everpl Documentation

Release 0.4.0-alpha

Sergey Kostyuk

May 30, 2018

User Documentation

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` always look like: `protocol://domain:port/api/rest/v1`

Where:

- `protocol` is either `http` or `https` for unsecured and secured (TLS) HTTP connection;
- `domain` is a fully-qualified domain name or IP address of evepl instance you are connecting to;
- `port` is a port used for HTTP connection;
- `api/rest/` is a constant part of an address;
- `v1` indicates the currently used version of the REST API.

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.

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.

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

7.4.1 Thing object

General thing object has the following structure:

capabilities A list of Capabilities, supported by this Thing. For more information see *Capabilities* section of documentation.

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)

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.

Meanwhile, Actuator Things usually (but not always[2]) provide some additional fields:

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).

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'.

The exact set of fields and their values may vary for different types of things. For detailed information, please refer to the *Possible Capabilities* and *Generic Thing Types* sections of documentation.

Example of an Actuator 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:

[2] Only the presence of `commands` field is granted for Actuators. For more information about available fields please refer to the *Capabilities* section of documentation.

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.

8.6 Streaming API

Streaming API has its own subset of errors in addition to the errors defined above. All errors with identifiers starting from `5000` and to `5999` including are considered as Streaming API-specific errors.

8.6.1 Error 5000: Timeout

This error can be thrown on attempts to use a Streaming API. It may indicate that:

- the server was expected to receive a message from a client in the specified time window but such message wasn't sent.

This error indicates some issue with the client-side code and should be fixed by client's developer. In some situations server may wait a message from a client application in the specified time window (not later than X time units after some point of time). For example, the client must to send Authentication message not later than 20 seconds from the connection establishment (as defined in *Streaming API* section of documentation). You must to send messages in the specified time windows, otherwise you will receive this (5000) error.

8.6.2 Error 5001: Invalid frame type

This error can be thrown on attempts to send a frame using a Streaming API. It may indicate that:

- the frame sent has type that is different from expected.

This error indicates some issue with the client-side code and should be fixed by client's developer. For now the only supported type of WebSocket frame is TEXT frame. TEXT frames are then parsed as JSON objects and interpreted as Streaming API Messages. You must not to use binary frames or any other frames for transferring Streaming API Messages.

8.6.3 Error 5002: Invalid frame content

This error can be thrown on attempts to send a frame using a Streaming API. It may indicate that:

- the content of the specified TEXT frame is not a JSON object.

This error indicates some issue with the client-side code and should be fixed by client's developer. For now all the messages passed via Streaming API must to be encoded as JSON objects according to the rules defined in *Streaming API* section of documentation. You must to encode Messages as JSON objects and transfer them in TEXT WebSocket frames. Otherwise the mentioned (5002) error will be thrown.

8.6.4 Error 5003: Message format violation

This error can be thrown on attempts to send a message using a Streaming API. It may indicate that:

- the received message is a valid JSON object is not a valid Message object;
- some fields of Message are missing or have an appropriate type.

This error indicates some issue with the client-side code and should be fixed by client's developer. For now all the messages passed via Streaming API must to be encoded as JSON objects according to the rules defined in *Streaming API* section of documentation. You must to encode Messages as JSON objects and transfer them in TEXT WebSocket frames. Otherwise the mentioned (5003) error will be thrown.

The name of erroneous field is specified in `devel_message` field of Error message.

8.6.5 Error 5004: Session was resumed on another connection

This error can be thrown on already opened Streaming API connections. It may indicate that:

- some client initiated a new Streaming API connection using the same access token and connected to the same Session.

This error usually indicates some issue with the client-side code and should be fixed by client's developer. In some situations, client applications keeps old connections unclosed while attempting to establish a new one. In such situations, the old connection is closed with the specified error - 5004. To avoid this error, please, close the old Streaming API connections before the new connection will be opened.

8.6.6 Error 5010: Invalid message type (not Control)

This error can be thrown on attempts to use a Streaming API. It may indicate that:

- the server was expected to receive a Control Message from a client but the message received is not a Control Message.

This error indicates some issue with the client-side code and should be fixed by client's developer. In some situations server may wait a message from a client application with the specified type: either Control Message or Data Message. To define either the received Message is Control or Data Message, the `type` field is used according to the *Streaming API* section of documentation. You must to send messages with a type, appropriate to the current situation, otherwise you will receive this (5010) error.

8.6.7 Error 5011: Invalid message type (not Data)

This error can be thrown on attempts to use a Streaming API. It may indicate that:

- the server was expected to receive a Data Message from a client but the message received is not a Data Message.

This error indicates some issue with the client-side code and should be fixed by client's developer. In some situations server may wait a message from a client application with the specified type: either Control Message or Data Message. To define either the received Message is Control or Data Message, the `type` field is used according to the *Streaming API* section of documentation. You must to send messages with a type, appropriate to the current situation, otherwise you will receive this (5011) error.

8.6.8 Error 5020: Invalid message topic

This error can be thrown on attempts to use a Streaming API. It may indicate that:

- the server was expected to receive a Message with the specified topic from a client but the topic of the received message is different from expected.

