CSE476 – MOBILE COMMUNICATION NETWORKS

TERM PROJECT REPORT

aIoT

AI and IoT Based Smart House System

Kerem Göbekcioğlu

Diyar İsi

Ahmet Hamza Şiyak

Github Repository

https://github.com/diyarrr/aIot

Demo Video

https://www.youtube.com/watchv=_OYygv8Y9Ik

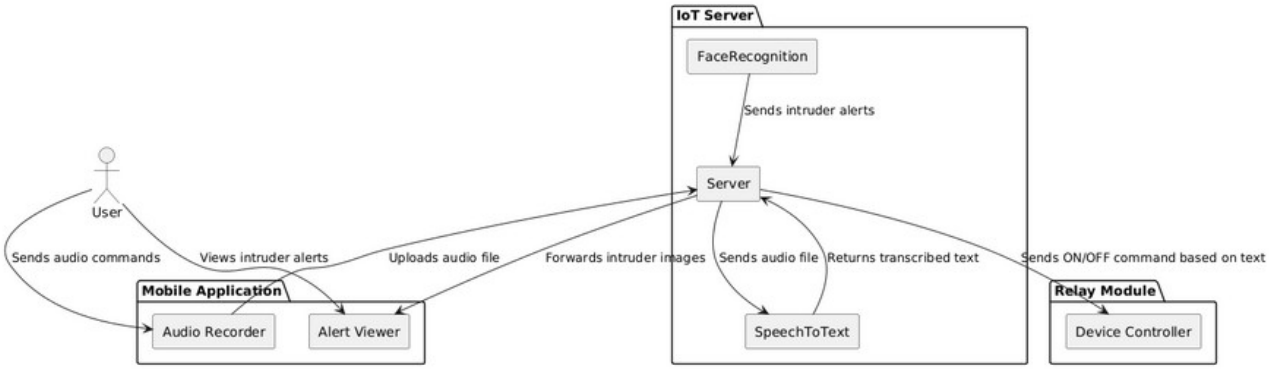
Project Goal and Overview

The goal of this project is to create an IoT-based system that combines different technologies to make life easier and more secure. The system connects three main parts: a mobile application, an IoT server, and a relay module. Together, these parts work to provide smooth communication, process data, and control devices effectively. The project includes several important components that work together to achieve its purpose.

Components of the System

1. **IoT Server**:
    - **Speech Processing Service**:
        - Converts audio data from the mobile application into actionable text using speech recognition techniques.
        - Generates commands based on the recognized text, such as "Turn on the lamp," and sends these commands to the relay module for device control.
    - **Face Recognition Service**:
        - Monitors live video feeds to detect and identify faces.
        - Sends images and alerts to the mobile application when unknown individuals are detected.
    - Acts as the core processing unit, integrating data from multiple sources and managing communication between components.
2. **Relay Module**:
    - Receives ON/OFF commands from the server to control connected devices, such as a lamp.
    - Ensures real-world actions based on the processed commands from the IoT server.
3. **Mobile Application**:
    - Acts as the interface for user interaction with the system.
    - Sends audio files (voice commands) to the server for processing.
    - Receives alerts and images of intruders detected by the face recognition service for enhanced security monitoring.

System Architecture(Component Diagram)



1) IoT Server

The IoT server in this project is a system that combines audio processing, speech recognition, and face recognition to automate tasks and enhance security. It uses Python and JavaScript to convert speech into text and detect faces, making it a reliable tool for handling audio and video data. The server runs on a Raspberry Pi, which is controlled remotely via SSH, allowing for easy management and configuration.

System Design

The system design of the IoT server consists of two core components, each responsible for distinct functionalities that collectively form a cohesive and efficient system:

1. **Speech Processing Service**: This component is designed to handle audio data by converting various audio file formats into `.wav` format and subsequently performing speech-to-text conversion using a robust speech recognition library. It processes user-uploaded audio files, extracts meaningful transcriptions, and provides the output for further application needs.

2. **Face Recognition Service**: This component focuses on identifying individuals in video streams. It utilizes pre-encoded reference images to detect and recognize faces in real-time. When unknown individuals are detected, the system generates alerts, ensuring timely responses to potential security concerns.

1) Speech Processing Service

Key Modules:

1. **AudioHandler**:

```
const { spawn } = require('child_process');

function processAudio(filePath, callback) {
  console.log('Processing audio file:', filePath);

  // Spawn a child process to run the Python script
  const pythonProcess = spawn('python3', ['speech_to_text.py', filePath]);

  let transcription = '';

  // Capture standard output from the Python process
  pythonProcess.stdout.on('data', (data) => {
    transcription += data.toString();
  });

  // Capture standard error output
  pythonProcess.stderr.on('data', (data) => {
    console.error('Python error:', data.toString());
  });

  // Handle process exit
  pythonProcess.on('close', (code) => {
    if (code !== 0) {
      callback(new Error('Error processing audio'));
    } else {
      console.log('Transcription:', transcription.trim());
      callback(null, transcription.trim());
    }
  });
}

module.exports = processAudio;
```

- Manages the interaction between the Node.js backend and the Python script for audio processing.
- Spawns a child process to execute the Python script and retrieves the transcription.
- Handles standard output and error streams for the Python process to ensure smooth communication.

2. **SpeechToText (Python)**:

```python
import sys
from pydub import AudioSegment
import speech_recognition as sr
import os

# convert file format to .wav
def convert_to_wav(file_path):
    audio = AudioSegment.from_file(file_path)
    wav_path = "converted.wav"
    audio.export(wav_path, format="wav")
    return wav_path

# convert speech to text
def transcribe_audio(file_path):
    recognizer = sr.Recognizer()
    with sr.AudioFile(file_path) as source:
        audio_data = recognizer.record(source)

    try:
        text = recognizer.recognize_google(audio_data)
        print(text)  # Print to stdout for Node.js
    except sr.UnknownValueError:
        print("Error: Could not understand the audio")
    except sr.RequestError:
        print("Error: Could not request results from the speech recognition service")
    finally:
        # Clean up the converted file
        if os.path.exists(file_path):
            os.remove(file_path)

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Error: No file path provided")
        sys.exit(1)

    input_file_path = sys.argv[1]
    converted_file_path = convert_to_wav(input_file_path)
    transcribe_audio(converted_file_path)
```

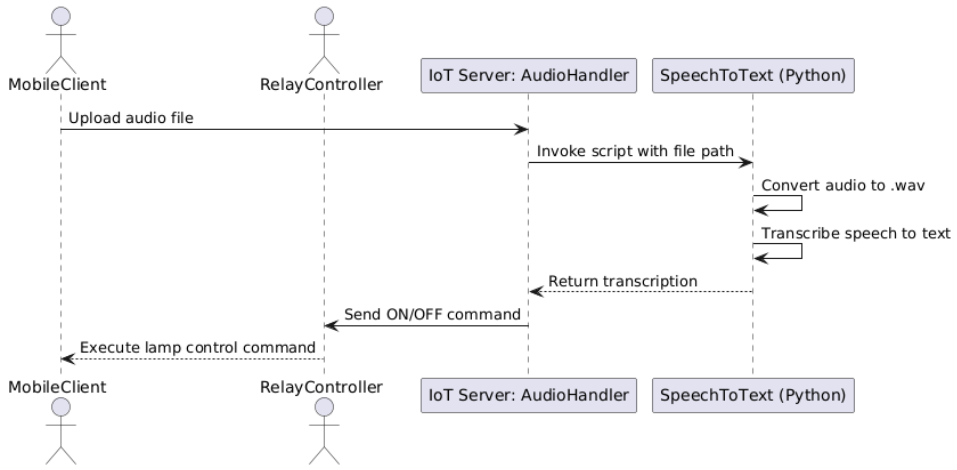
   - Converts audio files to `.wav` format using the `pydub` library to standardize input.
   - Utilizes `speech_recognition` to transcribe the audio into text.
   - Implements error handling for cases where the audio cannot be understood or the speech recognition service fails.

