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

Studiengang Medieninformatik

Studienarbeit App-Programmierung WiSe 2022/23

von

Annika **Stadelmann**

Entwicklung eines Parkleitsystems

Bearbeitungszeitraum:	von	22. November 2022
	bis	17. Januar 2023

Inhaltsverzeichnis

Kapitel 1

Einleitung

Im Rahmen der Studienarbeit sollte eine App als Parkleitsystem für die neun Parkmöglichkeiten um den Altstadtring in Amberg entwickelt werden. Neben der Anzeige der Parkflächen auf einer Karte, ist es wichtig, auch detaillierte Informationen anzuzeigen. Dabei handelte es sich um Kosten pro Stunde, den Namen des Parkhauses oder Parkplatzes, die Anzahl der verfügbaren, belegten und gesamten Parkplätze, sowie ein aufsteigender, gleichbleibender oder fallender Trend. Des Weiteren muss dem Nutzer auch mitgeteilt werden, ob und wann das Parkhaus geöffnet ist und ob es überhaupt befahren werden kann.

Neben der Anzeige der Daten sollte es auch die Möglichkeit geben, eine Navigation zur Parkmöglichkeit zu starten, sobald der Nutzer sich in einem bestimmten Radius um die Örtlichkeit befindet. Wenn das sogenannte Geofence betreten wird, soll neben der Navigationsmöglichkeit als Toast auch noch Text als Sprachausgabe ausgegeben werden, um den Nutzer darauf hinzuweisen, welches Parkhaus am nächsten ist und befahren werden könnte. Damit der Nutzer nun nicht durch unpassende Sprachausgabe in unangenehme Situationen gebracht wird, sollte die Möglichkeit gegeben sein, den Ton auszuschalten. Außerdem sollen die Parkmöglichkeiten auch als Liste angezeigt und favorisiert werden können.

Um die aktuellen Parkhausdaten zu bekommen, soll eine API verwendet werden, welche die Daten bereitstellt und regelmäßig aktualisiert. Die Daten sind im XML-Format und unter folgendem Link zu finden: `https://parken.amberg.de/wp-content/uploads/pls/pls.xml`

Im Folgenden werden die Architektur und die Implementierung vorgestellt und näher beschrieben. Wie die zu entwickelnde Anwendung zu starten ist, wird im letzten Kapitel beschrieben.

Kapitel 2

Architektur

Die App sollte mit dem JavaScript-Framework react-native programmiert werden. Dieses Framework wird eingesetzt, wenn die App sowohl auf mobilen IOS-, als auch Android-Geräten funktionieren soll. Auf diese Art wird Code geschrieben werden, welcher für beide Plattformen verwendet werden kann [1]. Als Programmiersprache wird TypeScript eingesetzt. TypeScript ist wie JavaScript, nur mit Typensicherheit. So können keine Verwirrungen bezüglich des Typs einer Variable aufkommen [2]. Zusammen mit TypeScript und dem Framework react-native wird zusätzlich noch Expo genutzt. Expo ist ein Framework, welches Werkzeuge zur Unterstützung der Entwicklung einer react-native-App bietet [3].

2.1 Mockups und Komponenten

Um die Anforderungen nicht völlig durcheinander in Quell-Code zu „pressen", werden vorher Prototypen entworfen, welche das Design der App darstellen sollen. Dies schafft Struktur und ermöglicht es, systematisch vorzugehen. Ziel der Mockups ist es, ein Design zu schaffen, welches möglichst einfach und verständlich alle relevanten Informationen auf einen Blick zeigt und es dem Nutzer ermöglicht, sich ebenfalls weitere Details anzeigen zu lassen, wenn dies gewünscht ist. Anhand dieser Mockups gelingt es nun, Komponenten herauszuarbeiten, welche anschließend in Quell-Code umgewandelt werden sollen.

Da die Implementierung mit react-native erfolgt und statt Klassen Komponenten für die Benutzeroberfläche verwendet werden, werden auch die Mockups in Komponenten aufgeteilt. Die Komponente, welche alle anderen beinhalten soll, ist die ‚App'. Hier findet sich auch ein Button oben rechts, der in Abbildung 2.1(a) zu sehen ist. Eine weitere Komponente bildet die Karte, welche sich über den größten Teil des Bildschirms erstreckt. Wird auf einen Marker geklickt, der eine Parkmöglichkeit anzeigt, so öffnet sich die Komponente der Beschreibung. Diese ist beispielsweise in Abbildung 2.1(b) zu sehen. Hier sollen Kacheln angezeigt werden, welche die einzelnen Informationen liefern. Außerdem soll hier auch der Button angezeigt werden, welcher die Navigation startet.

(a) Dieser Entwurf zeigt die Komponente ‚App' mit dem einem Button oben rechts in der Ecke. Außerdem befindet sich hier auch die Komponente mit der Karte.

(b) Zusätzlich zur Karten-Kompnente in der ‚App' ist hier auch noch die Komponente der Beschreibung zu sehen, welche sich beim Klick auf eine Parkmöglichkeit in der Karte öffnen soll. Die Kacheln zeigen die relevanten Informationen an.

Abbildung 2.1: Diese beiden Entwürfe zeigen jeweils die ‚App'-Komponente und die darin befindliche Karten-Komponente. In der rechten Abbildung öffnet sich zusätzlich die Beschreibung-Komponente, welche die Karte verkleinert. (Quelle: Eigens gezeichnete Mockups)

Wird auf das Zahnrad in Abbildung 2.1(a) in der Komponente ‚App' geklickt, so soll sich eine Komponente mit den Einstellungen, wie in Abbildung 2.2(a) zu sehen, öffnen, in welcher der Ton an- und ausgeschaltet werden kann oder auch die Parkmöglichkeiten als Liste angezeigt werden können. Um die Parkmöglichkeiten als Liste zu sehen, wird eine weitere Komponente benötigt, wie sie in Abbildung 2.2(b) erkennbar ist. Die Elemente der Liste werden mit den Favoriten zuerst und dann nach dem Alphabet sortiert.

(a) Die Komponente, welche das Einstellungsmenü beinhaltet. Ein Klick auf den Ton schaltet diesen entweder ein oder aus. Die Kachel darunter zeigt bei einem Klick die Parkmöglichkeiten als Liste an.

(b) Hier befindet sich die Liste der Parkmöglichkeiten, welche alphabetisch sortiert ist. Die Favoriten finden sich hier ganz oben.

Abbildung 2.2: Auf den Entwürfen ist links das Einstellungsmenü zu sehen. Wird auf ‚Parkplätze' oder in der Beschreibung-Komponente auf ‚mehr' geklickt, so wird zur Ansicht der Details einer Parkmöglichkeit, was auf der rechten Seite zu sehen ist, navigiert. (Quelle: Eigens gezeichnete Mockups)

Wird in der Liste auf eine Parkmöglichkeit oder in der Beschreibung einer einzelnen auf die Kachel mit der Beschriftung "mehr"geklickt, öffnet sich eine Detailansicht, welche alle vorhandenen Informationen des Parkhauses anzeigt. Zu sehen ist dies in Abbildung 2.3 Diese Übersicht bildet die letzte Komponente.

