everpl Documentation

Release 0.3.draft

Sergey Kostyuk

May 04, 2018

User Documentation

1 Getting Started **3**

2 Installation **5**
- 2.1 Preface . . . 5
- 2.2 System Requirements . . . 5
- 2.3 Automatic Installation Steps . . . 6
- 2.4 Manual Installation Steps . . . 6

3 First Run **7**

4 Integrations **9**

5 Client Applications **11**

6 Local network discovery **13**
- 6.1 General information . . . 13
- 6.2 How to discover an everpl hub . . . 14

7 REST API **15**
- 7.1 General information . . . 15
- 7.2 Protected resources . . . 15
- 7.3 Authentication . . . 16
- 7.4 Things . . . 16
- 7.5 Placements . . . 19

8 Handling Errors **21**
- 8.1 Error Response Format . . . 21
- 8.2 General . . . 22
- 8.3 Authorization and authentication . . . 23
- 8.4 Things . . . 25
- 8.5 Placements . . . 26

9 Streaming API **27**

10 Capabilities **29**

11 Possible Capabilities **31**
- 11.1 Actuators . . . 31

11.2 Has State . . . 31
11.3 Is Active . . . 32
11.4 On/Off . . . 32
11.5 Open/Closed . . . 33
11.6 Multimode . . . 34
11.7 Has Brightness . . . 34
11.8 Has Color HSB . . . 35
11.9 Has Color RGB . . . 35
11.10 Has Value . . . 36
11.11 Has Volume . . . 36

12 Indices and tables **37**

Everthing Platform is an open-source IoT-enabled automation platform. It allows to operate different types of devices, set-up automation rules (if-this-than-that), store and process a history of events and all of that autonomously from cloud services and Internet connection (if you want it).

Documentation on Everthing Platform is slitted into a couple of sections:

- *User Documentation*
- *External API Documentation*
- *Developer Documentation*

Platform itself is hosted on GitHub: https://github.com/s-kostyuk/adpl/

CHAPTER 1

Getting Started

Actually, to use Everthing Platform you will need to have two main components installed:

- the platform itself;
- and some client applications.

Platform is just an application that directly controls every object in your system: devices, other applications, their interconnection and interaction. It is in charge of setting-up and connection of all components, reading of their states and sending commands. And provides all its functions to client applications.

Client application is a some software that makes interaction with the platform and end-user itself possible. It is connected to the platform and allows to use all its features via some user-friendly interface.

If you are a developer, you can develop own client application based on API section of this documentation. Otherwise, you can choose one on the *Client Applications* page.

In the next chapters you will find how to install[1] Everthing Platform and how to run it[2] for the first time. Just click 'next' button to continue.

[1] Documentation page: *Installation*
[2] Documentation page: *First Run*

CHAPTER 2

Installation

2.1 Preface

As was mentioned[1], you need two pieces of software to use the platform:

- the platform itself;
- and some client application.

This tutorial is mostly related to the platform itself. For details about the installation and usage of client applications, please visit the *Client Applications* page.

2.2 System Requirements

Minimum System Requirements:

- Python 3.5[2]
- bash

Recommended System Requirements:

- Python 3.5 or newer
- UNIX-like operating system (like macOS and Linux-based systems)
- hardware support of protocols like Bluetooth, ZigBee and so on for different *Integrations*

[1] Documentation page: *Getting Started*

[2] async/await expressions which are commonly used in the platform was introduced only in Python 3.5. In a case if you need a support of older versions of python - please, endorse this issue: #22.

2.3 Automatic Installation Steps

1. Download an archive with the latest stable release of platform from its repository: https://github.com/s-kostyuk/adpl/releases

Note: You can also download the latest development (unstable) version here: https://github.com/s-kostyuk/adpl by clicking a 'clone or download' button.

2. Extract archive content to some directory. Remember its placement (path).
3. Open terminal emulator, switch to the everpl's project directory:

```
cd /path/to/everpls/directory
```

4. Install an everpl package using pip:

```
pip3 install .
```

5. Now it's possible to run everpl application by simply calling an `everpl` command:

```
everpl
```

Installation finished!

