# Go-Lab

Global Online Science Labs for Inquiry Learning at School

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

# Specifications of the Smart Device for Remote labs

| | |
|---|---|
| Editor | Sten Govaerts (EPFL) and Christophe Salzmann (EPFL) |
| Dissemination Level | Public |
| Status | Final |

## *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*

# Table of Contents

# 1 Introduction

Within the Go-Lab project, online laboratories (referred hereafter as online labs) have been divided into three general categories:

- *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.

## *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).

Up to now, tightly coupled client-server solutions have been designed and im-

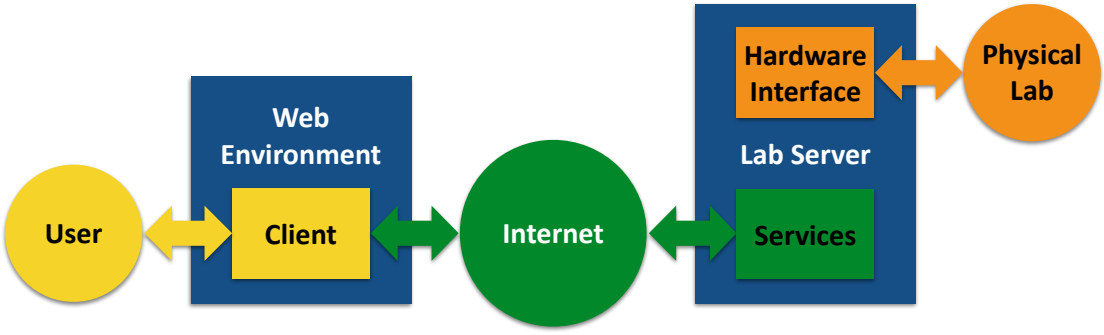


**Figure 1: Typical client-server remote lab architecture**

plemented 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.

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).

From the *lab owner perspective*, the main Go-Lab value proposition is *the pos-*

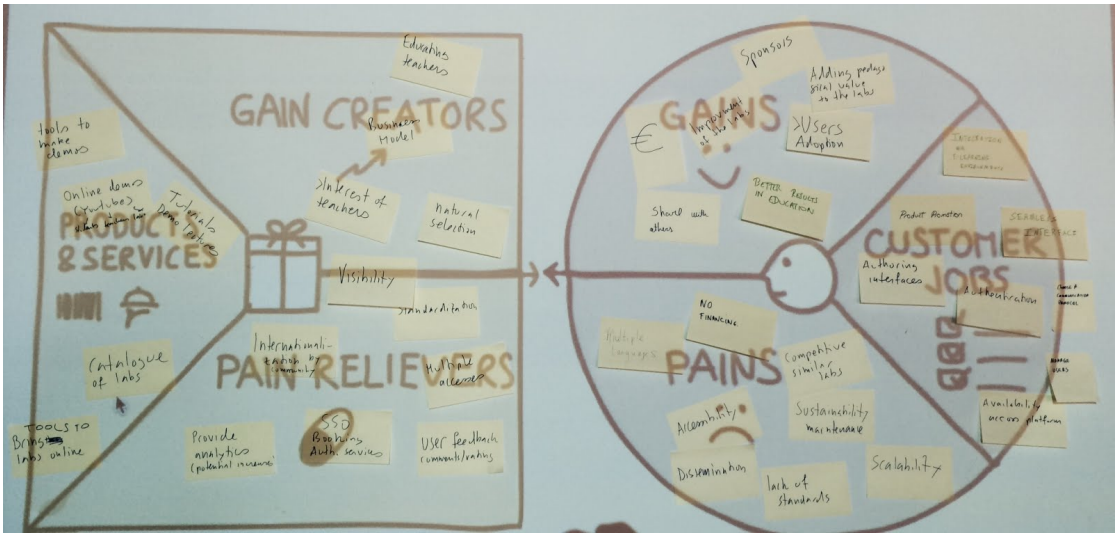


**Figure 2: The value-proposition canvas as filled by lab owners**

*sibility to combine technical resources and pedagogical structure in an ILS*, and *to make visible their own labs in the Go-Lab repository*. 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.

