EXOSITE

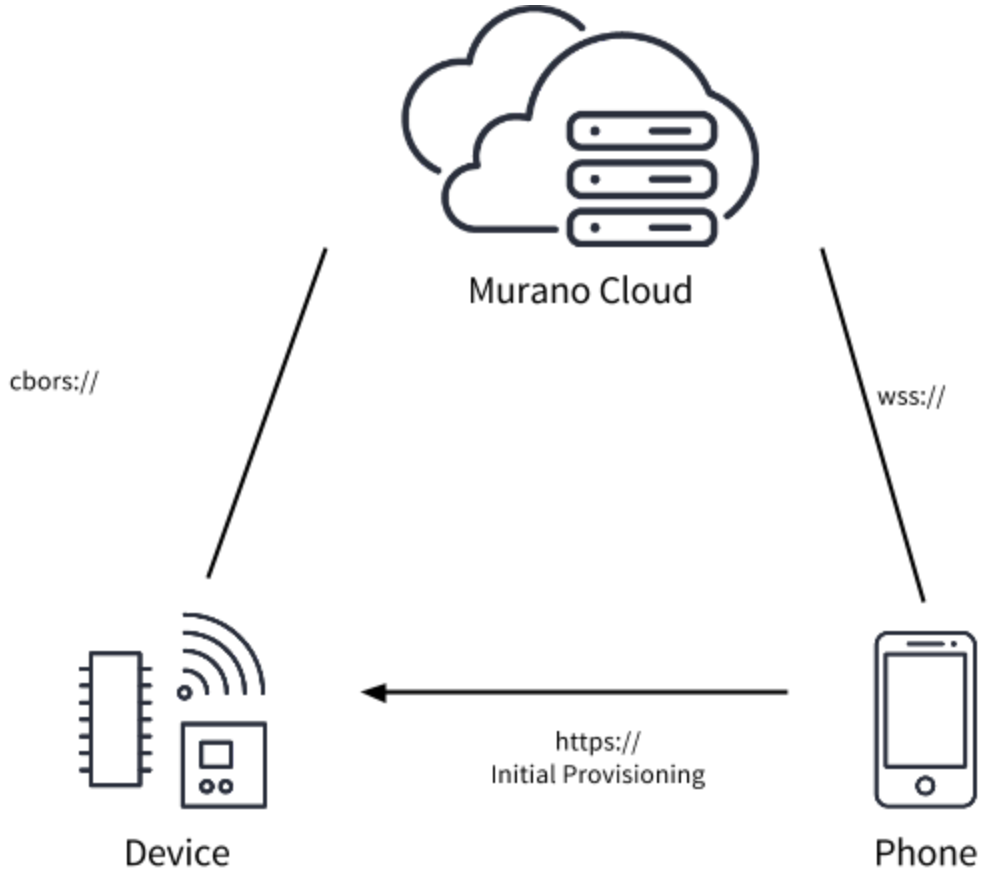
Chungear Demo Application Specification

Data Architecture Specification - DAS

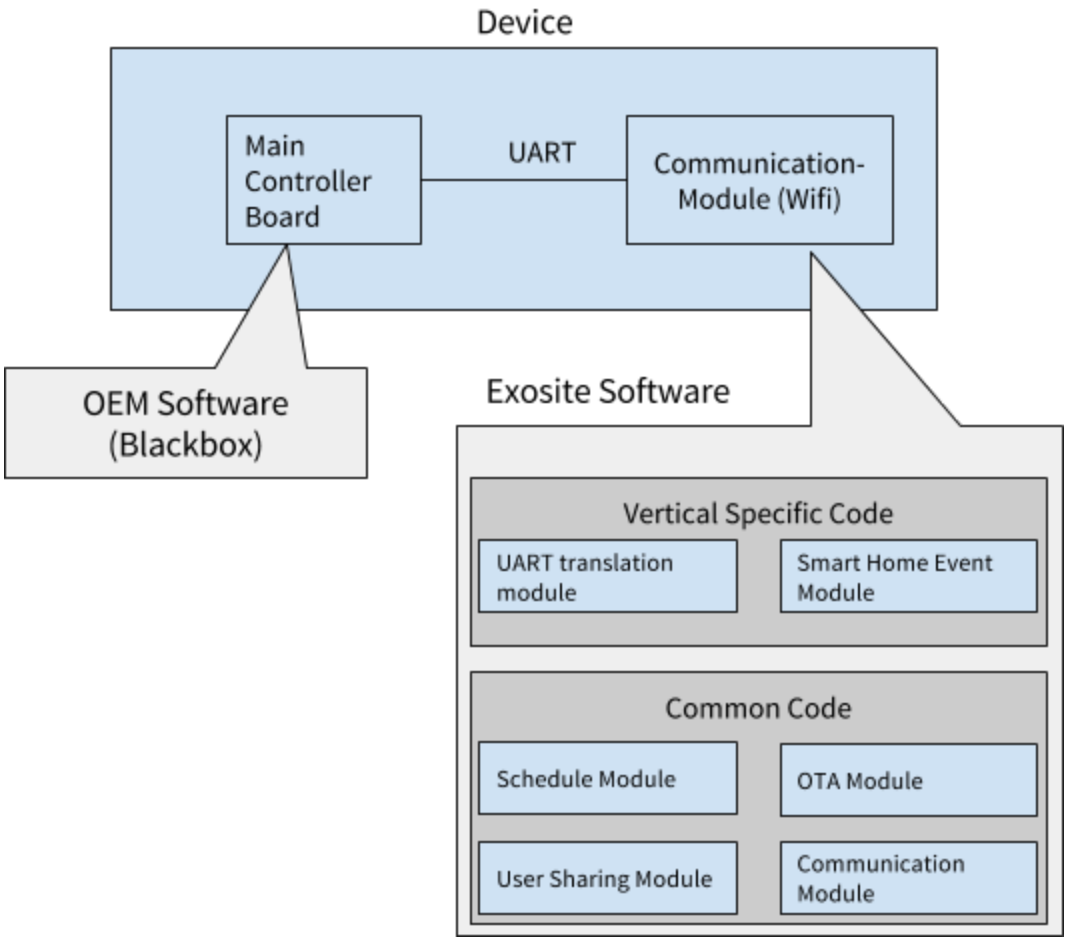
Revision C
25 October 2016

Date	Revision	Description	Author / Editor	Reviewer
14 Oct 2016	A	1st draft	DL	DL
25 Oct 2016	B	Updates based on the kick-off meeting decisions	SC, JY, GC	DL
28 Oct 2016	C	H04 updated to countdown timer on and H05 for countdown timer off	SC	-

