Light Service HTTP/TLS/JSON

Interface Design Description

Abstract
This document describes an HTTP protocol with TLS payload security and JSON payload encoding variant of the **Light** service.

Contents

1 Overview

This document describes the **Light** service interface, which enables basic control of light emission within the lamp system. The primary purpose of this service is to switch the light on or off. The interface focuses on binary control, with no adjustments for intensity or color, ensuring reliable and straightforward operation.

It's implemented using protocol, encoding as stated in below table:

Profile type	Type	Version
Transfer protocol	HTTP	2
Data encryption	TLS	1.3
Encoding	e.g. JSON	RFC 8259 [1]
Compression	N/A	-

Table 1: Communication and semantics details used for the **Light** service interface

This document provides the Interface Design Description IDD to the *[Light] – 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 **Light** service HTTP/TLS/JSON interface in detail. Both in terms of its operations (Section 2) and its information model (Section 3).

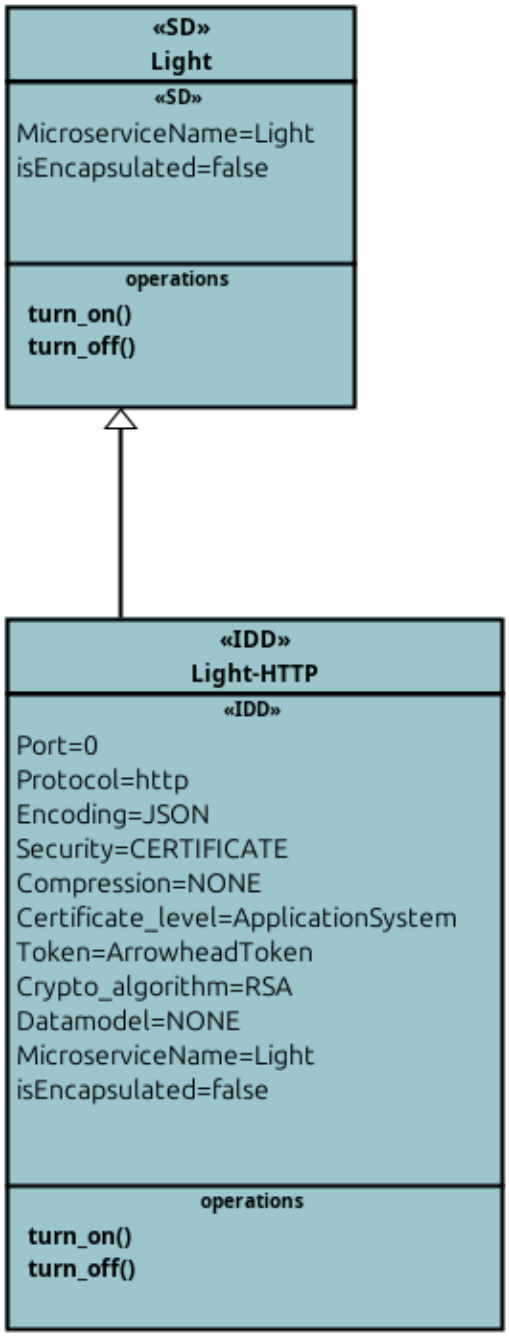

Figure 1: SysML model of the Light interface, its operations, data models and implementation.

2 Service Operations

The interfaces of the Light service, its operations, data models and implementation are provided below. An SysML service overview is found in Figure 1.

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 Light SD document, as well as input and output types.

All operations in this section respond with the status code `200 Ok` 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.

2.1 POST /light/turn_on

Operation: turn on
Input: LightstateForm
Output: LightstateResult

Called to turn on the light emission, as exemplified in Listing 1.

```
POST /light/turn_on HTTP/2

{
    "version": 1,
    "lightEmission": true
}
```

Listing 1: A turn on invocation.

```
{
 "version": 1,
 "emssionOld": boolean,
 "emissionNew": boolean
}
```

Listing 2: A turn on response.

2.2 POST /light/turn_off

Operation: turn off
Input: LightstateForm
Output: LightstateResult

Called to turn off the light emission, as exemplified in Listing 3.

```
POST /light/turn_off HTTP/2

{
    "version": 1,
    "lightEmission": false
}
```

Listing 3: A turn off invocation.

```
{
 "version": 1,
 "emssionOld": boolean,
 "emissionNew": boolean
}
```

Listing 4: A turn off response.

3 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 3.3, which are used to represent things like hashes, identifiers, and texts.

3.1 struct LightstateForm

This structure is used to represent a light state.

Object Field	Value Type	Description
“lightEmission”	Boolean	State of light emission. Default false

3.2 struct LightstateResult

This structure is used to give information about the previous and the new state of the light.

Object Field	Value Type	Description
“emissionOld”	Boolean	State of light emission before change.
“emissionNew”	Boolean	State of light emission after change.

3.3 Primitives

As all messages are encoded using the JSON format [2], 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 [3], 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 Light SD document without a direct equivalent among the JSON types. Concretely, we define the Light 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.

3.3.1 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”

3.3.2 alias id = Number

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

3.3.3 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”.

3.3.4 alias Name = String

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

3.3.5 alias SecureType = String

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

3.3.6 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”.

3.3.7 alias Version = Number

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

4 References

[1] T. Bray, “The JavaScript Object Notation (JSON) Data Interchange Format,” RFC 8259, Dec. 2017. [Online]. Available: https://rfc-editor.org/rfc/rfc8259.txt

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

[3] 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

5 Revision History

5.1 Amendments

No.	Date	Version	Subject of Amendments	Author
1	2024-10-12	4.6.2		Max Lütkemeyer

5.2 Quality Assurance

No.	Date	Version	Approved by
1	2024-10-12	4.6.2	