

Entwurfsdokumentation

PSEKochbuch Exzellenzkoch

Magnus Bühler Wolf H. Lauppe Pascal Maier
Thomas Weidmann Lea Strauch

Wintersemester 2019/2020, 22. Dezember 2019

Version	Autor	Datum	Status	Kommentar
1.0	PSE3	22.12.2019	Abgabe	Pflichtenheft v1.0 mit Ergänzung der Admin-Rolle

Tabelle 0.1: Versionshistorie

Inhaltsverzeichnis

1	Zielbestimmung	1
2	Architektur	2
2.1	Prämisse	2
2.2	App	3
2.2.1	Layered Architecture	3
2.2.2	Domain Layer	4
2.2.3	Data Layer	4
2.2.4	UserInterface Layer	5
2.2.5	Zusammenfassung	5
2.3	Server	5
2.3.1	Einleitung	5
2.3.2	Technologiestack	6
2.3.3	Struktur	6
2.4	Deployment	7
2.4.1	Server-Deployment	7
2.4.2	App-Deployment	7
2.5	Authentifizierung und Authorisierung	8
2.5.1	Adminrolle	8
2.6	Entwurfsentscheidungen	9
3	Interaktion zwischen Komponenten	10
3.1	Interaktion von App Komponenten	10
3.2	Interaktion von Serverkomponenten	11
3.3	App- und Server-Kommunikation	11
4	Paketeinteilung und Klasseninteraktion	12
4.1	App	13
4.1.1	User Interface Layer	13
4.1.2	View	13
4.1.3	ViewModel	20
4.1.4	Domain Layer	21
4.1.5	Data Layer	23
4.2	Server	23
4.2.1	Api und Model Pakete	23
4.2.2	Konfiguration- und Ausführungspakete	24
4.2.3	Kommunikation zwischen Controllerpaket und Servicepaket	25
4.2.4	Kommunikation zwischen Servicepaket und Daopakete	25

5	Klassenbeschreibung Userinterface	27
5.1	View	27
5.1.1	Toolbar und Appmenü	27
5.1.2	CreateRecipeFragment	29
5.1.3	RecipeListFragment	31
5.1.4	AddEditCommentFragment	32
5.1.5	DisplaySearchListFragment	33
5.1.6	UserSearchFragment	35
5.1.7	PublicRecipeSearchFragment	36
5.1.8	FeedFragment	38
5.1.9	ProfileDisplayFragment	39
5.1.10	ProfileEditFragment	41
5.1.11	RegistrationFragment	43
5.1.12	ChangePasswordFragment	45
5.1.13	LoginFragment	46
5.1.14	EditTagFragment	48
5.1.15	SearchWithTagsFragment	49
5.1.16	AdminFragment	51
5.1.17	ShoppingListDisplayFragment	53
5.1.18	FriendListFragment	54
5.1.19	GroupMemberListFragment	56
5.1.20	CreateGroupFragment	57
5.1.21	FriendGroupFragment	58
5.1.22	GroupFragment	60
5.1.23	RecipeDisplayFragment	63
5.1.24	FavouriteFragment	65
5.2	ViewModel	66
5.2.1	DisplaySearchListViewModel	66
5.2.2	CreateRecipeViewModel	67
5.2.3	RecipeListViewModel	68
5.2.4	AddEditCommentViewModel	69
5.2.5	UserSearchViewModel	70
5.2.6	PublicRecipeSearchViewModel	71
5.2.7	FeedViewModel	72
5.2.8	ProfileDisplayViewModel	73
5.2.9	ProfileEditViewModel	74
5.2.10	RegistrationViewModel	74
5.2.11	ChangePasswordViewModel	75
5.2.12	LoginViewModel	75
5.2.13	EditTagViewModel	76
5.2.14	SearchWithTagsViewModel	76
5.2.15	AdminViewModel	77
5.2.16	ShoppingListDisplayViewModel	78
5.2.17	FriendListViewModel	78
5.2.18	GroupMemberListViewModel	79
5.2.19	CreateGroupViewModel	79
5.2.20	FriendGroupViewModel	80
5.2.21	GroupViewModel	80
5.2.22	RecipeDisplayViewModel	81

5.2.23	FavouriteViewModel	82
6	Beschreibung Domain Layer Interfaces	83
6.1	Repository	83
6.2	Authentifizierung	83
6.3	ImportExport	84
7	Klassenbeschreibung Domain Layer	85
7.1	Domain Entities	85
7.1.1	User	85
7.1.2	Group	86
7.1.3	«Interface» Recipe	86
7.1.4	privateRecipe	87
7.1.5	publicRecipe	87
7.1.6	IngredientAmountList	88
7.1.7	IngredientChapter	88
7.1.8	IngredientAmount	89
7.1.9	Unit	89
7.1.10	Feed	89
7.1.11	Comment	90
7.2	Authentification	90
7.3	ImportExport	90
7.3.1	Export	91
7.3.2	Import	92
8	Klassenbeschreibung Data Layer	93
8.1	Repository	93
8.1.1	Modell für SQLite-Datenbank	93
8.1.2	Daoklassen für die Datenbank	97
8.2	Datenbank	101
8.2.1	ClientDatenbank	101
9	Klassenbeschreibung REST-Schnittstelle	106
9.0.1	Paging	106
9.0.2	Optionale Parameter	106
9.1	API/REST-Schnittstelle	107
9.1.1	UserApi	107
9.1.2	PublicRecipeApi	108
9.1.3	FriendRequestApi	110
9.1.4	GroupApi	112
9.1.5	FavouritesApi	115
9.1.6	AdminApi	115
9.1.7	CommentApi	117
9.2	DTOs	119
10	Klassenbeschreibung Server	121
10.1	Model und Dao	121
10.1.1	Modell der Datenbank für den Server	121
10.1.2	Dao der Datenbank für den Server	127

10.2 Datenbank	132
10.2.1 ServerDatenbank	132
11 Aktivitäts- und Zustandsdiagramme	139
11.0.1 Rezept erstellen und veröffentlichen	139
11.0.2 Rezept löschen	143
12 Sequenzdiagramme	144
12.1 Rezept erstellen und veröffentlichen	144
12.2 Lazy Loading	145
12.3 Favorit hinzufügen und entfernen	146
13 Klassendiagramm	147

1 Zielbestimmung

Dieses Dokument bietet eine Vorlage zur Modellierung des Projektes „Exzellenzkoch“ und basiert auf den Anforderungen, die im Pflichtenheft spezifiziert wurden. Im Rahmen dieses Projektes wird das Modell einer Kochbuchapp mit zugehörigen Strukturen, sowie das Modell eines Webserver, beschrieben. Es wird sowohl textuell, als auch anhand von UML-Diagrammen vorgegeben, wie die Software aufzubauen ist, wie einzelne Komponenten miteinander agieren und wie der Webserver eingebunden wird.

2 Architektur

2.1 Prämissen

Im Aufbau des Projekts wird das Konzept der „Clean Architecture“ umgesetzt [1] [2]. Das bedeutet, dass eine klare Modularisierung besteht. Kernfunktionen und Geschäftslogik werden von der Benutzeroberfläche, sowie von den Datenquellen getrennt. Somit sind die einzelnen Komponenten der Architektur leicht austauschbar. Die Grafik 2.3 zeigt dieses Prinzip. Die grundlegenden Strukturelemente des Projekts liegt in „Entity“. In „Use Case“ ist die Funktionalität der Entities enthalten. Im oberen Teil des äußeren Rings ist die direkte Benutzeroberfläche, in der Eingaben verarbeitet werden, sowie die Abstraktion „Presenter“, welche Logik zum Verarbeiten der Nutzereingaben stellt, zu sehen. Im unteren Teil bietet „Repository“ eine Schnittstelle zu den verschiedenen Datenquellen, welche - genau wie „UI“ - reale Einrichtungen (z.B. Datenbanken und Webserver) darstellen. Dabei ist zu beachten, dass sich Abhängigkeiten einer äußeren Schicht nur auf innere Schicht beziehen dürfen, was anhand der unausgefüllten Pfeile sichtbar ist. **Innere Schichten dürfen also keinerlei Wissen von Aufbau und Funktionalität von höher liegenden Schichten haben.** Der Kern „Entity“ hat demnach keine Abhängigkeiten. Dies ist das Prinzip der *Dependency Inversion*. Der Datenfluss fließt trotz der Abhängigkeits-Regel bidirektional durch alle Schichten hindurch, da z.B. Daten, die aus einer Datenquelle geladen werden, auch in der Oberfläche sichtbar gemacht und dort ggf. auch geändert werden. Dies ist durch gewisse Entwurfsentscheidungen möglich, auf die in späteren Kapiteln noch näher eingegangen wird.

Clean Architecture
Dependency Inversion

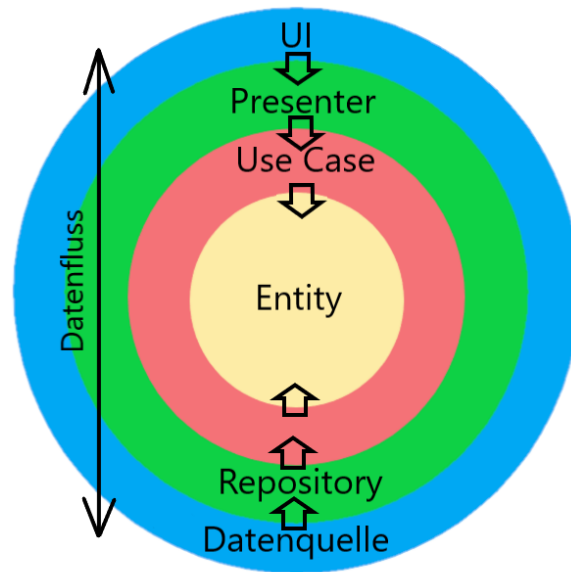


Abbildung 2.1: Clean Architecture Prinzip grafisch dargestellt

2.2 App

2.2.1 Layered Architecture

Wenn man sich die konzentrischen Ringe von Robert C. Martins Clean Architecture einmal vertikal aufgeschnitten vorstellt, sieht man gut, dass die Clean Architecture zu einer Schichtenarchitektur führt, die den Code klar strukturiert.

- **User Interface Layer** (UI und Presenter)
- **Domain Layer** (Use Cases und Entities)
- und **Data Layer**. (Repository, Datenquelle)

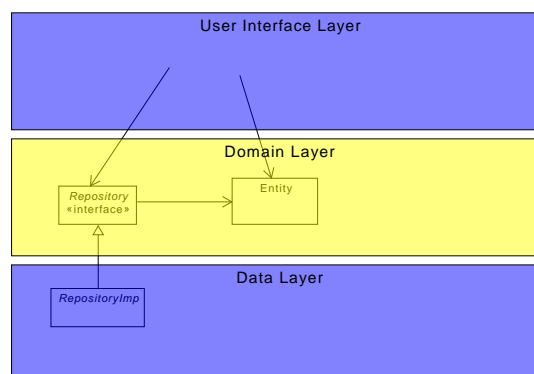


Abbildung 2.2: Layer mit Dependency Inversion

2.2.2 Domain Layer

Im Kern der Architektur steht die *Business Entities*, die unsere Domäne modellieren. Diese beschreiben die „realen“ Objekte unserer Domäne und die Methoden, die mit ihnen möglich sind. Genauer gesagt also Entitäten, wie zB. Rezepte, die es auch gäbe, wenn man das Kochbuch in Papierform führen würde.

In diesem Entwurf ist das sogenannte „Domain Driven Design“ umgesetzt [1] [3], das heißt, dass die Kernentitäten szenariobasiert ermittelt wurden. Somit sind sie aus Szenariensicht von der Domäne aus und nicht von der Datenbank her modelliert.

Repository-Schnittstelle

Zusätzlich zu den Entitäten gehört zu dem innersten Kern der Clean Architecture die Repositoryschnittstelle für die einzelnen Entities.

Diese Schnittstelle kapselt, die durch die Datenzugriffsschicht persistierten Objekte und den Zugriff auf sie, unabhängig davon, ob sie lokal in der Datenbank gespeichert oder per Webservice zur Verfügung gestellt werden. Die Repository-Schnittstelle ist dabei als Schnittstelle (abstrakte Klasse) modelliert und stellt aus Sicht der Domain Logik, die nötigen Speicherfunktionen und Suchfunktionen, die die Anwendung braucht, dar.

2.2.3 Data Layer

Die Datenschicht implementiert die Repositoryschnittstelle. Konzeptionell verhält sich das Repository aus Schnittstellensicht, wie eine Liste von Fachobjekten mit den bereitgestellten Methoden zum Laden und Speichern neuer Elemente.

In der Datenschicht wird zum einen die Persistierung in der lokalen SQLite Datenbank mit den in Android Bibliothek *Room* und zum anderen das Laden und Speichern der Objekte über die REST-Schnittstelle des Servers mit *Retrofit* umgesetzt. Die Bibliotheken sind von Google in den Android Jetpack Developer Guides empfohlen und werden deswegen eingebunden.

Jedes Repository wird mithilfe von DAOs (Data Access Objects) implementiert, die den Zugriff auf einzelne Tabellen in der Datenbank mit der relationalen Datenbanklogik zum Laden und Speichern kapseln.

2.2.4 UserInterface Layer

MVVM

Widergrund!

Als Grundgerüst für die Strukturierung des User Interface Layer wird das Model-View-ViewModel (MVVM) Entwurfsmuster verwendet. Dabei beinhaltet „View“ die grafische Benutzeroberfläche, sowie Anweisungen, wie auf Nutzer- und App Eingaben reagiert werden soll.

Das „ViewModel“ (VM) überwacht die View. Werden durch App-Interaktionen Daten in der View angefordert oder verändert, so merkt das VM dies und gibt dem Model die entsprechenden Befehle weiter. Es ist also ein Vermittler zwischen View und Model. Durch die im Domain Layer beschriebenen Repositoryschnittstelle laden und speichern die ViewModellklassen Fachobjekte.

Das „Model“ in MVVM steht für die Logik der grundlegenden App Funktionen. Im Falle von Exzellenzkoch sind das genau die Entityklassen und Repositoryschnittstellen, wie sie im Domain Layer definiert werden.

2.2.5 Zusammenfassung

Ein Ziel dieser Architektur ist die Entkoppelung der sich ändern könnenden Schichten. So sind z.B. Nutzerschnittstelle und Datenschicht nur ein Detail.

Die Anwendung wird so aufgebaut, dass zum Beispiel eine völlig andere Nutzerschnittstelle statt der Androidapp, oder ein dateibasierter Speicher statt einer relationalen Datenbank, umsetzbar wäre, ohne, dass dies auf die anderen Schichten Einfluss hätte.

Ein Vorteil, der dadurch entsteht, ist die klare Abgrenzung der einzelnen Funktionen und die leichtere Testbarkeit und Wartbarkeit der einzelnen Subsysteme. Außerdem wird durch diese Modularisierung das ganze Projekt leicht erweiterbar gemacht.

2.3 Server

2.3.1 Einleitung

Registriert sich ein Nutzer in der App, so kann er mit anderen Nutzer interagieren. Diese Interaktion wird durch die Serverkomponente ermöglicht. Ein eingeloggtter User, kann mit der App auf seinem Smartphone Daten zum Server schicken und damit zum Beispiel Rezepte veröffentlichen. Andere eingeloggte User können dann nach diesen Rezepten suchen. Um dies zu ermöglichen, bietet der Server eine REST-Schnittstelle an. Unter verschiedenen „Endpoints“ werden, ähnlich wie bei der Repositoryschnittstelle, verschiedene Methoden angeboten, um Entities zu laden und zu speichern.

Beispiel

Ein HTTP GET auf den Endpunkt:

<http://psekochbuch.de/api/users/MagnumMandel>¹

liefert zum Beispiel das passende JSON-Dokument zurück:

```
{
  "userid": "MagnumMandel",
  ...
  "description": "Veganer essen meinem Essen das Essen weg."
}
```

Im Folgenden wird der Server mit seinen Komponenten näher beschrieben.

2.3.2 Technologiestack

Der Server wird als Spring-Boot-Anwendung mit Rest-Schnittstelle umgesetzt. Authentifizierung erfolgt nach dem OAuth 2.0 Standard mit JSON-Webtokens und soll mit Hilfe der Firebase Library verwirklicht werden. Die Daten werden mit Hilfe von Hibernate auf eine relationale Datenbank gemappt und gespeichert und geladen.

2.3.3 Struktur

Spring-Boot hat mit integrierter Dependency-injection und Annotationen einen klare Vorgabe, wie eine REST-Applikation umzusetzen ist. Mit verschiedenen Annotationen werden Klassen ausgezeichnet. Diese annotierten Klassen, die das Grundgerüst der Anwendung darstellen, werden Beans genannt. Der Spring IOC Container assembliert diese Beans dann, gemäß ihrer durch Annotationen bestimmten Rollen, zu einer Anwendung.

¹Dies ist nur ein grober Überblick, modulo Authentifizierung. Ist diese aktiviert, erfolgt die Verbindung TLS-verschlüsselt. Gegebenenfalls leitet der Resource Server (unser Server) einen nicht autorisierten User an den Authentication Server von Google weiter, und erst wenn dieser mit korrektem JSON-Access-Token anfragt wird das JSON-Dokument als Antwort geschickt

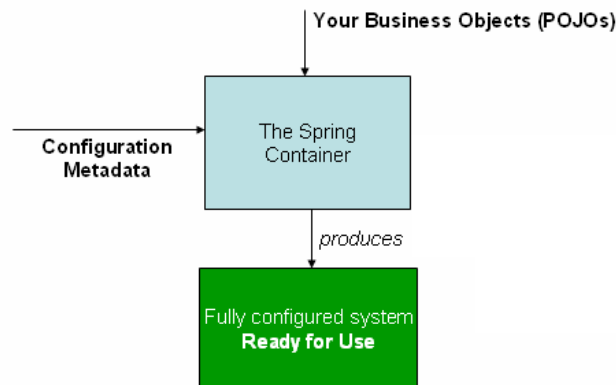


Abbildung 2.3: Spring Inversion of Control Container[4]

Der Server ist in verschiedene Bereiche aufgeteilt:

Zum einen gibt es eine Klasse, die die Einstellungen für den Server und die Logik für die Authentifizierung, wie z.B. die Verbindung zu einer Authentifizierungsschnittstelle wie Firebase und die Überprüfung des JSON-Tokens, beinhaltet.

Des weiteren gibt der Server eine API an, durch welche die Restschnittstelle mit ihren Endpunkten definiert wird.

Ein weiterer Bereich enthält die Logik für die Datenhaltung und Ansteuerung dieser API.

Die Paketeinteilung ist in Kapitel 4.2 erläutert.

2.4 Deployment

2.4.1 Server-Deployment

Als Springboot-Applikation ist die deploybare App des Servers ein JAR-File, dass den HTTP-Server Tomcat und alle Abhängigkeiten enthält. Es braucht kein vollwertigen Java EE Server konfiguriert und installiert werden, der dann einem WAR oder EAR bestückt. Das heißt als Umgebung auf unserem Bw-Cloud-server wird nur ein Linuxsystem mit installiertem JavaJDK und eine relationale Datenbank benötigt.

2.4.2 App-Deployment

Das Gradle Build-tool erstellt eine apk-Datei, die alle benötigten Bibliotheken, die nicht auf den Endgeräten vorausgesetzt werden und das Manifest, was die Installation spezifiziert, enthält. Die apk-Datei kann dann entweder über das Android Debug Bridge-Tool (adb) direkt auf ein Endgerät installiert oder über den *Google Play Marketplace* veröffentlicht werden.

2.5 Authentifizierung und Authorisierung

Die App und der Server sollen ein Rechtemodell umsetzen. Das bedeutet, dass Nutzer, je nach Status, unterschiedliche Rechte für die Nutzung der Funktionalität der App, besitzen. Beispielsweise soll jeder Nutzer, auch wenn er noch keinen Account hat, nach Rezepten suchen können. Das Updaten und Löschen eines öffentlichen Rezeptes (also das „Neuveröffentlichen“) soll nur dem jeweiligen Autor gestattet sein. Kommentare zu Rezepten posten können nur eingeloggte Nutzer mit Account. Ein Nutzer, der den Status „Admin“ hat, kann in der Rolle des Admin auch Rezepte fremder Nutzer löschen.

Das bedeutet, dass es eine Authentifizierung und eine Authentisierung von Nutzern im Server geben muss. Diese soll dem OAuth 2.0 Standard mit JSON-Webtokens genügen.

Bei OAuth 2.0 wird das Passwort nur beim Log-In verschickt. Als Antwort bekommt der Client zwei Token zurück. Weitere Anfragen werden nur mithilfe des ersten Tokens durchgeführt. Dieses Token *access token* genannt ersetzt das Passwort für eine bestimmte Zeit. Ist die Zeit abgelaufen, kann der Client mit dem zweiten Token, dem *refresh token* sich ein neues Token holen.

Das Access Token ist nicht zufällig generiert, sondern enthält die ID des Users und Meta-informationen, wie das Ablaufdatum des Tokens. Es ist mithilfe eines Schlüssels signiert, sodass es nicht gefälscht werden kann. Jeder Service kann somit sowohl den User, als auch die Metainformationen, aus dem JSON-Token abrufen, ohne sich diese Informationen merken zu müssen.

Um dies zu implementieren, werden die Funktionen der Google Firebase API angewendet. Diese besitzt, für die App-Komponente und für den Server, Libraries, die die meisten Teile des Protokolls transparent und standardisiert dem OpenID-Connect-Standard genügen. Das heißt, der Authorization-Server und damit auch die Passwort- und Accountverwaltung wird von Google gestellt, so wie es nach Abgrenzungskriterium A1 vorgegeben ist. Der eigens implementierte Server dieses Projekts, ist der Ressource-Server.

2.5.1 Adminrolle

Der oben beschriebene Aufbau von OAuth 2.0 und den JSON-Webtokens, erlaubt eine elegante Umsetzung der Adminrolle, durch Hinzufügen eines „Custom Claims“. Diesen kann der Operator des Servers über die Firebase-Konsole von Google für einzelne Accounts festlegen.

Dann ist dieser Custom Claim im Payload-Feld des JSON-Webtokens verfügbar und der Zugriff von den HTTP Endpoints auf die REST-Schnittstelle kann entsprechend beschränkt werden.

Das JSON-Web-Token hat also beispielsweise ein Feld wie folgt:

```
{  
  "loggedInAs" : "admin",  
}
```

Aufgrund dieses Feldes kann der Server, pro Zugriff auf eine spezielle Funktion, entscheiden, ob der Zugriff erlaubt wird oder nicht.

2.6 Entwurfsentscheidungen

Der Server wird wie beschrieben mit Spring Boot umgesetzt. Es wurde entschieden als Programmiersprache sowohl für den Server als auch für die App Kotlin zu verwenden. Als Build-automatisierungs-Tool wird für Server und App Gradle verwendet.

Inzwischen wird Kotlin und Gradle sowohl von Springboot als auch dem Androidframework mit einem gewissen Reifegrad unterstützt. Die Entscheidung dieselbe Sprache und dasselbe Buildsystem zu nehmen garantiert eine gewisse Vereinheitlichung der Tools in der Entwicklung. Kotlin als Sprache erlaubt eine objektorientierte Entwicklung, was eines der Ziele des PSE-Praktikums ist.

Spring Boot, hat mit seinem „convention-over-configuration“-Ansatz sehr klare Architektur- und Konfigurationsempfehlungen, was einen Aufbau mit guter Struktur vereinfacht.

Die Schnittstelle zwischen Server und App als REST-Schnittstelle mit DTOs zu definieren, ist ein gewisser Mehraufwand. Man könnte, da Server und App beide in Kotlin geschrieben sind, auch zum Beispiel RMI verwenden. Der Vorteil ist, dass mit der REST-Schnittstelle und den DTOs ein klarer Vertrag zwischen Server und App und eine klare Trennung erzielt wird. Dies erleichtert das Testen der einzelnen Komponenten und das Bearbeiten der App und Server in Subteams mit reduziertem Kommunikationsaufwand.

Auch ist REST inzwischen der vorherrschende Standard, um einen Webservice umzusetzen. Die Unterstützung ist gut in das Spring-Boot-Framework integriert.

Rest ermöglicht HTTP Technologien wie Caching zu Verwenden. Das ist ein Konfigurationspunkt für LoadBalancing, der eine spätere Skalierung auf mehrere Server einfach möglich machen würde, wenn dies irgendwann benötigt werden würde.

Der gewählte Ansatz mit SpringBoot ermöglicht es die Serveranwendung als eine deploybare Datei auf einem eigenen Server zu installieren. Das reduziert den Deploymentaufwand im Gegensatz zu einem Deployment mit einem eigenständigen Application Server beträchtlich.

Installation auf einem selbst administrierten Linux-Server bedeutet, dass Javaumgebung und relationale Datenbank selbst administriert werden müssen.

Eine Alternative zu dieser Herangehensweise wäre eine der vielen Cloud Plattformen zu wählen zum Beispiel Amazon Elastic Compute Cloud oder Google App Engine. Damit würden Schritte, wie das Aufsetzen einer Datenbank entfallen.

Gleichzeitig würde dies den Server an eine vendorspezifische Auswahl von Programmiersprachen APIs und Frameworks binden. Dies wird mit dem gewählten Ansatz für den Server versucht zu verringern.

3 Interaktion zwischen Komponenten

3.1 Interaktion von App Komponenten

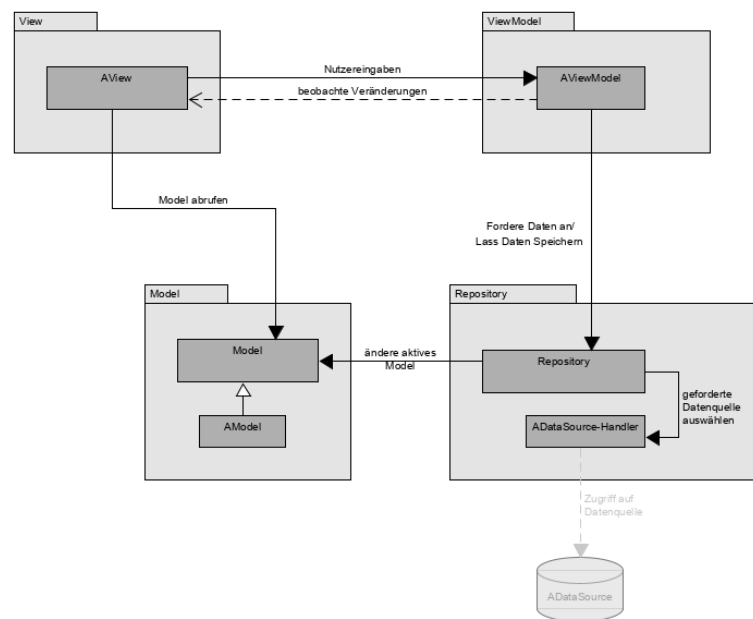


Abbildung 3.1: Interaktion der App-Komponenten

Diagramm 6.1 beschreibt die Interaktion zwischen den einzelnen App-Komponenten etwas spezifischer als im vorigen Kapitel. Zu beachten ist dabei, dass die angezeigten Komponenten „AView“, „AViewModel“, „AModel“ und „DataSource“ nur beispielhafte Instanzen darstellen. Jedes der ihnen zugrundeliegenden Pakete kann noch weitere Instanzen enthalten, die hier zur Anschaulichkeit vernachlässigt wurden. Die Interaktion der App-Komponenten funktioniert folgendermaßen: Ändert ein Nutzer Daten über die Benutzeroberfläche, so wird in der derzeit aktiven View-Komponente („AView“) eine Änderung vom zugehörigen ViewModel („AViewModel“) registriert. Dieses gibt an das Repository durch, welche Daten geladen oder verändert werden müssen. Das Repository fordert daraufhin entweder Daten vom Server an, bzw. veranlasst eine Aktualisierung der Serverdaten, oder es holt, bzw. aktualisiert die Daten aus der lokalen SQLite-Datenbank („ADataSource“). Außerdem aktualisiert das Repository das momentan aktive Model („AModel“) und gibt diesem die gerade geladenen oder aktualisierten Daten. Dadurch können neue Daten in der View angezeigt werden.

3.2 Interaktion von Serverkomponenten

In dem API Package des Servers(siehe Kapitel 4.2) befindet sich eine Menge an Interfaces, die angeben, wie man vom Client auf die API zugreifen kann. Diese Interfaces werden von den Controller-Klassen implementiert, insbesondere wird jede API-Klasse von einem zugehörigen Controller implementiert. Die Controller bearbeiten die Anfragen, die von den App-Klassen an den Server gestellt werden. Dabei rufen sie Serviceklassen auf, welche dann auf sogenannte „Data Access Objekte“ (DAOs) zugreifen, um entweder Daten zu laden oder Daten zu speichern. Die Serviceklassen geben, wenn Daten angefragt wurden, die geladenen Daten an die Controller zurück und durch die Controller werden dann die Daten an den Client geschickt. Bevor ein Client Daten anfragen oder senden kann, muss dieser erst Authentifiziert werden. Dazu wird ein JSON-Web-Authentication-Token, welcher bei der Anfrage mitgesendet wird, durch die Firebaselibrary überprüft.

3.3 App- und Server-Kommunikation

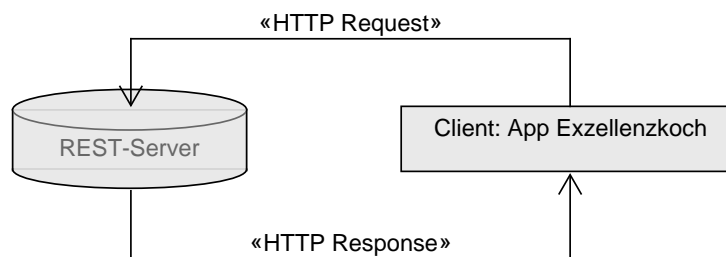


Abbildung 3.2: Server-AppKommunikation über REST

Der Server bietet eine REST-Schnittstelle. Über diesen kommunizieren App und Server miteinander. Hierbei wird das Request-Response Verfahren verwendet, wie in Abbildung 3.2 vereinfacht gezeigt. Der Client schickt eine Anfrage an den Server. Eine Anfrage verwendet das HTTP-Protokoll. Entweder wird eine GET-, POST-, PUT- oder DELETE-Anfrage gesendet. Der Server bearbeitet die Anfrage und gibt eine Antwort zurück. Die Daten werden als JSON-Dateien versendet. Die Verbindung zwischen Client und Server besteht, solange die Anfrage und Antwort durchgeführt, oder eine Zeitbegrenzung überschritten wird. Statt dass die App ganze Entity-Objekte an den Server sendet, werden diese in „Data Transfer Objects“ (DTOs) abstrahiert. Die DTOs sind so aufgebaut, dass sie auf die Server-Datenbank Tabellen gemapped werden können. Mehr zum Aufbau der DTOs ist in Kapitel 9.2 zu finden.

4 Paketeinteilung und Klasseninteraktion

4.1 App

4.1.1 User Interface Layer

4.1.2 View

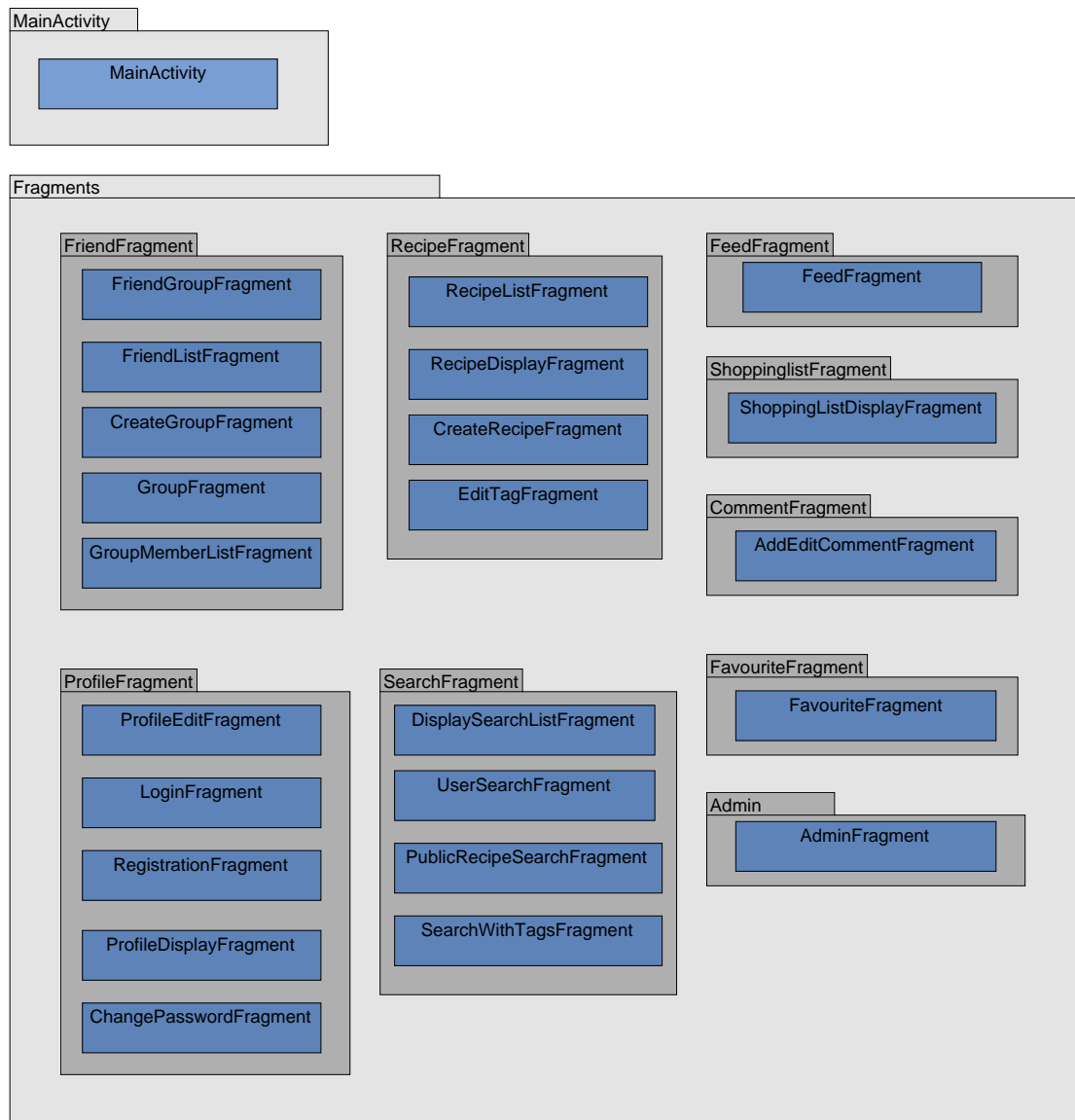


Abbildung 4.1: Einzelne Pakete des User Interface Layers

Die in Abbildung 5.24 gezeigten Klassen sind die Klassen der View. Die View erhält die Benutzereingabe direkt. Die grafische Oberfläche besteht aus einer Hauptaktivität, auch MainActivity, die durch ein Menü Fragments aufrufen kann. Das Menü ist ein eigenes Fragment, welches in der Toolbar der MainActivity zu sehen ist (mehr dazu in Kapitel 5.1). Die Fragments sind Klassen, denen .xml-Dateien zugeordnet sind und die die verschiedenen Oberflächen darstellen.

Abbildung 4.2 zeigt einen groben Überblick über die Interaktion zwischen allen Fragments und der MainActivity. Die Pfeile zwischen den Fragments zeigen an, welche Fragments uni- oder bidirektionale Aufrufe aufeinander haben. Da das Menü in der Toolbar stets sichtbar ist, gibt es von jedem Fragment auch eine Interaktion „Menü öffnen“, welche in 4.2 im Sinne der Übersichtlichkeit nicht extra dargestellt wird. Was ebenfalls nur exemplarisch modelliert ist, ist der „Zurück“-Button des Smartphones, über den jedes Fragment verlassen werden kann. Nutzt man den „Zurück“-Button, gelangt man zurück auf das vorher angezeigte Fragment. Gibt es kein vorheriges Fragment mehr, weil man auf der Startseite der App ist, verlässt man die App mit dem Button. Folgende Grafiken beschreiben nun genauer die Interaktionen zwischen den Fragments.

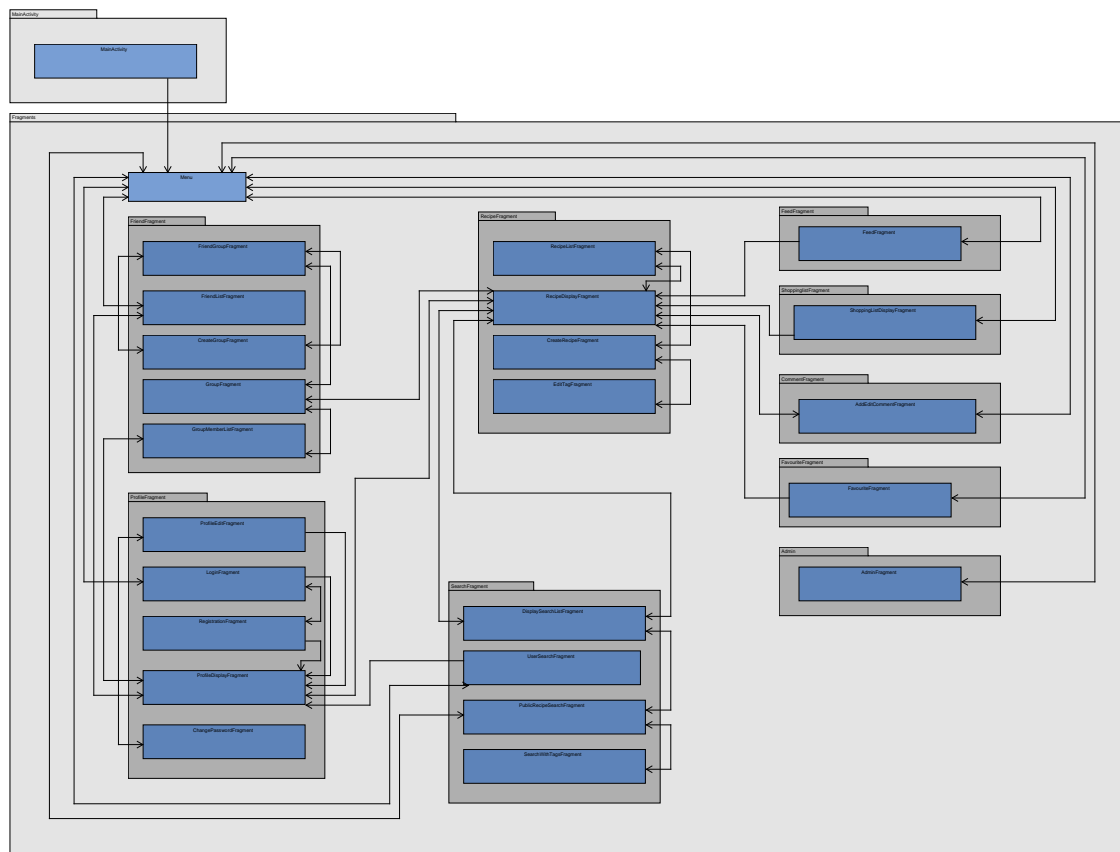


Abbildung 4.2: Interaktion zwischen den Fragments und der Main-Activity

FriendFragment Paket

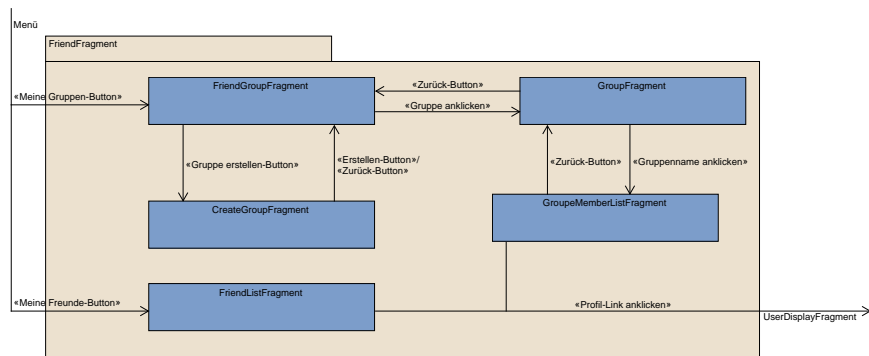


Abbildung 4.3: FriendFragment Paket

Das FriendFragmentPaket ist dafür zuständig, die graphischen Oberflächen der Freunde- und Gruppenverwaltung zu erstellen. Sowohl die Übersicht über die Gruppen eines Nutzers, als auch die Liste der Freunde eines Nutzer haben einen eigenen Menüpunkt, deswegen sind diese Fragments über das Appmenü erreichbar.

