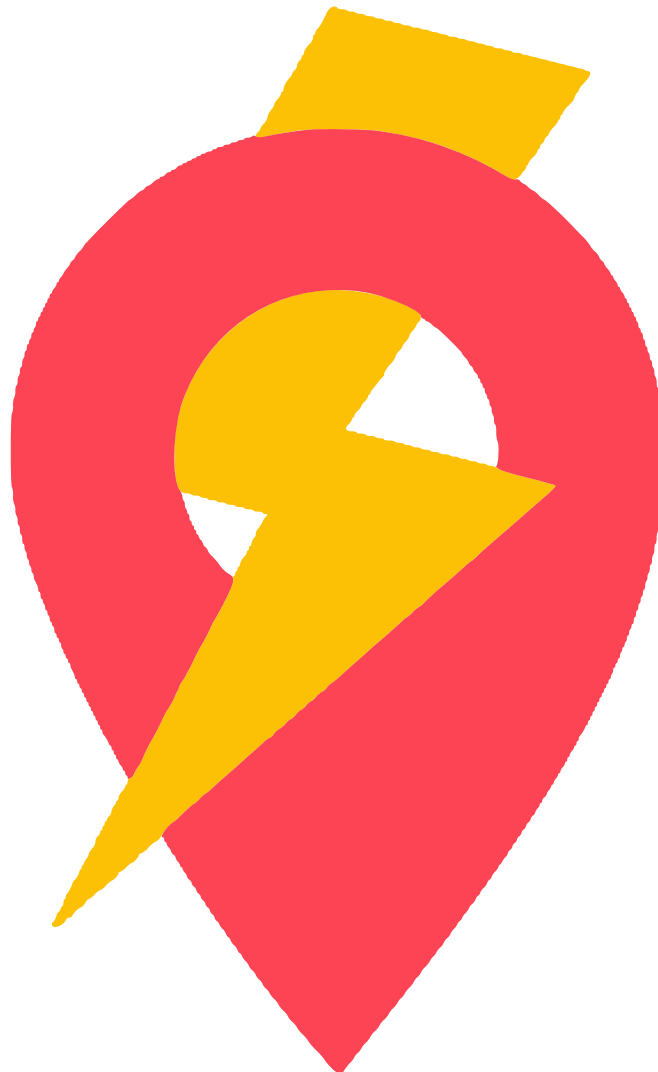


CLEVER CHARGE

DOKUMENTATION



TILO STEINMETZER - 769895

JÖRG QUICK - 762025

Inhaltsverzeichnis

1. Inhaltliche Auseinandersetzung mit der Aufgabe	2
1.1 Aufgabe	2
1.2 Anforderungen and die App	2
1.3 Sonstige Vorgaben.....	3
2. Ablauf der Entwicklung	3
2.1 Praktikums Termine	3
2.2 Implementierungsablauf	4
3. Design	5
3.1 Google Map Screen	5
3.2 Search Screen	6
3.3 Einstellung Screen	7
3.4 Techniker Screen	8
3.5 Landing Screen	9
4. Konkrete Umsetzung im ersten Prototyp.....	10
4.1 Beginn.....	10
4.2 Anforderungen, welche wir umsetzen konnten.....	10
4.3 Anforderungen, welche wir nicht umsetzten konnten	10
4.4 Grundlegender App Aufbau	11
4.5 Activities vs. Fragments	12
4.6 WebAPI und Datenbank	13
4.7 Distanz Berechnung.....	13
4.8 Favoriten, Defekte und Nutzerdaten	14
4.9 Filterfunktion im Suchbildschirm	14
4.10 RecyclerView	15
4.11 Google Map	15
4.12 Persistente Daten	16
4.13 Mehrsprachigkeit	17
4.14 Landscape Mode	17
5. Usability Test	17
6. Fazit	17

1. Inhaltliche Auseinandersetzung mit der Aufgabe

1.1 Aufgabe

Die Aufgabe war die Entwicklung einer App, mit Hilfe User an Hand von deren Standort Ladesäulen in der Nähe angezeigt bekommen. Ebenso sollen User unabhängig von Ihrem Standort Ladesäulen an einer von deren festgelegten Position ermitteln können. Zusätzlich soll ein Suchradius wählbar sein. Die Liste aller Ladestationen wird von der Bundesnetzagentur zur Verfügung gestellt. Darüber hinaus soll die App einen „Normalen“ und einen „Mitarbeiter“ Modus besitzen. Mitarbeiter sollen mehr bzw. interne Informationen angezeigt bekommen und in der Lage sein, defekte Ladesäulen zu warten. Die App sollte so nutzerfreundlich wie möglich gestaltet werden. Das ganze Projekt unterliegt gewissen Rahmenbedingungen, auf die wir in den folgenden Abschnitten weiter eingehen werden. Dazu liegt sowohl Java-Code, als auch Bilder als Beispiel vor.

1.2 Anforderungen and die App

Der erste Teil der Rahmenbedingungen sind die s.g. User Stories. Diese wurden im ersten Praktikumstermin gemeinsam mit allen Studenten formuliert und festgelegt. Eine Zusammenfassung der User Stories ist im Folgenden dargestellt.

Als Nutzer möchte ich ...

- Probleme melden können
- Nach Ladesäulen divers filtern können
- Ladestationen in der Nähe angezeigt bekommen
- freie Ladeplätze angezeigt bekommen
- verfügbare Betreiber angezeigt bekommen
- Strompreise angezeigt bekommen
- Die Ladegeschwindigkeit angezeigt bekommen
- Steckertypen angezeigt bekommen
- Infos über Ladevorgang angezeigt bekommen
- eine Routenplanung zur ausgewählten Ladesäule bekommen
- eine bestimmte Adresse eingeben können
- wissen, wann mein Auto vollgeladen ist
- die Öffnungszeiten angezeigt bekommen
- den Akkustand meines Autos immer wissen
- wissen wo mein Auto steht
- Favoriten speichern können

Als Mitarbeiter möchte ich ...

- defekte Ladesäulen angezeigt bekommen, um diese schnell bearbeiten zu können.
- reparierte Ladesäulen wieder freigeben können, damit diese wieder in Betrieb gehen können
- Angaben zu den defekten Ladesäulen erhalten, damit ich meine Arbeit effizient planen kann
- nach Ladesäulen in meiner Nähe filtern können, um meine Arbeitsrouten besser planen zu können.

1.3 Sonstige Vorgaben

Das Projektumfeld konnten wir uns nicht aussuchen. Die App sollte mit Hilfe der Programmiersprache Java und der IDE Android Studio umgesetzt werden. Die IDE Android Studio beinhaltet bereits einen Android Emulator, welcher benötigt wird um die App nativ auf einem Endgerät zu testen. Für die Codesicherung haben wir GitLab benutzt, für die Zusammenarbeit wurde das Kommunikationsprogramm Discord verwendet. Für die Entwürfe der Screen wurde Adobe XD benutzt.

