



Specifications of the Lab Owner and Cloud Services (initial) – M21 revision

Sten Govaerts

► To cite this version:

Sten Govaerts. Specifications of the Lab Owner and Cloud Services (initial) – M21 revision. [Research Report] Go-Lab Project. 2014. hal-01201938

HAL Id: hal-01201938

<https://telearn.hal.science/hal-01201938>

Submitted on 18 Sep 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Go-Lab

Global Online Science Labs for Inquiry Learning at School

*Collaborative Project in European Union's Seventh Framework Programme
Grant Agreement no. 317601*



Deliverable D4.1

Specifications of the Lab Owner and Cloud Services (initial) – M21 revision

Editor	Sten Govaerts (EPFL)
Date	30 th July, 2014
Dissemination Level	Public
Status	Final



©2014, Go-Lab consortium

The Go-Lab Consortium

Beneficiary Number	Beneficiary Name	Beneficiary short name	Country
1	University Twente	UT	The Netherlands
2	Ellinogermaniki Agogi Scholi Panagea Savva AE	EA	Greece
3	École Polytechnique Fédérale de Lausanne	EPFL	Switzerland
4	EUN Partnership AISBL	EUN	Belgium
5	IMC AG	IMC	Germany
6	Reseau Menon E.E.I.G.	MENON	Belgium
7	Universidad Nacional de Educación a Distancia	UNED	Spain
8	University of Leicester	ULEIC	United Kingdom
9	University of Cyprus	UCY	Cyprus
10	Universität Duisburg-Essen	UDE	Germany
11	Centre for Research and Technology Hellas	CERTH	Greece
12	Universidad de la Iglesia de Deusto	UDEUSTO	Spain
13	Fachhochschule Kärnten - Gemeinnützige Privatstiftung	CUAS	Austria
14	Tartu Ulikool	UTE	Estonia
15	European Organization for Nuclear Research	CERN	Switzerland
16	European Space Agency	ESA	France
17	University of Glamorgan	UoG	United Kingdom
18	Institute of Accelerating Systems and Applications	IASA	Greece
19	Núcleo Interactivo de Astronomia	NUCLIO	Portugal

Contributors

Name	Institution
Sten Govaerts, Christophe Salzmann, Denis Gillet, Wissam Halimi	EPFL
Pablo Orduña	UDEUSTO
Danilo Garbi Zutin	CUAS
Anjo Anjewierden (internal reviewer)	UT
Simon Schwantzer (internal reviewer)	IMC

Legal Notices

The information in this document is subject to change without notice. The Members of the Go-Lab Consortium make no warranty of any kind with regard to this document, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The Members of the Go-Lab Consortium shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material. The information and views set out in this deliverable are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

Executive Summary

This deliverable elaborates on the Go-Lab vision for the integration of existing and new remote laboratories, with the Go-Lab infrastructure. For instance, remote labs could integrate with the Go-Lab booking system (see D4.2), the learning analytics services (see D4.2) or specific apps (e.g., a data viewer app could read from sensors and actuators of any remote lab).

For new remote laboratories, this deliverable provides a client-server architecture, based on the Smart Device paradigm, with specifications that detail the required services, protocols and data formats. Next to that, additional services are suggested that can ease integration with the Go-Lab infrastructure or support required services. Furthermore, we provide guidelines for functionalities internal to the Smart Device. The Smart Device paradigm conceptualizes and embeds the lab-owner services.

On the other hand, for already existing labs, Go-Lab proposes the Smart Gateway paradigm, which aims to make legacy labs compatible with the Smart Device paradigm. Through this compatibility, such legacy labs can be integrated in the Go-Lab infrastructure. Due to the wide technical variety of legacy remote labs, we provide different levels of compatibility through the Smart Gateway. The Smart Gateway paradigm conceptualizes and embeds the cloud services.

This deliverable is a revised version of the D4.1 deliverable submitted in M12 and contains improved, extended and more detailed specifications. The 'partner' deliverable D4.3 provides implementation examples and documentation of the software packages to develop remote labs using the Smart Device and Smart Gateway paradigms. An update of the specifications presented in this deliverable will be provided in D4.5 (M30).

Table of Contents

1	Introduction	8
1.1	Remote labs	8
1.2	Value Proposition	9
1.3	Integration Levels	11
2	Lab Owner or Plug Services	13
2.1	The Smart Device Architecture	14
2.1.1	The Smart Device in the Go-Lab Infrastructure	14
2.2	Smart Device Protocols and Technical details	16
2.3	Smart Device Services and Functionalities	17
2.3.1	Introduction	17
2.3.2	List of Services and Functionalities	18
2.3.3	Metadata Service	21
2.3.4	Sensor Service – getSensorData	37
2.3.5	Actuator Service – sendActuatorData	42
2.3.6	User Activity Logging Service – getLoggingInfo	44
2.3.7	Client Application Service – getClients	45
2.3.8	Models service – getModels	46
2.4	Smart Device Interactions	47
2.4.1	Authentication and Booking	47
2.4.2	Interaction Modes	48
2.4.3	WebSocket Channeling	50
2.4.4	Lab Instruments as Complex Sensors	51
2.5	The Revised Smart Device Specification Compared to the M12 Version	52
3	Cloud Services	53
3.1	Introduction	53
3.2	Requirements for the Smart Gateway	54
3.2.1	Functional Requirements for the Smart Gateway	54
3.2.2	Non-functional Requirements for the Smart Gateway	55
3.3	Review of Legacy Lab Platforms	55
3.4	Requirements for the Smart Gateway and Comparison with Other Systems	57
3.5	Specification and Architecture of the Smart Gateway	58
3.5.1	Architecture	60
3.5.2	Specifications of the Plug-in System	65
3.5.3	The Protocol Translator	71
3.6	Benefits for Lab Owners	73
4	Standardisation	75
5	Conclusion	76
6	Appendix A: Smart Device metadata specification details	77

6.1	Extensions for WebSockets	77
6.2	Extensions for Concurrency mechanisms	78
6.3	Additional Minimal Extensions	79
6.3.1	Data Types	79
7	Appendix B: The Metadata Specification for an Example Smart Device	80
7.1	RED Smart Device	80
7.1.1	Metadata Specification	80
7.1.2	Example Requests and Responses to the Smart Device Services	97
7.2	Running example Smart Device	101
7.2.1	Metadata Specification	101
7.2.2	Example Requests and Responses to the Smart Device Services	121
8	Appendix C: Value-proposition Canvas for Lab owner and Cloud Services	129
8.1	The value-proposition canvas methodology	129
8.2	Detailed Results	131
8.2.1	Customer	131
8.2.2	Product	133
8.3	Analysis of the results	134
8.3.1	Customer	134
8.3.2	Product	136
8.3.3	Conclusion	137
9	Appendix D: Lab Owner Workshop	139
9.1	Value Proposition	139
9.2	Smart Device and Interfacing	140
9.3	Smart Gateway and Federation	140
	References	142

1 Introduction

Within the Go-Lab project, online laboratories (referred hereafter as online labs) have been divided into three general categories (see Deliverable D2.1 & DoW):

- *Virtual labs*, which are simulations with animations of scientific experiments available on the Web.
- *Remote labs*, which have real physical equipment with their instrumentation accessible at distance (such remote labs are also called rigs).
- *Data sets*, which contain measurements gathered using real scientific instruments such as telescopes. Data sets can be analysed and visualised using dedicated online tools.

Task 4.1 and Task 4.2 are focusing on remote labs, which is the category for which there are significant challenges in terms of plugging and sharing. To plug remote labs online, Go-Lab is proposing in this deliverable the Smart Device paradigm, which conceptualises and embeds the lab owner services (see Section 2). To share remote labs online, Go-Lab is proposing the Smart Gateway paradigm, which conceptualises and embeds the cloud services (see Section 3).

1.1 Remote labs

Remote labs typically rely on a client-server architecture (see Figure 1). The services of the lab server are enabling communication through the Internet with the real equipment and its instrumentation. The physical lab is typically connected through a hardware interface with the lab server thanks to analog and digital inputs (connected to sensors) and outputs (connected to actuators). The lab server can be a microcontroller, a computer or a remote laboratory management system (RLMS). The lab server has to ensure on one hand that no ungranted access is possible and on the other hand that the lab is in a proper operational state. The client is enabling remote user interaction as a standalone application or as a component in a Web environment. The client typically enables the observation of the lab (e.g. through a live video stream), configuration at distance, setting of parameters and visualization of data or measurement (in batch or in real-time while the lab is operated).

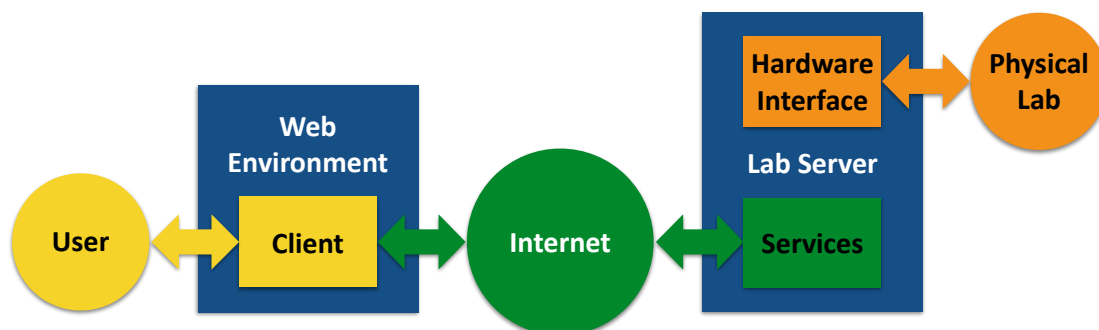


Figure 1: Typical client-server remote lab architecture

Up to now, tightly coupled client-server solutions have been designed and implemented to enable interaction with and management of remote labs. This strong coupling and the lack of standardisation is impairing the ability for lab owners to easily plug and share their remote labs with different platforms, and for users to interact with these labs and with their peers in their preferred environment. Lab owners cannot easily adapt solutions developed by others to their own infrastructures. Teachers or students interested to exploit remote labs from various providers have to install a different solution for each of them and cannot personalize it to their needs.

While the interfacing of a remote lab will always rely on ad hoc solutions because of the large variety of the physical equipment and the associated instrumentation (orange part in Figure 1), the way remote labs are made accessible through the Internet can be standardised (green part in Figure 1). This deliverable describes how new remote labs or legacy labs can be made accessible in a standardised way, directly as Smart Devices or through the Smart Gateway. Such standardisation enables a decoupling of the client and the server and a decoupling of the interaction and management services. In other words, users can develop the clients they need to be integrated in the environments they want. In Go-Lab, the clients are developed as OpenSocial apps (see D5.1) that can be integrated in social media platforms or learning management systems supporting this Web application standard.

1.2 Value Proposition

Following the recommendations of the reviewers during the first review, Go-Lab has carried out activities to clarify its value proposition for lab owners in order to foster adoption of the Go-Lab technology and/or to integrate their labs in the Go-Lab infrastructure.

Two main events have been organised. First, an internal workshop has been organised after the General Assembly in Madrid (March 17-19, 2014) with the members of the technical cluster to define the value proposition from their own perspective and discuss the Smart Device specification. Then, a discussion was held with selected lab owners during the Smart Device and Smart Gateway workshop (also held in Madrid, June 4-6, 2014). See Appendix D for more details and the objectives of this workshop. These value propositions were created using the value-proposition canvas methodology (see Section 8.1 for details). In the two events, the core Go-Lab features have been introduced and then discussed. They can be summarised as follows:

1. Go-Lab is promoting science education through *inquiry learning* with *on-line labs* (coupling in an optimal way a proven learning methodology and engaging learning resources).
2. Go-Lab is supporting inquiry learning activities with *inquiry learning spaces* (ILS), which can be created freely by teachers and used in their regular activities (teachers are still in control of their learning scenarios and they can integrate Go-Lab resources in their usual classroom activities).

3. Go-Lab is enabling, through the *ILS Platform*, the creation of inquiry learning spaces that combine a rich set of content and services (including support applications), as well as *structure* (through tabs enforcing typical inquiry learning phases).
4. Go-Lab is offering, through its *Lab Repository*, a social scheme for sharing *online labs* freely, *support apps* and *inquiry learning spaces*.
5. Go-Lab is providing scaffolding applications and services which rely on *learning analytics* and support *interoperability* while *enforcing privacy*.

From a Go-Lab technical partner perspective, the major impacts of Go-Lab are the following for teachers:

- a standardised interface for booking labs;
- interoperability between labs and apps, thus enabling richer user experience and better integration of the lab with the pedagogical approaches;
- support for learning analytics.

and for lab owners:

- a robust and secure implementation of the specification to build their lab, resulting in faster development and more robust and secure deployment;
- specification and software packages enabling an easy plug of existing labs online to enable reuse of existing labs in Go-Lab;
- reuse of client apps thanks to standardisation;
- easy addition of extra functionality provided by the Go-Lab infrastructure which can make the lab more user-friendly and can speed up development, e.g. a booking mechanism or learning analytics services;
- interoperability with existing apps (e.g. a data viewer app can visualise data of any Smart Device), which again can speed up development and increase the attractiveness of the lab for users;
- integration of the labs in various platforms and environments thanks to open standards (e.g. OpenSocial, WebSockets & HTML5).

A more detailed analysis of the value-proposition canvas completed by Go-Lab partners can be found in Appendix C.

From the *lab owner perspective*, the main Go-Lab value proposition is *the possibility to combine technical resources and pedagogical structure in an ILS* (bullet 3 above), and *to make visible their own labs in the Go-Lab repository* (bullet 4 above). The motivation for the latter is to attract funding to support the development of their resources and the exploitation of their facilities. As a consequence, the Smart Gateway is seen as an effective means to make their labs available in the Go-Lab repository and compatible with the ILS Platform, with a minimal adaptation on their side. Figure 2 illustrates the result of the value-proposition canvas exercise with lab owners.

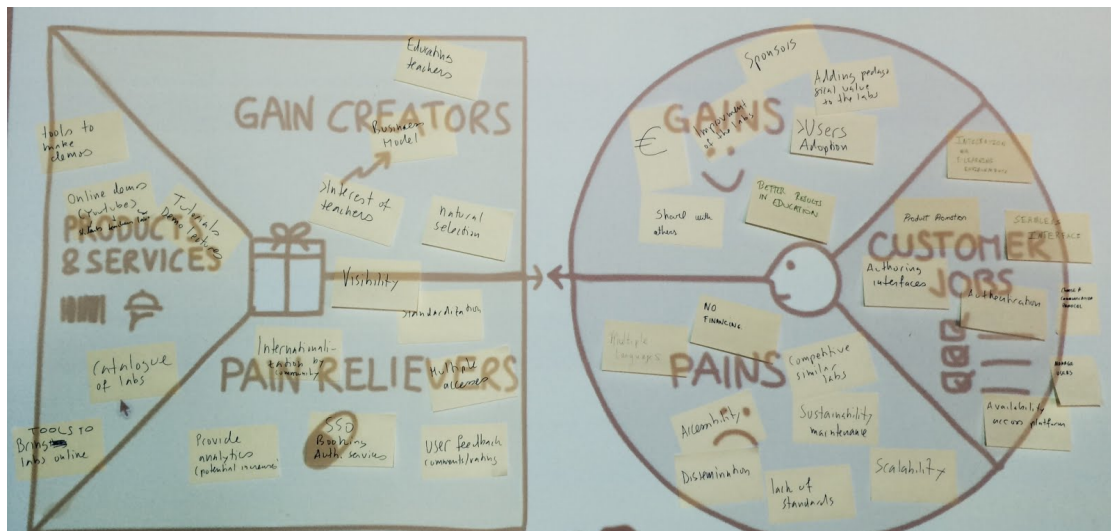


Figure 2: The value-proposition canvas as filled by lab owners

1.3 Integration Levels

To broaden the adoption of the proposed paradigms and ease the integration of new or legacy labs in inquiry learning activities, Go-Lab enables various levels of integration which can be chosen according to the possible resources a lab owner can invest.

Full integration: A lab owner implements the Smart Device specification (see Section 2) on a chosen lab server and makes the necessary adaptation to interface it with the real equipment and its instrumentation. An OpenSocial client app is developed, which can be fully integrated in inquiry learning spaces, exploits the available Go-Lab services and the lab interoperates with the available Go-Lab support apps.

Intermediary integration: A lab owner exploiting a legacy remote laboratory management system implements a plug-in to interface an existing remote lab through the Smart Gateway (see Section 3). An OpenSocial client is developed, which can be fully integrated in inquiry learning spaces, exploits the available Go-Lab services and the lab can interoperate with the available Go-Lab support apps. Different sublevels of integration are possible for this intermediary integration, for more details see Section 3.

Low integration: A lab owner exploiting a legacy remote laboratory management system integrates its existing client as an iFrame wrapped in an OpenSocial app. This app can be integrated in inquiry learning spaces, but cannot exploit the available GoLab services, neither interoperate with support apps.

This deliverable is split in two major parts: first, it introduces the Smart Device specification for the lab owner services in Section 2, and afterwards, it

elaborates on the Smart Gateway for the cloud services in Section 3. Furthermore, Section 4 reports on the progress of the standardisation efforts so far, and Section 5 summarises and concludes this deliverable. Additionally, several appendices are provided to further detail some of the work described in this deliverable, i.e., details on the extension of the metadata description language in Appendix A, an example of Smart Device metadata for a lab in Appendix B, a detailed report of the value-proposition canvas created by Go-Lab partners in Appendix C, and a report on the lab owner workshop in Appendix D.

2 Lab Owner or Plug Services

Through the Smart Device specification, lab owners can plug their labs easily into the Go-Lab infrastructure (as described in Task 4.1 of the DoW). The Smart Device paradigm revisits the traditional client-server approach on which many remote lab implementations rely. The main differences between existing implementations and the Smart Devices are the complete decoupling of the server and the client. This decoupling removes the umbilical cord between the two so that they can live their own separate life. While in a traditional client-server service-oriented architecture (SOA), the service and client share a specification that is often unique to them, the Smart Device defines one common specification for all Smart Devices. This Smart Device specification describes well-defined communication and interfaces between client-server and sufficient information is provided by the server to generate the client applications or reuse existing client applications based on this Smart Device specification. Since the specification is common to many Smart Devices, client apps are not tightly coupled to one server, which encourages interoperability and reuse.

This document specifies both the interface and the communication specification so that a Smart Device-based solution can easily be plugged in the Go-Lab infrastructure. Similarly client applications can be reused or automatically designed and plugged in the GoLab infrastructure. This paradigm can be extended to the Smart Gateway (see Section 3) where a dedicated proxy extends the functionality of an existing solution that does not yet have the required capabilities.

Smart Devices mainly provide services to access the real world through actuators and sensors (Thompson, 2005). The Smart Device interface or API differs from traditional solutions that often provide a monolithic interface without the possibility to access a specific service. There is no assumption regarding the communication channels for Smart Devices (Cascado et al., 2011). The Internet is the de facto choice for online labs (Auer, Pester, Ursutiu, & Samoila, 2003; Salzmann & Gillet, 2008). In addition, in Go-Lab the specific choice of open Web technologies to enable a broader compatibility and adoption has been made. Proprietary technologies will specifically be avoided since they break the core ubiquitous access requirement.

The Smart Device may not necessarily provide a User Interface (UI), but often propose minimal UI that can be rendered at the client side. This means that the client device may render UIs from different providers. Since the Smart Device fosters and promotes the reuse of existing client applications. Web browsers are the preferred environments to render the UI at the client side. There is often a direct relation between a Smart Device service and the app to render the information within the client UI. For example, an oscilloscope app renders the Voltage evolution measured by a sensor of the Smart Device.

The Smart Device provides *services* and *functionalities*. A *service* represents, for instance, a sensor or an actuator that is made available to the outside world (i.e., the client) through the API. Services are fully described and documented

so that a client can use them without further explanation. A *functionality* is an internal behavior provided by the Smart Device, there may be exchange between functionalities and the client application through services but not necessarily. While the required services are fully described through the API, the functionalities are only recommended and best practice guidelines are provided. For example, there can be an actuator service that enables the client application to set the voltage of a motor connected to the server and a functionality that checks that the maximum voltage is not exceeded (and corrects it if needed). The actuator service is well described by the Smart Device metadata, on the other hand the internal validation mechanism is left to the lab owner's discretion since it will be mainly ad-hoc. Still, such a mechanism has to be implemented to ensure that client applications will not break the server and the connected equipment.

2.1 The Smart Device Architecture

As mentioned, the Smart Device specification provides a well-defined set of interfaces that enables communication between the remote lab and external services and applications. Figure 3 provides a basic architecture with a few examples of interactions with the Smart Device. The Figure illustrates a Smart Device that provides a set of interfaces (Section 2.3 describes the interfaces defined as services in more detail). Some interfaces are required, some optional (see Section 2.3). The Smart Device abstracts the implementation of the remote lab. Hence, the specification does not define the communication between the Smart Device and the Remote Lab component in Figure 3. The communication in the left part of Figure 3 is what the Smart Device specifies, namely the protocols and data formats of the interfaces of the Smart Device (i.e., the 'metadata', 'client', 'sensor', 'actuator' and 'logging' interface in Figure 3). For instance, a metadata repository can retrieve the metadata of any Smart Device, index it and provide a lab search engine. Because the interfaces are well-defined, client apps can be reused among Smart Devices. For example, one Data Viewer Client or Learning Analytics Client could retrieve data from any Smart Device and present it to the user.

Additionally, the Smart Device specifies a metadata format that describes the Smart Device, its functionalities and its services. In the remainder of this chapter, we will explain this metadata and each service and functionality.

2.1.1 The Smart Device in the Go-Lab Infrastructure

As described above, the well-defined interfaces of the Smart Device, ensure that a client app and a service can communicate with any Smart Device, if needed. This section will discuss the Go-Lab platforms and services that interact with the Smart Device. The overview component UML diagram is shown in Figure 4. In addition to enabling user interaction with the remote lab, the Smart Device interacts and enables the following features in the Go-Lab infrastructure:

- *Publishing labs on the Lab Repository:* A lab owner can publish any lab on

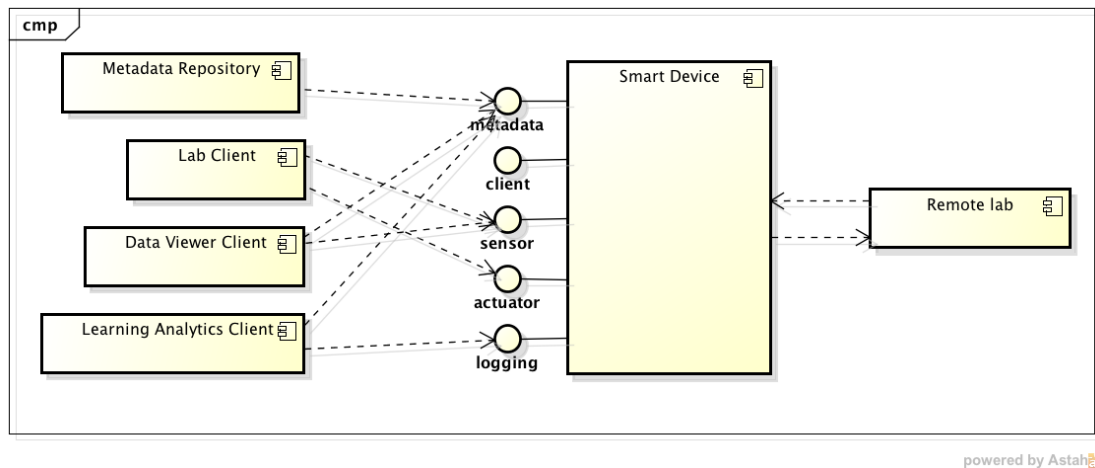


Figure 3: UML Component diagram of different clients making use of the most common Smart Device services (arrows represent calls).

the Go-Lab Lab Repository¹ (see D5.2). If a lab supports the Smart Device specification, the metadata of the Smart Device can be retrieved and parts of the required metadata (see D2.1) can be automatically filled in. Additionally, the client apps to control the lab can be added automatically. In step 1.1 in Figure 4, the Metadata Annotator of the Lab Repository can retrieve the metadata of the Smart Device and the client apps provided by the lab owner in step 1.2.

- *Retrieving lab metadata:* The automatically created metadata can then be retrieved by the Learning Analytics backend services for analysis purposes (see D4.2). Step 2 illustrates how the lab metadata can be retrieved from the Metadata Manager in the Lab Repository by the Artefact Manager, as documented in D4.2.
- *Tracking user activity:* The Smart Device contains a user activity logging service that enables the delivery of learning analytics. Step 3.1 shows how an app on the ILS Platform (see D5.2) can retrieve user activity information from the Smart Device and passes it to Shindig (step 3.2), to the ILS Tracking Agent (step 3.3) and then to the Action Logging Service of the Learning Analytics Backend Services where the user activity is stored. This process is explained in more detail in D4.2.
- *Booking a lab:* The Smart Device itself does not contain a booking mechanism, but makes use of existing booking mechanisms. If a Smart Device requires booking, a user retrieves a booking authentication token from the Booking System and with this authentication token, the user can authenticate with the Smart Device, as described in D4.2. The Smart Device itself only contains logic to validate the authentication token provided by the user. Step 4 illustrates that the Smart Device has an Authentication component that validates authentication tokens with the Booking System

¹Golabz, <http://www.golabz.eu>

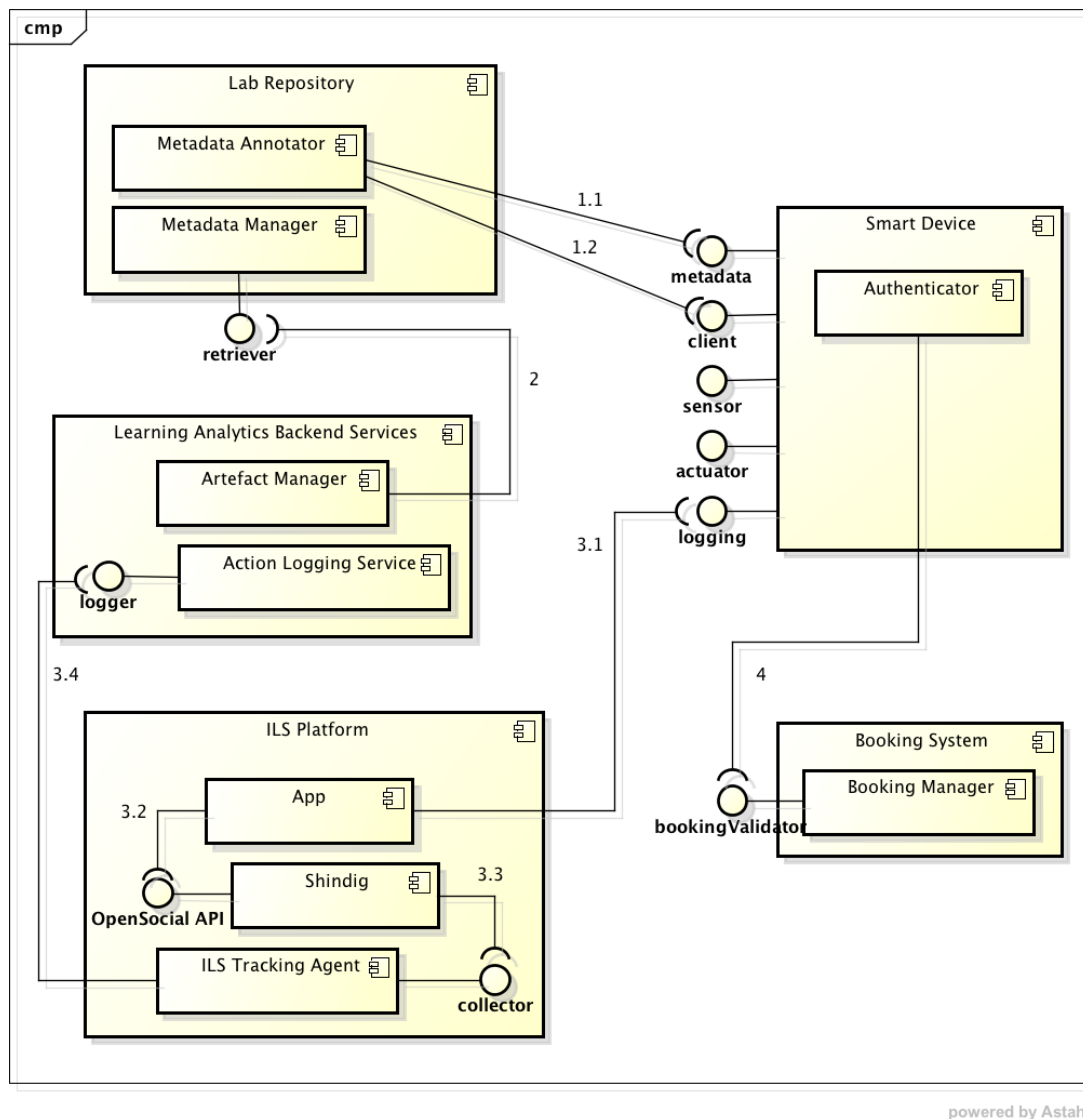


Figure 4: UML Component diagram of the interactions between different Go-Lab services and the Smart Device.

(see D4.2 and Section 2.4.1).

Note, the above features will only be available if the corresponding Smart Device services are implemented. Publishing and retrieving lab metadata will work for any Smart Device because the metadata service is required, but the tracking of user activity makes use of an optional logging service and the booking will obviously only be available when booking is needed. In Section 2.3, we will further elaborate on the different Smart Device services and whether they are required.

2.2 Smart Device Protocols and Technical details

Since, we want to enable access to remote laboratories via the Internet and use a Web-based ILS Platform (see D5.2), the Smart Device should enable

Web clients to connect to its interfaces. Therefore, we will rely on standardised Web protocols to provide the data transfer between the Smart Device and external services and applications. Typically, widely used candidates are HTTP and recently also WebSockets. There are different types of HTTP-based Web Services available, such as SOAP and REST. The problem with most HTTP-based Web Services is that they are synchronous and follow a request-response schema. In such solutions, data can often only be ‘pulled’ from the server and the server cannot initiate a ‘push’ of information to the clients (Push technology)². Remote laboratory experiments, often require asynchronous data transfer. For instance, an experiment that takes a long, unknown time to complete, should preferably be able to push its results to the clients. This is possible using HTTP-based solutions, but the solutions are often very inefficient, since they use the underlying synchronous methods, e.g. long polling³.

WebSockets⁴ on the other hand are asynchronous by nature and allow both pulling and pushing. This provides a bi-directional, full-duplex communication channel between the server and the browser. WebSockets are a recent technology, but are currently supported by all modern browsers⁵. Since WebSockets enable both push and pull technology in an efficient way and with less programming effort than for instance REST and SOAP, we have selected WebSockets as the protocol for the Smart Device services. Only the metadata service, which just requires text retrieval, will be provided via HTTP GET, so it is very easily accessible and can just be hosted as a file on a Web server.

