

HSLU Mobile Apps - Android Jetpack Compose - AI First

Wirtschaftsprojekt Herbstsemester 2025

Autoren: Raphael Eiholzer und Samuel Kurmann

Betreuer: Jürg Nietlispach

Hochschule Luzern – Departement Informatik

Rotkreuz, Schweiz

11. Dezember 2025

Wirtschaftsprojekt an der Hochschule Luzern – Informatik

Titel: HSLU Mobile Apps - Android Jetpack Compose - AI First

Student: Samuel Kurmann

Student: Raphael Eiholzer

Studiengang: BSc Informatik

Jahr: 2025

Betreuungsperson: Jürg Nietlispach

Experte: Martin Vogel

Auftraggeber: Jürg Nietlispach / Departement Informatik

Codierung / Klassifizierung der Arbeit:

Öffentlich Vertraulich

Eidesstattliche Erklärung Ich erkläre hiermit, dass wir die vorliegende Arbeit selbständig und ohne unerlaubte fremde Hilfe angefertigt haben, alle verwendeten Quellen, Literatur und andere Hilfsmittel angegeben haben, wörtlich oder inhaltlich entnommene Stellen als solche kenntlich gemacht haben, das Vertraulichkeitsinteresse des Auftraggebers wahren und die Urheberrechtsbestimmungen der Hochschule Luzern respektieren werden.

Ort / Datum, Unterschrift _____

Ort / Datum, Unterschrift _____

Ausschliesslich bei Abgabe in gedruckter Form:

Eingangsvišum durch das Sekretariat auszufüllen

Rotkreuz, den _____

Višum: _____

Abstract / Zusammenfassung oder Management Summary

Todo Raphi

Inhaltsverzeichnis

Abstract	i
1. Problem, Fragestellung, Vision	1
1.1. Ausgangslage und Problemstellung	1
1.2. Ziel der Arbeit und erwartete Resultate	1
2. Stand der Forschung oder Stand der Praxis/Technik	2
3. Ideen und Konzepte	3
4. Methode(n)	4
4.1. Gewünschte Methoden und Vorgehen	4
4.2. Kreativität, Methoden, Innovation	4
4.3. Projektmanagement	4
4.3.1. Stakeholder	4
4.3.2. Agiles Vorgehen	4
4.3.3. Planung	5
4.3.4. Risikoanalyse	6
5. Realisierung	8
5.1. Applikationsarchitektur und Konfiguration	8
5.1.1. Komponenten und Bibliotheken	8
5.1.2. BuildConfig im Projekt	10
5.1.3. Fastlane	12
5.2. Common-Komponenten	13
5.2.1. CommonApplication	13
5.2.2. CommonDomain	14
5.2.3. CommonInfrastructure	15
5.2.4. CommonPresentation	18
5.2.5. CommonUtil	19
5.3. Features	20
5.3.1. AppInit	22
5.3.2. News	25
5.3.3. Blog	26
5.3.4. Mensa	27
5.3.5. Events	28
5.3.6. Info/About	29
5.3.7. Settings	30
5.3.8. Weblinks	31
5.3.9. CampusRoomSearch	32
5.3.10. Timetable	33
6. Validierung und Evaluation	34
6.1. Teststrategie – Android Multi-Tenant App (Jetpack Compose)	34
6.1.1. Ausgangslage	34
6.1.2. Zielsetzung	34
6.1.3. Umsetzung	34
6.1.4. Struktur und Modularität	35
6.1.5. Zielwerte und Qualität	35

6.1.6. Fazit	35
6.2. Evaluation AI Tools	36
6.2.1. Aufgabenstellung für die AI Tools	36
6.2.2. Antworten	46
6.2.3. Vergleich	63
6.2.4. Detaillierter Fragen zu Cursor	63
6.2.5. Fazit	63
7. Ausblick	66
A. Anhänge	67
A.1. Zusatzmaterial	67
A.2. Protokolle und Statusberichte	67
A.2.1. Kickoff-Meeting (19.09.2025)	68
A.2.2. Projektstatus-Meeting (25.09.2025)	70
A.2.3. Projektstatus-Meeting (16.10.2025)	72
A.2.4. Projektstatus-Meeting (30.10.2025)	74
A.2.5. Projektstatus-Meeting (13.11.2025)	76
A.2.6. Projektstatus-Meeting (11.11.2025)	78
A.2.7. Status-Report (16.10.2025)	79
A.2.8. Status-Report (30.10.2025)	80
A.2.9. Status-Report (13.11.2025)	82
A.2.10. Status-Report (11.12.2025)	84
Abkürzungs-, Abbildungs-, Tabellen-, KI-, Formelverzeichnis	86

1. Problem, Fragestellung, Vision

Beschreibe das Problem, die Zielsetzung und die übergeordnete Vision des Projekts.

1.1. Ausgangslage und Problemstellung

Die HSLU-Informatik betreibt eigene Mobile-Apps für die Departemente Informatik sowie Technik & Architektur auf den Plattformen *iOS* und *Android*. Im Rahmen einer Forschungsarbeit wurde die Codebasis für *iOS* stark erweitert; diese Änderungen müssen nun in die *Android*-Codebasis auf Basis von Jetpack Compose übernommen werden.

Das bestehende *Android*-Projekt basiert noch auf XML-Layouts und ist nicht an das aktuelle Backend-API angebunden. Dies soll im Rahmen von AI-First-Techniken erfolgen. In diesem Zusammenhang sollen verschiedene Möglichkeiten, Ansätze und Tools evaluiert werden. Die passendste Technologie wird daraufhin für die konkrete Umsetzung verwendet.

1.2. Ziel der Arbeit und erwartete Resultate

Ziel: Eine vollständig funktionsfähige Jetpack-Compose-App soll im Google Play Store veröffentlicht werden. Die Umsetzung erfolgt als Single Codebase mit Multi-Feature- und Multi-Tenant-Unterstützung unter Anwendung von Domain-Driven Design (DDD).

Erwartete Resultate:

- Projektmanagement-Artefakte: Statusberichte, Projektplan inklusive fortlaufender Risikoanalyse, Issues/Epics und Stakeholderübersicht.
- App-Artefakte:
 - Evaluation und Auswahl geeigneter AI-Ansätze und Tools für Mobile Engineering.
 - Implementierung der fehlenden Features (bzw. Neuaufbau) in Jetpack Compose mit Tests und Dokumentation für aktuelle targetSDKs.
 - Release des neuesten Entwicklungsstands via GitLab CI/CD und Fastlane in den Google Play Store.
- HSLU-Artefakte: Projektdokumentation gemäss HSLU-Standards sowie Zwischen- und Schlusspräsentation.

2. Stand der Forschung oder Stand der Praxis/Technik

Erkläre den aktuellen Stand der Technik, Forschung oder Praxis zum Thema. Todo

3. Ideen und Konzepte

Führe die entwickelten Ideen, Konzepte oder Ansätze aus, die im Projekt verfolgt werden. Todo

4. Methode(n)

Beschreibe die verwendeten Methoden, Vorgehensmodelle oder Frameworks.

4.1. Gewünschte Methoden und Vorgehen

Inkrementelles, iteratives, agiles Vorgehen. Regelmässige Treffen mit dem Auftraggeber finden statt, inklusive Statusbericht am Vorabend. Der Statusbericht umfasst erledigte Aufgaben, den aktuellen Zwischenstand, eine Risikobewertung sowie die nächsten geplanten Arbeiten. Zur Nachvollziehbarkeit von Fortschritt und Aufgaben wird ein Projektmanagement-Tool verwendet, das eine transparente Verfolgung der Issues ermöglicht.

4.2. Kreativität, Methoden, Innovation

Im Projekt sollen moderne Technologien eingesetzt und evaluiert werden, wie beispielsweise AI-gestützte Entwicklungswerkzeuge, Jetpack Compose oder CI/CD-Automatisierungen. Die Unterstützung durch künstliche Intelligenz wird aktiv in den Entwicklungsprozess eingebunden, dokumentiert und reflektiert. Je nach Erkenntnissen im Projektverlauf wird die Gewichtung des AI-Anteils flexibel angepasst, um die bestmöglichen Resultate zu erzielen.

4.3. Projektmanagement

4.3.1. Stakeholder

- Auftraggeber / Dozent: Jürg Nietlispach
- Experte: Martin Vogel
- Projektteam: Raphael Eiholzer, Samuel Kurmann

4.3.2. Agiles Vorgehen

Im Projekt wurde nach dem agilen Vorgehensmodell *Scrum* gearbeitet. Die Sprints wurden jeweils über zwei Wochen definiert, wodurch sich insgesamt sieben Sprints ergaben. Am Ende jedes Sprints fand eine Besprechung mit dem Auftraggeber statt, um den Fortschritt zu präsentieren und den nächsten Sprint zu planen.

Zu Beginn wurden alle Aufgaben in GitLab als Issues erfasst und in das Product Backlog aufgenommen. Während der Sprintplanung wurden die Issues nach Priorität ausgewählt und dem kommenden Sprint zugeordnet. Dieses Vorgehen ermöglichte eine strukturierte und transparente Projektabwicklung mit klaren Zuständigkeiten und Fortschrittskontrolle.

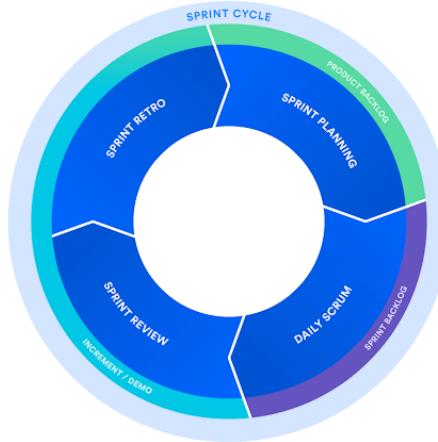


Abbildung 1.: Darstellung des Scrum-Prozesses im Projekt
(atlassian_was_nodate)

4.3.3. Planung

Zu Beginn des Projekts wurde eine detaillierte Planung der auszuführenden Arbeitsschritte erstellt. Dazu erfolgte zunächst eine Analyse der bestehenden iOS-App und ein Vergleich mit der bisherigen Jetpack-Compose-Version. Die wichtigsten Meilensteine wurden in einem Plan zusammengeführt, um den Projektfortschritt wöchentlich zu überprüfen und Abweichungen frühzeitig zu erkennen.

Meilensteinplanung WIPRO "HSLU Mobile Apps - Android Jetpack Compose - AI First"

* = Meilenstein, ☰ = Projektmeeting
Die grünen Balken symbolisieren den geplanten Zeitraum und die blauen Balken den effektiv benötigten Zeitraum. Diese Zeittafel soll einen groben Überblick liefern, wann die wichtigsten Projektschritte stattfinden. Die effektive Projektplanung und Umsetzung folgt nach iterativem Vorgehen in insgesamt 7 Sprints.

	Sprint 1	Sprint 2	Sprint 3	Sprint 4	Sprint 5	Sprint 6	Sprint 7									
	PW01	PW02	PW03	PW04	PW05	PW06	PW07	PW08	PW09	PW10	PW11	PW12	PW13	PW14		Januar
Tätigkeit																
Kickoff																
Überarbeitung und Abgabe der Aufgabenstellung	🕒															
Signierung der Aufgabenstellung	*	🕒														
Erstellung Projektplan	🕒															
Projektstart & Evaluationsphase																
Einrichtung Projekte	🕒															
Recherche zu AI-Tools, Festlegung der Bewertungskriterien	🕒															
Tests, engere Auswahl und Evaluation der gewählten AI-Tools	🕒															
Einarbeitung und studieren der bisherigen Projektarbeiten	🕒															
Erarbeitung Product Backlog	🕒															
Umsetzung und Verifikation																
Schrittweise Portierung der iOS-Features zu Android	🕒	*	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	
Automatisiertes Testing	🕒															
Projektdokumentation	🕒															
Vorbereitung Zwischenpräsentation								*								
Abschlussphase (Auswerten)																
Abschluss Projektdokumentation & Fazit													*			
Vorbereitung Abschlusspräsentation													🕒		*	

Abbildung 2.: Meilensteinplan des Projekts

Die definierten Meilensteine wurden in GitLab abgebildet und die einzelnen Arbeiten als Issues erfasst und den entsprechenden Meilensteinen zugeordnet. Die Aufgaben wurden in zwei Kategorien unterteilt:

- *Common-Arbeiten*: allgemeine App-Funktionalitäten, die von mehreren Modulen genutzt werden (z. B. Netzwerkkommunikation)
- *Feature-Arbeiten*: spezifische App-Funktionen oder Menüpunkte, die unabhängig voneinander entwickelt wer-

den können

Die Aufgaben wurden so gestaltet, dass sie von einer einzelnen Person vollständig bearbeitet werden konnten. Die Reihenfolge der Issues wurde zu Projektbeginn grob festgelegt, aber jeweils in der Sprintplanung überprüft und bei Bedarf angepasst.

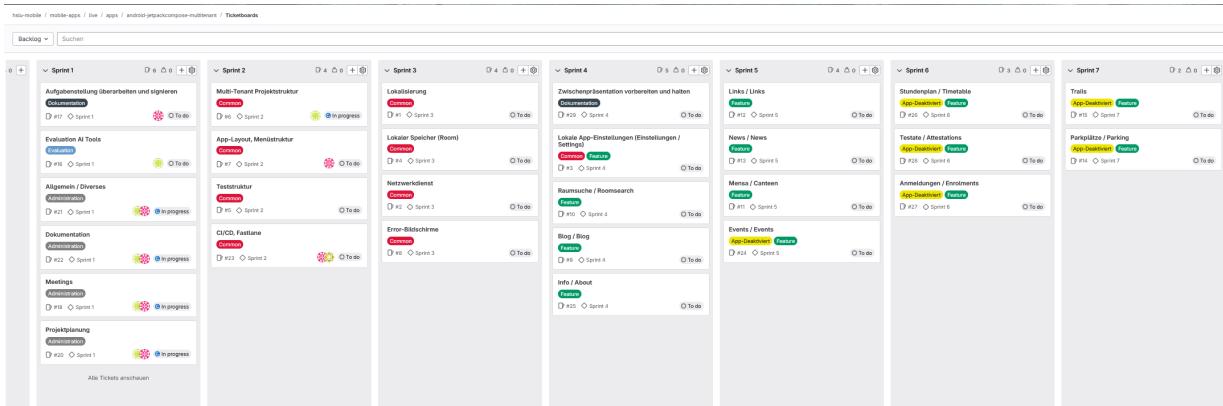


Abbildung 3.: Beispielhafte Darstellung des Backlogs in GitLab

Jedes Issue wurde als User Story formuliert und mit einer Definition of Done (DoD) versehen. Die Projektmitglieder konnten sich selbst Issues zuweisen, wodurch eine klare Arbeitsaufteilung gewährleistet war. Für jede Aufgabe wurde die aufgewendete Zeit direkt im Issue dokumentiert, um den Arbeitsaufwand über die gesamte Projektlaufzeit nachvollziehen zu können.

Raumsuche / Roomsearch

Offen Ticket erstellt vor 2 Wochen von Raphael Eiholzer

Als Nutzer:in möchte ich Räume schnell finden und mir den zugehörigen Plan/Ort anzeigen lassen, damit ich mich auf dem Campus effizient orientieren kann.

