**SRI LANKA INSTITUTE OF INFORMATION TECHNOLOGY**

A picture containing drawing

Description automatically generated

Fire Alarm Monitoring system

Distributed system Assignment 2

**1. Diagrams**

**High level architecture Diagram**

**A close up of a map

Description automatically generated**

**Sequence Diagram**

**A close up of text on a white background

Description automatically generated**

**2. Service Interfaces**

**2.1 REST Api**

REST Api was developed with Node JS and it has following service interfaces.

**Get all the sensors**

Service can be obtained using the endpoint - <http://localhost:4000/getAllSensors>

Request Type - GET

Users can get all the sensors using this end point. This endpoint is used by web client and desktop client to display sensors to users. Web client directly accesses the api and desktop client access it through RMI server.

See **appendix A** for the code

**Update sensor**

Service can be obtained using the endpoint - <http://localhost:4000/updateSensor/:id>

Request Type - PATCH

Privileged users such as admin can update sensors by sensor Id. In the desktop client this service has been used and user can search a room by giving the floor number and room number then update the room number and floor number and status of the sensor.

See **appendix B** for the code

**Delete Sensor**

Service can be obtained using the endpoint - <http://localhost:4000/deleteSensor/:id> Request

Type - DELETE

Privileged users such as admin can delete sensors by sensor Id. In the desktop client this service has been used and user can search a room by giving the floor number and room number then delete the sensor.

See **appendix C** for the code

**Add Sensor**

Service can be obtained using the endpoint - <http://localhost:4000/addSensor>

Request Type - POST

Privileged users such as admin can delete sensors by sensor Id. In the desktop client this service has been used and user can add a sensor by giving location and active status.

See **appendix D** for the code.

**Add Room Details**

Service can be obtained using the endpoint - <http://localhost:4000/addRoomDetails> Request

Type - POST

Privileged users such as admin can add users to the relevant room using this service. Admin can add location and user details such as email, mobile number.

See **appendix E** for the code.

**Get Room Details**

Service can be obtained using the endpoint - <http://localhost:4000/getRoomDetails/:id>

Request Type - GET

In case of emergency situation contact details such as email and phone number of the relevant user can be obtained this service interface. Users of this service can give roomId and get necessary details.

See **appendix F** for the code.

2.2 Email Service

Node Mailer Email service has been bound with REST Api and Service can be used with <http://localhost:4000/sensdEmail>. Message and the receiver’s email should be send in the message body. Email service has been used for sending emails to users when an emergency such as CO2 level or Smoke level rises above 5.occurred. Email service is used by RMI server.

Request Type – POST

See **appendix G** for the code

2.3 Email Service

TeleSign SDK service has been bound with REST Api and Service can be used with [http://localhost:4000/sensSMS](http://localhost:4000/sensSMSl). Receiver’s mobile number should be send in the message body. SMS service has been used for sending SMS to users when an emergency occurred such as CO2 level or Smoke level rises above 5. SMS service is used by RMI server.

Request Type – POST

See **appendix H** for the code

2.4 RMI Server

RMI server is connected with REST Api and Desktop client to communicate between REST Api and Desktop client. RMI server gets sensor details from REST Api every 15 seconds. When a sensor exceeds the CO2 level or Smoke level above five server gets the user details of the location and send an email and a SMS to the user. RMI Server sends sensor details every 30 seconds upon the client request.RMI Server uses java swing timer to achieve this.

This is the code snippet which used by server to register in registry and bind temperature\_Sensor\_impl object to Registry

Temperature\_Sensor\_Impl obj = new Temperature\_Sensor\_Impl();

//Registering in the registry

Registry registry = LocateRegistry.getRegistry();

//Bind object to the registry

registry.bind("sensor", obj);

See **appendix I**  for the code

2.4.1 RMI\_Sensor\_Interface

This interface has all the remote methods which are used by the remote desktop client.

See **appendix J** for the code

2.4.2 RMI\_Sensor\_Impl

This is the implementation of the RMI\_Sensor\_Interface. This class creates an object of each class of the following and call the methods and return to the Desktop Client.