RecipeFragment Paket

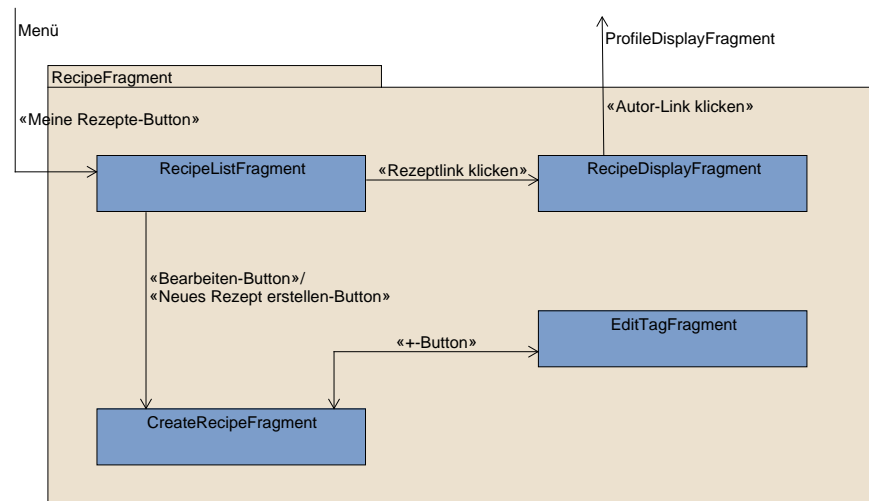


Abbildung 4.4: RecipeFragment Paket

Das RecipeFragmentPaket ist dafür zuständig die graphischen Oberflächen der Rezeptansichten. Wichtig ist die Unterscheidung von öffentlichen und privaten Rezepten. Öffentliche Rezepte haben ein festes Format und haben kein einziges Attribut = null, wobei private Rezepte nicht formatiert sein müssen und leere Attribute haben können. Daher ist die Anzeige der zwei Rezepttypen verschieden. Das RecipeListFragment stellt eine Liste der privaten Rezepte eines Autors dar. Es kann über das Appmenü aufgerufen werden.

ProfileFragment Paket

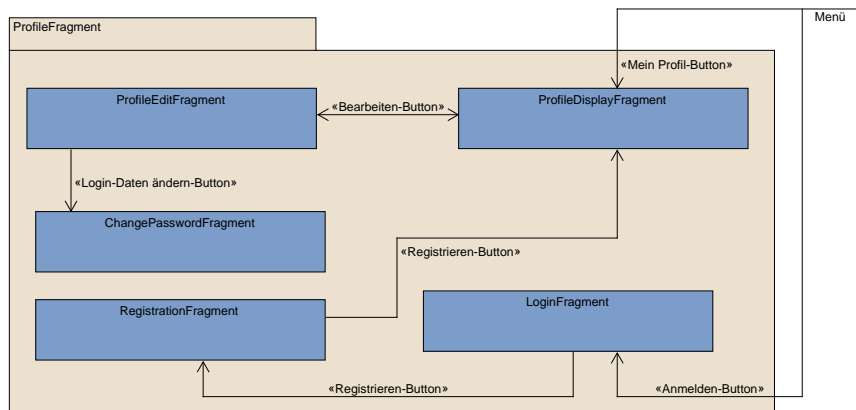


Abbildung 4.5: ProfileFragment Paket

Das ProfileFragment Paket ist dafür zuständig, eine Graphische Oberfläche für die Nutzerverwaltung bereitzustellen. Ein angemeldeter Nutzer kann über das Appmenü das eigene Profil aufrufen. Dann wird das ProfileDisplayFragment angezeigt. Über den „Bearbeiten“-Button kommt der Nutzer auf das ProfileEditFragment seines Profils. Möchte er sein Passwort ändern, kann er über den „Login-Daten ändern“-Button das ChangePasswordFragment öffnen, um das zu tun. Das LoginFragment ist über das Appmenü erreichbar. Über den „Registrieren“-Button gelangt man auf das RegistrationFragment. Hat sich in diesem Fragment ein Nutzer registriert und klickt auf den „Registrieren“-Button, wird er auf das ProfileDisplayFragment seines neu erstellten Profils geleitet.

SearchFragment Paket

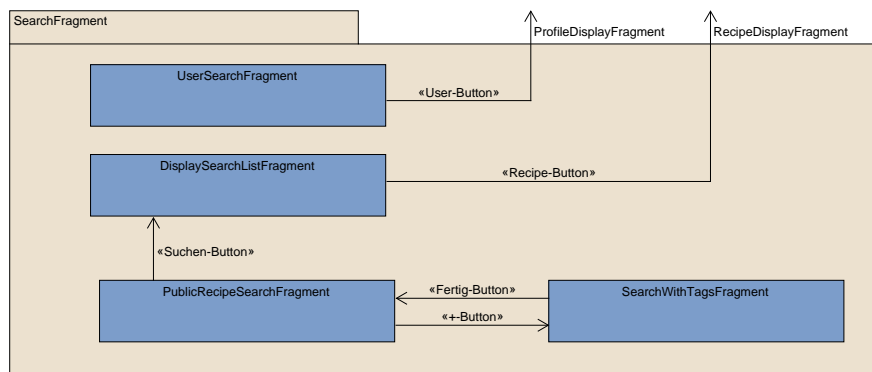


Abbildung 4.6: SearchFragment Paket

Das SearchFragment Paket ist dafür zuständig die graphische Oberfläche für die verschiedenen Suchfunktionen bereit zu stellen.

ShoppingListFragment Paket

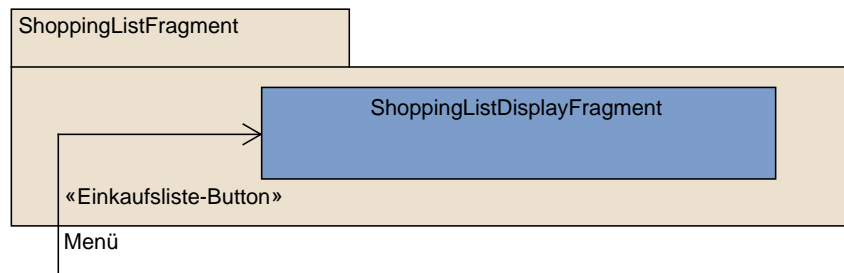


Abbildung 4.7: ShoppinglistFragment Paket

Das ShoppingListFragment Paket ist dafür zuständig, die Oberfläche der Einkaufsliste darzustellen. Das Fragment ist über das Appmenü erreichbar.

FeedFragment Paket

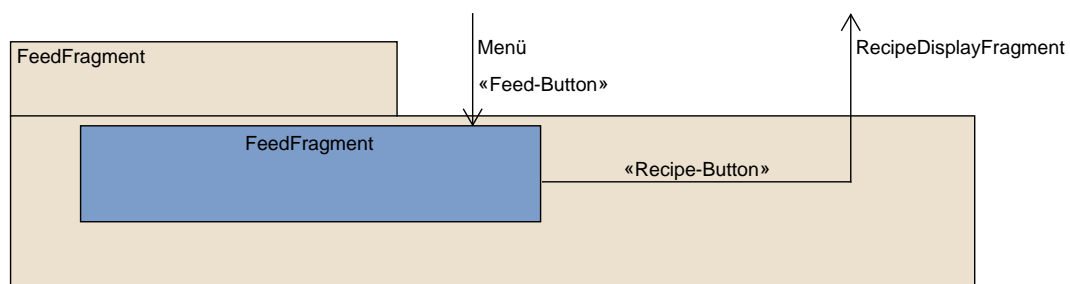


Abbildung 4.8: FeedFragment Paket

Das FeedFragment Paket ist dafür zuständig, die Oberfläche des Feeds darzustellen. Wenn diese Funktionalität implementiert ist, ist der Feed die Startseite der App und durch das Appmenü erreichbar. Klickt ein User auf einen der Buttons im Feed, so wird er auf das entsprechende Rezept geleitet und das RecipeDisplayFragment öffnet sich.

FavouriteFragment Paket

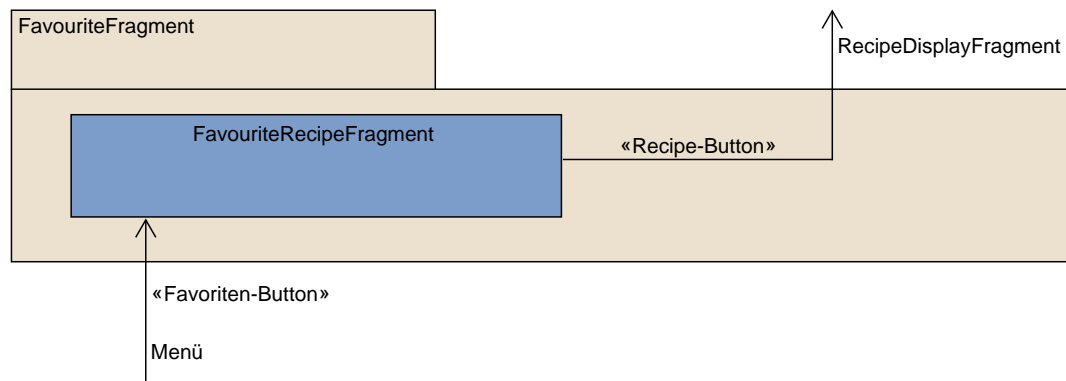


Abbildung 4.9: FavouriteFragment Paket

Das FavouriteFragment Paket ist dafür zuständig, die Oberfläche der Favoritenliste darzustellen. Diese hat einen eigenen Punkt im Appmenü und ist somit von diesem aus erreichbar. Verlassen wird das Fragment entweder über den „Zurück“-Button, oder, indem der Nutzer auf ein angezeigtes Rezept in der Favoritenliste klickt. Dann wird das RecipeDisplayFragment geöffnet.

AddEditCommentFragment Paket

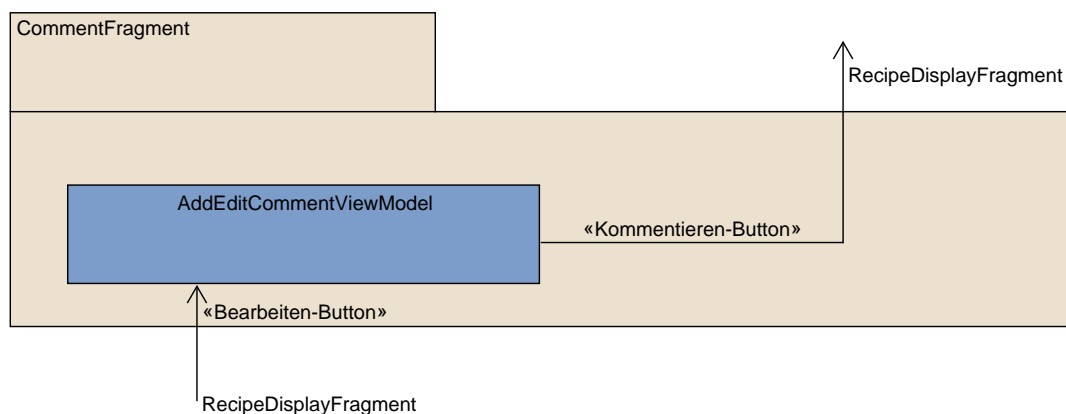


Abbildung 4.10: FavouriteFragment Paket

Das AddEditCommentFragment Paket ist dafür zuständig, die Oberfläche der Kommentarbearbeitung darzustellen. Das Fragment wird aufgerufen, wenn im RecipeDisplayFragment ein angemeldeter Nutzer bei einem seiner eigens erstellten Kommentare auf „bearbeiten“ drückt, oder wenn ein angemeldeter Nutzer auf den „Kommentar schreiben“-

Button drückt. Wird der „kommentieren“-Button gedrückt, so wird das vorherige Recipe-DisplayFragment wieder angezeigt.

AdminFragment Paket

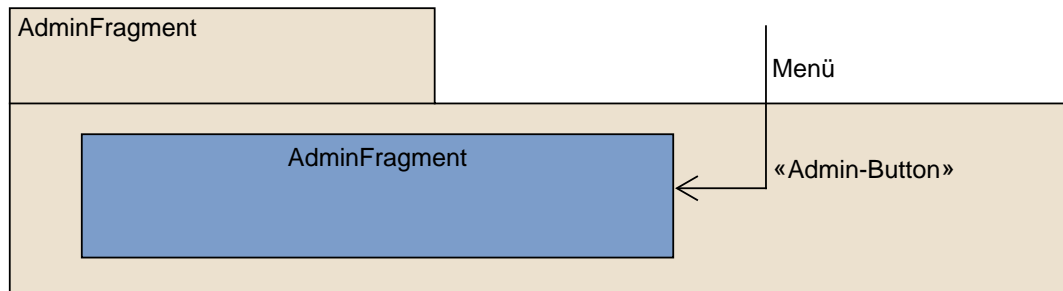


Abbildung 4.11: AdminFragment Paket

Das AdminFragment Paket ist dafür zuständig, die Oberfläche des Admins darzustellen. Über diese hat der Admin die Möglichkeit gemeldete Rezepte, Nutzer, oder Gruppen manuell zu überprüfen und zu entfernen. Ist ein Nutzer als Admin angemeldet, so ist dieses Fragment über das Appmenü aufrufbar.

4.1.3 ViewModel

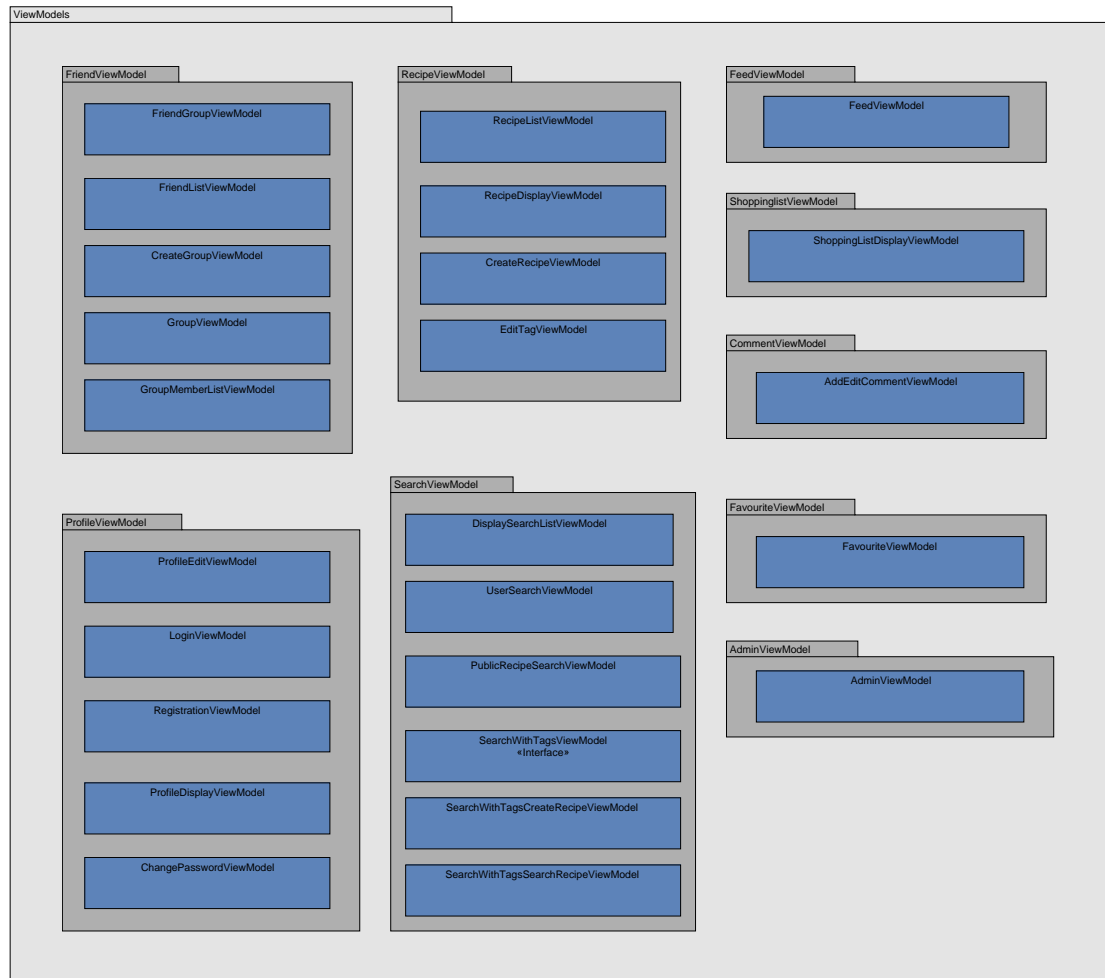


Abbildung 4.12: Die Packages mit enthaltenen ViewModel-Klassen

An sich sieht das ViewModel-Paket von der Struktur her analog so aus, wie die View-Struktur, da jedes Fragment ein ViewModel benötigt. Das `SearchWithTags` Fragment ist die einzige Ausnahme, da es von zwei verschiedenen Fragments aus aufgerufen wird und deswegen unterschiedlicher Funktionalitäten bedarf (mehr dazu in Kapitel 5.2). Was auch anders ist, sind die Verbindungen. Die ViewModels haben untereinander keinerlei Interaktion, sondern geben Daten und Anfragen durch Befehle an das Repository zu den Datenquellen durch. Jede VM-Klasse hat also eine Verbindung zum Repository.

4.1.4 Domain Layer

Domain Entities

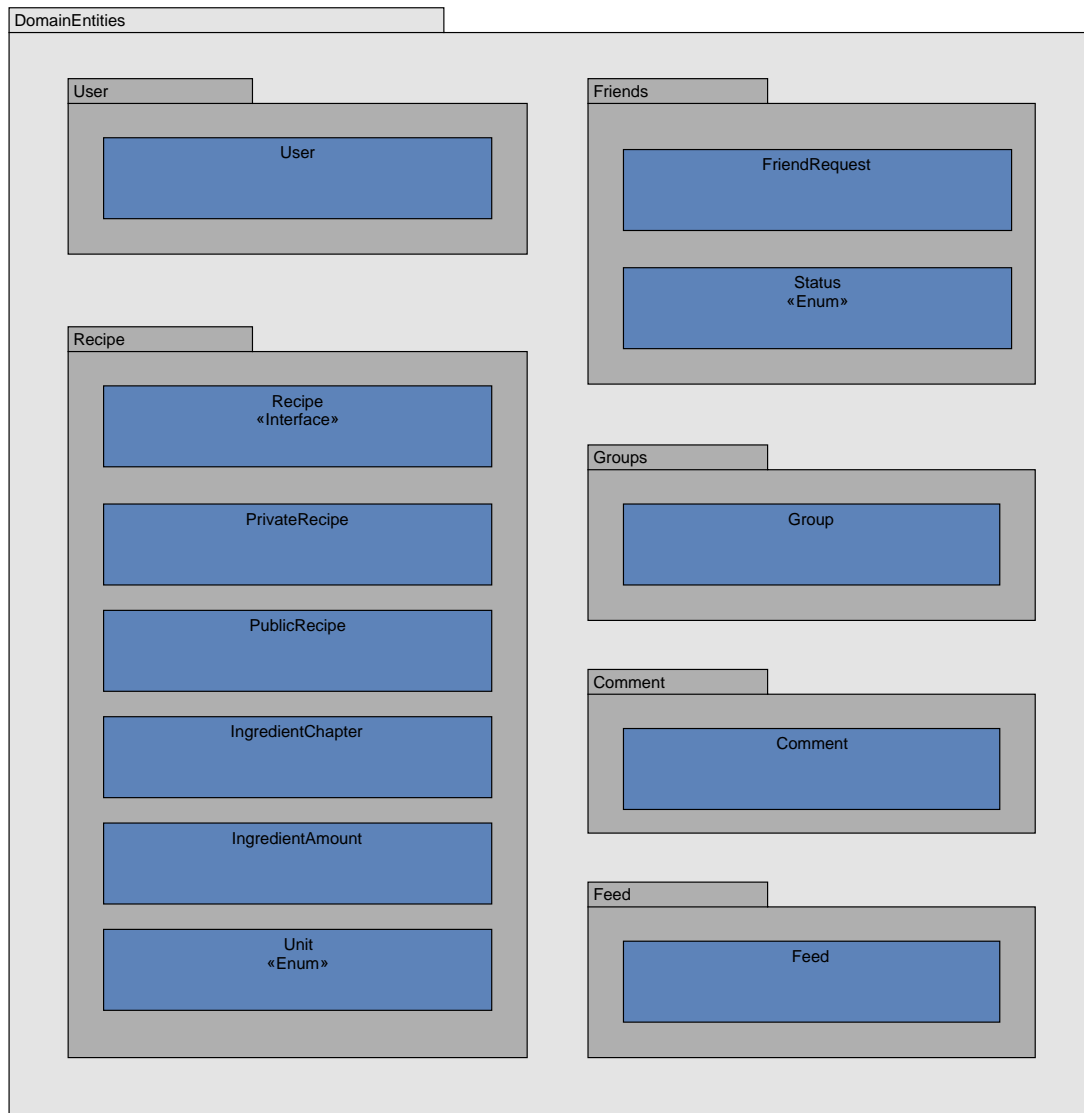


Abbildung 4.13: Das Domain Entity-Paket mit Klassen

Die Domain Entities stellen auf der Appseite den Kern des Models dar. Sie liegen in einem Paket und interagieren miteinander, aber nicht mit Klassen außerhalb ihres Paketes. Die Interaktion der Entities ist in Kapitel 6.1 ausführlicher erklärt.

Domain Layer Interfaces

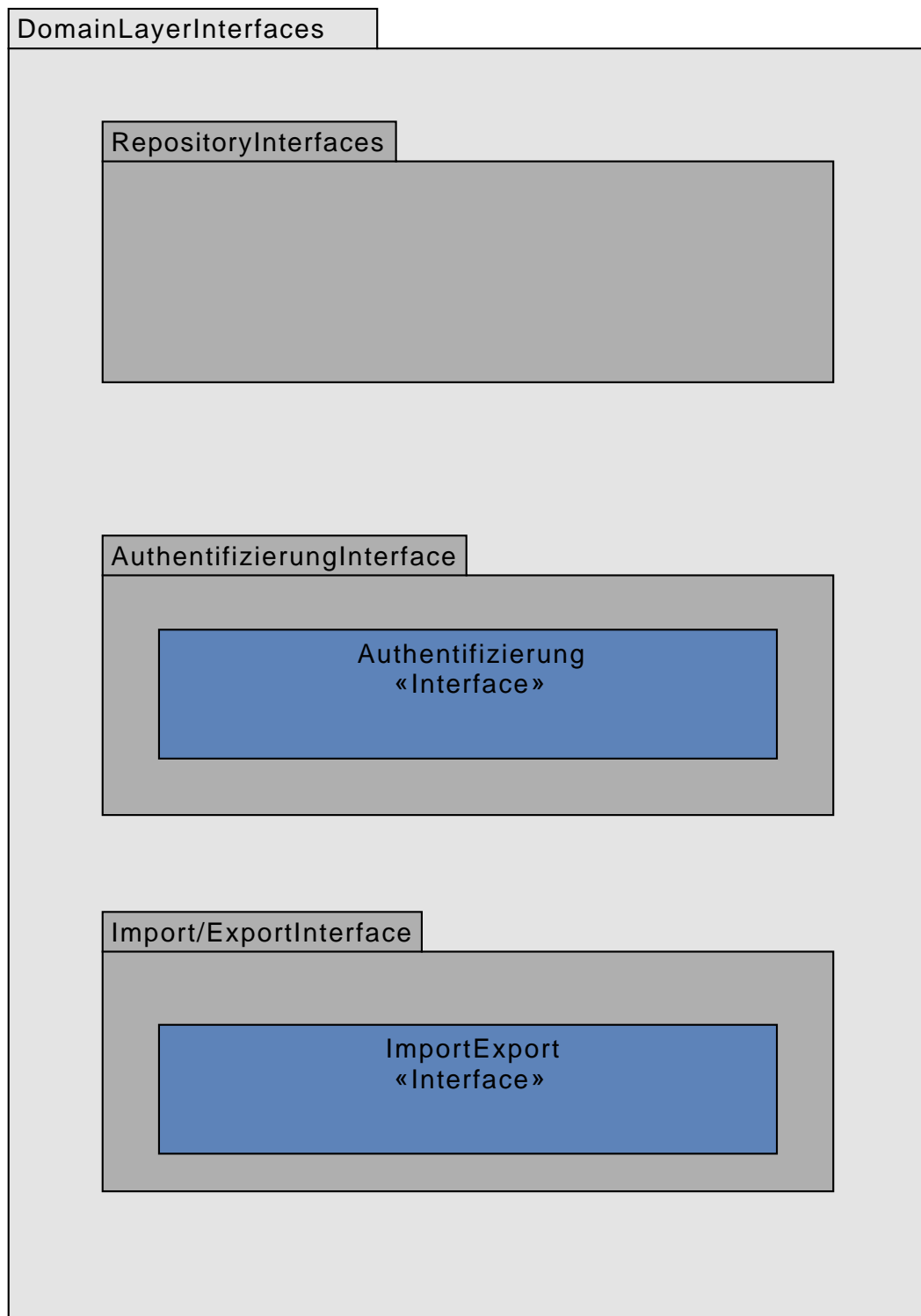


Abbildung 4.14: Die Interfaces des Domain Layers

Im Domain Layer liegen einige Interfaces, die für verschiedene Funktionalitäten gebraucht werden. Unter anderem sind die Repository-Interfaces, das ImportExport-Interface und das Authentifizierungs-Interface hier enthalten. Diese bieten Schnittstellen, um Bibliotheken oder eigene Funktionale Klassen (zB. das Repository) gekapselt einzubinden. Im Repository-Interfaces-Package sind verschiedene Interfaces enthalten. Für jede Klasse im Repository (siehe folgenden Abschnitt), ist ein Interface im Package enthalten, welches die Methoden und Attribute der Klasse vorgibt.

4.1.5 Data Layer

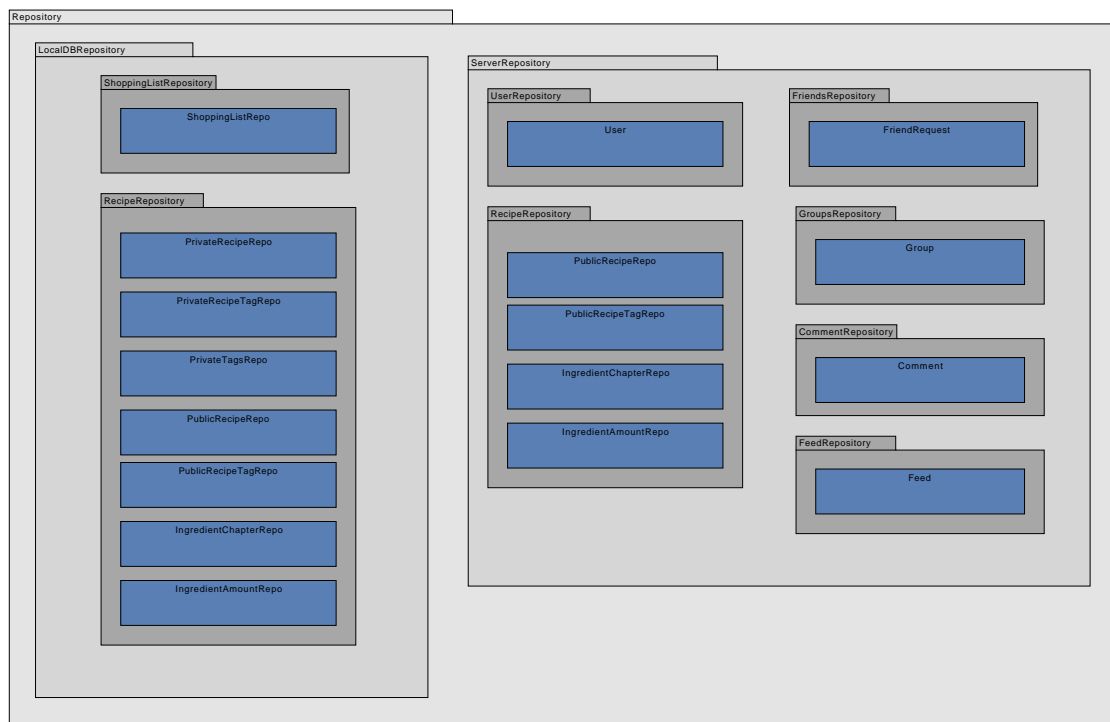


Abbildung 4.15: Das Paket mit Klassen des Repositories

Das Repository enthält für fast jede Domain Entity eine Klasse, in der Methoden zur Datenverwaltung dieser Entity enthalten sind. Es ist aufgeteilt in ein Package für die lokale SQLite Room-Datenbank und eines für den Webserver.

4.2 Server

4.2.1 Api und Model Pakete

In der Abbildung 4.16 werden alle Pakete und Klassen aufgelistet, welche für die Kommunikation mit den Clients (Api/Controller), behandeln von den Anfragen (Services) und den

Zugriff auf die Datenbank (Dao und Model) zuständig sind. Die DTOs sind die Klassen bzw. Objekte, welche über das Netzwerk zwischen Client und Server versendet werden.

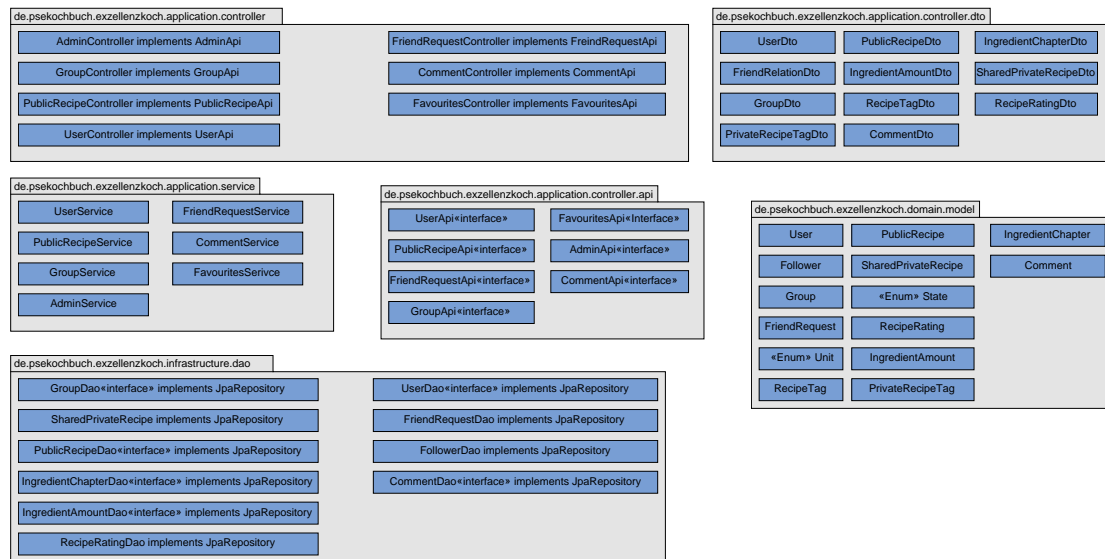


Abbildung 4.16: Die Packages und Klassen für die ServerApi und die Anbindung an die Datenbank

4.2.2 Konfiguration- und Ausführungspakete

In der folgenden Abbildung 4.17 werden alle Pakete und Klassen aufgelistet, welche für das Konfigurieren, Starten von dem Server und Authentifizieren von Nutzern zuständig sind.

Das Paket Security beinhaltet Klassen, die für die Authentifizierung von Nutzern, die den Dienst anfragen, verwendet werden. Die Klassen verwenden Spring Boot Security und Firebase. Das Paket exzellenzkoch enthält nur die Klasse ExzellenzkochApplication. Diese Klasse enthält die Main-Methode und startet den Dienst. Die Konfigurationsklassen befinden sich im Paket config. In WebSecurityConfig wird die Konfiguration der Spring Boot Security festgelegt und in der Klasse FirebaseConfig die Konfigurationen für die Firbaseverbindung. Im Paket exceptions befinden eine Exceptionenklasse, welche geworfen wird, wenn eine Element in der Datenbank nicht gefunden wird.

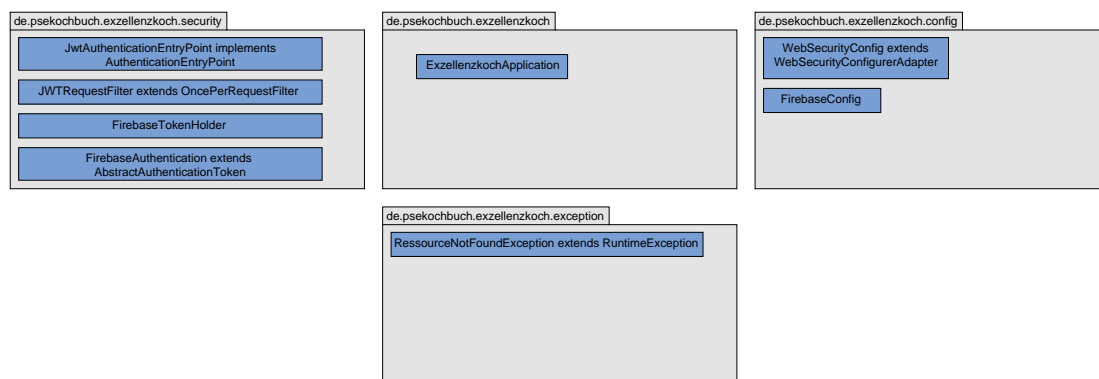


Abbildung 4.17: Die Packages und Klassen für die Konfiguration und Ausführung des Servers und Authentifizierung von Nutzern

4.2.3 Kommunikation zwischen Controllerpaket und Servicepaket

4.18 zeigt den Zugriff der Controller-Klassen auf die Service-Klassen. Jeder Controller greift auf seinen Service zu.

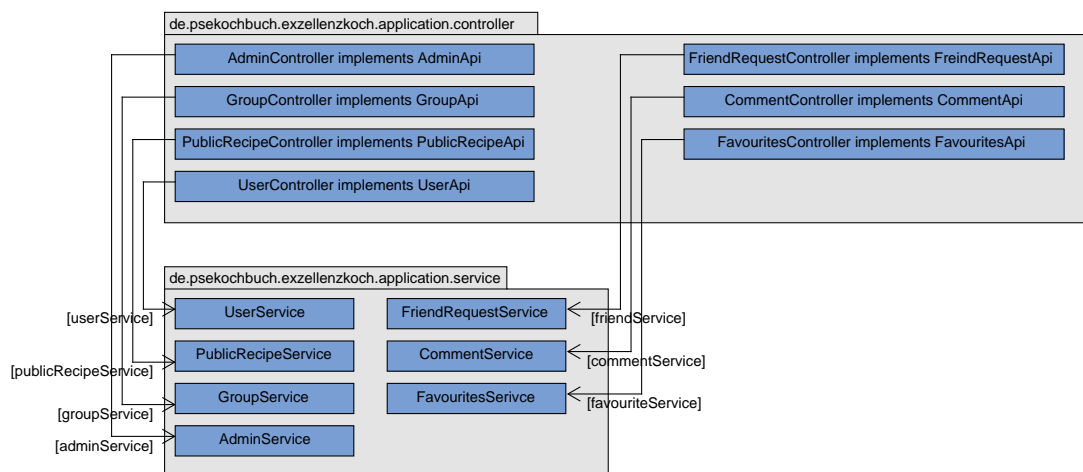


Abbildung 4.18: Die Packages und ihre Klassen und die Kommunikation zwischen Service und Controller

4.2.4 Kommunikation zwischen Servicepaket und Daopaket

In 4.19 wird der Zugriff der Service-Klassen auf die Dao-Klassen abgebildet. Ein Service kann auf verschiedene Dao zugreifen und ein Dao wird auch von verschiedenen Service-Klassen verwendet.

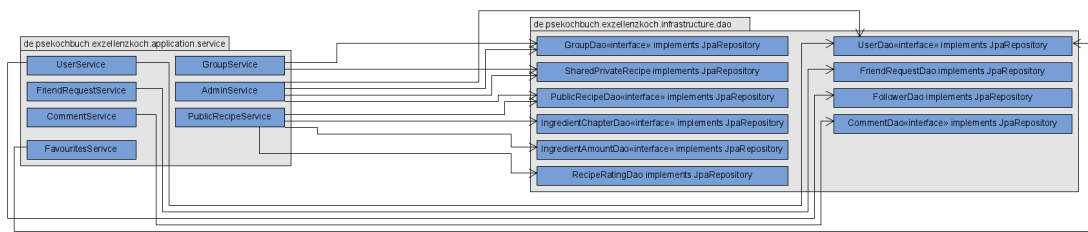


Abbildung 4.19: Die Packages und ihre Klassen und die Kommunikation zwischen Services und Dao

5 Klassenbeschreibung Userinterface

In diesem Kaptiel werden konkrete Attribute und Funktionalitäten der Klassen der App vorgestellt.