Note: You can also install everpl package in the "Development Mode". Why you may need it and what with mode provides is described by the following link:[4]

2.4 Manual Installation Steps

1. Download an archive with the latest stable release of platform from its repository: https://github.com/s-kostyuk/adpl/releases
2. Extract archive content to some directory. Remember its placement (path).
3. Open terminal emulator, switch to the platform's directory:

```
cd /path/to/platforms/directory
```

4. Install all needed dependencies that are listed in `requirements.txt`[3] file. The most simple way to do this is to use pip:

```
pip3 install -r requirements.txt
```

5. Now it's possible to run the main execution file:

```
bash ./dpl/run.sh
```

Installation finished!

[4] Information about "Development Mode" of package installation process: https://packaging.python.org/tutorials/distributing-packages/#working-in-development-mode

[3] Requirements file is placed in the root of platform's directory, for example: https://github.com/s-kostyuk/adpl/blob/devel/requirements.txt

CHAPTER 3

First Run

CHAPTER 4

Integrations

CHAPTER 5

Client Applications

For now there are only two official cleint applications for everpl: an Android and single-page web application.

An Android client is an open-source application located at https://github.com/dot-cat/creative_assistant_android. It's supported by the project author and is developed carefully with attention to software architecture, libraries and used software development approaches.

A web-client is much less production-ready. The only task it was created for is to test and demonstrate the newest features of everpl. Therefore it's quite unstable and much less elaborate. Frankly speaking, web-client was originally started as a laboratory work :). The client is hosted at https://srgk.gitlab.io/test-bootstrap2/. Its source code is available at https://gitlab.com/srgk/test-bootstrap2

CHAPTER 6

Local network discovery

6.1 General information

Starting from v0.3 of the platform all everpl instances (everpl hubs) are able to be discovered in a local network by default.

Hubs announce their presence and can be discovered using a Zeroconf (Avahi/Bonjour) protocol - Zero Configuration Networking protocol. This protocol allows services to announce their presence in the system, to assign constant domain names in a ".local" domain zone, to resolve such domain names and to look for a specific service in the system.

For more information about Zeroconf you can read an article on Medium titled "Bonjour Android, it's Zeroconf". It tells about Zeroconf protocols in general, about Bonjour/Avahi approach and how it relates with client applications and service discovery.

Unfortunately, Zeroconf (and UDP multicast in general) isn't supported by modern web browsers.

For more detailed information see:

- DNS-SD protocol website: http://www.dns-sd.org/, covers service discovery part of functionality;
- mDNS protocol website: http://www.multicastdns.org/, covers domain name association in ".local" domain zone;
- and corresponding RFCs.

For testing purposes you can use such handy tools as:

- Avahi-Discover GUI utility for Linux: https://linux.die.net/man/1/avahi-discover
- Service browser for Android: https://play.google.com/store/apps/details?id=com.druk.servicebrowser&hl=en_US
- dns-sd CLI tool for macOS

6.2 How to discover an everpl hub

In order to discover an everpl hub you need to use one of the Zeroconf libraries (like build-in NSD for Android) and search for a service type `_everpl._tcp`. By default such devices will have a name defined as "everpl hub @ hostname". To access an everpl REST API on a device you can use name and port, defined in Hostname (Server) and Port fields of a discovery response correspondingly.

Here is an example of a complete discovery response (as displayed by console avahi-browse utility):

```
= virbr0 IPv4 everpl hub @ hostname_was_here              _everpl._tcp         local
   hostname = [hostname_was_here.local]
   address = [192.168.20.1]
   port = [10800]
   txt = []
```

CHAPTER 7

REST API

7.1 General information

REST API is the base external API that is provided by platform. It is recommended to use with unstable network connections, for getting of access tokens and for occasional updates of resource statuses. For receiving of instant notifications on resource updates please take a look in *Streaming API* section of documentation.

In this documentation you will also find such value as `BASE_URL`. The `BASE_URL` is a value that points to the base URL of REST API. It consists of protocol specification (http or https), hostname or an IP address of platform instance, port and the rest of REST API path. Keep in mind that the hostname and port of platform instance can be changed in various circumstances (like ip address renewal, moving between different networks and so on).

