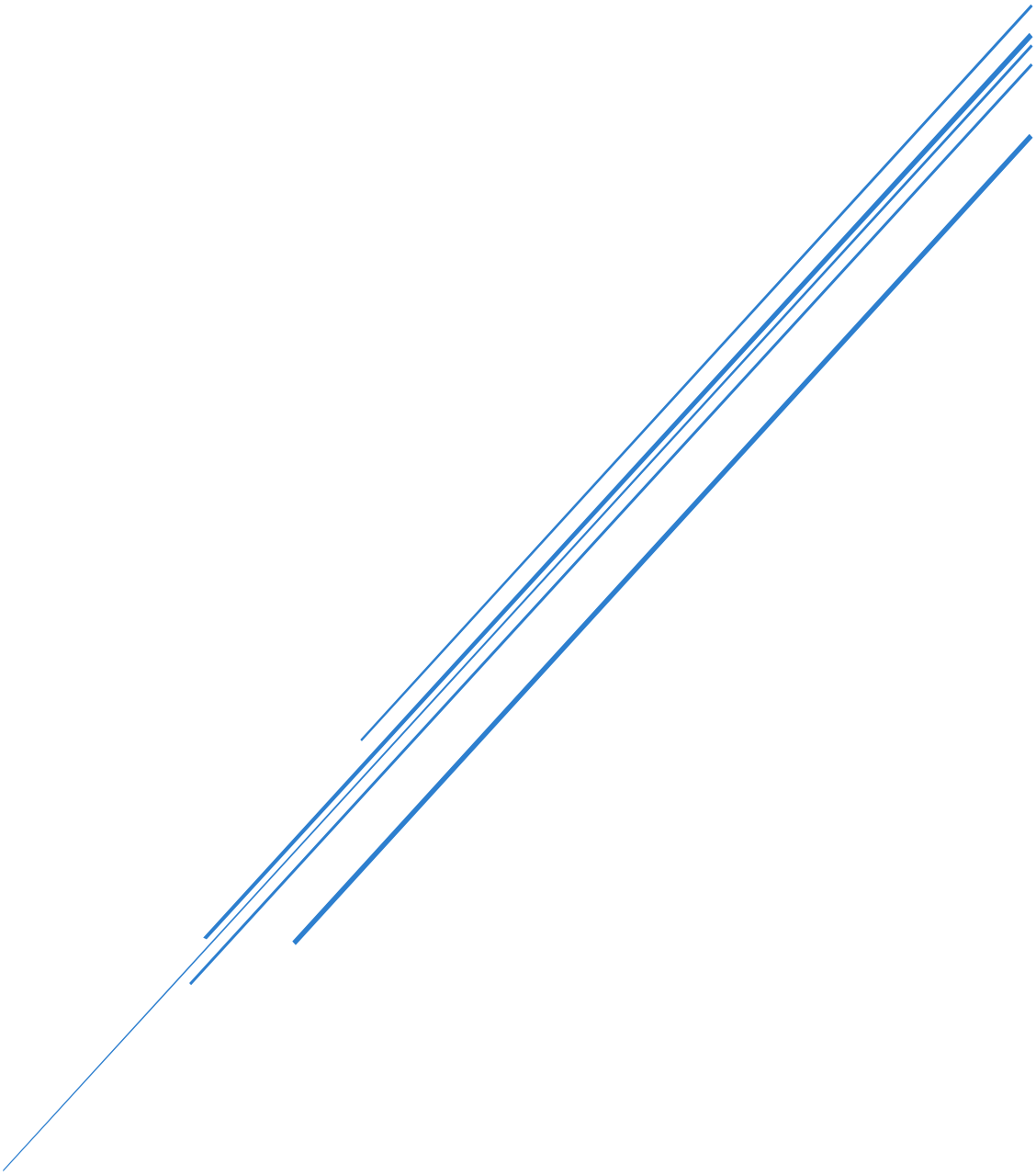
Automatic Door Control System

Ly Vien Minh Khoi – 103844366



Swinburne University of Technology

SWE30011 – IoT Programming

Table of Contents

Summary

In the age of automation control and digitalisation, there arise a need for an automatic door security system that relies on external signals and data to ensure timely but appropriate protection against malicious parties. While the current implementations of a security system utilize very sensitive sensors to detect human presence, along with sophisticated and complex mechanisms of identification that are very effective in ensuring optimal security; they lack contextual awareness, which may add another layer of reliability to the system without adding complexity and can even serve to replace some functionality that may be redundant. Furthermore, the abilities to track, monitor, and manually control the status of each component within such a system are also often missing, which when implemented will grant users significantly more control over their security, making the system overall more valuable. These realizations lead to the proposal of an Internet of Things (IoT) system that measure, detect, process both digital and analogues signals from the surrounding environment to manipulate actuators to serve security purposes that can also be managed by system owners via a graphical interface. This is the basis of the Automatic Door Control System that is proposed and developed in this project.

This system employs two sensors, one measure digital signals while the other measure analogue signals from its surrounding. The digital sensor, being an ultrasonic sensor, measure the distance of an incoming object by utilizing the principle of ultrasonic transmittance and reflection. This distance was then compared with a specific distance threshold to determine whether a person is within the vicinity. The analogue sensor, on the other hand, measure the brightness level of its surrounding by detect the level of electrical resistance within it. This is the working principle of a photoresistor, which is utilized to determine whether the system is being deployed during the day or at night. These two Boolean variables obtained from these sensors can be used to control two actuators: a servo motor representing the door hinge and a LED representing the alarm. If a person is detected to be nearby during daytime, the door will be opened by the servo motor while the alarm stays disable with the LED turned off. If there are no person detected nearby, the door stays close, and the alarm stays disable. At night, the door is kept always closed while the alarm will be enabled when a person is detected nearby, simulating malicious parties trying to enter the accommodation.

Based on these working principles, this system can be used to reinforce security at small-scaled stores that often close at night. Thus, in order to enhance its functionality, another feature was added to count the number of customers enter the store in a given period of time. This parameter can represent the traffic flow and can be used to estimate the stores performance in a time frame if further developed. Moreover, the system also allows store owners to enter the store at night, bypassing the automatic control logic by switching the operation mode to manual. In this mode, users can manually control the door and alarm LED without relying on the sensors signals, which can be useful in situations where stock needs to be refilled, or store maintenance needs to be done at nighttime.

To facilitate seamless integration of all aforementioned features, the system's intelligence is divided into a physical layer and an edge layer, which interact using serial communication. The sensors and actuators lie on the physical layer; while the ELEGOO Uno microcontroller interface with those components, as well as the Virtual Machine (VM) Raspberry Pi edge server lie on the edge layer. The edge server, specifically, is responsible for processing the logics to estimate customer counts, manual/auto mode toggle, and manual control of the actuators, all of which can be accessed through a web interface that was also hosted on the server. This web interface, which lies on an additional abstract layer called application layer, was developed using python Flask and HTML/CSS frontend presentation with MySQL database connection for continuous data logging and fetching. This edge computing approach allows minimal latency and superior processing power while also allowing store owners to have full access to all features of the system easily and remotely.

Conceptual Design

Block Diagram

The architecture of the Automatic Door Control System proposed can be found in the block diagram of Figure 1. Here, it can be seen that the abstract layers with the software and hardware components within them are clearly defined, with the system's intelligence composed of the physical and edge layer. These layers control the behaviour of the system, facilitating data generation, transmission, processing, and analysis. The application layer consists of the MySQL database connection, a well as the web interface implementation, with python Flask as backend and HTML/CSS along with JavaScript as frontend. The user layer is the system owner's control of the system, which is the lowest layer and mainly interface with the application layer to assign control of the components upward.

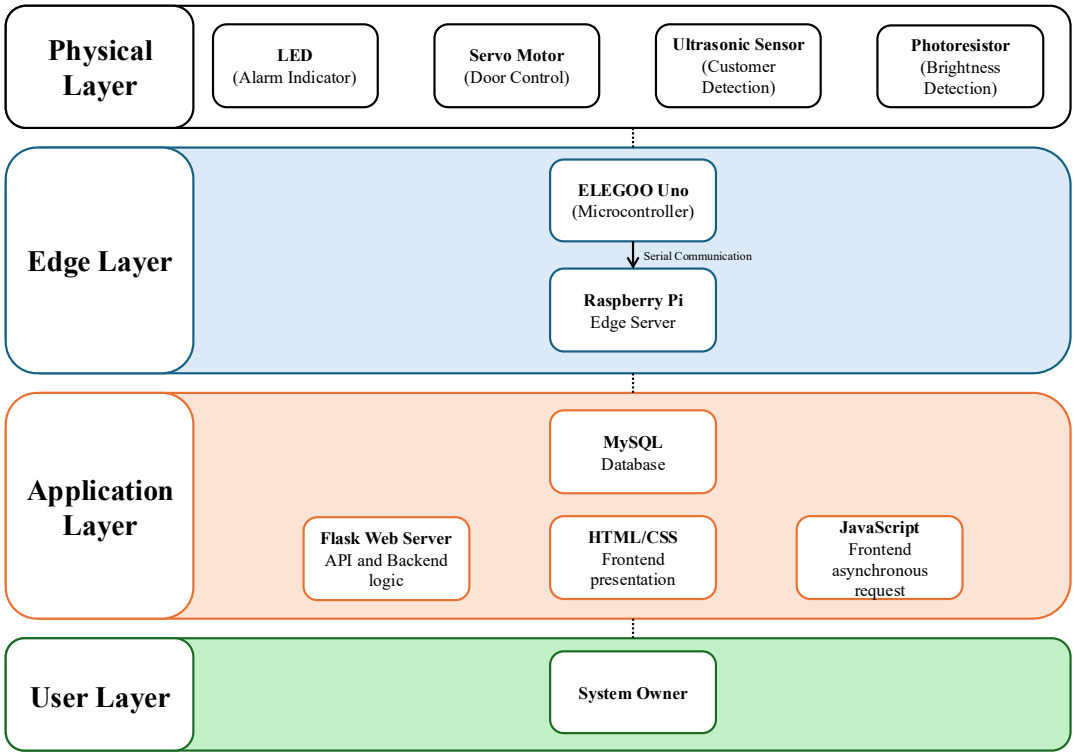


Figure 1. Automatic Door IoT system block diagram, with abstract layers defined.