In addition to these two decisions, we make the following recommendations for the protocols of the Smart Device:

- The Web server for the metadata best runs on *port 80 or 443* to ensure access behind institutional and corporate firewalls.
- The WebSocket server preferably also runs on *port 80* for the same reasons.

2.3 Smart Device Services and Functionalities

2.3.1 Introduction

As mentioned above, a Smart Device consists of a set of well-defined services that enable interoperability with external applications and internal functionalities. In this section, we will elaborate on the Smart Device’s services and functionalities. First, we provide an overview of all services and functionalities. Then, we will elaborate on each service and functionality separately.

First, we will introduce some terminology:

- We use the terms *sensors* and *actuators* to reflect the information direction

²Push Technology – Wikipedia, http://en.wikipedia.org/wiki/Push_technology

³Long Polling – Wikipedia, http://en.wikipedia.org/wiki/Push_technology#Long_polling

⁴Websocket specification – Wikipedia, <http://tools.ietf.org/html/rfc6455>

⁵Can I use Web Sockets?, <http://caniuse.com/websockets>

relative to the Smart Device. In this section, the various representations of sensors and actuators will be defined. For example, a sensor enables the reading of a value from a thermometer. An actuator enables the setting of a value, for example setting a motor voltage.

- Sensors and actuators can be *physical* (temperature sensor), *virtual* (computed speed derived from a position measurement) or *complex* that represents an aggregation of sensors/actuators (knobs that form the front panel of an oscilloscope).
- Both the sensor and the actuator can be configured, see the metadata service in Section 2.3.3.

2.3.2 List of Services and Functionalities

This section lists required and optional services, and functionalities of a Smart Device. We will elaborate on each one in the next sections.

Required Services:

Metadata service: This service returns a description of the lab, its mechanisms and external services. The information provided by this service should be sufficient to programmatically define a UI and the related client-server communication.

Sensor service: This service returns data from a ‘sensor’ of the remote lab.

Actuator service: This service allows to control an ‘actuator’ of the remote lab.

Optional Services:

Client app service: a list of lab client applications may be provided by the Smart Device.

User activity logging service: This optional service provides a method to retrieve logged user actions. This service can be based on the ‘Logging and Alarms’ functionality discussed below.

Models service: Various information about the connected equipment can be sent to the client application. For example, a 3D (or 2D) graphical model describing the connected equipment could be defined in the form of a VRML file. Similarly, a mathematical model describing the connected equipment can be sent in the form of dynamical equations, these equations could be used to simulate the equipment at the client side. Both these models could be read or modified through the means of virtual sensors/actuators.

Functionalities – Best Practices:

Internal functionalities are suggestions to be implemented in the Smart Device. Therefore, we provide only best practices. These functionalities are often ad-hoc and strongly related to the connected equipment, it is thus difficult to give a precise specification.

Authentication functionality: The Smart Device does not comprise a booking system. It can make use of an external booking system, such as the Go-Lab

booking system (see D4.2). As described in D4.2, the Go-Lab booking system will provide an authentication token upon the creation of a booking. With this authentication token, a user can connect to the Smart Device. The Smart Device then needs to contact the booking system to validate whether the user is currently allowed to access the Smart Device. Thus, the Smart Device implementation requires limited effort, compared to providing its own authentication and booking mechanisms. We will briefly summarise the D4.2 specification in Section 2.4.1.

Self and known state functionality: This functionality is recommended and its precise implementation is left to the lab owner's discretion. This functionality ensures that the remote lab is left in a proper state after the current experimentation session is completed so that the next user will be able to use it. Similarly, after a power outage the system should be able to come back to a predefined state. Remote experiments are supposed to be conducted remotely and thus no one is expected to be in the neighbourhood of the experiment to put it back in a known state. In addition, connections could occur anytime during the day or night. Thus, the system should be as autonomous as possible. This implies an adequate design of the experiment and a defensive software design that is able to adapt to 'any' situation. The lab owner should implement the following procedures in the Smart Device and its hardware:

- automatic initialization at startup (regular or power outage)
- reset to a known state after the last client disconnect
- calibration after some time if it makes sense

Security and local control: This functionality is recommended and its precise implementation is left to the lab owner's discretion. At all time the security of the server and its connected equipment must be ensured. All commands received or computed should be validated before being applied to the connected equipment. This step may require the addition of a local controller to track the state of the connected equipment, for example a speed increase may need to follow a ramp before being applied to a motor. The controller parameters could be read or modified through the means of virtual sensors and actuators. Experienced lab owners know that users will try to put the system to its limits. These limits are not only the physical limit of a given sensor/actuator, but the pattern of the applied signal to a given sensor over time may also need to be considered. For example applying 5V to a 10V motor is without risk if it is applied once. On the other hand, applying a +/- 5V square pattern to the same motor for an hour may destroy it. Since the Smart Device may be connected to the real world via its actuators, it is essential to validate all values applied the actuators considering potential external constraints. This validation process could be simplified by an adequate design of the experiment itself, this may include an additional sensor to measure the environments in which the Smart Device operates. The lab owner should implement the following procedures in the Smart Device:

- *value validation* before applying them to the actuator, considering actuator range and other temporal considerations.

- *actuator state validation* to check if the command to be applied is safe for the outside world.

Logging and alarms: This functionality provides a way to log session information as well as user interactions. In addition, it can also include logging information specific to the lab itself. In case of problems (e.g. malfunction or power outage) alarms may be automatically triggered by this functionality. The Smart Device will be online unattended for an extended period of time. It is primordial to have a method to perform post mortem (after the problem occurred) analysis. The user action is the first information to be logged, this information can be accessible via the user activity logging service (see Section 2.3.6). But extra information should also be logged, for example the state of the system via the available sensors and the environment (for example room temperature) if there is a method to measure it. Note that sensors may be available internally to the Smart Device but not necessarily accessible via the sensor service. The lab owner could implement the following information in the Smart Device:

- log user actions
- log the complete system state
- log its surrounding state

Also, by definition the Smart Device is connected to the Internet and has no knowledge of the client device, thus it needs to take proper action to save itself from abuse. A firewall or a DMZ⁶ may protect it from external intruders or menaces. While some hostile actions may be reduced using such mechanisms, the Smart Device should add internally additional measures:

- validate the requests sent by the client even though the client is correctly identified
- throttle continuous requests of a malicious client application
- log all internet connections and request for later analysis

If an unexpected event occurs, its potential danger should be assessed by the Smart Device and eventually an alarm may be triggered (and the logger). This alarm may follow a completely different path than the other information path. An alarm concerning the lack of the Internet connection should not be sent to the lab owner through the internet but via for example SMS.

Local simulation: a local simulation might be proposed to the client if the equipment is used by someone else. The simulation data and parameters could be read or modify through virtual sensors/actuators. A mathematical model describing the state of the physical equipment connected may be available. This model can be made available to the client via the Models service and the client application designer may decide to use this model to simulate the connected physical equipment. This simulation requires computational resources that may not be available at the client device. Thus it is possible to perform this computa-

⁶Demilitarized Zone (DMZ), [http://en.wikipedia.org/wiki/DMZ_\(computing\)](http://en.wikipedia.org/wiki/DMZ_(computing))

tion at the server side and send the result to the client application using virtual sensors and actuators.

2.3.3 Metadata Service

The metadata service is a required service that is at the core of the interoperability provided by the Smart Device specification. This service provides on the one hand a general description of the lab, which is useful for publishing a Smart Device lab into the lab repository as discussed in Section 2.1.1. On the other hand the metadata provides technical details of the lab. This technical information creates the interoperability features of a Smart Device and can be used to generate user interfaces automatically.

First, this section will elaborate on different, existing web service description languages and explain which description language we have chosen. Second, we explain the design choices behind the metadata specification. Afterwards, the metadata for the required services is described and we explain how one can add metadata for extra services. Finally, we specify how the metadata should be made accessible from a Smart Device.

Comparison of Web Service Description Languages

To describe the metadata of a Smart Device we investigated several options to describe Web service specifications. The main goal was not to reinvent the wheel and use robust and complete specifications if possible. Furthermore, some specifications allow the automatic generation of client applications. Since we did not find Web service description languages specific to the WebSocket protocol, we have considered SOAP and REST-based description languages for inspiration.

One of the most popular Web service description languages is WSDL⁷, which originally focused strongly on SOAP Web services and provides better support for RESTful Web services since version 2.0. However, currently software support for WSDL 2.0 is often poor⁸. WSDL is also XML-based and in all other specifications we have opted for JSON. On request of the JavaScript community, JSON-WSP⁹ was created. Nonetheless, the JSON-WSP specification did not gain traction.

There are also description languages dedicated to RESTful services. WADL (Hadley, 2009) is an XML-based description language which can be considered as the REST equivalent of the original WSDL specification for SOAP. Similar to WADL is RSDL¹⁰, also an XML-based language but more focused on the structure of the Web service URIs. Another option is RAML¹¹ that uses the YAML format and also relies on markdown for descriptions and JSON Schema¹².

⁷Web Services Description Language (WSDL) 1.1, <http://www.w3.org/TR/wsdl>

⁸Web Services Description Language – Wikipedia, http://en.wikipedia.org/wiki/Web_Services_Description_Language

⁹JSON-WSP – Wikipedia, <http://en.wikipedia.org/wiki/Jsonwsp>

¹⁰RESTful Service Description Language (RSDL), <http://en.wikipedia.org/wiki/RSDL>

¹¹RESTful API Modeling Language (RAML), <http://raml.org/>

¹²JSON Schema specification – JSON Schema: core definitions and terminology <http://json-schema.org/>

For the Smart Device specification, we have opted for Swagger¹³. Swagger is a JSON-based description language meant for RESTful APIs, but we have easily extend it to WebSockets. Swagger aims to describe a web service for both humans and computers. Therefore, it is strongly focused on automatically generating user interfaces¹⁴, which is one of our goals. Swagger is based on JSON Schema to specify the data format of requests and responses. Due to the large and growing list of supporting tools, Swagger is getting good uptake. As mentioned, to achieve the requirements of the Smart Device specification, we had to extend the Swagger specification in a limited way, so our adapted Swagger version still holds all of the Swagger qualities. This was achieved by adding support for WebSockets as a protocol and included some extra metadata blocks as well as some extra values for Swagger fields to fit our needs. In the remainder of this section, we will elaborate on how we have applied and extended Swagger for the Smart Device Specification.

Smart Device Metadata Concepts

As stated above, the goal of the Smart Device metadata is manifold:

- describe the lab (e.g., who is the contact person and describe what the aims of the lab are)
- describe integration with other Go-Lab services (e.g., authentication details with the booking service)
- describe concurrency mechanisms of the lab (e.g., how does the lab allow observations, while someone is doing an experiment?)
- describe and define the services that the Smart Device provides (e.g., specify the format of the requests and responses of a service)

Additional requirements are that the metadata specification should be easily extendable if the Smart Device developer wants to add services. Furthermore, for a simple Smart Device, it would be good if the developer does not have to learn details of the Swagger specification. Based on these requirements, we have made the following design choices:

- *Sensor & actuator metadata service*: The metadata that describes the available sensors and actuators is provided by a service. In this way a developer of a simple Smart Device needs to edit just a few lines in the metadata and does not need to add complex descriptions of actuators and sensors. The Smart Device software packages provided by Go-Lab can already implement these services so the developer just has to add the return values in the code. This also enables the developer to keep the sensor and actuator metadata very close to the actual implementation of

schema-core, <http://json-schema.org/latest/json-schema-core.html>

¹³Swagger website, <http://swagger.wordnik.com/>

¹⁴To showcase the automatic user interface generation in Swagger, they have a demo available that allows anyone through a simple user interface to interact with a sample Web service, see <http://petstore.swagger.wordnik.com/>. This UI is completely generated solely based on the Swagger description

the logic that measures sensors and controls actuators.

- *Service names*: Each service needs to have a method name ('nickname' in Swagger parlance) (e.g., the service for the sensor metadata is called 'getSensorMetadata') and each request and response of a service needs to pass the method name. By adding this extra metadata to the service communication data format, it is possible to channel calls to different services over one WebSocket (see Section 2.4.3 for more details). Furthermore, the nicknames are used to control access to services (more on this below).

There are other small design choices made that do not have global impact. These will be discussed in the following sections where needed.

General Smart Device Metadata Specification

The official Swagger RESTful API documentation specification can be found on <https://github.com/wordnik/swagger-spec/blob/master/versions/1.2.md>. The Swagger specification is typically split over multiple files. The file with the general metadata is physically located in the root path of the service (e.g. <http://smartlab.golab.eu/service>). Then there are different files for each separate service (e.g. if the sensor service is located at <http://smartlab.golab.eu/service/sensor> then there will be a Swagger file with the specification of the sensor service). In the case of WebSockets this makes less sense, since there is not necessarily an HTTP path available. So we have opted to provide one specification file, containing the general metadata and all service-specific metadata. Appendix A contains a full Swagger specification for a Smart Device. This example is also available on GitHub.¹⁵

This section will introduce the general structure of the adapted Swagger file. The example code snippet below (Listing 2.3.3), demonstrates five parts: (1) Swagger related metadata, (2) a list of APIs, (3) the authorisation mechanisms, (4) the Smart Device concurrent access mechanisms and (5) information of the service in general.

Listing 2.1: The general structure of the Smart Device metadata based on Swagger.

```
{
  "apiVersion": "1.0.0",
  "swaggerVersion": "1.2",
  "basePath": "http://redlab.epfl.ch/smartdevice",
  "apis": [
    {
      "path": "/client",
      "description": "Operations about clients for the lab",
      ...
    },
  ],
}
```

¹⁵The Swagger specification for an example Smart Device, <https://github.com/Go-Lab/smart-device-metadata>


```

    {
      "path": "/sensor",
      "description": "Operations about sensors",
      ...
    },
    {
      "path": "/actuator",
      "description": "Operations about actuators",
      ...
    },
    {
      "path": "/",
      "description": "A general endpoint that allows to access any
        operation of any service",
      ...
    }
  ],
  "authorizations": {
    "goLabBooking": {
      "type": "apiKey",
      "passAs": "query",
      "keyName": "authToken",
      "authServiceUrl": "http://booking.golabz.eu/auth"
    }
  },
  "concurrency": {      /* Swagger extension: */
    "interactionMode": "synchronous",
    "concurrencyScheme": "roles",
    "roleSelectionMechanism": ["race", "interruptor"],
    "roles": [
      {
        "role": "observer",
        "selectionMechanism": ["race"],
        "availableApis": ["getSensors"]
      },
      {
        "role": "controller",
        "selectionMechanism": ["race"]
      },
      {
        "role": "admin",
        "selectionMechanism": ["interruptor"]
      }
    ]
  },
  "info": {
    "title": "RED Lab Smart Device",
    "description": "This is an example implementation of the Go-Lab
      Smart Device in LabView and demonstrates a mechatronics

```

```

        remote lab running at EPFL",
        "termsOfServiceUrl": "http://helloeverb.com/terms/",
        "contact": "christophe.salzmann@epfl.ch",
        "license": "Apache 2.0",
        "licenseUrl": "http://www.apache.org/licenses/LICENSE-2.0.html"
    }
}

```

Swagger-related Metadata: Looking closer at the sample snippet, the Swagger specification requires the following fields to declare the version of Swagger and the API. The version of Swagger should not be changed by the developer.

Listing 2.2: Swagger-related metadata example

```

"apiVersion": "1.0.0",
"swaggerVersion": "1.2",

```

API Metadata: The abridged snippet below lists all API endpoints of the Smart Device and the root URL path of the service in the 'basePath' field. Each API endpoint has a path and a basic description. The other fields will be discussed in the sections specific to each service. We can also provide a general API endpoint, with the path '/', where a WebSocket can connect to any service of the Smart Device. This is useful to channel calls in one WebSocket (see Section 2.4.3 for more details).

Listing 2.3: API metadata example

```

"basePath": "http://redlab.epfl.ch/smartdevice",
"apis": [
  {
    "path": "/client",
    "description": "Operations about clients for the lab",
    ...
  },
  ...
  {
    "path": "/",
    "description": "A general endpoint that allows to access any
      operation on any service"
    ...
  }
]

```

Authorisation Metadata: Swagger supports common REST-based authentication and authorisation mechanisms, e.g., OAuth. For the Go-Lab booking system (see D4.2), we have decided to use token-based authorisation, which is very similar to the 'apikey' type that Swagger supports by default, but it is a temporary API key for the duration of the booking. The snippet below also defines the authorisation service endpoint of the booking system. If future implementa-

tions prove that this is insufficient, we will provide a custom Go-Lab solution in D4.5.

Listing 2.4: Authorisation metadata example

```
"authorizations": {
  "goLabBooking": {
    "type": "apiKey",
    "passAs": "query",
    "keyName": "authToken",
    "authServiceUrl": "http://booking.golabz.eu/auth"
  }
}
```

Concurrent Access Metadata: We have extended Swagger to be able to model the concurrency models of remote labs, so client applications can handle the different mechanisms appropriately. Different concurrency mechanisms exist and it is up to the lab owner to decide upon the appropriate scheme for his lab. The 'concurrency' metadata field is meant to describe such mechanisms, as shown in the following example snippet:

Listing 2.5: Concurrent access metadata example

```
"concurrency": {      /* Swagger extension: */
  "interactionMode": "synchronous", /* can also be 'asynchronous' */
  "concurrencyScheme": "roles", /* can also be 'concurrent' then all
    users have access at the same time */
  "roleSelectionMechanism": ["race", "interruptor"], /* can also be
    'queue', 'fixed role', 'dynamic role' */
  "roles": [
    {
      "role": "observer",
      "selectionMechanism": ["race"],
      "availableApis": ["getSensors"] /* a list of paths or
        operation nicknames */
    },
    {
      "role": "controller",
      "selectionMechanism": ["race"]
    },
    {
      "role": "admin",
      "selectionMechanism": ["interruptor"]
    }
  ]
}
```

One can interact with a lab in a synchronous or asynchronous way. If the lab is synchronous, the users are interacting directly with the lab. If there are other

people using the lab concurrently the user is aware of their actions. If the lab is asynchronous, the user typically prepares an experiment, submits it to the lab and waits to get results back. Users are not aware of other users in this case.

The rest of this metadata is for synchronous labs, since asynchronous labs can internally deal with concurrency issues. There are typically two concurrency schemes possible 'concurrent' and 'roles' (modelled in the 'concurrencyScheme' field). Either the lab allows users to operate the experiment at the same time or provides different user roles to control access to the experiment. Like in other computer access control mechanisms, user roles limit the control to the system, in our case the services. Different mechanisms (modelled in the 'roleSelectionMechanism') are used to switch roles, we have identified the following options:

- *fixed role*: The user cannot be promoted from one role to another, e.g. the teacher can control the Smart Device but the students can only observe.
- *dynamic role*: The user's role can change during the session, e.g. a user who is observing an experiment can at a later point control it.
- *race*: The user who tries to access the Smart Device at the time no other user is using it, gets access. If the Smart Device is busy, the user has to retry until it is available.
- *queue*: When a user accesses the Smart Device, she is added to a waiting queue and she will get access when the others before her have finished.
- *interruptor*: The user has the ability to abort the session of the currently active user and take control of the Smart Device.

Finally, each role can be described in the 'roles' field. Each role has a name (in the 'role' field) and the role selection mechanisms that this role has are listed in 'selectionMechanism'. Furthermore, the role can declare which services will be accessible if you are assigned that role in the 'availableApis' field. This concurrency related metadata will be validated in future Smart Device implementations and will be updated if needed in D4.5.

General metadata: The general metadata provides information about the lab, such as the name, a short description, a contact person, and licensing information. These are default Swagger fields that can be useful when publishing a lab on the Lab Repository. The snippet below provides an example:

Listing 2.6: General metadata example

```
"info": {
  "title": "RED Lab Smart Device",
  "description": "This is an example implementation of the Go-Lab
    Smart Device in LabVIEW, it demonstrates a mechatronics remote
    lab running at EPFL",
  "termsOfServiceUrl": "http://redlab.epfl.ch/terms/",
  "contact": "christophe.salzmann@epfl.ch",
  "license": "Apache 2.0",
```

```

    "licenseUrl": "http://www.apache.org/licenses/LICENSE-2.0.html"
  }

```

Service Metadata Specification

As mentioned each Smart Device service needs to be declared in the Swagger specification. To do this, a JSON object needs to be added to the 'apis' field of Swagger and optionally new data models need to be added to the 'models' field (see below). However, we have tried to design the specification, so that for simple Smart Devices developers do not need to learn how to describe a service in the metadata. Go-Lab provides reusable service metadata descriptions for the sensor, actuator and logging services. In this section we will present a simple example of a service Swagger specification, namely the user activity logging service. Afterwards we will elaborate on the metadata for the sensor and actuators.

User activity logging service – getLoggingInfo:

The user activity logging service is described in Section 2.3.6 in more detail. But essentially, we want to return ActivityStream objects (see D5.1) of the user activity, to the client. The following snippet provides the description of the service and its data models:

Listing 2.7: An example of a service declaration – the user activity logging service.

```

"apis": [
  {
    "path": "/logging",
    "description": "Returns the user activity of the current user in
      ActivityStream format",
    "protocol": "WebSocket",
    "operations": [
      {
        "method": "Send",
        "nickname": "getLoggingInfo",
        "summary": "Streams the current logging information of
          the user activities and the lab activities",
        "notes": "Returns a JSON array of Activity Stream
          objects, see http://activitystrea.ms/",
        "type": "LoggingInfoResponse",
        "webSocketType": "text",
        "produces": "application/json",
        "parameters": [
          {
            "name": "message",
            "description": "the payload for the getLoggingInfo
              service",
            "required": true,
            "paramType": "message",
            "type": "SimpleRequest",

```

```

        "allowMultiple": false
      }
    ]
  },
  "responseMessages": [
    {
      "code": 401,
      "message": "Unauthorised access. The authentication token
        is not valid"
    },
    {
      "code": 402,
      "message": "Too many users"
    },
    {
      "code": 405,
      "message": "Method not allowed. The requested method is
        not allowed by this server."
    },
    {
      "code": 422,
      "message": "The request body is unprocessable"
    }
  ]
}
],
"models": {
  "LoggingInfoResponse": {
    "id": "LoggingInfoResponse",
    "required": [
      "method", "logs"
    ],
    "properties": {
      "method": {
        "type": "string"
      },
      "logs": {
        "type": "array",
        "items": {
          "type": "object",
          "description": "An Activity Stream object. This JSON
            object should follow the ActivityStreams 1.0 JSON
            specification described at
            http://activitystrea.ms/specs/json/1.0/"
        }
      }
    }
  }
}
},

```

```

    "SimpleRequest": {
      "id": "SimpleRequest",
      "required": [
        "method"
      ],
      "properties": {
        "authToken": {
          "type": "string"
        },
        "method": {
          "type": "string",
          "description": "The method should be equal to the
            nickname of one of the provided services."
        }
      }
    }
  }
}

```

To add a service one needs to add a JSON object to the 'apis' array (here only the discussed element is shown for brevity) and add JSON objects describing the necessary data models to the 'models' array (if applicable).

The API object contains the path and description as mentioned before, and also an optional 'protocol' field to express that the service uses WebSockets. This is our extension to Swagger to support WebSockets and it could also have the value 'HTTP' in case REST services are needed. Then the Swagger specification declares a list of 'operations' that contain all services and 'responseMessages' that contain all error messages that the service can return (relying on HTTP status codes (*Hypertext Transfer Protocol (HTTP) Status Code Registry (RFC7231)*, 2014)). In this case, there is only one service with the nickname 'getLoggingInfo'. Then one can specify the protocol method, in case of WebSockets this is 'Send'¹⁶. If the HTTP protocol would be used, the methods can be GET, PUT, POST, DELETE, etc. Another Go-Lab extension to support WebSockets is 'WebSocketType' to enable the configuration of 'text' or 'binary' WebSockets. Binary WebSockets can make the transmission of binary data much more efficient, e.g. this is useful for video streaming. Additional documentation can be provided in the 'summary' and 'notes' fields. Next, the service arguments and results can be configured. The 'type' field contains either a JSON Schema primitive data type^{17,18} or the ID of a model in the 'models' list. In this case it refers to the 'LoggingInfoResponse' data model, which we will elaborate on below. Then one can also model the response media type using the

¹⁶When WebSockets are used, the 'Send' method can be omitted since there typically is only one WebSocket method. However in some WebSocket dialects other methods can be available that have to be defined, e.g. Socket.io also has a 'emit' method (see <http://socket.io/docs/>).

¹⁷Swagger RESTful API Documentation Specification, <https://github.com/wordnik/swagger-spec/blob/master/versions/1.2.md>

¹⁸JSON Schema specification – JSON Schema: core definitions and terminology json-schema-core, <http://json-schema.org/latest/json-schema-core.html>

'produces' field, which can contain any Internet Media Type (Freed, Baker, & Hoehrmann, 2014). This can be particularly useful for a service that returns for example images or structured text. The 'parameters' field contains a list of arguments that can be passed to the service. Typically this is only one field and it is a request data model. We have provided a simple request model, namely 'SimpleRequest'. More complex request models can of course be defined by the developer when needed.

Both the request and response data models are available in the 'models' array. The models used by other services have been omitted in Listing 2.7 for brevity. These models are expressed in JSON Schema. JSON Schema models have an 'id', 'required' and 'properties' fields. The 'id' field is required and is used to reference the model, e.g. in the 'type' fields. The 'properties' field contains a list of fields of the data model. The 'required' field lists all fields of the 'properties' field that must be provided in the data model. The elements of the 'properties' array express the data type of each of the data model fields. For example, the 'SimpleRequest' model has an 'authToken' field of the type 'string' and a 'method' field of the type 'string'. As mentioned before, this 'method' field should contain the nickname of the service, i.e., 'getLoggingInfo'. Looking at the 'LoggingInfoResponse' data model, one notices again the 'method' field and the 'logs' field which is an array with JSON objects that should be ActivityStream objects. We did not express the whole ActivityStream data model here.

For more information on how to add a new service we refer to the Swagger specification (*Swagger RESTful API Documentation Specification*, n.d.), the JSON Schema specification and Appendix A, which lists how we have extended Swagger.

Sensor Metadata Service – getSensorMetadata:

As mentioned above, the metadata that describes the Smart Device sensors and actuators are provided via a service and not in the metadata description itself. In this section we will elaborate on the sensor metadata. We will describe how one can call the service and the responses returned. The Swagger sensor service specification can be found in Appendix B.

The service is called 'getSensorMetadata' and can be called with a 'SimpleRequest' data model, which is just a JSON object with a 'method' field and an optional authentication token (which is not needed to retrieve metadata):

Listing 2.8: Request example of getSensorMetadata service.

```
{
  "method": "getSensorMetadata"
}
```

This returns an array of sensors describing each sensor made available to the outside world (i.e., a client or external service). The following example shows two sensors: a 3D acceleration sensor and a video stream.

Listing 2.9: Response example from getSensorMetadata service.

```

{
  "method": "getSensorMetadata",
  "sensors": [
    {
      "sensorId": "3Dacc",
      "fullName": "3D acceleration",
      "description": "the 3D acceleration of the robot handle",
      "websocketType": "text",
      "produces": "application/json",
      "values": [
        {
          "name": "X",
          "unit": "m*s^-2",
          "lastMeasured": "2014-06-23T18:25:43.511Z",
          "rangeMinimum": -100.00,
          "rangeMaximum": 100.00,
          "rangeStep": 0.10,
          "updateFrequency": 10 /* in Hertz */
        },
        /* Repeat for 'Y' and 'Z' acceleration */
        {
          "name": "Y",
          ...
        },
        {
          "name": "Z",
          ...
        }
      ]
    }
  ]
}

```

Several sensor or actuator configurations may be required. In the 'configuration' field, the different parameters to configure the sensor can be described using JSON Schema compliant primitive types or models.

```

  "configuration": [
    {
      "parameter": "precision",
      "description": "The precision is expressed as a power
        of 10, e.g. to allow a precision of 0.01 the value
        will be -2 (from 10^-2).",
      "type": "int"
    }
  ],

```

The accelerometer will continuously stream information to the client. The 'user-ModifiableFrequency' field describes if the interval can be modified or not.

```

        "accessMode": {
            "type": "push",
            "nominalUpdateInterval": 100, /* in ms */
            "userModifiableFrequency": true
        }
    },

```

Streaming video to the client is an essential service that a Smart Device should provide through a sensor. Actually the video image could either be seen as a pixmap (array of pixel values) or as an encoded image, for example JPEG encoded. The later being 10% to 90% smaller in size. The example below shows the image encoded as a JPEG image (specified as a media type in the 'produces' field). The JPEG encoding results in binary data that may contain specific characters that could be interpreted as control characters, thus either the JPEG data is transmitted through a binary WebSocket (recommended) or it is BinHex prior to be sent using a textual WebSocket (defined in the 'websocketType' field). The metadata also describes which configuration parameters are exposed to the client.

Listing 2.10: Example of a video stream sensor and its configuration

```

{
    "sensorId": "video",
    "fullName": "video stream",
    "description": "front camera video stream",
    "websocketType": "binary",
    "singleWebSocketRecommended": true,
    "produces": "image/jpeg",
    "values": [
        {
            "name": "front",
            "lastMeasured": "2014-06-23T19:25:43.511Z",
            "updateFrequency": 10
        }
    ],
    "configuration": [
        {
            "parameter": "width",
            "type": "int"
        },
        {
            "parameter": "height",
            "type": "int"
        },
        {
            "parameter": "compression",
            "description": "The JPEG compression quality, ranging
                from 0 (lowest quality) to 100 (highest quality).",

```

```

        "type": "float"
      },
      {
        "parameter": "colourFilter",
        "description": "The colour value in an array of 3
          decimal RGB values",
        "type": "array",
        "items": "int"
      }
    ],
    "accessMode": {
      "type": "stream",
      "nominalUpdateInterval": 10,
      "userModifiableFrequency": true
    }
  }
]
}

```

Each *sensor* carries the following information:

- *sensorId*: the ID that will be used to identify the sensor, '3D-acc'.
- *fullName*: the sensor full name, ex. '3D acceleration'.
- *description*: a more detailed description of the sensor, 'the 3D acceleration of the robot handle'.
- *websocketType*: the type of WebSocket, it can either be 'text' or 'binary', the default is 'text'. 'binary' WebSockets are mainly used for video streaming. Using binary WebSockets is more efficient, since the data does not need to be BinHex-ed.
- *produces*: defines the Internet media type (Freed et al., 2014) of the response provided by the sensor service, it is typically `application/json` for a JSON encoded response. However for a video streaming sensor that supports JPEG compression, it should be `image/jpeg`.
- *values[]*: describes the array of values for a given sensor. For a single value sensor like a temperature sensor the 'values' array contains only one element. For a complex sensor like an accelerometer, the 'values' array contains several elements, for example 3 elements, one for each individual X-Y-Z acceleration of the arm handle. Values contain a name and unit, and can additionally have the last measured time stamp and a range minimum, maximum and iteration step of the range in which the values safely operate. Furthermore, if the value is automatically or continuously measured the rate at which the measurement is updated can be defined in Hertz (s^{-1}) in the 'updateFrequency' field.
- *configuration[]*: describes the possible configuration parameters that are applicable to the sensor. When requesting a sensor value, the here de-

defined configuration parameters can be used to adjust the sensor. These parameters are described by providing a name (in the 'parameter' field) and the data type using JSON Schema data types. One can also model array types, as demonstrated in the 'colorFilter' configuration parameter of the video stream sensor. For complex configuration parameters it would also be possible to refer to a JSON Schema data model.