See **appendix K f**or the code

HTTP\_GET class has the method for getting all the sensors from the REST Api. . This method communicate with the REST Api to get all the sensors from the database

HTTP\_GET\_ONE class has the method for getting one sensor with the given Id from the REST. . This method communicate with the REST Api to get a sensor from the database

HTTP\_POST class has the method for adding sensor details. This method communicate with the REST Api to add a sensor to the database.

HTTP\_PATCH class has the method for Updating one sensor with the given Id. This method communicate with the REST Api to update a sensor on the database

HTTP\_DELETE class has the method for deleting one sensor with the given Id. This method communicate with the REST Api to delete a sensor from the database

Send\_Email class has the Http post method to send an post request to send an email to the users.

Send\_SMS class has the Http post method to send an post request to send an SMS to the users.

Get\_User\_Details has the get request to get user details. This method is connected with the REST Api to get details from the database.

See **appendix L** for the code

2.5 Desktop Client

Desktop client has been used java swing for user interfaces and apache HTTP library to manage HTTP requests.

Desktop client gets sensor details from the RMI server every thirty seconds. From the desktop client RMI server is accessed through a Remote Object. Methods to add, edit, get, update sensors from the RMI Server has been implemented. If the CO2 level or Smoke level rises above five senor is displayed in Red color. If an sensor is inactive sensor data is displayed zero. Desktop client has an admin panel to add, update, delete sensors. Admin should use valid credentials to login.

Desktop client use the remote object and use the methods which are implemented in sensor\_implemetation class. Following code snippet is used whenever the client needs to use remote object.

Temperature\_Sensor\_Interface obj = (Temperature\_Sensor\_Interface) Naming

.*lookup*("//localhost/sensor");

See **appendix M** for the code

2.6 Web Client

Web client was developed using React which a javascript library. Axios library has been used to send asynchronous http requests.

Web client gets sensor details and updates the details every ten seconds from the REST Api. If the CO2 level or Smoke level rises above five sensor is Red colored. Inactive sensors are displayed as inactive and levels are zero

See **appendix N** for the code

2.7 Sample sensor application

This application simulates the behavior of sensors and send random numbers between one to ten to the database through REST Api. This application is developed in react.

See **appendix O** for the code

2.8 Database

Mongo DB has been used to store data since since it is easy work with node js and mongo DB together. Atlas cluster has been used to save data. REST Api is connected with Atlas Cluster so there is no more configuration needed.

**Appendices**

Appendix A

// create the endpoint(URL) for get all sensor details

router.get("/getAllSensors", async (req, res, next) => {

try {

// find all sensors in the db and wait for the response

const response = await Sensors.find();

// send a json response

res.json(response);

} catch (e) {

console.log(e);

}

});

Appendix B

// create the endpoint(URL) for update sensor details

router.patch("/updateSensor/:id", async (req, res, next) => {

// pass a id as the request param

const newId = req.body.floorNo + req.body.roomNo; // change id of the sensor

try {

const updatedSensor = await Sensors.updateOne(

// finds the first document that matches the filter and applies the specified update modifications.

{id: req.params.id}, // checks the id of the sensor

{id: req.params.id}, // checks the id of the sensor

{

$set: {

// set values in the sensor using request body

id: newId,

active: req.body.active,

floorNo: req.body.floorNo,

roomNo: req.body.roomNo

},

}

);

res.json(updatedSensor); // send the json response

} catch (e) {

console.log(e); // handle errors

}

});

Appendix C

router.delete("/deleteSensor/:id", async (req, res, next) => {

try {

const response = await Sensors.deleteOne({id: req.params.id}); // Remove a single document from the collection based on a query filter.

res.json(response); // send a json response

} catch (e) {

console.log(e); // handle errors

}

});