The `BASE_URL` may look like this: `http://localhost:10800/api/v1/`

or like this: `https://hostname.local/api/v1/`

7.2 Protected resources

There are two types of API resources in the platform:

- protected;
- and unprotected.

Protected resources are resources that can be viewed or modified only by an authorized user. Unprotected resources are resources that can be accessed by any user, including anonymous users.

To access protected resources you will need to authenticate and obtain a special access token[1]. Then this token must to be passed in `Authorization` HTTP header on each request to protected resource.

The process of obtaining of access token is described in *Authentication* section. Related error responses are described in *Handling Errors* section of documentation. Possible errors: 2100, 2101, 2110.

[1] See also: Access token definition in OAuth specs

7.3 Authentication

As was mentioned in the previous section, you need to obtain an access token to read or modify protected resources (which are the majority of resources). An access token itself is a unique secret alphanumeric string that is specific exactly to one user on exactly one client application instance. As a usual username-password combination it allows to uniquely identify the user and to perform all operations on his or her behalf. So threat it with care and store securely.

To retrieve an access token you need to send user credentials on `/auth` endpoint in POST request.

URL structure `BASE_URL/auth`

Method `POST`

Headers

Content-Type `application/json`

Request Body

```
{
    "username": "your_username_here",
    "password": "your_password_here"
}
```

In a case of success you will get the similar response:

Status Code 200

Headers

Content-Type `application/json`

Response Body

```
{
    "message": "authorized",
    "token": "90ff4ba085545c1735ab6c29a916f9cb8c0b7222"
}
```

In a case of authentication error you will receive one of the responses listed in *Handling Errors* section of documentation. Possible errors: 1000, 1001, 1003, 2000, 2001, 2002.

7.4 Things

Thing is a sort of basic concept in platform. Thing represent some item of the system, i.e. some physical device or software application.

7.4.1 Thing object

General thing object has the following structure:

commands A list of commands that can be sent to this Thing

is_active A boolean field that indicates if this Thing is in one of the 'active' states (like 'playing' for player or 'on' for lighting).

is_available A boolean field that indicates if this Thing is available for communication (like fetching data, updating Things state and sending commands).

last_updated A floating-point value, UNIX time that indicates the time of latest update (of state field or any other field)

state A string, indicates the current state of Thing (type-specific). For example, for lighting it can take on the following values: 'on', 'off' and 'unknown'.

friendly_name Some user-friendly name of this particular thing that can be modified and directly displayed to user.

type Some type-related information. Its format is still unstable.

id A string (for now), some machine-friendly unique identifier of specific thing.

placement A string (for now), an identifier of placement where this Thing is currently placed (positioned). See *Placements* section for detailed information about placements.

The exact set of fields and their values may vary for different types of things. For detailed information, please refer to the FIXME section of documentation.

Example of Thing object:

```
{
    "commands": [
        "activate",
        "deactivate",
        "toggle",
        "on",
        "off"
    ],
    "is_active": false,
    "is_available": true,
    "last_updated": 1505768807.4725718,
    "state": "unknown",
    "friendly_name": "Kitchen cooker hood",
    "type": "switch",
    "id": "F1",
    "placement": "R2"
}
```

7.4.2 Fetching all Things

To fetch all Things, you need to perform the following request:

URL structure `BASE_URL/things/`

Parameters

placement Enables filtering of things by placement. Use it like `?placement=R1` to get a list of things positioned in `R1` placement.

type Enables filtering of things by their type. Use it like `?type=lighting` to get a list of things that have a type of `lighting`.

Method `GET`

Headers

Authorization `your_auth_token_here`

An example of response body is placed here: https://git.io/v5xz3.

7.4.3 Fetching specific Thing

To fetch a specific Thing, you need to perform the following request:

URL structure `BASE_URL/things/{id}`

Method `GET`

Headers

Authorization `your_auth_token_here`

Notes Replace `{id}` part of the URL with an identifier of requested Thing object.

