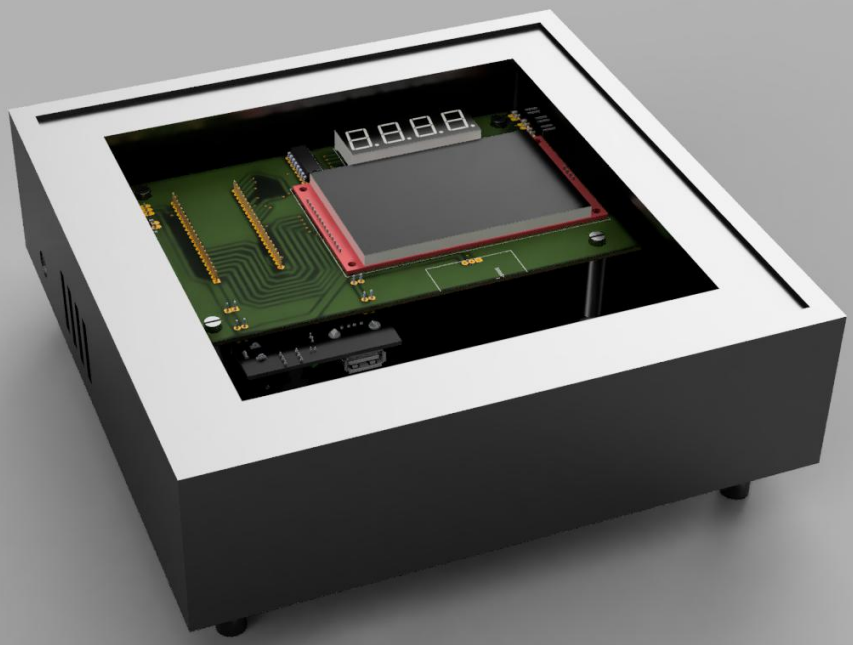
Final Documentation

Aaro Esko, Eeva-Kaisa Kohonen, Arttu Yli-kujala

May 2024



Contents

1 Preview

This is our project on making a nice home decor that blends in to it's environment and displays daily key information that you need without needing to check your phones different apps. In this document we go over all the parts of making our product and give information and codes so that anybody can follow our steps and make this product themselves and expand on this idea.

So our idea is to make a nice looking and small smart mirror on which we can display weather, buss schedule, notes and have a morning alarm, which of course is none other than Phil Collins - 'Another day in paradise'.

We go over on creating 3D model, electronic schematics, PCB, etching and putting this all together. Then we go over coding the ESP32S2 using Arduino IDE and then we move on the server side code.

2 3D modelling

The 3D modelling started from basic idea to make a clean box that had a couple panels, so that we could easily access the insides during debug. I started creating this design in Fusion 360 since it's a nice CAD program and it has free license for students.

2.1 Base frame

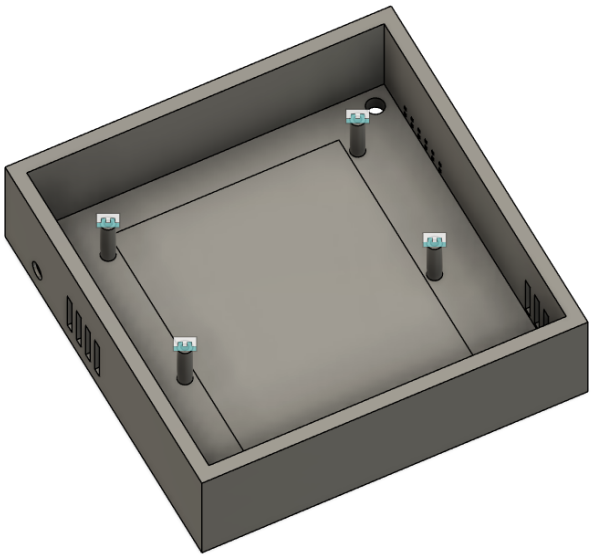


Figure 1: Base frame

So I created a basic frame that has some holes for ventilation, buzzer and buttons(fig. 1). On the inside it has 4 poles on which top we will place our PCB once it's made. The poles have a 10mm hole on the center where we can place M3 heat-set insert to which we can later screw in M3 bolts to secure the PCB. These poles are highly elevated since we will put components on both sides of the PCB. Also this model has a removable back lid which can bee seen in that same figure on the bottom of the inside. Also one more hole can be found on the bottom through which we can put our power cable.

2.2 Lid

Then also on the inside we have 4 holes. One in each corner which will hold our lid.

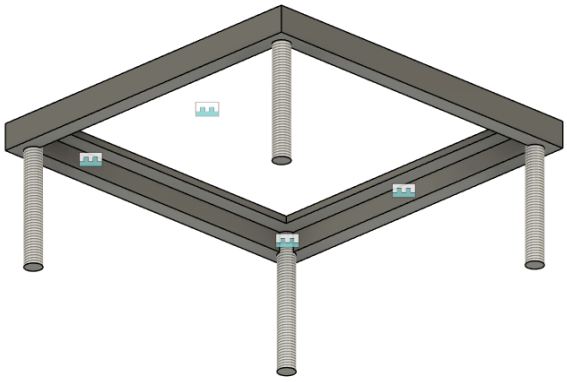


Figure 2: Lid

This image is the lid that which 4 poles are inserted in the 4 corner holes of the base frame (fig. 2). They are made to fit very precisely so that we get a stable and tight fit. The small indent on which the acrylic panel with the tint is placed. Then the 4 poles are inserted as far as wanted into the base frame so in the end the acrylic panel sits tightly over the PCB and components. This lid configuration makes it easy to modify our setup and no hard placement with glue or anything else is needed since everything is adjustable.

