Chelsea Case-Miller

Portfolio 2022-2023

Table of Contents

Workplace Project 1

- Temperature Monitor Web Dashboard

Overview

[B8][S9] The building manager and I have been working on getting the HVAC system working to the best standard we can, as many of the employees in the building either state it is too warm or more likely too cold.
The problem with monitoring temperatures was that the existing web interface for the HVAC controller had you go into each unit individually just to check their temperature to confirm or deny the “acceptable” heat of the room. This was a major time waste as the web interface was not the most responsive and at the time, we had upwards of 15 units that we may have needed to check if there were any complaints.

[B9] My idea was to build a simple, effective web dashboard to instantly see all the temperatures with even a 24-hour overview to be able to better judge future temperatures.

Tools

The Tools used during the project were; Pycharm for python and Flask development, VS Code for the HTML templates that were made by my colleague Jay, Edge Dev console to take a closer look at the site we were web scraping from, Gitea for our GIT version management and finally Drone CI for our CI/CD needs.

My approach and work

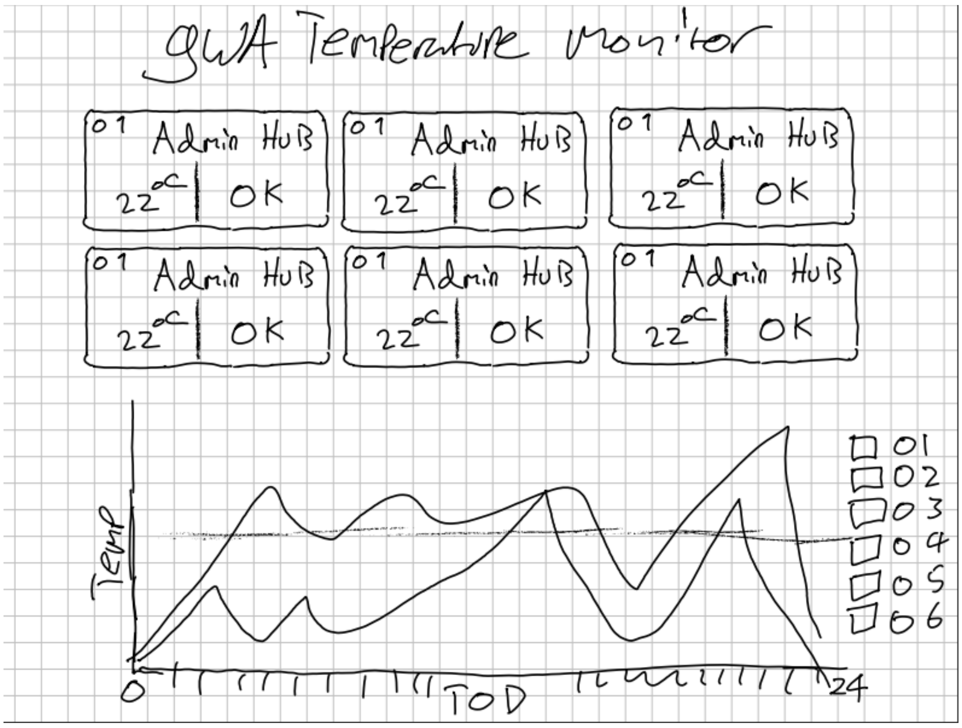
[K5][K1] Creating this project, like many of my other projects, will be developed in an agile manor, this means that the project will follow the following steps:

Plan, Design, Code & Test, Release, Feedback

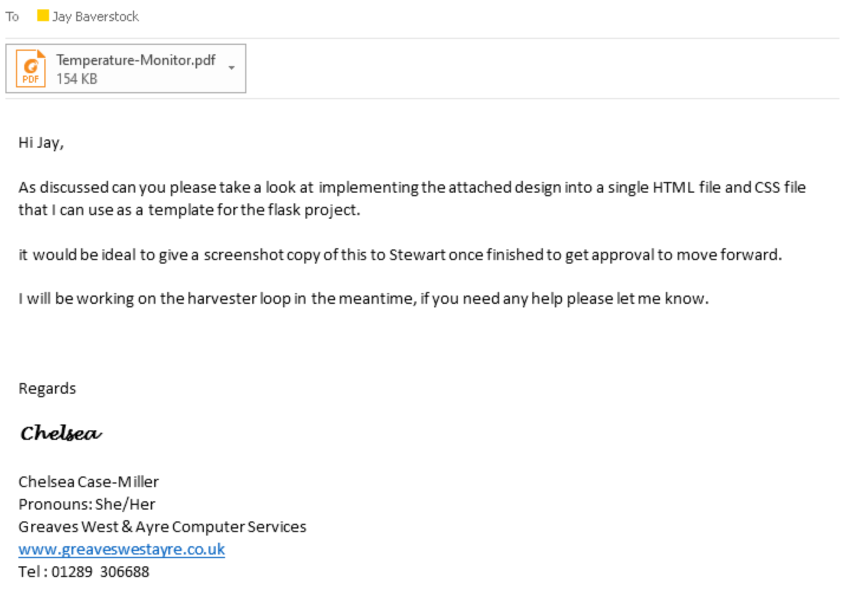
After each round of this loop the next feature or the issues outlined in the feedback stage are then planned, designed and so on continuously or until the stakeholders are happy with the programs release.

The same structure applies to future bug fixing or program maintenance.

[K4, S15] As outlined in my overview, there was a need to have an “at a glance” dashboard for the temperature of each room in the office.
I raised the Idea with the building manager, as they are one of the main people responsible for the HVAC system.
The building manager was happy to proceed with this idea, as it would help them monitor the building day in, and day out.
[B7][S8] I then sketched up an idea of how this may look to the end users and used this as an aid to convey my idea further to the building manager making sure we were all clear on the requirements and end goal of the project.



[B7][B4] After this had been presented I then set up a meeting with my colleague Jay to understand the work needed to be done on the front-end visuals since this fits in with his technical interests of late.