This error indicates some issue with the client-side code and should be fixed by client's developer. In some situations server may wait a message from a client application with the specified topic. You must to send messages with a topic, appropriate to the current situation, otherwise you will receive this (5020) error. To define what message topic is expected in the current situation, please refer to the *Streaming API* section of documentation. The expected topic of a message is defined in `devel_message` field of Error message.

8.6.9 Error 5030: Invalid message body content

This error can be thrown on attempts to use a Streaming API. It may indicate that:

- the server expected to find some information in a body content of received message but such information is missing or is invalid (by type or value).

This error indicates some issue with the client-side code and should be fixed by client's developer. In some situations client must to send messages with the specified type, topic and the message body content. You must to send messages with bodies as defined in the *Streaming API* section of documentation, otherwise you will receive this (5030) error. The name of the missing or erroneous field is defined in `devel_message` field of Error message.

CHAPTER 9

Streaming API

Starting from v0.4.0 of everpl the new type of API is provided: the Streaming API.

The streaming API allows to receive updates of system objects (like Things), push notifications and other system events in real-time: just after such events had happened. And more of it: client applications are able to choose what events they are concerned about and what events they wanna subscribe to.

9.1 General Information

For now the Streaming API is implemented over WebSocket two-way communication protocol[1]. Once WebSocket connection is established and authorization procedure was complete, an everpl instance and a client application instance are allowed to send messages to each other.

The format of such messages is described in the next section.

9.2 Connection procedure

To connect to the Streaming API, you need to open a WebSocket connection to the server using the following address: `protocol://domain:port/api/streaming/v1/`

Where:

- `protocol` is either `ws` or `wss` for unsecured and secured (TLS) WebSocket connection;
- `domain` is a fully-qualified domain name or IP address of evepl instance you are connecting to;
- `port` is a port used for WebSocket connection (usually the same as used for REST API connection);
- `api/streaming/` is a constant part of an address;
- `v1` indicates the currently used version of the Streaming API;
- the trailing slash (`/`) is mandatory.

[1] WebSocket protocol is fully documented in RFC 6455

In order to connect to the Streaming API, there is no need to supply additional request headers on WebSocket handshake. But all clients are must to send an *Authentication* request **immediately** after WebSocket connection was established. If the Authentication request will not be sent within 20 seconds after connection, then the server is allowed to terminate a connection after that period was expired.

After the Authentication procedure was passed, both sides are allowed to start a normal communication over WebSocket connection.

9.3 Message Format

Message is a JSON object, transmitted as a Text data frame[2] over WebSocket connection. Generally any Message consists of the specified fields:

timestamp float, the moment of time when this message was generated, formatted as a UNIX time number (i.e. number of seconds passed from 1 January 1970 UTC).

type string, indicates if the message carries some data (data message) or control information (control message). Can take either `data` or `control` value.

topic string, a topic of this message (described in detail below).

body Another JSON object. The type and format of this object is dependent on a topic of the message.

message_id integer, an optional field, a temporary identifier of a message that allows to acknowledge that a message was received by a client. Such identifiers may be reused by server, so two distinct messages can receive the same identifiers in different points of time. Is provided only if the *Message Retention* was enabled for the corresponding message topic.

All messages belong to one of two types: **Control Messages** or **Data Messages**. Control messages are intended to carry information to control communication using the Streaming API. All of them are described in a *Control Messages* section of documentation. Data messages are intended to carry some useful data from client to the server and in reverse direction.

Message topics allow to group messages by... topics. By types of the messages they belong to, by type of the event they describe. For example, there can be topics for each object in the system, the group of messages notifying about object creation, deletion or updates. Or there is a topic for some push notifications, that must to be delivered to user (for example, that might be notifications about a detected motion in a building or about a critical issue in the system). And so on.

The content of the message body is greatly dependent on a topic. Each topic is allowed to specify its own set of available fields that will carry an additional information about what exactly happened in the system or what to do with that.

9.4 Sessions and data retention

All the communication between the server and client is processed in a scope of a Session. Session is a scope of time between the user was logged in on a client device (client application) and the user was logged out on on a device or an access for such device was revoked.

One session always corresponds to exactly one user and to exactly one authorized device or application. Sessions are identified by the corresponding access tokens, specified by client applications in client *Authentication* procedure.

All the subscriptions (as described below in *Topics and subscriptions* section) **are** saved between connections. So, if a WebSocket connection will be interrupted for any reason (such as issues with a network connection or a graceful disconnection of a client), then all the subscriptions and other session-related data will be restored.

[2] About Text data frames in the WebSocket protocol: RFC 6455 Section 5.6

The only exception in this rule is the following. If the client access will be revoked and a client Session will be terminated by server for any reason, than all Session-related data will be **deleted** from a server. And, as result, you'll need to start everything from scratch.

Also, clients are allowed to ask a server to store last all messages for specific topic until their delivery will be explicitly acknowledged by clients. For more information about this feature, see the *Message Retention* section of documentation.