Appendix D

// create the endpoint(URL) for add sensor to the mongodb database

router.post("/addSensor", async (req, res, next) => {

// pass a asynchronous function

let {active, floorNo, roomNo, smokeLevel, co2Level} = req.body; // extract the request body data to variables(ES6)

let count = floorNo + roomNo; // create an id for the sensor

let data = {

// store body request data into data variable and set id to the count value

id: count,

active: active,

floorNo: floorNo,

roomNo: roomNo,

smokeLevel: smokeLevel,

co2Level: co2Level,

};

try {

const response = await Sensors.create(data); // call the mongodb create method and wait for the completion

res.send(

JSON.stringify({success: "sensor added", code: "reg", sensor: response})

); // send response to user

} catch (e) {

console.log(e); // handle errors

}

});

Appendix E

router.post("/addRoomDetails", async (req, res, next) => {

let {floorNo, roomNo, customerPhone, customerMail} = req.body; // extract the request body data to variables(ES6)

let count = floorNo + roomNo; // create an id for the room

let data = {

// store body request data into data variable and set id to the count value

id: count,

floorNo: floorNo,

roomNo: roomNo,

customerPhone: customerPhone,

customerMail: customerMail,

};

try {

const response = await RoomDetails.create(data); // create a new room in db

res.send(

JSON.stringify({success: "sensor added", code: "reg", room: response})

); // send json response

} catch (e) {

console.log(e);

}

});

Appendix F

router.get("/getRoomDetails/:id", async (req, res, next) => {

try {

const response = await RoomDetails.findOne({id: req.params.id});

res.json(response); // send the json response

} catch (e) {

console.log(e);

}

});

Appendix G

router.post("/sendEmail", async (req, res, next) => {

const receiverEmail = req.body.receiverEmail; // get the reciver email address from body of the request

const senderMail = "rashmikammg@gmail.com"; // set emailmaddress of sender

const password = "Rashmika@123"; // set password of sender

// allow less secure feature on in chrome

// link - https://myaccount.google.com/lesssecureapps

// use node mailer module for send mails

try {

let transporter = nodemailer.createTransport({

service: "gmail", // use gmail as the email service

port: 25, // port number

secure: false, // true for 465, false for other ports

auth: {

user: senderMail,

pass: password,

},

tls: {

rejectUnauthorized: false,

},

});

let HelperOptions = {

from: senderMail, // sender address

to: receiverEmail, // list of receivers

subject: "Warning Message", // Subject line

text: "", // plain text body

html: `

<h4>Fire Alert!!!</h4>

<p>\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*</p>

<p>${req.body.location} is on Fire!!!</p>

<p>Take necessary actions!!!</p>

<p>Call 110 Fire & Rescue</p>

<p>\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*</p>

`,

};

HTML version of the message

transporter.sendMail(HelperOptions, (error, info) => {

// send mail with defined transport object

if (error) {

return console.log(error);

}

console.log("The message was sent!");

res.json(info); // send the json response

});

} catch (e) {

console.log(e);

}

});

Appendix H

router.post("/sendSMS", async (req, res, next) => {

/\*

for sending sms , use TeleSignSdk package (module) and use its trial account

\*/

const receiverPhoneNo = req.body.phoneNo; // get the reciever's phone no in request body to a variav

try {

const customerId = "67D18FB0-CB55-4314-89AE-6B5493C10B65";

const apiKey = "jhskmQo+4k9Y1BQpUARdzftA8prXNDMzwXMdFMNRFFdKGbB11zP+6u172GEeyrmj3azqCoTDau/N9q98LK1TSw

/\*

customerID and APIKEYS are auto generated by telesign website - credentials to use this service

(authentication details)

\*/

const rest\_endpoint = "https://rest-api.telesign.com";

const timeout = 10 \* 1000; // 10 secs

const client = new TeleSignSDK(

customerId, // create a object with these properties

apiKey,

rest\_endpoint,

timeout // optional

// userAgent

);

const message = "Fire Alert! Take an action!"; // message

const messageType = "ARN"; // ARN = Alerts, Reminders, and Notifications; OTP = One time password; MKT = Marketing

console.log("## MessagingClient.message ##");

function messageCallback(error, responseBody) {

/\*

this callback function executed after clinet.sms.message() function has finished executing

