ComfortLife SDK

Release 0.3.1

ComfortLife

May 18, 2022

PREFACE

CHAPTER

ONE

ABOUT COMFORTLIFE

ComfortLife is an extremely flexible and scalable IoT platform designed to enable retailers, service providers, user electronics OEMs, and system integrators to quickly and easily deploy their own connected device ecosystems to maximize business opportunities with IoT.

ComfortLife makes this possible by addressing, at all levels, the challenges inherent in the fast-developing IoT environment, and providing innovative solutions designed with the future in mind. At the forefront of these challenges is solving the problem of multiple protocols, brands, and data sources providing users with app overload when attempting to control different devices. ComfortLife unifies these disparate elements in one platform, easing your product's adoption. In addition, a strong focus on the technology builds intelligent relationships and communication between devices, creating the structure to more easily expand your family of devices.

ComfortLife also addresses time-to-market and mass deployment challenges, simplifying and accelerating chipset enablement and application development, providing solutions that improve video streaming integration in the smart home.

CHAPTER

TWO

CHANGELOG

Based on Keep A ChangeLog Follows Semantic Versioning

2.1 [0.3.1] - 2020/08/31

2.1.1 Changed

- Moved Quick-Start section to top level

2.2 [0.3.0] - 2020/08/05

2.2.1 Added

- Added PDF download for html output
- DevPortal section added
- Quick Start in 5 minutes added
- Changing numbers for 'Quick Start in 5 minutes steps'
- Changing numbers for 'Development' steps

2.2.2 Changed

- Moved changelog and about sections into a new supersection
- Changed versions in order to comply to semver
- Devportal documentation is now up to date with the latest devportal
- Moved sample provisioning and register into their parent steps (1 and 2)

2.3 [0.2.0] - 2020/07/31

2.3.1 Added

- Added a changelog page at the root of the documentation
- Added status, power, voltage and current capabilities
- Added settings action payload

2.3.2 Changed

- Changed changelog to follow Keep A Log

2.4 [0.1.0] - 2020/06/01

2.4.1 Added

- Added logo and favicon
- Added bearer payload documentation
- Added basic usage documentation
- Added payload documentation
- Initialized the documentation

CHAPTER

THREE

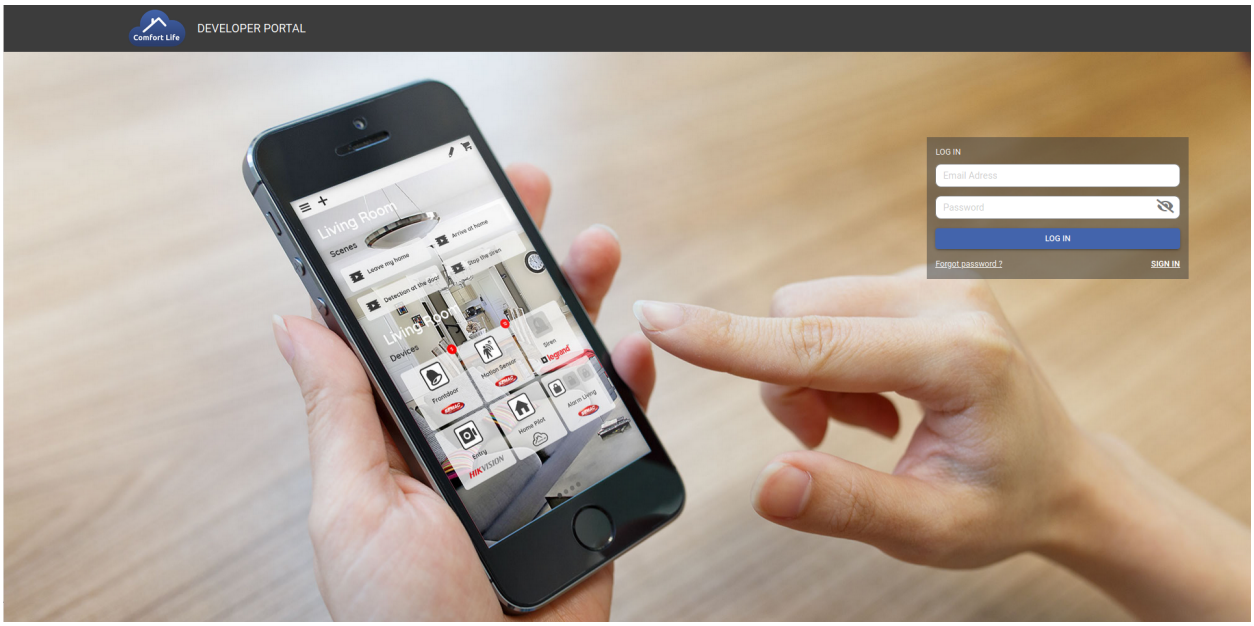
QUICK START IN 5 MINUTES

This topic introduces how to use the DevPortal, and how to create products quickly.

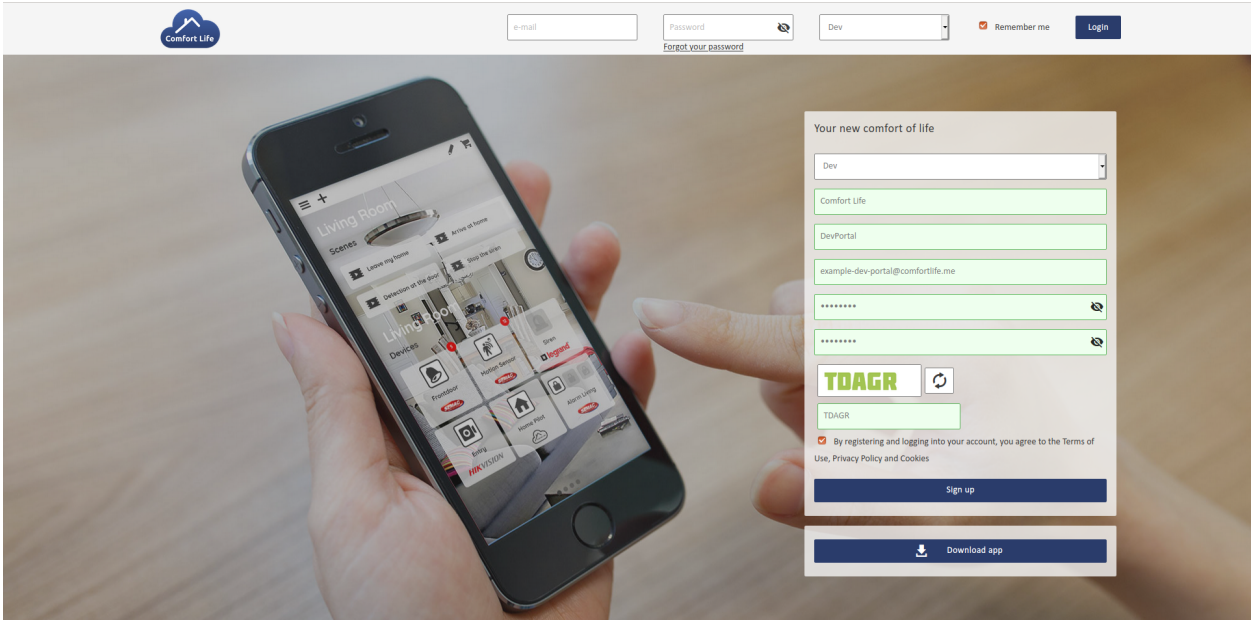
3.1 *Step 1 : Setup environment*

3.1.1 Register to CL Developer Portal

You may go to our website .

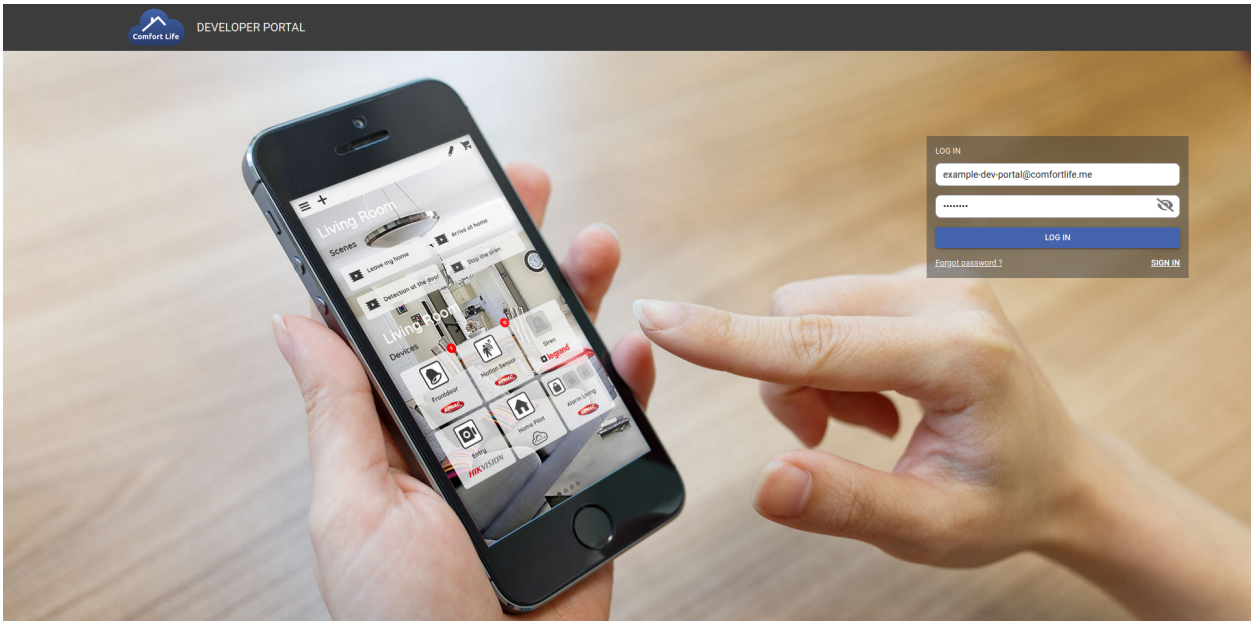


Register your new account on ‘Dev’ server if it is not already done.



(You can also login with a Comfortlife account.)

Please check your email to validate your account and go back on Developer Portal .



