Introduction:

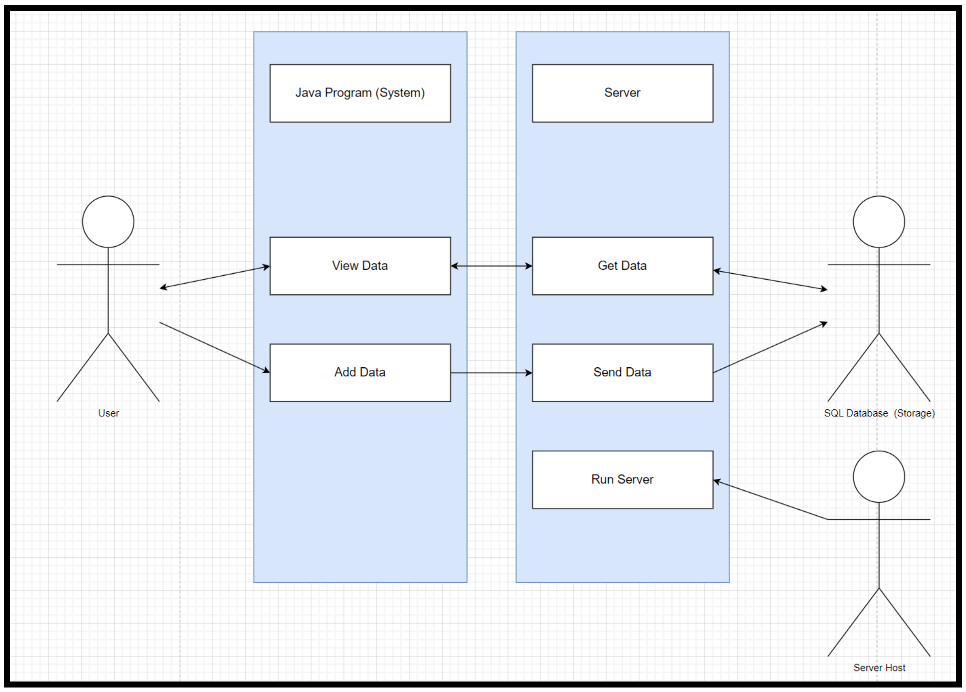
In aims to address society's goal of reducing carbon emissions, companies are employing various strategies to monitor environmental impact. By achieving this goal, they are able to support in protecting the planet, making it a more habitable environment for the future generations.

In this project we were tasked with creating a software which can be used to support companies in monitoring their carbon footprint. To achieve this, the software we plan to design will not only gather and store the information entered, but also allow information to be viewed by the user which demonstrates the impacts made to society. Information will be provided in a user-friendly way with the graphical interface making it accessible to all. In facilitating detailed evaluations of carbon emissions, the CO2 software which we develop will allow organisations to make informed decisions before implementing methods, improving the efficiency of change. These systems are vital to companies as they aim to contribute more towards sustainability and environmental efficiency.

Diagrams:

UML Diagrams:

Use-Case Diagram:



A use-case diagram is a form of UML which is able to visualise the interactions between users and a system. It assists in understanding the functionality of a system, aiding in creating understanding of what the system does and how users interact with it.

The Java program user interacts with the system using two primary methods, these include data entry and data retrieval.

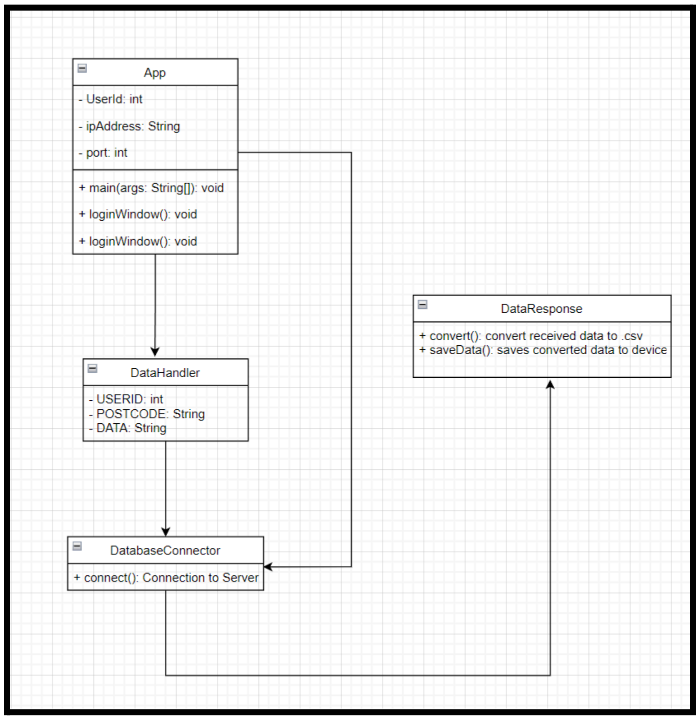
- Data Entry: This process involves users inputting data into the Java program. This then sends the information to the server to be processed and stored within the SQL database for retrieval at a later date.

- Data Retrieval: When a program user wants to view the data stored in the database. The Java program connects to the SQL database, before requesting the stored information. Once gathered, the information is then reciprocated to the Java program where it is presented to users in a readable format.

These interactions between user and the software demonstrate how connections between the user, java program, and SQL database commence whilst ensuring optimal data management and efficiency.

The last area to mention is the server host, this process is what runs the software on both ends meaning that when a user opens the java program, the SQL database goes online meaning information can be accessed.

Class Diagram:



Class diagrams are UML's which illustrate the structure of a system by displaying the classes, attributes, operations and relationships of objects. It is displayed in a blueprint format where the systems architecture can be visualised by the reader.

Our class diagram is broken into four operations, these include:

- App: The app process is the function in which users are able to interact with the server, it initializes the User Id, and Port, whilst connecting to the IP address of the local network. This then logs the user into the software where data is entered and read.

- Data Handler: The data handler section takes the code, processes it and uploads it to the SQL Database where it can be stored for latter use.

- Database Connector: This connects the java program to the server.

- Data Response: This function converts the data entered by the user into a csv format, compressing it and saving it to be used at a later date. This then allows the information to be saved as a csv file which can be read by a user without the software.

Documentation:

Development Timeline

Intro

Here we will be adding screenshots of code, console and UI throughout development along with notes relating to the current progress of the project.

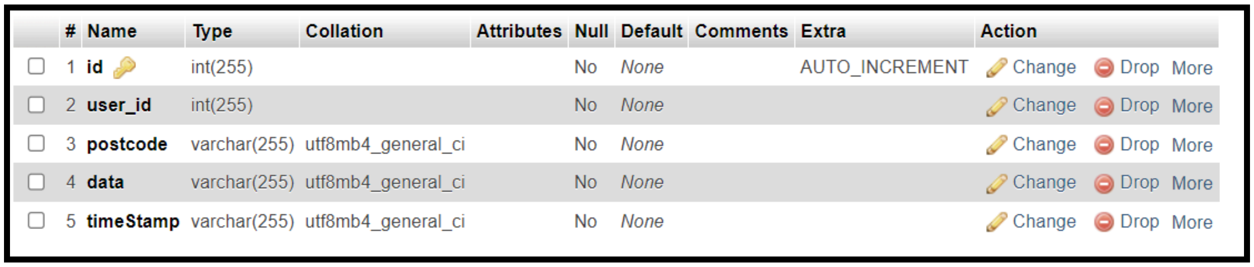
Content

Part One (pre-testing)
Our first course of action was to create a rough plan of what we would be doing for the project, here we created the UML's and wireframe for the project. We used websites and tools such as draw.io and wireframe.cc to create our pro-production documents

Next we began getting re-familiar with the java programming language, on our separate machines. Getting some basic parts of the project together and getting an idea of our separate skills and how they will fit together during development.

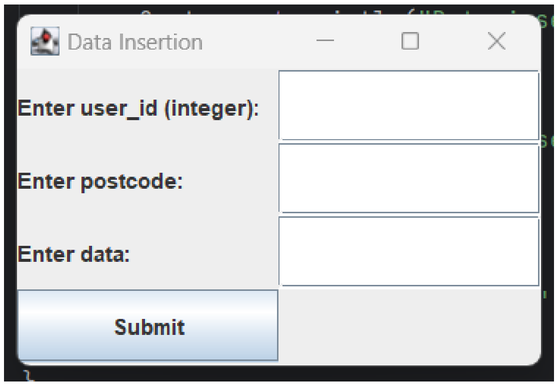
This is where we began to combine our code and created a github repository for better version control and group coding.

Created our local-hosted database using XAMPP, MYSQL and PHPMyAdmin

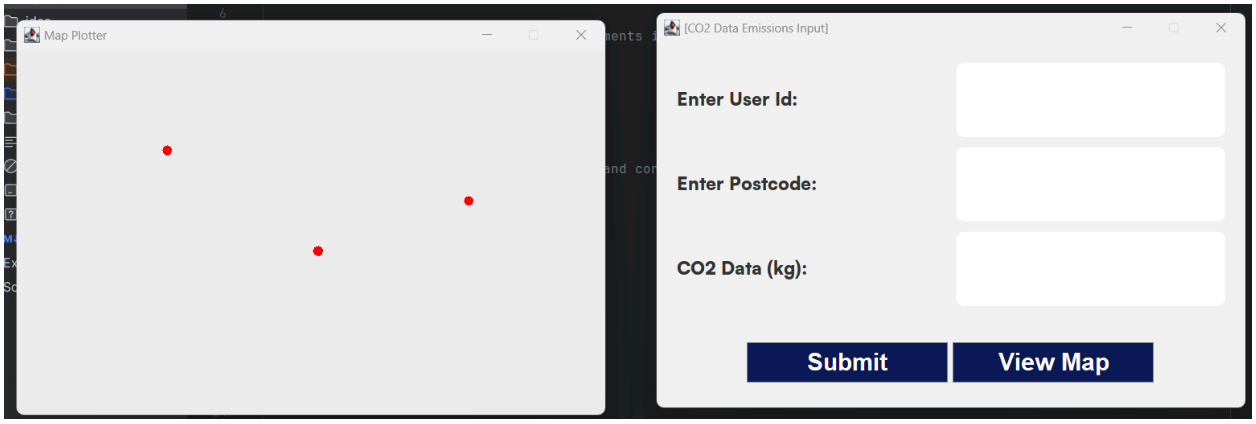


#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	id	int(255)			No	*None*		AUTO_INCREMENT	Change Drop More
2	user_id	int(255)			No	*None*			Change Drop More
3	postcode	varchar(255)	utf8mb4_general_ci		No	*None*			Change Drop More
4	data	varchar(255)	utf8mb4_general_ci		No	*None*			Change Drop More
5	timeStamp	varchar(255)	utf8mb4_general_ci		No	*None*			Change Drop More

We had the beginnings of a UI fit with user input fields and which would save the entered id, postcode and data to the database .

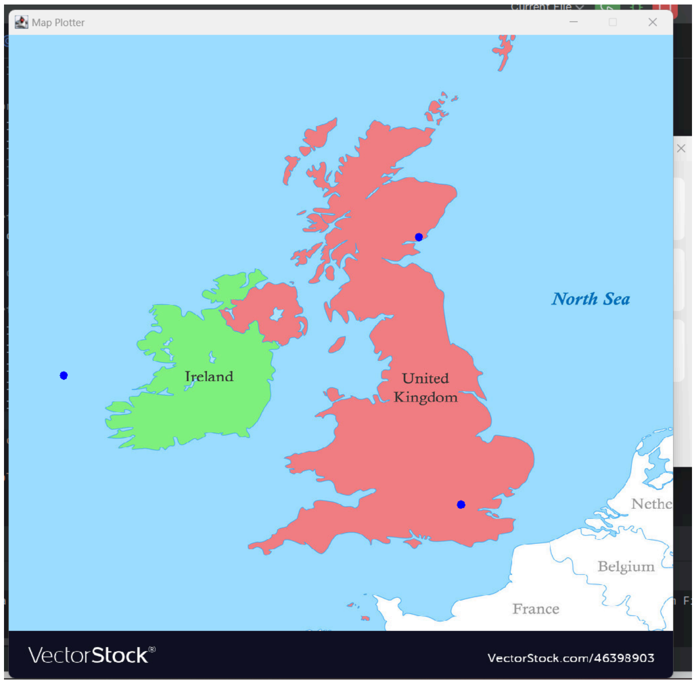


Now we were trying to add a map window to our application and got a second window to open on a button click with some plotted points at hard coded window (x,y) positions.

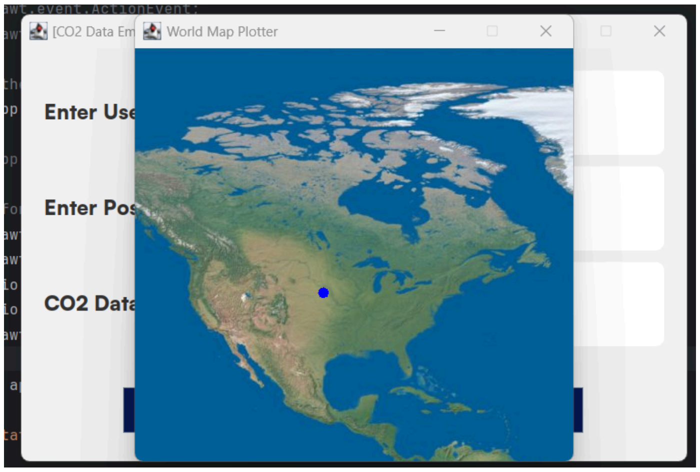


Next we wanted to be able to plot points on a map, to do this we used a placeholder image to get a rough idea of how it might work.

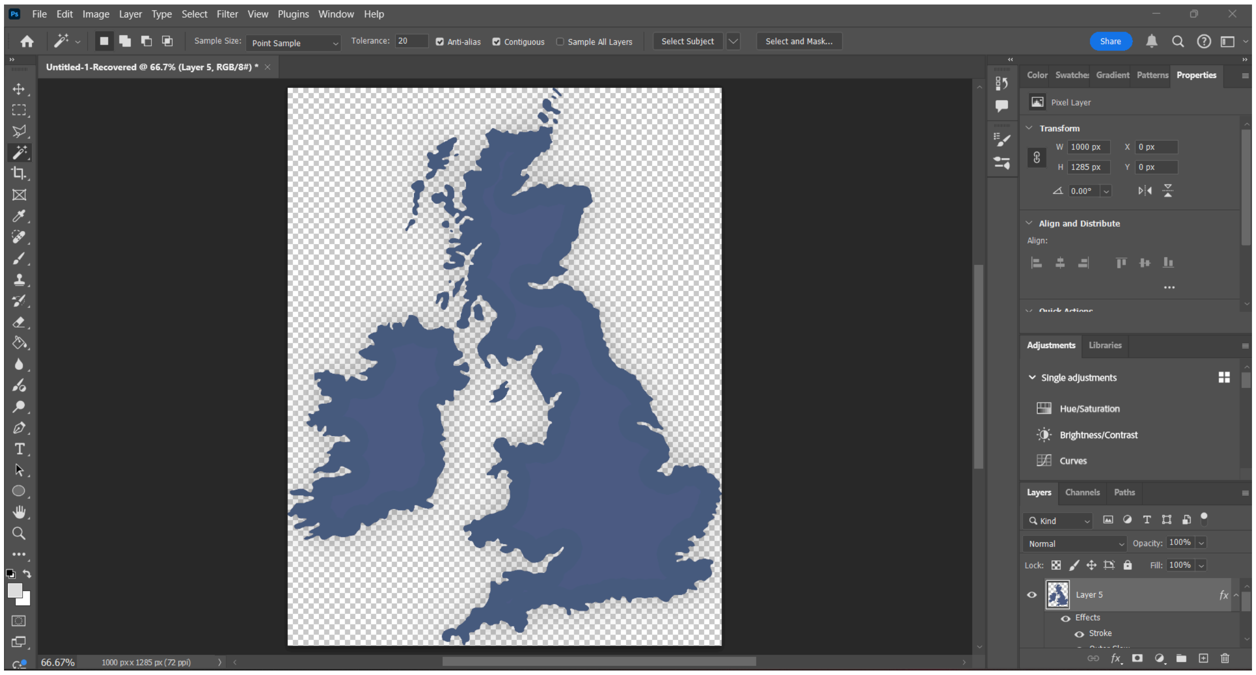
We ran into issues working out how we would be able to convert the data from the user (postcodes) into an (x,y) window position.



Using a new method of point plotting using a full world map image at a 2:1 aspect ratio and converting geographical latitude and longitude coordinates to x,y window coordinates.
This method was a success and I was able to get real world coordinates from google maps and and add them to the code and view it plotted on the map this version was centered on the US.
However we still had the issue of not being able to plot postcodes.



Taking a break from the code we created a map of the uk in photoshop in our chosen colour scheme.



