query.

18.2.6 Representation Transport

Resource representation is transmitted between client and server by using HTTP 1.1 protocol, as defined by IETF RFC-2616. *Details added if necessary.*

18.2.7 Resource Identification

POIs are identified by UUID strings (universally unique) IETF RFC-4122.

18.2.8 Links and References

POI data contain URLs of POI related relevant resources.

18.2.9 Versions

The POI Data Provider API and data model represented as JSON schema are versioned according to the FI-WARE release number.

18.2.10 Faults

18.2.10.1 *Synchronous Faults*

In this section, we provide the complete list of possible fault elements and error code, and if it is expected in all requests or not.

Fault Element	Associated Error Codes	Expected in All Requests?
GET	`400 Bad Request`, `500 Internal Server Error`	[YES]
POST	`400 Bad Request`, `500 Internal Server Error`	[YES]
DELETE	`400 Bad Request`, `500 Internal Server Error`	[YES]

A textual description of the fault that occurred will be provided in addition to the HTTP error code.

18.3 API Operations

The following section provides the detail for each RESTful operation giving the expected input and output for each request. For simplicity sake the API operations have been grouped into following categories: *Platform*, *Query*, and *Update*.

18.3.1 Platform

Verb	URI	Description
GET	/get_components	Provide the POI data components supported by this server

18.3.1.1.1 */get_components*

Return the list of POI data components available from this server.

```
GET /get_components
```

Sample result:

```
{
  "components": [
    "fw_core"
  ]
}
```

18.3.2 Query

Verb	URI	Description
GET	/radial_search	Provide the POIs within given circle
GET	/bbox_search	Provide the POIs within given bounding box
GET	/get_pois	Provide the pois listed by the query

18.3.2.1.1 */radial_search*

Return the data of POIs within a given distance from a given location.

```
GET /radial_search?{parameters}
```

Mandatory parameters:
`lat=`*latitude* - Latitude of the center of the search circle [degrees]

`lon=`*longitude* - Longitude of the center of the search circle [degrees] Optional parameters:
`radius=`*radius* - Radius of the search circle [meters], default is implementation dependent
`category=`*category* - POI category/categories to be included to results. Several categories can be given by separating them with commas. If this parameter is not given, all categories are included.
`component=`*component* - POI data component name(s) to be included to results. Several component names can be given by separating them with commas. If this parameter is not given, all components are included.
`max_results=`*max_results* - Maximum number of POIs returned.
`begin_time=`*begin_time* - When time of interest begins. *See 'Time format' below.* Optional, requires end_time.
`end_time=`*end_time* - When time of interest ends. *See 'Time format' below.* Required, if begin_time is defined.
`min_minutes=`*min_minutes* - Minimum time of availability in minutes. Optional. If begin_time is defined, default: a short time > 0.

Time format

Basic rule: ISO 8601 adaptation format [1] is used for times. However, it is allowed to leave the time zone definition out. If time zone is missing, the local time zone of the POI is used. *This specification does not require implementation of time zone functionality.* E.g.: '2014-01-23', '2014-01-23T13:34'

Sample query:

```
GET http://{poi server}/radial_search?lat=65.059254&lon=25.470997&radius=250&category=cafe,restaurant&begin_time=2014-01-23T11:30&end_time=2014-01-23T13:00&min_minutes=30&max_results=2
```

Sample result:

```
{
  "pois": {
    "6be4752b-fe6f-4c3a-98c1-13e5ccf01721": {
      "fw_core": {
        "category": "cafe",
        "location": {
          "wgs84": {
            "latitude": 65.059334,
            "longitude": 25.4664775
          }
```

```
        },
        "name": {
          "": "Aulakahvila"
        }
      }
    },
    "4c6754d9-0bb1-4962-b604-6f3fcc08a9ec": {
      "fw_core": {
        "category": "restaurant",
        "location": {
          "wgs84": {
            "latitude": 65.0581807,
            "longitude": 25.4667163
          }
        },
        "name": {
          "": "Discus"
        }
      }
    }
  }
}
```

18.3.2.1.2 */bbox_search*

Return the data of POIs within a given bounding box.

```
GET /bbox_search?{parameters}
```

Mandatory parameters:
`north=`*latitude* - Latitude of the northern edge of the bounding box [degrees]
`south=`*latitude* - Latitude of the southern edge of the bounding box [degrees]
`east=`*longitude* - Longitude of the eastern edge of the bounding box [degrees]
`west=`*longitude* - Longitude of the western edge of the bounding box [degrees]
Optional parameters:
`category=`*category* - POI category/categories to be included to results. Several categories can be given by separating them with commas. If this parameter is not given, all categories are included.
`component=`*component* - POI data component name(s) to be included to results. Several component names can be given by separating them with commas. If this parameter is not given, all components are

included.
`max_results=`*max_results* - Maximum number of POIs returned.
`begin_time=`*begin_time* - When time of interest begins. *See 'Time format' below.* Optional, requires end_time.
`end_time=`*end_time* - When time of interest ends. *See 'Time format' below.* Required, if begin_time is defined.
`min_minutes=`*min_minutes* - Minimum time of availability in minutes. Optional. If begin_time is defined, default: a short time > 0.

Sample query:

```
GET http://{poi server}/bbox_search?north=65.1&south=64.9&east=25.6&west=25.3&category=cafe,restaurant&max_results=2
```

Results as in /radial_search.

18.3.2.1.3 */get_pois*

Return the data of POIs listed in the query. This is intended to get additional information - other components - about interesting POIs.

```
GET /get_pois?{parameters}
```

Mandatory parameters:
`poi_id=`*UUID* - UUID of the POI. Several UUIDs can be given by separating them with commas.
Optional parameters:
`component=`*component* - POI data component name(s) to be included to results. Several component names can be given by separating them with commas. If this parameter is not given, all components are included.
`get_for_update=true` - The components requested are returned with all language and other variants and possible metadata for inspection and edit.

Sample query:

```
GET http://{poi server}/get_pois?poi_id=30ddf703-59f5-4448-8918-0f625a7e1122&component=fw_core
```

Sample result:

```
{
  "pois": {
    "30ddf703-59f5-4448-8918-0f625a7e1122": {
```

```
      "fw_core": {
        "category": "cafe",
        "location": {
          "wgs84" {
            "latitude": 65.059334,
            "longitude": 25.4664775
          }
        },
        "name": {
          "": "Aulakahvila"
        }
      }
    }
  }
}
```

18.3.3 Update

Update operations should require authorization in order to prevent misuse. The details of the authorization mechanism are out of the scope of this specification.

Verb	URI	Description
POST	/add_poi	Add a new POI to the database
POST	/update_poi	Update existing POI data
DELETE	/delete_poi	Delete existing POI from database

18.3.3.1.1 */add_poi*

This function is used for adding a new POI entity into a database. The POI data is given as JSON in HTTP POST request. It generates a UUID for the new POI and returns it to the client in JSON format including the timestamp of the POI creation. The client can include different data components to the new POI by sending them along with the request.

The POSTed JSON must include only the content of a single POI, i.e. it must not contain a UUID as key as it is automatically generated by the server.

Example request:

```
POST http://{poi server}/add_poi
```

```
{
  "fw_core": {
       "category": "cafe",
       "location": {
         "wgs84" {
              "latitude": 65.059334,
              "longitude": 25.4664775
         }
       },
       "name": {
         "": "Aulakahvila"
       }
  }
}
```

Sample result:

```
{
  "created_poi": {
    "uuid": "30ddf703-59f5-4448-8918-0f625a7e1122",
       "timestamp": 1394525977
  }
}
```

18.3.3.1.2 */update_poi*

This function is used for updating data of an existing POI entity. Existing data components can be modified or new ones can be added. Each data component contains a 'last modified' timestamp in order to prevent concurrency issues.

The updated POI data is given as JSON in HTTP POST request. The server responds with HTTP status messages indicating the success or failure of the operation.

Example request:

```
POST http://{poi server}/update_poi
{
  "30ddf703-59f5-4448-8918-0f625a7e1122": {
```

```
    "fw_core": {
         "category": "cafe",
         "location": {
           "wgs84" {
                "latitude": 65.059334,
                "longitude": 25.4664775
           }
         },
         "name": {
           "": "Aulakahvila"
         },
         "description": {
           "": "Cafe at the Univesity of Oulu"
      }
    }
  }
}
```

Sample result:

```
HTTP/1.1 200 OK
POI data updated succesfully
```

18.3.3.1.3 */delete_poi*

Delete existing POI using HTTP DELETE request. The UUID of the POI to be deleted is given in the request as a URL parameter

Example request:

```
DELETE  http://{poi  server}/delete_poi?poi_id=30ddf703-59f5-4448-8918-
0f625a7e1122
```

Example result:

```
HTTP/1.1 200 OK
POI entity deleted succesfully
```

18.4 POI Data Model Definitions

All the different data components that have been specified are described in this section. Also all the generic utility data types that are utilized in the different POI data components are defined. Each component is described with a examplary JSON structure and a JSON schema definition.

The data components are JSON objects that comprise the POI data structure. The components are identified by their keys. The key identifies the structure and interpretation of the component.

18.4.1 FW POI Data Components

Specifications of POI data components defined in this project.

NOTE: In JSON schemas the `$ref` definitions refer to *utility types*.

18.4.1.1 *fw_core*

fw_core POI data component contains the essential information about a POI, such as the name, geographical location and category. This information is used for indexing the POI database thus enabling search functionality.

JSON example

```
 {
   "fw_core": {
     "category": "restaurant",
     "name": {
       "": "Aularavintola"
     },
     "url": {
       "_def": "fi",
       "fi":                    "http://www.uniresta.fi/ravintolat/kaikki-
ravintolat/linnanmaa/aularavintola.html"
     },
     "label": {
       "_def": "en",
       "fi": "Unirestan Aularavintola Oulun yliopistolla.",
       "en": "Uniresta Lobby Restaurant at University of Oulu",
       "sv": "Uniresta Resgaurang av Oulus Universitet",
       "es": "Restaurante Aularavintola"
     },
     "location": {
```

```
        "wgs84": {
          "latitude": 65.059264,
          "longitude": 25.4664307
        }
      },
      "thumbnail":
"https://dl.dropboxusercontent.com/u/55717919/aularavintola_tb.jpg",
      "description": {
        "_def": "en",
        "fi": "Oulun yliopiston syd\u00e4mess\u00e4, keskusaulassa sijaitseva 550-paikkainen Aularavintola palvelee kahdessa kerroksessa yli 3 000 asiakasta p\u00e4ivitt\u00e4in. Aularavintolasta l\u00f6yd\u00e4t useita erilaisia vaihtoehtoja. Kahden lounaslinjaston lis\u00e4ksi sinua palvelee Wokkipaja, joka valmistaa ala carte-annokset odottaessasi ja Salad Bar, jonka valikoimasta sinulle kootaan salaatti toiveittesi mukaan.",
        "en": "Lunch buffet and ala carte restaurant of 550 seatings."
      },
      "source": {
        "name": "Open Alleymap",
        "website": "http://www.example.org/open_alleymap",
        "id": "49115cb0-8286-11e3-baa7-0800200c9a66",
        "license": "http://www.example.org/alley_license.txt"
      },
      "last_update": {
        "timestamp": 1390305508467,
        "responsible": "28509ff0-8294-11e3-baa7-0800200c9a66"
      }
    }
  }
```


JSON schema


```
    "fw_core": {
      "description": "For spatial search and finding that interesting one",
      "type": "object",
      "properties": {
```

```
        "category": {
          "description": "A descriptive tag for narrowing the search: cafe, museum, etc.",
          "type": "string"
        },
        "location": {
          "description": "Location of the POI",
          "$ref": "#/definitions/location"
        },
        "geometry": {
          "title": "Geometrical form of the POI",
          "description": "Format: Open Geospatial Consortium's 'Well-known text' ISO/IEC 13249-3:2011",
          "type": "string"
        },
        "short_name": {
          "description": "Short name to be shown on the map or in a narrow list",
          "$ref": "#/definitions/intl_string_31"
        },
        "name": {
          "description": "Full name of the POI",
          "$ref": "#/definitions/intl_string"
        },
        "label": {
          "description": "More info to complement the name, if enough space",
          "$ref": "#/definitions/intl_string_127"
        },
        "description": {
          "description": "Text to facilitate decision to be interested or not",
          "$ref": "#/definitions/intl_string"
        },
        "thumbnail": {
```

```
          "description": "A small picture to be shown on a list, scene
or map",
          "type": "string",
          "format": "uri"
        },
        "url": {
          "description": "URL to get more info",
          "$ref": "#/definitions/intl_uri"
        },
        "last_update": {
          "$ref": "#/definitions/update_stamp"
        },
        "source": {
          "$ref": "#/definitions/source"
        }
      }
    }
```

18.4.1.2 *fw_time*

fw_time defines temporal availability of the services of the POI. Two types of schedules are defined: **open** tells when the facility is open for business or visiting, **show_times** defines times of performances, presentations, etc. where participation from the beginning to the end is required or recommended. Notation for the **schedule** is defined in the utility type `schedule`.

JSON example

```
/*  opening_hours=Mo  10:00-12:00,12:30-15:00;  Tu-Fr  08:00-12:00,12:30-
15:00;
    Sa 08:00-12:00 */


 {
   "fw_time": {
     "type": "open",
     "schedule": {
       "or": [
         {
           "and": [
             {"wd": [1]},
```

```
            {
              "or": [
                {"bhr": [10], "ehr": [12]},
                {"bhr": [12,30], "ehr": [15]}
              ]
            }
          ]
        },
        {
          "and": [
            {"wd": [2, 3, 4, 5]},
            {
              "or": [
                {"bhr": [8], "ehr": [12]},
                {"bhr": [12,30], "ehr": [15]}
              ]
            }
          ]
        },
        {"wd": [6], "bhr": [8], "ehr": [12]}
      ]
    },
    "source": {
      "name": "Open Alleymap",
      "website": "http://www.example.org/open_alleymap",
      "id": "49115cb0-8286-11e3-baa7-0800200c9a66",
      "license": "http://www.example.org/alley_license.txt"
    },
    "last_update": {
      "timestamp": 1390305508467,
      "responsible": "28509ff0-8294-11e3-baa7-0800200c9a66"
    }
  }
}
```

JSON schema

```
   "fw_time": {
     "description": "",
     "type": "object",
     "properties": {
       "type": {
         "description": "Open - available thru open time, show_times -
available at beginnings of shows",
         "enum": ["open", "show_times"]
       },
       "time_zone": {
         "description": "TBD. Local time is assumed. Standardized
notation including daylight savings time reference is needed."
       },
       "schedule": {
         "description": "",
         "$ref": "#/definitions/schedule"
       }
     },
     "last_update": {
       "$ref": "#/definitions/update_stamp"
     }
   }
```

18.4.1.3 *fw_xml3d*

fw_xml3d contains the required information for linking a 3D model stored in an XML3D format to a POI.

JSON example

```
 {
   "fw_xml3d": {
     "model_id": "coffee3",
     "model": "<group id=\"coffee3\"
transform=\"https://dl.dropboxusercontent.com/u/17661643/coffee.xml#t_I
cosphere\"><group
shader=\"https://dl.dropboxusercontent.com/u/17661643/coffee.xml#Coffee
\"><mesh
src=\"https://dl.dropboxusercontent.com/u/17661643/coffee.xml#mesh_Icos
phere_Coffee\" type=\"triangles\"/></group></group>",
     "source": {
       "name": "Open Alleymap",
       "website": "http://www.example.org/open_alleymap",
       "id": "49115cb0-8286-11e3-baa7-0800200c9a66",
```

```
        "license": "http://www.example.org/alley_license.txt"
      },
      "last_update": {
        "timestamp": 1390305508467,
        "responsible": "28509ff0-8294-11e3-baa7-0800200c9a66"
      }
    }
}
```

JSON schema

```
    "fw_xml3d":{
      "description": "3D description",
      "type": "object",
      "properties": {
        "model_id": {
          "description": "ID for XML3D engine",
          "type": "string"
        },
        "model": {
          "description": "Model for XML3D engine",
          "type": "string"
        }
      },
      "last_update": {
        "$ref": "#/definitions/update_stamp"
      }
    }
```

18.4.1.4 *fw_contact*

Fw_contact component is intended to provide contact information of the main service related to the POI when applicable. Postal address can be omitted, if the visiting address includes all the information.

JSON example

```
{
   "fw_contact": {
     "visit": "Santa Claus Village?, Napapiiri, 96930 Rovaniemi, Finland",
     "postal": ["Santa Claus Village?","Napapiiri","96930 Rovaniemi","Finland"],
     "mailto": "travel.info@rovaniemi.fi",
     "phone": "+358 16 346270",
```

```
      "source": {
        "name": "Open Alleymap",
        "website": "http://www.example.org/open_alleymap",
        "id": "49115cb0-8286-11e3-baa7-0800200c9a66",
        "license": "http://www.example.org/alley_license.txt"
      },
      "last_update": {
        "timestamp": 1390305508467,
        "responsible": "28509ff0-8294-11e3-baa7-0800200c9a66"
      }
    }
  }
```

JSON schema

```
  "fw_contact": {
    "properties": {
      "visit": {
        "description": "Visiting address good for a taxi driver or
Google Maps",
        "type": "string"
      },
      "postal": {
        "description": "Postal address. One string per line",
        "type": "array",
        "items": {
          "type": "string"
        }
      },
      "mailto": {
        "description": "Email address",
        "type": "string"
      },
      "phone": {
        "description": "Phone number",
        "type": "string"
      },
      "last_update": {
        "$ref": "#/definitions/update_stamp"
      }
    }
  }
```

18.4.1.5 *fw_media*

Fw_media component is intended to provide links to available related media items to the user. Currently available media types are photo, video, audio, and folder. If a folder link provides JSON of this format, it can be shown using the same mechanism.

JSON example

```
{
   "fw_media": {
     "entities": [
       {
         "type": "photo",
         "short_label": {
           "en": "Sunset at sea"
         },
         "caption": {
           "en": "Sunset on the Bothnian Bay, Northwest from Hailuoto summer 2013"
         },
         "description": {
           "": "Lorem ipsum dolor sit amet, consectetur adipisci elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi consequat.",
           "fi": "Oli mukava retki."
         },
         "thumbnail": "http://www.example.org/sunset_on_sea_tbn.jpg",
         "url": "http://www.example.org/sunset_on_sea.jpg",
         "copyright": "Photo: Ari Okkonen"
       },
       {
         "type": "audio",
         "short_label": {
           "": "Säkkijärven polkka"
         },
         "url": "http://www.example.org/sakkijarven_polkka.mp3"
       }
```

```
    ],
    "last_update": {
      "timestamp": 1390305508467,
      "responsible": "28509ff0-8294-11e3-baa7-0800200c9a66"
    }
  }
}
```

JSON schema

```
  "fw_media": {
    "description": "Media items related to this item",
    "type": "object",
    "properties": {
      "entities": {
        "description": "",
        "type": "array",
        "items": {
          "description": "",
          "type": "object",
          "properties": {
            "type": {
                "description": "what kind of media item this is",
                "enum": [
                  "folder",
                  "photo",
                  "video",
                  "audio"
                ]
            },
            "short_label": {
              "description": "To be shown along the item",
              "$ref": "#/definitions/intl_string_31"
            },
            "caption": {
              "description": "To be shown along the item",
              "$ref": "#/definitions/intl_string_127"
            },
            "description": {
              "description": "More info about the item",
              "$ref": "#/definitions/intl_string"
            },
            "thumbnail": {
              "description": "A small picture to be shown in the
list",
              "type": "string",
```

```
                "format": "uri"
              },
              "url": {
                "description": "URL of the actual media item",
                "type": "string",
                "format": "uri"
              },
              "copyright": {
                  "description": "Copyright clause and/or attribution of
the item",
                  "type": "string"
              }
            }
          }
        }
      },
      "last_update": {
        "$ref": "#/definitions/update_stamp"
      }
    }
```

18.4.1.6 *fw_relationships*

fw_relationships allows defining one-to-many relations between POIs. The relation is defined as a triple: subject-predicate-objects. The subject is the source of the relation, the predicate defines the relation type and objects are the targets of the relation. The key of the predicate defines the ontology. "fw" is the ontology defined in FI-WARE project.