Abbildung 2.3: Diese Komponente bildet die Detailsansicht der jeweiligen Parkmöglichkeit. Zu erreichen ist diese über die Beschreibung- oder die Einstellung-Komponente. (Quelle: Eigens erstellte Übersicht.)

In Summe sind also zunächst fünf Komponenten geplant, welche in der Komponente ‚App' zu finden sind. Ob dies tatsächlich so umsetzbar ist, wird sich in der Implementierung zeigen.

2.2 Datenbank

Um die Daten nun auch sauber vom Code zu trennen, müssen Datenbanktabellen angelegt werden. Die erste Tabelle ‚parkingarea' wird mit den Daten der Parkmöglichkeiten gefüllt. Hierzu bekommt jedes der neun Parkmöglichkeiten eine ID zur Identifikation. Weitere Spalten in der Tabelle beinhalten:

Name
Der Name des Parkhauses oder Parkplatzes.

Adresse
Die Straße oder mehrere Straßen, über die die Parkmöglichkeit angefahren werden kann.

Öffnungszeiten
Wie lange die jeweilige Parkmöglichkeit geöffnet hat.

Preis pro Stunde
Wie viel pro Stunde bezahlt werden muss, um sein Fahrzeug zu parken.

Einfahrtshöhe
Wie hoch das Fahrzeug maximal sein darf, um in das Parkhaus zu fahren. Im vorliegenden Fall gibt es auch Parkplätze ohne maximale Einfahrtshöhe.

Favorit
Dieser Wert gibt an, ob es sich um eine favorisiertes Parkmöglichkeit handelt oder nicht. In dem Fall gibt der Wert 1 eine favorisierte Parkmöglichkeit an, wohingegen 0 anzeigt, dass eine Parkmöglichkeit nicht favorisiert ist.

Latitude
Latitude ist die geographische Breite oder auch der Breitengrad. In Zusammenspiel mit dem Längengrad ergibt sich die exakte Position der Parkmöglichkeit auf der Erde.

Longitude
Die geographische Länge oder auch Längengrad.

In Abbildung 2.4 sind alle Spalten der Tabelle ‚parkingarea' mit ihren zugehörigen Datentypen zu erkennen.

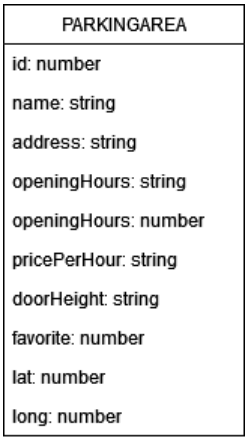

Abbildung 2.4: Die Spalten und ihre Datentypen der Datenbanktabelle ‚parkingarea'. Hier werden in Kapitel 3 alle Parkmöglichkeiten eingetragen. (Quelle: Eigens erstellte Übersicht.)

Die zweite Datenbanktabelle erhielt den Namen ‚parkingareadetails' und sollte mit den Daten der API gefüllt werden. Neben einer einzigartigen ID finden sich hier folgende Spalten:

ID des jeweiligen Parkhauses
Um die Details der Parkmöglichkeiten denen der anderen Tabelle zuordnen zu können, wird zu jedem Datensatz die ID der Parkmöglichkeit gespeichert, zu der die Informationen gehören.

Anzahl der Parkplätze
Die Anzahl der Gesamtheit aller Parkplätze der jeweiligen Parkmöglichkeit, unabhängig davon, ob diese belegt oder frei sind.

Anzahl belegter Parkplätze
Die Anzahl der Parkplätze, die aktuell belegt sind.

Anzahl freier Parkplätze
Die Anzahl der Parkplätze, auf denen noch geparkt werden kann.

Trend
Der Trend gibt an, ob die Anzahl belegter Parkplätze steigt, sinkt oder gleich bleibt. Wenn beispielsweise viele Menschen auf einmal in die Parkmöglichkeit einfahren, so steigt der Trend. Der Trend kann die Werte 0 für ‚gleichbleibend', 1 für ‚steigend' oder -1 für ‚fallend' annehmen.

Status
Der Status, in welchem sich die Parkmöglichkeit befindet. Hierfür wird zwischen ‚OK', ‚Ersatzwerte', ‚Manuell' oder ‚Störung' unterschieden.

Geschlossen
Dieser Wert zeigt an, ob die Parkmöglichkeit aktuell geöffnet oder geschlossen hat. Bei 0 ist die Parkmöglichkeit offen, bei 1 geschlossen.

Datum und Uhrzeit der Daten
Um die aktuellsten Daten zu verwenden, müssen Datum und Uhrzeit, zu welcher die Daten erstellt sind, gespeichert werden.

In Abbildung 2.5 sind nochmals alle Spaltenbezeichnungen und die dazugehörigen Datentypen aufgelistet.

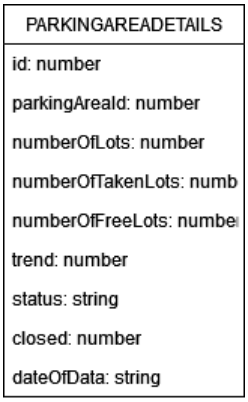

Abbildung 2.5: Um die Daten der API verwenden zu können, wird eine Tabelle ‚parkingareadetails' erstellt. Zu sehen sind hier die Spaltenbezeichnungen und die dazugehörigen Datentypen. (Quelle: Eigens erstellte Übersicht.)

Kapitel 3

Implementierung

In diesem Kapitel wird die Implementierung beschrieben. Zunächst werden die Komponenten und Klassen der Anwendung vorgestellt. Die gezeichneten Mockups unterscheiden sich nur minimal von der tatsächlichen Anwendung. Im Großen und Ganzen ist das Design aber gleich geblieben, was bei Betrachtung von Abbildung 3.1 auffällt.

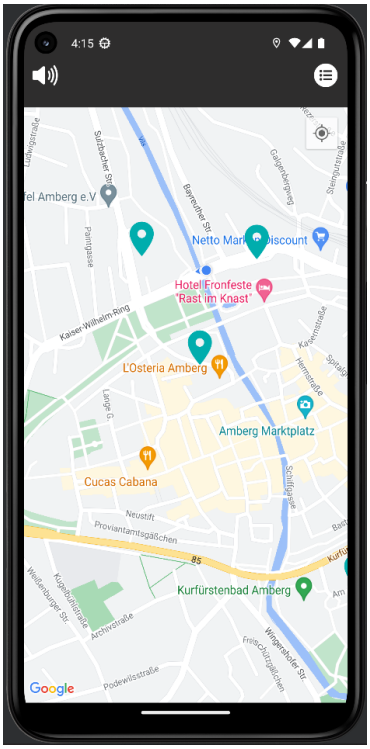

Abbildung 3.1: Die Startseite der fertigen Anwendung, auf welcher nun die Parkmöglichkeiten zu sehen und durch Klick auf einen der türkisblauen Marker nähere Informationen zu finden sind. Oben rechts kann auch eine Liste der Parkmöglichkeiten aufgerufen werden. In der linken oberen Ecke kann der Ton an- und ausgeschaltet werden. (Quelle: Screenshot der erstellten Anwendung.)