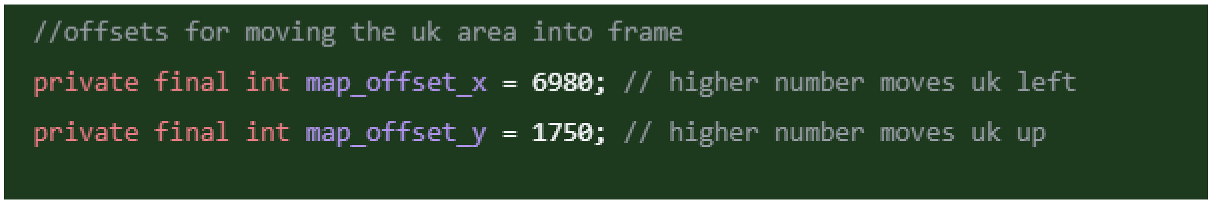
Sample Size: Point Sample Tolerance: 20 Anti-alias Contiguous Sample All Layers

W 1000 px X 0 px
H 1285 px Y 0 px
0.00°

Normal Opacity: 100%
Fill: 100%

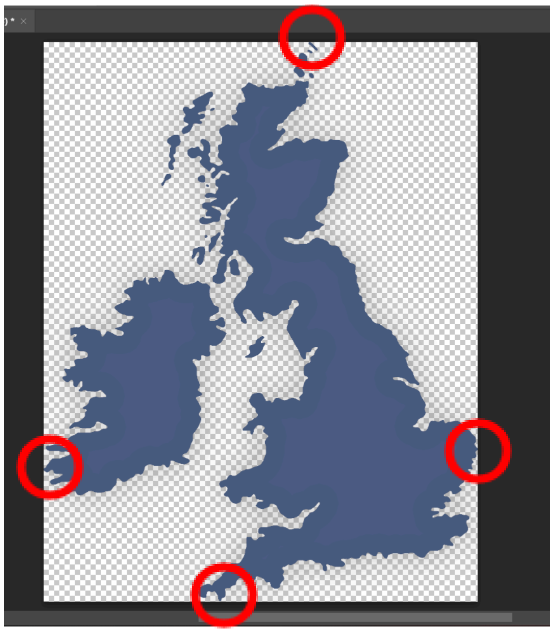
66.67% 1000 px x 1285 px (72 ppi)

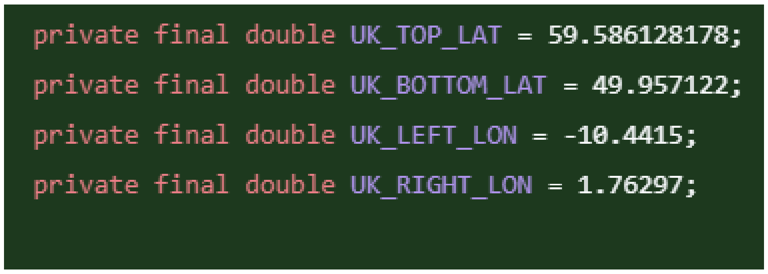
Next we needed to center the world map on the uk Centered. To do this we added x and y offsets to move the maps position in the window to show the uk. We also adjusted the zoom level to have the uk on the whole screen.



```
//offsets for moving the uk area into frame
private final int map_offset_x = 6980; // higher number moves uk left
private final int map_offset_y = 1750; // higher number moves uk up
```

Now we needed to change the image from the world map to our new uk one. To do this we used Google Maps to get the coordinates of the edge points on our uk map image that were touching the borders, (circled below) this allowed me to line up and scale my map to correctly fit within the (now deleted) world map environment I had created. For example, the south most point was lands end at “49.957122”.

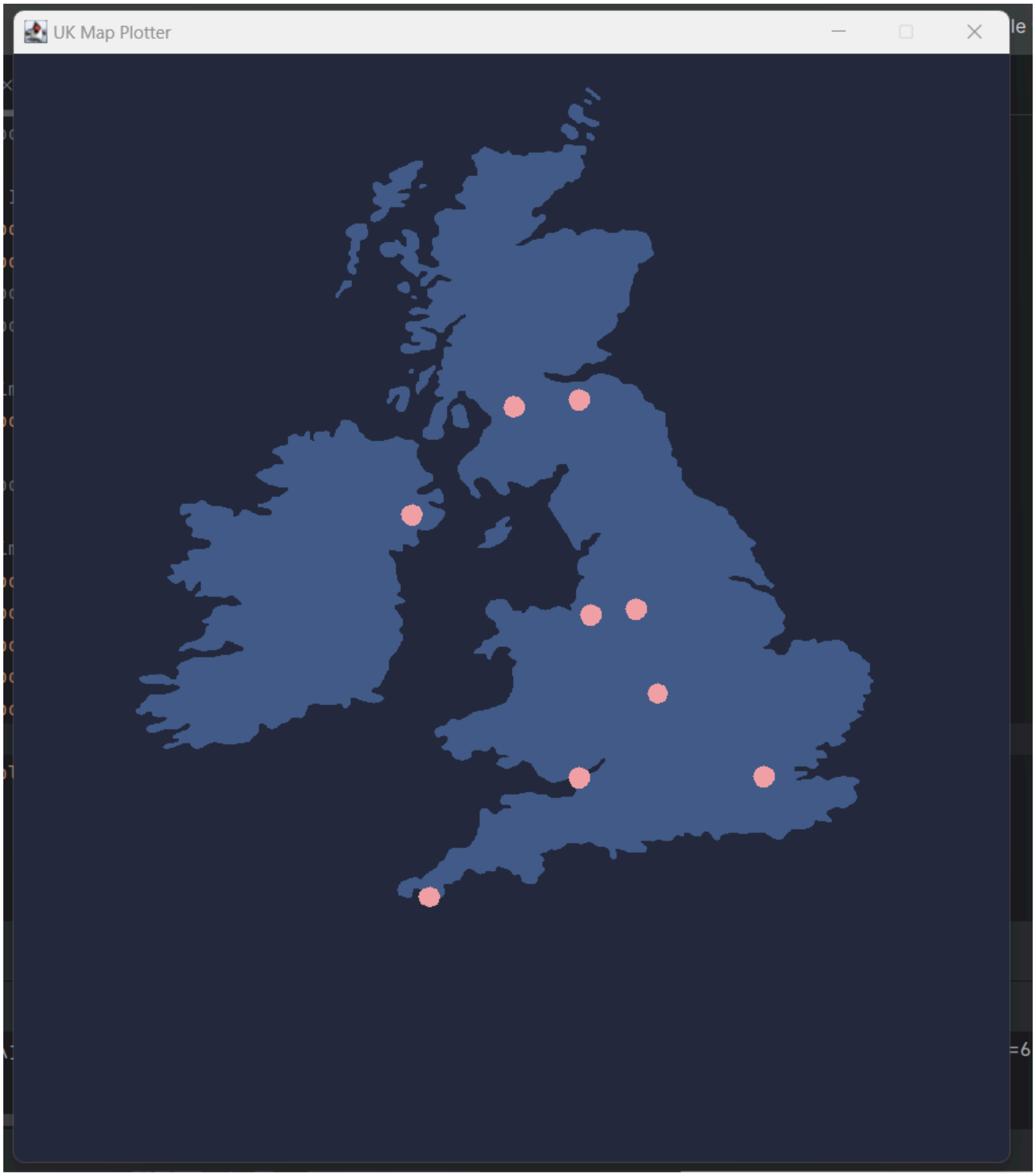




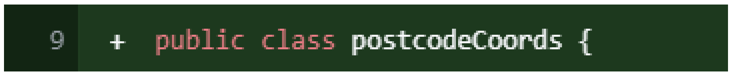
```
private final double UK_TOP_LAT = 59.586128178;
private final double UK_BOTTOM_LAT = 49.957122;
private final double UK_LEFT_LON = -10.4415;
private final double UK_RIGHT_LON = 1.76297;
```

After also changing the colour of the background and the plotted points size and colour we have our results below.

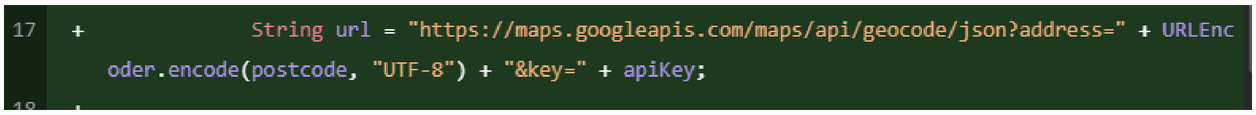
The new map:



Going back to the maps code, we found an API from google cloud that was able to convert postcodes into global coordinates (Latitude and Longitude)

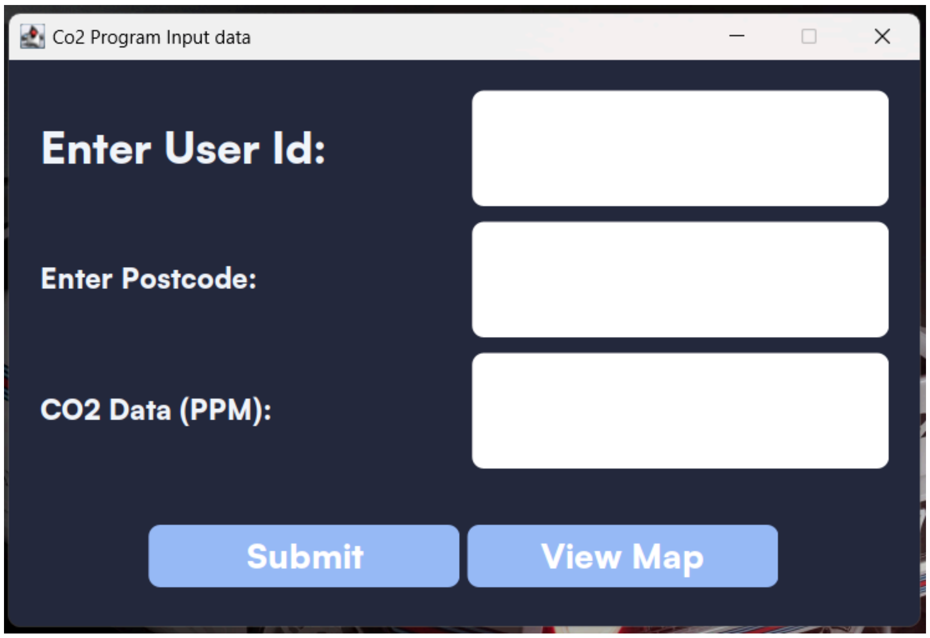


```
+ public class postcodeCoords {
```

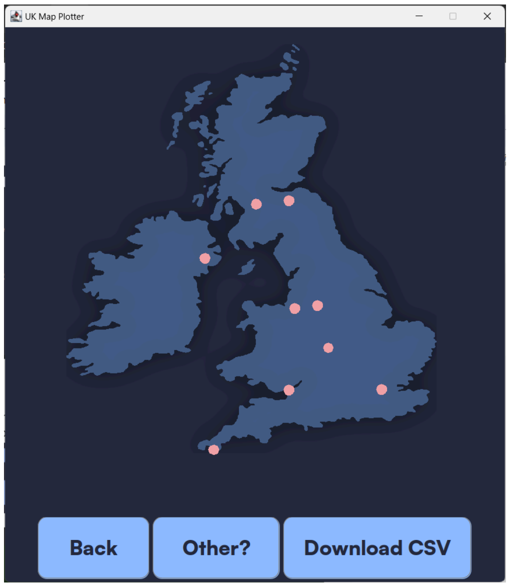


```
+             String url = "https://maps.googleapis.com/maps/api/geocode/json?address=" + URLEncoder.encode(postcode, "UTF-8") + "&key=" + apiKey;
```

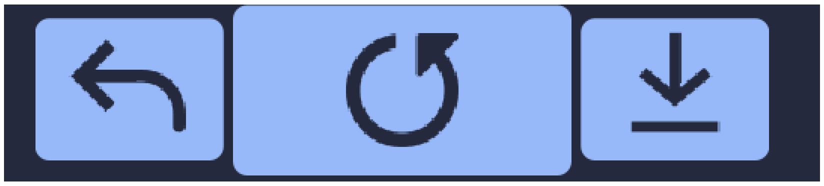
Next we changed our UI to fit the maps colours and also changed the font.



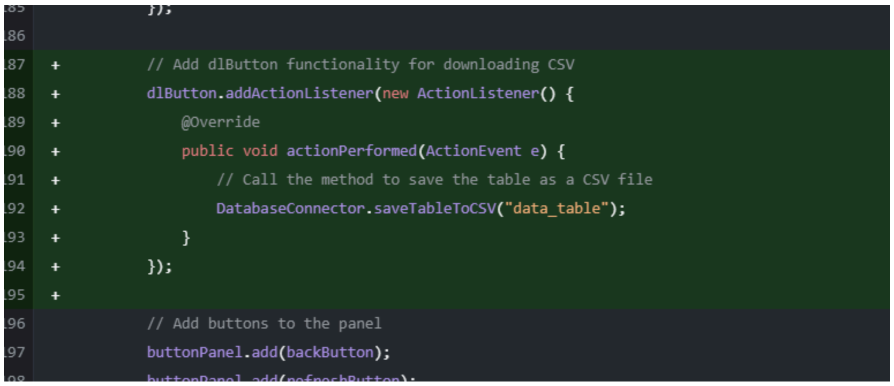
Now we needed to add functionality to download a CSV of the data from the database, so we decided the space below the map would be a good spot for a Back button and a Download button. We also knew we wanted a third button but at this point hadn't decided what that would be.



Added the refresh button, so the users can check if other users have added more readings whilst they had the map open. And also gave the buttons some new image icons.

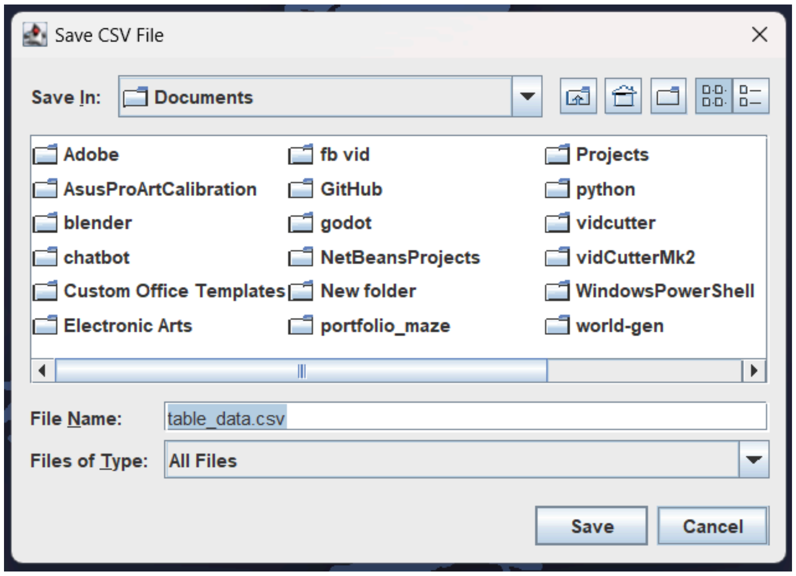


Refresh button



```
        });

+        // Add dlButton functionality for downloading CSV
+        dlButton.addActionListener(new ActionListener() {
+            @Override
+            public void actionPerformed(ActionEvent e) {
+                // Call the method to save the table as a CSV file
+                DatabaseConnector.saveTableToCSV("data_table");
+            }
+        });
+
        // Add buttons to the panel
        buttonPanel.add(backButton);
```

Download csv button


Save CSV File

Save In: Documents

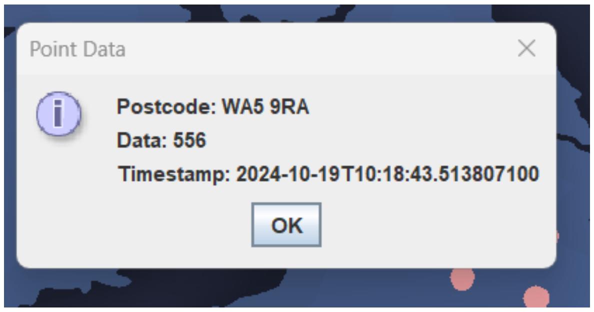
Adobe, AsusProArtCalibration, blender, chatbot, Custom Office Templates, Electronic Arts, fb vid, GitHub, godot, NetBeansProjects, New folder, portfolio_maze, Projects, python, vidcutter, vidCutterMk2, WindowsPowerShell, world-gen

File Name: table_data.csv

Files of Type: All Files

Next we wanted the plotted data points on our map to be clickable and display data for the user.

```
addMouseListener(new MouseAdapter() {
    @Override
    public void mouseClicked(MouseEvent e) {
        Point clickedPoint = e.getPoint();
        for (PointData positionData : positions) {
            if (isPointClicked(clickedPoint, positionData.getPoint())) {
```

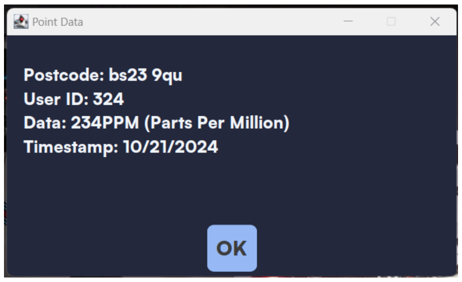

Point Data

Postcode: WA5 9RA
Data: 556
Timestamp: 2024-10-19T10:18:43.513807100

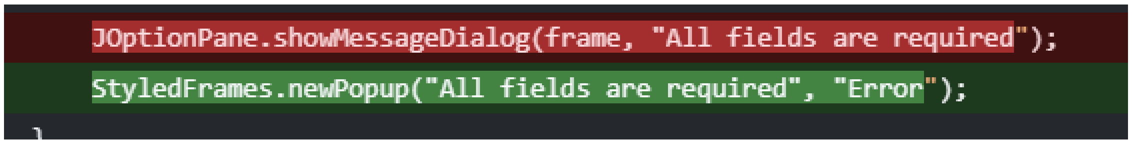
```
String message = "Postcode: " + positionData.getPostcode() + "\n"
        + "Data: " + positionData.getData() + "\n"
        + "Timestamp: " + positionData.getTimestamp();
JOptionPane.showMessageDialog(null, message, "Point Data", JOptionPan
```

Many of our popups for error handling and information used the built in JOptionPane.showMessageDialog() class, however this doesn't look very appealing and does not match our other app colours.

