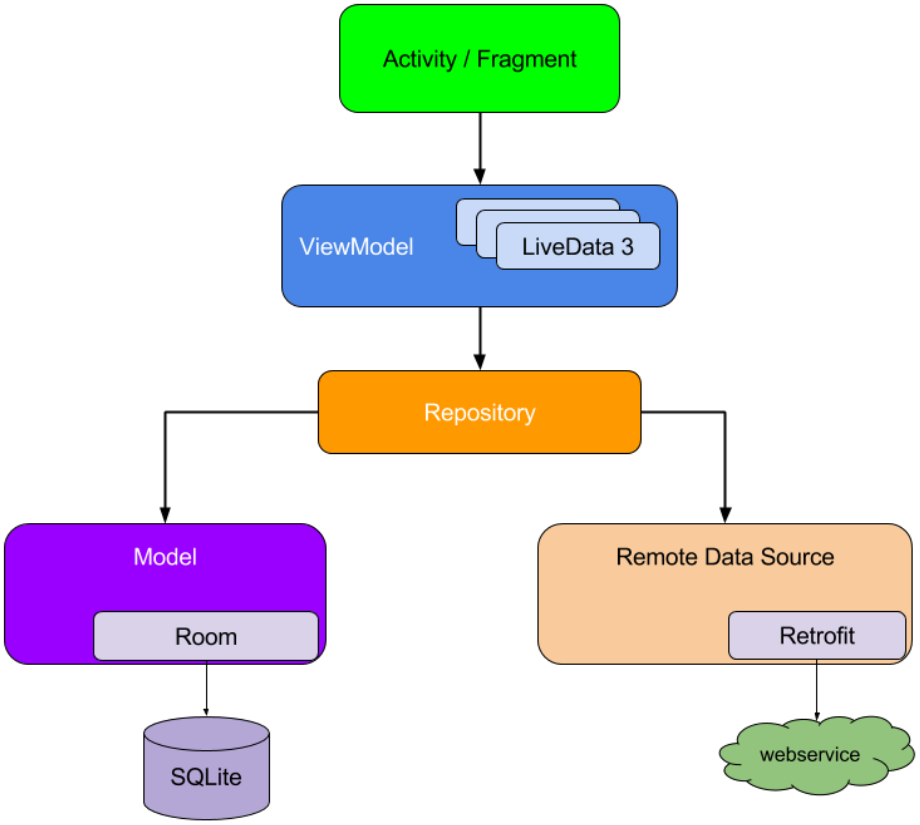
Software Dokumentation

Client Architektur



Dem Nutzer am nächsten ist das “**View**”. Dies ist die UI. Sowohl deren .xml (das UI Layout), als auch das Fragment/die Activity (UI Controller).

Als nächstes kommt das **ViewModel**. Diese Ebene versorgt die View-Ebene mit Daten und bearbeitet Datenbezogene Nutzeranfragen. Das heißt dass das Anpassen der Daten/Informationen an/für das jeweilige View-Element ebenfalls hier stattfinden und nicht weiter unten. Das ViewModel ist an die jeweiligen Komponenten des Views gebunden. Allerdings nur vom View aus, d.h. das ViewModel kennt die View-Ebene nicht.

Darunter sitzt das **Repository**. Es stellt der App die jeweilig benötigten Daten simpel bereit. Von hier werden die eigentlichen Datenquellen aufgerufen. Das Caching passiert ebenfalls hier.

Unter dem **Repository** stehen die Datenspeicherorte. Eine “**Remote Data Source**”, in unserem Fall ein Webservice, und eine **lokale Speichermethode**, **Room**.

Die lokale Speichermethode/Datenbank Room ist bei der App als “einzelne Quelle an Wahrheit” gehalten. D.h. Webservice Antworten werden in die Datenbank gespeichert und sofort alle Verweise aktualisiert, sodass Probleme wie Formatierungsunterschiede bei “gleichen Daten” behoben werden.

D.h. selbst wenn Daten vom Webservice gezogen werden (statt Cache) werden diese erst in die Datenbank eingespeist, dann aus dieser vom Repository an das ViewModel weitergeleitet.