- *accessMode*: provides more information on how the sensor can be accessed. Some sensors can only measure once and others provide a continuous stream of data. Such differences can be modelled using the access mode 'type', which can be 'pull' for sensors that only measure once, and 'push' for sensors who keep on providing measurements over time. For 'push' sensors, one can specify the nominal update interval and whether the measurement frequency can be modified by the user (using respectively the 'nominalUpdateInterval' and 'userModifiableFrequency'). The access mode type can also be 'stream', for instance for streaming video if more complex mechanisms are used than push technology.

As mentioned, both sensors and actuators can be configured, which means that the information goes both ways even for the sensor. For example, the image resolution of a webcam can be set through such configuration. Similarly for actuators some aspects may be set through configuration while the actual value is set through the actor value itself. For example the gain of a power amplifier can be specified through configuration while the actual value that needs to be amplified is set via the actuator value variable. It is expected that sensors and actuators are rarely configured, if a configuration is constantly changed this might indicate that the configuration should be better expressed as a virtual sensor/actuator.

Each sensor value carries the following information when it makes sense:

- *name (required)*: the sensor value name, for example 'X' for the acceleration toward the X axis. for a single value sensor, the name can be omitted.
- *unit*: the unit of the sensor value, for example $m s^{-2}$ for the X acceleration. The set of possible units is almost infinite and each lab owner has probably his preferred set of units. Thus it is difficult to impose units. As a best practice, we recommend to use the SI units (Taylor & Thompson, 2008) and the SI derived units¹⁹.
- *lastMeasured*: the timestamp when the sensor was last measured.
- *rangeMinimum*: the measurement of a sensor can have a lower and upper bound. This field presents the minimum of the interval, if there is one.
- *rangeMaximum*: the maximum of the measurement interval.
- *rangeStep*: The precision of the sensor can be limited or it can only measure at certain points. This field allows to model the discrete steps that a sensor supports.

¹⁹SI Derived Units – Wikipedia, http://en.wikipedia.org/wiki/SI_derived_unit

- *updateFrequency*: how many times per second the sensor is refreshed. This field is expressed in Hertz (s^{-1}).

Actuator Metadata Service – getActuatorMetadata:

Similar to the sensor metadata, the actuator metadata is also provided via a service, named 'getActuatorMetadata'. The service is very similar to the sensor metadata service, so we will only provide examples and discuss the differences.

The service can be called with a 'SimpleRequest' data model, which is just a JSON object with a 'method' and an optional 'authToken' field:

Listing 2.11: Request example of getActuatorMetadata service.

```
{
  "method": "getActuatorMetadata"
}
```

The response is structured as in the following example (see Appendix A & B for details):

Listing 2.12: Response example of getActuatorMetadata service.

```
{
  "method": "getActuatorMetadata",
  "actuators":
  [
    {
      "actuatorId": "motor",
      "fullName": "Wheel motor",
      "description": "operate the motor of the wheel",
      "websocketType": "text",
      "produces": "application/json",
      "consumes": "application/json",
      "values": [
        {
          "name": "left",
          "unit": "radian",
          "rangeMinimum": 0.00,
          "rangeMaximum": 3.14,
          "rangeStep": 0.10,
          "updateFrequency": 10,
          "lastMeasured": "2014-06-23T19:25:43.511Z"
        },
        {
          "name": "right",
          "unit": "radian",
          "rangeMinimum": 0.00,
          "rangeMaximum": 3.14,
          "rangeStep": 0.10,
          "updateFrequency": 10,
          "lastMeasured": "2014-06-23T19:25:43.511Z"
        }
      ]
    }
  ]
}
```

```

        }
      ],
      "configuration": [
        {
          "parameter": "precision",
          "description": "The precision is expressed as a power
            of 10, e.g. to allow a precision of 0.01 the value
            will be -2 (from 10^-2).",
          "type": "int"
        }
      ],
      "accessMode": {
        "type": "push",
        "nominalUpdateInterval": 100,
        "userModifiableFrequency": true
      }
    }
  ]
}

```

As one can see the actuator metadata is almost identical to the sensor metadata. The following differences can be found:

- *actuatorId*: The identifier field has a different name, but identical purpose as for a sensor.
- *consumes*: The 'consumes' field models what data type can be input into the actuator. By default this is JSON, as modelled here 'application/json'. But identical to the 'produces' field, one can set it to any Internet Media Type (Freed et al., 2014).

2.3.4 Sensor Service – `getSensorData`

The sensor and the actuator services are at the core of the Smart Device interaction and they share many elements. The data exchange between clients and the Smart Device is mainly done using these two services.

It is envisioned that for each sensor or actuator there is an equivalent method to render the information at the client side. Typically a client app could render the sensor/actuator information transmitted via a WebSocket. By parsing the Smart Device metadata information these apps could partially adapt to the Smart Device service, this will enable app reuse with other Smart Devices. Similarly, a basic app could be replaced by a more advanced one. For example, let us assume that the Smart Device provides a temperature measurement 'S1', every second. The metadata provides the required information for the app to connect to the corresponding WebSocket. The basic app will just update a text field in the browser (see Figure 5).

The lab user may be interested in having the temperature evolution over time. Another app may provide the mechanism to locally store the last minutes of



Figure 5: The basic app that renders the temperature sensor S1

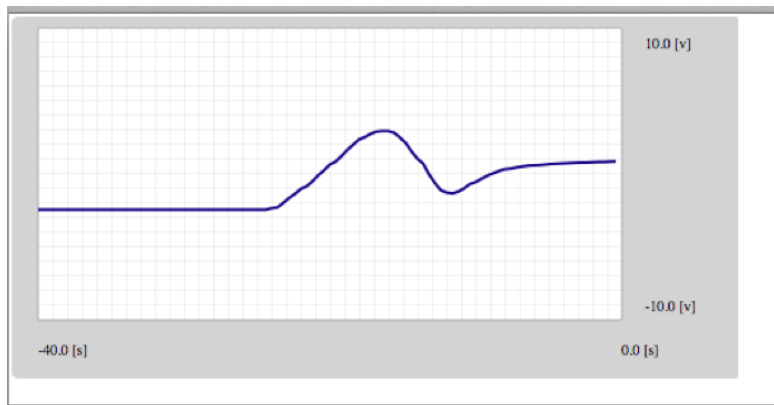


Figure 6: The advanced app, which locally stores the temperature and displays them as a curve.

measurements and display it as a curve (see Figure 6). There is absolutely no change made on the Smart Device service, the ‘advanced’ app uses the same metadata information and connects to the same WebSocket. The app design is left to the app developer. The examples in Figure 5 and 6 illustrate a possible scenario of the metadata use reflected in app design.

The sensors and actuators can be:

- *real*: represents a physical sensor on the Smart Device, for example a temperature sensor or an angular position measurement.
- *virtual*: represents a computed sensor, for example a speed measurement derived from a position measurement. Virtual sensors and actuators are also used to interface internal functionalities when required.
- *complex*: represents the aggregation of sensors/actuators, for example buttons on the front panel of an oscilloscope.

The data structure returned by a sensor or sent to an actuator may vary depending on the number of values and the measurement data structures (see Figure 7). The data structure contains three fields to enable flexibility. In the ‘valueNames’ field, the names of the sensor or actuator value is listed as returned by the sensor or actuator metadata services, `getSensorMetadata` and `getActuatorMetadata` (see Section 2.3.3). Then the actual data for each value is listed. The data as well as the ‘lastMeasured’ timestamps are listed at the

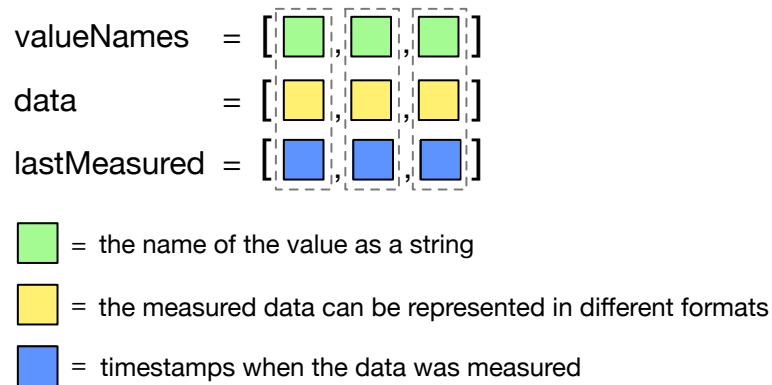


Figure 7: Sensor and actuator data structures.

same position as the value name. So all information of one value is at the same array index as indicated by the dashed lines in Figure 7. Finally, the 'lastMeasured' array contains the timestamps related to when the value name with the same index was measured. This timestamp array is optional and should not be included when sending data to set an actuator. The elements in the data array can be of different formats. It can be:

- *a single value*, for example temperature
- *an array of values* representing a set of single values over time, for example temperatures over the last minute
- *aggregated values* representing a sensor or actuator value that returns more than 1 value, for example a 3D accelerometer (which is not split in separate values).
- *an array of aggregated values* representing a set of aggregated values over time, for example 3D acceleration over the last minute. Here the data structure can be modelled in two different ways. In the first case, each value measured at one second is modelled as a value. In the second case, it is a single value that contains a data array with 60 elements (one for each second) and there is an additional 'lastMeasured' array containing the timestamps.
- *complex data structures* can be used when sensors and actuators require input and output that is not definable in primitive variables or arrays. For instance complex JSON objects or binary data might be required in some cases.

As a complex data structure, a video camera can be seen as a single value sensor that returns a compressed image, but it can also be seen as an array of values when considering each pixel of the image bitmap or it can be seen binary value with JPEG encoded data. The choice between the three representations is left to the lab owner.

Listing 2.13 shows an example request to the `getSensorData` service. Optionally, an access role from the concurrency role list (see Section 2.3.3) can be

added to express the access rights the client wants to have. If no `accessRole` is available, the Smart Device can decide the role. The Smart Device will decide whether these rights can be given and react accordingly.

Listing 2.13: Request example of `getSensorData` service.

```
{
  "authToken": "dskds909ds8a76as675sa54",
  "method": "getSensorData",
  "accessRole": "controller",
  "sensorId": "3D-pos",
  "updateFrequency": 20,
  "configuration": [
    {
      "parameter": "precision",
      "value": 2
    }
  ]
}
```

The `getSensorData` could return the response in Listing 2.14. It might be more efficient to handle data at the client in the form of arrays of values than to handle complex data structures interleaved with timestamps. Therefore, the Smart Device will place the measured data of its ‘valueNames’ in the ‘data’ array and the timestamps in another ‘lastMeasured’ array at the same index as the corresponding sensor value in the ‘valueNames’ array. For instance, sensor value ‘X’ has measurement 12.37 at time ‘2014-06-23T18:28:43.511Z’.

Listing 2.14: An example response of Listing 2.13 for a 3D position sensor with multidimensional data.

```
{
  "method": "getSensorData",
  "sensorId": "3D-pos",
  "accessRole": "controller",
  "responseData": {
    "valueNames": ["X", "Y", "Z"],
    "data": [12.37, 23.51, 43.18],
    "lastMeasured": [
      "2014-06-23T18:28:43.511Z",
      "2014-06-23T18:28:43.511Z",
      "2014-06-23T18:28:43.511Z"
    ]
  }
}
```

Listing 2.14 illustrates the response if the access role is ‘controller’. Imagine that there is a reason for the Smart Device not to provide access to this sensor by another role, e.g. the ‘observer’ role. In this case, the Smart Device returns the

Listing 2.15 response. Now an 'observerMode' field is returned that provides extra info with waiting information that can be used to display to the user how long he has to wait and how many people are in front of him (see Section 2.4.2 for details). The 'queueSize' and 'queuePosition' field enable to display the position in the queue and the 'estimatedTimeUntilControl' provides the waiting time in seconds until the user can take control of the lab.

Listing 2.15: An example response of Listing 2.13 for an 'observer' role.

```
{
  "method": "getSensorData",
  "sensorId": "3D-pos",
  "accessRole": "observer",
  "observerMode": {
    "queueSize": 4,
    "queuePosition": 3,
    "estimatedTimeUntilControl": 190
  }
}
```

The Smart Device may offer the possibility to configure the video sensor (see Listing 2.16), if there is a 'configuration' field with the necessary parameters present in the sensor metadata as for example described in Listing 2.10. This option can be very useful to adapt for example to the client screen by reducing the transmitted image size, thus reducing the amount of data sent to a smart-phone compared to the amount sent to a desktop computer. Similarly the image compression level might be controlled. The sensor metadata tells which settings are exposed to the client (see Listing 2.10 for an example).

Listing 2.16: Request example of the getSensorData service for a 'video' sensor with configuration.

```
{
  "authToken": "dskds909ds8a76as675sa54",
  "method": "getSensorData",
  "accessRole": "controller",
  "sensorId": "video",
  "updateFrequency": 25,
  "configuration": [
    {
      "parameter": "width",
      "value": 640
    },
    {
      "parameter": "height",
      "value": 480
    },
    {
      "parameter": "compression",
      "value": 92.3
    }
  ]
}
```

```

    },
    {
      "parameter": "colorFilter",
      "value": [60, 27, 229]
    }
  ]
}

```

The pace at which the data (e.g. video images) are sent can also be controlled. If for some reason, the user temporarily needs to throttle the video stream, the client application can ask the Smart Device to reduce the number of images per second sent via the 'updateFrequency' field, assuming the 'userModifiableFrequency' field in the 'accessMode' field is true. The sending may also be halted for some period of time by setting the 'updateFrequency' field to 0, and then setting the updateFrequency larger than 0 will resume the sending. It is up to the client application designer to decide if she wants to take advantage of these features.

Listing 2.17: An example request to stop sending data.

```

{
  "method": "getSensorData",
  "sensorId": "video",
  "accessRole": "controller",
  "updateFrequency": 0
}

```

2.3.5 Actuator Service – sendActuatorData

The actuator service is very similar to the sensor service (see Section 2.3.4), hence most of the fields are equivalent. The actuator may be simple, virtual or complex. Each value carries the needed information about its representation. The actuator could also be configured (for example the gain of the amplifier could be set using the configuration, similarly the firmware of an embedded controller could be updated after validation of course). The main difference with the sensor service is the fact that the sendActuatorData method allows the user to actually set the desired actuator value. As the following example shows:

Listing 2.18: An example response for the sendActuatorData service.

```

{
  "authToken": "dskds909ds8a76as675sa54",
  "method": "sendActuatorData",
  "accessRole": "controller",
  "actuatorId": "motor",
  "valueNames": ["left"],
  "data": [17.90]
}

```

The internal functionality of the Smart Device should first validate the value sent (see the Security and local control functionality in Section 2.3.2) prior to applying it to the actuator itself. The actuator may also be controlled by another client. While there is no concurrency issue for the sensor, the access to the actuator needs to be moderated. Various schemas can be implemented by the lab owner to internally manage the actuator access (see Section 2.4.2). In the following examples, we will assume one of the most common scenarios, a user can either control the lab or can observe what others are doing (respectively using the controller and observer role). Given that the user has a controller role, the actuator may acknowledge the value sent via the payload. The payload is optional and the format is not specified. As a good practice we recommend to return the data of the actuator in the same format as the request data format. The returned actuator data in the payload can be used to update the actuator value representation on the client application (see Section 2.4.2 for an example). Or it can also be useful to provide more information about the status of the actuator or lab that can be useful to share with the users. The client can assume that the actuator has fulfilled the request when no errors are returned.

Listing 2.19: An example response for the `sendActuatorData` service to a user with the controller role.

```
{
  "method": "sendActuatorData",
  "accessRole": "controller",
  "lastMeasured": "2011-07-14 19:43:37 +0100",
  "payload": {
    "actuatorId": "motor",
    "valueNames": ["left"],
    "data": [17.90]
  }
}
```

If the actuator is currently used by another client, a specific payload, 'observer-Mode', will return some information regarding the time the user has to wait prior to get access to the actuator, similar to the example in Section 2.3.4.

Listing 2.20: An example response for the `sendActuatorData` service to a user with the observer role.

```
{
  "method": "sendActuatorData",
  "accessRole": "observer",
  "observerMode": {
    "queueSize": 7,
    "queuePosition": 4,
    "estimatedTimeUntilControl": 736
  }
}
```

Furthermore, the client app of a user with the ‘interruptor’ role can abort the actuator control of another user. The way the conflict is resolved is defined by the lab owner and/or the client application. There could be constant user interruptions or this role could only be granted to a few users.

Listing 2.21: Interrupting an on-going session of another user

```
{
  "authToken": "dskds909ds8a76as675sa54",
  "method": "sendActuatorData",
  "actuatorId": "3D-pos",
  "accessRole": "interrupt",
  "valueNames": ["X", "Y", "Z"],
  "data": [12.34, 48.39, 83.92]
}
```

2.3.6 User Activity Logging Service – getLoggingInfo

The user activity logging service has been discussed in the metadata service, where it was used as an example of adding an optional service to the specification (see Section 2.3.3). The user activity logging service returns logged user actions or lab info in the ActivityStream JSON format. In D5.1, we decided on this format for the exchange of user interaction data. In this section we will provide examples how one can access the service and its responses.

The service can be called with a ‘SimpleRequest’ data model, which is just a JSON object with a ‘method’ field and an optional ‘authToken’ field to authenticate the user (which is used here since this can be privacy sensitive data):

Listing 2.22: A request example to the getLoggingInfo service

```
{
  "authToken": "dskds909ds8a76as675sa54",
  "method": "getLoggingInfo"
}
```

The service then returns a response similar to the following sample snippet:

Listing 2.23: A response example to the getLoggingInfo service

```
{
  "method": "getLoggingInfo",
  "logs": [
    {
      "verb": "access",
      "published": "2011-02-10T15:04:55Z",
      "language": "en",
      "actor": {
        "objectType": "person",
        "id": "urn:utwente:person:anjo:anjewierden",
        "displayName": "Anjo Anjewierden",

```

```

        "url":
          "http://www.utwente.nl/gw/ist/medewerkers/wetenschappelijke_staf/
          anjo_anjewierden/",
        "image": {
          "url":
            "http://www.utwente.nl/gw/ist/medewerkers/wetenschappelijke_staf/
            anjo_anjewierden/anjo_anjewierden-1.jpg",
          "mediaType": "image/jpeg",
          "width": 133,
          "height": 177
        }
      },
      "object" : {
        "objectType": "sensor",
        "id": "urn:redlab:epfl:ch/3D-pos"
        "url": "http://redlab.epfl.ch/smartdevice/sensors/3D-pos",
        "displayName": "3D position"
      },
      "target" : {
        "objectType": "lab",
        "id": "urn:redlab:epfl:ch/smartdevice",
        "displayName": "RED Lab",
        "url": "http://redlab.epfl.ch/smartdevice/"
      }
    }
  ]
}

```

Again, the method name is returned and a list of ActivityStream objects. The ActivityStream objects will be pushed to the client as they become available.

2.3.7 Client Application Service – getClient

This required service provides links to the client applications that are provided by the lab owner to operate the lab. The service is called 'getClient'. The implementation technology of the clients is not strongly specified, but Go-Lab advocates OpenSocial gadgets (Marum, n.d.), since they effortlessly run on the Go-Lab ILS platform (see D5.1 and D5.2).

A list of client applications can be requested using the following 'PublicRequest' call for the method 'getClient'.

Listing 2.24: A request example to the getClient service

```

{
  "method": "getClient"
}

```

Upon which a list like in the following example snippet can be returned:

Listing 2.25: A response example to the getClients service

```

{
  "method": "getClients",
  "clients": [
    {
      "type": "OpenSocial gadget",
      "url": "http://superlab.epfl.ch/client/dataviewer.xml"
    },
    {
      "type": "OpenSocial gadget",
      "url": "http://superlab.epfl.ch/client/video.xml"
    },
    {
      "type": "OpenSocial gadget",
      "url":
        "http://superlab.epfl.ch/client/experiment-operator.xml"
    }
  ]
}

```

Each element in the ‘clients’ list contains a ‘type’ and a ‘url’. The client type declares which type of application it is. In the current version of the Smart Device specification, we have identified the following types: ‘OpenSocial Gadget’, ‘W3C widget’, ‘Web page’, ‘Java WebStart’ and ‘Desktop application’. This can be extended in the future. Within Go-Lab, we advocate the use of OpenSocial Gadgets to ensure interoperability (see D5.2).

2.3.8 Models service – getModels

This service is optional and can provide several models of the physical lab (i.e. the instrumentation) and the theory behind the experiment. For instance, a 3D graphical model of the lab instrumentation can be provided. With this graphical model, a client app can generate a GUI that provides a 3D scale object that student can manipulate to understand the whole setup. Together with a theoretical or mathematical model of the experiment, a client app could be build that provides a simulation of the lab. This is useful to provide an interactive version of a remote lab, which could be used by students when the lab is already in use (i.e. to provide a better observer mode).

Due to the wide range of existing formats to express graphical and theoretical models (e.g. VRML²⁰, X3D²¹ & MathML²²), we do not limit the Smart Device specification to one specific option. The choice of the model language is up to the lab owner. However the lab owner should define in the service definition which format is returned by the getModels service. This can be done through the ‘produces’ field and a media type. We do not provide an example definition

²⁰Virtual Reality Modeling Language (VRML), <http://en.wikipedia.org/wiki/VRML>

²¹X3D, <http://en.wikipedia.org/wiki/X3D>

²²MathML, <http://www.w3.org/Math/>

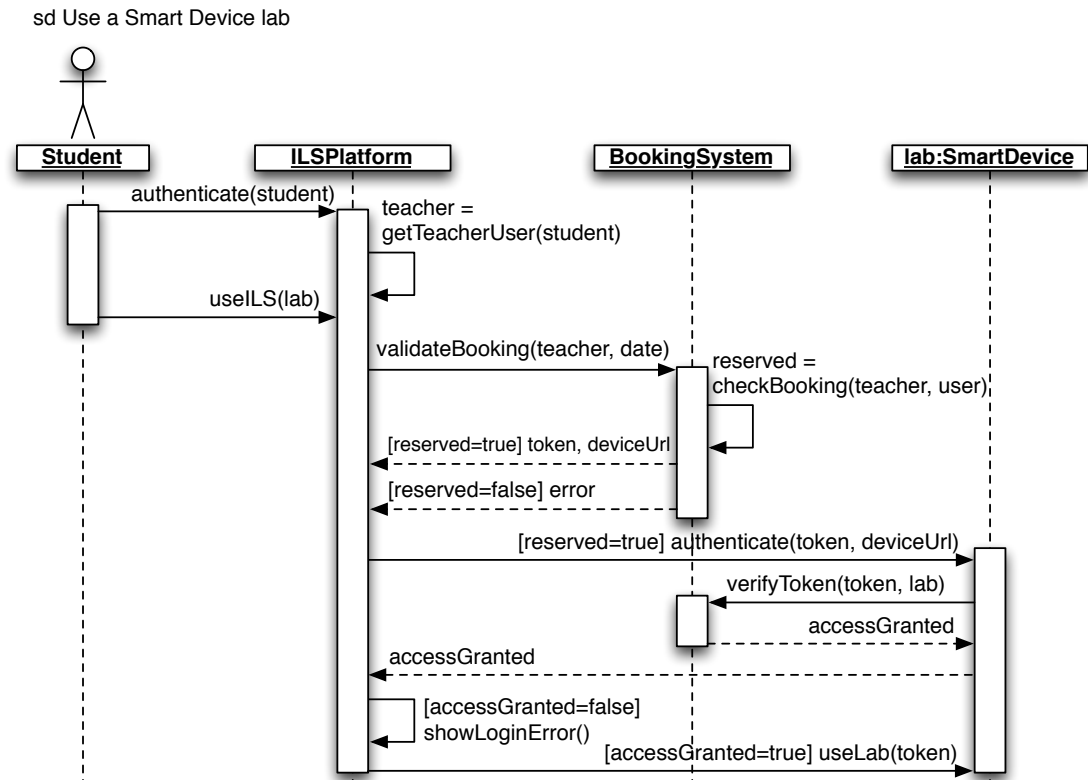


Figure 8: UML sequence diagram of the interaction between the Go-Lab booking system and a Smart Device.

for the `getModels` service, but we refer to Section 2.3.3 for more information on how to define such a service for the Smart Device.

2.4 Smart Device Interactions

In this section, we elaborate on several interaction scenario's between client apps, external services and the Smart Device.

2.4.1 Authentication and Booking

A Smart Device can require booking. To support this, Go-Lab provides its own simple booking system, as specified in D4.2. In this section, we will briefly summarise the D4.2 specification related to the booking system and the Smart Device. The booking use case itself is not recapitulated here, since it solely involves the booking system.

Since we decided in D4.2 that the Go-Lab booking system will manage the complete booking calendar of a Smart Device, the Smart Device only needs to validate the booking token with the Go-Lab booking system. Figure 8 illustrates how a client app with a reservation can use a Smart Device. The student can use the Smart Device with the teacher's reservation for a specific date. The booking has to be validated with the Booking System. If the booking is available, a token will be returned. This is the authentication token that can be used to access the Smart Device services and this is the token that was

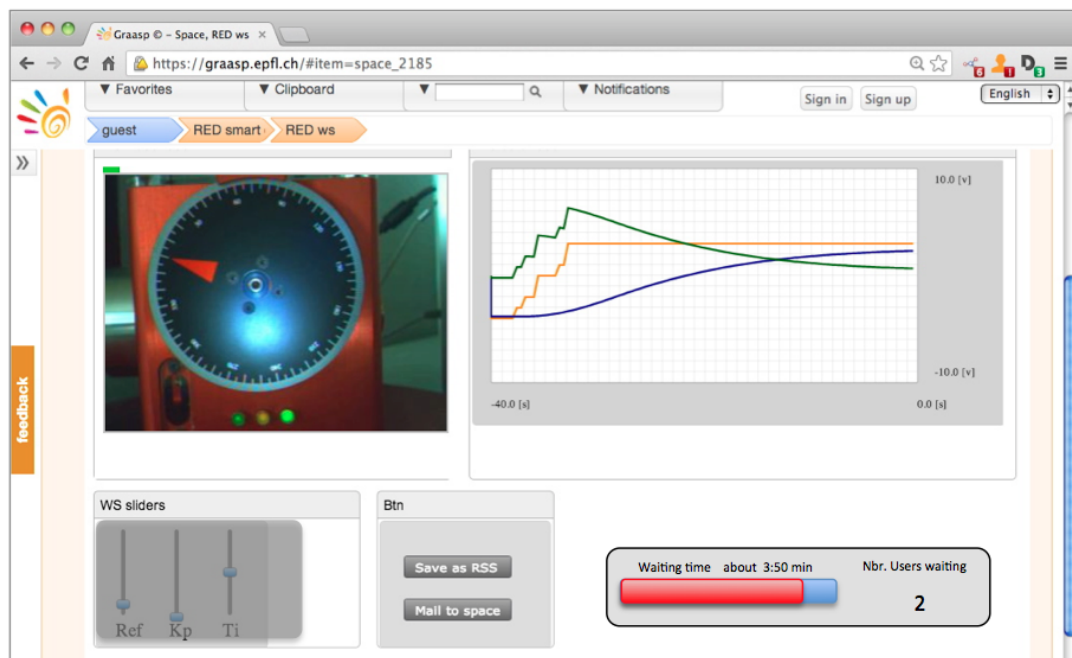


Figure 9: UI mockup of a waiting queue visualisation.

modelled as the Swagger API key in Section 2.3.3. The Smart Device then validates this token with the Booking System. This is the only functionality that a Smart Device needs to implement to enable booking, as discussed in Section 2.3.2. Essentially this means connecting securely to the Booking System to validate the authentication token. When the token is valid, the user can access the Smart Device. Note, that in this updated Smart Device specification, the `authenticateToken(token, deviceUrl)` and `useLab(token)` calls are actually integrated in the calls to the different services, which can require an authentication token, e.g. for the user activity logging service (see Section 2.3.6) the request itself contains the authentication token:

Listing 2.26: An example request to the `getLoggingInfo` service using an authentication token

```
{
  "authToken": "dskds909ds8a76as675sa54;",
  "method": "getLoggingInfo"
}
```

2.4.2 Interaction Modes

One of the challenges when dealing with remote labs is to handle concurrent connections (see Section 2.3.3). While this issue can be resolved through booking, additional measures can be taken to enhance the user experience. In some scenarios these features might also be desirable from a pedagogical point of view.

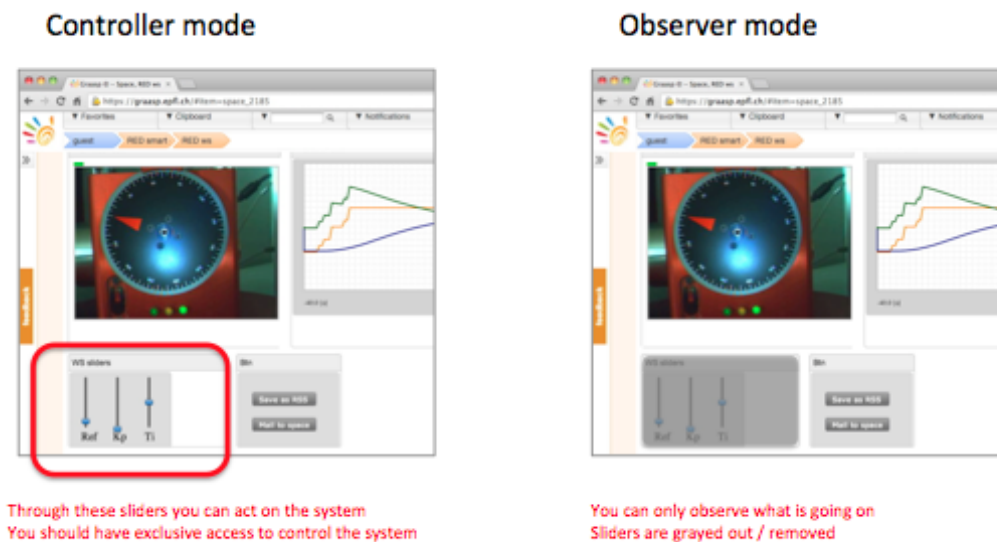


Figure 10: UI mockups of potential client apps used with an observer and controller role.