To solve these problems we made a separate"StyledFrames" class for our new pop up windows
The new windows

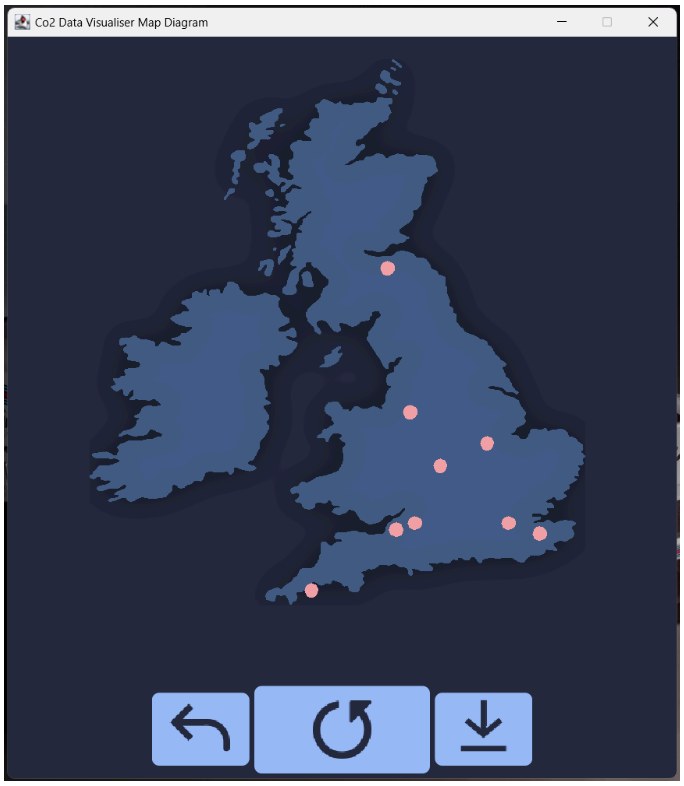

Point Data

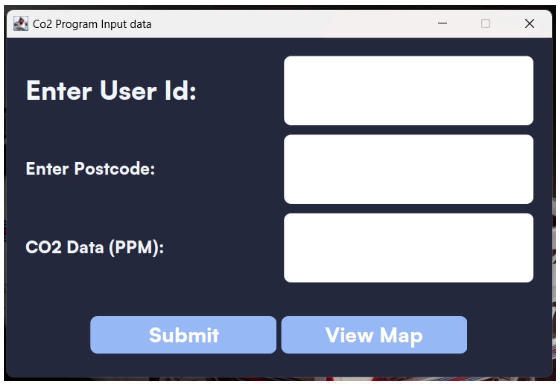
Postcode: bs23 9qu
User ID: 324
Data: 234PPM (Parts Per Million)
Timestamp: 10/21/2024



```
    JOptionPane.showMessageDialog(frame, "All fields are required");
    StyledFrames.newPopup("All fields are required", "Error");
```

At this point the application was complete and in working order, and so we were ready for our first stage of testing, Acceptance Testing.

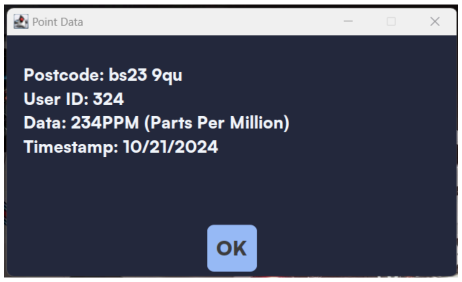



Co2 Program Input data

Enter User Id:

Enter Postcode:

CO2 Data (PPM):


Point Data

Postcode: bs23 9qu
User ID: 324
Data: 234PPM (Parts Per Million)
Timestamp: 10/21/2024

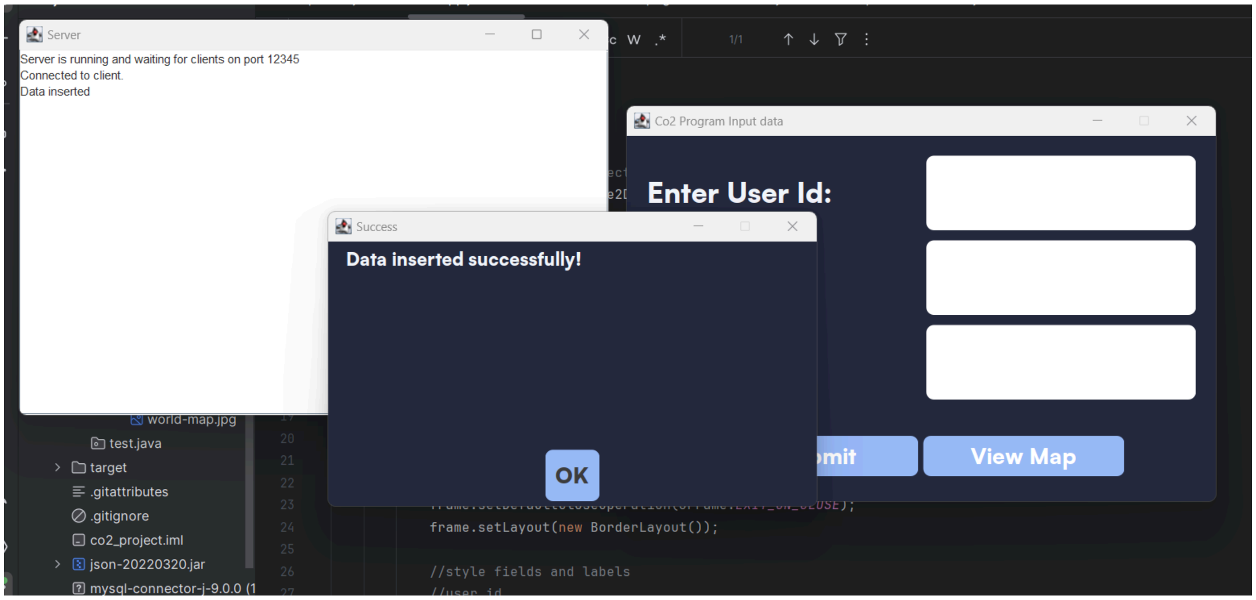
Link to a video demonstration of the application:
Co2LoggingDemonstration.mp4
https://drive.google.com/file/d/1eUxr1dobym_lbWCwfuFMrMQ3x6Qctfdi/view?usp=sharing

Acceptance Testing results:
For an in depth look at the acceptance testing you may go to the dedicated section, here will show a rough summary for the readability of the documentation.

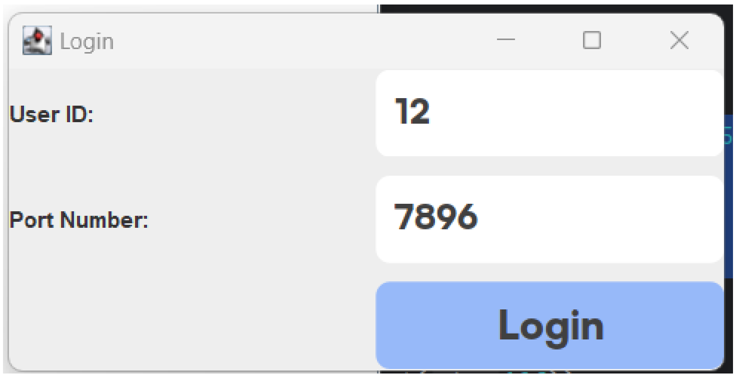
Testing results***:***
- The requirement to ***use socket programming*** not met
- Having ***“Separate server and client applications”*** not met

Next Steps:
- Add a server application that listens for clients on a set port and ip.
- Have the client app ask for an IP and port number before allowing the user to enter their ID, Postcode etc.
- Change how the data is handled by first sending it to the server application and then having the **server** add it to the database.
- Change the map windows method of retrieving data to use the server.

Part Two (Post testing)
Added a server app that will create the server for connecting.
The client connects to the server at a set ip and port


```
Server is running and waiting for clients on port 12345
Connected to client.
Data inserted
```
Data inserted successfully!
```
frame.setLayout(new BorderLayout());

//style fields and labels
```

Now we needed to allow the user to pick a port number, do do this we added “login” popup on start of the client app where the user can set their user id and port number.

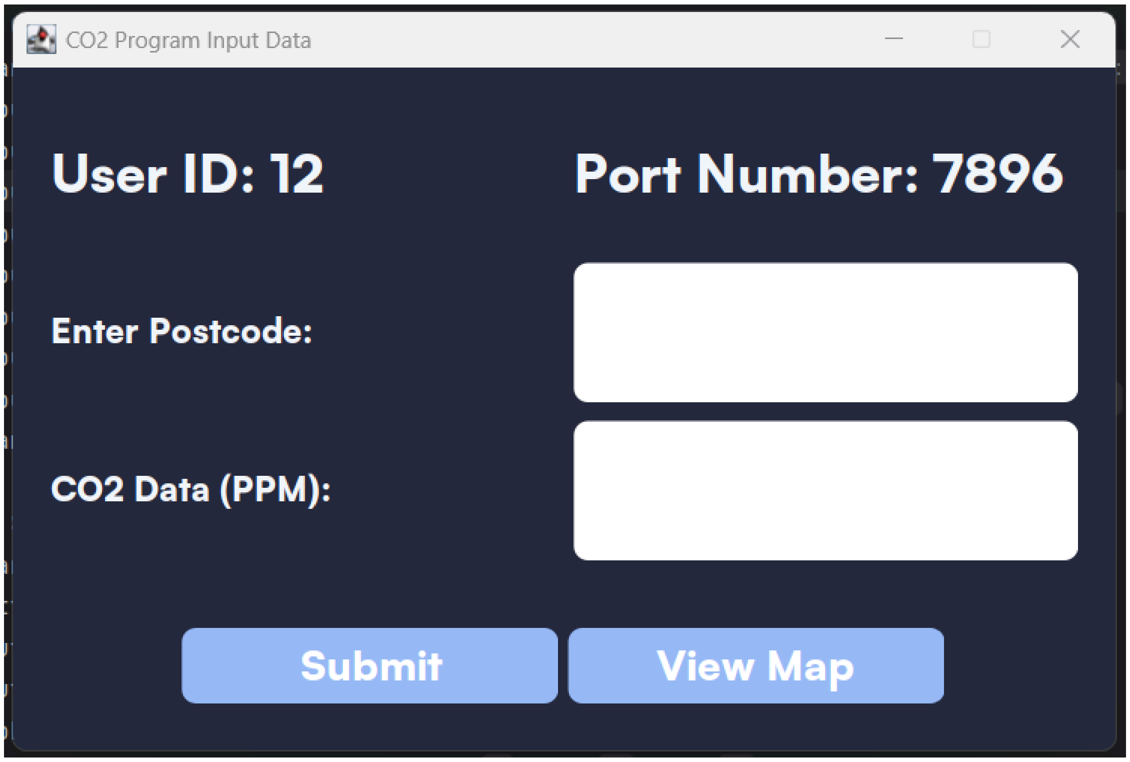


Login

User ID: 12

Port Number: 7896

We then changed the app layout to remove the now unneeded User id field and replaced it with the port information.



CO2 Program Input Data

User ID: 12

Port Number: 7896

Enter Postcode:

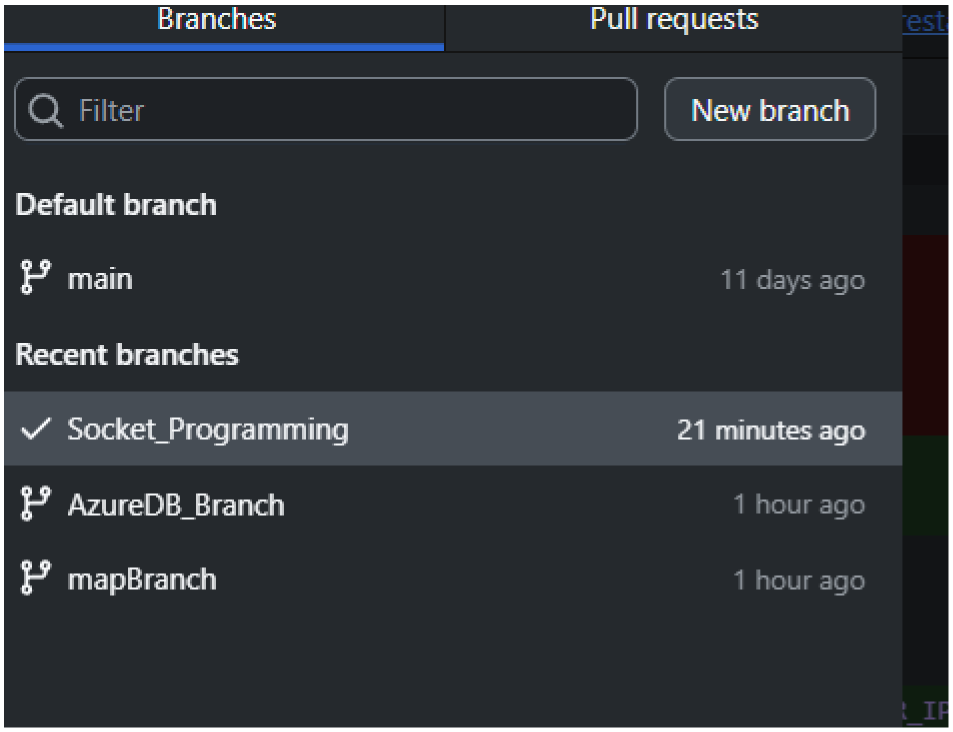
CO2 Data (PPM):

Next we needed to make the app work across multiple devices, this is so that we only need to run one server application and then anyone running a client application can connect and send data to the server, to be put in the database.

To do this we will need to also add an IP address field for the incoming clients to connect to.
The IP address entered by the client will need to be the same as the one the server is hosted on

To get started in this we created a new branch as a blank slate to work on the socket programming and connecting to their ip from other machines.

Here we plan on making a command line style app as a proof of concept before moving the code into our main branch.



Default branch

main 11 days ago

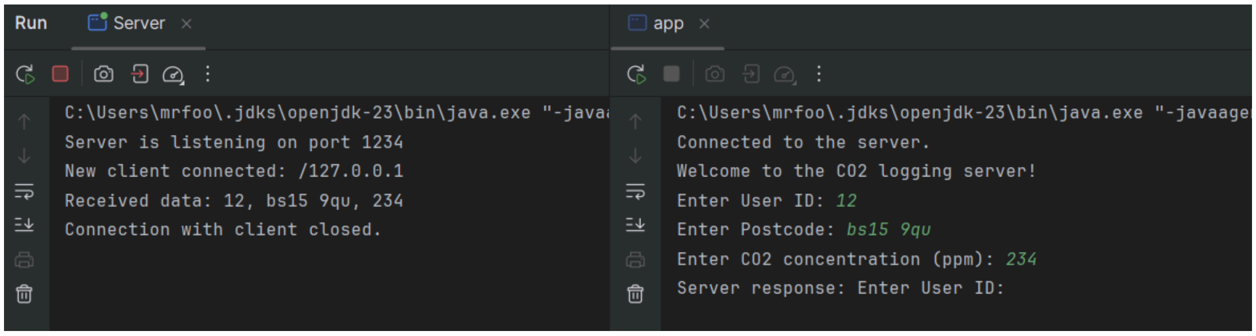
Recent branches

Socket_Programming 21 minutes ago

AzureDB_Branch 1 hour ago

mapBranch 1 hour ago

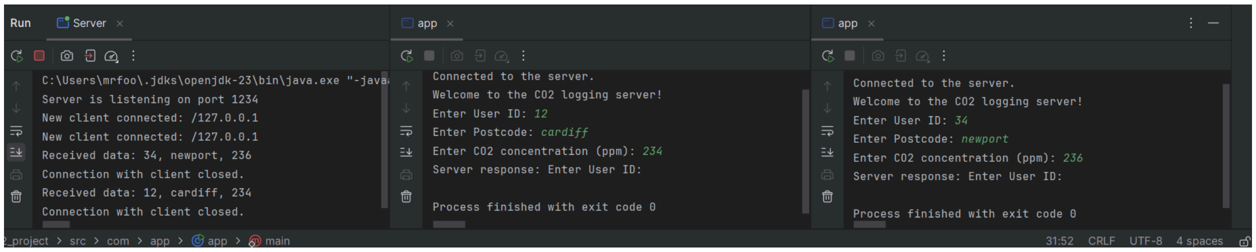
We successfully got the server app to get information from the client application.



```
C:\Users\mrfoo\.jdks\openjdk-23\bin\java.exe "-java
Server is listening on port 1234
New client connected: /127.0.0.1
Received data: 12, bs15 9qu, 234
Connection with client closed.
```

```
C:\Users\mrfoo\.jdks\openjdk-23\bin\java.exe "-javaage
Connected to the server.
Welcome to the CO2 logging server!
Enter User ID: 12
Enter Postcode: bs15 9qu
Enter CO2 concentration (ppm): 234
Server response: Enter User ID:
```

Operational with multiple users.



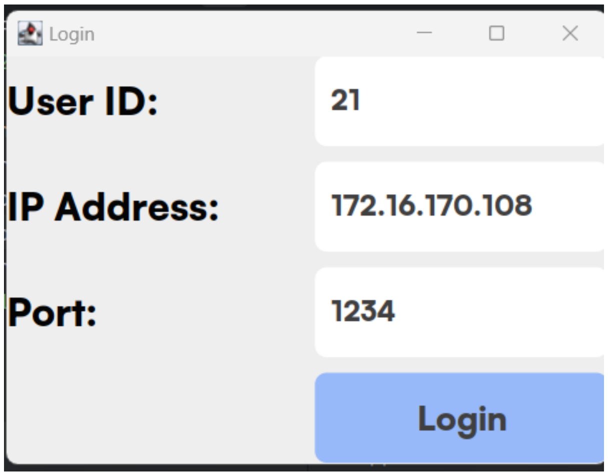
```
C:\Users\mrfoo\.jdks\openjdk-23\bin\java.exe "-java
Server is listening on port 1234
New client connected: /127.0.0.1
New client connected: /127.0.0.1
Received data: 34, newport, 236
Connection with client closed.
Received data: 12, cardiff, 234
Connection with client closed.
```

```
Connected to the server.
Welcome to the CO2 logging server!
Enter User ID: 12
Enter Postcode: cardiff
Enter CO2 concentration (ppm): 234
Server response: Enter User ID:

Process finished with exit code 0
```

```
Connected to the server.
Welcome to the CO2 logging server!
Enter User ID: 34
Enter Postcode: newport
Enter CO2 concentration (ppm): 236
Server response: Enter User ID:

Process finished with exit code 0
```

Run
Server
app
app
_project > src > com > app > app > main
31:52 CRLF UTF-8 4 spaces

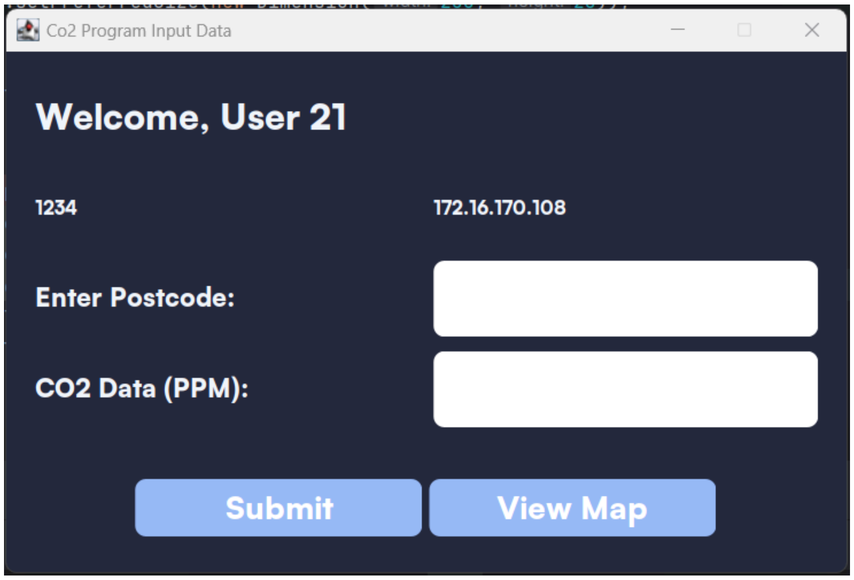
Next back on the main branch we edited the login screen to add the ip address so field and changed the font of the labels to match the rest of the application.



