Metropolia University of Applied Sciences

Internet of Things

Group Project

Web Interface for ABB Ventilation Controller
Technical Documentation

Authors:
Theresa Brankl
Maya Hornschuh
Janine Paschek
Simon Schädler

Submitted on: October 25, 2021

Contents

1 Introduction

VentPro is a web interface for controlling an ABB ventilation controller. The interface displays all available information about the connected IoT device and enables the user to control the ventilation system using a website.

The system consists of an IoT device, a server, and a web interface. The IoT device controls the speed of a connected fan and measures the current air pressure regularly. The device is connected to the server which provides the web interface allowing users to set a specific pressure or fan speed. Also, the interface displays current and former sensor data received from the IoT device to the user.

This technical documentation provides specific information about the implementation of both the front end and back end of the system. It does not include any descriptions of how to use the web interface itself. This information can be found in the user manual.

1.1 Installation

This project is based on Node which needs to be installed to run the server. Please visit nodejs.org and follow the instructions to install node (LTS or latest version). The following node packages are required to run the server. The packages can be installed by running *"npm install <package>"*.

- body-parser
- cookie-parser
- ejs
- express
- express-session
- sqlite3
- ws

The server also requires a running MQTT broker. To connect the server to a running broker, please open the */src/server.js* file and check the MQTT configuration section. Please enter the correct IP, port, and topics by adjusting the following parameters:

```
const mqtt_ip = "mqtt://localhost";
const mqtt_port = "1883";
const mqtt_topic_pub = "controller/settings";
const mqtt_topic_sub = "controller/status";
```

After installing the required packages and setting up the MQTT parameters, the server can be started by running *"node src/server.js"*. The server display status information in the terminal if everything started up correctly.

To sign in on the web interface, the *admin* account can be used by signing in with the default password "*admin*". The password can be changed at the setting page. The settings page also allows the admin account to add new users to the system.

2 Back end implementation

The server provides users with all necessary information for using the web interface. It serves requested web pages and data in general but also establishes the connection to an IoT device and stores different kinds of data consistently. The server's functionality is implemented in a central JavaScript file, the *server.js*.

2.1 Global database

The server uses an SQLite database which is set up as a single global database to store all kinds of data. The database (*/src/data/data.db*) contains the following five tables to store data:

```
CREATE TABLE users(username TEXT, hash TEXT, role TEXT);
CREATE TABLE log_users(timestamp INT, user TEXT);
CREATE TABLE pressure(timestamp INT, pressure INT);
CREATE TABLE fan_speed(timestamp INT, fan_speed INT);
CREATE TABLE target_values(id TEXT, value INT);
```

The *users* table contains the login information about all registered users while *log_users* keeps track of all login activities which is described separately in section 2.3. On the other hand, *pressure* and *fan_speed* are tables for data logging only. These tables store all sensor data received from the ventilation controller. The *target_values* table contains the current values of the target speed and target pressure requested by the user as well as the current mode of the system (see chapter 2.5 for more information).

2.2 Routing

The server uses Express to manage the routing. To prevent unauthenticated users from accessing data, every route refers to the *auth_user(req, res, next, redirect, arg_dyn = '')* function to authenticate users and to check their permissions (chapter 2.3.) The following example shows the routing for */control_panel*:

```
app.get('/control_panel', async (req, res, next) => {
   auth_user(req, res, next, 'control_panel');
});
```

In this example, the server will pass all necessary parameters for authenticating the user and returning a response as well as the information about which route was called. If the user is logged in, the server will render the */src/views/control_panel.ejs* file as a response and return it to the client.

2.3 Authentication and authorization

Authentication and authorization are both very important for this interface. Only logged-in users are allowed to open the page. Also, not every logged-in user has permission to use all available features. Therefore, on each request a client sends, the server will try to authenticate the user before providing any information. In this project, session based authentication is used to authenticate users as well as to log them in and out.

2.3.1 Front end authentication

When a user initially connects to the web interface, the server will automatically redirect to the logout page. There, the user can enter a username and a password which will then be sent to the server.

