



Projektmanagement

Projektdokumentation – WG-Planer –

Version 0.1

Autor: Moritz Müller, Rosa Kern,
Zusibell Jimenez

Kurs: TINF19AI1

Historie der Dokumentversionen

Version	Datum	Autor	Änderungsgrund / Bemerkungen
0.1	03.06.2020	Projektteam	Ersterstellung

Inhaltsverzeichnis

- 1 Einleitung
 - 1.1 Allgemeines
 - 1.1.1 Abkürzungen
 - 1.1.2 Ablage, Bezüge zu anderen Dokumenten
 - 1.2 Projektstammdaten
 - 1.2.1 Projekttitel
 - 1.2.2 Auftraggeber
 - 1.2.3 Projektteam
- 2 Details Projektauftrag
 - 2.1 Problemstellung
 - 2.2 Zielgruppe
 - 2.3 Lösung
 - 2.4 Nutzen für Auftraggeber
 - 2.5 Übersicht Projektanforderungen
 - 2.5.1 Nicht funktionale Anforderungen
 - 2.5.2 Funktionale Anforderungen
 - 2.6 Kosten
 - 2.7 Zeitmanagement
 - 2.8 Scrum Board
 - 2.9 Architekturmodell
- 3 Datenbank
 - 3.1 ER-Modell
 - 3.2 Abbildung der Tabellen
 - 3.3 Normalform
 - 3.4 Integritätsbedingungen
 - 3.5 Datenpflege
 - 3.6 Datensicherheit
- 4 UI Übersicht
 - 4.1 Beschreibung (Funktion, SQL Abfragen)

1 Einleitung

1.1 Allgemeines

1.1.1 Abkürzungen

Abkürzung	Begriff
DB	Datenbank
UI	User Interface
AWS	Amazon Web Services
NF	Normalform

1.1.2 Ablage, Bezüge zu anderen Dokumenten

Unser Projekt und die Dokumentation liegen in einem GIT Repository.

1.2 Projektstammdaten

1.2.1 Projekttitel

„WG-Planer“

1.2.2 Auftraggeber

Roche Diagnostics

Die **Roche Diagnostics International AG** ist eine Tochter des Pharmakonzerns Hoffmann-La Roche.

1.2.3 Projektteam

Name	E-Mail	Matrikelnummer
Rosa Kern	S191076@student.dhbw-mannheim.de	8831544
Moritz Müller	S191505@student.dhbw-mannheim.de	8890225
Zusibell Jimenez	S191006@student.dhbw-mannheim.de	9922138

2 Details Projektauftrag

2.1 Problemstellung

Mit dem Beginnen eines neuen Lebensabschnittes stellt nicht nur die Universität, sondern auch eine neue Wohnsituation für die meisten Studenten eine Herausforderung dar. In Wohngemeinschaften herrscht meistens eine mangelnde Struktur und Kommunikation, welche zu Terminbeeinträchtigungen führt und die Studierenden in ihrem Zeitmanagement einschränkt. Dies lässt vermeidbare Probleme im Alltag entstehen.

Das grundlegende Problem ist dabei die fehlende Kommunikationsmöglichkeit.

Die entstehenden Konflikte beginnen beim Kollidieren der Pläne für das tägliche Duschen und gehen bei der Verwaltung der regelmäßigen Aufgaben im Haushalt weiter. Oft gibt es nur wenige bzw. eine Waschmöglichkeit, wodurch Leerläufe oder Wartezeiten entstehen. Außerdem sind Termine oder Veranstaltungen nicht online einzusehen, sondern hängen nur aus, weshalb sie oftmals übersehen werden.

Diese Probleme können mit einer zentralen Kommunikations- und Planungsmöglichkeit behoben werden.

2.2 Zielgruppe

Aufgrund der Problemstellung lässt sich unsere Zielgruppe auf WG Bewohner eingrenzen. Je mehr Mitglieder die WG besitzt desto mehr kann die App helfen, da die Kommunikation und Übermittlung von Informationen mit steigender Personenanzahl oft schlechter und unübersichtlicher wird.

2.3 Lösung

Unser Ziel ist es die Lösungen dieser Probleme in einer Anwendung zu vereinigen und ein zentrales Tool zur Organisation und Kommunikation für Wohngemeinschaften zu entwickeln.

Unsere App beinhaltet zwei Hauptelemente. Zum einen gibt es einen Termin- bzw. Organisationskalender, welcher jedem Mitglied der WG zugänglich ist. Dabei erhält jeder Nutzer eine eigene Farbe zur Differenzierung. In diesem Kalender werden für jeden Mitbewohner die Aufgaben im Haushalt eingetragen und können nach Bearbeitung als „erledigt“ markiert werden. Ebenso können normale Termine (Treffen mit Hausmeister, Reparaturen, WG-Meetings, Gemeinsames Kochen etc.) oder Zeitliche Strukturen (Wann geht man duschen? Wann hat man vor zu waschen? etc.) eingetragen werden.

Es wird außerdem eine Einkaufsliste geben. Dadurch kann gezielter und strukturierter der Haushalt geplant werden.

Zur Nutzung von Haushaltsgeräten (Waschmaschine/ Trockner) gibt es eine Funktion, mit der ein Nutzer jederzeit überprüfen kann, ob bzw. wie lange eine Maschine in Benutzung ist.