- View für Raumsuche mit Textfeld und Trefferliste in Jetpack Compose umgesetzt.
- Detailansicht für ausgewählten Raum erstellt und über Navigation erreichbar.
- Zoom / Verschiebung / Auswahl bei Detailansicht analog iOS umgesetzt
- Informationsbildschirm für ausgewählten Raum dargestellt

vor 1 Woche von Raphael Eiholzer bearbeitet



Design hinzufügen

Erstelle Merge Request

Status	Bearbeiten
To do	
Beauftragte	Bearbeiten
Keine - weise dich selbst zu	
Labels	Bearbeiten
Feature	
Übergeordnet	Bearbeiten
Keine	

Abbildung 4.: Beispiel einer User Story mit Definition of Done (DoD)

4.3.4. Risikoanalyse

Zu Beginn des Projekts wurde eine Risikoanalyse erstellt, um die grössten Risiken frühzeitig zu identifizieren und entsprechende Gegenmassnahmen zu planen. Die Risiken wurden hinsichtlich Eintrittswahrscheinlichkeit und Auswirkung bewertet. Das Produkt dieser beiden Faktoren bestimmte den Gesamtrisiko-Wert und erlaubte es, die kritischsten Risiken zu priorisieren.

Für jedes Risiko wurden präventive Massnahmen festgelegt, um die Eintrittswahrscheinlichkeit oder die Auswirkungen zu minimieren. Nach Umsetzung dieser Massnahmen wurde die Bewertung erneut vorgenommen, um verblei-

bende Risiken zu identifizieren und ihre Entwicklung über die Projektlaufzeit zu beobachten.

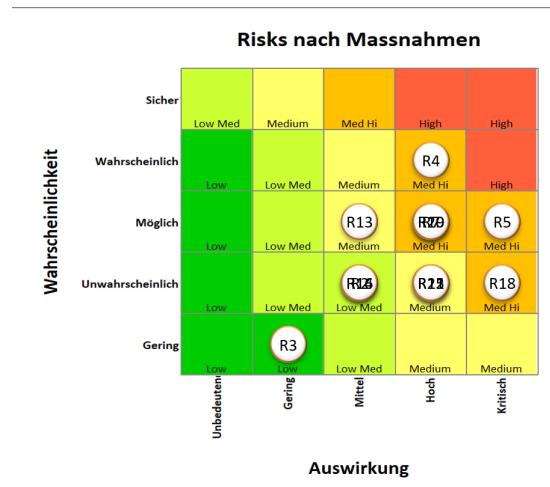


Abbildung 5.: Risikoanalyse und Risikomatrix des Projekts

Die Risikoanalyse wurde während der gesamten Projektdauer regelmässig überprüft und aktualisiert. Am Ende jedes Sprints diskutierten die Studierenden gemeinsam die aktuelle Risikosituation und leiteten mögliche Anpassungen ein. Die drei grössten Risiken wurden zusätzlich an den Auftraggeber übermittelt, um Transparenz über den Projektfortschritt und potenzielle Herausforderungen sicherzustellen.

5. Realisierung

5.1. Applikationsarchitektur und Konfiguration

Einleitung

Das Ziel der Entwicklung war es, eine modulare und erweiterbare Applikation zu schaffen. Dazu wurde auf die bestehende iOS-Projektstruktur aufgebaut und das Android-Projekt analog dazu umgesetzt.

Im Wesentlichen wurde die Anwendung in die beiden Hauptmodule `common` und `features` gegliedert:

- **common:** Enthält alle allgemeinen und wiederverwendbaren Komponenten, wie z. B. UI-Elemente, Utility-Klassen und grundlegende Architekturnausteine.
- **features:** Beinhaltet die einzelnen Funktionsmodule der App, die jeweils auf spezifische Anwendungsbereiche oder Features ausgerichtet sind.

Dabei gilt die Abhängigkeitsregel, dass `features`-Module auf Inhalte aus `common` zugreifen dürfen, nicht jedoch umgekehrt – um zirkuläre Abhängigkeiten und eine enge Kopplung zu vermeiden.

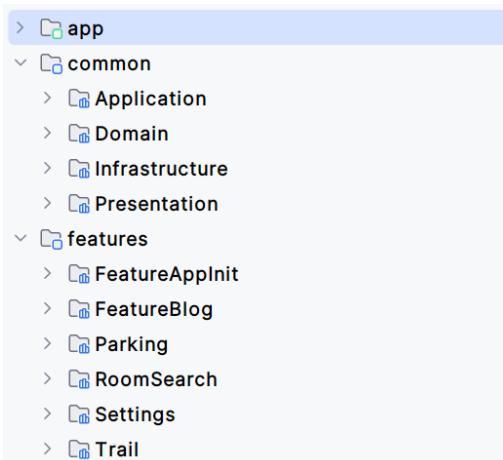


Abbildung 6.: Projektstruktur im Android Studio

5.1.1. Komponenten und Bibliotheken

Zu Beginn des Projekts wurde der bestehende Build-Prozess aus dem früheren `android-xml-multitenant`-Projekt analysiert und modernisiert. In der älteren Variante wurden noch veraltete SDKs und Bibliotheken eingesetzt, die nicht mehr dem aktuellen Stand der Android-Entwicklung entsprachen. Im Zuge der Neumodernisierung mit Jetpack Compose wurde daher der gesamte Build auf eine aktuelle Versionen der Bibliotheken umgestellt.

Dies umfasste die Aktualisierung auf ein neues Android-SDK-Level, die Migration auf eine moderne Kotlin-Version (2.1.21) sowie die Aktualisierung der zentralen Bibliotheken.

Im Folgenden werden die wichtigsten Plugins, SDK-Einstellungen und externen Abhängigkeiten beschrieben, die die Grundlage der neuen Projektstruktur bilden.

5.1.1.1. Build Plugins

Im Projekt werden folgende Gradle-Plugins verwendet:

- **com.android.application**: Basisplugin für Android-Apps.
- **org.jetbrains.kotlin.android**: Aktiviert Kotlin-Unterstützung für Android.
- **org.jetbrains.kotlin.plugin.compose**: Integration von Jetpack Compose in Kotlin.
- **com.google.devtools.ksp**: Kotlin Symbol Processing (KSP) für Codegenerierung (z. B. Hilt, Room).
- **dagger.hilt.android.plugin**: Aktiviert Dependency Injection mit Hilt.

Diese Plugins stellen die Grundlage für Build-Prozess, Codegenerierung und Abhängigkeitsverwaltung dar. Zusätzliche Projekt-Module (:common, :features) werden über `implementation project(...)` eingebunden, um eine klare Trennung zwischen gemeinsamem Code und Feature-spezifischer Logik zu ermöglichen.

5.1.1.2. SDK-Konfiguration

Das Projekt wurde im Zuge der Modernisierung auf die aktuellste Android-SDK-Version angehoben, um die neuesten Funktionen, Sicherheitsmechanismen und Performance-Verbesserungen der Plattform zu nutzen. Seit dem Release von **Android 16 (SDK 36, Juni 2025)** verwendet das Projekt sowohl die `compileSdk` als auch die `targetSdk` in Version 36. Dadurch ist sichergestellt, dass die App alle modernen APIs nutzt und gleichzeitig mit den aktuellen Play-Store-Richtlinien kompatibel bleibt.

- **compileSdk**: 36
- **targetSdk**: 36
- **minSdk**: 30

Mit der `minSdk`-Version 30 wird weiterhin Android 11 (veröffentlicht im Jahr 2020) unterstützt, was eine breite Geräteabdeckung sicherstellt. Damit ist die App auch auf Geräten lauffähig, die bis zu fünf Jahre alt sind, und deckt somit den im Projekt festgelegten Zielzeitraum von **maximal 3–4 Jahren** typischer Gerätelebensdauer ab (siehe Anhang A.2.3). Diese Konfiguration gewährleistet, dass die Anwendung moderne Android-Technologien nutzt, ohne dabei die Kompatibilität zu verbreitet eingesetzten Geräten zu verlieren. Sie entspricht zudem den Empfehlungen der offiziellen Android-Dokumentation (`noauthor_uses-sdk_nodate`).

5.1.1.3. Wesentliche Bibliotheken und Abhängigkeiten

Im Folgenden sind die wichtigsten externen Bibliotheken aufgeführt, die in der `build.gradle`-Datei des App-Moduls verwendet werden:

- **Jetpack Compose** (`androidx.compose.ui`, `material`, `material3`): Für deklaratives UI-Design mit reaktiver State-Verwaltung.
- **AndroidX Core und Lifecycle** (`core-ktx`, `lifecycle-runtime-ktx`, `activity-compose`): Basisfunktionen und Lifecycle-Unterstützung für moderne Android-Komponenten.
- **Navigation Compose** (`androidx.navigation:navigation-compose`): Zur Verwaltung von Navigation und Routen in Compose-Anwendungen.
- **Hilt (Dependency Injection)** (`com.google.dagger:hilt-android`, `androidx.hilt:hilt-navigation-compose`): Ermöglicht modulare, testbare und skalierbare Architektur durch Dependency Injection.
- **DataStore** (`androidx.datastore:datastore-preferences`): Moderne, asynchrone und typsichere Alternative zu SharedPreferences.
- **JUnit 5 und Android Test Libraries**: Einheitliches Testframework mit Unterstützung für Unit- und Instrumentation-Tests (`org.junit.jupiter`, `androidx.test.ext:junit`, `espresso-core`).
- **Accompanist** (`pager`, `pager-indicators`, `webview`, `permissions`): Erweiterungsbibliotheken von Google zur Ergänzung von Jetpack Compose (z. B. ViewPager, WebView, Berechtigungsdialoge).
- **ZXing & AltBeacon** (`com.google.zxing:core`, `org.altbeacon:android-beacon-library`): QR-Code-Scanning und Beacon-Erkennung für die Trail-Funktion.
- **Gson**: Für JSON-Serialisierung und -Deserialisierung.

5.1.2. BuildConfig im Projekt

Im Android-Projekt wurde die BuildConfig-Struktur zentralisiert und tenant-spezifisch erweitert. Ziel war es, alle wichtigen Konfigurationswerte und Feature-Flags pro Tenant konsistent an einer Stelle zu definieren, um Redundanzen und Hardcodings zu vermeiden.

5.1.2.1. Idee und Motivation

In der vorherigen Umsetzung waren Konfigurationen über verschiedene Klassen verteilt, beispielsweise in `HsluiApplication` oder einzelnen Modulen. Dies erschwerte die Wartung und Übersicht. Neu werden sämtliche Tenant-spezifischen Werte zentral im `app/build.gradle` verwaltet. Dort sind alle BuildConfig-Felder je Flavor (`hslui` und `hsluta`) definiert und werden beim Build-Prozess automatisch generiert.

5.1.2.2. Implementierung

Jeder Tenant besitzt im Abschnitt `productFlavors` eigene Einträge. Dabei werden Variablen wie `TENANT_ID`, `API_BASE_URL` oder `CLIENT_TOKEN` über `buildConfigField` gesetzt. Zur Laufzeit greift der Code direkt über `BuildConfig.FELDNAME` auf diese Werte zu. Somit wird je nach aktivem Flavor (z. B. `hsluiDebug` oder `hslutaRelease`) automatisch die passende Konfiguration geladen.

Tenant	Base URL	Client Token
HSLU I	<code>https://hslui.mobile-hslu.ch</code>	xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
HSLU TA	<code>https://hsluta.mobile-hslu.ch</code>	xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx

Tabelle 5.1.: Tenant-spezifische BuildConfig-Werte

5.1.2.3. Beispiel aus `build.gradle`

```

1 productFlavors {
2     create("hslui") {
3         dimension = "tenant"
4         applicationIdSuffix = ".hslui"
5         versionNameSuffix = "-HSLUI"
6         buildConfigField("String", "TENANT_ID", "\"hslui\"")
7         buildConfigField("String", "TENANT_NAME", "\"HSLU I\"")
8         buildConfigField("String", "API_BASE_URL", "\"https://hslui.mobile-hslu.ch\"")
9         buildConfigField("String", "CLIENT_TOKEN", "\"xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
10          \"")
11     }
12     create("hsluta") {
13         dimension = "tenant"
14         applicationIdSuffix = ".hsluta"
15         versionNameSuffix = "-HSLUTA"
16         buildConfigField("String", "TENANT_ID", "\"hsluta\"")
17         buildConfigField("String", "TENANT_NAME", "\"HSLU TA\"")
18         buildConfigField("String", "API_BASE_URL", "\"https://hsluta.mobile-hslu.ch\"")
19         buildConfigField("String", "CLIENT_TOKEN", "\"xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
20          \"")
21     }
22 }
```

5.1.2.4. Verwendung im Code

In der `MainActivity` oder anderen Klassen wird direkt auf die generierten Felder zugegriffen, z. B.:

```
1 val tenantId = BuildConfig.TENANT_ID
```

```
2 val tenantName = BuildConfig.TENANT_NAME
3 val tenantDisplayName = BuildConfig.TENANT_DISPLAY_NAME
4 val tenantLogoName = BuildConfig.TENANT_LOGO_NAME
```

5.1.2.5. Vorteile

- Klare Trennung zwischen den Tenants (hslui, hsluta)
- Kein Hardcoding von URLs oder Tokens im Code
- Automatische Tenant-Auswahl über Build Variants
- Einheitliche Nutzung von Feature-Flags für Module (z. B. ENABLE_PARKING, ENABLE_TRAIL)

5.1.2.6. Dateistruktur

```
app/
└── build.gradle
src/
└── hslui/
    ├── java/ch/hslu/i/config/
    └── java/ch/hslu/i/di/
└── hsluta/
    ├── java/ch/hslu/ta/config/
    └── java/ch/hslu/ta/di/
└── main/java/ch/hslu/mobileapps/ui/
    └── MainActivity.kt
```

5.1.2.7. Zusammenfassung

Durch die zentrale Definition aller Tenant-Konfigurationen im `build.gradle` konnte die bisher verstreute Logik vereinheitlicht werden. Die App verwendet beim Build automatisch die jeweils korrekten Werte und Feature-Flags, ohne dass diese im Code hart verdrahtet sind. Dies erhöht die Wartbarkeit, reduziert Fehlerquellen und ermöglicht eine klare Trennung der Umgebungen für HSLU I und HSLU TA.

5.1.3. Fastlane

Die bestehende Android APP, welche nicht auf Jetpackcompose sondern XML basiert, hat bereits für beide Tenants ein Fastfile in welchem diverse Lanes für den CI/CD Prozess definiert sind. Daran möchten wir anknöpfen und die bestehende Funktionalität anpassen und verbessern. Im Moment gibt es pro Tenant ein Fastfile, wenn man diese aber vergleicht, gibt es sehr viele Redundanzen und das möchten wir insofern verbessern, das es nur noch ein Fastfile gibt und per Parameter entweder HSLU_I spezifische oder HSLU_TA spezifische Variablen verwendet werden und so ein Grossteil der Redundanzen eliminiert werden können.

5.1.3.1. Fastfile

Erklärungen Lanes/Funktionen

Parameter definierte Aufrufe

5.2. Common-Komponenten

5.2.1. CommonApplication

todo

5.2.2. CommonDomain

todo

5.2.3. CommonInfrastructure

5.2.3.1. Netzwerk

Die gesamte iOS Netzwerkarchitektur wurde auf Android migriert. Dabei wurde versucht eine identische Architektur beizubehalten und gleiche Methodenamen sowie API-Struktur zu verwenden. Weiter wurden auch die Patterns konsistent über beide Plattformen gehalten. Es wurden moderne Android best Practices verwendet wie Kotlin Coroutines, StateFlow für reaktives UI, Hilt Dependency Injection und eine Type-safe API mit Compile-Time-Checks. Sowohl wurde auch hier der Multitenant Ansatz berücksichtigt und je nach build Variants werden die korrekten URLs und Tokens verwendet, somit konnten Redundanzen verhindert werden.