Login on the site and accept the privacy

3.2 *Step 2 : Create you new product*

3.2.1 Provisioning Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <cl.h>

void help(const char *app_name);
void provisioning();

int main(int argc, char **argv)
{
    int c;

    while ((c = getopt (argc, argv, "h?")) != -1)
    {
        switch(c)
        {
            case 'h':
            case '?':
            default:
                help(argv[0]);
        }
    }

    provisioning();

    return 0;
}

// ****************************************

void help(const char *app_name)
{
    printf("%s.\n\n", app_name);

    printf("Provisioning example.\n\n");

    printf("Usage: %s [-h|-?]\n\n", app_name);

    printf("   -h          This help\n");

    printf("Examples:\n");
    printf("%s\n", app_name);

    exit(1);
}

void provisioning_status_handler(const cl_provisioning_event_t status)
{
    switch (status)
    {
```

(continues on next page)

(continued from previous page)

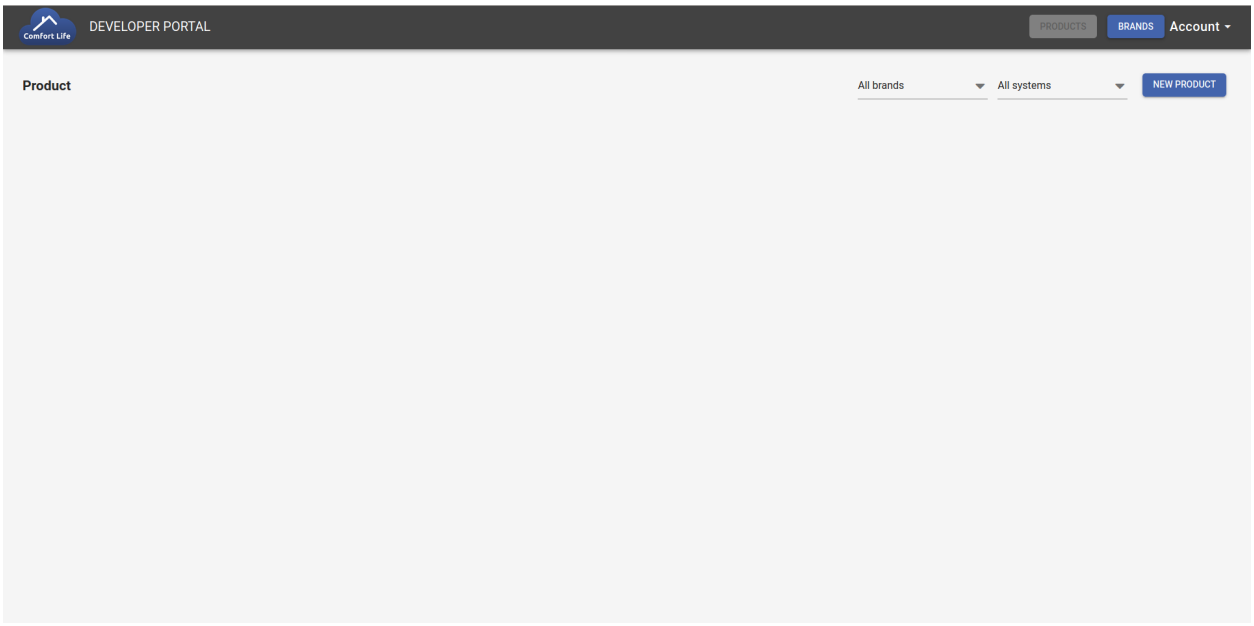
```
        case CL_PROVISIONING_ERROR:
            printf("Provisioning error.\n");
            break;

        case CL_PROVISIONING_BIND_ERROR:
            printf("Bind error.\n");
            break;

        case CL_PROVISIONING_WAITING_CLIENT:
            printf("Waiting a client.\n");
            break;

        case CL_PROVISIONING_CLIENT_CONNECTED:
            printf("Client connected.\n");
            break;

        case CL_PROVISIONING_CLIENT_DISCONNECTED:
            printf("Client disconnected.\n");
            break;

        case CL_PROVISIONING_DATA_RECEIVED:
            printf("Data received.\n");
            break;

        case CL_PROVISIONING_RESPONSE_SENT:
            printf("Response sent.\n");
            break;

        case CL_PROVISIONING_FINISHED:
            printf("Finished.\n");
            break;
    }
}

/*
{"region":"[REGION]","associationToken":"[ASSOCIATION_TOKEN]","ssid":"[SSID]",
↪"password":"[PASSWORD]"}
*/
void provisioning_data_handler(const unsigned char *recvbuf, size_t recvlen)
{
    printf("Data received: %s\n", recvbuf);
}

void provisioning()
{
    cl_return_t result;

    printf("[.] Initialize context...\n");
    strcpy(cl_handle.developer_key, DEVELOPER_KEY);
    strcpy(cl_handle.iot_brand_name, IOT_BRAND_NAME);
    strcpy(cl_handle.iot_model_name, IOT_MODEL_NAME);
    strcpy(cl_handle.iot_submodel_name, IOT_SUBMODEL_NAME);
    strcpy(cl_handle.iot_system, IOT_SYSTEM);

    printf("[.] Initialize...\n");
    result = cl_provisioning_init(&cl_handle);
    switch (result) {
```

(continues on next page)

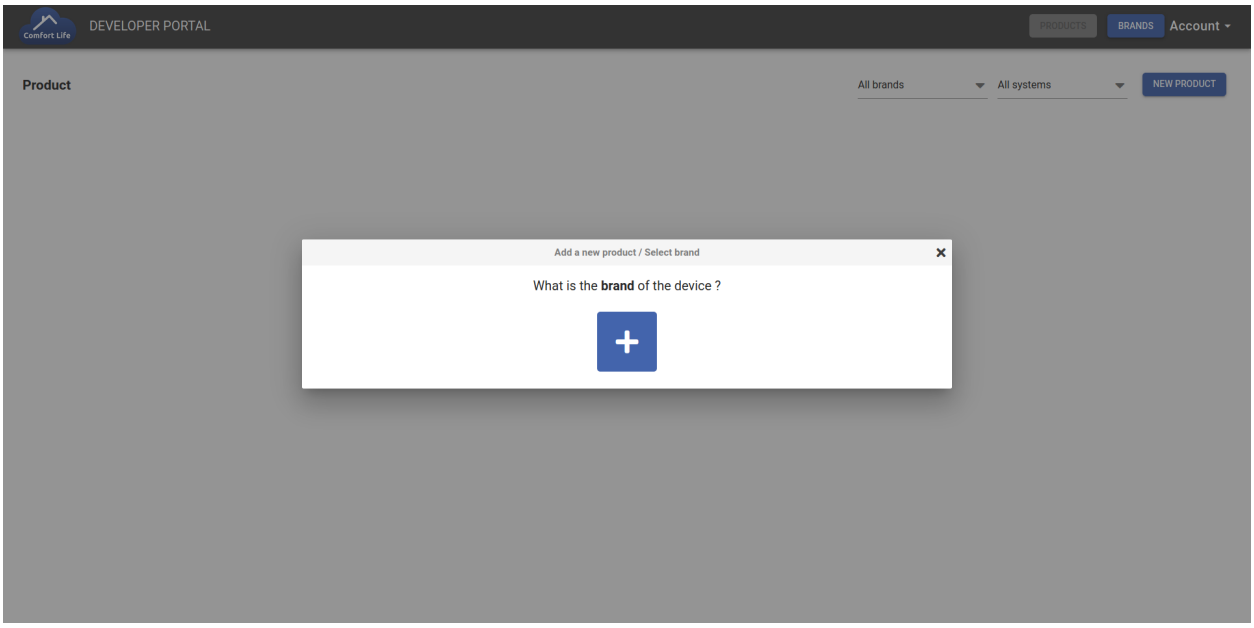
(continued from previous page)

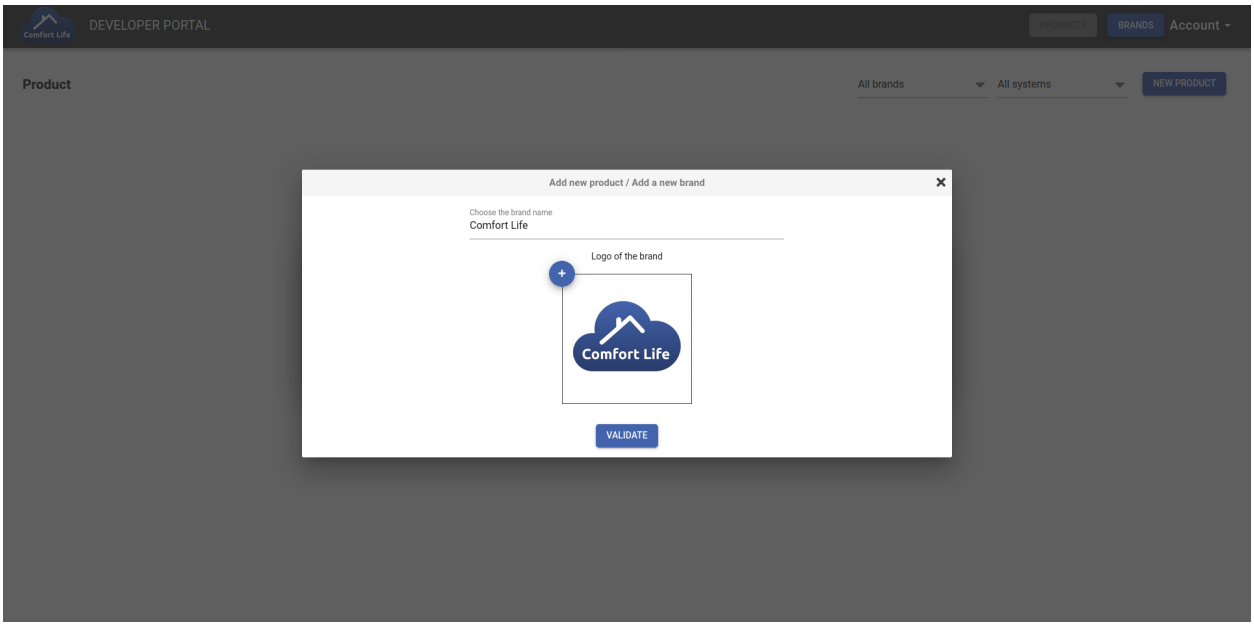
```
        case CL_SUCCESS:
            printf("[+]. Initialization ok.\n");
            break;

        case CL_FAILURE:
        default:
            // OTHER
            printf("[-]. Failed to init the library provisioning.\n");
            break;
    }

    printf("[.]. Initialize callbacks...\n");
    cl_provisioning_get_status(provisioning_status_handler);
    cl_provisioning_get_data(provisioning_data_handler);

