getLightSensors HTTP/JSON

Interface Design Description

Abstract
This document describes an HTTP protocol and JSON payload encoding variant of the ”getLightSensors” service.

Contents

1 Overview **3**

2 Service Operations **4**

3 Service Interface **4**
3.1 operation getLightSensors 4

4 Data Models **5**
4.1 struct LampResponseDTO 5
4.2 struct Metadata 5
4.3 Primitives 5

5 References **7**

6 Revision History **8**
6.1 Amendments 8
6.2 Quality Assurance 8

1 Overview

This document describes the ”getLightSensors” service interface, which provides and receives data from the sensors to the Controller when this one send one of its periodic requests. This request is done using HTTP and the response is sent as a JSON value.

This document provides the Interface Design Description IDD to the *”getLightSensors – Service Description* document. For further details about how this service is meant to be used, please consult that document.

The rest of this document describes how to realize the ”getLightSensors” service interface in detail. Both in terms of its operations (Section 2) and its information model (Section 4).

2 Service Operations

The interfaces of the ”getLightSensors” service, its operations, data models and implementation are provided below.

In particular, each subsection first names the HTTP protocol method and path used to call the function, after which it names an abstract function from the getLightSensors SD document, as well as input and output types.

All operations in this section respond with the status code `201 Created` if called successfully. The error codes are, `400 Bad Request` if request is malformed, `401 Unauthorized` if improper client side certificate is provided, `500 Internal Server Error` if Service Registry is unavailable.

3 Service Interface

This section lists the functions that must be exposed by getLightSensors services in alphabetical order. Each subsection first names the HTTP method and path used to call the function, after which it names an abstract function from the SD getLightSensors document, as well as any input and output types. All functions are assumed to respond with the HTTP status code `204 No Content` if called successfully, as none of them respond with any data.

3.1 GET /Light-sensor

Operation: getLightSensors
Output: sensorList

```
GET /Light-sensor HTTP/1.1

{
[ {
  "id" : 1,
  "value" : "976.7"
}, {
  "id" : 2,
  "value" : "1000"
}, {
  "id" : 3,
  "value" : "1000"
},

...

{
  "id" : N,
  "value" : "1000"
}]
}
```

Listing 1: An getLightSensors invocation for a system with N sensors.

4 Data Models

Here, all data objects that can be part of the service calls associated with this service are listed in alphabetic order. Note that each subsection, which describes one type of object, begins with the *struct* keyword, which is meant to denote a JSON Object that must contain certain fields, or names, with values conforming to explicitly named types. As a complement to the primary types defined in this section, there is also a list of secondary types in Section 4.3, which are used to represent things like hashes, identifiers and texts.

4.1 struct LampResponseDTO

This structure is used to get the sensor data.

Object Field	Value Type	Description
”id”	String	Sensor ID.
”value”	String	Luminosity measured (lux).

4.2 struct Metadata

A JSON Object which maps String key-value pairs.

4.3 Primitives

As all messages are encoded using the JSON format [1], the following primitive constructs, part of that standard, become available. Note that the official standard is defined in terms of parsing rules, while this list only concerns syntactic information. Furthermore, the Object and Array types are given optional generic type parameters, which are used in this document to signify when pair values or elements are expected to conform to certain types.

JSON Type	Description
Value	Any out of Object, Array, String, Number, Boolean or Null.
Object <A>	An unordered collection of [String: Value] pairs, where each Value conforms to type A.
Array <A>	An ordered collection of Value elements, where each element conforms to type A.
String	An arbitrary UTF-8 string.
Number	Any IEEE 754 binary64 floating point number [2], except for *+Inf*, *-Inf* and *NaN*.
Boolean	One out of `true` or `false`.
Null	Must be `null`.

With these primitives now available, we proceed to define all the types specified in the Service Discovery Register SD document without a direct equivalent among the JSON types. Concretely, we define the Service Discovery Register SD primitives either as *aliases* or *structs*. An *alias* is a renaming of an existing type, but with some further details about how it is intended to be used. Structs are described in the beginning of the parent section. The types are listed by name in alphabetical order.

4.3.1 alias DateTime = String

Pinpoints a moment in time in the format of ”YYYY-MM-DD HH:mm:ss”, where ”YYYY” denotes year (4 digits), ”MM” denotes month starting from 01, ”DD” denotes day starting from 01, ”HH” denotes hour in the 24-hour format (00-23), ”MM” denotes minute (00-59), ”SS” denotes second (00-59). ” ” is used as separator between the date and the time. An example of a valid date/time string is ”2020-12-05 12:00:00”

4.3.2 alias id = Number

An identifier generated for each Object that enables to distinguish them and later to refer to a specific Object.

4.3.3 alias Interface = String

A String that describes an interface in *Protocol-SecurityType-MimeType* format. *SecurityType* can be SECURE or INSECURE. *Protocol* and *MimeType* can be anything. An example of a valid interface is: ”HTTPS-SECURE-JSON” or ”HTTP-INSECURE-SENML”.

4.3.4 alias Name = String

A String that is meant to be short (less than a few tens of characters) and both human and machine-readable.

4.3.5 alias SecureType = String

A String that describes an the security type. Possible values are *NOT_SECURE* or *CERTIFICATE* or *TOKEN*.

4.3.6 alias URI = String

A String that represents the URL subpath where the offered service is reachable, starting with a slash (”/”). An example of a valid URI is ”/temperature”.

4.3.7 alias Version = Number

A Number that represents the version of the service. And example of a valid version is: 1.

5 References

[1] T. Bray, “The JavaScript Object Notation (JSON) Data Interchange Format,” RFC 7159, 2014, RFC Editor. [Online]. Available: https://doi.org/10.17487/RFC7159

[2] M. Cowlishaw, “IEEE Standard for Floating-Point Arithmetic,” *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, July 2019. [Online]. Available: https://doi.org/10.1109/IEEESTD.2019.8766229

6 Revision History

6.1 Amendments

No.	Date	Version	Subject of Amendments	Author
1	2020-12-05	4.4.1		Szvetlin Tanyi
2	2021-07-14	4.4.1	Minor edits	Jerker Delsing
3	2022-01-17	4.4.1	Minor edits	Jerker Delsing

6.2 Quality Assurance

No.	Date	Version	Approved by
1	2022-01-17	4.4.1	Jerker Delsing