VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

NHAT DANG MIN

THE UPGRADE AND DESIGN OF REMOTE MONITOR AND CONTROL FOR CLIMATE CHAMBER

Thesis instructions accepted 19/02/2017

Technology and Telecommunication
2018

VAASA UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Dang Min
Title	The Upgrade and Design of Remote Monitor and Control for Climate Chamber
Year	2018
Language	English
Pages	62
Name of Supervisor	Jani Ahvonen

The climate chamber, or the temperature and humidity chamber, is a device located in Technobothnia Education and Research Center, used for testing the endurance of electrical devices by stimulating different temperature and humidity conditions. Inside the chamber, the dedicated system consists of a PC running a control software that is responsible for climate simulation.

Prior to previous study and design of the climate chamber controlling software by Simachew Tibebu and the remote monitoring system developed by Habtamu Ashengo, the Technobothnia Education and Research Center grew their interest in a new system, which is capable to control and monitor the climate chamber remotely.

Based on the need for a new system, the thesis was projected to upgrade the previous version of the system and enable the possibility to control the chamber from anywhere. The main objective of this project was to design a new system, running on the Linux-based devices, such as Raspberry Pi or a Linux computer, to operate as a server with the ability to monitor the chamber, to store information related to the chamber and to be able to control the system via web service.

The requirements of the system were determined by the desired functionalities of the new system along with the previous study of the chamber's operations. These following functions were the results:

- A running server on a Linux-based device connected directly via COM port to the chamber
- A user interface with capabilities to manage programs, to manage control parameters (PID-control), to monitor the chamber status and to operate the program remotely.

The system was developed in JavaScript/NodeJS on Linux development environment, with the server powered by HapiJS and the user interface powered by the MithrilJS framework.

Due to the delay in development of this project, the goals were not fully achieved. With limited access to the chamber during the summer, the testing process was conducted with chamber simulating program, providing basic procedures, for instance, monitoring

the chamber state and information, managing chamber’s programs and control parameters without validation. Though this project was approached with new technology and different method than the previous two, testing would be considered as unnecessary at this stage as other students can continue with this project later.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

PC	**P**ersonal **C**omputer
PID	**P**roportional, **I**ntegral, **D**erivative
JS	**J**ava**S**cript
REST	**Re**presentational **S**tate **T**ransfer
NPM	**N**ode **P**ackage **M**anager
ECMA	**E**uropean **C**omputer **M**anufacturers **A**ssociation
SPA	**S**ingle **P**age **A**pplication
API	**A**pplication **P**rogram **I**nterface
HTTP	**H**yper **T**ext **T**ransfer **P**rotocol
ODM	**O**bject **D**ocument **M**apper
CSS	**C**ascading **S**tyle **S**heet
SASS	**S**yntactically **A**wesome **S**tyle **S**heets
OS	**O**perating **S**ystem
ID	**Id**entifier
MVC	**M**odel **V**iew **C**ontroller
DB	**D**ata**b**ase
JSON	**J**ava**S**cript **O**bject **N**otation
URL	**U**niform **R**esource **L**ocator
DOM	**D**ocument **O**bject **M**odel

LIST OF FIGURES

LIST OF TABLES

LIST OF SNIPPETS

1 INTRODUCTION

The climate chamber, or so-called environment chamber, is a device replicating different environmental conditions to test the endurance of the electrical devices or industrial products under such environment.

The climate chamber, used by Electrical Department of Technobothnia Education and Research Center, focuses on two parameters: *temperature* and *humidity*. The chamber consists of several components:

- A test area, an environment stimulated area where test object is put in.
- Sensors, a series of sensors placed in the system to collect the environment parameters inside the chamber.
- Peripherals, various devices stimulating the desired environment condition, for instance, a heater, a cooler, and a humidifier.
- A control box, a device to control the state of peripheral devices.
- A control PC, a dedicated computing unit connected directly to a control box that runs a single program to monitor and to control the chamber.

The objective of this project is to communicate with the control box through the server, with the end-user controlling the chamber via the user-interface. Figure-1 below shows the simple design of the system.

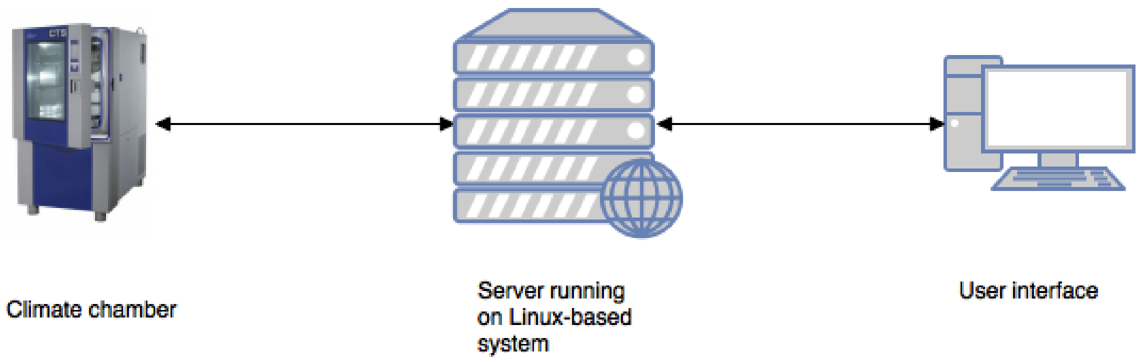


Figure 1. Software structure.

Figure 2 explains the connection between the climate chamber, the old interface and the new system.

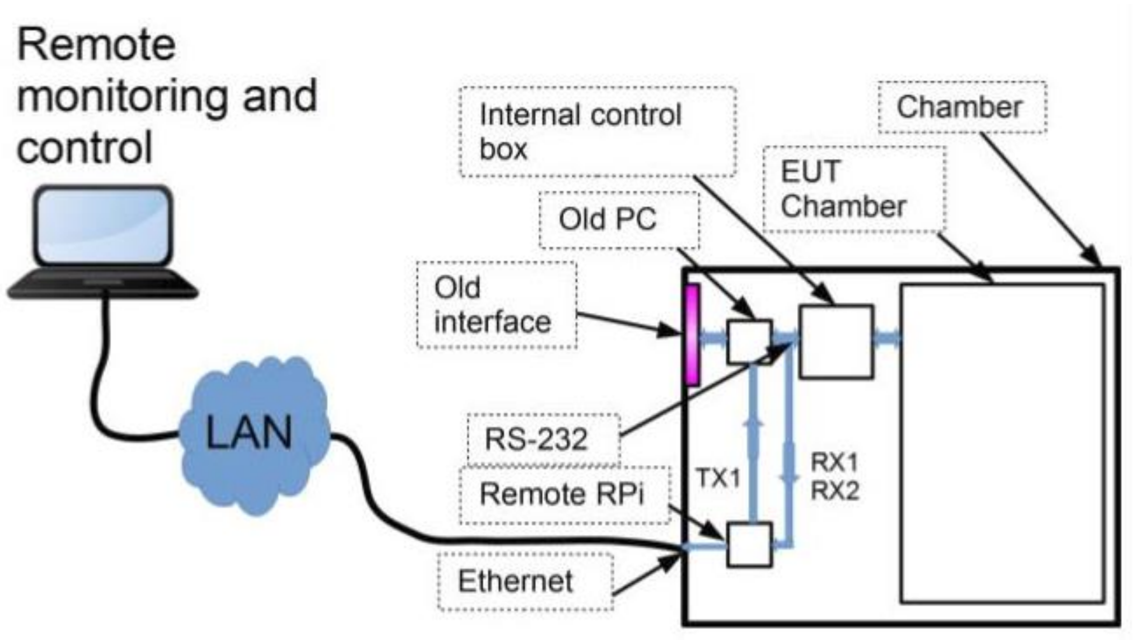


Figure 2. Block diagram explaining the connection between the new system and the chamber

1.1 Background

The chamber provides the software running in Windows environment. The software accepts user inputs as parameters for controlling the chamber. The environmental test program is a set of different stages of testing, called steps. Each step of the program contains desired temperature, humidity and related parameters for testing purposes. Besides the program management, the software offers PID controlling parameters management, where the user can manually adjust the PID parameters.

On the chamber monitor, the real-time value of temperature and humidity is displayed. In addition, when the testing program is run, the information related to on/off state of the peripherals are showed on screen.

When a testing program is running, the real-time value of the temperature and humidity are reported back to the control box, then sent to control PC to calculate and give the decision on the state of the peripherals.

The control PC stores the information of the run program, including the changes in temperature and humidity over time, then uses the information to display a graph of temperature and humidity as the function of time.

Besides the monitor, the chamber has a small keyboard including a number pad and various functional keys, a floppy disk drive and an additional serial communicating port. The floppy disks are used as the medium of storage of the chamber.

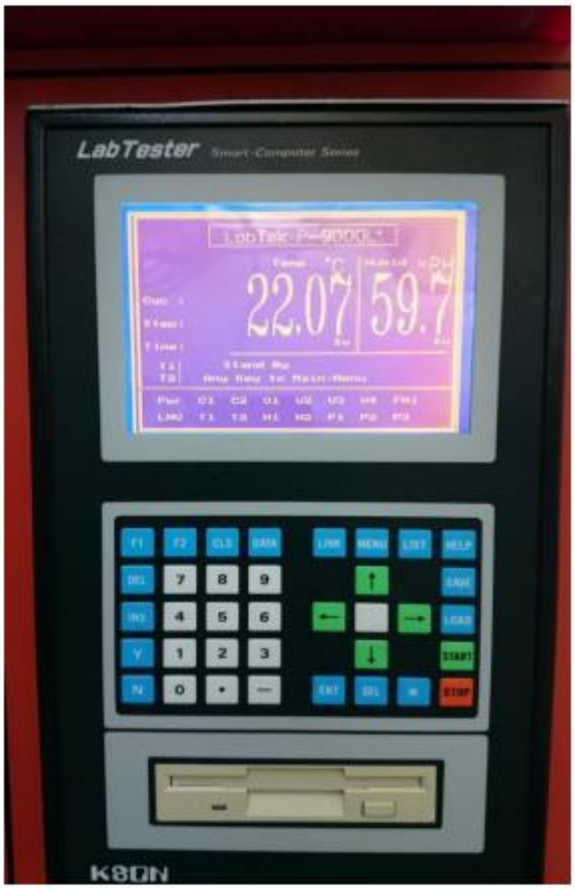


Figure 3. Climate chamber monitor, keyboard and floppy drive.

1.2 Objective

The focus of this project is to design a web application that enables the user to interact with the chamber from everywhere with an Internet connection. The application consists of two components: the web server and the user interface. The application is installed on a Linux-based system, which connects directly to the chamber via the RS232 communication port. The server is responsible for controlling the chamber by replicating the controlling algorithm of the provided software, while the user interface is responsible for the communication between the user and the server.

2 RELEVANT TECHNOLOGIES

The relevant technologies and tools used for developing the application are documented in this chapter.

2.1 JAVASCRIPT

JavaScript is one of the most popular programming languages for developing a web application. It is a high-level scripting language that allows developers to handle responsiveness of the web page.

2.1.1 NodeJS

NodeJS is an open source framework which was first introduced by Ryan Dahl in 2009. NodeJS provides a server-side solution with JavaScript, which means that the web application can be developed with JavaScript only. In addition, NodeJS is suitable for developing a web application, REST services and the Internet of Things. The advantages of NodeJS are:

- NodeJS simplifies the development process by eliminating the need of using two different languages for developing server-side and client-side separately. Moreover, it enables the ability to reuse the components and resources when developing an application with JavaScript.
- NodeJS is event-driven, which means that when an event is triggered, its function will start executing.
- NodeJS is single threaded. For a normal web application, as the client sends a request to the server, a new thread is created to handle the request and response to the client. However, in NodeJS, when a request sent to the server, it is put into an event queue and sequently processed according to the event queue.
- NodeJS emphasizes the non-blocking programming idea, so-called asynchronous programming. Blocking programming is a term referred to a sequence of functions triggered in order, one waits for the other to finish its task, while in non-blocking programming, all functions are called at once, though which function finishing its task first, will give return value first, no matter the function order.