9.5 Message Retention

Message Retention feature allows to keep the all the last messages for specific topics until their delivery will be specifically acknowledged by a client.

The server is guarantied to save up to 100 retained messages **per client**. After this limit was reached, the server is allowed to **remove** all the old messages above this limit. So please, enable message retention only when you it's really needed. And if the client device is expected to go offline for a long period of time - please, remove all unneeded subscriptions that have message retention enabled **before** disconnection.

It's needed to enable message retention explicitly for each topic. To enable message retention, just set a `retain_messages` value to `true` on topic subscription, as described in *Topic subscriptions* section of documentation.

With the message retention enabled, the client **must** acknowledge the delivery of each message using the following special message:

```
{
    "timestamp": 123456.76,
    "type": "control",
    "topic": "delivery_ack",
    "body": {
        "message_id": 12
    }
}
```

Where:

- `type` value is constantly equal to `control`;
- `topic` value is constantly equal to `delivery_ack`;
- `timestamp` is set to the current UNIX time (`123456.76` on example);
- `message_id` value is an integer, a temporary identifier of a message to be acknowledged.

Retained messages are allowed to be re-sent until their delivery will be acknowledged by a client. The time between attempts to re-send a message will grow exponentially until the delivery wil be confirmed by a client.

On re-connection all retained messages are re-sent immediately after the client authentication.

9.6 Topics and subscriptions

Topic is a string of the following format: `topic/subtopic/subtopic`

Each topic has a hierarchical structure:

- the first part (topic layer; `topic` in example) is root topic for that category of messages;
- the second and the following parts are sub-topics, sub-categories of messages.

Topic layers are separated between each other with a forward slash sign (`/`; the topic layer separator). The number of such topic layers is unlimited in theory, but in practice rarely exceeds the number of three. Please note, that there is no slash at the beginning of the topic.

All topics are case-sensitive, so such strings as `my_topic` and `My_topic` correspond to the entirely different topics.

9.6.1 Topic subscriptions

As was mentioned earlier, once WebSockets connection is established, client applications are able to subscribe to different topics.

To subscribe to a topic, a client application must to send the following message:

```
{
    "timestamp": 123456.76,
    "type": "control",
    "topic": "subscribe",
    "body": {
        "target_topic": "here/is/your/topic",
        "retain_messages": false
    }
}
```

Where:

- `type` value is constantly equal to `control`;
- `topic` value is constantly equal to `subscribe`;
- `timestamp` is set to the current UNIX time (`123456.76` on example);
- `target_topic` value is set the topic you want to subscribe onto (`here/is/your/topic` on example);
- `retain_messages` is an optional boolean parameter that enables message retention for this topic; set to `false` (disabled) by default.

In response to that message you will receive the following message with an empty body:

In response to that message you will receive the following message:

```
{
    "timestamp": 123456.76,
    "type": "control",
    "topic": "subscribe_ack",
    "body": {
        "target_topic": "here/is/your/topic"
    }
}
```

Where `target_topic` is the same topic that was specified in the `subscribe` message.

9.6.2 Wildcard subscriptions

In addition to the individual per-topic subscriptions, you are able to subscribe to several topics at once. To do so, you have a pair of additional operators: `+` and `#`.

The `+` operator is equal to the "any name on this level of hierarchy" meaning. For example, if you will subscribe to the `things/+/updated` topic, then you will receive messages from topics like `things/door1/`

`updated`, `things/player1/updated` but that doesn't means that you will receive messages from topics like `placements/place1/updated`, `things/player1/updated`, `things` or others automatically.

The `#` operator can be present only as the last symbol in the topic string and means "subscribe to all messages with topics below the specified level of hierarchy". For example, `things/#` allows to subscribe to any updates (creation, deletion and modification) of any Thing in the system (topics like `things/door1/updated`, `things/player1/updated` and `things/door1/deleted`). And such subscriptions as `things/player1/#` allows to watch for all updates of a specific Thing in the system.

Please note that such operator as `*` and partial match topics like `things/pla*er1/updated` are **not** supported by the platform. Such strings as `topic/subtopic/foo+`, `topic/subtopic/foo+bar`, `topic/#/subtopic` and `topic/subtopic/+foo` are also considered invalid.

9.6.3 Unsubscribe from a topic

To unsubscribe to a topic, a client application must to send the following message:

```
{
    "timestamp": 123456.76,
    "type": "control",
    "topic": "unsubscribe",
    "body": {
        "target_topic": "here/is/your/topic"
    }
}
```

Where:

- `type` value is constantly equal to `control`;
- `topic` value is constantly equal to `subscribe`;
- `timestamp` is set to the current UNIX time (`123456.76` on example);
- `target_topic` value is set the topic you want to unsubscribe from (`here/is/your/topic` on example).

In response to that message you will receive the following message:

```
{
    "timestamp": 123456.76,
    "type": "control",
    "topic": "unsubscribe_ack",
    "body": {
        "target_topic": "here/is/your/topic"
    }
}
```

Where `target_topic` is the same topic that was specified in the `unsubscribe` message.

