



IOT Training - Final Project

Third Eye

Work Done By:

Asmaa Abdelkader - 2203145

Christine Guirguis Labib - 2206162

Habiba Abdallah Hammad - 2203167

Rana Ahmed Mohamed Gaber - 2203180

Yara Ibrahim Yehia Allam - 2203178

Table of Contents

1. Overview	
2. Flutter App Design & Implementation	
3. Maquette Design	
4. Logic, ESP32, and Sensors Circuitry	
5. Computer Vision Using the ESP-CAM	
6. MQTT and System Lifecycle	
7. Results and Conclusion	

1. Overview

Third Eye is a flutter-based IoT system designed to aid those visually impaired and/or disabled individuals with interactions with the world around them. The idea started with the wish to incorporate computer vision into our final project, using an ESP-CAM, as well as adding a twist to the regular walking aid cane.

The application itself has a simple interface: a user only needs to provide some key details about themselves - including an address and emergency contact details - and they're good to go. They can access those details easily via the landing page and can then navigate to two separate dashboards: one for real-time readings of their heart rate (thanks to a sensor intuitively placed at the handle of the stick), and one for the objects encountered by the ESP-CAM in real life.

This is tied together using the MQTT protocol and HiveMQ as our cloud platform for storing readings, Firebase for user authentication and account management, and its cloud Firestore for handling user profile data.

2. Flutter App Design and Implementation

2.1. UI Design

The starting point was the logo; I felt the need to add an identity to the project, subtle as it may be. An eye felt appropriate and would easily blend in anywhere in the application, and from there it was a simple jump to the title idea: 'Third Eye'.



Next, I experimented with some widgets and ideas in Flutterflow - all pictured below. I came up with a rough color palette, too, initially.



As you can see, I chose to stick with more muted blues, showcasing a clean look and giving an overall unobtrusive feel to the app.

I also stuck to a specific font from the google_fonts library: Sora.

This is demonstrated in the UI design:



Welcome back!

Email Address

Enter your email...

Password

Enter your password...



Login

Don't have an account? Create →



Get Started

Create your account below.

Email Address

Enter your email...

Password

Enter your password...




Confirm Password

Enter your password...





Create Account

← Login Already have an account?


 **Complete Profile**


Complete Profile




Welcome,

[username]

