Entwicklung einer Android Food Tracking App in der Programmiersprache Kotlin

Johannes Franz & Normen Krug

Wintersemester 2017/2018

Vorgelegt bei Prof. Dr. Michael Stepping

Inhaltsverzeichnis

1 Einleitung

Ziel ist es eine Open Source App zu entwickeln, die einen Rückschluss von sich zugenommener Nahrung auf Symptome zu ermöglichen. Dabei helfen die in der App eingetragenen Datenpunkte und Fotos diese z.B. mit einer allergischen Reaktion zu verknüpfen. Zielgruppe sind Menschen, die wegen Erkrankungen aus ihren zu sich zugenommenen Speisen und Getränken Rückschlüsse auf ihr Wohlbefinden treffen wollen, um zukünftig solche Speisen zu meiden.

2 Motivation

Durch eigenen Bedarf motiviert entstand die Idee zu dieser App. Um aus wiederkehrenden Mahlzeiten eine Unverträglichkeit abzuleiten, stellt die App entsprechende Funktionen bereit. Bei dieser Arbeit war außerdem das Ziel möglichst viele neue Technologien zu verwenden. So wurde bei allen Möglichkeiten der schwierigere Weg gewählt, der jedoch zu neuen Erkenntnissen führte.
Beispiele dafür sind:
Kotlin, LaTeX, InAPp-SQLite,Dagger, Room, Recycler View, Model View Presenter, Spinner Menu in der Toolbar

3 Kotlin als Programmiersprache

Für die App wurde die Programmiersprache Kotlin verwendet, welche auf der Google IO im Mai 2017 als offizielle Sprache für Android eingeführt wurde. Kotlin hat einen besser lesbaren Syntax als Java. Wie bei Java wird auch bei Kotlin der Bytecode für die Java Virtual Machine übersetzt. Insgesamt ist Kotlin der Programmiersprache Swift 3 bzw 4 syntaktisch sehr nahe, was den Umgang als Entwickler mobiler Anwendungen zusätzlich erleichtert. Die Entscheidung dieses Sprache zu wählen war wieder davon getrieben etwas neues auszuprobieren.
Kotlin Code Beispiel:

```
mMapView.getMapAsync { mMap ->
        googleMap = mMap

        // For showing a move to my location button
        if ((activity as SecondMainActivity).checkLocationPermission()) {
                googleMap!!.isMyLocationEnabled = true
        }
        googleMap!!.setOnInfoWindowClickListener(this)
        var mdb = AppDatabase.getInMemoryDatabase(activity.applicationContext)
        val mealList = mdb.mealModel().getAllMeal()

        for (item in mealList) {
                var imageBitmap = MediaStore.Images.Media.getBitmap(view!!.
                    ↪ context.contentResolver, Uri.parse(item.imagePath))
                val time: GregorianCalendar = GregorianCalendar()
                time.timeInMillis = item.time
                val markerRepresentation = MarkerRepresentation(imageBitmap,
                    ↪ time, item.foodname)

                if (googleMap != null) {
                        var markerOpt = MarkerOptions().position(LatLng(item.lat
                            ↪ , item.lng)).title(item.foodname)
                                        .snippet(item.shortDescription)
                        var marker1 = googleMap!!.addMarker(markerOpt)
                        marker1.tag = markerRepresentation
                        markers.add(marker1)
                        googleMap!!.addMarker(markerOpt)?.tag =
                            ↪ markerRepresentation
                }
        }
}
```

4 Entwurfsphase

In der Entwurfsphase wurde keine Zeit verschwendet und somit wurde der Fokus auf Handskizzen gelegt. Der Zwischenschritt diese Skizzen in einem professionellen Entwurf nachzubauen sparte Zeit, so dass sich direkt mit der Programmierung beschäftigt werden konnte.

4.1 Main Screen

Der Main Screen hat während der Entwicklungsphase viele Veränderungen durchlaufen. So ist die TabBar von unten nach oben gelaufen. Die Datumsauswahl wurde durch eine sinnvollere Gruppierung ersetzt.

