# COS 211 – DEVELOPMENT OF MOBILE APPLICATION

|  |  |  |
|---|---|---|
| Name | : | Chaw Thiri Win |

## Contents

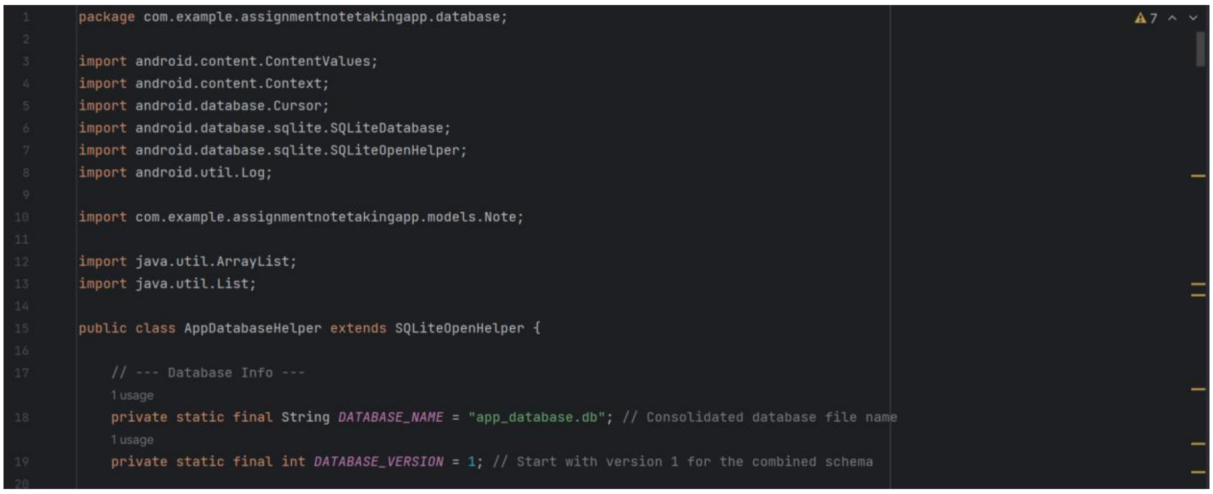
## Introduction

The design, development, and assessment of an Android note-taking application made as a component of the COS211-Development of Mobile Application curriculum are presented in this documentation. This project's main goal is to give students hands-on experience using Java and SQLite to create a real-world mobile application while following industry best practices for application testing, data management, and user interface design. Along with other features like user login, note classification, and profile customization, the app lets users write, edit, delete, and search notes. Every stage of the project has been meticulously documented, starting with the initial database and user interface design and continuing through implementation, extensive testing, and critical review.

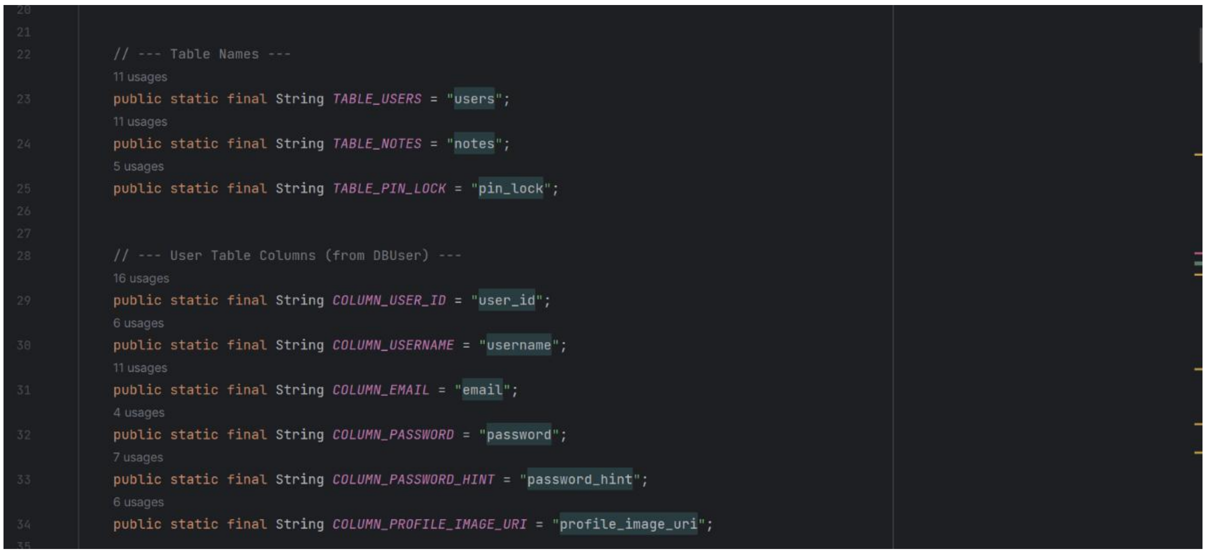
# Task 1

## 1.1. AppDatabaseHelper.java

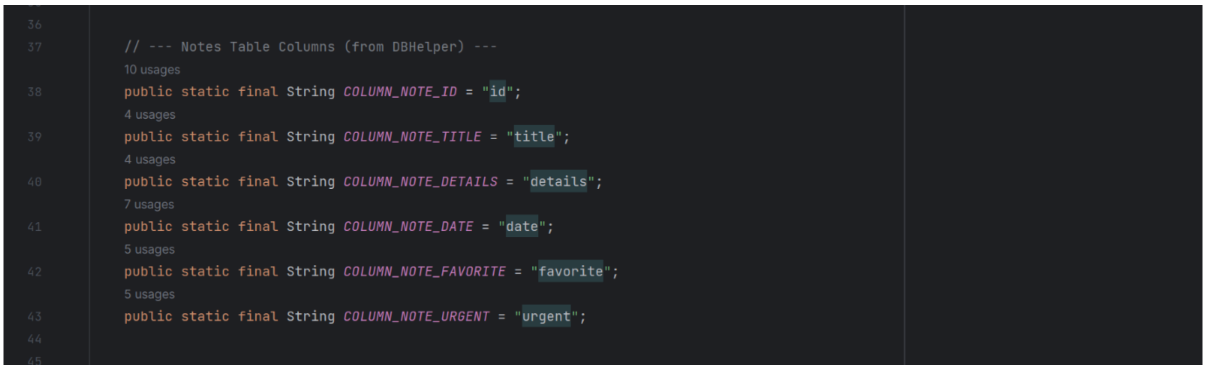
The Note-Taking App's AppDatabaseHelper class is an essential component that controls communication with the local SQLite database. It ensures effective and safe note and user data storage, retrieval, updating, and deletion within the program. Simple methods for CRUD tasks are provided by the class, which takes away the complexities of database administration and raw SQL queries. Unique identifiers, titles, details, user-related attributes, and profile image URI are among the note information that the database structure is designed to support. The onCreate method is used to create the schema by running SQL statements to create tables. Using parameterized SQL queries to guard against SQL injection and guarantee data integrity, the class provides the fundamental CRUD operations—create, read, update, and delete.



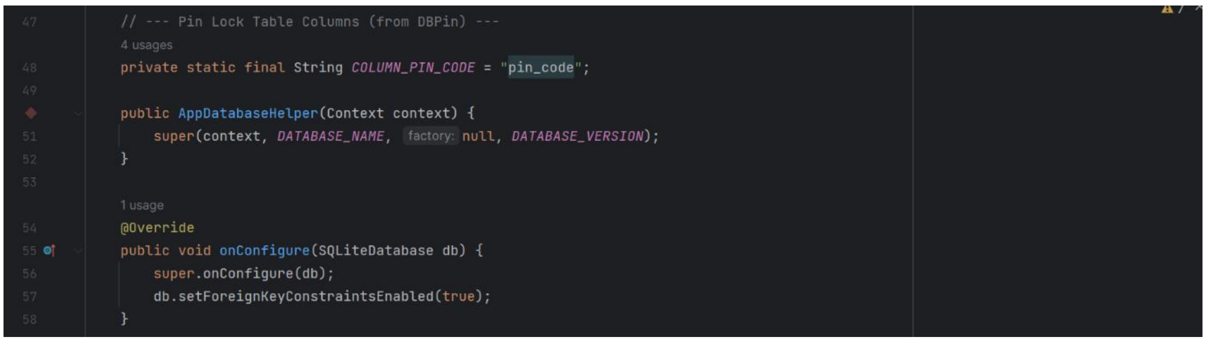
```
package com.example.assignmentnotetakingapp.database;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.util.Log;

import com.example.assignmentnotetakingapp.models.Note;

import java.util.ArrayList;
import java.util.List;

public class AppDatabaseHelper extends SQLiteOpenHelper {

    // --- Database Info ---
    1 usage
    private static final String DATABASE_NAME = "app_database.db"; // Consolidated database file name
    1 usage
    private static final int DATABASE_VERSION = 1; // Start with version 1 for the combined schema
```



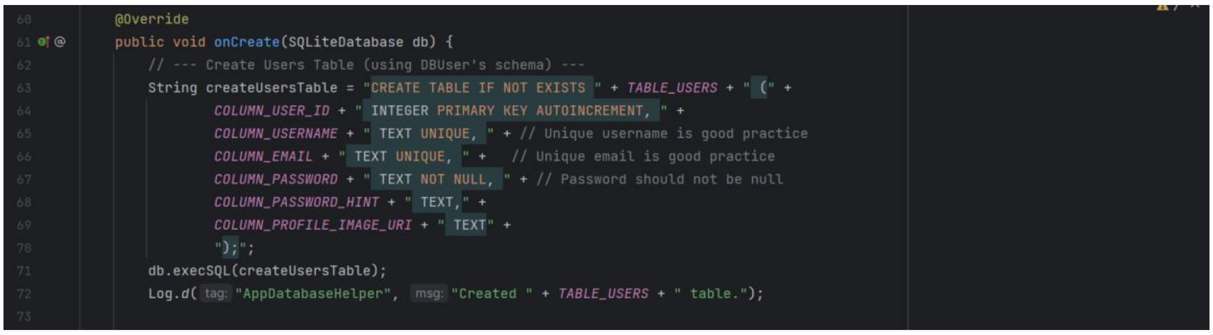
```

    // --- Table Names ---
    11 usages
    public static final String TABLE_USERS = "users";
    11 usages
    public static final String TABLE_NOTES = "notes";
    5 usages
    public static final String TABLE_PIN_LOCK = "pin_lock";


    // --- User Table Columns (from DBUser) ---
    16 usages
    public static final String COLUMN_USER_ID = "user_id";
    6 usages
    public static final String COLUMN_USERNAME = "username";
    11 usages
    public static final String COLUMN_EMAIL = "email";
    4 usages
    public static final String COLUMN_PASSWORD = "password";
    7 usages
    public static final String COLUMN_PASSWORD_HINT = "password_hint";
    6 usages
    public static final String COLUMN_PROFILE_IMAGE_URI = "profile_image_uri";
```



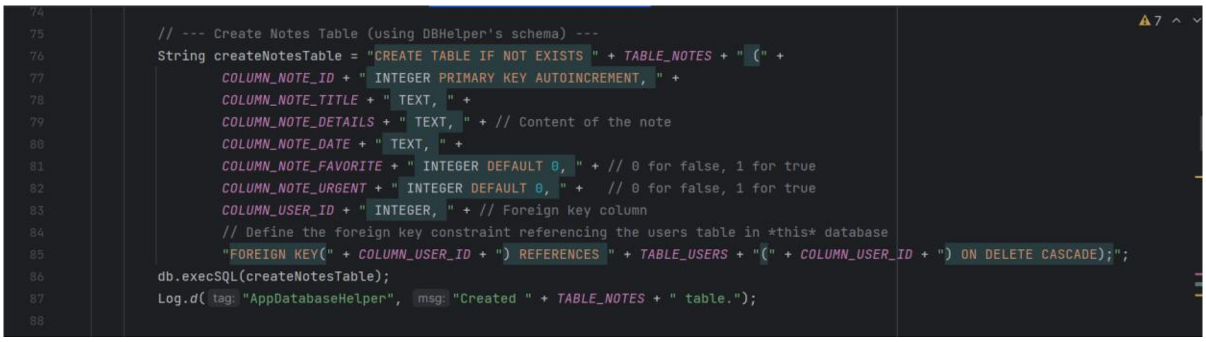
```java
// --- Notes Table Columns (from DBHelper) ---
10 usages
public static final String COLUMN_NOTE_ID = "id";
4 usages
public static final String COLUMN_NOTE_TITLE = "title";
4 usages
public static final String COLUMN_NOTE_DETAILS = "details";
7 usages
public static final String COLUMN_NOTE_DATE = "date";
5 usages
public static final String COLUMN_NOTE_FAVORITE = "favorite";
5 usages
public static final String COLUMN_NOTE_URGENT = "urgent";
```



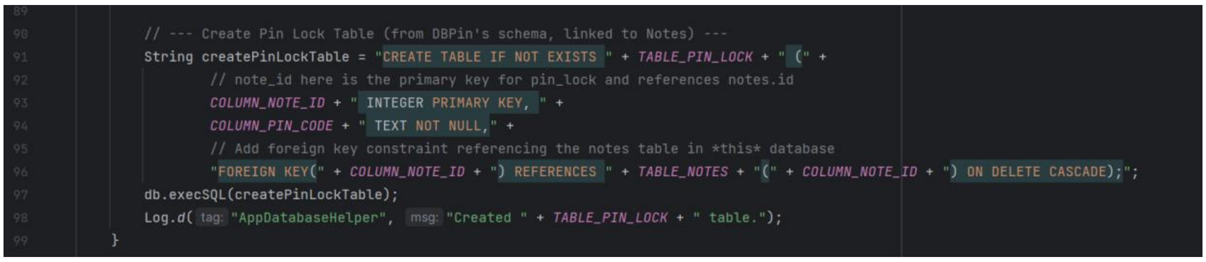
```java
// --- Pin Lock Table Columns (from DBPin) ---
4 usages
private static final String COLUMN_PIN_CODE = "pin_code";

public AppDatabaseHelper(Context context) {
    super(context, DATABASE_NAME,  factory: null, DATABASE_VERSION);
}

1 usage
@Override
public void onConfigure(SQLiteDatabase db) {
    super.onConfigure(db);
    db.setForeignKeyConstraintsEnabled(true);
}
```



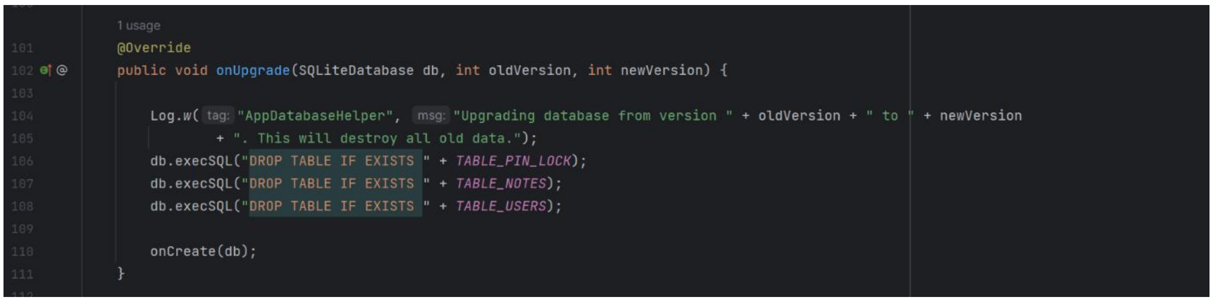
```java
@Override
public void onCreate(SQLiteDatabase db) {
    // --- Create Users Table (using DBUser's schema) ---
    String createUsersTable = "CREATE TABLE IF NOT EXISTS " + TABLE_USERS + " (" +
            COLUMN_USER_ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
            COLUMN_USERNAME + " TEXT UNIQUE, " + // Unique username is good practice
            COLUMN_EMAIL + " TEXT UNIQUE, " +   // Unique email is good practice
            COLUMN_PASSWORD + " TEXT NOT NULL, " + // Password should not be null
            COLUMN_PASSWORD_HINT + " TEXT," +
            COLUMN_PROFILE_IMAGE_URI + " TEXT" +
            ");";
    db.execSQL(createUsersTable);
    Log.d( tag: "AppDatabaseHelper",  msg: "Created " + TABLE_USERS + " table.");
```



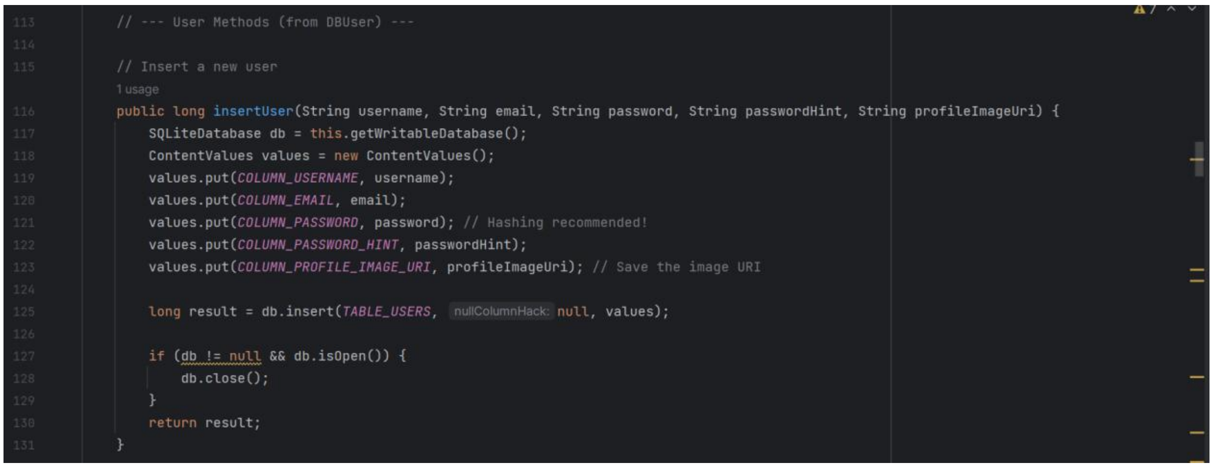
```java
    // --- Create Notes Table (using DBHelper's schema) ---
    String createNotesTable = "CREATE TABLE IF NOT EXISTS " + TABLE_NOTES + " (" +
            COLUMN_NOTE_ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
            COLUMN_NOTE_TITLE + " TEXT, " +
            COLUMN_NOTE_DETAILS + " TEXT, " + // Content of the note
            COLUMN_NOTE_DATE + " TEXT, " +
            COLUMN_NOTE_FAVORITE + " INTEGER DEFAULT 0, " + // 0 for false, 1 for true
            COLUMN_NOTE_URGENT + " INTEGER DEFAULT 0, " +   // 0 for false, 1 for true
            COLUMN_USER_ID + " INTEGER, " + // Foreign key column
            // Define the foreign key constraint referencing the users table in *this* database
            "FOREIGN KEY(" + COLUMN_USER_ID + ") REFERENCES " + TABLE_USERS + "(" + COLUMN_USER_ID + ") ON DELETE CASCADE);";
    db.execSQL(createNotesTable);
    Log.d( tag: "AppDatabaseHelper",  msg: "Created " + TABLE_NOTES + " table.");
```