2. Ablauf der Entwicklung

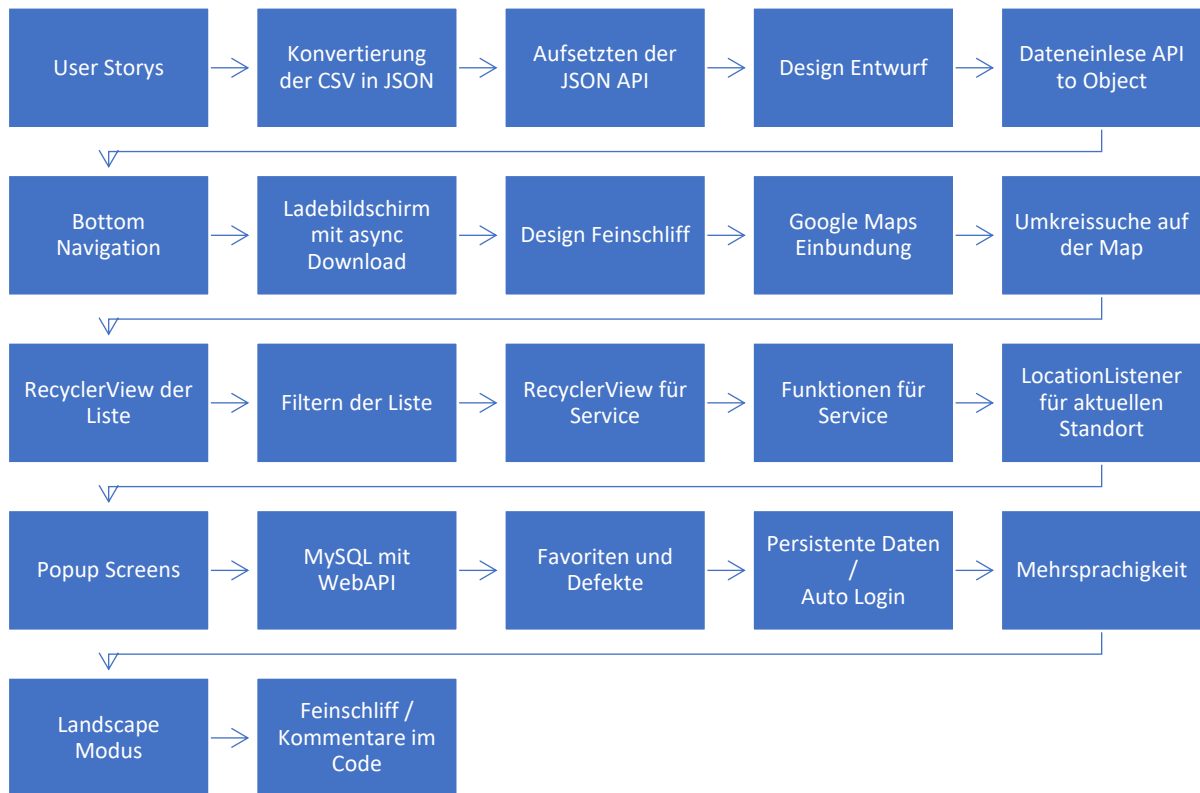
2.1 Praktikums Termine

Durch die vorgegebenen Praktikums Termine und damit in Verbindung stehenden Anforderungen an die App wurde der gesamten Entwicklung von vorne herein Struktur gegeben.

Termin	Datum	Inhalt
1	04.11.2021	Gemeinsames erstellen der User Stories.
2	18.11.2021	Einführung in Android Studio. Screens / Activities / Fragmente / Navigationbar erstellen.
3	2.12.2021	Abläufe festlegen. Einbettung des Adapters für den RecyclerView.
4	16.12.2021	RecyclerView mit dem Adapter verknüpfen. RecyclerView updaten etc.
5	20.01.2022	Usability Test durch Prof. Dr. Wiedling. Abnahme der PVL.

2.2 Implementierungsablauf

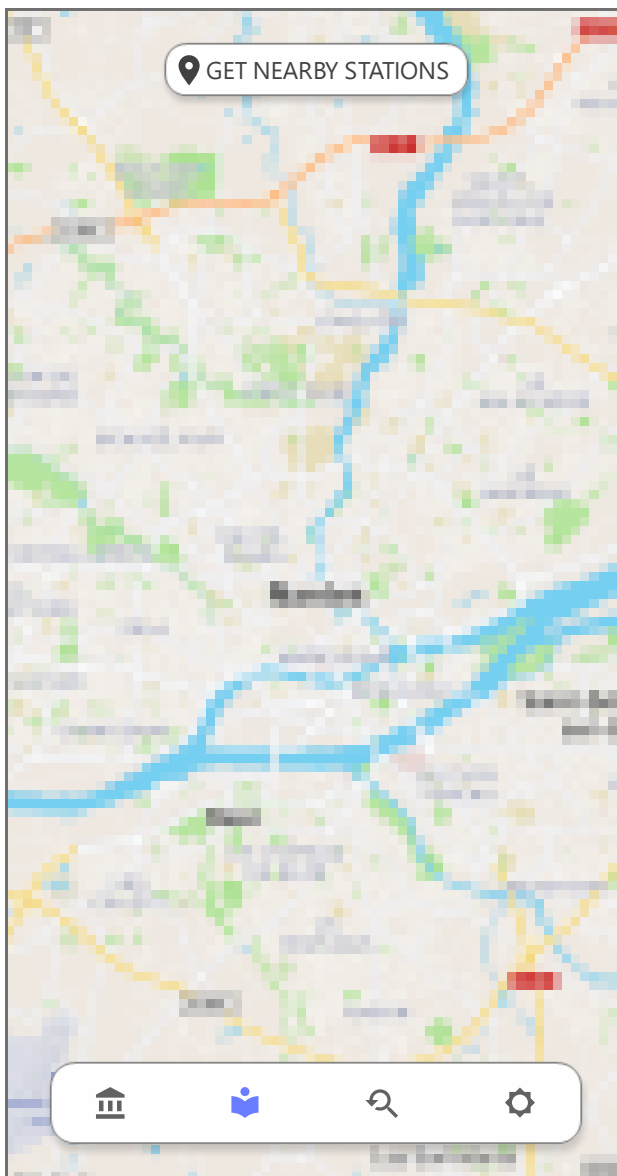
In der folgenden Abbildung ist unser grober Arbeitsablauf skizziert.



3. Design

3.1 Google Map Screen

Von Anfang an war uns relativ klar was wie unsere App aus zu sehen hat. Wir hatten uns direkt dafür entschieden, dass die „Map“ der zentrale Screen für den Benutzer sein soll. Durch die Bottom Navigation Bar kann der User bequem auf alle anderen Elemente zugreifen. Dabei bleibt der Zustand des Screens gespeichert.



Wir haben uns für die Bottom Navigation Bar entschieden, da wir eine einhändige Bedienung der App ermöglichen wollten. Somit konnten wir den Backbutton, welcher meist in der oberen linken Ecke zu finden ist weglassen, da bei den heutigen Handydisplays ein Erreichen der oberen linken Ecke bei einer einhändigen Benutzung im Grunde unmöglich ist. Die in der Leiste anzuwählenden Menüpunkte haben wir durch Symbole ersetzt welche dem Benutzer bildlich vermitteln sollen, auf welchem Screen er ist und auf welchen er geleitet werden würde, wenn er diese anwählt. Dadurch spart der neue Benutzer Einarbeitungszeit, wodurch dieser nicht ewig lesen muss und sein Workflow im Suchen und Finden einer Ladesäule vereinfacht wird.

Bei der Karte selbst haben wir uns für eine eingebettete Google Maps entschieden, da wir uns damit Arbeit sparen konnten und wir keine neue eigene Karte entwickeln mussten. Auch sind die meisten Menschen mit dem Erscheinungsbild und der Bedienung dieser Karte vertraut, was den Einstieg in unsere App ebenso erleichtert wie ein gewisses Vertrauen vermittelt.