UML Diagrams

The activity diagram describing the workflow of the ELEGOO Uno microcontroller is depicted in Figure 2, which encapsulates the two modes of operation of the system as well as the conditions handling from the data of both sensors. The microcontroller controls the sensors based on whether a person is nearby and whether the brightness of the working environment is sufficient in auto mode; and based on edge server's command in manual mode. The data collected from the physical components are then sent to the edge server, while the system continuously operates in a loop until manual shut down, indicating by the absence of the end node.

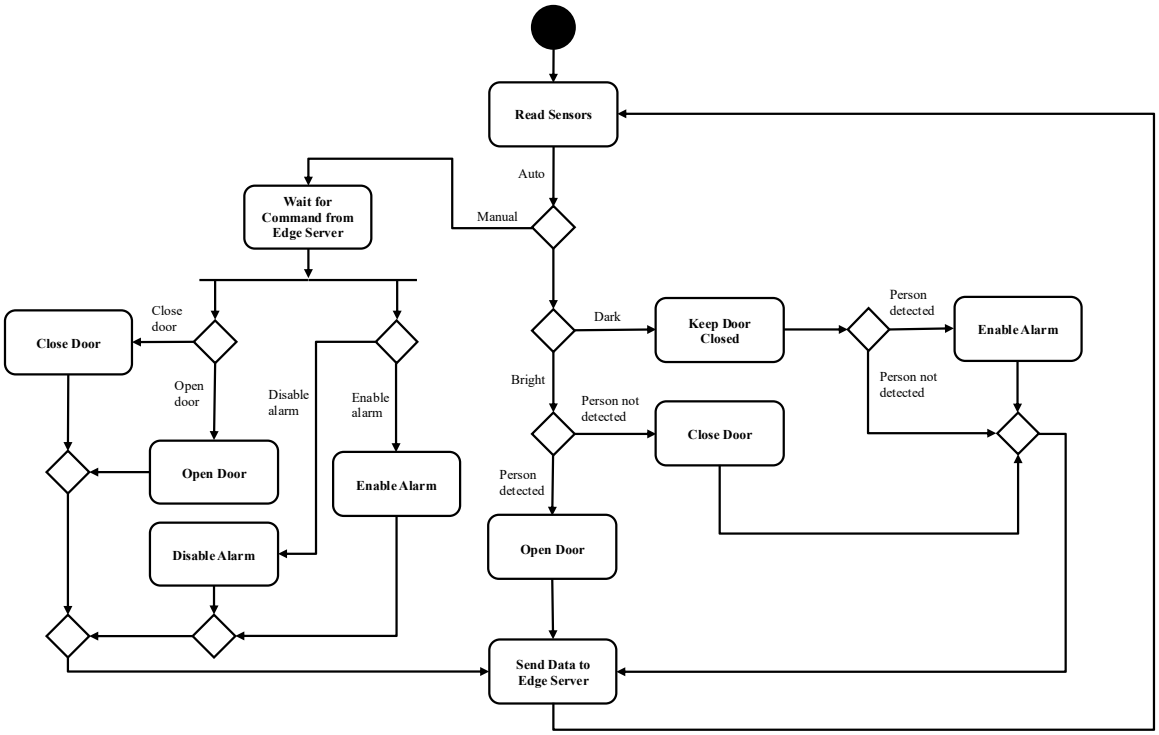


Figure 2. Activity diagram of the ELEGOO Uno microcontroller.

Similarly, the activity diagram for the Raspberry Pi edge server, runs on a VM, can be found in Figure 3. The data received from the microcontroller are saved in a database, from which the web interface will fetch, display and continuously update. The interface will accept five inputs from users: start and end dates, count, manual/auto toggle, open/close door toggle, and enable/disable alarm toggle, with the last four can be accessed by the corresponding buttons. User can only count the number of customers via the count button when they input a start and end dates, representing the period in which they want the customer count to be calculated. Likewise, the open/close door toggle button and enable/disable alarm toggle button can only be pressed when the operation mode is manual, meaning that they have to press the manual/auto toggle button to ensure that this is the case before they can manually toggle door or alarm state. The edge server also operates indefinitely in a loop, continuously receive and data from microcontroller to database, and process user input until error occurs.

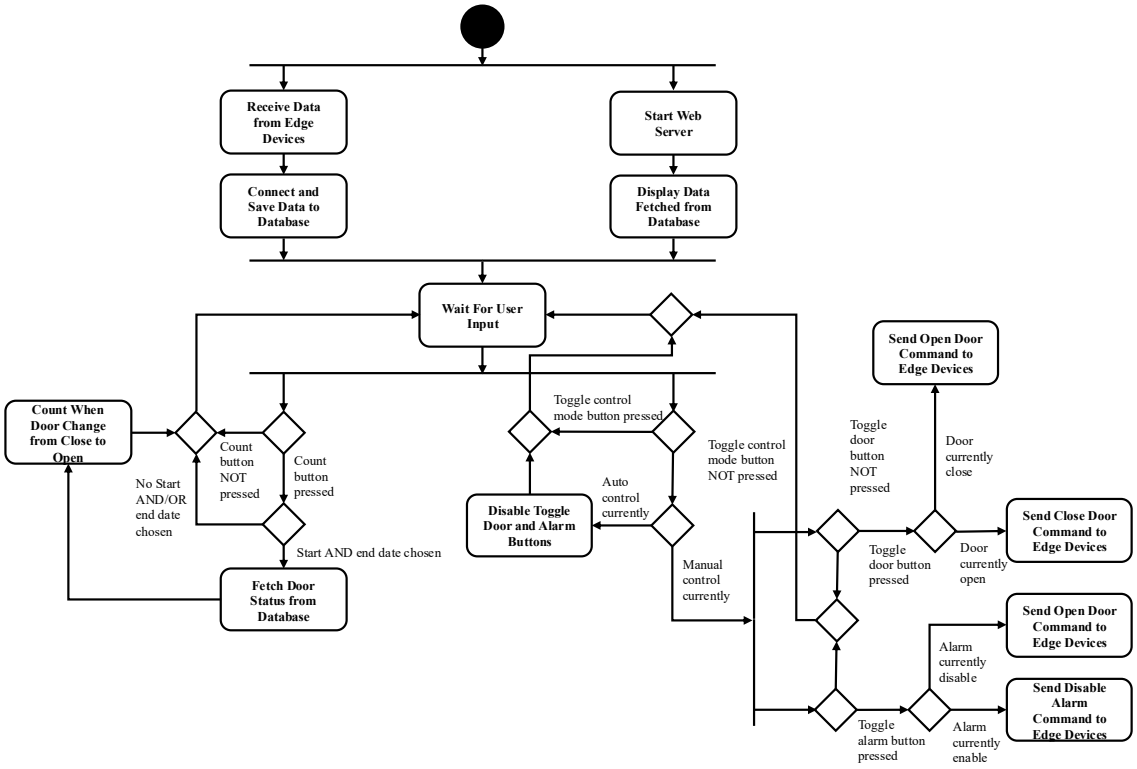


Figure 3. Activity diagram of the VM Raspberry Pi edge server.

Implementation

Sensors

Digital – Ultrasonic Sensor

As depicted in Figure 4, the digital sensor used in this project is the HC-SR04 ultrasonic sensor, which is an electronic component that use to estimate the distance from itself to an object.