The default mode of interaction is one-to-one where one client is connected to the Smart Device. A given time slot could have been negotiated via the booking mechanism. In addition to the booking mechanism, or when no booking mechanism is present, the Smart Device can implement a queue with a priority or a FIFO access policy. In the case of a FIFO queue, the client should be informed about the number of users ahead and the estimated waiting time (see Figure 9). It is up to the UI designer to decide how to present this information, but the Smart Device will provide the information. The client application developer may decide to implement support for the different access roles of the Smart Device (as discussed in Section 2.3.3). All the needed information is provided by the Smart Device.

Figure 9 illustrates that the client app is waiting another 3:50 min prior to get access to the experiment. If an observer role is available (see Section 2.3.3), s/he can observe actions made by others in the meantime.

As described in Section 2.3.3, different roles can be defined for a Smart Device that supports concurrency. For instance, a controller role that allows the client to act on the remote equipment which translates basically to setting an actuator value. In the observer role the client could visualize the information returned by the sensors. In this mode the actuator values may also be updated to reflect the modifications made by the user that currently controls the Smart Device, but the user with the observer role cannot modify them.

The developer of the client application can take advantage of the information returned by the Smart Device to for example show/hide (or gray-out) the sliders that permits to change the actuator values according to the current state, see Figure 10.

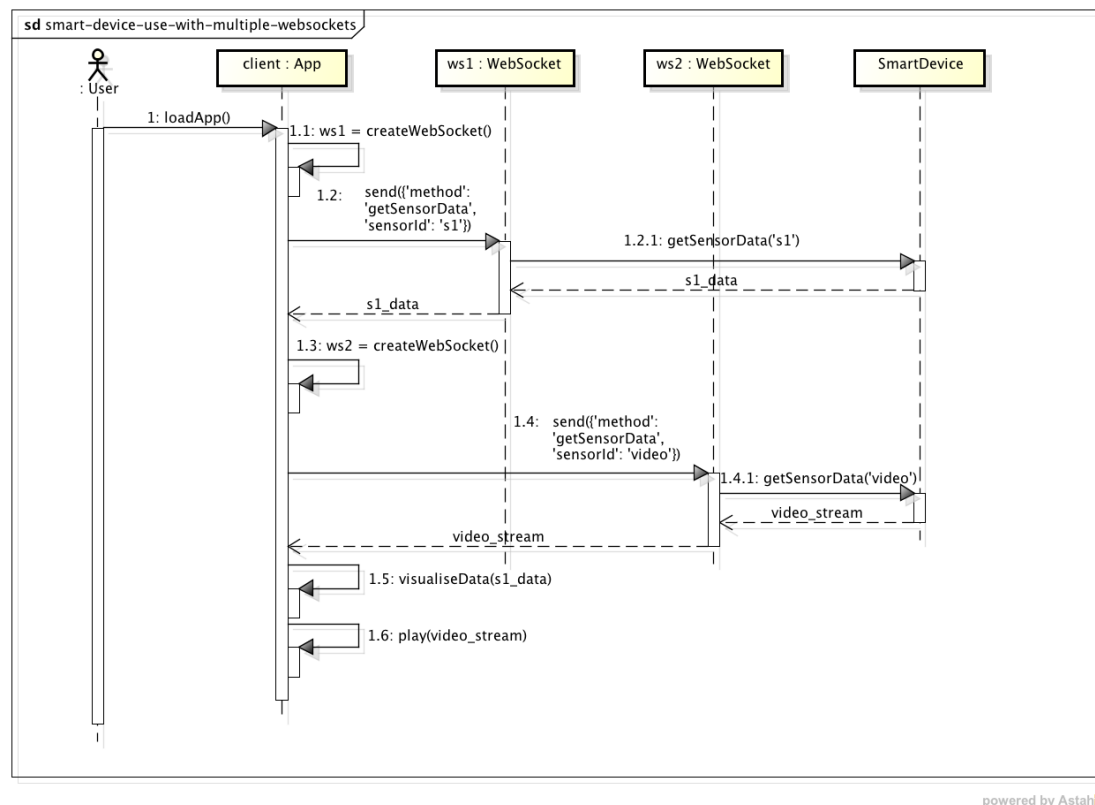


Figure 11: UML Sequence diagram on the use of multiple WebSockets for connecting to multiple Smart Device services.

2.4.3 WebSocket Channeling

There is some concern that opening many (>10) WebSockets²³ can impact browser performance. This could be the case if a Smart Device has many services, or if one would use a separate WebSocket for each sensor or actuator. Figure 11 illustrates this scenario with two Web Sockets (adding more is identical). The app in Figure 11 calls two Smart Device services and for each a separate Web Socket is created, once the data is returned, it is visualised and presented to the user. This is a typical example of a lab client opening multiple WebSockets.

As mentioned opening many WebSockets may cause browser performance issues, however in the use case of Figure 11 it actually makes sense. The first WebSocket retrieves JSON sensor data over a textual WebSocket. The second WebSocket needs to retrieve a video stream, which is a binary stream of data and hence uses a binary WebSocket to increase performance. Common

²³Microsoft Internet Explorer 10 sets the maximum open WebSockets to 6. After editing the Windows Registry this can be increased to 128, see http://msdn.microsoft.com/en-us/library/ie/ee330736%28v=vs.85%29.aspx#websocket_maxconn. For most browsers (except Microsoft Internet Explorer) this issue should be resolved, e.g. Firefox has set the maximum number of open WebSockets to 200 in recent versions, see <https://developer.mozilla.org/pl/docs/WebSockets>. However, we need to take this technical limitation still into account due to default-configured Internet Explorer clients.

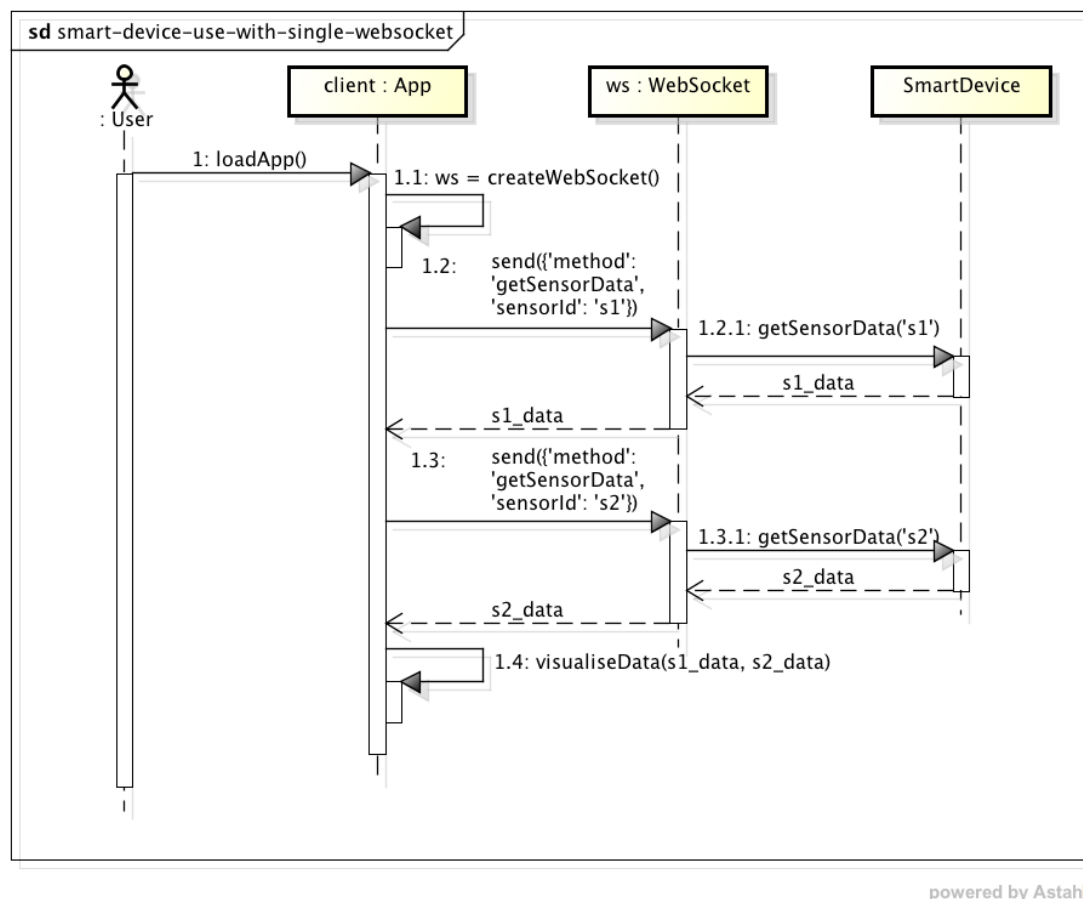


Figure 12: UML Sequence diagram on the use of a single WebSocket for connecting to multiple Smart Device services.

sense should be used to channel different service calls into one WebSocket if performance is affected. The fact that they are textual or binary WebSockets and whether it is recommended to create a new WebSocket can be expressed in the Smart Device metadata of the sensors (see Section 2.3.3).

Figure 12 illustrates an app that creates just one WebSocket to access two different Smart Device sensors. Of course, this WebSocket could be reused by other services that require a textual WebSocket.

2.4.4 Lab Instruments as Complex Sensors

A complex sensor/actuator represents a collection of related sensors/actuators. The idea is to aggregate sensor/actuator when it make sense. This aggregation is left to the lab owner. Typically such a collection represents a set of sensor/actuator that form an instrument (during the Madrid meeting with external experts, see Section 1.2, such aggregation was requested by the experts). An instrument is for example an oscilloscope, in this case all the knobs of the oscilloscope's front panel will be aggregated as one actuator with different values (similarly to the 3D accelerometer sensor). This actuator will be accessible through one WebSocket instead of one WebSocket for each knob (assuming

that WebSocket are not channeled).

2.5 The Revised Smart Device Specification Compared to the M12 Version

This section briefly highlights the differences and additions of this revised specification compared to the M12 D4.1 specification.

- *Metadata*: More detailed and well-defined metadata descriptions using and extending the Swagger specification.
- *Protocol*: More detailed and well-defined Smart Device interfaces and data exchange formats for all required services.
- *Concurrency*: More detailed metadata on concurrency mechanisms supported by the Smart Device.
- *Authentication*: The authentication mechanisms are well-defined and based on the specifications of D4.2.
- *Sensor types*: Different types of sensors have been defined which make the Smart Device specification more flexible and extendable.
- *Configuration*: Sensors and actuators can be configured per client and per call. This can be very useful for mobile devices.
- *WebSocket channeling*: Mechanisms to improve WebSocket use have been defined.
- *Terminology*: Some terminology has changed since M12. This list provides an overview, but is not exhaustive:
 - *service*: a software component to exchange data between the Smart Device and external clients or services.
 - *functionality*: a software feature that is used internally in the Smart Device
 - *sensor*: provides output information to the client
 - *actuator*: allows input from the client to operate the Smart Device
 - *instrument*: bundles a given set of sensors or actuators and can be exposed through the smart device as one complex sensor or actuator.

3 Cloud Services

As Cloud Services (see Go-Lab DoW and Task 4.2), we understand a set of services designed to enable access to the Go-Lab infrastructure to lab owners of legacy lab systems. In the scope of this deliverable, legacy lab systems are all online labs not designed according to the Smart Device specification described in Section 2.

3.1 Introduction

In order to plug an existing legacy lab system into the Go-Lab infrastructure, we have identified the following possibilities (see Section 1.3):

- Redesign of the online lab system according to the Smart Device specification
- Integration of a lab client app via an iFrame in the ILS Platform
- Integration via the Smart Gateway

The first approach was described in detail in the previous section of this document. It is the recommended approach if lab owners are developing a new system from scratch (or are willing to redesign it) and want to plug it into Go-Lab since it would ensure full compatibility. The second approach is simple to implement since it basically consists of embedding the existing client app in the ILS Platform via an iFrame (for details about the ILS Platform: see deliverable D5.2). In this approach there is absolutely no communication between the lab and the ILS Platform. The third approach is the integration of the legacy lab system via the Smart Gateway. It ensures a higher degree of compatibility, compared to the second approach (iFrame), since the Smart Gateway will already provide Go-Lab services. For example, by relying on the Smart Gateway, the lab owner will not need to implement the integration with the Go-Lab portal for the Go-Lab booking system, for metadata services or in general for any service defined in Go-Lab, since the Smart Gateway will implement them. This approach is the recommended option if the lab owner is not willing or has no resources to redesign the legacy lab system according to the Smart Device specification.

The Smart Gateway is the core of the Cloud Services. It can be added as a proxy between the legacy lab system and the Go-Lab infrastructure to ensure a higher level of compatibility and compliance with the Smart Device specification. The level of compatibility offered by this approach will be described in detail in the following sections.

Due to the high heterogeneity of online lab systems it was decided that the Smart Gateway should be as flexible and generic as possible to support a variety of these systems. Following these requirements, we opted for a plug-in architecture. Each plug-in should implement the communication with the legacy lab platform and its implementation is dependent on the desired level of integration (as defined in Figure 13). Thus, the Smart Gateway relies on plug-ins to bridge the communication with the legacy lab system. Plug-ins for specific

legacy lab systems shall be developed by the lab owners with support from the Go-Lab team. For online laboratories managed by a Remote Lab Management System (RLMS) a single plug-in can be developed: if a plug-in for an RLMS is provided, all the laboratories managed by this RLMS will automatically be available under the Go-Lab infrastructure. Plug-ins for well-known RLMS such as WebLab-Deusto or iLab Shared Architecture are provided by Go-Lab (see D4.3).

This chapter is structured as follows. First the functional and non-functional requirements of the Smart Gateway are described, followed by a comparison with existing projects and well known Remote Lab Management Systems (RLMS). The following sections describe in detail the architecture of the Smart Gateway and the different options to integrate legacy lab systems into to Go-Lab ILS platform using the plug-in architecture of the Smart Gateway along with the advantages and disadvantages of each approach. Next, the different possibilities to deploy the Smart Gateway are described. The last section describes an optional component of the the Smart Gateway, called Protocol Translator.

3.2 Requirements for the Smart Gateway

Legacy lab platforms are very heterogeneous since they were developed to meet different requirements. For example, some platforms require the user to authenticate for the purpose of tracking user actions, while others do not care about the user's identity. Some legacy platforms support lab session booking while others use a queuing mechanism or even a merged schema combining booking and queuing. The Smart Gateway should make no assumptions about the legacy lab system in terms of their own functional requirements. In this section we will present the functional and non-functional requirements of the Smart Gateway.

3.2.1 Functional Requirements for the Smart Gateway

Based on the previous considerations we identified the following functional requirements of the Smart Gateway:

- *R1: Publishing user interfaces in an OpenSocial container.* In order to ensure integration with the ILS Platform (see D5.2) the legacy client app should be packaged in an OpenSocial container so that it can easily be embedded (like any other Go-Lab app) into an ILS. This process should be completely transparent for teachers: in the sense that they do not need to know if a lab app is being served by the Smart Gateway or by a native Go-Lab Smart Device.
- *R2: Support existing legacy laboratory authentication mechanisms, if present.* If the legacy lab system requires users to authenticate prior to launching a lab client app, the Smart Gateway should be able to bridge this authentication and provide valid credentials to the legacy lab system to authorise the launch of a specific lab client.
- *R3: Connect to existing legacy laboratories scheduling mechanisms, if*

present. If the legacy lab requires some scheduling mechanism (e.g. queueing or calendar-based booking), it should be managed by the Smart Gateway, so other Go-Lab components are not aware of it or they integrate these mechanisms.

- *R4: Support external Go-Lab add-on services, such as the Go-Lab booking system. This support is optional for each remote laboratory.* The Smart Gateway should provide a bridge between the Go-Lab booking system and the legacy booking system, provided that the lab requires this functionality. This is different from R3 since the legacy system might for instance provide a queue for each individual session which is covered by R3, and for R4 it optionally could be managed at group level with a calendar guaranteeing exclusivity for that group.
- *R5: Retrieve metadata from the legacy lab (if available) or provide forms to author metadata content.* Part of mimicking the behaviour of a Smart Device consists in providing the metadata services described by the Smart Device specification.
- *R6: Provide basic management tools for the Smart Gateway administrator.* The lab owner should be able to add, remove, and list supported laboratories and provide public links to be added to the Go-Lab portal. The lab owner should also be able to take individual decisions on each supported laboratory, such as when it is available for booking at Go-Lab level (see D4.2).

3.2.2 Non-functional Requirements for the Smart Gateway

Additionally we were able to identify the following non-functional requirements:

- *R1: Extensible architecture to support a wide variety of remote laboratories.* A measure of success of the Smart Gateway is supporting a wide range of existing remote laboratories, remote laboratory management systems and virtual laboratories.
- *R2: Provide technical incentives to laboratory owners to be willing to adopt the Smart Gateway.* By providing further technical incentives to laboratory owners, they might want to support the integration in the Smart Gateway for additional reasons to the ones provided by Go-Lab directly. The incentives mentioned are described in detail in D4.3.
- *R3: Documentation.* Provide documentation for lab owners on how to integrate their lab system in the Smart Gateway.
- *R4: Administration.* Flexible deployment and easy maintainability of the Smart Gateway. We make no assumptions on where the Smart Gateway should be deployed and we do not foresee a centralized deployment.

3.3 Review of Legacy Lab Platforms

Remote Lab Management Systems (RLMS) are software frameworks that aggregate common functionalities to manage online laboratories. For example,

these functionalities can include user and lab session management, storage of experimental data, scheduling of lab session, user activity tracking and support for resources federation. The need for RLMSs arose when the number of online lab systems began to increase and scalability became an important issue. Managing several online lab installations around a single framework allowed for a reuse of common functionalities and separation of tasks, namely the actual laboratory development (e.g. interface with the lab equipment and the experiment logic) and the management part (authentication, authorization, lab booking, etc). Thanks to the functionality provided by the RLMS, online lab developers could concentrate on the actual experiment logic and pedagogical aspects of the client application while the other management functionality was delegated to the RLMS. Any new features included in the RLMS would be automatically available to all labs managed by this system. Today there are a few of these systems available.

WebLab-Deusto: is an open source remote laboratory management system (Orduña et al., 2011), originally developed in the University of Deusto and used to develop laboratories in other institutions in Slovakia¹, Brazil², France and Colombia, as well as a number of schools in Europe.

The main scheduling mechanism is a priority queue. It supports a federation model (Orduña, 2013) where one institution can share their laboratories with other institutions, without exchanging users or credentials. This federation model is both transitive (if university A shares lab X with university B, then university B can share lab X with C) and supports load balancing among copies of the same laboratory in the same institution and cross-institution (federated load balance). At the time of this writing, it does not support any booking mechanism. It provides administration panels to support the inclusion of these federated laboratories.

Regarding interoperability, bidirectional bridges with the iLab Shared Architecture (Orduña, Bailey, DeLong, López-de Ipiña, & García-Zubia, 2014) and UNR FCEIA (Orduña et al., 2013) have been built. This way, WebLab-Deusto can manage local users and permissions on these federated laboratories and vice versa.

iLab Shared Architecture: The iLab project started at MIT in 1998, with the goal to develop a distributed software toolkit and middleware service infrastructure to support online laboratories, and promote sharing among schools and universities on a worldwide scale (Harward et al., 2008; Hardison & Garbi Zutin, 2011; Sancristobal et al., 2010). Therefore, MIT implemented the iLab Shared Architecture (ISA), focusing on fast platform-independent laboratory development, scalable access for students, and efficient management for lab providers, while preserving the autonomy of the faculty actually teaching the students. It defines two types of laboratories: Batched and Interactive. Batched labs are those where experiments are completely specified prior to submission and execution

¹<http://weblab.chtf.stuba.sk>

²<http://weblabduino.pucsp.br/weblab/>

without human intervention, where the student specifies the entire course of the experiment before the experiment begins. Interactive labs, on the other hand require taking control of the laboratory, therefore, an interactive experiment must commit the laboratory hardware to a single user for the duration of the session – typically 20 minutes to an hour – and this may require scheduling.

Labshare Sahara: Led by the University of Technology, Sydney, Labshare Sahara is a joint initiative of the Australian Technology Network: Curtin University of Technology, Queensland University of Technology, RMIT University, University of South Australia, and the University of Technology, Sydney (<http://www.labshare.edu.au/>). This project aims at creating a national network of shared remotely accessible laboratories. To do this, they have developed a framework for setting up a heterogeneous remote laboratory of physical apparatus containing many labs of many types called SAHARA (Lindsay, Stumpers, & others, 2014; D. B. Lowe, Berry, Murray, & Lindsay, 2009; D. Lowe, Murray, Lindsay, & Liu, 2009).

Library of Labs – LiLa is an initiative of eight universities and three enterprises for the mutual exchange of and access to virtual and remote laboratories. (Bellido, Villagra, & Mateos, 2010; Tetour, Boehringer, & Richter, 2011) To accomplish this task, the SCORM (*SCORM - Home*, n.d.) standard has been modified to communicate with remote online labs. LiLa also builds a portal through which the access to virtual labs and remote experiments is granted. It includes services like a scheduling system, connection to library resources, a tutoring system, 3D-environment for online collaboration. Although the LiLa project bundles labs in SCORM packages, proper interoperability among the different labs is not always possible since SCORM has not been designed for interactive labs and there is lack of support of the latest versions of SCORM (see deliverable D5.2).

ViSHub: As part of the Global Excursion European project, the Virtual Science Hub (ViSHub) portal has been developed. All the contents are publicly available, becoming a portal where different types of virtual and remote laboratories (with no authentication, authorization or scheduling mechanism) are available altogether with other resources (such as documentation, links, etc.). Advanced APIs are provided, so it is possible to search for custom resources and embed them in other frameworks. However, these APIs are not intended to easily integrate virtual and remote laboratories that are not already open and no federation protocol is used with those laboratories.

3.4 Requirements for the Smart Gateway and Comparison with Other Systems

As shown in Table 1, the design goals of the Smart Gateway are completely different from those of the existing remote lab management systems or legacy labs available. It is designed to be a bridge between a legacy lab system and the Go-Lab ILS Platform.

Table 1: Comparison between existing platforms and the Smart Gateway functional requirements. Legend: requirement supported (+), partially supported (~) & not supported (-)

Smart Gateway Functional Requirements	WebLab Deusto	ISA	LabShare Sahara	LiLa	ViSHub
R1	-	-	-	-	-
R2	~ ^a	~ ^a	~ ^a	-	-
R3	~ ^a	~ ^a	~ ^a	-	-
R4	~ ^a	~ ^a	~ ^a	+ ^b	-
R5	-	-	-	+	+
R6	+	~	~	+	+

^aThrough ad hoc interoperability bridges (WebLab - ISA, ISA - WebLab, Labshare - ISA), not with other laboratories. These bridges support authentication and scheduling, and they do not support the Go-Lab add-on services but they support their own services including authorization or scheduling.

^bIt does not support the Go-Lab add-on services, but it supports the LiLa services, which included a booking mechanism.

3.5 Specification and Architecture of the Smart Gateway

The Smart Gateway aims to support the integration of existing laboratories, as if they were fulfilling the Smart Device specifications described in Section 2. Ideally, even legacy lab owners will be willing to develop their remote laboratories using the Smart Device specification to benefit from the features provided by it, such as reusability of client code, simplicity or interaction with other services. However, this ideal situation will not always be the case, and the more complex the legacy labs are, the more likely lab owners will be willing to find a straightforward solution that enables them to integrate their laboratory in Go-Lab even if they drop support of certain features. A tradeoff between those features and the implementation efforts will always be present.

For this reason, the Smart Gateway encompasses different integration levels for existing legacy laboratories. These integration levels are illustrated in Figure 13.

As shown in the figure, the levels are:

1. *iFrame in Smart Gateway*. If the laboratory does not require any authentication mechanism, it can be directly integrated as an enhanced iframe: it technically would be an iframe but when added, the administrator can fill basic metadata about the laboratory, and it will be provided as an OpenSocial application. Interoperability with the rest of the Go-Lab infrastructure will be provided by the Smart Gateway. For example, the Smart Gateway will provide metadata services for other Go-Lab components. This metadata can be added to the Smart Gateway for that particular lab client application, so the Go-Lab portal consumes it. The Smart Gateway will be integrated with the booking system provided by the Go-Lab Portal, so tech-

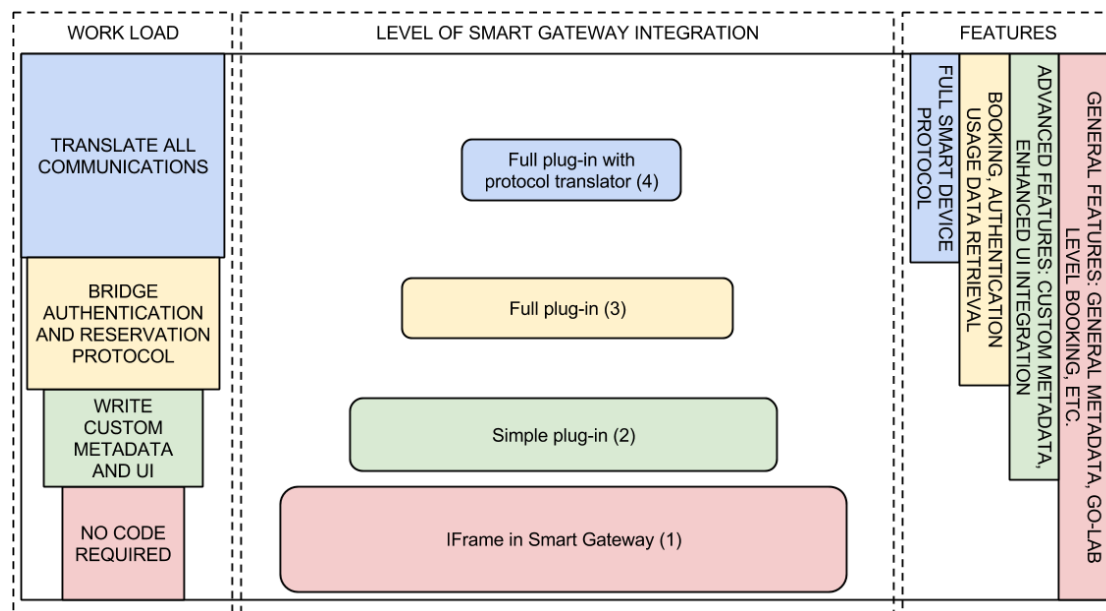


Figure 13: Levels of Smart Gateway integration

nically any web application managed through the Smart Gateway could optionally be booked, while it would still be accessible from the Internet.

2. *A simple version of the plug-in.* Can be implemented by simply submitting a custom request to the laboratory to be loaded. For example, a remote laboratory that requires authentication could easily be integrated by always logging in with a guest / demo account within the Smart Gateway. The plug-in might have all the credentials hardcoded. Another laboratory which does not provide any authentication mechanism will always redirect the user to the final laboratory, using a given URL.
3. *A full version of the plug-in,* where all the required reservation features by the remote laboratory are matched. For example, a remote laboratory may require that users are identified with a unique identifier, or support a custom protocol that finally generates a URL that is directly loaded by the client.
4. *A full version of the plug-in with a protocol translator,* where in addition to the full version of the plug-in, it generates a metadata URL that matches the Smart Device specification, so different clients can connect to the provided services. This requires a big effort, since all the communications to the client must be rewritten or an alternative communication channel must be provided to access the sensors and actuators by the particular laboratory.

In this way, the lab owner can decide on the trade-off of efforts and provided features. If it is possible, the lab owner can provide a very simple plug-in that only creates a redirection to the final laboratory, or even it can use the existing iframe plug-in to provide certain basic metadata. If this is not possible (since the

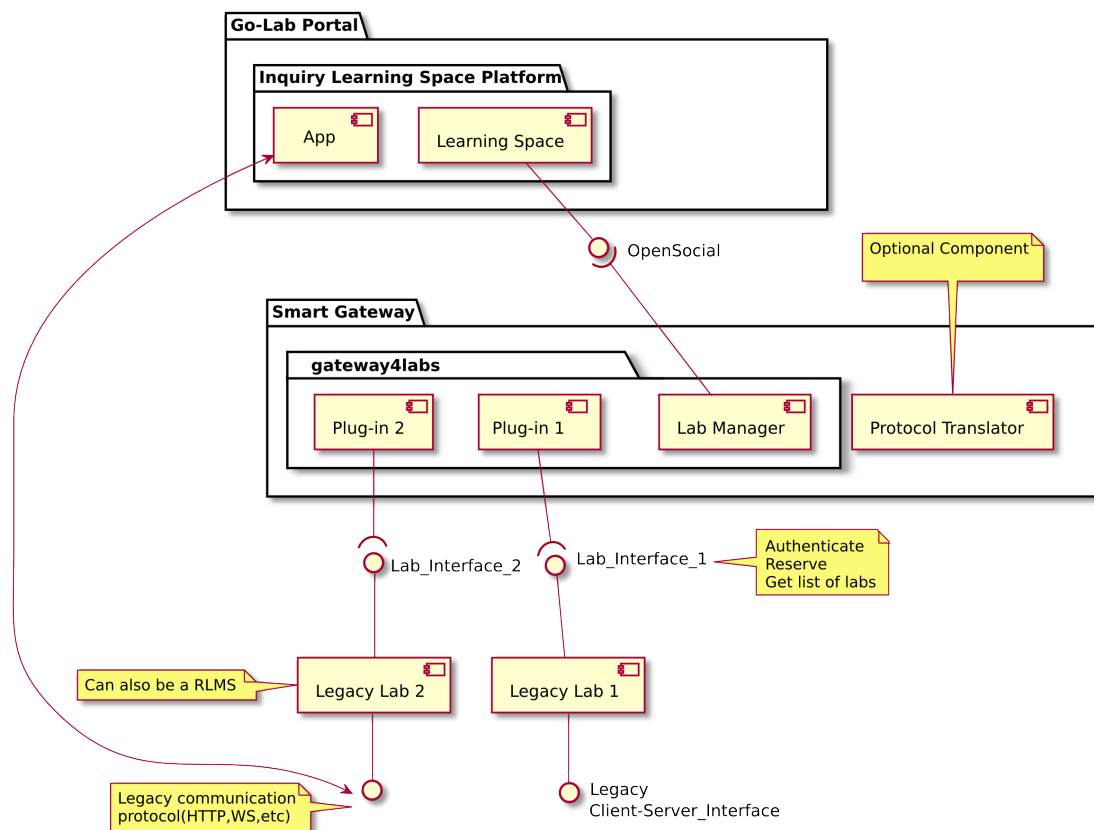


Figure 14: Cloud services architecture

remote laboratory requires some authentication, for instance), then a full version of the plug-in is required. These plug-ins only manage the reservation process or the metadata. Finally, if the lab owner wants to embrace the full specification, the lab owner must implement a protocol translator, which is essentially a service that takes all the requests in the format defined by the Smart Device and converts them to the original format used by the remote laboratory.