    printf("[.]. Start...\n");
    result = cl_provisioning_start();
    switch (result) {
        case CL_SUCCESS:
            printf("[+]. Success.\n");
            break;

        case CL_FAILURE:
        default:
            // OTHER
            printf("[-]. Error during the provisioning.\n");
            break;
    }

    printf("[.]. Deinit...\n");
    result = cl_provisioning_deinit();
    switch (result) {
        case CL_SUCCESS:
            printf("[+]. Deinit ok.\n");
            break;

        case CL_FAILURE:
        default:
            // OTHER
            printf("[-]. Failed to deinit the library provisioning.\n");
            break;
    }
}
```

- Click on **NEW PRODUCT** on the top-right of the screen.



Create and put your logo for your new brand or use your existing one.





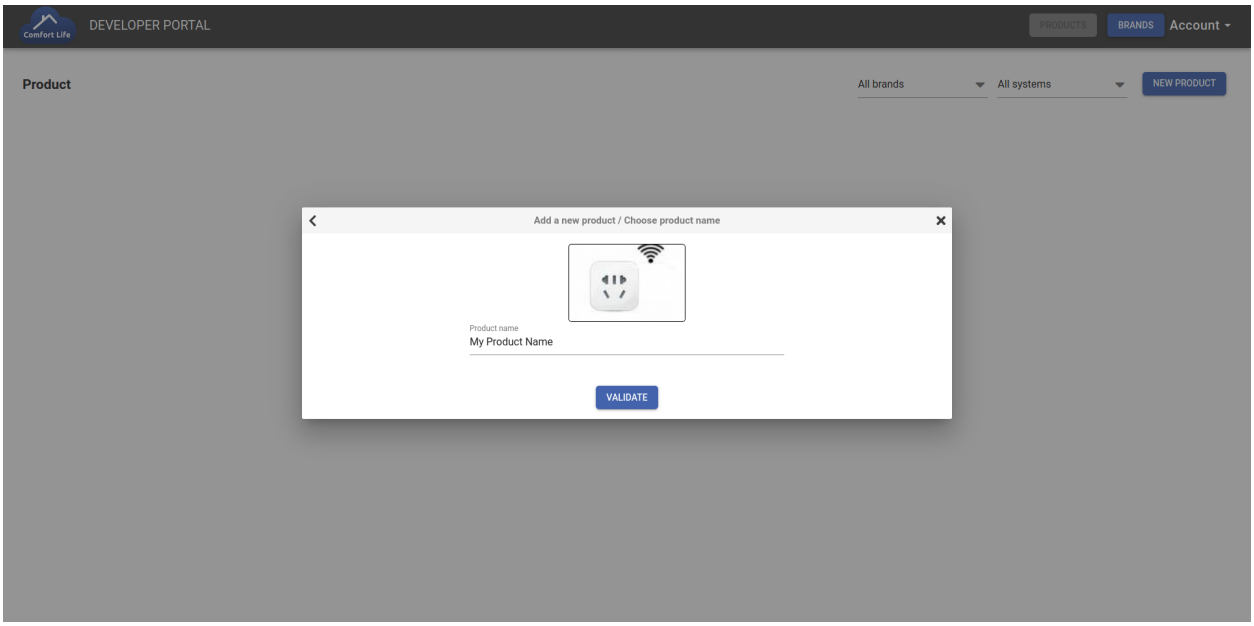
Warning: The name you write for the creation is the one that the users will see. The one that will be used for development will be given to you later.

Click on **VALIDATE** to go to the next step.

- Next, select the Family Type for your new product.

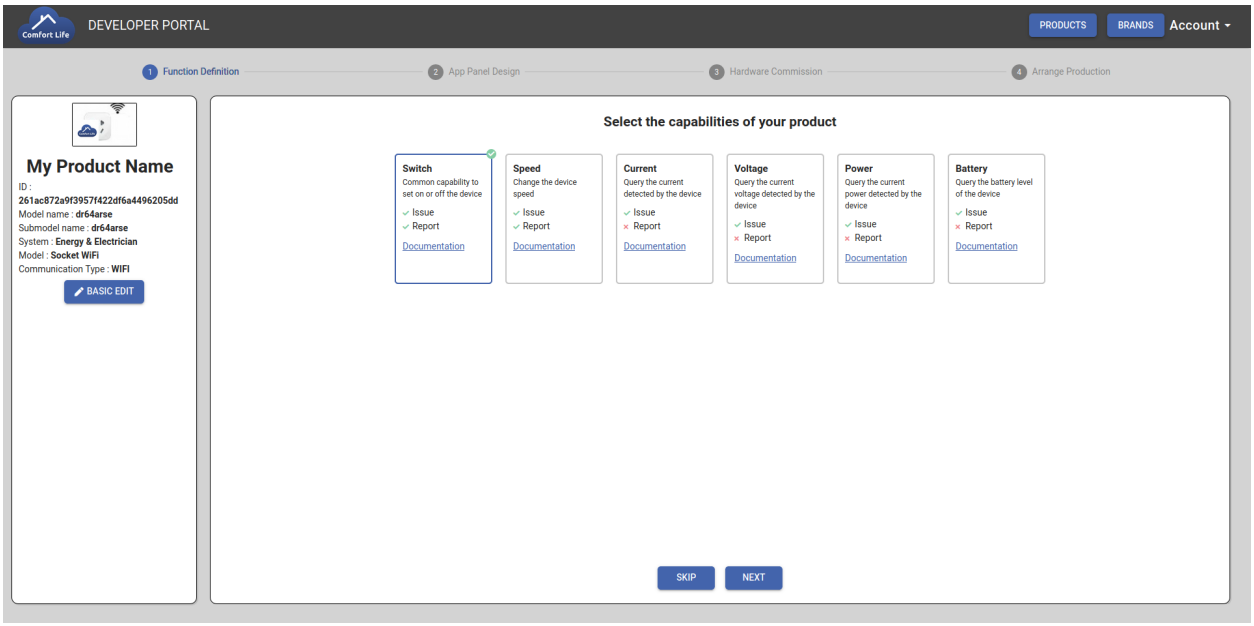
Note: Each Family Type with a '501' image are not yet available but will be later.

- Next, define your product name.



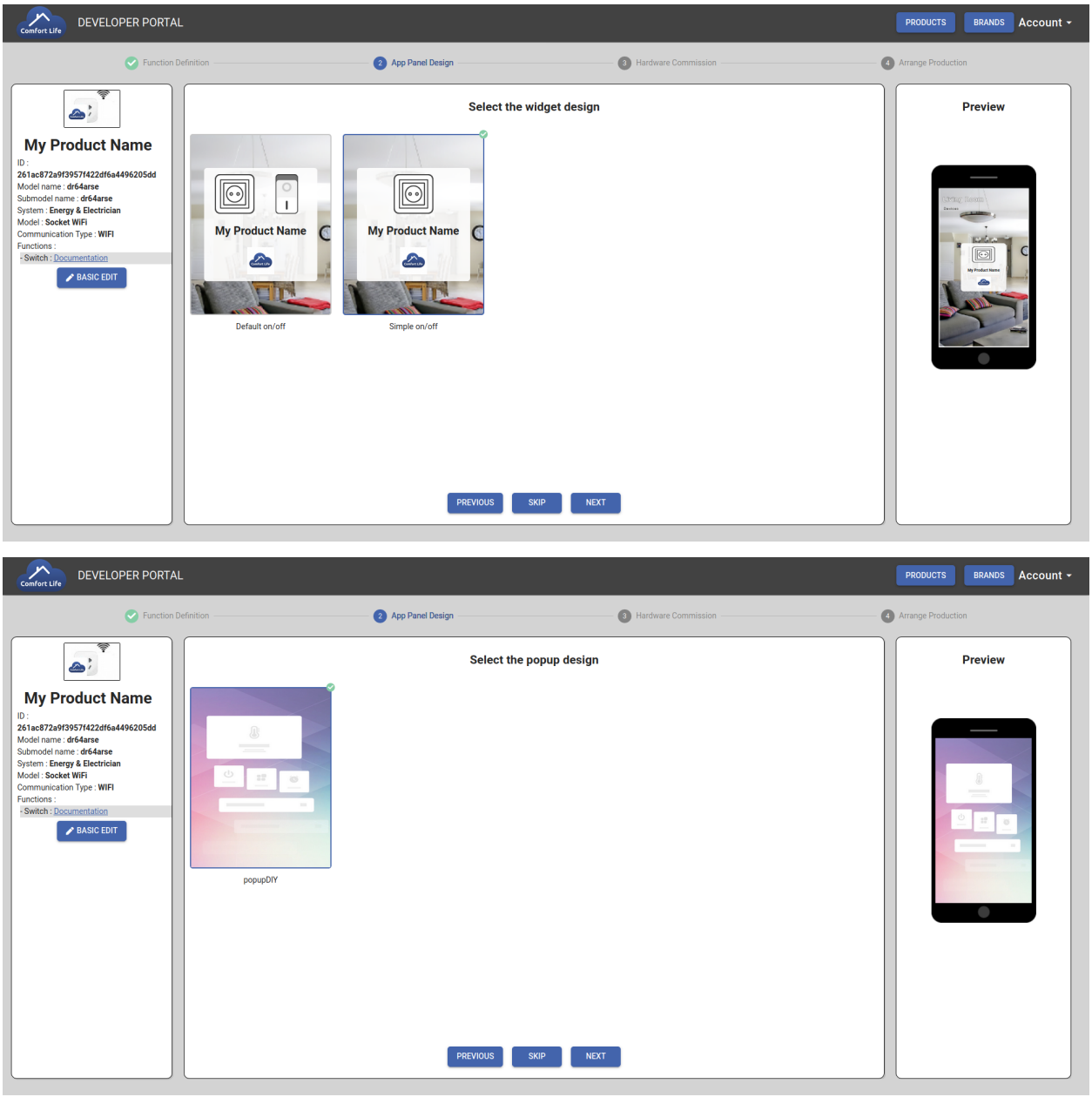
Warning: As for the brand, the name you write here is the one that the users will see. The one that will be used for development will be given to you later.

- Next, choose the functions of your product. You need, at least, one function.



Note: *You can also skip this step. This will not affect the Firmware or the SDK generated later but only the functionality in the ComfortLife Application.*

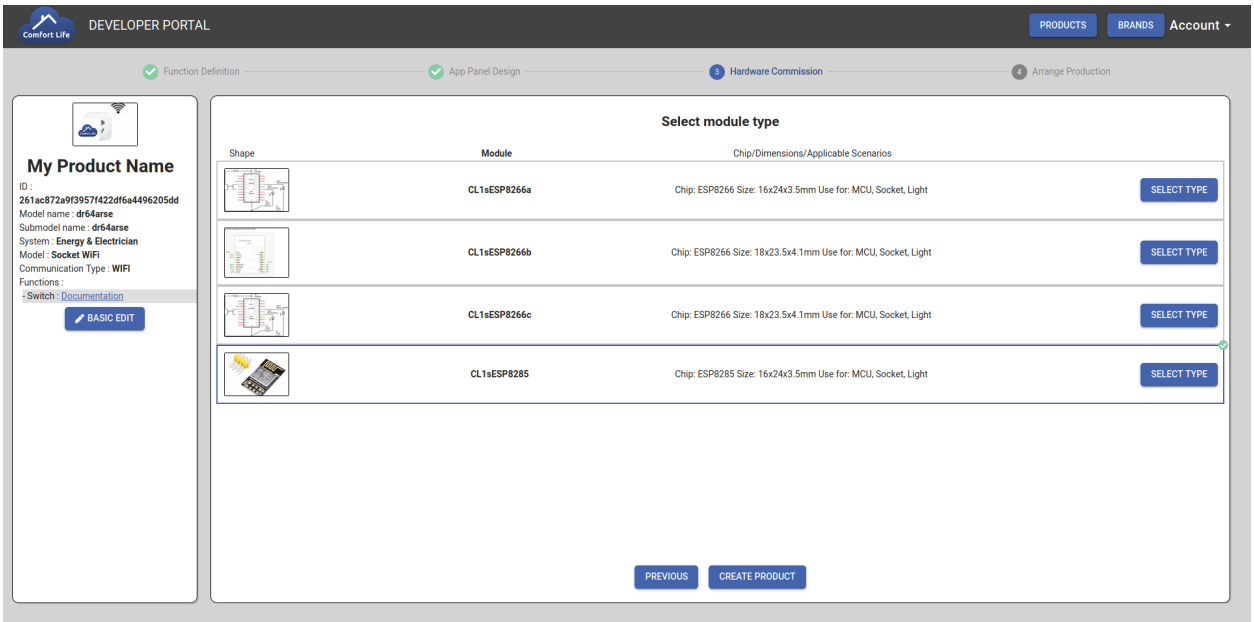
- Next step is the selection of the widget design and the popup design for the ComfortLife application.



Select your preferred choice and go to the next step.

Note: *You can skip this step too. This will not affect the Firmware or the SDK generated later but only the functionnality in the ComfortLife Application.*

- Next, this step is the most important.



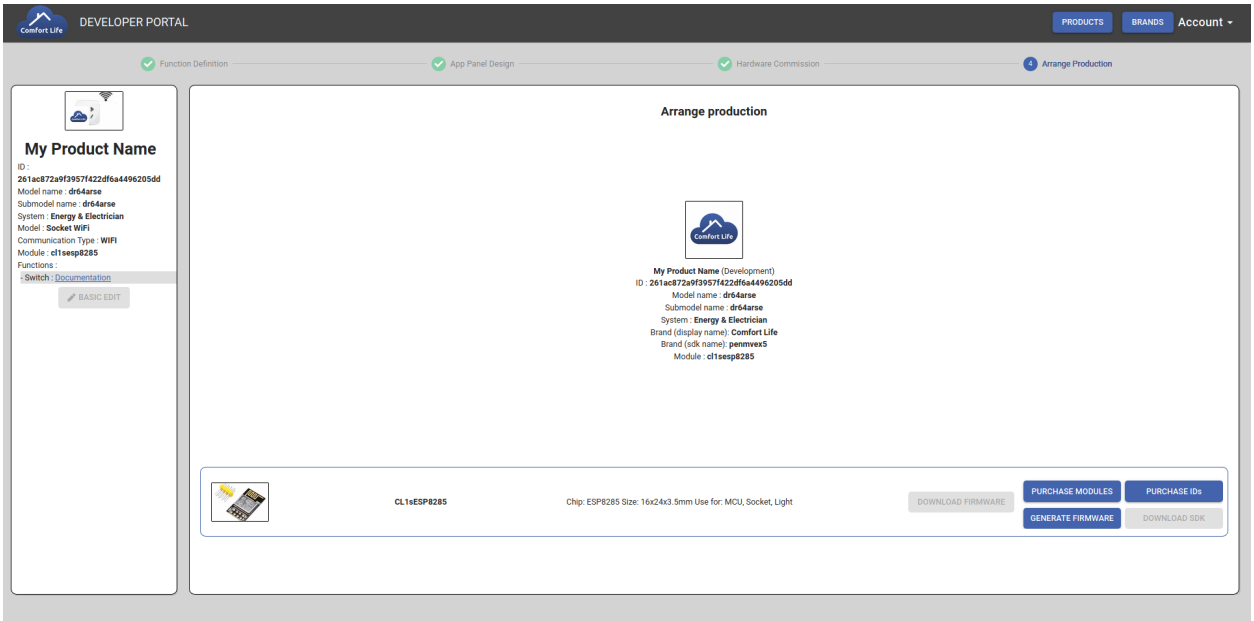
If you don't have a module, select the one that suits you best.

Otherwise, select one with the same chip as yours.

> **Warning:** Consult the overview on the left side of the screen to be sure of your choices. The next step is the creation of your product on our servers and it will not be possible to come back to make a change !

Click on **CREATE PRODUCT**.

- Next, there is the definitive page for your product.

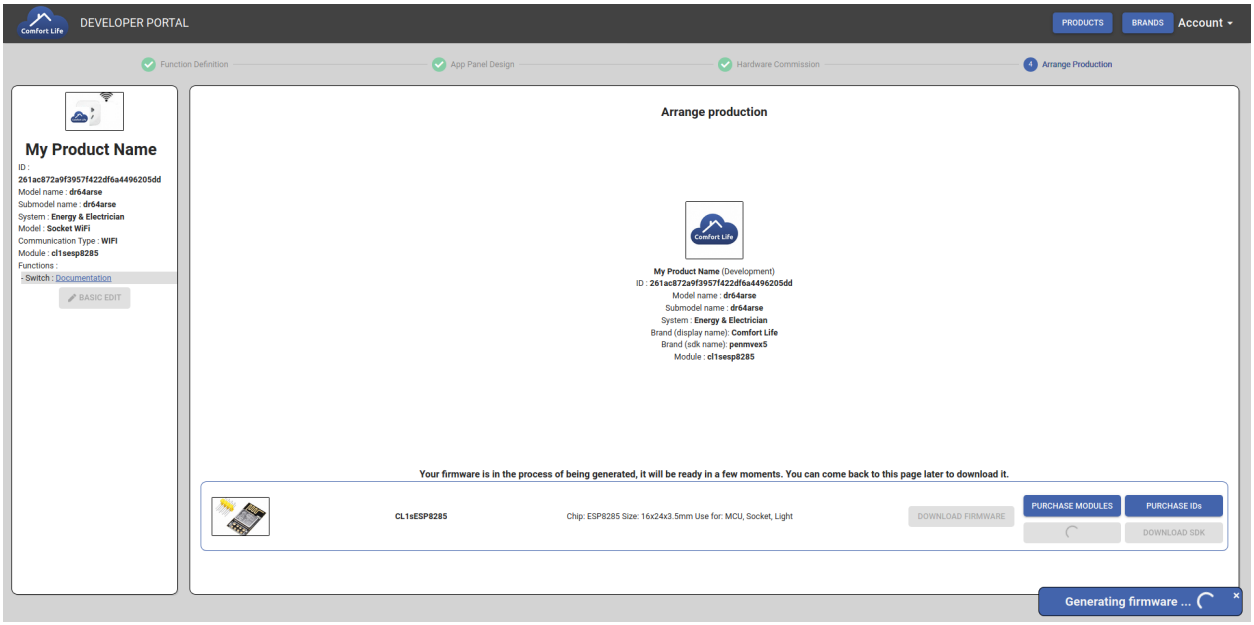


You can here purchase IDs and modules, generate/regenerate and download your firmware, and download our SDK.

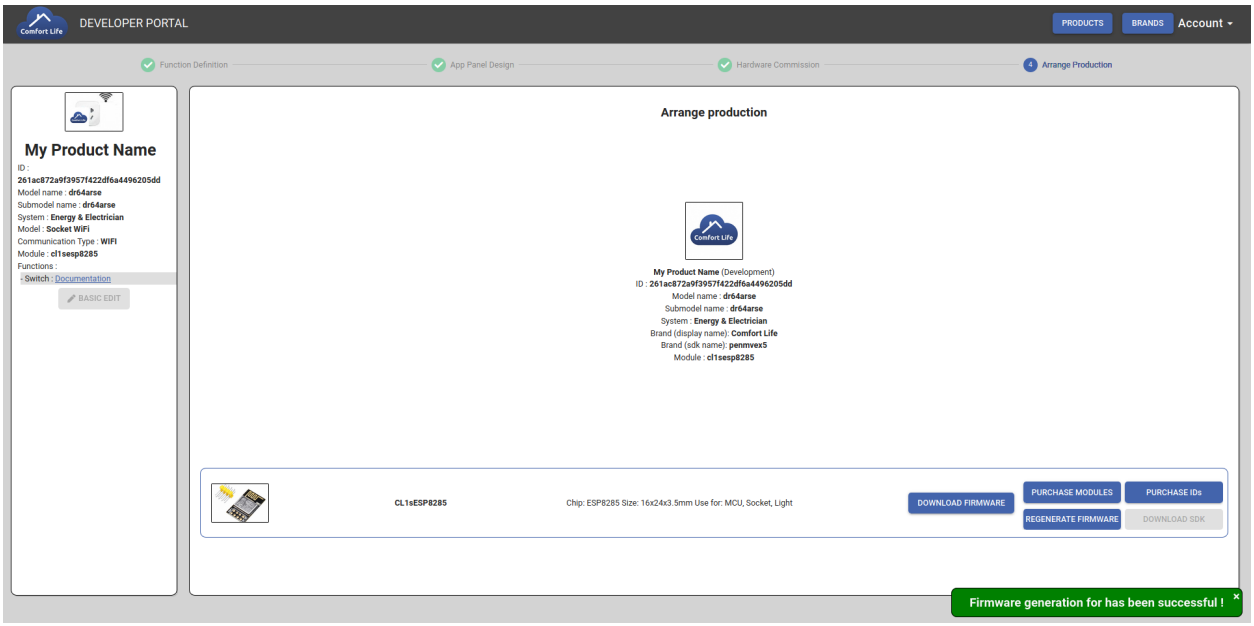
The first step generates the firmware for you. It is custom-made and may take a few moments. You can come back on

this page later in order to download your firmware when it is ready or wait for it.

Note: *(You will see a notification telling you that the firmware is being generated, if not, try to generate it again).*



When it is ready, the download button will no longer be disabled and you can download your firmware.