9.7 Authentication

Authentication is performed just after WebSocket connection was established. To perform an authentication, you need to send your access token[3] in the following message:

[3] About how to get an access token is described in *REST API* section of documentation, Authentication sub-section.

```
{
    "timestamp": 123456.76,
    "type": "control",
    "topic": "auth",
    "body": {
        "access_token": "here_is_your_token"
    }
}
```

Where:

- `type` value is constantly equal to `control`;
- `topic` value is constantly equal to `auth`;
- `timestamp` is set to the current UNIX time (`123456.76` on example);
- `access_token` value is set the your access token to be used (`here_is_your_token` on example).

In response to that message you will receive the following message with an empty body:

```
{
    "timestamp": 123456.76,
    "type": "control",
    "topic": "auth_ack",
    "body": {}
}
```

Once authenticated, you are able to transmit other messages as described on this page.

9.8 Handling Errors

If there is any error happened in communication, you will receive a special message with a topic `error`. Such messages have the following format:

timestamp float, the moment of time when this message was generated, formatted as a UNIX time number (i.e. number of seconds passed from 1 January 1970 UTC).

type string, constantly set to the `control`.

topic string, constantly set to the `error`.

body Another JSON object. Information about an error in the format described in the *Handling Errors* section of documentation.

Error messages share the common error codes and a format of a body as described in *Handling Errors* section of documentation. So, it's recommended to use the same error handling code for both Streaming API and REST API errors if possible.

Here is an example of an error message:

```
{
    "timestamp": 123456.76,
    "type": "control",
    "topic": "error",
    "body": {
        "error_id": 2101,
        "devel_message": "Invalid access token",
        "user_message": "Access token was revoked. Please, authenticate."
```

(continues on next page)

(continued from previous page)

```
    }
}
```

9.9 Message Types

As was mentioned earlier, there can be different types of messages with different message bodies for different topics. We already talked about three special types of messages: error messages (*Handling Errors*), authentication (*Authentication*) and subscription (*Topics and subscriptions*) messages.

Below is a small recap of special message types and a description of some general message types.

9.9.1 Control Messages

1. **`error`** Indicates an error in communication using Streaming API, described above in the *Handling Errors* section of documentation.
2. **`subscribe`** Allows streaming client to subscribe on a specific topic. Described above in the *Topic subscriptions* section of documentation.
3. **`subscribe_ack`** An acknowledgement packet, sent by a server on successful subscription. Described above in the *Topic subscriptions* section of documentation.
4. **`unsubscribe`** Allows streaming client to unsubscribe from a specific topic. Described above in the *Unsubscribe from a topic* section of documentation.
5. **`unsubscribe_ack`** An acknowledgement packet, sent by a server if the subscription was successfully cancelled. Described above in the *Unsubscribe from a topic* section of documentation.
6. **`delivery_ack`** An acknowledgement packet, sent by a **client** if a message with the specified identifier was successfully received. Described above in the *Message Retention* section of documentation.

9.9.2 Object-Related Messages

Object-Related messages are responsible for notification of client application about the created, updated or deleted objects in the system. All of such messages has the following structure:

timestamp float, the moment of time when this message was generated, formatted as a UNIX time number (i.e. number of seconds passed from 1 January 1970 UTC).

type string, constantly set to the `data`.

topic string, topic in the following format: `{object_category}/{object_id}/{what_happened}`.

body Another JSON object. The DTO of the modified object or `null` if the specified object was deleted.

Where:

- `{object_category}` is one of the following values: `things`, `placements`, `users` for Things, Placements and Users correspondingly[4];
- `{object_id}` is a unique identifier of the specified object;

[4] Information about all that types of objects can be found at the *REST API* section of documentation in corresponding sub-sections.

- `{what_happened}` is one of the following values: `created`, `updated`, `deleted` for messages about the created, updated and deleted objects correspondingly;
- the body contents the current state of an object in a corresponding format[4].

So here is an example of such message:

```
{
    "timestamp": 1505768807.4725718,
    "type": "data",
    "topic": "things/F1/updated",
    "body": {
        "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"
    }
}
```

9.9.3 Notifications

> **Warning: Unstable API**
>
> Notifications API and a format of Notifications is not yet stabilized. Please, check this page later for updates.

Notifications are messages that are supposed to be directly showed to the user of a client application. They have the following format:

timestamp float, the moment of time when this message was generated, formatted as a UNIX time number (i.e. number of seconds passed from 1 January 1970 UTC).

type string, constantly set to the `data`.

topic string, constantly set to `notifications`.

body Another JSON object. Contains the following fields:

title string, a title of the notification

text string, an optional field, text to be displayed in notification

image_url string, an optional field, a link to the image to be displayed in notification

Where optional fields can be omitted (absent) or set to `null`.

> **Warning:** Maybe such field as “urgency” or other fields must to be added?

9.10 P.S.

If any of the information above reminded you MQTT protocol - it is no accident. The topic format was greatly inspired by the one in MQTT protocol. But other things (like the authorization and subscription procedures, the set of provided features and underlying implementation) are different.