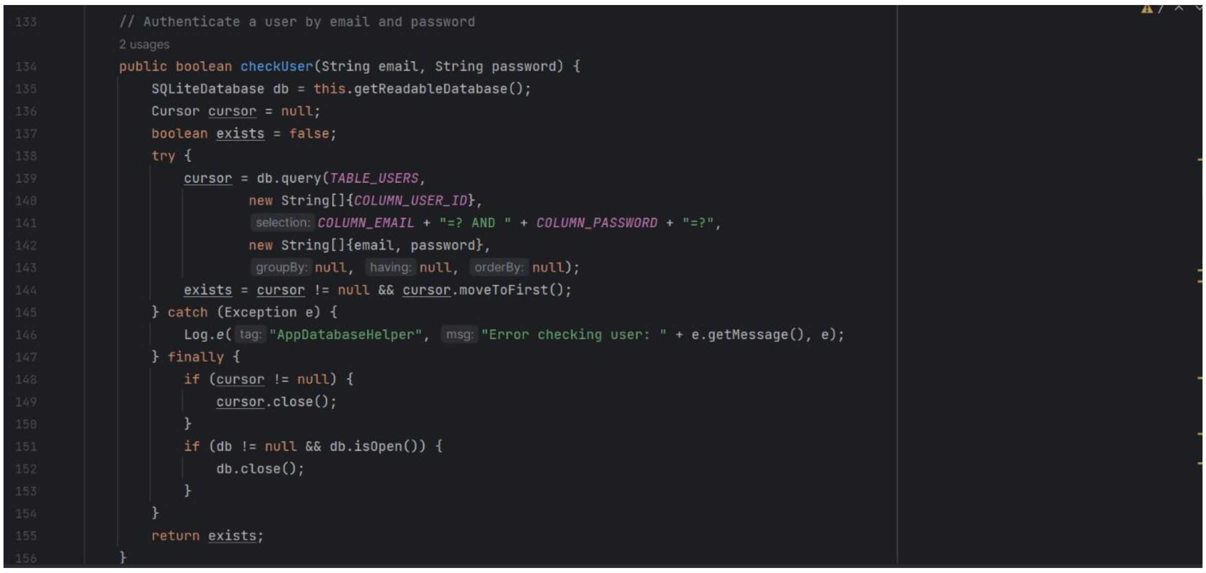
```java
    // --- Create Pin Lock Table (from DBPin's schema, linked to Notes) ---
    String createPinLockTable = "CREATE TABLE IF NOT EXISTS " + TABLE_PIN_LOCK + " (" +
            // note_id here is the primary key for pin_lock and references notes.id
            COLUMN_NOTE_ID + " INTEGER PRIMARY KEY, " +
            COLUMN_PIN_CODE + " TEXT NOT NULL," +
            // Add foreign key constraint referencing the notes table in *this* database
            "FOREIGN KEY(" + COLUMN_NOTE_ID + ") REFERENCES " + TABLE_NOTES + "(" + COLUMN_NOTE_ID + ") ON DELETE CASCADE);";
    db.execSQL(createPinLockTable);
    Log.d( tag: "AppDatabaseHelper",  msg: "Created " + TABLE_PIN_LOCK + " table.");
}
```



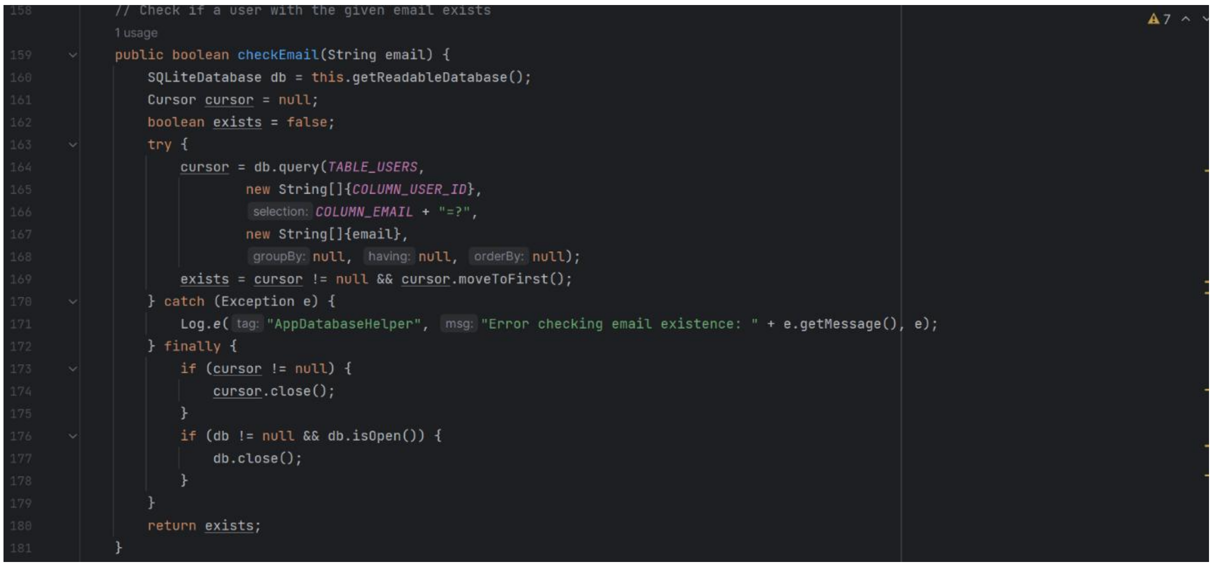
```java
1 usage
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {

    Log.w( tag: "AppDatabaseHelper",  msg: "Upgrading database from version " + oldVersion + " to " + newVersion
            + ". This will destroy all old data.");
    db.execSQL("DROP TABLE IF EXISTS " + TABLE_PIN_LOCK);
    db.execSQL("DROP TABLE IF EXISTS " + TABLE_NOTES);
    db.execSQL("DROP TABLE IF EXISTS " + TABLE_USERS);

    onCreate(db);
}
```



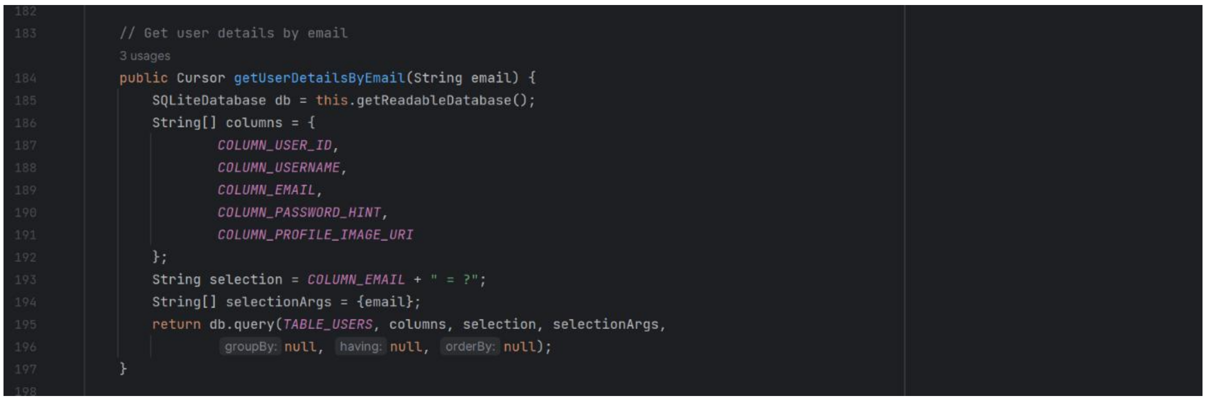
```java
// --- User Methods (from DBUser) ---

// Insert a new user
1 usage
public long insertUser(String username, String email, String password, String passwordHint, String profileImageUri) {
    SQLiteDatabase db = this.getWritableDatabase();
    ContentValues values = new ContentValues();
    values.put(COLUMN_USERNAME, username);
    values.put(COLUMN_EMAIL, email);
    values.put(COLUMN_PASSWORD, password); // Hashing recommended!
    values.put(COLUMN_PASSWORD_HINT, passwordHint);
    values.put(COLUMN_PROFILE_IMAGE_URI, profileImageUri); // Save the image URI

    long result = db.insert(TABLE_USERS,  nullColumnHack: null, values);

    if (db != null && db.isOpen()) {
        db.close();
    }
    return result;
}
```



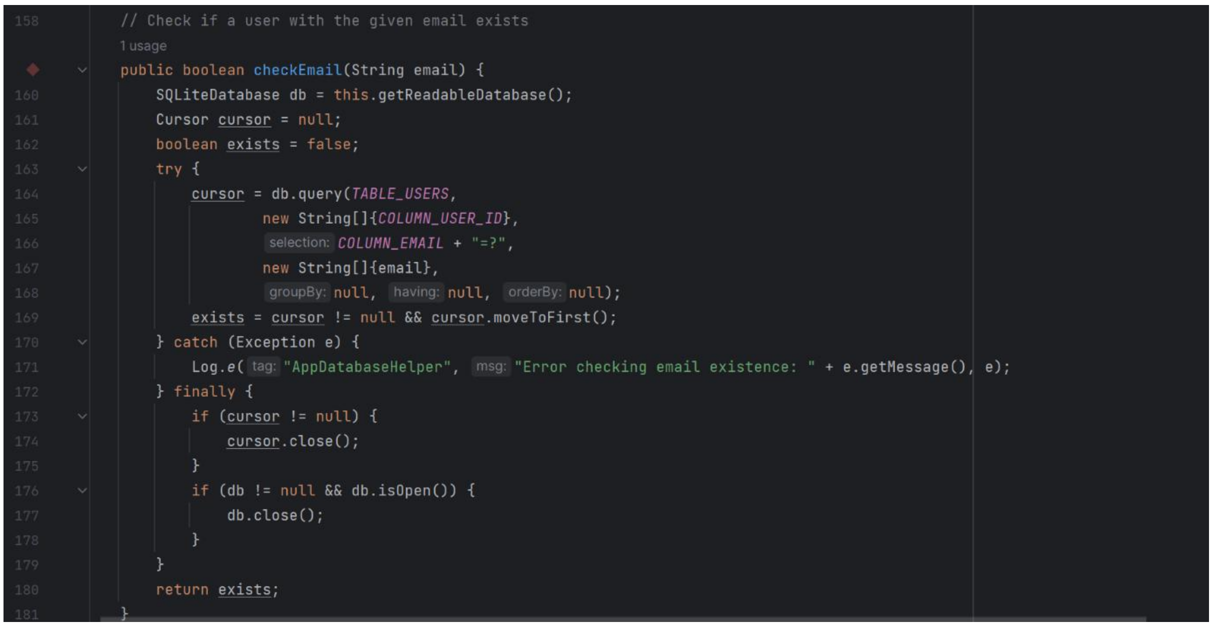
```java
// Authenticate a user by email and password
2 usages
public boolean checkUser(String email, String password) {
    SQLiteDatabase db = this.getReadableDatabase();
    Cursor cursor = null;
    boolean exists = false;
    try {
        cursor = db.query(TABLE_USERS,
                new String[]{COLUMN_USER_ID},
                 selection: COLUMN_EMAIL + "=? AND " + COLUMN_PASSWORD + "=?",
                new String[]{email, password},
                 groupBy: null,  having: null,  orderBy: null);
        exists = cursor != null && cursor.moveToFirst();
    } catch (Exception e) {
        Log.e( tag: "AppDatabaseHelper",  msg: "Error checking user: " + e.getMessage(), e);
    } finally {
        if (cursor != null) {
            cursor.close();
        }
        if (db != null && db.isOpen()) {
            db.close();
        }
    }
    return exists;
}
```



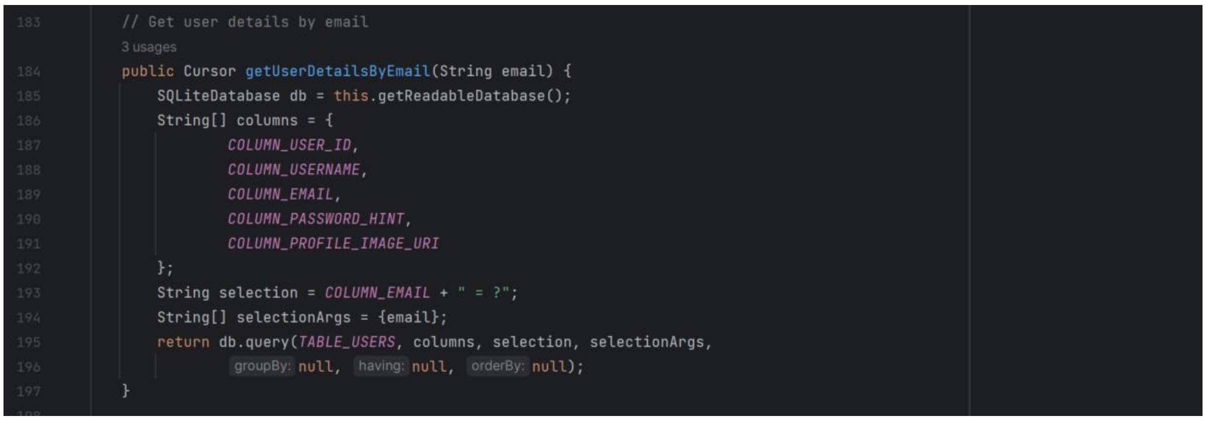
```
// Check if a user with the given email exists
1 usage
public boolean checkEmail(String email) {
    SQLiteDatabase db = this.getReadableDatabase();
    Cursor cursor = null;
    boolean exists = false;
    try {
        cursor = db.query(TABLE_USERS,
                new String[]{COLUMN_USER_ID},
                selection: COLUMN_EMAIL + "=?",
                new String[]{email},
                groupBy: null,  having: null,  orderBy: null);
        exists = cursor != null && cursor.moveToFirst();
    } catch (Exception e) {
        Log.e( tag: "AppDatabaseHelper",  msg: "Error checking email existence: " + e.getMessage(), e);
    } finally {
        if (cursor != null) {
            cursor.close();
        }
        if (db != null && db.isOpen()) {
            db.close();
        }
    }
    return exists;
}
```



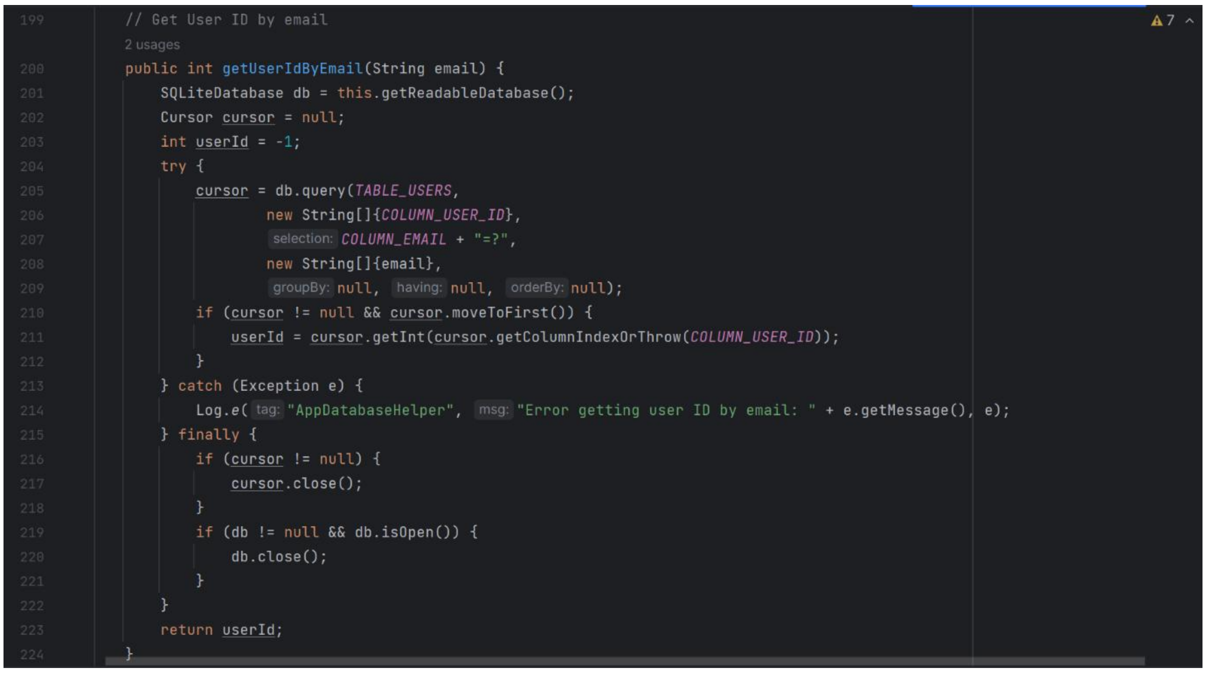
```
// Get user details by email
3 usages
public Cursor getUserDetailsByEmail(String email) {
    SQLiteDatabase db = this.getReadableDatabase();
    String[] columns = {
            COLUMN_USER_ID,
            COLUMN_USERNAME,
            COLUMN_EMAIL,
            COLUMN_PASSWORD_HINT,
            COLUMN_PROFILE_IMAGE_URI
    };
    String selection = COLUMN_EMAIL + " = ?";
    String[] selectionArgs = {email};
    return db.query(TABLE_USERS, columns, selection, selectionArgs,
            groupBy: null,  having: null,  orderBy: null);
}
```



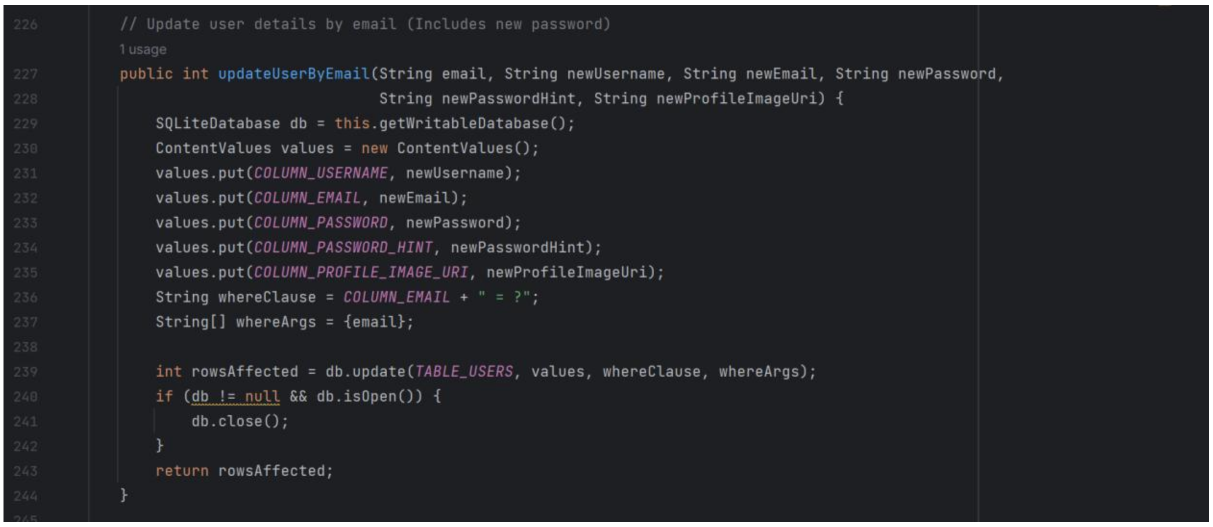
```
// Check if a user with the given email exists
1 usage
public boolean checkEmail(String email) {
    SQLiteDatabase db = this.getReadableDatabase();
    Cursor cursor = null;
    boolean exists = false;
    try {
        cursor = db.query(TABLE_USERS,
                new String[]{COLUMN_USER_ID},
                selection: COLUMN_EMAIL + "=?",
                new String[]{email},
                groupBy: null,  having: null,  orderBy: null);
        exists = cursor != null && cursor.moveToFirst();
    } catch (Exception e) {
        Log.e( tag: "AppDatabaseHelper",  msg: "Error checking email existence: " + e.getMessage(), e);
    } finally {
        if (cursor != null) {
            cursor.close();
        }
        if (db != null && db.isOpen()) {
            db.close();
        }
    }
    return exists;
}
```