Architektur Übersicht

Kern-Komponenten

Komponente	iOS	Android
NetzwerkMonitor	NWPathMonitor	ConnectivityManager
CommonNetworkService	async/await	suspend fun
NetworkConfig	struct	data class
UrlConstants	static let	object
Dependency Injection	EnvironmentObject	Hilt
State Management	@Published	StateFlow

Tenant-Konfiguration

Tenant	Base URL	Client Token	Build Variant
HSLU I	hslui.mobile-hslu.ch	xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx	hslui
HSLU TA	hsluta.mobile-hslu.ch	xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx	hsluta

Datei-Struktur

```

Common/Infrastructure/src/main/java/ch/hslu/mobileapps/infrastructure/
  network/
    CommonNetworkService.kt
    NetworkMonitor.kt .3 NetworkConfig.kt .2 config/
    TenantUrlProvider.kt
  di/
    NetworkModule.kt
app/src/hslui/java/ch/hslu/i/
  config/
    UrlConstants.kt
    TenantUrlProviderImpl.kt
  di/
    HsluiNetworkModule.kt
app/src/hsluta/java/ch/hslu/ta/
  config/
    UrlConstants.kt
    TenantUrlProviderImpl.kt
  di/
    HslutaNetworkModule.kt

```

Tests

Unittests

Integrationtests

5.2.3.2. Storage

Todo Sämi

5.2.4. CommonPresentation

5.2.4.1. Lokalisierung

Die Lokalisierung stellt die sprachabhängige Darstellung von Texten und Labels in der App sicher. Es basiert auf dem bestehenden Android-Resource-System und wurde so erweitert, dass dynamisch geladene Inhalte (z. B. Menüpunkte aus dem Bootstrapping) ebenfalls lokalisiert dargestellt werden können.

Neue Umsetzung

Die grundlegende Logik blieb unverändert, da die Mehrsprachigkeit bereits funktionierte. Folgende Anpassungen wurden vorgenommen:

- Sprachwahl erfolgt weiterhin automatisch über die Geräteeinstellung, keine separate App-Sprache erforderlich.
- Ressourcenbasiertes System: Deutsch als Standard in `values/`, Englisch in `values-en/` (appweit und in jedem Feature-Modul).
- Nutzung der UI-Strings erfolgt wie üblich über `stringResource(R.string...)` bzw. `context.getString(...)`.

Übersetzung dynamischer Labels

Für dynamisch geladene Inhalte, etwa Menüpunkte aus Remote-DTOs, wird die Sprache zur Laufzeit anhand der Systemeinstellung bestimmt. Die Zuordnung erfolgt über `Locale.getDefault()`:

- Deutsch (de) → LabelDE / `BootstrapResourceUrlDE`
- Englisch (Standard) → LabelEN / `BootstrapResourceUrlEN`

Ein Beispiel der Umsetzung in Kotlin:

```
1 val Label: String
2     get() = when (Locale.getDefault().language.lowercase(Locale.getDefault())) {
3         "de" -> LabelDE ?: LabelEN ?: "Unbenannt"
4         else -> LabelEN ?: LabelDE ?: "Unnamed"
5     }
```

Beispiel: Ressourcenbasiertes UI-Label

```
1 <!-- values/strings.xml -->
2 <string name="Roomsearch_NavItem">Raumsuche</string>
3 <string name="Settings_NavItem">Einstellungen</string>
4
5 <!-- values-en/strings.xml -->
6 <string name="Roomsearch_NavItem">Roomsearch</string>
7 <string name="Settings_NavItem">Settings</string>
```

Listing 5.1: Default (Deutsch) vs. Englisch in `strings.xml`

Damit ist die App vollständig zweisprachig ausgelegt. Labels aus Remote-Datenquellen werden zur Laufzeit automatisch auf die passende Sprache gemappt, wodurch eine konsistente Benutzererfahrung entsteht.

5.2.5. CommonUtil

todo

5.3. Features

Die Applikation ist konsequent feature-orientiert aufgebaut. Sämtliche fachlichen Funktionalitäten der App sind in eigenständigen Feature-Modulen gekapselt, welche sich im Verzeichnis `/features` befinden. Ein Feature repräsentiert dabei jeweils einen abgegrenzten Anwendungsbereich (und Menüpunkt) der App, wie beispielsweise *Blog*, *News*, *Mensa*, *Events*, *Raumsuche*, *Stundenplan*, *Einstellungen* oder *ApplInit*. Jedes Feature ist als separates Android-Library-Modul umgesetzt und kann unabhängig entwickelt, getestet und gewartet werden. Im Falle unserer Android-App hat auch jedes Feature einen eigenen API-Endpunkt.

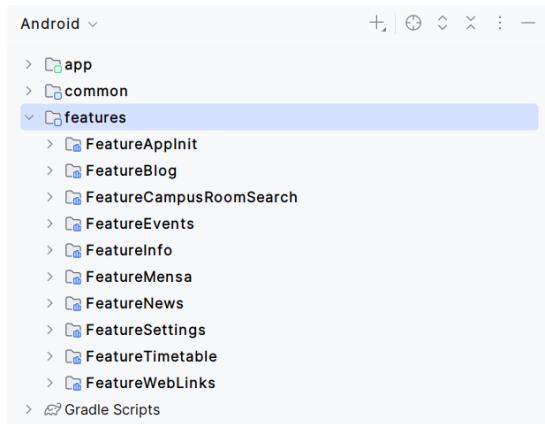


Abbildung 7.: Features in Android-Studio

Ziel und Rolle der Features

Features bilden die funktionale Ebene der Applikation. Sie enthalten alle Bestandteile, die für eine bestimmte Funktionalität erforderlich sind. Durch diese Aufteilung wird sichergestellt, dass einzelne Funktionen klar voneinander getrennt sind und Änderungen möglichst lokal auf ein einzelnes Feature beschränkt bleiben.

Ein zentrales Architekturprinzip ist, dass Features ausschliesslich von den Common-Modulen abhängen dürfen, jedoch nicht voneinander (Einige Ausnahme: *ApplInit*). Dadurch werden zyklische Abhängigkeiten vermieden und die Architektur bleibt langfristig erweiterbar und wartbar.

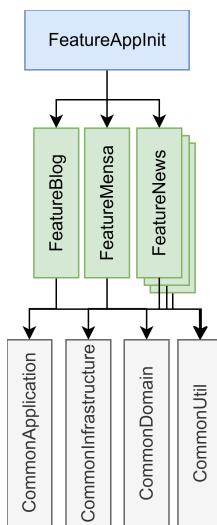


Abbildung 8.: Abhängigkeiten in Pfeilrichtung erlaubt/vorhanden

Aufbau eines Features

Alle Features folgen grundsätzlich der gleichen Struktur. Die Struktur und Benennung der Ordner wurden bewusst an der iOS-App orientiert, sodass Entwickler, die an beiden Projekten arbeiten, die entsprechenden Code-Stellen plattformübergreifend schnell wiederfinden.

Ein Feature gliedert sich typischerweise in folgende Bereiche:

- **Domain:** Enthält die fachlichen Modelle und Domain-Objekte des Features. Diese Schicht ist unabhängig von Android- oder UI-spezifischen Frameworks und bildet die Grundlage für Business-Logik und Datenverarbeitung.
- **Services:** Beinhaltet die Logik zum Laden, Synchronisieren und Verarbeiten von Daten. Hier befinden sich unter anderem Loader-Klassen, welche für den Zugriff auf Backend-APIs, die lokale Speicherung sowie Synchronisationsmechanismen verantwortlich sind.
- **View:** Enthält die UI-Komponenten des Features, umgesetzt mit Jetpack Compose. Die Views reagieren reaktiv auf Zustandsänderungen der ViewModels und stellen die Daten dem Nutzer dar.
- **Resources:** Feature-spezifische Ressourcen wie Strings oder weitere Assets, die unabhängig von anderen Features gepflegt werden können.

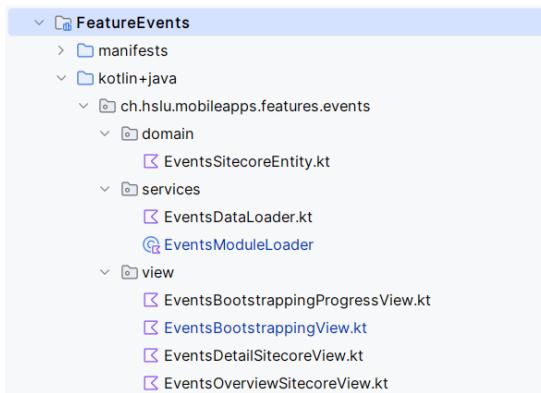


Abbildung 9.: Struktur der Features am Beispiel Event

Das Feature *AppInit* unterscheidet sich dabei grundsätzlich von den übrigen Features, da es nicht eine fachliche Funktionalität bereitstellt, sondern für die initiale Konfiguration, das Bootstrapping und die Steuerung der Applikation verantwortlich ist.

5.3.1. AppInit

Das Feature *AppInit* bildet die Grundlage für das dynamische Laden und Initialisieren der App-Module zur Laufzeit (Bootstrapping). Die Idee ist analog zur iOS-Implementierung: Menüeinträge werden nicht statisch im Code definiert, sondern dynamisch über das Netzwerk geladen, lokal gespeichert und bei Bedarf aktualisiert. So kann beispielsweise ein Menüpunkt remote aktiviert oder deaktiviert werden, ohne dass eine neue App-Version verteilt werden muss.

Ziel und Motivation

Bereits im Vorgängerprojekt (*Android-XML*) existierte ein ähnlicher Mechanismus. In der neuen Version sollte die Funktionsweise aber an die Architektur der iOS-App angepasst und somit überarbeitet werden. Zusätzlich soll die App auch offline funktionieren, indem lokale Daten als Fallback dienen. Ein weiteres Ziel ist die Versionsprüfung, um veraltete App-Versionen zu erkennen und den Nutzer zum Update zu leiten.

Umsetzung / Funktionsweise

Beim Start der App wird zunächst das *AppInit*-Feature geladen, das abhängig von der Konfiguration eine Bootstrapping-Ansicht anzeigt. In der aktuellen Implementierung erfolgt dies über einen einfachen Splashscreen, der den Initialisierungsprozess visuell begleitet. Ein Austauschen dieses Screens wäre aber theoretisch über die Config jederzeit möglich.

Das (*AppBootstrappingViewModel*) übernimmt dann im Hintergrund (während der Ladebildschirm angezeigt wird) den Ablauf des Bootstrappings. Das Bootstrapping besteht aus zwei zentralen Schritten: der Versionsprüfung sowie dem Laden der Features (Menüpunkte).

Versionsprüfung

Während des Semesters wurde auch die iOS-App weiterentwickelt. Ein neues Feature, das beide Plattformen betrifft, ist die versionierte API des Backends, um bei Bedarf mehrere Versionen von Daten parallel anbieten zu können. Aus diesem Grund war es erforderlich, dieses Feature auch in der Android-App umzusetzen, da beide Applikationen auf dasselbe Backend zugreifen.

Der Versionscheck ist im Feature *AppInit* integriert und stellt sicher, dass die verwendete App-Version mit dem Backend kompatibel ist. Der Ablauf des Versionschecks ist wie folgt:

1. **Netzwerkprüfung:** Zuerst wird geprüft, ob eine Netzwerkverbindung besteht. Ohne Verbindung kann die Version nicht zuverlässig bestimmt werden, daher wird `Unable_Determine_Version` zurückgegeben und der Prozess fortgesetzt (tolerant).
2. **Lokale Version extrahieren:** Die lokale API-Version wird aus der *AppBootstrappingConfig* gelesen.
3. **Remote-Version abrufen:** Die App ruft die Backend-Version über `moduleLoaderVersionUrl` ab. Die Antwort ist ein JSON-Objekt mit einem Feld `Item`, das die Versionsnummer enthält (z. B. "V1"). Fehlt die Antwort oder das Feld, wird ebenfalls `Unable_Determine_Version` zurückgegeben.
4. **Versionsvergleich:**

$$\text{VersionStatus} = \begin{cases} \text{Please_Upgrade}, & \text{wenn } \text{lokaleVersion} < \text{remoteVersion} \\ \text{Dont_Upgrade}, & \text{wenn } \text{lokaleVersion} \geq \text{remoteVersion} \end{cases}$$

Das Ergebnis ist ein Versionscheck, der erkennt, wenn die lokale App-Version veraltet ist und ein Update erforderlich wird. Gleichzeitig ist das Feature *tolerant* ausgelegt und versucht, die App dennoch zu laden, falls die Versionsnummer nicht ermittelt werden kann (z. B. bei nicht erreichbarer API). Dadurch wird das System robuster und weniger fehleranfällig gegenüber temporären Backend- oder Netzwerkproblemen.

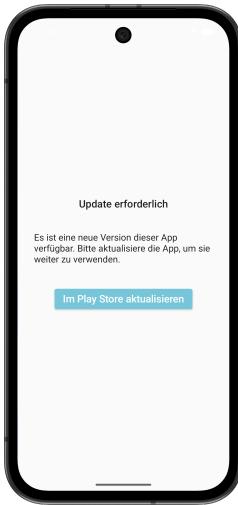


Abbildung 10.: Aufforderung zum Versionsupdate

Laden der Features

Nach erfolgreicher Versionsprüfung erfolgt im zweiten Schritt das Laden der verfügbaren Features (In diesem Fall die verfügbaren Menüpunkte in der App). Dabei werden die Moduldefinitionen über das Backend geladen und mit den lokal gespeicherten Daten synchronisiert.

Dabei wird zuerst geprüft, ob bereits Modul- und Synchronisationsdaten im lokalen Speicher vorhanden sind. Anhand eines Zeitstempels wird entschieden, ob die lokal gespeicherten Daten (Modulliste) noch aktuell sind oder ob ein Update vom Backend erforderlich ist. Liegt eine neuere Version der Modulliste vor, werden die Daten aus dem Netzwerk geladen (und dann lokal wieder persistiert). Falls keine Netzwerkverbindung besteht oder der Abruf fehlschlägt, werden die zuletzt gespeicherten Module als Fallback verwendet. Dadurch bleibt die App auch im Offline-Betrieb funktionsfähig.

$$\text{SyncStatus} = \begin{cases} \text{Load_Remote}, & \text{wenn } \text{lokalerZeitstempel} \neq \text{remoteZeitstempel} \\ \text{Use_Local}, & \text{wenn } \text{lokalerZeitstempel} = \text{remoteZeitstempel} \end{cases}$$

Nach dem Laden werden die Module entsprechend der gespeicherten Reihenfolge sortiert.

Grundsätzlich wäre es möglich, diese Reihenfolge noch dynamisch (also nach Nutzer-Präferenz) anzupassen. Mit dem aktuellen Menükonzept, das auf vier vordefinierten Tab-Gruppen basiert, ergibt dies jedoch wenig Mehrwert und wurde daher bewusst nicht freigeschaltet. Änderungen im Backend oder das Hinzufügen neuer Module können dennoch dazu führen, dass sich die Modulreihenfolge automatisch anpasst.

Abschliessend werden ausschliesslich aktivierte Module berücksichtigt und an die Navigationskomponenten übergeben. Diese bauen daraus zur Laufzeit die sichtbare App-Navigation auf.