 **27°C** **Wednesday**

**Last Detected
Heartrate**
 **89 BPM**

**Last Seen
Object**
 **Person**

Quick Actions

Track Heart Rate

Detect Objects



My Account

Name

Age

Address

Phone Number

Emergency Contact Name

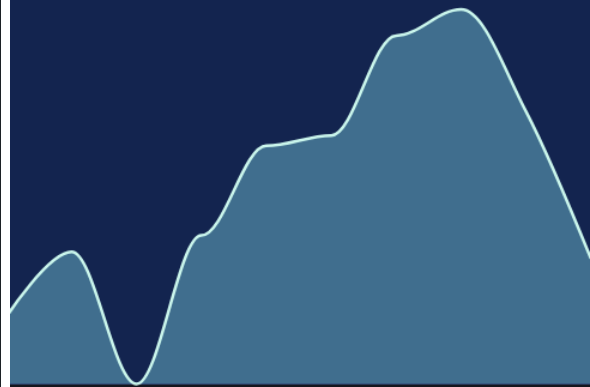
Emergency Contact Number

Remove Account



Track Heart Rate

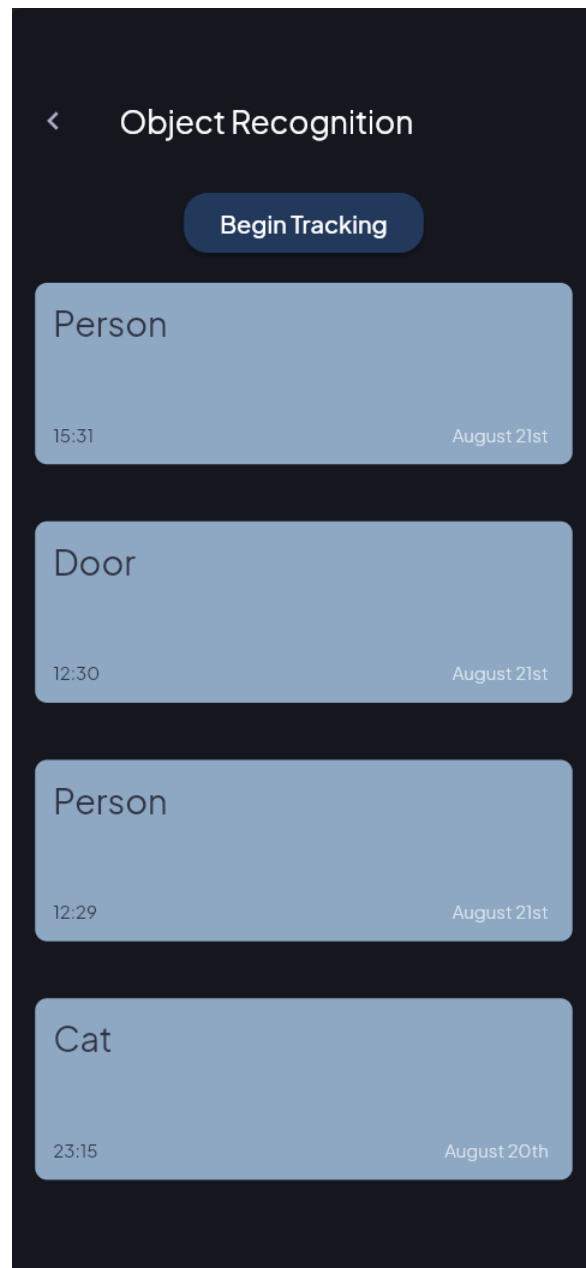
101 BPM



*Your heartrate is elevated, please find
a place to rest.*

Please place your finger on
the sensor to begin tracking...

