Notification platform using air quality sensors and IoT

P2 - A larger program developed by a group

Daniel Møller, Kristian Johansen, Martin Sinkbæk Thomsen, Rasmus Secher

Computer Science
Aalborg University
Denmark
From February To May 2020

Title: ***Notification platform using air quality sensors and IoT***

Theme: ***IoT Event Manager***
Project period: ***February - May***

AALBORG UNIVERSITET
STUDENTERRAPPORT

Første Studieår
Datalogi
Strandvejen 12-14
9000 Aalborg
http://tnb.aau.dk

Project group: ***C1-3***
Participants:

- Daniel Møller
- Kristian Johansen
- Martin Sinkbæk Thomsen
- Rasmus Secher

Supervisor: ***Ramoni Adeogun***

Page Count: 76 Pages
Appendix: Page A to A
Finished: May 26, 2020

Abstract

Bad indoor air quality is an every-day issue and has a serious impact on a humans attention span. People who work in indoor environments, or are students who spend a lot of their day indoor, are to an extent affected by this issue and would benefit greatly by getting the means to battle this. As we, the writers of this project, are students ourselves and believe that some of the locations used to teach us have sub optimal air quality we find it highly motivational to pursue the task of discovering what makes bad indoor environment and furthermore how to combat this. We have therefore come up with a platform on which the actual air quality can be monitored and warnings can be issued. We have made this platform using a website as the foundation of the front end using HTML and JavaScript. The back end which we have made includes a server capable of communicating with a database storing the data from different sensor units capable of monitoring live air quality data. We have succeeded in this task and have ended up with a working platform on which air quality may be monitored, provided air quality sensors have been set up beforehand, for live data.

Preface

This report was written by a group of students from Aalborg University studying Computer Science, written between February 2020 and May the same year. The reason for this choice of topic is that the group often find themselves in study environments which have a tendency to develop bad indoor air quality. Mainly rooms such as auditoriums and group rooms. This report consists of 6 main chapters, going from the introduction to the ending section. Some problems came up during the progress of this report, mainly the outbreak of COVID-19, drastically changed the work methods which were used. The development progress of the report was fairly linear, with first the problem analysis to determine the scope of the problem, followed by modelling, programming and implementation, and lastly experimentation and conclusion.

Glossary

- **IAQ** : Indoor Air Quality
- **RH** : Relative Humidity
- **TVOCs** : Total Volatile Organic Compounds
- **IoT** : Internet of things
- **API** : Application programming interface
- **ppm** : Parts per million
- **HTML** : HyperText Markup Language
- **CSS** : Cascading Style Sheets
- **JS** : Javascript
- **SQL** : Structured Query Language
- **HTTP** : HyperText Transfer Protocol
- **JSON** : JavaScript Object Notation)
- **YAML** : YAML Ain't Markup Language
- **MySQL** : Standard name
- **MSSQL** : Mircrosoft SQL
- **AC** : Aircondition

Table of contents

1 Introduction

IoT, also known as Internet of Things, is all about connecting multiple devices to the internet. Not only computers or smartphones but for example a coffee machine, a fridge or in the case of this report, air quality monitoring sensors. This project is all about handling the events generated by these IoT devices (air quality sensors) in an intelligent and streamlined manner. It would be a great power to be able to predict when a rooms air quality may turn to the worse, and it is this power that we seek to deploy onto a program.

The issue of predicting when a room may become uncomfortable to be in, solely based on factors such as temperature and CO_2, is a subject which has been considered more and more often in recent times[2, p. 228]. The reason for this is that more and more studies have shown that bad IAQ reduces productivity, and increases other discomforts such as headaches and sweating[13, p. 1675]. It would not help to only notify when a room has bad air quality, since that information on its own cannot change the actual air quality. Therefore it is not only interesting to predict when a rooms air quality becomes poor, but also to give some sort of advice or warning beforehand, so that occupants can take precautionary action. These predictions can take use of statistical models and algorithms to help give a better idea on which actions should be taken.

Such actions could be to open a window or to turn on the air conditioning in the room. Therefore, the issue at hand is that it is difficult to know when the air quality has become bad, due to the fact that the only symptoms humans get are discomfort, and at that point it is already too late to take preemptive measures, and precious work time may have gone to waste. An increase in CO_2 has a negative effect on the learning ability of humans[13, p. 1675], as just a CO_2 concentration of over 1000 ppm (parts per million) has a significant impact on ones learning ability. This issue can be reduced or removed entirely by properly informing occupants in a room in time, and thereby helping their learning ability[8]. This is a project that will make use of IoT subjects, since it is very useful to be able to place sensors anywhere with an internet connection. This will therefore be a project about making an IoT event manager that is capable of giving preemptive warning on poor air quality in a room.

2 Problem Analysis

2.1 What is bad air quality?

There are several factors in play when it comes to determining indoor air quality (IAQ), such as ventilation rates, total volatile organic compounds (TVOCs), relative humidity (RH), CO_2 and temperature. These parameters all have an impact on the quality of the air[2, p. 229].

According to a literature review[2], ventilation rates over 8 l/s-p (litres per second per person) and CO_2 concentrations below 1000 ppm has shown an improvement in health and comfort for occupants in the room in question and can improve the learning process. Furthermore, lower ventilation rates showed increased nasal patency, asthmatic symptoms and risk of viral infections. When combined with increased CO_2 concentrations, it also showed impaired attention span, concentration loss and tiredness. Moreover, increased CO_2 concentrations also resulted in increased concentration of TVOCs.[2]

Indoor moulds also showed a strong relation to IAQ and were directly related to temperature and RH, in a way so higher temperature and higher rates of RH showed an increased concentration in indoor moulds[2]. By looking further into the previously mentioned literature review[2], thresholds of the parameters indicating bad air quality in an indoor environment can be established.

To sum this up, five parameters have been looked in to; ventilation rates, TVOCs, RH, CO_2 concentrations and temperature. Measuring ventilation rates is difficult without access to the ventilation system in question and as such ventilation rates will not be covered in this project. CO_2 concentration, temperature and RH can, however, be measured with a sensor that can be put almost anywhere, and will therefore be considered in this project. TVOCs can also be measured, however it requires specialised equipment that is fairly expensive[3][5][4] to work with and can be difficult to get correct measurements on, therefore TVOCs will not be considered in this project.

The following thresholds will be used in this report[2, p. 252]:
CO_2: 1000 PPM
Temperature: 25 Degrees
RH: 45%

These thresholds derive from the conclusion of the literature review[2, p. 252], where it is concluded that CO_2 concentrations below 1000 PPM can improve health and comfort of the occupants and lowering the temperature below 25 degrees can improve health and productivity of the occupants. An article[7, p. 225] on air humidity perceives around 43% RH to be a normal level, therefore the threshold for RH is set slightly higher at 45%. If one of the parameters exceeds its threshold, the IAQ can be seen as being bad.

2.2 Stakeholders

The effect on humans in bad IAQ is universal, and it does not matter what institution or workplace it is regarding, whether it is a school[2] or an office building[10]. This being said, this project will focus on schools and universities since it is a center of learning, and where bad IAQ has the biggest impact[2, p. 228]. The reason for the greater impact is that children and young people are affected to a greater extend than adults[2, p. 228], and have therefore lead to more strict building requirements regarding schools. This however does not mean that it is always followed correctly, and even a cost saving action that on an administrations level seems fine, can have a serious impact on the students.

When it comes to indoor air quality on schools and universities, there are several stakeholders included. In the front line, there are the students and teachers, who are getting the direct effect of bad air quality. A humans focus and work performance is significantly decreased by high levels of CO_2 and other air pollutants by as much as 95% effective work in basic activity, down to 50% efficiency[6, p.4]. From the same report, this means that an average CO_2 level of over 1000 ppm has a significant impact on how well the students perform. Since the same effects would happen to the teachers, it means that their performance also decrease, which in turn could result in a worsened learning experience by the students. There have also been conducted studies about the amount of complaints that are connected to ventilation types[11], and they have shown that buildings with sub optimal air ventilation have an increased amount of complaints regarding air quality. A room that can be used as an example, is an auditorium at a university. Here there are usually a lot of students in a single room, and it results in a large amount of CO_2 and heat being generated. If improper ventilation is installed, this could lead to a bad learning experience for the students.

Administration is also a part of this problem, not only if their own offices have bad indoor air quality, but also due to the fact that their job is to make sure that students learn. It has been found that CO_2 concentrations higher than 1000 ppm results in an increase in absenteeism by 10-20%[2], this brings the issue to the administration, since this is something that they are supposed to prevent and bring to a minimum.

Another stakeholder to consider is the buildings management staff, whose job is to make sure that all the buildings system works as intended. That include the ventilation systems installed, if any. It may be of use for this staff to have information regarding what to do with rooms that have bad IAQ and with that information, they can make their job easier.

It can thereby be concluded that the issue is present at all levels of a school, however it is those who are in the front line of the problem, eg. students and teachers, who feels the problem the most. Therefor the focus of this report will be to improve the situation for students and teachers.

2.3 Existing systems

Measuring bad indoor environment is a topic that several other studies have conducted. Three reports will be taken as examples in this section[1][12][9]. These systems have in common to measure indoor air quality and give some information to a user. The way that these systems achieve this, is by having one or multiple small devices capable of monitoring air quality in a facility, that transmits data to a server. The general idea can be seen, with inspiration from the source[12, fig. 1].

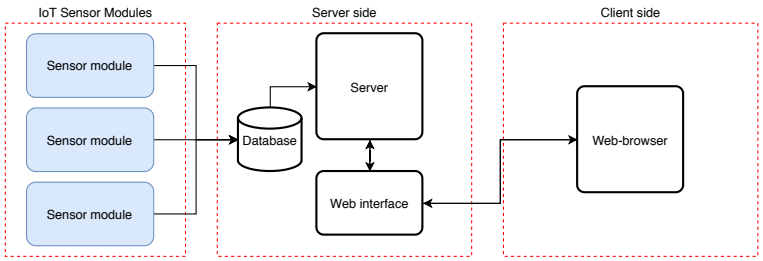


Figure 1: *Figure with inspiration from [12, fig. 1], showing three sections, sensor modules, server side and a client side*

The example systems[1][12][9] all use the general structure as can be seen in figure 1, since it is easy to implement and maintain. It is easy to implement because the sensor modules are a fairly standalone unit, which may require connection to the mains power supply, while others rely on battery. Then the only other thing that is missing is a server, to take care of the data that is being transmitted from the sensors. The sensor units themselves consists of a sensor and some kind of micro controller with access to the internet. An example of such micro controller could be an ESP2688[14]. This controller would be augmented with an array of sensors, such as CO_2, RH and temperature.

The web page can be made so that a user can directly see that data, that the sensor units are measuring. This interface is in all three examples written in HTML and CSS with some JavaScript serving as a driving factor for more advanced features. Most of these systems consist of a lot of sensors, that all send data to a server. The reason to have multiple sensors spread out, is that you can get a much more precise overview of the entire facility's air quality, than you would be able to from just a single sensor. An example of how spread out the sensors can be, can be seen in the following image, with inspirations from sources:[12, fig. 5].

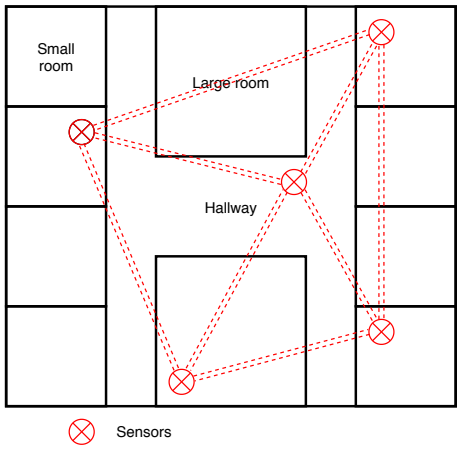


Figure 2: *Figure with inspiration from [12, fig. 5], showing an example of sensor unit dispersion*

Here it can be seen, that these units have been spread out to give a better overview on how the IAQ is. What can be done different from these other systems, is to actually put it to use and make a notification system, to inform the occupants of a room about a developing poor IAQ. An improvement to these systems can be to make them predict IAQ, based on historic data. This system must be able to predict when bad IAQ is going to happen, and give some preemptive actions to the user.

2.4 Thesis statement

The problem of bad IAQ has been analysed, and it has been concluded that the problem is indeed real, and that it has large implications. Parameters such as CO_2, RH and temperature have all been determined to be the most significant factors for bad IAQ. The stakeholders, students and teachers, have also been determined, since they are in the front line of the problem. Following this is the solution section, where a set of requirements for the solution is set up, as well as documentation on it. A thesis statement can now be made, to connect the problem analysis and the solution section:

How can a web-application be developed to monitor Indoor Air Quality (IAQ) parameters and generate preemptive alerts to appropriate stakeholders on recommended actions when an abnormality in IAQ level is detected?

2.5 Requirements

We now present the requirements that a solution to the problem of bad indoor air quality could have. In order for the solution to function as intended, several requirements have been set for the solution. A website would be a fitting platform, since it would not require any other programs installed other than a browser, and access to the internet. It is therefore easy to access the platform, and can be opened up on the fly. To complement this website, a server is needed to host the website and act as the interface where the client can get sensor data, prediction data and warnings from the database. The server will be made in Node.js, since it is easy to work with. The server will get its data from an SQL database, this will make it easy to store a vast amount of data and the data storage easily expandable and flexible. Since all data is gathered through the server by an interface, no direct client-to-database interaction is possible, hence enhancing security. The languages that will be used are HTML, CSS and JS for the client, since they are easy to work with and makes the platform easy to develop in a short time frame. The system must be able to employ a multitude of sensors, to increase the data set and thereby make more accurate predictions. The prediction system must be able to take historical data from the database and make accurate predictions on how the IAQ will progress. These predictions will be used to give the users a useful warning on when the IAQ becomes bad and some possible solutions on what to do about it. Lastly the admin page is to give an admin an easy and simple way to edit sensor info, room info and sensor thresholds, so that one does not have to do it manually to the database. A summation of these requirements and a short description can be seen in Table 1.

Requirements	Description
Website	The program consists of a website, which is the platform users will utilise
Node.js Server	The program will run on a server, which keeps track of all backend data
SQL Database	The server will also have a database, containing all sensor data
Specific languages	The website should be written in HTML/CSS/JS and the server in Node.js
Multiple sensors	The program should take input from multiple sensors to evaluate different
Prediction system	The program should be able to make predictions on a rooms air quality
Warning system	The program should warn the user if the IAQ becomes too bad
Admin Page	The program should have a managed interface to edit room/sensor/threshold data

Table 1: Requirements for solution

2.6 Feasibility

The feasibility of the previous requirements can be accessed to determine if they are plausible to uphold. This section will base the feasibility on how other projects were made[1][12][9]. These projects are described in section 2.3.

The creation of a website and a server is not something new and the aforementioned three projects use websites. The use of a database is also not something new, since it is one of the most widely used methods of data storage. The languages HTML, CSS and JS is also widely used, and was also used in those three projects[12, p.63]. The aforementioned three systems employed multiple sensors, meaning it should also possible to include multiple sensors in this project. The creation of a prediction system is however something the three other systems did not make. It is however not impossible to make, since one can make the use of prediction algorithms to accurately predict such. If such predictions can be made, then it is a clear and straightforward task to implement a warning system, that displays it to the user, as well as some possible solution to the issue. The aforementioned systems were developed by 4-5 people per system, which is similar to the manpower of our project which is 4 people. Therefore we believe that the scope of the system will be similar although the exact time frame for the development of their respective systems are not publicly known.

3 Model

The model section is describing how the program should be made, using flowcharts and text. All the following flowcharts, as well as flowcharts later on, use the following legend:

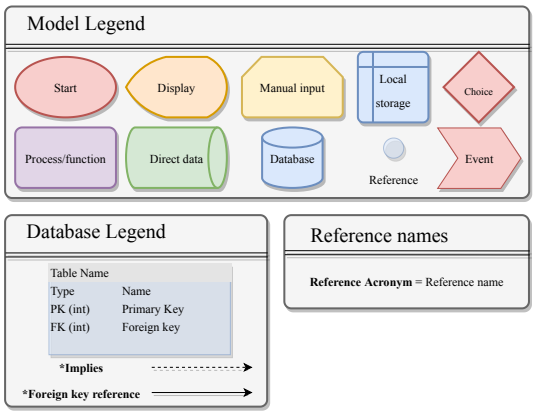


Figure 3: *Legend of the items used in the following models*

A general overview can also be made to show how the website works. Showing that the admin page, user page, server and database is separated.