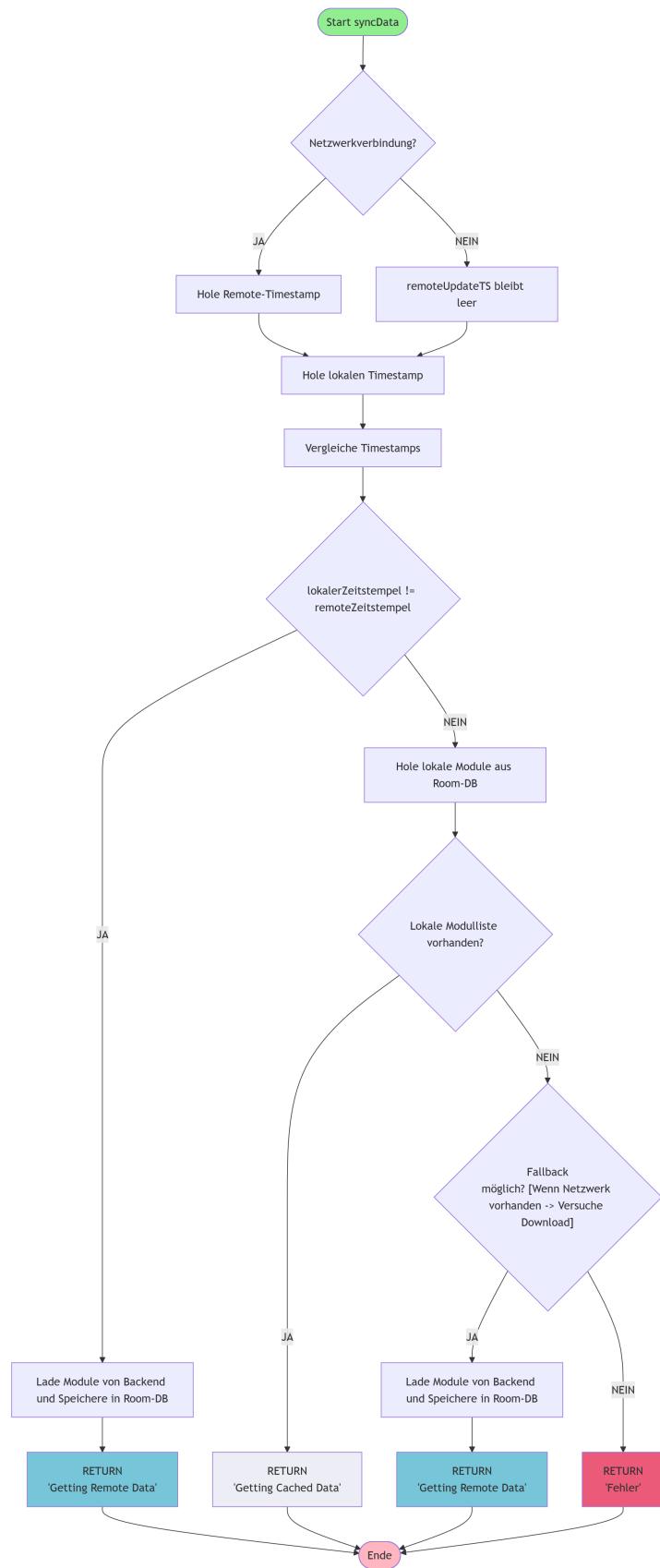


Abbildung 11.: Flussdiagramm Menü-Synchronisation

5.3.2. News

Todo Raphi

5.3.3. Blog

Todo Raphi

5.3.4. Mensa

Todo Raphi

5.3.5. Events

Todo Raphi

5.3.6. Info/About

Todo Raphi

5.3.7. Settings

Todo Raphi

5.3.8. Weblinks

Todo Raphi

5.3.9. CampusRoomSearch

Todo Sämi

5.3.10. Timetable

Todo Raphi

6. Validierung und Evaluation

Beschreibe, wie die Lösung überprüft, getestet oder bewertet wurde.

6.1. Teststrategie – Android Multi-Tenant App (Jetpack Compose)

Todo Raphi: Überarbeitung

6.1.1. Ausgangslage

In der bisherigen mobilen Applikationslandschaft – sowohl bei der iOS-App als auch bei der älteren Android-App auf XML-Basis – war die Testabdeckung sehr gering. Es existierten nur wenige automatisierte Tests, meist oberflächliche oder technische Tests ohne echten Bezug zur Geschäftslogik. Dadurch war die Qualitätssicherung stark manuell geprägt, und Änderungen am Code konnten nur mit hohem Aufwand überprüft werden.

Mit der Neuentwicklung der Android-App auf Basis von Jetpack Compose soll dieser Zustand deutlich verbessert werden. Ziel ist es, eine strukturierte, modulare und wartbare Teststrategie aufzubauen, die langfristig eine hohe Codequalität und Stabilität sicherstellt.

6.1.2. Zielsetzung

- Aufbau einer einheitlichen Testarchitektur über alle Module hinweg
- Erhöhung der Testabdeckung, insbesondere in der Business Logic und bei den ViewModels
- Klare Trennung zwischen Unit-, Integrations- und UI-Tests
- Einführung moderner Testframeworks mit JUnit 5 (Jupiter)

6.1.3. Umsetzung

6.1.3.1. Testarten und Vorgehen

6.1.3.1.1. Unit Tests (src/test/) Ziel: Isolierte Tests einzelner Klassen und Methoden ohne Android-Abhängigkeiten

Frameworks: JUnit 5, Mockito oder MockK

Fokus: Business Logic, Datenmodelle und ViewModels

Vorgehen:

- Jede logische Komponente (z. B. Parser, Berechnungen, ViewModels) erhält eigene Unit Tests
- Abhängigkeiten werden gemockt, um reine Logiktests zu ermöglichen
- Tests folgen dem AAA-Prinzip (Arrange – Act – Assert) und klaren Namenskonventionen

Beispiel:

```
1 @Test
2 fun `should return correct sum when valid input provided`() {
3     val calculator = Calculator()
4     val a = 5
```

```
5     val b = 3
6     val result = calculator.add(a, b)
7     assertEquals(8, result)
8 }
```

6.1.3.1.2. Integration- und UI-Tests (src/androidTest/)

Ziel: Überprüfung des Zusammenspiels mehrerer Komponenten im echten Android-Kontext

Frameworks: JUnit 5, Espresso, Jetpack Compose Testing

Fokus: UI-Verhalten, Datenfluss, Netzwerk- und Datenbankinteraktionen

Vorgehen:

- Tests laufen mit Android Emulator oder Gerät
- Überprüfen reale Benutzerinteraktionen (z. B. Klicks, Eingaben, Navigationsflüsse)
- Fokus auf Haupt-Userflows und kritische Systempfade

6.1.4. Struktur und Modularität

Jedes Modul (z. B. App, Common, Features) verfügt über eigene Testverzeichnisse:

- src/test/ → Unit Tests
- src/androidTest/ → Instrumentierte Integration- und UI-Tests

Diese klare Trennung ermöglicht modulunabhängiges Testen, parallele Ausführung und einfache Erweiterung der Testbasis.

6.1.5. Zielwerte und Qualität

- Unit-Test-Abdeckung: mindestens 80 %
- Integration/UI-Tests: Fokus auf kritische Anwendungsfälle
- CI/CD-Integration: Automatische Testausführung bei jedem Build
- Langfristig: Schrittweise Ausweitung auf alle bestehenden Module

6.1.6. Fazit

Mit der neuen Jetpack-Compose-Architektur wurde eine klare und moderne Teststruktur geschaffen, die auf JUnit 5 basiert und sowohl Unit- als auch Android-Tests konsequent trennt. Damit wird erstmals eine nachhaltige Testbasis gelegt, um Qualität, Stabilität und Wartbarkeit der App langfristig sicherzustellen.

6.2. Evaluation AI Tools

Im Projektauftrag unserer WIPRO wird definiert, dass wir mit dem «AI-First» Ansatz die Android APP angehen sollen. Damit wir das sinnvoll machen können und nicht für jede erdenkliche KI ein Abo abschliessen müssen, möchten wir die geeignete KI evaluieren. Wir haben uns dazu entschlossen die Evaluation zum einen aus wissenschaftlichen Daten aufzubauen und zum anderen das Selbstexperiment mit den Gratisversionen der einzelnen KI's zu machen. Nachdem wir die einzelnen Tools evaluiert haben, werden wir uns für eines entscheiden und da dann auch die kostenpflichtige Version verwenden, um alle Features brauchen zu können und diesen «AI-First» Ansatz auch korrekt umsetzen können.

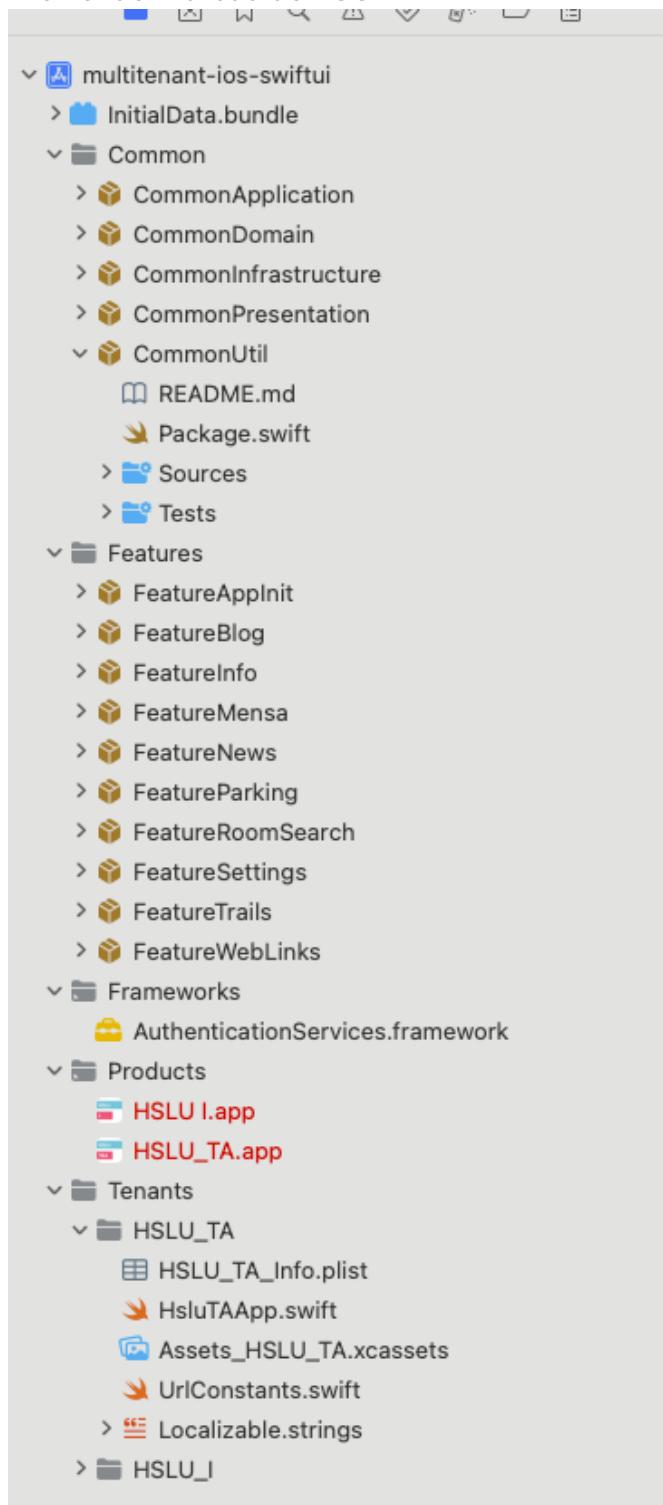
6.2.1. Aufgabenstellung für die AI Tools

Wir haben diversen KI Tools welche wir als gut empfinden, da wir sie bereits verwenden oder viel gutes gehört haben oder aufgrund der wissenschaftlichen Daten, die gleiche Aufgabe gestellt. Nachfolgend die Aufgabenstellung, welche wir den Tools gestellt haben:

Aufgabenstellung

Wir müssen eine bestehende iOS APP, welche in Swift geschrieben ist, nach Android Jetpackcompose erstellen. Dazu soll falls sinnvoll Domain Driven Design verwendet werden.

Hier ist der Aufbau der iOS APP:



Hier ist noch einige codeausschnitte:

```
multitenant-ios-swiftui ) Features > FeatureBlog > Package.swift > No Selection
1 // swift-tools-version: 5.8
2 // The swift-tools-version declares the minimum version of Swift required to build this package.
3
4 import PackageDescription
5
6 let package = Package(
7   name: "FeatureBlog",
8   defaultLocalization: "de",
9   platforms: [.iOS(.v16)],
10  products: [
11    // Products define the executables and libraries a package produces, and make them visible to other packages.
12    .library(
13      name: "FeatureBlog",
14      targets: ["FeatureBlog"]),
15  ],
16  dependencies: [
17    // Dependencies declare other packages that this package depends on.
18    .package(path: "/Common/CommonApplication"),
19    .package(path: "/Common/CommonDomain"),
20    .package(path: "/Common/CommonInfrastructure"),
21    .package(path: "/Common/CommonPresentation"),
22    .package(path: "/Common/CommonUtil")
23  ],
24  targets: [
25    // Targets are the basic building blocks of a package. A target can define a module or a test suite.
26    // Targets can depend on other targets in this package, and on products in packages this package depends on.
27    .target(
28      name: "FeatureBlog",
29      dependencies: ["CommonApplication", "CommonDomain", "CommonInfrastructure", "CommonPresentation", "CommonUtil"]),
30    .testTarget(
31      name: "FeatureBlogTests",
32      dependencies: ["FeatureBlog", "CommonApplication", "CommonDomain", "CommonInfrastructure", "CommonPresentation", "CommonUtil"]),
33  ]
34 )
35
36
```

```
$ multitenant-ios-swiftui ) Common > CommonPresentation > Package.swift > No Selection
1 // swift-tools-version:5.7
2 // The swift-tools-version declares the minimum version of Swift required to build this package.
3 // RANT: Are you serious APPLE? Declaring relevant Meta-Data in Code-Comments? Because Increasing the number above defines the actual version of the used Packagemanager.
4
5 import PackageDescription
6
7 let package = Package(
8   name: "CommonPresentation",
9   defaultLocalization: "de",
10  platforms: [.iOS(.v16)],
11  products: [
12    // Products define the executables and libraries a package produces, and make them visible to other packages.
13    .library(
14      name: "CommonPresentation",
15      //type: .dynamic,
16      targets: ["CommonPresentation"]),
17  ],
18  dependencies: [
19    // Dependencies declare other packages that this package depends on.
20    // .package(url: /* package url */, from: "1.0.0"),
21    .package(path: "/Common/CommonApplication"),
22    // .package(url: "https://github.com/Swinject/Swinject.git", from: "2.8.2"),
23  ],
24  targets: [
25    // Targets are the basic building blocks of a package. A target can define a module or a test suite.
26    // Targets can depend on other targets in this package, and on products in packages this package depends on.
27    .target(
28      name: "CommonPresentation",
29      dependencies: ["CommonApplication"]),
30    .testTarget(
31      name: "CommonPresentationTests",
32      dependencies: ["CommonPresentation", "CommonApplication"]),
33  ]
34 )
35
```

```

// Ressources: FileManger: https://www.appypie.com/filemanager-files-swift-how-to/
// Async-Stuff:
https://stackoverflow.com/questions/71556293/how-can-i-avoid-that-my-swift-async-method-runs-on-the-main-thread-in-swiftui/73015435#73015435

import Foundation
import Network
import SwiftUI

import CommonDomain
import CommonInfrastructure
import CommonUtil
import OSLog

open class CommonApplicationBaseModuleLoader<I:CommonAppBaseModuleItemAPI
DTO, T:CommonAppBaseModuleAPIDT0<I>,
C:CommonModuleBaseLoaderConfig> : ObservableObject
{
    @Published public var loadingStatus :
CommonModuleLoaderStatus<I> = .Not_Initialized()

    public var tenantConfig : AppTenantConfig
= AppTenantConfig()
    public var appConfig : AppModuleLoaderConfigProtocol
= CommonModuleBaseLoaderConfig()
    public var networkService : CommonNetworkService
= CommonNetworkService()
    public var storageService : CommonStorageService
= CommonStorageService()

    var logger = Logger(subsystem: "ch.hslu.mobileapp",
category: "hsldui " + String(describing:
CommonApplicationBaseModuleLoader.self))

    public init() { }

    open func setup(tenantConfig : AppTenantConfig, appConfig: C,
storageService: CommonStorageService, networkService :
CommonNetworkService)
    {
        self.tenantConfig = tenantConfig
        self.appConfig = appConfig
        self.storageService = storageService
        self.networkService = networkService

        self.loadingStatus = .Initialized()
    }
}