- After clicking on **DOWNLOAD FIRMWARE**, check it in the download area of your browser.

CHAPTER

FOUR

COMFORTLIFE KIT

The ComfortLife Kit supports establishment of connections of device to cloud, device to device, device to client and client to device. It assists devices in setting up tunnel connections to provide private pathways for transmission via a public network.

The device will send to the Comfort Life platform the informations that you defined on the developer portal

The ComfortLife Kit includes:

- Library files (./lib)
- API definition and declaration (./include)
- Sample codes (./sample)
- Library Dependencies (./deps)
- Toolchains to cross-compile (./toolchains)

4.1 Basic Concepts

4.1.1 Terminology

Server A machine, maintained by ComfortLife, to handle connection among devices and clients

Device An equipment made by a vendor that is capable of ComfortLife Platform integration to enable clients to build connection, even if the device is put behind NAT.

Client A terminal connecting to devices for in-between data to be communicated

Developper Key A 32-chars unique identification key issued by ComfortLife Developer Portal for each developer / manufacturer that requires to use ComfortLife Kit

Account creation available `on the developper portal`_.
Device will use the Developer key to communicate with Server

IOT brand name A name that you have to choose when you declare your new Device on the Developer Portal

Device declaration available `on the developper portal`_.
Device will use the IOT brand name to register to Server

IOT model name An unique name that you have to choose when you declare your new Device on the Developer Portal

Device declaration available `on the developper portal`_.
Device will use the IOT submodel name to register to Server

IOT submodel name An unique name that you have to choose when you declare your new Device on the Developer Portal.

If you have a submodel name different of model name.

Else you can put the same name: submodel name = model n.

Device declaration available `on the developper portal`_.
Device will use the IOT submodel name to register to Server

IOT system The category you need to choose for your Device when you declare it on the Developer Portal

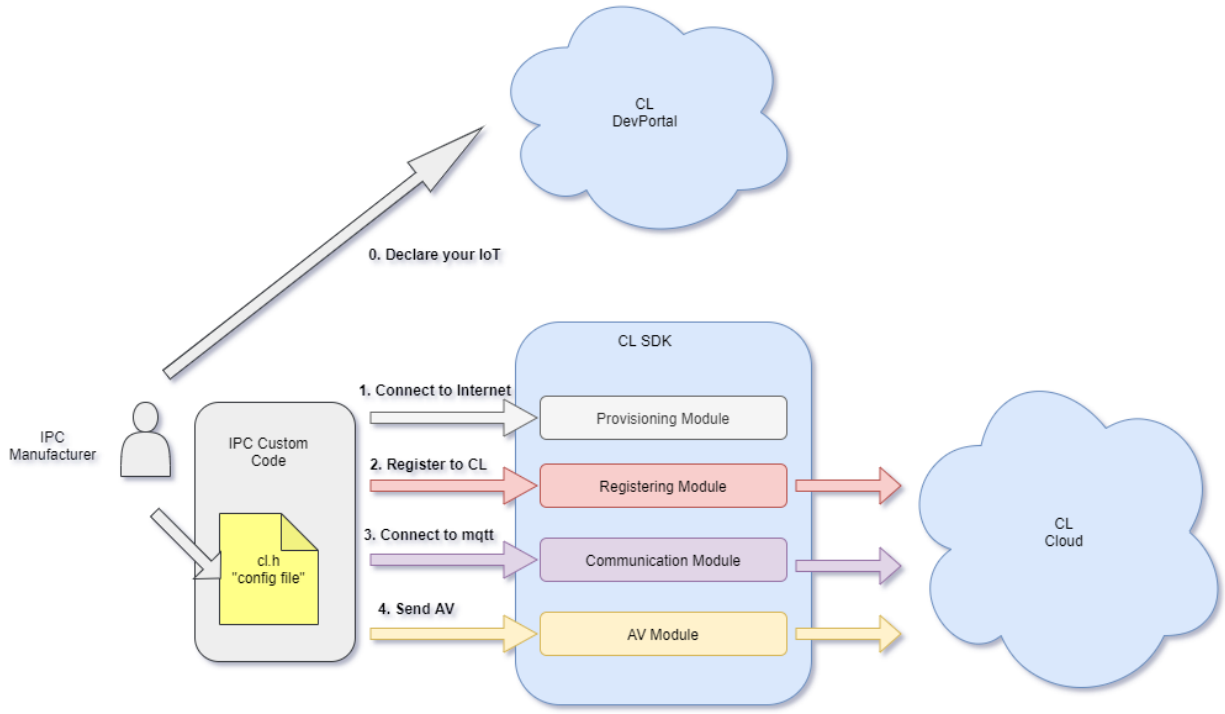
Device will use the IOT system to register to Server

4.1.2 Supported hardware platform

ComfortLife Kit supports several popular development boards

Board name	Library folder
LinkIt 7688	lib/Linux/MIPS_MT7688_4.6.3
Linux x86_64	lib/Linux/x86_64
Hisilicon Hi3516cv300	lib/Linux/arm-hisiv500-linux
Ingenic Xburst V0.1	lib/Linux/mips-gcc472-glibc216-32bit

4.2 General Architecture



4.3 Development

Follow theses steps in order to setup your device on the ComfortLife Platform.

4.3.1 *Step 3 : Register your IoT to CL Cloud*

Register Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <cl.h>

#define VERSION "0.1.0"

#define REGION "na"
#define ASSOCIATION_TOKEN "4c268ad585630e9f"

void help(const char *app_name);
void registration_handler(const unsigned char *str_json, size_t length);
void registration();
```