2.3 Back lid

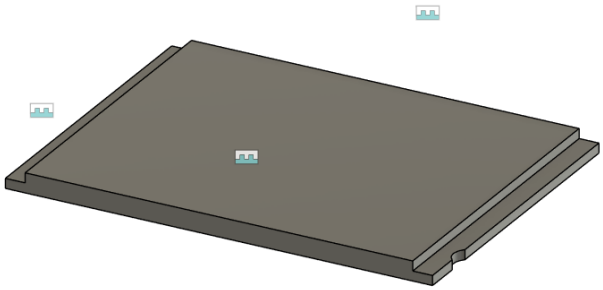


Figure 3: back lid

The back lid (fig. 3) doesn't contain anything special. It is just made for easy access to the bottom side of the PCB where in our design we will fit the power supply module and the ESP32S2.

2.4 More

For more precise information I have included this all to our GitHub page. It includes the Fusion 360 files for the whole project and also .STEP files for the model.

The model can be converted straight into a .stl file to be printed on the 3D printer on most cases but I recommend using other slicers like Cura to slice the file before printing because your Fusion 360 design might have weird meshes sometimes that slicers like Cura can fix automatically.

3 Schematics

This schematic (fig. 4) was also made using Fusion 360. Parts included in the schematics are:

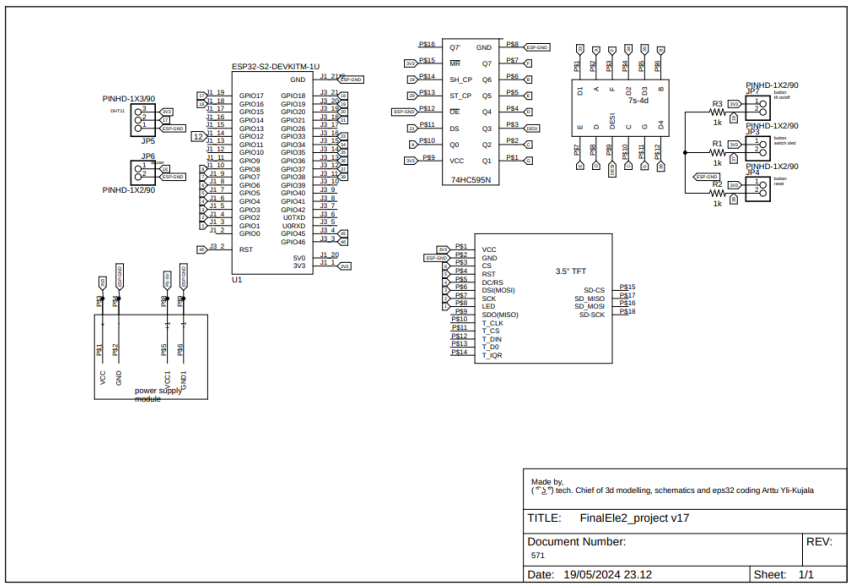


Figure 4: Schematics

3.1 Parts

- ESP32-S2-devkit
- MB102 Power supply module
- basic buzzer
- 3× basic buttons
- 3× 1k ohm resistor
- DHT11 module
- SN74HC595N shift register
- 3.5” TFT SPI LCD screen
- 4-digit 7-segment display

which most can be found from the Arduino starter kit.

3.2 Components

On the schematics we can see all the connections between components clearly. Starting with the ESP32-S2 we can see that we are left with 15 GPIO pins to spare so you can add some more components in your design.

3.2.1 ESP32S2 pins

The pins from 1-6 go to the LCD screen.
Pin 16 is for the buzzer and pin 17 is for reading DHT11 data.

Pin 45 doesn’t need to be connected to the reset pin of esp32 if you want to make the reset work through programming like we ended up doing.

Pins 38, 37 and 18 are pins for the three buttons because these left more space for choosing trace paths on the PCB design.

Pins 19-21 control the shift register and and pins 33-36 control the 4-digit 7-segment digit pins. As later note I recommend to use an extra pin for the 4-digit 7-segment decimal pin rather than connecting it to the shift register like we did. Because of this I ended making two different functions in the esp32 code which is not bad but can be made simplier when connecting this decimal directly to esp32.

3.2.2 MB102 Power supply

The Power is supplied by the MB102 power supply. This supply is limited to 700mA current. Because this we ended having small problem when we plugged everything on the PCB since the current wasn't enough sometimes. But there is no reason to panic since the power supply also has a USB port that we the power supply is on we can but a USB cable to go from this port to the ESP32 so it can provide separate power for the ESP32. The ESP32-S2 can be either supplied through the Vin pins or the USB cable and has internal components that even though both the USB cable and Vin are connected it will use the USB cable to supply itself. So you can also remove the connection of 3.3v from the power supply to the ESP32S2 if wanted and just plug the USB cable. This power supply is good since it's easy and can also provide 5V if needed.

3.2.3 LCD screen

The LCD display uses ILI9488 driver and the back light consumes about 90mA and while drawing we estimate about 160mA from sources so it's not too much. It's nice 320×480 3.5' inch display that can be supplied with only 3.3V.

