Program for Splitting Expenses Implementierungsbericht

Praxis der Softwareentwicklung
Wintersemester 2024/25

PSE-Team

24. April 2025

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Inhaltsverzeichnis

1 Einleitung

Anschließend an das Entwurfsheft wird im Folgenden der Stand der Implementierung des *Program for Splitting Expenses*, kurz *PSE*, beschrieben. Zudem wird der Implementierungsprozess reflektiert und ein Ausblick auf die Qualitätssicherungsphase gegeben.

1.1 Stand der Implementierung

Der Entwurf wurde bis auf die in Abschnitt 1.3 ausgeführten Änderungen vollständig implementiert. Damit sind, unter Ausnahme von funktionseinschränkenden Bugs, auch alle im Pflichtenheft festgelegten Musskriterien abgedeckt.

Die Abdeckung der Wunschkriterien wollen wir im Folgenden erläutern.

⟨*RC*1⟩, ⟨*RC*2⟩ und ⟨*RC*14⟩ wurden so wie im Entwurf ausgearbeitet implementiert. ⟨*RC*3⟩, die Verfügbarkeit der Benutzeroberfläche auf Deutsch und Englisch, wurde mittels der von Android bereitgestellten Hilfsmittel zur Sprachlokalisierung analog zum Entwurf implementiert; die Sprache der Benutzeroberfläche entspricht der Systemsprache des Android-Geräts, auf dem die App eingesetzt wird. Der verwendete Identity-Provider kann wie in ⟨*RC*10⟩ gefordert auf dem Server konfiguriert werden, indem der Inhalt der Konfigurationsdatei angepasst wird; dies wurde wie im Entwurf beschrieben umgesetzt.

Weder entworfen, noch implementiert, wurden ⟨*RC*4⟩, ⟨*RC*5⟩, ⟨*RC*6⟩, ⟨*RC*7⟩, ⟨*RC*8⟩, ⟨*RC*9⟩, ⟨*RC*11⟩, ⟨*RC*12⟩ und ⟨*RC*13⟩.

1.2 Artefakte

Das PSE wird in Form einer *PSE-Distribution* verteilt. Diese enthält die folgenden Artefakte:

- `INSTALL.txt` enthält Installations– und Verteilungsanweisungen zu den anderen Artefakten der PSE-Distribution.

- `pse-server_<version>_amd64.deb` ist ein Debian-Binärpakt für Debian 12 (bookworm) auf `x86_64` Architektur. Dieses installiert den PSE-Server sowie ein F-Droid repository und die notwendige Umgebung, um die Android `apk`-Datei zu veröffentlichen.
- `pse-android_<version>.apk` ist eine Android Applikationsdatei, die die fertig gebaute Android-Applikation enthält, wobei die PSE-Serveradresse auf `https://pse-app.com/` festgelegt ist. Um eine andere Serveradresse zu verwenden, kann die App, wie in `INSTALL.txt` beschrieben, selbst kompiliert werden.
- `pse_<version>.tar.xz` ist ein Debian-Quellcodepaket, welches den gesamten Quellcode des PSE-Server und der Android-App enthält.

1.3 Entwurfsänderungen

1.3.1 Nichtverwendung von OAuth

Im folgenden soll genauer dargelegt werden, weshalb wir uns gegen *OAuth*[1] für die Sicherung geschützter API Endpunkte entschieden haben.

- OAuth ist an sich zunächst nur ein Protokoll für Zugriffsdelegation.
- Die Beweggründe und entsprechende Designentscheidungen hinter OAuth überschneiden sich nur bedingt mit unseren:
  - Die einzige Partei, an die Zugriff delegiert wird, sind wir selbst. OAuths Fähigkeit, zwischen verschiedenen Parteien zu unterscheiden, kommt für uns nie zum Einsatz und verkompliziert nur die Verteilung.
  - Wir müssen zur Initiierung des Anmeldeprozesses lediglich einen Token des OIDC Providers gegen einen eigenen austauschen. Dazu genügt eine einfache Anfrage an unseren Server ohne Involvierung des Nutzers. Es existieren keine OAuth Schnittstellen für diesen Zweck, da OAuth auf die Anmeldung des Nutzers im Browser ausgelegt ist. Die entsprechende Schnittstelle, der Authorization Endpoint, ist entsprechend unangemessen für unseren Use Case und müsste nicht-implementiert gelassen werden.
  - Alle anderen OAuth Schnittstellen abgesehen von der Refresh Schnittstelle hat für uns keine Relevanz. Beispielsweise benötigen wir keine Token Introspection API, da wir das Format unserer eigenen Tokens frei wählen können und entsprechend schon kennen.

