

Dokumentation des Projekts DTSharing
für das Modul “Entwicklung interaktiver
Systeme”

Betreuer

Prof. Dr. Gerhard Hartmann

Prof. Dr. Kristian Fischer

Ngoc-Anh Dang

Jorge Pereira

Studierende

Thomas Friesen

Johannes Kutsch

thomas.friesen@smail.th-koeln.de

johannes.kutsch@smail.th-koeln.de

11092095

11090517

Inhaltsverzeichnis

1. Einleitung	Seite 8
1.1 Nutzungsproblem	Seite 8
1.2 Ziel	Seite 8
2. Proof of Concepts	Seite 9
2.1 Durchführung	Seite 9
2.1.1 Erweitertes Matching	Seite 9
2.1.2 Koordinaten des Benutzers ermitteln	Seite 11
2.1.3 Haltestellen im Umkreis um eine Koordinate ermitteln	Seite 12
2.1.4 Verschlüsseln sensibler Daten	Seite 13
2.1.5 API Zugriff	Seite 14
2.2 Fazit	Seite 15
3. Anforderungsanalyse	Seite 16
3.1 User Profiles	Seite 16
3.1.1 Fazit	Seite 17
3.2 Deskriptive Aufgabenmodellierung	Seite 18
3.2.1 Dauerticketbesitzer	Seite 18
3.2.2 Mitfahrer	Seite 18
3.3 Anforderungen an das System	Seite 19
3.3.1 Funktionale Anforderungen	Seite 19
3.3.2 Organisationale Anforderungen	Seite 20
3.3.3 Qualitative Anforderungen	Seite 20
3.3.4 Technische Anforderungen	Seite 20
3.3.5 Usability Anforderungen	Seite 20
3.4 Style Guide V1.0	Seite 22
4. Design/Testen/Entwicklung	Seite 23
4.1 Level 1: Arbeit neu modellieren	Seite 23
4.1.1 Präskriptive Aufgabenmodellierung	Seite 23
4.1.1.1 Dauerticketbesitzer	Seite 23
4.1.1.2 Mitfahrer	Seite 24
4.1.2 Mockup	Seite 25
4.1.2.1 Login	Seite 25
4.1.2.2 Suchmaske	Seite 26

4.1.2.3 Matches	Seite 26
4.1.2.4 Match Profil	Seite 27
4.1.2.5 Chat	Seite 28
4.1.2.6 Fahrten	Seite 29
4.1.3 Evaluation des Mockup	Seite 29
4.1.4 Prototyp V1.0	Seite 30
4.1.4.1 Login	Seite 30
4.1.4.2 Suchmaske	Seite 31
4.1.4.3 Navigation	Seite 31
4.1.4.4 Matches	Seite 32
4.1.4.5 Match Profil	Seite 32
4.1.4.6 Chatübersicht	Seite 33
4.1.4.7 Chat	Seite 33
4.1.5 Evaluation des Prototyp V1.0	Seite 34
4.1.6 Prototyp V1.1	Seite 35
4.1.6.1 Navigation	Seite 35
4.1.6.2 Eigene Fahrten	Seite 36
4.1.6.3 Matches	Seite 36
4.1.7 Evaluation des Prototyp V1.1	Seite 37
4.2 Level 2: Screen Design Standards	Seite 38
4.2.1 Screen Design Standards	Seite 38
4.2.1.1 Cards	Seite 38
4.2.1.2 Date-Separator	Seite 39
4.2.2 Style Guide V2.3	Seite 40
4.2.3 Prototyp V2.0	Seite 42
4.2.3.1 Login	Seite 42
4.2.3.2 Suchmaske	Seite 43
4.2.3.3 Matches	Seite 43
4.2.3.4 User Profil	Seite 44
4.2.3.5 Chatübersicht	Seite 44
4.2.3.6 Chat	Seite 45
4.2.3.7 Fahrten	Seite 45
4.2.4 Evaluation des Prototyp V2.0	Seite 46
4.2.5 Prototyp V2.1	Seite 46

4.2.5.1 Verbindungen	Seite 47
4.2.6 Evaluation des Prototyp V2.1	Seite 47
4.3 Level 3: Detailed User Interface Design	Seite 48
4.3.1 Prototyp V3.0	Seite 48
4.3.1.1 Suchmaske - Umkreissuche	Seite 48
4.3.1.2 Suchmaske - Picker	Seite 49
4.3.1.3 Matches - Dialoge	Seite 50
4.3.1.4 Chat - Bewertung	Seite 51
4.3.1.5 Menüs	Seite 52
4.4 Fazit	Seite 52
5. Datenstrukturen	Seite 53
5.1 Datenstruktur - Client	Seite 53
5.2 Datenstruktur - Server	Seite 54
5.2.1 GTFS-Daten	Seite 54
5.2.2 Benutzerdaten/Eingetragene Fahrten	Seite 54
5.3 Datenübertragung	Seite 55
5.3.1 Fahrten zwischen zwei Bahnhöfen ermitteln	Seite 55
5.3.2 Eine Fahrt eintragen	Seite 57
5.4 Datenschutz	Seite 58
6. Modellierung der Architekturmerkmale	Seite 59
6.1 Ressourcen und Topics	Seite 59
6.1.1 synchrone Kommunikation	Seite 59
6.1.2 asynchrone Kommunikation	Seite 60
6.2 Skizzierung der Anwendungslogik	Seite 61
6.2.1 Client	Seite 61
6.2.1.1 Umkreissuche	Seite 61
6.2.1.2 Verlauf	Seite 62
6.2.1.3 Durchschnittliche Bewertung	Seite 64
6.2.2 Server	Seite 65
6.2.2.1 Trips ermitteln	Seite 65
6.2.2.2 Erweitertes Matching	Seite 68
6.2.2.3 Suchagent	Seite 70
7. Anhang	Seite 71
7.1 deskriptive Aufgabenmodellierung	Seite 71

7.1.1 deskriptive HTA Dauerticketbesitzer	Seite 71
7.1.2 deskriptive HTA Mitfahrer	Seite 72
7.2 präskriptive Aufgabenmodellierung	Seite 73
7.2.1 präskriptive HTA Dauerticketbesitzer	Seite 73
7.2.2 präskriptive HTA Mitfahrer	Seite 74
7.3 User Profiles	Seite 75
7.3.1 Merkmale	Seite 75
7.3.2 Dauerticketbesitzer	Seite 75
7.3.3 Mitfahrer	Seite 77
7.3.4 Kundenservice	Seite 79
7.4 Architekturdiagramm	Seite 80
7.5 präskriptives Kommunikationsmodell	Seite 80
7.6 ER-Diagramm Client	Seite 81
7.7 ER-Diagramm Server	Seite 82
7.8 Stakeholderanalyse	Seite 83
7.9 Ressourcen	Seite 85

Abbildungsverzeichnis

Proof of Concepts

Abbildung 1: Erweitertes Matching	Seite 9
Abbildung 2: GTFS stops	Seite 10
Abbildung 3: GTFS stop_times	Seite 10
Abbildung 4: Ermitteln der Koordinaten	Seite 11
Abbildung 5: Sortierte Umkreissuche	Seite 12
Abbildung 6: Verschlüsselung Sensibler Daten	Seite 13
Abbildung 7: API Zugriff	Seite 14
Abbildung 8: XML Schema VRS	Seite 14

Mockup

Abbildung 9: Login	Seite 25
Abbildung 10: Kennwort vergessen	Seite 25
Abbildung 11: Registrieren	Seite 25
Abbildung 12: Suchmaske	Seite 26
Abbildung 13: Matches	Seite 26

Abbildung 14: Profil	Seite 27
Abbildung 15: Chats	Seite 28
Abbildung 16: Chat	Seite 28
Abbildung 17: Fahrten	Seite 29
Prototyp V1.0	
Abbildung 18: Login	Seite 30
Abbildung 19: Passwort vergessen	Seite 30
Abbildung 20: Registrieren	Seite 30
Abbildung 21: Suchmaske	Seite 31
Abbildung 22: Navigation	Seite 31
Abbildung 23: Matches	Seite 32
Abbildung 23: Match Profil	Seite 32
Abbildung 24: Chatübersicht	Seite 33
Abbildung 25: Chat	Seite 33
Prototyp V1.1	
Abbildung 26: Navigation	Seite 35
Abbildung 27: Eigene Fahrten	Seite 36
Abbildung 28: Matches	Seite 36
Screen Design Standards	
Abbildung 29: Cards	Seite 38
Abbildung 30: Date-Separator	Seite 39
Prototyp V2.0	
Abbildung 31: Login	Seite 42
Abbildung 32: Passwort vergessen	Seite 42
Abbildung 33: Registrieren	Seite 42
Abbildung 34: Suchmaske	Seite 43
Abbildung 35: Matches	Seite 43
Abbildung 36: User Profil	Seite 44
Abbildung 37: Chatübersicht	Seite 44
Abbildung 38: Chat	Seite 45
Abbildung 39: Fahrten	Seite 45
Prototyp V2.1	
Abbildung 40: Verbindungen	Seite 47
Prototyp V3.0	

Abbildung 41: Umkreissuche	Seite 48
Abbildung 42: Picker - Datum	Seite 49
Abbildung 43: Picker - Uhrzeit	Seite 49
Abbildung 44: Picker - Ticket	Seite 49
Abbildung 45: Dialog - Eintragen	Seite 50
Abbildung 46: Dialog - Mitfahren	Seite 50
Abbildung 47: Dialog - Bewertungen	Seite 51
Abbildung 48: Menü - Suchmaske	Seite 52
Abbildung 49: Menü - Chat	Seite 52
Anhang	
Abbildung 50: deskriptive HTA Dauerticketbesitzer	Seite 71
Abbildung 51: deskriptive HTA Mitfahrer	Seite 72
Abbildung 52: präskriptive HTA Dauerticketbesitzer	Seite 73
Abbildung 53: präskriptive HTA Mitfahrer	Seite 74
Abbildung 54: Architekturdiagramm	Seite 80
Abbildung 55: präskriptives Kommunikationsmodell	Seite 80
Abbildung 56: ER-Diagramm Client	Seite 81
Abbildung 57: ER-Diagramm Server	Seite 82

Tabellenverzeichnis

Tabelle 1: Style Guides V1.0	Seite 22
Tabelle 2: Style Guides V2.3	Seite 40
Tabelle 3: Ressourcen Übersicht	Seite 59
Tabelle 4: Topics	Seite 60
Tabelle 5 : Merkmale	Seite 75
Tabelle 6: Studierender	Seite 75
Tabelle 7: wissenschaftlicher Mitarbeiter einer Hochschule	Seite 76
Tabelle 8: Angestellter mit Jobticket	Seite 76
Tabelle 9: Auszubildender mit Jobticket	Seite 77
Tabelle 10: Auszubildender ohne Jobticket	Seite 77
Tabelle 11: Angestellter ohne Jobticket	Seite 78
Tabelle 12: Mittelloser	Seite 78
Tabelle 13: Kundenservice	Seite 79

Tabelle 14: Stakeholderanalyse

Seite 84

Tabelle 15: Ressourcen

Seite 85

1. Einleitung

1.1 Nutzungsproblem

Zu bestimmten Zeiten bietet das Dauerticket die Möglichkeit eine zusätzliche Person kostenlos mit der Bahn mitzunehmen. Momentan gibt es außerhalb von regionalen Facebook-Gruppen und Regionalen Foren keine Möglichkeit sich mit fremden Personen zusammenzuschließen um das Ticket gemeinsam zu nutzen. Die Suche nach potentiellen Partnern gestaltet sich also sehr mühselig, weshalb die Möglichkeit eine zusätzliche Person kostenlos mitzunehmen oft ungenutzt bleibt.

1.2 Ziel

Das Ziel des Projektes ist es, eine Plattform zu schaffen, welche es ermöglicht Kontakt zwischen einander unbekannten Benutzern herzustellen, deren Route ganz oder teilweise miteinander übereinstimmt und so die gemeinsame Nutzung eines bereits vorhandenen Dauertickets ermöglicht. Die Nutzung dieser Plattform soll die Kommunikation zwischen Dauerticketbesitzern und deren potentiellen Mitfahrern erleichtern und das Sharing von Dauertickets soll deshalb zunehmen.

2 Proof of Concepts

Nachfolgend wird die Durchführung der im Konzept erstellten Proof of Concepts beschrieben.