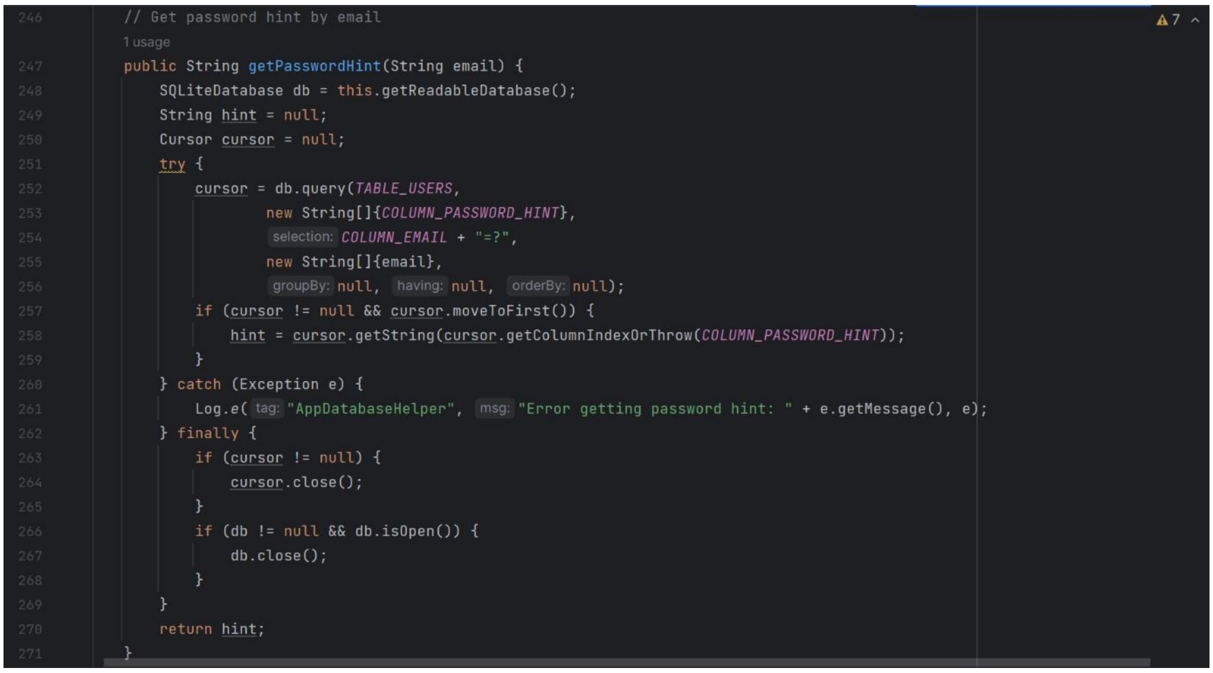
```java
// Get user details by email
3 usages
public Cursor getUserDetailsByEmail(String email) {
    SQLiteDatabase db = this.getReadableDatabase();
    String[] columns = {
            COLUMN_USER_ID,
            COLUMN_USERNAME,
            COLUMN_EMAIL,
            COLUMN_PASSWORD_HINT,
            COLUMN_PROFILE_IMAGE_URI
    };
    String selection = COLUMN_EMAIL + " = ?";
    String[] selectionArgs = {email};
    return db.query(TABLE_USERS, columns, selection, selectionArgs,
            groupBy: null, having: null, orderBy: null);
}
```



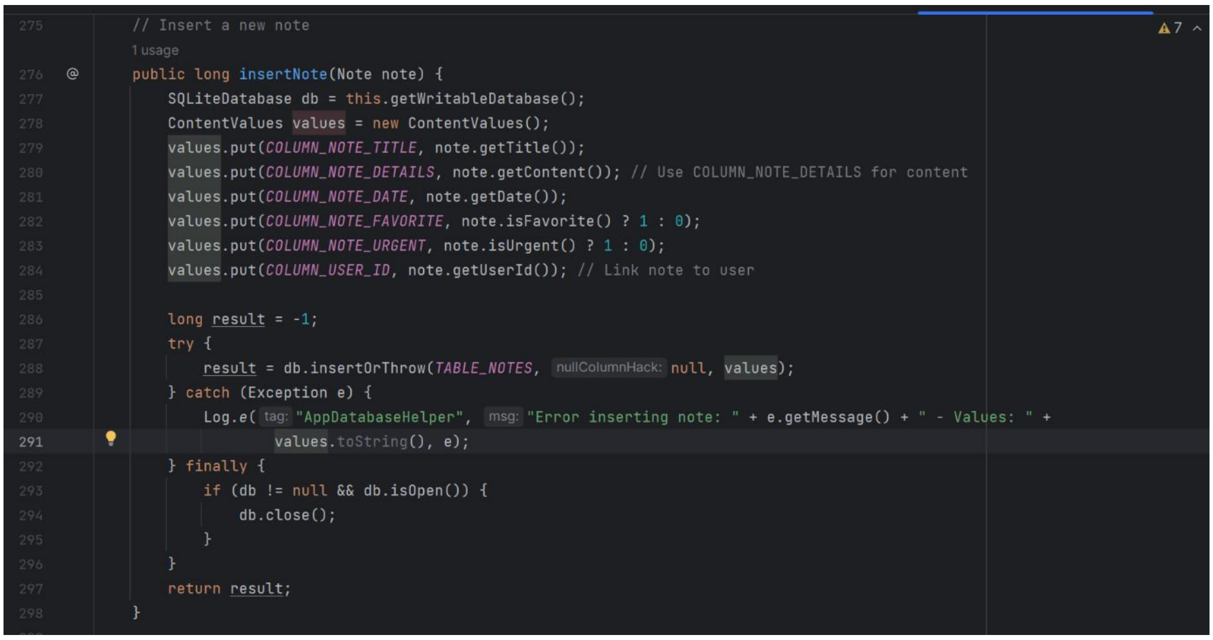
```java
// Get User ID by email
2 usages
public int getUserIdByEmail(String email) {
    SQLiteDatabase db = this.getReadableDatabase();
    Cursor cursor = null;
    int userId = -1;
    try {
        cursor = db.query(TABLE_USERS,
                new String[]{COLUMN_USER_ID},
                selection: COLUMN_EMAIL + "=?",
                new String[]{email},
                groupBy: null, having: null, orderBy: null);
        if (cursor != null && cursor.moveToFirst()) {
            userId = cursor.getInt(cursor.getColumnIndexOrThrow(COLUMN_USER_ID));
        }
    } catch (Exception e) {
        Log.e( tag: "AppDatabaseHelper", msg: "Error getting user ID by email: " + e.getMessage(), e);
    } finally {
        if (cursor != null) {
            cursor.close();
        }
        if (db != null && db.isOpen()) {
            db.close();
        }
    }
    return userId;
}
```



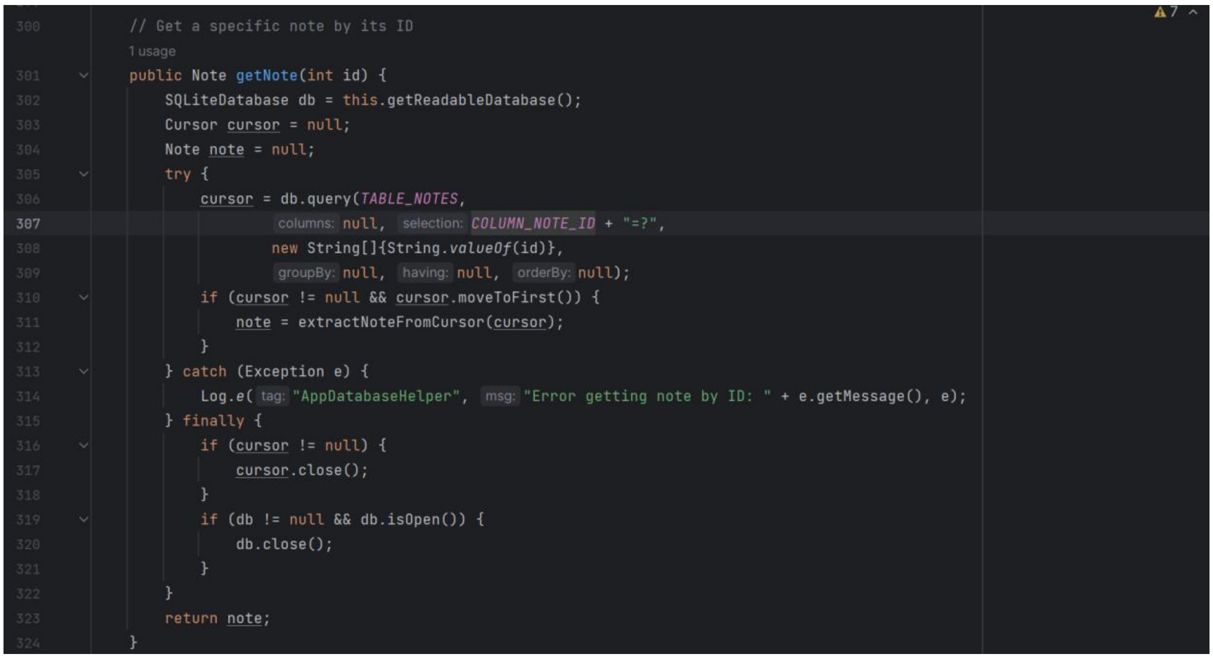
```java
// Update user details by email (Includes new password)
1 usage
public int updateUserByEmail(String email, String newUsername, String newEmail, String newPassword,
                             String newPasswordHint, String newProfileImageUri) {
    SQLiteDatabase db = this.getWritableDatabase();
    ContentValues values = new ContentValues();
    values.put(COLUMN_USERNAME, newUsername);
    values.put(COLUMN_EMAIL, newEmail);
    values.put(COLUMN_PASSWORD, newPassword);
    values.put(COLUMN_PASSWORD_HINT, newPasswordHint);
    values.put(COLUMN_PROFILE_IMAGE_URI, newProfileImageUri);
    String whereClause = COLUMN_EMAIL + " = ?";
    String[] whereArgs = {email};

    int rowsAffected = db.update(TABLE_USERS, values, whereClause, whereArgs);
    if (db != null && db.isOpen()) {
        db.close();
    }
    return rowsAffected;
}
```



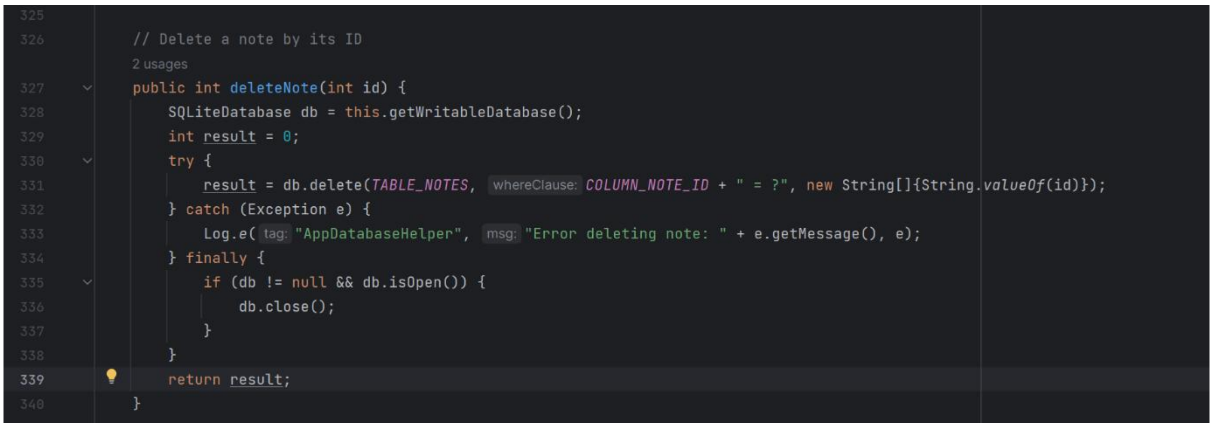
```java
// Get password hint by email
1 usage
public String getPasswordHint(String email) {
    SQLiteDatabase db = this.getReadableDatabase();
    String hint = null;
    Cursor cursor = null;
    try {
        cursor = db.query(TABLE_USERS,
                new String[]{COLUMN_PASSWORD_HINT},
                selection: COLUMN_EMAIL + "=?",
                new String[]{email},
                groupBy: null,  having: null,  orderBy: null);
        if (cursor != null && cursor.moveToFirst()) {
            hint = cursor.getString(cursor.getColumnIndexOrThrow(COLUMN_PASSWORD_HINT));
        }
    } catch (Exception e) {
        Log.e( tag: "AppDatabaseHelper",  msg: "Error getting password hint: " + e.getMessage(), e);
    } finally {
        if (cursor != null) {
            cursor.close();
        }
        if (db != null && db.isOpen()) {
            db.close();
        }
    }
    return hint;
}
```



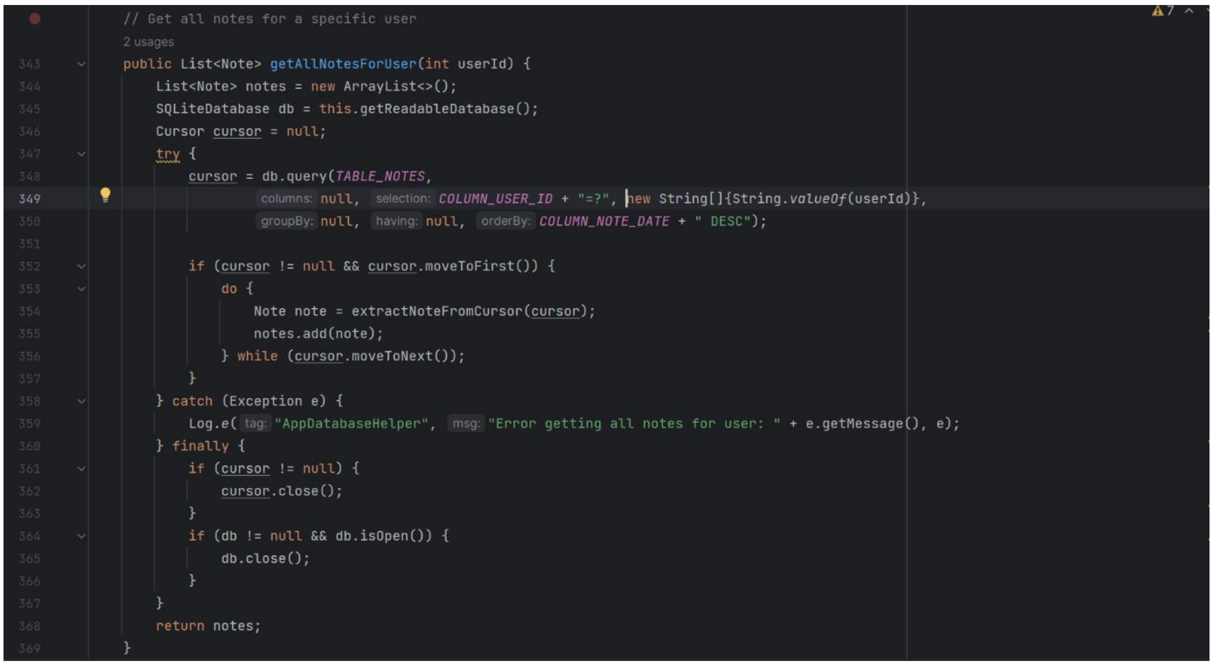
```java
// Insert a new note
1 usage
public long insertNote(Note note) {
    SQLiteDatabase db = this.getWritableDatabase();
    ContentValues values = new ContentValues();
    values.put(COLUMN_NOTE_TITLE, note.getTitle());
    values.put(COLUMN_NOTE_DETAILS, note.getContent()); // Use COLUMN_NOTE_DETAILS for content
    values.put(COLUMN_NOTE_DATE, note.getDate());
    values.put(COLUMN_NOTE_FAVORITE, note.isFavorite() ? 1 : 0);
    values.put(COLUMN_NOTE_URGENT, note.isUrgent() ? 1 : 0);
    values.put(COLUMN_USER_ID, note.getUserId()); // Link note to user

    long result = -1;
    try {
        result = db.insertOrThrow(TABLE_NOTES,  nullColumnHack: null, values);
    } catch (Exception e) {
        Log.e( tag: "AppDatabaseHelper",  msg: "Error inserting note: " + e.getMessage() + " - Values: " +
                values.toString(), e);
    } finally {
        if (db != null && db.isOpen()) {
            db.close();
        }
    }
    return result;
}
```



