Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Henk van der Sloot
Matrikelnummer: 577448
Harun Dastekin
MatrikelNummer: 551006

Grundlagen mobiler Anwendungen
Meilenstein 4
DoYourDishes

0. Gliederung

Github:
https://github.com/h3nk42/DoYourDishes

1. Projekt Übersicht

Die zentrale Idee unseres Projekts ist es, die wiederkehrend anfallenden und zu erledigenden Aufgaben eines Haushalts in einem Plan darzustellen und gleichzeitig einen Überblick zu schaffen, wer, wie viele Aufgaben erledigt hat. Nutzerinnen sollen sich mit ihren Mitbewohnern über einen Klienten zusammenfinden, um zentral einen Haushaltsplan zu erstellen. In diesem soll es möglich sein Aufgaben anzulegen, darüber zu bestimmen, wer Mitglied ist und es soll einsehbar sein, wer die meisten *Punkte* gesammelt hat, indem jene Aufgaben erledigt wurden. Damit mehrere Benutzerinnen denselben Plan bearbeiten und sehen können, haben wir dies durch einen Android Klienten als Benutzeroberfläche sowie einen NodeJS Backend Dienst zum Austausch und zur Persistierung der Daten ermöglicht. Um Nutzer am Backend Dienst zu unterscheiden, verwenden wir eine Benutzername und Passwort Kombination als Anmeldung.

Die Aufgaben werden nachfolgend in diesem Dokument als "*Tasks*" bezeichnet.
Die Mitglieder bzw. Benutzer des Plans werden nachfolgend in diesem Dokument als "*User*" bezeichnet.

Der **erste Hauptbereich** ist der Android Klient und er besteht aus den drei Bereichen *Login/Register*, *Home und Plan Overview.*

1. *Login/Register* -
   -> Benutzerkonto erstellen oder bei einem Anmelden
2. *Home* -

   -> Benutzerkonto löschen

   -> Falls Benutzerin Mitglied eines Plans: Plan Overview öffnen, Plan löschen

   -> Falls nicht: Plan erstellen
3. *Plan Overview* -

   -> Bietet eine übersicht über den Inhalt des Plans

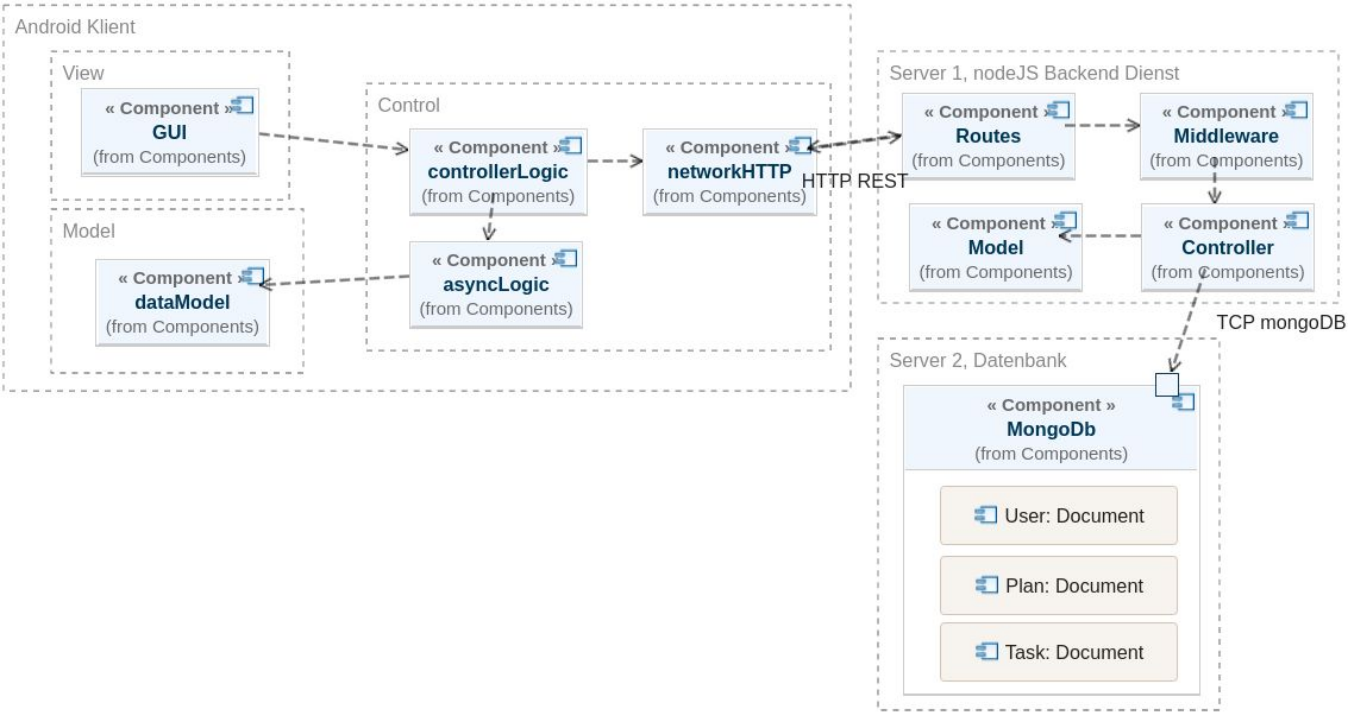
   -> Tasks: Task erstellen, Task löschen, Tasks anzeigen

   -> User: User hinzufügen, User entfernen, Punkte eines Users einsehen

   -> Score: Eine Visualisierung der Punkte der User

Der **zweite Hauptbereich** ist der Backend Dienst, dieser bietet eine REST Programmierschnittstelle, welche über das HTTProtokoll erreichbar ist und greift auf die zur Anwendung gehörende Datenbank per TCProtokoll zu.

2. Komponenten Übersicht und Architektur



Das Entwurfsmuster folgt sowohl im Klienten, als auch im Backend Dienst dem MVC Entwurfsmuster, welches an dieser Stelle ja ausreichend bekannt ist.

3. Beschreibung der Android Komponenten

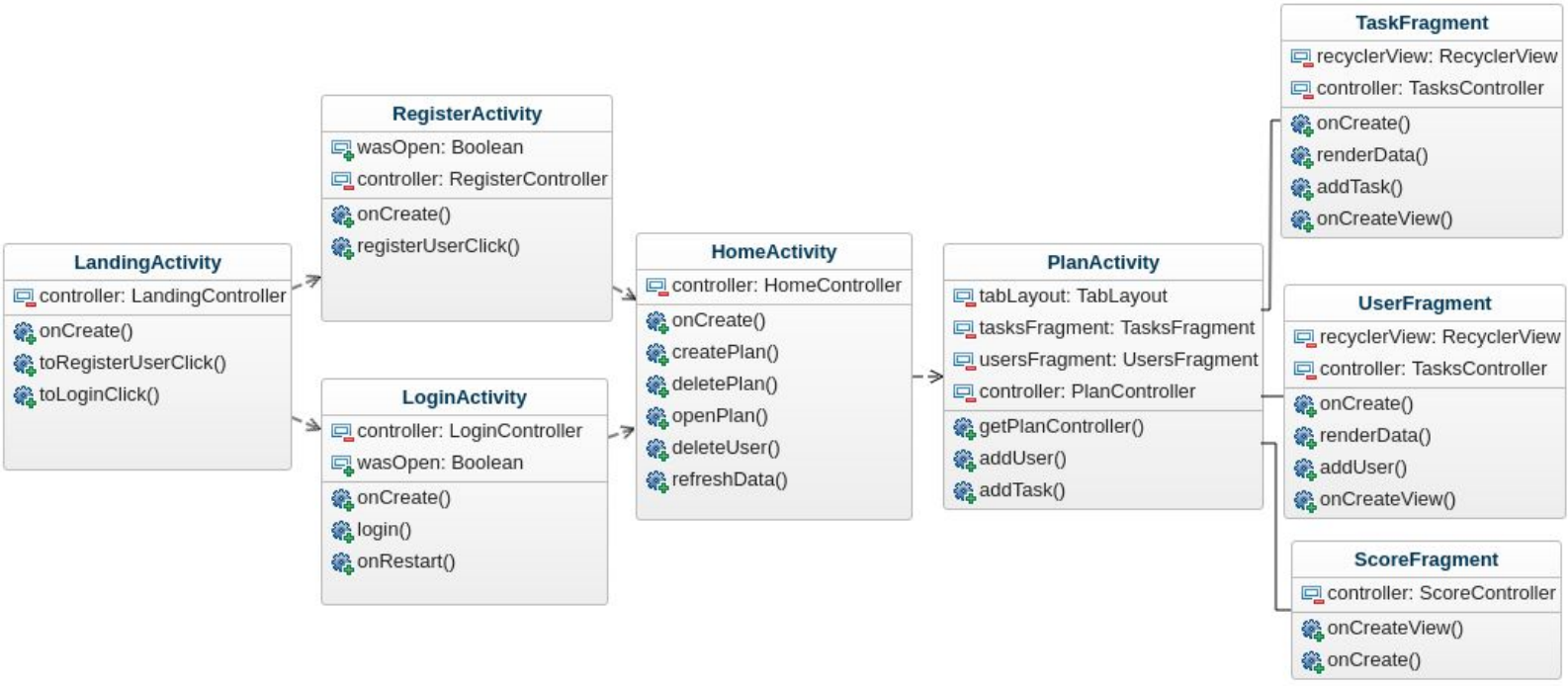
3.1 *GUI / View*

3.1.1 Zusammenfassung

Die GUI Komponente beinhaltet alle Dateien, welche zur Erstellung der Android Benutzeroberfläche benötigt werden. Hierbei handelt es sich entweder um **Activities** oder um **Fragments**

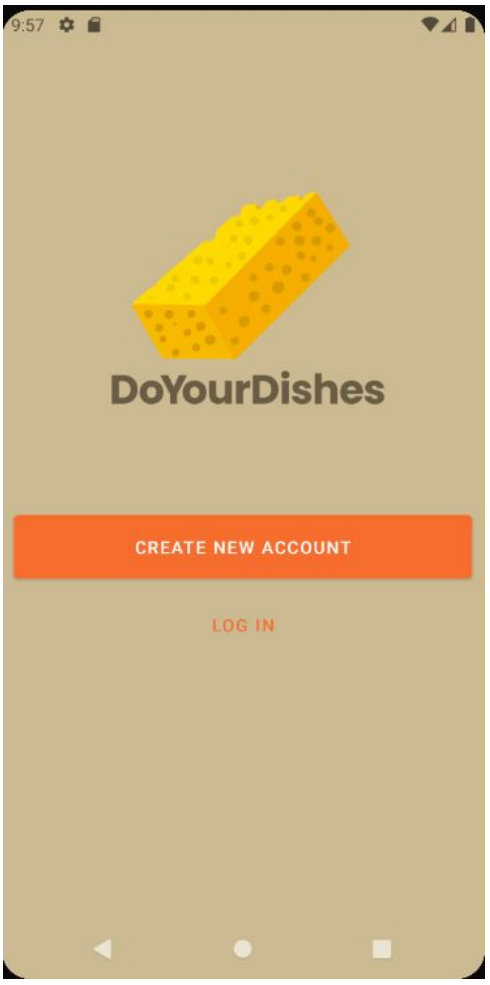
Ihre Hauptaufgabe besteht darin eine Schnittstelle zwischen den **XML Layoutdateien** und der *ControllerLogic* Komponente zu bilden. In dieser Komponente wird **keine Logik** implementiert. Es werden lediglich Variablen erzeugt, welche an die kontrollierende Komponente übergeben werden um es dieser zu ermöglichen auf Eingaben wie Text oder das Drücken eines Knopfes des Benutzers zu reagieren. Die GUI steht in enger Abhängigkeit mit Android, da die Activities die Klasse “AppCompatActivity” erweitern und die Fragments das selbe mit der Android “Fragment” Klasse tun.

3.1.2 Übersichtsdiagramm, Klassen mit Userflow



Die Abfolge der Bereiche unseres Android Klienten ist linear. Startet ein User die Anwendung, so landet er als erstes in der ***LandingActivity***, anschließend kann er weiter zur ***RegisterActivity*** oder zur ***LoginActivity*** übergehen. Beide Wege enden in der ***HomeActivity***, in welcher der User den Zustand “eingeloggt” hat. Die letzte Activity ist die ***PlanActivity***, diese Besteht aus einem an der Oberseite des Bildschirms befestigten Leiste, welche eine Navigation zwischen 3 Fragments erlaubt. Diese Fragments gehören zur ***PlanActivity*** und bestehen aus ***Tasks***, ***Users*** und ***Score*** Möchte der User in der Anwendung zurückgehen, kann er das über den physikalischen “Back Button” tun, mit der er die jeweils aktuelle Activity beendet.