2.1 Durchführung

2.1.1 Erweitertes Matching

The screenshot shows a POSTMAN interface with the following details:

- Method:** GET
- URL:** http://localhost:3000/matches/offers/4520024052016/3/6
- Authorization:** No Auth
- Headers:** (1)
- Body:** (Pretty, Raw, Preview, JSON) - Contains a JSON response with two objects. Each object has fields: _id, user_id, trip_id, unique_trip_id, departure_sequence_id, and target_sequence_id. The first object's target_sequence_id is 7, and the second object's target_sequence_id is 3.
- Status:** 200 OK Time: 18 ms

```

1. TYPE
2. TRIP_ID
3. DATE
4. DEPARTURE_SEQUENCE_ID
5. TARGET_SEQUENCE_ID
  
```

```

1. _id: "574438913441f47269eb7673",
2. user_id: 12345,
3. trip_id: 45200,
4. unique_trip_id: 4520024052016,
5. departure_sequence_id: 1,
6. target_sequence_id: 7
7. ,
8. {
9.   _id: "574438b83441f47269eb7675",
10.  user_id: 12346,
11.  trip_id: 45200,
12.  unique_trip_id: 4520024052016,
13.  departure_sequence_id: 3,
14.  target_sequence_id: 7
15. }
16. ]
  
```

Abbildung 1: Erweitertes Matching

Das Matching, welches für den Rapid Prototypen entwickelt wurde, wird im erweiterten Matching um die Funktionalität des Zwischeneinstieges erweitert. In Abbildung 1 dargestellt ein Ausschnitt eines GET-Requests mit zugehöriger Response an die URI:

```
/matches/:type/:unique_trip_id/:departure_sequence_id/:target_sequence_id
```

Um an die benötigten Informationen zu kommen ist es nötig auf zwei GTFS Datenbanken zuzugreifen. Zum einen die Datenbank `stops`¹, über welche man die `stop_id` der eingetragenen Start- und Zielhaltestelle ermittelt. Mit dieser zuzüglich der vorangegangenen gewählten Uhrzeit erhält man aus der Datenbank `stop_times`² die `trip_id` der Fahrt.

```
|> db.stops.findOne()
{
  "_id" : ObjectId("57240e30d81916eda732ebe7"),
  "stop_id" : "1",
  "stop_code" : "",
  "stop_name" : "Köln, Heumarkt",
  "stop_desc" : "",
  "stop_lat" : "50.935714",
  "stop_lon" : "6.960010",
  "zone_id" : "",
  "stop_url" : "",
  "location_type" : "0",
  "parent_station" : "",
  "stop_timezone" : ""
}
```

¹Abbildung 2: GTFS stops

```
|> db.stop_times.findOne()
{
  "_id" : ObjectId("57240e30d81916eda732ebea"),
  "trip_id" : "1",
  "arrival_time" : "14:20:00",
  "departure_time" : "14:20:00",
  "stop_id" : "8467",
  "stop_sequence" : "26",
  "stop_headsign" : "",
  "pickup_type" : "0",
  "drop_off_type" : "0",
  "shape_dist_traveled" : ""
}
```

²Abbildung 3: GTFS stop times

Die `unique_trip_id` wird aus der `trip_id` und dem zuvor gewählten Datum zusammengesetzt und bildet somit einen für diesen Trip und Tag einzigartigen Indikator. Als letzter Schritt muss die `departure_sequence_id` und `target_sequence_id` des Suchenden größer|gleich bzw kleiner|gleich der des Anbietenden sein. Die verbleibenden Ergebnisse werden als Matches an den Client zurückgegeben.

2.1.2 Koordinaten des Benutzers ermitteln

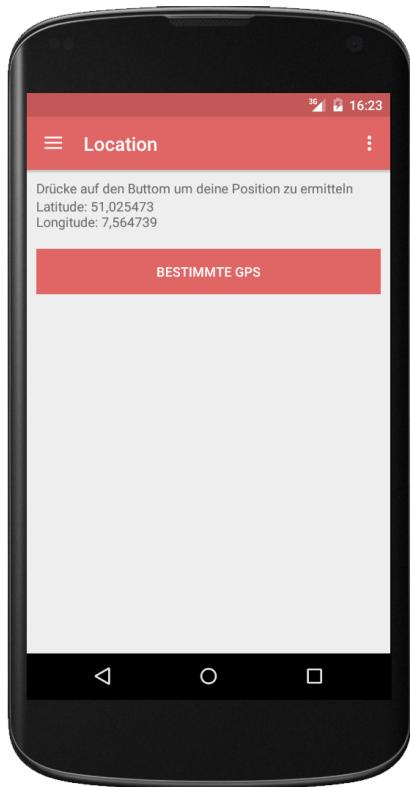


Abbildung 4: Ermitteln der Koordinaten

Bei der Ermittlung der Koordinaten des Benutzers wird die Android Framework Location API `android.location` genutzt. Google empfiehlt die Verwendung der Google Location Services API `com.google.android.gms.location`, da diese Akkurate, Energiesparender und einfacher zu nutzen sei, allerdings werden bei der Nutzung dieses Dienstes alle Geolocation Daten an Google gesendet, womit nicht jeder Benutzer einverstanden ist. Ein weiterer Punkt, der für die Nutzung des Android Framework und gegen die Nutzung der Google API spricht, ist, dass DTSharing die Koordinaten nur für die Umkreissuche benötigt und somit eine kurze Update der Location ausreichend ist.

Da ein Update der Koordinaten bis zu 10 Sekunden in Anspruch nehmen kann und der Benutzer unter Umständen die Umkreissuche sofort wahrnehmen möchte werden zwei Verfahren der Ermittlung von Koordinaten kombiniert.

Zum einen wird die letzte bekannte Position des Benutzers ausgelesen und für die Umkreissuche verwendet. Die Genauigkeit dieses Vorgehens hängt davon ab, wann das Smartphone des Benutzers das letzte mal den Standortdienst in Anspruch genommen hat und wie weit der Benutzer sich von diesen Koordinaten entfernt hat, kann jedoch unter idealen Voraussetzungen bereits ein befriedigendes Ergebnis liefern. Darauffolgend wird ein Update der Koordinaten aufgerufen. Ist dieser Vorgang abgeschlossen werden die neuen Koordinaten nochmals für die Umkreissuche verwendet und die Array-Liste somit erneuert. Abschließend wird die Bestimmung des Standortes angehalten um den Batterieverbrauch zu reduzieren.

2.1.3 Haltestellen im Umkreis um eine Koordinate ermitteln

The screenshot shows a POSTMAN interface with a GET request to `http://localhost:3000/stations?radius=2000&lat=50.935778&lon=6.947691`. The response body is a JSON array containing three objects:

```

1 [
2   {
3     "latitude": 50.935778,
4     "longitude": 6.947691,
5     "distance": 0,
6     "key": "Köln, Neumarkt"
7   },
8   {
9     "latitude": 50.934182,
10    "longitude": 6.945084,
11    "distance": 255,
12    "key": "Köln, Mauritiuskirche"
13  },
14  {
15    "latitude": 50.939981,
16    "longitude": 6.950451,
17    "distance": 506
}

```

Abbildung 5: Sortierte Umkreissuche

Für die Umkreissuche wird die Open Source Bibliothek *Geolib*¹ von *Manuel Bieh*² verwendet, da diese bereits alle für die Umkreissuche von DTSharing benötigten Funktionalitäten, wie das Ermitteln aller Einträge, welche sich in einem bestimmten Radius um eine Koordinate befinden und das Sortieren aller Einträge nach der Distanz zu einer bestimmten Koordinate, bietet.

In Abbildung 5 dargestellt ein Ausschnitt eines GET-Requests mit zugehöriger Response an die URL:

```
/stations?radius=integer&lat=double&lon=double
```

Um an die benötigten Informationen zu kommen ist es nötig auf die GTFS Datenbanken `stops`^{Abbildung 2} zuzugreifen. Diese beinhaltet alle Haltestellen samt ihren Latitude (Längengrad) und Longitude (Breitengrad) Daten. Als erstes werden alle stops in einer `forEach` Schleife durchlaufen und bei jedem Eintrag wird durch die Funktion `geolib.isPointInCircle(object point, object center, integer radius)`, welche true oder false liefert, überprüft, ob sie sich in einem bestimmten Radius zum Benutzer befinden. Da dieses Einträge noch nicht nach der Distanz zum Standort den Benutzers Sortiert sind, wird im nächsten Schritt durch die Funktion `geolib.orderByDistance(object point, object mixedCoords)` ein Objekt erzeugt in dem alle Einträge nach Distanz sortiert aufgelistet werden.

¹ <https://github.com/manuelbieh/Geolib> (27.05.2016)

² <https://www.manuel-bieh.de/ueber-mich/zur-person.html> (27.05.2016)

2.1.4 Verschlüsseln Sensibler Daten

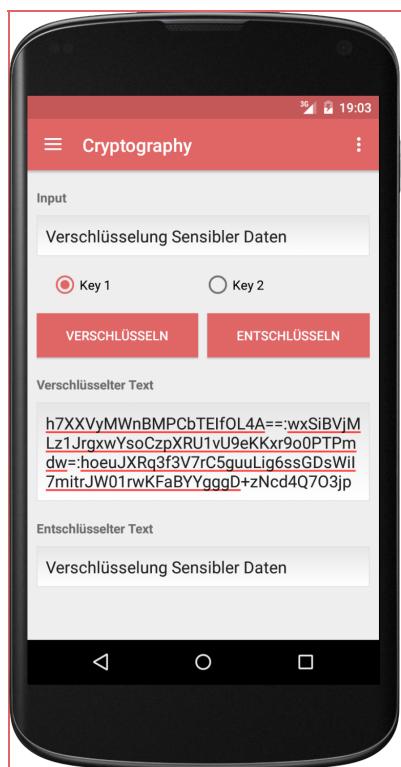


Abbildung 6: Verschlüsselung Sensibler Daten

Zum Verschlüsseln Sensibler Daten wurde die Standalone Java Klasse namens *Tozny*³ gewählt, welche Open Source ist. Diese bietet verschiedene Vorteile. Zum einen lässt sie sich, da es sich um eine Standalone Java Klasse handelt, leicht in Projekte einbinden. Zum anderen ist die daraus resultierende Verschlüsselung sehr sicher. In ihrem *Blog*⁴ gehen sie auf typische Fehler, die bei der Kryptographie gemacht werden ein und warnen davor, dass unsichere und falsche Verfahren bei der Suche nach einer passenden Kryptographielösung stärker bewertet und geteilt werden als richtige. Tozny eignet sich als Kryptographielösung für DTSharing, da es dadurch, dass mit cipherblocks gearbeitet wird, kompatibel mit Strings ist. Zum anderen wird ein Initialization Vector (IV) verwendet, welcher jedes mal Zufällig generiert wird und bei jeder Verschlüsselung einfließt. Somit erhält man trotz gleichem Klartext und Key andere Verschlüsselungen.

Durch ein sogenanntes padding wird erreicht, dass alle Blöcke gleich lang sind. Auch wenn der zu verschlüsselnde Text kurz ist.

Durch `AesCbcWithIntegrity.SecretKeys key = AesCbcWithIntegrity.generateKey();` wird ein zufälliger Key erzeugt. Dieser kann, sofern er verschickt oder gesichert werden muss, in einen String umgewandelt werden. Durch den Aufruf der Funktion

```
AesCbcWithIntegrity.CipherTextIvMac cipherTextIvMac =
    AesCbcWithIntegrity.encrypt("some text", key);
```

kann ein String verschlüsselt werden. Auch dieser kann, sofern er verschickt oder gesichert werden muss, durch `String cipherTextString = cipherTextIvMac.toString()` in einen String umgewandelt werden. Möchte man diesen nun wieder entschlüsseln muss über den Konstruktor die Klasse wiederhergestellt und anschließend entschlüsselt werden.

```
CipherTextIvMac cipherTextIvMac = new CipherTextIvMac (cipherTextString);
String plainText = AesCbcWithIntegrity.decryptString(cipherTextIvMac, key);
```

³ <https://github.com/tozny/java-aes-crypto> (27.05.2016)

⁴ <http://tozny.com/blog/encrypting-strings-in-android-lets-make-better-mistakes/> (27.05.2016)

2.1.5 API Zugriff

The screenshot shows a POSTMAN interface with a GET request to `http://localhost:3000/test/vrsapi?stop=Gummersbach_Bf`. The response body is an XML document containing information about two stops:

```

<?xml version="1.0" encoding="iso-8859-15"?>
<Response>
  <ObjectInfo>
    <Objects>
      <Stop ID="3589">
        <ASSID>3883</ASSID>
        <NumberID NumberRange="VRSNr">48101</NumberID>
        <NumberID NumberRange="GlobalID">de:5374:48101</NumberID>
        <Name Type="Synonym">Gummersbach Bahnhof</Name>
        <Name>Gummersbach Bf</Name>
        <Coordinate srsName="urn:adv:crs:ETRS89_UTM32">399443,5653364</Coordinate>
        <Class>Stop</Class>
        <Group ID="1402" Type="City"></Group>
        <Group ID="6328" Type="District"></Group>
        <InfoMessages></InfoMessages>
      </Stop>
      <Stop ID="3580">
        <ASSID>3874</ASSID>
        <NumberID NumberRange="VRSNr">48501</NumberID>
        <NumberID NumberRange="GlobalID">de:5374:48501</NumberID>
        <Name Type="Synonym">Dieringhausen Bahnhof</Name>
      </Stop>
    </Objects>
  </ObjectInfo>
</Response>

```

Abbildung 7: API Zugriff

Der VRS bietet zwei Möglichkeiten Fahrplandaten zu beziehen. Zum einen die Nutzung der ~Monatlich aktualisierten GTFS Daten zum anderen der Zugriff auf die XML-API. Diese nimmt Anfragen in Form von XML-Request entgegen und liefert als Antwort eine XML-Response.