3.5.1 Architecture

The Smart Gateway is composed of two main components. The first one is gateway4labs (G4L), the second one is the protocol translator. Gateway4labs manages the reservation and integration process of the remote or virtual laboratory into the ILS platform, providing some of the services (such as metadata or exporting lab client apps as OpenSocial gadgets). The protocol translator is an independent and optional component which translates all the communications of the remote laboratory converting them to the specifications of the Smart Device.

Therefore, to integrate an existing remote laboratory, the lab owner must either adopt the Smart Device paradigm (previous sections in this deliverable, which requires redeveloping most of the communications and rely on third party gadgets or implement them if they do not exist or they are not suitable for the laboratory) or develop a plug-in in gateway4labs (which only acts as a bridge that

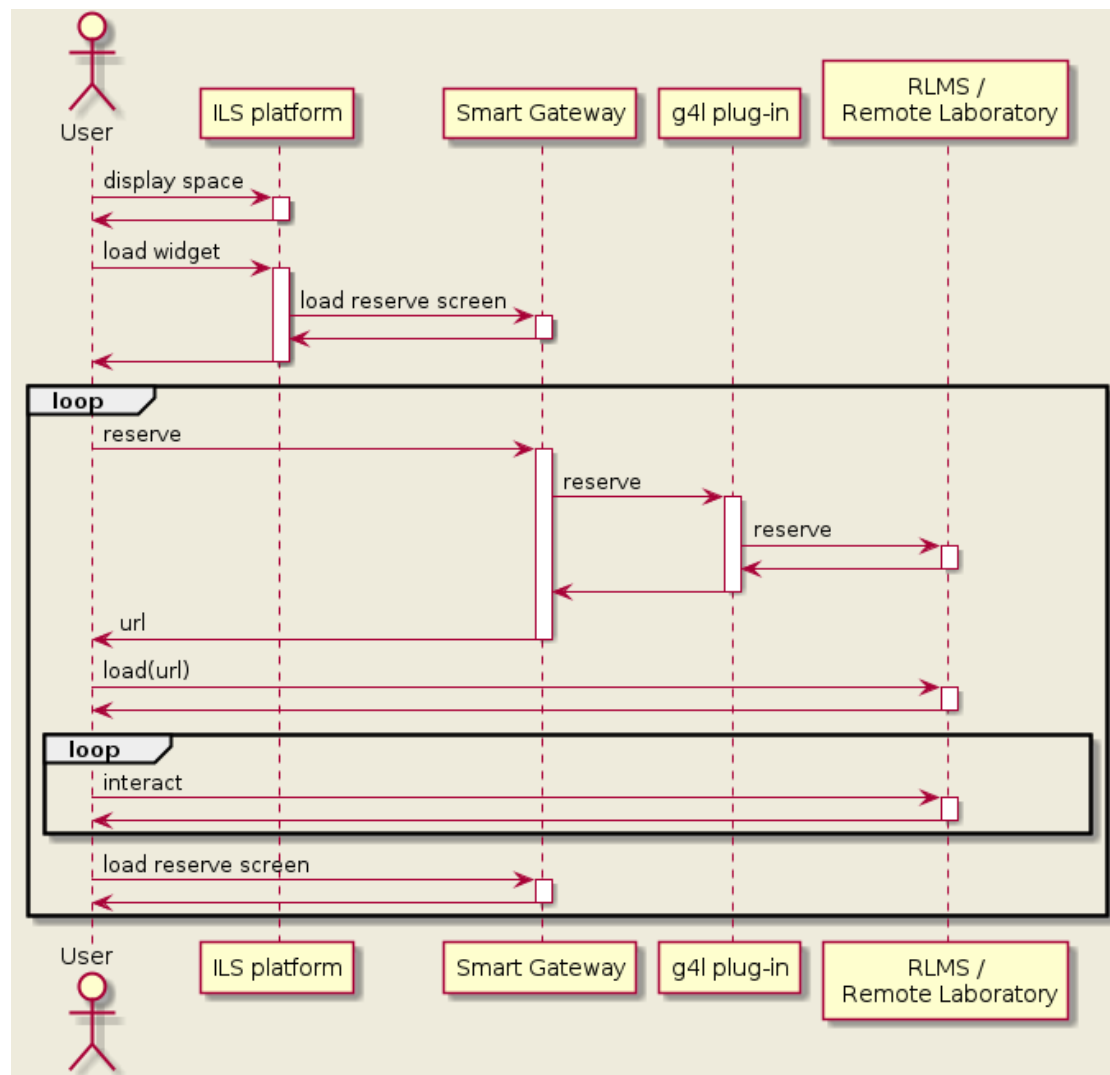


Figure 15: UML sequence diagram of the reservation process via the Smart Gateway

redirects the user to the final laboratory, so a smaller effort is required). If the latter is selected, the lab owner can optionally implement the protocol translator to bridge also all the communications.

As shown in Figure 14, gateway4labs consists of a middle component (called LabManager) that supports OpenSocial (used in the ILS platform, see D5.2) and a plug-in system. Using this plug-in system, every online laboratory can be integrated by developing a custom plug-in that makes requests to the remote laboratory. This plug-in does not manage the communications or internals of the remote laboratory; instead it redirects the ILS Platform user to the remote laboratory.

To describe this process, the sequence diagram in Figure 15 shows how each component interacts to provide lab access to the user. Basically, the user uses the ILS platform, which loads an OpenSocial app generated by the Smart Gate-

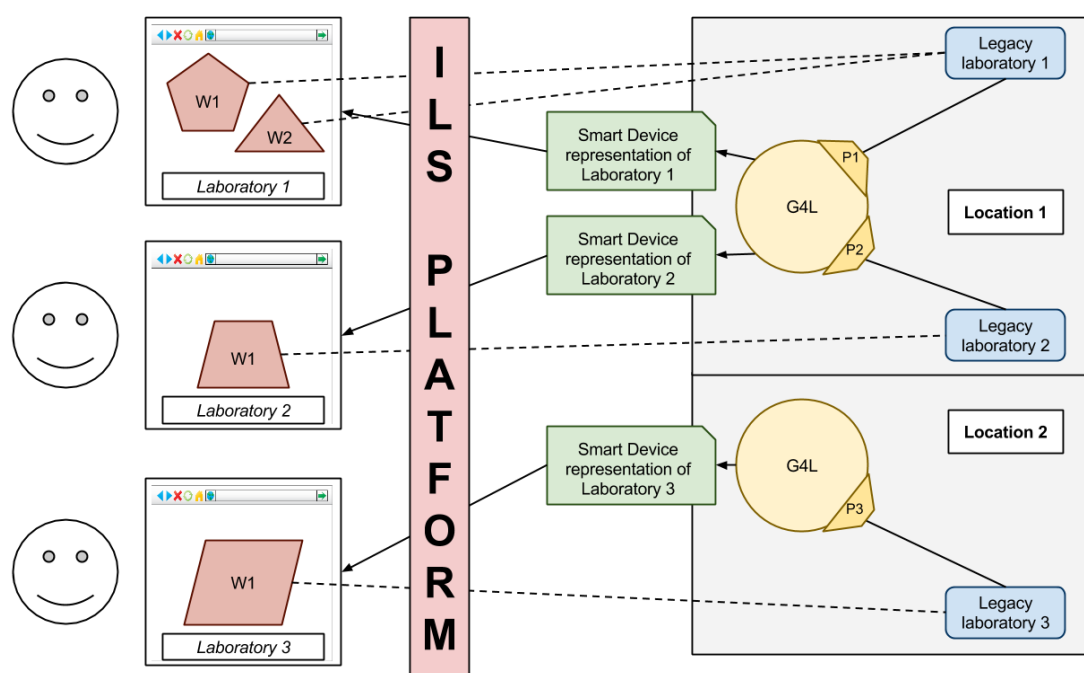


Figure 16: Example scenario of gateway4labs

way (gateway4labs LabManager). Whenever the user starts the reservation process, the Smart Gateway (gateway4labs LabManager) calls the reserve method of the particular plug-in. This plug-in knows how to interact with the final remote laboratory, so it will create the request using the protocol of the remote laboratory, and a URL to the final system will be returned. This URL should contain some kind of token or unique identifier to identify the current reservation, if the remote laboratory needs such system. Once the final location is opened, the user will interact directly with the final remote laboratory, using whatever technology is used by the remote laboratory.

Regarding the deployment, Figure 16 shows an example scenario, where two lab owners have three legacy laboratories (blue boxes in Figure 16) and two gateway4labs deployments (yellow circles in Figure 16). Location 1 has two legacy laboratories, managed with two different plug-ins (P1 and P2) in the same gateway4labs instance, which provides different Smart Device representations. Location 2 has its own gateway4labs instance with another plug-in for its own laboratory. This aims to represent that gateway4labs is not a component that aims to be centralized, while it can be centralized, as discussed later.

However, different deployment schemas are supported. As shown in Figure 16, one scheme would be to locate all components into one location (as illustrated in Figure 17 focused on a single location). Given that gateway4labs is Open Source, it is possible to deploy it in the same location as the Legacy Laboratory. This requires deploying the whole gateway4labs infrastructure (Python, MySQL, web server, public IP address, etc.). The advantages are that the latency in the reservation is low (the plug-in connects to the Legacy laboratory in the same

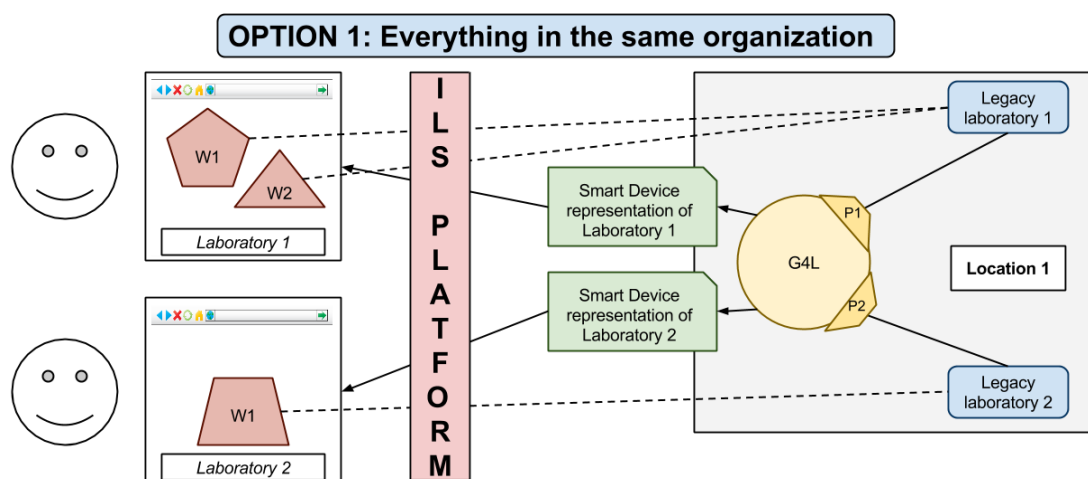


Figure 17: Smart Gateway deployment: first option: gateway4labs, plug-in and legacy laboratory in the same location.

network), it is independent from other actors (such as other providers), and upgrades in the plug-ins (such as changes in the Legacy laboratories) can be made without contacting third parties.

A second option is to have gateway4labs deployed in a different location. For example, gateway4labs could be deployed and maintained on the servers of one of the Go-Lab partners, including the plug-in, and the plug-in would connect to the Legacy laboratory located elsewhere. This is represented in Figure 18. The advantage of this option is that the deployment and maintenance of gateway4labs is not required by the laboratory owner. However, there is a slightly higher latency in the reservation process (since the requests have to go to one more institution), and the potential failures are doubled: the system will not be accessible when the legacy laboratory is down, but also when the gateway4labs location is down. Additionally, updates in the plug-in must be synchronized between both institutions. If a parameter is modified in the legacy laboratory and it must be changed in the plug-in, both must perform this change.

A third option (see Figure 19) is to decouple the gateway4labs plug-in from the gateway4labs deployment. By using a RESTful API, gateway4labs in the first location might not contain the plug-in itself, and it would be maintained in the same location where the legacy laboratory is. This way, while the latency would still need to cross through both institutions in the reservation process, the maintenance of the plug-in is held in the same location where the legacy laboratory is. If one parameter is going to be changed, it can be synchronized automatically without contacting the gateway4labs server.

Table 2 describes the different advantages and drawbacks from the three deployment options described above. As just described, regarding the deployment at the legacy laboratory location, Option 2 requires no maintenance, while Option 1 requires the full deployment and Option 3 requires the deployment and maintenance of the plug-in. Option 1 benefits from not relying on other actors

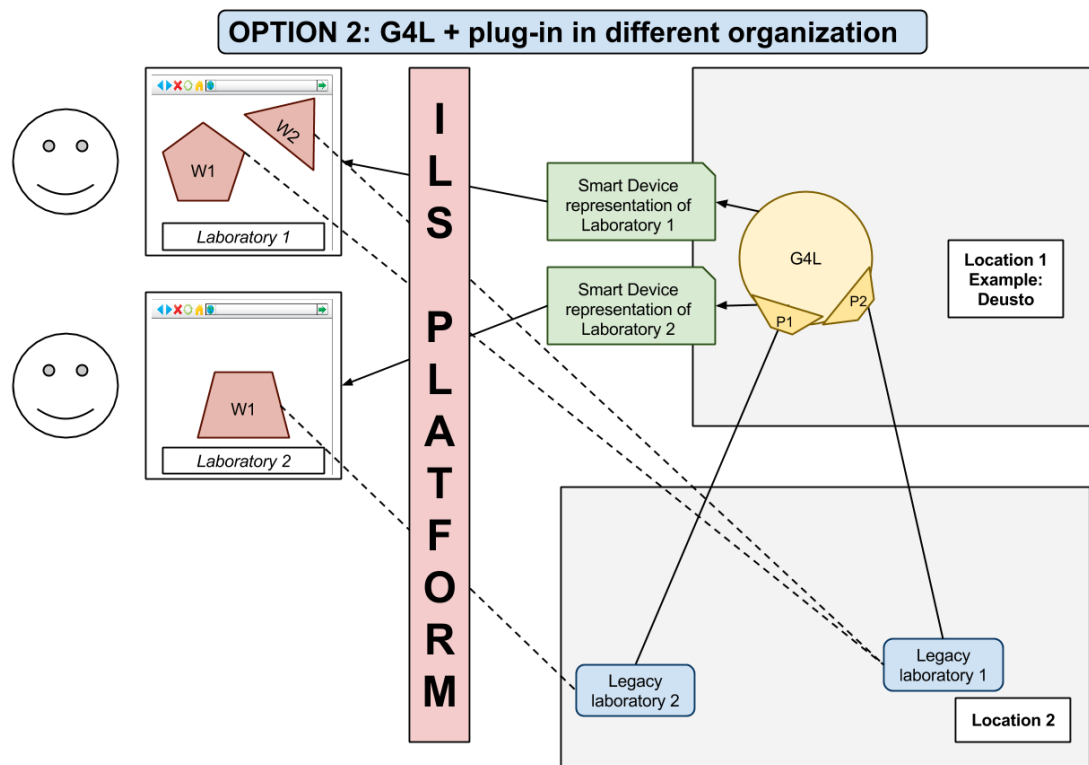


Figure 18: Shared deployment of gateway4labs: gateway4labs deployed in a location different to the legacy laboratory.

and getting a minimum latency in the reservation process since the request does not need to go through two different institutions. Regarding the maintenance of the plug-in for updates in the legacy laboratory that could affect the plug-in, in Options 1 and 3 the plug-in is deployed in the same location, so it is not a problem.

Table 2: Advantages and disadvantages of the different deployment options.

	Option 1	Option 2	Option 3
Deployment at lab side	Full	None	Plug-in deployment
Maintenance required?	Yes	No	Plug-in maintenance
Latency in reservation?	No	Yes	Yes
Multiple actors	No	Yes	Yes
Update management	Lab side	Intermediate side	Lab side

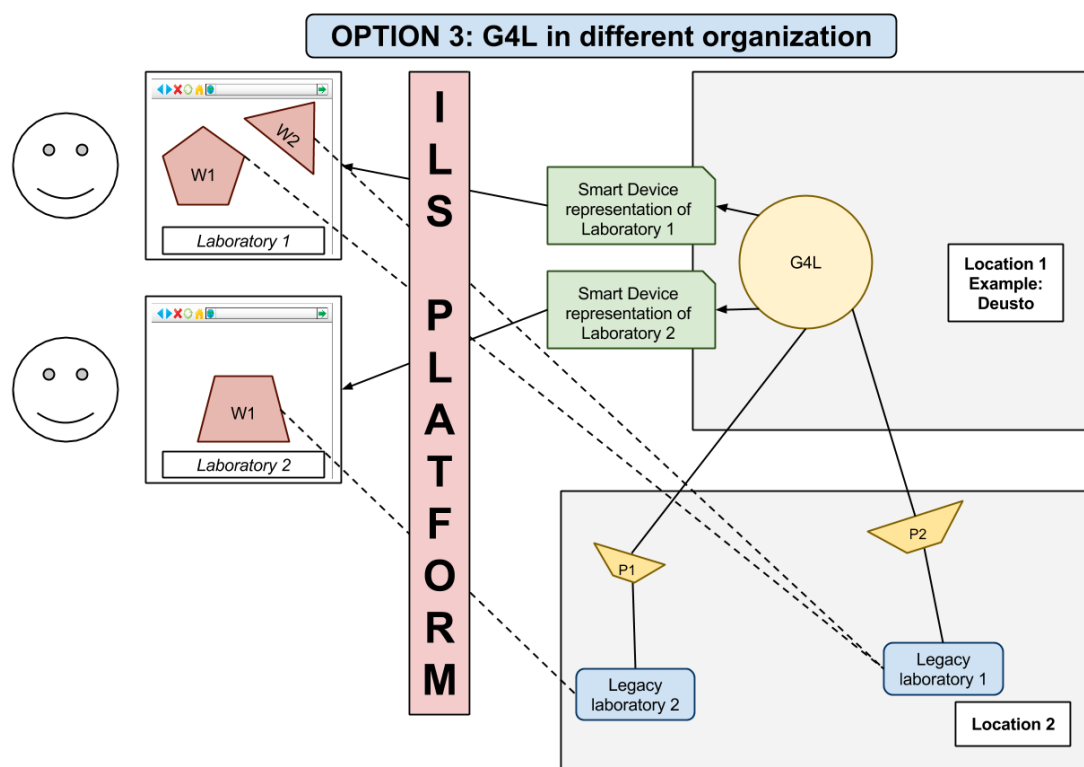


Figure 19: Plug-ins located in the same location as the legacy laboratories.

3.5.2 Specifications of the Plug-in System

The Plug-in system adds flexibility to the Smart Gateway since it allows lab owners to integrate their legacy online labs without the need to modify the gateway4labs software. A plug-in for a particular lab can be developed either by implementing its interface with the native gateway4labs API using the programming language of gateway4labs or by exposing its methods via an HTTP interface. This HTTP interface will be described in this section. For the methods described below basic HTTP Authentication is used.

A common argument in all the calls is the `context_id` argument. A particular plug-in (e.g., the ISA plug-in) could be used in more than one location (e.g., the ISA server in MIT and the ISA server in the University of Queensland in Australia (UQ)). Therefore, the Smart Gateway must allow that one plug-in supports more than one *context*, so the same plug-in stores the settings for each context, where one context could be MIT with certain credentials and other UQ with other credentials. The following HTTP request includes all necessary information and returns the current version of the supported APIs:

HTTP request: GET `BASE_URL/test_plugin?context_id=IDENTIFIER`

The Labmanager will call first this method, to determine which methods are supported by this version and can thus be used. If one method in the future requires

Table 3: Parameters of the `test_plugin` method

Parameter	Description	Parameter Type	Model
<code>context_id</code>	Can be used to Identify a specific lab or RLMS	URL	string

Table 4: Parameters of the `capabilities` method

Parameter	Description	Parameter Type	Model
<code>context_id</code>	Can be used to Identify a specific lab or RLMS	URL	string

an additional argument, a different version will be returned. The parameters are listed in Table 3 and an example response is shown in the following listing.

Listing 3.1: Test-plugin Response Example

```
{
  "valid": true,
  "g4l_api_version": "1.0"
}
```

HTTP request: GET `BASE_URL/capabilities?context_id=IDENTIFIER`

This HTTP call returns a list of optional capabilities provided by the plug-in. The only supported case at the time of this writing is the support for splitting the user interface in different apps. If this is not supported by the laboratory, there will be two methods less, and the labmanager will simply load the standard URL in the OpenSocial gadget. The parameters are listed in Table 4 and an example response in the listing below.

Listing 3.2: Capabilities Response Example

```
{
  "capabilities": ["widget"]
}
```

HTTP request: GET `BASE_URL/labs?context_id=IDENTIFIER`

Table 5: Parameters of the `labs` method

Parameter	Description	Parameter Type	Model
<code>context_id</code>	Can be used to Identify a specific lab or RLMS	URL	string

This HTTP request returns the list of available laboratories. In many plug-ins that support a single laboratory, this will be a fixed list with a single element. In other cases, it will call the remote laboratory to retrieve the list. For example, in the case of iLab or WebLab-Deusto, it actually lists the available laboratories for the credentials provided in the plug-in configuration. The parameters are listed in Table 5 and an example response is available in the listing below.

Listing 3.3: Labs Response Example

```
{
  "labs": [
    {
      "laboratory_id": "Sample_lab_1",
      "autoload": false,
      "name": "Sample Laboratory 1",
      "description": "This is an example of a laboratory"
    },
    {
      "laboratory_id": "Sample_lab_1",
      "autoload": true,
      "name": "Sample Laboratory 2",
      "description": "This is an example of a laboratory"
    }
  ]
}
```

HTTP request:

```
GET BASE_URL/widgets?context_id=IDENTIFIER&laboratory_id=LAB_ID
```

This returns a list of the available widgets (in our case OpenSocial apps). The parameters are listed in Table 6 and a response example in the listing below.

Listing 3.4: Widgets Response Example

```
{
  "widgets": [
    {
      "name": "Camera1",
      "description": "Left camera"
    }
  ]
}
```

Table 6: Parameters of the `widgets` method

Parameter	Description	Parameter Type	Model
<code>context_id</code>	Can be used to Identify a specific lab or RLMS	URL	string
<code>laboratory_id</code>	Identifies the labs for which the widgets should be retrieved	URL	string

```

    },
    {
      "name": "Camera2",
      "description": "Right camera"
    }
  ]
}

```

HTTP request:

GET `BASE_URL/widget?context_id=IDENTIFIER&widget_name=WIDGET`

X-G4L-reservation-id: (*reservation identifier obtained in `/reserve`*)

This HTTP request returns a URL to be loaded, given an existing reservation identifier, and a widget name. This URL should include enough information to enable the remote laboratory to display only the target fragment of the website. The parameters are listed in Table 7 and an example response with URL is illustrated in the listing below.

Listing 3.5: Widget Response Example

```

{
  "url": "BASE_URL/Camera1/?reservation_id=12345"
}

```

HTTP request: GET `BASE_URL/test_config?context_id=IDENTIFIER`

This request checks if the provided configuration works correctly. The parameters are listed in Table 8 and an example response is shown in the listing below.

Listing 3.6: Test-Config Response Example

```

{
  "valid": true,

```

Table 7: Parameters of the `widget` method

Parameter	Description	Parameter Type	Model
<code>context_id</code>	Can be used to Identify a specific lab or RLMS	URL	string
<code>widget_name</code>	Name of the widget for which the URL should be returned	URL	string
<code>X-G4L-reservation-id</code>	Reservation ID	header	string

Table 8: Parameters of the `test_config` method

Parameter	Description	Parameter Type	Model
<code>context_id</code>	Can be used to Identify a specific lab or RLMS	URL	string

```

    /* If false:
    "error_messages": [ "Error message 1", "Error message 2" ]
    */
}

```

HTTP request: POST `BASE_URL/reserve?context_id=IDENTIFIER`

This request performs a reservation and it returns both: a reservation identifier and a URL to be loaded. It receives generic information such as the user name (if available), the institution (if available), certain user properties (if available), and other arguments, such as the language used in the ILS platform or the URL that it should be loaded when finished so the Labmanager performs a new request. A complete list is presented in the data model of Listing 3.7. The method parameters are listed in Table 9 and an example response can be seen in the listing below.

Listing 3.7: Request Model Schema

```

request {
  laboratory_id (string): unique identifier for the lab,
  username (string),
  institution (string, optional),
  general_configuration_str (string, optional),
  particular_configurations (string, optional),
  request_payloa (string, optional),
  user_properties (string, optional),

```

Table 9: Parameters of the `reserve` method

Parameter	Description	Parameter Type	Model
<code>context_id</code>	Can be used to Identify a specific lab or RLMS	URL	string
<code>request</code>	A dictionary of terms to create a reservation	Body	Request model schema

```

    locale (string) : language used in the ILS platform,
    back (string, URL): address to send back the user once finished,
}

```

Listing 3.8: Reserve Response Example

```

{
  "load_url": "BASE_URL/Lab/url_bo_be_loaded",
  "reservation_id": "12345"
}

```

HTTP request: GET `BASE_URL/setup?context_id=IDENTIFIER&back_url=ADDRESS`

Additionally, certain systems require configuration settings. For example, many remote laboratories require a set of authentication credentials. In those cases, the plug-in must store these credentials somewhere, and the Smart Gateway administrator must configure them in a web application. This web application must be provided by the plug-in, so it can store the data in a local database.

So to implement this, the HTTP specification provides this method, which must return a URL that will be used to redirect the Smart Gateway administrator to it. In that URL, the plug-in is expected to provide the forms and required steps to configure the plug-in itself. This method also receives a `back_url` parameter to redirect the Smart Gateway administrator back to wherever he was before being redirected to the plug-in.

The plug-in developer must make sure that the URL contains some kind of secret so only authenticated users can change the settings of the plug-in. The parameters are listed in Table 10.

Listing 3.9: Setup Response Example

```

{
  "url": "http://somewhere-else/#secret=asecret"
}

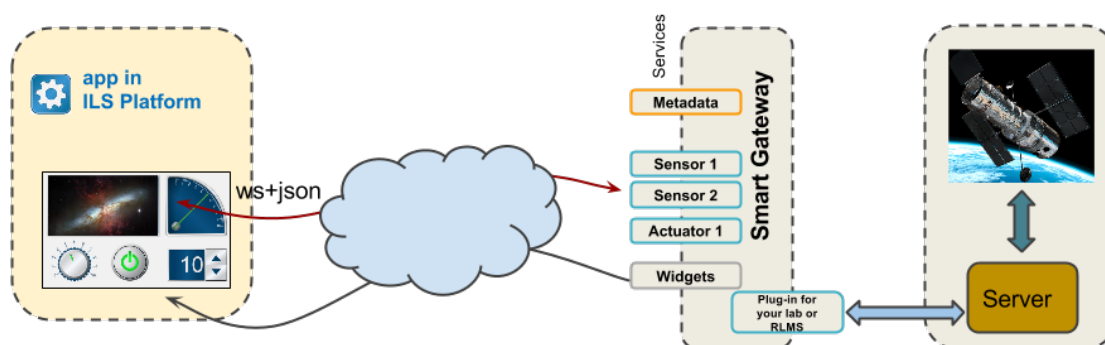
```

Table 10: Parameters of the setup method

Parameter	Description	Parameter Type	Model
context_id	Can be used to Identify a specific lab or RLMS	URL	string
back_url	Address to redirect the Smart Gateway administrator once the plug-in has been configured	URL	string

3.5.3 The Protocol Translator

The protocol translator is an additional and optional component of the Smart Gateway that mainly repackages legacy communication (legacy lab specific) and expose them as services compliant with the Smart Device specification (using WebSockets & JSON) (see Figure 20). Its implementation is lab specific and will usually serve one single legacy lab since the messages exchanged by these systems are often different and domain-specific.

**Figure 20: Protocol translator.**

For example, remote laboratory management systems such as WebLab-Deusto, ISA or Labshare Sahara manage all the laboratory reservations in the same way, but the communication of each laboratory is different. ISA could have a radioactivity laboratory and an electronics laboratory, and while both are managed with the same authentication and scheduling mechanisms, the number and type of sensors and actuators are completely different. For this reason, while a single plug-in for ISA is needed for all the ISA laboratories, a specific protocol translator for each particular laboratory must be implemented. This is detailed in Figure 21, where two RLMS require only one plug-in each, while each of the laboratories managed by each RLMS require its own protocol translator. In the case of Remote Laboratories not managed by a RLMS but by their own management system, the cardinality is different, since both a plug-in and a protocol translator would be required.

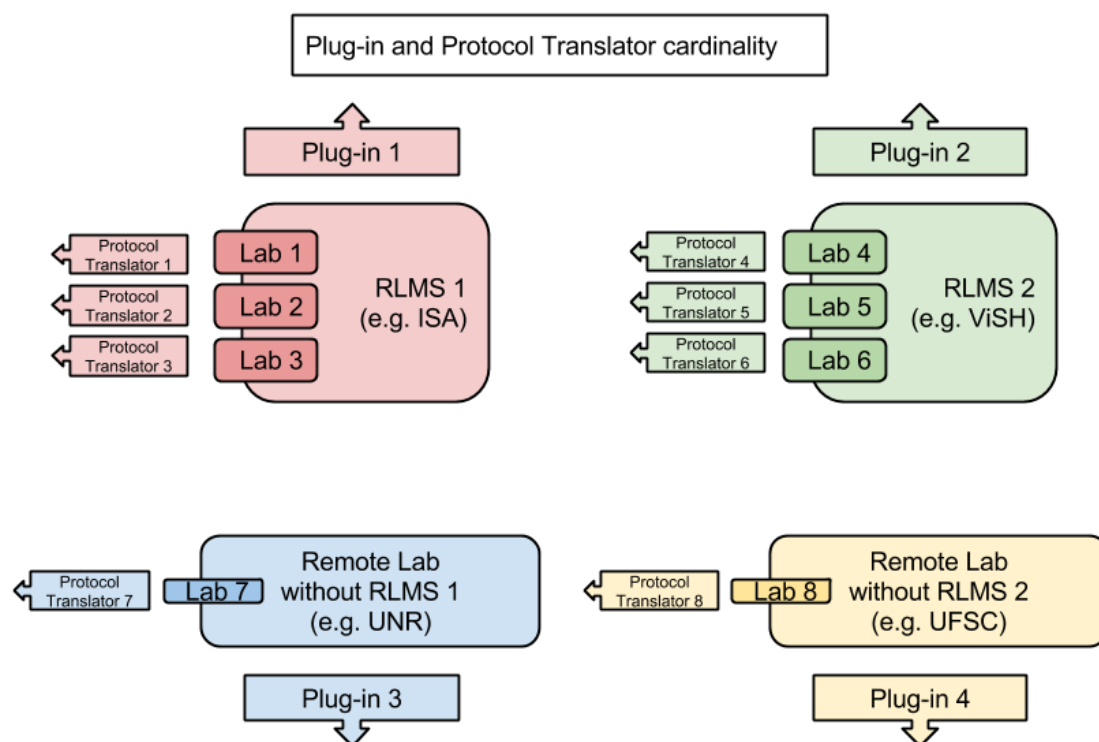


Figure 21: Plug-in and Protocol Translator Cardinality

From the Smart Device specification (see Section 2) the protocol translator must implement at least the following required services:

- Metadata service
- Sensor service
- Actuator service

Integrating a legacy lab with a protocol translator ensures a full compatibility with the Smart Device specification, however this approach is costly in terms of implementation effort since all messages exchanged between legacy client and server should be mapped to fit the sensor and actuator model of a Smart Device. The implementation efforts also vary depending on the legacy lab characteristics. For example, some legacy labs run asynchronously (or in batch mode), what poses constraints regarding the access to each sensor and actuator individually. A prototype and proof of concept of a batched online lab with protocol translator will be implemented and reported in D4.5 and D4.7.