```

```

// 1) Does sync of remote / local data
// 2) Does sync of fresh data / with local data, that was
ordered by the user
// The last step may differ in another scenario ->
consider using it via dependency injection?
open func process(implicit:Bool) async -> (Bool, String,
String, [I])? // AppNewsModuleItemAPIDTO
{
    DispatchQueue.main.async { self.loadingStatus =
.Processing(["Syncing ... "]) }

    switch(self.appConfig.moduleLoaderType)
    {
        case "Hybrid":

            let result = await self.syncData()

            if (result.0)
            {
                if !result.3.isEmpty
                {
                    DispatchQueue.main.async {
self.loadingStatus = .Success(result.3) }
                        if(!implicit) { return (true,
result.1, result.2, result.3) }
                }
                else
                {
                    DispatchQueue.main.async {
self.loadingStatus = .ErrorOn(["Data", "No Data"]) }
                        if(!implicit) { return (false, "Data",
"No Data", []) }
                }
            }
            else
            {
                DispatchQueue.main.async {
self.loadingStatus = .ErrorOn([result.1, result.2]) }
                    if(!implicit) { return (false, result.1,
result.2, []) }
            }
        }

        case "Local":

            DispatchQueue.main.async { self.loadingStatus =
.ErrorOn(["", "Not implemented"]) }
                if(!implicit) { return (false, "", "Not
Implemented", []) }

        case "Remote":

```

```

        DispatchQueue.main.async { self.loadingStatus
= .ErrorOn(["", "Not implemented"]) }
            if(!implicit) { return (false, "", "Not
Implemented", []) }

        default :

            DispatchQueue.main.async { self.loadingStatus
= .ErrorOn(["", "Not implemented"]) }
                if(!implicit) { return (false, "", "Not
Implemented", []) }
            }

        return nil
    }

// Does a reset of the module - data-wise
// Deletes everything and re-triggers a process / sync
data of the module
open func reset(implicit : Bool) async -> (Bool, String,
String, [I])? // AppNewsModuleItemAPIDTO
{
    // do such things only when we guarantee network-
access...
    // in order to prevent cumbersome app-behaviour
    let hasConnection = await
self.networkService.hasConnection()
    let remoteHasData = await
self.networkService.isRemoteReachable(url:
self.tenantConfig.monitoringEndpoint)

    if(hasConnection && remoteHasData)
    {
        let syncFileDeleted = await
self.storageService.deleteFile(fileName:
self.appConfig.moduleLoaderSyncLocalFileName)
        let dataFileDeleted = await
self.storageService.deleteFile(fileName:
self.appConfig.moduleLoaderDataLocalFileName)

        if (syncFileDeleted && dataFileDeleted)
        {
            return await self.process(implicit: implicit)
        }
        else
        {
            return (false, "Data", "Unable to delete
Module-File", [])
        }
    }
}

```

```

        else
        {
            if(!remoteHasData)
            {
                return (false, "Network", "Remote Server is
not reachable!", [])
            }
            else
            {
                return (false, "Network", "No Network
connection available!", [])
            }
        }
    }

    // 1. Try to Get Last Update Timestamp
    //      If there is response -> see whether cached file is
available with same timestamp
    //      If there is no response -> see
    //      If there is no Connection -> try to grab local
version
    open func syncData() async -> (Bool, String, String, [I])
// AppNewsModuleItemAPIDTO
{
    var remoteSyncData : (CommonAppBaseModuleSyncAPIDTO?, Data)
    var remoteUpdateTS : String = ""
    var localUpdateTS : String = ""

    var canUpdate = false;
    var updateNeeded = false;
    var storeSyncInfo = false;

    if(await self.networkService.hasConnection())
    {
        // Get Update-Timestamp
        remoteSyncData = await
self.networkService.getAsJsonObject(url:
self.appConfig.moduleLoaderSyncURL) as
(CommonAppBaseModuleSyncAPIDTO?, Data)
        if(remoteSyncData.0 != nil)
        {
            remoteUpdateTS =
remoteSyncData.0!.ModuleLastUpdated
        }
        else
        {
            canUpdate = false; // Remote not available
        }
    }

    // Get the File From Storage ->
}

```

```

        let localSyncData : CommonAppBaseModuleSyncAPIDTO?
= await self.storageService.getFile(fileName:
self.appConfig.moduleLoaderSyncLocalFileName)
        if(localSyncData != nil)
        {
            localUpdateTS =
localSyncData!.ModuleLastUpdated
        }

        if(localUpdateTS == "" && remoteUpdateTS == ""))
{
            canUpdate = false
            updateNeeded = true
}
        if(localUpdateTS == "" && remoteUpdateTS != ""))
{
            canUpdate = true
            updateNeeded = true
            storeSyncInfo = true;
}

        if(localUpdateTS != "" && remoteUpdateTS == ""))
{
            canUpdate = false
            updateNeeded = false // is not decideable
because remote info is missing, that can be compared to local
version
}
        if(localUpdateTS != "" && remoteUpdateTS != ""))
{
            if(localUpdateTS == remoteUpdateTS)
{
                canUpdate = true
                updateNeeded = false
}
            else
{
                canUpdate = true
                updateNeeded = true
                storeSyncInfo = true
}
}

        if storeSyncInfo
{
            let _ = await
self.storageService.storeFile(fileName:
self.appConfig.moduleLoaderSyncLocalFileName, rawData:
remoteSyncData.1)
}

```

```

        }

        if canUpdate && updateNeeded
        {
            // else sync data from remote ...
            let (localTmpPath, _, _) = await
self.networkService.getAsDownload(url:
self.appConfig.moduleLoaderUrl) as (URL?, String, String)
            if localTmpPath != nil
            {
                let _ = await
self.storageService.moveFile(fromFileName: localTmpPath!,
toFileName: self.appConfig.moduleLoaderDataLocalFileName)
                    let remoteData : T? = await
self.storageService.getFile(fileName:
self.appConfig.moduleLoaderDataLocalFileName)

                    // self.loadingStatus = .Success
                    return (true, "Data", "Getting Remote Data",
remoteData!.List!)
            }
            else
            {
                return (false, "Data", "Unable to fetch remote
data", [])
            }
        }
        else
        {
            if !canUpdate && updateNeeded
            {
                return (false, "Network", "Unable to fetch
remote data", [])
            }
            else
            {
                // canUpdate && !updateNeeded
                // !canUpdate && !updateNeeded

                let localData : T? = await
self.storageService.getFile(fileName:
self.appConfig.moduleLoaderDataLocalFileName)
                if localData != nil
                {
                    // self.loadingStatus = .Success_Cached
                    // DispatchQueue.main.async {
self.loadingStatus = .Success_Cached }
                    return (true, "Data", "Getting Cached
Data", localData!.List!)
                }
                else

```

```
        {
            return (false, "Data", "Unable to fetch
local data", [])
        }
    }
}
```

Frage

Wie würdest du nun vorgehen und was ist wichtig zu beachten?

Kannst du die Aufgabe analysieren und in einem Word Dokument zusammenfassen wie wir Vorgehen sollten?

Ziel war es dann, die erhaltenen Antworten zu prüfen und miteinander zu vergleichen um zumindest einen Anhaltspunkt zu erhalten, welches Tools die detaillierteste, aber auch für uns beste Analyse liefert.

6.2.2. Antworten

6.2.2.1. ChatGPT

Die Antwort von ChatGPT zu der Fragenstellung ist sehr gut und in einer vernünftigen Tiefe für eine erste Analyse herausgekommen. Preis: Pro Version 23Euro/month

Nachfolgend die Antwort von ChatGPT:

Analyse der Aufgabenstellung und Vorgehensweise

Die Aufgabenstellung besteht darin, eine bestehende iOS-App, welche in Swift entwickelt wurde, auf Android mit Jetpack Compose zu portieren. Dabei soll – wo sinnvoll – Domain Driven Design (DDD) angewendet werden. Die gezeigten Codeausschnitte geben Einblick in die Struktur der iOS-App.

Analyse der Aufgabenstellung

Wesentliche Punkte der Aufgabenstellung:

- Die iOS-App ist in Swift entwickelt und nutzt verschiedene Common-Komponenten wie Network, Storage, Logging und Konfigurationsobjekte.
- Es gibt ein zentrales Loader-Modul (`CommonApplicationBaseModuleLoader`), das u. a. folgende Aufgaben übernimmt:
 - Initialisierung der Module
 - Synchronisation von Remote- und Local-Data
 - Fehler- und Statusbehandlung (Loading, Success, Error)
 - Reset-Mechanismen (Daten löschen, erneute Synchronisation)
- Der Prozess ist stark asynchron aufgebaut (async/await in Swift).

Empfohlenes Vorgehen bei der Portierung

1. Analyse der bestehenden Architektur:

- Verstehen der Domain-Logik (ModuleLoader, Sync, Reset, Status).
- Dokumentation der bestehenden Schnittstellen und Abhängigkeiten.

2. Übertrag auf Android-Welt:

- Jetpack Compose als UI-Framework.
- Nutzung von Kotlin Coroutines (statt Swift async/await).
- Verwendung von Android-spezifischen Services (z. B. Room für Storage, Retrofit/OkHttp für Network, DataStore für Konfiguration).

3. Domain Driven Design:

- Klare Trennung von Domain, Application, Infrastructure und UI.
- Abbildung der Business-Logik in einer `Domain`-Schicht, die unabhängig von UI und Frameworks bleibt.
- ModuleLoader als zentrales Aggregat in der Domain-Schicht.

4. Synchronisations-Logik portieren:

- Abbildung des Sync/Reset-Verhaltens in Kotlin.
- Behandlung von Edge-Cases (kein Netzwerk, Remote nicht erreichbar, keine lokalen Daten).

5. Architektur auf Android:

- MVVM oder Clean Architecture als Basis.
- ViewModels, die den `CommonApplicationModule`-Ansatz abbilden.
- StateFlows oder LiveData für Statusänderungen ('loadingStatus').

6. Testbarkeit sicherstellen:

- Dependency Injection (z. B. Hilt oder Koin).
- Unit-Tests für Domain-Logik (Sync, Reset, Fehlerbehandlung).
- UI-Tests für Compose-Oberflächen.

7. Wichtige Punkte zu beachten:

- Unterschiedliche Plattformparadigmen (Swift async vs. Kotlin Coroutines).
- Fehler- und Statusbehandlung muss konsistent übertragen werden.
- Möglichst viel Domain-Logik unabhängig von Frameworks halten.
- DDD nur dort anwenden, wo es Mehrwert bringt (nicht alles erzwingen).

Fazit

Die Portierung erfordert eine sorgfältige Analyse der bestehenden Architektur und eine saubere Übertragung in die Android-Welt. Jetpack Compose bietet für die UI eine moderne Grundlage, während Domain Driven Design für eine klare Struktur sorgt. Besonders wichtig ist die saubere Trennung der Schichten, die Wiederverwendbarkeit und die Sicherstellung von Testbarkeit.

6.2.2.2. Grok

Die KI von X hat ebenfalls eine Antwort gegeben, womit diverse gute Ansätze beschrieben sind und man eine gute Übersicht erhält, was gemacht werden muss und wie dies möglich sein kann. Preis: SuperGrok 30/month

Nachfolgend die Antwort von Grok:

****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 : CommonAppBaseModuleItemAPIIDTO> :
ViewModel() {

 val loadingStatus: MutableStateFlow<CommonModuleLoaderStatus<I>> =
MutableStateFlow(CommonModuleLoaderStatus.NotInitialized)

// ... Weitere Properties

suspend fun syncData(): Triple<Boolean, String, List<I>> {

 var remoteSyncData: Pair<CommonAppBaseModuleSyncAPIIDTO?, 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!

### 6.2.2.3. DeepSeek

DeepSeek hat aus meiner Sicht sehr detailliert die Informationen aus der Aufgabenstellung und vor allem den Screenshots und dem Code gezogen. Dementsprechend wurden auch direkt Entwürfe für die Android App generiert und Strategien aufgezeigt. Nebst einer detaillierten Checkliste, was alles erledigt werden muss, gab es auch bereits einige Codebeispiele als Antwort. Preis: Free (bzw. keine Möglichkeit gefunden ein Abo abzuschliessen oder dafür zu zahlen wie bei den anderen)

Nachfolgend die Antwort von DeepSeek:

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)
    ├── FeatureApplInit
    ├── 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?

6.2.2.4. Cursor

Die Antwort von Cursor ist aus meiner Sicht auch korrekt aber nicht gleich detailliert, kann aber denke ich durch gewisses Nachfragen immer noch eine grosse Hilfe für das Projekt sein. Cursor ist nicht die gewöhnliche Web KI, sondern kann als Programm heruntergeladen werden und verwendet werden. Dabei kann ein gesamtes Projekt geöffnet werden und man kann da direkt zu einzelnen Dateien eine Frage stellen. Es gibt auch zum Beispiel ein Add On zu VS-Code. Für Android Studio gibt es ebenfalls Plugins. Cursor ist aber an sich keine KI sondern ein verbessertes Frontend, womit diverse Models geladen werden können. Durch das frontend bietet es aber einem grossen Vorteil zu den direkten Models. siehe auch: <https://cursor.com/docs/models> Preis: Pro Version 20/month

Nachfolgend die Antwort von Cursor:

Zusammenfassung der Analyse

Ich habe die PDF-Aufgabenstellung analysiert und ein detailliertes Dokument erstellt, das folgende Bereiche abdeckt:

1. Code-Analyse der iOS App

- **Architektur-Pattern**: Modulare Struktur mit Domain, Infrastructure und Utility Layers
- **Wichtige Features**: Async/Await, Network Service, Storage Service, Hybrid Data Loading
- **Datenfluss**: Sync-Prozess, Caching, Update-Mechanismus, Error Handling

2. Empfohlene Vorgehensweise

- **Phase 1**: Domain Driven Design Setup mit Kotlin
- **Phase 2**: Infrastructure Layer (Ktor + Room Database)
- **Phase 3**: Presentation Layer (Jetpack Compose)
- **Phase 4**: Dependency Injection (Hilt)

3. Technologie-Stack

- **Core Libraries**: Jetpack Compose, Kotlin Coroutines, Hilt, Room, Ktor
- **Architecture Components**: ViewModel, Repository Pattern, Use Cases, StateFlow

4. Migration-Strategie

- Bottom-Up Approach
- Feature-by-Feature Migration
- Comprehensive Testing Strategy

Erstellte Dateien

1. **`Analyse_iOS_zu_Android_Migration.md`** - Markdown-Version
2. **`Analyse_iOS_zu_Android_Migration.html`** - HTML-Version (kann in Word konvertiert werden)

Nächste Schritte

Das HTML-Dokument können Sie einfach in Microsoft Word öffnen und als .docx speichern. Die Analyse bietet Ihnen:

- **Detaillierte Code-Beispiele** für die Android-Implementierung
- **Schritt-für-Schritt Anleitung** für die Migration
- **Architektur-Empfehlungen** basierend auf Domain Driven Design
- **Technologie-Vergleiche** zwischen iOS und Android

Die Analyse berücksichtigt die spezifischen Anforderungen Ihrer iOS App und bietet eine solide Grundlage für die Android-Entwicklung mit Jetpack Compose.