In Abbildung 7 dargestellt ein Ausschnitt eines GET-Requests mit zugehöriger Response an die URL:

```
/test/vrsapi?stop=String
```

Die XML-Request an die VRS API ist nach einem bestimmten Schema aufgebaut, welches in der Schnittstellenbeschreibung des VRS beschrieben ist.

```

<?xml version="1.0" encoding="ISO-8859-15"?>
<Request>
  <ObjectInfo>
    <ObjectSearch>
      <String>Gummersbach_Bf</String>
      <Classes>
        <Stop/>
        <Address/>
        <POI/>
      </Classes>
    </ObjectSearch>
    <Options>
      <Output>
        <SRSName>urn:adv:crs:ETRS89_UTM32</SRSName>
      </Output>
    </Options>
  </ObjectInfo>
</Request>

```

Ein Schema wird in Abbildung 8 repräsentiert. `<String> ... </String>` stellt den Suchbegriff dar. Durch die drei Klassen `<Stop/><Address/><POI/>` wird der Suchraum angegeben/erweitert. Wobei Stop für die Haltestellen, Address für Adressen und POI für Point of Interest stehen.

Abbildung 8: XML Schema VRS

2.2 Fazit

Die Umsetzung der Proof of Concepts ist im großen und ganzen unproblematisch verlaufen. Die Umkreissuche und das Verschlüsseln Sensibler Daten setzt auf Open Source Bibliotheken, da diese bereits die von DTSharing benötigten Funktionalitäten aufweisen. Dadurch das diese von ihrer Community validiert wurden ist eine zuverlässige Funktionalität dieser Bibliotheken gegeben. Die Ermittlung der Koordinaten wird über die Android Framework API vorgenommen und verzichtet somit auf den externen Dienst der Google API, da die Vorteile die diese bietet für DTSharing nicht relevant sind. Das Erweiterte Matching sollte aufgrund der etablierten Datenstruktur der GTFS Daten zuverlässig funktionieren setzt in diesem Stadium jedoch bereits korrekt ermittelte Daten voraus, welche sich aus den beiden vorangegangen Schritten ergeben, welche für dieses Proof of Concept nicht implementiert wurden. Ein Zugriff auf die API des VRS konnte ohne größere Probleme durchgeführt werden, allerdings wurde im Laufe der Projektentwicklung festgestellt das dieser für die aktuellen Funktionen nicht benötigt wird. Sollten weitere Funktionalitäten wie etwa ein Verspätungsalarm implementiert werden und somit Echtzeitdaten gebraucht werden kann jedoch auf die Erfahrungen die bereits gesammelt wurden zurückgegriffen werden.

3. Anforderungsanalyse

Die Anforderungsanalyse ist die erste Prozess-Stufe des Usability Engineering Lifecycles. In dieser Stufe wird die aktuelle Situation der Aufgabenbewältigung ermittelt und analysiert. Dazu werden in User-Profiles, welche auf der Stakeholderanalyse basieren, die wesentlichen Merkmale der Benutzer herausgearbeitet. Im nächsten Schritt werden die Aufgaben dieser Benutzer in einer deskriptiven Aufgabenmodellierung dargestellt. Als Methode für die Aufgabenmodellierung wurde ein Hierarchische Task Analyse gewählt. Eine HTA erlaubt es komplexe Aktivitäten durch eine Dekomposition in einzelne Schritte Detailiert zu analysieren und ist somit gut dafür geeignet komplexe Abläufe darzustellen. Auf die im Lifecycle vorgesehene Ausführliche Analyse der Hard- und Software-Restriktionen und Möglichkeiten wird verzichtet, da diese bereits zu genüge in der Analyse der User-Profiles betrachtet werden. Auf Basis dieser Artefakte werden dann erste Anforderungen an das System gestellt, welche im Verlauf der gesamten Entwicklung, Aufgrund neuer Erkenntnisse, iterativ überarbeitet wurden. Anschließend wurden erste Style Guides erstellt, welche ebenfalls im Verlauf der gesamten Entwicklung iterativ überarbeitet wurden.

3.1. User Profiles

Bei der Erstellung der User Profiles wurden als erstes sinnvolle Merkmale für die User-Profiles spezifiziert. Bei der Auswahl der Merkmale wurde sich stark an den Vorschlägen aus dem "Draft zum kleinen Handbuch der Mensch-Computer Interaktion"⁵ von Prof. Dr. Hartmann orientiert, welche auf die für den Kontext relevanten Merkmale reduziert wurden. Nachdem diese Merkmale feststanden wurden Daten zu den Benutzern erhoben. Die für die User-Profiles benötigten Daten konnten größtenteils aus der Stakeholderanalyse (siehe Punkt 7.8.) ermittelt werden. Diese Daten wurden dann in Gesprächen mit Stakeholdern verifiziert und wenn nötigt ergänzt. Anschließend wurden die einzelnen User-Profiles generalisiert. Wie bereits in der Stakeholderanalyse konnten die einzelnen User-Profiles den Gruppen Dauerticketbesitzer, Mitfahrer und Kundenservice zugeordnet werden. Die Merkmale und User Profiles sind im Anhang unter Punkt 7.3. aufgelistet.

⁵ Prof. Dr. Gerhard Hartmann: Draft zum kleinen Handbuch der Mensch-Computer Interaktion, 12.02.2013, Seite 370

3.1.1. Fazit

Die Analyse der User-Profiles ergab, dass bei den Benutzergruppen der Dauerticketbesitzer und Mitfahrer, zwei Motivationen ausschlaggebende für die Mitnahme bzw. Mitfahrt sind. Die gemeinsame Reise und die vergünstigte Reise bzw. die Möglichkeit durch die Mitnahme etwas Geld dazuzuverdienen. Als Motivation zur Nutzung von DTSharing spielt die Effizienz dieser Lösung im Vergleich zu anderen Lösungsmöglichkeiten die größte Rolle. Den meisten Benutzern ist außerdem die Funktionsweise von Vermittlungsplattformen bekannt, sodass die Funktionsweise von DTSharing dem Benutzer nicht erläutert werden muss. Dadurch, dass allen Benutzern ein Smartphone zur Verfügung steht, wird die Anfangs getroffene Wahl der Entwicklung einer Applikation für Android bestätigt. Viele Benutzer sind zudem mit der Applikation DB-Navigator vertraut, eine Anlehnung an das Design dieser Applikation wäre somit vorteilhaft um die Zugehörigkeit zur Domäne Bahn zu verdeutlichen und dem Benutzer eine intuitive Bedienung zu erleichtern.

3.2. Deskriptive Aufgabenmodellierung

Die deskriptive Aufgabenmodellierung beschreibt die aktuelle Vorgehensweise der Benutzergruppen Dauerticketbesitzer und Mitfahrer. Dazu wird jede Benutzergruppe in einer separaten HTA analysiert. Da eine HTA beliebig detailliert durchgeführt werden kann, muss darauf geachtet werden, dass der Aufwand, welcher zum erreichen einer höheren Detailliertheit betrieben wird, in einer angemessenen Relation zu dem Nutzen, der aus dem Erreichen dieser Detailliertheit gezogen werden kann, steht. Die deskriptive Aufgabenmodellierung ist im Anhang unter Punkt 7.1. aufzufinden.

3.2.1. Dauerticketbesitzer

Für die Benutzergruppe Dauerticketbesitzer wurde das Goal "Einen Mitfahrer für eine bestimmte Strecke finden, der auf dem eigenen Dauerticket mitgenommen werden kann" ermittelt. Zum erreichen dieses Goals müssen zwei Subgoals, einmal die Suche nach dem passenden Mitfahrer und die Absprache der Fahrtdetails mit dem Mitfahrer, erfüllt sein. Bei der Suche nach einem Mitfahrer werden drei Plattformen, Regionale Foren, Regionale Facebookgruppen und die Applikation DB Mitfahrer genauer analysiert. Weitere Plattformen die zur Suche geeignet wären wurden bewusst nicht in der HTA aufgeführt, da die Suche nach Mitfahrern auf diesen Plattformen nach einem ähnlichen Schema wie auf den bereits analysierten abläuft. Nachdem das erste Subgoal erfolgreich abgeschlossen wurde, also ein Mitfahrer gefunden wurde, müssen weitere Fahrtdetails mit dem Mitfahrer abgesprochen werden. Dazu wird der Mitfahrer entweder über die Vermittlungsplattform oder über eine externe Möglichkeit kontaktiert. Nachdem alle Details erfolgreich mit dem Mitfahrer abgesprochen wurden steht der gemeinsamen Fahrt nichts mehr im Wege.

3.2.2. Mitfahrer

Für die Benutzergruppe Mitfahrer wurde das Goal "Einen Dauerticketbestitzer finden, auf dessen Dauerticket man eine bestimmte Strecke mitgenommen werden kann" ermittelt. Zum erreichen dieses Goals müssen zwei ähnliche Subgoals wie bei der Benutzergruppe Dauerticketbesitzer erfüllt sein, einmal die Suche nach dem passenden Dauerticketbesitzer und die Absprache der Fahrtdetails mit dem Dauerticketbesitzer. Bei der Suche nach einem Dauerticketbesitzer werden die selben Plattformen wie bei der Suche nach einem Mitfahrer benutzt. Nachdem das erste Subgoal erfolgreich abgeschlossen wurde, also ein passender Dauerticketbestitzer gefunden wurde, müssen weitere Fahrtdetails mit dem Dauerticketbestitzer abgesprochen werden. Dazu wird der Dauerticketbestitzer entweder über die Vermittlungsplattform oder eine externe Möglichkeit kontaktiert. Nachdem alle Details

erfolgreich mit dem Dauerticketbestitzer abgesprochen wurden steht der gemeinsamen Fahrt nichts mehr im Wege.

3.3. Anforderungen an das System

Im folgenden wurden Anforderungen spezifiziert, welche bei der Entwicklung von DTSharing beachtet werden müssen. Die Anforderungen wurden in Funktionale Anforderungen, Organisationale Anforderungen, Qualitative Anforderungen, Technische Anforderungen und Usability Anforderungen unterteilt. Sie wurden durch eine Analyse der User-Profiles, der Stakeholderanalyse und der deskriptiven Aufgabenmodellierung ermittelt.

3.3.1. Funktionale Anforderungen

Der Benutzer muss...

- sich registrieren können
- sich einloggen können
- Informationen zu seinem Profil hinzufügen können
- andere Benutzer bewerten können
- anhand seiner Reisedaten Matches erhalten
- Reisedaten bequem per Umkreissuche wählen können
- eine Announce schalten können
- mit seinem Match kommunizieren können

Das System muss...

- eine Loginmöglichkeit bieten
- Benutzern die Möglichkeit geben Profile anzulegen
- eine Bewertung von Benutzern ermöglichen
- Dauerticketanbieter und -suchenden korrekt Matchen
- beim Matching die Möglichkeit des früheren Aussteigens oder späteren Zusteigens berücksichtigen
- Angebot und Nachfrage korrekt in der Datenbank hinterlegen
- den Benutzer informieren, wenn ein Match gefunden wurde
- den Standort des Benutzers per GPS ermitteln können
- alle Haltestellen im Umkreis um eine Koordinate ermitteln
- ein Chatsystem zur Kommunikation zwischen Benutzern beinhalten

3.3.2. Organisationale Anforderungen

- Die Durchführung des Projektes muss mit einem Projektplan geplant werden
- Die Entwicklung des Systems muss im Rahmen des "Usability Engineering Lifecycles" von Mayhem erfolgen
- Bei der Durchführung des Projektes soll dieser Projektplan bestmöglich eingehalten werden
- Der Erstellungsprozess des Systems soll in einer Dokumentation festgehalten werden
- Die Realisierbarkeit des Projektes soll mit der Hilfe von PoC's überprüft werden

3.3.3. Qualitative Anforderungen

Das System...