3.1.3 Innere Struktur und Details



3.1.3.1 LandingActivity

Die **LandingActivity** besitzt zwei Buttons, welche den User jeweils in die nächste Activity weiterleiten. Die Verknüpfung der Methoden und Buttons ist in der zur Activity gehörenden Layoutdatei wie folgt definiert:

```
android:onClick="toLoginClick"
```

Hierdurch kann in der Activity die Methode wie folgt deklariert werden:

```
public void toLoginClick(View view){
    landingController.goToLogin();
}
```

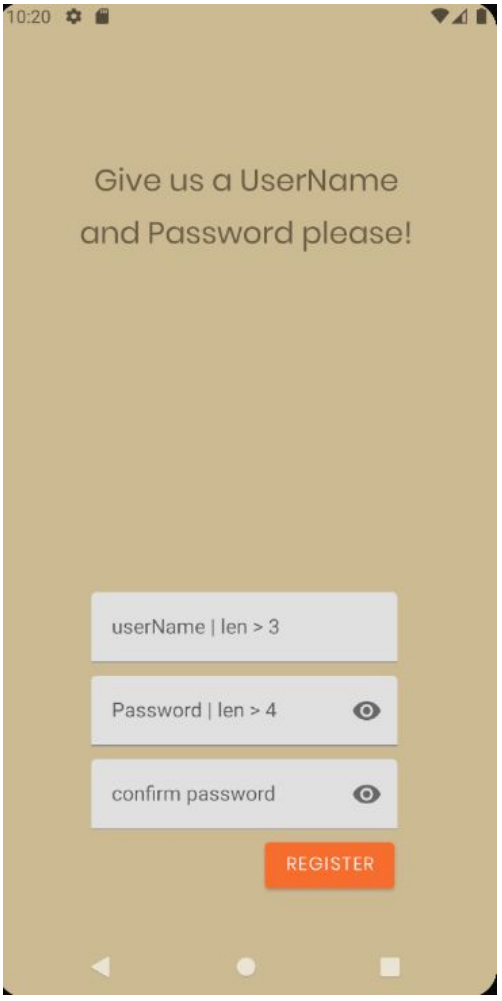
Auf diesem Weg kommt die Activity ohne einen “onClickListener” aus, was sehr gut zum MVC Entwurfsmuster passt, denn die eigentliche Methode wird in der **LandingController** Klasse implementiert, welche zur Komponente **ControllerLogic** gehört.

3.1.3.2 LoginActivity

Die **LoginActivity** verfügt über zwei EditText Felder und einen Button. Die Verlinkung erfolgt analog. Das Umgehen mit Benutzereingaben erfolgt nicht in dieser Klasse, sondern im **LoginController**. Die Instanziierung und Übergabe der Eingabefelder passiert wie folgt:

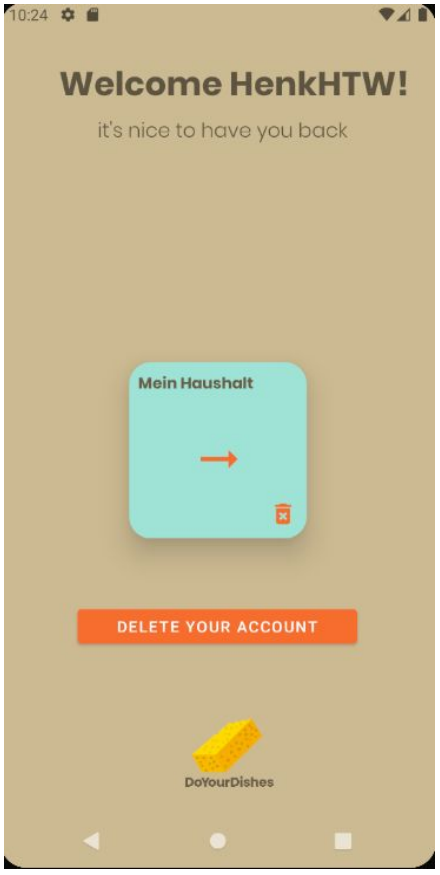
Bei erfolgreicher Übermittlung von Daten leitet, startet die **LoginActivity** die **HomeActivity** und übergibt dieser in den extra Feldern des Intents Informationen über den angemeldeten User

```
this.loginController = new LoginController(
        findViewById(R.id.toLoginButton),
        (EditText) findViewById(R.id.userNameEditTextView),
        (EditText) findViewById(R.id.passwordEditTextView),
         _mainActivity: this);
```

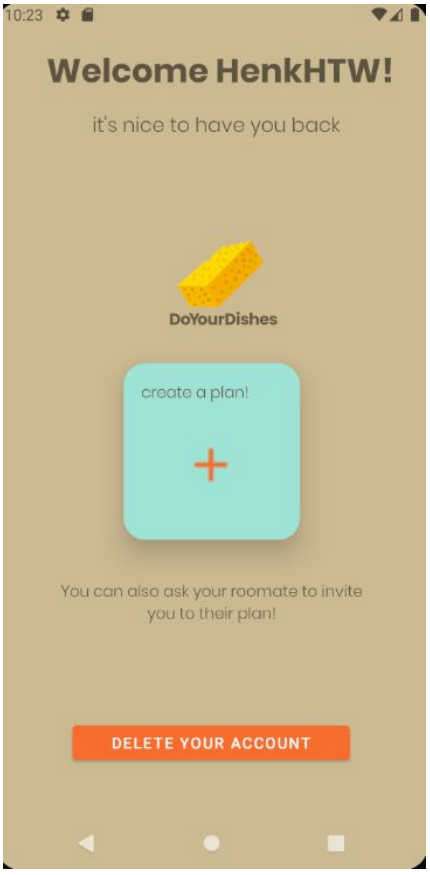


3.1.3.3 RegisterActivity

Die **RegisterActivity** ähnelt stark der **LoginActivity**. Sie besitzt lediglich ein weiteres EditText Feld zur Bestätigung des Passworts. Verlinkung und Methoden Implementation analog.



PlanActivity in Plan



PlanActivity nicht in Plan

3.1.3.4 HomeActivity

Die **HomeActivity** ist der Mittelpunkt der Anwendung. Verlinkung und Methoden Implementation analog. Das besondere an dieser Activity ist, dass sie Informationen im Intent übergeben bekommt, damit sie direkt im onCreate Zeitpunkt weiß, ob der angemeldete User Mitglied eines Plans ist.

```
Intent intent = getIntent();
homeController = new HomeController(
        intent.getStringExtra( name: "TOKEN"),
        intent.getStringExtra( name: "PLANNAME"),
        intent.getStringExtra( name: "USERNAME"),
        intent.getStringExtra( name: "USERPLANID"),
        intent.getStringExtra( name: "PLANOWNER"),
         _homeActivity: this);
```

Außerdem besitzt sie zwei **Layoutdateien**, eine wird verwendet falls der User Mitglied eines Plans ist, die Andere falls dies nicht der Fall ist.

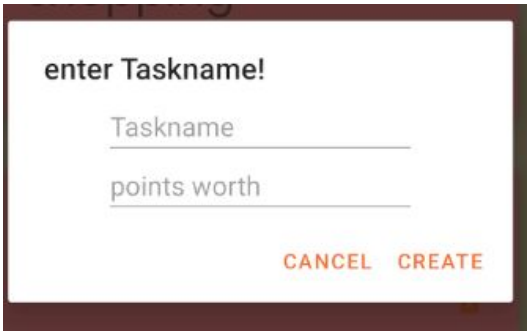
In ihrer onCreate Methode wird die **Login** oder **RegisterActivity** beendet.

Tasks Fragment

3.1.3.5 PlanActivity

Die **PlanActivity** besteht aus einem **TabLayout** welches sich an der Oberseite des Bildschirms befindet. Es gibt drei Inhalte die Angezeigt werden. Der erste besteht aus den Tasks eines Plans. Diese werden als dynamische Menge angezeigt ermöglicht durch eine **RecyclerView**, welche einen **RecyclerViewAdapter** und einen **ViewHolder** verwendet um die einzelnen **Tasks** als Items anzuzeigen.

Die Einzelnen Tasks verändern die Farbe je nachdem, wie lange es her ist, dass sie das letzte mal erledigt worden sind. Außerdem wird angezeigt wieviele Punkte eine **Task** bringt und es gibt die möglichkeit die **Task** zu erledigen oder zu löschen. Ganz unten ist noch ein Button, welcher dazu dient eine neue Task zu erstellen. Dieser öffnet einen Dialog der eine individuelle **View** beinhaltet, mit zwei EditText feldern, sowie Buttons zum bestätigen oder abbrechen.



Die beschriebene Funktionalität ist wieder in zugehörigen Controller Komponenten implementiert.

Users Fragment

Der zweite Reiter beinhaltet das **UserFragment**. Der Aufbau ist analog zum **TaskFragment** mit eigener **RecyclerView**, **Adapter** und **ViewHolder**. Die einzelnen User werden durch **CardViews** abgebildet und geben den Benutzernamen sowie die aktuelle Punktzahl an.

Es gibt außerdem einen Button welcher den jeweiligen **User** löscht, sowie unten einen Button um einen **User** hinzuzufügen.

Somit besteht die **PlanActivity** nicht nur aus der einen Activity Klasse sondern auch aus den unteren Fragment Klassen **TasksFragment** und **UsersFragment**.

Das **ScoreFragment** ist nicht fertig geworden.

3.1.4 Schnittstellen

Die **Activites** und **Fragments** der **GUI** implementieren ihre jeweiligen Programmierschnittstellen. Diese Schnittstellen zeigen an welche Operationen die Activity dem User oder anderen Klassen anbietet.

```
public interface LandingActivityInterface {
    void toRegisterUserClick(View view);

    void toLoginClick(View view);
}
```

```
public interface LoginActivityInterface {
    void login(View view);
}
```

```
public interface RegisterActivityInterface {
    void registerUserClick(View view);
}
```

```
public interface UsersFragmentInterface {
    void renderData(List<User> usersToRender);
    void addUser();
}
```

```
public interface HomeActivityInterface {
    void createPlan(View view);
    void deletePlan(View view);
    void openPlanActivity(View view);
    void deleteUser(View view);
}
```

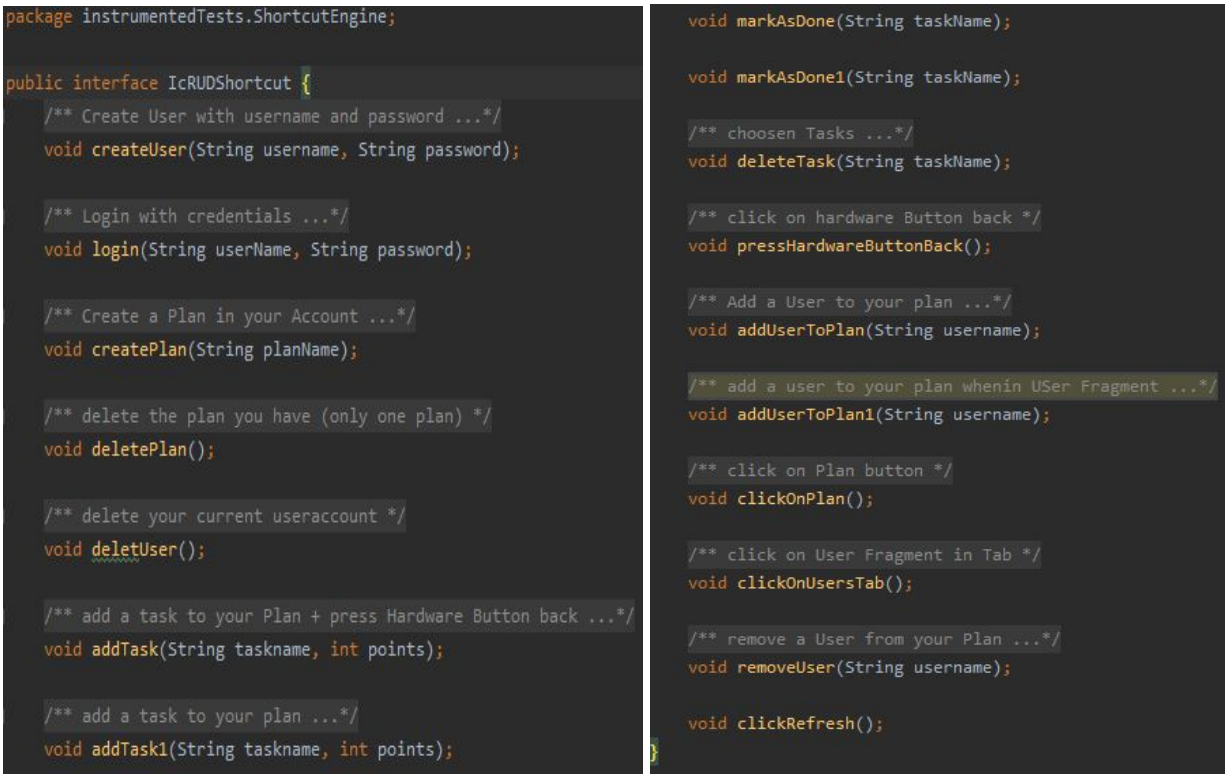
```
public interface TasksFragmentInterface {
    void renderData(List<Task> tasksToRender);
    void addTask();
}
```

3.1.5 Testkonzept

3.1.5.1 ShortcutEngine

Die Instrumented Tests haben zur Vereinfachung des Testablaufs 3 Klassen. Diese sind im Package **ShortcutEngine** zu finden.