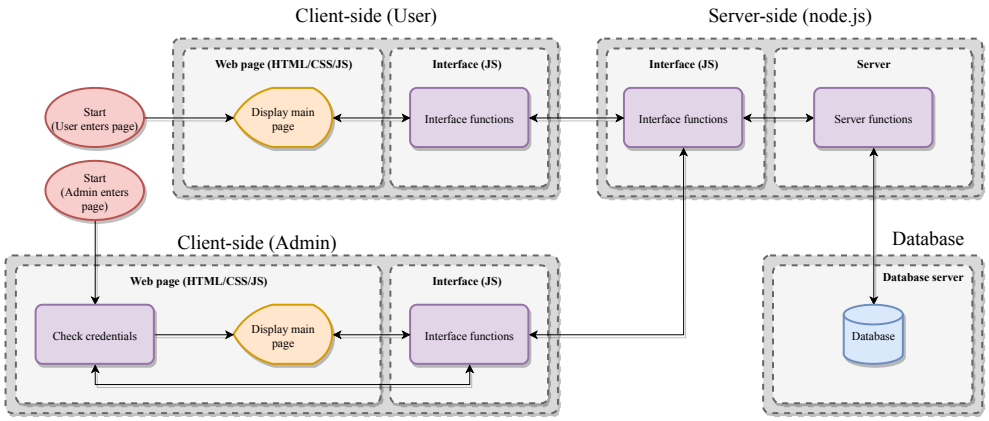


Figure 4: *Overview of the following models, showing the client sides, both user and admin, the server side and the database*

3.1 User side

The solution is split into two parts, the first part is for the users and the second part is for admins. The user section will be looked at first

3.1.1 User Client-side

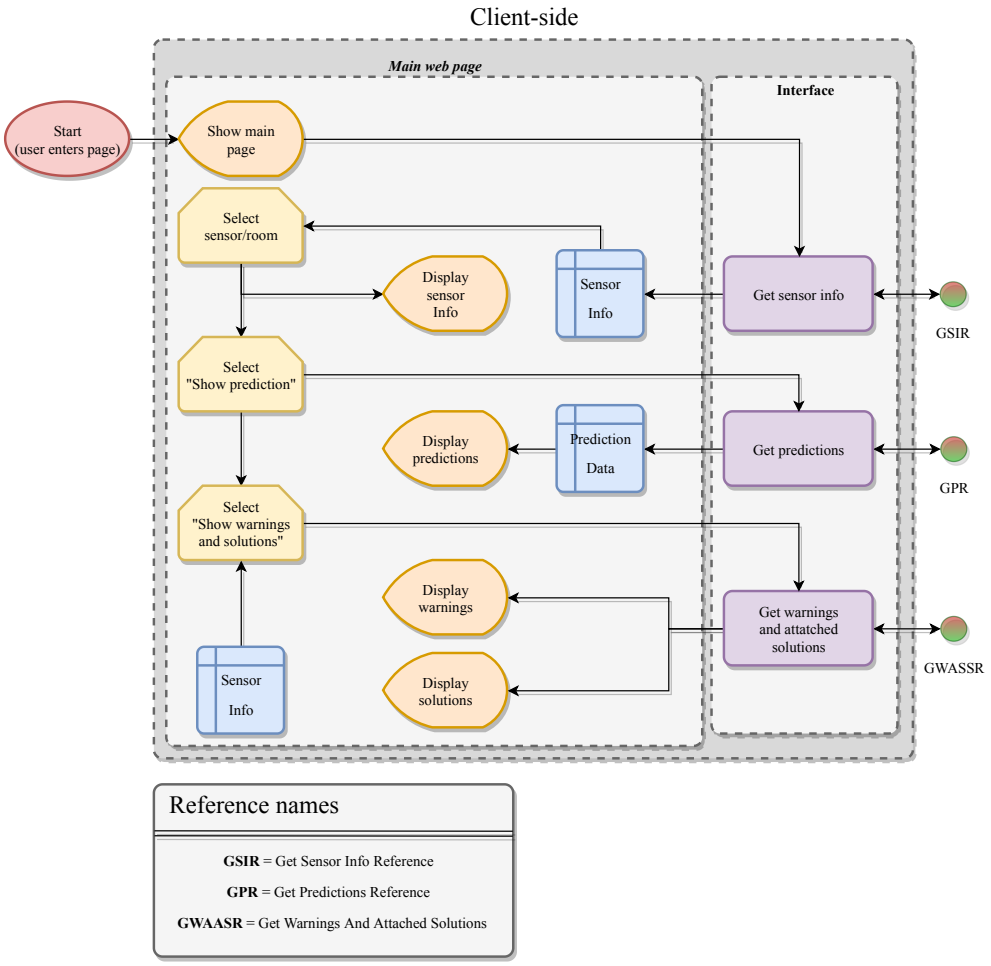


Figure 5: *Model of client-side, showing the main page functions and interface*

The model for the user client-side, as can be seen in figure 5, consist of a single web page and an interface to communicate with the server. When a user enters the web page, it calls the server to get general sensor information.

This information consists of rooms, which sensors are in that room, sensor types, etc..

This data is then used to populate a drop down menu, that contains all the different rooms which the system has. The user can then select a room, where the general data on that room is displayed, and an option is given to get prediction data on when the IAQ gets bad. This again calls the server to get data, and returns it to local storage and shows it to the user. The method for showing the data can be in a graph, or just a number until bad IAQ.

Lastly the user gets an option to get warnings and solution based on the prediction data. If the user selects it, the web page will again send a request to the server for data and display it.

3.1.2 User Server-side

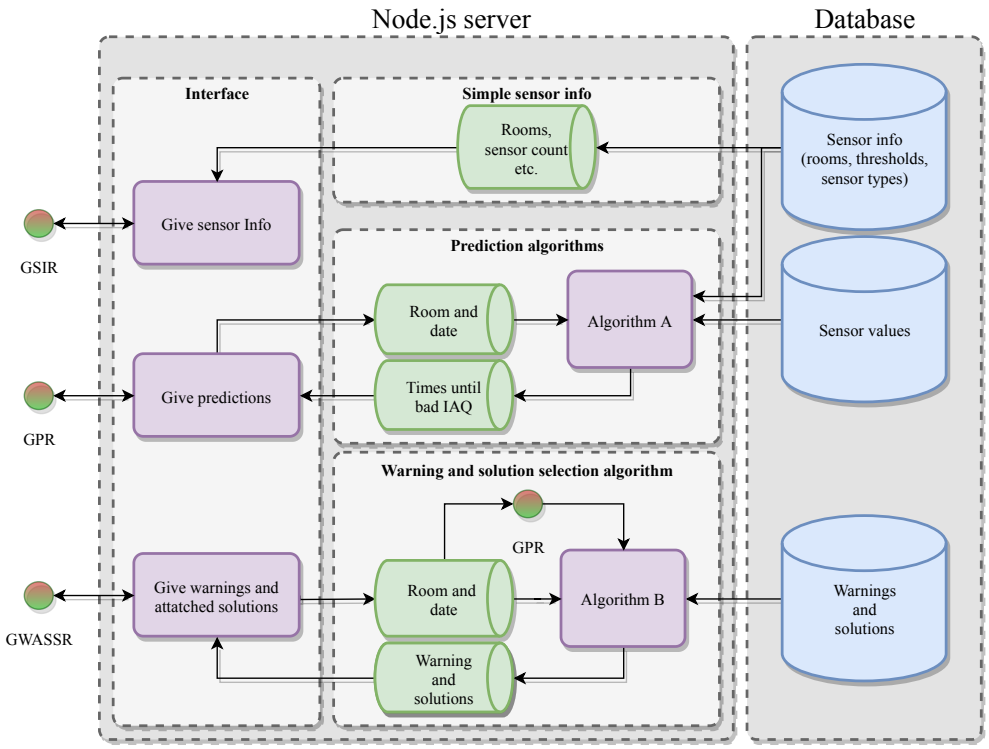


Figure 6: *Model of server-side, showing an interface, that communicates with three sub parts of the server and lastly a database*

The model for the user server-side code of the website can be seen in figure 6, which is depicted as two large boxes serving as; Node.js server and Database. The database will be explained later in this section. Within the Node.js server there is depicted several smaller boxes; "Interface", "Simple sensor info", "Prediction algorithm" and "Warning and solution selection algorithm". Furthermore the Node.js draws info from the Database through various queries. "Simple sensor info" gets data attached to sensors from the database through a query, such as which room a sensor belongs to, how many sensors there are, what type a sensor is etc. This data is then sent to the interface.

"Prediction algorithm" receives a specific room and data from the inter-

face and sends it into Algorithm A, which contains one or more algorithms designed to predict when the air quality of the given room will turn bad. This algorithm will output an array of the times of day that the IAQ have been bad historically. A more detailed description comes after this section.

"Warning and solution selection algorithm" receives some sensor info and a time until the IAQ turns bad, calculated in the Prediction algorithms box, which it sends into Algorithm B containing an algorithm to determine which warning and solutions it should display and recommend. This data is then sent to the interface.

The interface is where the server receives requests from the client-side and responds with info or data related to the request. The server gives three different types of responses; Give sensor info, Give prediction and Give warnings and attached solutions. In Give sensor info the server receives a request for sensor info and the server pulls that data from the Simple sensor info box and then sends the data to the client through a response. The procedure is the same for the two other responses, only the request and the data the server responds with is different; in Give predictions the server responds with the estimated time until the IAQ turns bad and in Give warnings and attached solutions the server responds with a warning and possible solutions.

Server-side Algorithm A

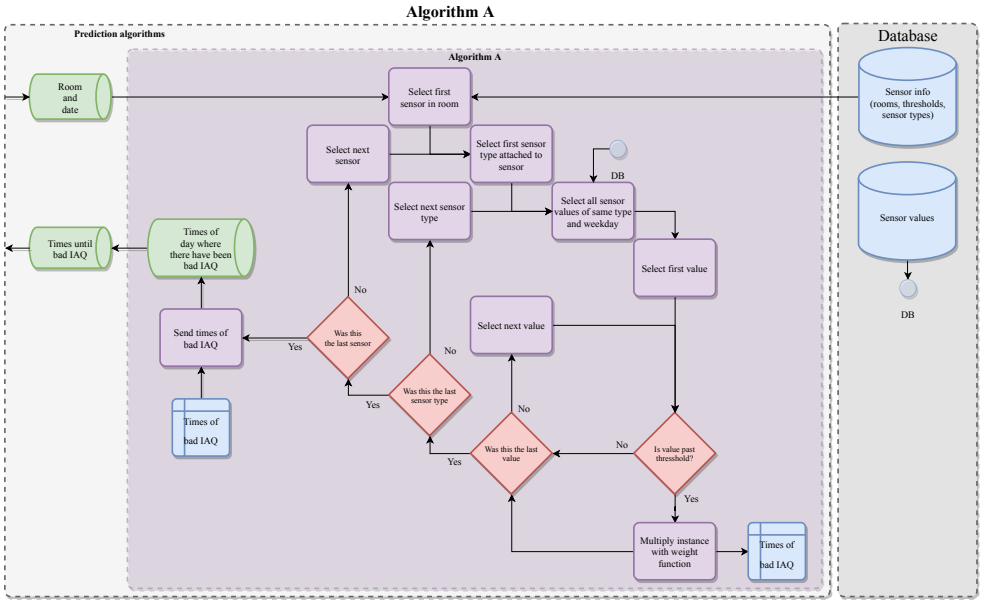


Figure 7: *Model of Algorithm A, showing the input consisting of date and room id, and returns an array of predicted chance of bad IAQ*

The Algorithm A is used to get prediction data based on what room is given, and the time of day. The algorithm will run through all the sensors in a room, and select the data entries that match the weekday given. E.g. if the date "Friday" is given, then the algorithm will choose all the previous entries that happened on a Friday. The different sensor types are then evaluated, to see if they have passed their preset thresholds, and if in the case that they have, the time and value is put into a local storage, to be send later. This continues for all the sensors in a room. In the end, all the data on bad IAQ is send out of the algorithm.

The way the data is returned is the time of the day split up in intervals, this could be 5 min intervals. These intervals are then populated with the times that there have been bad IAQ in the past, so that if there are bad IAQ two times in the same interval, the count of bad IAQ is added up. A example of such a graph can be seen in figure 8:

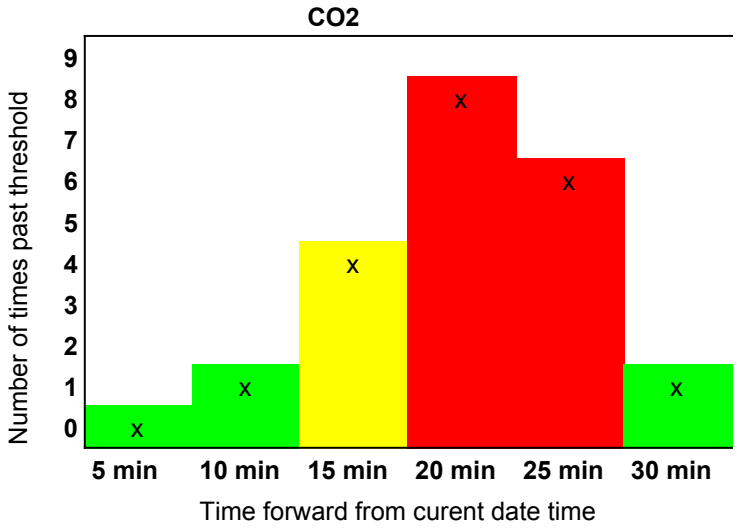


Figure 8: *Return style of Algorithm A, represented as an x-y coordinate system*

The y-axis values will then be weighted, so that if there are two instances, for example, of the threshold being passed:

- 1. The first is 1 day old
- 2. The second is 8 days old

Then the two would be weighted, by a function, depending on how old the entry is. As an example the one that is one day old could end up having a weighted value of 0.9 and the second one a value of 0.4, together they give a total 1.3. This is then the value that is represented in the y axis. This value can then now be divided by how many weeks the algorithm should look back. This could be that it looks back 3 weeks. This will result in an percentage output of $1.3/3 = 0.43 = 43\%$ chance of the value passing its threshold again within the next x minutes.

Server-side Algorithm B

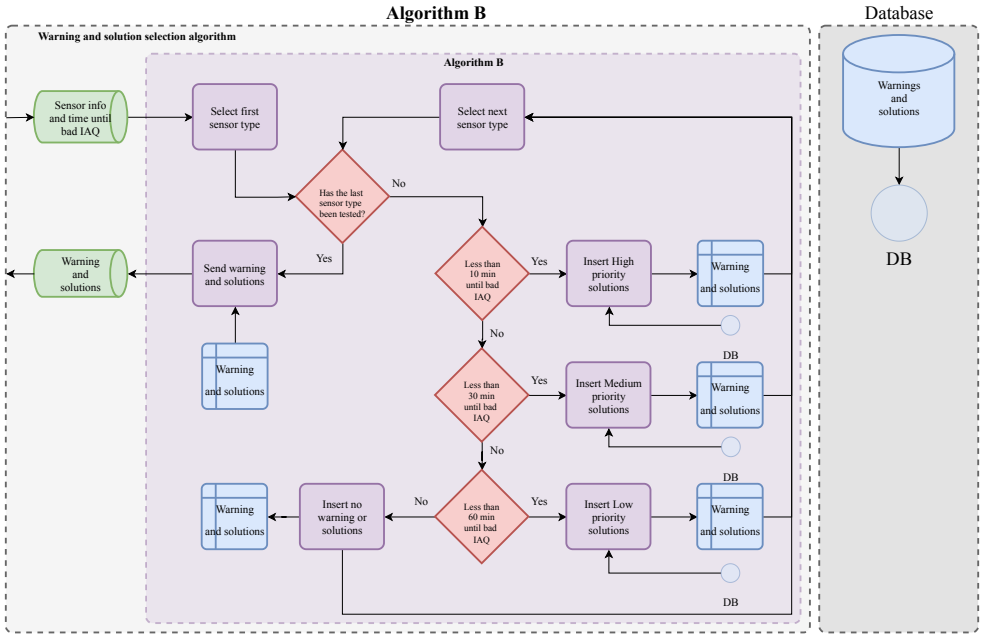


Figure 9: *Model of Algorithm B, showing the input of predictions and returns an array of potential warnings*

The model for the server-side Algorithm B shows the last of the algorithm where warnings are put into action. The algorithm is fed with information such as sensor info and time until bad IAQ. From here it flows through the model starting with the module of selecting the first sensor type and then checking whether or not it is the last sensor available.

Each sensor goes through a set of if statements to check whether the IAQ will exceed their thresholds within the next 10 min, 30 min or 60 min. These intervals correspond to one of three tiers of solutions being either "high priority", "medium priority" or "low priority". If one of these if-statements comes out as true a sufficient warning is then found from the "Warning and solutions" database which corresponds to the priority given and the given sensor info.

For example there could a situation where a prediction has been conducted and informs that there is less than 10 minutes until the room enters

a state of bad IAQ. This will then trigger the "high priority" tier and will then find sufficient warnings and solutions to combat this. An example of such a solution could be to take a break from work and leave windows and doors in an open position to encourage airflow.

In the case that a sensor type does not trigger a true statement from neither of the three if-statements the next sensor type will be chosen and the cycle continues. If a warning is activated it will be transferred to local storage of the server through the "warning and solutions". The information is then sent back to the "warning and solution" module on the server. Multiple warnings can also be sent, e.g. if both CO_2 and temperature are too high.

3.2 Admin side

The admin side of the solution also consists of a client and server part. These models are quite large, since they consists of a lot of get and set functions.

3.2.1 Admin Client-side

The admin client side consist of several smaller sections. These are sensor edit page and warnings and solutions edit page, as shown in figure 10.

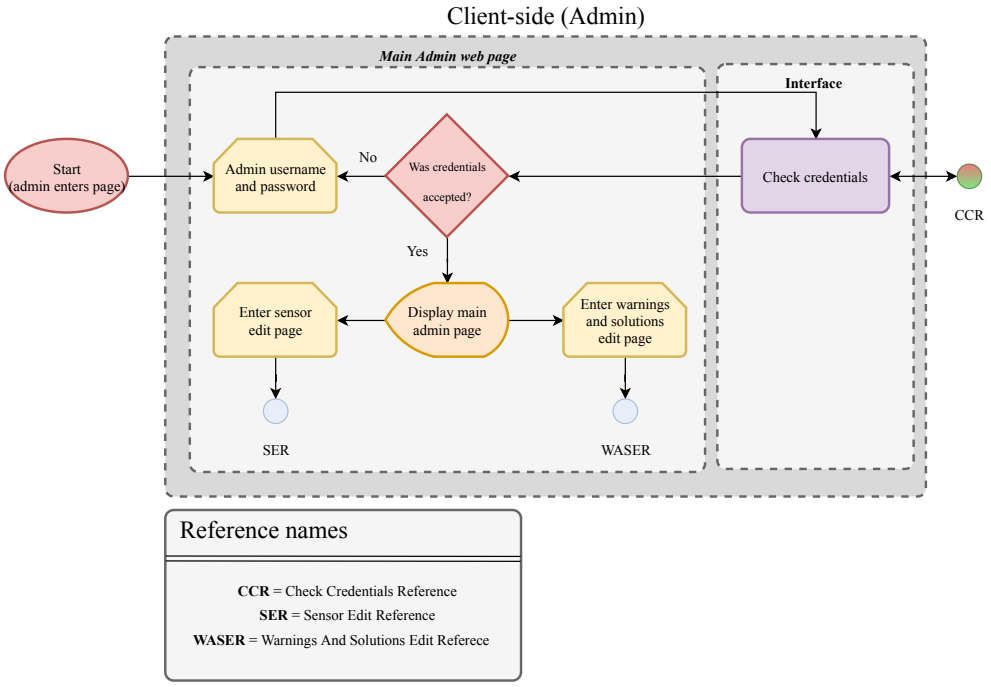


Figure 10: *Model of the admin client-side login page, where credentials are checked and the user gets redirected further if they are correct.*

As the first step of the admin page the user is prompted to input a username and a password, where it is then checked in the server to see if it is correct. If it is, the admin is presented with two options, to go to the Sensor Edit Reference page (SER) or the Warnings And Solutions Edit Reference Page (WASER).