Every page of the interface provides a log-out button, which enables the user to manually log out. If a user wants to log out, the client will send a basic get request to log out from the server. The server will log the user out and redirect to the logout page.

2.3.2 User database

The *users* table of the central database is used to store the login information of all users (chapter 2.1). Next to the username and the hashed password, the table also contains a role parameter. The role indicates whether the user has admin privileges or not.

New entries to user database can be added by admins only. Aside from manually adding an entry by directly connecting to the database and adding data with SQL commands, the web interface also provides a form to add new users. This form also is available for admins only (chapter 3.4.2).

2.3.3 Routing

All routes that the server handles require the client to authenticate before serving any page or data. The only exception to this is the */logout* route, which deletes the current session and returns the *views/logout.ejs* file to the client without authentication. All other routes call the function *auth_user(req, res, next, redirect)* (chapter 2.3.4) and pass a string of the requested redirect as a parameter. The function will check the authentication parameters provided by the client. If the provided information is valid, the function will call the requested function and return information to the client.

2.3.4 Back end user authentication

Session-based authentication uses session cookies to authenticate users. To log in, the client sends the username and password to the server using a POST request. Then, the *login_user* method generates a hash based on the provided username and password and compares it with the hashes stored in the database. If the username and password match the information in the database, the user is authenticated successfully. The server will then create a new session with a unique session

ID. This ID will be returned to the client as a session cookie and will be appended to all following requests to the server.

When a user has signed in, the server needs to keep track of this activity. After connecting to the database, a new entry will be added to the *log_users* table (chapter 2.1). This allows the server to provide a list of all login activities of all users which can then get displayed on the settings page of the web interface (chapter 3.4).

Every time another request is received from any client, the server needs to check if the client is authenticated before executing any method or returning any data. Therefore, all routes call the method *auth_user(req, res, next, redirect, arg_dyn = '')* and pass all necessary connection parameters as well as information about the requested redirect. This function checks, if the user is authenticated and logged in by comparing the passed session ID with the currently active sessions. If no session ID or an invalid one has been received, the server will redirect to the logout page and ask the user to sign in. If the session ID is valid, the server reads the users' permission from the database as this is relevant for some functionalities. After authenticating the user, the function returns the requested page or calls a method to perform specific actions depending on the *redirect* parameter. This parameter can either contain a direct path of a file that should be returned, but can also indicate a number of functions that need to be called.

If the user wants to sign out, the client sends a get request to the */req_logout* route. After receiving this request, the server will delete the current session and redirect the user to the logout page. Deleting the session will automatically invalidate all incoming requests with the former session ID. The user is logged out.

2.3.5 User authorization

Users are allowed to use most of the features provided by the web interface. Still, some actions can get performed by authorized users only. For example, only admins are allowed to add a new user to the system or to see all users' login activity. As described in chapter 2.3.2, the *users* database has an attribute that tells the role of each user. There are two roles, the *default* and the *admin* role. If the client requests a service that is available to admins only, or that returns different results depending on the users' role, the *auth_user()* function will pass the current users' role as an argument to the function, that performs the requested actions. Then, the server decides if the user is authorized or not. In general, the server returns one of the following status codes on requests that require specific privileges.

200	'OK'	The requested action was successfully executed
403	'Forbidden'	The user has no permission to execute the requested action
409	'Conflict'	The action could not be executed because of conflicting arguments

The server sends those status codes alongside the resulting data or message. Then, the client deals with received data, takes the user to a different page, or displays an alert depending on the result.

2.4 Settings functionality

The settings page of the web interface displays the users' login activity and provides a form for changing the password. If the user has admin privileges, the login activity of all users will be displayed. Also, admins can add a new user to the system by entering a new username and password in the provided form.

The server needs to check permissions and send only the data the client is allowed to see. It must also tell the client to enable and disable the form for adding a new user depending on the users' permissions. Also, the server must always check permissions before adding a new user to the system even if the form is disabled because attackers could active the form manually or send custom requests to the server telling it to create new users.