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 Multi-Mode

Formal Capability Name `multi_mode`

Provided Fields

Field Name `current_mode`

Field Values A string from of the `available_modes` list

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

Field Name `available_modes`

Field Values List of strings

Field Description Signs all available modes 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.

Multi-Mode 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. All available device modes are listed in `available_modes` field. The content of `available_modes` list is defined by Thing Type and provided Capabilities.

11.7 Has Brightness

Formal Capability Name `has_brightness`

Provided Fields

Field Name `brightness`

Field Values floating point values in the range between 0.0 and 100.0 (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 a floating point value in the range from 0.0 (zero) to 100.0. Actuator Has Brightness devices are able to change their brightness with a `set_brightness` command. Usually normal people call Actuator Has Brightness devices "dimmable" devices.

11.8 Has Color HSB

Formal Capability Name `has_color_hsb`

Provided Fields

Field Name `color_hue`

Field Values A floating point value between 0.0 including and 360.0 not including.

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

Field Name `color_saturation`

Field Values An floating-point value between 0.0 and 100.0 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 values of hue and saturation correspondingly

Command Description Sets the specified color hue and saturation for the Thing. Brightness must to be set separately, see *Has Brightness* Capability description for details.

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 Color Temperature

Formal Capability Name `has_color_temp`

Provided Fields

Field Name `color_temp`

Field Values Integer between 1000 and 10000 including. Color temperature value, expressed in Kelvins.

Field Description Specifies the current color temperature of a Thing in Kelvins.

Provided Commands

Command Name `set_color_temp`

Command Params `color_temp` - the new value of color temperature to be set.

Command Description Sets the color temperature for a Thing.

Color Temperature devices are devices that have the "color temperature" property. The color temperature is expressed in Kelvins and can take integer values from 1000 to 10000 including. The color temperature of light source or other Actuator can be set with `set_color_temp` command. If the Thing doesn't support specified color temperature value (i.e. it's too low or too high for this Thing), then the color temperature will be set to the nearest supported value. For example, the minimum value is 2000 and the maximum value is 6500 K for majority of light bulbs available on the market. It's recommended for client applications to put some marks on the scale for Warm White (2700 K), Cool White (4000 K) and Daylight (5000 K) values.

11.11 Has Temperature

Formal Capability Name `has_temperature`

Provided Fields

Field Name `temperature_c`

Field Values Floating point, temperature in degrees of **Celsius**.

Field Description Expresses the current temperature measured by a Thing.

Provided Commands No commands provided

Has Temperature devices are devices that have the "temperature" property. The value of "temperature_c" property is expressed in degrees of Celsius, Fahrenheits are not supported for now.

It's supposed that the value of "temperature" property can'be changed by user and represents the current, real temperature of controlled object. For other purposes, please refer to Capability and Thing types which provide a "target_temperature" property.

11.12 Has Position

Formal Capability Name `has_position`

Provided Fields

Field Name `position`

Field Values Floating point number which represents the position of an object in numbers from 0.0 to 100.0 including.

Field Description Expresses the current position of a Thing.

Provided Commands

Command Name `set_position`

Command Params `position` - the new value of position to be set.

Command Description Sets the new position for a Thing.

Has Position devices are devices that have the "position" property. This property allows to set a position of an object using only one single dimension. For example, it can represent the position of a shade (50% unrolled, 20% of window covered, etc.), the width of an opening (for gates, sliding doors, valves) and so on.

11.13 Fan Speed

Formal Capability Name `fan_speed`

Provided Fields

Field Name `fan_speed`

Field Values Floating point numbers from 0.0 to 100.0 including.

Field Description Expresses the current fan rotation speed in percents.

Provided Commands

Command Name `set_fan_speed`

Command Params `fan_speed` - the new value of fan speed to be set.

Command Description Sets the new fan speed for a Thing.

Fan Speed devices are devices that have a build-in and externally controllable (at least monitored) fan. For example, that can be heaters, some HVACs and fans itself (as separate devices).

The speed of some fans can be changed only by a constant step. For such cases (for example, for table fans with only 3 speeds), the whole range will be separated on the corresponding number of segments. For example, it'll be 0-25, 26-50, 51-75 and 76-100 for a generic fan with speeds 0 (stopped), 1, 2 and 3 correspondingly.

11.14 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.

11.15 Play/Stop

Formal Capability Name `play_stop`

Provided Fields

Field Name `state`

Field Values string: `playing`, `stopped`

Field Description Signs if the playback for this Thing (for some kind of player) is in progress (playing) or stopped.

Provided Commands

Command Name `play`

Command Params No params needed

Command Description Starts the playback.

Command Name `stop`

Command Params No params needed

Command Description Stops the playback.

Play/Stop devices are devices that can play some media (i.e. music, video, radio, media stream, etc.) and which have basic controls for playback. Uses the "state" field to define the current playback state and corresponding commands to stop and resume playback.

11.16 Pausable

Formal Capability Name `pausable`

Provided Fields

Field Name `state`

Field Values string: `paused`

Field Description Signs if the activity for this Thing (playing, recording, etc.) is paused.