\*/

if (error === null) {

// check message sent successfully

console.log(

`Messaging response for messaging phone number: ${receiverPhoneNo}` +

` => code: ${responseBody["status"]["code"]}` +

`, description: ${responseBody["status"]["description"]}`

);

res.json("send sms successfully"); // send json response

} else {

console.error("Unable to send message. " + error); // log the errors

}

}

client.sms.message(messageCallback, receiverPhoneNo, message, messageType);

/\*

call the sms.message() method in TeleSignSDK client object with a callback

send message to given user phone no

\*/

} catch (e) {

console.log(e);

}

});

Appendix I

public class Server {

public static void main(String[] args) throws RemoteException, MalformedURLException, AlreadyBoundException {

Temperature\_Sensor\_Impl obj = new Temperature\_Sensor\_Impl();

//Registering in the registry

Registry registry = LocateRegistry.getRegistry();

//Bind object to the registry

registry.bind("sensor", obj);

System.out.println("Server Started");

//Timer object to schedule method call every five seconds

Timer t= new Timer(0,null);

t.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

try {

//Server gets sensors every fifteen seconds

HTTP\_GET http\_get = new HTTP\_GET();

List<Sensor> senosrs = http\_get.GET();

for (Sensor s : senosrs) {

System.out.println(s.get\_id());

if(s.getCo2Level()>5 || s.getSmokeLevel()>5) {

//Getting user details of the relevant floor and room

GetUserDetails getUserDetails = new GetUserDetails();

UserDetails userDetails = getUserDetails.GetUserDetails(s.getFloorNo()+s.getRoomNo());

//This will be displayed in emails as the location to display in Email

String location = "Floor : " + s.getFloorNo() + " Room : " + s.getRoomNo();

System.out.println(location);

//Creating SendEmail object and call send email method

SendEmail sendEmail = new SendEmail();

sendEmail.sendEmail(userDetails.getCustomerMail(),location, s.getCo2Level(), s.getSmokeLevel());

//Creating SendSMS object and call sendSMS method

SendSMS sendSMS = new SendSMS();

sendSMS.sendSMS(userDetails.getCustomerPhone());

}

}

} catch (Exception ex) {

ex.printStackTrace();

}

}

});

t.setRepeats(true);

//Set timer for fifteen seconds

t.setDelay(15000);

t.start();

}

}

Appendix J

import java.rmi.Remote;

import java.util.List;

import http\_methods.Sensor;

import sms\_email\_services.UserDetails;

public interface Temperature\_Sensor\_Interface extends Remote {

public void addSensor(boolean active,String floorNo,String roomNo,int co2Level,int smokeLevel) throws Exception;

public void editSensor(String updateID,boolean active,String floorNo,String roomNo,int co2Level,int smokeLevel) throws Exception;

public List<Sensor> getSensor() throws Exception;

public void deleteSensor(String sensorID) throws Exception;

public Sensor getOneSensorByID(String ID) throws Exception;

}

Appendix K

public class Temperature\_Sensor\_Impl extends UnicastRemoteObject implements Temperature\_Sensor\_Interface {

private static final long serialVersionUID = 1L;

protected Temperature\_Sensor\_Impl() throws RemoteException {

super();

}

// Method to add new Sensors

public void addSensor(boolean active, String floorNo, String roomNo, int co2Level, int smokeLevel) throws Exception {

// Creating a object of HTTP\_POST and call post method with parameters

HTTP\_POST httpPost = new HTTP\_POST();

httpPost.POST(active, floorNo, roomNo, co2Level, smokeLevel);

}

// Method to edit Sensors

public void editSensor(String updateID, boolean active, String floorNo, String roomNo, int co2Level, int smokeLevel)throws Exception {

// Creating a object of HTTP\_PATCH and call patch method with parameters

HTTP\_PATCH httpPatch = new HTTP\_PATCH();

httpPatch.UPDATE(updateID, active, floorNo, roomNo, co2Level, smokeLevel);

}

// Method to get all Sensors

public List<Sensor> getSensor() throws Exception {

// Calling the method in HTTP\_GET class

HTTP\_GET httpGet = new HTTP\_GET();