Die genutzten Farben werden generiert. Grundlage ist die zuvor ausgewählte dunkelgraue Hintergrundfarbe. So wird sichergestellt, dass die Farben auch wirklich zusammenpassen [4]. Eine Bibliothek für die Icons ist standardmäßig bei Expo-Projekten integriert, dennoch müssen die Bezeichnungen zu den passenden Icons gefunden werden, was über die react-native-vector-icons-Directory gut geklappt hat [5]. Für das Icon und den Ladescreen der App wird ein Icon von Material-Icon von Google verwendet [6].

3.1 Komponenten

Die Komponenten bilden alle Funktionen der App, welche visuell auf der Benutzeroberfläche wahrgenommen werden. Neben der Anzeige der Elemente beinhalten sie aber dennoch auch Funktionalität, wie beispielsweise das Aufrufen von Methoden der Datenbank-Klasse ‚DbConnectionService'. Alle Komponenten, mit Ausnahme der ‚App', besitzen ein eigenes Interface. Dieses Interface beinhaltet alle Elemente, die als Properties in die Komponente hineingegeben werden und die jeweils dazugehörigen Typen. Properties können als Parameter einer Komponente verstanden werden. Dies können neben Strings, Zahlenwerten oder boolschen Werten auch Funktionen oder andere Objekte sein.

3.1.1 Komponente ‚App'

Die ‚App'-Komponente kann sozusagen mit der ‚main()'-Funktion anderer Programmiersprachen verglichen werden. Alles, was in dieser Funktion seinen Platz findet, findet sich auch in der Anwendung selbst wieder. Alle notwendigen Komponenten, die die Anwendung benötigt, werden hier eingefügt.

Zu Beginn werden hier die Tabellen der Datenbank erstellt, sofern diese noch nicht existieren. Diese Komponente ist auch zur Weitergabe von Daten an andere Komponenten zuständig. Ebenso nimmt sie Daten von Komponenten entgegen oder auch von der API. Durch den Hook ‚useEffect' wird sichergestellt, dass die Daten nach einer bestimmten Zeit erneut abgefragt werden. Der Code hierfür kommt aus dem Internet und ist in Listing 3.1 zu sehen.

```
1  useEffect(() => {
2     const intervalCall = setInterval(() => {
3        dbConnectionService.getData();
4     }, MINUTES_MS);
5     return () => {
6        clearInterval(intervalCall);
7     };
8  }, []);
```

Listing 3.1: In diesem ‚useEffect'-Hook werden nach der Zeit, welche sich hinter der Variable ‚MINUTES_MS' verbirgt, die Daten der API abgerufen, was über die Funktion in Zeile 3 geschieht. (Quelle: [7])

Desweiteren gehen auch die Daten des Eventhooks ‚useGeofenceEvent', welcher in Abschnitt 3.3 erklärt wird, hier ein. Durch die Properties der anderen Komponenten werden die Daten dann an die richtigen Stellen gebracht. Wird ein Geofence betreten, so wird hier die Sprachausgabe gestartet. Der Button für das Ein- und Ausschalten des Tons, sowie seine Verwaltung befinden sich ebenfalls in der ‚App'. Wird eine andere Ansicht geöffnet oder der Button zum Stummschalten der Sprachausgabe betätigt, so wird ein ‚useEffect'-Hook getriggert, welcher die Sprachausgabe direkt beendet.

Neben verschiedenen ‚set'- und ‚get'-Funktionen, die für die Weitergabe von Daten zwischen Komponenten verantwortlich sind, befindet sich hier auch das Element, welches benötigt wird, um Toasts auszugeben. Alle Bestandteile in der ‚View'-Komponente werden von ‚<RootSiblingParent>' umrahmt, damit nachher an verschiedenen Stellen der Befehl aus Listing 3.2 aufgerufen werden kann und somit in der Anwendung einen Toast anzeigt [8].

```
1  Toast.show(errorMessages.noApiConnectionMessage, {
2    duration: Toast.durations.LONG,
3    position: Toast.positions.BOTTOM,
4  });
```

Listing 3.2: Ein Beispiel des Aufrufs eines Toasts aus der Datei ‚DbConnectionService.ts'. (Quelle: Eigene Implementierung)

3.1.2 Komponente ‚ParkingMap'

Die wohl wichtigste Komponente der Anwendung ist die, welche die Karte beinhaltet. Diese findet sich hier. Zunächst wird beschrieben, welche Properties mitgeliefert werden. Hierzu ist es hilfreich, das zugehörige Interface ‚IParkingMap' mit den Typen der Properties zu betrachten:

handleParkingAreaId(id: number): void
Hineingegeben wird eine Funktion ‚handleParkingAreaId', welche nichts zurückgibt. Parameter ist hier ‚id', der den Typ Nummer hat. Diese Funktion bringt die jeweilige ID der Parkmöglichkeit nach draußen in die ‚App' und kann dort weiterverarbeitet werden.

handleParkingAreaDescription(parkingAreaDescription: boolean): void
‚handleParkingAreaDescription' ist eine Funktion, welche nichts zurückgibt. Der Parameter dieser Funktion ist ein boolscher Wert namens ‚parkingAreaDescription'. Diese Property hat in etwa denselben Nutzen, wie die eben beschriebene. Nur wird hier der ‚App' mitgeteilt, ob die Beschreibung einer Parkmöglichkeit geöffnet werden soll oder nicht.

mapStyle: StyleProp<ViewStyle>
Diese Property ist für das Design der ‚ParkingMap' verantwortlich. Je nachdem, ob die Beschreibung der Parkmöglichkeit angezeigt wird oder nicht, ist die ‚ParkingMap' größer oder kleiner.

Diese Komponete übernimmt die Anzeige der aktuellen Position des Nutzers und das Bilden der Geofences. Damit das funktioniert, wird die Bibliothek Location von expo benötigt [10]. Zur Nutzung von Geofences und der Anzeige der aktuellen Position wird die Erlaubnis des Nutzers für das GPS benötigt. Ohne diese Zustimmung wird dem Nutzer eine Meldung angezeigt, dass weder das Geofencing, noch die Anzeige der aktuellen Position funktionieren. Damit das Geofencing funktionieren kann, muss mit Hilfe der Expo-Bibliothek Task-Manager der mit einem Namen definierte Task gefunden werden, der das Geofencing startet [9].

Um nun auch die Karte und die Marker an den Positionen der Parkmöglichkeiten anzeigen zu lassen, werden zusätzlich die Komponenten ‚MapView' und ‚Marker' der Bibliothek ‚react-native-maps' benötigt. Je nach Gerät wird dann bei Android „Google Maps" und bei IOS „Apple Maps" verwendet. Um nun die Marker an den richtigen Stellen zu plazieren, wird die Liste aller Objekte der Parkmöglichkeiten mit der Funktion ‚map()' durchgegangen und an jeder Stelle der Parkhäuser ein Marker gesetzt. Die Daten der Parkhäuser sind hier nicht aus der Datenbank, sondern aus dem Objekt-Array ‚AllParkingAreas', welches in Abschnitt 3.5 näher erklärt wird. Wird auf einen Marker geklickt, kommen die beiden Funktionen der Properties ins Spiel. Der Aufruf der Funktion ‚handleParkingAreaId' mit der ID der angeklickten Parkmöglichkeit als Parameter lässt die ‚App' wissen, welche Parkmöglichkeit angeklickt wurde. Wenn die Funktion ‚handleParkingAreaDescription' mit dem Parameter ‚true' aufgerufen wird, wird sichergestellt, dass sich dann immer die Beschreibung, welche in Abbildung 3.2 zu sehen ist, öffnet.