In addition to the advantages of NodeJS, it offers far more than that. With NodeJS in particular and JavaScript in general, not only pure JavaScript is applicable, but any other script code, for instance, CoffeeJS, ECMA script, TypeScript, can be transpiled into JavaScript. JavaScript popularizes the idea of functional programming, which helps to reduce the number of code line significantly.

2.1.2 NPM

NPM – Node Package Manager – as indicated in its name, it is a package manager that goes along with NodeJS. With the non-stop development of NodeJS, thousands of different modules and packages were created and shared among developers, NPM plays its role as a supportive software that helps to install necessary packages for application, run scripts and share code with any other npm user.

In an application, npm indicates core dependency packages in package.json file. Therefore, when the application is published, it can be installed with the same packages on any other platforms.

2.1.3 Webpack

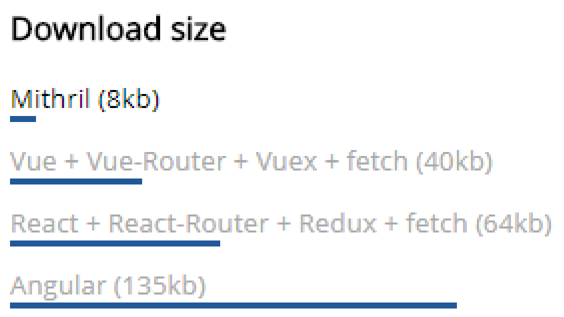
Webpack is a JavaScript bundler that gathers all the assets, including JavaScript code, images, fonts and styling into a dependency graph. Webpack helps to transform the alternative JavaScript into globally used JavaScript that can be executed in the browser. Moreover, it secures the possibilities of missing assets, since the application cannot be deployed without missing assets, for instance, images or styling files.

2.1.4 HapiJS

HapiJS is a JavaScript framework created by mobile web team at Walmart Labs. HapiJS is best-suited for building web services such as JSON APIs, HTTP proxies, and single-page/multi-page web application. HapiJS helps simplify the process of building web infrastructure, and provides developers with a focus on building web application functions.

2.1.5 MithrilJS

MithrilJS is a client-side JavaScript framework, which is suitable for building single page application (SPA). Its advantages are lying in its size and speed of performance.



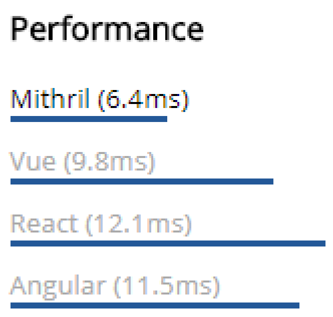


Figure 4. MithrilJS compared to other frameworks

2.1.6 Node Serialport

Node Serialport is an npm package that provides support for serial communication among devices.

2.1.7 Mongoose

Mongoose is an npm package, playing a role as a connection between an application and MongoDB. In other words, it is an Object Document Mapper. Mongoose provides methods for typecasting, validation, query building and business logic hooks and more. /Mongoose introduction/

2.2 Stylus

Stylus is an alternative syntax for writing CSS. It is influenced by SASS and LESS and was first introduced in 2015.

2.3 MongoDB

MongoDB is an open-source cross-platform database program. It stores data in a form of JSON-like documents. Data in MongoDB can be flexible, meaning that fields can be varied between documents and its structure can be changed over time. The advantages when using MongoDB in this application is that its document models can be mapped to the objects without any conflicts.

2.4 Atom

Atom is an open-source and free text editor developed by GitHub. Atom is platform independent, which can be operated on Linux, MacOS, and Windows. Atom is embedded Git Control.

3 UNDERSTANDING THE OLD SYSTEM

This project's goal is to upgrade and design a new system providing the remote control and monitor for the climate chamber. Therefore, the study related to the machine must be carried out to decide the number of functions, the number of event handlers and the important data related to the chamber as well as the communication between the system and the chamber.

Below are the entities that are the spiral of the old system which needs to be retained in the new application.

3.1 Test programs

Test programs are sets of environmental tests that carried out by the climate chamber. Each program contains various steps and the number of cycles, each of the steps includes information relating to testing purpose.

3.2 Steps

Steps are stages of an individual environmental test. Parameters are set in each step. The parameters include set values, which are temperature and humidity, the time of which step has to be done in and wait for status, that is a piece of information indicates an action in case step cannot be done in the assigned time. For each program, the steps are processed one after another and repeat (if there is a cycle left) with set temperature and humidity, providing the climate changes inside the chamber.

3.3 Cycle

The cycle is the number of the set that the steps are repeated. For instance, if the program is set to run for 3 cycles, then the steps are processed and repeated again until it finishes 3 sets of steps.

3.4 Components of the climate chamber.

Besides the control box that oversees the flow of the chamber, there are other elements that play important roles in the machine. The sensors, placed inside the chamber, collect the temperature and humidity then report back to the control box, later used for the PID control parameters calculation. The cooler, heater and humidifier are components to be controlled by the control box to replicate the desired environment inside the chamber. Valves are used for the coolers to control the intake and outtake from the cooler to the chamber.

3.5 Communication

The communication between the chamber and the control PC is conducted via RS232 cable, called serial communication. The attributes of the protocol include a baud rate of 9600, 8 bits of data, stop bit enabled and no parity is set.

The message contains the following components:

Start bit – STX	Start of the message – 02 in hexadecimal
ID1	The first ID – value ranges from 30 to 39 in hexadecimal.
ID2	The second ID – value ranges from 40 to 4F in hexadecimal
COMMAND	The control command sent to the chamber
EXT	The end mark of the control command – 03 in hexadecimal.
CHKDSK	Checksum
END	End of the message – 0D in hexadecimal.

Table 1. Message components.

The communication has 4 major types of messages, classified by their ID2: A (41 HEX), B (42 HEX), I (49 HEX) and O (4F HEX). These 4 types are passed back and forth as a pair for the communication. The sequence of the communication is:

- First, the O message is sent from the control box and waits for the response from the chamber, which is the ACK message – 06 in hexadecimal
- Then the I message is sent out and waits for the I response message
- The A message is dispatched and waits for the A response, containing the temperatures and humidity of the chamber.
- Finally, the B message is delivered, and the chamber reply with B response to turn off the communication.

The messages A, B and I from the control PC are static, while the message O is dynamic, containing the control message for the control box. The message O has 26 bytes and contains control information. Besides 6 bytes for start bit, end bit, ID1, ID2, EXT and CHKDSK, the remaining 20 bytes provide control to the control box. Starting from 13th byte to 22nd byte, the message is constructed with this component:

0010 $H_1H_2T_1T_2$	H1	State of humidifier 1
	H2	State of humidifier 2
	T1	State of heater 1
	T2	State of heater 2
0010 $P_3P_2L_1$LNV	P3	Time signal 3
	P2	Time signal 2
	P1	Time signal 1
	LNV	State of extra cooler-gas insertion valve
0010 $00C_1V_4$	0	
	0	
	C1	State of compressor 1
	V4	State of refrigerant valve 4
0010 $V_3V_2/C_2V_1FN_1$	V3	State of refrigerant valve 3
	V2/C2	State of refrigerant valve 2 and compressor 2
	V1	State of refrigerant valve 1
	FN1	State of condenser cooling fan
0010 XXXX	Humidifier power 1	Power of the humidifier
0010 XXXX	Humidifier power 2	Power of the humidifier
0010 XXXX	Heater power 1	Power of the heater
0010 XXXX	Heater power 2	Power of the heater

Table 2. Message contents

3.6 Control method

The control mechanism for the chamber is implemented inside the control PC as PID control theory. The PID control, with proportional – integral – derivative as abbreviation respectively, is a control mechanism that is based on the feedback loop. The value of the chamber is measured via a sensor, reported back to the control PC. The difference between the set value and the measured value is then calculated and applied to calculate the control output. The control output depends on not only the differences between measured value and set value but also the PID control parameter, set by the user. The purpose of the PID control is to bring the measured value to be approximate to the set value.

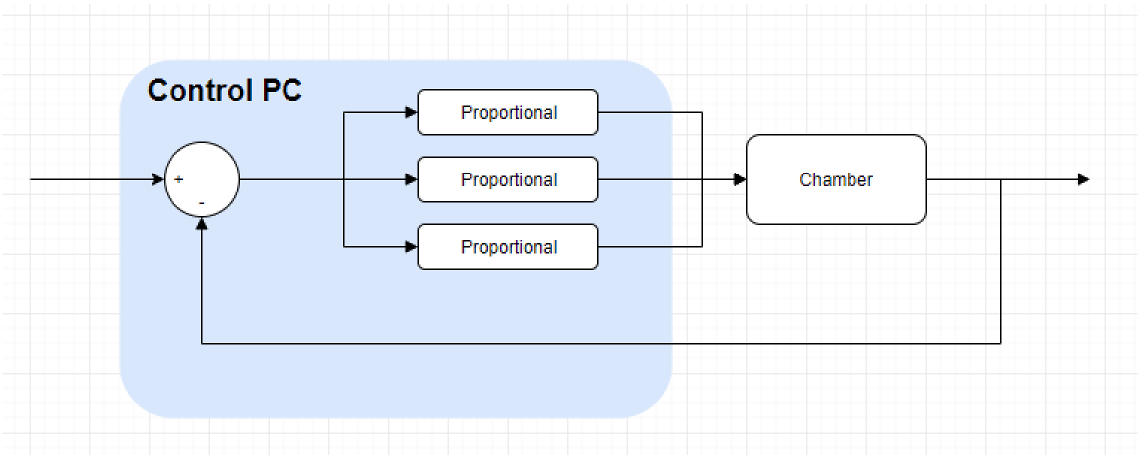


Figure 5. PID control model.

4 SOFTWARE DESCRIPTION

The goal of this project to build a climate chamber remote monitoring and controlling system that can be accessed anywhere with the Internet connection. The application is divided into 2 separated subsystems, the server, and the client-side web page. The server system is operated on any Linux-powered system, for instance, Raspberry Pi or Linux computer. The server-side system is responsible for handling requests from the client side, communicate with the database and the climate chamber. The client side provides a user interface to access and control the system via a browser.

The server must be able to communicate constantly with the control box of the climate chamber in order to collect the temperature and humidity from the chamber and to be able to send control commands to the control box to control the system to archive the desired test program goals. In addition, the server has to be able to do the calculation and provide control box with a decision based on the state of the chamber.

On the other hand, the client web page must be able to communicate with the server via a web socket to receive the temperature and humidity of the chamber, the information related to operating test programs. Moreover, the client-side must provide the user with the capability to control the test program resources, has access to view history of running program and be able to plot a graph for the user to observe for further studies.

4.1 Project constraints

The communication between the software and the climate chamber has been defined by the old system and previous studies, therefore, this project will inherit that characteristic of the previous project, providing further studies since the two projects developed in different programming languages.

On the other hand, this project was developed during the summertime, so it lacks of interaction with the real climate chamber, and the only available source is the simulated software which replicated the chamber. Therefore, further studies for this project must be carried out in order to have a final production.

4.2 Graphical user interface design

This application was aimed to be a single page application. The interface was provided by the server with a simple template, and the remaining components were generated dynamically by MithrilJS. The components included: the tab navigation, the serial connection status, the display of climate chamber state, the program window with step management, the PID window and the history window.

These components are developed based on MVC – Model View Controller. The interaction with the interface is implemented using JavaScript, as it was event-driven language.

4.2.1 General design approach

The general approach for this single page application interface is that the template is provided by the server when accessing the page from the browser, then the components are generated and connected on initial function and render to the page. The general look of the page contains:

- Static title
- Left sidebar: includes the tab header and the serial connection status.
- Main view: this module includes other subviews, for instance, climate chamber state module, program module, PID module, and the history module.