// Storing return values in a list

List<Sensor> sensors = httpGet.GET();

return sensors;

}

// Method to delete one Sensor with given ID

public void deleteSensor(String sensorID) throws Exception {

// Calling delete in HTTP\_DELETE with ID

HTTP\_DELETE httpDelete = new HTTP\_DELETE();

httpDelete.DELETE(sensorID);

}

// Method to get one Sensor with given ID

public Sensor getOneSensorByID(String ID) throws Exception {

// Calling GET\_ONE in HTTP\_GET\_ONE with ID

HTTP\_GET\_ONE getOne = new HTTP\_GET\_ONE();

Sensor sensor = getOne.GET(ID);

// Return the sensor

return sensor;

}

}

Appendix L

public class HTTP\_GET {

//REST api URL to get all sensors

private final String URL = "http://localhost:4000/getAllSensors";

public List<Sensor> sensors;

public List<Sensor> GET() throws IOException, InterruptedException {

System.out.println("called");

// Building HttpClient

HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder().GET().header("accept", "application/json").uri(URI.create(URL))

.build();

HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());

//Object mapper to map string with sensor object

ObjectMapper mapper = new ObjectMapper();

// Getting json String and map it Sensor class and generate sensor objects List

sensors = mapper.readValue(response.body(), new TypeReference<List<Sensor>>() {

});

//Return sensors

return sensors;

}

}

public class HTTP\_GET\_ONE {

Sensor sensor = new Sensor();

public Sensor GET(String ID) throws IOException, InterruptedException {

// api get one user URL + ID

String URL = "http://localhost:4000/getSensor/" + ID;

// Building HttpClient

HttpClient client = HttpClient.newHttpClient();

// Crating http Get request and saving the response

HttpRequest request = HttpRequest.newBuilder().GET().header("accept", "application/json").uri(URI.create(URL))

.build();

HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());

// Getting json String and map it Sensor class and generate sensor object

ObjectMapper mapper = new ObjectMapper();

this.sensor = mapper.readValue(response.body(), Sensor.class);

System.out.println(this.sensor);

return sensor;

}

}

public class HTTP\_POST {

public void POST(boolean active, String floorNo, String roomNo, int co2Level, int smokeLevel) throws ClientProtocolException, IOException {

// Building CloseableHttpClient

JSONObject json = new JSONObject();

json.put("active", active);

json.put("floorNo", floorNo);

json.put("roomNo", roomNo);

json.put("co2Level", 0);

json.put("smokeLevel", 0);

// Building CloseableHttpClient

CloseableHttpClient httpClient = HttpClientBuilder.create().build();

//Sending a Http Post request to REST api URL to Add sensor

HttpPost request = new HttpPost("http://localhost:4000/addSensor");

//Converting json object to string and add it to parameters

StringEntity params = new StringEntity(json.toString());

request.addHeader("content-type", "application/json");

request.setEntity(params);

//Execute the request and storing the response

HttpResponse response = httpClient.execute(request);

//Printing status

System.out.println(response.getStatusLine().toString());

}

}

public class HTTP\_PATCH {

public void UPDATE(String updateID,boolean active, String floorNo, String roomNo) throws ClientProtocolException, IOException {

//Creating json object and add all the values into it

JSONObject json = new JSONObject();

json.put("active", active);

json.put("floorNo", floorNo);

json.put("roomNo", roomNo);

// Building CloseableHttpClient

CloseableHttpClient httpClient = HttpClientBuilder.create().build();

//Sending a Http patch request to REST api URL to update sensors

HttpPatch request = new HttpPatch("http://localhost:4000/updateSensor/"+updateID);

//Converting json object to string and add it to parameters

StringEntity params = new StringEntity(json.toString());

request.addHeader("content-type", "application/json");

request.setEntity(params);

//Execute the request and storing the response

HttpResponse response = httpClient.execute(request);

//Printing status

System.out.println(response.getStatusLine().toString());