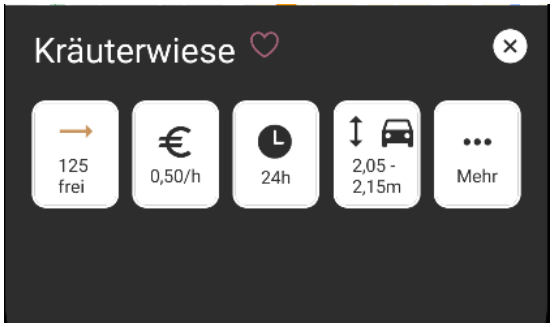

Abbildung 3.2: Die Komponente ‚ParkingAreaDescription', welche bei Klick auf einen Marker aufgerufen wird. (Quelle: Screenshot der erstellten Anwendung.)

3.1.3 Komponente ‚ParkingAreaDescription'

In dieser Komponente werden die relevantesten Daten einer Parkmöglichkeit in Kacheln angezeigt, wie in Abbildung 3.2 zu sehen ist. Auch für die Properties dieser Komponente gibt es ein Interface ‚IParkingAreaDescription':

dbConnectionService: DbConnectionService
Dies ist eine Instanz des ‚DbConnectionService', welche in der ‚App' am Anfang erstellt wird. Diese Property ist für die Datenbankverbindung zuständig.

id: number
Mit der Nummer ID kommt die ID der angeklickten Parkmöglichkeit in die

Beschreibung. Nun kann mit Hilfe von ‚dbConnectionService' in der Datenbank nach den passenden Daten gesucht werden.

geofenceEventData: IEventData[]
Die Daten des Events ‚useGeofenceEvent' werden jeweils in einem Objekt mit dem Interface ‚IEventData' gespeichert. Jedes Objekt, was eine Parkmöglichkeit zeigt, findet Platz in einem Array. Dieses bekommt auch die ‚ParkingAreaDescription', damit unterhalb der Informationskacheln ein Button für die Navigation angezeigt werden kann, falls der Nutzer sich im Geofence einer Parkmöglichkeit befindet. Dieser Button ist auch auf Abbildung 3.3 zu sehen. Wenn dieser betätigt wird, wird zuerst gefragt, ob eine Weiterleitung zu Google Maps oder Apple Maps erwünscht ist. Wird das bejaht, so wird, je nachdem, ob es sich bei dem Gerät um Android oder IOS handelt, zu Google Maps oder Apple Maps weitergeleitet. Hier ist dann die gewünschte Strecke eingetragen und es muss nur noch die Navigation gestartet werden.

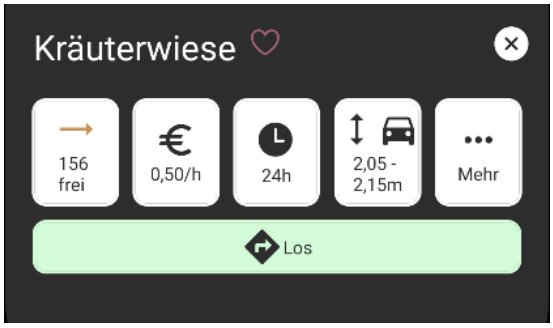

Abbildung 3.3: Die Komponente ‚ParkingAreaDescription', welche bei Klick auf einen Marker aufgerufen wird. Hier befindet sich der Nutzer im Geofence des Parkplatzes ‚Kräuterwiese' und kann über den länglichen unteren Button eine Navigation dahin starten. (Quelle: Screenshot der erstellten Anwendung.)

handleShowParkingAreaDescription(parkingAreaDescription: boolean): void
Diese Funktion vom Typ ‚void' mit der dazugehörigen boolschen Variable lässt diese Komponente schließen. Wird dieser Wert auf ‚false' gesetzt, so ist diese Funktion dafür zuständig, dies der App mitzuteilen, welche wiederum für die Anzeige dieser Komponente zuständig ist. Es wird geprüft, ob ‚parkingAreaDescription' ‚true' oder ‚false' anzeigt. Bei ‚true' öffnet sich diese Komponente, bei ‚false' wird sie geschlossen. Dies geschieht beispielsweise, wenn der kleine weiße Kreis in der rechten Ecke in Abbildung 3.3 betätigt wird.

handleParkingAreaDetails(parkingAreaDetails: boolean): void
Der boolsche Parameter der Funktion zeigt an, ob auf den ‚Mehr'-Button gedrückt wird. Ist der Button gedrückt worden, so wird die Komponente ‚ParkingAreaDetails' geöffnet.

handleParkingAreaData(parkingAreaData: IParkingArea): void
Diese Funktion beinhaltet ein Objekt des Typs ‚IParkingArea', welches das Interface für einen Eintrag einer Parkmöglichkeit ist und weiter unten beschrieben

wird. Die Daten zur angeklickten Parkmöglichkeit gelangen so zur ‚App'.

handleParkingAreaDetailsData(parkingAreaDetails: IParkingAreaDetails): void
Diese Property macht genau das gleiche, wie die vorherige: Es werden Daten zur ‚App' getragen. In dem Fall handelt es sich um die Daten, wie voll die Parkmöglichkeit zum aktuellen Zeitpunkt ist.

handleDataBaseError(databaseError: boolean): void
Diese Funktion beinhaltet einen boolschen Parameter, welcher der ‚App' mitteilt, ob es zu einem Fehler in der Datenbank gekommen ist.

handleVolume(volume: boolean): void
Werden die Navigation mit dem ‚Los'-Button oder die Details mit dem ‚Mehr'-Button geöffnet, so wird der ‚App' mitgeteilt, dass der Ton der Sprachausgabe beendet werden soll. Würde dies weggelassen werden, so würde die Sprachausgabe fortgesetzt werden, wenn der Nutzer möglicherweise schon gar nicht mehr in der Anwendung, sondern schon bei der Navigation ist.

In der ‚ParkingAreaDescription'-Komponente werden die Daten der angeklickten Parkmöglichkeit aus beiden Datenbanken abgerufen und dann in Kacheln dargestellt. Um den Code übersichtlicher zu gestalten, sind die Kacheln eine weitere Komponente: ‚ParkingAreaDescriptionItemContainer', welche in Unterabschnitt 3.1.4 näher beschrieben wird. Zudem kann hier das Parkhaus favorisiert werden, was direkt in die Datenbank eingetragen wird. Kommt es zu einem Datenbankfehler, so wird dies dem Nutzer über ein Alert mitgeteilt. Damit die Abfrage der Datenbank klappt und die Daten vor dem Rendern in den einzelnen Komponenten der ‚ParkingAreaDescription'-Komponente stehen, muss die Abfrage asynchron ablaufen. Das heißt, dass gewartet werden muss, bis die Daten da sind, bevor sie angezeigt werden.

