Password Manager — Project Documentation

Project: Password Manager (Android)

Author / Candidate: Pradeep Kumar

Date: 2025-12-11

---

1. Project Overview

This is a simple, secure, locally-stored Password Manager Android app built using Jetpack Compose. The app allows users to add, view, edit and delete stored credentials (account name, username/email, password). Passwords are encrypted before storage using AES encryption and decrypted when displayed to the user. The project follows a clean MVVM architecture (View → ViewModel → Repository → Database).

The app uses a local SQLite database (`password_manager.db`) via a `PasswordDatabaseHelper` and a small repository layer to keep data access isolated from UI logic. The app is built with Material3 UI components and Compose bottom sheets and dialogs for input and interactions.

---

2. High-level Functionalities

- **Add Password** — Add account name, username/email and password.
- **View List** — Home screen lists all saved passwords (masked) in a scrollable LazyColumn.
- **View Details** — Tap a list item to open a bottom sheet showing full details (with show/hide password toggle).
- **Edit Password** — Edit existing password/username/account via an edit dialog from details screen.
- **Delete Password** — Delete an entry with a confirmation alert dialog.
- **Encryption** — Passwords are encrypted before storing in SQLite and decrypted for display in memory.
- **Exception handling** — All DB and crypto operations are wrapped with try/catch; operations return booleans or safe fallbacks on error.

---

3. Technologies & Libraries

- **Language:** Kotlin
- **UI:** Jetpack Compose (Material3)
- **Architecture:** MVVM (ViewModel + Repository)
- **Local DB:** SQLite (via `SQLiteOpenHelper` — `PasswordDatabaseHelper`)
- **Crypto:** AES (AESCipher utility using PBKDF2 + AES/CBC/PKCS5Padding with random IV)
- **Build system:** Gradle

- **Minimum SDK:** Android 6.0+ recommended for modern crypto APIs (but PBKDF2 & Cipher approach works across versions)

  Note: No external remote storage or cloud provider; everything stays local to the device.

---

4. Project Structure (Packages & Major Files)

```
com.example.passwordmanager
├─ database
│  └─ PasswordDatabaseHelper.kt     # SQLite helper (create/update/delete/read)
+ encryption integration
├─ repository
│  └─ PasswordRepository.kt         # Wraps DB calls, exposes simple API
├─ security
│  └─ AESCipher.kt                  # Encrypt/decrypt utilities (PBKDF2 -> AES/
CBC/PKCS5Padding)
├─ ui
│  └─ screens
│     ├─ HomeScreen.kt
│     ├─ AccountRow.kt
│     ├─ AddNewAccountSheet.kt
│     ├─ AccountDetailsSheet.kt
│     ├─ EditPasswordDialog.kt
│     └─ AppTextFieldWithValue.kt
├─ ui.model
│  └─ PasswordItem.kt               # Data model (id: Int? , accountName,
userName, password)
├─ viewmodel
│  ├─ PasswordViewModel.kt
│  └─ PasswordViewModelFactory.kt
└─ MainActivity.kt
```

---

5. Database Schema

Database file: `password_manager.db`

Table: `passwords`

Columns:

- `id` INTEGER PRIMARY KEY AUTOINCREMENT
- `accountName` TEXT NOT NULL

- `username` TEXT NOT NULL
- `password` TEXT NOT NULL -- **encrypted** text stored here

Example `CREATE TABLE` used:

```
CREATE TABLE passwords (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  accountName TEXT NOT NULL,
  username TEXT NOT NULL,
  password TEXT NOT NULL
);
```

---

6. Data Model

```
data class PasswordItem(
    val id: Int? = null,
    val accountName: String,
    val userName: String,
    val password: String       // decrypted in memory, encrypted in DB
)
```

Notes:

- `id` is nullable because newly created items haven't been assigned a DB id yet.
- In-memory `password` is always the decrypted value shown to the user when required.

---

7. MVVM Flow (detailed)

1. **View (Compose)** — `HomeScreen`, BottomSheets and Dialogs collect user input and call ViewModel functions.
2. **ViewModel** — `PasswordViewModel` receives calls from UI, performs minimal validation, and delegates to the `PasswordRepository`.
3. **Repository** — `PasswordRepository` coordinates data operations and calls the `PasswordDatabaseHelper`.
4. **Database Helper** — `PasswordDatabaseHelper` executes SQL, performs encryption on write and decryption on read. All DB calls are wrapped in `try/catch` and return booleans or empty lists on failure.

Example call to save:

```
UI -> viewModel.addNewAccount(account, user, pass)
viewModel -> repository.addAccount(PasswordItem(...))
repository -> db.insertPassword(item) // encrypt inside DB helper
```

Example flow for update/delete uses `id` as the deterministic key:

```
viewModel.updateAccount(oldItem, newItem) -> repository.updateAccountById(id,
newItem) -> db.updatePasswordById(id, newValues)
```

---

8. Encryption Details

Implementation: `AESCipher.kt`

Key derivation & cipher:

- `PBKDF2WithHmacSHA256` (PBKDF2) is used to derive a 256-bit AES key from a secret & salt (both configurable in code). This provides key stretching.
- AES with `AES/CBC/PKCS5Padding` is used with a **random 16-byte IV** each encryption call. The IV is prepended to ciphertext and stored (Base64) in the DB so decryption can reconstruct the cipher parameters.