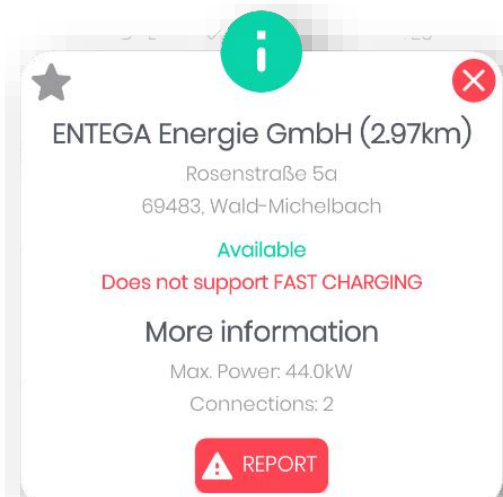
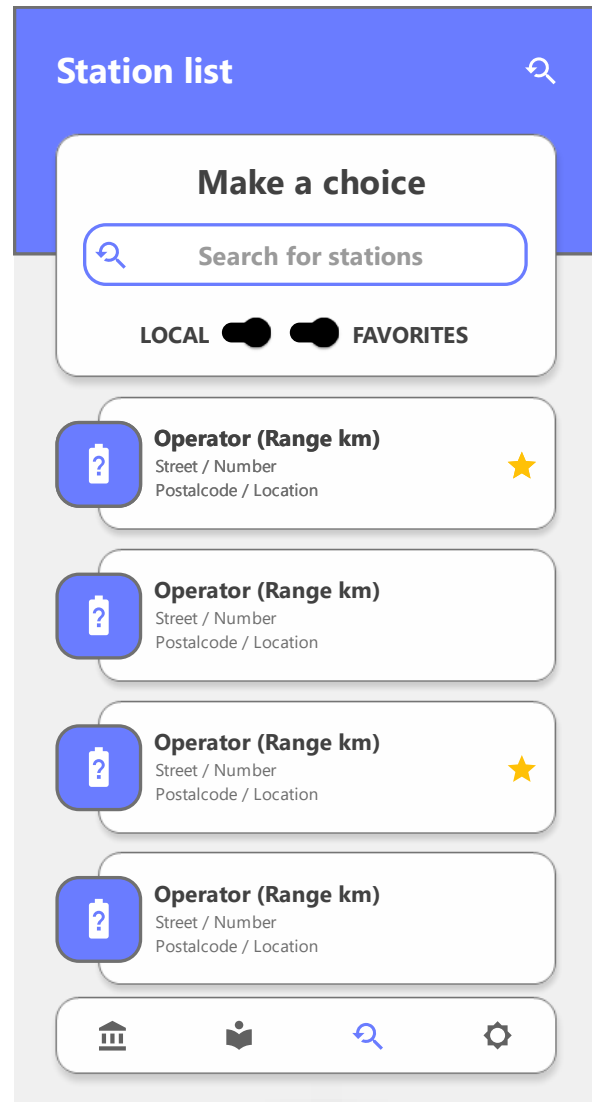
Die Ladesäulen werden im Prototyp durch den Standardmarker von Google Maps, jedoch nur in blau, symbolisiert. Defekte Ladesäulen

werden in Rot und Favoriten in Gelb dargestellt. Diese kann der Nutzer dann anklicken um mehr Informationen, wie Anbieter, Ladegeschwindigkeit oder die genaue Adresse, über die jeweilige Ladesäule in einem Popup zu erhalten. In dieser Detailansicht kann der Nutzer auch einen Defekt an der Ladesäule melden, wodurch diese Säule anderen Nutzern als „defekt“ angezeigt wird.

3.2 Search Screen

Der nächste Reiter in der Bottom Navigation ist der Search Screen. Hier soll es dem Benutzer möglich sein, nach seinen Belieben Ladesäulen zu filtern.

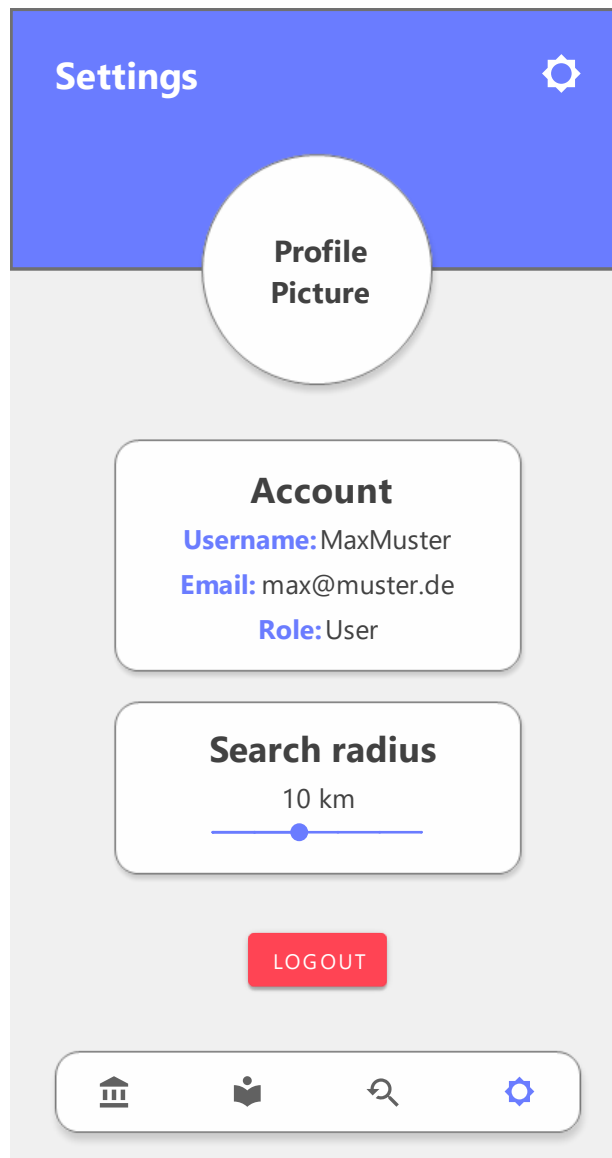
Wir haben alle Ladesäulenattribute filterbar gemacht, was bedeutet, dass der Benutzer nach Betreiber, Straße, Standort, Postleitzahl und Ladegeschwindigkeit filtern kann. Die nächste Ladesäule vom aktuellen Standort wird immer ganz oben gelistet. Wie auf dem Bild zu erkennen, haben wir einen CardView für die zentralen Elemente oben mittig verwendet. In diesem CardView haben wir oben mittig zentriert eine Suchleiste platziert. Direkt unter der Suchleiste findet der Benutzer zwei Schieberegler, mit welchen er einen Lokalen Suchmodus aktivieren kann, wodurch er nur Ergebnisse erhält, die in einem Bereich, den er zuvor in den Einstellungen festgelegt hat angezeigt werden. Als Default Suchradius wurden 10km angesetzt. Mit dem zweiten Schieberegler kann der Benutzer sich nur die Ergebnisse welche auch in seinen Favoriten sind anzeigen lassen. Beide Regler sind unabhängig voneinander nutzbar, d.h. man kann auch nach lokalen Favoriten suchen. Die Ergebnisse werden dem Benutzer direkt darunter in einem Kacheldesign ausgegeben, wodurch die Einzelnen Ladesäulen klar voneinander abgetrennt werden. In den einzelnen Kacheln haben wir links ein Symbol welches dem Nutzer den Status der Ladesäule vermittelt. Damit der Nutzer nicht erst die Jeweilige Säule anklicken muss um die Ladegeschwindigkeit zu erfahren. Auch haben wir dem Symbol unterschiedliche Farben gegeben, damit auch bei schnellerem Scrollen eine Schnellladesäule von einer Normalen unterschieden werden kann oder deutlich wird, ob sich die Ladesäule in den Favoriten befindet. Sonst findet man in der Kachel noch den Betreiber die Relative Entfernung und die genaue Adresse der Ladesäule. Alle Items in der Liste sind anklickbar, wodurch ein Popup ausgelöst wird, welches genauere Informationen über die angewählte Ladesäule ausgibt. Ebenso kann man in diesem Popup einen Defekt melden, oder die Säule zu seinen Favoriten hinzufügen.