To Jay Baverstock

Temperature-Monitor.pdf
154 KB

Hi Jay,

As discussed can you please take a look at implementing the attached design into a single HTML file and CSS file that I can use as a template for the flask project.

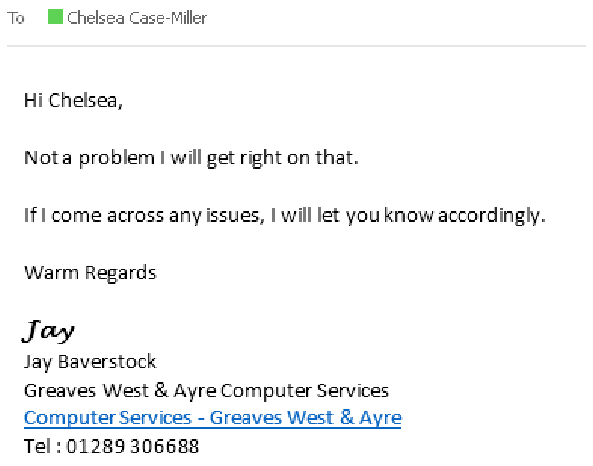
it would be ideal to give a screenshot copy of this to Stewart once finished to get approval to move forward.

I will be working on the harvester loop in the meantime, if you need any help please let me know.

Regards

Chelsea

Chelsea Case-Miller
Pronouns: She/Her
Greaves West & Ayre Computer Services
www.greaveswestayre.co.uk
Tel : 01289 306688


To Chelsea Case-Miller

Hi Chelsea,

Not a problem I will get right on that.

If I come across any issues, I will let you know accordingly.

Warm Regards

Jay
Jay Baverstock
Greaves West & Ayre Computer Services
Computer Services - Greaves West & Ayre
Tel : 01289 306688

[B4][S15][K4] I had sent Jay a follow-up email after our meeting to both give him a copy of the design and to offer my help in-case of any issues they may have faced along the way.

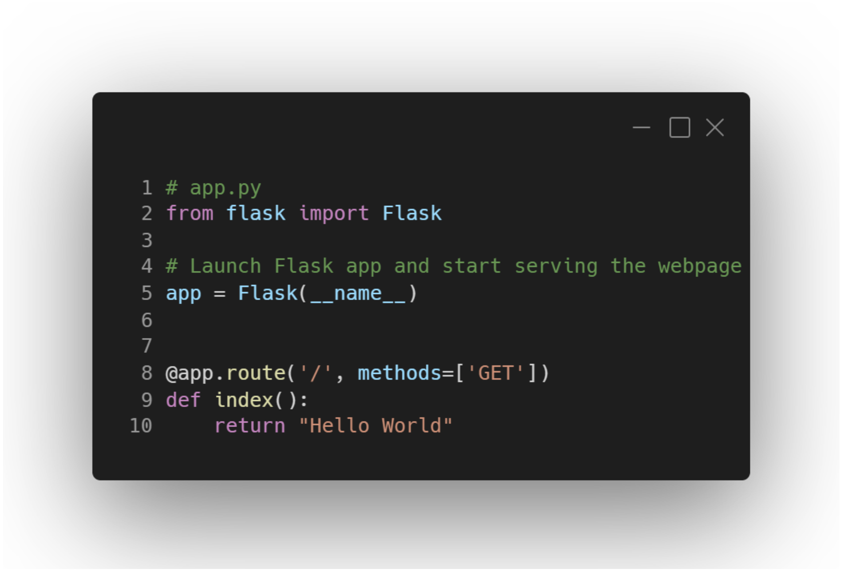
Whilst Jay was working on the front end from the brief and sketch, I had provided I started working on the back end in Python. We used Python with Flask to implement web server request handling so that we could publish this to our web server and be able to instantly view the data of the program whilst keeping the code small, manageable and easy to come back to in case of any issues in the future.

When I was planning the creation of the back end, I had planned some main steps that I knew I would need to take to complete the task, these were:

- Handle requests from the webserver, since this is view only we only need one GET request route.
- Scrape and parse the existing HVAC web page for the correct temperature information.
- Store, retrieve and purge data from the local database.
- Populate the HTML front-end that Jay was working on.

This was my starting list to get an idea of how the project would be structured and the main features I needed to include.

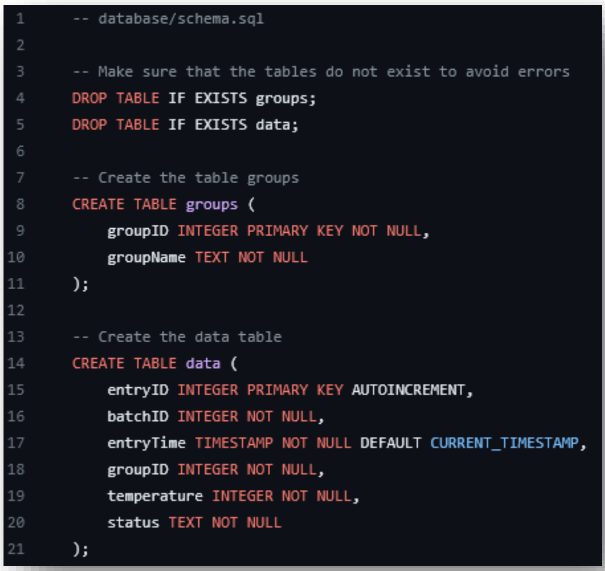
After reviewing my list, I started to create the basic Flask layout to handle an incoming GET request and return a simple "Hello World" to check all was working.



```python
# app.py
from flask import Flask

# Launch Flask app and start serving the webpage
app = Flask(__name__)


@app.route('/', methods=['GET'])
def index():
    return "Hello World"
```

After creating the basic layout of the flask application, the next task I gave myself was creating the database schema and scraping the data from the web page.

