Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektrotechnik, Medien und Informatik

Studiengang Medieninformatik

App-Programmierung Studienarbeit
Wintersemester 2022/23

von

Philip **Bartmann**

Dokumentation der Entwicklung und Implementierung einer App zur Visualisierung von Daten der Parkhäuser Ambergs mithilfe des Frameworks react-native

Inhaltsverzeichnis

Kapitel 1

Problemstellung

Ziel dieser Studienarbeit war, das vorhandene Parkleitsystem Ambergs zu nutzen, um eine App zu entwickeln, welche die Daten der Parkhäuser übersichtlich und benutzerfreundlich darstellt. Hierbei wurden die Daten als XML-Datei live eine API der Stadt Amberg zur Verfügung gestellt. Über `https://parken.amberg.de/wp-content/uploads/pls/pls.xml` können diese abgefragt werden. Die App soll es dem Nutzer ermöglichen alle Daten der Parkhäuser einsehen zu können, um eine fundierte Entscheidung zu treffen, wo er parken will. Zur Entscheidungsfindung werden von der Stadt Amberg Daten für freie Parkplätze, die stündlichen Kosten des Parkens, der Trend, also ob sich das Parkhaus gerade füllt oder leert, die Koordinaten der einzelnen Parkhäuser auf der Karte und weiteres bereitgestellt. Über die Koordinaten werden dem Nutzer Markierungen auf einer Karte gegeben, bei denen er die Daten der Parkhäuser auslesen kann. Diese Markierungen werden auch als Liste dargestellt, um eine Alternative Orientierungsmöglichkeit zu bieten. Auch werden Benachrichtigungen erzeugt, falls der Nutzer in die Nähe eines Parkhauses kommt, damit naheliegende Parkhäuser einfacher betrachtet werden können. Zudem soll eine simple Navigation möglich sein, um den Nutzer nach seiner Entscheidung zu dem gewünschten Parkhaus zu leiten. Zuletzt soll die App sinnvolle Einstellungsmöglichkeiten bieten, welche der Nutzer verwenden kann, um die App zu konfigurieren.

Zur Entwicklung der App wurde das Framework react-native verwendet, womit native Apps entwickelt werden können [1]. Dies bedeutet, dass die Apps speziell für eine bestimmte Plattform, wie Android oder iOS, entwickelt werden. Der Vorteil bei react-native ist, dass hierfür nicht für jede Plattform eigener Code geschrieben werden muss, sondern Webtechnologien verwendet werden können und sich react-native unter der Haube um die plattformspezifische Umsetzung kümmert.

Im Folgenden wird beschrieben, wie die App aufgebaut ist und wie die einzelnen Funktionen umgesetzt wurden. Zudem werden auch zusätzliche Funktionen vorgestellt, welche bei der Entwicklung als sinnvoll erschienen und deshalb für eine bessere Benutzerfreundlichkeit auch implementiert wurden.

Kapitel 2

Vorbereitung und Vorgehensweise

Bevor die App entwickelt werden konnte waren einige Vorbereitungen nötig. Zuerst wurde eine Umgebung benötigt, in der die Entwicklung stattfinden konnte. Dafür wurde zuerst ein Template installiert, welches von Expo bereitgestellt wurde[2]. Expo ist ein weiteres Framework, welches die Arbeit mit react-native erleichtert. Durch Expo lassen sich Pakete für die Apps installieren, Entwicklungsserver starten und vieles mehr. Es wurde sich hier für ein Template mit TypeScript entschieden, da es sich dabei um typsicheres JavaScript handelt, wobei viele Fehler schon in der Entwicklung gefunden werden können.

