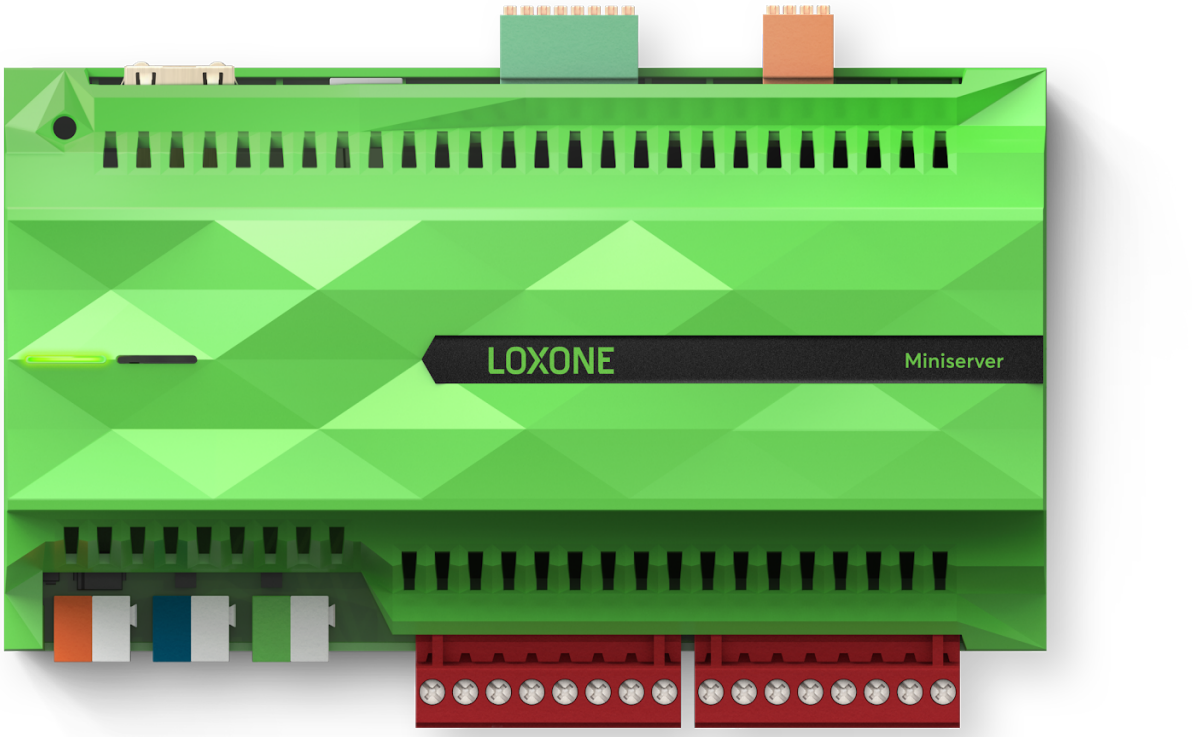
LOXONE Create Automation

COMMUNICATING WITH THE LOXONE MINISERVER

Version 11.0

Date 2020.05.08



In order to modernize our interface, we improved our Miniserver, allowing it to communicate with WebSocket clients that conform to the RFC6455. This document explains what is to be done, in order to communicate with our Miniserver on the basis of an RFC6455 websocket. Parts of this document will cover communicating via HTTP-Requests too (e.g. Secured Commands, Hashing or the Structure File).

Arguments in commands are wrapped in curly brackets, e.g.: “jdev/test/{the argument}”. So when {the argument} is “yeah” it will look like this: “jdev/test/yeah” - no more curly brackets here.

Important Notes

Since November 2019, our new generation of the Miniserver has been introduced. The new generation supports TLS Protocols such as WSS and HTTPS. For details see chapter Using HTTPS/WSS

As of Loxone Config 10.3 new hashing algorithms for passwords have been introduced. Starting with future versions, passwords will be hashed using SHA256. In order to ensure compatibility for future versions, please make sure to follow the updated instructions listed here.

As of Loxone Config 10.2 JSON Web Tokens (https://jwt.io/) have been introduced. Using legacy tokens is still supported, but deprecated in version 10.2 and will be removed in future versions. In order to ensure compatibility for future versions, please make sure to follow the updated instructions listed here.

As of Loxone Config 9.3 password based authentication has been removed in favor of token based authentication. Token based authentication has been introduced in Loxone Config 9.0.

Note: Using HTTP Basic Authentication for HTTP Requests is still available for debugging and testing purposes. Since credentials passed along using basic authentication may be intercepted and compromised, it is strongly recommended to use token based authentication.

This document has been revised and only contains info on token based authentication. Please make sure your implementation is ready for tokens.

Table of contents

Setting up a connection

What do we need?

- a websocket client implementation conforming to the RFC6455
- the IP or URL of the Miniserver (including the port)
    - beware, if the Miniserver is using the Loxone CloudDNS service, you need to resolve it first. The websocket-redirect won't work. (see CloudDNS)
- valid credentials (user & password)

Step-by-step guide

Earlier versions of this document did cover password based authentication and authentication without making use of encryption. This guide, however, is based on using encryption and tokens.