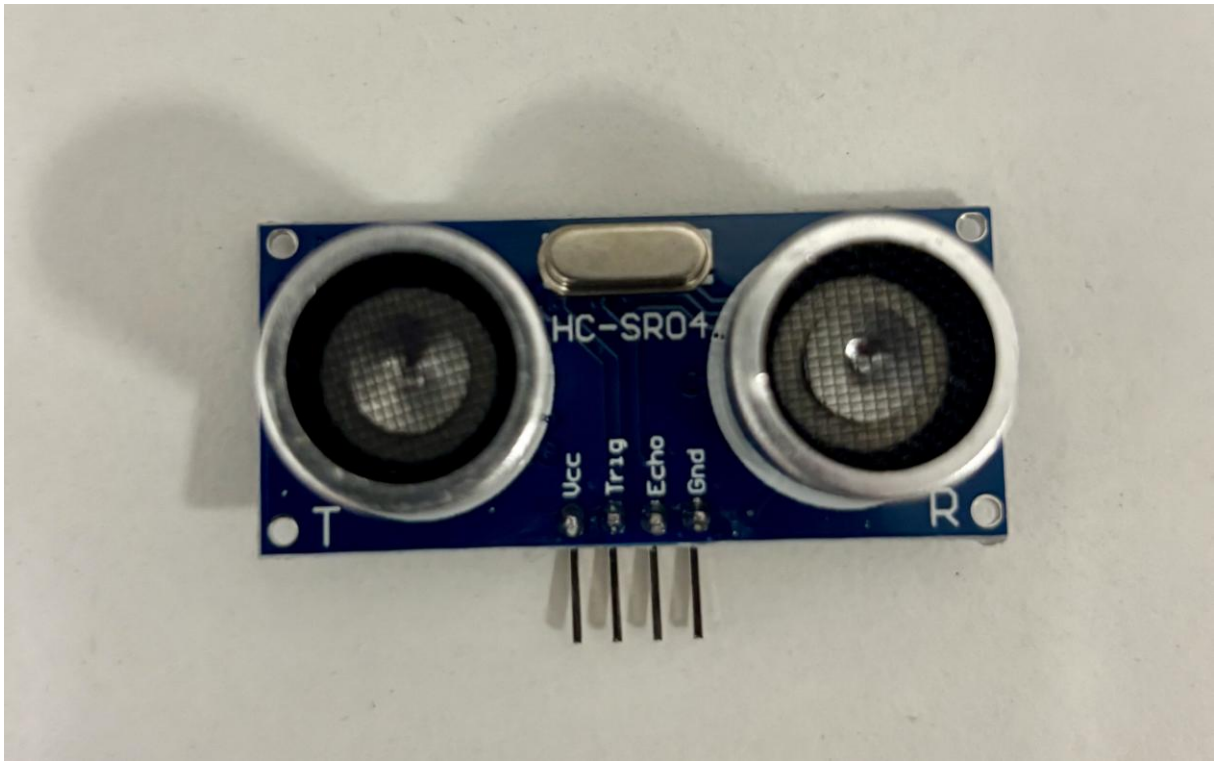


Figure 4. HC-SR04 Ultrasonic Sensor.

This sensor consists of a trigger part and an echo part, with the trigger part convert a digital signal sent from the microcontroller to an ultrasonic sound wave, which is a high frequency sound that human's ear cannot detect. This sound wave propagates through air at the speed of sound $V_{sound} = 343\ m/s$ until it hits an object in the vicinity and gets reflected back to the sensor. The echo part of the sensor then receives and convert this sound wave to a digital signal as output. The microcontroller can then measure the time it took from when the signal was being sent by the trigger part to when the echo part received the reflected signal via the built-in function pulseIn. This function returns a time in microsecond, which can be used to calculate the distance based on the equation $D = \frac{t \times V_{sound}}{2}$, with D and t being the distance of the object and the returned time respectively. The factor of 2 appears on the denominator of the equation is due to the fact that the sound wave needs to travel the required distance twice (both to and from the object) before reaching the echo part. This distance can finally be compared with an arbitrary threshold, which in this project is assumed to be 40 cm, to determine whether the object detected is sufficiently closed to the sensor to initiate actuators control. It is important to note that the speed of sound and the time measured need to be converted to appropriate units before calculation so that the distance obtained is in the same unit as the threshold for better clarity and comparison.

Analogue – Photoresistor (Light Sensor)

Figure 5 demonstrates an electronic component called photoresistor, which is the analogue sensor chosen for this project.

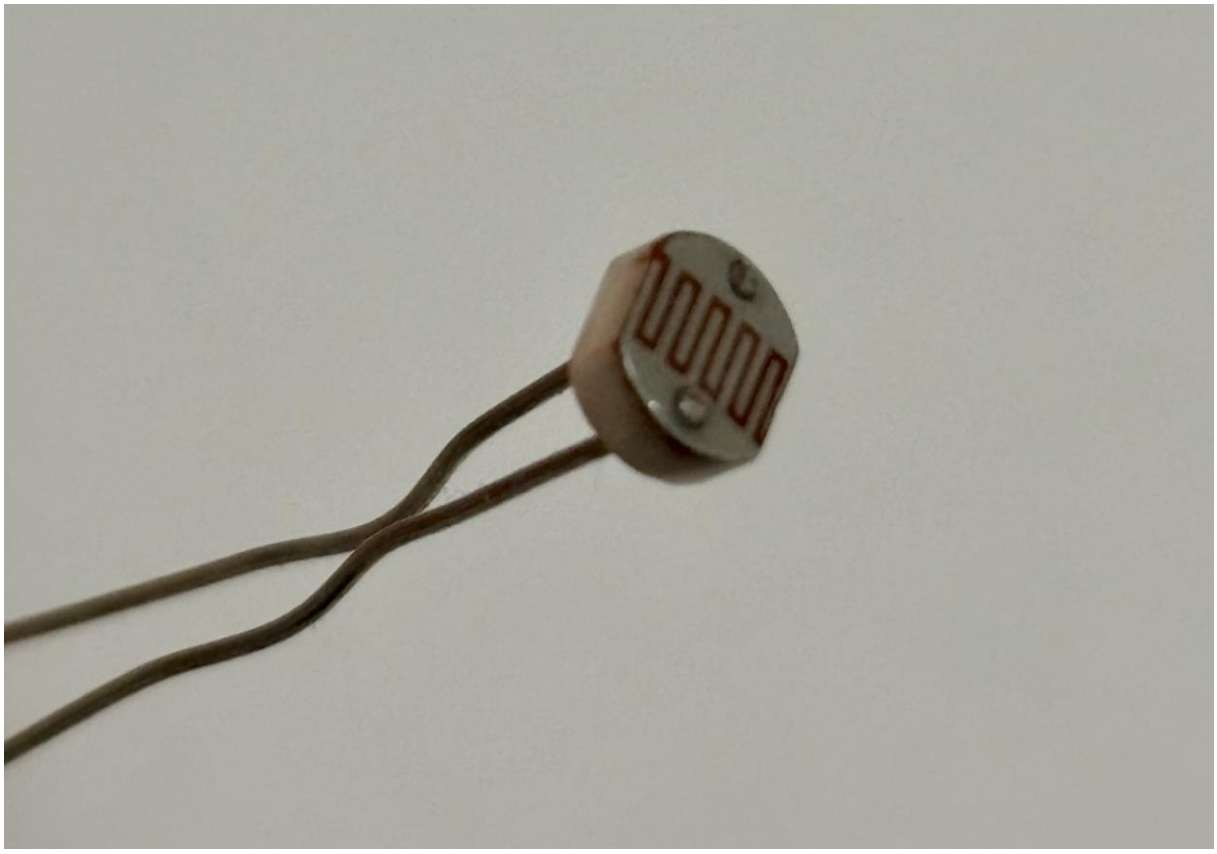


Figure 5. Photoresistor (Light Sensor).

This sensor behaves like a resistor, with resistance change depends on the light level it is receiving. The higher the light level, the lower the resistance, hence the higher the voltage allowed to flow through it. Utilizing this principle, the microcontroller read the analogue voltage signal from this sensor as the integer data type, which has a value ranging from 0 to 1023, corresponding to the voltage level the sensor is withstanding, which has a value ranging from 0 to 0 to 5V. This voltage is a proxy variable to actual light level, which is measured in a logarithmic unit lux, and thus can act as a pseudo light level indicator that can be used to determine whether the operating environment is bright or dark. The threshold value to which this voltage variable is compared to takes on the value of 400, which is an arbitrary value that is calibrated from actual testing and prototyping.

Actuators

Door Hinge – Servo Motor

The door hinge of the Automatic Door Control System is controlled by the SG90 servo motor, which is a high torque motor that offers full controllability over its angle of rotation. The component can be seen in Figure 6.

