Portierungsplan – iOS-App zu Android mit Jetpack Compose und DDD

Version: 1.0

Datum: 27. September 2025

Autor: Grok 4 (xAI)

1. Einleitung

Die Aufgabe umfasst die Portierung einer Swift-basierten iOS-App auf Android unter Verwendung von Jetpack Compose. Domain-Driven Design (DDD) soll wo sinnvoll integriert werden, um die modulare Struktur zu stärken. Der Fokus liegt auf Daten-Synchronisation (remote/local) basierend auf der Klasse `CommonApplicationBaseModuleLoader`.

Der Originaltext aus dem bereitgestellten Dokument (Aufgabenstellung_fuerAIs.pdf) beschreibt:

- Die Notwendigkeit, eine bestehende iOS-App (in Swift) auf Android (Jetpack Compose) zu portieren, mit optionaler Anwendung von DDD.

- Den Aufbau der iOS-App: Eine modulare Struktur mit Common-Modulen (Application, Domain, Infrastructure, Presentation, Util) und Features (z. B. AppInit, Blog, Mensa, News).

- Code-Ausschnitte einer zentralen Klasse für Daten-Sync, inklusive Imports, Initialisierung, Prozessierung, Reset und Sync-Logik.

- Die Frage: Wie vorgehen, was beachten, Analyse und Zusammenfassung in einem Word-Dokument.

2. Analyse der Aufgabe

- **Aktueller Stand der iOS-App**:

  - Root-Ordner: multitenant-ios-swiftui.

  - Common-Module: Für geteilte Logik (Application, Domain, Infrastructure, Presentation, Util).

  - Sources: Tests und Features (FeatureAppInit, FeatureBlog, FeatureMensa, FeatureNews, FeatureParking, FeatureRoomSearch, FeatureSettings, FeatureTrails, FeatureWebLinks).

  - Frameworks: AuthenticationServices.

  - Products: HSLU.app.

  - Tenants: HSLU.TA (mit App-Info, Assets, Constants, Localizable-Strings, UI).

- **Kernfunktionen aus dem Code**:

  - Die Klasse `CommonApplicationBaseModuleLoader` handhabt asynchrone Daten-Synchronisation (remote/local), Netzwerkprüfungen, Storage (Datei-basiert), Status-Updates und Reset-Funktionen.

  - Unterstützte Modi: Hybrid (implementiert), Local/Remote (nicht implementiert).

  - Abhängigkeiten: Foundation, Network, SwiftUI, Common-Module, OSLog.

  - Wichtige Methoden: `setup`, `process`, `reset`, `syncData` – mit Logik für Timestamp-Vergleiche, Fallback auf Cache und Fehlerbehandlung.

- **Herausforderungen**:

  - Plattformwechsel: Swift zu Kotlin, SwiftUI zu Compose, async/await zu Coroutines.

  - DDD-Integration: Sinnvoll für komplexe Domänen (z. B. Sync als Domain Service), aber nicht übertreiben.

  - Multi-Tenancy: Tenant-Configs beibehalten.

3. Vorgeschlagener Ansatz

Ein schrittweiser, iterativer Plan für die Portierung:

Phase	Dauer (geschätzt)	Aktivitäten
Vorbereitung und Analyse	1–2 Wochen	- Vollständiger Code-Review des iOS-Codes. - Identifikation von DDD-Elementen (Bounded Contexts, Entities, Repositories). - Setup von Android Studio, Kotlin, Jetpack Compose, Coroutines, Retrofit, Room, Hilt. - Prüfung auf Kotlin Multiplatform für shared Code.
Architektur-Design	1 Woche	- Anwendung von Clean Architecture oder DDD-Layering (Domain, Data, Presentation, Infrastructure). - Module-Struktur: Ähnlich iOS (z. B. separate Gradle-Module für CommonDomain). - Portierung der Kernklasse als ViewModel mit StateFlow.
Implementierung	4–6 Wochen	- Core-Logik portieren (Sync, Network, Storage). - Features iterativ umsetzen (z. B. News-Modul zuerst). - DDD anwenden: Data classes für Entities, Use Cases für Business-Logik. - UI mit Compose (Composables für Views).
Testing und Optimierung	2 Wochen	- Unit-Tests (Domain), Integration-Tests (Data), UI-Tests (Compose). - Offline-Support, Performance-Tests, Sicherheit (z. B. Permissions). - Debugging von Sync-Fehlern (z. B. Timestamps, Netzwerk).
Deployment und Maintenance	Laufend	- Build und Release (APK/AAB, Play Store). - Monitoring (Firebase). - Erweiterbarkeit durch DDD sicherstellen.