SER Part 1

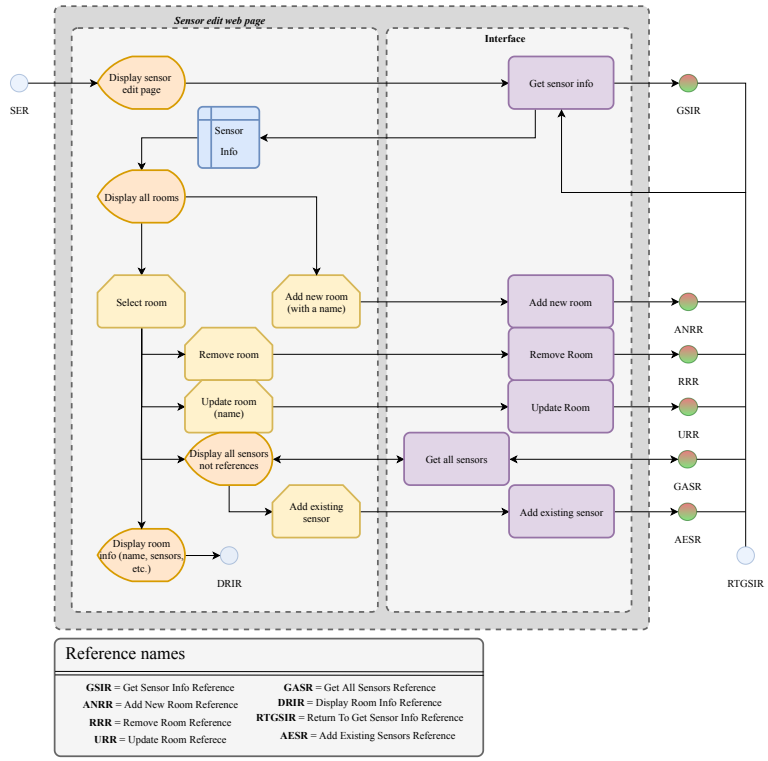


Figure 11: *Model of the SER side part 1, showing the sensor edit functions, and the connecting interface*

This is the top of the SER model. The page starts with prompting the server for sensor info, and will then save the data and lastly displays it. The admin can now choose to select a room, or add a new one. If the admin chooses to add a new one, the admin will give it a name, and send it to the interface, that calls a function on the node.js server. When that function has been called, the program returns to the top to get fresh sensor information from the server.

If the admin chooses a room, the admin is given the option to edit the room name or to remove the room, as well as displaying what sensors are attached to the room, and another box that shows all the sensors that are not attached. The admin can then choose to add an existing sensor to the room.

SER Part 2

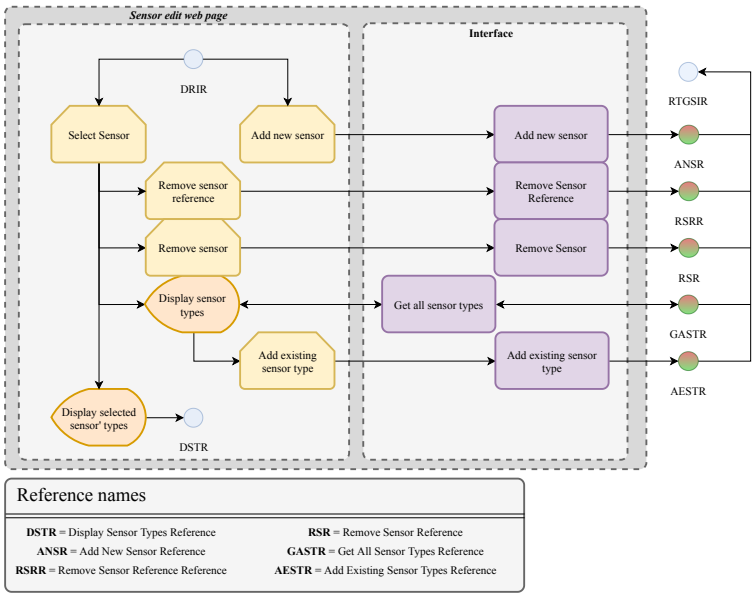


Figure 12: *Model of the SER side part 2, showing the sensor edit functions, and the connecting interface*

The next part of the SER page, is where the admin can choose to make a new sensor, or select a sensor that already exists. The admin can choose to remove the sensor from the room, or remove it from the entire system. Lastly the sensor types are shown just as the room showed which sensors it has. These types could be RH, temperature or CO_2, or a whole different type.

SER Part 3

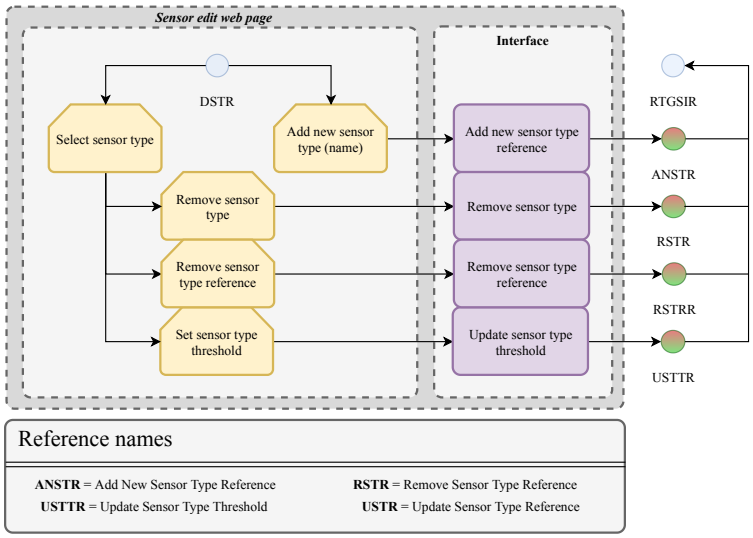


Figure 13: *Model of the SER side part 3, showing the sensor edit functions, and the connecting interface*

The last part of SER is much like the other parts, however here there is also an option to update the threshold of a sensor type.

WASER Part 1

The Warning And Solution Edit Reference (WASER) side of the admin page is next.

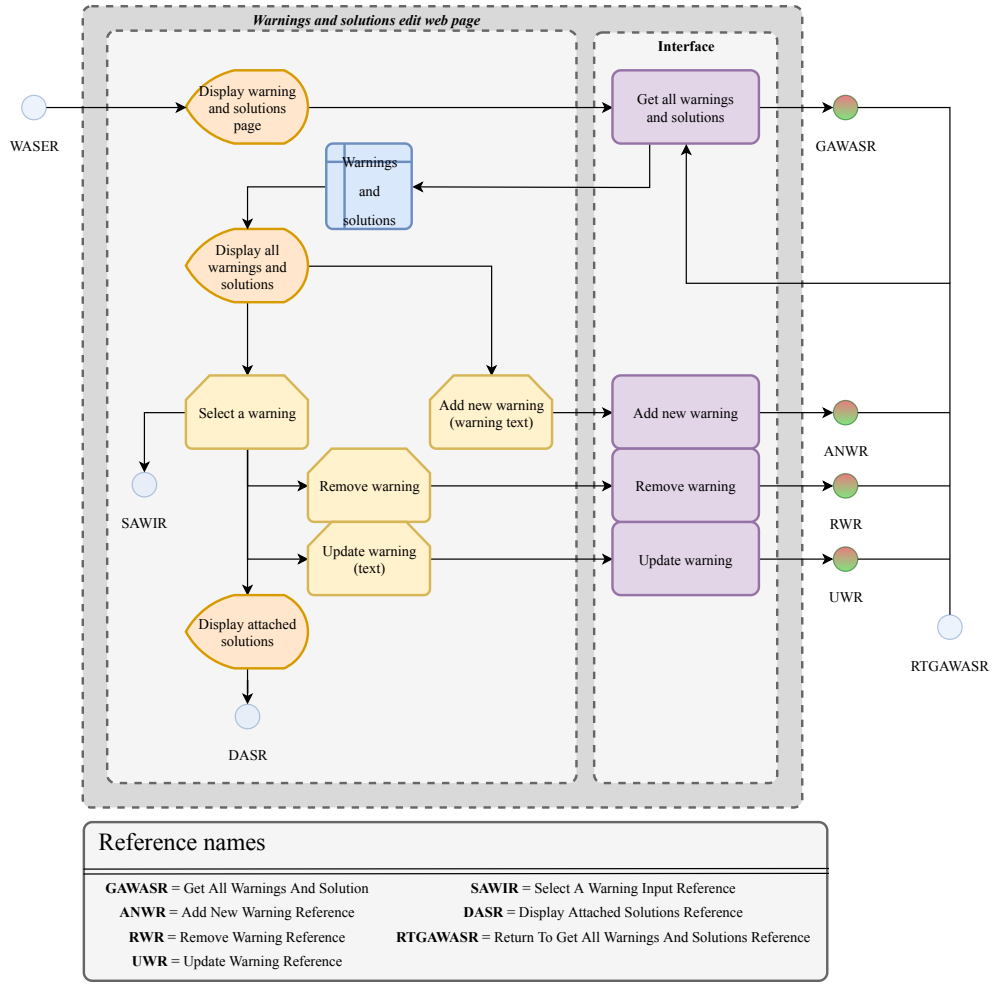


Figure 14: *Model of the WASER side part 1, showing the sensor edit functions, and the connecting interface*

The WASER side of the admin client is again much like the SER side, in which the admin mostly adds, edits and or removes data. The WASER sections will not be explained into detail, given the fact that the structure of this section is very similar to the structure of the SER section.

WASER Part 2

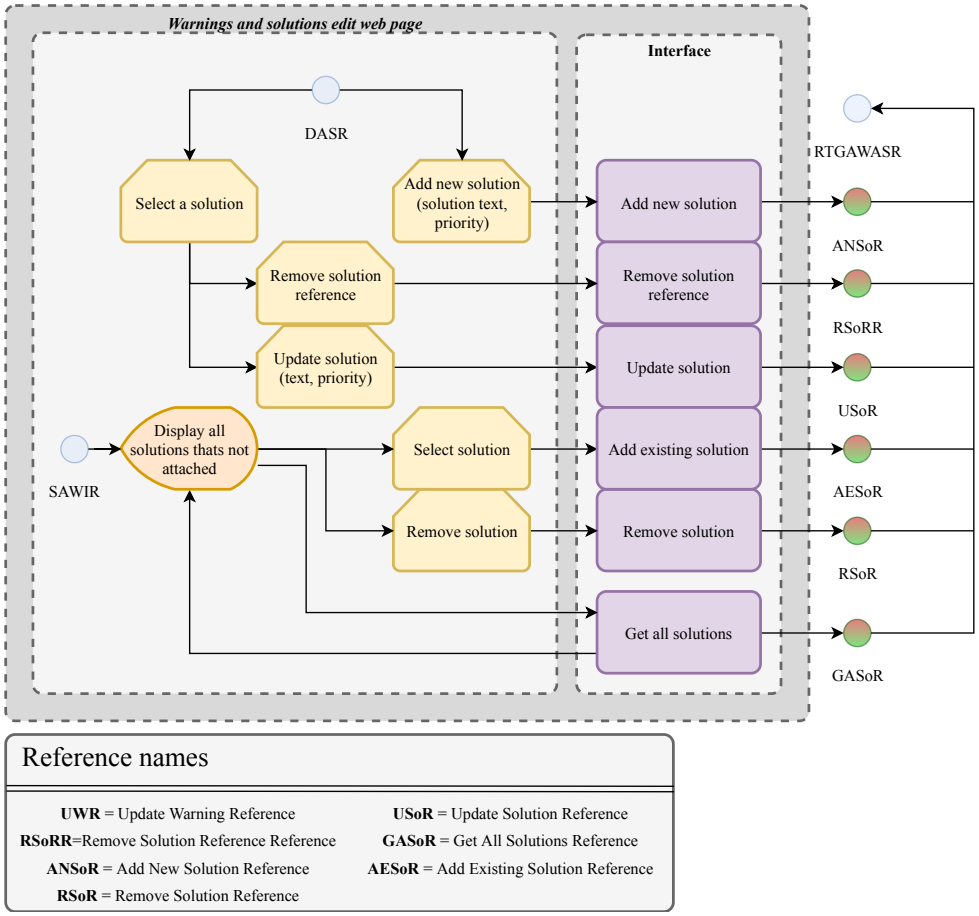


Figure 15: *Model of the WASER side part 2, showing the sensor edit functions, and the connecting interface*

3.2.2 Admin Server-side

The admin side of the server is again split into a WASER references side and a SER referenced side. However they all share a common function, to check credentials, as can be seen in figure 16:

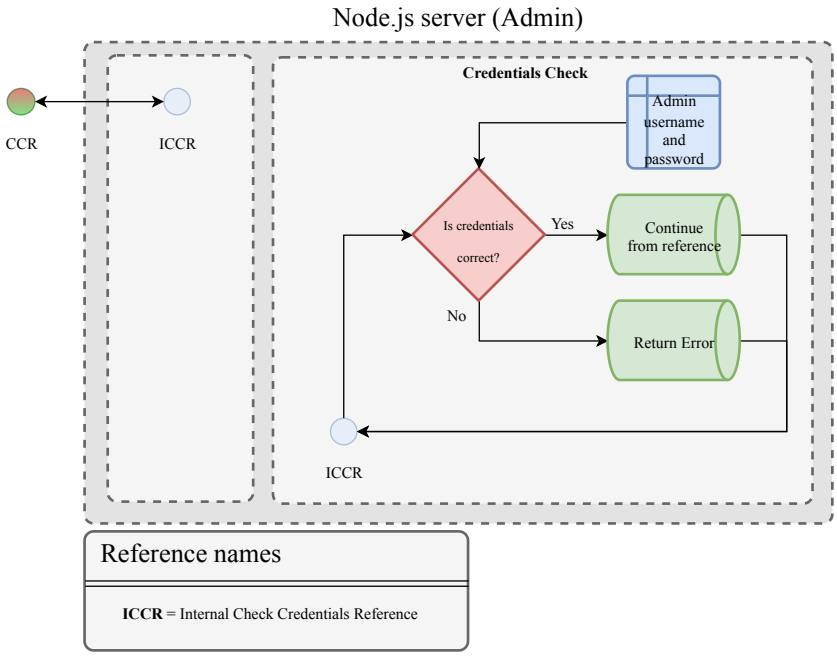


Figure 16: *Model of the admin server side, to check credentials*

Here the credentials for the admin is saved on the node.js server, and checked whenever a admin resource is references. The following models are very simple, and will not be explained into greater detail, however the general idea is that the credentials of the user is checked, to make sure that the user is allowed to edit the database. The green direct data boxes show what data is refereed to in to the database.

SER Server Part 1

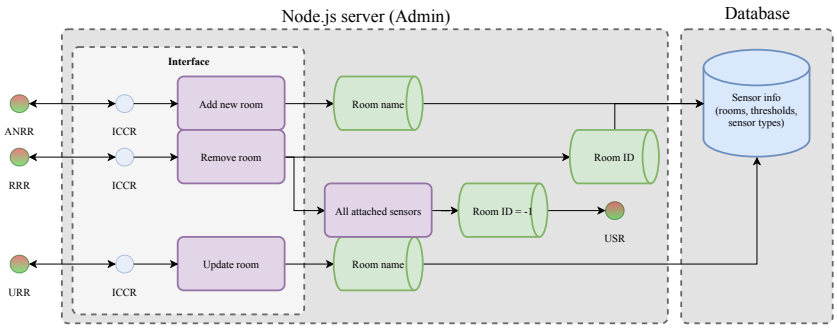


Figure 17: *Model of the SER Server part 1, showing the interface, its backend functions and connectiong to database*

SER Server Part 2

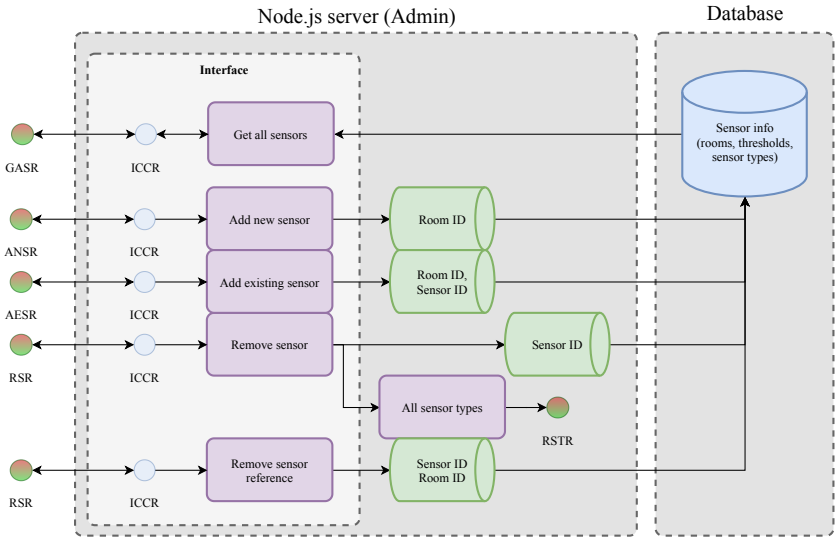


Figure 18: *Model of the SER Server part 2, showing the interface, its backend functions and connectiong to database*