7.4.4 Sending commands to a Thing

Starting from the v0.3 of everpl it's possible to send commands to the Actuators - to the Things that are able to execute some commands.

Each command can have its own set of arguments, the list of the allowed commands is specified in the `commands` field for each Actuator Thing. The list of available commands and their set of possible arguments is determined by the list of capabilities implemented by the specified Thing.

To send a command to an Actuator Thing you need to send a POST request using an `/execute` sub-resource of a Thing in question:

URL structure `BASE_URL/things/{id}/execute`

Method `POST`

Headers

Authorization `your_auth_token_here`

Content-Type `application/json`

Request Body

```
{
        "command": "the_name_of_the_command",
        "command_args": {}
}
```

Notes Replace `{id}` part of the URL with an identifier of requested Thing object.

The presence of the both `command` and `command_args` fields is mandatory.

The value of the `command` field must to be a string - the name of the command to be executed; this value is must to be an element from the `commands` field of the specified Thing.

The value of the `command_args` field must to be a dictionary of keyword- arguments for the command with keys as strings and values as specified in the Thing's documentation. It's allowed to pass an empty dictionary as the value of the `command_args` field if there is no additional arguments needed for an execution of the specified command.

In a case of success your command will be send on execution and you will get a similar response:

Status Code 202

Headers

Content-Type `application/json`

Response Body

```
{
        "message": "accepted"
}
```

In a case of an pre-execution (validation) error you will receive one of the responses listed in *Handling Errors* section of documentation. Possible errors: 1000, 1001, 1003, 1005, 2100, 2101, 2110, 3100, 3101, 3102, 3103, 3110.

7.5 Placements

Placement is a some static position in a building / city / other area. In homes it usually corresponds to one room.

7.5.1 Placement object

Placement object has the following structure:

id A string (for now), some machine-friendly unique identifier of specific thing.

friendly_name Some user-friendly name of this particular placement that can be modified and directly displayed to user.

image_url A URL to related picture of this placement (room).

Example of Placement object:

```
{
    "id": "R1",
    "friendly_name": "Corridor",
    "image_url": "http://www.gesundheittipps.net/wp-content/uploads/2016/02/Flur_547-
↪1024x610.jpg"
}
```

7.5.2 Fetching all Placements

To fetch all Placements, you need to perform the following request:

URL structure `BASE_URL/placements/`

Method `GET`

Headers

Authorization `your_auth_token_here`

An example of response body is placed here: https://git.io/v5x6S.

7.5.3 Fetching specific Placement

To fetch a specific Placement, you need to perform the following request:

URL structure `BASE_URL/placements/{id}`

Method `GET`

Headers

Authorization `your_auth_token_here`

Notes Replace `{id}` part of the URL with an identifier of requested Placement object.

CHAPTER 8

Handling Errors

Unfortunately, always there is something that could go wrong while processing of API requests. Connection can be lost, token can be expired, some exception can be unhandled and so on. Stuff happens. And you must be ready to that.

Here is the complete list of responses for different types of API errors. Errors are grouped by main platform's subsystems and each error type has its own identifier.

8.1 Error Response Format

If some request resulted in an error, than platform instance returns a response with HTTP status code not less than 400 and JSON-encoded body with an additional information about an error.

A format of request body is the following:

```
{
  "error_id": "int, an identifier of an error",
  "devel_message": "Some message for developers",
  "user_message": "Some message that can be directly displayed to the user",
  "docs_url": "A link to the related section in platform's documentation"
}
```

Regarding HTTP status codes:

- codes starting from 400 are error codes;
- codes >= 400 and < 500 indicate client-side errors;
- codes >= 500 indicate server-side errors.

8.2 General

8.2.1 Error 1000: Unsupported content-type

This error can be thrown on POST requests. It may indicate that:

- a client application forgot to set `Content-Type` request header;
- or `Content-Type` header value points to unsupported type of content.

This error indicates some issue with the client-side code and should be fixed by client's developer.

For now only one type of request content is supported and can be read: `application/json`. In future additional content-types may be supported like `application/xml`. Extra information about content-types in general can be found on Wikipedia and MDN.