5.1 View

5.1.1 Toolbar und Appmenü

Das Userinterface der App besteht aus einer einzigen Activity („MainActivity“), aus der heraus verschiedene Fragments aufgerufen werden können. Zu jedem Fragment gehört ein ViewModel. In der MainActivity ist eine Toolbar definiert. In dieser Toolbar ist das Appmenü enthalten. Das Appmenü ist ein eigenes Fragment. Es wird über den Menübutton, der in der Toolbar liegt, aufgerufen und bietet über Buttons Zugriff zu verschiedenen ausgewählten Fragments. Diese Fragments sind:

- RecipeListFragment: in Abb. 5.1 dargestellt durch Button: „Meine Rezepte“
- ShoppigListDisplayFragment: in Abb 5.1 dargestellt durch Button: „Einkaufsliste“
- FavouriteFragment: in Abb 5.1 dargestellt durch Button: „Admin“
- LoginFragment: in Abb 5.1 dargestellt durch Button: „Login“
- AdminFragment: in Abb 5.1 dargestellt durch Button: „Admin“
- FriendlistFragment (wenn **W14** implementiert ist): in Abb 5.1 dargestellt durch Button: „Freunde“
- FriendGroupFragment (wenn **W14** implementiert ist): in Abb 5.1 dargestellt durch Button: „Gruppen“
- Feed Fragment (wenn **W15** implementiert ist): in Abb 5.1 dargestellt durch Button: „Feed“

Implementiert: **F19, F25, F71**

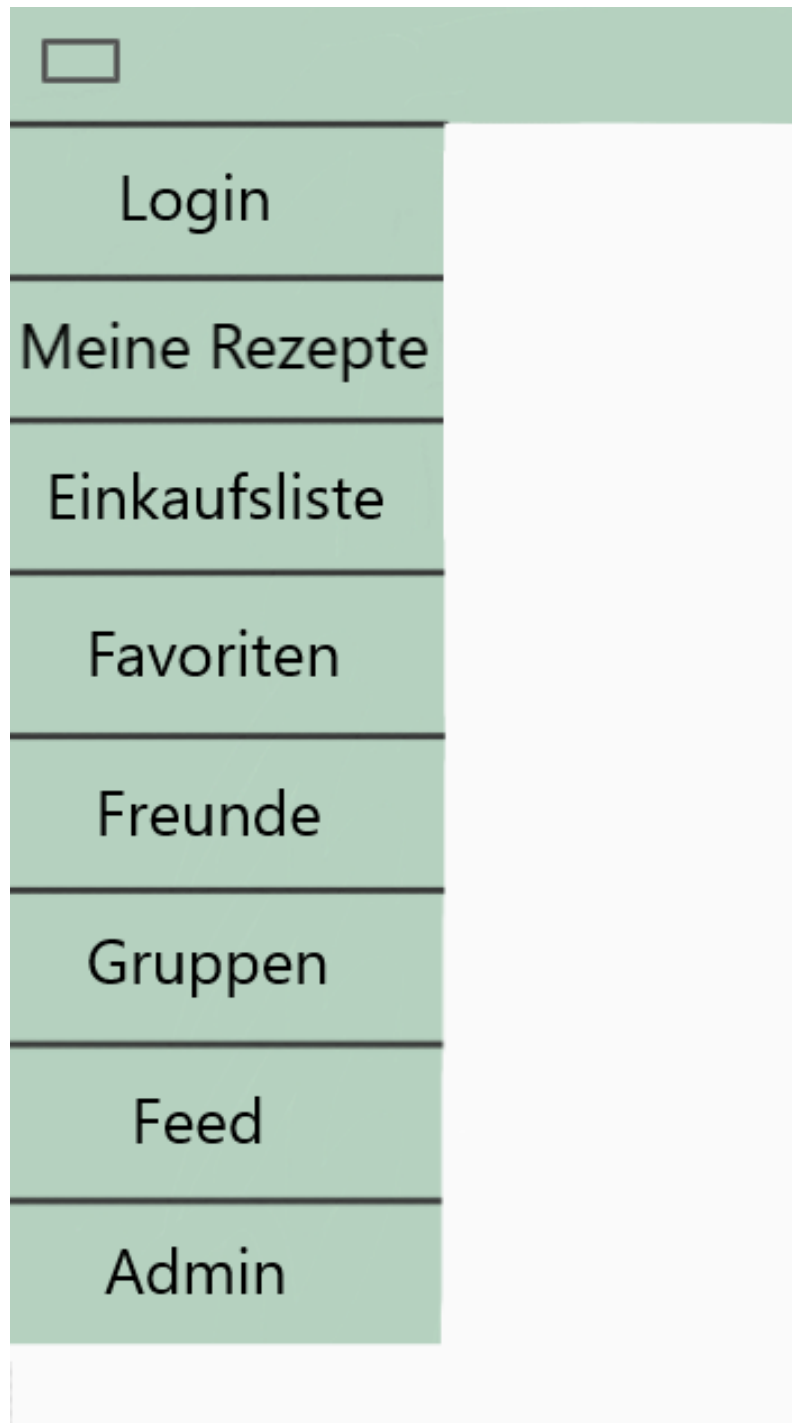
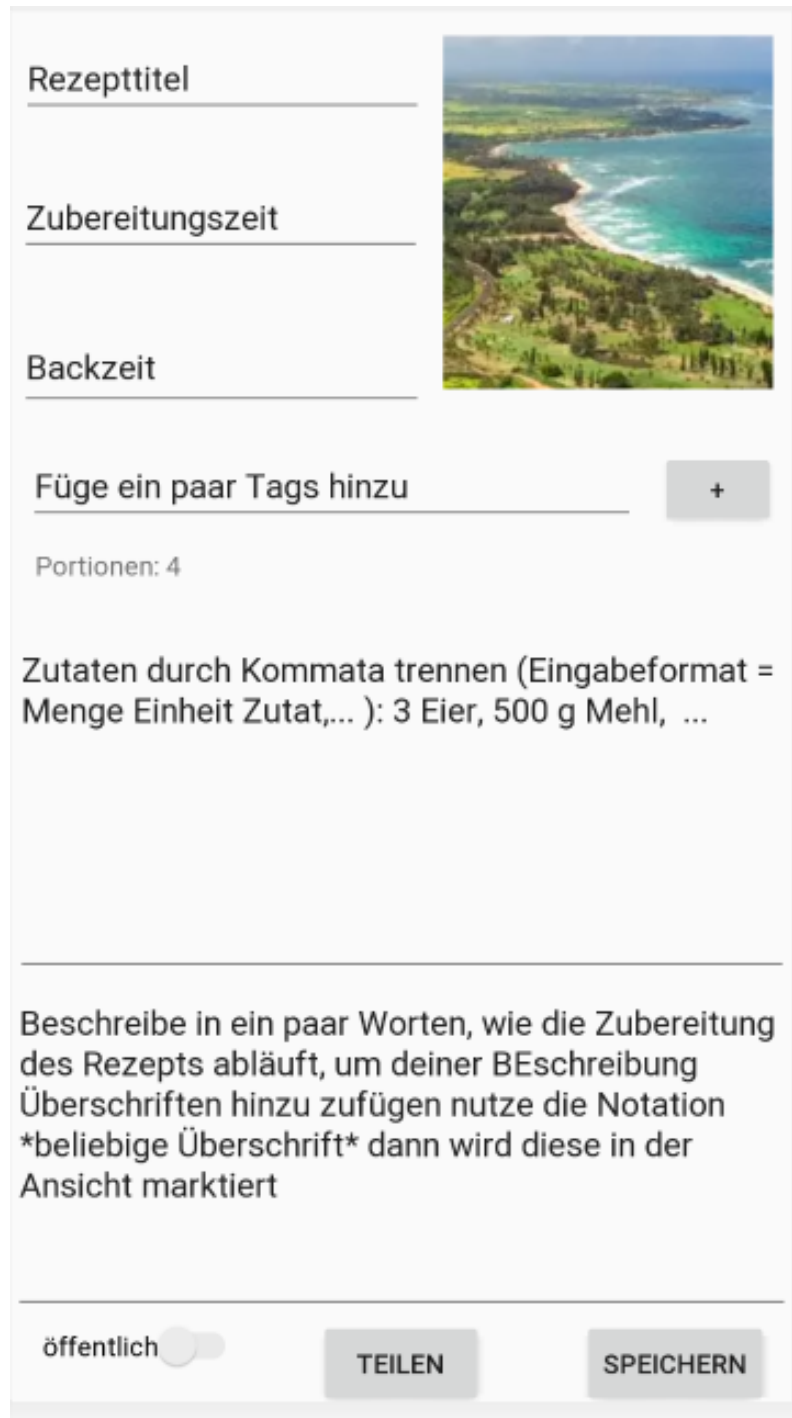


Abbildung 5.1: Toolbar mit angezeigtem Appmenü

Im Folgenden werden die einzelnen Fragments mitsamt ihren Inhalten beschrieben.

5.1.2 CreateRecipeFragment



The screenshot shows a mobile application form for creating a recipe. It features several input fields and a text area, all with light gray borders. On the right side, there is a square image placeholder showing a coastal landscape with a green hill, a sandy beach, and turquoise water. Below the form fields, there is a toggle switch for 'öffentlich' (public) and two buttons labeled 'TEILEN' (share) and 'SPEICHERN' (save).

Rezepttitel

Zubereitungszeit

Backzeit

Füge ein paar Tags hinzu

Portionen: 4

Zutaten durch Kommata trennen (Eingabeformat = Menge Einheit Zutat,...): 3 Eier, 500 g Mehl, ...

Beschreibe in ein paar Worten, wie die Zubereitung des Rezepts abläuft, um deiner Beschreibung Überschriften hinzu zufügen nutze die Notation *beliebige Überschrift* dann wird diese in der Ansicht markiert

öffentlich

TEILEN

SPEICHERN

Abbildung 5.2: CreateRecipeFragment

Implementiert: **F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F13, F14, F15, F16, F20, F26, F32, F28, F29, F30, F31, F74, F75**

- Rezepttitel-EditText: Hier kann der Nutzer einen Titel für sein Rezept eingeben

- Zubereitungszeit-EditText: Die Zeit, die zur Vorbereitung benötigt wird.
- Backzeit-EditText: Die Zeit, die das Gericht gebacken warten muss.
- Tagfeld-TextView: Hier kann der Nutzer Tags für sein Rezept hinzufügen.
- Portionen-TextView: Das Portionen Feld
- PortionenAnzahl-EditText: Die Anzahl an Portionen, die man bei der Zubereitung des Rezepts bekommt.
- Zutatenfeld-EditText: Alle Zutaten, die man zur Zubereitung des Gerichts braucht. Das Eingabeformat ist für jede Zutat "Menge Einheit Zutat", damit der Konvertierungsschritt zu einem öffentlich einsehbarem Rezept reibungslos abläuft.
- Zubereitung-Textfeld: Hier kann der Nutzer mit Text beschreiben, wie man das Rezept schrittweise zubereitet. Damit der Nutzer Überschriften in seiner Beschreibung hervorzuheben kann, gibt es die Möglichkeit die hervorzuhebenden Zeichen innerhalb zweier *-Symbole zu schreiben, wodurch beim Konvertieren zu einem öffentlichen Rezept diese Zeichen geparsed werden. Um andere Rezepte in seinem Text zu verlinken, kann der Nutzer den HTTP-Link eines Rezepts einfügen, um auf dieses Rezept verweisen. Diesem Link kann er ein Wort zuordnen, was statt des Links angezeigt wird.
- öffentlich-Switch: Falls der Nutzer sein Rezept veröffentlichen will kann er dies über den Schieberegler. Ist ein Rezept schon öffentlich und der Nutzer will dieses wieder als privat deklarieren ist das auch über diesen Regler machbar.
- Speichern-Button: Der Nutzer kann hierüber das Rezept in seiner Rezeptliste speichern. Falls der öffentlich-Schieberegler aktiviert ist, erfolgt die eine Vollständigkeitskontrolle und die Konvertierung.
- Teilen-Button: Der Nutzer hat die Möglichkeit hierüber das Rezept in andere Applikationen zu exportieren.
- Teilen-Button: Über den Speicher Button kann man das Rezept an externe Applikationen exportieren.

5.1.3 RecipeListFragment

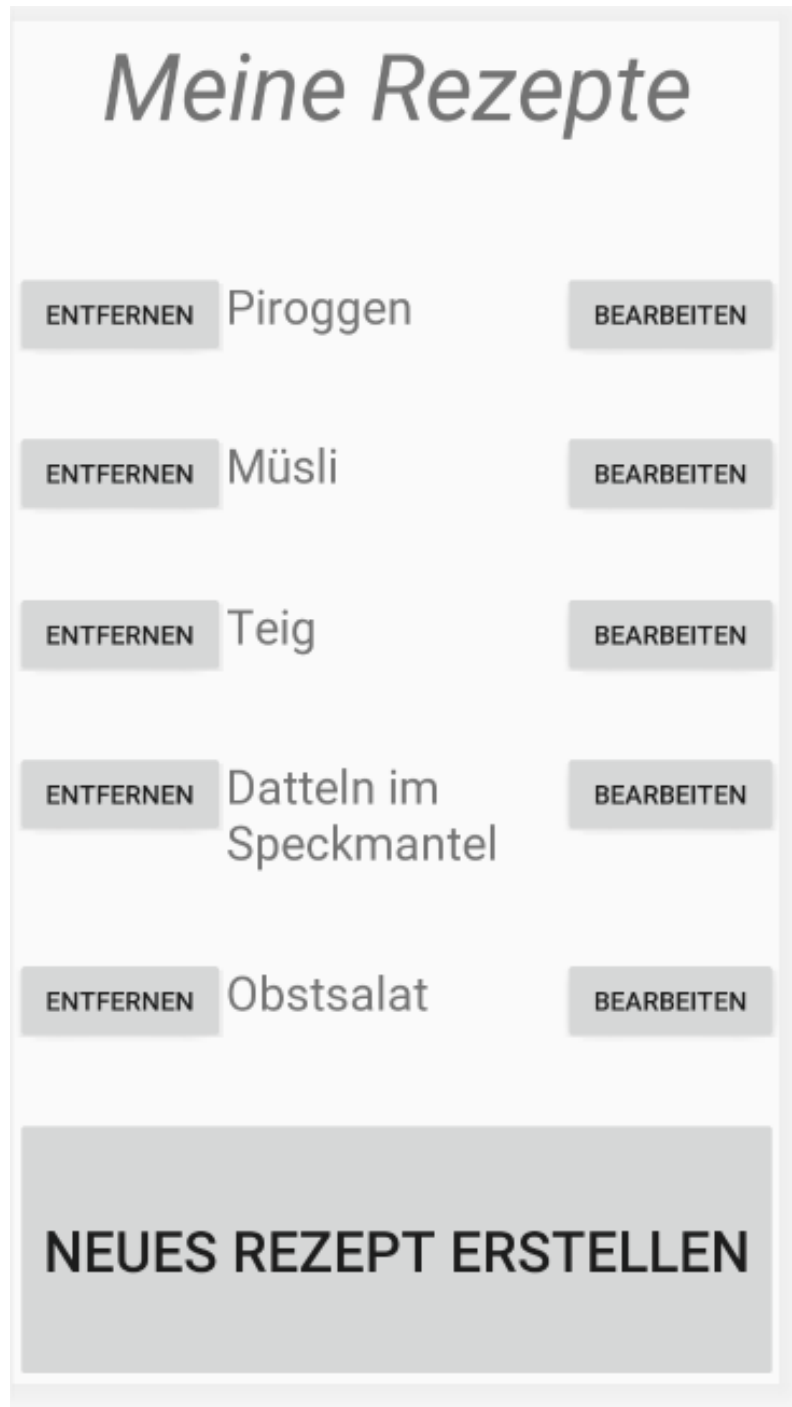


Abbildung 5.3: RecipeListFragment

Implements: **F17**, **F76**, **F77**

- header-TextView: Meine Rezepte Überschrift.
- Rezeptliste mit je:

- Bearbeiten-Button: Hierüber kann der Nutzer sein Rezept bearbeiten.
- Rezeptname-TextView: Anzeige des Rezeptnamens.
- Entfernen-Button: Hierüber kann das jeweilige Rezept gelöscht werden
- Neues Rezept Erstellen-Button: Hierüber kann der Nutzer ein neues Rezept erstellen

5.1.4 AddEditCommentFragment

Neuer Kommentar

Das Rezept ist super
lecker :D. Bei mir braucht der
Kuchen 10 Minuten länger im
Ofen.

KOMMENTIEREN

Abbildung 5.4: AddEditCommentFragment

Implementiert: **F37, F90**

- header-TextView: Hier wird "Neuer Kommentar angezeigt"
- Eingabe-Textfeld: Hier kann der Nutzer ein Kommentar in textlicher Form schreiben.
- Kommentieren-Button: veröffentlicht den Kommentar damit dieses in dem "Recipe-DisplayFragment" gesehen werden kann

5.1.5 DisplaySearchListFragment

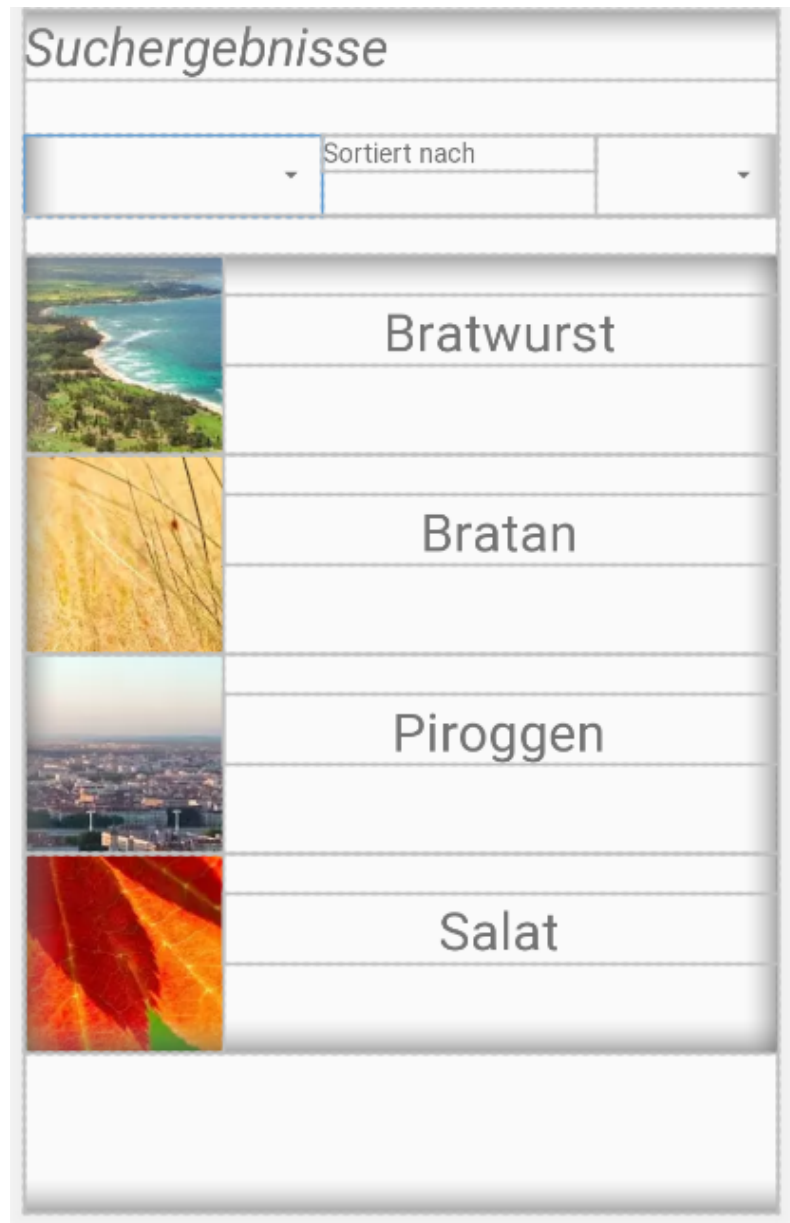


Abbildung 5.5: DisplaySearchListFragment

Implementiert: **F53, F54, F55, F58, F88**

- Auswahlbox-Spinner: Der Nutzer hat die Auswahl zwischen absteigend und aufsteigend.
- Auswahlbox-Spinner: Der Nutzer hat die Auswahl zwischen verschiedenen Suchfiltern.
- Rezeptliste-ScrollView: Hier warten alle Suchergebnisse sortiert angezeigt. Der Nutzer kann die jeweiligen Rezepte auswählen, und wird dann auf das "RecipeDisplayFragment" weitergeleitet.

5.1.6 UserSearchFragment

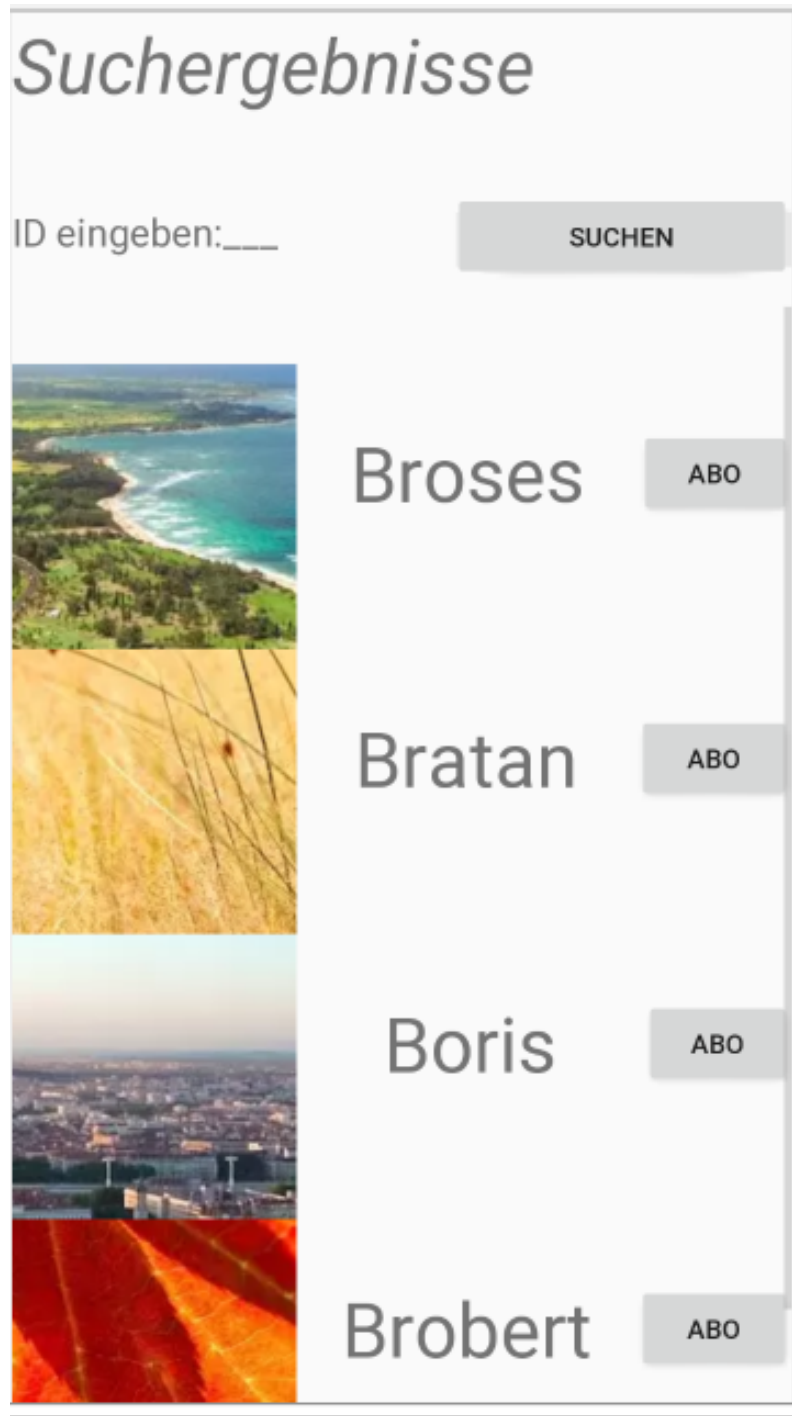


Abbildung 5.6: UserSearchFragment

Implementiert: **F48, F59, F60, F61, F62, F98**

- header-TextView: "Suchergebnisse"
- ID-EditText: Hier kann die Nutzer-ID des gesuchten Nutzers eingetragen werden.

- Suchen-Button: Der Nutzer wählt den Suchen Button und bekommt eine Ergebnisliste von Nutzern angezeigt, die auf die eingetragene ID passen.
- Nutzerliste: Hier werden alle Suchergebnisse angezeigt.
- Abo-Button: Hierüber kann man einem Nutzer abonnieren.

5.1.7 PublicRecipeSearchFragment

Tippe um nach einem Rezepttitel zu suchen

Füge Zutaten hinzu, nach denen du suchen willst

Karotten

Butter

HINZUFÜGEN

Suche nach Mindestbewertung

★ ★ ★ ★ ★

Mit Tags suchen

+

Zeit: 10min.

- süß

- vegan

SUCHEN

Abbildung 5.7: PublicRecipeSearchFragment

Implementiert: **F51, F52, F56, F57**

- Titelsuche-Textfield: Der Nutzer kann hier einen Rezepttitel eingeben, nachdem gesucht werden soll
- Zutatenhinzufügen-Textfield & Button: über das Textfeld kann der Nutzer eine Zutat eingeben und über den Button in eine Liste eintragen
- Zutatenliste-TextView: Hier werden alle eingegebenen Zutaten gespeichert, mit denen gesucht werden soll
- Zeit-TextView: Zeit Anzeige
- Zeitminuten-EditText: Minuten Eingabe
- Tag-suchen-Button: Hierüber kommt der Nutzer zur Suche mit Tags
- Mindestbewertung: Textanzeige und Rating-Bar-Button
- Tag-Anzeige: Hier werden die Tags angezeigt, die zur Suche genutzt werden sollen
- Suchen-Button: Wenn der Nutzer seine Suchoptionen eingegeben hat, kann er über den Button die Suche starten

5.1.8 FeedFragment





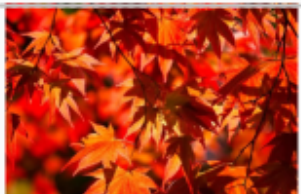

Feed	
	Eintopf
	Zweitopf
	Linseneintopf mit Spätzle
	Linseneintopf
	Linseneintopf mit Spätzle
	Linseneintopf ohne Spätzle

Abbildung 5.8: FeedFragment

Implementiert: **F70, F72**

- header-TextView: "FeedÄnzeige"
- RezeptListe: Hier werden alle Rezepte des Feeds angezeigt und können vom Nutzer ausgewählt werden. Dann kommt der Nutzer auf das jeweilige "RecipeDisplayFragment".

Der Feed zeigt die aktuellsten Rezepte an, die veröffentlicht wurden. Bei jedem neuen Laden des Feeds wird die Ansicht aktualisiert.

5.1.9 ProfileDisplayFragment

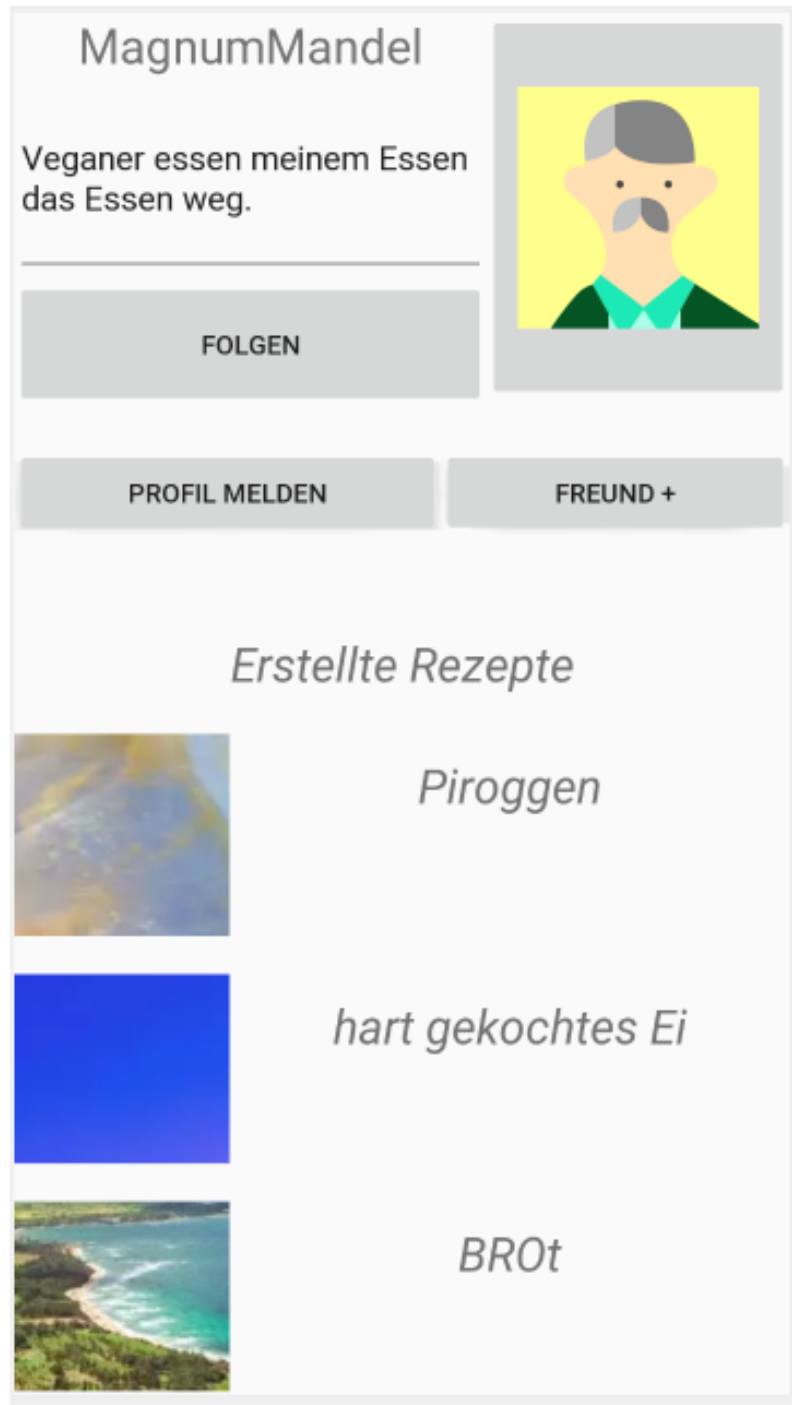


Abbildung 5.9: ProfileDistplayFragment


Implementiert: **F47**, **F63**, **F91**

- Namensfeld-TextView: Hier wird die Benutzer-ID angezeigt
- Folgen-Button: Hier kann man dem angezeigten Nutzer folgen.

- Beschreibung-TextView: Hier wird angezeigt, wie sich der Nutzer in ein paar wenigen Sätzen beschrieben hat.
- Profilbild-ImageView: Der Benutzer kann ein Profilbild anzeigen lassen. Falls er dies nicht will, wird ein Standardbild angezeigt.
- Profilmelden-Button: falls ein Profil unangemessene Inhalte enthält, kann dieses hierdurch gemeldet werden, worauf es manuell von einem Admin geprüft wird.
- Freund+-Button: Nutzer können hierdurch anderen Nutzern eine Freundschaftsanfrage schicken
- Erstellte Rezepte-Liste: Hier werden alle öffentlich gestellten Rezepte des Nutzers angezeigt. Wenn man eins auswählt, kommt man direkt zum jeweiligen "RecipeDisplayFragment".

5.1.10 ProfileEditFragment

Gib eine persönliche ID ein



schreibe etwas über dich

Meine Vorbilder sind Christopher KOHLumbus und Chris BREAD

LOGIN-DATEN ÄNDERN

SPEICHERN

PROFIL LÖSCHEN

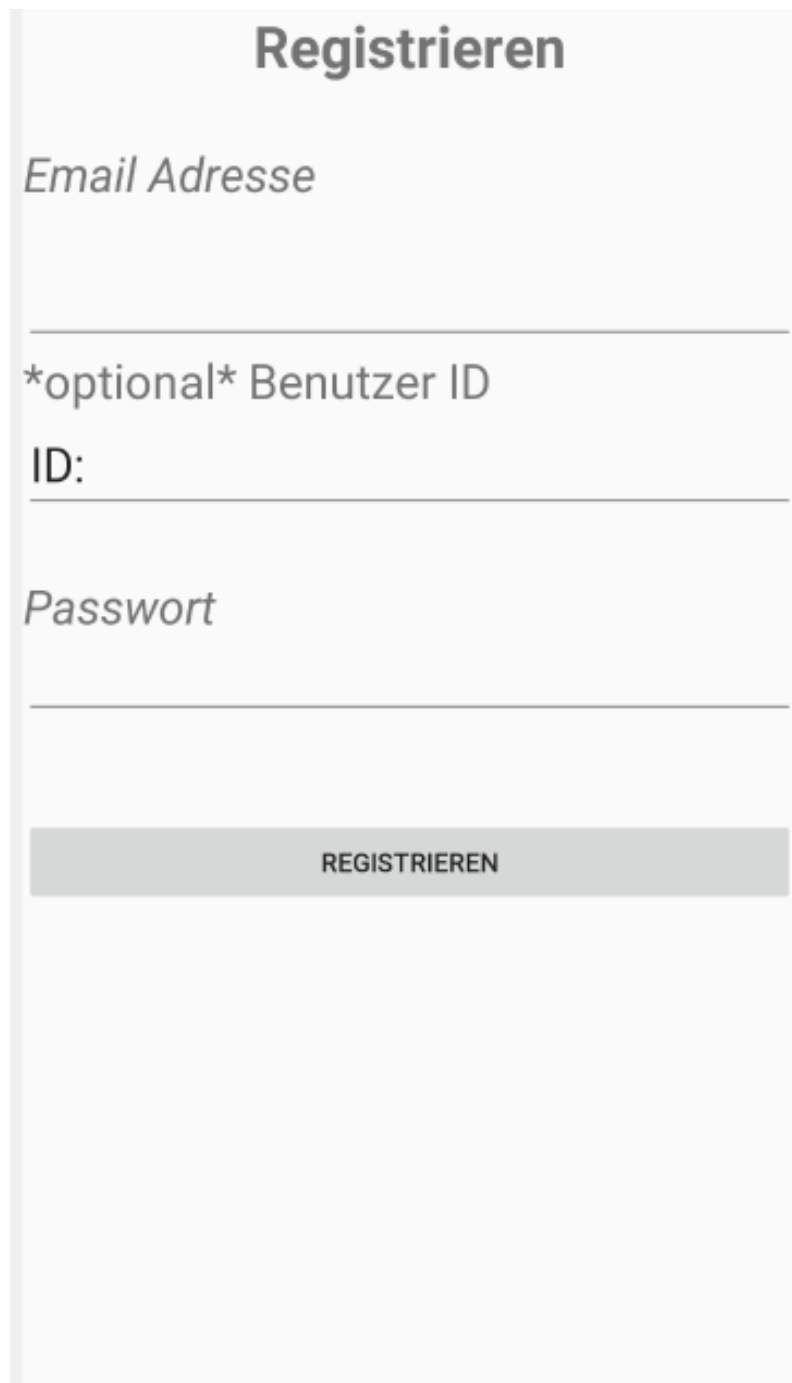
Abbildung 5.10: ProfileEditFragment

Implementiert: **F41, F43, F44, F45, F46, F42**

- BenutzerID-EditView: Hier kann die Benutzer-ID eingetragen werden
- Profilbild-ImagView: Hier kann der Nutzer ein Profilbild hochladen und ändern.
- Beschreibung-Textfield: Hier kann sich der Nutzer in ein paar Worten selbst beschreiben. Dies ist auch in der öffentlichen Ansicht des Nutzerprofils sichtbar.
- Login-Daten-Ändern-Button: Falls der Nutzer seine Anmelde-Daten ändern möchte, kann er dies mit diesem Button machen
- Speichern-Button: der Nutzer speichert über diesen Button die Änderungen für sein Profil.
- löschen-Button: Hierüber entfernt der angemeldete Nutzer sein Profil, wodurch alle seine Daten, bis auf die privaten Rezepte, gelöscht werden.

Auf diesem Fragment kann der Nutzer alle öffentlich angezeigten Informationen ändern. Will er seine Daten zum Einloggen ändern erreicht er dies über den Login Daten ändern Button.

5.1.11 RegistrationFragment



The image shows a registration form titled "Registrieren". It contains three input fields: "Email Adresse", "*optional* Benutzer ID", and "Passwort". Each field is followed by a horizontal line. Below the "Passwort" field is a grey button labeled "REGISTRIEREN".

Registrieren

Email Adresse

optional Benutzer ID

ID:

Passwort

REGISTRIEREN

Abbildung 5.11: RegistrationFragment

Implementiert: **F38, F39**

- header-TextView:"RegistrierenKopfzeile
- Email-Adresse-Textfield: Hier kann der Nutzer seine Email Adresse angeben
- Passwort-EditText:Hier kann der Nutzer sein Passwort angeben

- Benutzer ID -EditText: Hier kann der Nutzer eine Wunsch ID eintragen. Macht er dies nicht wird bei der Registrierung automatisch eine erstellt.
- Registrieren-Button: Wenn der Nutzer seine Daten eingetragen hat, kann er über diesen Button die Registrierung abschließen

Bei der Registrierung hat der Nutzer drei Eingabefelder gegeben. Die Pflichtfelder für eine erfolgreiche Registrierung sind die Email-Adresse und Passwort. Gibt der Nutzer diese gar nicht oder fehlerhaft an, wird der Prozess nicht weitergeführt. Des Weiteren kann der Nutzer eine BenutzerID für sich festlegen. Die Benutzer-ID und das Passwort können im Nachhinein noch geändert werden und sind nicht permanent. Die Email hingegen ist ab der erfolgreichen Registrierung nicht mehr änderbar.

5.1.12 ChangePasswordFragment

The image shows a mobile application screen for changing a password. At the top, the title "Neues Passwort eingeben" is displayed in a large, italicized, gray font. Below the title is a password input field represented by ten black dots. A horizontal line separates the input field from the bottom section. In the bottom section, there is a gray rectangular button with the text "PASSWORT ÄNDERN" in all caps.

Abbildung 5.12: ChangePasswordFragment

Implementiert: **F40**

- Passwort-Textfield: Hier kann der angemeldete Nutzer sein neues Passwort eingeben.
- Passwort ändern - Button: Hierüber bestätigt der Nutzer sein neues Passwort

Da die Applikation die Account Funktionalität über Firebase verwaltet, werden die eingetragenen Daten direkt weitergegeben und so wenig wie möglich selbst gehalten.

5.1.13 LoginFragment

The image shows a login form titled "Einloggen". It contains two input fields: "Email" and "Passwort". Below the "Passwort" field is a "LOGIN" button. Underneath the button is a link that says "Ich habe noch keinen Account". At the bottom of the form is a "REGISTRIEREN" button.