[1] `https://oauth.net/code/java/`

Der Vorteil von OAuth wäre gewesen, dass wir uns auf bereits *bestehende Implementierungen*[2] (mit Anpassungen) hätten verlassen können. Jedoch ist auch dies nicht ohne Vorbehalt möglich, da wir zur Zeit des Entwurfs bereits einen bestimmten Anwendungsfall durch unser Pflichtenheft festgelegt hatten. Für diesen hätte die entsprechende Bibliothek korrekt konfiguriert werden müssen, Beispielsweise die Wahl von datenbankgestützten Refresh Tokens, um das im Pflichtenheft beschriebene Beenden von Sessions zu ermöglichen.

Für Authorization Server gibt es im wesentlichen zwei Arten von bestehenden Implementierungen:

Es gibt Bibliotheken, die die Schnittstelle automatisch zusammenbauen, sodass man lediglich Strategien wie Token-Formatwahl und Token-Speicher implementieren muss. Dadurch werden jedoch keine sicherheitskritischen Implementationen vermieden im Vergleich zu unserer Implementation, da wir die Implementierung unserer Wahl von Token-Formaten ebenfalls einfach an Bibliotheken wie *Java-JWT*[3] abgeben.

Ferner gibt es Server Frameworks wie *Spring*[4] und *MitreID*[5], welche erfordert hätten, dass wir entweder den ganzen Server um diese herum aufbauen oder ihn in zwei Teile spalten. Den Server in zwei Teile zu spalten wäre auch die Variante, die fertige Authorization Server wie *Nimbus*[6] und *Keycloak*[7] erforderlich gemacht hätten. Zudem hätten diese Optionen die Verteilung wesentlich verkompliziert und die Verwendung persistenter Schlüssel auf dem Server erfordert, die wir mit unserem eigenen System vermieden haben.

1.3.2 Verwendete Bibliotheken

Folgend aufgelistet sind zusätzlich zu den im Entwurf bereits festgelegten verwendete Bibliotheken inklusive ihres Verwendungsgrunds:

Unit-Tests
: Wir verwenden *JUnit 5*[8] für die Unit-Tests auf dem Server und *JUnit 4*[9] auf dem Client, da letzteres die für Android am besten unterstützte und dokumentierte Version ist.

[2] `https://oauth.net/code/java/`
[3] Siehe Abschnitt 1.3.2
[4] `https://spring.io/`
[5] `https://mitreid-connect.github.io/`
[6] `https://connect2id.com/products/nimbus-oauth-openid-connect-sdk`
[7] `https://www.keycloak.org/`
[8] `https://junit.org/junit5/`
[9] `https://junit.org/junit4/`

Zusätzlich verwenden wir die Testing Utilities `kotlinx-coroutines-test`[10] sowie `androidx.test`[11].

Java-JWT
Um, wie im Entwurf angegeben, Ktor für den Umgang mit JWTs zu nutzen, benötigen wir zusätzlich *Java-JWT*[12], da diese Bibliothek die eigentlichen Schnittstellen zur Validierung von JWTs sowie die zur Generierung von JWTs benötigten JWT Builder zur Verfügung stellt, während das Ktor JWT Plugin diese lediglich in den öffentlichen Schnittstellen verwendet und eine Möglichkeit bietet, sie direkt in den Kontrollfluss des Servers einzubinden.

Coil
Zum Anzeigen von Profilbildern mit Jetpack Compose verwenden wir *Coil*[13]. Coil verwaltet das Herunterladen und Cachen dieser Profilbilder.

Mocking
Sowohl für das Mocking von Interfaces als auch für die Verifikation von Methodenaufrufen in Unit-Tests verwenden wir *MockK*[14].

Logging
Für Logging verwenden wir *Log4j*[15], *SLF4J*[16] sowie *TerminalConsoleAppender*[17].

CLI
Wir verwenden *JOpt Simple*[18], um das einfach gehaltene CLI für den Server umzusetzen.

