



Program for Splitting Expenses Entwurf

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	7
1.1. Anforderungsergänzungen	7
1.1.1. Einladungslinks	7
1.1.2. Fehlermeldungen	7
1.1.3. Profilbilder von Gruppen	10
2. Architektur	11
2.1. Struktureller Aufbau	11
2.1.1. Systemstruktur	11
2.1.2. Verteilung	12
2.2. Architektur	15
2.2.1. Client-Server Architektur	15
2.2.2. Client Struktur	15
2.2.3. Client Entwurfsentscheidungen	16
2.2.4. Server Struktur	18
2.2.5. Server Entwurfsentscheidungen	19
2.2.6. Authentifizierung	20
2.2.7. Datenbank	20
2.2.8. Beispielhafte Interaktion der Architekturkomponenten	21
2.3. Komponentenspezifikation	24
2.3.1. Client	24
2.3.2. Server	24
2.4. Schnittstellenspezifikation	27
2.4.1. SQL Schema	27
2.4.2. REST API	29
2.4.3. UI Framework	32
2.4.4. Client-interne Schnittstellen	32
2.4.5. Server-interne Schnittstellen	33
2.5. Externe Bibliotheken	33
2.5.1. Jetpack Compose	33
2.5.2. Ktor	34
2.5.3. Ktorm	34
2.5.4. AppAuth	34

2.5.5.	kotlinx.serialization	34
2.5.6.	Jetpack DataStore	35
3.	Feinentwurf	36
3.1.	Systemweiter Entwurf	36
3.1.1.	Modellierung von Geldbeträgen	36
3.1.2.	Data Transfer Objects (DTOs)	36
3.1.3.	Authentifizierungsablauf	36
3.2.	Server Entwurf	40
3.2.1.	Allgemeine Datentypen	44
3.2.2.	Serverschnittstellen	44
3.2.3.	Request Handler	50
3.2.4.	Request Executor	51
3.2.5.	Nutzerverwaltung	51
3.2.6.	Gruppenverwaltung	51
3.2.7.	Buchungsverwaltung	52
3.2.8.	Datenbankzugriff	52
3.3.	Client Entwurf	57
3.3.1.	UI	57
3.3.2.	ViewModels	59
3.3.3.	Navigation	61
3.3.4.	Lokalisierung	64
3.3.5.	Model Fassade	64
3.3.6.	Model Repositories	75
3.3.7.	Model Data Layer	77
3.3.8.	Preferences	80
4.	Glossar	82
A.	Anhang	83
A.1.	SQL-Schema Definition	83
A.2.	API-Definition	85

Abbildungsverzeichnis

1.1.	Neue Oberfläche für die Gruppenerstellung	8
1.2.	Ein Toast, der eine Fehlermeldung anzeigt	9
2.1.	Systemkomponenten	12
2.2.	Verteilungsdiagramm	13
2.3.	Beispielhafte Interaktion der Architekturkomponenten auf dem Client bei Erstellung einer neuen Buchung	22
2.4.	Beispielhafte Interaktion der Architekturkomponenten auf dem Server bei Erstellung einer neuen Buchung	23
2.5.	Client Komponenten	24
2.6.	Model Komponenten	25
2.7.	Server Komponenten	26
2.8.	Datenbankschema	28
3.1.	Data Transfer Objects	37
3.2.	Auth Code Flow mit PKCE	39
3.3.	Beginn einer Session	41
3.4.	Token Refresh	42
3.5.	Accountdeaktivierung	43
3.6.	Schnittstelle des <i>Request Handler</i>	44
3.7.	Schnittstelle des <i>Request Executor</i>	45
3.8.	Authentifizierungsschnittstelle der Nutzerverwaltung	45
3.9.	Autorisierungsschnittstelle der Gruppenverwaltung	46
3.10.	Nutzeradministrationsschnittstelle der Nutzerverwaltung	47
3.11.	Gruppenadministrationsschnittstelle der Nutzerverwaltung	48
3.12.	Buchungsadministrationsschnittstelle der Nutzerverwaltung	49
3.13.	Detailentwurf des <i>Request Handler</i>	50
3.14.	Allgemeine Datentypen auf dem Server	53
3.15.	Detailentwurf des <i>Request Executor</i>	54
3.16.	Detailentwurf der Nutzerverwaltung	55
3.17.	Detailentwurf der Gruppenverwaltung	56
3.18.	Detailentwurf der Buchungsverwaltung	56
3.19.	Detailentwurf des Datenbankzugriffs	56

3.20. Datenklassen der ViewModels	58
3.21. Views und Viewmodels für das Hauptmenü	62
3.22. Views und Viewmodels für Gruppen	63
3.23. Navigationsgraph der Clientanwendung	65
3.24. Observe-Refresh mit Flows	67
3.25. Model Exceptions	68
3.26. ModelFacade Schnittstelle	70
3.27. User und Group Schnittstelle	72
3.28. SettleRecommendation und Transaction Datenklassen	73
3.29. Implementierung der Fassade	74
3.30. Repositories	78
3.31. Model-interne Datenklassen	79
3.32. RemoteAPI Schnittstelle	81
3.33. Detailentwurf von Preferences und SessionStore	81

1. Einleitung

Anschließend an das Pflichtenheft wird im folgenden der Entwurf des *Program for Splitting Expenses*, kurz *PSE*, beschrieben.

1.1. Anforderungsergänzungen

Während des Entwurfsverfahrens sind uns bestimmte Änderungen und Ergänzungen, die wir an den Anforderungen aus dem Pflichtenheft vornehmen müssen, aufgefallen. Diese werden hier erläutert.

1.1.1. Einladungslinks

Anders als im Pflichtenheft angegeben, werden Einladungslinks erst generiert und angezeigt, nachdem eine Gruppe erstellt wurde. Abbildung 1.1 stellt eine Skizze der neuen Oberfläche für die Gruppenerstellung dar.

1.1.2. Fehlermeldungen

Die Benutzeroberfläche der Clientanwendung muss in der Lage sein Fehlermeldungen anzuzeigen. Fehlermeldungen, die durch eine fehlerhafte Eingabe entstanden sind, werden dabei in rotem Text unter dem entsprechenden Eingabefeld angezeigt und unerwartete Fehler, wie etwa ein Verbindungsverlust zum Server, werden als Toast über der Oberfläche dargestellt. Eine Skizze dieser Toasts ist in Abbildung 1.2 zu sehen. Wie auch im Pflichtenheft ist diese nur eine Skizze und wird nicht unbedingt der exakten finalen Oberfläche entsprechen.

The image shows a mobile application screen titled "Create New Group". At the top left is a back arrow icon. Below the title is a text input field with the placeholder "Name" and the text "WG Ausgaben". Below the input field is a purple rounded button labeled "Create".

Abbildung 1.1.: Neue Oberfläche für die Gruppenerstellung

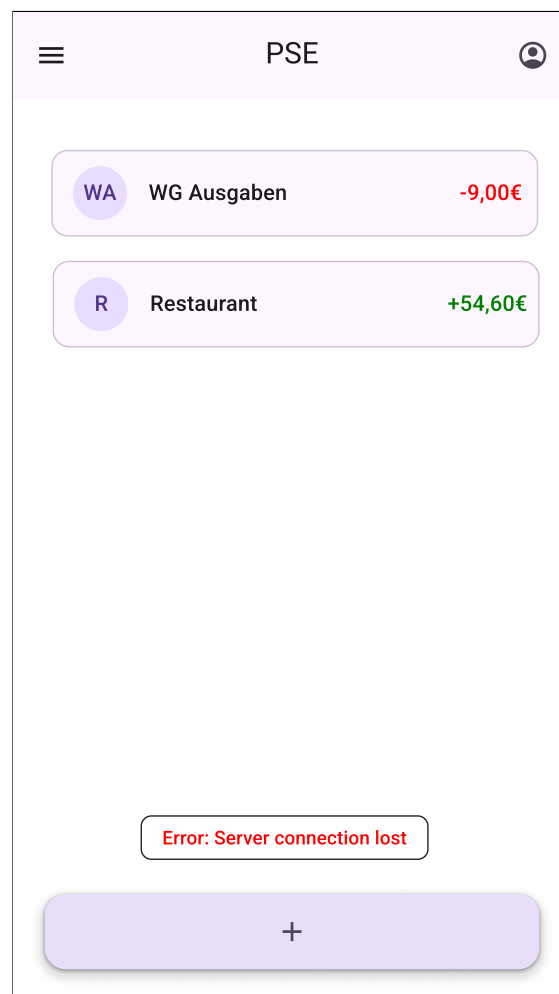


Abbildung 1.2.: Ein Toast, der eine Fehlermeldung anzeigt

1.1.3. Profilbilder von Gruppen

Die Profilbilder von Gruppen werden anhand der Initialen des Gruppennamens bestimmt.

2. Architektur

2.1. Struktureller Aufbau

2.1.1. Systemstruktur

Das *Program for Splitting Expenses* soll in drei Hauptkomponenten gegliedert werden: eine **Android App**, einen **Applikationsserver** und eine **SQL Datenbank**. Dabei werden erstere zwei in Kotlin entwickelt. Zudem findet die Authentifizierung mittels eines OpenID Connect kompatiblen **Identity Providers** statt und die App kommuniziert über eine HTTPS REST Schnittstelle mit dem Server.

2.1.1.1. Aufgaben der Systemkomponenten

Das Zusammenspiel der Systemkomponenten ist als Komponentendiagramm in Abbildung 2.1 zu erkennen. Es folgt eine Beschreibung ihrer Aufgaben.

Android App (kurz „App“ bzw. „Client“ für App-Instanzen)

Die App ist neben dem OpenID Connect (OIDC) Provider die einzige Komponente, die mit dem Endnutzer interagiert. Sie wird als apk-Datei bereitgestellt und vom Endnutzer in Android installiert. Notwendig für das Funktionieren der App ist die HTTPS Kommunikation mit dem Server, sowie Kommunikation mit dem Identity Provider. Die App fragt vom Server Teile des Systemzustands ab, auf die der Nutzer Zugriff hat, und zeigt diese an.

Applikationsserver (kurz „Server“)

Der Applikationsserver verarbeitet im wesentlichen Anfragen an die Datenbank. Dabei wird sichergestellt, dass die Anfragen erlaubt und gültig sind. Somit garantiert er die Integrität des in der Datenbank liegenden Systemzustands. Außerdem stellt er sicher, dass Nutzer nur Zugriff auf bestimmte Teile des Systemzustands erhalten.

SQL Datenbank (kurz „Datenbank“)

Die SQL Datenbank ist für die Datenhaltung im System verantwortlich und enthält den gesamten Systemzustand. Sie interagiert alleine mit dem Applikationsserver.

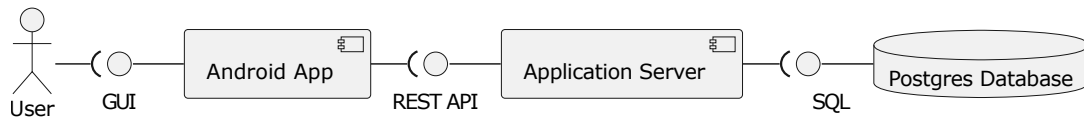


Abbildung 2.1.: Systemkomponenten

2.1.1.2. Externe Systeme

OpenID Connect Identity Provider Der Identity Provider prüft und liefert die Identitäts- und Berechtigungsnachweise des Nutzers. Beim Login liefert er zum einen die Nutzer ID, sowie die nötigen Credentials für den Zugriff auf den Nutzeraccount.

2.1.1.3. Minimale Voraussetzungen für Systemfunktion

Grundsätzlich werden für ein funktionierendes Gesamtsystem alle Subsysteme benötigt. Im Falle eines temporären Ausfalls von App oder Server genügt jedoch eine Wiederherstellung der ausgefallenen Subsysteme, um das Gesamtsystem wiederherzustellen. Kritisch ist aufgrund der enthaltenen Daten die Datenbank. Diese sollte regelmäßig gesichert werden und besonders geschützt werden.

Im Falle eines temporären Ausfalls des Identity Providers kann das System alle Aktivitäten, abgesehen von der Anmeldung, ungestört durchführen.

2.1.2. Verteilung

Das verteilte PSE-System besteht aus fünf Komponenten, wie in Abbildung 2.2 zu sehen ist. Der OpenID Connect Provider ist ein externes System und nicht Teil des Verteilungsprozesses von PSE.

Die Verteilung der restlichen Komponenten ist unterteilt in die Verteilung der *Android Phone*- und der *Server*-Komponenten.

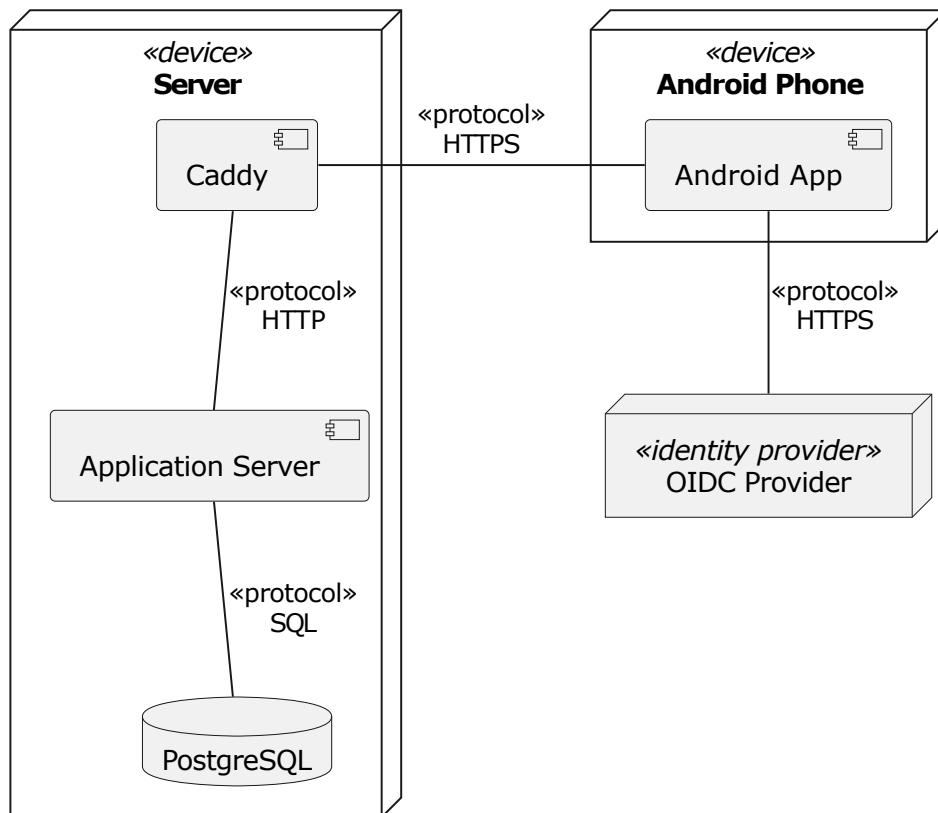


Abbildung 2.2.: Verteilungsdiagramm

2.1.2.1. Verteilung des Servers

Für den Server wird eine einzige deb-Datei bereitgestellt. Diese kann auf Debian 12 Systemen installiert werden und installiert sowohl die eigentliche Serversoftware, als auch eine PostgreSQL-Datenbank und den Webserver Caddy, der verwendet wird, um ein TLS-Zertifikat zu erhalten und HTTPS zu unterstützen. Bei der Installation des bereitgestellten Debian-Pakets wird außerdem die Domain, unter der der Server erreichbar ist, sowie die Konfigurationsparameter des OIDC Providers abgefragt. Damit wird dann die Serversoftware, Caddy und PostgreSQL entsprechend konfiguriert, sodass der Server im Anschluss direkt verwendet werden kann. Der PSE Application-Server verwendet ein systemd-Unit zur Verwaltung, sodass die von systemd bereitgestellten Werkzeuge wie `systemctl` und `journalctl` zur Verwaltung des Servers genutzt werden können.

Dieser Verteilungsansatz wurde gewählt, da die Serversoftware unter Debian 12 laufen muss und der gewählte Ansatz auf bestehende und erprobte Lösungen des Debian-

Betriebssystems zurückgreift. Auf diese Weise wird die Verteilung dahingehend vereinfacht, dass sie ähnlich zur Verteilung anderer Software unter Debian abläuft.

2.1.2.2. Distribution der App

Die App wird als einzige *Android Phone*-Komponente über ein eigenes F-Droid¹-repository bereitgestellt und kann dann über F-Droid installiert werden, nachdem der Nutzer das Repository seinem F-Droid Store hinzugefügt hat. Auf diese Weise kann der Nutzer die App über eine bekannte App Store-Anwendung installieren und der Betreiber von PSE kann die App selbst bereitstellen.

Das bereitgestellte Debian-Paket des Servers setzt die F-Droid Server Tools und ein F-Droid Repository auf, sodass im Anschluss die App in dieses Repository eingefügt werden kann. Dieser Vorgang ist unter https://f-droid.org/docs/Setup_an_F-Droid_App_Repo/ beschrieben.

2.1.2.3. Verteilung auf einer bwcloud-Instanz

In diesem Abschnitt wird die beispielhafte Verteilung auf einer bwcloud²-Instanz beschrieben. Dazu wird vorausgesetzt, dass eine bwcloud-Instanz mit Debian bookworm und root-Zugriff über das SSH-Protokoll besteht. Außerdem wird vorausgesetzt, dass diese bwcloud-Instanz über eine im öffentlichen Internet geroutete IP-Adresse global erreichbar ist und es eine Domain gibt, die auf die bwcloud-Instanz zeigt. Im folgenden Beispiel wird angenommen, dass diese Domain `pse-app.com` ist. Durch Ersetzen aller Vorkommnisse von `pse-app.com` durch eine andere Domain ist es möglich, das System unter einer anderen Domain bereitzustellen.

Weiterhin wird angenommen, dass die in Abschnitt 3.1.3 beschriebenen benötigten Konfigurationsparameter für den OIDC Provider bekannt sind.

Die Verteilung funktioniert dann wie folgt:

1. Hochladen des PSE Debian-Pakets sowie der PSE apk-Datei auf den Server: `$ scp pse.deb pse.apk root@pse-app.com:`
2. Anmelden bei der bwcloud-Instanz: `$ ssh -t root@pse-app.com:pse.deb`
3. Installieren des eben hochgeladenen Debian-Pakets: `# apt install ./pse.deb`

¹<https://f-droid.org/>

²<https://www.bw-cloud.org/>

4. Während des Installationsprozesses muss auf Nachfrage der Domainname, also `pse-app.com`, sowie die in Abschnitt 3.1.3 beschriebenen Konfigurationsparameter für den OIDC Provider angegeben werden. Aus diesen Eingaben wird dann die Konfigurationsdatei für PSE erstellt.
5. In das Verzeichnis `/etc/fdroid` wechseln: `# cd /etc/fdroid`
6. Ausführen der Schritte 4. bis 9., die unter https://f-droid.org/en/docs/Setup_an_F-Droid_App_Repo/#overview beschrieben sind.
7. Bereitstellen der hochgeladenen apk-Datei: `# fdroid deploy`

2.2. Architektur

In diesem Abschnitt wird die Architektur des Systems auf hoher Abstraktionsebene erklärt und begründet.

2.2.1. Client-Server Architektur

PSE basiert auf einer Client-Server Architektur. Der komplette Zustand des Systems, wie zum Beispiel Accounts und Buchungen, werden auf Serverseite in der Datenbank persistiert. Der Client, also die Android App, ruft dann über eine REST API die Daten ab. Im Gegensatz zu einer Peer-to-Peer Architektur ist so die Systemstruktur einfacher und resistenter gegen eventuelle Angriffe.

2.2.1.1. Kein Polling

Der Client soll ausdrücklich nicht per Polling Daten vom Server abfragen. Jede Anfrage an den Server kann also auf eine Aktion des Nutzers zurückgeführt werden und jede Aktion des Nutzers verursacht nur endlich viele Anfragen, für gewöhnlich eine. So wird kein überflüssiger Datenverkehr zwischen Server und Client verursacht.

2.2.2. Client Struktur

Der Client wird dem MVVM Muster entsprechend in die drei Schichten **View**, **Model** und **ViewModel** unterteilt. Diese Schichten sind intransparent, die **View** Komponente interagiert also beispielsweise nicht mit dem zugrundeliegenden **Model**, sondern nur mit dem **ViewModel**.

View

Dient zur Beschreibung der UI und bestimmt wie der vom **ViewModel** gelieferte UI Zustand angezeigt werden soll.

ViewModel

Beschreibt den Zustand der UI, sowie die Verarbeitung von Nutzereingaben. Dafür wird der vom Model erhaltene Modellzustand aufbereitet und Nutzereingaben auf passende Operationen im Model abgebildet.

Model

Modelliert und verwaltet die lokale Sicht auf den Serverzustands und ist für die Interaktion mit dem Server zuständig. Die Model Komponente bietet **ViewModel** eine Sicht auf die Daten des Systems, welche diese wiederum in eine für eine **View** passende Sicht übersetzen.

2.2.2.1. Schichtenarchitektur für das Model

Die **Model** Komponente selbst wird in *Facade*, *Repositories* und *Data Layer* Komponenten unterteilt.

Facade

Stellt die Schnittstelle nach außen bereit und handhabt äußere Interaktion mit der **Model** Komponente.

Repositories

Die Repositories Komponente verwaltet den eigentlichen Modellzustand und definiert die Logik nach der dieser erneuert und konsistent gehalten wird.

Data Layer

Abstrahiert und handhabt die Kommunikation mit dem Server.

2.2.3. Client Entwurfsentscheidungen

2.2.3.1. Vorteile einer Schichtenarchitektur

Die Einführung mehrerer Schichten führt zu einer größeren Entwurfskomplexität, bedeutet unserer Meinung nach in unserem Fall jedoch eine einfacher zu wartende und erweiterbarere App.

Die Trennung der UI Strukturbeschreibung von der UI Zustandslogik erlaubt einfachere Anpassungen des UI Aussehens und eine klarere und natürliche Aufgabentrennung.

Weniger offensichtlich ist die Notwendigkeit der **Model** Komponente auf Seite des Clients, da systemweit der Modellzustand gewissermaßen auf dem Server liegt. Schließlich wird der Modellzustand auf dem Client auch nur vom Serverzustand abgeleitet.

Warum das ViewModel vom Model trennen? Theoretisch könnte man in der App allein den UI Zustand verwalten und bei Bedarf diesen mit Server API Anfragen aktualisieren. Die Datenbedürfnisse der UI können jedoch nicht zwingend 1:1 in API Calls übersetzt werden. Dies soll anhand der folgenden Beispielen erläutert werden.

Benötigte Daten vs. mögliche API Calls Gegeben sei ein Screen, der die Namen der Nutzer innerhalb einer Gruppe anzeigen soll. Dann wäre es auf UI Seite am einfachsten, pro Nutzer jeweils einmal den Namen abzufragen. Auf API Seite ist es jedoch am besten, alle Namen in einer Anfrage abzufragen.

Alte Zustände Es ist nützlich auf UI Seite vor dem erhalten einer Antwort einer API Anfrage bereits einen alten Zustand anzuzeigen, da sich dieser nicht zwingend geändert hat. Die Speicherung dieses alten Zustands sollte aber keine Aufgabe der UI sein.

Um der UI eine gewisse Unabhängigkeit und konzeptuelle Trennung von der Server API zu geben, ist es deshalb sinnvoll einen zusätzlichen Modellzustand zu definieren, der als Brücke zwischen den beiden dient. Dieser Zustand wird dann von der **Model** Komponente verwaltet.

Die Vorteile eines solchen Entwurfs sind also:

- UI Entwicklung ist unabhängig von den Details der Server API
- Modellzustand kann unabhängig von der UI behandelt und beispielsweise gespeichert werden

2.2.3.2. Verwaltung des Modellzustands

Der vom **Model** verwaltete Modellzustand kann entweder In-Memory oder in einer lokalen Datenbank gehalten werden. Obwohl eine lokale Datenbank bei größeren Apps von Vorteil ist und Persistenz über App-Neustarts hinaus erlauben würde, haben wir uns entschieden, keine lokale Datenbank zu verwenden. Da die PSE App eine Internetverbindung voraussetzt (siehe Pflichtenheft Abschnitt 2.1 Betriebsbedingungen), rechtfertigt ein persistenter Modellzustand nicht die damit verbundene zusätzliche Komplexität der **Model** Komponente.

2.2.3.3. Erneuerung des Modellzustands

Der lokale Modellzustand soll stets möglichst aktuell vorliegen. Da weder Polling (siehe Abschnitt 2.2.1.1) betrieben wird, noch die API Websockets oder ähnliche Streams anbietet, bedeutet dies, dass bei bestimmten UI Übergängen, z.B. von einem Screen zum nächsten, der Zustand durch eine Serveranfrage aktualisiert werden muss. Da auf den meisten Screens aber nur ein Teil des Zustands angezeigt wird, muss auch nur ein Teil des Zustands aktualisiert werden. Das **Model** muss dem **ViewModel** also eine Schnittstelle zur partiellen Zustandsaktualisierung bieten. Dies ist die *Refresh* Schnittstelle, mit der bestimmte Teile des Modellzustands erneuert werden können.

2.2.3.4. Abfrage des Modellzustands

Für die Aktualisierung des Modellzustands gibt es zwei Möglichkeiten:

„Pull“

Das **ViewModel** liest bei Bedarf den Modellzustand.