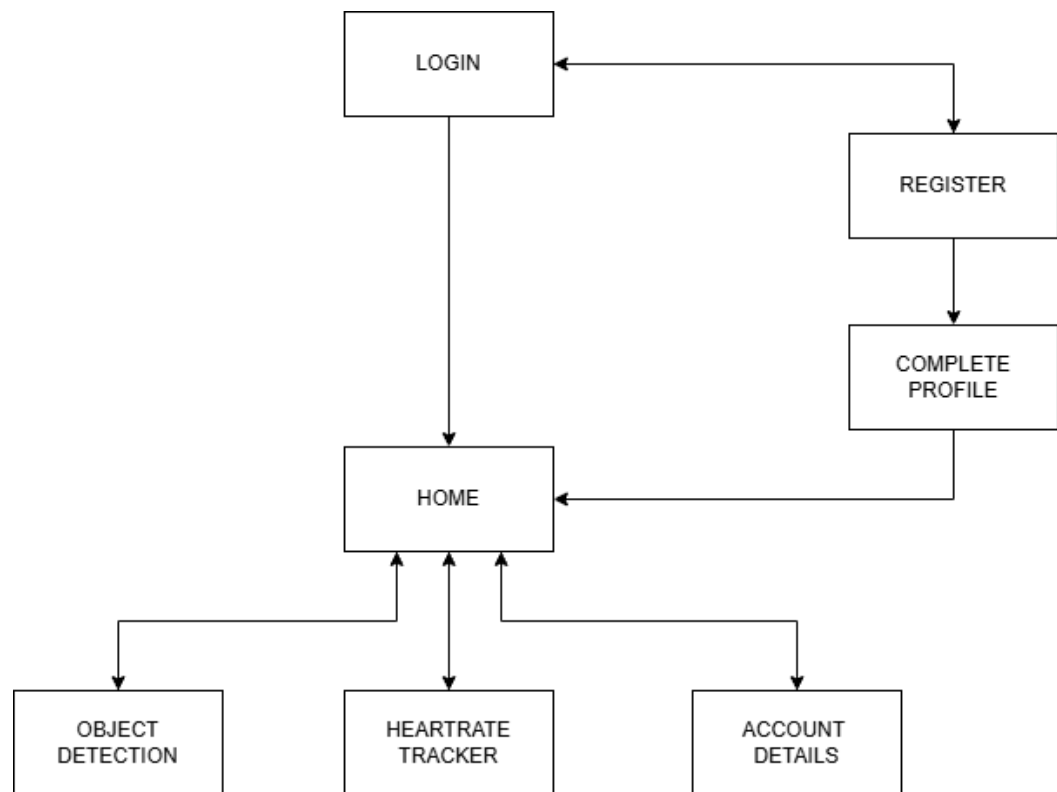
Begin Tracking



It's important to note that these designs - while very, very close to the actual application as you can see in the demo - do have some slight differences in detail and placements.

These are also available on the provided drive.

2.2. Application Flow



The login page is what the user first encounters, and it contains a prompt to register if the user doesn't have a pre-existing account. The register page also has the same prompt to go back to the login page, in case the button was clicked on accident.

Upon registering, the user is asked to complete their profile with a few extra details, most notably emergency contact details as well as the user's address and age.

The home page has a profile icon in the app bar, which displays the details entered earlier, and options to log out or remove the current account. It also features a widget welcoming the user and showing the current date and time, a 'last message' display for the last readings of the object recognition and heart rate sensors. Lastly, a 'quick actions' tab allows the user to jump straight into either measuring their heart rate or checking out what objects their camera is detecting.

The heart rate tracker showcases a simple real-time line chart, and a message based on the reading. If the heart rate is elevated, it tells the user to find a place to rest.

The object recognition tab is a simple page, listing the objects detected and the time they were seen by the camera.

2.3. Code Specifics & Thoughts

The easiest part was probably the MQTT connection, since we've already experimented with it in a previous assignment. However, one thing I've noticed is a problem with subscribing to two topics simultaneously; messages begin to be received two at a time. Otherwise, sending and receiving messages was simple thanks to the MQTT_client package.

Firebase was slightly trickier. Logging a user in and out was unproblematic, as well as removing an account altogether. The trick was saving and fetching the user details. This is where Firestore came in handy; a collection of users was made, to which the given details were saved. When retrieving, a query was made with the user's current email, and the documents returned by the query were processed as individual strings, or user details.

```
final String? email =
  FirebaseAuth.instance.currentUser?.email;
CollectionReference collref =
  FirebaseFirestore.instance
    .collection("Users");
await collref.add({
  "Age": age,
  "Address": address,
  "Emergency Name": emergencyname,
  "Emergency Phone": emergencyphone,
  "Name": name,
  "Phone": phone,
  "Email":email,
});
Navigator.pushAndRemoveUntil(
  context,
  PageTransition(
    type: PageTransitionType.rightToLeft,
    duration:
      const Duration(milliseconds: 200),
    reverseDuration:
      const Duration(milliseconds: 200),
    child: const Home(),
  ), // PageTransition
  (route) => false,
);
},
```

```

_getusername() async {
  try {
    // Get the current user's email
    String? user = FirebaseAuth.instance.currentUser?.email;
    print(user);
    if (user == null) {
      print("User email is null");
      return;
    }
    // Query Firestore for the user's document based on email
    QuerySnapshot querySnapshot = await FirebaseFirestore.instance
      .collection("Users")
      .where("Email", isEqualTo: user)
      .get();
    if (querySnapshot.docs.isEmpty) {
      print("No documents found for the user email");
      return;
    }
    // Process the documents returned by the query
    querySnapshot.docs.forEach((doc) {
      name = doc['Name'] as String;
      age = doc['Age'] as String;
      address = doc['Address'] as String;
      phone = doc['Phone'] as String;
      emergencyname = doc['Emergency Name'] as String;
      emergencyphone = doc['Emergency Phone'] as String;
      print("Saved all the data");
    });
  } catch (e) {
    print("An error occurred: $e");
  }

  setState() {
    name = name;
    age = age;
    address = address;
    phone = phone;
    emergencyname = emergencyname;
    emergencyphone = emergencyphone;
  });
}

```

For the heart rate tracker, I contemplated using the sync-fusion package, before settling on FI-Chart. After trying to test out smaller examples, it finally came together rather nicely.