2.4.1 Changing the password

Each users' password is saved as a hashed value in the database. If a user sends a request to change the password, the server first needs to generate a new hash using a PBKDF2 function. The server uses the username as the salt parameter and generates a 64-bit hash by iterating 10.000 times:

```
crypto.pbkdf2(password, username, 100000, 64, 'sha512', (err, key) => {
    if (err) throw err;

    let hash = key.toString('hex');
    change_password(req, res, next, username, hash);
});
```

After a new hash has been generated, the server connects to the database and updates the current users' entry in the *users* table by replacing the old hash with the new one. It is important, to catch errors while generating the hash and updating the database. The user might be unable to log back into the interface if the system suggests that a new password has been set successfully but hasn't To prevent this, the server will respond with an internal error code to the client. This will trigger an alert telling the user that something went wrong. If the password was changed successfully, the server responds with a status 200 (OK). The client will display an alert to confirm to the user, that his password has been updated.

2.4.2 Adding a new user

Adding a new user works quite similarly to changing a password, but this is allowed to users with admin privileges only. The server checks the users' permission (chapter 2.3.5) and generates a new hash just as described in chapter 2.4.1 with the difference that the hash is being generated based on the passes parameters (username, password) instead of the current users' credentials. After generating the has, the server adds a new entry to the *users* table of the database with the new username and hash. The server then returns a response with a status code to tell the client if the requested action could be performed successfully.

2.4.3 Login activity

If the client sends a request to fetch all login activities, the server first checks whether the user has admin privileges or not. If the user is an admin, the server simply reads and returns all entries from the *log_users* table of the database. If the user is not authorized to see all users' login activity, the server selects only the current users' entries from the database:

```
db.each(`SELECT * FROM log_users WHERE user = "` + user + `"`, (err, row) => {
    if (err) {
      console.error(err.message);
    }

    let time = new Date(row.timestamp)
    let time_formatted = time.toLocaleString();
    log.push({"timestamp":time_formatted + ': ',"user":row.user});
});
```

Each entry read gets added to a JSON array with its' timestamp being formatted to a readable format. After all the entries have been read, the JSON array will be returned to the client as the response.

2.5 Data transfer

The system requires different kinds of communication. First, the IoT device needs to send sensor data to the server which then logs and saves this data. But the data must also be sent to the client on request. Also, the client needs to be able to request a new target pressure or fan speed by sending a request to the server. The server then has to pass this request to the IoT device.

While some of the messages are being sent frequently, some are not. To deal with those requirements, three kinds of communication are used in this system: MQTT, WebSocket, and basic HTTP requests.

2.5.1 Communication between the oT device and the server

MQTT is a messaging protocol designed for IoT applications and uses a separate MQTT broker to pass messages between clients. MQTT uses a publish/subscribe system to transport data.

The IoT device connects to the broker, publishes messages on a topic to send sensor data to the server, and subscribes to a different topic to receive commands sent by the user. The server on the other hand subscribes to the sensor data topic and publishes commands on the second topic.

By default, the system uses the topic *controller/status* to pass sensor data. Once subscribed to the topic, the serve initializes an event handler on all incoming messages. Messages on that topic are being sent in the following format:

```
{"nr":1, "speed":31, "setpoint":10, "pressure":10, "auto":true, "error":false}
```

Once a new message is being received, the event handler triggers and stores the received data in the database. To enable the server to serve a set of sensor data for pressure or fan speed only in a fast way, those sets of data are being stored in separate tables (chapter 2.1).

2.5.2 Communcation between the server and the client

While HTTP requests are a great way to send a single set of data as a response to a single request, this is not very convenient for sending data on a frequent basis.

WebSocket is a communication protocol providing full-duplex communication between a server and one or several clients. Using WebSocket, the client can establish a communication channel to the server. Data can be sent in both directions at all times and by setting up event listeners, both the server and the client can also react to received messages at all times.

If the client needs a set of sensor data to display on the control panel, it fetches the data using an HTTP request and receives a single set of data in return. But for showing live data (section2.5.2) or displaying warnings (chapter 2.5.2), the system uses WebSocket communication.