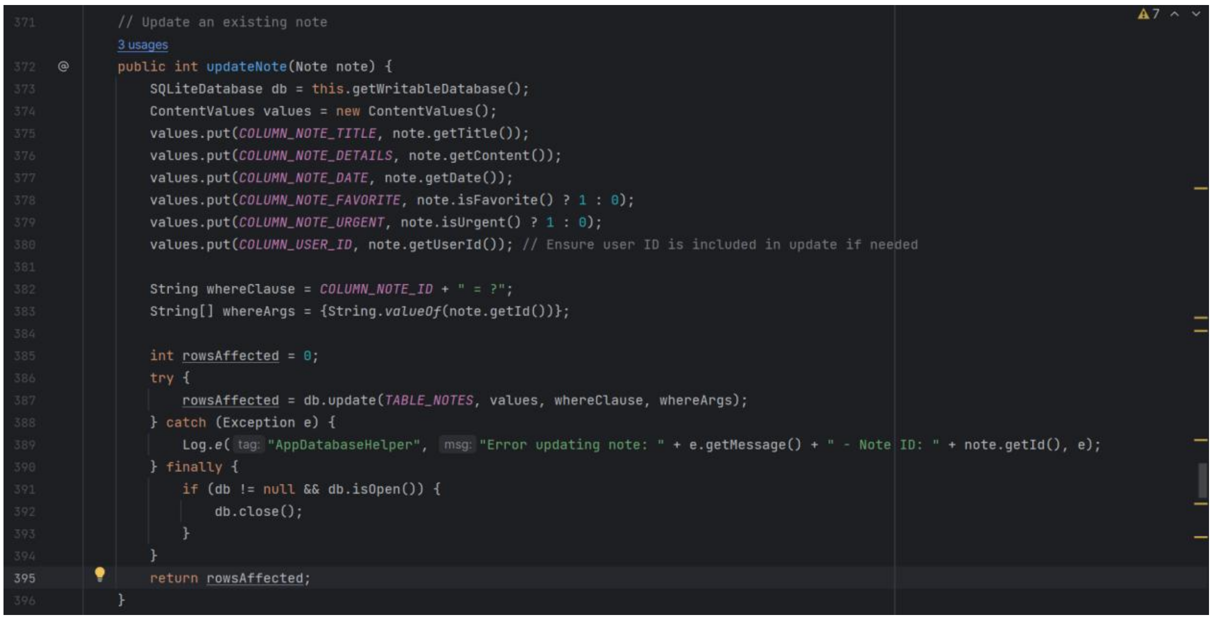
```
// Get a specific note by its ID
1 usage
public Note getNote(int id) {
    SQLiteDatabase db = this.getReadableDatabase();
    Cursor cursor = null;
    Note note = null;
    try {
        cursor = db.query(TABLE_NOTES,
                columns: null, selection: COLUMN_NOTE_ID + "=?",
                new String[]{String.valueOf(id)},
                groupBy: null, having: null, orderBy: null);
        if (cursor != null && cursor.moveToFirst()) {
            note = extractNoteFromCursor(cursor);
        }
    } catch (Exception e) {
        Log.e( tag: "AppDatabaseHelper", msg: "Error getting note by ID: " + e.getMessage(), e);
    } finally {
        if (cursor != null) {
            cursor.close();
        }
        if (db != null && db.isOpen()) {
            db.close();
        }
    }
    return note;
}
```



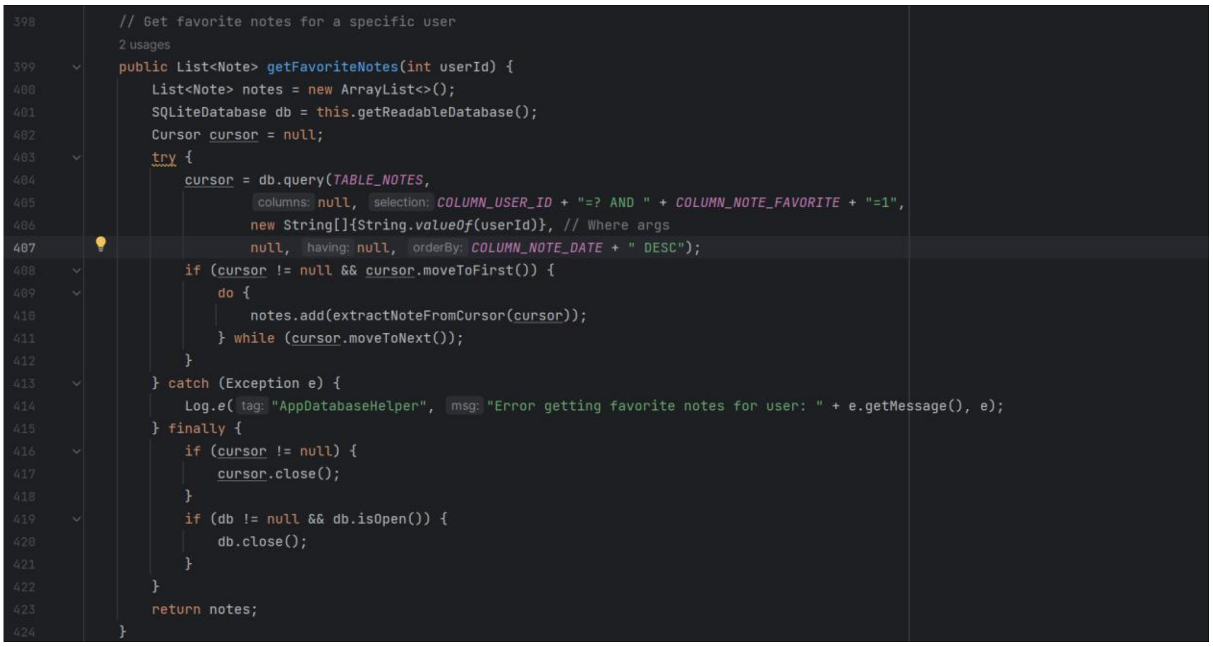
```
// Delete a note by its ID
2 usages
public int deleteNote(int id) {
    SQLiteDatabase db = this.getWritableDatabase();
    int result = 0;
    try {
        result = db.delete(TABLE_NOTES, whereClause: COLUMN_NOTE_ID + " = ?", new String[]{String.valueOf(id)});
    } catch (Exception e) {
        Log.e( tag: "AppDatabaseHelper", msg: "Error deleting note: " + e.getMessage(), e);
    } finally {
        if (db != null && db.isOpen()) {
            db.close();
        }
    }
    return result;
}
```



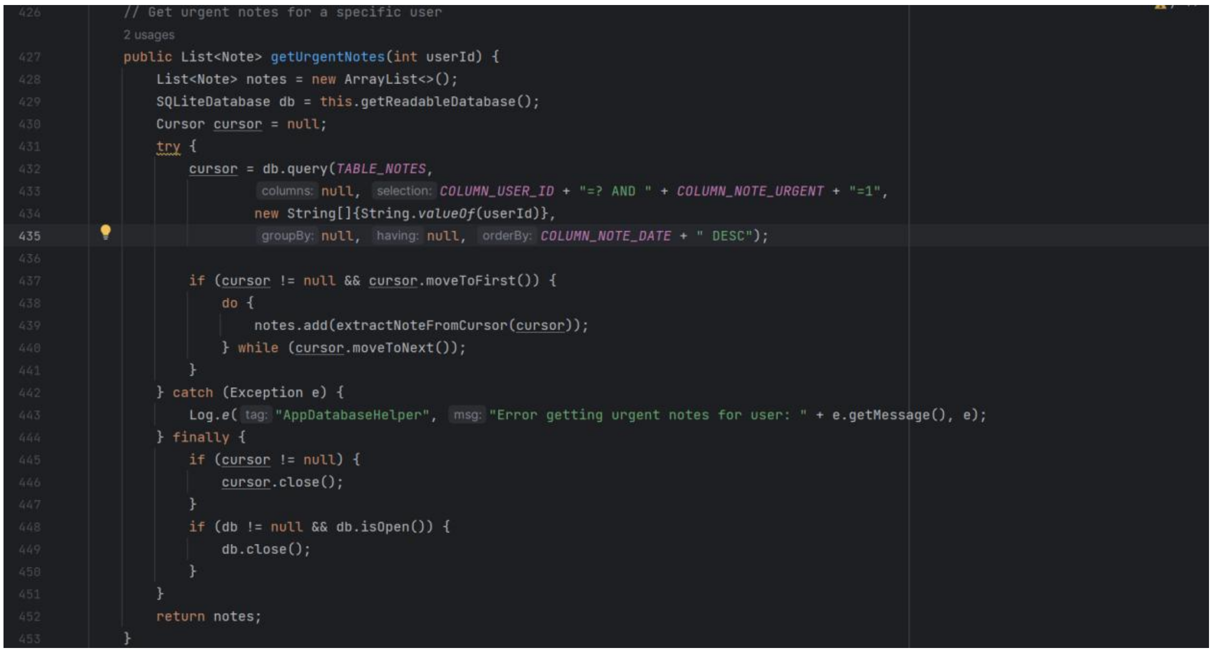
```
// Get all notes for a specific user
2 usages
public List<Note> getAllNotesForUser(int userId) {
    List<Note> notes = new ArrayList<>();
    SQLiteDatabase db = this.getReadableDatabase();
    Cursor cursor = null;
    try {
        cursor = db.query(TABLE_NOTES,
                columns: null, selection: COLUMN_USER_ID + "=?", new String[]{String.valueOf(userId)},
                groupBy: null, having: null, orderBy: COLUMN_NOTE_DATE + " DESC");

        if (cursor != null && cursor.moveToFirst()) {
            do {
                Note note = extractNoteFromCursor(cursor);
                notes.add(note);
            } while (cursor.moveToNext());
        }
    } catch (Exception e) {
        Log.e( tag: "AppDatabaseHelper", msg: "Error getting all notes for user: " + e.getMessage(), e);
    } finally {
        if (cursor != null) {
            cursor.close();
        }
        if (db != null && db.isOpen()) {
            db.close();
        }
    }
    return notes;
}
```



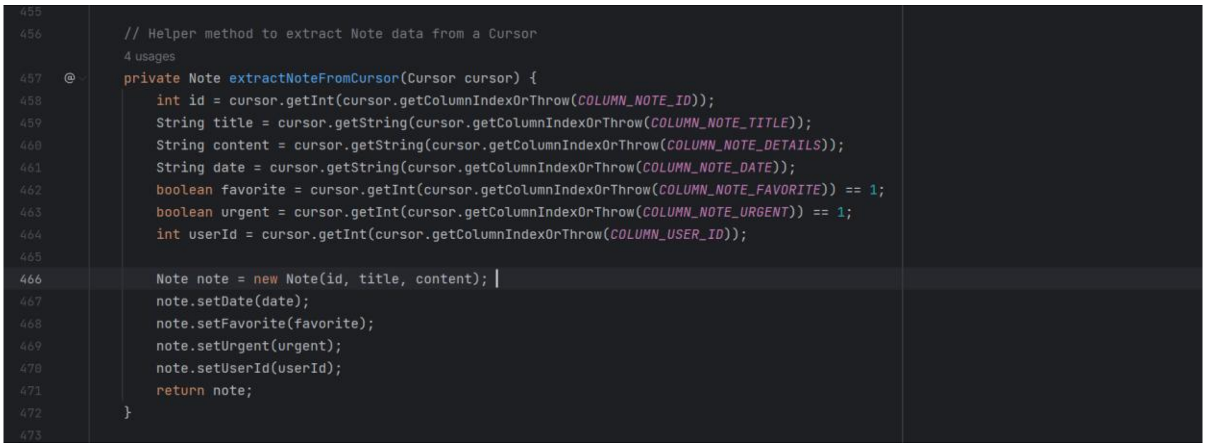
```java
// Update an existing note
3 usages
public int updateNote(Note note) {
    SQLiteDatabase db = this.getWritableDatabase();
    ContentValues values = new ContentValues();
    values.put(COLUMN_NOTE_TITLE, note.getTitle());
    values.put(COLUMN_NOTE_DETAILS, note.getContent());
    values.put(COLUMN_NOTE_DATE, note.getDate());
    values.put(COLUMN_NOTE_FAVORITE, note.isFavorite() ? 1 : 0);
    values.put(COLUMN_NOTE_URGENT, note.isUrgent() ? 1 : 0);
    values.put(COLUMN_USER_ID, note.getUserId()); // Ensure user ID is included in update if needed

    String whereClause = COLUMN_NOTE_ID + " = ?";
    String[] whereArgs = {String.valueOf(note.getId())};

    int rowsAffected = 0;
    try {
        rowsAffected = db.update(TABLE_NOTES, values, whereClause, whereArgs);
    } catch (Exception e) {
        Log.e( tag: "AppDatabaseHelper",  msg: "Error updating note: " + e.getMessage() + " - Note ID: " + note.getId(), e);
    } finally {
        if (db != null && db.isOpen()) {
            db.close();
        }
    }
    return rowsAffected;
}
```



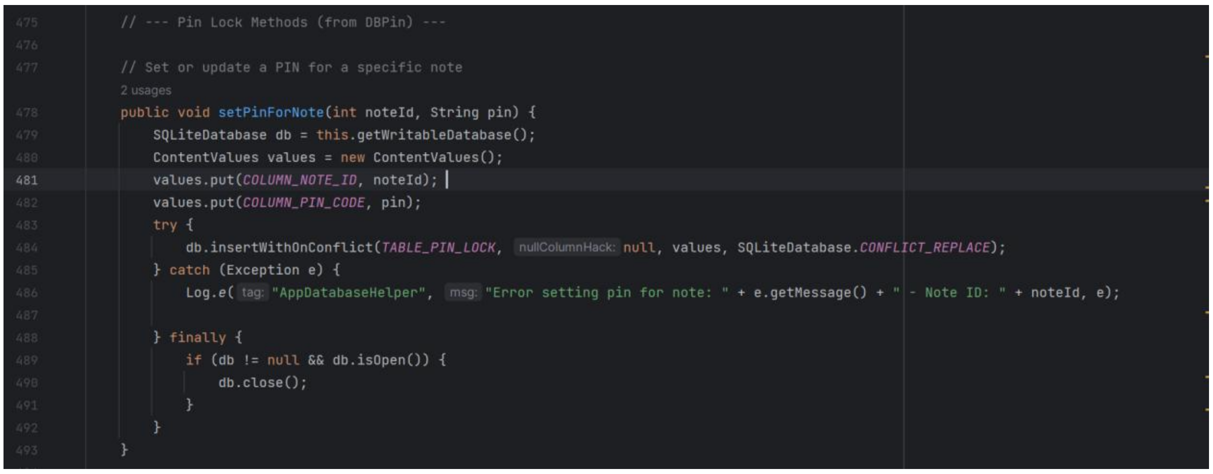
```java
// Get favorite notes for a specific user
2 usages
public List<Note> getFavoriteNotes(int userId) {
    List<Note> notes = new ArrayList<>();
    SQLiteDatabase db = this.getReadableDatabase();
    Cursor cursor = null;
    try {
        cursor = db.query(TABLE_NOTES,
                 columns: null,  selection: COLUMN_USER_ID + "=? AND " + COLUMN_NOTE_FAVORITE + "=1",
                new String[]{String.valueOf(userId)}, // Where args
                null,  having: null,  orderBy: COLUMN_NOTE_DATE + " DESC");
        if (cursor != null && cursor.moveToFirst()) {
            do {
                notes.add(extractNoteFromCursor(cursor));
            } while (cursor.moveToNext());
        }
    } catch (Exception e) {
        Log.e( tag: "AppDatabaseHelper",  msg: "Error getting favorite notes for user: " + e.getMessage(), e);
    } finally {
        if (cursor != null) {
            cursor.close();
        }
        if (db != null && db.isOpen()) {
            db.close();
        }
    }
    return notes;
}
```



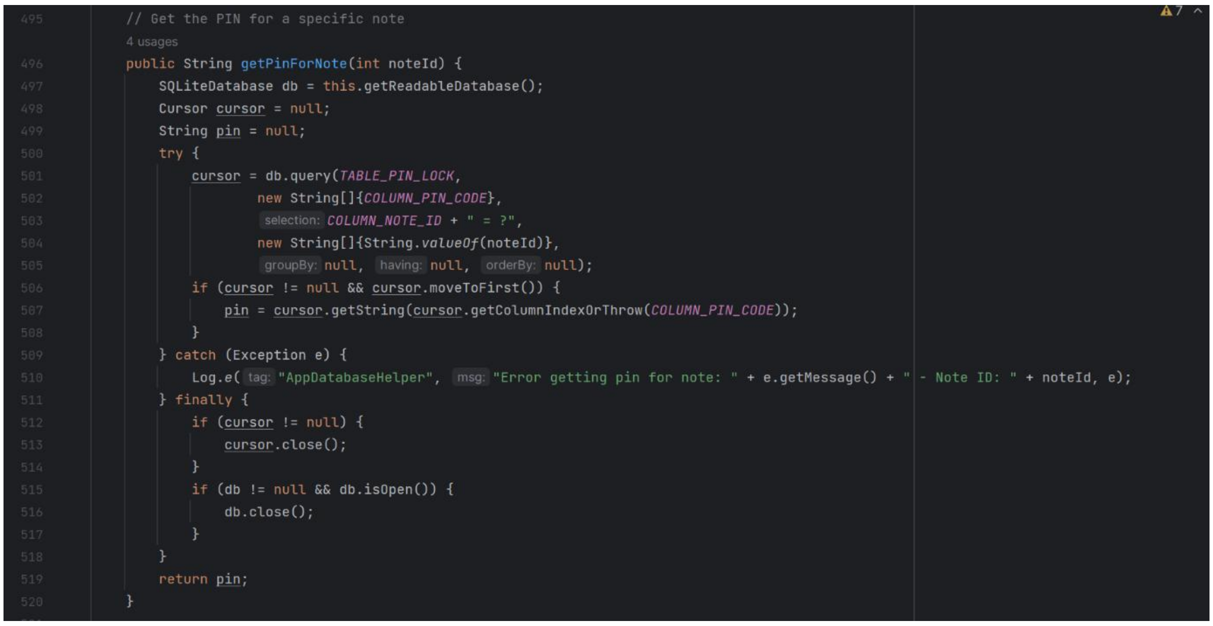
```java
    // Get urgent notes for a specific user
    2 usages
    public List<Note> getUrgentNotes(int userId) {
        List<Note> notes = new ArrayList<>();
        SQLiteDatabase db = this.getReadableDatabase();
        Cursor cursor = null;
        try {
            cursor = db.query(TABLE_NOTES,
                    columns: null, selection: COLUMN_USER_ID + "=? AND " + COLUMN_NOTE_URGENT + "=1",
                    new String[]{String.valueOf(userId)},
                    groupBy: null, having: null, orderBy: COLUMN_NOTE_DATE + " DESC");

            if (cursor != null && cursor.moveToFirst()) {
                do {
                    notes.add(extractNoteFromCursor(cursor));
                } while (cursor.moveToNext());
            }
        } catch (Exception e) {
            Log.e( tag: "AppDatabaseHelper", msg: "Error getting urgent notes for user: " + e.getMessage(), e);
        } finally {
            if (cursor != null) {
                cursor.close();
            }
            if (db != null && db.isOpen()) {
                db.close();
            }
        }
        return notes;
    }
```