„Push“

Das **ViewModel** wird bei Änderungen im Modellzustand benachrichtigt.

Wir haben uns für eine Kombination der beiden Optionen entschieden, da so mehr Flexibilität geboten ist und eine spätere Einführung von Websockets oder Ähnlichem erleichtert wird. Über die *Observe* Schnittstelle informiert das **ViewModel** das **Model**, wenn es einen bestimmten Teil des Modellzustands „beobachten“ möchte. Das **Model** gibt dann direkt den angefragten Zustand zurück und benachrichtigt zusätzlich das **ViewModel** später bei Änderungen. Die genaue Umsetzung dieser *Observe* Schnittstelle wird unter Abschnitt 3.3.5.1 beschrieben.

2.2.4. Server Struktur

Wir verwenden auf dem Server eine Schichtenarchitektur mit drei intransparenten Schichten. Die Schichten gliedern sich in **Request Handler** und **Request Executor** als oberste Schicht, die *Verwaltungskomponenten* als mittlere Schicht und die *Datenbankabstraktion* als unterste Schicht. Die einzelnen Komponenten der Schichten sind in Abbildung 2.7 zu sehen.

Oberste Schicht: Request Handler und Request Executor Die oberste Schicht unserer Schichtenarchitektur und somit die höchste Abstraktionsebene auf dem Server ist durch die Komponenten **Request Handler** und **Request Executor** gegeben, auf die die Bearbeitung der API Calls aufgetrennt ist. Letzterer setzt alle Anfragen, die unser Server bearbeiten können soll, als einzelne Funktionen um, während Ersterer dafür verantwortlich ist, API Calls den richtigen solcher Funktionen zuzuordnen. Bevor der **Request Handler** eine gestellte Anfrage durch den **Request Executor** ausführen lässt, übersetzt er die mit dem API Call übermittelten Daten in die internen Datenklassen des Servers und überprüft mittels der Authentifizierungs- und Autorisierungskomponente, ob die für die gestellte Anfrage erforderlichen Rechte überhaupt gegeben sind.

Die Trennung in **Request Handler** und **Request Executor** hat den Vorteil, dass man zu einem späteren Zeitpunkt manche der REST API Endpunkte beispielsweise durch Remote Procedure Calls ersetzen kann, und dafür nur den **Request Handler** anpassen muss.

Mittlere Schicht: Verwaltungskomponenten Der **Request Handler** und der **Request Executor** greifen zur konkreten Umsetzung der Server Logik auf Verwaltungskomponenten zu, die jeweils nur für Gruppen, Nutzer oder Buchungen verantwortlich sind und die konkret durchzuführenden Aktionen implementieren. Diese Verwaltungskomponenten bilden die mittlere Ebene unserer Schichtenarchitektur.

Unterste Schicht: Datenbankabstraktion durch Ktorm Die Verwaltungskomponenten verwenden Ktorm³, um mit der Datenbank zu interagieren. Die konkreten Datenbankzugriffe werden hierbei durch Ktorm abstrahiert, wodurch wir die Verwaltungskomponenten unabhängig von der Wahl unserer Datenbank implementieren können. Die durch Ktorm gegebene Abstraktion der Datenbank bildet die unterste Schicht unserer Schichtenarchitektur.

2.2.5. Server Entwurfsentscheidungen

2.2.5.1. Verwendung von Ktorm als Datenbankabstraktion

Als Datenbankabstraktion verwenden wir Ktorm. Wir haben uns hierbei aber gegen die Verwendung von ORM-Funktionalität entschieden. Denn die durch Ktorm durchführbaren, im Kotlin-Typensystem typensicheren SQL-Anfragen genügen uns bei unserem minimal gehaltenen Datenbankschema. Außerdem ist durch den Verzicht auf ORM-Funktionalität jederzeit klar, wann und wo Datenbankzugriffe stattfinden, was insbesondere beim Debugging hilfreich sein kann.

³Siehe Abschnitt 2.5.3

2.2.6. Authentifizierung

Der Login soll mithilfe von OpenID Connect (OIDC) verlaufen. Der verwendete Identity Provider (IdP), z.B. Google oder Github, wird auf dem Server konfiguriert und vom Client abgefragt.

Nach Anmeldung beim IdP wird der erhaltene ID Token⁴ einmal verwendet, um den Nutzer beim Server zu authentifizieren. Anschließend werden unsere eigens auf dem Server generierten Tokens als Berechtigungsnachweis für die neue Session verwendet.

Dieses System hat folgende Vorteile:

- Es müssen keine Passwörter (bzw. Hashes) gespeichert werden. Das gilt allgemein für Berechtigungsnachweise, die sich nicht beliebig invalidieren lassen.
- Systeme zur Änderung oder Wiederherstellung von Passwörtern sind nicht nötig. E-Mail Adressen werden ebenfalls nicht benötigt.
- Über OIDC ist es möglich, einfache Nutzerinformationen wie den Anzeigenamen vom IdP zu erhalten, sodass wir diese bei Accounterstellung übernehmen können.
- Die Ausstellung eigener Tokens erlaubt die Verwaltung von individuellen Sessions.
- Die einmalige Verwendung des ID Tokens schafft Unabhängigkeit von der Implementation des IdP.
- Eine Alternative wäre gewesen, einen Token des IdP für jede Anfrage zu verwenden und diesen immer wieder zu erneuern. Dies hätte aber dazu geführt, dass wir Sessions weder IdP-unabhängig unterscheiden, noch invalidieren könnten, da die OpenID-Spezifikation zu viel offen lässt und sich bestimmte IdPs (Google) auch an das Spezifizierte nicht immer halten.

2.2.7. Datenbank

PSE soll als Datenbank PostgreSQL in der Version 15 verwenden. Es wird auf eine relationale SQL-Datenbank gesetzt, da relationale Datenbanken einerseits weit verbreitet und so unter Entwicklern gut bekannt sind und andererseits durch Transaktionen starke Konsistenzgarantien bei parallelem Zugriff bieten. PostgreSQL wurde gewählt, da es spezifische Datentypen, wie UUID und MONEY für Daten bietet, die PSE in der Datenbank speichern muss und die Datenbank so beispielsweise bereits garantieren kann, dass gespeicherte UUIDs gültig sind. PSE verwendet Version 15 von PostgreSQL, da dies die

⁴https://openid.net/specs/openid-connect-core-1_0.html#CodeIDToken

Version ist, die auf Debian 12 bereitgestellt wird und die Serversoftware unter Debian 12 laufen muss.

2.2.8. Beispielhafte Interaktion der Architekturkomponenten

Es folgt jeweils eine beispielhafte Interaktion der in Abschnitt 2.2.2 bzw. 2.2.4 eingeführten Architekturkomponenten innerhalb des Clients bzw. Servers.

Wir verwenden hierfür ein beispielhaftes Szenario, bei dem ein Nutzer eine neue Buchung innerhalb einer Gruppe anlegen will. Hierzu hat er bereits die Parameter der neuen Buchung in die Oberfläche eingegeben und soeben den Button zum Hinzufügen der Buchung betätigt.

Die Interaktion der Architekturkomponenten auf dem Client im Falle unseres Beispiels sind in Abbildung 2.3 als Sequenzdiagramm dargestellt. Es folgt eine ausführlichere Beschreibung:

1. **View** meldet **ViewModel** welcher Button betätigt wurde
2. **ViewModel** interpretiert diese Aktion und meldet die Interpretation zusammen mit den in **ViewModel** gehaltenen Buchungsparametern dem **Model**.
3. **Model** ruft den passenden Endpunkt des Servers an
4. Server gibt die erfolgreich erstellt Buchung zurück
5. **Model** erneuert den eigenen Modellzustand mit der neuen Buchung
6. Als Konsequenz dieser Erneuerung meldet **Model** dem **ViewModel** über die Observe Schnittstelle den neuen Zustand
7. **Model** meldet im Kontext der in Schritt 2. aufgerufenen Aktion, den Erfolg
8. **ViewModel** erkennt den Erfolg und navigiert entsprechend von der Buchungserstellungssicht zur Gruppensicht
9. **View** benötigt nun den UI Zustand für die Gruppensicht
10. **ViewModel** konstruiert den passenden UI Zustand durch Anfrage des Modellzustands
11. **ViewModel** empfängt den angefragten Modellzustand
12. **View** empfängt den neuen UI Zustand

Auf dem Server läuft die Interaktion der Architekturkomponenten für unser Beispiel nach dem in Abbildung 2.4 zu erkennenden Sequenzdiagramm ab.

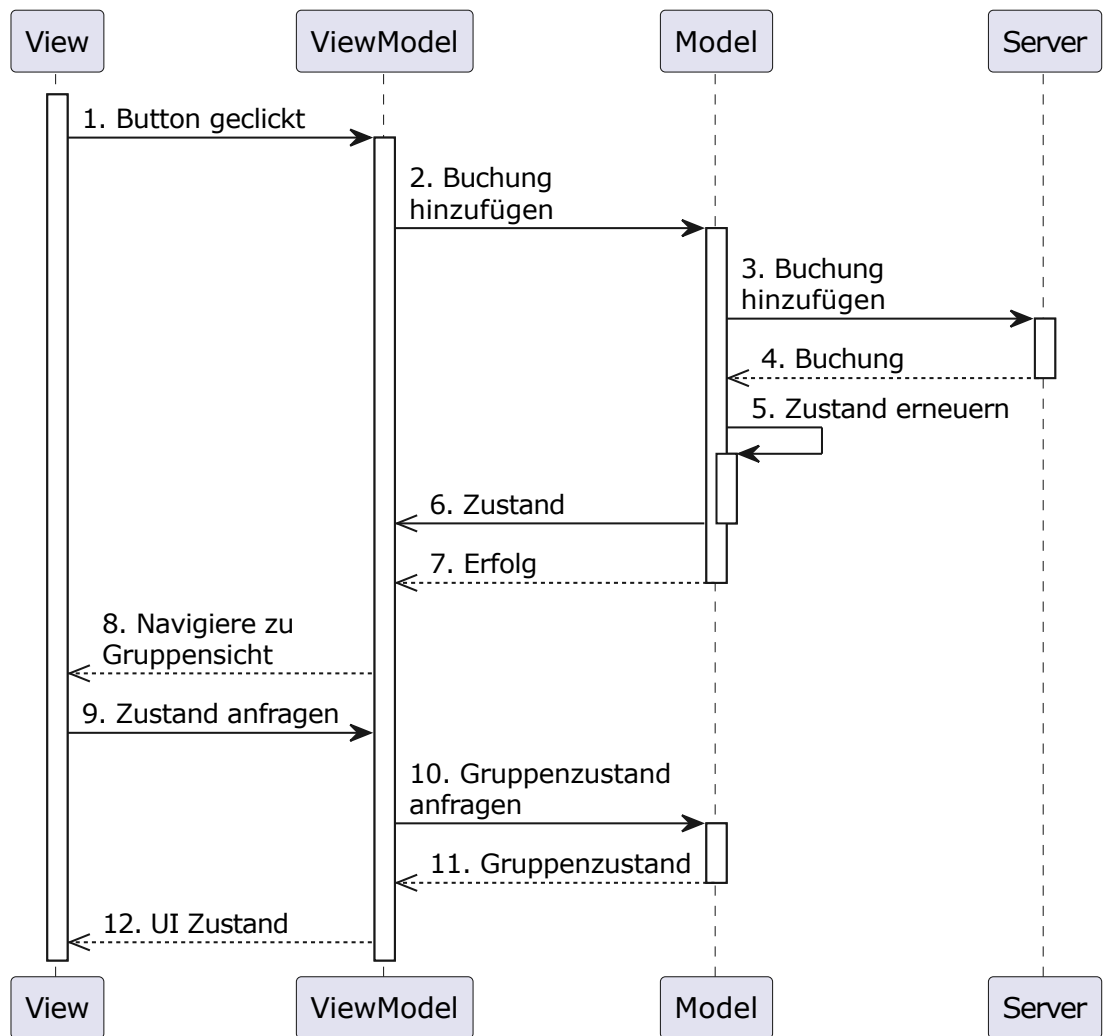


Abbildung 2.3.: Beispielhafte Interaktion der Architekturkomponenten auf dem Client bei Erstellung einer neuen Buchung

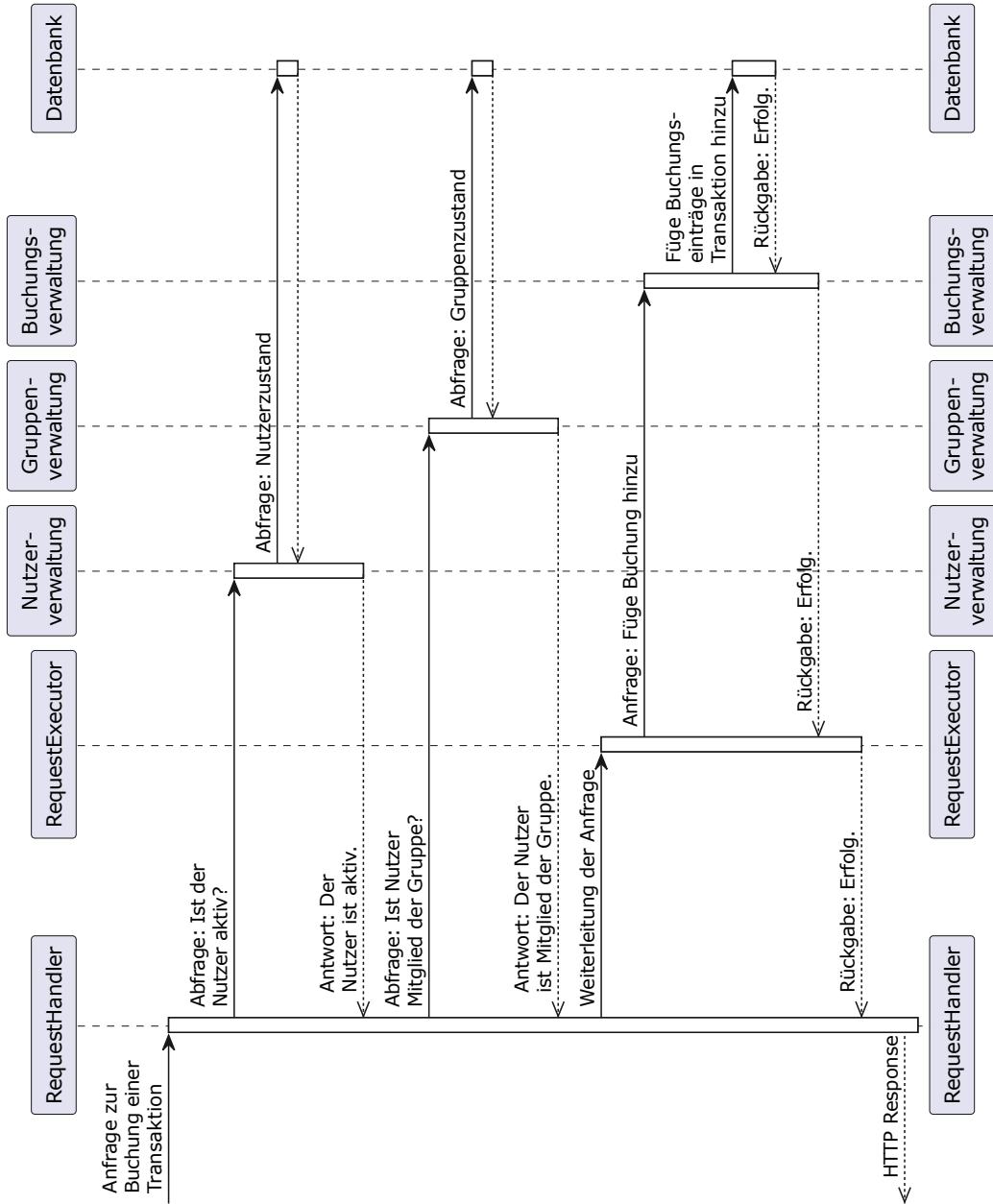


Abbildung 2.4.: Beispielhafte Interaktion der Architekturkomponenten auf dem Server bei Erstellung einer neuen Buchung

2.3. Komponentenspezifikation

2.3.1. Client

Der Client ist in fünf Komponenten aufgeteilt (siehe Abbildung 2.5).

Die **View**, **ViewModel** und **Model** Komponenten funktionieren wie in Abschnitt 2.2.2 beschrieben. Zusätzlich wird eine **Authentication** Komponente benötigt, die mit dem IdP interagiert und den ID Token erhält. Außerdem werden Einstellungen und sonstige zu persistierende Daten mit der **Persistence** Komponente persistiert.

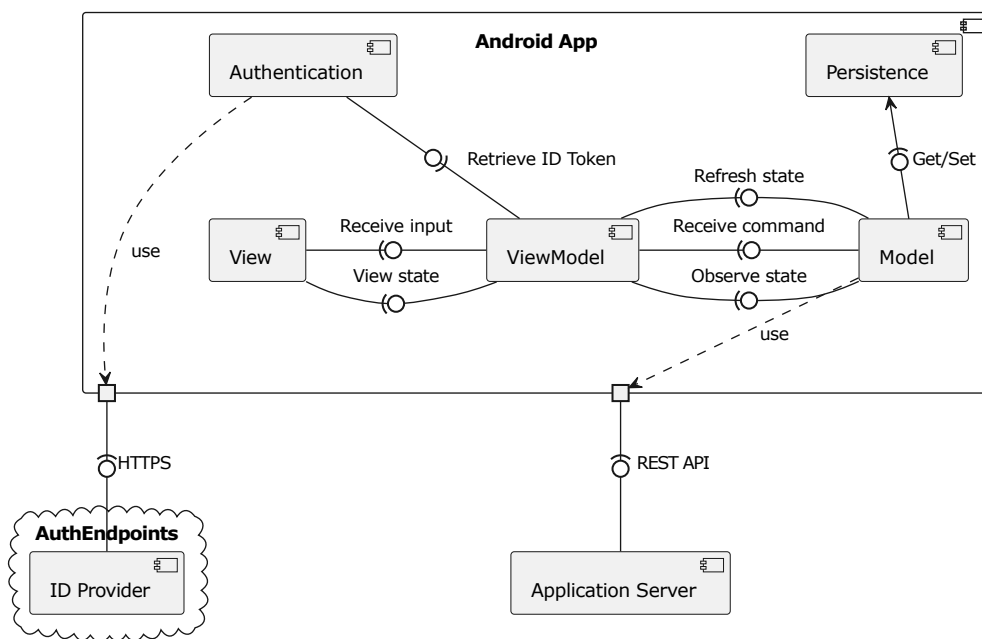


Abbildung 2.5.: Client Komponenten

Die **Model** Komponente selbst ist in drei Komponenten aufgeteilt: Eine **Facade** Komponente, die als Brücke zwischen **Model** und außen dient, eine **Repositories** Komponente, welche den Modellzustand verwaltet und die **Data Layer** Komponente, welche für die Interaktion mit dem Server verantwortlich ist. Siehe dazu Abbildung 2.6.

2.3.2. Server

Der Server ist in sechs Komponenten aufgeteilt (siehe Abbildung 2.7).

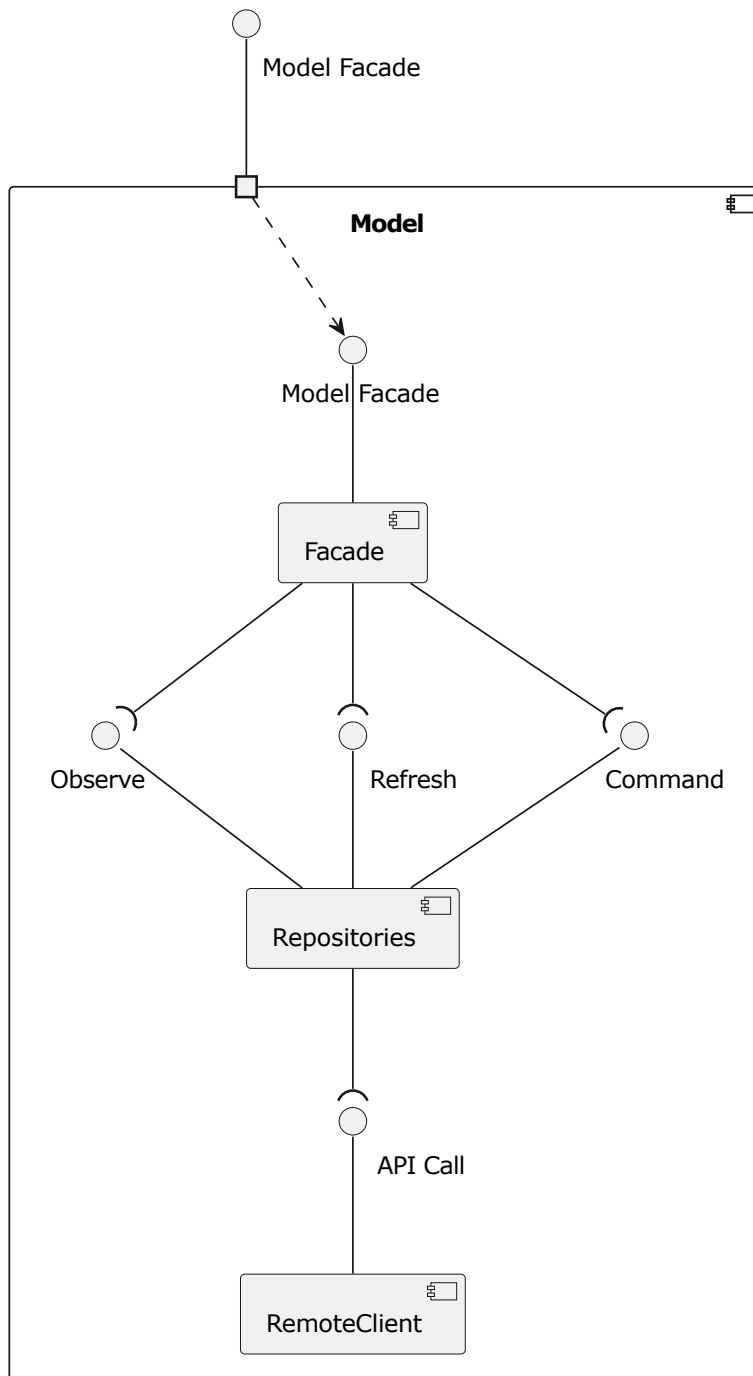


Abbildung 2.6.: Model Komponenten

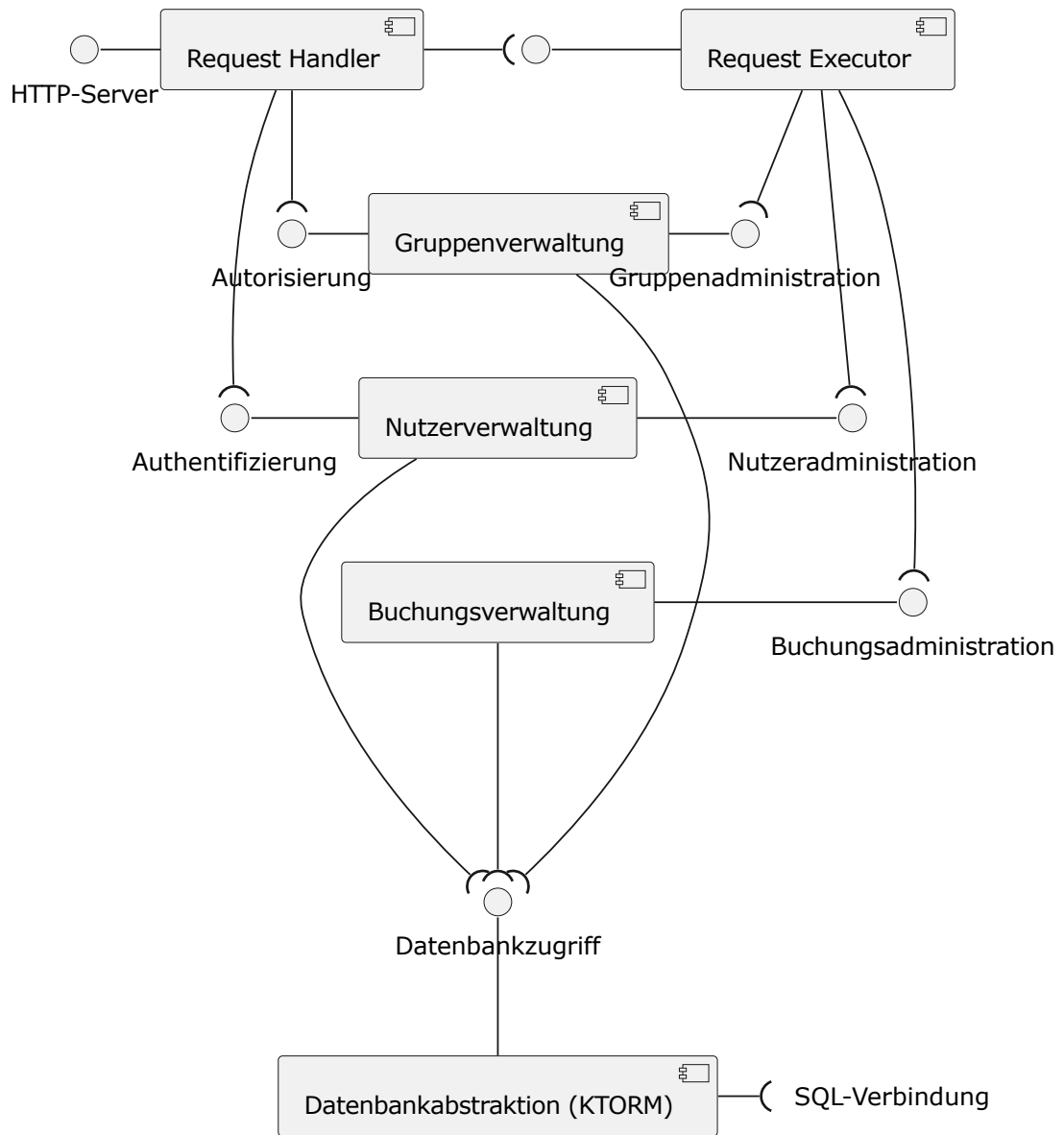


Abbildung 2.7.: Server Komponenten

Der **Request Handler** ist dafür zuständig, Anfragen über die REST-API entgegenzunehmen und die empfangenen Daten in die internen Datenklassen des Servers umzuwandeln. Außerdem überprüft der **Request Handler** mithilfe der entsprechenden Schnittstellen der *Nutzer-* und *Gruppenverwaltung*, ob der anfragende Nutzer dazu berechtigt ist, die gestellte Aktion durchzuführen. Letztlich ausgeführt werden Anfragen vom **Request Executor**. Dieser nimmt Anfragen vom **Request Handler** entgegen und führt diese aus. Dazu greift er auf *Nutzer-*, *Gruppen-* und *Buchungsverwaltung* zu, die jeweils Nutzer, Gruppen bzw. Buchungen verwalten. Diese drei Komponenten haben über Ktorm Zugriff auf die Datenbank. Wurde eine Anfrage ausgeführt, gibt der **Request Executor** das Ergebnis in internen Datenklassen zurück, der **Request Handler** wandelt dieses in API Antworten um.