3.3 Einstellung Screen

Um eine vertraute Umgebung zu schaffen, haben wir eine Einstellungsseite gemacht. Hier ist es dem Nutzer möglich, Accountinformationen einzusehen, den Suchradius anzupassen und sich aus der App auszuloggen.

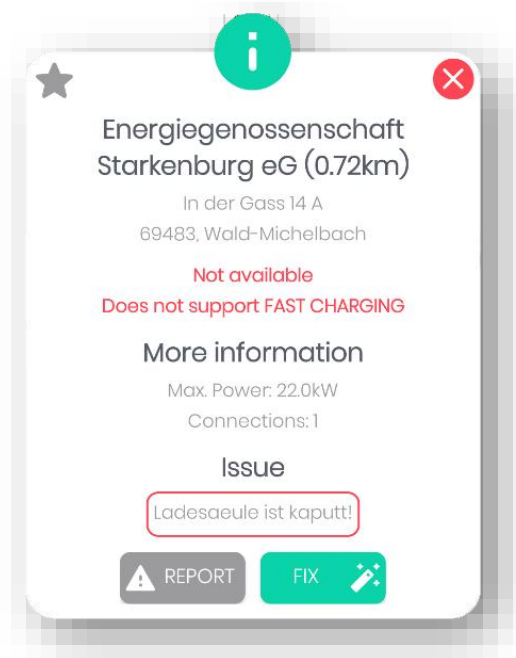
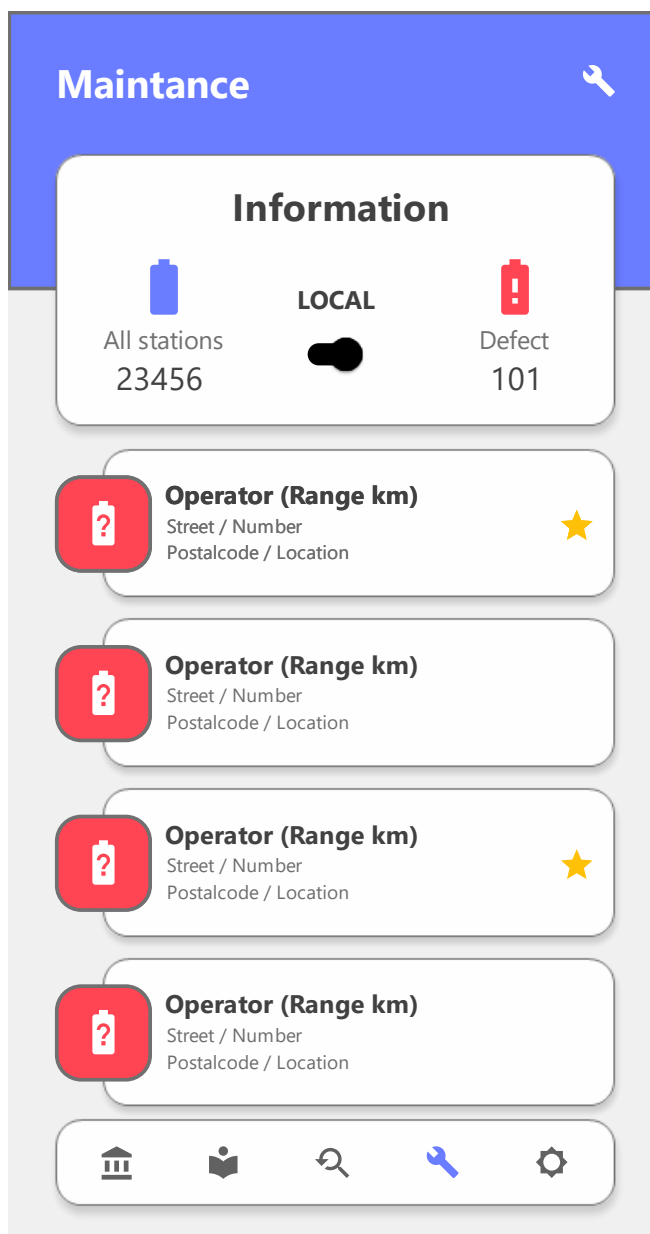
Das Design hierbei ist dabei fast identisch zu allen anderen Screens. Oben mittig ist ein dummy Profilbild zu sehen. Hier könnte man später noch die Funktion eines eigenen Nutzerprofilbildes implementieren. Die nächsten beiden Abschnitte sind wieder mit einem CardView realisiert worden. Im ersten CardView werden die Accountinformationen des Benutzers angezeigt. Zu sehen ist der Username, die E-Mail-Adresse und die Rechteinstufung des Accounts. Bei normalen Benutzern wird hier „User“ angezeigt, bei Service Technikern wird „Dev“ angezeigt. Im CardView darunter kann der aktuelle Suchradius mit Hilfe eines Sliders angepasst werden. Abschließen ist der Logout Button zu sehen.



3.4 Techniker Screen

Bei unserer App ist die Oberfläche des Service Technikers direkt mit in die App integriert. Beim Login werden die Rechte des Benutzers abgefragt. Ist der Benutzer ein Mitarbeiter, so hat er in der Bottom Navigation einen zusätzlichen Reiter der mit einem Schraubenschlüssel zu erkennen ist. Hinter diesem Fragment versteckt sich die Benutzeroberfläche des Technikers.