Kommunikation zwischen Client und Server

Models des Clients:

Die Models des Webservices sind 3 Kategorien unterteilt, die erste Kategorie sind Requests, diese werden mit einem Postrequest abgeschickt, um einen Eintrag in der Datenbank des Servers anzulegen. Sie haben dieselben Attribute wie die der Requestklassen des Servers. Die zweite Kategorie sind die Webservicemodels, das sind die Entity Klassen des Servers, wenn beispielsweise der Client eine Story anfordert, dann sendet der Server ein Objekt dieser Klasse als JSON String, nachdem Empfangen dieses Strings wird es in ein Objekt der Serverklasse wieder umgewandelt. Die dritte Kategorie ist der Webresponse, diese wird von dem Server zurückgesendet, wenn ein Login oder Registerrequest abgeschickt wurde. Diese Klasse besitzt 2 Variablen, zum einen “success”, der den Erfolg des Einfügens in der Datenbank als Boolean hinterlegt und des weiteren “message”, dort ist der JWT Token in enthalten.

Es wurde ein Converter implementiert, der die Models des Servers in die Models der Roomdatenbank konvertiert, um diese denn in die Roomdatenbank zu speichern.

Kommunikation:

Die Kommunikation für die storyrelevanten Objekten werden über die Storyrepository durchgeführt, die userrelevanten Objekten über die Userrepository. Beide Repository werden jeweils als Singleton im Viewmodel instanziiert. Die clientseitige Webkommunikation wird mit dem Framework “Ktor” realisiert.

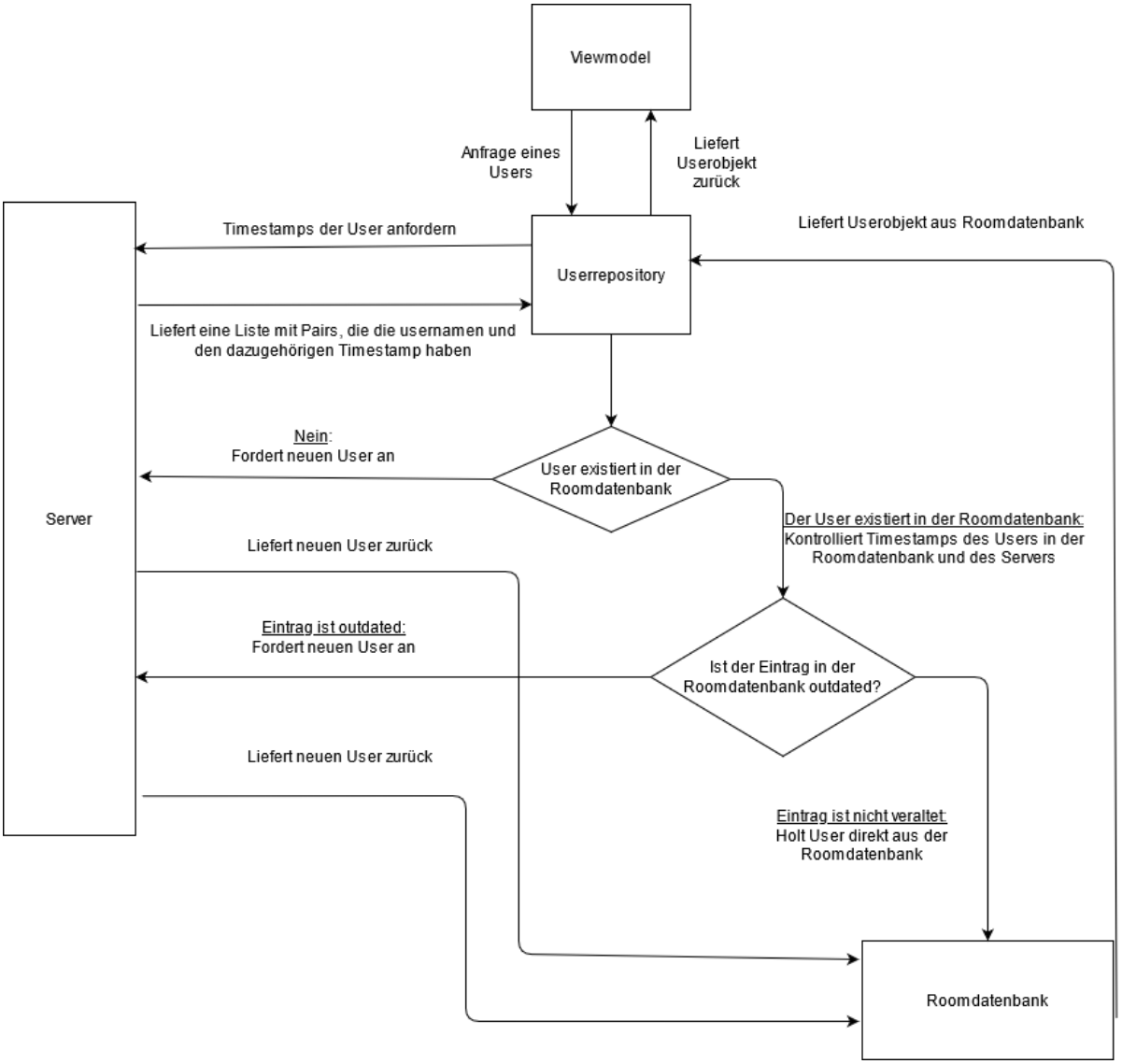
Wenn die UI etwas aus der Datenbank abruft, werden zuerst die Timestamps der angefragten Objekte von dem Server angefordert und nicht sofort das ganze Objekt sofort, um den Datenverkehr zu minimieren. Danach wird kontrolliert, ob es das Objekt in der Roomdatenbank existiert, falls es nicht existiert, wird das Objekt neu angefordert und in der mit der aktuellen Zeit als Timestamp gespeichert, falls es in der Roomdatenbank vorhanden ist, wird der Timestamp des serberseitigen Datenbankeintrages des Objektes mit dem des Timestamp des Objektes in der Roomdatenbank verglichen, falls der Eintrag in der Roomdatenbank veraltet ist, wird ein neues Objekt angefordert, falls das Objekt aus der Roomdatenbank aktuell ist, wird dieses zurückgegeben und kein neues angefordert. Der Rückgabetyp der Funktionen an die View sind immer Klassen aus der Roomdatenbank.