Relation ontology of FI-WARE project

```
contains An area contains points or smaller areas. A group or collection
contains items that may be any objects.
has_entry defines a point for entering an area or a large building.
```

JSON example

```
{
    "fw_relationships": [
        {
            "subject": "9202ee90-9fcb-11e3-a5e2-0800200c9a66",
            "predicate": {
                "fw": "contains"
            },
            "objects": [
                "8e1962e0-9fcc-11e3-a5e2-0800200c9a66",
                "c3f6e810-9fcc-11e3-a5e2-0800200c9a66",
                "d117ceb0-9fcc-11e3-a5e2-0800200c9a66",
                "df46f100-9fcc-11e3-a5e2-0800200c9a66"
            ]
```

```
        },
        {
            "subject": "9202ee90-9fcb-11e3-a5e2-0800200c9a66",
            "predicate": {
                "fw": "has_entry"
            },
            "objects": [
                "9d9f9340-ae7e-11e3-a5e2-0800200c9a66",
                "b7c8f680-ae7e-11e3-a5e2-0800200c9a66"
            ]
        },
        {
            "subject": "83891460-ae81-11e3-a5e2-0800200c9a66",
            "predicate": {
                "fw": "contains"
            },
            "objects": [
                "9202ee90-9fcb-11e3-a5e2-0800200c9a66"
            ]
        },
        {
            "subject": "f42d98d0-ae81-11e3-a5e2-0800200c9a66",
            "predicate": {
                "fw": "contains"
            },
            "objects": [
                "9202ee90-9fcb-11e3-a5e2-0800200c9a66"
            ]
        }
    ]
}
```

JSON schema

```
{
   "title": "Relationships",
   "type": "object",
   "properties": {
       "fw_relationships": {
           "description": "List of relationships the POI is connected
to.",
           "type": "array",
           "items": {
               "description": "specifies ONE to MANY relation between
POIs or other entities",
               "type": "object",
               "properties": {
```

```
                    "subject": {
                        "description": "UUID of the ONE in the
relation",
                        "type": "string"
                    },
                    "predicate": {
                        "description": "type of the relation",
                        "type": "object",
                        "additionalProperties": {
                            "description": "defines the type of the
relation within ontology defined by the key",
                            "type": "string"
                        }
                    },
                    "objects": {
                        "type": "array",
                        "items": {
                            "description": "UUIDs of the MANY in the
relation",
                            "type": "string"
                        }
                    },
                    "last_update": {
                        "$ref": "#/definitions/update_stamp"
                    }
                }
            }
        }
    }
}
```

18.4.1.7 *fw_marker*

fw_marker enables storing marker related information to a POI to be used e.g. in augmented reality applications.

JSON example

```
{
    "fw_marker": {
        "alvar_3x3": {
            "code": 33,
            "image_ref": "http://www.example.com/x_marker.gif"
        }
    }
}
```

JSON schema

```
{
   "title": "marker",
   "type": "object",
   "properties": {
       "fw_marker": {
           "description": "This contains technology dependent
properties only. Property keys define the technology used.",
           "type": "object",
           "properties": {
               "alvar_3x3": {
                   "description": "3x3 marker used in Alvar (VTT Oulu)
marker tracking system is used as an example here.",
                   "type": "object",
                   "properties": {
                       "code": {
                           "description": "code embedded to marker as
defined by Alvar",
                           "type": "integer"
                       },
                       "image_ref": {
                           "description": "image of the marker e.g. for
printing",
                           "type": "string",
                           "format": "uri"
                       }
                   }
               }
           },
           "additionalProperties": {
               "description": "image_ref is a recommended property in
other marker techniques, also",
               "type": "object",
               "properties": {
                   "image_ref": {
                       "description": "image of the marker e.g. for
printing",
                       "type": "string",
                       "format": "uri"
                   }
               }
           }
       }
   }
}
```

18.4.2 Utility Types

General structured data types used in components

18.4.2.1 *location*

JSON schema

```
{
  "title": "Location Data",
  "description": "",
  "type": "object",
  "properties": {
    "wgs84": {
      "title": "World Geodetic System",
      "description": "",
      "type": "object",
      "properties": {
        "latitude": {
          "description": "degrees north",
          "type": "number"
        },
        "longitude": {
          "description": "degrees east",
          "type": "number"
        },
        "altitude": {
          "description": "meters up from sea level, optional",
          "type": "number"
        },
      }
    }
  },
  "additionalProperties": {
      "title": "Possible other coordinate systems",
      "description": "Contents depend on the system defined by the
key",
      "type": "object",
    }
  }
}
```

18.4.2.2 *intl_string*

JSON schema

```
{
  "title": "Internationalized string",
```

```
  "description": "",
  "type": "object",
  "properties": {
    "_def": {
      "title": "Default language override",
      "description": "Optional key of the default language for this
string",
      "type": "string",
    },
    "": {
      "title": "Language independent text",
      "description": "",
      "type": "string",
    },
    "en": {
      "title": "Text in language 'en'",
      "description": "",
      "type": "string",
    }
  },
  "additionalProperties": {
      "title": "Text in language <key>",
      "description": "The key is ISO 639-1 Language Code",
      "type": "string",
    }
  }
}
```

18.4.2.3 *intl_string_<length>*

JSON schema

```
{
  "title": "Internationalized string",
  "description": "",
  "type": "object",
  "properties": {
    "_def": {
      "title": "Default language override",
      "description": "Optional key of the default language for this
string",
      "type": "string",
    },
    "": {
      "title": "Language independent text",
      "description": "",
```

```
      "type": "string",
      "maxLength": <length>
    },
    "en": {
      "title": "Text in language 'en'",
      "description": "",
      "type": "string",
      "maxLength": <length>
    }
  },
  "additionalProperties": {
      "title": "Text in language <key>",
      "description": "The key is ISO 639-1 Language Code",
      "type": "string",
      "maxLength": <length>
    }
  }
}
```

18.4.2.4 *intl_uri*

JSON schema

```
{
  "title": "Internationalized URI",
  "description": "",
  "type": "object",
  "properties": {
    "_def": {
      "title": "Default language override",
      "description": "Optional key of the default language for this
link",
      "type": "string",
    },
    "": {
      "title": "Language independent URI",
      "description": "",
      "type": "string",
      "format": "uri"
    },
    "en": {
      "title": "URI for language 'en'",
      "description": "",
      "type": "string",
      "format": "uri"
    }
  },
```

```
  "additionalProperties": {
      "title": "URI for language <key>",
      "description": "The key is ISO 639-1 Language Code",
      "type": "string",
      "format": "uri"
    }
  }
}
```

18.4.2.5 *schedule*

This defines the availability of a thing. This may be used to define service times, open times, show times, etc. The notation utilizes elementary interval definitions and operations from set theory to produce more complex definitions.

JSON example

```
/* opening_hours=Mo 10:00-12:00,12:30-15:00; Tu-Fr 08:00-12:00,12:30-
15:00;
    Sa 08:00-12:00 */


var opening_hours = {
        "or": [
          {
            "and": [
              {"wd": [1]},
              {
                "or": [
                  {"bhr": [10], "ehr": [12]},
                  {"bhr": [12,30], "ehr": [15]}
                ]
              }
            ]
          },
          {
            "and": [
              {"wd": [2, 3, 4, 5]},
              {
                "or": [
                  {"bhr": [8], "ehr": [12]},
```

```
                    {"bhr": [12,30], "ehr": [15]}
                  ]
                }
              ]
            },
            {"wd": [6], "bhr": [8], "ehr": [12]}
          ]
        };
```

JSON schema

```
{
  "properties": {
    "or": {
      "description": "union - valid when any of subschedules is valid",
      "type": "array",
      "items": {
        "#ref": "schedule"
      }
    },
    "and": {
      "description": "intersection - valid when all of subschedules are
valid",
      "type": "array",
      "items": {
        "#ref": {"schedule"}
      }
    },
    "not": {
      "description": "complement - valid when the subschedule is not
valid",
      "#ref": "schedule"
      }
    },
    "wd": {
      "description": "weekday: valid on listed weekdays,
1=monday,...,7=sunday",
      "type": "array",
      "items": {
        "type": "integer",
        "minimum": 1,
        "maxmum": 7
      }
    },
```

```
    "bhr": {
      "description": "begin hour [hour-integer, minute-integer, second-
number]. End zeros can be omitted.",
      "type": "array",
      "items": {
        "type": "integer",
      }
    },
    "ehr": {
      "description": "end hour [hour-integer, minute-integer, second-
number]. End zeros can be omitted.",
      "type": "array",
      "items": {
        "type": "integer",
      }
    },
    "bev": {
      "description": "begin event [year, month, day, hour, minute -
integers, second-number]. End zeros can be omitted.",
      "type": "array",
      "items": {
        "type": "integer",
      }
    },
    "eev": {
      "description": "end event [year, month, day, hour, minute -
integers, second-number]. End zeros can be omitted.",
      "type": "array",
      "items": {
        "type": "integer",
      }
    },
    "bdate": {
      "description": "begin date [ month, day] until the end of the
year",
      "type": "array",
      "items": {
        "type": "integer",
      }
    },
    "edate": {
      "description": "end date [ month, day] from the beginning of the
year",
      "type": "array",
      "items": {
        "type": "integer",
      }
    },
```

```
  }
}
```

18.4.2.6 *source*

JSON schema

```
  "source": {
    "properties": {
      "name": {
        "description": "Source of the component information",
        "type": "string"
      },
      "website": {
        "description": "Link to a related webpage",
        "type": "string",
        "format": "uri"
      },
      "id": {
        "description": "UUID of the source profile",
        "type": "string"
      },
      "license": {
        "description": "Link to applied license",
        "type": "string",
        "format": "uri"
      }
    }
  }
```

18.4.2.7 *update_stamp*

JSON schema

```
"update_stamp": {
  "properties": {
    "timestamp": {
      "description": "When updated - to catch conflicts",
      "type": "number",
      "format": "utc-millisec"
    },
    "responsible": {
      "description": "Who updated, UUID of the profile",
      "type": "string",
    }
  }
}
```

19 FIWARE OpenSpecification MiWi 2D-3D-Capture

Name	FIWARE.OpenSpecification.MiWi.2D-3D-Capture
Chapter	Advanced Middleware and Web-based UI,
Catalogue-Link to Implementation	2D/3D Capture
Owner	Cyberlightning Ltd., Tharanga Wijethilake

19.1 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.org and similar pages in order to understand the complete context of the FI-WARE project.

19.2 Copyright

- Copyright © 2013 by Cyberlightning Ltd.

19.3 Legal Notice

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

19.4 Overview

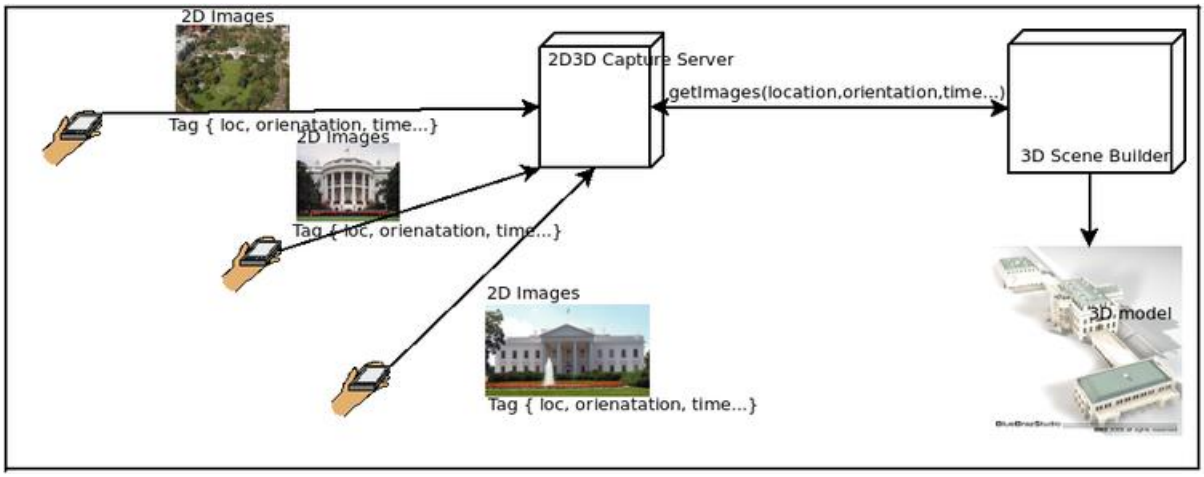
Ideally this service is consists of following set of capabilities.

- Capturing 2D and 3D visual information using standard browser API.
- Identify and capture relevant meta information that improves user experience, accuracy and performance.
- Using "tagging" techniques to encapsulate information with the image.
- Publish/Store/Submit tagged multimedia in known/requested public services.
- RESTfull API should be used to access multimedia for the use of other GE such as Cloud rendering GE, Augmented Reality GEs, Virtual Characters GEs.

2D 3D Capturing is capture contextual information related a 2D 3D scene of the surrounding so that the data can be used to provided to or as services. Location information, lighting information, device orientation, heading direction are the necessary contextual information and based on the service these

other information available to the browser can be used. Air pressure, time the image is taken are some of the available important meta data can also be used as "Tagging content". Tagging can be similar to the contemparary geo tagging but highly dependent on the image format(JPEG, PNG, ogg, wmv, ect....). Exif extension method can be used if the format used is JPEG and for image formats such as PNG another method can be used. Format of the image in image capture depends on the usage scenario. For example in a situation where image processing is minimal JPEG compresssion can be better over PNG lossless images.

19.4.1 Simple Use Case of Advanced data capture



As depicted in the image above, photos taken from devices combined with the metadata shown as Tag are transferred to the 2D3D capture server. 2D 3D capture server API then can be used to dig up image based on time, location for use of other applications where 2D or 3D visual information is required. In the use case described here 3D scene builder can be a part of other GEs such as POI. Another use case is, using tagged images published by other services or devices such as Flickr to build a 3d model of the current location so that a person such as lost tourist could find his way. Elaborating on the "Tag" and its content, images are geo-tagged in order to improve the user experience in social networks images. 3D information can be used similarly to improve the quality of the user experience and meta data that can be associated with a model improves number of use cases 3d modelling can be used. Most importantly information such as orientation of the device at the moment which the photo is taken can be used to improve the accuracy and it also helps to filter out less useful images from the useful once. In a hypothetical security situation where two images taken from known locations by two security cameras at the same time could be useful in identifying a criminal who happened to be at that location. By using meta data such as lighting conditions the accuracy of the 3d model of the criminal can be improved and will avoid innocent being targeted.

19.5 Basic Concepts

2D3D capture Capturing of 2D/3D data consists of 3 main aspects

- 2D/3D Data Capture
- Metadata capturing and Tagging/associating

- Data Streaming/Saving

19.5.1 Data Capture

Cameras of mobiles device's provide data in the form of still images and streams of 2D. There are cameras equipped with stereoscopic cameras yet they too are managed by software to create the 3D images using the stereoscopic data provided by the setting. HTML 5 and the standard W3C provides standard functions to obtain access to device cameras and sensors. Though there are W3C specifications browsers do not support a universal API. Abstract API is required to accommodate the functional differences of browsers. In relation with real time and actual representation of 3D data most popular today is GEO tagging photos. In modern mobile devices there are more information available to optimize the process placing photos in actual places so that they would provide accurate information to the 3D model creators. Hence capturing sensor information and encapsulate them with the Image is the primary target of this architecture document.

There some other means that anticipate future development of the 3D rendering of scene using actual real time data but they have rendered out of scope from this document. They are considered to be important and API's are designed to extend the functionality with such means.Capturing audio data as 3D for such purposes is, at the moment, a highly theoretical. One important aspect of audio is echo which is useful in detecting the proximity from the adjacent obstacles.

Capturing depth element from the 2D content from the device directly is highly desired. Current implementation will make an attempt to provide means to acheive this goal by identifying necessary information to extract depth from current images. This is a workaround at the server end in place for this as current processing powers of handheld devices is insufficient. Data capturing components should make the data available for back end 2D-3D capture server over http connection and transport mechanisms support loss less and lossy compressions and encoding mechanisms which contemporary browsers provides by supporting PNG, GIFF as well as JPEG images. By combining with services such as adaptive streaming (eg . [https://developer.mozilla.org/en-US/docs/DASH_Adaptive_Streaming_for_HTML_5_Video | DASH) similar results can be obtained for videos as well. Video streams are out of the scope of this document but algorithms developed to manipulate sensor data are still valid for video streams obtained from mobile devices.

This aspect of 2D3D capturing is dependent on the Device API's provided by the browsers.

19.5.2 MetaData

Necessary metadata that is required to respond to a service request are identified as follows.

Geo-location (Longitude, Latitudes)

Placing an Image on a map is achieved with this information

Altitude

Altitude gives the 3rd dimension of a map and also assist to identify images taken from air and multistory buildings.

Air pressure

In case altitude is not present for considerable difference proper altitude approximation can be made.

Orientation

This is the most important aspect which helps to place the photo in real 3D map. Device may be angled when the image is taken hence important to understand the portion of a scene covered by the particular. It also helps to filter out most appropriate images to use as stereoscopic images for model building.

Heading direction

in a single position (Longitude, Latitudes) the full circle a user may be facing provides another filtering parameter.

Lighting

In most mobile devices main camera is located opposite to the light sensor but secondary camera is aligned with the sensor. This parameter is one of the filtering parameters to identify proper images that accurately depict the scene

Time

Meta data information mentioned above are part of the GPS standard specification[1] for W3C and most importantly this is a part of Mozilla API[2][3] [4]. Abstract API needs to be defined to harmonize the data extraction from a range of browsers.

Advanced geo tag that needs to be embedded can be of following form in json.

```
    1.First three digits explain the length of the metadata.eg 345.
    2.Metadata is of the following format
    message {
         "type":"image",              // Image/Video
         "time":"2014.3.4_14.40.30",
//year_month_day_hours_minutes_Seconds This is used as the Image name
         "ext":"png",                  //Image compression type. JPG and
PNG are supported
         "DeviceType":"Mobile",       //Mobile/Desktop
         "DeviceOS":"Badda",           //Operating system
         "Browsertype":"Firefox",     //Browser application
         "position":  {
               "lon":25.4583105,     //Longiture
               "lat":65.0600797,     //Latitude
               "alt":-1000,          //Altitude
               "acc":48.38800048828125 //Accuracy
```

```
          },
          "device":  {                //Seonser information
                "ax":0,   "ay":0,   "az":0,   "gx":0,   "gy":0,   "gz":0,
"ra":210.5637, "rb":47.5657, "rg":6.9698,  //Accellerometer information
                "orientation":"potrait"
          },
          "vwidth":480,
          "vheight":800
      }
```

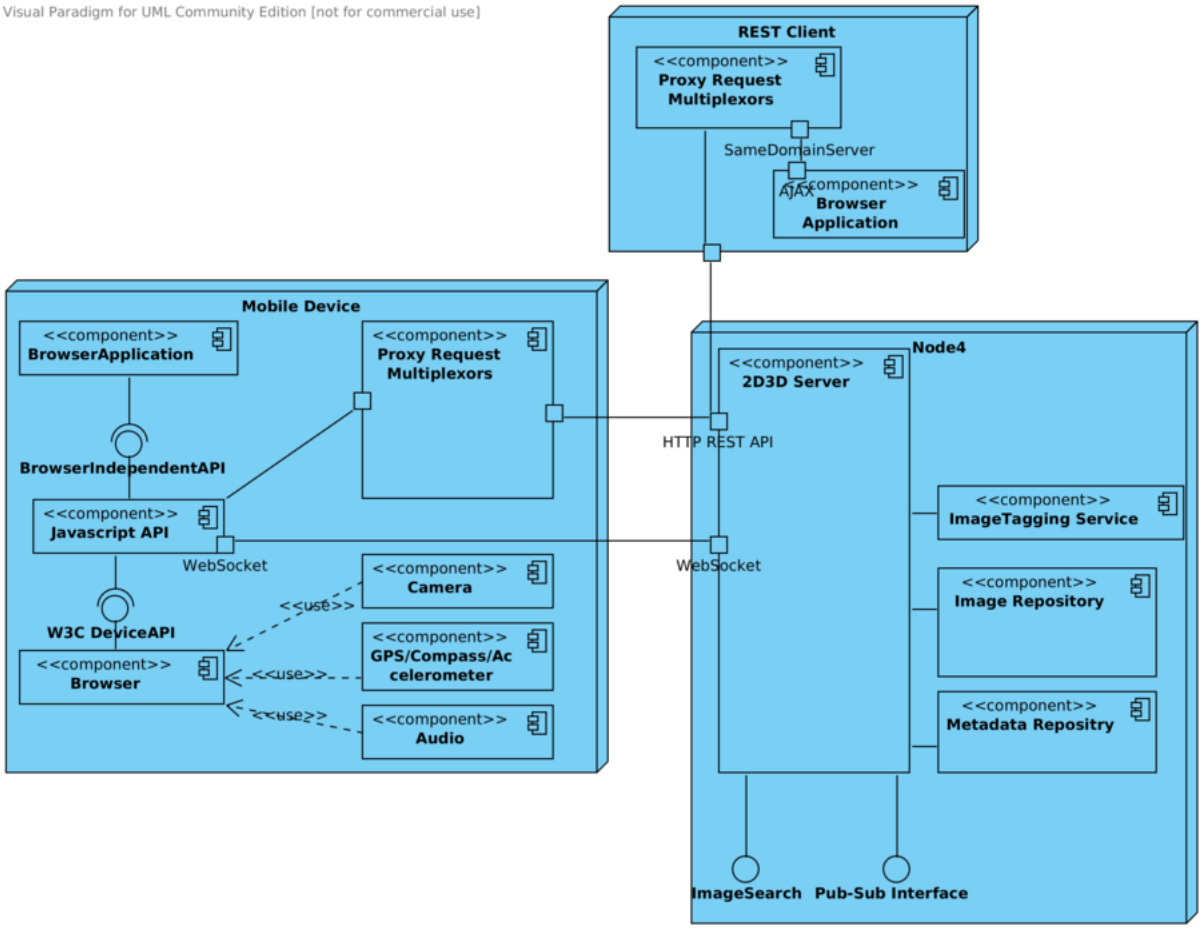
Currently audio is considered meta data due to the reason that it does not provide specific information to 3D context with contemporary technologies but remains to be a important data with useful use cases as mentioned in the data capture topic. Apart from those use cases could be used in value added services So audio remains a meta data as well.

