THIRD EYE SURVEILLANCE

Capstone Project

HRUSHIEKSH PAWAR

Trainer: Priyanka

Date: 24/06/2024

Day 1: Automotive Security Introduction and Environment Setup, Camera Integration and Motion Detection

Task 1: Learn about automotive security systems, focusing on camera-based surveillance for moving and parked vehicles.

Objective:

Gain an understanding of automotive security systems, focusing specifically on camera-based surveillance for both moving and parked vehicles.

Detailed Exploration:

1. Introduction to Camera-Based Surveillance in Vehicles:

Purpose: Camera-based surveillance systems are designed to enhance the security of vehicles by monitoring and recording activities both inside and outside the vehicle. These systems help in preventing theft, vandalism, and other malicious activities.

2. Components of Camera-Based Surveillance Systems:

Cameras: Different types are used, such as front facing, rear facing, and 360degree cameras.

Sensors: Motion sensors detect movement around the vehicle and trigger recording.

Storage: Local (SD cards) or cloud storage for saving footage.

Connectivity: Wi-Fi, Bluetooth, or cellular connectivity for remote monitoring and alerts.

3. Surveillance for Moving Vehicles:

Front Facing Cameras: Capture the road ahead and record driving incidents.

Rear Facing Cameras: Assist in reversing and monitoring traffic behind the vehicle.

Dash Cams: Continuous recording to capture any incidents on the road.

Event Triggered Recording: Starts recording during sudden braking, collisions, or sharp turns.

Example Scenario:

While driving, a front facing camera records an incident where another car suddenly swerves into your lane, providing crucial evidence for insurance claims.

4. Surveillance for Parked Vehicles:

Motion Detection: Triggers recording when movement is detected around the vehicle.

Night Vision: Ensures clear footage even in lowlight conditions.

Alerts: Sends notifications to the owner's smartphone if suspicious activity is detected.

Remote Monitoring: Allows vehicle owners to view live footage from their smartphones.

Example Scenario:

A motion sensor detects movement near a parked vehicle at night. The camera starts recording, and an alert is sent to the owner's phone, enabling them to view the footage in real-time.

5. Key Features:

Continuous and Event Triggered Recording: Ensures no important event goes unrecorded.

High Resolution Video: Captures clear and detailed footage.

Wide Angle Lenses: Cover more area around the vehicle.

Tamper Detection: Alerts if the camera is tampered with or obstructed.

6. Benefits:

Enhanced Security: Deters potential thieves and vandals.

Evidence Collection: Provides reliable evidence for incidents, aiding in insurance claims and legal matters.

Peace of Mind: Allows owners to feel secure about their vehicle's safety.

Task 2: Set Up an Android Development Environment Including Emulator Setup, Camera Access, and Background Processing

Objective:

Set up an Android development environment, including configuring an emulator, creating a new project with camera access permissions, and enabling background processing capabilities.

Step 1: Set Up Android Studio and Emulator

1. Download and Install Android Studio:

Visit the [Android Studio download page] (https://developer.android.com/studio).

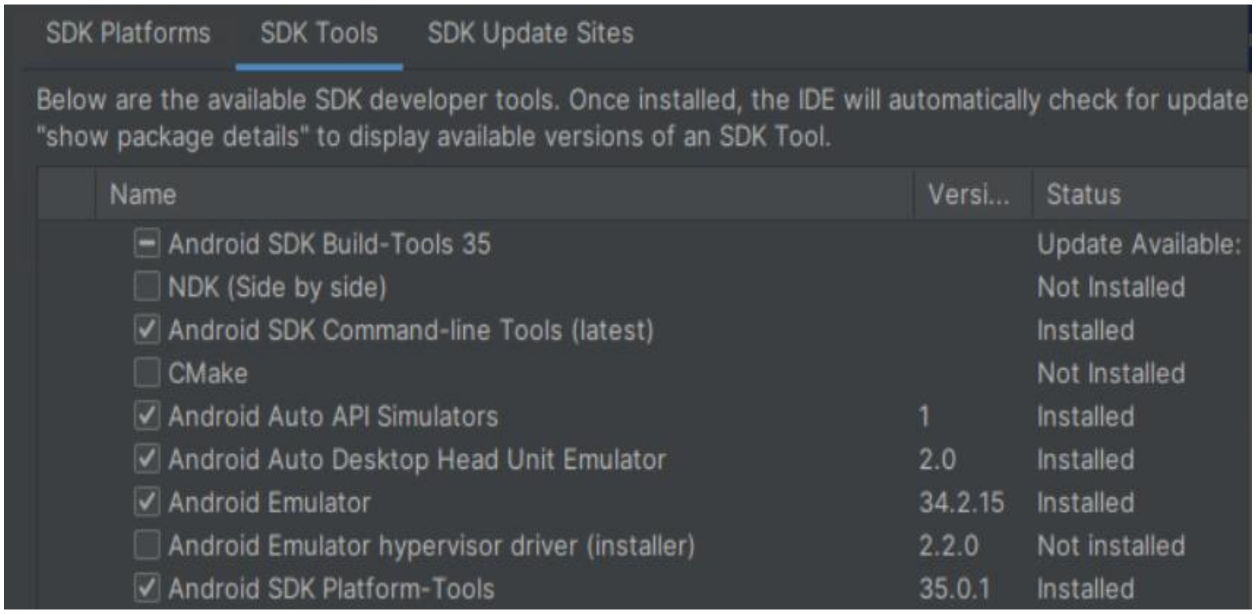
Download and install Android Studio based on your operating system.

Follow the installation instructions to set up the IDE.

2. Launch Android Studio:

Open Android Studio after installation.

Complete the initial setup wizard to install necessary SDK components and tools.



Below are the available SDK developer tools. Once installed, the IDE will automatically check for update
"show package details" to display available versions of an SDK Tool.

Name	Versi...	Status
▬ Android SDK Build-Tools 35		Update Available:
☐ NDK (Side by side)		Not Installed
✓ Android SDK Command-line Tools (latest)		Installed
☐ CMake		Not Installed
✓ Android Auto API Simulators	1	Installed
✓ Android Auto Desktop Head Unit Emulator	2.0	Installed
✓ Android Emulator	34.2.15	Installed
☐ Android Emulator hypervisor driver (installer)	2.2.0	Not installed
✓ Android SDK Platform-Tools	35.0.1	Installed

3. Set Up Android Emulator:

Click on "Configure" in the Android Studio welcome screen.

Select "AVD Manager" and click on "Create Virtual Device".

Choose a device definition (e.g., Pixel 4) and click "Next".

Select a system image (e.g., Q or R release) and click "Next".

Customize the emulator settings as needed and click "Finish" to create the emulator.

Step 2: Create a New Android Project

1. Start a New Project:

Click "Start a new Android Studio project" from the welcome screen.

Select "Empty Activity" and click "Next".

2. Configure Your Project:

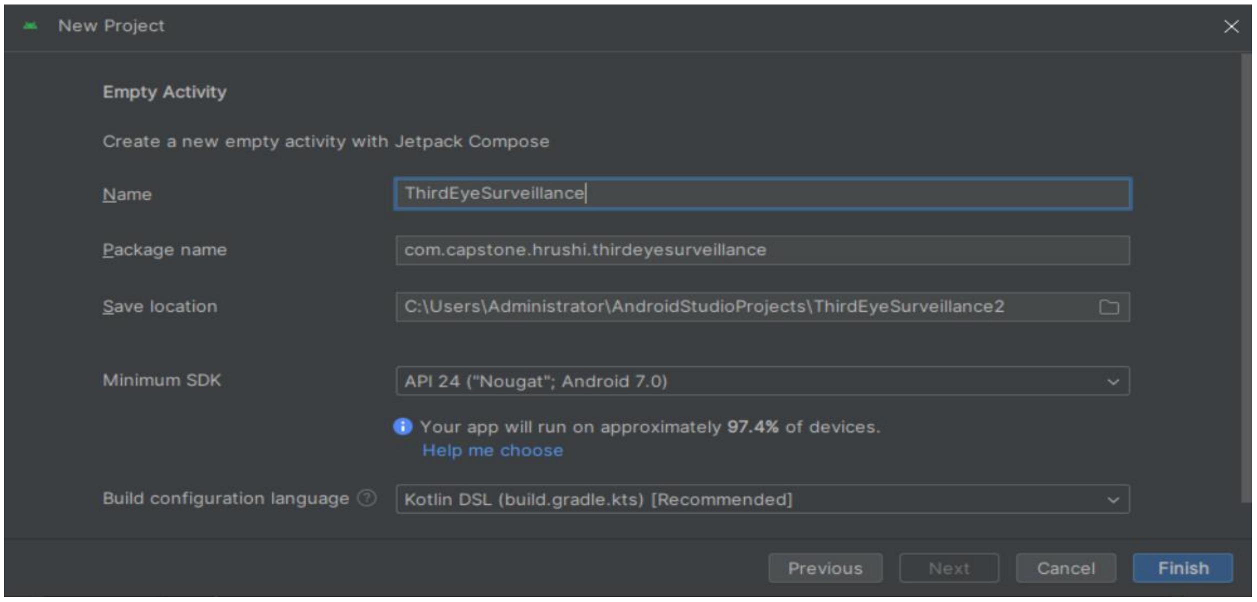
Enter a name for your project (e.g., "ThirdEyeSurveillance").

Specify the package name (e.g., “com.example.thirdeyesurveillance”).

Choose the language as "Java".

Select the minimum SDK version (e.g., API 21: Android 5.0 Lollipop).

Click "Finish" to create the project.



Empty Activity

Create a new empty activity with Jetpack Compose

Name: ThirdEyeSurveillance

Package name: com.capstone.hrushi.thirdeyesurveillance

Save location: C:\Users\Administrator\AndroidStudioProjects\ThirdEyeSurveillance2

Minimum SDK: API 24 ("Nougat"; Android 7.0)

Your app will run on approximately **97.4%** of devices.
Help me choose

Build configuration language: Kotlin DSL (build.gradle.kts) [Recommended]

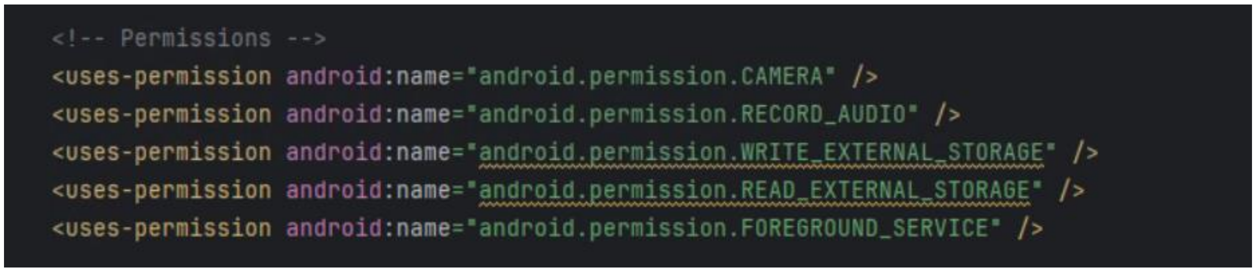
Step 3: Configure Permissions for Camera Access

1. Open the Manifest File:

Navigate to "app/src/main/AndroidManifest.xml".

2. Add Camera Permission:

Add the following permissions inside the "<manifest> " tag:



```
<!-- Permissions -->
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.RECORD_AUDIO" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.FOREGROUND_SERVICE" />
```

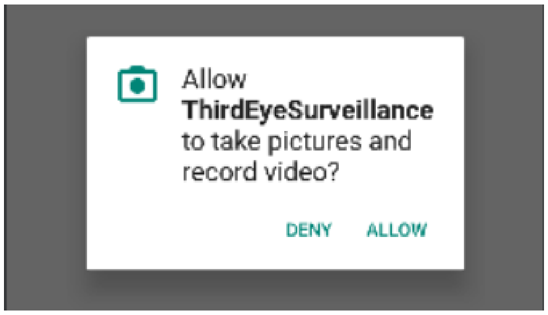
Step 4: Implement Camera Access

1. Modify MainActivity.java:

Open "MainActivity.java" located at "app/src/main/java/com/example/thirdeyesurveillance/".

2. Request Camera Permission:

Implement runtime permission requests and handle permission results:



Allow **ThirdEyeSurveillance** to take pictures and record video?

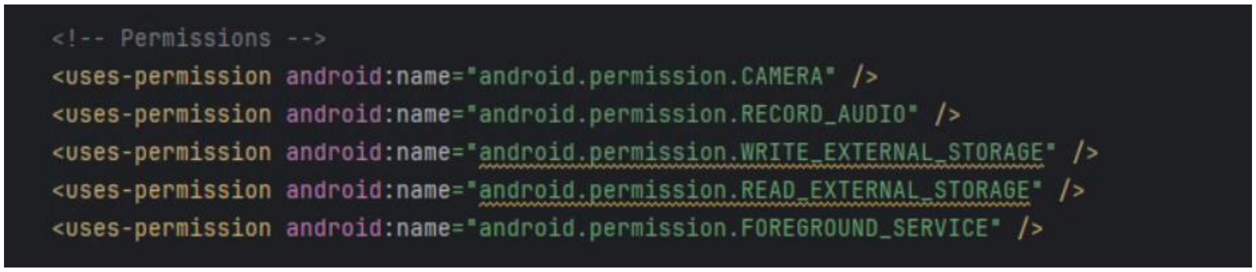
By following these detailed steps, you will have set up an Android project with necessary permissions, ready for further development and testing on an Android emulator.

Task 3: Integrate the camera API to capture live video feeds and still images from within an Android application.

To integrate camera functionality into your Android application using the provided code snippet, we'll focus on ensuring proper permission handling and demonstrate how to open the camera using an intent. Below is a step-by-step guide based on your existing “MainActivity” code:

Step 1: Update “AndroidManifest.xml”

Ensure that your “AndroidManifest.xml” includes the camera permission declaration:



```xml
<!-- Permissions -->
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.RECORD_AUDIO" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.FOREGROUND_SERVICE" />
```

Step 2: Modify “MainActivity.java”

1. Declare Constants and Variables:

Initialize necessary constants and variables, including permission request codes and tags for logging:

The constants define unique codes used for identifying permission requests and camera activity. They help manage the permission request and camera launch processes effectively within the app.

2. Override “onCreate” Method:

In “onCreate”, check for camera permission. If granted, initialize camera functionality; otherwise, request permission:

This method initializes the activity when it is created. It sets the user interface layout and checks if the camera permission is already granted. If granted, it proceeds to open the camera; otherwise, it requests the necessary permission.

3. Check Camera Permission:

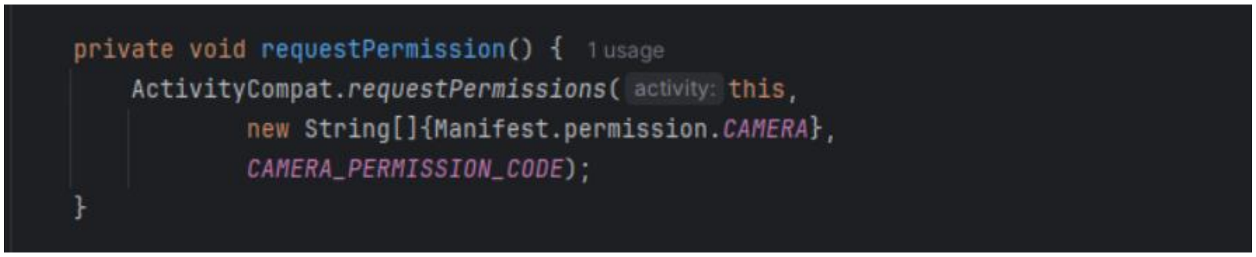
Implement “checkPermission” method to check if camera permission is granted:

This method checks whether the app has been granted camera permission by the user. It returns a Boolean value indicating the presence or absence of the required permission.

4. Request Camera Permission:

Implement “requestPermission” method to request camera permission if not granted:

This method handles the process of requesting camera permission from the user. If the permission has not been granted, it prompts the user to grant the necessary camera access.

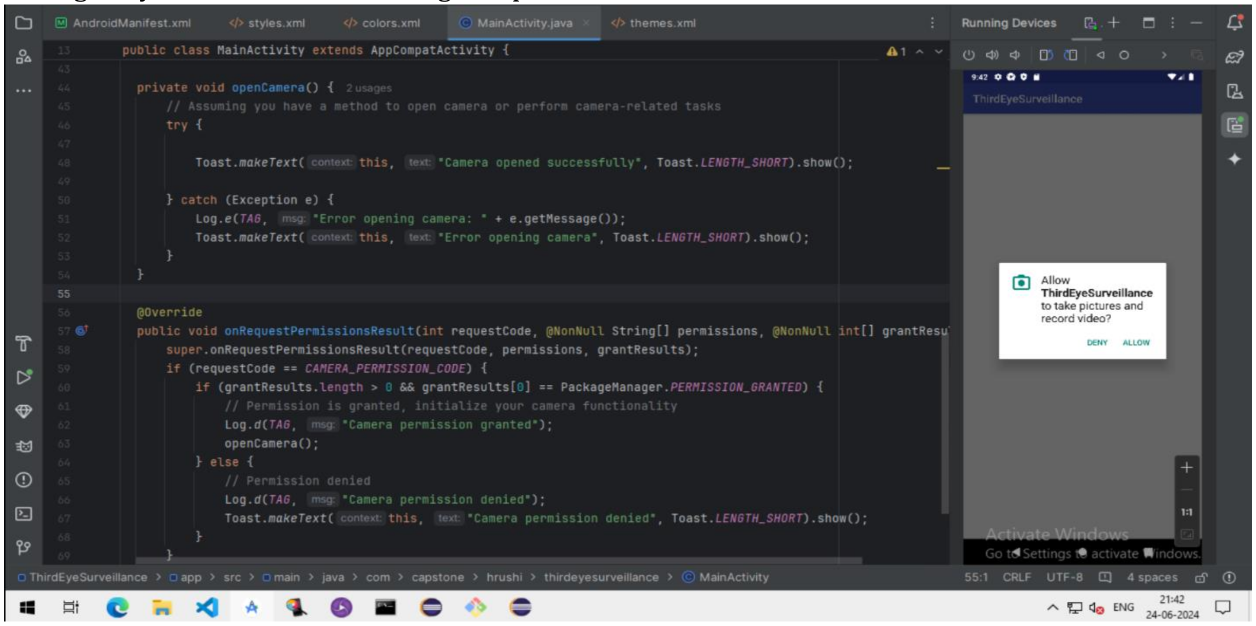


```java
private void requestPermission() {  1 usage
    ActivityCompat.requestPermissions( activity: this,
            new String[]{Manifest.permission.CAMERA},
            CAMERA_PERMISSION_CODE);
}
```

5. Handle Permission Request Result:

Override “onRequestPermissionsResult” to handle the result of the permission request:

This method is responsible for launching the camera application to capture an image. It uses an intent to open the default camera app on the device and includes error handling to manage any issues that arise during this process.



```java
public class MainActivity extends AppCompatActivity {

    private void openCamera() {  2 usages
        // Assuming you have a method to open camera or perform camera-related tasks
        try {

            Toast.makeText( context: this,  text: "Camera opened successfully", Toast.LENGTH_SHORT).show();

        } catch (Exception e) {
            Log.e(TAG,  msg: "Error opening camera: " + e.getMessage());
            Toast.makeText( context: this,  text: "Error opening camera", Toast.LENGTH_SHORT).show();
        }
    }

    @Override
    public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions, @NonNull int[] grantResu
        super.onRequestPermissionsResult(requestCode, permissions, grantResults);
        if (requestCode == CAMERA_PERMISSION_CODE) {
            if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                // Permission is granted, initialize your camera functionality
                Log.d(TAG,  msg: "Camera permission granted");
                openCamera();
            } else {
                // Permission denied
                Log.d(TAG,  msg: "Camera permission denied");
                Toast.makeText( context: this,  text: "Camera permission denied", Toast.LENGTH_SHORT).show();
            }
        }
```

Allow **ThirdEyeSurveillance** to take pictures and record video?