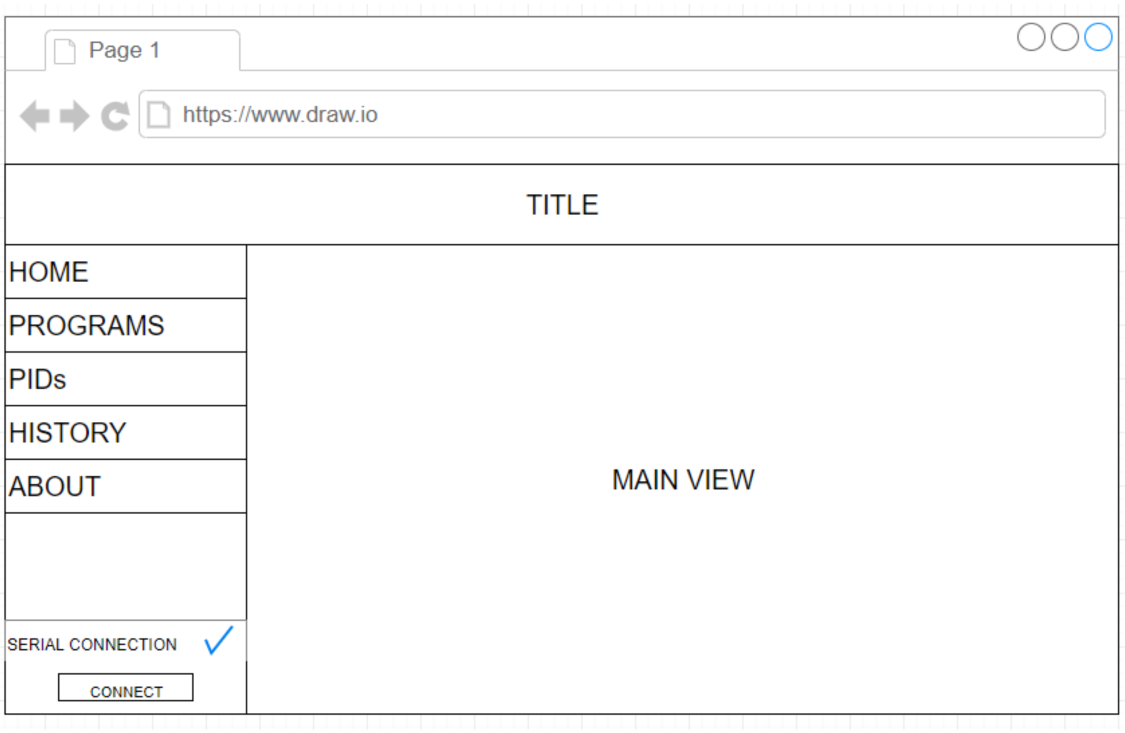


Figure 6. User interface design.

4.2.1.1 The header

The header is located on top of the page, indicates the title or the name of the application.

4.2.1.2 The sidebar

The sidebar is placed on the left side of the page, consists of 2 subcomponents: the tab header and the serial status module.

- **The tab header**

The tab header consists of the header, which can be clicked to switch the main view. Each header click will change the URL of the page.

- **The serial status module**

The serial status module is at the bottom of the sidebar, consists of 2 parts:

- Serial status: indicates the state of the serial connection between the server and the climate chamber.
- Connect button: button to signal the server to open the connection to the climate chamber, this button is disabled in case serial connection has been established

4.2.1.3 The main view

The main view is the container for 4 subcomponents: the program view, the display of chamber information, the PID management view and the history view. These 4 subcomponents switch according to the selected header.

- **The home window**

The home view consists of the running program information, which usually empty when no program is run, the temperature and humidity of the chamber, displayed as meters, and the below set of squares, indicated states of the climate chamber components, for instance, heaters, humidifiers, …

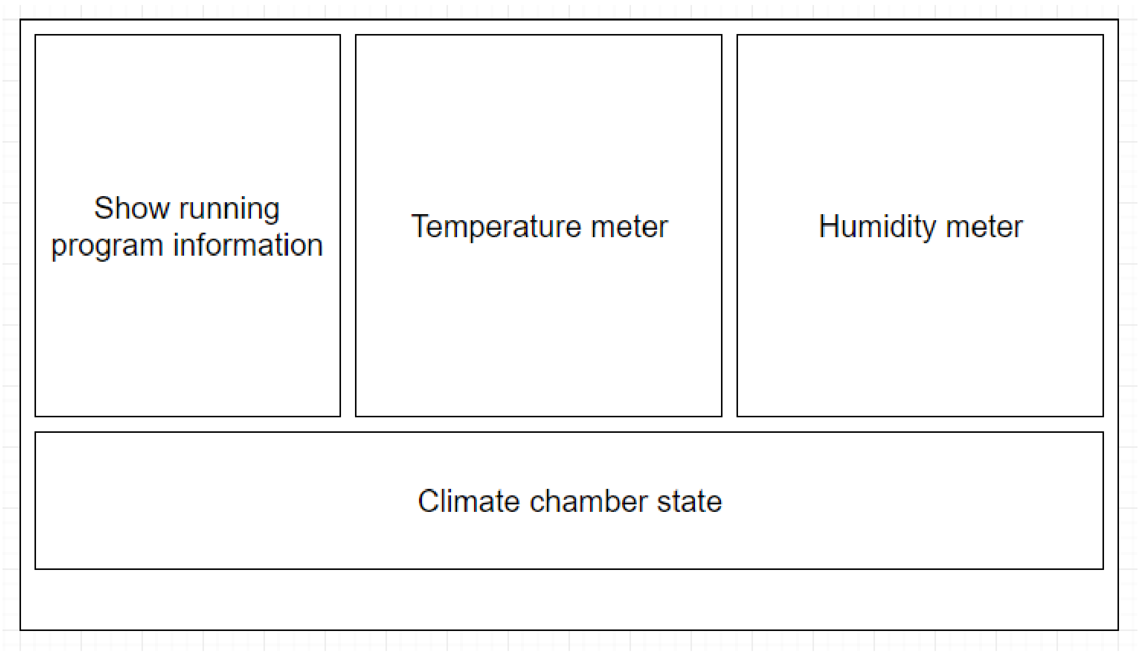


Figure 7. Application home window design

- **The program window**

The program view consists of 2 subcomponents: the programs list on the left side and the steps list of the program on the right side.

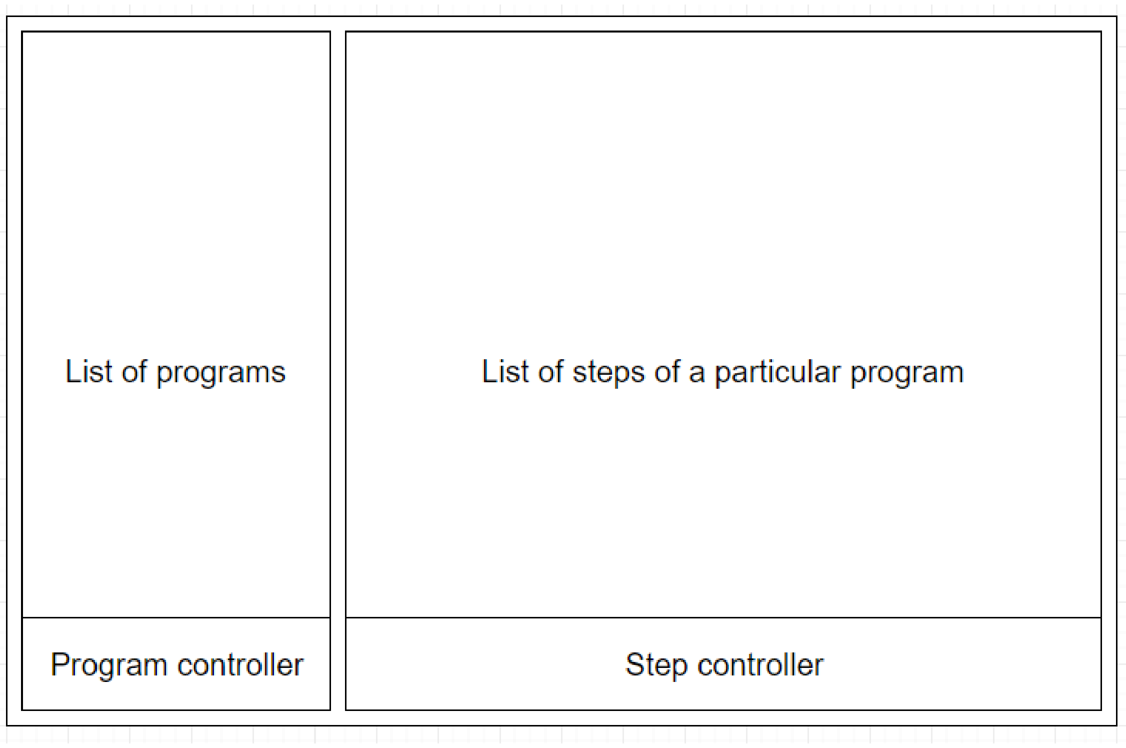


Figure 8. Application program window design

- **The list of programs**

The program list will list down all the available or created programs on top of the view. These programs can be clicked to switch the steps view on the right side and trigger other control at the bottom. Each program on the list has its name, cycles and the created date. The program controller is located at the bottom of the program list view. The program controller is divided into smaller sections. The first section consists of 3 control buttons: create a new program, edit currently selected program and remove the selected program. The create and edit program button when clicked will popup the small form on top of the control view. The form includes information about the program: the name of the new program or the selected program, number of cycles.

When the program on the list is selected, the second section will be triggered, otherwise, these control buttons will be disabled. The second section consists of two buttons: start the program and stop the program.

- **The steps list.**

The steps list has the same structure as the program list. On top of the list is the list of steps that has been created, and on the bottom, the control buttons are located. The control buttons are: create, edit and delete the step. The steps can be clicked to choose in the same manner as the program list.

- **The PID window**

The PID view contains 2 subviews: the temperature and the humidity PID view. The two views have the same structure with each other. It is a table with the main content is the list of the PID parameters value and a field indicated the default PID that can be used for starting the program. At the bottom is the controls: create, edit and delete PID parameters.

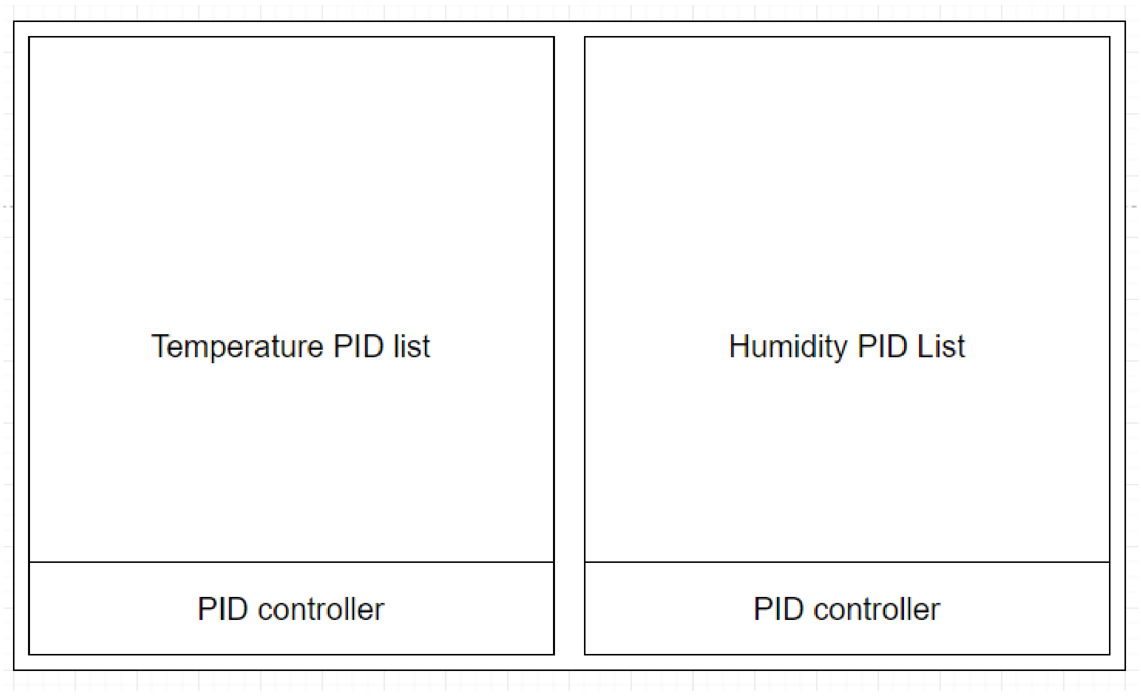


Figure 9. Application PIDs window design.