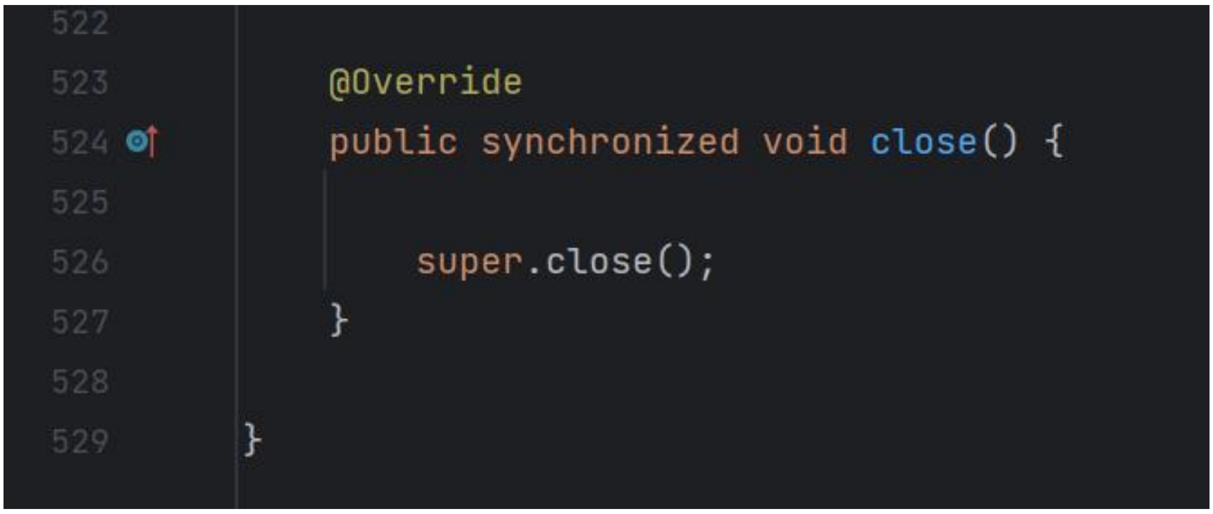
```java
    // Helper method to extract Note data from a Cursor
    4 usages
    private Note extractNoteFromCursor(Cursor cursor) {
        int id = cursor.getInt(cursor.getColumnIndexOrThrow(COLUMN_NOTE_ID));
        String title = cursor.getString(cursor.getColumnIndexOrThrow(COLUMN_NOTE_TITLE));
        String content = cursor.getString(cursor.getColumnIndexOrThrow(COLUMN_NOTE_DETAILS));
        String date = cursor.getString(cursor.getColumnIndexOrThrow(COLUMN_NOTE_DATE));
        boolean favorite = cursor.getInt(cursor.getColumnIndexOrThrow(COLUMN_NOTE_FAVORITE)) == 1;
        boolean urgent = cursor.getInt(cursor.getColumnIndexOrThrow(COLUMN_NOTE_URGENT)) == 1;
        int userId = cursor.getInt(cursor.getColumnIndexOrThrow(COLUMN_USER_ID));

        Note note = new Note(id, title, content);
        note.setDate(date);
        note.setFavorite(favorite);
        note.setUrgent(urgent);
        note.setUserId(userId);
        return note;
    }
```



```java
    // --- Pin Lock Methods (from DBPin) ---

    // Set or update a PIN for a specific note
    2 usages
    public void setPinForNote(int noteId, String pin) {
        SQLiteDatabase db = this.getWritableDatabase();
        ContentValues values = new ContentValues();
        values.put(COLUMN_NOTE_ID, noteId);
        values.put(COLUMN_PIN_CODE, pin);
        try {
            db.insertWithOnConflict(TABLE_PIN_LOCK, nullColumnHack: null, values, SQLiteDatabase.CONFLICT_REPLACE);
        } catch (Exception e) {
            Log.e( tag: "AppDatabaseHelper", msg: "Error setting pin for note: " + e.getMessage() + " - Note ID: " + noteId, e);

        } finally {
            if (db != null && db.isOpen()) {
                db.close();
            }
        }
    }
```



```
    // Get the PIN for a specific note
    4 usages
    public String getPinForNote(int noteId) {
        SQLiteDatabase db = this.getReadableDatabase();
        Cursor cursor = null;
        String pin = null;
        try {
            cursor = db.query(TABLE_PIN_LOCK,
                    new String[]{COLUMN_PIN_CODE},
                    selection: COLUMN_NOTE_ID + " = ?",
                    new String[]{String.valueOf(noteId)},
                    groupBy: null,  having: null,  orderBy: null);
            if (cursor != null && cursor.moveToFirst()) {
                pin = cursor.getString(cursor.getColumnIndexOrThrow(COLUMN_PIN_CODE));
            }
        } catch (Exception e) {
            Log.e( tag: "AppDatabaseHelper",  msg: "Error getting pin for note: " + e.getMessage() + " - Note ID: " + noteId, e);
        } finally {
            if (cursor != null) {
                cursor.close();
            }
            if (db != null && db.isOpen()) {
                db.close();
            }
        }
        return pin;
    }
```



```
    @Override
    public synchronized void close() {

        super.close();
    }

}
```

## 1.2. Table Structure

2. users Table

| Column Name | Data Type | Constraints |
|---|---|---|
| user_id | INTEGER | PRIMARY KEY AUTOINCREMENT |
| username | TEXT | UNIQUE |
| email | TEXT | UNIQUE |
| password | TEXT | NOT NULL |
| password_hint | TEXT | NULLABLE |
| profile_image_uri | TEXT | NULLABLE |

3. notes Table

| Column Name | Data Type | Constraints |
|---|---|---|
| id | INTEGER | PRIMARY KEY AUTOINCREMENT |
| title | TEXT | NULLABLE |
| details | TEXT | NULLABLE |
| date | TEXT | NULLABLE (usually formatted as a string like "yyyy-MM-dd") |
| favorite | INTEGER | DEFAULT 0 (0 = false, 1 = true) |
| urgent | INTEGER | DEFAULT 0 (0 = false, 1 = true) |
| user_id | INTEGER | FOREIGN KEY REFERENCES users(user_id) ON DELETE CASCADE |

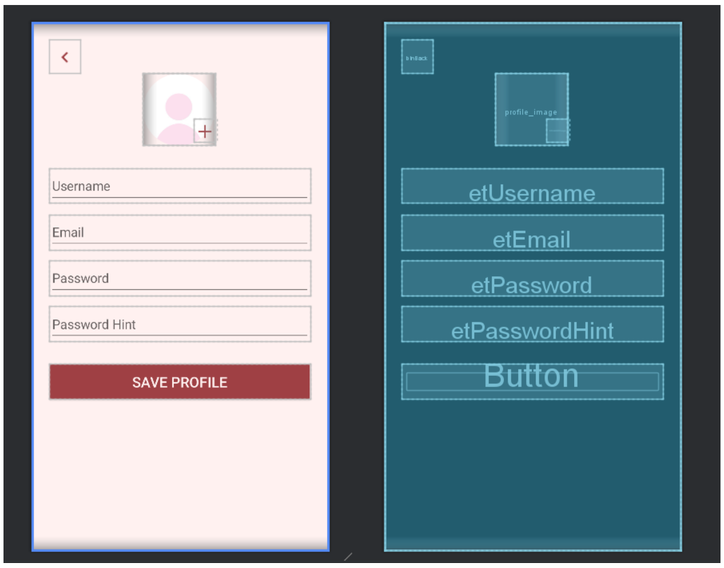
4. pin_lock Table

| Column Name | Data Type | Constraints |
|---|---|---|
| id | INTEGER | PRIMARY KEY (references notes.id) |
| pin_code | TEXT | NOT NULL |
| (Foreign Key) | | FOREIGN KEY(id) REFERENCES notes(id) ON DELETE CASCADE |

## Task 2

### 1. activity_edit_profile.xml

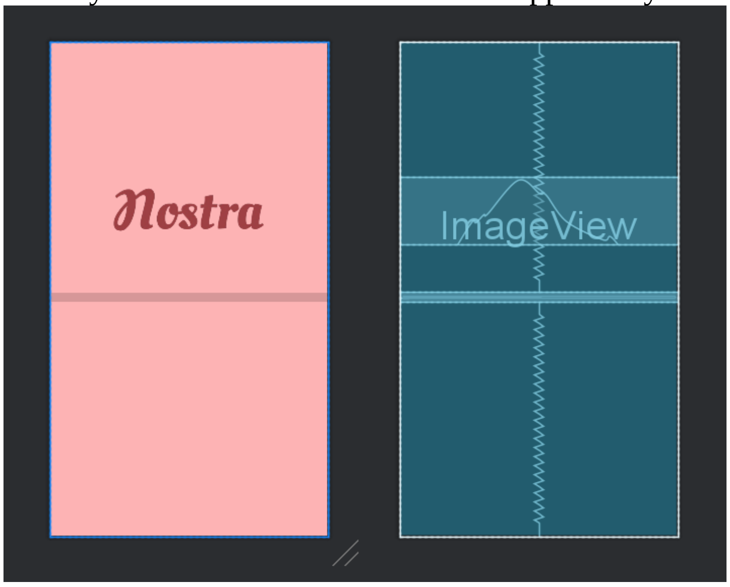
This layout uses a vertical LinearLayout to arrange profile editing elements. It includes an image button for navigation, a profile image with an add image button, and EditText fields for username, email, password, and password hint. A "Save Profile" button is provided at the bottom. The layout uses padding and spacing for a clean, user-friendly interface. The design emphasizes easy data entry and profile image updating.



activity_edit_profile.xml

### 2. activity_loading.xml

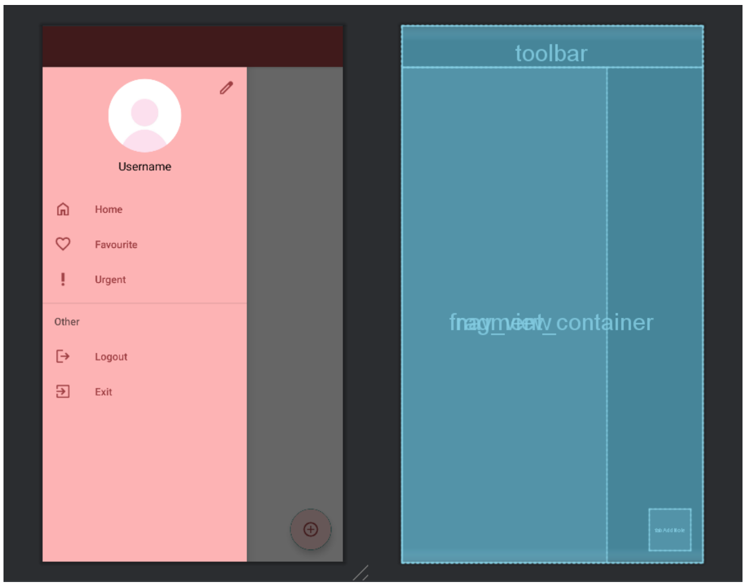
This file defines a RelativeLayout with a centered logo image and a horizontal ProgressBar. The logo is spaced from the top, and the progress bar is placed below it, scaled for visibility. The background color sets a consistent theme. This layout is used to indicate loading or initialization processes. It visually communicates to users that the app is busy.



activity_loading.xml

## 3. activity_main.xml

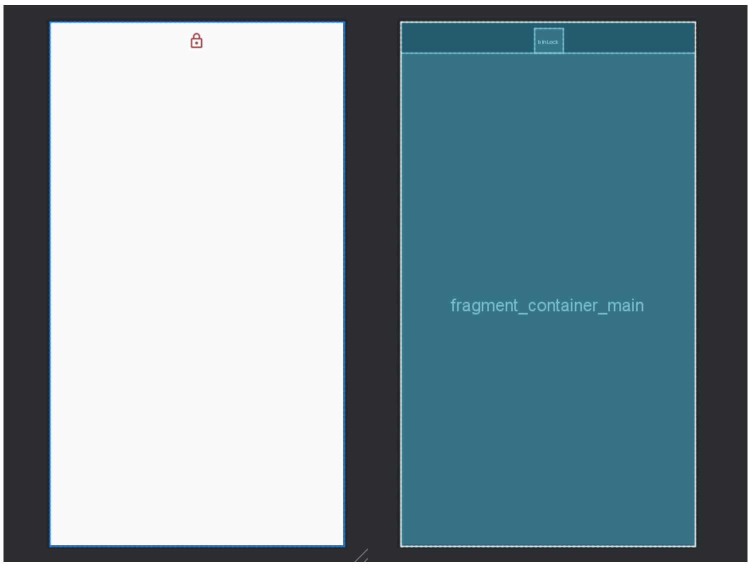
This layout is based on DrawerLayout, supporting a navigation drawer and main content area. It includes a toolbar at the top, a fragment container for dynamic content, and a floating action button for adding new notes. The navigation drawer is styled and includes a header and menu items. The structure supports both navigation and quick actions. It's designed for flexible and interactive app navigation.



activity_main.xml

## 4. activity_main2.xml

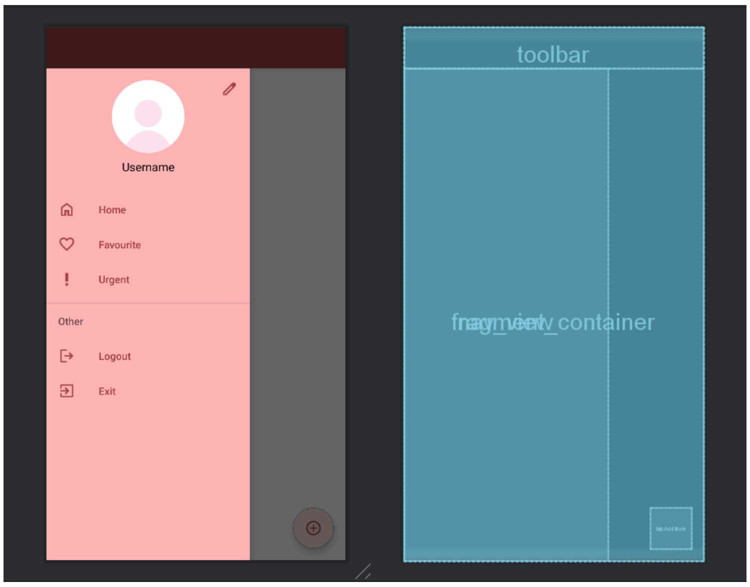
This layout uses ConstraintLayout to position a lock button at the top and a FrameLayout container below. The lock button is centered horizontally, and the fragment container fills the remaining space. The design is simple, focusing on modular content swapping. It is likely used for secure or specialized app screens. The layout allows dynamic UI updates within the container.



activity_main2.xml

## 5. activity_main3.xml

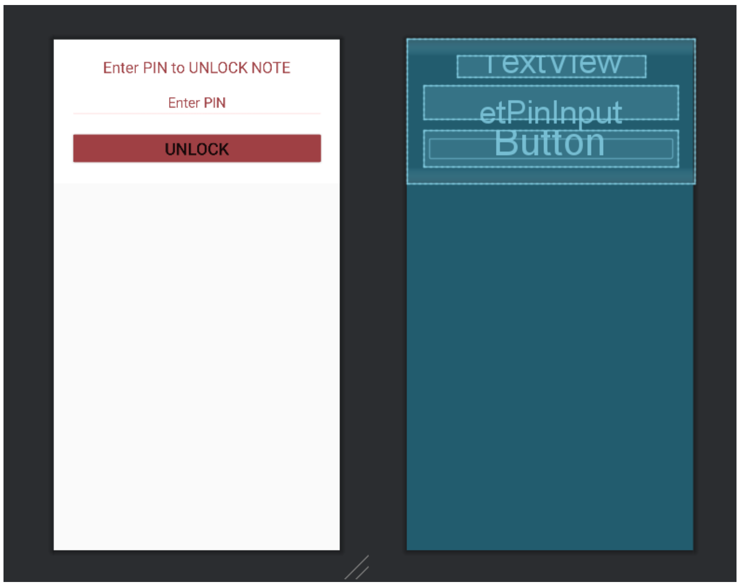
Similar to activity_main.xml, this layout also uses DrawerLayout with a navigation drawer and main content area. It features a toolbar, a fragment container, and a floating action button for note creation. The navigation drawer provides quick access to different sections. The layout supports multitasking and efficient navigation. Its structure is suitable for apps with multiple main features.



activity_main3.xml

## 6. dialog_pin_input.xml

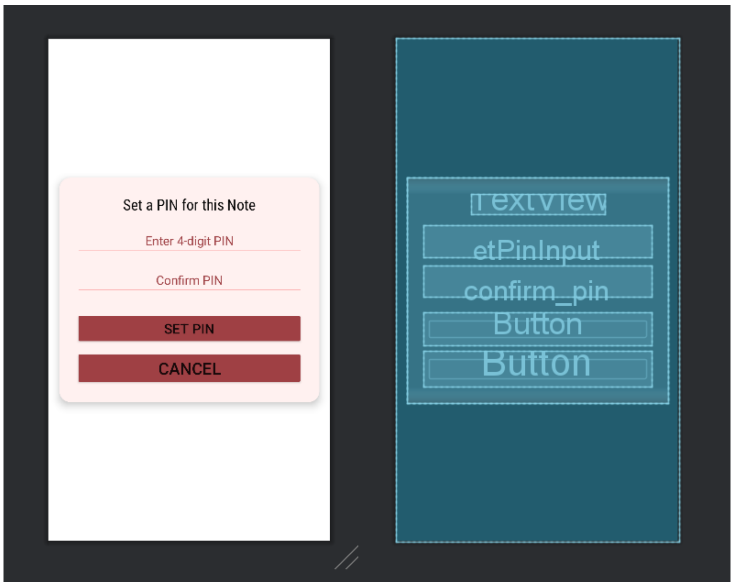
This layout creates a vertical dialog for entering a PIN. It includes a prompt, an EditText for PIN input (limited to 4 digits), and an unlock button. The design uses color cues and centered text for clarity. The layout is compact and focused on secure PIN entry. It is used for unlocking protected notes.



dialog_pin_input.xml

## 7. dialog_set_pin.xml

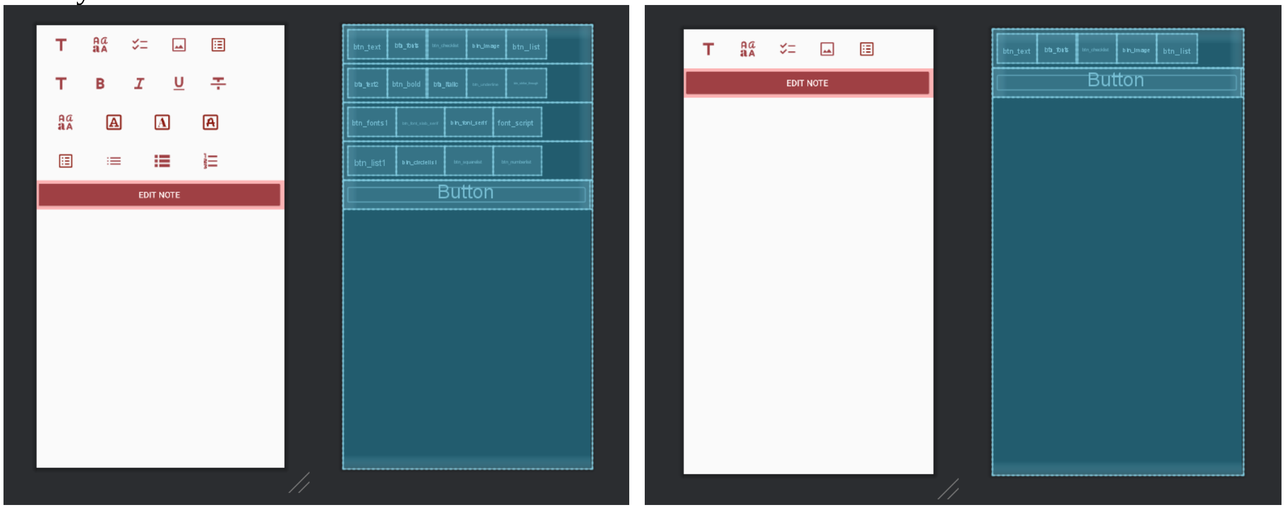
This file defines a CardView-based dialog for setting a new PIN. Inside, a vertical LinearLayout holds a prompt, two EditTexts (for PIN and confirmation), and a submit button. The card has rounded corners and elevation for a modern look. The layout guides users through PIN creation. It emphasizes clarity and security in the PIN setup process.