}

}

public class HTTP\_DELETE {

public void DELETE(String sensorID) {

Thread newThread = new Thread(() -> {

try {

CloseableHttpClient httpClient = HttpClientBuilder.create().build();

try {

// creating a http Delete Request

HttpDelete request = new HttpDelete("http://localhost:4000/deleteSensor/" + sensorID);

request.addHeader("content-type", "application/json");

// Execute the request and storing response in HttpResponse object

HttpResponse response = httpClient.execute(request);

} catch (Exception ex) {

System.out.println(ex);

} finally {

try {

// Closing the Client

httpClient.close();

} catch (IOException e) {

e.printStackTrace();

}

}

} catch (Exception e) {

e.printStackTrace();

}

});

newThread.start();

}

}

Appendix M

btnAdd.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

String floorNumber, roomNumber;

try {

// If input fields are empty display an error

if (floorNoTextField.getText().isBlank() || roomNoTextField.getText().isBlank()) {

JFrame f = new JFrame();

JOptionPane.showMessageDialog(f, "Empty Fields");

}

else {

// Getting input fields and assigning them to variables

floorNumber = floorNoTextField.getText();

roomNumber = roomNoTextField.getText();

// Getting remote object

Temperature\_Sensor\_Interface obj = (Temperature\_Sensor\_Interface) Naming

.lookup("//localhost/sensor");

// Calling add sensor method in remote object

obj.addSensor(active, floorNumber, roomNumber, 0, 0);

// Displaying added Dialog

JFrame f = new JFrame();

JOptionPane.showMessageDialog(f, "Sensor Added");

// Setting input fields empty

floorNoTextField.setText("");

roomNoTextField.setText("");

}

} catch (Exception ex) {

ex.printStackTrace();

}

}

});

search.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

// Getting values from textfields

floorNo = floorNoFind.getText();

roomNo = roomNoFind.getText();

try {

// Creating remote object

Temperature\_Sensor\_Interface obj = (Temperature\_Sensor\_Interface) Naming

.lookup("//localhost/sensor");

// call getOneSensorByID and store there the return object

Sensor sensor = obj.getOneSensorByID(floorNo + roomNo);

// Setting values to display to edit

floorNoEdit.setText(sensor.getFloorNo());

roomNoEdit.setText(sensor.getRoomNo());

//Setting fields empty

floorNoFind.setText("");

roomNoFind.setText("");

if (sensor.isActive()) {

activeSelect.setSelected(true);

} else {

activeSelect.setSelected(false);

}

} catch (Exception e1) {

JFrame f = new JFrame();

// if Null exception occured Dialog message will be displayed

JOptionPane.showMessageDialog(f, "No Sensor Found");

}

}

});

public void actionPerformed(ActionEvent e) {

try {

// Creating remote object

Temperature\_Sensor\_Interface obj = (Temperature\_Sensor\_Interface) Naming

.lookup("//localhost/sensor");

// Calling edit sensor methods

obj.editSensor(floorNo + roomNo, activeSelect.isSelected(), floorNoEdit.getText().toString(),

roomNoEdit.getText().toString());

activeSelect.setSelected(false);

floorNoEdit.setText("");

roomNoEdit.setText("");

// Updated success dialog

JFrame f = new JFrame();

JOptionPane.showMessageDialog(f, "Sensor Updated");

} catch (Exception ex) {

System.out.println(ex);

}

}

}

delete.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

try {

// Creating remote object

Temperature\_Sensor\_Interface obj = (Temperature\_Sensor\_Interface) Naming

.lookup("//localhost/sensor");

// Calling delete method with ID , ID = roomNo + floorNo

obj.deleteSensor(floorNoEdit.getText().toString() + roomNoEdit.getText().toString());

// Display a dialog

JFrame f = new JFrame();

JOptionPane.showMessageDialog(f, "Sensor Deleted");

floorNoEdit.setText("");

roomNoEdit.setText("");

} catch (Exception e2) {

}

}

});

public SensorComponent(String floorNo, String roomNo, int co2Level, int smokeLevel, Boolean status) {

setBorder(new LineBorder(new Color(0, 139, 139), 1, true));