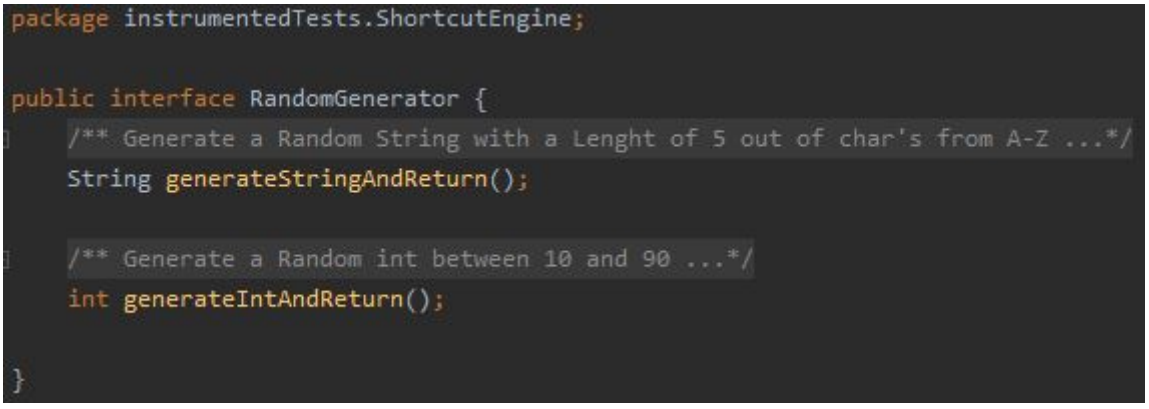
1. Das **ICrudShortcut** Interface bietet dazu diverse Funktionalitäten an.



```
package instrumentedTests.ShortcutEngine;

public interface IcRUDShortcut {
    /** Create User with username and password ...*/
    void createUser(String username, String password);

    /** Login with credentials ...*/
    void login(String userName, String password);

    /** Create a Plan in your Account ...*/
    void createPlan(String planName);

    /** delete the plan you have (only one plan) */
    void deletePlan();

    /** delete your current useraccount */
    void deletUser();

    /** add a task to your Plan + press Hardware Button back ...*/
    void addTask(String taskname, int points);

    /** add a task to your plan ...*/
    void addTask1(String taskname, int points);

    void markAsDone(String taskName);

    void markAsDone1(String taskName);

    /** choosen Tasks ...*/
    void deleteTask(String taskName);

    /** click on hardware Button back */
    void pressHardwareButtonBack();

    /** Add a User to your plan ...*/
    void addUserToPlan(String username);

    /** add a user to your plan whenin USer Fragment ...*/
    void addUserToPlan1(String username);

    /** click on Plan button */
    void clickOnPlan();

    /** click on User Fragment in Tab */
    void clickOnUsersTab();

    /** remove a User from your Plan ...*/
    void removeUser(String username);

    void clickRefresh();
}
```

Die **CrudEngine** realisiert das **ICrudShortcut**. Die **CrudEngine** prüft zudem jeweils am Ende des Aufrufs auf check/match der Views.

2. Das **RandomGenerator** Interface bietet die Möglichkeit einen zufällig generierten String für den Test mit dem Usernamen/Passwort/Plan Name/Tasknamen zu erzeugen und einen zufällig generierten Integerwert für die Taskpunkte.



```
package instrumentedTests.ShortcutEngine;

public interface RandomGenerator {
    /** Generate a Random String with a Lenght of 5 out of char's from A-Z ...*/
    String generateStringAndReturn();

    /** Generate a Random int between 10 and 90 ...*/
    int generateIntAndReturn();

}
```

Die Klasse **RandomStringGeneratorForLogin** realisiert das Interface **RandomGenerator**.

3. Die Klasse ToastMatcher erbt den TypeSafeMatcher und prüft den Toast String im aktuelleln View.

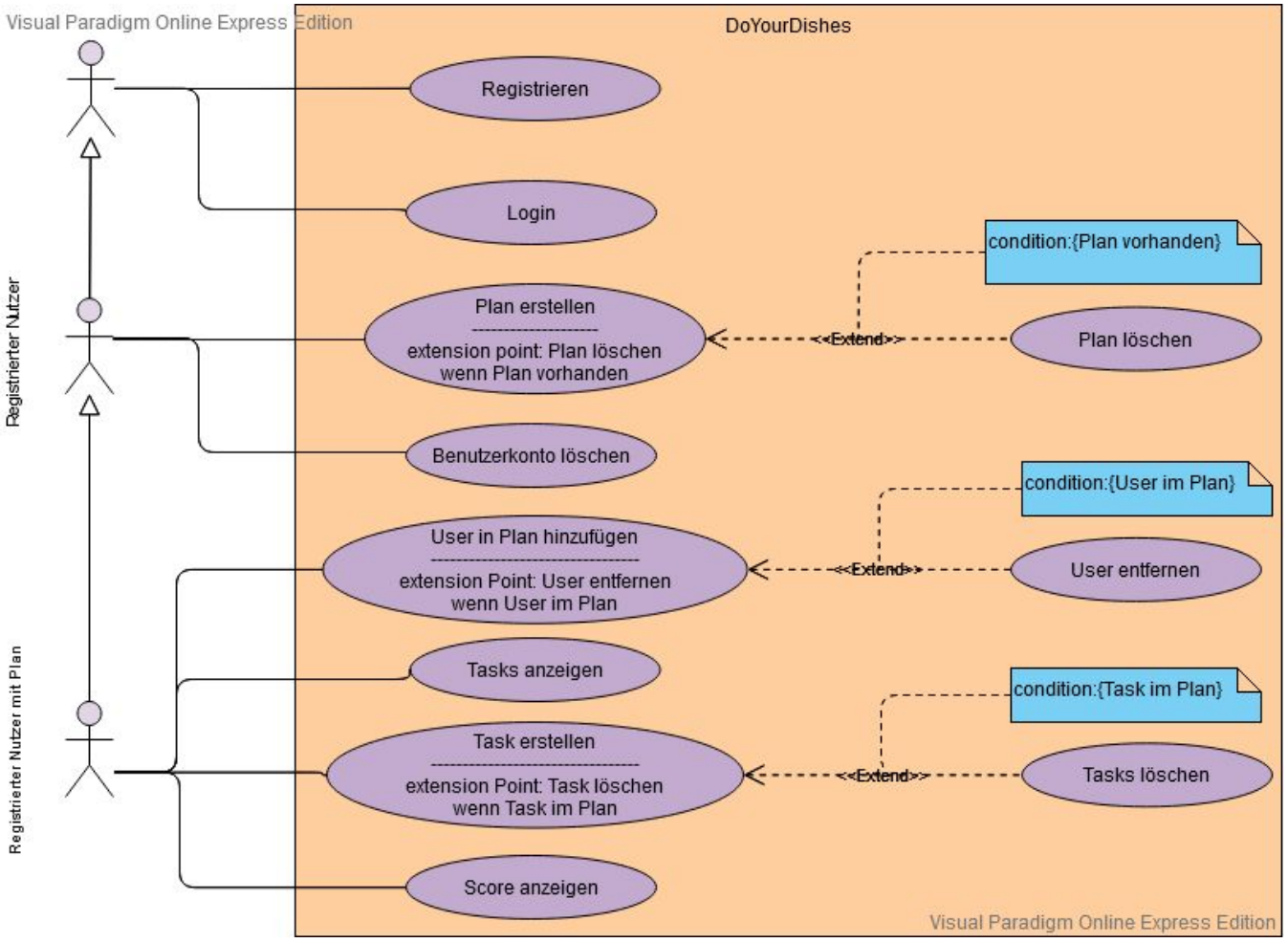
3.1.5.2 E2E-Test

Diese Komponente wird mit dem UI Testframework Espresso getestet.

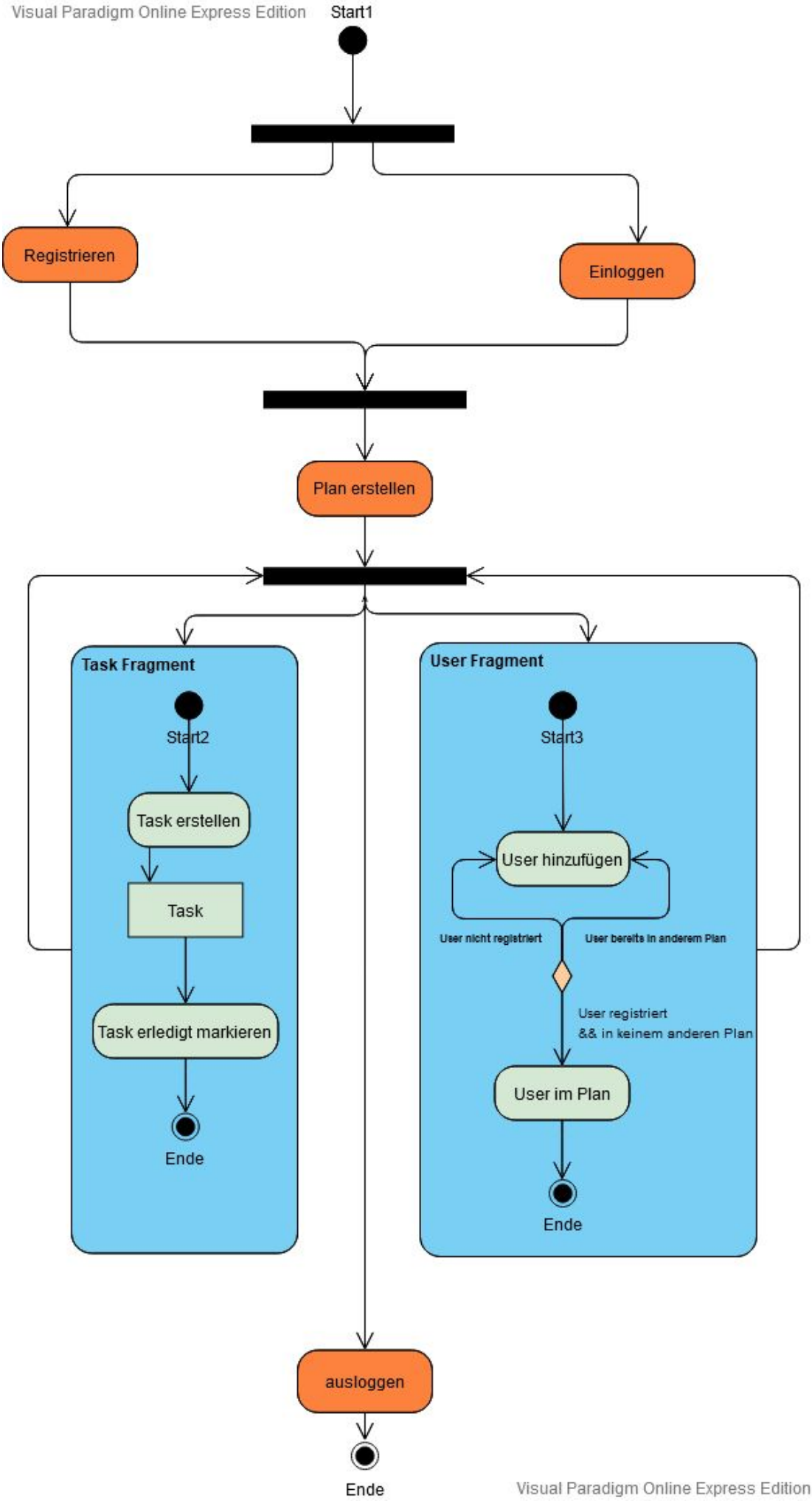
Zur Erläuterung des Testkonzepts wird der End-to-End (E2E) - Test der Anwendung genutzt.