1. ensure the Miniserver is reachable
    a. our apps are using an HTTP(s)-Request that returns both the Miniservers Mac-Address and it's config version: "**{ipOrUrl}:{port}/jdev/cfg/apiKey"**
    b. Miniservers of our new generation will have an attribute "httpsStatus" in the response, see chapter Using HTTS/WSS for details.
2. Acquire the Miniservers public key via "**jdev/sys/getPublicKey**" -> {publicKey}
    a. Store on the client
    b. X.509 encoded key in PEM format
3. open a WebSocket connection
    a. use the following path: "**ws://{ipOrUrl}:{port}/ws/rfc6455**"
        i. Note: WSS is supported for Miniservers of the second generation. See Using HTTPS/WSS
    b. specify "**remotecontrol**" as Sec-WebSocket-Protocol.
4. Generate a AES256 key -> **{key}** (Hex)
5. Generate a random AES iv (16 byte) -> **{iv}** (Hex)
6. RSA Encrypt the AES key+iv with the **{publicKey}** -> {encrypted-session-key} (Base64)
    a. "{key}:{iv}" is the payload that needs to be encrypted using RSA
    b. This key and iv will be used to AES-encrypt messages on the websocket (e.g. for authentication or token acquisition). More info on encrypted commands.
7. Pass encrypted session-key to Miniserver via

"**jdev/sys/keyexchange/{encrypted-session-key}**"

8. Generate a random salt, hex string (length may vary, e.g. 2 bytes) -> **{salt}**
    a. This salt will be used to AES-encrypt messages on the websocket (e.g. for authentication or token acquisition). More info on encrypted commands.
9. There are two options now:
    a. If a token exists, then authenticate as described in Authenticating using tokens
    b. If a token needs to be acquired, proceed as described in Acquiring tokens
10. After either successful token based authentication or acquiring a new token, the socket is authenticated and ready to go.

Update the {salt} on your behalf, eg. every hour. This prevents replay attacks on compromised websocket connections.

- {cipher}: "nextSalt/{prevSalt}/{nextSalt}/{cmd}" encrypted with the session key (AES)

What can go wrong?

- If you provide invalid credentials you'll receive a message with the response code 401.
- If you don't authenticate within the first few seconds after establishing the socket connection, you'll receive a message with the response code 420 and the socket will close.
- If you do not authenticate, but try to send any other command but the command required to authenticate or acquire a token, you will receive a response with code 400.
- If you are blocked due too many failed login attempts, the Miniserver will block you for a certain time. The WebSocket closes right after it opens with a Close Code of 4003
- Up to 31 concurrent clients can receive live status updates. The "hasEventSlots" attribute of the jdev/cfg/api request indicates whether or not slots are available.
- First generation Miniservers can serve up to 48 http connections, the new generation is capable of serving up to 256 http connections. See Error-Code Section for details.

Using HTTPS/WSS

Since November 2019 the new generation of the Miniserver is available. Unlike our Miniserver Generation 1, the new generation allows for using TLS protocols such as WSS or HTTPS.

Basic Info on HTTPS/WSS

HTTPS basically is HTTP, but TLS (Transport Layer Security) is used for communication. The same is true for WSS and WS. For using TLS, servers must provide **a certificate** using which the client can both verify the authenticity of the server and encrypt all communication transferred between them.

Our Miniservers use a certificate that is automatically obtained when using our CloudDNS service (referred to as **CloudDNS-Certificate** here). Alternatively, you can upload your own certificate onto your Miniserver using Loxone Config (referred to as **custom-certificate** here).

What needs to be done to use HTTPS/WSS

This document goes into detail on how to set up a connection using WS or HTTP, which is still supported by Miniservers of the new Generation. Only little needs to be changed for making use of HTTPS & WSS.

- Use the TLS protocols HTTPS and WSS instead of HTTP and WS
- Use the Hostname instead of an IP-Address
    - Certificates cannot be created for IP-Addresses, only for hostnames such as “loxone.com”.
    - Connecting to an IP using TLS will result in certificate errors, as the authenticity cannot be verified.

How to determine if the Miniserver supports HTTPS/WSS?

Especially when connecting to a Miniserver for the first time, it may not be known whether or not TLS-Protocols are supported. There are different ways to find out, depending on whether or not you are trying to connect locally or from the internet.

Ask the Miniserver itself

When connected locally, try a reachability check request such as “http://{miniserverIP}/jdev/cfg/apiKey”. This request won’t transfer any confidential data, but will contain an attribute “httpsStatus” which has the value 1 if TLS protocols are available, and a value of 2 if the Miniserver has a certificate but its expired.

Ask our CloudDNS service

If a Miniserver supports WSS/HTTPS, the response described in the CloudDNS-Section will contain two additional attributes “**IPHTTPS”** and “PortOpenHTTPS”. These attributes are the counterpart for

“IP” and “PortOpen” for using HTTPS/WSS and are only available for Miniservers supporting TLS.

