Gerne! Ich erstelle eine Word-Dokument-Vorlage mit der vollständigen Analyse und dem Vorgehensplan.

Analyse und Migrationsplan: iOS SwiftUI zu Android Jetpack Compose

1. Einleitung

Auftragsziel
Portierung einer bestehenden iOS-App (Swift/SwiftUI) zu Android mittels Jetpack Compose unter Berücksichtigung von Domain-Driven Design (DDD) und Multi-Tenancy-Fähigkeit.

Ausgangslage
- **iOS-App**: Modular aufgebaut, multi-tenant-fähig
- **Technologie**: SwiftUI, Swift Package Manager, async/await
- **Architektur**: Klare Trennung in Common- und Feature-Module
- **Daten-Sync**: Hybrid-Ansatz (Remote/Local) mit Offline-Fähigkeit

---

2. Analyse der bestehenden iOS-App

2.1 Modulstruktur
```
multitenant-los-swiftui/
 ├── Common/ (Shared Libraries)
│    ├── CommonApplication
│    ├── CommonDomain
│    ├── CommonInfrastructure
│    ├── CommonPresentation
│    └── CommonUtil
 ├── Features/ (Feature Modules)
│    ├── FeatureAppInit
│    ├── FeatureBlog
│    ├── FeatureMensa
│    └── ...
└── Tenants/ (Multi-Tenancy)
    ├── HSLU/
    └── HSLU_TA/
```

2.2 Multi-Tenancy-Implementierung
- Tenant-spezifische Konfigurationen (Assets, URLs, Strings)
- Dynamische Lade-Mechanismen
- Gemeinsame Codebasis mit tenant-spezifischen Anpassungen

2.3 Technische Besonderheiten
- **Daten-Synchronisation**: Hybrid (Remote/Local), Caching-Strategie
- **Async-Handling**: Swift async/await mit Status-Updates
- **Modulare Abhängigkeiten**: Klare Trennung via Swift Packages

---

3. Architektur-Entwurf für Android

3.1 Gesamtarchitektur
Domain-Driven Design (DDD) mit Clean Architecture:
- **Domain Layer**: Entities, Use Cases, Repository Interfaces
- **Data Layer**: Repositories, Data Sources (Remote, Local)
- **Presentation Layer**: ViewModel + Jetpack Compose

3.2 Modulstruktur (Android)
```
app/
├── base/ (Common Module)
├── features/ (Feature Modules)
│   ├── blog/
│   ├── mensa/
│   └── ...
├── tenants/ (Multi-Tenancy Config)
└── core/ (Shared Infrastructure)
```

3.3 Multi-Tenancy-Architektur
- Dynamische Resource-Loading (Strings, Assets)
- Tenant-spezifische Dependency Injection
- Konfigurations-Management zur Laufzeit

---

4. Technische Umsetzungsstrategie

4.1 Technologie-Stack
iOS	**Android Äquivalent**
SwiftUI	Jetpack Compose
Swift Package Manager	Gradle Modules
async/await	Kotlin Coroutines
FileManager	Room + DataStore
URLSession	Retrofit
@Published/ObservableObject	StateFlow/ViewModel

4.2 Daten-Synchronisation
```kotlin
```

```
// Android-Implementierung des Sync-Mechanismus
class SyncManager(
    private val networkService: NetworkService,
    private val localStorage: LocalStorage
) {
    suspend fun syncData(): SyncResult {
        // Implementierung analog zur iOS-Logik
    }
}
```

4.3 Dependency Injection
- **Dagger Hilt** für dependency management
- Tenant-spezifische Module zur Laufzeit

---

5. Detaillierter Migrationsplan

Phase 1: Grundgerüst (Wochen 1-4)
- [ ] Android-Projekt-Struktur einrichten
- [ ] Gradle-Module für Common-Komponenten
- [ ] Basis-Architektur (DDD) implementieren
- [ ] Netzwerk-Schicht (Retrofit)
- [ ] Datenbank (Room) einrichten

Phase 2: Core-Features (Wochen 5-10)
- [ ] Multi-Tenancy-Infrastruktur
- [ ] Sync-Mechanismus portieren
- [ ] Dependency Injection einrichten
- [ ] Basis-UI-Komponenten (Compose)

Phase 3: Feature-Module (Wochen 11-20)
- [ ] FeatureBlog portieren
- [ ] FeatureMensa portieren
- [ ] FeatureNews portieren
- [ ] Weitere Features nacheinander

Phase 4: Finalisierung (Wochen 21-24)
- [ ] Testing (Unit Tests, UI Tests)
- [ ] Performance-Optimierung
- [ ] Tenant-spezifische Anpassungen
- [ ] Store-Release vorbereiten

---

6. Risikoanalyse und Herausforderungen

6.1 Technische Risiken
- **Plattform-Unterschiede**: iOS/Android-APIs nicht 1:1 übertragbar
- **Performance**: Große lokale Datenspeicherung optimieren
- **Sync-Komplexität**: Fehlerbehandlung bei Netzwerkproblemen

6.2 Lösungsansätze
- **Abstraktion**: Plattform-spezifische Code-Wrapper
- **Testing**: Umfangreiche Testabdeckung für kritische Pfade
- **Iterative Entwicklung**: Frühzeitige Validierung komplexer Komponenten

---

7. Erfolgskriterien

7.1 Funktionale Anforderungen
- ✅ Vollständige Feature-Parität zur iOS-App
- ✅ Multi-Tenancy unterstützt
- ✅ Offline-Fähigkeit erhalten
- ✅ Performance vergleichbar mit iOS-Version

7.2 Nicht-funktionale Anforderungen
- ✅ Saubere Architektur (DDD)
- ✅ Wartbarer und erweiterbarer Code
- ✅ Umfassende Testabdeckung
- ✅ Google Play Store Guidelines konform

---

8. Empfehlungen

8.1 Technische Entscheidungen
- **Kotlin Multiplatform (KMP)** für zukünftige Cross-Platform-Entwicklung evaluieren
- **Compose Navigation** für type-safe Navigation verwenden
- **Material Design 3** für konsistentes UI/UX

8.2 Entwicklungsprozess
- **Agile Methodik** mit 2-wöchigen Sprints
- **Code Reviews** für qualitätssicherung
- **CI/CD Pipeline** für automatisiertes Testing und Deployment

---

9. Zusammenfassung

Die Migration von iOS SwiftUI zu Android Jetpack Compose ist ein komplexes, aber machbares Vorhaben. Durch die Beibehaltung der modularen Architektur und die Anwendung von Domain-Driven Design kann eine wartbare und skalierbare Android-App

entstehen. Der oben beschriebene Plan bietet einen strukturierten Rahmen für eine erfolgreiche Portierung.

Empfohlener nächster Schritt: Detaillierte Spezifikation für das "Common"-Modul und die Multi-Tenancy-Infrastruktur erstellen.

Dokumentversion: 1.0
Erstellt am: $(date)
Autor: AI-Assistent
Status: Entwurf zur Diskussion

Möchtest du, dass ich dieses Dokument als Word-Datei (.docx) für dich generiere, oder sollen bestimmte Abschnitte noch angepasst werden?