6. Open Camera:

Implement the “openCamera” method to launch the camera using an intent. Replace the placeholder code with your actual camera launch implementation:

This method processes the result of the permission request. If the user grants the camera permission, it proceeds to open the camera. If the permission is denied, it logs this event and informs the user via a toast message.

Integrating the camera API to capture live video feeds and still images within an Android application:

1. Setup Camera Permissions:

Ensure your app has the necessary permissions by adding “CAMERA” and “WRITE_EXTERNAL_STORAGE” permissions to your “AndroidManifest.xml” file.

Request runtime permissions in your activity if the device runs Android 6.0 (Marshmallow) or higher.

2. Initialize Camera:

Use Android's “Camera2” API for a modern approach to camera functionality.

Initialize the camera by getting a “CameraManager” and opening the camera using “CameraDevice”.

3. Configure Camera Preview:

Create a “SurfaceTexture” or “SurfaceView” to display the live camera feed.

Set up a “CaptureRequest.Builder” with the “TEMPLATE_PREVIEW” template to configure the camera for preview mode.

Bind the camera output to the preview surface using a “CaptureSession”.

4. Handle Camera Lifecycle:

Manage the camera's lifecycle within the “onResume” and “onPause” methods to open and close the camera respectively.

Implement appropriate error handling and cleanup in the camera lifecycle methods.

5. Capture Still Images:

Configure a “CaptureRequest.Builder” with the “TEMPLATE_STILL_CAPTURE” template for capturing still images.

Trigger the image capture by calling “capture()” on the “CameraCaptureSession”.

Save the captured image to storage using an “ImageReader” and its “OnImageAvailableListener”.

6. Capture Video Feeds:

Set up a “MediaRecorder” to handle video recording.

Configure the “MediaRecorder” with the desired video and audio source, output format, encoder, and file destination.

Start and stop the video recording within the camera capture session.

7. Handle Image and Video Storage:

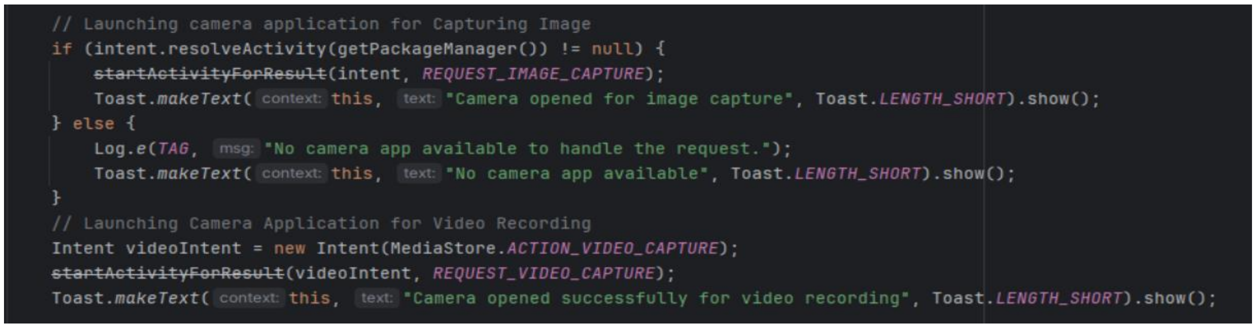
Save captured images and videos to the device's external storage.

Ensure proper file naming conventions and handling of storage permissions.

8. Optimize for Performance:

Optimize camera initialization and release to avoid memory leaks and ensure smooth operation.

Handle different device configurations and orientations to ensure the camera works seamlessly across various devices.



```
// Launching camera application for Capturing Image
if (intent.resolveActivity(getPackageManager()) != null) {
    ~~startActivityForResult~~(intent, REQUEST_IMAGE_CAPTURE);
    Toast.makeText( context: this,  text: "Camera opened for image capture", Toast.LENGTH_SHORT).show();
} else {
    Log.e(TAG,  msg: "No camera app available to handle the request.");
    Toast.makeText( context: this,  text: "No camera app available", Toast.LENGTH_SHORT).show();
}
// Launching Camera Application for Video Recording
Intent videoIntent = new Intent(MediaStore.ACTION_VIDEO_CAPTURE);
~~startActivityForResult~~(videoIntent, REQUEST_VIDEO_CAPTURE);
Toast.makeText( context: this,  text: "Camera opened successfully for video recording", Toast.LENGTH_SHORT).show();
```

Conclusion:

By following these technical steps, you can effectively integrate the camera API into your Android application to capture live video feeds and still images. This integration enhances the application's functionality, allowing users to capture and record media directly within the app.

Task 4: Implement basic motion detection algorithms that trigger photo capture when movement is detected around the vehicle.

Implementing motion detection algorithms for automotive surveillance involves several detailed steps. This process ensures that the system can detect motion around a vehicle and capture images for security purposes. Below is a comprehensive explanation of each step, followed by a flowchart for better understanding.

Steps to Implement Motion Detection:

1. Capture Frames from Camera Feed:

Description: Continuously capture frames from the camera feed for analysis.

How it Works:

Use Android's Camera API to access the live video stream from the device's camera.

Set up a frame capture mechanism at a predefined frame rate (e.g., 10 frames per second) to ensure a steady flow of images.

Purpose: Provides a continuous stream of images that are essential for detecting changes and motion.

2. Convert Frames to Grayscale:

Description: Simplify the images by converting them to grayscale.

How it Works:

Convert each captured frame from its original RGB (RedGreen-Blue) color format to grayscale using algorithms that average pixel values across color channels.

Grayscale images reduce computational load compared to color images, making subsequent image processing steps more efficient.

Purpose: Focuses analysis on changes in brightness rather than color, which is sufficient for motion detection and reduces processing time.

3. Calculate Frame Difference:

Description: Compare consecutive frames to identify changes indicating motion.

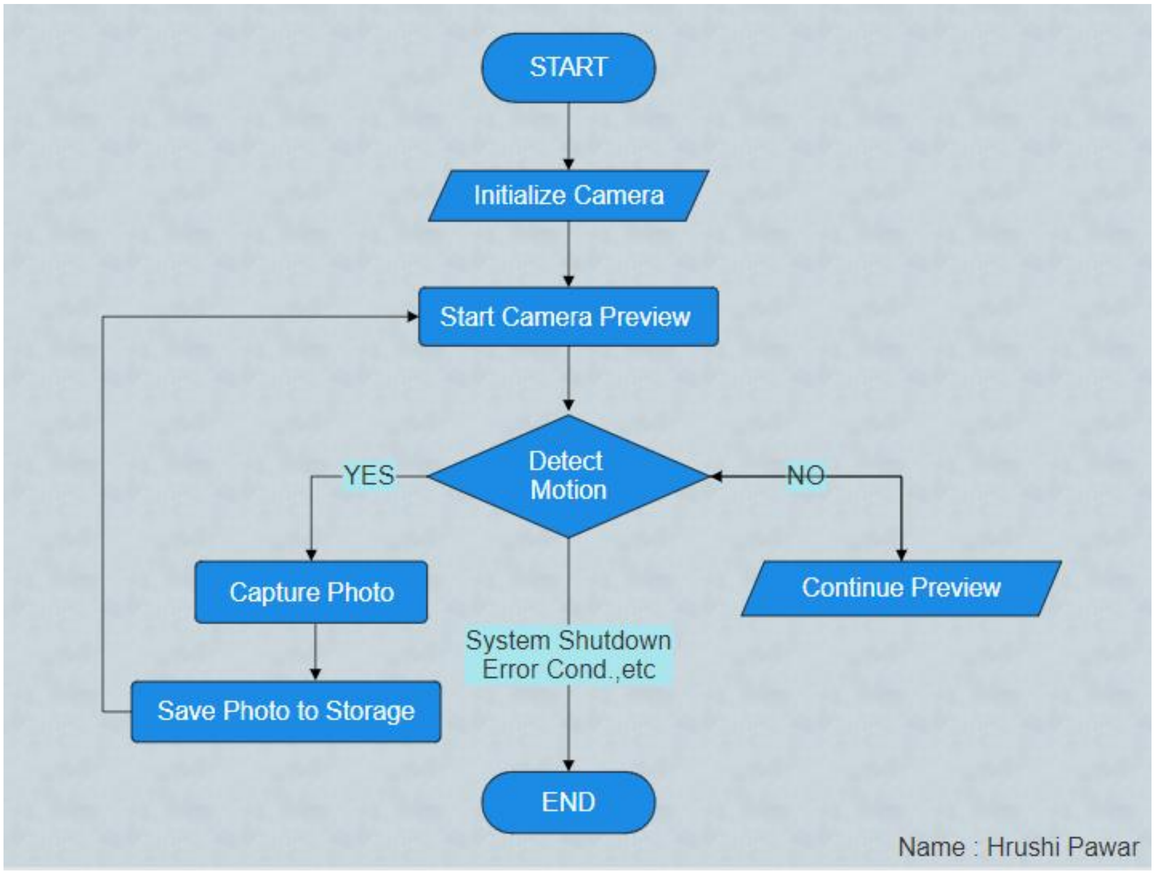
How it Works:

Subtract pixel values of the current grayscale frame from the previous grayscale frame.

Generate a difference image where pixels with significant changes indicate potential motion.

Apply mathematical operations or algorithms to enhance the detection of changes, such as absolute differences or thresholding techniques.

Purpose: Pinpoints areas in the frame where there are noticeable changes, indicating potential motion around the vehicle.



4. Apply Threshold to Differences:

Description: Isolate significant changes by applying a threshold to the frame differences.

How it Works:

Convert the difference image into a binary image where pixels above a certain threshold are considered part of the motion.

Adjust the threshold dynamically based on lighting conditions and environmental factors to minimize false positives or negatives.

Purpose: Filters out minor changes and noise, focusing analysis on substantial motion events.

5. Detect Contours of Motion:

Description: Identify specific areas where motion is detected by finding contours.

How it Works:

Use contour detection algorithms, such as OpenCV's findContours function, on the binary image.

Extract boundaries that enclose regions with significant pixel differences, representing potential motion.

Apply techniques like morphological operations (e.g., dilation and erosion) to refine detected contours and improve accuracy.

Purpose: Provides precise localization of motion within the frame, enabling targeted action such as photo capture.

6. Trigger Photo Capture on Motion Detection:

Description: Capture a photo when significant motion is detected.

How it Works:

Implement logic to trigger the camera capture functionality of the Android device upon detecting significant contours or areas of motion.

Use intents or direct camera APIs to initiate photo capture, ensuring the capture is instantaneous and aligned with motion detection events.

Purpose: Records visual evidence of detected motion events for security or monitoring purposes.

7. Save Captured Photos:

Description: Save the photos taken when motion is detected for later review.

How it Works:

Manage storage operations to create and save photo files in a specified directory on the device's internal or external storage.

Ensure each photo file is uniquely named and organized based on timestamp or other metadata for easy retrieval and review.

Purpose: Maintains a historical record of detected motion events, providing documentation for security incidents or surveillance activities.

8. Manage Motion Detection Sensitivity:

Description: Adjust the sensitivity of the motion detection algorithm to balance detection accuracy and minimize false alarms.

How it Works:

Finetune parameters such as frame rate, threshold levels, contour detection algorithms, and image preprocessing techniques.

Conduct rigorous testing and validation to optimize sensitivity settings based on realworld scenarios and environmental conditions.

Purpose: Ensures the motion detection system reliably identifies significant motion events while minimizing false positives from ambient noise or lighting changes.

Day 2: Vibration Detection and Event Driven Capture

Task 1: Develop a system to detect vibrations using the vehicle's builtin sensors or external hardware connected to an Android device.

Introduction

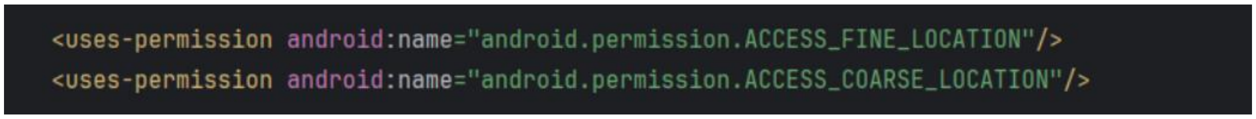
In this task, we aim to create an Android application capable of detecting vibrations using the device's builtin accelerometer sensor. This system is essential for applications requiring realtime monitoring of vehicle movements, security systems, or automation triggers based on physical motion.

Steps to Implement

1. Setup Android Project

Initialize a new Android project in Android Studio with necessary permissions and dependencies.

Ensure the “AndroidManifest.xml” includes permissions to access device sensors (“ACCESS_FINE_LOCATION” for accelerometer) and other required features.



```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
```

Explanation:

Permissions ensure the application can access device sensors like the accelerometer (“ACCESS_FINE_LOCATION”), crucial for accurate vibration detection.

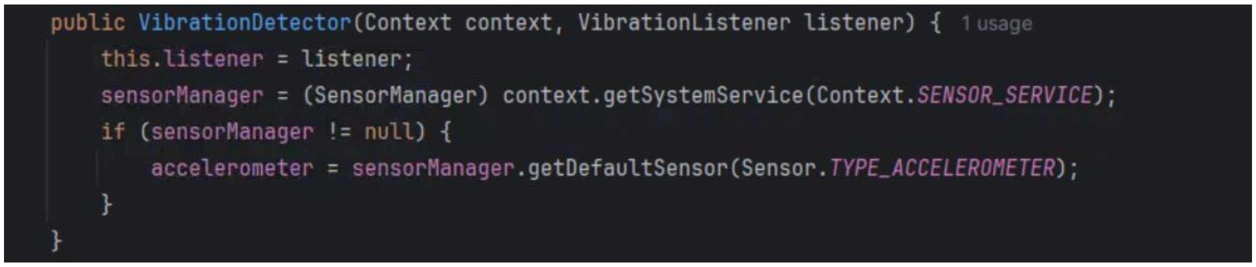
2. Initialize SensorManager

Utilize “SensorManager” to access device sensors, specifically the accelerometer.

Obtain the default accelerometer sensor for sensor readings within the “onCreate()” method.

Explanation:

“SensorManager” manages all device sensors (“Context.SENSOR_SERVICE”), essential for accessing accelerometer data (“Sensor.TYPE_ACCELEROMETER”).



```java
public VibrationDetector(Context context, VibrationListener listener) {  1 usage
    this.listener = listener;
    sensorManager = (SensorManager) context.getSystemService(Context.SENSOR_SERVICE);
    if (sensorManager != null) {
        accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
    }
}
```

“SensorManager” is initialized to access device sensors.

We retrieve the default accelerometer sensor (“Sensor.TYPE_ACCELEROMETER”) from the system.

3. Register SensorEventListener

Explanation:

We register a “SensorEventListener” to listen for changes in accelerometer readings, enabling realtime monitoring of the device's movement.

“SensorEventListener” is implemented to handle accelerometer data changes (“onSensorChanged”) and accuracy changes (“onAccuracyChanged”).

The listener is registered (“sensorManager.registerListener”) with the accelerometer sensor using “SENSOR_DELAY_NORMAL” for normal sensor data rate.