- muss das Eingangs Identifizierte Problem lösen
- soll die Funktionalen Anforderungen erfüllen
- soll möglichst Fehlerfrei und Zuverlässig laufen
- soll die Erfordernisse der Stakeholder erfüllen

3.3.4. Technische Anforderungen

- Das System soll keine privaten | sensiblen | persönlichen Daten über Google Cloud Messaging senden
- Das System soll mit seiner Systemarchitektur der geplanten Architektur entsprechen
- Der Client soll über HTTP-Verben, auf einzigartige URI's zugreifen und so mit dem Server kommunizieren
- Das System soll eine API zwischen Client und Server nach dem REST Architekturstil beinhalten
- Daten die zwischen Client und Server ausgetauscht werden sollen im JSON Format übermittelt werden

3.3.5. Usability Anforderungen

Die Benutzung des Systems soll...

- intuitiv sein
- einfach zu erlernen sein
- belohnend sein
- zufriedenstellend sein

Das System soll...

- effektiv sein
- effizient sein
- unterstützend sein
- motivierend sein

3.4. Style Guide V1.0

Anhand der Ergebnisse der Anforderungsanalyse wurde ein erster Style Guide erstellt. Dieser beinhaltet Richtlinien, welche als Grundlage für die Gestaltung der Anwendung dienen sollen. Bei der Erstellung des Style Guides wurde sich an der Anwendung "DB-Navigator" Orientiert, da diese bereits viele Benutzern bekannt ist. Außerdem wird dadurch die Zugehörig zu der Domäne Bahn verdeutlicht. Der Style Guide wurde aufgrund neuer Erkenntnisse im Verlauf der gesamten UI-Entwicklung iterativ überarbeitet. Die aktuelle Version ist unter Punkt 4.2.2. zu finden.

Style Guides V1.0				
Schrift				
Schriftfamilie	Myriad Pro			
Schriftschnitt	Mengentext Regular; Headline Bold; Italic besonders			
Satzarten	Linksbündig Rauhsatz			
Farben				
Primary Dark	Primary	Grey Dark	Grey	Grey Light
#c45858	#e06666	#a4a4a4	#cacaca	#e1e1e1
Elemente				
Space between Elements				
Height Width	10dp			
Button Primary				
Fill	default: Primary; active: Primary Dark			
Border	none			
Radius	none			
Font	Regular; Center; White; Uppercase			
Button Default				
Fill	default: Grey; active: Grey Dark			
Border	none			
Radius	none			
Font	Regular; Center; Black; Uppercase			
Input				
Fill	Gradient: {Grey Light -> White}			
Border	solid 1dp; Grey Light			
Radius	none			
Font (Hint)	Regular; Left; Grey Dark			
Font	Regular; Left; Black			
Icons				
Type	Monocolor Vector			
Fill	Grey Dark			

Tabelle 1: Style Guides V1.0

4. Design/Testen/Entwicklung

Die zweite Prozessstufe des Usability Engineering Lifecycles ist in drei Level unterteilt. In dem ersten Level wird die Arbeit neu modelliert, im zweiten Level werden Screen Design Standards ausgearbeitet und im dritten Level wird das User Interface detailliert entwickelt. Während der gesamten Prozessstufe wird ein Prototyp entwickelt, welcher mehrere Evaluations und Redesign-Zyklen durchläuft. Ebenso werden die entworfenen Artefakte aufgrund neuer Erkenntnisse ständig überarbeitet.

4.1. Level 1: Arbeit neu modellieren

Im ersten Level wird die Arbeit neu modelliert. Dazu wurde eine präskriptive Aufgabenmodellierung durchgeführt, welche anhand der in der Anforderungsanalyse gewonnenen Kenntnisse die Arbeit neu modelliert. Daraufhin wurden auf Basis der präskriptiven Aufgabenmodellierung und der während der Anforderungsanalyse entwickelten Style Guides und Anforderungen an das System erste Mockups erstellt, welche nach einer Validierung mit Stakeholdern als Grundlage für einen ersten UI-Prototypen verwendet wurden.

4.1.1. Präskriptive Aufgabenmodellierung

In der präskriptiven Aufgabenmodellierung wird der Ablauf der Aufgabenbewältigung des Benutzers neu modelliert. Als konzeptionelles Modell wurde für die präskriptive Aufgabenmodellierung ebenso wie für die deskriptiven Aufgabenmodellierung eine HTA gewählt. Die Benutzergruppen Dauerticketbesitzer und Mitfahrer wurden wie bereits in der deskriptiven Aufgabenmodellierung in zwei Separaten Diagrammen modelliert. Die Präskriptive Aufgabenmodellierung ist im Anhang unter Punkt 7.2. aufzufinden.

4.1.1.1. Dauerticketbesitzer

Das bereits in der deskriptiven Aufgabenmodellierung ermittelte Goal "Einen Mitfahrer für eine bestimmte Strecke finden, der auf dem eigenen Dauerticket mitgenommen werden kann" bleibt für die Benutzergruppe Dauerticketbesitzer in der präskriptiven Aufgabenmodellierung bestehen. Zum Erreichen dieses Goals müssen die Subgoals "Einen passenden Mitfahrer suchen", "Fahrtdetails mit Mitfahrer absprechen" und wenn gewünscht "Mitfahrer bewerten" erfüllt sein. Die Suche nach einem Mitfahrer und die Absprache mit diesem verlaufen nun über die Plattform DTSharing, die Möglichkeiten über die Plattformen, welche in der deskriptiven Aufgabenmodellierung aufgezeigt wurden, nach einem Mitfahrer zu suchen bzw. mit ihm zu kommunizieren existieren weiterhin, wurden allerdings der besseren Übersichtlichkeit halber nicht erneut modelliert. Nach abgeschlossener Fahrt kann

der Dauerticketbesitzer seinen Mitfahrer bewerten. Da Benutzer mit schlechten Bewertungen weniger Mitfahrer finden erhöht ein Bewertungssystem die Zuverlässigkeit der Benutzer.

4.1.1.2. Mitfahrer

Die präskriptive Aufgabenmodellierung der Benutzergruppe Mitfahrer ist ähnlich wie die präskriptive Aufgabenmodellierung der Benutzergruppe Dauerticketbesitzer aufgebaut. Also Goal wurde wie bereits in der deskriptiven Aufgabenmodellierung "Einen Dauerticketbestizer finden, auf dessen Dauerticket man eine bestimmte Strecke mitgenommen werden kann" ermittelt. Zum erreichen dieses Goals müssen drei Subgoals, einmal die Suche nach dem passenden Dauerticketbestizer, die Absprache der Fahrtdetails mit dem Dauerticketbestizer und wenn gewünscht eine Bewertung des Dauerticketbestizers nach abgeschlossener Fahrt erfüllt sein. Die Suche nach einem Dauerticketbestizer und die Absprache mit diesem verlaufen nun über die Plattform DTSharing, die Möglichkeiten über die Plattformen, welche in der deskriptiven Aufgabenmodellierung aufgezeigt wurde nach einem Dauerticketbestizer zu suchen bzw. mit ihm zu Kommunizieren existieren weiterhin, wurde allerdings der besseren Übersichtlichkeit halber nicht erneut modelliert. Nach abgeschlossener Fahrt kann der Dauerticketbesitzer seinen Mitfahrer bewerten. Da Benutzern mit schlechten Bewertungen weniger Mitfahrer finden erhöht ein Bewertungssystem die Zuverlässigkeit der Benutzer.

4.1.2. Mockup

Die präskriptive Aufgabenmodellierung wurde als Grundlage zur Entwicklung eines Wireframe-artigen Mockups genutzt. Da bei der Analyse der präskriptiven Aufgabenmodellierung nur sehr wenige Unterschiede zwischen den beiden Benutzergruppen Mitfahrer und Dauerticketbesitzer festgestellt wurden, hat man sich dafür entschieden nur eine Anwendung für beide Benutzergruppen zu entwickeln. Das Mockup umfasst 9 Seiten, welche die Hauptfunktionen von DTSharing darstellen. Bei der Entwicklung des Mockups wurde keinen Wert auf das Layout gelegt, da es nur dazu dienen soll die wichtigsten Fenster und deren Inhalt festzulegen.

4.1.2.1. Login

LOGIN	
Anmelden	
E-Mail	
Kennwort	
Abschicken	
Kennwort vergessen	Registrieren

KENNWORT VERGESSEN	
Zurück	Kennwort vergessen
E-Mail	
Abschicken	

REGISTRIEREN	
Zurück	Registrieren
E-Mail	
Kennwort	
Kennwort wiederholen	
Vorname	
Nachname	
Geburtstag	
Geschlecht	
Abschicken	

Abbildung 9: Login

Abbildung 10: Kennwort vergessen

Abbildung 11: Registrieren

Beim Mock-up der Kategorie Login wurde darauf geachtet dem Benutzer durch einer ihm vertrauten Anmeldung, Kennwort vergessen und Registrieren Screens das Einloggen, Registrieren und zurücksetzen seines Kennwertes zu ermöglichen. Da der erste Schritt zur Nutzung von DTSharing die Anmeldung bzw die Registration ist wurde ebenso darauf geachtet nur die nötigsten Informationen zu verlangen um die Absprungschwelle gering zu halten. Wichtig ist hierbei jedoch, dass der Benutzer nicht zu Anonym sein darf, da dieser sich sonst zu sicher fühlt und somit die Seriösität von Angebot und Nachfrage nicht gewährleistet werden kann.

4.1.2.2. Suchmaske

SUCHMASKE		
Navigation		
Chats	Suchmaske	Fahrten
Einstellungen		Beenden
Start		
Ziel		
Datum Uhrzeit		
Ticket		
Abschicken		

Nach Erfolgreichem Login gelangt der Benutzer zur Suchmaske und es wird erstmals eine Navigation präsentiert. Die Suchmaske selbst soll simpel und selbsterklärend sein. Zusätzlich wurde sich stark an der Suchmaske des DB Navigators orientiert, da der Benutzer von DTSharing in der Regel mit diesem Vertraut ist und somit einer intuitiven Suchanfrage nichts im Wege steht.

Abbildung 12: Suchmaske

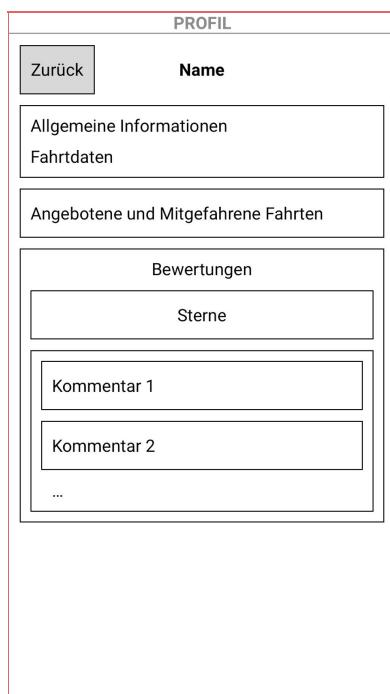
4.1.2.3. Matches

MATCHES	
Zurück	Matches
Match 1	
Match 2	
...	
als Suchend eintragen	

Hat der Benutzer die Suchanfrage abgeschickt erhält er eine Liste mit zutreffenden Matches. Jeder Eintrag Match soll ausschließlich die wichtigsten Informationen darstellen. Benötigt der Benutzer weitere Informationen über den Anbieter/Suchenden kann er durch tippen auf diesen zur Profilseite gelangen. Sollte kein Match gefunden werden oder dem Benutzer kein Match zusagen, besitzt dieser die Möglichkeit sich selbst als Suchend/Anbietend einzutragen.

Abbildung 13: Matches

4.1.2.4. Match Profil



Das Match Profil soll es dem Benutzer ermöglichen sich vorab ein Bild des Anbietenden/Suchenden zu machen um somit zu entscheiden ob er den Match wahrnimmt und diesen kontaktiert. Das Profil führt unter anderem Allgemeine Informationen wie Bild, Name und Alter sowie Fahrtdataen. Um die Seriosität, Aktivität und Zuverlässigkeit des Anbieters/Suchenden darzustellen wird auf eine Statistik über Angebote und Mitgefahrene Fahrten sowie Bewertungen gesetzt.