Einloggen

Email

Passwort

LOGIN

Ich habe noch keinen Account

REGISTRIEREN

Abbildung 5.13: LoginFragment

Implementiert: **F50**

- Email-Textfield: Hier kann der Nutzer seine Email angeben
- Passwort-Textfield: Hier kann der Nutzer sein Passwort angeben
- Login-Button: Wenn der Nutzer seine Daten eingetragen hat, kann er sich über diesen Button einloggen.
- Registrieren-Button: Falls der Nutzer noch kein Account hat, kann er auf den Registrieren Button klicken, um auf das RegistrationFragment geleitet zu werden.

5.1.14 EditTagFragment

The screenshot displays the 'EditTagFragment' interface. At the top, the title 'Tagliste' is centered in a large, bold font. Below the title is a list of six tags, each preceded by a checkbox and followed by a grey button labeled 'ENTFERNEN'. The tags are: 'vordefinierter Tag 1', 'vordefinierter Tag 2', 'vordefinierter Tag 3', 'vordefinierter Tag 4', 'ersteller Tag 1', and 'erstellter Tag 2'. Below the list is a grey button with a '+' sign. Underneath the button is a text label: 'Persönliche Tags kannst du später deinen eigenen Rezepten hinzufügen'. At the bottom of the form is a wide grey button labeled 'SPEICHERN'.

Tag	Checkbox	Action
vordefinierter Tag 1	<input type="checkbox"/>	ENTFERNEN
vordefinierter Tag 2	<input type="checkbox"/>	ENTFERNEN
vordefinierter Tag 3	<input type="checkbox"/>	ENTFERNEN
vordefinierter Tag 4	<input type="checkbox"/>	ENTFERNEN
ersteller Tag 1	<input type="checkbox"/>	ENTFERNEN
erstellter Tag 2	<input type="checkbox"/>	ENTFERNEN

+

Persönliche Tags kannst du später deinen eigenen Rezepten hinzufügen

SPEICHERN

Abbildung 5.14: EditTagFragment

Implementiert: **F73**

- Tag-Textfield: Hier kannn der Nutzer seine neuen Tags eingeben

- +-Button: Falls der Nutzer neue Eingabefelder braucht, kann er über den Button diese anfordern
- Speichern-Button: Wenn der Nutzer seine Tags eingetragen hat, kann er über den Speichern Button die Erstellung abschließen

5.1.15 SearchWithTagsFragment

Tags hinzufügen

FERTIG

Tags zum Hinzufügen einfach auswählen

- ☐ süß
- ☐ unkompliziert
- ☐ brunch
- ☐ vegetarisch
- ☐ Vegan

Abbildung 5.15: SearchWithTagsFragment

- Tag-Auflistung: Hier werden alle vom User gesammelten Tags aufgelistet. Diese kann der Nutzer über eine Auswahlbox markieren.
- Fertig-Button: Wenn der Nutzer seine gewünschten Tags gesammelt hat, mit denen er suchen will, kann er über den Fertig- Button zurück zur Rezeptsuche gehen, wobei die Tags mitgenommen werden.

5.1.16 AdminFragment



Abbildung 5.16: AdminFragment

- **Nutzer-Auflistung:** Hier sieht der Admin alle gemeldeten Nutzer. Er kann sich die Profile ansehen und einzelne nicht erlaubte Einträge (Bild, ID, Beschreibung) entfernen
- **Rezept-Auflistung:** Hier sieht der Admin eine Liste von allen gemeldeten Rezepten. er kann diese manuell überprüfen und direkt über den Button entfernen.

- Gruppen-Auflistung: Hier sieht der Admin alle Gruppen, die gemeldet wurden. Falls ein Gruppenname nicht erlaubt ist, kann der Admin über den Button die Gruppe auflösen.

Der Admin hat über die Applikation die Möglichkeit Rezepte und Gruppen direkt zu entfernen. Falls ein Nutzer ein unpassenden Namen, Beschreibung, oder Bild hat muss sich der Admin dieses manuell ansehen und entscheiden, ob nur das jeweilige Feld aus der Datenbank austrägt, oder den Nutzer komplett entfernt.

5.1.17 ShoppingListDisplayFragment

Einkaufsliste

20 Kilogramm Zucker	<input checked="" type="checkbox"/>
Banana 15 stk.	<input checked="" type="checkbox"/>
20 Gramm Mehl	<input type="checkbox"/>
Freshavacadoo	<input type="checkbox"/>
Backofenpommes 500 g.	<input type="checkbox"/>
20 Gramm Mehl	<input type="checkbox"/>

LEEREN

Abbildung 5.17: ShoppingListDisplayFragment

Implementiert: **F21**, **F24**

- Zutaten-Auflistung: Hier werden alle Zutaten, die der Nutzer seiner Einkaufsliste hinzugefügt an angezeigt. Diese kann er über eine Checkbox abhaken.
- leeren-Button: Falls die Liste nicht mehr benötigt wird, kann der Nutzer diese über

den Button leeren.

5.1.18 FriendListFragment

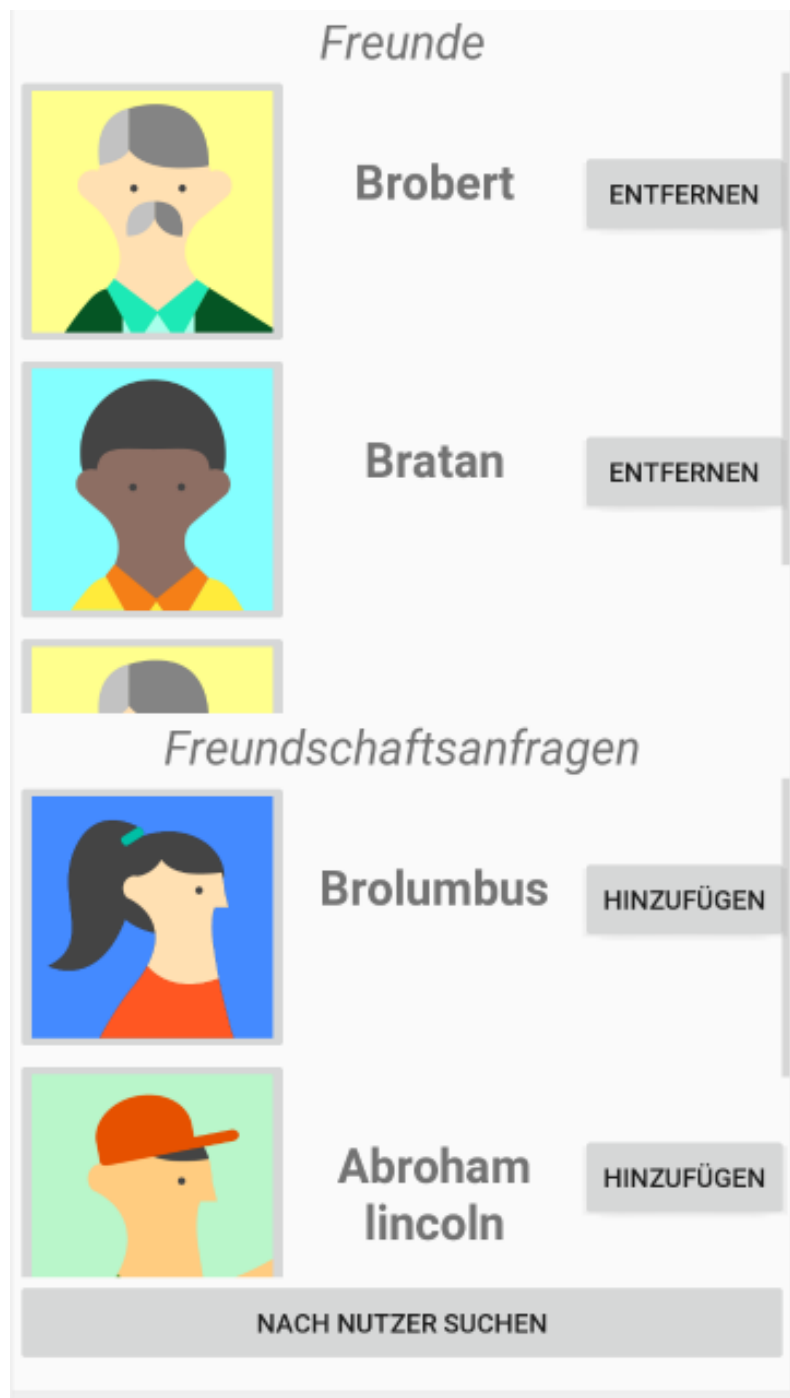


Abbildung 5.18: FriendListFragment

Implementiert: **F64, F65, F66, F92**

- Freundesauflistung: Hier werden alle schon bestätigten Freunde des Nutzers angezeigt. Wenn man eine Person anklickt, kommt man zur jeweiligen Profilansicht.
- Freundschaftsanfragen: Hier werden alle Nutzer angezeigt, die eine Freundschaftsanfrage gesendet haben. Wenn man eine Person anklickt, kommt man zur jeweiligen Profilansicht.
- Suchen-Button: Hierüber kommt man zu einer Ansicht, die eine Benutzersuche bereitstellt
- Entfernen-Button: Der Freund wird aus der Freundesliste entfernt
- Hinzufügen-Button: Der Freund wird der Freundesliste hinzugefügt

5.1.19 GroupMemberListFragment

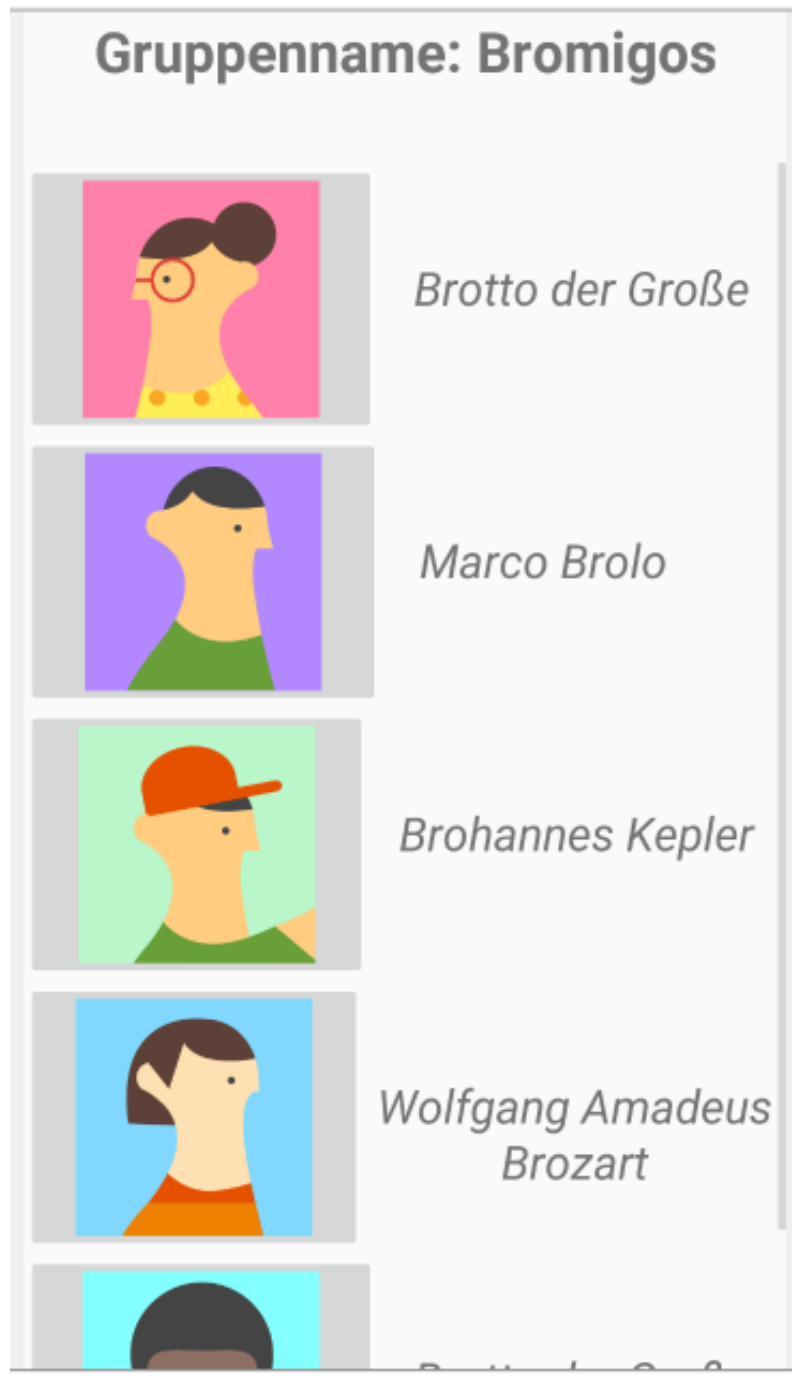


Abbildung 5.19: GroupMemberListFragment

- Gruppenname-TextView: Name Der Gruppe
- Gruppenmitglieder-Auflistung: Hier werden alle Mitglieder der Gruppe aufgelistet. Wenn man sie auswählt, kommt man zur jeweiligen Profilansicht.

5.1.20 CreateGroupFragment

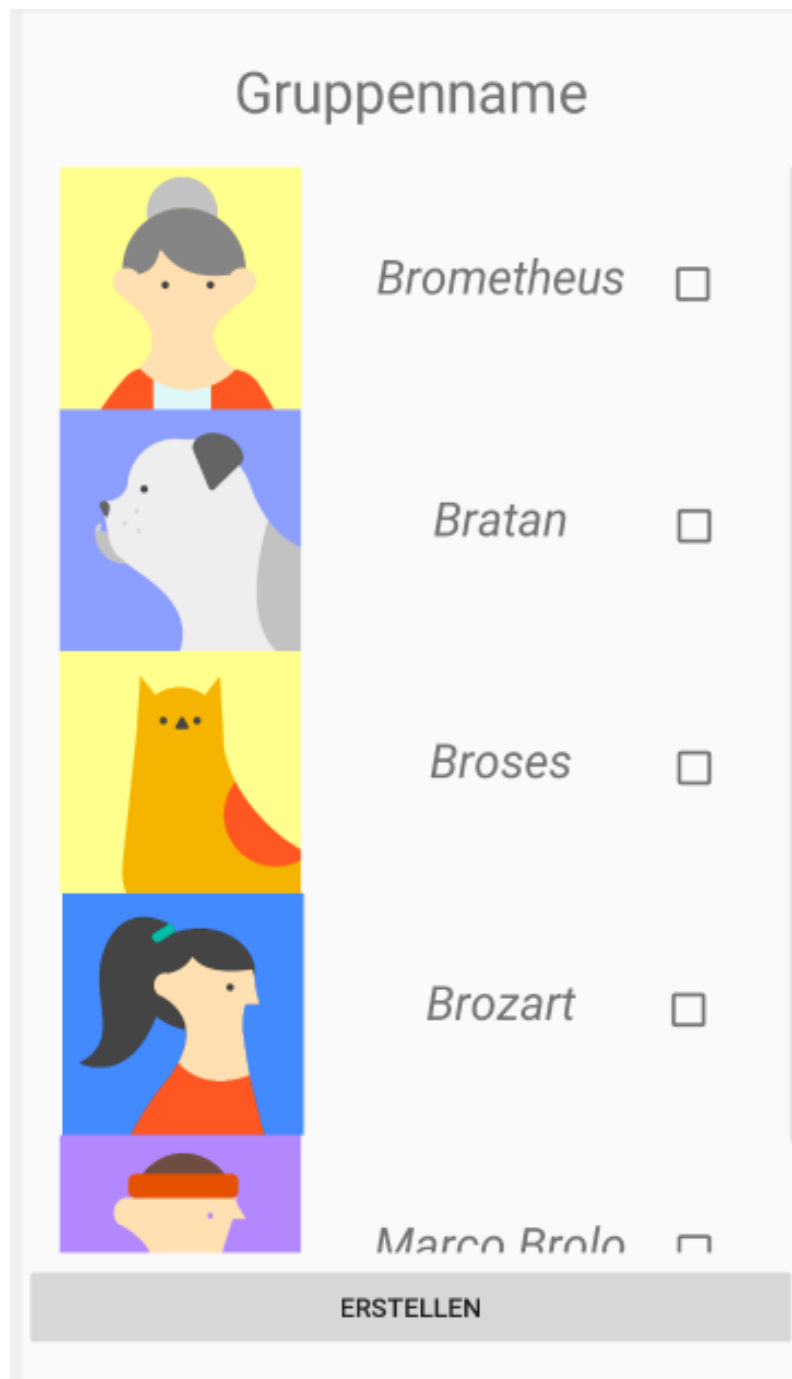


Abbildung 5.20: CreateGroupFragment

Implementiert: **F67, F68, F93, F94** Wir müssen auch sorgen, dass im nachhinein Mitglieder hinzugefügt werden können, dafür kann ja dieses Fragment recycled werden

- Gruppenname-TextView: Gruppenname wird angezeigt
- Freunde-Auflistung: Hier werden alle Freunde des Nutzers aufgelistet. Über die Check-

box auf der rechten Seite kann man eine Gruppe beim Erstellen der Gruppe hinzufügen. Drückt man auf einen Nutzer, kommt man nicht auf die jeweilige Profilseite.

5.1.21 FriendGroupFragment

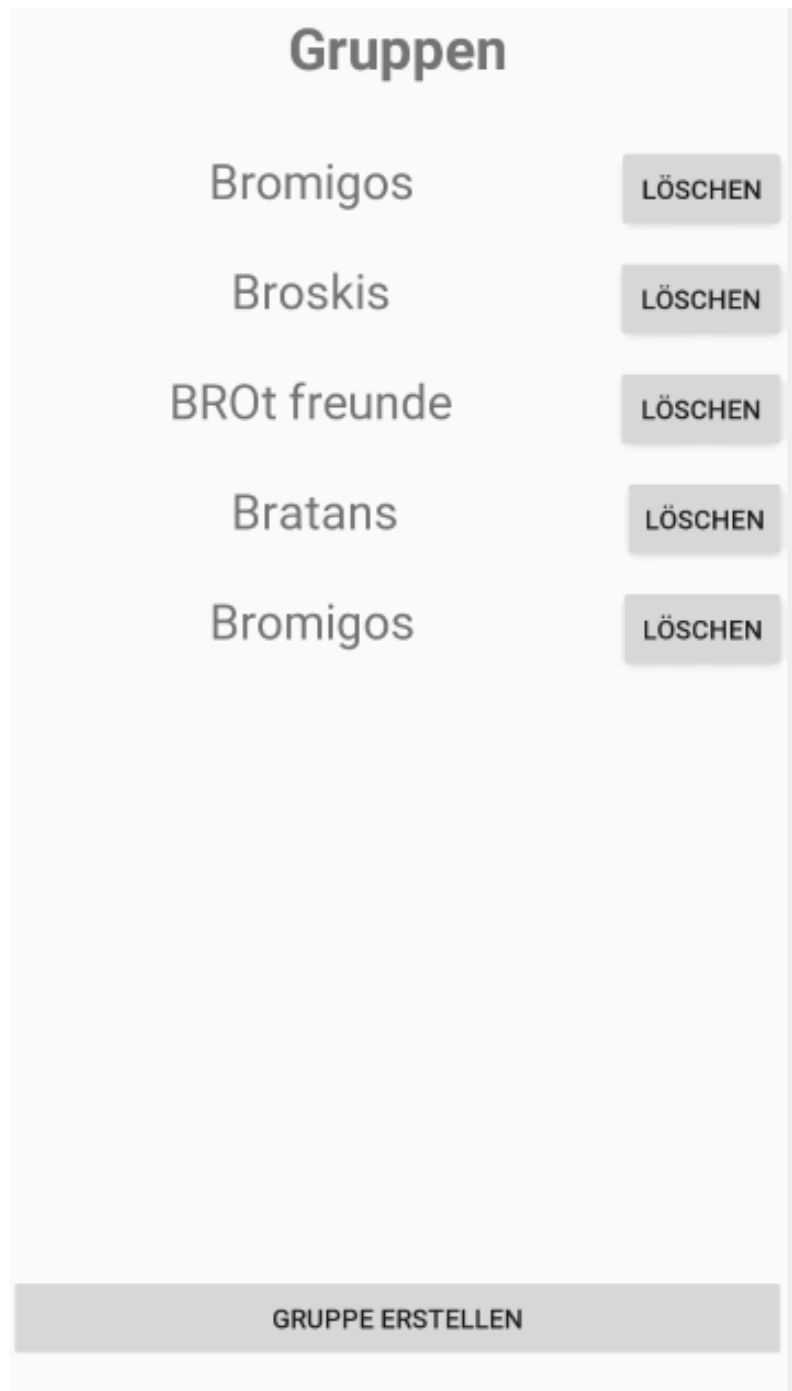


Abbildung 5.21: FriendGroupFragment

Implementiert: **F68, F95**

- Gruppen-Auflistung: Hier werden alle Gruppen, in denen der Nutzer Mitglied ist, aufgelistet.
- Löschen-Button
 - Falls er der Admin ist, wird die Gruppe komplett entfernt
 - Falls er nicht der Admin ist, entfernt er nur sich aus der Gruppe
- Gruppe-erstellen-Button: Hierüber kommt der Nutzer auf die Gruppe-Erstellen Ansicht.

5.1.22 GroupFragment

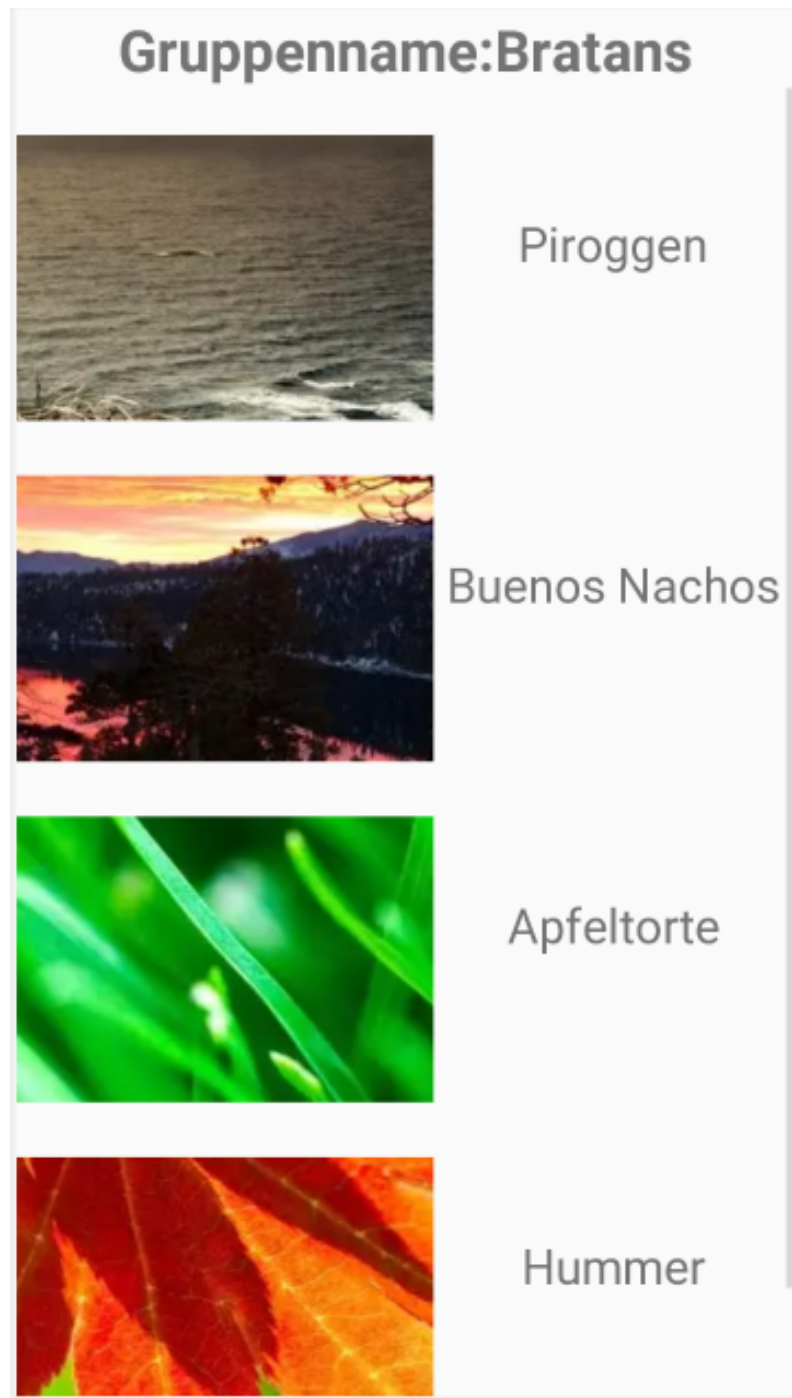


Abbildung 5.22: GroupFragment

Implementiert: **F69**

- Gruppenname: Hier wird der Gruppenname angezeigt und wenn man darauf klickt, kommt man auf die Gruppenmitglieder Ansicht
- Rezept-Auflistung: Hier werden alle geteilten Rezepte aufgelistet. Wenn man diese

anklickt, kommt man auf die Rezept bearbeiten Ansicht. Da man private Rezepte in Gruppen teilen kann, können diese unvollständig sein. Daher kann man diese nur in der bearbeiten Ansicht anschauen.

5.1.23 RecipeDisplayFragment

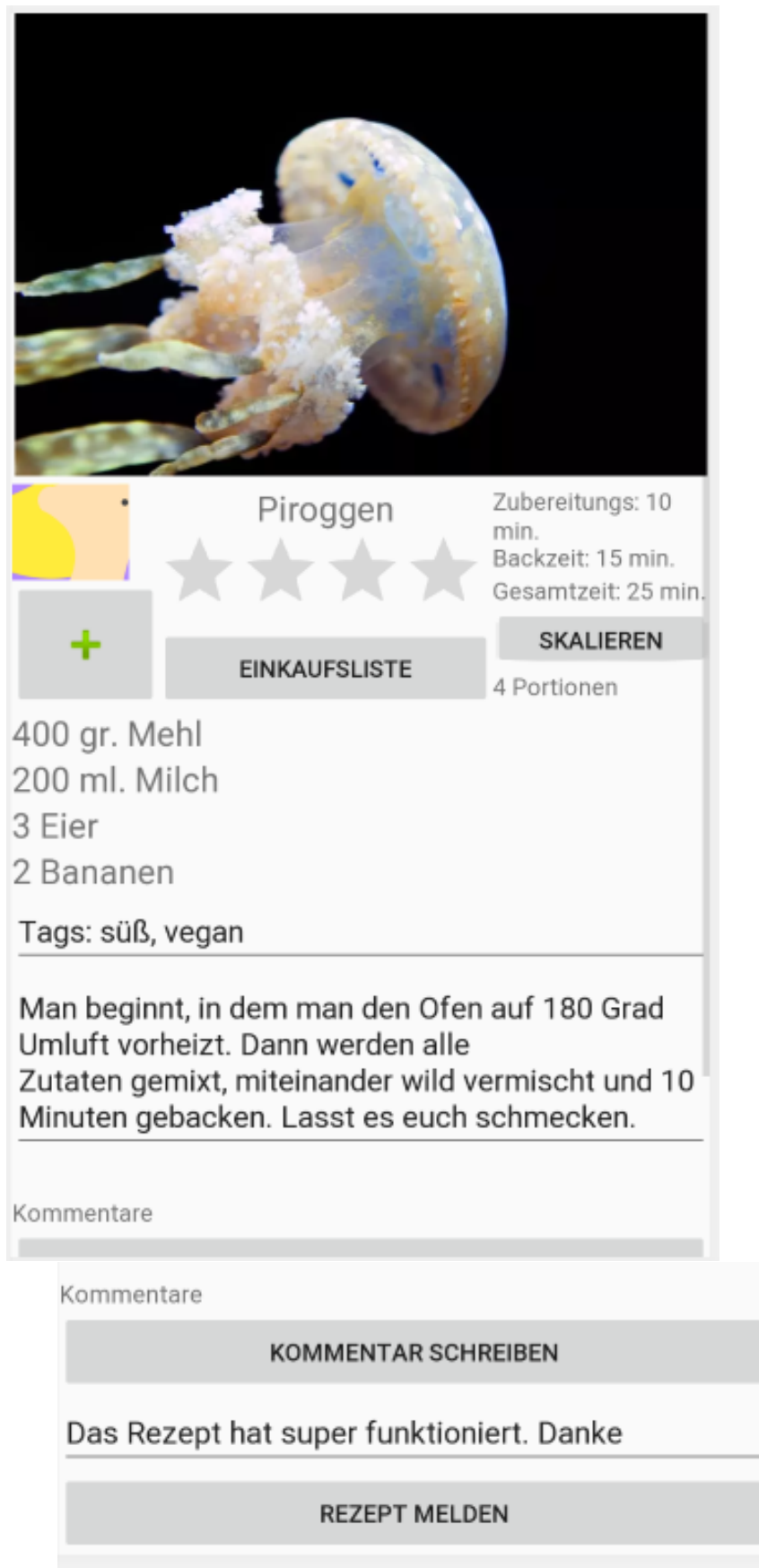


Abbildung 5.23: RecipeDisplayFragment

Implementiert: **F18, F22, F23, F33, F34,F35, F36**

- Rezeptbild: Hier wird das Rezeptbild angezeigt
- Rezepttitel: Der Name des Rezepts
- Bewertungsfeld: Hier kann jeder angemeldete Nutzer eine Bewertung für das Rezept angeben
Falls das Rezept schon vom Nutzer bewertet wurde, wird diese Bewertung angezeigt
- Zubereitungszeit: Hier steht die Zeit, die für die Zubereitung benötigt wird
- Backzeit: Hier steht die Zeit, die das Gericht gebacken werden muss
- Gesamtzeit: Diese ergibt aus der Zubereitungs- und der Backzeit
- Einkaufsliste-Button: Hierüber kann der Nutzer die Zutaten in seine Einkaufsliste eintragen. Daraufhin kommt er auch in die Einkaufslisten Ansicht
- skalieren-Button: Wenn der Nutzer eine neue Menge eingibt, werden die Portionen umgerechnet.
- Profilbild: Hier wird das Profilbild des Rezepterstellers angezeigt. Wenn man dieses anklickt, kommt man auf die Profilansicht des Erstellers.
- Hinzufügen-Button: Wenn man diesen anklickt, wird das Rezept in der Favoritenliste hinzugefügt
- Zutaten-Liste: Hier werden alle für das Rezept benötigten Zutaten aufgelistet
- Tag-Liste: Hier werden alle von dem Autor eingetragenen Tags angezeigt
- Beschreibung: Hier steht die Zubereitungsbeschreibung für das Rezept
- Kommentar schreiben - Button: Hier kann ein angemeldeter Nutzer ein Kommentar zu dem jeweiligen Rezept verfassen. Hierüber wird man auf das AddEditComment-Fragment weitergeleitet. Klick man auf ein Kommentar, das von einem selbst ist, wird man auf das AddEditCommentFragment weitergeleitet.
- Kommentarliste: Hier werden alle zu dem Rezept schon verfassten Kommentare aufgelistet.
- Melden-Button: Hierüber kann das Rezept gemeldet werden. Daraufhin erscheint das Rezept in der Liste des Admins, der dieses dann manuell überprüfen kann.

5.1.24 FavouriteFragment

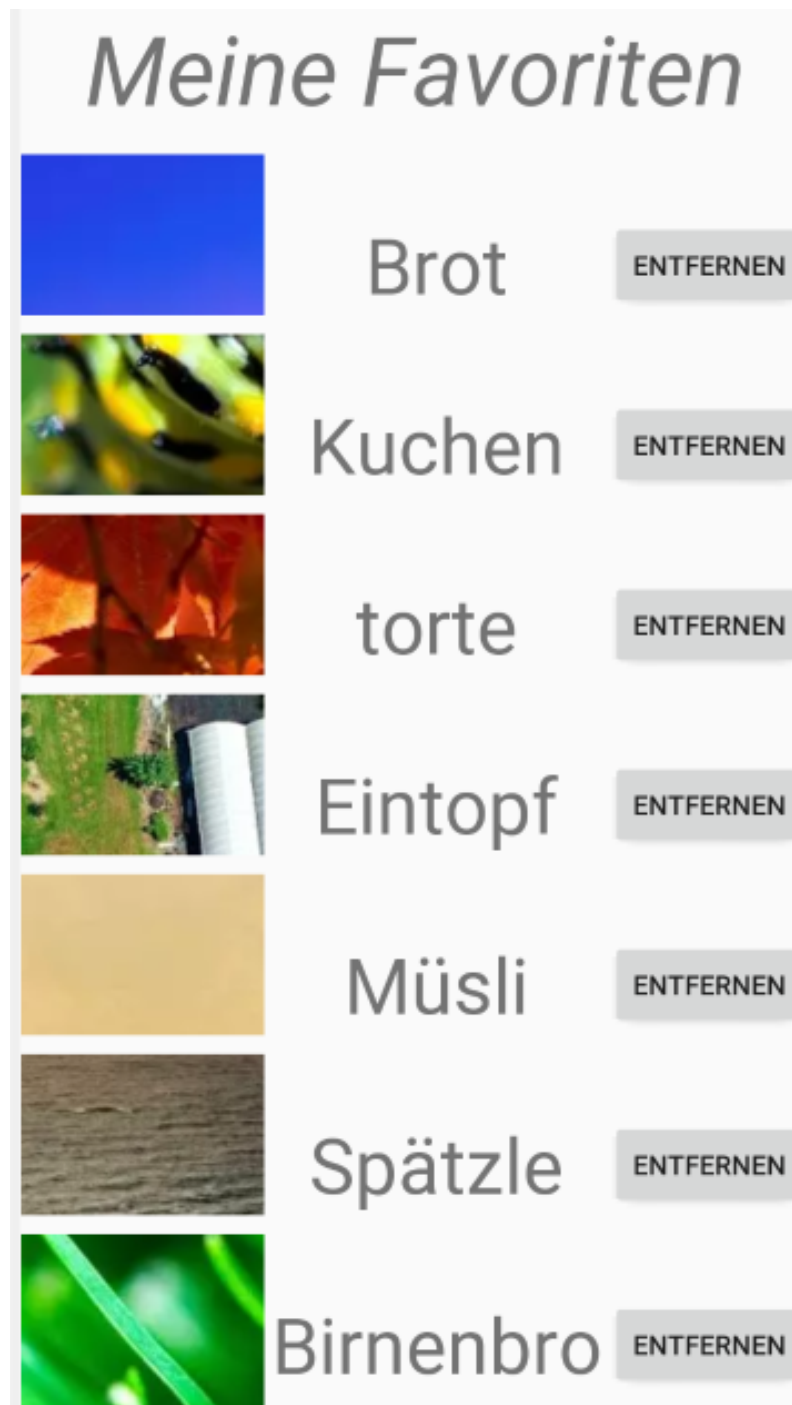


Abbildung 5.24: FavouriteFragment

- Rezeptliste: Hier werden alle favorisierten Rezepte angezeigt. Diese kann der Nutzer einzeln aus der Favoritenliste entfernen. Wenn auf ein Rezept geklickt wird, kommt man auf die jeweilige Rezeptansicht des Rezepts.
- entfernen-Button: Das Rezept wird gelöscht

5.2 ViewModel

Im Folgenden werden die den Fragments zugehörigen ViewModel-Klassen beschrieben. Vorweg sei gesagt, dass alle ViewModels in der onCreate()-Methode der Main-Activity eingebunden werden müssen. Jedes einzelne ViewModel wird außerdem in dem ihm zugehörigen Fragment instanziiert. Somit wird beim allerersten Aufruf des Fragments sein ViewModel erstellt, sodass dieses Daten für das Fragment bereitstellen kann. Da Die Viewmodel an einigen Stellen einen redundanten Aufbau besitzen, werden die wichtigen Eigenheiten des ViewModels in der folgenden Beschreibung hervorgehoben. Ist ein Repository eingezeichnet, so sind die darin beschriebenen Methoden aus Übersichtsgründen nur auf das jeweilige ViewModel abgestimmt. Im Entwurf wird es dennoch nur eine Klasse geben, das alle vorkommenden Methoden der ViewModel beinhaltet. Der Ablauf ist jeweils immer gleich, dass ein Fragment sein zugehöriges ViewModel erstellt, worüber der Zugriff auf das Repository stattfindet. Damit ist die Oberfläche der Applikation sauber von der Datenquelle getrennt. Die Relation zwischen dem jeweiligen ViewModel und dem Fragment ist über ein Observer Pattern beschrieben. Exemplarisch ist das im DisplaySearchListViewModel näher erläutert. Dieses Observer-Pattern soll für die weiteren ViewModel implizit angenommen werden, da es aus Redundanzgründen und der Übersichtlichkeit nicht modelliert wurde. Ähnlich redundant ist der Aufbau der Fragmente, der prinzipiell den Klassen entspricht, wie oben beschrieben (Siehe Kapitel 5.1). Daher wurden die Fragmente nur skizzenhaft modelliert und nur eingefügt, wo es der Übersicht hilft.

5.2.1 DisplaySearchListViewModel

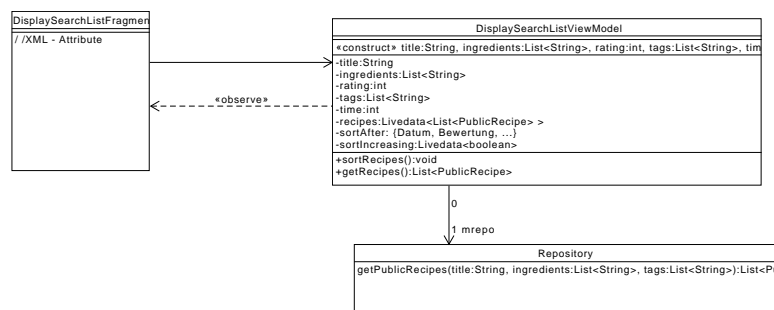


Abbildung 5.25: DisplaySearchList

- **DisplaySearchListViewModel(title:String, ingredients:List<String>, rating:int, tags:List<String>, time:int):constructor** Im Konstruktor werden alle Suchparameter übergeben **F53, F54, F56**
- **sortRecipes():void** Diese Methode sortiert die aufgelisteten Rezepte nach dem gewünschten Kriterium. Der Nutzer kann das Kriterium in einer Ausklappbaren Leisten im Fragment auswählen. Diese Kriterien beinhalten beispielsweise: Datum, Bewertung,... **F55, F58**
- **getRecipes() :List<PublicRecipe>** Diese Methode lädt die jeweiligen Rezepte, die

auf die Eingaben des "PublicRecipeSearchFragment" passen, aus dem Repository.

Observer Entwurfsmuster

Observer Durch den Observer ist es möglich eine saubere Kapselung der Schichten in nur eine Richtung zu halten. Die View Schicht kennt die ViewModel schicht, jedoch nicht andersherum. Um trotzdem aktuell zu bleiben, nutzt man das Observer Pattern. Die ViewModel Schicht hält die View Schicht nicht als Assoziation, sondern wird automatisch bei einer Änderung benachrichtigt, dass sich was geändert hat. Damit gehört es zu den Verhaltensmustern und dient der Weitergabe von Änderungen an einem Objekt an von diesem Objekt abhängige Strukturen.