19.5.3 Data Streaming/Saving

Out of the scope from the API but is included for completion. Basic requirements are similar to images but intermediate components that needs streaming needs to be identified to make the architecture component.

Conceptually capturing should support saving media locally with tagged information as well as publishing to required services after tagging. Tagging Streams remain a major challenge and it is yet to realize means for such implementations. For streaming from portable devices data capture server can use similar services to DASH like services which assists adaptive streaming YouTube like services should be able to adapted and tagged content from the capture server. For still images services such as Flickr already provides services but the capture service should support publishing and extracting tagged image data from public services for pub sub or on demand services.

19.6 Generic Architecture



Above diagram is a very high level diagram and it can only be considered as a conceptual architecture of a prototype. Generic Architecture of 2D/3D capture is multiplier client server architecture where there is a client residing on capturing device, content capture server in the back end and main storage.

Architecture consists of three main components.

- Mobile Clients
- Public Server Application
- REST Clients

19.6.1 Mobile Clients

Mobile Clients are main information capturing devices in 2D3D capture applications. Contemporary browsers provide APIs but standard APIs are far from being used by any of the available browsers. Contemporary mobile applications are in need for a Unifying API to address the problem and this API is one such attempt. This is a very limited API and parameters such as orientation information which expected to be obtained from device compass are adjusted to be able to feed as data to 3D contexts such as WebGL. Accuracy of the information are dependent on manual calibration of the device.

Contemparary web technologies such as AJAX, HTML5 provide means for this type of services.

19.6.2 Public Server Application - 2D3D Capture server

Main feature of the service is providing a repository for images and their meta information. Server stores meta data in data repository and also they are embedded in images when they are inserted to the repository. Server consists of three interfaces and one internal Module.

- EventManager
- REST server
- Web socket Server are the three interfaces and internal module consists of Database interface functions that are required to access and retrieve data from and to the MYSQL database.

Rest server provides means to

- Carry out basic server functions (Start/Stop)
- Upload Images
- Query Images and relevant information
- Subscribe For Event from Event manager

Event Manager Provides Means to update subscribers about latest updates to the server. Purpose of this is to carry out image processing in real time "As It Happens".

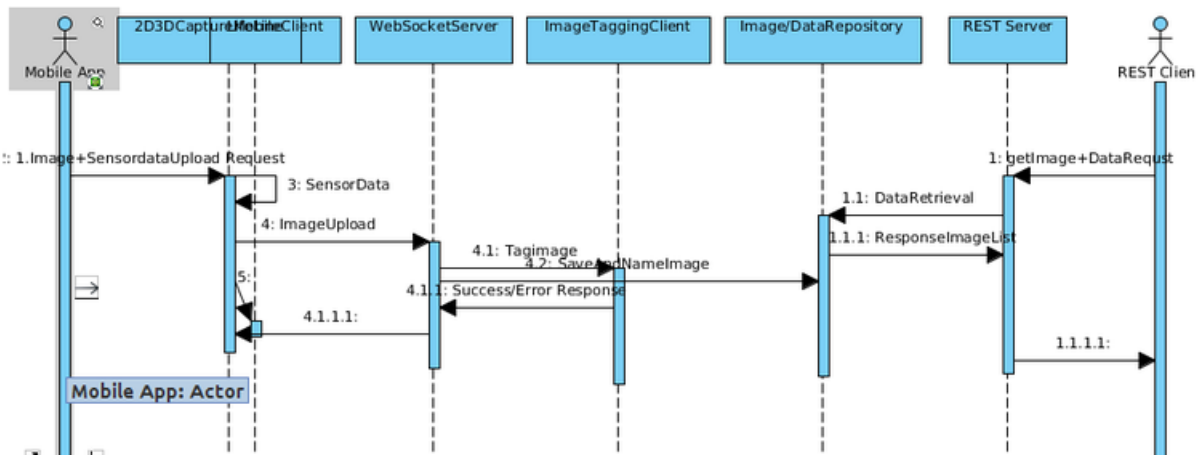
Web Socket server is the back bone of Event manager. It also provides means to connect and save images using the Mobile API.

Combination of Database layer and a web server technologies can be used to realize implementation goals of this component.

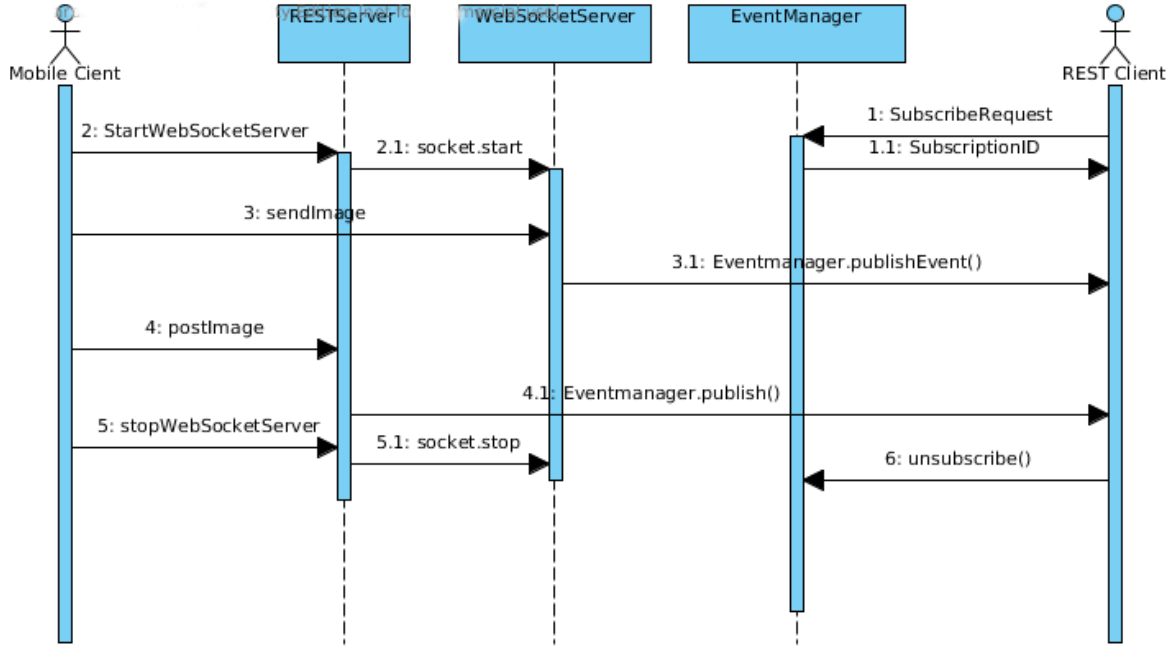
19.7 Main Interactions

Following diagram depicts very basic interactions of 2D3D capture.

- an user decided to upload content to a information capturing service and
- a single service requesting services from a single available device.



Sequence Diagram of the real time publisher subscriber scenario is as follows.



These two very preliminary interactions require following interfaces from the 2D3D capture server, purely on an information capture point of view those are the necessary transactions related to a server.

Interactions concerning 2D-3D capture can be directly related with GE requests. Follwing hypothetical situation comprises of some interactions that concerns this specification. Service Based on POI can demand data capture and POI IDs can be used as meta-data. **2D-3D-Capture** could use contemporary 3D services such as youtube 3D and Flickr 3D community hence the requirement for capture and publish data in public services can be supported. CloudRendering, RealVirtualInteraction and VirtualCharacters require much finer details such as extracting texture material on real time and off line. Further AugmentedReality GE live feed or an Image can request the feed annotated based on the facing direction and the location. SceneAPI,DataflowProcessing GEs also indirectly affect 2D 3D capture API as they depend on the update of the data repositories and streams. These requirements can be generalized for generic service enablers to define advanced interactions.

19.8 Basic Design Principles

- Development of this GE focused on few scenarios.
- API should define necessary functions leaving room for customization.
- Most important design principle is provide room for future developments. This includes catering for increasing number of sensors that portable devices area equipped with and supporting advanced data capturing mechanisms.
- Persist heterogeneity of browser echo systems from the application developer.

19.9 Detailed Specifications

Following is a list of Open Specifications linked to this Generic Enabler. Specifications labeled as "PRELIMINARY" are considered stable but subject to minor changes derived from lessons learned during last interactions of the development of a first reference implementation planned for the current Major Release of FI-WARE. Specifications labeled as "DRAFT" are planned for future Major Releases of FI-WARE but they are provided for the sake of future users.

19.9.1 Open API Specifications

The API Specification is still under development and will be released close to the first software release. The following is an early draft of the APIs, subject to change.

- 2D/3D Capture Open API Specification

19.10 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

Annotations

Annotations refer to non-functional descriptions that are added to declaration of native types, to IDL interface definition, or through global annotations at deployment time. The can be used to express security requirements (e.g. "this string is a password and should be handled according the security policy defined for password"), QoS parameters (e.g. max. latency), or others.

AR

AR → Augmented Reality

Augmented Reality (AR)

Augmented Reality (AR) refers to the real-time enhancement of images of the real world with additional information. This can reach from the rough placement of 2D labels in the image to the perfectly registered display of virtual objects in a scene that are photo-realistically rendered in the context of the real scene (e.g. with respect to lighting and camera noise).

IDL

IDL → Interface Definition Language

Interface Definition Language

Interface Definition Language refers to the specification of interfaces or services. They contain the description of types and function interfaces that use these types for input and output parameters as well as return types. Different types of IDL are being used including CORBA IDL, Thrift IDL, Web

Service Description Language (WSDL, for Web Services using SOAP), Web Application Description Language (WADL, for RESTful services), and others.

Middleware

Middleware is a software library that (ideally) handles all network related functionality for an application. This includes the setup of connection between peers, transformation of user data into a common network format and back, handling of security and QoS requirements.

PoI

PoI → Point of Interest

Point of Interest (PoI)

Point of Interest refers to the description of a certain point or 2D/3D region in space. It defines its location, attaches meta data to it, and defines a coordinate system relative to which additional coordinate systems, AR marker, or 3D objects can be placed.

Quality of Service (QoS)

Quality of Service refers to property of a communication channel that are non-functional, such a robustness, guaranteed bandwidth, maximum latency, jitter, and many more.

Real-Virtual Interaction

Real-Virtual Interaction refers to Augmented Reality setup that additionally allow users to interact with real-world objects through virtual proxies in the scene that monitor and visualize the state in the real-world and that can use services to change the state of the real world (e.g. switch lights on an off via a virtual button the 3D scene).

Scene

A *Scene* refers to a collection of objects, which are be identified by type (eg. a 3D mesh object, a physics simulation rigid body, or a script object.) These objects contain typed and named data values (composed of basic types such as integers, floating point numbers and strings) which are referred to as attributes. Scene objects can form a hierarchic (parent-child) structure. A HTML DOM document is one way to represent and store a scene.

Security

Security is a property of an IT system that ensures confidentiality, integrity, and availability of data within the system or during communication over networks. In the context of middleware it refers to the ability of the middleware to guarantee such properties for the communication channel according to suitably expressed requirements needed and guarantees offer by an application.

Security Policy

Security Policy refers to rules that need to be fulfilled before a network connection is established or for data to be transferred. It can for example express statements about the identity of communication partners, properties assigned to them, the confidentiality measures to be applied to data elements of a communication channel, and others.

Synchronization

Synchronization is the act of transmitting over a network protocol the changes in a scene to participants so that they share a common, real-time perception of the scene. This is crucial to eg. implementing multi-user virtual worlds.

Type Description

Type Description in the context of the AMi middleware refers to the internal description of native data types or the interfaces described by an IDL. It contains data such as the name of a veriable, its data type, the hierarchical relations between types (e.g. structs and arrays), its memory offset and alignment within another data type, and others. Type Description are used to generate the mapping of native data types to the data that needs to be transmitted by the middleware.

Virtual Character

Virtual Character is a 3D object, typically composed of triangle mesh geometry, that can be moved and animated and can represent an user's presence (avatar) in a virtual world. Typically supported forms of animation include skeletal animation (where a hierarchy of "bones" or "joints" controls the deformation of the triangle mesh object) and vertex morph animation, where the vertices of the triangle mesh are directly manipulated. Virtual character systems may support composing the character from several mesh parts, for example separate upper body, lower body and head parts, to allow better customization possibilities.

20 2D/3D Capture Open API Specification

20.1 RESTful and JavaScript API

The programming API is split in two at this point of development. The split is not final in any form, and primarily serves as a testing ground which allows gathering of material for further processing. The first part of the API is build for the browser as a JavaScript library. The library allows web developer to query capture interfaces, as well as capture data from them in a way it is harmonized across different browsers. The API functionality has been tested on both mobile and desktop browsers. The second part of the API is a RESTful interface for saving and collecting sensor data. At the moment the API allows saving if captured image files with advanced GeoTag information. The aim of it is twofolder: first, it serves as a network service where the device can push data and second, it acts as a datastorage to more advanced processing backends, such as 3D construction (not implemented yet)

20.1.1 JavaScript API

This API is based on the following W3C specifications(geoAPI,OrienationAPI) and current implementation does not assume harmonization of this API through out the browser spectrum. Hence isXXXXSupported set of functions.These functions avoid run-time exceptions due to unsupported W3C functions.

```
 isDeviceOrientationSupported: Returns true if the browser is able to
obtain calculated orientation readings from the accelerometer.

 isDeviceMotionSupported: Returns true if the browser is able to obtain
readings from the accelerometer.

 isGeolocationSupported: Returns true if the browser is able to obtain
readings from the GPS sensors.

 hasMediaSupport: Returns true if the browser is able to obtain camera
feed from the local camera to the video element
```

Following set of functions accesses the interfaces.

- showVideo()

Start streaming video from the device camera to a available Video element in the DOM.

```
   function
registerForDeviceMovements(onLocationSuccess,onLocationError,onMotion  ,
options)
```

If the location is found onLocationSuccess is triggered in case of a failure onLocationError function is triggered. Speed is calculated using GPS and onMotion function updates the speed for every GPS update received.

- getCurrentLocation()

Returns the GPS coordinates of the current location.

```
function getCurrentLocation (callback,options)
```

On successful retrieval of current location callback function is triggered.

- registerDeviceMotionEvents()

Registers to device motion data from the accelerometer. On successful event retrieval handlacceleration, handleAccelerationWithGravity, handleRotation callback functions are triggered respectively. Values are averaged over 2 past values to avoid errors and simple error correction methods are used to avoid sudden value changes.

```
function registerDeviceMotionEvents(handlacceleration,
handleAccelerationWithGravity, handleRotation)
```

- registerDeviceOrentationEvent()

Registers to device orientation data and calls eventHandlingFunction in case of device orientation is changed in along x, y and x axis.

```
function registerDeviceOrentationEvent(eventHandlingFunction)
```

Contemporary major browsers do not provide access to compass or the gyroscope. This values are calculated from the accelerometer. Device dependent Orientation API is implemented to adjust these sensor values to 3D contexts.

- registerAmbientLightChanges()

```
function registerAmbientLightChanges(handleLightValues)
```

- subscribe()

This function subscribes to receive data from all of the above functions are call back functions are provided to obtain data

```
function
subscribe(onLocationSearchSuccess,onLocationServiceSearchError,onMotion
,  handlacceleration,handleAccelerationWithGravityEvent,handleRotation,
handleOrientationChanges)
```

Existing implementation depends on WebSocket API media transfer and json ans a standard for massaging and streaming.

- snapshot()

This functions takes a snapshot of the current video feed and appends on the webpage.

- sendImage()

Uploads an Image to a designated server using web sockets. Depending on the server setup up and running.

- postImage()

Uploads an Image to a designated server using REST POST.

Following code snippet can be used to take a image and then upload the image with necessary data.

```
   dAPI = new FIware_wp13.Device("localhost","remote 2d3dCapture
server", "local server port" , "WebSocketPORT","LG_stereoscopic" ,"REST
server Port");

   dAPI.subscribe(onLocationSearchSuccess,
onLocationServiceSearchError, onMotion, handlacceleration,
handleAccelerationWithGravityEvent, handleRotation,
handleOrientationChanges);

   function upladlImageWithPost(){

        dAPI.snapshot();

        dAPI.postImage();

   }
```

User And Programmer Guide provides detailed instructions on how to update the APIs and how to use them.

20.1.2 RESTFull API (PRELIMINARY)

The RESTful image storage server API is not included because is highly experimental state, and publishing at this point will cause only confusion. At the same time the image server as such will not be a final deliverable anyway, but a part of larger system, hence the API is not yet here.

- http://XXXXXXX:XXXX/startwebsocketserver

Starts the websocket server

- http://XXXXXXX:XXXX/closewebsocketserver

Close Web socket Server

- http://XXXXXXX:XXXX/postBinaryImage

Post an Image to be saved in the repository

- http://XXXXXXX:XXXX/getAllImageData

Retrieve all the data in the database

- http://XXXXXXX:XXXX/getLocationImageData'

Get Images that are closest to a given GPS location

21 FIWARE OpenSpecification MiWi AugmentedReality

Name	FIWARE.OpenSpecification.MiWi.AugmentedReality
Chapter	Advanced Middleware and Web-based UI,
Catalogue-Link to Implementation	AugmentedReality
Owner	University of Oulu, Antti Karhu

21.1 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.org and similar pages in order to understand the complete context of the FI-WARE project.

21.2 Copyright

- Copyright © 2013 by University of Oulu

21.3 Legal Notice

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

21.4 Overview

The goal of the Augmented Reality Generic Enabler is to provide a high-level application programming interface, which can be used in developing various kinds of HTML5 Augmented Reality applications, which run on web browsers without any specific plug-ins. These applications rely on the functionality of the other GEs, like XML3D Technology, POI Data Provider, etc. Such AR applications will provide additional virtual content on top of the real world surroundings, and enhance the user's view of the physical world with information that is not directly visible.

21.4.1 Target Usage

The anticipated use-cases are diverse, but the basic functionality in typical use-cases will be similar. With the camera and sensors in a smartphone and tablet devices, the future applications would use the Augmented Reality GE to capture 2D/3D video streams of the environment, determine the location and pose of the user in this environment, and embed virtual content on top of it. In more basic use-cases, augmented reality applications will overlay simple head up displays, images or text into the user's field of view. In more complex use-cases augmented reality applications will display sophisticated 3D models rendered in such a way that they are blended into the surrounding of the natural scene, appearing indistinguishable from it. The AR interface will focus on the less complex use-cases, and the virtual

content will most likely be overlay objects rendered over video feed, because real-time registration and tracking of the real world is computationally expensive. Especially in the case of more complex vision based techniques. Although the computational power of mobile devices have greatly increased, mobile web browsers are still lacking in computationally intensive tasks. Furthermore, sensor data needed in location-based techniques must be highly accurate.

21.4.2 Example Scenario

One use-case could be, that AR application searches nearby restaurants from remote service (POI Data Provider), fetches the locations of restaurants and displays information like restaurant menu, opening hours, user reviews and so on.