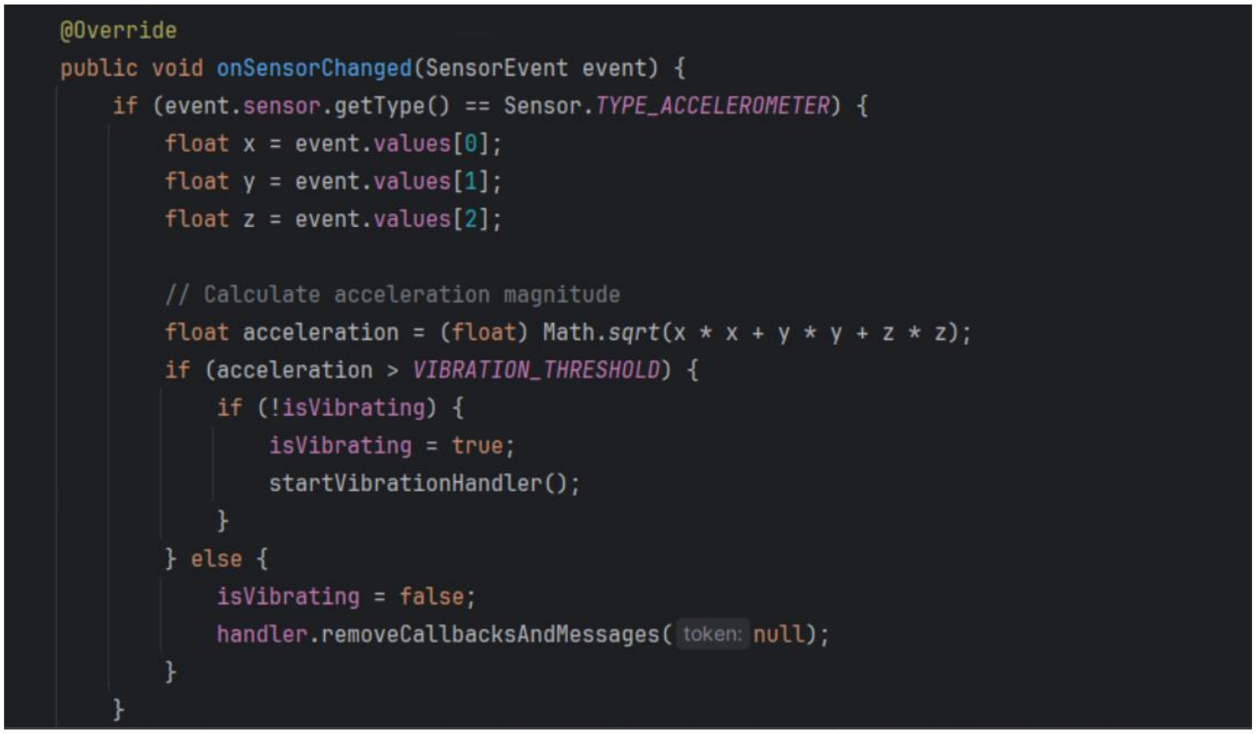
4. Handle Sensor Data Changes

Explanation:

Implement logic to detect vibrations based on accelerometer data changes, calculating acceleration magnitude and triggering actions upon exceeding a threshold.

“onSensorChanged” method processes accelerometer data changes (“event.values”) to calculate acceleration magnitude.

The “VIBRATION_THRESHOLD” determines when vibration is detected, logging the magnitude if exceeded.

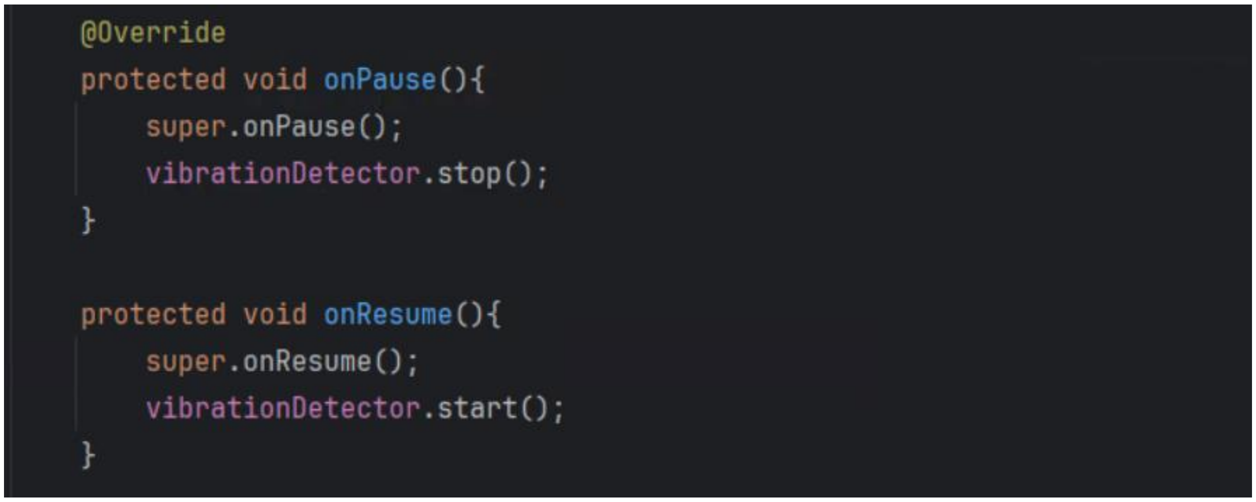


```
@Override
public void onSensorChanged(SensorEvent event) {
    if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
        float x = event.values[0];
        float y = event.values[1];
        float z = event.values[2];

        // Calculate acceleration magnitude
        float acceleration = (float) Math.sqrt(x * x + y * y + z * z);
        if (acceleration > VIBRATION_THRESHOLD) {
            if (!isVibrating) {
                isVibrating = true;
                startVibrationHandler();
            }
        } else {
            isVibrating = false;
            handler.removeCallbacksAndMessages( token: null);
        }
    }
```

5. Unregister SensorEventListener

Explanation:

Ensure to unregister the “SensorEventListener” when the app is paused or stopped to conserve battery and prevent unnecessary sensor updates.



```java
@Override
protected void onPause(){
    super.onPause();
    vibrationDetector.stop();
}

protected void onResume(){
    super.onResume();
    vibrationDetector.start();
}
```

In this snippet:

“onPause” method unregisters the “SensorEventListener” (“sensorEventListener”) when the activity is paused.

Unregistering prevents continuous sensor updates when the app is not in the foreground, conserving battery life.

6. Request Permissions

Request necessary permissions to access the accelerometer sensor, ensuring the app has the required access to monitor device movement.

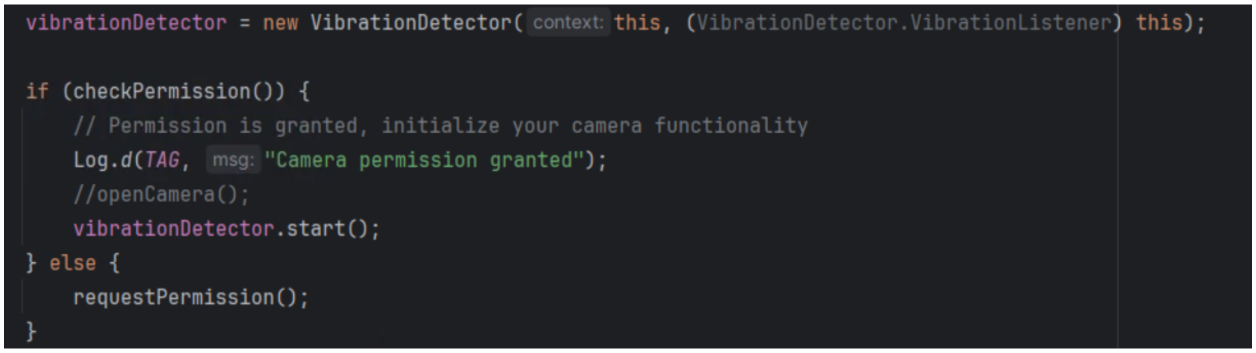
Permission (“Manifest.permission.ACCESS_FINE_LOCATION”) is requested if not already granted (“checkSelfPermission”).

Requesting permission ensures the app can access accelerometer data for vibration detection.

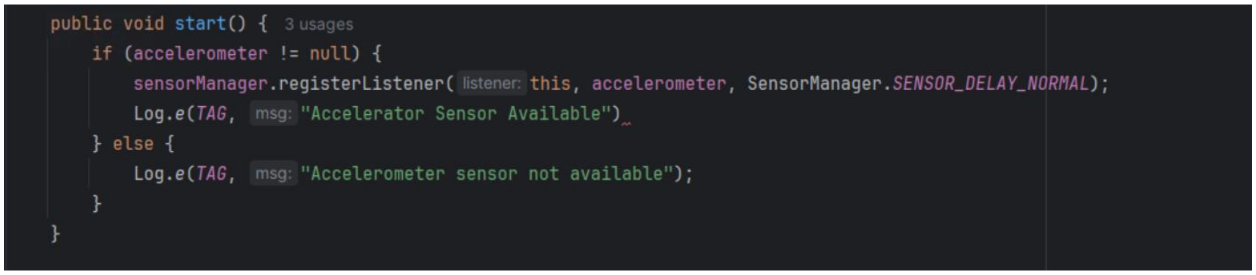
7. Handle Permission Results

Explanation:

Handle the result of the permission request to initialize sensor functionality if permission is granted or provide feedback if denied.



```
vibrationDetector = new VibrationDetector( context: this, (VibrationDetector.VibrationListener) this);

if (checkPermission()) {
    // Permission is granted, initialize your camera functionality
    Log.d(TAG,  msg: "Camera permission granted");
    //openCamera();
    vibrationDetector.start();
} else {
    requestPermission();
}
```

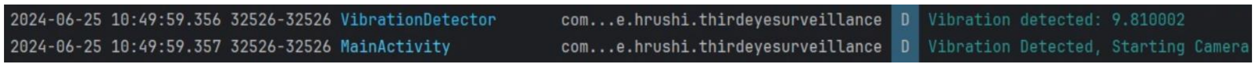


```
    public void start() {  3 usages
        if (accelerometer != null) {
            sensorManager.registerListener( listener: this, accelerometer, SensorManager.SENSOR_DELAY_NORMAL);
            Log.e(TAG,  msg: "Accelerator Sensor Available")
        } else {
            Log.e(TAG,  msg: "Accelerometer sensor not available");
        }
    }
```

In this snippet:

"onRequestPermissionsResult" method handles permission result ("requestCode == SENSOR_PERMISSION_CODE").

If permission is granted ("PackageManager.PERMISSION_GRANTED"), sensor functionality ("registerListener") is initialized; otherwise, a toast message informs the user of denied access.



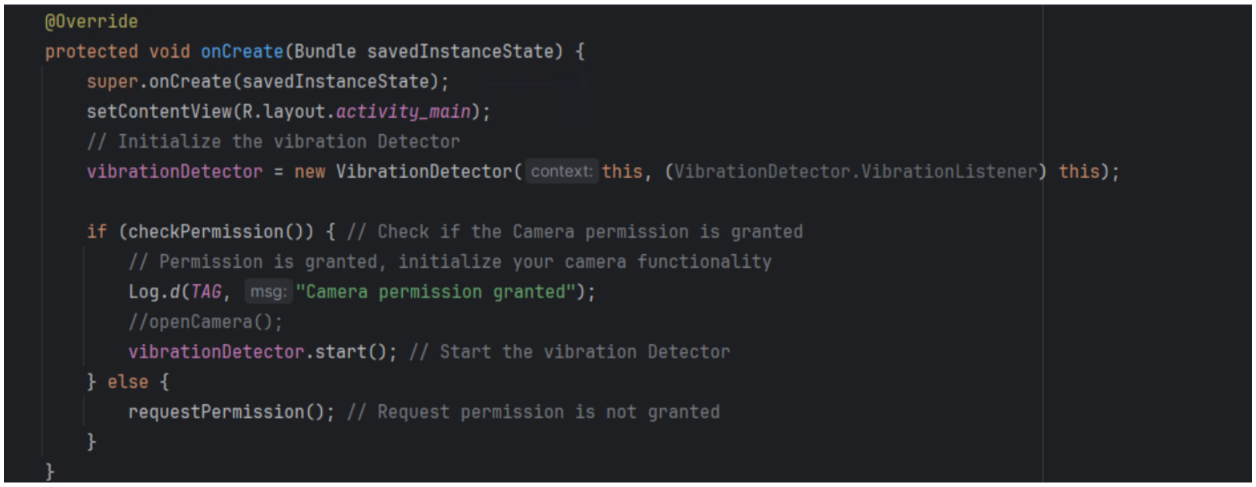
```
2024-06-25 10:49:59.356 32526-32526 VibrationDetector       com...e.hrushi.thirdeyesurveillance  D  Vibration detected: 9.810002
2024-06-25 10:49:59.357 32526-32526 MainActivity            com...e.hrushi.thirdeyesurveillance  D  Vibration Detected, Starting Camera
```

These steps provide a detailed approach to developing a system for detecting vibrations using the vehicle's builtin sensors or external hardware connected to an Android device. Each code snippet focuses on critical functionality while explanations offer clear insights into their purpose and implementation within the overall system.

Task 2: Set up an eventdriven mechanism that initiates the camera to take photos every 2 seconds when vibrations are detected while parked.

Step 1: Initialize Camera Permissions and Vibration Detector

In this step, we initialize the camera permissions and the vibration detector within the "MainActivity" class. The goal is to ensure the app has the necessary permissions to access the camera. If the permission is granted, we start the vibration detector. Otherwise, we request the necessary permissions.



```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    // Initialize the vibration Detector
    vibrationDetector = new VibrationDetector( context: this, (VibrationDetector.VibrationListener) this);

    if (checkPermission()) { // Check if the Camera permission is granted
        // Permission is granted, initialize your camera functionality
        Log.d(TAG,  msg: "Camera permission granted");
        //openCamera();
        vibrationDetector.start(); // Start the vibration Detector
    } else {
        requestPermission(); // Request permission is not granted
    }
}
```

Explanation:

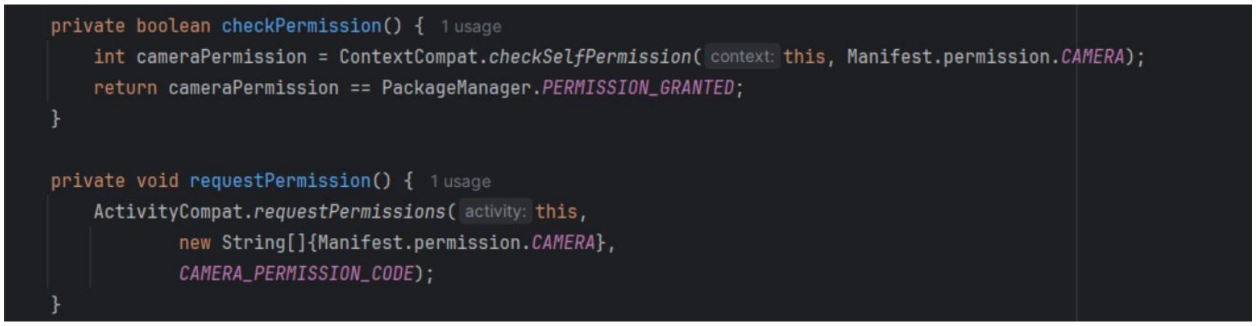
We define “CAMERA_PERMISSION_CODE” for handling camera permission requests.

“VibrationDetector” is instantiated, linking it with the “MainActivity”.

We check if camera permissions are granted using “checkPermission”. If granted, the vibration detector is started; otherwise, we request permissions using “requestPermission”.

2. Check and Request Camera Permission

These methods are responsible for checking if the camera permission is granted and requesting it if not. This ensures the app can access the camera when required.



```java
private boolean checkPermission() {  1 usage
    int cameraPermission = ContextCompat.checkSelfPermission( context: this, Manifest.permission.CAMERA);
    return cameraPermission == PackageManager.PERMISSION_GRANTED;
}

private void requestPermission() {  1 usage
    ActivityCompat.requestPermissions( activity: this,
            new String[]{Manifest.permission.CAMERA},
            CAMERA_PERMISSION_CODE);
}
```

Explanation:

“checkPermission” checks if the camera permission is granted and returns a boolean result.

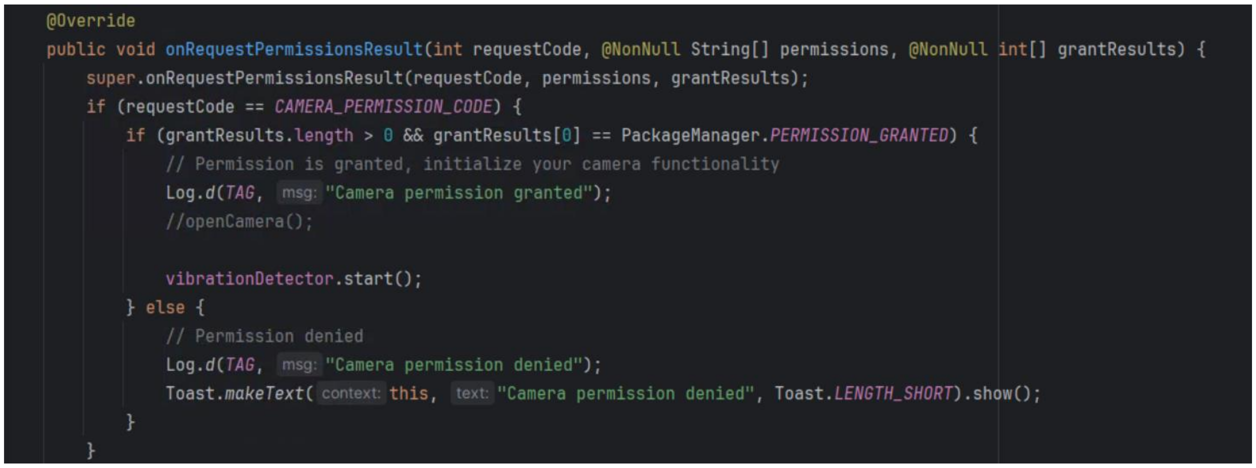
“requestPermission” requests the camera permission if it hasn't been granted yet, using “ActivityCompat.requestPermissions”.

3. Handle Permission Request Result

Explanation:

This method handles the result of the camera permission request. If the permission is granted, we start the vibration detector.

Code Snippet:



```java
@Override
public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions, @NonNull int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);
    if (requestCode == CAMERA_PERMISSION_CODE) {
        if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            // Permission is granted, initialize your camera functionality
            Log.d(TAG, msg: "Camera permission granted");
            //openCamera();

            vibrationDetector.start();
        } else {
            // Permission denied
            Log.d(TAG, msg: "Camera permission denied");
            Toast.makeText( context: this, text: "Camera permission denied", Toast.LENGTH_SHORT).show();
        }
    }
```

Explanation:

“onRequestPermissionsResult” checks the result of the permission request.