(continues on next page)

(continued from previous page)

```
int main(int argc, char **argv)
{
    int c;

    while ((c = getopt (argc, argv, "vh?")) != -1)
    {
        switch(c)
        {
            case 'v':
                printf("%s", VERSION);
                exit(1);
                break;

            case 'h':
            case '?':
            default:
                help(argv[0]);
        }
    }

    registration();

    return 0;
}

// ****************************************

void help(const char *app_name)
{
    printf("%s v%s.\n\n", __FILE__, VERSION);

    printf("Register this device to the server.\n\n");

    printf("Usage: %s [-h|-?] [-v]\n\n", __FILE__);

    printf("   -h          This help\n");
    printf("   -v          Get the version\n\n");

    exit(1);
}

/*
{"id":"[DEVICE_ID]","plantId":"[PLANT_ID]","mqtt":{"mqttUsername":"[MQTT_USERNAME]",
↪"mqttPassword":"[MQTT_PASSWORD]"}}
*/
void registration_handler(const unsigned char *str_json, size_t length)
{
    printf("json: %s\n", str_json);
}

void registration()
{
    char json[512];
    cl_return_t result;

    printf("[.]. Initialize context...\n");
    strcpy(cl_handle.developer_key, DEVELOPER_KEY);
```

(continues on next page)

(continued from previous page)

```
    strcpy(cl_handle.iot_brand_name, IOT_BRAND_NAME);
    strcpy(cl_handle.iot_model_name, IOT_MODEL_NAME);
    strcpy(cl_handle.iot_submodel_name, IOT_SUBMODEL_NAME);
    strcpy(cl_handle.iot_system, IOT_SYSTEM);

    // INIT
    printf("[.]. Initialize...\n");
    result = cl_register_init(&cl_handle);
    switch (result) {
        case CL_SUCCESS:
            printf("[+]. Initialization ok.\n");
            break;

        case CL_FAILURE:
        default:
            // OTHER
            printf("[-]. Failed to init the library register.\n");
            break;
    }

    printf("\n");

    printf("[.]. Set the region...\n");
    cl_register_set_region(REGION);

    /*
    {
        "mac": "%s",
        "brand": "%s",
        "submodel": "%s",
        "version": "%s",
        "ip": "%s",
        "associationToken": "%s"
    }
    */
    // REGISTRATION
    printf("[.]. Registration...\n");
    memset(json, 0, sizeof(json));
    sprintf(json, "{ \
                    \"mac\":\"%s\", \
                    \"brand\":\"%s\", \
                    \"submodel\":\"%s\", \
                    \"version\":\"%s\", \
                    \"ip\":\"%s\", \
                    \"associationToken\":\"%s\" \
                    }",
                    "Mac_Ingenic_#01",
                    IOT_BRAND_NAME,
                    IOT_SUBMODEL_NAME,
                    VERSION,
                    "192.168.1.10",
                    ASSOCIATION_TOKEN
                    );
    result = cl_register_registration(json, registration_handler);
    switch (result) {
        case CL_SUCCESS:
            printf("[+]. Registered successfully on backend.\n");
```

(continues on next page)

(continued from previous page)

```
            break;

        case CL_CONNECTION_FAILED:
            printf("[-]. CL_CONNECTION_FAILED.\n");
            break;

        case CL_DEVICE_NOT_FOUND:
            printf("[-]. CL_DEVICE_NOT_FOUND.\n");
            break;

        case CL_FAILURE:
        default:
            // OTHER
            printf("[-]. Unable to register on backend.\n");
            break;
    }

    printf("\n");

    // DEINIT
    printf("[.]. Deinit...\n");
    result = cl_register_deinit();
    switch (result) {
        case CL_SUCCESS:
            printf("[+]. Deinit ok.\n");
            break;

        case CL_FAILURE:
        default:
            // OTHER
            printf("[-]. Failed to deinit the library register.\n");
            break;
    }
}
```

Now that IoT is connected to Internet, next step is to register to CL Cloud so that device can be authenticated and registered to ComfortLife ecosystem.

You will have to use informations received during Provisioning step here

Steps:

- Start **Registering module** with **REGION** and **ASSOCIATION_TOKEN** parameters received during **Provisioning**.
- If registration is successful, **CL CLoud** will return **Device credentials** that you will need to use later : *({“id”:”[DEVICE_ID]”,”plantId”:”[PLANT_ID]”,”mqtt”:{“mqttUsername”:”[MQTT_USERNAME]”,”mqttPassword”:”[MQ*

4.3.2 *Step 4 : Connect to MQTT broker*

Once successfully registered to CL Cloud, you are granted to connect to **CL MQTT Broker** to send informations and receive commands.