5.2.2 CreateRecipeViewModel

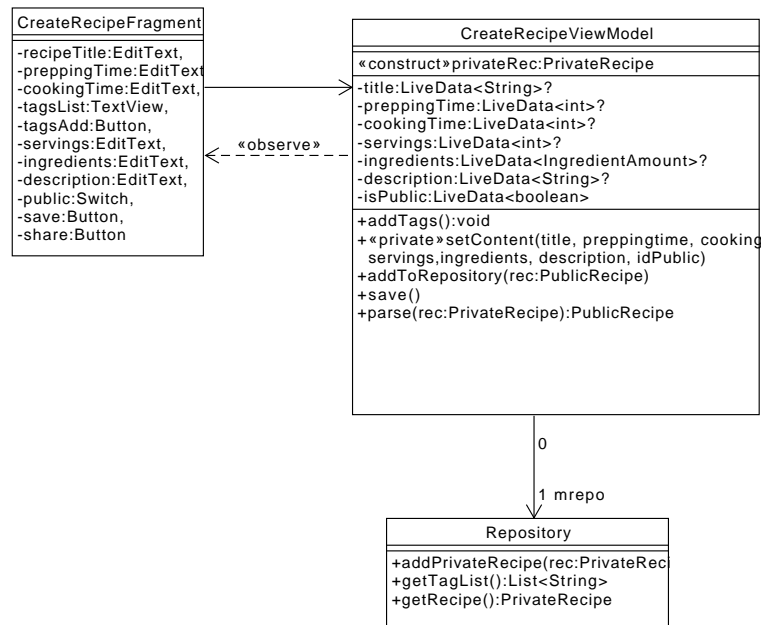


Abbildung 5.26: CreateRecipeViewModel

- **CreateRecipeViewModel(privateRec: PrivateRecipe): constructor** Wenn ein bestehendes Rezept bearbeitet werden soll, wird dieses im Konstruktor übergeben und sonst ein null Pointer **F15**
- **addTags(): void** öffnet das SearchWithTagsCreateRecipeFragment
- **setContent(title: String, preppingtime: Int, cookingtime: Int, servings, ingredients: List<String>, description: String, idPublic: String): void** schreibt die bisher eingetragenen LiveData-Attribute (!=null) in das Rezept **F2, F3, F4, F5, F6, F7, F8, F9, F10**

- `addToRepository(rec:PrivateRecipe):void` ruft erst `setContent(...)` auf und gibt dann das `privateRecipe` an das Repository zum Speichern in der SQLite DB
- `parse(rec:PrivateRecipe) :PublicRecipe` wandelt das `PrivateRecipe` `rec` in ein `PublicRecipe` um. Dazu prüft die Methode alle Attribute des `PrivateRecipe`. Außerdem werden die HTTP-Links aus der Zubereitungsbeschreibung geprüft. Ist ein Link nicht gültig, wird der Nutzer mithilfe eines Alert-Dialogs gewarnt **F32**
- `save():void` unterscheidet, ob `isPublic=true,false`. `true`: rufe `parse()` mit `privateRec` als Parameter auf und schicke dann das Resultat ins Repository zum Speichern auf dem Server `false`: speichert Rezept nur lokal **F1, F11, F14**
- `addToRepository(parse(privateRec)):void` `false`: rufe `addToRepository(privateRecipe)` auf und lasse somit das `privateRec` vom Repository in der SQLite DB speichern. **F13, F16**

5.2.3 RecipeListViewModel

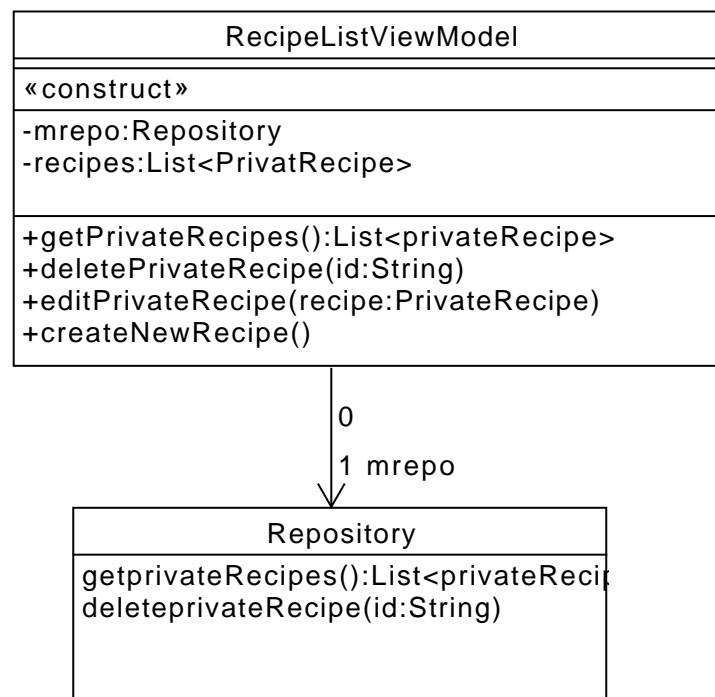


Abbildung 5.27: RecipeListViewModel

- `getPrivateRecipes():List<PrivateRecipe>` alle privaten Rezepte werden geladen.
- `deletePrivateRecipe(id:String):void` löscht ein privates Rezept. Falls dieses öffentlich ist, wird dieses auch vom Server entfernt **F17**

- `editPrivateRecipe(recipe:PrivateRecipe):void` Diese Methode löst einen Fragment wechsel aus, wobei das ausgewählte Rezept in das "CreateRecipeFragment" übergeben wird
- `createnewRecipe():void` Diese Methode löst einen Fragmentwechsel aus, wobei auf das "CreateRecipeFragment" mit unausgefülltem Template gewechselt wird

5.2.4 AddEditCommentViewModel

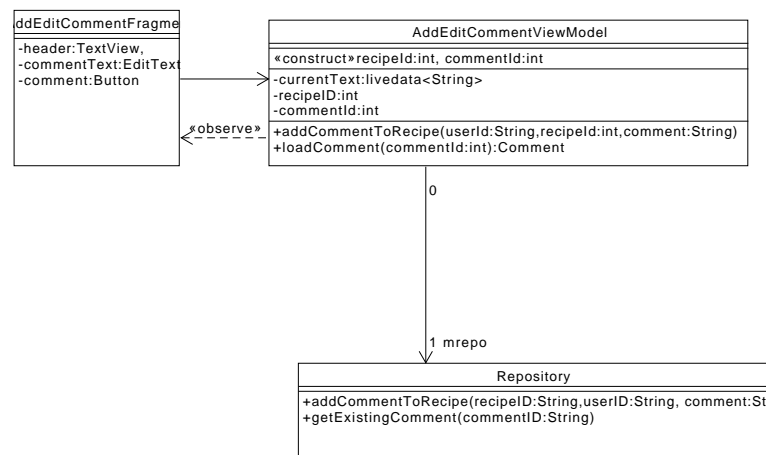


Abbildung 5.28: AddEditCommentViewModel

- `AddEditCommentViewModel(recipeId:int, commentId:int) :construct` Wenn ein Kommentar bearbeitet werden soll, wird die `commentId` hier übergeben
- `addCommentToRecipe(...) :void` Ordnet dem Rezept über das `Repository` ein neues Kommentar hinzu, falls die `commentID` noch nicht existiert. Falls die schon existiert, wird das bisherige ersetzt.
- `loadComment(id:int):Comment` fragt über das `Repository` den Server an, ob ein Kommentar mit der gegebenen ID existiert. Falls ja wird dieses zurückgegeben.

Der Nutzer ist hier, um ein Kommentar zu einem Rezept zu schreiben. Falls er ein schon bestehendes Kommentar ändern will, wird dieses schon in dem Textfeld angezeigt. Will der Nutzer ein neues Kommentar verfassen ist das Textfeld leer. Falls der Nutzer ein leeres Textfeld speichern will, kehrt der Nutzer auf das "RecipeDisplayFragment" zurück, ohne dass ein Kommentar hinzugefügt wird.

5.2.5 UserSearchViewModel

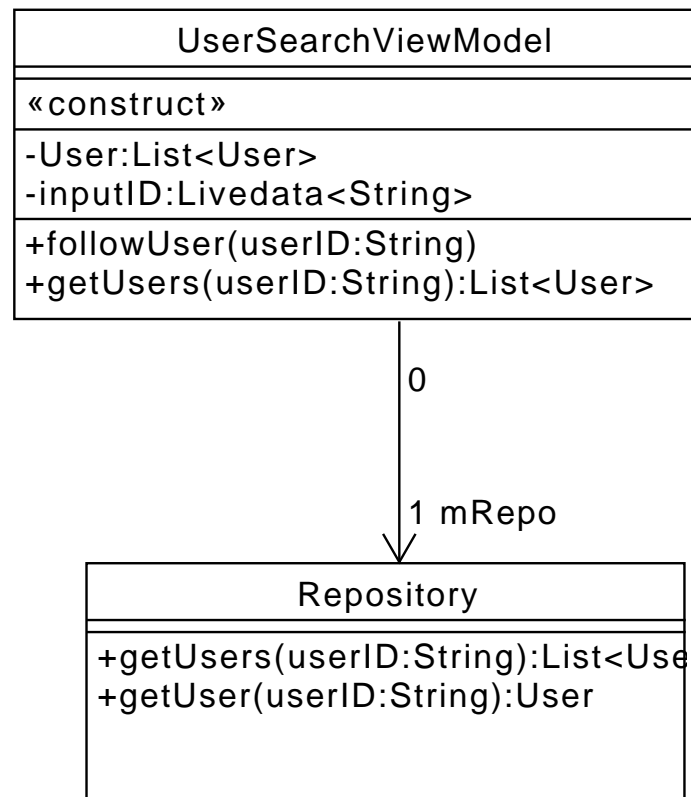


Abbildung 5.29: UserSearchViewModel

- `followUser(userID:String):void` fügt den übergebenen User in die Abonnementliste des Senders ein. **F48**
- `getUsers(userID:String):List<User>` gibt die User, deren ID mit userID beginnt, zurück. **F59, F60, F61**

Jeder Nutzer kann über die Sucheingabe die Benutzer-ID eines registrierten Nutzers eingeben. Klickt er dann auf „suchen“, so werden in einer Liste alle passenden Ergebnisse angezeigt. Die Benutzer-ID muss eindeutig sein. Trotzdem ergibt es Sinn, auch verschiedene Suchergebnisse anzuzeigen, da der Suchende eventuell nur ein Präfix der eigentlichen ID kennt, oder eingegeben hat. Demnach sollen alle Nutzer angezeigt werden, die auch diesen Präfix beinhalten. Im Pflichtenheft wurde beschrieben, dass die Benutzersuche durch mehrere Suchfilter eingeschränkt werden kann. Da die Architektur der Applikation in ihrer Grundfunktion jedoch nicht darauf ausgelegt ist, weil vorerst die ID der primäre eindeutige Bezeichner für einen angemeldeten Nutzer ist, wird diese Funktion nicht implementiert.

5.2.6 PublicRecipeSearchViewModel

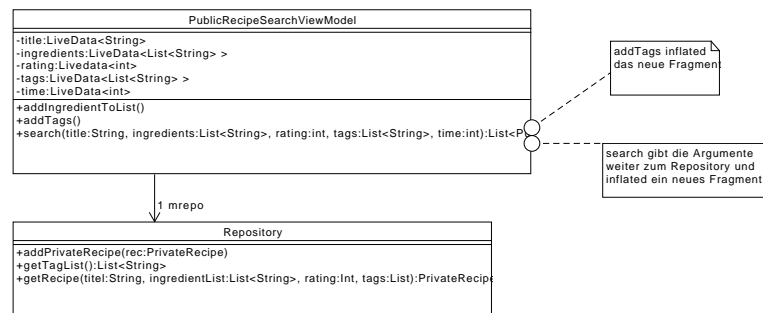


Abbildung 5.30: PublicRecipeSearchViewModel

- `addIngredientToList():void` Fügt der Suchliste Zutaten hinzu.
- `addTags():void` öffnet das SearchWithTagsFragment
- `search(title:String, ingredients:List<String>, rating:int, tags:List<String>, time:int):void` Diese Methode übergibt dem DisplaySearchListFragment die Suchparameter **F51**, **F52**

Die Spezifikation, dass Zubereitungsbeschreibung ein Suchfilter darstellt, wurde aus verschiedenen Gründen nicht in den Entwurf übernommen. Die Suchkernfunktionalität der Applikation liegt auf der Suche mit Titel, Zutaten und Tags. Die Beschreibung ist jedoch mehr für den Zubereitungsprozess eines Rezepts gedacht. Daher wurde er aus der Suche herausgenommen.

5.2.7 FeedViewModel

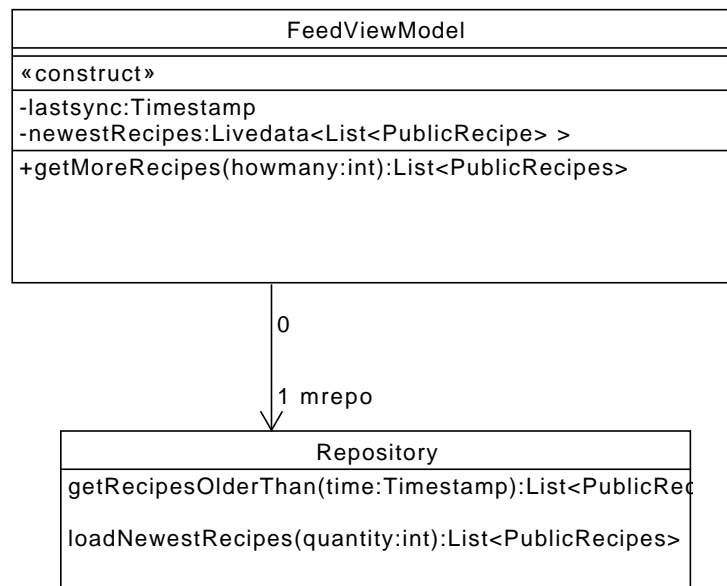


Abbildung 5.31: FeedViewModel

- **FeedViewModel()** :construct Beim instanziiieren werden die neusten Rezepte geladen
F72
- **getMoreRecipes(howmany:Int)** :List<PublicRecipe> lädt eine bestimmte Anzahl an öffentlichen Rezepten

5.2.8 ProfileDisplayViewModel

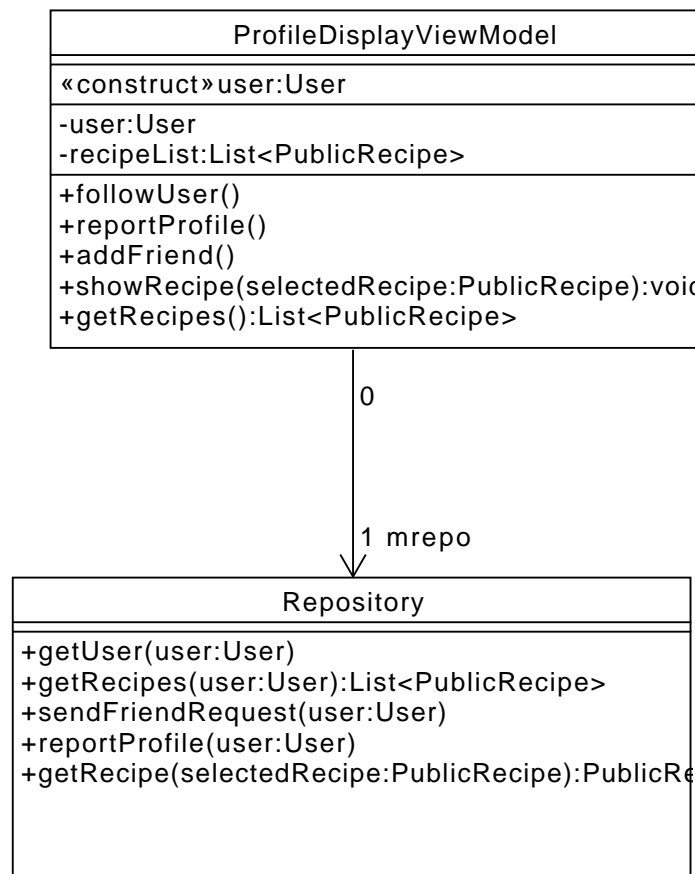


Abbildung 5.32: ProfileDisplayViewModel

- `ProfileDisplayViewModel(user:User) :construct` Beim Instanzieren wird der zu beobachtende Benutzer übergeben **F47**
- `followUser(user:User):void` schickt über das Repository eine Anfrage einem Nutzer zu folgen. **F48**
- `reportProfile(user:User):void` meldet einen Nutzer, der folglich vom Admin geprüft wird.
- `addFriend():void` Schickt dem Nutzer eine Freundschaftsanfrage **F63**
- `showRecipe(selectedRecipe:PublicRecipe):void` Zeigt das als Parameter übergebene Rezept in einem neuen Fragment an
- `getRecipes(selectedRecipe:PublicRecipe) List<PublicRecipe>` Bei der Auswahl eines Rezeptes wird dieses als Parameter an das RecipeDisplayFragment weitergegeben, das aufgerufen wird.

5.2.9 ProfileEditViewModel

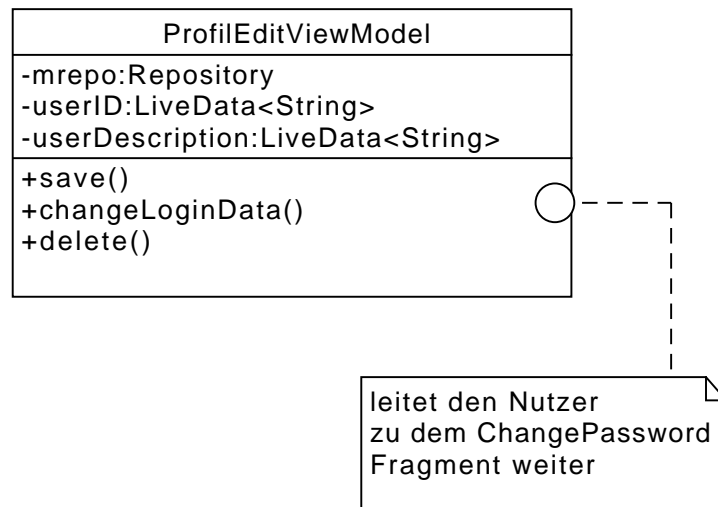


Abbildung 5.33: ProfileEditViewModel

- `ProfileEditViewModel(user:User) :construct` Im Konstruktor wird das Profil, das zu bearbeiten ist, übergeben
- `save():void` speichert das Profil über das Repository **F41, F43, F44, F45, F46**
- `changeLoginData():void` leitet den Nutzer auf das ChangePasswordFragment
- `delete() :void` löscht das Profil und alle davon veröffentlichten Rezepte **F42**

5.2.10 RegistrationViewModel

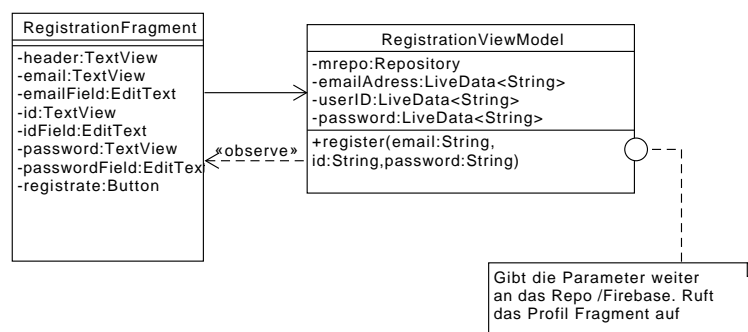


Abbildung 5.34: RegistrationViewModel

- `register(email:String, id:String, password:String) :void` registriert den Nutzer und speichert die Nutzerdaten. **F38, F39**

5.2.11 ChangePasswordViewModel

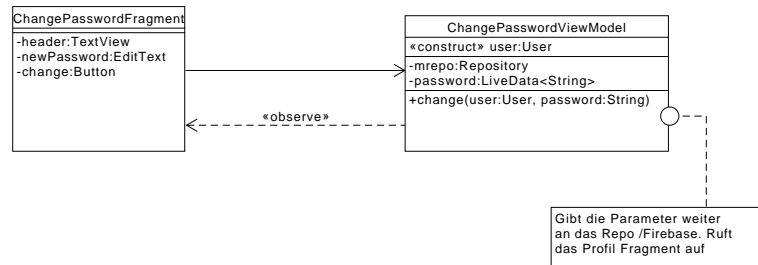


Abbildung 5.35: ChangePasswordViewModel

- `ChangePasswordViewModel(user:User) :construct` Im Konstruktor wird das Profil, dessen Passwort zu verändern ist, übergeben
- `change(user:User, password:String):void` ändert das aktuelle Passwort eines angemeldeten Nutzers zu dem neuen Passwort. **F40**

5.2.12 LoginViewModel

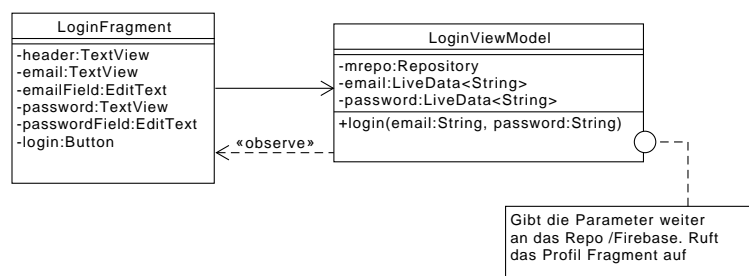


Abbildung 5.36: LoginViewModel

- `login(email:String, password:String) :void` überprüft die Eingabedaten und loggt den Nutzer ein. **F50**

5.2.13 EditTagViewModel

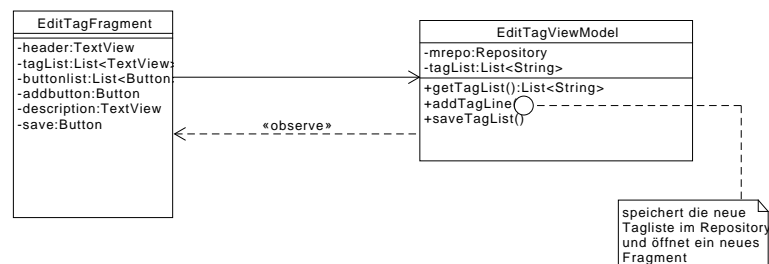


Abbildung 5.37: EditTagViewModel

- `getTagList()` `List<String>` lädt die Tagliste des Nutzers
- `addTagLine()` `:void` fügt der Ansicht eine neue Eingabezeile hinzu.
- `saveTagList()` `:void` speichert die erneuerte Tagliste über das Repository **F73**

5.2.14 SearchWithTagsViewModel

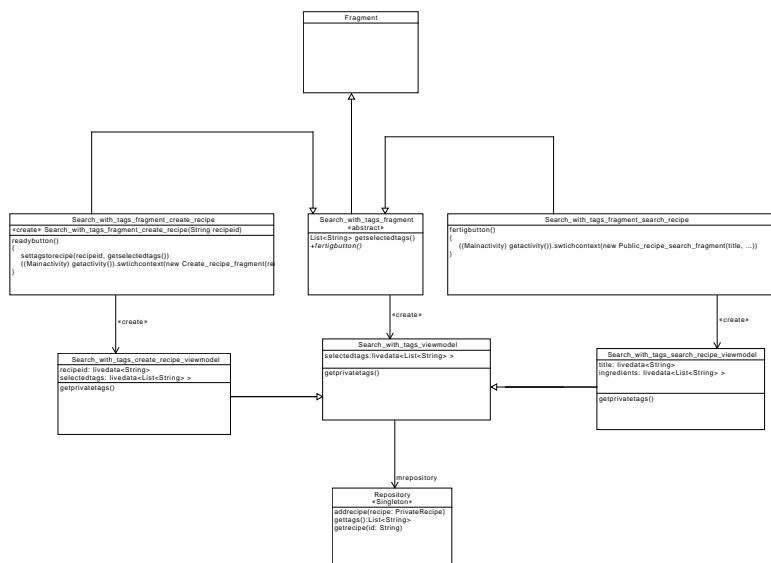


Abbildung 5.38: SearchWithTagsViewModel

Hier findet eine Unterscheidung statt, da es die Möglichkeiten gibt vom **searchWithTags-Fragment** zum **SearchRecipeFragment**, oder zum **CreateRecipeFragment** zu kommen. Damit diese Pfade voneinander getrennt sind, wurden hier zwei Kindklassen erstellt, die den gemeinsamen Teil als Elternklasse gekapselt haben.

- **searchWithTagsCreateRecipeViewModel**

`readyButton():void` lädt die ausgewählten Tags und wechselt zum `CreateRecipeFragment`

- `searchWithTagsSearchRecipeViewModel`

`getprivateTags() List<String>` gibt die Liste von Tags zurück

5.2.15 AdminViewModel

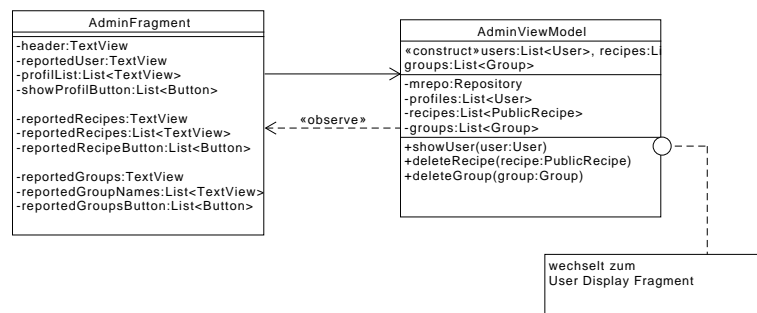


Abbildung 5.39: AdminViewModel

- `showUser(user:User):void` zeigt das Profil eines Nutzers, den der Admin bearbeiten kann. Bei der Bearbeitung eines Nutzerprofils muss der Admin über die Datenbank einträge ausserhalb der App arbeiten.
- `deleteRecipe(recipe:PublicRecipe) :void` löscht das Rezept aus der öffentlichen Ansicht heraus und wird beim Suchen daher nicht mehr angezeigt.
- `deleteGroup(group:Group) :void` entfernt die Gruppe aus der Datenbank, sodass sie nicht mehr für die Nutzer existiert.

5.2.16 ShoppingListDisplayViewModel

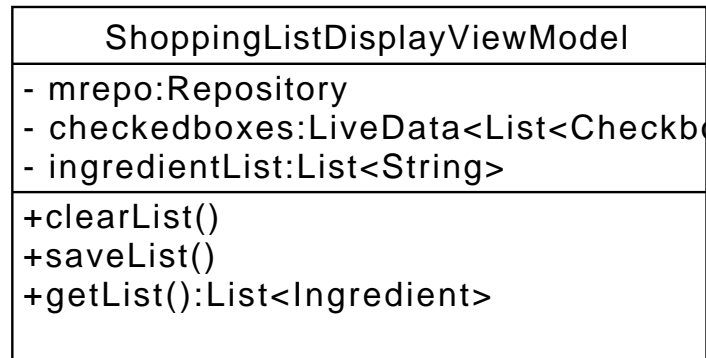


Abbildung 5.40: ShoppingListDisplayViewModel

- `clearList() :void` entfernt alle Einträge der Shoppinglist
- `saveList() :void` speichert die Liste mit allen abgehakten Zutaten **F21, F24**
- `getList() :List<Ingredient>` gibt die Liste an Zutaten der Liste zurück.

5.2.17 FriendListViewModel

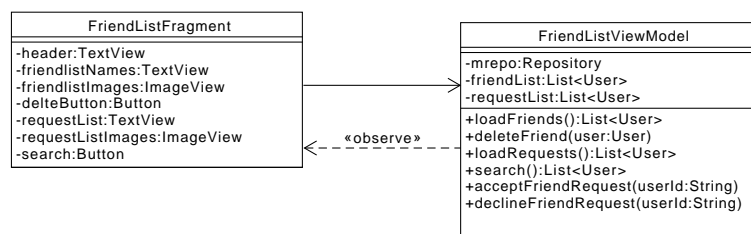


Abbildung 5.41: FriendListViewModel

- `loadFriends() :List<User>` lädt alle Freunde eines Nutzers und gibt diese als Liste zurück (F66)
- `deleteFriend(user:User) :void` löscht einen Freund als der Freundesliste des Nutzers **F65**
- `loadRequests() :List<User>` lädt alle offenen Freundschaftsanfragen des Nutzers
- `search() :void` leitet den Nutzer das UserSearchFragment weiter

- `acceptFriendRequest(userId:String) :void` Nimmt die Freundschaftsanfrage, ausgehen von dem Benutzer mit der `userId` an **F64**
- `declineFriendRequest(userId:String) :void` Nimmt die Freundschaftsanfrage, ausgehen von dem Benutzer mit der `userId` an **F64**

5.2.18 GroupMemberListViewModel

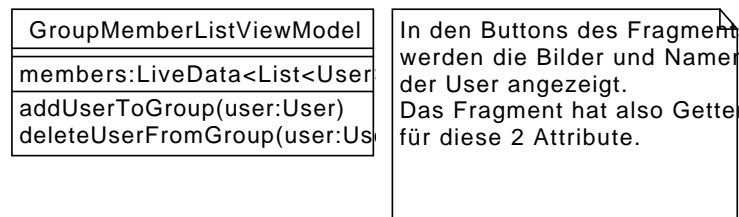


Abbildung 5.42: GroupMemberListViewModel

- `GroupMemberListViewModel(groupId:int) :constructor` Im Konstruktor wird übergeben, die Mitglieder welcher Gruppe übergeben werden sollen
- `addUserToGroup(user:User) :void` Fügt einen Benutzer zu der Gruppe hinzu **F67**
- `deleteUserFromGroup(user:User) :void` Entfernt einen Benutzer von einer Gruppe **F67**

5.2.19 CreateGroupViewModel

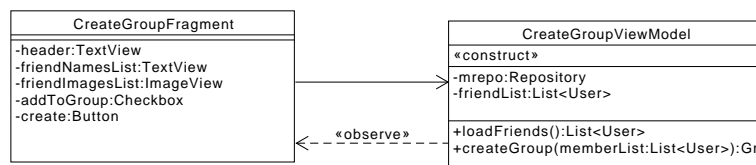


Abbildung 5.43: CreateGroupViewModel

- `loadFriends() :List<User>` lädt alle Freunde des Nutzers und gibt diese als Liste zurück
- `createGroup(memberList:List<User>) :Group` erstellt eine neue Gruppe mit allen ausgewählten Freunden. **F68**

5.2.20 FriendGroupViewModel

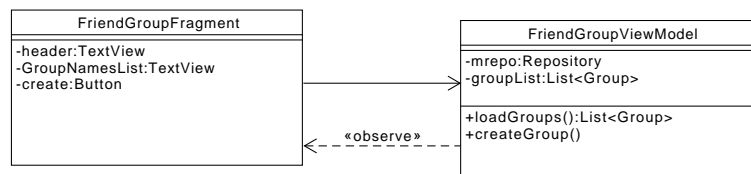


Abbildung 5.44: FriendGroupViewModel

- `loadGroups() :List<Group>` lädt alle Gruppen, in den der Nutzer Mitglied ist
- `createGroup() :void` leitet den Nutzer auf das createGroupFragment weiter

5.2.21 GroupViewModel

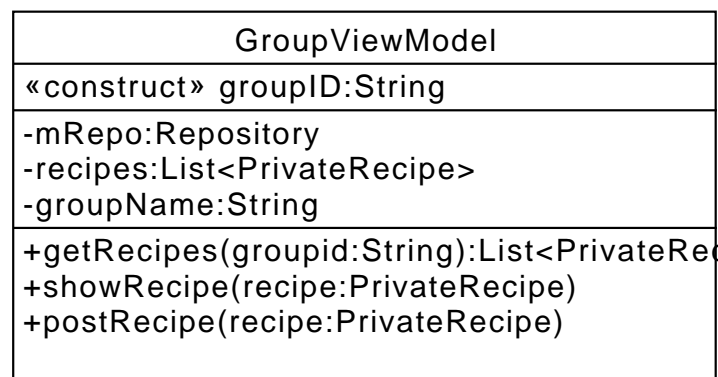


Abbildung 5.45: GroupViewModel

- `GroupViewModel(groupId:int) :constructor` Im Konstruktor wird übergeben, um welche Gruppe es sich handelt
- `getRecipes(groupId:String) :List<PrivateRecipe>` liefert alle privaten Rezept, die in dieser Gruppe gesendet wurden.
- `showRecipe(recipe:PrivateRecipe) :void` wechselt zu dem CreateRecipeFragment
- `postRecipe(recipe:PrivateRecipe) :void` schickt ein Rezept in die Gruppe **F69**

5.2.22 RecipeDisplayViewModel

RecipeDisplayViewModel
construct: recipe:PublicRecipe
-mRepo:Repository -recipe:PublicRecipe -portions:Livedata<int> -creator:User -rating:int
+favorite() +getUser(recipe:PublicRecipe):User +addToShoppinglist() +scale() +rate() +markAsEvil() +comment(comment:Comment)

Abbildung 5.46: RecipeDisplayViewModel

- `RecipeDisplayViewModel(recipe:PrivateRecipe)` :constructor Im Konstruktor wird übergeben, welches Rezept angezeigt werden soll **F33, F35**
- `favorite()` :void falls der Nutzer angemeldet ist, favorisiert dieser das angezeigte Rezept **F18**
- `getUser(recipe:PublicRecipe)` :User gibt den Ersteller eines öffentlichen Rezeptes zurück
- `addToShoppinglist()` :void fügt alle Zutaten des Rezeptes mit eingestellten Mengenangaben zur Shoppinglist hinzu **F22, F23**
- `scale()` :void skaliert das Rezept mit der eingegebenen Portionenzahl **F20**
- `rate()` :void gibt die vom Benutzer eingegebene Bewertung ab **F34**
- `markAsEvil()` :void markiert das Rezept in der Onlinedatenbank als gemeldet
- `comment(comment:Comment)` :void Es wird in das AddEditCommentFragment gewechselt, wenn comment Null Pointer, dann wird ein Neues Kommentar erstellt, wenn nicht, dann wird dieses Kommentar bearbeitet **F36, F37**

5.2.23 FavouriteViewModel

FavouriteViewModel
-mRepo:Repository -recipes:List<PublicRecipe>
+showRecipe(recipe:PublicRecipe) +removeFromFav(recipe:PublicRecipe)

Abbildung 5.47: FavouriteViewModel

- `showRecipe(recipe:PublicRecipe) :void` wechselt in das RecipeDisplayFragment mit dem eingegebenen Rezept
- `removeFromFav(recipe:PublicRecipe) :void` löscht das Rezept von der Favoritenliste

6 Beschreibung Domain Layer Interfaces

In diesem Kapitel werden die Interfaces des Domain Layers beschrieben. Diese Interfaces liegen quasi auf dem „Use Case“ Ring der Clean Architecture und kapseln die

6.1 Repository

Repository Entwurfsmuster

Es dient als Schnittstelle zwischen der Domänenschicht und der Datenzugriffsschicht. Es ist insbesondere in den Situationen hilfreich, in denen es viele unterschiedliche Domänenklassen oder viele unterschiedliche Zugriffe auf die Datenzugriffsschicht gibt.

Die Repository-Interfaces enthalten die Methodenköpfe der Repository-Methoden, jedoch keine Implementierung dieser. Zu jedem Interface des Repositorys gibt es eine Klasse, die dieses Implementiert. Da das Repository ein Einzelstück ist (es soll ja die „Single Source of Truth“ zwischen der inneren Schicht und den Datenquellen sein), gibt es nur ein Package, an Klassen, die die Methoden der Interfaces implementieren. Die Abstraktion ist in diesem Falle im Grunde nur zur Kapselung nötig.

Das Repository-Interface enthält die Methodenköpfe der Repository-Methoden.

6.2 Authentifizierung

Authentication «Interface»
+registerate(userId:String, email:String, password:String, activity:Activity) +login(email:String, password:String, activity:Activity) +logout() +pwEdit(pw:String,activity:Activity) +usernameEdit(username:String,activity:Activity) +userDelete(activity:Activity) -saveToken() +getToken(activity:Activity):String

Abbildung 6.1: Authentification Schnittstelle

Das Authentifizierungs-Interface definiert Methoden, die zur Authentifizierung eines Nutzers nötig sind. Dieses Interface wird dann von Klassen, die eine Authentifizierungsschnittstelle definieren, implementiert. Im Fall von Exzellenzkoch ist diese Klasse dann eine, die die Methoden von Firebase in den vordefinierten Interface-Methoden aufruft.

6.3 ImportExport

Im ImportExport-Interface stehen die nötigen Methodenvorgaben, die die Intents brauchen:

```
importExport.shareRecipe(Recipe) : void  
importExport.parseIntent(Intent) : privateRecipe
```

7 Klassenbeschreibung Domain Layer

Dieses Kapitel beschreibt die Klassen des Domain Layers, welche durch die vorigen Interfaces schon umrissen wurden. Das Diagramm 13.1 gibt eine Übersicht die Domain Entities der Anwendung und ist am Ende des Dokuments in größer angehängt.

7.1 Domain Entities

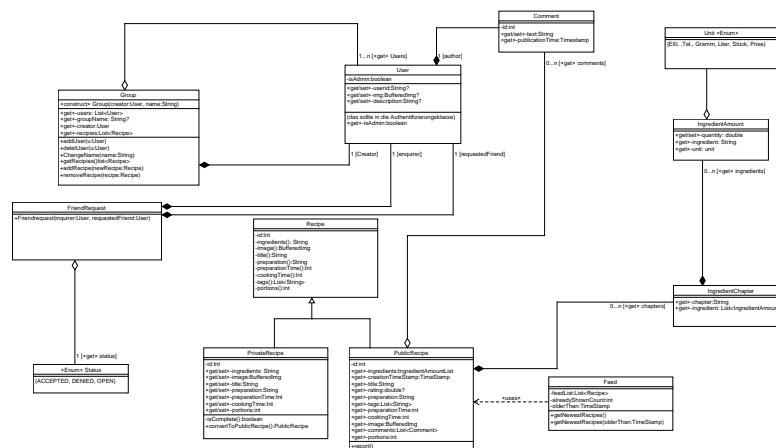


Abbildung 7.1: Übersicht über die Domain Entities

7.1.1 User

Die Klasse User beschreibt das Profil eines angemeldeten Nutzers.