Lastly, each of the heart rate tracker and object detection tabs include 'begin tracking' buttons. I cannot stress how important those were – not including them would lead to a user initialization error; the system would try to subscribe to the topic before the user was even finished initializing. Annoying, yes, but unfortunately necessary. And it did add to the suspense of waiting to see if it actually works.

That's about it for the application. Reminder that the code is well commented and is provided fully on the drive as well as on the provided GitHub repo link.

3. Maquette Design

Parts Used:

- ESP-32
- ESP-CAM
- Ultrasonic Sensor
- Heart rate Sensor
- Speaker Module
- Amplifier
- Power Source

Following much of the idea of a regular cane, a carbon fiber stick was outfitted with two 3D-printed components:

- A component box with a clear opening for the ultrasonic sensor's transmitter/receiver, and an acrylic lens cover for the camera. The back of the box has convenient holes for the speaker audio.
- A handle at the top of the stick with a slot for the heart rate sensor for easy access.
- Wires from the heart rate sensor travel down the carbon fiber stick and through the box - connecting all components to the power source.

This gave much of the look and feel of a regular walking cane. With the weight of the box averaging around 350 grams with the components included and wired, it may seem a bit on the heavier side, but the carbon fiber's lightness makes up for it.

The length of the stick is also very appropriate, resting at a comfortable 80 cm.

Overall, actually navigating with the stick wasn't too odd of an experience.

4. Logic, ESP32 and Sensor Circuitry

Two different code files were burned onto each of the ESPs.

The ESP-CAM had one sole purpose: identifying the object in front of it and relaying it to the broker. The regular ESP32 handled the rest of the workload, managing the flow between the speaker and its amplifier, the heart rate and ultrasonic sensors, as well as the messages received from the ESP-CAM through the broker.

The flow of the system was as follows:

- The heart rate sensor continuously provides data whenever the finger is placed near the sensor, data that is fed directly to the broker, and by extension, the application whenever the user switches to the heart rate tracker tab.
- The ultrasonic sensor data is handled in-house without resorting to the broker: whenever any object is deemed too close (by calculating the distance from the waves' duration of travel) the user is directly notified via the speaker. This is also done to avoid any delays.
- The speaker is also responsible for the incoming data from the ESP-CAM, announcing the detected object out loud to alert the user.
- To help make the speaker louder, an amplifier is attached.
- Meanwhile, the user can access the ESP-CAM data and the heart rate sensor via the application with the help of the MQTT Protocol, as well as access their data and easily login/logout with the help of Firebase and the Firestore.

5. Computer Vision Using the ESP-CAM

The computer vision model was slightly tricky, given that we first needed to upload the model on the ESP-CAM itself by transforming the model into a *tf*lite file and then into a C array to optimize the storage as much as possible and make it readable for the ESP.

However, even after optimization the file was still too large for the ESP, especially that we're using the YOLO pre-trained model. And that is when we got a better idea to make third-eye a reality: we decided to use the ESP-CAM only as a camera that captures pictures by uploading code that connects it directly to the Internet and provides a URL to the camera, after which we embed the camera's URL with python code utilizing the pre-trained model YOLOV8 imported it from the *ultralytics* library. This was used to detect objects, and the code is supposed to stream every label it detects to the RTDB.

The MQTT broker fetches the label and sends it to the flutter app as well as the speaker, so it can warn the user of their surroundings.