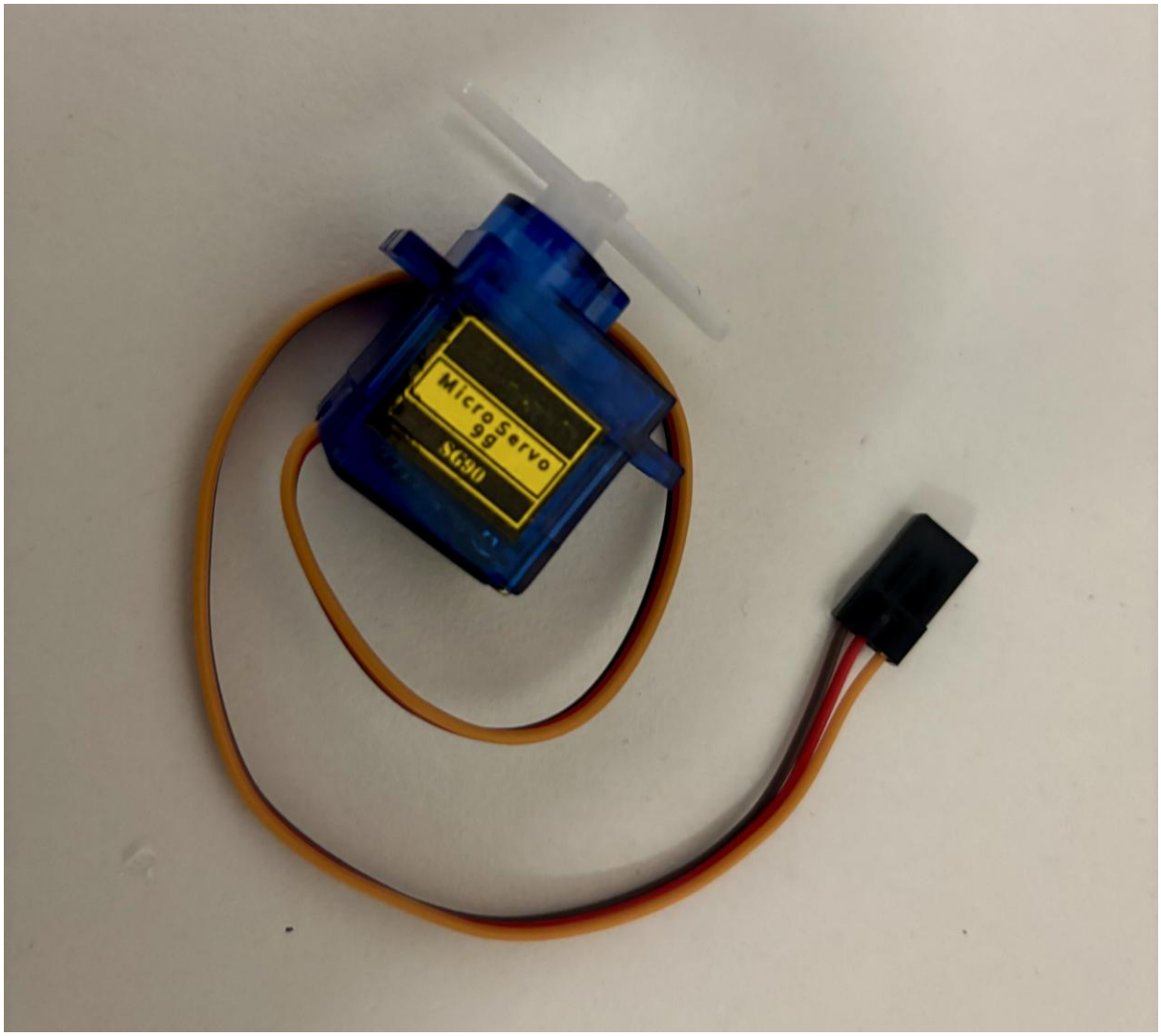


Figure 6. SG90 Servo Motor for Door Control.

This actuator consists of a series of interlocking gears made of plastic stored in a plastic casing, making it extremely light weight despite its impressive functionality. In order to precisely control its rotation angle, the motor utilise the Pulse Width Modulation (PWM) technique to simulate analogue control using only pulses of digital signal. In PWM, pulse of digital signal, which only takes on a value of minimum and maximum voltage, stays at maximum for a fixed period before immediately set to minimum for another fixed period. The period in which the pulse stays at maximum called pulse width while the entire period of the pulse changing from maximum to minimum is called a cycle, which gets repeat with a different pulse width for a short duration. The proportion of all the pulse width during that duration over the duration time corresponds to an analogue value that the servo motor can use to control its angle of rotation. The ELEGOO Uno microcontroller uses the Arduino Integrated Development Environment (IDE) to program its board, in which the Servo library is included that can fully interface with the PWM technique to control the servo motor without implementing it from scratch. The microcontroller sends a PWM signal to the pin that the motor is connected to based on the desirable angle, which in this project being 0 degree for door closing and 90 degrees for door opening. The logic for each of these scenarios can be seen in Figure 2, where the door only opens when there are people detected by the ultrasonic sensor during the day, which is determined by the photoresistor. At night or when there are no people nearby, the door remains closed, preventing malicious or unauthorized parties to enter the accommodation. The door can also be opened manually when the system's operation mode change to manual, allowing system owner to enter regardless of person detected or brightness level.

Alarm – LED

The alarm indicator is represented by a red LED, which can be found in Figure 7.

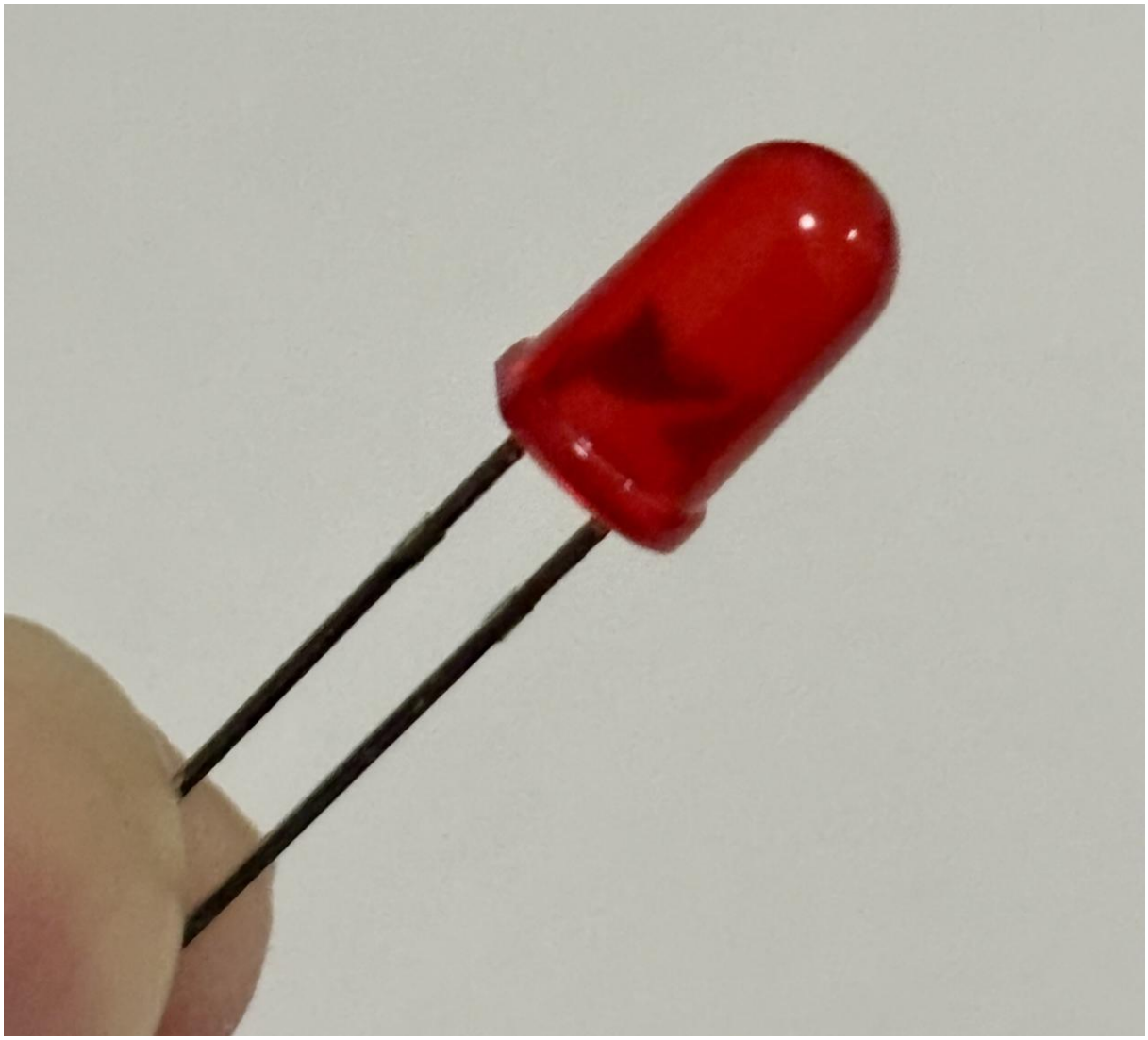


Figure 7. Red LED for Alarm Indicator.

The LED is a simple electrical component that lit up when there is a digital signal send to it. This is enabled according to the conditions set in Figure 2, which is only in the presence of a person during nighttime. This situation indicates that the person detected may be malicious or unauthorized parties trying to enter the accommodation, which severely compromises the system's owner privacy and security. Similar to the door hinge actuated by the servo motor, the alarm indicated by the LED can also be controlled manually in manual operation mode, allow system's owner to enable or disable the component at will.

The entire system's physical connection can be seen in Figure 8.