# 2 The Smart Device Specification

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 thighly 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 GoLab 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 **??**) 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

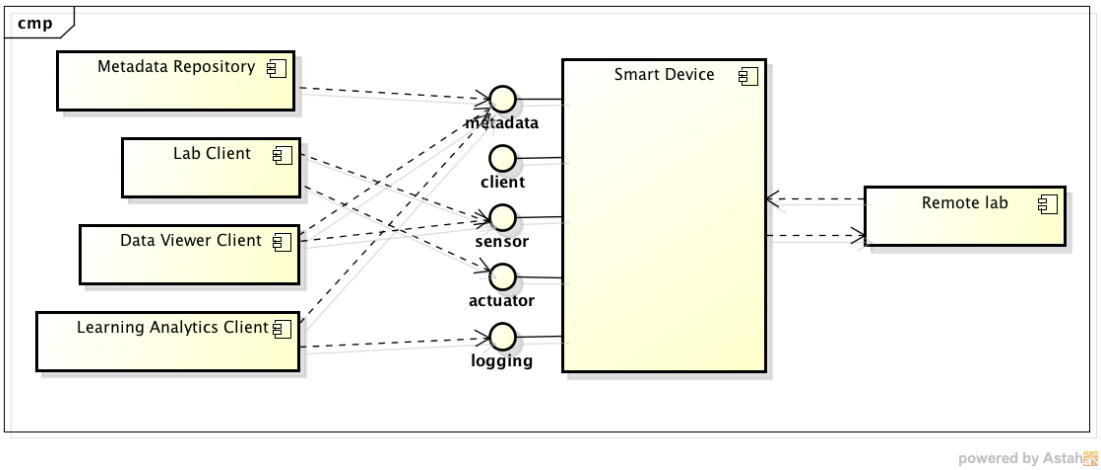


**Figure 3: UML Component diagram of different clients making use of the most common Smart Device services (arrows represent calls).**

the Go-Lab Lab Repository[1] (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