Generally speaking, the code wasn't complicated, it was an issue of library dependencies and which version of each library works with which.

```
import cv2
from ultralytics import YOLO
import urllib.request
import numpy as np
import time
from firebase_admin import credentials
from firebase_admin import db
import firebase_admin

# URL of the camera
url = 'http://192.168.105.68/cam-hi.jpg'
#firebase credentials
firebase_cred = {
    "type": "service_account",
    "project_id": "finalproject-15cde",
    "private_key_id": "9beb4e351ba56574cd2d24a0ef9690c0a133695c",
    "private_key": "-----BEGIN PRIVATE KEY-----\nMIIEvgIBADANBgkqhkiG9w0BAQEFAASCBKggSwAgEAAoIBAQC0zUyeHU4N2lpo\nbH2bnvHsVJ8pAeSUAN\n",
    "client_email": "firebase-adminsdk-5scc0@finalproject-15cde.iam.gserviceaccount.com",
    "client_id": "112552998265671227150",
    "auth_uri": "https://accounts.google.com/o/oauth2/auth",
    "token_uri": "https://oauth2.googleapis.com/token",
    "auth_provider_x509_cert_url": "https://www.googleapis.com/oauth2/v1/certs",
    "client_x509_cert_url": "https://www.googleapis.com/robot/v1/metadata/x509/firebase-adminsdk-5scc0%40finalproject-15cde.iam.gserv\n",
    "universe_domain": "googleapis.com"
}

cred = credentials.Certificate(firebase_cred)
firebase_admin.initialize_app(cred, {"databaseURL": "https://finalproject-15cde-default-rtdb.europe-west1.firebaseio.com/"})
# creating reference to root node
ref = db.reference("/")
```

We started by importing all the necessary libraries. Then, after uploading the streaming code on the ESP-CAM and getting the URL, we copy paste it to our model to be able to access the camera, on the other hand, which is the RTDB, we generate a private key which gives us a JSON file containing all the credentials we need to access the RTDB in addition to the URL of the RTDB.

Using the *firebase_cred* variable that contains the credentials to create a certificate object, we initialize the app using the *cred* variable that holds the certificate object and the URL of the RTDB, so our app is now ready to interact with the RTDB.

The following implemented functions are a core part of the code:

1. *Predict and Detect*

```
def predict_and_detect(chosen_model, img, conf=0.5):  
    img = cv2.rotate(img, cv2.ROTATE_90_CLOCKWISE)  
    results = chosen_model.predict(img, conf = conf)  
    print(f"Predict and Detect function output: {type(results)}")  
    return img, results
```

This function takes three arguments: the chosen model (in this case, YOLOV8), the image and the confidence. We start by rotating the image ninety degrees clockwise due to its position in the component box, then we predict the objects by passing the image and the confidence to the predict functions. It then returns the rotated image and the results.

2. *Extract Prediction Text*

```
def extract_prediction_text(results):  
    prediction_texts = []  
    for result in results:  
        for box in result.bboxes:  
            prediction_texts.append(result.names[int(box.cls[0])])  
    return prediction_texts
```

This function is used to get the text label from the predictions. Creating an empty list for the prediction texts, we loop over the results and each bounding box in the results, using the names attribute on the index of the predicted class to get our desired label and add it to our list. This list is returned after looping through all the bounding boxes of all predictions.

3. Send to RTDB

```
def send_to_RTDB(results):
    detection_results = results[1]
    text = str(extract_prediction_text(detection_results))
    db.reference("/ESP").push().set({"label": text})
```

The last key function to be implemented is the one responsible for sending the label list to the RTDB. We get the detection results from the second index of the results variable; the first index holds the rotated image, while the second holds the actual output. We pass this to the extract prediction text function previously implemented and cast the result to a string. This way, we can send it to Firebase. We push the label to the RTDB under the reference label /ESP and assign the label key to all the labels sent.

4. Main Loop

```
while True:
    try:
        #to Capture images from the camera URL
        img_resp = urllib.request.urlopen(url)
        imgnp = np.array(bytearray(img_resp.read()), dtype=np.uint8)
        im = cv2.imdecode(imgnp, -1)

        if im is None:
            print("Error: Could not capture image.")
            break

        result_img, detection_results = predict_and_detect(model, im , conf=0.5)

        cv2.imshow('Object Detection', result_img)
        send_to_RTDB((result_img, detection_results))

        time.sleep(2)

        if cv2.waitKey(5) & 0xFF == 27:
            break

    except urllib.error.URLError as e:
        print(f"Failed to connect to the camera: {e.reason}")
        break
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
        break

cv2.destroyAllWindows()
```


In the main loop, we implement a try-catch clause. We start by making an HTTP request to get access to the camera using the camera's URL and save it in a variable *img_resp* from which we will be able to read the data.