Ein Newsfunktion stellt den zweiten Teil unserer App dar, welche die Übermittlung von Neuigkeiten oder generellen Informationen gewährleistet.

Die App schreiben wir in Java und nutzen Andoid Studio. Unsere Datenbank wird auf einem AWS Server laufen.

2.4 Nutzen für Auftraggeber

Ein dualer Student in unserem Projektteam arbeitet bei Roche. Dieser wohnt in einer großen WG und ist somit in unserer Zielgruppe.

Oftmals erfolgt die Kommunikation in einer WG ausschließlich über Whatsapp oder persönlich zwischen einzelnen WG-Bewohnern. Dabei gehen oft wichtige Informationen verloren oder werden vergessen und eine Planung ist nicht möglich. Auch der Austausch von Neuigkeiten oder wichtigen Informationen geschieht oft nur teilweise, da die Informationen nicht, nur teilweise oder falsch weitergegeben werden. Unsere App stellt somit eine einzigartige Lösung dar, welche das Leben von Studenten sehr erleichtern kann.

Betrachtet man die Opportunitätskosten bei weiterhin schlechter Planung, kann diese Unstrukturiertheit und die Probleme im Zeitmanagement auf den Uni-Alltag übergreifen. Die Leistungen des Studenten können sinken und im schlimmsten Fall zu einer Exmatrikulation führen. Da die Unternehmen bestmögliche Leistungen von den dualen Studenten erwarten, kommt es somit zu einem Verlust von Leistung eines Mitarbeiters oder auch zu hohen Kosten. Falls der Student exmatrikuliert wird, folgt daraus ein Verlust für das Unternehmen durch überflüssige Ausgaben für den Studenten.

Insgesamt kann man mit unserer Software bessere, getaktetere und einheitlichere Zeitabläufe gewährleisten. Es kommt somit zu Zeiteinsparungen und festen Strukturen im Alltag. Dabei können Verspätungen des Studenten minimiert werden, der Alltag wird stressfreier und übersichtlicher, wodurch das Risiko einer Exmatrikulation minimiert wird. Auch werden Termine präziser eingehalten. Die Gesamtkommunikation, Planung und somit auch das soziale Zusammenleben verbessert sich. Die Fürsorgepflicht des Arbeitgebers ist einfacher erfüllbar.

Somit wird der Student leistungsfähiger und somit wertvoller für das Unternehmen.

2.5 Übersicht Projektanforderungen

2.5.1 nicht funktionale Anforderungen:

- Benutzerfreundlichkeit: Die App soll simpel aber funktional sein. Man soll sich anhand von Grafiken oder Beschreibungen gut zurecht finden.
- Zuverlässigkeit: Da man die App im Alltag nutzen soll, sollte die Oberfläche zuverlässig mit der DB zusammenarbeiten, damit alle Informationen so aktuell wie möglich sind.
- Änderbarkeit: Der Code sollte leicht zugänglich und Änderbar sein. (Git-Repository)

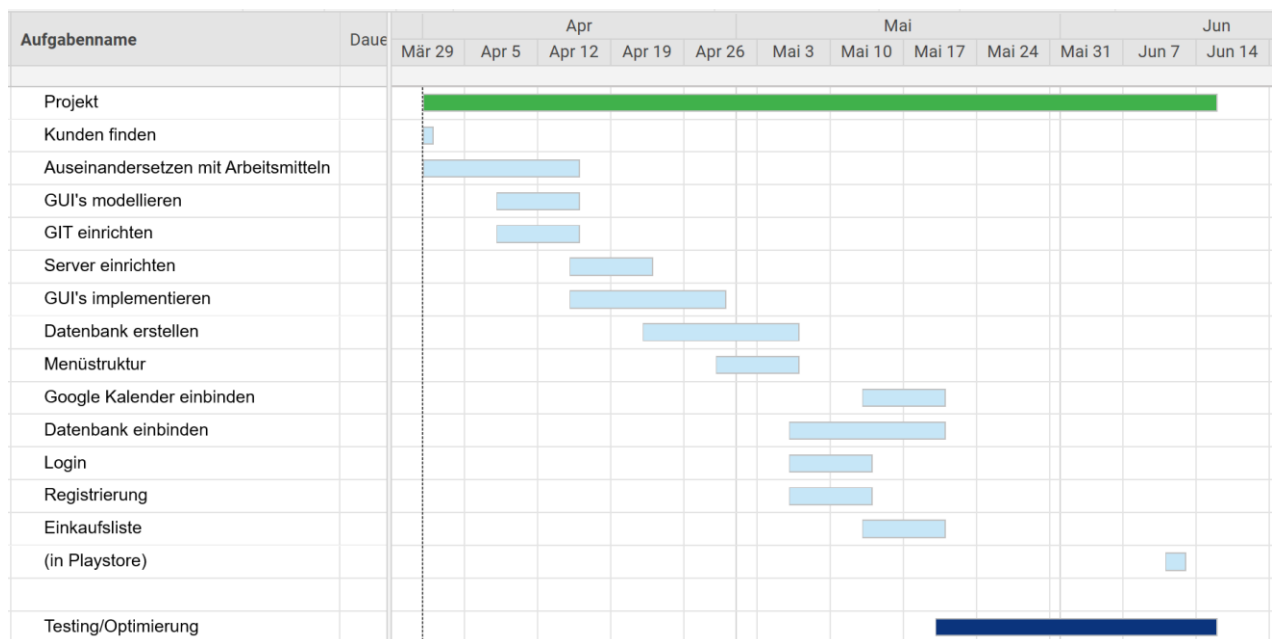
2.5.2 funktionale Anforderungen:

- Es soll ein Nutzerverzeichnis (Login/ Register) geben.
- Man soll eine WG erstellen oder einer WG beitreten können.
- Es soll ein Menü zur Verwaltung von der WG geben.
- Es soll einen Kalender geben, in dem jeder User der WG Termine etc. eintragen kann und die Eintragungen der anderen sehen kann.
- Es soll eine Einkaufsliste geben, auf der man Items hinzufügen kann. Man kann diese außerdem abhaken.
- Es soll außerdem eine News Funktion geben bei der jeder User Neuigkeiten mit den anderen WG Bewohnern teilen kann.

2.6 Kosten

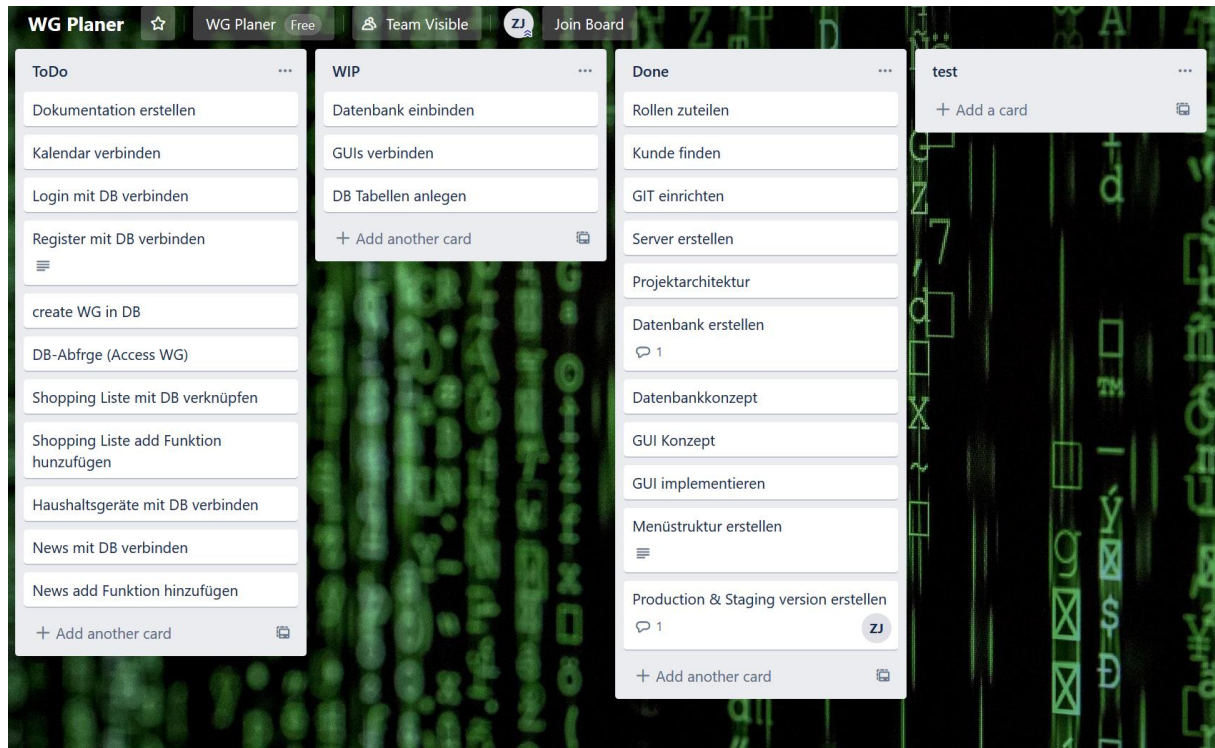
Die Problemstellung kann in einem Semester bearbeitet werden. Dabei besteht das Team aus 4 Studenten, wobei die Kosten als Bearbeitungszeit interpretiert werden können. Da wir einen kostenlosen Server für 12 Monate nutzen, fallen vorerst keine Serverkosten an.

