

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

17. 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

Expertenperson: 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 _____

*Geistiges Eigentum gemäss der Studienordnung für die Ausbildung
an der Hochschule Luzern, FH Zentralschweiz*

I. Abstract

Im Rahmen des Wirtschaftsprojekts *HSLU Mobile Apps – Android Jetpack Compose – AI First* wurde eine bestehende Android-Applikation der Hochschule Luzern technisch neu umgesetzt. Ausgangslage war eine veraltete Android-App auf Basis von XML-Layouts sowie eine moderne iOS-Applikation, welche als Referenz diente. Ziel des Projekts war es, eine zeitgemäße Android-Applikation mit Jetpack Compose zu realisieren, die funktional an die iOS-Version angeglichen ist, aktuelle Android-Technologien nutzt und produktiv im Google Play Store veröffentlicht werden kann.

Die Umsetzung erfolgte über ein agiles Vorgehen mit zweiwöchigen Sprints. Zu Projektbeginn wurde gemeinsam mit dem Auftraggeber ein klarer Rahmen definiert, inklusive regelmässiger Status-Meetings und transparenter Aufgabenverwaltung über GitLab. Die Entwicklung wurde modular aufgebaut. Gemeinsame, fachlich unabhängige Komponenten wurden in zentralen common-Modulen umgesetzt, während einzelne Features wie News, Blog, Mensa, Events oder Raumsuche jeweils als eigene Module realisiert wurden. Diese Struktur wurde dabei von der bestehenden iOS-Applikation übernommen, um eine einheitliche Architektur über beide Plattformen hinweg sicherzustellen.

Ein Schwerpunkt des Projekts lag auf dem AI-First-Ansatz. KI-gestützte Entwicklungswerkzeuge wurden getestet und dann aktiv in den Entwicklungsprozess eingebunden. AI wurde unter anderem für Code-Analysen, Übersetzungen von bestehendem iOS-Code nach Kotlin, Unterstützung bei neuen Features sowie für das Testing eingesetzt. Die Erfahrungen zeigen, dass AI insbesondere bei klar abgegrenzten Aufgaben und gut formulierten Prompts eine deutliche Zeitersparnis ermöglicht, gleichzeitig aber weiterhin fundierte Programmierkenntnisse notwendig bleiben.

Als Resultat steht eine moderne, modular aufgebaute Android-Applikation auf Basis von Jetpack Compose zur Verfügung, die Multi-Tenant-fähig ist, aktuelle Android-SDKs unterstützt und automatisiert über CI/CD und Fastlane ausgeliefert werden kann. Ergänzend dazu wurden alle Projektartefakte wie Dokumentation, Risikoanalyse, Sprint-Backlogs, Statusberichte und Protokolle vollständig erstellt, sodass eine Weiterentwicklung des Projekts problemlos möglich ist.

Inhaltsverzeichnis

I. 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 Praxis und Technik	2
2.1. Apps	2
2.2. Backend	3
3. Ideen und Konzepte	4
3.1. Konzept zur Übernahme der iOS-Applikation auf Android	4
3.2. Technologien und Architekturkonzept	4
3.3. Multi-Tenant- und Backend-Konzept	5
3.4. Konzept für Testing und Qualität	5
3.5. Geplantes Vorgehen zum AI-First-Ansatz	6
3.6. Abgrenzung	6
4. Methoden	7
4.1. Vorgehen und gewünschte Methoden	7
4.2. Kreativität und Innovation	7
4.3. Domain Driven Design	7
4.4. Projektmanagement	8
4.4.1. Stakeholder	8
4.4.2. Planung	8
4.4.3. Agiles Vorgehen	9
4.4.4. Eingesetzte Programme/Tools	10
4.4.5. Risikomanagement	11
5. Realisierung	12
5.1. Applikationsarchitektur und Konfiguration	12
5.1.1. Projektkonfiguration und Komponenten	12
5.1.2. SDK-Konfiguration	13
5.1.3. App-Modul	13
5.1.4. Lokalisierung	15
5.1.5. Fastlane	16
5.2. Common-Komponenten	18
5.2.1. CommonApplication	18
5.2.2. CommonDomain	20
5.2.3. CommonInfrastructure	22
5.2.4. CommonPresentation	26
5.2.5. CommonUtil	27
5.3. Features	28
5.3.1. AppInit	30
5.3.2. News	33
5.3.3. Blog	35
5.3.4. Mensa	36
5.3.5. Events	37
5.3.6. Info-, Einstellungen- und WebLinks-Features	38

5.3.7. CampusRoomSearch	39
5.3.8. Timetable	42
6. Validierung und Evaluation	46
6.1. Validierung	46
6.1.1. Technische Validierung	46
6.1.2. Validierung der Anforderungen und Projektziele	47
6.1.3. Validierung des Nutzens	47
6.2. Evaluation AI Tools	48
6.2.1. Aufgabenstellung für die AI Tools	48
6.2.2. Auswertung	48
6.2.3. Vergleich	49
6.2.4. Detaillierter Fragen zu Cursor	49
6.2.5. Fazit	50
6.2.6. Erfahrungen mit AI-First Ansatz	50
7. Ausblick	52
7.1. Reflexion	52
7.1.1. Fazit AI-First Programmierung	52
7.1.2. Team-Fazit	53
7.1.3. Persönliche Reflexion Raphael	54
7.1.4. Persönliche Reflexion Samuel	55
7.2. Ausblick	55
7.2.1. Persönlicher Ausblick	55
7.2.2. Mögliche weiterführende Arbeiten	55
8. Anhänge	56
8.1. Aufgabenstellung	56
8.2. Zusatzmaterial	58
8.2.1. AI-First: Weitere Inhalte	58
8.2.2. AppBootstrappingViewModel	59
8.2.3. CommonApplicationBaseModuleLoader	60
8.2.4. Sprintboard – Sprint 1	61
8.2.5. Sprintboard – Sprint 2	62
8.2.6. Sprintboard – Sprint 3	63
8.2.7. Sprintboard – Sprint 4	64
8.2.8. Sprintboard – Sprint 5	65
8.2.9. Sprintboard – Sprint 6	66
8.2.10. Sprintboard – Sprint 7	67
8.2.11. Zeitmanagement - Meilensteine	68
8.2.12. Risikoanalyse - Risikomatrix	69
8.2.13. Aufgabenstellung für AI Tools	71
8.2.14. Antwort von ChatGPT	80
8.2.15. Antwort von Grok	82
8.2.16. Antwort von Deepseek	87
8.2.17. Antwort von Cursor	92
8.2.18. Zeiterfassung	93
8.2.19. Backend	94
8.3. Protokolle und Statusberichte	95
8.3.1. Kickoff-Meeting (19.09.2025)	96
8.3.2. Projektstatus-Meeting (25.09.2025)	98
8.3.3. Projektstatus-Meeting (16.10.2025)	100
8.3.4. Projektstatus-Meeting (30.10.2025)	102

8.3.5. Projektstatus-Meeting (13.11.2025)	104
8.3.6. Projektstatus-Meeting (11.12.2025)	106
8.3.7. Status-Report (16.10.2025)	107
8.3.8. Status-Report (30.10.2025)	108
8.3.9. Status-Report (13.11.2025)	110
8.3.10. Status-Report (11.12.2025)	112
9. Abkürzungs-, Abbildungs-, Tabellen-, KI-, Formelverzeichnis	114
9.1. Abkürzungsverzeichnis	114
9.2. Abbildungsverzeichnis	114
9.3. Tabellenverzeichnis	115
9.4. KI-Verzeichnis	115
9.5. Formelverzeichnis	115
10. Literaturverzeichnis	116

1. Problem, Fragestellung, Vision

1.1. Ausgangslage und Problemstellung

Die Hochschule Luzern betreibt für die Departemente Informatik sowie Technik & Architektur eigene mobile Applikationen auf den Plattformen iOS und Android. Während die iOS-Applikation in den vergangenen Semestern weiterentwickelt wurde, befindet sich die Android-Version aktuell in einem teilweise veralteten Zustand.

Auf Android existiert einerseits eine veröffentlichte Applikation auf Basis klassischer XML-Layouts, welche weder an das aktuelle Backend-API angebunden ist noch alle Features der iOS-Version enthält. Andererseits liegt eine nicht veröffentlichte Jetpack-Compose-Applikation vor, die zwar bereits auf einem deklarativen UI-Ansatz basiert, jedoch ebenfalls nicht mehr dem aktuellen Stand der Technik entspricht und funktional hinter der iOS-Version zurückbleibt.

Diese Situation führt dazu, dass der Entwicklungsstand der Android-Applikationen weit hinter der iOS-Applikationen zurückliegt. Eine konsistente App-Landschaft über beide Plattformen hinweg ist nicht gegeben. Gleichzeitig besteht der Anspruch, neue Features, die auf iOS bereits umgesetzt wurden, auch auf Android bereitzustellen. Vor diesem Hintergrund ergibt sich das Bedürfnis, die bestehende Android-Jetpack-Compose- Applikation grundlegend zu modernisieren und funktional an die iOS-Applikationen anzugeleichen.

1.2. Ziel der Arbeit und erwartete Resultate

Ziel dieses Projekts ist die Umsetzung einer modernen Android-Applikation auf Basis von Jetpack Compose, welche an die bestehende iOS-App der Hochschule Luzern angeglichen ist. Die Applikation soll aber auch Android- Technologien und Architekturkonzepte nutzen, wartbar und erweiterbar aufgebaut sein, um dann produktiv über den Google Play Store veröffentlicht werden zu können.

Die Umsetzung soll als modulare Single-Codebase mit Unterstützung für mehrere Features und Mandanten (Multi-Tenant) erfolgen. Die Architektur soll sich an der bestehenden iOS-Applikation orientieren, um eine möglichst einheitliche Plattformstruktur zu erreichen, damit künftige Arbeiten auf beiden Plattformen einfacher zu erledigen sind.

Ein weiterer zentraler Aspekt der Arbeit soll der Einsatz eines AI-First-Ansatzes sein. Ziel ist es, verschiedene KI-gestützte Entwicklungswerzeuge im Kontext der mobilen App-Entwicklung zu evaluieren und deren Einsatzmöglichkeiten, Grenzen und Mehrwerte praktisch zu untersuchen. Die gewonnenen Erkenntnisse sollen dokumentiert und reflektiert werden. (Die vollständige Aufgabenstellung ist zu finden im Anhang: 8.1).

Die erwarteten Resultate lassen sich wie folgt zusammenfassen:

- **App-Artefakte:**
 - Eine produktiv einsetzbare Android-Applikation auf Basis von Jetpack Compose
 - Umsetzung und Integration der fehlenden bzw. neu aufgebauten Features
 - Anbindung an das aktuelle Backend-API
 - Unterstützung aktueller Android-SDKs und Geräte
 - Automatisierter Build- und Release-Prozess mittels GitLab CI/CD und Fastlane
- **Projektmanagement-Artefakte:**
 - Sprint-basierte Planung und Umsetzung mit dokumentierten Issues und Epics
 - Regelmässige Statusberichte und Sitzungsprotokolle
 - Fortlaufende Risikoanalyse und Meilensteinübersicht
- **Dokumentation und Evaluation:**
 - Projektdokumentation gemäss HSLU-Standards
 - Evaluation und Reflexion des AI-First-Ansatzes im Entwicklungsprozess
 - Zwischen- und Schlusspräsentation der Projektergebnisse

2. Stand der Praxis und Technik

2.1. Apps

Die Hochschule Luzern betreibt für die Departemente Informatik sowie Technik & Architektur eigene mobile Applikationen auf den Plattformen iOS und Android. In den vergangenen Wirtschaftsprojekten und Forschungsarbeiten wurden diese Apps laufend weiterentwickelt. Dabei arbeiteten jeweils unterschiedliche Studierende an den Projekten, jedoch nicht immer parallel auf beiden Plattformen. Aus diesem Vorgehen sind mehrere Projekte entstanden, die sich heute in unterschiedlichen Entwicklungsständen befinden. Alle Projekte werden zentral auf GitLab verwaltet und versioniert.

Aktuell existieren drei unterschiedliche Versionen der Applikation:

- ios-swiftui-multitenant
- android-xml-multitenant
- android-jetpackcompose-multitenant

	A	android-jetpackcompose-multitenant	Erstellt 08.02.2024	⋮
	A	android-xml-multitenant	Erstellt 08.02.2024	⋮
	I	ios-swiftui-multitenant	Erstellt 08.02.2024	⋮

Abbildung 1.: Projekte in GitLab

Die Projekte **ios-swiftui-multitenant** sowie **android-xml-multitenant** verfügen jeweils über eine produktive Version der Applikation, welche im Apple App Store respektive im Google Play Store veröffentlicht ist. (Diese produktiven Stände sind in der Abbildung 2 grün markiert.)

Zusätzlich existiert im Projekt **ios-swiftui-multitenant** ein Entwicklungsstand (*Dev-Branch*), der bereits weitere Features enthält, jedoch noch nicht veröffentlicht wurde. Dieser Entwicklungsstand stellt aktuell den funktional vollständigsten und modernsten Stand der Applikation dar. (Er ist in der Abbildung 2 blau dargestellt und dient als Referenz für die Android-Neuentwicklung.)

Das Projekt **android-jetpackcompose-multitenant** ist aktuell nicht produktiv. An diesem Projekt wurde seit längerer Zeit nicht mehr aktiv gearbeitet, weshalb es technisch als veraltet betrachtet werden muss. Ziel des Projekts ist es, den aktuellen Entwicklungsstand aus dem **ios-swiftui-multitenant**-Dev-Branch funktional auf Android mit Jetpack Compose nachzubauen und weiterzuentwickeln.

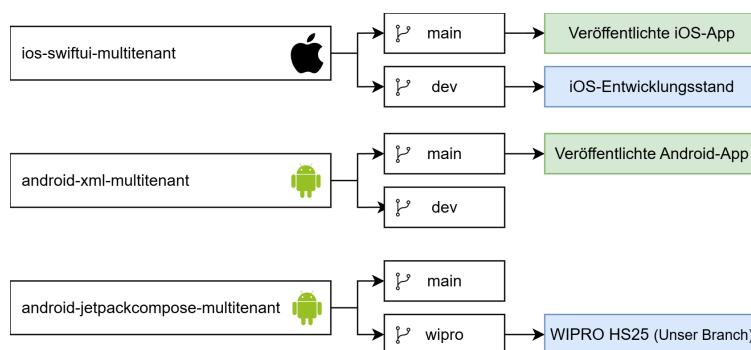


Abbildung 2.: Projekte mit (relevanten) Branches

Die folgenden Abbildungen zeigen den Stand der bestehenden mobilen Applikationen im September 2025, also zum Zeitpunkt vor Projektbeginn.

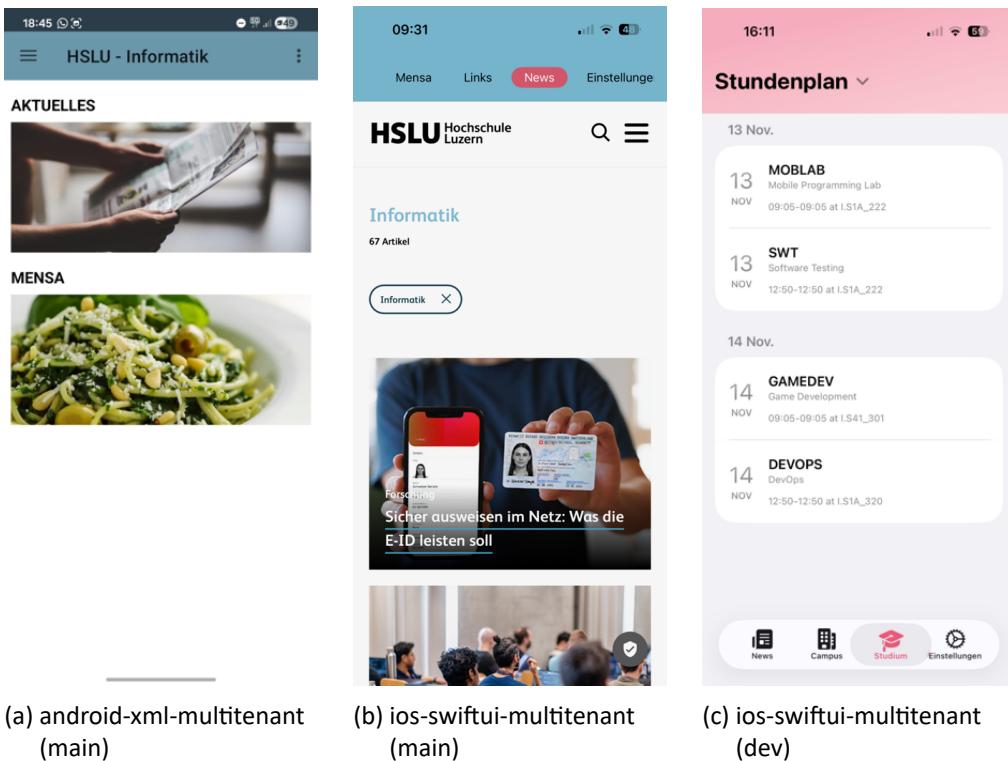


Abbildung 3.: Screenshots der bestehenden Apps

2.2. Backend

Für alle Applikationen existiert ein gemeinsames Backend, auf das sowohl die iOS- als auch die Android-Apps zugreifen. Über dieses Backend werden unter anderem Informationen zu den einzelnen Modulen geladen.

Dieses Konzept ermöglicht eine Entkopplung zwischen App und Inhalt. Änderungen an Daten oder Modulinhalten können zentral im Backend vorgenommen werden, ohne dass die Applikationen neu veröffentlicht werden müssen. Dadurch wird die Wartbarkeit verbessert und neue Inhalte können schneller und flexibler bereitgestellt werden (siehe Anhang 8.2.19).

3. Ideen und Konzepte

3.1. Konzept zur Übernahme der iOS-Applikation auf Android

Die Android-Applikation soll auf Basis von Jetpack Compose funktional auf denselben Stand gebracht werden wie die aktuelle Dev-Version der iOS-Applikation. Die iOS-App dient dabei aus unserer Sicht als vollständige Referenz, sowohl in Bezug auf den Funktionsumfang als auch auf den grundsätzlichen Aufbau der Applikation. Ziel ist es, dass beide Plattformen möglichst ähnlich strukturiert sind, um zukünftige Arbeiten zu vereinfachen und den Einstieg für neue Entwicklerinnen und Entwickler zu erleichtern.

Eine vollständige 1:1-Übernahme ist dabei nicht in allen Bereichen möglich, da sich iOS und Android sowie die Programmiersprachen Swift und Kotlin in einigen Aspekten unterscheiden. Dennoch sollen das grundlegende Vorgehen, die Projektstruktur sowie die Benennung von Modulen und Komponenten so weit wie möglich angeglichen werden. Wie bereits bei der iOS-Applikation soll auch die Android-App klar in Common-Komponenten und einzelne Feature-Module unterteilt werden, damit fachlich unabhängige Logik zentral wiederverwendet werden kann.

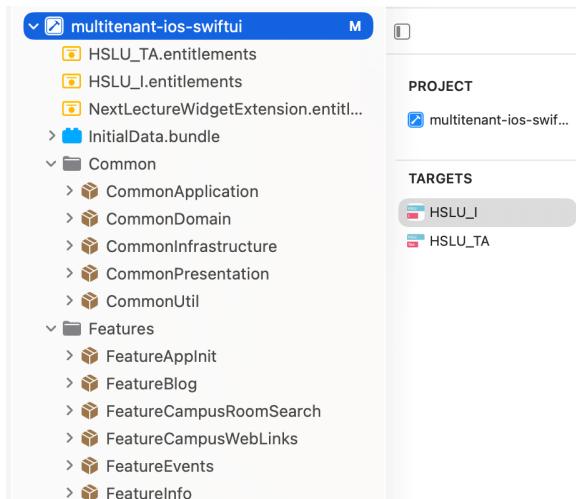


Abbildung 4.: iOS-Projektstruktur in XCode

3.2. Technologien und Architekturkonzept

Ein weiteres Ziel ist es, moderne Android-Technologien einzusetzen und veraltete Komponenten zu ersetzen. Dabei soll aber nicht alles Bestehende pauschal ersetzt werden. Android-Frameworks und Komponenten, die in früheren Hochschul-Apps bereits eingesetzt wurden und weiterhin aktuell sowie empfohlen sind, sollen nach Möglichkeit weiterverwendet werden. Wo sinnvoll, werden diese Komponenten auf den neuesten Stand gebracht und aktualisiert, anstatt sie komplett neu zu implementieren.