Workflow:

1. Audio files are uploaded by the mobile client to the server.
2. The `AudioHandler` module invokes the Python script (`speech_to_text.py`) with the file path as an argument.
3. The Python script:
   - Converts the audio file to `.wav`.

- Transcribes the audio using Google Speech Recognition.

4. The transcription result is returned to the server.
5. Based on the transcription, the server sends an ON/OFF command to the relay controller (e.g., "Turn on the lamp").



2) Face Recognition Service

Key Modules:

1. **FaceRecognitionUtils**:

```
import face_recognition


# encode the given images
def load_images(reference_images):

    img_encodings = {}
    for name, image_path in reference_images.items():
        image = face_recognition.load_image_file(image_path)
        face_encoding = face_recognition.face_encodings(image)[0]
        img_encodings[name] = face_encoding

    return img_encodings

# take the video capture image and compare with the database images
def recognize_faces(frame, reference_face_encodings):
    face_locations = face_recognition.face_locations(frame)
    face_encodings = face_recognition.face_encodings(frame, face_locations)

    recognized_faces = []
    for face_location, face_encoding in zip(face_locations, face_encodings):
        matches = face_recognition.compare_faces(list(reference_face_encodings.values()), face_encoding)
        name = 'Unknown'
        if True in matches:
            matched_index = matches.index(True)
            name = list(reference_face_encodings.keys())[matched_index]
        recognized_faces.append((face_location, name))
    return recognized_faces
```

- Loads and encodes reference images (e.g., known individuals).

- Detects and recognizes faces in video frames by comparing them with the reference encodings.
- Returns the names of recognized individuals or 'Unknown' for unrecognized faces.

2. **VideoCaptureUtils**:

```
import cv2


def start_video_capture(camera_index=0):
    video_capture = cv2.VideoCapture(camera_index)
    return video_capture

def release_video_capture(video_capture):
    video_capture.release()
    cv2.destroyAllWindows()
```

- Manages the video capture process using OpenCV.
- Initializes the camera and ensures proper cleanup after processing.

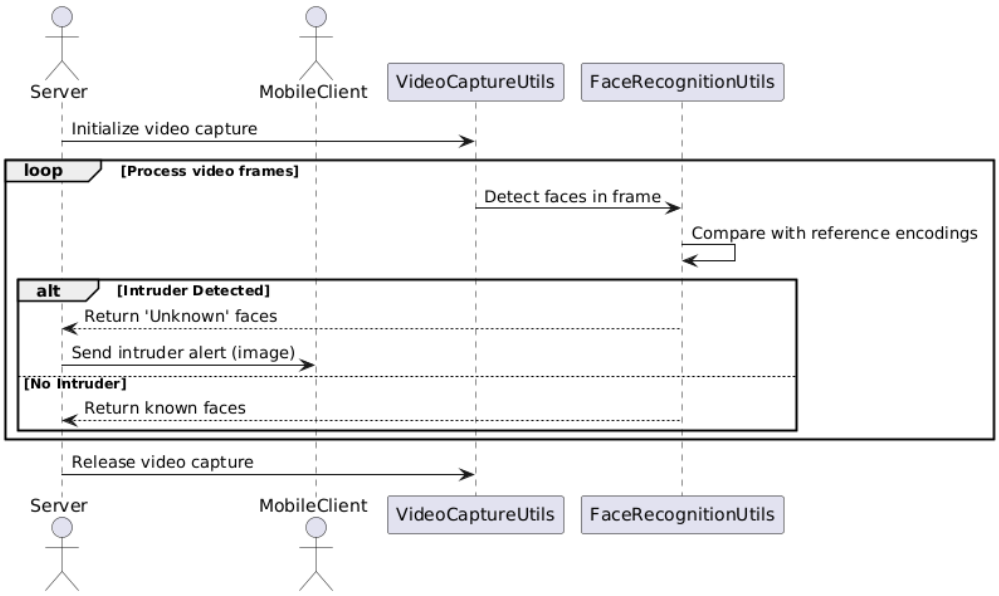
3. **MainFaceRecognition**:

```python
import cv2
import face_recognition_utils as fr_utils
import video_capture_utils as vc_utils
import requests
import base64

# Load reference images
reference_images = {
    'Diyar': 'images/diyar.jpg',
}

# Load reference face encodings
reference_face_encodings = fr_utils.load_images(reference_images)

# Initialize video capture
video_capture = vc_utils.start_video_capture()


def send_intruder_alert(frame, server_url):
    # Convert the frame (image) to JPEG format and encode it to base64
    _, buffer = cv2.imencode('.jpg', frame)
    image_base64 = base64.b64encode(buffer).decode('utf-8')

    try:
        response = requests.post(f'{server_url}/log-detection', data=image_base64, headers={'Content-Type': 'application/octet-stream'})
        if response.status_code == 200:
            print("Intruder alert sent successfully.")
        else:
            print(f"Failed to send intruder alert. Status code: {response.status_code}")
    except Exception as e:
        print(f"Error sending intruder alert: {e}")


# Process video frames for face recognition
while True:
    ret, frame = video_capture.read()
    if not ret:
        break

    # Get recognized faces in the current frame
    recognized_faces = fr_utils.recognize_faces(frame, reference_face_encodings)

    # Check if there is at least one person and all are unknown
    if recognized_faces:  # Ensure at least one face is detected
        intruder_detected = all(name == 'Unknown' for (_, _, _, _), name in recognized_faces)
    else:
        intruder_detected = False  # No faces detected, no intruder

    # Debug output
    print(f"Intruder Detected: {intruder_detected}")

    # Send alert if an intruder is detected
    if intruder_detected:
        send_intruder_alert(frame, "http://localhost:3000")

    # Draw rectangles and labels on the frame
    for (top, right, bottom, left), name in recognized_faces:
        cv2.rectangle(frame, (left, top), (right, bottom), (0, 255, 0), 2)
        font = cv2.FONT_HERSHEY_DUPLEX
        cv2.putText(frame, name, (left + 6, bottom - 6), font, 0.5, (255, 255, 255), 1)

    #cv2.imshow('Face Recognition', frame)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Release video capture and close windows
vc_utils.release_video_capture(video_capture)
```

- Processes each video frame for face detection and recognition.
- Sends alerts to the mobile client if an intruder is detected.
- Draws rectangles around detected faces and labels them in the video stream for debugging or visualization purposes.

Workflow:

1. The video feed is captured using `VideoCaptureUtils`.
2. For each frame:
    - Faces are detected, and their encodings are compared against the reference encodings using `FaceRecognitionUtils`.
    - If unrecognized faces are detected, an alert (including the frame image) is sent to the mobile client.
3. The system continues to monitor the video feed until the process is stopped (e.g., by pressing `q`).