1.3.3 Verteilungsprozess

Anders als im Entwurf beschrieben, wird der Server nicht auf einer bw-cloud Instanz verteilt, da es beim Testen der bw-cloud mehrfach zu Netzwerkproblemen, insbesondere in Verbindung mit IPv6 kam. Stattdessen wird der Server nun auf einer Debian 12 Instanz eines anderen Anbieters verteilt, der Verteilungsprozess erfolgt jedoch analog.

[10] `https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-test/`
[11] `https://android.github.io/android-test/`
[12] `https://github.com/auth0/java-jwt`
[13] `https://coil-kt.github.io/coil/compose/`
[14] `https://mockk.io/`
[15] `https://logging.apache.org/log4j/2.x/`
[16] `https://www.slf4j.org/`
[17] `https://github.com/Minecrell/TerminalConsoleAppender`
[18] `https://jopt-simple.github.io/jopt-simple/`

Außerdem haben sich im Vergleich zum Entwurf kleinere Änderungen an den notwendigen Schritten zur Verteilung des Servers ergeben. Insbesondere haben sich die Verzeichnisse einiger Dateien geändert um besser mit dem *Filesystem Hierarchy Standard*[19] übereinzustimmen. Der aktuelle Verteilungsprozess ist in der Datei `INSTALL.txt` zu finden.

1.3.4 Bestätigungsdialog bei Gruppenbeitritt über einen Einladungslink

Um ⟨*F*8⟩ *Beitritt zu einer Gruppe* so umzusetzen, dass der Bestätigungsdialog, der nach Öffnen eines gültigen Einladungslinks in der App angezeigt werden soll, bereits Informationen über die Gruppe enthalten darf, für die der Nutzer sich im Prozess des Beitretens befindet, müssen wir ⟨*RM*14⟩ leicht relaxieren, indem wir von nun an zulassen, dass Nutzer, denen der Einladungslinks zu einer Gruppe bekannt ist, von der Regelung ausgenommen werden. Dies bedeutet effektiv, dass Nutzer Informationen über eine Gruppe anfordern dürfen, von der sie bisher kein Mitglied sind, es aber ausschließlich durch Verwendung ihnen bereits bekannter Information (dem gültigen Einladungslink) werden könnten. Damit hat diese Änderung keinerlei Auswirkungen auf unsere Sicherheitsbedenken, die zur ursprünglichen Einführung von ⟨*RM*14⟩ geführt haben.

1.3.5 REST API

An der REST API haben wir verglichen zum Entwurf minimale Änderungen vorgenommen.

Insbesondere haben wir für die Authentizifierung den zuvor verwendeten `/login` Endpoint in zwei separate Endpoints aufgeteilt, da dies eine leichtere Unterscheidung der Art des im Header verschickten Tokens ermöglichte.

Außerdem haben wir den weiteren Endpoint `/join/inviteToken/info` sowie ein dazugehöriges DTO hinzugefügt, um für den in Abschnitt 1.3.4 erwähnten Bestätigungsdialog bereits vor dem Beitritt zur Gruppe den Anzeigenamen der Gruppe abfragbar zu machen.

[19] `https://refspecs.linuxfoundation.org/FHS_3.0/fhs-3.0.html`

1.3.6 Client

Im Folgenden werden alle Entwurfsänderungen, die aussschließlich den Client betreffen, geschildert.

1.3.6.1 Exception Handling

Im Entwurf war vorgesehen, dass für jeden Fehler, der im Model auftritt, eine Exception geworfen wird, die dann möglichst weit oben in der Composable Hierarchie der Views gefangen und als Snackbar angezeigt wird. Dieser Ansatz scheitert allerdings daran, dass Jetpack Compose keine `try-catch` Blöcke um Views unterstützt. Anstatt die Model Exceptions in den Views zu fangen, fangen wir sie daher stattdessen direkt in den ViewModels und schicken gegebenenfalls auftretende Fehlermeldungen über einen `SharedFlow` an die `Scaffold` Composable der Views, welche diese dann wie vorgesehen in der Snackbar anzeigt. Um dies zu ermöglichen, erben unsere ViewModels nicht mehr von `ViewModel`, sondern von einer eigenen `ErrorFlowViewModel` Klasse, welche die reguläre `ViewModel` Klasse durch einen solchen Flow und Methoden zum automatischen Fangen und Behandeln von Exceptions erweitert.