Login

User ID:	21
IP Address:	172.16.170.108
Port:	1234

User id, port number and ip address get taken from the login form and used for connection and saving to the database. We added a welcome message to show the user id and placed the connection information below, port and IP address.



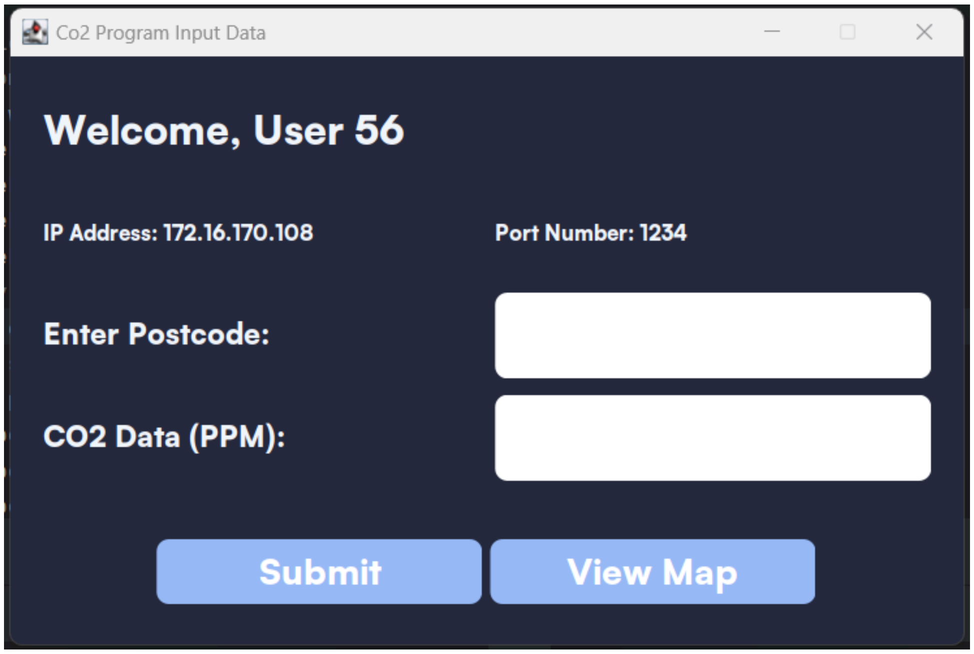
Co2 Program Input Data

Welcome, User 21

1234 172.16.170.108

Enter Postcode:	
CO2 Data (PPM):	

Swapped the order of the port number and ip address and added labels.



Co2 Program Input Data

Welcome, User 56

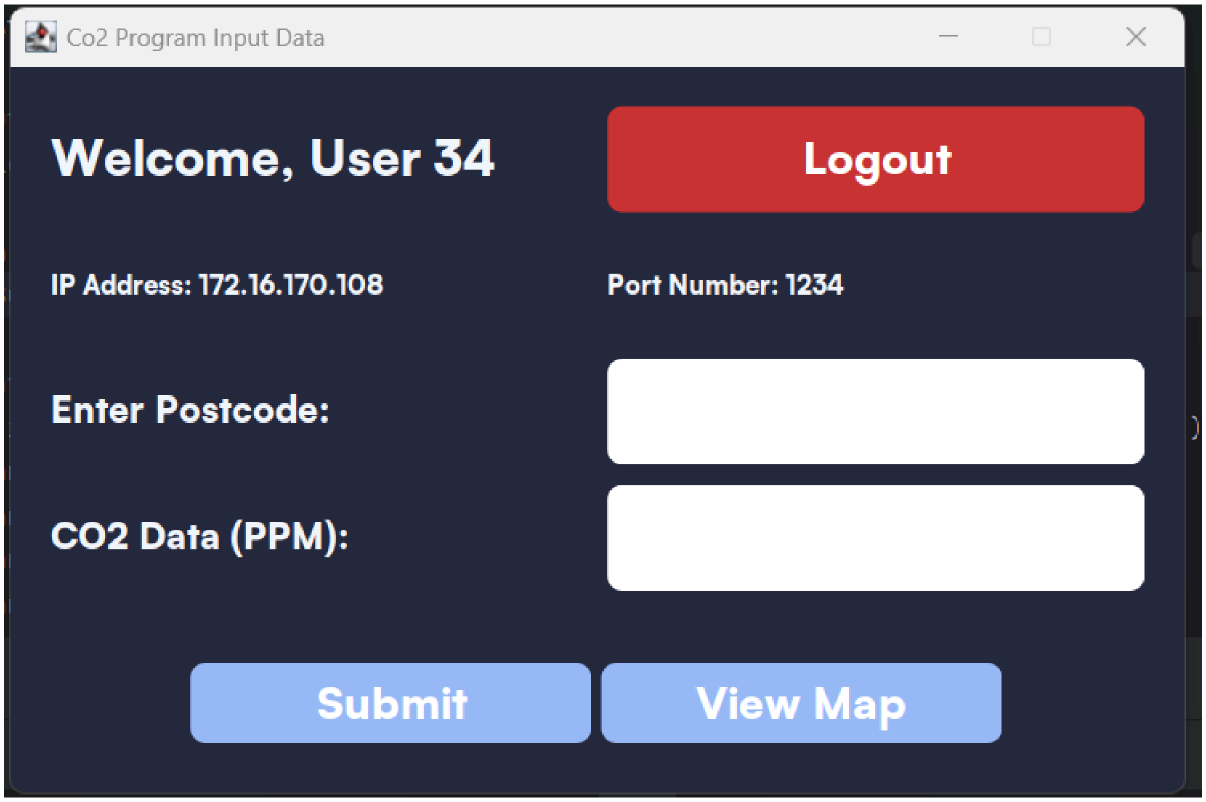
IP Address: 172.16.170.108

Port Number: 1234

Enter Postcode:

CO2 Data (PPM):

Next we added a logout button, this can be used to go back to the login screen and change the set Port, IP or User ID.



Co2 Program Input Data

Welcome, User 34

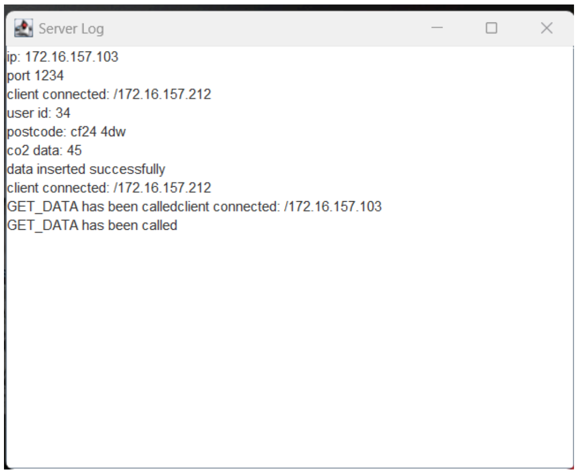
IP Address: 172.16.170.108

Port Number: 1234

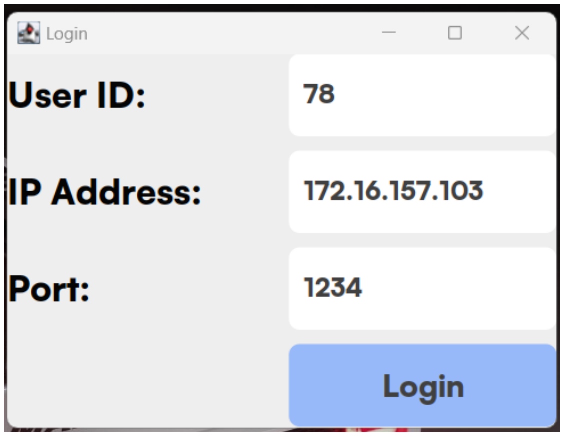
Enter Postcode:

CO2 Data (PPM):

Completed version for testing:



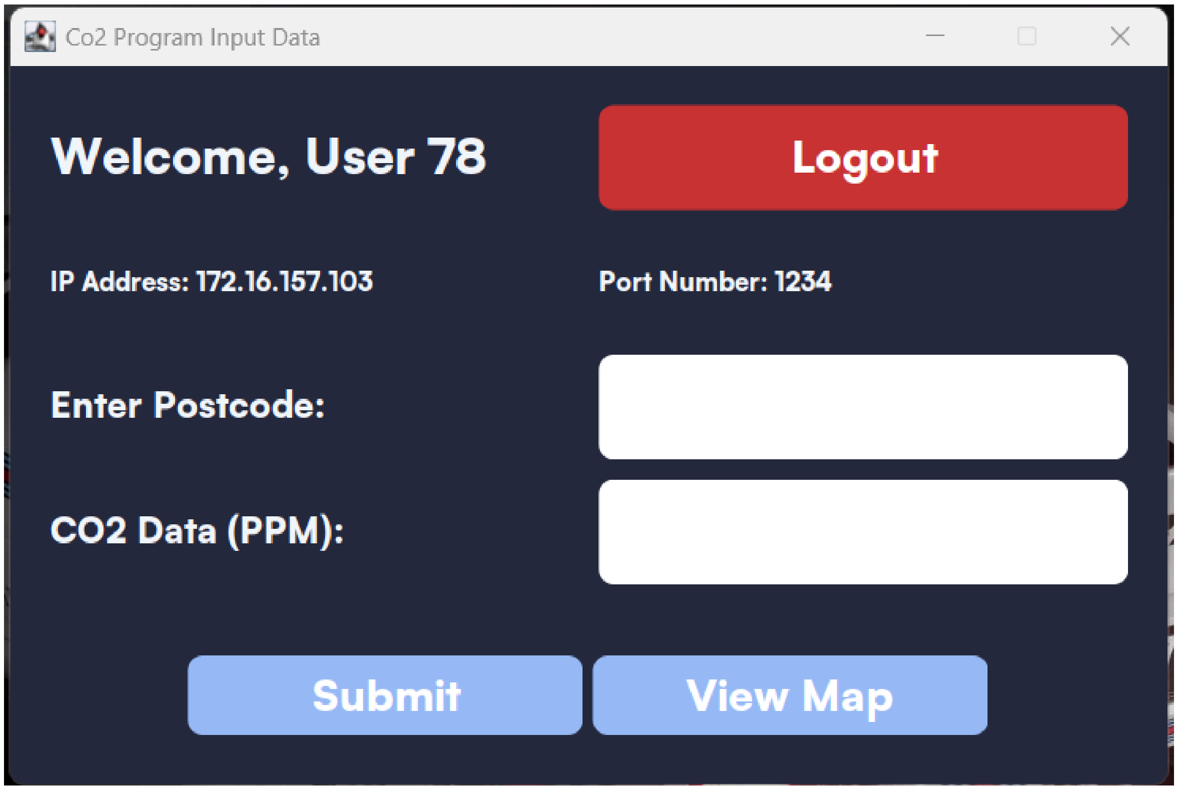
```
ip: 172.16.157.103
port 1234
client connected: /172.16.157.212
user id: 34
postcode: cf24 4dw
co2 data: 45
data inserted successfully
client connected: /172.16.157.212
GET_DATA has been calledclient connected: /172.16.157.103
GET_DATA has been called
```



User ID: 78

IP Address: 172.16.157.103

Port: 1234



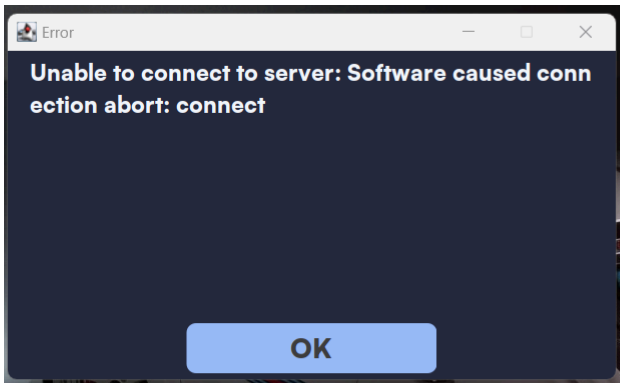
Welcome, User 78

IP Address: 172.16.157.103

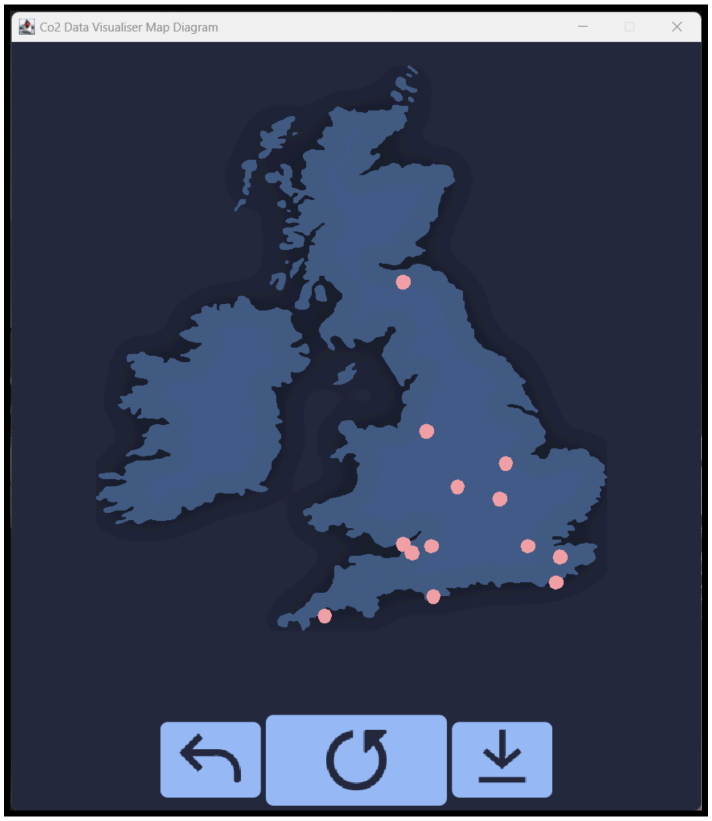
Port Number: 1234

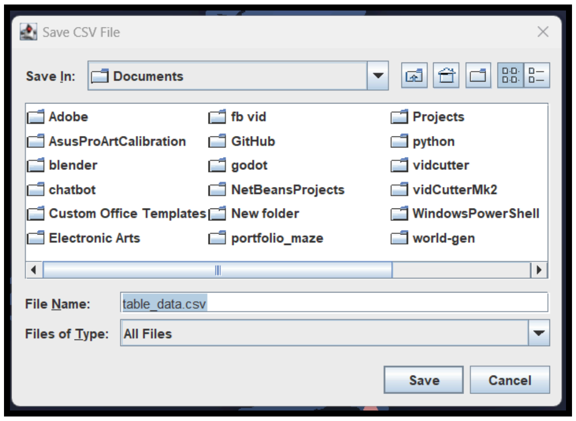
Enter Postcode:

CO2 Data (PPM):



Unable to connect to server: Software caused conn
ection abort: connect




Save CSV File

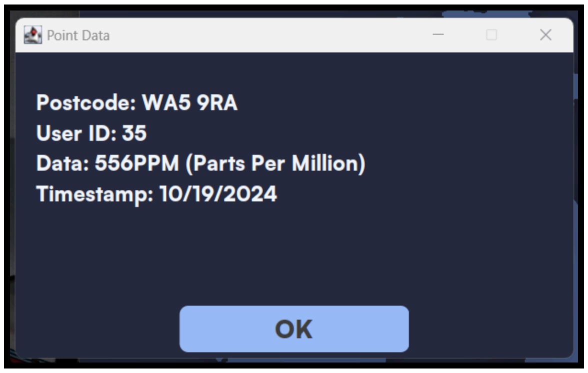
Save In: Documents

Adobe, AsusProArtCalibration, blender, chatbot, Custom Office Templates, Electronic Arts, fb vid, GitHub, godot, NetBeansProjects, New folder, portfolio_maze, Projects, python, vidcutter, vidCutterMk2, WindowsPowerShell, world-gen

File Name: table_data.csv

Files of Type: All Files

Save Cancel


Point Data

Postcode: WA5 9RA
User ID: 35
Data: 556PPM (Parts Per Million)
Timestamp: 10/19/2024

OK

<u>Execution Details</u>:

<u>Client Side</u>:

Step One - Once the software is running, enter a valid user ID, IP Address and Port Number which connect to the corresponding server.

Step Two - Press Login (This should open the portal for data entry.)

Step Three - Once the portal is open, users are able to enter a valid postcode and CO2 Reading (Measured in parts per million.) which saves to the database.

Step Four - Press the submit button, this uploads the data entered into the database where it's saved for future evaluation.

<u>Map Section</u>:

Step One - Press the View Map button on the main portal to open the data display section.

Step Two - Upon opening the map, users can click on data entries. This action will open a window displaying all the details of the entered data, such as the Postcode, User Id, Parts Per Million, and Time Uploaded.

Step Three - Press “OK” to return to the data entries.

Step Four - Press the refresh button to refresh the data entries. This updates the window with any recent entries.

Step Five - Press the file button to save the file. This saves the file in a CSV format which can be read without the software.

Step Six - Press return to continue entering data on the portal. Alternatively, close the application.

Server Side:

Step One - Run the server.

Step Two - Take note of the IP Address and Port which allows connection to the database.

Classes implementation

The databaseConnector class shown in the diagram above was implemented in the code to aid with handling the connections between the client and server. In using a separate class to handle this request we are able to open and close all connections when needed, this means connections aren't open for longer periods of time than required. Optimizing the softwares functionality.

The class titled app is responsible for controlling the visual aspect of the software, this class functions calling other 'classes and functions' which provides users with an easy-to-use UI for optimal performance. An example of how the app class operates could be demonstrated with 'map', where all locations are sourced from previously entered data and plotted on a map of the UK (United Kingdom).

