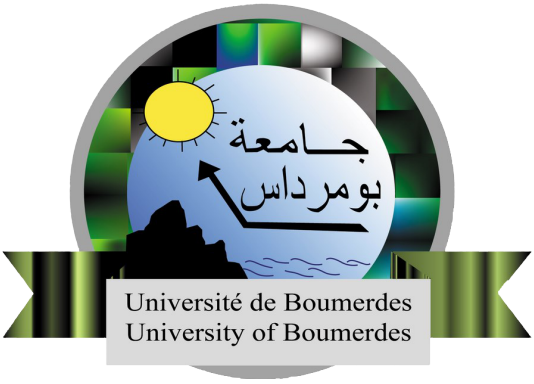


Institute of Electrical and Electronic Engineering

IGEE

Licence Project's Report

Implementation of a Smart University via Internet Of Things

Author:

- Sehili Chams Eddine

Supervisor: Dr. Tabet Youcef

Tuesday 4th June, 2024

Abstract

The integration of IoT (Internet Of Things) in a smart university promises revolutionary advancements in campus management and safety. Administratively, a user-friendly website centralizes session scheduling, with sessions activating automatically at scheduled times, aided by an ID card recognition system for real-time attendance tracking.

On the technical side, power supply management ensures operational continuity, while environmental sensors provide alerts in emergencies. The team also monitors water levels for proactive resource management. Facial recognition at entry points and a smart parking system enhance security and streamline campus management.

The significance of IoT and MQTT (Message Queuing Telemetry Transport) lies in their role as enablers of seamless communication and integration among various devices and systems, crucial for real-time monitoring and control in smart university applications.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Introduction to IoT

The Internet of Things (IoT) refers to the interconnected network of physical devices—such as appliances, vehicles, and other items—that are embedded with software, sensors, and connectivity. This connectivity allows these devices to collect and exchange data, enabling more efficient and automated systems. The IoT involves the networking of these physical objects to communicate and sense interactions either among themselves or with the external environment. In the coming years, IoT technology is expected to revolutionize daily life, with significant advancements in areas like medicine, energy, agriculture, smart cities, and smart homes[1].

IoT comprises a system of interrelated devices, from computing devices and mechanical machines to objects, animals, and people, all equipped with unique identifiers. This network facilitates the transfer of data without requiring human-to-human or human-to-computer interaction.

.1 History of IoT

The concept of IoT has evolved over decades, beginning with a simple internet-connected vending machine at Carnegie Mellon University in 1982, which reported its inventory and status. This early example demonstrated the potential for remote monitoring. In 1990, an internet-connected toaster marked another step toward the modern IoT, allowing users to control it remotely. The term "Internet of Things" was coined by Kevin Ashton in 1999, which formally described this network of interconnected devices.

Significant milestones in the development of IoT include the LG Smart Fridge in 2000, which allowed remote management of its contents, and the advent of smartwatches in 2004, which brought IoT to wearable technology. The iPhone in 2007, with its integration of IoT capabilities, transformed smartphones into hubs for various services and devices. Subsequent advancements included IoT in automotive diagnostics, smart TVs, image recognition with Google Lens, voice-activated assistants like Amazon's Echo, and semi-autonomous driving with Tesla's Autopilot.

.2 Key Components of IoT

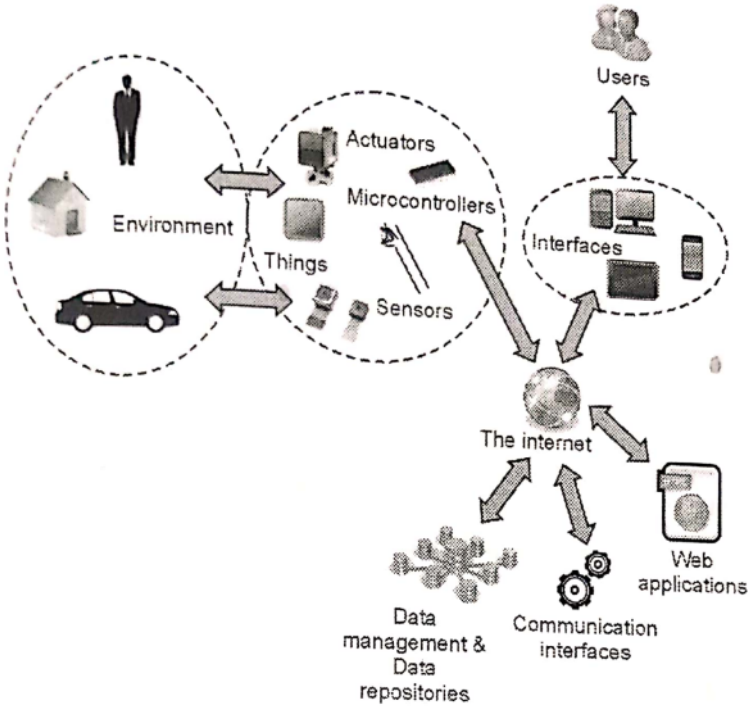
IoT systems consist of four main components: devices or sensors, connectivity, data processing, and user interfaces. Devices and sensors are crucial as they collect data from the environment. This data is then transmitted over a network, processed to extract meaningful information, and presented through user interfaces for decision-making.

Currently, over 9 billion devices are connected to the Internet, and this number is expected to reach 20 billion soon. IoT applications span various domains, requiring components such as low-power embedded systems, various sensors (temperature, image, gyro, obstacle, RF, IR, gas, LDR, and ultrasonic distance sensors), control units, cloud computing, big data, and networking connectivity.

.3 Building IoT

There are two primary approaches to building IoT networks: creating a separate network solely for physical objects or expanding the existing Internet to incorporate more devices, necessitating advanced technologies like robust cloud computing and rapid big data storage.

In the near future, IoT is anticipated to become more complex and widespread, offering "anytime, anyplace, anything" connectivity. Essential enablers of IoT include RFIDs, sensors, nanotechnology, and smart networks, facilitating efficient and scalable systems that adapt dynamically to changing environments and user needs.



.4 Characteristics and Challenges

IoT systems are characterized by their scalability, efficiency, and dynamic nature. They must handle an increasing number of connected devices, maintain secure and reliable data transfer, and ensure interoperability among diverse hardware and software configurations. Security and privacy concerns, potential for hacking, dependence on technology, and the need for standardization pose significant challenges. Despite these issues, IoT promises improved efficiency, convenience, and data-driven decision-making, transforming various sectors and everyday life.

.5 Application Domains and Advantages

IoT applications are prevalent in manufacturing, healthcare, security, and retail, among other fields. Modern applications include smart grids, smart cities, home automation, healthcare

monitoring, disaster detection, traffic monitoring, and wearable devices. The advantages of IoT include enhanced efficiency, better monitoring and control, data-driven insights, cost savings, and improved decision-making.

However, IoT also has its disadvantages, such as security risks, privacy concerns, dependence on technology, interoperability challenges, and high initial costs. Addressing these issues is crucial for the continued growth and success of IoT technologies.

.6 IoT in Smart Universities

In the context of a smart university, IoT technology significantly enhances campus management and safety protocols. The implementation involves several interconnected components that work together seamlessly. Central to this system is the use of an MQTT broker, which acts as a middle point for communication between various devices such as servers and Raspberry Pi units. The MQTT protocol allows devices to publish messages to specific topics and subscribe to topics of interest, ensuring efficient and reliable data transfer.

Academic Session Management

For example, in managing academic sessions, the server automatically checks the schedule stored in the database. When the time for a session arrives, the server publishes relevant information, such as student IDs, teacher ID, class name, and time, to a designated MQTT topic. The Raspberry Pi, which subscribes to this topic, receives the data and prepares the attendance list. This list is then published back to another topic, which the server subscribes to. The server captures this attendance data and updates the database, making it accessible to the administration through their dashboard.

Campus Facility Monitoring

The technical team also benefits from IoT integration, particularly in monitoring and controlling campus facilities. Sensors deployed across the campus, including in laboratories and water tanks, measure parameters like humidity, temperature, gas detection, fire detection, water levels, and power supply to different blocks. The Raspberry Pi collects this data periodically and publishes it to relevant topics. The server, subscribing to these topics, receives the data, stores it in the database, and displays real-time updates on the technical team's dashboard. This real-time monitoring allows the technical team to respond promptly to changes, ensuring operational continuity and safety.

Parameter Modification

Furthermore, the technical team can modify specific parameters within the university, such as turning on the water pump or supplying power to different blocks (teaching block, administration block, or outside environment). When the technical team makes these changes on their dashboard, the updates are stored in the database. The server detects these changes and publishes them to the appropriate MQTT topics. The Raspberry Pi, subscribed to these topics, receives the instructions and adjusts the university's parameters accordingly.

Campus Security and Access Control

Additionally, the smart recognition system enhances campus security and access control. When the system authorizes a person, it publishes a message to a related MQTT topic. The Raspberry Pi, subscribed to this topic, receives the message and triggers actions such as opening doors for the authorized individual. This seamless communication ensures that only authorized personnel can access restricted areas, thereby enhancing security.

Smart Parking System

The smart parking system also leverages IoT for efficient management of parking spaces. When a new car arrives, the system checks if the car is authorized by the database. If authorized, it finds an available spot and publishes this information to a related MQTT topic. The Raspberry Pi, receiving this information, displays the parking spot details on an LCD screen and opens the parking gate for the car to enter and park in the designated spot. This automated process not only improves parking efficiency but also enhances user convenience.

Impact of IoT in Smart Universities

The integration of IoT technology in the smart university project enhances administrative efficiency, improves attendance monitoring, strengthens campus safety measures, optimizes resource management, and streamlines parking management. Through collaborative efforts between the administration and the technical team, the smart university project establishes a dynamic and secure learning environment, ready to meet the evolving needs of students and staff.

MQTT Protocol

.1 What is MQTT?

MQTT is a messaging protocol, based on standards, designed for machine-to-machine communication. It is commonly used by smart sensors, wearables, and other Internet of Things (IoT) devices that need to exchange data over networks with limited bandwidth. MQTT is favored for its ease of implementation and efficient data transmission capabilities, facilitating communication between devices and the cloud, and vice versa [2].

.2 Why is the MQTT protocol important?

MQTT has become a key protocol for IoT data transmission due to the following benefits:

Lightweight and Efficient

MQTT requires minimal resources for implementation on IoT devices, making it suitable even for small microcontrollers. For instance, a minimal MQTT control message can be as small as two bytes. Additionally, MQTT message headers are designed to be small, optimizing network bandwidth usage.

Scalable

MQTT's implementation involves a minimal amount of code and consumes very little power. The protocol includes features that support communication with a large number of IoT devices, enabling connections with millions of devices.

Reliable

MQTT is designed to handle unreliable cellular networks with low bandwidth and high latency. It includes features to minimize reconnection time and offers three levels of quality of service (QoS) to ensure message delivery reliability: at most once (0), at least once (1), and exactly once (2).

Secure

MQTT simplifies the process of encrypting messages and authenticating devices and users, using modern authentication protocols such as OAuth, TLS1.3, and Customer Managed Certificates.

Well-Supported

MQTT is supported by various programming languages, including Python, which allows developers to implement the protocol quickly and with minimal coding in different types of applications.

.3 What is the principle behind MQTT?

MQTT operates on the publish/subscribe model. Unlike traditional client-server communication, where clients request data from servers, MQTT decouples the message sender (publisher) from the message receiver (subscriber) using a message broker. The broker filters incoming messages from publishers and distributes them to the appropriate subscribers, providing three types of decoupling:

Space Decoupling

Publishers and subscribers are unaware of each other's network locations and do not exchange information like IP addresses or port numbers.

Time Decoupling

Publishers and subscribers do not need to operate or have network connectivity simultaneously.

Synchronization Decoupling

Publishers and subscribers can send and receive messages independently, without waiting for each other.

.4 What are MQTT components?

MQTT implements the publish/subscribe model with clients and brokers:

MQTT Client

An MQTT client can be any device, from servers to microcontrollers, running an MQTT library. When sending messages, it acts as a publisher, and when receiving messages, it acts as a subscriber.

MQTT Broker

The MQTT broker coordinates message exchanges between clients. Its responsibilities include receiving and filtering messages, identifying subscribed clients, and delivering messages. It also handles tasks such as:

- Authorizing and authenticating MQTT clients
- Passing messages to other systems for further analysis
- Managing missed messages and client sessions

MQTT Connection

Communication between clients and brokers begins with an MQTT connection. Clients initiate this by sending a CONNECT message to the broker, which responds with a CONNACK message to confirm the connection. Both clients and brokers require a TCP/IP stack for communication, and clients only connect with the broker, not directly with each other.

.5 How does MQTT work?

The operation of MQTT involves the following steps:

1. An MQTT client establishes a connection with the MQTT broker.
2. Once connected, the client can publish messages, subscribe to specific messages, or both.
3. The MQTT broker forwards received messages to the interested subscribers.

Let's delve into more details:

.6 MQTT Topic

Topics are keywords that the MQTT broker uses to filter messages for clients. Topics are organized hierarchically, similar to a directory structure. For instance, in a smart home system with devices on different floors, topics might be organized as follows:

- ourhome/groundfloor/livingroom/light
- ourhome/firstfloor/kitchen/temperature

.7 MQTT Publish

MQTT clients publish messages containing a topic and data in byte format, which can be text, binary data, XML, or JSON. For example, a lamp in a smart home might publish a message to the topic livingroom/light.

.8 MQTT Subscribe

MQTT clients send a SUBSCRIBE message to the broker to receive messages on specific topics. This message includes a unique identifier and a list of subscriptions. For instance, a smart home app might subscribe to the topic light to track the status of lights in the house.

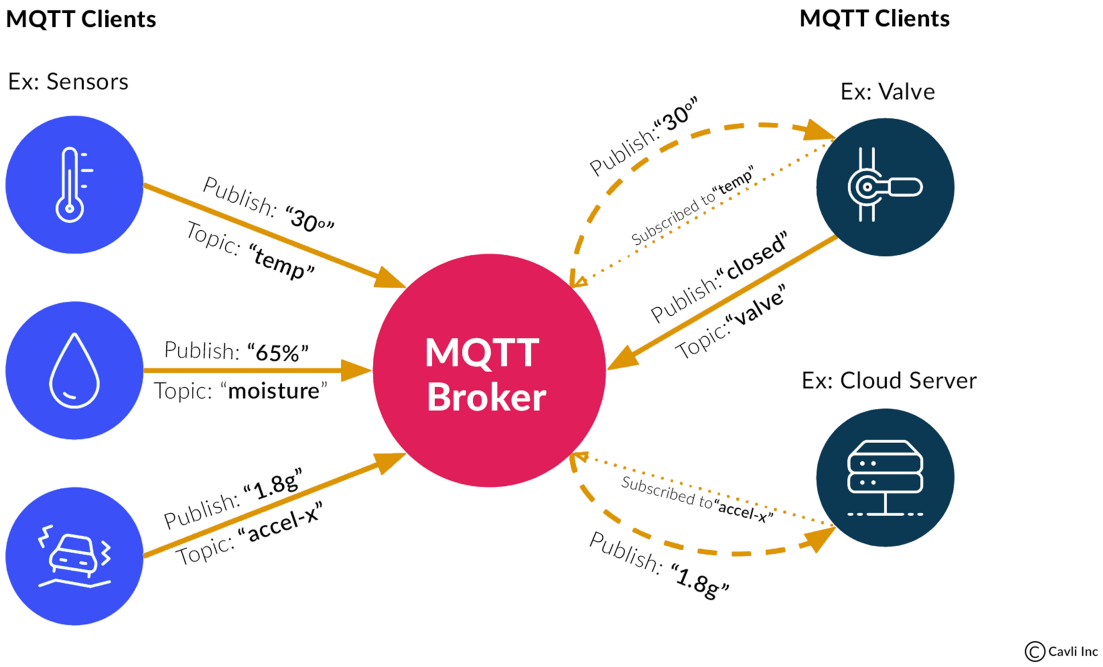


Figure 1.1: Representation of MQTT Broker Facilitating Communication Between Devices

Face Recognition System

.1 What is Face Recognition?

Face recognition is the process of taking a face in an image and actually identifying who the face belongs to. Face recognition is thus a form of person identification.

Early face recognition systems relied on an early version of facial landmarks extracted from images, such as the relative position and size of the eyes, nose, cheekbone, and jaw. However, these systems were often highly subjective and prone to error since these quantifications of the face were manually extracted by the computer scientists and administrators running the face recognition software.

As machine learning algorithms became more powerful and the computer vision field matured, face recognition systems started to utilize feature extraction and classification models to identify faces in images.

Not only are these systems non-subjective, but they are also automatic — no hand labeling of the face is required. We simply extract features from the faces, train our classifier, and then use it to identify subsequent faces.

Most recently, we've started to utilize deep learning algorithms for face recognition. State-of-the-art face recognition models such as FaceNet and OpenFace rely on a specialized deep neural network architecture called siamese networks.

These neural networks are capable of obtaining face recognition accuracy that was once thought impossible (and they can achieve this accuracy with surprisingly little data).

.2 Face Recognition Pipeline

A Face Recognition pipeline can be divided into three major stages:

Face Detection

Face detection is the initial step in the Face Recognition pipeline where the system identifies and localizes faces within an image or a video frame. This stage is crucial as it provides the regions of interest (ROIs) containing faces for further processing.

Methods and Models:

- **SSD (Single Shot MultiBox Detector)**: SSD is a popular deep learning-based object detection model. It efficiently detects objects, including faces, in images with high accuracy. SSD operates by predicting bounding boxes and class labels directly from feature maps at multiple scales. It's known for its speed and accuracy, making it suitable for real-time applications.
- **YOLO (You Only Look Once)**: YOLO is another state-of-the-art object detection model known for its speed and accuracy. YOLO divides the input image into a grid and predicts bounding boxes and class probabilities for each grid cell. It's favored for its real-time performance and has been widely used for face detection in various applications.

Note: Detailed explanations of YOLO will be provided in the "Face Recognition System" chapter.

Feature Extraction

After detecting faces, the next step is to extract discriminative features that represent each face uniquely. Feature extraction aims to transform face images into compact representations that capture essential facial characteristics.

Siamese Network: A Siamese Network is a neural network architecture designed for comparing two inputs and determining their similarity or dissimilarity. In the context of face recognition, Siamese Networks learn embeddings (feature representations) of faces by comparing pairs of images. The network consists of two identical sub-networks (twins) with shared weights. It learns to minimize the distance between embeddings of the same person's face while maximizing the distance between embeddings of different faces. This way, it creates a meaningful feature space where faces of the same person are clustered together, facilitating accurate recognition.

Feature Matching

Once feature representations are extracted, the final stage involves matching these features to a database of known faces to determine the identity of the input face.

Methods:

- **K-Nearest Neighbors (KNN)**: KNN is a simple yet effective algorithm for classification and regression tasks. In the context of face recognition, KNN matches the extracted features of an input face with the features of known faces in the database. It classifies the input face based on the majority

.3 Final Touch

After completing the face detection, feature extraction, and feature matching stages, the next step is to implement the face recognition model within the Smart University system. This involves several key tasks:

First, we need to create a comprehensive database of embeddings for all individuals within the university, including teachers, students, and employees. This database will store the unique facial embeddings generated for each person, which will serve as a reference for real-time identification.

Once the database is set up, we will develop software that can handle real-time streaming at the entry points of the university. This software will continuously extract faces from the stream feed, generate embeddings for these faces, and compare them with the stored embeddings in the database. This comparison will help determine whether an individual is authorized to enter the premises.

To integrate this face recognition system with the university's security infrastructure, we will use the MQTT protocol for IoT communication with Raspberry Pi device. When an authorized face is identified, the system will publish a message via MQTT to trigger the opening of the entry door, ensuring seamless and secure access control.

By implementing these steps, we will ensure that the face recognition system is fully operational, providing robust security and efficient access management for the Smart University.

Smart Parking System

In this chapter, we aim to develop a software system capable of recognizing the license plates of cars entering the parking lot, identifying them in a database, and if they are authorized,

sending a message to a Raspberry Pi to open the gate for the respective car. We will use a pre-trained YOLO model on a license plate database and Optical Character Recognition (OCR) [3] to extract the plate number.

Website Development

The website is divided into four different parts:

.1 Administration Part

In this part, the administration of the university is responsible for creating new student, teacher, and technical team profiles. Additionally, they have the capability to schedule class sessions by specifying the group, teacher, class number, and associated time. This class scheduling process is automated by the Raspberry Pi at the specified time, and the attendance records will be stored in the database. After that, they can see this attendance information on their dashboard.

.2 Teachers' Part

Teachers, upon logging into their profiles, can view their own absence information as well as the absences of the students they teach. This information is derived from the absence lists recorded in the database.

.3 Students' Part

Students, upon logging into their profiles, can view their own absence information. This information is derived from the absence lists recorded in the database.

.4 Technical Team Part

The technical team, upon accessing their profiles, can view all data records in real-time. These records include:

- Laboratory environment: humidity, temperature, gas detection, and fire detection status.
- Water tank level.
- Current status of the water pump and power supply to different parts of the university.

Chapter 2

Face Recognition System

As we mentioned in the face recognition section of the previous chapter, in order to implement a face recognition system, we need to follow a specific pipeline:

Face Detection:

This section is responsible for extracting the face from the input image to pass it to the next process (feature extraction). To achieve this, we need an efficient object detection model that can run at a high frame rate to improve system inference speed and accuracy.

In this process, we chose to work with the YOLO (You Only Look Once) model due to its ability to infer images at high frame rates and with high accuracy. Specifically, we selected YOLO version 8 (YOLOv8), as illustrated by Ultralytics, since it is the fastest model compared to other YOLO versions. Additionally, it offers simple training steps that can be accomplished with just a few lines of code and is user-friendly after training. It is well-supported by Ultralytics, unlike SSD, which is complex in both the training and inference processes.

> *Note:* The accuracy of SSD may be slightly better than YOLOv8, but in terms of frame rate inference, YOLOv8 significantly outperforms SSD.

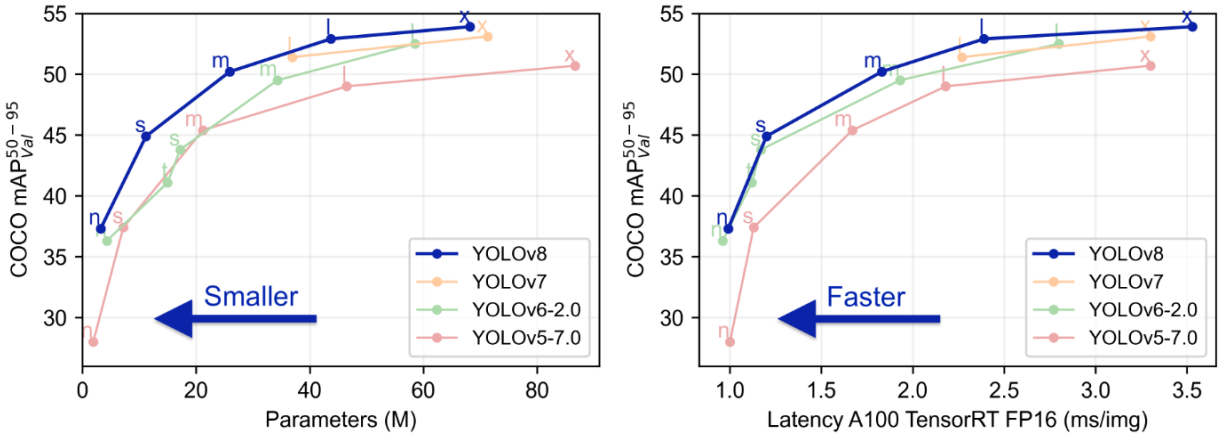


Figure 2.1: YOLOv8 versus previous versions of YOLO

Model	Frames Per Second (FPS)	Accuracy
YOLOv8	40-155	Sacrifices some accuracy
SSD	22-46	Good accuracy

Figure 2.2: SSD Versus YOLOv8 in terms of inference speed

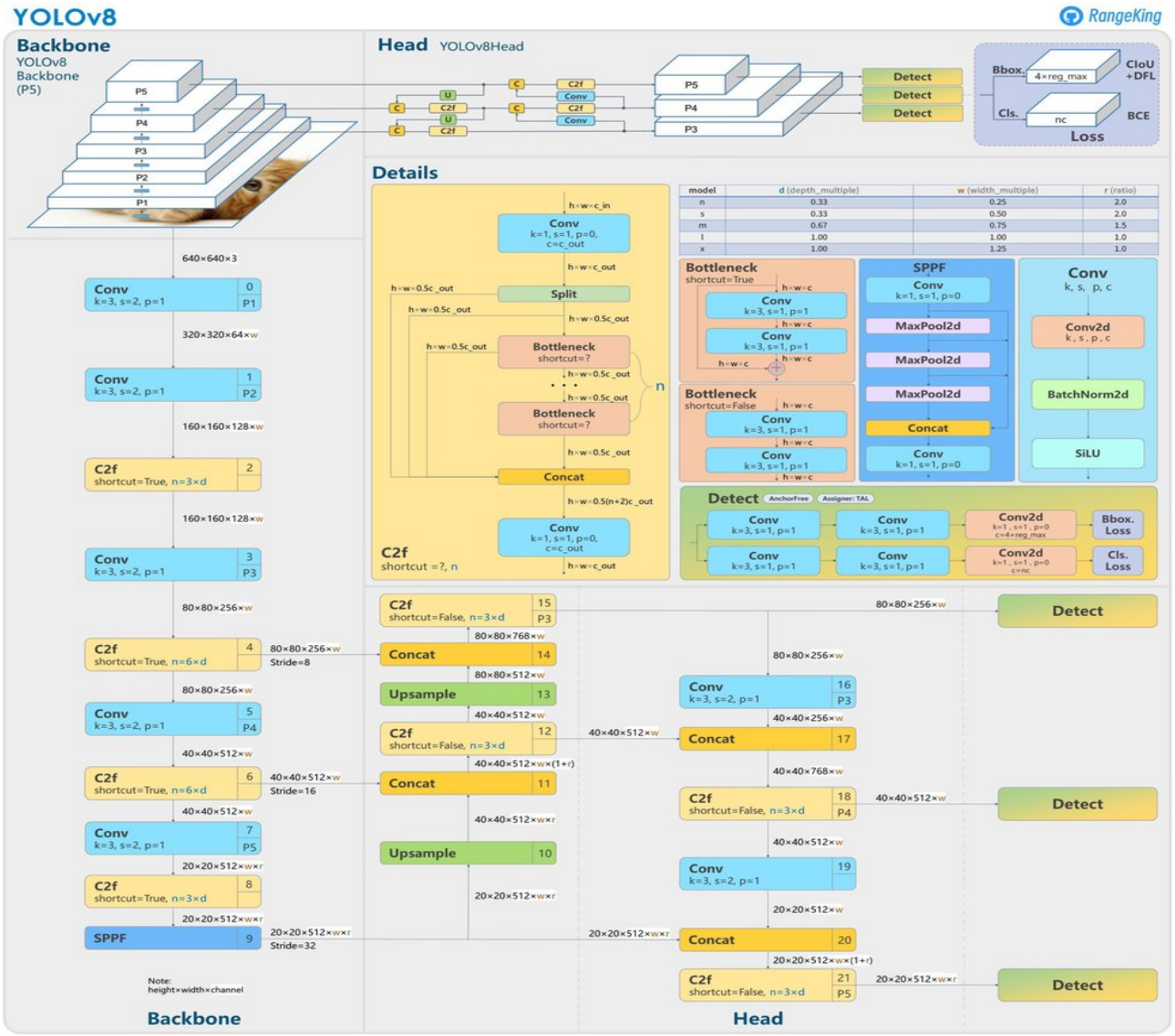
Let's go ahead and explain YOLOv8:

.1 YOLOv8 Model Architecture

The YOLOv8 architecture consists of the following components:

Input: It accepts a 640 x 640 x 3 image input.