Zudem haben wir die im Entwurf vorgesehenen `ModelException` überarbeitet, um alle möglichen Fehlerfälle abzudecken. Diese vom Model erzeugten Exceptions werden dann im ViewModel aufgefangen und erzeugen lokalisierte Fehlermeldungen an den Nutzer.

1.3.6.2 Navigation

Ähnlich zu den Exceptions in Abschnitt 1.3.6.1 besitzt unsere `BaseViewModel` Basisklasse Abstraktionen zur Navigation. Im Entwurf hat diese direkt einen `NavController` bekommen. Allerdings ist dies etwas problematisch, da es dies die ViewModels zu sehr an die Jetpack Compose Implementierung koppelt. Stattdessen werden jetzt über einen Channel NavigationEvents geschickt, die dann in der `MainActivity` verarbeitet werden.

1.3.6.3 Änderungen an den ViewModels

Da die ViewModels sehr eng mit dem eigentlichen UI Code der Views, welcher erst in der Implementierung finalisiert wurde, zusammenhängen, weichen einige dieser von der Beschreibung im Entwurf ab. Insbesondere geben einige Attribute, die vorher Flows zurückgegeben hätten jetzt StateFlows zurück, da dies dem ViewModel ermöglicht,

einen Startwert festzulegen, anstatt diesen in der View setzen zu müssen. Um außerdem die ViewModels nicht bei jeder navigation neu zu erstellen, überschreibt jede von diesen außerdem eine `onEntry` Methode, die in der in Abschnitt 1.3.6.1 beschriebenen Basisklasse definiert ist. Diese wird bei jedem neuen Laden der View ausgeführt und setzt den internen Zustand dieses, soweit nötig, zurück.

1.3.6.4 Erweiterung der Fassade

Im Zuge der Entwicklung sind uns zusätzliche Methoden aufgefallen, die von der Fassade angeboten werden müssen. Diese sind:

`isAuthenticated(): Boolean`
: Prüft und antwortet, ob eine gültige Sitzung existiert.

`getUserById(id: String): User`
: Gibt einen Nutzer mit der gegebenen ID zurück

`getGroupById(id: String): Group`
: Gibt eine Gruppe mit der gegebenen ID zurück

1.3.6.5 Aufteilung der Repository Klassen

Die Repository Klassen aus dem Entwurf, also beispielsweise `UserRepo`, haben wir für bessere Modularität weiter aufgeteilt. Die *Observe* Methoden zum beobachten des Zustands sind jetzt teil der `UserRepoObservables` Schnittstelle, während die *Zustandsändernden* und *Refresh* Operationen teil von `UserRepoRemoteCalls` sind. Beide Schnittstellen werden von der `UserRepo` Schnittstelle geerbt. Diese Aufteilung erlaubt es uns die Logik zur Verwaltung des lokalen Zustands von der serverbezogenen Logik zu trennen. `UserRepoObservables` wird von `LocalUserRepo` implementiert, welches zusätzliche Methoden zur Aktualisierung des lokalen Zustands bietet und von `RemoteUserRepo` verwendet wird. `RemoteUserRepo` ist dann das effektiv genutzte `UserRepo`.

Analog wurden auch `GroupRepo` und `TransactionRepo` aufgeteilt.

1.3.6.6 Vereinfachung des Gruppenzustands

Die `GroupShortData` und `GroupFullData` Datenklassen aus dem Entwurf haben wir durch die `GroupData` Klasse ersetzt, welche die Informationen von beiden vereint. Entsprechend wurden auch die Methoden des `GroupRepo` angepasst.

1.3.6.7 Transparentere Sitzungen in RemoteAPI

Um den Zustand einer Sitzung von außen abfragen zu können, wird der `RemoteAPI` zwei weitere Methoden hinzugefügt:

`refreshSession(): Boolean`
: Überprüft, ob eine Sitzung existiert und erneuert diese, falls notwendig.

`registerOnSessionDestroyed(callback)`
: Erlaubt es, ein Callback anzugeben, welches bei Verlust der Sitzung aufgerufen wird, um externe Daten aufzuräumen. Die Sitzung geht gewöhnlich beim Abmelden verloren.

1.3.6.8 Änderungen an ViewModels

Alle entworfenen ViewModels erben in der Implementierung zusätzlich von `BaseViewModel`. Dieses ist für die Fehlerbehandlung zuständig und vereinfacht die Implementierung der eigentlichen ViewModels.