HTTP status code: 400.

8.2.2 Error 1001: Failed to decode request body

This error can be thrown on POST requests. It may indicate that:

- a passed request body is not a valid JSON, XML or other file format that was declared in `Content-Type` header;
- the value of `Content-Type` header doesn't correspond to the content of request body.

This error indicates some issue with the client-side code and should be fixed by client's developer.

HTTP status code: 400.

8.2.3 Error 1003: Server-side issue

This error can be thrown on any request. It may indicate that:

- a request was completely valid but server caught some internal error.

In this situation there is nothing to do from the client-side. Please, contact an administrator of the platform and platform's developers if needed to resolve this issue.

HTTP status code: 500.

8.2.4 Error 1004: Method not allowed

This error can be thrown on all requests. It may indicate that:

- a request method like GET, POST, PUT and so own is not supported for this resource (URL, endpoint).

This error indicates some issue with the client-side code and should be fixed by client's developer. For the full list of available resources and corresponding HTTP methods, please take a look in *REST API* page of documentation.

HTTP status code: 405.

8.2.5 Error 1005: Resource not found

This error can be thrown on all requests. It may indicate that:

- the specified resource was deleted, moved or was not existing at all.

In case of this error please double-check the specified URL. For example, you can have a spelling error, an extra slash symbol or a missing one. If you are sure that the specified URL is valid, than it means that the corresponding resource or object was deleted. This is fine. Just be ready to that.

HTTP status code: 404.

8.3 Authorization and authentication

This section is related to the errors in authorization and authentication processes.

8.3.1 Error 2000: Missing username

This error can be thrown on POST requests on `/auth` endpoint. It may indicate that:

- a client application forgot to pass 'username' field in request body;
- a client application passed a username that is equal to null.

This error indicates some issue with the client-side code and should be fixed by client's developer. Do not allow to user to send an empty username field.

> **Warning:** This behaviour may be changed if 'insecure' mode will be introduced. Please, take a look in this pull request to get more information: pull#15.

HTTP status code: 400.

8.3.2 Error 2001: Missing password

This error can be thrown on POST requests on `/auth` endpoint. It may indicate that:

- a client application forgot to pass 'password' field in request body;
- a client application passed a password that is equal to null.

This error indicates some issue with the client-side code and should be fixed by client's developer. Do not allow to user to send an empty password field.

> **Warning:** This behaviour may be changed if 'insecure' mode will be introduced. Please, take a look in this pull request to get more information: pull#15.

HTTP status code: 400.

8.3.3 Error 2002: Invalid username and password combination

This error can be thrown on POST requests on `/auth` endpoint. It may indicate that:

- the user specified a non-existing username;
- the user specified an invalid password value.

This error indicates some issue from the user-side. In this case please, help to user to log into system and provide some related suggestions.

HTTP status code: 401.

8.3.4 Error 2100: Missing Authorization header

This error can be thrown on all requests on protected resources. It may indicate that:

- the client application forgot to pass an `Authorization` header in HTTP request;
- the value of this header is null.

This error indicates some issue with the client-side code and should be fixed by client's developer. You must to pass a non-empty authorization header while accessing to protected resources. To get more information about the authorization process, please take a look into *Protected resources* section of documentation.

> **Warning:** This behaviour may be changed if 'insecure' mode will be introduced. Please, take a look in this pull request to get more information: pull#15.

HTTP status code: 400.

8.3.5 Error 2101: Invalid access token

This error can be thrown on all requests on protected resources. It may indicate that:

- the access token was revoked;
- the access token was invalid from the start.

This error indicates that the access token must to be renewed. In this case it is recommended to redirect user to authorization page. To get more information about the authorization process, plese take a look into *Protected resources* section of documentation.

> **Warning:** This behaviour may be changed if 'insecure' mode will be introduced. Please, take a look in this pull request to get more information: pull#15.

HTTP status code: 400.

8.3.6 Error 2110: Permission Denied

This error can be thrown on all requests on protected resources. It may indicate that:

- the user doesn't have an access to this resource;
- the user doesn't have a permission to modify this resource;

- the specified access token doesn't permit to process this request for some other reason.