2) Relay Module(ESP8266 Wi-Fi Relay)

The purpose of this module is to control a lamp using voice commands from a mobile device. The system receives commands from the server and controls the relay to turn the lamp on or off.

Hardware Structure

The module is designed using the **ESP8266 Wi-Fi Relay Module**, which integrates an **ESP8266 Wi-Fi chip** and a relay on the same board. The operational details are as follows:

- **ESP8266 Module**: Establishes a Wi-Fi connection and communicates with the server to process received commands.
- **Relay**: Controls the flow of mains electricity to a lamp, enabling the lamp to be turned on or off.
- **Power Supply**: The board operates with a DC power source ranging between 5-12V. The power source provides energy to both the ESP8266 module and the relay.

The relay manages the high-voltage electricity from the mains, allowing it to control a physical device, such as a lamp, effectively. The ESP8266 module fully manages the relay's operation.

Wi-Fi Connection

The module connects to a local Wi-Fi network using **SSID and password** credentials. This connection facilitates data exchange between the server and the ESP8266. If the Wi-Fi connection fails, there is no backup or security mechanism, as the system is designed to function only within the local network and does not require additional security measures.

Functionality and Communication

The module connects to a local Wi-Fi network using **SSID and password** credentials. This connection facilitates data exchange between the server and the ESP8266. If the Wi-Fi connection fails, there is no backup or security mechanisThe ESP8266 module processes commands received from the server via POST requests at the /relay endpoint. The processing steps are as follows:

```
// POST isteği ile röle kontrolü
server.on("/relay", HTTP_POST, [](AsyncWebServerRequest *request){},
  NULL,
  [](AsyncWebServerRequest *request, uint8_t *data, size_t len, size_t index, size_t total) {
    String body = String((char*)data); // Gelen veriyi al
    Serial.println("Gelen POST isteği:");
    Serial.println(body);

    // JSON verisini ayrıştır
    StaticJsonDocument<200> jsonDoc;
    DeserializationError error = deserializeJson(jsonDoc, body);

    if (error) {
      Serial.println("JSON ayrıştırma hatası!");
      request->send(400, "application/json", "{\"error\":\"Invalid JSON\"}");
      return;
    }

    // Röle ve durum bilgilerini al
    int relay = jsonDoc["relay"];
    String state = jsonDoc["state"];

    if (relay > 0 && relay <= NUM_RELAYS) {
      if (state == "ON") {
        digitalWrite(relayGPIOs[relay - 1], RELAY_NO ? LOW : HIGH); // Röleyi aç
        Serial.println("Röle Açıldı");
      } else if (state == "OFF") {
        digitalWrite(relayGPIOs[relay - 1], RELAY_NO ? HIGH : LOW); // Röleyi kapat
        Serial.println("Röle Kapandı");
      } else {
        request->send(400, "application/json", "{\"error\":\"Invalid state\"}");
        return;
      }
      request->send(200, "application/json", "{\"success\":\"Röle güncellendi\"}");
    } else {
      request->send(400, "application/json", "{\"error\":\"Invalid relay number\"}");
    }
  });
```

1. The module receives a POST request and extracts its content.
2. It parses the incoming data in JSON format, expecting the following fields:

- relay: The relay number (e.g., 1).
- state: The desired state of the relay ("ON" or "OFF").

3. If the JSON format is invalid, the module responds with an error:

```
{
    "error": "Invalid JSON"
}
```

4. If the relay number and state are valid, the module processes the command:

- For "ON", it activates the relay.
- For "OFF", it deactivates the relay.

5. After successfully processing the command, the module sends a response to the server:

```
{
    "success": "Relay updated"
}
```

Testing Process and Challenges

During the design and development of the module, various tests were conducted to ensure the system worked as expected. This testing process involved verifying the programming of the module and its control of the relay, as well as addressing technical challenges encountered during implementation. Below is a detailed explanation of the process and the solutions applied to overcome the challenges:

Programming and Connection Process

The ESP8266 Wi-Fi Relay Module used in this project did not have a direct interface for connecting to a computer. Therefore, an **Arduino Nano** board was used as a bridge to program the module:

1. **Using the Arduino Nano as a Bridge**:
   - The **RESET** pin and **GND** pin of the Arduino Nano were connected to disable its active mode, making it act solely as a bridge.
   - The Arduino Nano’s **RX** pin was connected to the module’s **RX** pin, and its **TX** pin to the module’s **TX** pin. This allowed programming the ESP8266 via serial communication.
2. **Entering Flash Mode**:
   - To upload code, the ESP8266 module had to be set in **flash mode**. This was achieved by connecting the GPIO0 pin of the module to GND, enabling the module to enter programming mode.
   - Once in flash mode, the code was successfully uploaded.

Power Issues and Solution

One of the major challenges encountered was providing sufficient power to the ESP8266 module:

1. **Power During Programming**:
   - The 3V power supplied by the Arduino Nano was sufficient to operate the ESP8266 module during the programming (flash) mode.
   - Code upload was successfully performed under these conditions.
2. **Power Issues in Working Mode**:
   - After the programming was completed, the module was taken out of flash mode and set to normal operating mode. However, the 3V power supply was insufficient to sustain the module in this mode.
   - As a result, the module could not function properly, and the relay control could not be tested.
3. **Solution: External Power Supply**:
   - A **9V DC power supply** was connected externally to provide sufficient power to the module during normal operation.
   - This resolved the power issue, enabling stable operation of the module and allowing relay control to be tested effectively.

Testing Stages

The functionality of the module was tested through different scenarios:

1. **Wi-Fi Connection**:
   - The module successfully connected to the local Wi-Fi network using the predefined SSID and password, obtaining a valid IP address.
   - The status of the Wi-Fi connection was monitored and verified through the serial monitor.

2. **Relay Control**:

- POST requests from the server were tested, and the relay responded to the commands in the JSON payload:
  - "relay": 1 and "state": "ON" turned the relay on.
  - "relay": 1 and "state": "OFF" turned the relay off.
- The physical status of the relay was observed, with a clicking sound indicating its operation, and the lamp’s state changing accordingly.

3. **Error Scenarios**:

- Invalid JSON data was sent, and the ESP8266 module correctly returned an error response.
- Wi-Fi disconnection scenarios were tested, and the module successfully reconnected to the network.

3) Mobile App

The **Smart House Controller** is an Android mobile application designed to provide smart home functionalities. It enables **real-time intruder detection notifications** and **voice-based room light control**. The app integrates with a **Raspberry Pi 4 server** equipped with a camera for intruder detection and ESP-based devices for controlling lights.
The app uses modern technologies such as:

- **MVVM Architecture** for clean separation of concerns.
- **Hilt** for Dependency Injection.
- **Retrofit** for server communication.
- **NanoHTTPD** for hosting a lightweight HTTP server locally on the mobile device.
- **SharedPreferences** for dynamic IP configuration.

1. Key Features

1. **Intruder Detection**:
   - Receives intruder alerts as Base64-encoded images via a local HTTP server.
   - Displays the image to the user via a notification.
2. **Voice-Controlled Lights**:
   - Records and sends voice commands to the server.
   - The server processes the command and turns the lights on/off using ESP hardware.
3. **Dynamic IP Configuration**:
   - Users input the Raspberry Pi server's IP address.
   - The app stores and reuses the IP address using SharedPreferences.