1 Overview


Murano Cloud
cbors://
wss://
https://
Initial Provisioning
Device
Phone

1.1 Device Details

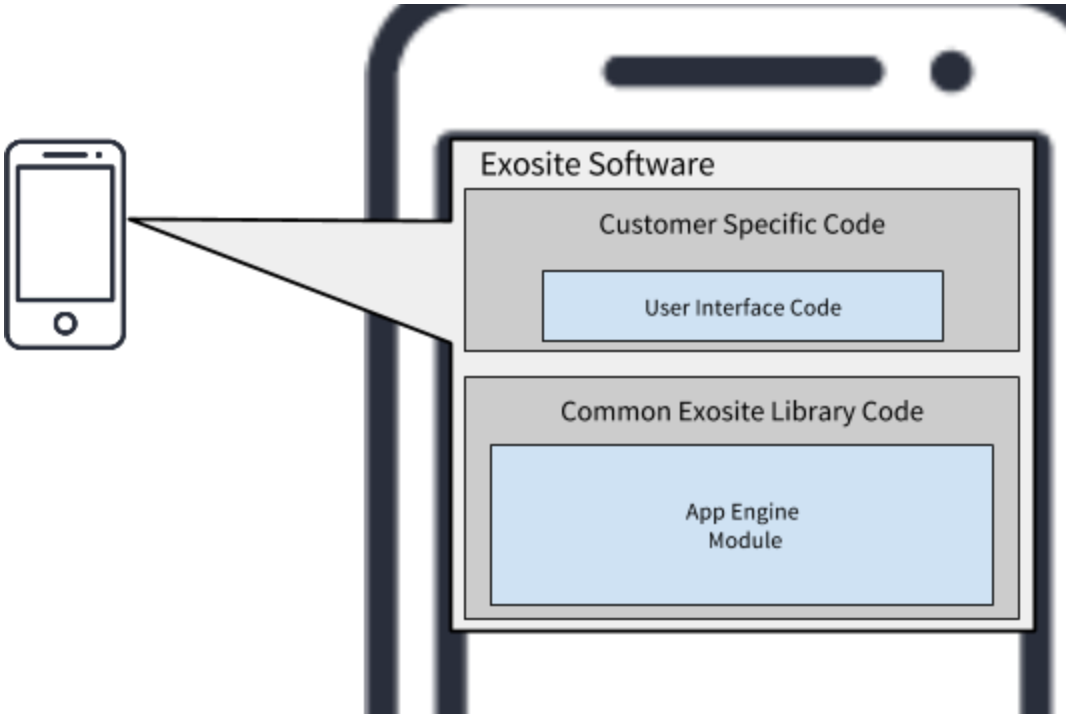


In the IoT Device itself the Main Controller Board is provided by the OEM. It is assumed that this board is already functioning and has a common HVAC feature set. The communication module on the other side will be stocked with Exosite firmware to cover the communication side.

The Wifi-Modules software is designed to be quickly adaptable by customers for different hardware environments. As a result the code is split into two major module categories:

1) Common Code
   These modules are expected to stay the same across many applications or just to require very minimal changes. It is assumed that 80% of the total code is in this common modules section.
2) Vertical Specific Code
   These modules are known to need changes per device hardware or device model configuration depending on whether the internal communication protocol is different (UART translation module) or whether the feature set that is supported varies (Smart Home Event Module)

1.2 Phone Details



The complete phone software implementation is expected to come from Exosite and provided to customers for customization - with focus on the UI. For this reason the split between Vertical Specific Code and Common Code as in the device side is here even stronger. All UI code should be isolated in it's own physical files so it can be changed e.g. for branding without affecting any of the application logic.

The business logic on the other side should stay the same 80%-90% of the time and designed to not care about UI changes.

2 Terminology

2.1 Message Types

Throughout this specification there are three different types of messages:

2.1.1 Requests

Every request expects to receive a response. HTTP is by default always request response. For WebSockets requests will be encoded as:

```
{"id": <id>, "request": <request_name>, "data": <request_data>}
```

Specifically the <id> field is used to map requests to their responses as the WebSockets protocol is asynchronous by nature.

2.1.2 Responses

Responses when using WebSockets are encoded as:

```
{"id": <id>, "status": <status>, "data": <response_data>}
```

2.1.3 Events

Events do not require responses. This specification uses Events only from Cloud => Phone communication to inform of data changes. Events do not need responses. For WebSockets events are encoded as:

```
{"event": <event_name>, "data": <event_data>}
```

2.1.4 Example of color coding

	Request/Response (Command Line Bash)
Request	curl https://192.168.1.1:32051/info
Response	HTTP/1.1 200 OK Date: Mon, 30 Sep 2016 16:18:20 GMT Content-Length: 52 Connection: close Content-Type: text/plain; charset=utf-8 serial=<sn>&last_error=<last_error>
Error Response	HTTP/1.1 400 Bad Request Date: Mon, 30 Sep 2016 16:18:20 GMT Content-Length: 6 Connection: close Content-Type: text/plain; charset=utf-8 error!
Event	