2.7 Zeitmanagement



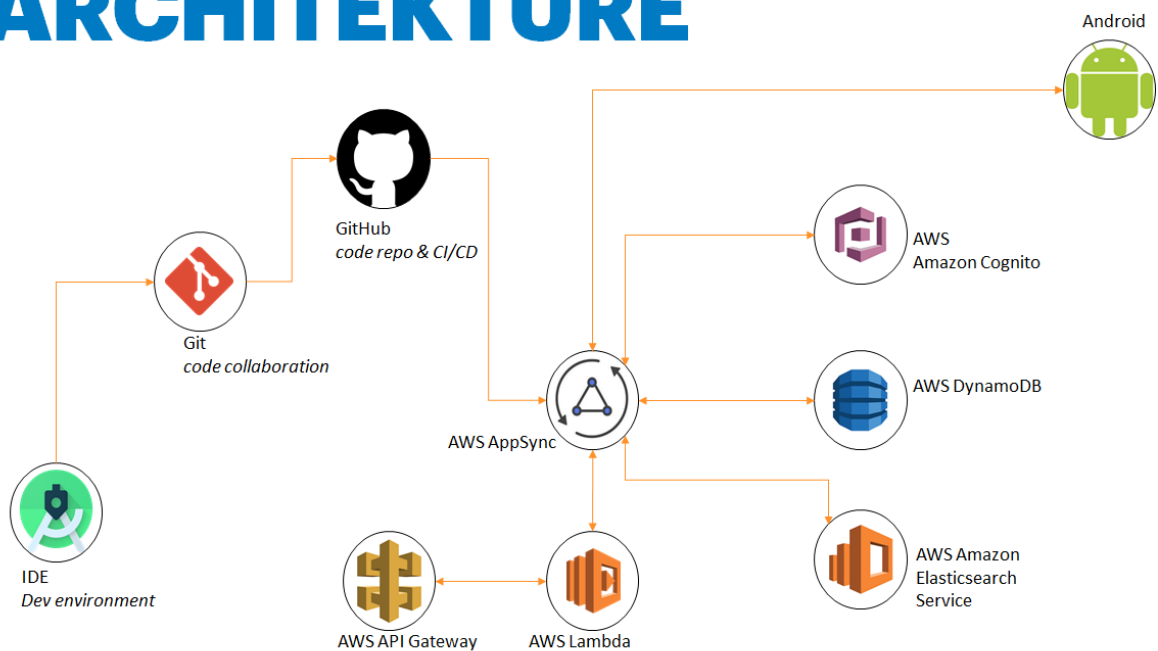
2.8 Scrum-Board

Beispiel von dem 22.04.2020:



2.8 Architekturmodell

ARCHITEKTURE



Wir Nutzen von AWS folgende Services:

- amazon cognito
- amazon lambda
- AWS appSync (APIs - GraphQL)
- AWS Amplify
- DynamoDB
- CloudFormation
- S3

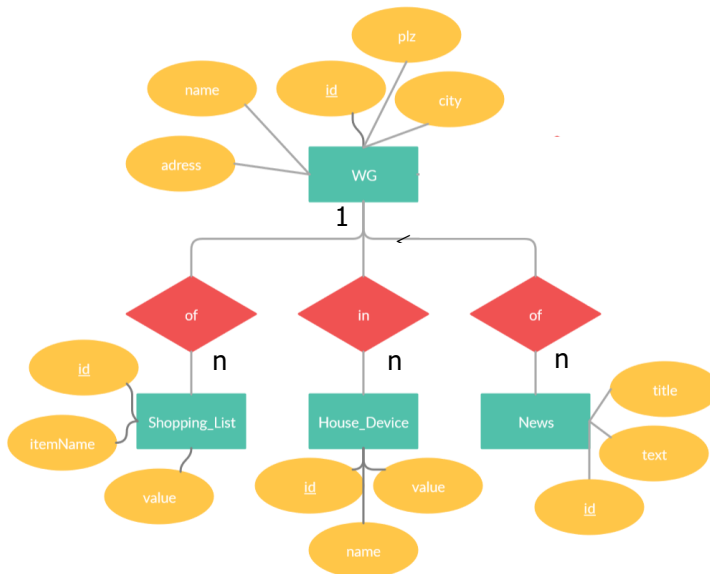
Die Daten der Nutzer werden in dem Amazon Userpool gespeichert. Alle restlichen Daten werden in verschiedene Tabellen in unsere DynamoDB gespeichert. Über den AWS AppSync können wir die Features von AWS nutzen. Unser Projekt mit allen Ressourcen etc. liegt in einem Git-Repository.

Verbindung von AWS über AWS AppSync:

```
1  {
2    "AppSyncApiName": "wgplaner",
3    "DynamoDBBillingMode": "PAY_PER_REQUEST",
4    "DynamoDBEnableServerSideEncryption": "false",
5    "AuthCognitoUserPoolId": {
6      "Fn::GetAtt": [
7        "authwgplaner81671fa3",
8        "Outputs.UserPoolId"
9      ]
10   }
11 }
```

3 Datenbank

3.1 ER-Modell



3.2 Abbildung der Tabellen

1. **Jeden Entitätstyp als Tabelle darstellen**
 2. **WG** (*id*, *name*, *adress*, *city*, *plz*)
 3. **Shopping_List** (*id*, *itemName*, *value*)
 4. **House_Device** (*id*, *name*, *value*)
 5. **News** (*id*, *title*, *text*)
2. **Jede n:m Beziehung als Tabelle darstellen**
/ keine m:n Beziehung
3. **Jede 1:n/1 Beziehung +Attribut als Tabelle darstellen**
Keine 1:n/1 Beziehung mit Attribut vorhanden
4. **- Jede 1:n/1 Beziehung ohne Attribut als Tabelle (freie Beziehung)**
nicht vorhanden

- E2 erhält primär Schlüssel von E1 (zwingende Beziehung)
nicht vorhanden

3.3 Normalform

1. NF

Ein **Relationstyp** (Tabelle) befindet sich in der **ersten Normalform (1NF)**, wenn die **Wertebereiche** der Attribute des Relationstypen **atomar** sind.

-> Die Wertebereiche sind atomar, da Adresse schon in dem ER-Modell auseinandergezogen wurde.

2. NF

Ein Relationstyp (Tabelle) befindet sich genau dann in der **zweiten Normalform (2NF)**, wenn er sich in der ersten Normalform (1NF) befindet und jedes Nichtschlüsselattribut von jedem Schlüsselkandidaten voll funktional abhängig ist.