Architecture: A combination of CNNs, upsampling layers, and shortcuts, as illustrated in the following image:



Output: It outputs detections at three scales:

1. **Large objects:** It outputs Tensor of size

$$20 \times 20 \times 512$$

   - Splits the image into

$$20 \times 20 \ \textit{grids}$$

2. **Medium objects:** It outputs Tensor of size

$$40 \times 40 \times 512$$

   - Splits the image into

$$40 \times 40 \ \textit{grids}$$

3. **Small objects:** It outputs Tensor of size

$$80 \times 80 \times 256$$

   - Splits the image into

$$80 \times 80 \ \textit{grids}$$

Regarding the depth of the tensors, it can be a bit confusing. From my knowledge and research experience with various YOLO model architectures (e.g., YOLOv2, YOLOv3, etc.), the depth of the output tensors is typically calculated in two ways depending on the number of detections per grid:

1. For single-class detection per grid cell:

$$\text{depth} = (B \times 4 + 1) + nc$$

   where:

   - B: Number of anchor boxes
   - 4: For the regression detections typically 4 (x, y bounding box coordinates in the image + height, width of the bounding box)
   - 1: For probability confidence or objectness score
   - nc: Number of classes in the dataset

2. For multi-class detection per grid cell:

$$\text{depth} = B \times (4 + 1 + nc)$$

Note: YOLO typically splits the image into grids as shown above, and in each grid, it produces a number of bounding boxes where each bounding box can be characterized by certain parameters depending on the type of classification in each grid cell.

If it is a single-class detection per grid cell, each bounding box will have only five parameters:

$$(x, y, h, w, P) : (x \text{ coordinate}, y \text{ coordinate}, \text{ height}, \text{ width}, \text{ probability confidence})$$

and all those bounding boxes in the grid will detect only one class:

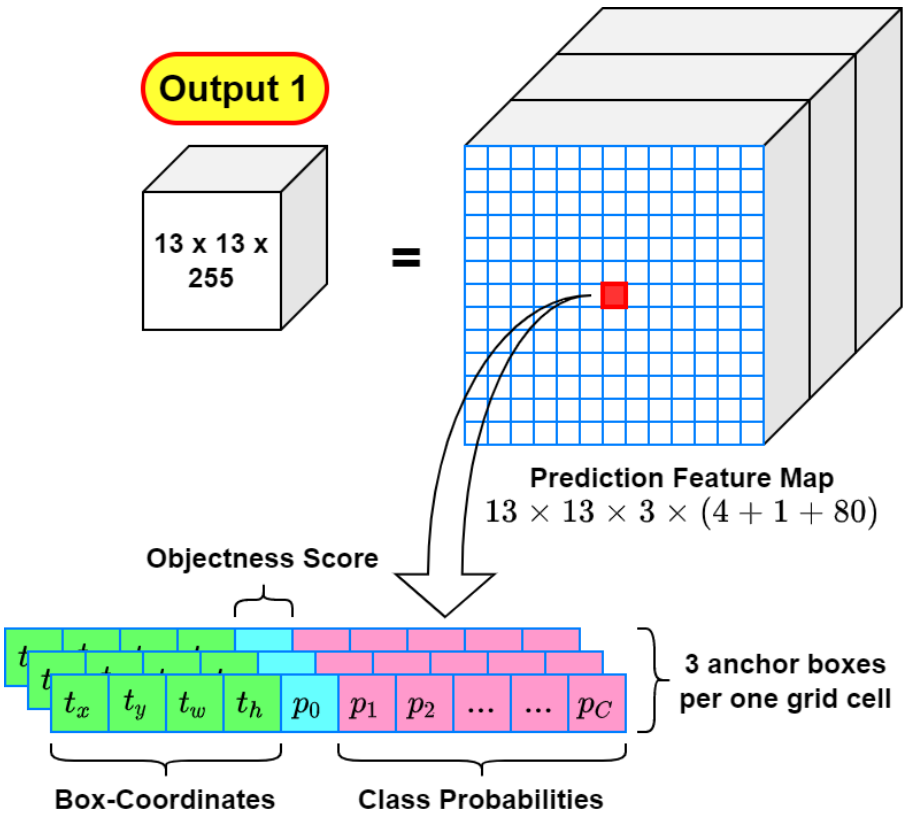
$$(c_1, c_2, c_3, \ldots, c_n)$$

This is like a probability distribution across different classes where the class with the highest probability will be chosen.

For multi-class detection per grid cell, in this case, each bounding box per grid cell will contain:

$$(5 + nc) \text{ parameters}$$

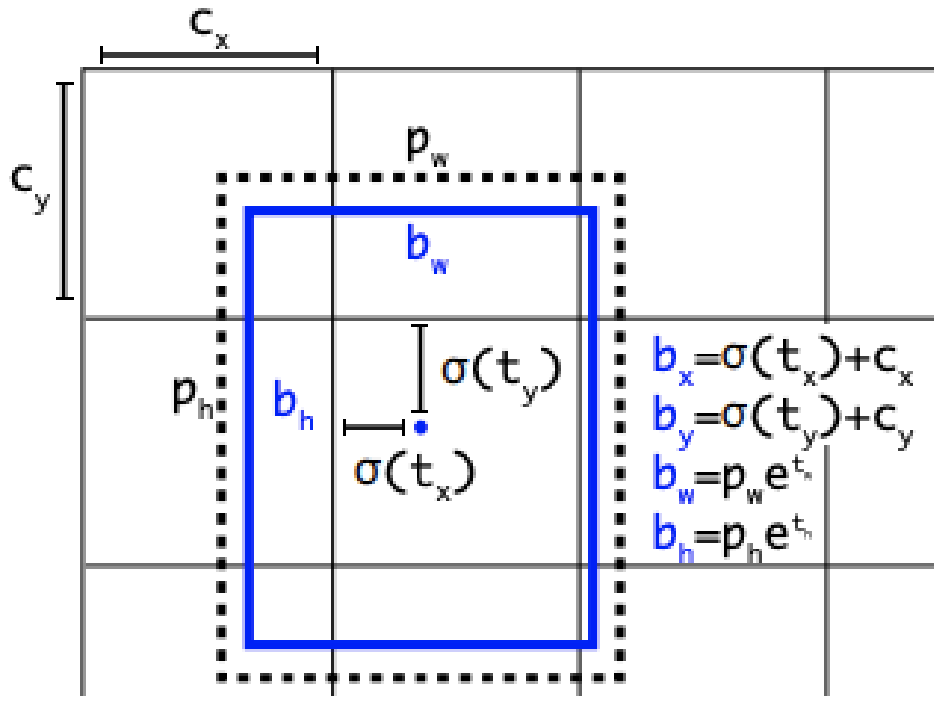
where 5 indicates (x, y, h, w, P) and nc indicates the number of classes. This means that each bounding box will indicate a specific class in the grid cell. We can represent the tensor as shown in the following image:



Particularly, YOLO outputs a feature map as indicated in the image. The values in the tensor result from a regression, so they are real numbers. To obtain:

$$(x, y, h, w)$$

we need to follow the method indicated in this image:



As shown, the (x, y, h, w) parameters correspond to (b_x, b_y, b_h, b_w), respectively. The x and y values are obtained by applying the sigmoid function to t_x and t_y, then adding the index of the grid. This mapping provides the location of the center of the box relative to the specific grid cell. Adding the grid index will make the center location relative to the image in term of grids. We then multiply these x and y values by the ratio of the image height or width to the number of grid cells per height or width to obtain their actual positions in the image.

Simultaneously, we apply the exponential function to t_h and t_w (to ensure positive values for hight and width) and then multiply these by p_h and p_w, respectively, to get the h and w parameters in terms of the grids. To convert these to the actual image size, we multiply them by the ratio of the image height or width to the number of grid cells per height or width.

The confidence score p is obtained by applying the sigmoid function to p_0, giving the probability of an object being present in the grid cell.

Finally, the class probabilities $c_1, c_2, c_3, \ldots, c_n$ are found by applying the softmax function to $p_1, p_2, p_3, \ldots, p_c$. This produces a probability distribution where the sum is 1, and the highest probability corresponds to the predicted class for the box within the grid cell.

Note: For more detailed and advanced information concerning YOLO, please check these references [4] ,[5],[6] and [7]

.2 YOLOv8 Model Training

Training a YOLO model is straightforward. Here's a step-by-step guide based on my experience:

First, we collect a dataset of images containing faces. Then, we label these images using appropriate software, such as ImageLabel. This process generates a folder containing label files, where each label file is a text file with bounding box information in the format (class, x, y, height, width). Here, 'class' represents the class label (e.g., 0, 1, 2, 3, etc.), and x, y, height, and width are the coordinates and dimensions from the previous section.

Next, we split the data into two folders: `Train` and `Validation`. Inside each of these folders, we create two subfolders: `images` and `labels`. The `images` folder contains the images, and the `labels` folder contains the label text files .

To get started with the training, we install the Ultralytics library using the following command:

```
pip install ultralytics
```

Then, we train our model using the following code:

```
from ultralytics import YOLO

# Load a pretrained model (recommended for training)
model = YOLO("yolov8n.pt")

# Train the model
results = model.train(data="custom_data.yaml", epochs=100, imgsz=640)
```

Listing 2.1: YOLOv8 Training

Let's explain what this code does:

- First, we import the `YOLO` class from Ultralytics.
- We define a model using pretrained weights, typically from the COCO dataset. This helps the model start with some knowledge about general objects.
- Then, we specify the training parameters:
  - `epochs`: This is the number of times the model iterates over the entire dataset. We set it to 100 epochs.
  - `imgsz`: This parameter sets the size of the input images. We use 640, which means each image is resized to 640x640 pixels.

- Additionally, we can specify:
  - `batch_size`: This determines how many images the model processes before updating its weights. Instead of updating after each epoch, it updates after processing a batch of images.

Once the training is complete, we find the best model weights saved in the directory **"runs/train/weights/best.pt"**. These weights represent the best performance during training in terms of mean Average Precision (mAP) accuracy.

To use the trained model for real-time inference, we write the following code:

```
from ultralytics import YOLO
import cv2

video = cv2.VideoCapture(0)
model = YOLO("best.pt")  # Load the best pretrained model

while True:
    ret, image = video.read()
    if ret:
        result = model(image)
        boxes = result[0].boxes
        for bbox in boxes:
            bbox_coords = [int(coord) for coord in bbox.xyxy[0]]
            print(bbox_coords)
```

Listing 2.2: YOLOv8 Real-Time Inference

In this script:

- We use OpenCV to capture video from the webcam.
- We load the best model weights using `YOLO("best.pt")`.
- In a continuous loop, we read frames from the webcam and pass them to the model for inference.
- For each detected object, we extract the bounding box coordinates and print them. These coordinates can be used to extract faces from the image and pass them to the next step in the pipeline, which is feature extraction.

Now, let's proceed to explain the feature extraction process.

Feature Extraction

Feature extraction plays a pivotal role in the face recognition system pipeline as it serves as the core component for identifying individuals. The accuracy of the system heavily relies on the precision of feature extraction. When done correctly, it enables the system to generate nearly identical feature vectors for the same person across different images, ensuring a high level of accuracy in recognition. Conversely, it also ensures that feature vectors for different individuals are distinctly different, creating a substantial separation between them.

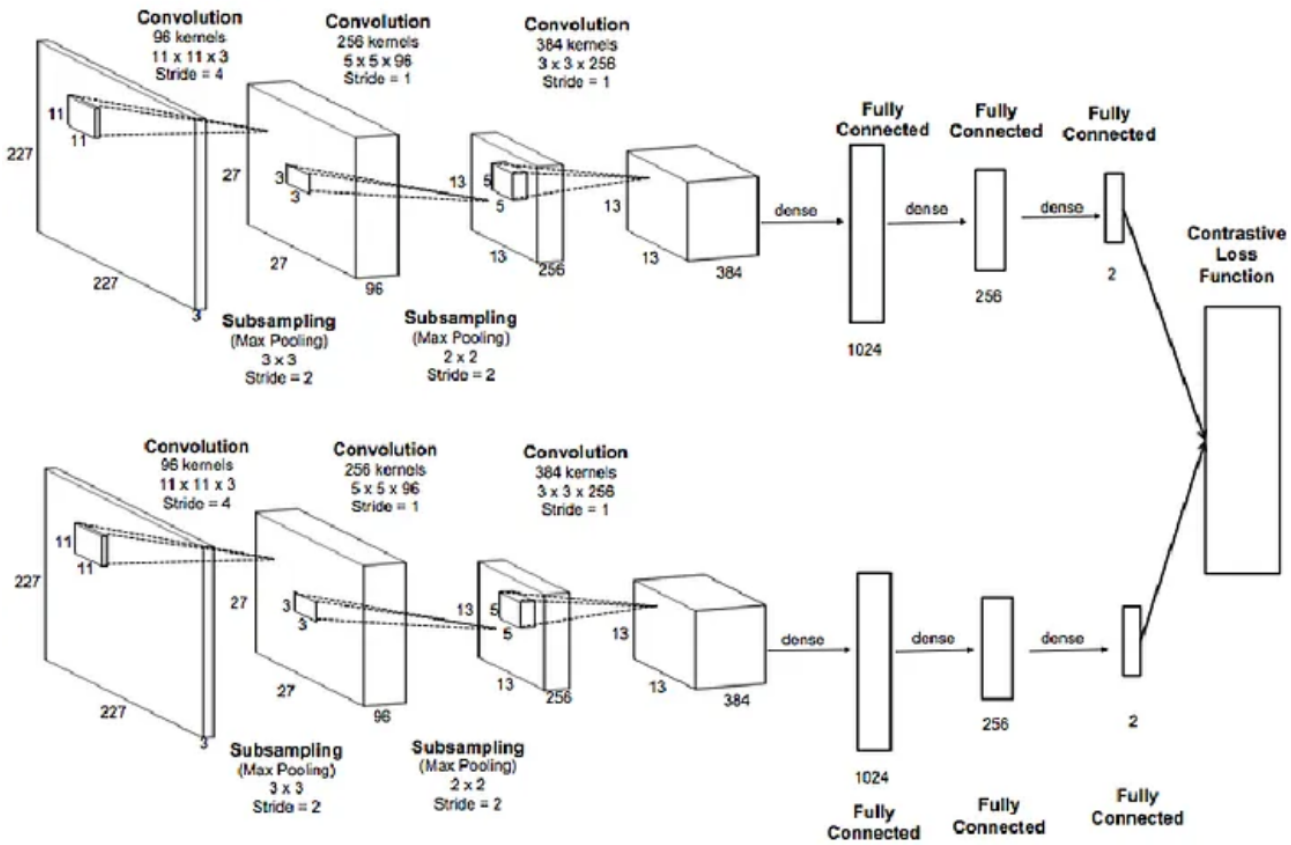
The primary objective is to develop a method that consistently embeds similar faces in a consistent manner, while ensuring that embeddings for different individuals are as dissimilar as possible. Essentially, our goal is to minimize the distances between embeddings of similar faces and maximize the distances between embeddings of different faces. This approach enhances the system's ability to accurately distinguish between individuals, leading to robust and reliable face recognition.

for this pupose A crucial and useful network model is the Siamese Neural Network architecture.

The Siamese Neural Network architecture consists of two identical sub-networks, often called "twins," that share the same weights and parameters. Each sub-network takes an input image and passes it through several convolutional layers, pooling layers, and fully connected layers. The output of each sub-network is a learned embedding, which represents the input image in a lower-dimensional space[8].

The embeddings from the two subnetworks are then compared using a distance metric, such as the L1 distance or Euclidean distance. The distance metric measures the similarity between the embeddings, where a smaller distance indicates a higher similarity.

The following image illustrates the architecture of a Siamese network:



.1 Training a Siamese Network:

A Siamese neural network for facial recognition is trained using the Anchor-Positive-Negative (APN) approach [9]. In this approach:

- **Anchor**: The face of the person.
- **Positive**: The same face of the person from another look.
- **Negative**: A different face.

The loss function used to train this Siamese network is called the triplet loss function. This loss function tries to maximize the distance between the anchor and the negative, and minimize the distance between the anchor and the positive [10] ,[11].

the formula for the tripled loss finction is given by :

$$\sum_{i}^{N} \left[\|f(x_i^a) - f(x_i^p)\|_2^2 - \|f(x_i^a) - f(x_i^n)\|_2^2 + \alpha \right]$$

After training the Siamese network with the triplet loss function, we will have a network capable of extracting features from face images. The next step is to perform feature matching. An easy way to implement this is using KNN (K-Nearest Neighbors).

We first collect a set of embedded faces using the Siamese network and store these embeddings in the database. Then, during real-time streaming, we start by extracting faces from the image, passing the extracted face through the Siamese network to obtain an embedding vector. We apply KNN on that embedding vector with the stored dataset, identifying the person by finding the closest embedded vector in the database to this embedding. The distance should be under a specific threshold to determine if the face belongs to the identified person or not.

Implementation

In this section we will develop software that can handle real-time streaming at the entry points of the university. This software will continuously extract faces from the stream feed, generate embeddings for these faces, and compare them with the stored embeddings in the database. This comparison will help determine whether an individual is authorized to enter the premises.

To integrate this face recognition system with the university's security infrastructure, we will use the MQTT protocol for IoT communication with Raspberry Pi device. When an authorized face is identified, the system will publish a message via MQTT to trigger the opening of the entry door, ensuring seamless and secure access control.

For the face extraction part, we will use the YOLOv8 pre-trained model on faces data [12]. For the subsequent procedures, we will use the `face-recognition` library [13], which contains both feature extraction and feature matching functionalities.

The software is implemented in the following Python code:

```
import face_recognition
import time
import threading
import numpy as np
import os
import cv2
import paho.mqtt.client as mqtt
from facenet_pytorch import InceptionResnetV1
from ultralytics import YOLO
names = np.array([], dtype=object)
```

```
faces=[]
client=mqtt.Client(client_id="587364535763")

for image in os.listdir('faces'):
      face=cv2.imread(f"faces/{image}")
      encodings=face_recognition.face_encodings(face)[0]

      faces.append(encodings)
      print(len(faces))
      names=np.append(names,image.split('.')[0])
model1=YOLO('yolov8n_100e.pt')

class Process:
        def __init__(self) :
            self.access=False
            self.frame=None
            self.rec=None
            self.name='checking'
            self.bbox=None
            self.prev_name=None
            self.name_check=None
            self.image=None
            self.finish=True
            t1=threading.Thread(target=self.process_all_time)
            t2=threading.Thread(target=self.process_frames)
            t1.start()
            t2.start()

        def process_frames(self):
            while self.finish :
               if self.rec  :
                        time.sleep(3)
                        print('helo')
                        self.frame=cv2.cvtColor(self.image,cv2.COLOR_BGR2RGB
    )

                        locations=face_recognition.face_locations(self.frame
    )


                        encoding=face_recognition.face_encodings(self.frame,
    locations)
                        if len(encoding)>0 and len(encoding) < 2 :

                            comparition=face_recognition.compare_faces(faces
    ,encoding[0])
                            self.name_check=names[comparition]


                            if len(self.name_check)==0:
                                self.name='cheking'
                                self.prev_name=self.name


                            elif len(self.name_check)==1:

                                 if self.prev_name != self.name_check[0]:
                                       self.prev_name=self.name_check[0]
                                       self.publish()
```

```
                                self.name= self.name_check[0]

        def publish(self):
                        client.connect('127.0.0.1', 1883)
                        client.publish("artifitial","access")
                        time.sleep(0.5)
                        client.disconnect()
        def process_all_time(self):
            video=cv2.VideoCapture(0)
            while True :
                self.rec,self.image=video.read()
                if self.rec :
                    result=model1(self.image)  # use self.image instead of
    self.frame
                    r=result[0]
                    boxes=r.boxes
                    if len(boxes)==1:
                        self.bbox=[int(i) for i in (boxes[0]).xyxy[0]]

                        cv2.putText(self.image, self.name, (self.bbox[0],
    self.bbox[1]), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 0), 2)
                        top_left = (self.bbox[0], self.bbox[1])
                        top_right = (self.bbox[2], self.bbox[1])
                        bottom_left = (self.bbox[0], self.bbox[3])
                        bottom_right = (self.bbox[2], self.bbox[3])

                        # Define the color and thickness of the lines
                        color = (255, 255, 0)  # yellow in BGR color space
                        thickness = 2
                        var=15
                        # Draw the four lines (edges of the rectangle)
                        cv2.line(self.image, top_left, (self.bbox[0], self.
    bbox[1]+var), color, thickness)
                        cv2.line(self.image, top_left, (self.bbox[0]+var,
    self.bbox[1]), color, thickness)
                        cv2.line(self.image,(self.bbox[2]-var, self.bbox[3])
     , bottom_right, color, thickness)
                        cv2.line(self.image,(self.bbox[2], self.bbox[3]-var)
     , bottom_right , color, thickness)
                        cv2.line(self.image,(self.bbox[0], self.bbox[3]-var)
     ,bottom_left , color, thickness)
                        cv2.line(self.image,bottom_left ,(self.bbox[0]+var,
    self.bbox[3]),  color, thickness)
                        cv2.line(self.image,top_right ,(self.bbox[2], self.
    bbox[1]+var) , color, thickness)
                        cv2.line(self.image,(self.bbox[2]-var, self.bbox[1])
     , top_right, color, thickness)
                    cv2.imshow('image', self.image)

                    if cv2.waitKey(1)==ord("q"):
                        break
            video.release()
            cv2.destroyAllWindows()
            self.finish=False
if __name__ =="__main__":
      target_start=Process()
```

Listing 2.3: Python code for real-time face recognition

Imports and Initialization

First, we start by importing the necessary libraries:

```
import face_recognition
import time
import threading
import numpy as np
import os
import cv2
import paho.mqtt.client as mqtt
from facenet_pytorch import InceptionResnetV1
from ultralytics import YOLO
```

Listing 2.4: Importing Libraries

- `face_recognition`: For facial recognition tasks.
- `time`: To add delays.
- `threading`: For running multiple processes simultaneously.
- `numpy (np)`: For numerical operations, particularly with arrays.
- `os`: For interacting with the operating system.
- `cv2`: OpenCV for image processing and computer vision tasks.
- `paho.mqtt.client as mqtt`: For MQTT protocol to publish/subscribe messages.
- `InceptionResnetV1`: From `facenet_pytorch`, not used in the provided code but generally for face embeddings.
- `YOLO`: For object detection using the YOLO (You Only Look Once) model.

Next, we initialize a NumPy array [14] to contain the names of the persons and an empty list to store the embeddings that correspond to each person. We also initialize an MQTT client.

```
names = np.array([], dtype=object)
faces=[]
client = mqtt.Client(client_id="587364535763")
```

Listing 2.5: Initializing Variables

Load and Encode Faces

We then fill these two by looping through all the images in the folder `faces` at the current directory. We extract the title of the image, which is the name of the person, and insert it in the `names` NumPy array. We embed the image and store the embedded face in the `faces` list.

```
for image in os.listdir('faces'):
    face = cv2.imread(f"faces/{image}")
    encodings = face_recognition.face_encodings(face)[0]
    faces.append(encodings)
    print(len(faces))
    names = np.append(names, image.split('.')[0])
```

Listing 2.6: Loading Faces and Creating Encodings

YOLO Model Initialization

We load the YOLO model for object detection using a pretrained weights file `yolov8n_100e.pt`.

```
model1 = YOLO('yolov8n_100e.pt')
```

Listing 2.7: Loading YOLO Model

Class Definition

Then, we create a class that has three functions:

- `process_all_time`: Processes the real-time stream at high FPS to not make the user interface look bad.
- `process_frames`: Responsible for checking the frame every 3 seconds, performing feature embedding, and then matching the embedding with the existing face embeddings. The output is filtered and passed to the NumPy array `names` to get the name of the person's face.
- `publish`: Publishes a message with MQTT.

```
class Process:
    def __init__(self):
        self.access = False
        self.frame = None
        self.rec = None
        self.name = 'checking'
        self.bbox = None
        self.prev_name = None
        self.name_check = None
        self.image = None
        self.finish = True
        t1 = threading.Thread(target=self.process_all_time)
        t2 = threading.Thread(target=self.process_frames)
        t1.start()
        t2.start()
```

Listing 2.8: Class Definition

Process Frames Method

```
    def process_frames(self):
        while self.finish:
            if self.rec:
                time.sleep(3)
                self.frame = cv2.cvtColor(self.image, cv2.COLOR_BGR2RGB)
                locations = face_recognition.face_locations(self.frame)
                encoding = face_recognition.face_encodings(self.frame,
    locations)
                if len(encoding) > 0 and len(encoding) < 2:
                    comparison = face_recognition.compare_faces(faces,
    encoding[0])
                    self.name_check = names[comparison]
                    if len(self.name_check) == 0:
                        self.name = 'checking'
                        self.prev_name = self.name
                    elif len(self.name_check) == 1:
                        if self.prev_name != self.name_check[0]:
                            self.prev_name = self.name_check[0]
                            self.publish()
                        self.name = self.name_check[0]
```

Listing 2.9: Process Frames Method

This method checks frames every 3 seconds, detects faces, compares with known faces, and updates the name based on the comparison results. It also publishes a message if a new face is detected [”if the name changes”].