3.2.4 DHT11 module

For reading room temperature we used DHT11 module because it's easy to use and calibrate and supports 3.3V supply.

3.2.5 Shift register and 4-digit 7-segment display

Shift register and 4-digit 7-segment display are simple and the connections can be seen clearly on the schematic picture. Using shift register you save 8 ESP32 GPIO pins for other purposes. These all work with 3.3V supply and on my opinion no extra resistors are needed that you can usually find on 4-digit 7-segment projects.

3.2.6 buttons

The buttons are basic electronic buttons that can be found in any starter kit and website. The holes in the 3D design are specially made so that the 4 different prongs of the buttons fit trough these perfectly. The connections can be seen on the schematics clearly and the three resistors are used as pull down resistor. I just used 1k resistors but you can use any resistor above 220 ohms and bigger the better.

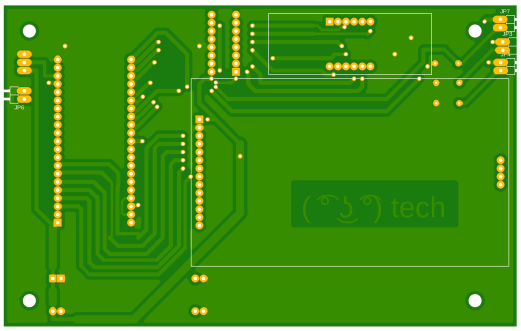
3.3 More

You can find more information on the schematics on our Github page.

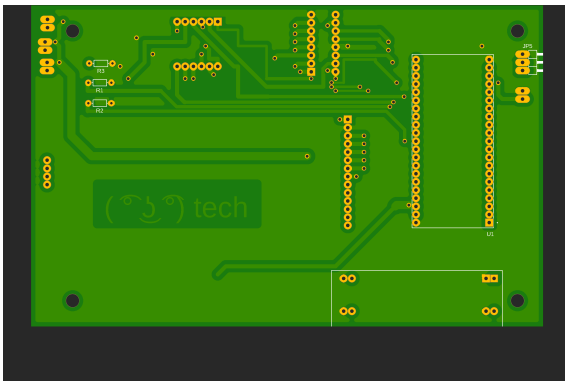
And also on making this all in Fusion 360 you will find out that the Fusion 360 doesn't include most of these parts itself but you have to create the footprints and 3D models yourself. This is not hard but rather simple stuff, but you need to be careful about the measurement making everything precisely as they are. The components don't have to have 3D models but it's not hard since most of them can be found on https://grabcad.com/library web page where you can find and import the .STEP files for the components to Fusion 360 or you can create your own 3D models.

4 PCB

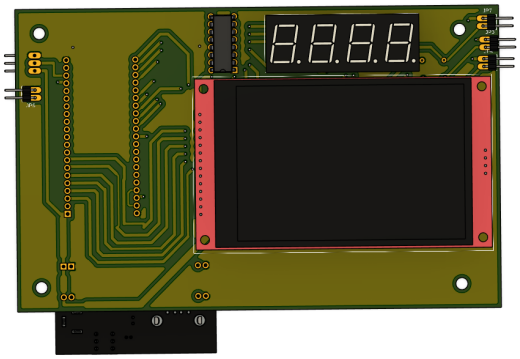
The PCB design was also made simply with Fusion 360 since it has all the things we need and also it's easy to configure the board measurements.



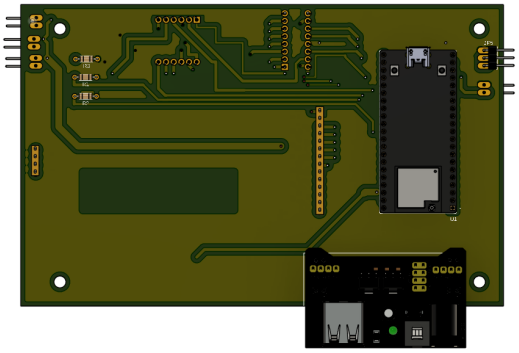
(a) Topside CAM of PCB



(b) Bottom CAM of PCB



(c) Topside 3D model



(d) Bottom 3D model

On these figures (fig. 6.5) we can see the PCB model CAM photos and the 3D models. As you can see the ESP32S2 and power supply module is placed on the bottom side of the PCB. This is made so that we have easy access to these parts when we remove the back lid so that debugging can be made easily and also these parts have LED lights so when they are on the backside they don't shine through the acrylic panel tint.

4.1 Traces

As you can see in the CAM pictures the trace width varies a depending how much space there is. They vary from 0.6-2.54 mm. There is no issue on making this all 0.6 if you want or manually tracing them better so that you can make them bigger. Using 0.6 at least didn't show any problems but bigger is always better.

Minimum clearance of 0.3 mm was used between traces and ground planes. This had a good result so that traces weren't too close each other while soldering or etching the board to result any issues.

Also it's good to make sure to have about 5mm clearance between the board outline and any small traces if you etch the board yourself. Big traces, vias and pads are no problem but if small trace happens to travel on this zone, there might be some issues since the board outline usually has some issues.

4.2 Vias

The vias are made to have 0.8mm drill hole so that they are easy to drill and 0.8mm wire can be passed through them and soldered.