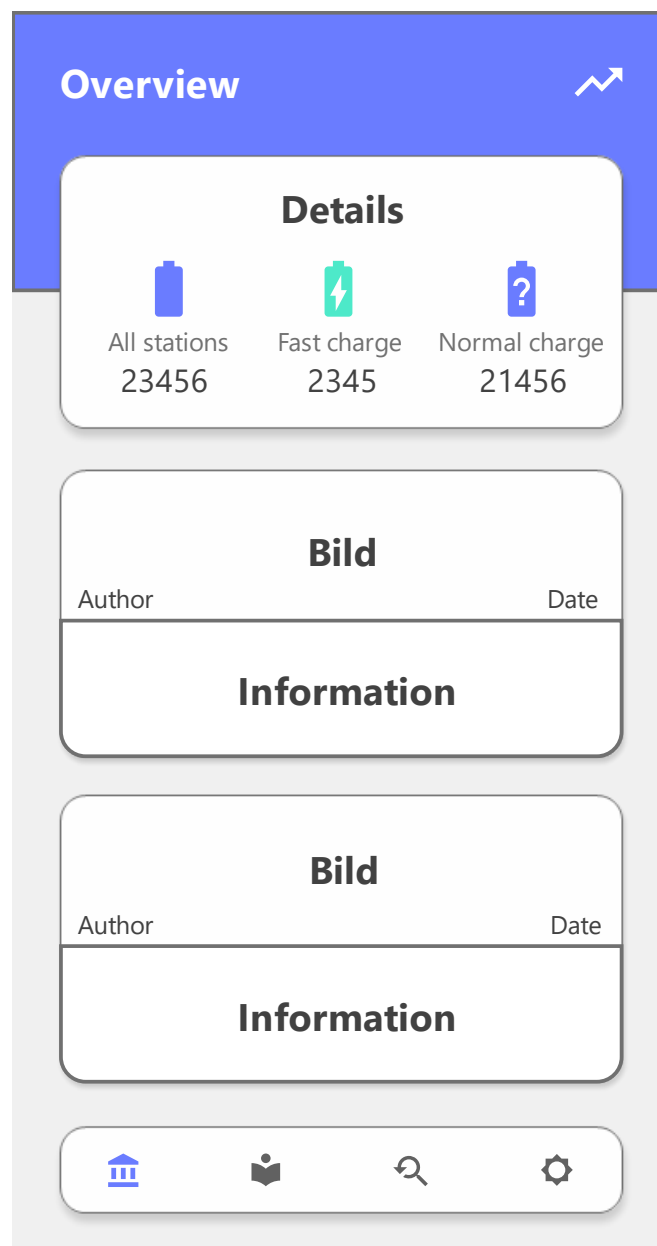
Auf diesem Screen zu sehen ist eine Liste mit allen gemeldeten Ladesäulen. Der Header hat sich soweit verändert, dass nur noch ein „Local“-Schieberegler vorhanden ist. Die Suchleiste wurde ersetzt durch die Anzahl aller Ladesäulen und durch die Anzahl an defekten Ladesäulen. Somit ist es dem Techniker möglich an Hand von seinem Standort nach defekten Ladesäulen zu filtern und somit ein effizientes Arbeiten zu ermöglichen. Auch hier sind die Items in der Liste anklickbar. Der Techniker sieht in dem Popup den Meldegrund und einen Button um die Ladesäule wieder freizugeben.



3.5 Landing Screen

Der Landing Screen soll dem Benutzer eine kurze Übersicht über relevante Informationen geben. Hier werden App Patches und Updates zu den Ladesäulen visuell dargestellt.

Da wir ein einheitliches Design haben, ist der Landing Screen stark angelehnt an die Screens mit RecyclerView. Oben zu sehen sind alle Ladesäulen, alle Schnellladevorrichtungen und alle Ladevorrichtungen mit normaler Ladefunktion. Auch hier wird wieder ein CardView benutzt. Darunter in einem RecyclerView sind die Informationen aufgelistet. Nicht in dem Modell zu sehen ist die Funktion „Mark as read“! Dem Benutzer werden so die Nachrichten nicht erneut angezeigt.



4. Konkrete Umsetzung im ersten Prototyp

4.1 Beginn

Bevor wir mit dem ersten Programmieren der App begonnen haben, haben wir uns damit auseinandergesetzt, was uns an Materialien zur Verfügung steht. Dabei hat sich schnell herausgestellt, dass wir zum Beispiel keine Echtzeit Kommunikation mit den Ladesäulen oder E-Autos aufbauen können, wodurch für uns die Umsetzung vieler Ideen unmöglich war.

Da wir unser Grunddesign der App im Grunde schon fertig hatten, bevor wir mit der Umsetzung in den Prototypen begonnen haben, konnten wir dieses schnell in Android Studio mit Hilfe des XML Editor rekonstruieren.

4.2 Anforderungen, welche wir umsetzen konnten

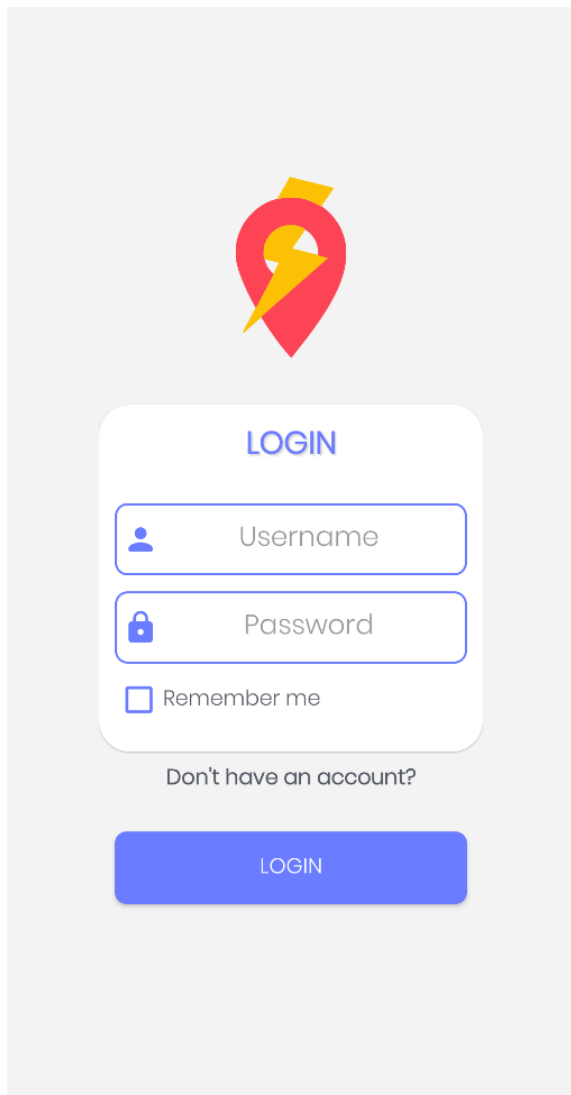
- Probleme melden
- Filtern der Ladestationen
- Ladestationen in der Nähe (Umkreissuche)
- Ladestationen in der Nähe einer gesetzten Position
- Ausgabe aller Information zu einer Ladesäule
- Favoriten speichern
- Defekte Stationen melden
- Freigabe von reparierten Ladesäulen
- Fehlerinformationen zu defekten Stationen
- Persistentes Speichern von Daten

4.3 Anforderungen, welche wir nicht umsetzen konnten

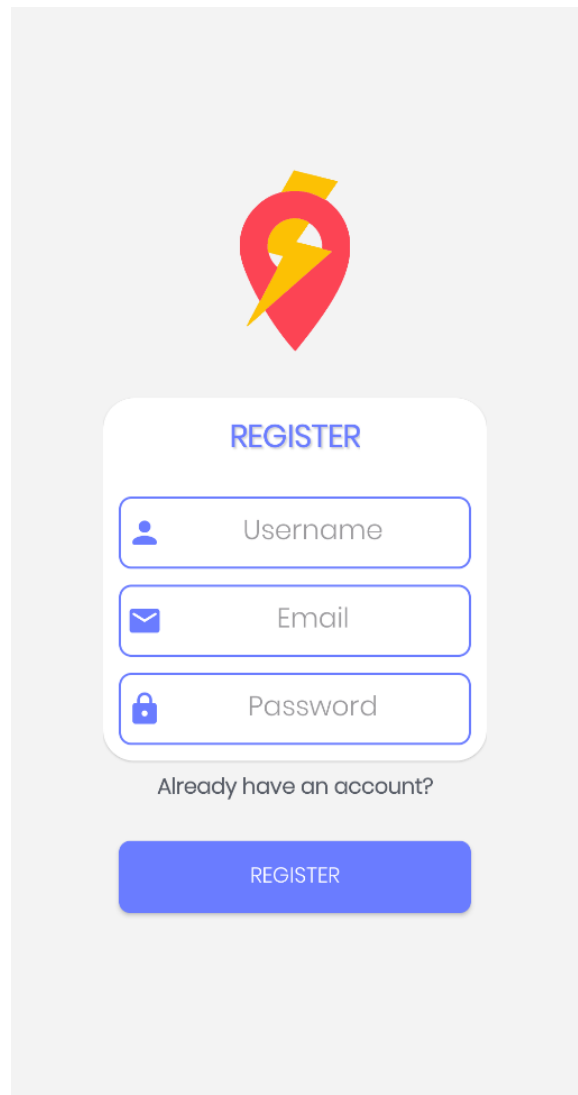
- Freie Plätze an der jeweiligen Ladestation anzeigen, da diese Information nicht in den uns zur Verfügung stehenden Daten inkludiert war, war auch die Umsetzung nicht möglich.
- Für die Kosten pro kWh standen uns auch nicht die Daten zur Verfügung.
- Auch die Informationen über den Ladevorgang wie momentan Geschwindigkeit oder Akkustand des Autos standen uns, durch fehlende Kommunikation mit dem Auto, nicht zur Verfügung.
- Der Routenplaner wäre in der Umsetzung möglich gewesen, jedoch hätte dies die Praktikumsanforderung weit überschritten und den Arbeitsaufwand enorm in die Höhe getrieben, weswegen wir uns doch gegen diesen entschieden hatten.
- Die aktuelle Position des Autos ist wie auch dessen Ladezustand konnten wir nicht umgesetzt, da wir keine Kommunikation zwischen App und dem jeweiligen Auto herstellen konnten.