21.5 Basic Concepts

21.5.1 3rd Party Web Service

3rd Party Web Service provide the actual content (text, images, 3D models, etc) displayed by the augmented reality application. The content resides on servers and can be queried using their specific APIs.

Related FI-WARE generic enablers:

- FIWARE.OpenSpecification.MiWi.GISDataProvider
- FIWARE.OpenSpecification.MiWi.POIDataProvider
- FIWARE.OpenSpecification.MiWi.2D-3D-Capture

21.5.2 Registration and Tracking

Registration and tracking is the process of aligning a virtual object with a real scene in the three-dimensional space. For smartphone and tablet device applications, object tracking involves either location sensors or an image recognition system or a combination of the two. Mobile augmented reality techniques can be roughly classified into two categories based on the type of registration and tracking technology used: location-based and vision-based.

21.5.2.1 *Location Based*

Location based techniques determine the location and orientation of a mobile device using sensors such as GPS, compass, gyroscope and accelerometer, and then overlay the camera display with information relevant to the user's location or direction. This information is usually obtained from remote services, which provide location based information.

21.5.2.2 *Vision Based*

Vision based techniques try to obtain information of the shape and location of the real world objects in the environment, using image processing techniques or predefined markers. The information is then used to align virtual content in the real world surroundings. These techniques may be subdivided into two main categories: Marker based and markerless tracking.

21.5.2.2.1 *Marker Based Tracking*

Markers create a link between the real physical world and the virtual world. Once a marker is found in a video frame, it is possible to extract its position, pose and other properties. After that the marker properties can be used to render some virtual object upon it. These markers are usually simple monochrome markers, which can be detected easily using less complex image processing algorithms.

1. Marker: A fiducial marker is defined as a square box that is divided into a grid. The outside cells of the grid are always black and the inside cells can be either white or black. The inside cells are used to encode data in binary form.

2. Image Marker: An image marker is very similar to the above defined marker with the difference being that the resolution of the grid is a bit larger and the inside cells are selected to form a shape or logo.

21.5.2.2.2 *Markerless Tracking*

Markerless tracking tracks features and/or predefined images or objects from the environment instead of fiducial marks. The specific features are recognized by image feature analysis. Markerless tracking is more flexible, because any part of the real environment may be used as a target that can be tracked in order to place virtual objects.

21.5.2.2.3 *Alvar*

Alvar is an augmented reality software library developed by VTT Technical Research Centre of Finland. The main functionality of Alvar is to detect predefined markers from input video data and to return information (transformation and visibility) about the found markers as a response.

21.5.3 Rendering

In basic computer graphics, the virtual scene is projected on an image plane using a virtual camera and this projection is then rendered on screen. In augmented reality the virtual content is rendered on top of the camera image. Therefore, the virtual camera has to be identical to the device's real camera, hence the optical characteristics of the virtual and real camera must be similar. This guarantees that the virtual objects in the scene are projected in the same way as real world objects.

Related FI-WARE generic enablers:

- FIWARE.OpenSpecification.MiWi.3D-UI

21.6 Generic Architecture

The Augmented Reality GE architecture consist of the following components:

- Location-based tracking and registration: The Location-based Tracking and Registration component is used to sense relevant information from the real-world surroundings. This is realized by combining the following W3C specifications: Geolocation, DeviceOrientation, DeviceLight, DeviceProximity into the HTML Sensor Events component. Geolocation API is used to locate the mobile device's location. DeviceOrientation API is used to sense the amount the device

is leaning side-to-side, front-to-back and the direction the device is facing. DeviceProximity API is used to sense the distance between the mobile device and a nearby physical object. DeviceLight API is used to to sense the light intensity level in the surrounding environment.

- Vision-based Registration and Tracking: The Vision-based Registration and Tracking component is used to detect markers from a video stream obtained from a mobile device's camera. This is realized by combining the HTML MediaStream interface, ALVAR and Xflow. The HTML MediaStream interface provides access to a video stream from a local camera. The video stream is used as a input to JavaScript version of ALVAR, which detects predefined markers and returns information (transformation and visibility) about the discovered markers. At the moment the functionality of the JavaScript version is limited to marker-based tracking (1) Markers and (2) Image Markers. ALVAR functionality is encapsulated into a Xflow dataflow node. Hence the ALVAR can be easily placed into the DOM structure.

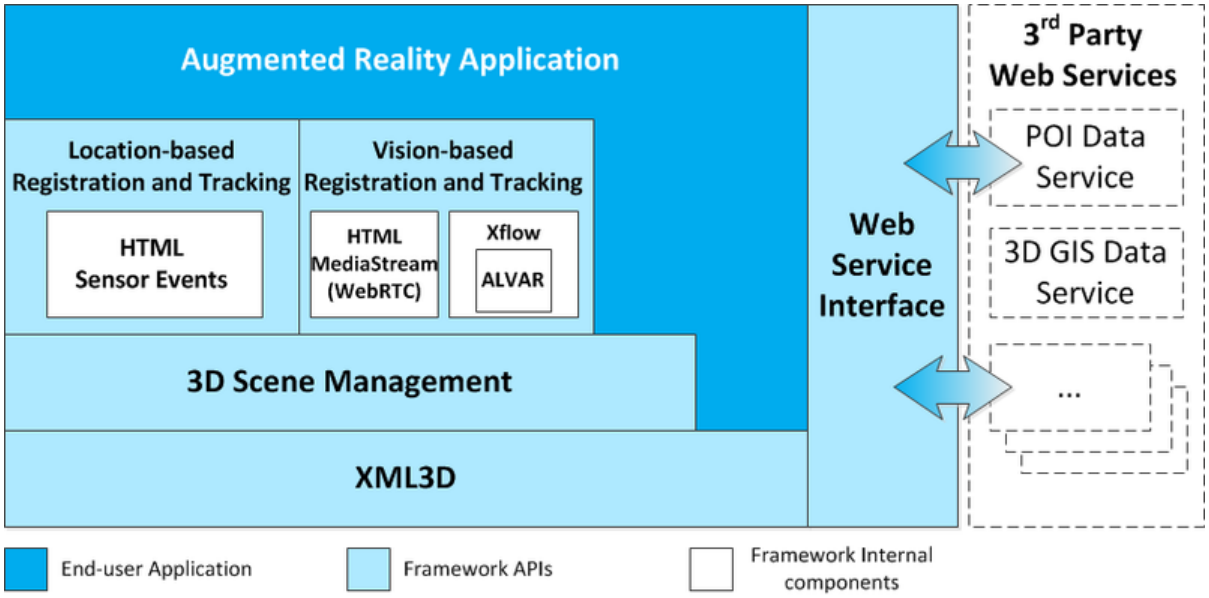
- 3D Scene Management: The 3D Scene Management component is an interface between the data obtained from the previous two components and XML3D. For example, a mobile web AR application can use the orientation of a mobile device to manipulate the virtual camera, or use the detected markers and GPS coordinates for placing 3D virtual objects in the 3D scene. However, the data obtained from Location- and Vision-based Registration and Tracking components is not in a form that XML3D can use directly. The GPS location is provided in terms of a latitude, longitude pair on the WGS84 coordinates system. The device orientation is provided in terms of a set of intrinsic Tait-Bryan angles of type Z-X'-Y'. The marker transform is provided in terms of 4x4 transform matrix. In order to use this information it must be converted into a right-handed, three-dimensional Cartesian coordinate system used by XML3D. The 3D Scene Management provides this exact functionality.

- XML3D: The XML3D component contains the representation of the virtual 3D scene and is responsible for rendering it. Detailed description of XML3D can be found here: FIWARE.OpenSpecification.MiWi.3D-UI. However, XML3D is not natively supported in browsers, hence it is emulated by using xml3d.js, which is a Polyfill implementation of XML3D.

- Web Service Interface: The Web Service Interface component manages the communication between a mobile web AR application and the 3rd Party Web Services using standard web technologies. The communication takes typically place either using RESTful API over HTTP, or through a WebSocket, for instance, when the requested data has to be streamed at high frequency. The Web Service Interface component requires that the 3rd Party Web Services are using the JSON format to represent the provided data.

- 3rd Party Web Services: The 3rd Party Web Services provide the content, such as text, photos, videos, audio and 3D objects, displayed in a mobile web AR application. The content usually comprises spatial information like GPS coordinates which enables the content to be retrieved and visualized correctly in a mobile web AR application. In example, the 3rd Party Web Service might be a POI or a 3D GIS Data Service, which provide functionality for searching content from a specific location based on GPS coordinates.

The API architecture is modular and each of its components is independent. Therefore, one can use only the components needed in a specific augmented reality application.



21.7 Main Interactions

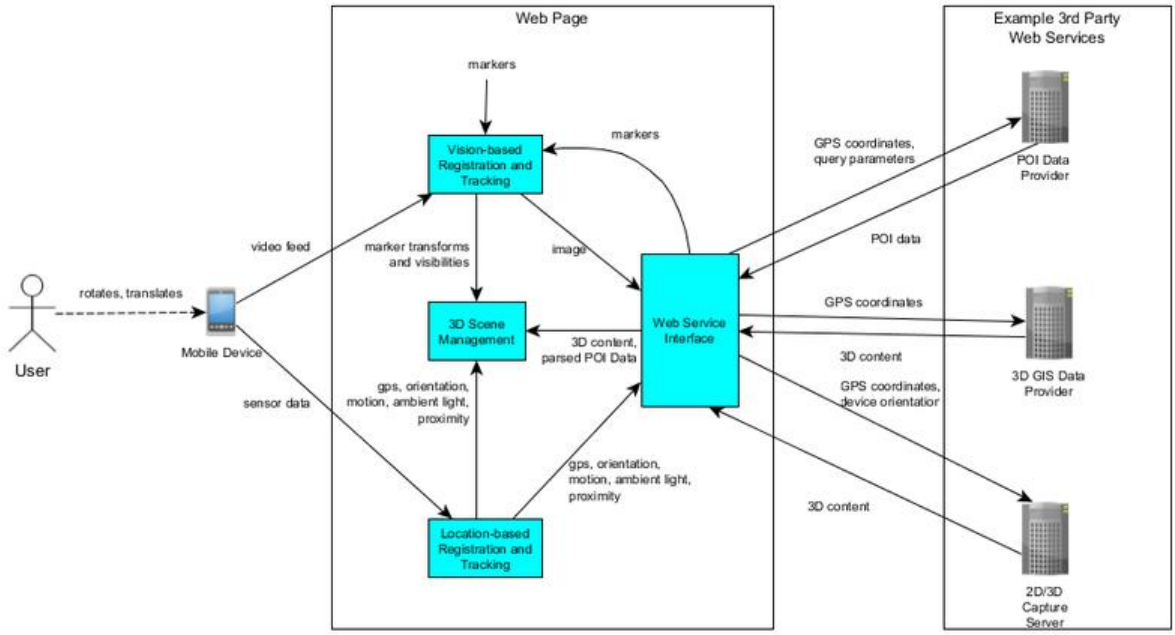


Figure describes the flow of data, both between the AR application components, and between the components and some 3rd party web services. In a basic example case the GPS coordinates is used to query nearby POIs from the POI Data Provider via the Web Service Interface. Only one POI is returned, it includes POI's physical location in wgs84 format and an 3D icon model. First the icon's location in the virtual scene is calculated, and then the icon is added in the scene, both is done by using the 3D scene management. Next, the user rotates the device and the Location-based Registration and Tracking component uses the 3D Scene Management component to pan the virtual camera according to the device orientation. Finally, when the device is facing the physical location of the POI, user sees the icon on the device screen.

21.8 Basic Design Principles

- AR applications should run directly on modern web browsers, with no need for plug-ins.
- Modular open source architecture.

21.9 Detailed Specifications

Following is a list of Open Specifications linked to this Generic Enabler. Specifications labeled as "PRELIMINARY" are considered stable but subject to minor changes derived from lessons learned during last interactions of the development of a first reference implementation planned for the current Major Release of FI-WARE. Specifications labeled as "DRAFT" are planned for future Major Releases of FI-WARE but they are provided for the sake of future users.

21.9.1 Open API Specifications

- Augmented Reality Open API Specification

21.10 Re-utilised Technologies/Specifications

- ALVAR
- HTML5 Geolocation
- HTML5 DeviceOrientation
- XML3D

Related GEs:

- FIWARE.OpenSpecification.MiWi.GISDataProvider
- FIWARE.OpenSpecification.MiWi.POIDataProvider
- FIWARE.OpenSpecification.MiWi.2D-3D-Capture

21.11 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

Annotations

Annotations refer to non-functional descriptions that are added to declaration of native types, to IDL interface definition, or through global annotations at deployment time. The can be used to express security requirements (e.g. "this string is a password and should be handled according the security policy defined for password"), QoS parameters (e.g. max. latency), or others.

AR

AR → Augmented Reality

Augmented Reality (AR)

Augmented Reality (AR) refers to the real-time enhancement of images of the real world with additional information. This can reach from the rough placement of 2D labels in the image to the perfectly registered display of virtual objects in a scene that are photo-realistically rendered in the context of the real scene (e.g. with respect to lighting and camera noise).

IDL

IDL → Interface Definition Language

Interface Definition Language

Interface Definition Language refers to the specification of interfaces or services. They contain the description of types and function interfaces that use these types for input and output parameters as well as return types. Different types of IDL are being used including CORBA IDL, Thrift IDL, Web Service Description Language (WSDL, for Web Services using SOAP), Web Application Description Language (WADL, for RESTful services), and others.

Middleware

Middleware is a software library that (ideally) handles all network related functionality for an application. This includes the setup of connection between peers, transformation of user data into a common network format and back, handling of security and QoS requirements.

PoI

PoI → Point of Interest

Point of Interest (PoI)

Point of Interest refers to the description of a certain point or 2D/3D region in space. It defines its location, attaches meta data to it, and defines a coordinate system relative to which additional coordinate systems, AR marker, or 3D objects can be placed.

Quality of Service (QoS)

Quality of Service refers to property of a communication channel that are non-functional, such a robustness, guaranteed bandwidth, maximum latency, jitter, and many more.

Real-Virtual Interaction

Real-Virtual Interaction refers to Augmented Reality setup that additionally allow users to interact with real-world objects through virtual proxies in the scene that monitor and visualize the state in the real-world and that can use services to change the state of the real world (e.g. switch lights on an off via a virtual button the 3D scene).

Scene

A *Scene* refers to a collection of objects, which are be identified by type (eg. a 3D mesh object, a physics simulation rigid body, or a script object.) These objects contain typed and named data values (composed of basic types such as integers, floating point numbers and strings) which are referred to as attributes. Scene objects can form a hierarchic (parent-child) structure. A HTML DOM document is one way to represent and store a scene.

Security

Security is a property of an IT system that ensures confidentiality, integrity, and availability of data within the system or during communication over networks. In the context of middleware it refers to the ability of the middleware to guarantee such properties for the communication channel according to suitably expressed requirements needed and guarantees offer by an application.

Security Policy

Security Policy refers to rules that need to be fulfilled before a network connection is established or for data to be transferred. It can for example express statements about the identity of communication partners, properties assigned to them, the confidentiality measures to be applied to data elements of a communication channel, and others.

Synchronization

Synchronization is the act of transmitting over a network protocol the changes in a scene to participants so that they share a common, real-time perception of the scene. This is crucial to eg. implementing multi-user virtual worlds.

Type Description

Type Description in the context of the AMi middleware refers to the internal description of native data types or the interfaces described by an IDL. It contains data such as the name of a veriable, its data type, the hierarchical relations between types (e.g. structs and arrays), its memory offset and alignment within another data type, and others. Type Description are used to generate the mapping of native data types to the data that needs to be transmitted by the middleware.

Virtual Character

Virtual Character is a 3D object, typically composed of triangle mesh geometry, that can be moved and animated and can represent an user's presence (avatar) in a virtual world. Typically supported forms of animation include skeletal animation (where a hierarchy of "bones" or "joints" controls the deformation of the triangle mesh object) and vertex morph animation,

where the vertices of the triangle mesh are directly manipulated. Virtual character systems may support composing the character from several mesh parts, for example separate upper body, lower body and head parts, to allow better customization possibilities.

22 Augmented Reality Open API Specification

22.1 JavaScript Library

The augmented reality GE will provide a JavaScript library, which implements the components specified in the Generic Architecture part. The JavaScript library includes a programming API for each component and a framework for creating the needed components.

This is a description of the JavaScript programming APIs.

22.1.1 Framework

- start()

  Starts the framework, and gets the video feed.

- createSensorManager()

  Creates Sensor API object, which implements the Location-based tracking and registration component.

- createARManager()

  Creates AR API, which implements the Vision-based Registration and Tracking component.

- createSceneManager()

  Creates the Scene API, which implements the 3D Scene Management component.

- createConnection()

  Creates the Connection API, which implements the Web Service Interface component.

22.1.2 Sensor API

Sensor API is used for creating sensor listeners. The Sensor API is based on the following W3C specifications Geolocation, DeviceOrientation, DeviceLight, DeviceProximity.

The supported sensor types:

```
SensorType
   orientation
   motion
   light
   proximity
```

- getAvailableSensors()

Returns an array of available sensor types.

- getSensorListeners()

  Returns a dictionary of currently active sensor listeners.

- listenSensor(sensorType)

  Returns sensor listener for the given sensor type.

- hasGPS()

  Returns true if the device has a GPS sensor.

- getCurrentPosition(successCallback, errorCallback, options)

  Attaches the given callback functions to "one-shot" position request. Uses the HTML5 Geolocation API getCurrentPosition() method to get the device's position.

- watchPosition(successCallback, errorCallback, options)

  Attaches the given success callback function to updated position as the device moves. Uses the HTML5 Geolocation API getCurrentPosition() method to get the device's position updates.

22.1.3 AR API

AR API is used for registering and tracking for three kind of markers: 3x3 fiducal markers, 5x5 fiducal markers and image markers.

To use the Xflow interface of Alvar JavaScript, the following code listing must be added inside the xml3d tag into HTML document.

```
 <data id="MarkerDetector"
     compute="Marker5x5Transforms, Marker3x3Transforms,
imageMarkerTransforms,
     Marker5x5Visibilities, Marker3x3Visibilities,
imageMarkerVisibilities, perspective
     = detect(arvideo, Marker5x5, Marker3x3, imageMarkers,
allowedImageMarkerErrors, flip)">
         <bool name="flip">false</bool>
         <int name="allowedImageMarkerErrors"></int>
         <int name="imageMarkers"></int>
         <int name="Marker3x3"></int>
         <int name="Marker5x5"></int>
         <texture name="arvideo">
             <video autoplay="false"></video>
```

```
        </texture>
</data>
```

All input parameters for the detect Xflow operator are optional meaning that the parameters' XHTML elements must exist but the element values can be empty.

arvideo	The type of this parameter is XML3D texture and in this case should contain video stream from devices local camera.
Marker5x5	The type of this parameter is XML3D int and it must contain zero or more IDs of ALVAR JavaScript's built-in 5x5Markers in a whitespace-separated list. These markers can be created with a separate marker creator which can be downloaded from [1] as a part of the ALVAR library.
Marker3x3	The type of this parameter is XML3D int and it must contain zero or more IDs of ALVAR JavaScript's built-in 3x3Markers in a whitespace-separated list. The IDs of these markers are between values of 0 and 63.
imageMarkers	The type of this parameter is XML3D int and it must contain zero or more custom image marker IDs in a whitespace-separated list. The IDs can be same as ALVAR JavaScript's built-in marker IDs. The only limitation for the image markers is that they need to be square and contain a black border. Testing the suitable border width and image marker content is up to the application developer, since these properties depend on the nature of the application and the usage environment. The marker image is converted to grayscale format in ALVAR JavaScript so the color information is discarded.
allowedImageMarkerErrors	The type of this parameter is XML3D int and it must contain zero or more whitespace-separated values of allowed errors to be used in image marker detection. Each value is related to a value with the same position in the imageMarkers list. For example, the second value in the allowedImageMarkers is the error value for the second marker ID in the imageMarkers. Thus, the marker IDs, to which the application developer wants to define a custom error value, should be put in the beginning of the imageMarkers list. The error values are relative to the used image marker content complexity so the application developer should try out which error values suit best for different markers. The default value for an image marker error in ALVAR JavaScript is 0.625 times marker's width/height. This is used if all or some of the values are left empty.
flip	The type of this parameter is XML3D bool and its value can be either true or false, the default value being false. When the value is true, the resulting transform matrices from marker detection are flipped so that they respond to flipped camera feed.