4.3 Pads

All the module pads (ESP32S2, MB102 etc.) were made to have 1.1mm pad holes to fit headers so that the components could be easily plugged in and changed if needed. The pad size on the layer is 1mm so that it has good size to put the solder on and that the pads still don't touch each others

The resistor drill holes were size of 1mm to have tighter fit and made the soldering bit easier.

4.4 Screw holes

The screw holes that can be seen on each corner are made to be drilled with 10mm drill and you can then use M3 bolts to fit the board on the base frame (fig. 1)

4.5 Ground plane

It's good to have ground plane everywhere else where there is nothing else. This cools the PCB down and also reduces cross talk.

The ground planes are connected to each other on both sides by vias all around the board to ensure the grounds are properly grounded.

5 Making PCB

5.1 Etching

We etched our own PCB on blank two side copper PCB board. For this you need expose the unwanted parts of the PCB board to UV. Easiest way is to print your design on see through film and this can be done in Fusion 360 on PCB design by selecting wanted layers and then going to document -¿ print tab and make a pdf file that has all the wanted traces as black on the image and the rest be white that you want the UV light to shine on.

5.1.1 Exposure to UV

After you have made the film you can place it on top of the PCB copper plate and put it into a UV light machine or you can also put it under normal table lamp within 5 to 10cm away and keep it there about 9-12 minutes depending on how light sensitive your PCB plate is. Placing the film is precise work because otherwise you will have holes in wrong places.

Then you do the same thing with the other side.

5.1.2 Chemicals

After the UV exposure we have gotten rid of the protective coating on the copper in the unwanted places and now we can use NaOH to make the unwanted copper removable by HCl. So basically after exposure we wash the board in NaOH solution for about 1min and then we wash it in HCl solution until the unwanted copper is removed completely

5.2 Drilling and soldering

5.2.1 Drilling

On this model we used drill sizer 10mm,1.1mm,1m and 0.8mm for the holes. So the drills need to be changed couple times. Another way is to make every hole 1.1mm if wanted. The sizes for each hole are declared in section 4.

5.2.2 Soldering

The major components (ESP32S2, LCD-screen, 4d 7s display, shift register and power supply) are designed to be placed on headers for easy plugin and removing which I recommend doing also.

The buttons, DHT11 module and buzzer are not connected straight into the PCB but in the holes you should put pin headers so later you can place these components somewhere on the 3D model and make the connection with wires.

Only the three resistors are soldered directly as they are on the PCB.

6 Putting everything together

6.1 3D prints

So now that everything is done. The 3D models are printed and the PCB is etched we can but everything together. The 3D modelling parts with together tightly with no need for extra holding.

6.2 Acrylic panel with tint

We use acrylic see trough panel as our main panel and put a reflective window tint on it. The tint can be placed with simply with soap and water. Then this panel is fitted in it's whole in the lid indentation and is kept in it's place by precise fitting and by pushing the lid far enough so that the PCB and it's components push on the panel.

Also it's beneficial to add a black carton behind the panel and cut it precisely to ensure no unwanted light is getting through the panel from the inside of the box

6.3 PCB placement

For placing the PCB you should firstly place M3 heat-set inserts into the base frame poles with soldering iron. Then place the PCB on top and use M3 screws to screw in the PCB tightly.

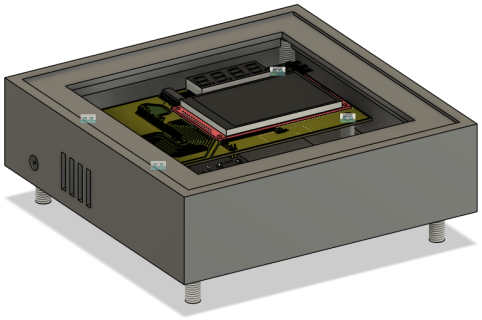
6.4 Buttons, DHT11 module, buzzer and wiring

Now that major components are put together you can place the buttons on the outside right wall where you can find 12 small holes. In these holes the simple button legs can be fitted and you can plug female-to-female wire to these so they can be plugged in easily to the PCB. Then neatly taping this wires tightly you can make tension so that the buttons stay in place and are easily removable without needing to glue these buttons.

DHT11 module should be placed on either side by the ventilation holes. You can either tape it or glue in it's place. Then using female-to-female cables you can connect it to the PCB.

Lastly the buzzer can be fitted in the left side wall small circular hole that is measured precisely for it so no glue is necessarily needed. This is also then connected with female-to-female cables.

6.5 Final looks



(e) Final 3D model look

(f) Picture

For more information about the final looks you should check our GitHub page presentation and videos.

7 ESP32S2 coding

My ESP32S2 code can be found on our GitHub page. It is commented well so I will only go through some major points on this documentation.

7.1 Libraries

We used some libraries gotten from the Arduino IDE library tab and they are:

- WiFi.h
- HTTPClient.h
- Arduino_JSON.h
- SPI.h
- TimeLib.h
- pitches.h
- PNGdec.h
- DHT_Async.h
- TFT_eSPI.h

7.1.1 WiFi.h

Basic library that provides the ESP32 S2 with WiFi capabilities for wireless connection and such

7.1.2 HTTPClient.h

With the WiFi connection this library enables the ESP32S2 to send HTTP requests like API.get and POST so we can recieve and send data to our server

7.1.3 Arduino_JSON.h

This library is used to parse and generate the data that we send and receive from our server and display all this easily.