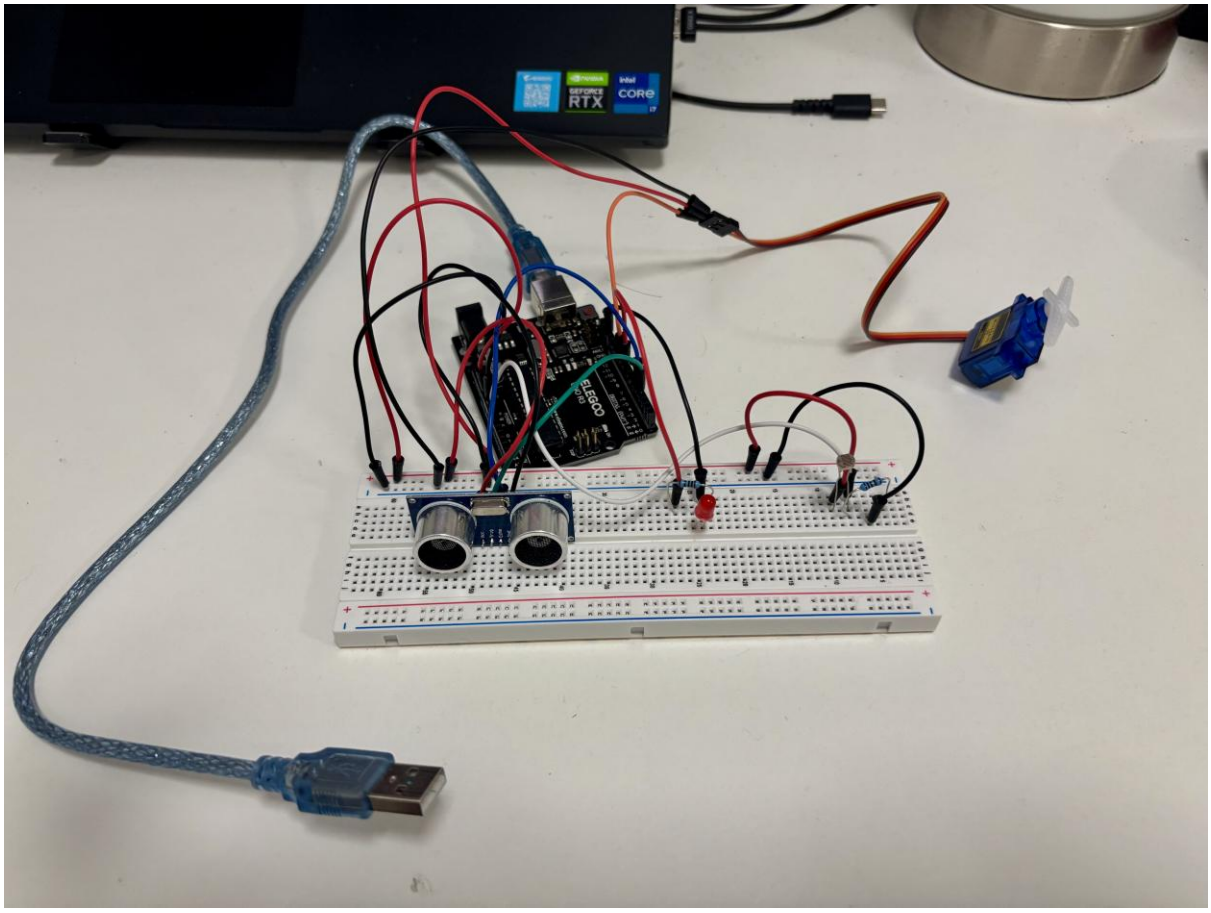


Figure 8. Entire System Physical Connection.

Software/Libraries

Serial Communications

As described in Figure 1, the two devices lie in the physical layer in the architecture: ELEGOO Uno microcontroller and VM Raspberry Pi edge server communicate with one another via serial communication, which transmit data one bit of a time. This communication method is often used to facilitate wired communication between microcontrollers with the same data transmission speed or baud rate (bit per second). The standard baud rate that is often set in Arduino or Arduino-like microcontrollers that used ATMega328p microprocessor is 9600, meaning that the edge server also needs to set the baud rate to the same number for successful communication. These devices also employ the standard Universal Asynchronous Receiver Transmitter (UART) serial protocol to send and receive data, which is common and easy to implement within the architecture. The protocol can be initiated by the built-in library Serial for both Arduino IDE and python with the common baud rate passed in as a parameter.

The data is sent from the microcontroller by utilizing the functions Serial.print and Serial.println, which send each instance of data as a line with the comma delimiter for each parameter. This makes each line of the data sent take on the form: [“Distance”: distance, “Light”: light level, “Manual Control”: Manual or Auto, “Door”: Open or Close, “Alarm”: On or Off], with the new line symbol at the end of each line to separate each data instance after finishing an Arduino loop. The presence of delimiters makes the data received on the edge server side easier to parse and save in the corresponding data structure.

The edge server receives the data from the microcontroller by initiating a serial object that handle data via a port on the edge server machine. The data needs to be decoded from its raw from of bit before data parsing and processing due to the microcontroller automatically encodes the data transmitted into

bit before initiated UART serial communication. This also works in reverse, when the edge server trying to send data representing buttons' command to the microcontroller. The command can be "manual", "door", or "alarm" corresponding to the toggle of control mode, door state, and alarm state respectively. This is done via the same port that was used to receive data initially by writing the encoded data to the serial object. The microcontroller can then receive the data by calling the function Serial.readStringUntil with the new line character as parameter. This data can be compared with the command expected and assign Boolean flags such as manualControl or doorOpened correspondingly.

Database

Additionally, Figure 1 also specifies that the database used is MySQL, which is an open-source relational database management system that can be easily installed and initialized on Linux-based devices like that of Raspberry Pi. The system is initialized by setting the username and password, before a database can be created via the CREATE DATABASE SQL query. The database used for this project is called doorlog, in which a table called logs was created via a python function that connect to the database and execute query using the cursor function. This table contains six columns of time, distance, light, control, door, and alarm, each of which corresponds to a parameter that get sent from the microcontroller each iteration. While the last five parameters get insert into the table at the time the edge server receive the data, the time parameter gets created by utilizing the library datetime in python to ensure that it reflect the timestamp when the data received on the edge server side instead of when they were created at the microcontroller side. This enhances reliability an accuracy, since the microcontroller internal clock can easily deteriorate with time as opposed to the universal time counter built within the software of the edge server. By instantly logging the data to the database as they are received, the edge server can fetch the data any time it needs without interfacing directly with the microcontroller. This decoupled the data generation and data processing logic, making it possible to fetch historical data even when the microcontroller stops working. Furthermore, it also reduces the workload on the less powerful microcontroller, which can only process and respond to a limited number of requests. This workflow can be seen in the functions that allow edge server to constantly display the latest logging entry or to count number of customers based on the change in door state from close to open. These processed can be executed independently from the microcontroller, which distributes the workload evenly across multiple interfaces and thus enhances the system's performance.

Webserver

In addition to handle receive data and database connection, the edge server is also responsible for hosting a webserver that serves as a graphical use interface for system owners to monitor and control their system remotely. This webserver is run on the main thread of the program, while the continuous data receiving and database logging run on a separate asynchronous thread, allowing two indefinite processes to be executed simultaneously without blocking one another. The implementation of such a webserver required the development of both backend, which holds the logics for controlling each functionality of the web page, and frontend, which present the functionality in an aesthetic and visually pleasing way. The overall architecture can be seen in the application layer of Figure 1.

Backend

The web page hosted on the edge server utilizes python's Flask micro web framework to program the backend logic. This is a basic and lightweight web framework that relies on route functions to control the web page effectively. In essence, the web page developed needs to have these functionalities:

1. Constantly display the latest logging entry of the database, including all parameters generated by the microcontroller.
2. Display the customer counts between a start datetime and an end datetime when user input the necessary fields and press the count button.

3. Allow users to switch control modes from auto to manual and vice versa via the control toggle button.
4. In manual mode, allow users to switch door state from close to open and vice versa via the door toggle button.
5. In manual mode, allow users to switch alarm state from on to off and vice versa via the alarm toggle button.

Each of these functionalities required a separate route in addition to a base route that only return the HTML template. The route redirect users to a different Uniform Resource Locator (URL) correspond to the functionality it handles meaning that users can also access these functionalities via modifying the URL of the web page without interacting with the page's elements at all. However, it is also important to note that the majority of these routes' function return the redirect function with the URL for the base route as the parameter. This means that the web page will refresh itself with the functionality executed when user press a button (or when navigating to the corresponding URL) instead of loading a new page altogether. This remove the needs to create a separate HTML file for each functionality, which is important when developing the frontend of the web page. The route corresponds to the first functionality is the only route that return a jsonify method with either a dictionary containing all latest log data or None as the parameter. This will return a JSON file, to which the frontend will process and continuously display the result via the execution of a JavaScript script. This process is further discussed in the next sub section.

Frontend

The frontend of the web page implemented was developed using HTML structuring and CSS styling to present its functionalities and elements in an appropriate format. While HTML utilizes elements like div, form, or h enclosed in angle brackets for both start and end tags to encapsulates and divides the display content into different area or level; CSS is responsible for giving these elements, identified by class, specified styling with custom colour scheme, font size, font family, size, and alignment. The collaborative usage of these languages facilitates the ability to create a simple, yet presentable web interface that is easy to navigate and customizable. Moreover, the elements present on the interface can also be further controlled by JavaScript, which acts as a collaborator between the backend and frontend logics. These scripts, placed in the HTML script tag, request specific variables on data structures from the backend to the frontend to dynamically updates these frontend elements without redirecting or refreshing the page. This is especially useful when displaying the logging data from the database, as well as when requesting the calculated customer count to be shown on the interface.