E2E-Schritt	Test Name	Testmethode	Erfolg
Register User1	createUserGutTest1()	Espresso	Ja
Register User2	createUserGutTest1()	Espresso	Ja
User2 Create a Plan	createPlanGutTest2()	Espresso	Ja
User2 add's User1 to Plan	addAnotherUserToYourPlanGutTest1()	Espresso	Ja
Login User1	loginUserGutTest1()	Espresso	Ja
User1 add Task	createTaskGutTest1()	Espresso	Ja
Login User2	loginUserGutTest1()	Espresso	Ja
mark TASK as DONE	markAsDoneTaskGutTest1()	Espresso	Ja
Remove USER1 off of Plan	deleteUserOffOfPlan()	Espresso	Ja
delete Plan	deletePlanGutTest1()	Espresso	Ja
Delete User2	deleteUserGutTest1()	Espresso	nein
Start App + Login User1	logInExistingUser()	Espresso	ja

3.1.5.3 Anwendungsfalldiagramm E2E



3.1.5.2 Aktivitätsdiagramm E2E



3.2 *dataModel*

3.2.1 Zusammenfassung

Die dataModel Komponente besitzt drei Klassen: Plan, Task und Model. Sie definieren in unserem Android Klienten welche Attribute die Datentypen besitzen, welche in der Datenbank gespeichert werden sollen. Außer Gettern und Settern besitzen die Klassen über keine Funktionalität außer ihrem Konstruktor.

3.1.2 Übersicht

3.1.3 Innere struktur

Das dataModel besitzt über keine komplexe nennenswerte innere Struktur. Es wird von der controllerLogic sowie der asyncLogic Komponente verwendet.

3.1.4 Schnittstellen

Schnittstellen sind der Konstruktor und die getter,setter Methoden.

3.1.5 Testkonzept

Das DataModel wurde zu beginn des Projektes Testgetrieben entwickelt. Die Tests wurden mit Junit durchgeführt.

Nicht alle Tests wurden aufgeführt, da einige Redundant durch andere Tests.

Package. **Class**	**Was wird getestet**	**Test Name**	**Testmethode**	**Erfolg**
PlanTests	PlanOwner festlegen	gutTestsetNameOfPlan1()	JUnit	ja
	Name leer („“)	schlechtTestsetName() expected Exception: IllegalArgumentException.class	Junit	Ja
	Plan Name wird nicht angegeben	fehlerFallTestsetName() expected Exception: NullPointerException.class	Junit	Ja
	Mehrere Tasks werden erstellt	gutTestsetTask()	Junit	Ja
	Leere Taskliste aufrufen	schlechTestsetTask2()	Junit	Ja
	Füge drei Nutzer dem Plan hinzu	gutTestsetUsers2()	Junit	Ja
	Name leer („“)	fehlerFallTestgetName() expected Exception: IllegalArgumentException.class	Junit	Ja
	Erstelle drei User, erstelle zwei Tasks in gemeinsamen Plan	gutTestBigTest()	Junit	Ja

TaskTests	kein Taskname, kein zugehöriger Plan und keine Punkte	fehlerFallLeererKostruktor() expected Exception: NullPointerException.class	Junit	Ja
	Ein Task wird 0 erreichbaren Punkte erstellt	fehlerFallNullPunkte() expected Exception: IllegalArgumentException.class	Junit	Ja
	Ein Task wird mit einer negativen Zahl an erreichbaren Punkten erstellt	fehlerFallNegativePunktzahl() expected Exception: IllegalArgumentException	Junit	Ja
UserTests	Registrierter Nutzer legt Plan Name fest	gutTestSetPlan()	Junit	Ja
	Nicht Registrierter Nutzer möchte Plan Name ändern	schlechtTestSetPlanWennUserOhneParameter() expected Exception: NullPointerException.class	Junit	Ja
	Registrierter Nutzer ändert Nutzer Namen	gutTestsetUserName()	Junit	Ja
	Nicht Registrierter Nutzer möchte Namen festlegen	schlechtTestsetUserNameWennKeinUserVorhanden() expected Exception: NullPointerException.class	Junit	Ja
	Nicht Registrierter Nutzer fragt Punkte ab	schlechtTestgetPointsInPlanWennKeinPlanVorhanden() expected Exception: NullPointerException.class	Junit	Ja

	Abfrage nach Punktzahl in Plan ohne Punkte	grenzTestgetPointsInPlan() expected Exception: IllegalArgumentException.class	Skipped: @Ignore	Ja
	Negative Punktzahl für Registrierten Nutzer	schlechtTestsetPointsNegativePoints() expected Exception: IllegalArgumentException.class	Junit	Ja
	Setze Negative erreichte Punktezahl für Nutzer (nicht registriert)	fehlerfallSetPointsInPlanNegativMitUserOhneParameter() expected Exception: NullPointerException.class	Junit	Ja
	Erstelle Nutzer mit einem Plan und 500 erreichten Punkten im Plan	gutTestBigTest()	Junit	ja

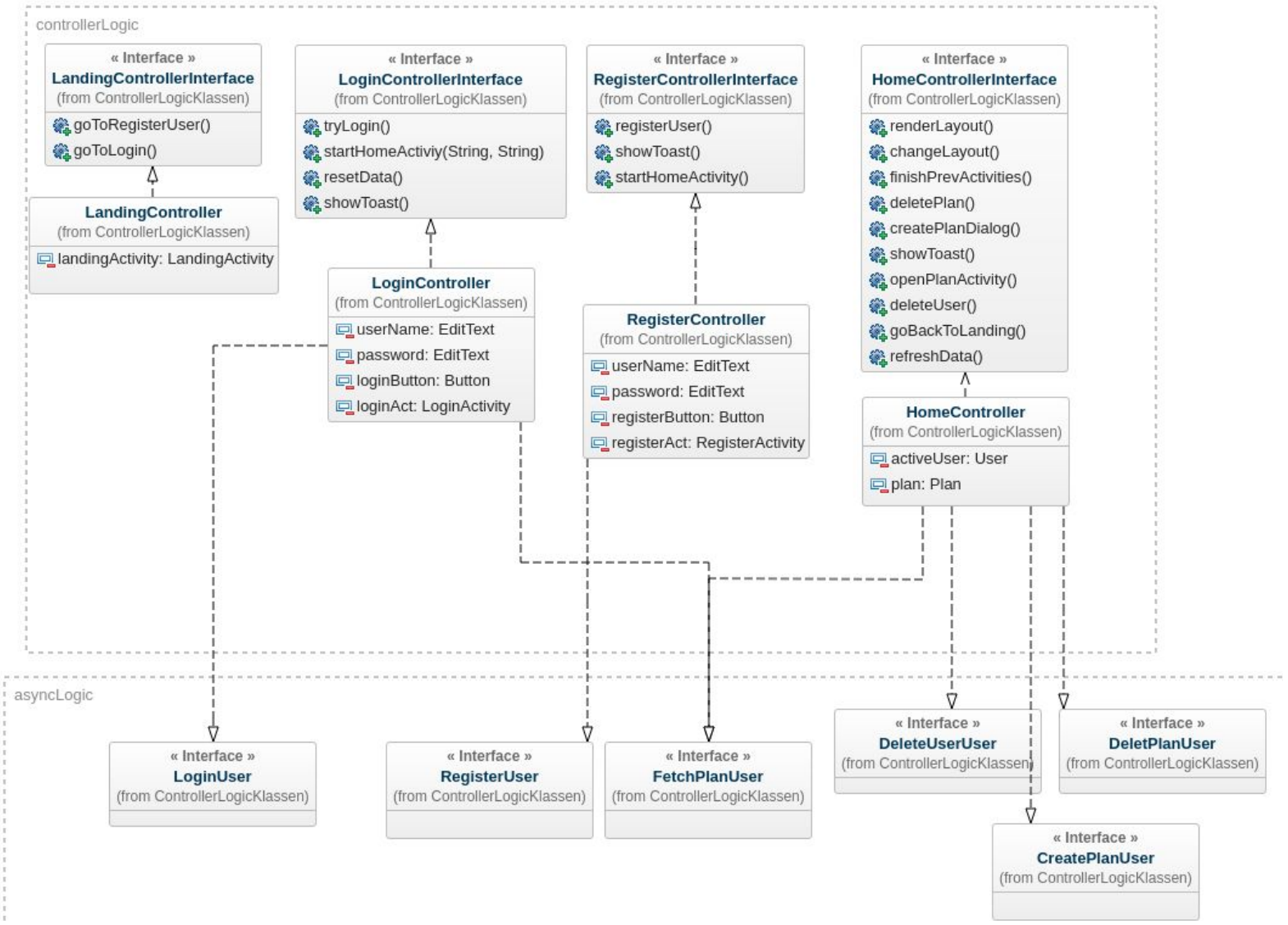
3.3 *controllerLogic*

3.3.1 Zusammenfassung

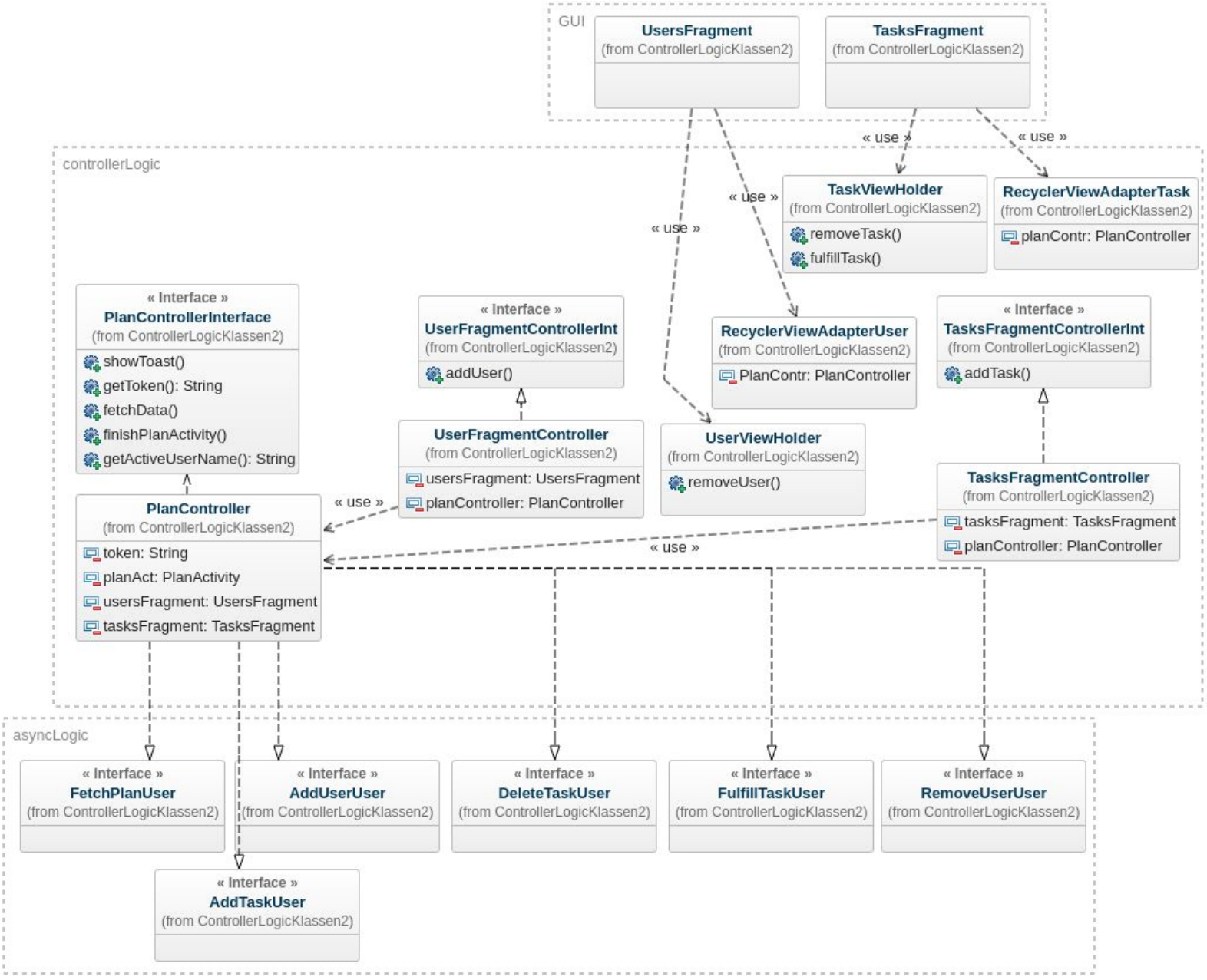
Die **controllerLogic** Komponente beinhaltet alle Dateien die für die ausgeführte Logik der **GUI** Komponente zuständig sind. In unserem Android Klienten besitzt jede **Activity** und jedes **Fragment** eine eigene **Controller Klasse** und somit **Instanz**, welche sämtliche Funktionalität beinhaltet. Wie zum Beispiel: **onClick** Methoden, **User Eingaben** verarbeiten, **Daten** aus dem **Backend Klienten** laden, Veränderung der **Layouts**, **Anzeigen der geladenen Daten**, etc.