The dataHandler class swaddles the inputted data into an object which allows for simple referencing running throughout the code. In doing this, the database connector is able to function more efficiently as the entire object can be sent through the server rather than individual strands of data. The dataResponse class functions by retrieving data once the user wants to process files. WIth additional time spent developing this, we can improve the basic processing through adding a data and time section when data is input. This would allow the user to develop graphics which show the change of CO2 emissions across an area for a certain time period.

Use of OOP

Our code is split into multiple files and functions to be called by the app class. We designed our code this way to allow us to keep track of what code was running at a time and to help find errors easier. Although OOP principles such as inheritance and polymorphism weren't necessary for this code we did use encapsulation to stop data from being modified accidentally when it has been accessed; this is shown when we create an object for the user's inputs instead of just keeping them as individual variables. Our design of splitting similar functions off into their own files is an example of abstraction, as splitting the complex systems into smaller part to be called upon when needed allows for these problems to be made much simpler, in future we would focus more into polymorphism to allow more redundancy so if someone decided to enter a number in a string format "twenty" instead of the expected "20" we could accommodate to this and used a modified version of the function to take the string and convert it to a float first and then process the data as usual.

Use of threads

The use of threads within' our software supports in improving both efficiency and functionality whilst preventing excessive system stress. An area we we're asked to consider when developing the system included ensuring it ran accordingly with users requests whilst having concurrent users. This was needed as it was aimed to be used by large corporations with multiple employees who will all be entering data simultaneously to one another.

The way we used threads within' our CO2 programme was achieved by the server creating a new thread for each new user running the service, this meant each user had an independent service reducing the chance of data congestion. Each independent thread would handle the different operations inputted by users meaning when a request to add data to the SQL database occurred the dedicated thread would process information before interacting with the database to store uploaded information.

This process is similar to that which manages data retrieval. When users make a request to the server to draw information from the SQL database the required documentation is fetched and formatted to be displayed to the user. Using thread pools also assisted in the efficiency of our program as it controlled the number of threads online. This prevents the server getting overwhelmed due to exponential connections and balances the user load.

To ensure that the errors of one thread did not interfere with that of another, we implemented error handling services improving system stability. As this program was developed to be run by multiple users, we developed the service so that no impact was made to other users. The final use of threads is in relation to scalability of the service. We've developed the software so that its able to handle an increasing number of users without major impacts on running speeds, this means as more users connect, more threads can be created to maintain processing.

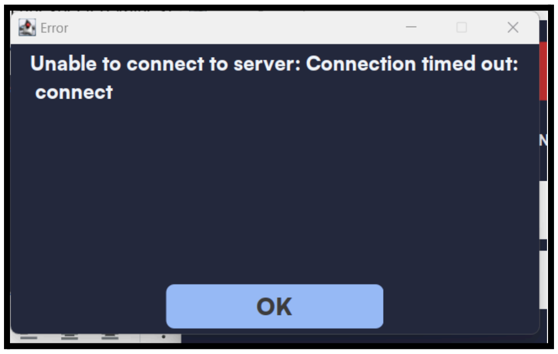
Exception and Error Handling

Our project uses a multitude of user based inputs due to the nature of its purpose, due to this, it's critical that we implement exceptional data handling services to ensure all inputs are valid and wont cause issues within the database or application. In addition to this, we must also implement exception and error handling services in-case of unintended server side issues a client may run into, examples of our implementation are shown below:

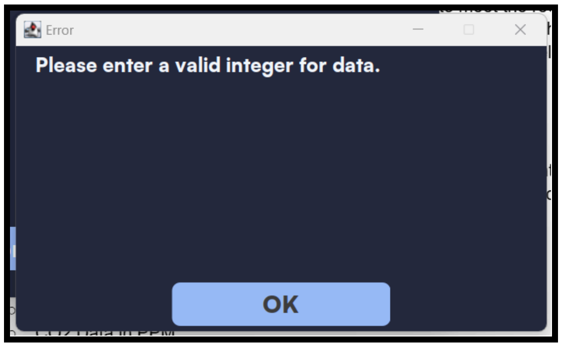
- No internet connection: In this instance, users will not be able to connect to the server which is hosted online. As a result, users will not be able to send or receive data making the software inoperable.

- Invalid inputs: If the user inputs data to the database containing unexpected values, it can result in server-side errors and cause corruption within the database.

Examples of Exception and Error handling:



Unable to connect to server: Connection timed out: connect



Please enter a valid integer for data.

By incorporating adequate exception and error handling mechanisms, we are able to significantly reduce the impact made as a result of user error. These mechanisms enable us to prevent system crashes before they happen meaning the overall integrity of the service is preserved. This approach improves the system's resilience, virtually eradicating the risk of breakage.

Coding Skills and Best Practices:

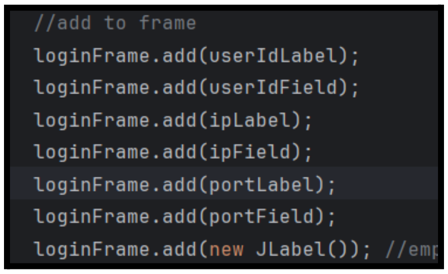
<u>Variable Names:</u>

Comprehensive variable names are a critical aspect of code for the purposes of readability. An example of how this can make an impact is when working in teams, if there are new additions to the code, clear variable names support in making it easier to understand how the code operates.
Our code uses the camelCase naming system (uppercase first letters of each word with the exception of the first word) for an assortment of reasons. Firstly, it feels more natural to type in contrast to other variable names. Snake_case is a prime example of this, where reaching for the underscore can be cumbersome whilst also disrupting the flow of typing.

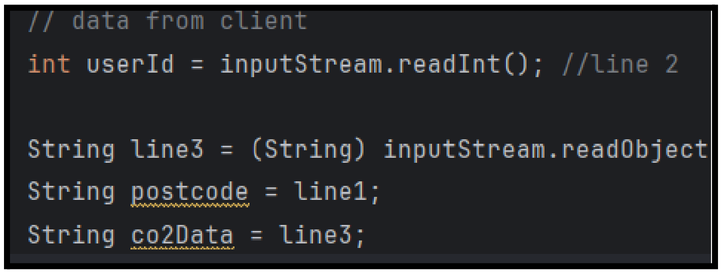
Another example of why we chose to use camelCase variables is in relation to it being the recommended naming convention for Java. It aligns with the in-built functions and libraries of most IDE's allowing for consistency to improve readability and maintainability. As this was a group project, understanding others' work was an essential process to complete sections of code. In following the same naming conventions, potential confusion is limited leading to an increase in efficiency.

One final reason for using camelCase within our code is due to its support in distinguishing areas of code. Class names typically use PascalCase, a variable where each word starts with an uppercase letter. In incorporating camelCase into our code, we are able to add clarity and readability to the code which makes any testing and debugging at a later date easier.

Example of variable names:



```
//add to frame
loginFrame.add(userIdLabel);
loginFrame.add(userIdField);
loginFrame.add(ipLabel);
loginFrame.add(ipField);
loginFrame.add(portLabel);
loginFrame.add(portField);
loginFrame.add(new JLabel()); //emp
```



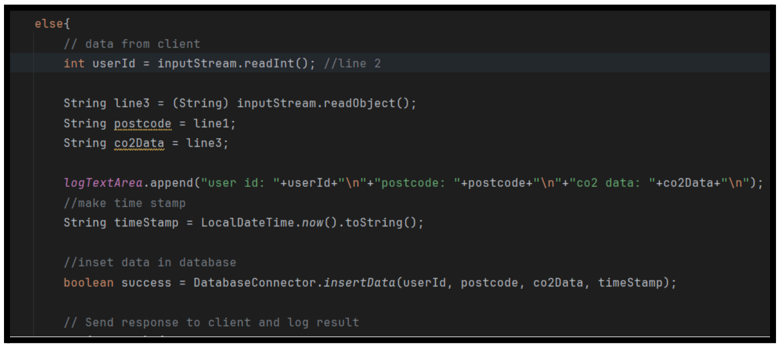
```
// data from client
int userId = inputStream.readInt(); //line 2

String line3 = (String) inputStream.readObject
String postcode = line1;
String co2Data = line3;
```

Commenting:

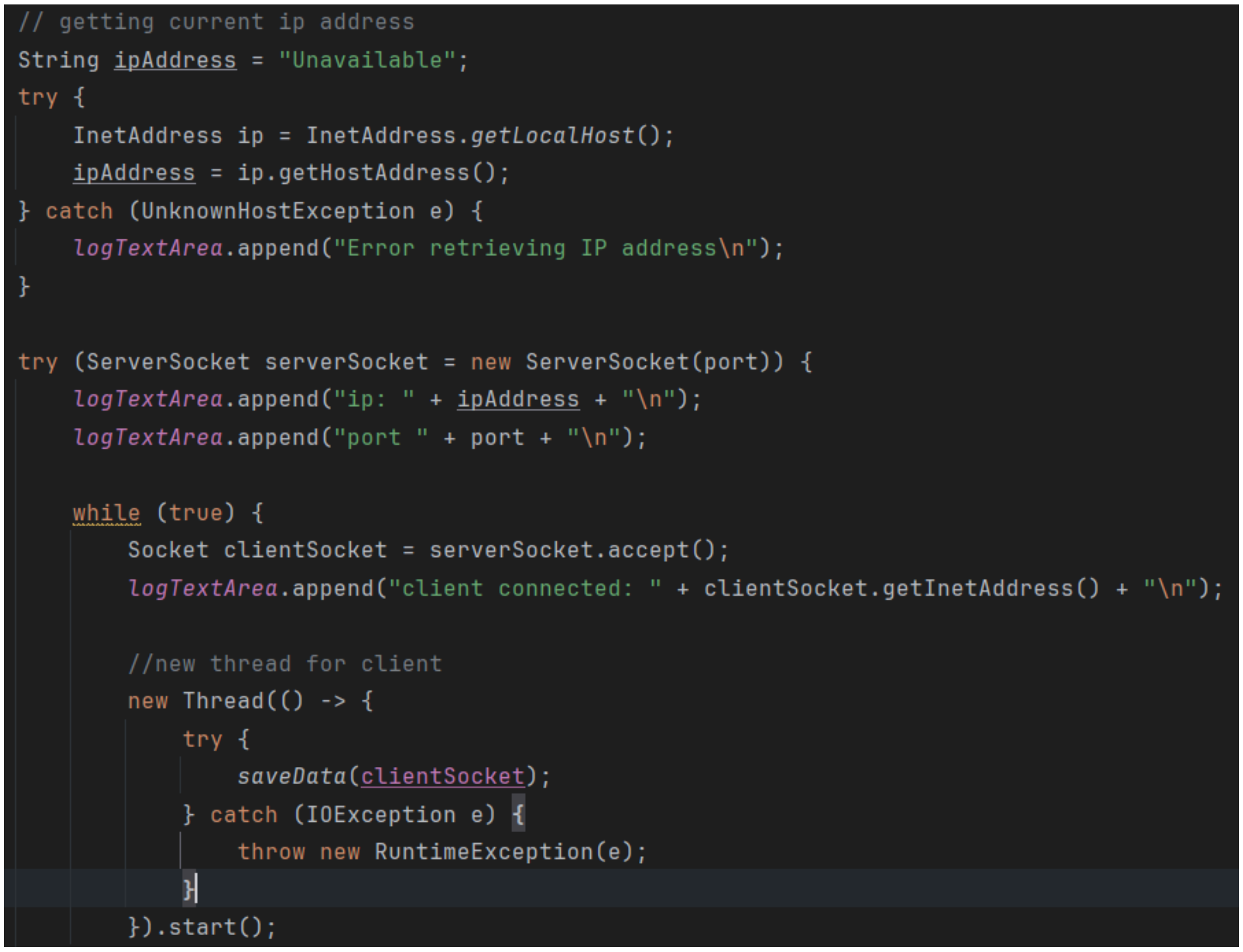
Whilst working as a team, it's important to include well structured comments within the code to streamline development. The main area in which commenting is able to help with collaboration is to support understanding. With new additions being made by an array of people, commenting can aid in reducing confusion, leading to more efficient workrates for faster completion. This is especially important when working to a deadline.

Despite not being relevant to this project, commenting can also be a useful tool in open sourced projects. It can be used as a teaching tool where areas of code are converted into common English language which makes it more legible to a wider range of users. Users are able to inspect code and use the comments to follow each process gaining an understanding of the direction being taken.

Example of commenting:

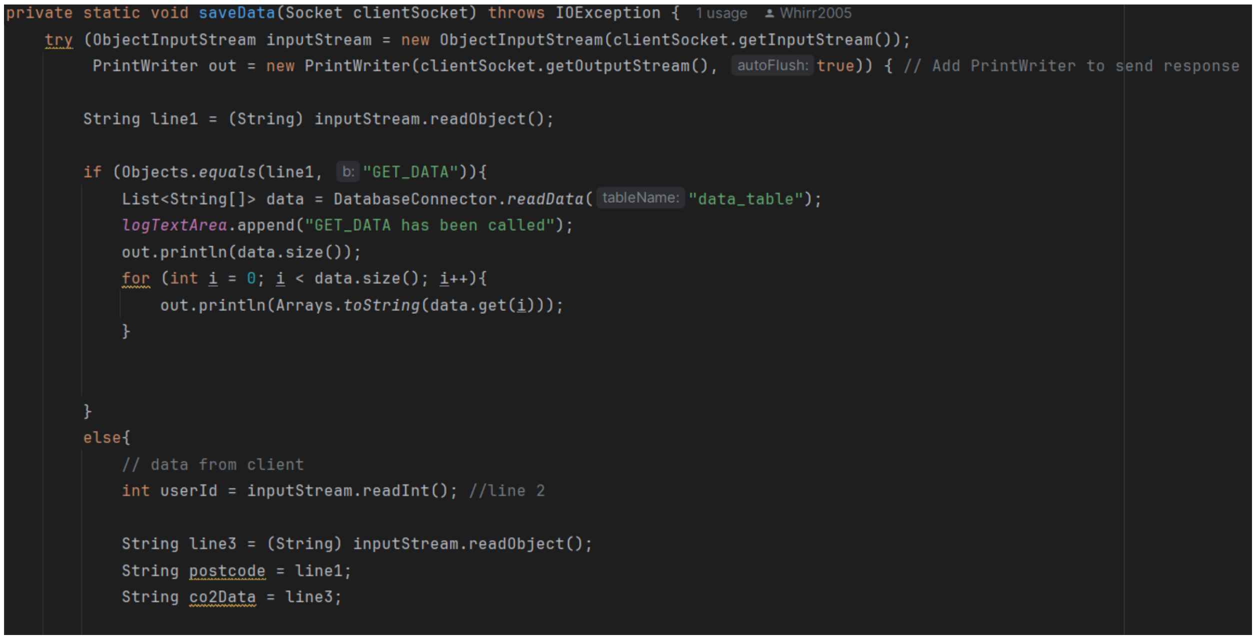


```
else{
    // data from client
    int userId = inputStream.readInt(); //line 2

    String line3 = (String) inputStream.readObject();
    String postcode = line1;
    String co2Data = line3;

    logTextArea.append("user id: "+userId+"\n"+"postcode: "+postcode+"\n"+"co2 data: "+co2Data+"\n");
    //make time stamp
    String timeStamp = LocalDateTime.now().toString();

    //inset data in database
    boolean success = DatabaseConnector.insertData(userId, postcode, co2Data, timeStamp);

    // Send response to client and log result
```



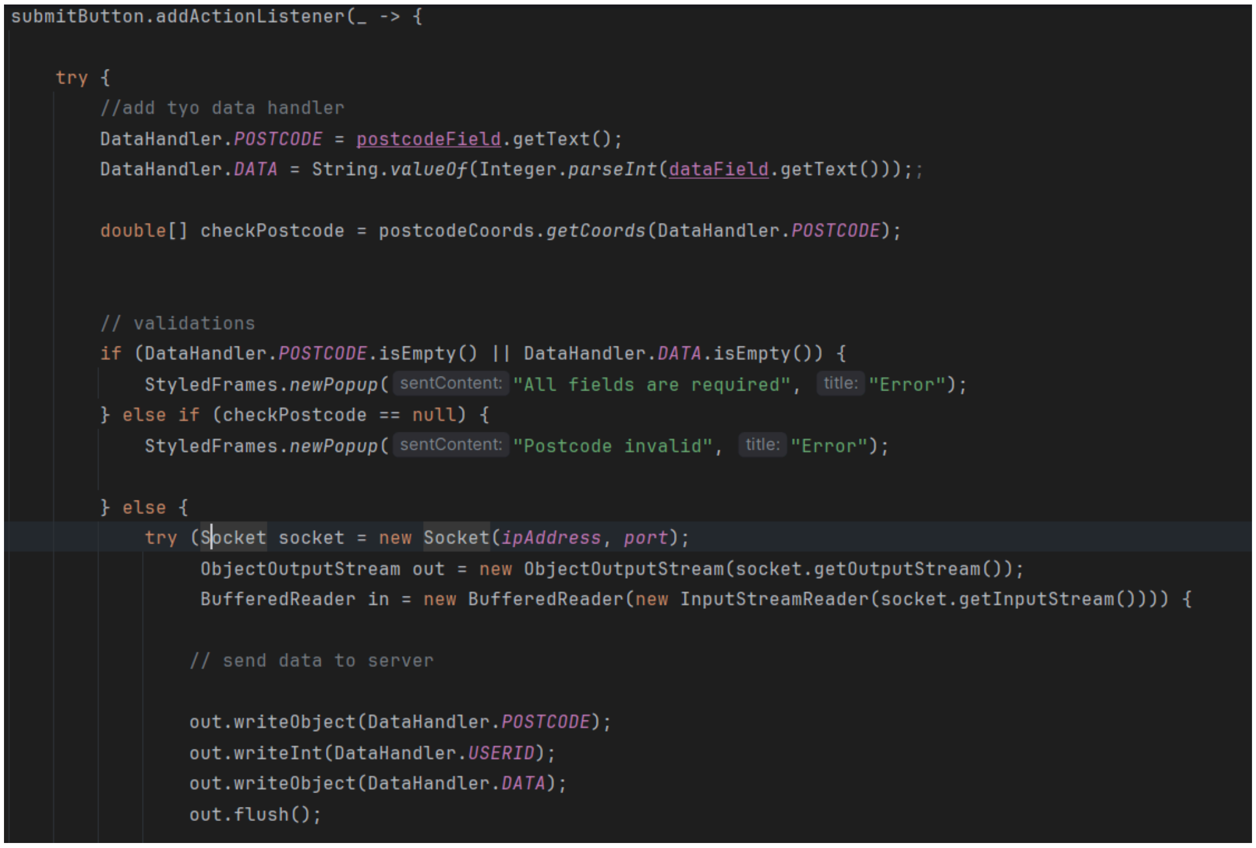
```java
// getting current ip address
String ipAddress = "Unavailable";
try {
    InetAddress ip = InetAddress.getLocalHost();
    ipAddress = ip.getHostAddress();
} catch (UnknownHostException e) {
    logTextArea.append("Error retrieving IP address\n");
}

try (ServerSocket serverSocket = new ServerSocket(port)) {
    logTextArea.append("ip: " + ipAddress + "\n");
    logTextArea.append("port " + port + "\n");

    while (true) {
        Socket clientSocket = serverSocket.accept();
        logTextArea.append("client connected: " + clientSocket.getInetAddress() + "\n");

        //new thread for client
        new Thread(() -> {
            try {
                saveData(clientSocket);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }).start();
```