You will have to use informations received during Registration step to connect to MQTT Broker

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <cl.h>

#define VERSION "0.1.0"

#define REGION "dev"
#define MQTT_USERNAME "vdp_874707ce2b8c0f0c187f143ee9a0dc14"
#define MQTT_PASSWORD "70dde408c087991673d9"
#define DEVICE_ID "fa8652f4c7569b946db446f51c4502ac"
#define PLANT_ID "plant_db"

void help(const char *app_name);

void mqtt_connection_status_handler(const cl_mqtt_conn_event_t status)
{
    switch (status)
    {
        case CL_MQTT_CONN_CONNECTED:
            printf("Connection etablished.\n");
            break;

        case CL_MQTT_CONN_DISCONNECTED:
            printf("Connection stopped.\n");
            break;
    }
}
void mqtt_data_handler(const unsigned char *str_json, size_t length)
{
    printf("Data received: %s\n", str_json);

    /*
     * Example informations json to send to the server
     * You can send informations any time
    {
      "event": "alert",
      "version": "0.1.0",
      "uid": "[COMMAND_UNIQUE_ID]",
      "modules": [{
        "id": "[DEVICE_ID]",
        "status": [
          {
            "capability": "call",
            "value": "pending"
          }
```

(continues on next page)

(continued from previous page)

```
        ]
      }]
    }*/
    static int command_id = 0;
    char command_id_str[10] = {0};
    sprintf(command_id_str, "%d", ++command_id);
    char data_to_send[1024];
    sprintf(data_to_send,
        "{\"event\":\"%s\",\"version\":\"%s\",\"uid\":\"%s\",\"modules\":[{\"id\":\"
↪%s\",\"status\":[{\"capability\":\"call\",\"value\":\"pending\"}]}]}",
        "alert",
        VERSION,
        command_id_str,
        DEVICE_ID);

    int result = cl_mqtt_send_data(data_to_send);
    switch (result)
    {
        case CL_SUCCESS:
            printf("Data sent.\n");
            break;

        case CL_FAILURE:
        default:
            printf("Data not sent.\n");
            break;
    }
}

int main(int argc, char **argv)
{
    int c;
    while ((c = getopt (argc, argv, "vh?")) != -1)
    {
        switch(c)
        {
            case 'v':
                printf("%s", VERSION);
                exit(1);
                break;

            case 'h':
            case '?':
            default:
                help(argv[0]);
        }
    }

    cl_return_t result;

    // INIT
    cl_mqtt_conn_t mqtt_conn;
    strcpy(mqtt_conn.region, REGION);
    strcpy(mqtt_conn.username, MQTT_USERNAME);
    strcpy(mqtt_conn.password, MQTT_PASSWORD);
    strcpy(mqtt_conn.device_id, DEVICE_ID);
    strcpy(mqtt_conn.plant_id, PLANT_ID);
```

(continues on next page)

(continued from previous page)

```
    printf("[.]. Initialize context...\n");
    strcpy(cl_handle.developer_key, DEVELOPER_KEY);
    strcpy(cl_handle.iot_brand_name, IOT_BRAND_NAME);
    strcpy(cl_handle.iot_model_name, IOT_MODEL_NAME);
    strcpy(cl_handle.iot_submodel_name, IOT_SUBMODEL_NAME);
    strcpy(cl_handle.iot_system, IOT_SYSTEM);

    result = cl_mqtt_init(&cl_handle);
    switch (result) {
        case CL_SUCCESS:
            printf("[+]. Mqtt initialized.\n");
            break;

        case CL_DEVICE_NOT_FOUND:
        case CL_CONNECTION_FAILED:
        case CL_FAILURE:
        default:
            printf("[-]. Failed during initializing mqtt.\n");
            break;
    }

    cl_mqtt_get_connection_status(mqtt_connection_status_handler);
    cl_mqtt_get_data(mqtt_data_handler);

    #if 1
        cl_mqtt_verbose();
    #endif

    // CONNECT
    result = cl_mqtt_connect(mqtt_conn);
    switch (result) {
        case CL_SUCCESS:
            printf("[.]. Connection ended.\n");
            break;

        case CL_DEVICE_NOT_FOUND:
        case CL_CONNECTION_FAILED:
        case CL_FAILURE:
            printf("[-]. Connection failed.\n");
            break;
    }

    // deinit
    cl_mqtt_deinit();

    return 0;
}

// ****************************************

void help(const char *app_name)
{
    printf("%s v%s.\n\n", app_name, VERSION);

    printf("Daemon for the communication with the server.\n");
    printf("This application listens the commands coming from the server.\n\n");
```

(continues on next page)

(continued from previous page)

```
    printf("Usage: %s [-h|-?] [-v]\n\n", app_name);

    printf("   -h          This help\n");
    printf("   -v          Get the version\n\n");

    exit(1);
}
```

4.3.3 *Step 5 : Send Audio/Video stream*

MQTT & UDP payload (previous steps required)

If you want to start a stream through the application, your device will receive the following JSON payload on MQTT and UDP.

```
{
 "action":"command",
 "timestamp":"2020-02-13 14:32:13",
 "uid":"UID",
 "modules":[
     {
         "id":"deviceID",
         "device":{
             "bearerId":"deviceID",
             "status":[
                 {
                     "capability":"start_stream",
                     "value":{
                         // Ignore this part currently
                     }
                 }
             ]
         }
     }
 ],
 "clientId":"applicationId"
}
```

And when you leave the stream on the application you receive the same payload but with *"capability":"stop_stream"*.

Start the audio and/or video stream

Please refer to the samples *./sample/linux/x86_64/av*

1. Initialise the cl_av module with the *cl_av_init* function
2. Create a rtp channel to the IP and port of your choice with the *cl_av_start* function.
3. Send Audio/Video with the *cl_av_send_audio* or *cl_av_send_video*

In order to visualise the stream. You can use the ComfortLife application if you followed the previous steps. Otherwise you can use a tool like VLC (rtp with vlc) to view the stream.

CHAPTER

FIVE

PAYLOAD DOCUMENTATION

This chapter contains the documentation regarding the JSON used to communicate between the devices, the clients and the cloud of the ComfortLife environment.

You'll find in the sections of this document the payloads you will need to use in order to ensure a proper working of your device within the ComfortLife environment.

5.1 Common Payload

5.1.1 Payload

All the other the payloads are based on this common payload.

```
{
  "action": "[ACTION]",
  "event": "[EVENT]",
  "timestamp":"[TIMESTAMP]",
  "uid":"[PAYLOAD_UID]",
  "authorization": "[AUTHORIZATION]",
  "bearerId":"[BEARER_DEVICE_UID]",
  "plantId": "[PLANT_UID]",
  "clientId": "[CLIENT_ID]",
  "modules": [
      {
          "id":"[MODULE_UID]",
          "name":"[MODULE_NAME]",
          "type":"[MODULE_TYPE]",
          "device":{
              "bearerId":"[BEARER_DEVICE_UID]",
              "associateStatus":"[ASSOCIATE_STATUS]",
              "model":"[DEVICE_MODEL_NAME]",
              "oid":"[DEVICE_OID]",
              "bearerId":"[BEARER_DEVICE_UID]",
              "brandIdentifier":"[BRAND_IDENTIFIER]",
              "modelIdentifier":[DEVICE_MODEL_ID],
              "submodel":"[DEVICE_SUBMODEL_NAME]",
              "brand":"[DEVICE_BRAND]",
              "systems":[
                  "[SYSTEM_NAME]",
                  ...
              ],
              "status":[
```

(continues on next page)

(continued from previous page)

```
                    {
                        "capability":"[CAPABILITY_NAME]",
                        "value":"[VALUE]",
                        "index":[INDEX]
                    },
                    ...
                ],
                "uartFrame":"[UART FRAME]",
                "code":"[YCB_CODE]",
                "[SUBDEVICE_NAME]s":[
                    {same payload found in "modules" above},
                    ...
                ],
                "upgrade":{
                    "file": "[FIRMWARE_NAME]",
                    "size": [FIRMWARE_SIZE],
                    "checksum": "[FIRMWARE_CHECKSUM]"
                },
                "localStream":{
                    "ip": "[CAMERA IP]",
                    "port": [CAMERA VIDEO PORT]
                }
            }
        }
    ],
    "status":[ // You may find this status array if the command is targetting no
↪modules in particular - ie any device receiving this should try to execute it
        {
            "capability":"[CAPABILITY_NAME]",
            "value":"[VALUE]",
            "index":[INDEX]
        },
        ...
    ],
    "topic": "[TOPIC]",
    "mqttTopics":{
        "subscribeTo": [
            "topic/to/subscribe/to"
        ],
        "publishTo": [
            "topic/to/publish/to"
        ]
    }
}
```

5.1.2 Description

Here you'll find a description of some of the fields. All the fields described here are optional.

[ACTION] set to specify informations below should be executed. The value is specifying the action type :

- command
- syncScenarios
- executeScenario

- newTopics
- upgrade
- reboot
- get_info

[EVENT] set to specify informations below are the new state of the specified elements. The value is specifying the event type :

- silent (should not notify the end user visually - the only purpose is to update values)
- infos
- alert (should trigger a notification)

[TIMESTAMP] unix time when the payload was generated (with the Y-m-d H:i:s format)

[PAYLOAD_UID] uid identifying the payload

[AUTHORIZATION] uid of the user owning this device

[PLANT_UID] uid of the plant to which this device belongs

[MODULE_UID|NAME|TYPE] uid/name/type of the module

[BEARER_DEVICE_UID] if the device is managed by an other one (a HomePilot for example), here is the uid of the manager

[ASSOCIATE_STATUS] the associate status of the device seen by the sender of the payload. This value can only be one of the following :

- DISSOCIATED
- PENDING
- LEARNING
- ASSOCIATED

[DEVICE_MODEL_NAME] unique name identifying the device model ("ycb", "prf", ... for example)

[DEVICE_MODEL_ID] the unique integer identifying the device model

[DEVICE_SUBMODEL_NAME] the unique name identifying the device submodel ("YC-4000B", "PRF-100", ... for example)

[DEVICE_BRAND] the brand name of the device

[SYSTEM_NAME] the unique name identifying on of the device systems ("energy", "home", ... for example)

[CAPABILITY_NAME] name of the capability

[VALUE] the value associated to the capability

[INDEX] the index of this capability for this device. It can be useful for devices with hidden subdevices OR indexed identical capabilities

[TOPIC]

the topic on which this payload has been published to. It may not appear if not relevant

mqttTopics may not appear

[FIRMWARE_NAME] url of the firmware (http://ip:port/cw/CW-100_0_3_0_PROD for example)

[FIRMWARE_SIZE] size of the firmware

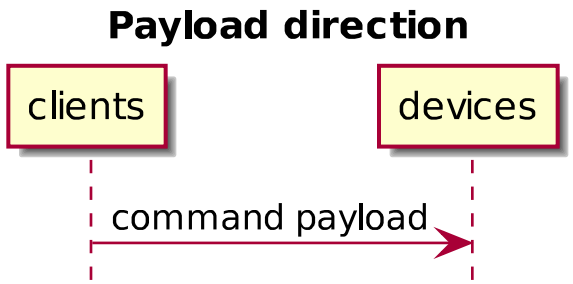
[FIRMWARE_CHECKSUM] checksum of the firmware

5.2 Capabilities

To send command to devices or receive events, you have to use the capabilities. A capability is a keyword a device is configured with to handle different types of actions. For each device model, you can discover available capabilities using this API : https://eu.api.iotcl.pro/v1.0/models/{[}YOUR_MODEL_NAME]

5.2.1 Command

When a device receive this payload, the device has to execute a command related to the capability provided. This payload will be send by the frontend most of the time.



Payload