3.3.2 Innere Struktur

3.3.2.1 1. Klassendiagramm



3.3.2.1 2. Klassendiagramm



Die einzelnen Controller sind voneinander unabhängig, nur die **User** und **TasksFragmentController** verwenden eine Instanz des **PlanControllers**, welcher die asynchronen Prozesse aufruft, die in den Fragments ausgeführt werden sollen. Die **RecyclerViewAdapter** arbeiten eng mit der **GUI**, sie legen Fest welches Layout die einzelnen Elemente der **RecyclerView** in den **Fragments** bekommen. die **ViewHolder** Klassen bekommen einzelne **User** oder **Tasks** und benutzen die Getter Methoden des **Models** um bspw. den **UserName** oder den **TaskName** in der **RecyclerView** richtig anzuzeigen. Außerdem verwalten sie die kleinen Buttons die zur **User** bzw **Task Cardview** gehören (delete User, delete Task, Fulfill Task) denn die müssen ja beim **onClick** die **TaskId** bzw. den **UserName** kennen, um die richtigen Daten an die **asyncLogic** Komponente zu übertragen. Auch die Logik welche dafür sorgt, dass abhängig vom Zeitstempel einer Task, die Task eine andere Farbe bekommt ist im TaskViewHolder implementiert.

3.3.4 Schnittstellen

Die Schnittstellen werden gebildet durch die Konstruktoren der Controller Klassen und deren Interfaces, welche den Klassendiagrammen entnommen werden können.

3.3.4 Testkonzept

Die Grundidee Testkonzepts der ControllerLogic wurde mit Espresso umgesetzt. Im folgenden sind die einzelnen Test aller Funktionalitäten der App aufgeführt.

Was wird getestet	**Test Name**	**Testmethode**	**Erfolg**
Registriere Nutzer	createUserGutTest1()	Espresso	ja
Login Nutzer	loginUserGutTest1()	Espresso	
Lösche Nutzer	deleteUserGutTest1()	Espresso	Nein
Lösche anderen Nutzer aus dem Plan	deleteUserOffOfPlan()	Espresso	Ja
Erstelle einen Plan	createPlanGutTest2()	Espresso	ja
Füge anderen Nutzer dem Plan hinzu	addAnotherUserToYourPlanGutTest1()	Espresso	Ja
Erstelle Task in Plan	createTaskGutTest1()	Espresso	Ja
Markiere Task als erledigt	markAsDoneTaskGutTest1()	Espresso	Ja
Nutzername zu kurz	creatUserSchlechtTest1()	Espresso	Ja
Passwort zu kurz	creatUserSchlechtTest2()	Espresso	Ja
Passwort stimmt nicht überein	creatUserSchlechtTest3()	Espresso	Ja
Falsche Logindaten	LoginSchlechtTest1()	Espresso	Ja
Richtige Logindaten	LoginGutTest()	Espresso	Ja
Lösche Task nachdem er erledigt wurde	deleteTaskGutTest3()	Espresso	Ja

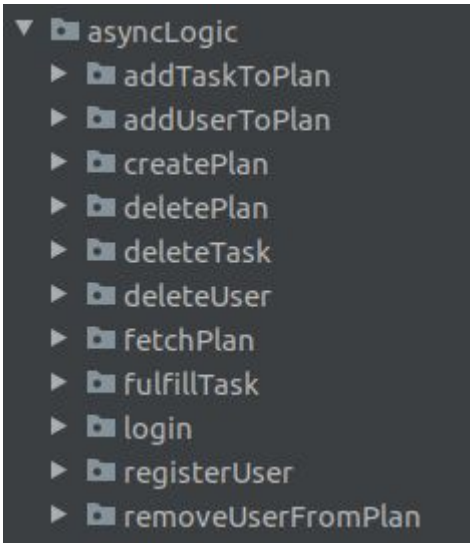
3.4 *asyncLogic*

3.4.1 Zusammenfassung

Die asyncLogic Komponente implementiert sämtliche asynchrone Anfragen welche an den Backend Dienst gesendet werden. Sie wird von der controllerLogic Komponente aufgerufen, baut einen HTTP body und verwendet dann die NetworkHTTP Komponente um diesen zu senden. Die HTTP Antwort welche vom Server zurück kommt, wird in der asyncLogic entgegen genommen und in Form gebracht, in dem beispielsweise Daten über User aus dem JSON format in unseren eigenen Datentyp “User” umgewandelt werden.

3.4.2 Übersicht

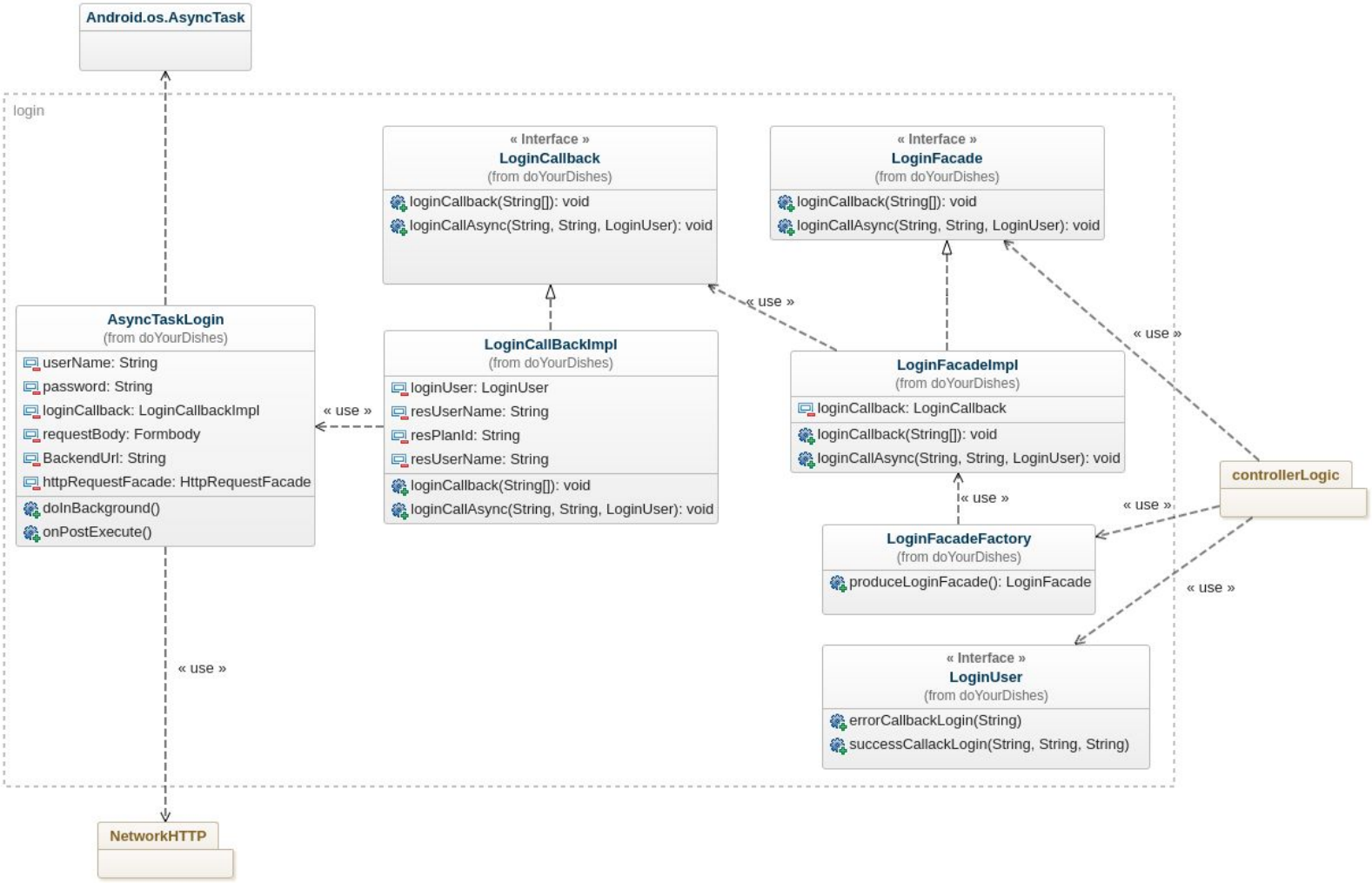
Die Komponente besteht aus mehreren Unterkomponenten die alle dem Fassaden Entwurfsmuster folgen.



Eine Übersicht der **Package Struktur** in der Android App Komponente **asyncLogic**, Jede unter Komponente **ähnelt im Aufbau stark** der folgenden Login Komponente

3.4.3 Innere Struktur

3.4.3.1 Klassendiagramm



Dieses Diagramm zeigt den inneren Aufbau der Login Komponente innerhalb der **asyncLogic** komponente. Sie stellt eine **LoginFacadeFactory** zur verfügung, über welche **LoginFacades** erstellt werden können. Der Zugriff auf die Implementation erfolgt ausschließlich über diese **LoginFacade**. Selbige greift auf das **LoginCallback** Interface zu.

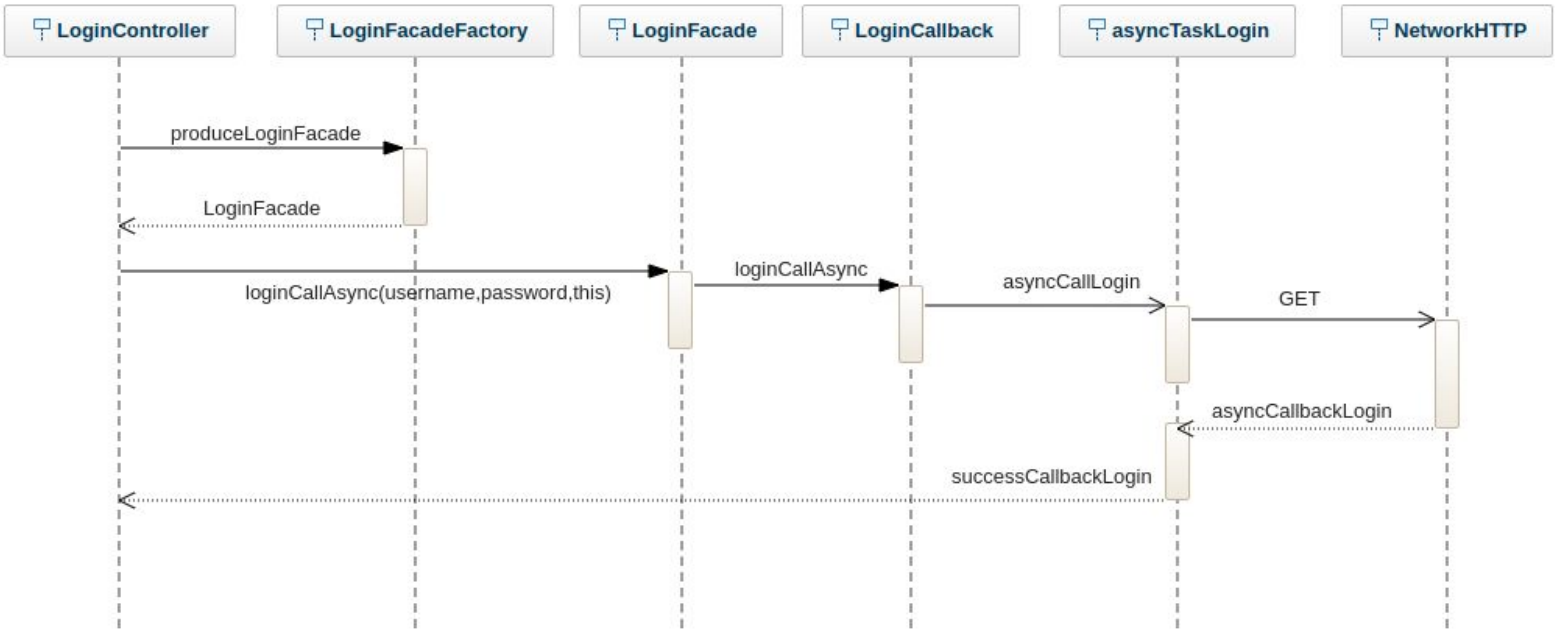
Der Kern der Komponente ist die **AsyncTaskLogin** Klasse, sie erweitert die Android Klasse AsyncTask und führt die Methode **doInBackground** in einem Hintergrundprozess aus, welcher abgelöst ist vom UI Prozess der Android Anwendung. In diesem Hintergrundprozess wird der HTTP Request an den Backend Dienst gesendet und sobald die HTTP Antwort zurückgekehrt ist, wird die **onPostExecute** Methode aufgerufen, welche wiederum die **loginCallback** Methode in der Klasse **LoginCallbackImpl** aufruft. In dieser Klasse wird die Antwort, welche im JSON Format vom Backend Dienst geliefert wird über die **JSONObject library** in Strings oder Listen mit Usern bzw. tasks umgewandelt.

```
JSONObject response = httpRequestFacade.POST( path: BackendURL + "/auth/login", requestBody,  token: "");
Log.d(TAG,  msg: "doInBackground response: " + response);
if (response.has( name: "token")) {
    responseArr[0] = "loginSuccess";
    responseArr[1] = response.getString( name: "token");
```