[S3][k10] The schema for the database was a simple task of identifying the data I wanted to store and normalising the structure from there.



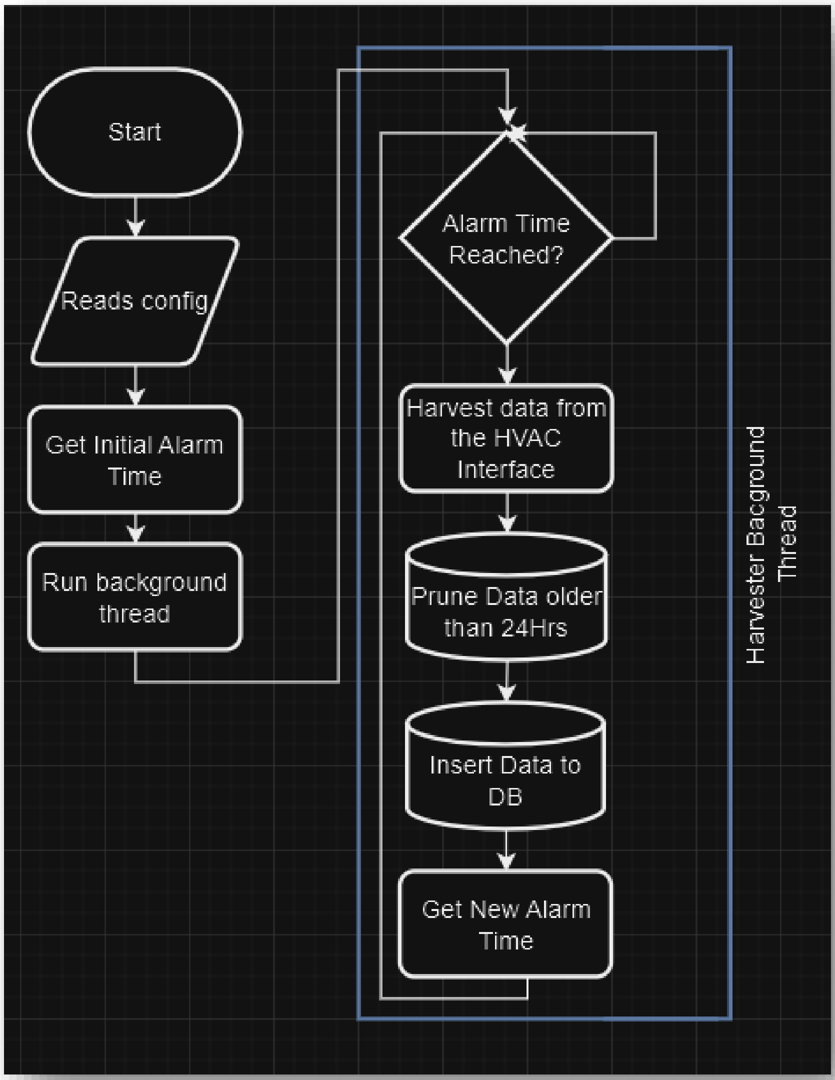
```sql
-- database/schema.sql

-- Make sure that the tables do not exist to avoid errors
DROP TABLE IF EXISTS groups;
DROP TABLE IF EXISTS data;

-- Create the table groups
CREATE TABLE groups (
    groupID INTEGER PRIMARY KEY NOT NULL,
    groupName TEXT NOT NULL
);

-- Create the data table
CREATE TABLE data (
    entryID INTEGER PRIMARY KEY AUTOINCREMENT,
    batchID INTEGER NOT NULL,
    entryTime TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    groupID INTEGER NOT NULL,
    temperature INTEGER NOT NULL,
    status TEXT NOT NULL
);
```

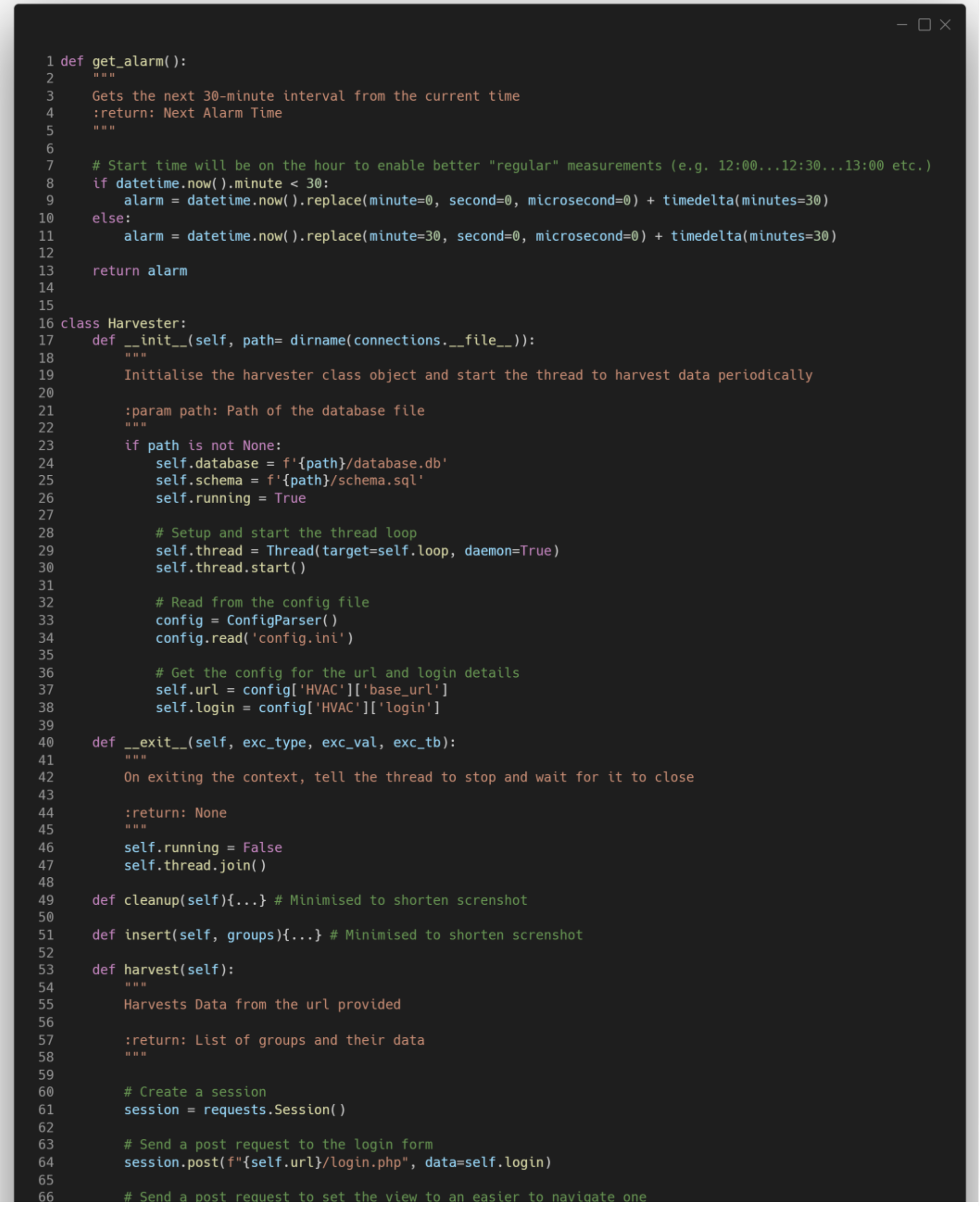
I had identified that the only field that could be normalised was the group name, I therefore added another table for groups and gave each group a corresponding id and used this as a foreign key in the data table, this greatly reduces the database file size as it only needs to store one integer per line of data for the group rather than up to around 15 characters which would add considerable overhead to the storage requirements.

