Contents

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`

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`

Authentication:

When the WebSocket connection is opened, the first step is to authenticate. This can be done by sending a login (#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.

```
{
  "Request": "wsConfig",
  "Params": [
    {
      "Request": "getResults",
      "Interval": 1000 // Interval in milliseconds (ms). Minimum interval is
100ms.
    },
    {
      "Request": "getSpectrumResults",
      "Interval": 5000 // Interval in milliseconds (ms). Minimum interval is
100ms.
    }
  ]
}
```

`Requests`: An array of request objects.

`Request`: The type of data request (e.g., "getResults", "getSpectrumResults").

`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:

```
{
  "Request": "wsConfig",
  "Status": "ok",
  "Response": {
    "message": "Configuration applied successfully"
  }
}
```

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

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"
```

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

`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": "XXXXXXXXXXXXXXX",
    "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. During it's activation the Svantek device must be connected to the hardware (PC, laptop, server, Raspberry Pi etc), otherwise an error will be returned.

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
        },
        "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.

Response Object:

`app`: 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`).

`ui`: An object containing UI configuration.

- `liveView`: An object containing live view configuration.
  - `resultTypes`: A semicolon-separated string of result types (e.g., `"LAF;LAeq;LCpeak"`).
  - `currentTab`: The current tab in the live view (e.g., `"liveResultsTab"`).
- `thresholdView`: An object containing threshold view configuration.
  - `thresholdOrange`: The orange threshold value (e.g., `45`).
  - `thresholdRed`: The red threshold value (e.g., `75`).
  - `k`: The k is the coefficient which is added to live (e.g., `0`).
  - `resultType`: The result type for the threshold view (e.g., `"LAF"`).

`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.

Request Format:

Request Body:

```
{
  "Request": "setConfig",
  "Params": {
    "app": {
      "tcpPort": 8000,
      "echo": true,
      "wsEnabled": false,
      "wsPort": 8001
    },
    "ui": {      // Optional: List of result types to fetch.
      "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`).

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

`liveView`: An object containing live view configuration.

`resultTypes`: A semicolon-separated string of result types (e.g., `"LAF;LAeq;LCpeak"`).

`currentTab`: The current tab in the live view (e.g., `"liveResultsTab"`).

`thresholdView`: An object containing threshold view configuration.

`thresholdOrange`: The orange threshold value (e.g., `45`).

`thresholdRed`: The red threshold value (e.g., `75`).

`k`: The k value (e.g., `0`).

`resultType`: The result type for the threshold view (e.g., `"LAF"`).

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.

downloadFiles

Description:

The `downloadFiles` 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.

`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": "1.0.6"
  }
}
```

`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.