Trial & Error

If the Miniserver isn’t in the local network and the only info is it’s hostname (e.g. my.smarthome.com:7777), the only option is trial & error. Launch a reachability check request “https://my.smarthome.com:7777/jdev/cfg/apiKey” - if it succeeds, the Miniserver supports TLS-Protocols.

What hostname to use with CloudDNS?

As mentioned, certificates can only be created for hostnames and not IP addresses. To make a connection using TLS the following steps need to be taken:

1. Request the current IP & Port from the CloudDNS
2. Create a hostname containing the IP and port returned by the IPHTTPS attribute of the response.
    a. Split up IPHTTPS into the {ip} and {port}
    b. Clean up the {ip} → {cleaned-ip}
        i. IPv4: Replace dots (“.”) with minuses (“-”)
        ii. IPv6: Replace colons (“:”) with minuses (“-”) and remove the brackets (“[“ & “]”) at the beginning and end.
    c. Create hostname “{cleaned-ip}.{snr}.dyndns.loxonecloud.com:{port}”
        i. {snr} is your Miniservers Serial-Number.
        ii. E.g.: “200-12-14-24.{snr}.dyndns.loxonecloud.com:4523”

How can I connect locally with TLS?

As long as your Miniserver is capable of TLS and is using our CloudDNS-Certificate, you can. You need to create a hostname just like if you would connect from the internet, but use your Miniservers local IP and port. E.g. “192-168-1-47.{snr}.dyndns.loxonecloud.com:443”

If the Miniserver is using a custom certificate, connecting with it via it’s local IP will result in certificate verification errors.

General Info

In order to communicate with the Miniserver, a few things have to be known. This is a short textual explanation of the communication, details can be found further below.

CloudDNS

You can request the current IP Address of your Miniserver using an HTTP Request: *"dns.loxonecloud.com/?getip&snr={ SNR of Miniserver }&json=true"*. The Request returns the current IP Address and Port of the Miniserver represented as JSON. Here is a list of the important attributes in that response.

- Code
  - 200 = everything okay
  - 403 = Miniserver is not registered with Loxone, CloudDNS disabled
  - 405 = Miniserver not reporting to CloudDNS
  - 409 = Unsecure Password in place, external access blocked.
  - 412 = port not opened
  - 418 = Denied, see responseJSON-Attribute for details
  - 481 = Miniserver could not connect to the remote connect service
  - 482 = Miniserver connection via remote connect was aborted due to a timeout
  - 483 = Miniserver makes a scheduled restart
- IP
  - Current IP & port reported by the Miniserver (for WS or HTTP)
- PortOpen
  - Indicates whether or not the port reported is open, indicates configuration issues on the Miniserver
- LastUpdated
  - Date & Time of the last IP update made by the Miniserver
- IPHTTPS & PortOpenHTTPS
  - Not available for Miniserver Generation 1
  - See chapter Using HTTPS/WSS for details

Remote Connect Service

Since Version 11.0 Miniservers may be reachable externally via our Remote Connect Service. This service does NOT work for Miniservers Generation 1, as it requires encryption.

Those miniservers will be accessible via our Cloud DNS service, on client side no changes are necessary. Remote Connect only supports using HTTPS/WSS.

Controls

In the following document a "Control" is a Function/Block-Function like "Virtual Input", "Virtual

State", "Intelligent Room Controller", "Sauna" etc.

Commands

The Miniserver will answer every command it receives, it will return a TextMessage as confirmation. This message contains the command it has received, an HTTP-Status-Code as success indicator and a value. The value is the info that was requested or the state after a control-command was executed - but beware, these are not fully implemented, so instead rely on the state updates.

- a control command looks like
  - "*jdev/sps/io/{uuid}/{command}*"
- all commands can either start with "*jdev/*" or just "*dev/*"
  - *"jdev/"* in JSON-Format (Beware, the answer will contain 'dev' instead of 'jdev')
  - *"dev/"* delivers responses in XML-Format and is deprecated

Encryption

Available since 7.4.4.14

Miniserver Generation 2

Miniservers of our most recent generation are now capable of using HTTPS/WSS, so every information transferred between the Miniserver and the client is encrypted without further steps.

Please note, that some commands (e.g. GetToken, RefreshToken, ..) require application layer encryption, even though TLS is in place.

Miniserver Generation 1

The hardware specifications of the Miniserver do not allow full featured SSL encryption, it would put the CPU under too much pressure leading to delayed responses to events. In order to still be able to send data to the Miniserver in a secure way (in addition to hashing), Command Encryption was introduced.