Um die App testen zu können wurde zudem ein Smartphone benötigt. Ein reales Smartphone ist zwar möglich, jedoch kann hier nicht die Position des Nutzers nach Belieben verändert werden, was sehr unvorteilhaft für die Entwicklung einer App ist, welche bestimmte Ereignisse nur an bestimmten Positionen ermöglicht. Also wurde hier über Android Studio ein Android-Emulator installiert. Es wurde sich für einen Emulator des Pixel 3a XL Smartphones entschieden, da hier ein ausreichend großer Bildschirm vorhanden ist.

Ein weiterer Vorbereitungsschritt bestand darin, die statischen Daten der Parkhäuser zu finden. Durch die API der Stadt Amberg können nur Daten abgefragt werden, die sich ändern, also dynamische Daten, wie freie Parkplätze oder der Trend mit dem sich das Parkhaus füllt oder leert. Hier fehlen jedoch Daten wie die Koordinaten des Parkhauses oder die stündlichen Preise für das Parken diese Daten wurden zuerst unter der URL `https://www.amberg.de/leben-in-amberg/mobilitaet/parken` bereitgestellt, während der Entwicklung der App wurden die Daten jedoch auf die Webseite `https://www.amberg.de/parken` verschoben. Hier konnten die Koordinaten der einzelnen Parkhäuser über eine Webseiteninterne OpenStreetMap-Karte ausgelesen werden, wobei die Koordinaten für die spätere Navigation ein wenig angepasst wurden, sodass diese genau auf den Eingängen zu den einzelnen Parkhäusern liegen. Die Preise und Öffnungszeiten wurden dann durch die weiteren Informationen auf der Webseite und eine von der Stadt Amberg bereitgestellte umfassende Übersicht über die Preise als PDF-Datei zusammengestellt. Die PDF-Datei ist unter dem Link `https://www.amberg.de/fileadmin/Mobilitaet/Parkpreise_Uebersicht.pdf` verfügbar. Zudem wurden

noch nützliche Zusatzinformationen übernommen, wie zum Beispiel, dass das Parken für Gäste des Kurfürstenbades auf dem Parkplatz des Kurfürstenbades und dem Parkhaus Kurfürstengarage frei ist.

Nachdem alle Daten gesammelt waren, mussten diese noch persistent gespeichert werden. Hier gab es zwei Möglichkeiten: Die erste ist die Speicherung in einer Datenbank, hier ist SQLite unter react-native verfügbar. Die zweite und modernere Variante ist das Paket async-storage. Zuerst wurde versucht eine Datenbank aufzusetzen über das expo-sqlite Paket, welches von wieder Expo bereitgestellt wird [3]. Hier mussten SQL-Abfragen benutzt werden, um die Daten zu speichern und auszulesen. Da dies zu Unmengen an Code führte, wurde die zweite Möglichkeit versucht [4]. Das async-storage Paket speichert Daten als Schlüssel-Wert Paare, das heißt die Daten bekommen bei der Speicherung einen Schlüssel als String zugewiesen, mit dem sie danach identifiziert und wieder ausgelesen werden können. Die Daten müssen dabei auch als String gespeichert werden. Der Vorteil bei dieser Methode ist, dass JavaScript-Objekte einfach in einen String konvertiert und beim Auslesen wieder in ein JavaScript-Objekt rekonvertiert werden können. Auch sind keine Datenbankstrukturen nötig, wie Tabellen und Beziehungen zwischen diesen.

In Listing 2.1 ist das Format der statischen Daten am Beispiel des Parkhauses am Ziegeltor zu sehen.

```
id: 4,
name: "Am Ziegeltor",
coords: {
  latitude: 49.44864,
  longitude: 11.85684,
},
numberOfParkingSpots: 200,
pricingDay: {
  firstHour: 1,
  followingHours: 0.5,
  maxPrice: 5,
  startHours: "08:00",
  endHour: "19:00",
},
pricingNight: {
  firstHour: 0.5,
  followingHours: 0.5,
  maxPrice: 1.5,
  startHours: "19:00",
  endHour: "08:00",
},
openingHours: {
  startHour: "00:00",
  endHour: "24:00",
},
additionalInformation: "FLEXI-Ticket moeglich",
favorite: false,
```