Der Server wird über eine Main-Klasse gestartet, welche zunächst eine Konfigurationsdatei einliest und dann die einzelnen Komponenten erstellt, wobei die benötigten Schnittstellen mittels Dependency Injection an die Komponenten übergeben werden.

2.4. Schnittstellenspezifikation

2.4.1. SQL Schema

Abbildung 2.8 zeigt das SQL Schema, welches von PSE verwendet wird. Die Abbildung befindet sich auch im Anhang als `schema.pdf`.

Das Schema speichert Nutzer, Gruppen und Transaktionen in den Tabellen `users`, `groups` und `transactions`. `groups` und `transactions` enthalten jeweils direkt die in $\langle RM16 \rangle$, $\langle RM19 \rangle$, $\langle RM25 \rangle$, $\langle RM26 \rangle$ festgelegten, zu speichernden Daten. Im Fall von Benutzern ist dies nicht möglich, da Accounts deaktiviert werden können, ohne dass sich die Transaktionen in Gruppen ändern. Da die `transactions`-Tabelle direkt `users` referenziert, können Einträge in der `users`-Tabelle beim deaktivieren von Accounts nicht gelöscht werden. Aktive Nutzer haben deshalb einen weiteren Eintrag in der Tabelle `active_users`, welcher die Daten enthält, die nur für aktive Nutzer gespeichert werden müssen.

Daneben gibt es noch zwei Verbindungstabellen um $n:m$ -Beziehungen abzubilden: `membership` modelliert die Gruppenmitgliedschaften und `balance_changes` die Saldoänderungen der Nutzer innerhalb von Transaktionen. Außerdem werden die persistierten Refresh Tokens zu den aktiven Sitzungen angemeldeter Nutzer gehasht in `refresh_tokens` gespeichert.

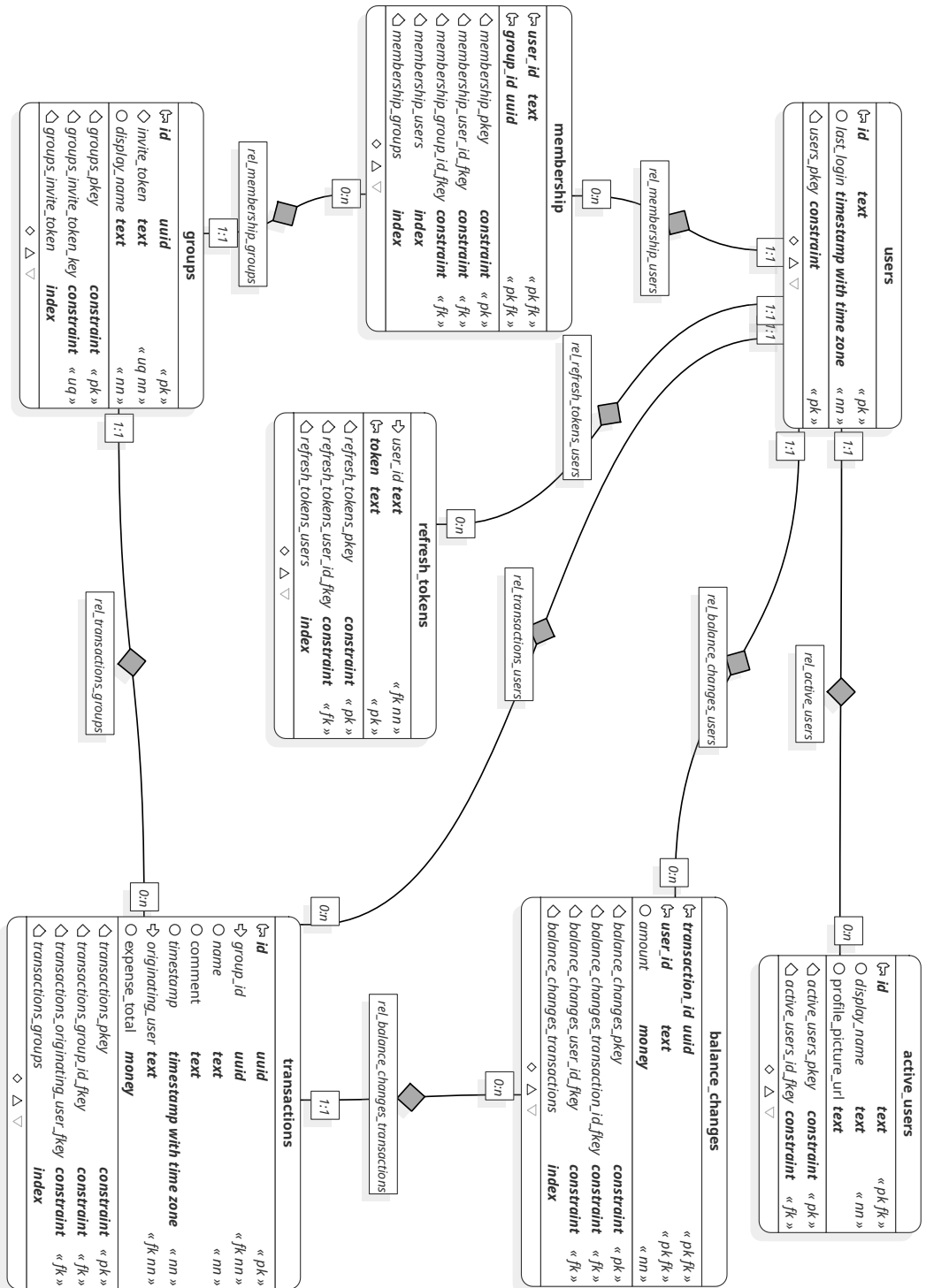


Abbildung 2.8.: Datenbankschema

Weiterhin enthält das Schema einige Indizes, die es erlauben, effizient nach Spalten in Tabellen zu suchen, die nicht Teil des primären Schlüssels sind. Die konkreten SQL Befehle zum Anlegen des Schemas finden sich in Anhang A.1.

2.4.2. REST API

Wir verwenden zur Kommunikation zwischen Client und Server eine REST API. Clients können so jederzeit HTTP Requests⁵ an den zentralen Server schicken, der ihnen in einer HTTP Response antwortet. Benötigt ein Client Daten vom Server, so fragt er diese mittels eines solchen Requests ab. Möchte ein Client irgendeine Aktion durchführen, so schickt er einen entsprechenden Request an den Server.

Dies ist mittels einer REST API einfach durch die verschiedenen standardisierten HTTP-Methoden, wie GET, POST oder DELETE umsetzbar.

Die API haben wir gemäß der OpenAPI-Spezifikation⁶ beschrieben. Für bessere Lesbarkeit kann diese Spezifikation in einem Editor, wie z.B. editor.swagger.io, geöffnet werden. Eine Visualisierung dieser Spezifikation findet sich aber auch unter Anhang A.2.

Für die Beschreibung der Request und Response Daten nutzen wir Data Transfer Objects (DTOs), die in Abschnitt 3.1.2 beschrieben werden. In der OpenAPI-Spezifikation ist außerdem hinterlegt, nach welchem JSON Schema die DTOs serialisiert werden, obwohl dies letztendlich von der Serialisierungsbibliothek abstrahiert wird.

2.4.2.1. API Entwurfsentscheidungen

Bulk Requests Wir verwenden im Allgemeinen Bulk Requests, um Informationen abzufragen, da wir so die Anzahl an benötigten Requests deutlich verringern können. So können etwa die Daten von mehreren Nutzern gleichzeitig in einem Request abgefragt werden. Um Informationen hingegen zu ändern, verwenden wir für jede solche änderbare Information eine eigene Schnittstelle, da so die Atomarität von Änderungen einfach zu garantieren ist.

GET für Datenabfragen Für das Abfragen von Daten sind GET Requests vorgesehen, welche teilweise auch einen Request Body enthalten. Der Standard sieht keinen Body für GET vor und lässt offen, wie ein HTTP Server diesen behandelt. Diese Abweichung ist allerdings weitverbreitet und lässt sich auch bei großen Projekten wie z.B. Elasticsearch⁷ beobachten.

⁵Diese sind im Deployment zwischen Client und Reverse Proxy per TLS verschlüsselt

⁶Siehe Datei `api.yaml`

⁷<https://www.elastic.co/elasticsearch>

Vorteile einer HTTP basierten API Da unsere Schnittstelle nur auf dem HTTP-Protokoll basiert, ist sie plattformunabhängig und ermöglicht in der Zukunft verschiedene Client Implementierungen. Außerdem wird der Datenaustausch zwischen Server und Clients durch DTOs und automatischer Serialisierung zu JSON durch `kotlinx.serialization` stark vereinfacht und standardisiert. Die verwendeten DTOs sind in Abschnitt 3.1.2 zu finden.

Sicherung der Endpunkte Der Erfolg der Anfrage hängt bei den meisten hier beschriebenen Endpunkten auch vom Authentifizierungs- und Autorisierungsstand des Clients ab. Ein unauthentifizierter Client kann zum Beispiel keine Gruppendaten lesen und auch ein angemeldeter Nutzer kann nur die Daten der Gruppen in denen er Mitglied ist, lesen.

Versionierung Um spätere Erweiterungen und Änderungen der API zu erleichtern, haben wir uns entschieden, diese zu versionieren, beginnend bei Version v1. Alle Endpunkte der initialen Version besitzen daher den Präfix `/v1`.

2.4.2.2. API Endpunkte

Es folgt eine Beschreibung des Verwendungszwecks eines jeden Endpoints der Version v1 der REST API. Der Versionspräfix wird hier der Lesbarkeit halber ausgelassen. Eine technische Spezifikation der Schnittstelle, die Aufschluss über die genaue Verwendungsart liefert, wird in Anhang A.2 beschrieben.

/login

Diese Route verwendet der Client für die Authentifizierung beim Server. Der genaue Ablauf ist in Abschnitt 3.1.3.3 geschildert.

Umgesetzte Funktionen: `<F1>`

/settings

Diese Route verwendet der Client, um bestimmte Einstellungen des Servers, wie die für OpenID Connect benötigten Informationen (siehe Abschnitt 3.1.3.1) oder die verwendete Währung abzufragen.

Umgesetzte Funktionen: `<F1>`

/groups

Diese Route verwendet der Client, um Informationen über (mehrere) Gruppen abzufragen und neue Gruppen zu erstellen.

Umgesetzte Funktionen: `<F5>`, `<F6>`

/groups/{groupId}/displayName

Diese Route verwendet der Client, um den Anzeigenamen einer Gruppe zu ändern.

Umgesetzte Funktionen: <F9>

/groups/{groupId}/regenerateInviteLink

Diese Route verwendet der Client, um den Einladungslink einer Gruppe neu zu generieren.

Umgesetzte Funktionen: <F7>

/groups/{groupId}/kick/{userId}

Diese Route verwendet der Client, um einen Nutzer aus einer Gruppe zu entfernen.

Umgesetzte Funktionen: <F10>

/transactions

Diese Route verwendet der Client, um alle Transaktionen aus (mehreren) Gruppen abzufragen, sowie um (mehrere) Transaktionen in (mehreren) Gruppen zu verbuchen.

Umgesetzte Funktionen: <F11>, <F12>

/balances

Diese Route verwendet der Client, um die Saldi (mehrerer) Nutzer bezüglich allen gemeinsamen Gruppen bzw. allen Nutzern aus (mehreren) Gruppen abzufragen.

Umgesetzte Funktionen: <F13>

/me

Diese Route verwendet der Client, um Informationen über den angemeldeten Nutzer abzufragen, sowie den Account des angemeldeten Nutzers zu deaktivieren.

Umgesetzte Funktionen: <F4>

/me/description

Diese Route verwendet der Client, um den Anzeigenamen des angemeldeten Nutzers zu ändern.

Umgesetzte Funktionen: <F3>

/usersInCommonGroups

Diese Route verwendet der Client, um alle Nutzer herauszufinden, mit denen der angemeldete Nutzer eine Gruppe gemeinsam hat.

/users

Diese Route verwendet der Client, um Informationen über (mehrere) Nutzer abzufragen.

`/join/{inviteToken}`

Diese Route verwendet der Client, um den angemeldeten Nutzer in eine Gruppe hinzuzufügen. Außerdem finden Nutzer, die die App noch nicht installiert haben, unter dieser Route eine Webseite vor, die Anweisungen zeigt, wie die App zu installieren ist und man anschließend der gewollten Gruppe beitreten kann.

Umgesetzte Funktionen: `<F8>`

2.4.3. UI Framework

Für die graphische Benutzeroberfläche soll das Jetpack Compose Framework⁸ verwendet werden. Mit diesem werden die bereits im Pflichtenheft skizzierten Oberflächen umgesetzt.

2.4.4. Client-interne Schnittstellen

2.4.4.1. View–ViewModel Schnittstelle

ViewModel bietet **View** eine *View State* Schnittstelle um **View** Änderungen im UI Zustand mitzuteilen. Für das Weitermelden von Nutzereingaben bietet **ViewModel** die *Receive Input* Schnittstelle.

Die genaue Umsetzung dieser Schnittstellen mit Jetpack Compose wird in Abschnitt 3.3.1.2 erläutert.

2.4.4.2. ViewModel–Model Schnittstelle

Observe Das **Model** bietet nach außen eine *Observe* Schnittstelle, mit der der Modellzustand beobachtet werden kann. Der Beobachter wird bei Änderung des beobachteten Zustands benachrichtigt.

Refresh Um die über *Observe* beobachteten Daten aktuell zu halten, bietet das **Model** die *Refresh* Schnittstelle. Mit dieser kann ein spezifischer Teil des lokalen Zustands durch Anfrage an den Server aktualisiert werden.

Command Über diese Schnittstelle werden zustandsändernde Befehle an das **Model** übermittelt. Eine Namensänderung `<F3>` stellt beispielsweise einen solchen Befehl dar.

⁸Siehe Abschnitt 2.5.1

2.4.5. Server-interne Schnittstellen

Im Folgenden wird die Funktionalität der einzelnen Schnittstellen zwischen den Server Komponenten kurz beschrieben.

Detailliertere Beschreibungen zu den Schnittstellen finden sich in Abschnitt 3.2.2.

Authentifizierung Die Authentifizierungsschnittstelle prüft, ob eine Anfrage tatsächlich vom entsprechenden Nutzer stammt und kann zur Verwaltung von Refresh Tokens verwendet werden.

Autorisierung Die Autorisierungsschnittstelle prüft, ob ein Nutzer die Berechtigung hat, eine bestimmte Aktion durchzuführen. Das bedeutet insbesondere zu überprüfen, ob ein Nutzer Mitglied einer Gruppe ist, da innerhalb von Gruppen keine Berechtigungsabstufungen vorgesehen sind.

Nutzeradministration Die Nutzeradministration bietet die Möglichkeit, Informationen über Nutzer abzurufen, zu verändern und Nutzer zu deaktivieren.

Gruppenadministration Die Gruppenadministration bietet die Möglichkeit, Gruppen zu erstellen, Mitglieder zu Gruppen hinzuzufügen und zu entfernen und Eigenschaften von Gruppen zu ändern.

Buchungsadministration Die Buchungsadministration bietet die Möglichkeit, Buchungen zu Gruppen hinzuzufügen und bestehende Buchungen abzurufen.

Datenbankzugriff Der Datenbankzugriff erfolgt direkt über Ktorm.

2.5. Externe Bibliotheken

2.5.1. Jetpack Compose

Android bietet für die Programmierung der Benutzeroberfläche zwei verbreitete Bibliotheken: *Jetpack Compose*⁹ und *Android Views*. Während man in *Android Views* die UI in XML beschreibt, verwendet man in *Compose* sogenannte „Composable“ Funktionen, um die UI direkt in Kotlin deklarativ zu beschreiben. *Jetpack Compose* ist die von Android empfohlene Lösung und bietet gegenüber *Views* (neben der moderneren Programmierschnittstelle) entscheidende Vorteile. Zum Beispiel ist die Synchronisierung der angezeigten UI mit Programmdaten in *Compose* besonders leicht.

⁹<https://developer.android.com/compose>

2.5.2. Ktor

Für die HTTPS Kommunikation zwischen Client und Server verwenden wir die weit verbreitete *Ktor*¹⁰ Bibliothek. *Ktor* ist eng mit Kotlin integriert und unterstützt somit direkt asynchrone Programmierung mit Kotlin Coroutines. Ein weiterer Vorteil sind dafür verfügbare Plugins, die das Erstellen von Sessions im Server erleichtern.

2.5.3. Ktorm

Wir verwenden *Ktorm*¹¹ für den Zugriff auf die Datenbank. *Ktorm* bietet eine Domain Specific Language, die es erlaubt, das Datenbankschema in Kotlin zu definieren und im Kotlin-Typensystem typensichere SQL Anfragen an die Datenbank zu senden. Die von *Ktorm* angebotene ORM-Funktionalität verwenden wir aus in Abschnitt 2.2.5.1 genannten Gründen nicht.

2.5.4. AppAuth

Für die Authentifizierung mit OIDC verwenden wir *AppAuth-Android*¹². Die Verwendung dieser Library ist Standard im Android Ecosystem und wird von den meisten OIDC Providern empfohlen.

Sie übernimmt große Teile der Logik für die Öffnung des WebViews und der Auslesung des Authentifizierungsergebnisses aus diesem. Die Rolle der Library im Authentifizierungsablauf wird in Abschnitt 3.1.3 genauer beschrieben.

2.5.5. kotlinx.serialization

Für die JSON Serialisierung von per HTTP verschickten Objekten verwenden wir die Bibliothek *kotlinx.serialization*¹³. Diese ist die von empfohlene Android Serialisierungsbibliothek und wird zusätzlich von Teilen von Jetpack Compose, sowie anderen Libraries benötigt.

¹⁰<https://ktor.io/>

¹¹<https://www.ktorm.org>

¹²<https://github.com/openid/AppAuth-Android>

¹³<https://github.com/Kotlin/kotlinx.serialization>

2.5.6. Jetpack DataStore

Für das Persistieren von Key-Value Paaren auf dem Android Gerät des Nutzers nutzen wir *Jetpack DataStore*¹⁴.

¹⁴<https://developer.android.com/jetpack/androidx/releases/datastore>

3. Feinentwurf

Im Feinentwurf beschreiben wir im Detail, wie die im Kapitel 2 beschriebene Architektur in Kotlin umgesetzt werden soll.

3.1. Systemweiter Entwurf

3.1.1. Modellierung von Geldbeträgen

Jegliche Geldbeträge werden im Server und im Client entsprechend $\langle Q7 \rangle$ als `java.math.BigDecimal` modelliert. So werden Abweichungen verhindert, die es beispielsweise bei der Verwendung von Floating Point Arithmetik gäbe.

3.1.2. Data Transfer Objects (DTOs)

Abbildung 3.1 zeigt die DTO Klassen.

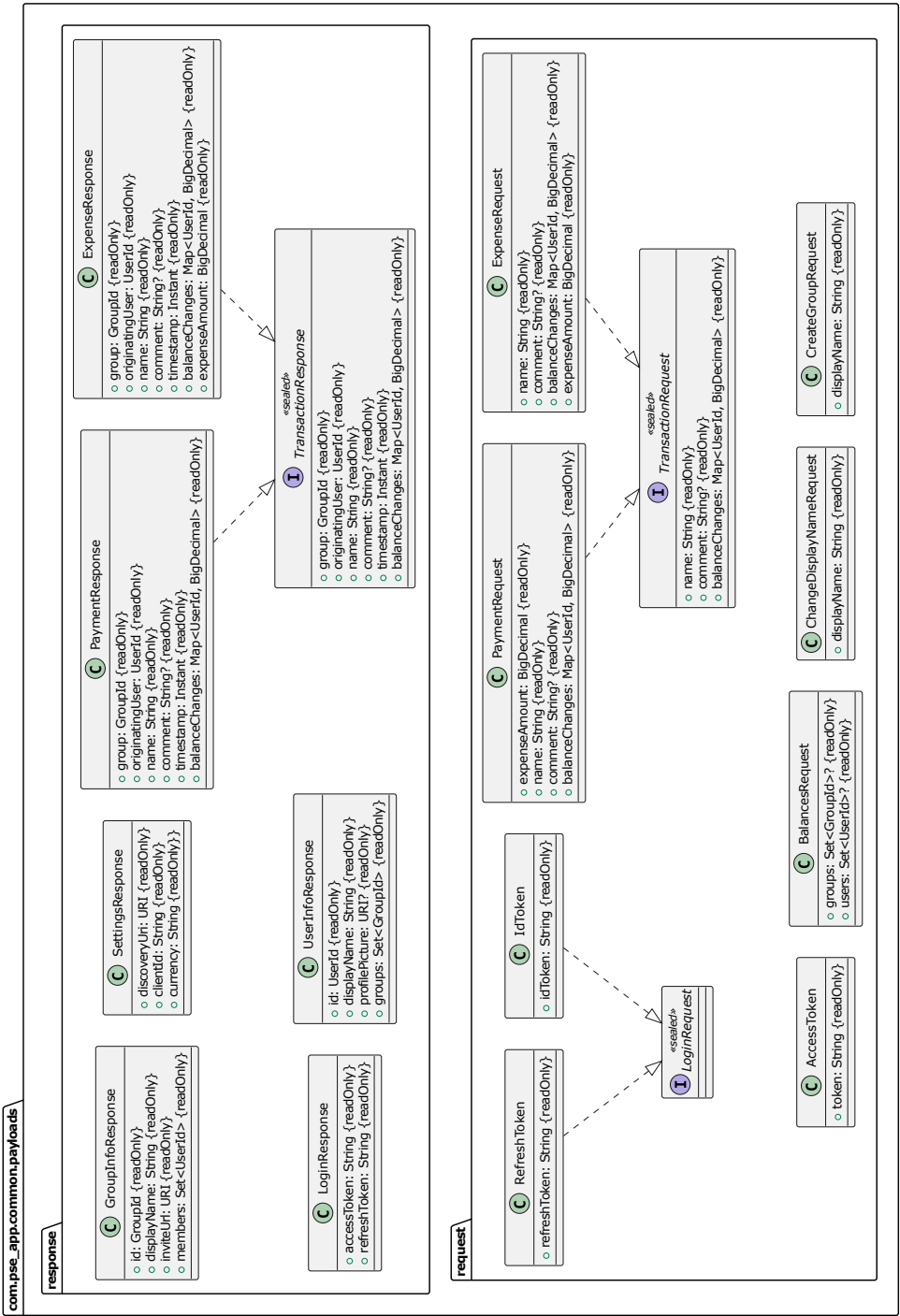
DTOs erleichtern die einheitliche Kommunikation zwischen Client und Server. Mittels der `kotlinx.serialization` Library können sie ohne weiteren Aufwand auf Server oder Client serialisiert und dann als JSON im HTTP Body geschickt werden. Der Empfänger kann sie dann mit der gleichen Library einfach deserialisieren.

Für die genaue Verwendung der DTOs siehe die API Spezifikation in Anhang A.2.

3.1.3. Authentifizierungsablauf

3.1.3.1. Provider Informationen

Die für die Authentifizierung mit dem IdP nötigen Informationen werden vom Server über den Endpunkt `/v1/settings` bereitgestellt. Relevant für die Authentifizierung sind:



Client ID

Die OIDC Client ID des Systems wird bei jeder Anfrage an den IdP benötigt und identifiziert das System. Der Server darf nur Tokens des IdP akzeptieren, wenn sie die Client ID des System als Audience Claim enthalten.

