

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

12. 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 selbstständig 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:
Eingangsvisum durch das Sekretariat auszufüllen

Rotkreuz, den _____ Visum: _____

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	14
5.2.1. CommonApplication	14
5.2.2. CommonDomain	16
5.2.3. CommonInfrastructure	18
5.2.4. CommonPresentation	22
5.2.5. CommonUtil	25
5.3. Features	27
5.3.1. AppInit	29
5.3.2. News	32
5.3.3. Blog	34
5.3.4. Mensa	35
5.3.5. Events	36
5.3.6. Info/About	37
5.3.7. Settings	38
5.3.8. Weblinks	39
5.3.9. CampusRoomSearch	40
5.3.10. Timetable	43
6. Validierung und Evaluation	47
6.1. Teststrategie – Android Multi-Tenant App (Jetpack Compose)	47
6.1.1. Ausgangslage	47
6.1.2. Zielsetzung	47
6.1.3. Umsetzung	47
6.1.4. Struktur und Modularität	48
6.1.5. Zielwerte und Qualität	48

6.1.6. Fazit	48
6.2. Evaluation AI Tools	49
6.2.1. Aufgabenstellung für die AI Tools	49
6.2.2. Auswertung	49
6.2.3. Vergleich	50
6.2.4. Detaillierter Fragen zu Cursor	51
6.2.5. Fazit	54
7. Ausblick	55
A. Anhänge	56
A.1. Zusatzmaterial	56
A.2. Protokolle und Statusberichte	56
A.2.1. Kickoff-Meeting (19.09.2025)	57
A.2.2. Projektstatus-Meeting (25.09.2025)	59
A.2.3. Projektstatus-Meeting (16.10.2025)	61
A.2.4. Projektstatus-Meeting (30.10.2025)	63
A.2.5. Projektstatus-Meeting (13.11.2025)	65
A.2.6. Projektstatus-Meeting (11.11.2025)	67
A.2.7. Status-Report (16.10.2025)	68
A.2.8. Status-Report (30.10.2025)	69
A.2.9. Status-Report (13.11.2025)	71
A.2.10. Status-Report (11.12.2025)	73
A.2.11. Aufgabenstellung für AI Tools	75
A.2.12. Antwort von ChatGPT	84
A.2.13. Antwort von Grok	86
A.2.14. Antwort von Deepseek	91
A.2.15. Antwort von Cursor	96
Abkürzungs-, Abbildungs-, Tabellen-, KI-, Formelverzeichnis	97

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.

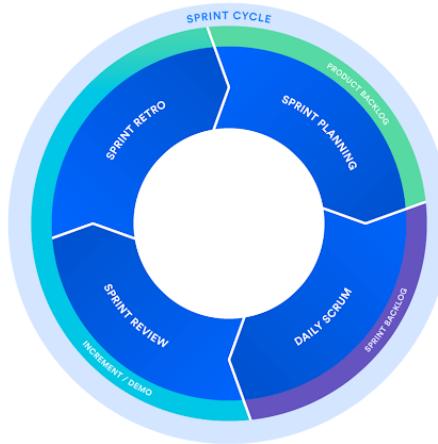


Abbildung 1.: Darstellung des Scrum-Prozesses im Projekt
(atlassian_atlassian_nodate)

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.

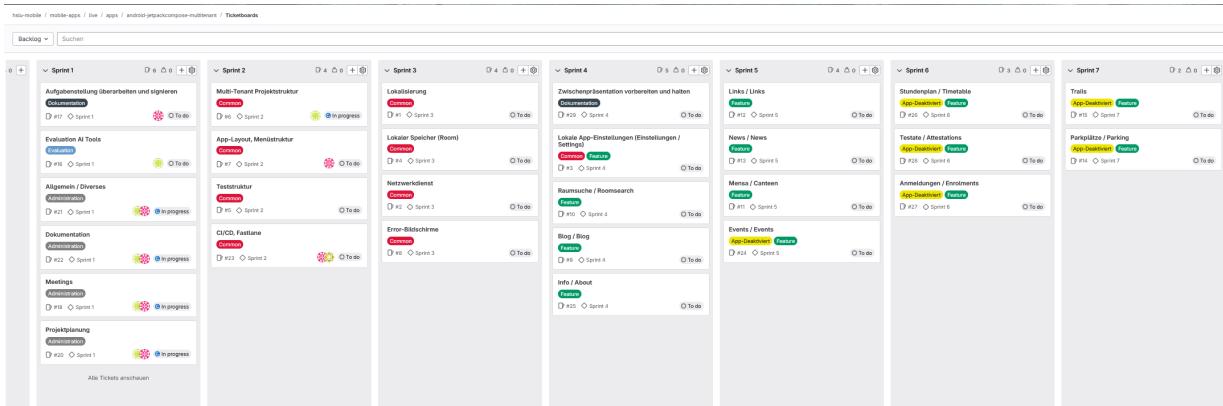


Abbildung 3.: Beispielhafte Darstellung des Backlogs in GitLab

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

Bearbeiten ⋮

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

0 0 0

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.

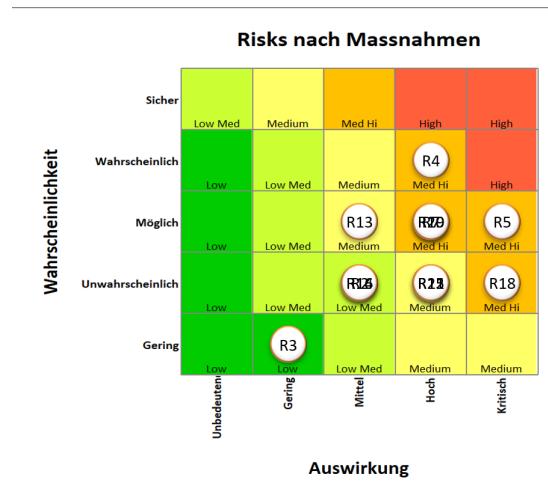


Abbildung 5.: Risikoanalyse und Risikomatrix des Projekts

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.

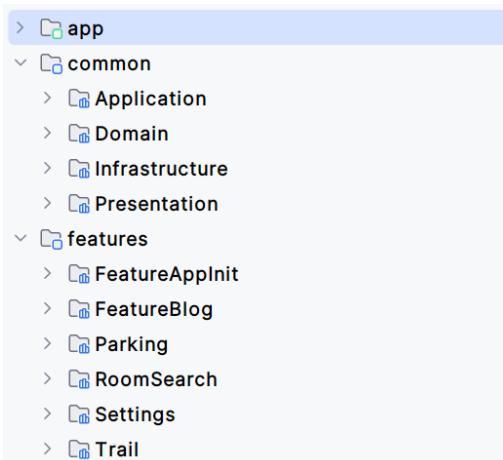


Abbildung 6.: Projektstruktur im Android Studio

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_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	https://hslui.mobile-hslu.ch	xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
HSLU TA	https://hsluta.mobile-hslu.ch	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. Dateistuktur

```
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

Das Fastlane-Setup für die Android-App automatisiert den Build- und Deployment-Prozess für beide Tenants (HSLU I und HSLU TA). Anstatt separate Fastfiles pro Tenant zu verwenden wie das im Android-XML Projekt der Fall ist, wurde ein einheitliches, parametrisiertes Fastfile implementiert, das Redundanzen eliminiert und die Wartbarkeit verbessert. Das Fastfile unterstützt Debug- und Release-Builds sowie automatische Uploads zu Google Play Store in verschiedenen Tracks (Beta, Internal, Production).

Ziel und Motivation

Das Hauptziel besteht darin, die CI/CD-Pipeline zu vereinfachen und Redundanzen zu eliminieren. Durch die Parametrisierung mit dem `tenant`-Parameter kann ein einziges Fastfile für beide Tenants verwendet werden, was die Wartbarkeit erheblich verbessert. Zusätzlich werden alle sensiblen Daten (Keystores, API-Keys) sicher in einem separaten Git-Repository gespeichert und verschlüsselt, sodass sie nicht im Haupt-Repository liegen.

Ein weiteres wichtiges Ziel ist die Automatisierung des gesamten Build- und Deployment-Prozesses, von der Kompilierung über die Signierung bis hin zum Upload in den Google Play Store. Dies reduziert manuelle Fehler und beschleunigt den Release-Prozess erheblich.

Umsetzung / Funktionsweise

Die Implementierung basiert auf einem zentralen `before_all`-Block, der die Initialisierung und Konfiguration für alle Lanes übernimmt. Das Fastfile verwendet Dotenv für die Verwaltung von Secrets und parametrisiert alle tenant-spezifischen Werte. Der `before_all`-Block wird vor jeder Lane ausgeführt und übernimmt die Validierung des `tenant`-Parameters, die Definition von Pfaden sowie das Mapping von Tenant-Namen zu Android-Modulen und Package-IDs.

Die tenant-spezifischen Umgebungsvariablen (Keystore-Passwörter, API-Keys, etc.) werden aus `.env.secret` geladen und in generische Variablen gemappt, sodass alle Lanes die gleichen Variablennamen verwenden können.

Lanes Das Fastfile definiert vier Haupt-Lanes, die in der folgenden Tabelle beschrieben sind:

Lane	Beschreibung
<code>build_debug</code>	Erstellt einen Debug-Build ohne Signierung. Führt <code>gradle clean assemble</code> für das entsprechende Tenant-Modul aus und kopiert die Artefakte in den Output-Ordner. Wird hauptsächlich für lokale Tests verwendet.
<code>build_release</code>	Erstellt einen signierten Release-Build. Lädt den verschlüsselten Keystore aus dem Zertifikats-Repository, entschlüsselt ihn, erstellt die <code>keystore.properties</code> -Datei für Gradle und führt <code>gradle clean bundle</code> aus. Das signierte AAB wird in den Output-Ordner kopiert.
<code>beta</code>	Baut ein signiertes Release-Bundle und lädt es in den Google Play Store hoch. Standardmäßig wird der Beta-Track verwendet, kann aber über den <code>track</code> -Parameter überschrieben werden (z. B. <code>track:internal</code>). Lädt sowohl Keystore als auch Google Play API-Key aus dem Zertifikats-Repository. Ruft intern <code>build_release</code> auf und lädt anschließend das AAB hoch.
<code>release</code>	Analog zu <code>beta</code> , lädt jedoch explizit in den Production-Track des Google Play Stores. Wird für finale Releases verwendet.

Tabelle 5.2.: Übersicht der Fastlane-Lanes

Hilfsfunktionen Das Fastfile verwendet mehrere Hilfsfunktionen zur Verwaltung von Zertifikaten und Konfiguration:

- `download_from_certs_repo`: Klont das Zertifikats-Repository und entschlüsselt eine Datei (typischerweise den Keystore) mit OpenSSL (AES-256-CBC).

- `download_from_certs_repo_full`: Erweiterte Variante, die sowohl Keystore als auch Google Play API-Key entschlüsselt.
- `create_keystore_properties`: Erstellt die `keystore.properties`-Datei dynamisch, die von Gradle für die Signierung verwendet wird.
- `clean_directory`: Entfernt temporäre Zertifikatsdateien nach dem Build, um Sicherheitsrisiken zu minimieren.

Verwendung und Sicherheit

Die Verwendung erfolgt durch Aufruf von Fastlane mit dem `tenant`-Parameter: `fastlane <lane> tenant:hslui` oder `fastlane <lane> tenant:hsluta`. Ohne diesen Parameter bricht Fastlane mit einer Fehlermeldung ab.

Alle sensiblen Daten (Keystores, API-Keys) werden in einem separaten Git-Repository gespeichert und mit OpenSSL verschlüsselt (AES-256-CBC). Die Passwörter für die Entschlüsselung werden aus `.env.secret` geladen, das nicht im Repository gespeichert wird, sondern von der CI/CD-Pipeline zur Laufzeit erstellt wird. Die Keystore-Properties-Datei wird dynamisch erstellt und nach dem Build gelöscht, um Sicherheitsrisiken zu minimieren.

Die Upload-Funktionalität unterstützt verschiedene Tracks im Google Play Store. Der Standard-Track für `beta` ist "beta", kann aber über den `track`-Parameter überschrieben werden (z. B. `track:internal`). Changelogs werden bewusst übersprungen, da diese manuell im Google Play Console verwaltet werden.

Das Fastfile ist erweiterbar für weitere Tenants, indem einfach ein neuer `when`-Fall im `before_all`-Block hinzugefügt wird und die entsprechenden Umgebungsvariablen in `.env.secret` definiert werden.

5.2. Common-Komponenten

5.2.1. CommonApplication

Das Common Application Modul stellt die zentrale Anwendungsschicht für die modulare Multi-Tenant Android-Anwendung dar. Es implementiert eine generische, wiederverwendbare Architektur für das Laden, Synchronisieren und Verwalten von Feature-Modulen über verschiedene Datenquellen hinweg. Die Kernkomponente ist die abstrakte Basisklasse `CommonApplicationBaseModuleLoader`, die von allen Feature-Modulen erweitert wird.

Ziel und Motivation

Das Hauptziel besteht darin, Code-Duplikation zu vermeiden und eine konsistente Datenlade- und Synchronisationslogik über alle Feature-Module hinweg zu gewährleisten. Durch eine abstrakte Basisklasse, die den gesamten Datenlade- und Synchronisationsprozess kapselt, müssen konkrete Feature-Module nur noch ihre spezifischen Typparameter und Konfigurationen bereitstellen. Dies reduziert den Implementierungsaufwand erheblich und stellt sicher, dass alle Module die gleichen Qualitätsstandards in Bezug auf Fehlerbehandlung, Offline-Funktionalität und Performance-Optimierungen einhalten.

Ein weiteres wichtiges Ziel ist die Implementierung einer intelligenten Synchronisationsstrategie, die sowohl Bandbreite als auch Batterie schont. Durch den Vergleich von lokalen und remote Timestamps wird nur dann eine Aktualisierung durchgeführt, wenn tatsächlich neue Daten verfügbar sind, was eine Offline-First-Architektur ermöglicht.

Umsetzung / Funktionsweise

Die Umsetzung basiert auf dem Template-Method-Pattern mit generischen Typparametern. Die Basisklasse verwendet drei Typparameter: I für den Item-Typ, T für den Container-Typ und C für den Konfigurationstyp. Die Klassendefinition sieht folgendermassen aus:

```
1 abstract class CommonApplicationBaseModuleLoader<I, T, C>(
2     // ...
3 ) : ViewModel(), ModuleLoaderContract<I>
```

Der Loader verwaltet seinen Zustand über einen `StateFlow<CommonModuleLoaderStatus<I>>`, der verschiedene Zustände unterstützt: `Not_Initialized`, `Initialized`, `Processing`, `Success`, `Success_Cached` und `ErrorOn`. Die `process()`-Methode orchestriert den gesamten Datenladevorgang und wählt basierend auf dem `moduleLoaderType` die entsprechende Strategie. Die Methode prüft zunächst die Netzwerkverbindung und wählt dann zwischen Remote-, Cache- oder Hybrid-Modus.

Der Hybrid-Modus implementiert eine intelligente Synchronisationslogik, die auf Timestamp-Vergleichen basiert. Zuerst wird über die Sync-URL ein JSON-Objekt abgerufen, das das Feld `ModuleLastUpdated` enthält. Gleichzeitig wird versucht, den lokalen Timestamp aus der Room-Datenbank zu laden. Die Entscheidungslogik vergleicht diese beiden Timestamps:

```
1 if (remoteTimestamp > localTimestamp) {
2     // Daten aktualisieren
3 } else {
4     // Lokale Daten verwenden
5 }
```

Wenn eine Aktualisierung notwendig ist, werden die Daten heruntergeladen und mit Gson deserialisiert. Dabei wird ein `TypeToken` verwendet, um die korrekte Generik-Auflösung zur Laufzeit zu gewährleisten:

```
1 val type = object : TypeToken<T>() {}.type
2 val data = gson.fromJson<T>(jsonString, type)
```

Nach erfolgreichem Parsing wird überprüft, ob die Daten gespeichert werden sollen. Die `shouldStoreData()`-Methode kann von Subklassen überschrieben werden, um bestimmte Datentypen nicht zu speichern. Wenn die Daten gespeichert werden sollen, werden sie zusammen mit dem `lastUpdated`-Timestamp als `ModuleDataEntity` in der Room-Datenbank gespeichert.

Weitere Informationen

Aspekt	Beschreibung
Dependency Injection	Die Integration mit Feature-Modulen erfolgt über Dependency Injection mit Dagger Hilt. Ein konkretes Feature-Modul wie <code>MensaModuleLoader</code> erbt von <code>CommonApplicationBaseModuleLoader</code> und spezifiziert die Typparameter I, T und C. Die Klasse wird mit der <code>@HiltViewModel</code> -Annotation versehen und erbt von der Basisklasse.
UI-Integration	In <code>ViewModels</code> oder <code>Composables</code> kann der Loader direkt verwendet werden. Der <code>loadingStatus</code> wird als <code>StateFlow</code> bereitgestellt und kann mit <code>collectAsState()</code> beobachtet werden, um reaktive UI-Updates zu ermöglichen. Die <code>process()</code> -Methode wird typischerweise in einem <code>LaunchedEffect</code> aufgerufen.
Asynchrone Operationen	Alle asynchronen Operationen werden über Kotlin Coroutines abgewickelt. Der Loader erbt von <code>ViewModel</code> , was automatisch einen <code>viewModelScope</code> bereitstellt, der alle Coroutines automatisch abbricht, wenn das <code>ViewModel</code> zerstört wird. Netzwerk- und Speicheroperationen werden explizit auf <code>Dispatchers.IO</code> ausgeführt, um den Main-Thread nicht zu blockieren.
Fehlerbehandlung	Die Fehlerbehandlung ist mehrschichtig implementiert: <ul style="list-style-type: none"> Netzwerk-Fehler werden durch <code>networkService.hasConnection()</code> und <code>isRemoteReachable()</code> erkannt Parse-Fehler werden durch try-catch-Blöcke bei der Gson-Deserialisierung abgefangen Speicher-Fehler werden bei Room-Datenbankoperationen abgefangen und geloggt
Performance-Optimierungen	Performance-Optimierungen umfassen Lazy Loading, Caching in der Room-Datenbank und inkrementelle Updates, wobei nur bei geänderten Timestamps Daten neu geladen werden. Die Offline-First-Strategie stellt sicher, dass lokale Daten Priorität haben, wenn keine Netzwerkverbindung verfügbar ist.

Tabelle 5.3.: Technische Aspekte des `CommonApplicationBaseModuleLoader`

5.2.2. CommonDomain

Das Common Domain Module stellt die Domänenschicht der Anwendung dar und enthält alle zentralen Geschäftsobjekte, Data Transfer Objects (DTOs), Enums und Konfigurationsklassen, die die Kernentitäten der Anwendung repräsentieren. Es handelt sich um ein reines Domänenmodul ohne Abhängigkeiten zu Android-Frameworks, das den Prinzipien der Clean Architecture folgt und eine klare Trennung zwischen Domänenlogik und Infrastruktur ermöglicht.

Ziel und Motivation

Das Hauptziel des Common Domain Modules besteht darin, eine zentrale Domänenschicht zu schaffen, die von allen anderen Modulen verwendet werden kann, ohne direkte Abhängigkeiten zu Infrastruktur- oder Anwendungsschichten zu haben. Dies ermöglicht eine saubere Architektur, bei der Geschäftsobjekte unabhängig von ihrer konkreten Implementierung definiert werden können. Durch die Verwendung von DTOs für die API-Kommunikation wird eine klare Schnittstelle zwischen der Anwendung und externen Datenquellen geschaffen.

Ein weiteres wichtiges Ziel ist die Vermeidung von Code-Duplikation durch die Bereitstellung von Basisklassen wie `CommonAppBaseModuleItemAPIDTO` und `CommonAppBaseModuleAPI`, die von allen Feature-spezifischen DTOs erweitert werden können. Dies stellt sicher, dass alle Module eine konsistente Datenstruktur verwenden und gemeinsame Funktionalität zentralisiert wird.

Umsetzung / Funktionsweise

Die Struktur des Moduls basiert auf einer klaren Trennung in verschiedene Pakete: `dto` für Data Transfer Objects, `enums` für Aufzählungstypen, `config` für Konfigurationsklassen, `loader` für Loader-Interfaces und `protocols` für Protokoll-Definitionen.

Die Basis-DTOs definieren die gemeinsame Struktur für alle Modul-DTOs. `CommonAppBaseModuleItemAPIDTO` stellt die Basis für einzelne Modul-Items dar:

```
1 abstract class CommonAppBaseModuleItemAPIDTO {
2     abstract val id: String
3     abstract val title: String
4 }
```

`CommonAppBaseModuleAPIDTO` ist ein generischer Container für Modul-Daten, der sowohl einzelne Items als auch Listen unterstützt:

```
1 data class CommonAppBaseModuleAPIDTO<I>(
2     val items: List<I>? = null,
3     val item: I? = null,
4     val ModuleLastUpdated: String? = null
5 )
```

Feature-spezifische DTOs erben von diesen Basisklassen. Beispielsweise erbt `AppMensaModuleItemAPIDTO` von `CommonAppBaseModuleItemAPIDTO` und fügt mensa-spezifische Felder hinzu. Die Verwendung von Gson-Annotationen wie `@SerializedName` ermöglicht die korrekte Deserialisierung von JSON-Daten aus der API.

Die Enum-Klassen definieren verschiedene Zustände und Typen der Anwendung. `CommonModuleLoaderStatus` ist eine sealed class, die alle möglichen Zustände eines Modul-Loaders repräsentiert:

```
1 sealed class CommonModuleLoaderStatus<I> {
2     object Not_Initialized : CommonModuleLoaderStatus<Nothing>()
3     data class Success<I>(val data: I) : CommonModuleLoaderStatus<I>()
4     // ...
5 }
```

`CommonAppModuleType` definiert alle verfügbaren Modultypen der Anwendung und bietet eine `parse()`-Methode zur Konvertierung von Strings.

Die Konfigurationsklassen definieren die Einstellungen für verschiedene Module. `CommonModuleBaseLoaderConfig` stellt die Basis-Konfiguration für Modul-Loader bereit.

Das `ModuleLoaderContract` Interface definiert den Vertrag für alle Modul-Loader und stellt die grundlegenden Methoden wie `setup()` und `process()` bereit.

Weitere Informationen

Aspekt	Beschreibung
Abhängigkeiten	Das Modul hat minimale Abhängigkeiten: <code>androidx.core:core-ktx</code> für Android Core Utilities, <code>com.google.code.gson:gson</code> für JSON-Serialisierung und <code>kotlinx-coroutines-core</code> für Coroutines-Unterstützung. Dies stellt sicher, dass das Modul keine Android-spezifischen Abhängigkeiten hat und theoretisch auch in anderen Kontexten verwendet werden könnte.
Gson-Annotationen	Die DTOs verwenden Gson-Annotationen für die Serialisierung und Deserialisierung. Die <code>@SerializedName</code> -Annotation ermöglicht es, JSON-Feldnamen zu mappen, die nicht den Kotlin-Namenskonventionen entsprechen. Dies ist besonders wichtig bei der Integration mit externen APIs, die möglicherweise andere Namenskonventionen verwenden.
Computed Properties	Einige DTOs wie <code>AppInitModuleItemAPIDTO</code> enthalten computed properties, die basierend auf der aktuellen Locale die richtige Sprache auswählen. Die Implementierung verwendet <code>Locale.getDefault().language</code> und einen when-Ausdruck, um zwischen deutschen und englischen Titeln zu wählen.
Sealed Classes	Die Verwendung von sealed classes für Status-Enums ermöglicht exhaustive when-Ausdrücke in Kotlin, was die Typsicherheit erhöht und Compiler-Warnungen bei fehlenden Fällen auslöst. Dies verbessert die Code-Qualität und reduziert potenzielle Laufzeitfehler.
Clean Architecture	Das Modul folgt den Prinzipien der Clean Architecture, indem es keine Abhängigkeiten zu anderen Schichten hat. Alle anderen Module können das Domain-Modul verwenden, aber das Domain-Modul selbst hängt nur von Standard-Kotlin- und minimalen Android-Bibliotheken ab. Dies ermöglicht eine einfache Testbarkeit und Wartbarkeit des Codes.

Tabelle 5.4.: Technische Aspekte des CommonDomain Moduls

5.2.3. CommonInfrastructure

5.2.3.1. CommonNetwork

Das Common Network Modul stellt die zentrale Netzwerkschicht der Anwendung dar und bietet eine einheitliche Schnittstelle für alle Netzwerkoperationen. Es abstrahiert die Komplexität der Netzwerkommunikation und bietet typsichere Methoden für HTTP-Requests, JSON-Parsing und Datei-Downloads. Die gesamte iOS Netzwerkarchitektur wurde auf Android migriert, wobei eine identische Architektur beibehalten wurde und gleiche Methodennamen sowie API-Struktur verwendet werden. 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. Der Multitenant-Ansatz wurde berücksichtigt, sodass je nach Build-Varianten die korrekten URLs und Tokens verwendet werden, wodurch Redundanzen verhindert werden konnten.

Ziel und Motivation

Das Hauptziel besteht darin, eine zentrale, wiederverwendbare Netzwerkschicht zu schaffen, die von allen Feature-Modulen verwendet werden kann, ohne dass jedes Modul seine eigene Netzwerk-Implementierung benötigt. Durch die Verwendung von Kotlin Coroutines und suspend-Funktionen wird eine moderne, asynchrone API bereitgestellt, die den Main-Thread nicht blockiert und eine reaktive Programmierung ermöglicht.

Ein weiteres wichtiges Ziel ist die Konsistenz zwischen iOS und Android. Die Architektur wurde von iOS migriert, wobei versucht wurde, eine identische Struktur beizubehalten und gleiche Methodennamen sowie API-Struktur zu verwenden. Dies erleichtert die Wartung und das Verständnis des Codes über beide Plattformen hinweg.

Die Verwendung von Hilt für Dependency Injection ermöglicht eine saubere Trennung von Konfiguration und Implementierung, während der Multitenant-Ansatz sicherstellt, dass verschiedene Build-Varianten (hslui, hsluta) automatisch die korrekten URLs und Tokens verwenden.

Umsetzung / Funktionsweise

Die Implementierung basiert auf Android's HTTP-Client-Bibliotheken und Kotlin Coroutines. Der CommonNetworkService ist die zentrale Service-Klasse, die alle Netzwerkoperationen kapselt:

```

1 class CommonNetworkService(
2     private val httpClient: OkHttpClient,
3     private val tenantUrlProvider: TenantUrlProvider
4 ) {
5     suspend fun getAsJsonObject(url: String): JsonObject?
6     // ...
7 }
```

Die Architektur folgt einem klaren Komponentenmodell, das die iOS-Architektur widerspiegelt. Die folgende Tabelle zeigt die Entsprechungen zwischen iOS- und Android-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

Tabelle 5.5.: Entsprechungen zwischen iOS- und Android-Komponenten

Der NetworkMonitor überwacht die Netzwerkverbindung und stellt Informationen über die aktuelle Verbindungs-

qualität bereit. Die Implementierung verwendet Android's **ConnectivityManager** für die Erkennung von Netzwerkänderungen.

Die Tenant-Konfiguration ermöglicht es, verschiedene Build-Varianten mit unterschiedlichen URLs und Tokens zu verwenden:

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

Tabelle 5.6.: Tenant-Konfiguration für verschiedene Build-Varianten

Weitere Informationen

Aspekt	Beschreibung
Asynchrone Operationen	Alle Netzwerkoperationen werden asynchron über Kotlin Coroutines ausgeführt. Die Verwendung von suspend-Funktionen stellt sicher, dass alle Operationen nicht-blockierend sind und den Main-Thread nicht beeinträchtigen. Dies ermöglicht eine reaktive Programmierung und verbessert die Performance der Anwendung.
Type-Safe API	Die API ist type-safe mit Compile-Time-Checks. Methoden wie <code>getAsJsonObject()</code> , <code>getAsJsonString()</code> und <code>getAsDownload()</code> bieten typsichere Rückgabewerte und reduzieren Laufzeitfehler.
Multitenant-Unterstützung	Der Multitenant-Ansatz wird durch <code>TenantUrlProvider</code> und tenant-spezifische Implementierungen unterstützt. Je nach Build-Variant werden automatisch die korrekten URLs und Tokens verwendet, wodurch Redundanzen verhindert werden.
Dependency Injection	Das Modul verwendet Hilt für Dependency Injection. Tenant-spezifische Module (<code>HsluiNetworkModule</code> , <code>HslutaNetworkModule</code>) stellen die entsprechenden Konfigurationen bereit. Dies ermöglicht eine saubere Trennung von Konfiguration und Implementierung.
Netzwerk-Monitoring	Der <code>NetworkMonitor</code> überwacht kontinuierlich die Netzwerkverbindung und stellt Informationen über die Verbindungsqualität bereit. Dies ermöglicht es der Anwendung, auf Netzwerkänderungen zu reagieren und entsprechende Strategien zu implementieren.
iOS-Kompatibilität	Die Architektur wurde von iOS migriert, wobei versucht wurde, eine identische Struktur beizubehalten. Gleiche Methodennamen und API-Struktur erleichtern die Wartung und das Verständnis des Codes über beide Plattformen hinweg.
StateFlow für reaktives UI	StateFlow wird für reaktives State-Management verwendet, ähnlich wie <code>@Published</code> in SwiftUI. Dies ermöglicht es, UI-Komponenten reaktiv auf Netzwerkstatusänderungen zu reagieren.

Tabelle 5.7.: Technische Aspekte des CommonNetwork Moduls

5.2.3.2. CommonStorage

Das Common Storage Modul stellt die zentrale Speicherschicht der Anwendung dar und bietet eine einheitliche Schnittstelle für persistente Datenspeicherung. Es verwendet Android's Room-Datenbank als Backend und abstrahiert alle Datenbankoperationen hinter einem einfachen Service-Interface. Der `CommonStorageService` ermöglicht es, verschiedene Arten von Daten zu speichern: AppInitModule-Konfigurationen, Modul-JSON-Daten, Synchronisations-Timestamps und Dateien.

Ziel und Motivation

Das Hauptziel besteht darin, eine zentrale, wiederverwendbare Speicherlösung zu schaffen, die von allen Feature-Modulen verwendet werden kann, ohne dass jedes Modul seine eigene Datenbank-Implementierung benötigt. Durch die Verwendung von Room als Backend wird eine robuste, typsichere und performante Datenbankzugriffsschicht bereitgestellt, die automatisch SQL-Abfragen generiert und Compile-Zeit-Validierung bietet.

Ein weiteres wichtiges Ziel ist die Abstraktion der Datenbankkomplexität. Anstatt dass jedes Modul direkt mit Room-Entities und DAOs arbeitet, bietet der `CommonStorageService` eine einfache, suspend-Funktion-basierte API, die DTOs aus der Domäenschicht verwendet. Dies ermöglicht es, die Datenbank-Implementierung zu ändern, ohne dass die Anwendungsschicht betroffen ist.

Die Verwendung von Kotlin Coroutines für alle Datenbankoperationen stellt sicher, dass Blockierungen des Main-Threads vermieden werden und die Anwendung reaktionsfähig bleibt. Alle Operationen werden automatisch auf `Dispatchers.IO` ausgeführt, was eine optimale Performance gewährleistet.

Umsetzung / Funktionsweise

Die Implementierung basiert auf Android's Room-Persistenzbibliothek. Die `AppDatabase` ist die zentrale Datenbankklasse, die alle Entities und DAOs verwaltet:

```

1 @Database(
2     entities = [ModuleDataEntity::class, FileEntity::class, ...],
3     version = 1
4 )
5 abstract class AppDatabase : RoomDatabase() {
6     abstract fun moduleDataDao(): ModuleDataDao
7     // ...
8 }
```

Der `CommonStorageService` verwendet das Singleton-Pattern für die Datenbankinstanz und bietet eine Factory-Methode zur Erstellung.

Für die Speicherung von Modul-Daten wird die `ModuleDataEntity` verwendet, die JSON-Daten als String speichert:

```

1 @Entity(tableName = "module_data")
2 data class ModuleDataEntity(
3     @PrimaryKey val moduleId: String,
4     val jsonData: String,
5     val lastUpdated: String
6 )
```

Die `storeModuleData()`-Methode konvertiert DTOs in Entities und speichert sie in der Datenbank.

Die `getModuleData()`-Methode lädt Daten aus der Datenbank und gibt sie als JSON-String zurück.

Für AppInitModule-Daten werden DTOs in Entities konvertiert. Die `storeAppInitModules()`-Methode mappt eine Liste von DTOs zu Entities.

Die `getAppInitModuleOrder()`-Methode demonstriert eine komplexere Abfrage, die Daten aus mehreren Ta-

bellen kombiniert.

Für Dateispeicherung wird die `FileEntity` verwendet, die Dateien als `ByteArray` speichert:

```

1 @Entity(tableName = "files")
2 data class FileEntity(
3     @PrimaryKey val fileId: String,
4     val data: ByteArray,
5     val mimeType: String
6 )

```

Weitere Informationen

Aspekt	Beschreibung
Asynchrone Operationen	Alle Datenbankoperationen werden asynchron über Kotlin Coroutines ausgeführt. Die Verwendung von <code>withContext(Dispatchers.IO)</code> stellt sicher, dass alle Operationen auf dem IO-Dispatcher laufen und den Main-Thread nicht blockieren. Dies ist besonders wichtig für grössere Datenmengen oder komplexe Abfragen.
Fehlerbehandlung	Die Fehlerbehandlung erfolgt durch try-catch-Blöcke, die Fehler loggen und <code>null</code> oder <code>false</code> zurückgeben, anstatt Exceptions zu werfen. Dies ermöglicht es den aufrufenden Komponenten, elegant mit Fehlern umzugehen, ohne dass die gesamte Anwendung abstürzt.
Singleton-Pattern	Die Datenbank verwendet das Singleton-Pattern mit thread-sicherer Initialisierung. Die <code>getDatabase()</code> -Methode verwendet <code>synchronized</code> , um sicherzustellen, dass nur eine Instanz der Datenbank erstellt wird, auch bei gleichzeitigen Zugriffen von mehreren Threads.
Datenbank-Migrationen	Die Verwendung von <code>fallbackToDestructiveMigration(true)</code> bedeutet, dass bei Schema-Änderungen die Datenbank neu erstellt wird. Dies ist für Entwicklung geeignet, sollte aber in Produktion durch Migrationen ersetzt werden.
DAOs	Die DAOs verwenden Room-Annotationen für typsichere SQL-Abfragen. Beispielsweise verwendet <code>ModuleDataDao @Query</code> -Annotationen für benutzerdefinierte Abfragen. Die DAO-Interfaces werden mit <code>@Dao</code> annotiert und enthalten suspend-Funktionen für asynchrone Datenbankzugriffe.
OnConflictStrategy	Die Verwendung von <code>OnConflictStrategy.REPLACE</code> bei Insert-Operationen stellt sicher, dass vorhandene Einträge aktualisiert werden, anstatt Fehler zu verursachen. Dies ist besonders nützlich für Synchronisationsoperationen, bei denen Daten regelmäßig aktualisiert werden.
Dateispeicherung	Der Service unterstützt auch Dateispeicherung, was für Features wie CampusRoom-Search wichtig ist, die PDF-Dateien speichern müssen. Die <code>storeFile()</code> - und <code>getFile()</code> -Methoden ermöglichen es, beliebige Binärdaten zu speichern und abzurufen, ohne dass externe Dateisystem-Zugriffe erforderlich sind.

Tabelle 5.8.: Technische Aspekte des CommonStorage Moduls

5.2.4. CommonPresentation

Das Common Presentation Module stellt eine Sammlung wiederverwendbarer UI-Komponenten für Jetpack Compose bereit, die von allen Feature-Modulen verwendet werden können. Es bietet standardisierte Komponenten für Buttons, Textfelder, Progress-Indikatoren, Dialoge, Fehleranzeigen und weitere UI-Elemente, die eine konsistente Benutzeroberfläche über die gesamte Anwendung hinweg gewährleisten.

Ziel und Motivation

Das Hauptziel besteht darin, Code-Duplikation zu vermeiden und eine konsistente Benutzeroberfläche zu schaffen, indem gemeinsame UI-Komponenten zentralisiert werden. Durch die Verwendung von wiederverwendbaren Komponenten wird sichergestellt, dass alle Feature-Module das gleiche Design-System verwenden und Änderungen am Design zentral vorgenommen werden können.

Ein weiteres wichtiges Ziel ist die Bereitstellung einer generischen Bootstrapping-Progress-View, die mit jedem Modul-Loader verwendet werden kann, der das `ModuleLoaderContract` Interface implementiert. Dies ermöglicht es, den gesamten Datenlade- und Synchronisationsprozess mit einer einheitlichen UI zu visualisieren, ohne dass jedes Feature-Modul seine eigene Implementierung erstellen muss.

Umsetzung / Funktionsweise

Die Komponenten sind als Composable-Funktionen implementiert und folgen den Jetpack Compose Best Practices. Die wichtigste Komponente ist `CommonBootstrappingProgressView`, die eine generische Implementierung für das Anzeigen von Lade- und Synchronisationsstatus bietet:

```
1 @Composable
2 fun CommonBootstrappingProgressView(
3     loader: ModuleLoaderContract<*>,
4     successView: @Composable () -> Unit
5 ) {
6     val status by loader.loadingStatus.collectAsState()
7     // ...
8 }
```

Die Komponente reagiert auf Statusänderungen des Modul-Loaders und zeigt entsprechend den aktuellen Zustand an. Bei `Not_Initialized` wird die `setup()`-Funktion aufgerufen, bei `Initialized` wird automatisch `process()` aufgerufen, und bei `Success` wird die übergebene `successView` angezeigt.

Für Buttons werden standardisierte Komponenten bereitgestellt. `PrimaryButton` ist eine wiederverwendbare Button-Komponente mit konsistentem Styling:

```
1 @Composable
2 fun PrimaryButton(
3     text: String,
4     onClick: () -> Unit,
5     modifier: Modifier = Modifier
6 ) {
7     Button(onClick = onClick, modifier = modifier) {
8         Text(text)
9     }
10 }
```

Die `AutoCompleteTextField` Komponente bietet eine Suchfunktion mit automatischer Vervollständigung.

Die `LogoSplashView` Komponente zeigt einen animierten Splash-Screen mit Logo und Untertitel.

Die `WebViewScreen` Komponente bietet eine integrierte WebView-Implementierung mit Unterstützung für interne

und externe Browser.

Weitere Informationen

Aspekt	Beschreibung
Material Design 3	Alle Komponenten verwenden Material Design 3 und folgen den Jetpack Compose Best Practices. Die Komponenten sind vollständig reaktiv und reagieren auf State-Änderungen durch StateFlow oder andere State-Management-Mechanismen.
GenericErrorView	Die GenericErrorView Komponente bietet eine standardisierte Fehleranzeige mit optionaler Retry-Funktionalität.
Dependency Injection	Das Modul verwendet Hilt für Dependency Injection, insbesondere für View-Models wie <code>WebViewViewModel</code> . Die WebView-Implementierung unterstützt JavaScript, Zoom-Kontrollen und verschiedene Cache-Modi basierend auf der Netzwerkverfügbarkeit.
Flexibilität	Die Komponenten sind so designed, dass sie flexibel anpassbar sind durch Modifier-Parameter und Lambda-Funktionen für Callbacks. Dies ermöglicht es, die Komponenten in verschiedenen Kontexten zu verwenden, während das grundlegende Design und Verhalten konsistent bleibt.
Integration mit Common Domain	Das Modul integriert sich nahtlos mit dem Common Domain Modul, indem es die <code>ModuleLoaderContract</code> Interface und <code>CommonModuleLoaderStatus</code> Enum verwendet, was eine lose Koppelung zwischen Presentation- und Application-Schicht gewährleistet.

Tabelle 5.9.: Technische Aspekte des CommonPresentation Moduls

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

Das Common Util Module stellt eine Sammlung von wiederverwendbaren Utility-Funktionen bereit, die von allen Feature-Modulen verwendet werden können. Es bietet Funktionalitäten für Datumsformatierung, HTML-Content-Parsing und JSON-Datenlade-Operationen. Diese Utilities zentralisieren häufig verwendete Operationen und vermeiden Code-Duplikation über die gesamte Anwendung hinweg.

Ziel und Motivation

Das Hauptziel besteht darin, häufig verwendete Utility-Funktionen zu zentralisieren und eine konsistente Implementierung über alle Feature-Module hinweg zu gewährleisten. Durch die Bereitstellung von wiederverwendbaren Funktionen für Datumsformatierung, HTML-Parsing und JSON-Datenlade-Operationen wird sichergestellt, dass alle Module die gleichen Formatierungs- und Parsing-Regeln verwenden.

Ein weiteres wichtiges Ziel ist die Abstraktion komplexer Operationen wie HTML-Parsing und Content-Block-Extraktion, die in mehreren Features (News, Blog, Events) benötigt werden. Durch die zentrale Implementierung können Fehlerbehebungen und Verbesserungen an einer Stelle vorgenommen werden, die dann allen Modulen zugute kommen.

Umsetzung / Funktionsweise

Das Modul besteht aus drei Hauptkomponenten: `DateFormatter` für Datumsformatierung, `HtmlContentParser` für HTML-Parsing und `JsonDataLoader` für JSON-Datenlade-Operationen.

Die `DateFormatter` Klasse bietet Funktionen zur Formatierung von ISO-Datumsstrings in deutsches Format. Die `formatDate()`-Methode konvertiert ein ISO-formatierte Datum (yyyy-MM-dd'T'HH:mm:ss) in deutsches Format (dd.MM.yyyy):

```

1 object DateFormatter {
2     fun formatDate(isoDate: String): String {
3         val inputFormat = SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss", Locale.US)
4         val outputFormat = SimpleDateFormat("dd.MM.yyyy", Locale("de", "CH"))
5         return outputFormat.format(inputFormat.parse(isoDate) ?: Date())
6     }
7 }
```

Die `HtmlContentParser` bietet Funktionen zum Parsen von HTML-Content und Extraktion von Content-Blöcken. Die `stripOutHtml()`-Extension-Funktion entfernt HTML-Tags und dekodiert HTML-Entities:

```

1 fun String.stripOutHtml(): String {
2     return this.replace(Regex("<[^>]*>"), "")
3     .replace("&nbsp;", " ")
4     .replace("&ldquo;", "\"")
5     // ...
6 }
```

Die `parseContentBlocks()`-Funktion extrahiert strukturierte Content-Blöcke aus HTML und erzeugt eine Liste von ContentBlock-Objekten (z. B. Paragraph, Subheading, Blockquote).

Die `JsonDataLoader` Klasse bietet generische Funktionen für JSON-Datenlade-Operationen. Die `syncData()`-Funktion lädt Daten von Remote oder aus dem lokalen Cache und implementiert eine Offline-First-Strategie.

Die `reset()`-Funktion ermöglicht das Zurücksetzen und Neu-Synchronisieren von Daten, indem der lokale Cache gelöscht und die Daten erneut vom Remote-Server geladen werden.

Weitere Informationen

Aspekt	Beschreibung
DateFormatter	Die <code>DateFormatter</code> verwendet die Zeitzone Europe/Zurich und das Locale "de-CH" für schweizerdeutsche Formatierung. Dies stellt sicher, dass alle Datumsformatierungen in der Anwendung konsistent sind und der schweizerischen Konvention entsprechen.
Offline-First Strategie	Die <code>JsonDataLoader</code> implementiert eine Offline-First-Strategie, bei der zuerst versucht wird, Daten aus dem Netzwerk zu laden, und bei fehlender Verbindung auf den lokalen Cache zurückgegriffen wird. Alle Operationen werden asynchron über Kotlin Coroutines auf Dispatchers.IO ausgeführt, um den Main-Thread nicht zu blockieren.
Generische Funktionen	Die Verwendung von generischen Funktionen in <code>JsonDataLoader</code> ermöglicht es, die Funktionen mit verschiedenen Datentypen zu verwenden, während die Parsing-Logik von den aufrufenden Modulen bereitgestellt wird. Dies gewährleistet Flexibilität bei gleichzeitiger Wiederverwendbarkeit.
Abhängigkeiten	Das Modul hat minimale Abhängigkeiten: nur Android Core, Common Domain und Common Infrastructure. Dies stellt sicher, dass die Utilities leichtgewichtig bleiben und keine unnötigen Abhängigkeiten einführen.

Tabelle 5.10.: Technische Aspekte des CommonUtil Moduls

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.

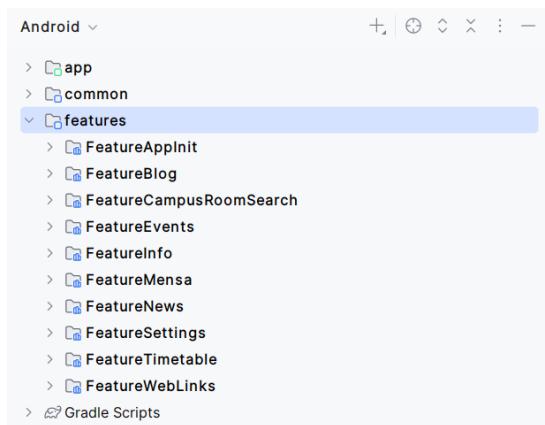


Abbildung 7.: Features in Android-Studio

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 (Einzige Ausnahme: *ApplInit*). Dadurch werden zyklische Abhängigkeiten vermieden und die Architektur bleibt langfristig erweiterbar und wartbar.

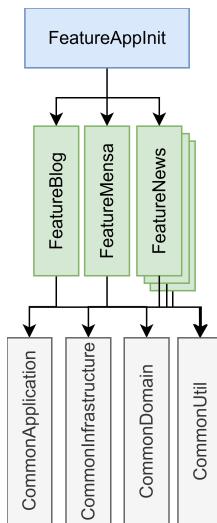


Abbildung 8.: Abhängigkeiten in Pfeilrichtung erlaubt/vorhanden

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.

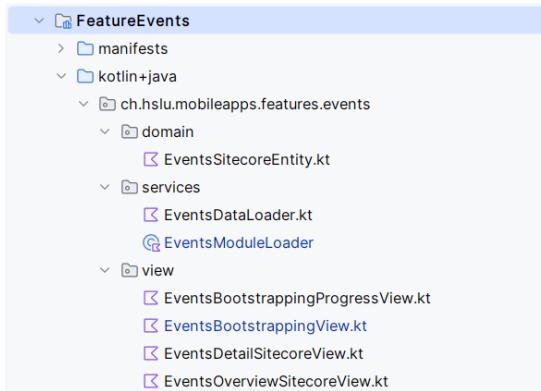


Abbildung 9.: Struktur der Features am Beispiel Event

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.

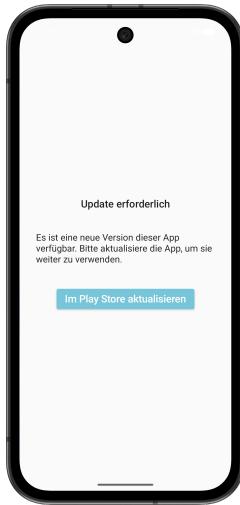


Abbildung 10.: Aufforderung zum Versionsupdate

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.

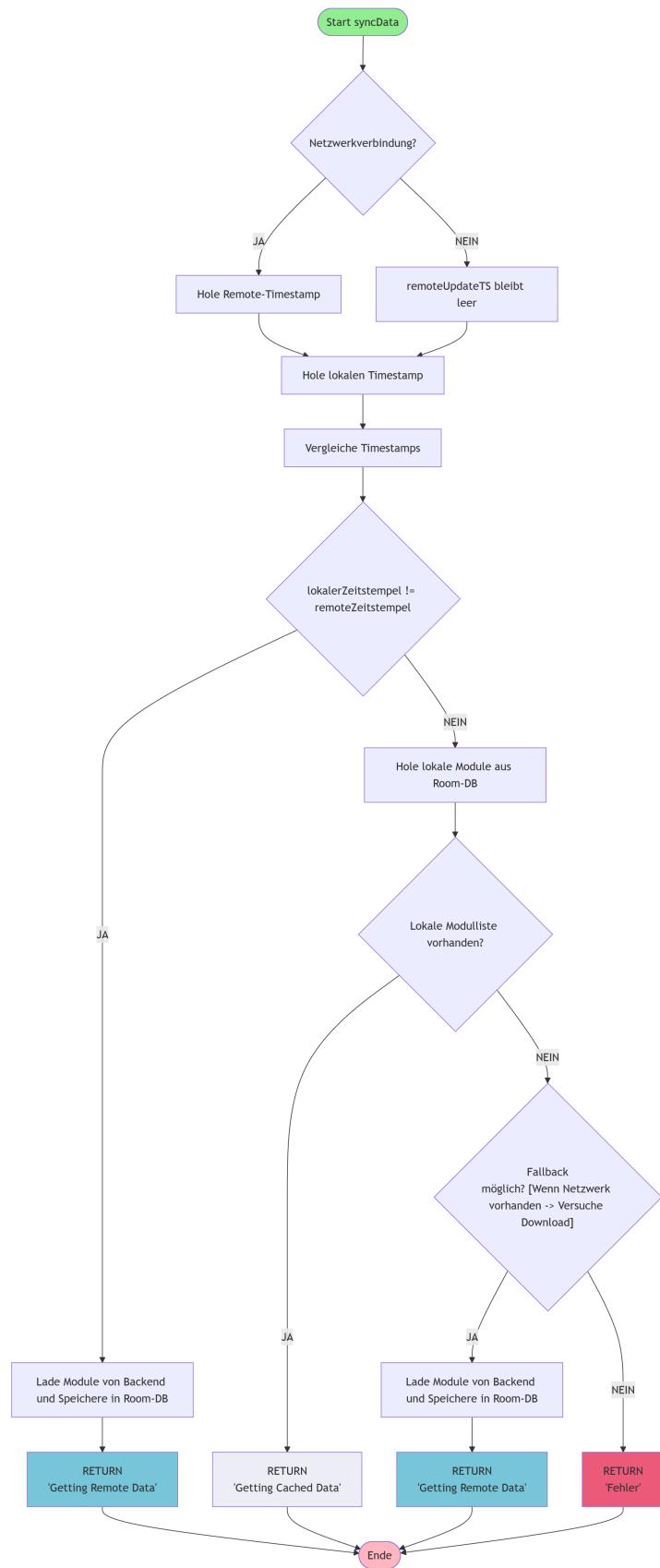


Abbildung 11.: Flussdiagramm Menü-Synchronisation

5.3.2. News

Das Feature *FeatureNews* dient dazu, die aktuellen Neuigkeiten der HSLU über die WordPress-basierte Plattform <https://news.hslu.ch/> direkt in der App anzuzeigen. Ursprünglich war vorgesehen, die Inhalte über eine einfache WebView zu laden, doch im Laufe des Semesters wurde die Lösung vollständig auf eine JSON-basierte Darstellung umgestellt.

Motivation und Zielsetzung

Im ersten Schritt wird der anzugezeigende News-Link nicht hart im Code definiert, sondern von der API geladen. Dadurch können die App-Verantwortlichen die Quelle jederzeit anpassen, ohne dass die App neu kompiliert werden muss. Zudem ermöglicht dies eine saubere Trennung für unterschiedliche Tenants (z. B. *HSLU-I* und *HSLU-TA*), da beide auf verschiedene APIs zugreifen und somit unterschiedliche News-Quellen verwenden können.

Die ursprüngliche Umsetzung im früheren XML-basierten Android-Projekt verwendete die Accompanist-WebView (`noauthor_accompanist_nodate`). Da diese Library jedoch als *deprecated* markiert wurde, ersetzen wir sie durch die native `android.webkit.WebView`, welche alle benötigten Funktionen bereitstellt.

Während des Semesters fiel jedoch dann die Entscheidung, auf beiden Plattformen (iOS und Android) auf WebViews zu verzichten. Gründe dafür waren insbesondere Datenschutzaspekte, fehlende Kontrolle über Tracking-Mechanismen externer Websites sowie die Tatsache, dass WebViews Abhängigkeiten zu fremden Cookies, Skripten und Datenschutzrichtlinien erzeugen.

Die Lösung wurde deshalb auf ein JSON-basiertes Rendering umgestellt: Die App lädt nicht mehr die komplette Webseite, sondern nur die strukturierten Daten des Artikels und rendert den Inhalt vollständig nativ.

Funktionsweise JSON-View

Die News werden über die WordPress-REST-API von `news.hslu.ch` als JSON geladen. Der Download erfolgt über den `NewsDataLoader`, der auf der gemeinsamen Utility-Klasse `JsonDataLoader` basiert. Diese stellt ein hybrides Ladeverhalten bereit:

- Bei aktiver Internetverbindung werden die Daten remote geladen und lokal gecached.
- Ohne Internet werden die News aus dem Cache gelesen (falls vorhanden).

Die JSON-Daten werden anschliessend in das Domain-Objekt `NewsPostWordpress.kt` konvertiert, das Titel, Beschreibung, Inhalt, Publikationsdatum, Autor sowie das *featured image* enthält.

Um die Inhalte darzustellen, werden die HTML-Fragmente der WordPress-API mit einem HTML-Parser verarbeitet. Der Parser extrahiert Absätze, Überschriften, Blockquotes und wandelt HTML-Entities wie oder “ in reguläre Zeichen um. Der Parser erzeugt daraus eine strukturierte Liste von Content-Blöcken (z. B. Subheading, Paragraph), die anschliessend vom Compose-Renderer (`NewsContentRenderer`) visuell aufbereitet werden. Die grundlegende Logik des Parsers basiert auf regulären Ausdrücken und Entity-Decoding.

Darstellung der News

Die Darstellung erfolgt dann vollständig in Jetpack Compose und besteht aus zwei Bereichen:

- NewsOverviewWordpressView: Zeigt eine Übersicht aller verfügbaren News. Jede News wird als Karte mit Titel, Bild, Kurzbeschreibung und Datum dargestellt. Die View beobachtet den Ladezustand des NewsDataLoader und reagiert reaktiv auf Statusänderungen.
- NewsDetailWordpressView: Öffnet einen einzelnen Artikel als Dialog oder Vollbildansicht. Die View rendert den gesamten strukturierten Inhalt, erlaubt das Ausklappen langer Texte und bietet einen Button „*Im Browser öffnen*“, falls der Nutzer den Originalartikel betrachten möchte.

Herausforderungen und Einschränkungen

Durch den Wechsel von der WebView zur nativen JSON-Darstellung ist die App nun darauf angewiesen, dass die HTML-Struktur der WordPress-Beiträge stabil bleibt. Während der Parser viele HTML-Entitäten und Formatierungen zuverlässig erkennt, ist es technisch nicht möglich, alle denkbaren Formatierungsvarianten eindeutig zu interpretieren.

Beispielsweise kann ein Redaktor einen Titel visuell fett und gross darstellen, ohne ihn als `<h1>`–`<h6>` zu markieren. In der WebView wäre dies optisch korrekt, aber unser Parser kann diese Formatierung nicht als Titel erkennen und behandelt sie als normalen Text. Solche Abweichungen liegen ausserhalb des Einflussbereichs der App und sind nur durch redaktionelle Disziplin vermeidbar.