- setMarkerCallback(callback)

Sets a callback function for detected markers, the function has six input parameters callBackFunction(Marker3x3Transforms, Marker5x5Transforms, imageMarkerTransforms, Marker3x3Visibilities, Marker5x5Visibilities, imageMarkerVisibilities).

- addMarker(markerId, markerType)

  Adds the given marker id, into the set of markers that Alvar tracks.

  markerType can be Marker3x3, Marker5x5, imageMarker.

22.1.4 Scene API

Scene API is used for manipulating the elements in a xml3d scene. The actual xml3d scene can be defined in the web page using tags such as, mesh, group, transform, view, shader, etc. More information about how to use the xml3d can be found here: XML3D Open API Specification

- setPositionFromGeoLocation(curLoc, elemLoc, xml3dElement, minDistance, maxDistance)

  Positions the given xml3dElement(virtual object) into the virtual scene by using the given parameters: curLoc is the current gps location of the device, elemLoc is the gps location of the xml3dElement, and the calculated distance is clamped between minDistance and maxDistance.

- setCameraOrientation(deviceOrientation)

  Replaces the existing orientation of the virtual camera with the given device orientation.

- translateCameraFromGps(curLoc, gpsPoint, maxStep)

  Translates the camera from current gps location to gpsPoint. The translation is discarded if the distance between the current and new location exceeds the maxStep.

- translateCameraFromMotion(deviceMotion)

  Translates the camera according to the acceleration from deviceMotion event.

- setCameraMotionTranslationStepSize(stepSize)

  The given step size value defines the resolution of the virtual camera movement.

- setCameraDegreesOfFreedom(heave, sway, surge, yaw, pitch, roll)

  The given Boolean parameters define the freedom of degrees that the virtual camera currently has.

  heave: allows virtual camera to move up and down.

  sway: allows virtual camera to move left and right.

  surge: allows virtual camera to move forward and backward.

  yaw: allows virtual camera to rotate around y-axis.

pitch: allows virtual camera to rotate around x-axis.

roll: allows virtual camera to rotate around z-axis.

- setTransformFromMarker(markerTransform, xml3dElement, rotateX)

  Sets the given marker transform to the given xml3dElement. if rotateX is true the given xml3dElement is rotated 90 degrees.

- setCameraVerticalPlane(degrees)

  Sets the camera vertical plane into the given input degrees.

- addObjectToBillboardSet(xml3dElement)

  Adds the given xml3dElement into a billboard set. Objects that belong to billboard set, are always facing towards the virtual camera.

- getActiveCamera()

  Returns the xml3d active view element.

- getDistance(gpsPoint1, gpsPoint2)

  Returns the distance (meters) and bearing (radians) between the given gps coordinates.

22.1.5 Communication API

Communication API is used for handling the basic communication with 3rd party services(Other GEs).

- addRemoteService(serviceName, sourceURL)

  Adds a new remote service with the given service name and url. Remote service, such as POI Data Provider, must provide a RESTful API for communication.

- queryData(serviceName, restOptions, successCallback, errorCallback)

  Builds the REST query based on the given REST options and sends XMLHttpRequest to the given remote service. If the query is successful, the success callback function will handle the remote service's response message.

- sendData(serviceName, message, succesCallback, errorCallback)

  Sends the given message to the given remote service.

- listenWebsocket(url)

  Opens a websocket and connects it to given url.

23 FIWARE OpenSpecification MiWi RealVirtualInteraction

Name	FIWARE.OpenSpecification.MiWi.RealVirtualIntegration
Chapter	Advanced Middleware and Web-based UI,
Catalogue-Link to Implementation	Real/Virtual Interaction
Owner	Cyberlightning Ltd., Tomi Sarni

23.1 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.org and similar pages in order to understand the complete context of the FI-WARE project.

23.2 Copyright

- Copyright © 2013 by Cyberlightning Ltd

23.3 Legal Notice

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

23.4 Overview

A key aspect of Augmented Reality is that virtual content is not just presented embedded within the context of the real world, but that it should also allow users to interact actively with real objects and the objects to provide input to the user. An example is a real object that visualizes its internal (non-visible) state overlaid on the real object using AR techniques. The user can then affect the real object by interacting with the virtual object using multi-touch gestures, which are relayed to the real object using services. Many of the necessary tools are available by the other topics in this GE (e.g. presentation of overlaid content, services to modify the state of the UI, services calls initiated from UI components). This higher-level GE deals with defining the aspects necessary to put them together and provide any necessary missing components and glue them together.

23.4.1 Target Usage

Real Virtual Interaction generic enabler (GE) provides means for connecting real world devices consisting of sensors and actuators in to augmented or virtual reality applications. Since the real world sensors and actuators are not complex enough to contain necessary logic to publish themselves outside their

immediate domain there needs to be a external service that is able to access these devices and to be able to share the access to other services and also directly to end-users. This service provides security, data base for storing history and offline data, scalability and other cloud-like features that make it easier for application and service developers to make use of the devices in various purposes. This GE also provides a practical prototype for publishing sensor and actuator information application developers derived from NGSI 9/10 format developed earlier in FI-WARE.

23.4.2 Example Scenario

An example scenario could involve an end-user with an augmented reality software that relies on a Point-of-Interest (POI) service to create meaningful content in the augmented reality. This POI service is among other things connected to Real Virtual Interaction (RVI) service. After obtaining the GPS location of the end-user the POI service gathers information regarding nearby sensors and actuators by using a initiating a spatial query to RVI service. The RVI service search the data base for suitable candidates and returns a list of sensors and actuators in a RESTful data format. The augmented reality software then displays these sensors and actuators for the end-user. For instance if there was a temperature sensor within the same room with the end-user the augmented reality software would display the temperature readings from that sensor for the user. Also if there would be a light switch actuator in the same room, the augmented reality software could display 3D image of a button to turn on/off the lights from that room.

23.5 Basic Concepts

23.5.1 Internet of Things (IoT) concept

The Internet of Things refers to uniquely identifiable objects and their virtual representations in an Internet-like structure. According to some estimates more than 30 billion devices will be wirelessly connected to the Internet of Things [1]. Equipping all objects in the world with minuscule identifying devices or machine-readable identifiers could transform daily life. For instance, business may no longer run out of stock or generate waste products, as involved parties would know which products are required and consumed. One's ability to interact with objects could be altered remotely based on immediate or present needs, in accordance with existing end-user agreements. IoT concept is being researched in EU by Internet of Things Europe [2].

23.5.2 Augmented Reality

Augmented reality is a concept where images from a camera are being augmented with virtual 3D models of objects or characters based on location and use of specially designed markers.

23.5.3 Request/Response

In this GE request/response offers other service developers to send HTTP POST/GET queries to real virtual interaction backend service to access and obtain sensor and actuator information for instance based on their geographic location.

23.5.4 Publish/Subscribe

WebSocket protocol provides a full-duplex connection between a browser run web application and real virtual interaction backend service[3].Websocket has been defined in IETF's [4]. In this GE WebSocket acts as an enabler for end-users to publish and subscribe to real time data flow with sensors and actuators.

23.5.5 RESTful Data Format

A JavaScript Object Notation (JSON) type of document that consist of parent-child relationships, and uses JSON Objects or JSON Arrays as building blocks.

23.6 Generic architecture

23.6.1 Device Entity

Represents end-point IoT devices in figure 1. In this GE the decision was to implement the connection between end-point devices and the back-end server using uniform datagram packets (UDP). UDP is often used in traffic where delivery is not guaranteed. Although it is possible to implement higher level logic to make sure that a sensor event was delivered using call back events. In default though sensor event delivery is not verified. UDP is also suitable in situations where there are a sensor events published in high frequency intervals, on such situation it is not critical to deliver all packets similar to video streams for instance. In this GE the device was implemented as an Android software application. Since there can exist upper limits for UDP packet size we have also included data compression technique to allow more information to be send in one packet. The device in this GE can consist of one or more sensors or actuators. The device itself contains communication logic and operating system.

23.6.2 Real Virtual Interaction Back-end Server

The back-end server consist of connection interfaces for clients and third party services. The interface in this contexts differs from device-to-server communication in a way that the server offers means of conversation between clients and other services and the server. The device-to-server is only a socket offering means of connection. By this we mean that it is up to the sensors to handle HTTP POST query for instance and it is up to the server to broker sensor events and parse the information and store it in a data base. The data base contains sensor and actuator information containing history of events. The information is being categorized in relational manner using UUIDs as data base keys to differentiate devices. The message handler component is the core server component responsible of maintaining subscription lists and brokering sensor events and actuator commands accordingly.

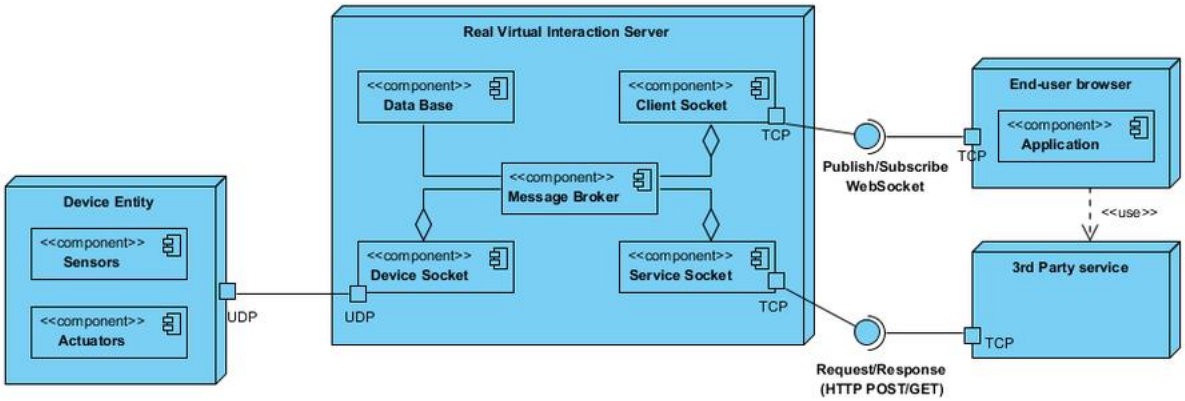


Figure 1: Deployment diagram representing the interaction between real world devices, real virtual interaction service, and 3rd party services and end-user client applications.

23.6.3 Third Party Services

With third party services this GE refer to cloud type of services that apply big data to create contexts suitable for their clients. A practical example in this GE is a POI service that connects sensors and actuators to points of interest based on geographic location of end-user using the POI service. This connection is handled using request/response type of asynchronous queries. A third party server may define a spatial query of a real world geographic area and the real virtual interaction server provides found sensors and actuators to the given search criteria. Alternatively third party services may broker commands to actuators from end-users connected to their service or offer direct access to the real virtual interaction server socket.

23.6.4 End-user Application

End-user application refers to users using some form web application or web service through a desktop or hand held device application. In this GE we provide sample client using Web Socket to subscribe to sensor events in real time.

23.7 Main Interactions

23.7.1 End-point Devices

With fairly recent advances in sensors and actuators technologies have made it possible for such devices to be applied in everyday usage due to lowered size, power consumption and mainly cost. Although most of the sensors apply their "awareness" on a personal area network (PAN) level around their location of installation, there exists a vision of IoT where such devices would be equipped with ability to connect to resources across Internet. It is worthwhile to be aware that often these devices contain lightweight microprocessors Bluetooth or similar connectivity thus not able to directly connect to Internet resources. In such cases they need a gateway and Mesh networking protocol to relay messages across their local domain to Internet resources. Also to conserve power many wireless sensors only publish information either on request basis or in specific intervals. Without existence of a server and a data base this information could be lost [6]. Also further down the topology

FIWARE.OpenSpecification.IoT.Gateway.DataHandling should be implemented to gateway device to store information in temporary buffers in case connections are lost to either back-end or to IoT devices and resend when connection is established. This component could also do some traffic managing in case the amount of sensor data blocks the network connections.

Since majority of devices containing raw sensors and actuators offer vendor specific and proprietary software solutions, in this GE we chose Android devices as a basis for testing implementation. Android devices contain both real sensors and actuators and ability to connect to Internet resources in variety of ways. Also Android devices are widely available and do not require purchasing a specific device for testing.

23.7.2 Real Virtual Interaction Back-end Service

Middle-ware IoT service is the core component of this GE. The purpose of this middle-ware service is to gather together devices containing sensors and actuators. The middle-ware service offers a data base component for storing necessary connection information and history or latest data published by a sensor. Since IP addresses often change the real virtual interaction server needs to maintain the information based on UUIDs [7]. The middle-ware service also provides a layer of security that allows publishing sensor data publicly in the Internet, but also to shield them from direct access. FIWARE.OpenSpecification.IoT.Backend.IoTBroker GE **states** that The IoT Broker is stateless in the sense that no context information is stored by it. Its role in the Internet of Things is not to serve as a central data repository, but as a central point which retrieves and aggregates data from various sources on behalf of applications. In Real Virtual Interaction GE this is being extended as virtual environments are highly context-defined and thus we are likely managing a particular set of IoT devices tied together by their type or geographic location for instance. Thus mere broker in this case is not enough to act between an end-user client and an IoT device. In this GE the main purpose is provide RESTful means for connecting devices with very limited logical capabilities for cloud type of services and directly to end-users. How this information is being applied by augmented reality software developers is not being tackled in this GE, we merely provide means for accessing actuators and receiving data from sensors.

23.7.3 Services and Applications Utilizing Device Data

Similar to FIWARE.OpenSpecification.I2ND.CDI GE, the aim Real Virtual Interaction GE is to provide access to devices that may contain multiple sensors and actuators or directly to an individual sensor. Changing state of a sensor may be resolved by sending a HTTP POST call request brokered by a middleware service to the particular sensor. This is the case in FIWARE.OpenSpecification.Data.PubSub GE where it presents a way for hand held devices to receive in a specific area by subscribing through Bluetooth devices and receiving information based on users location within indoor spaces. Considering that the end-user may be interested to see live feed from a sensor or be able to control a device in a way that requires real-time control of the device through remote interface, we will need to implement different method in respect to HTTP. Web Sockets offer this ability. In architectural representation shown in figure 1 suggestion is that a third party middle-ware service such as a point-of-interest (POI) type of service will request and parse the information to the end-user client interested about a specific POI. The third party middle-ware can then provide connection details to a web socket to a particular sensor or set of sensors.

23.7.4 RESTful Data Format

There needs to exist a format how devices publish information regarding sensors and actuators. Considering that the end-user client applications or third party services may not be aware of specifics regarding a sensor or an actuator, there needs to exist a format that contains necessary elements so that the data can be dynamically applied. Also there needs to be way to also publish what kind of call backs are available to verify that an actuator command has been implemented for instance. Also there needs to be a way to embed a parent context in situations where the device itself is not able to connect to Internet, but rather uses a gateway device for the connection. In this case the gateway device should embed its signature in parent-child relationship. Whether the RESTful format is embedded with gateway context there needs to be a "dataformat_version" key among the parent fields. The FI-WARE release is designated with version "1.0". The backend will drop packets that have different version or lack the field. The data format version number will not be stored nor passed to connected clients applications. The designed data format has been derived from FI-WARE NGSI-10 Open RESTful API Specification in to a modified JavaScript Object Notation (JSON) format.

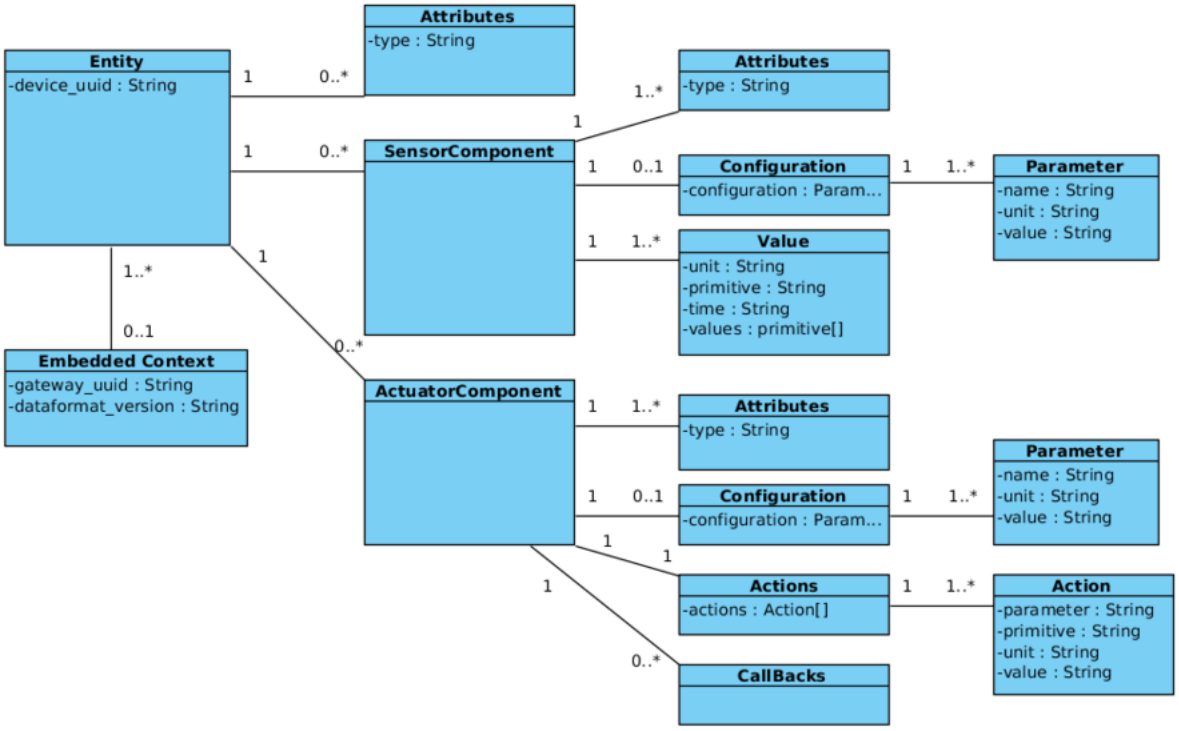


Figure 2: Class diagram depicting the structure of the data format which devices containing sensors and actuators can use to publish information.

Entity in this context represents a physical device that may contain zero or more sensors and zero or more actuators. It is expected that a device would at least contain either one sensor or one actuator. Entity device could be either thought as a independent device that has capability to send messages to Internet resources (real virtual interaction server) or at least transmit messages forward in local area network(LAN) or personal area network(PAN) by using RJ-45, Wi-Fi, Bluetooth or similar connection methods. In this particular case an another device that has an Internet connection and receives this Wi-Fi or Bluetooth message should wrap the RESTful data format with its context. This enables real virtual interaction server or similar to be aware that the device itself requires a broker gateway device in order to

transmit or received messages. Entities, sensors and actuators all can have attributes. These attributes are sorted in key-value pairs. For instance en entity may have two attributes: {"name" : "TI CC2540", "gps" : [65.1003, 25.3321]}. Although the API specification examples will propose few suggestion regarding what attributes should be included, the design itself supports dynamic key-value pairs. The "key" must be of String primitive data type. Value may be any primitive data type supported by JSON schema. Sensors and actuators may have configuration parameters. Parameter refers to a configurable variable. For instance, a sensor may have configurable variable for interval of sensor events in milliseconds. A parameter should contain the description of the configurable variable and what value options are available and SI unit of the values.

Each sensor may publish one or more value entries in a single RESTful data format string. Each value should at least have the following attributes: SI unit for the values, time stamp for the measurement, primitive data type of the values and one or more entries with same primitive data type. Value may contain additional key-value pairs. It is expected that a sensor only publishes one type of value, addition value entries are similar in type but may have different entries and time stamp. Actuators can have one or more Actions. In this way the actuators can publish to clients how to the action can be triggered. Action should publish name or type of the action, what values are available for the action and what SI unit they are in. Also an action should indicate the current state of the actuator. Callbacks are for verifying the send message is being handled. For instance a callback events could be simple ACK,CON, RST type of message to indicate new status or to confirm that the message was received.

23.8 Basic Design principles

- Modular open source back-end implementation that offers a RESTful API for application developers using JSON.
- Enable both asynchronous and synchronous subscription/publish methods for application developers by enabling HTTP/WEBSOCKET options.
- Work on Session, Presentation and Application layers of OSI model [8].