2.2 Protocols

Name	Description	RFC
HTTP	HTTP(s) protocol is the simplest protocol for requests/response interaction. This protocol is used mostly during the provisioning phase. **Protocols:** https:// - s for secure transport	https://tools.ietf.org/html/rfc2616
TLS	Security and encrytpion layer used to protect HTTPS, WebSockets and CBOR streaming. **Protocols:** tls://	https://tools.ietf.org/html/rfc5246
WebSocket	WebSockets are based on top HTTP and used between the Cloud and the Phone for asynchronous communication **Protocols:** wss:// - s for secure transport	https://tools.ietf.org/html/rfc6455
CBOR	CBOR is a JSON compatible efficient encoding. The main purpose for Exosite to use it is to safe bandwidth and memory on embedded devices. **Protocols:** cbors:// - s for secure transport	https://tools.ietf.org/html/rfc7049
JSON	JSON is the defacto standard encoding scheme in the internet. It is used broadly within this specification to define data semantic.	https://tools.ietf.org/html/rfc7159

The communication protocols are used in following phases are:

Phase 1: Provisioning	Mobile Phone => Device (initial provisioning)	https:// (Device is Server)
	Device => Cloud (initial provisioning)	cbors:// (Murano is Server)
Phase 2: Data Exchange	Mobile Phone => Cloud	wss:// (Murano is Server)
	Device => Cloud (data flow)	cbors:// (Murano is Server)

2.3 Data Types

Name	Type	Example	Description
<event_name>,	String - 20 bytes [a-z_]	"provision"	The event or request name according to the

<request_name>			command reference.
<id>	Numeric String - [1-9]+	123	A id number that is used to match request and response in WebSocket and CBOR communication.
<product_id>, <pid>	String - 20 alnum bytes		Exosite identifier for the product
<pwd>	String	"s3cr3t"	Password of the Wifi-Network, different limits depending on wifi security.
<request_data>, <response_data>, <event_data>	String - Json	"data"	Data payload encoded in json for the data protocol
<role>	String Enum - ["guest", "owner"]	"guest"	Identifies a user level as guest user or owner of the device
<sec>	String - 4 alnum bytes	"open"	Can be one of 'wep', 'wpa', 'wpa2' or 'open' to choose the authentication mode of the target wifi
<secret>	String - 20 bytes hex	"ABCDEF1234ABCDEF1234"	Secret used for handshake
<sn>	String - 17 bytes hex	"AB:CD:EF:12:34:54"	MAC of the Wifi Chip
<ssid>	String - 32 alnum bytes	"ACCompany-AC001"	Service Set IDentifier

2.4 Provisioning Overview

Provisioning will be done local in the current wifi the flow is like this:

1) Device is new / or reset
2) Device goes into AP mode
3) Phone app is opened for the first time
4) Phone app asks for user to login / registration
5) Phone-User can choose to sign up via Email or Facebook
6) Phone Signup and login finish
7) Phone looks and find Device in AP mode
   SSID is hardcoded to "<Vendor>-<Model>"
8) Phone send provisioning details (wifi-ssid and wifi-password) via https
   GET 192.168.1.1/provision?ssid=<ssid>&pwd=<pwd>&sec=wep/wpa/wpa2/open
9) Device answers with secret and serial number in the body
   serial=<sn>& secret=<secret>
10) Device tries wifi credentials
11) Phone connects to cloud and sends custom provision request:
    wss://<smarthome>.exosite.com/ws => "PROVISION:<secret>"
12) Device successfully connects to Internet and Provisions against OnePlatform
13) Device sends device secret to dataport
    POST <smarthome>.m2.exosite.com/stack/v1/alias?secret=<secret>
14) Device starts broadcasting its serial number via mdns

15) Cloud receives device write event
    If no phone is found cloud stores data into key-value store list for later retrieval
16) Cloud assigns user owner-level-rights to this device for the waiting phone user.
17) Phone receives message about connected device.
18) Phone can now control the device until it is reset.