[1] Golabz, `http://www.golabz.eu`

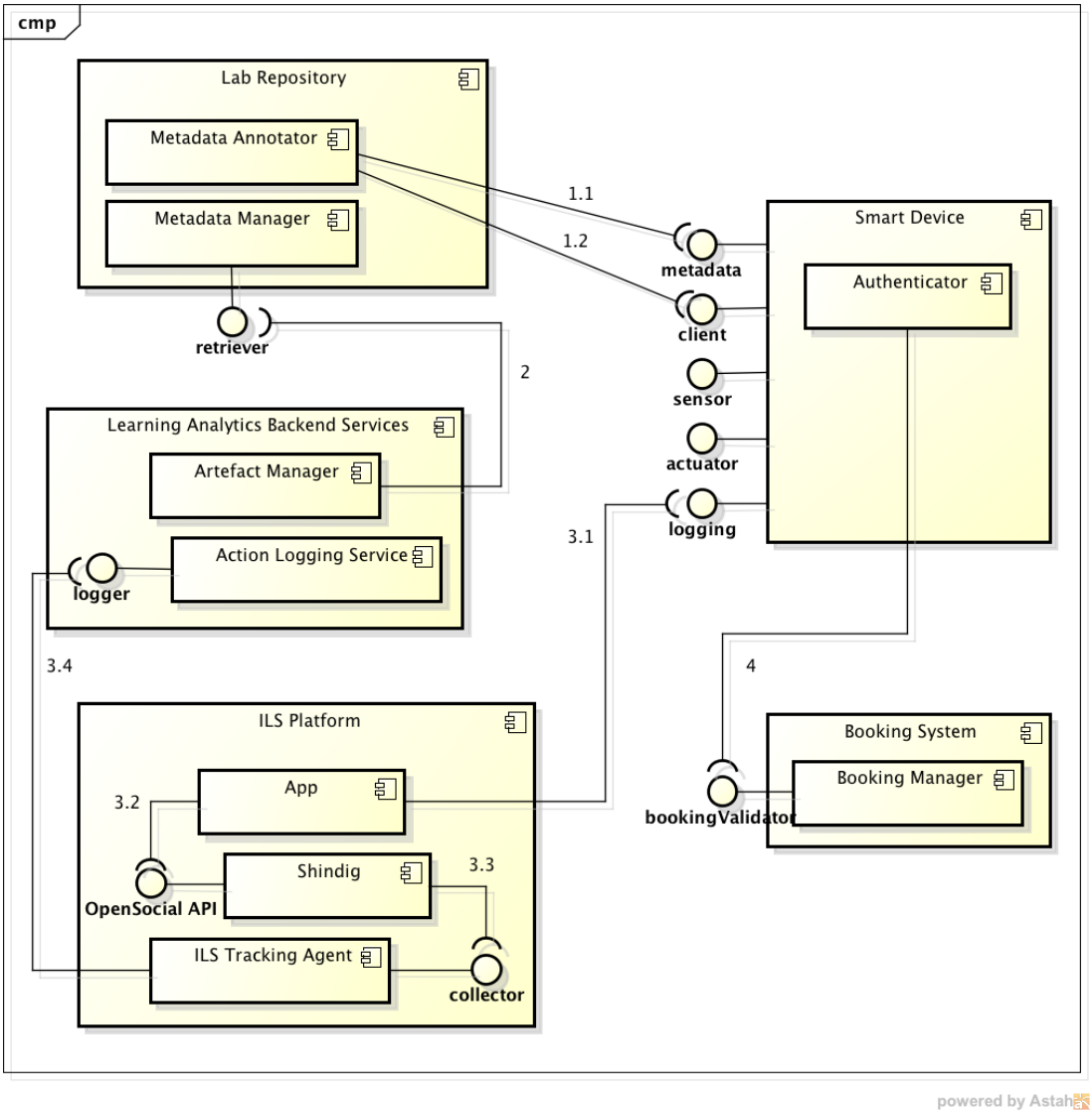


**Figure 4: UML Component diagram of the interactions between different Go-Lab services and the Smart Device.**

(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 polling[3].

WebSockets[4] 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 browsers[5]. 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

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

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

[4]Websocket specification – Wikipedia, `http://tools.ietf.org/html/rfc6455`

[5]Can 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 occured) 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 DMZ[6] 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-

[6] Demilitarized Zone (DMZ), `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 WSDL[7], 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 poor[8]. WSDL is also XML-based and in all other specifications we have opted for JSON. On request of the JavaScript community, JSON-WSP [9] 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 RSDL[10], also an XML-based language but more focused on the structure of the Web service URIs. Another option is RAML[11] that uses the YAML format and also relies on markdown for descriptions and JSON Schema[12].

[7] Web Services Description Language (WSDL) 1.1, `http://www.w3.org/TR/wsdl`

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

[9] JSON-WSP – Wikipedia, `http://en.wikipedia.org/wiki/Jsonwsp`

[10] RESTful Service Description Language (RSDL), `http://en.wikipedia.org/wiki/RSDL`

[11] RESTful API Modeling Language (RAML), `http://raml.org/`

[12] JSON Schema specification – JSON Schema: core definitions and terminology json-

For the Smart Device specification, we have opted for Swagger[13]. 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 interfaces[14], 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`