- **DDD-Elemente im Detail**:

  - **Entities/Aggregates**: Immutable data classes für DTOs (z. B. AppNewsModuleItemAPIDTO).

  - **Repositories**: Interfaces für Storage/Network-Abstraktion (z. B. SyncRepository mit Hilt-Injection).

  - **Domain Services/Use Cases**: Für Sync-Prozesse (z. B. SyncDataUseCase).

  - **Value Objects**: Für Configs (z. B. AppTenantConfig).

  - Anwenden nur wo wertschöpfend: Bei einfachen Features vermeiden, um Komplexität zu reduzieren.

4. Wichtige Beachtungspunkte

- **Technische Unterschiede**:

  - Async: Coroutines/Flow statt async/await; StateFlow für @Published-Variablen.

  - Storage: Room oder Files statt FileManager; JSON-Serialisierung mit Kotlinx.

  - Network: ConnectivityManager/Retrofit statt NWPathMonitor; Permissions handhaben.

  - UI: Declarative Compose statt SwiftUI; States für reaktive Updates.

- **Risiken und Best Practices**:

  - Netzwerk: Offline-Fallback, Reachability-Checks, Zeitzonen bei Timestamps.

  - Performance: Caching optimieren, Battery/Network schonen.

  - Sicherheit: API-Endpoints schützen, Datenverschlüsselung.

  - DDD: Selektiv nutzen, um Boilerplate zu vermeiden; Fokus auf Modulare Erweiterbarkeit.

  - Testing: Vollständige Abdeckung, insbesondere für Hybrid-Modus.

  - Rechtlich: DSGVO-Konformität bei Daten-Sync (User-Consent).

- **Ressourcen**:

  - Tools: Android Studio, Git.

  - Zeit: Ca. 8–12 Wochen für ein kleines Team.

  - Externe Ressourcen: Links aus dem Original (z. B. FileManager-Tutorial, StackOverflow für Async).

5. Nächste Schritte

- Detaillierten Review des vollständigen iOS-Codes durchführen.

- Prototyp für Sync-Modul in Android erstellen.

- Team-Diskussion zur Priorisierung von Features.

Anhang: Beispiel für portierten Code (Kotlin)

Hier ein Auszug einer portierten Version der `syncData`-Methode:

```kotlin
class CommonApplicationBaseModuleLoader<I : CommonAppBaseModuleItemAPIDTO> :
ViewModel() {
    val loadingStatus: MutableStateFlow<CommonModuleLoaderStatus<I>> =
MutableStateFlow(CommonModuleLoaderStatus.NotInitialized)

    // ... Weitere Properties

    suspend fun syncData(): Triple<Boolean, String, List<I>> {
        var remoteSyncData: Pair<CommonAppBaseModuleSyncAPIDTO?, ByteArray>? = null
        var remoteUpdateTS = ""
        var localUpdateTS = ""
        var canUpdate = false
        var updateNeeded = false
        var storeSyncInfo = false

        if (networkService.hasConnection()) {
            remoteSyncData = networkService.getAsJsonObject(appConfig.moduleLoaderSyncURL)
            remoteUpdateTS = remoteSyncData?.first?.moduleLastUpdated ?: ""
            // ... Fortsetzung der Logik analog zum Swift-Code
        }
        // ... Vollständige Implementierung mit Coroutines und Retrofit
    }
```

}

```

Dieses Dokument kann direkt in Microsoft Word kopiert und formatiert werden. Falls Anpassungen benötigt werden, lass es mich wissen!
```