Publish Method

```
    def publish(self):
        client.connect('127.0.0.1', 1883)
        client.publish("artificial", "access")
        time.sleep(0.5)
        client.disconnect()
```

Listing 2.10: Publish Method

This method connects to the MQTT broker, publishes a message, and then disconnects. It publishes a message to the topic "artificial", which will be received by the Raspberry Pi and trigger the door to open.

Process All Time Method

```
    def process_all_time(self):
        video = cv2.VideoCapture(0)
        while True:
            self.rec, self.image = video.read()
            if self.rec:
                result = model1(self.image)
                r = result[0]
                boxes = r.boxes
                if len(boxes) == 1:
                    self.bbox = [int(i) for i in (boxes[0]).xyxy[0]]
                    cv2.putText(self.image, self.name, (self.bbox[0], self.
    bbox[1]), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 0), 2)
                    top_left = (self.bbox[0], self.bbox[1])
                    top_right = (self.bbox[2], self.bbox[1])
                    bottom_left = (self.bbox[0], self.bbox[3])
                    bottom_right = (self.bbox[2], self.bbox[3])
                    color = (255, 255, 0)
                    thickness = 2
                    var = 15
                    cv2.line(self.image, top_left, (self.bbox[0], self.bbox
    [1] + var), color, thickness)
                    cv2.line(self.image, top_left, (self.bbox[0] + var, self
    .bbox[1]), color, thickness)
                    cv2.line(self.image, (self.bbox[2] - var, self.bbox[3]),
     bottom_right, color, thickness)
                    cv2.line(self.image, (self.bbox[2], self.bbox[3] - var),
     bottom_right, color, thickness)
                    cv2.line(self.image, (self.bbox[0], self.bbox[3] - var),
     bottom_left, color, thickness)
                    cv2.line(self.image, bottom_left, (self.bbox[0] + var,
    self.bbox[3]), color, thickness)
                    cv2.line(self.image, top_right, (self.bbox[2], self.bbox
    [1] + var), color, thickness)
                    cv2.line(self.image, (self.bbox[2] - var, self.bbox[1]),
     top_right, color, thickness)
                cv2.imshow('image', self.image)
                if cv2.waitKey(1) == ord("q"):
                    break
        video.release()
        cv2.destroyAllWindows()
        self.finish = False
```

Listing 2.11: Process All Time Method

This method captures video frames continuously, uses YOLO model to detect objects (faces) in the frame, draws bounding boxes and labels, and displays the frame. It stops if the 'q' key is pressed.

Main Execution

Finally, we start the process by creating an instance of the `Process` class, starting the video processing and frame processing in separate threads.

```
if __name__ == "__main__":
    target_start = Process()
```

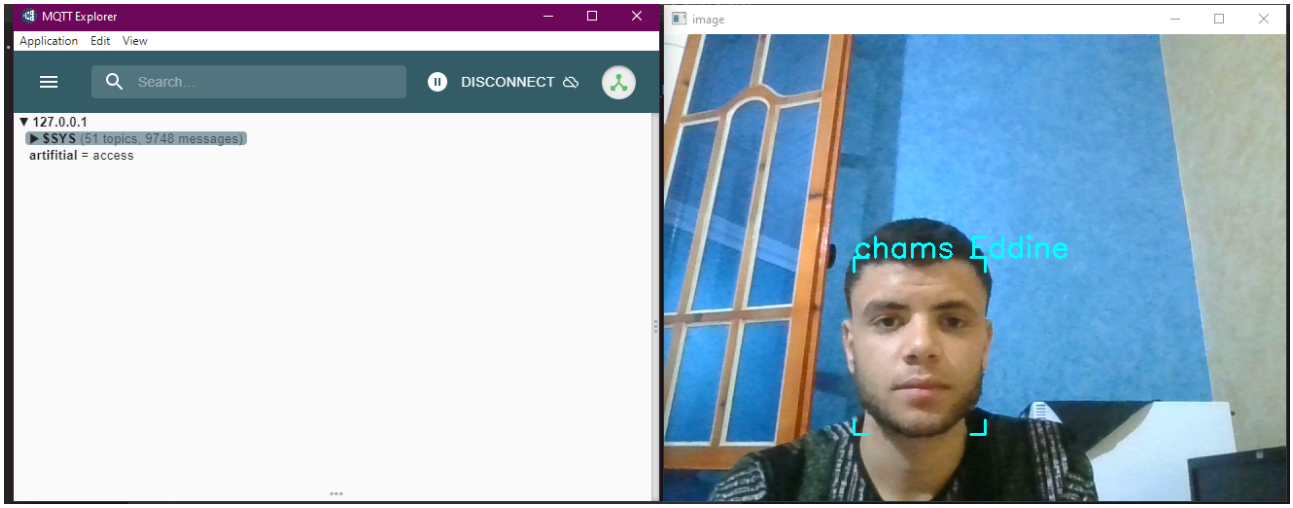
Listing 2.12: Main Execution

Testing:

We test our model by running:

`python face_rec.py`

where `face_rec.py` is the name of the file. The following image shows how the model recognizes faces:



As shown in the image, the face recognition system successfully recognizes faces and publishes messages, as indicated by the "artificial=access" message in the MQTT Explorer on the left.

Chapter 3

Smart Parking System

License Plate Recognition and parking management

In this section, we aim to develop a software system capable of recognizing the license plates of cars entering the parking lot, identifying them in a database, and if they are authorized, sending a message to a Raspberry Pi to open the gate for the respective car. We will use a pre-trained YOLO model on a license plate database and Optical Character Recognition (OCR) [3]to extract the plate number.

The code for the software is as follows:

```
from ultralytics import YOLO
import cv2
import paho.mqtt.client as mqtt
import time
import easyocr
import json

# Initialize MQTT client
client = mqtt.Client(client_id="587364535763")

# List of authorized license plate numbers
authorized_plates = ['1773711116', '0365510842', '00866112116']

# Parking location status
parking_location = {
    1: '',
    2: '',
    3: '',
   4:'',
    5:'',
    6:''}
# Load models
model = YOLO('Yolobest.pt')
reader = easyocr.Reader(['en'], gpu=True)

# Load video
cap = cv2.VideoCapture(0)


def process_frame(frame):
    """
    Process a single video frame to detect and read license plates.
    """
    results = model(frame)
    detections = results[0].boxes
    for box in detections:
        x1, y1, x2, y2 = [int(i) for i in box.xyxy[0]]
        license_plate_crop = frame[y1:y2, x1:x2, :]
        license_plate_gray = cv2.cvtColor(license_plate_crop, cv2.
   COLOR_BGR2GRAY)
```

```
40
41          result = reader.readtext(license_plate_gray)
42          if result:
43              text = result[0][-2].strip().replace(' ', '')
44              if text in authorized_plates:
45                  message = assign_parking_location(text)
46                  if message:
47                      send_mqtt_message("artificial", message)
48
49              # Annotate the frame with the detected license plate number
50              cv2.putText(frame, text, (x1, y1 - 5), cv2.FONT_HERSHEY_SIMPLEX,
    0.5, (255, 0, 255), 2)
51
52          # Draw the bounding box on the frame
53          cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 255), 2)
54
55      return frame
56
57
58  def send_mqtt_message(topic, message):
59      """
60      Send a message via MQTT.
61      """
62      try:
63          client.connect('127.0.0.1', 1883)
64          client.publish(topic, json.dumps(message))
65          time.sleep(2)
66          client.disconnect()
67      except Exception as e:
68          print(f"Error sending MQTT message: {e}")
69
70
71  def assign_parking_location(license_plate):
72      """
73      Assign a detected license plate to an available parking location.
74      If the license plate is already assigned, remove it (car leaving).
75      """
76      for spot, plate in parking_location.items():
77          if plate == license_plate:
78              parking_location[spot] = ''
79              print(f"Car with license plate {license_plate} left parking spot
    {spot}")
80              return {"car_out": True}
81
82      for spot, plate in parking_location.items():
83          if plate == '':
84              parking_location[spot] = license_plate
85              print(f"Assigned {license_plate} to parking spot {spot}")
86              return {"car_on": True, "Park_number": spot}
87
88      print("No available parking spots")
89      return {"car_on":False ,"Park_number": "not available"}
90
91
92  def main():
93      while True:
94          ret, frame = cap.read()
95          if not ret:
96              break
```

```
97
98          processed_frame = process_frame(frame)
99          cv2.imshow('Processed Video', processed_frame)
100
101         if cv2.waitKey(1) == ord('q'):
102             break
103
104     cap.release()
105     cv2.destroyAllWindows()
106
107
108 if __name__ == "__main__":
109     main()
```

Listing 3.1: Smart Parking software

.1 Explanation of the Smart Parking Software

The `process_frame` function starts by extracting the plate number from the car and then checks if it is authorized or not. When the car is authorized, it calls the `assign_parking_location` function, which assigns the right spot the car needs to park in or returns a message indicating no available spot if the parking is full. Additionally, the same function will empty the car spot if the car is leaving and return a message indicating the car's departure.

The reason for performing both entering and leaving management with the same function is that, in a real application, this software will be linked with two cameras: one at the entering point of the parking and the other at the leaving point. The two frame captures from the two cameras will be combined into one frame and then passed to the `process_frame` function. This way, the logic of the software will be correct: when it detects the plate for the first time, it means the car is entering the parking, and when it detects the plate the second time, it means the car is leaving the parking.

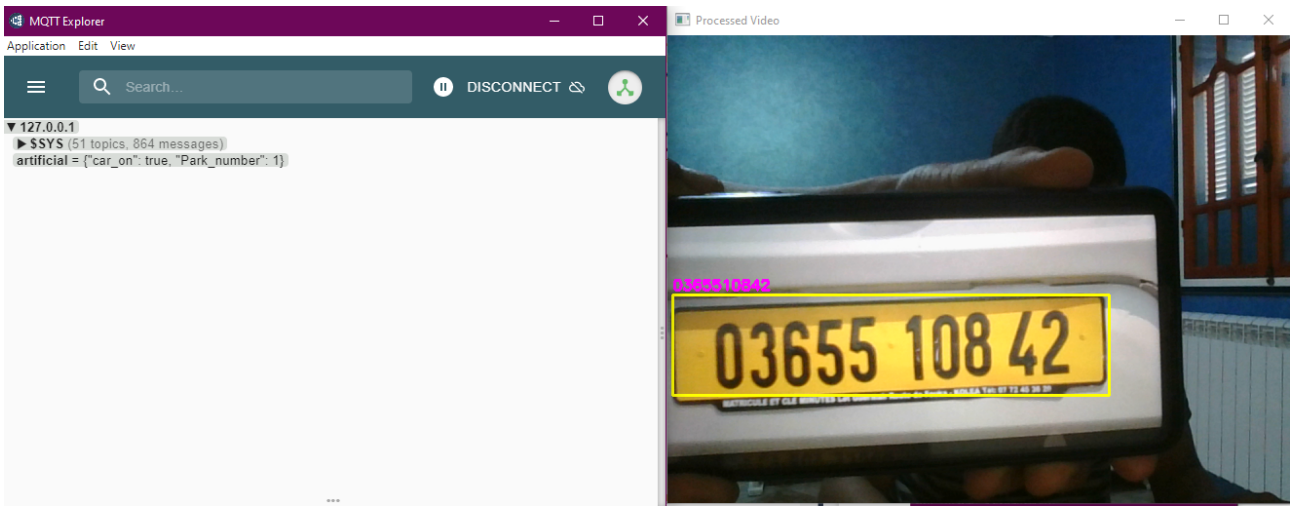
After the parking management is done with `assign_parking_location`, different messages are returned depending on the situation. Then, the `send_mqtt_message` function is used to send those messages to the Raspberry Pi to perform the required actions.

Finally, the `process_frame` function returns a frame with a bounding box around the plate and its number as a label. This frame is used by the `main` function to display it in the user interface of this software.

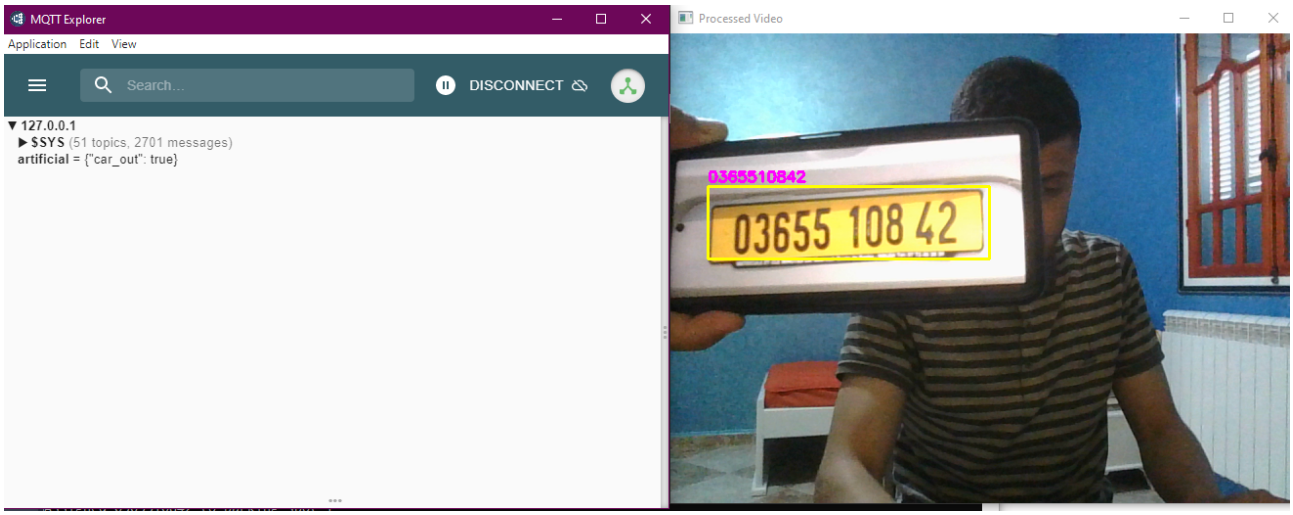
.2 Testing:

The software effectively identifies the license plate numbers, verifies the authority of cars, manages parking spot assignments, and publishes messages to open the parking gate. The code accurately handles both the entry and exit of vehicles, ensuring smooth operation of the smart parking system:

Car entering handling:



Car leaving handling:



Chapter 4

Website Development

Website Architecture

The website is divided into four different parts:

.1 Administration Part:

In this part, the administration of the university is responsible for creating new students, teachers, and technical team profiles. Additionally, they have the capability to schedule class sessions by specifying the group, teacher, class number, and associated time. This class scheduling process is automated by the Raspberry Pi at the specified time, and the attendance records will be stored in the database. After that, they can see this attendance information on their dashboard.

.2 Teachers' Part:

Teachers, upon logging into their profiles, can view their own absence information as well as the absences of the students they teach. This information is derived from the absence lists recorded in the database.

.3 Students' Part:

Students, upon logging into their profiles, can view their own absence information. This information is derived from the absence lists recorded in the database.

.4 Technical Team Part:

The technical team, upon accessing their profiles, can view all data records in real-time. These records include:

- Laboratory environment: humidity, temperature, gas detection, and fire detection status.
- Water tank level.
- Current status of the water pump and power supply to different parts of the university.

Additionally, they have the ability to control the water pump status, and power supply parameters in real-time by sending the changes they want to implement. Further details about the technical team's functionalities will be explained in their specific section.

In the website, the server we have chosen to work with is Django [15] due to several advantages:

- Django provides a high level of security, ensuring the safety of data.
- It has a robust ORM (Object-Relational Mapping) system, making database interactions seamless.
- Django's admin interface is highly customizable and efficient for administrative tasks.
- It supports rapid development and scalability.

Additionally, since Django is written in Python, it offers more flexibility in integrating IoT components. The MQTT protocol, which is well supported by Python, makes its integration easier compared to other servers using different languages.

Project Preparation

Getting started with Django is straightforward.We follow these simple steps to set up our project:

1. **Install Django**: Django can be easily installed using the Python package manager, pip. We open our command line interface and run the following command:

   ```
   pip install django
   ```

   This command will download and install Django and all its dependencies.

2. **Create a New Project**: Once Django is installed, we can create a new Django project using the following command:

   ```
   django-admin startproject smart_university
   ```

   This command creates a new directory named "smart_university" containing the necessary files and folder structure for our Django project. Inside the "smart_university" directory, we'll find:

   - **manage.py**: This is a command-line utility that lets us interact with our Django project. We can use it to perform various tasks such as running the development server, creating database migrations, and managing our project's applications.
   - **smart_university**: This folder serves as the Django main application folder. Inside, we'll find:
     - **settings.py**: This file contains all the settings for our Django project, including database configuration, static files, middleware, and more. It's crucial for configuring our Django server.
     - **asgi.py** and **wsgi.py**: These files are entry points for ASGI (Asynchronous Server Gateway Interface) and WSGI (Web Server Gateway Interface), respectively. They are used to deploy our Django project on various web servers.
     - **urls.py**: This file specifies the URL routing for our Django project. It defines the mapping between URLs and views, allowing Django to route incoming requests to the appropriate view functions. By default, it includes the route '/admin' which is handled by the function `admin.site.urls`, a pre-defined URL pattern for the Django admin site.
   - **Applications**: Django allows us to scale our project by creating multiple applications, each handling a specific set of functionalities. In our case, we'll add the following apps:
     - **api**: This app is responsible for handling API requests to the server.
     - **user_interface**: This app handles the user interface part of the website and its different requests.
     - **data_collect**: This app is responsible for storing data collected from Raspberry Pi device related to the technical team in the database.
     - **users**: This app is responsible for storing users in the database.

Initially, the structure of each app folder will include various Python files, each serving a specific role:

- **models.py**: This file defines the database models for the app, including database tables and their relationships.
- **views.py**: This file contains the view functions that handle incoming HTTP requests and return HTTP responses.
- **urls.py**: Similar to the project-level urls.py, this file specifies URL routing for the app, mapping URLs to views.
- **forms.py** (if applicable): This file defines forms used in the app for user input validation and processing.
- **admin.py** (if applicable): This file allows us to register app models with the Django admin site for easy management.
- **tests.py** (if applicable): This file contains unit tests for testing app functionality.
- **migrations/**: This directory contains database migration files generated by Django's migration system.

Creating Apps: We can create new Django apps using the following command:

```
python manage.py startapp app_name
```

We replace "app_name" with the desired name of our app. This command will generate the necessary files and folder structure for our new app within the project directory.

Configuring the settings.py file

Configuring the settings.py file is a crucial step in setting up our Django project. This file contains various settings that determine the behavior and functionality of our project. Let's walk through some important configurations we'll typically find in settings.py:

.1 Application Configuration

The `INSTALLED_APPS` setting specifies the list of applications installed in our Django project. Each application provides a set of functionalities and can be reused across multiple projects. Here's an example configuration:

```
# Application definition
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'users.apps.UsersConfig',
    'user_interface.apps.UserInterfaceConfig',
    'data_collect.apps.DataCollectConfig',
]
```

Listing 4.1: Application Configuration in `settings.py`

- **In this configuration**:

- `rest_framework`: This is the Django REST Framework, a powerful toolkit for building Web APIs. It provides a set of reusable components for handling requests, serializing data, authentication, and more, making it easier to develop RESTful web services in Django.
- `users.apps.UsersConfig`:Provides an Entry point to the custom `users` application.
- `user_interface.apps.UserInterfaceConfig`:Provides an Entry point to the custom `user_interface` application.
- `data_collect.apps.DataCollectConfig`:Provides an Entry point to the custom `data_collect` application.
- `api.apps.ApiConfig`:Provides an Entry point to the custom `api` applicationr.

.2 Database Configuration

The `DATABASES` setting specifies the configuration for our database. Django supports various database engines such as PostgreSQL, MySQL, SQLite, and Oracle. this configuration is for PostgreSQL:

```
# Database configuration
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'smart_university',
        'USER': 'chamseddine',
        'PASSWORD': '123456789',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

Listing 4.2: Database configuration in `settings.py`

- **In this configuration**:
  - `ENGINE`: Specifies the database backend to use (e.g., PostgreSQL).
  - `NAME`: Specifies the name of the database.
  - `USER`: Specifies the username for accessing the database.
  - `PASSWORD`: Specifies the password for accessing the database.
  - `HOST`: Specifies the host where the database server is running.
  - `PORT`: Specifies the port on which the database server is listening.

.3 Static Files Configuration

The `STATIC_URL` and `STATICFILES_DIRS` settings are used to configure static files in our project [16]. Static files include CSS, JavaScript, and image files that are served directly to users' browsers. Here's an example configuration:

```
# Static files configuration
STATIC_URL = 'static/'
MEDIA_URL='/images/'
STATICFILES_DIRS=[os.path.join(BASE_DIR,'static')]
```

Listing 4.3: Static files configuration in `settings.py`

- **In this configuration**:
  - `STATIC_URL`: Specifies the URL prefix for static files.
  - `MEDIA_URL`: Specifies the URL prefix for media files (e.g., images).
  - `STATICFILES_DIRS`: Specifies the directories where Django will look for static files. In this example, `STATICFILES_DIRS` points to a directory named 'static' within the project directory.

The 'static' folder typically contains CSS, JavaScript, and image files used in our project's frontend. It's organized into subfolders based on file types and provides a convenient way to manage and serve static assets.

After preparing the project folder with its different applications and setting up its configurations to serve our needs, the next step is database tables creation.

Database Models

The database plays an essential role across numerous applications, serving as the cornerstone of data storage and connectivity. Particularly in server environments, it stands as a primary necessity, managing the organization and interlinking of information. Essentially, it forms the foundation of a website's infrastructure.

In many server setups, SQL (which stands for Structured Query Language) is often utilized for creating databases, although it has a more challenging learning curve. However, with Django, this process takes a different path. Django empowers developers by enabling them to create databases directly within the `models.py` file of their application's folder. Here, tables are represented as classes, with their respective fields serving as attributes. This change in approach simplifies the database creation process, making it more intuitive.

As we progress through the journey of creating our website's database, understanding these concepts becomes increasingly accessible and comprehensible.

.1 Database Design

Before we go ahead and start creating our database models (tables), let's first explain how the database is structured and linked in our smart university project, as shown in the following diagram:

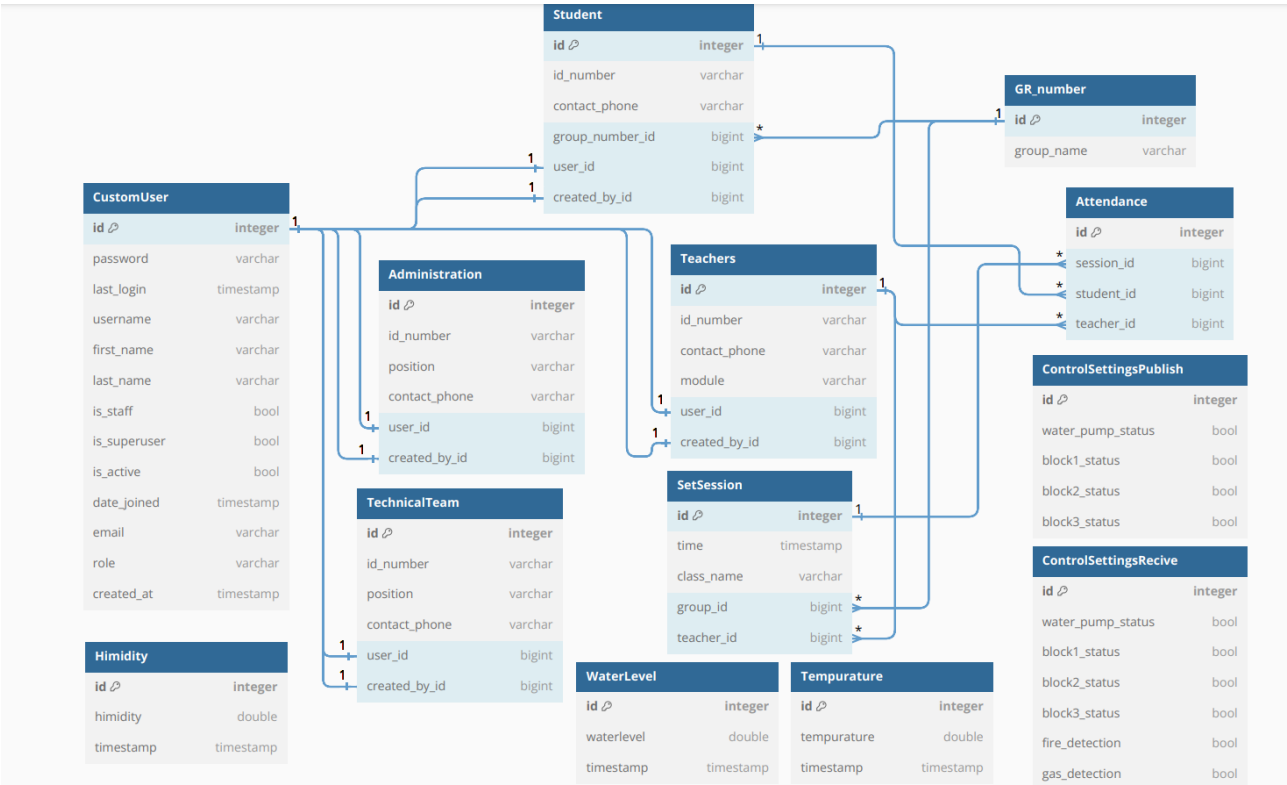


Figure 4.1: Database representation

This diagram illustrates the different database tables that we have in our project. Let's go ahead and explain each one of them:

1. **Custom User**:

   Basically, Django provides us with a simple database user model (table) that contains only the username, password, is_staff, is_active, and is_superuser fields. This simple user model is sufficient for small applications, but in larger applications that require users to have roles and extra information, the Django user model can be overridden to create a new custom user model. This custom user model includes the default Django user details plus additional fields. In our case, we added the email and role fields.

   This is important for providing authorities to different users based on their roles. Additionally, during the login process, this step is necessary to redirect each user to their profile depending on their role type (Student,Teacher ,Admin,Technical Team).

   Once we create these custom users, we need to move on and create different types of profiles (Student, Teacher, Administration, Technical Team) and then assign those profiles to the custom users. In our project, we have four different profile models (tables):

   - **Teacher**:

     This profile stores the information of the teacher users, including:

     – `user_id`: the ID of the custom user this profile belongs to (assigned to a custom user with the role of teacher).

     – `module`: represents the module that the teacher teaches.

     – `contact_phone`: the phone number of the teacher.

     – `id_number`: the ID number of the teacher's card.

- `created_by_id`: the ID of the custom user that creates this teacher (typically, the custom user's role will be admin).

- **Student**:

  This profile stores the information of the student users, including:

  - `user_id`: the ID of the custom user this profile belongs to (assigned to a custom user with the role of student).
  - `contact_phone`: the phone number of the student.
  - `id_number`: the ID number of the student's card.
  - `created_by_id`: the ID of the custom user that creates this student (typically, the custom user's role will be admin).
  - `group_number_id`: the ID of the group that the student belongs to.

- **Administration**:

  This profile stores the information of the administration users, including:

  - `user_id`: the ID of the custom user this profile belongs to (assigned to a custom user with the role of admin).
  - `position`: the position of the user in the administration.
  - `contact_phone`: the phone number of the admin user.
  - `id_number`: the ID number of the admin's card (if we want to provide attendance recording for admins).
  - `created_by_id`: the ID of the custom user that creates this admin (typically, the custom user's role will be admin, typically a superuser admin indicated by `is_superuser = True`).

- **Technical Team**:

  This profile stores the information of the technical team users, including:

  - `user_id`: the ID of the custom user this profile belongs to (assigned to a custom user with the role of technical team).
  - `position`: the position of the technical team user in the technical group (head, normal employee, etc.).
  - `contact_phone`: the phone number of the technical team user.
  - `id_number`: the ID number of the technical team member's card (if we want to provide attendance recording for admins).
  - `created_by_id`: the ID of the custom user that creates this technical team member (typically, the custom user's role will be technical team).

Note: As illustrated in the diagram, the relationship between each of the (`created_by_id`, `user_id`) fields in the profiles models (Student, Teacher, Administration, and Technical Team) and the `id` field of the custom user is one-to-one. This means that only one custom user is accepted per each profile, and only one admin can create a particular profile. This helps us keep track of the profiles created by each admin. Another note is when we come to the administration part, especially in users creation, we will see how the profile is created automatically by creating users and how the admin that creates a particular user is registered in the `created_by` field.

2. `GR_number`:

   In this database model (table), we store the information of the groups of the promotion in the university. It typically contains:

   - `group_name`: represents the number of the group (e.g., G1, G2, G3, etc.).

   The `GR_number` ID field has a one-to-many relation with `group_number_id` in the Student model and with `group_id` in the SetSession model (which will be explained in a moment). This means that many students can belong to the same group, and also many sessions can be associated with one group.

3. `SetSession`:

   This model stores the information that the administration fills on their website according to the session scheduling. It contains:

   - `time`: the time of the session.
   - `class_name`: the name of the class in the university (e.g., C115, C116, etc.).
   - `teacher_id`: the ID number of the teacher's card that is going to teach the session.
   - `group_id`: the ID of the `GR_number` that the teacher will teach (e.g., if G1 has ID 0 and this group is associated with the session, then `group_id` = 0).

4. `Attendance`:

   This model is created to store the attendance list (absences) and it contains the following fields:

   - `session_id`: the ID of the `SetSession` model the absence was recorded in.
   - `student_id`: the ID of the student that was absent (if not absent, it will be null).
   - `teacher_id`: the ID of the teacher that was absent (if not absent, it will be null).

   The many-to-one relationship between `(session_id, student_id, teacher_id)` and the ID of each of the `SetSession`, `Teacher`, `Student` models indicates that in many attendance recordings, the same teacher or student can be absent, and the same `SetSession` can be used in many attendance recordings of different students with one teacher per one `Session`.*

5. `ControlSettingsPublish`:

   This model is used to store the different parameters set up by the technical team and to be published to the Raspberry Pi. It typically contains:

   - `water_pump_status`: The status of the water pump (On or Off).
   - `block1_status`: The status of the power supply to block 1.
   - `block2_status`: The status of the power supply to block 2.
   - `block3_status`: The status of the power supply to block 3.

6. `ControlSettingsReceive`:

   This model is used to store the different parameters received from the Raspberry Pi and to be shown to the technical team. These parameters include:

   - `water_pump_status`: The status of the water pump (On or Off).
   - `block1_status`: The status of the power supply to block 1.

- `block2_status`: The status of the power supply to block 2.
- `block3_status`: The status of the power supply to block 3.
- `fire_detection`: The fire detection status in the laboratory.
- `gas_detection`: The gas detection status in the laboratory.

7. `Temperature`:

   This model is to store the humidity record of the laboratory received from the Raspberry Pi and to be streamed to the technical team. It contains the following fields:

   - `temperature`: The value of the humidity.
   - `timestamp`: The time of the record.

8. `Humidity`:

   This model is to store the temperature record of the laboratory received from the Raspberry Pi and to be streamed to the technical team. It contains the following fields:

   - `humidity`: The value of the temperature.
   - `timestamp`: The time of the record.

9. `WaterLevel`:

   This model is to store the water level record of the water tank received from the Raspberry Pi and to be streamed to the technical team. It contains the following fields:

   - `water_level`: The percentage of the water in the tank.
   - `temperature`: The value of the temperature.

.2 Database Build with Django:

After understanding the database structure and the role of each model, let's implement them in `models.py`. Each application in our project will have its database setup, so we will create these models for each application's `models.py`:

Users Application:

First, we import necessary packages:

```
from django.db import models
from django.contrib.auth.models import AbstractUser
from django.utils.translation import gettext_lazy as _
```

Listing 4.4: Import necessary packages in `models.py`

Then we start building our custom user using the following code:

```
class CustomUser(AbstractUser):
    class Role(models.TextChoices):
        ADMIN = "ADMIN", _("Admin")
        STUDENT = "STUDENT", _("Student")
        TEACHER = "TEACHER", _("Teacher")
        TECHNICAL_TEAM = "TECHNICAL_TEAM", _("Technical Team")
    email = models.EmailField(unique=True)
    role = models.CharField(max_length=50, choices=Role.choices, default=
   Role.ADMIN)

    def __str__(self):
```

```
        return f"{self.username} - {self.role}"
```

Listing 4.5: Build custom users in `models.py`

In this code:

- We define a custom user model named `CustomUser`, which inherits from Django's built-in `AbstractUser` model. This allows us to extend the functionality of the default user model.
- We define a nested class `Role`, which contains choices for the role of the user. Each choice is a tuple containing a code and a human-readable name, wrapped with `_()` for translation.
- The `email` field is defined as an `EmailField` to store the email address of the user. It is set as unique to ensure each email is associated with only one user.
- The `role` field is defined as a `CharField` with choices set to the options defined in the `Role` class. It represents the role of the user and has a default value of `ADMIN`.
- The `__str__()` method is overridden to return a string representation of the user object, displaying the username and the assigned role.

user_interface Application:

First, we import necessary packages:

```
from django.db import models
from django.utils import timezone
from users.models import CustomUser
```

Listing 4.6: Import necessary packages in `models.py`

Then we create our models:

1. `GR_number` model:

```
class GR_number(models.Model):
    group_name = models.CharField(max_length=100)

    def __str__(self):
        return self.group_name
```

Listing 4.7: Build `GR_number` model in `models.py`

In this code:

- The `GR_number` model represents a group and has a single field `group_name` to store the name of the group.
- The `__str__()` method returns the group's name, providing a readable representation of the group.

2. `Student` model:

```
class Student(models.Model):
    user = models.OneToOneField(
        CustomUser,
        on_delete=models.CASCADE,
        limit_choices_to={'role': CustomUser.Role.STUDENT},
        related_name='student_profiles',
        null=True,
        blank=False
    )
    created_by= models.OneToOneField(
        CustomUser,
```

```
        on_delete=models.SET_NULL,
        limit_choices_to={'role': CustomUser.Role.ADMIN},
        related_name='created_students',
        null=True,
        blank=False
    )
    id_number = models.CharField(max_length=20, default="0123456789")
    contact_phone = models.CharField(max_length=15, default="213-540-028-098")
    group_number = models.ForeignKey(GR_number, on_delete=models.SET_NULL, null=True)

    def __str__(self):
        return f'{self.user.first_name} {self.user.last_name}'
```

Listing 4.8: Build `Student` model in `models.py`

In this code:

- The `Student` model has a one-to-one relationship with the `CustomUser` model, ensuring each student profile is linked to a unique user.
- The `created_by` field is also a one-to-one relationship with `CustomUser`, indicating the admin who created the student profile.
- Additional fields include `id_number`, `contact_phone`, and `group_number`, the latter being a foreign key to the `GR_number` model.
- The `__str__()` method returns the student's full name.

3. `Teacher` model:

```
class Teacher(models.Model):
    user = models.OneToOneField(
        CustomUser,
        on_delete=models.CASCADE,
        limit_choices_to={'role': CustomUser.Role.TEACHER},
        related_name='teacher_profiles',
        null=True,
        blank=False
    )
    id_number = models.CharField(max_length=20, default="0123456789")
    module = models.CharField(max_length=100, default='not_yet')
    contact_phone = models.CharField(max_length=15, default="213-540-028-098")
    created_by= models.OneToOneField(
        CustomUser,
        on_delete=models.SET_NULL,
        limit_choices_to={'role': CustomUser.Role.ADMIN},
        related_name='created_teachers',
        null=True,
        blank=False
    )

    def __str__(self):
        return f'{self.user.first_name} {self.user.last_name}'
```

Listing 4.9: Build `Teacher` model in `models.py`

In this code:

- The `Teacher` model links to `CustomUser` with a role limited to "TEACHER".

- It includes fields like `id_number`, `module`, and `contact_phone`.
- The `created_by` field links to the admin who created the teacher profile.
- The `__str__()` method returns the teacher's full name.

4. `Administration` model:

```
class Administration(models.Model):
    user = models.OneToOneField(
        CustomUser,
        on_delete=models.CASCADE,
        limit_choices_to={'role': CustomUser.Role.ADMIN},
        related_name='administration_profiles',
        null=True,
        blank=False
    )
    created_by= models.OneToOneField(
        CustomUser,
        on_delete=models.SET_NULL,
        limit_choices_to={'role': CustomUser.Role.ADMIN},
        related_name='created_admin_team',
        null=True,
        blank=False
    )
    id_number = models.CharField(max_length=20, default="0123456789")
    position = models.CharField(max_length=100)
    contact_phone = models.CharField(max_length=15, default="
   213-540-028-098")

    def __str__(self):
        return f'{self.user.first_name} {self.user.last_name}'
```

Listing 4.10: Build `Administration` model in `models.py`

In this code:

- The `Administration` model links to `CustomUser` with a role limited to "ADMIN".
- It includes fields like `id_number`, `position`, and `contact_phone`.
- The `created_by` field links to the admin who created the administration profile.
- The `__str__()` method returns the admin's full name.

5. `TechnicalTeam` model:

```
class TechnicalTeam(models.Model):
    user = models.OneToOneField(
        CustomUser,
        on_delete=models.CASCADE,
        limit_choices_to={'role': CustomUser.Role.TECHNICAL_TEAM},
        related_name='technicalteam_profiles',
        null=True,
        blank=False
    )
    id_number = models.CharField(max_length=20, default="0123456789")
    position = models.CharField(max_length=100)
    contact_phone = models.CharField(max_length=15, default="
   213-540-028-098")
    created_by= models.OneToOneField(
        CustomUser,
        on_delete=models.SET_NULL,
```

```
        limit_choices_to={'role': CustomUser.Role.ADMIN},
        related_name='created_technical_team',
        null=True,
        blank=False
    )

    def __str__(self):
        return f'{self.user.first_name} {self.user.last_name}'
```

Listing 4.11: Build `TechnicalTeam` model in `models.py`

In this code:

- The `TechnicalTeam` model links to `CustomUser` with a role limited to "TECHNICAL_TEAM".
- It includes fields like `id_number`, `position`, and `contact_phone`.
- The `created_by` field links to the admin who created the technical team profile.
- The `__str__()` method returns the technical team member's full name.

6. `SetSession` model:

```
class SetSession(models.Model):
    group = models.ForeignKey(GR_number, on_delete=models.CASCADE)
    teacher = models.ForeignKey(Teacher, on_delete=models.CASCADE)
    time = models.DateTimeField(default=timezone.now)
    class_name = models.CharField(max_length=50, choices=[('class_1', '
    class_1'), ('class_2', 'class_2')])

    def __str__(self):
        return f'{self.time}'
```

Listing 4.12: Build `SetSession` model in `models.py`

In this code:

- The `SetSession` model includes a foreign key to `GR_number` and `Teacher`, indicating which group and teacher are associated with the session.
- The `time` field records the session time, and `class_name` specifies the class.
- The `__str__()` method returns the session time.

7. `Attendance` model:

```
class Attendance(models.Model):
    session = models.ForeignKey(SetSession, on_delete=models.CASCADE)
    teacher = models.ForeignKey(Teacher, on_delete=models.SET_NULL, null
    =True, blank=True)
    student = models.ForeignKey(Student, on_delete=models.SET_NULL, null
    =True, blank=True)

    def __str__(self):
        return f'{self.session}'
```

Listing 4.13: Build `Attendance` model in `models.py`

In this code:

- The `Attendance` model includes foreign keys to `SetSession`, `Teacher`, and `Student`, tracking which teacher or student attended a specific session. When filling these fields, one of them will be null since we want to register either a student or a teacher for each object (row) of this model (table) in the database.

- The `__str__()` method returns the session information.

data_collect Application:

First, we import necessary packages:

```
from django.db import models
```

Listing 4.14: Import necessary packages in `models.py`

Then we create our models:

1. `ControlSettingsPublish` model:

```
class ControlSettingsPublish(models.Model):
    water_pump_status = models.BooleanField(default=False)
    block1_status = models.BooleanField(default=False)
    block2_status = models.BooleanField(default=False)
    block3_status = models.BooleanField(default=False)

    def __str__(self):
        return "Control Settings"
```

Listing 4.15: Build `ControlSettingsPublish` model in `models.py`

In this code:

- The `ControlSettingsPublish` model has four Boolean fields representing the status of various components (water pump and blocks 1 to 3).
- The `__str__()` method returns a string "Control Settings" for easy identification.

2. `ControlSettingsRecive` model:

```
class ControlSettingsRecive(models.Model):
    water_pump_status = models.BooleanField(default=False)
    block1_status = models.BooleanField(default=False)
    block2_status = models.BooleanField(default=False)
    block3_status = models.BooleanField(default=False)
    fire_detection = models.BooleanField(default=False)
    gas_detection = models.BooleanField(default=False)

    def __str__(self):
        return "Control Settings"
```

Listing 4.16: Build `ControlSettingsRecive` model in `models.py`

In this code:

- The `ControlSettingsReceive` model extends the `ControlSettingsPublish` model by adding fields for fire and gas detection. We will access this model when receiving information from the Raspberry Pi to display it in real-time on the technical team dashboard.
- The `__str__()` method returns a string "Control Settings" for easy identification.

3. `Temperature` model:

```
class Temperature(models.Model):
    temperature = models.FloatField(default=0.0)
    timestamp = models.DateTimeField(auto_now_add=True)
```

Listing 4.17: Build `Temperature` model in `models.py`

In this code:

- The `Temperature` model has a `FloatField` for storing temperature readings and a `DateTimeField` to automatically record the timestamp when a reading is created.

4. `Humidity` model:

```
class Humidity(models.Model):
    humidity = models.FloatField(default=0.0)
    timestamp = models.DateTimeField(auto_now_add=True)
```

Listing 4.18: Build `Humidity` model in `models.py`

In this code:

- The `Humidity` model has a `FloatField` for storing humidity readings and a `DateTimeField` to automatically record the timestamp when a reading is created.

5. `WaterLevel` model:

```
class WaterLevel(models.Model):
    waterlevel = models.FloatField(default=0.0)
    timestamp = models.DateTimeField(auto_now_add=True)
```

Listing 4.19: Build `WaterLevel` model in `models.py`

In this code:

- The `WaterLevel` model has a `FloatField` for storing water level readings and a `DateTimeField` to automatically record the timestamp when a reading is created.

.3 Running Migrations:

After completing this process, we open the command line at the same directory as the `manage.py` file and run:

```
python manage.py makemigrations
```

This command inspects the `models.py` files in each application for any changes and prepares migration files, which are used to apply those changes to the database schema.

Then we run:

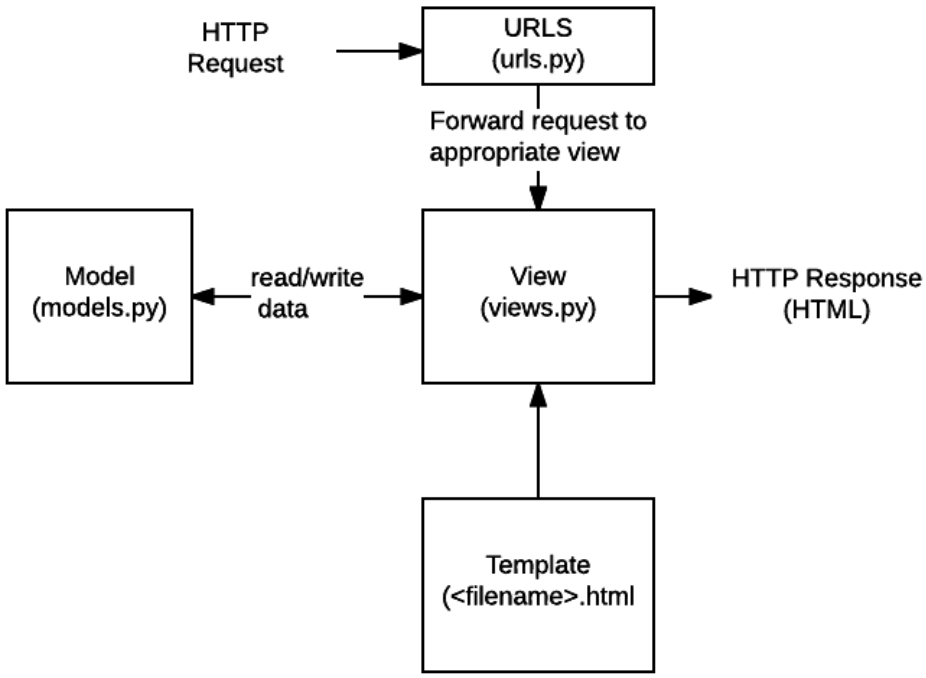
```
python manage.py migrate
```

This command applies the migrations generated by `makemigrations` to the database, creating the tables and columns as defined in the models.

After doing this, we can move on to the next step and start working on the website backend.

Website Backend Functionalities

To begin implementing our Django backend, it's essential to understand how Django handles requests. The following image illustrates the process:



When the user requests a specific URL root, it sends a request to `urls.py`. Then, `urls.py` forwards this request to the appropriate function in `views.py`. This function can either return some information from the database, or registers information to it (communicating with `models.py` for this purpose), or perform another task. We will understand how these functions handle requests as we proceed with the implementation of our Django backend.

.1 Login and Logout

Login

The login process is a critical step in any web application. In Django, it is implemented in the following function:

```
@notLoggedUsers
def login_user(request):
    if request.method == 'POST':
        username = request.POST.get('username')
        password = request.POST.get('password')
        user = authenticate(request, username=username, password=password)
        if user is not None:
            if user.role == 'STUDENT':
                login(request, user)
                return redirect('student_dashboard')  # Redirect to student
    dashboard
            elif user.role == 'ADMIN':
                login(request, user)
                return redirect('admin_dashboard')  # Redirect to admin
    dashboard
            elif user.role == 'TEACHER':
                login(request, user)
                return redirect('teacher_dashboard')  # Redirect to teacher
    dashboard
            elif user.role == 'TECHNICAL_TEAM':
                login(request, user)
                return redirect('technical_team_dashboard')  # Redirect to
    technical team dashboard
```

```
20      return render(request, 'AI/user_registration/login_student.html')
```

Listing 4.20: Login function in `views.py`

Explanation:

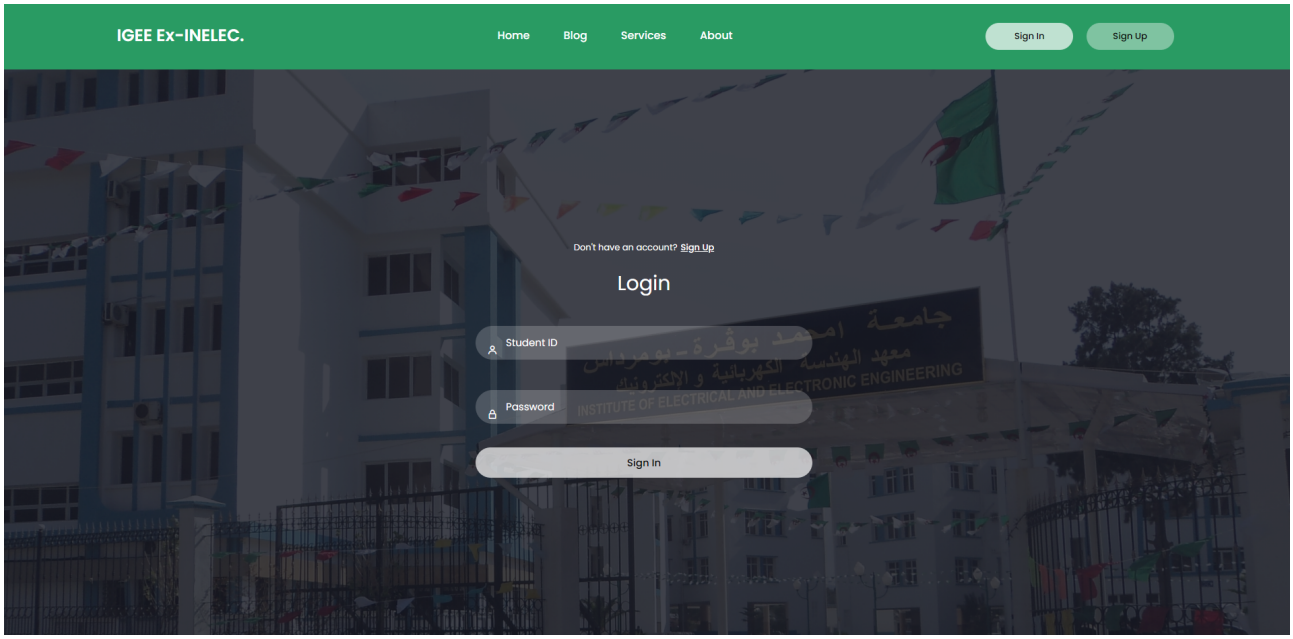
The `@notLoggedUsers` decorator ensures that already logged-in users are redirected to their profile pages instead of the login page.

In the `login_user` function, we check if the request method is POST, which indicates a form submission with login credentials. We then attempt to authenticate the user using the provided username and password.

If the authentication is successful and the user's role is identified as STUDENT, ADMIN, TEACHER, or TECHNICAL_TEAM, the user is logged in using Django's `login` function, and the function redirects the user to the appropriate dashboard based on their role.

If the authentication fails or the request method is not POST, the function renders the login page template for the user to retry logging in.

Below is an image of the login page:



This is how the `login_user` function is configured in `urls.py`:

```
1  from django.urls import path, include
2  from . import views
3
4  urlpatterns = [
5      path('login/', views.login_user, name='login'),
6  ]
```

Listing 4.21: Configuring `login_user` in `urls.py`

Explanation:

In the `urls.py` file, we define URL patterns that map URLs to the corresponding view functions. Here, we map the URL `'login/'` to the `login_user` view function and name this URL pattern `'login'`. This setup allows users to access the login functionality by navigating to the `'login/'` URL.

Logout

The logout process is implemented in the following function:

```
def userLogout(request):
    logout(request)
    return redirect('login')
```

Listing 4.22: Logout function in `views.py`

This is how the `userLogout` function is configured in `urls.py`:

```
from django.urls import path, include
from . import views

urlpatterns = [
    path('logout/', views.userLogout, name='logout'),
]
```

Listing 4.23: Configuring `userLogout` in `urls.py`

Explanation:

The `userLogout` function logs out the current user by calling Django's built-in `logout` function, and then redirects the user to the login page.

In `urls.py`, we map the URL `'logout/'` to the `userLogout` view function and name this URL pattern `'logout'`. This setup allows users to log out by navigating to the `'logout/'` URL, which will call the `userLogout` function, log out the user, and redirect them to the login page.

after completiting login and logout steps we the need to understand each one of the profiles :

.2 Administration Dashboard

creating users or scheduling session

As explained before, the administration part is responsible for either creating users or scheduling sessions by selecting time, group, teacher, and class name. These tasks are accomplished by the following view function:

```

@login_required(login_url='login')
@forAdmins
def admin_dashboard(request):
    admin_info = {
        'first_name': request.user.first_name,
        'last_name': request.user.last_name,
    }

    if request.method == 'POST':
        form = SetSessionForm(request.POST)
        form1 = UserRegistrationForm(request.POST)

        if form.is_valid():
            form.save()
        else:
            print("SetSessionForm errors:", form.errors)

        if form1.is_valid():
            # Extract data from form submission
            id_number = request.POST.get('id_number')
            contact_phone = request.POST.get('contact_phone')
```