[13]Swagger website, `http://swagger.wordnik.com/`

[14]To 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",
                ...
        },
```

[15] The 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://helloreverb.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

[16]When 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/`).

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

[18]JSON 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-

fined 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 then 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 $ms^{-2}$ for the X acceleration. The set of possible units is almost infinite and each lab owner has probably his prefered 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 units[19].
- *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.

[19] SI 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**

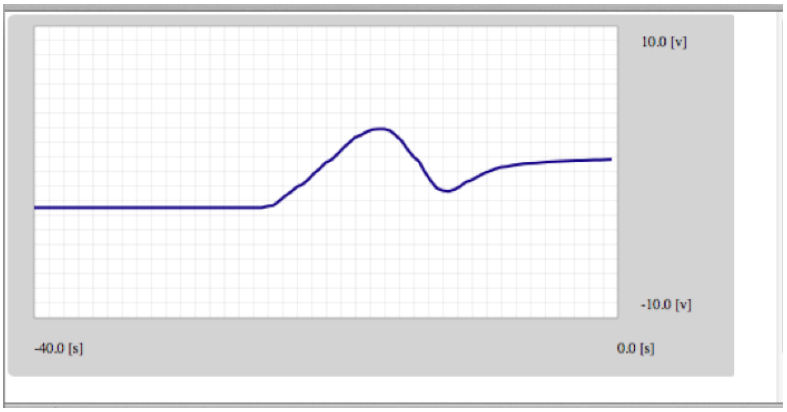


**Figure 6: The advanced app, which locally stores the temperature and displays them as a curve.**

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

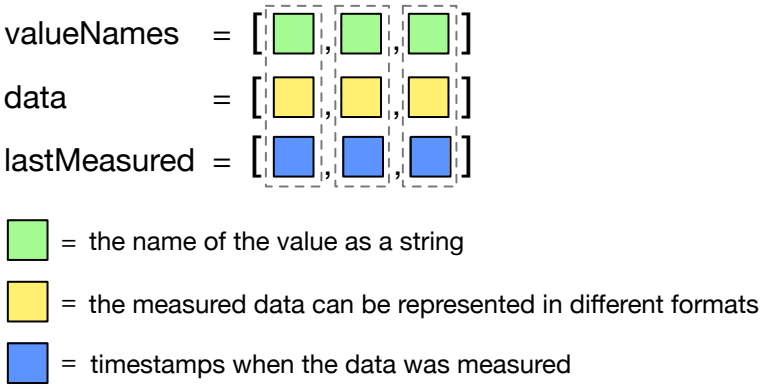


**Figure 7: Sensor and actuator data structures.**

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 smartphone 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 reponses.

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 – getClients

This required service provides links to the client applications that are provided by the lab owner to operate the lab. The service is called 'getClients'. 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 'getClients'.

**Listing 2.24: A request example to the getClients service**