2. Architecture Overview

The app follows the **MVVM (Model-View-ViewModel)** design pattern to ensure clean code, modularity, and testability.

- **Model**: Contains the repository and data sources.
- **View**: Jetpack Compose screens handle user interaction.
- **ViewModel**: Manages UI state and logic.

The app communicates with the Raspberry Pi server using **Retrofit** and hosts a local HTTP server with **NanoHTTPD**.

3. Technologies Used

Technology	Purpose
MVVM Architecture	Separates UI, logic, and data layers.
Jetpack Compose	Simplifies UI development with Kotlin.
Hilt	Provides Dependency Injection

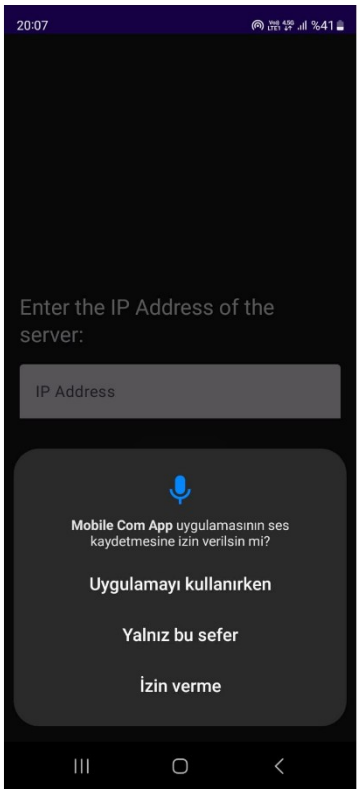
Technology	Purpose
	for modular code.
Retrofit	Handles server communication (HTTP requests).
NanoHTTPD	Hosts a lightweight HTTP server in the app.
SharedPreferences	Saves user-input IP address for persistence.
Coroutines	Manages asynchronous tasks efficiently.

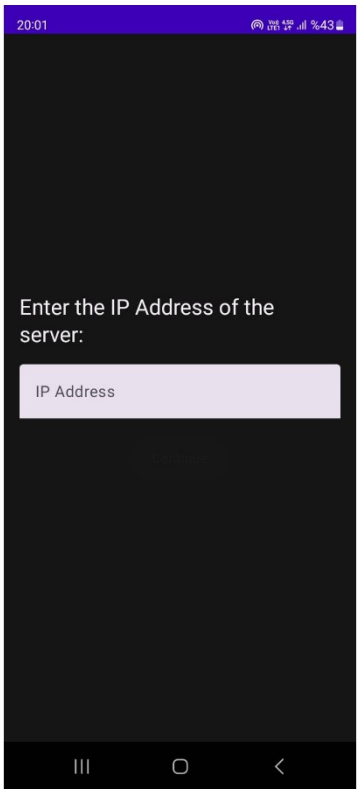
6. Application Flow

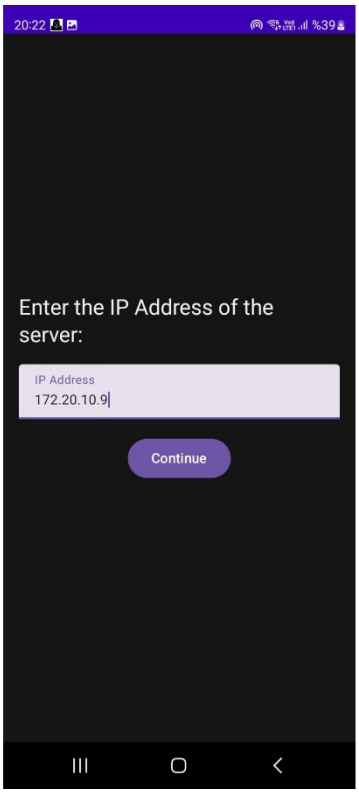
1. **IP Address Screen**
   Users input the Raspberry Pi server's IP address, which is saved for future app usage.
2. **Audio Screen**
   - Users can start/stop recording audio commands and play them.
   - The app sends recorded audio to the server for processing.
   - The server responds with "on/off" commands for controlling room lights.
3. **Intruder Alert**
   - The app hosts a **local HTTP server** (NanoHTTPD) to listen for intruder alerts.
   - The server sends a **Base64 image** to the app.
   - The app decodes the image and shows it via **notifications**.

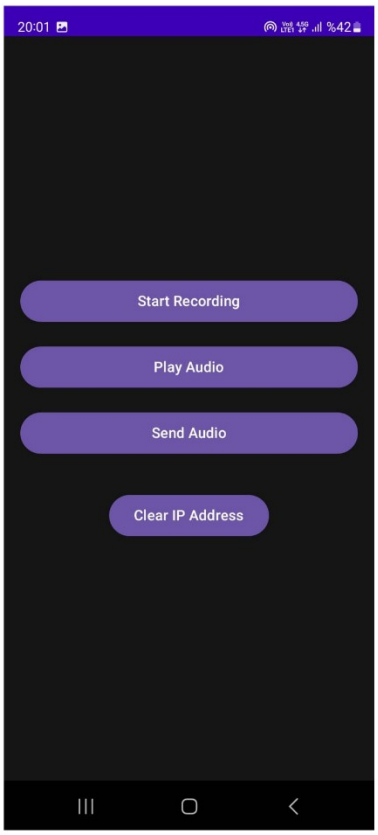
7. App Images

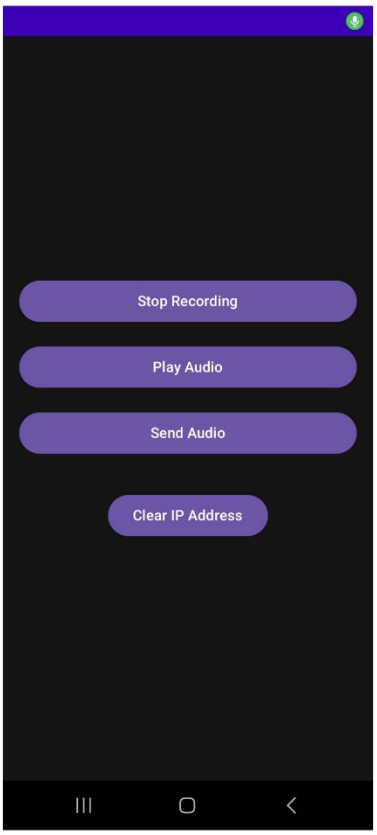
These images demonstrate the flow in our mobile app.

