Contents

Version: 1.1.7

Get Started

Authentication

General Error Handling

WebSocket Configuration

Securing the connection

MQTT Configuration

Code examples

login - Login into SvanLINK

logout - Logout

setNewPassword - change access password

licenseStatus - get license status

setLicense - activate new licence key

getStatus - Get device statuses

getResults - Get measurement results

getSpectrumResults - Get spectrum results

startMeasurement - Start measurements

stopMeasurement - Stop measurements

getConfig - Get configuration

setConfig - Set configuration

getSetup - Get setup from connected device

setSetup - Upload setup to connected device

copySetup - Copy setup from device to device(s)

sendRawCommand - Send # type command

getFileList - Get file list from devices

downloadFile - Download single file from device

downloadFileList - Download multiple files from devices

deleteFile - delete file from device

getVersion - get current app version

checkUpdate - check for updates

downloadUpdate - download updates

installUpdate - install updates

MQTT requests:

setMqttConfig - Set MQTT broker configuration

getMqttConfig - Get MQTT broker configuration

addMqttTopic - Add MQTT topic

deleteMqttTopic - Delete MQTT topic

getMqttTopicList - Get list of MQTT topics

WebSocket only requests:

subscribeChannel - subscribe to channel

getChannelList - get list of channels

deleteChannel - delete channel

Get Started

To get started with the SvanLINK application, follow these steps:

Linux

1. Download the zip file containing the application.
2. Unpack the zip file:

```
unzip svanlink_os_osType.zip
```

3. Navigate to the unpacked directory:

```
cd svanlink
```

4. Make the binaries executable:

```
chmod +x svanlink
chmod +x restart
```

5. Run the application:

```
sudo ./svanlink
```

Windows

Run the installation file and follow the instructions.

The API requests by default are sent to local address on TCP port `8000`.

In browser under the local address on port `80` (default http port) should be available UI of the app. For example: `http://localhost`

Mac OS

Run the installation file and follow the instructions.

The API requests by default are sent to local address on TCP port `8000`.

In browser under the local address on port `3000` (default http port) should be available UI of the app. For example: `http://localhost:3000`

Authentication

Overview

SvanLINK API uses token-based authentication to secure access to all endpoints. Authentication is required for all API requests except for the initial login request.

Authentication Flow

1. **Login**: Send a login request with your password to obtain an authentication token.
2. **Token Usage**: Include the token in all subsequent API requests using the `token` parameter.
3. **Logout**: Use the logout request to invalidate the token when you're done.

Token Lifecycle

Creation: Tokens are generated upon successful login and are cryptographically secure.

Usage: Tokens must be included in the request body of all authenticated endpoints.

Expiration: Tokens remain valid until explicitly logged out or the SvanLINK application is restarted.

Invalidation: Tokens can be invalidated by calling the logout endpoint or by restarting the SvanLINK application.

Security Considerations

Token Storage: Store tokens securely and never expose them in client-side code or logs.

HTTPS/WSS: Use secure connections (HTTPS for HTTP API, WSS for WebSocket) when transmitting tokens over the network.

Token Rotation: Regularly log out and re-authenticate to obtain new tokens.

Network Security: Ensure your network is secure, especially when using the default localhost configuration.

Password Security: Use strong passwords and change the default password using the setNewPassword endpoint.

Error Handling

When authentication fails, the API will return an error response:

```
{
  "Request": "endpointName",
  "Status": "error",
  "StatusMessage": "Invalid token"
}
```

WebSocket Authentication

For WebSocket connections, authentication can be performed in two ways:

1. **Login Request**: Send a login request as the first message after establishing the WebSocket connection.
2. **Token Only**: Send just the token string (without wrapping it in a request object) if you already have a valid token.

Example Authentication Flow

```
// 1. Login to obtain token
{
  "Request": "login",
  "Params": {
    "password": "your_password"
  }
}

// 2. Use token in subsequent requests
{
  "Request": "getStatus",
  "token": "your_token_here"
}

// 3. Logout when done
{
  "Request": "logout",
  "token": "your_token_here"
 }
```

General Error Handling

Overview

All SvanLINK API endpoints follow a consistent error response format. Understanding these error patterns will help you handle errors gracefully in your applications.

Standard Error Response Format

When an error occurs, the API returns a JSON response with the following structure:

```
{
  "Request": "endpointName",
  "Status": "error",
  "StatusMessage": "Description of the error"
}
```

Error Response Fields

`Request`: Reflects back the original request type that caused the error.

`Status`: Always set to `"error"` for failed requests.

`StatusMessage`: A human-readable description of what went wrong.

Common Error Types

Authentication Errors: Invalid or missing tokens, wrong passwords

Parameter Errors: Missing required parameters, invalid parameter formats

Device Errors: Device not found, device not connected, device communication issues

File Errors: File not found, file access denied, file format issues

System Errors: Internal server errors, unhandled exceptions

General Error Example

For unhandled or unexpected errors, the API returns a generic error message:

```
{
  "Request": "endpointName",
  "Status": "error",
  "StatusMessage": "An unhandled error occurred during execution. Please try again!"
}
```

Error Handling Best Practices

Always check the Status field: Verify if the request was successful before processing the response.

Handle specific errors: Check the StatusMessage for specific error conditions and handle them appropriately.

Implement retry logic: For transient errors, implement exponential backoff retry mechanisms.

Log error details: Log both the Request and StatusMessage for debugging purposes.

User-friendly messages: Translate technical error messages into user-friendly notifications.

HTTP Status Codes

In addition to the JSON error response, the API may also return appropriate HTTP status codes:

200 OK: Request successful (even if Status is "error" in JSON)

400 Bad Request: Invalid request format or parameters

401 Unauthorized: Authentication required or invalid credentials

403 Forbidden: Valid authentication but insufficient permissions

404 Not Found: Endpoint or resource not found

500 Internal Server Error: Server-side error

`WebSocket Configuration`

Description:

This configuration is used to set up WebSocket communication. The configuration specifies the intervals at which different types of data are sent automatically. The settings are saved, and on reconnection, SvanLINK will continue to send results according to the configured intervals. The maximum number of simultaneous connections is 20 clients.

The configuration must be sent via WebSocket connection. Default port is `8001`

From version 1.1.0 was introduced such a concept as channels. Channel represents under a name a request (or set of requests) which will be preconfigured and sent automatically according to configured intervals. This allows to listen to different sets of requests by different clients.

Authentication:

When the WebSocket connection is opened, the first step is to authenticate. This can be done by sending a login request or by sending only the token (as a string) which was generated previously.

Request Format:

Request Body:

To configure the broadcast of data the SvanLINK will accept a list of "Requests". Each item in the list contains a desired type of request (the requests are described below in the documentation) and an interval (ms) at which the data must be sent.

```
{
  "Channel": "main",  //Optional, default is "main"
  "Requests": [
    {
      "Request": "getResults",
      "Interval": 1000 // Interval in milliseconds (ms). Minimum interval is 100ms.
    },
    {
      "Request": "getSpectrumResults",
      "Interval": 5000 // Interval in milliseconds (ms). Minimum interval is 100ms.
    }
  ]
}
```

`Channel`: The name of the channel. Default is `"main"`.

`Requests`: An array of request objects.

`Request`: The type of data request. Currently available requests are:

- `getFileList` - Get list of files
- `getResults` - Get measurement results
- `getSpectrumResults` - Get spectrum results
- `getStatus` - Get device statuses
- `getVersion` - Get version information
- `licenseStatus` - Get license status
- `sendRawCommand` - Send raw command to device
- `startMeasurement` - Start measurement

`Interval`: The interval in milliseconds (ms) at which the data is sent. Minimum interval is 100ms.

Success Response:

The response indicates that the configuration has been successfully applied, and data will be sent automatically according to the configured intervals.

Response Format:

```
{
  "Channel": "main",
  "Status": "ok",
  "Response": {
    "message": "Configuration applied successfully"
  }
}
```

`message`: A message indicating that the configuration has been successfully applied.

Channel Management Workflow:

1. Use getChannelList to see all available channels
2. Use subscribeChannel to subscribe to a channel
3. Use deleteChannel to remove unwanted channels
4. Recreate channels by sending WebSocket configuration messages

Securing the connection (optional)

Description:

To secure the connection between the client and the SvanLINK application, you can configure SSL certificates. This will enable encrypted communication, ensuring that data transmitted between the client and the server is secure.

Configuration Steps:

1. Obtain SSL certificates:
   You need two files: `cert.pem` (the certificate) and `key.pem` (the private key).
2. Place the certificate and key files in the `certificates` folder within the SvanLINK application directory:

```
mv path/to/cert.pem certificates/
mv path/to/key.pem certificates/
```

3. On startup, SvanLINK will automatically detect the presence of these files and switch on encryption for communication.