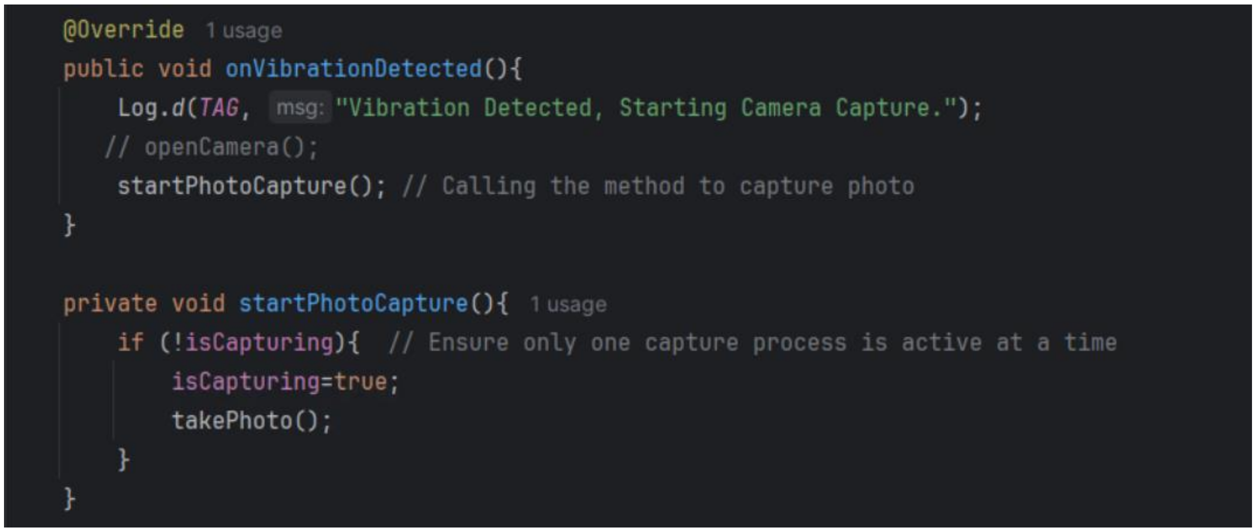
If the permission is granted (“grantResults[0] == PackageManager.PERMISSION_GRANTED”), we start the vibration detector.

If the permission is denied, we log the result and show a Toast message to inform the user.

4. Detect Vibrations and Trigger Camera

Explanation:

Implement the “VibrationListener” interface and define the “onVibrationDetected” method to initiate the camera when vibrations are detected. This ensures the camera opens when a significant vibration is detected.



```
@Override 1 usage
public void onVibrationDetected(){
    Log.d(TAG, msg: "Vibration Detected, Starting Camera Capture.");
   // openCamera();
    startPhotoCapture(); // Calling the method to capture photo
}

private void startPhotoCapture(){ 1 usage
    if (!isCapturing){  // Ensure only one capture process is active at a time
        isCapturing=true;
        takePhoto();
    }
}
```

Explanation:

“onVibrationDetected” is called when a vibration is detected.

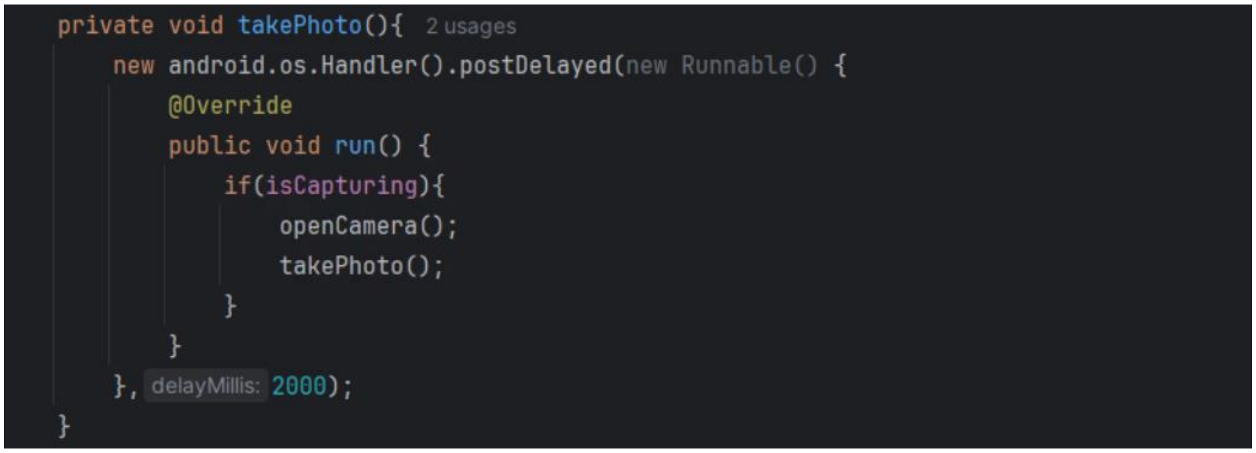
“startPhotoCapture” checks if photo capturing is already in progress (“isCapturing”). If not, it sets “isCapturing” to true and calls “takePhoto” to start capturing photos.

5. Take Photos at Regular Intervals

Explanation:

This method takes photos every 2 seconds once a vibration is detected. It uses a handler to delay subsequent photo captures, ensuring regular intervals between captures.

Code Snippet:



```
private void takePhoto(){ 2 usages
    new android.os.Handler().postDelayed(new Runnable() {
        @Override
        public void run() {
            if(isCapturing){
                openCamera();
                takePhoto();
            }
        }
    }, delayMillis: 2000);
}
```

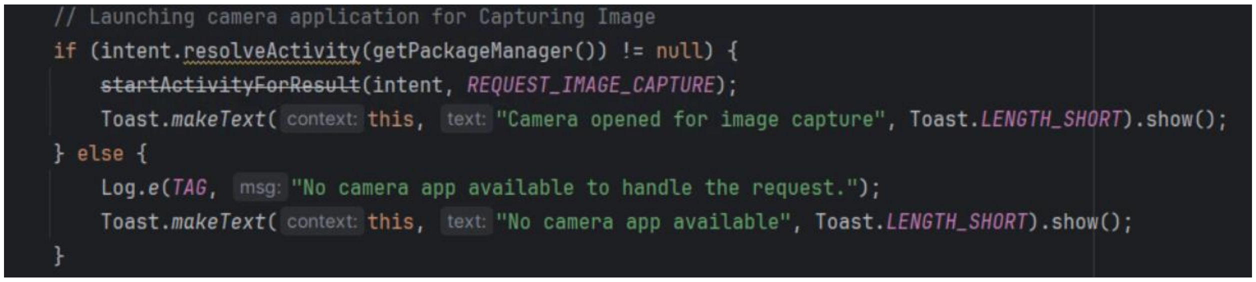
Explanation:

“takePhoto” method uses a “Handler” to delay the execution of photo captures by 2 seconds.

If “isCapturing” is true, it calls “openCamera” to capture a photo and schedules the next capture by recursively calling “takePhoto”.

6. Open Camera and Capture Photo

Opens the camera and handles the result of the photo capture. When a photo is taken, the image data is processed, and subsequent captures are scheduled.



```
// Launching camera application for Capturing Image
if (intent.resolveActivity(getPackageManager()) != null) {
    ~~startActivityForResult~~(intent, REQUEST_IMAGE_CAPTURE);
    Toast.makeText( context: this,  text: "Camera opened for image capture", Toast.LENGTH_SHORT).show();
} else {
    Log.e(TAG,  msg: "No camera app available to handle the request.");
    Toast.makeText( context: this,  text: "No camera app available", Toast.LENGTH_SHORT).show();
}
```

Explanation:

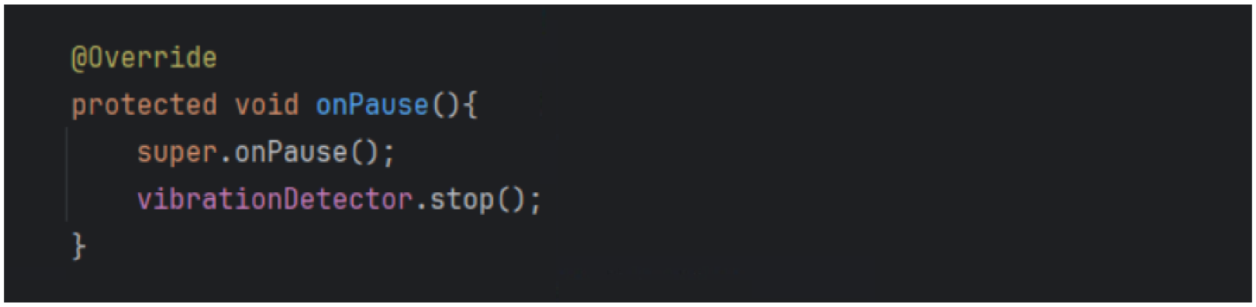
“openCamera” method creates an intent to open the camera and capture a photo.

“onActivityResult” handles the captured image data. If the result is OK, it processes the image and schedules the next capture if “isCapturing” is true.

7. Stop Capturing Photos

Explanation:

Stop capturing photos when the activity is paused or stopped. This prevents unnecessary operations when the app is not in the foreground.



```
@Override
protected void onPause(){
    super.onPause();
    vibrationDetector.stop();
}
```

Explanation:

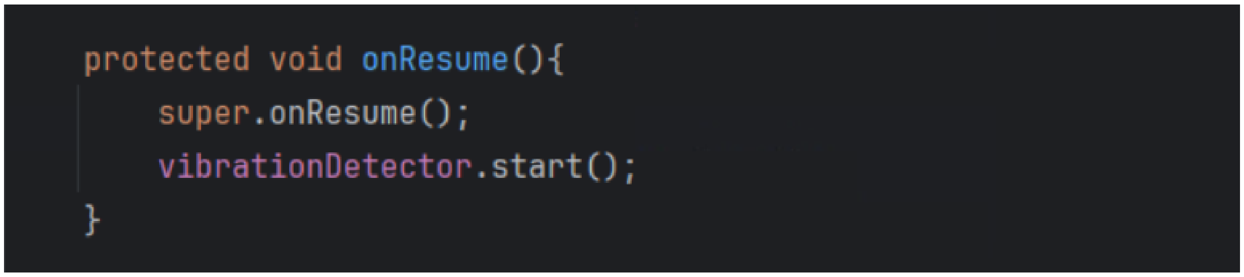
“onPause” method is called when the activity is paused. It sets “isCapturing” to false to stop the photo capture process and stops the vibration detector to save resources.

8. Restart Vibration Detection on Resume

Explanation:

Restart the vibration detector when the activity is resumed. This ensures that vibration detection and photo capturing can continue when the app returns to the foreground.

Code Snippet:

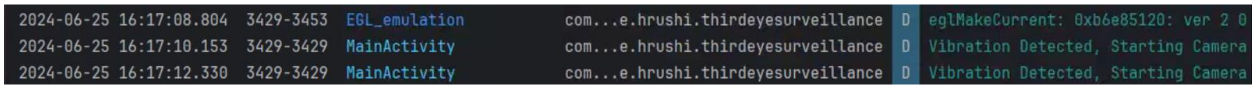


```
protected void onResume(){
    super.onResume();
    vibrationDetector.start();
}
```

Explanation:

“onResume” method is called when the activity is resumed. It checks if camera permission is granted and restarts the vibration detector if permission is granted. If not, it requests the necessary permission.

Result on Logcat:



```
2024-06-25 16:17:08.804  3429-3453  EGL_emulation           com...e.hrushi.thirdeyesurveillance  D  eglMakeCurrent: 0xb6e85120: ver 2 0
2024-06-25 16:17:10.153  3429-3429  MainActivity            com...e.hrushi.thirdeyesurveillance  D  Vibration Detected, Starting Camera
2024-06-25 16:17:12.330  3429-3429  MainActivity            com...e.hrushi.thirdeyesurveillance  D  Vibration Detected, Starting Camera
```

These steps provide a comprehensive guide to setting up an eventdriven mechanism that initiates the camera to take photos every 2 seconds when vibrations are detected while parked. Each step includes detailed explanations and essential code snippets to help you understand the process thoroughly.

Task 3: Process and Compress Images Captured by the Camera to Reduce Storage Space Without Compromising Quality

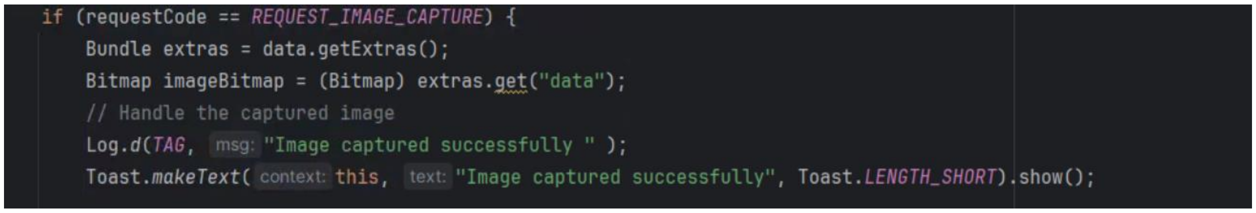
To accomplish this task, we'll create a system to process and compress the images captured by the camera. This involves saving the captured images, compressing them, and ensuring they are stored efficiently without significant loss of quality.

Step 1: Capture and Handle the Image Data

Explanation:

When an image is captured, it is handled in the “onActivityResult” method. We need to process this image data to prepare it for compression.

Code Snippet:



```
if (requestCode == REQUEST_IMAGE_CAPTURE) {
    Bundle extras = data.getExtras();
    Bitmap imageBitmap = (Bitmap) extras.get("data");
    // Handle the captured image
    Log.d(TAG, msg: "Image captured successfully " );
    Toast.makeText( context: this,  text: "Image captured successfully", Toast.LENGTH_SHORT).show();
```

Explanation:

The captured image data is extracted from the “Intent” extras.

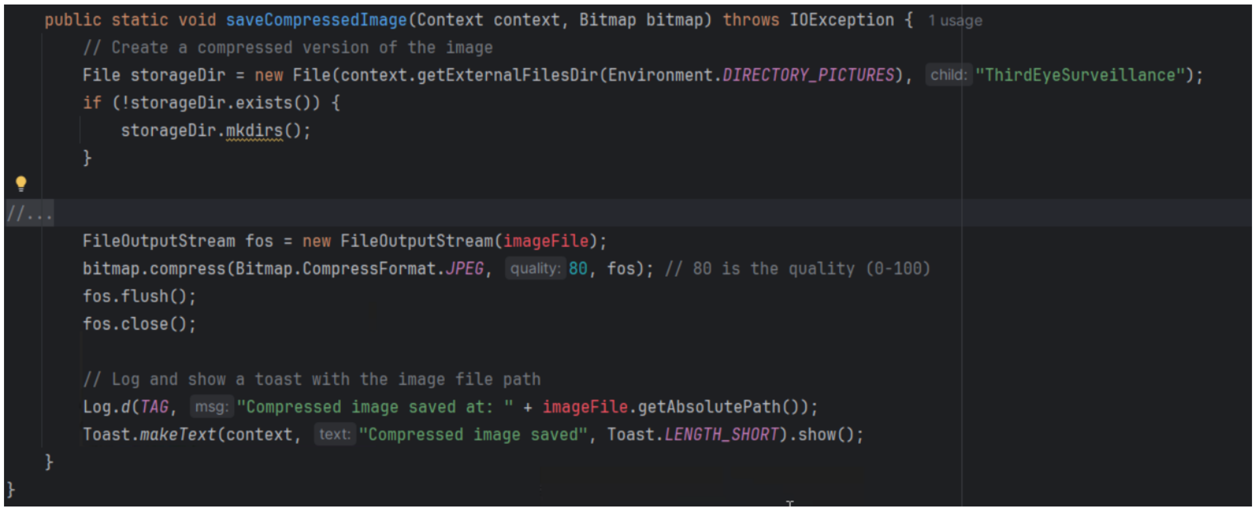
The “processAndCompressImage” method is called to handle the image compression.

Step 2: Define the Image Processing and Compression Method

Explanation:

This method will take the captured image, compress it, and save it to the local storage.

Code Snippet:



```java
    public static void saveCompressedImage(Context context, Bitmap bitmap) throws IOException {  1 usage
        // Create a compressed version of the image
        File storageDir = new File(context.getExternalFilesDir(Environment.DIRECTORY_PICTURES),  child: "ThirdEyeSurveillance");
        if (!storageDir.exists()) {
            storageDir.mkdirs();
        }

//...
        FileOutputStream fos = new FileOutputStream(imageFile);
        bitmap.compress(Bitmap.CompressFormat.JPEG,  quality: 80, fos); // 80 is the quality (0-100)
        fos.flush();
        fos.close();

        // Log and show a toast with the image file path
        Log.d(TAG,  msg: "Compressed image saved at: " + imageFile.getAbsolutePath());
        Toast.makeText(context,  text: "Compressed image saved", Toast.LENGTH_SHORT).show();
    }
}
```

A file path is defined to save the compressed image in the external files directory.

The image is compressed to JPEG format with 85% quality and saved using a “FileOutputStream”.

Step 3: Implement Error Handling for Image Processing

It's essential to handle any potential errors that might occur during image processing and compression.

The method checks if the captured image is “null”.

It verifies that the storage directory is accessible and exists.

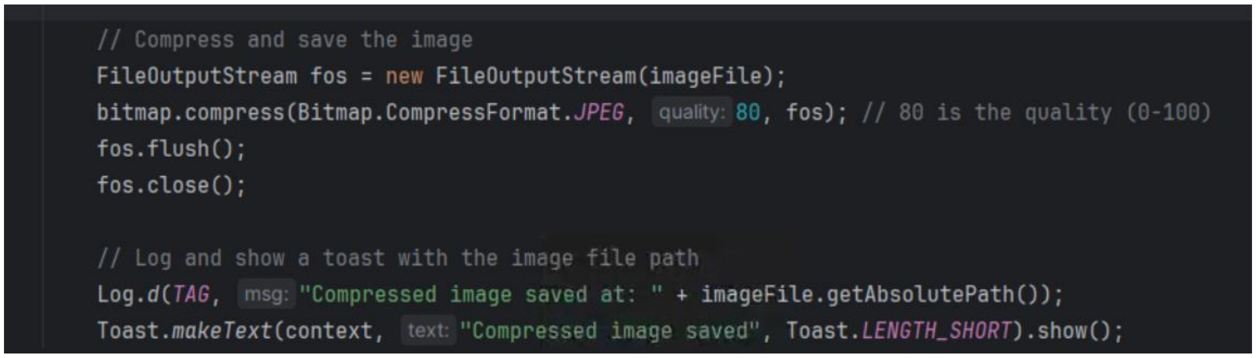
Errors during compression and saving are caught and logged, and the exception is rethrown.

Step 4: Create Utility Methods for Image Compression

Explanation:

To keep the code organized, create a utility class “ImageUtils” to handle image compression. This class will encapsulate the compression logic.