Abbildung 4.1: Main Screen

4.2 Hinzufügen eines Eintrags

Der *Add Screen* hat sich während der Entwicklung nur wenig verändert. Um es dem Nutzer einfach zu machen werden hier die aktuellen Werte einfach übernommen. Wichtige Felder lassen sich in einem *Edit Screen* bearbeiten. Dieser wurde um einzelne Elemente erweitert, die beim Hinzufügen noch keine Rolle spielen. Beispiele dafür sind die *SeekBar* und die *Effekt Description*.

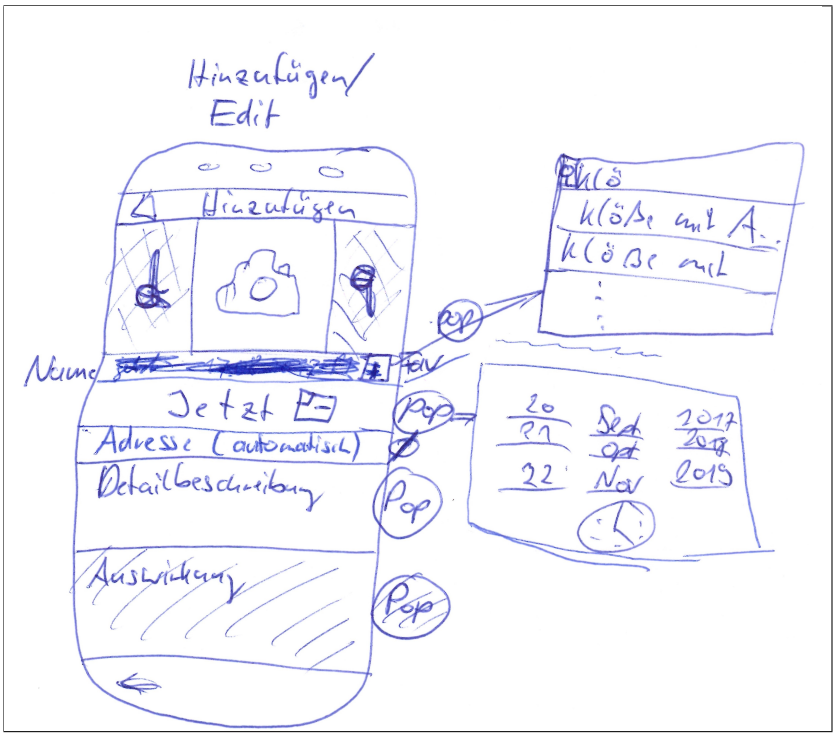

Abbildung 4.2: Hinzufügen eines Eintrags

4.3 Popover im Main Screen

Da Popover keinen guten Stil in der Benutzerführung darstellen, wurden sie durch bessere UI Features ersetzt.

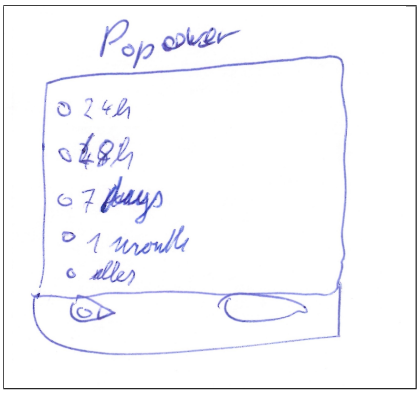

Abbildung 4.3: Popover Screen

4.4 Settings

Da die Komplexität der App im Hintergrund abläuft und Settings oft verwirrend sind wurde diese Idee letztendlich verworfen.
Das Drei-Punkte-Menü stellt Funktionen bereit, die in den Settings sonst nicht so einfach auffindbar wären.