**dialog_set_pin.xml**

## 8. floatingbar_main.xml

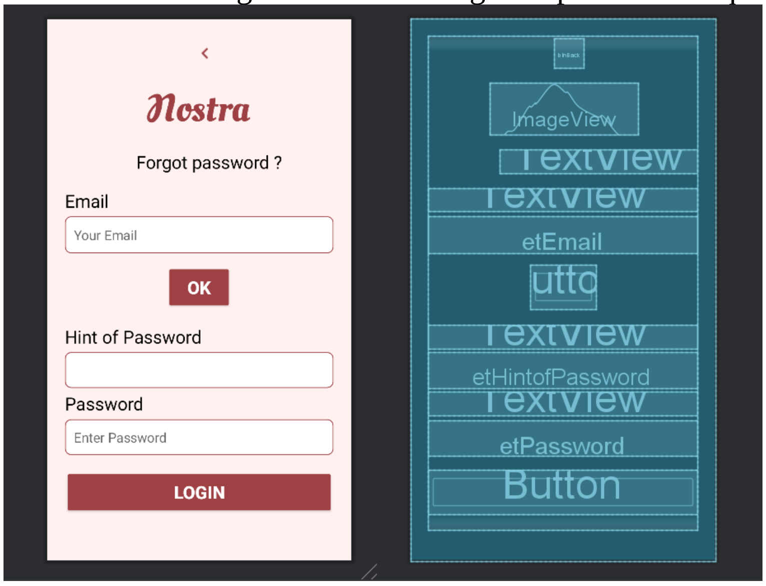
This file defines a horizontal LinearLayout that serves as a floating toolbar. It contains several ImageButtons for actions like undo, redo, speech-to-text, and background color selection. Each button uses an icon and is spaced for easy access. The layout is intended to float above other UI components, providing quick access to note editing tools. Its compact design keeps essential actions readily available.



floatingbar_main.xml

## 9. forgot_password.xml

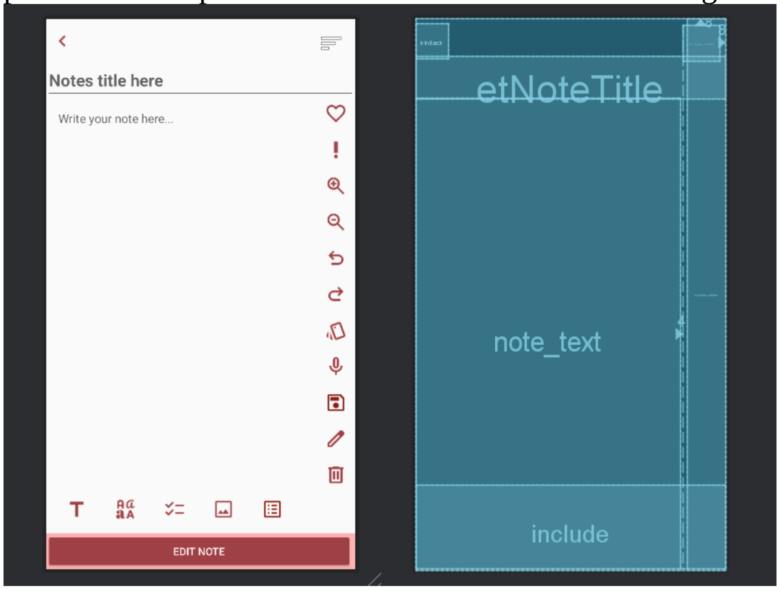
This layout provides a vertical LinearLayout for the password recovery screen. It includes an EditText for entering the user's email and a button to submit the request. There's also a TextView for instructions and feedback. The design is straightforward, focusing on helping users recover their account access. It guides users through the process of requesting a password hint or reset.



forgot_password.xml

## 10. fragment_note_detail.xml

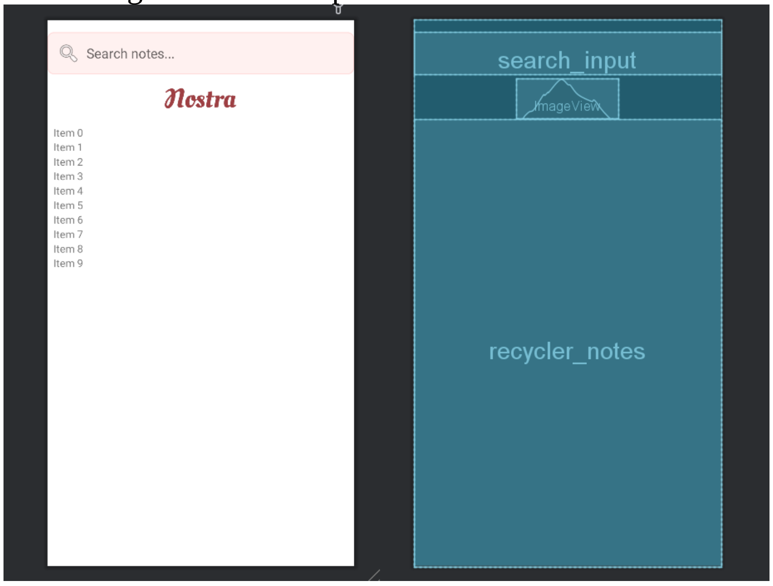
This file uses a ScrollView containing a vertical LinearLayout to display the details of a note. It includes EditTexts for the note title and content, buttons for saving and deleting, and toggle buttons for marking as favorite or urgent. There are also options for formatting, speech input, and background color changes. The layout supports both viewing and editing a single note in detail. It provides a comprehensive interface for rich note management.



fragment_note_detail.xml

### 11. fragment_notes.xml

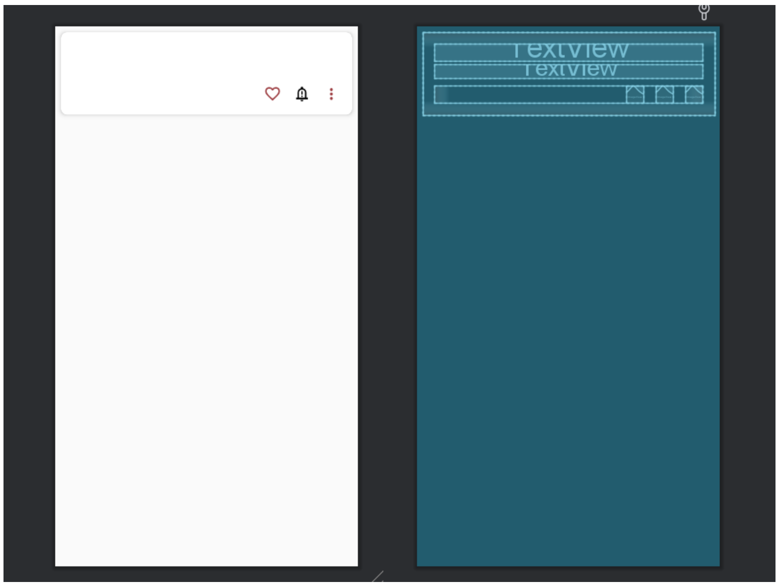
This layout features a vertical LinearLayout with a search bar at the top, filter buttons (all, favorite, urgent), and a RecyclerView for displaying notes. There's also a TextView for empty state messages when no notes are present. The design enables searching, filtering, and browsing notes efficiently. It is optimized for dynamic content updates and user interaction. The structure supports easy navigation and management of multiple notes.



fragment_notes.xml

### 12. item_note.xml

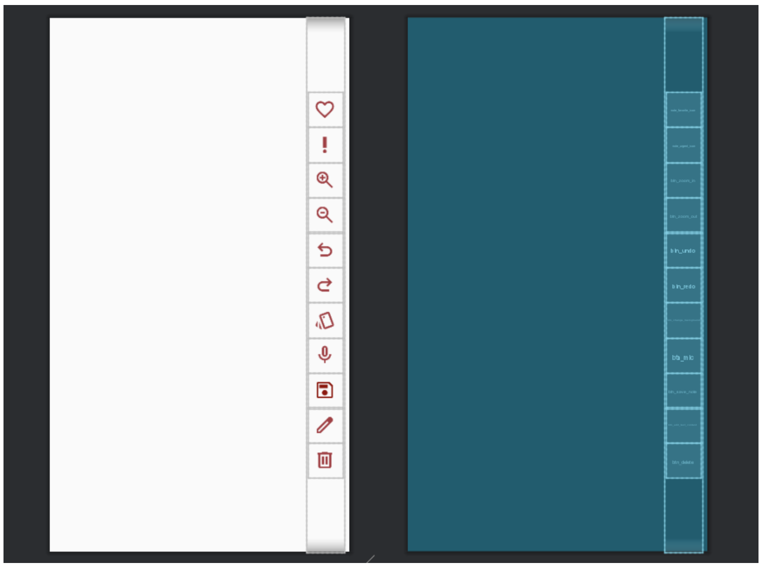
This file defines the layout for a single note item in a list, using a CardView. Inside, there are TextViews for the note's title, content preview, and date, along with ImageViews or icons to indicate favorite and urgent status. The card is styled with padding and elevation for visual separation. The layout is compact and optimized for RecyclerView lists. It allows users to quickly scan and interact with individual notes.



item_note.xml

### 13. layout_right_sidebar.xml

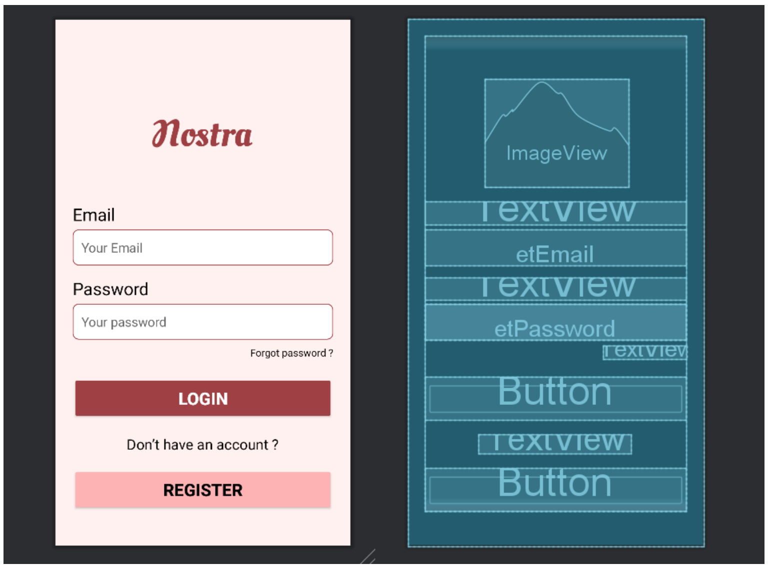
This layout uses a vertical LinearLayout to create a sidebar, typically for additional navigation or options. It includes buttons or icons for actions like editing, deleting, or changing note settings. The sidebar is styled with background color and spacing to distinguish it from the main content. It is designed to slide in from the right and provide contextual actions. The structure enhances usability by grouping related tools together.



layout_right_sidebar.xml

### 14. login.xml

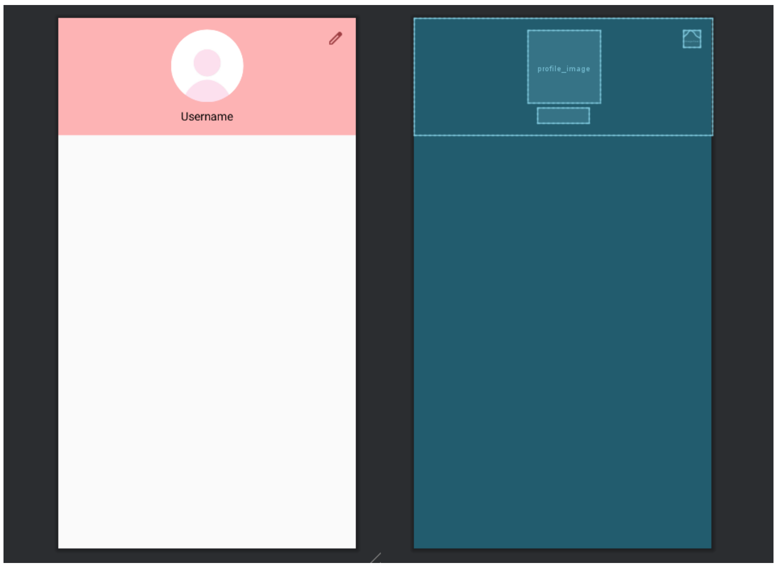
This file defines the login screen using a vertical LinearLayout. It includes EditText fields for email and password, a login button, and TextViews for navigation to signup or password recovery. The layout is padded for comfort and uses hints for clarity. The design is simple and user-friendly, focusing on quick and secure authentication. It ensures users can easily access their accounts.



login.xml

### 15. nav_header.xml

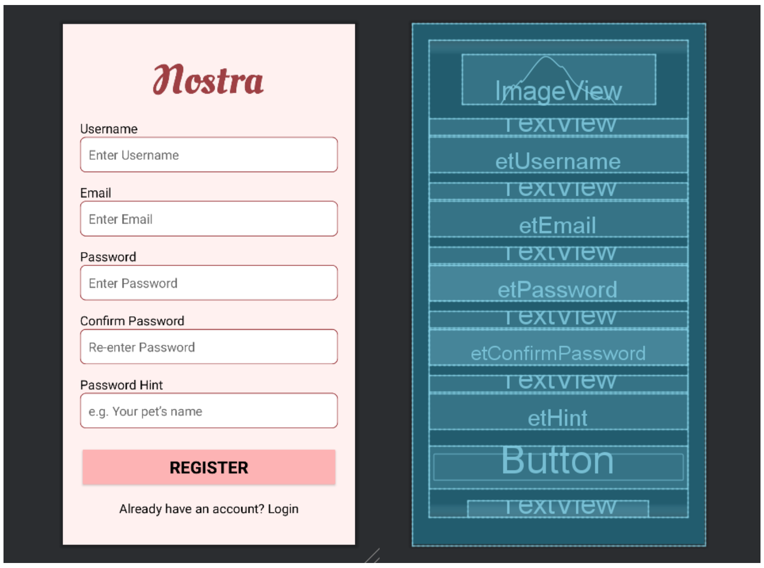
This layout creates a header for the navigation drawer using a vertical LinearLayout. It features an ImageView for the user's profile picture, and TextViews for displaying the username and email. The background color matches the app's theme for visual consistency. The header personalizes the navigation drawer and provides quick account identification. Its centered elements create a welcoming user experience.



nav_header.xml

### 16. signup.xml

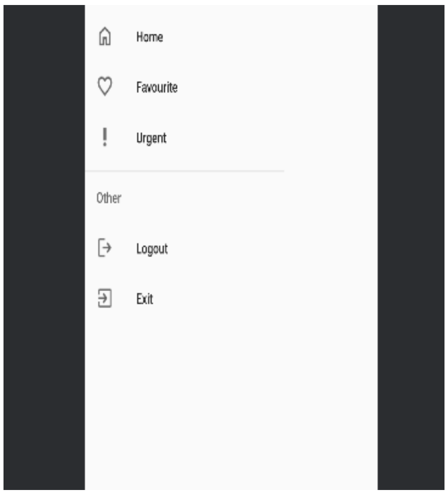
This layout uses a vertical LinearLayout for the signup screen. It contains EditText fields for username, email, password, confirm password, and password hint, along with a signup button. There's also a TextView to navigate to the login screen. The design is clean and organized, guiding users through the registration process. It ensures all necessary information is collected for account creation.



signup.xml

**17. nav_menu.xml**

The XML file outlines the app's navigation drawer menu structure, featuring three main items (Home, Favourite, and Urgent) with icons and titles, and an "Other" section for Logout and Exit options, ensuring easy access to core app sections and account actions.



## Task 3

### 1. AndroidManifest.xml

In order to configure the Android app, this file is necessary. It contains the package name, permissions, and all registered activities. It establishes the appearance and behavior of the program, makes sure it has the required access (such as internet, storage, and audio recording), and chooses which activity starts first. For seamless startup and operation, the manifest also incorporates the application with the Android operating system.

**Major functions:**

- Declares permissions for audio, storage, and internet.
- Sets app icon, label, theme, and backup rules.
- Registers all activities and sets the launcher.
- Configures intent filters for app launching.

### 2. EditProfileActivity.java

The EditProfileActivity.java is an activity that allows users to view and edit their profile details, including username, email, password, password hint, and profile image. It retrieves user data from the database, displays it, and allows users to select a new profile image. The activity validates inputs, checks email uniqueness, and updates the database, ensuring data integrity and proper resource release.

**Major functions:**

- loadProfileData(String email): Loads user data from the database and populates the UI fields.
- openImageChooser(): Opens the system file picker to select a profile image, requesting persistent URI permissions.
- onActivityResult(...): Handles the result of image selection and manages URI permissions.
- saveProfile(): Validates inputs, checks for email uniqueness, and updates the user record.
- onDestroy(): Closes the database helper to avoid resource leaks.

### 3. ForgotPasswordActivity.java

This activity helps users in recovering their password by retrieving a password hint or allowing password reset. Users provide their email, and the app fetches the corresponding hint from the database. If found, the hint is displayed, otherwise, an error message appears. The activity also offers a login option, navigation controls, and database helper closure when the activity is destroyed.

**Major functions:**

- onCreate(Bundle savedInstanceState): Initializes UI components and sets up listeners for hint retrieval, login, and navigation.
- btnOk.setOnClickListener: Fetches and displays the password hint for the entered email.
- btnLogin.setOnClickListener: Validates credentials and attempts login.
- btnBack.setOnClickListener: Navigates back to the login activity.
- onDestroy(): Ensures the database helper is closed.

### 4. LoadingActivity.java

This activity acts as a splash screen for the application, with a ProgressBar that updates incrementally. It uses a Handler and postDelayed to manage the animation and transition. Once the progress bar reaches 100%, it redirects the user to the LoginActivity for a smooth startup experience.

**Major functions:**

- onCreate(Bundle savedInstanceState): Sets up the splash screen and schedules a delayed transition to the main activity.
- Use of Handler.postDelayed: Implements the timed splash effect.
- The progress bar: Visually indicates loading progress.
- No explicit navigation is shown, but typically the next activity would be started after the timeout.
- No resource management is necessary as no database or external resources are used.

### 5. LoginActivity.java

The btnLogin click listener is the primary gateway for user authentication in the application. It validates email and password input, stores login status, and navigates users to MainActivity. The btnRegister button directs users to SignupActivity and ForgotPasswordActivity. The activity provides login flow options, registration, and password recovery options, and ensures proper resource management by closing the database helper when destroyed.

**Major functions:**

- onCreate(Bundle savedInstanceState): Sets up UI and listeners for login, registration, and password recovery.
- btnLogin.setOnClickListener: Handles login logic and credential validation.
- btnRegister.setOnClickListener: Navigates to the registration screen.
- btnForgotPassword.setOnClickListener: Navigates to the password recovery screen.
- onDestroy(): Closes the database helper.

### 6. MainActivity.java

The main activity after a user logs in serves as the central navigation hub, initializing the toolbar, setting up the navigation drawer, and attaching listeners for menu items. The loadProfileHeader method populates the drawer's header with the logged-in user's profile image and username. The logoutUser method clears session data and redirects the user to the LoginActivity for secure logout.