- **The history window**

The history view has an option to choose the program that runs on specific date and time. After selecting the program, the graph will be displayed below.

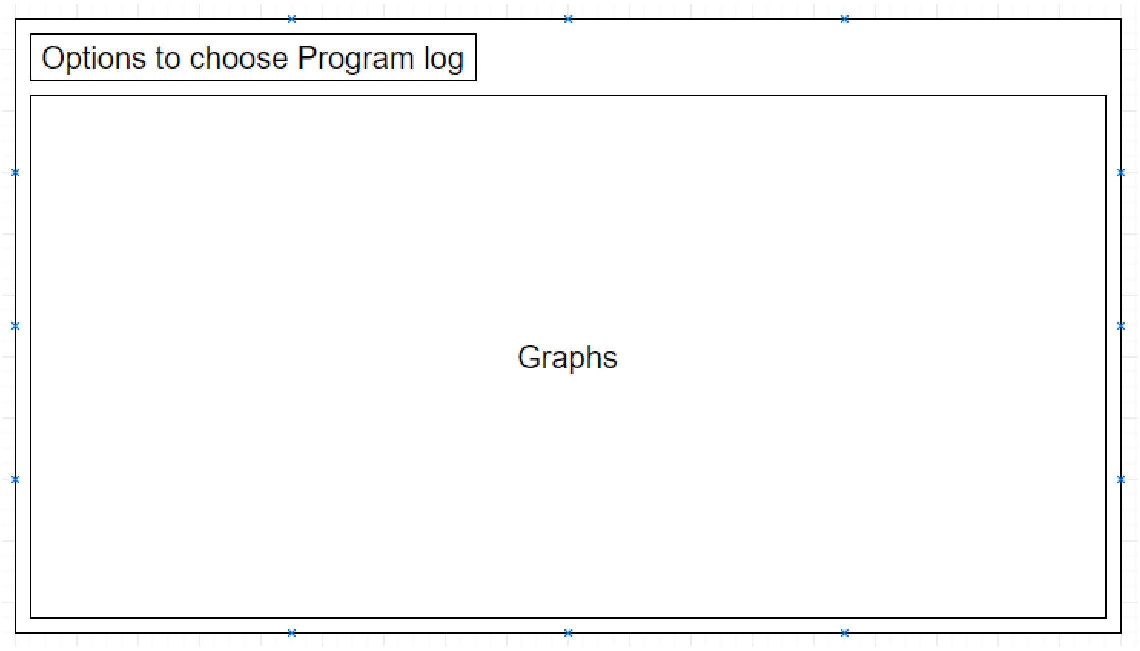


Figure 10. Application History window design

4.3 Data Model

The database of choice for this project is MongoDB. The data models of this application include:

- User model: Model for storing user information when registered to use this application
- Program model: Model for the testing program, storing basic information, such as program name, number of cycles
- Step model: model for storing steps of the program, each model has a hook to program model via ObjectID of according program.
- PID model: used for storing PID parameter information.
- History Model: used for marking date time of which a program is operated.

The detail of each table is listed in the following snippets.

```
const ProgramModel = new Schema({
  name: { type: String, required: true, unique: true },
  cycles: { type: Number, required: true }
}, {
  timestamps: true
})
```

Snippet 1: Program Model

```
const StepModel = new Schema({
  order: { type: Number },
  program_id: { type: ObjectId, required: true },
  temperature: { type: Number, default: 0 },
  humidity: { type: Number, default: 0 },
  time: { type: String },
  wait: { type: String },
  options: { type: Array, default: [false, false, false]}
}, {
  timestamps: true
})
```

Snippet 2: Step Model

```
const PidModel = new Schema({
  default: { type: Boolean },
  type: { type: String, enum: ['temperature', 'humidity'] },
  proportional: { type: Number },
  integral: { type: Number },
  derivative: { type: Number }
})
```

Snippet 3: PID Model

```
const HistoryModel = new Schema({
  program_id: { type: ObjectId, required: true, ref: 'program' },
  temperature_pid: { type: ObjectId, required: true, ref: 'pid' },
  humidity_pid: { type: ObjectId, required: true, ref: 'pid' }
}, { timestamps: true })
```

Snippet 4: History Model

5 IMPLEMENTATION

5.1 Webpack configuration

Webpack configuration was stored in a webpack.config.js file in root folder of the project. In this file, output file, output directory were specified, loaders for transpiling ECMA6 script, Stylus, a set of plugins were declared for Webpack to run. It is worth noticing that in this project, Webpack Dashboard, a plugin that supports visualizing Webpack compiling project, was applied in order to fasten up the front-end development process since it would watch changes in front-end related files and re-compile the files so changes are made immediately.

The content of the webpack configuration file includes:

- Context: specifying source directory of the original code files
- Entry: mapping each code file to destination file.
- Output: specifying output directory, output file name, and public access directory via a webpage.
- The module contains loaders according to file type, used for transpiling the source file into commonly used programming language supported by browsers.
- Plugins: additional plugins that support Webpack functionality
- Watch: webpack observes changes in files and act accordingly.

```
const path = require('path')
const Webpack = require('webpack')
const ExtractTextPlugin = require('extract-text-webpack-plugin')
const DashboardPlugin = require('webpack-dashboard/plugin')

const srcPath = path.resolve(__dirname, 'src/js')
const outputPath = path.resolve(__dirname, 'public/dist')

module.exports = {
  context: srcPath,
  entry: {
    'js/index': './index.es6',
    'css/index': '../css/index.styl',
    'hotMiddleware': 'webpack-hot-middleware/client'
  },
  output: {
    path: outputPath,
    filename: '[name].js',
    publicPath: '/public'
  },
  module: {
    rules: [
      { test: /\.es6$/, exclude: /node_modules/, loader: 'babel-loader' },
      { test: /\.css$/, use: [
        { loader: 'style-loader' },
        { loader: 'css-loader' }
      ] },
      {
        test: /\.styl$/,
        loader: ExtractTextPlugin.extract({ fallback: 'style-loader', use: 'css-loader!stylus-loader' })
      },
      { test: /\.(png|jpg|gif|svg|ttf|woff|woff2)$/, loader: 'url-loader'},
      { test: /\.node$/, use: 'node-loader' }
    ]
  },
  plugins: [
    new ExtractTextPlugin('./css/index.css'),
    new DashboardPlugin(),
    new Webpack.HotModuleReplacementPlugin(),
    new Webpack.NoErrorsPlugin()
  ],
  resolve: {
    extensions: ['.js', '.json', '.node', '.es6'],
    modules: [srcPath, 'node_modules'],
    //mainFields: ["loader", "main"]
  },
  resolveLoader: {
    extensions: ['.js'],
    modules: ['node_modules'],
    //mainFields: ["loader", "main"]
  },
  watch: true,
  target: 'web'
}
```

Snippet 5. Webpack configuration snippet.

In the webpack configuration, the Hot Module Replacement Plugin is used to help boost up development speed, DashboardPlugin provides visual view when compiling files with Webpack.

In the server file – index.js, the Webpack, Webpack Dashboard module and the config file are required. These requirements are needed to enable the Hot Module Replacement features:

```
const Webpack = require('webpack')
const Dashboard = require('webpack-dashboard/plugin')
```

Snippet 6. Include Webpack – Webpack Dashboard modules

```
const wpConfig = require('../../webpack.config')
```

Snippet 7. Include Webpack configuration file

```
const compiler = Webpack(wpConfig)
compiler.apply(new Dashboard())

const host = Config.server.host
const port = Config.server.port

const devMiddleware = require('webpack-dev-middleware')(compiler, {
  host, port, noInfo: true,
  historyApiFallback: true,
  publicPath: wpConfig.output.publicPath,
  quiet: true
})

const hotMiddleware = require('webpack-hot-middleware')(compiler, {
  log: () => {}
})
```

Snippet 8. Configure Webpack features

The Webpack features are later registered as server extensions.

```
server.ext('onRequest', (request, reply) => {
  devMiddleware(request.raw.req, request.raw.res, err => {
    if (err) {
      return reply(err)
    }

    return reply.continue()
  })
})

server.ext('onRequest', (request, reply) => {
  hotMiddleware(request.raw.req, request.raw.res, err => {
    if (err)
      return reply(err)

    return reply.continue()
  })
})
```

Snippet 9. Webpack features registered.

5.2 Binary handling

In this project, the communication with the chamber is operated via the RS232 protocol. However, the command that is sent to the chamber work in form or binary, and JS lacks this feature. Therefore, a binary handler is needed to help with converting a number to binary accordingly.

```
function transform (value, bits) {
  var rslt = value >= 0 ? Array(bits).fill(0) : Array(bits).fill(1)

  var v
  if (isNumber(value)) {
    v = value.toString().charCodeAt(0)
  } else if (isString(value)) {
    v = value.charCodeAt(0)
  }

  while(v >= 0 && --bits >= 0) {
    if (v - Math.pow(2,bits) >= 0) {
      rslt[rslt.length -bits-1] = 1
      v -= Math.pow(2,bits)
    }
  }

  return rslt
}
```

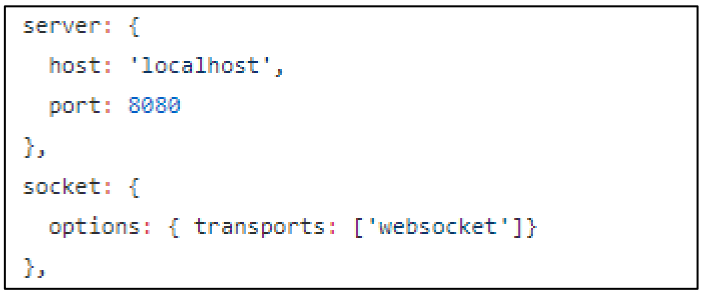
Snippet 10. Function for transforming number to binary function.

The function accepts 2 parameters, value as an integer and number of bits. Then it initializes an array of bits, fills with 0 if a value is greater or equal to 0, otherwise, fill with 1.

The value is transformed into hexadecimal code. The hexadecimal value is later transformed into its match binary code as an output of the function.

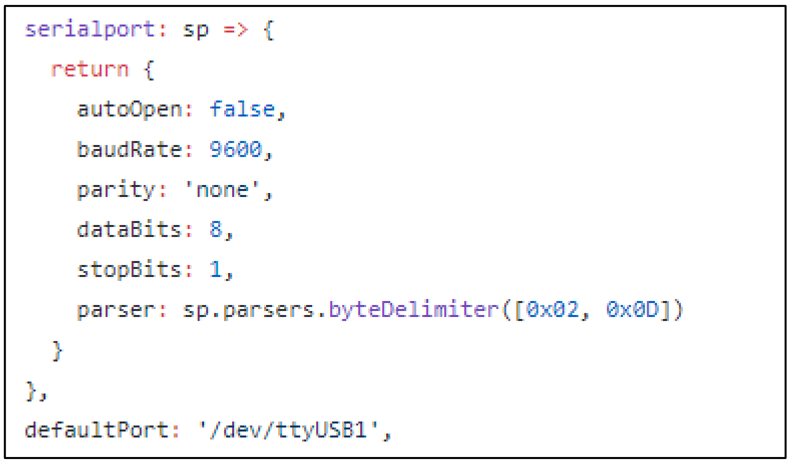
5.3 The configuration file

The configuration file exports the global configuration for the application, including server configuration, serial communication port configuration, web socket configuration, database configuration. The exports configuration is in JSON formatted. Notable configurations are shown in below snippets:



```
server: {
  host: 'localhost',
  port: 8080
},
socket: {
  options: { transports: ['websocket']}
},
```

Snippet 11. Server and web socket configuration.



```
serialport: sp => {
  return {
    autoOpen: false,
    baudRate: 9600,
    parity: 'none',
    dataBits: 8,
    stopBits: 1,
    parser: sp.parsers.byteDelimiter([0x02, 0x0D])
  }
},
defaultPort: '/dev/ttyUSB1',
```

Snippet 12. Serial communication port and its default communication port configuration.

```
  database: {
    host: '127.0.0.1',
    port: 27017,
    db: 'climate-chamber'
  }
```

Snippet 13. Database connection configuration.