4.4 Grundlegender App Aufbau

Generell besteht unsere App aus drei Activities, dem Ladebildschirm, der Login-/Registeractivity und der Main Activity. Im Ladebildschirm wird die Liste aller Ladesäulen online abgerufen und in einem Array gespeichert. Der Login Bildschirm soll dem Nutzer ermöglichen, mit Hilfe eines Benutzerkontos, sich bei der App anzumelden. Liegt kein Konto vor, so kann er sich ein Konto erstellen. Die eigentliche Benutzeroberfläche ist in der Main Activity zu finden.



The login screen features a red location pin icon with a yellow lightning bolt at the top center. Below it is a white rounded rectangle with a blue border. Inside this rectangle, the word "LOGIN" is centered at the top in blue. There are two input fields: the first is labeled "Username" with a person icon, and the second is labeled "Password" with a lock icon. Below these fields is a checkbox labeled "Remember me". At the bottom of the white rectangle, the text "Don't have an account?" is displayed. Below the white rectangle is a solid blue button with the word "LOGIN" in white.



The register screen features the same red location pin icon with a yellow lightning bolt at the top center. Below it is a white rounded rectangle with a blue border. Inside this rectangle, the word "REGISTER" is centered at the top in blue. There are three input fields: the first is labeled "Username" with a person icon, the second is labeled "Email" with an envelope icon, and the third is labeled "Password" with a lock icon. Below these fields is the text "Already have an account?". Below the white rectangle is a solid blue button with the word "REGISTER" in white.

4.5 Activities vs. Fragments

Activity	Fragment
Activity is an application component that gives a user interface where the user can interact.	The fragment is only part of an activity, it basically contributes its UI to that activity.
Activity is not dependent on fragment	Fragment is dependent on activity. It can't exist independently.
we need to mention all activity it in the manifest.xml file	Fragment is not required to mention in the manifest file
We can't create multi-screen UI without using fragment in an activity,	After using multiple fragments in a single activity, we can create a multi-screen UI.
Activity can exist without a Fragment	Fragment cannot be used without an Activity.
Creating a project using only Activity then it's difficult to manage	While Using fragments in the project, the project structure will be good and we can handle it easily.
Lifecycle methods are hosted by OS. The activity has its own life cycle.	Lifecycle methods in fragments are hosted by hosting the activity.
Activity is not lite weight.	The fragment is the lite weight.

Quelle: <https://www.geeksforgeeks.org/difference-between-a-fragment-and-an-activity-in-android/>

Im oberen Bild sind die Unterschiede von einer Activity und einem Fragment zu sehen.

Da wir eine Bottom Navigation Bar benutzen und keine Navigation Bar, haben wir uns für die Synergie von Activities und Fragmenten entschieden. Ein Fragment kann nur in Abhängigkeit von einer Activity bestehen, wobei auf einer Activity mehrere Fragmente existieren können. Durch die Benutzung von Fragmenten ist der Datenaustausch wesentlich einfacher zu gestalten. Fragmente haben eine höhere Performance als Activities.

4.6 WebAPI und Datenbank

Bei der Einlese der Ladesäulen haben wir uns für das schreiben einer kleinen API entschieden, da wir mit dieser nicht die Liste mit den Ladesäulen lokal auf dem Handy speichern müssen. Dadurch wird wurde es auch für uns einfacher die Liste der Ladesäulen beim Endnutzer aktuell zu halten ohne für jede neue Säule ein App Update über den jeweiligen Store zu pushen. Auch ermöglicht dies uns Benutzer Konten zu verwalten, sodass zum Beispiel die Favoriten von einem Gerät auf ein Neues übernommen werden, oder das gleiche Konto auf zwei Endgeräten verwendet werden kann. Die App fragt die API bei jedem Starten der App ab. Die Nutzerdaten werden nach dem Login abgerufen. Jegliches Datenformat was von der API zurückgegeben wird erfolgt per JSON. Im Gegensatz zu CSV gibt es für JSON eine Menge Libraries, die JSON Objekte direkt in Java Objekte konvertieren können. Somit legen wir eine Arrayliste von dem Objekt „ChargingStation“ an. Auf die Daten des Objektes kann mit Hilfe von „get“-Funktionen zugegriffen werden. Eine Veränderung der Daten ist nicht notwendig, daher keine „set“-Funktionen. Dies geschieht in der DownloadService Klasse.

Für die Konvertierung wird die Library Gson von Google benutzt. Diese wurde mit Hilfe von Gradle implementiert.

Gson Library Link: <https://mvnrepository.com/artifact/com.google.code.gson/gson/2.7>

Jegliche API-Abfrage läuft bei uns im Hintergrund (async). Dies wurde mit Hilfe der Standardlibrary „CompletableFuture“ realisiert.

Unsere Datenbank läuft auf einem Rootserver als MySQL Datenbank. Da Android Studio eine direkte Verbindung zu einer MySQL Datenbank verbietet (Sicherheitsgründe) waren wir gezwungen uns auch dafür eine WebAPI zu schreiben. Diese WebAPI wandelt http Requests in MySQL Befehle um und gibt die angeforderten Daten als JSON dem Gerät zurück. Diese Daten werden wiederum mit der Gson Library in ein API Response Objekt umgewandelt. Mittels eines Request Codes kann der Status der Abfrage geprüft werden.