3.1.4 Komponente ‚ParkingAreaDescriptionItemContainer'

Diese Komponente ist die Kachel der ‚ParkingAreaDescription'. Auch sie hat ein Interface für die Properties. Dieses beinhaltet aber nur den Text, einen oder zwei Namen für Icons und die Farbe des jeweiligen Icons. Ansonsten besitzt diese Komponente keinerlei Funktionalität. Sie ist nur für das Anzeigen zuständig und, dass in der ‚ParkingAreaDescription'-Komponente weniger redundanter Code ist.

3.1.5 Komponente ‚ParkingAreaList'

In Abbildung 3.4 ist diese Komponente zu sehen. Hier sind die Parkmöglichkeiten mit Favoriten zuerst als anklickbare Kacheln alphabetisch aufgelistet.

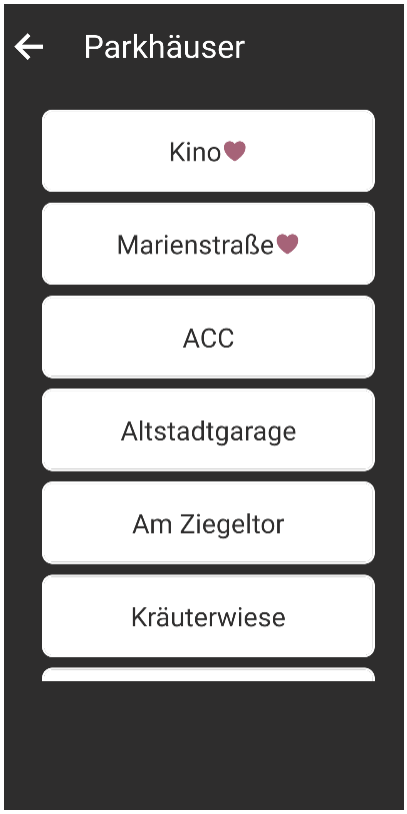

Abbildung 3.4: Die Komponente ‚ParkingAreaList', zu welcher über einen Klick auf das Icon oben rechts in der App navigiert wird. Die Liste ist mit Favoriten zuerst, alphabetisch geordnet. (Quelle: Screenshot der erstellten Anwendung.)

Die ‚ParkingAreaList'-Komponente besitzt auch ein Interface für ihre Properties. So können die Typen dieser sichergestellt werden. Das Interface sieht folgendermaßen aus:

dbConnectionService: DbConnectionService
Diese Property vom Typ DbConnectionService ist wichtig für die Abfrage der Daten aus der Datenbank.

handleShowParkingAreaList(showParkingAreaList: boolean): void
Diese Funktion ist dafür zuständig, die ‚ParkingAreaList'-Komponente zu schließen, wenn auf den Zurück-Pfeil gedrückt wird. In Abbildung 3.4 ist die Liste mit dem Zurück-Pfeil zu sehen.

handleParkingAreaDescription(parkingAreaDescription: boolean): void
Die ‚ParkingAreaDescription'-Komponente wird bei Drücken auf den Zurück-Pfeil geschlossen, damit die Beschreibung nicht ohne Daten geöffnet ist.

handleParkingAreaDetails(parkingAreaDetails: boolean): void
Diese Funktion erfüllt denselben Zweck wie ‚handleParkingAreaDetails' in Unterabschnitt 3.1.3.

handleParkingAreaData(parkingAreaData: IParkingArea): void
Auch hier tut die Funktion das Gleiche, wie ‚handleParkingAreaData' in Unterabschnitt 3.1.3.

handleParkingAreaDetailsData(parkingAreaDetails: IParkingAreaDetails): void
In Unterabschnitt 3.1.3 unter ‚handleParkingAreaDetailsData' ist beschrieben, was auch diese Funktion macht.

handleDataBaseError(databaseError: boolean): void
Die Beschreibung dieser Property befindet sich auch in Unterabschnitt 3.1.3 unter ‚handleDataBaseError'.

Die Überschrift dieser Komponente geht aus der Komponente ‚ParkingAreaListHeadig' hervor, welche in Unterabschnitt 3.1.8 zu finden ist. Ist bei der Datenbank-Abfrage in dieser Komponente kein Fehler aufgetreten, so werden die Namen der Parkmöglichkeiten als Flatlist dargestellt. Die Kacheln, in welchen sich die Namen der Parkmöglichkeiten befinden, gehen aus der Komponente ‚ParkingAreaListItem' hervor, welche in Unterabschnitt 3.1.6 zu finden ist. Wird auf eine dieser Kacheln geklickt, so kommt es zu einer weiteren Abfrage der Datenbank, wobei ‚handleParkingAreaData' und ‚handleParkingAreaDetailsData' der eben beschriebenen Properties mit den Daten gefüllt werden. Die Property ‚handleParkingAreaDetails' wird dann auf ‚true' gesetzt, um die Komponente in Unterabschnitt 3.1.7 zu öffnen. Je nachdem, ob ein Fehler bei der Datenbankabfrage auftritt, wird der Parameter in ‚handleDataBaseError' auf ‚true' oder ‚false' gesetzt.

3.1.6 Komponente ‚ParkingAreaListItem'

Diese Komponente besitzt, wie alle anderen auch ein Interface. Da es sich um anklickbare Kacheln handelt, muss dieser Komponente die Funktion mitgeteilt werden, welche beim Klick ausgeführt werden soll. Für die Ausführung der Funktion braucht es zudem noch die ID der Parkmöglichkeit. Um den Namen und, falls die Parkmöglichkeit favorisiert ist, ein Herz anzuzeigen, müssen auch noch der anzuzeigende Name als String und ein numerischer Wert für die Favorisierung mitgegeben werden. 0 heißt, dass die Parkmöglichkeit nicht unter die Favoriten fällt. 1 bedeutet, dass diese favorisiert ist.

3.1.7 Komponente ‚ParkingAreaDetails'

Wie bereits beschrieben, öffnet sich diese Komponente beim Klick auf ‚mehr' in Abbildung 3.2 oder bei Auswahl einer Parkmöglichkeit in Abbildung 3.4. Diese Komponente dient lediglich zum Anzeigen von Daten. Dennoch kann auch hier eine Parkmöglichkeit zu den Favoriten hinzugefügt oder entfernt werden. Zu sehen ist diese Ansicht in Abbildung 3.5.

Abbildung 3.5: Die Details zu einer Parkmöglichkeit. Hier werden alle vorhandenen Daten zur Parkmöglichkeit angezeigt. Sollte diese geschlossen sein oder aus anderen Gründen nicht befahrbar sein, wird dies dem Nutzer mit einem Text in einer roten Kachel mitgeteilt. Der Nutzer kann auch hier Parkmöglichkeiten favorisieren oder von den Favoriten entfernen. (Quelle: Screenshot der erstellten Anwendung.)