**Major functions:**

- onCreate(Bundle savedInstanceState): Initializes the main UI and checks login status.
- Navigation logic: Redirects to login or other activities as needed.
- Resource management: Ensures any helpers or listeners are properly released.
- UI setup: Prepares the main interface for user interaction.
- Handles app exit gracefully.

### 7. MainActivity2.java

This activity manages detailed note viewing, creation, editing, and locking, using EXTRA_ACTION to determine if a new note is being added or edited. It features showUnlockDialog for PIN access and showSetPinDialog for setting PINs. It serves as a robust container for NoteDetailFragment, integrating security measures for user notes and managing navigation to detail or edit screens.

**Major functions:**

- onCreate(Bundle savedInstanceState): Sets up the UI and loads data from the database.
- Data retrieval: Fetches notes or categories for display.
- Item interaction: Handles clicks for viewing, editing, or deleting items.
- Navigation: Moves to detail or edit activities as needed.
- onDestroy(): Closes database connections.

### 8. MainActivity3.java

This activity is a splash screen with a fixed-duration delay, launching the LoginActivity after a SPLASH_DELAY of 2000 milliseconds (2 seconds) delay. It provides a brief visual introduction to the application, loading relevant data based on user selection, allowing editing, and saving changes to the database. Navigation controls allow returning to previous screens, and resource cleanup is handled during the destruction phase.

**Major functions:**

- onCreate(Bundle savedInstanceState): Loads specific data for display or editing.
- Data binding: Populates UI with note or setting details.
- Save logic: Updates the database with any changes.
- Navigation: Returns to previous activities.
- onDestroy(): Releases resources.

### 9. NoteAdapter.java

This class handles click events, manages view holders, and connects data to the RecyclerView or ListView UI component. It warns the adapter of changes and guarantees effective view recycling. It inflates layouts and binds data to views while displaying a list of notes in a RecyclerView. A NoteClickListener interface is also managed by it.

**Major functions:**

- onCreateViewHolder(ViewGroup, int): Inflates item layouts and creates view holders.
- onBindViewHolder(ViewHolder, int): Binds note data to the item views.
- getItemCount(): Returns the number of notes.
- Click listeners: Handles user actions on each note item.
- Data update methods: Refresh the UI when the data set changes.

### 10. AppDatabaseHelper.java

This class controls all database functions, including note CRUD operations, profile management, and user authentication. It offers ways to add, update, query, and remove records in addition to defining the database schema. The helper makes sure that database access is safe and effective across the entire app. Version control and database generation are handled by it. In order to manage resources, database connections are opened and closed as necessary.

**Major functions:**

- User authentication: Checks credentials and retrieves user information.
- Note CRUD: Methods for creating, reading, updating, and deleting notes.
- Profile management: Updates and retrieves user profile data.
- Database schema: Defines tables and columns for users and notes.
- Resource management: Opens and closes database connections.

### 11. SignupActivity.java

This activity handles user registration, allowing new users to create an account by entering a username, email, password, and password hint. It validates all input fields, checks for existing emails, and ensures password requirements are met before creating a new user in the database. On successful registration, the user is redirected to the login screen.

**Major functions:**

- Validates registration input fields.
- Checks for unique email addresses.
- Creates new user accounts in the database.

### 12. User.java

This model class represents a user in the application. It stores user-related information such as ID, username, email, password, and associated note IDs. The class provides getter and setter methods for each property, enabling easy access and modification of user data throughout the app.

**Major functions:**

- Stores and manages user data.
- Provides getter and setter methods for user properties.

### 13. Note.java

This model class defines the structure of a note in the app. It includes fields for note ID, title, content, date, favorite and urgent status, pin code, and user ID. The class offers constructors for creating new notes or loading existing ones and provides getter and setter methods for all properties.

**Major functions:**

- Represents note data and properties.
- Supports creation and modification of notes.

### 14. NotesFragment.java

This fragment displays a list of notes for the current user, supporting features like searching, filtering by category (all, favorite, urgent), and real-time updates. It allows users to edit, delete, mark notes as favorite or urgent, and search notes by title. The fragment interacts with the database and updates the UI using a RecyclerView and a custom adapter.

**Major functions:**

- Loads and displays user notes by category.
- Supports searching and filtering notes.
- Handles note editing, deletion, and status updates.

**15. PinActivity.java**

This activity manages PIN protection for individual notes. It provides dialogs for users to set a 4-digit PIN on a note or unlock a note by entering the correct PIN. The activity interacts with the database to securely store and retrieve PINs, and provides user feedback through dialogs and toast messages. Input validation ensures PINs are exactly four digits and match on confirmation.

**Major functions:**

- Shows dialogs to set or unlock a note PIN.
- Stores and verifies PINs using the database.
- Validates PIN input and handles user feedback.

**16. NoteDetailFragment.java**

This fragment handles the detailed view and editing of notes. It allows users to create, view, edit, and delete notes, with features like text formatting, undo/redo, speech-to-text input, background customization, and toggling favorite or urgent status. The fragment manages UI elements such as toolbars and sidebars, and interacts with the database for all note operations, ensuring a rich and interactive note-taking experience.

**Major functions:**

- Loads, displays, saves, and deletes notes from the database.
- Supports undo/redo, speech input, and background changes.
- Allows marking notes as favorite or urgent and toggling edit mode.

## Task 4

1. **Create (Insert) Note:**

- **Test Name :** Creating new note
- **Test Data :** Create a note with a title, content, and mark it as "Favorite."
- **Test Procedure :** Use the app's add note functionality
- **Expected Result:** New note should be added and that note have to be appears in the notes list
- **Actual Result:** Figure : 4.1.1, 4.1.2, 4.1.3

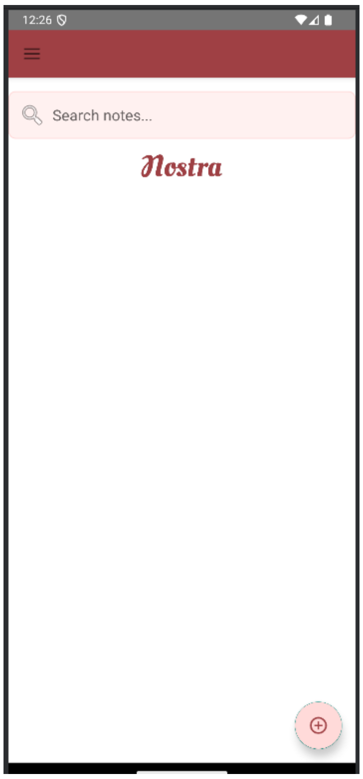


Figure : 4.1.1

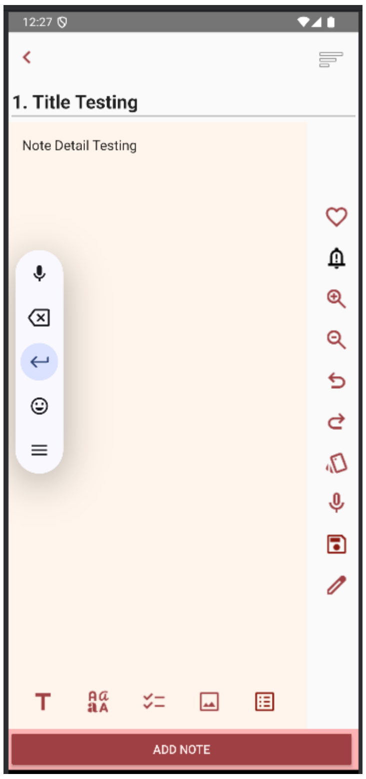


Figure : 4.1.2

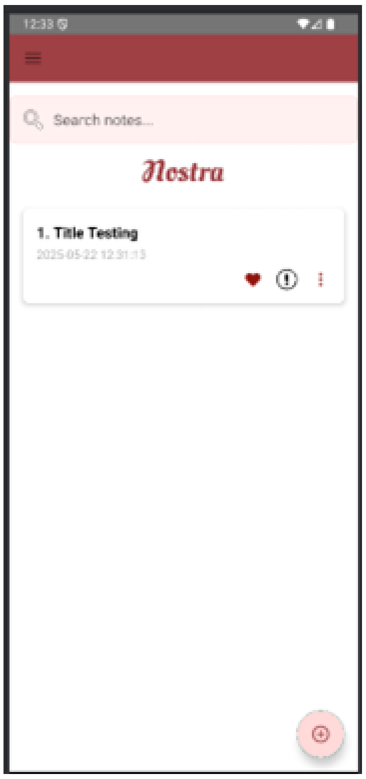


Figure : 4.1.3

2. **Read (View) Notes:**

- **Test Name:** Displaying all saved notes.
- **Test Data:** Existing notes in the database.
- **Test Procedure:** Open the home screen.
- **Expected Result:** All notes are listed.
- **Actual Result:** Figure 4.2.1

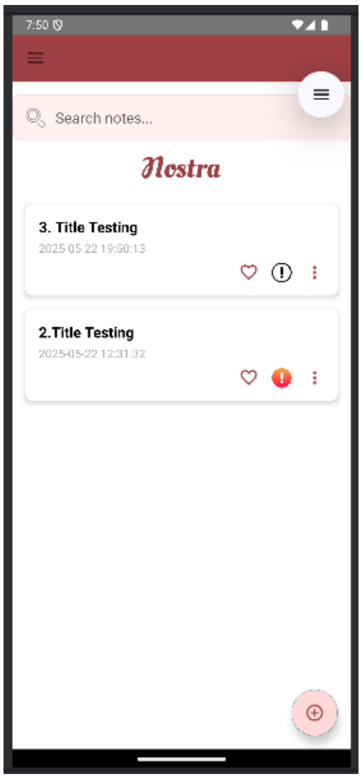


Figure : 4.2.1

3. **Update (Edit) Note:**

- **Test Name:** Test editing an existing note's title or content.
- **Test Data:** Existing note with updated title and content.
- **Test Procedure:** Use the edit feature on a selected note, modify its title and content, and save.
- **Expected Result:** Note updates with new information and reflects changes in the list.
- **Actual Result:** Figure 4.3.1, 4.3.2

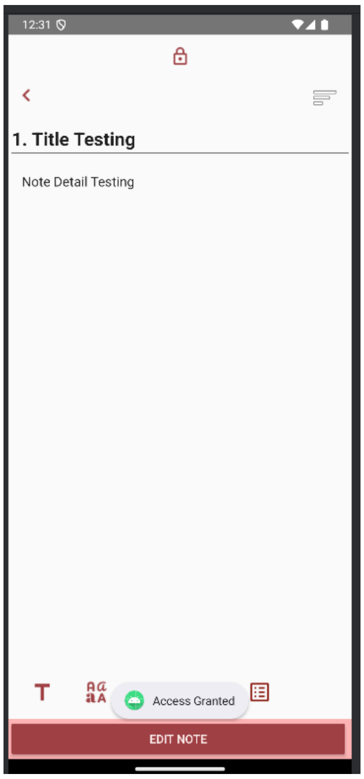


Figure : 4.3.1

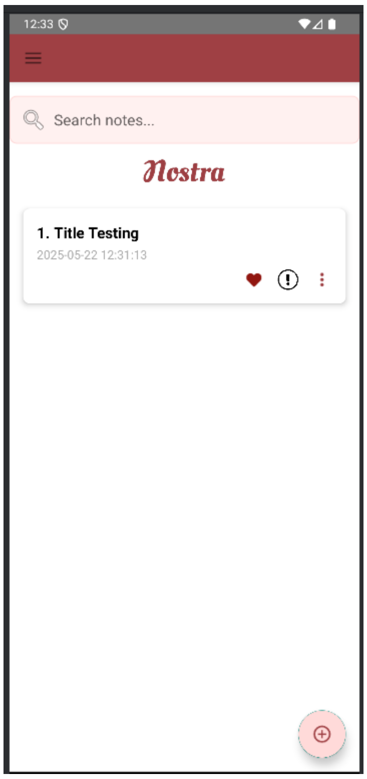


Figure : 4.3.2

4. **Delete Note:**

- **Test Name:** Test deleting a note from the list.
- **Test Data:** Existing note.
- **Test Procedure:** Use the delete feature on a selected note and confirm deletion.
- **Expected Result:** Note is successfully removed from the list.
- **Actual Result:** Figure 4.4.1, 4.4.2

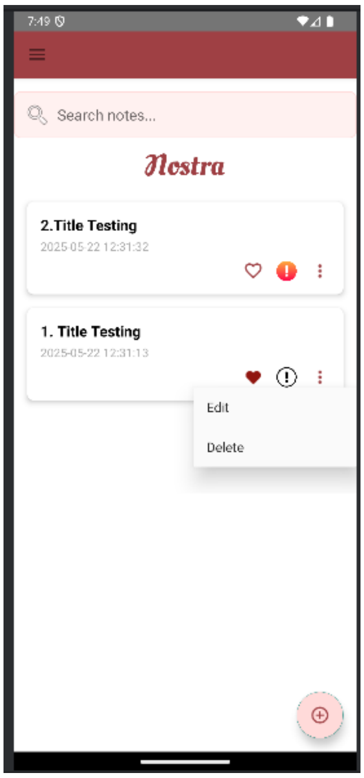


Figure : 4.4.1

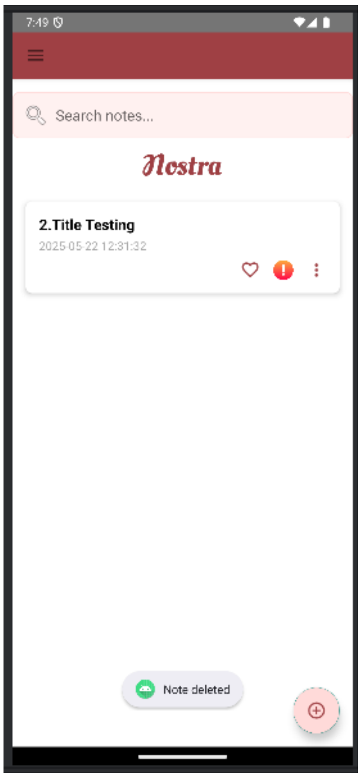


Figure : 4.4.2

5. **Search Note:**

- **Test Name:** Test searching for a note by title.
- **Test Data:** Notes with distinct titles, and a specific search keyword.
- **Test Procedure:** Use the search function with a keyword from an existing note's title.
- **Expected Result:** Only matching notes are shown in the search results.
- **Actual Result:** Figure 4.5.1, 4.5.2

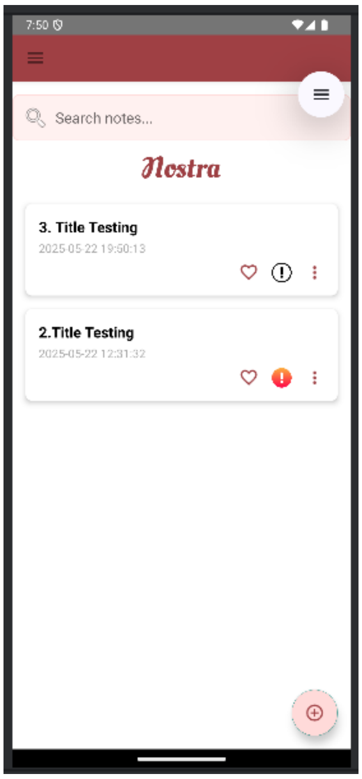


Figure : 4.5.1

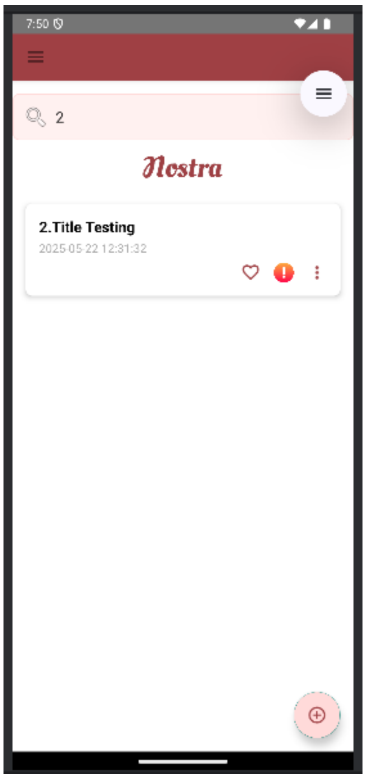


Figure : 4.5.2

6. **Profile Editing Functionality:**

- **Test Name:** Updating user profile (username, email, password hint, and image).
- **Test Data:** Existing user profile, new username, updated email, a new password hint, a selected profile image, and the current password for authentication.
- **Test Procedure:** Log in, navigate to edit profile, input new data, select an image, enter current password, save changes, and re-login to verify.
- **Expected Result:** Profile details (username, email, password hint, image) are updated and correctly displayed after saving and re-accessing the profile.
- **Actual Result:** Figure 4.6.1, 4.6.2, 4.6.3

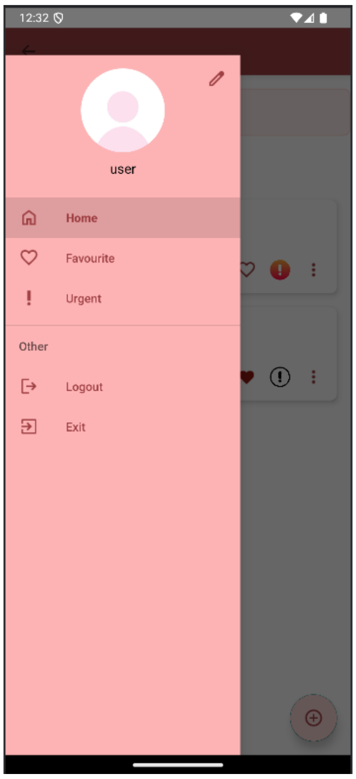


Figure : 4.6.1

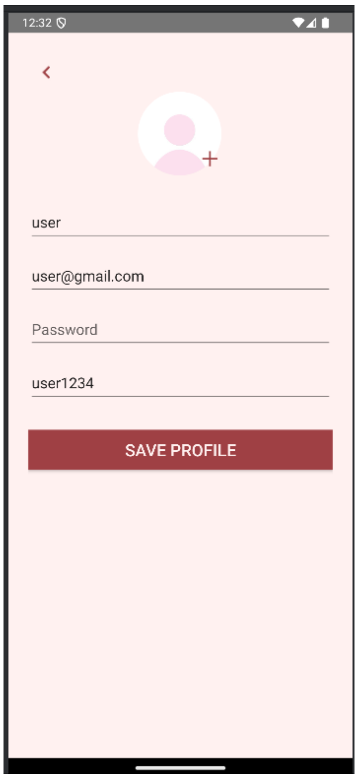
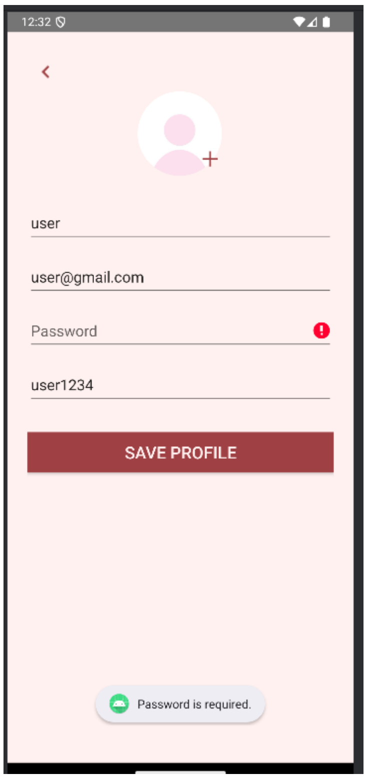


Figure : 4.6.2



Password is required.