Secured Commands

We have the possibility to use a "visualization password" for Controls (set in Loxone Config), those passwords are added to the commands as described below:

1. request the visualization password from the user - {visuPw}
2. request a {key}, {salt} and the used hashing algorithm {hashAlg} from the Miniserver ("*jdev/sys/getvisusalt/{user}*")
   a. {user} = the user whos visu password has been entered
3. Create an {hashAlg} hash (SHA1, SHA256,..) of "{visuPw}:{salt}" -> {visuPwHash}
4. Create an HMAC-SHA1 or HMAC-SHA256 hash using the uppercase {visuPwHash} and the

{key} (see Hashing) - {hash}

5. send "*jdev/sps/**ios/{hash}**/{uuid}/{command}*"
    a. response has Code 200 if password was correct and command could be executed
    b. a response with Code 500 means the password was incorrect

Hashing

1. the key from the "*jdev/sys/getkey*", "jdev/sys/getkey2" or "jdev/sys/getvisusalt" responses are hex-encoded
2. create a hash from the desired text (user:passHash, visuPwHash, token) using HMAC-SHA1 or HMAC-SHA256 with the {key} received in the answer
    a. To create 'passHash' & 'visuPwHash' use the hashing algorithm {hashAlg} that is defined in the answer of the corresponding requests
3. encode the hash back to hex

Tokens

Tokens have been introduced in Loxone Config 9. They are used for authentication instead of passwords. Tokens can expire, be revoked without changing the password - and they allow for a much more refined permission handling. See separate section for details on how to handle tokens.

Permissions

Along with the introduction of tokens, a refined permission handling has been implemented. When requesting a token, the desired permission is to be specified. For establishing connections, only two permissions are important: the permission for the web (ID = 2) and the permission for the app (ID = 4).

One of these two permissions must be specified when acquiring a token for communication (opening a websocket or sending HTTP-Requests). "Web" means the token will have a short lifespan and an a token with the "App" permission will last for a longer period (4 weeks).

States

In order to receive the states of sensors and actuators, state-updates need to be enabled on the socket. This is done by sending "**jdev/sps/enablebinstatusupdate**" to the Miniserver. The Miniserver will only publish the states of sensors and actuators that are used in the user interface.

After this command is sent, the client will receive large initial Event-Tables containing Value-, Text-, Daytimer- and Weather-States. These Event-Tables contain the current states of all sensors and actuators.

The client needs to store these infos for the time the connection is active, because further on the Miniserver will only inform the client on changes to these states.

These states are always sent out as an UUID paired with some sort of value, using the Structure-File (see LoxAPP3.json) and the UUID you can find out what value belongs to which control.

Understanding the messages

As mentioned in the chapter on how to setup a connection, messages sent by the Miniserver are always prequeled by a binary message that contains a MessageHeader. So at first you'll receive the binary Message-Header and then the payload follows in a separate message.

Message Header

The message header is used to distinguish what kind of data is going to be sent next and how large the payload is going to be. In some cases, the Miniserver might not know yet how large the payload is going to be. For these cases a flag indicates that the size is estimated (see Estimated). The header is sent as a separate data packet before the actual payload is transmitted. This way the clients know ahead how large the payload is going to be. Based on this info, clients know how long it's going to take and respond accordingly (UI or timeouts).

Structure of the Message Header

The Message Header is an 8-byte binary message. It always starts with 0x03 as first byte, the second one is the identifier byte, which gives info on what kind of data is received next. The third byte is used for information flags and the fourth byte is reserved and not used right now. The last 4 bytes represent an unsigned integer that tells how large the payload is going to be.

1st Byte	2nd Byte	3rd Byte	4th Byte	5th Byte	6th Byte	7th Byte	8th Byte
0x03	Identifier	InfoFlags	rsvd	len	len	len	len

```
typedef struct {
    BYTE cBinType;          // fix 0x03
    BYTE cIdentifier;       // 8-Bit Unsigned Integer (little endian)
    BYTE cInfo;             // Info
    BYTE cReserved;         // reserved
    UINT nLen;              // 32-Bit Unsigned Integer (little endian)
} PACKED WsBinHdr;
```

1st Byte: fix 0x03

2nd Byte: Identifier

8-bit Unsigned Integer (little endian)

The identifier byte is used to distinguish between the different kinds of messages

Identifer	Message-Type
0	Text-Message
1	Binary File
2	Event-Table of Value-States
3	Event-Table of Text-States
4	Event-Table of Daytimer-States
5	Out-Of-Service Indicator - presumably due to an Firmware-Update. No message is going to follow this header, the Miniserver closes the connection afterwards, the client may try to reconnect.
6	Keepalive response (after sending “keepalive”, the Miniserver will respond with this identifier - therefore the connection is up and running!)
7	Event-Table of Weather-States

3rd Byte: Info

The 3rd Byte of the Header is used to provide additional information regarding the incoming message.

1st Bit	2nd Bit	3rd Bit	4th Bit	5th Bit	6th Bit	7th Bit	8th Bit
Estimated	rsvd	rsvd	rsvd	rsvd	rsvd	rsvd	rsvd

1st Bit: Estimated

In order to get fast info of how big the next incoming data will be, a Header with the Estimated-Bit set, tells, that the given size is only estimated (eg. Gateway Miniservers, ..)

An Estimated-Header is **always** followed by an exact Header to be able to read the data correctly!

4rd Byte: reserved

5th-8th Byte: length of payload

32-bit Unsigned Interger (little endian)
The size of the payload, may be estimated. This info can be used to adopt timeouts etc.