Die Properties, welche durch das Interface ihren Typ erhalten, sind folgende:

dbConnectionService: DbConnectionService
Diese Propertie ist dafür zuständig, dass beim Drücken auf das Herz in der oberen rechten Ecke eine Parkmöglichkeit in der Datenbank als Favorit oder kein Favorit eingetragen werden kann.

handleShowParkingAreaDetails(parkingAreaDetails: boolean): void
Diese Funktion wird benötigt, wenn der Pfeil in der oberen linken Ecke gedrückt wird. Betätigt der Nutzer den Pfeil-Button, so wird ‚parkingAreaDetails' auf ‚false' gesetzt und die Ansicht aus Abbildung 3.5 wird geschlossen.

parkingAreaData: IParkingArea
Das Objekt, welches Parkhausdaten, wie die Einfahrtshöhe oder den Preis, beinhaltet.

parkingAreaDetailsData: IParkingAreaDetails
Das Objekt, welches die Daten der API beinhaltet, welche unter ‚Parkplätze' in Abbildung 3.5 dargestellt sind.

databaseError: boolean
: Ist diese Property ‚true', so bekommt der Nutzer eine rot gefärbte Kachel mit einem Text angezeigt, der ihn darauf hinweist, dass die Daten nicht angezeigt werden können. In diesem Fall ist die Aktion zu wiederholen.

Da hier die Kacheln ebenfalls aus jeweils den gleichen ‚View'-Komponenten und demselben Design bestehen, wird hier wieder eine Komponente ‚ParkingAreaDetailsItem' erstellt, welche sich darum kümmert, die Elemente einheitlich anzuzeigen. Die Überschrift kommt ebenfalls durch die Komponente ‚ParkingAreaListHeading' in Unterabschnitt 3.1.8 zustande, wie die Überschrift in Unterabschnitt 3.1.5.

Komponente ‚ParkingAreaDetailsItem'

Diese Komponente ist ebenfalls nur zum Anzeigen von Dateien. Ihre Properties sind der Text, der in der Kachel jeweils oben kleiner angezeigt wird und jener Text, welcher etwas größer darin steht und abhängig von der angeklickten Parkmöglichkeit ist. Außerdem wird eine boolsche Variable hineingegeben, die dieser Komponente mitteilt, ob es sich um die Anzeige eines Fehlers handelt. Ist das der Fall, werden die Kacheln rot gefärbt.

3.1.8 Komponente ‚ParkingAreaListHeading'

Diese Komponente beinhaltet das Design der Überschrift, wie in Abbildung 3.4 oder Abbildung 3.5 zu sehen. Als Property werden hier lediglich die Funktion, die beim Klick auf den Zurück-Pfeil erfolgen soll und der Text, welcher dargestellt werden soll, hineingegeben.

3.2 Datenbankverbindung ‚DbConnectionService'

Zur Speicherung der Daten der Parkmöglichkeiten wird SQLite verwendet. Sämtliche Funktionen, die Datenbankzugriff benötigen, sind in der Klasse ‚DbConnectionService' niedergeschrieben. Mit Hilfe eines Beispiels in dieser Expo-Bibliothek konnte eine Datenbank erstellt werden, wie in Listing 3.3 zu sehen. Diese Funktion wird im Konstruktor der Klasse aufgerufen. eine Instanz der Klasse wird in der ‚App' erstellt und an alle Komponenten, die Datenbankzugriff benötigen, weitergegeben.

```
1  private openDatabase = () => {
2     if (Platform.OS === "web") {
3        return {
4           transaction: () => {
5              return {
6                 executeSql: () => {},
7              };
8           },
9        };
10    }
```

```
11   const db = SQLite.openDatabase("db.db");
12   return db;
13 };
```

Listing 3.3: Das Erstellen einer Datenbank mit Hilfe eines Beispiels der genutzten Expo-Bibliothek SQLite. (Quelle: [12])

Diese Klasse besitzt folgende Funktionen:

public createTables()
Mit dieser Funktion werden die Tabellen ‚parkingarea' und ‚parkingareadetails' mit einem SQL-Befehl erstellt, falls diese nicht existieren. Ist die Tabelle ‚parkingarea' leer, so wird diese mit den Daten des Objekt-Arrays aus Abschnitt 3.5 gefüllt.

public getData()
Hier werden die Daten der API mittels ‚fetch()' geholt und mit einem XML-Parser in ein Objekt umgewandelt [13]. Um die Umlaute im Namen und dem Status für einen Nutzer leserlich zu machen, wird das Encoding danach noch geändert [14]. Anschließend werden die Daten in die Tabelle ‚parkingAreaDetails' mit Hilfe der folgenden Funktion eingefügt.

private insertIntoDetailsTable(name: string, dateOfData: string, numberOfLots: number, numberOfTakenLots: number, numberOfFreeLots: number, trend: number, status: string, closed: number, ctr: number)
Mit dieser Funktion werden die Daten in die Tabelle ‚parkingareadetails' eingefügt. Danach wird der jeweils älteste Datensatz gelöscht, sodass pro Parkhaus nur drei Datensätze existieren. Das ist wichtig, da dann in jedem Fall Daten angezeigt werden können, auch, wenn diese schon älter sind. Der Nutzer sieht also immer direkt Daten und nicht nur leere Elemente.

public async getDataFromParkingAreaTable(parkingAreaId: number)
Mit Hilfe der ID wird in der Tabelle ‚parkingarea' nach der richtigen Parkmöglichkeit gesucht. Die Daten werden in einem Objekt des Typs ‚IParkingArea' gespeichert. Da diese Funktion asynchron ist, gibt sie ein Promis zurück. Dies ist notwendig, da sonst die Benutzeroberfläche gerendert sind, bevor die Datenbankabfrage getätigt werden kann. So wird zunächst die Abfrage getätigt und danach die Benutzeroberfläche gerendert.

public async getDataFromParkingAreaDetailsTable(parkingAreaId: number)
Diese Funktion tut dasselbe wie ‚public async getDataFromParkingAreaTable (parkingAreaId: number)'. Nur werden hier die Daten aus der ‚parkingareadetails'-Tabelle genommen und in einem Objekt des Typs ‚IParkingAreaDetails' gespeichert.

public async getParkingAreas()
Aus oben genannten Gründen ist auch diese Funktion wieder asynchron. Diese Abfrage liefert alle Parkmöglichkeiten der ‚parkingarea'-Tabelle geordnet nach Favoriten und Namen. Dieses Ergebnis hat den Typ ‚SQLResultSetRowList',

welcher aus der SQLite Bibliothek kommt, und kann mit dem Anhang ‚._array' in ein Array umgewandelt werden.

public setFavoriteParkingArea(favorite: number, name: string)
Diese Funktion ist für das Updaten der Parkmöglichkeit zuständig. Mit einem SQL-Befehl wird die Parkmöglichkeit mit dem mitgegebenen Namen ‚name' aktualisiert und der Wert, ob die Parkmöglichkeit favorisiert wird oder nicht, wird auf ‚favorite' gesetzt. Hierbei bedeutet 0 kein Favorit und 1, dass die Örtlichkeit favorisiert ist.

Bei jeglichen Datenbankfehlern wird ein Alert ausgegeben, um dem Nutzer mitzuteilen, dass etwas schief gelaufen ist. In so einem Fall empfiehlt es sich, die Aktion zu wiederholen oder die App neu zu starten [12].