Provided Commands

Command Name `pause`

Command Params No params needed

Command Description Pauses the current activity.

Pausable devices are devices that can pause the current activity (i.e to temporarily stop it with keeping of a current position). Usually provided by some kinds of Players or Recorders. For Actuator Pausable Things the "pause" command can be used to pause the current activity (i.e. the playback, recording and so on).

Usually implemented alongside with Play/Stop Capability.

11.17 Track Switching

Formal Capability Name `track_switching`

Provided Fields No fields provided

Provided Commands

Command Name `next`

Command Params No params needed

Command Description Switches the playback to the next track, video or stream.

Command Name `previous`

Command Params No params needed

Command Description Switches the playback to the previous track, video or stream.

Track Switching devices are devices that can switch between the current, previous and the next track, song, file, video or stream in the playback queue. Usually implemented by Players. Track Switching devices aren't obliged to support playlists, switching to specific tracks in the queue and so on. For support of the mentioned features please refer to the corresponding Capabilities.

Usually implemented alongside with Play/Stop and Pausable Capabilities.

11.18 Track Info

Formal Capability Name `track_info`

Provided Fields

Field Name `track_info`

Field Values String

Field Description Contains the information about a current playing song, movie, stream or another media in a form of a single human-readable string.

Provided Commands No commands provided

Track Info devices are devices that can display information about the current playing media. The type of this information can be arbitrary and is not specified by this document. It's not even supposed to be parsed by other devices. The only thing that must to be granted is that the track_info field value must to be human-readable without any additional processing.

For support of information about the song name, movie name, artists, current playing TV program and so on please refer to the corresponding Capabilities and Thing types.

11.19 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.

11.20 Is Muted

Formal Capability Name `is_muted`

Provided Fields

Field Name `is_muted`

Field Values boolean: `true` or `false`

Field Description Indicates if the Thing was muted.

Provided Commands

Command Name `mute`

Command Params No params needed

Command Description Mutes the Thing.

Command Name `unmute`

Command Params No params needed

Command Description Unmutes the Thing - moves the Thing from a "muted" state.

Is Muted devices are devices that have the "is_muted" property - the indicator of either device was muted (i.e. has temporarily disabled sounding) or not. Actuator Is Muted devices are able to be muted and unmuted with `mute` and `unmute` commands correspondingly.

11.21 Multi-Source

Formal Capability Name `multi_source`

Provided Fields

Field Name `current_source`

Field Values An integer value, an index of the current source from the `available_sources` list.

Field Description Contains an identifier of the source which is currently chosen.

Field Name `available_sources`

Field Values An ordered list of strings.

Field Description An list of human-readable names for all available sources.

Provided Commands

Command Name `set_source`

Command Params `source` - a new value for the `current_source` field.

Command Description Sets the specified source for this Thing.

Multi-Source devices are devices that can play, display or use in any other way information from one of several information sources. The good example of such device is a computer monitor. Computer monitor often can display information from several inputs as HDMI, VGA or DisplayPort input. Or a speaker system which can play a sound from coaxial, optical, HDMI, Bluetooth or AUX inputs.

For such devices as TVs, home theaters and other multi-functional devices please refer to the *Multi-Mode* Capability documentation.

CHAPTER 12

Generic Thing Types

While talking about Things, the two important field was mentioned: a “type” field and a “capabilities” field.

A “capabilities” field was discussed earlier in detail , in *Capabilities* section of documentation. Capabilities define what devices are capable of, what such devices can do and provide.

But the meaning of a “type” field was left almost not discussed. This flaw will be fixed below, in Generic Thing Types section of documentation

12.1 General Information

There is huge variety of Things available on the market. There are some crazy devices that combine functions of a light bulb and a speaker[1], toaster and printer[2], or Bluetooth-enabled toasters[3], or fridges with Tizen onboard[4]...

But most of the time, it’s possible to pick out a primary functionality of a device and thus classify it to one of generic, common device types. And this, in turn, allows developers to provide the most relevant information to the user. At least, an appropriate device icon :).

Below, there is a list of generic Thing types that are recommended to be supported by both client applications and device Bindings in Integration packages. For every generic Thing type, there are device icons, lists of Capabilities and other parameters recommended to be implemented by Bindings and supported by client applications. Use the left navigation menu to find any device type quickly.

12.2 Value Sensor

type “value_sensor”

inherits from none

[1] Light bulb *speakers* or light bulb *with* speakers? Sony LSPX-100E26J
[2] Toasteroid: http://kck.st/2b5uRHy
[3] Why not to add a display and Bluetooth audio support too? https://goo.gl/VRKYp5
[4] Samsung Family Hub

icon two random digits or a sensor icon

capabilities “has_value”

A generic type of sensors which represent their results of measurements in numbers of an unspecified unit. Such values must to be displayed to user in the following manner: “current value is %d”, where “%d” is an placeholder for the measured value. Must to be used rarely, only if there is no more specific device type declared.