The actual messages

Text-Messages

Text-Messages are supported since day one of WebSockets. They are handled by all WebSocket implementations out there - so having a separate message-header telling there will be a Text-Message wouldn't be necessary. But in order to stay consistent in our application protocol, we did add it for those messages as well.

Text-messages are received as responses to commands, but our Structure-File and other XML- or JSON-Files are sent as Text-Messages too.

Binary Files

If you download files (e.g. images, statistic-data) from the Miniserver, you will receive a binary file. As mentioned before, files with text-content (e.g. the LoxAPP3.json) will be be delivered as a text-message, so you don't have to decode it.

Event-Tables

Incoming events are always grouped as tables according to their type (Value, Text, Daytimer, Weather). Each Event-Entry in these tables has it's own UUID, so you can assign the values to the correct Controls.

Event-Table of Value-States

Value-States are the simplest form of a state update, they consist of one UUID and one double value each, so their size is always 24 Bytes.

Binary-Structure of a Value-Event

```
typedef struct {
    PUUID uuid;      // 128-Bit uuid
    double dVal;     // 64-Bit Float (little endian) value
} PACKED EvData;
```

Event-Table of Text-States

Text-States are more complex since their size varies based on the text they contain. That is why they do not only consist of an UUID and the text, but also an unsigned int that specifies how long the text is.

- The UUID-Icon is used by the "Status"-Control (see Icons)
- If textLenght is not a multiple of 4 then padding bytes are appended, that are to be ignored.

Binary-Structure of a Text-Event

```
typedef struct {    // starts at multiple of 4
    PUUID uuid;                 // 128-Bit uuid
    PUUID uuidIcon;             // 128-Bit uuid of icon
    unsigned long textLength;   // 32-Bit Unsigned Integer (little endian)
```

```
    // text follows here
} PACKED EvDataText;
```

Event-Table of Daytimer-States

Like Text-States, Daytimer-States do not have a fixed size, it varies on how many Daytimer-Entries there are per Daytimer.

"nEntries" tells the number of daytimer-entries which the package contains.

Analog Daytimer: each entry does have it's value
Digital Daytimer: an existing entry means "on", no entry means "off"

Binary-Structure of a Daytimer-Event-Table

```
typedef struct {
    PUUID uuid;             // 128-Bit uuid
    double dDefValue;       // 64-Bit Float (little endian) default value
    int nrEntries;          // 32-Bit Integer (little endian)
    // entries (EvDataDaytimerEntry) follows here
} PACKED EvDataDaytimer;
```

Binary-Structure of a Daytimer-Entry

```
typedef struct {
    int nMode;  // 32-Bit Integer (little endian) number of mode
    int nFrom;  // 32-Bit Integer (little endian) from-time in minutes since midnight
    int nTo;    // 32-Bit Integer (little endian) to-time in minutes since midnight
    int bNeedActivate;      // 32-Bit Integer (little endian) need activate (trigger)
    double dValue;          // 64-Bit Float (little endian) value (if analog daytimer)
} PACKED EvDataDaytimerEntry;
```

Event-Table of Weather-States

If an active Weather-Abo is up and running, we also get Weather-State Updates.

Each Weather-Event-Table contains info about the up-to-dateness (in Seconds since 2009, UTC) of the Weather-Information, the number of entries and the entries itself.

Binary-Structure of a Weather-Event-Table

```
typedef struct {
    PUUID uuid;                 // 128-Bit uuid
    unsigned int lastUpdate;    // 32-Bit Unsigned Integer (little endian)
    int nrEntries;              // 32-Bit Integer (little endian)
    // entries (EvDataWeatherEntry) follows here
} PACKED EvDataWeather;
```

Binary-Structure of a Weather-Entry

```
typedef struct {
    int timestamp;              // 32-Bit Integer (little endian)
    int weatherType;            // 32-Bit Integer (little endian)
    int windDirection;          // 32-Bit Integer (little endian)
    int solarRadiation;         // 32-Bit Integer (little endian)
```

```
    int relativeHumidity;          // 32-Bit Integer (little endian)
    double temperature;            // 64-Bit Float (little endian)
    double perceivedTemperature;   // 64-Bit Float (little endian)
    double dewPoint;               // 64-Bit Float (little endian)
    double precipitation;          // 64-Bit Float (little endian)
    double windSpeed;              // 64-Bit Float (little endian)
    double barometicPressure;      // 64-Bit Float (little endian)
} PACKED EvDataWeatherEntry;
```

The UUID

The Miniserver uses UUIDs in order to uniquely identify controls, in- or outputs. That is why the states that are published using Event-Tables have one UUID for each state, so that the values can be linked to their controls. Each UUID has a fixed size of 128 Bit.

Structure-Files such as “LoxAPP2.xml” (deprecated) or the new, tidied up “LoxAPP3.json” are providing the information on what UUID is related to what control, or to what in- or output. They can be acquired by sending “data/LoxAPP2.xml” or “data/LoxAPP3.json” to the Miniserver.