Data minimization

Every time a user loads the control panel page of the web interface, the client fetches a set of data to plot for both pressure and fan speed. By default, the control panel plots an all-time chart of the data. As the IoT device publishes data every few seconds, the server has a huge set of data stored in its' database. Returning a set of data with many thousands of entries not only results in an unpleasantly long loading time but also has an enormously bad impact on the performance of the control panel itself. Life data would be displayed in a laggy way, the control panel would react very slowly to any user input, and resizing the browsers' window could even trigger freezes.

To get rid of this problem, the server reduces traffic and improves the interface performance by shortening the set of data each time before returning it to the client. This does not affect the data stored in the database but only the data being displayed on the control panel.

The sensor data is being read and returned as a JSON array. By testing several array sizes it turned out that arrays with up to 1.000 entries each do not affect the loading time or performance in a critical way but still allow the plots to display data in a smooth way. Therefore, each set of data read from the database is being shortened before being returned to the client using the following function.

```
function shorten_array(arr) {
    let res = arr;
    let comp = arr;
    while(res.length > 1000) {
        res = [];
        let idx = 0;
        for(let i = 0; i < comp.length; i++) {
            res[idx] = comp[i];
            idx++;
            i++;
        }
        comp = res;
    }
    return res;
}
```

This function checks the length of the initial array read from the database. If the array contains more than 1.000 entries, each second element is being dropped and the remaining elements form the new array. This procedure gets repeated as long as the resulting array still has more than 1.000 elements. In the end, the resulting array has between 500 and 1.000 elements. This array will then get returned to the client and the data contained will be displayed on the control panel afterward.

Live data

While the user is using the control panel, all sensor data that is currently being received should automatically be added to the plots. The user should be able to see live data all the time. To do so, data gets send using the WebSocket connection. Each time a user establishes a WebSocket connection to the server, a new MQTT subscriber is being set up to listen to all incoming messages containing sensor data. Every time an MQTT message is being received, the pressure, setpoint, fan speed, and error parameters will immediately get passed to the client via WebSocket. This ensures that the server provides the most current sensor data to the client all the time without sending unnecessary requests or data in between.

Warings

If the client sets the target pressure to a value that could not be reached in a reasonable time, the IoT device will indicate that by setting the error bit in the data messages sent via MQTT. If this happens, the system needs to tell the user by displaying a warning on the web interface.

While the user is using the control panel, the error indicator is automatically being passed to the client alongside the sensor data (chapter 2.5.2) but this data is not being sent if the user is using another interface page. To do so, the client establishes a WebSocket communication to the server on the other interface pages as well indicating that only errors and no sensor data should be sent. The server then again sets up an MQTT subscriber on all incoming messages but the event listener will only pass the error indicator to the client to reduce traffic.

Another problem is, that the IoT device keeps sending errors until the target pressure is reached. This could either happen after some more time or after the user has set a new, reachable target pressure. As the server should not send multiple error messages to the client that would trigger an alert each time a new set of sensor data is being received, some kind of filter is essential.

To do so, the server sets a bit indicating if the error bit was set in the last message received. This allows the server to recognize new errors. If a new error is being received, the error gets passed to the client. Otherwise, the error bit of the WebSocket message will be set to *false* even if the error is still active. The client will then only display an alert once per each new error.

Sending commands

Next to receiving data, the client also needs to be able to send commands to the IoT device to set a new target pressure or fan speed. As this command is only being sent once in a while, the client does

not send this command via the established WebSocket connection but using a basic HTTP POST request. The server then passes this command to the IoT device by publishing a message on the *controller/settings* topic in one of the following formats:

```
{"auto": true, "pressure": 30}
{"auto": false, "speed": 60}
```

If the client requested a specific target pressure, the *auto* bit will be set to *true* to indicate that the system needs to run in automatic mode. If a target fan speed is being requested, the manual mode is active which will be indicated by setting the *auto* bit to *false*. After sending the command, the IoT device should immediately react by adjusting the current fan speed and mode depending on the new parameters.

Storing identifiers

Depending on the current state of the system, different values for mode, target pressure, and target fan speed are currently set. To automatically display the correct values on the control panel, the server needs to keep track of those parameters. Each time a user switches the mode or sets a new target value, the server stores this value into a variable. While loading the control panel, the client can then fetch this data and display the current state instantly.