```
{
    "method": "getClients"
}
```

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. VRML[20], X3D[21] & MathML[22]), 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

[20] Virtual Reality Modeling Language (VRML), http://en.wikipedia.org/wiki/VRML
[21] X3D, http://en.wikipedia.org/wiki/X3D
[22] MathML, http://www.w3.org/Math/

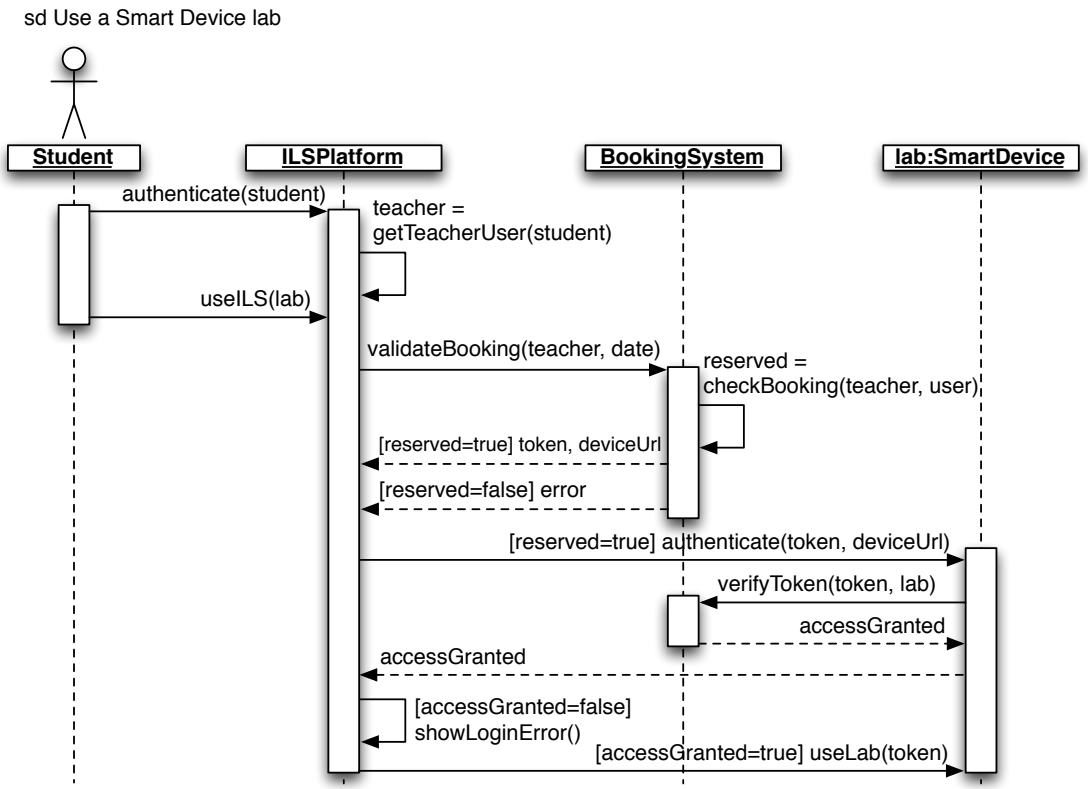


**Figure 8: UML sequence diagram of the interaction between the Go-Lab booking system and a Smart Device.**

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

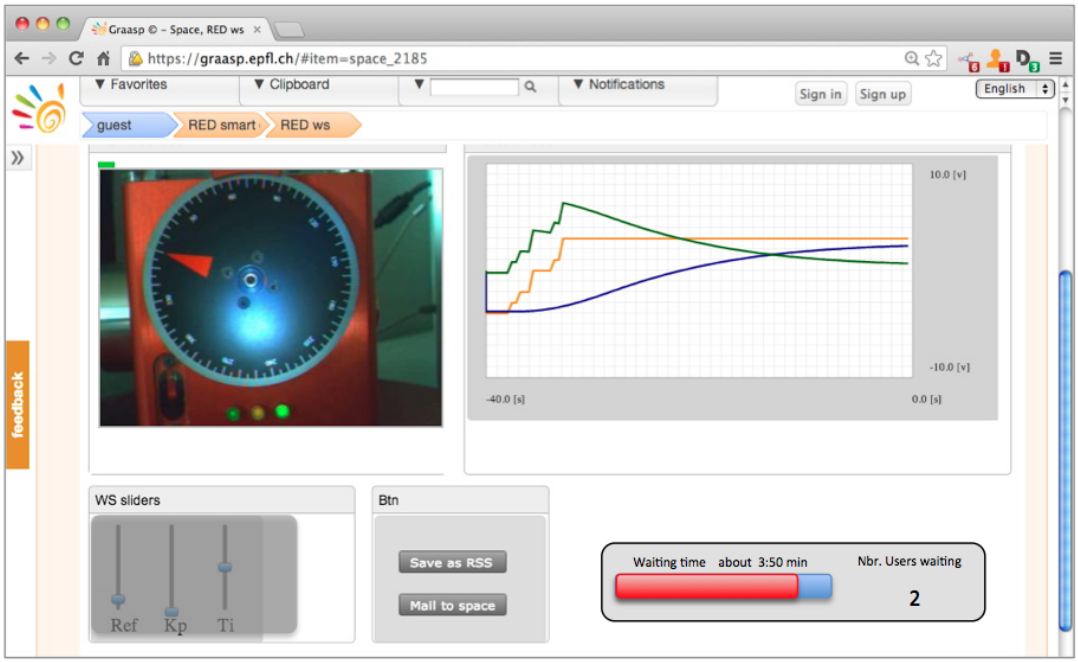


**Figure 9: UI mockup of a waiting queue visualisation.**

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.

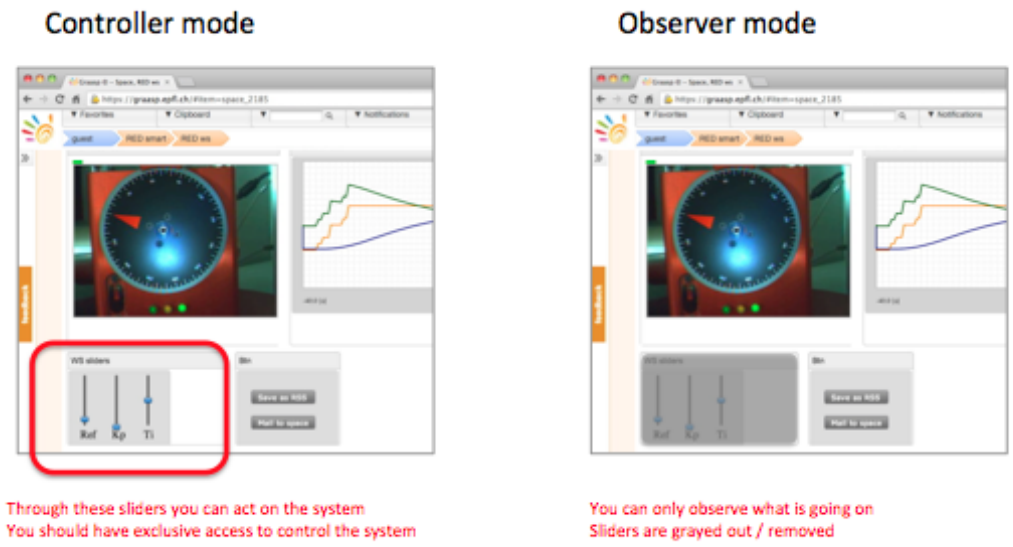


**Figure 10: UI mockups of potential client apps used with an observer and controller role.**

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.