Verification:

Once the certificates are in place, restart the SvanLINK application. You should see a log message indicating that the WebSocket server is starting with SSL:

```
"Starting UI with SSL" or
"Starting TCP server with SSL" or
"Starting WebSocket server with SSL"
```

If the certificates are not found, the WebSocket server will start without SSL:

```
"Starting UI without SSL" or
"Starting TCP server without SSL" or
"Starting WebSocket server without SSL"
```

MQTT Configuration

Description:

MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol designed for IoT (Internet of Things) applications. SvanLINK supports MQTT to enable automatic publishing of measurement data, device status, and other information to an MQTT broker, allowing integration with external systems, dashboards, and data processing platforms.

How MQTT Works in SvanLINK:

SvanLINK acts as an MQTT client that connects to an MQTT broker. Once configured, SvanLINK can automatically publish data to the broker at specified intervals. The system uses a topic-based configuration similar to WebSocket channels, where each topic represents a set of requests that are published periodically.

Key Concepts:

MQTT Broker: The server that receives and distributes messages. You need to configure the broker host, port, and authentication credentials.

Topics: Named channels where data is published. Each topic can have multiple request types configured with different intervals.

Automatic Publishing: Once configured, SvanLINK automatically publishes data according to the configured intervals without requiring individual API requests.

Persistent Configuration: MQTT topics and broker settings are saved and persist across application restarts.

Configuration Workflow:

1. **Configure MQTT Broker**: Use setMqttConfig to set up the broker connection (host, port, username, password, SSL certificate if needed).
2. **Verify Configuration**: Use getMqttConfig to retrieve and verify the current broker settings.
3. **Create Topics**: Use addMqttTopic to configure which data requests should be published and at what intervals.
4. **Manage Topics**: Use getMqttTopicList to view all configured topics, and deleteMqttTopic to remove unwanted topics.

Available MQTT API Requests:

The following API endpoints are available for configuring and managing MQTT functionality:

setMqttConfig - Configure the MQTT broker connection settings (host, port, credentials, SSL certificate).

getMqttConfig - Retrieve the current MQTT broker configuration.

addMqttTopic - Create or update an MQTT topic with specified requests and intervals.

deleteMqttTopic - Delete a specific MQTT topic and stop its automatic publishing.

getMqttTopicList - Get a list of all configured MQTT topics and their configurations.

Supported Request Types:

The following request types can be configured for automatic MQTT publishing:

`getFileList` - Get list of files

`getResults` - Get measurement results

`getSpectrumResults` - Get spectrum results

`getStatus` - Get device statuses

`getVersion` - Get version information

`licenseStatus` - Get license status

`sendRawCommand` - Send raw command to device

`startMeasurement` - Start measurement

Message Format:

When data is published to the MQTT broker, each message includes:

Topic Name: The configured topic name (e.g., `"main"`, `"deviceStatus"`)

Timestamp: A timestamp in the format `"YYYY-MM-DDTHH:mm:ss.SSS"`

Data: The response data from the configured request, formatted as JSON

Security Considerations:

Authentication: MQTT broker credentials (username and password) are stored securely in a separate file.

SSL/TLS: You can configure SSL certificates for secure encrypted connections to the MQTT broker.

Network Security: Ensure your MQTT broker is properly secured and accessible only to authorized clients.

Notes:

The minimum interval for publishing requests is 100 milliseconds (ms).

MQTT topics work similarly to WebSocket channels, allowing multiple clients to subscribe to the same data streams.

If the MQTT broker connection is lost, SvanLINK will attempt to reconnect automatically.

Code Examples

```
// JavaScript
async function makeRequest() {
  try {
    // Create the payload
    const payload = {
      Request: "getResults",
      Params: {
        results: ["LAeq", "LCeq"],
        devices: [123456, 654235],
        average: "true"
      },
      token: "userToken"
    };

    // Send the POST request
    const response = await fetch('http://localhost:8000/', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(payload)
    });

    // Check if response is OK
    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }

    // Parse and print the JSON response
    const data = await response.json();
    console.log(JSON.stringify(data, null, 2));
  } catch (error) {
    console.error('Error:', error.message);
  }
}

// Execute the request
makeRequest();
```

```
# Python
import requests
import json

def make_request():
  try:
    # Create the payload
    payload = {
      "Request": "getResults",
      "Params": {
        "results": ["LAeq", "LCeq"],
        "devices": [123456, 654235],
        "average": "true"
      },
      "token": "userToken"
    }

    # Send the POST request
    url = "http://localhost:8000/"
    response = requests.post(url, json=payload)

    # Check if response is OK
    response.raise_for_status()

    # Parse and print the JSON response
    print(json.dumps(response.json(), indent=2))
  except requests.exceptions.RequestException as e:
    print(f"Error: {e}")

# Execute the request
make_request()
```

```
# cURL
curl -X POST http://localhost:8000/ \
  -H "Content-Type: application/json" \
  -d '{
    "Request": "getResults",
    "Params": {
      "results": ["LAeq", "LCeq"],
      "devices": [123456, 654235],
      "average": "true"
    },
    "token": "userToken"
  }'
```

```
//Java
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import com.google.gson.Gson;
import com.google.gson.JsonObject;
import com.google.gson.JsonArray;

public class HttpPostExample {
  public static void main(String[] args) {
    try {
      // Create the JSON payload
      JsonObject payload = new JsonObject();
      payload.addProperty("Request", "getResults");

      JsonObject params = new JsonObject();
      JsonArray results = new JsonArray();
      results.add("LAeq");
      results.add("LCeq");
      params.add("results", results);

      JsonArray devices = new JsonArray();
      devices.add(123456);
      devices.add(654235);
      params.add("devices", devices);

      params.addProperty("average", "true");
      payload.add("Params", params);
      payload.addProperty("token", "userToken");

      // Convert payload to string
      Gson gson = new Gson();
      String jsonPayload = gson.toJson(payload);

      // Create HTTP client
      HttpClient client = HttpClient.newHttpClient();

      // Build the HTTP request
      HttpRequest request = HttpRequest.newBuilder()
        .uri(URI.create("http://localhost:8000/"))
        .header("Content-Type", "application/json")
        .POST(HttpRequest.BodyPublishers.ofString(jsonPayload))
        .build();

      // Send the request and get the response
      HttpResponse response = client.send(request,
HttpResponse.BodyHandlers.ofString());

      // Parse and print the JSON response
      JsonObject responseJson = gson.fromJson(response.body(), JsonObject.class);
      System.out.println(gson.toJson(responseJson));

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

```
// Go
package main

import (
        "bytes"
        "encoding/json"
        "fmt"
        "net/http"
)

func main() {
        // Define the payload structure
        type Params struct {
                Results  []string `json:"results"`
                Devices  []int    `json:"devices"`
                Average  string   `json:"average"`
        }

        type Payload struct {
                Request string  `json:"Request"`
                Params  Params  `json:"Params"`
                Token   string  `json:"token"`
        }

        // Create the payload
        payload := Payload{
                Request: "getResults",
                Params: Params{
                        Results:  []string{"LAeq", "LCeq"},
                        Devices:  []int{123456, 654235},
                        Average:  "true",
                },
                Token: "userToken",
        }

        // Marshal the payload to JSON
        jsonData, err := json.Marshal(payload)
        if err != nil {
                fmt.Println("Error marshaling JSON:", err)
                return
        }

        // Create the HTTP request
        url := "http://localhost:8000/"
        req, err := http.NewRequest("POST", url, bytes.NewBuffer(jsonData))
        if err != nil {
                fmt.Println("Error creating request:", err)
                return
        }
        req.Header.Set("Content-Type", "application/json")

        // Send the request
        client := &http.Client{}
        resp, err := client.Do(req)
        if err != nil {
                fmt.Println("Error sending request:", err)
                return
        }
        defer resp.Body.Close()

        // Read and parse the response
        var result map[string]interface{}
```

```
        if err := json.NewDecoder(resp.Body).Decode(&result); err != nil {
                fmt.Println("Error decoding response:", err)
                return
        }

        // Pretty print the JSON response
        resultJSON, err := json.MarshalIndent(result, "", "  ")
        if err != nil {
                fmt.Println("Error formatting response:", err)
                return
        }
        fmt.Println(string(resultJSON))
}
```

API Documentation - SvanLINK v1.1.7

`login`

Description:

The `login` API is used to authenticate a user by providing a password. If the password is correct, the user is granted access to the system and receives a token.

Request Format:

Request Body:

```
{
  "Request": "login",
  "Params": {
    "password": "Svantek312"
  }
}
```

`Request`: The type of request, which is `"login"` in this case.

`Params`: An object containing the parameters for the request.

- `password`: The password for authentication.

Success Response:

If the password is correct, the response will indicate a successful login and provide a token.

Response Format:

```
{
  "Request": "login",
  "Status": "ok",
  "Response": {
    "token": "userToken",
    "message": "Login successful"
  }
}
```

`Request`: Reflects back the original request type (`"login"`).

`Status`: The status of the request (`"ok"` for successful requests).

`Response`: An object containing the response data.

`token`: The token provided upon successful login. **It must be supplied into any other request described below.**

`message`: A message indicating that the login was successful.

Error Response:

If the password is incorrect, the response will indicate an error.

Response Format:

```
{
  "Request": "login",
  "Status": "error",
  "StatusMessage": "Wrong password"
}
```

`Request`: Reflects back the original request type (`"login"`).

`Status`: The status of the request (`"error"` for failed requests).

`StatusMessage`: A message indicating the reason for the error (e.g., `"Wrong password"`).

Note. It is highly recommended to change it to keep your data and settings safe. Use `setNewPassword` request to do this.

`logout`

Description:

The `logout` API is used to log out a user by invalidating the provided token. This will end the user's session.

Request Format:

Request Body:

```
{
  "Request": "logout",
  "token": "userToken"
}
```

`Request`: The type of request, which is `"logout"` in this case.

`Params`: An object containing the parameters for the request.
- `token`: The token for the user session to be invalidated.

Success Response:

If the token is valid, the response will indicate a successful logout.

Response Format:

```
{
  "Request": "logout",
  "Status": "ok",
  "Response": {
    "message": "Logout successful"
  }
}
```

`Request`: Reflects back the original request type (`"logout"`).

`Status`: The status of the request (`"ok"` for successful requests).

`Response`: An object containing the response data.
- `message`: A message indicating that the logout was successful.

Error Response:

If the token is invalid, the response will indicate an error.

Response Format:

```
{
  "Request": "logout",
  "Status": "error",
  "StatusMessage": "Invalid token"
}
```

`Request`: Reflects back the original request type (`"logout"`).

`Status`: The status of the request (`"error"` for failed requests).

`StatusMessage`: A message indicating the reason for the error (e.g., `"Invalid token"`).

`setNewPassword`

Description:

The `setNewPassword` API allows an authenticated user to change their password. The user must provide their current password and enter the new password twice for confirmation. The new password must be at least 8 characters long and contain at least one uppercase letter, one lowercase letter.

Request Format:

Request Body:

```
{
  "Request": "setNewPassword",
  "Params": {
    "oldPassword": "currentPassword",      // The current password.
    "newPassword": "newPassword",       // The new password.
    "newPassword2": "newPassword"       // Repeat of the new password for confirmation.
  },
  "token": "userToken"
}
```

`oldPassword` (required): The user's current password.

`newPassword` (required): The new password to set.

`newPassword2` (required): The new password repeated for confirmation. Must match `newPassword`.

`token` (required): Authentication token for the session.

Success Response:

If the password is changed successfully, the response will indicate success.

Response Format:

```
{
  "Request": "setNewPassword",
  "Status": "ok",
  "Response": {
    "message": "Password changed successfully"
  }
}
```

`Request`: Reflects back the original request type (`"setNewPassword"`).

`Status`: The status of the request (`"ok"` for successful requests).

`Response`: An object containing the result message.

Error Responses:

If the request fails, an error response is returned.

Incorrect Password:

```
{
  "Request": "setNewPassword",
  "Status": "error",
  "StatusMessage": "Incorrect password"
}
```

`StatusMessage`: A message describing the reason for the error (e.g., incorrect old password, new passwords do not match, etc.).

Note. To reset the password to default it is needed to remove the `password` file with the same name.

On Linux systems it is located in the SvanLINK's folder.

On Windows it is located in `C:\Users\YourUserName\AppData\Local\Svantek\SvanLINK\`

`licenseStatus`

Description:

The `licenseStatus` API retrieves the current license status of the device. The response includes the status of the license, the license end date, the device ID, and a message regarding the license status.

Request Format:

Request Body:

```
{
  "Request": "licenseStatus",
  "token": "userToken"
}
```

Success Response:

The response contains the current license status of the device.

Response Format:

```
{
  "Request": "licenseStatus",
  "Status": "ok",
  "Response": {
    "licenseStatus": "active",
    "deviceId": "XXXXXXXXXXXXXXXX",
    "licenseMessage": "The license is active"
  }
}
```

`licenseStatus`: The current status of the license (active, expired, noLicense, error).

`deviceId`: The ID of the device.

`licenseMessage`: A message regarding the license status.

`setLicense`

Description:

The `setLicense` API activates a new license key for the device. You can provide the license key encoded in a Base64 string, and the API will activate it for the device.

Important note. There are 2 types of licenses: hardware and device. "Hardware licence" attaches the app to the current hardware configuration (PC, laptop, server, Raspberry Pi etc) on which the app is running. The "device license" is attached to Svantek device.

Request Format:

Request Body:

```
{
  "Request": "setLicense",
  "Params": {
    "key": "License key file encoded in Base64" // License key encoded in Base64
  },
  "token": "userToken"
}
```

`key`: The license key encoded in a Base64 string.

Success Response:

The response indicates whether the license key was successfully activated for the device.

Response Format:

```
{
  "Request": "setLicense",
  "Status": "ok",
  "Response": {
    "licenseStatus": "active",
    "deviceId": "1424422045401",
    "licenseMessage": "The license was activated successfully"
  }
}
```

`licenseStatus`: The current status of the license (e.g., "active").

`deviceId`: The ID of the device.

`licenseMessage`: A message regarding the license status.

`getStatus`

Description:

The `getStatus` API retrieves the status of specified devices. You can request specific device serial numbers, and the API will return the corresponding status data in the response.

Request Format:

Request Body:

```
{
  "Request": "getStatus",
  "Params": {
    "devices": [113200, 223200] // Optional: List of device serial numbers.
  },
  "token": "userToken"
}
```

`devices` (optional): A list of device serial numbers. If not provided, status data for all available devices will be returned.

Success Response:

The response contains the status data for each requested device.

Response Format:


```
{
  "Request": "getStatus",
  "Status": "ok",
  "Response": [
    {
      "serial": 113200,
      "type": "SV 303",
      "deviceName": "Demo_1",
      "firmware": "1.03.9",
      "battery": 90,
      "memorySize": 16874,
      "memoryFree": 30,
      "deviceWarning": [
        "Message 1 from device",
        "Message 2 from device"
      ],
      "measurementStatus": "start",
      "measurementType": "1/3 Octave",
      "intPeriod": 30
    },
    {
      "serial": 223200,
      "type": "SV 303",
      "deviceName": "Demo_2",
      "firmware": "1.03.7",
      "battery": 33,
      "memorySize": 65535,
      "memoryFree": 54,
      "deviceWarning": [
        "Message 1 from device",
        "Message 2 from device"
      ],
      "measurementStatus": "stop",
      "measurementType": "1/1 Octave",
      "intPeriod": 0
    }
  ]
}
```


Response Fields:

`Request`: Reflects back the original request type (`"getStatus"`).

`Status`: The status of the request (`"ok"` for successful requests).

`Response`: An array of objects containing status data for the requested devices.

Device Object:

`serial`: The serial number of the device.

`type`: The type of the device (e.g., `"SV 303"`).

`deviceName`: The name of the device.

`firmware`: The firmware version of the device.

`battery`: The battery level of the device (percentage).

`memorySize`: The total memory size of the device (MB).

`memoryFree`: The free memory of the device (percentage).

`deviceWarning`: An array of warning messages from the device.

`measurementStatus`: The measurement status of the device (e.g., `"start"`, `"stop"`).

`measurementType`: The type of measurement being performed (e.g., `"1/3 Octave"`, `"1/1 Octave"`).

`intPeriod`: The integration period for the measurement (seconds, `0` for infinity).

`getResults`

Description:

The `getResults` API retrieves measurement results for specified devices. You can request specific result types and device serial numbers, and the API will return the corresponding data in the response.

Request Format:

Request Body:

```
{
  "Request": "getResults",
  "Params": {
    "results": ["LAeq", "LCeq"],      // Optional: List of result types to fetch.
    "devices": [123456, 654235],      // Optional: List of device serial numbers.
    "average": "true",                // Required: Whether to return averaged results.
    "vibrationsDB": false             // Optional: Whether to show vibrations in dB
instead of mm/s or m/s².
  },
  "token": "userToken"
}
```

`results` (optional): A list of result types to retrieve. Examples include:

- `"LAeq"`: Equivalent continuous sound level A-weighted.
- `"LCeq"`: Equivalent continuous sound level C-weighted.

If omitted, all available result types will be returned.

`devices` (optional): A list of device serial numbers. If not provided, data for all available devices will be returned.

`average` (required): A boolean value (`"true"` or `"false"`) indicating whether to return averaged results.

`vibrationsDB` (optional): A boolean value indicating whether to show vibrations in dB instead of mm/s or m/s². Defaults to `false`.

`token` (required): Authentication token for the session.

Success Response:

The response contains the requested measurement data for each device.

Response Format:

```
{
  "Request": "getResults",
  "Status": "ok",
  "Response": [
    {
      "serial": 123456,
      "type": "SV 303",
      "result": [
        {
          "name": "LAeq",
          "value": 100.0,
          "unit": "dB"
        },
        {
          "name": "LCeq",
          "value": 112.1,
          "unit": "dB"
        }
      ]
    },
    {
      "serial": 654235,
      "type": "SV 303",
      "result": [
        {
          "name": "LAeq",
          "value": 98.2,
          "unit": "dB"
        },
        {
          "name": "LCeq",
          "value": 88.8,
          "unit": "dB"
        }
      ]
    }
  ]
}
```

Response Fields:

`Request`: Reflects back the original request type (`"getResults"`).

`Status`: The status of the request (`"ok"` for successful requests).

`Response`: An array of objects containing measurement data for the requested devices.

Device Object:

`serial`: The serial number of the device.

`type`: The type of the device (e.g., `"303"`).

`result`: An array containing the results for the device.

- `name`: The type of result (e.g., `"LAeq"`, `"LCeq"`).

`value`: The measured value for the result.

`unit`: The unit of the result (e.g., `"dB"` for decibels).

Error Responses:

Missing Parameters:

```
{
  "Request": "getResults",
  "Status": "error",
  "Error": "Missing required parameters"
}
```

Invalid Parameter Format:

```
{
  "Request": "getResults",
  "Status": "error",
  "Error": "Invalid parameter format"
}
```

Notes:

If neither `results` nor `devices` are provided, the API will return all available data for all devices.

The `average` parameter is mandatory and determines whether the results should be averaged across measurements.

`getSpectrumResults`

Description:

The `getSpectrumResults` API retrieves spectrum results for specified devices. You can request specific spectrum types and device serial numbers, and the API will return the corresponding data in the response.

Request Format:

Request Body:

```
{
  "Request": "getSpectrumResults",
  "Params": {
    "devices": [123456, 654235],            // Optional: List of device serial numbers.
    "vibrationsDB": false                   // Optional: Whether to show vibrations in
dB instead of mm/s or m/s².
  },
  "token": "userToken"
}
```

`devices` (optional): A list of device serial numbers. If not provided, data for all available devices will be returned.

`vibrationsDB` (optional): A boolean value indicating whether to show vibrations in dB instead of mm/s or m/s². Defaults to `false`.

`token` (required): Authentication token for the session.

Success Response:

The response contains the requested spectrum data for each device.

Response Format:

```
{
  "Request": "getSpectrumResults",
  "Status": "ok",
  "Response": [
    {
      "serial": 123456,
      "type": "SV 303",
      "status": "ok",
      "measurement": "start",
      "resultLabel": "LZeq",
      "resultType": "1/1 Octave",
      "measurementMode": "sound",
      "results": [
        {name: "31.5", value: 59.55, unit: "dB"}
        {name: "63", value: 47.22, unit: "dB"}
        {name: "125", value: 41.01, unit: "dB"}
        {name: "250", value: 41.21, unit: "dB"}
        {name: "500", value: 38.36, unit: "dB"}
        {name: "1.0k", value: 38.69, unit: "dB"}
        {name: "2.0k", value: 40.36, unit: "dB"}
        {name: "4.0k", value: 35.02, unit: "dB"}
        {name: "8.0k", value: 30.13, unit: "dB"}
        {name: "16k", value: 32.06, unit: "dB"}
        {name: "A", value: 44.92, unit: "dB"}
        {name: "C", value: 61.96, unit: "dB"}
        {name: "Z", value: 72.56, unit: "dB"}
      ]
    },
    {
      "serial": 654235,
      "type": "SV 303",
      "status": "ok",
      "measurement": "start",
      "resultLabel": "LZeq",
      "resultType": "1/1 Octave",
      "measurementMode": "sound",
      "results": [
        {name: "31.5", value: 60.11, unit: "dB"}
        {name: "63", value: 44.22, unit: "dB"}
        {name: "125", value: 32.01, unit: "dB"}
        {name: "250", value: 41.23, unit: "dB"}
        {name: "500", value: 38.54, unit: "dB"}
        {name: "1.0k", value: 38.69, unit: "dB"}
        {name: "2.0k", value: 40.36, unit: "dB"}
        {name: "4.0k", value: 35.34, unit: "dB"}
        {name: "8.0k", value: 35.13, unit: "dB"}
        {name: "16k", value: 32.23, unit: "dB"}
        {name: "A", value: 44.92, unit: "dB"}
        {name: "C", value: 61.96, unit: "dB"}
        {name: "Z", value: 66.56, unit: "dB"}
      ]
    }
  ]
}
```

Response Fields:

`Request`: Reflects back the original request type (`"getSpectrumResults"`).

`Status`: The status of the request (`"ok"` for successful requests).

`Response`: An array of objects containing spectrum data for the requested devices.

Device Object:

`serial`: The serial number of the device.

`type`: The type of the device (e.g., `"303"`).

`status`: "ok" if no error detected.

`measurement`: The measurement status of the device (e.g., `"start"`, `"stop"`).

`resultLabel`: The label of the spectrum result.

`resultType`: The type of spectrum result (e.g., `"1/1 Octave"`, `"1/3 Octave"`).

`measurementMode`: The mode of measurement `"sound|vibrations"`.

`results`: An array containing the spectrum data for the device.

- `name`: The type of frequency.
- `data`: The spectrum data array.
- `unit`: The unit of the result (e.g., `"dB"` for decibels).

Error Responses:

Missing Parameters:

```
{
  "Request": "getSpectrumResults",
  "Status": "error",
  "Error": "Missing required parameters"
}
```

Invalid Parameter Format:

```
{
  "Request": "getSpectrumResults",
  "Status": "error",
  "Error": "Invalid parameter format"
}
```

Notes:

If neither `spectrumTypes` nor `devices` are provided, the API will return all available spectrum data for all devices.

`startMeasurement`

Description:

The `startMeasurement` API starts measurements for specified devices. You can request specific measurement types and device serial numbers, and the API will start the corresponding measurements.

Request Format:

Request Body:

```
{
  "Request": "startMeasurement",
  "Params": {
    "devices": [123456, 654235]           // Optional: List of device serial numbers.
  },
  "token": "userToken"
}
```

`devices` (optional): A list of device serial numbers. If not provided, measurements for all available devices will be started.

Success Response:

The response confirms that the requested measurements have been started for each device.

Response Format:

```
{
  "Request": "startMeasurement",
  "Status": "ok"
}
```

Response Fields:

`Request`: Reflects back the original request type (`"startMeasurement"`).

`Status`: The status of the request (`"ok"` for successful requests).

Notes:

If neither `measurementTypes` nor `devices` are provided, the API will start all available measurements for all devices.

`stopMeasurement`

Description:

The `stopMeasurement` API stops measurements for specified devices. You can request specific measurement types and device serial numbers, and the API will stop the corresponding measurements.

Request Format:

Request Body:

```
{
  "Request": "stopMeasurement",
  "Params": {
    "devices": [123456, 654235]           // Optional: List of device serial numbers.
  },
  "token": "userToken"
}
```

`devices` (optional): A list of device serial numbers. If not provided, measurements for all available devices will be stopped.

Success Response:

The response confirms that the requested measurements have been stopped for each device.

Response Format:

```
{
  "Request": "stopMeasurement",
  "Status": "ok",
}
```

Response Fields:

`Request`: Reflects back the original request type (`"stopMeasurement"`).

`Status`: The status of the request (`"ok"` for successful requests).

Notes:

If neither `measurementTypes` nor `devices` are provided, the API will stop all available measurements for all devices.

`getConfig`

Description:

The `getConfig` API retrieves the current configuration for specified devices. You can request specific configuration parameters and device serial numbers, and the API will return the corresponding configuration data in the response.

Request Format:

Request Body:

```
{
  "Request": "getConfig",
  "token": "userToken"
}
```

Success Response:

The response contains the requested configuration data for each device.

Response Format:

```
{
    "Request": "getConfig",
    "Status": "ok",
    "Response": {
          "app": {
            "tcpPort": 8000,
            "echo": true,
            "wsEnabled": false,
            "wsPort": 8001,
            "activeUI": true,
            "disabledExternalLogin": true,
            "svannet": true,
            "mqtt": {
              "enabled": true,
              "username": "admin",
              "port": 1883,
              "host": "127.0.0.1"
            }
        },
        "ui": {
            "liveView": {
                "resultTypes": "LAF;LAeq;LCpeak",
                "currentTab": "liveResultsTab"
            },
            "thresholdView": {
                "thresholdOrange": 45,
                "thresholdRed": 75,
                "k": 0,
                "resultType": "LAF"
            }
        }
    }
}
```

Response Fields:

`Request`: Reflects back the original request type (`"getConfig"`).

`Status`: The status of the request (`"ok"` for successful requests).

`Response`: An object containing configuration data for the requested devices.

`app`: An object containing application configuration.

Response Object:

- `tcpPort`: The TCP port number (e.g., `8000`).
- `echo`: A boolean indicating if echo is enabled (e.g., `true`). If it is enabled, the server will echo back the request in `"Echo"` parameter.
- `wsEnabled`: A boolean indicating if WebSocket is enabled (e.g., `false`).
- `wsPort`: The WebSocket port number (e.g., `8001`).
- `activeUI`: A boolean indicating if the web UI is active (e.g., `true`).

`disabledExternalLogin`: A boolean indicating if live results are availaible without password for external clients (e.g., `true`).

`svannet`: A boolean indicating if the SvanLINK will connect Svantek devices to SvanNET (e.g., `true`).

`mqtt`: An object containing MQTT configuration.

- `enabled`: A boolean indicating if the MQTT is enabled (e.g., `true`).
- `username`: The username for the MQTT (e.g., `"admin"`).
- `port`: The port number for the MQTT (e.g., `1883`).
- `host`: The host for the MQTT (e.g., `"127.0.0.1"`).

`ui` (optional): An object containing web UI configuration.

`setConfig`

Description:

The `setConfig` API sets the configuration for specified devices. You can provide specific configuration parameters and device serial numbers, and the API will update the corresponding configuration data.

Note: If the user sends `"default": true` in the request, all settings will be reset to their default values.

Request Format:

Request Body:

```
{
  "Request": "setConfig",
  "Params": {
    "app": {
      "tcpPort": 8000,
      "echo": true,
      "wsEnabled": false,
      "wsPort": 8001,
      "activeUI": true,
      "disabledExternalLogin": true,
      "svannet": true,
      "mqtt": {
        "enabled": true,
        "username": "admin",
        "port": 1883,
        "host": "127.0.0.1",
        "ssl": true,
        "caCert": "base64_encoded_cert",
        "certFile": "base64_encoded_cert",
        "keyFile": "base64_encoded_key"
      }
    },
    "ui": {      // Paarameters that are used to configure the web UI.
      "liveView": {
        "resultTypes": "LAF;LAeq;LCpeak",
        "currentTab": "liveResultsTab"
      },
      "thresholdView": {
        "thresholdOrange": 45,
        "thresholdRed": 75,
        "k": 0,
        "resultType": "LAF"
      }
    }
  },
  "token": "userToken"
}
```

`app` (required): An object containing application configuration.

`tcpPort`: The TCP port number (e.g., `8000`).

`echo`: A boolean indicating if echo is enabled (e.g., `true`). If it is enabled, the server will echo back the request in `"Echo"` parameter.

`wsEnabled`: A boolean indicating if WebSocket is enabled (e.g., `false`).

`wsPort`: The WebSocket port number (e.g., `8001`).

`activeUI`: A boolean indicating if the web UI is active (e.g., `true`).

`disabledExternalLogin`: A boolean indicating if ;ive results are availaible without password for external clients (e.g., `true`).

`svannet`: A boolean indicating if the SvanLINK will connect Svantek devices to SvanNET (e.g., `true`).

`mqtt`: An object containing MQTT configuration.

- `enabled`: A boolean indicating if the MQTT is enabled (e.g., `true`).
- `username`: The username for the MQTT (e.g., `"admin"`).
- `port`: The port number for the MQTT (e.g., `1883`).
- `host`: The host for the MQTT (e.g., `"127.0.0.1"`).
- `ssl`: A boolean indicating if the MQTT is using SSL/TLS (e.g., `true`).
- `caCert`: The CA certificate for the MQTT (e.g., `"base64_encoded_cert"`).
- `certFile`: The certificate file for the MQTT (e.g., `"base64_encoded_cert"`).
- `keyFile`: The key file for the MQTT (e.g., `"base64_encoded_key"`).

`ui` (optional): An object containing web UI configuration.

Success Response:

The response confirms that the configuration has been set for each device.

Response Format:

```
{
  "Request": "setConfig",
  "Status": "ok"
}
```

Response Fields:

`Request`: Reflects back the original request type (`"setConfig"`).

`Status`: The status of the request (`"ok"` for successful requests).

`getSetup`

Description:

The `getSetup` API retrieves the setup configuration for a specified device. You can request the setup configuration for a specific device serial number, and the API will return the corresponding setup data in the response.

Request Format:

Request Body:

```
{
  "Request": "getSetup",
  "Params": {
    "device": 85609 // Device serial number
  },
  "token": "userToken"
}
```

`device`: The serial number of the device for which the setup configuration is requested.

Success Response:

The response contains the setup configuration data for the requested device.

Response Format:

```
{
  "Request": "getSetup",
  "Status": "ok",
  "Response": {
    "setupfile": "file encoded in Base64 string format"
  }
}
```

`setupfile`: The setup configuration file encoded in a Base64 string format.

setSetup

Description:

The `setSetup` API uploads a setup configuration to a specified device. You can provide the setup configuration file encoded in a Base64 string, and the API will upload it to the specified device.

Request Format:

Request Body:

```
{
  "Request": "setSetup",
  "Params": {
    "device": 85609, // Device serial number
    "overwrite": true, // Whether to overwrite the existing setup
    "file": "file encoded in Base64 string" // Setup configuration file encoded in
Base64
  },
  "token": "userToken"
}
```

`device`: The serial number of the device to which the setup configuration is uploaded.

`overwrite`: A boolean indicating whether to overwrite the existing setup.

`file`: The setup configuration file encoded in a Base64 string.

Success Response:

The response indicates whether the setup configuration was successfully uploaded to the device.

Response Format:

```
{
  "Request": "setSetup",
  "Status": "ok"
}
```

`message`: A message indicating the result of the setup upload.

copySetup

Description:

The `copySetup` API copies a setup configuration from a source device to one or more target devices. You can specify the source device, target devices, and whether to overwrite the existing setup on the target devices.

Request Format:

Request Body:

```
{
  "Request": "copySetup",
  "Params": {
    "sourceDevice": 85609, // Source device serial number
    "targetDevices": [112233, 453789], // List of target device serial numbers
    "overwrite": true // Whether to overwrite the existing setup on the target devices
  },
  "token": "userToken"
}
```

`sourceDevice`: The serial number of the source device from which the setup configuration is copied.

`targetDevices`: A list of serial numbers of the target devices to which the setup configuration is copied.

`overwrite`: A boolean indicating whether to overwrite the existing setup on the target devices.

Success Response:

The response indicates whether the setup configuration was successfully copied to the target devices.

Response Format:

```
{
  "Request": "copySetup",
  "Status": "ok"
}
```

`message`: A message indicating the result of the setup copy operation.

sendRawCommand

Description:

The `sendRawCommand` API allows you to send # commands directly to the connected device. These commands must follow the format and syntax specified in the device's user manual. The API sends the command to the device and returns the response.

Request Format:

Request Body:

```
{
  "Request": "sendRawCommand",
  "Params": {
    "devices": [123456, 654235],      // Optional: List of device serial numbers.
    "command": "#1,U?,N?;"            // The raw command to send to the device
  },
  "token": "userToken"
}
```

`command`: The raw command string to be sent to the device. This must follow the syntax specified in the device's user manual.

`token`: The authentication token for the session.

Success Response:

The response contains the result of the command execution on the device.

Response Format:

```
{
  "Request": "sendRawCommand",
  "Status": "ok",
  "Response": [
    {
      "serial": 123456,
      "response": "#1,U303,N123456;"
    },
    {
      "serial": 654235,
      "response": "#1,U303,N654235;"
    }
  ]
}
```

`result`: A message indicating the result of the command execution.

Notes:

Ensure that the `command` parameter follows the syntax and format specified in the device's user manual.

The `token` parameter is mandatory and must be valid for the session.

Use this API with caution, as sending incorrect commands may result in unexpected behavior or errors on the device.

`getFileList`

Description:

The `getFileList` API retrieves a list of files stored on devices. All parameters in `Params` are optional and act as filters. If a parameter is omitted, no filtering is applied for that field.

Request Format:

Request Body:

```
{
  "Request": "getFileList",
  "Params": {
    "devices": [3502],                    // Optional: List of device serial numbers to
filter.
    "mode": "flat",                       // Optional: "flat" for a flat list, "tree"
for hierarchical (default: "flat").
    "types": ["TXT", "SVL"],              // Optional: List of file types/extensions to
filter.
    "startDate": "2025-05-13 10:30:00",   // Optional: Start date/time (YYYY-MM-DD
HH:mm:ss) to filter files created after this date.
    "endDate": "2025-5-13 11:00:00"       // Optional: End date/time (YYYY-MM-DD
HH:mm:ss) to filter files created before this date.
  },
  "token": "userToken"
}
```

`devices` (optional): List of device serial numbers. Only files from these devices will be listed.

`mode` (optional): `"flat"` for a flat file list, `"tree"` for a directory tree. Default is `"flat"`.

`types` (optional): List of file types/extensions to include (e.g., `"TXT"`, `"SVL"`).

`startDate` (optional): Only include files created after this date/time.

`endDate` (optional): Only include files created before this date/time.

`token` (required): Authentication token.

Success Response:

The response contains a list of files for each device matching the filters, including file name, size, creation date, and type.

Response Format:

```
{
  "Request": "getFileList",
  "Status": "ok",
  "Response": [
    {
      "serial": 123456,
      "type": "SV 303",
      "files": [
        {
          "name": "/S1.SVL",
          "size": 1203,
          "dateCreated": "2025-05-13 10:50:40",
          "type": "TXT"
        },
        {
          "name": "/W1.WAV",
          "size": 270336,
          "dateCreated": "2025-05-13 10:52:28",
          "type": "SVL"
        }
      ]
    }
  ],
}
```

Response Fields:

`Request`: Reflects back the original request type (`"getFileList"`).

`Status`: The status of the request (`"ok"` for successful requests).

`Response`: An array of objects, one per device, each containing:

- `serial`: Device serial number.
- `type`: Device type/model.
- `files`: Array of file objects:
  - `name`: Full file path/name.
  - `size`: File size in bytes.
  - `dateCreated`: File creation date/time (YYYY-MM-DD HH:mm:ss).
  - `type`: File type/extension (e.g., `"TXT"`, `"SVL"`).

Notes:

All `Params` fields are optional and act as filters. If omitted, no filtering is applied for the request.

If `types` is omitted, all file types are included.

The `mode` parameter controls whether the file list is flat or hierarchical.

downloadFile

Description:

The `downloadFile` API allows you to download a specific file from a device. The response is a stream of bytes representing the file content.

Request Format:

Request Body:

```
{
  "Request": "downloadFile",
  "Params": {
    "device": 123456,                  // Required: Device serial number.
    "mode": "byPath|current",          // Required: Mode for file download.
    "value": "/S1.SVL|SVL"     // Required: Depending on mode, either file path or file
type.
  },
  "token": "userToken"
}
```

`device` (required): The serial number or identifier of the device.

`mode` (required): Mode for file download. Use `"byPath"` to specify the file path, or `"current"` to download the current file.

`value` (required): Depending on the mode:

- `"byPath"`: Full file path (e.g., `"/S1.SVL"`).
- `"current"`: File type (e.g., `"SVL"`, `"WAV"`).

`token` (required): Authentication token.

Response:

The response is a stream of bytes representing the requested file. The content type and headers are set for file download.

The file is returned as a binary stream (not JSON).

Appropriate headers (e.g., `Content-Disposition`, `Content-Type`) are set for file download.

Notes:

Use this endpoint to download a single file from the specified device.

Make sure to handle the response as a file/binary stream in your client.

downloadFileList

Description:

The `downloadFileList` API allows you to download multiple files from a device in a single request. The response is a stream of bytes, as a ZIP archive containing the requested files.

Request Format:

Request Body:

```
{
  "Request": "downloadFileList",
  "Params": {
    "device": 123456,                // Required: Device serial number or identifier.
    "files": [
      "path/file001.svl",
      "path/file002.wav"
    ],                               // Required: List of file paths to download.
    "omit": true                     // Optional: Whether to omit files that do not
exist on the device.
  },
  "token": "userToken"
}
```

`device` (required): The serial number or identifier of the device.

`files` (required): List of file paths to download from the device.

`token` (required): Authentication token.

`omit` (optional): If set to `true`, files that do not exist on the device will be omitted in the archive received in respones, otherwise error will be returned. Default is `false`.

Response:

The response is a stream of bytes representing the requested files, typically as a ZIP archive.

The files are returned as a binary stream (not JSON).

Appropriate headers (e.g., `Content-Disposition`, `Content-Type: application/zip`) are set for file download.

Notes:

Use this endpoint to download multiple files from the specified device in a single request.

Make sure to handle the response as a file/binary stream in your client.

`deleteFile`

Description:

The `deleteFile` API allows you to delete a specific file from a device. The request must specify the device serial number and the full path to the file to be deleted.

Request Format:

Request Body:

```
{
  "Request": "deleteFile",
  "Params": {
    "device": 113200,  // Required: Device serial number.
    "path": "/L4.SVL"  // Required: Full file path to delete.
  },
  "token": "userToken"
}
```

`device` (required): The serial number of the device.

`path` (required): The full path to the file to be deleted on the device.

`token` (required): Authentication token.

Success Response:

The response indicates whether the file was successfully deleted from the device.

Response Format:

```
{
  "Request": "deleteFile",
  "Status": "ok",
  "Response": {
    "message": "File deleted successfully"
  }
}
```

`Request`: Reflects back the original request type (`"deleteFile"`).

`Status`: The status of the request (`"ok"` for successful requests).

`Response`: An object containing a message about the result.

Notes:

Use this endpoint to delete a single file from the specified device.

If the file does not exist, an error message will be returned in the response.

Deleting files is irreversible. Use with caution.

`getVersion`

Description:

The `getVersion` API retrieves the current version of the SvanLINK application. This is useful for checking which version of the software is running on the server.

Request Format:

Request Body:

```
{
  "Request": "getVersion",
  "token": "userToken"
}
```

`Request`: The type of request, which is `"getVersion"` in this case.

`token`: (required) Authentication token for the session.

Response:

The response contains the current version of the SvanLINK application.

Response Format:

```
{
  "Request": "getVersion",
  "Status": "ok",
  "Response": {
    "version": "1.1.7"
  }
}
```

`Request`: Reflects back the original request type (`"getVersion"`).

`Status`: The status of the request (`"ok"` for successful requests).

`Response`: An object containing the version string.

checkUpdate

Description:

The `checkUpdate` API checks if a new version of the SvanLINK application is available. It returns information about the latest version, its release date, and changelog.

Request Format:

Request Body:

```
{
  "Request": "checkUpdate",
  "token": "userToken"
}
```

`Request`: The type of request, which is `"checkUpdate"` in this case.

`token`: (required) Authentication token for the session.

Success Response:

The response indicates whether an update is available and provides details about the latest version.

Response Format:

```
{
  "Request": "checkUpdate",
  "Status": "ok",
  "Response": {
    "available": true,
    "critical": false,
    "version": "1.0.0",
    "date": "2025-05-01",
    "changelog": "- Added new features\n- Fixed bugs\n- Improved performance"
  }
}
```

`available`: `true` if a newer version is available, `false` otherwise.

`critical`: `true` if the update is critical, `false` otherwise.

`version`: The latest available version.

`date`: Release date of the latest version (YYYY-MM-DD).

`changelog`: Description of changes in the latest version.

Error Response:

```
{
  "Request": "checkUpdate",
  "Status": "error",
  "StatusMessage": "Unable to check for updates"
}
```

`StatusMessage`: A message describing the reason for the error (e.g., network issues, authentication failure).

Notes:

This endpoint is used to inform users if a new version of SvanLINK is available for download.

Requires a valid authentication token.

downloadUpdate

Description:

The `downloadUpdate` API downloads the latest available update for the SvanLINK application. It initiates the download process or returns the current status if already in progress or completed.

Request Format:

Request Body:

```
{
  "Request": "downloadUpdate",
  "token": "userToken"
}
```

`Request`: The type of request, which is `"downloadUpdate"` in this case.

`token`: (required) Authentication token for the session.

Success Response:

The response indicates the status of the update download process.

Response Format:

```
{
  "Request": "downloadUpdate",
  "Status": "ok",
  "Response": {
    "status": "downloading|downloaded"
  }
}
```

`status`: `"downloaded"` if the update has been downloaded, `"downloading"` if the download is in progress.

Error Response:

```
{
  "Request": "downloadUpdate",
  "Status": "error",
  "StatusMessage": "Unable to download update"
}
```

`StatusMessage`: A message describing the reason for the error (e.g., network issues, authentication failure).

Notes:

This endpoint is used to download the latest update for SvanLINK.

Requires a valid authentication token.

After a successful download, use `installUpdate` to install the update.

`installUpdate`

Description:

The `installUpdate` API installs the latest downloaded update for the SvanLINK application. It starts the update process and may require elevated privileges depending on the operating system.

Request Format:

Request Body:

```
{
  "Request": "installUpdate",
  "token": "userToken"
}
```

`Request`: The type of request, which is `"installUpdate"` in this case.

`token`: (required) Authentication token for the session.

Success Response:

The response indicates that the update process has started.

Response Format:

```
{
  "Request": "installUpdate",
  "Status": "ok",
  "Response": {
    "status": "updating"
  }
}
```

`status`: `"updating"` means the update process has started successfully.

Error Response:

```
{
  "Request": "installUpdate",
  "Status": "error",
  "StatusMessage": "Error message"
}
```

`StatusMessage`: A message describing the reason for the error (e.g., missing updater file, permission issues).

Notes:

This endpoint requires that the update has already been downloaded using `downloadUpdate`.

Requires a valid authentication token.

The update process may restart or stop the application.

setMqttConfig

Description:

The `setMqttConfig` API configures the MQTT broker connection settings for SvanLINK. This includes the broker host, port, authentication credentials, and enables or disables the MQTT service. After configuration, the application will restart to apply the changes.

Request Format:

Request Body:

```
{
  "Request": "setMqttConfig",
  "Params": {
    "enabled": true,                    // Optional: Enable or disable MQTT service
(boolean)
    "host": "mqtt.example.com",         // Optional: MQTT broker host address
    "port": 1883,                       // Optional: MQTT broker port number
    "username": "mqtt_user",            // Optional: MQTT broker username
    "password": "mqtt_password",        // Optional: MQTT broker password
    "ssl": true,                        // Optional: A boolean indicating if the MQTT
is using SSL/TLS (e.g., true).
    "caCert": "base64_encoded_cert",    // Optional: The CA certificate for the MQTT
(e.g., "base64_encoded_cert").
    "certFile": "base64_encoded_cert",  // Optional: The certificate file for the MQTT
(e.g., "base64_encoded_cert").
    "keyFile": "base64_encoded_key"     // Optional: The key file for the MQTT (e.g.,
"base64_encoded_key").
  },
  "token": "userToken"
}
```

`enabled` (optional): A boolean value to enable or disable the MQTT service. When set to `true`, the MQTT service will be active.

`host` (optional): The hostname or IP address of the MQTT broker.

`port` (optional): The port number for the MQTT broker connection (typically `1883` for non-SSL or `8883` for SSL).

`username` (optional): The username for authenticating with the MQTT broker.

`password` (optional): The password for authenticating with the MQTT broker. This is stored securely in a separate file.

`certificate` (optional): SSL certificate file encoded in Base64 format for secure MQTT connections (TLS/SSL).

`token` (required): Authentication token for the session.

Success Response:

If the configuration is successfully applied, the response will indicate success. Note that the application will restart after a successful configuration update.

Response Format:

```
{
  "Request": "setMqttConfig",
  "Status": "ok"
}
```

`Request`: Reflects back the original request type (`"setMqttConfig"`).

`Status`: The status of the request (`"ok"` for successful requests).

Error Response:

If the request fails, an error response is returned.

Response Format:

```
{
  "Request": "setMqttConfig",
  "Status": "error",
  "StatusMessage": "Error message describing what went wrong"
}
```

Notes:

This endpoint requires a valid authentication token and an active license.

All parameters in `Params` are optional. Only the parameters provided will be updated.

After a successful configuration update, the SvanLINK application will automatically restart to apply the changes.

The password is stored securely in a separate file and is not included in the main configuration.

If a certificate is provided, it will be used for secure TLS/SSL connections to the MQTT broker.

To disable the MQTT service, set `enabled` to `false`.

`getMqttConfig`

Description:

The `getMqttConfig` API retrieves the current MQTT broker configuration settings, including the broker host, port, username, and enabled status. Note that the password is not returned for security reasons.

Request Format:

Request Body:

```
{
  "Request": "getMqttConfig",
  "token": "userToken"
}
```

`Request`: The type of request, which is `"getMqttConfig"` in this case.

`token` (required): Authentication token for the session.

Success Response:

The response contains the current MQTT broker configuration.

Response Format:

```
{
  "Request": "getMqttConfig",
  "Status": "ok",
  "Response": {
    "enabled": true,
    "host": "mqtt.example.com",
    "port": 1883,
    "username": "mqtt_user",
    "ssl": true
  }
}
```

`Request`: Reflects back the original request type (`"getMqttConfig"`).

`Status`: The status of the request (`"ok"` for successful requests).

`Response`: An object containing the MQTT configuration:

- `enabled`: A boolean indicating whether the MQTT service is enabled.
- `host`: The MQTT broker hostname or IP address.

`port`: The MQTT broker port number.

`username`: The MQTT broker username.

Notes:

This endpoint requires a valid authentication token and an active license.

The password is not returned in the response for security reasons.

If a configuration value is not set, it may be `null` or omitted from the response.

addMqttTopic

Description:

The `addMqttTopic` API creates or updates an MQTT topic configuration. This allows you to configure which data requests should be automatically published to the MQTT broker at specified intervals. The configuration is similar to WebSocket channel configuration.

Request Format:

Request Body:

```
{
  "Request": "addMqttTopic",
  "Params": {
    "Topic": "main",  // Optional: Topic name. Default is "main"
    "Requests": [
      {
        "Request": "getResults",
        "Interval": 1000  // Interval in milliseconds (ms). Minimum interval is 100ms.
      },
      {
        "Request": "getSpectrumResults",
        "Interval": 5000  // Interval in milliseconds (ms). Minimum interval is 100ms.
      }
    ]
  },
  "token": "userToken"
}
```

`Params` (required): A configuration object containing:

`Topic` (optional): The name of the MQTT topic. Default is `"main"`.

`Requests` (required): An array of request objects that will be automatically published to the MQTT broker:

`Request`: The type of data request to publish. Available requests include:

- `getFileList` - Get list of files
- `getResults` - Get measurement results
- `getSpectrumResults` - Get spectrum results
- `getStatus` - Get device statuses
- `getVersion` - Get version information
- `licenseStatus` - Get license status
- `sendRawCommand` - Send raw command to device
- `startMeasurement` - Start measurement

`Interval`: The interval in milliseconds (ms) at which the request is published. Minimum interval is 100ms.

`token` (required): Authentication token for the session.

Success Response:

If the MQTT topic is successfully created or updated, the response will indicate success.

Response Format:

```
{
  "Request": "addMqttTopic",
  "Status": "ok"
}
```

`Request`: Reflects back the original request type (`"addMqttTopic"`).

`Status`: The status of the request (`"ok"` for successful requests).

Error Response:

If the request fails, an error response is returned.

Response Format:

```
{
  "Request": "addMqttTopic",
  "Status": "error",
  "StatusMessage": "missing config param" or "config must be in json"
}
```

Note: The error message "missing config param" refers to the `Params` object being missing or empty. The error message "config must be in json" indicates that `Params` must be a valid JSON object.

Notes:

This endpoint requires a valid authentication token. A license is not required.

If a topic with the same name already exists, it will be updated with the new configuration.

The configuration is saved persistently and will be restored when the application restarts.

Data will be automatically published to the MQTT broker according to the configured intervals.

Each published message includes a timestamp in the format `"YYYY-MM-DDTHH:mm:ss.SSS"`.

Use `getMqttTopicList` to see all configured MQTT topics.

`deleteMqttTopic`

Description:

The `deleteMqttTopic` API deletes a specific MQTT topic configuration. This will stop all automatic data publishing for that topic and remove it from the configuration.

Request Format:

Request Body:

```
{
  "Request": "deleteMqttTopic",
  "Params": {
    "topic": "main"  // Required: The name of the MQTT topic to delete
  },
  "token": "userToken"
}
```

`topic` (required): The name of the MQTT topic to delete.

`token` (required): Authentication token for the session.

Success Response:

If the topic is successfully deleted, the response will indicate success.

Response Format:

```
{
  "Request": "deleteMqttTopic",
  "Status": "ok",
  "StatusMessage": "The topic has been deleted successfully"
}
```

`Request`: Reflects back the original request type (`"deleteMqttTopic"`).

`Status`: The status of the request (`"ok"` for successful requests).

`Response`: An object containing a confirmation message.

Error Response:

If the request fails, an error response is returned.

Missing Parameter:

```
{
  "Request": "deleteMqttTopic",
  "Status": "error",
  "StatusMessage": "The \"topic\" parameter is missing"
}
```

Topic Doesn't Exist:

```
{
  "Request": "deleteMqttTopic",
  "Status": "error",
  "StatusMessage": "Error delete topic"
}
```

Notes:

This endpoint requires a valid authentication token. A license is not required.

Deleting a topic will:

Stop all periodic data publishing for that topic

Remove the topic from the MQTT configuration

Save the updated configuration to persistent storage

The deletion is permanent and cannot be undone. The topic configuration will need to be recreated if needed.

Use `getMqttTopicList` to see available topics before attempting to delete one.

`getMqttTopicList`

Description:

The `getMqttTopicList` API retrieves a list of all configured MQTT topics and their configurations. This includes information about which requests are being published and at what intervals.

Request Format:

Request Body:

```
{
  "Request": "getMqttTopicList",
  "token": "userToken"
}
```

`Request`: The type of request, which is `"getMqttTopicList"` in this case.

`token` (required): Authentication token for the session.

Success Response:

The response contains all configured MQTT topics and their associated request configurations.

Response Format:


```
{
  "Request": "getMqttTopicList",
  "Status": "ok",
  "Response": [
    {
      "Topic": "main",
      "Requests": [
        {
          "Request": "getResults",
          "Interval": 1000
        },
        {
          "Request": "getSpectrumResults",
          "Interval": 5000
        }
      ]
    },
    {
      "Topic": "deviceStatus",
      "Requests": [
        {
          "Request": "getStatus",
          "Interval": 15000
        }
      ]
    }
  ]
}
```


Response Fields:

`Request`: Reflects back the original request type (`"getMqttTopicList"`).

`Status`: The status of the request (`"ok"` for successful requests).

`Response`: An object containing all configured MQTT topics:

- `[topic_name]`: Each topic is represented by its name as a key:
  - `Requests`: An array of request objects configured for this topic:
    - `Request`: The type of request (e.g., `"getResults"`, `"getSpectrumResults"`).
    - `Interval`: The interval in milliseconds (ms) at which the request is published.

Notes:

This endpoint requires a valid authentication token. A license is not required.

The response includes all topics that have been configured, including the default `"main"` topic if it exists.

Each topic contains a list of requests that are automatically published at the specified intervals.

If no topics are configured, the response will be an empty object `{}`.

Use this request to discover available topics before subscribing to them or before deleting them with `deleteMqttTopic`.

`subscribeChannel`

Description:

The `subscribeChannel` API allows a WebSocket client to subscribe to a specific channel for receiving real-time data. This request can only be sent via WebSocket connection and enables the client to receive periodic updates from the specified channel.

Note: This request is WebSocket-only and cannot be sent via HTTP/TCP connections.

Request Format:

Request Body:

```
{
  "Request": "subscribeChannel",
  "Params": {
    "channel": "main"
  }
}
```

`Request`: The type of request, which is `"subscribeChannel"` in this case.

`Params`: An object containing the parameters for the request.

- `channel` (required): The name of the channel to subscribe to. If the specified channel doesn't exist, the client will be automatically subscribed to the `"main"` channel.

Success Response:

If the subscription is successful, the response will indicate that the client has been subscribed to the specified channel.

Response Format:

```
{
  "Request": "subscribeChannel",
  "Status": "ok",
  "Response": {
    "message": "Subscribed to channel successfully"
  }
}
```

`Request`: Reflects back the original request type (`"subscribeChannel"`).

`Status`: The status of the request (`"ok"` for successful requests).

`Response`: An object containing the response data.

`message`: A message indicating that the subscription was successful.

Error Responses:

No Channel Specified:

```
{
  "Request": "subscribeChannel",
  "Status": "error",
  "StatusMessage": "The "channel" parameter is missing"
}
```

Channel Doesn't Exist:

```
{
  "Request": "subscribeChannel",
  "Status": "error",
  "StatusMessage": "The channel doesn't exist. Subscribing to the 'main' channel."
}
```

Notes:

This request can only be sent via WebSocket connection (port `8001` by default).

Authentication is required before subscribing to channels. Send a login request or provide a valid token first.

If the specified channel doesn't exist, the client will be automatically subscribed to the `"main"` channel.

Subscribing to a new channel will automatically unsubscribe the client from all other channels.

Once subscribed, the client will receive periodic updates according to the channel's configuration.

Available channels can be retrieved using the `getChannelList` request.

`getChannelList`

Description:

The `getChannelList` API retrieves a list of all available WebSocket channels and their configurations. This request can only be sent via WebSocket connection and returns information about all configured channels, including their associated requests and intervals.

Note: This request is WebSocket-only and cannot be sent via HTTP/TCP connections.

Request Format:

Request Body:

```
{
  "Request": "getChannelList"
}
```

`Request`: The type of request, which is `"getChannelList"` in this case.

Success Response:

The response contains all configured WebSocket channels and their associated request configurations.

Response Format:

```
{
  "Request": "getChannelList",
  "Status": "ok",
  "Response": [
    {
      "Channel": "main",
      "Requests": [
        {
          "Request": "getResults",
          "Interval": 1000
        },
        {
          "Request": "getSpectrumResults",
          "Interval": 5000
        }
      ]
    },
    {
      "Channel": "deviceStatus",
      "Requests": [
        {
          "Request": "getStatus",
          "Interval": 15000
        }
      ]
    }
  ]
}
```

Response Fields:

`Request`: Reflects back the original request type (`"getChannelList"`).

`Status`: The status of the request (`"ok"` for successful requests).

`Response`: An object containing all configured channels.

- `[channel_name]`: Each channel is represented by its name as a key.
  - `Requests`: An array of request objects configured for this channel.
    - `Request`: The type of request (e.g., `"getResults"`, `"getSpectrumResults"`).
    - `Interval`: The interval in milliseconds (ms) at which the request is sent.

Notes:

This request can only be sent via WebSocket connection (port `8001` by default).

Authentication is required before retrieving the channel list. Send a login request or provide a valid token first.

The response includes all channels that have been configured, including the default `"main"` channel.

Each channel contains a list of requests that are automatically sent at the specified intervals.

If no channels are configured, the response will be an empty object `{}`.

Use this request to discover available channels before subscribing to them with subscribeChannel.

This configuration creates a new channel or updates an existing one with the specified requests and intervals.

`deleteChannel`

Description:

The `deleteChannel` API allows a WebSocket client to delete a specific channel and its associated configuration. This request can only be sent via WebSocket connection and permanently removes the channel from the system, including all its configured requests and intervals.

Note: This request is WebSocket-only and cannot be sent via HTTP/TCP connections.

Request Format:

Request Body:

```
{
  "Request": "deleteChannel",
  "Params": {
    "channel": "main"
  }
}
```

`Request`: The type of request, which is `"deleteChannel"` in this case.

`Params`: An object containing the parameters for the request.
- `channel` (required): The name of the channel to delete.

Success Response:

If the channel is successfully deleted, the response will indicate that the channel has been removed from the system.

Response Format:

```
{
  "Request": "deleteChannel",
  "Status": "ok",
  "StatusMessage": "The channel has been deleted successfully"
}
```

Error Responses:

No Channel Specified:

```
{
  "Request": "deleteChannel",
  "Status": "error",
  "StatusMessage": "The "channel" parameter is missing"
}
```

Channel Doesn't Exist:

```
{
  "Request": "deleteChannel",
  "Status": "error",
  "StatusMessage": "Error, the channel doesn't exist!"
}
```

Notes:

This request can only be sent via WebSocket connection (port `8001` by default).

Authentication is required before deleting channels. Send a login request or provide a valid token first.

Deleting a channel will:

- Remove the channel from the WebSocket configuration
- Stop all periodic requests associated with that channel
- Disconnect all clients subscribed to that channel
- Save the updated configuration to persistent storage

The deletion is permanent and cannot be undone. The channel configuration will need to be recreated if needed.

Use getChannelList to see available channels before attempting to delete one.

Clients that were subscribed to the deleted channel will need to subscribe to a different channel to continue receiving updates.