// Status is Active and CO2 level or Smoke level is higher than Five

// Component is RED

if (status == true && (co2Level > 5 || smokeLevel > 5)) {

setBackground(new Color(255, 160, 122));

// If sensor is active and smoke and CO2 levels are below 6

} else {

setBackground(new Color(248, 248, 255));

}

setForeground(Color.WHITE);

setLayout(null);

String co2LevelS;

String smokeLevelS;

// if Sensor is inactive CO2 and Smoke level will be displayed as 0

if (status == false) {

co2LevelS = "0";

smokeLevelS = "0";

} else {

co2LevelS = Integer.toString(co2Level);

smokeLevelS = Integer.toString(smokeLevel);

}

String activateStatus;

// setting sensor status to display

if (status) {

activateStatus = "Activate";

} else {

activateStatus = "Inactive";

}

}

public static void displaySensors(List<Sensor> sensorList) {

frame.setBounds(100, 100, 751, 500);

//Removing previous items from panel

panel.removeAll();

//Looping through the sensor List

for (Sensor s : sensorList) {

//Creating a component

SensorComponent component = new SensorComponent(s.getFloorNo(), s.getRoomNo(), s.getCo2Level(),

s.getSmokeLevel(), s.isActive());

component.setVisible(true);

//Add component to panel

panel.add(component);

}

}

Appendix W

import React from "react";

import {

MDBRow,

MDBCol,

MDBCard,

MDBCardBody,

MDBIcon,

MDBBtn,

} from "mdbreact";

const Sensor = (props) => {

const { item } = props; // extract the props data

console.log("item", item);

/\*

this component is for get the sensor details and show them in a card view.

use conditional rendering for the changing colors of sensors.

\*/

if ((item.smokeLevel > 5 || item.co2Level > 5) && item.active === true) {

// check smokelevel or co2 level is above 5

return (

<MDBCol lg="4" md="12" className="mb-lg-0 mb-4">

<MDBCard className="red">

{" "}

{/\* set color to red \*/}

<MDBCardBody className="white-text">

<h5 className="mb-4">sensor id : {item.id}</h5>

<div className="d-flex justify-content-center">

<div className="card-circle d-flex justify-content-center align-items-center">

<MDBIcon icon="users" className="light-blue-text" />

</div>

</div>

<h2 className="font-weight-bold my-4">

smoke level : {item.smokeLevel}

</h2>

<h2 className="font-weight-bold my-4">

co2 level : {item.co2Level}

</h2>

<h5 className="font-weight-bold my-4">location</h5>

<h5 className="mb-4">

floor no : {item.floorNo} room no : {item.roomNo}

</h5>

{item.active ? (

<h5 className="font-weight-bold my-4">status : active</h5>

) : (

<h5 className="font-weight-bold my-4">status : not active</h5>

)}

</MDBCardBody>

</MDBCard>

<br></br>

</MDBCol>

);

} else if (!item.active) {

// check sensor is active or not

return (

<MDBCol lg="4" md="12" className="mb-lg-0 mb-4">

<MDBCard className="green">

{" "}

{/\* set color to green \*/}

<MDBCardBody className="white-text">

<h5 className="mb-4">sensor id : {item.id}</h5>

<div className="d-flex justify-content-center">

<div className="card-circle d-flex justify-content-center align-items-center">

<MDBIcon icon="users" className="light-blue-text" />

</div>

</div>

<h2 className="font-weight-bold my-4">smoke level : 0</h2>

<h2 className="font-weight-bold my-4">co2 level : 0</h2>

<h5 className="font-weight-bold my-4">location</h5>

<h5 className="mb-4">

floor no : {item.floorNo} room no : {item.roomNo}

</h5>

{item.active ? (

<h5 className="font-weight-bold my-4">status : active</h5>

) : (

<h5 className="font-weight-bold my-4">status : not active</h5>

)}

</MDBCardBody>

</MDBCard>

<br></br>

</MDBCol>

);

this.setState(

{

sensors: response.data, // set the sensors array with response data

},

() => {

console.log("this state ", this.state);

}

);