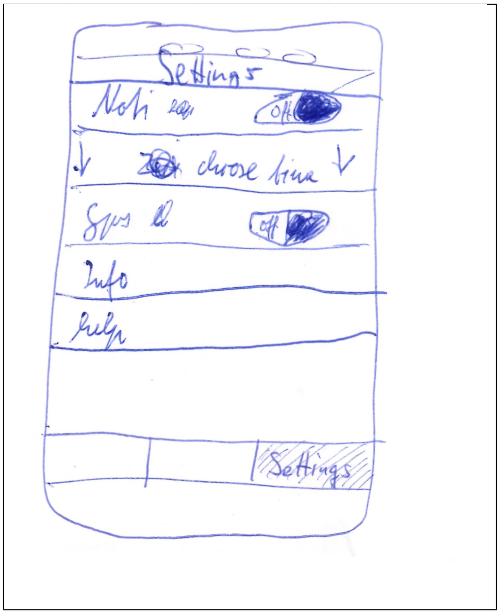

Abbildung 4.4: Settings Screen

4.5 Navigation Drawer Menü

Der auch unter "Hamburger Menü"bekannte *Navigation Drawer* war einer der ersten Ideen für die App. Viele Funktionen später als Tab Bar bzw. als Icons in der oberen rechten Ecke umgesetzt wurden, sind diese Buttons an die genannten Stellen hin verschoben worden. Im Sinne der Anwenderfreundlichkeit wurde sich am Ende gegen den Navigation Drawer entschieden.

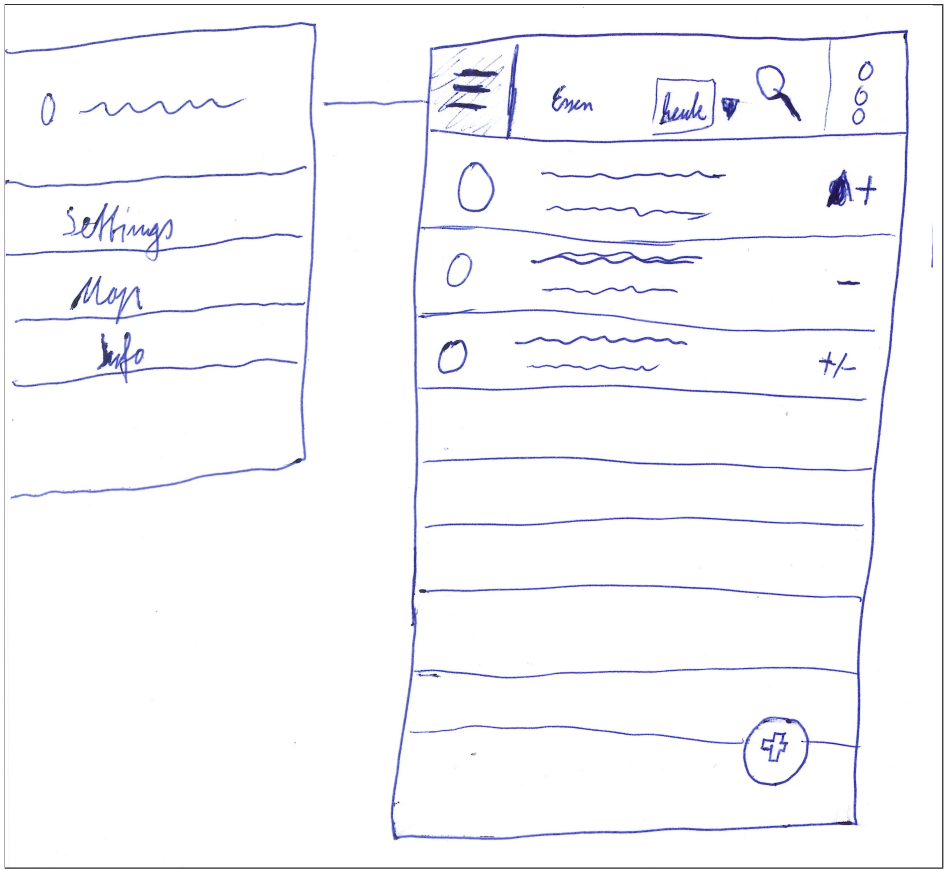

Abbildung 4.5: Navigation Drawer Menü Screen

4.6 Karten Screen

Der Karten Screen zeigt eine Google Map. Darin repräsentieren Pins die Standorte an denen Essens Einträge hinzugefügt wurden.

Abbildung 4.6: Karten Screen

5 Umsetzung

5.1 Datenbank

Für die Speicherung der Daten wurde sich für eine SQLite Datenbank entschieden. Das hat mehrere Vorteile. Zum einen lässt sich der Inhalt dieser Datenbank sehr einfach exportieren, um auf einem anderen Gerät wieder eingespielt werden zu können, zum anderen bietet eine Datenbankstruktur viele Möglichkeiten zur Datenanalyse mit schnellem Zugriff ohne wie bisher mit Files arbeiten zu müssen.