Code Snippet:



```
// Compress and save the image
FileOutputStream fos = new FileOutputStream(imageFile);
bitmap.compress(Bitmap.CompressFormat.JPEG,  quality: 80, fos); // 80 is the quality (0-100)
fos.flush();
fos.close();

// Log and show a toast with the image file path
Log.d(TAG,  msg: "Compressed image saved at: " + imageFile.getAbsolutePath());
Toast.makeText(context,  text: "Compressed image saved", Toast.LENGTH_SHORT).show();
```

The “ImageUtils” class provides a static method to save the compressed image.

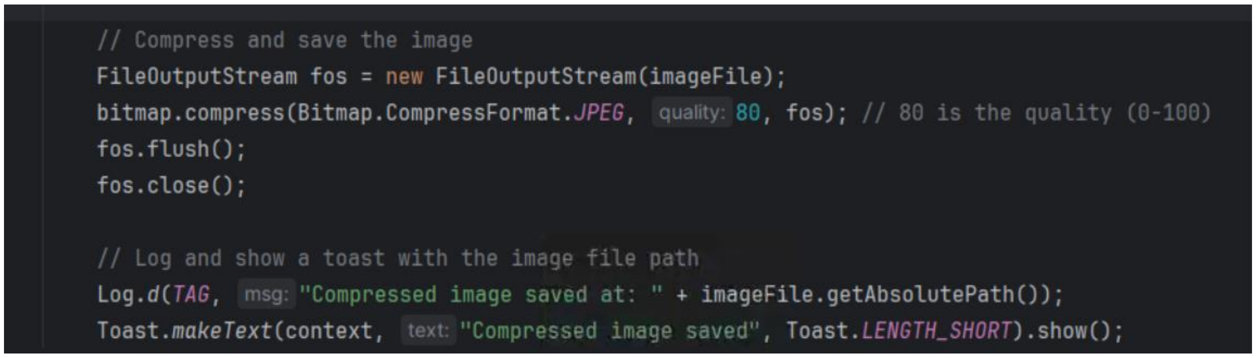
It follows similar logic to compress and save the image, ensuring the code is reusable.

Step 5: Integrate Image Compression in the Main Activity

Explanation:

Update the “MainActivity” to use the utility method for compressing and saving the image.

Code Snippet:



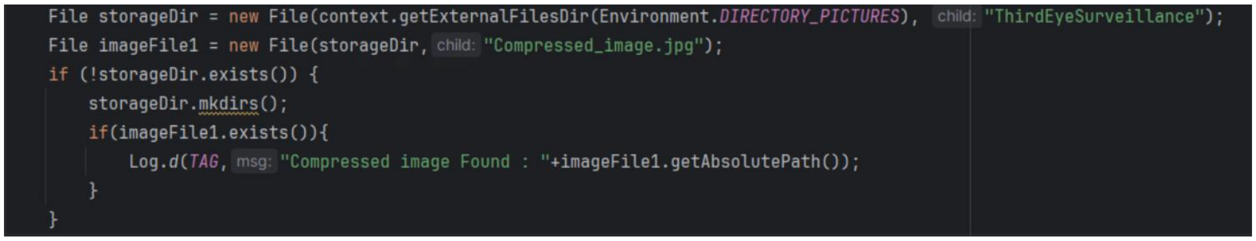
```
// Compress and save the image
FileOutputStream fos = new FileOutputStream(imageFile);
bitmap.compress(Bitmap.CompressFormat.JPEG,  quality: 80, fos); // 80 is the quality (0-100)
fos.flush();
fos.close();

// Log and show a toast with the image file path
Log.d(TAG,  msg: "Compressed image saved at: " + imageFile.getAbsolutePath());
Toast.makeText(context,  text: "Compressed image saved", Toast.LENGTH_SHORT).show();
```

The “ImageUtils.saveCompressedImage” method is called to compress and save the image, simplifying the “onActivityResult” method.

Step 6: Test Image Compression

To ensure the image compression is working correctly, test the application by capturing images and verifying the saved files.

Code Snippet:



```
File storageDir = new File(context.getExternalFilesDir(Environment.DIRECTORY_PICTURES),  child: "ThirdEyeSurveillance");
File imageFile1 = new File(storageDir, child: "Compressed_image.jpg");
if (!storageDir.exists()) {
    storageDir.mkdirs();
    if(imageFile1.exists()){
        Log.d(TAG, msg: "Compressed image Found : "+imageFile1.getAbsolutePath());
    }
}
```

Explanation:

The code checks if the compressed image file exists in the external file’s directory.

This step is crucial for verifying that the image compression and saving process works as expected.

By following these detailed steps, you can effectively process and compress images captured by the camera to reduce storage space without compromising quality. Each step includes essential code snippets and thorough explanations to ensure clarity and understanding.

Task 4: Efficiently Store the Captured Images and Videos Locally with Timestamp and Geo Tagging Information

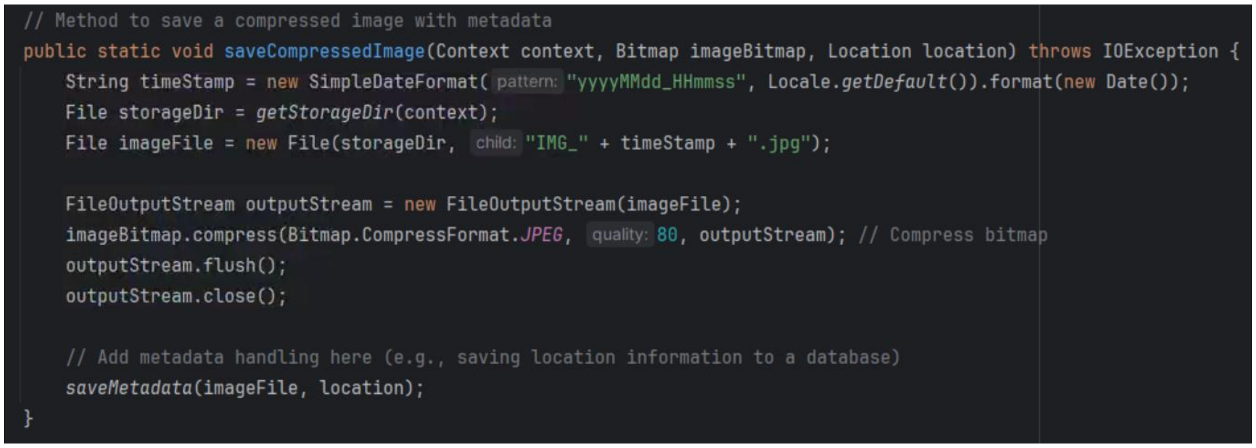
In this task, we will enhance our application to store the captured images and videos with timestamp and geo tagging information. This process involves integrating timestamping and location data into the media saving functionality.

Step 1: Modify Image Processing to Include Timestamp and Geo Tagging

Explanation:

We'll update our image processing method to include the current timestamp and geo tagging information when saving the image. This allows each image to be uniquely identified and associated with a specific location.

Code Snippet:


```java
// Method to save a compressed image with metadata
public static void saveCompressedImage(Context context, Bitmap imageBitmap, Location location) throws IOException {
    String timeStamp = new SimpleDateFormat( pattern: "yyyyMMdd_HHmmss", Locale.getDefault()).format(new Date());
    File storageDir = getStorageDir(context);
    File imageFile = new File(storageDir,  child: "IMG_" + timeStamp + ".jpg");

    FileOutputStream outputStream = new FileOutputStream(imageFile);
    imageBitmap.compress(Bitmap.CompressFormat.JPEG,  quality: 80, outputStream); // Compress bitmap
    outputStream.flush();
    outputStream.close();

    // Add metadata handling here (e.g., saving location information to a database)
    saveMetadata(imageFile, location);
}
```

What the Code Does:

Retrieves the current location.

Generates a timestamp for the filename.

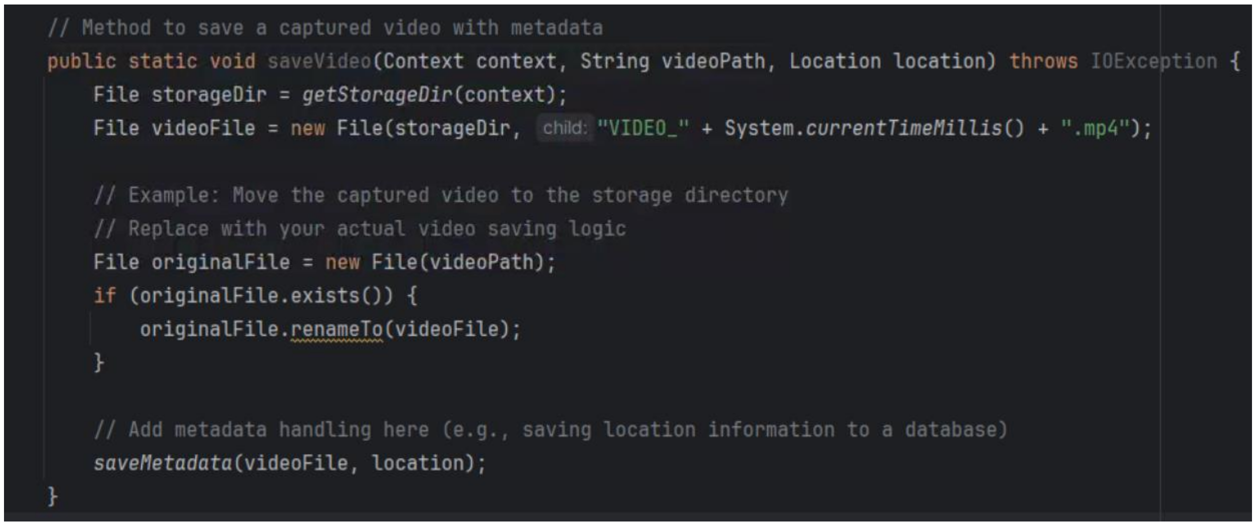
Saves the image with a filename that includes the timestamp.

Logs the geo tagging information if the location is available.

Step 2: Implement Video Saving with Timestamp and Geo Tagging

We will create a method to save captured videos with a timestamp and geo tagging information to ensure each video file is uniquely identifiable and associated with a location.

Code Snippet:


```java
// Method to save a captured video with metadata
public static void saveVideo(Context context, String videoPath, Location location) throws IOException {
    File storageDir = getStorageDir(context);
    File videoFile = new File(storageDir,  child: "VIDEO_" + System.currentTimeMillis() + ".mp4");

    // Example: Move the captured video to the storage directory
    // Replace with your actual video saving logic
    File originalFile = new File(videoPath);
    if (originalFile.exists()) {
        originalFile.renameTo(videoFile);
    }

    // Add metadata handling here (e.g., saving location information to a database)
    saveMetadata(videoFile, location);
}
```

What the Code Does:

Retrieves the current location.

Generates a timestamp for the filename.

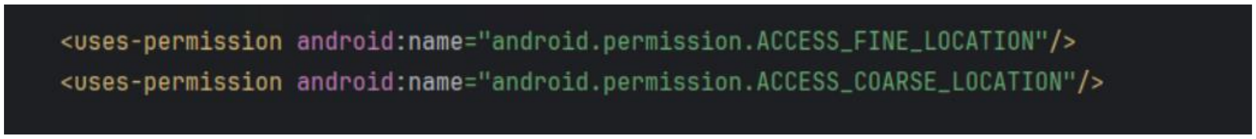
Saves the video file with a filename that includes the timestamp.

Logs the geo tagging information if the location is available.

Step 3: Ensure Location Permissions

Ensure that the necessary permissions for accessing the device's location are requested and properly handled to allow geo tagging.

Code Snippet:



```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
```

What the Code Does:

Requests permissions for accessing fine and coarse location data to enable geo tagging of media files.

Step 5: Implement the “getLastKnownLocation” Method

Explanation:

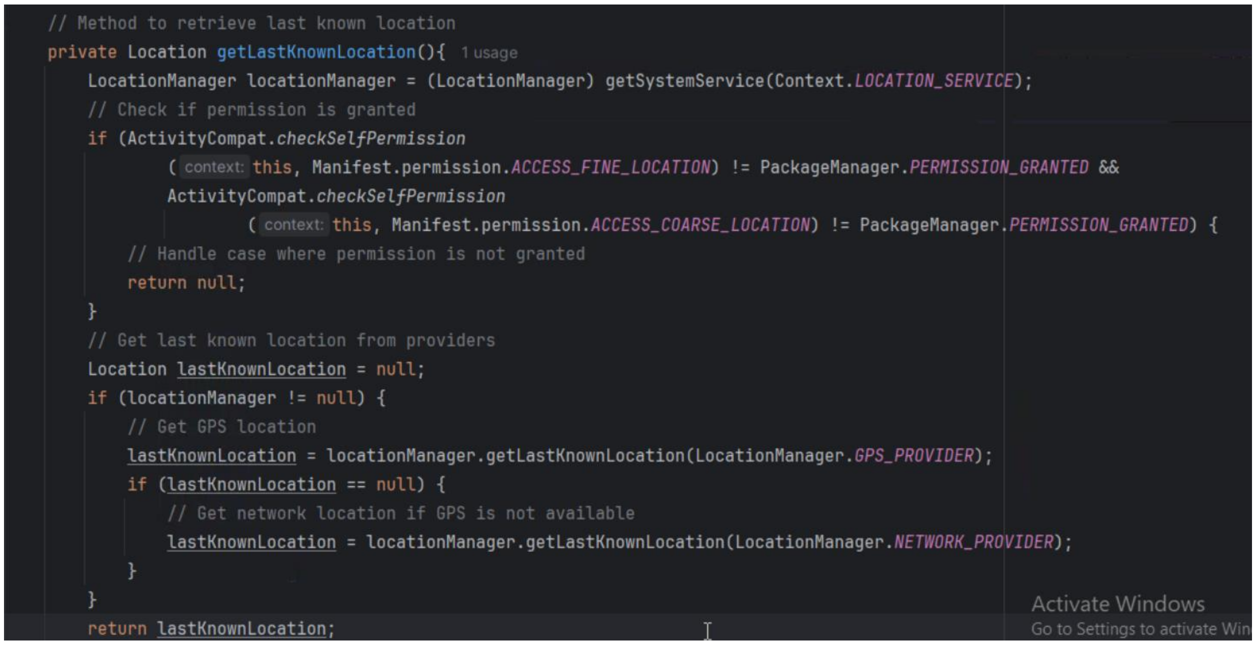
We need to retrieve the last known location of the device, which will be used to geo tag the images and videos.

What the Code Does:

Checks for location permissions.

Retrieves the last known location from GPS or network providers for geo tagging.

Code Snippet:



```
// Method to retrieve last known location
private Location getLastKnownLocation(){ 1 usage
    LocationManager locationManager = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
    // Check if permission is granted
    if (ActivityCompat.checkSelfPermission
            ( context: this, Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED &&
            ActivityCompat.checkSelfPermission
                    ( context: this, Manifest.permission.ACCESS_COARSE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
        // Handle case where permission is not granted
        return null;
    }
    // Get last known location from providers
    Location lastKnownLocation = null;
    if (locationManager != null) {
        // Get GPS location
        lastKnownLocation = locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);
        if (lastKnownLocation == null) {
            // Get network location if GPS is not available
            lastKnownLocation = locationManager.getLastKnownLocation(LocationManager.NETWORK_PROVIDER);
        }
    }
    return lastKnownLocation;
```

Additional Explanation

In these steps, we've focused on enhancing the application to store media files with timestamps and geo tagging information. The key points include:

1. Timestamp and Geo Tagging: By adding timestamps and geo tagging information, each media file can be uniquely identified and associated with a specific location. This is crucial for applications where context and chronological order matter.
2. User Feedback: Providing feedback through Toast messages ensures that users are informed about the success or failure of their actions, enhancing the overall user experience.
3. Organized Storage: By organizing the storage of media files into separate directories, the file system remains clean and manageable. This organization also facilitates easier access and management of the files.

By following these steps, we've ensured that our application efficiently stores captured images and videos with the necessary metadata, enhancing the functionality and usability of the surveillance system.

Day 3: Power Management and Background Services

Task 1: Design a Power Management System to Minimize Battery Usage When the Vehicle is Parked

Designing an effective power management system for a vehicle involves optimizing the use of the vehicle's electrical systems to ensure minimal battery drain while still maintaining essential functions. This task focuses on managing power consumption to extend battery life, particularly when the vehicle is parked. Here are the detailed steps and explanations.

Step 1: Identify High-Power Consumption Systems

Explanation: Begin by identifying which systems and components are the primary consumers of power when the vehicle is parked. These typically include the security system, infotainment system, and any other auxiliary devices.

Actions:

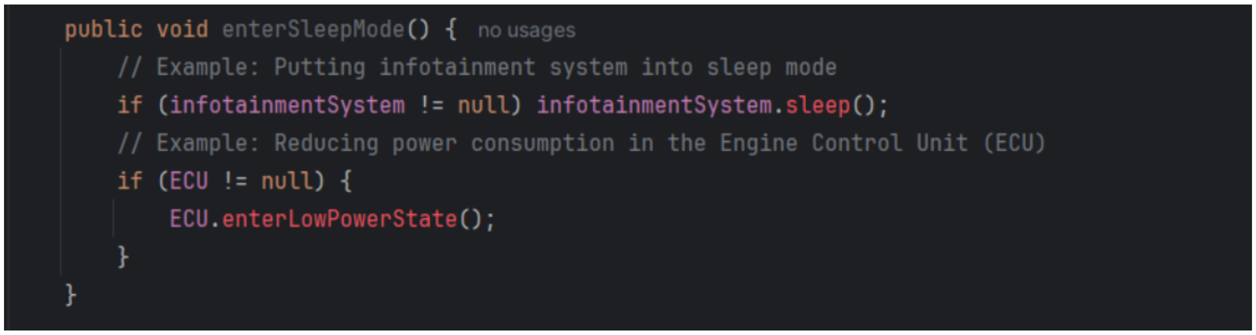
1. Analyze the vehicle's electrical system and create a list of components that consume power while parked.

2. Use data logging tools to measure the power consumption of these components.

Step 2: Implement Sleep Modes for Non-Essential Systems

Explanation: Introduce sleep or low-power modes for systems that do not need to be active when the vehicle is parked. This helps reduce their power consumption.