Im Rahmen des Projekts kommen deshalb unter anderem folgende Android-Technologien zum Einsatz:

- **Jetpack Compose:** Deklaratives UI-Framework für moderne Benutzeroberflächen.
- **Kotlin Coroutines & Flow:** Vereinfachte asynchrone Verarbeitung und reaktiver Umgang mit Datenströmen.
- **Room Persistence:** Strukturierte und typsichere lokale Datenhaltung.
- **Hilt (Dependency Injection):** Klare Verwaltung von Abhängigkeiten und bessere Testbarkeit.

Auch auf Architekturebene soll ein sauberer Aufbau verfolgt werden, der sich auch teilweise an der bestehenden iOS-Lösung orientiert. Dazu werden bekannte Architektur- und Design-Patterns eingesetzt, um die App übersichtlich und erweiterbar zu halten. Dazu zählen unter anderem:

- **Clean Architecture:** Klare Trennung von fachlicher Logik, Infrastruktur und Darstellung.
- **Repository Pattern:** Einheitlicher Zugriff auf Netzwerk- und lokale Datenquellen.
- **Feature-basierte Modulstruktur:** Jedes Feature ist als eigenes Modul umgesetzt, gemeinsame Logik liegt in zentralen Common-Modulen.
- **Template- und Strategy-Pattern:** Flexible und wiederverwendbare Module-Loader für unterschiedliche Datenlade-Strategien.

Zusätzlich wird bei der Umsetzung der Benutzeroberfläche auf moderne UI-Patterns gesetzt, die gut mit Jetpack Compose zusammenspielen. Ziel ist eine reaktive, wiederverwendbare und gut wartbare UI, die sich flexibel an unterschiedliche Features anpassen lässt.

Konkret kommen dabei unter anderem folgende UI-Konzepte zum Einsatz:

- **Material Design 3:** Verwendung moderner Material-Komponenten mit einheitlichem Theming über die gesamte App.
- **Reaktives UI-Management:** Einsatz von StateFlow in Kombination mit `collectAsState()`, um UI-Zustände reaktiv und nachvollziehbar abzubilden. LaunchedEffect wird für Side-Effects in Compose verwendet.
- **Component-Based UI:** Wiederverwendbare Composables werden im Common/Presentation-Modul abgelegt. Styling erfolgt über das Modifier-Pattern, Interaktionen über Lambda-Callbacks, um eine flexible Nutzung in verschiedenen Features zu ermöglichen.

Abschliessend soll die Android-Applikation über einen automatisierten Build- und Release-Prozess produktiv im Google Play Store veröffentlicht werden. Dafür wird Fastlane eingesetzt, um Releases reproduzierbar und zuverlässig auszuliefern.

3.3. Multi-Tenant- und Backend-Konzept

Die bestehende App-Landschaft der Hochschule Luzern basiert bereits auf einem Multi-Tenant-Ansatz, welcher im Rahmen dieses Projekts weiterverfolgt wird. Ziel ist es, mit einer gemeinsamen Codebasis mehrere Applikationen für unterschiedliche Departemente zu betreiben. Aktuell betrifft dies die Departemente Informatik (I) sowie Technik & Architektur (TA). Aus derselben Codebasis können dadurch mehrere App-Varianten erzeugt werden, die sich in deren Inhalten unterscheiden.

Die Inhalte der Applikation werden nicht fest in der App hinterlegt, sondern zur Laufzeit aus einem zentralen Backend geladen. Dazu gehören unter anderem Menüstruktur, Inhalte oder Texte. Dieser Ansatz ermöglicht es, Inhalte anzupassen oder zu erweitern, ohne dass dafür ein neues App-Update im App Store oder Play Store veröffentlicht werden muss.

3.4. Konzept für Testing und Qualität

Die Qualität der Applikation soll durch den Einsatz von automatisierten Tests sichergestellt werden. In den bisherigen iOS- und Android-Projekten wurde das Thema Testing etwas vernachlässigt, was im Rahmen dieses Projekts verbessert werden soll.

Geplant ist der Einsatz von Unit- sowie einfachen Integrations- und UI-Tests, um zentrale Funktionen der App abzusichern. Dafür werden unter anderem Frameworks wie JUnit und Espresso eingesetzt. Die konkrete Umsetzung und Erfahrungen damit werden im Kapitel zur technischen Validierung beschrieben (siehe Abschnitt 6.1.1).

3.5. Geplantes Vorgehen zum AI-First-Ansatz

Für den geplanten AI-First-Ansatz soll künstliche Intelligenz bewusst und aktiv in den Entwicklungsprozess eingebunden werden. Ziel ist es nicht, die Entwicklung vollständig zu automatisieren, sondern AI als unterstützendes Werkzeug im Arbeitsalltag der Entwickler zu nutzen.

AI soll insbesondere dort eingesetzt werden, wo sie einen praktischen Mehrwert bietet. Dazu wird zu Beginn des Projekts eine erste Evaluationsphase durchgeführt (siehe Abschnitt 6.2). In dieser Phase wird geprüft, welche AI-Tools sich sinnvoll in den Entwicklungsprozess integrieren lassen und welche davon tatsächlich eine Unterstützung im Alltag bieten.

AI-Tools, die sich in dieser Evaluationsphase als hilfreich erweisen, werden während der weiteren Entwicklung aktiv eingesetzt. Sie sollen den Entwicklungsprozess begleiten, etwa beim Verständnis von bestehendem Code, bei der Umsetzung einzelner Features oder bei wiederkehrenden Aufgaben.

Zum Abschluss des Projekts wird ein Fazit zum AI-First-Ansatz gezogen. Dabei wird festgehalten, welche Tools sich gut bewährt haben, wo deren Stärken und Grenzen liegen und in welchen Situationen ihr Einsatz sinnvoll ist. Ziel ist es, diese Erkenntnisse weiterzugeben und anderen Entwicklerinnen und Entwicklern aufzuzeigen, wo sich der Einsatz von AI im aktuellen Stand der Technik lohnt.

3.6. Abgrenzung

Dieses Projekt fokussiert sich ausschliesslich auf die technische Umsetzung der Android-Applikation auf Basis der bestehenden iOS-App. Die iOS-Version dient dabei als funktionale Referenz und wird mit Jetpack Compose auf Android nachgebaut und modernisiert.

Es wurden keine Nutzungstests oder Befragungen mit Studierenden durchgeführt. Ebenso war eine Anpassung des bestehenden Backends nicht Teil des Projekts.

4. Methoden

4.1. Vorgehen und gewünschte Methoden

In diesem Abschnitt wird kurz beschrieben, welche Vorgehensweise und Methoden bereits vor Projektstart gemeinsam mit dem Auftraggeber festgelegt wurden. Die Aufgabenstellung sah ein inkrementelles, iteratives und agiles Vorgehen vor (siehe die Aufgabenstellung unter Abschnitt 8.1). Dieses Vorgehen wurde zu Beginn des Projekts gemeinsam besprochen und als passend für den Projektumfang eingeschätzt.

Konkret wurde vereinbart, alle zwei Wochen ein Meeting mit dem Auftraggeber durchzuführen. In diesen Meetings wurde jeweils der aktuelle Projektstand vorgestellt, offene Punkte und Risiken besprochen sowie das weitere Vorgehen abgestimmt. Zusätzlich sollte der Projektfortschritt jederzeit transparent nachvollziehbar sein. Dafür wurde GitLab als zentrales Tool zur Aufgabenverwaltung verwendet, in dem der aktuelle Stand laufend gepflegt wurde. Dieses Vorgehen hat sich über die gesamte Projektdauer bewährt und wurde konsequent umgesetzt.

Die regelmässigen Meetings waren insgesamt sehr hilfreich und konstruktiv. Durch den engen Austausch konnten Anforderungen frühzeitig überprüft und bei Bedarf angepasst werden. Auf diese Weise fand bereits während der Entwicklung eine formale Validierung der Anforderungen statt (siehe Abschnitt 6.1.2). Alle Sitzungsprotokolle und Statusberichte sind im Anhang dokumentiert (siehe Anhang 8.3).

4.2. Kreativität und Innovation

Ein weiterer Schwerpunkt des Projekts lag auf dem Einsatz moderner Technologien und Entwicklungsansätze. Vorgesehen war unter anderem die Evaluation aktueller Android-Technologien wie Jetpack Compose sowie der Einsatz von CI/CD-Automatisierungen und AI-gestützten Entwicklungswerkzeugen.

Diese Vorgaben wurden im Projekt aktiv umgesetzt. Die App wurde auf Basis aktueller Bibliotheken und SDK-Versionen entwickelt, sodass auch moderne Geräte und neue Android-Versionen unterstützt werden. Die Entwicklung folgte dabei einem *AI-first*-Ansatz, bei dem KI-Tools gezielt zur Unterstützung bei Architekturentscheidungen, Code-Erstellung, Refactorings und Dokumentation eingesetzt wurden. Die Nutzung dieser Werkzeuge wurde laufend reflektiert und je nach Projektsituation angepasst, um einen sinnvollen Mehrwert für die Entwicklung zu erzielen (siehe 7.1.1).

4.3. Domain Driven Design

Domain Driven Design (DDD) war als Grundlage für die Projektstruktur vorgegeben. Ziel war es, die fachlichen Domänen der App klar zu trennen und im Code sauber abzubilden.

Die Anwendung ist in klar abgegrenzte Bereiche unterteilt, wobei gemeinsame und fachlich unabhängige Komponenten in common-Modulen liegen und fachspezifische Logik in den jeweiligen features-Modulen umgesetzt ist. Jedes Feature bildet dabei eine eigene fachliche Domäne ab. Diese Struktur war bereits in der bestehenden iOS-Applikation umgesetzt und wurde für das Android-Projekt bewusst in gleicher Form übernommen.

4.4. Projektmanagement

4.4.1. Stakeholder

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

4.4.2. Planung

Um zu Beginn einen Überblick über das Projektmanagement zu erhalten, wurden alle bekannten Termine wie Abgaben, Präsentationen und Meilensteine in einer Excel-Tabelle gesammelt. Diese diente während der gesamten Projektdauer als eine Art Meilensteinplan und half dabei, den zeitlichen Rahmen des Projekts übersichtlich darzustellen.

Auf dieser Grundlage konnten die anstehenden Arbeiten sinnvoll in einzelne Sprints aufgeteilt werden. Die Excel-Übersicht wurde insbesondere zu Beginn des Projekts genutzt, während die weitere Detailplanung und Aufgabenverwaltung anschliessend agil über GitLab erfolgte.

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

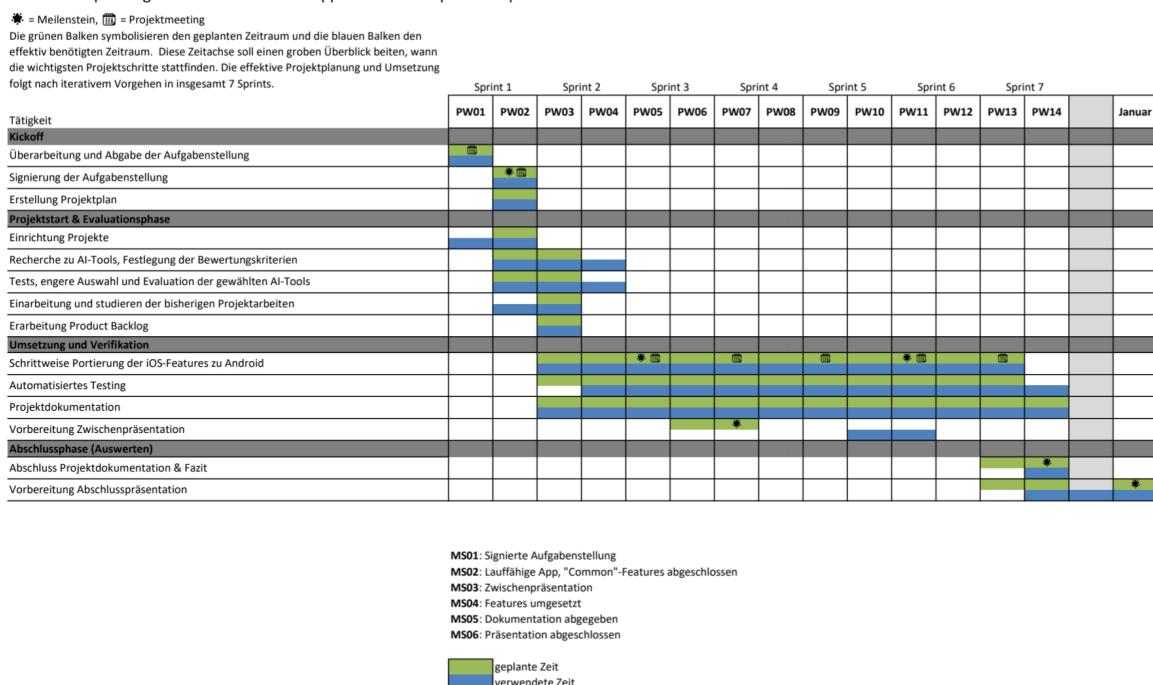


Abbildung 5.: Meilensteinplan des Projekts

4.4.3. Agiles Vorgehen

Die Umsetzung des Projekts erfolgte nach einem agilen Vorgehen, angelehnt an Scrum. Die Arbeit wurde in zweiwöchige Sprints unterteilt, woraus sich insgesamt sieben Sprints ergaben. Vor jedem Sprint wurde im Team gemeinsam besprochen, welche Aufgaben im kommenden Zeitraum umgesetzt werden sollen und wer welche Arbeiten übernimmt.

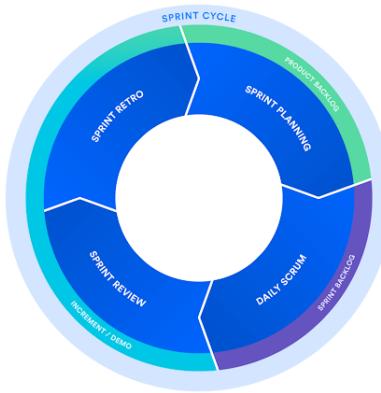


Abbildung 6.: Darstellung des Scrum-Prozesses im Projekt
(Atlassian, n. d.)

Dieses Vorgehen ermöglichte ein effizientes Arbeiten, da Aufgaben flexibel verteilt und bei freier Kapazität jederzeit neue Issues übernommen werden konnten. Da alle Aufgaben und der aktuelle Projektstand in GitLab erfasst waren, hatte auch der Auftraggeber jederzeit Einblick in den Fortschritt des Projekts.

Zur Nachvollziehbarkeit des Projektverlaufs sind die Sprint-Backlogs als grafische Übersicht im Anhang dokumentiert und zeigen den Fortschritt über die gesamte Projektdauer hinweg (siehe 8.2.4).

Abbildung 7.: Backlogs in GitLab zu Projektbeginn

4.4.3.1. Issues

Alle Aufgaben im Projekt wurden als Issues in GitLab erfasst. Diese wurden meist als einfache User Stories formuliert und mit einer kurzen *Definition of Done* (DoD) ergänzt. Dadurch war von Beginn an klar, was zu einer Aufgabe gehört und wann sie als abgeschlossen gilt. Dies half, die Anforderungen besser zu verstehen und Missverständnisse zu vermeiden.

Für jede Aufgabe wurde die aufgewendete Zeit direkt im jeweiligen Issue dokumentiert. So konnte der Arbeitsaufwand über die gesamte Projektlaufzeit hinweg nachvollzogen werden und es war jederzeit ersichtlich, wie viel Zeit in welche Themen investiert wurde (siehe Anhang 8.2.18).

The screenshot shows a GitLab issue card for a feature request titled "Raumsuche / Roomsearch". The card includes a "Status" section with "To do" selected, a "Beauftragte" section with "Keine - weise dich selbst zu", a "Labels" section with "Feature", and an "Übergeordnet" section with "Keine". The main text area describes the user story: "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." Below this, a bulleted list details the implementation: "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.", and "Informationsbildschirm für ausgewählten Raum dargestellt.". At the bottom, there are buttons for "Design hinzufügen" and "Erstelle Merge Request".

Abbildung 8.: Beispiel einer User Story mit Definition of Done (DoD)

4.4.4. Eingesetzte Programme/Tools

Während der Projektdauer wurden diese Programme und Tools genutzt, um die Entwicklung, Dokumentation und die Organisation des Projekts zu unterstützen.

Werkzeug	Einsatz im Projekt
Git	Versionsverwaltung des Quellcodes
GitLab	Issue-Tracking, Sprint-Planung und Projektübersicht
OneNote	Persönliche Notizen, Skizzen und Screenshots während der Entwicklung
OneDrive	Zentrale Ablage für Projektdokumente und organisatorische Dateien
Latex	Erstellung und Formatierung der Projektdokumentation
VS Code	Editor zur Erstellung der Dokumentation
GitHub	Hosting des Dokumentations-Quellcodes
Android Studio	Entwicklungsumgebung für die Android-Applikation
Cursor	Entwicklungsumgebung mit AI-Features
Microsoft Teams	Kommunikation im Team und Meetings mit dem Auftraggeber

Tabelle 4.1.: Eingesetzte Werkzeuge im Projekt

4.4.5. Risikomanagement

Zu Beginn des Projekts wurde eine Risikoanalyse erstellt, um die grössten Risiken frühzeitig zu identifizieren und entsprechende Gegenmassnahmen zu planen. Die Risiken wurden jeweils 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 verbleibende Risiken zu identifizieren und ihre Entwicklung über die Projektlaufzeit zu beobachten.

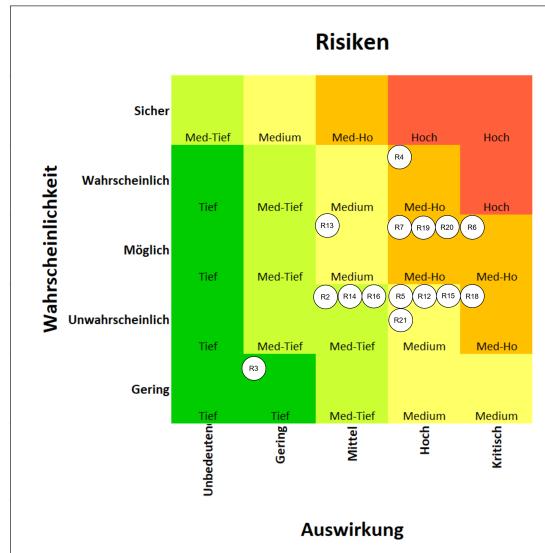


Abbildung 9.: Risikoanalyse und Risikomatrix des Projekts

Die Risikoanalyse wurde während der gesamten Projektdauer regelmässig überprüft und aktualisiert. Am Ende jedes Sprints wurde die aktuelle Risikosituation besprochen und mögliche Massnahmen eingeleitet. Die drei grössten Risiken wurden zusätzlich an den Auftraggeber übermittelt, um Transparenz über den Projektfortschritt und potenzielle Herausforderungen sicherzustellen. Die gesamte Risikoanalyse ist im Anhang zu finden (siehe Anhang 8.2.12).

5. Realisierung

5.1. Applikationsarchitektur und Konfiguration

Das Ziel der Entwicklung war es, eine modulare und erweiterbare Applikation zu schaffen. Dazu wurde sich an der iOS-Projektstruktur orientiert und das Android-Projekt analog dazu umgesetzt, um eine einheitliche Architektur über beide Plattformen hinweg zu gewährleisten.

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

- **common** (Abschnitt 5.2): Enthält alle allgemeinen und wiederverwendbaren Komponenten, wie z. B. UI-Elemente, Utility-Klassen und grundlegende Architekturelemente.
- **features** (Abschnitt 5.3): 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 zu vermeiden.

Zusätzlich existiert das Modul `app`, das den Einstiegspunkt der Applikation (*Application-* und *MainActivity*-Ebene) enthält und tenant-spezifische Konfigurationen bündelt, wie etwa Startkonfigurationen, Feature-Aktivierungen und API-Endpunkte. (Abschnitt 5.1.3)

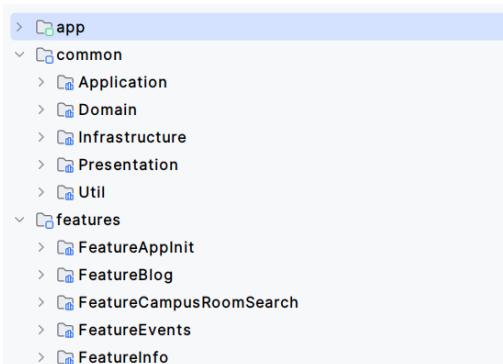


Abbildung 10.: Projektstruktur im Android Studio

5.1.1. Projektkonfiguration und Komponenten

Zu Beginn des Projekts wurde entschieden, das bestehende Entwicklungsprojekt `android-jetpackcompose-multitenant` weiterzuverwenden. Diese Entscheidung wurde bewusst getroffen, um vorhandene Strukturen und Konfigurationsansätze zu übernehmen. In der Praxis zeigte sich jedoch, dass ein Grossteil des bestehenden Codes aber im Verlauf des Projekts ohnehin neu implementiert wurde.