De-Provisioning (when the device was owned by somebody else):

1) The device is reset
2) Device goes into AP mode (wifi-direction optional)
3) ... provisioning by the new user ...
4) The cloud receives "secret"
5) Cloud finds existing user with user owner-level-rights who used a different secret
6) Cloud deletes rights for existing users and assigns new user owner-level-rights

Locality Detection:

1) The device is broadcasting its serial number via mdns

3 Exosite ESH Protocol Definitions

The exosite smart home (esh) protocol definitions define different device types and semantic for values types.

Name	Example	Description
class	0	0: HouseHold Devices (家庭電器)
device_id	'101'	Describes the type of device. This will also enable/disable different command sets per device. 0: General (通用) F: Fan (電扇)
fields	["H00"]	Fields are defined per device family. Each device_id family has a different set of definitions.
esh_version	"4.00"	The current version of the ESH protocol, the current version is "4.00"
brand	"Company"	The name of the device offering company.
model	"AX4200"	The name of this product model.

3.1 Fields for device_id = 101 (Fan)

Field Name	Value Range	R/W	Description
H00	0: Off 1: On	R+W	On/Off
H01	0: Mode 0 1: Mode 1 Natural wind 2: Mode 2 3: Mode 3	R+W	Fan Mode
H02	0: Automatic Speed 1-32: Speed level	R+W	Fan Speed
H03	Min: -128 Max: 127	R	Indoor Temperature Celsius
H04	Min: 0 Max: 2359	R+W	Timer on This specifies the time of the day on which the fan should

			enable itself. The time encoded as decimal. E.g. 0100 means 1am 1330 means 1:30pm and so on
H05	Min: 0 Max: 2359	R+W	Timer off This specifies the time of the day on which the fan should **disable** itself. The time encoded as decimal.
H06	0: Forward 1: Reverse	R+W	Fan Rotation
H07	Min: 0 Max: 1440	R	Current in 0.1 Amp units
H08	Min: 0 Max: 1440	R	Voltage in V
H09	Min: 0 Max: 1440	R	Power factor
H10	Min: 0 Max: 1440	R	Power in W
H0a	Min: 0 Max: 1440	R	kwh in 0.1 kWh units
H0b	0: Off 1: On	R+W	Light
H0c	Min: 0 Max: 100	R+W	Light dimming % 100: Means full brightness 0: Mean light is off

4 Device Communication Reference

4.1 Device Provisioning Steps

4.1.1 Device is new / or reset

The device when starting up needs to check internal persistent storage first:

- If the device has no <cik> and no wifi credentials it should go into AP mode
- If <cik> and wifi credentials are present then the device should instead directly start to try to connect to the provisioned wifi **- without timeout**.

The device needs to have a reset button that will reset <cik> and wifi credentials and reboot the device so it can get it into AP mode and be provisioned again.

4.1.2 Device goes into AP mode

The device goes into AP mode and creates a new SSID with the name: "<Vendor>-<Model>"

The Wifi-Network needs to use WPA2 with AES-CCMP encryption. The password should be hardcoded to be "<vendor>"

The device sets it's own **IP-Address to 192.168.1.1 and starts listening on port 32051** for HTTPS connections.

At this point it only supports only two HTTPS commands below:

4.1.3 Device responds to info requests

The info requests purpose is to get information about most recent errors as well as the MAC address of the device:

	Request/Response (Command Line Bash)
Request	curl https://192.168.1.1:32051/info
Response	HTTP/1.1 200 OK Date: Mon, 30 Sep 2016 16:18:20 GMT Content-Length: 52 Connection: close Content-Type: text/plain; charset=utf-8 serial=<sn>&last_error=<last_error>

4.1.4 Device answers with secret and serial number in the body

The provision request provides the device with WIFI-Access point information for internet connectivity:

	Request/Response (Command Line Bash)
Request	curl https://192.168.1.1:32051/provision?ssid=<ssid>&pwd=<pwd>&sec=<wep|wpa|wpa2|open>
Response	HTTP/1.1 200 OK Date: Mon, 30 Sep 2016 16:18:20 GMT Content-Length: 52 Connection: close Content-Type: text/plain; charset=utf-8 serial=<sn>&secret=<secret>

The secret needs to be a newly generated random secret. That will only be used for one connection attempt. The secret should be a hexadecimal string and be **20 characters long.**

All other requests should be responded with a 400 bad request response.