Ein weiteres Beispiel aus der **fetchPlan** Komponente welches mit **JSONArray** sowie **JSONObject** arbeitet:

```
try {
    response = httpRequestFacade.GET( path: BackendURL + "/plan/findPlanToOwner", requestBody, _token);
    if(response.has( name: "data")){
        Log.d(TAG,  msg: "doInBackground: " + response);
        responseArr[0] = "fetchPlanSuccess";
        responseArr[1] = response.getJSONObject("data").getString( name: "owner");
        responseArr[2] = response.getJSONObject("data").getString( name: "name");
        String planId = response.getJSONObject("data").getString( name: "_id");
        responseArr[3] = planId;
        JSONArray userArr = response.getJSONObject("data").getJSONArray( name: "users");
        users = new ArrayList<User>();
        for(int i = 0; i < userArr.length(); i++){
            JSONObject tempUser = userArr.getJSONObject(i);
            User newUser = new User(tempUser.getString( name: "userName"), planId, tempUser.getInt( name: "points"));
            users.add(newUser);
            Log.d(TAG,  msg: "doInBackground: " + newUser);
        }
        JSONArray taskArr = response.getJSONObject("data").getJSONArray( name: "tasks");
        tasks = new ArrayList<Task>();
        for(int i = 0; i < taskArr.length(); i++){
            JSONObject tempTask = taskArr.getJSONObject(i);
            BigInteger bigTimeStamp = new BigInteger( tempTask.getString( name: "lastTimeDone") );
            Task newTask = new Task(
                    tempTask.getString( name: "taskName"),
                    planId,
                    tempTask.getInt( name: "pointsWorth"),
                    bigTimeStamp,
                    tempTask.getString( name: "taskId"));
            tasks.add(newTask);
            Log.d(TAG,  msg: "doInBackground: " + newTask);
        }
```

3.4.3.2 Sequenzdiagramm



Dieses **Sequenzdiagramm** soll noch Mal veranschaulichen wie der Ablauf einer Asynchronen Anfrage an den **Backend Dienst** aussieht.

Die größte Schwierigkeit in dieser Komponente war es, sich zu überlegen wie man die Komponenten unabhängig von den Controllern implementiert, die sie einsetzen. Da wir eine asynchrone Antwort vom Backend Dienst bekommen, muss auch gewährleistet sein, dass die **asyncLogin Komponente** einen Wiedereinstiegspunkt besitzt, an dem die Daten der Backend Antwort richtig verarbeitet werden.

Für diesen Zweck implementieren die einzelnen Controller spezielle Interfaces. In diesem Fall das **LoginUser Interface**, welches auch in der obigen Grafik der asyncTask Login zu sehen ist.
Diese Interfaces bieten jeweils zwei Methoden an:

1.

```
void successCallbackLogin (String _token, String _resUserName, String _resUserPlanId);
```

Diese Methode wird in der Controller klasse die das **LoginUser Interface** implementiert dann aufgerufen, wenn die Login Antwort vom server as “**Success**” ausgewertet wurde. Sie übergibt dem Controller in diesem Beispiel den **Login Token** sowie Daten über den eingeloggten User.

2.

```
void errorCallbackLogIn(String errorInfo);
```

Diese Methode wird aufgerufen im Falle einer Fehlermeldung aus dem Backend Dienst. Beispielsweise falls der Benutzer falsche oder keine Daten eingegeben hat beim Loginversuch.

Die anderen asynchronen Funktionalitäten sind analog zum gezeigten Beispiel der Login Task aufgebaut, befolgen alle das Fassaden Entwurfsmuster und haben ihre eigenen Interfaces welche durch die Controller implementiert werden um den Response Callback zu ermöglichen.

3.4.4 Schnittstellen

Die Schnittstellen sollten schon deutlich geworden sein.

```
/**
 * this interface works with the asynchronous logIn task
 *  every activity can implement its own version of the interface to handle what is done with the response Values
 */
public interface LoginFacade {
    /**
     *  is called when the asyncTask login should be started/executed
     */
    void loginCallAsync(String userName, String password, LoginUser loginUser);
}
```

```
/**
 * Classes that want to use the AsyncTask LoginUser have to implement this interface,
 * so the asyncTask knows where to insert the response via callBacks
 */
public interface LoginUser {

    /**
     * gets called by AsyncTask when an error was responded
     * @param errorInfo holds info about the responseError
     */
    void errorCallbackLogIn(String errorInfo);

    /**
     * gets called by AsyncTask when a success was responded
     * @param _token holds the Token that got responded by http request
     * @param _resUserName  holds the Username that got responded by http request
     * @param _resUserPlanId holds the userPlanId that got responded by http request
     */
    void successCallbackLogin (String _token, String _resUserName, String _resUserPlanId);
}
```

```
public class LoginFacadeFactory {
    public static LoginFacade produceLoginFacade(){
        LoginCallBack loginCallBack = new LoginCallBackImpl();
        return new LoginFacadeImpl(loginCallBack);
    }
}
```

Wieder am Beispiel der Login Funktion, die anderen asynchronen Komponenten verhalten sich analog.

3.4.5 Testkonzept

Die AsyncLogic-Komponente wurde in den anderen Tests mitgetestet und wurde dahingehend nicht im einzelnen getestet

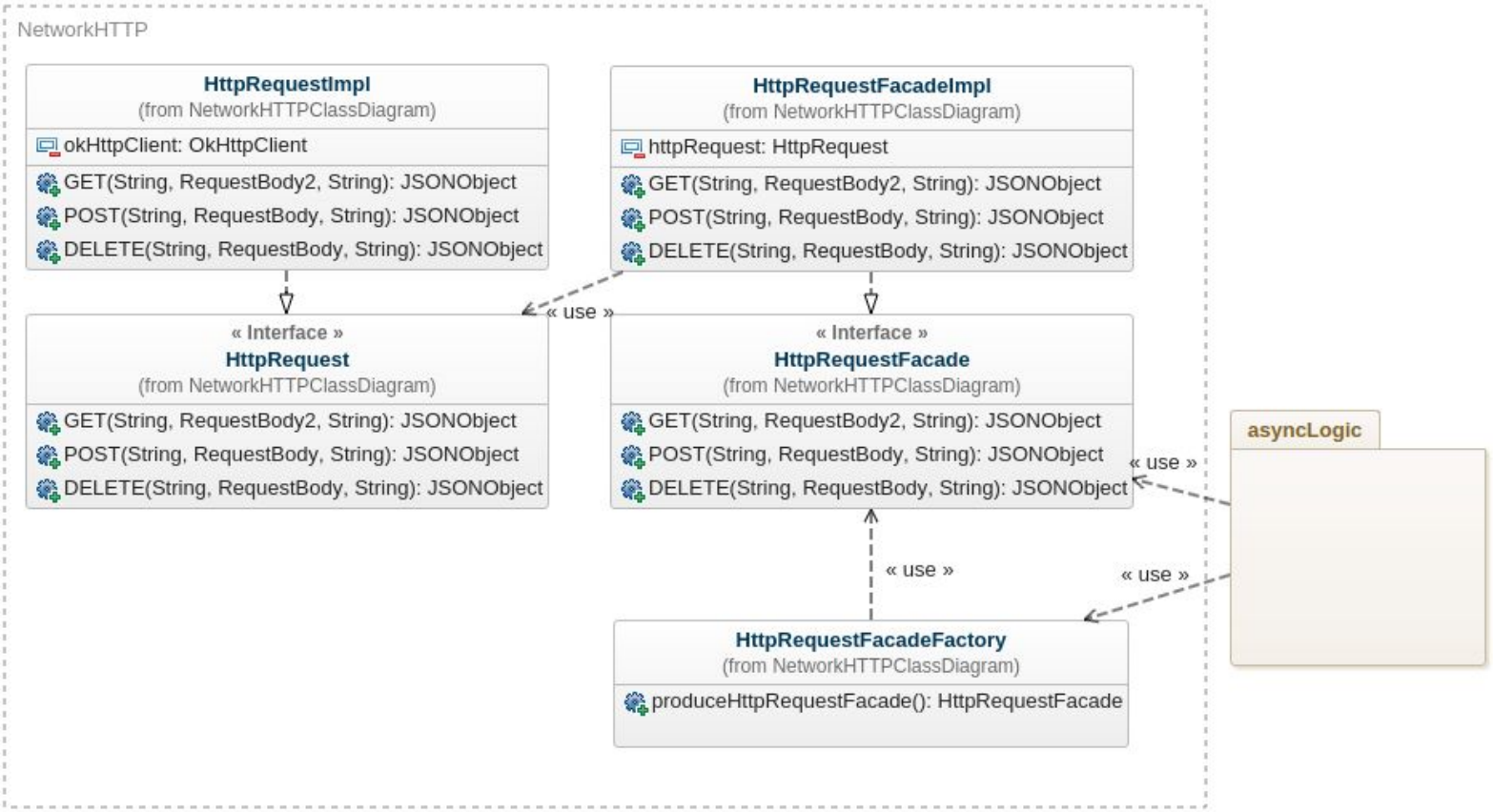
3.5 *networkHTTP*

3.5.1 Zusammenfassung / Übersicht

Diese Komponente ist bildet die Schnittstelle zum Internet in unserem Android Klienten. Sie bekommt von der asyncLogic Komponente eine HTTP Methode sowie einen passenden HTTP body und/oder Header mit Anmeldungs Token übergeben. Außerdem bekommt sie eine URL an die der Request gesendet werden soll. Es werden also keine relevanten Daten in dieser Komponente initialisiert, da sie alles übergeben bekommt.

3.5.2 Innere Struktur

3.5.2.1 Klassendiagramm



Die **networkHTTP** Komponente folgt auch dem **Fassaden Entwurfsmuster**. Für die Umsetzung des **HTTProtokolls** haben wir **OkHttp** benutzt.

OkHttp ist eine Java Bibliothek zur Realisierung von Http Verbindungen und Requests, die von Android komplett unabhängig ist. Zum unabhängigen Testen der **NetworkHttp** Komponente via Unit Tests ist OkHttp also eine gute Wahl.

```
@Override
public JSONObject GET(String path, RequestBody requestBody, String token) throws Exception {
    Request request = new Request.Builder()
            .url(path)
            .addHeader( name: "Authorization",  value: "Bearer " + token)
            .get()
            .build();
    try {
        Response response = client.newCall(request).execute();
        String jsonData = response.body().string();
        JSONObject jsonObject = new JSONObject(jsonData);
        return jsonObject;
    } catch (Exception e) {
        e.getLocalizedMessage();
        Log.d(TAG,  msg: "GET: " + e );
        throw new Exception(e);
    }
}
```

3.5.3 Schnittstellen

Die üblichen Schnittstellen des Fassaden Entwurfsmusters:

```
public interface HttpRequestFacade {
    JSONObject GET(String path, RequestBody data, String token) throws Exception;
    JSONObject POST(String path, RequestBody data, String token) throws Exception;
    JSONObject DELETE(String path, RequestBody data, String token) throws Exception;
}
```

```
public static HttpRequestFacade produceHttpRequestFacade(){
    HttpRequest httpRequest = new HttpRequestImpl();
    return new HttpRequestFacadeImpl(httpRequest);
}
```

3.5.4 Testkonzept

Die Network-Komponente wurde Clientseitig mit dem Mocking-Framework OkHttp getestet.