Da an dem Projekt über längere Zeit nicht mehr aktiv gearbeitet worden war, waren nahezu alle eingesetzten Bibliotheken und Konfigurationen veraltet. Entsprechend wurde zu Projektbeginn gezielt Zeit investiert, um den gesamten Build-Prozess und die Projektkonfiguration zu modernisieren. Dies umfasste die Aktualisierung auf Java 21, die Migration auf Kotlin 2.1.21 sowie das Upgrade auf das aktuelle Android-SDK-Level 36.

5.1.2. SDK-Konfiguration

Da das Projekt im Rahmen der Neuumsetzung vollständig geändert wurde, bot sich der passende Zeitpunkt, um auf die aktuelle Android-SDK-Version 36 zu wechseln. Diese ist seit der Veröffentlichung von **Android 16 (Juni 2025)** der empfohlene Standard („SDK: App architecture“, n. d.). Ziel war es, wieder auf dem aktuellen Stand der Android-Entwicklung zu sein, moderne APIs zu nutzen und eine möglichst langfristige Kompatibilität mit neuen Geräten sicherzustellen.

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

Im Projekt wurde bewusst entschieden, den Fokus auf aktuelle Geräte sowie Geräte mit einem maximalen Alter von etwa drei bis vier Jahren zu legen. Da sich die Nutzerbasis primär aus Studierenden zusammensetzt, ist davon auszugehen, dass mehrheitlich neuere Smartphones im Einsatz sind. Gleichzeitig wurde die **minSdk**-Version auf 30 gesetzt, um auch ältere Geräte weiterhin zu unterstützen und eine ausreichende Abdeckung sicherzustellen (siehe Anhang 8.3.3).

5.1.3. App-Modul

Das Verzeichnis `/app` stellt das Hauptmodul der Android-Applikation dar und ist der Einstiegspunkt der Anwendung. Es enthält die gesamte App-Konfiguration, die Initialisierung der Benutzeroberfläche sowie die tenant-spezifischen Projektstrukturen und Ressourcen für *HSLU I* und *HSLU TA*. Alle weiteren Module der App werden von diesem Modul aus konfiguriert und eingebunden, sodass sich in den übrigen Modulen (`common` und `features`) keine App- oder Tenant-spezifischen Konfigurationen mehr befinden und diese dadurch möglichst generisch und wiederverwendbar bleiben.

5.1.3.1. Multi-Tenant

Die Multi-Tenant-Fähigkeit der App wird über sogenannte *Product Flavors* (`hslui` und `hsluta`) realisiert. Tenant-spezifische Konfigurationen wie Tenant-ID, Name oder API-Tokens werden dabei direkt im `build.gradle` pro Flavor definiert. Zusätzlich existiert für jeden Flavor ein eigenes `AndroidManifest.xml`, in dem tenant-spezifische Einstellungen hinterlegt sind.

```

1 productFlavors {
2     create("hslui") {
3         dimension = "tenant"
4         buildConfigField("String", "TENANT_ID", "\"hslui\"")
5         buildConfigField("String", "TENANT_NAME", "\"HSLU I\"")
6         // ...
7     }
8     create("hsluta") {
9         dimension = "tenant"
10        buildConfigField("String", "TENANT_ID", "\"hsluta\"")
11        buildConfigField("String", "TENANT_NAME", "\"HSLU TA\"")
12        // ...
13    }
14 }
```

Listing 5.1: Auszug aus `build.gradle`

Build

Zur Build-Zeit kann beispielsweise in Android Studio ausgewählt werden, welche Variante der App gebaut werden soll. Während des Build-Prozesses werden das Hauptmanifest und das jeweilige Flavor-Manifest zusammengeführt. Dabei wird der Einstiegspunkt der App tenant-spezifisch gesetzt: Für hslui wird die HsluiApplication instanziiert, für hsluta entsprechend die HslutaTAAplication.

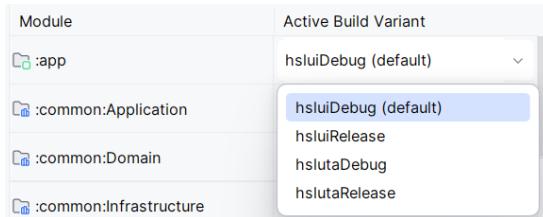


Abbildung 11.: Build-Variants in Android Studio

Runtime / Dependency Injection

Zu diesem Zeitpunkt ist die Applikation gestartet und es läuft bereits die tenant-spezifische Application-Klasse (HsluiApplication oder HslutaTAAplication). In diesen Klassen werden die Konfigurationen der App sowie der einzelnen Features vorgenommen. Mithilfe von Dependency Injection (in diesem Projekt umgesetzt mit *Dagger Hilt*) werden diese Konfigurationen instanziiert und der Applikation zur Verfügung gestellt, sodass sie innerhalb der App genutzt werden können. In diesem Schritt werden auch gemeinsam genutzte Services wie CommonNetworkService (Abschnitt 5.2.3.1) und CommonStorageService (Abschnitt 5.2.3.2) erzeugt. Diese werden als Singletons bereitgestellt, da sie zentral benötigt werden und während der gesamten Laufzeit der App nur einmal existieren sollen.

Zur Laufzeit verwendet die Anwendung dadurch automatisch die zum gewählten Tenant gehörenden URLs, Tokens und weiteren Konfigurationen. Auf diese Weise lassen sich mehrere App-Varianten aus derselben Codebasis erzeugen, ohne tenant-spezifische Logik im eigentlichen Anwendungscode zu vermischen. Nach dem Laden der App wird mit dem Bootstrapping (Laden der Features) über AppInit begonnen (Abschnitt 5.3.1).

Im Beispiel des News-Feature (Abschnitt 5.3.2) funktioniert die Dependency Injection so:

```

1 @Provides
2 fun provideFeatureNewsModuleLoaderConfig( provider: TenantUrlProvider):
3     FeatureNewsModuleLoaderConfig {
4         return FeatureNewsModuleLoaderConfig(
5             moduleLoaderUrl = "$base/API/MobileApps/Auth/News",
6             moduleLoaderDataLocalFileName = "news_module_data.json",
7             // ...
8         )
9 }
```

Listing 5.2: Provider-Methode

```

1 @HiltViewModel
2 class NewsDataLoader @Inject constructor(
3     val tenantConfig: AppTenantConfig, val featureConfig: FeatureNewsConfig,
4     val storageService: CommonStorageService, val networkService: CommonNetworkService
5 ) : ViewModel()
```

Listing 5.3: Injection im NewsDataLoader

Durch die Annotation `@HiltViewModel` wird der NewsDataLoader als von Hilt verwaltetes ViewModel registriert. Der mit `@Inject` annotierte Konstruktor ermöglicht es Hilt, alle benötigten Abhängigkeiten automatisch bereitzustellen und den NewsDataLoader ohne manuelle Initialisierung zu erzeugen.

5.1.4. Lokalisierung

Die App ist so konzipiert, dass sie in mehreren Sprachen verfügbar ist (aktuell Deutsch und Englisch). Eine grundlegende Lokalisierungslogik war bereits im ursprünglichen Android-Projekt vorhanden und wurde im Rahmen dieses Projekts weiterentwickelt.

Die Lokalisierung basiert weiterhin auf dem standardmässigen Android-Resource-System. (Android wählt automatisch die passende strings.xml basierend auf dem System-Locale) Dieses System wurde zusätzlich so ergänzt, dass auch dynamisch geladene Inhalte (beispielsweise Menüpunkte aus dem Bootstrapping-Prozess) zur Laufzeit korrekt lokalisiert dargestellt werden können.

Konzept

- Die Sprachauswahl erfolgt automatisch über die Systemeinstellung des Geräts; eine separate Auswahl der App-Sprache ist nicht vorgesehen.
- Es wird ein ressourcenbasiertes Lokalisierungskonzept verwendet: Deutsch ist als Standardsprache im Ordner values/ hinterlegt, Englisch im Ordner values-en/. Diese Ordner existieren jeweils pro Feature-Modul.
- Die Verwendung von UI-Strings erfolgt Android-typisch über stringResource(R.string....) in Compose beziehungsweise über context.getString(...).

```

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 5.4: Default (Deutsch) und Englisch in strings.xml

Übersetzung dynamischer Inhalte

Für dynamisch geladene Inhalte, wie beispielsweise Menüpunkte aus Remote-DTOs (siehe Abschnitt 5.3.1), wird die Sprache zur Laufzeit anhand der aktuellen Systemsprache bestimmt. Die Auswahl erfolgt über Locale.getDefault() und ordnet die passenden Sprachfelder zu:

```

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

Listing 5.5: Auswahl der passenden Sprachvariante

(Es wird zuerst versucht, die aktuell gesetzte Systemsprache zu verwenden. Ist für diese keine Übersetzung verfügbar, wird automatisch auf die jeweils andere unterstützte Sprache zurückgegriffen. Sollte auch diese nicht vorhanden sein, wird ein Fallback-Wert (z. B. 'Unnamed') verwendet.)

Damit sind alle Inhalte der App vollständig auf Deutsch und Englisch verfügbar. Weitere Sprachen könnten bei Bedarf einfach ergänzt werden, indem zusätzliche strings.xml-Dateien angelegt werden.

5.1.5. 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
build_debug	Erstellt einen Debug-Build ohne Signierung. Führt gradle clean assemble für das entsprechende Tenant-Modul aus und kopiert die Artefakte in den Output-Ordner. Wird hauptsächlich für lokale Tests verwendet.
build_release	Erstellt einen signierten Release-Build. Lädt den verschlüsselten Keystore aus dem Zertifikats-Repository, entschlüsselt ihn, erstellt die keystore.properties-Datei für Gradle und führt gradle clean bundle aus. Das signierte AAB wird in den Output-Ordner kopiert.
beta	Baut ein signiertes Release-Bundle und lädt es in den Google Play Store hoch. Standardmäßig wird der Beta-Track verwendet, kann aber über den track:-Parameter überschrieben werden (z. B. track:internal). Lädt sowohl Keystore als auch Google Play API-Key aus dem Zertifikats-Repository. Ruft intern build_release auf und lädt anschliessend das AAB hoch.
release	Analog zu beta, lädt jedoch explizit in den Production-Track des Google Play Stores. Wird für finale Releases verwendet.

Tabelle 5.1.: 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 Modul CommonApplication stellt gemeinsame Basisklassen für das Laden und Synchronisieren der Daten von Features (z.B. News, Blog,...) bereit. Das Herzstück ist dabei die Klasse CommonApplicationBaseModuleLoader, die Feature-Daten aus dem lokalen Speicher und der Backend-API liest/speichert. (Zusätzliches Diagramm dazu im Anhang 8.2.3).

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 CommonApplicationBaseModuleLoader, die den Datenlade- und Synchronisationsprozess bereitstellt, müssen konkrete Feature-Module (z.B. News, Blog,...) nur noch ihre spezifischen Typparameter und Konfigurationen bereitstellen. Dies reduziert den Implementierungsaufwand erheblich und stellt sicher, dass alle Module den gleichen Ladevorgang einhalten.

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>
```

Listing 5.6: Klassendefinition CommonApplicationBaseModuleLoader

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 }
```

Listing 5.7: Timestamp-Vergleich in CommonApplicationBaseModuleLoader

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)
```

Listing 5.8: Deserialisierung mit Gson

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 ViewModels oder Composables 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 ViewModel 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 5.2.: Technische Aspekte des `CommonApplicationBaseModuleLoader`

5.2.2. CommonDomain

Das Modul CommonDomain 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 CommonAppModuleItemAPIDTO und CommonAppModuleAPIDTO, 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. `CommonAppModuleItemAPIDTO` stellt die Basis für einzelne Modul-Items dar:

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

Listing 5.9: Klasse CommonModuleModuleItemAPIDTO

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

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

Listing 5.10: Klasse CommonModuleModuleAPIDTO

Feature-spezifische DTOs erben von diesen Basisklassen. Beispielsweise erbt `AppMensaModuleItemAPIDTO` von `CommonAppModuleItemAPIDTO` 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 }
```

Listing 5.11: Klasse CommonModuleLoaderStatus

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: androidx.core:core-ktx für Android Core Utilities, com.google.code.gson:gson für JSON-Serialisierung und kotlinx-coroutines-core 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 @SerializedName-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 AppInitModuleItemAPIDTO enthalten computed properties, die basierend auf der aktuellen Locale die richtige Sprache auswählen. Die Implementierung verwendet Locale.getDefault().language 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 5.3.: Technische Aspekte des CommonDomain Moduls

5.2.3. CommonInfrastructure

Das Modul CommonInfrastructure bietet die technische Basis für Netzwerk- und Datenzugriffe, die von anderen Modulen genutzt werden.

5.2.3.1. Network

Die Klasse CommonNetworkService stellt die zentrale Netzwerkschicht der Anwendung dar und bietet eine einheitliche Schnittstelle für alle Netzwerkoperationen. Es abstrahiert die Komplexität der Netzwerkkommunikation 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.

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 }
```

Listing 5.12: Klasse CommonNetworkService

Der NetworkMonitor überwacht die Netzwerkverbindung und stellt Informationen über die aktuelle Verbindungsqualität 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	yyyyyyy-yyyy-yyyy-yyyy-yyyyyyyyyy	hsluta

Tabelle 5.4.: 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.
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 5.5.: Technische Aspekte des CommonNetwork Moduls

5.2.3.2. Storage

Die Klasse `CommonStorageService` 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: ApInitModule-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 Domänenschicht 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 }
```

Listing 5.13: AppDatabase

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 )
```

Listing 5.14: Beispiel JSON-Speicher

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 Tabellen 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 )
```

Listing 5.15: Beispiel File-Speicher

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 5.6.: Technische Aspekte des CommonStorage Moduls

5.2.4. CommonPresentation

Das Modul CommonPresentation 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.

Umsetzung / Funktionsweise

Die Komponenten sind als Composable-Funktionen implementiert und folgen den Jetpack Compose Best Practices.

Die wichtigsten Elemente aus dem CommonPresentation-Modul sind:

- **GenericErrorView**: Eine einfache und zentrale Fehleranzeige, die eine Fehlermeldung übersichtlich darstellt. Optional kann ein „Retry“-Button angezeigt werden, mit dem der fehlgeschlagene Vorgang erneut ausgelöst werden kann. Diese Komponente wird in mehreren Features verwendet, um Fehler einheitlich darzustellen.
- **WebViewScreen**: Eine Composable-Komponente zur Anzeige externer Inhalte innerhalb der App. Sie stellt eine Android WebView bereit und unterstützt unter anderem JavaScript, Cookies und DOM Storage. Je nach Konfiguration können Inhalte entweder direkt in der App oder extern im Systembrowser geöffnet werden.
- **LogoSplashView**: Eine animierte Splash-Screen-Komponente, die beim Start der App ein Logo sowie einen Untertitel anzeigt. Nach Abschluss der Animation kann ein Callback ausgelöst werden, um automatisch zur nächsten Ansicht zu wechseln. Die Komponente wird während des App-Bootstrappings für den initialen Startbildschirm verwendet.
- **CommonBootstrappingProgressView**: Eine wiederverwendbare Composable-Komponente, die den Initialisierungs- und Ladeprozess von Modulen kapselt. Sie übernimmt automatisch die Ausführung von `setup()` und `process()` und stellt je nach Status eine Ladeansicht, eine Erfolgsansicht oder eine Fehleransicht dar. Dadurch kann der Bootstrapping-Prozess in allen Features einheitlich und ohne doppelten Code umgesetzt werden.

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

Listing 5.16: Klasse CommonBootstrappingProgressView

5.2.5. CommonUtil

Das Modul `CommonUtil` stellt zentrale Hilfsfunktionen bereit, die von allen Feature-Modulen genutzt werden. Dazu gehören unter anderem Datumsformatierungen, das Parsen von HTML-Inhalten sowie das Laden und Verarbeiten von JSON-Daten. Durch diese zentrale Sammlung werden häufig verwendete Operationen gebündelt und Code-Duplikationen vermieden.

Ziel des Moduls ist es, eine einheitliche und konsistente Nutzung solcher Hilfsfunktionen über die gesamte Applikation hinweg sicherzustellen. Komplexere Aufgaben, wie das Extrahieren von Inhalten aus HTML oder das Verarbeiten strukturierter Daten, werden an einer zentralen Stelle implementiert. Dadurch können Anpassungen oder Fehlerkorrekturen effizient vorgenommen werden und wirken sich automatisch auf alle betroffenen Features aus.

Umsetzung / Funktionsweise

Das Modul besteht aktuell 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 }
```

Listing 5.17: DateFormatter

Die `HtmlContentParser` bietet Funktionen zum Parsen von HTML-Content und Extraktion von Content-Blöcken. Dies wird unter anderem bei den Features News und Blog verwendet, um Daten aus der WordPress-Api in saubere Composable-Elemete zu konvertieren. 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 }
```

Listing 5.18: Beispiel HTML-Tags dekodieren

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.

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.

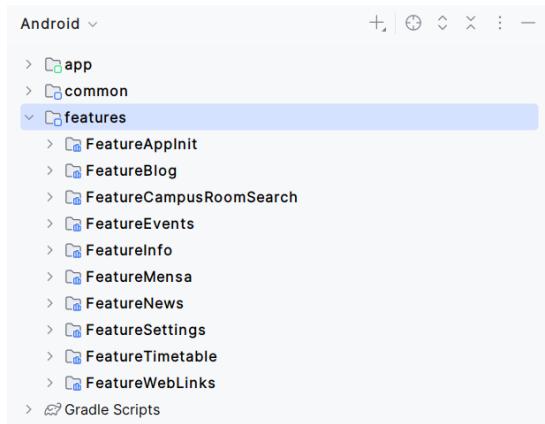


Abbildung 12.: Features in Android-Studio

Ziel und Rolle der Features

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

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

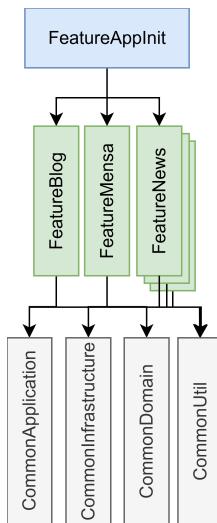


Abbildung 13.: Abhängigkeiten in Pfeilrichtung erlaubt/vorhanden

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.

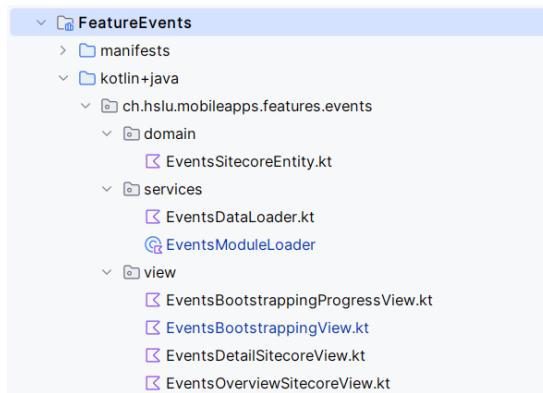


Abbildung 14.: Struktur der Features am Beispiel Event

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.

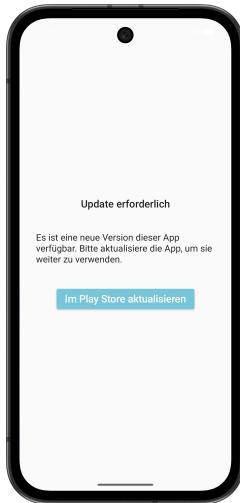


Abbildung 15.: Aufforderung zum Versionsupdate

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. Ein Sequenziagramm des gesamten Ablaufes ist im Anhang zu finden (siehe Anhang 8.2.2).