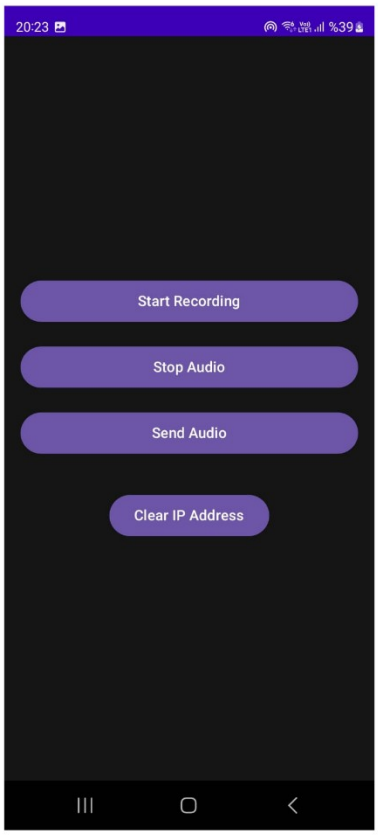

20:07
%41
Enter the IP Address of the server:
IP Address
Mobile Com App uygulamasının ses kaydetmesine izin verilsin mi?
Uygulamayı kullanırken
Yalnız bu sefer
İzin verme

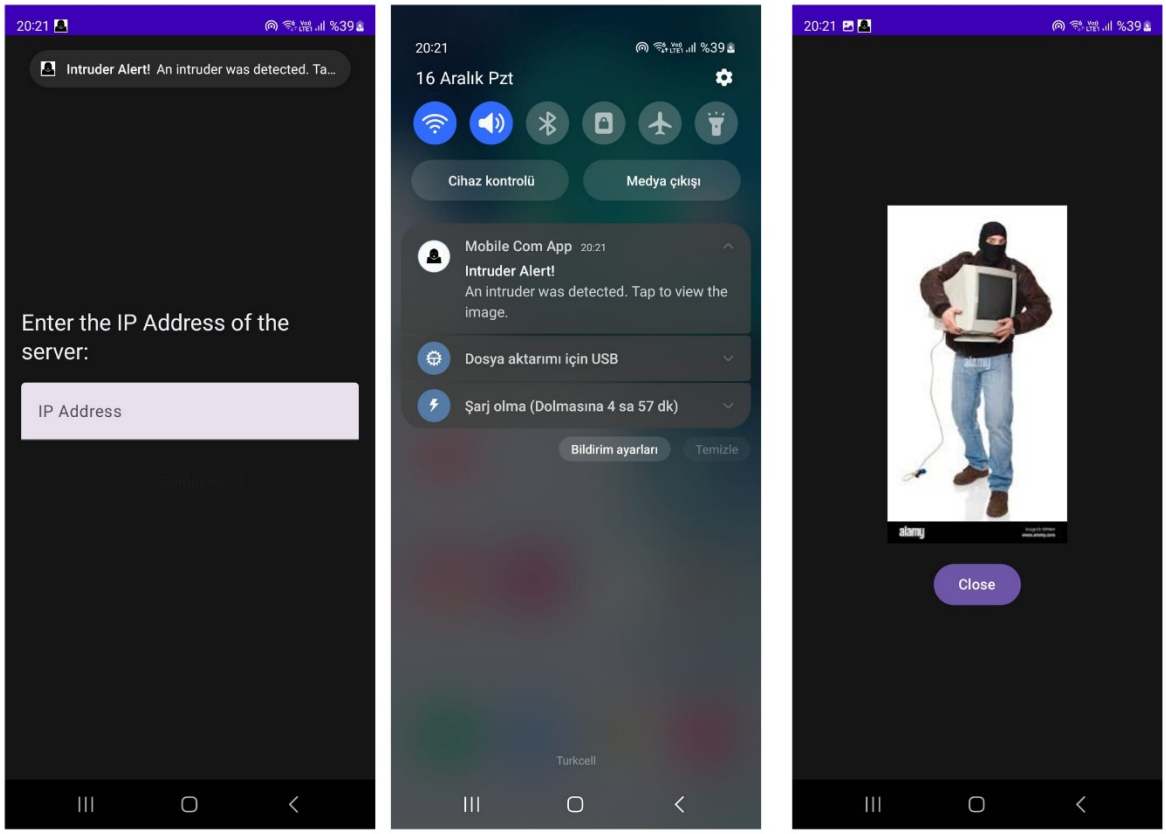

20:01
%43
Enter the IP Address of the server:
IP Address


20:22
%39
Enter the IP Address of the server:
IP Address
172.20.10.9
Continue


20:01
%42
Start Recording
Play Audio
Send Audio
Clear IP Address


Stop Recording
Play Audio
Send Audio
Clear IP Address


20:23
%39
Start Recording
Stop Audio
Send Audio
Clear IP Address



8. Code Implementation

Brief explanation for every class and interface in our mobile app code.

MainActivity: Acts as the entry point for the app and sets up navigation between screens.

IPAddressScreen: Allows the user to input and save the server IP address dynamically.

AudioScreen: Manages audio recording, playback, and sending commands to the server.

IntruderScreen: Displays the received intruder image sent by the server.

AudioViewModel: Handles business logic for the AudioScreen, managing recording and network operations.

AudioState: Represents the UI state for the AudioScreen with fields like isRecording, isPlaying, and errorMessage.

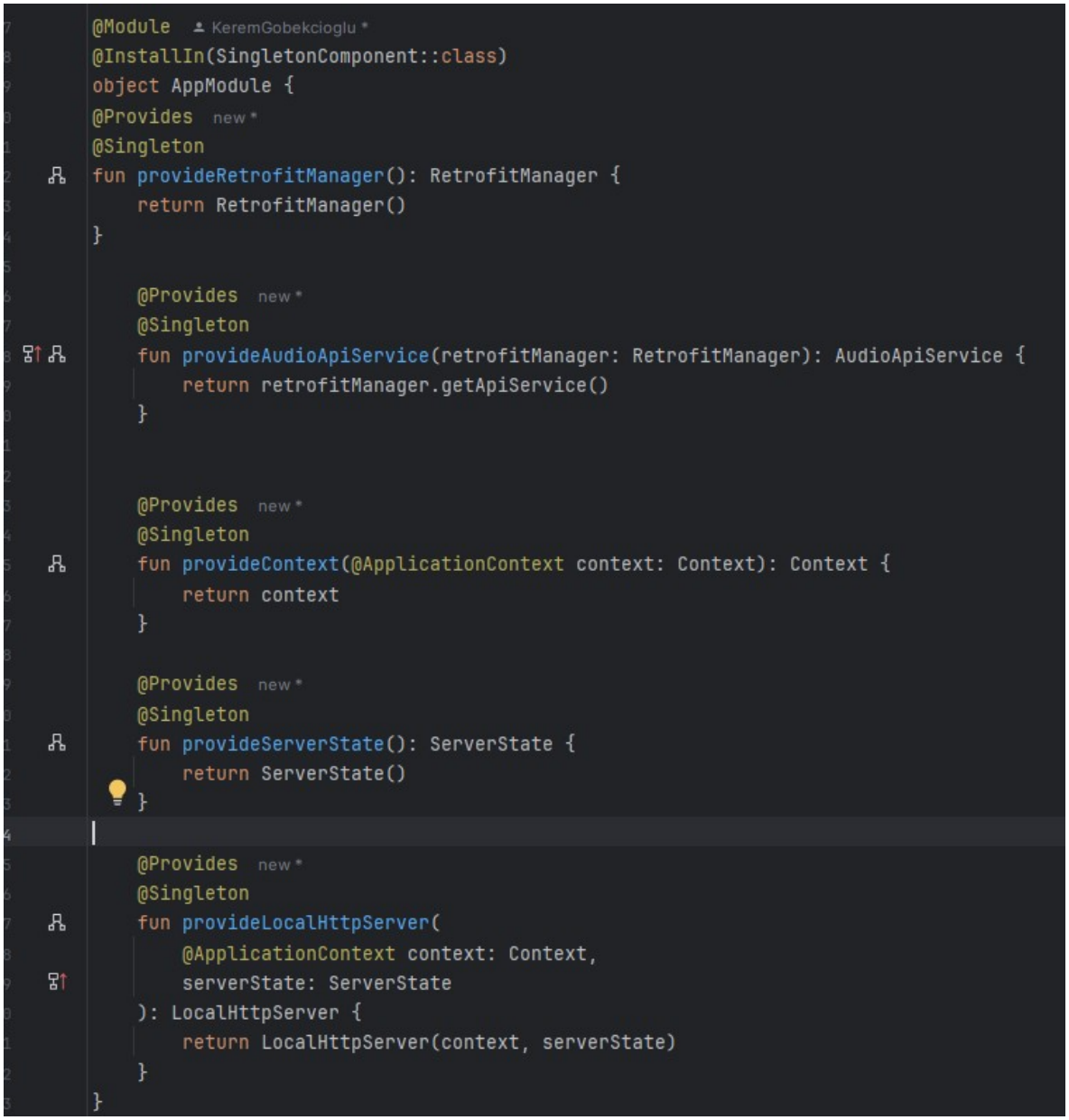
SharedPreferencesIpAddress: Provides utility functions to save, retrieve, and clear the server's IP address using SharedPreferences.