Then, we make a NumPy array using the data inside the *img_resp* variable cast into a byte array, casting that into an int8 NumPy array. We decode the image into a format that OpenCV can work with using the `cv2.imdecode(imp, -1)` function, before making an if-condition to check if the item variable is null.

If it is null, a message is printed that we couldn't capture any images.

Then we start the most important part of the loop, which is getting the predictions. We get the result image and the detection results from the predict and detect function, then display the result image that we got, and pass the detection results to the RTDB using the Send to RTDB function.

We use the sleep function to send detections to the RTDB every 2 seconds, after trying multiple streaming speeds, streaming each 2 seconds was the perfect choice.

We also make an if-condition using the `cv2.waitKey(5) & 0xFF == 27` which checks if the escape button was pressed to exit the loop.

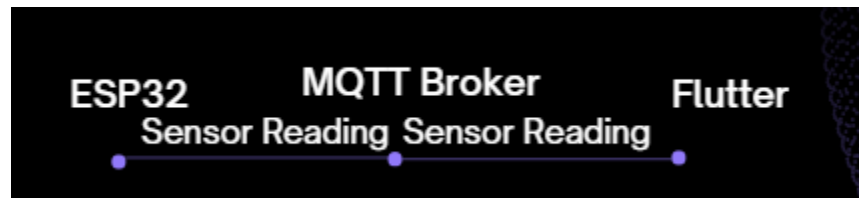
Finally, the catch condition throws an exception if the URL didn't give access to the camera so it gives a warning that we couldn't open the camera, or another exception if any other unexpected error occurred.

6. MQTT and System Lifecycle

This is where the cloud connections really shone through and came into play. We first needed to identify the needed connections.

6.1. Sensor Readings

To send sensor readings to and from the ESP32, we naturally used the HiveMQ MQTT broker and pretty much programmed it using the sample code that we covered earlier in the training with only replacements in the user credentials, therefore not many notes should be considered.



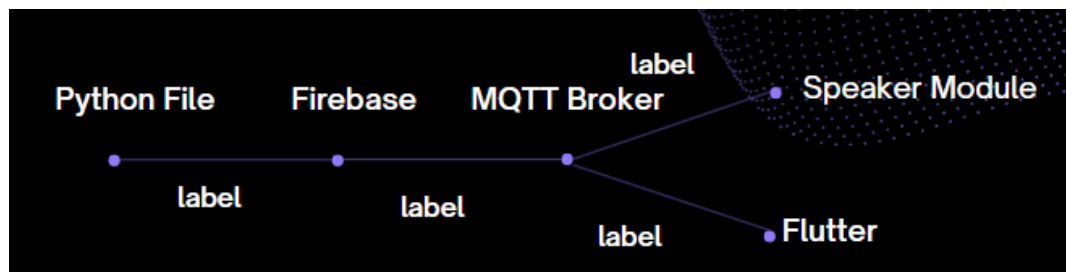
6.2. User Authentication

Secondly, we needed a database to hold the users of the Flutter application and authenticate them, and Firebase came to the rescue. After setting up the project to be a flutter/mobile application, adding the dependencies to the pubsec.yaml file was straightforward.

Yet, we still needed a way to fetch the user's data to be able to display it on the home page and account details page, since the authentication in Firebase only saved the email address, we had to use another solution inside Firebase itself, which was the Firestore. After creating what is essentially called a Firestore database we began adding the wanted fields and getting them using some libraries. Note that the section of the code was discussed above in the Flutter application section.

6.3. Labels for Computer Vision

The trickiest connection had to have been sending the label from the Python file which held what the ESP-CAM thinks the object is to the Flutter Application and the Speaker Module. After some research, we decided to send the data to a Firebase Realtime Database which will then be sent to the MQTT broker so it can be sent to the Speaker Module and the Flutter Application at the same time.



7. Results and Conclusion

The system lifecycle came together rather nicely and had fairly good results, although with room for improvement.

However, the goals in linking several technologies and frameworks in the form of mobile application development, cloud technologies, the MQTT protocol and additional measures such as computer vision and the ESP-CAM were most definitely achieved.