Attribute

- **id:int**
Eindeutige Identifikation des Profils
- **img:BufferedImage**
Profilbild eines Profils
- **description:String**
textuelle Beschreibung des Profils
- **isAdmin:boolean**
beschreibt, ob ein Profil Adminrechte hat oder nicht
- **email:String**
E-Mail Adresse eines Profils, mit dem sich der Nutzer registriert

Admin

Die Klasse Admin erbt von der Klasse User und ist ein Singleton. Er hat die Aufgabe sich um gemeldete Rezepte, Gruppen und Nutzerprofile zu kümmern und diese zu verwalten. Das Singleton, welches hier verwendet wird, ist eine leichte Abweichung des Standards. Die Instanziierung des Admins muss zu Beginn einmal manuell gemacht werden. Der Singleton ist ein Erzeugungsmuster, wobei hier kein globaler Zugriff auf die Instanziierung gegeben wird. Das Entwurfsmuster Einzelstück garantiert, dass es nur einen Admin geben kann. Der Admin ist zudem noch dafür zuständig, falls jemand ein Problem mit der Applikation, oder ähnliches hat sich darum zu kümmern. Damit ist für die Nutzer die Möglichkeit geschaffen über den Admin als sichere Zwischeninstanz Probleme der Applikation zu handhaben.

7.1.2 Group

Diese Klasse beschreibt eine Gruppe von Freunden, in der private Rezepte geteilt werden können.

Attribute

- friendGroup:List<User>
Alle Nutzer die sich in der Gruppe befinden
- friendGroupName:String
Name der Gruppe
- creator:User
Ersteller der Gruppe
- recipes:List<Recipe>
Rezepte die in der Gruppe geteilt wurden

7.1.3 «Interface» Recipe

Dieses Interface beschreibt die Mindestfunktionalität, die ein Rezept haben muss, um angezeigt werden zu können.

Methoden

- getingredients():String
gibt textuelle Beschreibung der Zutaten zurück
- getimage():BufferedImage
gibt Bild des Rezeptes zurück
- gettitle():String
gibt Titel des Rezeptes zurück
- getpreparation():String
gibt textuelle Beschreibung der Zubereitung zurück
- getpreparationtime():int
gibt benötigte Zeit in Minuten der Zubereitung zurück

- `getcookingTime():int`
gibt benötigte Koch- bzw. Schmorzeit der Zubereitung zurück
- `gettags():List<String>`
gibt die dem Rezept gegebenen Tags zurück
- `getportions():int`
gibt die Anzahl Portionen, für die die Zutaten ausgelegt sind, zurück

7.1.4 privateRecipe

Diese Klasse beschreibt ein privates Rezept.

Attribute

- `id:int`
Falls ein privates Rezept schon veröffentlicht wurde, ist dies die id des zugehörigen öffentlichen Rezeptes
- `ingredients:String`
Textuelle Beschreibung der Zutaten
- `title:String`
Titel des privaten Rezeptes
- `image:BufferedImage`
Bild des Rezeptes
- `preparation:String`
Textuelle Beschreibung der Zubereitung
- `preparationTime:int`
Benötigte Zeit in Minuten der Zubereitung
- `cookingTime:int`
Benötigte Koch- bzw. Schmorzeit der Zubereitung
- `tags:List<String>`
Liste der dem Rezept gegebenen Tags
- `portions:int`
Anzahl Portionen, für die die Zutaten ausgelegt sind

7.1.5 publicRecipe

Diese Klasse beschreibt ein öffentliches Rezept.

Attribute

- `id:int`
Eindeutige Identifikation des öffentlichen Rezeptes
- `ingredients:IngredientAmountList`
Zutaten des Rezeptes
- `creationTimeStamp:TimeStamp`
Datum und Uhrzeit der Veröffentlichung

- title:String
Titel des Rezeptes
- rating:double
bewertung des Rezeptes
- preparation:String
Textuelle Beschreibung der Zubereitung
- tags:List<String>
Liste der dem Rezept gegebenen Tags
- preparationTime:int
Benötigte Zeit in Minuten der Zubereitung
- cookingTime:int
Benötigte Koch- bzw. Schmorzeit der
- image:BufferedImage
Bild des Rezeptes Zubereitung
- comments:List<Comment>
Liste der dem Rezept gegebenen Kommentare
- portions:int
Anzahl Portionen, für die die Zutaten ausgelegt sind

7.1.6 IngredientAmountList

Diese Klasse beschreibt eine objektorientierte Zerlegung der Zutatenliste eines öffentlichen Rezeptes.

Attribute

- chapters:List<IngredientChapter>
Liste der Unterkapitel einer Zutatenliste eines Rezeptes
- portions:int
Anzahl an Portionen, für die das Rezept ausgelegt ist

7.1.7 IngredientChapter

Diese Klasse beschreibt ein Kapitel einer Zutatenliste eines Rezeptes.

Attribute

- chapter:String
Name des Kapitels
- ingredients:List<IngredientAmount>
Liste der Zutaten eines Kapitels

7.1.8 IngredientAmount

Diese Klasse beschreibt eine Zutat in einem Kapitel einer Zutatenliste.

Attribute

- ingredient:String
textuelle Beschreibung der Zutat
- unit:Unit
Einheit, in der die Zutat gemessen wird
- quantity:double
Anzahl an unit in der die Zutat benötigt wird

7.1.9 Unit

Diese Klasse beschreibt die Einheiten, in der Zutaten gemessen werden können.

Attribute

- Eßl
Einheit Esslöffel
- Tel
Einheit Teelöffel
- Liter
Einheit Liter
- g
Einheit Gramm
- kg
Einheit Kilogramm

7.1.10 Feed

Diese Klasse beschreibt den Feed an neuen Rezepten.

Attribute

- feedList:List<PublicRecipe>
Liste der neusten Rezepte
- alreadyShownCount:int
Anzahl der Rezepte, die von der App schon angezeigt wurden
- olderThan:TimeStamp
Datum und Uhrzeit, zu dem der Feed geladen wurde

7.1.11 Comment

Diese Klasse beschreibt einen Kommentar, der einem öffentlichen Rezept gegeben wurde.

Attribute

- `id:int`
Eindeutige Identifikation eines Kommentars
- `text:String`
Text des Kommentars
- `publicationTime:TimeStamp`
Datum und Uhrzeit, an dem ein Kommentar veröffentlicht wurde
- `author:User`
Verfasser des Kommentars

7.2 Authentication

Diese Klasse ist für die Authentifikation von einem Nutzer zuständig.

Attribute

- `auth:FirebaseAuth`
Zugangspunkt zur Firebaseauthentifikation SDK

Methoden

- `registrate(userId:String, email:String, password:String):void`
Ein neuer Nutzer wird auf Firebase registriert. Es wird die Nutzerid, Email und Passwort übergeben.
- `login(email:String, password:String):void`
Logt einen Nutzer auf Firebase an. Es wird die Email und das Passwort übergeben.
- `logout():void`
Meldet einen Nutzer in Firebase ab.
- `pwEdit(pw:String):void`
Ändert das Passwort eines Nutzers auf Firebase ab. Das neue Passwort wird übergeben.
- `userIdEdit(userId:String):void`
Ändert den Nutzernamen eines Nutzers ab. Es wird der neue Nutzername übergeben.
- `getToken():String`
Gibt den JWT-Token vom gerade angemeldeten Nutzer zurück.

7.3 ImportExport

```
importExport.shareRecipe(Recipe) : void  
importExport.parseIntent(Intent) : privateRecipe
```


Kriterium **W7** der zu entwickelten App definiert die Funktionalität, Rezepte teilen und importieren zu können.

Das Android Framework sieht dafür Intents und Intentfilter vor. Das oben beschriebene Interface kapselt dafür Funktionalität. Die Klasse, die das Interface implementieren, wird nun beschrieben.

7.3.1 Export

Wird `importExport.shareRecipe()` aufgerufen, wird aus dem als Parameter übergebenen Rezept ein `Textintent` generiert und an das Framework übergeben.

Beispielcode:

```
// Create the text message with a string
val sendIntent = Intent().apply {
    action = Intent.ACTION_SEND
    putExtra(Intent.EXTRA_TEXT, textMessage)
    type = "text/plain"
}
```

Das Android Framework kümmert sich dann darum, ein Dialogfenster mit kompatiblen Apps anzuzeigen, in denen das Rezept geteilt werden kann.

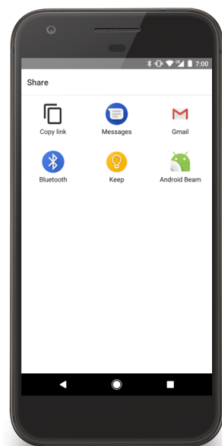


Abbildung 7.2: Dialogfenster zum Auswählen der App, die den Intent erhält [5]

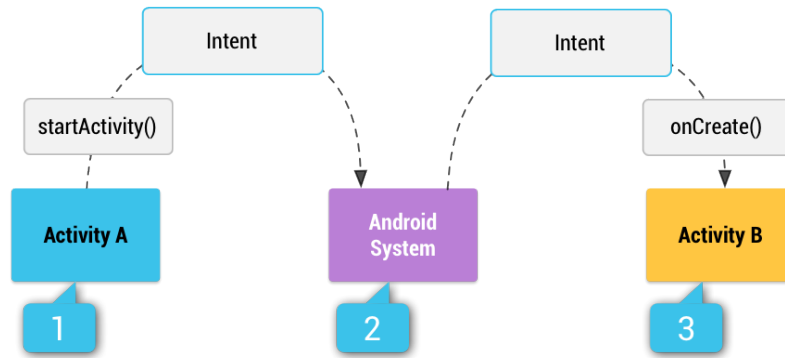


Abbildung 7.3: Schaubild, Übergabe von Intents zwischen Androidapps [5]

7.3.2 Import

Zum Importieren registriert die App Intentfilter mit einem `<intent-filter>` element in der Manifestdatei. Jeder Intentfilter spezifiziert den Typ einer Kategorie von Intents, die die App akzeptiert. So kann die App sich für HTML URLs und Text als Empfänger von Intents registrieren. Daraufhin wird vom System, die Activity aufgerufen, die dafür im Manifest registriert ist. Die Activity und das zugehörige ViewModel, was dieses Intent dann erhält, kann dann die Methode `parseIntent()` aufrufen. Diese dient dann dazu, aus dem Intent ein privates Rezept zu parsen.

8 Klassenbeschreibung Data Layer

In diesem Kapitel werden die Klassen des Data Layers beschrieben, insbesondere die Repository-Klassen.

8.1 Repository

Das Repository bietet Zugriff zu den Datenquellen. Im Falle von Exzellenzkoch sind da die lokale SQLite Datenbank, zu welcher der Zugriff durch die Room Persistence Library gestellt wird, und der Webserver. In diesem Kapitel soll es um das Modell der lokalen SQLite Datenbank gehen, der Webserver wird im Kapitel 9 behandelt.

8.1.1 Modell für SQLite-Datenbank

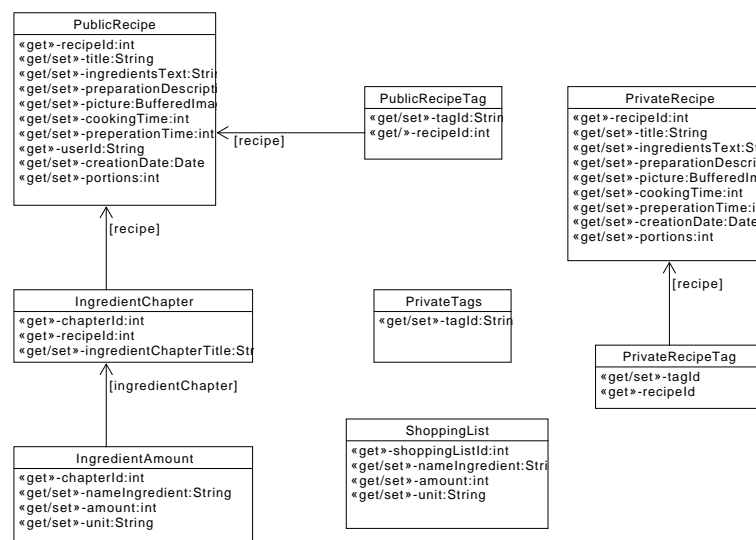


Abbildung 8.1: Klassendiagramm für das Modell für die SQLite Datenbank in der App

Das Diagramm zeigt Klassen die jeweils eine Tabelle von der SQLite Datenbank in der App abbilden. Diese Klassen werden in Verbindung mit DAOs, welche im folgenden Unterkapitel beschrieben werden, verwendet, um Daten von der SQLite Datenbank abzurufen. Im folgenden werden diese Klassen genauer beschrieben:

Für die öffentlichen Rezepte werden die Klassen `PublicRecipe`, `PublicRecipeTag`, `IngredientChapter` und `IngredientAmount` verwendet. Für die private Rezepte werden die Klasse `PrivateRecipe` und `PrivateRecipeTag` verwendet. Die Klasse `ShoppingList` ist für die Einkaufsliste und die `PrivateTag` für die private Tags, welche man erstellen kann.

PublicRecipe

Diese Klasse beschreibt ein öffentliches Rezept, welches von der Datenbank geladen oder gespeichert wird.

Attribute

- «get»-recipeId:int
Eindeutige Identifikation eines öffentlichen Rezeptes
- «get/set»-title:String
Title eines Rezeptes
- «get/set»-ingredientsText:String
Alle Zutaten als zusammenhängender Text
- «get/set»-preparationDescription
Zubereitungsbeschreibung eines Rezeptes
- «get/set»-picture:BufferedImage
Bild des Gerichts
- «get/set»-cookingTime:int
Dauer für das Kochen
- «get/set»-preperationTime:int
Dauer für die Zubereitung
- «get»-userId:String
Id des Nutzers, dem das Rezept gehört
- «get/set»-creationDate:Date
Das Datum an dem das Rezept erstellt wurde
- «get/set»-portions:int
Anzahl für wie viel Personen das Rezept gedacht ist

PublicRecipeTag

Diese Klasse repräsentiert ein Tag für ein Rezept.

Attribute

- «get/set»-tagId:String
Name des Tags
- «get/»-recipeId:int
Id des Rezeptes zu dem der Tag zugewiesen ist

IngredientChapter

Diese Klasse repräsentiert ein Kapitel von den Zutaten für ein Rezept.

Attribute

- «get»-chapterId:int
Die Id des Kapitels
- «get»-recipeId:int
Die Id eines Rezeptes zu dem ein Kapitel zugewiesen ist
- «get/set»-ingredientChapterTitle:String
Der Titel des Kapitels

IngredientAmount

Diese Klasse repräsentiert eine Zutat von einem Kapitel für ein Rezept.

Attribute

- «get»-chapterId:int
Id des Kapitels zu dem die Zutat zugeordnet ist
- «get/set»-nameIngredient:String
Name der Zutat
- «get/set»-amount:int
Menge der Zutat
- «get/set»-unit:String
Einheit der Zutat

PrivateTags

Diese Klasse repräsentiert ein privates Tag, welches erstellt werden kann.

Attribute

- «get/set»-tagId:String
Name des privaten Tags

ShoppingList

Diese Klasse repräsentiert einen Eintrag in der Einkaufsliste.

Attribute

- «get»-shoppingListId:int
Eindeutiger Id für ein Einkaufslisteneintrag
- «get/set»-nameIngredient:String
Name der Zutat
- «get/set»-amount:int
Menge der Zutat
- «get/set»-unit:String
Einheit der Zutat

PrivateRecipe

Diese Klasse repräsentiert ein privates Rezept.

Attribute

- «get»-recipeId:int
Eindeutige Identifikation eines öffentlichen Rezeptes
- «get/set»-title:String
Title eines Rezeptes
- «get/set»-ingredientsText:String
Alle Zutaten als zusammenhängender Text
- «get/set»-preparationDescription
Zubereitungsbeschreibung eines Rezeptes
- «get/set»-picture:BufferedImage
Bild des Gerichts
- «get/set»-cookingTime:int
Dauer für das Kochen

- «get/set»-preparationTime:int
Dauer für die Zubereitung
- «get/set»-creationDate:Date
Das Datum an dem das Rezept erstellt wurde
- «get/set»-portions:int
Anzahl für wie viel Personen das Rezept gedacht ist

PrivateRecipeTag

Diese Klasse repräsentiert ein Tag für ein Rezept.

Attribute

- «get/set»-tagId:String
Name des Tags
- «get/set»-recipeId:int
Id des Rezeptes zu dem der Tag zugewiesen ist

8.1.2 Daoklassen für die Datenbank

DAO Entwurfsmuster

Data Access Object (DAO) ist ein Entwurfsmuster, das den Zugriff auf unterschiedliche Arten von Datenquellen so kapselt, dass die angesprochene Datenquelle ausgetauscht werden kann, ohne dass der aufrufende Code geändert werden muss. Dadurch soll die eigentliche Programmlogik von technischen Details der Datenspeicherung befreit werden und flexibler einsetzbar sein. DAO ist also ein Muster für die Gestaltung von Programmierschnittstellen (APIs). Diese Klassen bieten die Schnittstellen für den Zugriff auf die SQLite Datenbank.

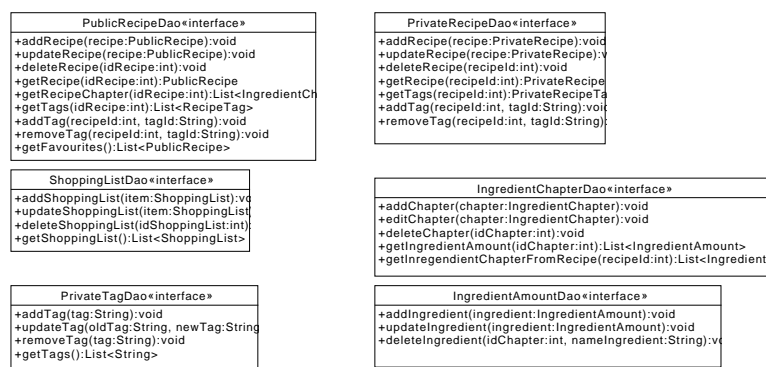


Abbildung 8.2: Klassendiagramm für die Daos für die SQLite Datenbank in der App

PublicRecipeDao«interface»

Diese Klasse gibt Schnittstellen für das öffentliche Rezept in der Datenbank an.

Methoden

- +addRecipe(recipe:PublicRecipe):void
Fügt das übergebene neue Rezept hinzu
- +updateRecipe(recipe:PublicRecipe):void
Aktualisiert einen Eintrag mit dem übergebenen Rezept
- +deleteRecipe(idRecipe:int):void
Löscht ein Rezept mit der entsprechenden Id
- +getRecipe(idRecipe:int):PublicRecipe
Gibt das Rezept mit der entsprechenden Id zurück
- +getRecipeChapter(idRecipe:int):List<IngredientChapter>
Gibt die Kapitel des Rezeptes zurück
- +getTags(idRecipe:int):List<RecipeTag>
Gibt die Tags eines Rezeptes zurück
- +addTag(recipeId:int, tagId:String):void
Fügt ein Tag zu einem Rezept hinzu
- +removeTag(recipeId:int, tagId:String):void
Entfernt ein Tag von einem Rezept
- +getFavourites():List<PublicRecipe>
Gibt alle Favouriten vom Nutzer zurück

IngredientChapterDao«interface»

Diese Klasse gibt Schnittstellen für das Zutatenkapitel eines Rezeptes in der Datenbank an.

Methoden

- +addChapter(chapter:IngredientChapter):void
Fügt das übergebene Kapitel hinzu
- +editChapter(chapter:IngredientChapter):void
Aktualisiert eine Kapitel mit dem übergebenen Kapitel

- +deleteChapter(idChapter:int):void
Löscht ein Kapitel mit der entsprechenden Id
- +getIngredientAmount(idChapter:int):List<IngredientAmount>
Gibt die Liste von Zutaten von einem Kapitel zurück
- +getInredientChapterFromRecipe(recipeId:int):List<IngredientChapter>
Gibt die Liste an Kapitel von einem Rezept zurück

IngredientAmountDao«interface»

Diese Klasse gibt Schnittstellen für Zutaten eines Kapitels eines Rezeptes in der Datenbank an.

Methoden

- +addIngredient(ingredient:IngredientAmount):void
Fügt die übergeben Zutat hinzu
- +updateIngredient(ingredient:IngredientAmount):void
Aktualisiert eine Zutat mit der übergeben Zutat
- +deleteIngredient(idChapter:int, nameIngredient:String):void
Löscht eine Zutat mit der entsprechenden Kapitelid und Zutatennamen

PrivateRecipeDao«interface»

Diese Klasse gibt Schnittstellen für das private Rezept in der Datenbank an.

Methoden

- +addRecipe(recipe:PublicRecipe):void
Fügt das übergebene neue Rezept hinzu.
- +updateRecipe(recipe:PublicRecipe):void
Aktualisiert einen Eintrag mit dem übergebenen Rezept
- +deleteRecipe(idRecipe:int):void
Löscht ein Rezept mit der entsprechenden Id
- +getRecipe(idRecipe:int):PublicRecipe
Gibt das Rezept mit der entsprechenden Id zurück
- +getTags(idRecipe:int):List<RecipeTag>
Gibt die Tags eines Rezeptes zurück

- +addTag(recipeId:int, tagId:String):void
Fügt ein Tag zu einem Rezept hinzu
- +removeTag(recipeId:int, tagId:String):void
Entfernt ein Tag von einem Rezept

ShoppingListDao«interface»

Diese Klasse gibt Schnittstellen für die Einkaufsliste in der Datenbank an.

Methoden

- +addShoppingList(item:ShoppingList):void
Fügt das übergebene neue ShoppingListelement hinzu.
- +updateShoppingList(item:ShoppingList):void
Aktualisiert ein ShoppingListelement mit dem übergebenen Element
- +deleteShoppingList(idShoppingList:int):void
Löscht einen Eintrag in der ShoppingList mit der entsprechenden Id
- +getShoppingList():List<ShoppingList>
Gibt die Einkaufsliste zurück

PrivateTagDao«interface»

Diese Klasse gibt Schnittstellen für die Einkaufsliste in der Datenbank an.

Methoden

- +addTag(tag:String):void
Fügt einen neuen privaten Tag hinzu
- +updateTag(oldTag:String, newTag:String):void
Aktualisiert einen private Tag
- +removeTag(tag:String):void
Löscht einen privaten Tag
- +getTags():List<String>
Gibt alle privaten Tags zurück

8.2 Datenbank

8.2.1 ClientDatenbank

Shoppinglist

Die Relation ShoppingList beinhaltet Informationen über die Einkaufsliste.

- shoppingListId
Id der eine ShoppingList-Eintrages
- nameIngrient
Name der Zutat
- unit
Einheit, in der die Zutat gemessen wird
- amount
Anzahl an unit in der die Zutat benötigt wird

PublicRecipe

Die Relation PublicRecipe beinhaltet Informationen über favorisierte Rezepte. Die Relation PublicRecipe wird über einen Fremdschlüssel in der Relation PublicRecipeTag und IngredientChapter adressiert und verweist selbst auf die Relation User.

- recipeId
Eindeutige Kennung des Rezeptes in der Serverdatenbank (Primärschlüssel)
- title
Titel des Rezeptes
- ingredientsText
Textuelle Beschreibung der Zutaten
- preparationDescription
Textuelle Beschreibung der Zubereitung
- picture
Bild des Rezeptes
- cookingTime
Dauer der Kochzeit in Minuten
- preparationTime
Dauer der Vorbereitung in Minuten
- userId

Eindeutige Kennung des Ersteller des Rezeptes in der Serverdatenbank (Fremdschlüssel)

- `creationDate`
Datum und Uhrzeit der Veröffentlichung des Rezeptes
- `portions`
Anzahl Portionen für die das Rezept ausgelegt ist

IngredientChapter

Die Relation `IngredientChapter` beinhaltet Informationen über die Unterkapitel in der Zutatenliste. Die Relation `IngredientChapter` wird über einen Fremdschlüssel in der Relation `IngredientAmount` adressiert und verweist selbst auf die Relation `PublicRecipe`.

- `chapterId`
Eindeutige Kennung des Unterkapitels (Primärschlüssel)
- `recipeId`
Zeiger auf das Rezept (Fremdschlüssel)
- `ingredientChapterTitle`
Title des Chapters

IngredientAmount

Die Relation `IngredientAmount` beinhaltet Informationen über die Zutaten in einem Unterkapitel einer Zutatenliste eines Rezeptes. Die Relation `IngredientAmount` verweist selbst auf die Relation `IngredientChapter`.

- `chapterId`
Zeiger auf das Unterkapitel (Fremdschlüssel)
- `nameIngredient`
Name der Zutat
- `unit`
Einheit, in der die Zutat gemessen wird
- `amount`
Anzahl an unit in der die Zutat benötigt wird

PublicRecipeTag

Die Relation `PublicRecipeTag` beinhaltet Informationen über die Tags eines veröffentlichten Rezeptes. Die Relation `PublicRecipeTag` verweist auf die Relation `PublicRecipe`.

- `tagId`
Name des Tags

- recipeId
Zeiger auf das Rezept (Fremdschlüssel)

PrivateTags

Die Relation PrivateTags listet alle privaten Tags auf.

- tagId
Name des Tags (Primärschlüssel)

PrivateRecipe

Die Relation PrivateRecipe listet alle von dem Nutzer erstellten Rezepte auf. Die Relation PrivateRecipe wird über einen Fremdschlüssel in der Relation PrivateRecipeTag adressiert.

- recipeId
Eindeutige Kennung des Rezeptes in dieser Tabelle (Primärschlüssel)
- title
Titel des Rezeptes
- ingredientsText
Textuelle Beschreibung der Zutaten
- preparationDescription
Textuelle Beschreibung der Zubereitung
- picture
Bild des Rezeptes
- cookingTime
Dauer der Kochzeit in Minuten
- preparationTime
Dauer der Vorbereitung in Minuten
- creationDate
Datum und Uhrzeit der Veröffentlichung des Rezeptes (null falls nicht veröffentlicht)
- portions
Anzahl Portionen für die das Rezept ausgelegt ist

PrivateRecipeTag

Die Relation PrivateRecipeTag weist jedem Rezept eine Menge an Tags zu. Die Relation PrivateRecipeTag verweist selbst auf die Relation PrivateRecipe.

- tagId
Name des Tags

- recipeId
Zeiger auf das Rezept

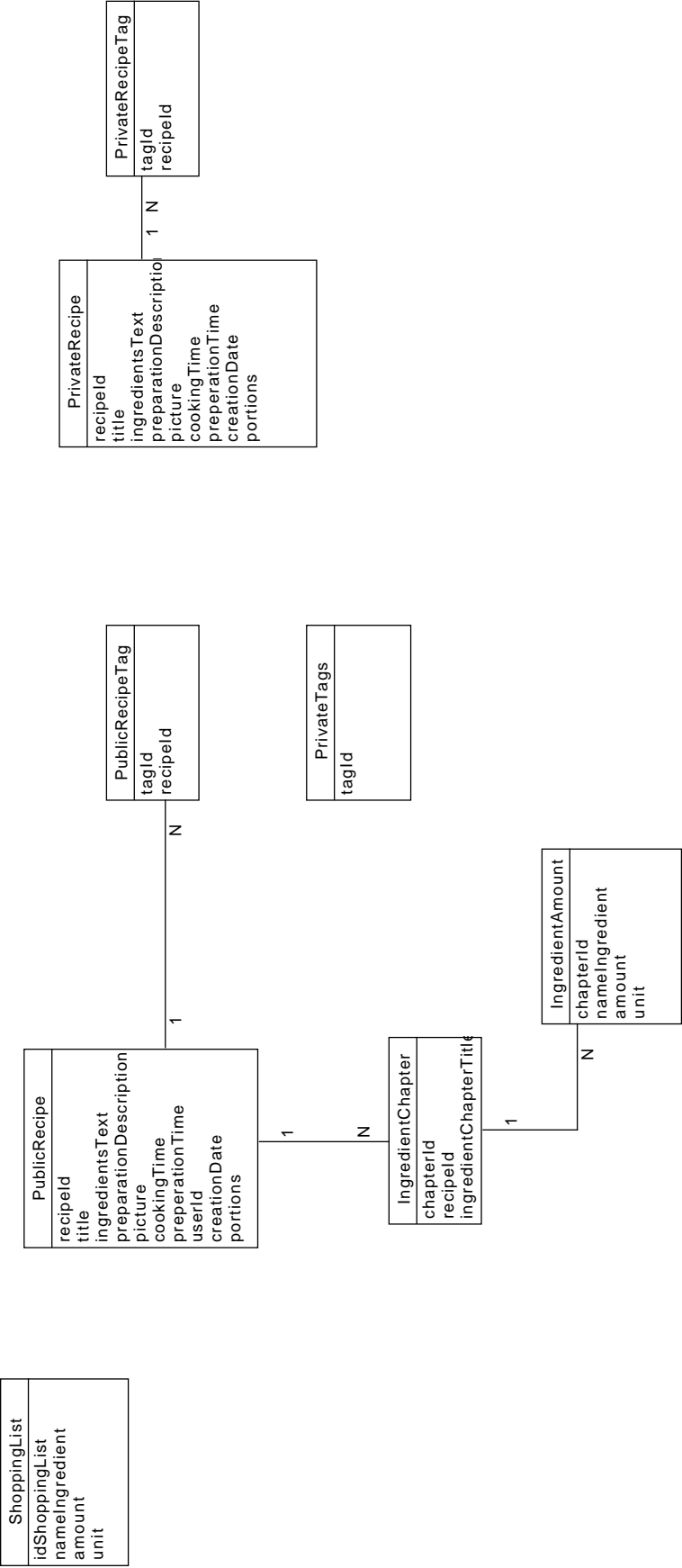


Abbildung 8.3: ER-Diagramm der Clientdatenbank

9 Klassenbeschreibung REST-Schnittstelle

Im Folgenden ist die REST-Schnittstelle beschrieben.

Um die Verknüpfung zu dem API-Paket des Servers zu beschreiben ist die jeweilige Methode aus der entsprechenden API-Klasse des Servers, die beim entsprechenden REST-Endpunkt aufgerufen wird, mit angegeben.

Die API Datentypen der Schnittstelle werden durch die *Data Transfer Objects* definiert. Diese Objekte bzw. Listen dieser Objekte werden, in JSON serialisiert, im Body gesendet und empfangen.

9.0.1 Paging

Get-Methoden, die eine Liste von Werten zurückliefern unterstützen Paging. Damit können Stück für Stück bzw. Page für Page die nächsten Objekte angefordert werden.

?page=1?size=10

Würde die ersten 10 Werte liefern.

page=2?size=100

die Objekte 101-200. (Beziehungsweise die Objekte 101-150, wenn es nur 150 gibt). Die erste Page hat die Nummer 1.

9.0.2 Optionale Parameter

Alle Parameter hinter dem Fragezeichen sind Optionale Parameter. Keiner dieser Parameter muss angegeben werden. Es können einer oder mehrere angegeben werden, um eine Suche weiter einzuschränken, bzw. eine Reihenfolge (order) festzulegen

Beispielsweise kann durch den optionalen Parameter 'Sort,, bei manchen Suchen die Reihenfolge spezifiziert werden.

9.1 API/REST-Schnittstelle

9.1.1 UserApi

HTTP Endpoint: **POST** /users

`createUser (user : UserDto) : void`

Diese Methode erstellt einen neuen Benutzer in der Serverdatenbank.

HTTP Status Codes: Wenn ein UserDto mit einer in dem UserDto Objekt gleichen userId existiert, wird **400 (Bad Request)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **PUT** /users/{userId}

`updateUser (user : UserDto) : void`

Diese Methode aktualisiert einen bereits bestehenden Benutzer in der Serverdatenbank.

HTTP Status Codes:

HTTP Endpoint: **DELETE** /users/{userId}

`deleteUser (userId : String) : void`

Diese Methode löscht einen Benutzer aus der Serverdatenbank.

HTTP Status Codes: Wenn die userId nicht mit dem JSON Web Token übereinstimmt und er nicht Admin ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **GET** /users/{userId}

`getUserById (userId : String) : UserDto`

Diese Methode gibt einen den UserDto mit userId zurück.

HTTP Status Codes: Bei Erfolg wird **200 (Ok)** zurückgegeben. Wenn die userId zu keinem Profil gehört wird **400 (Bad Request)** zurückgegeben.

HTTP Endpoint:

GET /users/?UserIdPrefix={userId}?page={page}?size={size} die Ergebnisse werden alphabetisch nach Userid sortiert zurückgegeben.

`searchUser (userId : String , page : int , size : int) : List < UserDto >`

Diese Methode gibt Nutzer zurück, deren `userId` einen Präfix haben, der zu dem Parameter der Methode passt.

HTTP Status Codes: Bei Erfolg wird **200 (Ok)** zurückgegeben. Wenn es keiner Profile gibt, die zu der `userId` passen, wird **400 (Bad Request)** zurückgegeben.

HTTP Endpoint: **PUT** `/users{userId}/followings{followingId}`

`addFollow(subscriberId:String, followedId:String):void`

Diese Methode speichert in der Serverdatenbank, dass der Benutzer mit der `subscriberId` dem Benutzer mit der `followerId` folgt.

HTTP Status Codes: Wenn ein `UserDto` mit einer in dem `UserDto` Objekt gleichen `userId` existiert, wird **400 (Bad Request)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

`\get{/users{userId}/followings?page={page}?size={size}}`
`getFollow(userId:String, page:int, size:int):List<UserDto>`

Diese Methode gibt alle Benutzer zurück, denen der Benutzer mit der `userId` folgt.

HTTP Status Codes: Wenn ein `UserDto` mit einer in dem `UserDto` Objekt gleichen `userId` existiert, wird **400 (Bad Request)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

`\delete{/users{userId}/followings/{followeId}}`
`removeFollow(subscriberId:String, followedId:String):void`

Diese Methode speichert in der Serverdatenbank, dass der Benutzer mit der `subscriberId` dem Benutzer mit der `followerId` nicht mehr folgt.

HTTP Status Codes: Wenn ein `UserDto` mit einer in dem `UserDto` Objekt gleichen `userId` existiert, wird **400 (Bad Request)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

9.1.2 PublicRecipeApi

HTTP Endpoint: **POST** `/recipies`

`addRecipe(recipe:PublicRecipeDto):void`

Diese Methode fügt ein neues veröffentlichtes Rezept der Serverdatenbank hinzu.

HTTP Status Codes: Wenn der Benutzer nicht eingeloggt ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **PUT** /recipies/{recipeId}

`updateRecipe(recipe: PublicRecipeDto): void`

Diese Methode aktualisiert ein bereits bestehendes Rezept in der Serverdatenbank.

HTTP Status Codes: Wenn der Benutzer nicht der Autor des zu updatenden Rezeptes und nicht Admin ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **DELETE** /recpies/{recipeId}

`deleteRecipe(recipeId: int): void`

Diese Methode löscht das Rezept mit der als Parameter übergebenen recipeId aus der Serverdatenbank.

HTTP Status Codes: Wenn der Benutzer nicht der Autor des zu löschenden Rezeptes und nicht Admin ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **GET** recipes/{recipeId}

`getRecipe(recipeId: int): PublicRecipeDto`

Diese Methode gibt das Rezept mit der als Parameter übergebenen recipeId zurück.

HTTP Status Codes: Wenn die idRecipe zu keinem Rezept gehört, wird **400 (Bad Request)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **PUT** /recipes/user/{userId}/rating

`setRating(recipeId: int, userId: String, value: int): void`

Diese Methode gibt dem Rezept mit der recipeId eine Bewertung von value des Benutzers mit userId.

HTTP Status Codes: Wenn die userId nicht mit dem JSON Web Token übereinstimmt, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **GET** /recipes/recipeId/rating

```
getAverageRating(recipeId:int):double
```

Diese Methode gibt die durchschnittliche Bewertung eines Rezeptes mit der recipeId zurück.

HTTP Status Codes: Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint:

GET /recipies?title={title}?tags={taglist}
?ingredients={ingredientlist}?date={date}?minrating=minrating
?sort={sortorder}?page={page}?size={size}

```
search(title:String, tags:List<String>, ingredients:List<String>,  
creationDate: Date, alreadyLoad:int, readCount:int, sortOrder:String, page:int
```

Diese Methode gibt alle Rezepte zurück, deren Titel an irgendeiner Stelle den String title haben, deren Tags eine Obermenge von tags, Zutaten eine Obermenge von ingredients, deren Veröffentlichungsdatum neuer als creationDate sind.

sortOrder dieser optionale parameter kann einen von zwei Werten haben

- „title“ - die Ergebnisse sind alphabetisch nach Titel sortiert.
- „date“ - es werden die neuesten Rezepte nach Rezeptalter absteigend zurückgegeben.
- „ranking“ - es werden die Rezepte nach Durchschnittsrating die höchsten zuerst zurückgegeben.

HTTP Status Codes: Wenn es keine Rezepte mit den gegebenen Kriterien gibt, wird **400 (Bad Request)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **GET** /reported/comments?page={page}?size={size}

```
reportRecipe(recipeId:int):void
```

Diese Methode setzt das markAsEvil Flag des Rezeptes mit recipeId.

HTTP Status Codes: Wenn es kein Rezept mit der gegebenen recipeId gibt, wird **400 (Bad Request)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

9.1.3 FriendRequestApi

HTTP Endpoint: **PUT** /friendrequests

```
publishRequest(request: FriendRelationDto)
```

Diese Methode sendet eine Freundschaftsanfrage in die Serverdatenbank.

HTTP Status Codes: Wenn diese Freundschaftsanfrage schon abgelehnt wurde oder die `userId` des Anfragenden nicht mit dem JSON Web Token übereinstimmt, wird **401 (Unauthorized)** zurückgegeben. Wenn die Anfrage schon angenommen wurde, wird **399???** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint:

GET /requests/open/requestedFriend/{userId}?page={page}?size={size}

```
getOpenRequestsforMe(userId:String, page:int, size:int) : List<String>
```

Diese Methode gibt alle offenen Freundschaftsanfragen, die an den Benutzer mit `userId` gerichtet sind, zurück.

HTTP Status Codes: Wenn die `userId` nicht existiert, wird **400 (Bad Request)** zurückgegeben. Wenn die `userId` nicht mit dem JSON Web Token übereinstimmt, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **GET** /friends/user/userId?page={page}?size={size}

```
getFriends(userId:String, page:int, size:int) : List<String>
```

Diese Methode gibt alle Benutzer zurück, deren Freundschaftsbeziehung zu dem Benutzer mit `userId`, akzeptiert wurde.

HTTP Status Codes: Wenn die `userId` nicht existiert, wird **400 (Bad Request)** zurückgegeben. Wenn die `userId` nicht mit dem JSON Web Token übereinstimmt, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **DELETE** /friendrequests/enquirer/{userId}/requestedFriend/{userId}

```
deleteFriend(enquirerId:String, requestedFriendId:String):void
```

Diese Methode löscht die Freundschaftsbeziehung zwischen den Benutzern mit `enquirerId` und `requestedFriendId`.

HTTP Status Codes: Wenn die `enquirerId` nicht mit dem JSON Web Token übereinstimmt, wird **401 (Unauthorized)** zurückgegeben. Wenn die `requestedFriendId` nicht existiert oder die Freundschaftsbeziehung gar nicht existiert, wird **400 (Bad Request)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

9.1.4 GroupApi

HTTP Endpoint: **POST** /groups

`createGroup(group : GroupDto) : void`

Diese Methode speichert eine neue Gruppe in der Serverdatenbank.