This error indicates that the user doesn't have an access to this resource for some reason. There is nothing to do from the client- side. In this situation please describe what was happened to user and help him/her to contact an administrator of platform's instance and to get a corresponding rights.

Warning: This behaviour may be changed if 'insecure' mode will be introduced. Please, take a look in this pull request to get more information: pull#15.

HTTP status code: 403.

8.4 Things

8.4.1 Error 3100: Not an Actuator

This error can be thrown on attempts to send a command on execution to the Thing. It may indicate that:

- the `/execute` sub-resource is not available for this instance;
- this instance isn't capable of command execution.

This error indicates some issue with the client-side code and should be fixed by client's developer. Do not allow to user to send any commands to the non-actuator objects.

HTTP status code: 404.

8.4.2 Error 3101: Missing 'command' value

This error can be thrown on attempts to send a command on execution to the Thing. It may indicate that:

- the client application forgot to pass a `command` value in a body of HTTP request;
- the value of this header is not a string (i.e. is a number, null or a value of some other type).

This error indicates some issue with the client-side code and should be fixed by client's developer. You must to pass a valid `command` value while sending of commands on execution to Actuators. To get more information about the `/execute` request and its format, please take a look into *Sending commands to a Thing* section of documentation.

HTTP status code: 400.

8.4.3 Error 3102: Missing 'command_args' value

This error can be thrown on attempts to send a command on execution to the Thing. It may indicate that:

- the client application forgot to pass a `command_args` value in a body of HTTP request;
- the value of the `command_args` key is not a mapping (dictionary).

This error indicates some issue with the client-side code and should be fixed by client's developer. You must to pass a valid `command_args` value while sending of commands on execution to Actuators. To get more information about the `/execute` request and its format, please take a look into *Sending commands to a Thing* section of documentation.

HTTP status code: 400.

8.4.4 Error 3103: Unacceptable command arguments

This error can be thrown on attempts to send a command on execution to the Thing. It may indicate that:

- the client application forgot to pass some non-optional argument in the `command_args` field of a body of HTTP request;
- the client application passed an unexpected extra (additional) command argument in the `command_args` field of a body of HTTP request;
- one of the command arguments haves an invalid type;
- one of the command arguments haves an invalid value.

This error indicates some issue with the client-side code and should be fixed by client's developer. You must to pass a valid `command_args` value while sending of commands on execution to Actuators. To get more information about the `/execute` request and its format, please take a look into *Sending commands to a Thing* section of documentation.

HTTP status code: 400.

8.4.5 Error 3110: Unsupported command

This error can be thrown on attempts to send a command on execution to the Thing. It may indicate that:

- the specified instance of Actuator doesn't support the requested command.

This error indicates some issue with the client-side code and should be fixed by client's developer. You must to pass the name of a command which is supported by the specified Thing instance in `command` field in request body. To get more information about the `/execute` request and its format, please take a look into *Sending commands to a Thing* section of documentation.

HTTP status code: 400.

8.5 Placements

There is no Placement-specific exceptions for now.

CHAPTER 9

Streaming API

Nothing is here (yet)

CHAPTER 10

Capabilities

As known, different devices implement different functionality. Some devices report current climate conditions like humidity, temperature and atmospheric pressure. Other devices like air conditioners, humidifiers and climate systems are able to change such conditions in the building. Other devices allow to play music, videos, display photos and so on.

In everpl such pieces of functionality which are implemented by specific devices (Things) are called Capabilities.

Each Capability is an abstract atomic piece of functionality which can be implemented or provided by some device (Thing). Each Capability can define some new properties (fields, data) of a Thing and/or commands that can be send to device for execution.

One device can have several different Capabilities. For example, there are already mentioned climatic devices which are capable of measuring temperature, relative humidity and, maybe, CO2 levels. There are RGB Lamps which can be turned on and off, change their brightness and even change their color. There are Smart-TVs which is capable of doing... a lot of stuff.

In general, different Capabilities can be mixed in arbitrary combinations. In REST API and internal representation of the Thing the list of supported capabilities is specified in `capabilities` property of a Thing.