The current state must be persistent and independent of reloading the web interface, connecting from different user accounts, or even server restarts. To retain the current state even after a server restart, this data needs to be stored in the *target_values* table of the database (chapter 2.1). The table contains a set of data indicating the current mode, target pressure, and target fan speed in the following format:

```
{"id": "pressure", "value": 30}
{"id": "fan_speed", "value": 60}
{"id": "mode", "value": 0}
```

On startup, the server will initially load those entries from the database to return this data on clients' requests.

3 Front end implementation

As the back end implementations' main task is to provide functionality, the front end implementation also needs to consider the design and usability of the interface. It is important to have a clear and consistent design on all pages. Also, the menu and every other way of providing user inputs need to be intuitive and consistent in terms of the design.

To keep a consistent design, the interface uses two basic colors - a dark gray (#262626) and a light blue (#90fbff) as a contrast color. Those two colors are used on all pages.

To make the interface as intuitive as possible, the menu, amount of pages, and amount of possible interactions are kept on a very basic level. The goal was to provide everything needed in a most simple and usable way to the user. For example: Even if the control panel could be split up into multiple subpages displaying the pressure and fan speed separately or having a single page for setting new target values, all those functionalities are integrated on one page on purpose.

In general, all positioning has been done using flexboxes which also makes it easy to make the interface responsive. The design as well as some animations are done using custom CSS.

3.1 Responsive header and navigation menu

The navigation menu is a central element of the web interface that is consistent on all pages and allows the user to navigate between the different subpages. The header needs to directly indicate what kind of interface the web page is and what options the user has. To do so, the header displays an animated fan next to the interfaces' title VentPro in the top left corner of the page. On the right side of the header, the navigation menu displays all available sub-pages with an intuitive icon next to each text.

As the whole interface needs to be responsive and can be used both on desktop computers and mobile devices, this also applies to the header. Depending on the current window size of the browser, the header adjusts its' width automatically. As the width is being reduced, the interface will first hide the texts of the navigation menu to reduce the required space. The text is being hidden instead of the icons because icons require way less space and are very intuitive as well. If the width is being reduced even more, the "VentPro" text will disappear as well. The rotating fan will always be displayed. This responsiveness is realized using the CSS *media screen*. The limiting values have been deduced by testing multiple values. The elements are now being hidden perfectly right before clashing into another element.

Apart from making the header responsive, it is also important to indicate which page is currently loaded in a nice way. To do so, the current pages' text and item in the navigation menu will be highlighted by displaying both in the blue contrast color while all other elements are colored white.

3.2 Help page

It is indispensable to provide simple instructions on how to use the web interface on the interface itself. A user should not need to reach out to the user manual but should be able to find all necessary explanations on an integrated help page. The help page provides information about both the control panel and the settings page and gives answers to typical questions. Displaying questions with their answers makes it way easier for the user to find a solution for a problem than just giving a large set of instructions.

At the top of the page, the user can directly navigate between the different sections or open the official user manual. To increase usability, the answers are not being displayed instantly. Instead, answers can be displayed by clicking any of the listed questions. Using an accordion-like CSS transition, answers will blend in smoothly right underneath the selected question.

3.3 Control panel

The control panel allows the user to switch between automatic and manual mode. In automatic mode, the user can set a target pressure between 0Pa and 120Pa. The ventilation system will automatically adjust the current fan speed depending on the requested target pressure. In manual mode on the other hand, the target fan speed can be set to a value between 0% and 100%. The ventilation will set the current fan speed to the requested value no matter the pressure.

Depending on the active mode, the input elements will be activated/deactivated (chapter 3.3.1). The system is either in automatic or manual mode and the user can set the target values of the current mode only. The mode can be changed using the switch displayed at the top of the control panel (chapter 3.3.2). After loading the control panel page, the client will fetch the current mode from the server and activate the correct control panel depending on the response.

Besides the input elements to set target values, the control panel displays a plot of current sensor data both for current pressure and fan speed (chapter 3.3.3). As well as three buttons to select data of different time intervals.