Um den asynchronen Programmablauf sicherzustellen werden “coroutines” verwendet. Coroutines werden in den Klammern, die nach dem Schlüsselwort “runBlocking” kommen, ausgeführt. Innerhalb dieses Blockes können asynchrone Funktionsaufrufe getätigt werden, wie zum beispiel Anfragen an den Server senden und die Antwort empfangen.

Die Funktionen, die in die Roomdatenbank schreiben bzw. aus dieser lesen, sind in einem “withContext(Dispatchers.IO)” Block geschrieben, um die Funktion auf einem Nebenthread

laufen zulassen, da auf dem Mainthread nicht in die Roomdatenbank geschrieben werden kann.

Beispiel eines Kommunikationsablaufes:



Backend:

Backend-Architektur:

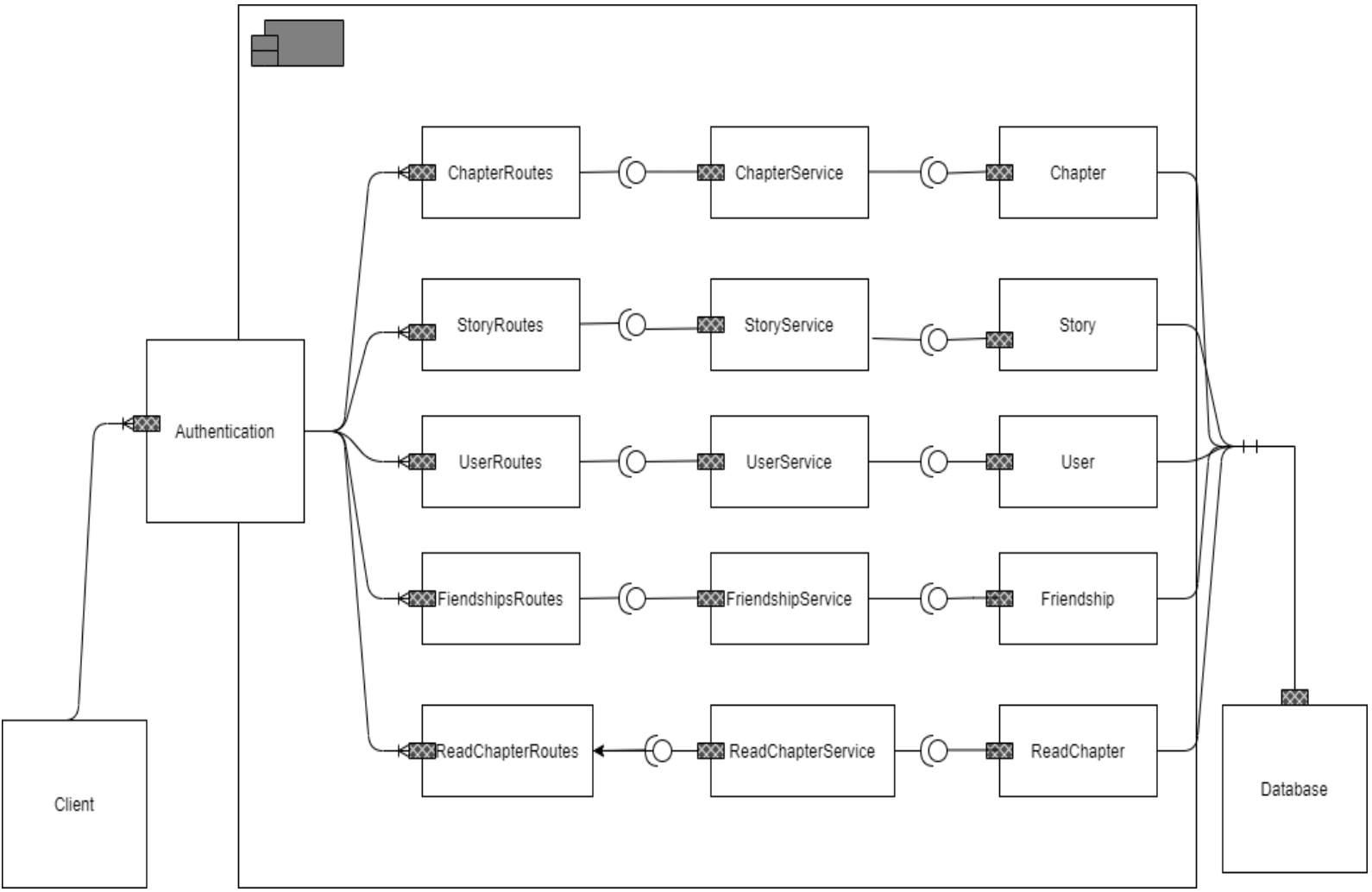
Die Architektur der Server-Anwendung basiert sich auf 5 haupt Schichten-Architektur.