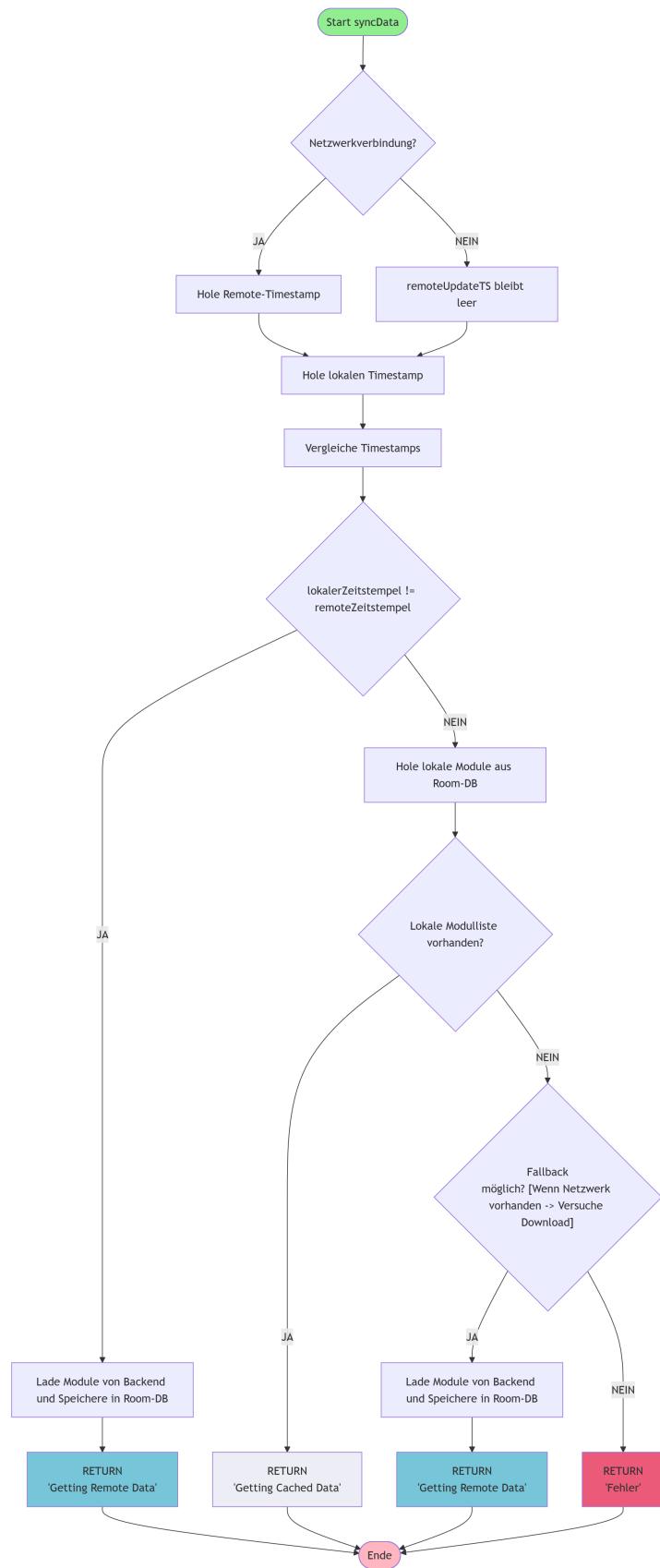


Abbildung 16.: Flussdiagramm Menü-Synchronisation

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 anzulegende 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 *Acccompanist-WebView* („Acccompanist: WebView wrapper for Jetpack Compose“, n. d.). Da diese Library jedoch als *deprecated* markiert wurde, ersetzten 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.

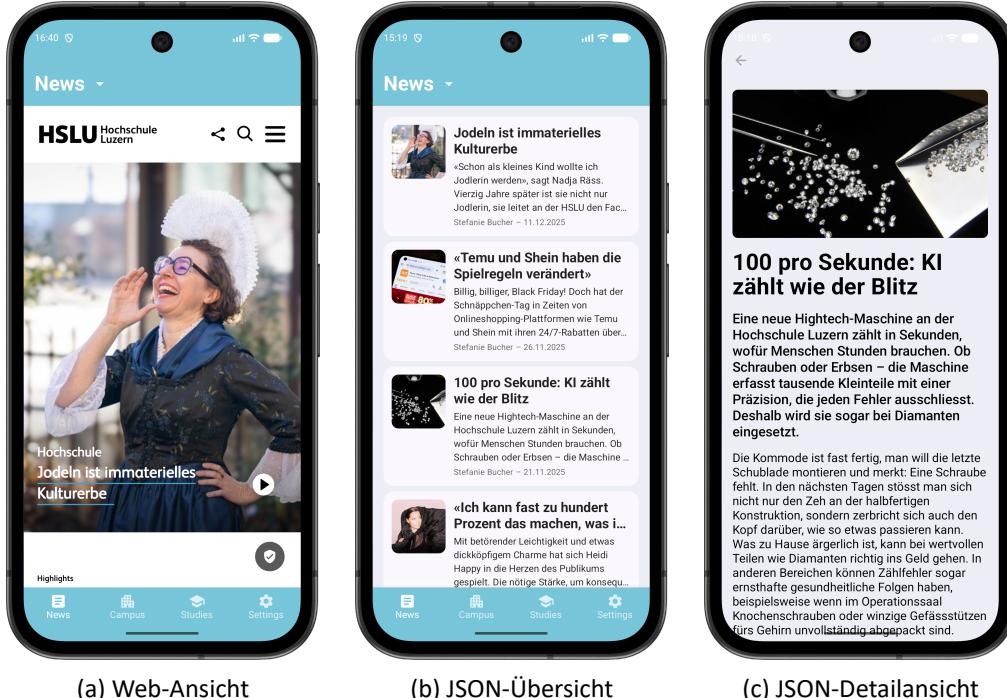


Abbildung 17.: Screenshots des News-Features

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 18.: Screenshots des Blog-Features



Abbildung 19.: 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:

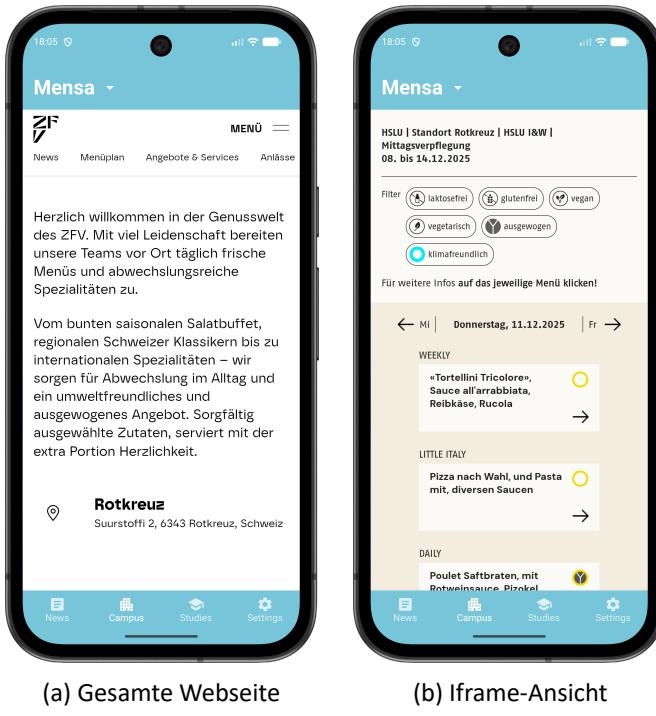


Abbildung 20.: Screenshots des Mensa-Features

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. („Sitecore: Digital Experience Solutions“, n. d.)

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.

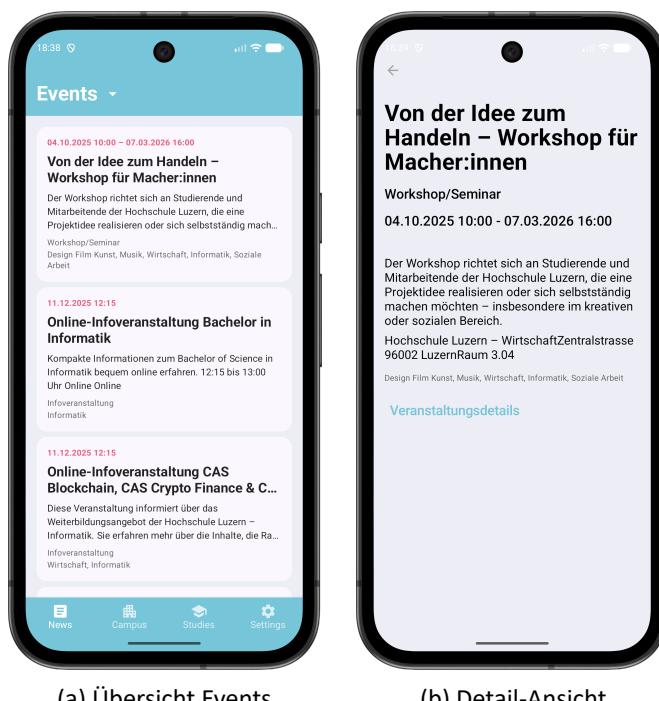


Abbildung 21.: Screenshots des Events-Features

5.3.6. Info-, Einstellungen- und WebLinks-Features

Die Features *Info*, *Einstellungen* und *WebLinks* übernehmen in der App kleinere Funktionen. Da sich diese Features in Aufbau und Darstellung stark ähneln, wurde ein grosser Teil des Layouts zentral im common-Modul umgesetzt und wiederverwendet, um Code-Duplikationen zu vermeiden und eine einheitliche Darstellung sicherzustellen.

WebLinks

Das Feature *WebLinks* stellt eine Liste von weiterführenden Links zur Verfügung, die auf externe Seiten der HSLU verweisen. Inhalte, die nicht direkt in der App umgesetzt sind, können so trotzdem einfach erreicht werden. Die Liste der Links wird über die Backend-API geladen, lokal zwischengespeichert und in einer Liste dargestellt. Beim Antippen eines Eintrags wird der jeweilige Link im externen Standard-Browser des Geräts geöffnet. Das Feature unterstützt Deutsch und Englisch, wobei die Sprache automatisch anhand der Geräteeinstellung gewählt wird. Falls eine Übersetzung nicht verfügbar ist, wird jeweils auf die andere Sprache zurückgegriffen (siehe 5.1.4).

Info

Das Feature *Info* (bzw. *About*) zeigt allgemeine Informationen zur App an. Dazu gehören unter anderem der Name des Tenants, die aktuell installierte App-Version, Kontaktinformationen, ein Blog-Link sowie eine Liste der bisherigen Mitwirkenden. Zusätzlich werden rechtliche Hinweise wie Disclaimer oder Copyright angezeigt. Diese Informationen werden ebenfalls über die Backend-API geladen. Inhalte werden nur angezeigt, wenn sie tatsächlich vorhanden sind, leere Felder bleiben ausgeblendet.

Einstellungen

Das Feature *Einstellungen* dient der technischen Verwaltung der App-Daten. Hier können gecachte Inhalte einzelner Module (z. B. News, Blog, Mensa oder WebLinks) manuell neu geladen oder zurückgesetzt werden. Dies ist bei Problemen mit veralteten Daten hilfreich. Zusätzlich enthält das Settings-Feature die Konfiguration für den Stundenplan, bei der der verwendete Kalender ausgewählt werden kann.

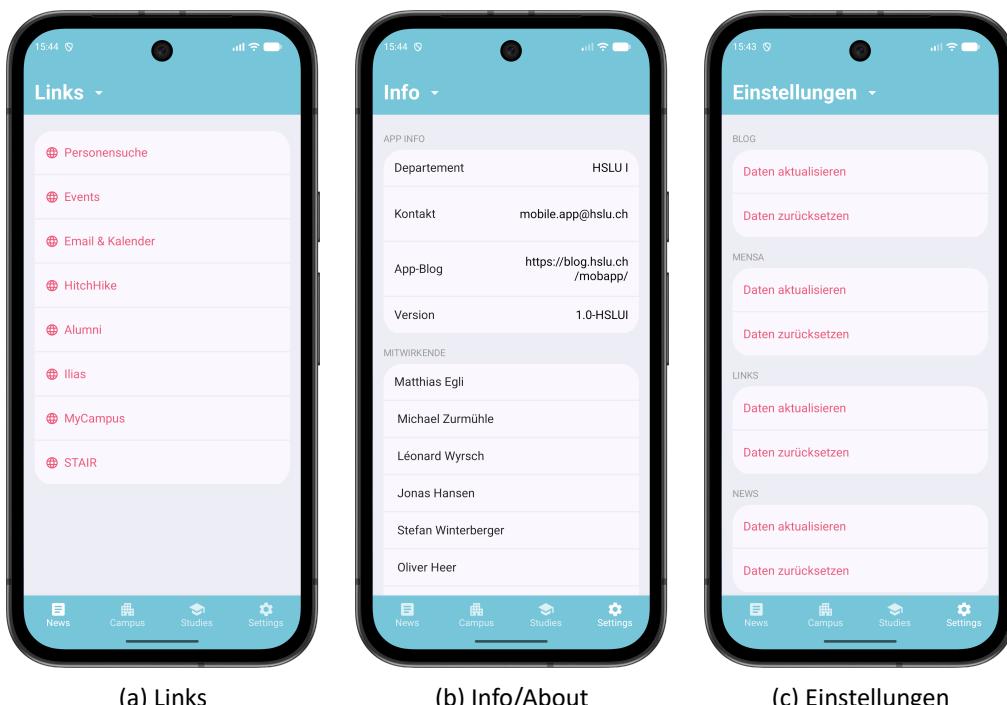


Abbildung 22.: Screenshots der Features

5.3.7. 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.

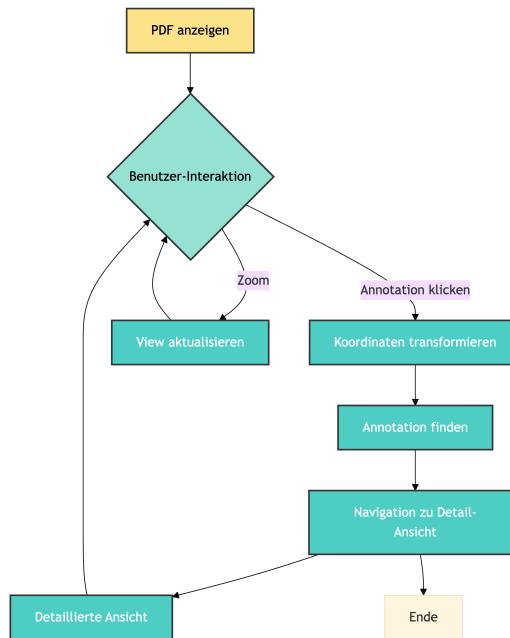


Abbildung 23.: Ablaufdiagramm

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 Annotationsen 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 5.7.: Wichtigste Funktionalitäten des CampusRoomSearch Features

5.3.8. 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 („Android *CalendarContract* | API reference“, n. d.). 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.

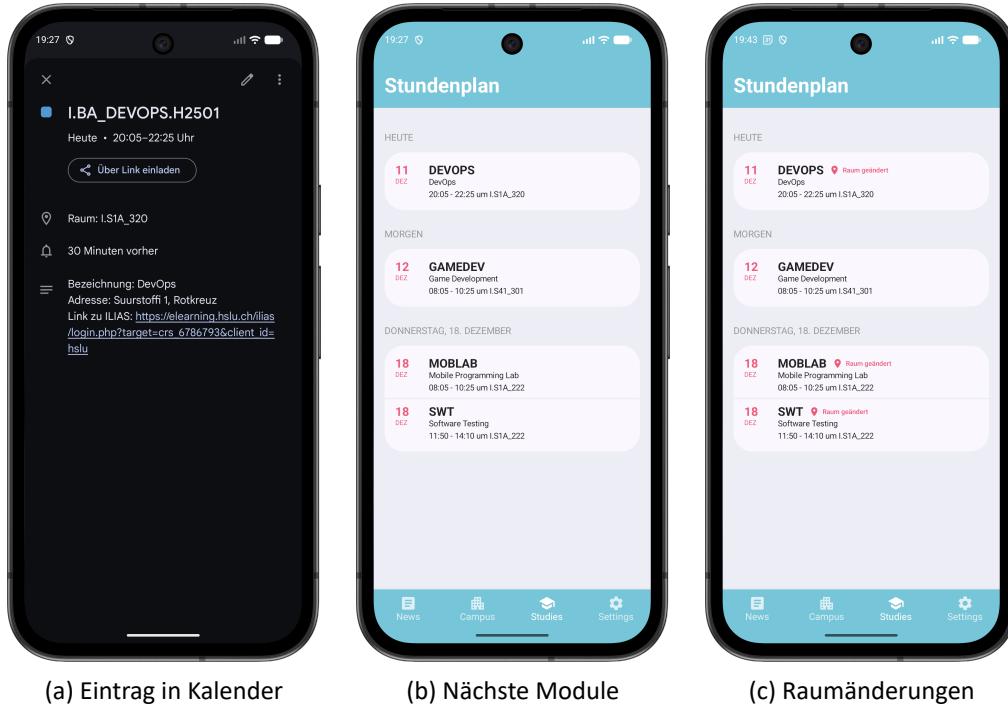


Abbildung 24.: Screenshots des Kalender-Features

5.3.8.1. 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 *Glance* umgesetzt und nutzt damit die Compose-basierte Widget-Architektur von Android („Android | Jetpack Glance“, n. d.).