Discovery URI

Die OIDC Discovery URI¹ ist ein durch den IdP bereitgestellter Endpunkt. Er liefert alle Informationen, die der Client benötigt, um sich mit dem IdP zu authentifizieren, darunter insbesondere die Adressen anderer OIDC Endpunkte.

3.1.3.2. OIDC

Die Anmeldung beim IdP findet bis auf Anfrage und Rückgabe der Provider Informationen auf Clientseite statt. Der verwendete Prozess ist der Authorization Code Flow with Proof Key for Code Exchange² (PKCE) und ist der empfohlene Flow für native Apps. Die Einzelheiten des Ablaufs werden durch die AppAuth-Android Library übernommen und korrekt implementiert, sodass unser Code lediglich die einzelnen Schritte einleiten muss.

Zu Beginn des Logins öffnet die App mithilfe von AppAuth in einem WebView eine Autorisierungsanfrage an den IdP. Intern generiert AppAuth hierbei eine Challenge mit assoziiertem Code Verifier und sendet die Challenge mit.

Nachdem der IdP die in seinen Augen nötigen Schritte zur Authentifizierung des Nutzers durchgeführt hat, leitet er an eine URI mit vom Client spezifizierten Schema weiter. Dies wird durch die App (mithilfe von AppAuth) abgefangen und aus der URI ein Authorization Code ausgelesen. Der Authorization Code wird dann in einem Token Exchange Request mit Code Verifier gegen einen ID Token umgetauscht.

Siehe dazu auch Abbildung 3.2.

3.1.3.3. Beginn einer Session

Nach Erhalt des ID Tokens sendet der Client diesen an den Login Endpunkt des Servers. Im Gegenzug stellt der Server einen Access und einen Refresh Token aus (nicht zu verwechseln mit OAuth Access und Refresh Tokens).

¹https://openid.net/specs/openid-connect-discovery-1_0.html

²<https://auth0.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow-with-pkce>

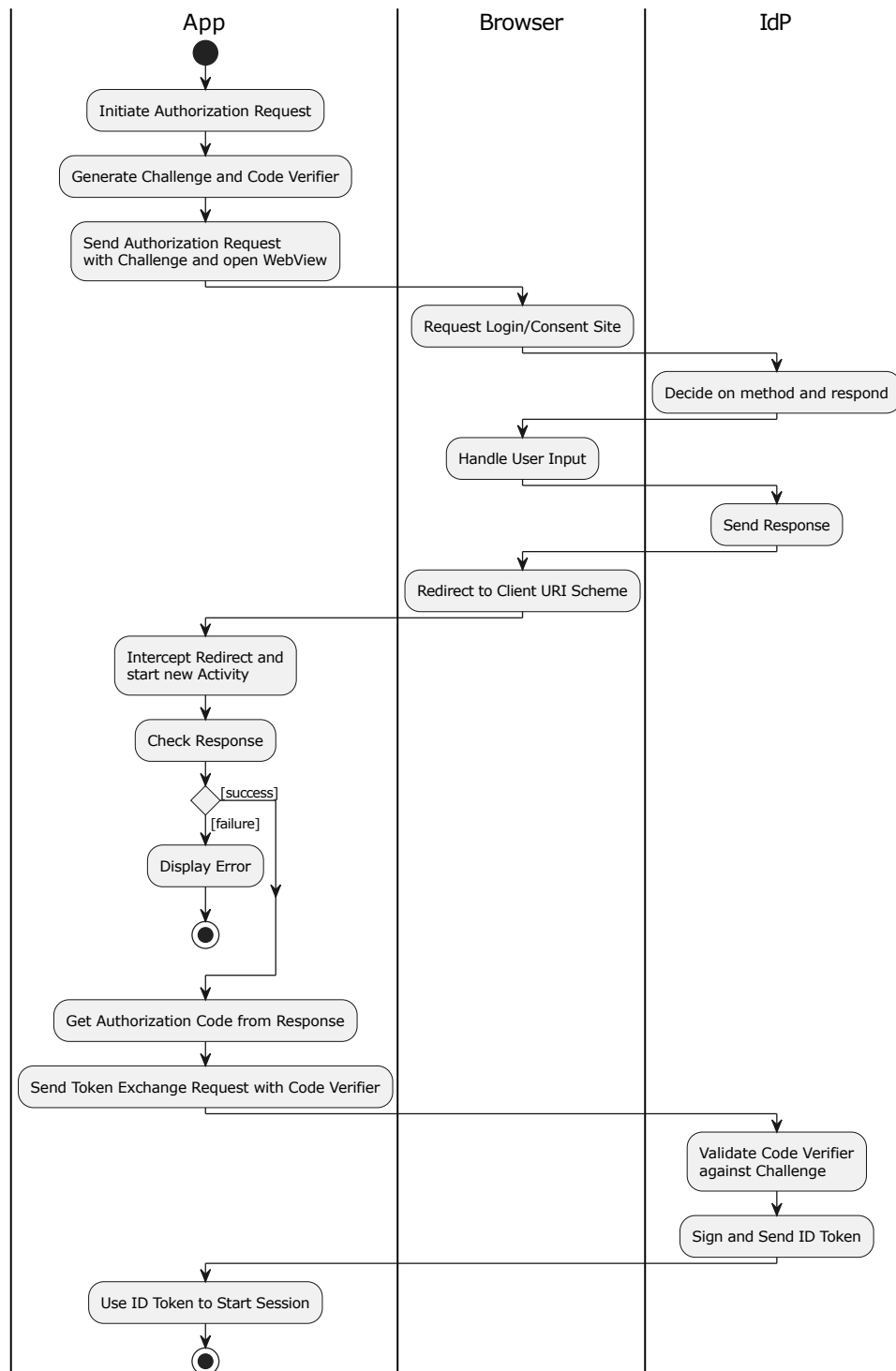


Abbildung 3.2.: Auth Code Flow mit PKCE

Der Access Token ist ein kurzlebiger, signierter Token im JWT Format (JSON Web Token), der Informationen wie Nutzer ID und Gültigkeitsdauer direkt enthält. Der Refresh Token ist ein langlebiger, beliebiger Token ohne Format, dessen Hash in der Datenbank gespeichert wird.

Siehe dazu auch Abbildung 3.3.

3.1.3.4. Verlauf einer Session

Der Access Token wird in regulären Anfragen, d.h. solchen der Gruppen- und Nutzerverwaltung, gesendet und dient der Authentifizierung des Nutzers durch Validierung der Session.

Wenn der Access Token ausläuft, muss ein neuer angefragt werden. Dazu sendet der Client seinen Refresh Token an den Login Endpunkt des Servers. Der Server validiert den Refresh Token über die Datenbank und stellt einen neuen Access und Refresh Token aus, wobei letzterer den alten ersetzt.

Siehe dazu auch Abbildung 3.4.

3.1.3.5. Ende einer Session

Bei Abmeldung wird die aktuelle Session auf Serverseite invalidiert (durch Löschung des Refresh Token Hashs aus der Datenbank) und auf der Clientseite werden alle Tokens vergessen.

Bei Accountdeaktivierung ist dies genauso, jedoch werden auf Serverseite alle Sessions invalidiert. Der genauere Verlauf der Deaktivierung ist in Abbildung 3.5 dokumentiert.

3.2. Server Entwurf

Im Folgenden wird der Server im Detail beschrieben. Dabei werden zunächst die Schnittstellen zwischen den in Abschnitt 2.3.2 vorgestellten Komponenten beschrieben. Anschließend wird auf die Komponenten jeweils einzeln eingegangen.

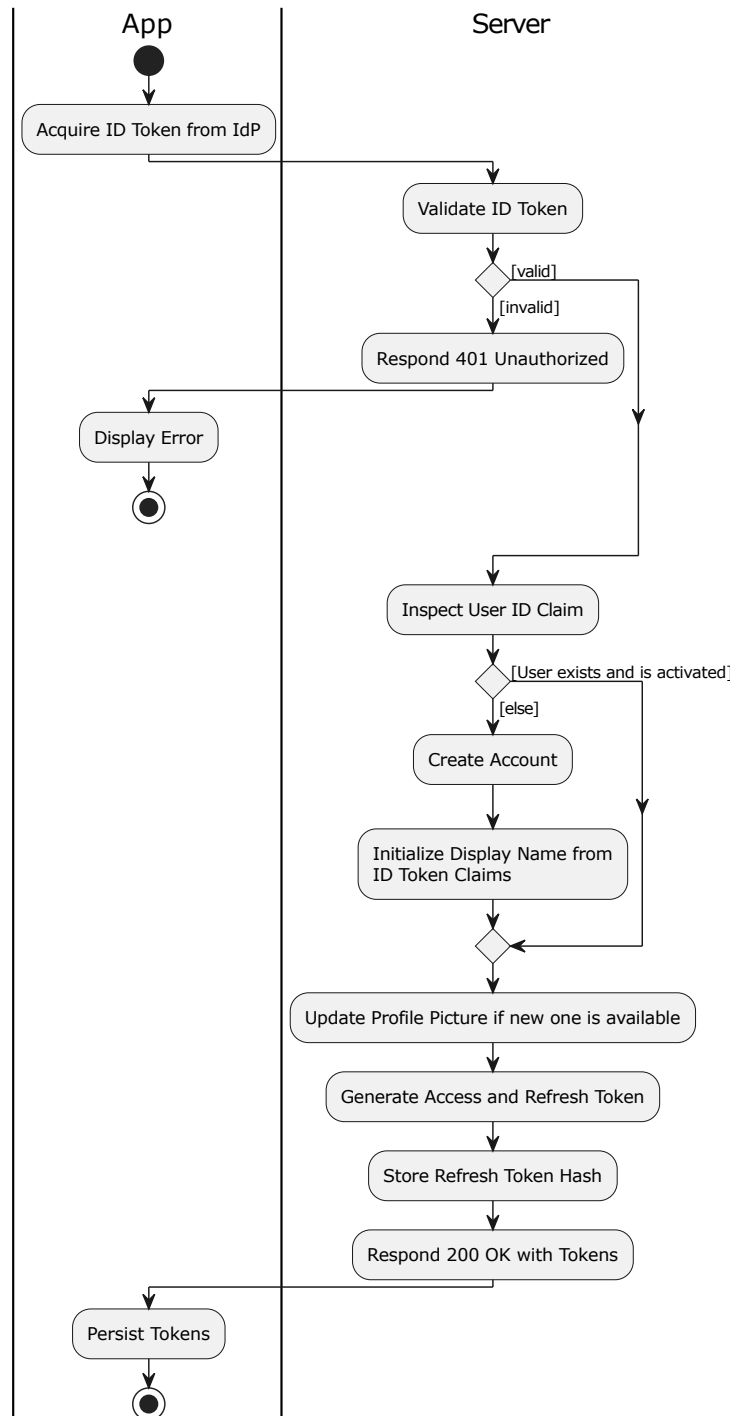


Abbildung 3.3.: Beginn einer Session

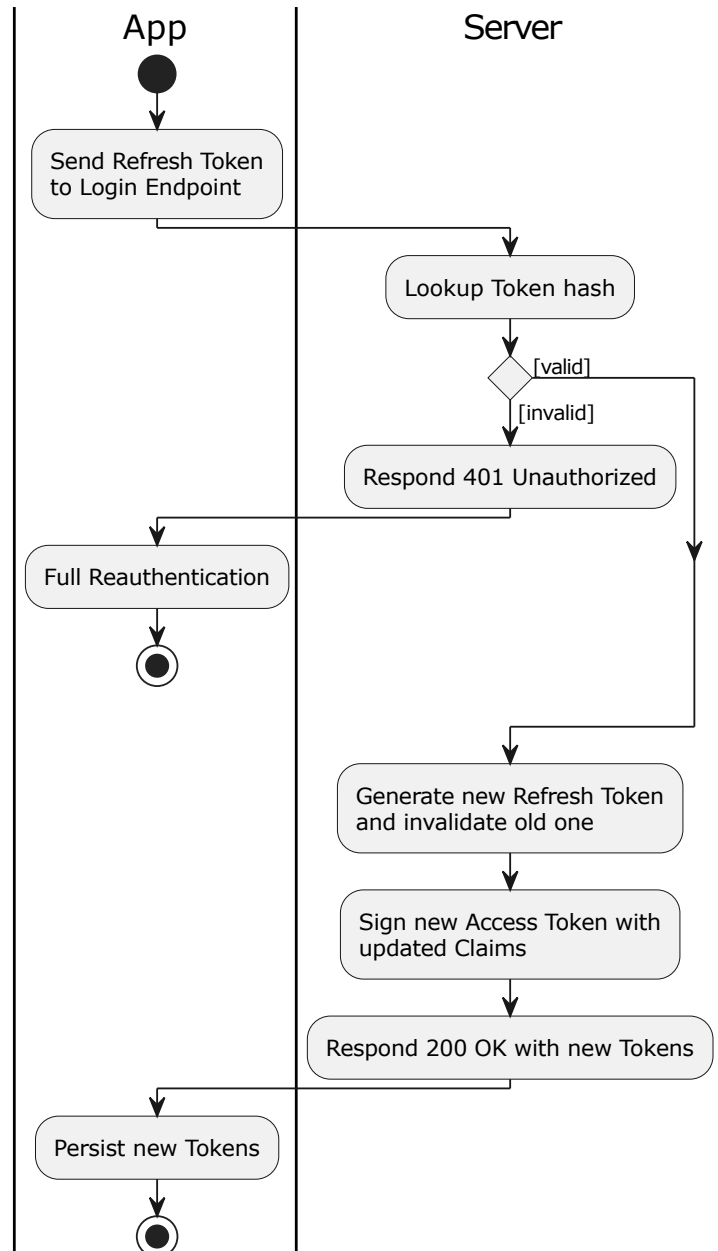


Abbildung 3.4.: Token Refresh



3.2.1. Allgemeine Datentypen

Abbildung 3.14 zeigt allgemeine Datenklassen, die überall auf dem Server verwendet werden. Diese sind unveränderlich (*immutable*).

Zunächst einmal gibt es die Klassen `UserId` und `GroupId`, die nur Wrapper um die tatsächlichen Bezeichner darstellen. Auf diese Weise ist es allerdings möglich, durch das Typsystem auszudrücken, dass sich bei einer Zeichenkette tatsächlich um eine Nutzer bzw. Gruppen ID handelt.

Die Klassen `UserInfo` und `GroupInfo` enthalten dann die weiteren Daten, die zu Nutzern und Gruppen gespeichert werden mit Ausnahme von Gruppenmitgliedschaften und Transaktionen. Für Nutzer- bzw. Gruppeninformationen mit Mitgliedschaften gibt es die Klassen `UserMembershipInfo` und `GroupMembershipInfo`.

Buchungen werden durch die Klasse `Transaction` modelliert. Gemäß $\langle RC14 \rangle$ gibt es genau zwei Subklassen von `Transaction`, nämlich `Expense` und `Payment`, wobei `Expense` zusätzlich einen Gesamtbetrag speichert.

Weiterhin gibt es den generischen Datentypen `Result<T>`, der entweder ein Ergebnis vom Typ `T` oder eine Fehlermeldung enthält. Auf diese Weise werden mögliche Fehler explizit gemacht, da Kotlin keine *checked exceptions* besitzt.

Um den Server gemäß $\langle RC10 \rangle$ konfigurierbar zu machen, gibt es außerdem die Klassen `Config` und `ConfigKey`. Letzteres ist eine Enumeration über alle bekannten Konfigurationsschlüssel. Die Klasse `Config` übernimmt einen Dateipfad und liest alle bekannten Konfigurationsschlüssel aus einer Datei³ im *properties*⁴-Format ein.

3.2.2. Serverschnittstellen

3.2.2.1. Request Handler



Abbildung 3.6.: Schnittstelle des *Request Handler*

³Beim Deployment des Servers wird diese Datei vom Debian-Paket erstellt

⁴[https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Properties.html#load\(java.io.Reader\)](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Properties.html#load(java.io.Reader))

Die Schnittstelle des **Request Handler** (siehe Abbildung 3.6) ermöglicht die Erstellung einer neuen Instanz, wobei hier im Konstruktor zum einen die zu verwendende Config sowie mittels Dependency Injection die konkreten Implementationen der vom **Request Handler** verwendeten Interfaces übergeben werden. Außerdem wird eine Methode zum Starten und Stoppen des HTTP Servers bereitgestellt. Implizit umfasst die Schnittstelle außerdem die in Abschnitt 2.4.2 beschriebene REST API, die über HTTP bereitgestellt wird.

3.2.2.2. Request Executor

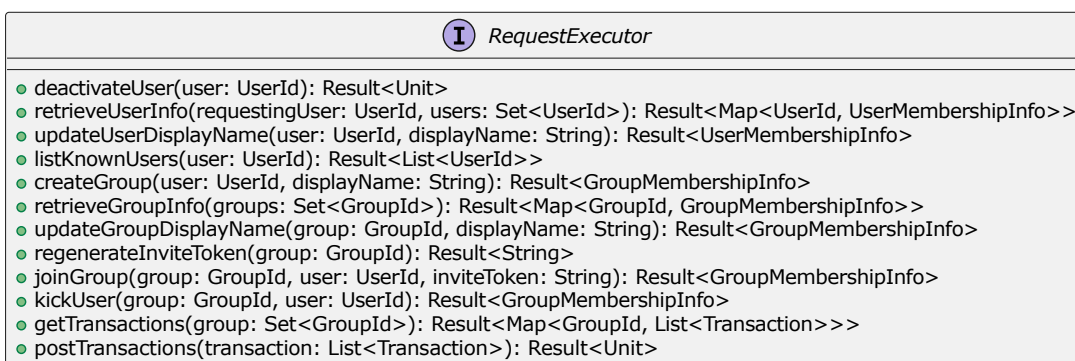


Abbildung 3.7.: Schnittstelle des *Request Executor*

Die Schnittstelle des **Request Executor** (siehe Abbildung 3.7) bietet für jede Route der REST API, die nicht ausschließlich der Anmeldung dient, eine Methode, die direkt die geforderte Funktionalität der jeweiligen Route abdeckt. Für alle diese Routen ist das Ergebnis der Methoden jeweils das Ergebnis, welches über die API an den Client zurückgesendet werden soll, oder im Fall eines Fehlers eine entsprechende Fehlermeldung.

3.2.2.3. Authentifizierungsschnittstelle

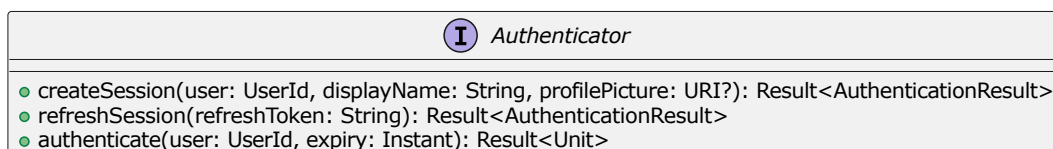


Abbildung 3.8.: Authentifizierungsschnittstelle der Nutzerverwaltung

Die Authentifizierungsschnittstelle (siehe Abbildung 3.8) erlaubt es, Nutzer zu authentifizieren und Refresh Tokens zu verwalten. Sie stellt folgende drei Methoden bereit:

createSession(UserId, String, URI?): Result<AuthenticationResult>

Die Methode `createSession` legt neue Sitzungen an und wird verwendet, wenn der Client die `/login`-Route mit einem ID Token verwendet. Dazu übernimmt sie die Nutzer ID, den initialen Anzeigenamen und falls vorhanden eine URI auf das Profilbild des Nutzers aus den Profildaten im gesendeten ID Token⁵. Sollte noch kein Nutzeraccount mit der gegebenen Nutzer ID existieren, so wird zuerst ein neuer Account angelegt, wobei der Anzeigename standardmäßig aus dem ID Token übernommen wird. Das Profilbild ersetzt, falls im ID Token angegeben, das vorherige Profilbild. Ein neuer Refresh Token wird angelegt und zusammen mit den Nutzerinformationen in einem `AuthenticationResult` zurückgegeben.

refreshSession(String): Result<AuthenticationResult>

Die Methode `refreshSession` erneuert Sessions und wird verwendet, wenn der Client die `/login`-Route mit einem Refresh Token verwendet. Dazu übernimmt sie den alten Refresh Token und findet den dazu assoziierten Nutzer. Der alte Refresh Token wird invalidiert und es wird ein neuer erzeugt, welcher zusammen mit den Nutzerinformationen in einem `AuthenticationResult` zurückgegeben wird.

authenticate(UserId, Instant): Result<Unit>

Die Methode `authenticate` überprüft, ob ein Nutzerzugriff mit einem Access Token zu einer bestimmten Ablaufzeit gültig ist. Dazu übernimmt sie die Nutzer ID des zu authentifizierenden Nutzers, sowie den Ablaufzeitpunkt des Access Token, welches für die Anfrage verwendet wurde. Das Ergebnis ist genau dann erfolgreich, wenn der Nutzer nicht deaktiviert ist und die aktuelle Serverzeit nicht hinter dem Ablaufzeitpunkt liegt.

3.2.2.4. Autorisierungsschnittelle



Abbildung 3.9.: Autorisierungsschnittelle der Gruppenverwaltung

⁵https://openid.net/specs/openid-connect-core-1_0.html#IDToken

Die Autorisierungsschnittstelle (siehe Abbildung 3.9) überprüft, ob ein Nutzer die Berechtigung hat, Aktionen innerhalb einer bestimmten Gruppe durchzuführen. Dazu stellt sie folgende Methode bereit:

authorize(user: UserId, group: GroupId): Result<Unit>

Die Methode authorize überprüft, ob der gegebene Nutzer aktiv und Mitglied der gegebenen Gruppe ist. Genau in diesem Fall ist das Ergebnis erfolgreich.

3.2.2.5. Nutzeradministration

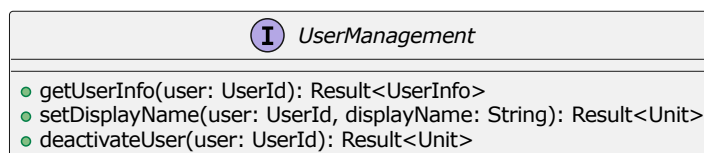


Abbildung 3.10.: Nutzeradministrationsschnittstelle der Nutzerverwaltung

Die Schnittstelle zur Nutzeradministration (siehe Abbildung 3.10) verwaltet Nutzer im System. Allerdings können hier keine neuen Nutzer angelegt werden, da dies nur über die Anmeldung mittels OpenID Connect möglich ist. Sie stellt folgende Methoden bereit:

getUserInfo(UserId): Result<UserInfo>

Die Methode getUserInfo nimmt eine Nutzer ID entgegen und ruft die Daten zu diesem Benutzer ab. Ist der Nutzer deaktiviert so wird ein UserInfo Objekt zurückgegeben, bei dem der Anzeigename der Nutzer ID entspricht und kein Profilbild existiert.

setDisplayName(UserId, String): Result<Unit>

Die Methode setDisplayName nimmt einen Nutzer ID und einen neuen Anzeigenamen für diesen Benutzer entgegen und aktualisiert den Anzeigenamen des Benutzers entsprechend. Ist der Nutzer deaktiviert oder existiert nicht, schlägt die Methode fehl.

deactivateUser(UserId): Result<Unit>

Die Methode deactivateUser nimmt eine Nutzer ID entgegen und deaktiviert den entsprechenden Nutzer. Ist der Nutzer bereits deaktiviert, passiert nichts. Existiert der Nutzer nicht, schlägt die Methode fehl. deactivateUser entfernt den Nutzer explizit nicht aus den Gruppen, in denen er ein Saldo von Null hat.

3.2.2.6. Gruppenadministration


 <i>GroupManagement</i>
<ul style="list-style-type: none"> • <code>createGroup(initialMember: UserId, displayName: String): Result<GroupId></code> • <code>getGroupInfo(group: GroupId): Result<GroupInfo></code> • <code>setDisplayname(group: GroupId, displayName: String): Result<Unit></code> • <code>getUserGroups(user: UserId): Result<Set<GroupId>></code> • <code>getGroupMembers(group: GroupId): Result<Set<UserId>></code> • <code>filterKnownUsers(actingUser: UserId, users: List<UserId>): Result<List<UserId>></code> • <code>addMember(group: GroupId, user: UserId): Result<Unit></code> • <code>removeMember(group: GroupId, user: UserId): Result<Unit></code> • <code>regenerateInviteToken(group: GroupId): Result<String></code>

Abbildung 3.11.: Gruppenadministrationsschnittstelle der Nutzerverwaltung

Die Schnittstelle zur Gruppenadministration (siehe Abbildung 3.11) verwaltet Gruppen und Gruppenmitgliedschaften im System. Sie stellt folgende Methoden bereit:

`createGroup(UserId, String): Result<GroupId>`

Die `createGroup` Methode nimmt eine Nutzer ID und einen Gruppennamen entgegen und erzeugt eine neue Gruppe mit dem gegebenen Gruppennamen. Der übergebene Nutzer ist initial das einzige Mitglied der Gruppe. Die Methode gibt die Gruppen ID der neu erzeugten Gruppe zurück.

`getGroupInfo(GroupId): Result<GroupInfo>`

Die Methode `getGroupInfo` nimmt eine Gruppen ID entgegen und ruft die Daten zu dieser Gruppe ab. Existiert die Gruppe nicht, schlägt die Methode fehl.

`setDisplayname(GroupId, String): Result<Unit>`

Die Methode `setDisplayname` nimmt eine Gruppen ID und einen neuen Anzeigenamen für diese Gruppe entgegen und aktualisiert den Anzeigenamen der Gruppe entsprechend. Existiert die Gruppe nicht, schlägt die Methode fehl.