7.1.4 SPI.h

This provides support with SPI(Serial Peripheral Interface) devices that in our case is the TFT SPI LCD screen

7.1.5 TimeLib.h

This is used for timekeeping. With this we can set our ESP32S2 keep time like a clock which we can then display on the 4-digit 7-segment display. The time can also be easily set from server once per ESP32S2 boot to your local timezone. With this library we don't need a RTC module.

7.1.6 pitches.h

This includes all the musical notes defined as certain tone for our buzzer. There are different pitches.h libraries out there so we provide the one that we used in our github page which is linked in beginning of this section.

7.1.7 PNGdec.h

This library is for decoding PNG images so that they can be displayd on the LCD screen.

7.1.8 DHT_Async.h

This library is used to read the data from DHT11. This specific library is used because it consumes less time than the basic DHT.h library.

7.1.9 TFT_eSPI.h

This is the main library used for displaying anything on the LCD screen. For this you have to be careful because you need to make some changes in the library folder User_Setup.h. We have provided the changed file in our GitHub page linked in the beginning of this section, but mainly you have to change the pins to your configuration and the driver should be ILI9488.

7.2 Weather pngs

When you go to our GitHub page you will also see lot of .h files called cloudy, rain, etc. These are byte arrays for the weather pictures that are printed on the LCD screen. This is where we need the PNGdec.h library and you should also include these (if wanted) in your file like they are provided in our page.

7.3 The rest of the code

The code is neatly packed into different functions and everything is commented. The functions are rather simple ones so I hope it's clear. Only thing that is not well optimized is the drawing of the weather pngs but otherwise it's neat coding.

8 Server

Big idea behind having this server-side implementation is to be able to do more complex data-processing there while also saving space on the microcontroller itself. Also this section covers the user-interfacing aswell. In this project the server-part consists of two main parts: the hidden back-end side (server) and the visible front-end side (website). In this section some main aspects of the code are introduced but more details on the code itself can be found in the source code comments.

8.1 Back-end

8.1.1 Development framework

Back-end side is managed by the server code which in this case was chosen to be done with python utilizing Flask web framework. Many other alternatives exist like Django or Express.js, but Flask was chosen because of it's simplicity and having a little bit personal exposure to it beforehand. Using these frameworks for server-side development is a standard, they provide lots of libraries, functions and easy-to use syntax for development. With Flask listening to a route and returning html template is simple as:

```
@app.route('/index')
def index():
          return render_template('index.html')
```

8.1.2 Database

In this application SQLite3 was used to maintain some very simple data. This was again chosen for it's simplicity and personal experience with this database engine. The capabilities of this database engine go far beyond what is used in this project, but there was no downside for using this instead of storing information e.g. in .txt or .csv -files. Using an actual database also makes it possible to scale the project and further develop more complicated data management.

The databases were first created in terminal. After making sure SQLite3 is actually installed on the system one can access SQLite3 interpreter with terminal. Database for current stop, shown in table 1, was created in terminal as follows:

1. Create new database and access it

```
>sqlite3 current_stop.db;
```

2. Create new table with

```
sqlite>CREATE TABLE stops (id INTEGER PRIMARY KEY, gtfsId TEXT,coordinates TEXT);
```

Now there is a database created with columns id,gtfsId and coordinates. They have datatypes int,string,string respectively. Adding information to database is done in the server code.

```
    conn = sqlite3.connect('/path/to/current_stop.db')
    cursor = conn.cursor()
    cursor.execute('UPDATE stop SET gtfsId = ?, coordinates = ?',
    (new_gtfsId, new_latlon))
    conn.commit()
    cursor.close()
```

First there is connection made on the database and cursor object setup for executing commands then method execute() is used to perform commands similarly as one would with the terminal interpreter. New values are marked with in the command and later set in the second parameter of execute() method. One important thing here is the commit() method which actually inserts the values. Cursor is closed at the end.

```
conn = sqlite3.connect('/path/to/current_stop.db')
cursor = conn.cursor()
cursor.execute('SELECT * FROM stop LIMIT 1')
db_info=cursor.fetchone()
current_gtfsId =db_info[1]
current_coordinates=db_info[2]
cursor.close()
```

In similar fashion we can get data from there. Data is returned as tuples and here the value from gtfsId column is parsed into current_gtfsId and coordinates into current_coordinates.

id	gtfsId	coordinates
1	HSL:12321	60.2031, 23.96

Table 1: current stop database

8.1.3 Serving for ESP32

Important aspect on the server side was to be able to serve the processed information in a neat format for ESP32. This was implemented by simply creating a specific route to listen and wait for requests from clients.

When the client (usually ESP32 in this project) makes a 'GET' request to the address www.aaroesko.xyz/`main_data?Kumpula` they will be served with JSON-file that looks like this (accessed from browser which makes it look neat):

This is achieved at the server code by the following simplified code:

```
@app.route('/main_data')
def main_data_function():

    data = {
"time_data":time_data,
"public_transport_data" : departure_dict,
"image_data": "(   )"
"weather_data": weather,
"note":note,
"secondary_weather_data":secondary_weather,
"dht_data":latest_dht_data,
"events":new_event_dict,
    }
```

```
    return jsonify(data)
```

Basically the data is wrapped in dictionary and then returned as JSON for the client who invokes the function by making a request to the address www.aaroesko.xyz/main_data?Kumpula.
The question mark and 'Kumpula' at the end of the route are something called query parameters which is our way of limiting access to this route.