LocalHttpServer: A lightweight HTTP server (using NanoHTTPD) that listens for intruder alerts and handles incoming image data.

AudioRepository: Defines the abstraction for network operations related to audio data (e.g., upload audio, send commands).
AudioRepositoryImpl: Implements the AudioRepository interface using Retrofit for actual server communication.
RetrofitManager: Manages the Retrofit instance and dynamically updates the base URL based on user input.
AppModule: A Hilt module providing dependencies such as RetrofitManager, AudioRepository, and ServerState.
RepositoryModule: A DI module that binds repository implementations (e.g., AudioRepositoryImpl) to their abstractions.
AudioApiService: Retrofit interface defining the HTTP API endpoints for sending audio and on/off commands.
ServerForegroundService: A background service that ensures the local HTTP server stays active in the foreground.
AudioPlayer (Interface): Abstracts audio playback functionality.
AudioRecorder (Interface): Abstracts audio recording functionality.
AudioPlayerImpl: Provides the concrete implementation for audio playback.
AudioRecorderImpl: Provides the concrete implementation for audio recording.
ServerState: Manages the state for server interactions, such as notifying the UI when an intruder alert is received.
Screen: Defines navigation routes for the app screens.

8.1 Dependency Injection (Hilt)

Dependency injection is used for Retrofit, Repositories, and other components.



```
@Module  KeremGobekcioglu *
@InstallIn(SingletonComponent::class)
object AppModule {
@Provides  new *
@Singleton
fun provideRetrofitManager(): RetrofitManager {
    return RetrofitManager()
}

    @Provides  new *
    @Singleton
    fun provideAudioApiService(retrofitManager: RetrofitManager): AudioApiService {
        return retrofitManager.getApiService()
    }


    @Provides  new *
    @Singleton
    fun provideContext(@ApplicationContext context: Context): Context {
        return context
    }

    @Provides  new *
    @Singleton
    fun provideServerState(): ServerState {
        return ServerState()
    }

    @Provides  new *
    @Singleton
    fun provideLocalHttpServer(
        @ApplicationContext context: Context,
        serverState: ServerState
    ): LocalHttpServer {
        return LocalHttpServer(context, serverState)
    }
}
```

```kotlin
@Module  KeremGobekcioglu
@InstallIn(SingletonComponent::class)
abstract class RepositoryModule {
    @Binds  KeremGobekcioglu
    @Singleton
    abstract fun bindAudioRepository(
        audioRepositoryImpl: AudioRepositoryImpl
    ): AudioRepository

    @Binds  KeremGobekcioglu
    @Singleton
    abstract fun bindAudioPlayer(
        audioPlayerImpl: AudioPlayerImpl
    ): AudioPlayer

    @Binds  KeremGobekcioglu
    @Singleton
    abstract fun bindAudioRecorder(
        audioRecorderImpl: AudioRecorderImpl
    ): AudioRecorder
}
```

```kotlin
@Singleton  KeremGobekcioglu
class RetrofitManager @Inject constructor() {

    private var retrofit: Retrofit = createRetrofit( baseUrl: "http://localhost") // Placeholder URL

    private fun createRetrofit(baseUrl: String): Retrofit {  KeremGobekcioglu
        return Retrofit.Builder()
            .baseUrl(baseUrl)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
    }

    fun updateBaseUrl(baseUrl: String) {  KeremGobekcioglu
        retrofit = createRetrofit(baseUrl)
    }

    fun getApiService(): AudioApiService {  KeremGobekcioglu
        return retrofit.create(AudioApiService::class.java)
    }
}
```

8.2 State Management in Audio Screen

AudioViewModel uses mutableStateOf to manage the recording, uploading, and error states.

```
@HiltViewModel  KeremGobekcioglu *
class AudioViewModel @Inject constructor(
    @ApplicationContext private val context: Context,
    private val audioPlayer : AudioPlayer,
    private val audioRecorder : AudioRecorder,
    private val repository : AudioRepository,
    private val serverState: ServerState, // Inject the HTTP server
) : ViewModel() {
    private val _state = mutableStateOf(AudioState())
    val state : State<AudioState> = _state
    private val outputFile : File = File(context.cacheDir,  child: "record.mp3")
    init {  KeremGobekcioglu
        (audioPlayer as? AudioPlayerImpl)?.onPlaybackComplete = {
            _state.value = _state.value.copy(isPlaying = false)
        }

        observeServerState()
    }
    private fun observeServerState() {  KeremGobekcioglu
        viewModelScope.launch {
            serverState.receivedImage.collect { file ->
                file?.let {
                    _state.value = _state.value.copy(receivedImage = it)
                }
            }
        }
    }

data class AudioState(  KeremGobekcioglu *
    val isRecording: Boolean = false,
    val isPlaying: Boolean = false,
    val isUploading: Boolean = false,
    val errorMessage: String? = null,
    val receivedImage: File? = null// To store the received image file
)
```

```
fun uploadAudio() {  KeremGobekcioglu *
    viewModelScope.launch {
        try {
            if (outputFile.exists() && !state.value.isRecording) {
                _state.value = _state.value.copy(
                    isUploading = true,
                    errorMessage = null
                )
                val requestFile = outputFile.asRequestBody("audio/mpeg".toMediaTypeOrNull())
                val multipartBody = MultipartBody.Part.createFormData( name: "audioFile", outputFile.name, requestFile)
                val response = repository.uploadAudio(multipartBody)
                if (response.isSuccessful) {
                    _state.value = _state.value.copy(
                        isUploading = false,
                        errorMessage = null
                    )
                } else {
                    Log.d( tag: "AudioViewModel",  msg: "Failed to upload code : ${response.code()} ${response.message()} ${response.errorBody()?.string()}")
                    _state.value = _state.value.copy(
                        isUploading = false,
                        errorMessage = "isUploading = false. Failed to upload audio: ${response.code()} ${response.message()} ${response.errorBody()?.string(
                    )
                }
            } else {
                _state.value = _state.value.copy(
                    errorMessage = "No audio file to upload"
                )
            }
        } catch (e: Exception) {
            Log.d( tag: "AudioViewModel" ,  msg: "Failed to upload audio: ${e.message} , ${e.cause} , ${e.stackTrace}")
            _state.value = _state.value.copy(
                isUploading = false,
                errorMessage = "Exception. Failed to upload audio: ${e.message} , ${e.cause} , ${e.stackTrace}"
            )
        }
    }
}
}
```