HTTP Status Codes: Wenn die `mainUserId` der `group` nicht mit dem JSON Web Token übereinstimmt, wird **401 (Unauthorized)** zurückgegeben. Wenn eine Freundschaftsbeziehung zwischen dem `mainUser` der Gruppe und den Mitgliedern nicht existiert, wird **400 (Bad Request)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **PUT** /groups/{groupId}

`updateGroup(group : GroupDto) : void`

Diese Methode aktualisiert eine Gruppe in der Serverdatenbank.

HTTP Status Codes: Wenn die `mainUserId` der `group` nicht mit dem JSON Web Token übereinstimmt, wird **401 (Unauthorized)** zurückgegeben. Wenn eine Freundschaftsbeziehung zwischen dem `mainUser` der Gruppe und den Mitgliedern nicht existiert, wird **400 (Bad Request)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **DELETE** /groups/{groupId}/user/{userId}

`deleteGroup(groupId : int) : void`

Diese Methode löscht eine Gruppe aus der Serverdatenbank.

HTTP Status Codes: Wenn die `mainUserId` der `group` nicht mit dem JSON Web Token übereinstimmt, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **GET** /groups/{groupId}

`getGroupId(groupId : int) : GroupDto`

Diese Methode gibt die Gruppe mit der `groupId` aus der Serverdatenbank zurück.

HTTP Status Codes: Wenn es keine Gruppe mit der `groupId` gibt, wird **400 (Bad Request)** zurückgegeben. Wenn die `userId` im JSON Web Token nicht in der Gruppe mit der `groupId` ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **PUT** /groups/{groupId}/user/{userId}

`addUserToGroup(userId:String , groupId:int):void`

Diese Methode fügt einen Benutzer mit der `userId` zur Gruppe mit `groupId` in der Serverdatenbank hinzu.

HTTP Status Codes: Wenn es keine Gruppe mit `idGroup` existiert oder wenn `idUser` nicht existiert, wird **400 (Bad Request)** zurückgegeben. Wenn die `mainUserId` der group mit `idGroup` nicht mit dem JSON Web Token übereinstimmt, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: `GET /groups?user={userId}?page={page}?size={size}`

`getGroups(userId:String , page:int , size:int):List<GroupDto>`

Diese Methode gibt alle Gruppen zurück, in denen Der Benutzer mit `userId` Mitglied ist zurück.

HTTP Status Codes: Wenn die `userId` nicht mit dem JSON Web Token übereinstimmt, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: `DELETE /groups/user/{userId}`

`removeUserFromGroup(userId:String , groupId:int):void`

Diese Methode löscht den Benutzer mit `idUser` aus der Gruppe mit `idGroup` aus der Serverdatenbank.

HTTP Status Codes: Wenn es keine Gruppe mit `idGroup` existiert oder wenn `idUser` nicht existiert, wird **400 (Bad Request)** zurückgegeben. Wenn die `userId` nicht mit dem JSON Web Token übereinstimmt, wird **401 (Unauthorized)** zurückgegeben. Wenn der Benutzer mit `idUser` sich nicht in der Gruppe mit `idGroup` befindet, wird **399** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: `GET /groups/{groupId}/recipies?page={page}?size={size}`

`getSharedRecipesFromGroup(groupId:int , page:int , size:int):List<SharedPrivateR`

Diese Methode gibt alle geteilten Rezepte in der Gruppe mit `groupId` zurück.

HTTP Status Codes: Wenn es keine Gruppe mit `idGroup` existiert, wird **400 (Bad Request)** zurückgegeben. Wenn die `userId` im JSON Web Token nicht in der Gruppe mit `idGroup` enthalten ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: `POST /groups/{groupId}/recipies`

`addSharedRecipeToGroup(groupId:int ,
sharedRecipe:SharedPrivateRecipeDto):void`

Diese Methode fügt der Gruppe mit groupId das Rezept sharedRecipe hinzu.

HTTP Status Codes: Wenn es keine Gruppe mit idGroup existiert, wird **400 (Bad Request)** zurückgegeben. Wenn die userId im JSON Web Token nicht in der Gruppe mit idGroup enthalten ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **DELETE** /groups/{groupId}/recipes/{recipeId}

`deleteSharedRecipeFromGroup(groupId:int , sharedRecipeId:int):void`

Diese Methode löscht ein in einer Gruppe mit groupId geteiltes Rezept mit idSharedRecipe.

HTTP Status Codes: Wenn es keine Gruppe mit idGroup existiert oder wenn es kein Rezept in der Gruppe mit idSharedRecipe existiert, wird **400 (Bad Request)** zurückgegeben. Wenn die userId im JSON Web Token nicht in der Gruppe mit idGroup enthalten ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **PUT** /groups/{groupId}/recipes/{recipeId}

`updateRecipe(groupId:int , recipe:SharedPrivateRecipeDto):void`

Diese Methode aktualisiert ein in einer Gruppe mit idGroup geteiltes Rezept.

HTTP Status Codes: Wenn es keine Gruppe mit idGroup existiert oder wenn es kein Rezept in der Gruppe mit idSharedRecipe existiert, wird **400 (Bad Request)** zurückgegeben. Wenn die userId im JSON Web Token nicht in der Gruppe mit idGroup enthalten ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **PUT** /groups/{groupId}/reported

`reportGroup(groupId:int):void`

Diese Methode meldet eine Gruppe.

HTTP Status Codes: Wenn die userID im JSON Web Token nicht in der Gruppe mit idGroup enthalten ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **PUT** /groups/{groupId}/recipes{recipeId}/reported

`reportSharedPrivateRecipe(recipeId:int):void`

Diese Methode meldet ein in einer Gruppe geteiltes Rezept mit idRecipe.

HTTP Status Codes: Wenn die userID im JSON Web Token nicht in der Gruppe mit idGroup enthalten ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

9.1.5 FavouritesApi

HTTP Endpoint: **PUT** /users/{userId}/favorites/{recipeId}

```
addFavRecipe(userId:String, recipeId:int):void
```

Diese Methode fügt das veröffentlichtes Rezept mit recipeId dem Benutzer mit userId als Favorit hinzu.

HTTP Status Codes: Wenn es kein Profil mit userID oder Rezept mit idRecipe existiert, wird **400 (Bad Request)** zurückgegeben. Wenn die userID nicht mit dem JSON Web Token übereinstimmt, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **GET** /users/{userId}/favorites?page={page}?size={size}

```
getFavRecipe(userId:String, page:int, size:int):List<PublicRecipeDto>
```

Diese Methode gibt alle favorisierten Rezept des Benutzers mit userId zurück.

HTTP Status Codes: Wenn es kein Profil mit der userID existiert, wird **400 (Bad Request)** zurückgegeben. Wenn die userID nicht mit dem JSON Web Token übereinstimmt, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **DELETE** /users/{userId}/favorites/{recipeId}

```
delFavRecipe(userId:String, recipeId):void
```

Diese Methode entfernt die Favorisierung des Benutzers mit userId auf das Rezept mit recipeId.

HTTP Status Codes: Wenn es kein Profil mit der userID existiert, wird **400 (Bad Request)** zurückgegeben. Wenn die userID nicht mit dem JSON Web Token übereinstimmt, wird **401 (Unauthorized)** zurückgegeben. Wenn das Profil mit userId das Rezept mit recipeId gar nicht favorisiert hat, wird **400 (Bad Request)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

9.1.6 AdminApi

HTTP Endpoint: **GET** /reported/recipes?page={page}?size={size}

```
getReportedPublicRecipe(userId:String, page:int, size:int):List<PublicRecipeDto>
```

Diese Methode gibt alle öffentlichen Rezepte zurück, die gemeldet wurden.

HTTP Status Codes: Wenn im JSON Web Token nicht isAdmin als Claim gesetzt ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **GET** /reported/sharedPrivateRecipies?page={page}?size={size}

```
getReportedSharedPrivateRecipe(userId:String , page:int , size:int ):List<SharedP
```

Diese Methode gibt alle in Gruppen versendeten privaten Rezepte zurück, die gemeldet wurden.

HTTP Status Codes: Wenn im JSON Web Token nicht isAdmin als Claim gesetzt ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **GET** /reported/users?page={page}?size={size}

```
getReportedUsers(userId:String , page:int , size:int ):List<UserDto>
```

Diese Methode gibt alle gemeldeten Benutzer zurück.

HTTP Status Codes: Wenn im JSON Web Token nicht isAdmin als Claim gesetzt ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **GET** /reported/groups?page={page}?size={size}

```
getReportedGroups(userId:String , page:int , size:int ):List<GroupDto>
```

Diese Methode gibt alle gemeldeten Gruppen zurück.

HTTP Status Codes: Wenn im JSON Web Token nicht isAdmin als Claim gesetzt ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **GET** /reported/comments?page={page}?size={size}

```
getReportedComments(userId:String , page:int , size:int ):List<CommentDto>
```

Diese Methode gibt alle gemeldeten Kommentare zurück.

HTTP Status Codes: Wenn im JSON Web Token nicht isAdmin als Claim gesetzt ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **DELETE** /recipies/{recipeId}/reported

```
deReportPublicRecipe(recipeId:String ):void
```

Diese Methode macht die Meldung eines öffentlichen Rezeptes rückgängig.

HTTP Status Codes: Wenn im JSON Web Token nicht isAdmin als Claim gesetzt ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **DELETE** /sharedprivaterecipes/{recipeId}/reported

`deReportSharedPrivateRecipe(recipeId : int) : void`

Diese Methode macht die Meldung eines geteilten privaten Rezeptes in einer Gruppe rückgängig.

HTTP Status Codes: Wenn im JSON Web Token nicht `isAdmin` als Claim gesetzt ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **DELETE** /user/{userId}/reported

`deReportUser(userId : int) : void`

Diese Methode macht die Meldung eines Benutzers rückgängig.

HTTP Status Codes: Wenn im JSON Web Token nicht `isAdmin` als Claim gesetzt ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **DELETE** /group/{groupId}/reported

`deReportGroup(groupId : int) : void`

Diese Methode macht die Meldung einer Gruppe rückgängig.

HTTP Status Codes: Wenn im JSON Web Token nicht `isAdmin` als Claim gesetzt ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **DELETE** /comment/{commentId}/reported

`deReportComment(commentId : int) : void`

Diese Methode macht die Meldung eines Kommentars rückgängig.

HTTP Status Codes: Wenn im JSON Web Token nicht `isAdmin` als Claim gesetzt ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

9.1.7 CommentApi

HTTP Endpoint: **POST** /comments

`addComment(comment : CommentDto) : void`

Diese Methode fügt einen Kommentar der Serverdatenbank hinzu.

HTTP Status Codes: Wenn die `userId` des Autors des `comment` nicht mit dem JSON

Web Token übereinstimmt, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **PUT** /comments/{commentId}

`updateComment(comment: CommentDto): void`

Diese Methode aktualisiert einen Kommentar in der Serverdatenbank.

HTTP Status Codes: Wenn die userId des Autors des comment nicht mit dem JSON Web Token übereinstimmt, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **DELETE** comment/{commentId}

`deleteComment(commentId: int): void`

Diese Methode löscht den Kommentar mit idComment aus der Serverdatenbank.

HTTP Status Codes: Wenn die userId des Autors des comment nicht mit dem JSON Web Token übereinstimmt und im JSON Web Token nicht isAdmin als Claim gesetzt ist, wird **401 (Unauthorized)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **GET** /recipes/{recipeId}/comments?page={page}?size={size}

`getComments(recipeId: int, page: int, size: int): List<CommentDto>`

Diese Methode gibt alle Kommentare des Rezeptes mit idRecipe zurück.

HTTP Status Codes: Wenn es kein Rezept mit idRecipe existiert, wird **400 (Bad Request)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

HTTP Endpoint: **PUT** /comment/commentId/reported

`reportComment(commentId: int): void`

Diese Methode meldet den Kommentar mit commentId.

HTTP Status Codes: Wenn es kein Rezept mit idRecipe existiert, wird **400 (Bad Request)** zurückgegeben. Bei Erfolg wird **200 (Ok)** zurückgegeben.

9.2 DTOs

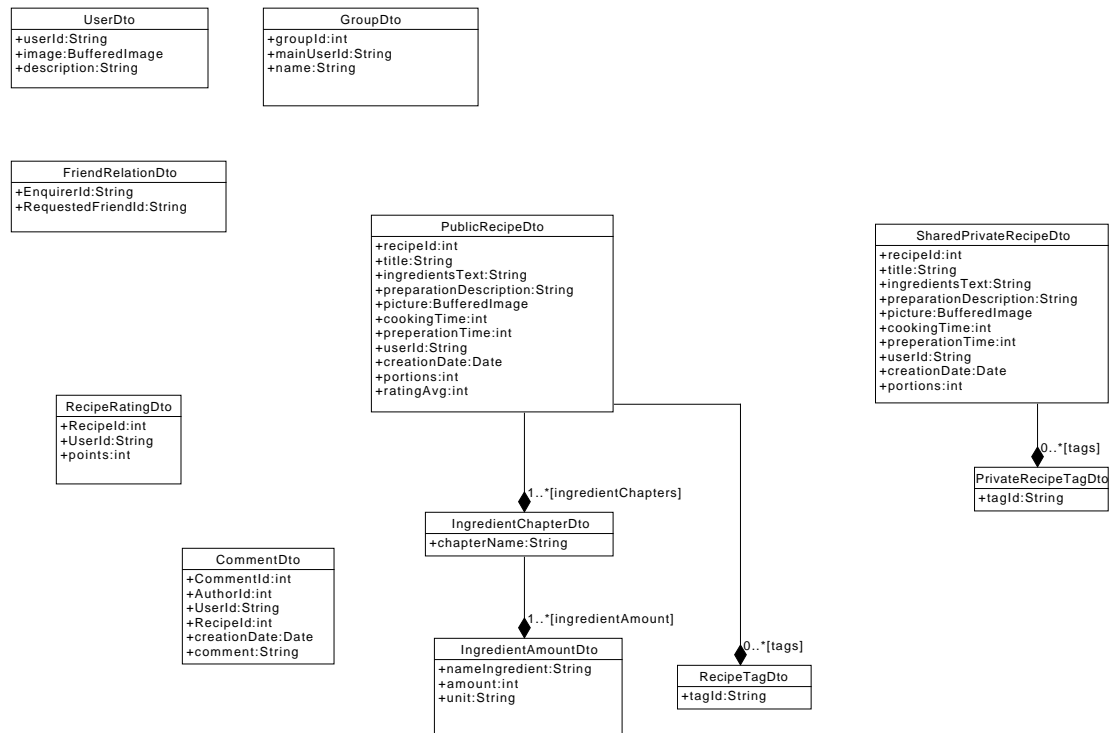


Abbildung 9.1: DTOs

Das Diagramm 13.2 modelliert alle DTOs (Data transfer objects). Es ist in größerer Ansicht am ende des Dokuments erneut angefügt. Zu sehen sind Klassen, deren Hauptzweck es ist Daten zu halten. Diese Objekte, werden über die RESTschnittstelle empfangen und gesendet. Zur Verdeutlichung der Aggregatbeziehung im UML-Diagramm, IngredientChapters, RecipeTags und PrivateRecipeTags werden im Recipe als geschachtelte Listen versendet. Sie brauchen deswegen in der RESTschnittstelle keine gesonderte API, Comments hingegen nicht. Sie können über eine gesonderte API gelesen und verändert werden.

Um das anschaulicher zu machen ist unten ein Bratapfelrezept mit drei Tags und einem seiner IngredientChapter "Zutaten Vanillesauce" als JSON serialisiert dargestellt:

```

1 {
2   "recipeId": 3,
3   "title": "Bratapfel",
4
5   ...
6
7   "ingredientChapters": [
8     {
9       "chapterName": "Zutaten Vanillesauce",
10      "IngredientAmount": [
11        {
12          "nameIngredient": "Zucker",

```

```
13         "amount": 2,  
14         "unit": "Essl."  
15     },  
16     {  
17         "nameIngredient": "Milch",  
18         "amount": 500,  
19         "unit": "Milliliter"  
20     }  
21 ]  
22 }  
23 ]  
24 "tags": [ "Nachtisch", "Weihnachten", "Familienklassiker" ]  
25 }
```


- «get/set»-userId:String
Die BenutzerId des Nutzers
- «get/set»-image:BufferedImage
Profilbild des Nutzers
- «get/set»-description:String
Personenbeschreibung des Nutzers
- «get/set»-groups:List<Group>
Liste von Gruppen in denen sich der Nutzer befindet
- «get/set»-favourites:List<PublicRecipe>
Liste von Rezept-Favoriten eines Nutzers
- «get/set»-markAsEvil:Boolean
Wird gesetzt wenn der Nutzer gemeldet wird

Group

Diese Klasse beschreibt eine Gruppe, die ein Nutzer erstellt hat.

Attribute

- «get»-groupId:int
Id der Gruppe
- «get»-mainUserId:User
Der Nutzer, der die Gruppe erstellt hat
- «get/set»-name:String
Name der Gruppe
- «get/set»-members:List<User>
Liste an Nutzern die in der Gruppe sind
- «get/set»-markAsEvil:Boolean
Wird gesetzt wenn die Gruppe gemeldet wird

FriendRequest

Diese Klasse beschreibt Freundesbeziehung zwischen zwei Nutzern.

Attribute

- «get/set»-enquirer:User
Nutzer der die Freundesanfrage stellt
- «get/set»-requestedFriend:User
Nutzer der Angefragt wird
- «get/set»-state:state
Status in dem sich die Anfrage befindet (ACCEPTED, DENIED, OPEN)

SharedPrivateRecipe

Diese Klasse repräsentiert ein privates Rezept das in einer Gruppe geteilt wurde.

Attribute

- «get»-recipeId:int
Eindeutige Identifikation eines öffentlichen Rezeptes
- «get/set»-title:String
Title eines Rezeptes
- «get/set»-ingredientsText:String
Alle Zutaten als zusammenhängender Text
- «get/set»-preparationDescription
Zubereitungsbeschreibung eines Rezeptes
- «get/set»-picture:BufferedImage
Bild des Gerichts
- «get/set»-cookingTime:int
Dauer für das Kochen
- «get/set»-preperationTime:int
Dauer für die Zubereitung
- «get»-user:User
Nutzer der das Rezept in der Gruppe geteilt hat
- «get/set»-creationDate:Date
Das Datum an dem das Rezept erstellt wurde
- «get/set»-markAsEvil:Boolean
Wird gesetzt wenn das Rezept gemeldet wird
- «get/set»-portions:int
Anzahl für wie viel Personen das Rezept gedacht ist

- «get/set»-group:Group
Gruppe in der sich das Rezept befindet

PrivateRecipeTag

Diese Klasse repräsentiert ein Tag für ein Rezept.

Attribute

- «get/set»-tagId:String
Name des Tags
- «get/»-recipe:SharedPrivateRecipe
Rezepte zu dem der Tag zugewiesen ist

PublicRecipe

Diese Klasse beschreibt ein öffentliches Rezept, welches von der Datenbank geladen oder gespeichert wird.

Attribute

- «get»-recipeId:int
Eindeutige Identifikation eines öffentlichen Rezeptes
- «get/set»-title:String
Title eines Rezeptes
- «get/set»-ingredientsText:String
Alle Zutaten als zusammenhängender Text
- «get/set»-preparationDescription
Zubereitungsbeschreibung eines Rezeptes
- «get/set»-picture:BufferedImage
Bild des Gerichts
- «get/set»-cookingTime:int
Dauer für das Kochen
- «get/set»-preperationTime:int
Dauer für die Zubereitung
- «get»-user:User
Nutzers, dem das Rezept gehört

- «get/set»-creationDate:Date
Das Datum an dem das Rezept erstellt wurde
- «get/set»-markAsEvil:Boolean
Wird gesetzt wenn das Rezept gemeldet wird
- «get/set»-portions:int
Anzahl für wie viel Personen das Rezept gedacht ist
- «get/set»-userFav:List<User>
Nutzer die das Rezept favorisiert haben

RecipeRating

Diese Klasse repräsentiert ein Bewertung für ein Rezept.

Attribute

- «get»-recipe:Recipe
Rezept zu dem die Bewertung gehört
- «get»-user:User
Nutzer der die Bewertung abgegeben hat
- «get/set»-points:int
Punkte die ein Nutzer zu dem Rezept gegeben hat

Comment

Diese Klasse repräsentiert ein Kommentar für ein Rezept.

Attribute

- «get»-idComment:int
Id des Kommentars
- «get»-user:User
Nutzer der den Kommentar erstellt hat
- «get»-recipe:PublicRecipe
Rezept zu dem der Kommentar zugewiesen ist
- «get/set»-comment:String
Text des Kommentares

- «get/set»-markAsEvil:Boolean
Wird gesetzt wenn der Kommentar gemeldet wird
- «get»-date:Date
Datum wann der Kommentar erstellt wurde

PublicRecipeTag

Diese Klasse repräsentiert ein Tag für ein Rezept.

Attribute

- «get/set»-tagId:String
Name des Tags
- «get/»-recipe:PublicRecipe
Rezept zu dem der Tag zugewiesen ist

IngredientChapter

Diese Klasse repräsentiert ein Kapitel von den Zutaten für ein Rezept.

Attribute

- «get»-chapterId:int
Die Id des Kapitels
- «get»-recipe:PublicRecipe
Rezept zu dem ein Kapitel zugewiesen ist
- «get/set»-ingredientChapterTitle:String
Der Titel des Kapitels

IngredientAmount

Diese Klasse repräsentiert eine Zutat von einem Kapitel für ein Rezept.

Attribute

- «get»-chapter:IngredientChapter
Kapitel zu dem die Zutat zugeordnet ist
- «get/set»-nameIngredient:String
Name der Zutat

- «get/set»-amount:int
Menge der Zutat
- «get/set»-unit:String
Einheit der Zutat (Eßl., Tel., g, Liter, Strück, ...)

10.1.2 Dao der Datenbank für den Server



Abbildung 10.2: Klassendiagramm für die Daos für den Datenbankzugriff auf dem Server

GroupDao«interface»

Diese Klasse gibt Schnittstellen für Gruppen in der Datenbank an.

Methoden

- +createGroup(group:Group):void
Fügt die übergebene Gruppe hinzu
- +updateGroup(group:Group):void
Aktualisiert einen Eintrag mit der übergebenen Gruppe
- +deleteGroup(groupId:int):void
Löscht eine Gruppe mit der entsprechenden Id
- +getGroup(groupId:int):Group
Gibt die Gruppe mit der entsprechenden Id zurück

- +addUserToGroup(user:User,group:Group):void
Fügt einen Nutzer einer Gruppe hinzu
- +getGroups(userId:String):List<Group>
Gibt die Gruppen eines Nutzers zurück
- +removeUserFromGroup(userId:String, groupId:int):void
Entfernt einen Nutzer von einer Gruppe
- +setMarkAsEvil(groupId:int, evil:boolean):void
Setzt den Zustand, wenn eine Gruppe gemeldet wurde

UserDao«interface»

Diese Klasse gibt Schnittstellen für Nutzer in der Datenbank an.

Methoden

- +createUser(user:User):void
Fügt den übergebenen Nutzer hinzu
- +updateUser(user:User):void
Aktualisiert einen Nutzer mit dem übergebenen Nutzer
- +deleteUser(idUser:String):void
Löscht einen Nutzer mit der entsprechenden Id
- +getUser(idUser:String):User
Gibt einen Nutzer mit der entsprechenden Id zurück
- +getUniqueID():String
Gibt eine Eindeutige Id für einen Nutzer zurück
- +setFavourite(fav:Favourites):void
Fügt ein Favorit hinzu
- +deleteFavourite(idUser:String, idRecipe:int):void
Löscht einen Favoriten
- +getFavourites(idUser:String):List<Recipe>
Gibt die Liste an Favoriten von einem Nutzer zurück
- +setMarkAsEvil(userId:int, evil:boolean):void
Setzt den Zustand, wenn ein Nutzer gemeldet wurde
- +searchUser(userId:String, count:int):List<User>
Sucht bestimmte Nutzer mit der übergebenen Id

SharedPrivateRecipeDao«interface»

Diese Klasse gibt Schnittstellen für die SharedPrivateRecipes in der Datenbank an.

Methoden

- +addRecipe(recipe:SharedPrivateRecipe):void
Fügt ein neues geteiltes privates Rezept hinzu
- +deleteRecipe(idRecipe:int):void
Löscht ein Rezept mit der entsprechenden Id
- +getRecipe(idRecipe:int):SharedPrivateRecipe
Gibt ein Rezept mit der entsprechenden Id zurück
- +getTags(idRecipe:int):PrivateRecipeTag
Gibt die Liste von Tags eines Rezeptes zurück
- +getSharedRecipe(groupId:int):List<Recipe>
Gibt eine Liste von Rezepten von einer Gruppe mit der entsprechenden Id zurück
- +addSharedRecipe(groupId:int, sharedRecipeId:int):void
Fügt ein geteiltes privates Rezept einer Gruppe hinzu
- +deleteSharedRecipe(groupId:int, sharedRecipeId:int):void
Entfernt ein Rezept von einer Gruppe
- +setMarkAsEvil(sharedRecipeId:int, evil:boolean):void
Setzt den Zustand, wenn ein geteiltes private Rezept gemeldet wurde

FriendRequestDao«interface»

Diese Klasse gibt Schnittstellen für die Freundschaftsanfragen in der Datenbank an.

Methoden

- +addRequest(request:FriendRequestDao):void
Fügt die übergebene Anfrage hinzu
- +deleteRequest(idEnquirer:String, idRequestedFriend:String):void
Löscht die übergebene Anfrage
- +getRequestForUser(idRequestedFriend:String):List<String>
Gibt eine Liste von Nutzern, die einen Nutzer angefragt haben
- +getFriends(idUser:String):List<String>
Gibt die Liste an Freunden eines Nutzers zurück

- +setRequest(idEnquirer:String, idRequestedFriend:String, state:String):void
Setzt den Status einer Anfrage

PublicRecipeDao«interface»

Diese Klasse gibt Schnittstellen für die öffentlichen Rezepte in der Datenbank an.

Methoden

- +addRecipe(recipe:PublicRecipe):void
Fügt das übergebene öffentliche Rezept hinzu
- +updateRecipe(recipe:PublicRecipe):void
Aktualisiert ein Rezept mit dem übergebenen Rezept
- +deleteRecipe(idRecipe:int):void
Löscht ein Rezept mit der entsprechenden Id
- +getRecipe(idRecipe:int):PublicRecipe
Gibt ein Rezept mit der entsprechenden Id zurück
- +getRecipeChapter(idRecipe:int):List<IngredientChapter>
Gibt eine Liste von Kapiteln von einem Rezept mit der entsprechenden Id zurück
- +getComments(idRecipe:int):List<Comment>
Gibt eine Liste von Kommentaren von einem Rezept mit der entsprechenden Id zurück
- +getTags(idRecipe:int):List<RecipeTag>
Gibt die Tags eines Rezeptes mit der entsprechenden Id zurück
- +getAvgRating(idRecipe:int):int
Gibt die Durchschnittsbewertung eines Rezeptes zurück
- +setMarkAsEvil(publicRecipeId:int, evil:boolean):void
Setzt den Zustand, wenn ein Rezept gemeldet wurde

FollowerDao«interface»

Diese Klasse gibt Schnittstellen für Follow-Einträge in der Datenbank an.

Methoden

- +addFollow(subscriber:String, followed:String):void
Fügt ein neuer Einträge für einen Follow ein

- +getFollows(userId:String):List<String>
Gibt eine Liste von Nutzern zurück, die der Nutzer folgt
- +removeFollow(subscriber:String):void
Löscht einen Eintrag in der Tabelle mit den entsprechenden Nutzern

CommentDao«interface»

Diese Klasse gibt Schnittstellen für Kommentaren von Rezepten in der Datenbank an.

Methoden

- +addComment(comment:Comment):void
Fügt ein neuen Kommentar einem Rezept hinzu
- +updateComment(comment:Comment):void
Aktualisiert einen Kommentar
- +deleteComment(idComment:int):void
Löscht einen Kommentar mit der entsprechenden Id
- +setMarkAsEvil(commentId:int, evil:boolean):void
Setzt den Zustand, wenn ein Kommentar gemeldet wurde
- +getComments(recipeId:int):List<Comment>
Gibt eine Liste von Kommentaren von einem Rezept zurück

IngredientChapterDao«interface»

Diese Klasse gibt Schnittstellen für Kapitel von Rezepten in der Datenbank an.

Methoden

- +addChapter(chapter:IngredientChapter):void
Fügt ein neues Kapitel einem Rezept hinzu
- +editChapter(chapter:IngredientChapter):void
Editiert ein Kapitel
- +deleteChapter(idChapter:int):void
Löscht ein Kapitel mit der entsprechenden Id
- +getIngredientAmount(idChapter:int):List<IngredientAmount>
Gibt eine Liste von Zutaten von einem Kapitel zurück

IngredientAmountDao«interface»

Diese Klasse gibt Schnittstellen für Zutaten von Kapiteln in der Datenbank an.

Methoden

- +addIngredient(ingredient:IngredientAmount):void
Fügt ein neue Zutat einem Kapitel hinzu
- +updateIngredient(ingredient:IngredientAmount):void
Aktualisiert eine Zutat von einem Kapitel
- +deleteIngredient(idChapter:int, nameIngredient:String):void
Löscht eine Zutat von einem Kapitel

RecipeRatingDao«interface»

Diese Klasse gibt Schnittstellen für Bewertungen von Rezepten in der Datenbank an.

Methoden

- +addRating(rating:RecipeRating)
Fügt eine Bewertung hinzu von einem Rezept
- +updateRating(rating:RecipeRating)
Aktualisiert eine Bewertung von einem Rezept

10.2 Datenbank

10.2.1 ServerDatenbank

User

Die Relation User beinhaltet Informationen über die Profile der angemeldeten Nutzer. Die Relation User wird über einen Fremdschlüssel in den Relationen UserGroup, Group, SharedPrivateRecipe, PublicRecipe, Favourites, RecipeRating und FriendRequests adressiert.

- userId
Eindeutige Kennung eines angemeldeten Nutzers, mit dem er sich anmeldet (Primärschlüssel)
- image
Profilbild der Nutzers

- description
Textuelle Beschreibung des Profils
- markAsEvil
Markiert gemeldete Nutzer

FriendRequest

Die Relation FriendRequest beinhaltet Informationen über noch ausstehende Freundschaftsanfragen. Die Relation FriendRequest verweist selbst auf die Relation User.

- enquirerId
BenutzerID des anfragenden Nutzers (Fremdschlüssel)
- requestedFriendId
BenutzerID des angefragten Nutzers (Fremdschlüssel)
- Status

Group

Die Relation Group beinhaltet Informationen über Freundesgruppen. Die Relation Group verweist auf die Relation User.

- groupId
Eindeutige Kennung einer Gruppe (Primärschlüssel)
- mainUserId
BenutzerID des Erstellers der Gruppe (Fremdschlüssel)
- name
Name der Gruppe
- markAsEvil
Markiert gemeldete Gruppen

UserGroup

Die Relation UserGroup weist den Freundesgruppen jeweils die Mitglieder zu. Die Relation UserGroup verweist selbst auf die Relation Group und User.

- groupId
definiert um welche Gruppe es sich handelt (Fremdschlüssel)
- userId
BenutzerID, welcher in der Gruppe mit groupId ist

SharedPrivateRecipe

Die Relation SharedPrivateRecipe beinhaltet Informationen über geteilte private Rezepte in Freundesgruppen. Die Relation SharedPrivateRecipe verweist über groupId zu einer Gruppe und verweist selbst auf die Relation User.

- **recipeId**
Eindeutige Kennung des Rezeptes in dieser Tabelle (Primärschlüssel)
- **title**
Titel des Rezeptes
- **ingredientsText**
Textuelle Beschreibung der Zutaten
- **preparationDescription**
Textuelle Beschreibung der Zubereitung
- **picture**
Bild des Rezeptes
- **cookingTime**
Dauer der Kochzeit in Minuten
- **preparationTime**
Dauer der Vorbereitung in Minuten
- **userId**
BenutzerID des Benutzers, der dieses Rezept geteilt hat (Fremdschlüssel)
- **creationDate**
Datum und Uhrzeit der Versendung des Rezeptes
- **markAsEvil**
Markiert gemeldete Rezepte
- **portions**
Anzahl Portionen für die das Rezept ausgelegt ist
- **groupId**
Verlinkung zu welcher Gruppe das Rezept gehört

PrivateRecipeTag

Die Relation PrivateRecipeTag weist jedem privaten Rezept, welches in einer Freundesgruppe geteilt wurde, eine Menge an Tags zu. Die Relation PrivateRecipeTag verweist selbst auf die Relation SharedPrivateRecipe.

- **tagId**
Name des Tags
- **recipeId**
Rezept, zu dem der Tag gehört (Fremdschlüssel)

PublicRecipe

Die Relation PublicRecipe beinhaltet Informationen über veröffentlichte Rezepte. Die Re-

lation PublicRecipe wird über einen Fremdschlüssel in den Relationen RecipeRating, Favourites, IngredientChapter, Comment, RecipeTag und SharedPrivateRecipe adressiert und verweist selbst auf die Relation User.

- recipeId
Eindeutige Kennung eines Rezeptes (Primärschlüssel)
- title
Titel des Rezeptes
- ingredientsText
Textuelle Beschreibung der Zutaten
- preparationDescription
Textuelle Beschreibung der Zubereitung
- picture
Bild des Rezeptes
- cookingTime
Dauer der Kochzeit in Minuten
- preparationTime
Dauer der Vorbereitung in Minuten
- userId
BenutzerID des Benutzers, der dieses Rezept geteilt hat (Fremdschlüssel)
- creationDate
Datum und Uhrzeit der Veröffentlichung des Rezeptes
- markAsEvil
Markiert gemeldete Rezepte
- portions
Anzahl Portionen für die das Rezept ausgelegt ist

RecipeTag

Die Relation RecipeTag weist jedem veröffentlichten Rezept eine Menge an Tags zu. Die Relation RecipeTag verweist selbst auf die Relation PublicRecipe.

- tagId
Name des Tags
- recipeId
Rezept, zu dem der Tag gehört (Fremdschlüssel)

Comment

Die Relation Comment weist jedem veröffentlichten Rezept eine Menge an Kommentaren

zu. Die Relation Comment verweist selbst auf die Relationen und User und PublicRecipe.

- commentId
Eindeutige Kennung des Kommentars
- userId
Nutzer, der den Kommentar geschrieben hat
- recipeId
Rezept, zu welchem der Kommentar gehört
- commit
Der eigentliche Kommentar
- markAsEvil
Markiert gemeldete Kommentare
- date
Das Datum wann der Kommentar stellt wurde

IngredientChapter

Die Relation IngredientChapter weist jedem veröffentlichten Rezept eine Menge an Unterkapiteln der Zutatenliste zu. Die Relation IngredientChapter wird über einen Fremdschlüssel in der Relation IngredientAmount adressiert und verweist selbst auf die Relation PublicRecipe.

- chapterId
Eindeutige Kennung des Unterkapitels
- recipeId
Zeiger auf das Rezept
- ingredientAmount

IngredientAmount

Die Relation IngredientAmount weist einem Unterkapitel der Zutatenliste eines veröffentlichten Rezeptes eine Menge an Zutaten zu. Die Relation IngredientAmount verweist selbst auf die Relation IngredientChapter.

- chapterId
Zeiger auf das Unterkapitel
- nameIngredient
Name der Zutat
- unit
Einheit, in der die Zutat gemessen wird

- amount
Anzahl an unit in der die Zutat benötigt wird

Favourites

Die Relation Favourites weist jedem angemeldeten Nutzer eine Menge an von ihm favorisierten Rezepten zu. Die Relation Favourites verweist selbst auf die Relationen PublicRecipe und User.

- recipeId
Zeiger auf Rezept
- userId
Zeiger auf Benutzer

RecipeRating

Die Relation RecipeRating weist jedem Rezept eine Menge an Bewertungen zu. Die Relation RecipeRating verweist selbst auf die Relationen User und PublicRecipe.