8.1.4 APIs

In this project two free API services were used: Digitransit (https://digitransit.fi/) and Openmeteo (https://open-meteo.com/). Both provide documentation about their use.

Digitransit uses GraphQl as query language to pull desired information from their system with API request. Simple GraphQL-query to Digitransit could look like this:

```
    {
  stops(name: "hertton") {
    gtfsId
    name
    code
    lat
    lon
  }
}
```

This would perform a query in Digitransit database and look for all stop names that are similar to "hertton". In the response we would receive information from all these stops and their stop id, name, code, latitude and longitude. These queries can be modified by removing different properties like lat or lon or adding others. Digitransit provides information about what properties can be fetched about certain entities (stops, routes etc.)

Following commented example shows how our server code uses similar query.

Listing 1: example code for Digitransit API request

```
def get_stop_id(stop_name:str):
    body = """
    query Stops($name: String!) {
      stops(name: $name) {
        gtfsId
        name
        code
      }
    }
    """

    subscription_key = 'your subscription key'
    api_url = 'https://api.digitransit.fi/routing/v1/routers/hsl/index/graphql'
    headers = {
        'digitransit-subscription-key': subscription_key,
        'Content-Type': 'application/json'
    }

    variables = {"name": stop_name}
    response = requests.post(url=api_url, headers=headers, json={'query': body, '
        ↪ variables': variables})
    json_response = response.json()

    gtfsid_info_dict = {
        stop["gtfsId"]: (stop["name"], stop["code"]) for stop in json_response["data"]["
            ↪ stops"]
    }

    return gtfsid_info_dict
```

First a function is defined which takes a certain stop name string as variable. The body of the query is then defined in which there is a special syntax for adding a variable inside this query. After the body we define essential variables for the request. In the variables dictionary we have our stopname variable. The API request itself happens in the requests.post() function which takes the previously defined parameters and jsonifies the query body. The API response is saved and converted into json format. After that for our purposes dictionary of the stop ids(gtfsid) was created and returned.

Openmeteo

Openmeteo provides ready python libraries for interfacing with their API. After importing the libraries, a request for a daily maximum temperature for the next seven days might look something like this:

```
url = "https://api.open-meteo.com/v1/forecast"
params = {
       "latitude": 60.1695,
       "longitude": 24.9354,
       "daily": "temperature_2m_max",
       "timezone": "Europe/Moscow"
}
responses = openmeteo.weather_api(url, params=params)
response = responses[0]
daily = response.Daily()
daily_temperature_2m_max = daily.Variables(0).ValuesAsNumpy()
```

The last variable now hold daily maximum temperatures for the next 7 days as numpy arrays. Similarly to as in the Openmeteo documentation, in the actual server code we have converted these into pandas (powerful data manipulation library) dataframe structures in the for more convenient data processing.

8.1.5 Testing APIs and server

In our case testing the workings of our main data route was simple because you could access it with browser just by simply typing the whole URL there. Often however there needs to be a full request body or the request method is something different than simple 'GET' request. In this case one might want to use some other software, for example Postman to easily send HTTP requests and view responses.

Following pictures show how Postman (VScode extension) looks when sending Post requests for changing DHT data in our own server and API-request to Digitransit.

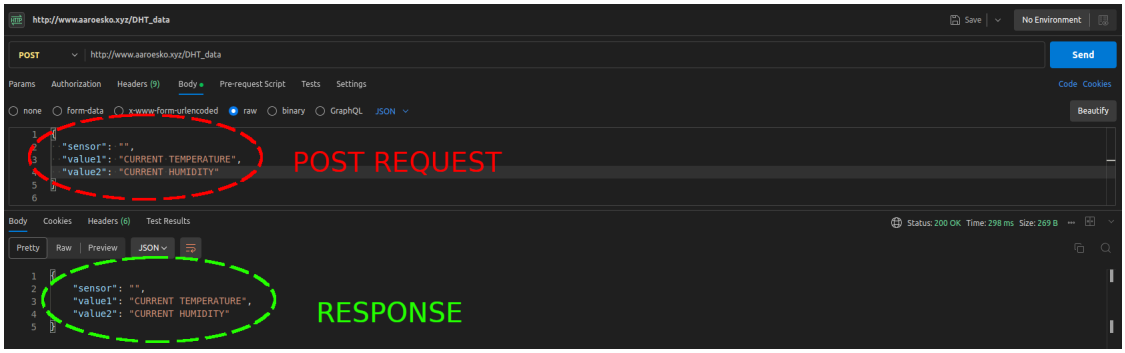


Figure 5: Sending post request to our route that accepts updates in the DHT-data. Similar request is sent from ESP32 to get the actual data. The request body is entered in raw JSON format and this is simply sent to the URL defined at the top. Response is shown at separate panel below the request. In this case everything went ok, status 200 was received and the response looks exactly as we defined in the post request.