5.4 The server.

The application will be operated once the server file is compiled. The server is initialized in the following steps.

5.4.1. Requirements

```
/* Server related modules */
const {Server} = require('hapi')
const Vision = require('vision')
const Inert = require('inert')
const Nunjucks = require('nunjucks')

/* Webpack related modules */
const Webpack = require('webpack')
const Dashboard = require('webpack-dashboard/plugin')

/* Websocke and Serial communication modules */
const Socket = require('socket.io')
const Serialport = require('serialport')

/* Server functionality modules */
const Command = require('./modules/commands')
const Chamber = require('./modules/chamber')
const Controller = require('./modules/controller')

/*Webpack configuration and server configuration file */
const wpConfig = require('../../webpack.config')
const Config = require('../config')

const { db } = require('./db')
const { endpoints } = require('./routes')
const { sendMsg } = require('./helpers')
```

Snippet 14. Module requirements of the server.

The first line of the snippet indicates that only the Server object of the HapiJS module is needed. Vision and Nunjucks modules are required to serve the front-end interface.

5.4.2. The serial connection.

The serial communication connection is the critical part of this project. The communication is supported by serialport NodeJS package.

“Node-Serialport provides a stream interface for low-level serial port code necessary to control Arduino chipset, x10 interfaces, Zigbee radios, highway signs, …. “. /Node-Serialport introduction/

The serial port communication is initialized at the same time with the server, yet open for communication. Once the server is running, the user can open the connection by click on the ‘Connect’ button of the serial communication status section. It will emit a message to the server to open the connection to the chamber and start the communication by sending messages back and forth, and the server will emit a web socket message to the client, containing information related to the chamber.

The initialization of the serial connection is described in the below snippets:

```
const serialport = new Serialport(Config.defaultPort, Config.serialport(Serialport))
```

Snippet 15. Serial connection initialization.

The serial connection is initialized with its configuration stored in the configuration file.

```
openSerial: callback => {
  if (!serialport.isOpen()) {
    serialport.open(err => {
      if (err) {
        callback({ error: true, message: 'Cannot open serialport', status: serialport.i
      } else {
        const interval_1 = setInterval(_ => emitter.emit('get-chamber-info'), 1000)
        if (connectionTimeout == null) {
          connectionTimeout = setInterval( _ => {
            if (connectionCounter >= 5) {
              emitter.emit('terminate-serial')
              connectionCounter = 0
              clearInterval(connectionTimeout)
              connectionTimeout = null
            } else {
              ++connectionCounter
            }
          }, 1000)
        }

        callback({ error: false, message: null, status: serialport.isOpen() })
      }
    })
  } else {
    callback ({ error: false, message: null, status: serialport.isOpen() })
  }
},
```

Snippet 16. Open serial port and start communication.

Once the server received a message from the client, it opens the connection, if the connection is open, it will start to send out a message every second. Otherwise, will send an error instead.

5.4.3. The database connection.

The database was configured in ‘db’ module. The database configuration was loaded from the global configuration file. These configurations were used to set up a connection with MongoDB. Once the server starts, the connection is open at the same time.

```
const Mongoose = require('mongoose')
const Config = require('../config')

Mongoose.connect(`mongodb://${Config.database.host}:
  ${Config.database.port}/${Config.database.db}`)
const db = Mongoose.connection

db.on('error', console.error.bind(console, 'Database connection error!'))
db.once('open', _ => {
  console.log('Database connection is successfully established!')
})

exports.Mongoose = Mongoose
exports.db = db
```

Snippet 17. Database connection.

5.4.4. The routing and handlers

The routing for the application is a simple JSON object, indicating the URL, its method (POST, GET, DELETE) and its handlers. The handlers are a set of functions specified in 'handlers.js'. Each handler will handle the request and response with according reply.

The details of the handlers will be discussed in the later section together with the client-side interface.

```
const handlers = require('./handlers')

exports.endpoints = [
  { method: 'GET', path: '/', handler: handlers.index },
  { method: 'GET', path: '/public/{param*}', handler: { directory: { path: 'public', listing: true } } },

  { method: 'GET', path: '/programs', handler: handlers.getPrograms },
  { method: 'GET', path: '/programs/{programId}', handler: handlers.getProgramById },
  { method: 'POST', path: '/programs/add', handler: handlers.addProgram },
  { method: 'POST', path: '/programs/edit', handler: handlers.editProgram },
  { method: 'DELETE', path: '/programs/remove', handler: handlers.removeProgram },

  { method: 'GET', path: '/steps/getOne/{_id}', handler: handlers.getStepById },
  { method: 'GET', path: '/steps/{programId}', handler: handlers.getSteps },
  { method: 'POST', path: '/steps/add', handler: handlers.addStep },
  { method: 'POST', path: '/steps/edit', handler: handlers.editStep },
  { method: 'DELETE', path: '/steps/remove', handler: handlers.removeStep },

  { method: 'GET', path: '/pids', handler: handlers.getPids },
  { method: 'GET', path: '/pids/{_id}-{type}', handler: handlers.getPidById },
  { method: 'GET', path: '/pids/default', handler: handlers.getDefaultPid },
  { method: 'POST', path: '/pids/add', handler: handlers.addPid },
  { method: 'POST', path: '/pids/set-default', handler: handlers.setDefaultPid },
  { method: 'DELETE', path: '/pids/remove', handler: handlers.removePid },

  { method: 'GET', path: '/history', handler: handlers.getHistories },
  { method: 'GET', path: '/history/{_id}', handler: handlers.getHistoryById },
]
```

Snippet 18. Routes and request handlers.

5.4.5. Initialize the Server.

The server is initialized with the default port and host.

```
const server = new Server()
server.connection(Config.server)
```

Snippet 19. Server initialization.

Following the initialization of the server, the extensions and necessary modules that provide functionality for the server are registered. After that, routes and view manager are added. After registering necessary modules, the server will start.

```
server.register([Config.good, Vision, Inert], err => {
  if (err) {
    throw err
  }

  // Register routes
  server.route(endpoints)

  // Nunjucks - view manager
  server.views( Config.engines(Nunjucks, __dirname))
})

server.start(err => {
  if (err) {
    throw err
  }

  console.log(`Server is running on ${server.info.uri}`)
})
```

Snippet 20. Modules, routes, view manager registration and server start.

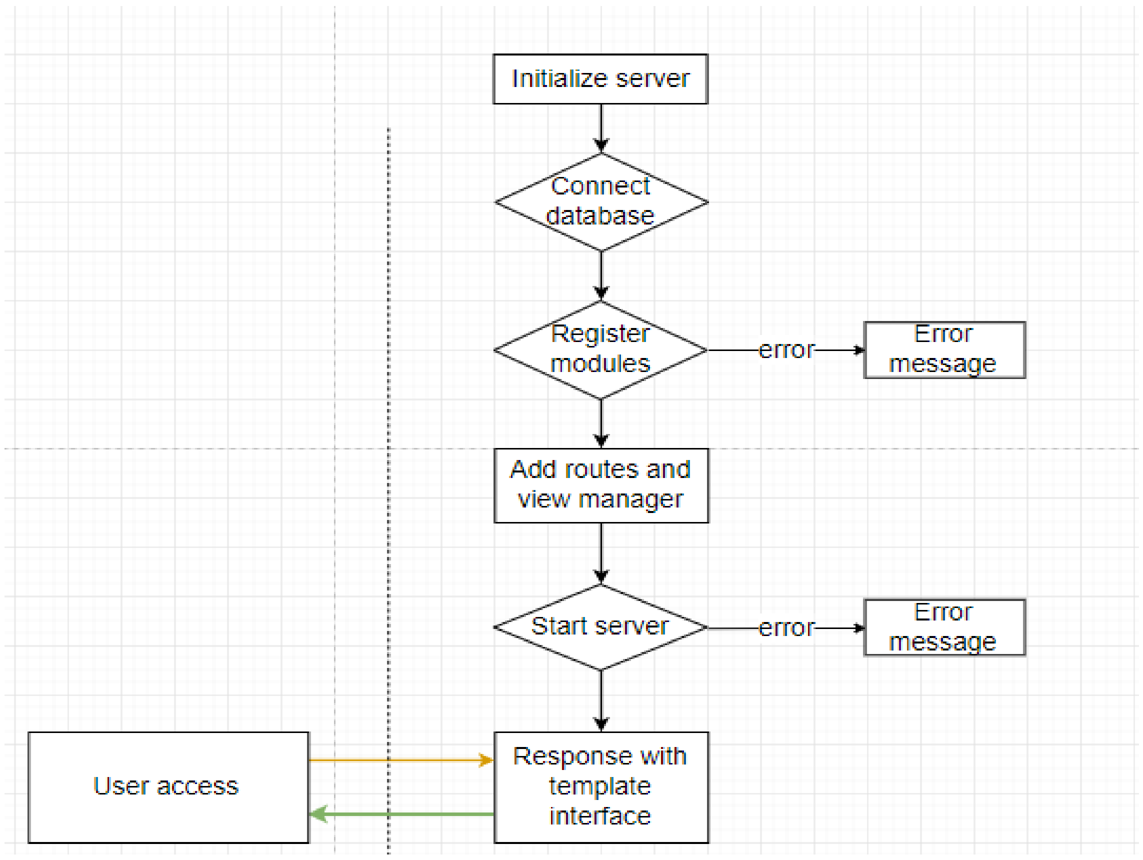


Figure 11. Server initialization process.

- **The Chamber Object.**

The chamber object is a simple JS object, storing information of the chamber, including dry and wet temperature and humidity. The chamber object has one method to assign the received data from the actual chamber to its properties.

```
function Chamber() {
  this.dryTemperature = null
  this.wetTemperature = null
  this.humidity = null

  this.setValue = (data) => {
    this.dryTemperature = data[0]
    this.wetTemperature = data[1]
    this.humidity = data[2]
  }
}

module.exports = Chamber
```

Snippet 21. The chamber object.

- **The command**

The command is one of the most important components that decide the communication with the chamber is successful or not. The command has 4 different types, classified by an alphabet letter: A, I, B, O. The first three are static, means its content will always the same, while the O message will contain the control message. The command object has the following attributes:

```
function Command () {
  this.ready = false
  this.idle = true
  this.isConnected = false

  this.htBlock = new Bitset(0)
  this.plBlock = new Bitset(0)
  this.cvBlock = new Bitset(0)
  this.vfBlock = new Bitset(0)
  this.tempPw1 = new Bitset(0)
  this.tempPw2 = new Bitset(0)
  this.humidPw1 = new Bitset(0)
  this.humidPw2 = new Bitset(0)

  this.temperaturePower = 0
  this.humidityPower = 0
```

Snippet 22. Command object's attributes.

The Bitset object is a user-defined object that replicates bitset object in C++ programming language.

The command object has these following usages:

- Read the data sent from the chamber
- Return buffer containing commands sent to the chamber.
- Return status of the chamber's components.

The command object is used by other object and its properties are manipulated in the provided methods.