The screen shot above shows the servers connection handling, when the server is started it automatically gets the current ip and listens on a previously determined port for a client to send a connection request. Once the server has accepted the connection, it will start to listen on the port for more data sent or another client's connection request. On our client-side code we have made the client connect and send the data as soon as connection is set up and once received the connection is closed, this is to help the server from having to hold multiple connections open at a time. The server is able to hold multiple connections by opening a new thread for each client that is connected this is future proofing due to the current small number of concurrent users and the connections being up for such a small time the whole server may be able to run on a single thread but once more users are active there would be errors.


```
private static void saveData(Socket clientSocket) throws IOException {  1 usage  Whirr2005
    try (ObjectInputStream inputStream = new ObjectInputStream(clientSocket.getInputStream());
         PrintWriter out = new PrintWriter(clientSocket.getOutputStream(),  autoFlush: true)) { // Add PrintWriter to send response

        String line1 = (String) inputStream.readObject();

        if (Objects.equals(line1,  b: "GET_DATA")){
            List<String[]> data = DatabaseConnector.readData( tableName: "data_table");
            logTextArea.append("GET_DATA has been called");
            out.println(data.size());
            for (int i = 0; i < data.size(); i++){
                out.println(Arrays.toString(data.get(i)));
            }


        }
        else{
            // data from client
            int userId = inputStream.readInt(); //line 2

            String line3 = (String) inputStream.readObject();
            String postcode = line1;
            String co2Data = line3;
```

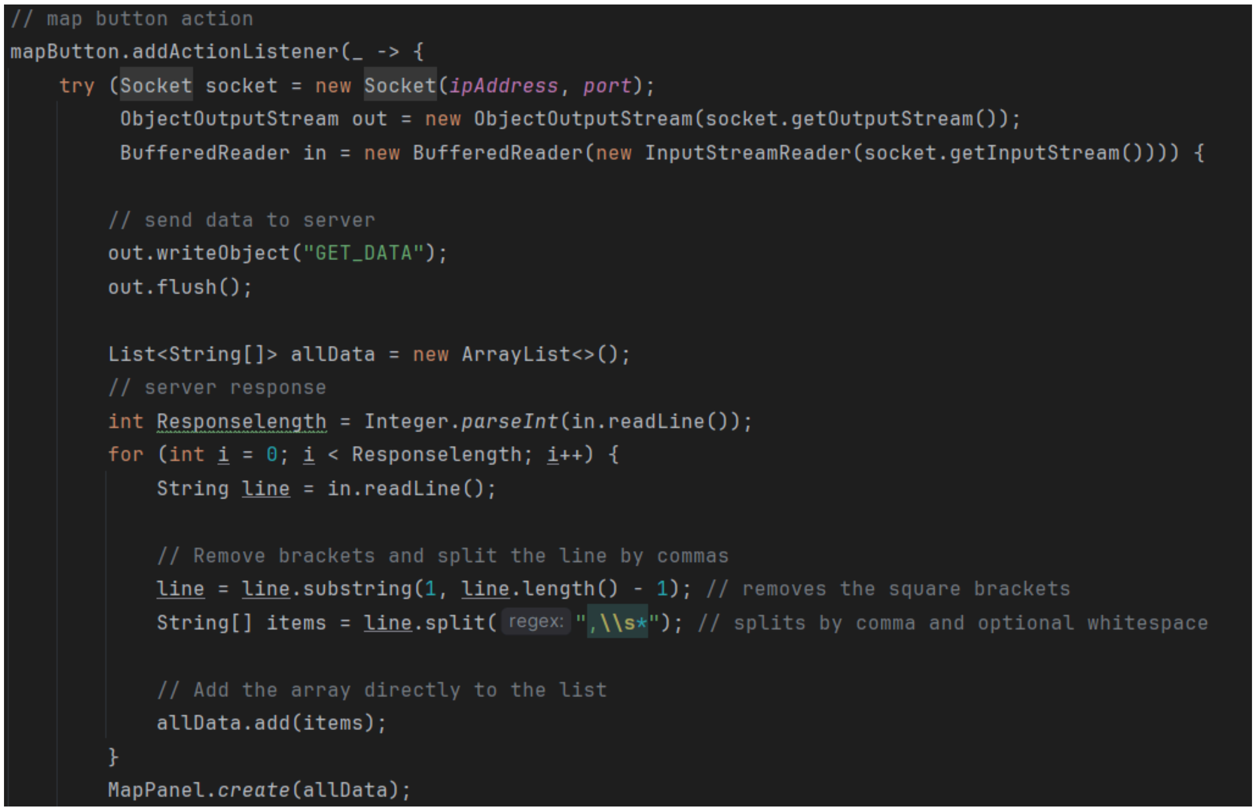
The code above takes the data sent from the client that is sent as an object and converts it back into the original strings. The server then takes these strings and connects to the database and adds the new record to the database. If the client has sent “GET_DATA” to receive the data from the database, the server will send a request to the database which will be responded with an array for the server to convert to a string to be sent to the client.

Client-Side Functions



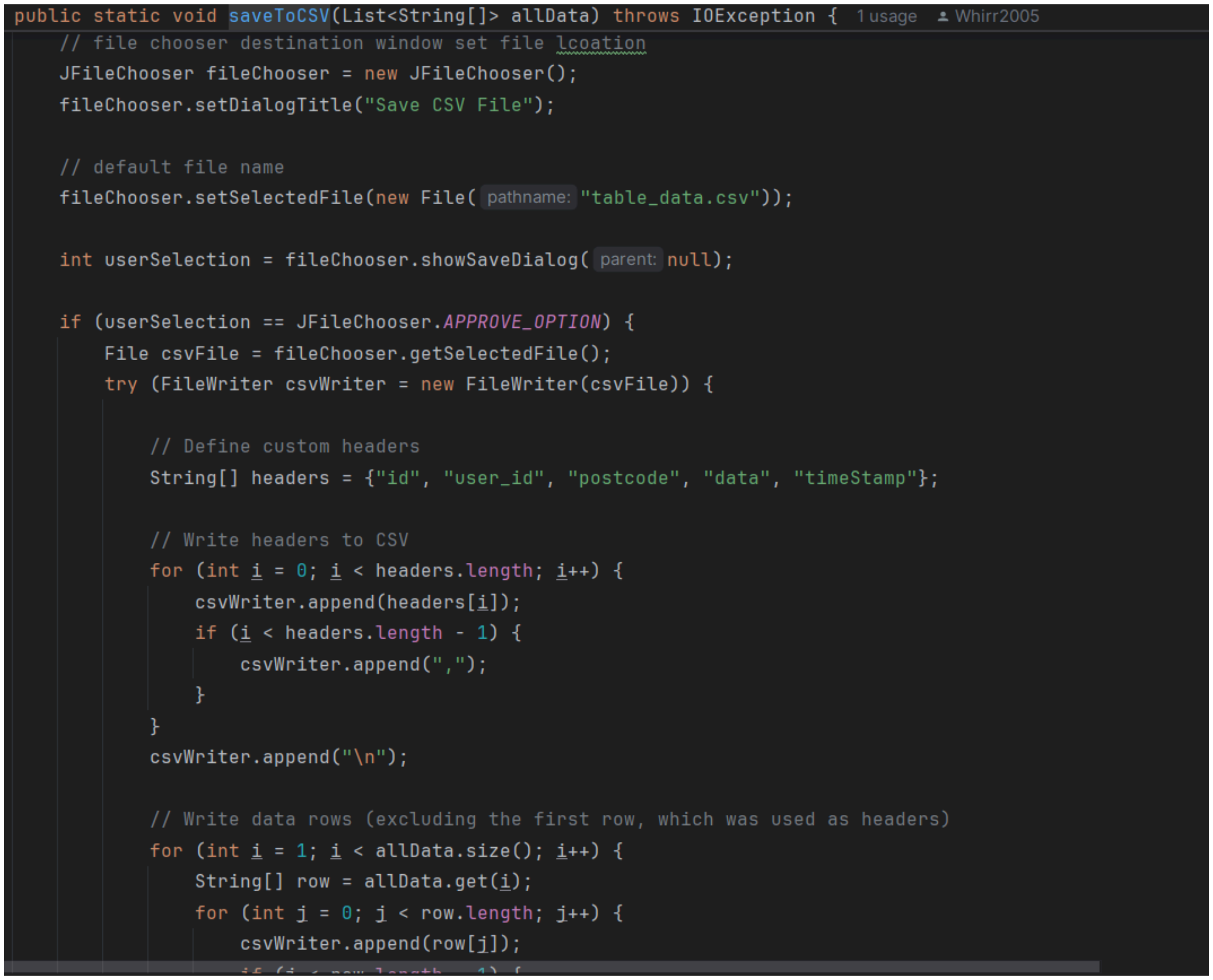
```java
submitButton.addActionListener(_ -> {

    try {
        //add tyo data handler
        DataHandler.POSTCODE = postcodeField.getText();
        DataHandler.DATA = String.valueOf(Integer.parseInt(dataField.getText()));;

        double[] checkPostcode = postcodeCoords.getCoords(DataHandler.POSTCODE);


        // validations
        if (DataHandler.POSTCODE.isEmpty() || DataHandler.DATA.isEmpty()) {
            StyledFrames.newPopup( sentContent: "All fields are required",  title: "Error");
        } else if (checkPostcode == null) {
            StyledFrames.newPopup( sentContent: "Postcode invalid",  title: "Error");

        } else {
            try (Socket socket = new Socket(ipAddress, port);
                 ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
                 BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()))) {

                // send data to server

                out.writeObject(DataHandler.POSTCODE);
                out.writeInt(DataHandler.USERID);
                out.writeObject(DataHandler.DATA);
                out.flush();
```

In the screenshot above code is shown that allows a user to enter data and then waits for the event of the user pressing the submit button to take the data. This is then written to an object that will be sent to the server for the data to be stored in the database. The connection to the server uses the server's ip and a previously decided port that the server will be listening on.



```java
// map button action
mapButton.addActionListener(_ -> {
    try (Socket socket = new Socket(ipAddress, port);
         ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
         BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()))) {

        // send data to server
        out.writeObject("GET_DATA");
        out.flush();

        List<String[]> allData = new ArrayList<>();
        // server response
        int Responselength = Integer.parseInt(in.readLine());
        for (int i = 0; i < Responselength; i++) {
            String line = in.readLine();

            // Remove brackets and split the line by commas
            line = line.substring(1, line.length() - 1); // removes the square brackets
            String[] items = line.split( regex: ",\\s*"); // splits by comma and optional whitespace

            // Add the array directly to the list
            allData.add(items);
        }
        MapPanel.create(allData);
```

The code above create a new connection to the server and sends a request for the server to get all the data from the database and to send it the client for it to be plotted onto the map from here the client is also able to save the data as a csv if they wish to do more data processing separate to our application, this could be a heat map to see what areas are getting the most users submitting data or take averages of areas to see what parts of the country have the worst co2 concentration.



```
public static void saveToCSV(List<String[]> allData) throws IOException {  1 usage  Whirr2005
    // file chooser destination window set file lcoation
    JFileChooser fileChooser = new JFileChooser();
    fileChooser.setDialogTitle("Save CSV File");

    // default file name
    fileChooser.setSelectedFile(new File( pathname: "table_data.csv"));

    int userSelection = fileChooser.showSaveDialog( parent: null);

    if (userSelection == JFileChooser.APPROVE_OPTION) {
        File csvFile = fileChooser.getSelectedFile();
        try (FileWriter csvWriter = new FileWriter(csvFile)) {

            // Define custom headers
            String[] headers = {"id", "user_id", "postcode", "data", "timeStamp"};

            // Write headers to CSV
            for (int i = 0; i < headers.length; i++) {
                csvWriter.append(headers[i]);
                if (i < headers.length - 1) {
                    csvWriter.append(",");
                }
            }
            csvWriter.append("\n");

            // Write data rows (excluding the first row, which was used as headers)
            for (int i = 1; i < allData.size(); i++) {
                String[] row = allData.get(i);
                for (int j = 0; j < row.length; j++) {
                    csvWriter.append(row[j]);
```

The function shown above adds the feature of being able to download all of the data in database as a .csv file doing this will allow the user to do further processing of data it works by taking the data used for the map but adds headers to a file then iterating over the array adding the data to the csv until a row is written then going to the next row and starting again.

Gathering data from the client.
The client side functions include sending data to the server.
Displaying data for the client.
Calculating map point positions.
Saving data into a csv for the client to download.

Testing

Now that we feel our application is in a deliverable state we are able to begin testing our software to ensure it is ready for deployment. To begin testing we need to create a test plan describing what test types we will use and showing the test data for each test. As the user is able to freely enter inputs into the text boxes this can cause security flaws and issues if not handled correctly. We will be using two test types, Unit testing and Acceptance testing. We will do an initial acceptance test first to ensure our program actually meets the requirements of the client brief before our unit testing so we may bring our program up to meet the requirements before properly Unit testing to test the functionality and find any bugs that may show up. After our unit testing we will do a more indepth acceptance testing section to completely check we satisfy all requirements.

Initial Acceptance Testing

For our initial user acceptance testing we will go through all the users requirements and make decisions on whether our solution meets them. The requirements outlined in the client brief are as follows:

- Clients can connect to the server
- Clients can submit the following information:
    - User ID
    - Postcode
    - CO2 Data in PPM
- The server should timestamp the submission and append it.
- Use object oriented programming.
- Separate client and server applications.
- Must use socket programming in the solution.
- Proper error handling and validation.

Results:

Clients can connect to the server:

In our application we do not use a server. Instead we use a database host which allows multiple users to input data from their application, but without "connecting to the server" this would be considered a **FAIL**.

Clients can submit data:

Our application uses sql to send information to our hosted database, including the user ID, postcode and CO2 data. This is easy to do and works with no errors. **PASS**

Server timestamps submissions:

Although not done by the server for reasons stated before, our program does timestamp the submissions into the database and saves them within the "TimeStamp" column. **PASS**

Uses object Oriented Programming:

Our java coded program uses object oriented programming (OOP) and makes use of classes and objects. **PASS**

Separate Client and Server applications:

As stated previously our application currently runs from a single client application that can be run on multiple machines to all insert data to the database. However this means this requirement is not met. **FAIL**

Using socket programming:

Our program does not make use of socket programming. **FAIL**

Proper error handling and validation:

We have implemented error handling and validation to all of our user input fields but they however have yet to be tested so cannot make a conclusion for this requirement. **UNTESTED**

Conclusion:

After our initial acceptance testing it is clear we need to change our programs approach, although our current solution is completely functional and would be effective for the required application and use case, as it is able to be used by multiple people simultaneously and allows users to input their data readings to the database and even view it visualised on a map. It does this without using socket programming and therefore does not meet a majority of the requirements and therefore we have decided as a group to go back into development and change the approach to more accurately fit the clients requirements.

The changes made to our project in order to more accurately fit the requirements are documented within the "Part Two (Post testing)" section of our projects documentation and development timeline. Link to changes

Unit Testing

Now that the necessary changes, discovered within our initial acceptance testing have been made to the project we are able to move on to the next stage of testing, unit testing. For this section of testing we will create a test plan to outline how we will carry out the tests, an expected results section to show how the results should look if they are successful, the actual results to show the actual results weather it is a pass or fail finally we will fix any errors and bugs found in the previous section and re test those against the expected results to get our new results.