The list of all Capabilities that can be provided by a Thing, the list of properties and commands they provide is specified on the *next page*.

CHAPTER 11

Possible Capabilities

So, here is the list of all Capabilities possible in the system.

11.1 Actuators

Formal Capability Name `actuator`

Provided Fields No fields provided

Provided Commands The list of provided commands is specified by other Capabilities

Actuators are devices that can "act", i.e. execute some commands, to change their state and the state of the outside world. For those devices the `/execute` endpoint is available in REST API and the corresponding `execute` method is available in the internal representation of a Thing.

All Things that are able to execute some commands must to support an `actuator` capability. Otherwise all commands, even if they are specified in "Provided Commands" section of this documentation, are supposed to be unavailable.

11.2 Has State

Formal Capability Name `has_state`

Provided Fields

Field Name `state`

Field Values The set of possible values is specified by other Capabilities

Field Description Some sign of the current Thing state

Provided Commands No specific commands are provided

Has State devices are devices that have the `state` property. The value of the property is some string which is directly mapped to one of the device states. The exact set of possible states is defined by a set of Capabilities provided by the device.

11.3 Is Active

Formal Capability Name `is_active`

Provided Fields

Field Name `is_active`

Field Values boolean: `true` or `false`

Field Description Signs if this Thing is in one of the "active" states.

Provided Commands

Command Name `activate`

Command Params No params needed

Command Description Sets this Thing to the one of the "active" states

Command Name `deactivate`

Command Params No params needed

Command Description Sets this Thing to the one of the "inactive" states

Command Name `toggle`

Command Params No params needed

Command Description Toggles the Thing between the opposite states. Activates the Thing if the current state isn't active and deactivates otherwise.

Is Active devices are devices that have the `is_active` property. The value of this property is a boolean with `true` mapped to the set of "active" states (i.e. working, acting, turned on) and `false` mapped to the set of "inactive" states (i.e. not working, not acting, turned off, stopped).

Is Active Capability must to be implemented if and only if the current state of the device can be clearly mapped to either "active" or "inactive" state.

Actuator Is Active devices must to implement such methods as `toggle`, `activate` and `deactivate`.

11.4 On/Off

Formal Capability Name `on_off`

Provided Fields

Field Name `is_powered_on`

Field Values boolean: `true` or `false`

Field Description Signs if this Thing is powered on.

Provided Commands

Command Name `on`

Command Params No params needed

Command Description Powers the Thing on

Command Name `off`

Command Params No params needed

Command Description Powers the Thing off

On/Off devices are devices that can be either powered "on" or "off". The current state of those devices can be determined by the value of the `is_powered_on` field. Actuator On/Off devices are able to be turned on and off with the on and off commands correspondingly.

If the device provides both on_off and is_active capabilities, then the `on` state is usually mapped to `true` value of `is_active` field and `off` state is mapped to `false`. `on` command is also mapped to the `activate` and `off` command is mapped to the `deactivate` command.

11.5 Open/Closed

Formal Capability Name `open_closed`

Provided Fields

Field Name `state`

Field Values string: `opened`, `closed`, `opening`, `closing`

Field Description Signs if this Thing (door, valve, lock, etc.) is opened, closed or in one of the transition states.

Provided Commands

Command Name `open`

Command Params No params needed

Command Description Opens the Thing

Command Name `close`

Command Params No params needed

Command Description Closes the Thing

Open/Closed devices are devices that can be in either “opened” or “closed” state. The current state of those devices can be determined b the value of the `state` field. In addition to the “opened” and “closed” states there are two transitional states possible: “opening” and “closing”. Actuator Open/Closed devices are able to be opened and closed with the `open` and `close` commands correspondingly.

If the device provides both `open_closed` and `is_active` capabilities, then the `open` and `opening` states are usually mapped to `true` value of `is_active` field and `close` with `closing` states are mapped to `false`. Also generic `activate` and `deactivate` commands are available for such devices with `activate` mapped to `open`, `deactivate` mapped to `close` and `toggle` toggles between the opposite states (from `opened` to `closed`, from `closed` to `opened`, from `opening` to `closed`, from `closing` to `opened`).