3.3.1 Activating and inactivating the panels

Depending on the current mode, only one of the two panels is active. The other one should still display the current sensor data, but the user should not be able to change any settings for that mode while it is inactive. Apart from disabling user inputs, the design of the inactive panel should change to clearly indicate to the user that it can not be used at the moment.

To disable all user inputs, the flexbox contains an overlay with the same dimensions and a transparent body. Setting the z-index of the overlay to a higher value will display it above the flexbox disabling the user to click any of the elements below. To activate the panel, the overlay is being hidden which makes the elements underneath accessible again.

To indicate that a panel is inactive, some filters are being applied to the panels' flexbox:

```
document.getElementById('control_manual').style.opacity =  0.5;
document.getElementById('control_manual').style.webkitFilter = "blur(1px)";
```

Reducing the opacity will fade out the panel to a percentage of 50%. Adding a blur effect then gives the final touch to make the panel look like it can not be used at the moment. Changing the values of the effects applied could increase the effect, but would also make the plot illegible. The chosen fulfill both requirements nicely. To reactivate a panel, the client just resets the opacity to 100% and clears the blur effect.

By default, both panels are disabled. Once the page is loaded, the client will fetch the current mode from the server. Depending on the response, one of the panels will get activated. This is essential because enabling the panels by default would allow the user to send invalid requests to the server while the client is fetching the current mode.

3.3.2 Switching the mode

The mode is stored on the server to prevent clients from sending requests of different modes at the same time. After loading the page, the client will initially fetch the current mode from the server, set the switch to the correct position and enable the active panel.

If the user switches modes, the client will send a post request to the server to change the global mode. If the server responds with a status 200 (OK), the client switches its' local mode and activates the other control panel. Otherwise, the switch will not be set and the other panel will remain inactive.

As the mode should not only be changed on the web interface and the server, but also at the ventilation system itself, the client also needs to send a command to the IoT device. If the mode got changed, the client will send a POST request with the current target value for either the pressure or fan speed to the server. The server will then pass the command to the IoT device via MQTT (chapter 2.5.2).

3.3.3 Plotting the data

To display sensor data on the control panel, the client generates two plots, one for pressure and one for fan speed data. The free open-source library Chart.js can be used to plot data in a lot of ways. In this case, basic lined plots will show the pressure and fan speed over time. The parameters of the plots can easily be adjusted to fit the given range of 0-120Pa for displaying the pressure and 0-100% for displaying the fan speed. The x-axis will automatically be labeled depending on the data that gets assigned to it. To display the timestamps in a readable format, the values get formatted manually before assigning them to the plot. Depending on the selected time interval, the plot will either show a date or a day time for each data point.

Apart from setting the correct range and labeling the axes, some design adjustments were made as well. To stick with the consistent design of the interface, the color of the plots has been set to the general blue highlight color. To improve readability, a slightly darker version of the same base color has been chosen which is still noticeable on disabled panels. To keep the nice and clean design even if huge sets of data are loaded, the circle radius of the single data points has been reduced. Also,

the tension parameter of the plots has been set to 0.4. This causes the client to calculate a smooth interpolation between the single data points. Interpolation makes it easier to see the reaction of the ventilation system and looks more realistic than linear interpolation.

Fetching data of the selected interval

After loading the control panel or after the user selected a different time interval (chapter 3.3.3) the client needs to fetch a set of data that should be displayed in the plot. To do so, an HTTP GET request is being sent to the server. The request includes information about the time interval as well as the requested kind of data. By requesting the sensor data for the pressure and fan speed separately, the traffic is being reduced and the overall performance is being improved. The data is being returned as a JSON array. After receiving the data, the client parses the data points and their associated timestamps to the plot by overwriting the former dataset.

Receiving and displaying live data

While datasets for the plots are being fetched after loading the control panel or selecting a different time interval, live data needs to be added to the plot frequently. As described in chapter 2.5.2, live data is being passed to the client via WebSocket messages as soon as they get received.

The client reacts to every incoming message and checks if valid sensor data has been received. Then, the data is being extracted and added to the plots dataset. After refreshing the plot using the Chart.js function *plot.update()*, the new data point will get added to the plot. This procedure is being done for both plots as every message contains a current value for both pressure and fan speed.