Das Application-server tier besitzt alle Mechanismen zur Verarbeitung der Anfragen von dem Client. Hier ist die Anwendungslogik vereint. Dazu gehören die Authentification, Rotes, die Models sowie die Services.

Der fünfte Schicht ist die Datenhaltungsschicht. Dieser Schicht beinhaltet die Datenbank und die Entitäten, die in Exposed geschrieben sind, und ist für das Speichern und Laden von Daten verantwortlich. Dies ist genau in dem Verzeichnis table bestanden.

Diese Architektur ist skalierbar, da für das Einfügen eines weiteren Models, nur eine neue Tabelle (und eine Entität), ein Service und eine Route erstellt werden muss.

Das folgende Komponentendiagramm stellt ein Überblick über die einzelnen Komponenten bzw. der Klassen und der Datenbank bis zum Client dar, die in der Kommunikation zusammenspielen.

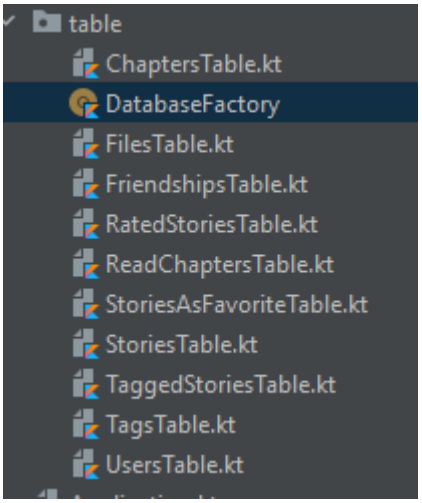


Backend-Kotlin-Komponenten:

Das „Kotlin-Komponent“, enthält die folgende Hauptmodule:

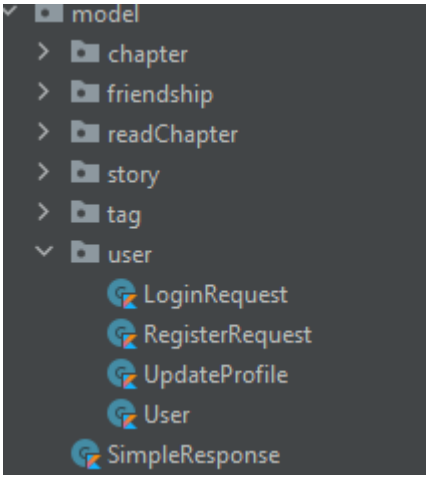
1. Table:

Table Modul ergänzt mit Hilfe von „Exposed.dao“ die Tabellen, für das Speichern der Daten von: Users, Stories, Chapters, ReadChapters, Rated Stories, Friendships, Tagged Stories, Favorite Stories. Jede .kt Datei enthält die Tabelle und die Entität eines Models. Die Tabellen sind von dem Typ “UUIDTable”. Die ID’s werden selbst zufällig generiert. Der Tabelle User wird beispielsweise die Daten von einem Student speichern, Die Daten wären „username“, „email“, „hashPassword“, “image”, “description”, “lastUpdate” ,und hat eine „ID“ und “username” als Primärschlüssel.



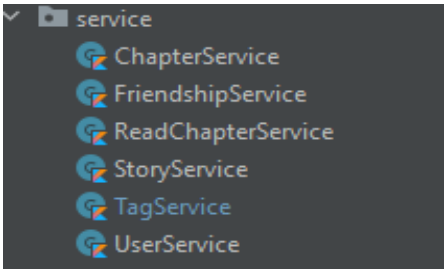
2. Models:

Model sind serialisierbare Klassen, die von den Entitäten der Datenbank sowie die Routes benutzt werden. Jede Entität besitzt die Funktion “toDTO”, die Objekte von Models erstellt werden. Hier sind die Kern Klassen, für die prinzipalen Nutzer bzw. Funktionalitäten zu finden.



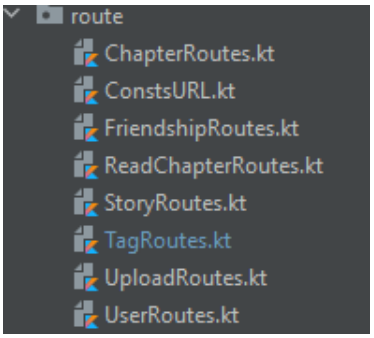
3. Services:

   Der “Service“ beinhaltet die Services bzw. die Funktionen für die jeweiligen Klassen, und stellt diese für die „Routes“ zur Verfügung. Ermöglicht beispielsweise das Schreiben, Entfernen oder Auslesen der Daten von der Datenbank. Für jede Entität wird ein Service benötigt.



4. Routes:

   Routing ist die Kern „KTOR“ Funktion für eingehende Anfragen in unser Server-Anwendung. Wenn der Nutzer eine Anfrage an eine bestimmte URL stellt, kann man mithilfe des Routing-Mechanismus definieren, wie diese Anfrage bearbeitet werden soll. Eine Route greift mittels dem Service auf die Datenbank. Beispielsweise wäre (/getAll) in „StoryRoutes“ der Pfad für das Aufrufen aller Stories, und dabei wird die zur Verfügung gestellte „StoryService.getAll()“ Funktion aufgerufen und alle Stories werden in einer Liste gespeichert und im JSON-Format zum Frontend zurückgeschickt.