[k7] My approach to the "harvester" was to create a separate module with a class so that I could contain all of the methods relating to harvesting the data, this way the code is easier to maintain and update in the future if the need arises. The use of a class also helps with code-reusability, since everything is contained within the class and class functions there is no need to re-write code more times than is necessary.

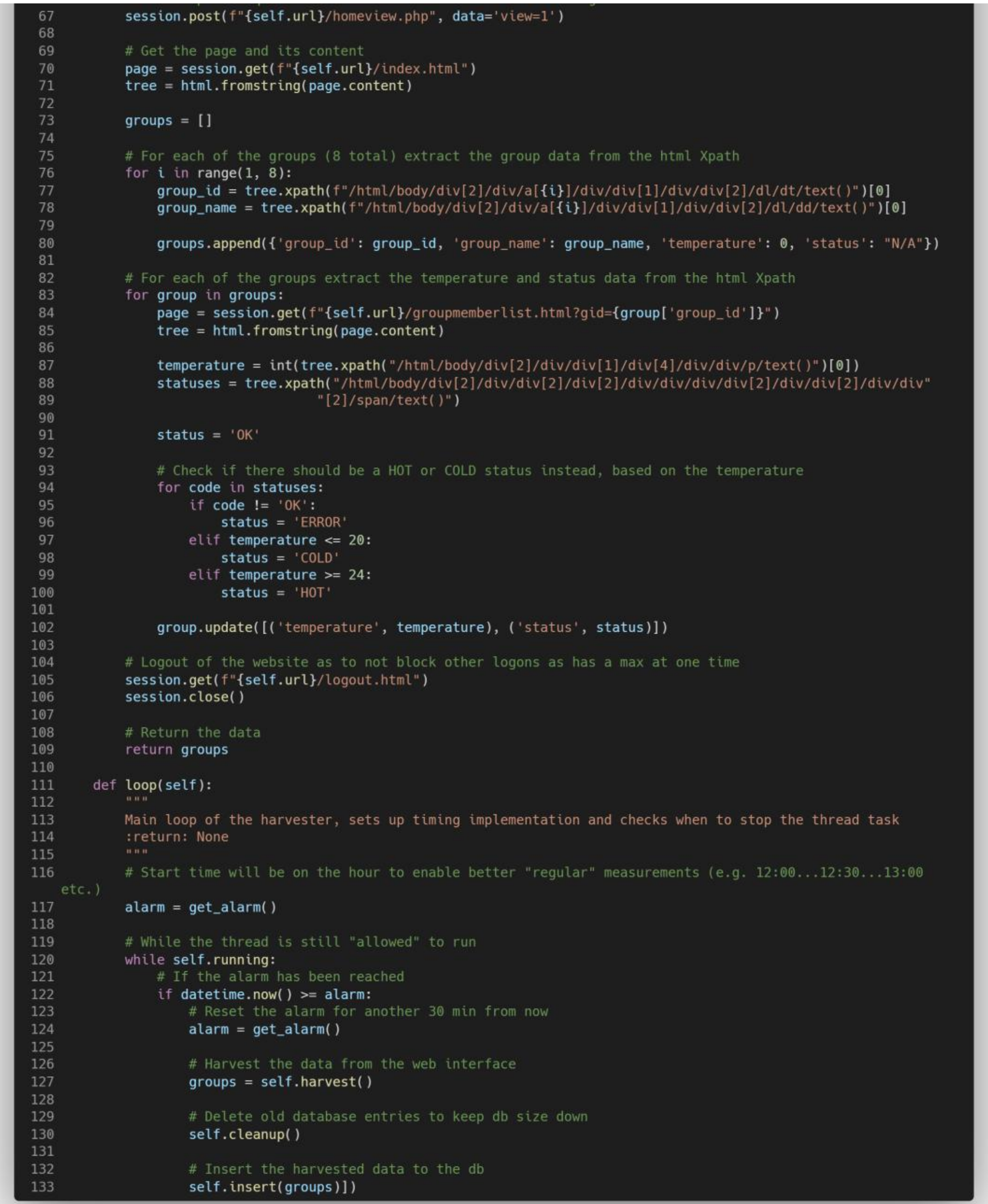
[S8] The basic flow of the harvester is as follows:

1. Harvester reads the config and starts a background thread.
2. The thread loops until the class is forced to stop.
   - 2.1. Gets the next alarm time which gets the next 30-minute interval from the current time (since we want data every 30 mins).
   - 2.2. When the alarm time is hit, the harvester actively collects data from the HVAC interface via requests and getting data from XPath pointers.
   - 2.3. Data that is older than 24hrs is then pruned from the database.
   - 2.4. The newly harvested data is then inserted into the database.
3. The loop then starts again.