Test plan

Test Data 1 - Test server application can run and open a server:

Run the server application
Verify server is running using the server log

Test Data 2 - Test client application can connect to the server with IP and Port Number:

Run the server application
Run the client application
In the server app check the ip and port number
In the client app enter the ip and port number
Press login
Enter the postcode and co2 readings
Press submit
Check server log for:
“client connected: /172.16.170.72” or other ip address

Test Data 3 - Test user can add data to the data entry fields:

Run the server application
Run the client application
In the server app check the ip and port number
In the client app enter the ip and port number
Press login
Click the postcode entry box
Attempt to enter text into the field
Click the Co2 data entry box
Attempt to enter text into the field

Test Data 4 - Test users data is inserted to the database:

Run the server application
Run the client application
In the server app check the ip and port number
In the client app enter the ip and port number
Press login
Click the postcode entry box
Attempt to enter text into the field
Click the Co2 data entry box

Attempt to enter text into the field
Press the submit
Go to the database
Refresh the “data_table” table
Check for the new data

Test Data 5 - Test the entry fields are validated and have error handling:

Run the server application
Run the client application
In the server app check the ip and port number
In the client app enter the ip and port number
Press login
Click the postcode entry box
Enter an various entries into the field such as:

- Integers
- Text
- Non existing post codes
- Blank entry

Press submit
Click the Co2 data entry box
Enter an various entries into the field such as:

- Text
- integers
- Blank entry

Press submit

Test Data 6 - Test map is able to get data from the server:

Add System.out.println(row[2]+” ”+row[3]); to line 37 of MapPanel.java
Run the server application
Run the client application
In the server app check the ip and port number
In the client app enter the ip and port number
Press login
In the client app press the “view map” button
In the server app check for:
“client connected: /172.16.170.72”
“GET_DATA has been called”
Check console for postcodes and data printed

Test Data 7 - Test map points are plotted:

Run the server application
Run the client application
In the server app check the ip and port number
In the client app enter the ip and port number
Press login
In the client app press the “view map” button
Map panel window should open
Check pink points are plotted at locations of data

Test Data 8 - Test data can be downloaded into a CSV:

Run the server application
Run the client application
In the server app check the ip and port number
In the client app enter the ip and port number
Press login
In the client app press the “view map” button
Map panel window should open
Press the download icon button
Use the save file window to choose a location
Locate the saved file on your machine

Test Data 9 - Test miscellaneous buttons functionality:

Run the server application
Run the client application
In the server app check the ip and port number
In the client app enter the ip and port number
Press login
In the client app press the “view map” button
Map panel window should open
Use buttons such as:

- Refresh
- Back
- Plotted point buttons
- Logout button

Expected results:

Expected Test Result 1

Server runs and displays port and ip in log

Expected Test Result 2

Client connects to server
Server displayed client ip with connection message

Expected Test Result 3

Text can be imputed in input fields

Expected Test Result 4

Database should contain new submitted data

Expected Test Result 5

Error messages popup in a window
Correct inputs can be added with correct message popup
Bad inputs won't\ be added to database

Expected Test Result 6

Server log shows “GET_DATA”
Client console shows data printed

Expected Test Result 7

Postcodes are plotted on the map in their correct position

Expected Test Result 8

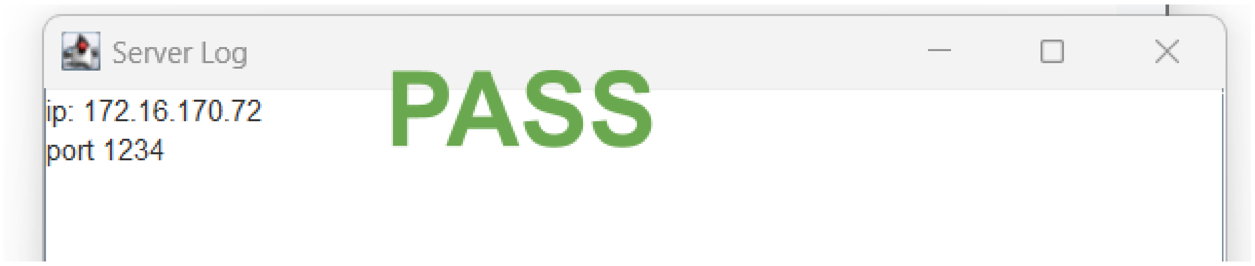
Csv file is saved locally to the users machine
Data inside csv matches the data in the database

Expected Test Result 9

Refresh button should close and re open map window
Back button closes map window
Plotted points should show popup with the data
Logout button takes user back to “login” window with ip and port number entry

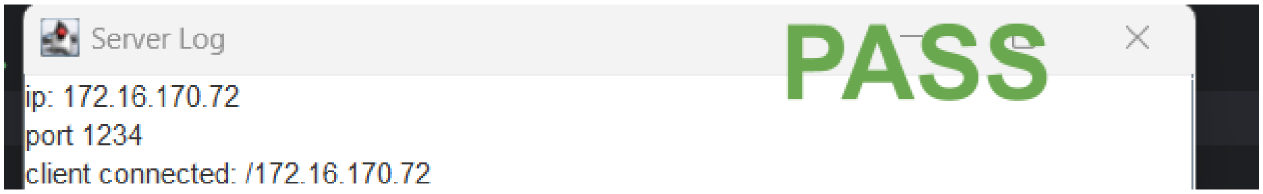
Actual results:

Actual Test Result 1



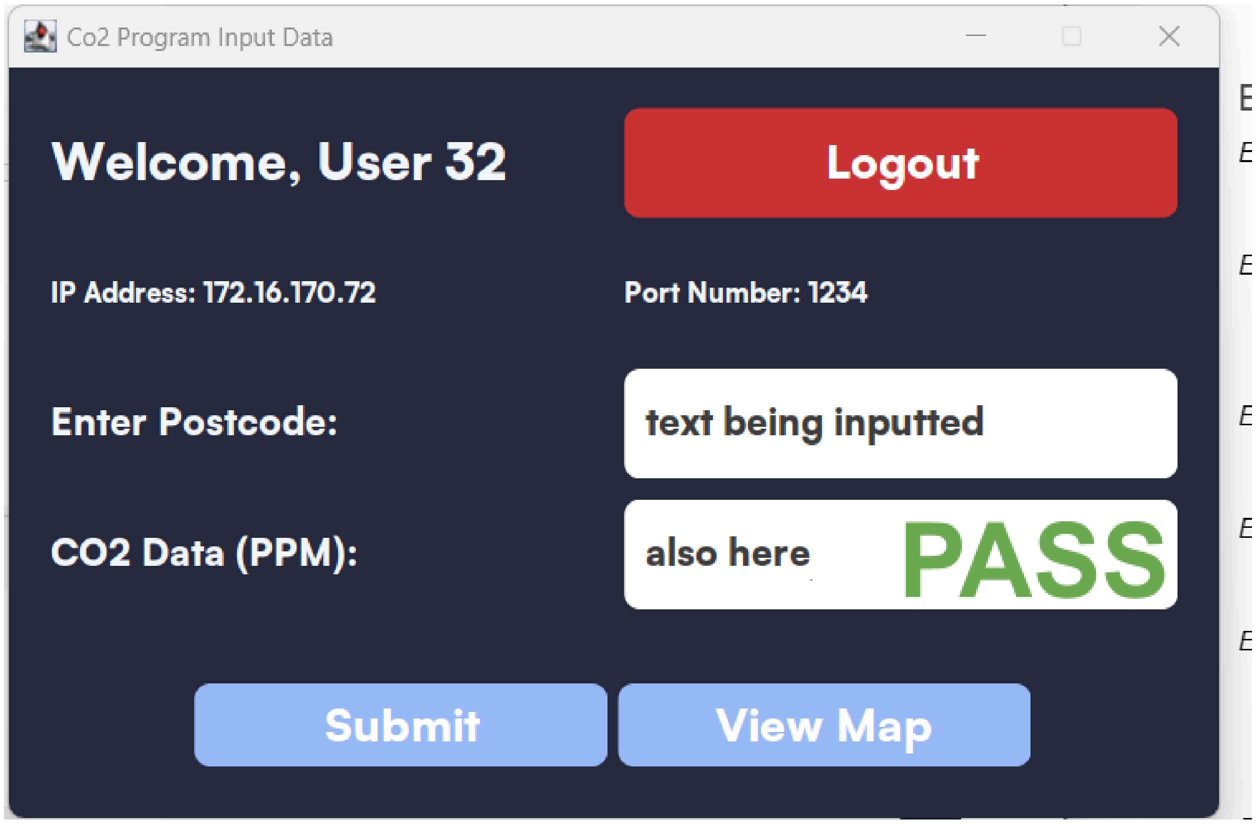
```
Server Log
ip: 172.16.170.72
port 1234
```

Actual Test Result 2



```
Server Log
ip: 172.16.170.72
port 1234
client connected: /172.16.170.72
```

Actual Test Result 3



Co2 Program Input Data

Welcome, User 32

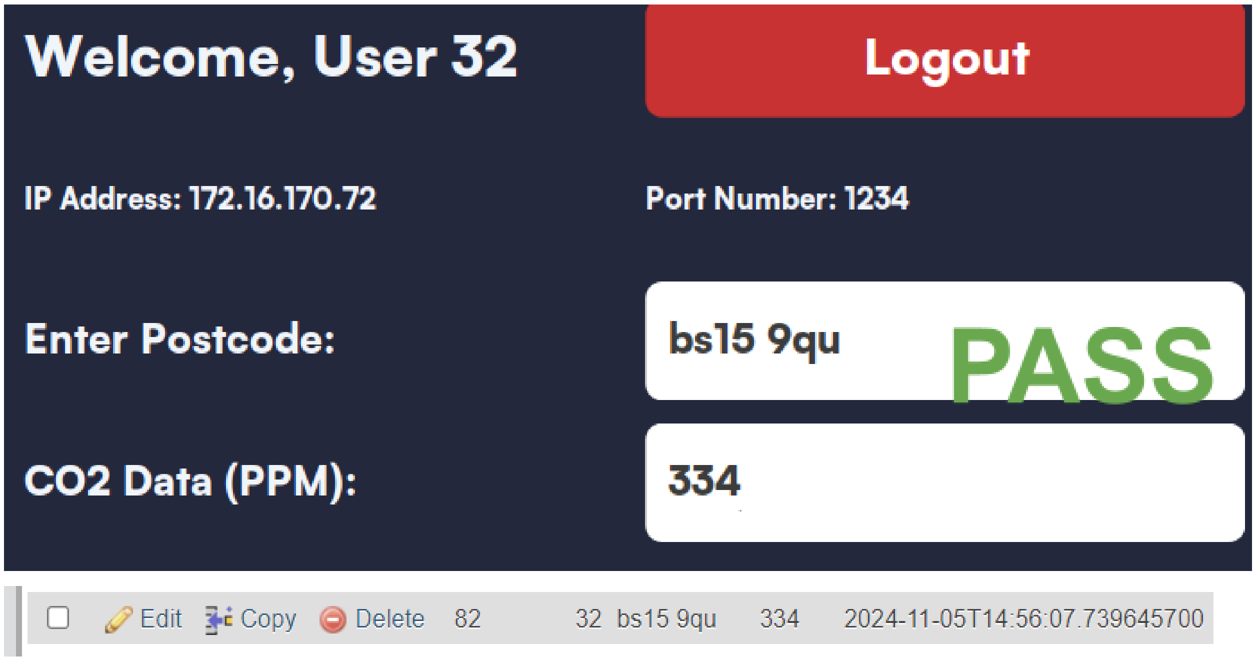
IP Address: 172.16.170.72

Port Number: 1234

Enter Postcode: text being inputted

CO2 Data (PPM): also here

Actual Test Result 4



Welcome, User 32

Logout

IP Address: 172.16.170.72

Port Number: 1234

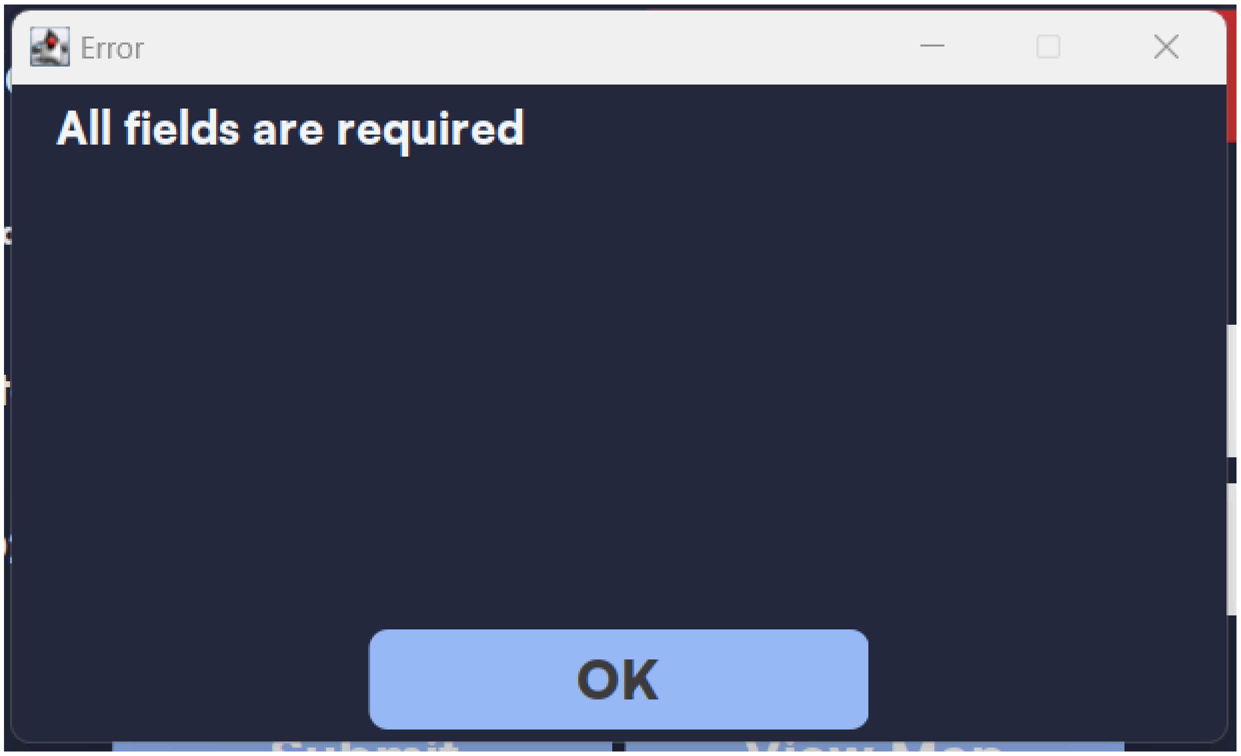
Enter Postcode: bs15 9qu

CO2 Data (PPM): 334

PASS

☐	Edit	Copy	Delete	82	32	bs15 9qu	334	2024-11-05T14:56:07.739645700

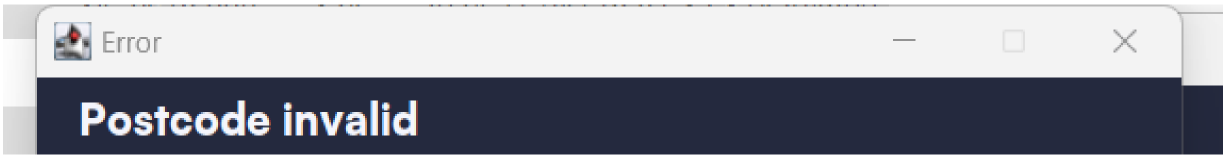
Actual Test Result 5



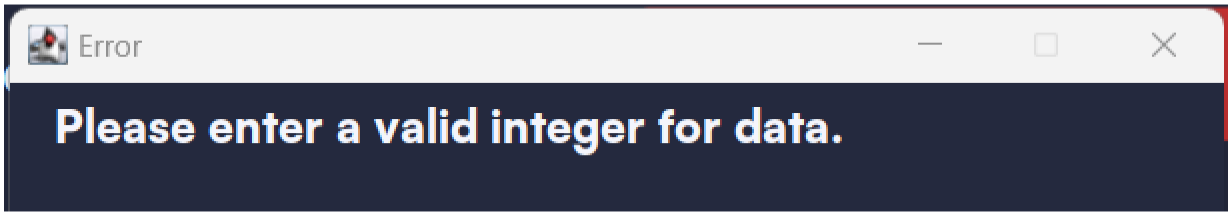
Error

All fields are required

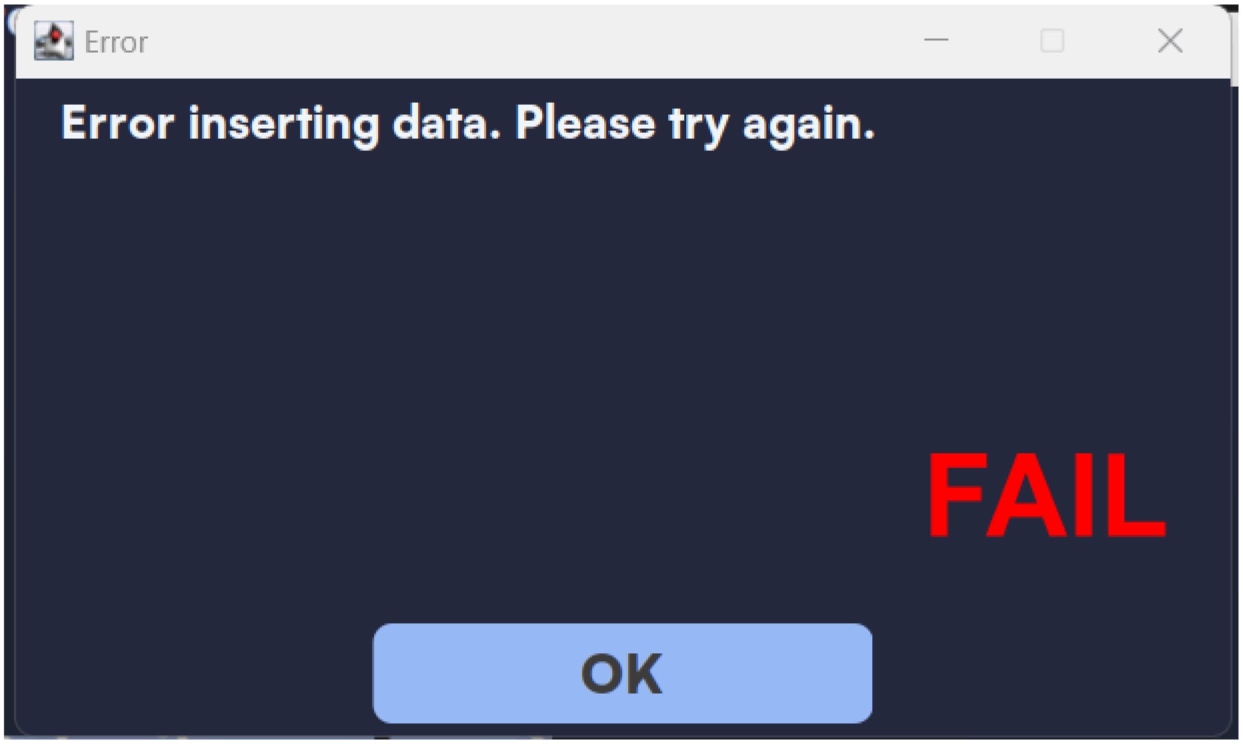
OK


Error

Postcode invalid


Error

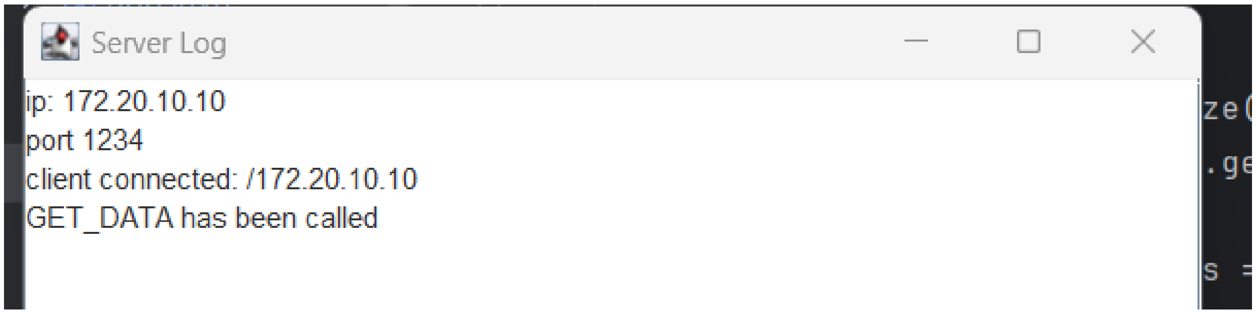
Please enter a valid integer for data.