12.3 Binary Sensor

type “binary_sensor”

inherits from “value_sensor”

icon “I/O” text or a similar-looking icon

capabilities “has_value”, “is_active”

The most primitive (but not necessary the base) type of Sensors in the system. Can have only one of two integer values: 1 (one) or 0 (zero). Where 1 is mapped to the “active” state and 0 is mapped to “not active”. Usually is inherited by more specific implementations of a Binary Sensor, including buttons, leakage sensors, contact sensors (detects an opening of a door or window), motion sensors and so on.

12.4 Button

type “button”

inherits from “binary_sensor”

icon an icon of a button

capabilities “has_value”, “is_active”

Represents all Buttons connected to the system. Its value is set to 1 (one) while the button is pressed and sets to 0 (zero) just after the button was released. There is no long press detection, double press detection and so on. Just “pressed” (1) and released (0). All the other functionality is the same as in Binary Sensor.

12.5 Switch

type “switch”

inherits from “binary_sensor”

icon an icon of a switch (or a reed switch)

capabilities “has_value”, “is_active”

Another kind of a Binary Sensor. Is a base type for devices which can preserve their state without the help of a user (i.e. user doesn’t need to keep the switch pressed). Physically, it can be a simple light switch, toggle switch, reed switch and on and on. As any other Binary Sensor, the “value” field value can be equal to either 1 or 0, where 1 is mapped to “active” and 0 is mapped to “not active”.

12.6 Contact Sensor

type "contact_sensor"

inherits from "switch"

icon an icon of a reed switch or an opened door without a handle

capabilities "has_value", "is_active", "has_state", "open_closed"

The special subtype of a Switch. Adds a new field to the list: a "has_state" field which can take either "opened" or "closed" values, where "opened" is equal to 1 and "closed" is equal to 0.

12.7 Motion Sensor

TBD. Has the same logic as Button

12.8 Leakage Sensor

TBD. Has the same logic as Switch

12.9 Temperature Sensor

type "temperature_sensor"

inherits from none

icon an icon of a thermometer

capabilities "has_temperature"

Temperature Sensor represents simple thermometers, temperature sensors which displays the current temperature of a controlled object: in-room air temperature, outside temperature, temperature of a human body, etc. If your device implements some features in addition to measuring of temperature - please, consider some other base types for your device.

12.10 Humidity Sensor

TBD. Almost the same as Temperature Sensor but measures humidity instead of temperature.

12.11 Climate Station

TBD. Combines functions of humidity, temperature, gas and air quality sensors.

12.12 Lock

type "lock"

inherits from none

icon an icon of a keyhole or padlock

capabilities "actuator", "has_state", "is_active", "open_closed"

Represents all kinds of controllable Locks. Allows to at least lock the controlled door, gate or another object. Unlocking capability is optional. The "state" field can take either one of the end state values ("opened" or "closed") or one of the transitional state values ("opening", "closing").

12.13 Door Actuator

type "door_actuator"

inherits from none

icon an icon of an opened door without a handle

capabilities "actuator", "has_state", "is_active", "open_closed"

Represents Actuator mechanisms which are able to open and close the physical door, gate or other similar object. The "state" field can take either one of the end state values ("opened" or "closed") or one of the transitional state values ("opening", "closing").

12.14 Shades

type "shades"

inherits from none

icon an icon of a window with shades

capabilities "actuator", "has_state", "is_active", "open_closed"

optional_capabilities "has_position"

Represents all kinds of shades - objects which cover the window and reduce the amount of light passed through it. Also named as sunblinds, shutters, louvers and so on. Their state can take either "opened" or "closed" values, where "opened" is equal to "active" and "closed" equal to "not active". Two transitional states are also possible: "opening" and "closing". Some shades can also provide a "has_position" capability that allows to set the position of shades in percents from 0 to 100, regarding to the area of window covered by shades.

12.15 Light

type "light"

inherits from none

icon pendant lamp icon

capabilities "actuator", "has_state", "is_active", "on_off"

Light is a common type for all lightning devices: LED strips, light bulbs, floor lamps and so on. The base functionality of such devices is to be turned on and off. And to emit light in the turned on state.

12.16 Dimmable Light

type "dimmable_light"

inherits from "light"

icon pendant lamp icon

capabilities "actuator", "has_state", "is_active", "on_off", "has_brightness"

Dimmable Light is a common device type for all lighting devices that can be dimmed, i.e. that can change their level of brightness. The rest of functionality is inherited from the base Light device type.

12.17 Color Temperature Light

type "ct_light"

inherits from "dimmable_light"

icon pendant lamp icon

capabilities "actuator", "has_state", "is_active", "on_off", "has_brightness", "has_color_temperature"

Color Temperature Light is a common device type for all lighting devices that can change their color temperature. The rest of functionality is inherited from the base Dimmable Light device type.

12.18 Color Light

type "color_light"

inherits from "ct_light"

icon colorized pendant lamp icon

capabilities "actuator", "has_state", "is_active", "on_off", "has_brightness", "has_color_temperature", "has_color_hsb"

optional_capabilities "has_color_rgb"

Color Light is a common device type for all lighting devices that can change their color of light. The rest of functionality is inherited from the base Color Temperature Light device type.