3.3 Eventhook ‚useGeofenceEvent'

Dies ist der selbstdefinierte Hook, der die Daten vom Taskmanager in die App bringt, in der sie weiter verteilt werden können. In der ‚ParkingMap'-Komponente wird das Geofencing mit einem Task gestartet. Im Hook wird der Task mit demselben Task-Namen definiert. Zudem gibt es ein Array mit Funktionen, die im Task abgearbeitet werden sollen. Vor der Definition des Hooks findet sich also der Code aus Listing 3.4.

```
1  let geofenceHandles: TaskManager.TaskManagerTaskExecutor[] = [];
2
3  TaskManager.defineTask(configStrings.geofenceTask, (data) => {
4     for (const handle of geofenceHandles) {
5        handle(data);
6     }
7  });
```

Listing 3.4: Dieser Code befindet sich vor der Definition des Eventhooks. Hier wird ein Array angelegt, welches Funktionen beinhaltet. Diese werden dann ab Zeile 11 im definierten Task abgearbeitet. (Quelle: Eigene Implementierung)

Im Eventhook werden dann zunächst Variablen definiert: Ein ‚useState' vom Typ ‚IEventData' und ein Objekt vom Typ ‚IEventData'. Dies ist notwendig, da bei Verwendung des Emulators bei höheren Abspielgeschwindigkeiten des Abfahrens der Route ein Geofence fälschlicherweise mehrmals betreten wird und so das Event mehrmals ausgelöst wird. Die Daten des ‚useState' aktualisieren sich jedoch nicht schnell genug, um zu überprüfen, ob das Geofence zum zweiten Mal hintereinander betreten und nicht verlassen wird. Aus diesem Grund wird das Objekt benötigt. Jedoch kann der ‚useState' auch nicht weggelassen werden, da in der ‚App' keine Daten ankommen, wenn nur das Objekt verwendet wird.

Als nächstes wird dann ein ‚useEffect' erstellt, in dem eine Funktion ‚handleIsInGeofence' definiert wird. Diese ist in Listing A.1 in Anhang A zu erkennen. Im Fehlerfall wird ein Toast ausgegeben, der dem Nutzer mitteilt, dass das Geofencing gerade nicht

möglich ist. Von Zeile 10 bis 17 wird überprüft, ob das Geofence mehrmals hintereinander betreten wird. Ist das der Fall, so kommt es zu einem ‚return' in Zeile 15. Ab Zeile 19 steht, was beim Betreten von Geofences passiert: Das Objekt und der ‚useState' werden aktualisiert und der Nutzer bekommt mit einem Toast mitgeteilt, wo er sich befindet. Verlässt der Nutzer das Geofence, so tritt der Code ab Zeile 30 in Kraft. Hier werden nur die Daten des Objekts und des ‚useStates' aktualisiert. Anschließend wird die Funktion ‚handleIsInGeofence' zu dem oben angelegten Array aus Funktionen hinzugefügt, um vom Taskmanager abgearbeitet zu werden. Danach wird die Funktion aus dem Array entfernt.

Der Eventhook gibt den ‚useState' zurück und die benötigten Daten landen in der ‚App', wo sie in ein Array aus Objekten aller Parkmöglichkeiten übertragen werden. Bei jeder Parkmöglichkeit bei der ‚ParkingAreaDescription'-Komponente wird so der ‚Los'-Button aus Abbildung 3.3 angezeigt, wenn sich der Nutzer im Geofence befindet.

3.4 Models

Damit die Datentypen von Objekten stimmen, werden Models in Form von Interfaces erstellt. Das hilft, um mehrfach benutzten Objekten immer denselben Datentyp mitzugeben. Der Entwickler wird so zudem schneller auf Unstimmigkeiten bezüglich des Datentyps aufmerksam gemacht.

3.4.1 Interface ‚IParkingArea'

Mit ‚IParkingArea' erstellte Objekte beinhalten alle Daten der Parkmöglichkeiten, die sich gar nicht oder seltener ändern. Diese sind in der nachfolgenden Tabelle 3.1 dargestellt. Die Schlüssel sind hierbei dieselben, wie die Spalten in der Datenbanktabelle ‚parkingarea'.

Schlüssel	Datentyp
id	number
name	string
address	string
openingHours	number
pricePerHour	string
doorHeight	string
favorite	0 | 1
lat	number
long	number

Tabelle 3.1: Die Schlüssel und ihre Datentypen des Interfaces ‚IParkingArea'.

3.4.2 Interface ‚IParkingAreaDetails'

Auch hier orientieren sich die Datentypen und die Schlüssel wieder an einer Datenbanktabelle. In diesem Fall ist das ‚parkingareadetails', wie in Tabelle 3.2 erkennbar ist.

Schlüssel	Datentyp
id	number
parkingAreaId	number
numberOfLots	number
numberOfTakenLots	number
numberOfFreeLots	number
trend	0 | 1 | -1
status	"OK" | "Ersatzwerte" | "Manuell" | "Störung"
closed	0 | 1
dateOfData	string

Tabelle 3.2: Die Schlüssel und Datentypen des Interfaces ‚IParkingAreaDetails'.

3.4.3 Interface ‚IEventData'

Da dieses Interface, wie oben beschrieben, auch häufiger benutzt wird, wird es in eine extra Datei ausgelagert. Benötigt wird es für den Umgang mit den Geofences. Dieses Interface besitzt zwei Schlüssel mit Datentypen, wie in Tabelle 3.3 zu sehen.

Schlüssel	Datentyp
parkingAreaName	string
enteredParkingArea	boolean

Tabelle 3.3: Die Schlüssel und Datentypen des Interfaces ‚IParkingAreaDetails'.

3.5 Parkhausdaten ‚AllParkingAreas'

Damit die Marker der ‚ParkingMap' gesetzt und die Daten in die Datenbanktabelle ‚parkingarea' eingefügt werden können, werden die Daten der Parkmöglichkeiten benötigt. Diese befinden sich in einer TypeScript-Datei, da dies hilft, die Datentypen zu bewahren. Die Darstellung der Daten ist wie bei einer JSON-Datei. Dennoch gibt es so die Möglichkeiten, Datentypen für die Werte der einzelnen Schlüssel festzulegen.

Als Interface für die Werte wird ‚IParkingArea' aus Unterabschnitt 3.4.1 verwendet. In Listing 3.5 ist diese Datei verkürzt dargestellt.

```
import { IParkingArea } from "./models/IParkingArea";

export const allParkingAreas: IParkingArea[] = [
{
   id: 4,
   name: "Am Ziegeltor",
   address: "Pfalzgrafenring",
   openingHours: 24,
   pricePerHour: "0,50",
   doorHeight: "2,10",
   favorite: 0,
   lat: 49.44854933741437,
   long: 11.856581325108372,
},
{
   id: 8,
   name: "Altstadtgarage",
   address: "Kaiser-Ludwig-Ring",
   openingHours: 24,
   pricePerHour: "1,00",
   doorHeight: "2,00",
   favorite: 0,
   lat: 49.447409726965354,
   long: 11.861757130222472,
},
];
```

Listing 3.5: Die verkürzte Darstellung der Datei, welche die Daten der Parkmöglichkeiten beinhaltet. Wäre Typensicherheit egal, könnten die Daten auch in eine JSON-Datei geschrieben werden. (Quelle: Eigene Implementierung)

3.6 Zentrale Verwaltung der Farben und Strings

Damit die Farben und Strings bei Änderungen nicht an jeder Stelle im Quellcode geändert werden müssen, werden diese zentral in zwei Dateien verwaltet. In der ‚colors'-Datei finden sich die Farben als gleichnamiges Objekt. Hierzu gibt es Schlüssel-Wert-Paare mit Bezeichnungen der Farben und den Farbwerten.