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>

```

Listing 5.19: Auszug aus `AndroidManifest.xml`

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.



(a) Zwei Vorlesungen

(b) Keine Vorlesungen

Abbildung 25.: Screenshots des Kalender-WIDGETS

6. Validierung und Evaluation

6.1. Validierung

6.1.1. Technische Validierung

6.1.1.1. Bisherige Testabdeckung

Die bisherigen App-Projekte der Hochschule (*android-xml-multitenant* und *ios-swiftui-multitenant*) verfügten bislang über kaum automatisierte Testabdeckung. Da wir diesen Aspekt jedoch als wichtig erachteten, haben wir uns zum Ziel gesetzt, zentrale Funktionen mit ausführbaren Tests abzusichern, um die Funktionalität und Stabilität der App besser gewährleisten zu können.

6.1.1.2. Technische Validierung der Android-Applikation

Die technische Validierung der Android-Applikation erfolgte sowohl über automatisierte Tests als auch durch kontinuierliches manuelles Testen während der Entwicklung. Ziel war es, Funktionalitäten schrittweise abzusichern und Fehler möglichst frühzeitig zu erkennen.

Unit Tests wurden im gesamten Projekt eingesetzt. Dabei handelte es sich meist um kleine, schnell ausführbare Tests, die während der Entwicklung laufend genutzt wurden. Gerade nach Änderungen am Code waren diese Tests hilfreich, da sie rasch aufzeigen konnten, ob bestehende Funktionalitäten weiterhin korrekt arbeiteten. So konnten Fehler früh erkannt werden, ohne die App jedes Mal manuell testen zu müssen.

Eine vollständige Testabdeckung aller Module und Features konnte im gegebenen Projektrahmen nicht umgesetzt werden, da die verfügbare Projektzeit begrenzt war und ein grosser Teil der Aufwände in die Entwicklung der Applikation investiert wurde (siehe Anhang 8.2.18). Die vorhandenen Tests wurden jeweils parallel zur Feature-Entwicklung erstellt. Das Ergänzen weiterer Tests ist problemlos möglich, da die bestehende Teststruktur modular aufgebaut ist und einfache Erweiterungen zulässt.

Nach der Abgabe der Dokumentation wird bis zur Abschlusspräsentation weiter am Projekt gearbeitet, mit einem stärkeren Fokus auf das Testing. Mit dieser zusätzlichen Zeit wird angestrebt, die Testabdeckung auf rund 80 % zu erhöhen.

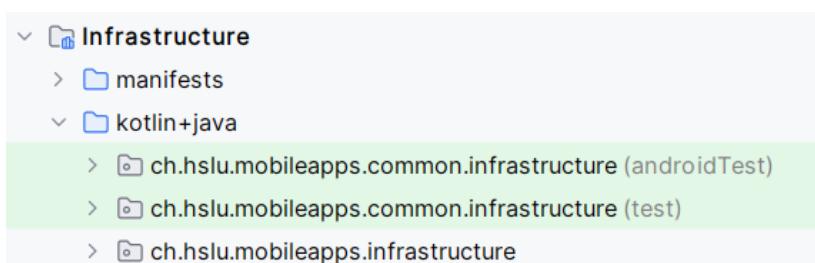


Abbildung 26.: Teststruktur im Android Studio

6.1.1.3. Manuelles Testen

Neben den automatisierten Tests wurde die Applikation während der Entwicklung regelmässig manuell getestet. Der Grossteil dieser Tests erfolgte mit dem Android-Emulator, da sich damit unterschiedliche Android-Versionen, Displaygrössen und Konfigurationen effizient abdecken lassen.

Zusätzlich wurde die App auf ausgewählten physischen Geräten getestet, konkret auf einem *Google Pixel* sowie einem *Samsung Galaxy S22*. Da Android jedoch auf einer sehr grossen Vielfalt an Geräten und Herstellerkonfigurationen läuft und nur eine begrenzte Anzahl an Testgeräten zur Verfügung stand, konnte keine breite Abdeckung aller Gerätetypen erreicht werden.

Dieses Vorgehen und die daraus resultierende Einschränkung bei den Tests auf realen Geräten wurden im Projektverlauf mit dem Auftraggeber besprochen und als ausreichend für den gegebenen Projektrahmen beurteilt. (siehe Anhang 8.3.8).

6.1.1.4. Eingesetzte Testframeworks

Im Projekt kamen folgende Testframeworks zum Einsatz:

- **JUnit 5:** Basis-Framework für Unit- und einfache Integrationstests. Es wird im gesamten Projekt eingesetzt, um einzelne Funktionen, Klassen und ViewModels zu validieren und deren Logik zu überprüfen.
- **MockK:** Mocking-Framework für Kotlin. Es wird verwendet, um Abhängigkeiten wie Services oder Repositories zu mocken, beispielsweise beim Testen von Netzwerkzugriffen oder Datenlade-Logik ohne echte API-Aufrufe.
- **Espresso:** Framework für UI- und Integrationstests im Android-Kontext. Es kommt bei einfachen Benutzerinteraktionen, Navigationsflüssen und grundlegenden UI-Validierungen zum Einsatz.

6.1.2. Validierung der Anforderungen und Projektziele

Die Validierung der fachlichen Anforderungen und Projektziele erfolgte iterativ über den gesamten Projektverlauf hinweg. Die Entwicklung wurde sprint-basiert organisiert, wobei alle Aufgaben und Features als Issues in GitLab erfasst und verwaltet wurden.

Alle zwei Wochen fanden Meetings mit dem Auftraggeber statt, in denen der aktuelle Projektstand vorgestellt, schriftlich dokumentiert und gemeinsam besprochen wurde. (siehe Anhang 8.3). Dabei wurden umgesetzte Features überprüft, offene Punkte geklärt und die Prioritäten für den nächsten Sprint festgelegt. Auf diese Weise konnte laufend validiert werden, ob das Projekt in die gewünschte Richtung entwickelt wird.

Durch das Issue-Tracking in GitLab war jederzeit ersichtlich, welche Aufgaben bereits abgeschlossen waren und welche noch offen standen. Am Ende des Projekts sind alle geplanten Issues geschlossen, was bedeutet, dass die vereinbarten Features umgesetzt wurden.

6.1.3. Validierung des Nutzens

Im Rahmen dieses Projekts wurde keine Validierung mit Studierenden durchgeführt, um zu überprüfen, ob die App im Alltag tatsächlich als nützlich wahrgenommen wird oder einen konkreten Mehrwert bietet. Eine solche Nutzerstudie hätte den inhaltlichen Rahmen des Projekts deutlich überschritten und war daher nicht Bestandteil des Auftrags.

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 8.2.3).

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 8.2.3).

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 8.2.3).

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 8.2.3).

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 8.2.3).

6.2.3. Vergleich

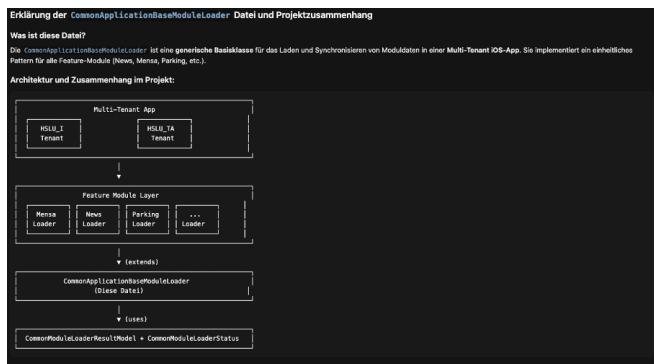
KI	Stärken / beobachtete Fähigkeiten	Schwächen / Unsicherheiten	Eignung
Grok (xAI)	xAI hat mit „grok-code-fast-1“ ein Modell für Entwickler-Aufgaben (Agentic Coding) vorgestellt; Effizienz/Qualität im Fokus; funktioniert auch als Backend in Tools wie Cursor.	Relativ jung; unklare Reife/Plattformtiefe; 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 (außer Enterprise/Pro); potenzielle Halluzinationen → Review nötig.	Top-Option neben Claude; stark für Architektur/Best Practices.
Cursor	IDE-Frontend, das verschiedene Modelle einbindet; verbessert Workflow und Kontext im Editor.	Keine eigene KI; Leistung hängt vom gewählten Modell ab.	Als Interface/Workflow-Booster zusammen mit starkem Modell nutzen (z. B. Grok/ChatGPT/Claude).

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.



(a) Analysebeispiel 1

Kernfunktionalitäten:

- Generische Typen:
CommonApplicationBaseModuleLoader<T, T>
// I: Item-ID (z.B. AppFeatureModuleId, MensaFeatureModuleId)
// T: Feature-Modul (z.B. MensaFeatureModule, NewsFeatureModule)
// C: Config (z.B. FeatureMensaModuleConfig)
- Drei Lade-Modi:
 - Hybrid: Intelligente Sync zwischen Remote|Local (Standard)
 - Local: Nur lokale Daten
 - Remote: Nur Remote-Daten (noch nicht implementiert)
- Intelligente Synchronisation (syncData()):
// Vergleicht Timestamps zwischen Remote und Local
// Lädt nur bei Bedarf neue Daten herunter
// Fallback auf lokale Daten bei Netzwerkproblemen
- SwiftUI Integration:
@Published public var loadingStatus : CommonModuleLoaderStatus<T>
// Reactive UI-Updates für Loading-States

Verwendung in konkreten Modulen:

```

// Beispield: MensaModuleLoader.swift
public class MensaModuleLoader : CommonApplicationBaseModuleLoader<MensaFeatureModuleId, MensaFeatureModuleConfig> {
    // ...
    @Published public var loadingStatus : CommonModuleLoaderStatus<MensaFeatureModuleId>
    // ...
}
// Erst alle Funktionalität, keine zusätzliche Implementierung nötig

```

(b) Analysebeispiel 2

Dependency Injection Pattern:

```

// In HSLUApp.swift / HSLUApp.swift
extension Application {
    func setupCommonModuleLoader() {
        appConfig: mensaConfig, // Tenant-spezifische URLs
        storageServices: storageService, // File-Management
        networkServices: networkService // HTTP-Client
    }
}

Zusammenhang mit CommonModuleLoader/ResultModel:
Die zuvor erklärte CommonModuleLoader/ResultModel 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:

- DRY-Prinzip: Einmal implementiert, überall verwendbar
- Konsistenz: Alle Module verhalten sich identisch
- Offline-First: Intelligente Caching-Strategie
- Multi-Tenant: Verschiedene Tenants, gleiche Logik
- Kurstabilität: Dependency Injection ermöglicht Mocking

(c) Analysebeispiel 3

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: // Zeigt Loading-Spinner
    case .loaded: // Zeigt Liste mit Daten
    case .error(let message): // Zeigt Fehler
}

```

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

(d) Analysebeispiel 4

Abbildung 27.: Analysebeispiele 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.

6.2.6. Erfahrungen mit AI-First Ansatz

Während der gesamten Projektlaufzeit haben wir Cursor aktiv im Entwicklungsprozess eingesetzt und dabei sowohl positive als auch negative Erfahrungen gesammelt. Diese Erkenntnisse sind wertvoll für die zukünftige Nutzung von KI-Tools in Softwareentwicklungsprojekten.

Stärken von Cursor

Einer der grössten Vorteile von Cursor war die Fehlersuche und Debugging-Unterstützung. Durch den vollständigen Projektkontext konnte Cursor Fehlerquellen schnell identifizieren und konkrete Lösungsvorschläge liefern. Besonders hilfreich war die Fähigkeit, Zusammenhänge zwischen verschiedenen Dateien und Modulen zu erkennen, was bei komplexen Architekturen wie unserer Multi-Feature- und Multi-Tenant-Struktur von grossem Wert war.

Ein weiteres herausragendes Anwendungsgebiet war das Portieren von iOS-Code nach Android. Da Cursor Zugriff auf beide Codebasen hatte, konnte es Swift-Code analysieren und entsprechende Kotlin-Äquivalente vorschlagen. Dies beschleunigte die Migration erheblich, insbesondere bei der Umsetzung der Common-Module, die eine 1:1-Übersetzung der iOS-Architektur erforderten. Cursor konnte dabei nicht nur syntaktische Unterschiede berücksichtigen, sondern auch plattformspezifische Besonderheiten wie die unterschiedliche Behandlung von Coroutines versus `async/await` in Swift erkennen.

Auch bei der Code-Analyse und Dokumentation war Cursor sehr nützlich. Komplexe Code-Stellen konnten schnell erklärt werden, und die Generierung von Kommentaren oder Dokumentation basierend auf dem vorhandenen Code funktionierte zuverlässig.

Grenzen und Herausforderungen

Trotz der vielen Vorteile stiessen wir auch auf Bereiche, in denen Cursor weniger effektiv war. Besonders auffällig war dies bei UI-bezogenen Aufgaben. Die Generierung von Jetpack Compose UI-Komponenten war oft unzureichend, da die KI Schwierigkeiten hatte, das visuelle Design und die gewünschte Benutzererfahrung korrekt zu interpretieren. Die generierten UI-Komponenten entsprachen häufig nicht den Design-Anforderungen und mussten manuell überarbeitet werden, was den Zeitvorteil zunichtemachte.

Ähnliche Herausforderungen traten bei der Generierung kompletter Code-Strukturen auf. Während Cursor bei der Implementierung einzelner Funktionen oder Methoden sehr hilfreich war, scheiterte es oft bei der Erstellung gröserer Architektur-Komponenten. Die generierten Strukturen entsprachen nicht immer den etablierten Patterns und Konventionen unseres Projekts, und die notwendigen Anpassungen waren teilweise zeitaufwändiger als eine manuelle Implementierung.

Ein weiterer limitierender Faktor war die Abhängigkeit von der Qualität der Prompts. Um gute Ergebnisse zu erzielen, mussten sehr detaillierte und spezifische Anfragen formuliert werden, was teilweise mehr Zeit in Anspruch nahm als die direkte Code-Implementierung.

Fazit der praktischen Anwendung

Insgesamt war Cursor ein wertvolles Werkzeug im Entwicklungsprozess, besonders für repetitive Aufgaben, Fehlersuche und Code-Migration. Für UI-Entwicklung und die Generierung komplexer Architekturen war der manuelle Ansatz jedoch oft effizienter. Der AI-First Ansatz hat sich somit als ergänzende, aber nicht ersetzende Methode erwiesen, die am besten in Kombination mit traditionellen Entwicklungsmethoden eingesetzt wird.

7. Ausblick

7.1. Reflexion

7.1.1. Fazit AI-First Programmierung

Der *AI-First*-Ansatz hat uns während der gesamten Projektdauer begleitet. Wir hatten bereits vor Projektbeginn Erfahrung im Umgang mit KI-Tools, jedoch wurde künstliche Intelligenz in diesem Projekt so intensiv eingesetzt wie zuvor noch nicht. Über die gesamte Laufzeit konnten dabei sowohl positive als auch kritische Erfahrungen gesammelt werden.

Insgesamt überwiegen für uns klar die positiven Aspekte. Die Zeitersparnis durch den Einsatz von AI war sehr gross und hat sich im Projektalltag deutlich bemerkbar gemacht. Da wir beide bereits mehrere Jahre Erfahrung in der Informatik haben und auch noch ohne AI-gestützte Entwicklung gearbeitet haben, war der Vergleich für uns gut möglich. Das Projekt hat bestätigt, dass AI den Entwicklungsprozess beschleunigt und viele alltägliche Aufgaben vereinfacht. Unsere Erwartungen an den Nutzen von AI wurden dabei grösstenteils erfüllt.

Wichtig ist aus unserer Sicht jedoch, dass der Output einer AI stark vom Input abhängt. Gute und präzise Prompts führen in der Regel auch zu brauchbaren Resultaten, während vage oder ungenaue Eingaben oft zu weniger hilfreichen Antworten führen. Aus diesem Grund möchten wir unsere Erfahrungen festhalten und weitergeben, da wir den bewussten Einsatz von AI als zentral erachten und diese Erfahrungen weitergeben möchten. Auch wenn sich die Möglichkeiten von AI sehr schnell weiterentwickeln, lassen sich heute noch klare Stärken und Grenzen erkennen.