Additionally, this approach assumes the use of a different protocol, so legacy client applications will not be reusable with the protocol translator, and the communications must be rewritten to be provided as Smart Device compliant services. One of the benefits of the Smart Device paradigm is that laboratory owners can reuse existing clients for other laboratories if they are complimentary to their services, so the whole client does not need to be reimplemented. However, those client pieces that have not been previously implemented by third parties

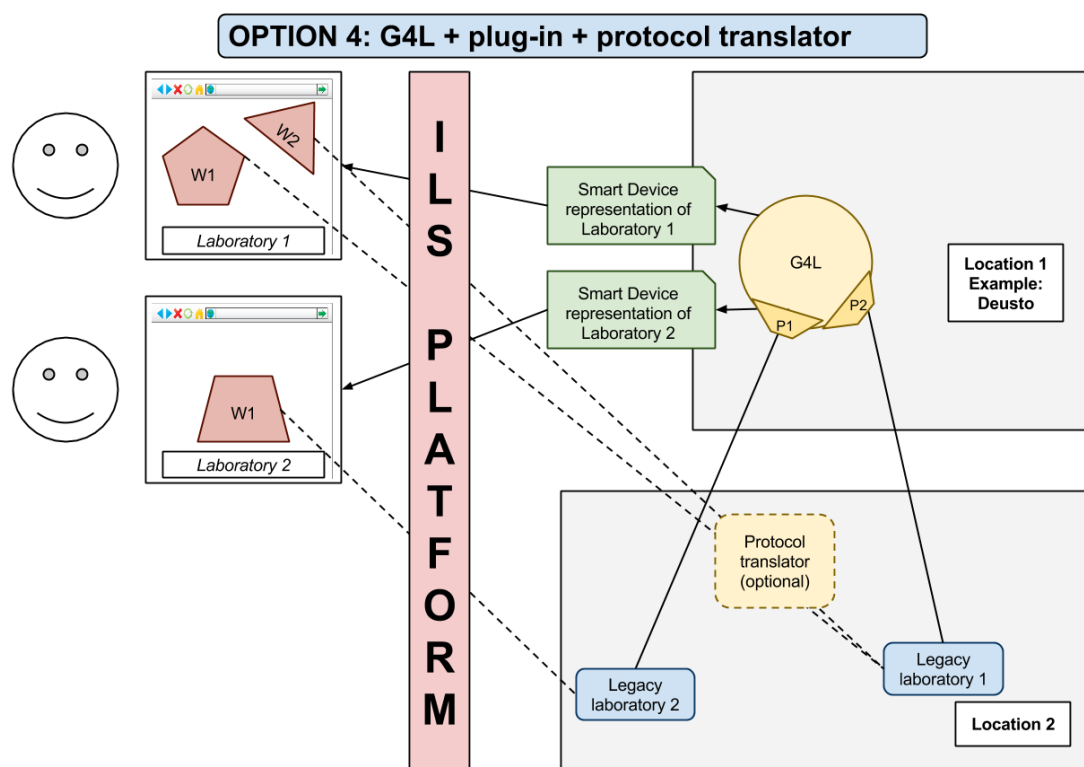


Figure 22: Smart Gateway deployment with a protocol translator

do need a reimplementation.

The most likely deployment configuration of the protocol translator is depicted in Figure 22. Since it is very specialized software and it is tightly coupled with the legacy lab system, it is likely to reside at the lab owner's side. This way, the latency would be minimal since the communication between the protocol translator and the legacy system would be local. The gateway4labs component could be deployed in any of the three options presented. As previously mentioned it is an optional component that requires a dedicated development for each legacy lab system. Its implementation efforts are comparable with the efforts necessary to redesign the legacy lab system according to the Smart Device specifications. Furthermore the use of a protocol translator would add a latency to the communication between the client app and legacy lab system since, even if it is in a local network, it needs to translate all the existing communications.

3.6 Benefits for Lab Owners

As previously outlined the main goal of the Go-Lab Cloud Services is attracting laboratory owners to the Go-Lab ecosystem.

- **Easy integration:** A flexible, pragmatic approach to integrate existing remote laboratories into gateway4labs through the plug-in mechanism and the management panels. The amount of code required is rather small, since it only acts as an initial bridge, and two interfaces (native API and HTTP API) are provided, so the Laboratory owner can support one or the

other. Multiple deployment schemas are supported, and many methods are optional.

- Additional Go-Lab incentives: Go-Lab will provide the laboratory owners several benefits. The most important one is the visibility of the laboratories. Thousands of students and teachers will be able to easily find the federated laboratories. Other benefits include the support of Go-Lab Add-on mechanisms, such as the booking mechanism. Certain remote laboratories might have a queue for managing students, which does not scale up to large amount of users. However, if the laboratory is integrated in gateway4labs and gateway4labs supports the booking mechanism of the Go-Lab portal, then the laboratory will be only available to students of groups which have booked the laboratory, reducing the amount of concurrent students.
- Use a federated approach: gateway4labs plug-ins support federation mechanisms if these are provided by the integrated systems. For example, WebLab-Deusto provides a federation protocol so one WebLab-Deusto system with 4 laboratories can share a subset of them to other WebLab-Deusto system. The plug-in of this WebLab-Deusto benefits of this feature so it translate requests from gateway4labs (which come from the ILS platform) as if it was an external WebLab-Deusto system requesting a laboratory for a local user. This federated approach enables remote laboratories to be also provided through their original portals or be integrated in other tools, while increasing their visibility by sharing them with Go-Lab.

4 Standardisation

To trigger and strengthen the adoption of the proposed standardisation of the lab-owner services as a Smart Device, WP4 and WP9 representatives are involved in the IEEE Working Group P1876¹ on Networked Smart Learning Objects for Online Laboratories. This group is sponsored by the IEEE Education Society. Institutions and projects cannot be represented in such a group. However, EPFL, UNED, CUAS and UD representatives are core members of this group and are contributing effectively thanks to the competencies and the solutions incubated, prototyped, implemented and validated in the Go-Lab project.

In addition to online meetings, two face-to-face meetings have already been held; one during the IEEE EDUCON conference 2013 in Berlin and one during the IEEE EDUCON conference 2014 in Istanbul. In the latter, it has been discussed to define the standard at three levels: A pedagogical level, a service level, and a communication protocol level. The pedagogical level is describing how to package resources in a standardised way and to enable their integration in learning environments (e.g. LMS, MOOC platforms or social media platforms). The ILS as defined in Go-Lab are instrumental at this level to conceptualize the pedagogical packaging. The service level is standardizing the way a client communicates with a remote lab. The abstraction layer provided by the Smart Device paradigm was well received as a proposal and has the potential to become the seed of the final IEEE specifications still to be drafted and finalised. Finally, the communication protocol level is standardizing the way all the loosely coupled services and platforms supporting the usage of remote labs could interoperate. The Smart Gateway proposal is showing directions and scheme for effective interoperability.

The draft structure of the IEEE P1876 has been discussed with lab owners during the Madrid workshop (see appendix D) and useful feedback has been collected. This feedback will be shared with the Work Group and contribute to align the standard with the Go-Lab vision and the lab owners' expectations.

¹IEEE Working Group P1876, <http://ieee-sa.centraldesktop.com/1876public/>

5 Conclusion

In this deliverable, we have provided revised specifications of the Smart Device (used for the lab owner services) and the Smart Gateway (used for the cloud services). Based on the recommendations of the reviewers, together with the outcome of continuous discussions among the Go-Lab partners and by collecting valuable feedback of external lab owners (see Section 1.2 and Appendix C), we have extended the Smart Device specifications in several areas (e.g. the metadata description or the configuration of sensors and actuators) (see Section 2). Apart from extending the specification, we put a lot of effort on more precisely defining the Smart Device services, the data formats and protocols to enable more well-defined interoperability between a Smart Device and a client app or external service. Furthermore, we have described our Smart Device using an existing, popular web service description language, Swagger, and added some Go-Lab specific extensions for our specific metadata needs (see Section 2.3.3). We have also showed various use cases of how the Smart Device specification can be used (see Section 2.4).

From our experience integrating RLMS in the Smart Gateway, we have also updated the specifications and architecture. Furthermore, different deployment options were discussed to optimise performance and management. Due to the implementation work involved for legacy labs to achieve full Smart Device compatibility, we have defined four integration levels, so lab owners can have a basic integration with the Go-Lab infrastructure with little effort.

Furthermore, in parallel with the work on the specifications, we have implemented several Smart Device prototypes and integrated multiple RLMS with the Smart Gateway. These prototypes are the first software release of the lab owner and cloud services and are documented in D4.3. In the future, we aim to integrate more labs in Go-Lab using either the Smart Device or the Smart Gateway. More specifically, we are collaborating with the external lab owners from our workshop (see Appendix D) to integrate their labs in Go-Lab, which will be documented in D4.7. These new labs will test our specifications and where weaknesses are found we will update them in D4.5.

6 Appendix A: Smart Device metadata specification details

The Swagger Web service description language supports REST Web services by default. To describe the Smart Device, we had to extend the Swagger specification as mentioned in Section 2.3.3. The two main reasons for this extension were:

- *WebSocket support*: Since Swagger only supports REST out of the box, we needed to extend Swagger to support WebSockets and the channeling of WebSockets.
- *Smart Device concurrency*: The concurrency mechanisms of the Smart Device needed to be described in its metadata. This was not supported in Swagger since this is typically not a feature of regular Web services.

We were able to limit further changes to the Swagger specification by providing some of the metadata via services, e.g. to retrieve the sensor and actuator metadata services are used. This section highlights what we have changed in the Swagger specification to support the Smart Device metadata. First, the changes needed for WebSockets will be described and then we will elaborate on the concurrency, to finalise with some small tidbits that were added for convenience and improved expression power. Overall, these limited adaptations show that the Swagger specification was an appropriate choice for the Smart Device.

6.1 Extensions for WebSockets

The extensions related to WebSockets are limited to the API JSON object in the main `apis` field. Listing 6.1 illustrates an example service description with the WebSocket extension.

Listing 6.1: Example API service description to illustrate the WebSocket extension

```
"apis": [
  {
    "path": "/logging",
    "protocol": "WebSocket",
    "produces": [
      "application/json"
    ],
    "operations": [
      {
        "method": "Send",
        "nickname": "getLoggingInfo",
        "summary": "Streams the current logging information of
          the user activities and the lab activities",
        "notes": "Returns a JSON array of Activity Stream
          objects, see http://activitystrea.ms/",
        "type": "LoggingInfoResponse",
        "websocketType": "text",
        "produces": "application/json",
```

```

        "parameters": [
            {
                "name": "message",
                "description": "the payload for the getLoggingInfo
                    service",
                "required": true,
                "paramType": "message",
                "type": "SimpleRequest",
                "allowMultiple": false
            }
        ],
        "responseMessages": [...]
    }
]

```

We have extended Swagger with the following fields:

- `protocol`: This field allows to specify whether the service is accessible via HTTP REST or WebSockets. This can also be used for other protocols, e.g. non HTTP-based streaming protocols or Tor.
- `websocketType`: This field is used to express the type of WebSocket the service uses to support regular and binary WebSockets. The possible values are: 'text' & 'binary'.
- `method` (repurposed & extended): The method field is normally used to express which HTTP method is used (e.g. GET or POST). It can be used to express the WebSocket method. Normally, this is 'Send' but Socket.io¹ provides additionally also an 'emit' method.
- `paramType` (repurposed & extended): The `paramType` field is reused but we have added the 'message' value to support WebSocket messages.

6.2 Extensions for Concurrency mechanisms

A root field was added to the Swagger specification, namely 'concurrency'. Listing 6.2 provides an example of the concurrency field. All sub-fields of this concurrency field are Go-Lab extensions and were extensively discussed in Section 2.3.3, thus we will not recapitulate it here.

Listing 6.2: Example of a concurrency field

```

"concurrency": {
    "interactionMode": "synchronous",
    "concurrencyScheme": "roles",
    "roleSelectionMechanism": ["race", "interruptor"],
    "roles": [
        {

```

¹Socket.io, <http://socket.io/>

```
        "role": "observer",
        "selectionMechanism": ["race"],
        "availableApis": ["getSensorData"]
    },
    {
        "role": "controller",
        "selectionMechanism": ["race"]
    },
    {
        "role": "admin",
        "selectionMechanism": ["interruptor"]
    }
]
}
```

6.3 Additional Minimal Extensions

6.3.1 Data Types

For some reason, the Swagger specification does not support all JSON Schema data types for the data models described in the 'models' field. To improve flexibility and expressiveness, we have added all JSON Schema data types and one for binary data. More specifically the following fields have been added:

- **object**: a JSON object
- **any**: any possible data type, e.g. a primitive JSON Schema type, null or a JSON object
- **binary**: for binary data

7 Appendix B: The Metadata Specification for an Example Smart Device

This appendix provides examples of the metadata for two Smart Devices. One example contains the metadata for the RED lab (see D4.3) and another example is from a fictitious lab that is used to illustrate some mechanisms and interaction possibilities in more detail. The latter has been mainly used as the running example throughout this deliverable. Both examples are available on GitHub and this appendix contains an exact copy to illustrate the status at the time this deliverable is submitted. This latest updated version of this specification can also be found on GitHub at <https://github.com/Go-Lab/smart-device-metadata>.

Both examples contain the metadata specification and a set of example calls that a client app makes to the Smart Device, and the respective responses of the Smart Device.

7.1 RED Smart Device

7.1.1 Metadata Specification

Listing 7.1 presents the metadata for the RED Lab Smart Device. As defined in this deliverable, the metadata of the actuators and sensors are provided through the `getSensorMetadata` and `getActuatorMetadata` services.

Listing 7.1: The RED Lab Smart Device metadata

```
{
  "apiVersion": "2.0.0",
  "swaggerVersion": "1.2",
  "basePath": "http://128.178.5.201:8080",
  "info": {
    "title": "RED 2.0 ws",
    "description": "Control the speed and the position of the disc.",
    "contact": "christophe.salzmann@epfl.ch",
    "license": "Apache 2.0",
    "licenseUrl": "http://www.apache.org/licenses/LICENSE-2.0.html"
  },
  "authorizations": {},
  "concurrency": {
    "interactionMode": "synchronous",
    "concurrencyScheme": "roles",
    "roleSelectionMechanism": ["race", "fixed role"],
    "roles": [
      {
        "role": "controller",
        "selectionMechanism": ["race"]
      }
    ]
  },
  "apis": [
    {
```

```

    "path": "/client",
    "protocol": "WebSocket",
    "produces": [
        "application/json"
    ],
    "operations": [
        {
            "method": "Send",
            "nickname": "getClients",
            "summary": "Return a list of all available clients",
            "notes": "Returns a JSON array with all the available
                clients",
            "type": "ClientResponse",
            "parameters": [
                {
                    "name": "message",
                    "description": "the payload for the getClients
                        service.",
                    "required": true,
                    "paramType": "message",
                    "type": "SimpleRequest",
                    "allowMultiple": false
                }
            ],
            "authorizations": { },
            "responseMessages": [
                {
                    "code": 402,
                    "message": "Too many users"
                },
                {
                    "code": 404,
                    "message": "Clients not found"
                },
                {
                    "code": 405,
                    "message": "Method not allowed. The requested
                        method is not allowed by this server."
                },
                {
                    "code": 422,
                    "message": "The request body is unprocessable"
                }
            ]
        }
    ]
},
{
    "path": "/sensor/",

```

```

    "protocol": "WebSocket",
    "produces": [
        "application/json"
    ],
    "operations": [
        {
            "method": "Send",
            "nickname": "getSensorMetadata",
            "summary": "List all sensors and their metadata",
            "type": "SensorMetadataResponse",
            "parameters": [
                {
                    "name": "message",
                    "description": "the payload for the
                        getSensorMetadata service",
                    "required": true,
                    "paramType": "message",
                    "type": "SimpleRequest",
                    "allowMultiple": false
                }
            ],
            "responseMessages": [
                {
                    "code": 402,
                    "message": "Too many users"
                },
                {
                    "code": 404,
                    "message": "No sensors found"
                },
                {
                    "code": 405,
                    "message": "Method not allowed. The requested
                        method is not allowed by this server."
                },
                {
                    "code": 422,
                    "message": "The request body is unprocessable"
                }
            ],
            "authorisations": {}
        },
        {
            "method": "Send",
            "nickname": "getSensorData",
            "summary": "Get data from the sensor with the given
                sensor identifier",
            "type": "SensorDataResponse",
            "parameters": [

```

```

        {
            "name": "message",
            "description": "The payload for the
                getSensorData service",
            "required": true,
            "type": "SensorDataRequest",
            "paramType": "message",
            "allowMultiple": false
        }
    ],
    "responseMessages": [
        {
            "code": 401,
            "message": "Unauthorised access. The
                authentication token is not valid"
        },
        {
            "code": 402,
            "message": "Too many users"
        },
        {
            "code": 404,
            "message": "No sensors found"
        },
        {
            "code": 405,
            "message": "Method not allowed. The requested
                method is not allowed by this server."
        },
        {
            "code": 422,
            "message": "The request body is unprocessable"
        }
    ]
}
]
},
{
    "path": "/actuator/",
    "protocol": "WebSocket",
    "produces": [
        "application/json"
    ],
    "operations": [
        {
            "method": "Send",
            "nickname": "getActuatorMetadata",
            "summary": "List all actuators and their metadata",
            "type": "ActuatorMetadataResponse",

```

```

    "parameters": [
      {
        "name": "message",
        "description": "the payload for the
          getActuatorMetadata service",
        "required": true,
        "paramType": "message",
        "type": "SimpleRequest",
        "allowMultiple": false
      }
    ],
    "responseMessages": [
      {
        "code": 404,
        "message": "No actuators found"
      },
      {
        "code": 405,
        "message": "Method not allowed. The requested
          method is not allowed by this server."
      },
      {
        "code": 422,
        "message": "The request body is unprocessable"
      }
    ],
    "authorizations": {}
  },
  {
    "method": "Send",
    "summary": "Send new data to the actuator with the
      given actuator identifier",
    "notes": "The parameters go into a JSON object send
      over the WebSocket",
    "type": "ActuatorDataResponse",
    "nickname": "sendActuatorData",
    "parameters": [
      {
        "name": "message",
        "description": "The payload for the
          sendActuatorData service",
        "required": true,
        "type": "ActuatorDataRequest",
        "paramType": "message",
        "allowMultiple": false
      }
    ],
    "responseMessages": [
      {

```

```

        "code": 401,
        "message": "Unauthorised access. The
                    authentication token is not valid"
    },
    {
        "code": 402,
        "message": "Too many users"
    },
    {
        "code": 404,
        "message": "No actuator not found"
    },
    {
        "code": 405,
        "message": "Method not allowed. The requested
                    method is not allowed by this server."
    },
    {
        "code": 422,
        "message": "The request body is unprocessable"
    }
]
}
]
},
],
"models": {
    "Client": {
        "id": "Client",
        "properties": {
            "type": {
                "type": "string",
                "description": "The type of client application",
                "enum": [
                    "OpenSocial Gadget",
                    "W3C widget",
                    "Web page",
                    "Java WebStart",
                    "Desktop application"
                ]
            },
            "url": {
                "type": "string",
                "description": "The URI where the client application
                            resides"
            }
        }
    },
    "ClientResponse": {

```

```

    "id": "ClientResponse",
    "properties": {
      "method": {
        "type": "string"
      },
      "clients": {
        "type": "array",
        "items": {
          "$ref": "Client"
        }
      }
    }
  },
  "Sensor": {
    "id": "Sensor",
    "required": [
      "sensorId", "fullName"
    ],
    "properties": {
      "sensorId": {
        "type": "string"
      },
      "fullName": {
        "type": "string"
      },
      "description": {
        "type": "string"
      },
      "websocketType": {
        "type": "string",
        "description": "the type of WebSocket. WebSockets can
          either be binary or textual.",
        "enum": [
          "text",
          "binary"
        ],
        "defaultValue": "text"
      },
      "singleWebSocketRecommended": {
        "type": "boolean",
        "description": "If this field is set to true it means
          that the smart device expects that a client opens
          a dedicated websocket for to read from this value",
        "defaultValue": false
      },
      "produces": {
        "type": "string",
        "description": "The mime-type of the data that is
          produced by this sensor. A list of mime types can

```

```

        be found at
        http://en.wikipedia.org/wiki/Internet_media_type",
        "defaultValue": "application/json"
    },
    "values": {
        "type": "array",
        "items": {
            "$ref": "Value"
        }
    },
    "configuration": {
        "type": "array",
        "description": "The configuration consists of an
            array of JSON objects that consist of parameter
            and type",
        "items": {
            "$ref": "ConfigurationMetadataItem"
        }
    },
    "accessMode": {
        "type": "AccessMode"
    }
}
},
"Value": {
    "id": "Value",
    "required": [
        "name"
    ],
    "properties": {
        "name": {
            "type": "string"
        },
        "unit": {
            "type": "string"
        },
        "rangeMinimum": {
            "type": "number",
            "format": "double"
        },
        "rangeMaximum": {
            "type": "number",
            "format": "double"
        },
        "rangeStep": {
            "type": "number",
            "format": "double"
        }
    },
    "lastMeasured": {

```



```

        "type": "date-time"
    },
    "updateFrequency": {
        "type": "number",
        "description": "The frequency in Hertz of which the
            sensor value updates",
        "format": "int"
    }
}
},
"ConfigurationMetadataItem": {
    "id": "ConfigurationMetadataItem",
    "required": [
        "parameter", "type"
    ],
    "properties": {
        "parameter": {
            "type": "string",
            "description": "The name of the configuration
                parameter"
        },
        "description": {
            "type": "string",
            "description": "This field can provide some more
                information on how this parameter should be used."
        },
        "type": {
            "type": "string",
            "description": "The data type of that this
                configuration parameters expects, e.g. number or
                string",
            "enum": [
                "integer",
                "long",
                "float",
                "double",
                "string",
                "byte",
                "boolean",
                "date",
                "dateTime",
                "object",
                "array",
                "any",
                "binary"
            ]
        }
    },
    "items": {
        "type": "string",

```

```

        "description": "This field should only be used when
                        the type is 'array'. It describes which types are
                        present within the array",
        "enum": [
            "integer",
            "long",
            "float",
            "double",
            "string",
            "byte",
            "boolean",
            "date",
            "dateTime",
            "object",
            "any",
            "binary"
        ]
    }
}
},
"AccessMode": {
    "id": "AccessMode",
    "properties": {
        "type": {
            "type": "string",
            "enum": [
                "push",
                "pull",
                "stream"
            ]
        }
    },
    "nominalUpdateInterval": {
        "type": "number",
        "format": "float"
    },
    "userModifiableFrequency": {
        "type": "boolean",
        "defaultValue": false
    }
}
},
"SimpleRequest": {
    "id": "SimpleRequest",
    "required": [
        "method"
    ],
    "properties": {
        "authToken": {
            "type": "string"
        }
    }
}

```

```

    },
    "method": {
      "type": "string",
      "description": "The method should be equal to the
        nickname of one of the provided services."
    }
  }
},
"SensorMetadataResponse": {
  "id": "SensorMetadataResponse",
  "required": [
    "method", "sensors"
  ],
  "properties": {
    "method": {
      "type": "string",
      "description": "The method should be equal to the
        nickname of one of the provided services."
    },
    "sensors": {
      "type": "array",
      "items": {
        "$ref": "Sensor"
      }
    }
  }
},
"SensorDataRequest": {
  "id": "SensorDataRequest",
  "required": [
    "authToken", "method", "sensorId"
  ],
  "properties": {
    "authToken": {
      "type": "string"
    },
    "method": {
      "type": "string",
      "description": "The method should be equal to the
        nickname of one of the provided services."
    },
    "sensorId": {
      "type": "string"
    },
    "updateFrequency": {
      "type": "number",
      "description": "The frequency in Hertz of which the
        sensor value updates",
      "format": "int"
    }
  }
}

```

```

    },
    "configuration": {
      "type": "array",
      "items": {
        "$ref": "ConfigurationItem"
      }
    },
    "accessRole": {
      "type": "string",
      "description": "This field contains one of the roles
        defined in the concurrency roles list. If
        accessRole is not defined, the controller role is
        assumed."
    }
  }
},
"ConfigurationItem": {
  "id": "ConfigurationItem",
  "required": [
    "parameter", "value"
  ],
  "properties": {
    "parameter": {
      "type": "string",
      "description": "The name of the configuration
        parameter"
    },
    "value": {
      "type": "any",
      "description": "The value to set the configuration
        parameter to. The type should equal the type given
        in the metadata for this sensor."
    }
  }
},
"SensorDataResponse": {
  "id": "SensorDataResponse",
  "required": [
    "method", "sensorId"
  ],
  "properties": {
    "method": {
      "type": "string",
      "description": "The method should be equal to the
        nickname of one of the provided services."
    },
    "sensorId": {
      "type": "string"
    }
  }
},

```

```

    "accessRole": {
      "type": "string",
      "description": "This field contains one of the roles
        defined in the concurrency roles list. If no roles
        are defined controller is returned. If the
        observer is returned, the observerMode field will
        be available with extra info on the status of the
        lab."
    },
    "responseData": {
      "type": "SensorResponseData",
      "description": "The data as measured by this sensor"
    },
    "payload": {
      "type": "any",
      "description": "This optional payload field can
        contain any JSON object that provides extra
        information on this sensor or the current
        measurement."
    },
    "observerMode": {
      "type": "ObserverMode",
      "description": "This field is only available if the
        accessRole field returns observer."
    }
  },
  "SensorResponseData": {
    "id": "SensorResponseData",
    "required": [],
    "properties": {
      "valueNames": {
        "type": "array",
        "description": "An ordered array with all the value
          names of this sensor. The same order will be
          applied to the data array and lastMeasured array.",
        "items": {
          "type": "string"
        }
      },
      "data": {
        "type": "array",
        "description": "An ordered array with all the data
          values of this sensor. Each data element in the
          array should be ordered in the same position of
          its corresponding value elements in the values
          array.",
        "items": {
          "type": "any"
        }
      }
    }
  }
}

```

```

    }
  },
  "lastMeasured": {
    "type": "array",
    "description": "An ordered array with all the data
      values of this sensor. Each data element in the
      array should be ordered in the same position of
      its corresponding value elements in the values
      array.",
    "items": {
      "type": "date-time"
    }
  }
},
"Actuator": {
  "id": "Actuator",
  "required": [
    "actuatorId", "fullName"
  ],
  "properties": {
    "actuatorId": {
      "type": "string"
    },
    "fullName": {
      "type": "string"
    },
    "description": {
      "type": "string"
    },
    "websocketType": {
      "type": "string",
      "description": "the type of WebSocket. WebSockets can
        either be binary or textual.",
      "enum": [
        "text",
        "binary"
      ],
      "defaultValue": "text"
    },
    "singleWebSocketRecommended": {
      "type": "boolean",
      "description": "If this field is set to true it means
        that the smart device expects that a client opens
        a dedicated websocket for to read from this value",
      "defaultValue": false
    }
  },
  "consumes": {
    "type": "string",

```

```

        "description": "The mime-type of the data that is
                        consumed by this actuator. A list of mime types
                        can be found at
                        http://en.wikipedia.org/wiki/Internet_media_type",
        "defaultValue": "application/json"
    },
    "produces": {
        "type": "string",
        "description": "The mime-type of the data that is
                        produced by this actuator. A list of mime types
                        can be found at
                        http://en.wikipedia.org/wiki/Internet_media_type",
        "defaultValue": "application/json"
    },
    "values": {
        "type": "array",
        "items": {
            "$ref": "Value"
        }
    },
    "configuration": {
        "type": "array",
        "description": "The configuration consists of an
                        array of JSON objects that consist of parameter
                        and type",
        "items": {
            "$ref": "ConfigurationMetadataItem"
        }
    },
    "accessMode": {
        "type": "AccessMode"
    }
},
"ActuatorMetadataResponse": {
    "id": "ActuatorMetadataResponse",
    "required": [
        "method", "actuators"
    ],
    "properties": {
        "method": {
            "type": "string",
            "description": "The method should be equal to the
                            nickname of one of the provided services."
        },
        "actuators": {
            "type": "array",
            "items": {
                "$ref": "Actuator"
            }
        }
    }
}

```

```

        },
        "description": "The list of actuator metadata
                        elements"
    }
}
},
"ActuatorDataRequest": {
    "id": "ActuatorDataRequest",
    "required": [
        "method", "actuatorId"
    ],
    "properties": {
        "authToken": {
            "type": "string"
        },
        "method": {
            "type": "string",
            "description": "The method should be equal to the
                            nickname of one of the provided services."
        },
        "actuatorId": {
            "type": "string"
        },
        "valueNames": {
            "type": "array",
            "description": "An ordered array with all the value
                            names of this sensor. The same order will be
                            applied to the data array and lastMeasured array.",
            "items": {
                "type": "string"
            }
        },
        "data": {
            "type": "array",
            "description": "An ordered array with all the data
                            values of this sensor. Each data element in the
                            array should be ordered in the same position of
                            its corresponding value elements in the valueNames
                            array.",
            "items": {
                "type": "any"
            }
        },
        "configuration": {
            "type": "array",
            "items": {
                "$ref": "ConfigurationItem"
            }
        }
    },
},

```



```

        "accessRole": {
            "type": "string",
            "description": "This field contains one of the roles
                           defined in the concurrency roles list. If
                           accessRole is not defined the controller role is
                           assumed."
        }
    },
    "ActuatorDataResponse": {
        "id": "ActuatorDataResponse",
        "required": [
            "method"
        ],
        "properties": {
            "method": {
                "type": "string",
                "description": "The method should be equal to the
                               nickname of one of the provided services."
            },
            "lastMeasured": {
                "type": "date-time"
            },
            "accessRole": {
                "type": "string",
                "description": "This field contains one of the roles
                               defined in the concurrency roles list. If no roles
                               are defined controller is returned. If the
                               observer is returned, the observerMode field will
                               be available with extra info on the status of the
                               lab."
            },
            "payload": {
                "type": "any",
                "description": "The payload can be useful for
                               describing a result that is returned, for instance
                               by using the SensorResponseData model. Since
                               results can differ from acknowledgements to result
                               data, the field is optional and can contain any
                               JSON object."
            },
            "observerMode": {
                "type": "ObserverMode",
                "description": "This field is only available if the
                               accessRole field returns observer."
            }
        }
    },
    "ObserverMode": {

```

```

    "id": "ObserverMode",
    "required": [],
    "properties": {
      "queueSize": {
        "type": "integer",
        "description": "Provides the length of the user
                        waiting queue that want to get control of the lab"
      },
      "queuePosition": {
        "type": "integer",
        "description": "Provides the position of the client
                        who made this call in the user waiting queue. This
                        value should be positive and smaller or equal to
                        queueSize."
      },
      "estimatedTimeUntilControl": {
        "type": "integer",
        "description": "The estimated waiting time from now
                        on until the client will get controllerMode
                        access. The time is expressed in seconds."
      }
    }
  }
}

```

7.1.2 Example Requests and Responses to the Smart Device Services

In this section, we will present a few example requests and responses that can be made to the RED Lab Smart Device.

getClients

Listing 7.2 and 7.3 illustrate respectively a request and response to the get-Clients service. Note, the URLs in the getClient response can differ.