SER Server Part 3

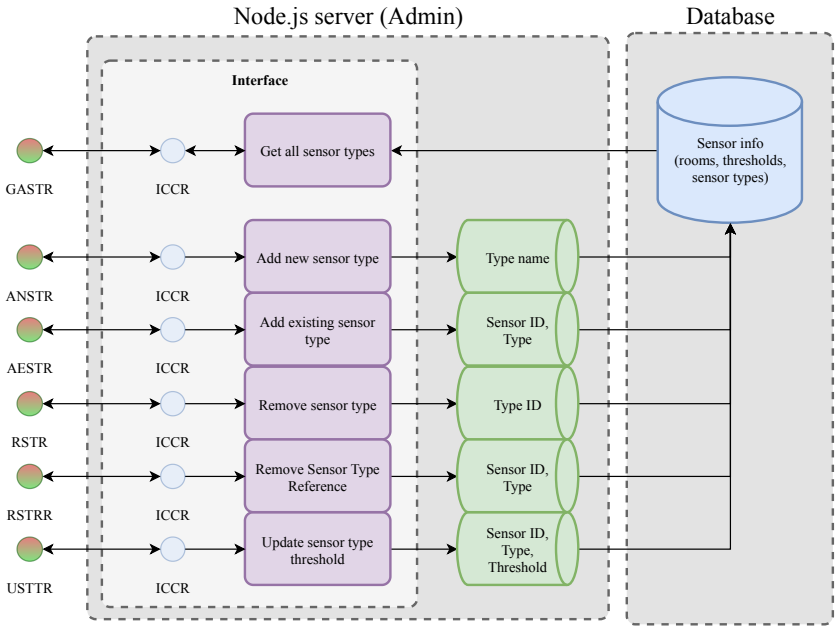


Figure 19: *Model of the SER Server part 3, showing the interface, its backend functions and connectiong to database*

WASER Server Part 1

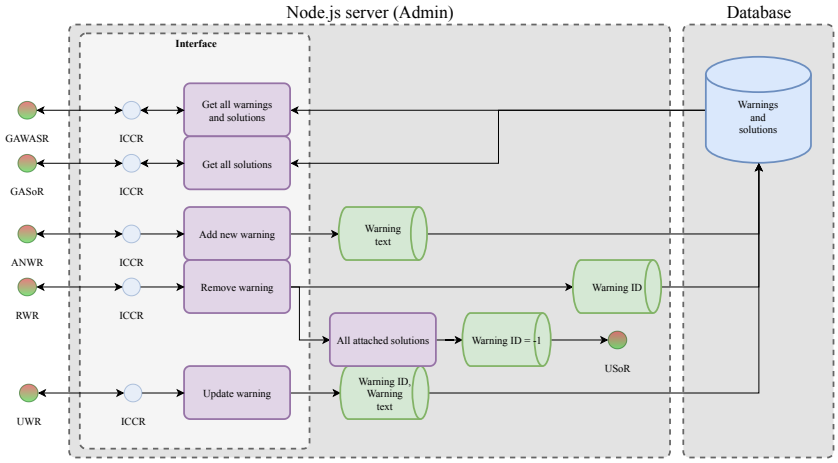


Figure 20: *Model of the WASER Server part 1, showing the interface, its backend functions and connectiong to database*

WASER Server Part 2

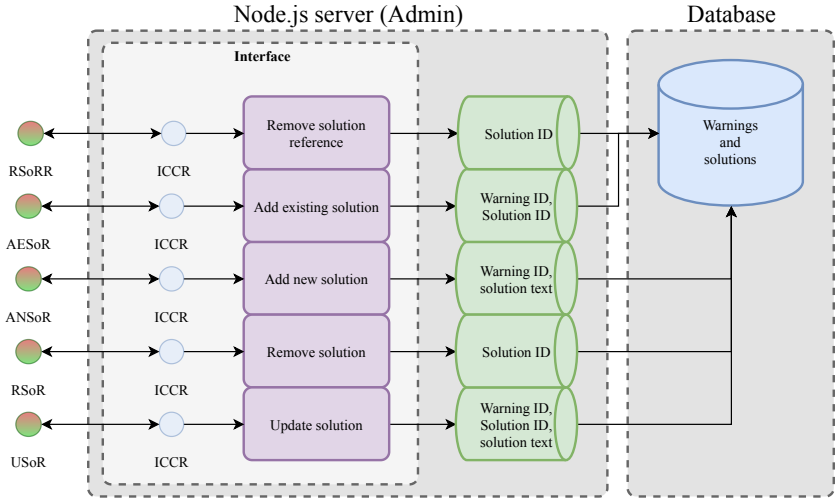


Figure 21: *Model of the WASER Server part 2, showing the interface, its backend functions and connectiong to database*

3.3 Database

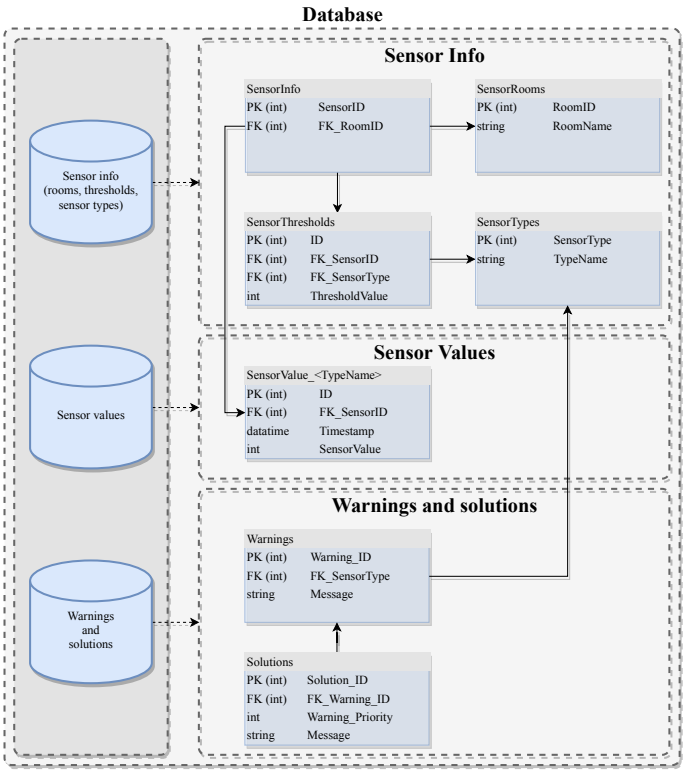


Figure 22: *Model of database, showing the three entry points to the database, and their tables*

The model for the websites database can be seen in figure 22, and consists of four different databases; "Sensor Info", "Sensor Values" and "Warnings and Solutions". These databases are here to provide the information that the node.js server needs. First off is the "Sensor Info" database.

The "Sensor Info" database contains basic information on the sensor network, such as rooms, sensors, sensor types and thresholds for these sensors. At its base there is a table called "SensorInfo", this is the table of sensors which are on the network, these sensors have a primary key, SensorID and a foreign key called FK_RoomID, which informs which room it is. This is then followed by the table "SensorRooms", which consists of the RoomID referenced before, as well as the name of the room. Then there is a table

called "SensorThresholds", and this table consist of an ID, two foreign keys and a threshold value. The first foreign key points to a given SensorID from before, and now also a SensorType, which points to the table "SensorTypes". Such said table consist of a primary key SensorType as well as the name of the sensor, this name could as an example be "CO_2" or "RH".

The next database is called "Sensor Values", it consists of the raw values that the sensors are sending. Here there are as many tables as there are sensor types, meaning that the sensor type "CO_2" values gets put into the table called "SensorValues_CO2". These values all have a foreign key to a SensorID, a timestamp of when the value was put into the database, and lastly the value itself.

The last database is the "Warnings and solutions", it consist of warnings to different sensor types, as well as some attached solutions. The table "Warnings" consist of an id called Warning_ID, a foreign key to a sensor type and a message. This message could be something like "CO_2 levels are too high!". The other table "Solutions" consist of a ID called Solution_ID, a foreign key to a warning id, an integer defining the priority of the solution (0 = high, 1 = medium and 2 = low) and a message.

4 Implementation

This section is about how the program got implemented in code, as well as the differences to the model. This section will again be using flowcharts as well as code snippets to describe the program. During the project GitHub was used, and the project can be found at the following link `https://github.com/kris701/P2-Project`

4.1 Node.js Server

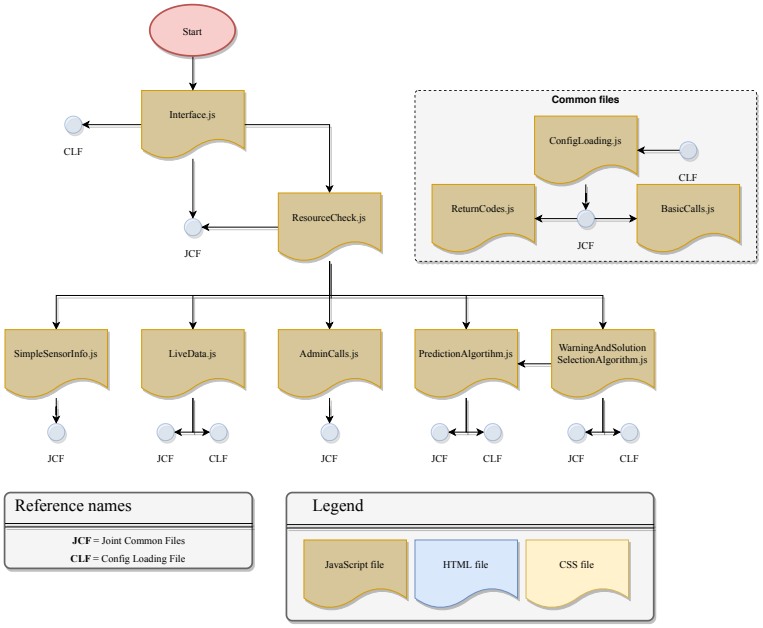


Figure 23: *File tree of the server side*

The figure 23 shows how the files on the server side are related to each other. It can be seen that there are some reference calls, to some common files, such as ReturnCode.js, ConfigLoading.js and BasicCalls.js. Some additions have also been made to the implementation, which were not in the model, mainly the ability to show live data, and loading server configurations from the database.

4.1.1 Return codes

All exported functions on the server side, returns objects between each other. The objects are structured so that there is a message and a return code with it. As an example, the admin section has a function called addNewWarning, this function adds a new warning to the database. If the function worked correctly, it will return the following object:

Listing 1: addNewWarning return object

```
1 returnValue = new retMSG(201, "Warning added successfully!");
```

The codes themselves are loosely based on HTTP codes, where the 200-299 range are success codes and the 400-499 range are error codes. The specific codes can be found in appendix[Section 8.1, Page A in appendix.].

4.1.2 Interface

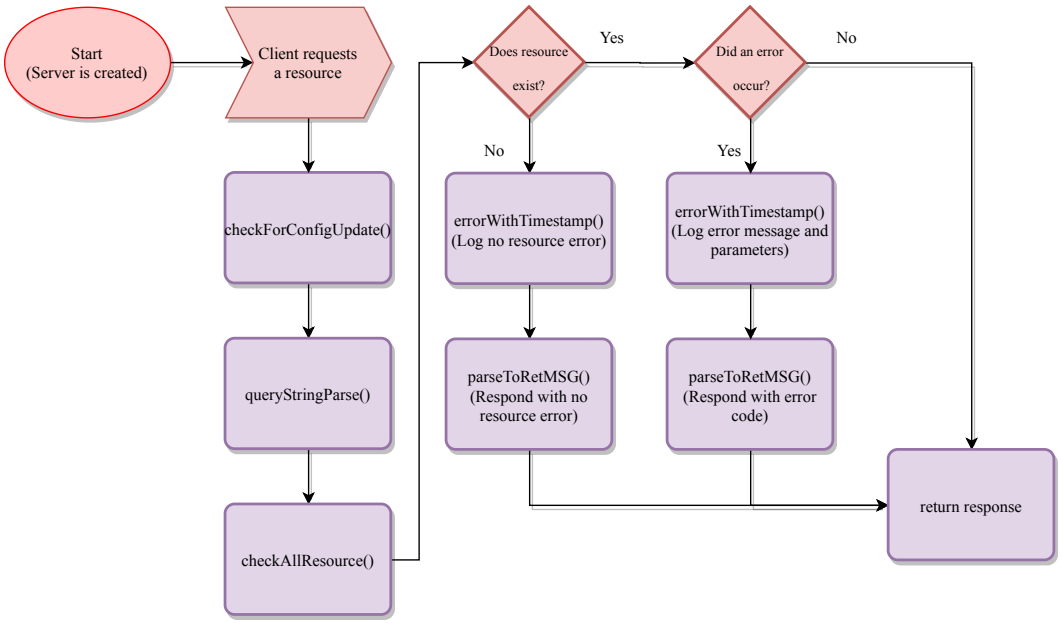


Figure 24: *Flowchart showing functionality of the interface*

The interface is where the server is created and all client requests are handled. Whenever a client requests a resource, the interface first checks if it needs to update the database connection configuration, and if needed it will update the configuration. The querystring the client sent with its request is then parsed into an object readable by the server. When the querystring has been parsed to an object, the server checks if the requested resource exists. The way that resources are checked is by a treeview method. This means that firstly it is checked if the root resource is requested, e.g. "/", which it will almost always be, then the roots subresources are checked, e.g. "getsensorinfo", "getpredictiondata" etc. If the request corresponds to any of these, the respective functions are executed. If it is an admin function then there is another "layer" to the resource tree. It is checked if the word "admin" is in the request, if it is this means that the server knows that the request consist of "/admin". The "admin" resource also have a function attached to it, to check credentials. This makes it so that all the subresources do not also have to check credentials, since it is done from the resource call itself. An example of how this resource treeview looks can be seen below:

Listing 2: Resource library snippet

```
class Resource {
    constructor(name, parameters, functionCall, subResourcesArray) {
        this.name = name;
        this.parameters = parameters;
        this.functionCall = functionCall;
        this.subResourcesArray = subResourcesArray;
    }
}

const ResourceLibrary = new Resource("/", [], function () { return true
    }, [
    new Resource("api-doc", [], getOpenAPI, []),
    new Resource("getlivedata", ["roomID", "date"], LDC.getLiveData, [])
        ,
    new Resource("getsensorinfo", [], SSIC.getSensorInfoQuery, []),
    new Resource("getpredictiondata", ["room", "date"], PAC.
        getPredictionDatetimeQuery, []),
    new Resource("getwarningsandsolutions", ["room", "date"], WASC.
        getWarningsAndSolutions, []),
    new Resource("admin", ["username", "password"], ACC.checkCredentials
        , [
        new Resource("login", [], function () { return new BCC.retMSG(RC
            .successCodes.CredentialsCorrect, "Credentials correct!") },
             []),
        new Resource("getallwarningsandsolutions", [], ACC.WASC.
            getAllWarningsAndSolutions, []),
        new Resource("addnewwarning", ["sensorType", "message"], ACC.
            WASC.addNewWarning, []),
        new Resource("removewarning", ["warningID"], ACC.WASC.
            removeWarning, []),
        new Resource("updatewarning", ["warningID", "message"], ACC.WASC
            .updateWarning, []),
        new Resource("addnewsolution", ["warningID", "priority", "
            message"], ACC.WASC.addSolution, []),
        ...
```

If a requested resource does not exist, the server logs this as an error and returns an error message telling the client that the resource does not exist. In the case that it does not exist, the server checks if the resource encountered an error and if so, logs the error message from the resource and returns that error message to the client as well. If no error occurred in the resource, the server returns the response from the resource to the client.

A querystring[15] is a part of the URL the client uses when it requests something from the server, and is most commonly marked with a question mark at the beginning of the string. Each value is set to an identifier for this value with an equal sign and all sets of identifiers and values are separated with an ampersand. A querystring can thus look like this:

Listing 3: querystring example

```
1 https://website/resourcename?param1=value1&param2=value2
```

In this case, resourcename is the resource the client requested and in the querystring, param1 and param2 are identifiers for their respective values value1 and value2.

4.1.3 Sensor info

The function to receive sensor info, is all contained in a single file "SimpleSensorInfo.js" This file consists mostly just of SQL queries to the database, to construct an object to the client side. For this there are two objects, which are used in the return object:

Listing 4: SimpleSensorInfo.js objects

```
...
class Sensor {
    constructor(sensorID, types) {
        this.sensorID = sensorID;
        this.types = types;
    }
}

class Room {
    constructor(roomID, roomName, sensors) {
        this.roomID = roomID;
        this.roomName = roomName;
        this.sensors = sensors;
    }
}
...
```

A flowchart can be made, representing how the function works:

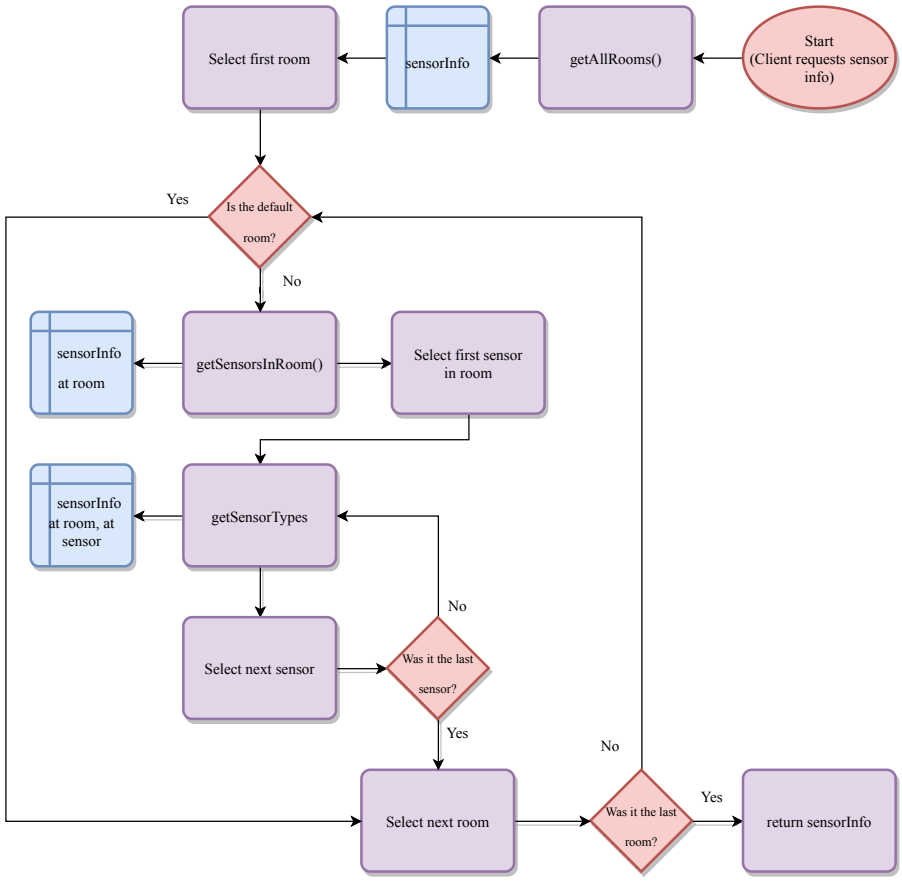


Figure 25: *Flowchart showing the functionality of the SimpleSensorInfo.js file*

When the client requests sensor info, the server first retrieves all rooms on the database, and puts them in the return object, as a "Room" object. The server then loops through all rooms, except for the default room, and retrieves all sensors in each room, which are stored under their linked room as a "Sensor" object. Each time it has retrieved a set of sensors, it loops through the sensors, retrieving the sensortypes of each sensor and storing it within the sensors data. Once the server has looped through all rooms, it returns the info it has stored.