-> Ist vorhanden

3. NF

Ein Relationstyp befindet sich genau dann in der **dritten Normalform (3NF)**, wenn er sich in der zweiten Normalform (2NF) befindet und kein Nichtschlüsselattribut transitiv von einem Kandidatenschlüssel abhängt.

-> Ist vorhanden

4. NF

Ein Relationenschema ist dann in der 4. Normalform, wenn es in der BCNF ist und nur noch triviale mehrwertige Abhängigkeiten (MWA) enthält.

-> ist vorhanden

5. NF

Eine Relation ist in 5NF, wenn sie in der 4NF ist und keine mehrwertigen Abhängigkeiten enthält, die voneinander abhängig sind.

-> ist vorhanden

3.4 Integritätsbedingungen

→WG

Attribut	Kurzbeschreibung	Datentyp	Wertebereich	NULL zulässig
id	Primärschlüssel	Int(10)	1-10 Stellen	No
Name	Primärschlüssel	String(50)	1-50 Zeichen	No
adress	Primärschlüssel	String(50)	1-50 Zeichen	No
city	Primärschlüssel	String(50)	1-50 Zeichen	No
Plz	Primärschlüssel	Int(5)	1-5 Stellen	No

→Shopping_List

Attribut	Kurzbeschreibung	Datentyp	Wertebereich	NULL zulässig
Id	Primärschlüssel	Int(10)	1-10 Stellen	No
itemName	Primärschlüssel	String(50)	1-50 Zeichen	No
value	Primärschlüssel	Int(1)	eine Stelle	yes

→House_Device

Attribut	Kurzbeschreibung	Datentyp	Wertebereich	NULL zulässig
Id	Primärschlüssel	Int(10)	1-10 Stellen	No
Name	Primärschlüssel	String(50)	1-50 Zeichen	No
value	Primärschlüssel	Int(1)	Eine Stelle	yes

→News

Attribut	Kurzbeschreibung	Datentyp	Wertebereich	NULL zulässig
Id	Primärschlüssel	Int(10)	1-10 Stellen	No
Text	Primärschlüssel	String(50)	1-50 Zeichen	No
title	Primärschlüssel	String(50)	1-50 Zeichen	No

3.5 Datenpflege

Alle Daten werden zentral auf einer DB gespeichert, auf welche nur wir (Entwicklerteam) zugreifen können. Durch das Datenbankmanagementsystem AWS werden die Daten sicher und zuverlässig verwaltet. Da alle Daten manuell eingepflegt werden, kann es zu Fehlern kommen. Die Daten müssen aber individuell eingegeben werden. Durch einen Login kann nicht jeder auf die sensiblen Daten (Adresse, Termine etc.) zugreifen und es ist eine Datensicherheit vorhanden.

Die Nutzer samt Daten sind in dem Amazon Userpool gespeichert.

3.6 Datensicherheit

Die DB ist nur für die Mitglieder des Projektteams zugänglich. Unsere App hat eine Login Funktion, welche eine wichtige Sicherheitsstufe darstellt. Mit seinen Login-Daten hat man nur zu seinen eigenen Daten Zugriff.

AWS

„Wir sind Mitglied in zahlreichen Vereinigungen zum Schutz von Privatsphäre und Datensicherheit. AWS besitzt zudem mehrere international anerkannte Zertifizierungen und Akkreditierungen, die eine Einhaltung der Richtlinien für die Prüfung durch externe Dritte nachweisen. AWS-Kunden haben die Kontrolle über ihre Inhalte und deren Speicherort.“

Quelle: <https://aws.amazon.com/de/compliance/amazon-information-requests/>

3.7 Code

Mit der `AWS::Cognito::UserPool`-Ressource wird ein Amazon Cognito-Benutzerpool erstellt. Um die Entität in der AWS CloudFormation-Vorlage zu deklarieren, verwenden wir die folgende Syntax:

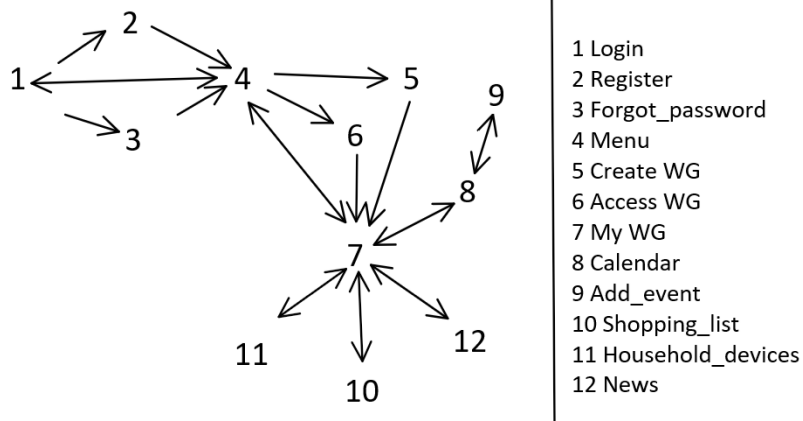
```
{
  "Type" : "AWS::Cognito::UserPool",
  "Properties" : {
    "AccountRecoverySetting" : AccountRecoverySetting,
    "AdminCreateUserConfig" : AdminCreateUserConfig,
    "AliasAttributes" : [ String, ... ],
    "AutoVerifiedAttributes" : [ String, ... ],
    "DeviceConfiguration" : DeviceConfiguration,
    "EmailConfiguration" : EmailConfiguration,
    "EmailVerificationMessage" : String,
    "EmailVerificationSubject" : String,
    "EnabledMfas" : [ String, ... ],
    "LambdaConfig" : LambdaConfig,
    "MfaConfiguration" : String,
    "Policies" : Policies,
    "Schema" : [ SchemaAttribute, ... ],
    "SmsAuthenticationMessage" : String,
    "SmsConfiguration" : SmsConfiguration,
    "SmsVerificationMessage" : String,
    "UsernameAttributes" : [ String, ... ],
    "UsernameConfiguration" : UsernameConfiguration,
    "UserPoolAddOns" : UserPoolAddOns,
    "UserPoolName" : String,
    "UserPoolTags" : Json,
    "VerificationMessageTemplate" : VerificationMessageTemplate
  }
}
```