Abbildung 14: Profil

4.1.2.5. Chat

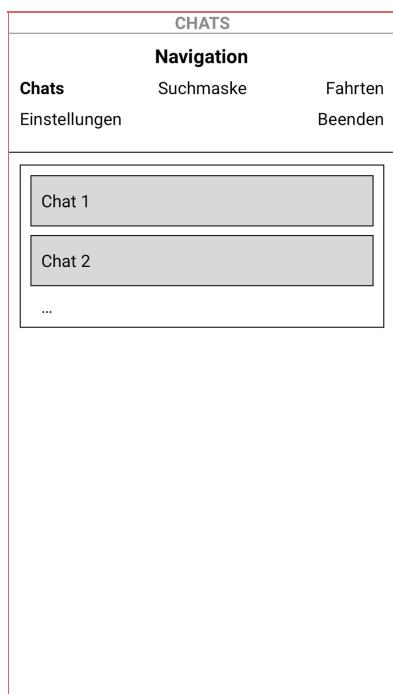


Abbildung 15: Chats

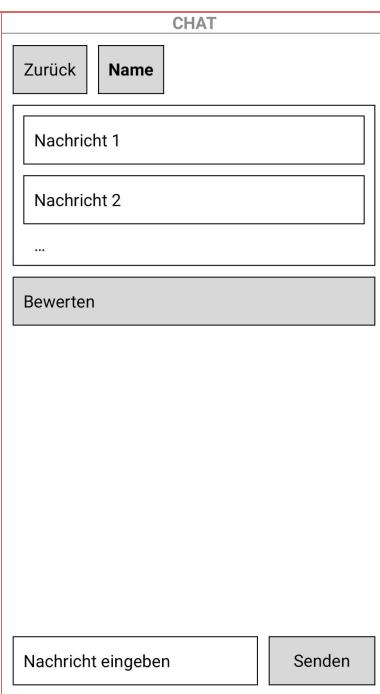
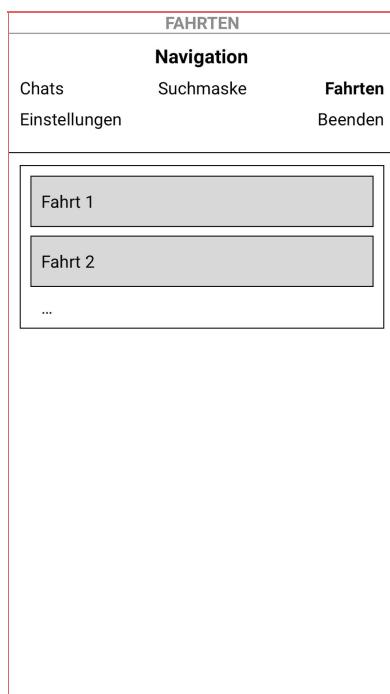


Abbildung 16: Chat

Bei der Kategorie Chat wurde sich an der Chatübersicht und Chatansicht eines typischen Messengers orientiert. Diese sind alle relativ gleich aufgebaut und bieten ähnliche Funktionalitäten. Die Besonderheit des Chats in DTSharing ist, dass dieser keinen Messenger ersetzen oder ihm Konkurrenz machen, sondern lediglich dem Informationsaustausch zwischen zwei Matches dienen soll. Die Chatansicht in DTSharing soll zusätzlich zum Austauschen von Nachrichten noch andere Funktionalitäten bekommen, wie die Möglichkeit den Anbietenden/Suchenden nach Abschluss der Fahrt zu bewerten.

4.1.2.6. Fahrten



Die Kategorie Fahrten stellt eine Übersicht aller vom Benutzer angebotenen Fahrten sowie von Fahrten an denen der Benutzer teilgenommen hat, welche noch nicht Abgeschlossen sind, dar. Somit sieht der Benutzer alle ihn Betreffenden Fahrten auf einen Blick.

Abbildung 17: Fahrten

4.1.3. Evaluation des Mockup

Bevor eine weitere, auf den Mockups aufbauende, detailliertere Erstellung des ersten Prototyps erfolgte wurde eine Evaluation der einzelnen Mockup Screens vorgenommen. Dazu wurden die Screens in einem Interview/Gespräch verschiedenen Stakeholdern gezeigt, welche die Aufgabe hatten, spontan aus dem Bauch heraus, ihre ersten Eindrücke zu vermitteln. Dabei hat sich zusammenfassend ergeben, dass das Mockup ein solides Grundgerüst für die Applikation darstellt es jedoch an Komfortfunktionen mangelt.

"Was ist wenn ich immer nur von Gummersbach nach Köln pendle. Dann will ich diese Route doch nicht immer wieder per Hand eintippen. In der Suchmaske ist doch unten noch ganz viel Luft. Da kann man ja sowas wie einen Verlauf einfügen"

Da das Mockup nur dazu dienen sollte die wichtigsten Fenster und deren Inhalt festzulegen kann hinsichtlich der Bedienelemente und des Layouts keine Evaluation eingeholt werden.

4.1.4. Prototyp V1.0

Es wird ein erster Prototyp nach den Styleguides Version 1, unter Beachtung des Mockups und der Erkenntnisse die sich aus der Evaluation dieses ergeben haben, erstellt. Der Prototyp umfasst 9 Seiten und beschäftigt sich erstmals mit dem Layout. Es wird auf Designentscheidungen eingegangen und am Ende eine Evaluation des ersten Prototypen vorgenommen.

4.1.4.1. Login

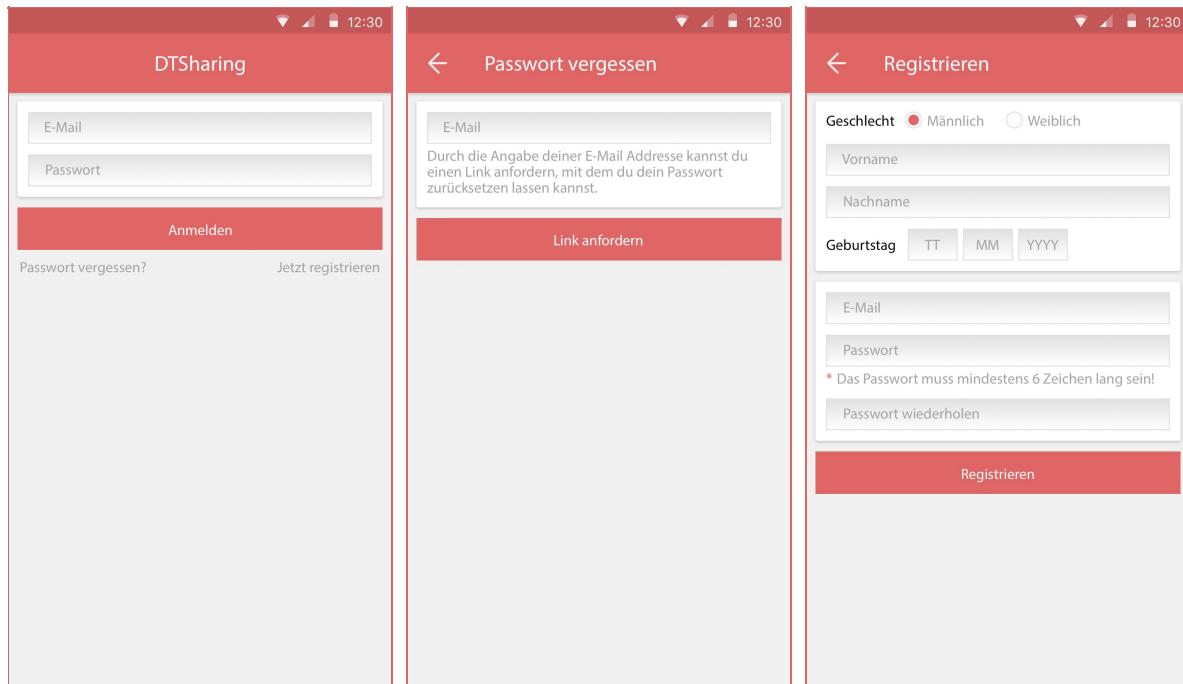


Abbildung 18: Login

Abbildung 19: Passwort vergessen

Abbildung 20: Registrieren

#Login Prototyp V2.0

Im großen und ganzen ähnelt der hier vorliegende Prototyp der Kategorie Login dem aus dem Mockup. Eine Besonderheit, welche den Prototypen V1.0 auszeichnet, ist die Verwendung von Cards welche die Zugehörigkeit von Informationen unterstützen. In der Suchmaske wird dies deutlicher dargestellt.

4.1.4.2. Suchmaske

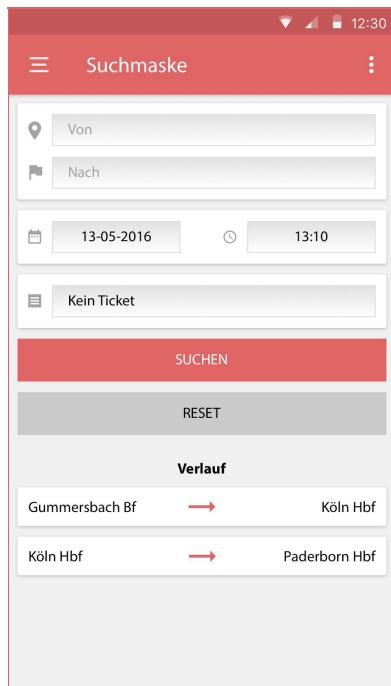


Abbildung 21: Suchmaske

Die Elemente der Suchmaske werden über Cards nach Art gruppiert. Somit ergibt sich aus Start- und Zielort die Gruppe "Wo?", aus Datum und Uhrzeit die Gruppe "Wann?" und aus der Wahl des Tickets die Gruppe "Wie?". Es wurde sich entschieden ohne Labels zu arbeiten und über Icons und Placeholder/Hints die Art eines Eingabefeldes darzustellen. Zusätzlich hat sich aus der Evaluation des Mockups ein Verlauf ergeben, welcher ebenfalls in den Prototypen eingeflossen ist und die letzten Suchanfragen als Shortcut auflistet.

#Suchmaske Prototyp 2.0

4.1.4.3. Navigation

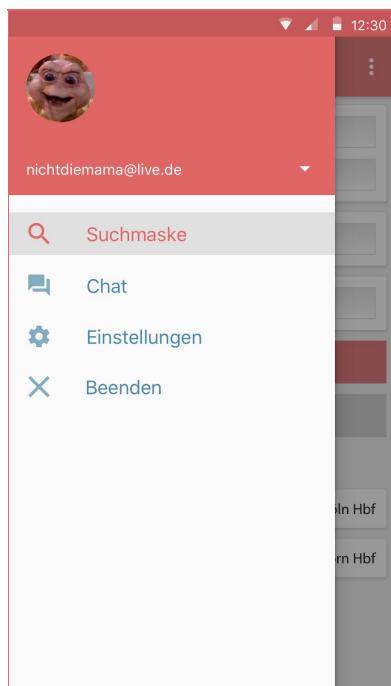


Abbildung 22: Navigation

Die Navigation wird über ein so genanntes Hamburger Menü realisiert und ist somit platzsparend und unauffällig. Es hat sich etabliert und stellt somit keine Hürde dar.

4.1.4.4. Matches

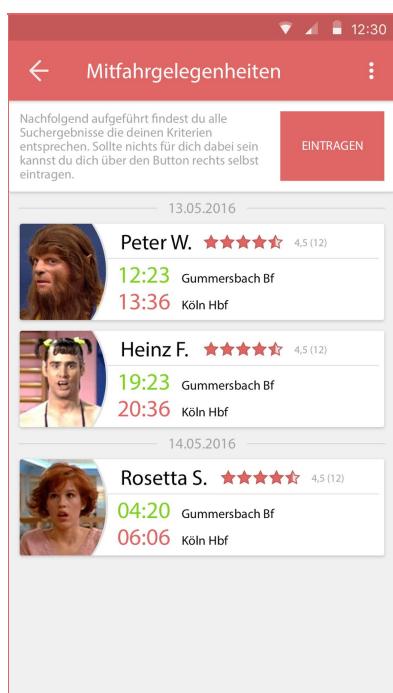
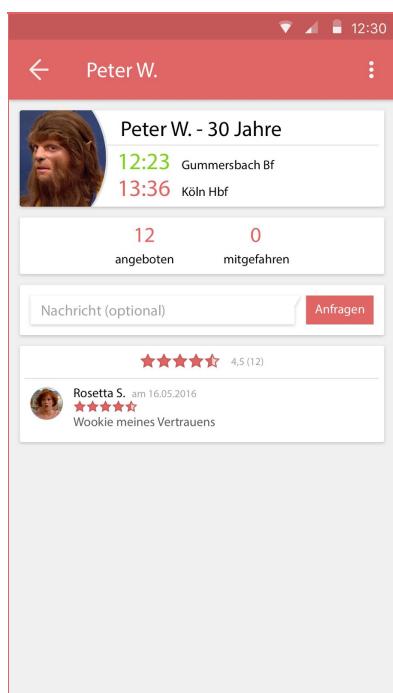


Abbildung 23: Matches

Das Hamburger Menü wird durch einen Zurück Button ersetzt, da die Ansicht der Matches hierarchisch der Suchmaske untergeordnet ist. Oberhalb der Liste mit den Matches wird eine kurze Beschreibung samt Button zum Eintragen der eigenen Fahrt angeboten. Dieser hat somit eine fixe Position und muss nicht gesucht werden. Anschließend folgt eine Liste mit allen gefundenen Matches. Die Einträge sollen nur die nötigsten Informationen aufweisen und somit wurde durch eine Gruppierung nach Datum vermieden, dass dieses unnötig oft wiederholt wird. Zusätzlich wird die Durchschnittliche Bewertung angezeigt und somit kann der Benutzer bereits hier anhand Bild, Bewertung und Reisedaten eine Entscheidung bezüglich der Auswahl des Matches treffen.