Stärken

Aus unserer Sicht eignet sich AI besonders gut für folgende Aufgaben:

- Analyse von bestehendem Code: AI kann sehr gut erklären, wie Code aufgebaut ist und wo sich welche Logik befindet, wodurch sich schnell ein Überblick gewinnen lässt.
- Übersetzung von bestehendem Code in andere Sprachen (z. B. Swift nach Kotlin), da der vorhandene Code als sehr präziser Prompt dient.
- Schreiben einzelner Funktionen oder Klassen, sofern diese nicht zu umfangreich sind.
- Umsetzung ähnlicher Features auf Basis bestehender Funktionalität (z. B. News → Blog), da vorhandener Code als gute Vorlage dient.
- Allgemeine Arbeitserleichterung im Vergleich zum klassischen Vorgehen (Problemsuche, Recherche im Internet, StackOverflow, Dokumentation).

Grenzen und Herausforderungen

Trotz der vielen Vorteile gibt es aus unserer Sicht auch klare Einschränkungen:

- Die Generierung von gut lesbarer und inhaltlich ausgewogener Dokumentation ist noch schwierig. Texte sind zwar nicht grundsätzlich schlecht, enthalten aber oft auffällige Füllwörter (beispielsweise „insbesondere“, „vollständig“, „speziell“), die dem Leser auffallen. Zudem werden teilweise wichtige Details ausgelassen, die eigentlich relevant für das Verständnis wären.
- Das Generieren von Diagrammen funktioniert aus unserer Sicht noch nicht besonders gut. Zwar konnte Cursor vor allem Ablauf- und Sequenzdiagramme relativ häufig erzeugen, und diese ergaben teilweise sogar noch Sinn, da der reine Programmablauf abgebildet wird.
- Bei eher abstrakten oder sogenannten „top-down“ Diagrammen, wie zum Beispiel Anwendungsfalldiagrammen, stösst die AI klar an ihre Grenzen. Hier müsste das Gesamtsystem wirklich verstanden werden, anstatt

nur den Codefluss wiederzugeben. Unser Eindruck ist, dass die AI dafür aktuell noch nicht ausreichend weit ist.

- Beim Erstellen von UI-Layouts liefert AI zwar grobe Strukturen, das Ergebnis entspricht jedoch selten den Erwartungen. Unserer Erfahrung nach kann die AI nur schwer einschätzen, wie ein Layout am Ende tatsächlich auf dem Bildschirm aussieht (siehe 8.2.1).
- Clean-Code-Prinzipien werden nicht immer konsequent eingehalten. Teilweise entstehen doppelte Implementierungen oder ungünstige Strukturen, da der Gesamtzusammenhang nicht vollständig erfasst wird.
- Syntaxfehler werden nicht immer erkannt und müssen explizit korrigiert werden (siehe 8.2.1).
- Aufgrund veralteter Trainingsdaten werden teilweise ältere oder nicht mehr empfohlene Bibliotheken vorgeschlagen, die manuell aktualisiert werden müssen.

Gesamtfazit

Aus unserer Sicht kann AI derzeit keine vollständige Applikation ohne fundierte Programmierkenntnisse ersetzen. Diese sind weiterhin notwendig, um Fehler zu beheben, Architekturentscheidungen zu treffen und eine saubere Struktur zu gewährleisten. Dennoch ist AI ein Werkzeug, auf das moderne Softwareentwicklung kaum mehr verzichten kann. Die Arbeitserleichterung ist gross, birgt aber auch die Gefahr, sich zu stark darauf zu verlassen. Ein bewusster Einsatz ist unserer Meinung nach daher entscheidend.

7.1.2. Team-Fazit

Zusammenarbeit im Team

Die Zusammenarbeit im Team war sehr gut. Beide haben durchgehend fleissig am Projekt gearbeitet. Die Kommunikation funktionierte ebenfalls sehr gut, da wir bereits mehrere Hochschulprojekte gemeinsam umgesetzt haben und die Arbeitsweise und Stärken des jeweils anderen gut kennen. Dadurch konnten Aufgaben effizient aufgeteilt werden und es war jederzeit klar, wer sich um welchen Bereich kümmert.

Projekt und Umsetzung

Im Projekt wurde insgesamt sehr viel umgesetzt und entsprechend auch viel Code geschrieben. Zu Beginn war das Projekt technisch relativ komplex, insbesondere durch Themen wie Multi-Tenant, Feature-Struktur und Backend-Anbindung. Nach einer kurzen Einarbeitungsphase konnte jedoch gut damit gearbeitet werden. Rückblickend hätten wir zu Projektbeginn eventuell noch ein paar zusätzliche Fragen an den Auftraggeber stellen können, um gewisse Punkte früher zu klären.

Der Projektstand war über die gesamte Laufzeit gut, es wurde kontinuierlich am Projekt gearbeitet und (fast) alle benötigten Features konnten umgesetzt werden. Die Dokumentation mit LaTeX empfanden wir als sehr praktisch, da sie (ähnlich wie Programmcode) sauber strukturiert, versioniert und direkt im Repository abgelegt werden konnte.

Positiv hervorzuheben ist auch der Umgang mit Änderungen im Projektumfang. Anpassungen wie JSON-basierte Views, API-Versionierung oder die Umsetzung der Raumsuche als PDF konnten während des Projektes aufgenommen und integriert werden. Auch wenn solche Änderungen im agilen Umfeld normal sind, sind wir der Meinung, dass dies im Rahmen der WIPRO gut umgesetzt wurde.

Nicht ganz zufrieden sind wir mit dem aktuellen Stand der Testabdeckung. Zwar wollten wir diese im Vergleich zu früheren Projekten bewusst verbessern, jedoch war gegen Projektende kaum noch Zeit dafür vorhanden. An diesem Punkt wird bis zur Abschlusspräsentation jedoch noch weiter gearbeitet.

Projektplanung

Die Projektplanung war zu Beginn nicht ganz einfach, da noch unklar war, wo genau mehr Zeit investiert werden muss. Der agile Ansatz hat sich hier klar bewährt. Die Projektmeetings mit dem Auftraggeber waren stets sehr hilfreich und konnten bei offenen Fragen oder Problemen weiterhelfen. Rückblickend wären wöchentliche Meetings teilweise sogar sinnvoll gewesen, da oft viele Punkte zu diskutieren waren und frühere Klärungen hilfreich gewesen wären.

Die Arbeit mit GitLab-Issues empfanden wir als sehr praktisch, da der Projektstand jederzeit sichtbar war. Dieses Vorgehen würden wir in zukünftigen Projekten wieder so wählen. Das Risikomanagement war aus unserer Sicht in Ordnung, da wir uns den Risiken während des Projekts stets bewusst waren. Teilweise war es jedoch etwas mühsam, da über längere Zeit unklar war, ob gewisse Features (z. B. der Parkplatzzähler) überhaupt noch benötigt werden.

Das agile Arbeiten insgesamt empfanden wir als sehr passend, insbesondere da wir beide berufsbegleitend studieren und nebenbei arbeiten. Es war nicht immer möglich, täglich Meetings abzuhalten, weshalb es sehr praktisch war, Aufgaben über Issues zu übernehmen und selbstständig abzuarbeiten.

Lerneffekt

Der Lerneffekt aus dem Projekt war für uns beide sehr gross. Zwar haben wir bereits die Module *MobPro* und *MobLab* besucht, konnten hier aber besonders von der Grösse und Komplexität der Applikation profitieren. Die modulare Architektur eignete sich sehr gut, um ein sauberes App-Design zu lernen und zu vertiefen.

Sehr spannend war zudem die intensive Arbeit mit AI-Tools. Dabei konnten wir viel darüber lernen, wie diese effizient eingesetzt werden können, um im Gesamtkontext Zeit zu sparen. Auch das Multi-Tenant-Konzept in Kombination mit der Backend-Anbindung war sehr interessant, da hier bereits eine saubere und gut durchdachte Lösung bestand, an der weitergearbeitet werden konnte.

Zusätzlich konnten wir unsere Kenntnisse in Jetpack Compose, moderner Android-Architektur sowie unseren realistischen Blick auf die Stärken und Grenzen von AI weiter vertiefen.

7.1.3. Persönliche Reflexion Raphael

Rückblickend bin ich mit meinem Einsatz im Projekt zufrieden. Ich habe durchgehend aktiv am Projekt mitgearbeitet, konnte viele Aufgaben selbstständig umsetzen und dabei sehr viel lernen. Besonders die Arbeit an einer umfangreichen und technisch anspruchsvollen Applikation war für mich sehr lehrreich.

Der *AI-First*-Ansatz war für mich persönlich eines der spannendsten Elemente des Projekts. Der Umgang mit verschiedenen AI-Tools und deren gezielter Einsatz im Entwicklungsprozess war sehr interessant. Mit der Zeit wurde ich deutlich besser darin, sinnvolle und präzise Prompts zu formulieren, was sich direkt auf die Qualität der Resultate ausgewirkt hat. Dieser Lernprozess war für mich klar spürbar und wird mir auch in zukünftigen Projekten weiterhelfen.

Was ich rückblickend anders machen würde, ist zu Projektbeginn mehr Zeit in das vollständige Verständnis des Projekts zu investieren, bevor mit der eigentlichen Umsetzung gestartet wird. Zu Beginn habe ich einige Dinge zu schnell umgesetzt, die später nochmals überarbeitet werden mussten. Eine gründlichere Einarbeitungsphase hätte hier vermutlich Zeit gespart.

Elemente, die ich in zukünftigen Projekten wieder genauso umsetzen würde, sind der *AI-First*-Ansatz, die regelmässigen Statusberichte und Meetings sowie das strukturierte Vorgehen bei der Abarbeitung der Issues. Diese Arbeitsweise hat sich für mich bewährt und wesentlich zum guten Projektfortschritt beigetragen.

7.1.4. Persönliche Reflexion Samuel

7.2. Ausblick

7.2.1. Persönlicher Ausblick

Das Projekt ist mit der Abgabe der Dokumentation noch nicht vollständig abgeschlossen. Aus persönlichem Interesse wird der Programmcode in der Zeit bis zur Abschlusspräsentation noch leicht überarbeitet und refactored, um kleinere Unschönheiten (auch im Zusammenhang mit dem AI-First-Ansatz) auszubessern. Zudem wird weiter an der Testabdeckung gearbeitet, da zwischen Abgabe der Dokumentation und der Abschlusspräsentation noch knapp einen Monat Zeit bleibt.

Am 15. Januar 2026 findet die Abschlusspräsentation statt. Anschliessend soll die App an den Auftraggeber Jürg Nielispach übergeben werden, der danach die Verantwortung für die Weiterentwicklung und den Betrieb der Applikation übernimmt.

7.2.2. Mögliche weiterführende Arbeiten

Als weiterführende Arbeiten ist der Weg offen, die Applikation schrittweise mit neuen Features zu ergänzen. Durch den modularen Aufbau lässt sich die App gut erweitern, ohne bestehende Funktionalitäten stark anzupassen.

Eine mögliche Weiterentwicklung wäre, Studierende gezielt zu befragen, welche weiteren Funktionen sie als sinnvoll erachten. Denkbar wären beispielsweise Erweiterungen wie eine Anzeige von Noten aus MyCampus, ein Credits-Rechner oder zusätzliche Modulbeschreibungen.

Solche Funktionen könnten mit dem aktuellen technischen Aufbau sowohl auf iOS als auch auf Android gut ergänzt werden und den Nutzen der App für Studierende weiter erhöhen.

8. Anhänge

8.1. Aufgabenstellung

Modul:	Dept I WIPRO HS25
Titel:	HSLU Mobile Apps - Android Jetpack Compose - AI First
Ausgangslage und Problemstellung:	<p>Die HSLU-Informatik betreibt eigene Mobile-Apps für die Departemente Informatik und Technik & Architektur auf den Plattformen iOS und Android. Im Rahmen einer Forschungsarbeit wurde die Codebasis für iOS stark erweitert; diese Änderungen müssen nun in der Android Jetpack Compose Codebase nachgerüstet werden. Das bestehende Android-Projekt basiert noch auf XML-Layout und ist nicht an das aktuelle Backend-API angebunden. Dies soll im Rahmen von AI-First Techniken erfolgen und in diesem Zusammenhang sollen diverse Möglichkeiten, Ansätze und Tools evaluiert werden. Mit der passendsten Technologie erfolgt dann die Umsetzung.</p>
Ziel der Arbeit und erwartete Resultate:	<p>Ziel: Jetpack-Compose-App im Google Store verfügbar. Umsetzung: Single-Codebase, Multi-Feature, Multi-Tenant, DDD.</p> <p>Resultate:</p> <ul style="list-style-type: none"> - Artefakte zum Projektmanagement: Statusberichte, Projektplan inkl. fortlaufender Risikoanalyse, Issues/Epics, Stakeholder - Artefakte zum App: <ul style="list-style-type: none"> o Evaluation und Auswahl geeigneter AI-Ansätze & Tools für Mobile Engineering o Implementation der fehlenden Features (bzw. from scratch) in Jetpack Compose, Testing & Dokumentation für aktuelle target-SDKs. o Release des neuesten Stands via GitLab CI/CD und Fastlane in den Google Play Store - Artefakte für HSLU: Projektdokumentation gemäss HSLU-Standards, Zwischen und Schlusspräsentation
Gewünschte Methoden, Vorgehen:	<p>Inkrementelles, iteratives, agiles Vorgehen</p> <ul style="list-style-type: none"> - Regelmässiges Treffen mit dem Auftraggeber inkl. Statusbericht am Vorabend. - Statusbericht umfasst: Erledigte Aufgaben, Zwischenstand, Risikobewertung, nächste Arbeiten - Nutzung von Projektmanagement-Tools zur Nachvollziehbarkeit von Issue-Tracking
Kreativität, Methoden, Innovation:	<p>Cutting-edge-Technologien einsetzen und evaluieren (z. B. moderne AI-Tools, Jetpack Compose, CI/CD-Automatisierungen)</p> <ul style="list-style-type: none"> - KI-gestützte Unterstützung für Entwickler aktiv in den Prozess einbinden, dokumentieren und reflektieren - Flexibilität, je nach Projekterkenntnissen die Gewichtung des AI-Anteils

	zu variieren
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.

Projektteam

Student:in 1:	Samuel Kurmann (samuel.kurmann@stud.hslu.ch), Raphael Eiholzer (raphael.eiholzer@stud.hslu.ch)
Betreuer:in:	Nietlispach Jürg

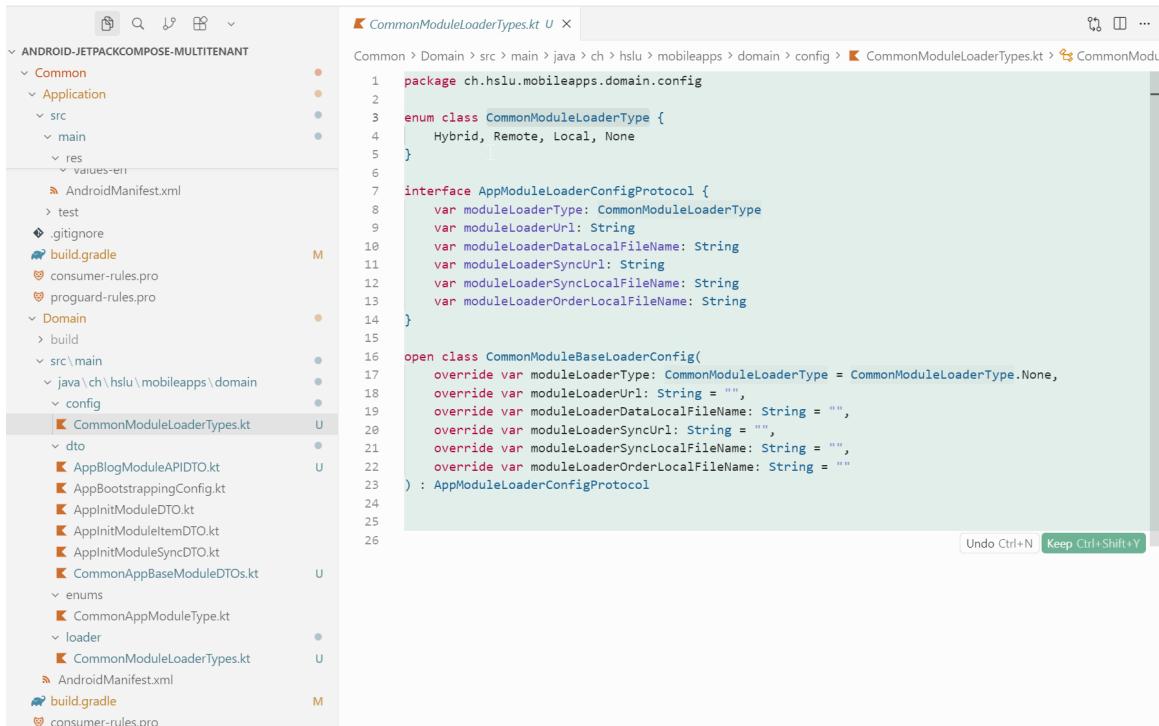
Auftraggeber

Firma:	HSLU Departement Informatik
Ansprechperson:	Jürg Nietlispach
Funktion:	Senior Wissenschaftlicher Mitarbeiter
Strasse:	
PLZ/Ort:	
Telefon:	041 349 30 82
E-Mail:	juerg.nietlispach@hslu.ch
Website:	

8.2. Zusatzmaterial

8.2.1. AI-First: Weitere Inhalte

In diesem Beispiel hat die AI die Clean-Code-Prinzipien nicht sauber eingehalten und teilweise mehrere Klassen in derselben Datei zusammengefasst.



The screenshot shows an IDE interface with a file tree on the left and a code editor on the right. The file tree shows a project structure for 'ANDROID-JETPACKCOMPOSE-MULTITENANT' with various modules like 'Common', 'Application', 'Domain', and 'config'. In the 'config' module, the file 'CommonModuleLoaderTypes.kt' is selected. The code editor displays the following Kotlin code:

```

package ch.hslu.mobileapps.domain.config

enum class CommonModuleLoaderType {
    Hybrid, Remote, Local, None
}

interface AppModuleLoaderConfigProtocol {
    var moduleLoaderType: CommonModuleLoaderType
    var moduleLoaderUrl: String
    var moduleLoaderDataLocalFileName: String
    var moduleLoaderSyncUrl: String
    var moduleLoaderSyncLocalFileName: String
    var moduleLoaderOrderLocalFileName: String
}

open class CommonModuleBaseLoaderConfig(
    override var moduleLoaderType: CommonModuleLoaderType = CommonModuleLoaderType.None,
    override var moduleLoaderUrl: String = "",
    override var moduleLoaderDataLocalFileName: String = "",
    override var moduleLoaderSyncUrl: String = "",
    override var moduleLoaderSyncLocalFileName: String = "",
    override var moduleLoaderOrderLocalFileName: String = ""
) : AppModuleLoaderConfigProtocol

```

Abbildung 28.: Von AI generierter Code mit Schwächen

Hier hat die AI ein Layout erstellt. Wie zu sehen ist, sind zwar alle Elemente vorhanden, das Ergebnis wirkt jedoch optisch nicht gelungen.



Abbildung 29.: Von AI generiertes Layout

8.2.2. AppBootstrappingViewModel

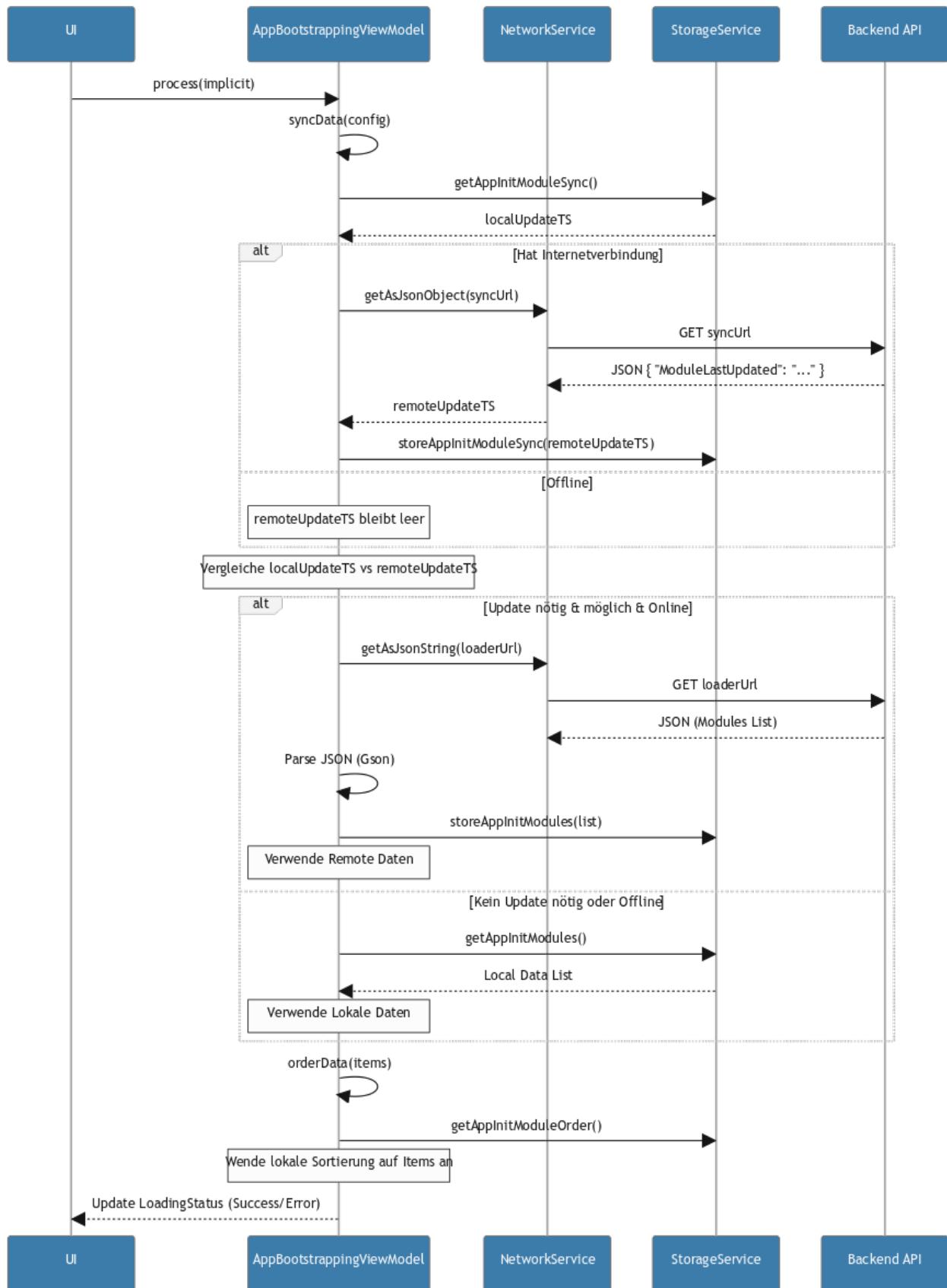


Abbildung 30.: Sequenzdiagramm der Klasse AppBootstrappingViewModel

8.2.3. CommonApplicationBaseModuleLoader

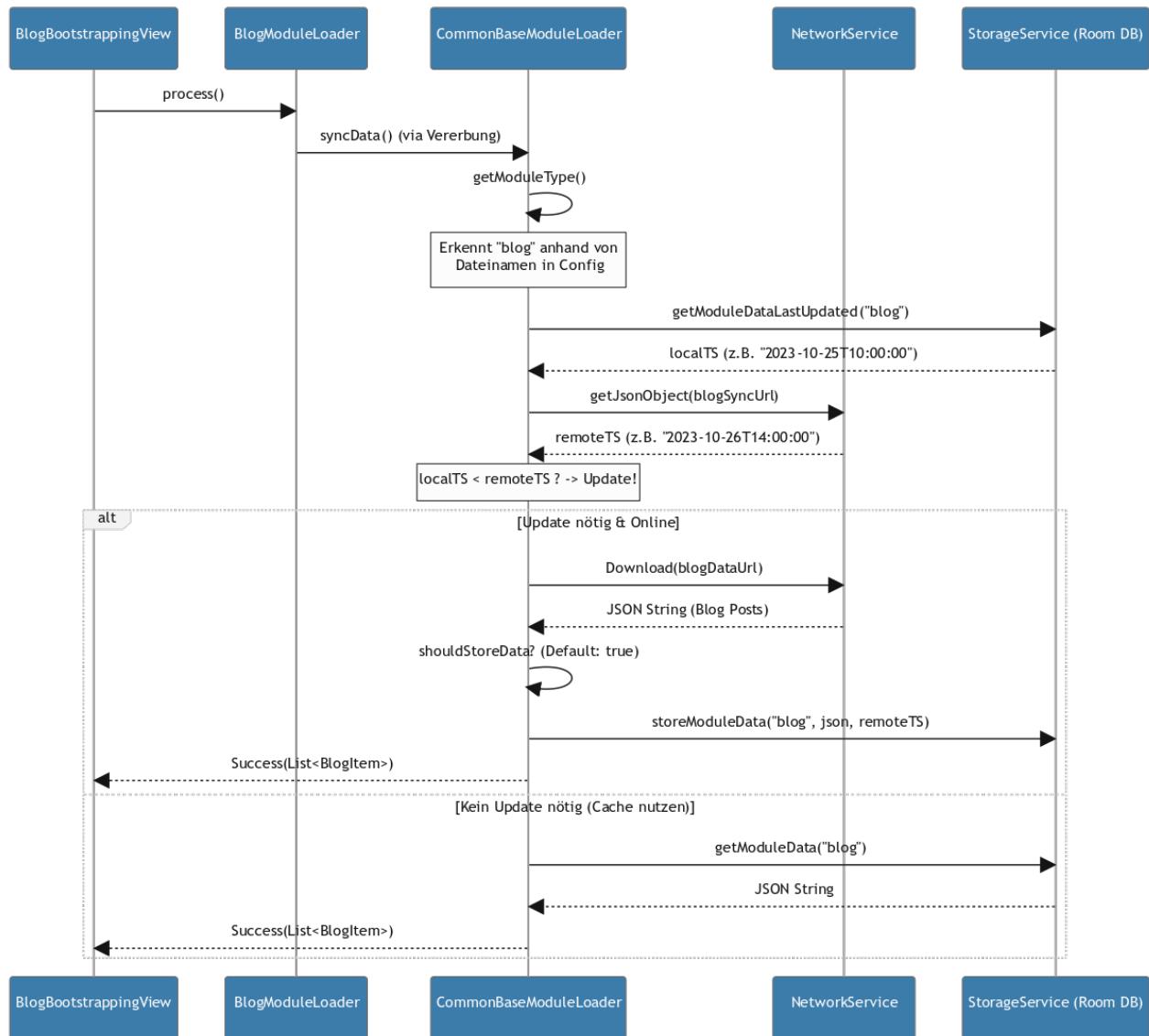


Abbildung 31.: Sequenzdiagramm der Klasse CommonApplicationBaseModuleLoader

8.2.4. Sprintboard – Sprint 1

hslu-mobile / mobile-apps / live / apps / android-jetpackcompose-multitenant / Ticketboards

Backlog Suchen

The image shows a ticket board with the following structure:

- Sprint 1:**
 - Aufgabenstellung überarbeiten und signieren (Documentation, To do)
 - Evaluation AI Tools (Evaluation, To do)
 - Allgemein / Diverses (Administration, In progress)
 - Dokumentation (Administration, In progress)
 - Meetings (Administration, In progress)
 - Projektplanung (Administration, In progress)
- Sprint 2:**
 - Multi-Tenant Projektstruktur (Common, In progress)
 - App-Layout, Menüstruktur (Common, To do)
 - Teststruktur (Common, To do)
 - CI/CD, Fastlane (Common, To do)
- Sprint 3:**
 - Lokalisierung (Common, To do)
 - Lokaler Speicher (Room) (Common, To do)
 - Netzwerkdienst (Common, To do)
 - Error-Bildschirme (Common, To do)
- Sprint 4:**
 - Zwischenpräsentation vorbereiten und halten (Documentation, To do)
 - Lokale App-Einstellungen (Einstellungen / Settings) (Common, Feature)
 - Raumsuche / Roomsearch (Feature, To do)
 - Blog / Blog (Feature, To do)
 - Info / About (Feature, To do)
- Sprint 5:**
 - Links / Links (Feature, To do)
 - News / News (Feature, To do)
 - Mensa / Canteen (Feature, To do)
 - Events / Events (App-Deaktiviert, Feature)
- Sprint 6:**
 - Stundenplan / Timetable (App-Deaktiviert, Feature)
 - Testate / Attestations (App-Deaktiviert, Feature)
 - Anmeldungen / Enrolments (App-Deaktiviert, Feature)
- Sprint 7:**
 - Trails (App-Deaktiviert, Feature)
 - Parkplätze / Parking (App-Deaktiviert, Feature)

At the bottom left, there is a link: "Alle Tickets anschauen".

8.2.5. Sprintboard – Sprint 2

hdlu/mobile / mobile-apps / live / apps / android-jetpackcompose-multitenant / Ticketboards

Backlog Suchen

Sprint 1: Allgemein / Diverses (Administration), Dokumentation (Administration), Meetings (Administration), Projektplanung (Administration), Multi-Tenant Projektstruktur (Common), App-Layout, Menüstruktur (Common), Teststruktur (Common), CI/CD, Testlane (Common). Status: In progress.

Sprint 2: Lokalisierung (Common), Lokaler Speicher (Room) (Common), Netzwerkdienst (Common), Error-Bildschirme (Common). Status: In progress.

Sprint 3: Zwischenpräsentation vorbereiten und halten (Documentation), Lokale App-Einstellungen (Einstellungen / Settings) (Common, Feature), Raumsuche / Roomsearch (Feature), Blog / Blog (Feature), Info / About (Feature). Status: To do.

Sprint 4: Links / Links (Feature), Mensa / Canteen (Feature), Events / Events (App-Dekriviert, Feature), Anmeldungen / Enrollments (App-Dekriviert, Feature). Status: To do.

Sprint 5: News / News (Feature), Mensa / Canteen (Feature), Events / Events (App-Dekriviert, Feature). Status: To do.

Sprint 6: Stundenplan / Timetable (App-Dekriviert, Feature), Testate / Attestations (App-Dekriviert, Feature), Anmeldungen / Enrollments (App-Dekriviert, Feature). Status: To do.

Sprint 7: Trails (App-Dekriviert, Feature), Parkplätze / Parking (App-Dekriviert, Feature), Evaluation AI Tools (Evaluation). Status: To do.

Closed: Aufgabenstellung überarbeiten und signieren Dokumentation (App-Dekriviert, Feature), Dokumentation (App-Dekriviert, Feature). Status: Done.

8.2.6. Sprintboard – Sprint 3

The screenshot shows a Jira Sprintboard interface with the following structure:

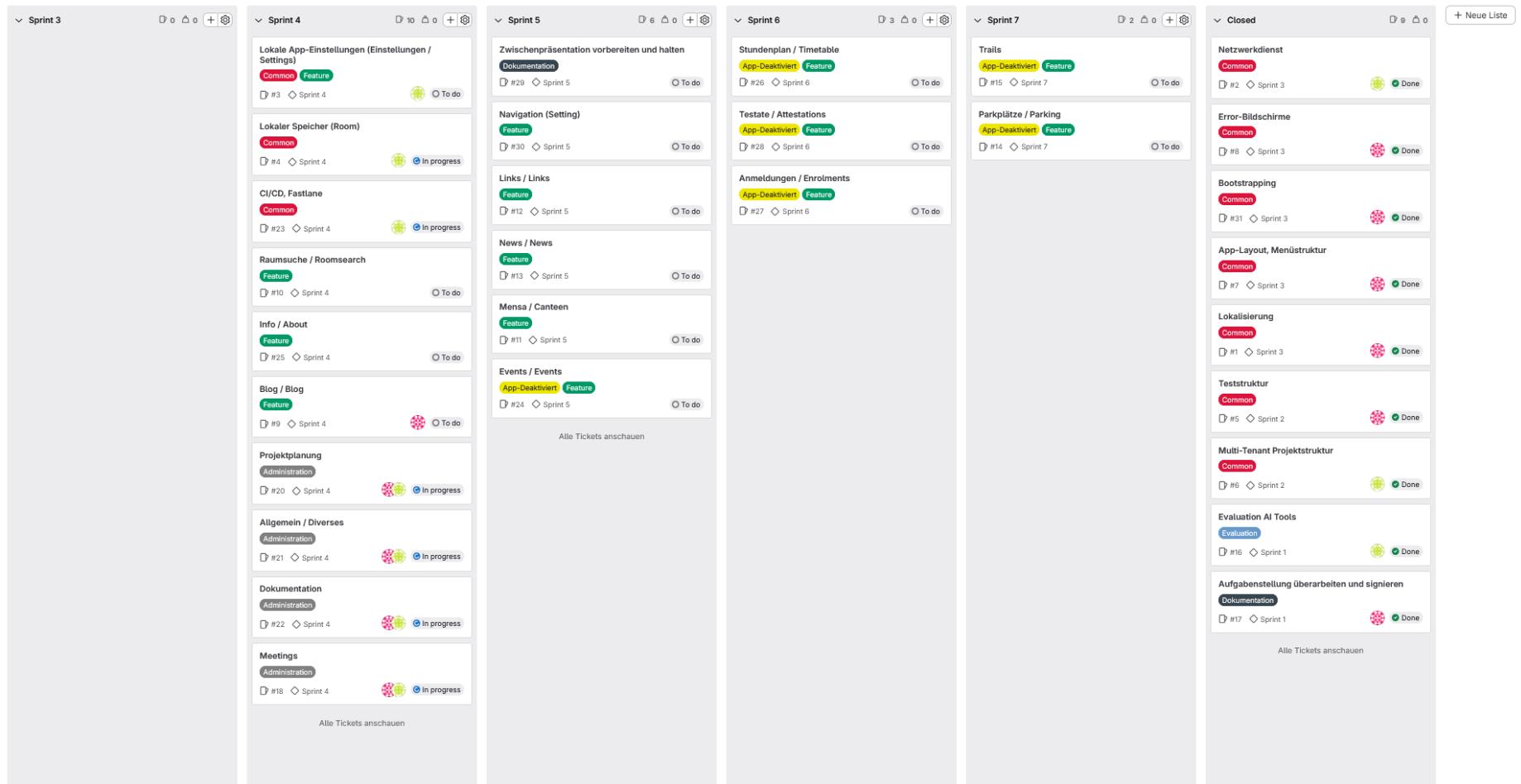
- Project:** android-jetpackcompose-multitenant
- Tickets:** 26
- Merge Requests:** 0
- Verwalten:** >
- Planen:** >
- Tickets:** 26
- Ticketübersichten:** (selected)
- Milestones:**
- Iterationen:**
- Wiki:**
- Anforderungen:**
 - Code >
 - Build >
 - Schützen >
 - Bereitstellung >
 - Betreiben >
 - Überwachen >
 - Analysieren >
- Backlog:** Suchen
- Suchen oder aufrufen ...**

The main area displays the following sprints:

- Sprint 2:**
 - Allgemein / Diverses (Administration): #21, Sprint 2, In progress
 - Dokumentation (Administration): #22, Sprint 2, In progress
 - Meetings (Administration): #20, Sprint 2, In progress
 - Projektplanung (Administration): #21, Sprint 2, In progress
 - CI/CD, Fastlane (Administration): #23, Sprint 3, In progress
- Sprint 3:**
 - Bootstrapping (Common): #31, Sprint 3, In progress
 - Lokalisierung (Common): #32, Sprint 3, In progress
 - Lokaler Speicher (Room) (Common): #33, Sprint 3, To do
 - Raumsuche / Roomsearch (Feature): #34, Sprint 4, To do
 - Error-Bildschirme (Common): #35, Sprint 3, To do
 - App-Layout, Menüstruktur (Common): #36, Sprint 3, To do
- Sprint 4:**
 - Zwischenpräsentation vorbereiten und halten (Documentation): #37, Sprint 4, To do
 - Lokale App-Einstellungen (Einstellungen / Settings) (Common, Feature): #38, Sprint 4, To do
 - Blog / Blog (Feature): #39, Sprint 4, To do
 - Info / About (Feature): #40, Sprint 4, To do
- Sprint 5:**
 - Navigation (Setting) (Feature): #41, Sprint 5, To do
 - Links / Links (Feature): #42, Sprint 5, To do
 - News / News (Feature): #43, Sprint 5, To do
 - Mensa / Canteen (Feature): #44, Sprint 5, To do
 - Events / Events (App-Deaktiviert, Feature): #45, Sprint 5, To do
- Sprint 6:**
 - Stundenplan / Timetable (App-Deaktiviert, Feature): #46, Sprint 6, To do
 - Testate / Attestations (App-Deaktiviert, Feature): #47, Sprint 6, To do
 - Anmeldungen / Enrolments (App-Deaktiviert, Feature): #48, Sprint 6, To do
- Sprint 7:**
 - Trails (App-Deaktiviert, Feature): #49, Sprint 7, To do
 - Parkplätze / Parking (App-Deaktiviert, Feature): #50, Sprint 7, To do
- Closed:**
 - Teststruktur (Common): #51, Sprint 2, Done
 - Multi-Tenant Projektstruktur (Common): #52, Sprint 2, Done
 - Evaluation AI Tools (Evaluation): #53, Sprint 1, Done
 - Aufgabenstellung überarbeiten und signieren (Documentation): #54, Sprint 1, Done

At the bottom left, there are links for "Was ist neu?" and "Hilfe".

8.2.7. Sprintboard – Sprint 4

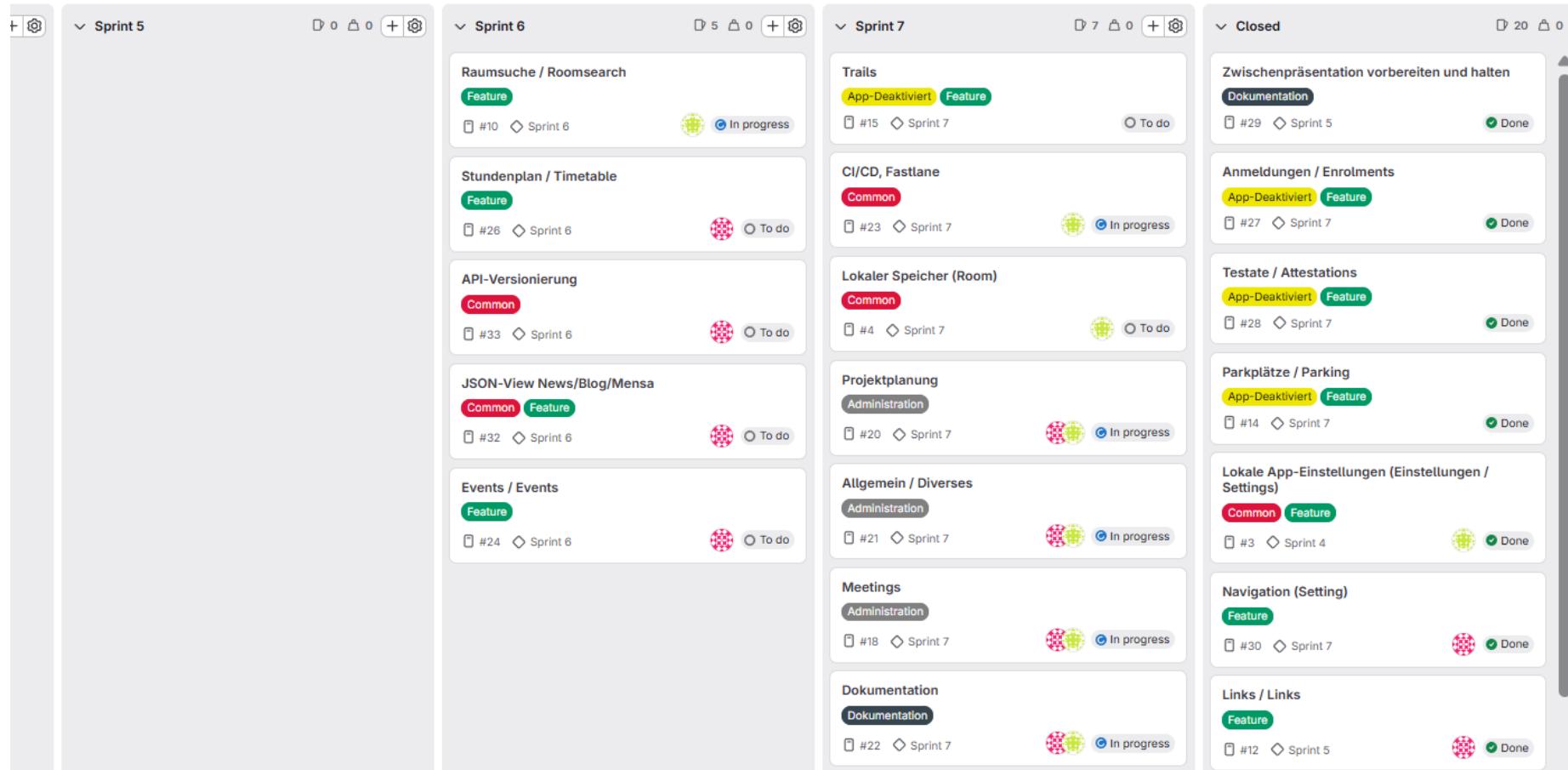


8.2.8. Sprintboard – Sprint 5

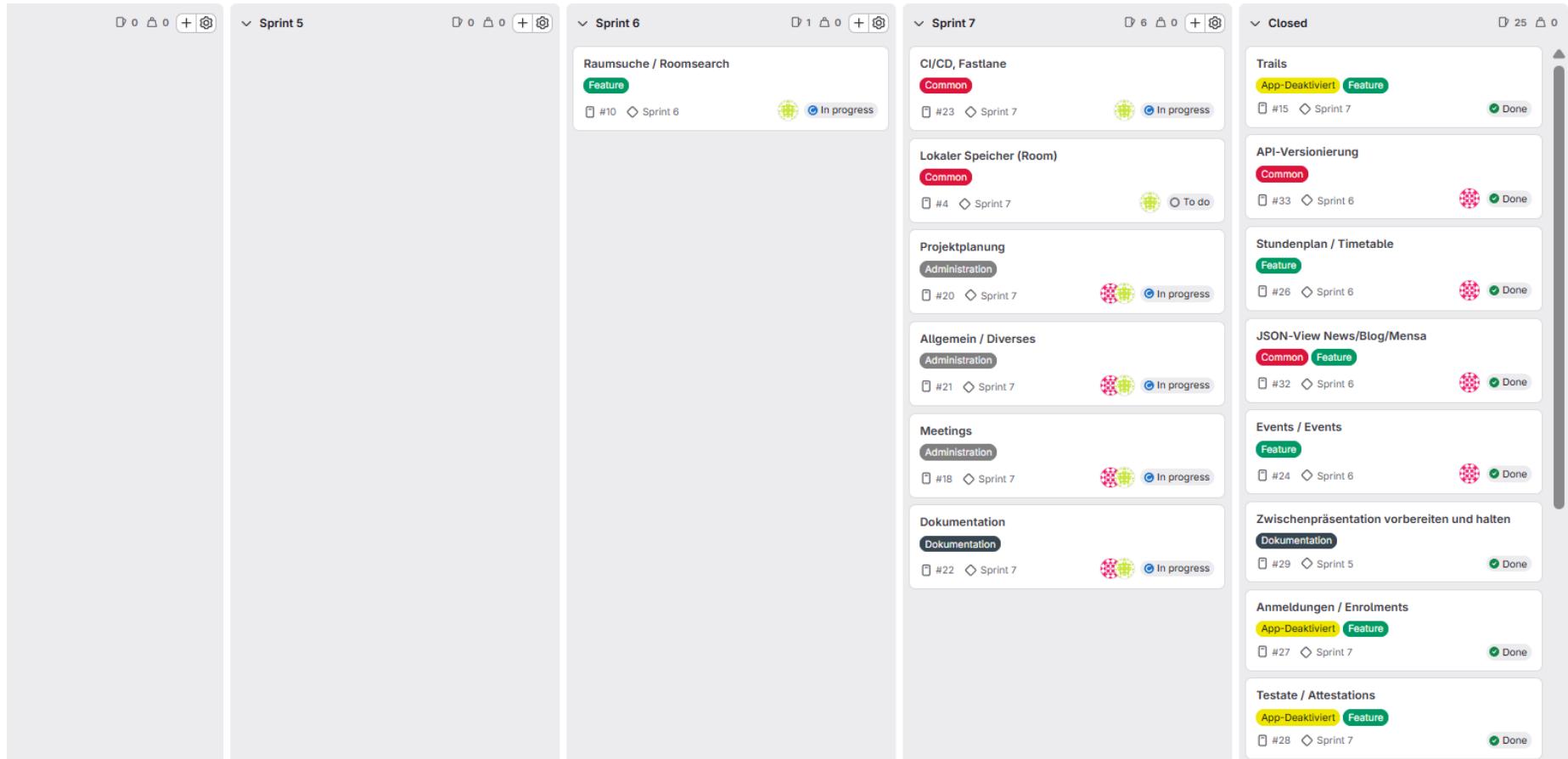
The screenshot shows a Jira Sprintboard interface with the following structure:

- Backlog**: A section at the top left.
- Sprint 3**: An empty column.
- Sprint 4**: An empty column.
- Sprint 5**: Contains 8 tasks:
 - Zwischenpräsentation vorbereiten und halten (Documentation)
 - Raumsuche / Roomsearch (Feature)
 - API-Versionierung (Common)
 - JSON-View News/Blog/Mensa (Common Feature)
 - Projektplanung (Administration)
 - Allgemein / Diverses (Administration)
 - Dokumentation (Documentation)
 - Meetings (Administration)
- Sprint 6**: Contains 3 tasks:
 - Events / Events (Feature)
 - Stundenplan / Timetable (Feature)
 - CI/CD, Fastlane (Common)
- Sprint 7**: Contains 1 task:
 - Traits (App-Deaktiviert, Feature)
- Closed**: A column on the right containing completed tasks:
 - Testate / Attestations (App-Deaktiviert, Feature)
 - Anmeldungen / Enrolments (App-Deaktiviert, Feature)
 - Parkplätze / Parking (App-Deaktiviert, Feature)
 - Lokaler Speicher (Room) (Common)
 - Lokale App-Einstellungen (Einstellungen / Settings) (Common Feature)
 - Navigation (Setting) (Feature)
 - Links / Links (Feature)
 - Info / About (Feature)
 - News / News (Feature)
 - Mensa / Canteen (Feature)

8.2.9. Sprintboard – Sprint 6



8.2.10. Sprintboard – Sprint 7

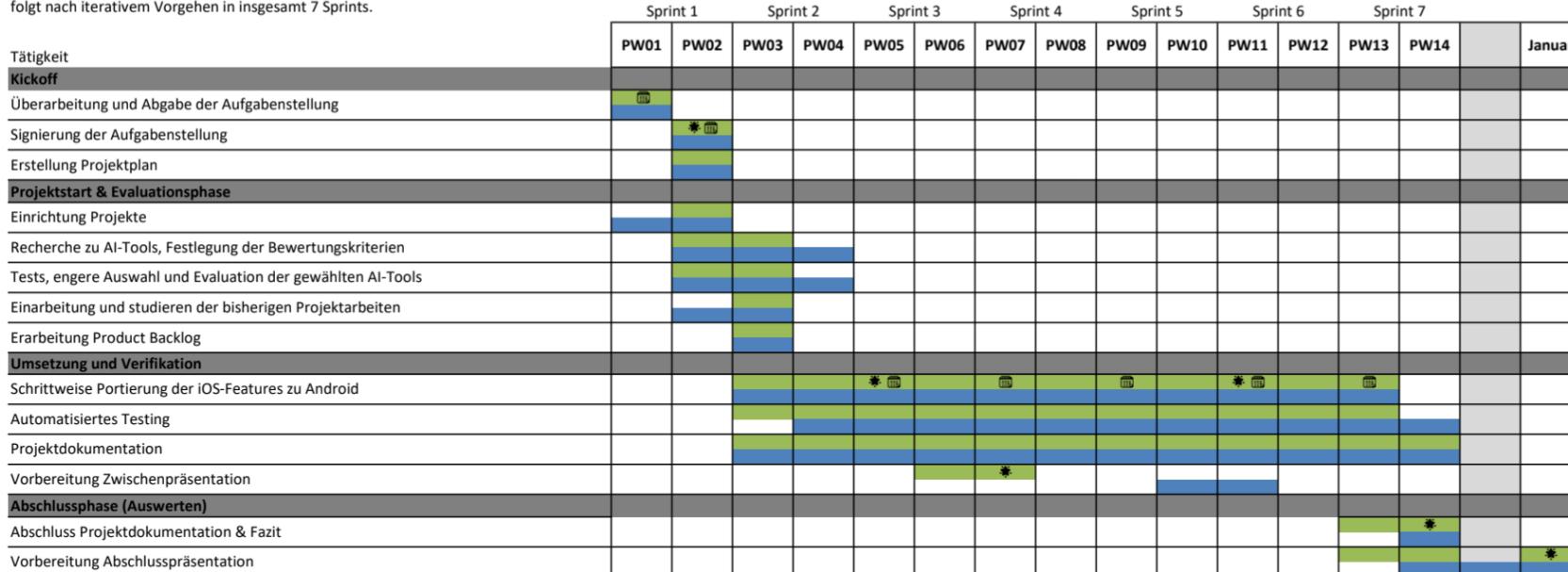


8.2.11. Zeitmanagement - Meilensteine

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 Zeitachse soll einen groben Überblick bieten, wann die wichtigsten Projektschritte stattfinden. Die effektive Projektplanung und Umsetzung folgt nach iterativem Vorgehen in insgesamt 7 Sprints.

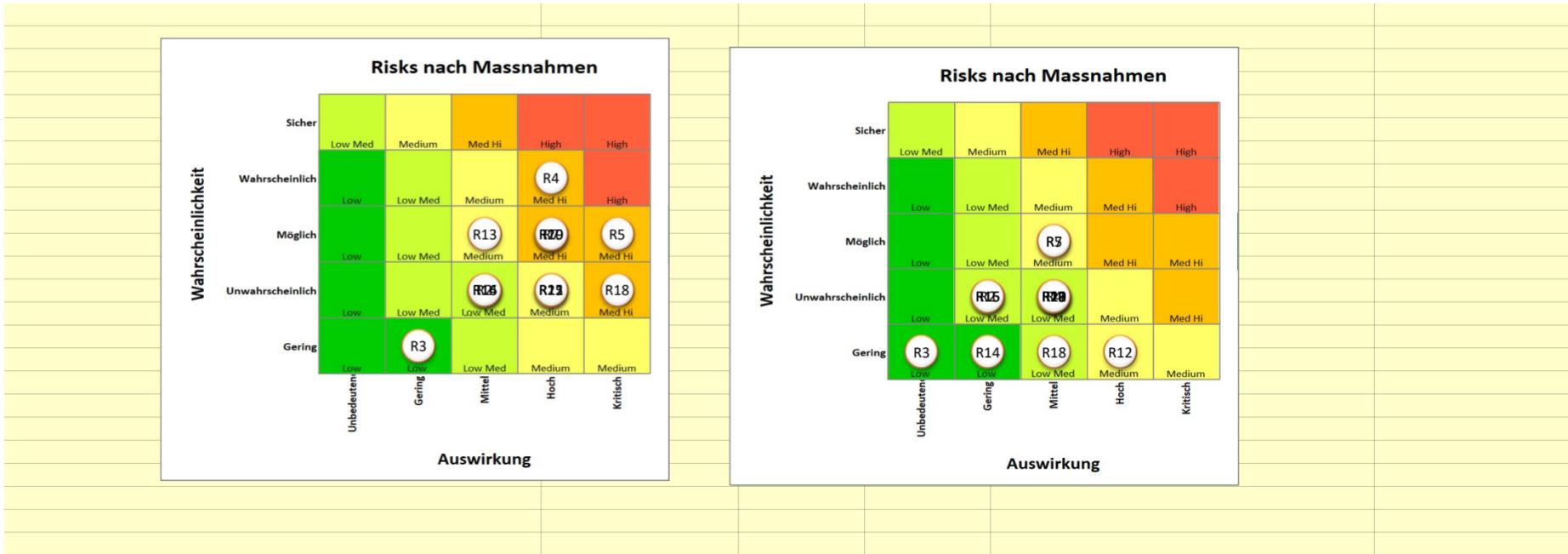


- MS01: Signierte Aufgabenstellung
- MS02: Lauffähige App, "Common"-Features abgeschlossen
- MS03: Zwischenpräsentation
- MS04: Features umgesetzt
- MS05: Dokumentation abgegeben
- MS06: Präsentation abgeschlossen



8.2.12. Risikoanalyse - Risikomatrix

ID	Risiko	Beschreibung des Risikos	Wahrscheinlichkeit	Auswirkung	Risk score	Präventive Massnahmen	Wahrscheinlichkeit '	Auswirkung '	Risk score '
R1 Technische Risiken									
R2	Multi-Tenant-Architektur	Ein Aufbau der Architektur gestaltet sich als sehr komplex, Gewisse Redundanzen in der Projektstruktur bleiben vorhanden.	Unwahrscheinlich	Mittel	6	Planung der Architektur vor der Entwicklung; Tests beim bauen der verschiedenen Apps	Unwahrscheinlich	Gering	4
R3	Integration von AI-Tools im Entwicklungsprozess	AI-Tools können im Entwicklungsprozess wenig Vorteile erbringen.	Gering	Gering	2	Frühzeitige Evaluierung der Tools mit Testscenarien; Dokumentierung dieser Ergebnisse	Gering	Unbedeutend	1
R4	CI/CD-Pipeline mit Fastlane und GitLab	Das automatische bauen/publizieren der App wird durch technische Probleme erschwert/verhindert.	Wahrscheinlich	Hoch	16	Frühes bauen und deployen, damit früh getestet und rechtzeitig reagiert werden kann	Unwahrscheinlich	Mittel	6
R5	Ausfall HSLU-Apps Backend oder GitLab	Ein benötigtes Programm für die Entwicklung ist für eine Zeitperiode nicht verfügbar.	Unwahrscheinlich	Mittel	6	Lokale Arbeiten bereithalten, um für ca. einen Tag auf Architektur verzichten zu können	Gering	Mittel	3
R6	Fachliche Herausforderung, fehlendes Know-How	Für die Programmierung gewisser Komponenten reicht das fachliche Know-How nicht aus.	Möglich	Kritisch	15	Schulungen vor Projektbeginn; Wissensaustausch im Team fördern; Rechtzeitig erkennen und um Hilfe bitten	Möglich	Mittel	9
R7	Komplexität der Jetpack Compose Migration	Eine Migration der bestehenden XML-Layouts zu JetPack Compose führt aus technischen Gründen zu Schwierigkeiten.	Möglich	Hoch	12	Schrittweise Migration; Essentielle Komponenten zuerst; Wichtige Features vorab testen	Möglich	Mittel	9
R8									
R11 Projektmanagement									
R12	Unklare / Falsch aufgefasste Anforderungen	Von den Projektmitarbeitern wurden Anforderungen falsch verstanden und deshalb falsch umgesetzt.	Unwahrscheinlich	Hoch	8	Regelmäßige Projektaufgaben an Auftraggeber; kurze Feedback-Zyklen, damit nicht viel Zeit verloren geht	Gering	Hoch	4
R13	Unrealistische Zeitplanung	Durch unrealistische Zeitplanung reicht am Ende des Projektes die Zeit nicht mehr für die Verfollständigung des Projektes.	Möglich	Mittel	9	Aufwandsschätzung durch beide Teammitglieder; Pufferzeiten einplanen; regelmäßige Überprüfung der Meilensteine	Unwahrscheinlich	Mittel	6