Die drei Methoden GET, POST, DELETE werden analog getestet. Zur Veranschaulichung der Testablauf für die Methode GET:
Zuerst wird ein Mock-Webserver erstellt, dieser liegt auf dem local-host und bekommt zu beginn einen festen Port zugewiesen. Anschließend definiert man Server Antworten welche linear als Antwort auf Web Anfragen gesendet werden. Ist der Server initialisiert, benutzen wir unsere HttpRequestFacade

um einen Beispiel GET Request an den Webserver zu senden. Die Response wird als JSONObject returned und anschließend per assert mit dem ursprünglichen testResponse String verglichen.

```
@Test
public void httpGutTest3() throws Exception {
    String testResponse = "{\"plan\":\"null\"}";

    MockWebServer server = new MockWebServer();
    server.enqueue(new MockResponse().setBody(testResponse));
    server.start();

    HttpUrl url = server.url( path: "/httpTest/sad");

    HttpRequestFacade httpFacade;
    httpFacade = HttpRequestFacadeFactory.produceHttpRequestFacade();
    FormBody requestBody = new FormBody.Builder()
            .add( name: "userName",  value: "harun")
            .add( name: "password",  value: "harun1")
            .build();

    JSONObject response = httpFacade.GET(url.toString(), requestBody, TOKENVONHARUN);

    Assert.assertEquals(testResponse, response.toString());
}
```

4. Beschreibung des Backend Dienstes

4.1 Zusammenfassung

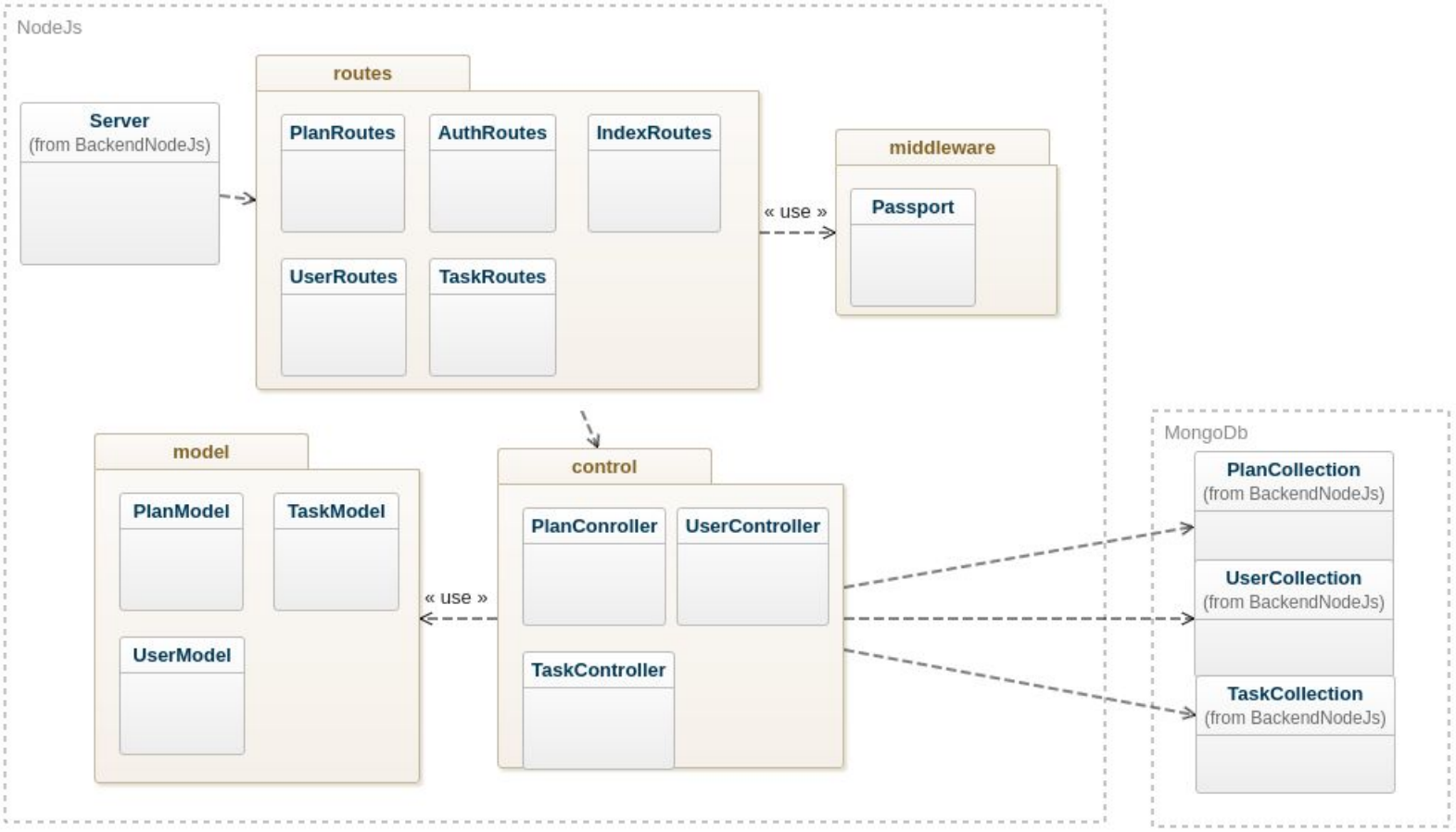
Wir hatten zwei große Anforderungen an unsere Anwendung:

1. Die User sollen ihre Daten persistent speichern können
2. Die User sollen sich mit anderen Usern austauschen können, auch wenn sie nicht im selben Raum sind.

Um diese Anforderungen zu erfüllen haben wir einen Backend Dienst implementiert, welcher auf **zwei Servern** gehostet ist. Der erste Teil ist eine **NodeJS** Anwendung, welche über einen Webserver und **HTTP** erreichbar ist. Der zweite Teil ist eine **MongoDB** die in Verbindung mit der **NodeJS** Anwendung Daten im **BSON** Format speichert und somit eine **No-SQL** Datenbank darstellt.

NodeJS ist eine Serverseitige Javascript Laufzeitumgebung, open-source und eignet sich sehr gut um kleine Projekte schnell umzusetzen. In Kombination mit der **Express Bibliothek**, lassen sich NodeJS basierte Web Dienste sehr gut umsetzen.
MongoDB ist eine **No-SQL** Dokumentendatenbank die sehr gut zu **NodeJS** passt, da sie Dokumente im BSON Format abspeichert, was eine erweiterung des JSON Formats ist. Außerdem eignet sich MongoDB hervorragend für Projekte die sich schnell wandeln können und ist ebenfalls open-source.

4.2 Übersicht



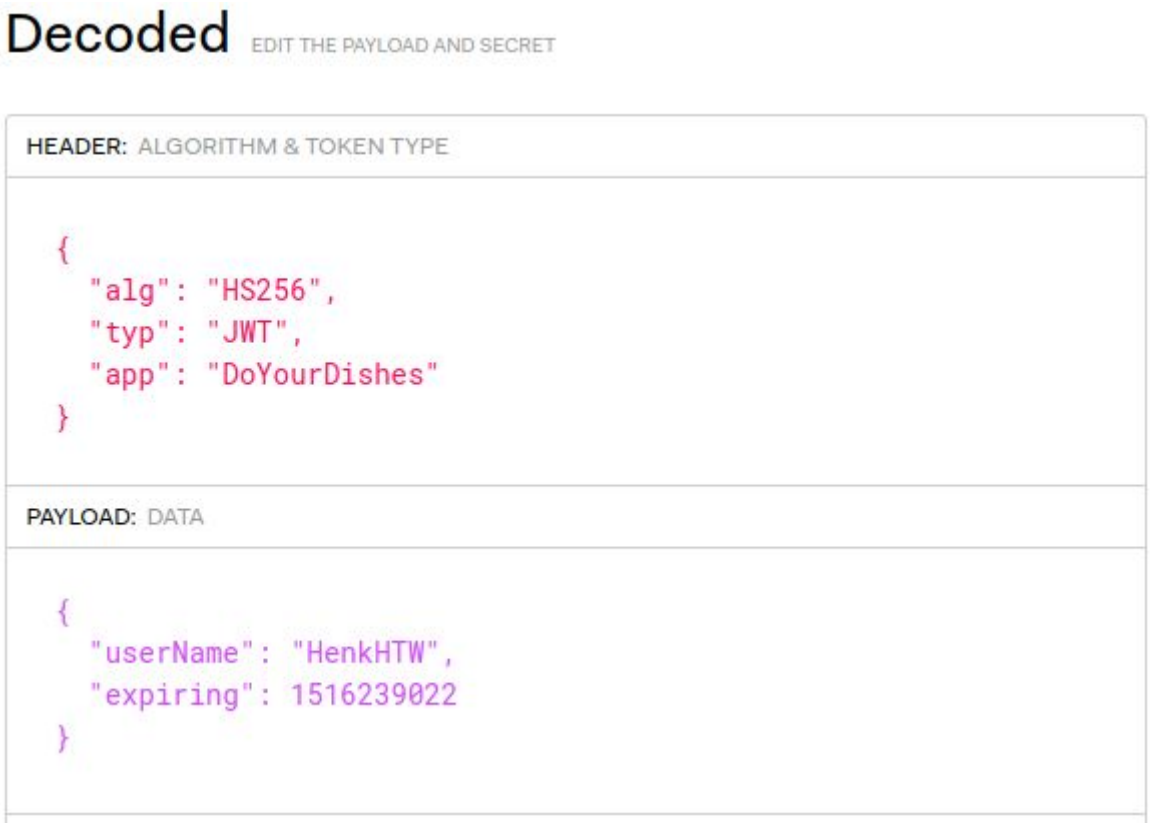
Die **Einstiegs Datei** welche auf dem Server ausgeführt wird ist **Server.js**. Hier wird mittels der **Express** Bibliothek ein **“App” Objekt** erzeugt, welches sich eignet um **HTTP Routen zu verwalten** und zu beantworten sowie “**Middleware” Dienste konfigurieren** kann (In unserem Fall die Anmeldung, über den “**Passport**” mit **Json Web Token**). Ist der Server gestartet läuft aus der Server.js Datei heraus ein Prozess, der auf einem definierten Port auf HTTP Requests wartet und diese auch beantwortet. Die verfügbaren Routen sind in der “**routes**” Komponente definiert. Die Logik welche ausgeführt wird liegt in der “control” Komponente. Die Definition der Dokumente die in der **MongoDB** abgelegt werden ist in der “model” Komponente implementiert.

4.3 Innere Struktur / Technik

Eine weitere Anforderung an unsere Anwendung war, dass sie eine Art **Autorisierung** umsetzt, sodass bespielsweise nur **Mitglieder** eines Plans auch dessen **Tasks** sehen können oder überprüfen können welche weiteren Mitglieder der jeweilige Plan enthält.
Wir brauchten also einen Weg um dem Backend Dienst zu beweisen dass ein User wirklich derjenige ist für den er sich ausgibt zu sein. Bei Webanwendungen gibt es diverse Optionen die über Cookies oder andere Tools arbeiten, die es einem Web Klienten erlauben einen Login Status sicher abzuspeichern.
In einem Android Klienten ist das natürlich nicht möglich, daher fiel unsere Entscheidung schnell auf einen Token, welcher wie ein Reisepass fungiert, den Json Web Token.