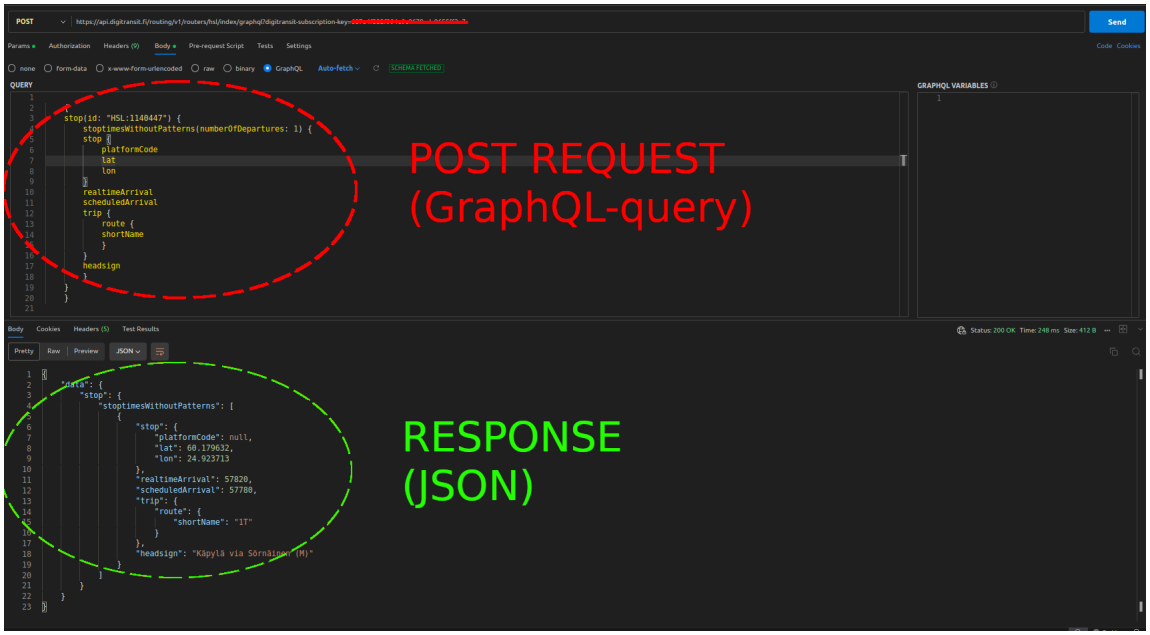


Figure 6: Sending request to Digitransit. No actual changes are made to the server despite Digitransit required POSt request. The request is also send as GraphQL-query and not JSON because Digitransit prefers this. Also the route now must have their subscription key included.

8.1.6 Libraries

Other libraries used in the server code are briefly introduced this table 2

Library	Explanation
Flask	A lightweight WSGI web application framework in Python, used for building web applications and APIs.
functools	
datetime	Working with dates and times.
requests	
openmeteo_requests	for Openmeteo API requests
requests_cache	Caching for HTTP requests
retry	A Python library (retry-requests) that adds retry logic to HTTP requests made using the requests library.
pandas	data manipulation
pytz	A Python library for accurate and cross-platform timezone calculations.
numpy	A fundamental package for scientific computing in Python, providing support for arrays and mathematical functions.
sqlite3	A Python module for interacting with SQLite databases, a lightweight, disk-based database engine.

Table 2: List of Libraries and Their Explanations

Front-end

Front-end includes the visible website and it's functionalities. Website is served to the client when they access certain route. For example:

```
\@app.route('/')
```

```
    def index():
        return render_template("index.html")
```

In Flask these html files are called templates and they are located in templates directory at the server. In our web application there is only single template to be served, main page (index). This index.html contains references also to css and javascript files giving the site it's style and dynamic functionalities, respectively. In our actual code the also is @login_required decorator which ensures the user has signed in before being able to access this route. Let's look at some main features of the main index file.

```
    <!DOCTYPE html>
<html lang="en" class="theme-light">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Screen control hub</title>
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
</head>
<body>
<div class="info-box">
  <h2>Submit a Note</h2>
  <div class="message-form">
    <form id="messageForm">
      <label for="messageInput">Your note:</label>
      <textarea id="messageInput" name="messageInput" rows="4" maxlength="25" placeholder
          ↪ ="Type your note here (max. length 19 chars.)..."></textarea>
      <button type="submit">Submit</button>
    </form>
  </div>
</div>
<script src="{{ url_for('static', filename='main.js') }}"></script>
<script src="{{ url_for('static', filename='calendarscript.js') }}"></script>
</body>
</html>
```

Code is structured using 'div' elements. This code shows structure of the note section but html code also includes divs for alarm, public transportation etc. All elements that we want to access with javascript or css need to have their identifier. In here we can see for example "info-box" class identifier and "messageForm" id. For javascript and css work together with this html code, they have to be loaded by the html code. Css we have loaded in the hed section of the code and script at the very end of the body. This is the standard way of loading css and javascript. The css and javascript codes were located in 'static' directory.

Style is applied to the page with the css file, styling the info-box element could look like this:

```
.info-box {
  font-family: monospace;
  color: #333;
  background-color: #f0f0f0;
  border-radius: 10px;
  box-shadow: 0 5px 5px rgba(0, 0, 0, 0.1);
  padding: 20px;
  margin: 20px auto;
  max-width: 400px;
  overflow: hidden;
}

@media only screen and (max-width: 480px) {
  .info-box {
    max-width: 90%;
    margin: 20px auto;
  }
}
```

Accessing the element happens with .info-box { } syntax and within the brackets different properties can be adjusted. Important part nowadays is having the website responsive which means making sure it looks also good on mobile devices. This is shown in the second part of the code. One good way for probing good settings in this part is to access developer tools in your desired browser and see how the html page look in different window sizes and set the values accordingly.