Listing 2.1: Format der statischen Daten der Parkhäuser am Beispiel des Parkhauses am Ziegeltor

In Listing 2.2 dagegen das Format der dynamischen Daten desselben Parkhauses.

```
1 "Aktuell": 36,
2 "Frei": 164,
3 "Gesamt": 200,
4 "Geschlossen": 0,
5 "ID": 4,
6 "Name": "Am Ziegeltor",
7 "Status": "OK",
8 "Trend": -1
```

Listing 2.2: Format der dynamischen Daten der Parkhäuser am Beispiel des Parkhauses am Ziegeltor

Nachdem diese Daten nun gespeichert werden konnten, war ein kontinuierliches Abfragen der API nötig, um immer die aktuellsten Daten zu besitzen. Hierfür wurde eine Funktionskomponente in react-native erstellt, welche im Ordner `ParkingAPI` in der Datei `useAPICall.ts` zu finden ist und bei Aufruf ein Intervall erzeugt. Dieses Intervall ruft nach einer bestimmten Zeit eine Funktion auf, welche die Daten der API anfragt. Um nicht zu viele Anfragen zu tätigen, aber trotzdem aktuelle Daten zu besitzen, ruft das Intervall die Funktion alle 60 Sekunden auf. Die Funktion zur Anfrage der API achtet zudem auch darauf, ob eine Internetverbindung besteht oder nicht. Falls keine Internetverbindung besteht, wird automatisch durch die fetch-Funktion, welche benutzt wird, um die API abzufragen, ein Fehler geworfen. Bei einem solchen Fehler werden die alten Daten, die noch im async-storage bestehen, verwendet. Falls keine solchen Daten bestehen, wird der Nutzer benachrichtigt, seine Internetverbindung zu prüfen. Falls jedoch die Daten aus der API abgefragt werden können, überschreiben die neuen Daten die alten im async-storage und die neuesten Daten stehen der App damit zur Verfügung. Über jedes dieser Ereignisse wird der Nutzer zudem über Benachrichtigungen informiert.

Da die Vorbereitung damit abgeschlossen war, konnte nun begonnen werden die App zu entwickeln. Hier stellte sich die Frage, welche Funktionalität zuerst entwickelt werden sollte. Es wurde sich zuerst für die Karte entschieden, da diese zum größten Teil durch Einbinden eines Expo-Pakets implementiert werden konnte. Für die Karte wurden zudem auch gleich die Konfigurationsmöglichkeiten umgesetzt. Danach wurde eine Liste entwickelt, mit welcher der Nutzer die Parkhäuser übersichtlich und mit Detail-Informationen betrachten konnte. Hier wurden auch wieder Konfigurationsmöglichkeiten implementiert. Zuletzt wurden die zusätzlichen sinnvollen Funktionen entwickelt, welche die App benutzerfreundlicher machen sollten.

Kapitel 3

Implementierung der Karte

Die Karte und alle Zusatzfunktionalitäten sind im Ordner `Map` zu finden. Ein erster Prototyp der Karte war sehr schnell implementiert. Hierfür musste nur das Paket react-native-maps installiert werden. Dieses Paket benutzt auf Android-Systemen die Google Maps API und auf iOS-Systemen die Apple Maps API, um eine Karte in der App anzeigen zu können. Wenn dieses Paket mit Expo installiert wird, sind sogar keine API-Keys nötig, um die Karten anzeigen und nutzen zu können. Um nun auch den Standort des Nutzers in der Karte zeigen zu können, wird das Expo-Paket expo-location benötigt [6]. Dieses Paket kann, wenn der Nutzer die Erlaubnis gibt, den Standort des Nutzers herausfinden und in die Karte einzeichnen. Wenn der Nutzer seine Erlaubnis nicht gibt, wird dies auch wahrgenommen und die Funktionalität der App ist dadurch eingeschränkt, da dann nicht mehr herausgefunden werden kann, wie nah der Nutzer einem Parkhaus ist. Um die Parkhäuser in die Karte einzuzeichnen wurde eine Liste aus den statischen Daten erzeugt, welche an das react-native-maps Paket übergeben wird.