4.3.1 Login / JWT (Json Web Token)

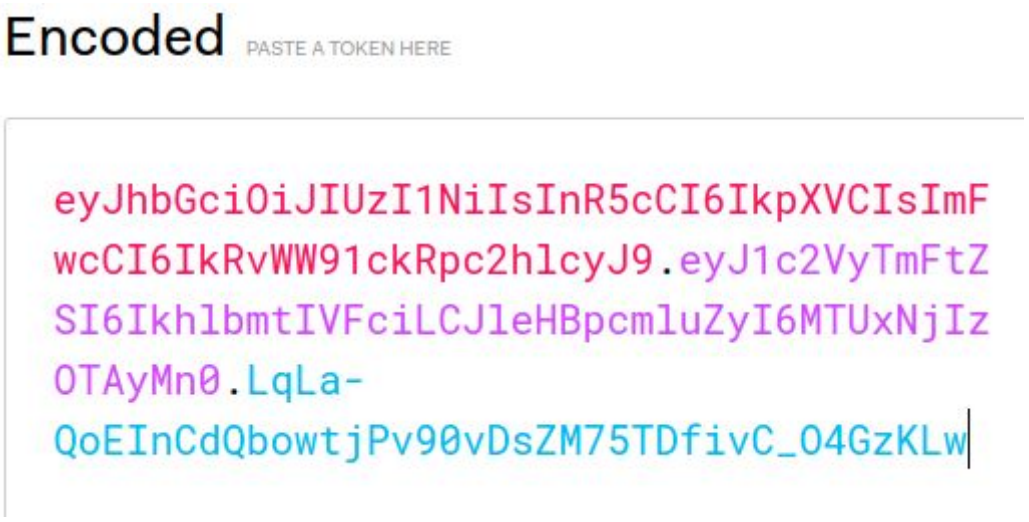
Der Json Web Token ist eine einfache aber sichere Methode Daten zwischen zwei Akteuren zu schicken. Akteur A (der nodeJs Klient) erstellt einen **Header** und eine **Payload**, beides im JSON Format.



Im **Header** stehen Metainformationen wie bspw. der verwendete Hash Algorithmus oder der Typ des Tokens.

In der **Payload** können beliebige Daten stehen. In unserem Fall ein Benutzername und der Moment bis zu dem der jeweilige Token gültig ist.

Der Teil der den Token sicher macht ist die Verifikations Signatur. Sie wird vom Ausstellenden Akteur **A** erstellt indem man den Header mit der Payload **und einem "secret"** durch einen festgelegten Hashalgorithmus abbildet.
Der entscheidende Punkt ist, dass der Akteur A ein Geheimnis verwendet, bspw. einen String, welches nur **A** kennt. So kann **A** überprüfen ob die Verifikations Signatur zum inhalt des Headers und der Payload passt und nur Akteure die das Geheimnis kennen, können denselben Hashwert erzeugen.



Nun schickt **A** den Token an Akteur **B**, der zuvor per Eingabe einer richtigen Benutzername und Passwort Kombination bewiesen hat, dass er Zugriff auf das angemeldete Konto haben darf. Anschließend schickt **B** den Token stets in seinen HTTP Requests im Header Feld mit und beweist **A** somit, dass er wirklich derjenige ist für den er sich ausgibt, ohne ständig seine Benutzername und Passwort Kombination übertragen zu müssen.

4.3.3 Code Snippet Login

```
exports.login = async (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return  retErr(res, errors,  errCode: 418,  customMessage: 'INVALID_INPUT');
    }
    const {userName, password} = req.body;
    const user = await User.findOne({userName: userName});
    if (user && user.userName) {
        const isPasswordMatched = await user.comparePassword(password);
        if (isPasswordMatched) {
            // Sign token
            const token = jwt.sign( payload: {userName: userName}, secret,
                 options: {
                    expiresIn: 86400,
                });
            const userToReturn = {...user.toJSON(), ...{token}};
            delete userToReturn.password;
            res.status(200).json(userToReturn);
        } else {
            return  retErr(res,  err: {},  errCode: 418,  customMessage: 'WRONG_USER_OR_PW');
        }
    } else {
        return  retErr(res,  err: {},  errCode: 418,  customMessage: 'WRONG_USER_OR_PW');
    }
}
```

Dies ist die Controller Methode für einen Login Request am Backend Dienst
Zuerst wird überprüft ob sowohl Password als auch Username im HTTP Body vorhanden sind.

Anschließend werden diese entnommen und es wird unter verwendung der **mongoose** libray mit **User.findOne({userName: userName});**
In der MongoDB nachgeschaut ob ein Benutzer unter dem userName vorhanden ist.

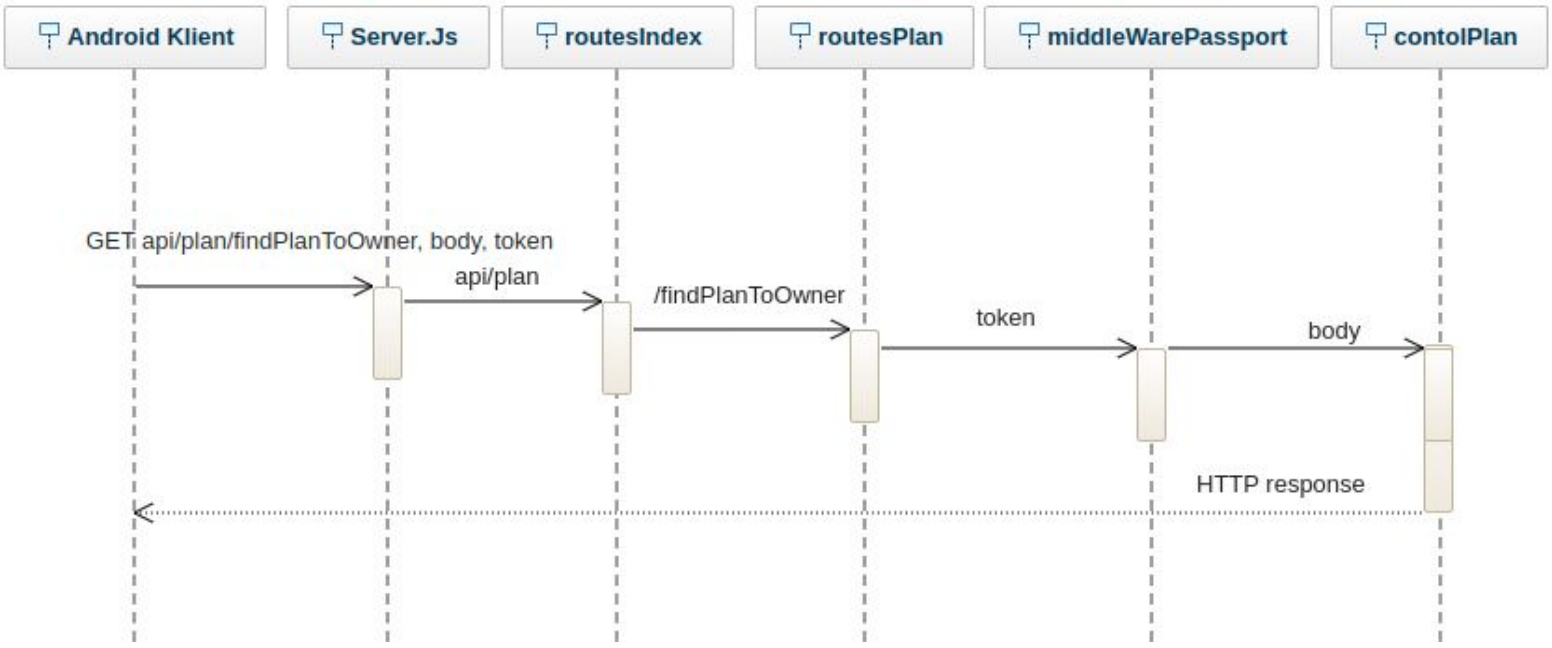
Falls vorhanden, wird das Passwort mit der **comparePassword** Methode, welche im userModel implementiert ist, gecheckt (da wir nur Hashes der Passwörter speichern werden die Hashes verglichen). Stimmt dies überein wird der oben beschriebene Token mit dem gesendeten Benutzernamen erstellt und als Response im JSON Format an den Android Klienten zurückgesendet.

4.3.4 Rest-Api

Im folgenden wird erläutert welche Routen unsere Rest-API anbietet und wie diese im Javascript definiert sind.

Beispiel **fetchPlan Sequenz**:

4.3.4.1 Sequenzdiagramm



4.3.4.2 Routen / Pfade

Route name	URL	HTTP Verb	Description
createPlan	/api/plan/createPlan	POST	Bekommt namen, liest JWT -> erstellt neuen Plan zum JWT user
deletePlan	/api/plan/deletePlan	DELETE	Bekommt planId, liest JWT und loescht den Plan falls JWT user = owner
addUser	/api/plan/addUser	POST	Bekommt userName, fuegt ihn zu Plan hinzu
findPlan	/api/plan/findPlan	GET	Liest JWT und antworet mit Plan dem der JWT user

Route name	URL	HTTP Verb	Description
createTask	/api/task/createTask	POST	Bekommt namen, Punkte , liest JWT -> erstellt neue Task im Plan des JWT user
delSingleTask	/api/task/delSingleTask	DELETE	Bekommt taskId, liest JWT und loescht den Plan falls JWT user = mitglied Task.plan
fulfillTask	/api/task/fulfillTask	POST	Bekommt taskId, liest JWT und fuegt JWT user punkte der Task hinzu (falls er im Task.Plan ist)

Route name	URL	HTTP Verb	Description
createUser	/api/user/createUser	POST	Nimmt username und password -> erstellt neuen user
deleteUser	/api/user/delUser	DELETE	Liest JWT aus, bekommt username und loescht den zugehoerigen user
login	/api/auth/login	POST	Bekommt username und password und antwortet mit JWT
whoAmI	/api/auth/whoAmI	GET	Liest JWT und antwortet mit den UserDaten zum JWT

4.3.3 Hosting / Server

Damit unsere Anwendung 24/7 nutzbar ist hatten wir zwei Optionen:

1. einen eigenen Webserver zur verfügung stellen und sowohl **NodeJS Klient** als auch **MongoDB** Anwendung auf diesem laufen lassen
2. **Heroku** in Verbindung mit **MongoDB Atlas** nutzen

Da wir beide keine Hardware übrig hatten um über unseren privaten Internetanschluss einen Server laufen zu lassen war schnell klar, dass wir uns für die 2. Option entscheiden.

Heroku ist ein cloud Platform as a Service Dienst, welcher es ermöglicht bis zu einer gewissen Anzahl an Aufrufen/Zeitintervall, kostenlos NodeJS Anwendungen auf einem Webserver bereitzustellen. **MongoDB Atlas** verfolgt ein ähnliches Prinzip bei der man 500mb MongoDB Einheiten bereitstellen kann.

Hierzu sollte gesagt sein, beide **Cloud Dienste** erfüllen **keinen weiteren Zweck** als eine **erreichbarkeit des Backend Dienstes** zu gewährleisten. Die Bereitstellung unserer Anwendung auf einem eigens betriebenen Server würde im Prozess minimal abweichen von dem was die Cloud Dienste für uns tun.

4.5 Testkonzept

Zum Testen einer Solchen NodeJs + ExpressJs Anwendung eignen sich zum einen die Javascript Assertion Library **Chai** sowie dessen Erweiterung **ChaiHTTP**

Bei den Tests handelt es sich um Integrationstests, die HTTP Anfragen simulieren und so sowohl die Funktionalität und richtige Implementierung der NodeJs Logik, als auch die Verbindung zur MongoDB testen.

Zu Beginn des Tests wird eine Verbindung zur (test)Datenbank aufgebaut und diese wird komplett gelöscht. Anschließend laufen Linear Tests durch, die aufeinander aufbauen

4.5.1 Beispiel Test

TestCase: **User erstellen, einloggen, whoami aufrufen -> sehen ob Token richtig ist**

```
describe( title: "User",  fn: () => {
    it( title: "(HAPPY PATH) should create a user, login and return userName with whoAmI",  fn: (done : Done ) => {
        chai.request(app) Agent
            .post( url: '/api/user/createUser', ) SuperAgentRequest
            .send( data: {userName: 'henk', password: 'iloveandroid'}) SuperAgentRequest
            .end( callback: (err, res : Response ) => {
                res.should.have.status( code: 200);
                res.body.should.be.a( type: 'object');
                chai.request(app) Agent
                    .post( url: '/api/auth/login', ) SuperAgentRequest
                    .send( data: {userName: 'henk', password: 'iloveandroid'}) SuperAgentRequest
                    .end( callback: (err, res : Response ) => {
                        res.should.have.status( code: 200);
                        res.body.should.be.a( type: 'object');
                        token = res.body.token;
                        chai.request(app) Agent
                            .get( url: '/api/auth/whoAmI') SuperAgentRequest
                            .auth(token,  options: { type: 'bearer' }) SuperAgentRequest
                            .end( callback: (err, res : Response ) => {
                                res.should.have.status( code: 200);
                                res.body.should.be.a( type: 'object');
                                chai.expect(res.body.data.userName).equal( value: 'henk');
                                done();
                            });
                    });
            });
    });
```

Vergleichbare Tests sind für alle denkbaren Anwendungsfälle implementiert zu finden im Backend Ordner Backend/tests/tests.js

Auch Edge Cases wie sehr lange Eingaben oder Sonderzeichen werden überprüft und vom NodeJs Dienst richtig behandelt.