Listing 7.2: Request to get all client apps

```

{
  "method": "getClient"
}

```

Listing 7.3: Response to get all client apps

```

{
  "method": "getClient",
  "clients": [
    {
      "type": "OpenSocial gadget",
      "url": "http://redlab.epfl.ch/client/s1.xml"
    },
  ],
}

```

```

    {
      "type": "OpenSocial gadget",
      "url": "http://redlab.epfl.ch/client/s2.xml"
    },
    {
      "type": "OpenSocial gadget",
      "url": "http://redlab.epfl.ch/client/a1.xml"
    }
  ]
}

```

getSensorMetadata

Listing 7.4 and 7.5 provide the request and response to get sensor metadata. Listing 7.5 shows the sensor metadata itself.

Listing 7.4: Request to get the sensor metadata

```

{
  "method": "getSensorMetadata"
}

```

Listing 7.5: Response to get the sensor metadata

```

{
  "method": "getSensorMetadata",
  "sensors": [
    {
      "sensorId": "position",
      "fullName": "position",
      "description": "the angular position of the wheel",
      "websocketType": "text",
      "singleWebSocketRecommended": true,
      "produces": "application/json",
      "values": [
        {
          "name": "angularPosition",
          "unit": "degree",
          "lastMeasured": "2014-06-23T18:25:43.511Z",
          "rangeMinimum": 30,
          "rangeMaximum": 330,
          "updateFrequency": 10
        }
      ],
      "accessMode": {
        "type": "push",
        "nominalUpdateInterval": 100,
        "userModifiableFrequency": false
      }
    }
  ],
}

```

```

    {
      "sensorId": "video",
      "fullName": "video feed",
      "description": "front camera video stream",
      "websocketType": "binary",
      "singleWebSocketRecommended": true,
      "produces": "image/jpeg",
      "values": [
        {
          "name": "video",
          "lastMeasured": "2014-06-23T18:28:43.617Z",
          "updateFrequency": 10
        }
      ],
      "accessMode": {
        "type": "push",
        "nominalUpdateInterval": 100,
        "userModifiableFrequency": false
      }
    }
  ]
}

```

getSensorData

Listing 7.6 and 7.7 show two requests for sensor data of the position and video sensor using the optional 'accessRole' field, which could be dropped here.

Listing 7.6: The request to get data for the 'position' sensor

```

{
  "method": "getSensorData",
  "sensorId": "position",
  "accessRole": "controller"
}

```

Listing 7.7: The request to get data for the 'video' sensor

```

{
  "method": "getSensorData",
  "sensorId": "video",
  "accessRole": "controller"
}

```

Listing 7.8 provides an example response for the position sensor.

Listing 7.8: The response to get data for the 'position' sensor

```

{
  "method": "getSensorData",

```

```

    "sensorId": "position",
    "accessRole": "controller",
    "responseData": {
      "valueNames": ["angularPosition"],
      "data": [54],
      "lastMeasured": ["2014-06-23T18:28:43.511Z"]
    }
  }
}

```

Listing 7.9 illustrates how the continuous stream of measurements can be stopped, by setting the 'updateFrequency' field to 0.

Listing 7.9: The request to interrupt the sensor measurement data flow

```

{
  "method": "getSensorData",
  "sensorId": "position",
  "updateFrequency": 0
}

```

getActuatorMetadata

Listing 7.10 and 7.11 show how the actuator metadata can be retrieved and the later presents the actuator metadata of the RED Lab Smart Device.

Listing 7.10: The request to get actuator metadata

```

{
  "method": "getActuatorMetadata"
}

```

Listing 7.11: The response to get actuator metadata

```

{
  "method": "getActuatorMetadata",
  "actuators": [
    {
      "actuatorId": "ref",
      "fullName": "reference",
      "description": "set the wheel position",
      "websocketType": "text",
      "produces": "application/json",
      "consumes": "application/json",
      "values": [
        {
          "name": "angularRef",
          "unit": "degree",
          "rangeMinimum": 30,
          "rangeMaximum": 330
        }
      ]
    }
  ]
}

```

```

    ],
    "accessMode": {
      "type": "push",
      "nominalUpdateInterval": 100,
      "userModifiableFrequency": false
    }
  }
]
}

```

getActuatorData

Listing 7.12 and 7.13 illustrated how an actuator can be set.

Listing 7.12: The request to set the reference actuator

```

{
  "method": "sendActuatorData",
  "accessRole": "controller",
  "actuatorId": "ref",
  "valueNames": ["angularRef"],
  "data": [84]
}

```

Listing 7.13: The response to set the reference actuator

```

{
  "method": "sendActuatorData",
  "lastMeasured": "2014-06-23T20:25:43.741Z",
  "accessRole": "controller",
  "payload": {
    "actuatorId": "ref",
    "valueNames": ["angularRef"],
    "data": [84]
  }
}

```

7.2 Running example Smart Device

7.2.1 Metadata Specification

Listing 7.14 presents the metadata for the running example Smart Device used throughout this deliverable.

Listing 7.14: The Example Smart Device metadata

```

{
  "apiVersion": "1.0.0",
  "swaggerVersion": "1.2",
  "basePath": "http://redlab.epfl.ch/smartdevice",
  "info": {

```

```

    "title": "RED Lab smart device",
    "description": "This is an example implementation of the Go-Lab
        smart device in LabView and demonstrates an mechatronics
        remote lab running at EPFL",
    "termsOfServiceUrl": "http://redlab.epfl.ch/terms/",
    "contact": "christophe.salzmann@epfl.ch",
    "license": "Apache 2.0",
    "licenseUrl": "http://www.apache.org/licenses/LICENSE-2.0.html"
},
"apis": [
    {
        "path": "/client",
        "protocol": "WebSocket",
        "produces": [
            "application/json"
        ],
        "operations": [
            {
                "method": "Send",
                "nickname": "getClients",
                "summary": "Return a list of all available clients",
                "notes": "Returns a JSON array with all the available
                    clients",
                "type": "ClientResponse",
                "parameters": [
                    {
                        "name": "message",
                        "description": "the payload for the getClients
                            service.",
                        "required": true,
                        "paramType": "message",
                        "type": "SimpleRequest",
                        "allowMultiple": false
                    }
                ],
                "authorizations": { },
                "responseMessages": [
                    {
                        "code": 402,
                        "message": "Too many users"
                    },
                    {
                        "code": 404,
                        "message": "Clients not found"
                    },
                    {
                        "code": 405,
                        "message": "Method not allowed. The requested
                            method is not allowed by this server."
                    }
                ]
            }
        ]
    }
]

```

```

        },
        {
            "code": 422,
            "message": "The request body is unprocessable"
        }
    ]
}
]
},
{
    "path": "/sensor/",
    "protocol": "WebSocket",
    "produces": [
        "application/json"
    ],
    "operations": [
        {
            "method": "Send",
            "nickname": "getSensorMetadata",
            "summary": "List all sensors and their metadata",
            "type": "SensorMetadataResponse",
            "parameters": [
                {
                    "name": "message",
                    "description": "the payload for the
                        getSensorMetadata service",
                    "required": true,
                    "paramType": "message",
                    "type": "SimpleRequest",
                    "allowMultiple": false
                }
            ],
            "responseMessages": [
                {
                    "code": 402,
                    "message": "Too many users"
                },
                {
                    "code": 404,
                    "message": "No sensors found"
                },
                {
                    "code": 405,
                    "message": "Method not allowed. The requested
                        method is not allowed by this server."
                },
                {
                    "code": 422,
                    "message": "The request body is unprocessable"
                }
            ]
        }
    ]
}

```



```

    }
  ],
  "authorisations": {}
},
{
  "method": "Send",
  "nickname": "getSensorData",
  "summary": "Get data from the sensor with the given
    sensor identifier",
  "type": "SensorDataResponse",
  "parameters": [
    {
      "name": "message",
      "description": "The payload for the
        getSensorData service",
      "required": true,
      "type": "SensorDataRequest",
      "paramType": "message",
      "allowMultiple": false
    }
  ],
  "responseMessages": [
    {
      "code": 401,
      "message": "Unauthorised access. The
        authentication token is not valid"
    },
    {
      "code": 402,
      "message": "Too many users"
    },
    {
      "code": 404,
      "message": "No sensors found"
    },
    {
      "code": 405,
      "message": "Method not allowed. The requested
        method is not allowed by this server."
    },
    {
      "code": 422,
      "message": "The request body is unprocessable"
    }
  ]
}
]
},
{

```

```

    "path": "/actuator/",
    "protocol": "WebSocket",
    "produces": [
        "application/json"
    ],
    "operations": [
        {
            "method": "Send",
            "nickname": "getActuatorMetadata",
            "summary": "List all actuators and their metadata",
            "type": "ActuatorMetadataResponse",
            "parameters": [
                {
                    "name": "message",
                    "description": "the payload for the
                        getActuatorMetadata service",
                    "required": true,
                    "paramType": "message",
                    "type": "SimpleRequest",
                    "allowMultiple": false
                }
            ],
            "responseMessages": [
                {
                    "code": 404,
                    "message": "No actuators found"
                },
                {
                    "code": 405,
                    "message": "Method not allowed. The requested
                        method is not allowed by this server."
                },
                {
                    "code": 422,
                    "message": "The request body is unprocessable"
                }
            ],
            "authorizations": {}
        },
        {
            "method": "Send",
            "summary": "Send new data to the actuator with the
                given actuator identifier",
            "notes": "The parameters go into a JSON object send
                over the WebSocket",
            "type": "ActuatorDataResponse",
            "nickname": "sendActuatorData",
            "parameters": [
                {

```

```

        "name": "message",
        "description": "The payload for the
            sendActuatorData service",
        "required": true,
        "type": "ActuatorDataRequest",
        "paramType": "message",
        "allowMultiple": false
    }
],
"responseMessages": [
    {
        "code": 401,
        "message": "Unauthorised access. The
            authentication token is not valid"
    },
    {
        "code": 402,
        "message": "Too many users"
    },
    {
        "code": 404,
        "message": "No actuator not found"
    },
    {
        "code": 405,
        "message": "Method not allowed. The requested
            method is not allowed by this server."
    },
    {
        "code": 422,
        "message": "The request body is unprocessable"
    }
]
}
]
},
{
    "path": "/logging",
    "protocol": "WebSocket",
    "produces": [
        "application/json"
    ],
    "operations": [
        {
            "method": "Send",
            "nickname": "getLoggingInfo",
            "summary": "Streams the current logging information
                of the user activities and the lab activities",
            "notes": "Returns a JSON array of Activity Stream

```

```

        objects, see http://activitystrea.ms/",
        "type": "LoggingInfoResponse",
        "websocketType": "text",
        "produces": "application/json",
        "parameters": [
            {
                "name": "message",
                "description": "the payload for the
                    getLoggingInfo service",
                "required": true,
                "paramType": "message",
                "type": "SimpleRequest",
                "allowMultiple": false
            }
        ]
    },
    ],
    "responseMessages": [
        {
            "code": 401,
            "message": "Unauthorised access. The authentication
                token is not valid"
        },
        {
            "code": 402,
            "message": "Too many users"
        },
        {
            "code": 405,
            "message": "Method not allowed. The requested method
                is not allowed by this server."
        },
        {
            "code": 422,
            "message": "The request body is unprocessable"
        }
    ]
}
],
"authorizations": {},
"concurrency": {
    /* Swagger extension: */
    "interactionMode": "synchronous", /* can also be 'asynchronous'
        */
    "concurrencyScheme": "roles", /* can also be 'concurrent' then
        all users have access at the same time */
    "roleSelectionMechanism": ["race", "interruptor"], /* can also
        be 'queue', 'fixed role', 'dynamic role' */
    "roles": [
        {

```

```

        "role": "observer",
        "selectionMechanism": ["race"],
        "availableApis": ["getSensorData"] /* a list of paths or
            operation nicknames */
    },
    {
        "role": "controller",
        "selectionMechanism": ["race"]
    },
    {
        "role": "admin",
        "selectionMechanism": ["interruptor"]
    }
]
},
"models": {
    "Client": {
        "id": "Client",
        "properties": {
            "type": {
                "type": "string",
                "description": "The type of client application",
                "enum": [
                    "OpenSocial Gadget",
                    "W3C widget",
                    "Web page",
                    "Java WebStart",
                    "Desktop application"
                ]
            },
            "url": {
                "type": "string",
                "description": "The URI where the client application
                    resides"
            }
        }
    },
    "ClientResponse": {
        "id": "ClientResponse",
        "properties": {
            "method": {
                "type": "string"
            },
            "clients": {
                "type": "array",
                "items": {
                    "$ref": "Client"
                }
            }
        }
    }
}

```

```

    }
  },
  "Sensor": {
    "id": "Sensor",
    "required": [
      "sensorId", "fullName"
    ],
    "properties": {
      "sensorId": {
        "type": "string"
      },
      "fullName": {
        "type": "string"
      },
      "description": {
        "type": "string"
      },
      "websocketType": {
        "type": "string",
        "description": "the type of WebSocket. WebSockets can
          either be binary or textual.",
        "enum": [
          "text",
          "binary"
        ],
        "defaultValue": "text"
      },
      "singleWebSocketRecommended": {
        "type": "boolean",
        "description": "If this field is set to true it means
          that the smart device expects that a client opens
          a dedicated websocket for to read from this value",
        "defaultValue": false
      },
      "produces": {
        "type": "string",
        "description": "The mime-type of the data that is
          produced by this sensor. A list of mime types can
          be found at
          http://en.wikipedia.org/wiki/Internet\_media\_type",
        "defaultValue": "application/json"
      },
      "values": {
        "type": "array",
        "items": {
          "$ref": "Value"
        }
      },
      "configuration": {

```

```

        "type": "array",
        "description": "The configuration consists of an
            array of JSON objects that consist of parameter
            and type",
        "items": {
            "$ref": "ConfigurationMetadataItem" /* extended
                Swagger with object type */
        }
    },
    "accessMode": {
        "type": "AccessMode"
    }
}
},
"Value": {
    "id": "Value",
    "required": [
        "name"
    ],
    "properties": {
        "name": {
            "type": "string"
        },
        "unit": {
            "type": "string"
        },
        "rangeMinimum": {
            "type": "number",
            "format": "double"
        },
        "rangeMaximum": {
            "type": "number",
            "format": "double"
        },
        "rangeStep": {
            "type": "number",
            "format": "double"
        },
        "lastMeasured": {
            "type": "date-time"
        },
        "updateFrequency": {
            "type": "number",
            "description": "The frequency in Hertz of which the
                sensor value updates",
            "format": "int"
        }
    }
}
},

```

```

"ConfigurationMetadataItem": {
  "id": "ConfigurationMetadataItem",
  "required": [
    "parameter", "type"
  ],
  "properties": {
    "parameter": {
      "type": "string",
      "description": "The name of the configuration
        parameter"
    },
    "description": {
      "type": "string",
      "description": "This field can provide some more
        information on how this parameter should be used."
    },
    "type": {
      "type": "string",
      "description": "The data type of that this
        configuration parameters expects, e.g. number or
        string",
      "enum": [
        "integer",
        "long",
        "float",
        "double",
        "string",
        "byte",
        "boolean",
        "date",
        "dateTime",
        "object",          /* extended Swagger with JSON
                           object type */
        "array",
        "any",             /* extended Swagger with any
                           type -- this represents any possible type */
        "binary"           /* extended Swagger with binary
                           type -- support for large binary config files
                           */
      ]
    },
  },
  "items": {
    "type": "string",
    "description": "This field should only be used when
      the type is 'array'. It describes which types are
      present within the array",
    "enum": [
      "integer",
      "long",

```



```

        "float",
        "double",
        "string",
        "byte",
        "boolean",
        "date",
        "dateTime",
        "object",          /* extended Swagger with JSON
                             object type */
        "any",             /* extended Swagger with any
                             type -- this represents any possible type */
        "binary"           /* extended Swagger with binary
                             type -- support for large binary config files
                             */
    ]
}
}
},
"AccessMode": {
    "id": "AccessMode",
    "properties": {
        "type": {
            "type": "string",
            "enum": [
                "push",
                "pull",
                "stream"
            ]
        },
    },
    "nominalUpdateInterval": {
        "type": "number",
        "format": "float"
    },
    "userModifiableFrequency": {
        "type": "boolean",
        "defaultValue": false
    }
},
},
"SimpleRequest": {
    "id": "SimpleRequest",
    "required": [
        "method"
    ],
    "properties": {
        "authToken": {
            "type": "string"
        },
    },
    "method": {

```

```

        "type": "string",
        "description": "The method should be equal to the
                        nickname of one of the provided services."
    }
}
},
"SensorMetadataResponse": {
    "id": "SensorMetadataResponse",
    "required": [
        "method", "sensors"
    ],
    "properties": {
        "method": {
            "type": "string",
            "description": "The method should be equal to the
                            nickname of one of the provided services."
        },
        "sensors": {
            "type": "array",
            "items": {
                "$ref": "Sensor"
            }
        }
    }
},
},
"SensorDataRequest": {
    "id": "SensorDataRequest",
    "required": [
        "authToken", "method", "sensorId"
    ],
    "properties": {
        "authToken": {
            "type": "string"
        },
        "method": {
            "type": "string",
            "description": "The method should be equal to the
                            nickname of one of the provided services."
        },
        "sensorId": {
            "type": "string"
        },
        "updateFrequency": {
            "type": "number",
            "description": "The frequency in Hertz of which the
                            sensor value updates",
            "format": "int"
        }
    },
    "configuration": {

```

```

        "type": "array",
        "items": {
            "$ref": "ConfigurationItem"
        }
    },
    "accessRole": {
        "type": "string",
        "description": "This field contains one of the roles
            defined in the concurrency roles list. If
            accessRole is not defined, the controller role is
            assumed."
    }
}
},
"ConfigurationItem": {
    "id": "ConfigurationItem",
    "required": [
        "parameter", "value"
    ],
    "properties": {
        "parameter": {
            "type": "string",
            "description": "The name of the configuration
                parameter"
        },
        "value": {
            "type": "any",          /* extended Swagger with any
                type -- this represents any possible type */
            "description": "The value to set the configuration
                parameter to. The type should equal the type given
                in the metadata for this sensor."
        }
    }
},
"SensorDataResponse": {
    "id": "SensorDataResponse",
    "required": [
        "method", "sensorId"
    ],
    "properties": {
        "method": {
            "type": "string",
            "description": "The method should be equal to the
                nickname of one of the provided services."
        },
        "sensorId": {
            "type": "string"
        }
    },
    "accessRole": {

```

```

        "type": "string",
        "description": "This field contains one of the roles
                        defined in the concurrency roles list. If no roles
                        are defined controller is returned. If the
                        observer is returned, the observerMode field will
                        be available with extra info on the status of the
                        lab."
    },
    "responseData": {
        "type": "SensorResponseData",
        "description": "The data as measured by this sensor"
    },
    "payload": {
        "type": "any",
        "description": "This optional payload field can
                        contain any JSON object that provides extra
                        information on this sensor or the current
                        measurement."
    },
    "observerMode": {
        "type": "ObserverMode",
        "description": "This field is only available if the
                        accessRole field returns observer."
    }
},
},
"SensorResponseData": {
    "id": "SensorResponseData",
    "required": [],
    "properties": {
        "valueNames": {
            "type": "array",
            "description": "An ordered array with all the value
                            names of this sensor. The same order will be
                            applied to the data array and lastMeasured array.",
            "items": {
                "type": "string"
            }
        },
        "data": {
            "type": "array",
            "description": "An ordered array with all the data
                            values of this sensor. Each data element in the
                            array should be ordered in the same position of
                            its corresponding value elements in the values
                            array.",
            "items": {
                "type": "any" /* extended Swagger with any
                             type -- this represents any possible type */
            }
        }
    }
}

```

```

    }
  },
  "lastMeasured": {
    "type": "array",
    "description": "An ordered array with all the data
      values of this sensor. Each data element in the
      array should be ordered in the same position of
      its corresponding value elements in the values
      array.",
    "items": {
      "type": "date-time"
    }
  }
},
"Actuator": {
  "id": "Actuator",
  "required": [
    "actuatorId", "fullName"
  ],
  "properties": {
    "actuatorId": {
      "type": "string"
    },
    "fullName": {
      "type": "string"
    },
    "description": {
      "type": "string"
    },
    "websocketType": {
      "type": "string",
      "description": "the type of WebSocket. WebSockets can
        either be binary or textual.",
      "enum": [
        "text",
        "binary"
      ],
      "defaultValue": "text"
    },
    "singleWebSocketRecommended": {
      "type": "boolean",
      "description": "If this field is set to true it means
        that the smart device expects that a client opens
        a dedicated websocket for to read from this value",
      "defaultValue": false
    }
  },
  "consumes": {
    "type": "string",

```

```

        "description": "The mime-type of the data that is
                        consumed by this actuator. A list of mime types
                        can be found at
                        http://en.wikipedia.org/wiki/Internet_media_type",
        "defaultValue": "application/json"
    },
    "produces": {
        "type": "string",
        "description": "The mime-type of the data that is
                        produced by this actuator. A list of mime types
                        can be found at
                        http://en.wikipedia.org/wiki/Internet_media_type",
        "defaultValue": "application/json"
    },
    "values": {
        "type": "array",
        "items": {
            "$ref": "Value"
        }
    },
    "configuration": {
        "type": "array",
        "description": "The configuration consists of an
                        array of JSON objects that consist of parameter
                        and type",
        "items": {
            "$ref": "ConfigurationMetadataItem" /* extended
                                                Swagger with object type */
        }
    },
    "accessMode": {
        "type": "AccessMode"
    }
},
"ActuatorMetadataResponse": {
    "id": "ActuatorMetadataResponse",
    "required": [
        "method", "actuators"
    ],
    "properties": {
        "method": {
            "type": "string",
            "description": "The method should be equal to the
                            nickname of one of the provided services."
        },
        "actuators": {
            "type": "array",
            "items": {

```

```

        "$ref": "Actuator"
      },
      "description": "The list of actuator metadata
        elements"
    }
  }
},
"ActuatorDataRequest": {
  "id": "ActuatorDataRequest",
  "required": [
    "method", "actuatorId"
  ],
  "properties": {
    "authToken": {
      "type": "string"
    },
    "method": {
      "type": "string",
      "description": "The method should be equal to the
        nickname of one of the provided services."
    },
    "actuatorId": {
      "type": "string"
    },
    "valueNames": {
      "type": "array",
      "description": "An ordered array with all the value
        names of this sensor. The same order will be
        applied to the data array and lastMeasured array.",
      "items": {
        "type": "string"
      }
    },
    "data": {
      "type": "array",
      "description": "An ordered array with all the data
        values of this sensor. Each data element in the
        array should be ordered in the same position of
        its corresponding value elements in the valueNames
        array.",
      "items": {
        "type": "any" /* extended Swagger with any
          type -- this represents any possible type */
      }
    },
    "configuration": {
      "type": "array",
      "items": {
        "$ref": "ConfigurationItem"
      }
    }
  }
}

```

```

    }
  },
  "accessRole": {
    "type": "string",
    "description": "This field contains one of the roles
    defined in the concurrency roles list. If
    accessRole is not defined the controller role is
    assumed."
  }
},
},
"ActuatorDataResponse": {
  "id": "ActuatorDataResponse",
  "required": [
    "method"
  ],
  "properties": {
    "method": {
      "type": "string",
      "description": "The method should be equal to the
      nickname of one of the provided services."
    },
    "lastMeasured": {
      "type": "date-time"
    },
    "accessRole": {
      "type": "string",
      "description": "This field contains one of the roles
      defined in the concurrency roles list. If no roles
      are defined controller is returned. If the
      observer is returned, the observerMode field will
      be available with extra info on the status of the
      lab."
    },
    "payload": {
      "type": "any",
      "description": "The payload can be useful for
      describing a result that is returned, for instance
      by using the SensorResponseData model. Since
      results can differ from acknowledgements to result
      data, the field is optional and can contain any
      JSON object."
    },
    "observerMode": {
      "type": "ObserverMode",
      "description": "This field is only available if the
      accessRole field returns observer."
    }
  }
}
}

```



```

    },
    "ObserverMode": {
      "id": "ObserverMode",
      "required": [],
      "properties": {
        "queueSize": {
          "type": "integer",
          "description": "Provides the length of the user
            waiting queue that want to get control of the lab"
        },
        "queuePosition": {
          "type": "integer",
          "description": "Provides the position of the client
            who made this call in the user waiting queue. This
            value should be positive and smaller or equal to
            queueSize."
        },
        "estimatedTimeUntilControl": {
          "type": "integer",
          "description": "The estimated waiting time from now
            on until the client will get controllerMode
            access. The time is expressed in seconds."
        }
      }
    },
    "LoggingInfoResponse": {
      "id": "LoggingInfoResponse",
      "required": [
        "method", "logs"
      ],
      "properties": {
        "method": {
          "type": "string"
        },
        "logs": {
          "type": "array",
          "items": {
            "type": "object",
            "description": "An Activity Stream object. This
              JSON object should follow the ActivityStreams
              1.0 JSON specification described at
              http://activitystrea.ms/specs/json/1.0/"
          }
        }
      }
    }
  }
}

```

7.2.2 Example Requests and Responses to the Smart Device Services

In this section, we will present a few example requests and responses that can be made with the running example specification.

getClientes

Listing 7.15: The request message to retrieve the client apps.

```
{  
  "method": "getClientes"  
}
```

Listing 7.16: The response message to retrieve the client apps.

```
{  
  "method": "getClientes",  
  "clients": [  
    {  
      "type": "OpenSocial gadget",  
      "url": "http://superlab.epfl.ch/client/dataviewer.xml"  
    },  
    {  
      "type": "OpenSocial gadget",  
      "url": "http://superlab.epfl.ch/client/video.xml"  
    },  
    {  
      "type": "OpenSocial gadget",  
      "url":  
        "http://superlab.epfl.ch/client/experiment-operator.xml"  
    }  
  ]  
}
```

getSensorMetadata

Listing 7.17: The request message to retrieve the sensor metadata.

```
{  
  "method": "getSensorMetadata"  
}
```

Listing 7.18: The response message to retrieve the sensor metadata.

```
{  
  "method": "getSensorMetadata",  
  "sensors": [  
    {  
      "sensorId": "3D-pos",  
      "fullName": "3D position",  
      "description": "the 3D position of the robot arm",  
    }  
  ]  
}
```

```

    "websocketType": "text",
    "produces": "application/json",
    "values": [
      {
        "name": "X",
        "unit": "cm",
        "lastMeasured": "2014-06-23T18:25:43.511Z",
        "rangeMinimum": 0.00,
        "rangeMaximum": 100.00,
        "rangeStep": 0.10,
        "updateFrequency": 10
      },
      {
        "name": "Y",
        "unit": "cm",
        "lastMeasured": "2014-06-23T18:25:43.511Z",
        "rangeMinimum": 0.00,
        "rangeMaximum": 100.00,
        "rangeStep": 0.10,
        "updateFrequency": 10
      },
      {
        "name": "Z",
        "unit": "cm",
        "lastMeasured": "2014-06-23T18:25:43.511Z",
        "rangeMinimum": 0.00,
        "rangeMaximum": 100.00,
        "rangeStep": 0.10,
        "updateFrequency": 10
      }
    ],
    "configuration": [
      {
        "parameter": "precision",
        "description": "The precision is expressed as a power
          of 10, e.g. to allow a precision of 0.01 the value
          will be -2 (from 10-2).",
        "type": "int"
      }
    ],
    "accessMode": {
      "type": "push",
      "nominalUpdateInterval": 100,
      "userModifiableFrequency": true
    }
  },
  {
    "sensorId": "video",
    "fullName": "video stream",

```

```

    "description": "front camera video stream",
    "websocketType": "binary",
    "singleWebSocketRecommended": true,
    "produces": "image/jpeg",
    "values": [
      {
        "name": "front",
        "lastMeasured": "2014-06-23T18:28:43.617Z",
        "updateFrequency": 10
      }
    ],
    "configuration": [
      {
        "parameter": "width",
        "type": "int"
      },
      {
        "parameter": "height",
        "type": "int"
      },
      {
        "parameter": "compression",
        "description": "The JPEG compression quality, ranging
          from 0 (lowest quality) to 100 (highest quality).",
        "type": "float"
      },
      {
        "parameter": "colourFilter",
        "description": "The colour value in an array of 3
          decimal RGB values",
        "type": "array",
        "items": "int"
      }
    ],
    "accessMode": {
      "type": "stream",
      "nominalUpdateInterval": 10,
      "userModifiableFrequency": true
    }
  }
]
}

```

getSensorData

Listing 7.19: The request message to retrieve the sensor data of a 3D position sensor with a configuration for the precision to be set to three decimal numbers. The controller access role is assumed and the Smart Device will react accordingly to the actual user role.

```
{
  "authToken": "dskds909ds8a76as675sa54",
  "method": "getSensorData",
  "sensorId": "3D-pos",
  "updateFrequency": 20,
  "configuration": [
    {
      "parameter": "precision",
      "value": 3
    }
  ],
  "accessRole": "controller"
}
```

Listing 7.20: The request message to gain the controller role of the sensor data of a 3D position sensor.

```
{
  "authToken": "dskds909ds8a76as675sa54",
  "method": "getSensorData",
  "sensorId": "3D-pos",
  "updateFrequency": 20,
  "accessRole": "interrupt",
  "configuration": [
    {
      "parameter": "precision",
      "value": -3
    }
  ]
}
```