4.1.4 Prediction Algorithm

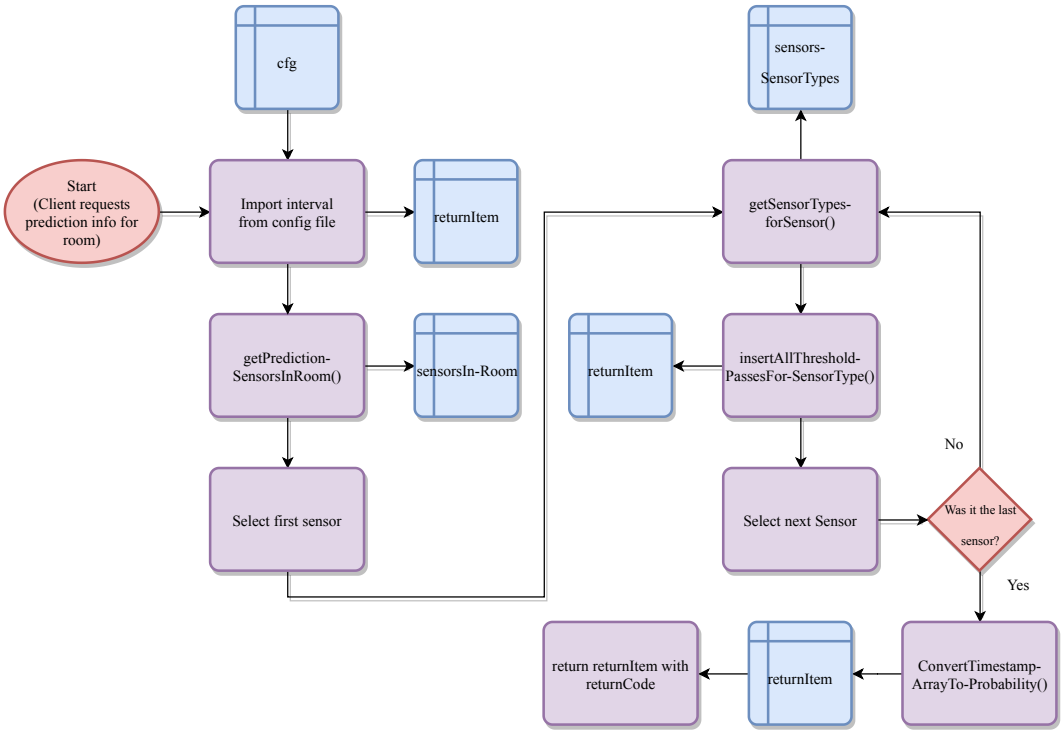


Figure 26: *Flowchart showing functionality of the prediction algorithm*

This function also has its own file "PredictionAlgorithm.js", and it is the largest and most complex part of this project. Whenever this function is called it firstly gets all the sensors in the selected room, as well as all their sensor types. For each of these sensor types, a query to the database is made to get all the times between the date selected and the amount of weeks which is looked back in time, in this case being 10 weeks. To assemble all the sensor types from all the sensors in a single object, it is firstly checked, in the return array, if there is already an object for the selected sensor type.

This means that if the first sensor has been run through, consisting of a CO_2 and RH sensor, when the next sensor is then run through, consisting of a CO_2 and Temperature sensor, the return array would end up consisting of only three not four objects. E.g.
"[CO2{...}, RH{...}, Temperature{...}]"
and not
"[CO2{...}, RH{...}, CO2{...}, Temperature{...}]".
This is done in the "insertAllThressholdPassesForSensorType" function. Whenever an existing sensor type is found, or that a new one is made, the server now fetches all the entries from the database, within the date selected, where the sensor threshold has been passed.

This returns a potentially large array of entries, which could be sized down. For that reason the array is split into intervals, as an example 15 min intervals, where only the first entry within that interval is selected. An example can be made:

- Timestamp: 12:02:00, value: 500
- Timestamp: 12:03:00, value: 550
- Timestamp: 12:04:00, value: 506

Will be turned into:

- Timestamp: 12:00:00 to 12:15:00, value: 500

This significantly reduces the size of data returned to the client, with minimal data loss.

When all the sensors have been run trough, all the timestamps gets converted to a probability. This is done in the "convertTimestampArrayToProbability" function. This is done by taking the timestamps of which the threshold was exceeded, weighing them, and dividing them by the amount of weeks the function looks at. The weight function is taking in the date that is being looked from, and gets the day difference to the timestamps in the array. The older the entry, the lower the value. It is done linearly, since it is the simplest way to weight the timestamps. The way that this function works, can be seen in the following snippet:

Listing 5: Weight class in PredictionAlgorithm.js

```
...
class WC {
    static getAgeWeight(timestamp, date) {
        let daysSince = WC.getDaysSince(timestamp, date);
        let weight = WC.weightConverter(daysSince);
        return weight;
    }

    static getDaysSince(timestamp, date) {
        let timestampDate = new Date(timestamp);
        let senderDate = new Date(date);
        return ((senderDate.getTime() - timestampDate.getTime()) /
            millisecondsPerDay);
    }

    static weightConverter(timeSince) {
        let retWeight = parseFloat(cfg.PAC_WC_A) * timeSince +
            parseFloat(cfg.PAC_WC_B);
        if (retWeight < 0)
            retWeight = 0;
        return retWeight;
    }
}
```

Where the "cfg.PAC_WC_A" and "cfg.PAC_WC_B" are the a and b in the linearly function, that can be configured from the database. When the entries have been weighted, summed and divided by max week reach, out comes an array of probabilities, that can be sent to the client.

The mathematical version of the prediction algorithm can be seen below:

$$p(\text{Will pass again}) = \frac{\sum_{n=1}^{n} W(E(i,t,int),t)}{n} \tag{1}$$

This can be generally seen as all the times in the past x weeks that the threshold have been passed at this date. Take that number, weighted by its age, and divide it by x weeks that it looks back, the output is then a value between 0 and 1 representing percentages.

First there is a function to get the time difference between two timestamps in days:

$$T(t_1, t_2) = \text{Time difference between } t_1 \text{ and } t_2 \text{ in days} \tag{2}$$

Then there is a function to get a weighted value from the age of the timestamp. The function is a linear function:

$$W(t_1, t_2) = T(t_1, t_2) * a + b \tag{3}$$

Where a is -0.028 and b is 2. The reason for this a, is so that if the time difference in days are the maximum, e.g. 70 days, then the weight would end up roughly being 0:

$$70 * -0.028 + 2 = -1.96 + 2 = 0.04 \tag{4}$$

The weight 0.04 is such a small value, that it does not really have much influence. This a, b and n can be easily changed, as they are simply just values inside the database.

$$E(x,t,int) = \text{First Passed threshold at t time, within int minutes, x weeks ago} \tag{5}$$

4.1.5 Warnings and Solutions Algorithm

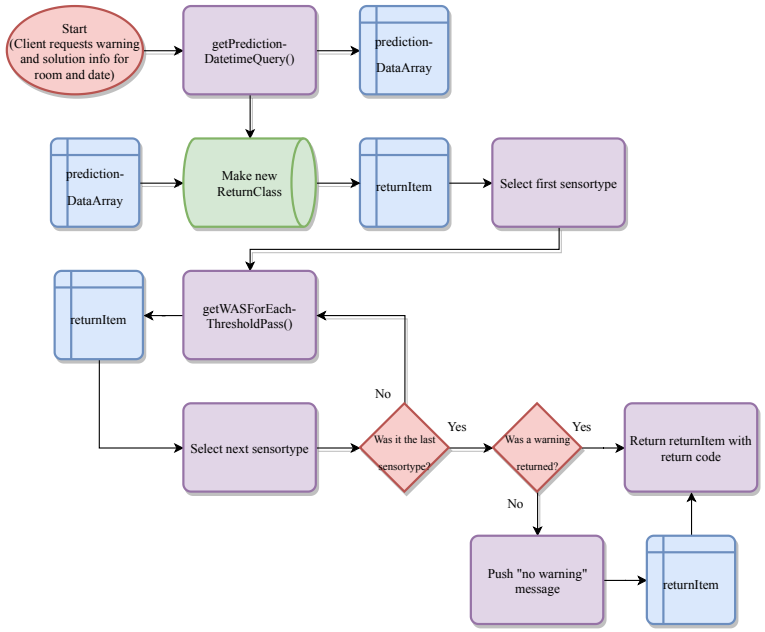


Figure 27: *Flowchart showing functionality of the warning and solution selection algorithm*

In this section, the server retrieves warnings and solutions for a prediction. When the client requests warnings and solutions for a specific room and time, the server first produces prediction data via the previous section. This is saved in an array, which is then run through for each sensor type in the array. For each sensor type the program runs through all threshold passes and gets warnings and solutions for each of them, which is saved in a return item. The program then checks if there was at least one warning found and if not, pushes a ”no warning” message to the return item and lastly sends the return item to the client with a return code.

4.1.6 OpenAPI

An openAPI has also been made for the project, to make interfacing between server and client side simpler. It also makes it simpler for external programs to use the data that the program provides. It utilises openAPI 3.0.0. The API describes how one should request data from the server, as well as the structure of the return data. The openAPI can be accessed on the website of this project, by the resource "/api-doc", this will return a JSON object describing the openAPI. The API can also be viewed in SwaggerHub, where one can also interact with the API (`https://app.swaggerhub.com/apis/kris701/P2Project/1.0.0-oas3#/`).

The API is split up into several sections, mainly the UserLevel and the AdminLevel. The UserLevel functions are only get functions, and requires no admin credentials to fetch. The AdminLevel is further split into four subsections, Get, Put, Delete and Path. All the admin functions requires that there is a username and password sent with them. The API is designed in YAML, an example of which can be seen below:

Listing 6: snippet of the openAPI

```yaml
openapi: 3.0.0
info:
  version: "1.0.0-oas3"
  title: iaq_mon_plat_api
  description: "The API for IAQ Monitoring Platform"


servers:
 - url: https://dat2c1-3.p2datsw.cs.aau.dk/node0/

components:
  parameters:
    adminUsername:
      in: query
      name: username
      required: true
      schema:
        type: string
      description: "The username of an admin"
    adminPassword:
      in: query
      name: password
      required: true
      schema:
        type: string
      description: "The password of an admin"
...
```

4.2 Client-side

The client side is split into two sections, one being the main page, accessible all the time and the admin pages, only accessible by admins.

4.2.1 Main page

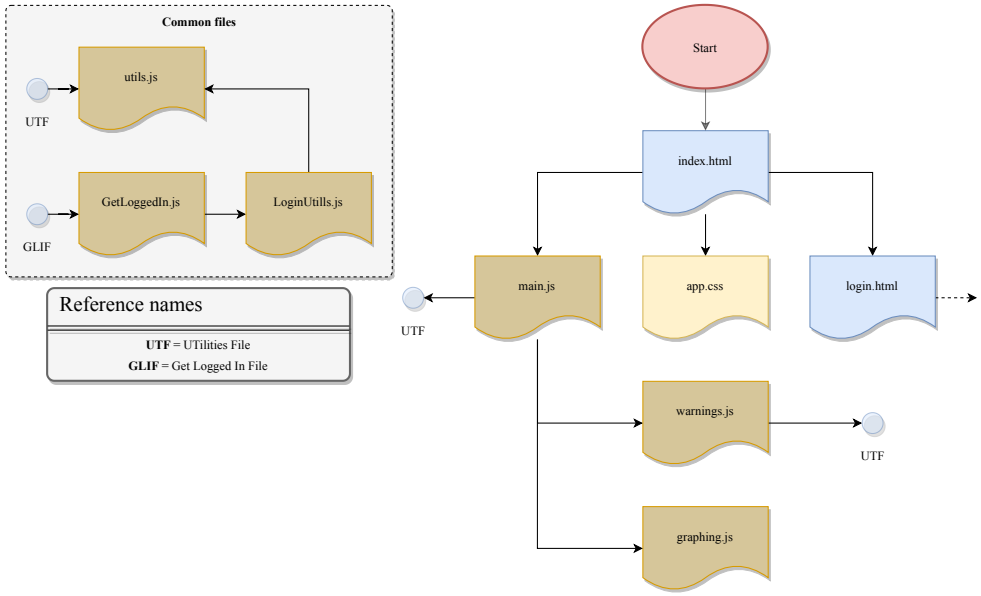


Figure 28: *Flowchart of file interaction on the main page*

The file interactions can be seen in figure 28. In general it is the index.html holding it all together, as it is the entry point onto the website. From there a reference to a html page called login.html is made, this is referring to the admin pages that will be explained later. Other than that, there are some common files, just as there were on the server side, where this time there are the files utils.js, GetLoggedIn.js and LoginUtils.js. As an additional library, the client side also uses the charts.js (`https://www.chartjs.org/`) to make it simpler to display graphs. The reason for using a external library for this, is that making one would require a significant amount of time, and would drain time and resources out of other more important parts of the project.

The majority of the action that happens on the main page is from one function, "roomChangeFunction()". This is the function that clears the en-

tire ”field”, fetches fresh data from the server and lastly displays said data. An overview on what happens in the function can be seen below:

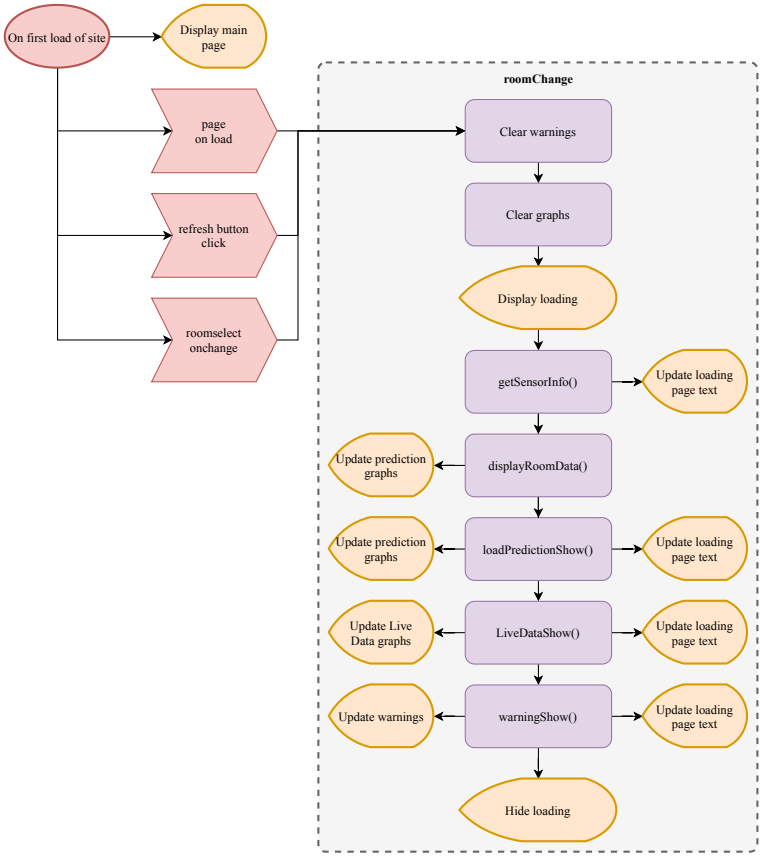


Figure 29: *Flowhcart over the roomChangeFunction*

It can be seen that there are several functions running from this. They all correspond to a ”function” on the server side, take ”getSensorInfo()” as an example that calls the server resource ”getsensorinfo”. The most interesting function to take a look at is the graphing functions, that takes in fresh data from the server and displays it in graphs. The ”loadPrediction-Show()” function is showed in the flowchart below. This function and the ”liveDataShow()” function are much the same, so only the prediction one will be explained:

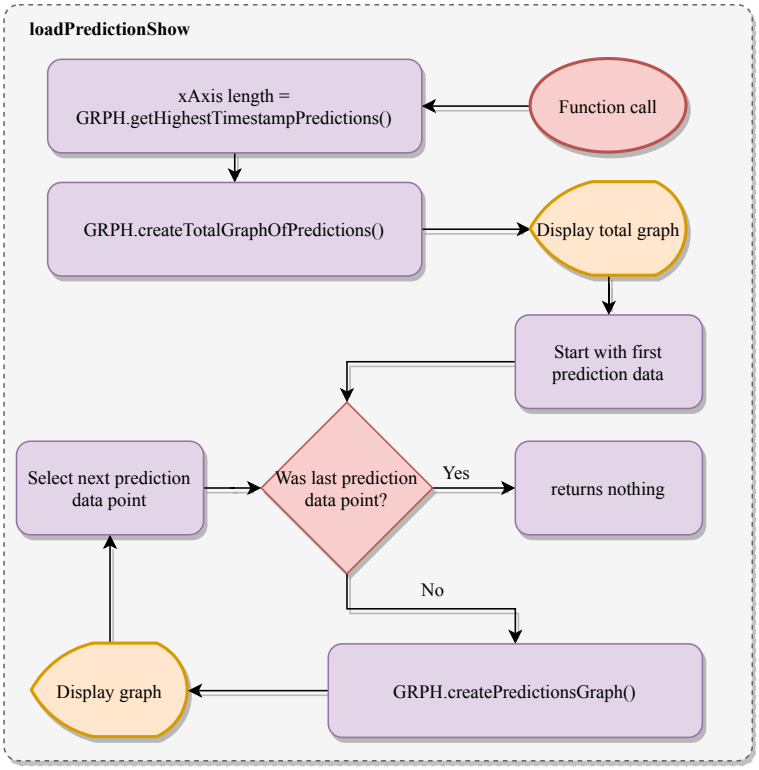


Figure 30: *Flowhcart over the loadPredictionShow function*

The function consists of several smaller functions, that gets a little bit complex. The "getHighestTimestampPrediction" function, looks through all the data, and finds the oldest timestamp available, which will then represent the depth of the x axis in the graph. This value is also used in displaying all the prediction graphs, so they are easier to compare next to each other, when they have a common x axis. The function "createTotalGraphOfPredictions()" function is a function that shows one graph with all the sensor type values in, to again make it easier to compare values. It consist of two major sub functions, as can be seen in the figure below:

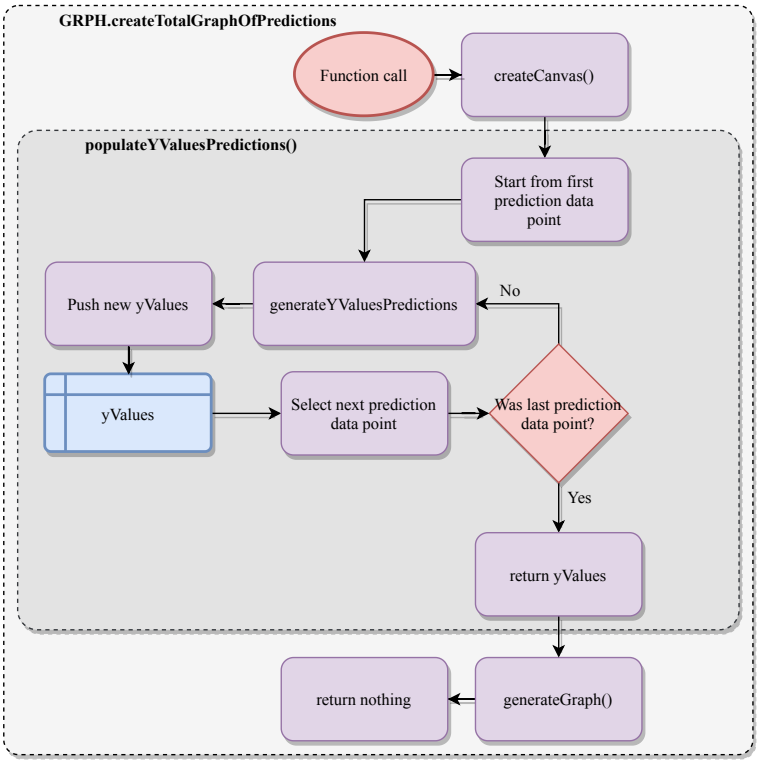


Figure 31: *Flowchart over the createTotalGraphOfPredictions function*

Here it can be seen that another function is called, "generateYValuesPredictions()", which is the function that populates the x axis with values. It is designed so that if there is a "hole" in the data received from the server, said hole will be filled out with zeros. An example of that can be made, if the client receives the following sets of points for the graph:

(x value, y value)
(1,50) (2, 54) (3, 63) (6, 45) (9, 55)

This will then be transformed into an array, that the graph.io library can understand:

(1,50) (2, 54) (3, 63) (4, 0) (5, 0) (6, 45) (7, 0) (8, 0) (9, 55)

A flowchart of its operation can be made, showing how the function operates:

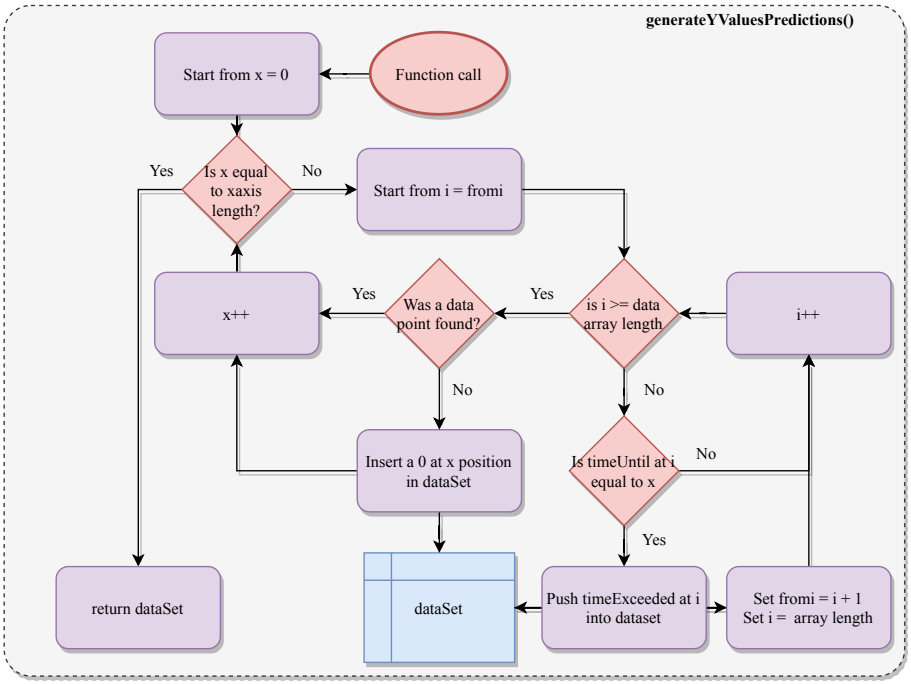


Figure 32: *Flowhcart over the generateYValuesPredictions function*

Where the function strides all x values from 0 until the maximum length of the x axis (found earlier) and puts a corresponding timestamps y value in an array, and if there is no matching timestamp, a zero is put in instead.

Finally there are also the function to show a graph over a single sensor type, it also uses the "generateYValuesPredictions()" function:

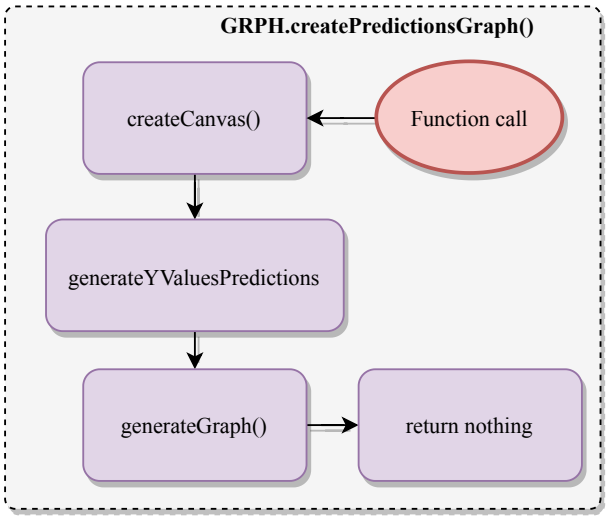


Figure 33: *Flowchart over the GRPH.createPredictionsGraph() function*

This concludes the displaying of graphs, and thereby also the main page implementation description. Most of the other functions in the main page are fairly rudimentary, and are therefore not described in this report. Interested readers may access these functions from the projects github page at `https://github.com/kris701/P2-Project/`

4.2.2 Admin pages

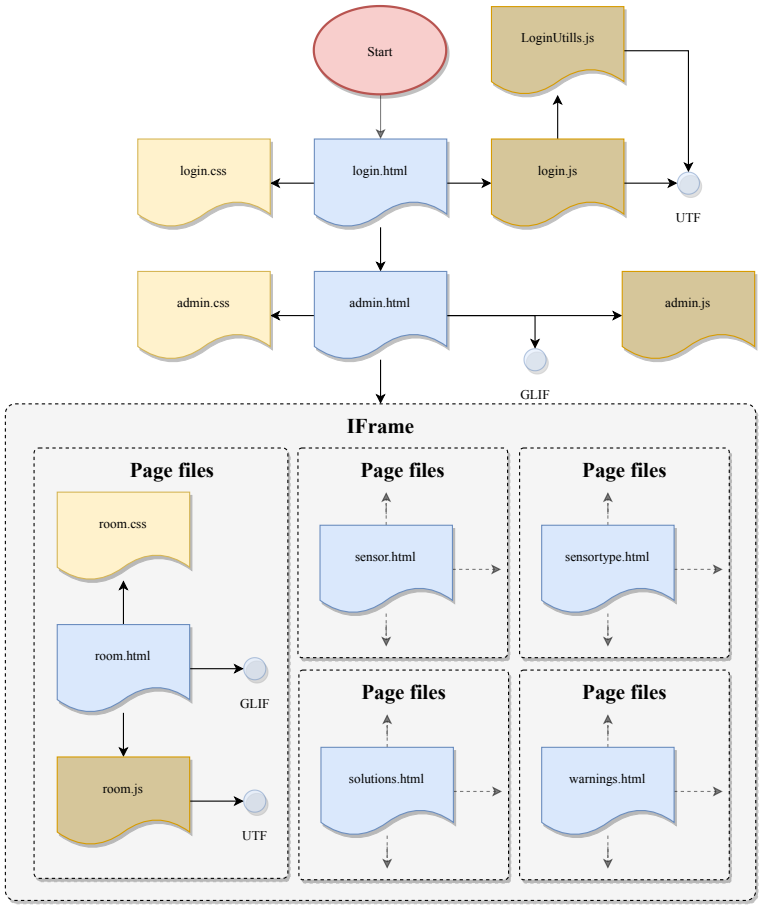


Figure 34: *Flowchart over file interaction on the admin page*

The admin pages use some of the same utilities files from the main page, however here there is an IFrame holding all the subpages of the admin page together. These pages are structured the same, with a html, css and js file for each, where the only difference is the name of the file. All these five pages can be reached from the admin.html page.

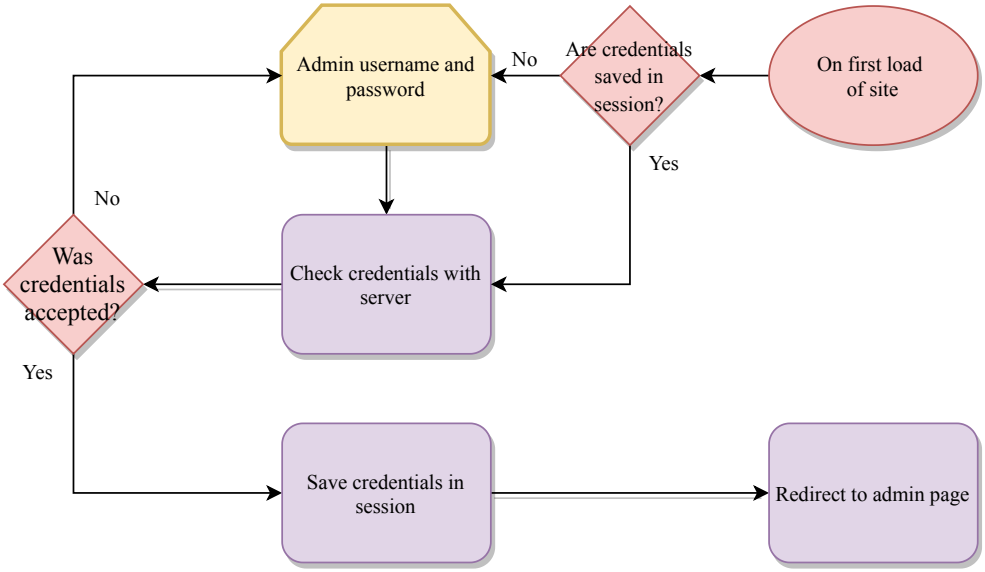


Figure 35: *Flowchart over login system*

The start of the admin page is the same as the model. The user will be prompted to enter username and password. These will be checked with the server to see if it's correct. If it's correct it will be saved in a session so the user won't have to enter username and password in that session again.

All the sub pages in the admin page is all structured the same way. To understand the structure, the room management page will be explained. On the page there are different functions, the name of these functions correlate to the name of the functions in the code. On the following page is shown a sample call tree of one buttons function in admin room management.

Listing 7: addNewRoomSubmitButton in room.html (HTML5)

```
...
<div id="addNewRoomMenu" style="display:none">
    <button id="addNewRoomBackButton" class="addRoomButton">Back</
        button>
    <input id="addRoomInput" class="inputClass" placeholder="Enter room
        name" />
    <button id="addNewRoomSubmitButton" class="addRoomButton">Add New
        Room</button>
</div>
...
```

Listing 8: addNewRoomSubmitButton in room.js (javascript)

```
...
let addNewRoomSubmitButton = document.getElementById("
    addNewRoomSubmitButton");
addNewRoomSubmitButton.onclick = addNewRoomSubmitButton_Clicked;
...
```

Listing 9: addNewRoomSubmitButton_Clicked in room.js (javascript)

```
...
async function addNewRoomBackButton_Clicked() {
    await initialLoad();
}

async function addNewRoomSubmitButton_Clicked() {
    await submitNewRoomButton();
...
```

Listing 10: submitNewRoomButton in room.js (javascript)

```
async function submitNewRoomButton() {
    let returnMessage = await UC.jsonFetch(
        "https://dat2c1-3.p2datsw.cs.aau.dk/node0/admin/addnewroom", [
            new UC.FetchArg("username", sessionStorage.getItem("username
                ")),
            new UC.FetchArg("password", sessionStorage.getItem("password
                ")),
            new UC.FetchArg("roomName", addRoomInput.value)
    ]);
    checkReturnCode(returnMessage, "New room added succesfully!");

    await initialLoad();
}
```

This way it is easy to find the desired function in the code and make changes if wanted. The page is setup to work so when you click on a function that function in the code is setup as an event setup that will show the appropriate functions for the chosen function. If needed the page will fetch relevant data from the database and/or will display function related to the chosen task.

The following flowcharts are there for reference, since they are structured the same way. Because of this there won't be any explaining of them.

Room management flowchart

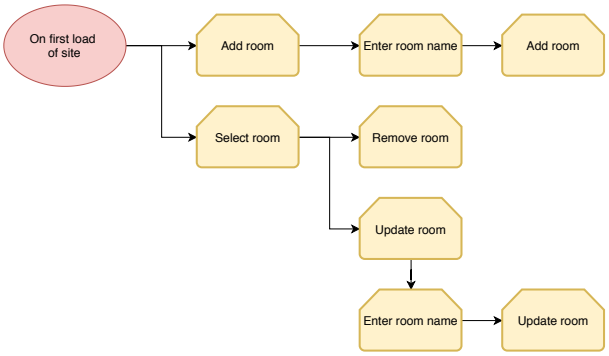


Figure 36: *Flowchart over room management page*

Sensortype management flowchart

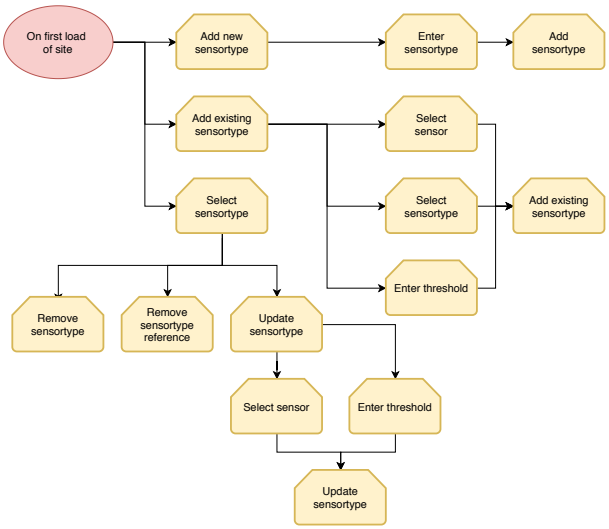


Figure 37: *Flowchart over sensortype management page*

Sensor management flowchart

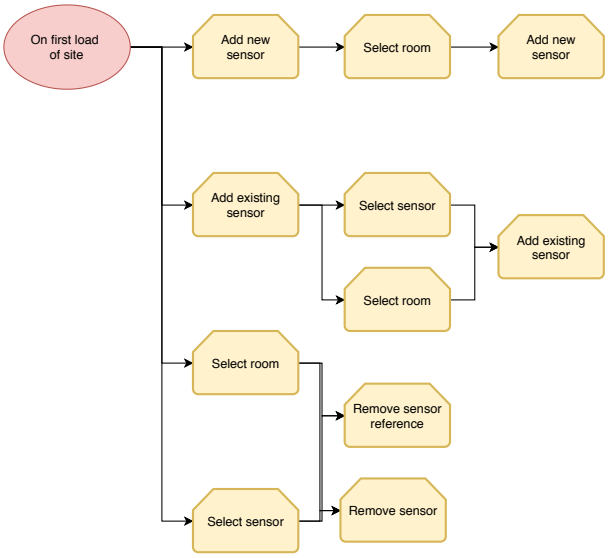


Figure 38: *Flowchart over sensor management page*

Warnings management flowchart

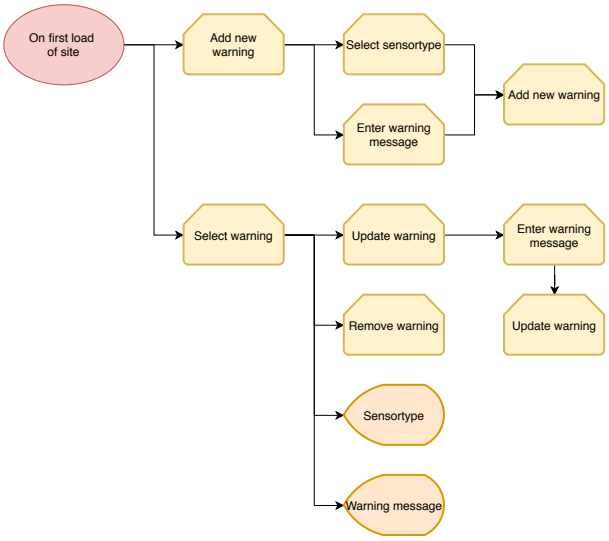


Figure 39: *Flowchart over warnings management page*

Solutions management flowchart

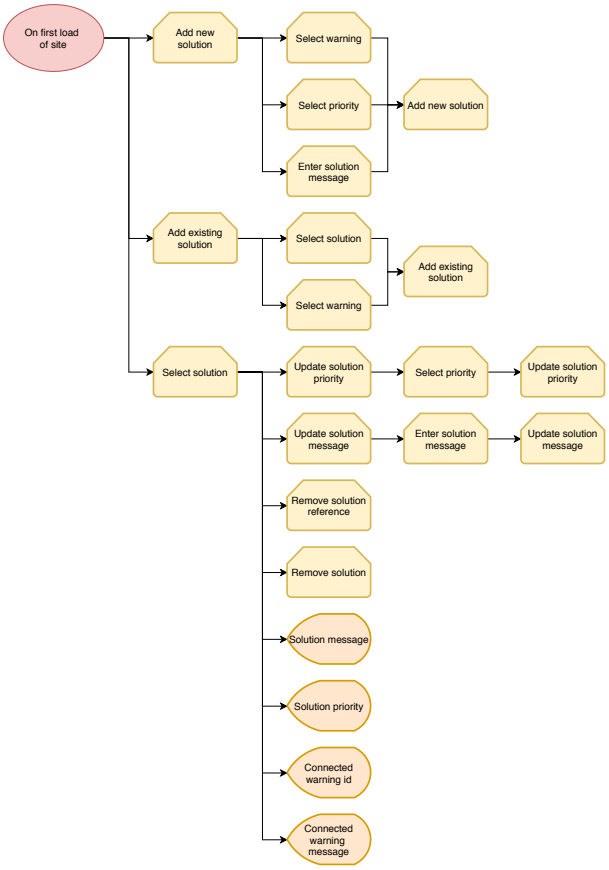


Figure 40: *Flowchart over solutions management page*

4.3 Database

The implementation of the database follows the model. The only difference is the addition of some more utilities tables and a table that contains solution priority:

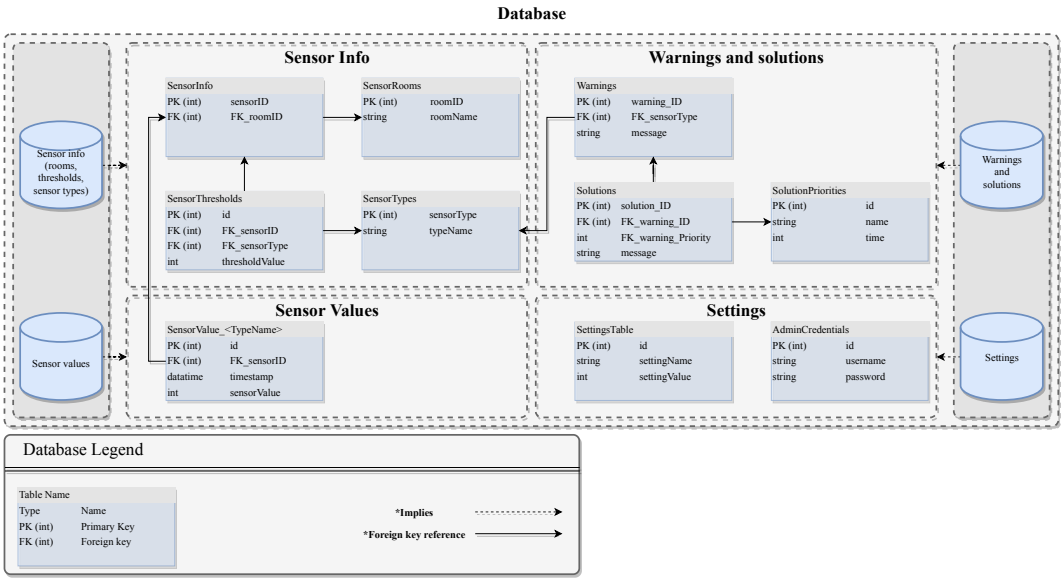


Figure 41: *Flowchart of the implementation of the database, with the addition of a settings table*

The solution priority table is there so that more solutions can be made, as well as edit the time frame for when they should trigger. The settingsTable consist of some static variables, which was directly in the node.js server before. However changing them required the server to restart, which is an unfavorable thing to do in web-dev. So those variables are placed in the database now, these then gets loaded if a set amount of time has passed since the last time it loaded its settings from the database. The setting to tell how long there should go between config updated is also placed in the database with the other settings.

The last addition table is the AdminCredentials table, which contain credentials for admin logins, again, to remove static variables from the server.

4.4 Program correctness

This section will discuss some of the more advanced part of the program correctness and prove that the algorithms are correct.

4.4.1 Prediction Algorithm

The simplest way to test if the algorithm works, is to try to run it manually, and compare the result to what comes out of the program. The core of the algorithm is the SQL query:

Listing 11: Prediction Algorithm SQL query

```
SELECT * FROM SensorValue_CO2 WHERE sensorID=2 AND TIME_TO_SEC(TIME(
    timestamp)) >= 39600 AND TIME_TO_SEC(TIME(timestamp)) <= 40140 AND
    sensorValue>=1000 AND timestamp >= "2020-02-06 11:00:00" AND
    timestamp <= "2020-04-16 11:00:00" AND WEEKDAY(timestamp) = 3
```

This query is set up to take all the sensor values from the table SensorValue_CO2, with the sensor id 2, timestamp is to be larger than 39600 seconds and less than 40140 seconds. That interval corresponds to the time 11:00:00 until 11:15:00. Then it also takes only those entries that have a sensor value above the threshold, in this case 1000 PPM. Then it also only takes values that is newer than the 6th of February, this corresponds to the date that its looking from, 16th of April, subtracted by 10 weeks. It then also only gets values whose timestamp are less than the date 16th of April. Lastly it only gets entries that is on the correct weekday, in this case 3 stands for Thursday. When entering this large query into the database, a table of five entries is outputted:

id	sensorID	timestamp	sensorValue
31138	2	2020-03-12 11:00:56	1077
31139	2	2020-03-12 11:06:06	1077
33249	2	2020-03-19 11:01:49	4715
33250	2	2020-03-19 11:06:58	4715
33251	2	2020-03-19 11:07:07	4715

The interval that is looked within is 15 minutes, so all in all this corresponds to an actual output of:

id	sensorID	timestamp	sensorValue
31138	2	2020-03-12 11:00:56	1077
33249	2	2020-03-19 11:01:49	4715

Since the first entry within this interval, is the one that is taken.

A look can now be taken on the equation from earlier:

$$p(\text{Will pass again}) = \frac{\sum_{n=1}^{n} W(E(i,t,int),t)}{n} \tag{6}$$

The values are now n = 10, t = 16Th of April at 11:00:00 and int = 15. We can then go through the sum equation, and assign the weights:

$$timestamp_1 = E(n, \text{6Th of April at 11:00:00}, 15) = \text{12Th of March at 11:00:56}$$

$$outWeight_1 = W(timestamp_1, \text{6Th of April at 11:00:00}) = 1.02$$

$$timestamp_2 = E(n, \text{6Th of April at 11:00:00}, 15) = \text{19Th of March at 11:01:49}$$

$$outWeight_2 = W(timestamp_2, \text{6Th of April at 11:00:00}) = 1.216$$

These can then be summed up, and put instead of the summation equation:

$$p(\text{Will pass again}) = \frac{1.02 + 1.216}{10} = 0.22 * 100 = 22\% \tag{7}$$

From doing this manually a predicted value of 22% is set. With the same settings on the website, a value of exactly 22% is also given, at the 15 min mark as can be seen in figure 42 below. This thereby proves that the prediction algorithm is correct.

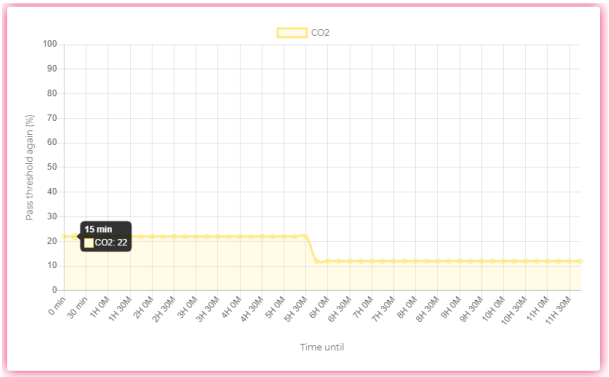


Figure 42: *Result for CO_2 on the project website, showing 22%*

4.5 Unit testing

It is of high importance to unit test code. Testing the code on a unit level basis, makes sure that the code written is of high quality, and simplifies trouble shooting. The method of unit testing that has been conducted is on an export level basis, where it is all exported functions, e.g. functions that can be called from other files, that is being unit tested.

Unit testing has been carried out on the server side, with the help of the test framework Mocha (`https://mochajs.org/`). All the unit tests use a general test class, consisting of the following test options:

Listing 12: *GeneralTests.js* Unit test options

```
module.exports.GTC = class {
    static shouldFailWithNoParameters(...) { ... }
    static shouldFailWithnoParametersSimple(...) { ... }
    static shouldReturnArray(...) { ... }
    static shouldReturnArrayDotData(...) { ... }
    static shouldReturnObject(...) { ... }
    static shouldReturnADateObject(...) { ... }
    static shouldReturnAString(...) { ... }
    static outputArrayMustBeLargerThanDotData(...) { ... }
    static shouldFailIfTargetIDIsDefaultID(...) { ... }
    static expectErrorCodeFromInput(...) { ... }
    static expectErrorCodeFromInputSimple(...) { ... }
    static shouldReturnDatabaseErrorWithInput(...) { ... }
    static shouldNotReturnCodeWithInput(...) { ... }
}
```

The tests are self explanatory, however those that end with "DotData" means that the return object from a given function, it is the object property "data" that is being evaluated.

As an example the test option shouldReturnArrayDotData() in comparison to shouldReturnArray() can be looked at:

Listing 13: *GeneralTests.js* DotData example

```
module.exports.GTC = class {
    ...
    static shouldReturnArray(functionCall) {
        it('Should return an array', async function () {
            let returnValue = await functionCall;
            expect(returnValue.message).to.be.an('array');
        });
    }

    static shouldReturnArrayDotData(functionCall) {
        it('Should return an array', async function () {
            let returnValue = await functionCall;
            expect(returnValue.message.data).to.be.an('array');
        });
    }
    ...
}
```

Where the "DotData" looks at the return message object ".message.data" instead of the direct object ".message". All in all there are 84 unit test for the server side, that all pass. Every exported function is unit tested, as so the calls between files and functions are correct. All the functions have at least one unit test, and usually multiple. An example on how many unit tests there can be, and how to use them, can be seen in the following snippet:

Listing 14: *AdminCallsTest.js* addSolution unit test

```
 ...
describe('addSolution function', function () {
    GTC.shouldFailWithNoParameters(ACC.WASC.addSolution());
    GTC.shouldReturnDatabaseErrorWithInput(ACC.WASC.addSolution(-99, 0,
        ""));
    GTC.shouldNotReturnCodeWithInput(ACC.WASC.addSolution(0, [], ""),
        successCodes.AddSolution);
    GTC.shouldNotReturnCodeWithInput(ACC.WASC.addSolution(0, 0, []),
        successCodes.AddSolution);
    GTC.shouldNotReturnCodeWithInput(ACC.WASC.addSolution([], 0, ""),
        successCodes.AddSolution);
});
...
```

4.6 Security

The level of security of the application can be discussed. There are several factors that have been made to the program, especially the server part, to make sure that the user can neither access data that they should not, nor be able to call resources that is disallowed. An overview on how requests from the client are handled can be seen here:

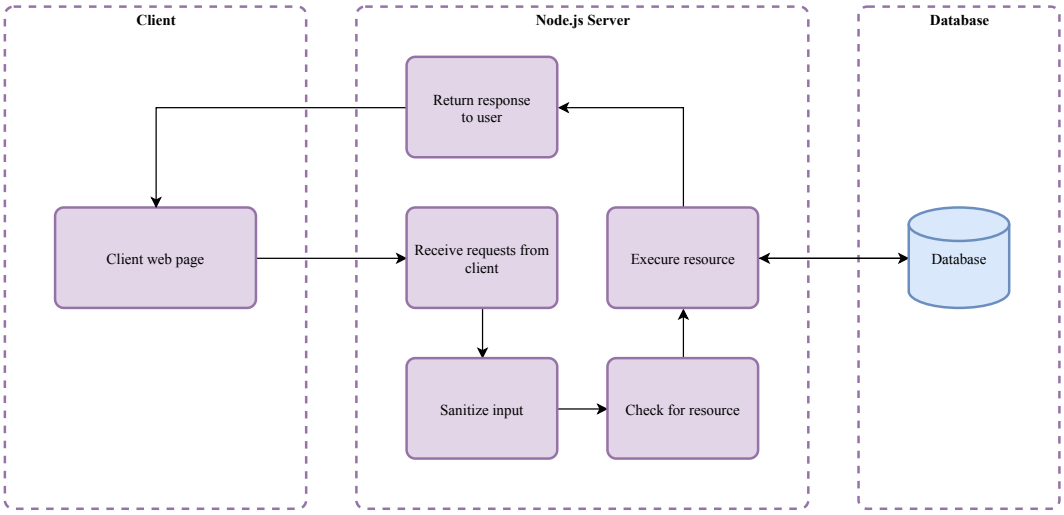


Figure 43: *Security overview, showing the layered security*

It can be seen that the user has no direct access to the database, this is to ensure that the user can not manually send queries to the database, that can potentially be harmful. In addition to that, all request inputs are sanitised, to make sure that the input is correctly formatted, and if not, returns an error. In conclusion, the security level of this project is good, owning to the properties of no direct access to the database and input sanitation.

5 Experimentation

In this chapter, larger experimentation with the program is performed, consisting of larger functionality tests, and not just unit tests. There are some static testing parameters that is used throughout the experimentation, these are descriped below:

Servers

For the testing of the program, servers were made available by Aalborg University. These servers were hosted by ITS, and the webpage can be accessed by the following URL:

`https://dat2c1-3.p2datsw.cs.aau.dk/`[a]

The servers from ITS also included a MySQL database, which is also used in this project.

Physical sensors

Physical sensors were originally going to be used for this, however the COVID-19 lockdown forced the university to close and so we were forced to find other means of getting data. This meant that real-time sensor measurements could not be made. There were however sensors prepared, which could have been used to take measurements, as can be seen in the following image.

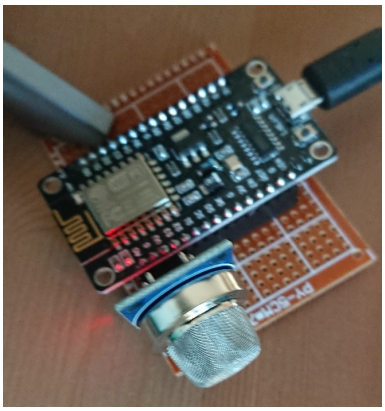


Figure 44: *One of the sensor module that would have been used, this one with a CO_2 sensor on it*

[a] Works as of 18-05-2020

Static dataset

As a backup, a dataset consisting of a lot of sensor values measured over the timespan of two months was used instead. This dataset is static, however the only thing that was done to these measurements was move the timestamp that they had up to current time, since their timestamps was from Oct-Nov 2019. Doing this allowed the data to appear as if it was being gathered and displayed on a live time frame when in fact it was simply old data reused for this purpose. This procedure of getting the data set up to current time was done simply by overwriting the time stamp with a new timestamp which suits our needs for live data better.

5.1 Experiments

5.1.1 Prediction Algorithm accuracy

To test the accuracy of the prediction algorithm, the web interface can be used to look at reoccurring threshold passes. If Room B is selected, firstly at the date 16-03-2020 at 05:00:00, only live data can be seen, this is because it is the oldest data there is for this room.

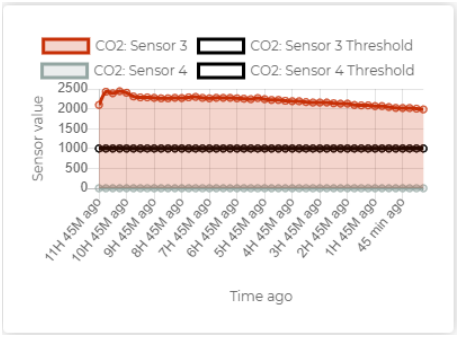
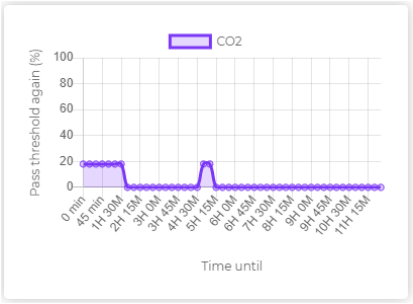
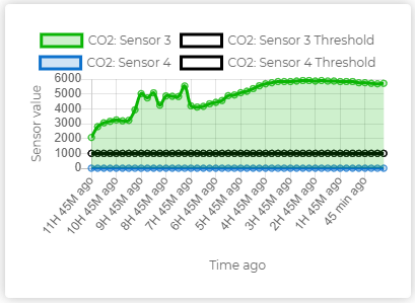


Figure 45: *16-03-2020 live data*

Here it can be seen that the sensor value for sensor 3 is way past its threshold. A baseline has now been set, that the prediction algorithm can work with. If the next week is then selected, at 23-03-2020 at the same time, there are now both live data and predictions. The predictions are now due to the fact that there is some historical data to base it on, however it is only one week, so the chance is still fairly low for it occurring again. The live data however can again be seen to have passed the threshold.