R14	Änderungen im Projektumfang	Der Projektumfang wird vom Projektleiter erweitert, da das Projekt gut voranschreitet und weitere Features in der Projektzeit umgesetzt werden könnten.	Unwahrscheinlich	Mittel	6	Nicht erwartet, aber keine grossen negativen Auswirkungen, da Zusatzfeatures nur in Frage kommen, wenn genügend Zeit dafür bleibt.	Gering	Gering	2
R15	Kommunikationsprobleme mit Auftraggeber	Der Auftraggeber fällt temporär aus oder muss sich anderen Arbeiten widmen. Regelmäßiges Feedback kann nicht eingeholt werden.	Unwahrscheinlich	Hoch	8	Regelmäßige Statusmeetings; Aufgaben länger in Zukunft planen	Unwahrscheinlich	Gering	4
R16	Krankheit von Student	Ein Student fällt wegen Krankheit aus und kann für gewisse Zeit nicht am Projekt arbeiten.	Unwahrscheinlich	Mittel	6	Projektdokumentation aktuell halten und gute Teamkommunikation, dass im Notfall andere Personen übernehmen kann	Unwahrscheinlich	Gering	4
R17 Qualitätsrisiken									
R18	Mangelhafte Dokumentation	Die Dokumentation wurde nicht genügend nachgeführt und weist Lücken auf.	Unwahrscheinlich	Kritisch	10	Dokumentation direkt schreiben, nachdem Arbeit dazu erledigt wurde;	Gering	Mittel	3
R19	Instabile App-Performance	Die App läuft aus technischen Gründen langsam oder stürzt ab.	Möglich	Hoch	12	Frühzeitige (Last-)Tests unter realen Bedingungen durchführen; Code-Optimierung als Teil des Entwicklungsprozesses	Unwahrscheinlich	Mittel	6
R20	Sicherheitslücken durch ungetestete Libraries oder APIs	Sicherheitslücken wurden unbewusst in die App eingespielt.	Möglich	Hoch	12	Verwendung nur von geprüften und aktualisierten Libraries; evtl. Security-Scans in CI/CD-Pipeline einbauen	Unwahrscheinlich	Mittel	6
R21	Fehlende Testdeckung	Die App wurde nicht genügend getestet und weist Fehler auf, die die Benutzbarkeit erschweren.	Unwahrscheinlich	Hoch	8	Teststrategie festlegen und umsetzen; evtl. Code-Coverage-Überwachung in CI/CD-Pipeline einbauen	Unwahrscheinlich	Mittel	6

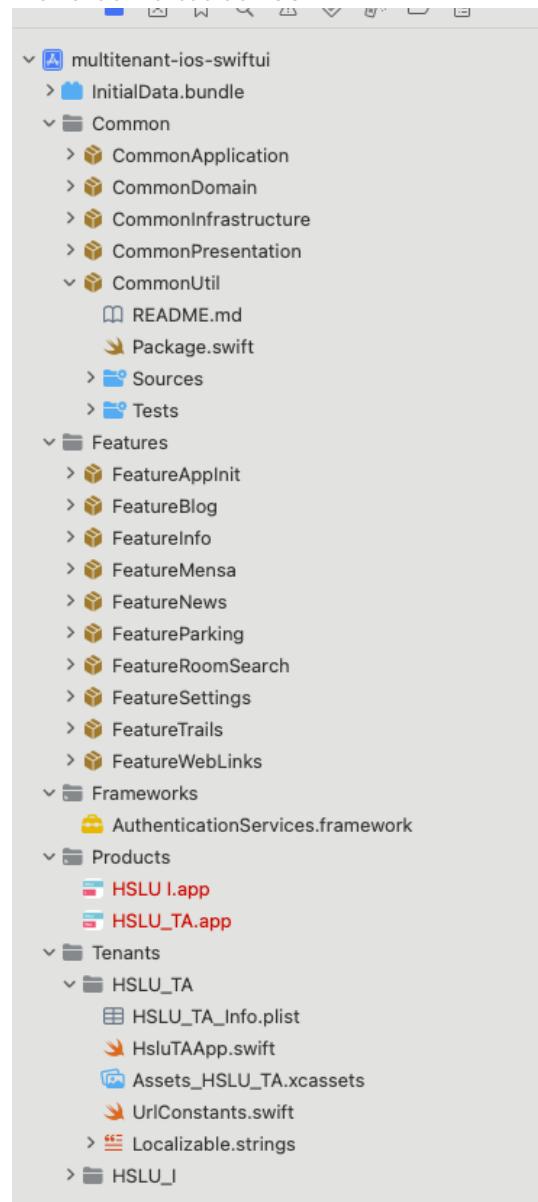


8.2.13. Aufgabenstellung für AI Tools

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])
// AppNewsModuleItemAPIDTO
{
    var remoteSyncData : (CommonAppBaseModuleSyncAPIIDT0?,
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
(CommonAppBaseModuleSyncAPIIDT0?, 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?

8.2.14. 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.

8.2.15. 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!

### 8.2.16. 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 Abhangigkeiten\*\*: Klare Trennung via Swift Packages

---

## ## \*\*3. Architektur-Entwurf fur 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?

## 8.2.17. 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.

### 8.2.18. Zeiterfassung

Während unseres Projektes haben wir unsere Arbeitszeit konsequent in den GitLab-Issues erfasst. Um nachvollziehen zu können, wie viel Zeit insgesamt investiert wurde – ohne diese regelmäßig manuell zusammenzählen – haben wir ein kurzes Skript geschrieben, das die erfassten Zeiten über die GitLab-API ausliest und aggregiert.

Das Ergebnis ist für uns besonders spannend, da es einen guten Überblick darüber gibt, in welchen Bereichen besonders viel Zeit investiert wurde. Die Auswertung ist jedoch nicht vollständig repräsentativ. So wurde der Aufwand für wiederverwendbare Features initial dem jeweiligen Hauptticket (z. B. *News*) zugeordnet. Diese Funktionalitäten konnten später auch in anderen Bereichen (z. B. *Blog*) genutzt werden, wodurch dort ein geringerer Zeitaufwand ausgewiesen ist.

Auch bei der Dokumentation zeigt sich ein plausibles Bild: Wir haben pro Seite mit ungefähr einer Stunde für saubere Dokumentation gerechnet, was sich in der Auswertung in etwa bestätigt.

Ticket	Zeit (h)
API-Versionierung	5.00
JSON-View News/Blog/Mensa	9.50
Bootstrapping	16.00
Navigation (Setting)	7.00
Zwischenpräsentation vorbereiten und halten	9.00
Testate / Attestations	0.00
Anmeldungen / Enrolments	0.00
Stundenplan / Timetable	11.00
Info / About	4.00
Events / Events	6.00
CI/CD, Fastlane	8.00
Dokumentation	46.00
Allgemein / Diverses	2.00
Projektplanung	20.25
Meetings	36.50
Aufgabenstellung überarbeiten und signieren	2.00
Evaluation AI Tools	18.00
Trails	0.00
Parkplätze / Parking	0.00
News / News	2.00
Links / Links	2.00
Mensa / Canteen	2.00
Raumsuche / Roomsearch	30.00
Blog / Blog	13.00
Error-Bildschirme	5.00
App-Layout, Menüstruktur	7.00
Multi-Tenant Projektstruktur	12.00
Teststruktur	3.00
Lokaler Speicher (Room)	19.00
Lokale App-Einstellungen (Settings)	8.00
Netzwerkdienst	15.00
Lokalisierung	4.00
<b>Gesamtzeit</b>	<b>322.25</b>

### 8.2.19. Backend

Das Backend stellt dabei die zentrale Schnittstelle für alle mobilen Applikationen dar und sorgt dafür, dass Inhalte plattformunabhängig bereitgestellt werden können. Module wie News, Blog, Mensa oder WebLinks werden serverseitig verwaltet und den Apps über definierte APIs zur Verfügung gestellt.

Durch diesen Ansatz können neue Inhalte oder Anpassungen vorgenommen werden, ohne dass eine neue Version der App im App Store oder Play Store veröffentlicht werden muss. Dies vereinfacht den Betrieb und reduziert den Wartungsaufwand auf Seiten der mobilen Applikationen deutlich.

Zudem ermöglicht das gemeinsame Backend eine einheitliche Datenbasis für iOS und Android, wodurch Unterschiede im Funktionsumfang oder im angezeigten Inhalt vermieden werden können.

Für den Betrieb der Applikationen existieren mehrere Backend-Umgebungen. Neben dem produktiven Backend für den regulären Betrieb gibt es separate Backends für die Departemente Informatik (I) sowie Technik & Architektur (TA). Zusätzlich steht eine eigene QA- bzw. Entwicklungsumgebung zur Verfügung, welche für Tests, neue Features und technische Anpassungen genutzt wird.

The screenshot shows a web-based application interface titled "HSLU QA". On the left, there is a dark sidebar menu with categories like "Module Management", "Generic Modules", "Campus Trails Module", and "Campus Weblinks". The main content area is titled "AppModuleTypes List" and displays a table with the following data:

#	Identifier	Status	Actions
1	ABOUT	2	
2	WEBLINKS	2	
3	MENSA	2	
4	ABOUT	2	
5	BLOG	2	
6	ROOMSEARCH	2	
7	NEWS	2	
8	CAMPUSROOMSEARCH	2	
9	EVENTS	2	
10	TIMETABLE	2	
11	PARKING	1	
12	ABOUT	2	
13	SETTINGS	2	

Abbildung 32.: Screenshot des QA-Backends

### 8.3. Protokolle und Statusberichte

Die Protokolle und Statusberichte dokumentieren den zentralen Kommunikationsfluss zwischen Auftraggeber und Studierenden und wurden jeweils vor den Meetings (Statusberichte) sowie nach den Meetings (Protokolle) erstellt, um den aktuellen Projektstand festzuhalten und die nächsten Arbeitsschritte zu planen.

### 8.3.1. 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)

### 8.3.2. 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

### 8.3.3. 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)

### 8.3.4. 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](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](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)

### 8.3.5. 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

### 8.3.6. Projektstatus-Meeting (11.12.2025)

## 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.

### 8.3.7. 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

### 8.3.8. 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.

### 8.3.9. 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"> <li>- In Android Studio gezielt Emulatoren für verschiedene Auflösungen und Android-Versionen anlegen</li> <li>- Views vor allem mit flexiblen Layout (z.B. Flexbox) anlegen</li> <li>- Nach physischen Testgeräten fragen</li> </ul>

### 8.3.10. 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"><li>• Klare Aufteilung der Kapitel</li><li>• Regelmässige interne Reviews</li><li>• Abgleich mit Bewertungsraster und früheren Beispielen</li></ul>
Versteckte Probleme durch spätes Refactoring: Refactorings kurz vor Abschluss können unbeabsichtigt bestehende Funktionalität beeinflussen oder neue Bugs einführen.	<ul style="list-style-type: none"><li>• Nur gezielte, klar abgegrenzte Refactorings</li><li>• Keine funktionalen Änderungen</li><li>• Kurze Regressionstests nach jedem grösseren Eingriff</li></ul>
Unentdeckte Plattform- oder Geräteabhängigkeiten: Bestimmte Probleme treten nur auf einzelnen Android-Versionen oder Gerätetypen auf und bleiben unbemerkt.	<ul style="list-style-type: none"><li>• Nur beschränkt beeinflussbar, Zahl der Testgeräte ist beschränkt</li><li>• Eventuell hat HSLU noch weitere Geräte</li></ul>

# 9. Abkürzungs-, Abbildungs-, Tabellen-, KI-, Formelverzeichnis

## 9.1. Abkürzungsverzeichnis

## 9.2. Abbildungsverzeichnis

1.	Projekte in GitLab . . . . .	2
2.	Projekte mit (relevanten) Branches . . . . .	2
3.	Screenshots der bestehenden Apps . . . . .	3
4.	iOS-Projektstruktur in XCode . . . . .	4
5.	Meilensteinplan des Projekts . . . . .	8
6.	Darstellung des Scrum-Prozesses im Projekt . . . . .	9
7.	Backlogs in GitLab zu Projektbeginn . . . . .	9
8.	Beispiel einer User Story mit Definition of Done (DoD) . . . . .	10
9.	Risikoanalyse und Risikomatrix des Projekts . . . . .	11
10.	Projektstruktur im Android Studio . . . . .	12
11.	Build-Variants in Android Studio . . . . .	14
12.	Features in Android-Studio . . . . .	28
13.	Abhängigkeiten in Pfeilrichtung erlaubt/vorhanden . . . . .	28
14.	Struktur der Features am Beispiel <i>Event</i> . . . . .	29
15.	Aufforderung zum Versionsupdate . . . . .	31
16.	Flussdiagramm Menü-Synchronisation . . . . .	32
17.	Screenshots des News-Features . . . . .	34
18.	Screenshots des Blog-Features . . . . .	35
19.	Screenshot aus dem Backend: Umschalten der Views (Blog & News) . . . . .	35
20.	Screenshots des Mensa-Features . . . . .	36
21.	Screenshots des Events-Features . . . . .	37
22.	Screenshots der Features . . . . .	38
23.	Ablaufdiagramm . . . . .	39
24.	Screenshots des Kalender-Features . . . . .	43
25.	Screenshots des Kalender-Widgets . . . . .	45
26.	Teststruktur im Android Studio . . . . .	46
27.	Analysebeispiele in Cursor . . . . .	50
28.	Von AI generierter Code mit Schwächen . . . . .	58
29.	Von AI generiertes Layout . . . . .	58
30.	Sequenzdiagramm der Klasse AppBootstrappingViewModel . . . . .	59
31.	Sequenzdiagramm der Klasse CommonApplicationBaseModuleLoader . . . . .	60
32.	Screenshot des QA-Backends . . . . .	94

### **9.3. Tabellenverzeichnis**

### **9.4. KI-Verzeichnis**

### **9.5. Formelverzeichnis**

## 10. Literaturverzeichnis

*Acccompanist: WebView wrapper for Jetpack Compose.* (n. d.). Verfügbar 11. Dezember 2025 unter <https://github.io/accompanist/web/>

*Android | Jetpack Glance* [Android Developers]. (n. d.). Verfügbar 11. Dezember 2025 unter <https://developer.android.com/develop/ui/compose/glance?hl=de>

*Android CalendarContract | API reference* [Android developers]. (n. d.). Verfügbar 11. Dezember 2025 unter <https://developer.android.com/reference/android/provider/CalendarContract>

*Atlassian. (n. d.). Atlassian: Was ist Scrum?* [Atlassian]. Verfügbar 16. Oktober 2025 unter <https://www.atlassian.com/de/agile/scrum>

*SDK: App architecture* [Android Developers]. (n. d.). Verfügbar 28. Oktober 2025 unter <https://developer.android.com/guide/topics/manifest/uses-sdk-element?hl=de>

*Sitecore: Digital Experience Solutions.* (n. d.). Verfügbar 11. Dezember 2025 unter <https://www.sitecore.com/de-de/solutions>