Also, after receiving a new message, the current values will be displayed as a number above the plots. Appending the current value to the title of the plot is a convenient way to display the current state. This makes it easier for the user to read the current values than hovering over the latest data point in the plot.

Selecting different time intervals

Each panel provides three buttons to choose between different intervals of time depending on what data should be displayed. Clicking on a button will trigger an event listener. The event listener will request a set of data from the server and convert the timestamps into the correct format.

Depending on the selected time interval, the x-axis of the plots will either show a date (*DD.MM.YYYY*) or a day time (*HH:MM:SS*) for each data point. The dataset returned by the server contains timestamps in milliseconds since 01.01.1970 00:00:00 UTC. If the selected time interval is "*all*", the timestamps will be displayed as dates. If "*today*" or "*last minute*" has been selected as the interval, the time of the measurement will be displayed instead. Both conversions are done using custom date prototypes that can be applied to a given date object.

After a new time interval has been selected, an indicator is being set to keep track of the current settings. This allows the client, to correctly format timestamps of the incoming live data before adding it to the plot.

3.3.4 Setting the target pressure and target fan speed

One of the main features of the web interface is setting a target pressure or a target fan speed for the ventilation controller. The target values can be set by adjusting one of the sliders displayed underneath the plots on the control panel. Sliders provide a smoother and more intuitive way of setting a value than typing a number into a basic text input field and were therefore chosen for this purpose. The possible range of the target pressure is from 0Pa to 120Pa while the fan speed can be set between 0% and 100%.

Fetching the current state of the system

The current target values are always being displayed right above the slider. This indicates the current state of the system, but also makes it easier to set the pressure or fan speed to a specific value. Of course, this value should always be up to date. To ensure that, the client will initially fetch the current target values from the server after loading the page. After receiving the response, the labels will be overwritten with the current values and both sliders will be set to that value as well.

Sending a new target pressure and fan speed

As soon as the user moves the slider to a different position, two things need to happen. The label above needs to update its' value to tell the user what exact value has been set. Also, the client needs to send a command to the IoT device to pass the new settings.

To handle user inputs on a slider, event handlers are used. Every time a user readjusts the slider, the updated target value should be sent to the server. But also, while the user is still moving the slider, the label above should already update its' text. This allows the user to set an exact target value without needing to readjust the slider a couple of times.

While the "*change*" event of a slider triggers, when a new position has been finally set, the "*input*" event already triggers every time the slider has reached a new position, even if it has not been released yet. To prevent the client to send a huge amount of unnecessary and unintentionally selected values, the "*change*" event is not being used for sending commands to the server. Otherwise, for example, moving the target pressure from 10Pa to 30Pa would trigger the client 20 times and 20 messages with new requests would be sent to the server and the ventilation controller. To prevent this from happening, the "*input*" event is used for sending commands. Sending the command itself is done by sending an HTTP POST request to the server. The body of the request indicates whether the pressure or fan speed has been set and passes the requested value as well. The server then handles this request and passes the command to the IoT device as described in chapter 2.5.2.

On the other hand, the label above the slider should behave in that exact way. Therefore, a second event listener has been added to trigger on "*change*" events of the sliders. After triggering, it will simply update the label to indicate the exact current position of the slider.

3.4 Settings page

The settings page is available for all logged-in users through the */settings* route. The settings page displays the same elements for each user. But depending on the users' permission, the content and allowed actions might differ. The settings page displays the login history depending on the user as well as two forms for changing the password and adding a new user to the system.

3.4.1 Changing the current users' password

The client displays a text input and a button to enable the user to set a new password. The placeholder property will ask the user to enter a password. Both, the input and the button are embedded into a form that has an event handler for "submit" events. Therefore, the user can set a new password by pressing the "OK" button or the "Enter" key. The input field has the required parameter set to prevent the user from sending a post request with empty parameters.

When the server receives the request to change the password from an authenticated user, it does not need to check permissions. All users are authorized to change their passwords. The server just generates a hash based on the username and the new password and updates the users' entry in the database. The server then will return a status message with a status code.