Figure : 4.6.3

7. **Pin Activity:**

- **Test Name:** Setting and using a PIN to lock/unlock notes.
- **Test Data:** An existing note and a 4-digit PIN (e.g., "1234") for setting and confirming.
- **Test Procedure:** Select a note, choose to lock it, enter a 4-digit PIN, confirm the PIN, then attempt to unlock the note using the correct PIN.
- **Expected Result:** The note is successfully locked and requires the correct 4-digit PIN to unlock; invalid PINs are rejected.
- **Actual Result:** Figure 4.7.1, 4.7.2, 4.7.3, 4.7.4, 4.7.5, 4.7.6

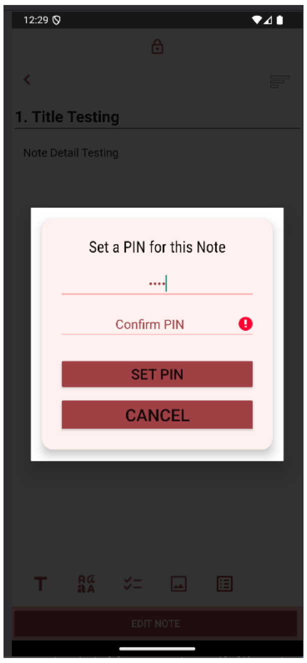


Figure : 4.7.1

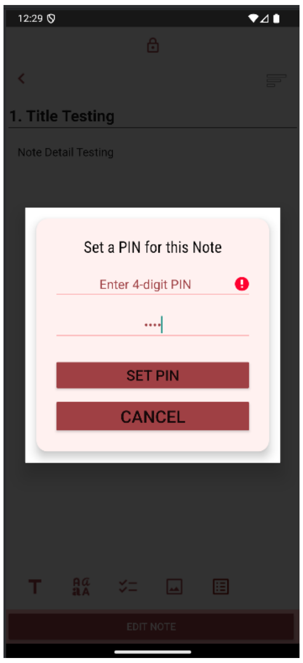
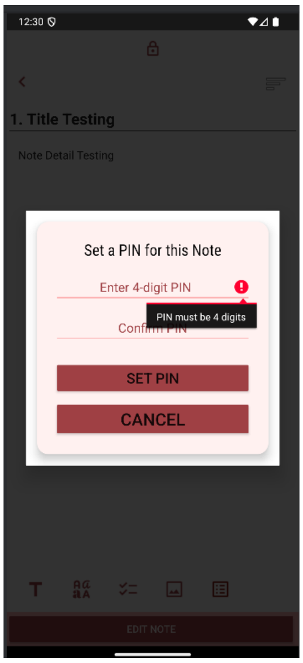
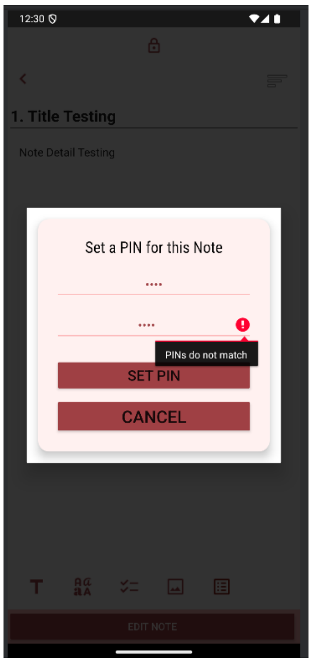


Figure : 4.7.2

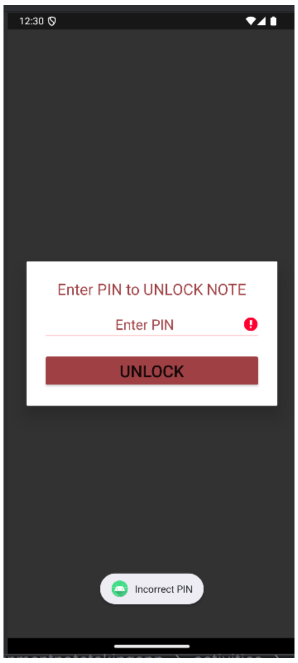


PIN must be 4 digits

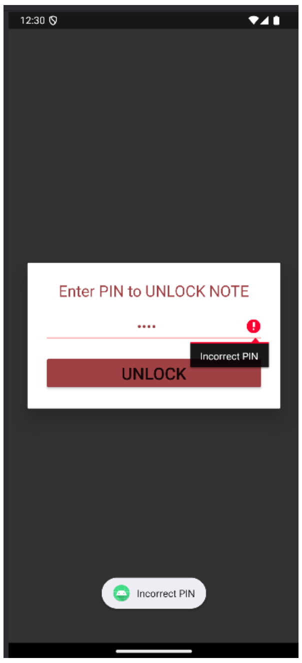
Figure : 4.7.3



PINs do not match



Incorrect PIN



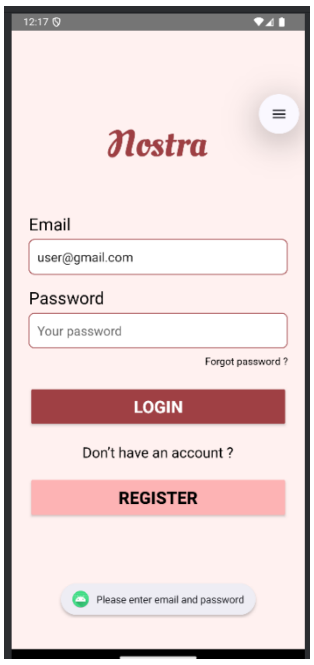
Incorrect PIN

Incorrect PIN

Figure : 4.7.4 Figure : 4.7.5 Figure : 4.7.6

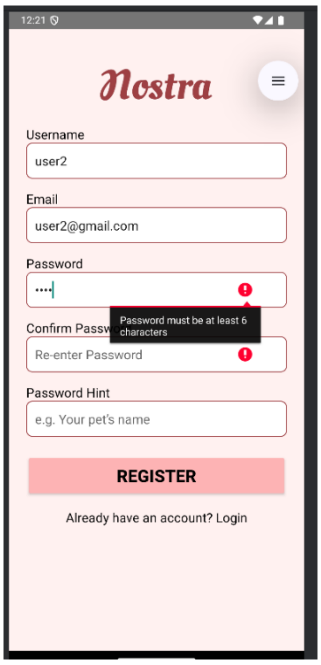
8. **Input Validation and Error Handling:**

- **Test Name:** Testing invalid/empty input during login and registration.
- **Test Data:** Empty fields for username/email/password during registration, invalid email format (e.g., "test@.com"), and incorrect login credentials.
- **Test Procedure:** Attempt to register with empty/invalid data; attempt to log in with incorrect credentials.
- **Expected Result:** Appropriate error messages are displayed for invalid input, and the system prevents saving of invalid data or successful login.
- **Actual Result:** Figure 4.8.1, 4.8.2, 4.8.3, 4.8.4, 4.8.5



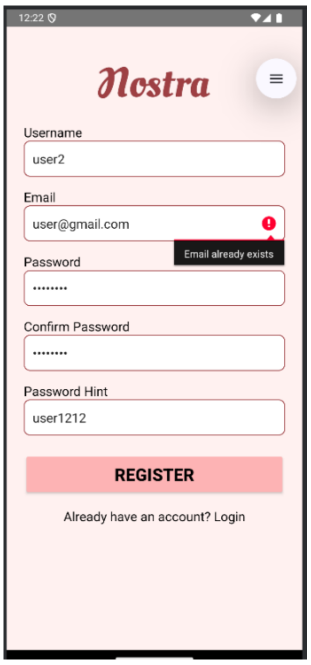
Email: user@gmail.com
Please enter email and password

Figure : 4.8.1



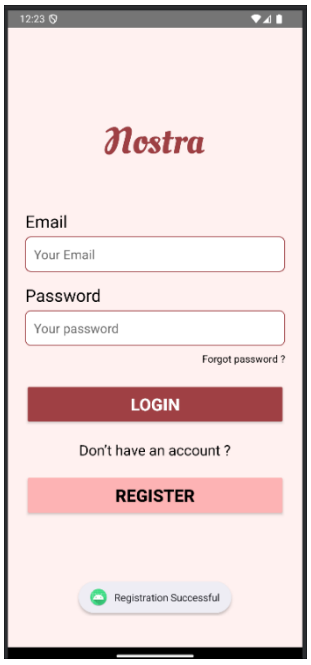
Username: user2
Email: user2@gmail.com
Password: ••••
Password must be at least 6 characters

Figure : 4.8.2



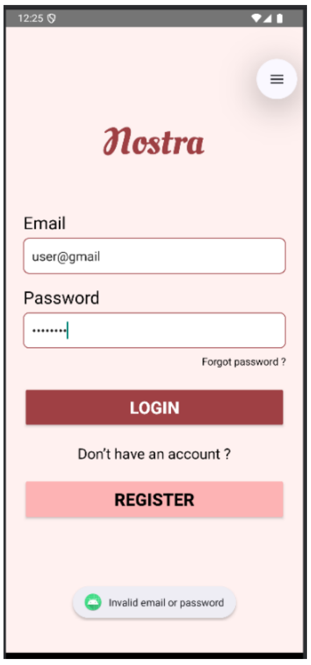
Username: user2
Email: user@gmail.com
Email already exists
Password: ••••••••
Confirm Password: ••••••••
Password Hint: user1212

Figure : 4.8.3



Registration Successful

Figure : 4.8.4



Email: user@gmail
Password: ••••••••
Invalid email or password

Figure : 4.8.5

9. **Category Test (Urgent & Favorite):**

- **Test Name:** Categorizing notes as "Urgent" and "Favorite" and filtering by category.
- **Test Data:** Multiple existing notes, with some marked as "Urgent" and others as "Favorite."
- **Test Procedure:** Mark a note as "Urgent," mark another as "Favorite," then navigate to the "Urgent" category view and the "Favorite" category view.
- **Expected Result:** Notes are correctly categorized; viewing "Urgent" or "Favorite" categories only displays notes marked with that specific status.
- **Actual Result:** Figure 4.9.1, 4.9.2, 4.9.3, 4.9.4

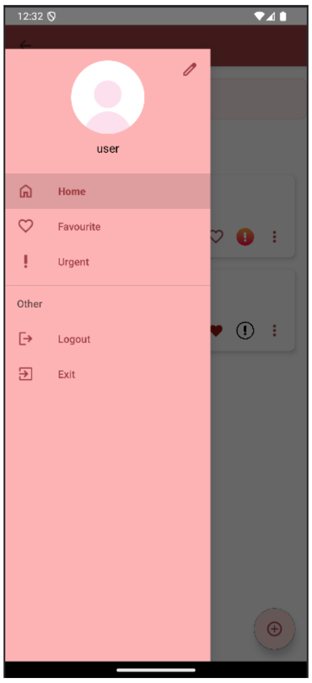


Figure : 4.9.1

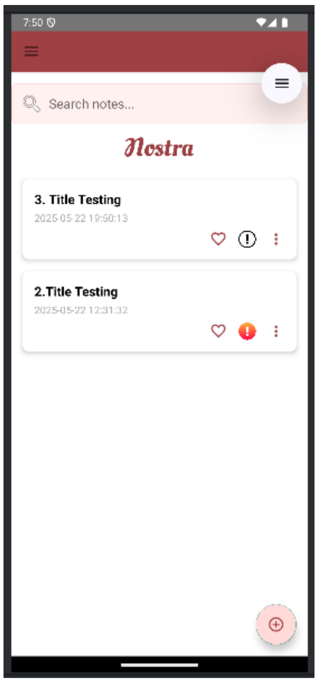


Figure : 4.9.2

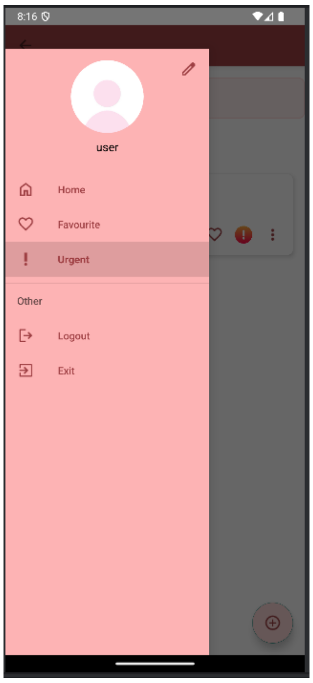


Figure : 4.9.3

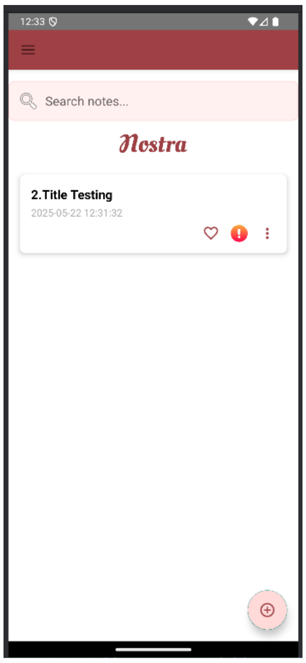


Figure : 4.9.4

10. **Forgot Password:**

- **Test Name:** Using the forgot password functionality with email.
- **Test Data:** Registered email with an associated password hint, and an unregistered email.
- **Test Procedure:** From the login screen, select "Forgot Password," enter a registered email, then try with an unregistered email.
- **Expected Result:** For a registered email, the password hint is displayed; for an unregistered email, an "Email not found" message appears.
- **Actual Result:** Figure 4.10.1, 4.10.2

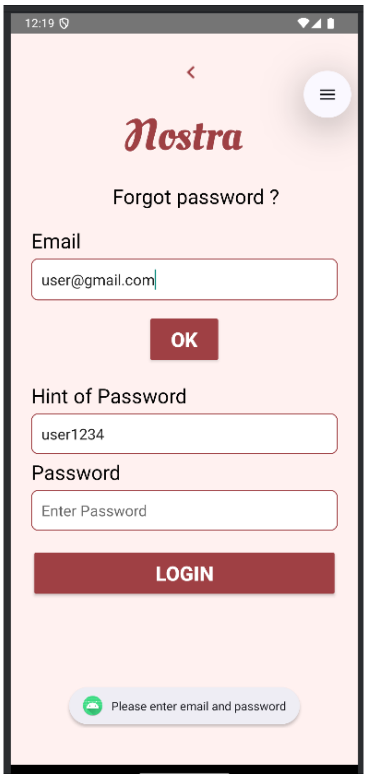


Figure : 4.10.1

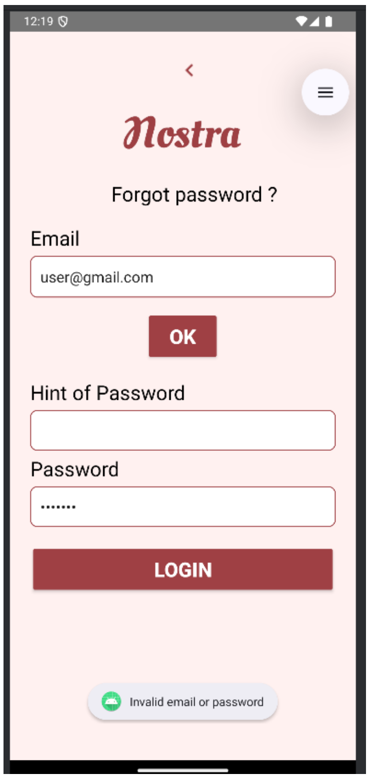


Figure : 4.10.2

## Task 5

This report provides a comprehensive evaluation of the Note-Taking App, focusing on its core features, strengths, weaknesses, proposed improvements, and deployment readiness.

**Features:**
The Note-Taking App stores all of the data locally in a SQLite database and lets users add, update, delete, and search notes. A main screen with all of the notes, detailed note views, and profile editing tools are all part of the user experience. User management and organization are improved with optional code features like note classification and user authentication. Voice-to-text input, per-note PIN locking for increased protection, and comprehensive note editing (text formatting, checklists) are additional features. To provide a customized experience, profile editing allows you to customize your username, email, password hint, and image.

**Strengths:**

- **UI (User Interface):** The app features is clean, and the design is intuitive design which contains Material Design components, making navigation straightforward. Actions like adding, editing, and deleting notes are easily accessible, and the use of RecyclerView/ListView ensures efficient note display. Rich editing tools and customizable backgrounds further enhance usability.
- **Security:** Local SQLite storage reduces exposure to network-based attacks. User authentication, password hints, and per-note PIN locking provide layers of security and privacy for sensitive data.
- **Maintenance:** The codebase is modular, with clear separation between database operations, UI activities, and profile management. This structure, along with descriptive method names and constants, simplifies future updates and maintenance.
- **Live Deployment:** The app is self-contained and works offline, making it suitable for deployment in environments with limited internet connectivity. Its reliance on local storage means no external server is required.

**Weaknesses:**

- **UI:** While functional, the UI could benefit from a more modern design, improved visual hierarchy, and enhanced accessibility features such as screen reader support and adjustable font sizes.
- **Security:** Data is stored unencrypted in SQLite, posing a risk if the device is compromised. Advanced authentication (e.g., two-factor authentication) and stronger password hashing are not implemented.
- **Maintenance:** The absence of comprehensive documentation, automated testing, and advanced error logging may hinder future maintenance. Some features, like image loading, may be sensitive to permission changes.
- **Live Deployment:** The lack of cloud backup or synchronization limits scalability and data recovery. Users cannot access notes across devices or restore data if the device is lost.

**Improvements:**
The app can be improved by including accessibility features, improving layout consistency, and adhering to contemporary UI requirements (Material Design 3). For cross-device access, use cloud synchronization, enhanced password hashing, and encryption for important data. Improve maintenance and user support by integrating analytics, automated testing, and strong account recovery.

**Live Deployment (Why/Why Not?):**
The app is ready for basic, local-only deployment due to its stability and offline functionality. However, for broader public deployment, enhancements in security, cloud synchronization, and operational monitoring are essential to meet modern user expectations and ensure data safety.

## In conclusion

In conclusion, the Note-Taking App's development provided important insights into the entire mobile application development process, from design concepts to final assessment. Within a user-friendly and flexible framework, the application effectively implements key features including user authentication, search capabilities, CRUD operations, and profile management. While it emphasized both the strengths and places for improvement, particularly in UI design, security, and scalability, extensive testing has guaranteed the dependability of key features. The app works well for local, personal use, but for wider distribution, more features like cloud synchronization and more robust security are advised. In addition to meeting the assignment requirements, this project establishes a strong basis for further advancement and practical implementation.