5. Main (Applikation.kt):

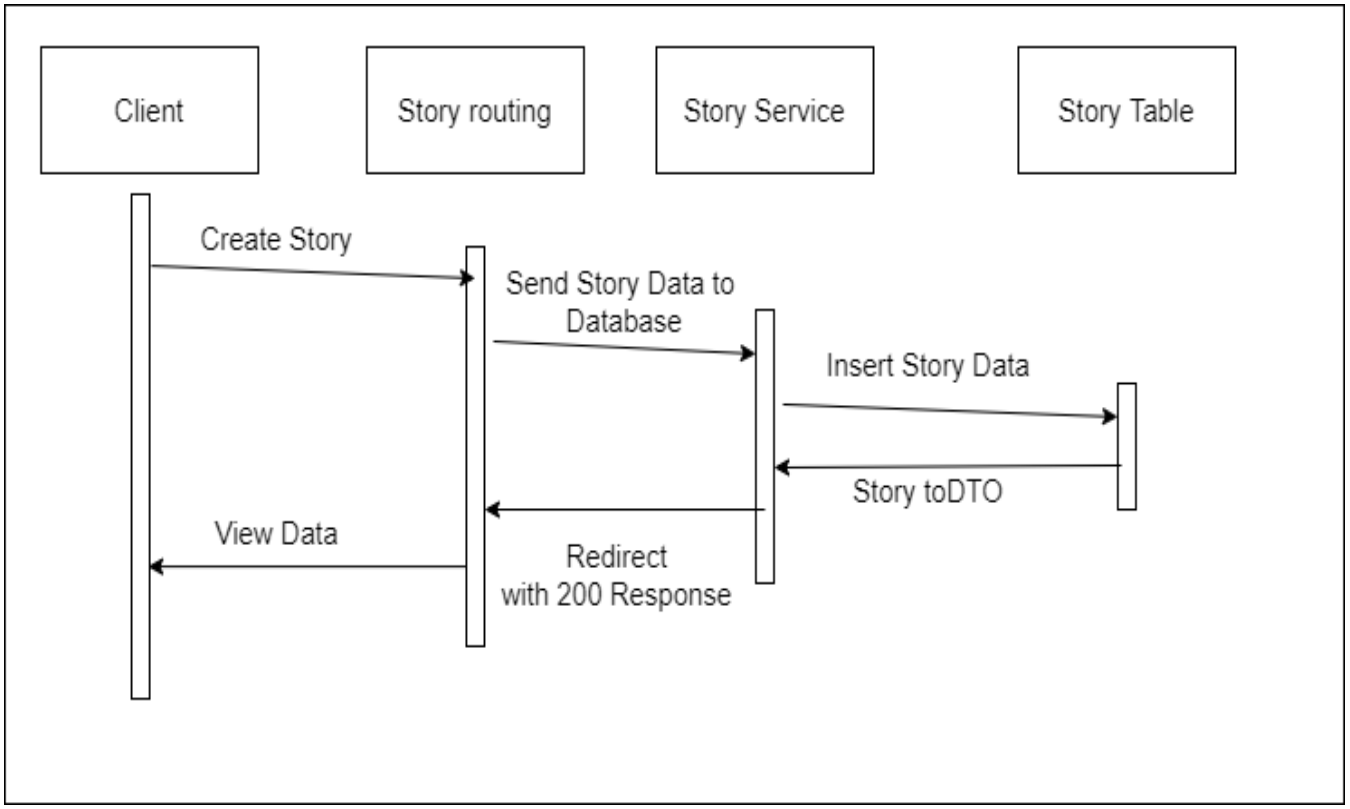
   Hier wird das Mainmodule definiert, was alle andere Modules kontrolliert. Von hier wird auch die Main-Methode aufgerufen. Sie ist sozusagen der Controller der Module (View und Backend).

Backend-Beispiel für die Funktionsweise:

1. Story erstellen:

   Wenn der “Create Story” Button in der Client-Seite geklickt wird, werden die Story-Daten in Json-Format in einem Post-Request an den Server gesendet. Der Benutzername wird hier an der Anfrage für die Authentication angehängt.

Im StoryRote.kt wird die Anfrage bearbeitet. Der StoryRoute erstellt über den StoryService den Story. Mit der Function transaction{..} in dem StoryService kann auf die StoryTable zugegriffen werden und das Story wird in der Datenbank erstellt. Bei erfolgreicher Storyerstellung wird ein 200 ok Status mit einem passenden Message von der aufgerufene Route zurückgegben, wenn das Story nicht erstellt werden kann, wird ein Fehlermeldung an den Nutzer zurükgegeben.
Um zum Beispiel das bereits erstellte Story anzufragen, wird die passende Route aufgerufen. Die Route ruft die passende Funktion in dem Service, und der Service greift auf die Tabelle zu. StoryEntity Klasse wird hier verwendet, was über die Funktion “toDTO” verfügt, die ein Objekt der Klasse Story mit den gespeicherten Daten zurückliefert. Die aufgerufene Route erhält das vom der “toDTO” ersetllte Objekt zurück, und sendet es zum Nutzer als serialisierte Objekt (Json-Format).