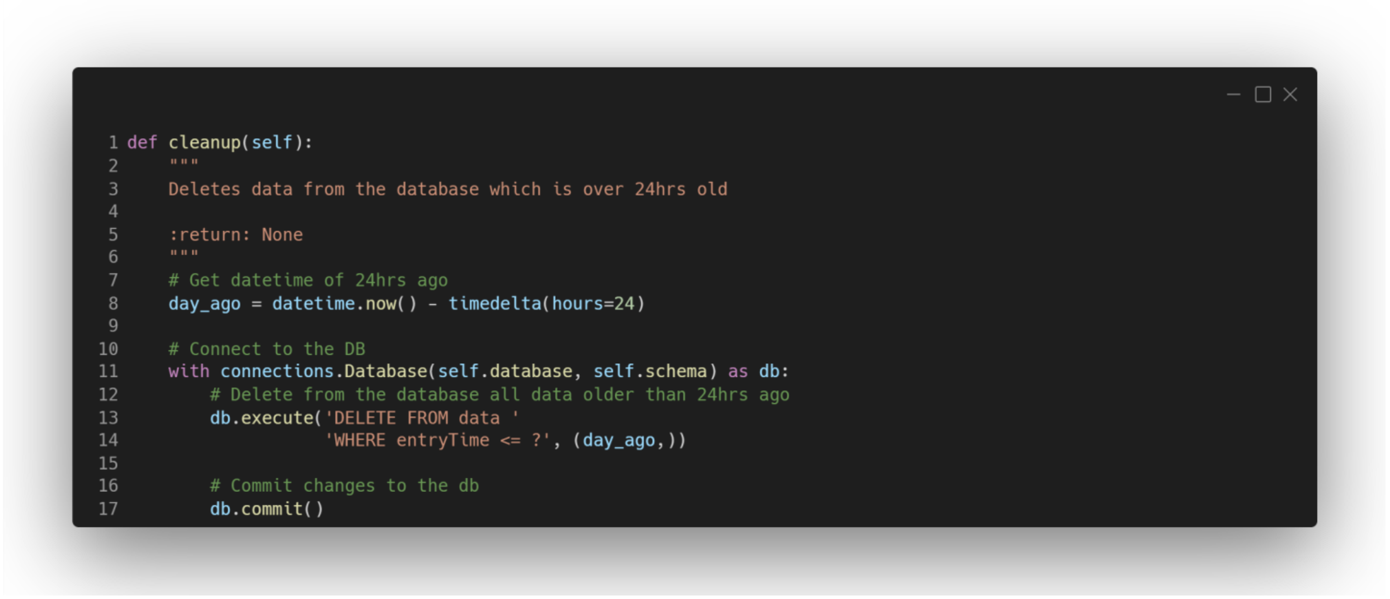
```python
def get_alarm():
    """
    Gets the next 30-minute interval from the current time
    :return: Next Alarm Time
    """

    # Start time will be on the hour to enable better "regular" measurements (e.g. 12:00...12:30...13:00 etc.)
    if datetime.now().minute < 30:
        alarm = datetime.now().replace(minute=0, second=0, microsecond=0) + timedelta(minutes=30)
    else:
        alarm = datetime.now().replace(minute=30, second=0, microsecond=0) + timedelta(minutes=30)

    return alarm

class Harvester:
    def __init__(self, path= dirname(connections.__file__)):
        """
        Initialise the harvester class object and start the thread to harvest data periodically

        :param path: Path of the database file
        """
        if path is not None:
            self.database = f'{path}/database.db'
            self.schema = f'{path}/schema.sql'
            self.running = True

            # Setup and start the thread loop
            self.thread = Thread(target=self.loop, daemon=True)
            self.thread.start()

            # Read from the config file
            config = ConfigParser()
            config.read('config.ini')

            # Get the config for the url and login details
            self.url = config['HVAC']['base_url']
            self.login = config['HVAC']['login']

    def __exit__(self, exc_type, exc_val, exc_tb):
        """
        On exiting the context, tell the thread to stop and wait for it to close

        :return: None
        """
        self.running = False
        self.thread.join()

    def cleanup(self){...} # Minimised to shorten screenshot

    def insert(self, groups){...} # Minimised to shorten screenshot

    def harvest(self):
        """
        Harvests Data from the url provided

        :return: List of groups and their data
        """

        # Create a session
        session = requests.Session()

        # Send a post request to the login form
        session.post(f"{self.url}/login.php", data=self.login)

        # Send a post request to set the view to an easier to navigate one
```



```python
        session.post(f"{self.url}/homeview.php", data='view=1')

        # Get the page and its content
        page = session.get(f"{self.url}/index.html")
        tree = html.fromstring(page.content)

        groups = []

        # For each of the groups (8 total) extract the group data from the html Xpath
        for i in range(1, 8):
            group_id = tree.xpath(f"/html/body/div[2]/div/a[{i}]/div/div[1]/div/div[2]/dl/dt/text()")[0]
            group_name = tree.xpath(f"/html/body/div[2]/div/a[{i}]/div/div[1]/div/div[2]/dl/dd/text()")[0]

            groups.append({'group_id': group_id, 'group_name': group_name, 'temperature': 0, 'status': "N/A"})

        # For each of the groups extract the temperature and status data from the html Xpath
        for group in groups:
            page = session.get(f"{self.url}/groupmemberlist.html?gid={group['group_id']}")
            tree = html.fromstring(page.content)

            temperature = int(tree.xpath("/html/body/div[2]/div/div[1]/div[4]/div/div/p/text()")[0])
            statuses = tree.xpath("/html/body/div[2]/div/div[2]/div[2]/div/div/div/div[2]/div/div[2]/div/div"
                                  "[2]/span/text()")

            status = 'OK'

            # Check if there should be a HOT or COLD status instead, based on the temperature
            for code in statuses:
                if code != 'OK':
                    status = 'ERROR'
                elif temperature <= 20:
                    status = 'COLD'
                elif temperature >= 24:
                    status = 'HOT'

            group.update([('temperature', temperature), ('status', status)])

        # Logout of the website as to not block other logons as has a max at one time
        session.get(f"{self.url}/logout.html")
        session.close()

        # Return the data
        return groups

    def loop(self):
        """
        Main loop of the harvester, sets up timing implementation and checks when to stop the thread task
        :return: None
        """
        # Start time will be on the hour to enable better "regular" measurements (e.g. 12:00...12:30...13:00
etc.)
        alarm = get_alarm()

        # While the thread is still "allowed" to run
        while self.running:
            # If the alarm has been reached
            if datetime.now() >= alarm:
                # Reset the alarm for another 30 min from now
                alarm = get_alarm()

                # Harvest the data from the web interface
                groups = self.harvest()

                # Delete old database entries to keep db size down
                self.cleanup()

                # Insert the harvested data to the db
                self.insert(groups)])
```