23.9 References

- 30 Billion devices connected to internet by 2020, ABI research 2013.
- Internet of Things Europe
- TCP, RFC793
- The WebSocket protocol RFC6455
- IEEE 802.15.4 WPAN specification
- Mesh networking
- UUID
- OSI reference model, wikipedia

23.10 Detailed Specifications

Following is a list of Open Specifications linked to this Generic Enabler. Specifications labeled as "PRELIMINARY" are considered stable but subject to minor changes derived from lessons learned during last interactions of the development of a first reference implementation planned for the current Major Release of FI-WARE. Specifications labeled as "DRAFT" are planned for future Major Releases of FI-WARE but they are provided for the sake of future users.

23.10.1 Open API Specifications

- Real Virtual Open API Specification

23.11 Re-utilised Technologies/Specifications

- FIWARE.OpenSpecification.IoT.Backend.IoTBroker
- FIWARE.OpenSpecification.IoT.Backend.DeviceManagement
- FIWARE.OpenSpecification.I2ND.CDI
- FIWARE.OpenSpecification.Data.PubSub
- FIWARE NGSI-10 Open RESTful API

23.12 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

Annotations

Annotations refer to non-functional descriptions that are added to declaration of native types, to IDL interface definition, or through global annotations at deployment time. The can be used to express security requirements (e.g. "this string is a password and should be handled according the security policy defined for password"), QoS parameters (e.g. max. latency), or others.

AR

AR → Augmented Reality

Augmented Reality (AR)

Augmented Reality (AR) refers to the real-time enhancement of images of the real world with additional information. This can reach from the rough placement of 2D labels in the image to the perfectly registered display of virtual objects in a scene that are photo-realistically rendered in the context of the real scene (e.g. with respect to lighting and camera noise).

IDL

IDL → Interface Definition Language

Interface Definition Language

Interface Definition Language refers to the specification of interfaces or services. They contain the description of types and function interfaces that use these types for input and output parameters as well as return types. Different types of IDL are being used including CORBA IDL, Thrift IDL, Web Service Description Language (WSDL, for Web Services using SOAP), Web Application Description Language (WADL, for RESTful services), and others.

Middleware

Middleware is a software library that (ideally) handles all network related functionality for an application. This includes the setup of connection between peers, transformation of user data into a common network format and back, handling of security and QoS requirements.

PoI

PoI → Point of Interest

Point of Interest (PoI)

Point of Interest refers to the description of a certain point or 2D/3D region in space. It defines its location, attaches meta data to it, and defines a coordinate system relative to which additional coordinate systems, AR marker, or 3D objects can be placed.

Quality of Service (QoS)

Quality of Service refers to property of a communication channel that are non-functional, such a robustness, guaranteed bandwidth, maximum latency, jitter, and many more.

Real-Virtual Interaction

Real-Virtual Interaction refers to Augmented Reality setup that additionally allow users to interact with real-world objects through virtual proxies in the scene that monitor and visualize the state in the real-world and that can use services to change the state of the real world (e.g. switch lights on an off via a virtual button the 3D scene).

Scene

A *Scene* refers to a collection of objects, which are be identified by type (eg. a 3D mesh object, a physics simulation rigid body, or a script object.) These objects contain typed and named data values (composed of basic types such as integers, floating point numbers and strings) which are referred to as attributes. Scene objects can form a hierarchic (parent-child) structure. A HTML DOM document is one way to represent and store a scene.

Security

Security is a property of an IT system that ensures confidentiality, integrity, and availability of data within the system or during communication over networks. In the context of middleware it refers to the ability of the middleware to guarantee such properties for the communication channel according to suitably expressed requirements needed and guarantees offer by an application.

Security Policy

Security Policy refers to rules that need to be fulfilled before a network connection is established or for data to be transferred. It can for example express statements about the identity of communication partners, properties assigned to them, the confidentiality measures to be applied to data elements of a communication channel, and others.

Synchronization

Synchronization is the act of transmitting over a network protocol the changes in a scene to participants so that they share a common, real-time perception of the scene. This is crucial to eg. implementing multi-user virtual worlds.

Type Description

Type Description in the context of the AMi middleware refers to the internal description of native data types or the interfaces described by an IDL. It contains data such as the name of a veriable, its data type, the hierarchical relations between types (e.g. structs and arrays), its memory offset and alignment within another data type, and others. Type Description are used to generate the mapping of native data types to the data that needs to be transmitted by the middleware.

Virtual Character

Virtual Character is a 3D object, typically composed of triangle mesh geometry, that can be moved and animated and can represent an user's presence (avatar) in a virtual world. Typically supported forms of animation include skeletal animation (where a hierarchy of "bones" or "joints" controls the deformation of the triangle mesh object) and vertex morph animation, where the vertices of the triangle mesh are directly manipulated. Virtual character systems may support composing the character from several mesh parts, for example separate upper body, lower body and head parts, to allow better customization possibilities.

24 Real Virtual Open API Specification

This GE will provide two APIs for accessing sensors or actuators:

- API for service developers
- API for application developers

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119]. The key word REMOTEHOST can be replaced by either IP address or DNS host name. Also port number for middleware service can differ from IANA standard which is 80 for the WebSocket and 8080 for HTTP. The key words GET,POST,PUT and DELETE are http methods and appear capitalized each time they occur in the specifications.

Security implementations are not included in this specifications as they are highly dependable on type of middleware service and chosen security level. For controlled public access api-keys or session-ids could be used. Alternatively for private access login information could be included in queries.

24.1.1 API examples for device application developers

The Realvirtualinteraction backend will listen to incoming UDP packets and will drop packets that do not conform to the RESTful data format specification (version 1.0). The payload string MAY be Gzip compressed. Below JSON string is an example how the device developers should public sensor/actuator information to the server. The "dataformat_version" field will be removed after the packet is being received by the server and will not be passed on to possible clients subscribed listening for incoming events. For instance the existing logic could be extended to include other fields such as API-KEY to ensure that only registered devices may publish to server. This could be the first step to add a layer of security.

```
{
    "dataformat_version": "1.0",
    "d23c058698435eff": {
        "d23c058698435eff": {
            "sensors": [
                {
                    "value": {
                        "unit": "uT",
                        "primitive": "3DPoint",
                        "time": "2014-02-19 09:40:06",
                        "values": [
                            17.819183349609375,
```

```
                    0.07265311479568481,
                    -0.4838427007198334
                ]
            },
            "configuration": [
                {
                    "interval": "ms",
                    "toggleable": "boolean"
                }
            ],
            "attributes": {
                "type": "orientation",
                "power": 0.5,
                "vendor": "Invensense",
                "name": "MPL magnetic field"
            }
        },
        {
            "value": {
                "unit": "uT",
                "primitive": "3DPoint",
                "time": "2014-02-19 09:40:06",
                "values": [
                    17.819183349609375,
                    0.07265311479568481,
                    -0.4838427007198334
                ]
            },
            "configuration": [
                {
                    "interval": "ms",
                    "toggleable": "boolean"
                }
            ],
```

```
                    "attributes": {
                        "type": "gyroscope",
                        "power": 0.5,
                        "vendor": "Invensense",
                        "name": "MPL magnetic field"
                    }
                },
                {
                    "value": {
                        "unit": "uT",
                        "primitive": "3DPoint",
                        "time": "2014-02-19 09:40:06",
                        "values": [
                            17.819183349609375,
                            0.07265311479568481,
                            -0.4838427007198334
                        ]
                    },
                    "configuration": [
                        {
                            "interval": "ms",
                            "toggleable": "boolean"
                        }
                    ],
                    "attributes": {
                        "type": "magneticfield",
                        "power": 0.5,
                        "vendor": "Invensense",
                        "name": "MPL magnetic field"
                    }
                },
                {
                    "value": {
                        "unit": "m/s2",
```

```
                    "primitive": "3DPoint",
                    "time": "2014-02-19 09:40:06",
                    "values": [
                        0.006436614785343409,
                        0.003891906701028347,
                        -0.5983058214187622
                    ]
                },
                "configuration": [
                    {
                        "interval": "ms",
                        "toggleable": "boolean"
                    }
                ],
                "attributes": {
                    "type": "linearacceleration",
                    "power": 1.5,
                    "vendor": "Google Inc.",
                    "name": "Linear Acceleration Sensor"
                }
            }
        ],
        "actuators": [
            {
                "configuration": [
                    {
                        "value": "100",
                        "unit": "percent",
                        "name": "viewsize"
                    }
                ],
                "actions": [
                    {
```

```
                                "value":
"[marker1,marker2,marker3,marker4,marker6,marker7,marker8,marker9,marke
r10,marker11,marker12,marker13,marker14,marker15,marker15,marker16,mark
er17,marker18,marker19]",
                                "primitive": "array",
                                "unit": "string",
                                "parameter": "viewstate"
                            }
                        ],
                        "callbacks": [
                            {
                                "target": "viewstate",
                                "return_type": "boolean"
                            }
                        ],
                        "attributes": {
                            "dimensions": "[480,800]"
                        }
                    }
                ],
                "attributes": {
                    "name": "Android device"
                }
            }
        }
}
```

24.1.2 API examples for application developers

Following code and header samples enable a real-time connection over TCP/IP to be formed with a server application. Once a connection is established, sensor events MAY be pushed to clients from server in real-time. The connection is full-duplex meaning that also a client MAY send messages directly to sensors in through a web server. This sort of full-duplex connection MAY be considered as publish/subscribe type of connection where client MAY choose which sensor to subscribe to receive event updates from. The web service SHALL provide the client a list of available sensors or OPTIONALLY a client MAY use third party service such as point-of-interest(POI) service to find sensors. WebSocket SHOULD be then used to form direct connection to the sensors through a IoT.Broker type of web server component.

JavaScript client sample:

```
function CreateWebSocket() {

      if ("WebSocket" in window) {
         ws = new WebSocket("ws://REMOTEHOST");

         ws.onopen = function() {
           alert("Connection established to web server");
         };
         ws.onmessage = function (evt) {
           alert("Message received from web server: " + evt.data);
         };
         ws.onclose = function() {
            alert("Connection is closed...");
         };
      }
      else {
         alert("WebSocket NOT supported by your Browser!");
      }
}
```

The above JavaScript example initiates a connection to a webserver and starts handshake with following HTTP header.

REQUEST HEADER:

```
GET /chat HTTP/1.1
Host: REMOTEHOST
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: LOCALHOST
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Server MUST respond with following HTTP header or handshake fails. Notice that the Sec-WebSocket-Accept key is unique and MUST be created by server instance. Detailed instructions can be found from RFC6455.

RESPONSE HEADER:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

24.1.3 API examples for service developers:

Following example show how backend services SHALL communicate between each other using HTTP GET/POST methods. OPTIONALLY other HTTP methods such as PUT and DELETE MAY be used, but they are not supported by the real virtual interaction backend deliverable.

24.1.3.1 *Request all sensors with bound by a specific spatial bounds*

A middleware web service SHOULD offer ways for other middleware services to specify retrievable devices by location and spatial bounds or OPTIONALLY by an IP address space. The spatial bound SHALL be either a square area with minimum and maximum values for coordinates, a circle with a centerpoint and radius or a complex shape.

Following example shows an example where POI middleware service requests all devices available within a specific circular area with a geo-coordinate center point and radius in meters.

Below is a sample code that can be used to form the following request query:

```
<form action="http://127.0.0.1:44446/" method="get">
<input type="hidden" name="action" value="loadById">
Device id: <input type="text" name="device_id"><br>
Maxresults: <input type="text" name="maxResults"><br>
<input type="submit" value="Submit">
</form>
```

Below is a sample request header as received by the real virtual interaction backend:

REQUEST HEADER:

```
GET    /?action=loadBySpatial&lat=65.4&lon=25.4&radius=1500&maxResults=1
HTTP/1.1
```

```
Host: 127.0.0.1:44446
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:25.0) Gecko/20100101 Firefox/25.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Below is a sample reponse header as send by the real virtual interaction backend.

RESPONSE HEADER:

```
 HTTP/1.1 200 OK
 Access-Control-Allow-Origin: *
 Content-Type: text/plain; charset=utf-8
 Content-Length: 1767

 Connection: close

{
    "8587cdb9a135fa2a": {
        "sensors": [
            {
                "values": [
                    {
                        "unit": "lx",
                        "time": "2014-02-17 14:35:53",
                        "values": 58.607452,
                        "primitive": "double"
                    }
                ],
                "attributes": {
                    "vendor": "Sharp",
                    "name": "GP2A Light sensor",
                    "power": 0.75,
                    "type": "light"
```

```
                },
                "configuration": [
                    {
                        "interval": "ms",
                        "toggleable": "boolean"
                    }
                ]
            },
            {
                "values": [
                    {
                        "unit": "uT",
                        "time": "2014-02-17 14:35:53",
                        "values": [
                            10.400009155273438,
                            -16.688583374023438,
                            -37.505584716796875
                        ],
                        "primitive": "3DPoint"
                    }
                ],
                "attributes": {
                    "vendor": "Invensense",
                    "name": "MPL Magnetic Field",
                    "power": 4,
                    "type": "magneticfield"
                },
                "configuration": [
                    {
                        "interval": "ms",
                        "toggleable": "boolean"
                    }
                ]
            },
```

```
            {
                "values": [
                    {
                        "unit": "hPa",
                        "time": "2014-02-17 14:35:53",
                        "values": 989.56,
                        "primitive": "double"
                    }
                ],
                "attributes": {
                    "vendor": "Bosch",
                    "name": "BMP180 Pressure sensor",
                    "power": 0.6700000166893005,
                    "type": "pressure"
                },
                "configuration": [
                    {
                        "interval": "ms",
                        "toggleable": "boolean"
                    }
                ]
            }
        ],
        "attributes": {
            "name": "Android device"
        },
        "actuators": [
            {
                "callbacks": [
                    {
                        "target": "viewstate",
                        "return_type": "boolean"
                    }
                ],
```

```
                "attributes": {
                    "dimensions": "[720,1184]"
                },
                "configuration": [
                    {
                        "unit": "percent",
                        "name": "viewsize",
                        "value": "100"
                    }
                ],
                "actions": [
                    {
                        "unit": "string",
                        "parameter": "viewstate",
                        "value":
"[marker1,marker2,marker3,marker4,marker6,marker7,marker8,marker9,marke
r10,marker11,marker12,marker13,marker14,marker15,marker15,marker16,mark
er17,marker18,marker19]",
                        "primitive": "array"
                    }
                ]
            }
        ]
    }
}
```


The response header returns radius and geo-coordinates which were set by the original request query. *Devices* JSONArray object contains all devices matching the query. Each device MAY contain multiple sensors and actuators.

24.1.3.2 *Request all data from a specific device by device UUID*

A device SHOULD be considered as a micro controller board with capabilitis required to generate an *uuid*. A device MAY contain any number of sensors and actuators, and in any combination. If the requested sensor or actuator does not have uuid the request MUST target the device containing the desired sensor or actuator.

Following example shows how a middleware service retrieves all available information regarding a specific device by using an *uuid* string identifier.

Below is a sample code that can be used to form the following request query:

```
<form action="http://127.0.0.1:44446/" method="get">
<input type="hidden" name="action" value="loadBySpatial">
Latitude: <input type="text" name="lat"><br>
Longitude: <input type="text" name="lon"><br>
Radius: <input type="text" name="radius"><br>
Maxresults: <input type="text" name="maxResults"><br>
<input type="submit" value="Submit">
</form>

Below is a sample request header as received by the real virtual interaction backend.
```

REQUEST HEADER:

```
GET /?action=loadById&device_id=440cd2d8c18d7d3a&maxResults=1 HTTP/1.1
Host: 127.0.0.1:44446
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:25.0) Gecko/20100101 Firefox/25.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Below is a sample response header as send by the real virtual interaction backend.

RESPONSE HEADER:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Content-Type: text/plain; charset=utf-8
Content-Length: 1767

Connection: close
```

```
{
    "440cd2d8c18d7d3a": {
        "actuators": [
            {
                "configuration": [
                    {
                        "unit": "percent",
                        "name": "viewsize",
                        "value": "100"
                    }
                ],
                "callbacks": [
                    {
                        "return_type": "boolean",
                        "target": "viewstate"
                    }
                ],
                "attributes": {
                    "dimensions": "[480,800]"
                },
                "actions": [
                    {
                        "unit": "string",
                        "primitive": "array",
                        "parameter": "viewstate",
                        "value":
"[marker1,marker2,marker3,marker4,marker6,marker7,marker8,marker9,marke
r10,marker11,marker12,marker13,marker14,marker15,marker15,marker16,mark
er17,marker18,marker19]"
                    }
                ]
            }
        ],
        "sensors": [
            {
```

```
                "configuration": [
                    {
                        "toggleable": "boolean",
                        "interval": "ms"
                    }
                ],
                "values": {
                    {
                        "unit": "rads",
                        "primitive": "3DPoint",
                        "values": [
                            21.117462158203125,
                            -0.9801873564720154,
                            -0.6045787930488586
                        ],
                        "time": "2013-12-10 10:07:30"
                    }
                },
                "attributes": {
                    "vendor": "Invensense",
                    "name": "MPL Gyro",
                    "power": 0.5,
                    "type": "gyroscope"
                }
            },
            {
                "configuration": [
                    {
                        "toggleable": "boolean",
                        "interval": "ms"
                    }
                ],
                "values": {
                    {
```

```
                    "unit": "ms2",
                    "primitive": "3DPoint",
                    "values": [
                        149.10000610351562,
                        420.20001220703125,
                        -1463.9000244140625
                    ],
                    "time": "2013-12-10 10:07:30"
                }
            },
            "attributes": {
                "vendor": "Invensense",
                "name": "MPL accel",
                "power": 0.5,
                "type": "accelerometer"
            }
        },
        {
            "configuration": [
                {
                    "toggleable": "boolean",
                    "interval": "ms"
                }
            ],
            "values": {
                {
                    "unit": "uT",
                    "primitive": "3DPoint",
                    "values": [
                        -0.08577163517475128,
                        0.16211289167404175,
                        9.922416687011719
                    ],
                    "time": "2013-12-10 10:07:30"
```

```
                    }
                },
                "attributes": {
                    "vendor": "Invensense",
                    "name": "MPL magnetic field",
                    "power": 0.5,
                    "type": "magneticfield"
                }
            },
            {
                "configuration": [
                    {
                        "toggleable": "boolean",
                        "interval": "ms"
                    }
                ],
                "values": {
                    {
                        "unit": "orientation",
                        "primitive": "3DPoint",
                        "values": [
                            -0.004261057823896408,
                            -0.017044231295585632,
                            0.019174760207533836
                        ],
                        "time": "2013-12-10 10:07:30"
                    }
                },
                "attributes": {
                    "vendor": "Invensense",
                    "name": "MPL Orientation (android deprecated
format)",
                    "power": 9.699999809265137,
                    "type": "orientation"
```

```
                    }
                }
            ],
            "attributes": {
                "name": "Android device"
            }
        }
}
```

The above response shows a general description how a JSON object returned by the middleware service could look like.

24.1.3.3 *Change value of a specific attribute of a sensor by ID, controller name, new value*

Middleware service MUST offer a way to change state of sensors or actuators. Sensors and actuators SHOULD publish configurable parameters. Middleware services MUST send state change requests as HTTP POST calls. POST request content MUST start with action definition.

Following example shows how to use HTTP POST request to turn change augmented reality marker on an Android application remotely.

Below is a sample code that can be used to form the following request query:

```
<form action="http://127.0.0.1:44446/upload" enctype="multipart/form-data" method="post">
Device id: <input type="text" name="device_id"><br>
Choose marker to upload: <input type="file" name="datafile" size="40"></br>
<input type="submit" value="Send">
```

REQUEST HEADER:

```
POST / HTTP/1.1
Host: 127.0.0.1:44446
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:25.0) Gecko/20100101 Firefox/25.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

```
Content-Type: application/x-www-form-urlencoded
Content-Length: 92

action=update&device_id=440cd2d8c18d7d3a&sensor_id=display&parameter=vi
ewstate&value=marker5
```

RESPONSE HEADER:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Content-Type: text/plain; charset=utf-8
Content-Length: 6

Connection: close

200 OK
```

Real virtual interaction backend uses the device_id parameter and looks up the IP address from reference table and passes only the content of the query forward to the particular sensor. If an IP address is found from the reference table, the server will respond with 200 OK without actually knowing whether the message reached its destination as the transport mechanism is UDP. Otherwise server will respond with 404 NOT FOUND.