Backend-Technologieentscheidung:

1. Kotlin (Ktor):

   Bei Kotlin (Ktor) ist das end-to-end Multiplattform-Anwendungs-Framework für die Arbeit an Webapplikationen, HTTP-Services sowie Mobile-und Browser-Anwendungen konzipiert. Für komplexe asynchrone Anwendungsfälle verwendet Ktor Koroutinen.

2. Kotlin (Exposed) und SQL:

   Exposed ist eine kompakte Bibliothek über dem JDBC-Treiber für die Kotlin-Sprache**.** Exposed unterstützt mehrere Datenbanksprachen unter anderem SQLite, während die Exposed Syntax bei allen Datenbanksprachen gleich bleibt und kann als mittlere Schicht fungieren und Abfragen in der DAO (Data Access Object) API einfach zu erlernen. Zudem stellt SQLLite die Achievability**.**

   Vorteile von Exposed:
   - Ein “Code” funktioniert mit allen unterstützten Datenbanken.
   - Kein SQL-Code selbst nötig.
   - Der SQL-Code kann als Logging- Statement ausgegeben werden.
   - Open-Source Apache License.

3. Routing Requests:

   Routing ist ein Feature, welches in der Applikation installiert wird und zur einfacheren Strukturierung von Page-Requests dient. Es gibt folgende Funktionen für Routing-Requests: Get, Post, Put, Delete, Head, Options Nach der Installation des Routings, öffnet man ein Routing. Hier kann man nun Requests einbauen - diese können willkürlich oder in einem sogenannten Routing-Tree angeordnet sein (https://ktor.io/docs/routing.html#routing-tree).

```
fun Application.module() {

        routing {

                post("/insert") { .....

                                }

                }

        }
```

4. Schichtenarchitektur:

   Eine Schicht kann nur mit einer unteren oder innerhalb derselben Schicht kommunizieren und nicht mit einer darüberliegenden.Mit der Schichten-Architektur ist die Anwendung einfach zu implementieren, und sie ist für ein kleines Budget und geringer Zeit gut geeignet. Die Schichtenarchitektur ist flexibel und erweiterbar, für ein neues Modell fügt man den gehörigen Service, Route und Table ein, ohne die stehende Implementierungen zu beschädigen.

5. JWT Authentication:

   Diese Bibliothek stellt die Authentifizierung der Nutzer sicher. Das heißt, dass beispielsweise nur ein angemeldeter Nutzer ein Story erstellen kann. Bei den