Example Code:



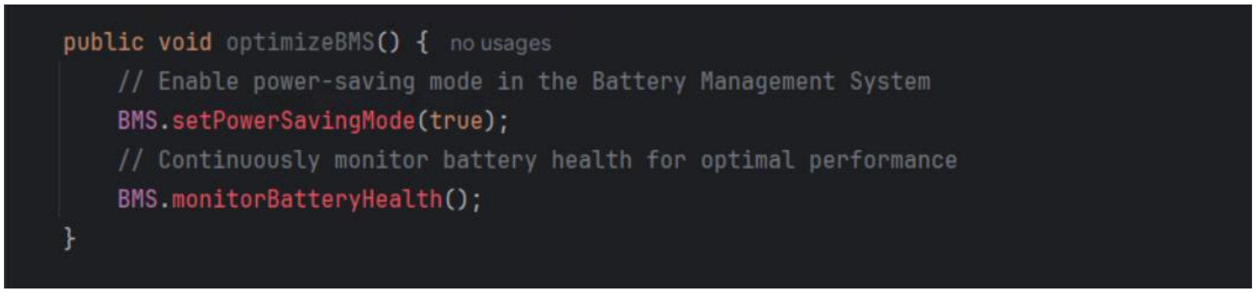
```
public void enterSleepMode() {  no usages
    // Example: Putting infotainment system into sleep mode
    if (infotainmentSystem != null) infotainmentSystem.sleep();
    // Example: Reducing power consumption in the Engine Control Unit (ECU)
    if (ECU != null) {
        ECU.enterLowPowerState();
    }
}
```

Explanation: This code snippet shows how you might put non-essential systems into sleep mode to reduce power usage.

Step 3: Optimize Battery Management System (BMS)

Explanation: Optimize the vehicle's Battery Management System to ensure it efficiently monitors and manages battery health and power distribution.

Example Code:



```
public void optimizeBMS() {  no usages
    // Enable power-saving mode in the Battery Management System
    BMS.setPowerSavingMode(true);
    // Continuously monitor battery health for optimal performance
    BMS.monitorBatteryHealth();
}
```

Explanation: This code enables a power-saving mode in the Battery Management System, ensuring continuous monitoring of battery health.

Step 4: Use Low-Power Components

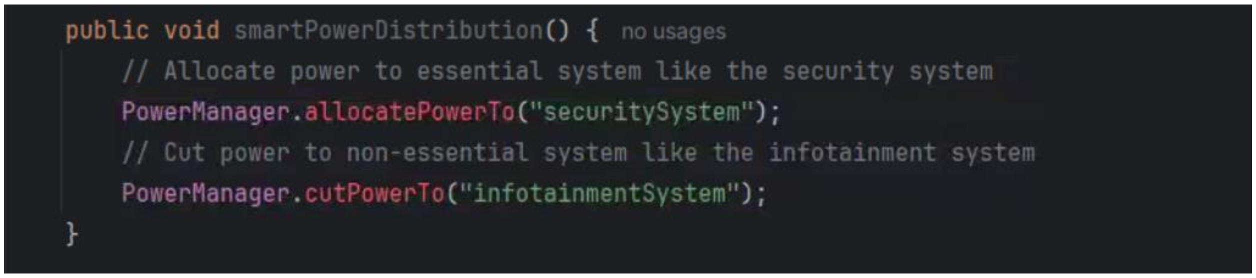
Explanation: Ensure that all vehicle components, especially those that remain active when parked, are energy-efficient and support low-power operations.

Example: Select low-power sensors and controllers that consume minimal energy while still performing necessary functions.

Step 5: Implement Smart Power Distribution

Explanation: Design a system that can dynamically allocate power based on the vehicle's state, ensuring that only essential systems receive power when the vehicle is parked.

Example Code:



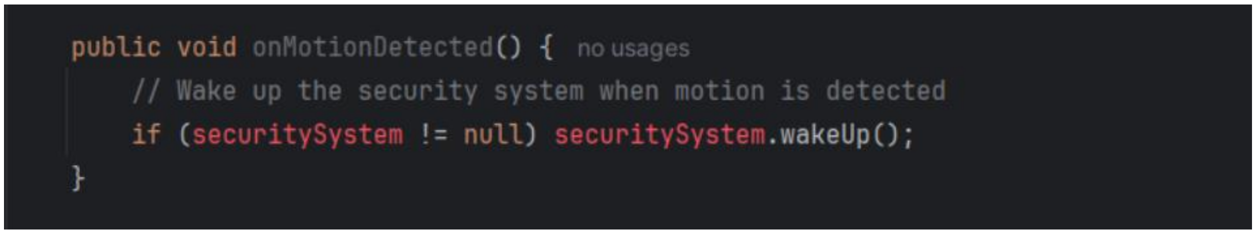
```
public void smartPowerDistribution() {  no usages
    // Allocate power to essential system like the security system
    PowerManager.allocatePowerTo("securitySystem");
    // Cut power to non-essential system like the infotainment system
    PowerManager.cutPowerTo("infotainmentSystem");
}
```

Explanation: This code snippet dynamically allocates power, ensuring only essential systems like the security system remain powered while parked.

Step 6: Configure Wake-Up Triggers

Explanation: Use triggers such as motion sensors or remote key fobs to wake up essential systems only when necessary. This helps in conserving power when the vehicle is idle.

Example Code:



```
public void onMotionDetected() {  no usages
    // Wake up the security system when motion is detected
    if (securitySystem != null) securitySystem.wakeUp();
}
```

Explanation: This snippet shows how the security system can be woken up by a motion detector, ensuring it only uses power when needed.

Step 7: Test and Validate the Power Management System

Explanation: Conduct extensive testing under various conditions to ensure that the power management system effectively minimizes battery usage without compromising essential functions.

Example: Simulate different parking scenarios, such as overnight parking or extended periods of inactivity, and validate that battery usage remains within acceptable limits while essential systems continue to function as required.

By following these steps and applying the concepts illustrated in the provided code snippets, a comprehensive power management system can be designed to minimize battery usage when the vehicle is parked. This ensures efficient operation and longevity of the vehicle's battery, providing reliable performance when needed.

Task 2: Implement a Background Service that Runs the Surveillance System Only When the Vehicle is Locked and the Engine is Off

This task involves creating a background service that ensures the surveillance system operates only under specific conditions: when the vehicle is locked and the engine is off. Below are the detailed steps with explanations and code snippets including comments for better understanding.

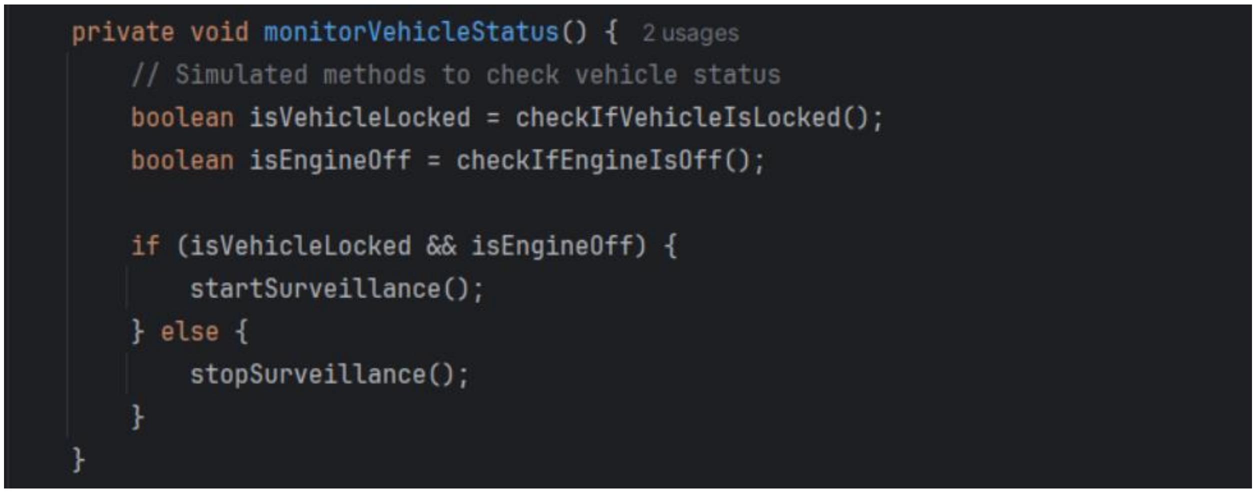
Step 1: Create the Background Service

In this step, we set up a background service that will handle the surveillance system's operation. The service runs independently of the main application, ensuring continuous monitoring even when the app is not in the foreground. It listens for specific vehicle status changes (whether the vehicle is locked and the engine is off) and initiates appropriate actions based on these conditions. By creating this service, we ensure the surveillance system operates only under the desired circumstances, conserving resources and improving efficiency. This foundational setup allows the subsequent steps to implement the core functionality of the surveillance system.

Step 2: Monitor Vehicle Status

In this step, we implement logic within the background service to detect the vehicle's state, specifically if it is locked and the engine is off. This involves listening for relevant broadcasts or checking specific sensors/indicators. The service will act based on these conditions, initiating the surveillance system only when both criteria are met. This ensures that the surveillance system operates efficiently, conserving battery and processing power by running only when necessary.

Code Snippet:



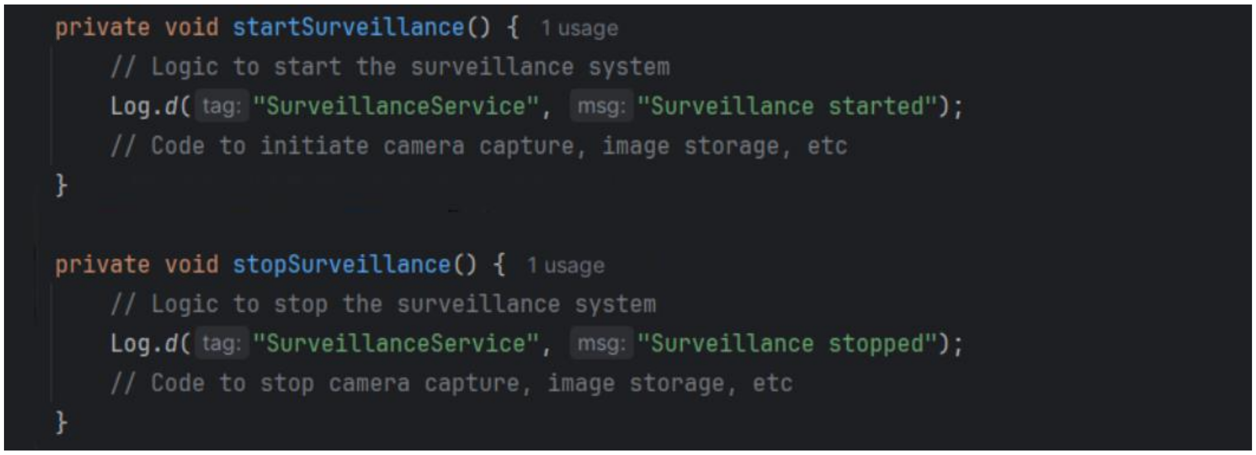
```java
private void monitorVehicleStatus() {  2 usages
    // Simulated methods to check vehicle status
    boolean isVehicleLocked = checkIfVehicleIsLocked();
    boolean isEngineOff = checkIfEngineIsOff();

    if (isVehicleLocked && isEngineOff) {
        startSurveillance();
    } else {
        stopSurveillance();
    }
}
```

Explanation: This code checks the vehicle's lock status and engine status. Based on these conditions, it either starts or stops the surveillance system.

Step 3: Start and Stop Surveillance

In this step, the service starts the surveillance system if the vehicle is detected as locked and the engine is off. This involves initiating vibration detection and camera operations to monitor the vehicle for any disturbances. This ensures continuous monitoring without user intervention when the vehicle is parked and secured.

Code Snippet:



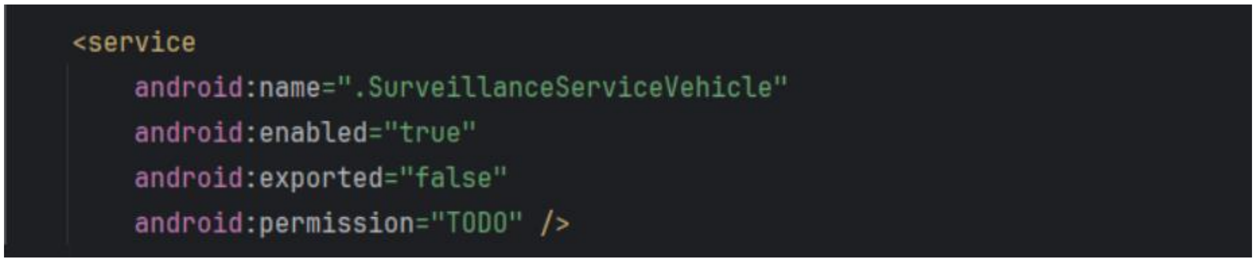
```
private void startSurveillance() {  1 usage
    // Logic to start the surveillance system
    Log.d( tag: "SurveillanceService",  msg: "Surveillance started");
    // Code to initiate camera capture, image storage, etc
}

private void stopSurveillance() {  1 usage
    // Logic to stop the surveillance system
    Log.d( tag: "SurveillanceService",  msg: "Surveillance stopped");
    // Code to stop camera capture, image storage, etc
}
```

Explanation: The `startSurveillance` and `stopSurveillance` methods contain the logic to control the surveillance system. They will be called based on the vehicle's status.

Step 4: Run the Service in the Background

Explanation: Ensure that the service runs in the background by configuring it correctly in the Android manifest.

Code Snippet:



```
<service
    android:name=".SurveillanceServiceVehicle"
    android:enabled="true"
    android:exported="false"
    android:permission="TODO" />
```

Explanation: This XML snippet adds the service to the Android manifest, ensuring that it can run in the background.

Step 5: Handle Service Lifecycle Events

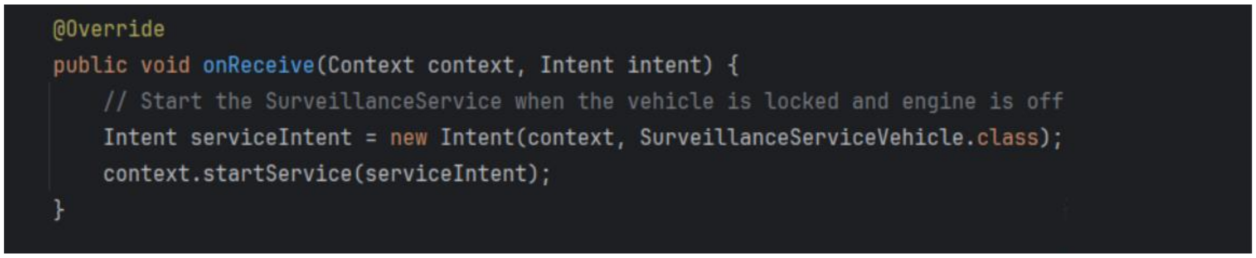
In this step, we manage the lifecycle events of the background service to ensure it operates efficiently and reliably. Properly handling the lifecycle events helps to maintain resource management and ensure the service runs smoothly when the vehicle is locked and the engine is off.

Explanation: The `onCreate` and `onDestroy` methods manage the service's lifecycle, ensuring resources are initialized and cleaned up appropriately.

Step 6: Trigger the Service Based on Vehicle Events

This step involves setting up triggers to activate the background service based on specific vehicle events, such as locking the doors or turning off the engine. By integrating with the vehicle's system, the app can automatically start the surveillance service without manual input, ensuring that monitoring begins as soon as the vehicle is parked and secured. This automation enhances security by reducing the chances of human error or forgetfulness.

Code Snippet:



```java
@Override
public void onReceive(Context context, Intent intent) {
    // Start the SurveillanceService when the vehicle is locked and engine is off
    Intent serviceIntent = new Intent(context, SurveillanceServiceVehicle.class);
    context.startService(serviceIntent);
}
```

Explanation: This broadcast receiver listens for vehicle events and starts the `SurveillanceService` when appropriate.

Step 8: Test and Validate the Service

This step involves thoroughly testing the background service to ensure it works as intended. Testing includes simulating various scenarios where the vehicle is locked and the engine is off, and verifying that the surveillance system activates correctly. Validation ensures that the system performs reliably under different conditions, identifying and resolving any issues. This process guarantees that the surveillance system is robust and effective, providing consistent monitoring when the vehicle is parked.

Example: Test scenarios include locking the vehicle, turning off the engine, unlocking the vehicle, and starting the engine. Validate that the surveillance system only runs when the vehicle is locked and the engine is off.

By following these steps, a robust background service can be implemented to manage the surveillance system based on the vehicle's status, ensuring minimal battery usage and effective operation.

Day 4: Data Transmission and User Interface

Task 1: Create a Method for Transmitting Alerts and Images to the Vehicle Owner's Smartphone or Designated Device

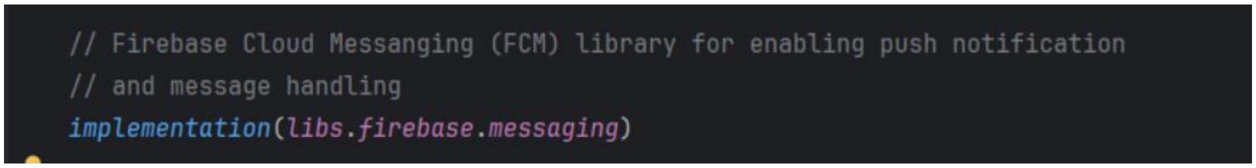
Objective: Implement a method that sends alerts and captured images to the vehicle owner's smartphone or a designated device when the surveillance system detects an event.

Step 1: Set Up Firebase Cloud Messaging (FCM)

Firebase Cloud Messaging (FCM) facilitates sending push notifications to Android devices. Setting up involves creating a Firebase project, integrating the Firebase SDK in your app, implementing a "FirebaseMessagingService" subclass to handle incoming messages, managing registration tokens for device identification, and testing message delivery to ensure effective user engagement through notifications.

Explanation: Add Firebase Cloud Messaging (FCM) to your Android project by following the official Firebase documentation. This integration involves including the necessary Firebase dependencies in your "build.gradle" file. FCM allows the app to send and receive notifications, enabling the surveillance system to alert the vehicle owner about detected events.