4.1.4.5. Match Profil



Das Match Profil ist hierarchisch den Matches untergeordnet. In diesem werden die bereits in der Übersicht ersichtlichen Informationen dargestellt jedoch noch erweitert. Somit erhält man eine Statistik über angebotene und mitgefahrenen Fahrten sowie die Zusammensetzung der Durchschnittlichen Bewertung. Um den Match zu kontaktieren wird dem Benutzer ein Nachrichtenfeld geboten, welches jedoch optional ist und ein Button.

#User Profil Prototyp V2.0

Abbildung 23: Match Profil

4.1.4.6. Chatübersicht

Nachdem der Benutzer seinen Match kontaktiert hat gelangt er zum Punkt 4.1.4.7, da dieser jedoch der Chatübersicht untergeordnet ist wird diese zunächst beschrieben.

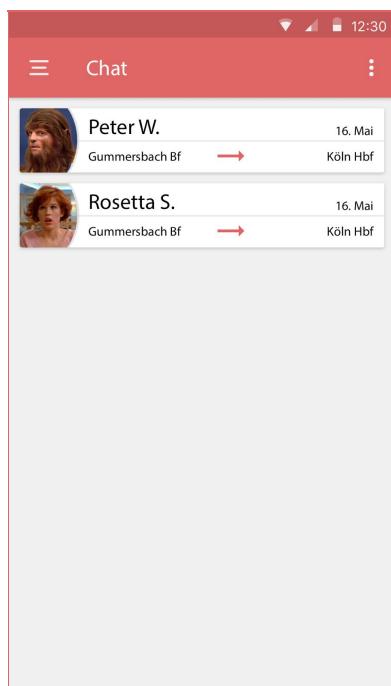


Abbildung 24: Chatübersicht

Die Chatübersicht stellt einen eigenen Menüpunkt in der Navigation dar und enthält eine Auflistung aller aktiven Chats, welche zusätzlich noch die aktiven Fahrten repräsentieren sollen. Diese ist stark an vorhandene Messenger angelehnt und schickt den Benutzer erneut in eine ihm vertraute Umgebung.

#Chatübersicht Prototyp V2.0

4.1.4.7. Chat



Abbildung 25: Chat

Der Chat wurde stark an WhatsApp angelehnt, damit auch dieser sich für den Benutzer vertraut anfühlt, erhält jedoch auch einige Elemente die diesen erweitern. Die Bewertung einer abgeschlossenen Fahrt findet im Chat statt.

4.1.5. Evaluation des Prototyp V1.0

Um die Usability des ersten Prototypen zu testen, wird dieser einer Evaluation unterzogen. Die einzelnen Folien des Prototypen wurden dazu mit einer Applikation namens *Flinto*⁶ miteinander verbunden, sodass ein erster Interaktiver UI-Prototyp entstanden ist. Die Tester hatten nun die Aufgabe, diesen in einem Gespräch durchzugehen und sowohl positives als auch negatives Feedback zu einzelnen Designentscheidungen und der daraus resultierenden Usability kundzutun. Dabei hat sich ergeben, dass es bei der Suchmaske, der Matchübersicht und der Chatübersicht samt Chat im Sinne eines Messengers keine großen Probleme, oder Verständnisschwierigkeiten gibt.

"Ah ok hier unten habe ich einen Verlauf. Wenn ich hier draufdrücke wird die Suchmaske ausgefüllt? Ah ja das ist praktisch!"

Die in der Evaluation identifizierten Probleme betreffen zum einem das Match Profil, da bei diesem nicht sofort ersichtlich ist, wie man diese Fahrt nun reservieren bzw den Anbietenden/Suchenden kontaktieren kann, da die Karte mit Formular und Button untergeht.

"Okay bei dem will ich mitfahren.. Aha Peter W. 30 Jahre.. 12 angeboten 0 mitgefahren.. ja.. Nachricht ah ok die ist Optional.. Anfragen. Ah damit kann ich die Fahrt jetzt reservieren?"

Zum anderen wurde nicht erkannt, dass die Chatübersicht auch noch die Funktion der Fahrtenübersicht darstellte.

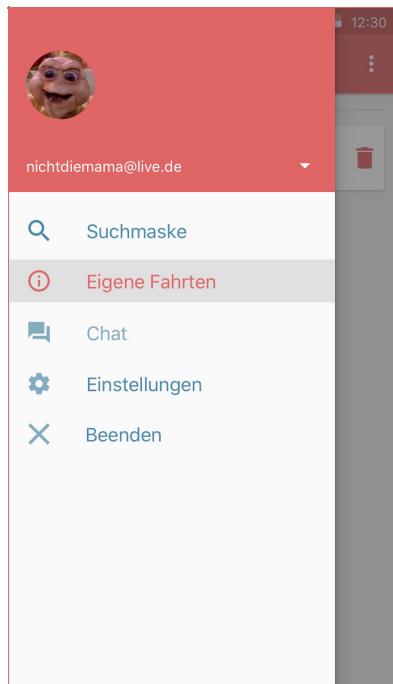
"Wie kann ich denn jetzt sehen, welche Fahrten ich selbst eingetragen habe bzw für welche Fahrten ich mich eingetragen habe?"

⁶ <https://www.flinto.com/mac> (27.05.2016)

4.1.6. Prototyp V1.1

Unter Berücksichtigung der Erkenntnisse die aus der Evaluation des Prototypen V1.0 gewonnen wurden wurde eine iterative Überarbeitung des Prototypen vorgenommen. Nachfolgend werden nur die Änderungen und Neuerungen dokumentiert. Screens welche nicht aufgeführt werden können somit als in diesem Iterationsschritt unverändert betrachtet werden.

4.1.6.1. Navigation



Aufgrund dessen, dass die Multifunktion des Chats nicht erkannt wurde wurde ein neuer Punkt in der Navigation und somit ein neuer Hauptpunkt "Eigene Fahrten" hinzugefügt.

Abbildung 26: Navigation

4.1.6.2. Eigene Fahrten

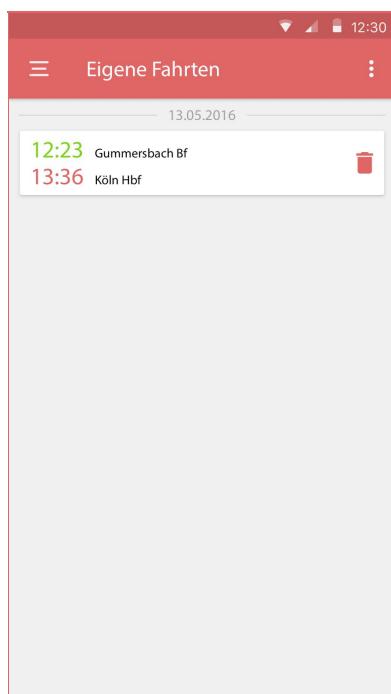


Abbildung 27: Eigene Fahrten

Es wurde ein neuer Hauptpunkt "Eigene Fahrten" hinzugefügt. Dieser bietet die Möglichkeit eine Übersicht über die eigenen eingetragenen Fahrten und Fahrten, bei denen der Benutzer die Rolle des Partners einnimmt, zu erhalten und diese zusätzlich zu Verwalten.

4.1.6.3. Matches

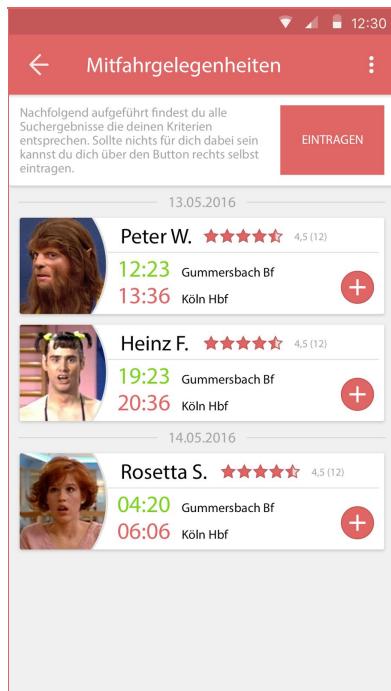


Abbildung 28: Matches

Aufgrund des in der Evaluation identifizierten Problems, der nicht sofort ersichtlichen Möglichkeit den Anbietenden/Suchenden zu kontaktieren, wird bereits in der Matchübersicht ein Button zum kontaktieren angeboten. Die Möglichkeit diesen im Match Profil zu kontaktieren entfällt. Somit hat der Benutzer die Möglichkeit, sofern ihm die nötigsten Informationen bereits ausreichen, eine Fahrt zu reservieren.

#Matches Prototyp V2.0

4.1.7. Evaluation des Prototyp V1.1

Der Vorgang der Evaluation des Prototypen V1.1 unterscheidet sich nicht von dem aus Punkt 4.1.5. Die Änderungen, welche sich aus der Evaluation des Prototypen V1.0 ergeben haben, haben dafür gesorgt, dass die dort identifizierten Probleme behoben wurden. Aufgrund eines neuen Testers konnte jedoch ein weiteres Problem identifiziert werden. Es ist aufgefallen, dass dieser beim durchgehen des Interaktiven UI-Prototypen das Hamburger Menü bzw die Navigation nicht geöffnet hat und somit die Hauptpunkte Chat und Eigene Fahrten nicht wahrgenommen wurden.

"Achso ihr habt ein Hamburger Menü. Meistens sind ja nur kleinere, unwichtige Punkte im Hamburger Menü deswegen war das jetzt für mich uninteressant"

Des Weiteren hat ein Tester geäußert, dass ihm die Möglichkeit fehlt zu sehen ob schon ein Mitfahrer zu seiner Fahrt gefunden wurde.

"Wie sehe ich denn jetzt ob sich schon einer für meine Fahrt eingetragen hat?"

Da diese Evaluation im großen und ganzen sehr zufriedenstellen verlaufen ist und eine erneute Iteration somit keinerlei weitere Erkenntnisse liefern würde wird das Level 1 des Life Cycle mit einem Ausblick auf Level 2 abgeschlossen.

Zur Behebung des hier identifizierten Problems und da die wenigen Hauptpunkte in dem Hamburger Menü sehr verloren wirken wird ein Wechsel auf sogenannte Tabs vorgenommen. Diese bieten verschiedene Vorteile und haben sich ebenso wie das Hamburger Menü etabliert. Zum einen bieten sie den Vorteil alle Hauptpunkte auf einen Blick zu erfassen und es ist somit leicht feststellbar wo man sich derzeit befindet. Des Weiteren ist es möglich bei ungelesenen Chatnachrichten über eine sogenannte Badge im Reiter Chats auf die Anzahl an ungelesenen Nachrichten aufmerksam zu machen.

4.2. Level 2: Screen Design Standards

Im Level 2 - den Screen Design Standards - wurden als erstes Screen Design Standards erstellt. Aufgrund des Screen Design Standards wurde dann der Styleguide überarbeitet. Diese wurden dann in den Prototypen übertragen, welcher in einem iterativen Evaluations und Redesign Zyklus überarbeitet wurde.

4.2.1. Screen Design Standards

Bei der Entwicklung der Screen Design Standards wurden die Guidelines des *Google Material Design*⁷ als Grundlage genommen. Die Material Design Richtlinien wurden von Google entwickelt um das Layout aller Androidanwendungen einheitlich zu Gestalten. Durch die Benutzung des Material Designs wird eine bekannte Umgebung geschaffen, welche eine intuitive Benutzung der Anwendung erleichtert. Da die Dokumentation des Material Designs sehr ausführlich ist wurden nur abweichende Design Standards ausgearbeitet, alle hier nicht aufgeführten Punkte wurden so aus den Material Design Richtlinien übernommen und in den überarbeiteten Styleguides verlinkt und können in der Dokumentation des Material Designs eingesehen werden.