`getUserGroups(UserId): Result<Set<GroupId>>`

Die `getUserGroups` Methode nimmt eine Nutzer ID entgegen und gibt die Gruppen IDs aller Gruppen zurück, in denen der Nutzer Mitglied ist. Existiert der Nutzer nicht, schlägt die Methode fehl.

`getGroupMembers(GroupId): Result<Set<UserId>>`

Die `getGroupMembers` Methode nimmt eine Gruppen ID entgegen und gibt die Nutzer IDs aller Nutzer zurück, die Mitglied der Gruppe sind. Existiert die Gruppe nicht, schlägt die Methode fehl.

filterKnownUsers(UserId, List<UserId>): Result<List<UserId>>

Die filterKnownUsers Methode nimmt die Nutzer ID eines *agierenden Nutzers*, sowie eine Liste von weiteren Nutzer IDs entgegen. Das Ergebnis ist eine Liste von Nutzer IDs, die genau diejenigen Nutzer IDs der Eingabeliste enthält, für deren Nutzer mindestens eine der folgenden Aussagen wahr ist:

- Der Nutzer ist der *agierende Nutzer*.
- Es existiert mindestens eine Gruppe, in der sowohl der zu überprüfende Nutzer, als auch der *agierende Nutzer* Mitglied sind.

Existiert der agierende Nutzer nicht, ist das Ergebnis die leere Liste.

addMember(GroupId, UserId): Result<Unit>

Die addMember Methode nimmt eine Gruppen und eine Nutzer ID entgegen. Ist der Nutzer kein Mitglied der Gruppe, so wird er der Gruppe hinzugefügt. Ist der Nutzer bereits Mitglied der Gruppe, passiert nichts. Existiert der Nutzer oder die Gruppe nicht, schlägt die Methode fehl.

removeMember(GroupId, UserId): Result<Unit>

Die removeMember Methode nimmt eine Gruppen und eine Nutzer ID entgegen. Ist der Nutzer Mitglied der Gruppe, so wird er aus der Gruppe entfernt. Dabei wird nicht geprüft, ob das Saldo des Nutzer Null beträgt. Besteht die Gruppe nach Entfernen des Nutzers nur noch aus deaktivierten Nutzern, wird sie gelöscht. Dabei werden auch alle Buchungen in der Gruppe gelöscht. Ist der Nutzer kein Mitglied der Gruppe, passiert nichts. Existiert der Nutzer oder die Gruppe nicht, schlägt die Methode fehl.

regenerateInviteToken(GroupId): Result<String>

Die regenerateInviteToken Methode nimmt eine Gruppen ID entgegen, erzeugt ein neues Einladungstoken und gibt dieses zurück. Existiert die Gruppe nicht, schlägt die Methode fehl.

3.2.2.7. Buchungsadministration


 TransactionManagement
<ul style="list-style-type: none"> • getTransactions(group: GroupId): Result<List<Transaction>> • postTransactions(transaction: List<Transaction>): Result<Unit> • getBalance(user: UserId, group: GroupId): Result<BigDecimal>

Abbildung 3.12.: Buchungsadministrationsschnittstelle der Nutzerverwaltung

Die Schnittstelle zur Buchungsadministration (siehe Abbildung 3.12) verwaltet Transaktionen in Gruppen. Sie stellt folgende Methoden bereit:

getTransactions(GroupId): Result<List<Transaction>>

Die getTransactions Methode nimmt eine Gruppen ID entgegen und gibt eine Liste von allen Transaktionen in der Gruppe zurück. Existiert die Gruppe nicht, schlägt die Methode fehl.

postTransactions(List<Transaction>): Result<Unit>

Die postTransactions Methode nimmt eine Liste von Transaktionen entgegen und fügt diese zu den entsprechenden Gruppen hinzu. Die Transaktionen werden unverändert übernommen. Insbesondere wird der Zeitstempel nicht auf die aktuelle Serverzeit gesetzt. postTransactions garantiert, dass am Ende der Methode entweder alle oder keine Transaktion hinzugefügt wurde. Existiert eine, in den Transaktionen referenzierte Gruppe oder ein, in den Transaktionen referenzierter Nutzer nicht, schlägt die Methode fehl.

getBalance(UserId, GroupId): Result<BigDecimal>

Die getBalance Methode nimmt eine Nutzer und eine Gruppen ID entgegen und gibt das Saldo des Nutzers in der Gruppe zurück. Existiert der Nutzer oder die Gruppe nicht, schlägt die Methode fehl.

3.2.3. Request Handler

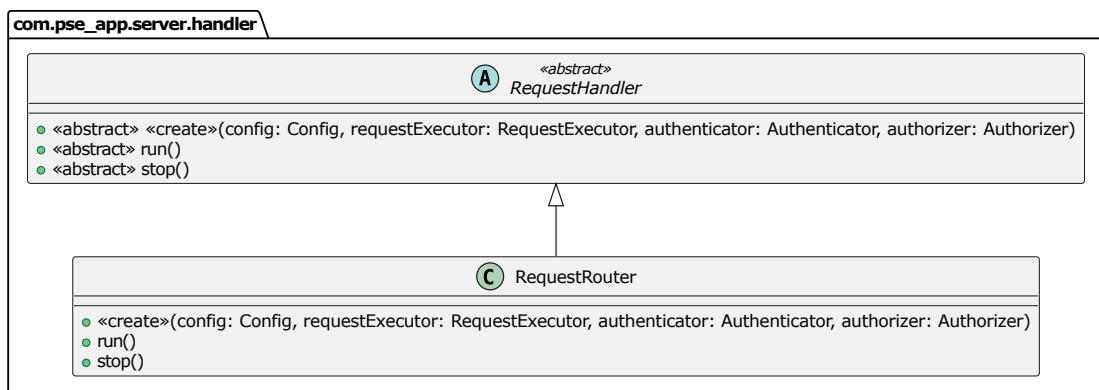


Abbildung 3.13.: Detailentwurf des *Request Handler*

Die *Request Handler*-Schnittstelle wird, wie in Abbildung 3.13 gezeigt, von einer Klasse `RequestRouter` bereitgestellt. Diese startet einen HTTP-Server mittels Ktor und stellt die in Abschnitt 2.4.2 beschriebene REST-API bereit. Bei einem Request an eine

der bereitgestellten Routen überprüft der `RequestRouter` zuerst unter Verwendung der ihm im Konstruktor übergebenen Instanzen der Authentifizierungs- sowie Autorisierungsschnittstelle, ob der Nutzer hinter dem Request die nötigen Rechte besitzt, um die von ihm angeforderte Aktion durchzuführen. Ist dies der Fall, so übersetzt der `RequestRouter` den Request in einen Aufruf einer der Methoden der *Request Executor*-Schnittstelle auf der ihm im Konstruktor übergebenen Instanz. Dient der Request lediglich der Anmeldung, so übersetzt der `RequestRouter` den Request direkt in Aufrufe der Authentifizierungsschnittstelle auf der ihm im Konstruktor übergebenen Instanz.

3.2.4. Request Executor

Die *Request Executor*-Schnittstelle wird, wie in Abbildung 3.15 gezeigt, von einer Klasse `RequestDispatcher` bereitgestellt. Diese hält Referenzen auf die Instanzen der Klassen `UserDispatcher`, `GroupDispatcher` und `TransactionDispatcher` und leitet die einzelnen Anfragen an diese weiter. Dabei werden Anfragen, die Nutzer betreffen an den `UserDispatcher` weitergegeben, Anfragen, die Gruppen bzw. Gruppenmitgliedschaften betreffen, werden an den `GroupDispatcher` weitergegeben und Anfragen, die Buchungen betreffen, werden an den `TransactionDispatcher` weitergegeben. Diese führen die Anfrage aus und greifen dazu auf die Nutzer-, Gruppen- und Buchungsadministration zu.

3.2.5. Nutzerverwaltung

Die Nutzerverwaltung besteht, wie in Abbildung 3.16 zu sehen, aus zwei Klassen: dem `UserManager` und dem `UserAuthenticator`. `userManager` stellt dabei die Schnittstelle zur Nutzeradministration bereit. Ein `userManager` enthält dabei außerdem stets einen `UserAuthenticator`, welcher die Authentifizierungsschnittstelle bereitstellt. Dieser kann über die Methode `getAuthenticator()` des `userManager` abgerufen werden. Zum Zugriff auf die Datenbank erhält der `userManager` bei der Erstellung ein `Database`-Objekt der *Ktorm*-Bibliothek.

3.2.6. Gruppenverwaltung

Die Gruppenverwaltung besteht, wie in Abbildung 3.17 zu sehen, aus zwei Klassen: dem `GroupManager` und dem `GroupAuthorizer`. `groupManager` stellt dabei die Schnittstelle zur Gruppenadministration bereit. Ein `groupManager` enthält dabei außerdem stets einen `GroupAuthorizer`, welcher die Autorisierungsschnittstelle bereitstellt. Dieser kann über die Methode `getAuthorizer()` des `groupManager` abgerufen werden. Zum Zugriff auf

die Datenbank erhält der GroupManager bei der Erstellung ein Database-Objekt der *Ktorm*-Bibliothek.

3.2.7. Buchungsverwaltung

Die Buchungsverwaltung besteht, wie in Abbildung 3.18 gezeigt, aus einer Klasse TransactionManager, welche die Buchungsadministration bereitstellt. Diese erhält zum Zugriff auf die Datenbank bei der Erstellung ein Database-Objekt der *Ktorm*-Bibliothek.

3.2.8. Datenbankzugriff

Der Datenbankzugriff geschieht über die *Ktorm*-Bibliothek. Um die Datenbankverbindung aufzubauen und die Datenbank beim ersten Start zu initialisieren, existiert das Singleton-Objekt DatabaseInitializer (Abbildung 3.19). Dieses stellt eine Methode connectDatabase bereit, die die Verbindung zum Datenbankserver aufbaut, die Datenbank initialisiert und ein Database-Objekt der *Ktorm*-Bibliothek zurückgibt.

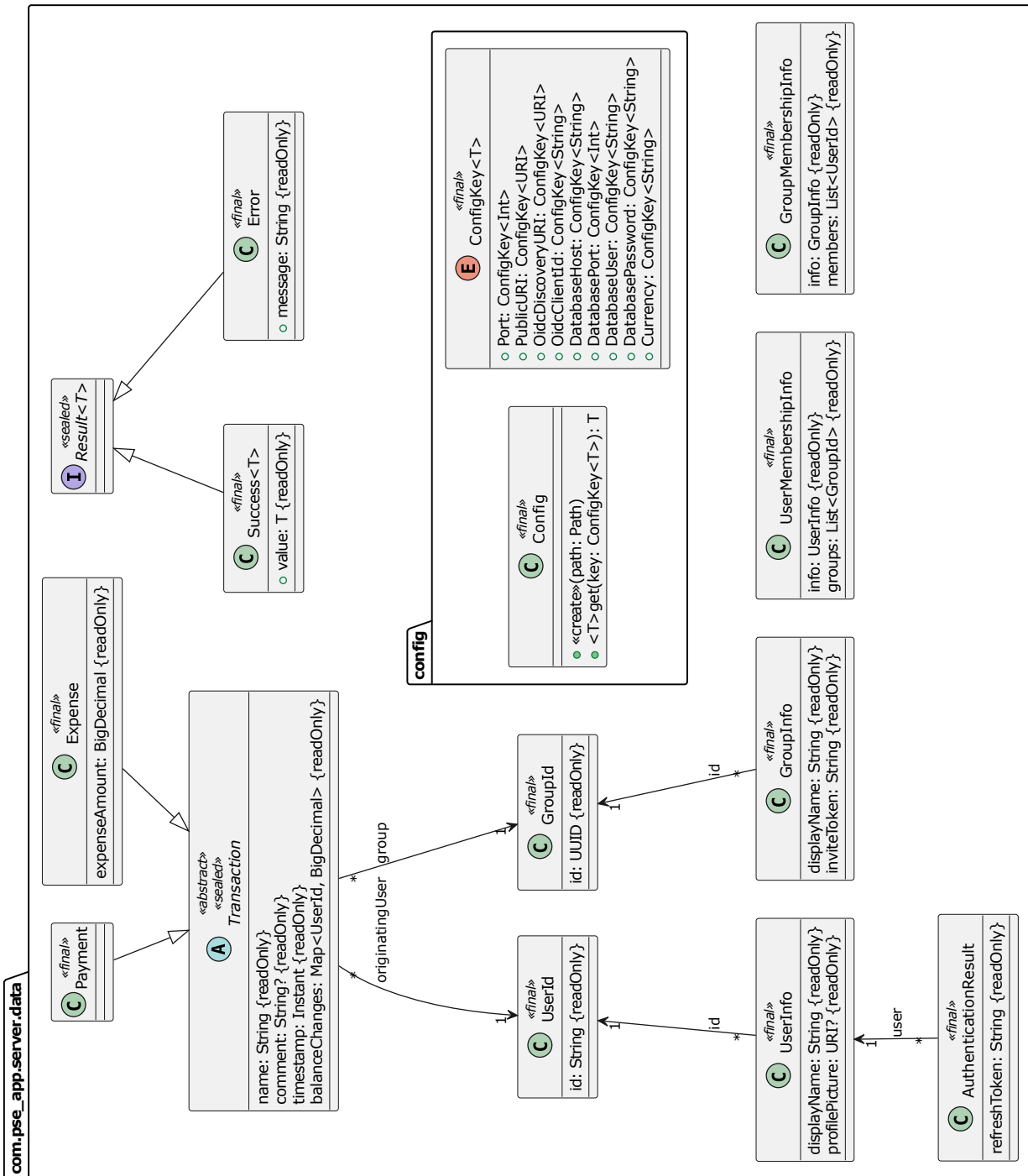


Abbildung 3.14.: Allgemeine Datentypen auf dem Server

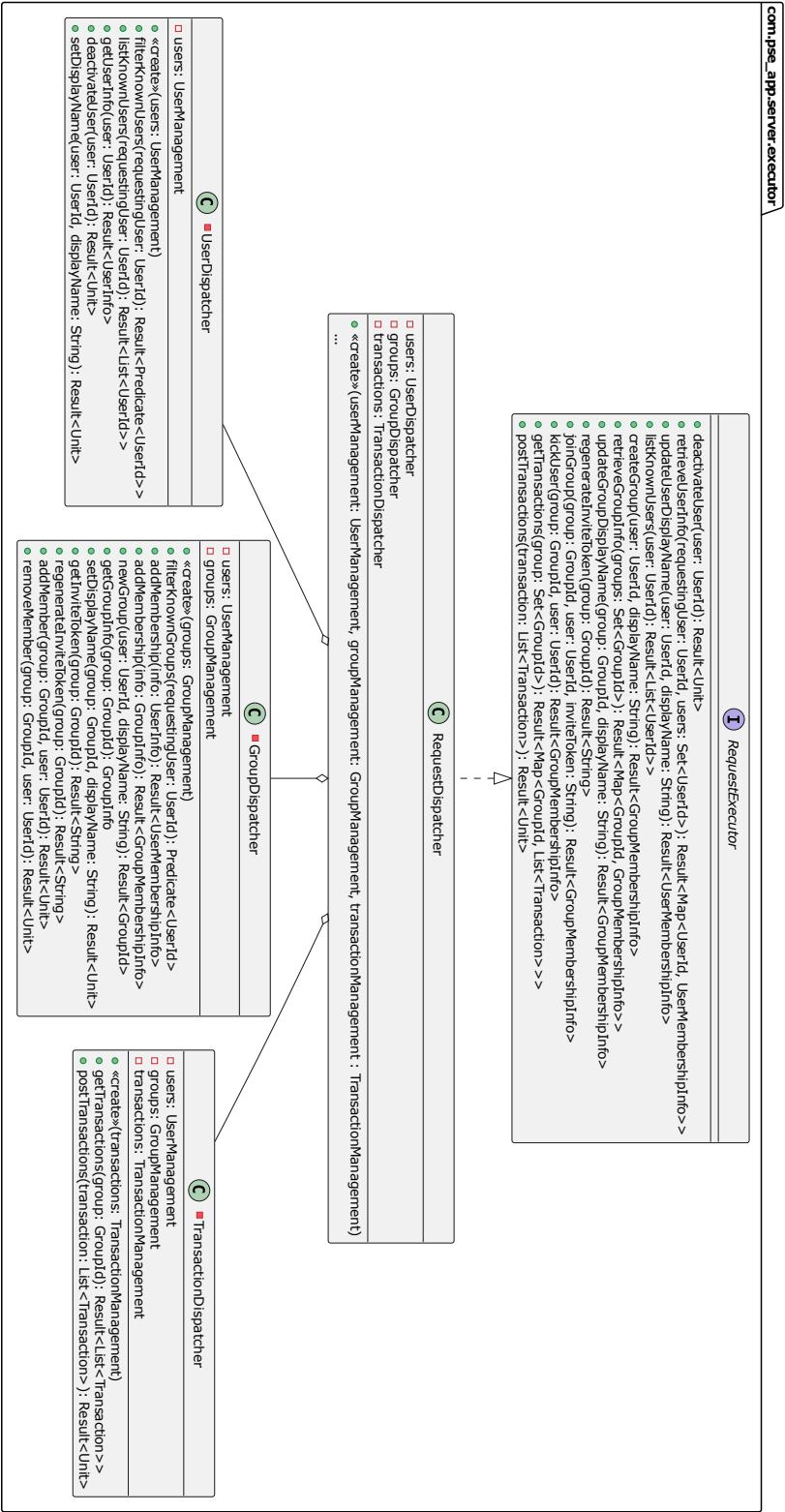


Abbildung 3.15.: Detailentwurf des *Request Executor*

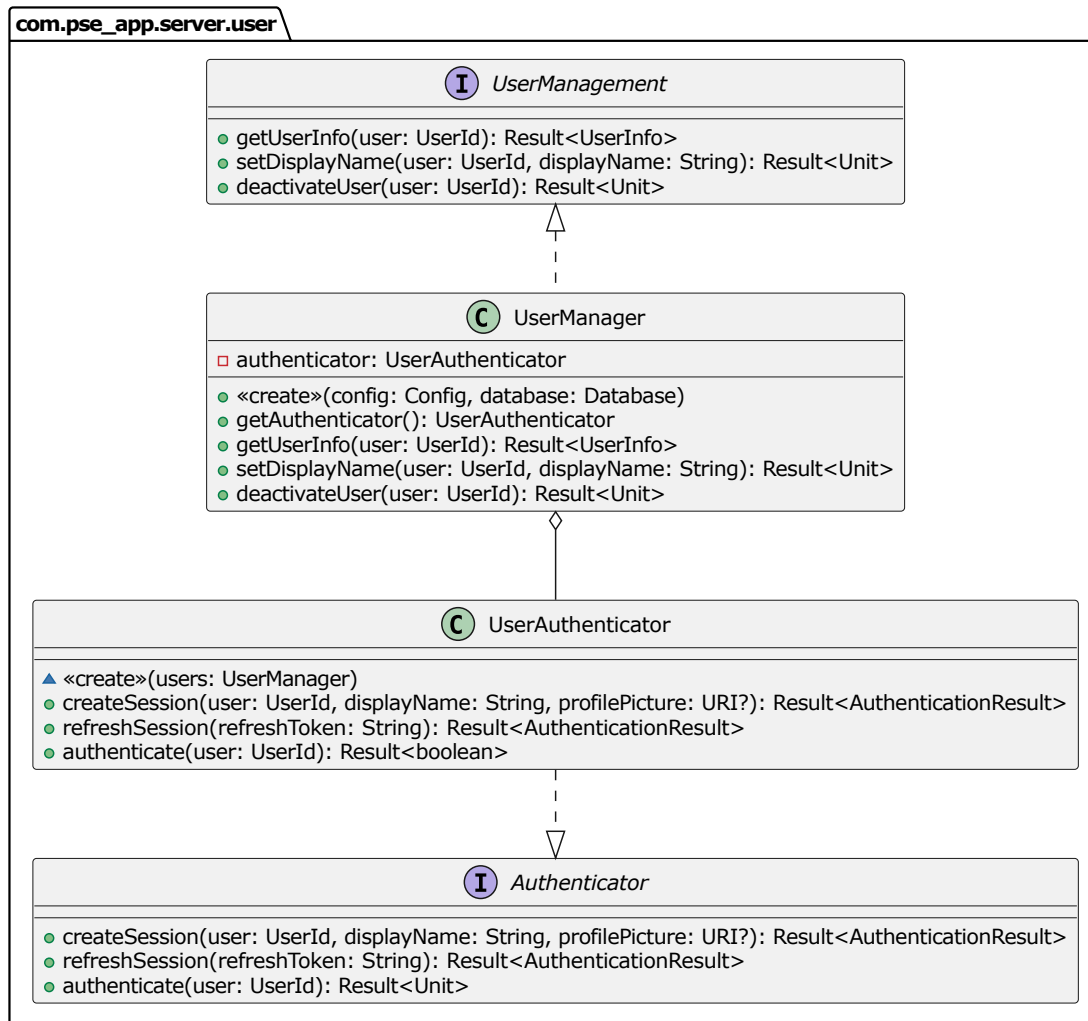


Abbildung 3.16.: Detailentwurf der Nutzerverwaltung

3. Feinentwurf

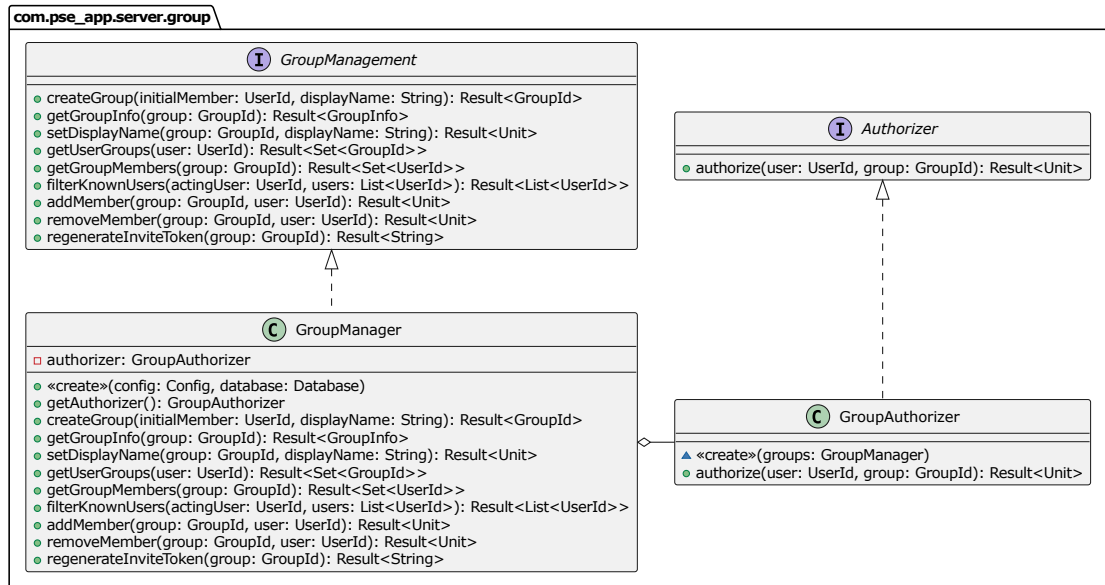


Abbildung 3.17.: Detailentwurf der Gruppenverwaltung

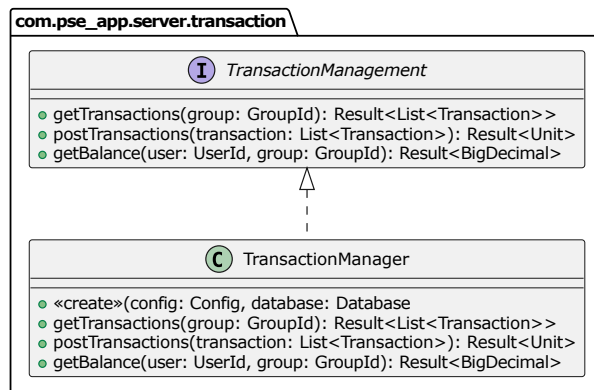


Abbildung 3.18.: Detailentwurf der Buchungsverwaltung

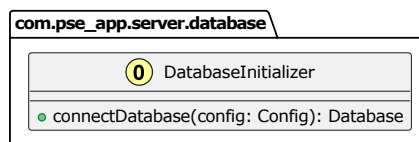


Abbildung 3.19.: Detailentwurf des Datenbankzugriffs

3.3. Client Entwurf

3.3.1. UI

3.3.1.1. View und ViewModel Umsetzung

Die Benutzeroberfläche und damit die View Komponente wird im Rahmen der Jetpack Compose Library in Form von sogenannten Composables beschrieben. Eine Composable ist eine mit `Composable` annotierte Funktion, mit der alle UI Elemente beschrieben werden. Unter anderem wird jeder Screen der App durch eine „Screen-level“ Composable beschrieben

Die ViewModel Komponente besteht hingegen aus mehreren einzelnen ViewModel Klassen. Dabei gehört zu jeder ViewModel Klasse genau ein Screen, dessen Zustand von eben diesem ViewModel verwaltet wird.