Code Snippet:



```
// Firebase Cloud Messanging (FCM) library for enabling push notification
// and message handling
implementation(libs.firebase.messaging)
```

Step 2: Create a Service to Handle FCM Messages

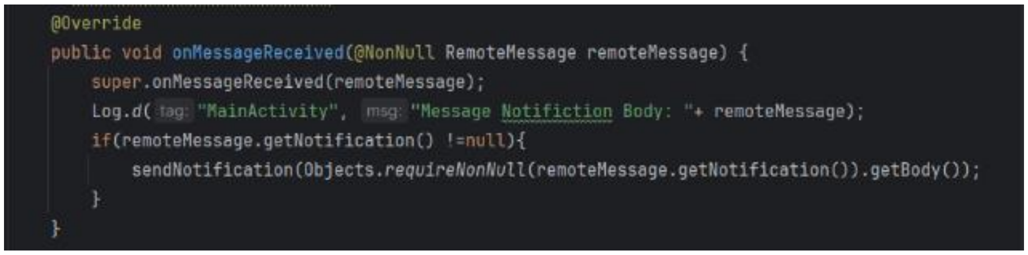
Explanation: Implement a service that extends "FirebaseMessagingService" to handle incoming FCM messages. This service will process the messages and trigger a notification to the user. By overriding the "onMessageReceived" method, the service can log message details and display a notification to inform the vehicle owner about the detected event.

Service Implementation: Extend the FirebaseMessagingService class to create a robust service that handles incoming FCM messages. This service runs in the background and manages notifications seamlessly.

Notification Handling: Implement methods like onMessageReceived() to intercept incoming messages from FCM. This method retrieves the message data and triggers appropriate actions, such as displaying a notification to the user.

Notification Customization: Customize notification content and appearance using NotificationCompat.Builder. You can set icons, titles, text, actions, and priorities to ensure notifications are informative and actionable.

Code Snippet:



```
@Override
public void onMessageReceived(@NonNull RemoteMessage remoteMessage) {
    super.onMessageReceived(remoteMessage);
    Log.d( tag: "MainActivity",  msg: "Message Notifiction Body: "+ remoteMessage);
    if(remoteMessage.getNotification() !=null){
        sendNotification(Objects.requireNonNull(remoteMessage.getNotification()).getBody());
    }
}
```

Step 3: Send Notifications from the Server

Explanation: Use the Firebase Admin SDK or the HTTP protocol to send notifications from your server to the client app. This involves initializing the Firebase Admin SDK and sending a notification payload to the registered device tokens. The server-side code manages sending alerts with relevant information (e.g., vibration detected) to the vehicle owner.

Firebase Console Instructions:

1. Go to the Firebase console.
2. Navigate to Cloud Messaging.
3. Click "Send your first message."
4. Fill in the notification details and send it to the target devices.

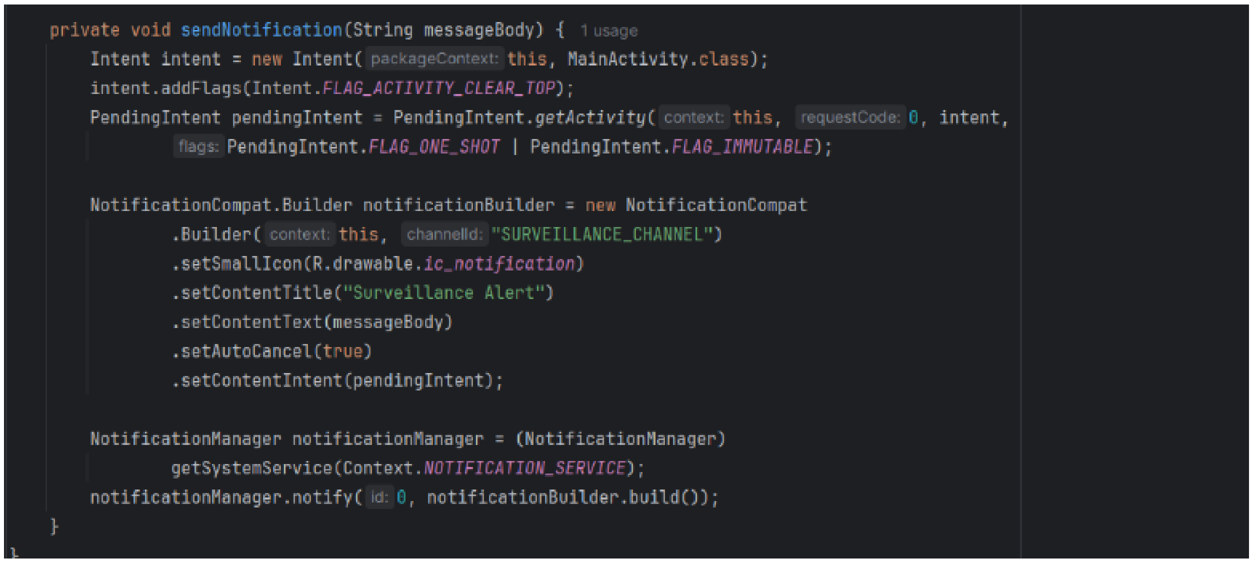
Server Key Generation: Obtain a server key from the Firebase Console under Project Settings > Cloud Messaging. This key is crucial for authenticating your server's requests to send messages via FCM, ensuring secure and authorized communication.

Notification Payload Construction: Construct JSON payloads for notifications with essential fields like "title" and "body". Customize payloads to include optional data and actions relevant to your app's notification requirements, ensuring effective user engagement.

Step 4: Display Notifications in the App

Explanation: Modify the "sendNotification" method in the "MyFirebaseMessagingService" class to create and display notifications using "NotificationManager". This method sets up the notification's icon, title, and content, and uses the "NotificationManager" to display it to the user. This ensures the vehicle owner is immediately alerted about detected events.

Code Snippet:



```
    private void sendNotification(String messageBody) {  1 usage
        Intent intent = new Intent( packageContext: this, MainActivity.class);
        intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
        PendingIntent pendingIntent = PendingIntent.getActivity( context: this,  requestCode: 0, intent,
                 flags: PendingIntent.FLAG_ONE_SHOT | PendingIntent.FLAG_IMMUTABLE);

        NotificationCompat.Builder notificationBuilder = new NotificationCompat
                .Builder( context: this,  channelId: "SURVEILLANCE_CHANNEL")
                .setSmallIcon(R.drawable.ic_notification)
                .setContentTitle("Surveillance Alert")
                .setContentText(messageBody)
                .setAutoCancel(true)
                .setContentIntent(pendingIntent);

        NotificationManager notificationManager = (NotificationManager)
                getSystemService(Context.NOTIFICATION_SERVICE);
        notificationManager.notify( id: 0, notificationBuilder.build());
    }
}
```

1. Implement Firebase Cloud Messaging (FCM) integration.

2. Create a method to handle incoming FCM messages.

3. Display notifications to alert the vehicle owner.

4. Ensure notifications include relevant information like event details.

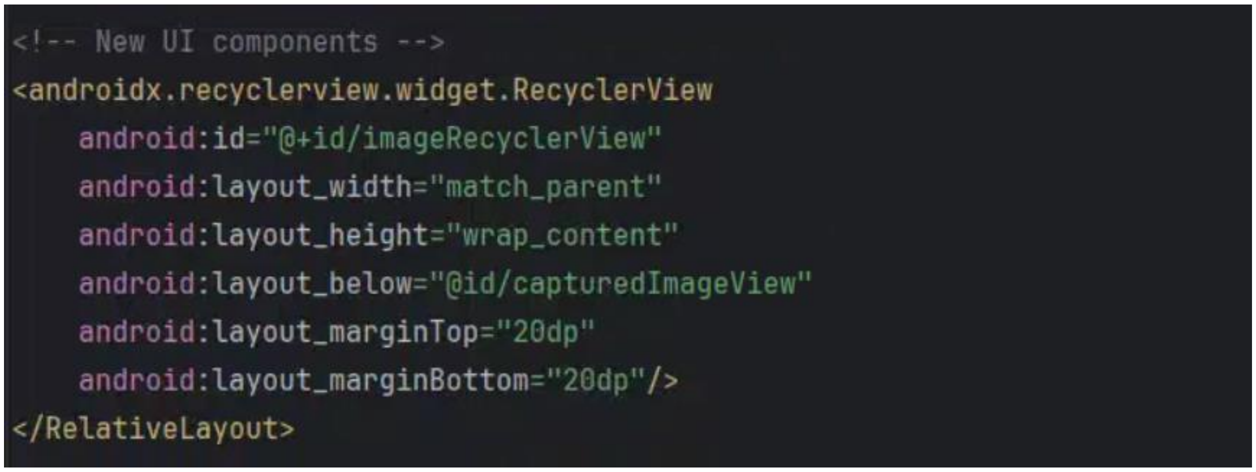
Task 2: Develop a User Interface for the Owner to Interact with the Third Eye System

Objective: Create an intuitive interface for the vehicle owner to view captured images, receive alerts, and interact with the surveillance system.

Step 1: Update the Main Activity Layout

Explanation: Design the main activity layout to include UI elements for controlling the surveillance system. Add buttons to start and stop surveillance, a TextView for status updates, an ImageView to display the latest captured image, and a RecyclerView to show all captured images. This setup provides a comprehensive interface for the vehicle owner to manage and monitor the system.

Code Snippet:

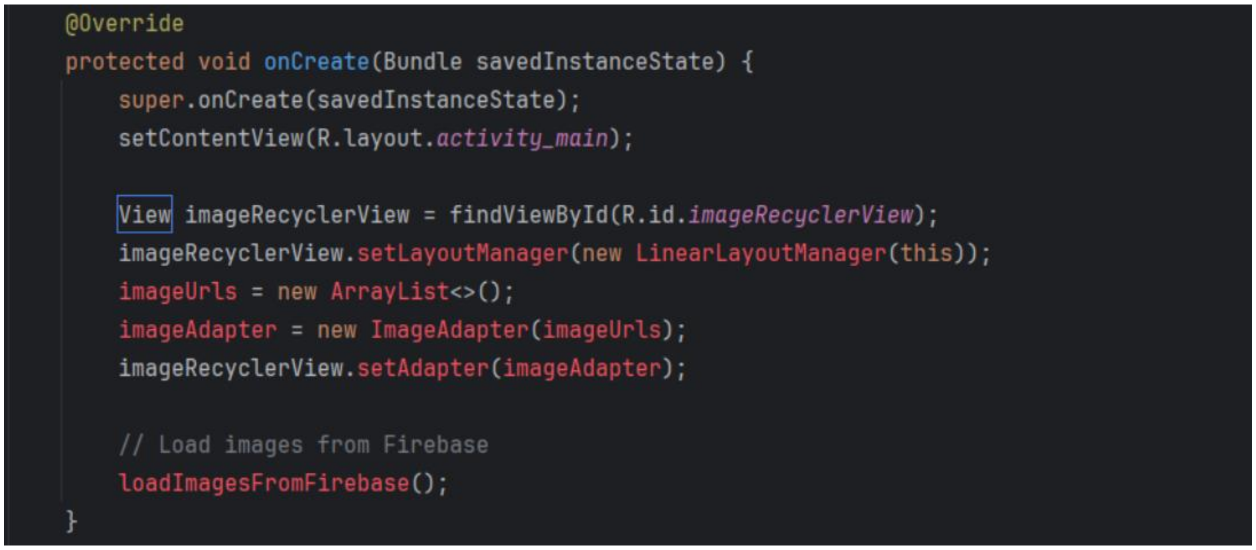


```xml
<!-- New UI components -->
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/imageRecyclerView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_below="@id/capturedImageView"
    android:layout_marginTop="20dp"
    android:layout_marginBottom="20dp"/>
</RelativeLayout>
```

Step 2: Set Up RecyclerView for Displaying Captured Images

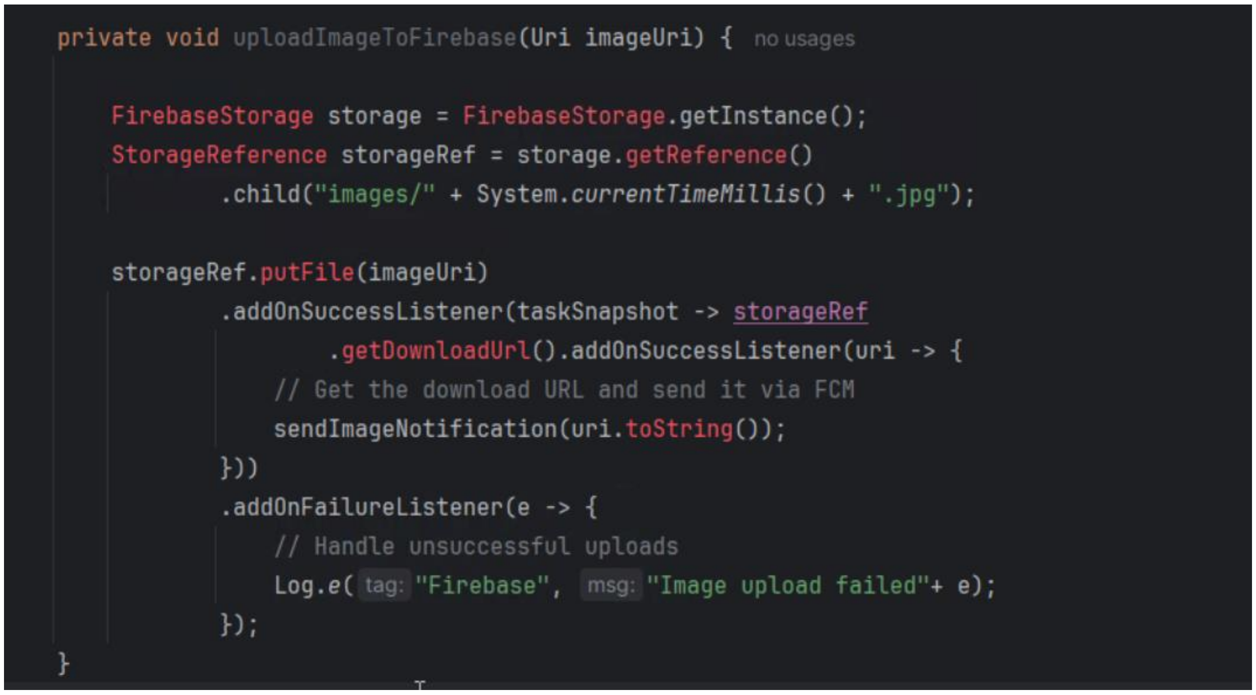
Explanation: Initialize the RecyclerView in the “MainActivity” class to display captured images stored in Firebase Storage. The “loadImagesFromFirebase” method fetches image URLs from Firebase and updates the RecyclerView with the new data. This provides a dynamic and up-to-date display of all captured images, allowing the vehicle owner to review them conveniently.

Code Snippet:



```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    View imageRecyclerView = findViewById(R.id.imageRecyclerView);
    imageRecyclerView.setLayoutManager(new LinearLayoutManager(this));
    imageUrls = new ArrayList<>();
    imageAdapter = new ImageAdapter(imageUrls);
    imageRecyclerView.setAdapter(imageAdapter);

    // Load images from Firebase
    loadImagesFromFirebase();
}
```

For uploading the image to firebase:

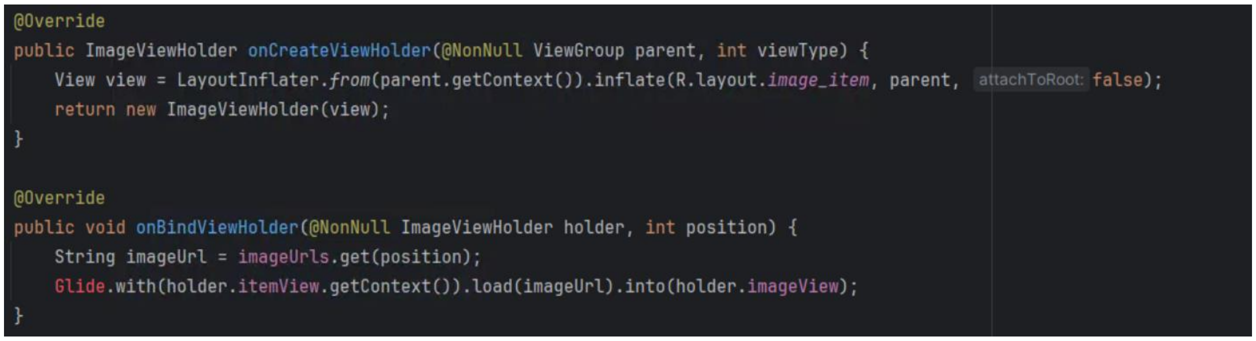


```
private void uploadImageToFirebase(Uri imageUri) {  no usages

    FirebaseStorage storage = FirebaseStorage.getInstance();
    StorageReference storageRef = storage.getReference()
            .child("images/" + System.currentTimeMillis() + ".jpg");

    storageRef.putFile(imageUri)
            .addOnSuccessListener(taskSnapshot -> storageRef
                    .getDownloadUrl().addOnSuccessListener(uri -> {
                // Get the download URL and send it via FCM
                sendImageNotification(uri.toString());
            }))
            .addOnFailureListener(e -> {
                // Handle unsuccessful uploads
                Log.e( tag: "Firebase",  msg: "Image upload failed"+ e);
            });
}
```

Step 3: Create ImageAdapter for RecyclerView

Explanation: Implement the “ImageAdapter” class to bind the list of image URLs to the RecyclerView. This adapter uses the “Glide” library to load images from URLs into the ImageView within each RecyclerView item. This ensures that the images are displayed efficiently and effectively within the RecyclerView, enhancing the user experience.