Binary-Structure of a UUID

```
typedef struct _UUID {
    unsigned long Data1;           // 32-Bit Unsigned Integer (little endian)
    unsigned short Data2;          // 16-Bit Unsigned Integer (little endian)
    unsigned short Data3;          // 16-Bit Unsigned Integer (little endian)
    unsigned char Data4[8];        // 8-Bit Uint8Array [8] (little endian)
} PACKED PUUID;
```

Converting a UUID to a string

```
CStringA str;
str.Format("%08x-%04x-%04x-%02x%02x%02x%02x%02x%02x%02x%02x",
    uuid.Data1,uuid.Data2,uuid.Data3,uuid.Data4[0],uuid.Data4[1],uuid.Data4[2],
    uuid.Data4[3],uuid.Data4[4],uuid.Data4[5],uuid.Data4[6],uuid.Data4[7])
```

Icons

In general, icons are used for rooms and categories, but also for displaying states, as in the “Status”-Control. UUIDs are already used to link states to controls and furthermore they are also used to identify icons and link them to groups (rooms or categories), or “Status”-Controls.

SVG

Along with the release of Loxone Config 6.0 a new format for icons was introduced: “Scalable Vector Graphics”, short “svg”. Previously the only image format supported was “Portable Network Graphics”, short “png”. SVGs are image descriptions in XML-Format, they are not only scaleable losslessly, but they can also be modified and animated.

Where to get them, where to put them

The structure file gives info what icon is to be used where. In new configurations you will mostly

find “svg”-Icons (e.g.: “00000000-0000-0020-2000000000000000.svg”). Since some Miniservers might make use of customized icons, there will still be some PNGs out there, even with newer Config-Versions. Those can be identified either by “.png” or the file appendix is simply missing (e.g. “00000000-0000-0020-2000000000000000”).

Images can be downloaded over the WebSocket by simply sending the UUID plus the type to the Miniserver. So “00000000-0000-0020-2000000000000000.svg” will return the SVG-File of this image, while “00000000-0000-0020-2000000000000000.png” will return the same image as PNG.

Text-States only contain UUIDs for icons without specifying the format. The only way to identify whether it is a PNG or an SVG is the response type given in the Message-Header when downloading the Icon. If it’s an Text-Message, then it’s an SVG (since it’s basically an XML-File), otherwise it’s an PNG and the Message-Header will indicate that it’s a binary file. So for all “Status”-Controls try to download an SVG and later decide based on the response whether it is PNG or SVG.

Caching

An icon is identified by an UUID per Miniserver. This UUID doesn’t change as long as the icon remains the same, regardless if you’re connected locally or remote. So you can reuse the downloaded icons locally and remote.

Structure-File: LoxAPP3.json

The Structure-File was mentioned a few times before in this document, it is the central element for creating a visualisation. It contains almost everything you need to know about the Miniserver.

Download and caching

The new structure-file can be downloaded by sending “data/LoxAPP3.json” to the Miniserver. Since it’s a text-file, the Miniserver will respond with a Message-Header on the websocket accordingly.

The Structure-File has got a field “lastModified that contains a timestamp, this is the date when the configuration of the Miniserver was last changed. So every time a connection to a Miniserver is established, the first thing to do after the WebSocket is up and running is to check if your cached version of the Structure-File is up to date. This is achieved by sending “jdev/sps/LoxAPPversion3” to the Miniserver and comparing its response to the value of the “lastModified” field of your cached Structure-File.

More info

For detailed information on the Structure File & the controls within please see the separate

document on the Structure File.

Command Encryption

Available since 8.1

As mentioned in the introduction section on encryption, Command Encryption is a technique that allows clients to encrypt commands that would usually be sent via plain text. It is based on AES256 and Public-key cryptography for the AES session key exchange.

A few commands aren't supported:

- Images/Icons (.svg, .png, camimage)
- Files (LoxAPP3.json*, fsget/fslist)
    - LoxAPP3.json is available - but only "enc" is supported, as encrypting the response would cause a heavy CPU load on Miniserver Gen 1.
- Statistic Files and Data

RSA	**AES**
• ECB • PKCS1 • Base64 with NoWrap	• CBC • ZeroBytePadding • Base64 with NoWrap • 16 Byte IV • 16 Byte Block size • 32 Byte Key length

Step-by-step Guide HTTP Requests

1. Acquire the Miniservers public key via "jdev/sys/getPublicKey" -> {publicKey}
    a. Store on the client
    b. Format: X.509 encoded key in ANS.1
2. Prepare your command -> {cmd}
    a. If authentication is required, append it to the {cmd} as described in Authenticating using tokens
3. Generate a random salt, hex string (length may vary, e.g. 2 bytes) -> {salt}
    a. Note: This is not the {userSalt} retrieved using the getkey2-command.
4. Prepend the salt to the actual message "salt/{salt}/{cmd}" ->{plaintext}
    a. Example for {cmd}: "jdev/sps/io/AI1/on"
5. Generate a AES256 key -> {key} (Hex)