REQUEST RECEIVED BY SENSOR:

```
action=update&device_id=440cd2d8c18d7d3a&sensor_id=display&parameter=vi
ewstate&value=marker5
```

24.1.4 Possible response codes from sensors

Following list includes those HTTP codes supported by the CoAP protocol. These codes SHOULD be returned by sensors or actuators if they are able. The middleware service MUST respond to all requests even if there is no response from a sensor. In such case the middleware SHALL implement time-out after which a appropriate response MUST be generated.

Code	Description	Reference
2.01	Created	[RFCXXXX]
2.02	Deleted	[RFCXXXX]
2.03	Valid	[RFCXXXX]
2.04	Changed	[RFCXXXX]

2.05	Content	[RFCXXXX]
4.00	Bad Request	[RFCXXXX]
4.01	Unauthorized	[RFCXXXX]
4.02	Bad Option	[RFCXXXX]
4.03	Forbidden	[RFCXXXX]
4.04	Not Found	[RFCXXXX]
4.05	Method Not Allowed	[RFCXXXX]
4.06	Not Acceptable	[RFCXXXX]
4.12	Precondition Failed	[RFCXXXX]
4.13	Request Entity Too Large	[RFCXXXX]
4.15	Unsupported Content-Format	[RFCXXXX]
5.00	Internal Server Error	[RFCXXXX]
5.01	Not Implemented	[RFCXXXX]
5.02	Bad Gateway	[RFCXXXX]
5.03	Service Unavailable	[RFCXXXX]
5.04	Gateway Timeout	[RFCXXXX]
5.05	Proxying Not Supported	[RFCXXXX]

25 FIWARE OpenSpecification MiWi VirtualCharacters

Name	FIWARE.OpenSpecification.MiWi.VirtualCharacters
Chapter	Advanced Middleware and Web-based UI,
Catalogue-Link to Implementation	Virtual Characters
Owner	LudoCraft Ltd., Lasse Öörni

25.1 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.org and similar pages in order to understand the complete context of the FI-WARE project.

25.2 Copyright

- Copyright © 2013-2014 by LudoCraft Ltd.

25.3 Legal Notice

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

25.4 Overview

25.4.1 Introduction

Video games and film have for long used systems which provide rendering, animation and interactions for virtual characters. However, virtual character technology is often proprietary and assets created for one virtual character system may not be easily reusable in another system.

This GE consists of an open standard and reference implementation for virtual characters on the Web, allowing for immersive, dynamic character interactions including multi-user experiences. Web applications will be able to create, display and animate virtual characters utilizing industry standard data formats. The characters can be composed of multiple mesh parts, to eg. allow easily swappable parts like upper or lower bodies, and attached objects such as clothing.

The virtual character functionality is implemented as a JavaScript library. The 3D-UI GE (which in turn uses the WebGL API) is utilized for the Entity-Component-Attribute based scene model, the hierarchical transformation graph and implementing the actual rendering: a virtual character becomes part of the scene hierarchy and can be manipulated using the scene model's functions.

If a virtual character is created into a network-synchronized scene utilizing the Synchronization Server GE, its appearance, movements and animations will be transmitted to all observers. This is not however required and the library can also function in purely local mode.

25.5 Basic Concepts

25.5.1 Virtual Character

An animatable 3D object, which may consist of one or more triangle mesh parts. For skeletal animation, may contain a hierarchy of bones (joints) which are 3D transforms (position, rotation, scale) to which the object's vertices refer for vertex deformation (skinning.) Any bone hierarchy can be used so the library is not limited to humanoid characters, though helper functions can be provided for common tasks related to humanoid characters, such as turning the head to look at a particular point in the 3D space. In addition to skeletal animation, vertex morph animations are supported.

25.5.2 Virtual Character Description

A lightweight description file utilizing JSON format, that lists the assets (mesh parts, materials and animations) used for instantiating a character (also referred to as an avatar) into the scene. The file also specifies how the parts are connected. Additionally, the character description can contain metadata, for example defining the name of the character's head bone. This metadata is utilized by the helper functions.

25.5.3 Skeletal Animation

A dataset which describes the changes to individual bone transforms over time to produce for example a walking animation. Typically this involves changing the position and rotation of the bones, and less typically scaling. All the animation data pertaining to a specific value (for example the position or rotation of a specific bone) is commonly referred as an animation track. An animation track consists of time,value -pairs called keyframes. When playing back the animation the successive keyframes are interpolated to give the appearance of smooth animation.

Applying skeletal animation to the mesh vertices (the process of skinning or vertex deformation) requires each vertex of the mesh to specify which bones influence its ultimate position, and to how large degree (blending weight)

Multiple animations can be played back on a character simultaneously (animation blending). This requires the possibility to specify the "priority" of animations in relation to others. If two animations both try to control the same bone simultaneously, the one with higher priority will ultimately take effect. Animations can also be played back only in part of the bone hierarchy, and the magnitude of their influence (ie. blending weight) can be controlled from "no effect" to "full effect". An example of playing animations only partially would be to play a walk animation only on the leg bones of a character, while its hands bones play a different animation, for example waving hands. This way the animations can avoid disturbing each other.

25.5.4 Vertex morph animation

A simpler form of animation, which similarly stores changes over time, but instead of animating bone transforms, it animates directly the positions of individual mesh vertices. This requires storing a larger amount of data, but can be useful for animations where the desired result can not be achieved with skeletal animation only. For example facial animations.

25.6 Generic Architecture

This GE is implemented as a JavaScript library, which can be divided into the following parts:

- Component implementations: AnimationController, Avatar. The Placeable and Mesh components are also needed, but are considered to be part of the 3D-UI GE implementation.
- Character description file parser
- Asset loaders

25.7 Main Interactions

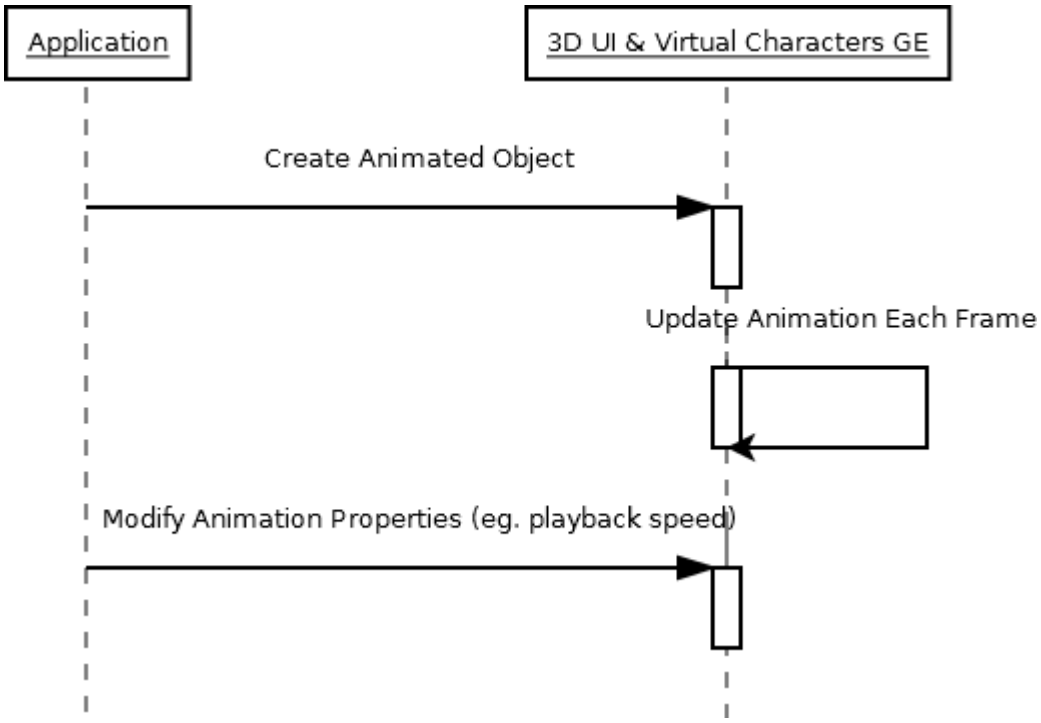
A character can be instantiated either by creating an Avatar component into an entity in the scene, which refers to a character description file, and automatically instantiates the necessary components, or by manually creating the necessary components (Placeable, Mesh, AnimationController) into an entity in the scene.

Once instantiated, animations specified for the character can be enabled and disabled, and the individual properties of animations can be controlled:

- Time position
- Blending weight and blending priority, and to which parts of the bone hierarchy to apply the animation
- Playback speed
- Whether an animation loops or not

The character's individual bones can also be controlled directly by setting their position, rotation or scale. So that animations and direct control do not conflict, doing this will disable keyframe animation on the bone in question. Animation on a bone can be re-enabled when direct control is no longer necessary.

A simple example of the interaction between the application and the GE is illustrated in the following sequence diagram:



25.8 Basic Design Principles

- Genericity. The Virtual Characters GE should be usable for any animating characters, not just humanoids.

25.9 References

- OpenAvatar, a proposed specification for virtual characters, which is not however publicly available as of yet
- Example of a lightweight avatar description file from the MeshMoon platform, which is based on realXtend

25.10 Detailed Specifications

Following is a list of Open Specifications linked to this Generic Enabler. Specifications labeled as "PRELIMINARY" are considered stable but subject to minor changes derived from lessons learned during last interactions of the development of a first reference implementation planned for the current Major Release of FI-WARE. Specifications labeled as "DRAFT" are planned for future Major Releases of FI-WARE but they are provided for the sake of future users.

25.10.1 Open API Specifications

- Virtual Characters Open API Specification

25.11 Re-utilised Technologies/Specifications

- FIWARE.OpenSpecification.MiWi.3D-UI

25.12 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

Annotations

Annotations refer to non-functional descriptions that are added to declaration of native types, to IDL interface definition, or through global annotations at deployment time. The can be used to express security requirements (e.g. "this string is a password and should be handled according the security policy defined for password"), QoS parameters (e.g. max. latency), or others.

AR

AR → Augmented Reality

Augmented Reality (AR)

Augmented Reality (AR) refers to the real-time enhancement of images of the real world with additional information. This can reach from the rough placement of 2D labels in the image to the perfectly registered display of virtual objects in a scene that are photo-realistically rendered in the context of the real scene (e.g. with respect to lighting and camera noise).

IDL

IDL → Interface Definition Language

Interface Definition Language

Interface Definition Language refers to the specification of interfaces or services. They contain the description of types and function interfaces that use these types for input and output parameters as well as return types. Different types of IDL are being used including CORBA IDL, Thrift IDL, Web Service Description Language (WSDL, for Web Services using SOAP), Web Application Description Language (WADL, for RESTful services), and others.

Middleware

Middleware is a software library that (ideally) handles all network related functionality for an application. This includes the setup of connection between peers, transformation of user data into a common network format and back, handling of security and QoS requirements.

PoI

PoI → Point of Interest

Point of Interest (PoI)

Point of Interest refers to the description of a certain point or 2D/3D region in space. It defines its location, attaches meta data to it, and defines a coordinate system relative to which additional coordinate systems, AR marker, or 3D objects can be placed.

Quality of Service (QoS)

Quality of Service refers to property of a communication channel that are non-functional, such a robustness, guaranteed bandwidth, maximum latency, jitter, and many more.

Real-Virtual Interaction

Real-Virtual Interaction refers to Augmented Reality setup that additionally allow users to interact with real-world objects through virtual proxies in the scene that monitor and visualize the state in the real-world and that can use services to change the state of the real world (e.g. switch lights on an off via a virtual button the 3D scene).

Scene

A *Scene* refers to a collection of objects, which are be identified by type (eg. a 3D mesh object, a physics simulation rigid body, or a script object.) These objects contain typed and named data values (composed of basic types such as integers, floating point numbers and strings) which are referred to as attributes. Scene objects can form a hierarchic (parent-child) structure. A HTML DOM document is one way to represent and store a scene.

Security

Security is a property of an IT system that ensures confidentiality, integrity, and availability of data within the system or during communication over networks. In the context of middleware it refers to the ability of the middleware to guarantee such properties for the communication channel according to suitably expressed requirements needed and guarantees offer by an application.

Security Policy

Security Policy refers to rules that need to be fulfilled before a network connection is established or for data to be transferred. It can for example express statements about the identity of communication partners, properties assigned to them, the confidentiality measures to be applied to data elements of a communication channel, and others.

Synchronization

Synchronization is the act of transmitting over a network protocol the changes in a scene to participants so that they share a common, real-time perception of the scene. This is crucial to eg. implementing multi-user virtual worlds.

Type Description

Type Description in the context of the AMi middleware refers to the internal description of native data types or the interfaces described by an IDL. It contains data such as the name of a veriable, its data type, the hierarchical relations between types (e.g. structs and arrays), its memory offset and alignment within another data type, and others. Type Description are used to generate the mapping of native data types to the data that needs to be transmitted by the middleware.

Virtual Character

Virtual Character is a 3D object, typically composed of triangle mesh geometry, that can be moved and animated and can represent an user's presence (avatar) in a virtual world. Typically supported forms of animation include skeletal animation (where a hierarchy of "bones" or "joints" controls the deformation of the triangle mesh object) and vertex morph animation,

where the vertices of the triangle mesh are directly manipulated. Virtual character systems may support composing the character from several mesh parts, for example separate upper body, lower body and head parts, to allow better customization possibilities.

26 Virtual Characters Open API Specification

26.1 Overview

The Virtual Characters library is interacted with through JavaScript objects, and more specifically components which are created to entities in the scene. The base scene model is shared with the 3D-UI and Synchronization GE's.

The displaying and animation of virtual characters is implemented by the co-operation of three components: **Placeable**, **Mesh** and **AnimationController**. The Placeable holds the transform (position, rotation and scale) of the character. The Mesh component holds information of the mesh and material assets used by the character, while the AnimationController controls the playback and blending of animations.

The automatic instantiation of a virtual character into a scene according to a description file is realized through a fourth component: **Avatar**.

26.2 Character instantiation

The only Attribute of the Avatar component is the reference ("appearanceRef") to the character description file (JSON format) it should use. Once the Avatar component is added to an entity, and the description file has been loaded successfully, the Avatar component will automatically create the necessary mesh component(s) into its entity, and into child entities if there are multiple parts.

An example:

```
avatar.appearanceRef = "MyDescriptionFile.json";
```

Alternatively, it is perfectly valid to not use the Avatar component, but to manually create the Mesh, Placeable & AnimationController components as necessary.

26.3 Character description file

The following shows through an example the structure of the character description file. The character in question has a main (top-level) skinned mesh on the top level and one attachment "part": a hat mesh attached to the character's head bone. The main mesh, as well as the parts can be individually transformed (position, rotate, scale). The rotations are described as Euler angles in degrees around the X, Y and Z axes. It is also valid to have no top-level mesh. Additional assets for materials (per submesh) and animations can be optionally specified. In this example the assets refer to Collada .dae files.

```
{
    "name"      : "Jack",
    "geometry"  : "Jack.dae",
    "transform" :
    {
```

```
        "pos": [0, 0, 0],
        "rot": [0, 0, 0],
        "scale": [1, 1, 1]
    },
    "materials" :
    [
        "JackBody.material",
        "JackHead.material"
    ]
    "parts" :
    [
        {
            "name"      : "Hat",
            "geometry"  : "Hat.dae",
             "transform" :
            {
                "pos": [0, 0, 0],
                "rot": [0, 0, 0],
                "scale": [1, 1, 1],
                "parentBone": "Bip01_Head"
            },
        }
    ],
    "animations" :
    [
        {
            "name" : "Walk",
            "src"  : "Jack@walk.dae"
        },
        {
            "name" : "Run",
            "src"  : "Jack@run.dae"
        }
    ]
```

```
}
```

26.4 Animation control

The AnimationController component contains the following functions for controlling animation playback. Animations are referred to with their string names. The playing animations will be automatically updated in conjunction with rendering each frame.

```
animationController.play(name, fadeInTime, crossFade, looped)
```

Start playback of an animation. fadeInTime specifies a time in seconds during which to smoothly blend the animation from zero blending weight to full weight. crossFade is a boolean which will cause other animations to be faded out as the playback of the new animation is started. looped is a boolean to indicate whether the animation should loop, or only play once.

```
animationController.playLooped(name, fadeInTime, crossFade)
```

Same as above, but the animation will always be looped.

```
animationController.stop(name, fadeOutTime)
```

Stop playback of an animation. fadeOutTime is the fade-out period in seconds, during which the blending weight is smoothly reduced to zero.

```
animationController.stopAll()
```

Stop playback of all animation of the character.

```
animationController.setAnimWeight(name, weight)
```

Set the blending weight of an animation between 0 (none) and 1 (full).

```
animationController.setAnimSpeed(name, speed)
```

Set the playback speed of an animation, where 1 is the original speed forward, 2 would be twice as fast, and -1 would be reverse with original speed.

27 FIWARE OpenSpecification MiWi InterfaceDesigner

Name	FIWARE.OpenSpecification.MiWi.InterfaceDesigner
Chapter	Advanced Middleware and Web-based UI,
Catalogue-Link to Implementation	Interface Designer
Owner	Adminotech, Cvetan Stefanovski

27.1 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.org and similar pages in order to understand the complete context of the FI-WARE project.

27.2 Copyright

- Copyright © 2013 by Adminotech Oy

27.3 Legal Notice

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

27.4 Overview

The Scene / Entity Component (abbreviated as EC) editor is an in-browser world editor that allows users to easily create, remove, and manipulate scene objects (further in the text "entities") through variety of tools. This editor in particular utilizes Scene and EC model, in other words, manipulates entities, components and attributes. Manipulations can be done through GUI that consist of three parts: scene tree, EC editor, and additional toolbar, or directly into the scene via 3D manipulation helper objects such as transform gizmo / axis tripods, grids etc. The GUI provides extensive editing of entities that cannot be otherwise done via a 3D manipulation helper, and also in most of the cases serves for fine-tuning of values.

27.5 Basic Concepts

27.5.1 3D vs. 2D

The editor will focus on providing capabilities for 3D scene and object manipulations. There will however be efforts made with the 2D-UI to integrate editing capabilities for WebComponent based 2D components.

27.5.2 Local Scene vs. Replicated Scene

The editor does not make any assumptions if the scene its manipulating is locally declared or it came from a server. It simply manipulates the current scene state, the changes will replicated back to the server if there is and active server connection, and the manipulated objects are replicated.

Moreover multi-user editing does not need any special logic from the editor. You will simply see other peoples changes if they do any on a replicated server. The editor does not implement any logic for "reserving" objects for editing, so that one user could edit an object at a time. If multiple users edit the same object the server will take care of applying the latest sent changes, this may result in weird situations but is out of the scope of the editor GE to take care of multi-user simultaneous edits on a particular object.

27.5.3 Scene tree

The scene tree shows all available entities on a scene that can be manipulated, along with the most basic info about their components, such as component name and component type.

27.5.4 EC editor

EC editor expands the content of an entity that is of interest so that attributes of components and the components themselves can be added/removed/edited.

27.5.5 Toolbar

A toolbar that consist of actions that are most used, repetitive, or just considered to be needed for scene manipulating. For example: undoing / redoing actions, creating primitive shapes (box, sphere, cone, torus), options such as "Snap to grid", creating movable entity, drawable entity, script entity etc.

27.5.6 3D manipulation helpers

Consist of transform gizmo and grids for a direct in-world manipulation of entities.