Additionally, devices can support an "has_color_rgb" capability which allows to set a color in RGB color units. This capability is optional because not all devices on the market support it. And often it's hard to determine a clear mapping between RGB and HSB color values.

12.19 Power Switch

type "power_switch"

inherits from none

icon switch icon

capabilities “actuator”, “has_state”, “is_active”, “on_off”

Power Switch type represents all power switches in the system. Such power switches include smart power outlets, circuit breakers, switches that are not Light switches and other similar devices. The only functionality of such devices is to turn connected load on and off.

If your power switch or power outlet implements an additional functionality or is not really a power switch - please, search for a more appropriate base type in this documentation or file an issue on GitHub[5].

12.20 Valve

type “valve”

inherits from none

icon valve icon

capabilities “actuator”, “has_state”, “is_active”, “open_closed”

Valve represents an externally controllable valve for gas, liquid or other matter which can be either in “opened” or “closed” state. Transitional states “opening” and “closing” are also possible.

In addition to valve-specific device states and commands, valves support an “is_active” capability where “active” is equal to “opened” and “not active” is linked to “closed”.

12.21 Fan

type “fan”

inherits from none

icon fan icon

capabilities “actuator”, “has_state”, “is_active”, “on_off”

Fans is the most primitive type of the climatic devices. Fans can be either in “on” or “off” states while fan speed control is not supported. Additional functionality like enabling and disabling heaters is not supported too.

12.22 Variable Speed Fan

type “vs_fan”

inherits from “fan”

icon fan icon

capabilities “actuator”, “has_state”, “is_active”, “on_off”, “fan_speed”

Variable Speed Fans are fans whose speed of rotation can be controlled. In the rest, it’s just a usual Fan described above.

[5] All issues can be reported on the project’s page: https://github.com/s-kostyuk/everpl/issues

12.23 Player

type "player"

inherits from none

icon "play" icon in a circle

capabilities "actuator", "has_state", "is_active", "play_stop"

optional capabilities "on_off", "has_volume"

Player is a base type for all kinds of players: audio players, video players, streaming players, radios and so on and so forth. Such devices doesn't allow to change tracks, pause the playback or do anything similar. They can be only in one of two states: "playing" and "stopped", where "playing" state is mapped to the "active" state while "stopped" to "not active".

The "on_off" Capability can be provided by real, hardware players. In such case, it's recommended to provide a separate button to control player's power and separate buttons to control playback.

Some players can also provide an "has_volume" capability but it's not absolutely necessary.

12.24 Pausable Player

type "pausable_player"

inherits from "player"

icon "play" icon in a circle

capabilities "actuator", "has_state", "is_active", "play_stop", "pausable"

optional capabilities "on_off", "has_volume"

Pausable Player type represents all Players which support pausing - temporarily stopping of playback with saving of the current playback position. In general, it's the same Player as described above with all its functions and limitations. The only thing that was added is an additional "paused" state and a corresponding "pause" command.

12.25 Track Player

type "track_player"

inherits from "pausable_player"

icon "play" icon in a circle

capabilities "actuator", "has_state", "is_active", "play_stop", "pausable", "track_switching", "track_info"

optional capabilities "on_off", "has_volume"

Track Player type represents all devices with an ability to switch between tracks: backward and forward. It inherits all the fields and behaviour provided by Pausable Player type but adds two additional commands: "next" and "previous". Also, there is a new field "track_info" added that allows to find general information about the current playing audio track, video, station or stream.

12.26 Playlist Player

TBD. Allows to view and manage playback playlist (or queue).

12.27 Positional Player

TBD. Reports the current playback position. Supports track rewinding.

12.28 Speaker

type "speaker"

inherits from none

icon speaker icon

capabilities "actuator", "has_state", "is_active", "on_off", "has_volume", "is_muted"

Speaker is a common device type for all Speakers (sound producing devices) with a single input source. The only thing they can do is to be turned on, off and regulate their volume (i.e. the level of loudness).

Please note that muted devices and devices with a volume set to zero are still considered as "active" devices. So, Speakers are considered to be in "active" state until they are not powered off.

12.29 Speaker System

type "speaker_system"

inherits from none

icon speaker system icon

capabilities "actuator", "has_state", "is_active", "on_off", "has_volume", "multi_source", "is_muted"

Speaker System is a common device type for all sound speakers and speaker systems that have multiple input sources. In addition to the base functionality of a Speaker, such devices allow to view, choose and change the sound source from the list of provided sources.

12.30 Sound System

TBD. Multi-functional device. Can be either music player or a multi-source speaker (i.e. Speaker System) depending on a current mode.

12.31 Display

TBD. Can be turned on, off and change screen brightness.

12.32 Multi-Source Display

TBD. Can change the source of a displayed picture.

12.33 TV

TBD. Multi-mode device which can be either Player, Streaming Player or Multi-Source Display, depending on the current mode.

12.34 Virtual Remote Control

TBD. A device that just provides a list of available commands, a list of corresponding virtual buttons and no feedback from the controlled system.

CHAPTER 13

Indices and tables

- genindex
- modindex
- search