Falls WordPress ein Update einführt oder die API plötzlich andere Strukturen liefert, kann dies zu Darstellungsproblemen führen. Im Notfall könnte das Backend temporär wieder auf die alte WebView-Variante zurückschalten, bis das neue Format korrekt unterstützt wird.

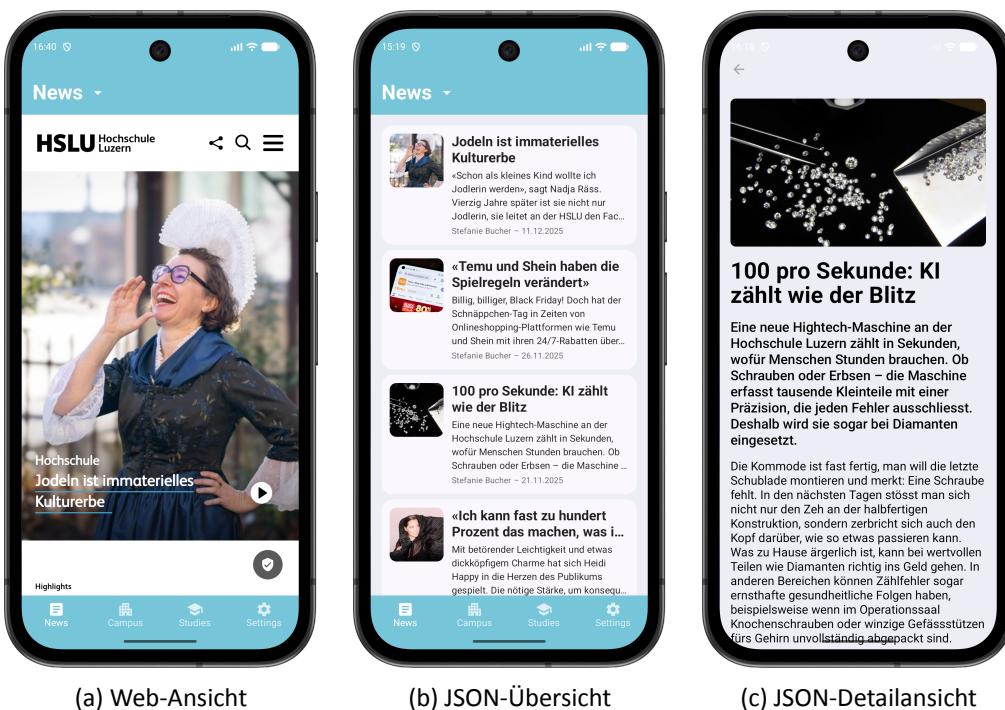


Abbildung 12.: Screenshots des News-Features

5.3.3. Blog

Das Feature *FeatureBlog* baut konzeptionell auf derselben Architektur wie das zuvor beschriebene *FeatureNews* auf, verfolgt jedoch einen leicht anderen Zweck: Es stellt die Blog-Beiträge der HSLU aus der WordPress-Installation <https://hub.hslu.ch/informatik/> dar. Der grundlegende technische Aufbau mit *ModuleLoader*, *DataLoader*, JSON-Synchronisation sowie HTML-Parsing ist identisch zum News-Feature, wodurch sich viele Komponenten wiederverwenden liessen. Auch hier ist ein Umschalten zwischen WebView- und JSON-View im Backend möglich, sodass bei Bedarf zwischen beiden Darstellungsvarianten gewechselt werden kann.

Die Umsetzung der UI folgt dabei denselben Prinzipien wie bei den News: Karten-basierte Übersicht, Detailansicht als Overlay/Dialog und Rendering des Inhalts über den bestehenden Parser und den *BlogContentRenderer*.

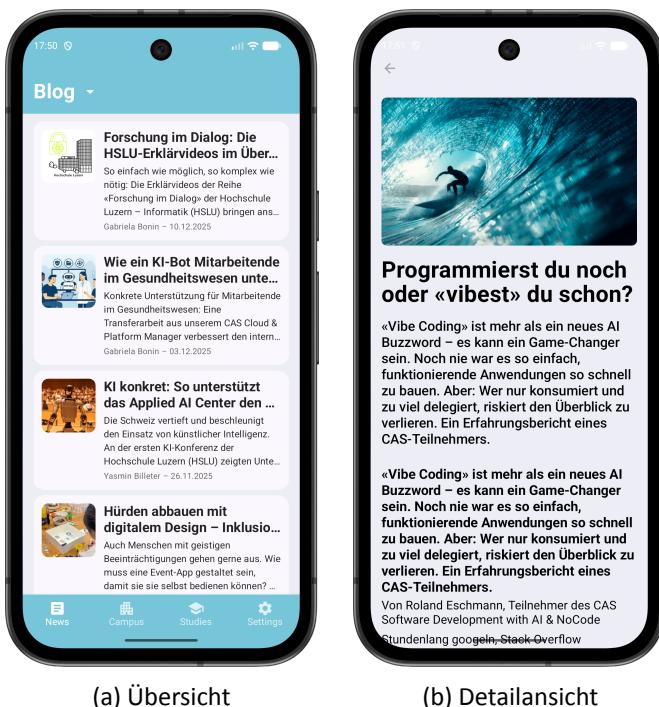


Abbildung 13.: Screenshots des Blog-Features



Abbildung 14.: Screenshot aus dem Backend: Umschalten der Views (Blog & News)

5.3.4. Mensa

Studierenden soll der aktuelle Menüplan des Mensa-Betreibers ZFV möglichst schnell und unkompliziert zur Verfügung stehen. Da zum Zeitpunkt der Entwicklung unklar war, ob ZFV eine stabile oder offiziell unterstützte API anbietet, wurde hier bewusst auf eine JSON-basierte Darstellung verzichtet und stattdessen eine WebView-Lösung umgesetzt. Die technische Umsetzung erfolgt über den MensaModuleLoader sowie eine einfache Compose-View, die wie zuvor schon News und Blog den WebViewScreen einbettet, um die Webseite anzuzeigen.

Der ZFV stellt den Menüplan neben der regulären Webseite auch als *iframe* bereit. Ein *iframe* erlaubt das Einbetten eines externen Webseitenabschnitts innerhalb einer anderen Website, wodurch nur ein bestimmter Teil des Inhalts geladen wird, statt der gesamten Seite (Hier konkret: Nur Menüplan statt kompletter Webseite). Diese Flexibilität wird im Backend genutzt: Dort kann konfiguriert werden, ob die gesamte Mensa-Webseite oder lediglich der eingebettete iframe-Inhalt angezeigt werden soll. Beide Varianten lassen sich einfach über die Hinterlegung der jeweiligen URL steuern und erfordern keine Anpassungen in der App selbst.

Die beiden Darstellungsoptionen sind hier zu sehen:

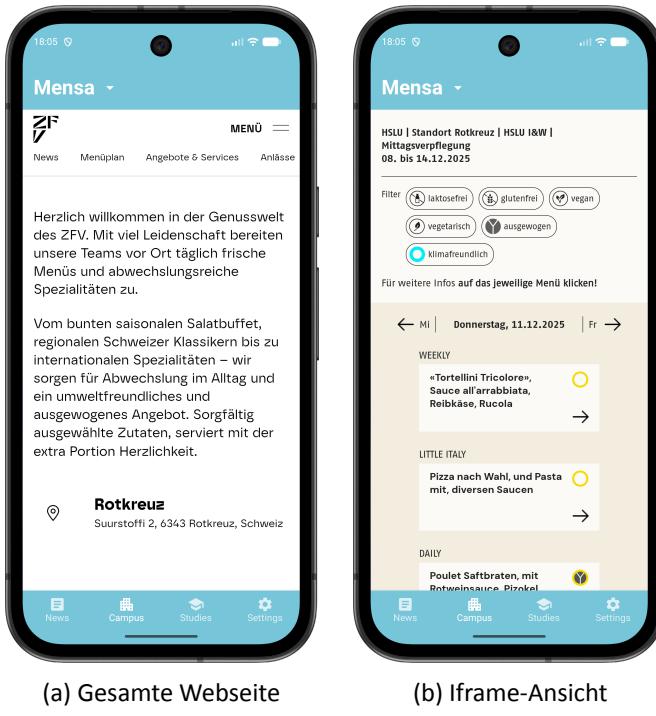


Abbildung 15.: Screenshots des Mensa-Features

5.3.5. Events

Das Feature *FeatureEvents* erweitert die App um die Möglichkeit, aktuelle Veranstaltungen der Hochschule Luzern direkt anzuzeigen. Im Gegensatz zu den zuvor beschriebenen Features *News* und *Blog*, die ihre Inhalte über die WordPress-API beziehen, basiert dieses Feature auf der Sitecore-Plattform der HSLU. Sitecore stellt Inhalte strukturiert über eine JSON-basierte API bereit und ist damit eine gute Grundlage für eine native Darstellung in der App. (noauthor_sitecore_nodate)

Die grundlegende Architektur folgt weiterhin dem bekannten Muster: Ein **ModuleLoader** stellt die Moduldefinitionen bereit, während der **EventsDataLoader** die Eventdaten lädt, in Kotlin-Domainobjekte umwandelt und diese der UI zur Verfügung stellt. Das JSON-Processing unterscheidet sich jedoch in einigen Punkten von News und Blog:

- Die Sitecore-API liefert andere Feldstrukturen, z.B. separate Eigenschaften für Start- und Enddatum.
- Der Eventtext enthält oft komplexere HTML-Fragmente, weshalb das Parsing stärker variieren kann.
- Die Events besitzen zusätzliche Metadaten wie *organizer*, *event type* oder Kategorien.

Die Daten werden in der App wieder in zwei Compose-Views dargestellt: einer Übersicht aller Events sowie einer Detailansicht (**EventsDetailSitecoreView**), die die Inhalte strukturiert rendert und zusätzlich den Link zur Eventseite der HSLU anbietet.

Um sicherzustellen, dass nur relevante Events des Departements Informatik angezeigt werden, kann das Backend Sitecore-spezifische Filter konfigurieren. So kann etwa der Query-Parameter **filters[] = 1621** gesetzt werden, ohne dass dieser Wert im App-Code hinterlegt sein muss. Die Filterlogik bleibt damit vollständig backend-gesteuert.

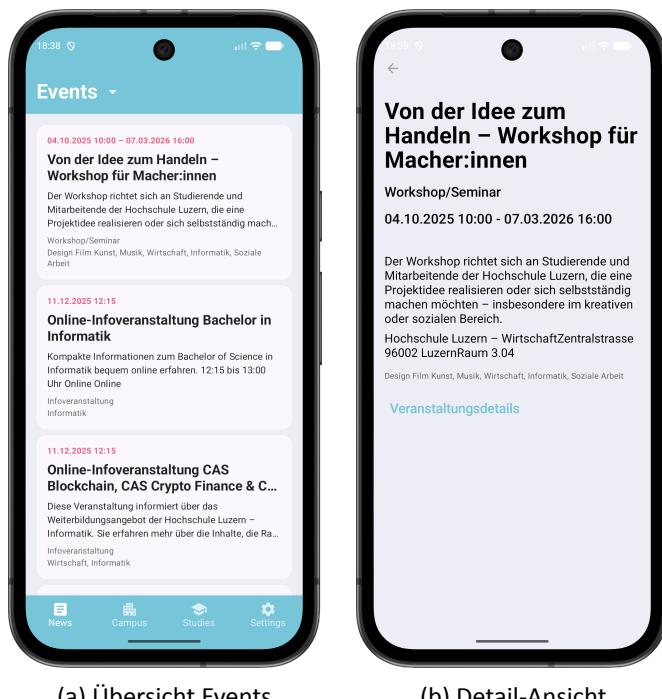


Abbildung 16.: Screenshots des Events-Features

5.3.6. Info/About

Todo Raphi

5.3.7. Settings

Todo Raphi

5.3.8. Weblinks

Todo Raphi

5.3.9. CampusRoomSearch

Das CampusRoomSearch Feature ermöglicht es Benutzern, Räume auf dem Campus zu suchen und zu finden. Es stellt interaktive PDF-Karten bereit, die Campus-Übersichten und Gebäudepläne enthalten, und ermöglicht es Benutzern, durch Klicken auf annotierte Bereiche in den PDFs zu navigieren und detaillierte Informationen zu Räumen, Gebäuden und Etagen abzurufen. Das Feature unterstützt sowohl deutsche als auch englische Sprachversionen und speichert PDF-Dateien lokal für Offline-Zugriff.

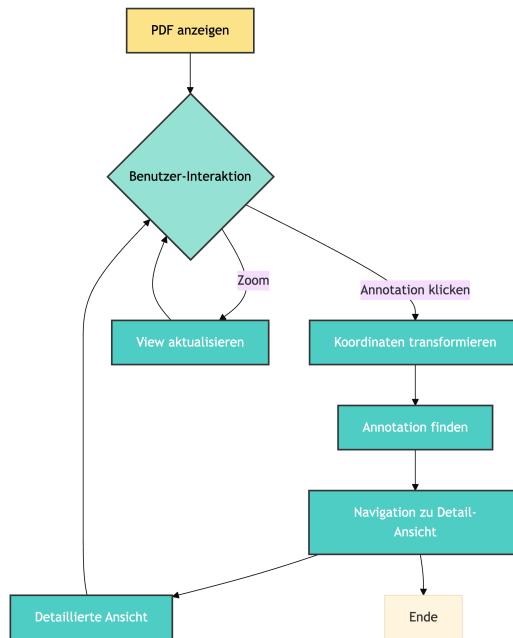


Abbildung 17.: Ablaufdiagramm

Ziel und Motivation

Das Hauptziel des CampusRoomSearch Features besteht darin, Studierenden und Mitarbeitenden eine intuitive Möglichkeit zu bieten, Räume auf dem Campus zu finden. Durch die Verwendung von interaktiven PDF-Karten mit Annotationen können Benutzer direkt auf Bereiche in den Karten klicken, um zu detaillierten Gebäude- und Etagenplänen zu navigieren. Dies bietet eine benutzerfreundliche Alternative zu textbasierten Suchfunktionen.

Ein weiteres wichtiges Ziel ist die Offline-Funktionalität. Da Campus-Pläne und Gebäudekarten häufig benötigt werden, werden alle PDF-Dateien lokal gespeichert, sodass das Feature auch ohne aktive Netzwerkverbindung vollständig funktionsfähig ist. Die Synchronisation erfolgt intelligent basierend auf Timestamp-Vergleichen, um unnötige Downloads zu vermeiden.

Umsetzung / Funktionsweise

Die Implementierung basiert auf einer zweistufigen Architektur: `CampusRoomSearchModuleLoader` lädt die Liste der verfügbaren Campus-Standorte, während `CampusRoomSearchDataLoader` die detaillierten Daten für einen spezifischen Campus-Standort lädt, einschließlich aller PDF-Dateien.

Der `CampusRoomSearchModuleLoader` erbt von `CommonApplicationBaseModuleLoader` und erweitert die Standard-Funktionalität, um nach dem Laden der Campus-Liste automatisch die detaillierten Daten für alle Items zu laden.

Die `loadRemoteDetailedData()`-Methode lädt die detaillierten Campus-Daten von der API und lädt anschlies-

send alle zugehörigen PDF-Dateien herunter.

Die PDFRoomSearchView Komponente rendert PDF-Dateien mit Android's PdfRenderer und unterstützt interaktive Annotationen. Die Komponente lädt PDFs aus der Room-Datenbank und ermöglicht Zoom- und Pan-Gesten:

```
1 @Composable
2 fun PDFRoomSearchView(
3     pdfData: ByteArray,
4     annotations: List<Annotation>,
5     onAnnotationClick: (Annotation) -> Unit
6 ) {
7     // PDF-Rendering mit PdfRenderer
8 }
```

Die Annotation-Erkennung funktioniert durch Koordinaten-Transformation. Wenn ein Benutzer auf das PDF klickt, werden die Tap-Koordinaten basierend auf dem aktuellen Zoom- und Pan-Zustand transformiert, um die entsprechende Annotation zu finden:

```
1 fun transformTapCoordinates(
2     tapX: Float, tapY: Float,
3     zoom: Float, panX: Float, panY: Float
4 ): Pair<Float, Float> {
5     return ((tapX - panX) / zoom, (tapY - panY) / zoom)
6 }
```

Der RoomsearchViewModel verwaltet die Suche nach Räumen und die Interaktion mit den SVG-Karten. Er verwendet JavaScript-Interfaces, um Kommunikation zwischen WebView und Android-Code zu ermöglichen:

```
1 interface IJavascriptHandler {
2     @JavascriptInterface
3     fun onBuildingClick(buildingId: String)
4 }
```

Weitere Informationen

Funktionalität	Beschreibung
Hybride Speicherstrategie	Das Feature verwendet eine hybride Speicherstrategie: JSON-Daten werden in der Room-Datenbank gespeichert, während PDF-Dateien als ByteArray in der FileEntity Tabelle gespeichert werden. Dies ermöglicht eine effiziente Speicherung und schnellen Zugriff auf die Dateien, ohne dass externe Dateisystem-Zugriffe erforderlich sind.
PDF-Annotationen	Die PDF-Annotationen werden aus den PDF-Metadaten extrahiert und als unsichtbare, aber klickbare Bereiche über dem PDF gerendert. Dies ermöglicht eine präzise Interaktion, während das ursprüngliche PDF-Layout erhalten bleibt. Die Koordinaten-Transformation berücksichtigt Zoom- und Pan-Zustände für eine genaue Erkennung.
Sprachauswahl	Die Sprachauswahl erfolgt automatisch basierend auf der System-Locale. Falls die bevorzugte Sprache nicht verfügbar ist, wird automatisch auf die alternative Sprache zurückgegriffen. Dies gewährleistet, dass das Feature immer funktionsfähig ist, unabhängig von der verfügbaren Sprachversion.
Intelligente Synchronisation	Die Synchronisation verwendet Timestamp-Vergleiche, ähnlich wie andere Module. Der CampusRoomSearchDataLoader prüft zuerst, ob lokale Daten vorhanden sind, und lädt nur dann Remote-Daten, wenn eine Aktualisierung notwendig ist oder keine lokalen Daten vorhanden sind. Dies reduziert unnötige Downloads und spart Bandbreite.
Hierarchische Navigation	Das Feature unterstützt mehrere Campus-Standorte, wobei jeder Standort seine eigenen Gebäude, Etagen und Räume hat. Die Navigation erfolgt hierarchisch: Campus → Gebäude → Etage → Raum, wobei jede Ebene durch entsprechende PDF-Karten und Annotationen dargestellt wird.
WebView-Integration	Die WebView-Integration für SVG-Karten ermöglicht interaktive Gebäudekarten, bei denen Benutzer auf Gebäude klicken können, um direkt zu den entsprechenden Etagenplänen zu navigieren. Die JavaScript-Interface-Implementierung ermöglicht eine bidirektionale Kommunikation zwischen der WebView und dem Android-Code.
Offline-Funktionalität	Alle PDF-Dateien werden lokal gespeichert, sodass das Feature auch ohne aktive Netzwerkverbindung vollständig funktionsfähig ist. Die Synchronisation erfolgt intelligent basierend auf Timestamp-Vergleichen, um unnötige Downloads zu vermeiden.
PDF-Rendering	Die PDF-Dateien werden mit Android's PdfRenderer gerendert und unterstützen Zoom- und Pan-Gesten. Die Komponente lädt PDFs aus der Room-Datenbank und ermöglicht eine flüssige Interaktion mit den Karten.

Tabelle 5.11.: Wichtigste Funktionalitäten des CampusRoomSearch Features

5.3.10. Timetable

Das Feature *Timetable* dient dazu, den persönlichen Stundenplan der Studierenden direkt in der HSLU-App anzuzeigen. Die Grundidee ist gleich wie bei der iOS-App: Die Applikation greift nicht direkt auf *MyCampus* zu, sondern setzt voraus, dass die Nutzer:innen ihren Studienkalender bereits über *MyCampus* per Link in ihre lokale Kalender-App eingebunden haben. Eine direkte *MyCampus*-Schnittstelle ist aktuell nicht vorgesehen, daher bleibt der lokal installierte Android-Kalender die einzige Datenquelle.

Zugriff auf den Android-Kalender

Android stellt über das `CalendarContract`-API einen strukturierten Zugriff auf Kalender und Termine bereit (noauthor_andro).

Die App kann damit:

- verfügbare Kalender des Geräts auslesen,
- die Ereignisse eines gewählten Kalenders für einen bestimmten Zeitraum laden,
- sowie alle Einträge nach Start- und Endzeit filtern.

Damit dieses Feature funktioniert, muss der Nutzer der App die `READ_CALENDAR`-Berechtigung erteilen. Beim Öffnen des Features erscheint die Abfrage für die Berechtigung automatisch. Anschliessend kann in der `TimetableCalendarSwitcher`-View ein Kalender ausgewählt werden. Die Wahl wird in den `SharedPreferences` gespeichert, sodass der Nutzer sie nicht bei jedem App-Start erneut treffen muss. Änderungen können jederzeit über Einstellungen der App vorgenommen werden.

Die eigentliche Verarbeitung der Kalendereinträge erfolgt über den `CommonCalendarDataLoader`, der mithilfe von `ContentResolver.query()` die Einträge über `CalendarContract Instances` abruft. Aus den Resultaten werden Objekte vom Typ `LectureDTO` instanziert, welche unter anderem Name, Notizen, Raum, Start-, Endzeit sowie diverse abgeleitete Informationen enthalten (z. B. Kurzname, formattiertes Datum, ILIAS-Link). Damit das Erstellen der `LectureDTO`-Objekte korrekt abläuft, müssen die Kalendereinträge vom Format so im Kalender vorhanden sein, wie sie auch von *MyCampus* bereitgestellt werden.

Darstellung in der App

Nach erfolgreichem Laden wird der Stundenplan für die nächsten sieben Tage aufbereitet und in einer strukturierten Übersicht dargestellt. Die Kalendereinträge werden gruppiert nach:

- Heute,
- Morgen,
- Datum (für spätere Tage).

Jede Tagesgruppe wird in einer eigenen Sektion dargestellt, wie in der `TimetableListView` ersichtlich. Innerhalb der Sektionen werden die Details pro Vorlesung angezeigt, darunter Kurzname, vollständiger Titel, Zeitfenster, Raum sowie weiterführende Informationen. Bei nicht vorhandenen zukünftigen Terminen wird automatisch die `TimetableEmptyView` angezeigt.

Erkennung von Raumänderungen

Ein praktischer Zusatznutzen dieses Features ist die Erkennung von Raumänderungen gegenüber der Vorwoche. Die App vergleicht frühere und aktuelle Einträge und markiert Vorlesungen visuell, wenn sich der Raum geändert hat. Dies verhindert, dass Studierende versehentlich in den falschen Raum gehen und erhöht die Alltagstauglichkeit des Features.

Ergebnis

Das Timetable-Feature bietet eine kompakte und alltagstaugliche Übersicht der kommenden Vorlesungen, gefiltert auf die relevanten Tage und ohne Ablenkung durch andere private Termine, wie sie in der Standard-Kalender-App erscheinen. Studierende erhalten damit einen schnellen, Zugang zu den Hochschulterminen direkt innerhalb der mobilen Applikation, ohne App- oder Kontextwechsel.

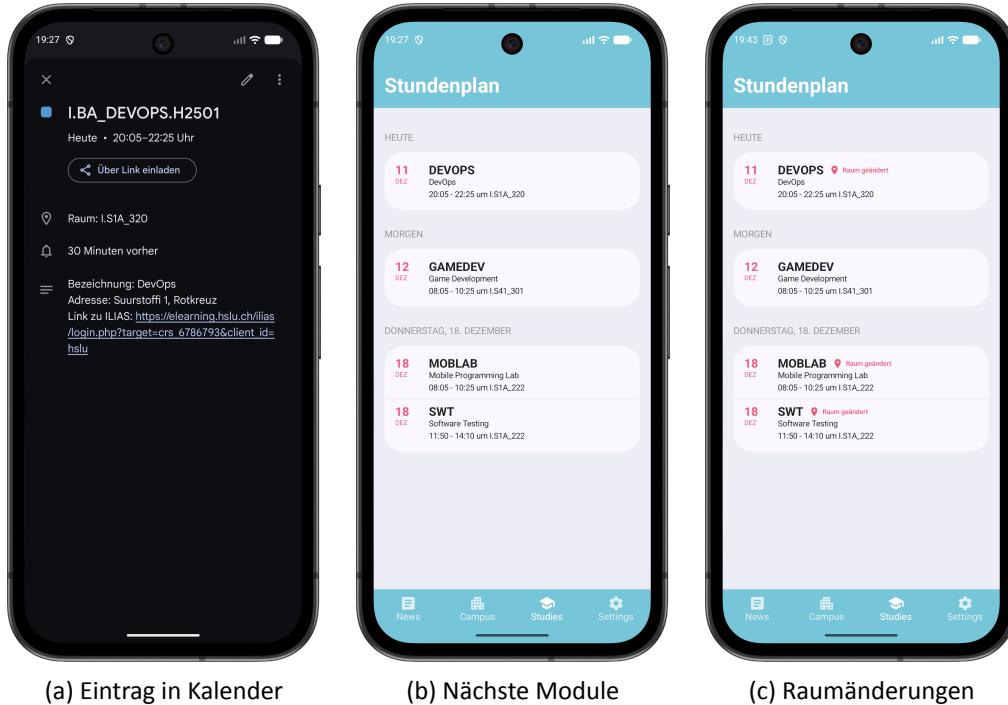


Abbildung 18.: Screenshots des Kalender-Features

5.3.10.1. Timetable-Widget

Zusätzlich zur Stundenplan-Ansicht innerhalb der App bietet das *Timetable*-Feature ein Home-Screen-Widget an, welches die nächsten Vorlesungen direkt auf dem Startbildschirm des Geräts anzeigt. Das Widget ist mit Jetpack *Compose* umgesetzt und nutzt damit die Compose-basierte Widget-Architektur von Android (noauthor_android_nodate-1).

Registrierung und Integration

Das Widget wird im `AndroidManifest.xml` registriert. Der `TimetableWidgetReceiver` reagiert auf das System-Event `APPWIDGET_UPDATE` und verweist über Meta-Daten auf die Widget-Konfiguration (`timetable_widget_info.xml`). Diese definiert unter anderem die Grösse und das automatische Update-Intervall (30 Minuten) des Widgets. Nach der Installation erscheint das Widget automatisch in der Widget-Auswahl des Systems.

Ein Beispiel der Registrierung im `AndroidManifest.xml`:

```

1 <application>
2     <receiver
3         android:name=".widget.TimetableWidgetReceiver"
4         android:exported="true">
5             <intent-filter>
6                 <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
7             </intent-filter>
8             <meta-data
9                 android:name="android.appwidget.provider"
10                android:resource="@xml/timetable_widget_info" />
11         </receiver>
12     </application>

```

Widget-Architektur

Für die Umsetzung wurde bewusst Jetpack Glance verwendet, da es die moderne, Compose-orientierte Alternative zum klassischen `AppWidgetProvider` ist. Die Struktur im Code ist wie folgt aufgebaut:

- `TimetableWidgetReceiver`: Einstiegspunkt für das System, leitet Events an das Widget weiter.
- `TimetableWidget`: Lädt die Daten und stellt über `provideContent{}` den UI-Baum bereit.
- `TimetableWidgetContent`: Compose-basierte Darstellung des Widgets, inklusive Fehler-, Leer- und Erfolgszuständen.

Die Daten werden über den `TimetableWidgetDataProvider` geladen. Da Widgets in einem separaten Prozess laufen und kein Dependency-Injection-Framework wie Hilt verfügbar ist, erfolgt der Kalenderzugriff direkt über `CalendarContract` sowie die gespeicherten Einstellungen aus den `SharedPreferences`. Das Widget zeigt bis zu vier kommende Vorlesungen des aktuellen Tages an und blendet bereits abgelaufene Termine automatisch aus.

Da Widgets ebenfalls keinen Zugriff auf `MaterialTheme` haben, wird die Farbe manuell aus den App-Ressourcen geladen. Über `getIdentifier()` wird der Farb-Resource-Name ermittelt und mit `ContextCompat.getColor()` in ein Compose-Color-Objekt umgewandelt, sodass das Widget trotzdem die definierte HSLU-Farbe verwenden kann.

UI-Konzept

Die grafische Darstellung ist kompakt gehalten, um auf der kleinen Widget-Flächen die nötigsten Informationen darzustellen. Je nach Zustand zeigt das Widget:

- eine Fehlermeldung (z. B. fehlende Kalenderberechtigung),
- die nächsten Vorlesungen des Tages inklusive Zeit, Kurzname und Raum,
- oder einen leeren Zustand („Keine Vorlesungen heute“).

Ergebnis

Das Timetable-Widget erweitert das Feature um eine praktische Ansicht, damit die Nutzer:innen den Stundenplan jederzeit einsehen können, ohne die App öffnen zu müssen.



Abbildung 19.: Screenshots des Kalender-WIDGETS

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 (JUnit 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 zu 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. 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. Die vollständige Aufgabenstellung ist im Anhang dokumentiert (siehe Anhang A.2).

6.2.2. Auswertung

Wir haben die folgenden KI-Tools evaluiert: ChatGPT, Grok (xAI), DeepSeek und Cursor. Jedes Tool hat die gleiche Aufgabenstellung erhalten und wurde auf seine Fähigkeiten zur Analyse und Unterstützung bei der Android-App-Entwicklung geprüft.

ChatGPT lieferte eine sehr gute Antwort mit vernünftiger Tiefe für eine erste Analyse. Die KI ist sehr stark bei Cross-Plattform-Wissen (iOS/Android, SwiftUI/Compose, Flutter, React Native) und bietet bewährte Architektur- und Testing-Hinweise. Die Pro-Version kostet 23 Euro pro Monat. Die vollständige Antwort von ChatGPT ist im Anhang dokumentiert (siehe Anhang A.2).

Grok von X hat ebenfalls eine gute Antwort gegeben, womit diverse gute Ansätze beschrieben sind und man eine gute Übersicht erhält, was gemacht werden muss und wie dies möglich sein kann. xAI hat mit „grok-code-fast-1ein Modell für Entwickler-Aufgaben (Agentic Coding) vorgestellt, das Effizienz und Qualität in den Fokus stellt. Die SuperGrok-Version kostet 30 Euro pro Monat. Die vollständige Antwort von Grok ist im Anhang dokumentiert (siehe Anhang A.2).

DeepSeek hat aus unserer 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. DeepSeek ist eine kostengünstige, vergleichbare Alternative mit der Möglichkeit zur Integration in IDE-Umgebungen. Die Nutzung ist kostenlos, es gibt keine Möglichkeit ein Abo abzuschliessen. Die vollständige Antwort von DeepSeek ist im Anhang dokumentiert (siehe Anhang A.2).

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, da die KI den Kontext des gesamten Projekts hat um möglichst sinnvollen Code zu erstellen. Die Pro-Version kostet 20 Euro pro Monat. Die vollständige Antwort von Cursor ist im Anhang dokumentiert (siehe Anhang A.2).

6.2.3. Vergleich

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 (auer 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).

Tabelle 6.1.: Auswertung der KI in Tabellenform

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.

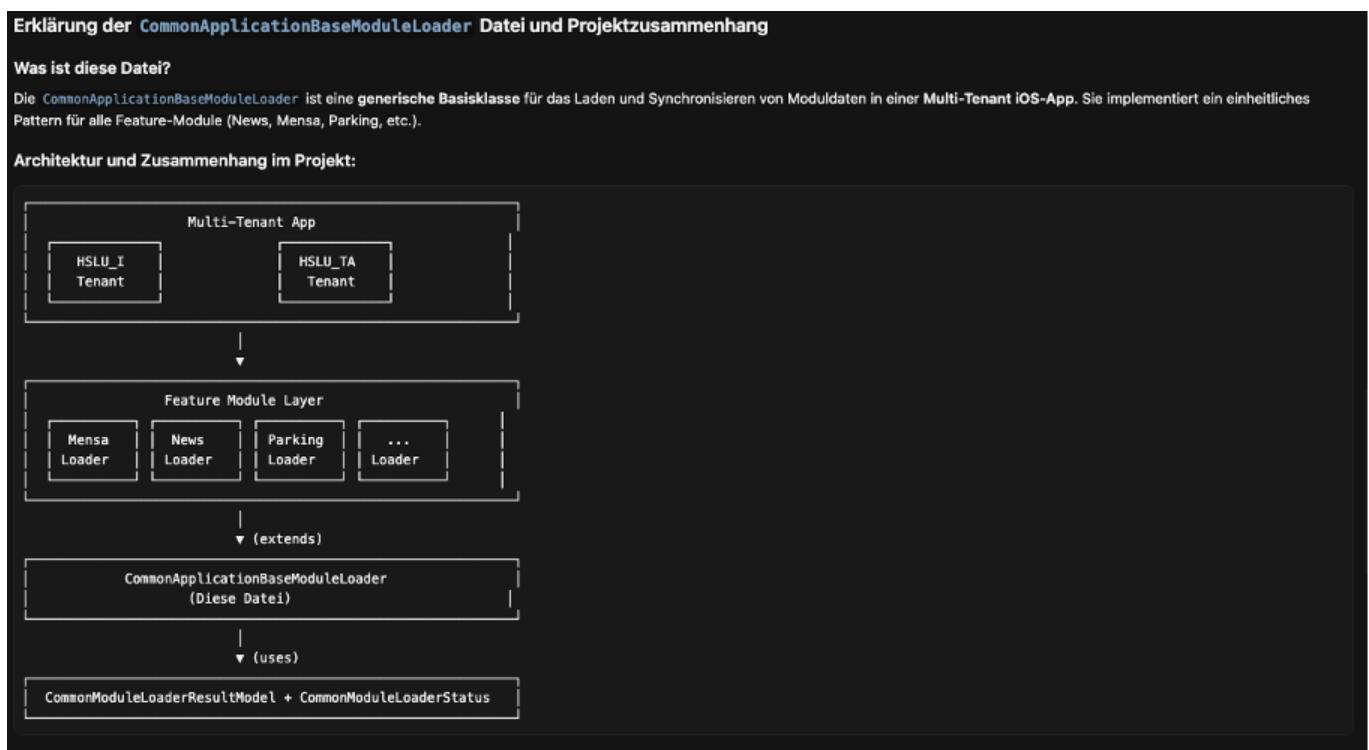


Abbildung 20.: Analysebeispiel in Cursor

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 21.: Analysebeispiel in Cursor

Dependency Injection Pattern:

```
// In HsluTAApp.swift / HsluIApp.swift
mensaModuleLoader.setup(
    tenantConfig: appTenantConfig,          // Tenant-spezifische URLs
    appConfig: mensaConfig,                // 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 22.: Analysebeispiel in Cursor

Typische Nutzung:

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

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

// UI reagiert automatisch auf loadingStatus-Änderungen
switch mensaLoader.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 23.: Analysebeispiel in Cursor

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.

Nachdem wir die Analyse machten haben wir uns mit dem Dozenten kurz geschlossen und er hat sich bei der HSLU informiert, welche AI Tools sie begünstigen. Daraus wurde dann klar, dass die HSLU sich für ChatGPT entschieden hat und uns somit die Entscheidung zum Teil abgenommen wurde. Da wir aber trotzdem so gut Erfahrungen mit Cursor gemacht haben, wollten wir darauf nicht verzichten. Cursor kann so konfiguriert werden, dass es im Hintergrund eine API eines LLM verwendet, womit wir die Vorgabe ChatGPT einhalten konnten, in dem wir Cursor so konfiguriert haben, dass es nur mit der ChatGPT API kommuniziert. Somit konnten wir einen guten Kompromiss eingehen und hatten ein Tool womit ein ganzes Projekt (fast) im Kontext der Anfrage verfügbar ist.

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:

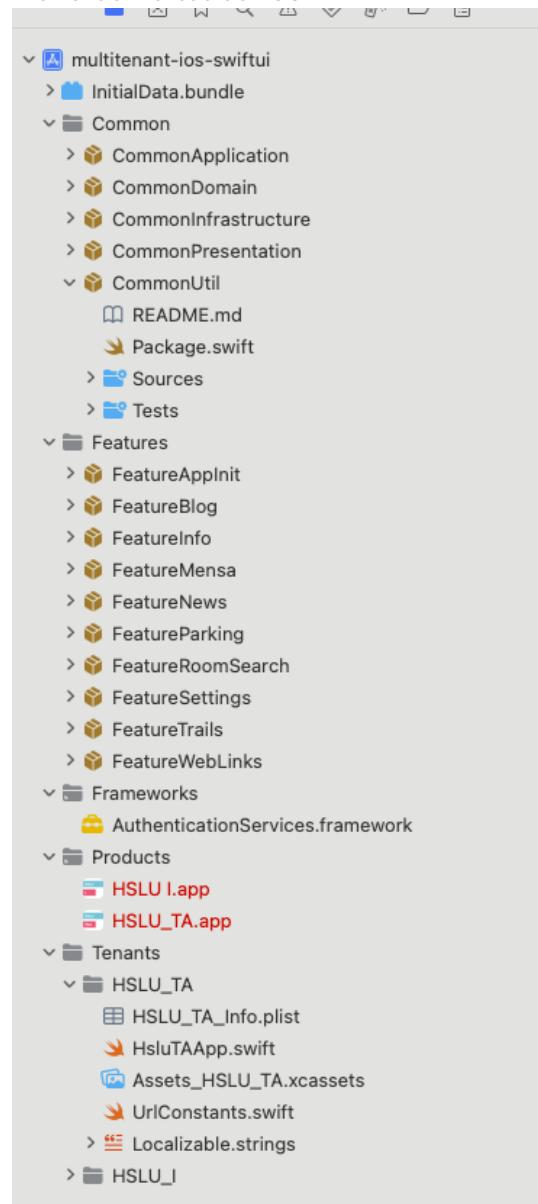
Risiko	Massnahme(n)
Trotz eingeplanter Zeit besteht die Gefahr, dass Inhalte inkonsistent, zu wenig tief oder nicht vollständig gemäss HSLU-Vorgaben ausgearbeitet sind.	<ul style="list-style-type: none"> • Klare Aufteilung der Kapitel • Regelmässige interne Reviews • Abgleich mit Bewertungsraster und 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

A.2.11. Aufgabenstellung für AI Tools

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:

```

1 // swift-tools-version: 5.0
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: "hslui " + 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:Boolean) 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])
// AppNewsModuleItemAPI DTO
{
    var remoteSyncData : (CommonAppBaseModuleSyncAPI DTO?, 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
(CommonAppBaseModuleSyncAPI DTO?, 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!)
                        }
                }
            
```

```
        {
            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?

A.2.12. 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.

A.2.13. 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. ApplInit, 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 (FeatureApplInit, 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 : CommonAppBaseModuleItemAPI> :
ViewModel() {

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

 // ... Weitere Properties

 suspend fun syncData(): Triple<Boolean, String, List<I>> {
 var remoteSyncData: Pair<CommonAppBaseModuleSyncAPI?, 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!

#### A.2.14. 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)
    |   ├── FeatureAppInit
    |   ├── FeatureBlog
    |   ├── FeatureMensa
    |   └── ...
    └── Tenants/ (Multi-Tenancy)
        ├── HSLU/
        └── HSLU_TA/
```

```

### \*\*2.2 Multi-Tenancy-Implementierung\*\*

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

### ### \*\*2.3 Technische Besonderheiten\*\*

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

---

## ## \*\*3. Architektur-Entwurf für Android\*\*

### ### \*\*3.1 Gesamtarchitektur\*\*

\*\*Domain-Driven Design (DDD) mit Clean Architecture:\*\*

- \*\*Domain Layer\*\*: Entities, Use Cases, Repository Interfaces
- \*\*Data Layer\*\*: Repositories, Data Sources (Remote, Local)
- \*\*Presentation Layer\*\*: ViewModel + Jetpack Compose

### ### \*\*3.2 Modulstruktur (Android)\*\*

```

app/

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

### ### \*\*3.3 Multi-Tenancy-Architektur\*\*

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

---

## ## \*\*4. Technische Umsetzungsstrategie\*\*

### ### \*\*4.1 Technologie-Stack\*\*

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

### ### \*\*4.2 Daten-Synchronisation\*\*

```kotlin

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

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

```

## 5. Detaillierter Migrationsplan

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

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

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

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

```

## 6. Risikoanalyse und Herausforderungen

### ### \*\*6.1 Technische Risiken\*\*

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

### ### \*\*6.2 Lösungsansätze\*\*

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

---

## ## \*\*7. Erfolgskriterien\*\*

### ### \*\*7.1 Funktionale Anforderungen\*\*

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

### ### \*\*7.2 Nicht-funktionale Anforderungen\*\*

- Saubere Architektur (DDD)
- Wartbarer und erweiterbarer Code
- Umfassende Testabdeckung
- Google Play Store Guidelines konform

---

## ## \*\*8. Empfehlungen\*\*

### ### \*\*8.1 Technische Entscheidungen\*\*

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

### ### \*\*8.2 Entwicklungsprozess\*\*

- \*\*Agile Methodik\*\* mit 2-wöchigen Sprints
- \*\*Code Reviews\*\* für Qualitätssicherung
- \*\*CI/CD Pipeline\*\* für automatisiertes Testing und Deployment

---

## ## \*\*9. Zusammenfassung\*\*

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

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

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

---

**\*\*Dokumentversion\*\*: 1.0**

**\*\*Erstellt am\*\*: \$(date)**

**\*\*Autor\*\*: AI-Assistent**

**\*\*Status\*\*: Entwurf zur Diskussion**

---

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

## A.2.15. 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.

# 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 . . . . .	27
8.	Abhängigkeiten in Pfeilrichtung erlaubt/vorhanden . . . . .	27
9.	Struktur der Features am Beispiel <i>Event</i> . . . . .	28
10.	Aufforderung zum Versionsupdate . . . . .	30
11.	Flussdiagramm Menü-Synchronisation . . . . .	31
12.	Screenshots des News-Features . . . . .	33
13.	Screenshots des Blog-Features . . . . .	34
14.	Screenshot aus dem Backend: Umschalten der Views (Blog & News) . . . . .	34
15.	Screenshots des Mensa-Features . . . . .	35
16.	Screenshots des Events-Features . . . . .	36
17.	Ablaufdiagramm . . . . .	40
18.	Screenshots des Kalender-Features . . . . .	44
19.	Screenshots des Kalender-Widgets . . . . .	46
20.	Analysebeispiel in Cursor . . . . .	51
21.	Analysebeispiel in Cursor . . . . .	52
22.	Analysebeispiel in Cursor . . . . .	52
23.	Analysebeispiel in Cursor . . . . .	53

## Tabellenverzeichnis

## KI-Verzeichnis

## Formelverzeichnis

# Literaturverzeichnis