```
            group_name = request.POST.get('group_name')
            Module = request.POST.get('Module')
            position = request.POST.get('position')
            is_supervisor = request.POST.get('is_supervisor')

            if form1.cleaned_data['role'] == "STUDENT":
                user = form1.save()
                print('User created')
                group_number = GR_number.objects.get(group_name=group_name)
                Student.objects.create(
                    created_by=request.user,
                    user=user,
                    id_number=id_number,
                    contact_phone=contact_phone,
                    group_number=group_number
                )

            if request.user.is_superuser and form1.cleaned_data['role'] == "
    ADMIN":
                if is_supervisor == 'YES':
                    user = form1.save(is_staff=True, is_superuser=True)
                elif is_supervisor == 'NO':
                    user = form1.save()
                Administration.objects.create(
                    created_by=request.user,
                    user=user,
                    id_number=id_number,
                    contact_phone=contact_phone,
                )

            if form1.cleaned_data['role'] == "TEACHER":
                user = form1.save()
                print('User created')
                Teacher.objects.create(
                    created_by=request.user,
                    user=user,
                    id_number=id_number,
                    contact_phone=contact_phone,
                    module=Module.lower(),
                )

            if form1.cleaned_data['role'] == "TECHNICAL_TEAM":
                user = form1.save()
                print('User created')
                TechnicalTeam.objects.create(
                    created_by=request.user,
                    user=user,
                    id_number=id_number,
                    position=position.lower(),
                    contact_phone=contact_phone,
                )
        else:
            print("UserRegistrationForm errors:", form1.errors)
    else:
        form = SetSessionForm()
        form1 = UserRegistrationForm()

```

```
79      return render(request, 'AI/administration/dashboard_administration.html'
        , {'form': form, 'form1': form1, 'admin_info': admin_info})
```

Listing 4.24: Admin Dashboard function in `views.py`

Explanation:

- `@login_required(login_url='login')`: This decorator ensures that only logged-in users can access the `admin_dashboard` view. If a user is not logged in, they are redirected to the `login` page.
- `@forAdmins`: This custom decorator (defined in `decorators.py`) restricts access to this view to users with admin privileges.
- `admin_info`: This dictionary stores the first and last names of the logged-in admin user to display in the dashboard.
- `if request.method == 'POST'`: This checks if the request is a POST request, which indicates that the admin is submitting a form (either to set a session or register a user).
- `form = SetSessionForm(request.POST)` and `form1 = UserRegistrationForm(request.POST)`: These lines instantiate the session setting form and user registration form with the submitted data.
- `if form.is_valid()`: This checks if the session form is valid. If valid, the session details are saved. If not valid, it is assumed the user is submitting a user registration form.
- `if form1.is_valid()`: This checks if the user registration form is valid. If valid, the form data is processed to create the user and their associated profile.
- Extracting data from the form submission: Variables like `id_number`, `contact_phone`, `group_name`, `Module`, `position`, and `is_supervisor` are extracted from the request to create a user.
- `if form1.cleaned_data['role'] == "STUDENT"`: This block handles student creation. It creates a `Student` object with the submitted data, automatically linking the created user to the student profile and setting the `created_by` field to the currently logged-in admin.
- `if request.user.is_superuser and form1.cleaned_data['role'] == "ADMIN"`:

  This block specifically handles the creation of admin users. It first verifies whether the logged-in user is a superuser and if the selected role in the form is "ADMIN". This check ensures that only admins with superuser privileges can create other admin users.

  Based on the `is_supervisor` value submitted in the form, the function distinguishes between two types of admin users: supervisors and regular admins. If `is_supervisor` is set to "YES", it indicates that the new admin user will have additional authority. In this case, the created user is assigned both `is_superuser` and `is_staff` attributes as `True`, granting them enhanced privileges within the system. Conversely, if `is_supervisor` is set to "NO", a regular admin user is created without the additional authority.

  After creating the admin user, the function proceeds to create an `Administration` object assigned to this user with the submitted data, ensuring proper management of administrative roles within the system.
- `if form1.cleaned_data['role'] == "TEACHER"`: This block handles teacher creation. It creates a `Teacher` object with the submitted data, linking the created user to the teacher profile and setting the `created_by` field to the currently logged-in admin.

- `if form1.cleaned_data['role'] == "TECHNICAL_TEAM"`: This block handles technical team member creation. It creates a `TechnicalTeam` object with the submitted data, linking the created user to the technical team profile and setting the `created_by` field to the currently logged-in admin.
- `else`: If the request is not a POST request, empty forms for setting sessions and user registration are instantiated.
- `return render(request, 'AI/administration/dashboard_administration.html', {'form': form, 'form1': form1, 'admin_info': admin_info})`: This renders the administration dashboard template with the forms and admin information.

Forms in Django

In Django, forms are used to handle user input. The following code shows how the forms are created in `forms.py` and then used in the view function.

```
class UserRegistrationForm(UserCreationForm):
    email = forms.EmailField(help_text='A valid email address, please.', required=True)
    role = forms.ChoiceField(choices=CustomUser.Role.choices)  # Add role field to the form

    class Meta:
        model = get_user_model()
        fields = ['first_name', 'last_name', 'username', 'email', 'password1', 'password2']

    def save(self, commit=True, is_active=True, is_superuser=False, is_staff=False):
        user = super(UserRegistrationForm, self).save(commit=False)
        user.email = self.cleaned_data['email']
        user.role = self.cleaned_data['role']  # Assign the role from form data
        user.is_active = is_active
        user.is_superuser = is_superuser
        user.is_staff = is_staff
        if commit:
            user.save()
        return user

class SetSessionForm(forms.ModelForm):
    class Meta:
        model = SetSession
        fields = '__all__'
```

Listing 4.25: User Registration and Set Session Forms in `forms.py`

Explanation:

- `class UserRegistrationForm(UserCreationForm)`: This class defines a form for user registration by extending Django's built-in `UserCreationForm`. It adds custom fields and validation.
- `email = forms.EmailField(help_text='A valid email address, please.', required=True)`: This field requires the user to input a valid email address. The `help_text` parameter provides a hint to the user.
- `role = forms.ChoiceField(choices=CustomUser.Role.choices)`: This field allows the user to select a role from predefined choices. The choices are defined in the `CustomUser.Role`

class.

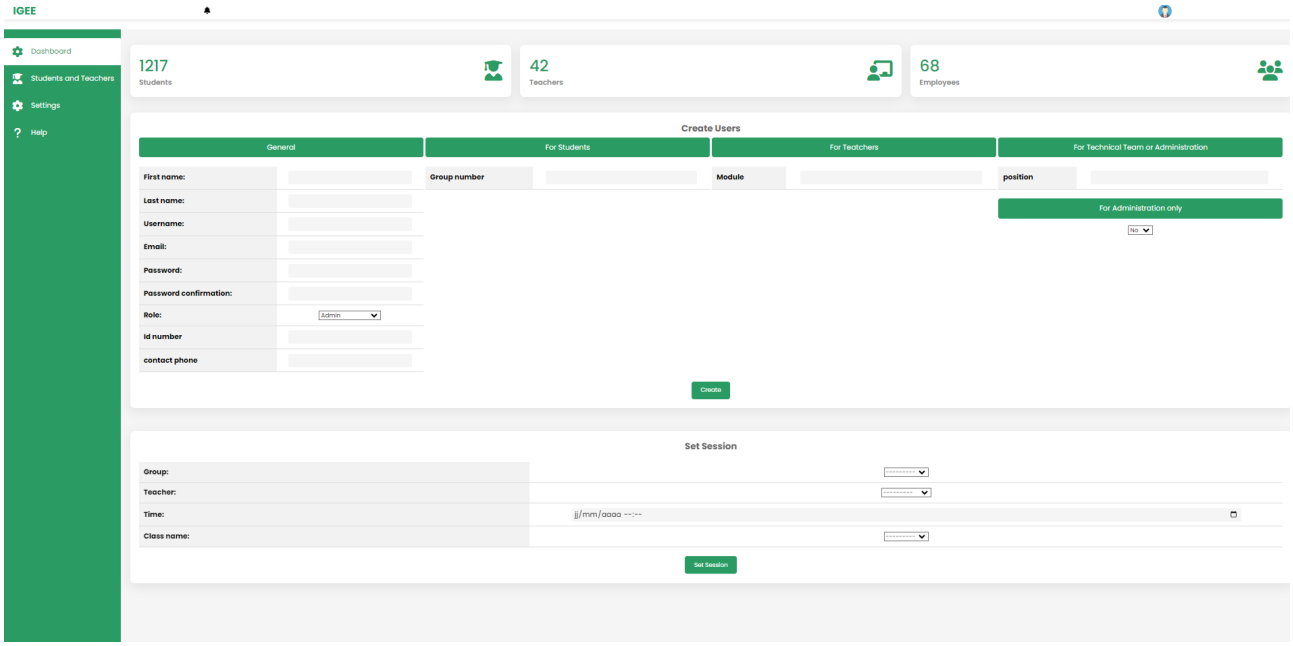
- `class Meta`: This inner class defines metadata for the form. It specifies the model to be used (`get_user_model()`) and the fields to include in the form (`first_name`, `last_name`, `username`, `email`, `password1`, and `password2`).
- `def save(self, commit=True, is_active=True, is_superuser=False, is_staff=False)`: This method overrides the default save method to add additional functionality. It saves the user's email and role, and sets the user's `is_active`, `is_superuser`, and `is_staff` status based on the parameters provided.
- `user = super(UserRegistrationForm, self).save(commit=False)`: This line calls the parent class's save method to create a user object without committing it to the database immediately.
- `if commit: user.save()`: If the `commit` parameter is `True`, the user object is saved to the database.
- `return user`: The user object is returned after being saved.

Explanation of `SetSessionForm`:

- `class SetSessionForm(forms.ModelForm)`: This class defines a form for setting up a session, extending Django's `ModelForm`. The `ModelForm` class in Django automatically generates a form based on the fields of a specified model.
- `class Meta`: This inner class defines metadata for the `SetSessionForm`. It specifies which model to use and which fields to include in the form.
- `model = SetSession`: This specifies that the form is based on the `SetSession` model.
- `fields = __all__`: This indicates that all fields from the `SetSession` model should be included in the form. It means that every attribute defined in the `SetSession` model will be represented as a form field.

These forms are then imported and used in the view function to handle user input for registering users and setting sessions.

and this is how the Dashboard looks to the administration :



For URLs configuration, we use:

```python
from django.urls import path, include
from . import views

urlpatterns = [
    path('admin_dashboard/', views.admin_dashboard, name='admin_dashboard'),
]
```

Listing 4.26: Configuring `admin_dashboard` in `urls.py`

In the `urls.py` file, we define URL patterns that map URLs to the corresponding view functions. Here, we map the URL `'admin_dashboard/'` to the `admin_dashboard` view function and name this URL pattern `'admin_dashboard'`. This setup allows users to access the administration dashboard functionality by navigating to the `'admin_dashboard/'` URL.

Attendance List

To grant the administration team access to the attendance lists of both teachers and students, the following function is implemented:

```python
@login_required(login_url='login')
@forAdmins
def admin_attendance_dashboard(request):
    admin_info = {
        'first_name': request.user.first_name,
        'last_name': request.user.last_name,
    }

    # Retrieve all attendance records
    attendances = Attendance.objects.all()

    # Create lists to store session information for rendering in the template
    session_teachers_info = []
    session_students_info = []

    # Loop through the attendance records
    for attendance in attendances:
        if attendance.teacher is not None:
            # Extract session data for teachers
            session_data = {
                'group': attendance.session.group.group_name,
                'teacher_name': f'{attendance.teacher.user.first_name} {attendance.teacher.user.last_name}',
                'class_name': attendance.session.class_name,
                'module': attendance.teacher.module,
                'time': attendance.session.time,
            }
            session_teachers_info.append(session_data)

    # Loop through the attendance records again for students
    for attendance in attendances:
        if attendance.student is not None:
            # Extract session data for students
            session_data = {
                'group': attendance.student.group_number,
                'student_name': f'{attendance.student.user.first_name} {attendance.student.user.last_name}',
                'class_teacher': f'{attendance.session.teacher.user.first_name} {attendance.session.teacher.user.last_name}',
                'module': attendance.session.teacher.module,
```

```
                'class_name': attendance.session.class_name,
                'time': attendance.session.time,
            }
            session_students_info.append(session_data)

    return render(request, 'AI/administration/attendance_list.html', {'
    session_teachers_info': session_teachers_info, 'session_students_info':
    session_students_info, 'user_info': admin_info})
```

Listing 4.27: Configuring `admin_attendance_dashboard` in `urls.py`

Explanation:

- **Decorator Usage**:
  - `@login_required(login_url='login')`: Ensures only logged-in users can access the function. Redirects to the login page if not logged in.
  - `@forAdmins`: Restricts access to users with admin privileges.
- **View Function**:
  - `admin_attendance_dashboard(request)`: Renders the attendance dashboard for the administration team.
- **Data Retrieval**:
  - `admin_info`: Stores admin user's first and last names.
  - `attendances = Attendance.objects.all()`: Retrieves all attendance records.
- **Data Processing**:
  - `session_teachers_info` and `session_students_info`: Lists to store session information.
  - Two loops iterate over attendance records:
    - **Teacher Sessions**:
      - Extracts session data for teachers if the attendance record is not empty.
    - **Student Sessions**:
      - Extracts session data for students if the attendance record is not empty.
- **Rendering**:
  - Renders the `attendance_list.html` template with processed attendance information and admin's information.

For URLs configuration, we use:

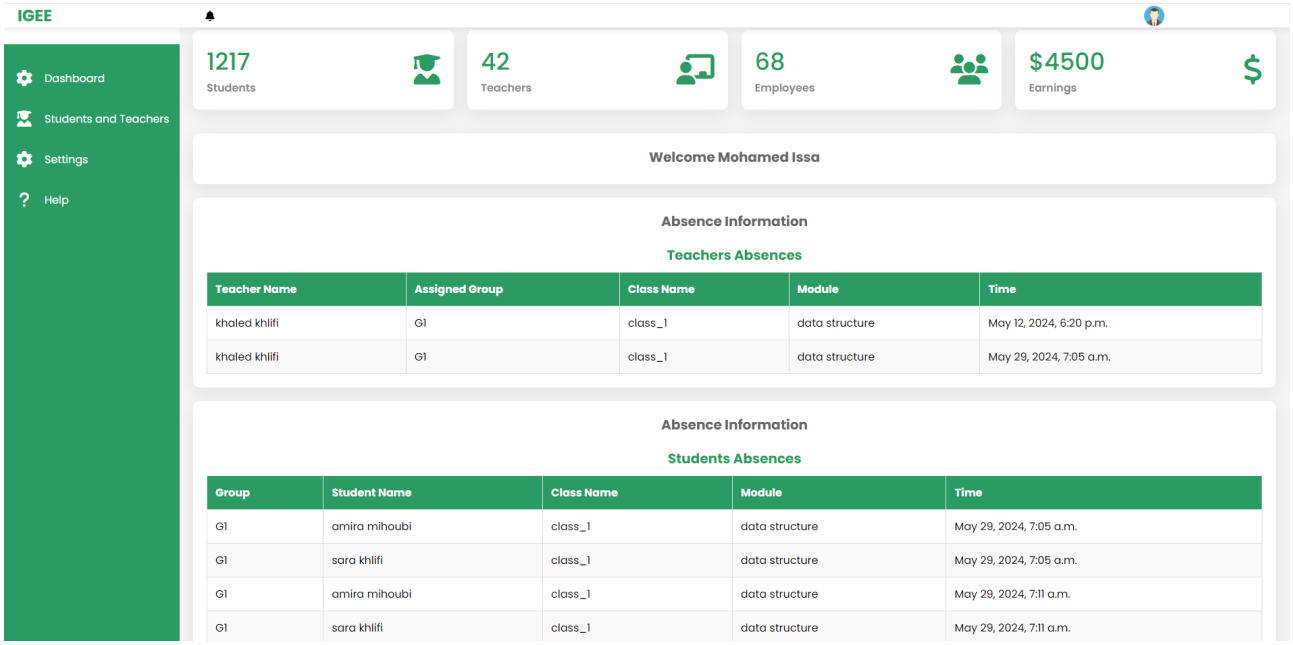
```
from django.urls import path, include
from . import views

urlpatterns = [
    path('admin_attandance_dashboard/', views.admin_attandance_dashboard,
    name='admin_attandance_dashboard'),
]
```

Listing 4.28: Configuring `admin_attandance_dashboard` in `urls.py`

In the `urls.py` file, we define URL patterns that map URLs to the corresponding view functions. Here, we map the URL `'admin_attandance_dashboard/'` to the `admin_attandance_dashboard`

view function and name this URL pattern `'admin_attandance_dashboard'`. This setup allows users to access the administration attendance dashboard functionality by navigating to the `'admin_attandance_dashboard/'` URL.

and this is how the Attendance List Dashboard looks to the administration :



.3 Teacher Dashboard

In this section, teachers can access their attendance lists as well as the attendance lists of the students they teach. The function responsible for handling the teacher dashboard is defined as follows:

```
1
2  @login_required(login_url='login_student')
3  def teacher_dashboard(request):
4      if request.user.role != 'TEACHER':
5          return redirect('login')
6
7      # Get the teacher object for the logged-in user
8      teacher = Teacher.objects.get(user=request.user)
9
10     # Find all sessions that contain this teacher
11     groups = GR_number.objects.all()
12     sessions_taught = SetSession.objects.filter(teacher=teacher)
13
14     # Get the teacher's sessions
15     teacher_attendance = Attendance.objects.filter(teacher=teacher)
16
17     # Create lists to store session information for rendering in the
       template
18     session_teacher_info = []
19     session_students_info = []
20     user_info = {
21         'first_name': request.user.first_name,
22         'last_name': request.user.last_name,
23     }
24
25     # Loop through the teacher's attendance records
26     for attendance in teacher_attendance:
```

```
        session_data = {
            'group': attendance.session.group.group_name,
            'student_name': attendance.session.group.group_name,  # Verify this
            'class_name': attendance.session.class_name,
            'module': teacher.module,
            'time': attendance.session.time,
        }
        session_teacher_info.append(session_data)

    # Loop through the sessions taught by the teacher
    for session in sessions_taught:
        # Find all attendance records for the current session
        attendances = Attendance.objects.filter(session=session)

        # Loop through the attendance records
        for attendance in attendances:
            # Check if the attendance record has a student
            if attendance.student is not None:
                session_data = {
                    'group': attendance.student.group_number.group_name,
                    'student_name': f'{attendance.student.user.first_name} {attendance.student.user.last_name}',
                    'module': session.teacher.module,
                    'class_name': session.class_name,
                    'time': session.time,
                }
                session_students_info.append(session_data)

    return render(request, 'AI/teachers/dashboard_teachers.html', {'session_teacher_info': session_teacher_info, 'session_students_info': session_students_info, 'user_info': user_info})
```

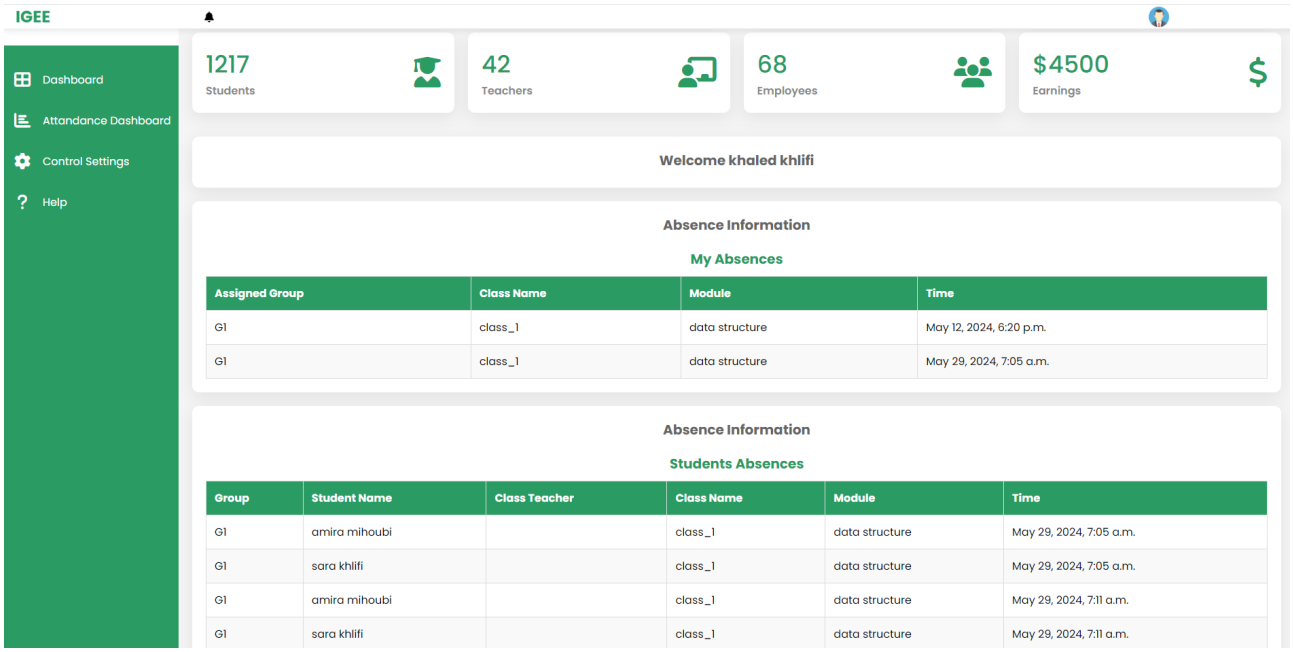
Listing 4.29: Handling Teacher Dashboard in `views.py`

Explanation:

- **Decorator Usage**:
  - `@login_required(login_url='login_student')`: Ensures only logged-in users can access the function. Redirects to the student login page if not logged in.
- **Role Check**:
  - Redirects to the login page if the logged-in user is not a teacher.
- **Data Retrieval**:
  - Retrieves the teacher object for the logged-in user.
  - Finds all sessions that contain this teacher.
  - Retrieves the teacher's attendance records.
- **Data Processing**:
  - Creates lists to store session information for rendering in the template.
  - Loops through the teacher's attendance records and extracts relevant session data.
  - Loops through the sessions taught by the teacher and extracts attendance information for each session's students.

- **Rendering**:
  - Renders the `dashboard_teachers.html` template with the processed attandance information and user's information.

The dashboard of the teacher is shown in the following image:



For URLs configuration, we use:

```
from django.urls import path
from . import views

urlpatterns = [
     path('teacher_dashboard/', views.teacher_dashboard, name='
    teacher_dashboard'),
]
```

Listing 4.30: Configuring `teacher_dashboard` in `urls.py`

In the `urls.py` file, we define URL patterns that map URLs to the corresponding view functions. Here, we map the URL `'teacher_dashboard/'` to the `teacher_dashboard` view function and name this URL pattern `'teacher_dashboard'`. This setup allows users to access the teacher dashboard functionality by navigating to the `'teacher_dashboard/'` URL.

.4 Student Dashboard:

In this section, students can also view their attendance records.

The function that handles the student dashboard is defined as follows:

```
@login_required(login_url='login')
def student_dashboard(request):
    # Check if the user is logged in as a student
    if request.user.role != 'STUDENT':
        return redirect('login')

    # Retrieve the student object associated with the logged-in user
    student = Student.objects.get(user=request.user)

    # Retrieve the attendance objects for the student
```

```
    attendance_objects = Attendance.objects.filter(student=student)

    # Create a list to store attendance information for rendering in the template
    attendance_info = []

    # Get user information
    user_info = {
        'first_name': request.user.first_name,
        'last_name': request.user.last_name,
    }

    # Loop through the attendance objects
    for attendance in attendance_objects:
        # Extract session information for each attendance record
        session_info = {
            'time': attendance.session.time,
            'class_name': attendance.session.class_name,
            'group': attendance.session.group.group_name,
            'teacher': f'{attendance.session.teacher.user.first_name} {attendance.session.teacher.user.last_name}',
            'module': attendance.session.teacher.module,
        }
        # Append session information to the list
        attendance_info.append(session_info)

    # Render the student dashboard template with attendance information and user information
    return render(request, 'AI/students/dashboard_students.html', {'attendance_info': attendance_info,'user_info':user_info})
```

Listing 4.31: Handling Student Dashboard in `views.py`

Explanation:

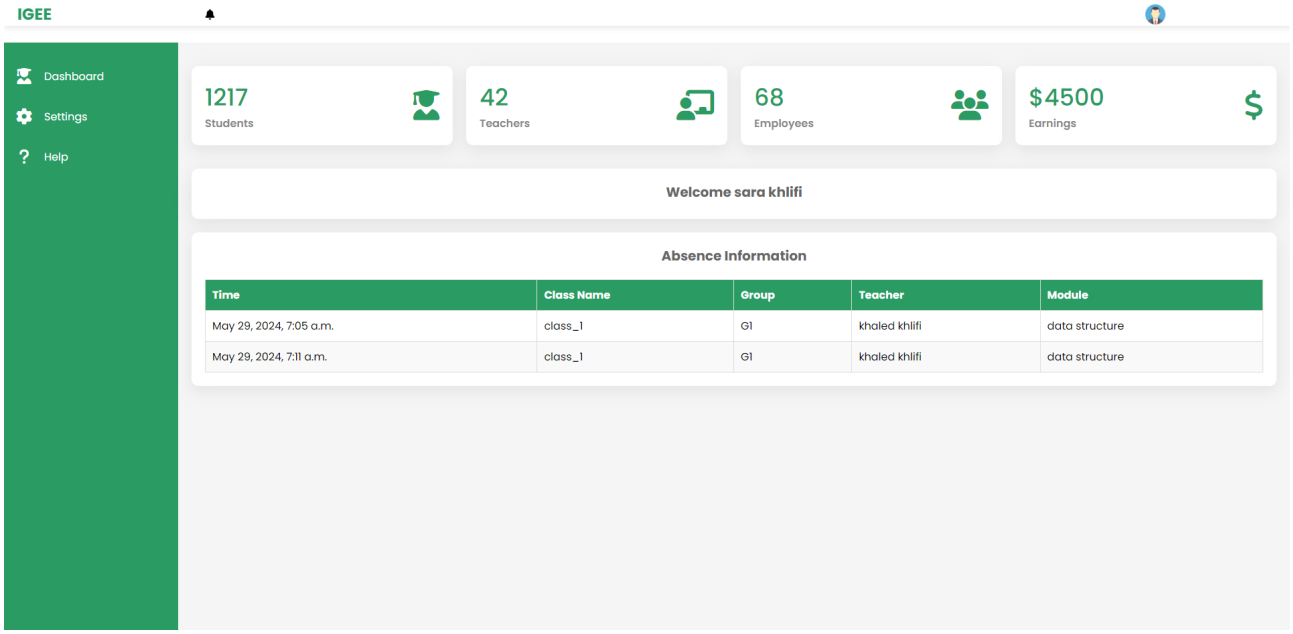
The `student_dashboard` function is responsible for rendering the student dashboard. It first checks if the logged-in user has the role of a student. If not, it redirects them to the student login page.

Next, it retrieves the student object associated with the logged-in user and fetches the attendance records for that student.

Then, it constructs a list of attendance information to be rendered in the template. This includes details such as the time of each session, class name, group name, teacher name, and module.

Finally, it renders the `dashboard_students.html` template, passing along the attendance information and user information as context.

The dashboard of the student is shown in the following image:



For URLs configuration, we use:

```
from django.urls import path
from . import views

urlpatterns = [
     path('student_dashboard/',views.student_dashboard,name='
    student_dashboard'),
]
```

Listing 4.32: Configuring `student_dashboard` in `urls.py`

This sets up the URL pattern '/student_dashboard/' to be handled by the `student_dashboard` view function, allowing users to access the student dashboard functionality by navigating to the '/student_dashboard/' URL.

.5 Technical Team Dashboard:

In this section, we implement two dashboards for the technical team. One dashboard displays real-time streaming of temperature, humidity, and water level data, while the other shows real-time receiving and control of various parameters, including the water pump status and the power supply to different blocks in the university.

Real-Time Streaming on user-side :

The function that handles real-time streaming data is defined as follows:

```
def get_latest_data(request):
    # Query the latest data from the database
    latest_temperature_data = Temperature.objects.latest('timestamp')
    latest_humidity_data = Humidity.objects.latest('timestamp')
    latest_water_level_data = WaterLevel.objects.latest('timestamp')

    # Extract the timestamp and data values
    timestamp_temperature = latest_temperature_data.timestamp
    temperature = latest_temperature_data.temperature
    timestamp_humidity = latest_humidity_data.timestamp
    humidity = latest_humidity_data.humidity
    timestamp_water_level = latest_water_level_data.timestamp
```

```
    water_level = latest_water_level_data.water_level

    # Return the data as JSON
    return JsonResponse({
        'timestamp_temperature': timestamp_temperature,
        'temperature': temperature,
        'timestamp_humidity': timestamp_humidity,
        'humidity': humidity,
        'timestamp_water_level': timestamp_water_level,
        'water_level': water_level
    })
```

Listing 4.33: Handling Real-Time Data in `views.py`

Explanation:

This view function queries the latest recorded data for temperature, humidity, and water level from their respective database models. It then extracts the timestamp and data values. Finally, it returns this data as a JSON response.

For URLs configuration, we use:

```
from django.urls import path
from . import views

urlpatterns = [
    path('get_latest_data/', views.get_latest_data),
]
```

Listing 4.34: Configuring URLs for Real-Time Data in `urls.py`

This configuration maps the URL 'get_latest_data/' to the `get_latest_data` view function.

Rendering Dashboard Template:

We have another view function to render the HTML template for the technical team dashboard:

```
@login_required(login_url='login_student')
def technical_team_dashboard(request):
    # Check if the user is logged in as a technical team member
    if request.user.role != 'TECHNICAL_TEAM':
        return redirect('login')

    # Render the technical team dashboard template
    return render(request, 'AI/technical_team/dashboard_technical_team.html'
    )
```

Listing 4.35: Rendering Dashboard Template in `views.py`

Explanation:

This view function checks if the logged-in user has the role of a technical team member. If not, it redirects them to the login page. Then, it renders the HTML template for the technical team dashboard.

For URLs configuration, we use:

```
from django.urls import path
from . import views

urlpatterns = [
    path('technical_team_dashboard/', views.technical_team_dashboard, name='
    technical_team_dashboard'),
```

```
]
```

Listing 4.36: Configuring URLs for Dashboard Template in `urls.py`

This configuration maps the URL 'technical_team_dashboard/' to the `technical_team_dashboard` view function, allowing technical team members to access their dashboard.

show the real-time-straming:

know in order to show the real-time-stream in the dashboard we use charts[17] where we create div elements [18] with a unique id for each one. then we use the following javascript code to render the charts on those divs and update them periodically :

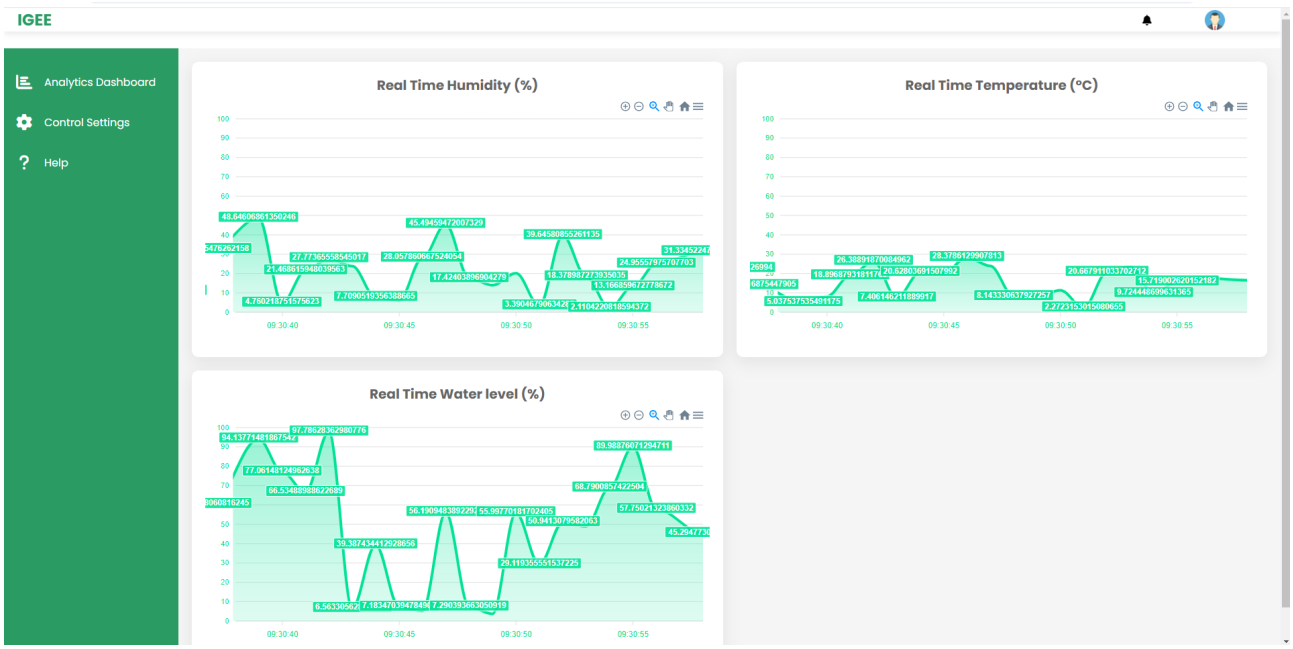
```
function fetchLatestData() {
  fetch("/get_latest_data/")
    .then((response) => response.json())
    .then((data) => {
      let x1 = new Date(data.timestamp_tempurature).getTime(),
        y1 = data.tempurature;
      let x2 = new Date(data.timestamp_himidity).getTime(),
        y2 = data.himidity;
      let x3 = new Date(data.timestamp_waterlevel).getTime(),
        y3 = data.waterlevel;
      chart.appendData([
        {
          data: [
            {
              x: x1,
              y: y1,
            },
          ],
        },
      ]);
      // Assuming you have another chart called chart1
      chart1.appendData([
        {
          data: [
            {
              x: x2,
              y: y2,
            },
          ],
        },
      ]);
      chart2.appendData([
        {
          data: [
            {
              x: x3,
              y: y3,
            },
          ],
        },
      ]);
    })
    .catch((error) => console.error("Error:", error));
}

// Update the chart with real-time data
```

```
47 window.setInterval(fetchLatestVoltageData, 1000);
```

Listing 4.37: javascript code for rendering charts

1. The `fetchLatestData()` function is defined, responsible for fetching the latest data from the server and updating the charts.
2. Inside this function, a `fetch()` request [19] is made to the "`/get_latest_data/`" endpoint to retrieve the latest data from the server.
3. The `.then()` method is used to handle the resolved promise (successful response) and parse the response body as JSON.
4. In the second `.then()` block, the extracted data is assigned to variables `x1`, `y1`, `x2`, `y2`, `x3`, and `y3`. These variables represent the timestamp and corresponding values for temperature, humidity, and water level.
5. The `appendData()` method is used to append the new data points to the respective charts (`chart`, `chart1`, `chart2`).
6. The `setInterval()` function is called to repeatedly execute the `fetchLatestData()` function at intervals of 1000 milliseconds (1 second), ensuring that the charts are updated with real-time data periodically.

this dashboard is shown on the following image :



Real-Time Receiving and Controlling on User-Side

In this section, we will implement real-time receiving and controlling of several parameters by the technical team. Before we proceed, let's understand some important concepts.

In traditional websites, when you submit data, the page often refreshes. This happens because the page content is generated or filled by the server. When you submit a form, the server processes your request and returns a new page or the same page with updated information, causing a page refresh. Similarly, when receiving new data from the server, the data will not be shown until you refresh the page because the server needs to generate the updated content.

This behavior is not suitable for real-time data receiving and monitoring. To address this, we will use a JavaScript framework called React [20]. React runs on the client side and controls

the state of elements on the page. This means that when the user needs to send or receive data, React (acting as a client-side server) handles the data transfer to and from the server in a continuous manner without causing the page to refresh.

React allows for dynamic updates to the user interface without the need for page reloads, making it ideal for real-time applications. In this project, we will leverage React to ensure that data can be sent and received in real time, providing a smooth and responsive user experience.

.6 Creating API Endpoints with Django RESTful API

First, let's complete the server functionality in order to interact with the user interface part. In this section, we are going to use the Django RESTful API concept [21], which is designed to handle different requests from the React user interface. The concept of an API is similar to functions in `views.py`, but it simplifies things significantly.

We will create API endpoints that receive two kinds of requests:

- `GET` request for retrieving the database information received from the Raspberry Pi.
- `POST` request for registering new parameter configurations in the database to be published then to the Raspberry Pi.

The API is implemented in `views.py` with the corresponding code:

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from data_collect.models import ControlSettingsRecive
from .serializers import ControlSettingsPublishSerializer,
    ControlSettingsReciveSerializer

class ControlSettingsView(APIView):
    def get(self, request):
        try:
            control_settings = ControlSettingsRecive.objects.first()
            serializer = ControlSettingsReciveSerializer(control_settings)
            return Response(serializer.data)
        except ControlSettingsRecive.DoesNotExist:
            return Response({"error": "ControlSettingsRecive does not exist"
    }, status=status.HTTP_404_NOT_FOUND)

    def post(self, request):
        serializer = ControlSettingsPublishSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.
    HTTP_400_BAD_REQUEST)
```

Listing 4.38: Defining API views in `views.py`

Explanation:

- `ControlSettingsView`: This class handles API requests related to control settings, extending `APIView` from Django REST framework.
- `get(self, request)`: This method handles `GET` requests. It attempts to retrieve the first `ControlSettingsRecive` object from the database. If found, it serializes the object and returns the serialized data as a JSON response. If the object does not exist, it returns a 404 error response.

- `post(self, request)`: This method handles `POST` requests. It deserializes the incoming data using `ControlSettingsPublishSerializer`. If the data is valid, it saves the new settings and returns the serialized data with a 201 status code. If the data is invalid, it returns the errors with a 400 status code.

The `ControlSettingsPublishSerializer` and `ControlSettingsReciveSerializer` are defined in `serializers.py` with the following code:

```
from rest_framework import serializers
from data_collect.models import ControlSettingsPublish,
   ControlSettingsRecive

class ControlSettingsPublishSerializer(serializers.ModelSerializer):
    class Meta:
        model = ControlSettingsPublish
        fields = [
            'water_pump_status',
            'block1_status',
            'block2_status',
            'block3_status',
        ]

class ControlSettingsReciveSerializer(serializers.ModelSerializer):
    class Meta:
        model = ControlSettingsRecive
        fields = [
            'water_pump_status',
            'block1_status',
            'block2_status',
            'block3_status',
            'fire_detection',
            'gas_detection',
        ]
```

Listing 4.39: Defining serializers in `serializers.py`

Explanation:

- `ControlSettingsPublishSerializer`: This serializer is used for serializing and deserializing data related to publishing control settings. It includes fields such as `water_pump_status`, `block1_status`, `block2_status`, and `block3_status`.
- `ControlSettingsReciveSerializer`: This serializer is used for serializing and deserializing data related to receiving control settings. It includes additional fields like `fire_detection` and `gas_detection`, along with the fields present in `ControlSettingsPublishSerializer`.

By implementing these serializers and views, we establish a RESTful API that allows the front-end (React) to interact with the back-end (Django) for real-time data handling and control functionalities.

Implementing Front end part with react :

In this section, we prepare the server-side setup to integrate with a React front-end application. We begin by rendering an empty template on the server, which will serve as the entry point for our React application. Inside this template, we create a div element with the id of "APP". This div will act as a placeholder that will be dynamically filled with the React front-end content once it's loaded.

To achieve this, we define a view function as follows:

```python
@login_required(login_url='login_student')
def control_settings_react(request):
    if request.user.role != 'TECHNICAL_TEAM':
        return redirect('login_student')

    return render(request, 'AI/technical_team/control_settings_react.html')
```

Listing 4.40: Defining the view function for rendering the empty template

This view function ensures that only users with the role of "TECHNICAL_TEAM" can access the page. It renders the specified template, `control_settings_react.html`, which initially contains only the empty *div* element with the id "APP".

Next, we configure the URL pattern to map to this view function:

```python
from django.urls import path
from . import views

urlpatterns = [
    path('control_settings_react/', views.control_settings_react, name='
    control_settings_react'),
]
```

Listing 4.41: URL configuration for the empty template view

This URL configuration ensures that requests to the specified URL, `/control_settings_react/`, are handled by the `control_settings_react` view function.

With this setup in place, we are now ready to develop and integrate our React front-end application, which will dynamically populate the empty template with interactive user interfaces and functionalities. the React code is as follows :

```javascript
import React, { useState, useEffect } from "react";
import Button from "@mui/material/Button";
import SendIcon from "@mui/icons-material/Send";
import { SwitchTextTrack } from "./Customswitch.js";
function getCookie(name) {
  const value = `; ${document.cookie}`;
  const parts = value.split(`; ${name}=`);

  if (parts.length === 2) {
    return parts.pop().split(";").shift();
  }
}
function App() {
  const [sendingData, setSendingData] = useState({
    water_pump_status: false,
    block1_status: false,
    block2_status: false,
    block3_status: false,
  });

  const [receivingData, setReceivingData] = useState({
    water_pump_status: false,
    block1_status: false,
    block2_status: false,
    block3_status: false,
    fire_detection: false,
    gas_detection: false,
  });

```

```
30  useEffect(() => {
31    // Fetch initial values from ControlSettings1
32    const fetchInitialValues = async () => {
33      try {
34        const response = await fetch("/api/control-settings/");
35        const data = await response.json();
36        setReceivingData(data);
37        setSendingData(data);
38      } catch (error) {
39        console.error("Error fetching initial values:", error);
40      }
41    };
42    fetchInitialValues();
43
44    // Polling to fetch updated values from ControlSettings1 every 2 seconds
45    setInterval(async () => {
46      try {
47        const response = await fetch("/api/control-settings/");
48        const data = await response.json();
49        setReceivingData(data);
50      } catch (error) {
51        console.error("Error fetching updated values:", error);
52      }
53    }, 2000);
54  }, []);
55
56  const handleSendData = async () => {
57    try {
58      const csrftoken = getCookie("csrftoken");
59      console.log(csrftoken);
60      await fetch("/api/control-settings/", {
61        method: "POST",
62        headers: {
63          "Content-Type": "application/json",
64          "X-CSRFToken": csrftoken,
65        },
66        body: JSON.stringify(sendingData),
67      });
68      console.log("Data sent successfully");
69    } catch (error) {
70      console.error("Error sending data:", error);
71    }
72  };
73
74  const handleToggleStatus = (key) => {
75    setSendingData((prevData) => ({
76      ...prevData,
77      [key]: !prevData[key],
78    }));
79  };
80
81  return (
82    <div className="container">
83      <h1>Control Settings</h1>
84      <div className="chart">
85        <h2>Receiving Data</h2>
86        <div className="receiving-data  control-settings">
87          <p className={receivingData.fire_detection ? "on" : "off"}>
88            <p>Fire Detection</p>
```

```
            <SwitchTextTrack checked={receivingData.fire_detection} />
          </p>
          <p className={receivingData.gas_detection ? "on" : "off"}>
            <p>Gas Detection</p>
            <SwitchTextTrack checked={receivingData.gas_detection} />
          </p>
          <p className={receivingData.water_pump_status ? "on" : "off"}>
            <p>Water Pump</p>
            <SwitchTextTrack checked={receivingData.water_pump_status} />
          </p>
          <p className={receivingData.block1_status ? "on" : "off"}>
            <p>Block 1</p>
            <SwitchTextTrack checked={receivingData.block1_status} />
          </p>
          <p className={receivingData.block2_status ? "on" : "off"}>
            <p>Block 2</p>
            <SwitchTextTrack checked={receivingData.block2_status} />
          </p>
          <p className={receivingData.block3_status ? "on" : "off"}>
            <p>Block 3</p>
            <SwitchTextTrack checked={receivingData.block3_status} />
          </p>
        </div>
      </div>
      <div className="sending-data chart">
        <h2>Sending Data</h2>
        <div className="control-settings ">
          <button
            className={sendingData.water_pump_status ? "on" : "off"}
            onClick={() => handleToggleStatus("water_pump_status")}
          >
            <p>Water Pump</p>
            <SwitchTextTrack checked={sendingData.water_pump_status} />
          </button>

          <button
            className={sendingData.block1_status ? "on" : "off"}
            onClick={() => handleToggleStatus("block1_status")}
          >
            <p>Block 1</p>
            <SwitchTextTrack checked={sendingData.block1_status} />
          </button>

          <button
            className={sendingData.block2_status ? "on" : "off"}
            onClick={() => handleToggleStatus("block2_status")}
          >
            <p>Block 2</p>
            <SwitchTextTrack checked={sendingData.block2_status} />
          </button>

          <button
            className={sendingData.block3_status ? "on" : "off"}
            onClick={() => handleToggleStatus("block3_status")}
          >
            <p>Block 3 </p>
            <SwitchTextTrack checked={sendingData.block3_status} />
          </button>
        </div>
```

```
        <div className="Send">
          <span className="Send-container">
            <Button
              variant="contained"
              endIcon={<SendIcon />}
              onClick={handleSendData}
              sx={{
                backgroundColor: "#299b63",
                "&:hover": {
                  backgroundColor: "#208053", // Change the hover color if needed
                },
              }}
            >
              Send
            </Button>
          </span>
        </div>
      </div>
    </div>
  );
}

export default App;
```

Listing 4.42: URL configuration for the empty template view

The provided code fetches the API for new data and registers control settings if the user submits changes. Control settings, represented by switches, toggle when data changes. The code fetches the API every 2 seconds, creating a real-time application where data changes occur without page refresh.

In the JavaScript code, the ‘App‘ component manages sending and receiving control settings data. It uses the ‘useState‘ and ‘useEffect‘ hooks to manage state and perform side effects. The ‘fetchInitialValues‘ function retrieves initial control settings data from the API and sets the state accordingly. Then, a polling mechanism fetches updated data every 2 seconds to ensure real-time updates.

The ‘handleSendData‘ function sends the updated control settings data to the API when the user clicks the ”Send” button. The ‘handleToggleStatus‘ function toggles the status of control settings when the user interacts with the switches.

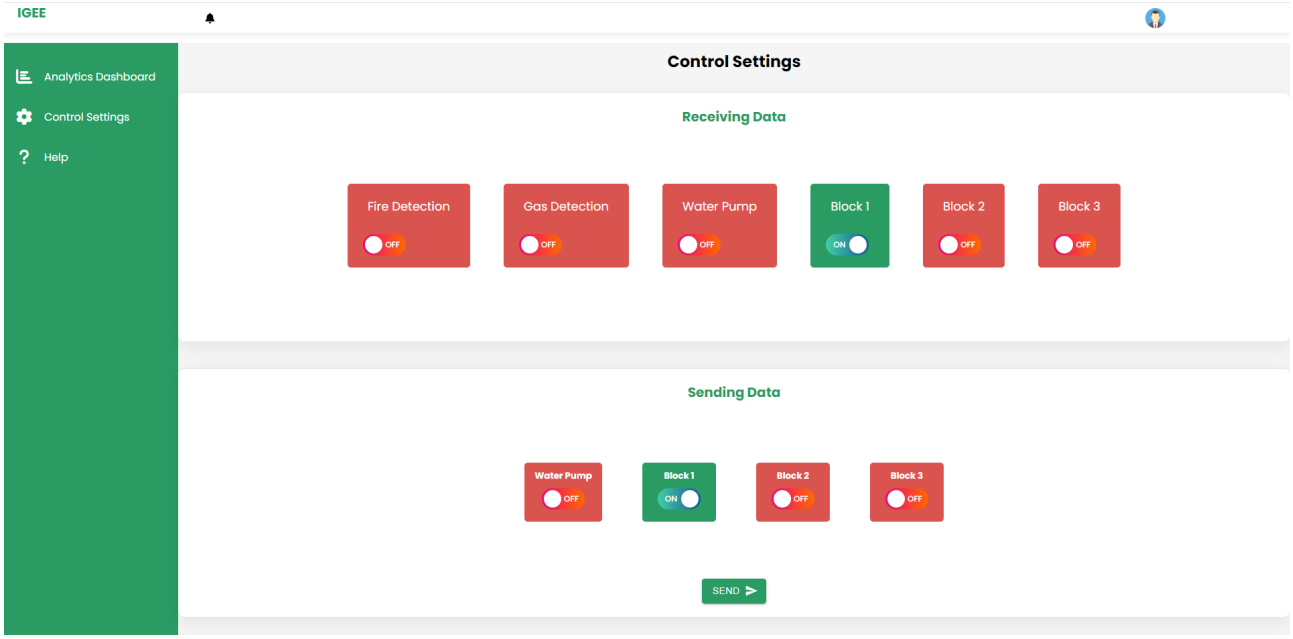
The React component renders the control settings interface, displaying both receiving and sending data sections. The receiving data section shows the current status of various control settings, while the sending data section allows the user to toggle control settings and send updated data to the API.

```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App.js";

const root = ReactDOM.createRoot(document.getElementById("app"));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

Listing 4.43: Defining the React Component for rendering the App component

Additionally, the entire component is rendered inside a div element with the id "App".

This is how the final result of the real-time reciving and controlling Dashboard :



IOT Integration for Real-time Updates

In this section, we will modify the structure of Django to make it suitable for IoT integration. We will add an additional file that will handle all IoT tasks called `task.py` .

We will configure this `task.py` file to run with the server. Let's explain each part in the `task.py` file.

First, we import all the packages needed and the database models:

```
from copy import deepcopy
import paho.mqtt.client as mqtt
import json
import time
from django.utils import timezone
from user_interface.models import SetSession, Student, Teacher, Attendance
from data_collect.models import *
broker_address='127.0.0.1'
port=1883
```

Listing 4.44: Import necessary packages and models in `task.py`

The MQTT broker (server) is accepting messages at the host address `127.0.0.1` and port `1883`. This setup ensures that our IoT devices can communicate with the server using the MQTT protocol.

.1 Automatic Session Scheduling at the Appropriate Time

This step is done with the following function:

```
def send_session_data(broker_address=broker_address, port=port):
    all_students = ''
    client = mqtt.Client(client_id="45767879")

    while True:
```

```
        current_time = timezone.now().strftime("%Y-%m-%d %H:%M:00+00:00")  # Format current time to match SetSession's time format
        print(current_time)
        try:
            session = SetSession.objects.get(time=current_time)
            students = Student.objects.filter(group_number=session.group)
            teacher = session.teacher
            class_name = session.class_name
            print('performed')
            print(students)
        except SetSession.DoesNotExist:
            # No active session
            print("error")
            time.sleep(5)  # Sleep for 5 seconds and check again
            continue

        for student in students:
            all_students += student.id_number + ' '

        # Create a JSON object with session data
        data = {
            "time": current_time,
            "students": all_students,
            "teacher": teacher.id_number,
            "class_name": class_name,
        }
        print(data)
        send = json.dumps(data)
        client.connect(broker_address, port)
        # Send the JSON session data to Raspberry Pi via MQTT
        for i in range(2):
            client.publish("session_data0", send)
        time.sleep(56)  # Sleep for 56 seconds before checking again
        client.disconnect()
        all_students = ''
```

Listing 4.45: Function to send session data in `task.py`

In this code:

- The `send_session_data` function initializes an MQTT client and enters an infinite loop to periodically check for active sessions.
- `current_time` is formatted to match the `SetSession` model's time format.
- The function tries to retrieve a session that matches the current time from the `SetSession` model.
- If a session is found, it retrieves the associated students, teacher, and class name.
- It then creates a JSON object containing the session data.
- The MQTT client connects to the broker, and the session data is published to the topic `"session_data0"`.
- The function waits for 56 seconds before checking again.
- If the session does not exist, this will trigger an error. The `try` and `except` blocks handle this by waiting for 5 seconds before checking again, ensuring that the function continuously check for active sessions.

Note: In order to send the data via MQTT, it needs to be in the format of strings. For that purpose, we use:

```
for student in students:
    all_students += student.id_number + ' '
```

which makes a string of space-separated IDs.

.2 Automatic Receiving and Registering Attendance List in the Database

This step is achieved with the following function:

```
def receive_and_process_session_data(broker_address=broker_address, port=port):
    def on_message(client, userdata, message):
        try:
            received_data = json.loads(message.payload.decode())
            teacher_id = received_data["teacher"]
            student_ids = received_data["students"]
            session_time = received_data["time"]

            if session_time:
                session = SetSession.objects.get(time=session_time)
                print('find_time')
            if student_ids:
                student_list = student_ids.lstrip().rstrip().split(' ')
                print(student_list)