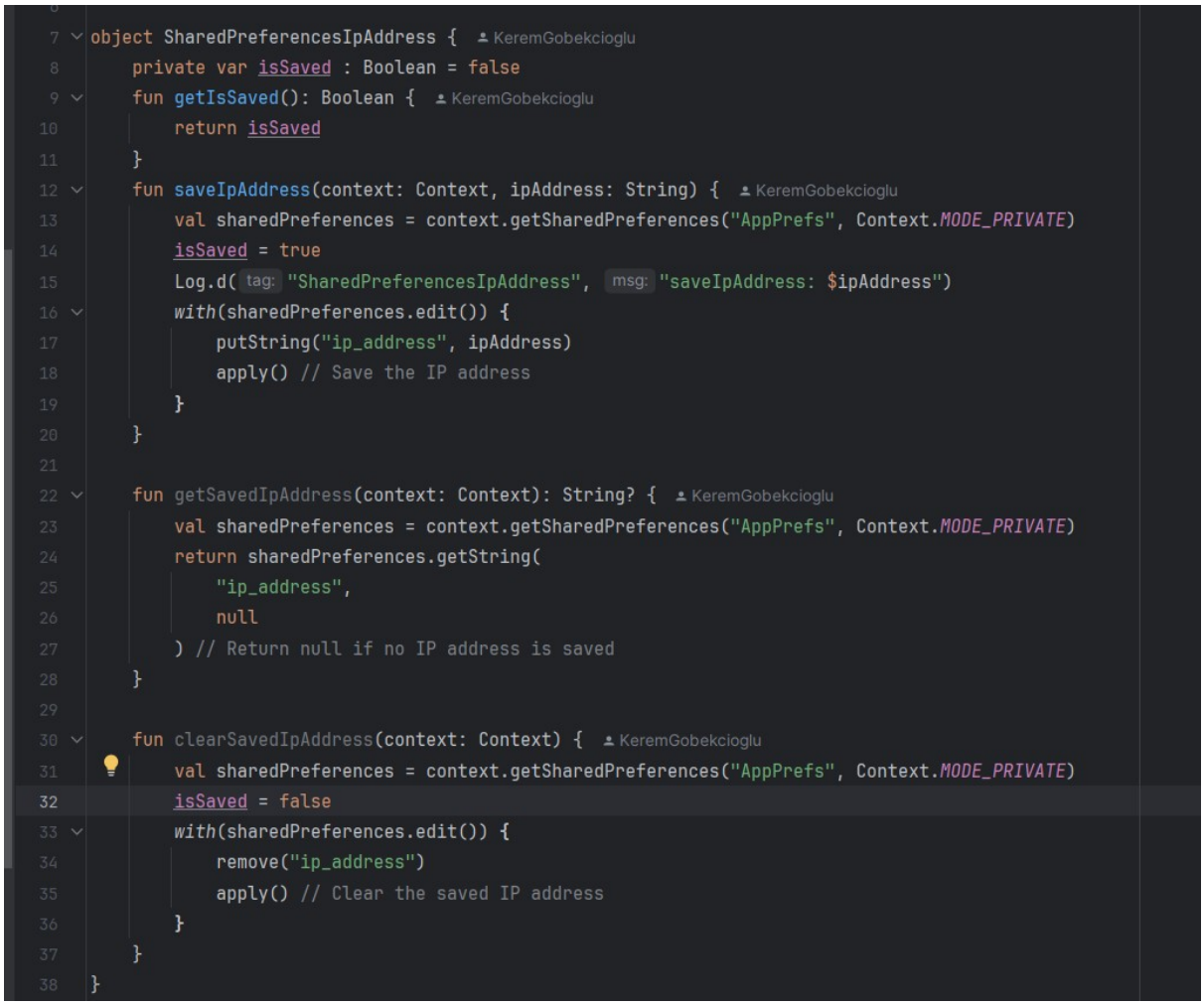
8.3 Local HTTP Server for Intruder Alerts

```
class LocalHttpServer(  KeremGobekcioglu
    private val context: Context,
    private val serverState: ServerState,
    port: Int = 8080
) : NanoHTTPD(port) {

    private val cacheDir = context.cacheDir

    @RequiresApi(Build.VERSION_CODES.O)  KeremGobekcioglu
    override fun serve(session: IHTTPSession?): Response {
        if (session?.method == Method.POST) {
            val body = mutableMapOf<String, String>()
            session.parseBody(body)

            val postData = body["postData"]
                ?: return newFixedLengthResponse(
                    Response.Status.BAD_REQUEST,
                    mimeType: "text/plain",
                    txt: "No data received"
                )

            // Save the received image
            val fileName = "intruder_image.jpg"
            val file = File(cacheDir, fileName)
            try {
                file.writeBytes(Base64.getDecoder().decode(postData))
            } catch (e: IllegalArgumentException) {
                Log.e( tag: "LocalHttpServer",  msg: "Invalid Base64 data: ${e.message} $postData")
                return newFixedLengthResponse(
                    Response.Status.BAD_REQUEST,
                    mimeType: "text/plain",
                    txt: "Invalid Base64 data"
                )
            }

            // Notify ServerState
            serverState.updateReceivedImage(file)
```

```
            }

            // Notify ServerState
            serverState.updateReceivedImage(file)

            // Trigger a notification for the user
            triggerNotification(file)
            serverState.clearReceivedImage()
            return newFixedLengthResponse( msg: "Image received and saved!")
        }

        return newFixedLengthResponse(
            Response.Status.BAD_REQUEST,
            mimeType: "text/plain",
            txt: "Only POST is supported."
        )
    }
private fun triggerNotification(imageFile: File) {  KeremGobekcioglu
    val intent = Intent(context, MainActivity::class.java).apply {
        flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_CLEAR_TASK
        putExtra( name: "image_path", imageFile.absolutePath)
    }
    val pendingIntent = PendingIntent.getActivity(
        context,
        requestCode: 0,
        intent,
        flags: PendingIntent.FLAG_UPDATE_CURRENT or PendingIntent.FLAG_IMMUTABLE
    )

    val channelId = "intruder_alert_channel"
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        val channel = NotificationChannel(
            channelId,
            name: "Intruder Alerts",
            NotificationManager.IMPORTANCE_HIGH
        )
        context.getSystemService(NotificationManager::class.java)?.createNotificationChannel(channel)
        )
        context.getSystemService(NotificationManager::class.java)?.createNotificationChannel(channel)
    }

    val notification = NotificationCompat.Builder(context, channelId)
        .setSmallIcon(R.drawable.intruder)
        .setContentTitle("Intruder Alert!")
        .setContentText("An intruder was detected. Tap to view the image.")
        .setPriority(NotificationCompat.PRIORITY_HIGH)
        .setAutoCancel(true)
        .setContentIntent(pendingIntent)
        .build()

    val notificationManager = context.getSystemService(NotificationManager::class.java)
    notificationManager.notify( id: 1, notification)
}


}
```

8.4 Dynamic IP Configuration

SharedPreferencesIpAddress manages saving and clearing the IP address.