2 Arbeitsweise

Im Folgenden wird unsere Arbeitsweise während des Implementierungsprozesses geschildert.

2.1 Verwendung von Versionskontrollsystemen

Als Versionskontrollsystem wurde Git verwendet. Wir verwendeten Feature Branches und bei Fertigstellung Merges in bzw. Rebases des `main`-Branches. Für kleinere Änderungen, die sofortigen Effekt haben sollten, wurde teilweise aber kein dedizierter Feature Branch erstellt.

Es wurde kein CI/CD verwendet. Stattdessen wurden vor jedem Merge in den `main`-Branch die bestehenden Tests manuell ausgeführt.

2.2 Kommunikation

Um im Team ausstehende Features sowie zu behebende Bugs zu dokumentieren, verwendeten wir GitLab Issues. Außerdem verwendeten wir einen gemeinsamen Discord Server, auf dem wir dedizierte Kommunikationskanäle für Diskussionen über verschiedene Komponenten bzw. Teile des Systems anlegten. So konnten wir uns gegenseitig schnell Feedback geben und Fragen beantworten.

2.3 Frühzeitige Unit-Tests

Einzelne Komponenten wollten wir möglichst früh testen, indem wir die von der Komponente verwendeten Schnittstellen mithilfe von MockK mocken. Wir können so z.B. testen, dass die richtigen Methoden in der richtigen Reihenfolge aufgerufen werden oder die zu testende Komponente für gewisse Inputs die korrekten erwarteten Outputs

liefert, vorausgesetzt, die Komponente erhält über die gemockten Schnittstellen die richtigen intermediären Daten.

Da ein Mocken unserer durch *Ktorm* gegebenen Datenbankabstraktion unrealistisch ist, beschlossen wir stattdessen, eine eigene Testumgebung anzulegen, bei der für jeden Unit-Test eine leere, nicht-persistente Datenbank vorliegt. Um eine Methode einer Komponente zu testen, legt man einen solchen Unit-Test an und initialisiert die Datenbank mit Testdaten, führt die zu testende Methode aus und überprüft dann den anschließend vorliegenden Inhalt der Datenbank auf Korrektheit.

Dies lieferte uns außerdem die Möglichkeit, die Verwaltungskomponenten mit *Test Driven Development* umzusetzen, was ein hohes Vertrauen in die Korrektheit unserer Implementierung lieferte.

2.4 Planung

Am Anfang der Implementierungsphase stellten wir das in Abbildung 2.1 gezeigte Gantt-Diagramm als Implementierungsplan auf. Dieser unterteilt die Implementierung in Arbeitspakete, die wir entsprechend der Arbeitsverteilung in der Entwurfsphase den entsprechenden Leuten zugeteilt haben. Jedes Arbeitspaket umfasst hierbei auch gleich das Schreiben dazugehöriger Unit-Tests.

2.5 Abhängigkeiten und Arbeitsaufteilung

Um den in Abbildung 2.1 dargestellten Plan aufstellen zu können, mussten wir Abhängigkeiten unter den Arbeitspaketen bestimmen. Aufgrund unseres modularen Entwurfs und der loosen Kopplung unserer Module, die jeweils nur von definierten Schnittstellen abhängen, ließen sich nach dem Anlegen eines Projektskelettes und der Implementierung von grundlegenden Strukturen wie beispielsweise häufig verwendeten Datenklassen die Arbeitspakete parallelisieren und auf die gesamte Gruppe aufteilen. Insbesondere entschieden wir uns wegen der klar definierten API zwischen Client und Server dazu, alle Arbeitspakete, ausschließlich der anfangs zu implementierenden gemeinsam verwendeten Komponenten, auf ein Server- und ein Client-Team aufzuteilen. Diese Aufteilung schien uns fair, aber flexibel zu sein.