4.2.1.1. Cards

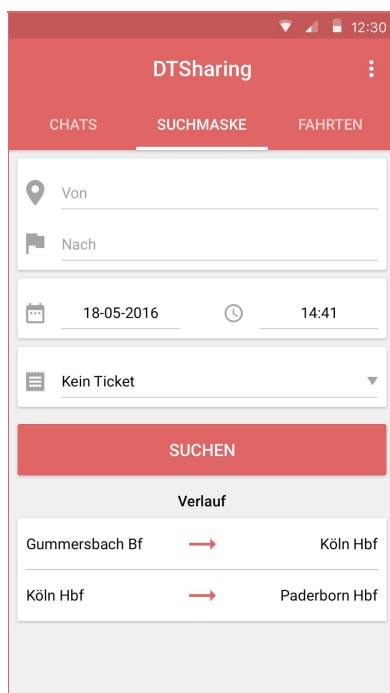
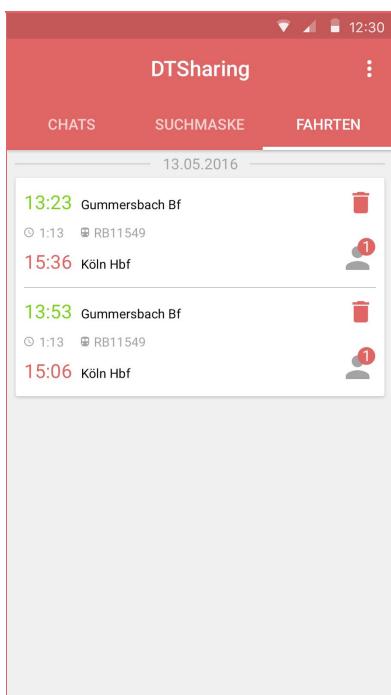


Abbildung 29: Cards

Cards werden nicht nur zum anzeigen von "wegwischbaren" Content benutzt, sondern auch um die Zusammengehörigkeit von einzelnen Elementen zu betonen. Eine Benutzung von Cards in diesem Sinne ist nicht genau so im Material Design vorgesehen, wird allerdings als Notwendig empfunden um eine hohe Gebrauchstauglichkeit zu gewährleisten.

⁷ <https://www.google.com/design/spec/material-design/introduction.html> (29.05.2016)

4.2.1.2. Date-Separator



Der Date-Separator ist ein weiteres Element dass nicht im Material Design vorkommt. Ein Date-Separator wird benötigt damit das Datum nicht mehrmals aufgeführt werden muss sondern nur einmal im Separator. Somit kann der Vorhandene Platz effektiver zum Anzeigen anderer Informationen genutzt werden. Durch die Kombination des Separators mit den Cards ist auf einen Blick ersichtlich welche Fahrten an welchem Tag stattfinden.

Abbildung 30: Date-Separator

4.2.2. Style Guide V2.3

Anhand des Material Designs wurden die Style Guides stark überarbeitet um den neuen Richtlinien zu entsprechen.

Style Guides V2.3						
Schrift						
Schriftfamilie	Roboto					
Schriftschnitt	Mengentext Regular; Headline Medium;					
Satzarten	Linksbündig Rauhsatz					
Farben						
Primary Dark	Primary	Grey Dark	Grey	Grey Light		
#c45858	#e06666	#555555	#a4a4a4	#f0f0f0		
Elemente						
Space between Elements						
Height Width	8dp 16dp					
Button Primary		Material Link: <i>Raised Buttons</i> ⁸				
Fill	default: Primary; active: Primary Dark					
Border	none					
Radius	2dp					
Shadow	Color: Black; Alpha: 20%; X:0 Y:1 Blur: 3					
Font	Weight: Medium; Align: Center; Color: White; Case: Uppercase					
Button Dialog		Material Link: <i>Flat Buttons</i> ⁹				
Fill	none					
Border	none					
Radius	none					
Font	Weight: Bold; Align: Center; Color: Primary; Case: Uppercase					
Button Card List		Material Link: <i>Avatar</i> ¹⁰				
Fill	Primary					
Border	none					
Shape	Circle					
Content	Color: White					
Text Inputs		Material Link: <i>Text Fields</i> ¹¹				
Fill	none					
Border	Thickness: 0.5dp; Color: Grey					
Radius	none					

⁸ <http://www.google.com/design/spec/components/buttons.html#buttons-flat-buttons> (29.05.2016)

⁹ <http://www.google.com/design/spec/components/buttons.html#buttons-style> (29.05.2016)

¹⁰ <https://www.google.com/design/spec/components/lists.html#lists-actions> (29.05.2016)

¹¹ <https://www.google.com/design/spec/components/text-fields.html> (29.05.2016)

Font (Hint)	Weight: Regular; Align: Left; Color: Grey
Font	Weight: Regular; Align: Left; Color: Black
Icons	Material Link: Icons¹²
Type	Monocolor Vector
Fill	Grey
Cards, Dialog	Material Link: Cards¹³
Fill	White
Border	none
Shadow	Color: Black; Alpha: 20%; X:0 Y:1 Blur: 3
Radius	2dp
Avatar	Material Link: Avatar¹⁴
Shape	Circle
Border	none
Shadow	none
Tabs	Material Link: Fixed Tabs¹⁵
Fill	Primary
Border	none
Shadow	none
Radius	none
Font	Weight: Medium; Align: Center; Color: White; Case: Uppercase
Dropdown	Material Link: Dropdown¹⁶
Fill	Primary
Border	none
Shadow	none
Radius	none

Tabelle 2: Style Guides V2.3

¹² <https://design.google.com/icons/> (29.05.2016)¹³ <https://www.google.com/design/spec/components/cards.html> (29.05.2016)¹⁴ <https://www.google.com/design/spec/components/lists.html#lists-behavior> (29.05.2016)¹⁵ <http://www.google.com/design/spec/components/tabs.html#tabs-specs> (29.05.2016)¹⁶ <https://www.google.com/design/spec/components/menus.html#menus-behavior> (29.05.2016)

4.2.3. Prototyp V2.0

4.2.3.1. Login

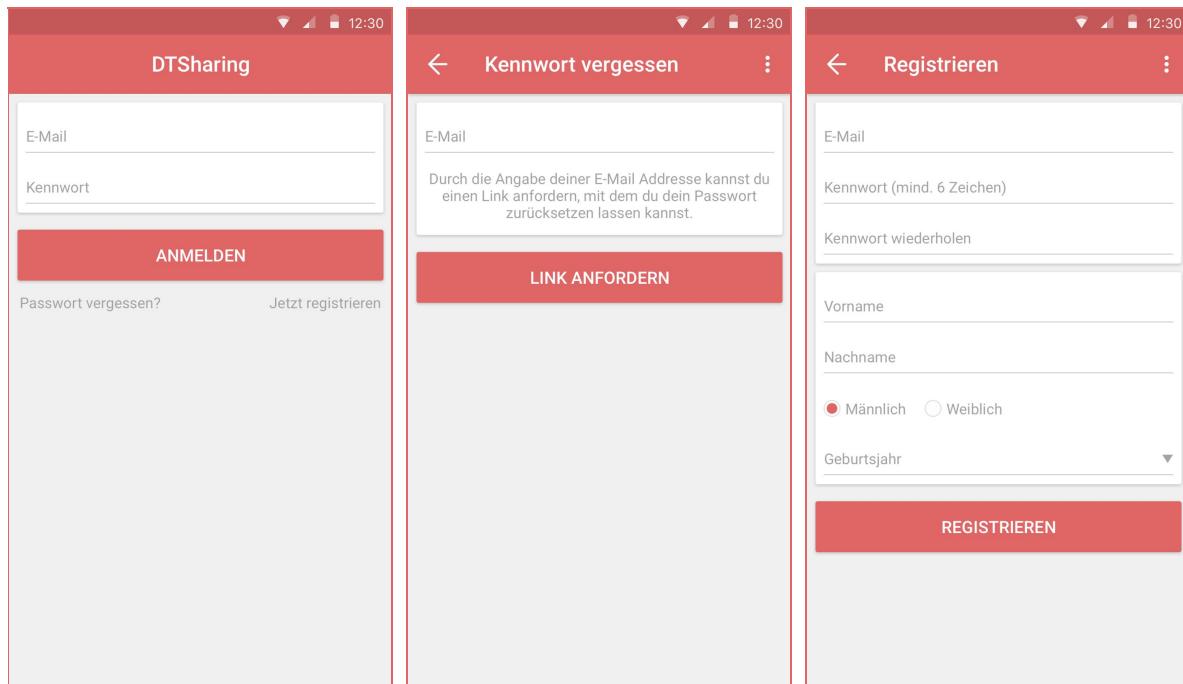


Abbildung 31: Login

Abbildung 32: Passwort vergessen

Abbildung 33: Registrieren

Die Kategorie Login hat Inhaltlich keine allzu starke Überarbeitung erfahren müssen. Wie in der Abbildung 31, dem Registrieren, zu sehen wurde die Ordnung der Felder etwas umgestellt und es wird lediglich das Geburtsjahr statt des genauen Geburtstags benötigt. Ansonsten sind die eigentlichen Änderungen die Anpassung an das Google Material Design.

#Login Prototyp V1.0

4.2.3.2. Suchmaske

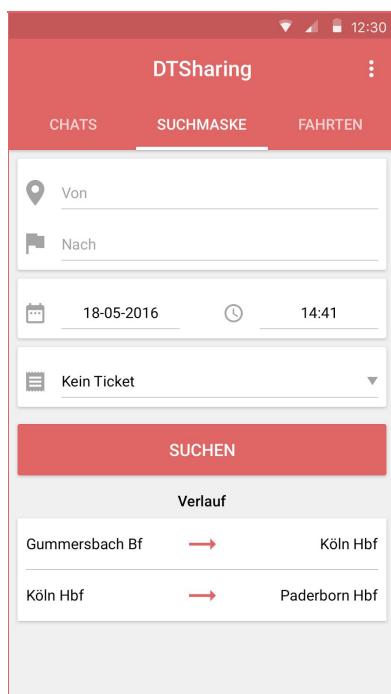


Abbildung 34: Suchmaske

Ebenso die Suchmaske hat Inhaltlich keine große Überarbeitung mehr erfahren müssen. Eine auffällige Neuerung, welche bereits als Ausblick in der Evaluation des Prototypen V1.1 erwähnt wurde, ist der Wechsel vom Hamburger Menü auf Tabs. Obwohl Tabs mehr Platz beanspruchen als ein Hamburger Menü, welches in der bereits vorhandenen Titelleiste untergebracht würde, sind sowohl Suchmaske als auch Verlauf immer noch auf einen Blick zu erfassen.

#Suchmaske Prototyp V1.0

4.2.3.3. Matches

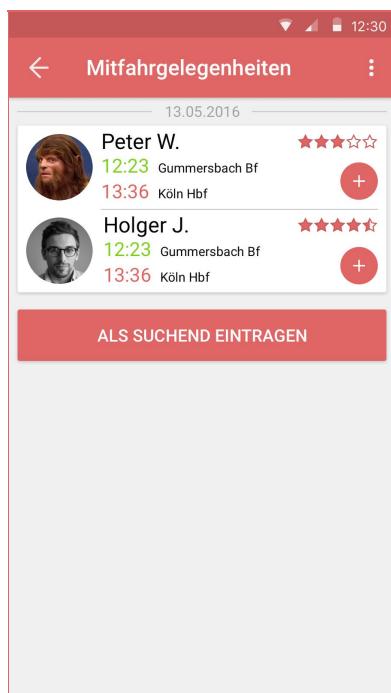
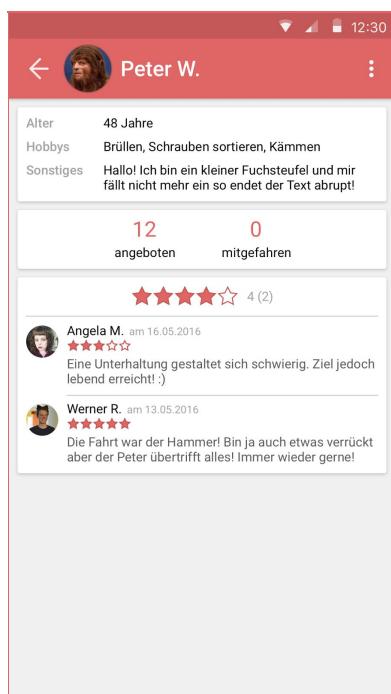


Abbildung 35: Matches

Auch bei Matchübersicht wurde im wesentlichen nur das Google Material Design angewandt und kleinere Anpassungen an der Positionierung von Elementen vorgenommen. Die größte Änderung hier ist die Reduktion des Eintragen Buttons inklusive der Beschreibung auf einen einzigen Button, welcher am Ende der Matches angezeigt wird. Somit soll vermieden werden, dass Benutzer sich sofort als Suchend / Anbietend eintragen statt die Matches durchzugehen, nur weil der Button an oberster Stelle steht.

#Matches Prototyp V1.1

4.2.3.4. User Profil

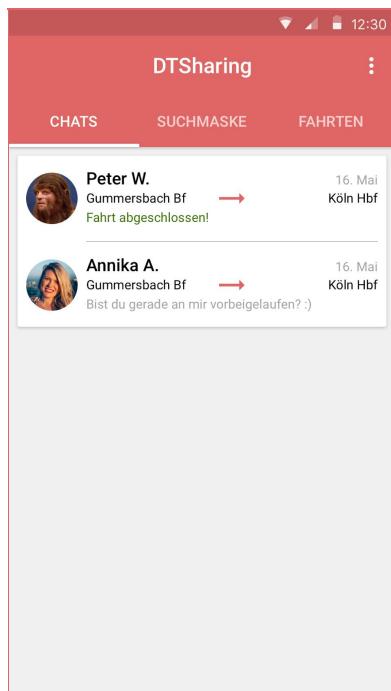


Die wesentliche Änderung im Match Profil ist, dass die Fahrtdata durch allgemeine Informationen ersetzt wurden und dieses somit in ein reines User Profil umfunktioniert wurde. Die Matchübersicht stellt somit die einzige Anlaufstelle zum kontaktieren eines Matches dar. Des Weiteren wurden auch hier das Google Material Design angewandt.