                # Retrieve the teacher and students
                students = Student.objects.filter(id_number__in=student_list)

                # Process the received data and create attendance records
                for student in students:
                    Attendance.objects.create(session=session, student=student)
                    print('done')

            if teacher_id:
                teacher = Teacher.objects.get(id_number=teacher_id)
                Attendance.objects.create(session=session, teacher=teacher)
                print('done')
        except:
            print('error')

    client = mqtt.Client(client_id='1354614613')
    client.on_message = on_message
    client.connect(broker_address, port)
    client.subscribe("session_data_recive")
    client.loop_forever()
```

Listing 4.46: Function to Record attandance data in task.py

Explanation:

This function facilitates the automatic reception and registration of attendance lists in the database. Here's how it works:

- The function `receive_and_process_session_data` is defined with parameters `broker_address` and `port`, which default to `"127.0.0.1"` and `1883`, respectively.

- Inside this function, an MQTT client object is created with the client ID set to '1354614613'.
- The `on_message` function is called whenever a message is received on the subscribed topic. It attempts to decode the received message payload as JSON and extracts the teacher ID, student IDs, and session time from the received data.
- If session time data is present, the function queries the `SetSession` model to retrieve the corresponding session based on the received time.
- If student IDs are available, they are processed to create attendance records for each student. The student IDs are stripped of leading and trailing whitespaces and split into a list. The function then queries the `Student` model to retrieve student objects based on their IDs and creates attendance records for each student associated with the retrieved session.
- If a teacher ID is present, the corresponding teacher object is retrieved, and an attendance record is created for the teacher associated with the session.
- The MQTT client connects to the specified broker address and port. It subscribes to the topic "session_data_recive" to listen for incoming session data messages.
- Exception handling is implemented to catch any errors that may occur during message processing. If an error occurs, an error message is printed to the console.
- The reason we check if the `student_ids` or the `teacher_id` is present is that the Raspberry Pi, when it finds the teacher absent, will send its ID plus the student IDs as null. Otherwise, when the teacher is present, it will send the teacher ID as null and the student IDs normally.
- In case both teacher and students are present, the Raspberry Pi will send both teacher ID and student IDs as null.

.3 Automatic Checking of Changes in ControlSettings and Sending Changes if Found

This step is achieved with the following function:

```
def send_control_settings_data(broker_address="127.0.0.1", port=1883):
    print(f"Connecting to broker at {broker_address}:{port}...")
    client = mqtt.Client(client_id="control_settings_sender")

    try:
        last_known_settings = {}

        while True:
            try:
                control_settings = ControlSettingsPublish.objects.last()
                current_settings = {
                    'water_pump_status': control_settings.water_pump_status,
                    'block1_status': control_settings.block1_status,
                    'block2_status': control_settings.block2_status,
                    'block3_status': control_settings.block3_status,
                }

                if current_settings != last_known_settings:
                    client.connect(broker_address, port)
                    print(f"Connected to broker at {broker_address}:{port}")
                    last_known_settings = deepcopy(current_settings)
                    send = json.dumps(current_settings)

                    client.publish("control_settings_publish", send)
```

```
                    time.sleep(2)
                    print(f"Sent control settings: {current_settings}")
                    client.disconnect()

            except Exception as e:
                print(f"Error sending control settings data: {str(e)}")
                time.sleep(5)
    except Exception as e:
        print(f"Error connecting to broker: {str(e)}")
```

Listing 4.47: Import necessary packages and models in `task.py`

Explanation:

This function facilitates the automatic checking of changes in ControlSettings and sending changes if found. Here's how it works:

- The function `send_control_settings_data` is defined with parameters `broker_address` and `port`, which default to "127.0.0.1" and 1883, respectively.
- Inside this function, an MQTT client object is created with the client ID set to "control_settings_sender".
- A loop is initiated to continuously monitor changes in ControlSettings.
- The function attempts to retrieve the latest ControlSettings from the `ControlSettingsPublish` model.
- The current ControlSettings are compared with the last known settings. If changes are detected, the MQTT client connects to the broker and publishes the updated ControlSettings to the topic "control_settings_publish".
- The function sleeps for 2 seconds after sending the control settings to avoid rapid consecutive transmissions.
- Exception handling is implemented to catch any errors that may occur during the process. If an error occurs while sending control settings data or connecting to the broker, an error message is printed to the console.

.4 Automatic Receiving, Processing, and Registering ControlSettings in the Database

This step is achieved with the following function:

```
def receive_and_process_control_settings(broker_address=broker_address, port=port):
    control_settings_data = {}

    # Retrieve the existing ControlSettingsRecive object from the database
    control_settings = ControlSettingsRecive.objects.first()

    # Define the on_message callback function
    def on_message(client, userdata, message):
        try:
            # Decode the received message payload and load it as JSON data
            received_data = json.loads(message.payload.decode())

            # Extract control settings data from the received JSON data
            control_settings_data['block1_status'] = received_data.get("block1_status")
            control_settings_data['block2_status'] = received_data.get("block2_status")
```

```
            control_settings_data['block3_status'] = received_data.get("block3_status")
            control_settings_data['fire_detection'] = received_data.get("fire_detection")
            control_settings_data['gas_detection'] = received_data.get("gas_detection")
            control_settings_data['water_pump_status'] = received_data.get("water_pump_status")

            # Check if the ControlSettingsRecive object exists in the database
            if control_settings:
                # Update the ControlSettingsRecive object with the received data
                control_settings.block1_status = control_settings_data.get('block1_status')
                control_settings.block2_status = control_settings_data.get('block2_status')
                control_settings.block3_status = control_settings_data.get('block3_status')
                control_settings.fire_detection = control_settings_data.get('fire_detection')
                control_settings.gas_detection = control_settings_data.get('gas_detection')
                control_settings.water_pump_status = control_settings_data.get('water_pump_status')

                # Save the changes to the database
                control_settings.save()

                print("Updated control settings:", control_settings_data)

        except Exception as e:
            print(f"Error processing control settings data: {str(e)}")

    # Create an MQTT client and configure the on_message callback function
    client = mqtt.Client(client_id='control_settings_receiver')
    client.on_message = on_message

    # Connect to the MQTT broker and subscribe to the control_settings_recive topic
    client.connect(broker_address, port)
    client.subscribe("control_settings_recive")

    # Start the MQTT client loop to listen for incoming messages indefinitely
    client.loop_forever()
```

Listing 4.48: Import necessary packages and models in `task.py`

Explanation:

This function enables the automatic receiving, processing, and registering of ControlSettings data in the database. Here's how it works:

- The function `receive_and_process_control_settings` is defined with parameters `broker_address` and `port`, which default to the values of `"127.0.0.1"` and `1883` defined elsewhere.
- It initializes an empty dictionary `control_settings_data` to store the received control settings data.

- It retrieves the existing `ControlSettingsRecive` object from the database, if available.
- Here, in this case, we are dealing only with the first row of `ControlSettingsRecive`, which represents the current settings in the university.
- Inside the function, a callback function `on_message` is defined to handle incoming MQTT messages. When a message is received, it decodes the payload and extracts control settings data.
- It checks if the `ControlSettingsRecive` object exists in the database. If it does, it updates the object with the received data and saves the changes to the database.
- Exception handling is implemented to catch any errors that may occur during the processing of control settings data.
- An MQTT client is created with the client ID set to `'control_settings_receiver'`. The `on_message` callback function is configured for the client.
- The client connects to the MQTT broker, subscribes to the `control_settings_recive` topic, and starts an infinite loop to listen for incoming messages.

The last step is to integrate task.py in wsgi.py and start the server as the following code show :

```
from django.core.wsgi import get_wsgi_application
from django.core.wsgi import get_wsgi_application
import os
import json
import paho.mqtt.client as mqtt
import threading
import time
from user_interface.models import *
from data_collect.models import *
from .task import *
broker_address='127.0.0.1'
port=1883
# Define the MQTT callback functions for 'order' and 'order1' topics

def on_connect(client, userdata, flags, rc):
    client.subscribe('data_topic')  # Subscribe to the 'order1' topic
    print("Subscribing to topic: tempurature_topic")

def on_message(client, userdata, msg):
    try:
        received_data = json.loads(msg.payload.decode())
        print(received_data)
        tempurature = received_data["temperature"]
        himidity = received_data["himidity"]
        waterLevel = received_data["waterLevel"]
        print('----------------------------------------pass')
        print(msg.topic+" "+str(msg.payload))
        tempurature = float(tempurature)
        himidity = float(himidity)
        waterLevel = float(waterLevel)
        Tempurature.objects.create(tempurature=tempurature)
        Himidity.objects.create(himidity=himidity)
        WaterLevel.objects.create(waterlevel=waterLevel)
        # Handle 'order1' message here
    except :
        print ('------------------------------------------------ error
   handeling data values ---------------------')
```

```
        pass
def subscribe(client, on_connect_callback):
    client.on_connect = on_connect_callback
    client.on_message = on_message

    client.connect(broker_address, port,keepalive=60)
    client.loop_forever()

# Initialize the MQTT clients
client = mqtt.Client(client_id='client_9')
client1 = mqtt.Client(client_id='client_100')
t = threading.Thread(target=subscribe, args=(client, on_connect))

t3=threading.Thread(target=receive_and_process_session_data)
t4=threading.Thread(target=send_session_data)
t5=threading.Thread(target=send_control_settings_data)
t6=threading.Thread(target=receive_and_process_control_settings)
t6.start()
t5.start()
t4.start()
t3.start()
t.start()