Tabelle 6.1.: Auswertung der KI in Tabellenform

| KI | Stärken / beobachtete Fähigkeiten | Schwächen / Unsicherheiten | Eignung |
|------------|---|--|---|
| Grok (xAI) | xAI hat mit „grok-code-fast-1“ ein Modell für Entwickler-Aufgaben (Agentic Coding) vorgestellt; Effizienz/Qualität im Fokus; funktioniert auch als Backend in Tools wie Cursor. | Relativ jung; unklare Reife-/Plattformtiefe; unklar, wie gut Apple/Android-SDKs abgedeckt sind. | Spannend mit aktueller Grok-Version; mit guter Prompt-Strategie evtl. starker Konkurrent zu Claude. |
| DeepSeek | Kostengünstige, vergleichbare Alternative; Integration in IDE-Umgebungen möglich. | Weniger öffentlich dokumentierte Benchmarks; evtl. unreifer bei Edge-Cases / Plattformcode. | Gute Ergänzung, v. a. wenn Kosten zählen; für kritische Teile menschliches Review einplanen. |
| ChatGPT | Sehr stark bei Cross-Plattform-Wissen (iOS/Android, SwiftUI/Compose, Flutter, RN); bewährt; gute Architektur- und Testing-Hinweise. | Token-Limits bei sehr großen Codebasen (außer Enterprise/Pro); potenzielle Halluzinationen → Review nötig. | Top-Option neben Claude; stark für Architektur/Best Practices. |
| Cursor | IDE-Frontend, das verschiedene Modelle einbindet; verbessert Workflow und Kontext im Editor. | Keine eigene KI; Leistung hängt vom gewählten Modell ab. | Als Interface/Workflow-Booster zusammen mit starkem Modell nutzen (z. B. Grok/ChatGPT/Claude). |

6.2.3. Vergleich

6.2.4. Detaillierter Fragen zu Cursor

Da wir von Cursor sehr begeistert sind, da es eine Applikation mit eigenem Frontend ist und wir ganze Ordnerstrukturen öffnen können (wie bei Visual Studio Code zum Beispiel), hat die KI direkt einen gesamten Überblick über die Codebasis. Dadurch ist es einfacher an bessere Informationen zu gelangen, da der Kontext einzelner Codestellen für die KI immer klar ist, nicht wie bei herkömmlichen Online-Tools wo nur der wirklich hochgeladene Code vorhanden ist und da teilweise aber auch nicht korrekte Interpretationen gemacht werden.

Nachfolgend einige Screenshots welche an einem Beispiel zeigen wie gut eine Datei analysiert wird, da der Kontext vorhanden ist. Es wurde gefragt, was die Datei "CommonApplicationBaseModuleLoader.swift" macht und wie der Zusammenhang zum Rest des Projektes ist: Hier sieht man sehr gut den Vorteil, da direkt die ganze Projektstruktur analysiert werden kann und so auch die Zusammenhänge besser klar sind für die KI. Zusätzlich werden auch relevante Codestellen hervorgehoben.

6.2.5. Fazit

Nach all den Informationen welche wir gesammelt haben sind wir zum Schluss gekommen, dass wir mit Cursor und einer der oben aufgeführten KIs verwenden wollen. Schlussendlich ist das wichtigste, dass die KI den Kontext des gesamten Projekts hat um möglichst sinnvollen Code zu erstellen und da hat Cursor klar den Vorteil. Die eigentlichen LLMs sind im Grundsatz nicht gross anders. Wir haben beide ein ChatGPT Pro Abo, welches aber für Cursor nicht gilt, da über die OpenAPI die Calls gemacht werden und da nicht dieses Abo zählt. Mit dem Cursor Pro Plan sollte man aber sehr gut fahren, vor allem wenn man den Auto-Modus eingeschaltet hat, da dann für kleinere Abfragen kostengünstige/kostenfreie APIs verwendet werden und nur für grosse Abfragen werden kostenpflichtige APIs aufgerufen, womit der Pro Plan eigentlich gut reichen sollte. Trotzdem kann es sein, gerade wenn mehrere Personen daran arbeiten, dass dieser Plan nicht reicht und man am Ende mehr zahlen muss.

Erklärung der CommonApplicationBaseModuleLoader Datei und Projektzusammenhang

Was ist diese Datei?

Die `CommonApplicationBaseModuleLoader` ist eine generische Basisklasse für das Laden und Synchronisieren von Moduldaten in einer Multi-Tenant iOS-App. Sie implementiert ein einheitliches Pattern für alle Feature-Module (News, Mensa, Parking, etc.).

Architektur und Zusammenhang im Projekt:



Abbildung 12.: Analysebeispiel in Cursor

Kernfunktionalitäten:

1. Generische Typen:

```

CommonApplicationBaseModuleLoader<I, T, C>
// I: Item-DTO (z.B. AppMensaModuleItemAPIDTO)
// T: Module-DTO (z.B. AppMensaModuleAPIDTO)
// C: Config (z.B. FeatureMensaModuleLoaderConfig)
    
```

2. Drei Lade-Modi:

- `.Hybrid`: Intelligente Sync zwischen Remote/Local (Standard)
- `.Local`: Nur lokale Daten
- `.Remote`: Nur Remote-Daten (noch nicht implementiert)

3. Intelligente Synchronisation (`syncData()`):

```

// Vergleicht Timestamps zwischen Remote und Local
// Lädt nur bei Bedarf neue Daten herunter
// Fallback auf lokale Daten bei Netzwerkproblemen
    
```

4. SwiftUI Integration:

```

@Published public var loadingStatus : CommonModuleLoaderStatus<I>
// Reaktive UI-Updates für Loading-States
    
```

Verwendung in konkreten Modulen:

```

// Beispiel: MensaModuleLoader.swift
public class MensaModuleLoader : CommonApplicationBaseModuleLoader<
    AppMensaModuleItemAPIDTO,           // Einzelnes Mensa-Item
    AppMensaModuleAPIDTO,              // Mensa-Modul mit Liste
    FeatureMensaModuleLoaderConfig // Mensa-spezifische Config
> {
    // Erbt alle Funktionalität, keine zusätzliche Implementierung nötig
}
    
```

Abbildung 13.: Analysebeispiel in Cursor

Dependency Injection Pattern:

```
// In HsluTAApp.swift / HsluIApp.swift
menaModuleLoader.setup(
    tenantConfig: appTenantConfig,          // Tenant-spezifische URLs
    appConfig: menaConfig,                  // Modul-spezifische Config
    storageService: storageService,        // File-Management
    networkService: networkService         // HTTP-Client
)
```

Zusammenhang mit CommonModuleLoaderResultModel:

Die zuvor erklärte `CommonModuleLoaderResultModel` wird hier intensiv verwendet:

```
// Rückgabe aller Lade-Operationen
return .success(items, description: "Getting remote data")
return .error("Unable to fetch remote data", source: "Network")
```

Vorteile dieser Architektur:

1. DRY-Prinzip: Einmal implementiert, überall verwendbar
2. Konsistenz: Alle Module verhalten sich identisch
3. Offline-First: Intelligente Caching-Strategie
4. Multi-Tenant: Verschiedene Tenants, gleiche Logik
5. Testbarkeit: Dependency Injection ermöglicht Mocking

Abbildung 14.: Analysebeispiel in Cursor

Typische Nutzung:

```
// In SwiftUI View
@ObservedObject var menaLoader: MensaModuleLoader

// Daten laden
let result = await menaLoader.process(implicit: false)

// UI reagiert automatisch auf loadingStatus-Änderungen
switch menaLoader.loadingStatus {
    case .Processing: // Zeige Loading-Spinner
    case .Success(let items): // Zeige Daten
    case .ErrorOn(let messages): // Zeige Fehler
}
```

Kurz gesagt: Diese Datei ist das Herzstück der Datenarchitektur – sie standardisiert, wie alle Feature-Module ihre Daten laden, cachen und synchronisieren.

Abbildung 15.: Analysebeispiel in Cursor

7. Ausblick

Welche zukünftigen Arbeiten, Verbesserungen oder Erweiterungen sind denkbar? Todo

A. Anhänge

A.1. Zusatzmaterial

A.2. Protokolle und Statusberichte

A.2.1. Kickoff-Meeting (19.09.2025)

Sitzungsprotokoll WIPRO - HSLU Mobile Apps

| | |
|------------|---|
| Datum/Zeit | 19.09.2025, 16:30- 17:30 |
| Ort | Online (Microsoft Teams) |
| Teilnehmer | Auftraggeber: Jürg Nietlisbach
Studenten: Raphael Eiholzer, Samuel Kurmann |
| Protokoll | Raphael Eiholzer |

Traktanden / Diskutierte Punkte

«Führung» durch bestehende Projekte

- Zu Beginn der Sitzung gibt der Auftraggeber einen kurzen Überblick über die bestehende Codebasis
- Bestehendes Android-Projekt existiert auf Basis von XML, ist aber noch nicht an «neues» Backend angebunden
- App auf iOS schon mit kompletter Funktionalität verfügbar, muss jetzt mit gleichen Features auf Android und Jetpack Compose «nachgeholt» werden

Projektfokus AI-First-Ansatz

- AI-Techniken sollen während der Entwicklung genutzt werden
- In einer «Evaluationsphase» wird festgelegt, auf welche AI-Tools während der Arbeit gesetzt werden soll. Diese Tools können auch als «Pro»-Version gekauft werden (z.B., wenn ein gemeinsam genutzter Account verwendet werden kann)
- Die Evaluationsphase soll nicht zwingend einzelne Features zwischen KI-Programmen direkt miteinander vergleichen, sondern eher hervorbringen, wie hilfreich diese Programme beim Entwicklungsprozess sind. Beispielsweise können verschiedene Use-Cases als Markdown an ein KI-Programm gegeben werden (z.B. schreibe mir 20 Unit-Tests). Die Nützlichkeit der einzelnen KI-Programme soll dann anhand dieser Ergebnisse erkennbar werden.
- Raphael Eiholzer stellte die Frage, wie die verschiedenen KI-Programme sinnvoll miteinander verglichen werden können. Da deren Ausgaben nicht deterministisch sind, kann eine Bewertung zwar zu einem Zeitpunkt korrekt sein, kurze Zeit später aber bereits durch andere oder bessere Ergebnisse überholt werden. Daher soll der Fokus nicht auf einer absoluten Bewertung liegen, sondern vielmehr auf der konkreten Nützlichkeit der Programme für den Programmierer im jeweiligen Entwicklungsprozess.

- Der KI-Ansatz wird fortlaufend dokumentiert. Abhängig von den Ergebnissen und den Diskussionen in den kommenden Sitzungen kann die Gewichtung dieses Themas innerhalb des Projekts variieren.

Release

- Der Programmcode wird auf GitLab hochgeladen
- Entweder kann der bestehende Programmcode erweitert werden, es besteht aber auch die Möglichkeit, komplett neu mit dem Projekt zu beginnen, wenn ein Weiterfahren als nicht sinnvoll erachtet wird.
- Die App wird mithilfe von Fastlane im PlayStore veröffentlicht. Hier kann sich am bestehenden XML-File zu Fastlane orientiert werden.

Herausforderungen

- Die Applikation soll für mehrere Mandanten entwickelt werden. Pro Mandat werden unterschiedliche Ressourcen (z.B. Bilder) gebraucht. Hier muss eine saubere Lösung gefunden werden, damit die Ressourcen strukturiert abgelegt werden können.

Projektmanagement

- Es handelt sich um ein Wirtschaftsprojekt der Hochschule Luzern. Deshalb muss das Projektmanagement auch nach HSLU-Standard erfolgen. Es soll schlussendlich für die HSLU ersichtlich sein, dass die Studenten in der Lage sind, ein Softwareprojekt eigenständig zu planen und umzusetzen (Issue-Tracking, Logbuch, Artefakte für Auftraggeber, Code, Dokumentation).
- In der Programmwahl sind die Studenten grundsätzlich frei (Z.B. Jira).
- Die Idee der Studenten ist es, die Projektdokumentation mit LaTeX zu schreiben. Der Stand wird dabei auf GitLab hochgeladen. Für den Dozenten wird daraus ein PDF-Dokument erstellt, um schnell auf die Dokumentation zugreifen zu können.

Nächste Schritte

- Aufgabenstellung abgeben auf Complesis
- Erstellung erster Artefakte (Grober Projektplan, Meilensteine)

Weitere Treffen

- Treffen mindestens alle zwei Wochen (regelmässig), nach Möglichkeit am Donnerstagmorgen oder Freitagnachmittag nach 15:00
- Vor dem Treffen senden die Studenten jeweils einen kurzen Statusbericht an den Auftraggeber (Erledigte Aufgaben, Zwischenstand, Risikobewertung, Nächste Arbeiten)

A.2.2. Projektstatus-Meeting (25.09.2025)

Sitzungsprotokoll WIPRO - HSLU Mobile Apps

| | |
|------------|---|
| Datum/Zeit | 25.09.2025, 09:00- 10:00 |
| Ort | Online (Microsoft Teams) |
| Teilnehmer | Auftraggeber: Jürg Nietlisbach
Studenten: Raphael Eiholzer, Samuel Kurmann |
| Protokoll | Raphael Eiholzer |

Traktanden / Diskutierte Punkte

Besprechung der Arbeiten seit letztem Meeting

- Von den Studenten wurde damit begonnen, die Projekte einzurichten, einen temporären Projektplan zu erstellen und die bestehenden Projektdokumentationen (von früheren WIPROS) durchzulesen.
- Die HSLU-Apps wurden ein erstes Mal heruntergeladen und getestet.

Projektplanung

- Es wurde anschliessend über die bis jetzt erstellte Projektplanung diskutiert.
- Bestehende Projektplanung/Meilensteinplanung ist nicht nach iterativem Vorgehen. Projektplan soll agil/iterativ aufgeteilt werden
- Transparenz: Es muss jederzeit ersichtlich sein, zu welchem Prozentsatz eine Aufgabe abgeschlossen ist
- Die Idee der Studierenden ist es, dies mithilfe der (dann erstellten) Issues auf GitLab zu verwalten. Es ist pro Issue möglich, zu erfassen, wie lange man ungefähr an dessen Umsetzung plant. Anschliessend können dann die effektiv gearbeiteten Stunden eingetragen werden. Also ist insgesamt eine transparente Nachverfolgung von geplantem Aufwand, effektivem Zeitverbrauch und Fortschritt pro Aufgabe möglich.

Projektrisiken

- Projektrisiken sollen ab jetzt gefunden/definiert werden. Dies kann in einer Risikomatrix (Den Studenten von vorherigen Projekten bekannt) z.B. in einem Excel-Dokument ausgearbeitet werden. Die Risiken sollen dann iterativ aktualisiert werden (Nachvollziehbar ablegen.)