Finally the javascript part which provides most of the functionalities of the website. Without this the website would be static website with no server communication nor user interaction.

Let's look at important snippet of javascript code that reoccurs in the main javascript code multiple times.

```
    document.getElementById("messageForm").addEventListener("submit", function(event) {
    event.preventDefault();
    var inputData = document.getElementById("messageInput").value;

    fetch("/note-receive", {
        method: "POST",
        headers: {
            "Content-Type": "application/json"
        },
        body: JSON.stringify({ data: inputData })
});
```

At the beginning the messageForm element is selected by its element id and EventListener is added on it. This means that the javascript is now listening to that html element and whether something happens on it. If user clicks submit then the function is invoked. In this case the function reads the value from the input field. Fetch() function is then used to send post request to the server which is listening to the route "/note-receive". This launches a function at the server side updating the notes database:

```
    @app.route('/note-receive',methods=['POST'])
    def update_the_note():
    data=request.json
    new_message=data.get('data')
    update_note_db(new_message)

    return data
```

It also returns the submitted data which in actual code was used for debugging.

8.1.7 Hosting the server

All of this needs to be hosted on a server computer. This means that all the files including server code, database files, front-end files etc. need to be accessible by the server computer. This can be your personal computer, but it needs to be running all the time for the server to actually function. To avoid this we used hosting platform pythonanywhere.com (https://www.pythonanywhere.com) because of personal familiarity with the platform. It allowed us to access their computing power and made it easy to deploy the website online, meaning connection to the domain name system etc. With this approach all of the files were uploaded to their file system which acts similarly to your local computers file system. Your desired domain can be rent from different domain registrar services.

9 Project summary and workload

Nearly all features from the initial plan were carried out into the final product. New ideation and features also came up during the development. In the the end the project consisted roughly n hours of work. Figure 7 shows to contribution matrix of the project and table 3 shows the estimated weekly and cumulative hours worked on this project during the course.

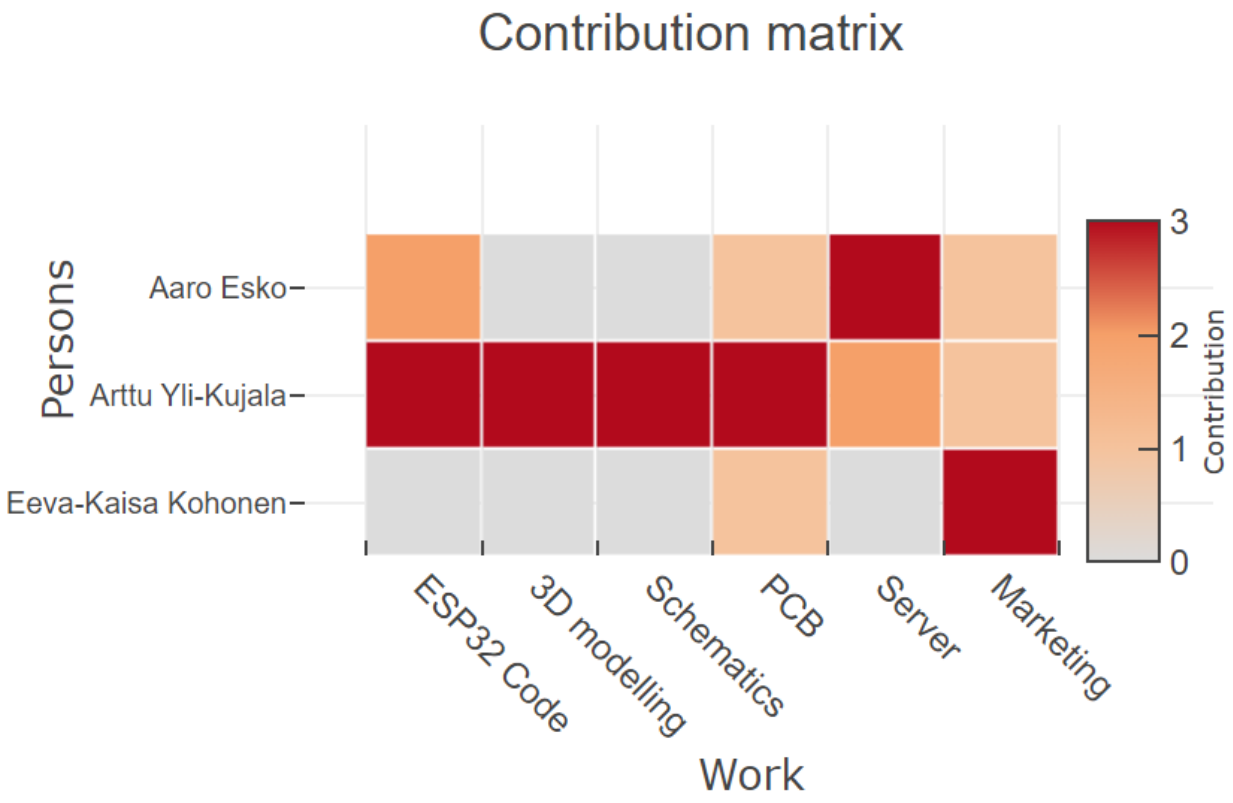


Figure 7: Contribution matrix

person	weekly hours	cumulative
Arttu Yli-Kujala	11	212
Aaro Esko	9.5	170

Table 3: complete work log