4.1.5 Device tries wifi credentials

The device should now try to connect to the provided Wifi-Credentials. The connection should be given up after a **maximum timeout of 60 seconds**. After this timeout the device should go back into AP mode and set the 'last_error' string to one of:

- 10: SSID not found
- 11: Password not correct
- 12: Unknown connection error

4.1.6 Device sends device secret to Murano

The connection to the Exosite Murano Cloud is created using a TLS stream to port 443. Inside this stream CBOR encoding is used to transport messages. The host name needs to be transported as part of the TLS handshake via SNI. The client is using a randomly generated client certificate to identify itself.

Here a reference TLS connection using unix openssl tool:

```
~$ openssl s_client -connect <product_id>.m2.exosite.io:443 -quiet \
      -servername <product_id>.m2.exosite.io \
      -key clientcert_and_key.pem -cert clientcert_and_key.pem
```

Further examples of this connection will use the "cborcat" Linux tool to exemplify messages being sent over the network:

	Request/Response

Request	cborcat -key clientcert_and_key.pem -c <product_id>.m2.exosite.io:443
Response	connected (press CTRL+C to quit)
Request	{"id": <sn>, "fw_version": <fw_version>}
Response	{"ack": true, "timestamp": <timestamp>}
Request	{"id": 1, "request": "provision", "data": {"sn": <sn>, "secret": <secret>}}
Response	{"id": 1, "status": "ok"}

The connect and send attempt should be given up after a **maximum timeout of 20 seconds - or when an status != "ok" is received more than 3 times**. After this timeout the device should go back into AP mode and set the 'last_error' string to one of:

- 20:Could not connect to Exosite (<details>)
- 21:Error status:<status>

4.1.7 Device starts broadcasting its serial number via mdns

The device starts advertising its presence and service via mdns

MDNS Parameter Name	Value
Service Name	<BRAND>-<MODEL>._exosh1._tcp.local ACCompany1-AC001._exosh1._tcp.local
Text	id=<sn> vs=<firmware_version> id=A0:24:85:42:E9:43 vs=1.00
Port	32051

TXT Record Key	Description
id	MAC Address of the wifi module
vs	Firmware version of the wifi module

Example: Publishing mdns dns-sd service on a Linux Ubuntu machine using avahi:

	Request/Response (Command Line Bash)
Request	avahi-publish-service ACCompany1-AC001 _exosh1._tcp 32051 id=A0:24:85:42:E9:43 vs=1.00

Response	Established under name 'ACCompany1-AC001'

Example: Browsing mdns dns-sd service on a Linux Ubuntu machine using avahi:

	Request/Response (Command Line Bash)
Request	avahi-browse _exosh1._tcp -r
Response	= wlan4 IPv4 ACCompany1-AC001 _exosh1._tcp local hostname = [xxx.local] address = [192.168.1.113] port = [32051] txt = ["vs=1.00" "id=A0:24:85:42:E9:43"]

4.1.8 Device connect to cloud control

This connection is used to control data interaction between the device and cloud.

Example: Using Linux cborcat tool for initial connection.