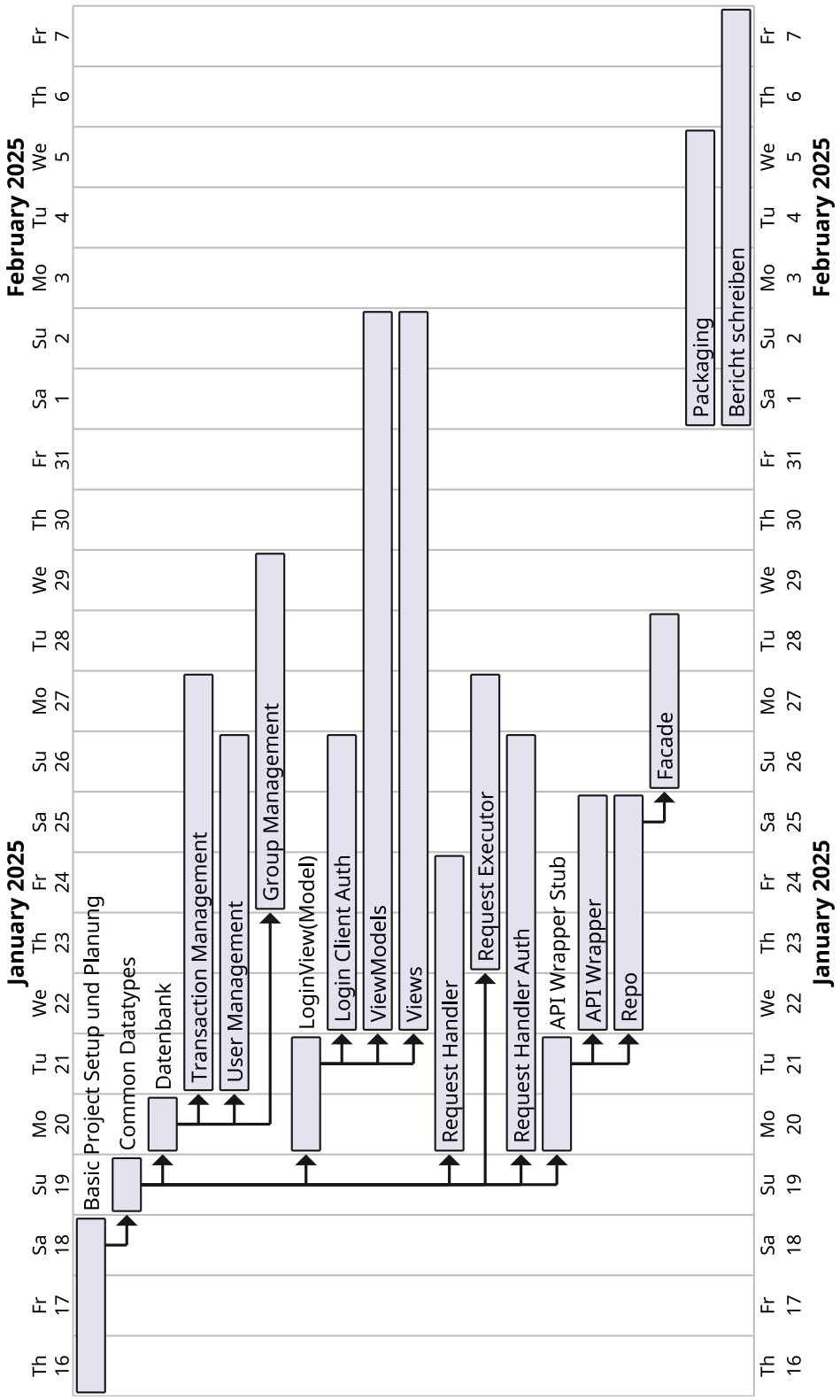

Abbildung 2.1: Implementierungsplan

2.6 Kritischer Pfad

Der kritische Pfad umfasst zuerst einmal die Arbeitspakete *Basic Project Setup und Planung* sowie *Common Datatypes*, da, wie in Abschnitt 2.5 erläutert, alles Andere auf diesen zwei Arbeitspaketen aufbaut. Von dort aus gibt es mehrere mögliche kritische Teilpfade, die es in Erwägung zu ziehen galt: Aufgrund unserer Entscheidung, die folgenden Arbeitspakete entweder auf das Client- oder Server-Team aufzuteilen, liegen auf einem solchen Teilpfad des kritischen Pfades entweder nur Client- oder nur Server-Arbeitspakete.

2.6.1 Server

Ein möglicher kritischer Teilpfad für die Implementierung des Servers umfasst das Arbeitspaket *Datenbank*, da auf diesem alle Verwaltungskomponenten aufbauen. Da wir hierbei aber von einer schnellen Fertigstellung ausgingen, schätzten wir das Risiko für eine Verzögerung hier als gering ein. Die restlichen Arbeitpakete des Servers lassen sich alle parallel implementieren und im Notfall flexibel auf andere Personen im Server-Team umverteilen, weshalb wir die gesamte Implementierung des Servers als recht risikoarm einstuften.