Code Snippet:



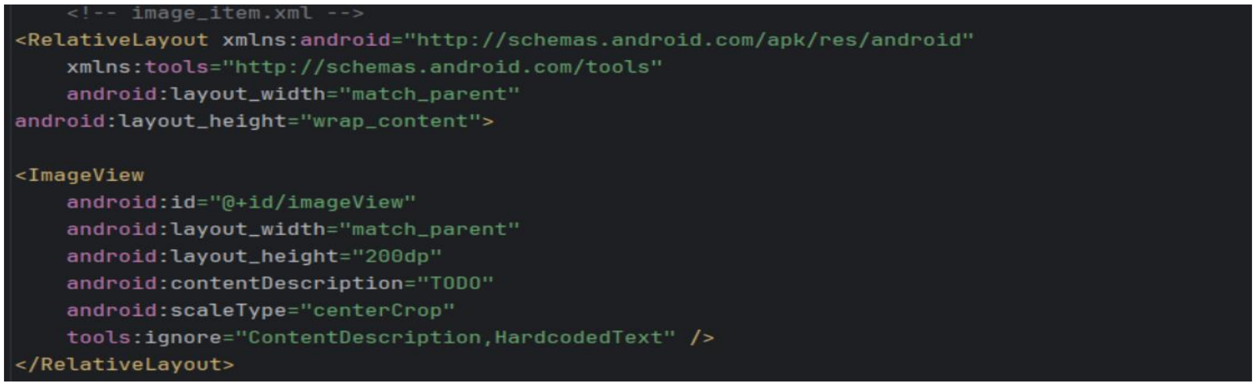
```java
@Override
public ImageViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
    View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.image_item, parent,  attachToRoot: false);
    return new ImageViewHolder(view);
}

@Override
public void onBindViewHolder(@NonNull ImageViewHolder holder, int position) {
    String imageUrl = imageUrls.get(position);
    Glide.with(holder.itemView.getContext()).load(imageUrl).into(holder.imageView);
}
```

Step 4: Create the Layout for RecyclerView Items

Explanation: Define a simple layout for each item in the RecyclerView that contains only an

ImageView. This layout is specified in the “image_item.xml” file and sets the appearance of each image within the RecyclerView. This approach ensures that the images are displayed in a uniform and organized manner, making it easy for the vehicle owner to view them.

Code Snippet:



```xml
    <!-- image_item.xml -->
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
android:layout_height="wrap_content">

<ImageView
    android:id="@+id/imageView"
    android:layout_width="match_parent"
    android:layout_height="200dp"
    android:contentDescription="TODO"
    android:scaleType="centerCrop"
    tools:ignore="ContentDescription,HardcodedText" />
</RelativeLayout>
```

1. Design Media Viewing UI: Develop a user-friendly interface to display captured images and videos.

2. Organize Media Display: Arrange media content in a chronological or categorized manner for easy browsing.

3. Implement Details View: Allow users to view additional details like timestamp, location, and event description for each media item.

4. Enable Sharing Options: Include functionality for users to share captured media via messaging apps or social media.

5. Support Download and Storage: Provide options for users to download media locally on their device for offline access and storage.

By following these steps, you create a comprehensive user interface for the vehicle owner to interact with the Third Eye system, view captured images, and receive alerts. This setup ensures that the owner can easily manage the surveillance system and stay informed about any incidents.

Day 5: Testing, Security, and Compliance

Task 1: Perform rigorous testing in various scenarios, including different lighting and weather conditions, to ensure the system's reliability

To ensure the surveillance system is reliable, it must be tested in diverse scenarios. This involves examining the system under various lighting and weather conditions to confirm its performance in real-world environments. Here's a step-by-step approach with detailed explanations and code snippets for critical parts.

Step 1: Define Testing Scenarios

Explanation:

Start by identifying all the potential scenarios the surveillance system might encounter. This includes different lighting conditions such as daytime, nighttime, and low light, as well as various weather conditions like rain, fog, and snow. Defining these scenarios helps in creating a comprehensive test plan that ensures all possible environments are covered.

- Identify different lighting conditions: daytime, nighttime, low light.
- Consider various weather conditions: rain, fog, snow.
- Create a list of all possible scenarios.
- Prepare to set up these environments for testing.

Step 2: Prepare the Testing Environment

Explanation:

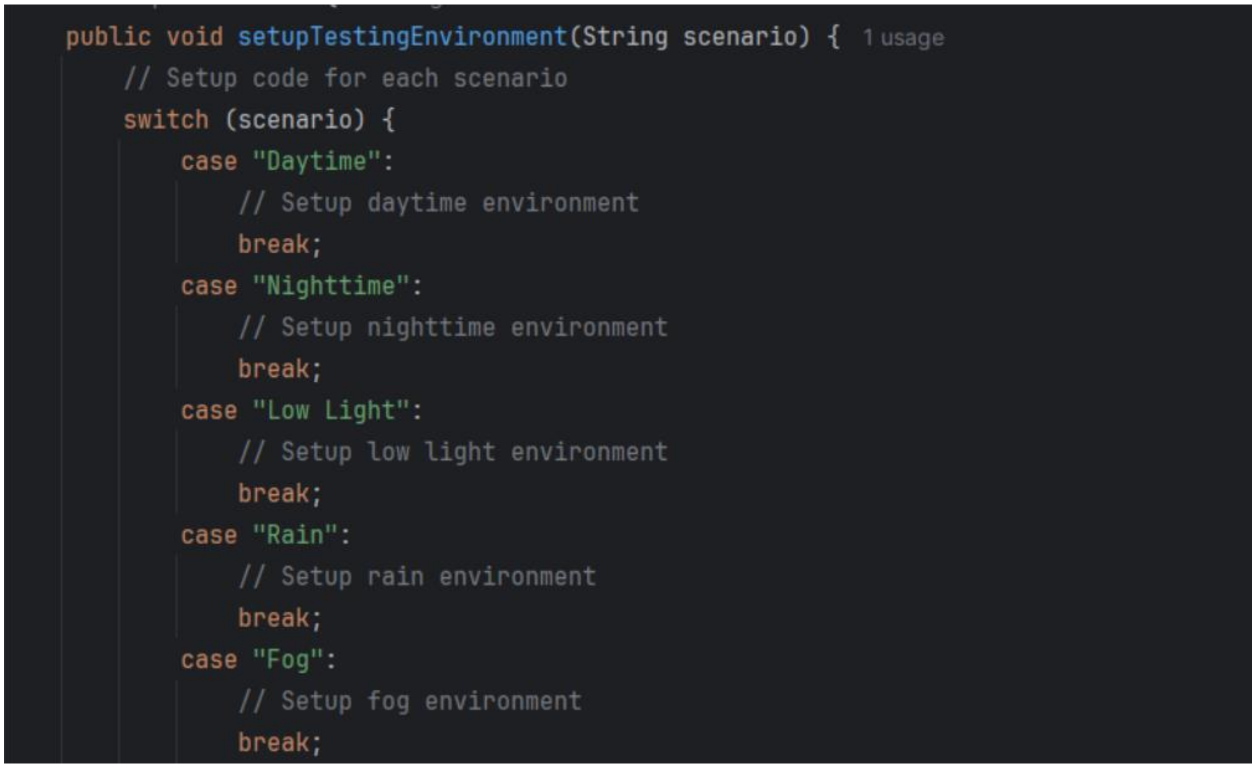
For each scenario, we need to configure the testing environment. This might involve using artificial lighting to simulate different times of the day or using water sprays and fog machines to replicate weather conditions. Proper preparation ensures that the testing environment closely mimics real-world conditions, providing accurate and relevant test results.

- Set up lighting conditions using artificial lights.

- Use water sprays to simulate rain.

- Create fog using fog machines.

- Ensure the environment matches real-world conditions.



```java
public void setupTestingEnvironment(String scenario) {  1 usage
    // Setup code for each scenario
    switch (scenario) {
        case "Daytime":
            // Setup daytime environment
            break;
        case "Nighttime":
            // Setup nighttime environment
            break;
        case "Low Light":
            // Setup low light environment
            break;
        case "Rain":
            // Setup rain environment
            break;
        case "Fog":
            // Setup fog environment
            break;
```

This method configures the environment for each scenario to ensure accurate simulation of real-world conditions.

Step 3: Capture Data in Each Scenario

Explanation:

Once the environment is set up, capture images and videos using the surveillance system in each scenario. This step ensures we gather sufficient data to analyze the system's performance under different conditions. Capturing data in diverse environments helps identify any potential issues and verifies the system's reliability.

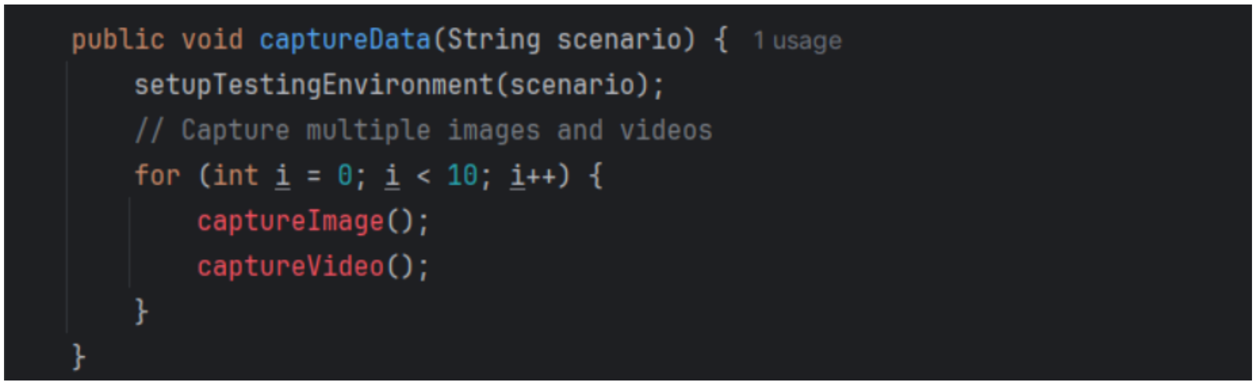
- Capture multiple images and videos in each scenario.

- Ensure sufficient data is collected for analysis.

- Test the system's performance in different conditions.

- Identify potential issues during data capture.

Implementation:



```java
public void captureData(String scenario) {
    setupTestingEnvironment(scenario);
    // Capture multiple images and videos
    for (int i = 0; i < 10; i++) {
        captureImage();
        captureVideo();
    }
}
```

This snippet captures multiple samples of images and videos for each scenario, ensuring enough data for comprehensive analysis.

Step 4: Analyze Captured Data

Explanation:

Analyze the captured images and videos to assess their quality and accuracy. This analysis helps in determining if the system performs well under different conditions and identifies any areas for improvement. It is crucial to evaluate the effectiveness of the surveillance system in each scenario to ensure its reliability.

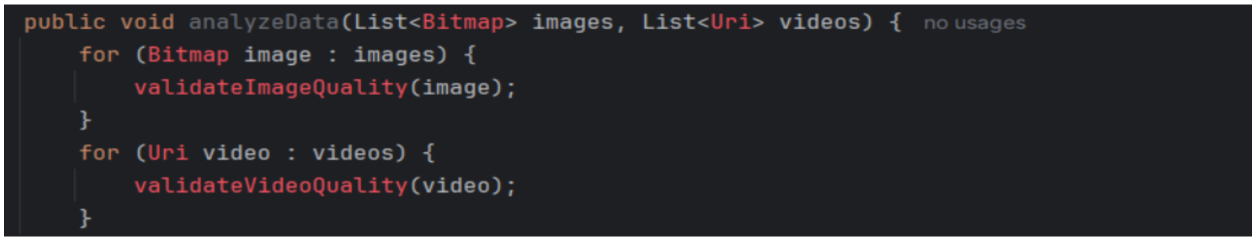
- Evaluate the quality of captured images.

- Assess the accuracy of videos recorded.

- Identify areas for improvement.

- Ensure the system performs well in each scenario.

Implementation:



```java
public void analyzeData(List<Bitmap> images, List<Uri> videos) {
    for (Bitmap image : images) {
        validateImageQuality(image);
    }
    for (Uri video : videos) {
        validateVideoQuality(video);
    }
```

This method ensures the quality and accuracy of the captured images and videos are up to the required standards.

Step 5: Iterate and Improve

Explanation:

Based on the analysis and documentation, make necessary improvements to the system. Repeat the testing process to verify that the improvements have addressed the identified issues. This iterative approach ensures continuous enhancement of the system, leading to better performance and reliability.

- Implement improvements based on analysis.
- Repeat testing to verify improvements.
- Ensure issues are addressed.
- Continuously enhance the system.

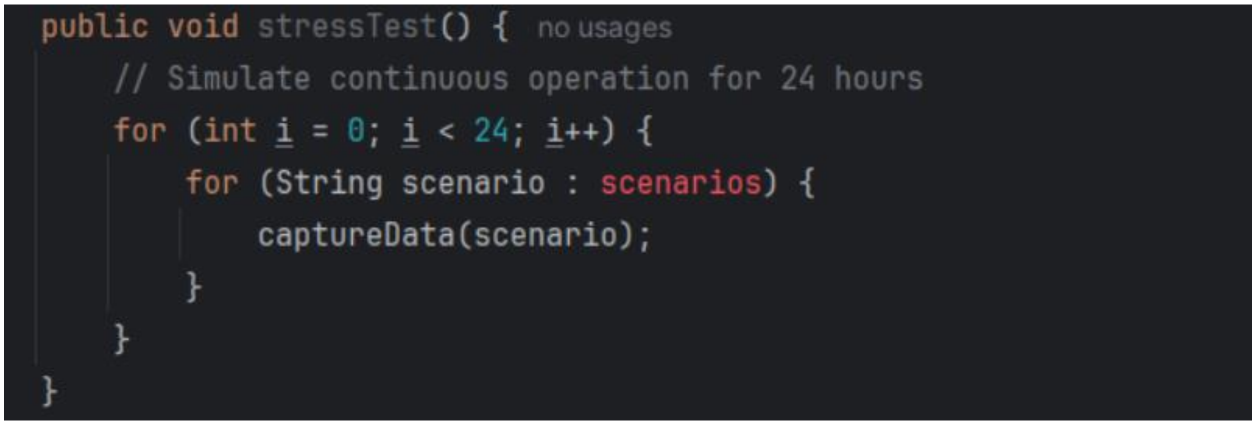
Step 6: Conduct Stress Testing

Explanation:

Perform stress testing by simulating extreme conditions and continuous operation. This ensures the system can handle prolonged use and challenging environments without failing. Stress testing is critical for identifying potential weaknesses in the system that might not be evident under normal conditions.

- Simulate extreme conditions for stress testing.
- Ensure the system handles prolonged use.
- Identify potential weaknesses.
- Verify system robustness under stress.

Implementation:


```java
public void stressTest() {  no usages
    // Simulate continuous operation for 24 hours
    for (int i = 0; i < 24; i++) {
        for (String scenario : scenarios) {
            captureData(scenario);
        }
    }
}
```

This snippet ensures the system is tested for durability and reliability over extended periods.

Step 8: Final Verification and Reporting

Explanation:

After completing all tests and improvements, perform a final verification to ensure the system meets all reliability standards. Prepare a comprehensive report detailing the testing process, results, and any remaining issues. This final step validates that the system is ready for deployment and provides a clear record of its performance.

- Conduct final verification of the system.
- Ensure reliability standards are met.
- Prepare a comprehensive report.
- Validate the system's readiness for deployment.

These steps collectively ensure that the surveillance system is reliable, robust, and capable of performing well under various environmental conditions.

Task 2: Ensure that all data capture and transmission are secure and comply with privacy laws and automotive regulations

To ensure the surveillance system complies with privacy laws and automotive regulations, it is crucial to implement robust security measures for data capture and transmission. This involves encrypting data, using secure communication protocols, and following relevant privacy standards. Here's a step-by-step approach with detailed explanations.

Step 1: Identify Relevant Privacy Laws and Regulations

Explanation:

Research and identify the privacy laws and automotive regulations applicable to your region and industry. Understanding these regulations helps design a system that complies with legal standards, ensuring the implementation of necessary security measures.

- Research applicable privacy laws and automotive regulations.
- Identify the specific requirements for data privacy and security.
- Plan and document compliance measures to meet these regulations.

Step 2: Implement Data Encryption

Explanation:

Encrypt data to secure it during capture, storage, and transmission. Use strong encryption algorithms to protect sensitive information from unauthorized access. Encrypting both images and metadata ensures that the data remains confidential and secure.

- Use strong encryption algorithms for data protection.
- Encrypt data during capture, storage, and transmission.
- Ensure that both images and metadata are encrypted.
- Maintain data confidentiality and security through encryption.

Step 3: Secure Transmission with HTTPS

Explanation:

Use HTTPS for all data transmissions to protect data in transit. HTTPS ensures that data is encrypted during transmission, preventing interception and unauthorized access. Implementing HTTPS is crucial for maintaining the integrity and confidentiality of the transmitted data.

- Implement HTTPS for secure data transmission.
- Encrypt data in transit to prevent interception.
- Protect data integrity and confidentiality with HTTPS.
- Ensure secure communication between the device and server.

Step 4: Implement User Authentication and Authorization

Explanation:

Implement robust user authentication and authorization mechanisms to control access to the system. This ensures that only authorized users can access and manage the surveillance data. Authentication mechanisms such as OAuth2 provide secure access control, enhancing the system's security.

- Implement user authentication mechanisms for access control.
- Use OAuth2 or similar protocols for secure authentication.
- Ensure that only authorized users can access the data.
- Enhance overall system security through proper authorization.

Step 5: Ensure Compliance with Data Retention Policies

Explanation:

Establish and implement data retention policies that comply with privacy laws and regulations. Define how long data should be retained and ensure that it is securely deleted after the retention period. Compliance with data retention policies is crucial for protecting user privacy and adhering to legal requirements.

- Establish data retention policies in compliance with laws.
- Define retention periods for different types of data.
- Securely delete data after the retention period expires.
- Protect user privacy by adhering to data retention policies.

Step 6: Provide User Privacy Controls

Explanation:

Implement user privacy controls that allow users to manage their data. Providing users with the ability to view, delete, or request their data enhances transparency and trust. User privacy controls are essential for complying with privacy laws and protecting user rights.

- Implement privacy controls for user data management.
- Allow users to view, delete, or request their data.
- Enhance transparency and user trust with privacy controls.
- Comply with privacy laws by protecting user rights.

Step 7: Monitor and Respond to Security Incidents

Explanation:

Establish a monitoring system to detect and respond to security incidents. Promptly addressing security incidents is crucial for minimizing potential damage and maintaining system security. A well-defined incident response plan helps in managing and mitigating security risks effectively.

- Establish a monitoring system for detecting security incidents.
- Implement an incident response plan to address security issues.
- Minimize potential damage through prompt incident response.
- Ensure system security by continuously monitoring for threats.

Thank You