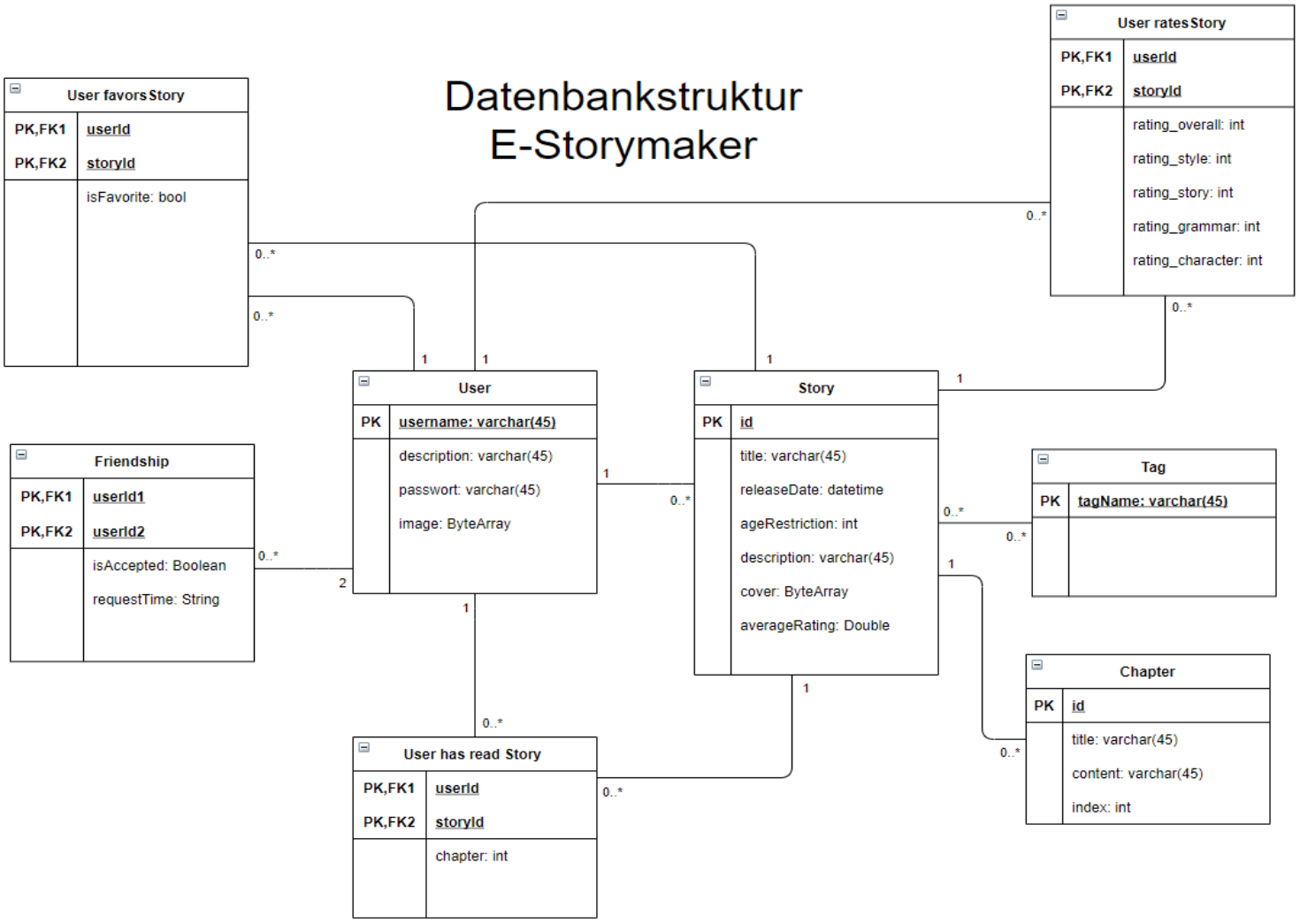
Anfragen oder Story-Erstellung wird den Username des Nutzers mit an den Server gesendet. Serverseitig wird der Username bestätigt und die Geschichte Erstellung oder das Anfragen von erstellten Geschichten wird verifiziert und abgeschlossen. Somit geht der Server-Applikation mit den Nutzern Einzeln um.

Die Room-Datenbank:

Zur Clientseitigen Speicherung der Daten wird eine Room-Datenbank verwendet, welche die Daten beim Schließen und Neustarten der App behält und sich Veränderungen der Server-DB holt, sollten die eigenen Daten zu alt sein.

Das Datenbank-Schema:

Das Schema beider Datenbanken ist nahezu identisch. Die Einträge der RoomDB enthalten jedoch ein weiteres Feld, welches den Zeitpunkt des Caching-vorgangs speichert, um



ermitteln zu können, ob die vorhandenen Einträge erneut von der Serverseitigen Datenbank geladen werden sollten.

Als Primärschlüssel wurde sich auf Strings geeinigt. In Tabellen, welche bereits einen unique String besitzen, namentlich der TagName der Tag-tabelle und der Username der User-tabelle, wurde dieser String als Primärschlüssel verwendet.
Bei der Modellierung der Datenbank wurde darauf geachtet die Regeln zur Normalform bis einschließlich Normalform-3 einzuhalten. Die Tabellen “User rates Story”, “User has read Story” und “User favors Story” voneinander getrennt, obwohl sie dieselben Primärschlüssel verwenden.

Shared Preferences:

Zur Speicherung des Nutzernamens und des JWT, welches verwendet wird um sich auf dem Server zu autorisieren, werden Shared Preferences verwendet, wodurch es möglich ist sich nach schließen und wieder-öffnen der App nicht erneut einloggen zu müssen.