



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 21, 2021

Contents

1	Introduction	3
1.1	Installation	3
2	Back end implementation	4
2.1	Global database	4
2.2	Routing	4
2.3	Authentication and authorization	5
2.3.1	Front end authentication	5
2.3.2	User database	5
2.3.3	Routing	5
2.3.4	User authentication	5
2.3.5	User authorization	6
2.4	Settings functionality	7
2.4.1	Changing the password	7
2.4.2	Adding a new user	7
2.4.3	Login activity	8
2.5	Data transfer	8
2.5.1	MQTT	8
2.5.2	WebSocket	8
3	Front end implementation	9
3.1	Responsive header and navigation menu	9
3.2	Help page	9
3.3	Control panel	9
3.3.1	Activating and inactivating the panels	10
3.3.2	Switching the mode	10
3.3.3	Plotting the data	10
3.3.4	Setting the target pressure and target fan speed	10
3.4	Settings page	10
3.4.1	Changing the current users' password	10
3.4.2	Adding a new user to the system	10
3.4.3	Displaying the login activity	11
3.5	Displaying warnings	11
3.6	Logging a user out	11

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.

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](#).

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.

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>"*.

- express
- ejs
- body-parser
- 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.

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, timestamp INT, 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, basic HTTP authentication is used to log in/log out users.

2.3.1 Front end authentication

When a user initially connects to the web interface, the server will return a 401 error code and ask for authentication. The browser will automatically show a form and ask the user to enter a username and a password. If the entered credentials are correct, the server will redirect to the landing page.

Every page of the interface provides a log-out button, which enables the user to manually log out. If a user logs out, the client will first send a basic get request to log out from the server. Then, the client sends an invalid authentication request and redirects to the logout page. Sending a separate logout request before the invalid authentication enables the server to tell apart the logout request from any other invalid authentication like logging in with incorrect credentials. This is important to keep track of the users that are currently logged in and can be used to log all login activities.

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 contains a timestamp of the users' last login (in milliseconds since 01.01.1970 00:00:00 UTC) as well as a role. The timestamp can be used to distinguish new logins and navigation between different subpages, but also for logging all login activities on the server (chapter 2.3.4). The role indicates whether the user has admin privileges or not.

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, that 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 User authentication

If a route is called by a client, it will call the *auth_user(req, res, next, redirect)* function with information about the requested service. The function reads authorization parameters from the request body and checks if valid data was received. If so, the function 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. If a client was successfully authenticated, the function checks if the request was a new login or just an authenticated request for a page or service. To do so, the current time is being compared to the users' timestamp in the database. Three cases can be detected that way:

- If the timestamp equals zero, it either is still zero from its first initialization or has been reset during a logout procedure. Therefore, the client is performing a new login. Then, the login will get logged and the timestamp in the users' database entry will get updated.
- If the difference between the current time and the timestamp of the users' last login is greater than 30 minutes, the request will be interpreted as a new login. Then, the login request will get logged and the timestamp in the users' database entry will get updated.
- If the difference between the current time and the timestamp of the users' last login is less than 30 minutes, the request will be interpreted as a request for changing the page or fetching data. The timestamp will not get updated and the request will not get logged in the database.

After detecting a new login by checking the conditions explained above, the server will add a new row to the *log_users* table (chapter 2.1) with a current timestamp and the current username.

After deciding, if the database needs to get updated because of a new login, the function switches depending on the passed *redirect* parameter. If a page is requested, the parameter will directly include the filename of the page that should be rendered. Otherwise, the corresponding function will get called to perform actions and return the requested data to the client.

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

2.5.1 MQTT

2.5.2 WebSocket

Data minimization

Live data

Warings

Sending commands

Storing identifiers

3 Front end implementation

3.1 Responsive header and navigation menu

3.2 Help page

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 while the target fan speed can be set in manual mode only. Depending on the active mode, the input elements will be activated/deactivated. Also, if a mode is inactive, its' panel will be displayed with reduced opacity and a blur filter. These effects will be removed on activation. The mode is stored on the server to prevent clients from sending requests of different modes at the same time. The whole system is either in automatic mode or in manual mode. 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. 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.

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 ??). The panel also displays three buttons to display data of different time intervals. Clicking on a button will trigger an event listener. The event listener will request a set of data from the server, change some parameters of the plot depending on how the data should be displayed, and setting up different ways of handling new incoming data.

3.3.1 Activating and inactivating the panels

3.3.2 Switching the mode

3.3.3 Plotting the data

Fetching data of the selected interval

Receiving and displaying live data

Selecting different time intervals

3.3.4 Setting the target pressure and target fan speed

Fetching the current state of the system

Sending a new target pressure and fan speed

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. If 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

3.6 Logging a user out