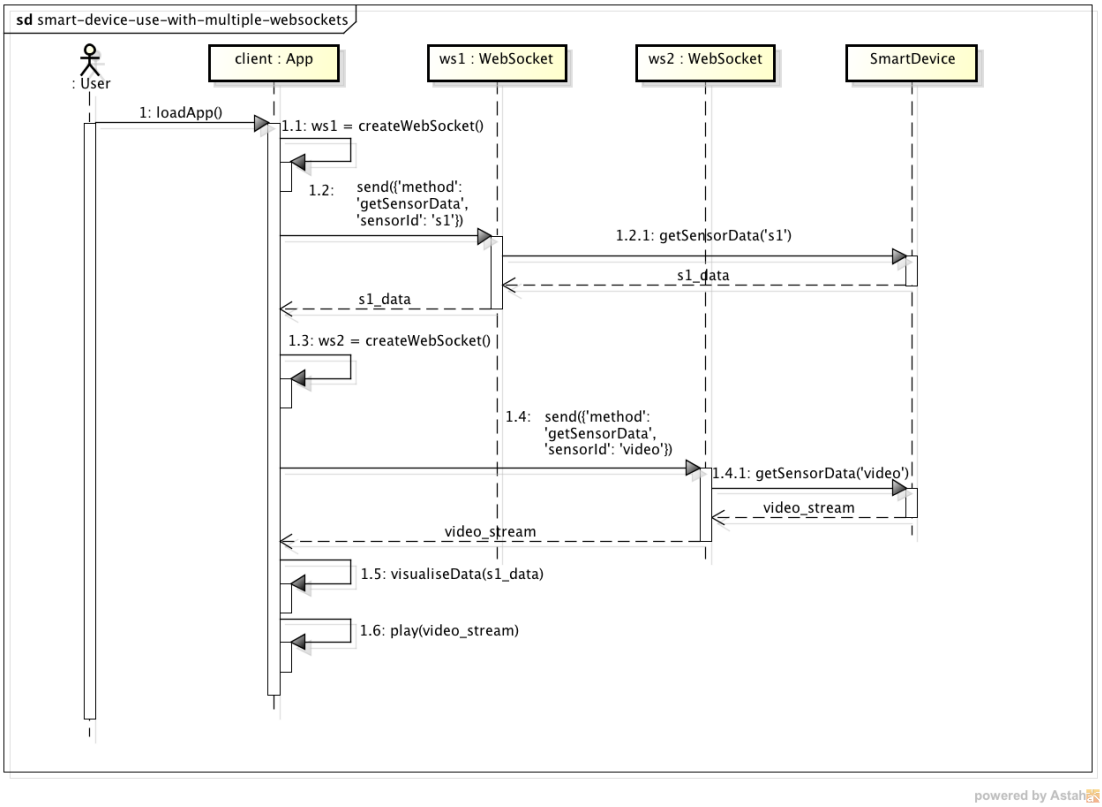


**Figure 11: UML Sequence diagram on the use of multiple WebSockets for connecting to multiple Smart Device services.**

### 2.4.3 WebSocket Channeling

There is some concern that opening many (>10) WebSockets[23] 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

[23] Microsoft 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.

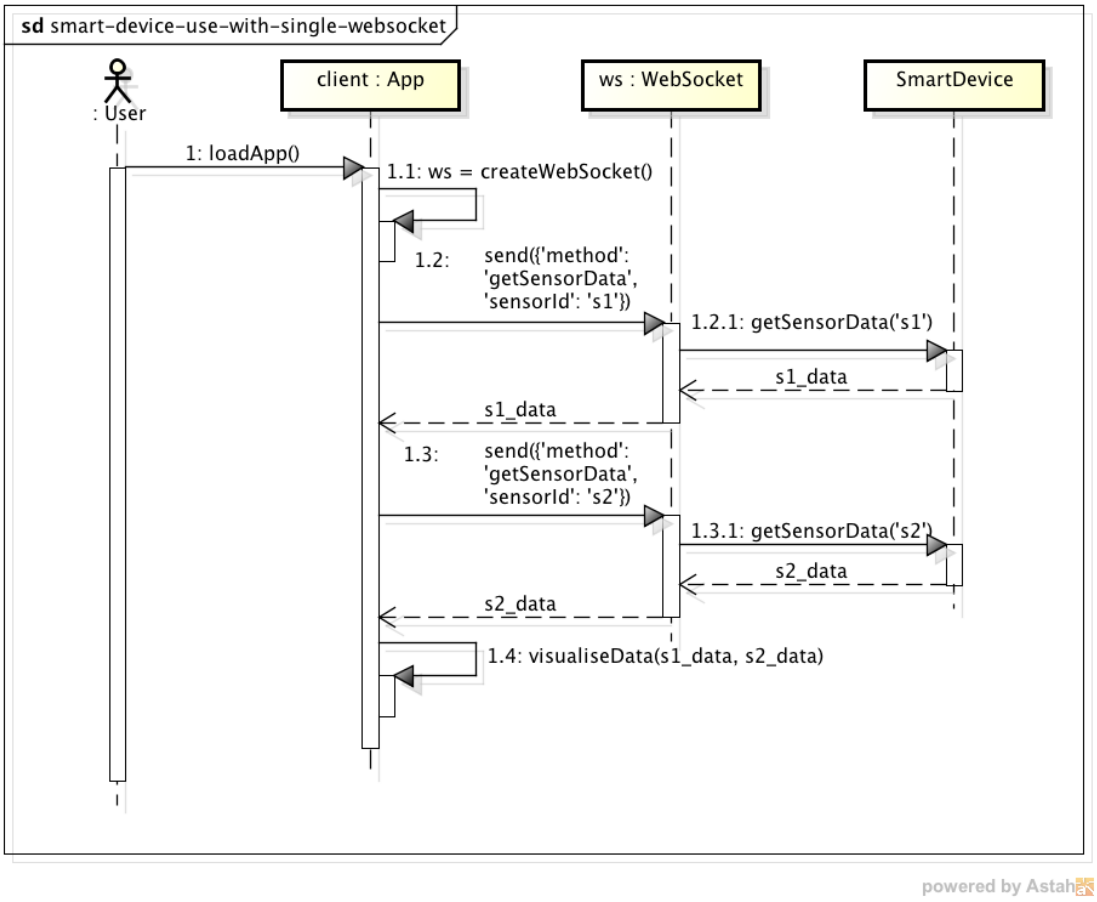


**Figure 12: UML Sequence diagram on the use of a single WebSocket for connecting to multiple Smart Device services.**

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. 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).

# 3 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.

## *3.1 Extensions for WebSockets*

The extensions related to WebSockets are limited to the API JSON object in the main `apis` field. Listing 3.1 illustrates an example service description with the WebSocket extension.

**Listing 3.1: 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.io[1] 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.

## 3.2 Extensions for Concurrency mechanisms

A root field was added to the Swagger specification, namely 'concurrency'. Listing 3.2 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 3.2: Example of a concurrency field**

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

[1] Socket.io, `http://socket.io/`

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

## 3.3 Additional Minimal Extensions

### 3.3.1 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

# 4 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.

## *4.1 RED Smart Device*

### 4.1.1 Metadata Specification

Listing 4.1 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 4.1: 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."
                }
            }
        }
    }
}
```