2.6.2 Client

Auf dem Client stellen die Arbeitspakete, die die Modell-Komponente betreffen, also *API Wrapper Stub*, *API Wrapper*, *Repo* und *Facade* einen kritischen Teilpfad dar. Wir identifizierten den Pfad, der diese Arbeitspakete enthält, als den kritischen Pfad unserer Implementierung. Desweiteren stellen die Arbeitspakete, die die View- bzw. ViewModel-Komponenten betreffen einen weiteren ähnlich kritischen Pfad dar.

3 Reflektion

Wir reflektieren im Folgenden unseren Implementierungsprozess.

3.1 Planungsumsetzung

Unsere Einschätzung für den Aufwand der Server Arbeitspakete sollte sich bewahrheiten. Alle Server Arbeitspakete konnten in der geplanten Zeit und teilweise sogar vor den geplanten Fristen fertiggestellt werden.

Für die Client Arbeitspakete haben wir teilweise länger als geplant gebraucht. Hier hätte es sich rückblickend gelohnt den aktuellen Fortschritt genauer im Blick zu behalten und die unterschiedlichen Komponenten kontinuierlich zu integrieren. Zudem wäre es nützlich gewesen für jede Komponente mehr als einen Verantwortlichen zu haben, um auftretende Bugs schneller behandeln zu können.

Server Den Server integrierten wir nach der Outside-In-Strategie, was aufgrund der sauber definierten Schnittstellen und den auch für Randfälle klar definierten Verhaltensweisen quasi reibungslos verlief.

Client Auf dem Client wurden jeweils ViewModel und dazugehöriges View stets gleichzeitig entwickelt und nach Verfügbarkeit direkt integriert. Parallel dazu wurde die Model Komponente entwickelt. So konnten alle, die am Client beteiligt waren, gleichzeitig arbeiten. Die Integration zwischen ViewModel und Model fand dann nach Verfügbarkeit statt.

Client-Server Die Integration zwischen Client und Server setzten wir als Big-Bang-Integration um. Wir gingen ursprünglich davon aus, dass diese Art der Integration aufgrund der klar definierten API und den verwendeten DTOs ohne große Probleme funktionieren sollte. Aufgrund von fehlender Dokumentation und einer nur halb typensicheren API traten bei unserer Verwendung von Ktor allerdings einige unerwartete Probleme auf, weshalb wir letztendlich doch ein paar Probleme mit der Integration hatten und diese länger dauerte als erwartet.

Rückblickend hätten wir den Server und Client vielleicht eher funktionsorientiert integrieren sollen, da wir so bereits anfangs auf die durch Ktor verursachten Probleme aufmerksam geworden wären.

3.2 Umgang mit Bugs

Fehler, die während der Implementierung einer Komponente durch die währenddessen oder davor entwickelten zugehörigen Unit-Tests auffielen, wurden sofort behoben. Wann immer wir bei der späteren Integration verschiedener Komponenten einen Fehler in einer von anderen Personen entwickelten Komponente feststellten, so schrieben wir einen kurzen Bug Report in den entsprechenden Discord Channel. Die Personen, die an den Komponenten beteiligt waren, hatten dann die Aufgabe, diesen Bug zügig zu beheben. Diese Art der Kommunikation ermöglichte direktes Feedback und Diskussion, sorgte aber aufgrund der Vermischung von Diskussionen und Bug Reports teilweise dafür, dass die Channel etwas unübersichtlich wurden, was wir aber durch die Verwendung von Threads zu minimieren versuchten. Im Nachhinein hätten sich für grundlegendere Bugs vielleicht GitLab Issues oder Bug Reports angeboten.

4 Ausblick

Bereits während der Implementierung haben wir unseren Code durch Unit-Tests und mittels manueller Integration getestet. Wir wollen nun einen Überblick über diese Tests sowie über bereits bekannte Bugs, die es in der Qualitätssicherungsphase zu beheben gilt, verschaffen.