[S3][K10] Once I had parsed the data, the next thing on my list was to insert this into the database and purge old data, as we are only interested in the past 24hrs of records.

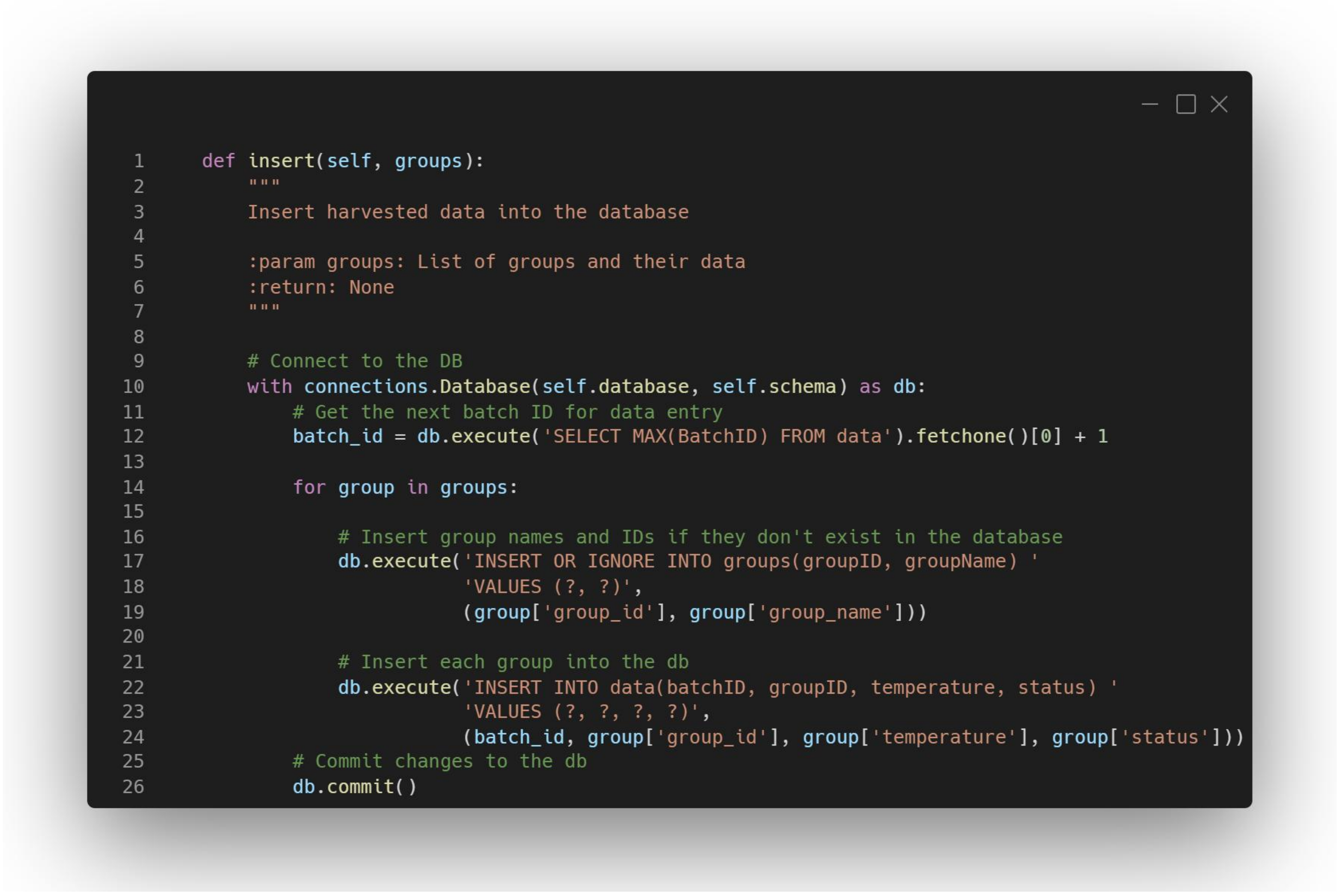
Purging the data was simply handled by a DELETE FROM SQL statement, which I passed in the date of 24hrs previous to the time of running.



```python
def cleanup(self):
    """
    Deletes data from the database which is over 24hrs old

    :return: None
    """
    # Get datetime of 24hrs ago
    day_ago = datetime.now() - timedelta(hours=24)

    # Connect to the DB
    with connections.Database(self.database, self.schema) as db:
        # Delete from the database all data older than 24hrs ago
        db.execute('DELETE FROM data '
                   'WHERE entryTime <= ?', (day_ago,))

        # Commit changes to the db
        db.commit()
```

The insert was a little more hands-on as I had designed the database to be normalised therefore any new groups that may have been added would go into their respective table and the temperature data could then be inserted.