5.1.1 Room

Room wurde als Datenbank Abstraktionsschicht verwendet und bietet einfachen Datenbankzugriff. Die Room Bibliothek arbeitet mit Annotations. Mithilfe der Annotation und einer Hilfsklasse wird die Datenbank modeliert.

```
@Entity(tableName = "meal")
data class Meal(@ColumnInfo(name = "foodname") var foodname: String,
                  @ColumnInfo(name = "shortDescription") var shortDescription
                      ↪ : String,
                  @ColumnInfo(name= "longDescription") var longDescription:
                      ↪ String,
                  @ColumnInfo(name= "effectDescription") var
                      ↪ effectDescription: String,
                  @ColumnInfo(name = "effect")var effect: Int,
                  @ColumnInfo(name = "time")var time: Long,
                  @ColumnInfo(name = "lat") var lat: Double,
                  @ColumnInfo(name = "lng") var lng: Double,
                  @ColumnInfo(name = "addressline") var addressline: String,
                  @ColumnInfo(name="imagePath") var imagePath: String)
                  {

    @ColumnInfo(name = "id")
    @PrimaryKey(autoGenerate = true) var id: Int = 0
```

Durch verschiede Annotations und einer Datenzugriffsklasse, werden ßtored procedureërstellt. Diese haben den Vorteil das sie zu Compilezeit ausgewertet werden. Dadurch sind Fehler leichter aufspührbar und die Zugriffe werden optimiert.

```
@Dao interface MealDao{
        @Query("select␣*␣from␣meal")
        fun getAllMeal(): List<Meal>

        @Query("select␣*␣from␣meal␣where␣foodname␣=␣:foodname")
        fun findMealByName(foodname: String): Meal

        @Query("select␣*␣from␣meal␣where␣id␣=␣:id")
        fun findMealById(id: Int) : Meal

        @Query("select␣*␣from␣meal␣where␣time␣>=␣:time")
        fun findAllMealsAfter(time: Long) : List<Meal>

        @Query("select␣*␣from␣meal␣where␣:start␣<=␣time␣AND␣:end␣>=␣time")
        fun findAllMealsAfter(start: Long, end: Long) : List<Meal>

        @Query("select␣*␣from␣meal␣where␣addressLine␣==␣:addressline")
        fun findAllMealsWithAddress(addressline: String) : List<Meal>

        @Query("select␣*␣from␣meal␣where␣foodName␣=␣:foodName␣AND␣time␣=␣:time
            ↪ ")
        fun findAllMealsByNameAndTime(foodName: String, time: Long) : Meal

        @Query("select␣*␣from␣meal␣where␣lat␣=␣:lat␣AND␣lng␣=␣:lng")
        fun findMealByLatLng(lat: Double, lng: Double): Meal

        @Query("select␣*␣from␣meal␣where␣effect␣<=␣1")
        fun findMealyByEffect(): List<Meal>

        @Insert(onConflict = 1)
        fun insetMeal(meal: Meal)

        @Update
        fun updateMeal(meal: Meal)

        @Delete
        fun deleteMeal(meal: Meal)

}
```

Quelle: `https://developer.android.com/topic/libraries/architecture/room.html`

5.1.2 CSV Export

Das Ausgabeformat CSV (Comma Separated Values) beschreibt den Aufbau einer Textdatei. Darin sind alle übergebenen Spaltennamen eingetragen. Die Reihenfolge der Spaltennamen gibt die Reihenfolge der Werte, die darunter aufgelistet sind wieder. Jeder Werteeintrag ist mit einem Return vom anderen getrennt.
Die App unterstützt den .CSV Export. Der Export wird als "FoodtrackerData.csv"File im Download Ordner gespeichert. Dieser Ordner ist auf jedem Android Gerät einsehbar.

5.2 Google Maps

Für die Karte ist die Google Maps API eingesetzt worden. Darin wird für jeden Eintrag ein Pin in der Karte dargestellt. Der API Key wurde dankenswerterweise von Normen bereitgestellt.

5.3 MVP

MVP ist ein Entwurfsmuster in der Softwareentwicklung. Dieses unterteilt die Applikation in drei Bestandteile: Model, View, Presenter.