4.1 Übersicht über die Unit-Tests

4.1.1 Server

Request Dispatcher Tests In den Request Dispatcher Tests wird die Funktionalität des `RequestDispatchers` getestet, die durch das von diesem implementierte Interface `RequestExecutor` garantiert wird. Hierzu mocken wir die vom `RequestDispatcher` durch Dependency Injection verwendeten Verwaltungskomponenten und garantieren, dass entweder die richtigen Methoden aufgerufen werden oder die Ergebenisse der Verwaltungskomponenten richtig verarbeitet und zurückgegeben werden. Außerdem wird überprüft, dass die Methoden des `RequestDispatchers` entsprechend scheitern, wenn die verwendeten Methoden der Verwaltungskomponenten scheitern.

Request Router Tests In den Request Router Tests wird die Funktionalität des `RequestRouters` getestet. Hierbei mocken wir die vom `RequestRouter` durch Dependency Injection verwendeten Komponenten und testen, dass der `RequestRouter` die in der REST API festgelegten HTTP-Requests korrekt abarbeitet, indem wir überprüfen, dass die richtigen Methoden der gemockten Komponenten aufgerufen werden oder der empfangene Response Status Code richtig ist bzw. der Response Body die richtigen Daten enthält.

Token Authenticator Tests In den Request Router Tests wird die Funktionalität der `TokenAuthenticator` Klasse. Dort sind zentral die Implementierungen der Validierung von Access Tokens und OIDC Tokens sowie der Generierung von Access Tokens verortet (also von allen Json Web Tokens). Da die Methoden lediglich verschiedene Schnittstellen

von `java-jwt` zusammenstecken, zielen unsere Tests darauf ab, auf High-Level zu unterscheiden, ob die relevanten Sicherheitsmerkmale von JWTs tatsächlich geprüft werden. Vor einfachen Tests, welche prüfen, dass die generierten Tokens auch valide sind stehen dabei vor allem die Tests im Vordergrund, welche prüfen, dass das Verfälschen einzelner Merkmale der JWTs darin resultiert, dass der Token abgelehnt wird.

Refresh Token Tests In den Refresh Token Tests wird die Funktionalät von der `RefreshToken` Klasse getestet. Es wird getestet, ob die Methoden tatsächlich die Dinge produzieren die sie Testen sollten. Da die Validität von Refresh Tokens direkt an das Vorhandensein in der Datenbank gebunden ist, gibt es keine Sicherheitsmerkmale, die sich in Unit Tests ausdrücken lassen.

Verwaltungskomponententests Die Verwaltungskomponenten werden jeweils einzeln getestet. Um eine Verwaltungskomponente zu testen verwenden wir die in Abschnitt 2.3 beschriebene nicht-persistente Datenbank, initialisieren diese mit einem Testszenario und testen dann die Korrektheit des Verhaltens von spezifischen Methoden der zu testenden Verwaltungskomponente. Hierbei werden alle uns bekannten Randfälle der jeweiligen Verwaltungskomponente überprüft.

Config Tests In den Config Tests testen wir, dass `Config` eine valide Konfigurationsdatei korrekt einliest und das Einlesen einer nicht-validen Konfigurationsdatei fehlschlägt. Außerdem wird das korrekte Parsen verschiedener Anbindungsmöglichkeiten der Datenbank überprüft.

Datenbanktests In den Datenbanktests überprüfen wir, dass die von uns für Unit-Tests verwendete nicht-persistente Datenbank korrekt initialisiert wird.

4.1.2 Client

Model Tests Wir bilden Klassen auf Unittestklassen ab. So hat zum Beispiel die `Model` Klasse eine `ModelTest` Unittestklasse, welche alle öffentlichen Methoden von `Model` testet. Ziel ist es alle öffentlichen Methoden zu testen. Dieses Ziel haben wir in der Model Komponente bisher nur zur Hälfte erreicht, weshalb wir die restlichen Unittests in der nächsten Phase schreiben werden.

Ausgenommen von den Unittests ist die `RemoteClient` Klasse, da sie minimale Logik enthält, und das Testen von dieser dem Testen von ktor entsprechen würde.

UI Tests Da die Views keine Logik enthalten, existieren für diese auch keine Unittests. Das gleiche gilt für einige ViewModel methoden, wie etwa die verschiedenen `navigate` Methoden, die nur eine Methode der Navigationsabstraktion aufrufen und damit nicht sinnvoll testbar sind.

4.2 Bekannte verbleibende Bugs

Uns sind beim Schreiben dieses Berichts keine Bugs bekannt.