Jede Screen Composable wird also mit einem passenden `AbstractViewModel` initialisiert. Die `AbstractViewModel` Schnittstelle definiert Flows und Aktionen, die für den Screen gebraucht werden und wird von einer passenden ViewModel Klasse implementiert. Dies erlaubt Tests, die ViewModels auszutauschen, ohne dabei etwas an den Views verändern zu müssen.

Abbildungen 3.21 und 3.22 zeigen sämtliche Screen Composables und die von ihnen erwarteten ViewModels. Diese Abbildungen befinden sich zusätzlich im Anhang als `views_main.pdf` und `views_group.pdf`.

3.3.1.2. View-ViewModel Schnittstelle

Die *View State* Schnittstelle, welche das UI Aussehen der Composables mit dem UI Zustand der ViewModels synchronisiert, haben wir mit Kotlin `StateFlow` implementiert. Ändert sich der geforderte Wert, schickt das ViewModel ein neues Zustandsobjekt über den Flow und Jetpack Compose rendert die View automatisch neu.

Um andersrum eine Aktion des Nutzers, beispielsweise eine Nutzereingabe in einem Textfeld, zu signalisieren, ruft die View eine der entsprechenden asynchronen Methoden des ViewModels auf. Dieses kontrolliert dann, wie die Zustandsänderung umgesetzt werden soll und kommuniziert insbesondere diese finale Änderung in der Regel über die Flows zurück an die View.

Um vom konkreten Model unabhängig zu sein, werden die Daten die an die View gegeben werden in Form von einfacheren Datenklassen ohne verschachtelte Flows übergeben. Diese sind in Abbildung 3.20 abgebildet.

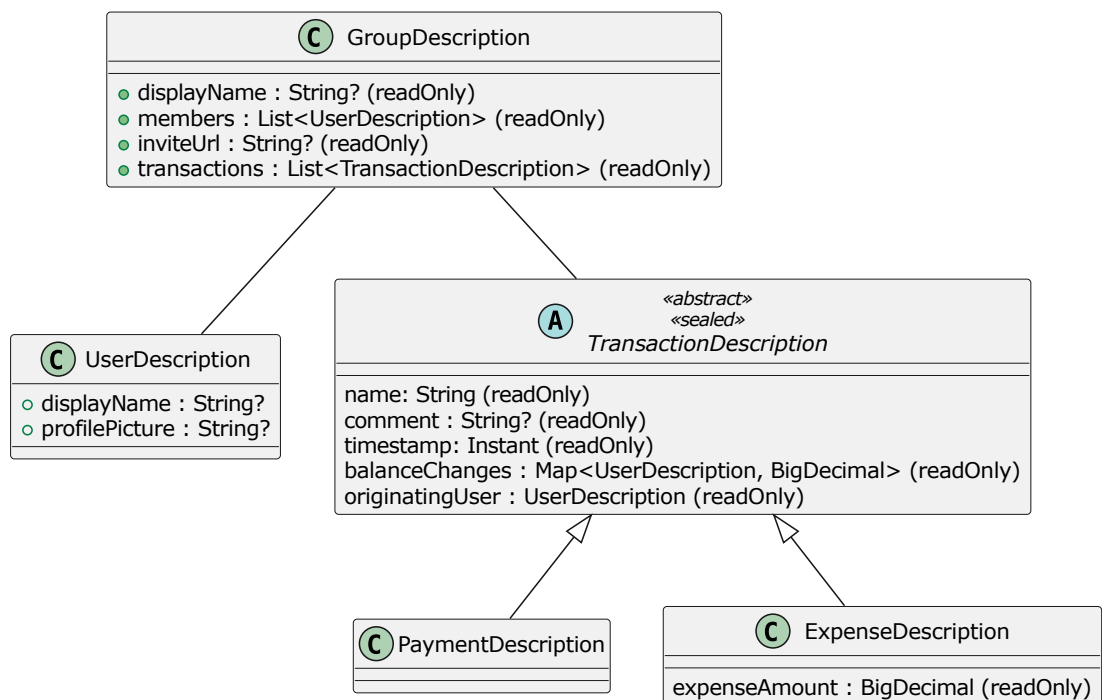


Abbildung 3.20.: Datenklassen der ViewModels

Auch Navigationseingaben und Buttons werden ausgeführt indem die View eine Methode des ViewModels ausführt, welche dann selbst mit dem Model kommuniziert, bzw. die aktive View und das aktive ViewModel wechselt.

3.3.2. ViewModels

Es folgt eine Beschreibung der einzelnen ViewModel Klassen. Diese sind zusätzlich noch in Abbildung 3.21 und 3.22 beschrieben.

MainMenuViewModel

Dieses ViewModel verwaltet das Hauptmenü. Es speichert den Zustand eines Hauptmenüs (MainMenuUIState), der eine Liste von Gruppen (groups) enthält. Der Nutzer kann über dieses ViewModel eine Gruppe auswählen (selectGroup(group: GroupDescription)), neue Gruppen erstellen (createGroup()), oder zu anderen Ansichten wie dem Nutzerprofil (navigateToProfile()) oder dem Settle-Up-Ansicht (navigateToSettleUp()) navigieren.

ProfileViewModel

Dieses ViewModel verwaltet das aktive Nutzerprofil. Es speichert dazu den aktuellen Zwischenwert des Anzeigenamens, der vom Nutzer in das entsprechende Textfeld eingetragen wurde, welcher durch die enterDisplayName Methode gesetzt werden kann.

Um den fertigen Anzeigenamen zu übernehmen, wird confirmChangeDisplayNameRequest ausgeführt, welche entweder true zurückgibt, falls der angegebene gültig war oder falsch, falls er es nicht war, in welchem Fall die View eine entsprechende Fehlermeldung anzeigt.

Darüber hinaus bietet das ViewModel Methoden, um den Nutzer abzumelden (logout()) oder sein Konto zu deaktivieren (deactivateAccount()).

CreateGroupViewModel

Dieses ViewModel ist für die Erstellung neuer Gruppen zuständig. Es verwaltet einen Zustand (CreateGroupUIState), der den Gruppennamen (groupName) des entsprechenden Textfelds und den aktuellen Einladungslink (inviteLink) enthält. Von der View kann der aktuelle Gruppennamen geändert (changeName(newName: String)), ein neuer Einladungslink erstellt (regenerateInviteLink()) und die neue Gruppe endgültig erstellt werden (createGroup()).

LoginViewModel

Dieses ViewModel verwaltet den Login-Prozess. Der Nutzer kann sich über dieses ViewModel anmelden (login()).

Bei Betätigung des Login-Buttons wird mithilfe von *AppAuth* ein *WebView* geöffnet, welcher die Anmeldeseite des Identity Providers anzeigt. Nachdem der Nutzer sich erfolgreich eingeloggt hat, fängt *AppAuth* die Antwort der Website ab, extrahiert den Authorisierungscode und leitet sie an einen spezifizierten Intent weiter. Nach einer Token Exchange Request, welche wieder mithilfe von *AppAuth* durchgeführt wird, wird der erhaltene ID Token an die *login* Methode der *ModelFacade* übergeben (siehe 3.3.5.3).

GroupViewModel

Das *GroupViewModel* verwaltet die Gruppen und ihre Buchungen. Es speichert den Zustand *GroupViewState*, der eine Liste der anzuzeigenden Transaktionen (*transactions*) enthält. Die View kann darüber neue Buchungen erstellen (*createTransaction()*) und zur *GroupSettingsView* wechseln (*navigateToGroupSettings()*).

GroupSettingsViewModel

Das *GroupSettingsViewModel* verwaltet die Gruppeneinstellungen. Es hält den Zustand der Gruppeneinstellungen (*GroupSettingsState*), einschließlich des Gruppennamens (*groupName*) und des Einladungslinks (*inviteLink*).

Die View kann darüber den Gruppennamen ändern (*changeGroupName(newName: String)*), einen neuen Einladungslink generieren (*regenerateInviteLink()*), den Link in die Zwischenablage kopieren (*copyInviteLinkToClipboard()*), einen Nutzer entfernen (*kickUser(user: UserDescription)*), oder die Gruppe verlassen (*leaveGroup()*).

ExpenseViewModel

Das *ExpenseViewModel* verwaltet die Ausgaben innerhalb einer Gruppe. Es speichert den Zustand einer Ausgabe (*ExpenseState*), der den Namen der Ausgabe (*name*), den gesamten Betrag der Ausgabe (*total*), sowie ein Mapping von Nutzern zu den für diese eingestellten Beträgen der Ausgabe (*userAmounts*) enthält.

Die View kann den Namen der Ausgabe ändern (*changeName(newName: String)*), die Beträge für einzelne Nutzer anpassen (*changeAmountForUser(user: UserDescription, newAmount: BigDecimal)*), und die Ausgabe abschließend bestätigen (*confirm()*). Zudem erlaubt das ViewModel die Navigation zur Zahlungsansicht (*navigateToPayment()*).

PaymentViewModel

Das *PaymentViewModel* verwaltet die Zahlungen innerhalb einer Gruppe. Es verwaltet den Zustand der Zahlungen (*PaymentState*), der Saldi der Nutzer (*userBalances*), überwiesene Beträge an Nutzern (*amountsSentToUsers*) und Kommentare zu Zahlungen (*comment*).

Das ViewModel bietet Methoden, um den an einen Nutzer gezahlten Betrag festzulegen (`setAmountForUser(user: UserDescription, amount: BigDecimal)`), die Kommentare zu ändern (`changeComment(newComment: String)`) und zur Ausgabensicht zu navigieren (`navigateToExpense()`).

SettleUpUserSelectionViewModel

Das `SettleUpUserSelectionViewModel` verwaltet die Auswahl von Nutzern für die Abgleichung. Es speichert ein Mapping von Nutzern zu vorgeschlagenen Beträgen (`getProposedTotals`).

Das ViewModel ermöglicht das Abrufen der Gesamtschuld des abgleichenden Nutzers (`getTotalOwed()`) und einer Liste von Nutzern, die für die Abgleichung ausgewählt werden können (`getUsers()`). Außerdem erlaubt es, Nutzer für die Abgleichung auszuwählen (`selectUser(user: UserDescription)`), womit auf die `SettleUpGroupSelectionView` gewechselt wird.

SettleUpGroupSelectionViewModel

Das `SettleUpGroupSelectionViewModel` verwaltet die Auswahl der Gruppen, die in die Abgleichung einbezogen werden sollen. Es speichert den Zustand `SettleUpGroupSelectionState`, einschließlich der in die Abgleichung einbezogenen Gruppen (`includedGroups`) und der Beträge, die in diesen Gruppen ausgeglichen werden sollen (`amountsGivenInGroup`).

Die View kann die Beträge für eine Gruppe ändern (`changeAmountForGroup(group: GroupDescription, newAmount: BigDecimal)`), Gruppen in die Abgleichung ein- oder ausschließen (`changeGroupSelection(group: GroupDescription, newSelectionState: Boolean)`), die Abgleichung abschließen (`settle()`), und zur vorherigen Ansicht navigieren (`navigateBack()`). Außerdem ermöglicht das ViewModel das Abrufen des Nutzers mit dem abgeglichen wird (`getUser()`).

3.3.3. Navigation

Die verschiedenen Oberflächen der App laufen alle in einer Android Activity. Dies wird durch das Compose Navigationssystem ermöglicht, welches je nach aktueller Route im Navigationsgraph, das angezeigte Screen Composable auswählt.

Der Eingangspunkt der App ist dabei die `EntryPoint Composable`. Aus Übersichtsgründen ist diese nicht in Abbildung 3.21, bzw. 3.22 enthalten. Sie enthält einen `NavHost` von dem aus alle Views bzw. ViewModels erreichbar sind und erstellt einen `NavController`, über den die ViewModels die aktive View wechseln können.

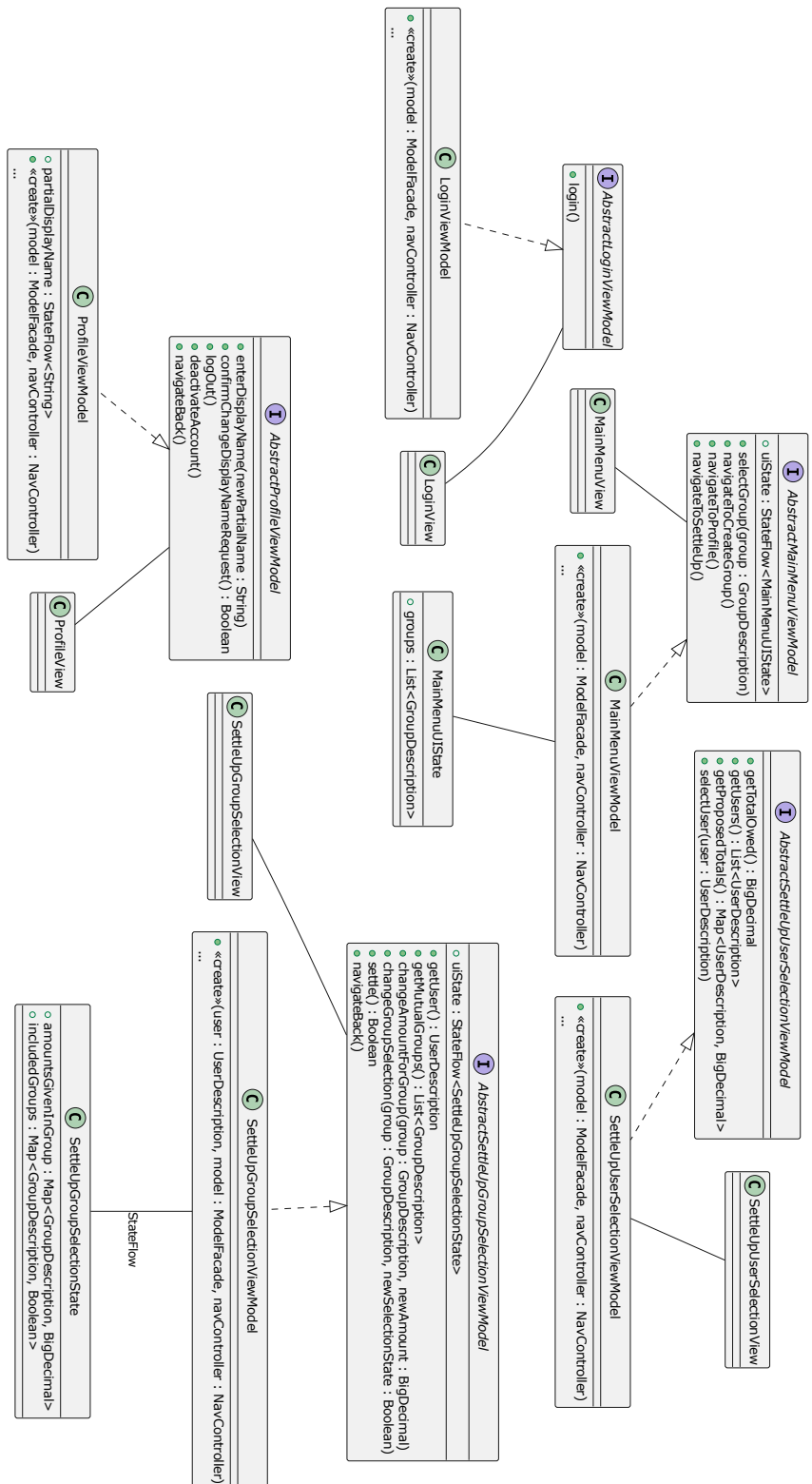


Abbildung 3.21.: Views und Viewmodels für das Hauptmenü

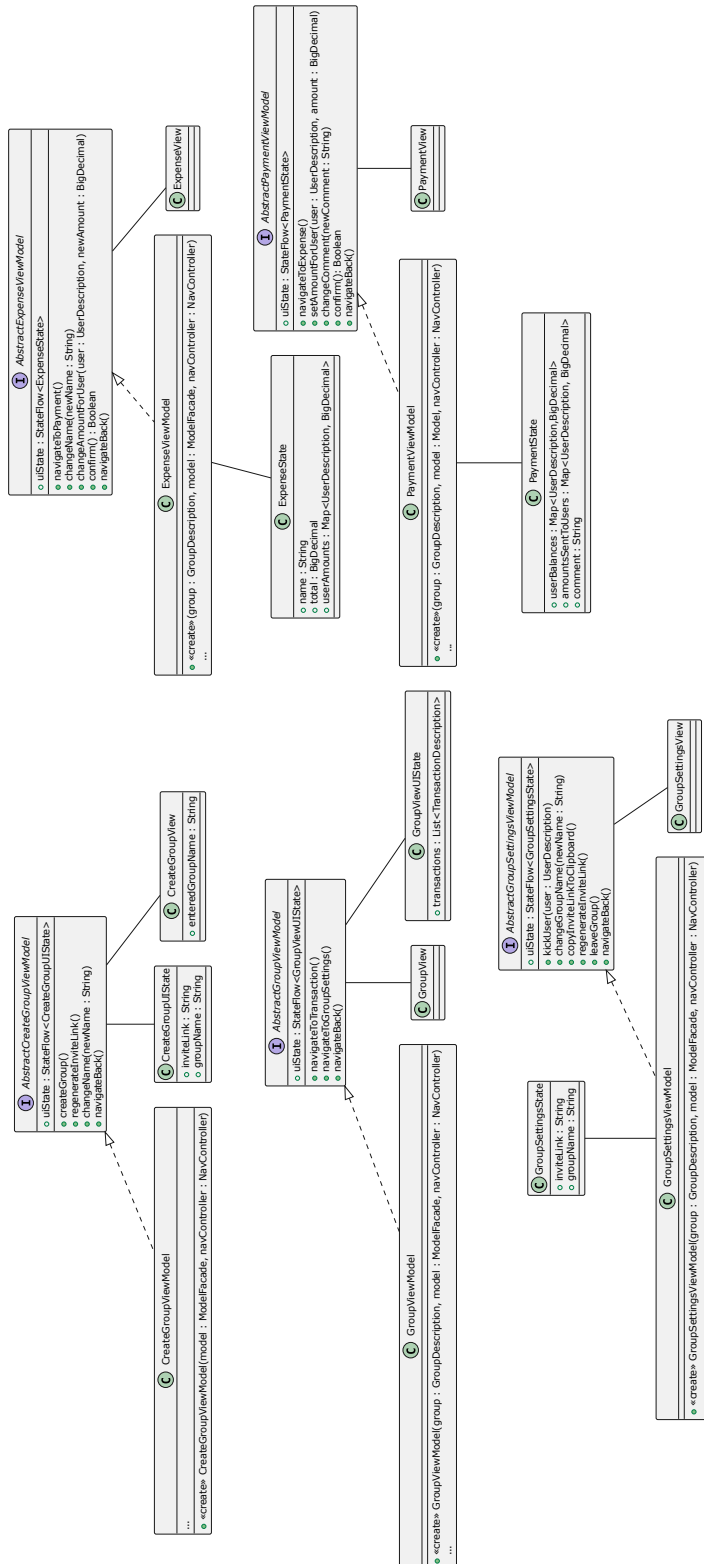


Abbildung 3.22.: Views und Viewmodels für Gruppen

Falls ein ViewModel nun die aktive View wechselt, navigiert es den NavController zur anderen Composable, welche dann dafür sorgt dass der EntryPoint neu gerendert wird und im NavHost die andere View auswählt.

3.3.3.1. Navigationsgraph

Der Navigationsgraph der Clientanwendung ist in Abbildung 3.23 gegeben. Jeder Zustand entspricht dabei einer View aus Abbildung 3.21 und 3.22. In der endgültigen Implementierung mit Jetpack Compose wird für jede dieser zusätzlich ein Navigationsobjekt mit den passenden Objektattributen existieren.

Die Übergänge des Diagramms beschreiben die Übergänge zwischen Views, wobei die Übergangsbeschreibung immer die Funktion des jeweiligen ViewModels beschreibt, welche die Navigation auslöst.

3.3.4. Lokalisierung

Für die Lokalisierung der App verwenden wir Androids Resource API. Konkret bedeutet dies, dass die zu lokalisierenden Strings in Resource Dateien gespeichert sind und Views diese durch einen Aufruf an Androids `stringResource` Funktion erhalten. Android wählt dann automatisch die Resource Datei aus, die zur Sprache des Users passt. Somit unterstützt die Benutzeroberfläche mehrere Sprachen $\langle RC3 \rangle$.

3.3.5. Model Fassade

Das Model bietet nach außen eine umfangreiche Schnittstelle `ModelFacade`, welche einen Großteil der Anfragen, die dem Model gestellt werden können, enthält. Alle restlichen Anfragen erfolgen über die User und Group Schnittstellen, deren Instanzen über `ModelFacade` erhalten werden. User und Group modellieren einen spezifischen Nutzer, bzw. eine spezifische Gruppe, und erlauben es, seine bzw. ihre Daten zu beobachten.

3.3.5.1. Kotlin Implementierung von Observe-Refresh

Die Refresh Schnittstelle wird durch *refresh* Methoden implementiert, welche einen bestimmten Teil des lokalen Zustands aktualisieren.

Für die in Abschnitt 2.2.3.4 beschriebene Observe Schnittstelle wird ein *observe* Mechanismus gebraucht, welcher den Beobachter bei Änderungen informiert. Solch ein



Abbildung 3.23.: Navigationsgraph der Clientanwendung

Mechanismus kann zum Beispiel durch Callbacks implementiert werden. Kotlin bietet hierfür jedoch sogenannte Flows⁶. Wir haben uns entschieden Flows zu verwenden, da sie von den verwendeten *Jetpack* Bibliotheken erstklassig unterstützt werden und mehr Klarheit gegenüber Callbacks bieten.

In Abbildung 3.24 wird beispielhaft die Verwendung von Flows gezeigt.

3.3.5.2. Error Handling

Anfragen an den Server können fehlschlagen. Dies ist zum Beispiel der Fall, wenn der Server nicht antwortet (Timeout), der Server einen Fehler meldet oder der Server die Session nicht akzeptiert. Aus diesem Grund werfen manche Methoden potentiell eine `TimeoutException`, `ServerErrorException` oder `AuthException`.

In den ViewModels werden diese Exceptions einheitlich abgefangen und verarbeitet. `TimeoutException` und `ServerErrorException` zeigen Fehlermeldungen wie in Abschnitt 1.1 beschrieben als Toast an. Eine `AuthException` bedeutet dagegen, dass die aktuelle Session nicht mehr gültig ist und der User sich neu anmelden muss. Daher wird in diesem Fall zur `LoginView` gewechselt und eine Fehlermeldung als Toast angezeigt.

Warum Exceptions für Fehlerbehandlung? Alternativ zu Exceptions könnten hier Results verwendet und die Fehler in jedem Fall einzeln behandelt werden. Da diese Fehler bei praktisch jedem Funktionsaufruf, der am Ende mit dem Server kommuniziert, entstehen können und die Fehlerbehandlung für jeden solchen Aufruf gleich ist, würde dies aber nur dazu führen, dass jeder solchen Aufruf in den ViewModels die Fehlerbehandlungslogik unnötig und fehleranfällig duplizieren muss. Exceptions erlauben es, die Fehlerbehandlung einheitlich für alle ViewModels zu gestalten, ohne, dass sich die eigentliche ViewModel Logik um diese kümmern muss.

3.3.5.3. ModelFacade

Die Schnittstelle wird von `Model` (siehe Abschnitt 3.3.5.9) implementiert.