The structure of the web page is a grid with one column and four rows. The first row house the latest log display panel, with six boxes distributed horizontally in a nested grid container, representing the value for six parameters saved in the database. The continuous displaying is controlled by a JavaScript function, which reads the JSON file returned by the current backend route, parse the file into appropriate data structures, and pass them into the boxes with the corresponding id for display. This process, along with others that will get discussed later, will repeat itself every second, initiated by the function setInterval in the script. On the other hand, the second row of the main grid contains the start and end datetime input, encapsulated in a flex box container that allow its items to stretch horizontally. The input field, identified by the input label, is the area where users can input a datetime either manually or chosen via a pop-up window that can be opened with a calendar-shaped button on the right side of the field. The filling of the field is specified to be a requirement before the Count button can be pressed, which resides in the third row of the grid as part of a button panel.

In the third row, four button corresponding to four functionalities of the web page resides: Count, Manual/Auto Control, Open/Close Door, and Enable/Disable Alarm. The last three buttons' name is also controlled by a JavaScript function, albeit more complex due to sophisticated conditionings. Since the name needs to be changed to the state opposite to the current state of the togglable system

components, regardless of whether that component is the control mode, door, or alarm, the function needs to check the current state before assigning the name to each button. Furthermore, because the Open/Close Door button and the Enable/Disable Alarm button are only active and ready to be pressed when the current control mode is manual, the function also needs to change the class of the corresponding buttons to “inactive” when the condition is met. This ensures that system owners know exactly which mode the system is currently operating in, along with which buttons they can press to control their system at any given time.

Finally, the fourth row of the main grid house the area for displaying the customer count when users requested via the Count button and the datetime input fields. This section is not visible until a request has been made by users and is controlled by two JavaScript functions: one extract the count result from the URL using a parameter name (in this case is “count”), while the other pass the count result into the customerCount element in the section. Since the methods used by the “/traffic” route, which is the route responsible for the count customer functionality, include POST, and the route return the base URL along with the count variable as a parameter, the count result will appear in the URL in the format “count?=c”, with c being the resulting number. This URL can thus be parse using Regular Expression to search and extract any expression that followed by query characters such as “?” or “&”, before saving them into appropriate data structures that get returned. This implementation allows the customer count to dynamically update without the frontend directly interface with backend, enhancing the decoupling properties of the system.

The visual representation of the web interface can be seen below.

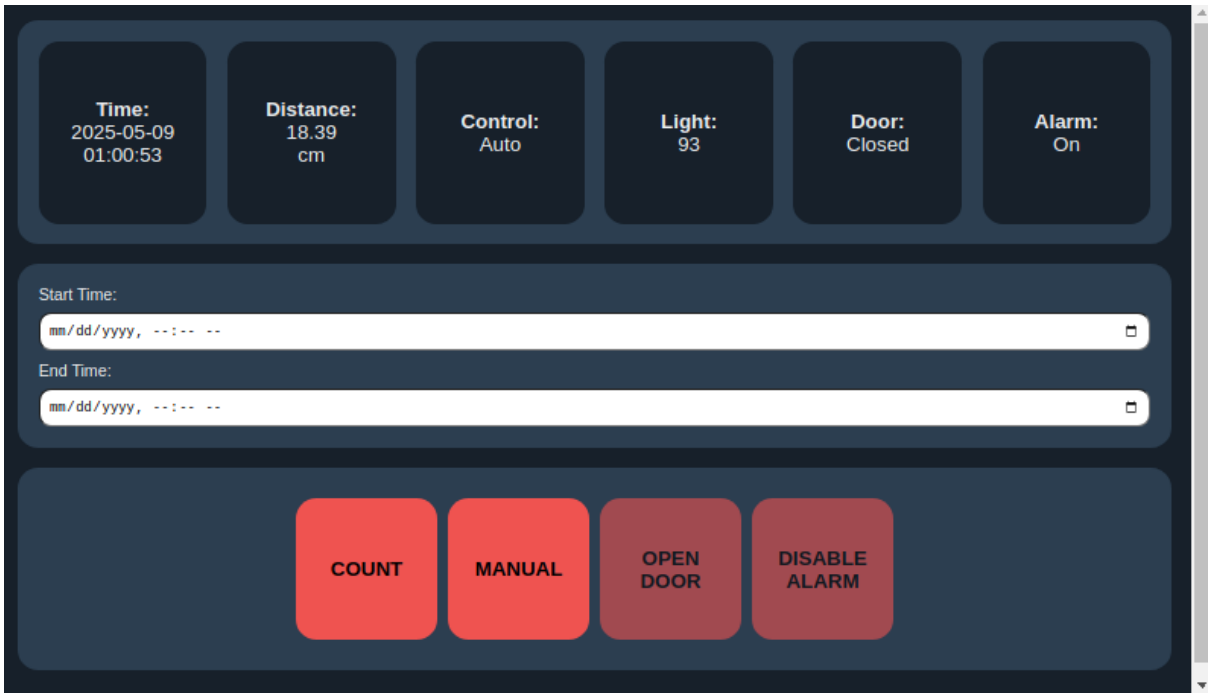


Figure 9. Default web interface.

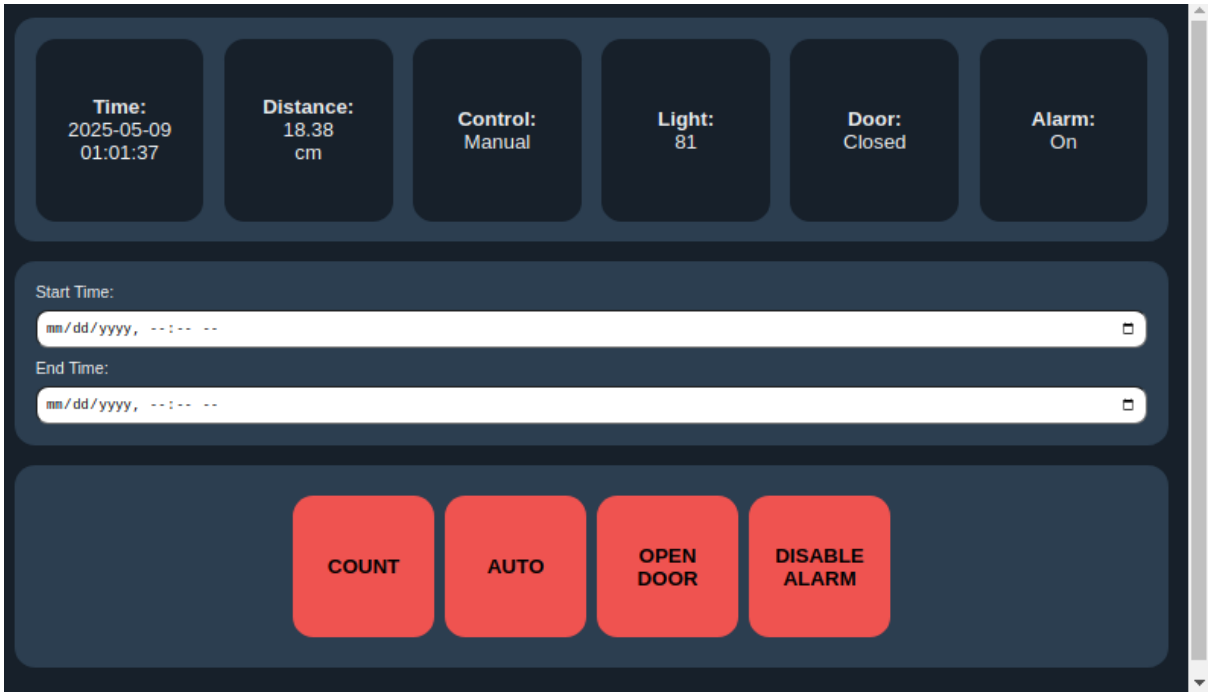


Figure 10. Web interface with Manual control operation mode.

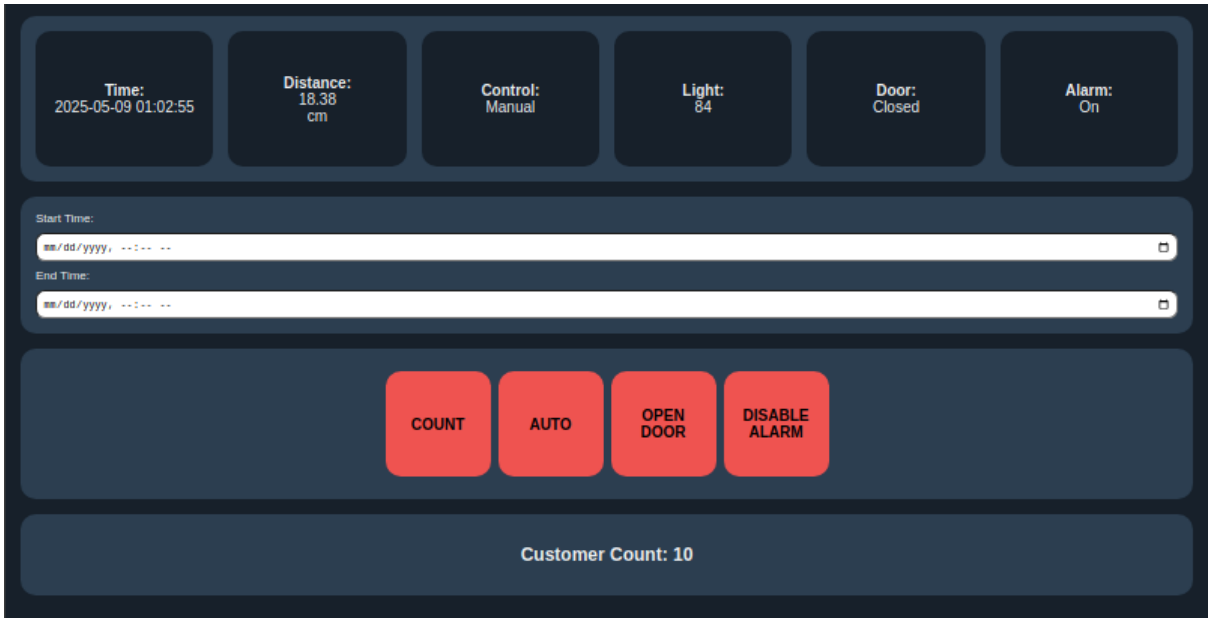


Figure 11. Web interface with customer counts display.