Abbildung 3.1: Ansicht der fertigen App mit der Karte und eingezeichneten Markern

Diese Markierungen der Parkhäuser wurden danach noch mit Icons versehen, welche den Trend anzeigen. Diese Icons stammen aus dem react-native-vector-icons Paket [8]. Die Icons aus diesem Paket werden in der gesamten App verwendet. Ein grüner Marker mit einem Pfeil nach unten bedeutet, dass sich das Parkhaus gerade leert.

Ein roter Marker mit einem Pfeil nach oben heißt, dass sich das Parkhaus füllt. Ein grauer Marker bedeutet, dass das Parkhaus einen gleichbleibenden Trend besitzt. Somit kann der Nutzer schon beim Starten der App sofort Informationen zu den Parkhäusern auslesen. Diese Marker mit der Karte sind in Abbildung 3.1 zu sehen. Hier ist der Nutzer als kleiner blauer Punkt mit einem Pfeil in Blickrichtung bei der OTH eingezeichnet.

Durch Anklicken der Marker ist es auch möglich, nähere Informationen zu den einzelnen Parkhäusern zu bekommen. Diese Ansicht ist in Abbildung 3.2 zur Theatergarage zu sehen. Hier wird angezeigt, wie viele Plätze in diesem Parkhaus noch frei sind. Zur besseren Benutzerfreundlichkeit wird dies noch mit einem Fortschritts-Kreis visualisiert, welcher aus dem react-native-progress Paket stammt [7]. Dieser zeigt den Füllstand des Parkhauses in Prozent an. Je größer der Prozentwert, desto mehr Parkplätze sind frei. Über diesem Kreis sind zwei Icons zu sehen. Der blaue Pfeil links bedeutet die Navigation. Durch Klicken auf dieses Icon wird der Nutzer zu diesem bestimmten Parkhaus navigiert, was später erläutert wird. Das rote Herz links bedeutet, dass dieses Parkhaus vom Nutzer zu den Favoriten hinzugefügt wurde. Wenn das Parkhaus nicht zu den Favoriten gehört, ist das Herz nicht ausgefüllt. Durch einen Klick auf dieses Icon kann der Nutzer zudem das Parkhaus zu den Favoriten hinzufügen oder wieder entfernen.