Thema DDD

- Das Projekt soll nach DDD erfolgen
- Orientierung an offiziellen Android-Guidelines
- Die Studenten müssen sich zuerst über DDD bei Android informieren

AI-Entwicklung

- Evaluation zunächst in Gratis-Version, ab 10. Oktober mit Pro-Version eines gewünschten Tools
- Ziel: Ermitteln, an welchen Stellen im Softwareentwicklungszyklus (z. B. Testen, Dokumentation) KI sinnvoll unterstützt
- Vorgehen:
 - o Erst einzelne Kandidaten testen / nach Möglichkeit direkt ausschliessen
 - o Dann Evaluation mit Favoriten durchführen
 - o Pro Entwicklungsschritt (Testing, Doku, etc.) prüfen, welchen Mehrwert KI bietet
- Schlussergebnis in dieser Form erwartet: Dokumentation für Leser: „Wo lohnt sich KI am meisten?“ & „Wie muss ich mit KI umgehen, um gute Ergebnisse zu erhalten?“

Projektüberblick

- Projekte sind: android-jetpackcompose-multitenant, android-xml-multitenant, ios-swiftui-multitenant
- Unsere Umsetzung liegt beim Projekt android-jetpackcompose-multitenant, orientiert an bestehender Funktionalität bei ios-swiftui-multitenant
- Backend:
 - o Zugangsdaten/Links werden vom Dozenten im Wiki ergänzt
 - o Grundidee: App fragt beim Bootstrap das Backend ab, welche Module freigeschaltet werden dürfen
 - o Beispiel: „Parking-Feature“ nur bei bestimmten Mandanten (aktuell eingestellt)
- Technische Punkte
 - o Pendant zu PDFKit (iOS) für Android finden und einbinden
 - o Entwicklung soll am Testsystem (Obwohl App auch bei Produktivsystem nur Leserechte hat)

Backlog / erwünschte Features

- Features können stark an Menüpunkten (bei Mobile App / Backend) orientiert werden
- Diese Menüpunkte nehmen die Studenten ins Backlog auf

Von Studenten nächste Vorgesehene Arbeiten

- Überblickgewinnung Projekte, Durchlesen Studentenprojekte, Starten der Evaluation mit AI, Projektplan / Meilensteinplan, Start Dokumentation mit LaTeX, Risikoanalyse, Erfassung des Produktbacklogs und User-Stories
- Nächstes Meeting: 16. Oktober 2025

A.2.3. Projektstatus-Meeting (16.10.2025)

Sitzungsprotokoll WIPRO - HSLU Mobile Apps

| | |
|------------|---|
| Datum/Zeit | 16.10.2025, 10:00- 11:10 |
| Ort | Online (Microsoft Teams) |
| Teilnehmer | Auftraggeber: Jürg Nietlisbach
Studenten: Raphael Eiholzer, Samuel Kurmann |
| Protokoll | Raphael Eiholzer |

Traktanden / Diskutierte Punkte

Allgemeines

- Status-Berichte sollen in Zukunft jeweils direkt als PDF versenden, nicht als Inhalt im E-Mail
- Konkreter Auftrag: Branch: hs25-wipro benennen (Aktuell: nur «wipro»)
- Bootstrapping als "Neues Feature" in Gitlab erfassen (Aktuell: Unterpunkt bei Issue zu Appstruktur)-> gleich umsetzen wie bei iOS

Abgeschlossene Arbeiten

- Erfassung des Projektes in der Multitenant-Projektstruktur
- Definierung der Teststruktur (Unit/Integrationstests) und Erstellung der (leeren) Testklassen
- Lauffähige App mit Menü-Struktur (gleiches Layout wie bei iOS) umgesetzt
- Anpassungen CI/CD, Fastlane

Nächste Arbeiten

- Lokalisierung wie bei iOS (Backend gibt Lokalisierung vor)
- Netzwerk fertig umsetzen
- Error-Seite (1 Generische, kann parametrisiert aufgerufen werden)
- Bootstrapping anschauen wie bei iOS

Risiken

- Technische Überforderung, Arbeit kann nicht fortgeschritten werden
 - o Lösungsansatz: Timeboxed loslegen und bei Nichterfolg Auftraggeber kontaktieren
- Bei der Projektplanung geht ein Feature vergessen und es wird knapp, dieses bis zum Projektende auch noch umzusetzen
 - o Lösungsansatz: Auftraggeber auffordern zu bestätigen, dass alle Features in Projektplanung vorhanden sind

Allgemeine Fragen zum Projekt für die Weiterarbeit

- Layout:
 - o Muss nicht zwingend gleich umgesetzt werden wie Layout bei iOS App
 - o Eventuell kann gleich das Layout aus dem *dev-branch* verwendet werden (4 Bottom-Nav Menüpunkte)
 - o Es kann eine "Separate-Layout-Schiene" für Android gefahren werden, um Android-spezifische Elemente auch nutzen zu können
 - o Grundsätzlich empfohlen: Tab-View wie von "Corina" konzipiert
- Fastfile
 - o Aktuelles Fastfile bei Android-XML ist produktiv (Und kann grundsätzlich so weiterverwendet werden)
 - o Bei Ordnerstruktur zur *Automation*: Bitte 1-zu-1 übernehmen von Ordnerstruktur
 - o Vorschlag, 1 Fastfile (Bis jetzt aus Konsistenzgründen 2) zu verwenden ist OK
 - o Beim Fastfile sollen "Beispielaufrufe" als Kommentar zuoberst ergänzt werden, um die Verwendung als Beispiel zu sehen
- Android Voraussetzungen:
 - o Grundsätzlich keine fixen Vorgaben
 - o Es sollen Geräte unterstützt werden, die (ca.) maximal 3-4 Jährig sind
 - o Es soll noch herausgefunden werden, was die minSDK bedeutet und auf welchen Wert sie sinnigerweise gesetzt werden soll (Aktuell: 26)

A.2.4. Projektstatus-Meeting (30.10.2025)

Sitzungsprotokoll WIPRO - HSLU Mobile Apps

| | |
|------------|---|
| Datum/Zeit | 30.10.2025, 10:00- 11:00 |
| Ort | Online (Microsoft Teams) |
| Teilnehmer | Auftraggeber: Jürg Nietlisbach
Studenten: Raphael Eiholzer, Samuel Kurmann |
| Protokoll | Raphael Eiholzer |

Traktanden / Diskutierte Punkte

Neuerungen bei iOS APP

- Bestehendes Problem bei Tracking-Einstellungen: Aktuell teilweise überlagernde Einstellungen (zu Consent Tracking); Einmal in den App-Einstellungen, in der WebView wird aber trotzdem noch einmal danach gefragt
- Idee: Neu werden News, Blog, Mensa per Rest-API (JSON) geholt
 - o hub.hslu.ch (<https://hub.hslu.ch/>) läuft auf WordPress-> Daher können per WordPress-API die Beiträge (und weiteres) gesammelt werden
 - o Beispiele:
 - <https://hub.hslu.ch/informatik/wp-json/wp/v2/posts>
 - https://hub.hslu.ch/informatik/wp-json/wp/v2/posts?per_page=20
 - o Neuer Code dazu ist/wird in Swift-App aktuell angepasst (Siehe [hier](#))
 - o Neu erscheint im View eine Liste mit Blogs-> Darüber gelangt man zur Detailansicht (bzw. Preview eines einzelnen Artikels)-> Von hier aus kann zum "Weiterlesen" zusätzlich Safari geöffnet werden
- Links:
 - o Hochschulübergreifende News sind hier zu finden: <https://news.hslu.ch/> (nicht mandatenspezifisch)
 - o Events: SiteCore, nicht WordPress:
[https://www.hslu.ch/en/api/eventlist/load/?page=1&per_page=25&filters\[\]=%E2%80%A6](https://www.hslu.ch/en/api/eventlist/load/?page=1&per_page=25&filters[]=%E2%80%A6)
- Bezug zum Android JetpackCompose Projekt:
 - o Es wäre gut, wenn neben WebView diese «JSON»-View auch zusätzlich noch integriert wird
 - o Nach Ressourcentyp (Web oder JSON) wie im Backend angegeben wird, wird in *BootstrappingProgressView* entschieden, welche View lokal genommen wird
- Dazu die konkrete Frage bezüglich Tracking-Einstellungen bei Android (Weswegen die View-Anpassung zum Thema wurde):

Antwort: Tracking-Einstellungen bei Android kaum ein Thema. Aktuell in XML-App auch nicht vorhanden (kann auch bei JetpackCompose ignoriert werden)

Besprechung zum Projektstand

- Lokalisierung:
 - o Bei iOS wird URL (für API) nicht statisch hinterlegt -> Dafür gibt es ein DTO (ApplInitModuleDTO) [Muss im Android-Code noch nachgebessert werden]
 - o Schlussendlich: Modul kennt nur "eine Sprache", beim Eintrittspunkt entscheidet App anhand Gerätesprache. Ziel: Es wird nur eine URL ans Feature weitergeleitet!

Neuerungen allgemein

- Mensa: Gibt eventuell in Zukunft einen REST-API Endpunkt, um Menü zu laden.
 - o <https://app.food2050.ch/de/v2/zfv/hslu,standort-rotkreuz/hslu-iandw>
 - o https://app.food2050.ch/_next/data/vwJquWCORICODjqE7xhAM/de/foodstoffi/foodstoffi/menu/foodstoffi/weekly.json
 - o Dies aber noch nicht fix, in Abklärung mit Mensa-Betreiber
 - o Aktuell reicht es aber, einfach das iFrame des Menüs einzubetten (Aktuell ist ganze Website der Mensa im WebView dargestellt):
<https://app.food2050.ch/de/foodstoffi/foodstoffi/menu/foodstoffi/weekly>
- Raumsuche: Eventuell gibt es in Zukunft eine Umstellung auf PDF (Auf iOS eleganter lösbar mit PDFKit)
 - o Deshalb noch nicht direkt mit Raumsuche-Feature bei Android starten
 - o Es stellt sich noch die Frage, wie das bei Android umzusetzen wäre (Aufgabe: ca. 1 Stunde evaluieren, ob es wie bei iOS ein PDFKit gibt?)

Diverses

- Zu Risiken:
 - o Zeitliche Komponente bei Verwendung von AI-Tools: Wenn Zeit verloren geht beim AI-First-Ansatz (z.B. durch langes Refactoring), nur einen «Time-Boxed» AI First Ansatz verwenden (z.B. nach 1 Stunde abbrechen und manuell implementieren). Aktuell ist aber die Auffassung der Studierenden, dass durch AI-First Zeit eingespart wird
- Zur AI-Evaluation:
 - o Aktuell sind vor allem subjektive Eindrücke bei der Verwendung entstanden. Evtl. Messgrößen definieren und damit testen (z.B. Zeit stoppen? Oder wieviele Refactorings/Reviews es dadurch mehr braucht)
- Priorisierung der nächsten Features:
 - o Am wichtigsten: WebView, danach Raumsuche, Stundenplan (Aktuell: Trais nicht)

A.2.5. Projektstatus-Meeting (13.11.2025)

Sitzungsprotokoll WIPRO - HSLU Mobile Apps

| | |
|------------|---|
| Datum/Zeit | 13.11.2025, 10:00- 11:10 |
| Ort | Online (Microsoft Teams) |
| Teilnehmer | Auftraggeber: Jürg Nietlisbach
Studenten: Raphael Eiholzer, Samuel Kurmann |
| Protokoll | Raphael Eiholzer |

Traktanden / Diskutierte Punkte

Neuerungen bei iOS App (API-Anpassungen)

- Aktuell ist die Bootstrapping-Mechanik noch nicht versioniert.
 - o Muss künftig versioniert werden (z. B. /v1, /v2), mit Rückwärtskompatibilität. Dies ist auf iOS aktuell in Entwicklung, muss auch bei Android umgesetzt werden
- Die App soll eine fix einprogrammierte API-Version enthalten.
- Es wird eine Versionierungs-Route eingeführt:
 - o API-Endpoint: /API/MobileApps/Auth/Version liefert die aktuellste API-Version.
 - o Die App prüft beim Start über ihre Application-Version-Konstante, ob ihre lokale Version älter als die Remote-Version ist.
 - o Ist dies der Fall, muss ein Update-Screen angezeigt werden.
 - o Umsetzung nun auch für Android vorgesehen.
- Backend-Update-Strategie:
- Die Backend-API wird erst dann aktualisiert, wenn die neue App-Version im Store verfügbar ist (damit «ältere» App-Versionen weiterhin lauffähig bleiben).
- Implementierung kann bereits vorbereitet, aber noch nicht getestet werden (QA-Backend noch ohne Implementierung).
 - o -> Ggf. vorerst auskommentieren.
- Die generische Error-View soll um Ansicht „Version-Clash“ erweitert werden (Anzeige: „Neue Version im Store verfügbar“).

Besprechung zum Projektstand

- Konfig-Klassen:
 - o Sind auf Android umgesetzt wie bei iOS mit Konfig-Klassen. Diese führen aber zu viel «Bloat»-Code.
 - o Ggf. prüfen, ob das noch «state-of-the-art» ist

- JSON-View
 - o kann sich an der bestehenden iOS-App orientieren.
 - o **Sitecore**
 - Filter 1645 = Sitecore-Filter für HSLU i.
 - Parameter „datasource“ muss zwingend angegeben werden.

Priorisierung der nächsten Arbeiten (Features)

1. Raumsuche
2. Events
3. Stundenplan
4. (Evtl. Trails später)
5. Nicht umzusetzen: Parkplätze, Testate, Anmeldungen.

Weitere Aufgaben / To-Dos

- Umsetzung der Versionierungslogik vorbereiten
- Review der Konfig-Klassen
- Refactoring der App-Init und UI-Anpassungen starten
- JSON-View gemäss iOS-Vorlage erstellen
- Nächstes Meeting: Präsentation vor Ort

A.2.6. Projektstatus-Meeting (11.11.2025)

Sitzungsprotokoll WIPRO - HSLU Mobile Apps

| | |
|------------|---|
| Datum/Zeit | 13.11.2025, 10:00- 11:30 |
| Ort | Online (Microsoft Teams) |
| Teilnehmer | Auftraggeber: Jürg Nietlisbach
Studenten: Raphael Eiholzer, Samuel Kurmann |
| Protokoll | Raphael Eiholzer |

Traktanden / Diskutierte Punkte

Besprechung zum Projektstand / weiteres Vorgehen

- Bezuglich Versionskontrolle: Der Link, der zur App im Play Store führt, wenn die App veraltet ist – Link zum Play Store in der TenantConfig setzen, dann kann die App diesen immer finden. Nachteil: Wenn sich der Link ändert, müsste die App neu kompiliert werden. Aktuell aber kein Thema, auch bei iOS nicht.
- Enddatum im Kalender-Widget muss noch eingetragen werden.
- Todo-Widget Theme: Da das Widget alleine ohne die App laufen muss, sind App-Konfigurationen wie Farben/Theme nicht vorhanden. Diese im Widget hart zu codieren wäre unschön. Hier muss noch eine gemeinsame Lösung gefunden werden.
- PDF-Box bei CampusRoomSearch verwenden, wenn Android-native schwierig ist und nicht funktioniert.
- Bericht: Individuelles Fazit jeweils notieren, wäre interessant für den Auftraggeber.