Resources

1. Ultrasonic Sensor: https://how2electronics.com/how-to-use-hc-sr04-ultrasonic-distance-sensor-with-arduino/
2. Photoresistor: https://projecthub.arduino.cc/tropicalbean/how-to-use-a-photoresistor-1143fd
3. Serial Input: https://forum.arduino.cc/t/serial-input-basics-updated/382007
4. Python Serial Communication: SWE30011 Tutorial 7 Task 4
5. MySQL Connection: SWE30011 Tutorial 7 Task 2
6. MySQL Installation: https://www.geeksforgeeks.org/how-to-install-mysql-on-linux/
7. MySQL User Creation: https://www.geeksforgeeks.org/how-to-install-mysql-on-linux/

8. MySQL Database Creation: https://www.geeksforgeeks.org/how-to-install-mysql-on-linux/
9. MySQL Table Creation: https://www.w3schools.com/mysql/mysql_create_table.asp
10. MySQL Insert Data: https://www.w3schools.com/mysql/mysql_insert_into_select.asp
11. MySQL Select Data: https://www.w3schools.com/mysql/mysql_select.asp
12. MySQL Select Data Order By: https://www.w3schools.com/mysql/mysql_orderby.asp
13. Flask Webserver Initialization: SWE30011 Tutorial 7 Task 4
14. HTML Basics: https://www.w3schools.com/html/
15. CSS Basics: https://www.w3schools.com/css/
16. HTML Align: https://www.geeksforgeeks.org/html-align-attribute/
17. CSS Grid: https://www.w3schools.com/css/css_grid.asp
18. CSS Grid Items: https://www.w3schools.com/css/css_grid_item.asp
19. CSS Box Model: https://www.w3schools.com/css/css_boxmodel.asp
20. HTML Datetime Form: https://www.w3schools.com/tags/att_input_type_datetime-local.asp
21. CSS Flex Container: https://www.w3schools.com/css/css3_flexbox_container.asp
22. CSS Flex Box: https://www.w3schools.com/css/css3_flexbox.asp
23. CSS Button: https://www.w3schools.com/tags/tag_button.asp
24. CSS Disabled Button: https://www.w3schools.com/tags/att_button_disabled.asp
25. HTML Colour Codes: https://htmlcolorcodes.com
26. CSS Font Family: https://developer.mozilla.org/en-US/docs/Web/CSS/font-family
27. Flask Jsonify: https://www.geeksforgeeks.org/use-jsonify-instead-of-json-dumps-in-flask/
28. JavaScript Fetch Data From Backend: https://www.slingacademy.com/article/integrating-fetch-api-results-into-the-javascript-dom/
29. JavaScript Assign Data To Frontend Elements: https://www.slingacademy.com/article/from-static-to-dynamic-updating-the-dom-in-real-time-with-javascript/
30. JavaScript Fetch Query String From URL: https://stackoverflow.com/questions/901115/how-can-i-get-query-string-values-in-javascript

Appendix

Microcontroller sketch

```
#include <Servo.h>

const int trigPin = 3;
const int echoPin = 4;
const int ldrPin = A0;
const int ledPin = 8;
const int servoPin = 9;

bool doorOpened = false;
bool alarmOn = false;

bool manualControl = false;  // Tracks if actuators can be trontrolled
manually

Servo doorServo;

void setup() {
  Serial.begin(9600);
```

```
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
  pinMode(ledPin, OUTPUT);
  doorServo.attach(servoPin);
  doorServo.write(0);  // Door closed
}

void loop() {
  // Distance calculation
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);
  long duration = pulseIn(echoPin, HIGH);
  float distance = duration * 0.034 / 2;
  bool personDetected = (distance < 50);

  // Light level calculation
  int lightLevel = analogRead(ldrPin);
  bool isBright = lightLevel > 400;

  // Check for serial commands
  if (Serial.available()) {
    String command = Serial.readStringUntil('\n');
    command.trim();

    if (command == "manual") {
      manualControl = !manualControl;  // Activate manual override
    }

    if (command == "door") {
      if (manualControl) {
        doorOpened = !doorOpened;  // Toggle door state
      }
    }

    if (command == "alarm") {
      if (manualControl) {
        alarmOn = !alarmOn;  // Toggle alarm state
      }
    }
  }

  // ---Door Control---
  if (!manualControl) {
    // Only apply automatic control if not in manual override
    if (personDetected && isBright) {
```

```
      // At day when there are people
      doorOpened = true;
    } else {
      // At night or no people
      doorOpened = false;
    }
  }

  // ---Alarm Control---
  if (!manualControl) {
    // Only apply automatic control if not in manual override
    if (personDetected && !isBright) {
      // At night when there are people
      alarmOn = true;
    } else {
      alarmOn = false;
    }
  }

  // Apply physical controls
  doorServo.write(doorOpened ? 90 : 0);
  digitalWrite(ledPin, alarmOn ? HIGH : LOW);

  // Serial output to edge server
  Serial.print("Distance:");
  Serial.print(distance);
  Serial.print(", Light:");
  Serial.print(lightLevel);
  Serial.print(", Manual Control:");
  Serial.print(manualControl ? "Manual" : "Auto");
  Serial.print(", Door:");
  Serial.print(doorOpened ? "Open" : "Closed");
  Serial.print(", Alarm:");
  Serial.println(alarmOn ? "On" : "Off");

  delay(500);  // Check every half a second
}
```

Python Script

```
import serial
import pymysql
from datetime import datetime
from flask import Flask, render_template, request, redirect, url_for, jsonify
import threading
import time

ser = serial.Serial('/dev/ttyACM0', 9600, timeout=1)
```

```
app = Flask(__name__)

def get_db_connection(host='localhost', user='root', password='12345678',
db='doorlog'):
    conn = pymysql.connect(host=host, user=user, password=password,
database=db)
    cur = conn.cursor()
    cur.execute('''
        CREATE TABLE IF NOT EXISTS logs (
            time DATETIME,
            distance FLOAT,
            light INT,
            control VARCHAR(20),
            door VARCHAR(20),
            alarm VARCHAR(20)
        )
    ''')
    conn.commit()
    return conn

def log_data():
    while True:
        try:
            if ser.in_waiting:
                line = ser.readline().decode().strip()
                parts = line.split(',')
                dist = float(parts[0].split(':')[1])
                light = int(parts[1].split(':')[1])
                control = parts[2].split(':')[1]
                door = parts[3].split(':')[1]
                alarm = parts[4].split(':')[1]
                now = datetime.now()

                conn = get_db_connection()
                cur = conn.cursor()
                cur.execute("INSERT INTO logs (time, distance, light, control,
door, alarm) VALUES (%s, %s, %s, %s, %s, %s)",
                            (now, dist, light, control, door, alarm))
                conn.commit()
                conn.close()


        except Exception as e:
            print("Error reading from Arduino:", e)
        time.sleep(1)

def fetch_latest_data(conn):
    cur = conn.cursor()
```

```
    cur.execute("SELECT * FROM logs ORDER BY time DESC LIMIT 1")
    data = cur.fetchone()
    return data


def fetch_data_column(start_time, end_time, conn, parameter):
    if parameter not in ['distance', 'light', 'control', 'door', 'alarm']:
        raise ValueError("Invalid column requested")

    cur = conn.cursor()
    query = f"SELECT {parameter} FROM logs WHERE time BETWEEN %s AND %s"
    cur.execute(query, (start_time, end_time))
    data = [row[0] for row in cur.fetchall()]
    conn.close()
    return data


def count_customer(door_states):
    count = 0
    for i in range(len(door_states) - 1):
        if door_states[i] == 'Closed' and door_states[i + 1] == 'Open':
            count += 1
    return round(count / 2)

@app.route('/')
def index():
    count = request.args.get('count')
    return render_template('index.html', count=count)

@app.route('/latest-log')
def latest_log():
    conn = get_db_connection()
    cur = conn.cursor()
    cur.execute("SELECT * FROM logs ORDER BY time DESC LIMIT 1")
    row = cur.fetchone()
    conn.close()

    if row:
        return jsonify({
            'time': row[0].strftime('%Y-%m-%d %H:%M:%S'),
            'distance': row[1],
            'light': row[2],
            'control': row[3],
            'door': row[4],
            'alarm': row[5]
        })
    return jsonify(None)
```

```python
@app.route('/traffic', methods=['GET', 'POST'])
def customer_traffic():
    count = None
    if request.method == 'POST':
        start = request.form['start']
        end = request.form['end']
        conn = get_db_connection()
        doors = fetch_data_column(start, end, conn, 'door')
        count = count_customer(doors)
        return redirect(url_for('index', count=count))

    return redirect(url_for('index'))

@app.route('/manual-control', methods=['POST'])
def manual_toggle():
    try:
        ser.write(b'manual\n')
        return redirect(url_for('index'))
    except Exception as e:
        return f"Error sending data to Arduino: {e}", 500

@app.route('/door-toggle', methods=['POST'])
def door_toggle():
    try:
        ser.write(b'door\n')
        return redirect(url_for('index'))
    except Exception as e:
        return f"Error sending data to Arduino: {e}", 500

@app.route('/alarm-toggle', methods=['POST'])
def alarm_toggle():
    try:
        ser.write(b'alarm\n')
        return redirect(url_for('index'))
    except Exception as e:
        return f"Error sending data to Arduino: {e}", 500


threading.Thread(target=log_data, daemon=True).start()

if __name__ == '__main__':
    app.run(debug=True)
```