In der `schema.graphql` Datei, werden die Columns der Tabellen aus der DB definiert. Mithilfe von diesem Schema können wir die einzelnen Zellen der Tabellen ansprechen, speichern und löschen.

Schema.graphql:

```
1  type WG @model @auth(rules: [{allow: private}]) {
2    id:ID!
3    name: String!
4    address: String!
5    city: String!
6    plz: String!
7    shoppingList: [ShoppingList] @connection(keyName:"WgShopList", fields: ["id"])
8    householdDevices: [HouseholdDevices] @connection(keyName:"WgDevices", fields: ["id"])
9    news: [News] @connection(keyName:"WgNews", fields:["id"])
10   calender: [Calendar] @connection(keyName: "WgCalendar", fields:["id"])
11 }
12
13 type ShoppingList @model @auth(rules: [{allow: private}])
14 @key(name:"WgShopList", fields:["wgID", "itemName", "value"]) {
15   id: ID!
16   wgID: ID!
17   itemName: String!
18   value: String
19   wg: WG @connection(fields:["wgID"])
20 }
21
22 type HouseholdDevices @model @auth(rules: [{allow: private}])
23 @key(name:"WgDevices", fields:["wgID","deviceName","value"]){
24   id: ID!
25   wgID: ID!
26   deviceName: String!
27   value: String
28   wg: WG @connection(fields:["wgID"])
29 }
30
31 type News @model @auth(rules: [{allow: private}])
32 @key(name:"WgNews", fields:["wgID", "title", "text"]) {
33   id: ID!
34   wgID: ID!
35   title: String!
36   text: String
37   wg: WG @connection(fields:["wgID"])
38 }
39
40
41 type Calendar @model @auth(rules: [{allow: private}])
42 @key(name:"WgCalendar", fields:["wgID", "events", "date", "time"]) {
43   id: ID!
44   wgID: ID!
45   events : String!
46   date: AWSDate!
47   time: AWSTime!
48   wg: WG @connection(fields:["wgID"])
49 }
```

4 UI Übersicht



Benutzte Farbe für Toolbar oben:



Font: Open Sans
Schriftgröße Button: 18sp
Schriftgröße Satz: 24sp

4.1 UIs Beschreibung + SQL-Abfragen

1:



Das ist das Hauptmenü bzw. der Login des WG-Planers.

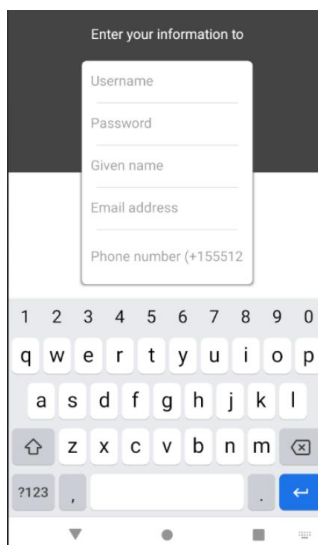
Hier hat man die Möglichkeit sich entweder anzumelden oder neu zu registrieren (Weiterleitung zu UI 2).

Für die Anmeldung der Nutzer nutzen wir den Amazon Userpool. Die Daten werden somit dort gespeichert, gelöscht und verglichen.

Verbindung zum Userpool:

```
40
41     mAWSAppSyncClient = AWSAppSyncClient.builder()
42         .context(getApplicationContext())
43         .awsConfiguration(new AWSConfiguration(getApplicationContext()))
44         .cognitoUserPoolsAuthProvider(new CognitoUserPoolsAuthProvider() {
45             @Override
46             public String getLatestAuthToken() {
47                 try {
48                     return AWSMobileClient.getInstance().getTokens().getIdToken().getTokenString();
49                 } catch (Exception e){
50                     Log.e("APPSYNC_ERROR", e.getLocalizedMessage());
51                     return e.getLocalizedMessage();
52                 }
53             }
54         }).build();
55
56     loginToMenu();
57
58
59 }
60
61
62 public void loginToMenu(){
63     Button navLogin_To_Menu = (Button) findViewById(R.id.btn_login);
64     navLogin_To_Menu.setOnClickListener(new View.OnClickListener() {
65         @Override
66         public void onClick(View v) {
67             startActivity(new Intent(LoginScreen.this, MainMenu.class));
68         }
69     });
70
71
72
73 }
74 }
```