11.6 Multimode

Formal Capability Name `multimode`

Provided Fields

Field Name `mode`

Field Values The list of provided values is specified by other Capabilities

Field Description Signs the current mode of functioning for this Thing.

Provided Commands

Command Name `set_mode`

Command Params `mode` - new value for the `mode`

Command Description Changes the mode of functioning of this Thing to the specified one.

If the device provides both `open_closed` and `is_active` capabilities, Multimode devices are able to work in different modes. By switching the mode of the device some Capabilities may become available for usage and some may gone. The current mode of the device is specified in the mode field. If the mode of the device was changed, then the list of capabilities and a set of available fields are altered to correspond to the current mode (FIXME: Is it reasonable?). Only one device mode an be chosen at a time. The current mode of the device can be set via set_mode command.

11.7 Has Brightness

Formal Capability Name `has_brightness`

Provided Fields

Field Name `brightness`

Field Values integer values in the range between 0 and 100 (including)

Field Description Specified the current level of brightness of a Thing

Provided Commands

Command Name `set_brightness`

Command Params `brightness` - the new value of brightness

Command Description Sets the specified level of brightness for the Thing

Has Brightness devices are devices that have the `brightness` property. The `brightness` property is an integer value in the range from 0 (zero) to 100. Actuator Has Brightness devices are able to change their brightness with a `set_brightness` command. Usually normal people call Actuator Has Brightness devices as "dimmable" devices.

11.8 Has Color HSB

Formal Capability Name `has_color_hsb`

Provided Fields

Field Name `color_hue`

Field Values An integer value between 0 and 359 including.

Field Description Specifies the current color of a Thing in HSB format.

Field Name `color_saturation`

Field Values An integer value between 0 and 100 including.

Field Description Specifies the current color of a Thing in HSB format.

Provided Commands

Command Name `set_color`

Command Params `hue`, `saturation` - the new value of hue and saturation correspondingly

Command Description Sets the specified color hue and saturation for the Thing

Has Color HSB devices are devices that have the "color" property. The color property value can be specified in HSB (hue, saturation, brightness) system. Actuator Has Color devices are able to change their color with a set_color command. Usually Color HSB profile is implemented by RGB Light Bulbs.

11.9 Has Color RGB

Formal Capability Name `has_color_rgb`

Provided Fields

Field Name `color_rgb`

Field Values A mapping with three keys: `red`, `green`, `blue`. The value for each key of the RGB mapping is an integer between 0 and 255 including.

Field Description Specifies the current color of a Thing in RGB format.

Provided Commands

Command Name `set_color`

Command Params `reg`, `green`, `blue` - the values of three color components: red, green and blue correspondingly

Command Description Sets the color for the Thing in RGB format.

Has Color RGB devices are devices that have the "color" property. The color property value can be specified in RGB (red, green, blue) system. Actuator Has Color devices are able to change their color with a set_color command. Usually Color RGB profile is implemented by color sensors.

11.10 Has Value

Formal Capability Name `has_value`

Provided Fields

Field Name `value`

Field Values Unspecified

Field Description Expresses some property of the Thing that can be specified as a single value.

Provided Commands

Command Name `set_value`

Command Params Unspecified

Command Description Sets the specified value for this Thing.

Has Value devices are devices that have the "value" property. This field and a corresponding property is rarely used in the real life. See `has_brightness`, `has_temperature`, `has_volume` and other similar Capabilities instead.

11.11 Has Volume

Formal Capability Name `has_volume`

Provided Fields

Field Name `volume`

Field Values The integer value between 0 and 100 including.

Field Description The value of volume (loudness) for this Thing.

Provided Commands

Command Name `set_volume`

Command Params `volume` - a new value of the volume for this Thing.

Command Description Sets the specified volume (loudness level) for this Thing.

Has Value devices are devices that have the "volume" property - the measure of loudness of how loud its sound is. Volume is an integer value in the range from 0 (zero) to 100. Actuator Has Volume devices are able to change their volume with a `set_volume` command.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search