	Request/Response (Command Line Bash)
Request	cborcat -key clientcert_and_key.pem -c <product_id>.m2.exosite.io:443
Response	connected (press CTRL+C to quit)

4.2 Device Data Interchange (after Provisioning)

4.2.1 Device => Cloud Communication

Request Reference:

Name	**Example**	**Description**
provision	{"id": <id>, "request": "provision", "data": { "sn": "<sn>", "secret": "<secret>" }}	After initial wifi connection send secret to cloud. The secret is a random number generated by the device. The serial number could be the devices MAC
	{"id": <id>, "response": "provision", "status": "ok"}	Response

status_change	{"id": <id>, "request": "status_change", "data": { "H00":0, "H01":0, }}	After initialization finished the device is going to send all relevant data in the specified interval to the cloud.
	{"id": <id>, "response": "status_change", "status": "ok"}	Response

4.2.2 Cloud => Device Communication

Defined Event List:

Req. Name	Example	Description
init	{"id": <id>, "request": "init", "data": { "fields": ["H00", "H01", "H02", "H03", "H04", "H05", "H0E", "H0F", "H10", "H11", "H14", "H17", "H20", "H21", "H28", "H29"] }}	Initialize fields that should be watched. Init needs to be acceptable multiple times.
	{"id": <id>, "response": "init", "status": "ok"}	Response
info	{"id": <id>, "request": "info"}	

	{"id": <id>, "response": "info", "status": "ok", "data": {
	"device": <sn>,
	"profile": {
	"esh": {
	"class":0,
	"esh_version":4.00,
	"device_id":"F",
	"brand":"ACCompany1",
	"model":"AC001"
	},
	"module": {
	"firmware_version":1.00,
	"mac_address":"AC123",
	"local_ip":"192.168.0.13",
	"ssid":"BRX13"
	},
	"cert": {
	"fingerprint": {
	"sha1":
	"DE28F4A4FFE5B92FA3C503D1A349A7F9962A8212"
	},
	"validity": {
	"not_before": "5/21/02"
	"not_after": "5/21/22"
	}
	}
	},
	"calendar": [
	{
	"start": "12:24",
	"end": "13:24",
	"days": [1,2,3,4,5,6,7],
	"active": 1,
	"esh": {
	"H00":1
	}
	}
		Response
		
		"class" - See above ESH definition
		"device_id" - See above ESH definition
		
		The "calendar" can contain up to 10 entries. Each entry should be executed on week days, by applying the ESH field values on the start time and reverting to the original values at the end of the time.

	], "fields": ["H00", "H01", "H02", "H03", "H04", "H05", "H0E", "H0F", "H10", "H11", "H14", "H17", "H20", "H21", "H28", "H29"]}	
set	{"id": <id>, "request": "set", "data": { "H00": 1 }}	Set ESH H'00 value to 1.
	{"id": <id>, "response": "set", "status": "ok"}	Success Response
	{"id": <id>, "response": "set", "status": "error", "message": <message>}	Error Response
get	{"id": <id>, "request": "get"}	Read device status.
	{"id": <id>, "response": "get", "status": "ok", "data": { "H00":0, "H01":0, ... }}	Response
ota	{"id": <id>, "request": "ota", "data": { "url": <url>, "sha1": <sha1> }}	Flash device with new firmware. The "url" contains an http(s) url to where the firmware should be downloaded from
	{"id": <id>, "response": "ota", "status": "ok"}	Response
cer	{"id": <id>, "request": "cer", "data": { "url": <url>, "cer": <cert>, "sha1": <sha1> }}	Update device data url and root certificate. "url" - specifies the new api url to use. such as "company1.m2.exosite.io" "cer" - specifies the actual certificate itself in hex encoding "sha" - is the sha1 fingerprint of the certificate in binary form
	{"id": <id>, "response": "cer", "status": "ok"}	Response

calendar	{"id": <id>, "request": "calendar", "data": [
	{
	"start": "12:24",
	"end": "13:24",
	"days": [1,2,3,4,5,6,7],
	"active": 1,
	"esh": {
	"H00":1
	}
	}
	]}
		Set new schedule.

	{"id": <id>, "response": "calendar", "status": "ok"}	Response