Mobile Application Development (COMP2008)

Assignment 2 (Part B) – Reflection & Research

Student Name: Ngoc Nhi Tran

Student ID: 21991486

Practical Reflection

In Assignment 1, I contributed to developing a basic Android application focused on using ViewModel with Fragments and simple data display, where I handled UI elements using XML layouts and implemented basic RecyclerView for listing cheese profiles. Key challenges included managing RecyclerView adapters for dynamic data updates and ensuring proper ViewModel usage to persist state during configuration changes. For instance, I addressed list management issues by creating a custom adapter that efficiently handled item insertions and deletions, reducing UI lag. In Assignment 2 Part A, I worked on the Pocket Library App, contributing to the multi-fragment architecture, experiencing API calling, connecting to Room database, advanced filtering system, real-time search, favorites management, and detailed item views.

One of the main technical challenges I faced in Assignment 1 was managing Fragment communication. For example, when the user selected an item in a list, I needed to send that data to a detailed Fragment without tightly coupling the UI components. I solved this by introducing a shared `ViewModel`, allowing both Fragments to observe the same state using `LiveData`. This approach helped me avoid the anti-pattern of directly referencing one Fragment from another, and it resulted in cleaner, lifecycle-aware communication.

In Assignment 2, I expanded these ideas by applying state-driven list management with a ViewModel. Instead of modifying UI elements directly, I kept my data inside `MutableStateFlow` (or `LiveData`) in the ViewModel, and the UI simply reacted to state changes. This separation of concerns made the app more stable, especially during configuration changes like rotation, where previously my RecyclerView would reset or lose its scroll position. Using ViewModel allowed the list to persist across lifecycle events.

Security concerns were minimal as the app handled non-sensitive data (cheese/library items), but I considered potential issues like runtime permissions for external storage if images were loaded from files. No user data was stored persistently, avoiding risks like data leakage, though in a real scenario, I would implement EncryptedSharedPreferences for any favourite data.

Cross-App Development (3 Marks)

Modern Android permission models, such as runtime permissions and scoped storage introduced in Android 10, significantly impact cross-app data sharing by requiring explicit user consent for accessing sensitive resources like files, contacts, or location.

These models enforce least-privilege principles, preventing unauthorised access and reducing privacy risks, but they have downsides—e.g., apps must request READ_EXTERNAL_STORAGE for file access, and users can revoke permissions anytime. This affects sharing by limiting implicit data flows, forcing developers to use secure, intent-based mechanisms to avoid vulnerabilities like cross-app exploitation.

Two real-world techniques for secure sharing without compromising consent are:

1. **Content Providers**: This framework allows apps to expose data via URIs, with custom permissions to control access. Strengths include fine-grained control (e.g., query-based sharing) and integration with Android's permission system, ensuring only authorized apps access data. In contrast, implementation complexity can lead to misconfigurations exposing data unintentionally, and it requires both apps to handle URIs properly. Improvements could involve automated permission auditing tools to detect leaks.
2. **FileProvider for File Sharing**: A specialised Content Provider for temporary file access, granting URI-based permissions via intents.

   Enhances security by revoking access automatically after use and supports scoped storage, respecting user consent. But limited to files (not arbitrary data) and potential performance overhead for large files. Potential improvements include better integration with cloud syncing for seamless cross-device sharing while maintaining encryption.

These techniques align with Android's emphasis on user privacy, but require careful testing to avoid over-permissions.

Compare Jetpack Compose’s Declarative UI Approach with Traditional Android XML-Based UI Design

Jetpack Compose uses a declarative paradigm where UI is described as functions in Kotlin, automatically updating based on state changes, while XML-based design is imperative, defining static layouts inflated at runtime with code bindings.

Advantages of Jetpack Compose: Concise code reduces boilerplate (e.g., no separate XML files), enables live previews for faster iteration, and simplifies complex animations and state management with built-in reactivity. It improves developer productivity by unifying UI logic in code.

Disadvantages: Steeper learning curve for XML-experienced developers, potential integration issues with legacy views, and less mature tooling for certain edge cases like accessibility.

Advantages of XML: Clear separation of UI design and logic, promoting reusability and collaboration (designers edit XML directly); mature ecosystem with extensive libraries and better backward compatibility.

Disadvantages: Verbose and error-prone (manual ID bindings), slower development for dynamic UIs, and harder to handle state-driven changes without additional code.

Startups or small agile teams building new apps would benefit most from Jetpack Compose due to faster prototyping and modern features, reducing time-to-market. Large enterprises with legacy codebases and diverse teams might prefer XML to avoid migration costs and leverage existing expertise, ensuring stability in production environments.

Evaluating Kotlin and Jetpack Compose for Enterprise-Level Applications

Kotlin's advanced features significantly enhance enterprise apps. Null safety distinguishes nullable (?) from non-nullable types, preventing `NullPointerExceptions` at compile-time and improving security by reducing runtime crashes that could expose vulnerabilities. Extension functions allow adding methods to existing classes without inheritance, promoting cleaner, more readable code and boosting productivity by reusing logic. Coroutines simplify asynchronous programming, replacing callbacks with sequential code, and improving performance in networked apps. Overall, these features lead to higher code quality (fewer bugs), better security, and increased productivity (concise syntax cuts development time by up to 40%).

Adopting Jetpack Compose in production has pros like accelerated UI development, better maintainability through composable functions, and strong performance for complex UIs via recomposition optimisation.

Cons include a learning curve for XML-to-Compose transitions, potential bugs in early adoption, and higher initial setup for hybrid apps. Security benefits from Kotlin integration, like type-safe UIs reducing errors. Industry support is robust: Google endorses both for Android, with companies like Netflix and Square using Kotlin in enterprise apps, and Compose gaining traction in production (e.g., in multiplatform projects). Long-term viability is high due to JetBrains' maintenance and community growth, making them suitable for scalable, secure enterprise applications.

Long-Running Operations in Android and Secure Networking

Best practices for long-running operations, especially network tasks, involve offloading them to background threads to maintain UI responsiveness. For network operations, use libraries like Retrofit with coroutines for API calls, ensuring HTTPS for encryption and token-based authentication (e.g., OAuth) to secure data transmission.

Executing long-running operations on the main thread is risky because it blocks UI rendering, leading to Application Not Responding (ANR) errors, app freezes, and poor user experience. This can cause dropped frames, delayed inputs, and even force-closes.

To mitigate, use: (1) Kotlin Coroutines for lightweight async code (e.g., launch on IO dispatcher); (2) WorkManager for persistent, deferrable tasks like uploads, handling retries and constraints; (3) AsyncTask (deprecated, but for simple cases) or Executors for threading. Always monitor with StrictMode or profiling tools to detect violations.

Reference

Android Developers. (2024, January 3). Keep your app responsive. https://developer.android.com/topic/performance/anrs/keep-your-app-responsive

Android Developers. (2024, June 6). Kotlin coroutines on Android. https://developer.android.com/kotlin/coroutines

Android Developers. (2025, February 10). Content providers. https://developer.android.com/guide/topics/providers/content-providers

Android Open Source Project. (n.d.). Scoped storage. https://source.android.com/docs/core/storage/scoped

Komilov, Y. (2024, September 25). Jetpack Compose vs XML in Android: A detailed comparison. Medium. https://medium.com/@YodgorbekKomilo/jetpack-compose-vs-xml-in-android-a-detailed-comparison-aa411f5391d9