}, 40000); // pass 40000 mili secs (40 seconds) --> every 40 seconds send an API request

} catch (e) {

console.log(e); // log the errors

}

}

render() {

const { sensors } = this.state; // extract the sensors details

console.log("sensors", sensors);

// use a grid view for view the sensors

return (

<section className="text-center my-5">

<h2 className="h1-responsive font-weight-bold text-center my-5">

Sensor Information

</h2>

<img

src="https://media.giphy.com/media/3ohs7UjgdqCnkEYcsE/giphy.gif"

width="5%"

></img>

<br></br>

<br></br>

<MDBRow>

{sensors.map((item) => {

{

/\* traverse through the sensors array \*/

}

return (

<Sensor item={item}>

{/\* pass each arry item data to senor item component for using props \*/}

</Sensor>

);

})}

</MDBRow>

</section>

);

}

}

export default Sensors;

} else {

return (

<MDBCol lg="4" md="12" className="mb-lg-0 mb-4">

<MDBCard className="indigo">

{" "}

{/\* set color to indigo if smoke level or co2 level less than 5 and active \*/}

<MDBCardBody className="white-text">

<h5 className="mb-4">sensor id : {item.id}</h5>

<div className="d-flex justify-content-center">

<div className="card-circle d-flex justify-content-center align-items-center">

<MDBIcon icon="users" className="light-blue-text" />

</div>

</div>

<h2 className="font-weight-bold my-4">

smoke level : {item.smokeLevel}

</h2>

<h2 className="font-weight-bold my-4">

co2 level : {item.co2Level}

</h2>

<h5 className="font-weight-bold my-4">location</h5>

<h5 className="mb-4">

floor no : {item.floorNo} room no : {item.roomNo}

</h5>

{item.active ? (

<h5 className="font-weight-bold my-4">status : active</h5>

) : (

<h5 className="font-weight-bold my-4">status : not active</h5>

)}

</MDBCardBody>

</MDBCard>

<br></br>

</MDBCol>

);

}

};

export default Sensor;

import {

MDBRow,

MDBCol,

MDBCard,

MDBCardBody,

MDBIcon,

MDBBtn,

} from "mdbreact";

import Sensor from "./sensor";

const axios = require("axios"); // use axios for connect to an API

class Sensors extends Component {

constructor(props) {

super(props);

this.state = {

sensors: [], // initially keep an empty sensors array in the state

};

}

async componentDidMount() {

try {

const resInitial = await axios.get("http://localhost:4000/getAllSensors");

/\*

Initial get request calls to API and this method executes only once.

Without this method first 40 seconds not any get request called to an API

\*/

console.log("response initial", resInitial.data);

this.setState(

{

sensors: resInitial.data, // Initially set the sensors array with response data

},

() => {

console.log("this state initial ", this.state);

}

);

setInterval(async () => {

/\* The setInterval() method calls a function or evaluates an expression at specified intervals (in milliseconds).

The JS setInterval() method will keep calling the specified function until clearInterval() method is called

\*/

const response = await axios.get("http://localhost:4000/getAllSensors");

// send a request to API for get all sensor details

console.log("response every 40 secs", response.data);

Appendix O

async updateSensor1() {

try {

const smokeLevel = Math.floor(Math.random() \* 10) + 1; // get a random value for smokelevel

const co2Level = Math.floor(Math.random() \* 10) + 1; // get a random value for co2Level

const reqBody = { smokeLevel, co2Level }; // create object and store levels

const id = 11; // id of the sensor

console.log("body sensor 1", reqBody);

const response = await axios // axios POST request for update the sensor 01

.request({

method: "POST",

url: `http://localhost:4000/updateSensorOnlyLevels/${id}`,

headers: {

"Content-Type": "application/json;charset=UTF-8",

"Access-Control-Allow-Origin": "\*",

},

data: JSON.stringify(reqBody), // converts JavaScript objects into strings. When sending data to a web server the data has to be a string.

});

const resData = await response;

console.log("responsee sensor 1", resData); // log the response

} catch (e) {

console.log(e);

}

}