(a) *23-03-2020 prediction data*



(b) *23-03-2020 live data*

If the next week is then selected again, at 30-03-2020 same time, it can be seen that there is no live data. The reason for this is that there are some holes in the dataset that is being used, this does however represent a real scenario, where a sensor could break down or not send data for some reason. It should however be noted that the predicted chance of the threshold being passed again is higher now, since it is now two times in the last two mondays that the threshold has been passed. Weighted, this corresponds to a 34% chance of it happening again:

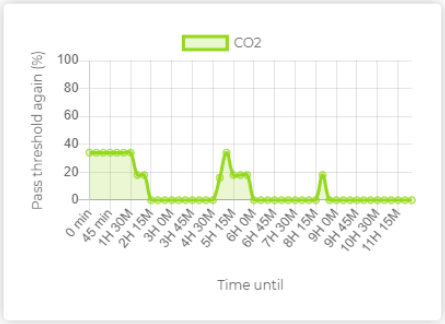
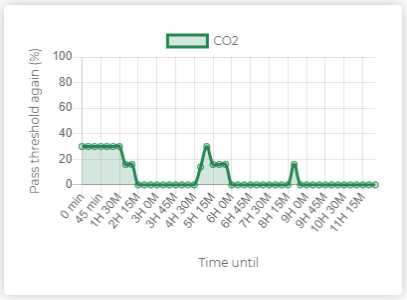
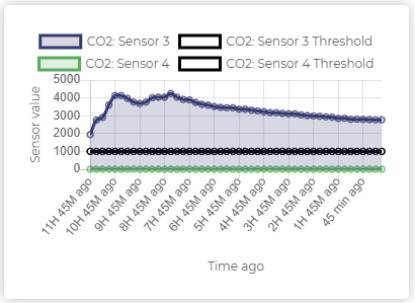


Figure 47: *30-03-2020 prediction data*

The next week at 06-04-2020 same time, the live data has again passed its threshold. It can however be seen that the predicted chance of it passing again is lowered. The reason for this is that the last week had no data, and the assumption is then made that the last mondays values were all 0.

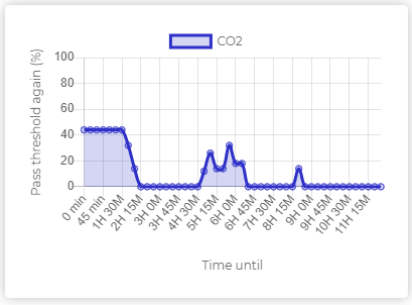


(a) *06-04-2020 prediction data*

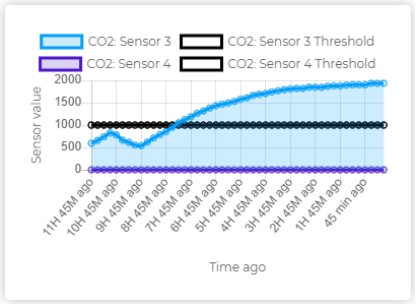


(b) *06-04-2020 live data*

The next week at 13-04-2020 same time, the live data is once more passing its threshold. Now it can be seen however that the predicted chance of it happening again has increased to a 44% chance:

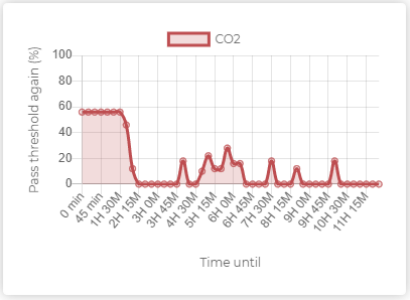


(a) *13-04-2020 prediction data*

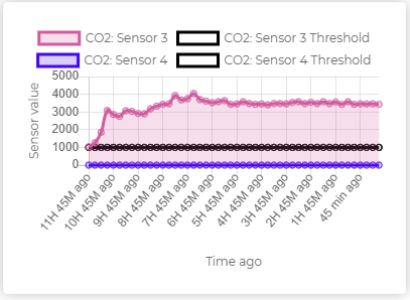


(b) *13-04-2020 live data*

The next week is the last week with data, at 20-04-2020 same time, the live data is again passing its threshold, and now the predicted chance of it happening again is all the way up to 56%:



(a) *20-04-2020 prediction data*



(b) *20-04-2020 live data*

The end of the dataset has now been reached, however if a look is taken on the next week, at 27-04-2020 same time, there is of course no live data, however it is predicted that there is a 66% chance of it happening again. This is a fair assumption to make, since at all the other live data that was available, the threshold was passed.

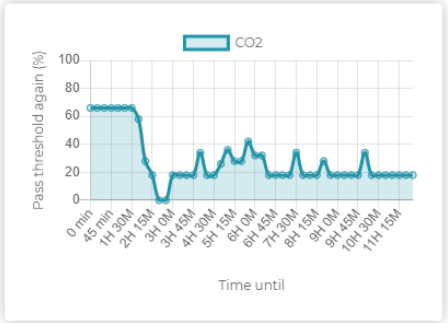


Figure 51: *27-04-2020 prediction data*

From here on out the predicted chances continue to fall the higher a week is looked into the future. This is, of course, because there is no more data, and that older data has a lower weight than newer data. The next week again at 04-05-2020 same time, it can now be seen that the predicted chance have fallen slightly.

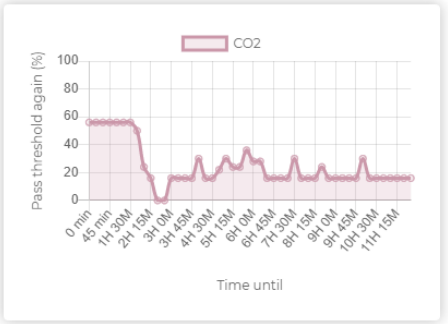


Figure 52: *04-05-2020 prediction data*

It can then be concluded that the algorithm is capable of predicting a reoccurring event, where the threshold of a sensor is passed. It can also be concluded that a larger dataset, and preferably a "live" dataset, would continue to increase the accuracy of the prediction algorithm.

5.1.2 Warning and Solution Display

One of the core functionalities of the program is to give a warning to the user, concerning developing bad IAQ. This functionality can be tested easily, since going onto the website, there will almost no matter what be displayed some warnings and solutions. An example can be seen below:

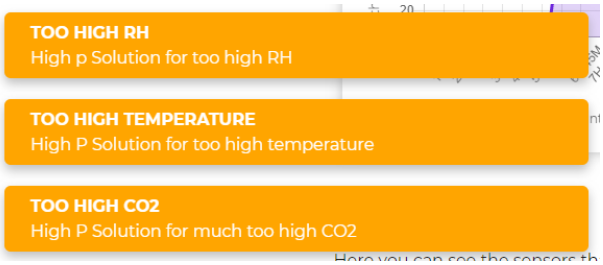


Figure 53: *Warnings and solutions from the 12-05-2020 at 08:57*

It can be seen that the warning priority system returns only high priority solutions. The reason for this is that the system only takes time into account, and not the predicted chance. This introduces a high level of inaccuracy to the warning system, however it should be noted that implementing a system to take the predicted chance into account as well would not be difficult to make.

For the purpose of this program, however, it can be tested if the priority system works. The priority times are set in the database at:

- **High Priority**: Less than 10 minutes
- **Medium Priority**: Less than 30 minutes
- **Low Priority**: Less than 60 minutes

These values can be changed anytime in the database. Next step is to find a date where a predicted sensor value reaches 0. An example can be found in Room B, at 07-05-2020 at 12:00:00, where the CO_2 reaches 0:

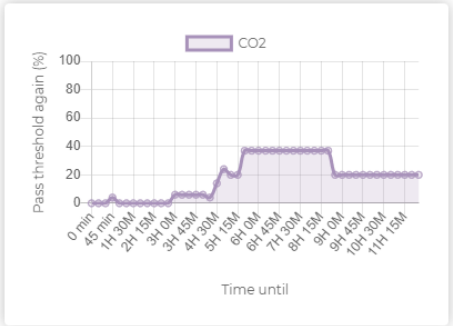


Figure 54: *The predicted chance of CO_2 passing its threshold again at the date 07-05-2020 at 12:00:00*

Here it can be see that the next point, that is larger than 0, is in 45 minutes, this should trigger a low priority warning, since it is lower than 60 minutes and higher than 30 minutes. It can be seen that the low priority warning gets triggered, in the following figure:

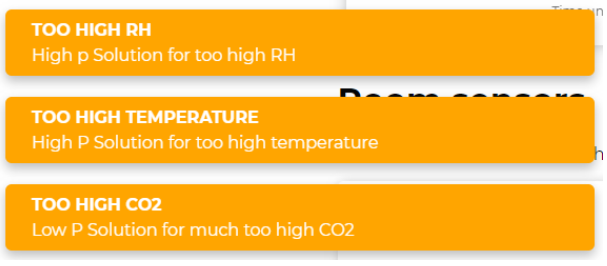


Figure 55: *Warnings and solution for the date 07-05-2020 at 12:00:00*

To trigger the medium priority warning, the time have to be set 30 min ahead, so that there is 15 minutes until the predicted threshold pass gets larger than 0. The result can be seen in the following figure:

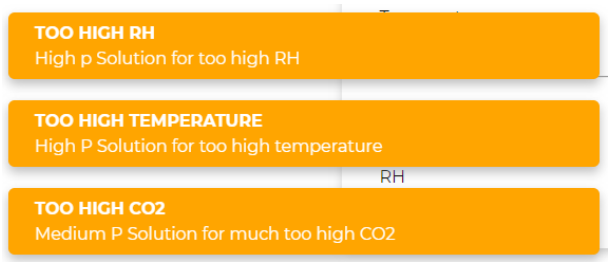


Figure 56: *Warnings and solution for the date 07-05-2020 at 12:30:00*

If the time was pushed forward by a further 15 min, then the high priority solution would be triggered, since there then are less than 10 minutes to the predicted threshold is passed. It can then be concluded that the warning and solution priority system works, even with its deficiencies.

6 Discussion

To summarise on what has been done throughout the project it is clear to say that the requirements have been met, and a platform created. The models have been followed to completion and then some. This means that the project now includes a website, a server and a database. As the project is nearing its conclusion phase, a deeper look can be taken into the meaning of the results which were found, and what more may be done. The experimentation section shows that the program works, however with its deficiencies. This shows that, in theory, the program could be deployed to a school, and it could help give the students and teachers a preemptive warning on bad IAQ.

These results can not only be used to help students and teachers directly, but also indirectly by giving them a better argument for old AC system to be replaced or an old one to be upgraded. This means that the solution also extends up to the administration of the school or institution by giving them concrete data, that they can use to give themselves a realistic view of how the IAQ is in their school.

As said there are some limitations to this solution. One of the major issues is the fact that the warning system is only based on time until, and not percentage chance of a threshold pass. This can be remedied by modifying the program, to also take percentages into account. Another point of limitation, is the fact that the prediction algorithm is a fairly simple one, and only accounts for historic data, and expects the times of bad IAQ to be persistent. As an example the algorithm does not give a proper prediction if a room is only occupied every other week, since it would also take the weeks that there are no one in the room as a data point.

Some improvement can thus be made, such as better warning system and prediction algorithm. Another point that can also help improve the accuracy of the prediction algorithm is making a system, that can auto set the thresholds for the sensors. Currently the thresholds are set statically, based on values found in section 2.1. There are however some parameters that is looked over when doing that, mainly the fact that the sensors are not calibrated, and that the position of the sensors can also have a large impact on its readings. E.g. a sensor sitting close to the floor, would get very different results compared to one on the ceiling.

7 Conclusion

To conclude upon the project it is seen as being successfully completed in correlation to the requirements which were set in the beginning of the project. The final product has been extended past the initial model with features such as depicting live data and including real sensors, although real sensors could in fact not be implemented as COVID-19 has seen to it that the university be shut down. Furthermore, successful tests, on both unit level as well as functionality level have been carried out to completion, and proves that the program works as intended.

It has also been shown that the program is of high quality, reasoned by the fact that it is unit tested. The program is also shown to have a high level of security, owning to the fact that all communication to the database, have to go through an API, that removed the possibility of intrusive SQL queries being executed on the database.

There are also some aspects of the program which may be improved upon. For example the prediction algorithm and some needed modification to the warning and solution selection algorithm. The program also suffers from the unavailability of physical sensors, that could have provided some interesting insight to the real time use of the program.

To sum it all up, a program has been made which can take in sensor data from multiple units, make a prediction on future readings and finally warn a user about an impeding bad IAQ.

Bibliography

[1] MOHIEDDINE BENAMMAR et al. "Real-Time Indoor Air Quality Monitoring Through Wireless Sensor Network". In: *International Journal of Internet of Things and Web Services* 2 (), pp. 7–13. URL: https://www.iaras.org/iaras/journals/caijitws/real-time-indoor-air-quality-monitoring-through-wireless-sensor-network. (accessed: 20.02.2020).

[2] Lia Chatzidiakou, Dejan Mumovic, and Alex James Summerfield. "What do we know about indoor air quality in school classrooms? A critical review of the literature". In: *Intelligent Buildings International* 4.4 (2012), pp. 228–259. DOI: 10.1080/17508975.2012.725530. eprint: https://doi.org/10.1080/17508975.2012.725530. URL: https://doi.org/10.1080/17508975.2012.725530. (accessed: 14.02.2020).

[3] Digi-Key. *SEN-14348*. URL: https://www.digikey.dk/product-detail/en/sparkfun-electronics/SEN-14348/1568-1706-ND/7652735?utm_adgroup=Evaluation%5C%20Boards%5C%20-%5C%20Expansion%5C%20Boards%5C%2C%5C%20Daughter%5C%20Cards&utm_source=google&utm_medium=cpc&utm_campaign=Shopping_Product_Development%5C%20Boards%5C%2C%5C%20Kits%5C%2C%5C%20Programmers&utm_term=&productid=7652735&gclid=CjwKCAjw7-P1BRA2EiwAXoPWA6CgOpBmlqoTsra-LJWcyiKZzw0kzYmTAaqYuFFmgx2c0rzGjXoqNBoCLDIQAvD_BwE.

[4] Elfa Distrelec. *BME680*. URL: https://www.elfadistrelec.dk/da/adafruit-bme680-sensor-5v-adafruit-3660/p/30129236?channel=b2c&price_gs=195&source=googleps&ext_cid=shgooaqdkda-blcss&kw=%5C%7Bkeyword%5C%7D&ext_cid=shgooaqdkda-P-CSS-Shopping-MainCampaign-Other&gclid=CjwKCAjw7-P1BRA2EiwAXoPWA8fPiWI8TVwHemyt2zAgUXIVYzaUZ5PpLb8gCQ_M0xWUfijZckJngBoCic4QAvD_BwE.

[5] Let Elektronik. *CCS811*. URL: https://let-elektronik.dk/shop/1500-biometri--gas/14193--air-quality-breakout---ccs811/?gclid=CjwKCAjw7-P1BRA2EiwAXoPWA2mV6Yqh-w-7PXdCPDubiYeTuTw1IEUZQbnXGjC7qRCV1TW8idDyPhoCVPIQAvD_BwE.

[6] William J. Fisk et al. "Is CO2 an Indoor Pollutant? Higher Levels of CO2 May Diminish Decision Making Performance". In: *ASHRAE JOURNAL* 55.3 (Mar. 2013). DOI: accessed:14.02.2020. URL: https://www.osti.gov/servlets/purl/1171812.

[7] Désirée Gavhed and Lena Klasson. *Perceived problems and discomfort at low air humidity among office workers*. Environmental Ergonomics - The Ergonomics of Human Comfort, Health, and Performance in the Thermal Environment. Elsevier Ltd., 2005. ISBN: 9780080455709. (accessed: 22.05.2020).

[8] Loes Geelen and A. Zijden. "Healthy learning at school!" In: *The European Journal of Public Health* 16 (Jan. 2006), pp. 44–44. URL: https://www.researchgate.net/publication/295275585_Healthy_learning_at_school.

[9] Jie He et al. "A high precise E-nose for daily indoor air quality monitoring in living environment". In: *Integration* 58 (2017), pp. 286–294. URL: https://www.sciencedirect.com/science/article/pii/S0167926016301985. (accessed: 20.02.2020).

[10] C. Huizenga et al. "Air Quality and Thermal Comfort in Office Buildings: Results of a Large Indoor Environmental Quality Survey". In: *Proceedings of Healthy Buildings* 3 (2006), pp. 393–397. URL: https://escholarship.org/uc/item/7897g2f8. (accessed: 18.05.2020).

[11] Michele R. Kinshella et al. "Perceptions of Indoor Air Quality Associated with Ventilation System Types in Elementary Schools". In: *Applied Occupational and Environmental Hygiene* 16.10 (2001). PMID: 11599544, pp. 952–960. DOI: 10.1080/104732201300367209. eprint: https://doi.org/10.1080/104732201300367209. URL: https://doi.org/10.1080/104732201300367209. (accessed: 18.03.2020).

[12] S. M. Saad et al. "Indoor air quality monitoring system using wireless sensor network (WSN) with web interface". In: *2013 International Conference on Electrical, Electronics and System Engineering (ICEESE)*. 2013. URL: https://ieeexplore.ieee.org/document/6895043.

[13] Usha Satish et al. "Is CO2 an Indoor Pollutant? Direct Effects of Low-to-Moderate CO2 Concentrations on Human Decision-Making Performance". In: *Environmental health perspectives* 120.12 (2012). URL: https://ehp.niehs.nih.gov/doi/full/10.1289/ehp.1104789. (accessed: 17.02.2020).

[14] Wikipedia. *ESP8826*. URL: https://en.wikipedia.org/wiki/NodeMCU. (accessed: 25.02.2020).

[15] Wikipedia. *Query String*. URL: https://en.wikipedia.org/wiki/Query_string. (accessed: 26.03.2020).

8 Appendix

8.1 Return Codes

All the error codes:

Code name	Value	Code name	Value
NoParameters	401	TargetIsDefaultID	402
PriorityOutsideRange	403	OutputNotAnArray	404
InputNotAnArray	405	NoSensorTypes	406
IDDoesNotExist	407	DatabaseError	408
InputNotAString	409	EmptyString	410
WrongInputCredentials	411	ResourceNotFound	412

And all the success codes:

Code name	Value	Code name	Value
AddWarning	201	RemoveWarning	202
UpdateWarning	203	AddSolution	204
RemoveSolution	205	UpdateSolution	206
AddExistingSolution	207	RemoveSolutionRef	208
AddRoom	209	RemoveRoom	210
UpdateRoom	211	AddSensor	212
RemoveSensor	213	AddExistingSensor	214
RemoveSensorRef	215	AddSensorType	216
AddExistingSensorType	217	RemoveSensorType	218
RemoveSensorTypeRef	219	UpdateSensorTypeThreshold	220
InsertSensorValue	221	GotWarningsAndSoluton	222
GotSimpleSensorInfo	223	GotPredictions	224
CredentialsCorrect	225	ReadOpenAPIFile	226
GotAllSensorTypes	227	GotAllSolutions	228
GotAllSensors	229	UpdateSensor	230
GotLiveData	231	GotAllSensorTypeValues	232
GotPriorityName	233	GotAllWarningsAndSolutions	234
GotAllPriorities	235		