```
{
 "action": "command",
 "timestamp":"[TIMESTAMP]",
 "uid":"[PAYLOAD_UID]",
 "modules": [
     {
         "id":"[MODULE_UID]",
         "device"|"scenario"|"service":{
             "bearerId":"[BEARER_DEVICE_UID]",
             "status":[
                 {
                     "capability":"[CAPABILITY_NAME]",
                     "value":"[VALUE]" //optional
                 },
                 ...
             ]
         }
     }
 ]
}
```

See *[TIMESTAMP]*, *[PAYLOAD_UID]*, *[MODULE_UID\NAME\TYPE]*, *[BEARER_DEVICE_UID]*, *[CAPABILITY_NAME]*, *[VALUE]*

Some available capabilities

start_stream

The `start_stream` capability is sent when a user wants to watch the camera on a device through the ComfortLife application. The `value` field of this payload will be:

```
{
      "capability":"start_stream",
      "value":
      {
              "devicePeerName": "[DEVICE PEER NAME RANDOM UID]",
              "peerName": "[USER/LOCAL PEER NAME RANDOM UID]",
              "videoPort": [VIDEO PORT],
              "audioPort": [AUDIO PORT]
              "password": "[RANDOM PASSWORD]",
              "mediatorHost": "[MEDIATOR HOST]",
              "mediatorPort": [MEDIATOR PORT],
              "relayHost": "[RELAY HOST]",
              "relayPort": [RELAY PORT]
      }
}
```

stop_stream

The `stop_stream` capability is sent when a user stops the video stream on the ComfortLife application. The `value` field of the payload is:

```
{
 "capability":"stop_stream",
 "value":
  {
   "peerName": "[USER/LOCAL PEER NAME RANDOM UID]"
  }
}
```

Future capabilities

resolution

This capability is used to set the video stream resolution. Currently the `value` field can take 3 values : `low`, `medium` or `high`

mirror and flip

The `value` field sets the mirror or flip mode to `enable` or `disable`

bitrate

The `value` field contains an integer that sets the bitrate of the video stream.

ircut

Sets the ircut mode of the device to `enable`, `disable` or `auto`.

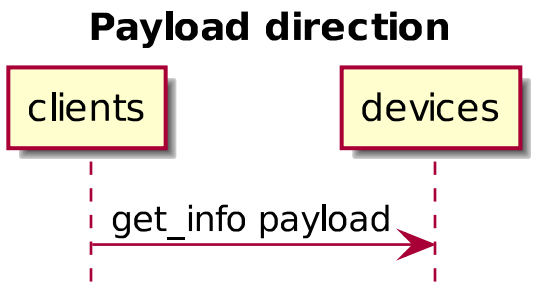
speaker_volume

Sets the speaker volume of the device. The value ranges between 0 and 100.

detection

`enable` or `disable` the pir detection

5.2.2 Get Info



The following payload is sent to request the targeted device to send its current capabilities values

Payload

```
{
   "action":"get_info",
   "timestamp":"[TIMESTAMP]",
   "uid":"[PAYLOAD_UID]",
   "modules": [
       {
           "id":"[MODULE_UID]",
           "device"|"scenario"|"service":{

           }
       }
```

(continues on next page)

(continued from previous page)

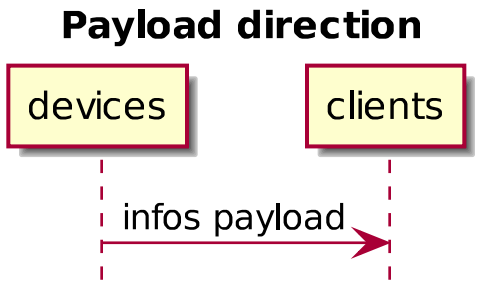
```
    ]
}
```

See *[TIMESTAMP]*, *[PAYLOAD_UID]*, *[MODULE_UID|NAME|TYPE]*

Then the device should respond with the Infos payload

5.2.3 Infos

The following payload is indicating new capability values for the specified device or scenario. You might use the "device" key most of the time. Use the "infos" value for the "event" key by default. If you the information is important to be notified to the end user (such as a device call, for example), replace the value by "alert"



Payload

```
{
 "event": "infos"|"alert"|"silent",
 "timestamp":"[TIMESTAMP]",
 "uid":"[PAYLOAD_UID]",
 "modules": [
     {
         "id":"[MODULE_UID]",
         "device"|"scenario"|"service":{
             "bearerId":"[BEARER_DEVICE_UID]",
             "associateStatus":"ASSOCIATED"|"PENDING",
             "status":[
                 {
                     "capability":"[CAPABILITY_NAME]",
                     "value":"[VALUE]"
                 },
                 ...
             ]
         }
     }
     ]
}
```

See *[TIMESTAMP]*, *[PAYLOAD_UID]*, *[MODULE_UID|NAME|TYPE]*, *[BEARER_DEVICE_UID]*, *[ASSOCIATE_STATUS]*, *[CAPABILITY_NAME]*, *[VALUE]*

5.2.4 Reboot payload

This payload is sent when someone wants to reboot the device.

Payload

```
{
 "action": "reboot",
 "timestamp":"[TIMESTAMP]",
 "uid":"[PAYLOAD_UID]",
 "modules": [
     {
         "id":"[MODULE_UID]",
         "device":{
             "bearerId":"[BEARER_DEVICE_UID]"
         }
     }
 ]
}
```

See *[TIMESTAMP]*, *[PAYLOAD_UID]*, *[MODULE_UID\|NAME\|TYPE]*, *[BEARER_DEVICE_UID]*

5.2.5 Upgrade payload

This payload is sent when the device must receive a firmware upgrade. The device must get the firmware through the `file` field.

Payload

```
{
   "action": "upgrade",
   "timestamp":"[TIMESTAMP]",
   "uid":"[PAYLOAD_UID]",
   "modules": [
       {
           "id":"[MODULE_UID]",
           "device":{
               "bearerId":"[BEARER_DEVICE_UID]"
                               "upgrade":{
                   "file": "[FIRMWARE_NAME]",
                   "size": [FIRMWARE_SIZE],
                   "checksum": "[FIRMWARE_CHECKSUM]"
               },
           }
       }
   ]
}
```

See *[TIMESTAMP]*, *[PAYLOAD_UID]*, *[MODULE_UID\|NAME\|TYPE]*, *[BEARER_DEVICE_UID]*, [FIRMWARE_NAME], [FIRMWARE_SIZE], [FIRMWARE_CHECKSUM]

5.2.6 Status, power, voltage and current

Capabilities used by electrical devices to log their consumption.

Payload

```
 "modules": [
  {
    "id": "[MODULE_UID]",
    "device": {
      "bearerId": "[BEARER_DEVICE_UID]",
      "status": [
        {
          "capability": "status",
          "value": "off"
        },
        {
          "capability": "power",
          "value": 0
        },
        {
          "capability": "voltage",
          "value": 0
        },
        {
          "capability": "current",
          "value": 0
        }
      ]
    }
  }
]
```

See *[BEARER_DEVICE_UID]*, *[MODULE_UID\|NAME\|TYPE]*

5.2.7 Action - settings

Paylaod for setting WiFi

```
{
 "action": "settings",
 "modules": [
   {
     "id": "[MODULE_UID]",
     "device": {
       "wifi": {
         "ssid": "SSID",
         "pwd": "password"
       }
     }
   }
 ]
}
```

See *[MODULE_UID\|NAME\|TYPE]*

5.3 Bearer device

A bearer device is a device that can manage other devices. In other words, it can forward payloads to them. In the payloads below, the BEARER_DEVICE_UID value refers to the bearer device uid while the MODULE_UID value refers to the managed device

[DEVICE_IDENTIFIER] unique identifier for this device on the bearer side

[DEVICE_TYPE] type of the device on the bearer side

[BRAND_IDENTIFIER] unique brand identifier in hexadecimal format

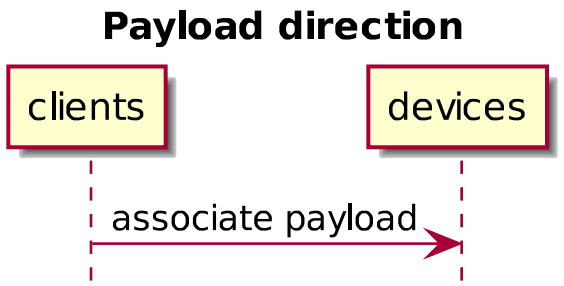
[BRAND_NAME] unique brand name key to identify this brand

[SUBMODEL_IDENTIFIER] unique submodel identifier in hexadecimal format (among other submodels in the same brand)

[SUBMODEL_NAME] unique submodel name key to identify this submodel. Most of the time, the submodel name is identical to its model name

5.3.1 Associate

To associate a device to be managed by a bearer, use the following payload. However such association needs to be validated (see "validate" below) :



Payload