### 4.1.2 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 4.2 and 4.3 illustrate respectively a request and response to the getClients service. Note, the URLs in the getClients response can differ.

**Listing 4.2: Request to get all client apps**

```
{
    "method": "getClients"
}
```

**Listing 4.3: Response to get all client apps**

```
{
    "method": "getClients",
    "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 4.4 and 4.5 provide the request and response to get sensor metadata. Listing 4.5 shows the sensor metadata itself.

**Listing 4.4: Request to get the sensor metadata**

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

**Listing 4.5: 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 4.6 and 4.7 show two requests for sensor data of the position and video sensor using the optional 'accessRole' field, which could be dropped here.

**Listing 4.6: The request to get data for the 'position' sensor**

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

**Listing 4.7: The request to get data for the 'video' sensor**

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

Listing 4.8 provides an example response for the position sensor.

**Listing 4.8: 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 4.9 illustrates how the continuous stream of measurements can be stopped, by setting the 'updateFrequency' field to 0.

**Listing 4.9: The request to interrupt the sensor measurement data flow**

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

### *getActuatorMetadata*

Listing 4.10 and 4.11 show how the actuator metadata can be retrieved and the later presents the actuator metadata of the RED Lab Smart Device.

**Listing 4.10: The request to get actuator metadata**

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

**Listing 4.11: 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 4.12 and 4.13 illustrated how an actuator can be set.

**Listing 4.12: The request to set the reference actuator**

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

**Listing 4.13: 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]
    }
}
```

## *4.2 Running example Smart Device*

### 4.2.1 Metadata Specification

Listing 4.14 presents the metadata for the running example Smart Device used throughout this deliverable.

**Listing 4.14: 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/"
                    }
                }
            }
        }
    }
}
```

### 4.2.2 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.

***getClients***

**Listing 4.15: The request message to retrieve the client apps.**

```
{
    "method": "getClients"
}
```

**Listing 4.16: The response message to retrieve the client apps.**

```
{
    "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"
        }
    ]
}
```

***getSensorMetadata***

**Listing 4.17: The request message to retrieve the sensor metadata.**

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

**Listing 4.18: 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 4.19: 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 4.20: 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 4.21: 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 4.22: 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 4.23: 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 4.24: The request message to retrieve the actuator metadata.**

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

**Listing 4.25: 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 4.26: The request message to set the motor actuator.**

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

**Listing 4.27: 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 4.28: 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 4.29: The request message to retrieve ActivityStreams logging information.**

```
{
```

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

**Listing 4.30: 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/"
            }
        }
    ]
}
```

## *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

Cascado, D., Sevillano, J. L., Fernández-Luque, L., Johan-GrÃ¸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

*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

Marum, M. (n.d.). *Opensocial 2.5.1 specification.* Retrieved 02/07/2014, from https://github.com/OpenSocial/spec

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/

*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

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