# print(timezone.now().strftime(f"%Y-%m-%d %H:%M:00+00:00 "))
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'iot.settings')
application = get_wsgi_application()
```

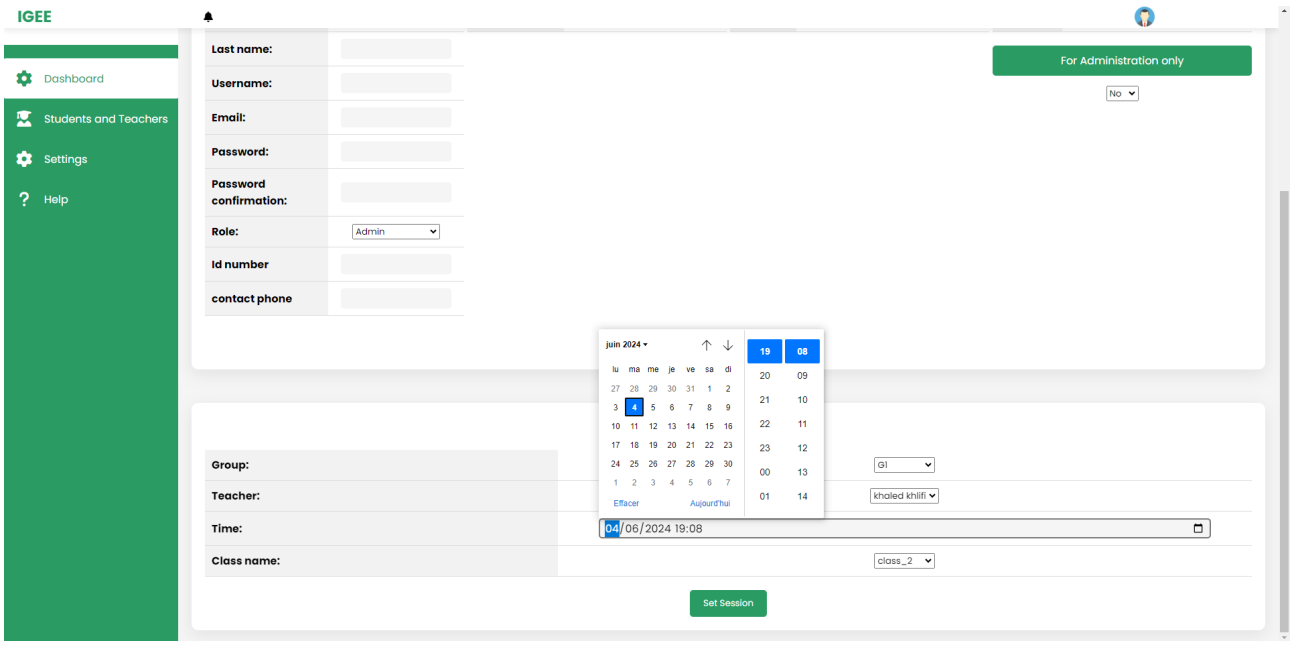
Listing 4.49: Integrate `task.py in wsgi.py`

The provided code runs the functions from the 'task.py' file within threads. Additionally, it creates MQTT clients to gather real-time humidity, temperature, and water level information and stores them in the database. These MQTT clients are also run within threads for asynchronous execution. Typically, the 'wsgi.py' file is utilized to trigger functions because the Django system executes this file each time the server runs.

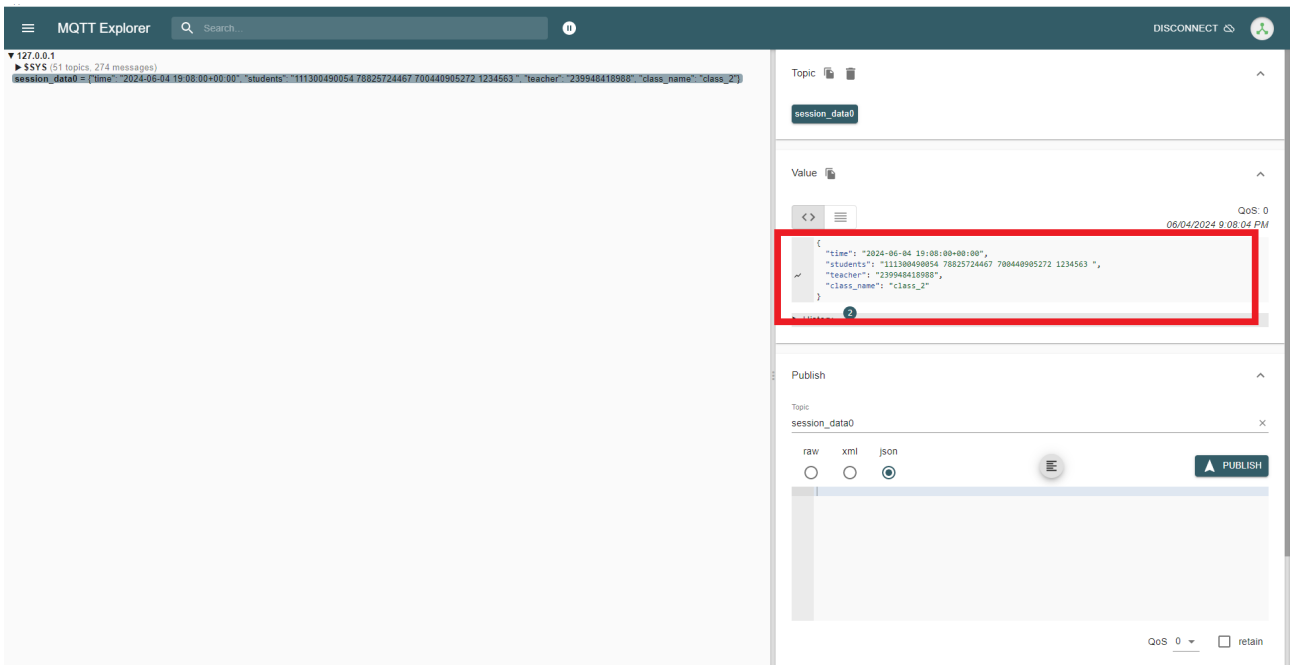
Simulating the IoT Side of the Project

.1 Session Scheduling and Attendance Monitoring

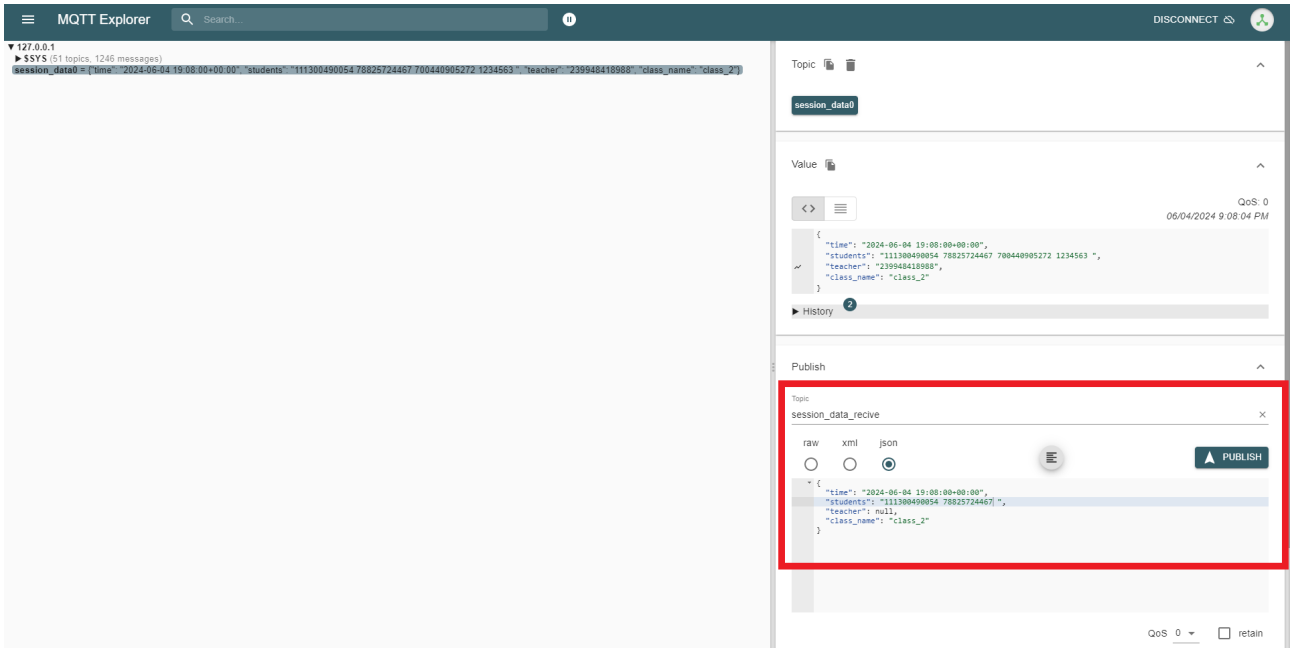
The administration can schedule sessions as illustrated below:



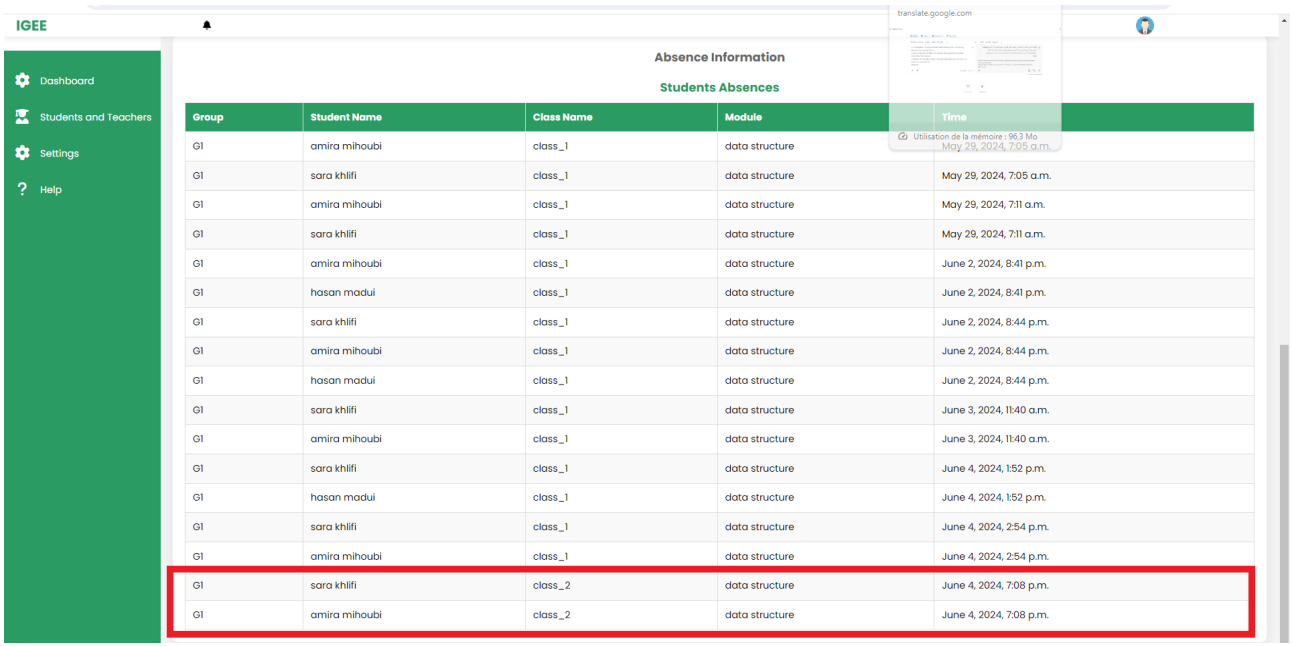
When the time of the session arrives, the server publishes its information to the Raspberry Pi. This session information arrives at the Raspberry Pi, as indicated by MQTT Explorer in the following image:



The Raspberry Pi then handles the session information, records the attendance, makes some changes to the received attendance information, and only submits the IDs of the absent students or teachers. The following image shows how the session information is changed, with two absences recorded and returned with the modified session information:



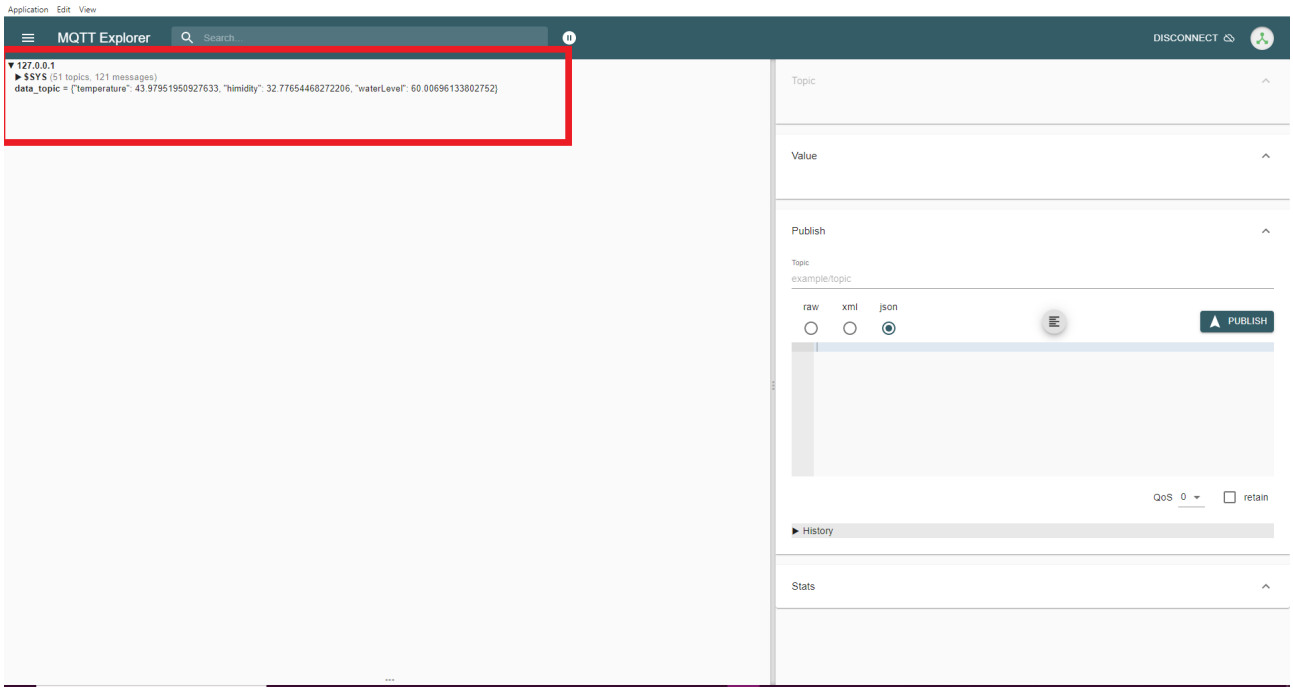
After this information is published, the absences are recorded in the database and shown in the attendance dashboard, as illustrated in the following image:



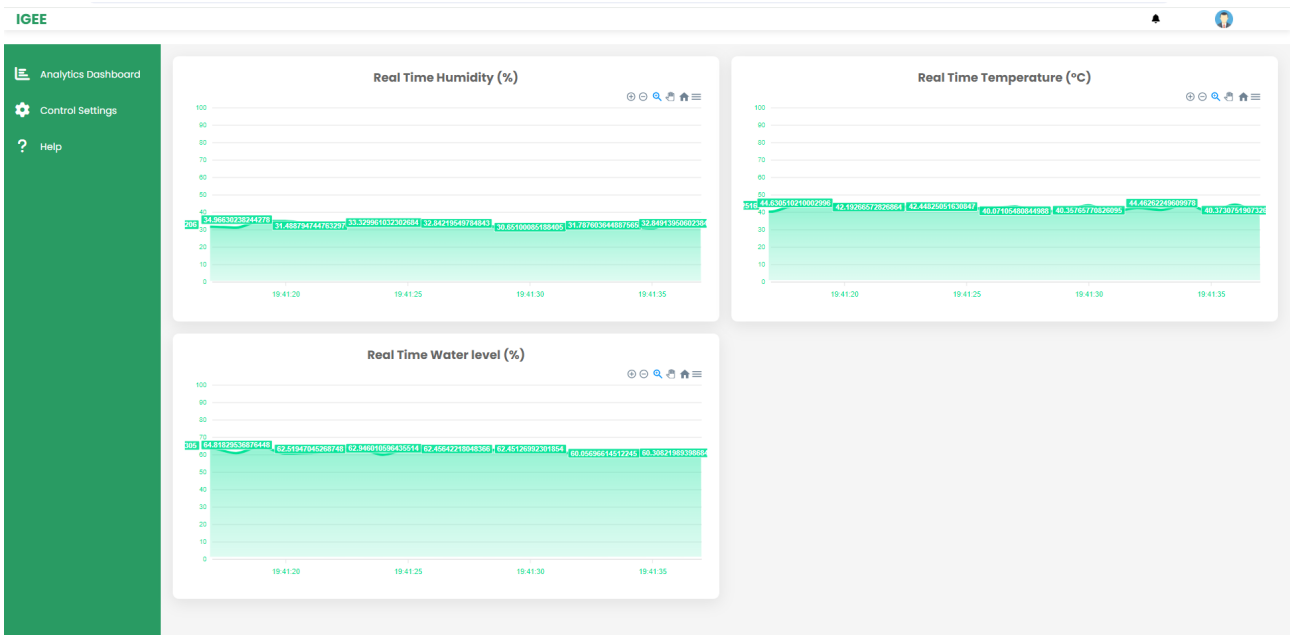
.2 University Environment Monitoring and Controlling

Humidity, Temperature, Water Level Monitoring

The Raspberry Pi is real-time recording those values and publishing them to the server. The server will receive those values and display them in the dashboard of the technical team. The following image shows how these data are received:



After receiving these data and recording them in the database, the server displays them in real-time on the dashboard as illustrated by the following image:

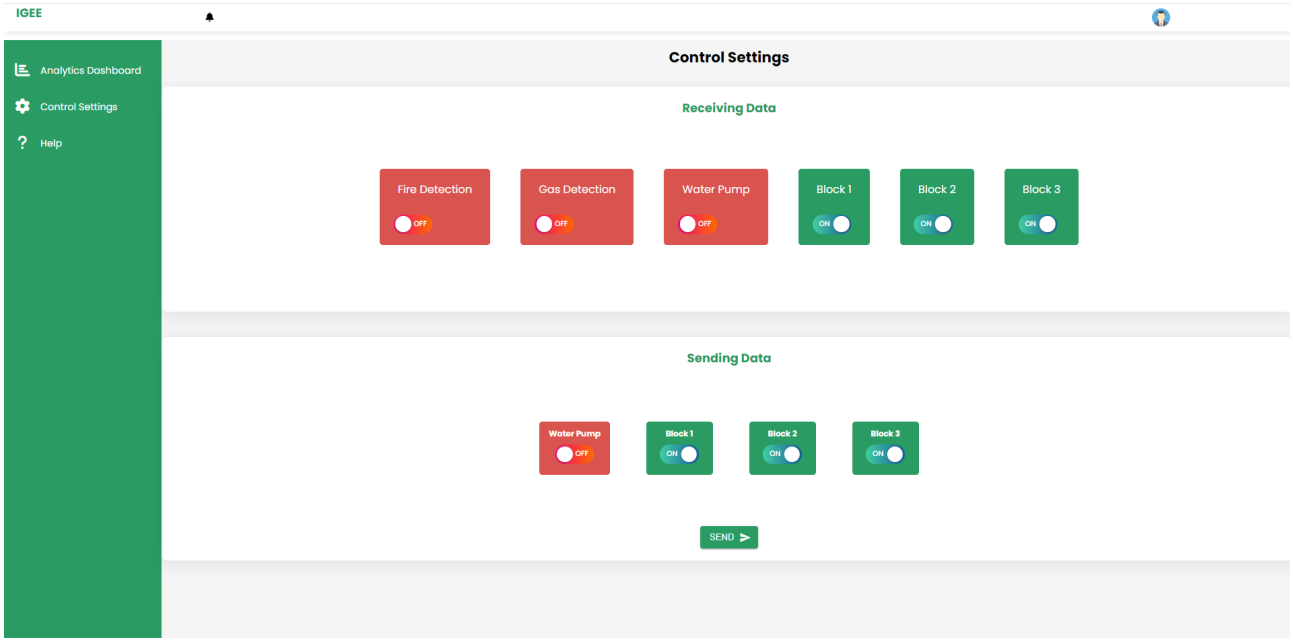


Real-time Monitoring and Controlling of Different Parameters

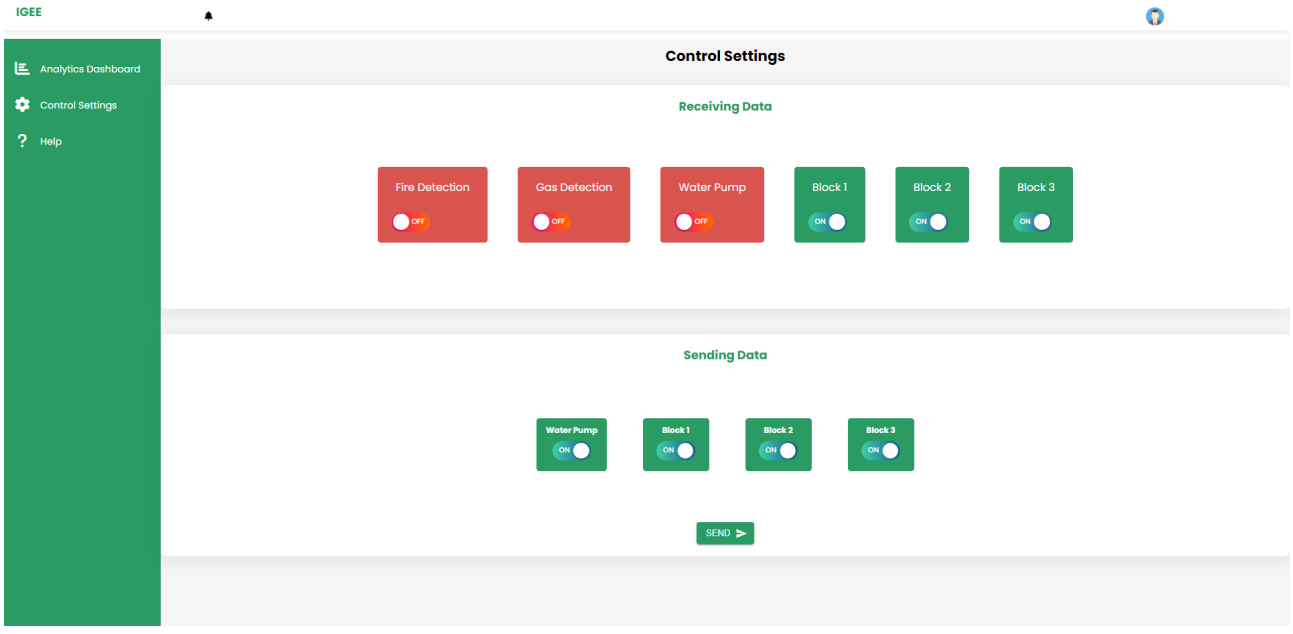
Typically, the technical team can see real-time records of different parameters like gas detection, fire detection, water pump status, power supply to different blocks [1,2,3] - typically teaching department, administration department, outside environment respectively. They can also control water pump status and power supply to different blocks [1,2,3].

Let's first start by making changes in the parameters and then register them to see what happens. The changes are shown in the following images:

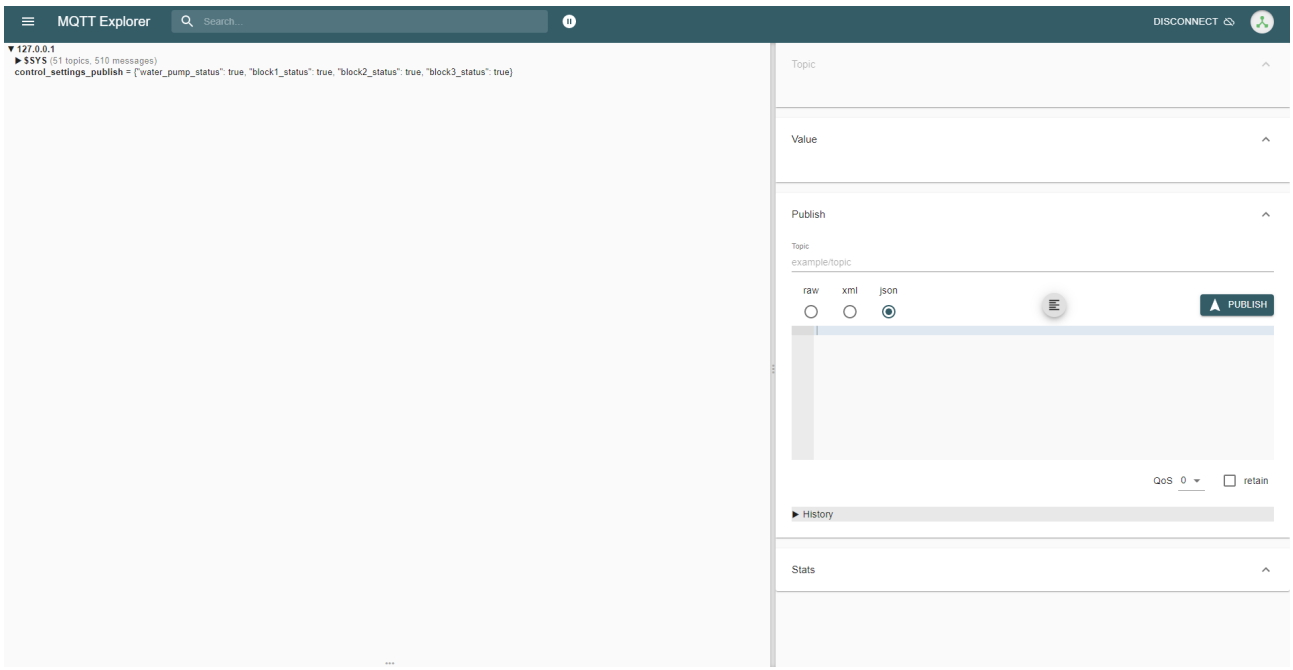
Before change:



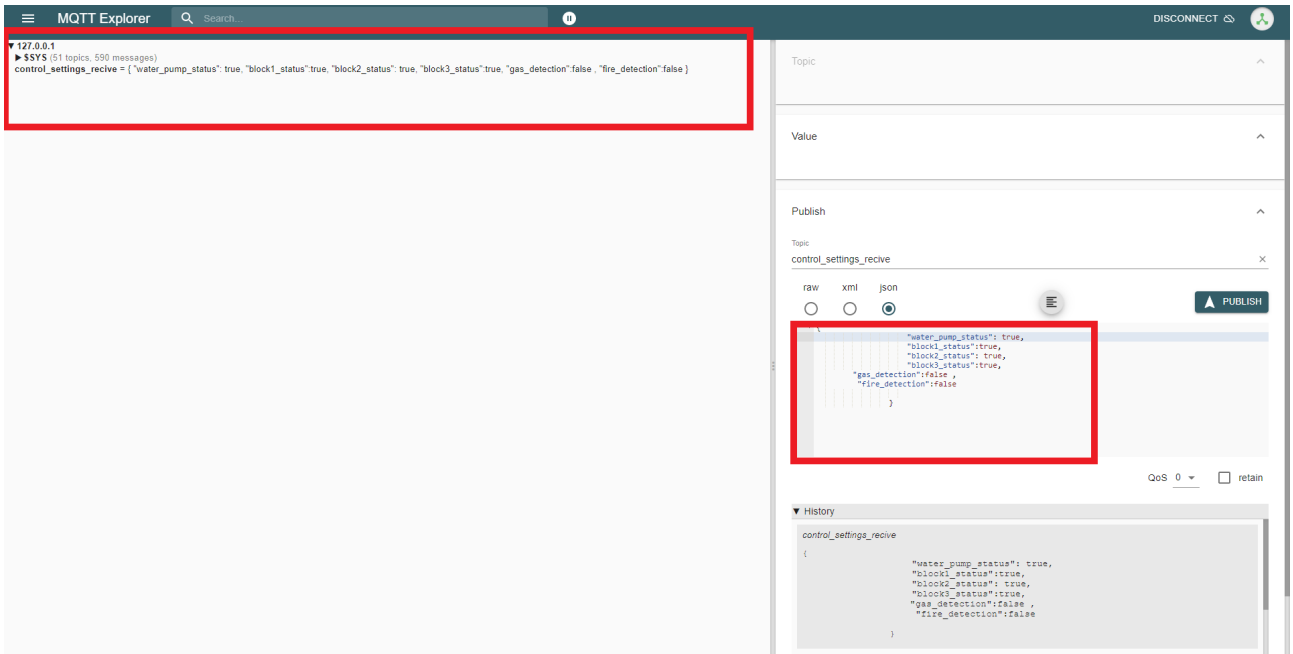
After change:



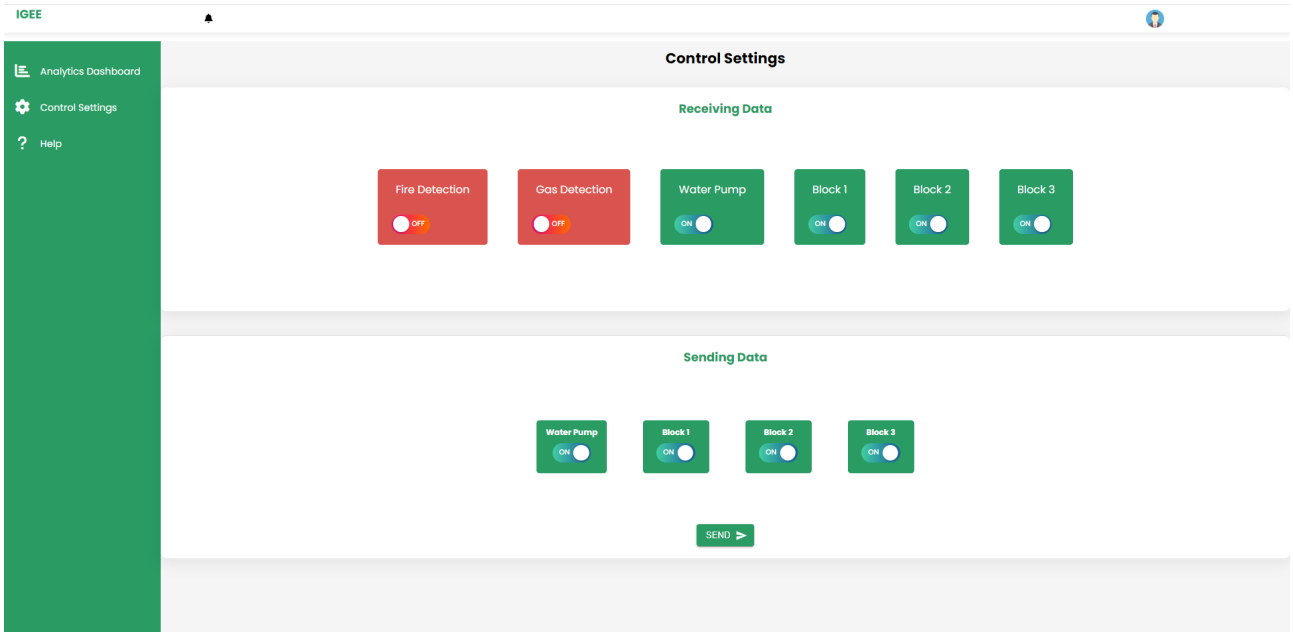
When the changes are saved, the server detects them and sends the following message to the Raspberry Pi as indicated by the following image:



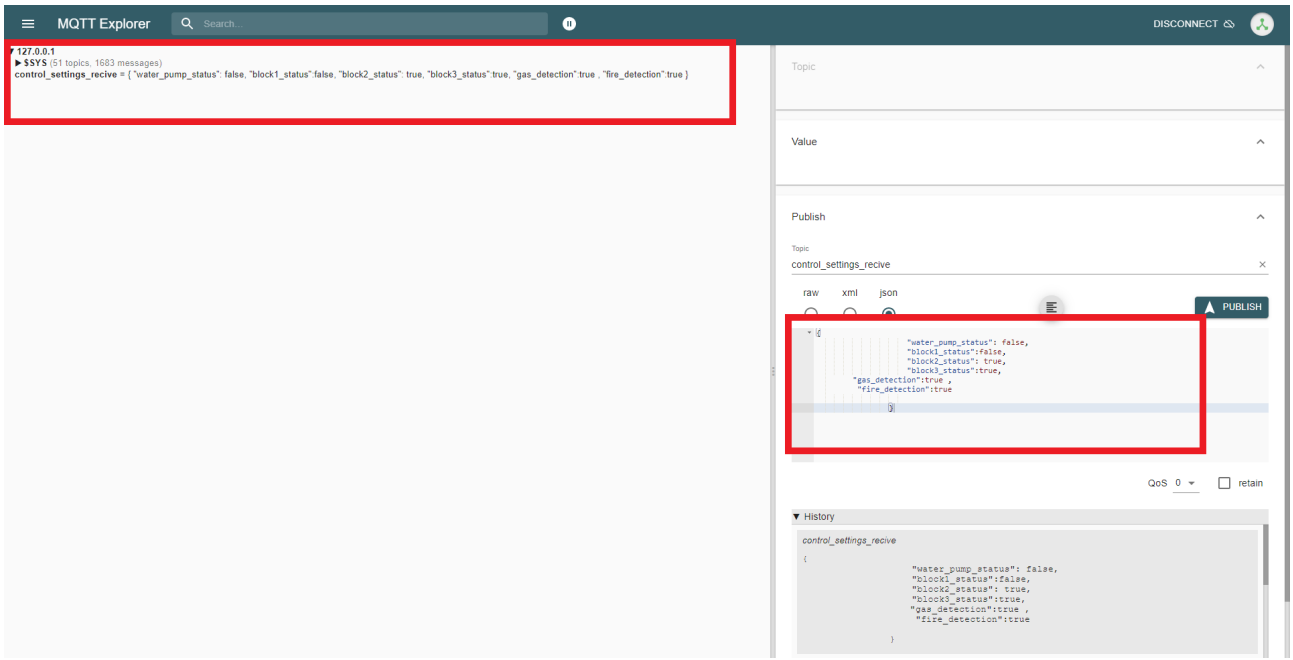
Then the Raspberry Pi will make those changes, and when it does, it will publish back a message to the server with the new university parameter status, as illustrated by the following image:



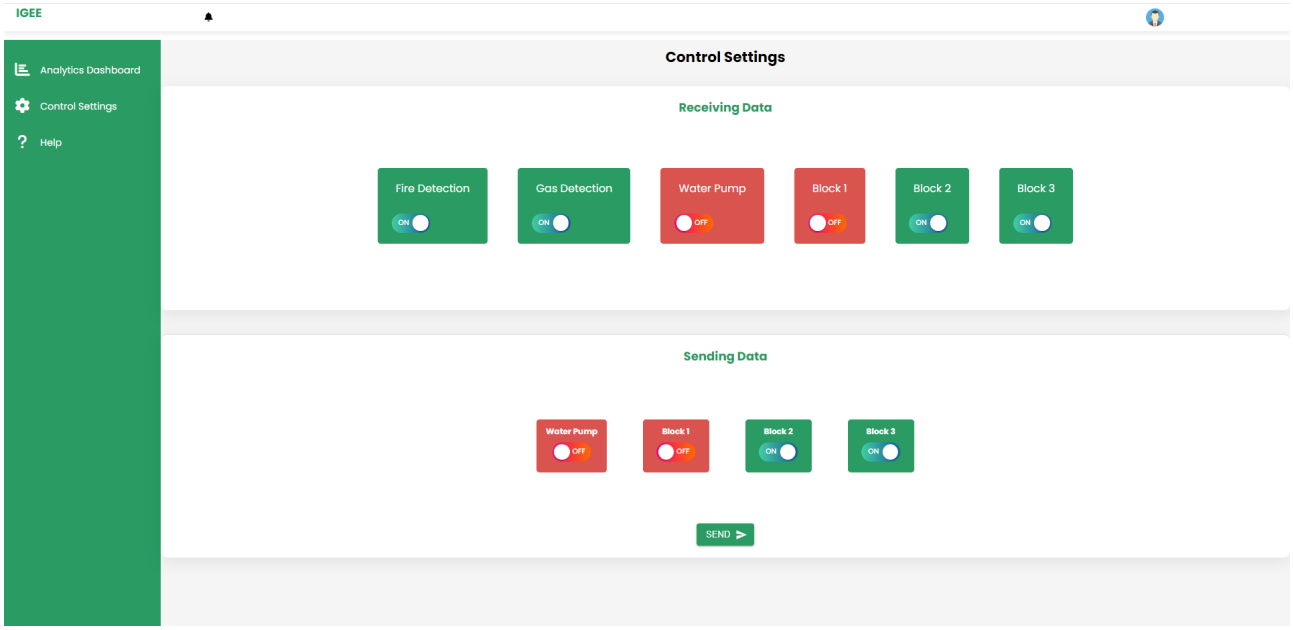
The server receives those changes, registers them in the database, and shows them on real-time streaming to the technical team dashboard, as the following image illustrates:



Also, if some settings changed on the Raspberry Pi side, it will detect those changes directly and publish them to the server, like in emergency situations (gas detection, fire detection). The power supply to the specified block will be cut off (in our case, block one). The Raspberry Pi will detect those changes in parameters and publish the change to the server, as illustrated by the following image:



Then the server receives these changes, registers them in the database, and then displays them in real-time to the technical team dashboard, as the following image illustrates:


IGEE
Analytics Dashboard
Control Settings
Help
Control Settings
Receiving Data
Fire Detection
ON
Gas Detection
ON
Water Pump
OFF
Block 1
OFF
Block 2
ON
Block 3
ON
Sending Data
Water Pump
OFF
Block 1
OFF
Block 2
ON
Block 3
ON
SEND

Chapter 5

Conclusion

Achievements

The integration of IoT technologies within the university setting has resulted in significant advancements in operational efficiency, security, and academic management. This project introduced several key components that together form a robust and comprehensive system, enhancing various aspects of the university environment.

Among the notable achievements are:

- **Face Recognition System**: Implemented a sophisticated face recognition system to enhance security by automating identity verification processes.
- **Smart Parking System**: Developed an intelligent parking system that optimizes space usage and provides real-time availability information, significantly improving parking efficiency.
- **Website Development and IoT Integration**: Created a dynamic website integrated with IoT devices, enabling real-time data handling and providing various dashboards for administration, teachers, students, and technical teams.
- **Attendance Monitoring and Session Scheduling**: Automated attendance monitoring and session scheduling, allowing both teachers and students to track attendance records easily and accurately.
- **Environmental Parameter Monitoring**: Enabled the technical team to monitor various environmental parameters of the university in real-time, enhancing the overall operational awareness and responsiveness.

The administration dashboard facilitates efficient session scheduling and attendance management, while the teachers' dashboard provides a comprehensive view of their schedules and student attendance. The students' dashboard offers visibility into their attendance records, promoting accountability. Furthermore, the technical team dashboard ensures continuous monitoring of environmental parameters, contributing to a safer and more controlled university environment.

By integrating these technologies, the project has significantly improved the security, academic management, and technical monitoring capabilities of the university, making the learning environment more efficient and secure.

The success of this project is underscored by its achievement of winning first place in the Nexus IoT competition organized by the IEEE Scientific Club, demonstrating its innovative approach and effective implementation.

Certificate

IEEE Student Branch University of Boumerdes

infodium

Nexus TECH COMPETITIONS

1

CERTIFICATE OF ACHIEVEMENT

This Certificate is awarded to

Smart University

in recognition of outstanding performance and achieving the First prize in the Nexus Tech Competitions orgranized by the IEEE Student Branch of Boumerdes.

DATE
27/11/2023

SIGNATURE

Université de Boumerdes
CLUB I.E.E.E
IGEE

Appendix A

*

Bibliography

[1] geeksforgeeks. "Introduction to internet of things (iot) – set 1." (2024), [Online]. Available: https://www.geeksforgeeks.org/introduction-to-internet-of-things-iot-set-1/.

[2] AWS. "Mqtt exploration." (2024), [Online]. Available: https://aws.amazon.com/what-is/mqtt/.

[3] ESYOCR. "Esyocr documentation." (2023), [Online]. Available: https://www.jaided.ai/easyocr/documentation/.

[4] A. Kathuria. "Yolo advanced explanation and implementation with pytorch." (2018), [Online]. Available: https://blog.paperspace.com/how-to-implement-a-yolo-object-detector-in-pytorch/.

[5] A. ibrahim. "A playlist guiding to a better understanding of advanced yolo algorithm." (2023), [Online]. Available: https://youtube.com/playlist?list=PLyhJeMedQd9TLcgIMzZFxHTGPpRiSD2Wi&si=r1PaR6nD4O2NZaPl.

[6] F. Jacob Solawetz. "What is yolov8? the ultimate guide." (2023), [Online]. Available: https://blog.roboflow.com/whats-new-in-yolov8/.

[7] ultralytics. "Ultralytics yolov8 repository." (2023), [Online]. Available: https://github.com/ultralytics/ultralytics.

[8] P. Kumar. "Siamese networks introduction and implementation." (2023), [Online]. Available: https://medium.com/@prabhattgs12345789/siamese-neural-network-enhancing-ai-capabilities-with-pairwise-comparisons-4f00e2dd8256.

[9] N. Hamdani, "Face detection and recognition using siamese neural network," 2023.

[10] A. Ng. "C4w4l03 siamese network." (2018), [Online]. Available: https://youtu.be/6jfw8MuKwpI?si=Du9x_Y9Y9uDRc-gX.

[11] A. Ng. "C4w4l04 triplet loss." (2018), [Online]. Available: https://medium.com/@prabhattgs12345789/siamese-neural-network-enhancing-ai-capabilities-with-pairwise-comparisons-4f00e2dd8256.

[12] J. López. "Yolov8 for face detection." (2023), [Online]. Available: https://github.com/ageitgey/face_recognition?tab=readme-ov-file.

[13] A. Geitgey. "The world's simplest facial recognition api for python and the command line." (2018), [Online]. Available: https://github.com/ageitgey/face_recognition?tab=readme-ov-file.

[14] NumPy. "Numpy documentation." (2024), [Online]. Available: https://numpy.org/doc/stable/.

[15] Django. "Django documentation." (2023), [Online]. Available: https://docs.djangoproject.com/en/5.0/.

[16] Djangocentral. "Configuring static files in django." (2024), [Online]. Available: https://djangocentral.com/static-assets-in-django/.

[17] APEXCHARTS. "Modern interactive open-source charts." (2024), [Online]. Available: https://apexcharts.com/.

[18] w3schools. "Html div tag." (2024), [Online]. Available: https://www.w3schools.com/tags/tag_div.ASP.

[19] JavaScript.info. "Fetch request." (2024), [Online]. Available: https://javascript.info/fetch,.

[20] React. "A javascript library for building user interfaces." (2024), [Online]. Available: https://legacy.reactjs.org/docs/getting-started.html.

[21] D. R. framework. "Django rest framework is a powerful and flexible toolkit for building web apis.s." (2024), [Online]. Available: https://www.django-rest-framework.org/.