- **The PID object**

The PID control output is calculated by the PID object. The PID object is initialized with provided PID fetched from the database, which stores the value of integral, derivative and proportional value. When a program is started, the target temperature or humidity is set, and before every cycle that the server sends a control message to the chamber, the PID object will calculate the output. The output is later used for controlling the humidifier and heater power.

```
function Pid (pid) {
  this.previousError = 0
  this.accumulator = 0

  this.ki = pid.integral
  this.kd = pid.derivative
  this.kp = pid.proportional
  this.dt = 1
  this.targetValue = null
  this.lastMeasure = null

  this.output = (measuredValue, dt) => {
    // console.log(`${this.targetValue} - ${measuredValue} - dt ${dt}`)
    let error = this.targetValue - measuredValue
    if (error > 190 || error < -190) {
      return 0
    }
    // console.log(`Error: ${this.error}`)
    this.accumulator += error * (dt/1000)

    let proportional = this.kp * error
    let integral     = this.ki * this.accumulator
    let derivative   = this.kd * ((error - this.previousError)/(dt/1000))
    // console.log(`PID: ${proportional} - ${integral} - ${derivative}`)

    let output = proportional + integral + derivative
    // console.log(`output: ${output}`)
    this.previousError = error

    console.log(output < 0 ? 0 : output > 255 ? 255 : parseInt(output))
    return output < 0 ? 0 : output > 255 ? 255 : parseInt(output)
  }
}

module.exports = Pid
```

Snippet 23. The PID object and its output calculation function.

5.2 The client-side interface

The graphical interface of the program is loaded whenever the user accesses the page. The server will respond with the ready-built template. However, the template is a simple blank page and all the functionalities and components are rendered by MithrilJS when loading the page.

5.2.1 The general view

The general view consists of the heading title of the application, the small panel on the left side with tab header and serial indicator and finally the main content container, which is dynamic depending on the chosen tab header.

5.2.2 The sidebar

The sidebar is the fixed section of the interface, located on the left side where it contains the navigator and the serial connection indicator.

- **The serial indicator**

The serial indicator is a simple box that contains information about the serial communication status. If the server and the chamber currently communicate with each other, the icon in the box will display as check with a green background, otherwise, it will be the cross icon with a red background. The serial status is checked right before the component is rendered on the page to make sure the page is updated with the server information. The Connect button when clicked will send a message to the server to trigger opening the communication port between the server and the chamber. The message is conducted through a web socket. Once the message is sent out, it will wait for the response from the server and redraw the state of the communication.

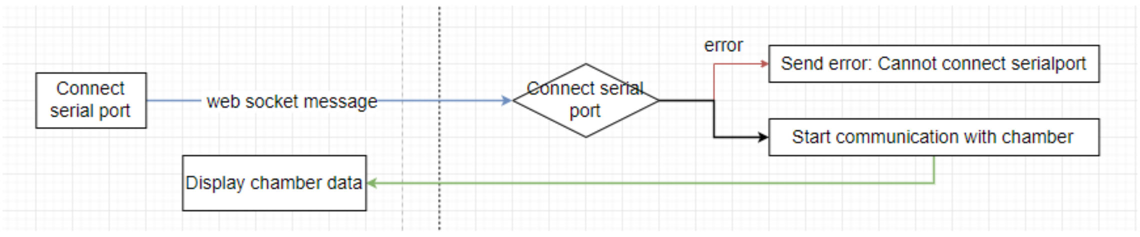


Figure 12. The connect serial communication button mechanism.

- **The navigator**

The navigator or the tab is controlled by MithrilJS built-in routing mechanism. The routing enables switching among different views of the application. Using this function of MithrilJS, the tab headers are linked with different views with the URL starting with

“#!/[view name]”. Different views are made ready and map with suitable links as follows:

```
const links = [
  { route: '/', name: 'Home', component: Display },
  { route: '/programs', name: 'Programs', component: Programs },
  { route: '/pid', name: 'PID', component: Pid },
  { route: '/graph', name: 'Graph', component: Graph },
  { route: '/about', name: 'About', component: About }
]
```

Snippet 24. Links object with specified routes, names and their matched components/views.

```
domready(_ => {
  const content = document.getElementById('js-content')

  m.route(content, '/', links.reduce((obj, link) => {
    obj[link.route] = link.component
    return obj
  }, {}))
})
```

Snippet 25. Route registration

When the Document Object Model (DOM) is fully loaded, the content container on the template with being routed with the default route, and at the same time, the components are mapped to the routing mechanism of MithrilJS.

5.2.3 The main view.

The main view takes the major area of the interface. It is a container for different views of the application, including home view, program management view, PID management view and the graphical view.

- **The home view**

The home view displays the information regarding the chamber, including chamber running program information, the chamber temperature and humidity and finally the chamber components' status. When the server and the chamber are not connected, this information is shown as empty, with a four-consecutive dash. Once the chamber and the server are connected and start their communication, the information will be display on the page, updated every 1 second. The communication between the server and the client is done via web socket.

- **The program view**

The program view provides visual management of the programs and their steps. The view includes 2 sections: the program list and its matched steps. The programs are listed on the left side of the view. When the user clicks on the program tab header, the page sends a request to the server to get the list of the programs. The programs are displayed with their name, a number of cycles and date created. Below the listed program, is the control section. Basic controls are creating, editing and deleting a program, and additional controls start and stop the program. The create program button is clicked to pop-up the form, including name and number of cycle. The edit button works in the same way as the create program, however, the fields will be filled with its value, and the delete button will send a request to the server to remove the program entity off the database. After receiving the response from the server, the program list will be redrawn with updated data.

```
/* Method: GET - Get all programs */
exports.getPrograms = (request, reply) => {
  Models.Program.find({}, (err, programs) => {
    if (err) {
      reply(Boom.badImplementation(err))
    }
    reply({ success: true, programs: programs})
  })
}
```

Snippet 26. Program fetching request handler.

```
/* Method: GET - Get program by ID */
exports.getProgramById = (request, reply) => {
  let res = { error: null }
  Models.Program.findOne({ _id: request.params.programId }, (err, program) => {
    if (!err && program) {
      Models.Step.find({ program_id: request.params.programId }, (err, steps) => {
        if (!err) {
          reply({ success: true, program: program, steps: steps })
        } else {
          reply({ success: false, error: err })
        }
      })
    } else if (err) {
      reply({ success: false, error: err })
    } else if (!program) {
      reply({ success: false, error: 'Cannot find program' })
    }
  })
}
```

Snippet 27. Program choosing and steps fetching handler.

The handler accepts a program id as its parameter, then uses this parameter to find a matched program and its steps, packs all the information in JSON format and sends back to the browser.

```
/* Method: POST - Add new program */
exports.addProgram = (request, reply) => {
  const payload = request.payload
  Models.Program.create(request.payload, err => {
    let res = {}
    if (err) {
      res = { success: false, err: err }
    } else {
      res = { success: true }
    }

    reply(res)
  })
}
```

Snippet 28. Program adding the handler.

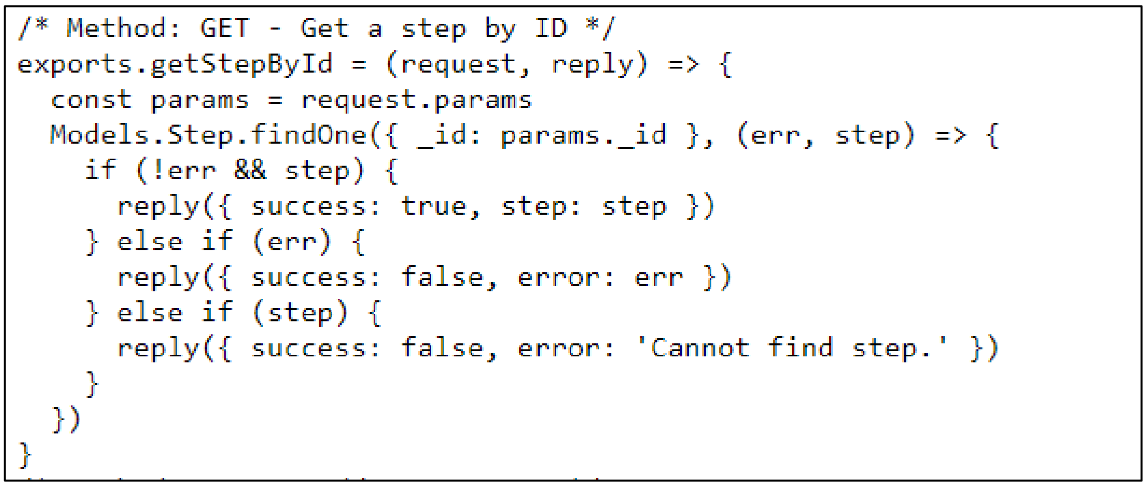
```
/* Method: POST - Edit program */
exports.editProgram = (request, reply) => {
  const payload = request.payload
  Models.Program.update({ _id: payload._id }, { name: payload.name, cycles: payload.cycles }, error => {
    if (error) {
      reply ({ success: false, error: error })
    }
    reply({ success: true, error: null })
  })
}
```

Snippet 29. Program editing handler.

```
/* Method: DELETE - remove a program */
exports.removeProgram = (request, reply) => {
  const payload = request.payload
  Models.Step.remove({ program_id: payload._id }, err => {
    if (err) {
      reply({ success: false, error: err })
    }

    Models.Program.remove(payload, err => {
      if (err) {
        reply({ success: false, error: err})
      }

      reply({ success: true })
    })
  })
}
```

Snippet 30. Program removing handler

The above snippets are for basic CRUD operations: create, update and delete a program. The program creating operation gets all necessary detail from a program creating a form to create a new program. The similar process is applied for updating a program, except it gets a program id and updates its value. For deleting a program, the handler first finds all related steps of the program using the program's id and remove all of them. After that, it proceeds to remove the program.



```
/* Method: GET - Get a step by ID */
exports.getStepById = (request, reply) => {
  const params = request.params
  Models.Step.findOne({ _id: params._id }, (err, step) => {
    if (!err && step) {
      reply({ success: true, step: step })
    } else if (err) {
      reply({ success: false, error: err })
    } else if (step) {
      reply({ success: false, error: 'Cannot find step.' })
    }
  })
}
```

Snippet 31. Steps fetching handler

Once a program is clicked or chosen from the list, its steps are fetched from the server and displayed on the right panel. The handler accepts a program's id and uses it for finding the matching steps. Steps are sorted according to their orders. The bottom of steps view will be the controls, including creating, editing and deleting the steps. For the creating and editing, the process is same as creating and editing a program. For deleting the steps, the process is different since steps have their order which is a number indicating the order of processing. In case of deleting the last step, it will be removed without any constraint. However, if a step has another step or other steps after it, the following steps' order will be updated.

```
/* Method: POST - Add a new step */
exports.addStep = (request, reply) => {
  const payload = request.payload
  Models.Step.where('program_id', payload.program_id).sort({ order: -1 }).limit(1).exec((err, step) => {
    if (err) {
      reply({ success: false, error: err })
    } else {
      let pl = {
        program_id: Mongoose.Types.ObjectId(payload.program_id),
        temperature: payload.temperature,
        humidity: payload.humidity,
        time: timeFormat(payload.time),
        wait: timeFormat(payload.wait),
        options: payload.options
      }
      if (isEmpty(step)) {
        pl.order = 1
      } else {
        pl.order = step[0].order + 1
      }

      Models.Step.create(pl, err => {
        if (err) {
          reply({ success: false, error: err })
        } else {
          reply({ success: true })
        }
      })
    }
  })
}
```

Snippet 32. Step adding handler.