Storage format in DB:

```
Base64(IV + CipherBytes)
```

Exception handling:

- Encryption and decryption are wrapped in try/catch. On failure, the code falls back to storing plaintext (only as a last resort) or returns a clear failure indicator. All exceptions are logged (or printStackTrace in this implementation) but the app will not crash.

Important security note:

- For deterministic matching and safe update/delete, the app uses `id` as the primary key. Re-encrypting plaintext will produce a different ciphertext because IV is random; therefore matching rows by ciphertext is unreliable.

---

9. UI Screens & Components (Descriptions)

9.1 HomeScreen (Main)

- Shows app title and a Floating Action Button to add new account.
- Displays a list of accounts using `LazyColumn` and `AccountRow` composable.
- The list is populated by `viewModel.getAllAccounts()` on launch and refreshed after add/update/delete.

9.2 AddNewAccountSheet (ModalBottomSheet)

- Collects `Account Name`, `Username / Email`, and `Password` via `AppTextFieldWithValue`.
- Validates non-empty input before calling `viewModel.addNewAccount(...)`.
- On success, dismisses sheet and refreshes the list.
- Theme-aware (uses `MaterialTheme.colorScheme`).

9.3 AccountDetailsSheet (ModalBottomSheet)

- Shows account details: Account Type (name), Username, and masked password with a visibility toggle.
- Provides **Edit** and **Delete** buttons.
- `Edit` opens `EditPasswordDialog` inside the sheet; `Delete` opens a confirmation `AlertDialog`.
- `onUpdate` callback passes both `oldItem` and `newItem` so updates use deterministic `id` routing.

9.4 EditPasswordDialog (AlertDialog)

- Pre-filled fields from selected account.
- Validates fields (non-empty); shows inline error message when validation fails.
- On valid Save, calls `onSave(newItem)` to the parent sheet.

9.5 AlertDialog (Delete Confirmation)

- Confirms destructive action.
- On confirm, calls `viewModel.deleteAccount(item)` which deletes by `id`.

---

10. Exception Handling Strategy

- **Database operations** (`insertPassword`, `getAllPasswords`, `updatePasswordById`, `deleteById`) are wrapped with `try/catch` and return `false`/empty lists on failures. Exceptions are printed to console for debugging.
- **Crypto operations** (encrypt/decrypt) are wrapped with `try/catch`. On encryption failure, insertion may fall back to storing plaintext (logged) — this is a fallback only during development; in production you might prefer to block the operation.

- **UI validation** prevents invalid data from being submitted (empty strings, trivially short passwords if desired).

---

11. How to Run (Setup & Build)

Prerequisites:

- Android Studio (Electric Eel or newer recommended)
- Android SDK (API 23+ recommended)
- Device or emulator with USB debugging enabled (for pulling DB)

Build & run:

1. Open project in Android Studio.
2. Build → Gradle sync.
3. Run app on an Emulator or connected device.

View DB (Debugging):

- Use Android Studio → Device File Explorer → `/data/data/com.example.passwordmanager/databases/password_manager.db` (Debug builds only), or use `adb` commands to pull the DB and open with DB Browser for SQLite.

ADB example to pull DB:

```
adb shell
run-as com.example.passwordmanager
cd databases
cp password_manager.db /sdcard/
exit
adb pull /sdcard/password_manager.db
```

---

12. Important Code Snippets (API Reference)

ViewModel

```
fun addNewAccount(account: String, user: String, pass: String): Boolean
fun getAllAccounts(): List<PasswordItem>
fun updateAccount(oldItem: PasswordItem, newItem: PasswordItem): Boolean
fun deleteAccount(item: PasswordItem): Boolean
```

Repository

```
fun addAccount(item: PasswordItem): Boolean
fun getAllAccounts(): List<PasswordItem>
fun updateAccountById(id: Int, newItem: PasswordItem): Boolean
fun deleteAccountById(id: Int): Boolean
```

DatabaseHelper (selected)

```
fun insertPassword(item: PasswordItem): Boolean
fun getAllPasswords(): List<PasswordItem>
fun updatePasswordById(id: Int, newItem: PasswordItem): Boolean
fun deleteById(id: Int): Boolean
```

Encryption utility

```
object AESCipher {
    fun encrypt(plainText: String): String
    fun decrypt(cipherText: String): String
}
```

---

13. Security & Production Notes

- **Secrets:** In the current AESCipher implementation a secret and salt are defined in code. For production, use safer key storage (Android Keystore with a hardware-backed key) or derive key from a user-provided master password.
- **Biometric unlock:** Consider gating the app with biometrics before revealing/decrypting passwords.
- **Migrations:** If you already had plaintext in DB, write a migration to encrypt existing rows safely.
- **Logging:** Avoid logging plaintext passwords in Logcat in production.

---

14. Future Enhancements (recommended)

- Move to **Room** for safer typed SQL and migrations.
- Add a **master password** (zero-knowledge) so device compromise doesn't disclose passwords.
- Add **biometric authentication** before showing details.
- Add **password strength meter** and an option for password generation.
- Sync with secure cloud (encrypted vault) if needed.

---

15. Notes for Evaluators / Interviewers

This project demonstrates:

- Solid **MVVM** architecture and separation of concerns.
- Practical handling of **local persistence** with SQLite.
- Proper UI composition with **Jetpack Compose** including bottom sheets and dialogs.
- Secure handling of sensitive data using **AES encryption**.
- Defensive programming — validation and exception handling.