When the client receives the result from the server, it will clear the input and display an alert to inform the user whether the password was successfully changed or not. If the password was changed, the user needs to log in with the new credentials. For user experience purposes, the client will automatically log out the user and redirect to the logout page.

3.4.2 Adding a new user to the system

Adding a new user to the system works similarly to changing the password. The client displays two input boxes and asks the user to enter a username and a password. Both inputs are required and the password input is of the type "password". That will hide the entered password and just display dots instead. After the user submitted the input, the client will send a post request to the server to add the new user to the system.

After receiving the request, the server needs to check the users' permission before adding a new user to the system. Only admin accounts are authorized to add new users. If the user does not have the required permission, the server will return a 403 'Forbidden' error to the client. Otherwise, the server will first check if the requested user already exists in the database. I the user already exists, a 409 'Conflict' error will be returned. If the client requested to add a user that does not already have an entry in the database, the server will first generate a hash based on username and password and then insert a new user to the database. The role of all users added to the system using the web interface is 'default'. The timestamp is initially set to 0 so the users' first login will automatically be handled as a new login (chapter 2.3.4). After successfully adding a new user, the server will return a 200 'OK' status to the client.

When the client receives the result from the server, it will clear the input fields and display an alert depending on the received status code.

3.4.3 Displaying the login activity

The client will automatically fetch a list of all login events the server has logged to the database. If the user has admin privileges, the server will return all login events, otherwise just those of the current user. The data gets returned as a JSON array. After receiving the data, the client checks the number of received events and divides it into a dynamic amount of pages with eight events each. This is done for user experience purposes only. The server then generates a flexbox for each of the eight events and adds it to the login history. It also adds two buttons to switch between the pages of data. If the user clicks one of the navigation buttons, the client will reset the displayed list and add the next pages' data by accessing the received data at another index. The buttons are only active if there are more pages available to load. If a page contains less than eight events, the client will add space holder items instead to keep the design consistent.

When the server receives a request for the login history it first checks the permission of the user. If the user has admin privileges, the server will select all data from the *log_users* table of the main database and return it to the user. The *log_users* table is part of the main database and logs all new logins in the following format:

```
CREATE TABLE log_users(timestamp INT, user TEXT);
```

If the user does not have admin privileges, the server will select all entries, where the username equals the user that send the request and return those to the client.

3.5 Displaying warnings

As described in chapter 2.5.2, the ventilation controller sets an error indicator in its' messages if the requested pressure could not be reached in a reasonable time. The server passes the information about this error alongside the sensor data while the user is using the control panel, but in a separate message while another page is currently loaded. Therefore, each page of the web interface establishes a WebSocket connection to the server to receive warnings without manually fetching this information regularly to reduce traffic.

As soon as the client receives a message with the error indicator being true, an alert will be displayed immediately. The alert will tell the user that the requested pressure could not be reached. Additionally, the alert will display the current pressure to let the user know how much the difference between the requested and actual pressure is at the moment:

```
alert('[WARNING]\r\nThe target pressure was set to ' + msg.setpoint +
    'Pa but could not be reached in a reasonable time!\r\nCurrent pressure: ' +
    msg.pressure + 'Pa');
```

It is important, that the user gets the information about an error immediately. Even if the user has set a new target pressure on the control panel and then switched to the help or settings page, the alert still needs to be displayed. Therefore, this alert function is active on all the interface pages except the logout page.

Although, with this implementation, the client would display the same warning each time the user reloads a page while the error is being active. This happens because reloading the page will establish

a new WebSocket connection. As described in chapter 2.5.2, the server will add a new error handler for each connection and therefore trigger an alert each time.

3.6 Login/logout page

The login/logout page is the only page of the web interface that does not show the navigation menu as those pages can only be accessed by logged in users. The only thing the login page provides is a form to enter a username and a password. After submitting, the client will send a post request to the server. If the entered credentials are valid, the server will return a session cookie (chapter 2.3.4) and redirect to the control panel. Otherwise, the server will rerender the login page. Rendering the login page as an ejs file allows the server to pass parameters. In this case, the server will pass an *error* text which will be displayed underneath the form if the entered username or password are not correct.