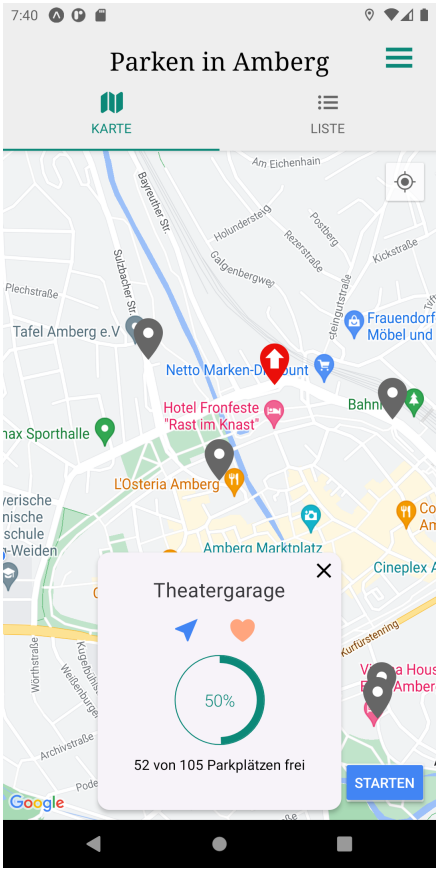

Abbildung 3.2: Detailansicht zum Parkhaus Theatergarage in der Karte

Nachdem die Karte fertig implementiert war, konnten die Anzeigen erstellt werden, dass sich der Nutzer einem Parkhaus nähert. Dies wurde zuerst über Geofencing versucht zu lösen. Geofencing bedeutet, dass die App kreisförmige Bereiche, welche vom Nutzer meist nicht sichtbar sind, um Koordinaten aufspannt. Bei Betreten und Verlassen eines dieser Bereiche wird ein Event mit den Informationen zu dem Bereich, also hier dem Parkhaus, gefeuert. Im expo-location Paket ist standardmäßig Geofencing enthalten. Dies wurde versucht einzufügen, jedoch kam es hierbei zu Problemen. Wenn der Nutzer in mehreren Geofences ist, kann es passieren, dass das Betreten eines anderen Geofences doppelt signalisiert ist. Wie in `https://github.com/expo/expo/issues/6283` zu sehen, besteht dieses Problem immer noch und es scheint keinen Fix hierfür zu geben. Deshalb wurde ein eigenes Geofencing implementiert, welches im Ordner `Geofencing` zu finden ist. Dafür musste die Luftlinienentfernung vom

Nutzer zu jedem Parkhaus berechnet werden, ob er in der Nähe eines Parkhauses ist. Zur Berechnung der Distanz wurde die bekannte Haversine Formel verwendet [9]. Diese kann über die Umwandlung der Winkelkoordinaten Latitude und Longitude ins Bogenmaß den Abstand zwischen zwei Punkten auf einer Kugel sehr genau berechnen.

Um nun den Abstand zu den Parkhäusern zu berechnen, wurde die Position des Nutzers benötigt. Dafür existiert wieder eine Funktion in expo-location, nämlich `startLocationUpdatesAsync`. Diese Funktion schickt kontinuierlich die aktuellen Koordinaten des Nutzers an einen Task, der vorher definiert wurde und im Hintergrund läuft. Der Task nimmt die Koordinaten an und übergibt diese an eine Funktion, welche den Abstand dieser Koordinaten zu jedem Parkhaus berechnet. Falls der Abstand kleiner als 200 Meter ist, wird der Nutzer als „in der Nähe des Parkhauses" angesehen. Falls der Nutzer vorher weiter entfernt als 200 Meter vom Parkhaus war, also außerhalb des „Geofences", erscheint ein Meldung, welchem Parkhaus der Nutzer sich gerade nähert und wie viele freie Parkplätze in diesem aktuell sind. Diese Meldung wird auch über ein text-to-speech Paket vorgelesen [10]. Die graphische Anzeige der Meldung ist in Abbildung 3.3 ersichtlich. In welchen Geofences der Nutzer gerade ist wird in einer Liste festgehalten, die später benötigt wird.

Abbildung 3.3: Anzeige, dass sich der Nutzer dem Parkhaus ACC nähert, welches 226 freie Parkplätze hat

Zuletzt wurde die Karte noch mit zwei Einstellungsmöglichkeiten konfigurierbar gemacht. Diese sind über das Burger-Menü rechts oben aufrufbar. Bei Klicken auf das Menü öffnet sich eine Liste mit zwei Einträgen, wie in Abbildung 3.4 sichtbar. Der erste Eintrag steuert, ob die Meldung, dass ein Geofence betreten wurde vorgelesen werden soll oder nicht. Wenn das Icon des Lautsprechers durchgestrichen ist, wird die Vorlese-Funktion ausgeschaltet. Standardmäßig ist diese Funktion angeschaltet. Der zweite Eintrag