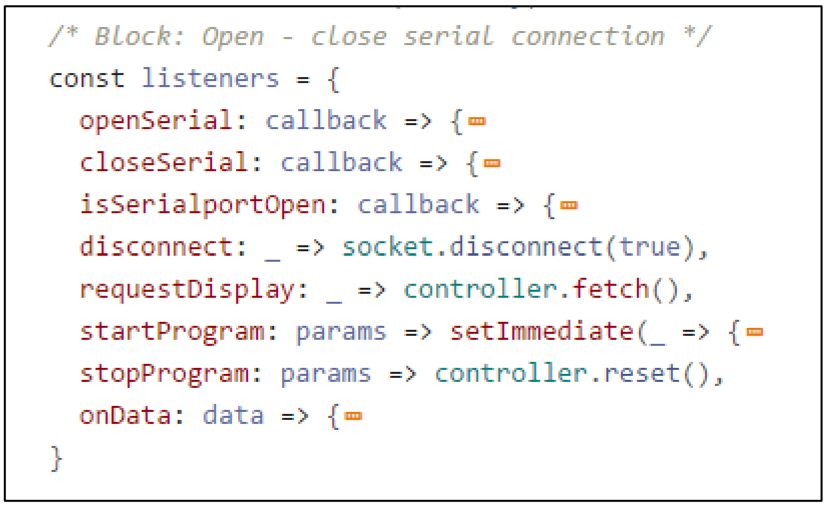
For adding a step to a program, the handler finds all previously created steps, sorts them by steps' order in descending order. If there is not any step, the newly created step will have it order as 1, otherwise, its order is the last step's order plus 1.

```
/* Method: DELETE - Remove a step */
exports.removeStep = (request, reply) => {
  Models.Step.findOne({_id: ObjectId(request.payload._id) }, (err, step) => {
    if (!err && step) {
      Models.Step.update({ order: {$gt: step.order} }, { $inc: {order:-1} }, { multi: true }, (err, s) => {
        console.log(s)
        if (err) {
          reply({ success: false, error: err })
        }
      })
      step.remove()
      reply({ success: true })
    } else if (!step) {
      reply({ success: false, error: 'Cannot find step.' })
    } else if (err) {
      reply({ success: false, error: err })
    }
  })
}
```

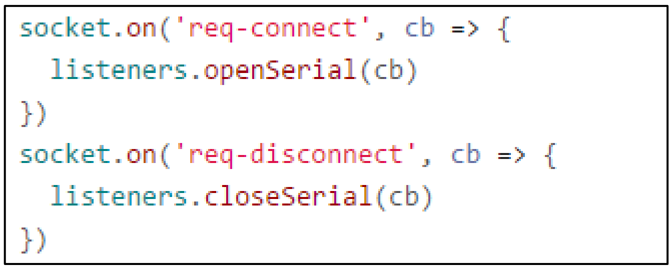
Snippet 33. Step removing handler

When a step is removed, the handler looks for steps that have lower order than the removed one and update their order accordingly. After that, the step is removed from the database.

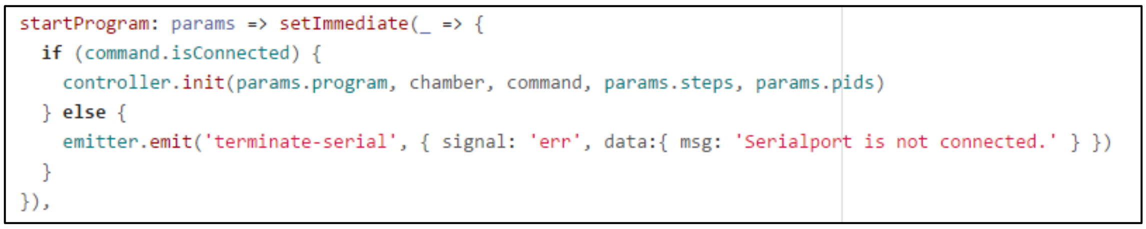
When a program is chosen to be started, the browser sends a web socket message to the server. The server first checks if it is communicating with the chamber. If yes, it will start the program by initializing the program controller, with necessary parameters, otherwise response with another web socket message indicating there’s no connection to the chamber.


```
/* Block: Open - close serial connection */
const listeners = {
  openSerial: callback => {⋯
  closeSerial: callback => {⋯
  isSerialportOpen: callback => {⋯
  disconnect: _ => socket.disconnect(true),
  requestDisplay: _ => controller.fetch(),
  startProgram: params => setImmediate(_ => {⋯
  stopProgram: params => controller.reset(),
  onData: data => {⋯
}
```

Snippet 34. List of listener actions, including start and stop a program.


```
socket.on('req-connect', cb => {
  listeners.openSerial(cb)
})
socket.on('req-disconnect', cb => {
  listeners.closeSerial(cb)
})
```

Snippet 35. Socket listener from the client.


```
startProgram: params => setImmediate(_ => {
  if (command.isConnected) {
    controller.init(params.program, chamber, command, params.steps, params.pids)
  } else {
    emitter.emit('terminate-serial', { signal: 'err', data:{ msg: 'Serialport is not connected.' } })
  }
}),
```

Snippet 36. Start a program

The initialization of the controller follows these steps:

- The server first checks if the program exists and has at least 1 step. If the condition is satisfied, the controller is initialized and a log recording temperature and humidity as a text file are recorded. Otherwise, the server will emit a message to indicate starting the program failed.

```
this.init = (program, chamber, cmd, steps, pids) => {
  if (isEmpty(this.program)) {
    this.program = program
    if (!isEmpty(steps)) {
      this.steps = steps
      this.currentStep = isEmpty(steps) ? {} : this.steps[this.currentIndex]
      this.programStart = new Date()
      this.temperaturePid = new Pid(pids.temperature)
      this.humidityPid = new Pid(pids.humidity)
      this.chamber = chamber
      this.command = cmd
      this.totalSec = 0

      History.create({
        program_id: this.program._id,
        temperature_pid: pids.temperature._id,
        humidity_pid: pids.humidity._id
      }, (error, history) => {
        if (error) {
          console.log(error)
        } else {
          this.logfile = `${directory}${history._id}.txt`
          console.log(this.program)
          fs.appendFile(this.logfile,
            `${this.program.name} - ${moment().format('MMMM Do YYYY, HH:mm:ss')} -
          ${this.program.cycles} - ${this.steps.length}\n`)
        }
      })

      this.setup()
    } else {
      emitter.emit('control', { signal: 'err', data: { msg: 'No step has been set.' } })
      this.reset()
    }
  } else {
    emitter.emit('control', {
      signal: 'err',
      data: {
        msg: this.program && this.program._id == program._id ?
          'This program is already started' : 'Another program is running.'
      }
    })
  }
}
```

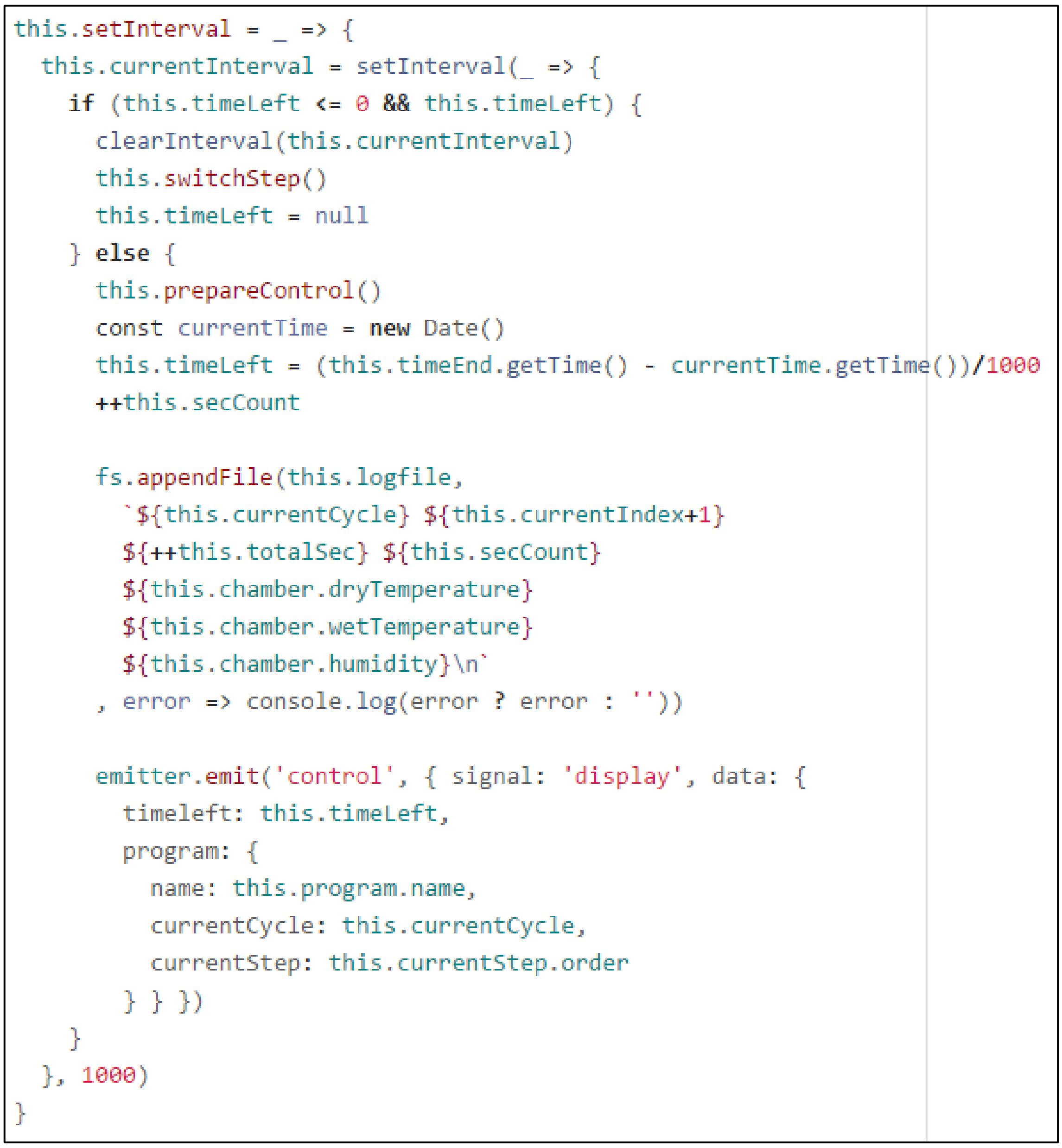
Snippet 37. Controller's method to initialize the program.

- The server then calculates the remaining time of the step, set up the environment to start communication to control the chamber to reach the program's goals.

```
this.setup = _ => {
  let time = this.currentStep.time.split(':')
  let ms = parseInt(time[0])*hour + parseInt(time[1])*minute
  const timeStart = new Date()
  this.secCount = 0
  this.timeEnd = new Date(timeStart.getTime() + ms)
  this.temperaturePid.targetValue = this.currentStep.temperature
  this.humidityPid.targetValue = this.currentStep.humidity
  this.command.idle = false
  this.setInterval()
}
```

Snippet 38. Controller's method for setup the program for running.

- Then for every 1 second, the server emits a signal to calculate the output sent to the chamber and provides an additional message to the client indicating the state of the program and the chamber. The output is calculated by first collecting the chamber data (temperature and humidity), secondly measuring time differences between the last calculation and current calculation (because JavaScript interval is not exactly 1 second), then apply the current value of temperature and humidity with time differences to PID object for calculating the output. Depend on the output and the differences in temperature and humidity, the controller object will manipulate the chamber operations accordingly.