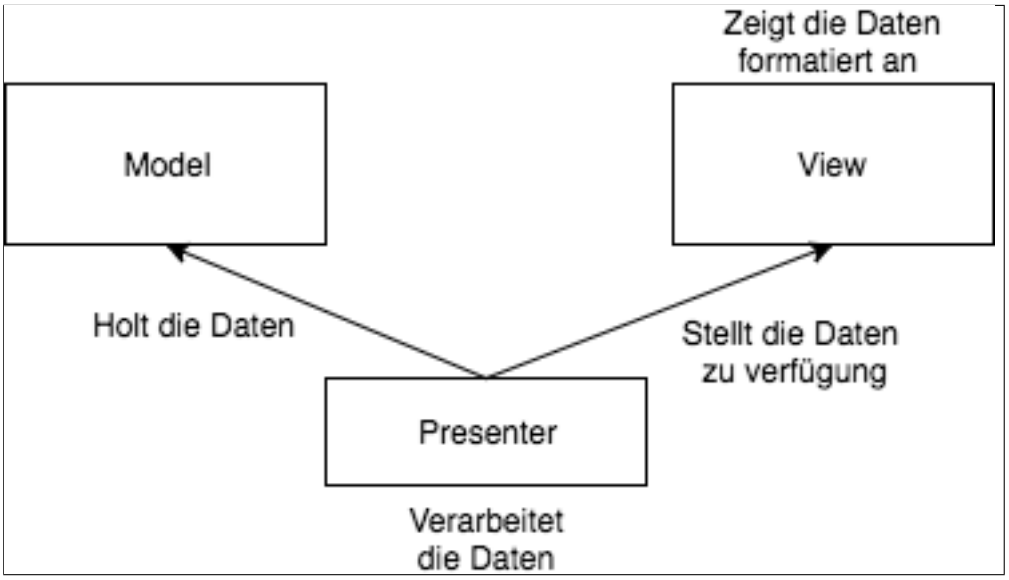

Abbildung 5.1: Model view presenter

Das Model hält die Daten die in der App verwendet werden. Sie strukturiert die Daten logisch. Der Presenter holt sich die Daten aus den Model und verarbeitet sie. Sie stellt die verarbeiten Daten bereit die die View anzeigt.

6 Herausforderungen

In jedem Projekt treten bei ansteigender Komplexität Probleme auf. Diese sind in diesem Abschnitt vermerkt.

- Aufwändige Integration von Dagger 2 die letztendlich verworfen werden musste. Mangelnde Dokumentation. Probleme beim Integrieren von bereits vorhandenen Projekten.
- Constraint Layouts bringen, wenn sie in einer NestedScrollView eingebunden sind gewisse Probleme. So liegt der zu bearbeitende Teil des Screens häufig außerhalb der Reichweite in der man diese bearbeiten kann.

7 Lessons learned

Dem Parkinsonschen Gesetz folgend wurde die Features während der vorhandenen Zeit umgesetzt. Da während der Appentwicklung die Stundenplan App der Hochschule betreut wurde, entstand gelegentlich ein Interessen- und Ressourcenkonflikt. Letztendlich wurden die Änderungen an der Schnittstelle leicht gewichtet behandelt, da diese einen Mehrwert für viele Hundert Studenten auf Android und iOS Seite mit sich brachten.

Zu Beginn des Projektes ist es schwer die Architektur im größeren Stil planen. Da Features nach und nach erweitert wurden, ist es nötig geworden die Architektur anzupassen, was zu Mehrarbeit geführt hat.

8 Weitere Arbeiten

In diesem Projektabschnitt konnten alle gesteckten Ziele erreicht werden. Da solch ein junges Projekt noch viele Möglichkeiten beinhaltet Funktionen zu verbessern und neue Funktionen einzuführen werden hier mögliche Punkte zur Anregung aufgelistet. Eine Erweiterung die sich um die Inhaltsstoffe der Mahlzeiten dreht und diese speziell auswertet, würde dem Nutzer zusätzliche Vorteile bieten. Diese Daten sind anonym genug um mittels Cloud Schnittstelle mit Werten anderer Personen verglichen werden kann Da das Projekt als Open Source Projekt auf GitHub zur Verfügung steht, sind forks und pull pull requests willkommen. Da das Projekt somit der Öffentlichkeit übergeben wurde, besteht durchaus ein Interesse unsererseits die App weiter zu entwickeln.

Abbildungsverzeichnis