Error

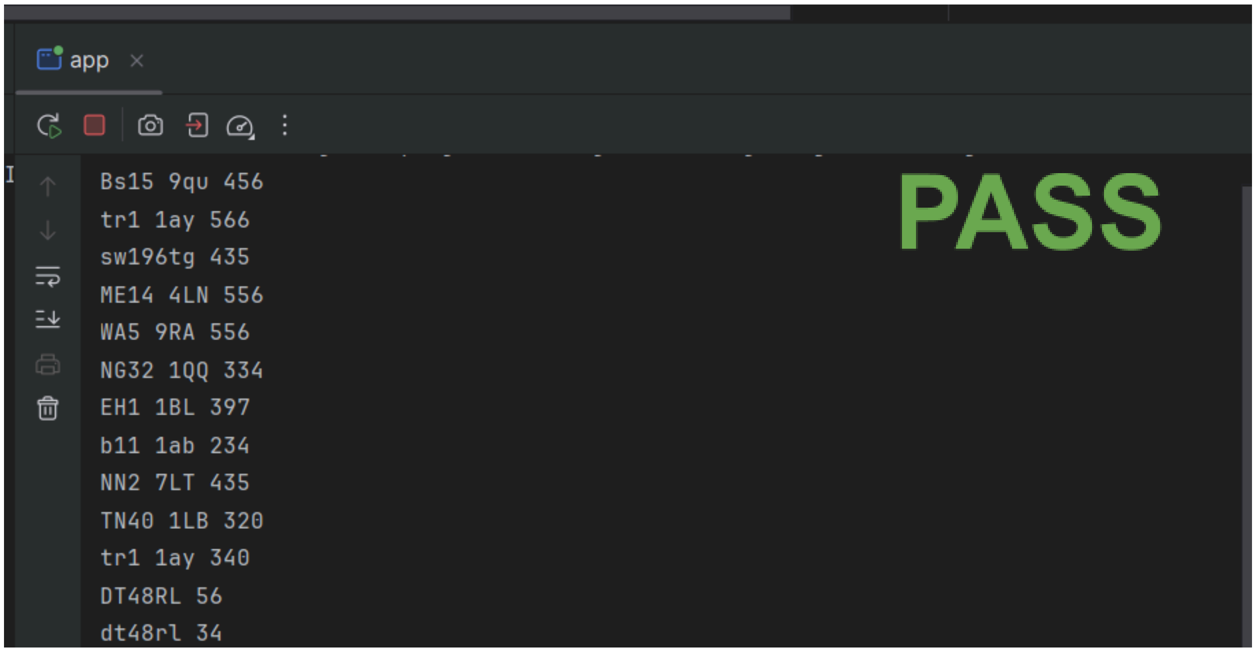
Error inserting data. Please try again.

When data is submitted correctly we can this popup claiming it was not inserted. All other popups were correct. Test can be counted as a partial fail as only a small portion didn't work correctly.

Actual Test Result 6

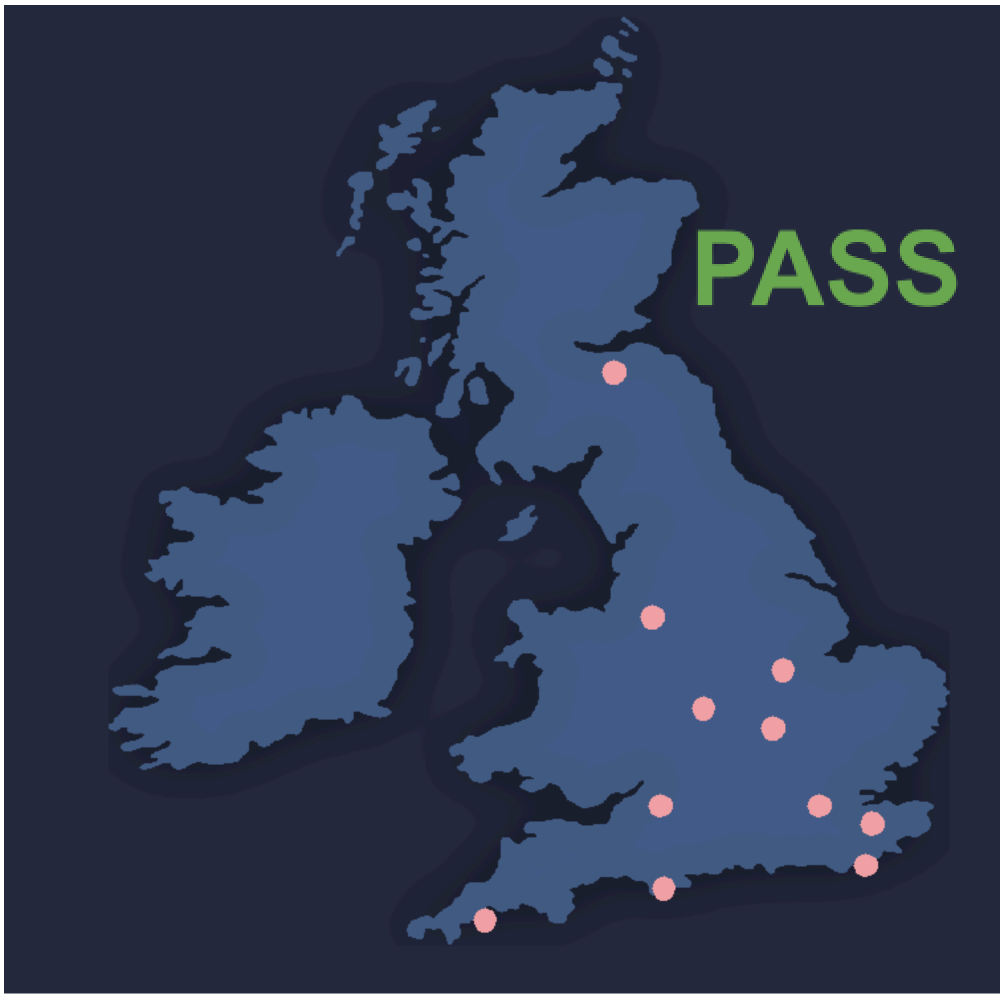


```
ip: 172.20.10.10
port 1234
client connected: /172.20.10.10
GET_DATA has been called
```

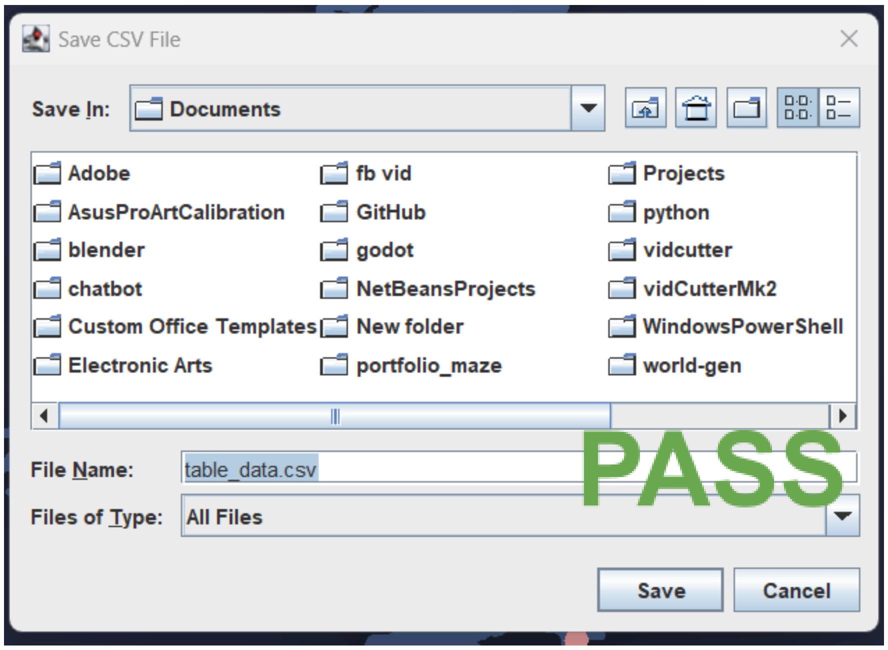


```
Bs15 9qu 456
tr1 1ay 566
sw196tg 435
ME14 4LN 556
WA5 9RA 556
NG32 1QQ 334
EH1 1BL 397
b11 1ab 234
NN2 7LT 435
TN40 1LB 320
tr1 1ay 340
DT48RL 56
dt48rl 34
```

Actual Test Result 7



Actual Test Result 8:

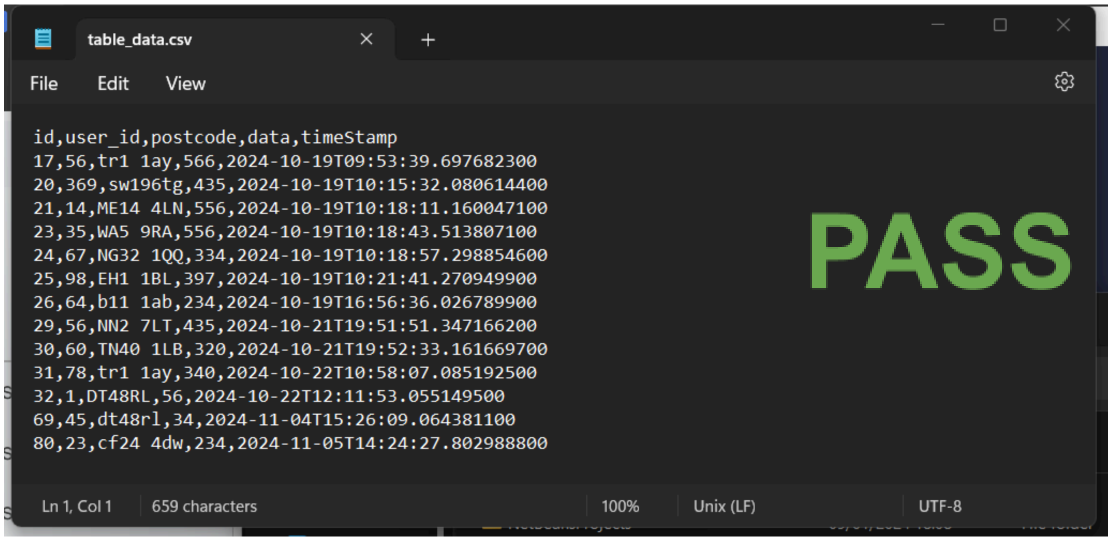

Save CSV File

Save In: Documents

Adobe
AsusProArtCalibration
blender
chatbot
Custom Office Templates
Electronic Arts
fb vid
GitHub
godot
NetBeansProjects
New folder
portfolio_maze
Projects
python
vidcutter
vidCutterMk2
WindowsPowerShell
world-gen

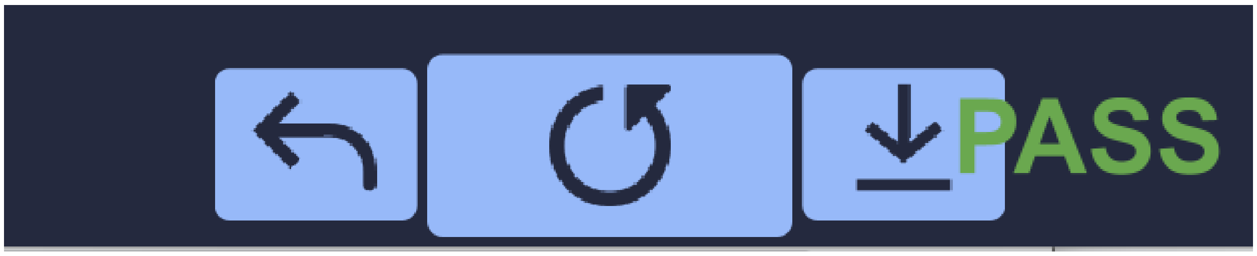
File Name: table_data.csv

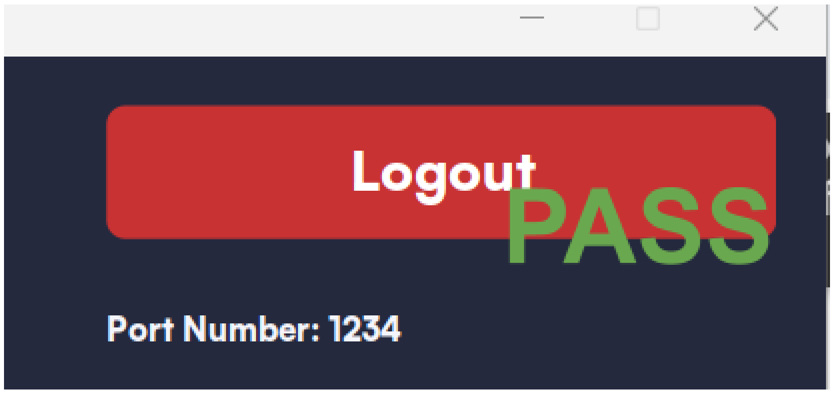
Files of Type: All Files



```
id,user_id,postcode,data,timeStamp
17,56,tr1 1ay,566,2024-10-19T09:53:39.697682300
20,369,sw196tg,435,2024-10-19T10:15:32.080614400
21,14,ME14 4LN,556,2024-10-19T10:18:11.160047100
23,35,WA5 9RA,556,2024-10-19T10:18:43.513807100
24,67,NG32 1QQ,334,2024-10-19T10:18:57.298854600
25,98,EH1 1BL,397,2024-10-19T10:21:41.270949900
26,64,b11 1ab,234,2024-10-19T16:56:36.026789900
29,56,NN2 7LT,435,2024-10-21T19:51:51.347166200
30,60,TN40 1LB,320,2024-10-21T19:52:33.161669700
31,78,tr1 1ay,340,2024-10-22T10:58:07.085192500
32,1,DT48RL,56,2024-10-22T12:11:53.055149500
69,45,dt48rl,34,2024-11-04T15:26:09.064381100
80,23,cf24 4dw,234,2024-11-05T14:24:27.802988800
```

Actual Test Result 9

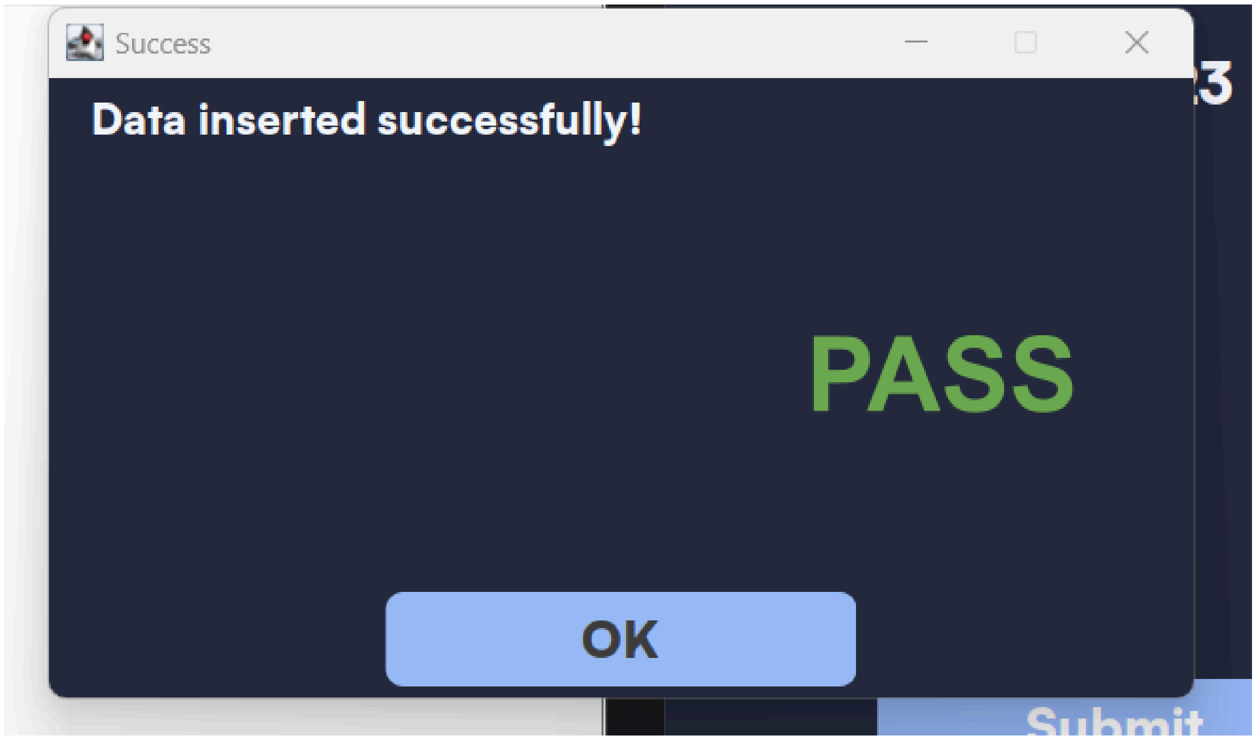




Buttons worked as expected.

Retests:

Test 5 Re-Test.


Success

Data inserted successfully!

Acceptance Testing:

Using the same requirements as used before in the initial acceptance testing section we can re-test our project against the current version, with the recent additions. We will only test against the areas that failed or were untested initially in order to save resources and time as we have not changed any of the areas that previously met requirements.

Results:

Clients can connect to the server:

With our new version of the program the client app is able to connect to the server application by entering the correct ip and port number. PASS

Separate Client and Server applications:

Our project has been separated into two working applications for both the server and the client app. PASS

Using socket programming:

We have now implemented socket programming successfully. PASS

Proper error handling and validation:

We have implemented and fully tested our error handling and validation fixing any bugs found in testing and it now works as intended. **PASS**