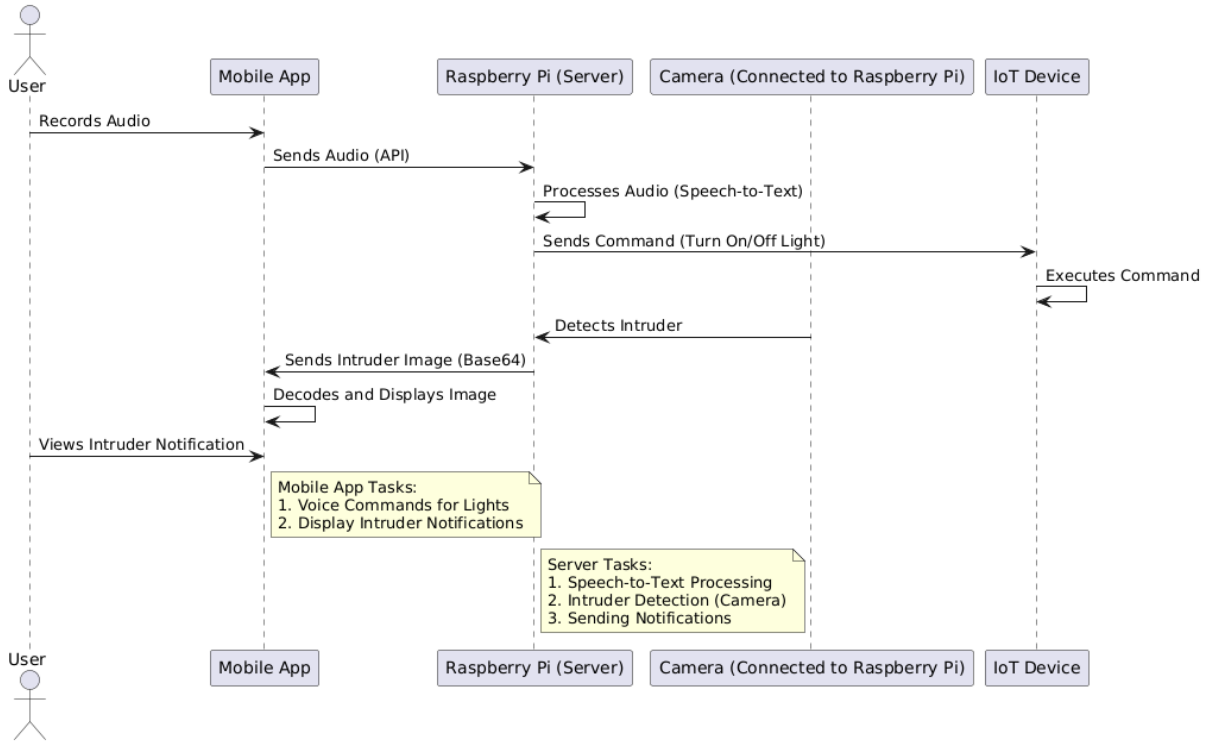


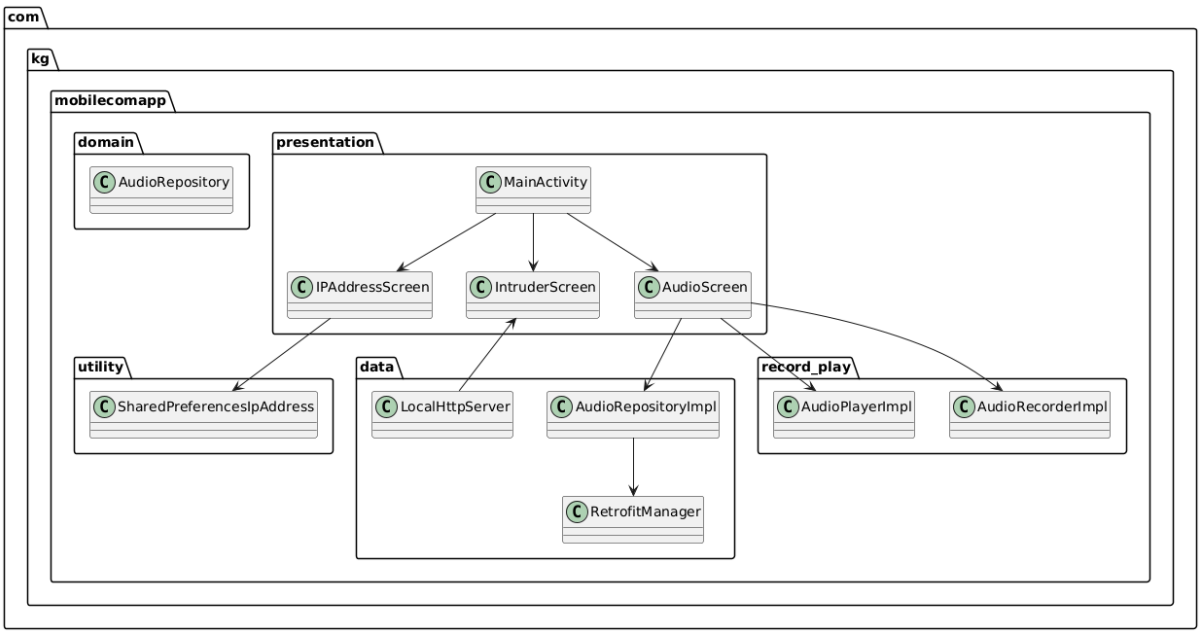
```
object SharedPreferencesIpAddress {  KeremGobekcioglu
    private var isSaved : Boolean = false
    fun getIsSaved(): Boolean {  KeremGobekcioglu
        return isSaved
    }
    fun saveIpAddress(context: Context, ipAddress: String) {  KeremGobekcioglu
        val sharedPreferences = context.getSharedPreferences("AppPrefs", Context.MODE_PRIVATE)
        isSaved = true
        Log.d( tag: "SharedPreferencesIpAddress",  msg: "saveIpAddress: $ipAddress")
        with(sharedPreferences.edit()) {
            putString("ip_address", ipAddress)
            apply() // Save the IP address
        }
    }

    fun getSavedIpAddress(context: Context): String? {  KeremGobekcioglu
        val sharedPreferences = context.getSharedPreferences("AppPrefs", Context.MODE_PRIVATE)
        return sharedPreferences.getString(
            "ip_address",
            null
        ) // Return null if no IP address is saved
    }

    fun clearSavedIpAddress(context: Context) {  KeremGobekcioglu
        val sharedPreferences = context.getSharedPreferences("AppPrefs", Context.MODE_PRIVATE)
        isSaved = false
        with(sharedPreferences.edit()) {
            remove("ip_address")
            apply() // Clear the saved IP address
        }
    }
}
```

9. Server-Side Integration

1. **Voice Commands**
   - Sends recorded audio as a Multipart request to the server.
   - The server processes the audio, converts it to text, and triggers the ESP to control lights.
2. **Intruder Detection**
   - The Raspberry Pi server uses a camera for detection.
   - When an intruder is detected, the server sends a Base64 image to the app's HTTP server.

10. UML Diagrams




com
kg
mobilecomapp
domain
AudioRepository
presentation
MainActivity
IPAddressScreen
IntruderScreen
AudioScreen
utility
SharedPreferencesIpAddress
data
LocalHttpServer
AudioRepositoryImpl
RetrofitManager
record_play
AudioPlayerImpl
AudioRecorderImpl