6. Generate a random AES iv (16 byte) -> {iv} (Hex)
7. Encrypt the {plaintext} with AES {key} + {iv} -> {cipher} (Base64)
8. URI-Component-Encode the {cipher} -> {enc-cipher}
9. Prepare the command-> {encrypted-command}
    a. "jdev/sys/enc/{enc-cipher}"
        i. only the command itself is encrypted
    b. "jdev/sys/fenc/{enc-cipher}"
        i. The Miniserver also AES Encrypts the response (Base64)
        ii. The mime-type is the one from the decrypted response
10. RSA Encrypt the AES key+iv with the {publicKey} -> {session-key} (Base64)
    a. "{key}:{iv}"
11. URI-Component-Encode the {session-key} -> {enc-session-key}
12. Append the session key to the {encrypted-command} -> {encrypted-command}
    a. "{encrypted-command}?sk={enc-session-key}"
13. Send the request!
    a. This request doesn't require credentials itself, the {cmd} may contain the credentials if needed
14. The Miniserver will decrypt and process the command.
    a. If it cannot be decrypted (invalid public key, unexpected salt change) it will return 401
15. The Miniserver will respond after the decrypted command was processed and return the value & status code as with a regular command.

Sending encrypted commands over the websocket

1. AES-Encrypt the command using the key, iv and salt negotiated during the websocket connection establishment "salt/{salt}/{cmd}" -> {cipher} (Base64)
2. URI-Component-Encode the {cipher} -> {enc-cipher}
3. Prepare the command-> {encrypted-command}
    a. "jdev/sys/enc/{enc-cipher}"
        i. only the command itself is encrypted
    b. "jdev/sys/fenc/{enc-cipher}"
        i. The Miniserver also AES Encrypts the response (Base64)
        ii. The mime-type is the one from the decrypted response
4. Send the command & if needed (fenc) AES-decrypt the response using the {key} and {iv} created when the connection was established.

Tokens

Available since 9.0, Updated in 10.2

Clients initially acquire a token using the users password. This token is stored and used instead of

the password for authentication. The following section will go into detail on how to work with tokens, assuming that all commands are sent on a secure, encrypted connection as explained in Encryption.

In order to simplify authentication/verification throughout the Loxone Smart Home, JSON Web Tokens have been introduced in version 10.2. In order to avoid breaking changes, version 10.2 introduces new web services for acquiring and refreshing JSON Web Tokens and a separate web service allowing to check if tokens are valid.

Acquiring, refreshing and authenticating with legacy tokens is still supported, but deprecated. Support for legacy tokens will be removed in future versions. It is highly recommended to move to JSON Web Tokens as soon as possible. Legacy tokens may be converted to JSON Web Tokens using the new refresh token command.

Acquiring tokens

Updated in 10.2

Acquiring a token is similar to password authentication in previous versions. Additionally to the "key", a "salt" is needed for acquiring a token. A token can be either acquired via HTTP requests or via a websocket - please note that command encryption is mandatory for requesting tokens.

- Acquire the "key", "salt" & "hashAlg" at once using "**jdev/sys/getkey2/{user}**"
  - **{user}** is the username for whom to acquire the token.
  - The "salt" retrieved will be referred to as **{userSalt}**
  - "hashAlg" is the hashing algorithm that should be used
- Hash the password including the user specific salt
  - **{pwHash}** is the uppercase result of hashing the string "**{password}:{userSalt}**" using the in the getkey2 command specified hashing algorithm ('hashAlg', e.g. SHA1, SHA256).
  - {userSalt} is part of the result of the getkey2-Request.
- Create the hash that includes the user name
  - **{hash}** is the string "**{user}:{pwHash}**" hashed with the key returned by the getkey2-Request using HMAC-SHA1 or HMAC-SHA256.
    Do not convert the result to upper or lower case, leave it unchanged. For details on the hashing process see Hashing
- Request a JSON Web Token **"jdev/sys/getjwt/{hash}/{user}/{permission}/{uuid}/{info}"**
  - This request **must be encrypted**. Unencrypted getjwt requests will be declined with 400 Bad Request. See command encryption for more details.
  - **{permission}** specifies the permission this token needs to grant. This integer impacts the tokens lifespan, e.g. a token with the web-permission (2) will last for a short period of time, while a token with the app-permission (4) will last for weeks.

    - The **{uuid}** identifies the client who is requesting the token on the Miniserver. It allows to look up all tokens a client has been granted. This is why the UUID should either be derived from your devices identity information or generated automatically and stored within the app. It has to be in the following format as this one: “098802e1-02b4-603c-ffffeee000d80cfd”.
    - The **{info}** contains a (Url-Encoded) text describing the client, e.g. “Thomas%20iPhone%20X”
- Store the response, it contains info on the lifespan, the permissions granted with that token and the JSON Web Token itself.
    - **{token}** is the JSON Web Token itself, it needs to be stored for authenticating.
    - {validUntil} represents the end of the tokens lifespan in seconds since 1.1.2009
    - {tokenRights} holds a bitmap, where a flag is set for each granted permission.
    - {unsecurePass} is set to true if a weak password is in place, it should result in a prominent warning for the user, asking to immediately change the password.
    - {key} can be used for subsequent commands, just like a getkey-Result.
- A websocket connection on which a token was acquired successfully is considered authenticated.