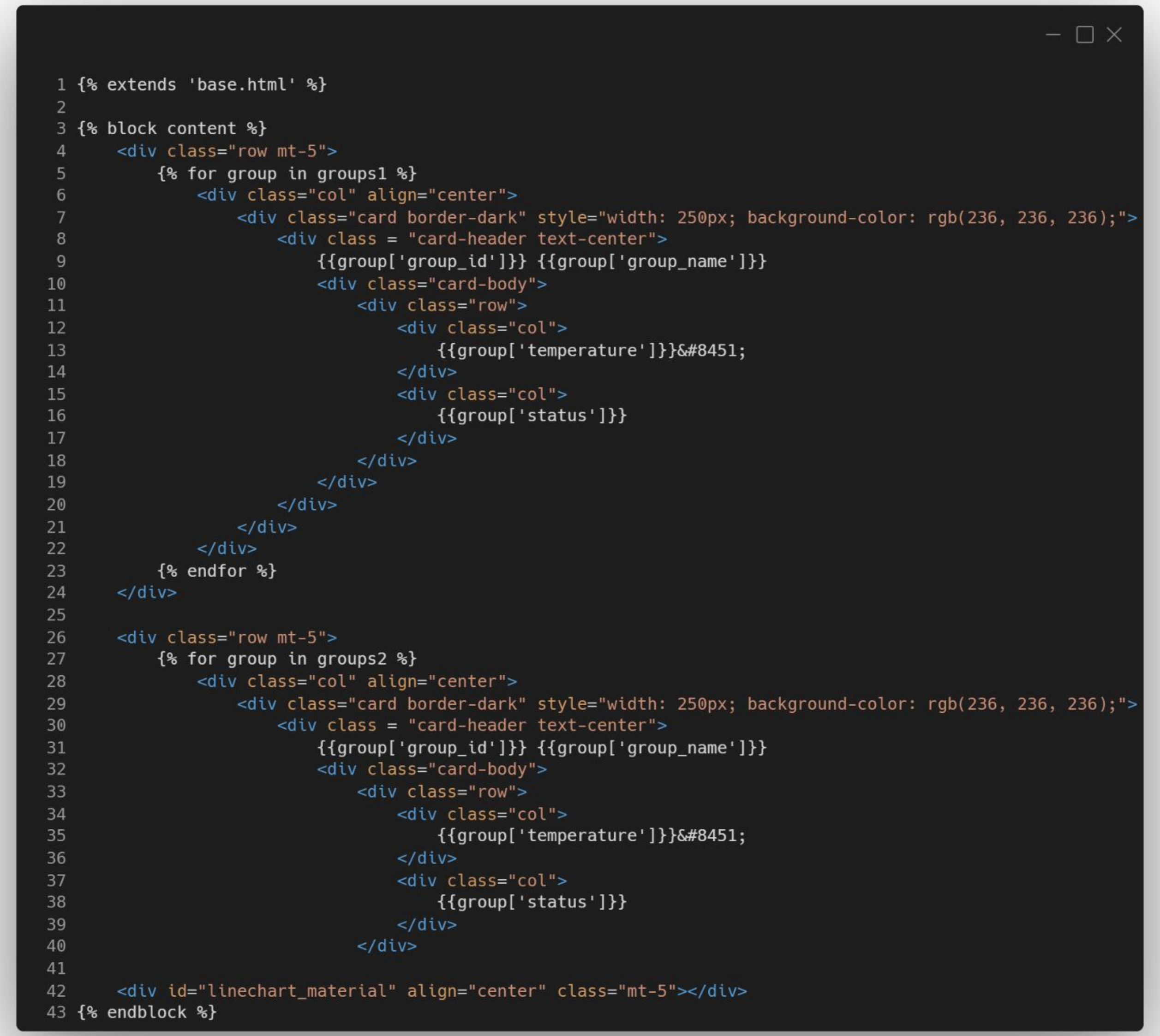
Again, I made sure to pass the variables as parameters in case of someone implementing any kind of SQL injection attacks even though it is an internal webpage I feel it always helps to keep security as secure as any other production-ready code, both for any unforeseen risks and keeping up the practice of security.



```python
def insert(self, groups):
    """
    Insert harvested data into the database

    :param groups: List of groups and their data
    :return: None
    """

    # Connect to the DB
    with connections.Database(self.database, self.schema) as db:
        # Get the next batch ID for data entry
        batch_id = db.execute('SELECT MAX(BatchID) FROM data').fetchone()[0] + 1

        for group in groups:

            # Insert group names and IDs if they don't exist in the database
            db.execute('INSERT OR IGNORE INTO groups(groupID, groupName) '
                       'VALUES (?, ?)',
                       (group['group_id'], group['group_name']))

            # Insert each group into the db
            db.execute('INSERT INTO data(batchID, groupID, temperature, status) '
                       'VALUES (?, ?, ?, ?)',
                       (batch_id, group['group_id'], group['temperature'], group['status']))
        # Commit changes to the db
        db.commit()
```

[S2] Now that the database connections had been developed and tested, the final task on my list was to get Jay's HTML and JavaScript front end and use them as templates for the live data.