Listing 7.21: The request message to retrieve the sensor data of a video sensor with a configuration for the dimensions, compression and colour filter of the video feed.

```
{
  "authToken": "dskds909ds8a76as675sa54",
  "method": "getSensorData",
  "sensorId": "video",
  "updateFrequency": 25,
  "accessRole": "controller",
  "configuration": [
    {
```

```

        "parameter": "width",
        "value": 640
    },
    {
        "parameter": "height",
        "value": 480
    },
    {
        "parameter": "compression",
        "value": 92.3
    },
    {
        "parameter": "colorFilter",
        "value": [60, 27, 229]
    }
]
}

```

Listing 7.22: The response message to retrieve the sensor data of a 3D position sensor.

```

{
    "method": "getSensorData",
    "sensorId": "3D-pos",
    "accessRole": "controller",
    "responseData": {
        "valueNames": ["X", "Y", "Z"],
        "data": [12.396, 23.681, 43.303],
        "lastMeasured": ["2014-06-23T18:28:43.511Z",
            "2014-06-23T18:28:43.511Z", "2014-06-23T18:28:43.511Z"]
    }
}

```

Listing 7.23: The request message to stop retrieving the sensor data of a 3D position sensor by setting the update frequency to 0.

```

{
    "method": "getSensorData",
    "sensorId": "3D-pos",
    "updateFrequency": 0
}

```

getActuatorMetadata

Listing 7.24: The request message to retrieve the actuator metadata.

```

{
    "method": "getActuatorMetadata"
}

```

Listing 7.25: The response message to retrieve the actuator metadata.

```

{
  "method": "getActuatorMetadata",
  "actuators":
  [
    {
      "actuatorId": "motor",
      "fullName": "Wheel motor",
      "description": "operate the motor of the wheel",
      "websocketType": "text",
      "produces": "application/json",
      "consumes": "application/json",
      "values": [
        {
          "name": "left",
          "unit": "radian",
          "rangeMinimum": 0.00,
          "rangeMaximum": 3.14,
          "rangeStep": 0.10,
          "updateFrequency": 10,
          "lastMeasured": "2014-06-23T19:25:43.511Z"
        },
        {
          "name": "right",
          "unit": "radian",
          "rangeMinimum": 0.00,
          "rangeMaximum": 3.14,
          "rangeStep": 0.10,
          "updateFrequency": 10,
          "lastMeasured": "2014-06-23T19:25:43.511Z"
        }
      ],
      "configuration": [
        {
          "parameter": "precision",
          "description": "The precision is expressed as a power
            of 10, e.g. to allow a precision of 0.01 the value
            will be -2 (from 10-2).",
          "type": "int"
        }
      ],
      "accessMode": {
        "type": "push",
        "nominalUpdateInterval": 100,
        "userModifiableFrequency": true
      }
    }
  ]
}

```

getActuatorData**Listing 7.26: The request message to set the motor actuator.**

```
{
  "authToken": "dskds909ds8a76as675sa54",
  "method": "sendActuatorData",
  "accessRole": "controller",
  "actuatorId": "motor",
  "valueNames": ["left"],
  "data": [1.90]
}
```

Listing 7.27: The response message to set the motor actuator.

```
{
  "method": "sendActuatorData",
  "lastMeasured": "2014-06-23T20:25:43.741Z",
  "accessRole": "controller",
  "payload": {
    /* could be useful for returning a result, but payload is
       optional */
    "actuatorId": "motor",
    "valueNames": ["left"],
    "data": [1.90]
  }
}
```

Listing 7.28: In case the user only has an observer role, this response message is returned. It provides data about how long the user will have to wait.

```
{
  "method": "sendActuatorData",
  "accessRole": "observer",
  "observerMode": {
    "queueSize": 7,
    "queuePosition": 4,
    "estimatedTimeUntilControl": 736
  }
}
```

getLoggingInfo**Listing 7.29: The request message to retrieve ActivityStreams logging information.**

```
{
```

```
"authToken": "dskds909ds8a76as675sa54;",  
"method": "getLoggingInfo"  
}
```

Listing 7.30: The response message to retrieve ActivityStreams logging information.

```
{  
  "method": "getLoggingInfo",  
  "logs": [  
    {  
      "verb": "access",  
      "published": "2014-06-23T18:25:43.511Z",  
      "language": "en",  
      "actor": {  
        "objectType": "person",  
        "id": "urn:example:person:martin",  
        "displayName": "Martin Smith",  
        "url": "http://example.org/martin",  
        "image": {  
          "url": "http://example.org/martin/image.jpg",  
          "mediaType": "image/jpeg",  
          "width": 250,  
          "height": 250  
        }  
      },  
      "object" : {  
        "objectType": "sensor",  
        "id": "urn:redlab:epfl:ch/3D-pos",  
        "url": "http://redlab.epfl.ch/smartdevice/sensors/3D-pos",  
        "displayName": "3D position"  
      },  
      "target" : {  
        "objectType": "lab",  
        "id": "urn:redlab:epfl:ch/smartdevice",  
        "displayName": "RED Lab",  
        "url": "http://redlab.epfl.ch/smartdevice/"  
      }  
    }  
  ]  
}
```

8 Appendix C: Value-proposition Canvas for Lab owner and Cloud Services

During a full-day meeting dedicated to D4.1, on Thursday, March 20 2014, Go-Lab WP4 partners collaboratively worked on a value-proposition canvas to better understand the needs of teachers and lab owners and how the Smart Device specification can alleviate these. The final result is displayed in Figure 23.

The remainder of this appendix explains the value-proposition canvas methodology and shows the results in detail. Furthermore, we provide an analysis of these results.

8.1 *The value-proposition canvas methodology*

In order to build a service that will be widely used, we should understand the needs of potential customers and provide a service that adequately fit these needs. For the requirement analysis, we rely on the Value Proposition Canvas (Osterwalder & Pigneur, 2010), a widely used methodology to think about this fit. The Value Proposition Canvas is part of the Business Model Canvas¹, which is a tool used by major organisations such as General Electric, PWC, Ernst & Young, Xerox, SAP, 3M, or NASA to reflect on novel products and services.

In a nutshell, the Value Proposition Canvas provides a simple yet powerful way to think how to establish a good fit between user expectations and a tool or service, which could be offered to users (see Figure 24).

The canvas has two sides. On the right side, the assumptions about the customers are listed independently of the tool or service. The jobs-to-be-done are the tasks that the customer wants to complete. This can be a functional job (e.g. solving a task), a social job (e.g. gaining status), an emotional job (e.g. feeling good), or basic needs (e.g. eating).

These jobs are associated with pains and gains. Identifying the pains and gains help to have empathy for the users in order to better answer their needs. Pains can be potential risks associated with the jobs, the costs in time or money, the things that make customers feel bad or that make them underperform, the barriers that prevent adoption, etc. Gains describe the benefits that customers expect, the potential things that would make them happier, make their job easier, or increase the adoption likelihood.

On the left side, there is the value proposition description. The idea is to list the features of the products and services that help customers perform their job. In order to highlight the fit between the features and the customer requirements, the features are described as gain creator and pain reliever. Gain creators describe how the product or service generates the gains identified for the customers with respect to the jobs to be done and pain relievers describe how the product or services relief the previously identified pains.

¹Business Model Generation, <http://www.businessmodelgeneration.com>



Figure 23: A photograph of the whiteboard with the final value-proposition canvas.

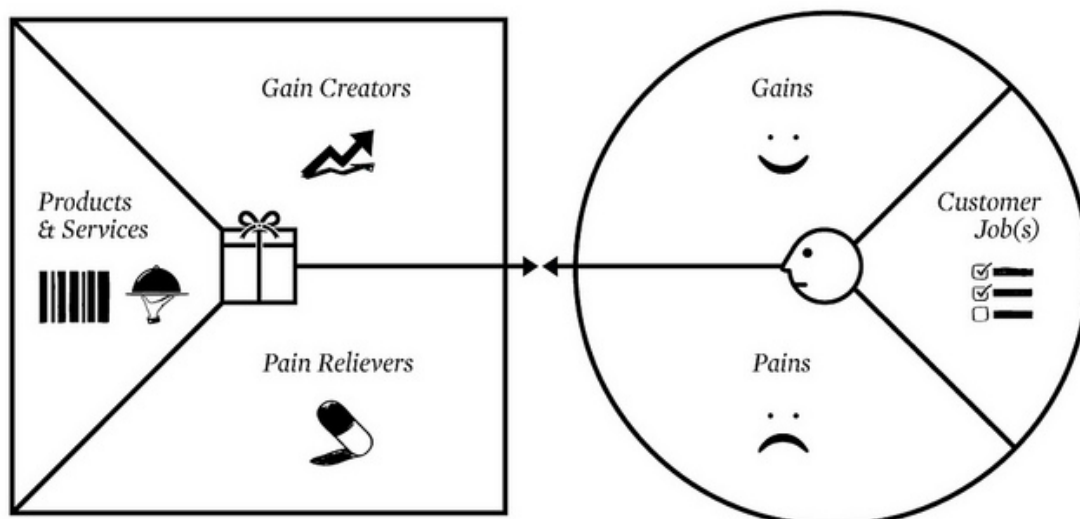


Figure 24: The Value Proposition Canvas

8.2 Detailed Results

The following section contains the transcript of all notes on the whiteboard in Figure 23. The transcripts are done literally and can contain spelling errors. This section is structured according to the different zones in the value-proposition canvas.

8.2.1 Customer

Customer Jobs

- Teachers
 - Easy/simple access
 - Use lab
 - He wants to have more labs
 - One entry point
 - He wants students to be able to repeat an experiment
 - He wants to know the results of the experiment
 - He wants to measure the student progress
 - He does not want to be aware of the lab maintenance (if possible)
 - Know how to use the lab
 - Know pedagogical contents available
 - Find practical experiences that fit the CV
 - He can define different levels of difficulty
 - Different hardware devices can be used
- Lab owners
 - He wants to have his labs 'safe'
 - He want to get more visibility online => more business
 - 'Remotify lab'
 - Use Go-Lab booking single sign-on Go-Lab infrastructure
 - Multi UI clients
 - Sustainability
 - Track lab usage: type of users, interests, interactions. . .

Pains

- Teachers
 - There is no remote lab for my needs
 - The Hw crashes during lab usage

- Complex labs which can not be simplified
- No standardisation UI /SSO/booking
- He has to explain several times the same concept
- Less experiential learning
- Not easy to use
- He has to plan the experiment for those who could not attend
- Attract the attention of the students
- Lab owners
 - He does not want to risk to change
 - Low usage
 - Only your own pedagogical contents
 - Upgrade / sustainability
 - He would like to control what is happening
 - He needs to send someone to install it
 - Give lots of instructions for audience
 - Remotify is difficult
 - Finding audience
 - Funding
 - i18n

Gains

- Teachers
 - Be involved in defining the space
 - Easily use of remote labs in class
 - Change the experience every year / for each class
 - User-tailored apps
 - He can know what other schools in other countries build as an experiment
 - Improve quality of teaching and learning
- Lab owner
 - More users for the lab
 - Less effort when developing
 - Interoperability
 - Does not start from scratch

- Durability, robust system
- Visibility / promotion
- Helps to continue in the market
- Portability of UI

8.2.2 Product

Products & Services

- Golabz / Graasp
- Collection of widgets
- Middleware (hw & sw) between the lab and UI, which can be different depending on the device, age, etc. Software lib.
- LA dashboard
- “Bible of Smart Device” - specifications
- Go-Lab booking system
- Best practices for remote labs
- Catalogs of labs
- Bartering platform

Pain Relievers

- Teachers
 - Portal for teachers
 - Catalog of labs + ILS
 - System for building different ILS
 - Lab / ILS tutor list
 - Build an attractive UI
 - Add videos, text, etc.
 - Booking system (authentication)
- Lab owner
 - Library (sw + hw) + steps to implement it
 - App composer -> translation
 - Test suite: performance benchmarking
 - Open standards
 - Publishing lab in Golabz
 - Bombproof system
 - Community of developers

- Good documentation
- Usage data
- Reuse of existing solutions through standardisation

Gain Creators

- Teachers
 - Lab catalogue
 - App catalogue
 - ILS catalogue
 - App composer
 - Learning analytics
 - ILS + booking
- Lab owner
 - Specifications + best practices
 - Go-Lab seal of approval
 - Standardised access to lab / infrastructure

8.3 Analysis of the results

In this section, we synthesised (grouped and rephrased) the notes of Section 8.2. The idea of this section is to elaborate and clarify the notes on the whiteboard of Figure 23.

To do this, we have first removed all notes directly related to other systems of the Go-Lab infrastructure (e.g. the lab repository & ILS platform) as well as unrelated notes to Smart Device. Second, we labelled the synthesised notes on different levels. The notes directly related to the Smart Device or Smart Gateway are in bold and with **(A)**. The notes that show an effect of using the Smart Device or Smart Gateway but are not directly caused by it are labelled with a **(B)**.

8.3.1 Customer

Customers Jobs:

- Teachers:
 - use remote labs in the course in an easy/user-friendly way (the technical details of the lab should be transparent)
 - have access to more online labs
 - ~~have one place to go to to find labs~~
 - allow students to repeat an experiment
 - ~~have access to the results of student experiments~~

- measure the student progress
 - ~~have access to manuals, tutorials, pedagogical material related to online labs and the experiences of other teachers~~
 - define the level of difficulty for students
 - wants to use labs on both computers & tablets
- Lab owners:
 - get more lab users: by increasing visibility
 - **(A) make lab secure applying secure software methodologies**
 - **(A) make existing labs remotely accessible**
 - **(A) support multiple UIs for a lab**
 - ~~make an online lab sustainable~~
 - **(B) track usage of a lab**

Pains

- Teachers:
 - ~~no appropriate remote labs are available~~
 - lab hardware crashes
 - ~~labs are too complex (on usability level)~~
 - **(A) each lab has its own UI, booking system, etc. (no standardisation/federation)**
 - ~~(hard to keep the attention of students)~~
- Lab owners:
 - **(A) maintenance and changing the implementation/upgrading of a lab is hard**
 - **(A) installation of a remote lab is hard**
 - ~~low lab usage and hard to find a wider audience~~
 - ~~the lab and the pedagogical approach are often integrated and cannot be changed~~
 - ~~sustainability (usage and funding)~~
 - ~~a lab can require a lot of documentation before users can operate it~~
 - **(A) it is hard to make a lab remotely accessible**
 - internationalisation is hard

Gains

- Teachers:
 - Easily use remote labs in the classroom

- ~~– Control which pedagogical models will be applied and define the content and structure for the students to use the lab (create an ILS)~~
 - ~~– Support for changing the UI and language for each age group~~
 - (A) Use apps that communicate with the labs for richer user experience and better integration of the lab with the pedagogical approaches.**
 - ~~– Share and use lesson plans of other teachers.~~
- Lab owner
 - ~~– Get more lab users through higher visibility & promotion~~
 - (A) Develop a remote lab with less effort**
 - (A) Get interoperability with different services that can enrich/improve your lab or make the development more efficient**
 - (A) Use an existing, robust, secure framework to build the lab on**
 - ~~– Get better sustainability~~
 - ~~– The lab UI can be used on different devices (portability)~~

8.3.2 Product

Products & Services

- ~~The Lab repository: Golabz~~
- ~~The ILS platform: Graasp~~
- Apps
- **(A) Middleware between lab hardware/software and Go-Lab infrastructure: software package and specification**
- Learning Analytics: dashboard, scaffolds, etc.
- Go-Lab booking system
- **(A) Best practices for implementing, running and managing remote labs.**
- ~~Bartering platform for tutoring~~

Pain Relievers

- Teachers:
 - ~~– Teacher portal: one access point~~
 - ~~– Catalog of labs & ILS~~
 - ~~– Platform to create ILS~~
 - ~~– Support & tutoring for labs and ILS~~
 - ~~– User-friendly user interface~~

- One unified booking system
- Lab owner:
 - **(A) Software library with middleware to implement a remote lab together with specifications and best practices**
 - * using open standards and well-documented software libraries
 - * standardisation of the specifications
 - Translation/Internationalisation tool
 - Test suite for performance benchmarking
 - Publishing lab in a public repository
 - Development support:
 - * Developer community
 - * proper documentation
 - Usage data and dashboard

Gain Creators

- Teachers:
 - ~~Lab, app and ILS catalog to limit the time searching for the required resources to prepare a course.~~
 - ~~app composer for editing and translating apps~~
 - learning analytics support
 - support for inquiry learning
 - unified simple booking mechanism
- Lab owner:
 - **(A) Clear specifications, documentation and best practices to ease development**
 - ~~“Go-Lab approved” badge to enable trust in a lab to motivate lab owners to optimise their lab~~
 - **(A) Standardised way to access a lab and the Go-Lab infrastructure**

8.3.3 Conclusion

Many notes on the value-proposition canvas are not directly related to the Smart Device and Smart Gateway specification, but are rather the effect of the whole Go-Lab infrastructure (i.e., the ILS Platform (see D5.2), the Lab Repository (see D5.2), the App Composer (see D5.2), the Learning Analytics service (see D4.2) and the Add-on Services (see D4.2)). Of course, since the Smart Device and Smart Gateway specification enables interoperability with various parts of the Go-Lab infrastructure, one could consider the specification as supporting or

even enabling these platforms and services. But when, we focus really on the specification and the software packages implementing the specification that Go-Lab provides, the major impacts of it are the following:

- For teachers:
 - a standardised UI for booking and potentially using labs
 - interoperability between lab and apps, thus enabling richer user experience and better integration of the lab with the pedagogical approaches.
 - better support for learning analytics
- For lab owners:
 - a robust, secure implementation of the specification to build their lab on, resulting in faster development and more robust and secure deployment
 - the specification and software packages enable a way to easily make existing labs available online
 - due to the shared specification, client UIs can be reused.
 - make easily use of extra functionality provided by the Go-Lab infrastructure which can make the lab more user-friendly and can speed up development, e.g. a booking mechanism or learning analytics services.
 - enable interoperability with existing apps, which again can speed up development and increase the attractiveness of the lab for users.
 - due to a clear specification and use of open standards the labs will be usable on more software platforms, and devices.

Furthermore, the value-proposition canvas exercise also uncovered the need to have clear specifications, as well as secure and robust Smart Device software libraries that enable lab owners to quickly develop and deploy labs with less effort. To completely achieve this, we also have to provide best practices and clear documentation. Probably, we will also need to support an online community of Go-Lab Smart Device developers and provide common functionality in the software packages.

In conclusion, teachers benefit of the Smart Device and Smart Gateway specification through the interoperability it enables with the other Go-Lab platforms and services, which creates the functionality they want (e.g. learning analytics and better user experience). On the other hand, the lab owners benefit from the specifications, because they are able to develop their labs faster on a secure and robust platform. Furthermore, the interoperability with the Go-Lab infrastructure enables additional functionality that the lab owners do not have to provide neither implement, e.g. lab owners do not have to implement their own booking system and can make use of the Go-Lab booking system.

9 Appendix D: Lab Owner Workshop

A workshop has been organised with lab-owners from June 3-6 in Madrid in order to get feedback from external experts (lab owners) on the draft specifications for the Smart Device (plug technologies) and the Smart Gateway (cloud services) to contribute to the preparation of this deliverable and to evaluate the possible adoption level.

Internal participants include 4 representatives of EPFL, one of CUAS, one of UD and 7 of UNED.

Four external experts have been invited. They have been selected in the framework of the REV conference as leaders of innovative and prominent remote lab management platforms not part of Go-Lab, and as such active lab owners:

- Akassio Miranda Silva from RExLab¹ Universidade Federal de Santa Catarina (Brazil) and visiting scholar at Edge Hill University (GB)
- Franz Schauer from Tomas Bata University (CZ) and in charge of Rem-LabNet (Schauer et al., 2014) (ISES²)
- Igor Titov from Labicom³ (RU)
- Rui Pedro Borralho Neto from IST⁴ and Linkare⁵ (PT)

During the first day, the Go-Lab project was presented and external participants introduced their platforms.

Labicom has a focus on sharing large scientific experiments. Their competences in interfacing complex instrumentation in LabVIEW is useful for Go-Lab.

Linkare has a focus on a Java framework with various applications, including remote Labs. Their architecture is based on standardised plug-ins to connect new labs. The end-user solution is a single Java application in which labs can be selected.

Rexlab is a non-profit organisation with a focus on offering remote labs to schools through mobile devices. They have an approach similar to Go-Lab focusing on HTML5 and Raspberry Pi.

RemLabNet is a remote lab management system (RLMS) exploited to provide access to a network of remote labs dedicated to secondary schools. They have about 40 very interesting labs. They rely on a single but essential scaffolding application: a data collector.

9.1 Value Proposition

An interesting discussion was held regarding the value proposition that Go-Lab could offer to lab owners. The main request was to provide a strong integration

¹RExLab, <http://rexlab.ufsc.br/?q=en>

²ISES, <http://www.ises.info/index.php/en/laboratory>

³Labicom, <https://labicom.net/home.php>

⁴IST, http://www.elab.ist.utl.pt/?page_id=118

⁵Linkare, <http://www.linkare.com/en/>

between technology and pedagogy (offered by the ILS) and to bring more users (adoption through demonstration and dissemination thanks to the repository). The latter being a trigger to additional national funding, customers or sponsors depending of the business model of the lab owners. The challenge associated to improving and sustaining remote labs was also mentioned (through user feedback and natural selection).

Support for authoring end-user interfaces was also mentioned as one of the most interesting technology to enable seamless integration (App composer and ILS Platform).

9.2 Smart Device and Interfacing

The Smart Device concept was introduced and the corresponding specifications were discussed in detail. It is clear that this part would be more interesting for new lab owners not having yet their own platform (so, the experts present in this workshop do not really correspond to the target public).

The abstraction of the sensors and actuators with their corresponding metadata were well received. The experts proposed to add the concept of 'instruments' to ease the representation of complex measurement devices that can be found in complex remote labs. We agreed on the fact that a remote lab is a physical experiment (the object of the observation) with sensors and actuators, as well as its instrumentation (also referred altogether as a rig in other frameworks). So, the Smart Device is the abstraction of this rig (which is a unit which should be booked as a whole).

The relevant need to add timestamps on all sensors and actuator samples was raised and accepted.

The parallel standardisation carried out in the framework of the IEEE P1876 work group was also discussed. The participants agreed that the Smart Device specifications should be integrated. They were however not in favour of having IEEE defining dedicated communication protocols for remote labs (which is not the objective of Go-Lab anyway). The lab owners appreciated the chance to have Go-Lab partners traveling to IEEE standardisation meetings as representatives of the European lab owners as they do not have resources to do so. The need to standardize remote labs at a pedagogical level was also questioned. The conclusion was that an ILS could be standardised using the existing SCORM / LOM standards (to be checked), while a remote lab itself should stay agnostic with respect to the pedagogy (the same lab can be used in very different settings and scenarios).

9.3 Smart Gateway and Federation

The Smart Gateway concepts and solutions were introduced and simple solutions to connect the platforms of the external experts were presented. All experts but labicom were positive regarding an incremental integration of their labs through the Smart Gateway, starting first with just the harvesting of their list of labs and basic schemes enabling authentication. They are also interested by

the opportunity to tackle internationalization through the App Composer.

They are however expecting to have Go-Lab sponsoring for such an integration using schemes to be defined in bilateral negotiations, as they have all trouble to sustain their developments and platforms. For Linkare, the main idea is to translate their client application in JavaScript (it would be a meta-client giving access to various labs). For Rexlab, the main idea is to integrate their lab in ILS through iframes and share experience about the hardware implementation of the Smart Device on embedded boards. They are really willing to show the Brazilian government that they provide access to more labs through the collaboration with Go-Lab (to get more funding). For RemLabNet, as they are developing their own platform, they are not interested by the Go-Lab services but they are strongly interested to enable interoperability between RLMS. They are also interested to advertize labs in the Go-Lab repository and continue the update of their end-user interfaces to Javascript.

As a conclusion, we figured out that we should communicate more about the Smart Gateway as a way to enable federation of remote labs for dissemination purpose in a broad collaboration framework between lab owners (it is more about what will become visible in the repository than what is the enabling technology). This should be achieved through manual harvesting carried out together by lab owners and Go-Lab representatives.

Following the discussion on the federation of labs, the metadata devised by WP2 were presented and discussed for about an hour. This feedback has been communicated to WP2.

References

- Auer, M., Pester, A., Ursutiu, D., & Samoila, C. (2003, Dec). Distributed virtual and remote labs in engineering. In *Industrial technology, 2003 IEEE international conference on* (Vol. 2, p. 1208-1213 Vol.2). doi: 10.1109/ICIT.2003.1290837
- Bellido, L., Villagra, V., & Mateos, V. (2010). Federated authentication and authorization for reusable learning objects. In (p. 1071-1074). IEEE. Retrieved 2014-07-01, from <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5492459> doi: 10.1109/EDUCON.2010.5492459
- Cascado, D., Sevillano, J. L., Fernández-Luque, L., Johan-Gr  ttum, K., Vognild, L. K., & Burkow, T. M. (2011). Standards and implementation of pervasive computing applications. In *Pervasive computing and networking* (pp. 135–158). John Wiley & Sons, Ltd. Retrieved from <http://dx.doi.org/10.1002/9781119970422.ch9> doi: 10.1002/9781119970422.ch9
- Freed, N., Baker, M., & Hoehrmann, B. (2014). *Media types* (Tech. Rep.). Retrieved from <http://www.iana.org/assignments/media-types/media-types.xhtml>
- Hadley, M. J. (2009). *Web application description language (WADL)* (Tech. Rep.). Sun Microsystems Inc. Retrieved from <http://java.net/projects/wadl/sources/svn/content/trunk/www/wadl20090202.pdf>
- Hardison, J., & Garbi Zutin, D. (2011). *The ilab shared architecture: A web services infrastructure to build communities of internet accessible laboratories*. IGI Global. Retrieved 2014-07-01, from <http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/978-1-61350-186-3>
- Harward, V., del Alamo, J., Lerman, S., Bailey, P., Carpenter, J., DeLong, K., ... Zych, D. (2008). The iLab shared architecture: A web services infrastructure to build communities of internet accessible laboratories. , 96(6), 931-950. Retrieved 2014-07-01, from <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4527087> doi: 10.1109/JPROC.2008.921607
- Hypertext transfer protocol (HTTP) status code registry (RFC7231)* (Tech. Rep.). (2014). IETF. Retrieved from <http://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml>
- Lindsay, E., Stumpers, B., & others. (2014). Remote laboratories: enhancing accredited engineering degree programs. Retrieved 2014-07-01, from <http://search.informit.com.au/documentSummary;dn=258335755728695;res=IELENG>
- Lowe, D., Murray, S., Lindsay, E., & Liu, D. (2009). Evolving remote laboratory architectures to leverage emerging internet technologies. , 2(4), 289-294. Retrieved 2014-07-01, from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5210092
- Lowe, D. B., Berry, C., Murray, S., & Lindsay, E. (2009). Adapting a remote laboratory architecture to support collaboration and supervision. , 5, 51-56. Retrieved 2014-07-01, from http://espace.library.curtin.edu.au/cgi-bin/espace.pdf?file=/2010/03/08/file_1/133701

- Marum, M. (n.d.). *Opensocial 2.5.1 specification*. Retrieved 02/07/2014, from <https://github.com/OpenSocial/spec>
- Orduña, P. (2013). *Transitive and scalable federation model for remote laboratories* (Doctoral dissertation, Universidad de Deusto, Bilbao, Spain). Retrieved from <http://paginaspersonales.deusto.es/porduna/phd/>
- Orduña, P., Bailey, P., DeLong, K., López-de Ipiña, D., & García-Zubia, J. (2014, January). Towards federated interoperable bridges for sharing educational remote laboratories. *Computers in Human Behavior*, 30, 389–395. Retrieved from <http://www.sciencedirect.com/science/article/pii/S0747563213001416> doi: 10.1016/j.chb.2013.04.029
- Orduña, P., Irurzun, J., Rodriguez-Gil, L., Zubía, J. G., Gazzola, F., & de Ipiña, D. L. (2011). Adding new features to new and existing remote experiments through their integration in weblab-deusto. *iJOE*, 7(S2), 33-39. Retrieved from <http://dblp.uni-trier.de/db/journals/ijoe/ijoe7.html#OrdunaIRZGL11>
- Orduña, P., Lerro, F., Bailey, P., Marchisio, S., DeLong, K., Perreta, E., ... García-Zubia, J. (2013, March). Exploring complex remote laboratory ecosystems through interoperable federation chains. In *2013 IEEE global engineering education conference (EDUCON)* (pp. 1200–1208). Berlin, Germany. doi: 10.1109/EduCon.2013.6530259
- Osterwalder, A., & Pigneur, Y. (2010). *Business model generation: A handbook for visionaries, game changers, and challengers*. Wiley. Retrieved from <http://books.google.ca/books?id=fklTIInjiPQAC>
- Salzmann, C., & Gillet, D. (2008). From online experiments to smart devices. *International Journal of Online Engineering (iJOE)*, Vol 4(SPECIAL ISSUE: REV2008), 50–54. Retrieved from <http://online-journals.org/index.php/i-joe/>
- Sancristobal, E., Castro, M., Harward, J., Baley, P., DeLong, K., & Hardison, J. (2010). Integration view of web labs and learning management systems. In (p. 1409-1417). IEEE. Retrieved 2014-07-01, from <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5492363> doi: 10.1109/EDUCON.2010.5492363
- Schauer, F., Krbec, M., Beno, P., Gerza, M., Palka, L., & Spilakova, P. (2014, Feb). REMLABNET - open remote laboratory management system for e-experiments. In *Remote engineering and virtual instrumentation (REV), 2014 11th international conference on* (p. 268-273). doi: 10.1109/REV.2014.6784273
- SCORM - home. (n.d.). Retrieved 2014-07-01, from <http://scorm.com/?gclid=CLWR46iYpb8CFQIewwodsqQA2w>
- Swagger RESTful API documentation specification. (n.d.). Retrieved 30/06/2014, from <https://github.com/wordnik/swagger-spec/blob/master/versions/1.2.md>
- Taylor, B. N., & Thompson, A. (2008). *The international system of units (SI)* (NIST Special Publication No. 330). National Institute of Standards and Technology. Retrieved from <http://physics.nist.gov/Pubs/SP330/sp330.pdf>

- Tetour, Y., Boehringer, D., & Richter, T. (2011). Integration of virtual and remote experiments into undergraduate engineering courses. In (p. GOLC1-1-GOLC1-6). IEEE. Retrieved 2014-07-01, from <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6143130> doi: 10.1109/FIE.2011.6143130
- Thompson, C. W. (2005). Smart devices and soft controllers. *IEEE Internet Computing*, 9(1), 82-85. Retrieved from <http://dblp.uni-trier.de/db/journals/internet/internet9.html#Thompson05>