HTML/CSS File

```html
<!doctype html>
<html>
```

```
<head>
  <title>Door Logs</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <style>
    body {
      background-color: #17202a;
      font-family: Helvetica, sans-serif;
      color: #eaecee;
      margin: 0;
      padding: 0;
    }

    .grid-container {
      display: grid;
      grid-template-columns: 1fr;
      grid-template-rows: auto auto auto auto;
      gap: 20px;
      padding: 20px;
      width: 100%;
      box-sizing: border-box;
    }

    .date-input-area {
      grid-column: 1;
      grid-row: 2;
      padding: 20px;
      background-color: #2c3e50;
      border-radius: 20px;
    }

    .date-input-form {
      display: flex;
      flex-direction: column;
      gap: 20px;
      align-items: flex-start;
    }

    .input-fields {
      display: flex;
      flex-direction: column;
      gap: 10px;
      flex-grow: 1;
      width: 100%;
    }

    .date-input-form .input-group {
      display: flex;
      flex-direction: column;
```

```
    align-items: flex-start;
    gap: 10px;
    width: 100%;
  }

  .date-input-form label {
    width: 100px;
  }

  .date-input-form input {
    flex-grow: 1;
    padding: 8px;
    border-radius: 12px;
    width: 100%;
    box-sizing: border-box;
  }

  .button-area {
    grid-column: 1;
    grid-row: 3;
    padding: 20px;
    display: flex;
    align-items: center;
    justify-content: center;
    background-color: #2c3e50;
    gap: 10px;
    flex-wrap: wrap;
    border-radius: 20px;
  }

  .log-container {
    display: grid;
    grid-template-columns: repeat(auto-fit, minmax(150px, 1fr));
    gap: 20px;
    padding: 20px;
    background-color: #2c3e50;
    grid-column: 1;
    grid-row: 1;
    text-align: center;
    border-radius: 20px;
  }

  .log-item {
    background: #17202a;
    padding: 15px;
    border-radius: 20px;
    font-size: 20px;
    display: flex;
```

```
    flex-direction: column;
    justify-content: center;
    align-items: center;
    min-height: 150px;
  }

  .button {
    background-color: #ef5350;
    border: none;
    color: black;
    padding: 10px;
    text-align: center;
    display: inline-block;
    text-decoration: none;
    font-size: 20px;
    margin: 10px 0;
    cursor: pointer;
    border-radius: 20px;
    min-width: 140px;
    width: 100%;
    max-width: 200px;
    min-height: 140px;
    height: 100%;
    max-height: 200px;
    font-weight: bold;
    font-family: Helvetica, sans-serif;
  }

  .button:hover:not(.inactive) {
    background-color: #b71c1c;
    color: black;
  }

  .button.inactive {
    cursor: not-allowed;
    opacity: 0.6;
  }

  .count-result {
    font-weight: bold;
    grid-column: 1;
    grid-row: 4;
    background-color: #2c3e50;
    padding: 20px;
    font-family: Helvetica, sans-serif;
    border-radius: 20px;
    text-align: center;
  }
```

```
  </style>
</head>

<body>
  <div class="grid-container">

    <!-- Log Area -->
    <div class="log-container">
      <div class="log-item">
        <strong>Time:</strong>
        <span id="time"></span>
      </div>
      <div class="log-item">
        <strong>Distance:</strong>
        <span id="distance"></span> cm
      </div>
      <div class="log-item">
        <strong>Control:</strong>
        <span id="control"></span>
      </div>
      <div class="log-item">
        <strong>Light:</strong>
        <span id="light"></span>
      </div>
      <div class="log-item">
        <strong>Door:</strong>
        <span id="door"></span>
      </div>
      <div class="log-item">
        <strong>Alarm:</strong>
        <span id="alarm"></span>
      </div>
    </div>

    <!-- Input Area -->
    <div class="date-input-area">
      <form method="POST" class="date-input-form" id="countForm"
action="/traffic" method="POST">
        <div class="input-fields">
          <div class="input-group">
            <label>Start Time:</label>
            <input type="datetime-local" name="start" required />
          </div>
          <div class="input-group">
            <label>End Time:</label>
            <input type="datetime-local" name="end" required />
          </div>
        </div>
```

```
      </form>
    </div>

    <!-- Button Panel Area -->
    <div class="button-area">
      <div style="display: flex; justify-content: center; width: 100%; max-
width: 140px;">
        <button type="submit" class="button" form="countForm">COUNT</button>
      </div>
      <form action="/manual-control" method="POST" style="margin: 0; display:
flex; justify-content: center; width: 100%; max-width: 140px;">
        <button type="submit" class="button" id="controlButton">MANUAL
CONTROL</button>
      </form>

      <form action="/door-toggle" method="POST" style="margin: 0; display:
flex; justify-content: center; width: 100%; max-width: 140px;">
        <button type="submit" class="button" id="doorButton">OPEN
DOOR</button>
      </form>

      <form action="/alarm-toggle" method="POST" style="margin: 0; display:
flex; justify-content: center; width: 100%; max-width: 140px;">
        <button type="submit" class="button" id="alarmButton">ENABLE
ALARM</button>
      </form>
    </div>

    <div class="count-result" id="countResultDiv" style="display: none;">
      <h2>Customer Count: <span id="customerCount">0</span></h2>
    </div>
  </div>

  <!-- Java Script -->
  <script>
    // Get the count parameter from URL
    function getParameterByName(name, url = window.location.href) {
      name = name.replace(/[\[\]]/g, '\\$&');
      var regex = new RegExp('[?&]' + name + '(=([^&#]*)|&|#|$)'),
        results = regex.exec(url);
      if (!results) return null;
      if (!results[2]) return '';
      return decodeURIComponent(results[2].replace(/\+/g, ' '));
    }
    // Update the count result from URL parameter
    function updateCountResult() {
      const count = getParameterByName('count');
```

```
    if (count !== null) {
      document.getElementById('customerCount').textContent = count;
      document.getElementById('countResultDiv').style.display = 'block';
    }
  }
  // Update button text based on current state
  function updateButtonText(controlStatus, doorStatus, alarmStatus) {
    const controlButton = document.getElementById('controlButton');
    const doorButton = document.getElementById('doorButton');
    const alarmButton = document.getElementById('alarmButton');
    if (controlStatus === 'Auto') {
      controlButton.textContent = 'MANUAL';
      doorButton.classList.add('inactive');
      alarmButton.classList.add('inactive');
      doorButton.disabled = true;
      alarmButton.disabled = true;
    } else {
      controlButton.textContent = 'AUTO';
      doorButton.classList.remove('inactive');
      alarmButton.classList.remove('inactive');
      doorButton.disabled = false;
      alarmButton.disabled = false;
    }
    if (doorStatus === 'Open') {
      doorButton.textContent = 'CLOSE DOOR';
    } else {
      doorButton.textContent = 'OPEN DOOR';
    }
    if (alarmStatus === 'On') {
      alarmButton.textContent = 'DISABLE ALARM';
    } else {
      alarmButton.textContent = 'ENABLE ALARM';
    }
  }
  // Constantly update the log
  function fetchLatestLog() {
    fetch('/latest-log')
      .then(response => response.json())
      .then(data => {
        if (data) {
          document.getElementById('time').textContent = data.time;
          document.getElementById('distance').textContent = data.distance;
          document.getElementById('light').textContent = data.light;
          document.getElementById('control').textContent = data.control;
          document.getElementById('door').textContent = data.door;
          document.getElementById('alarm').textContent = data.alarm;
          // Update button text based on status
          updateButtonText(data.control, data.door, data.alarm);
```

```
          }
        })
        .catch(error => console.error('Error fetching log:', error));
    }
    // Run on page load
    updateCountResult();
    fetchLatestLog();
    // Set interval for updates
    setInterval(fetchLatestLog, 1000);
  </script>
</body>

</html>
```