To do this I leveraged Flask's ability of HTML templating, using this feature I wrote some small python code, seen in the curly braces, in order to minimize the amount of HTML that may repeat and so that I could insert the correct values from the database.



```
{% extends 'base.html' %}

{% block content %}
    <div class="row mt-5">
        {% for group in groups1 %}
            <div class="col" align="center">
                <div class="card border-dark" style="width: 250px; background-color: rgb(236, 236, 236);">
                    <div class = "card-header text-center">
                        {{group['group_id']}} {{group['group_name']}}
                        <div class="card-body">
                            <div class="row">
                                <div class="col">
                                    {{group['temperature']}}&#8451;
                                </div>
                                <div class="col">
                                    {{group['status']}}
                                </div>
                            </div>
                        </div>
                    </div>
                </div>
            </div>
        {% endfor %}
    </div>

    <div class="row mt-5">
        {% for group in groups2 %}
            <div class="col" align="center">
                <div class="card border-dark" style="width: 250px; background-color: rgb(236, 236, 236);">
                    <div class = "card-header text-center">
                        {{group['group_id']}} {{group['group_name']}}
                        <div class="card-body">
                            <div class="row">
                                <div class="col">
                                    {{group['temperature']}}&#8451;
                                </div>
                                <div class="col">
                                    {{group['status']}}
                                </div>
                            </div>

    <div id="linechart_material" align="center" class="mt-5"></div>
{% endblock %}
```

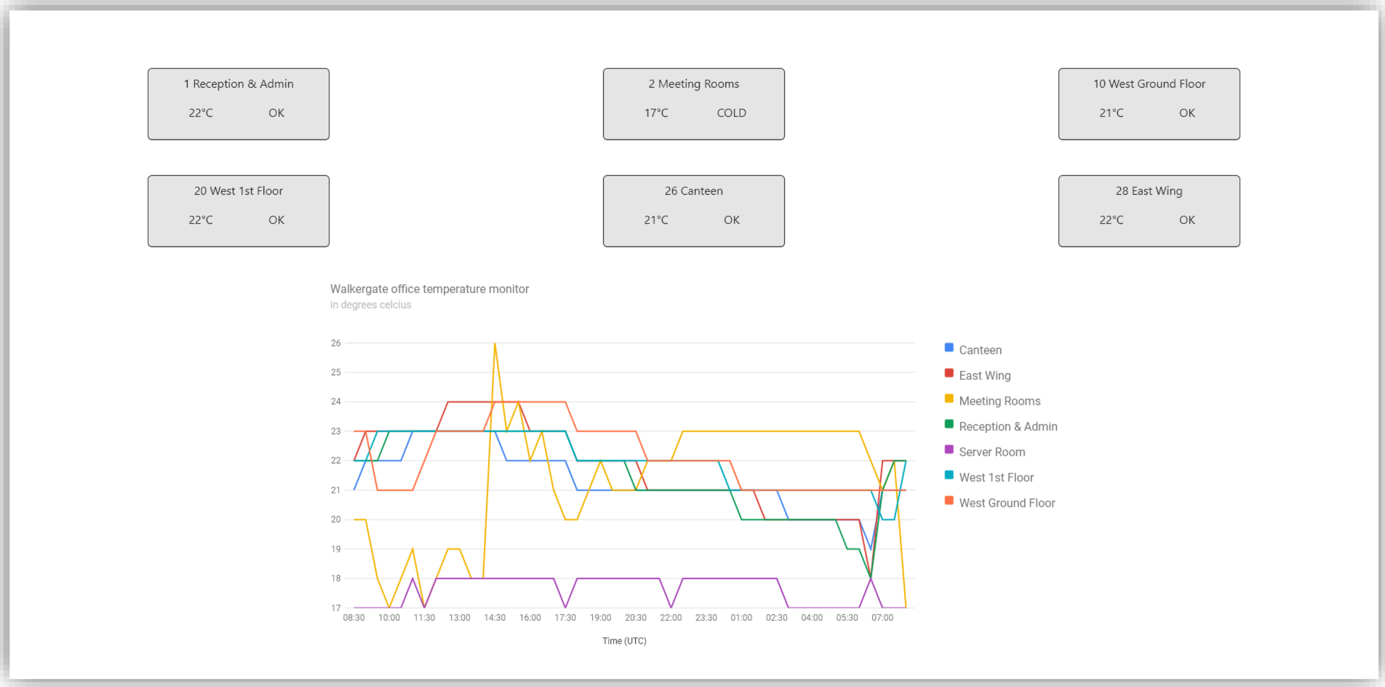
Once all of the bulk programming had been completed, it was time for me to deploy to production.
This step was done manually as there were no Unit Tests to require Drone CI to complete, however, I have plans on adding these in the future to enable development later in the project's life.

This project also did not have its production environment set up fully as I still had to install Python3.10, WFastCGI and configure IIS to run the Flask application.

There was one main issue after being deployed, this was that the background thread that was in charge of gathering data would not run when hosted on IIS, however, would work when debugging in PyCharm. I had done extensive research on why this was the case and found that the best solution, in this case, was to separate the data harvester thread from the Flask app and run this as a scheduled task.

After some more monitoring and testing, it is now working well and the building manager and I use this whenever we need to see the temperatures of the office.

The End Product



1 Reception & Admin
22°C OK

2 Meeting Rooms
17°C COLD

10 West Ground Floor
21°C OK

20 West 1st Floor
22°C OK

26 Canteen
21°C OK

28 East Wing
22°C OK

The End product links together the HTML and JavaScript programed by Jay and my backend database and web scraping work to provide this simple, easy to read informative page on the buildings temperature.

[B4] In the end, we provided an elegant solution to a time costly issue and even added the 24-hour graph feature as an added benefit.

[S2] The user interface is simple and does not require any navigation since it is a live one page view. This makes the dashboard as easy and simple to use as possible, but does not let down on the features displayed.

The building manager now uses this on a day-to-day basis to ensure we are keeping our staff as comfortable as we can.

“Thanks for putting this together.

It now shows the whole office that the system is working properly and is running to specifications.

It will stop all the individual requests to adjusting it for personal requirements.”

- Building Manager