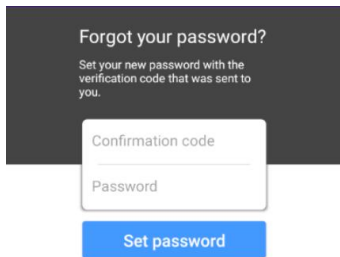
2:



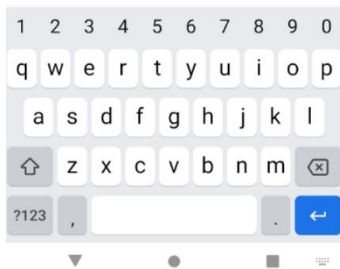
Mit dieser UI kann man sich einen neuen Account erstellen.

Dafür muss man jeweils die Werte für den Usernamen, das Passwort, die E-Mail und die Telefonnummer in den zugehörigen EditText schreiben.

3:

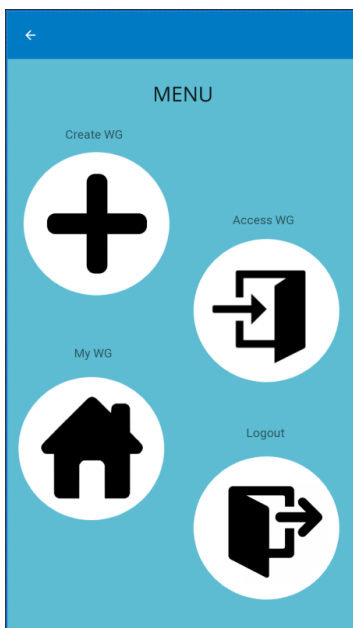


Auf dieser UI kann der Nutzer sein Passwort nachfragen.



SELECT E-Mail, password FROM User

4:



Auf der UI "Menü" kann der User seine WG verwalten.

Wenn der User eine WG erstellen möchte, kann er dies über den "Create WG" Button machen. Er wird zur UI 5 weitergeleitet.

Wenn der User einer bereits erstellten WG beitreten möchte, kann er dies über den Button "Access WG" machen. Man wird zur UI 6 weitergeleitet.

Wenn man bereits einer WG beigetreten ist, kann man die einzelnen Funktionen über den Button "My WG" erreichen. Man wird auf die UI 7 weitergeleitet.

Über den "Logout" Button kommt man zur UI 1 und damit zum Hauptmenü.

5:

Auf dieser UI kann man eine neue WG erstellen.

Dafür muss der User in den einzelnen EditText Feldern den WG-Namen, die Adresse, die Stadt und die Postleitzahl eingeben.

Mit dem Button "Create WG" werden die Werte aus den EditText Feldern in die DB in die Tabelle "WG" gespeichert.

Es wird außerdem ein WG_Code generiert den man seinen WG Mitbewohnern schicken kann. Mit diesem Code können andere User einer WG beitreten.

Nach dem Erstellen wird man zur UI 7 weitergeleitet.

INSERT INTO WG (id, name, adress, plz, city) VALUES (v, w, x, y, z)

6:

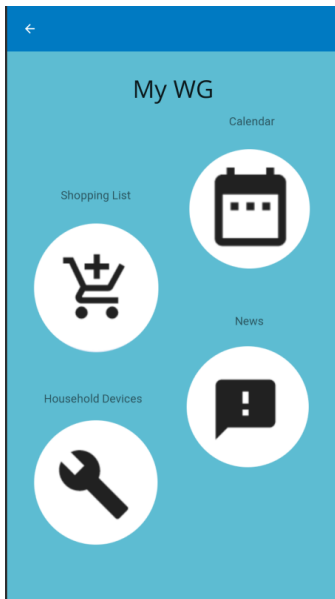
In der UI "Access WG" kann man mit einem vorher generierten Code einer WG beitreten.

Den WG_Code erhält der Ersteller einer WG. Mit diesem können andere User der WG beitreten. Dazu muss der Code in den EditText geschrieben werden. Dieser ist gleichzeitig die ID der WG. Mit dem Button "Input WG_Code" wird der Code mit der ID aus der DB in der Tabelle "WG" verglichen.

Ist der Code vorhanden wird man zu der UI 7 weitergeleitet.

SELECT id FROM WG

7:

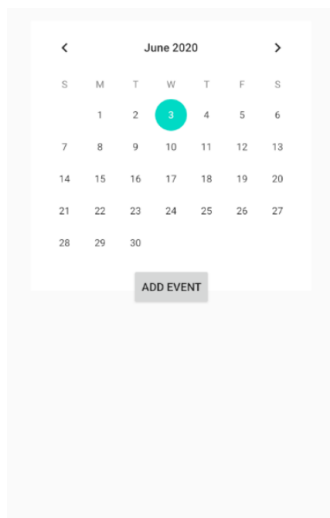


Die My-WG UI stellt ein Menü dar, welche auf die vier Funktionen für eine WG verweist.

Über den vier Icons ist jeweils ein Button (invisible). Der Kalender leitet zur 8. UI, der Einkaufswagen zu der 9. UI, der Schraubenschlüssel zur 10. UI und die Nachrichtenblase zur 11. UI.