Dokumentation der WebAPI: https://api.aurora-theogenia.de/clever_charge/

ChargingStation Klasse: https://api.aurora-theogenia.de/documentation/d2/dae/classde_1_1backxtar_1_1clevercharge_1_1data_1_1_charging_station.html

DownloadService: https://api.aurora-theogenia.de/documentation/dc/d75/classde_1_1backxtar_1_1clevercharge_1_1services_1_1_download_service.html

Loading Screen: https://api.aurora-theogenia.de/documentation/d9/db6/loading_screen_8java_source.html

4.7 Distanz Berechnung

Um die Distanz zwischen zwei Punkten zu berechnen braucht man im zwei dimensional nur den Pythagoras, da wir uns aber im drei Dimensionalen Raum auf einer Kugel befinden wird die Abstands Rechnung deutlich komplexer. Falls wir dies vernachlässigen würden, würden wir die direkte Distanz durch die Kugel hindurch berechnen, welche natürlich kürzer ist als die reelle Distanz, welche man auf der Erde zurücklegen müsste. Um die Exakten Distanz Werte zu berechnen benutzten wir die Haversine Formel, welche direkt die Längen- und Breitengrade des Zwei Punkte nimmt und die Distanz in Abhängigkeit zum Radius der Kugel liefert. Die Berechnung der Distanz erfolgt auf dem

Objekt ChargingStation. Hier erfolgt eine Unterscheidung mit Hilfe von Overloading. Somit kann die Distanz von Ladesäule und dem aktuellen Standort und einem gesetzten Marker berechnet werden.

Berechnung: https://api.aurora-theogenia.de/documentation/d4/df0/charging_station_8java_source.html#l00167

4.8 Favoriten, Defekte und Nutzerdaten

Da wir unsere Nutzerdaten nicht lokal, sondern auf unserem Server speichern, aus bereits genannten Gründen, müssen wir diese natürlich auf beiden Devices aktuell halten. Um die Daten vom Server aufs Handy zu bekommen werden nach dem Login die Userdaten mit Hilfe der API abgefragt und lokal gespeichert. Wenn ein Nutzer auf seinem Endgerät eine neue Ladesäule zu seinen Favoriten hinzufügt, muss dies natürlich dem Server mitgeteilt werden. Dies geschieht bei uns simultan zum Hinzufügen oder entfernen des Favoriten. Nach demselben Prinzip funktioniert auch das Melden einer defekten Ladesäule. Der Prozess der API-Abfrage geschieht im Hintergrund, wieder mit Hilfe von „CompletableFuture“. Um keinen Datenverlust zu erleiden werden alle Informationen auch lokal aktualisiert. Informationen zu Defekten an Ladesäulen können nur von Technikern eingesehen werden.

Favoriten ändern: https://api.aurora-theogenia.de/documentation/d3/d34/popup_service_8java_source.html#l00302

Defekt melden: https://api.aurora-theogenia.de/documentation/d3/de2/report_service_8java_source.html#l00125

Defekt löschen: https://api.aurora-theogenia.de/documentation/d3/d34/popup_service_8java_source.html#l00318

4.9 Filterfunktion im Suchbildschirm

Damit der Suchbildschirm einen Zweck überhaupt erfüllen kann mussten wir eine Funktion schreiben, welche es dem Nutzer ermöglicht die Liste der Ladestationen nach deren Eigenschaften zu filtern. Damit die Suche des RecyclerViews funktioniert, haben wir eine `filterList()` Funktion implementiert. Diese Funktion arbeitet mit Java-Streams. Java-Streams ermöglichen es, eine komplett sortierte / gefilterte Liste zurückzugeben. Dies erfolgt mit der Haus eigenen `filter()` Funktion der Streams. In dieser Funktion wird nach allen filterbaren Eigenschaften in der Objektliste gesucht. Wird ein Objekt gefunden, so wird das Objekt der neuen temporären Liste hinzugefügt. Somit bekommt man am Ende sehr effizient eine neue gefilterte Liste zurück. Wichtig hierbei ist, dass Java Stream mit Hilfe von Lambdas arbeiten. Vorteile von Java Streams ist ihre Effizienz und das Aneinanderreihen mehrerer Suchkriterien. Ein weiterer Vorteil von Java Streams ist die Möglichkeit das Ganze parallel laufen zu lassen. Dies bedeutet, dass eine Filterfunktion auf mehreren Kernen ausgeführt wird. Hierbei wird die zu filternde Liste in x beliebige Stücke aufgeteilt und jeder Kern übernimmt eine Teilliste. Am Ende werden alle gefilterte Teillisten wieder zusammengefügt. In unserem Beispiel haben wir jedoch einen normalen Stream, und keinen `parallelStream` verwendet. Damit der User eine neue Darstellung auf dem Screen erhält, muss jedoch die neue Liste dem Adapter übergeben werden. Dies erfolgt mit Hilfe einer Funktion im Adapter selbst. Anschließend muss dem Adapter noch mitgeteilt werden, dass sich die Daten in der Liste verändert haben. Dies wird mit `notifyDataSetChanged()` realisiert. Dadurch, dass der Nutzer nur eine Suchzeile hat und nicht ein Filtermenu in welchem man für jede Eigenschaft einzeln filtert, haben wir eine Suche die darauf ausgelegt ist so viele Ergebnisse wie möglich zu liefern, da es sich um Oder und keine Und Verknüpfungen bei Mehreren Suchbegriffen handelt. Dadurch wird es zwar schwieriger ein bestimmtes Ergebnis zu finden, doch einfacher überhaupt welche zu erhalten, was bei der

Überschaubaren Zahl der Ladesäulen in Deutschland nicht unbedingt von Nachteil ist. Auch wird da durch das Erlebnis des Nutzers verbessert, da der Suche Bildschirm Übersichtlicher und aufgeräumter ist und seltener Frust aufkommt, wenn bei eingestellten Filtern keine Ergebnisse geliefert werden.

Filterfunktion: <https://api.aurora-theogenia.de/documentation/d1/da2/ search fragment 8java source.html#I00210#>

Update List: <https://api.aurora-theogenia.de/documentation/d1/da2/ search fragment 8java source.html#I00253>

Update Adapter: <https://api.aurora-theogenia.de/documentation/d8/d92/ charging station adapter 8java source.html - I00204>

4.10 RecyclerView

Der RecyclerView ist eine verbesserte Variante des ListViews. Im Gegensatz zum ListView rendert der RecyclerView nur die Items, die momentan auf dem Bildschirm zu sehen sind. Dies ist ein immenser Leistungsvorteil gegenüber dem ListView. Ein RecyclerView beinhaltet immer folgende Punkte:

- Einen Konstruktor für die Initialisierung des Objektes selbst
- Globale Variablen
- Einen eigenen ViewHolder, der die Elemente eines XML Layouts initialisiert.
- Einer onCreateViewHolder() Methode die das Layout eines einzelnen Items vornimmt
- Einer onBindViewHolder() Methode, die Daten in die initialisierten Elemente des Holders schreibt
- Einer getItemCount() Methode, die die Größe der zu verwaltenden Liste zurückgibt

Zusätzlich haben wir noch eingebaut:

- Eine getItemViewType() Methode, die je nach Eigenschaft des Objektes das Design verändert (Favorit, Defekt, etc.)
- Eine updateAdapter() Methode die das setzen einer neuen Liste und gleichzeitig das notifyDataSetChanged() übernimmt.
- Ein RecyclerViewInterface, damit die einzelnen Elemente in der Liste anklickbar sind

Nach demselben Prinzip funktioniert auch die Filterung bei dem Wartungsscreen.