Authenticating using tokens

- Prepare the {hash}
    - **{hash}** is the outcome of hashing “**{token}**” along with the result of a getkey-Request using the HMAC-SHA1 or HMAC-SHA256 algorithm. (see Hashing)
    - As of version 10.0 both JSON Web Tokens and legacy tokens may be used for authentication.
- Prepare the {authCmd}
    - “**authwithtoken/{hash}/{user}**” for websockets
    - For HTTP-Requests “**?autht={hash}&user={user}**” is appended to the existing cmd.
- Encrypt and send the **{authCmd}**
    - Details on encrypted commands via Websocket
    - Details on encrypted commands via HTTP-Requests

Refreshing tokens

Updated in 10.2

Tokens have a limited lifespan, depending on the permissions granted with them. When this lifespan expires, tokens will no longer be valid. Refreshing a token will return a new token with the same permissions and an extended lifespan.

When passing a legacy token to the new refresh token command, a JSON Web Token will be returned instead.

- Send "**jdev/sys/refreshjwt/{tokenHash}/{user}**" via websocket (HTTP support not verified)
    - **{tokenHash}** is the outcome of hashing the **{token}** with the result of a getkey-Request. (see Hashing)
    - {user} is the user whose token is to be refreshed

This request will only succeed if the token is valid. If successful, the response will contain an updated **{validUntil}**-value, an updated **{unsecurePass}**-flag and a new **{token}** attribute.

Checking if tokens are valid

Available since 10.0

This request was introduced to allow verifying a that a token is still valid, without renewing it as in "refreshToken".

- Send "**jdev/sys/checktoken/{tokenHash}/{user}**" via websocket (HTTP support not verified)
    - **{tokenHash}** is the outcome of hashing the **{token}** with the result of a getkey-Request. (see Hashing)
    - {user} is the user whose token is to be refreshed

This request will only succeed if the token is valid. If successful, the response will contain an **{validUntil}**-value and an updated **{unsecurePass}**-flag. When changing passwords, this request can be used to determine if the new password is secure, by checking the {unsecurePass}-flag of a refresh-request afterwards

Killing tokens

Tokens can be explicitly invalidated (= "killed") too. It is recommended to kill a token as soon as it is no longer needed, as it helps keeping the Miniservers token storage clean.

- Send "jdev/sys/killtoken/**{tokenHash}**/{user}"
    - **{tokenHash}** is the outcome of hashing the **{token}** with the result of a getkey-Request. (see Hashing) {user} is the user whose token is to be killed

A killed token will no longer be usable.

Handling an active connection

Keeping the connection alive

The Miniserver has to watch over it's clients and has to keep them all informed on everything that's changed. In order to prevent sending updates to clients that aren't listening anymore, it will close

the connection if the client doesn't send anything for more than 5 minutes.

To prevent this, while not having to constantly query a control or alike, there is a special command called "keepalive". Whenever a client sends this command to the Miniserver, it will respond with a Message Header with the identifier 0x06. This command can be used to tell the Miniserver that the client is still there and listens on the WebSocket.

Detecting issues

Our websocket is used for remote control. Mostly this is being done by apps running on smartphones that don't always enjoy the best connection quality (poor carrier network, WiFi almost out of reach). A poor or broken connection might cause that the user looks at old outdated data. E.g.: the app could show that your garage door is closed, while it's fully open.

By repeatedly sending out the keepalive-Request, the time between request and response (8 byte Message-Header) can be used as an indicator for the connection quality. When the Miniserver sends large messages this might lead to mistakenly detecting a connection problem, since receiving the response might take a while. The payload size in the Message-Headers can be used to adopt timeouts accordingly.

Error-Codes

...Work in Progress

General Info

Error-Codes may be returned via HTTP-Header or as field “Code” in Results for a Requests.

Returned Error-Codes

- 200
  - The request was successful
- 401
  - Unauthorized: the requesting user was not authorized (invalid username/password)
  - Processing an encrypted request failed
- 403
  - The requesting user has not enough rights for the request
- 404
  - Unrecognized command
- 423
  - The requesting user is disabled
- 503
  - Service Unavailable; The Miniserver is restarting and not ready for requests
- 901
  - Maximum number of allowed concurrent connections reached

Changing User-Access-Codes

- 409
  - This code is already in use
- 406
  - Invalid code
- 403
  - Insufficient rights to change the code for the requested user
- 429
  - Brute force detected. To many requests.

Creating or editing an user

- 400
  - Username is already taken
- 404

    - Requested user or user group not found
- 405
    - Edit of last available administrator is not allowed
- 500
    - Malformed Json: the request is not in the expected format

Websocket Close Codes

General Info

Close codes give further information why the Miniserver closed the Websocket session

- 4004
    - Some user has been changed
- 4005
    - The user currently connected has been changed either by them self, or by another user
- 4006
    - The user trying to establish a connection has been disabled
- 4007
    - The Miniserver is currently performing an update
- 4008
    - The Miniserver don't have any event slots for the initiated Websocket session