⁶<https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-flow/>

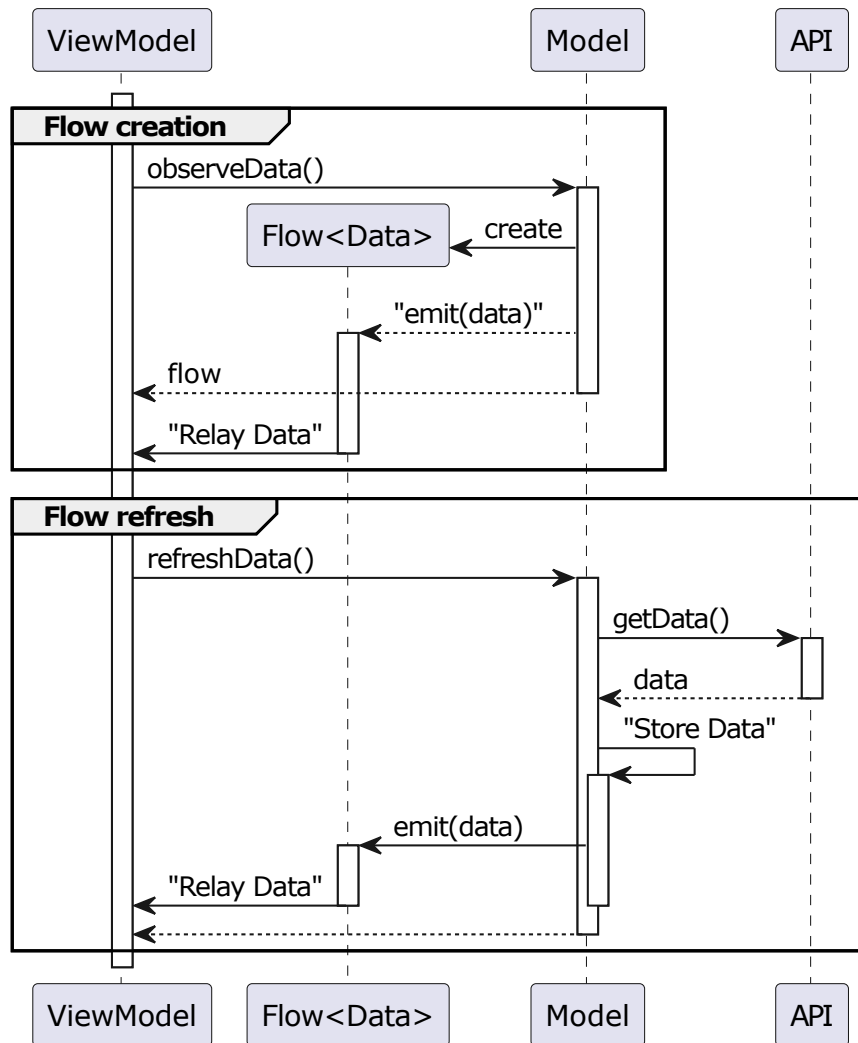


Abbildung 3.24.: Observe-Refresh mit Flows

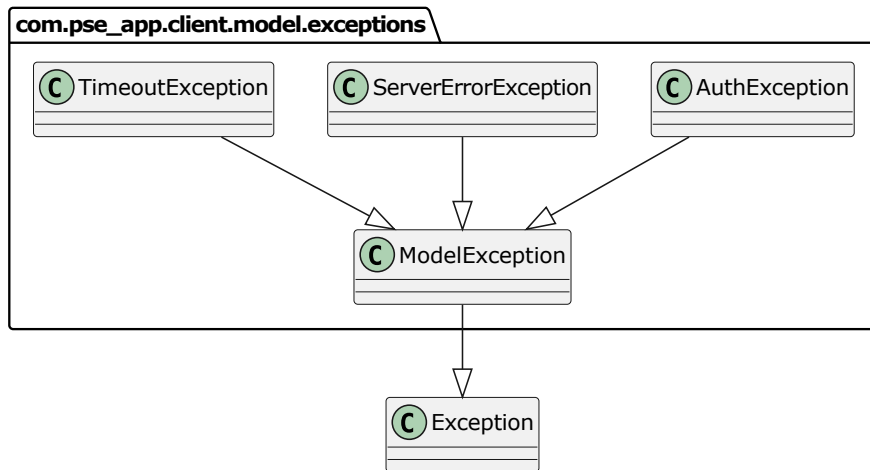


Abbildung 3.25.: Model Exceptions

Allgemeine Methoden

getCurrency(): Currency

Gibt die aktuell für die Benutzeroberfläche zu verwendende Währung zurück.

getAllSettleRecommendations(): List<SettleRecommendation>

Ruft durch Serveranfrage eine Liste von Beträgen zum Abgleichen ab. Jedes Objekt der zurückgegebenen Liste beschreibt einen möglichen Abgleich mit einem anderen Nutzer und enthält nach Gruppe aufgeschlüsselt den optimalen Ausgleichsbetrag. Siehe dazu auch 3.28.

Beobachtbare Flows

observeLocalUser(): Flow<User?>

Gibt einen Flow auf den aktiven Nutzer zurück. Dieser wird als *null* angegeben, falls die App abgemeldet ist.

observeGroups(): Flow<List<Group>>

Gibt einen Flow auf die Liste der Gruppen, in denen der aktive Nutzer Mitglied ist, zurück.

Refresh Operationen

refreshLocalUser()

Erneuert die Identität und den Daten des aktiven Nutzers.

refreshUsers(users: List<User>?=null)

Erneuert die Daten einer Menge an Nutzern. Falls keine Menge an Nutzern übergeben wird, werden alle dem Model bekannten Nutzer erneuert.

refreshMembership()

Erneuert die Identität und direkt mit der Gruppe verbundenen Daten der Gruppen in denen der Nutzer Mitglied ist. Nicht erneuert werden also die Mitglieder der Gruppe, sowie die Buchungen der Gruppe.

Zustandsändernde Operationen**login(idToken: String)**

Ruft die entsprechende Methode der RemoteAPI auf (siehe Abschnitt 3.3.7).

logout()

Ruft die entsprechende Methode der RemoteAPI auf und stellt sicher, dass der gespeicherte Modellzustand zurückgesetzt wird.

setUserDisplayName(name: String)

Ändert den Anzeigenamen des aktiven Nutzers. <F3>

deactivateLocalUser()

Diese Methode deaktiviert den angemeldeten Nutzer. Bei erfolgreichem Ausgang ist danach auch die Sitzung gelöscht. <F4>

createGroup(name: String): Group

Erstellt eine neue Gruppe mit dem gegebenen Namen und dem aktiven Nutzer als Mitglied. Die Daten zu dieser Gruppe sind nach Erfolg aktuell. <F5>

joinGroup(inviteToken: String): Group

Versucht den aktiven Nutzer einer Gruppe hinzuzufügen. <F8>

postTransactions(Map<Group, Transaction>)

Für eine Menge an Gruppen wird jeweils eine neue Buchung angefügt. <F11> und <F13>

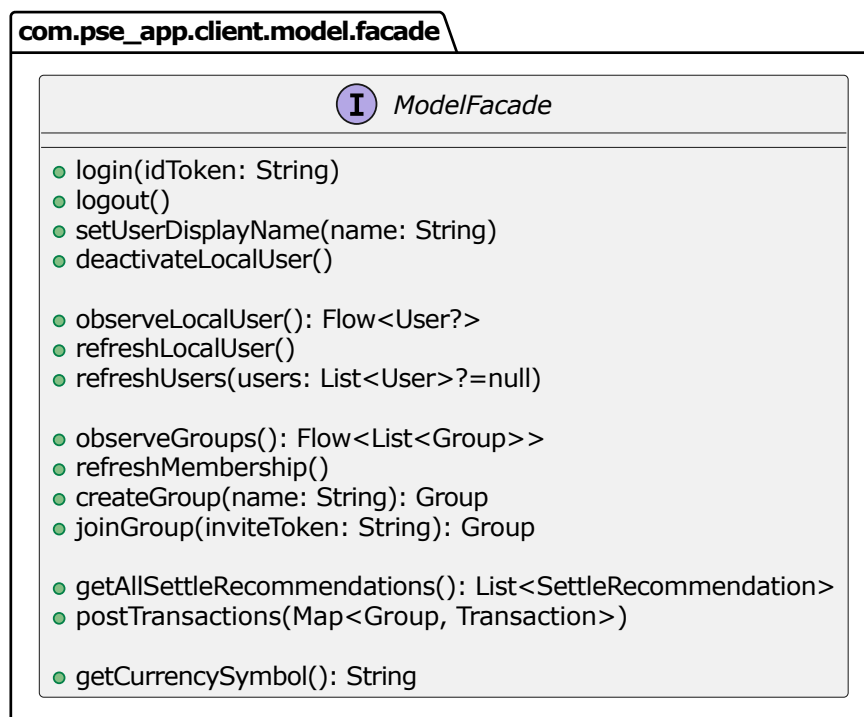


Abbildung 3.26.: ModelFacade Schnittstelle

3.3.5.4. User

Die User Schnittstelle stellt einen spezifischen Nutzer dar, bzw. die dem aktiven Nutzer über diesen sichtbaren Informationen. Siehe dazu auch Abbildung 3.27. Die Schnittstelle wird von 3.3.5.7 RepoUser implementiert.

User bietet Sichten auf die Daten dieses Nutzers. Eine refresh Methode fehlt, da User über ModelFacade.refreshUsers aktualisiert werden.

displayName: Flow<String?>

Flow auf den Anzeigenamen des Nutzers.

profilePicture: Flow<String?>

Flow auf die Url zum Profilbild des Nutzers.

3.3.5.5. Group

Die Group Schnittstelle stellt eine Gruppe dar und ist in Abbildung 3.27 dargestellt. Die Schnittstelle wird von 3.3.5.8 RepoGroup implementiert.

Beobachtbare Flows

displayName: Flow<String?>

Flow auf den Anzeigenamen der Gruppe.

members: Flow<List<User>>

Flow auf die Mitgliederliste der Gruppe.

inviteUrl: Flow<String?>

Flow auf den Einladungslink der Gruppe.

transactions: Flow<List<Transaction>>

Flow auf die mit der Gruppe verbundenen Transaktionen.

Refresh Operationen

refresh()

Erneuert alle Daten der Gruppe, sowie die Identitäten und Daten, der verbundenen Mitglieder und Transaktionen.

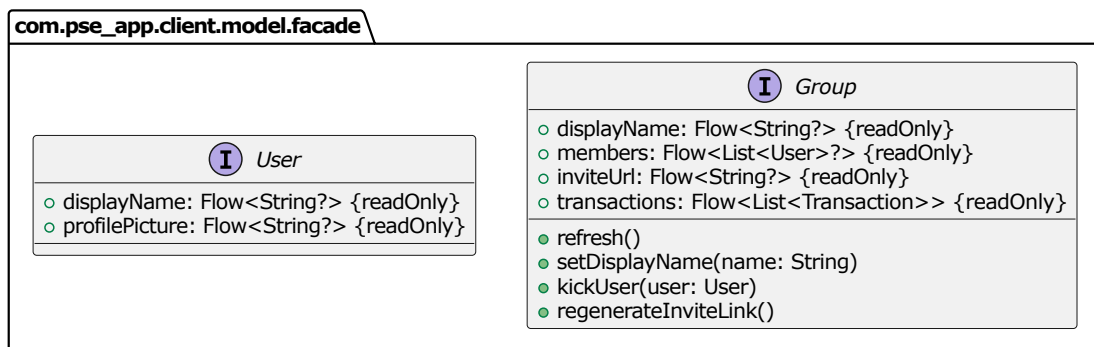


Abbildung 3.27.: User und Group Schnittstelle

Zustandsändernde Operationen

setDisplayName(name: String)

Ändert den Anzeigenamen der Gruppe. ⟨F9⟩

kickUser(user: User)

Entfernt ein Mitglied aus der Gruppe. ⟨F10⟩

regenerateInviteLink()

Frägt einen neuen Einladungslink für die Gruppe an. ⟨F7⟩

3.3.5.6. Datenklassen

Für die Darstellung von vorgeschlagenen Abgleichen wird die SettleRecommendation Datenklasse verwendet und für Transaktionen die Transaction Datenklasse.

Siehe dazu Abbildung 3.28

3.3.5.7. RepoUser

RepoUser implementiert User indem bei Zustands- und Änderungsanfragen die passenden Repository Methoden mit der Nutzer ID aufgerufen werden. Initialisiert wird der RepoUser deshalb mit einer Nutzer ID und Repository. `equals`⁷ und `hashCode`⁸ sollen so implementiert werden, das zwei RepoUser Objekte mit dem gleichen Bezeichner beim Vergleich als gleich resultieren.

⁷<https://kotlinlang.org/docs/equality.html>

⁸<https://kotlinlang.org/api/core/kotlin-stdlib/kotlin/-any/hash-code.html>

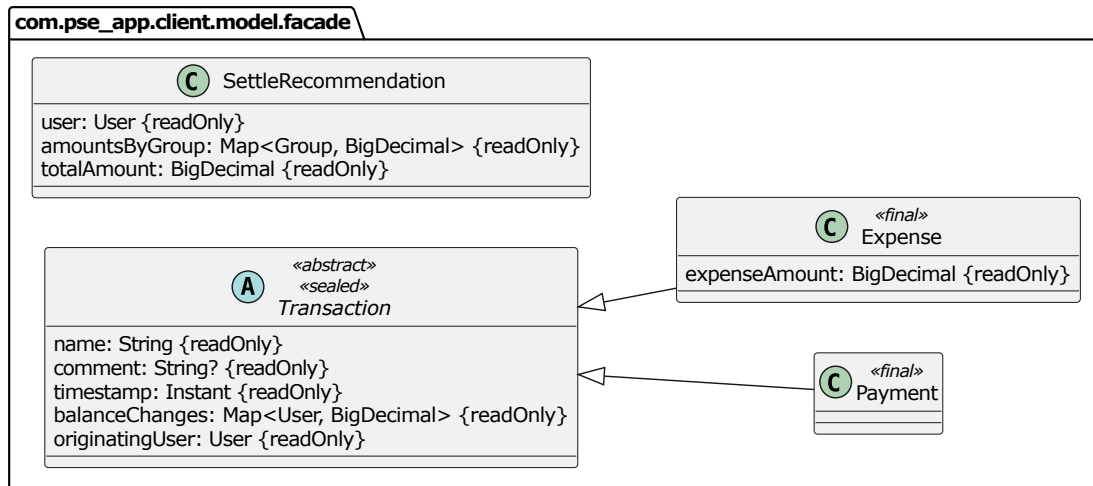


Abbildung 3.28.: SettleRecommendation und Transaction Datenklassen

3.3.5.8. RepoGroup

Implementiert Group indem bei Zustands- und Änderungsanfragen die passenden Repository Methoden mit der Gruppen ID aufgerufen werden. Initialisiert wird der RepoUser deshalb mit einer Gruppen ID und Repository. `equals`⁹ und `hashCode`¹⁰ sollen so implementiert werden, dass zwei RepoGroup Objekte mit dem gleichen Bezeichner beim Vergleich als gleich resultieren.

3.3.5.9. Model (Klasse)

Die ModelFacade Schnittstelle wird von der Model Klasse implementiert. Diese wird mit den Repositories, einer RemoteAPI und Preferences initialisiert. Die spezifische RemoteAPI Implementierung kann mittels dependency Injection flexibel ausgewählt werden. Preferences wird hingegen zur Persistierung von Key-Value Paaren verwendet.

⁹<https://kotlinlang.org/docs/equality.html>

¹⁰<https://kotlinlang.org/api/core/kotlin-stdlib/kotlin/-any/hash-code.html>

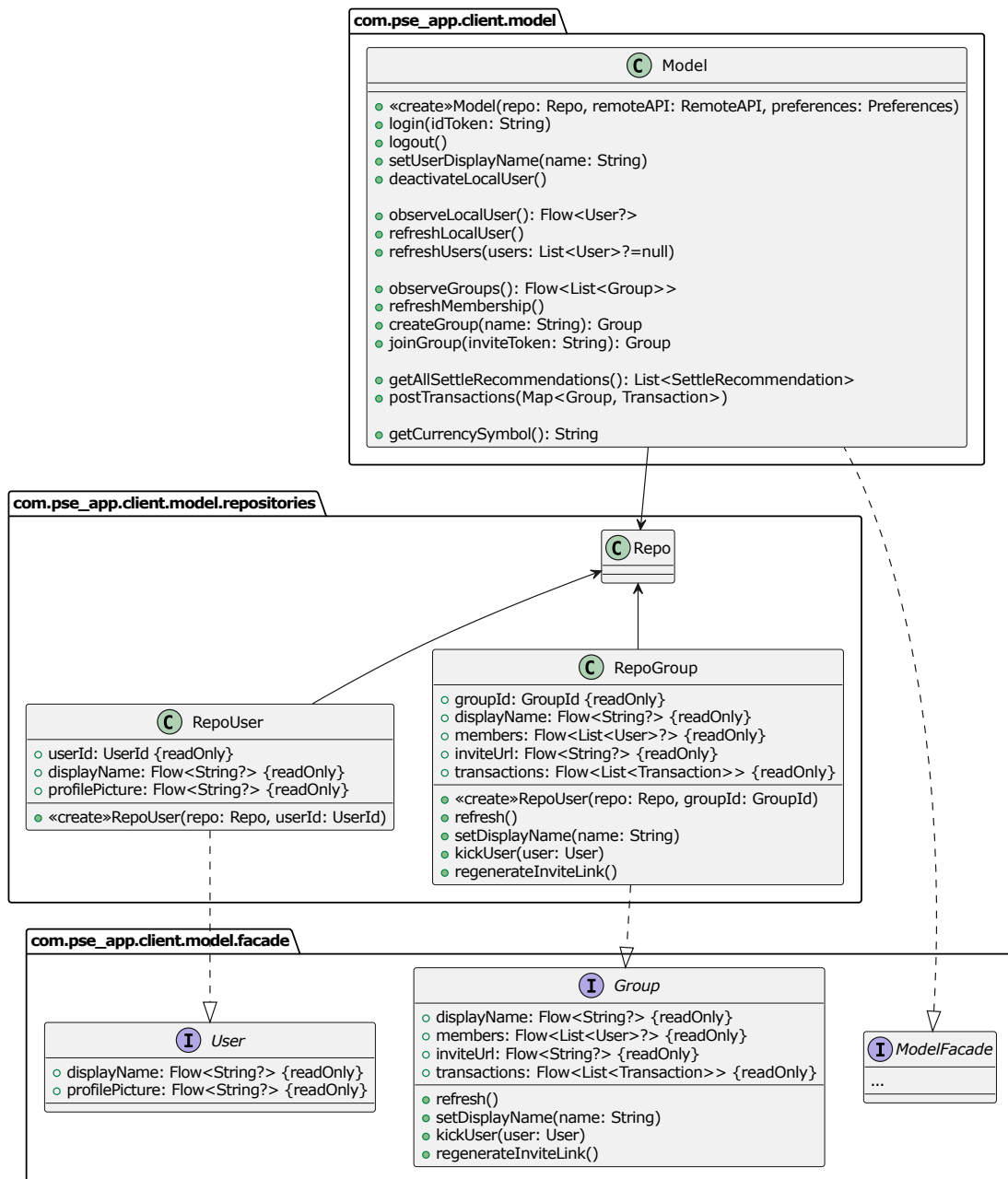


Abbildung 3.29.: Implementierung der Fassade

3.3.6. Model Repositories

Die Repositories Komponente wird im Model von den Klassen UserRepo, GroupRepo und TransactionRepo implementiert. Diese sind entsprechend ihrer Benennung für unterschiedliche Teile des Systems zuständig.

Die Repo Klasse bündelt die drei Repositories zusammen und wird von den RepoUser und RepoGroup Klassen verwendet. Alle Repositories werden mit der RemoteAPI initialisiert, damit sie mit dem Server kommunizieren können. Als zusätzlichen Parameter erhalten sie Repo selbst, da manche Operationen eventuell Methoden in anderen Repositories benötigen.

Siehe dazu auch Abbildung 3.30.

3.3.6.1. UserRepo

Die UserRepo Klasse bietet Methoden zum Abfragen und Ändern der Accounts.

Zustand

observeLocalUser(): Flow<UserData?>

Ein Flow auf die Daten des aktiven Accounts.

refreshLocalUser()

Erneuert die Daten des angemeldeten Accounts.

observeUser(userId: UserId): Flow<UserData?>

Ein Flow auf die Daten von einem Nutzer.

refreshUsers(userIds: List<UserId>? = null)

Aktualisiert die Daten von einer Menge an Nutzern.

Operationen

setDisplayname(name: String)

Der Name des angemeldeten Nutzers wird auf den gegebenen Namen geändert.

3.3.6.2. GroupRepo

Die GroupFacade Klasse ist für die Verwaltung von Nutzergruppen verantwortlich.

Zustand

observeAllGroupShort(): Flow<List<GroupShortData>>

Ein Flow auf die direkten Daten von allen Gruppen in denen der Nutzer Mitglied ist.

refreshAllGroupShort(): List<GroupShortData>

Aktualisiert die Kurzbeschreibungen von allen Gruppen in denen der Nutzer ist und gibt diese zurück.

observeGroup(groupId: GroupId): Flow<GroupFullData>

Ein Flow auf alle mit einer bestimmten Gruppe verbundenen Daten.

refreshGroup(groupId: GroupId)

Aktualisiert alle mit einer bestimmten Gruppe verbundenen Daten.

Operationen

createGroup(name: String)

Erstellt eine neue Gruppe mit dem gegebenen Namen und dem aktiven Nutzer als Mitglied. Die Daten zu dieser Gruppe sind nach Erfolg aktuell.

joinGroup(inviteToken: String)

Versucht den aktiven Nutzer einer Gruppe hinzuzufügen.

regenerateInviteLink(groupId: GroupId)

Generiert einen neuen Einladungslink für die Gruppe.

kickUser(groupId: GroupId, userId: UserId)

Versucht einen Nutzer aus einer Gruppe zu entfernen.

setDisplayname(groupId: GroupId, name: String)

Ändert den Anzeigenamen der Gruppe auf den gelieferten Namen.

3.3.6.3. TransactionRepo

Die TransactionRepo Klasse bietet Methoden zum Abfragen und hinzufügen von Buchungen.

observeTransactions(groupId: GroupId): Flow<List<TransactionData>>

Ein Flow auf die Liste der Transaktionen in einer Gruppe.

refreshTransactions(groupId: GroupId)

Aktualisiert die Liste an Transaktionen für eine Gruppe.

getAllSettleRecommendations(): List<SettleRecommendationData>

Berechnet optimale Abgleichbeträge je fremdem Nutzer und je Gruppe die der aktive Nutzer mit dem fremden Nutzer teilt.

postTransactions(Map<GroupId, TransactionData>)

Für eine Menge an Gruppen wird jeweils eine neue Buchung angefügt.

3.3.6.4. Datenklassen

Für den Datenaustausch innerhalb des Models definieren wir eine Menge an Datenklassen. Siehe dazu auch Abbildung 3.31.

UserData

Die Daten eines Nutzers. Enthält Bezeichner, Namen und Profilbild URL.

GroupShortData

Reduzierte Daten einer Gruppe. Enthält Bezeichner, Namen, Einladungslink und Saldo des aktiven Nutzers.

GroupFullData

Volle Daten einer Gruppe. Enthält zusätzlich zu GroupShortData auch die Bezeichner und Saldi aller Gruppenmitglieder.

TransactionData

Die Daten einer Transaktion. Enthält Namen, Kommentar, Timestamp, Author, Saldoänderungen und potentiell einen Ausgabebetrag.

SettleRecommendationData

Abgleichsvorschlag. Enthält eine Nutzer ID, vorgeschlagene Abgleichsbeträge pro mit dem Nutzer geteilt Gruppe und eine Summe dieser Beträge.

3.3.7. Model Data Layer

3.3.7.1. API Wrapper

Die REST-API des Servers wird über die RemoteAPI Schnittstelle benutzt. Diese Schnittstelle enthält bis auf den /v1/login Endpunkt alle REST Endpunkte als Methoden. Dabei werden die Response-Body DTOs der Endpunkte im Response Typen gewrappt, welcher zusätzlich einen HTTP Status Code mitteilt. Die Endpunkte sind bereits in Abschnitt 2.4.2.2 beschrieben. Die Details der HTTP Anfragen werden so vor dem Benutzer der API, nämlich der App Logik, versteckt.