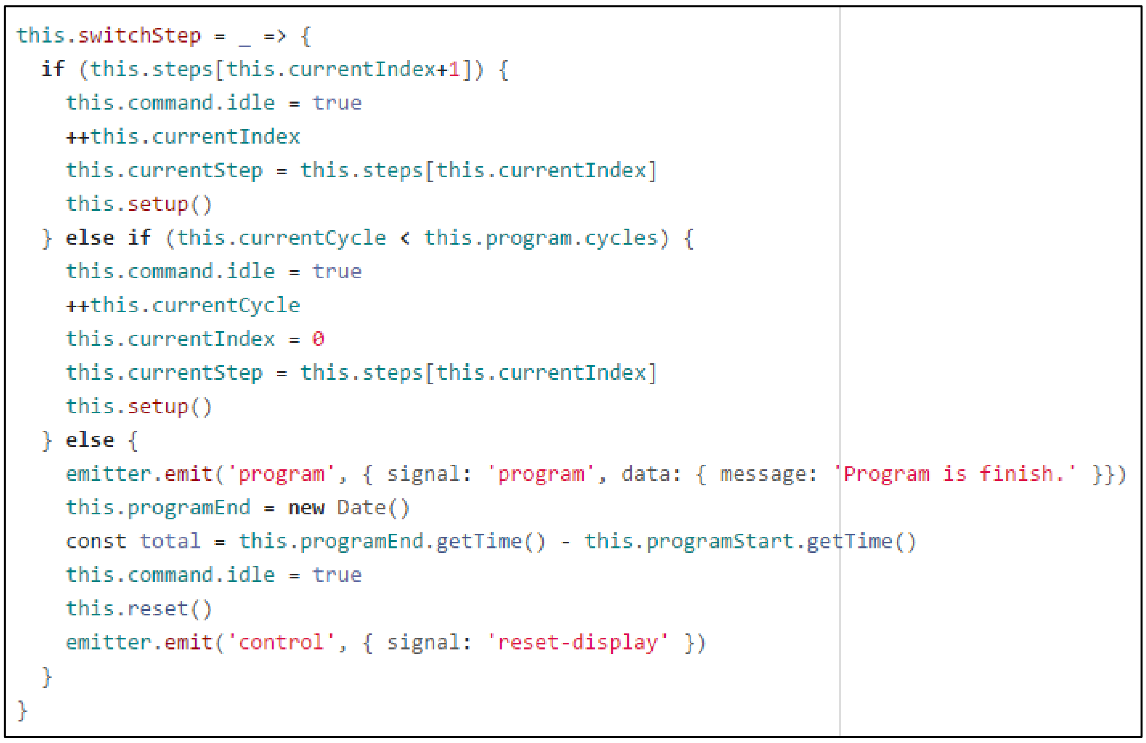
```
this.setInterval = _ => {
  this.currentInterval = setInterval(_ => {
    if (this.timeLeft <= 0 && this.timeLeft) {
      clearInterval(this.currentInterval)
      this.switchStep()
      this.timeLeft = null
    } else {
      this.prepareControl()
      const currentTime = new Date()
      this.timeLeft = (this.timeEnd.getTime() - currentTime.getTime())/1000
      ++this.secCount

      fs.appendFile(this.logfile,
        `${this.currentCycle} ${this.currentIndex+1}
        ${++this.totalSec} ${this.secCount}
        ${this.chamber.dryTemperature}
        ${this.chamber.wetTemperature}
        ${this.chamber.humidity}\n`
      , error => console.log(error ? error : ''))

      emitter.emit('control', { signal: 'display', data: {
        timeleft: this.timeLeft,
        program: {
          name: this.program.name,
          currentCycle: this.currentCycle,
          currentStep: this.currentStep.order
        } } })
    }
  }, 1000)
}
```

Snippet 39. Controller interval function for recording chamber status, calculating the output for the chamber, and emitting signal to the user interface.

```
this.prepareControl = _ => {
  const on = 1
  const off = 0

  const tempOutput = this.chamber.dryTemperature
  const humidOutput = this.chamber.humidity

  let dt
  if (this.lastMeasure == null) {
    this.lastMeasure = new Date()
    dt = this.lastMeasure - this.programStart
  } else {
    let t = new Date()
    dt = t - this.lastMeasure
    this.lastMeasure = t
  }

  this.command.temperaturePower = this.temperaturePid.output(tempOutput, dt)
  this.command.humidityPower = this.humidityPid.output(humidOutput, dt)

  if (tempOutput <= 0) {
    this.command.cvBlock.set(1, on)
    this.command.vfBlock.set(1, on)
    this.command.switchHeaters(off)
  } else {
    this.command.switchHeaters(on)
    this.command.switchCoolers(off)
  }

  if (humidOutput <= 0) {
    this.command.switchHumidifiers(off)
  } else {
    this.command.switchHumidifiers(on)
  }
}
```

Snippet 40. Controller function for calculating the output and giving a decision for the chamber’s operations.

- At the end of every step, the server checks for next available steps or next available cycles. If any step or cycle is remaining, the program will be shifted to the next one, else, the server will emit a message indicating the program has been done.



```
this.switchStep = _ => {
  if (this.steps[this.currentIndex+1]) {
    this.command.idle = true
    ++this.currentIndex
    this.currentStep = this.steps[this.currentIndex]
    this.setup()
  } else if (this.currentCycle < this.program.cycles) {
    this.command.idle = true
    ++this.currentCycle
    this.currentIndex = 0
    this.currentStep = this.steps[this.currentIndex]
    this.setup()
  } else {
    emitter.emit('program', { signal: 'program', data: { message: 'Program is finish.' }})
    this.programEnd = new Date()
    const total = this.programEnd.getTime() - this.programStart.getTime()
    this.command.idle = true
    this.reset()
    emitter.emit('control', { signal: 'reset-display' })
  }
}
```

Snippet 41. The step and cycle switching function.

For stopping a running program, a web socket message is emitted from the client to the chamber indicating a request for stopping a program. The controller will terminate all running interval function and reset all its parameters to default.

- **The PID window**

The PID view includes 2 PID tables, temperature, and humidity PID respectively. The PID table consists of PID list and the control at the bottom. The PIDs are fetched from the server via a HTTP request when PID tab header is clicked. The controls of the PID view are creating, editing and deleting a PID set.

```
/* Method: GET - Get all PID parameters */
exports.getPids = (request, reply) => {
  Models.Pid.find({}, (err, pids) => {
    if (err) {
      reply({ success: false, error: err })
    }

    reply({ success: true, pids: pids })
  })
}
```

Snippet 42. Request handler for fetching all available sets of PID values.

```
/* Method: GET - Get PID parameters by ID */
exports.getPidById = (request, reply) => {
  Models.Pid.findOne({ _id: request.params._id, type: request.params.type }, (err, pid) => {
    if (err) {
      reply({ success: false, error: err })
    }

    reply({ success: true, pid: pid })
  })
}
```

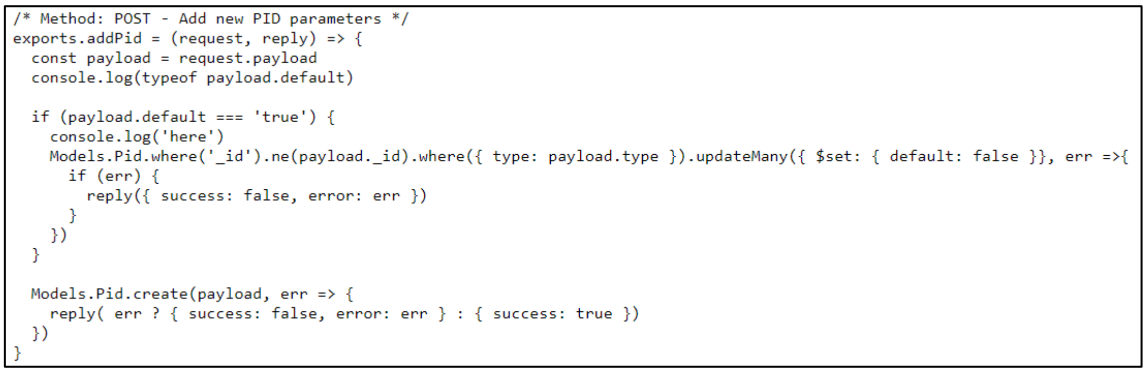
Snippet 43. Request handler for getting the default set of PID.

The function finds the set of PID by building a query with a default value of true, and select only 4 pieces of data: proportional, integral, derivative and type. Then executing the query and reply with found set of PID.

```
/* Method: POST - Set PID parameters as default */
exports.setDefaultPid = (request, reply) => {
  Models.Pid
    .where('_id').ne(request.payload._id)
    .where({ type: request.payload.type }).updateMany({ $set: { default: false } }, err => {
      if (err) {
        reply({ success: false, error: err })
      }

      Models.Pid.where({ _id: request.payload._id, type: request.payload.type }).update({ default: true }, err => {
        if (err) {
          reply({ success: false, error: err })
        }

        reply({ success: true })
      })
    })
}
```

Snippet 44. Set a default set of PID.

The request handler accepts a set of PID ID as a parameter, then looks for the other sets of PID and set all their default attribute to false. Then it searches for matching PID ID and set it as default.



```
/* Method: POST - Add new PID parameters */
exports.addPid = (request, reply) => {
  const payload = request.payload
  console.log(typeof payload.default)

  if (payload.default === 'true') {
    console.log('here')
    Models.Pid.where('_id').ne(payload._id).where({ type: payload.type }).updateMany({ $set: { default: false }}, err =>{
      if (err) {
        reply({ success: false, error: err })
      }
    })
  }

  Models.Pid.create(payload, err => {
    reply( err ? { success: false, error: err } : { success: true })
  })
}
```

Snippet 45. Adding a new PID.

The request handler first checks whether the new PID is set to be the default. If it is, the handler will remove the previous default set of PID. The handler then creates a new set of PID with given values.

- **The graphs window**

The graph view is a simple view with options on top of the view to select an operated program. The options are displayed as a program name with its operated date time. The data related to the running program is stored in a text file. Whenever a program starts, the server will create a new text file, named after the ID of the history object. In the text file, first line stores information regarding the running program, for instance, name, number of steps, number of cycles. After that, each row stores information of the chamber in order: current cycle, current step, total time running in second, total time running of a step in seconds, chamber dry temperature, chamber wet temperature and chamber humidity. Later, when choosing a history of a running program, the matching file will be read and parse its data into a variable and sent to the front-end for processing.

6 TESTING

The development of this application was conducted later than the scheduled timetable of the thesis application paper, which is during the summertime when the supervisors were away on summer breaks. Therefore, it was impossible to access the chamber during the development. Instead, a pre-built desktop application made by Simachew Tibebu, replicating the functionality of the chamber was used for testing purposes.

The simulation program stimulates the message of the chamber, with a generated value of temperature and humidity. The stimulation program and the application were run on the same computer and each connected to one serial communication port.

The application was first tested with the communication with the chamber and the database. For each stage of the development, the application was continuously tested with the stimulation program to make sure the different functionalities of the application work with the chamber. However, considering that this thesis work has been conducted in a completely new method and technology, the final testing is not necessary at this stage and furthermore, open a new topic for others to continue to fully develop the system.

7 CONCLUSION

This application was developed as an upgrade for the previous thesis work from the other students. The pre-study was carried out in order to have a deeper understanding the chamber. The choice of technology was inspired by another project which use NodeJS to implement control of a drone.

Finally, the application has delivered a server that can communicate with the chamber and a user-interface providing control to the system. Even though this project's goals were not achieved, the development of this application proves the ability to conduct embedded programming with JavaScript and serves as a good study work.

Future work

Because of the lack of interaction with the actual chamber, the application certainly has much room for improvement. Security for the system is one of the most critical points that has not been implemented yet. There are more of the interaction with the chamber that is yet to be analyzed and implemented, such as PID parameter validation happens when setting PID values, the chamber will have its validation on the values before re-cording to the database.

8 REFERENCES

Mongoose introduction - http://mongoosejs.com/

Node-Serialport npm package introduction - https://www.npmjs.com/package/serialport

HapiJS - https://www.npmjs.com/package/serialport

Setting up your front-end development environment with webpack and hapi - https://medium.com/@tkh44/setting-up-your-front-end-dev-environment-with-webpack-with-hapi-b352ab8b2f9c

The JavaScript runtime explained - https://www.infoworld.com/article/3210589/node-js/what-is-nodejs-javascript-runtime-explained.html

Node.js – Event loop - https://www.tutorialspoint.com/nodejs/nodejs_event_loop.htm

Stylus introduction - http://stylus-lang.com/

The Node.js event loop, timers and process.nextTick() - https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/

Overview of Blocking and Non-blocking - https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/

MithrilJS introduction - https://mithril.js.org/

ChartJS - http://www.chartjs.org/