Bei den Strings in der ‚strings'-Datei gilt dasselbe Prinzip. Hier gibt es aber mehrere Objekte, da die Texte gruppiert sind. Die Gruppierungen sind Folgende:

sqlQuerys
: Dieses Objekt enthält alle verwendeten SQL-Befehle. Sollte sich etwas daran ändern, kann das hier geschehen und es muss nicht im Code nach dem Befehl gesucht werden.

configStrings
Diese Kategorie enthält die API mit den Daten der Parkplätze der Parkmöglichkeiten und den Namen des Tasks, der für das Geofencing benutzt wird.

errorMessages
In diesem Objekt finden sich alle Fehler- und Warnmeldungen wieder.

outputText
Hier befinden sich alle weiteren Texte, die dem Nutzer im Laufe der Benutzung der Anwendung angezeigt werden.

Kapitel 4

Starten der App

Damit das Projekt auf dem eigenen Rechner genutzt werden kann, müssen zunächst alle verwendeten Bibliotheken installiert werden. Dies geschieht ganz einfach mit der Ausführung der Befehle aus Listing 4.1 in der Kommandozeile.

```
1 cd path/to/project
2
3 yarn
```

Listing 4.1: Diese Befehle müssen in die Kommandozeile eingegeben und anschließend ausgeführt werden, damit das Projekt verwendet werden kann. So werden die notwendigen Bibliotheken installiert.

Die Anwendung funktioniert auf Android. Gearbeitet wird auf einem ‚Pixel 5 API 33' im Android Emulator.

Um die App zu starten, muss zunächst Android Studio heruntergeladen und installiert werden. Nach dem Start von Android Studio muss über den Device Manager ein passender Emulator heruntergeladen werden. Anschließend muss der Emulator gestartet werden. Danach können die Befehle aus Listing 4.2 in der Kommandozeile ausgeführt werden.

```
1 cd path/to/project
2
3 yarn android
```

Listing 4.2: Um die Anwendung mit mit Android zu starten, sollte Android Studio heruntergeladen und installiert sein. Anschließend muss der passende Emulator über den Device Manager hinzugefügt und gestartet werden. Danach werden diese Befehle in die Kommandozeile eingegeben und ausgeführt.

Literaturverzeichnis

[1] Meta Platforms, Inc. (2023, Januar 05). React Native. Learn once, write anywhere. [online]. Verfügbar: https://reactnative.dev/.

[2] Microsoft. (2023, Januar 05). TypeScript is JavaScript with syntax for types. [online]. Verfügbar: https://www.typescriptlang.org/.

[3] Expo. (2023, Januar 05). What is Expo? An overview of Expo tools, features and services. [online]. Verfügbar: https://docs.expo.dev/introduction/expo/.

[4] ColorSpace. (2023, Januar 05). Never waste Hours on finding the perfect Color Palette again! [online]. Verfügbar: https://mycolor.space/?hex=%232E2D2D&sub=1.

[5] J. Arvidsson. (2023, Januar 05). react-native-vector-icons directory. [online]. Verfügbar: https://oblador.github.io/react-native-vector-icons/.

[6] Google (2023, Januar 15). Google Fonts. [online]. Verfügbar: https://fonts.google.com/icons?selected=Material%20Symbols%20Outlined%3Alocation_on%3AFILL%400%3Bwght%40400%3BGRAD%400%3Bopsz%4048

[7] F. Hameed. (2023, Januar 05). How to call an API every minute for a Dashboard in REACT. For those looking for functional components. [online]. Verfügbar: https://stackoverflow.com/questions/48601813/how-to-call-an-api-every-minute-for-a-dashboard-in-react.

[8] Horcrux. (2023, Januar 05). react-native-root-toast. [online]. Verfügbar: https://github.com/magicismight/react-native-root-toast.

[9] Expo. (2023, Januar 05). TaskManager. [online]. Verfügbar: https://docs.expo.dev/versions/latest/sdk/task-manager/.

[10] Expo. (2023, Januar 05). Location. [online]. Verfügbar: https://docs.expo.dev/versions/v47.0.0/sdk/location/.

[11] Airbnb. (2023, Januar 05). react-native-maps. [online]. Verfügbar: https://github.com/react-native-maps/react-native-maps.

[12] Expo. (2023, Januar 05) examples. [online]. Verfügbar: https://github.com/expo/examples/blob/master/with-sqlite/App.js#L14.

[13] Autor unbekannt. (2023 Januar, 05). fast-xml-parser. [online]. Verfügbar: `https://www.npmjs.com/package/fast-xml-parser`.

[14] Autor unbekannt. (2023 Januar, 05). html-entities. [online]. Verfügbar: `https://www.npmjs.com/package/html-entities`.

Abbildungsverzeichnis

Listingverzeichnis

Anhang A

Code-Ausschnitt

```
1  const handleIsInGeofence = ({ data, error }: any) => {
2     if (error) {
3        console.error(error);
4        Toast.show(errorMessages.noCurrentPosition, {
5           duration: Toast.durations.LONG,
6           position: Toast.positions.CENTER,
7        });
8     }
9
10    if (currentData.enteredParkingArea === true) {
11       if (
12       data.region.identifier === currentData.parkingAreaName &&
13       data.eventType === Location.GeofencingEventType.Enter
14       ) {
15          return;
16       }
17    }
18
19    if (data.eventType === Location.GeofencingEventType.Enter) {
20       currentData.enteredParkingArea = true;
21       currentData.parkingAreaName = data.region.identifier;
22       setEventData({
23          parkingAreaName: data.region.identifier,
24          enteredParkingArea: true,
25       });
26       Toast.show(outputText.inGeofenceMessage + data.region.identifier, {
27          duration: Toast.durations.LONG,
28          position: Toast.positions.CENTER,
29       });
30    } else if (data.eventType === Location.GeofencingEventType.Exit) {
31       currentData.enteredParkingArea = false;
32       currentData.parkingAreaName = data.region.identifier;
33       setEventData({
```

```
34            parkingAreaName: data.region.identifier,
35            enteredParkingArea: false,
36          });
37      }
38  };
```

Listing A.1: Diese Funktion wird beim Betreten oder Verlassen eines Geofences ausgeführt. Im Fehlerfall bekommt der Nutzer per Toast mit, dass etwas schief gelaufen ist. Von Zeile 10 bis 17 wird sichergestellt, dass dasselbe Event nicht fälschlicherweise mehrmals hintereinander auftritt. Anschließend werden die Werte, ob das Geofence verlassen oder betreten wird, gesetzt. Beim Eintritt in das Geofence wird der Nutzer mittels Toast informiert. (Quelle: Eigene Implementierung)