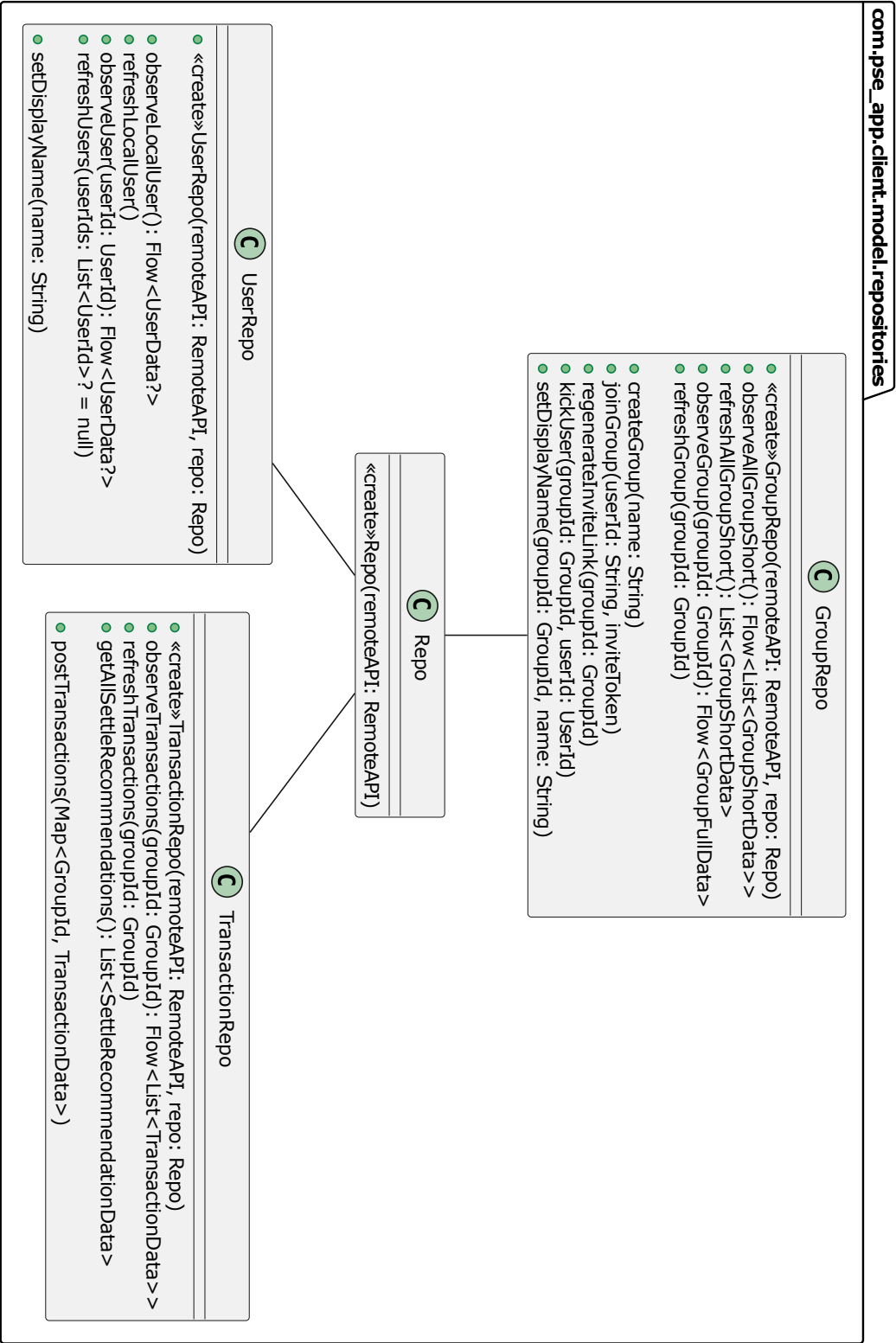


Abbildung 3.30.: Repositories

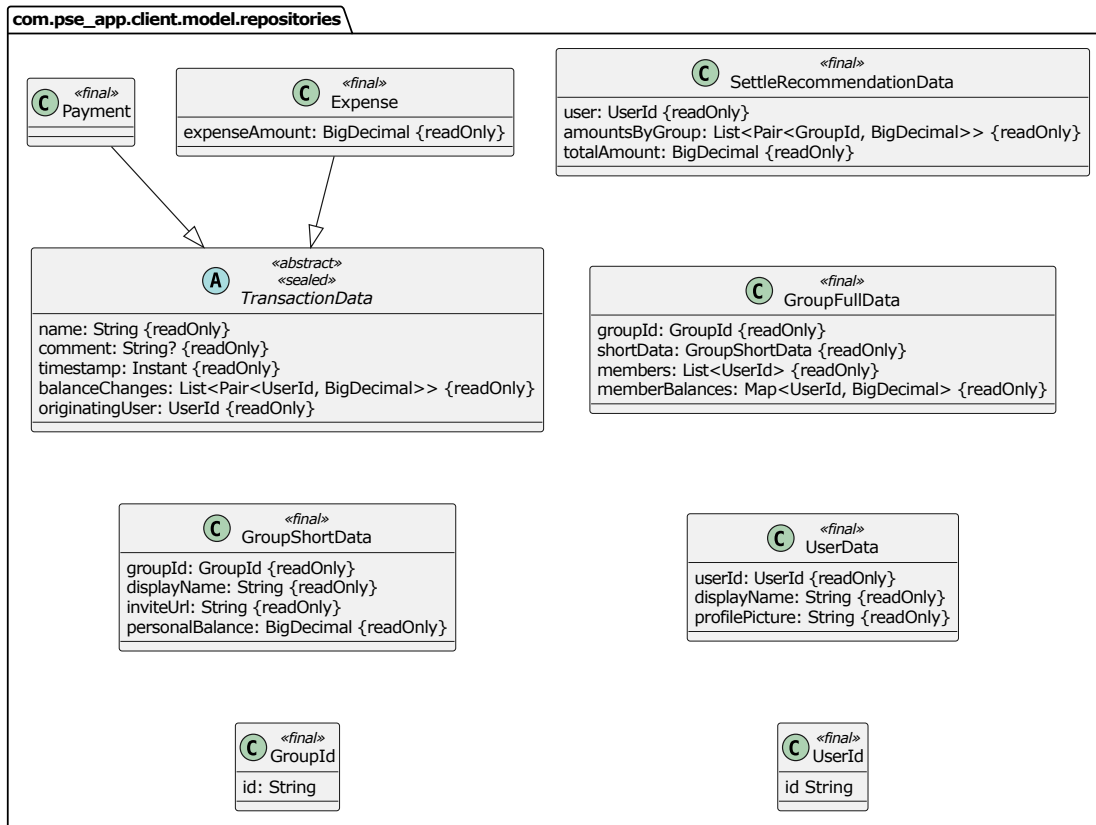


Abbildung 3.31.: Model-interne Datenklassen

Zusätzliche Methoden

login(idToken: String): Response<Unit>

Fragt neue Tokens beim Login Endpunkt des Servers an mit dem gegebenen ID Token. Die neuen Tokens werden persistiert und die alte Session gelöscht.

logout()

Führt die Abmeldung aus. Persistierte Session Daten werden gelöscht.

Implementiert wird RemoteAPI von RemoteClient. Der RemoteClient erhält zur Initialisierung eine Server-URL, einen Timeout Wert und einen SessionStore. Access und Refresh Token werden von RemoteClient verwaltet (inklusive automatischem Token Refresh bei Anfragen) und im SessionStore persistiert.

3.3.7.2. SessionStore

Die SessionStore Schnittstelle dient dem Persistieren von Sessions, inklusive Refresh und Access Tokens. Sie wird von der SessionPreference Klasse implementiert, welche Preferences zum Persistieren verwendet.

3.3.8. Preferences

Um Informationen, wie etwa die Session, über den Neustart der App hinaus zu persistieren wird die Preferences Klasse verwendet. Dazu bietet Preferences einfache get und set Methoden auf Key-Value Paare (Siehe Abbildung 3.33). Preferences verwendet dann für die Persistierung die Jetpack DataStore Library.

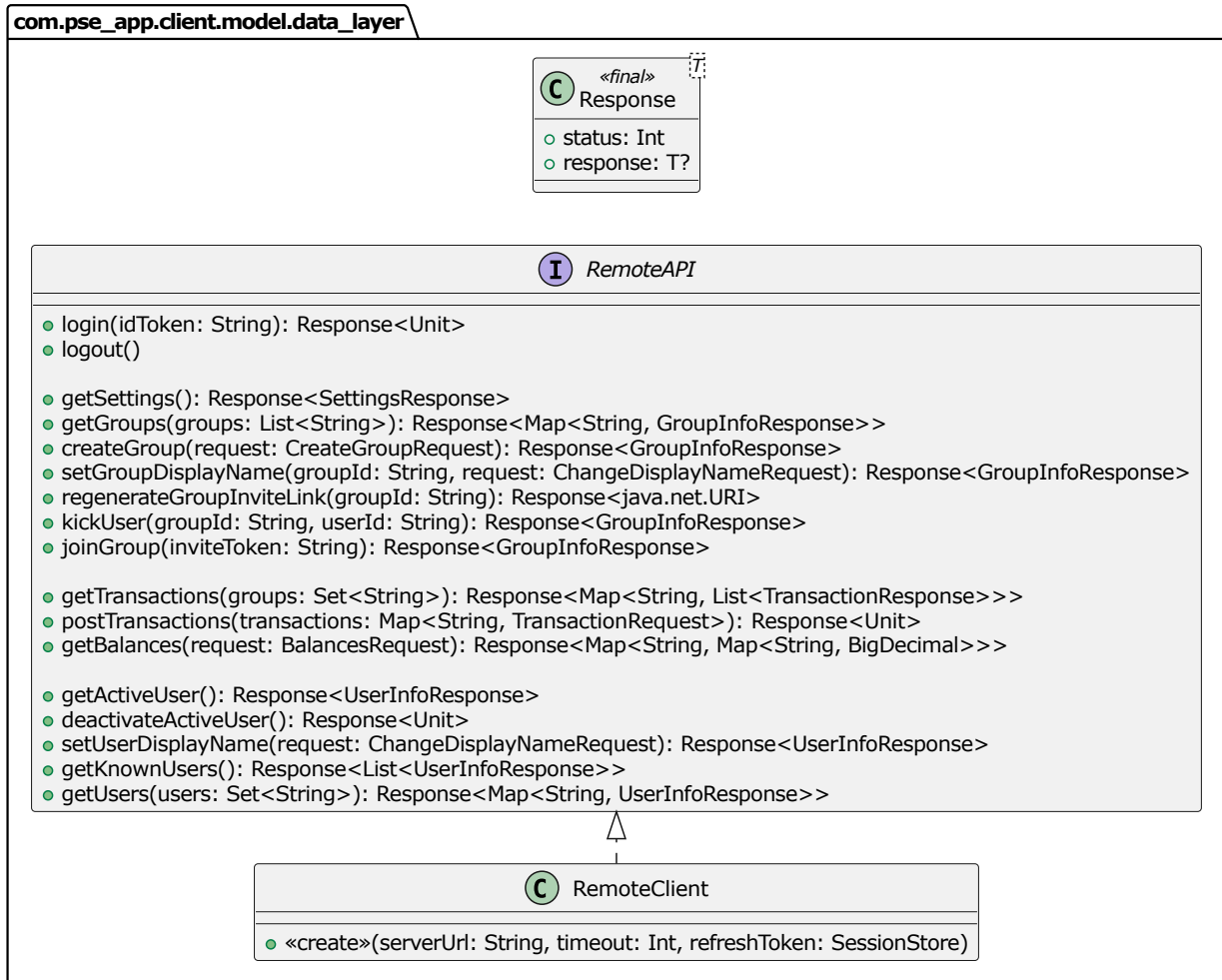


Abbildung 3.32.: RemoteAPI Schnittstelle

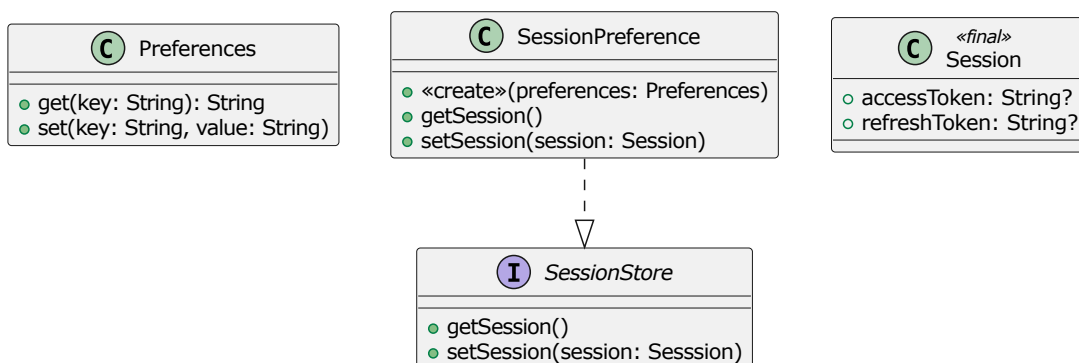


Abbildung 3.33.: Detailentwurf von Preferences und SessionStore

4. Glossar

<i>Client</i>	Die Android App, die als Teil des Systems auf dem Android Gerät des Endnutzers läuft.
<i>View</i>	Mit View wird entweder die View Komponente des Clients bezeichnet, oder die Composables aus denen die Komponente besteht.
<i>Nutzer ID</i>	Mit Nutzer ID wird der eindeutige Bezeichner eines Nutzeraccounts bezeichnet.
<i>Gruppen ID</i>	Mit Gruppen ID wird der eindeutige Bezeichner einer Gruppe bezeichnet.
<i>Identity Provider</i>	Externer Dienst, zum Beispiel Google, welcher Nutzern ihre Identität ausweist.
<i>OIDC</i>	OpenID Connect, der Standard welcher für die Kommunikation mit dem Identity Provider verwendet wird.

A. Anhang

A.1. SQL-Schema Definition

```
1 BEGIN;
2
3 CREATE TABLE users (
4     id TEXT PRIMARY KEY NOT NULL,
5     last_login TIMESTAMPTZ NOT NULL
6 );
7
8 CREATE TABLE active_users (
9     id TEXT PRIMARY KEY NOT NULL REFERENCES users(id),
10    display_name TEXT NOT NULL,
11    profile_picture_url TEXT
12 );
13
14 CREATE TABLE groups (
15     id UUID PRIMARY KEY NOT NULL,
16     invite_token TEXT UNIQUE NOT NULL,
17     display_name TEXT NOT NULL
18 );
19
20 CREATE TABLE membership (
21     user_id TEXT NOT NULL REFERENCES users(id),
22     group_id UUID NOT NULL REFERENCES groups(id),
23     PRIMARY KEY (user_id, group_id)
24 );
25
26 CREATE TABLE transactions (
27     id UUID PRIMARY KEY NOT NULL DEFAULT gen_random_uuid(),
28     group_id UUID NOT NULL REFERENCES groups(id),
29     name TEXT NOT NULL,
30     comment TEXT,
31     timestamp TIMESTAMPTZ NOT NULL DEFAULT NOW(),
32     originating_user TEXT NOT NULL REFERENCES users(id),
33     expense_total MONEY
34 );
35
36 CREATE TABLE balance_changes (
```

```
37     transaction_id UUID NOT NULL REFERENCES transactions(id),
38     user_id TEXT NOT NULL REFERENCES users(id),
39     amount MONEY NOT NULL,
40     PRIMARY KEY (transaction_id, user_id)
41 );
42
43 CREATE TABLE refresh_tokens (
44     user_id TEXT NOT NULL REFERENCES users(id),
45     token TEXT PRIMARY KEY NOT NULL
46 );
47
48 CREATE INDEX groups_invite_token ON groups USING HASH(invite_token);
49 CREATE INDEX membership_users ON membership USING HASH(user_id);
50 CREATE INDEX membership_groups ON membership USING HASH(group_id);
51 CREATE INDEX transactions_groups ON transactions USING HASH(group_id);
52 CREATE INDEX balance_changes_transactions ON balance_changes USING HASH(transaction_id);
53 CREATE INDEX refresh_tokens_users ON refresh_tokens USING HASH(user_id);
54
55 COMMIT;
```

A.2. API-Definition

API v1 OAS 3.1

default

POST /v1/login

Convert an OpenID Connect ID token into a access and refresh token

Parameters

Try it out

No parameters

Request body required

application/json

Example Value Schema

com.pse_app.LoginRequest

Collapse all (object | object)

Any of (object | object)

#0 IdToken

Collapse all object

idToken* string

#1 RefreshToken

Collapse all object

refreshToken* string

Responses

Code	Description	Links
200	access and refresh token retrieved successfully <div>Media type<div>application/json</div>Controls Accept header.</div>	No links

Code	Description	Links
	Example Value Schema	
	com.pse_app.LoginResponse ^ Collapse all object accessToken* string refreshToken* string	
400	Invalid ID token	No links

GET /v1/settings ^

Retrieve generic information about the server configuration. This includes OpenID information.

Parameters Try it out

No parameters

Responses

Code	Description	Links
200	Information about the server configuration retrieved successfully. Media type application/json <small>Controls Accept header.</small> Example Value Schema	No links
	com.pse_app.SettingsResponse ^ Collapse all object clientId* string discoveryUri* string currency string	

PUT /v1/groups ^

Create a new group

Parameters

Try it out

No parameters

Request body

application/json

Example Value

Schema

com.pse_app.CreateGroupRequest

^

Collapse all

object

displayName*

string

Responses

Code	Description	Links
201	<div>Group created successfully</div> <div>Media type</div> <div><div>application/json</div></div> <div>Controls Accept header.</div> <div>Example Value</div> <div>Schema</div> <div><div>com.pse_app.GroupInfoResponse</div><div>^</div><div>Collapse all</div><div>object</div><div><div>displayName*</div><div>string</div><div>id*</div><div>string</div><div>inviteUrl*</div><div>string</div><div>members*</div><div>></div><div>Expand all</div><div>array<string></div></div></div>	No links
400	Invalid group info	No links
401	User not logged in	No links

GET

/groups

^

Retrieve the group info of some groups

Parameters

Try it out

No parameters

Request body required

application/json

Groups to get the group info of

Example Value Schema

Set<GroupId> ^ Collapse all **array<string>**
Items **string**

Responses

Code	Description	Links
200	<p>Group infos retrieved successfully</p> <p>Media type</p> <p>application/json</p> <p><small>Controls Accept header.</small></p> <p>Example Value Schema</p> <p>Map<GroupId,GroupInfoResponse> ^ Collapse all object Additional properties > Expand all object</p>	No links
401	User not logged in	No links
404	Group not found	No links

PATCH /v1/groups/{groupId}/displayName



Change the display name of a group

Parameters

Try it out

Name	Description
<div><div>groupId * required</div><div>string</div><div>(path)</div></div>	<div>groupId</div>

Request body required

application/json

Example ValueSchema

com.pse_app.ChangeDisplayNameRequest ^ Collapse all object

displayName* string

Responses

Code	Description	Links
200	<div>Display name of group changed successfully</div> <div>Media type<div>application/json</div></div> <div>Controls Accept header.</div> <div>Example ValueSchema</div> <div><div>com.pse_app.GroupInfoResponse ^ Collapse all object</div><div><div>displayName* string</div><div>id* string</div><div>inviteUrl* string</div><div>members* > Expand all array<string></div></div></div>	No links
400	Invalid display name	No links
401	User not logged in	No links
404	Group not found	No links

POST

/groups/{groupId}/regenerateInviteLink

⌵

Regenerate the invite link of a group

Parameters

Try it out

Name	Description
groupId * required string (path)	<input type="text" value="groupId"/>

Responses

Code	Description	Links
200	<div>Group invite link regenerated successfully</div> <div>Media type</div> <div><div>application/json</div></div> <div>Controls Accept header.</div> <div>Example Value Schema</div> <div>java.net.URI ⌵ Collapse all object</div>	No links
401	User not logged in	No links
404	Group not found	No links

POST

/v1/groups/{groupId}/kick/{userId}

⌵

Kick a user from a group

Parameters

Try it out

Name

Description

groupId * required

string

(path)

string

(path)

userId * required

string

(path)

string

(path)

groupId

userId

Responses

Code	Description	Links
200	<div><div>Kicked user successfully</div><div>Media type</div><div><div>application/json</div></div><div>Controls Accept header.</div><div>Example Value Schema</div><div><div>com.pse_app.GroupInfoResponse ^ Collapse all object</div><div><div>displayName* string</div><div>id* string</div><div>inviteUrl* string</div><div>members* > Expand all array<string></div></div></div></div> <div>No links</div>	No links
401	User attempting to kick not logged in	No links
404	Group or user not found	No links
422	Balance of user to be kicked not zero	No links

GET /v1/transactions

Retrieve all transactions in some groups

Parameters

Try it out

No parameters

Request body required

application/json

Example Value

Schema

Set<GroupId> ^ Collapse all array<string>

Items string

Responses

Code	Description	Links
200	<div>Transactions retrieved successfully</div> <div>Media type</div> <div>application/json</div> <div>Controls Accept header.</div> <div>Example Value</div> <div>Schema</div> <div><div>Map<GroupId,List<TransactionResponse>> ^</div><div>Collapse all object</div><div>Additional properties > Expand all array<(object object)></div></div>	

 No links || 401 | User not logged in | No links |
| 404 | Group not found | No links |

PUT

/v1/transactions

Post transactions in groups

Parameters

Try it out

No parameters

Request body required

application/json

Transactions to post

Example Value Schema

Map<GroupId,TransactionRequest> ^ Collapse all object

Additional properties > Expand all (object | object)

Responses

Code	Description	Links
200	Transactions posted successfully	No links
400	Invalid transaction	No links
401	User not logged in	No links
404	Group not found	No links

GET /v1/balances ^

Retrieve the balances of users from common groups

Parameters

Try it out

No parameters

Request body required

application/json

Example Value Schema

com.pse_app.BalancesRequest ^ Collapse all **object**

groups > Expand all **null | array<string>**

users > Expand all **null | array<string>**

Responses

Code	Description	Links
200	Balances retrieved successfully	No links
	Media type	
	application/json	
	Controls Accept header.	
	Example Value Schema	
	Map<UserId,Map<GroupId,BigDecimal>> ^ Collapse all object	
	Additional properties > Expand all object	
401	User not logged in	No links
404	Group or user not found	No links

GET /v1/me ^

Retrieve user info

Parameters

Try it out

No parameters

Responses

Code	Description	Links
200	User info retrieved successfully	No links
	Media type <div>application/json</div> <div>Controls Accept header.</div> <div>Example Value Schema</div> <div><div>com.pse_app.UserInfoResponse ^ Collapse all object</div><div>displayName* string</div><div>groups* > Expand all array<string></div><div>id* string</div><div>profilePicture null string</div></div>	
401	User not logged in	No links

DELETE	/v1/me	
Deactivate user		
Parameters		Try it out
No parameters		
Responses		
Code	Description	Links
200	User deactivated successfully	No links
401	User not logged in	No links

PATCH	/v1/me/displayName	

Change the display name of the user

Parameters

[Try it out](#)

No parameters

Request body required

[application/json](#)

Example Value [Schema](#)

com.pse_app.ChangeDisplayNameRequest ^ Collapse all **object**

```
displayName* string
```

Responses

Code	Description	Links
200	User deactivated successfully	No links
	<div>Media type</div> <div>application/json</div> <div>Controls Accept header.</div> <div>Example Value Schema</div> <div>com.pse_app.UserInfoResponse ^ Collapse all object</div> <pre>displayName* string groups* ^ Collapse all array<string> Items string id* string profilePicture null string</pre>	
400	Invalid display name	No links
401	User not logged in	No links

GET /v1/usersInCommonGroups



Retrieve the user info of all users in common groups

Parameters

Try it out

No parameters

Responses

Code	Description	Links
200	<div>User infos retrieved successfully</div> <div>Media type</div> <div>application/json</div> <div>Controls Accept header.</div> <div>Example Value Schema</div> <div>List<UserInfoResponse> ^ Collapse all array<object></div> <div>Items > Expand all object</div>	No links
401	User not logged in	No links

GET

/v1/users

Retrieve the user info of some users

Parameters

Try it out

No parameters

Request body required

application/json

Users to retrieve the user info of

Example Value Schema

Set<UserId> ^ Collapse all **array<string>**
Items **string**

Responses

Code	Description	Links
200	Users infos retrieved successfully <div>Media type application/json <small>Controls Accept header.</small> Example Value Schema Map<UserId,UserInfoResponse> ^ Collapse all object Additional properties > Expand all object</div>	No links
401	User not logged in	No links
404	User not found	No links

POST /v1/join/{inviteToken} ^

Join a group via its invite token

Parameters Try it out

Name	Description
inviteToken * required string (path)	<input type="text" value="inviteToken"/>

Responses

Code	Description	Links
200	<div>Joined group successfully</div> <div>Media type</div> <div><div>application/json</div></div> <div>Controls Accept header.</div> <div>Example Value Schema</div> <div><div>com.pse_app.GroupInfoResponse</div> ^ Collapse all object<div><div>displayName* string</div><div>id* string</div><div>inviteUrl* string</div><div>members* > Expand all array<string></div></div></div>	No links
400	Invalid invite token	No links
401	User not logged in	No links
410	Invite token has been revoked	No links

GET /v1/invite/{inviteToken}

Retrieve a landing page displaying instructions to install the app and then join the group

Parameters

Try it out

Name	Description
<div>inviteToken * required</div> <div>string</div> <div>(path)</div>	<div>inviteToken</div>

Responses

Code	Description	Links
------	-------------	-------

Code	Description	Links
200	Landing page retrieved successfully	No links

Schemas



```
IdToken ^ Collapse all object
  idToken* string

RefreshToken ^ Collapse all object
  refreshToken* string

LoginRequest ^ Collapse all (object | object)
  Any of ^ Collapse all (object | object)
    #0 IdToken ^ Collapse all object
      idToken* string
    #1 RefreshToken ^ Collapse all object
      refreshToken* string

LoginResponse ^ Collapse all object
  accessToken* string
  refreshToken* string

SettingsResponse ^ Collapse all object
  clientId* string
  discoveryUri* string
  currency string

CreateGroupRequest ^ Collapse all object
  displayName* string

GroupInfoResponse ^ Collapse all object
```

```

displayName* string
id* string
inviteUrl* string
members* ^ Collapse all array<string>
  Items string

```

ChangeDisplayNameRequest ^ Collapse all object

```

displayName* string

```

URI ^ Collapse all object

TransactionRequest ^ Collapse all (object | object)

```

Any of ^ Collapse all (object | object)
  #0 ExpenseRequest ^ Collapse all object
    balanceChanges* ^ Collapse all object
      Additional properties string
    comment null | string
    expenseAmount* string
    name* string
  #1 PaymentRequest ^ Collapse all object
    balanceChanges* ^ Collapse all object
      Additional properties string
    comment null | string
    name* string

```

ExpenseRequest ^ Collapse all object

```

balanceChanges* ^ Collapse all object
  Additional properties string
comment null | string
expenseAmount* string
name* string

```

PaymentRequest ^ Collapse all object

```

balanceChanges* ^ Collapse all object
  Additional properties string
comment null | string
name* string

```

TransactionResponse ^ Collapse all (object | object)

Any of ^ Collapse all (object | object)

#0 ExpenseResponse ^ Collapse all object

balanceChanges* ^ Collapse all object

Additional properties string

comment null | string

expenseAmount* string

group* string

name* string

originatingUser* string

timestamp* string

#1 PaymentResponse ^ Collapse all object

balanceChanges* ^ Collapse all object

Additional properties string

comment null | string

group* string

name* string

originatingUser* string

timestamp* string

ExpenseResponse ^ Collapse all object

balanceChanges* ^ Collapse all object

Additional properties string

comment null | string

expenseAmount* string

group* string

name* string

originatingUser* string

timestamp* string

PaymentResponse ^ Collapse all object

balanceChanges* ^ Collapse all object

Additional properties string

comment null | string

group* string

name* string

originatingUser* string

timestamp* string

```
BalancesRequest ^ Collapse all object  
  groups ^ Collapse all null | array<string>  
    Items string  
  users ^ Collapse all null | array<string>  
    Items string
```

```
UserInfoResponse ^ Collapse all object  
  displayName* string  
  groups* ^ Collapse all array<string>  
    Items string  
  id* string  
  profilePicture null | string
```