RecyclerView: <https://api.aurora-theogenia.de/documentation/d8/d92/ charging station adapter 8java source.html>

Interface: <https://api.aurora-theogenia.de/documentation/de/daf/ recycler view interface 8java source.html>

onItemClick: <https://api.aurora-theogenia.de/documentation/d1/da2/ search fragment 8java source.html#I00170>

4.11 Google Map

Die GoogleMap haben wir zusätzlich zu der Listenansicht für den Benutzer integriert. Durch ein vertrautes System (Google) bieten wir so dem Benutzer die Möglichkeit visuell Informationen über den Standort der Ladesäulen zu bekommen. Dies geht jedoch mit einigen Problem ein Heer. Durch

die Verwendung eines LocationListeners, muss der Benutzer der App folgende Berechtigungen erteilen:

- ACCESS_FINE_LOCATION
- ACCESS_COARSE_LOCATION

Dies erfolgt bei Start der App. Ohne die Berechtigungen, ist die App nicht zu verwenden und die App schließt sich automatisch. Dies soll Exceptions vorbeugen. Realisiert wird das ganze mit Hilfe von einer Permissions-Abfrage. Ebenso eingebaut ist ein s.g. „permissionListener“. Dieser registriert, wann eine Permission gestattet oder abgelehnt wurde. Nur wenn ein bestimmter REQUEST_CODE zurückgegeben wird (alle Permissions erteilt) wird die App zur weiteren Verwendung freigegeben. Die Map an sich läuft auf einem extra Fragment, dem s.g. „MapFragment“. Dieses Fragment lädt bei initialem Start die Map asynchron. Dieses System synergisiert mit unserem LocationListener. Da der LocationListener immer automatisch die aktuelle Position mit Hilfe von Lat und Lon in einer Globalen Variable speichert, kann der aktuelle Standort immer auf der GoogleMap nachverfolgt werden. Selbst eingebaut auf der GoogleMap wurde ein Button um Ladestationen in der Nähe anzuzeigen. Bild ist oben im Designaufbau zu finden. Die verschiedenen Marker auf der GoogleMap haben wir mit Hilfe von Bitmaps realisiert.

GoogleMap: https://api.aurora-theogenia.de/documentation/da/da2/ map_fragment_8java_source.html

LocationListener: https://api.aurora-theogenia.de/documentation/d4/d79/ main_activity_8java_source.html#I00228

PermissionsListener: https://api.aurora-theogenia.de/documentation/d4/d79/ main_activity_8java_source.html#I00168

Abfrage: https://api.aurora-theogenia.de/documentation/d4/d79/ main_activity_8java_source.html#I00087

4.12 Persistente Daten

Da unsere App überwiegend die Daten online speichert, müssen wir nur sensible Daten lokal auf dem Gerät speichern. Android stellt uns dafür die s.g. „SharedPreferences“ zur Verfügung. Auf dem Gerät zu speichernde Daten werden mit Hilfe von einem Editor in die „SharedPreferences“ geschrieben und mit commit() gespeichert. Wichtig hierbei ist, dass immer ein s.g. Key angegeben werden muss. Dieser Key wird verwendet, um die Daten später zu identifizieren. In unseren „SharedPreferences“ werden nur folgende Daten gespeichert:

- Benutzername
- Benutzerpasswort
- Eingestellter Suchradius

Benutzername und Benutzerpasswort sind notwendig um eine Auto-Login Funktion bieten zu können. Der eingestellte Suchradius hätte man auch auf dem Server hinterlegen können. Diese Daten werden aber nur gespeichert, wenn das „remember me“ Häkchen gesetzt wurde! Nach einem Logout werden alle gespeicherten Daten gelöscht!

Daten speichern: https://api.aurora-theogenia.de/documentation/df/d17/ login_sign_in_fragment_8java_source.html#I00140

Auto-Login: https://api.aurora-theogenia.de/documentation/d9/d7f/ login_8java_source.html#I00060

Logout: https://api.aurora-theogenia.de/documentation/d4/d31/ settings_fragment_8java_source.html#I00149

4.13 Mehrsprachigkeit

Die Sprache in welcher die App mit dem User kommuniziert hängt bei uns mit der Systemsprache des Gerätes zusammen, denn nach dieser wird die Datei ausgewählt, welche die richtigen Ausgaben für die jeweilige Sprache hat. Aus dieser Datei werden dann im Betrieb die anzuzeigenden Texte in die „Textfelder“ eingefügt. Falls beim Start der App festgestellt wird, dass keine Übersetzungsdatei in der Systemsprache vorhanden ist wird die Standardsprache verwendet, welche in unserem Fall Englisch ist. Unsere App ist zum derzeitigen Stand in Deutsch und Englisch komplett übersetzt.

4.14 Landscape Mode

In unserem Fall haben wir das MapFragment in den Landscape Modus konvertierbar gemacht. Wir haben diesen Screen ausgewählt, da auf diesem wenige Elemente zu finden sind. Für den Landscape Mode muss normal ein extra XML Layout angefertigt werden. Möchte man jedoch, dass seine App nur im Portrait Modus verfügbar ist, so kann man dies in der Manifest Datei mit einem Parameter realisieren.

android:screenOrientation="portrait"

5. Usability Test

Die Nutzerfreundlichkeit der App haben wir getestet, indem wir diese Bekannten auf deren Handy gegeben habe und diesen alltäglichen Aufgaben gestellt haben, wie zum Beispiel die nächste Ladesäule heraus zu suchen. Dabei haben wir festgestellt, dass keine unserer Testsubjekte Probleme mit der Bedienung der App hatte und jeder die Aufgaben ohne Hilfestellungen oder Rückfragen erledigen konnte. Auch ohne einen normalen Benutzung Zeitraum von 20 bis 60 Sekunden, für zum Beispiel das heraus suchen der nächsten Ladesäule, zu überschreiten. Als Rückmeldung habe wir auch mitgeteilt bekommen, das durch das helle Pastellige Design die App keine Reiz Überflutung auslöse und wenn mal intensive Farben verwendet werden diese sich immer um etwas wichtiges drehe, wie die Art der Ladesäule.

6. Fazit

Gelernt haben wir im Laufe des Praktikums, dass eine genaue Vorstellung vom Ziel und dessen grober Umsetzung für die tatsächliche Umsetzung sehr hilfreich sind. Denn alle Punkte, welche wir bereits geplant hatten waren in der Umsetzung erheblich schneller erledigt als Entscheidungen, welche wir on the fly entscheiden mussten. Denn diese haben sich manchmal als falsch oder schwieriger herausgestellt, wodurch die Zeit, die wir investieren mussten, erheblich gesteigert wurde. Zum Beispiel hatten wir das Melden einer defekten Station nicht mit Hinterlegung eines Grundes erdacht, wodurch als diese Anforderung aufgetreten ist wir im Hintergrund einiges an fertigem code neu schreiben mussten, bis hin zur Datenbank, in welcher die Speicherung der Defekte geändert werden musste.

Ein weiterer Lerneffekt hat sich ergeben in der Hinsicht, dass wir als Entwickler auch Nutzer sind, welche eine Nutzerfreundliche Oberfläche zu Bedienung haben wollen. Denn die gewählte

Entwicklungsumgebung AndroidStudio und die verwendeten Sprachen Java/ XML sind inzwischen nicht mehr der Stand der Dinge. Denn die anstrengende Arbeit mit XML lässt sich in vielen Alternativen komplett Umgehen und auch hat Java nach gut 25 Jahren nicht die Spitzenposition, die es einst mal hatte. Nach persönlicher Recherche sind wir auf das Framework Flutter gestoßen, welches in unseren Augen für die App Entwicklung auf Android und IOS, was über AndroidStudio nicht möglich ist, die bessere Option ist. Auch der Sprachenwechsel, welcher mit Flutter einhergeht, auf Dart, ist für das Fach NZSE positiv, da diese Sprache C++ ähnliche Syntax benutzt, wodurch der Arbeitsaufwand welcher in das Aneignen einer neuen Sprache verringert wird und mehr Zeit und Fokus auf die wirkliche Nutzerzentrierung gelegt werden kann.