```
{
 "action": "associate",
 "timestamp":"[TIMESTAMP]",
 "uid":"[PAYLOAD_UID]",
 "modules": [
     {
         "id":"[MODULE_UID]",
         "device":{
             "identifier":"[DEVICE_IDENTIFIER]", // optional
             "type":"[DEVICE_TYPE]", // optional
             "bearerId":"[BEARER_DEVICE_UID]",
             "brandIdentifier":"[BRAND_IDENTIFIER]", // optional
             "brandName":"[BRAND_NAME]", // optional
             "modelIdentifier":[DEVICE_MODEL_ID], // optional
             "modelName":"[DEVICE_MODEL_NAME]", // optional
             "submodelIdentifier":"[SUBMODEL_IDENTIFIER]", // optional
             "submodelName":"[SUBMODEL_NAME]", // optional
```

(continues on next page)

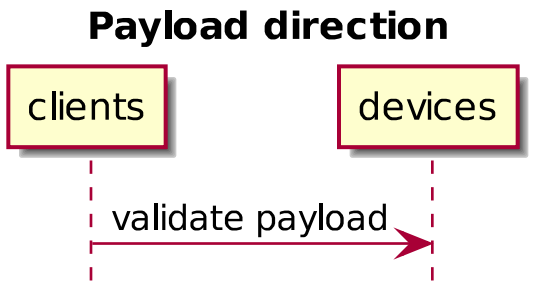
(continued from previous page)

```
            "code":"[YCB_CODE]", // optional
            "[SUBDEVICE_NAME]s":[
                {same payload found in "modules" above},
                ...
            ]
        }
    }
 ]
}
```

See *[TIMESTAMP]*, *[PAYLOAD_UID]*, *[BEARER_DEVICE_UID]*, *[DEVICE_MODEL_ID]*, *[DEVICE_MODEL_NAME]*, *[DEVICE_IDENTIFIER]*, *[DEVICE_TYPE]*, *[BRAND_IDENTIFIER]*, *[BRAND_NAME]*, *[SUBMODEL_IDENTIFIER]*, *[SUBMODEL_NAME]*

5.3.2 Validate

Once a device is successfully associated to their bearer (see "associate" above), it needs to be validated to be usable



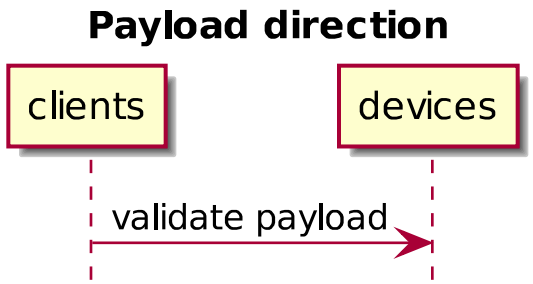
Payload

```
{
 "action": "validate",
 "timestamp":"[TIMESTAMP]",
 "uid":"[PAYLOAD_UID]",
 "modules": [
     {
         "id":"[MODULE_UID]",
         "device":{
             "bearerId":"[BEARER_DEVICE_UID]",
             "code":"[YCB_CODE]",
             "[SUBDEVICE_NAME]s":[
                 {same payload found in "modules" above},
                 ...
             ]
         }
     }
 ]
}
```

See *[TIMESTAMP]*, *[PAYLOAD_UID]*, *[BEARER_DEVICE_UID]*

5.3.3 Remove

You can remove a managed device from their bearer using such payload



Payload

```
{
 "action": "remove",
 "timestamp":"[TIMESTAMP]",
 "uid":"[PAYLOAD_UID]",
 "modules": [
     {
         "id":"[MODULE_UID]",
         "device":{
             "bearerId":"[BEARER_DEVICE_UID]"
         }
     }
 ]
}
```

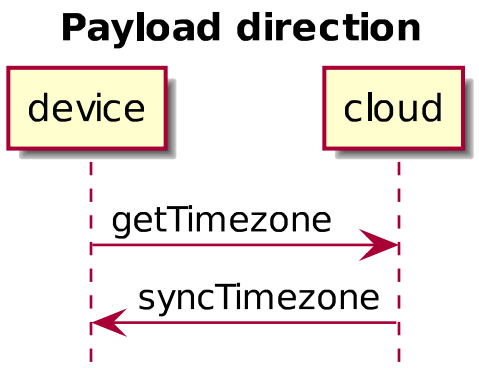
See *[TIMESTAMP]*, *[PAYLOAD_UID]*, *[BEARER_DEVICE_UID]*

5.4 Synchronization

The following payloads are used to ensure the device has synchronized some of their information with the cloud

5.4.1 Synchronizing Timezones

In order to set its timezone a device must send a `getTimezone` to the backend. Then it will answer with a `syncTimezone` payload.



Get Timezone

Payload

```
 {
"event": "getTimezone",
"timestamp": "[TIMESTAMP]",
"uid": "[PAYLOAD_UID]",
"modules": [
  {
    "id": "[MODULE_UID]",
    "device": {}
  }
]
}
```

See *[TIMESTAMP]*, *[PAYLOAD_UID]*, *[MODULE_UID|NAME|TYPE]*

Sync Timezone

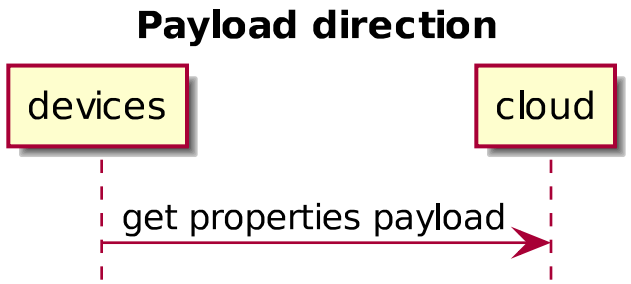
Payload

```
{
 "action":"syncTimezone",
 "timestamp":"[TIMESTAMP]",
 "uid": "[PAYLOAD_UID]",
     "modules": [
     {
          "id":"[MODULE_UID]",
          "device":{
              "timezone": "[TIMEZONE]",
              "timezoneOffset": [TIMEZONE_OFFSET], # integer value for the offset in␣
↪seconds from UTC. For example Asia/Shanghai is -28800
              "shortTimezone":"[SHORT_TIMEZONE]"
                      }
              }
     ]
}
```

See *[TIMESTAMP]*, *[PAYLOAD_UID]*, *[MODULE_UID|NAME|TYPE]*

5.4.2 Get Properties

Request for a sync properties from the cloud. Like the sync timezone the device has to send a `getProperties` payload and wait for `syncProperties` payload.

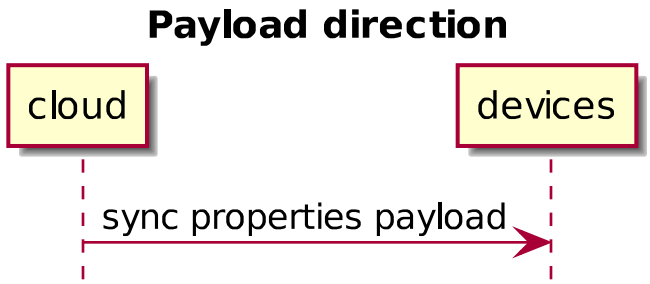


Payload

```
{
  "event": "getProperties",
  "uid": "[PAYLOAD_UID]",
  "timestamp": "[TIMESTAMP]",
  "modules": [
    {
      "id": "[MODULE_UID]",
      "device": null
    }
  ]
}
```

5.4.3 Sync properties

The cloud is requesting target devices to update their properties information. A property is a configuration of the device. For any gateway device (ie a device managing others), you should expect to get the `bearerId` and the `associatedDevices` keys. The latter is listing all devices managed by this gateway as well as their own properties.



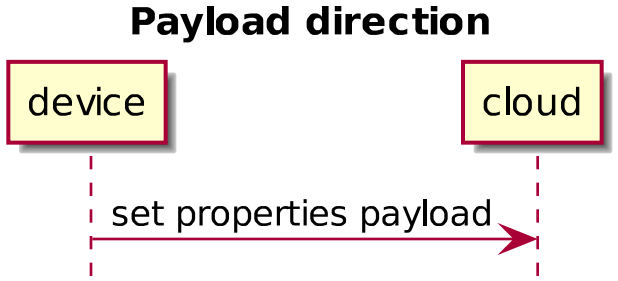
Payload

```
{
   "action":"syncProperties",
   "timestamp":"[TIMESTAMP]",
   "uid": "[PAYLOAD_UID]",
   "modules": [
       {
           "id":"[MODULE_UID]",
           "device":{
               "properties": {
                   "parentLocationId":"[LOCATION_UID]",
                   "key1":"value1",
                   "key2":"value2",
                   ...
               },
               "bearerId":"[BEARER_DEVICE_UID]",
               "associatedDevices":[
                   {
                       "id":"[MODULE_UID]",
                       "device":{
                           "properties":{
                               "parentLocationId":"[LOCATION_UID]",
                               "key1":"value1",
                               "key2":"value2",
                               ...
                           }
                       }
                   }
               ]
           }
       }
   ]
}
```

See *[TIMESTAMP]*, *[PAYLOAD_UID]*, *[BEARER_DEVICE_UID]*, *[MODULE_UID\|NAME\|TYPE]*

5.4.4 Set properties

The device sets its properties to the cloud. Use this if you want to store unique information about your device to be retrieved later from a `syncProperties` payload. When this command is successful, the device should expect an incoming `syncProperties` payload to validate the change.



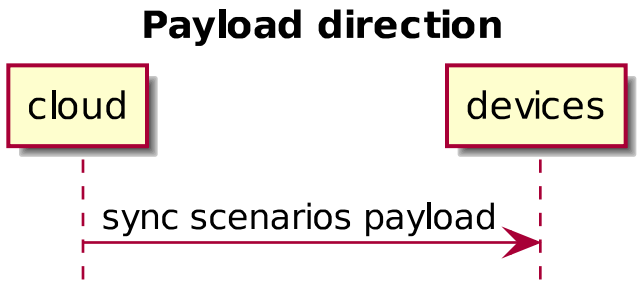
Payload

```
{
   "action":"setProperties",
   "timestamp":"[TIMESTAMP]",
   "uid": "[PAYLOAD_UID]",
   "modules": [
       {
           "id":"[MODULE_UID]",
           "device":{
               "properties": {
                   "foo":"bar",
                   "my-custom-property-key":"associated-value",
                   "property-to-remove":null,
                   ...
               }
           }
       }
   ]
}
```

See *[TIMESTAMP]*, *[PAYLOAD_UID]*, *[MODULE_UID|NAME|TYPE]*

5.4.5 Sync Scenarios

The cloud is requesting the devices to synchronize their scenario information. This feature can not be implemented by all devices.




```
{
  "action":"syncScenarios",
  "timestamp":"[TIMESTAMP]",
  "uid": "[PAYLOAD_UID]"
}
```

INDEX

Symbols

C

D

I

S