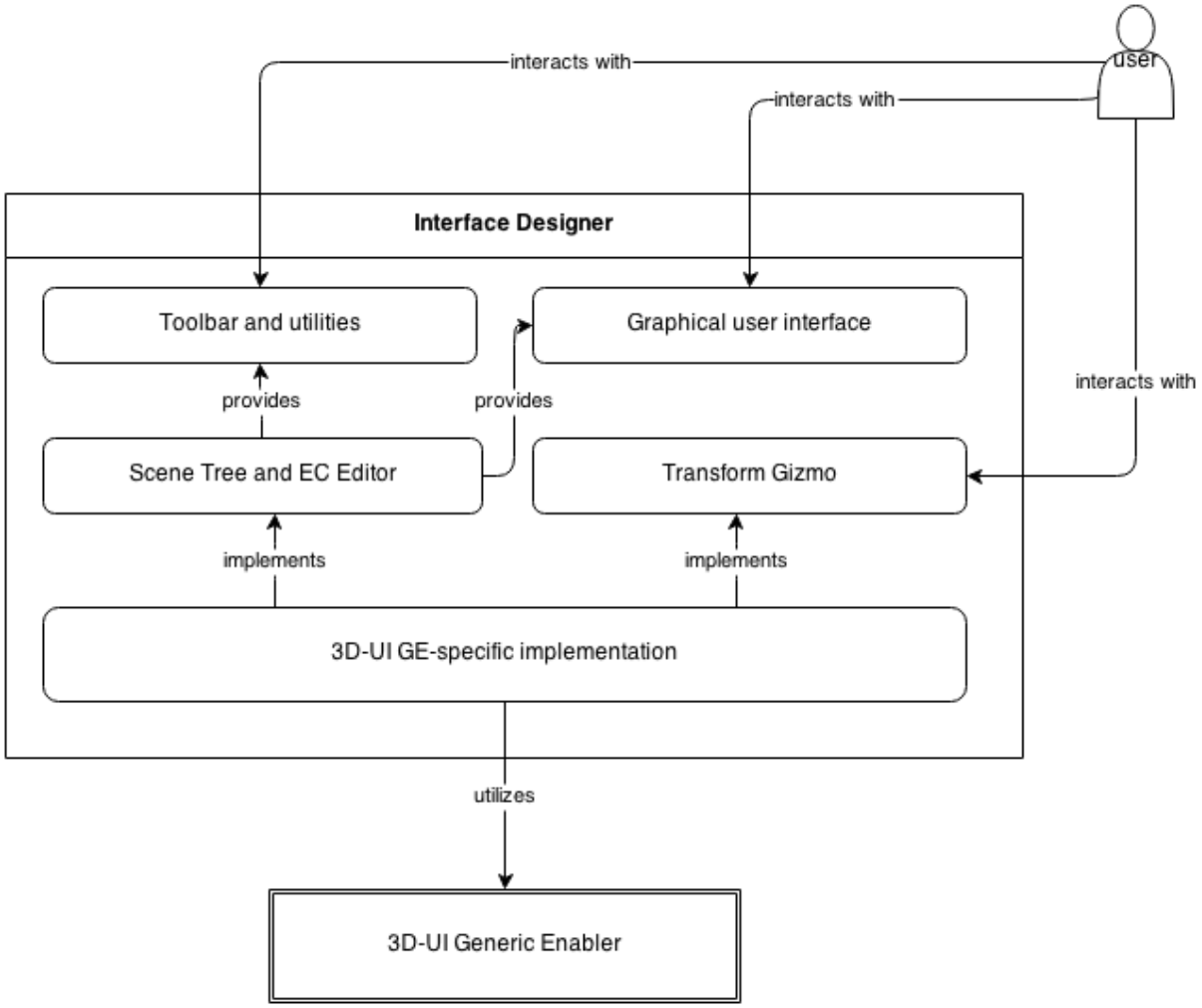
27.5.7 XML3D support

This GE also provides support for XML3D scenes. The key and only difference is that XML3D elements have the properties of both entities (they can have children elements) and components (they have their own attributes).

27.6 Generic Architecture

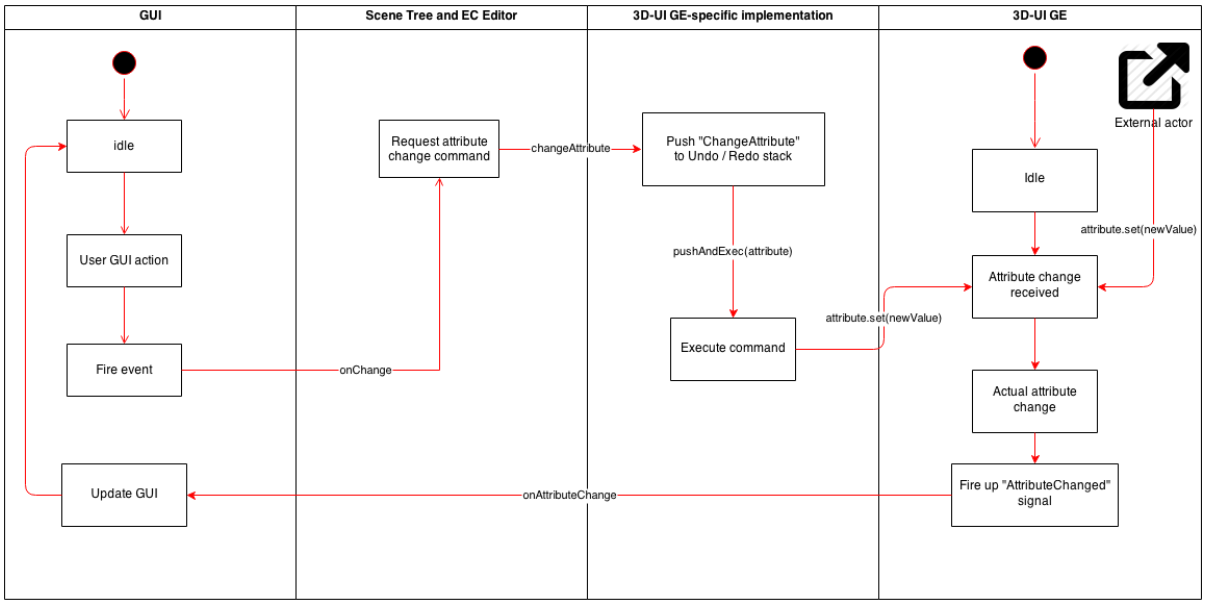
This application-oriented GE will be implemented into several modules, split into javascript libraries:

- Scene editor
- EC editor
- Toolbar manager
- direct 3D manipulator (transform gizmo)



The interface designer is implemented so that there is a main implementation (Scene Tree and EC Editor) that provides a graphical user interface and the toolbar, while 3D-UI specific implementations (f.ex. XML3D or WebTundra 3D-UI GEs) provide the internal communication between the editor and the GE. The user can interact with the scene tree via GUI; Transform gizmo serves only for modifying 3D world coordinates. Main implementation has own "wrapper" objects that wrap around the 3D-UI-specific implementation, so that different 3D-UI GEs can use the same code. The 3D-UI specific implementation also listens to all the changes that may come up in the 3D-UI by external means (for example, a scene that is hosted on a server and has two connected clients that try to manipulate objects in the scene). This is made for consistence of the current state of the scene and what the GUI shows.

Transform gizmo is independent from the main editor. The editor only holds an instance of the so-called "TransformEditor", since Transform gizmo is, in a way, an editor only for a transform attribute (position / rotation / scale) and takes care of attaching the transform gizmo to objects of interest.



The picture above shows an example of what happens internally when an attribute has been changed via the GUI. When the user changes the value of the input box that has an attribute assigned to, it will first fire up the "change" event coming from Javascript / jQuery. The event listener within the 3D-UI specific implementation will receive the request for adding an "AttributeChange" command into the undo / redo stack, and it will push a new instance of ChangeAttributeCommand, with passing attribute information as arguments. Once the command is pushed into the stack, it will call its exec() method that will actually set the new value for the observed attribute via the 3D-UI GE API. The 3D-UI GE will then fire up a signal called "onAttributeChange", and the event listener in the main implementation will then update the GUI to the current state. Also, if an external actor causes an attribute change, the main implementation will once again update the GUI. It is important to note that when an attribute is modified by external means, its change is not pushed into the undo / redo stack, as it might create unwanted results, such as the current user undoing a change that the external actor made. This will cause some undefined behavior, but it is out of the editor's scope to handle concurrent situations.

27.7 Main Interactions

- Apart from having a view of all available entities and its components, the **scene tree** is used to:
    - create / remove entities by hand, i.e. creating an empty entity and adding components to it;
    - double-clicking existing entity or component will expand to the EC editor, which components and attributes can be further manipulated there, and also that will trigger a transform gizmo and the entity's bounding box to appear in the origin point of the entity of interest;
    - grouping multiple entities to a single group, for a cleaner view, and manipulating a group's transform (which is merely a manipulation of a central pivot point of the grouped entities).

- **EC editor** shows an expanded view of the components and its attributes of an entity (See Scene and EC model), which allows
    - Creating / removing components and properties of components (name, replication flag, temporary flag);
    - Editing a component's attributes via input boxes (string, number, array, transform attributes) or checkboxes (boolean attributes);
    - Creating or removing attributes in case of editing a DynamicComponent (a user-defined component that has custom user-definable attributes for application-specific purposes).
- **Toolbar** that consists of actions such as:
    - Primitive entities to be created
    - Quickly create "most used" entities, such as drawable (entity with EC_Mesh), movable (entity with EC_Placeable), script (entity with EC_Script), skybox, waterplane...
    - Creating helper grids;
    - Selection tool (select single or multiple entities with a rectangle, similar to selecting multiple files on the desktop with the mouse);
    - Camera helper actions
- **3D manipulation helpers** include:
    - Transform gizmo axis tripod that locks translating / rotating / scaling an entity on a single axis;
    - Small planes in the zero point of the axis tripod that locks moving an entity on a single x0y, x0z, y0z plane;

27.8 Basic Design Principles

- Apart from the direct 3D manipulator (transform gizmo), all modules serve as an application to the main scene.
- The editor UI is built dynamically, and its code do not have to be modified if new components are added to the target GE core implementation.
- The editor UI always listens to all scene changes and it is updated appropriately, whenever an entity is edited from the editor or another third-party.
- The transform gizmo is an EC component that can be added from any other application GEs that may need the use of 3D manipulating, and works independently of the editors.

27.9 Detailed Specifications

Following is a list of Open Specifications linked to this Generic Enabler. Specifications labeled as "PRELIMINARY" are considered stable but subject to minor changes derived from lessons learned during last interactions of the development of a first reference implementation planned for the current Major Release of FI-WARE. Specifications labeled as "DRAFT" are planned for future Major Releases of FI-WARE but they are provided for the sake of future users.

27.9.1 Open API Specifications

- Interface Designer API Specification

27.10 Re-utilised Technologies/Specifications

- FIWARE.OpenSpecification.MiWi.3D-UI

27.11 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

Annotations

Annotations refer to non-functional descriptions that are added to declaration of native types, to IDL interface definition, or through global annotations at deployment time. The can be used to express security requirements (e.g. "this string is a password and should be handled according the security policy defined for password"), QoS parameters (e.g. max. latency), or others.

AR

AR → Augmented Reality

Augmented Reality (AR)

Augmented Reality (AR) refers to the real-time enhancement of images of the real world with additional information. This can reach from the rough placement of 2D labels in the image to the perfectly registered display of virtual objects in a scene that are photo-realistically rendered in the context of the real scene (e.g. with respect to lighting and camera noise).

IDL

IDL → Interface Definition Language

Interface Definition Language

Interface Definition Language refers to the specification of interfaces or services. They contain the description of types and function interfaces that use these types for input and output parameters as well as return types. Different types of IDL are being used including CORBA IDL, Thrift IDL, Web Service Description Language (WSDL, for Web Services using SOAP), Web Application Description

Language (WADL, for RESTful services), and others.

Middleware

Middleware is a software library that (ideally) handles all network related functionality for an application. This includes the setup of connection between peers, transformation of user data into a common network format and back, handling of security and QoS requirements.

PoI

PoI → Point of Interest

Point of Interest (PoI)

Point of Interest refers to the description of a certain point or 2D/3D region in space. It defines its location, attaches meta data to it, and defines a coordinate system relative to which additional coordinate systems, AR marker, or 3D objects can be placed.

Quality of Service (QoS)

Quality of Service refers to property of a communication channel that are non-functional, such a robustness, guaranteed bandwidth, maximum latency, jitter, and many more.

Real-Virtual Interaction

Real-Virtual Interaction refers to Augmented Reality setup that additionally allow users to interact with real-world objects through virtual proxies in the scene that monitor and visualize the state in the real-world and that can use services to change the state of the real world (e.g. switch lights on an off via a virtual button the 3D scene).

Scene

A *Scene* refers to a collection of objects, which are be identified by type (eg. a 3D mesh object, a physics simulation rigid body, or a script object.) These objects contain typed and named data values (composed of basic types such as integers, floating point numbers and strings) which are referred to as attributes. Scene objects can form a hierarchic (parent-child) structure. A HTML DOM document is one way to represent and store a scene.

Security

Security is a property of an IT system that ensures confidentiality, integrity, and availability of data within the system or during communication over networks. In the context of middleware it refers to the ability of the middleware to guarantee such properties for the communication channel according to suitably expressed requirements needed and guarantees offer by an application.

Security Policy

Security Policy refers to rules that need to be fulfilled before a network connection is established or for data to be transferred. It can for example express statements about the identity of communication partners, properties assigned to them, the confidentiality measures to be applied to data elements of a communication channel, and others.

Synchronization

Synchronization is the act of transmitting over a network protocol the changes in a scene to

participants so that they share a common, real-time perception of the scene. This is crucial to eg. implementing multi-user virtual worlds.

Type Description

Type Description in the context of the AMi middleware refers to the internal description of native data types or the interfaces described by an IDL. It contains data such as the name of a veriable, its data type, the hierarchical relations between types (e.g. structs and arrays), its memory offset and alignment within another data type, and others. Type Description are used to generate the mapping of native data types to the data that needs to be transmitted by the middleware.

Virtual Character

Virtual Character is a 3D object, typically composed of triangle mesh geometry, that can be moved and animated and can represent an user's presence (avatar) in a virtual world. Typically supported forms of animation include skeletal animation (where a hierarchy of "bones" or "joints" controls the deformation of the triangle mesh object) and vertex morph animation, where the vertices of the triangle mesh are directly manipulated. Virtual character systems may support composing the character from several mesh parts, for example separate upper body, lower body and head parts, to allow better customization possibilities.

28 Interface Designer API Specification

This is an Interface Designer javascript API. More functionality and details are to be added, and existing ones may be modified.

- Editor
  - setEnabled(toggle:bool) : void
    - Enables or disables the editor, depending on "toggle" value
    - Arguments: toggle:bool
    - This method does not return a value.
  - isEditorEnabled() : bool
    - Checks if the editor is enabled.
    - Arguments: none
    - Returns "true" if the editor is enabled and "false" if it is disabled.
  - switchPanels(ecPanel : bool) : void
    - Switches between Scene Tree panel and EC panel. If "ecPanel" is true, the EC panel will be shown, otherwise the Scene Tree editor is shown
    - Arguments: ecPanel:bool
    - This method does not return a value.
  - setSwitchPanelsShortcut(meta : string, key : string) : void
    - Sets a new shortcut instead of the default "Shift + E" for switching between EC and Scene Tree panels. The variable "meta" can accept values such as "ctrl", "shift", "alt" or "meta", while "key" accepts any key that is on the keyboard
    - Arguments: meta : string, key : string
    - This method does not return a value.
  - setToggleEditorShortcut(meta : string, key : string) : void
    - Sets a new shortcut instead of the default "Shift + S" for enabling / disabling the editor. The variable "meta" can accept values such as "ctrl", "shift", "alt" or "meta", while "key" accepts any key that is on the keyboard
    - Arguments: meta : string, key : string
    - This method does not return a value.

- selectEntity (entityPtr : EntityWrapper) : void
  - Assigns a target entity "entityPtr" to be edited. The editor UI will switch the EC panel and then build the UI in correspondence with the contents of the target entity
  - Arguments: entityPtr : EntityWrapper, a reference to an EntityWrapper instance.
  - This method does not return a value.

29 FI-WARE Open Specification Legal Notice (essential patents license)

29.1.1 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.

"Copyright Holders" of this FI-WARE Open Specification is/are the Party/Parties identified as the copyright owner/s on the wiki page of the FI-WARE Open Specification that contains the link to this Legal Notice.

29.1.2 Use Of Specification - Terms, Conditions & Notices

The material in this specification details, as of the date when Copyright Holders contributed it, a FI-WARE Generic Enabler Specification (hereinafter “Specification”) that is provided, in accordance with the terms, conditions and notices set forth below (“License”). 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.

29.1.3 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 royalty free 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 is an implementation of this Specification.

29.1.4 Patent License

“Specification Essential Patents” shall mean patents and patent applications, which are necessarily infringed by an implementation compliant with the Specification and which are owned by any of the Copyright Holders of this Specification. “Necessarily infringed” shall mean that no commercially and/or technical reasonable alternative exists to avoid infringement.

Each of the Copyright Holders, hereby agrees to grant you, on 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, import and/or distribute software implementations compliant with the Specification.

If you institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that this Specification constitutes direct or contributory patent infringement, then any patent licenses granted to you under this License for that Specification shall terminate as of the date such litigation is filed.

The FI-WARE Project Partners and/or Copyright Holders shall not be responsible for identifying patents for which a license may be required for any implementation of 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.

29.1.5 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 Holders.

For avoidance of doubt, the rights granted are only those expressly stated in this Legal Notice herein. No other rights of any kind are granted by implication, estoppel, waiver or otherwise

29.1.6 Disclaimer Of Warranty

WHILE THIS SPECIFICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE FI-WARE PROJECT PARTNERS AND THE COPYRIGHT HOLDERS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS SPECIFICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, WARRANTY OF NON INFRINGEMENT OF THIRD PARTY RIGHTS, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE.

YOU ARE SOLELY RESPONSIBLE FOR DETERMINING THE APPROPRIATENESS OF USING OR REDISTRIBUTING THIS SPECIFICATION AND ASSUME ANY RISKS ASSOCIATED WITH YOUR EXERCISE OF PERMISSIONS UNDER THIS LICENSE

IN NO EVENT AND UNDER NO LEGAL THEORY SHALL THE FI-WARE PROJECT PARTNERS AND THE COPYRIGHT HOLDERS BE LIABLE FOR ERRORS CONTAINED IN THIS SPECIFICATION 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 SPECIFICATION, 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 PATENT AND COPYRIGHT LICENSES GRANTED TO YOU TO USE THIS SPECIFICATION.

29.1.7 Trademarks

You shall not use any trademark, marks or trade names (collectively, "Marks") of the FI-WARE Project Partners or the Copyright Holders or the FI-WARE project without their prior written consent, except as required for reasonable and customary use in describing the origin of this Specification and reproducing the content of this Legal Notice.

29.1.8 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.org.

However there is no obligation on the part of the Copyright Holder to provide any review, improvement, bug fixes or modifications this Specification to address any reported ambiguities, inconsistencies, or inaccuracies.

30 FI-WARE Open Specification Legal Notice (implicit patents license)

30.1.1 General Information

"FI-WARE Project Partners" refer to Parties of the FI-WARE Project in accordance with the terms of the FI-WARE Consortium Agreement.

Copyright Holders of the FI-WARE Open Specification is/are the Party/Parties identified as the copyright owner/s on the wiki page of the FI-WARE Open Specification that contains the link to this Legal Notice.

30.1.2 Use Of Specification - Terms, Conditions & Notices

The material in this specification details a FI-WARE Generic Enabler Specification (hereinafter "Specification") and is provided in accordance with the terms, conditions and notices set forth below ("License"). 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.

30.1.3 Licenses

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, royalty free 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 is an implementation of this Specification.

30.1.4 Patent Information

The FI-WARE Project Partners and/or Copyright Holders shall not be responsible for identifying patents for which a license may be required for any implementation of 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.

30.1.5 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 Holders.

For avoidance of doubt, the rights granted are only those expressly stated in this Legal Notice herein. No other rights of any kind are granted by implication, estoppel, waiver or otherwise

30.1.6 Disclaimer Of Warranty

WHILE THIS SPECIFICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE FI-WARE PROJECT PARTNERS AND THE COPYRIGHT HOLDERS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS SPECIFICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, WARRANTY OF NON INFRINGEMENT OF THIRD PARTY RIGHTS, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE.

YOU ARE SOLELY RESPONSIBLE FOR DETERMINING THE APPROPRIATENESS OF USING OR REDISTRIBUTING THIS SPECIFICATION AND ASSUME ANY RISKS ASSOCIATED WITH YOUR EXERCISE OF PERMISSIONS UNDER THIS LICENSE

IN NO EVENT AND UNDER NO LEGAL THEORY SHALL THE FI-WARE PROJECT PARTNERS AND THE COPYRIGHT HOLDERS BE LIABLE FOR ERRORS CONTAINED IN THIS SPECIFICATION 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 SPECIFICATION, 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 PATENT AND COPYRIGHT LICENSES GRANTED TO YOU TO USE THIS SPECIFICATION.

30.1.7 Trademarks

You shall not use any trademark, marks or trade names (collectively, "Marks") of the FI-WARE Project Partners or the Copyright Holders or the FI-WARE project without prior written consent, except as required for reasonable and customary use in describing the origin of this Specification and reproducing the content of this Legal Notice.

30.1.8 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.org.

However there is no obligation on the part of the Copyright Holder to provide any review, improvement, bug fixes or modifications this Specification to address any reported ambiguities, inconsistencies, or inaccuracies.