- recipeId
Zeiger auf Rezept
- userId
Zeiger auf Benutzer
- points
Bewertung

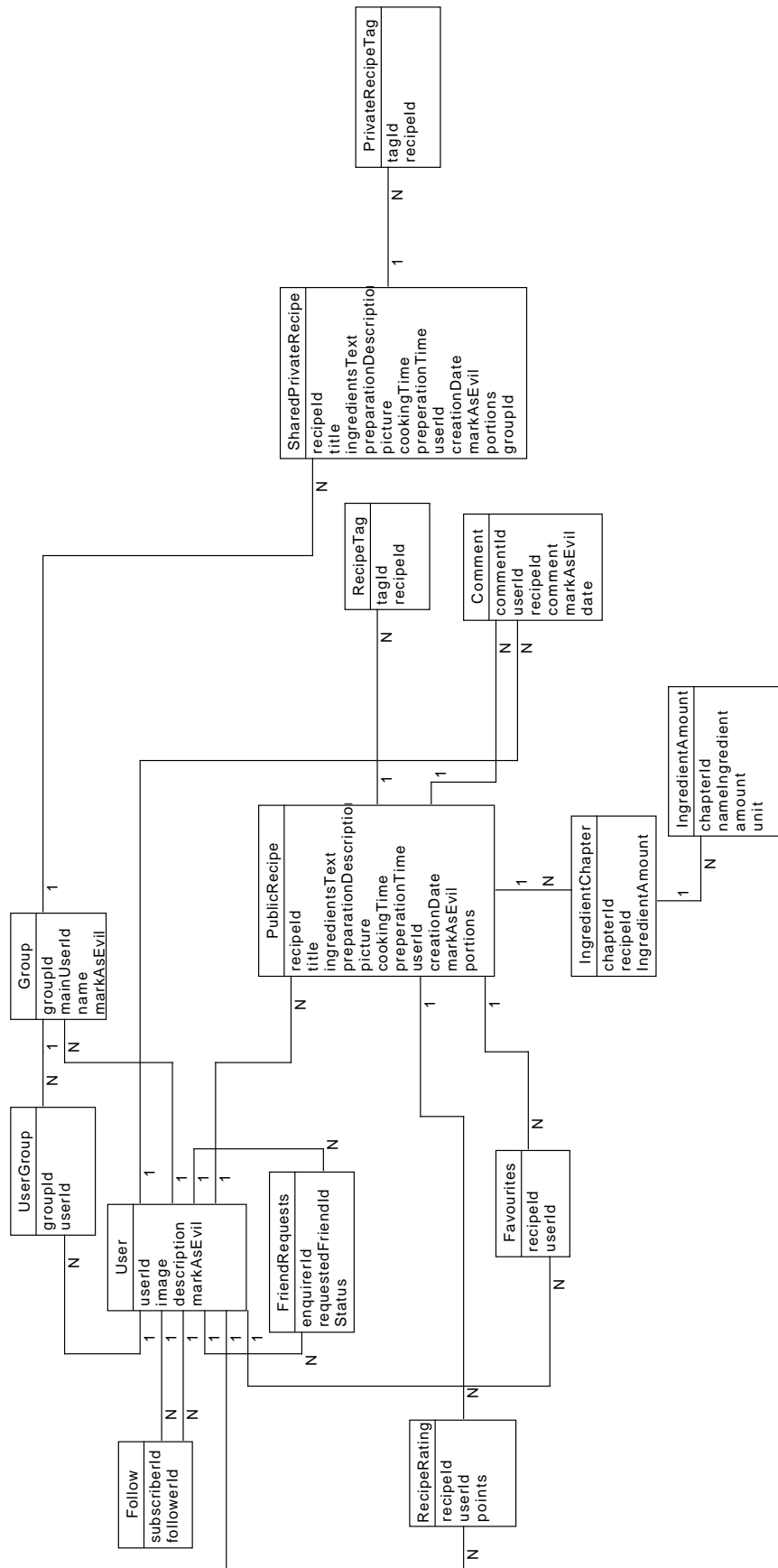


Abbildung 10.3: ER-Diagramm der Serverdatenbank

11 Aktivitäts- und Zustandsdiagramme

11.0.1 Rezept erstellen und veröffentlichen

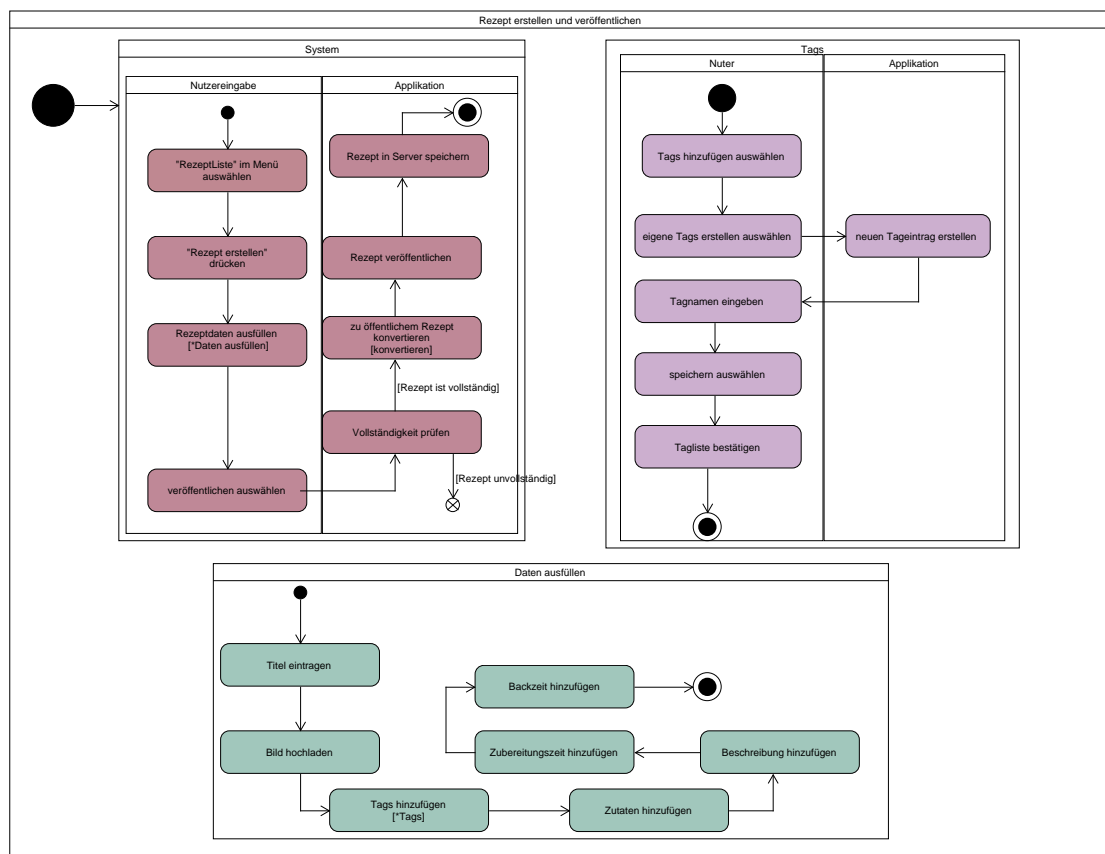


Abbildung 11.1: Rezept erstellen und veröffentlichen

Der Nutzer befindet sich in der Applikation und ist auf der Startseite. Nun will er ein Rezept erstellen und veröffentlichen. Dabei erstellt er noch ein eigenen neuen Tag.

System

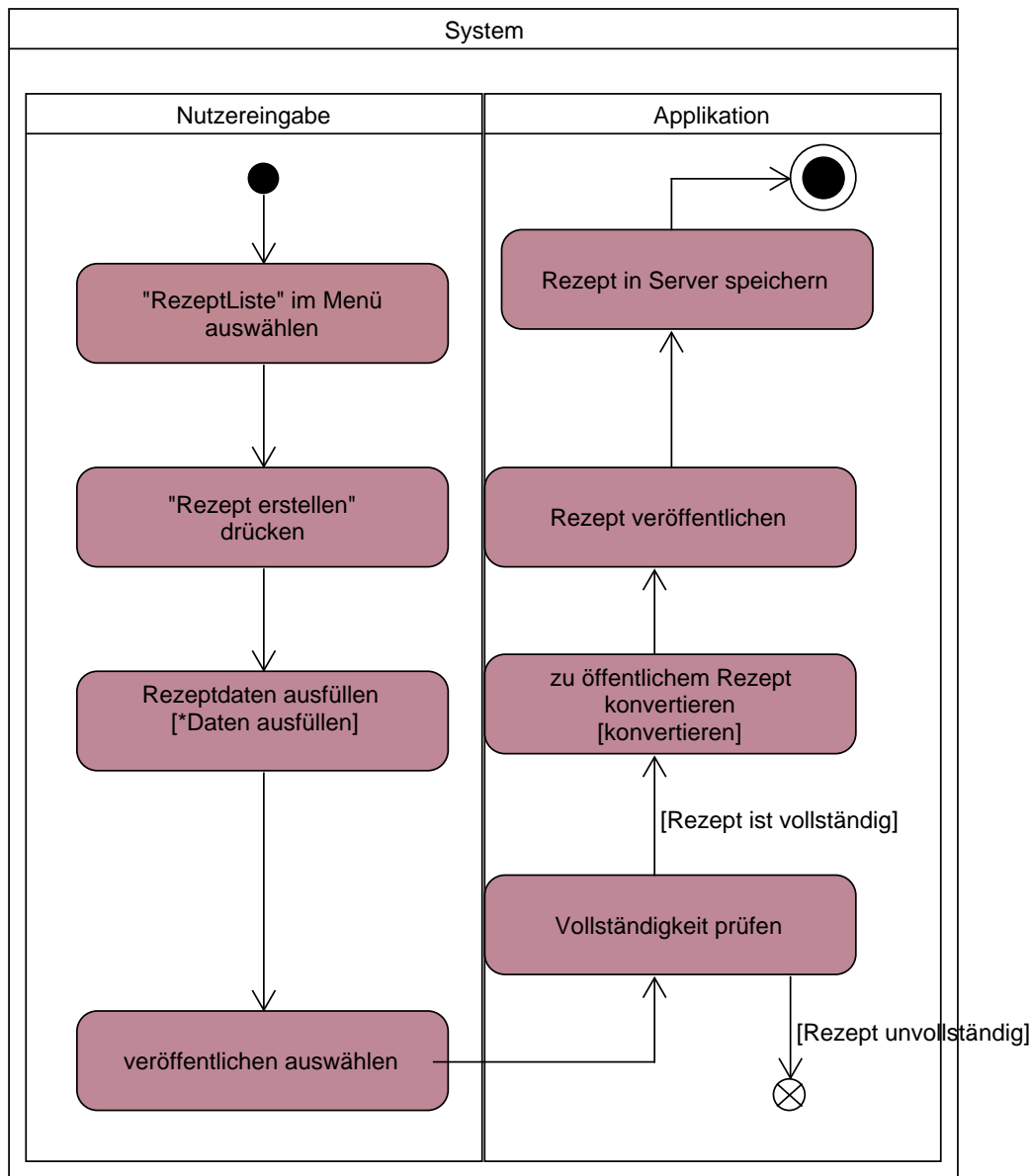


Abbildung 11.2: Rezept erstellen und veröffentlichen - System

Der Nutzer befindet sich in der App Ansicht und wählt im Menü den Menüpunkt "Rezeptliste" aus. Dabei wechselt das Fragment und es wird die Liste an selbst erstellten, vollständigen und unvollständigen Rezepten angezeigt. Neben der Liste befindet sich ein Button mit der Aufschrift "Rezept erstellen", den der Nutzer auswählt. Es öffnet sich daraufhin ein neues Fenster, das ein Rezepteingabe Template anzeigt. (siehe Daten ausfüllen Aktivitätsdiagramm). Als der Nutzer die Daten ausgefüllt hat, wählt er die Option aus, sein Rezept öffentlich anzuzeigen. Daraufhin wird das Rezept im Hintergrund von einem Parser auf Vollständigkeit überprüft. Ist das Rezept nicht vollständig, wird das Rezept in der Rezeptliste gespeichert, zu der der Nutzer im nächsten Schritt geleitet wird. Ist das

Rezept vollständig, wird es zu einem öffentlichen Rezept konvertiert (siehe konvertieren Aktivitätsdiagramm). Nachdem das Rezept in ein öffentliches konvertiert wurde wird es an den Server weitergeleitet, wo es in der Datenbank gespeichert wird.

Daten

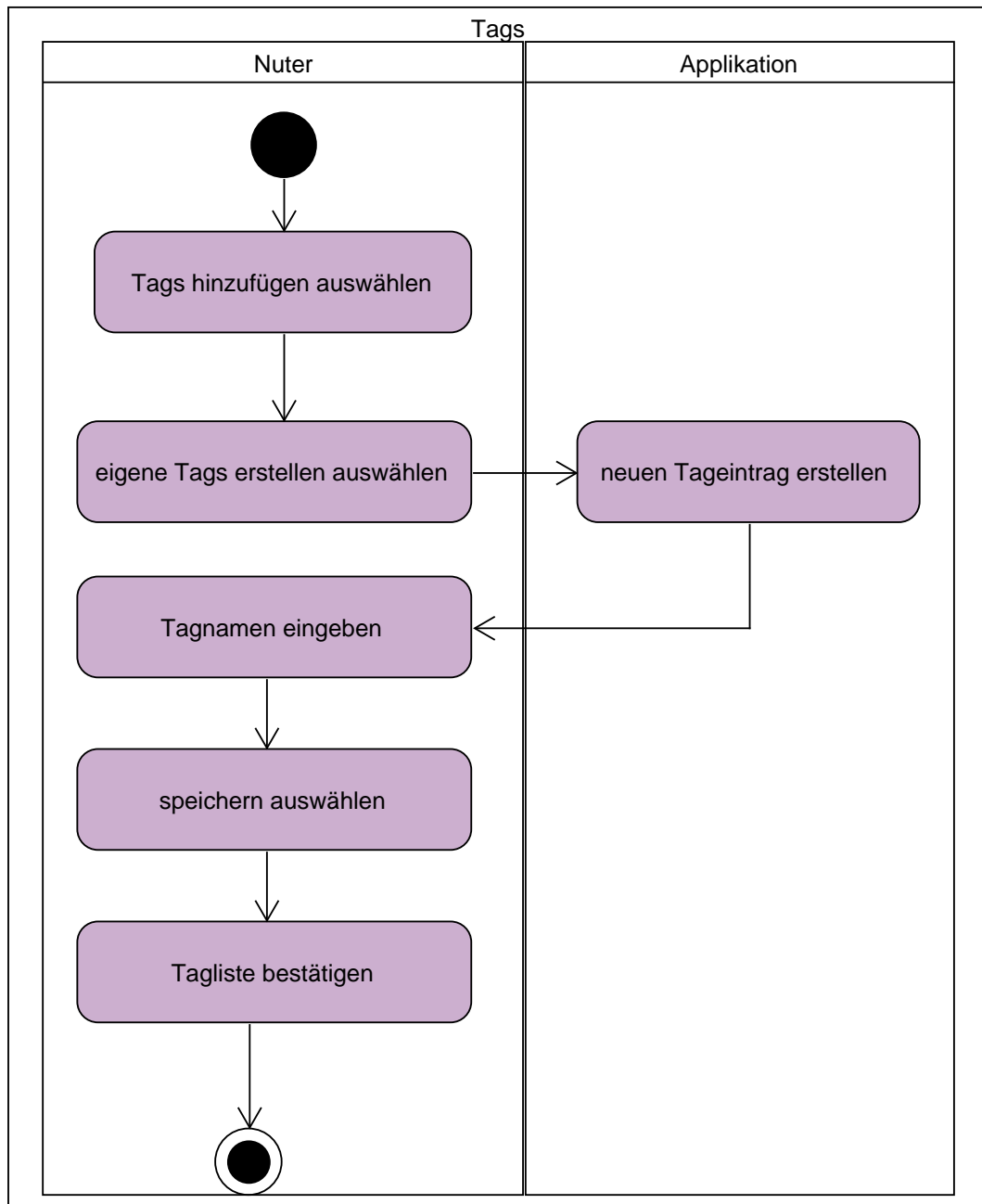


Abbildung 11.3: Rezept erstellen und veröffentlichen - Daten

Der Nutzer befindet sich in der Rezept erstellen Ansicht, wo ihm ein Eingabe Template angeboten wird. Zuerst trägt er einen Titel ein und lädt ein Bild des Rezeptes hoch . Danach wählt er de Button "Tags hinzufügenäus (siehe Tags hinzufügen Aktivitätsdiagramm). Daraufhin fügt er über eine Eingabe Zutaten zu dem Rezept hinzu. Nach den Zutaten beschreibt er die Zubereitung des Rezepts mit einem Freitext in dem dafür vorgesehenen Textfeld. Nun fügt er noch die Zubereitungs- und die Backzeit hinzu und hat damit seine Rezeptdaten vollständig ausgefüllt.

Tags

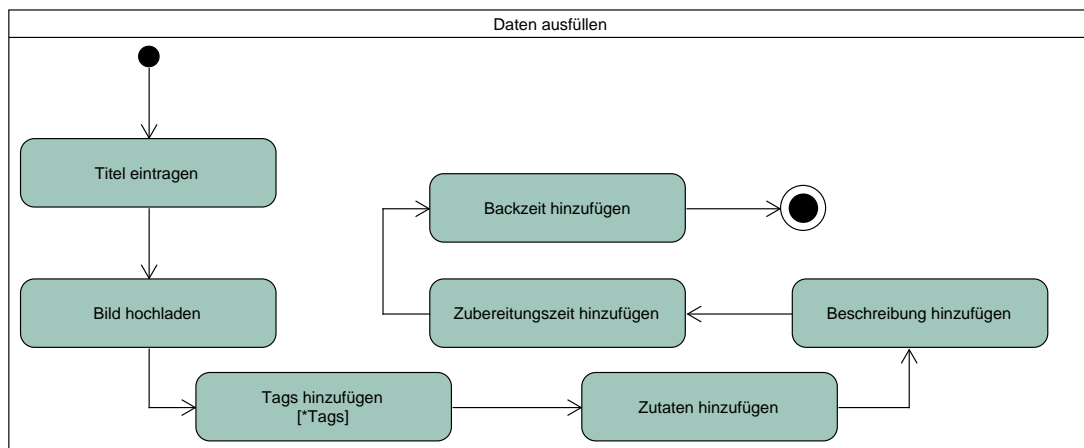


Abbildung 11.4: Rezept erstellen und veröffentlichen - Tags

Der Nutzer befindet sich in der Tag erstellen Ansicht. Zuerst wählt er die Funktion ein Tag hinzuzufügen. Daraufhin wählt er aus einen eigenen Tag zu erstellen. Dafür wird ein neuer Tag Eintrag erstellt, den der Nutzer noch einen Namen gibt. Zuletzt speichert er den neu erstellten Tag noch und bestätigt seine Tagliste für das jeweilige Rezept.

11.0.2 Rezept löschen

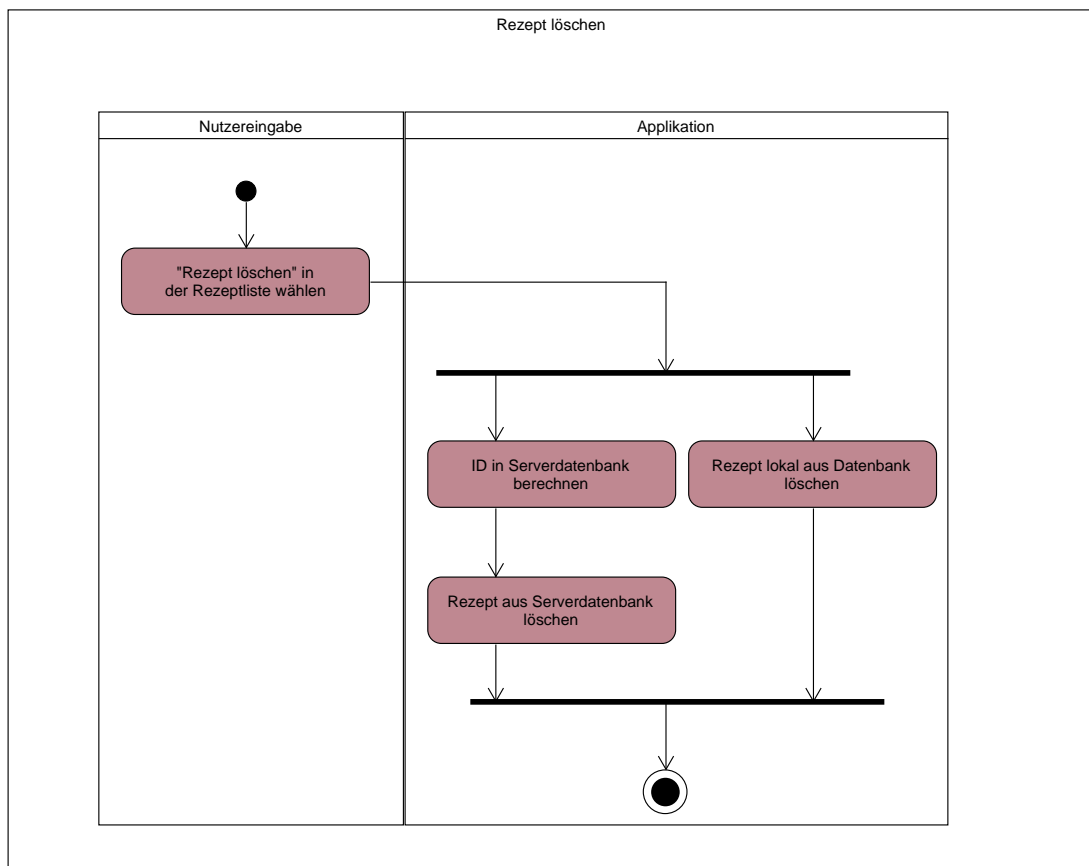


Abbildung 11.5: Rezept löschen

Der Nutzer will ein Rezept entfernen. Dieses kann auch öffentlich sein. Er wählt in der Rezeptliste den Entfernen Button, woraufhin die ID in der Serverdatenbank berechnet wird und das Rezept aus der lokalen Datenbank entfernt wird. Da nach Spezifikation das Rezept auch aus der öffentlichen Datenbank entfernt werden soll, wenn man das private Rezept löscht, wird das Rezept aus der Serverdatenbank gelöscht.

12 Sequenzdiagramme

12.1 Rezept erstellen und veröffentlichen

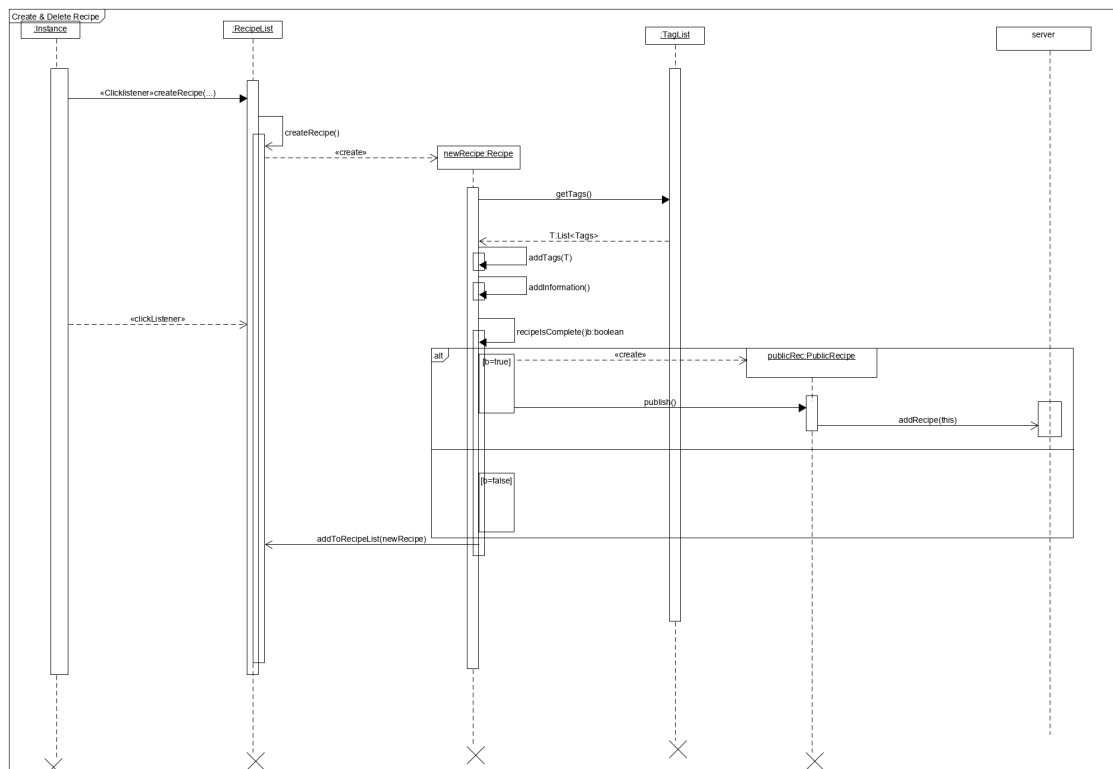


Abbildung 12.1: Create & Publish Recipe Sequence Diagram

Der Nutzer befindet sich in der App, genauer in seiner privaten Rezeptliste. Daraufhin wählt er "Rezept erstellen" aus und wird in das dafür vorgesehene Fragment weitergeleitet. Dort wird ein neues privates Rezept erstellt, mit Tags und Eingaben gefüllt und gespeichert. Da der Nutzer dieses auch veröffentlichen möchte, wird das Rezept auf Vollständigkeit geprüft und in ein öffentliches Rezept konvertiert. Nachdem es veröffentlicht wurde, kehrt der Nutzer zu seiner Rezeptliste zurück, wo das Rezept nun hinzugefügt wurde.

12.2 Lazy Loading

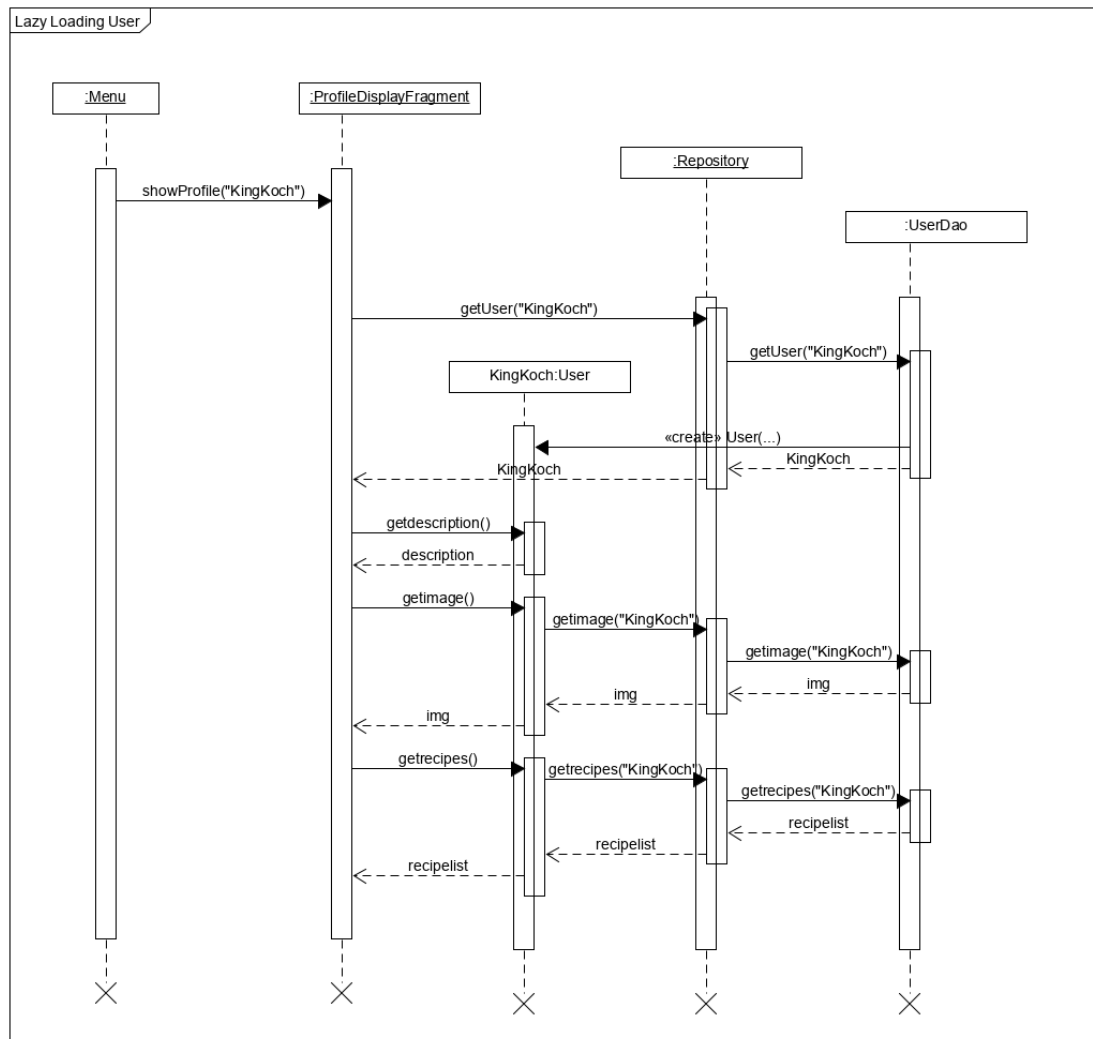


Abbildung 12.2: Lazy Loading User

Wenn ein Profil eines Benutzers angezeigt werden soll, wird ein User Objekt erstellt. Es werden manche Attribute direkt initialisiert, wie zum Beispiel die Zubereitung (`getdescription()`), aber zum Beispiel das Bild und die Kommentare werden erst vom Server geladen, wenn diese auch benutzt werden. Somit wird bei dem `getImage()` Aufruf erst aus dem Repository die Daten geladen und diese dann zurückgegeben.

12.3 Favorit hinzufügen und entfernen

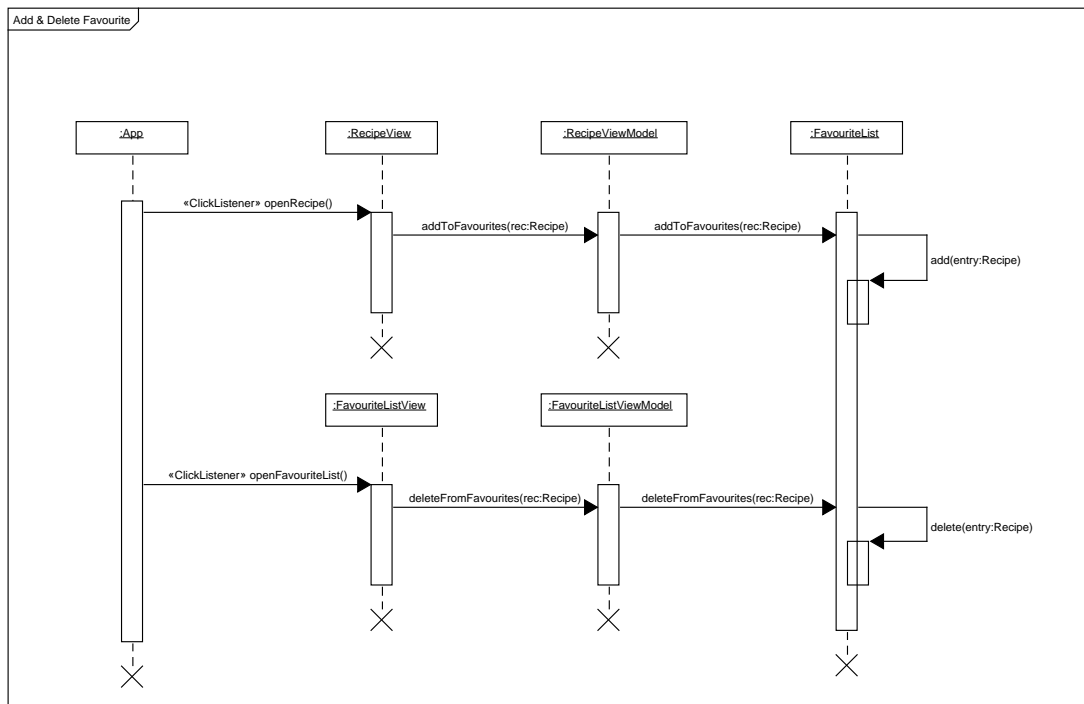


Abbildung 12.3: Favorit hinzufügen und entfernen

Der Nutzer hat in der Suche ein Rezept gefunden und öffnet dieses. In der Rezept Anzeige wählt er aus, dass Rezept zu seinen Favoriten hinzuzufügen. Über das ViewModel kann die Information verarbeitet werden und der Favoritenliste des Nutzers wird jenes Rezept hinzugefügt. Der Nutzer entscheidet sich dafür ein Rezept aus seiner Favoritenliste zu entfernen. Er wählt bei dem Rezept den Entfernen Button, woraufhin über das ViewModel das Rezept aus der Favoritenliste des Nutzers entfernt wird.

13 Klassendiagramm

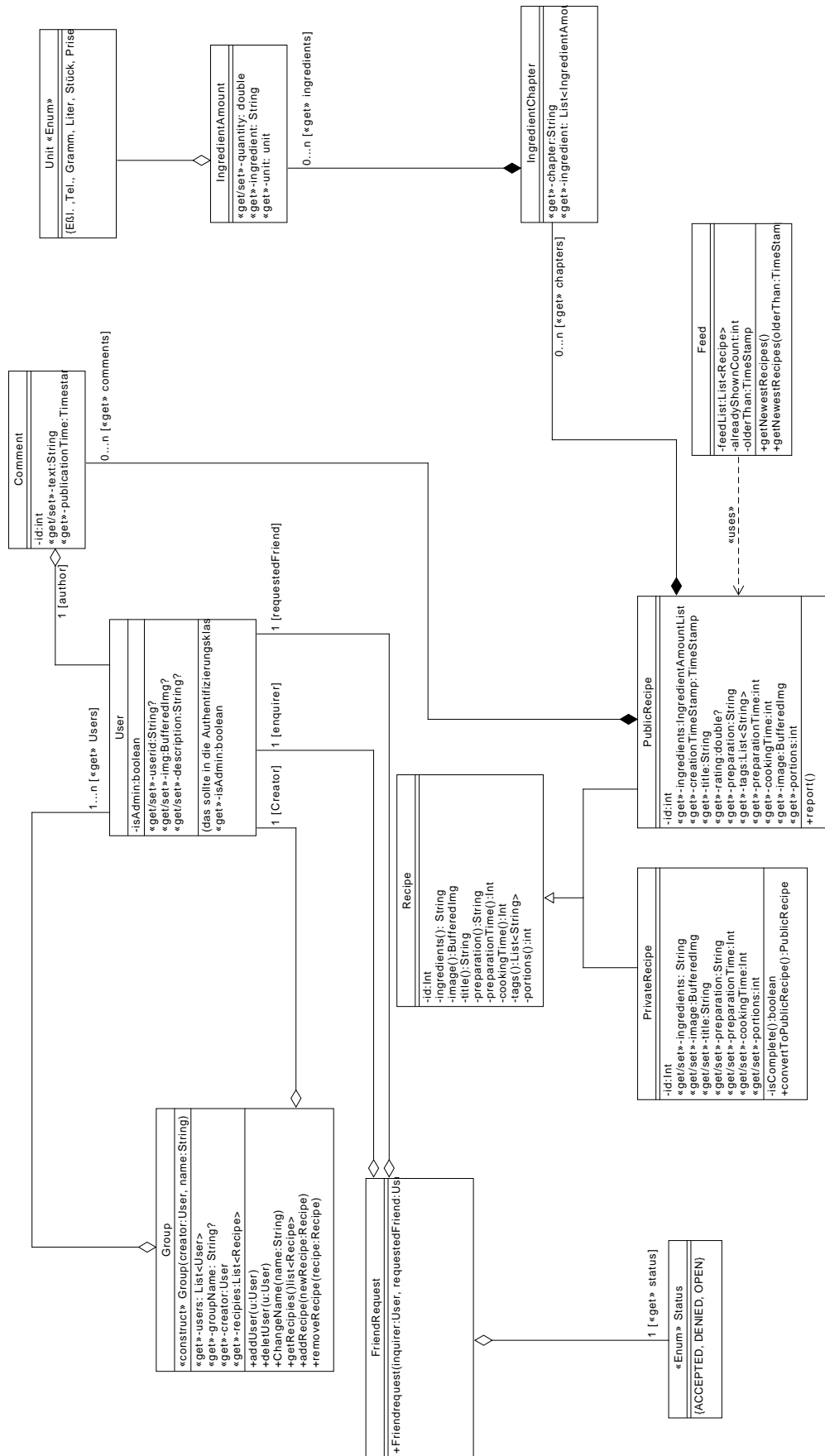


Abbildung 13.1: Domain Entities

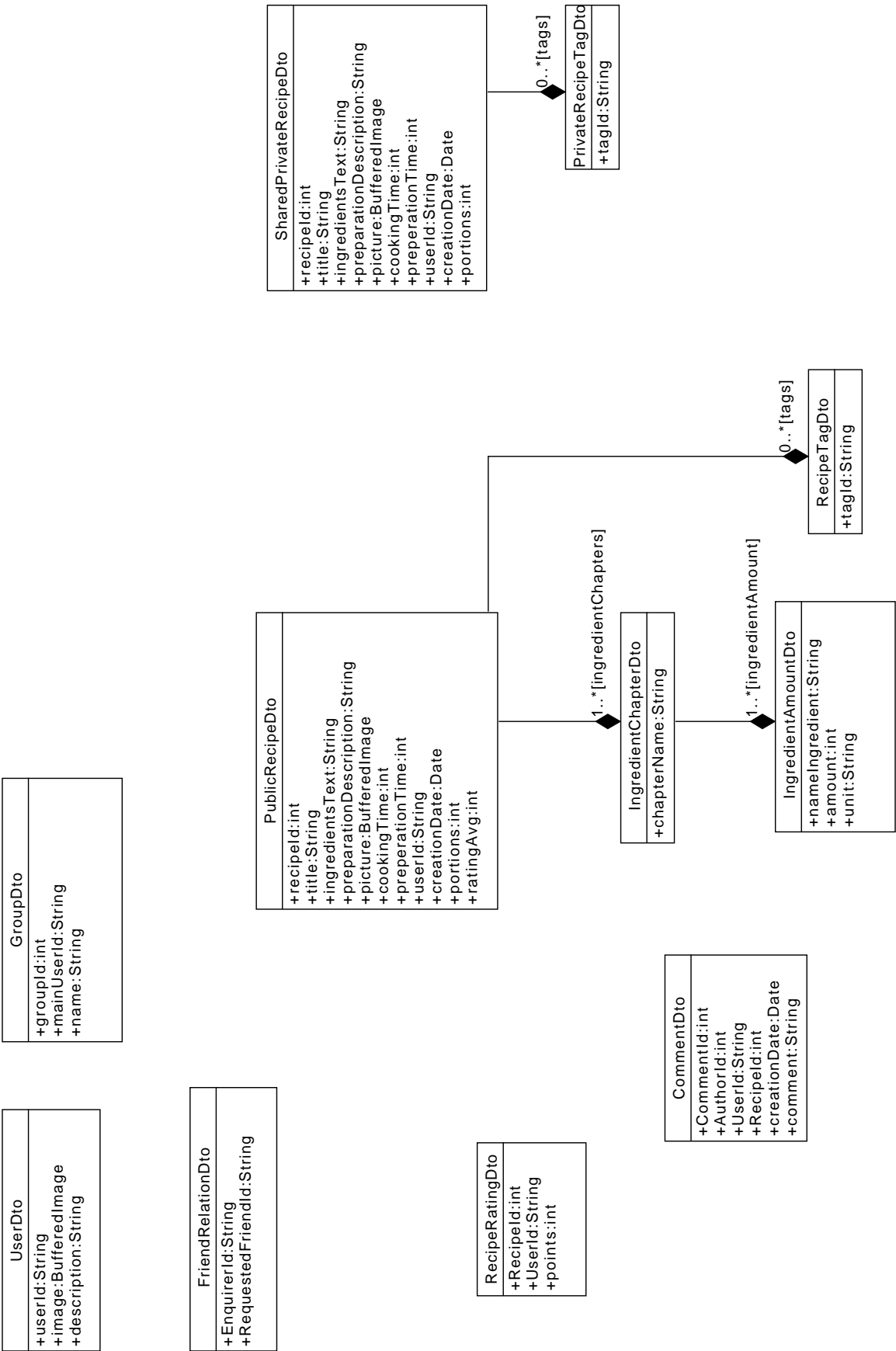


Abbildung 13.2: DTOs

Literaturverzeichnis

- [1] R. C. V. Martin, *Clean Architecture : a craftsmans guide to software structure and design*. Robert C. Martin series, Boston: Prentice Hall, [2018]. "With contributions by James Grenning and Simon Brown ; foreword by Kevlin Henney ; afterword by Jason Gorman".
- [2] R. Verdecchia, I. Malavolta, and P. Lago, "Guidelines for architecting android apps: A mixed-method empirical study," in *2019 IEEE International Conference on Software Architecture (ICSA)*, pp. 141–150, March 2019.
- [3] E. Evans, *Domain-driven design : tackling complexity in the heart of software*. Boston: Addison-Wesley, 1. print. ed., 2004.
- [4] "Spring Core Technologies, spring documentation." <https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html>. Accessed: 2019-12-22.
- [5] "Android Architecture Components, android developer documentation." <https://developer.android.com/topic/libraries/architecture>. Accessed: 2019-12-22.