A.2.7. Status-Report (16.10.2025)

Die seit letztem Meeting durchgeführten Arbeiten umfassen (Sprint 2 abgeschlossen):

- Projektmanagement
 - Erfassung der Arbeitsschritte als Issues in GitLab + Zeitplanung
 - Evaluation AI-Tools mit mehreren «Test-Kandidaten»
 - Hier konkret: Wir konnten die besten Ergebnisse mit «Cursor» erzielen. Dazu sonst mehr am Donnerstag.
- Projektfortschritt (Programmierung)
 - Erfassung des Projektes in der Multitenant-Projektstruktur
 - Definierung der Teststruktur (Unit/Integrationstests) und Erstellung der (leeren) Testklassen
 - Lauffähige App mit Menü-Struktur (gleiches Layout wie bei iOS) umgesetzt
 - Anpassungen CI/CD, Fastlane
- Dokumentation
 - Start der Dokumentation, Übernahme der HSLU-Struktur und Layout
 - Dokumentation zu Projektplanung und Risikoanalyse
 - Dokumentation zu bisher geschriebenem Programmcode (z.B. Teststruktur)

Und dies sind unsere nächsten Arbeitsschritte:

Wir «verlassen» hiermit die «kleine Evaluationsphase» und kennen uns nun schon etwas mit dem Projekt aus.

Das nächste Vorgehen wäre die Umsetzung diverser «Common»-Features, also den Features, die von der App generell (oder in diversen Modulen) gebraucht werden.

- Diverse «Common»-Features abschliessen / programmieren
 - Lokalisierung
 - Netzwerkdienst
 - Error-Bildschirme
- Abschluss CI/CD, Fastlane
- Dokumentation der ausgeführten Arbeitsschritte

Die nächsten Arbeitsschritte sind demnach Unter «Sprint 3» im Sprintboard zu sehen.

Die drei grössten Risiken zu diesem Zeitpunkt haben wir hier zusammengetragen:

| Risiko | Massnahme(n) |
|---|---|
| Fachliche Herausforderung, fehlendes Know-How | Wissensaustausch im Team fördern; Rechtzeitig erkennen und um Hilfe bitten |
| Schwierigkeiten bei der Einbringung der AI-Tools in die Entwicklung | Frühzeitige Evaluierung der Tools mit Testszenarien; Dokumentierung dieser Ergebnisse |
| Unrealistische Zeitplanung (Aktuell noch etwas schwierig den Aufwand einzelner Arbeitsschritte abzuwägen) | Aufwandsschätzung durch beide Teammitglieder; Pufferzeiten einplanen; regelmässige Überprüfung der Meilensteine |

A.2.8. Status-Report (30.10.2025)

Die seit letztem Meeting durchgeführten Arbeiten umfassen (Sprint 3 abgeschlossen):

- Projektmanagement
 - Aktualisierung der Projektrisiken dieses Sprints
 - Laufend: Aktualisierung der Tickets und des Backlogs, Zeiterfassung per Ticket
 - Entwicklungs-Branch: Leider konnte nicht wie besprochen der Branch von *wipro* auf *wipro-hs25* umbenannt werden. (Evtl. fehlende Berechtigung?)
- Projektfortschritt (Programmierung)
 - Bootstrapping-Mechanik analog iOS umgesetzt (Stichwort: *FeatureApplInit*)
 - Menüpunkte werden über Netzwerk abgeglichen, zwischengespeichert, angezeigt (wenn freigeschaltet)
 - *CommonFeatures*
 - Netzwerk-Dienst erneuert und verwendet
 - Storage-Dienst erneuert
 - Lokalisierung angeschaut und angewendet
 - Grundsätzlich nicht allzu viel angepasst, funktionierte grundlegend schon. Gerätesprache entscheidet «lokal», welche Strings (XML) verwendet werden.
 - Bootstrapping entscheidet anhand Gerätesprache jetzt, welche URLs geladen werden (z.B. für Mensa)
 - Generischer Error-Bildschirm erstellt, der von überall verwendet werden kann, implementiert
 - *SDK*:
 - *MinSDK* auf 30 gesetzt, *CompileSDK* auf 36. (Dies sollte wie letztes Mal besprochen, 3-4 jährige Geräte einschliessen, ist aber auch ein guter Kompromiss zu neuen Features.)
- Dokumentation
 - Umgesetzte *Common-Features* dokumentiert
 - Sprint war sehr «programmierlastig», Dokumentation muss jetzt noch etwas nachgeführt werden (Teilweise nur Notizen erstellt, muss noch «reingeschrieben» werden).

Und dies sind unsere nächsten Arbeitsschritte:

Wir denken, die wichtigsten Grundlagen des Programmes jetzt umgesetzt zu haben oder werden es in den nächsten Tagen noch tun (*CommonFeatures*). Das nächste Vorgehen wäre die Umsetzung erster Features. Mit der Umsetzung hätten wir dann auch gleichzeitig direkt getestet, ob die *CommonFeatures* auch richtig funktionieren (z.B. Netzwerkdienst beim Laden einer Website).

- Diverse Features, die jetzt begonnen werden:
 - WebView für die Webansicht folgender Module:
 - Mensa
 - News
 - Blog
 - App-Einstellungen
 - Evtl. Start Raumsuche (technisch vermutlich anspruchsvoller als andere Module)
- Dokumentation der ausgeführten Arbeitsschritte

Die drei grössten Risiken zu diesem Zeitpunkt haben wir hier zusammengetragen:

| Risiko | Massnahme(n) |
|--|--|
| Schlechte Codequalität durch Verwendung von AI-Tools | Ergebnisse von AI-generiertem Code stets manuell überprüfen und in Code-Reviews besprechen. |
| Mangelhafte oder unvollständige Dokumentation | Dokumentation fortlaufend während der Entwicklung ergänzen, nicht erst nach Abschluss von Sprints. Einführung kurzer wöchentlicher Doku-Reviews im Team. |
| App nicht lauffähig auf vielen Geräten (mangelnde Testabdeckung) | Tests zusätzlich auf physischen Geräten und unterschiedlicher Android-Versionen durchführen. |

A.2.9. Status-Report (13.11.2025)

Die seit letztem Meeting durchgeführten Arbeiten umfassen (Sprint 4 abgeschlossen):

- Projektmanagement
 - Aktualisierung der Projektrisiken dieses Sprints
 - Laufend: Aktualisierung der Tickets und des Backlogs, Zeiterfassung pro Ticket
 - Diverse Teambesprechungen zu Arbeitsfortschritt, Aufteilung und Vorgehen
- Projektfortschritt (Programmierung)
 - Bootstrapping-Mechanik: Verbessert, neuer *CommonApplicationBaseModuleLoader* stellt zentrale Dienste wie Storage oder Netzwerk (als *Singelton*) jetzt für alle Features bereit (analog iOS)
 - Module bleiben «im Hintergrund geöffnet», damit z.B. Blog nicht beim Modul-Wechsel neu geladen werden muss
 - In *HsluApplication/HsluTAAplication* können jetzt Konfigurationen pro Tenant für App-Features gesetzt werden. (Ähnlich zur *HsluApp.swift* bei iOS)
 - *CommonFeatures*
 - Storage-Dienst: Umgesetzt mit Funktionen, Moduldaten in RoomDB zu speichern
 - *TopNav* durch *TabbedDropdownNavigationView* mit den Tabs *News*, *Campus*, *Studies* und *Settings* ersetzt
 - «App»-Features:
 - *Blog*, *Mensa* und *News* mit generischer *WebView* (gemeinsam genutzt) umgesetzt, Funktionalität in eigentlichen Features dadurch nur noch sehr gering
 - *Links*, *Einstellungen* und *Info/About* mit generischer *MultiListView* umgesetzt
- Dokumentation
 - Programmierfortschritt festgehalten
 - Einige Erkenntnisse zum Arbeiten nach AI-First festgehalten, müssen noch im Team besprochen und sauber dokumentiert werden.

Und dies sind unsere nächsten Arbeitsschritte:

Wir haben in den letzten zwei Wochen viel an den Features programmiert und möchten noch etwas Zeit investieren für Refactoring (Da wir die App vom XML-Projekt übernommen haben, kann wohl noch etwas alter Code bereinigt werden) und uns diesmal etwas mehr auf die Dokumentation kümmern.

Bezüglich der App-Funktionen wären folgende Punkte noch offen:

- Roomsearch: Dieses Feature möchten wir gerne umsetzen, beim nächsten Meeting klären wir am besten wie (HTML oder PDF).
- Events, Parking, Stundenplan, Testate, Anmeldungen, Trails. Diese sind zwar (teilweise) bei iOS umgesetzt, aber wie wir gesehen haben, noch nicht veröffentlicht. Welche dieser Funktionen müssen umgesetzt werden, wo liegt die Priorität?
- Letztes Mal wurde besprochen, für Web-Features auch eine JSON-View zu erstellen. Wir haben dazu auch den neuen Code bei iOS gesehen (und getestet). Die Umsetzung bei Android ist noch nicht gemacht, ist aber schon angedacht («Verzweigung» besteht, View muss noch erstellt werden).
- Auf iOS gibt es Widgets haben wir noch gesehen. (Wird das auch auf Android benötigt?)

Die drei grössten Risiken zu diesem Zeitpunkt haben wir hier zusammengetragen:

| Risiko | Massnahme(n) |
|---|--|
| Zeitverzug durch unerwartete technische Schulden (zum Beispiel durch Altcode aus XML-Projekt) | Zeitpuffer für Aufräumarbeiten im Sprint 5 einplanen. |
| Geplante Features (Roomsearch, JSON-View, etc.) nicht rechtzeitig spezifiziert (z.B. Spezifizierung, ob PDF oder HTML-Ansicht) | Beim nächsten Meeting die Fragen zu den Features klären und festhalten. Feature-Beschreibung dann im Backlog ergänzen. |
| Inkompatibilitäten zwischen Android-Versionen / Geräten: Es ist schwierig auf verschiedenen Geräten produktiv zu testen (vor allem wenn noch weiterentwickelt wird). Vor allem auf diversen Hardwaregeräten zu testen schwierig, da keine Geräte bereitstehen.
Views nur auf Handy-Größen angepasst (Wie sieht's aus auf Tablets?) | <ul style="list-style-type: none"> - In Android Studio gezielt Emulatoren für verschiedene Auflösungen und Android-Versionen anlegen - Views vor allem mit flexiblen Layout (z.B. Flexbox) anlegen - Nach physischen Testgeräten fragen |

A.2.10. Status-Report (11.12.2025)

Die seit letztem Meeting durchgeführten Arbeiten umfassen (Sprint 5&6 abgeschlossen):

- Projektmanagement
 - Aktualisierung der Projektrisiken dieses Sprints
 - Laufend: Aktualisierung der Tickets und des Backlogs, Zeiterfassung pro Ticket
 - Diverse Teambesprechungen zu Arbeitsfortschritt, Aufteilung und Vorgehen
- Projektfortschritt (Programmierung)
 - API-Versionierung: Möglichkeit, die API in verschiedenen Versionen aufzurufen, lokal ein «Update-Bildschirm», falls App veraltet ist. (Umsetzung jetzt auch analog iOS)
 - «App»-Features:
 - *Blog, Mensa, News* jetzt aktualisiert, dass sie eine JSON-View unterstützen (je nach Einstellung im Backend) entweder für WordPress oder Sitecore
 - *Events*: Umsetzung komplettes Feature, ähnlich zu *Blog, Mensa* oder *News*
 - Komplette Umsetzung vom Feature *Stundenplan*, inklusive dessen Widget
 - Implementierung *RoomSearch* mit PDF
 - Diverse kleinere/grössere Refactorings, um die Codequalität zu verbessern (Teilweise duplizierte Codeabschnitte, oder eher «unschöne» Codestellen von AI generiert)
 - Testing auf physischen Endgeräten, Codeverbesserungen dadurch implementiert
 - Diverse kleinere Layout-Anpassungen/-Verbesserungen (+ aktuell Deaktivierung Dark-Mode, da nicht komplett umgesetzt)
- Dokumentation
 - Programmierfortschritt festgehalten
 - Dokumentation teilweise neu strukturiert, Inhaltsverzeichnis sollte sich ab jetzt nicht mehr gross ändern
 - Vorbereitung und Halten von Zwischenpräsentation in SW11

Und dies sind unsere nächsten Arbeitsschritte:

- Die App steht im Grundsatz und sollte an Funktionalität nicht mehr gross erweitert werden. Lediglich beim Feature RoomSearch müssen wir noch einige Tests am physischen Gerät machen. Leider stehen uns hier auch nicht so viele Test-PDFs bereit. Aber wir denken, dass das Feature generell läuft.
- Fastlane-Files müssen noch getestet werden.
- Die nächsten beiden Wochen werden wir uns aber sicher noch hauptsächlich um die Dokumentation kümmern. Hier haben wir noch einige komplett Tage Zeit, die wir investieren möchten, daher sollte es zeitlich gut passen.
- Zwischen der Abgabe der Dokumentation und der Abschlusspräsentation werden wir noch ein paar Refactorings im Code durchführen. Die Funktionalität wollen wir nicht mehr verändern. Doch sind uns (auch im Zusammenhang mit AI-Coding) noch ein paar Unschönheiten aufgefallen, die wir noch verbessern möchten.

Die drei grössten Risiken zu diesem Zeitpunkt haben wir hier zusammengetragen:

| Risiko | Massnahme(n) |
|---|--|
| Trotz eingeplanter Zeit besteht die Gefahr, dass Inhalte inkonsistent, zu wenig tief oder nicht vollständig gemäss HSLU-Vorgaben ausgearbeitet sind. | <ul style="list-style-type: none"> • Klare Aufteilung der Kapitel • Regelmässige interne Reviews • Abgleich mit Bewertungsraster und früheren Beispielen |
| Versteckte Probleme durch spätes Refactoring:
Refactorings kurz vor Abschluss können unbeabsichtigt bestehende Funktionalität beeinflussen oder neue Bugs einführen. | <ul style="list-style-type: none"> • Nur gezielte, klar abgegrenzte Refactorings • Keine funktionalen Änderungen • Kurze Regressionstests nach jedem grösseren Eingriff |
| Unentdeckte Plattform- oder Geräteabhängigkeiten: Bestimmte Probleme treten nur auf einzelnen Android-Versionen oder Gerätetypen auf und bleiben unbemerkt. | <ul style="list-style-type: none"> • Nur beschränkt beeinflussbar, Zahl der Testgeräte ist beschränkt • Eventuell hat HSLU noch weitere Geräte |

Abkürzungs-, Abbildungs-, Tabellen-, KI-, Formelverzeichnis

Abkürzungsverzeichnis

Abbildungsverzeichnis

| | | |
|-----|--|----|
| 1. | Darstellung des Scrum-Prozesses im Projekt | 5 |
| 2. | Meilensteinplan des Projekts | 5 |
| 3. | Beispielhafte Darstellung des Backlogs in GitLab | 6 |
| 4. | Beispiel einer User Story mit Definition of Done (DoD) | 6 |
| 5. | Risikoanalyse und Risikomatrix des Projekts | 7 |
| 6. | Projektstruktur im Android Studio | 8 |
| 7. | Features in Android-Studio | 20 |
| 8. | Abhängigkeiten in Pfeilrichtung erlaubt/vorhanden | 20 |
| 9. | Struktur der Features am Beispiel <i>Event</i> | 21 |
| 10. | Aufforderung zum Versionsupdate | 23 |
| 11. | Flussdiagramm Menü-Synchronisation | 24 |
| 12. | Analysebeispiel in Cursor | 64 |
| 13. | Analysebeispiel in Cursor | 64 |
| 14. | Analysebeispiel in Cursor | 65 |
| 15. | Analysebeispiel in Cursor | 65 |

Tabellenverzeichnis

KI-Verzeichnis

Formelverzeichnis

Literaturverzeichnis