Hintergrundfarbe: #5dbcd2

8:

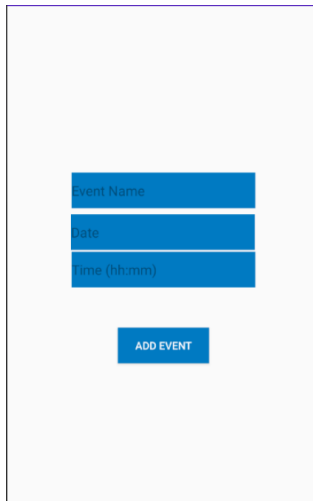


Diese UI stellt einen Google Kalender dar, welcher zur Planung und Organisation genutzt werden kann.

Der Kalender wird von Google importiert. Dieses Tool stellt eine gute Planung Möglichkeit von Terminen oder Pflichten dar.

Mit dem Button kann man ein neues Event hinzufügen

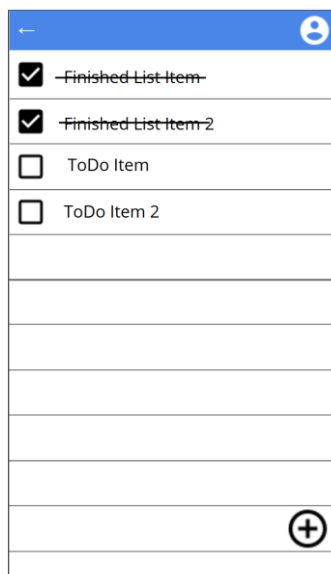
9:



Auf dieser UI kann man ein neues Event für den Kalender festlegen.

Das Event wird in der DB gespeichert und wird danach auf der Oberfläche 8 angezeigt.

10:



In der Einkaufsliste kann man neue Items hinzufügen oder andere als erledigt markieren.

In der Liste hat man auf der linken Seite CheckBox welchen man auf finished/ not finished stellen kann. (finished-1 not finished-0)

Die Namen der Items und die values (0/1) werden in der DB in der Tabelle "Shopping_List" gespeichert.

Mit dem Button unten rechts kann man ein Item hinzufügen bzw. in der DB speichern.

Hinzufügen von Items in Liste

ShoppingList.java Button zum Hinzufügen:

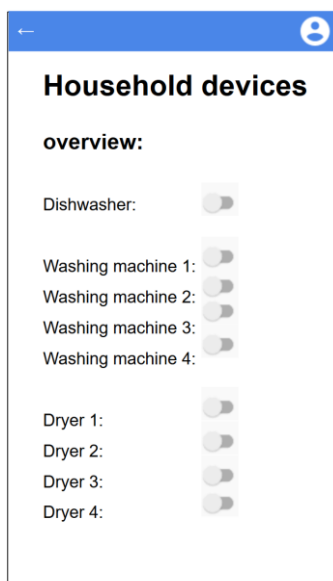
```
84         Button btnAddItem = (Button) findViewById(R.id.btn_addShoppingListItem);
85         btnAddItem.setOnClickListener(new View.OnClickListener() {
86             @Override
87             public void onClick(View view) {
88                 runMutation();
89             }
90         });
91     });
```

Funktion runMutation():

Hier wird der Text aus den zwei EditText-Feldern gezogen und in zwei String gespeichert. Die Strings werden dann in die Datenbank geschrieben.

```
109     public void runMutation(){
110         final String name = ((TextInputEditText) findViewById(R.id.ti_addItem)).getText().toString();
111         final String value = ((TextInputEditText) findViewById(R.id.ti_addValue)).getText().toString();
112         CreateShoppingListInput createShoppingListInput = CreateShoppingListInput.builder()
113             .wgID(wgCode)
114             .itemName(name)
115             .value(value)
116             .build();
117
118         mAWSAppSyncClient.mutate(CreateShoppingListMutation.builder().input(createShoppingListInput).build())
119             .enqueue(mutationCallback);
120
121         textInput.getText().clear();
122     }
123 }
```

11:



Diese UI gibt einen Überblick über alle Haushaltsgeräte und deren Aktivität in einer WG.

Zu jedem Gerät (linke Seite) gibt es einen Switch (rechte Seite), über welchen man einstellen kann ob das Gerät benutzt wird oder nicht.

Die Haushaltsgeräte sind in der DB in der Tabelle "House_Device" gespeichert. Dabei werden ihre ID, ihre Namen und deren Value (0/1) gespeichert. 0 steht dabei für "nicht in Benutzung" und 1 steht für "in Benutzung".

SELECT name, value FROM House_Device
INSERT INTO House_Device (id, name, value) VALUES (x, y, 0)

12:



Diese UI bildet eine News Funktion, welche es den Usern erlaubt eigene Neuigkeiten zu posten oder andere durchzulesen.

Die neuesten News sind immer ganz oben. News werden immer mit Titel und Text dargestellt. Mit dem Button unten rechts kann man eine neue Nachricht erstellen.

Diese Funktion kann für wichtige Nachrichten an alle (jemand zieht aus/ ein, jemand hat etwas zu verschenken), Meldungen von Mängeln oder für Organisatorische Mitteilungen genutzt werden.

Die News werden in der DB in der Tabelle "News" mit ihrer id, ihrem titel und dem dazugehörigen Text gespeichert.

SELECT title, text FROM News

INSERT INTO News (id, title, name) VALUES (x, y, z)