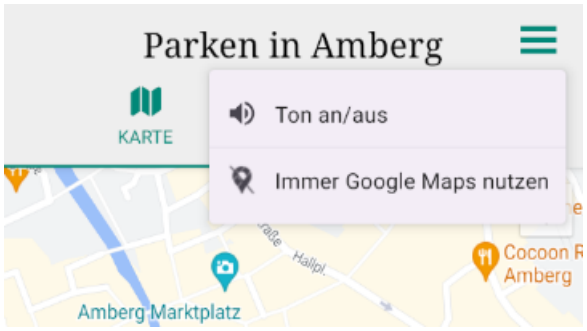

Abbildung 3.4: Liste an Einstellungen der Karte, welche über ein Burger-Menü verfügbar ist

gibt an, ob bei der Navigation ausschließlich zu Google Maps weitergeleitet werden, oder ob die interne Navigation der App, wenn möglich, genutzt werden soll. Auf diese Navigation wird in Kapitel 5 weiter eingegangen. Diese Einstellung ist standardmäßig ausgeschaltet, damit die interne Navigation bevorzugt wird. Die Werte beider Einstellungen werden zudem ebenfalls in async-storage gespeichert, damit der Nutzer die App nicht bei jedem Öffnen neu konfigurieren muss.

Damit ist die Implementierung der Karte abgeschlossen. Im Folgenden wird die Funktionalität und Umsetzung der Liste beschrieben.

Kapitel 4

Implementierung der Liste

Kapitel 5

Umsetzung einer Pseudo-Navigation

Kapitel 6

Zusätzliche Funktionen der App

- Tab-Navigation zwischen Liste und Karte
- Navigations-Knopf unten rechts mit Auswahl, ob nähestes Parkhaus oder Parkhaus in Geofence
- Genaue Preisdaten als Tabelle in Details zu Parkhaus
- Navigation von überall in der App zu jedem Parkhaus möglich

Kapitel 7

Ausblick und Fazit

- Pseudo-Navigation verbessern, vielleicht auch durch Directions API von Google => kostet Geld
- Mehr Einstellungsmöglichkeiten (Darkmode)
- Genauere Berechnung des nähesten Parkhauses => nicht über Luftlinie sondern über Weg => Directions API nötig

Literaturverzeichnis

[1] [1] „React Native · Learn once, write anywhere“. https://reactnative.dev/ (zugegriffen 11. Januar 2023).

[2] „Expo“. https://expo.dev/ (zugegriffen 13. Januar 2023).

[3] „SQLite“, Expo Documentation. https://docs.expo.dev/versions/latest/sdk/sqlite (zugegriffen 11. Dezember 2022).

[4] „React Native Async Storage“. AsyncStorage, 13. Januar 2023. Zugegriffen: 13. Januar 2023. [Online]. Verfügbar unter: https://github.com/react-native-async-storage/async-storage

[5] „react-native-maps“. react-native-maps, 13. Januar 2023. Zugegriffen: 13. Januar 2023. [Online]. Verfügbar unter: https://github.com/react-native-maps/react-native-maps

[6] „Location - Expo Documentation“. https://docs.expo.dev/versions/latest/sdk/location/ (zugegriffen 11. Dezember 2022).

[7] J. Arvidsson, „react-native-progress“. 6. Januar 2023. Zugegriffen: 8. Januar 2023. [Online]. Verfügbar unter: https://github.com/oblador/react-native-progress

[8] J. Arvidsson, „Multi-style fonts“. 7. Januar 2023. Zugegriffen: 7. Januar 2023. [Online]. Verfügbar unter: https://github.com/oblador/react-native-vector-icons

[9] „Distance on a sphere: The Haversine Formula“, Esri Community, 5. Oktober 2017. https://community.esri.com/t5/coordinate-reference-systems-blog/distance-on-a-sphere-the-haversine-formula/ba-p/902128 (zugegriffen 7. Januar 2023).

[10] „Speech“, Expo Documentation. https://docs.expo.dev/versions/latest/sdk/speech (zugegriffen 13. Januar 2023).

Abbildungsverzeichnis

Listings