#Match Profil Prototyp V1.0

Abbildung 36: User Profil

4.2.3.5. Chatübersicht



Auch die Chatübersicht hat keine großartigen Veränderungen erfahren. Eine Inhaltliche Änderung ist die Anzeige der letzten Nachricht, um eine Verwechslung mit der Fahrtenübersicht zu vermeiden. Des Weiteren wurde auch hier das Google Material Design eingearbeitet.

#Chatübersicht Prototyp V1.0

Abbildung 37: Chatübersicht

4.2.3.6. Chat



Abbildung 38: Chat

Beim Chat wurde zusätzlich zur Anpassung an das Google Material Design noch das Eingabefeld samt Button zum abschicken der Nachricht vergrößert, da diese Elemente etwas zu kleine geraten waren.

#Chat Prototyp V1.0

4.2.3.7. Fahrten

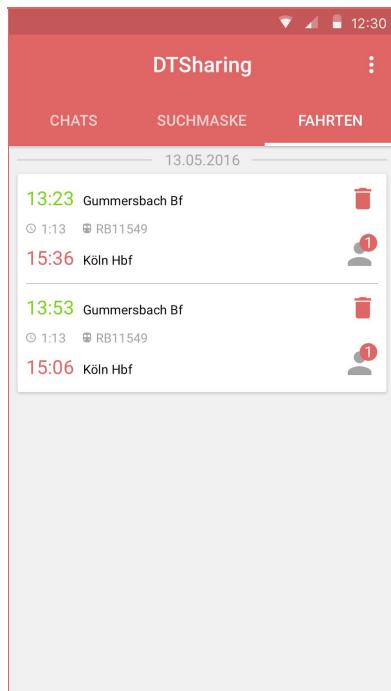


Abbildung 39: Fahrten

Zur Lösung des in der Evaluation des Prototypen V1.1 identifizierten Problems der fehlenden Information ob bereits ein Partner für eine Fahrt gefunden wurde, wurde ein Icon inklusive Badge implementiert welches diese darstellen soll. Zur näheren Anlehnung an den DB Navigator wurde ebenso die Information der Dauer als auch des Liniennamens eingefügt. Des Weiteren wurde auch hier das Google Material Design eingearbeitet.

#Eigene Fahrten Prototyp V1.1

4.2.4. Evaluation des Prototyp V2.0

Der Vorgang der Evaluation des Prototypen V2.0 unterscheidet sich nicht von dem aus Punkt 4.1.5. Somit wurde auch dieser zu einem Interaktiven UI-Prototypen gemacht und mit Testern in einem Gespräch durchgegangen. Dabei hat sich ergeben, dass der Einfluss des Google Material Designs, die Anlehnung an den etablierten DB Navigator sowie der Aufbau des Chats, welcher einem typischen Messenger ähnelt, eine intuitive Benutzung der Applikation gewährleisten.

"Bis jetzt konnte ich nichts auffälliges feststellen. Macht sogar irgendwie einen vertrauten Eindruck auf mich"

Eine Frage, welche aufgekommen und somit ein identifiziertes Problem darstellt, ist, dass von der Suchmaske direkt in die Matches gesprungen wird und die Benutzer somit unsicher sind, ob die eingetragenen Fahrten nun korrekten Fahrplandaten entsprechen.

"Hmm wie ist das denn jetzt? Kann man jede x-beliebige Uhrzeit eintragen oder wie läuft das? Sonst trägt jemand eine falsche Uhrzeit ein und man steht da..."

Daraus ergibt sich, dass noch ein Schritt zwischen Suchmaske und Matches implementiert werden muss, welcher eingetragene Reisedaten mit tatsächlichen abgleicht und dem Benutzer eine Auswahl an Verbindungen darstellt, ähnlich wie die Funktionsweise des DB Navigators.

4.2.5. Prototyp V2.1

Unter Berücksichtigung der Erkenntnisse die aus der Evaluation des Prototypen V2.0 gewonnen wurden wurde eine iterative Überarbeitung des Prototypen vorgenommen. Nachfolgend werden nur die Änderungen und Neuerungen dokumentiert. Screens welche nicht aufgeführt werden können somit als in diesem Iterationsschritt unverändert betrachtet werden.

4.2.5.1. Verbindungen

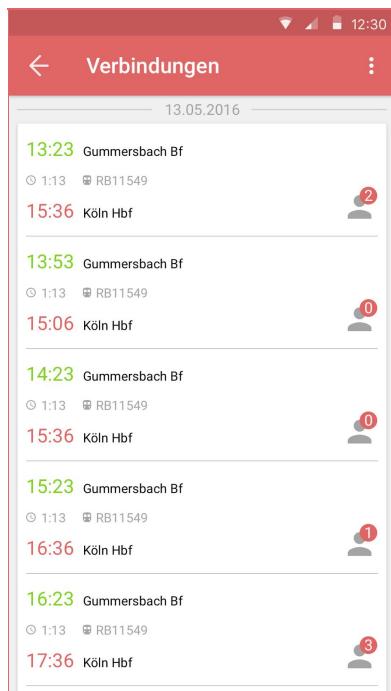


Abbildung 40: Verbindungen

Aufgrund des in der Evaluation des Prototypen 2.0 identifizierten Problems des fehlenden Schrittes zwischen Suchmaske und Matches wurde ein neuer Screen namens Verbindungen erstellt. Dieser wurde stark an den DB Navigator angelehnt und soll für die nötige Gewissheit seitens der Benutzer sowie dem abgleichen der eingetragenen Reisedaten mit tatsächlichen Reisedaten dienen. Zusätzlich wird auch hier ein Icon inklusive Badge dargestellt, welches vorab die Anzahl an Matches für diese Verbindung anzeigen und somit ein gezieltes Wählen von Verbindungen unterstützen soll.

4.2.6. Evaluation des Prototyp V2.1

Nachdem der Screen der Verbindungen implementiert wurde wurde eine erneute Evaluation vorgenommen. Diese ist vom Ablauf gleichzusetzen mit der des Prototypen V2.0. Dabei haben sich durchweg positive Erkenntnisse ergeben. Durch das Einfügen des Zwischenschrittes Verbindungen konnte beobachtet werden, dass die Unsicherheit seitens der Stakeholder genommen wurde, da der nun entstandene Workflow vom Ausfüllen der Suchmaske bis zur Auswahl des Matches nun viel mehr dem bereits bekannten des DB Navigators entspricht.

"Also ich finde das bis jetzt schon ganz gut. Habt ihr euch denn schon Gedanken gemacht, wie die Dialoge aussehen? Also wenn ich jetzt jemanden bewerten will?"

Da in dieser Evaluation keine weiteren Probleme identifiziert wurden und auch die Frage des Stakeholders eher in die Richtung des Detailed User Interface Designs geht kann die Iteration abgeschlossen und somit zum Level 3 übergegangen werden.

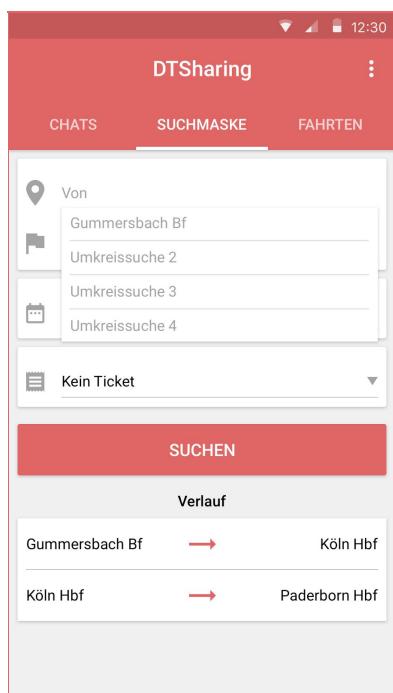
4.3. Level 3: Detailed User Interface Design

Im dritten Level, dem Detailed User Interface Design, werden einzelne Elemente des User Interfaces Designed werden einzelne Aspekte des User Interfaces detailliert ausgearbeitet, sodass ein Prototyp entsteht der das User Interface des fertigen Produktes repräsentiert.

4.3.1. Prototyp V3.0

Da bereits im Level 2 sehr detailliert gearbeitet wurde, können die dort Erarbeiteten Screens übernommen und durch die Ergänzung von Dialogen erweitert werden. Nachfolgend aufgeführt werden nur Implementationen der Dialoge dokumentiert.

4.3.1.1. Suchmaske - Umkreissuche



Das Autocomplete stellt eine Mischung aus Umkreissuche und Autocomplete dar. Die Ergebnisse der Umkreissuche sollen in der Liste oben stehen. Wird nun ein Suchbegriff eingetippt wird sowohl Umkreissuche als auch Autocomplete gefiltert. Dies bietet den Vorteil, dass der Benutzer für die Umkreissuche keine extra Funktion aufrufen muss. Zusätzlich wird dies im Stil des Google Material Designs realisiert und passt somit zum Gesamtbild.

Abbildung 41: Umkreissuche

4.3.1.2. Suchmaske - Picker

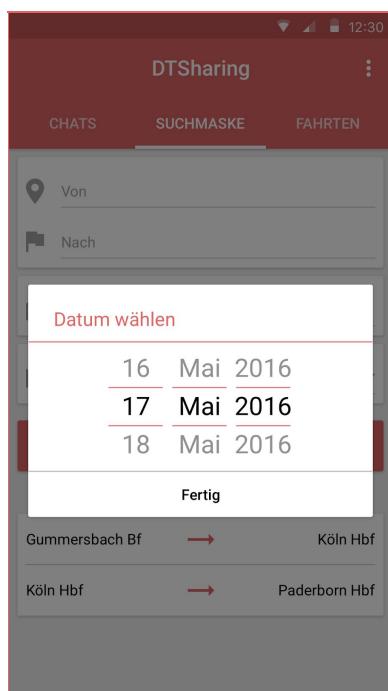


Abbildung 42: Picker - Datum

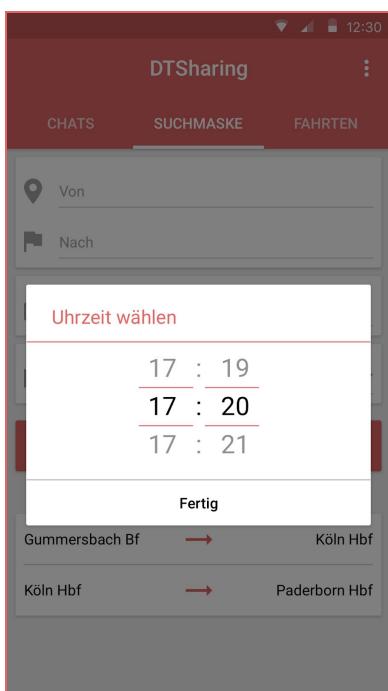


Abbildung 43: Picker - Uhrzeit

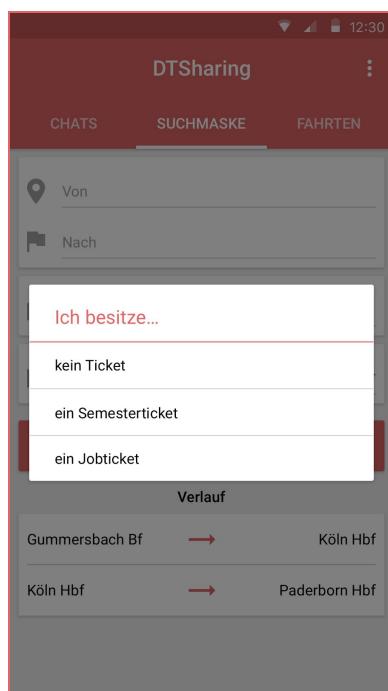


Abbildung 44: Picker - Ticket

Die Picker werden über das Android Framework realisiert. Das bedeutet, dass diese nicht selbst erstellt werden. Dies hat den Vorteil, dass diese dem Benutzer bereits bekannt sind, da sie bereits in anderen Applikationen genutzt werden und somit eine Intuitive Benutzung dieser ermöglicht wird. Erwähnenswert ist, dass die Picker sich je nach Android Version unterscheiden können. Dies hat jedoch keinen Einfluss auf die Intuitive Benutzung, da der Nutzer bereits mit seiner Version vertraut ist.

4.3.1.3. Matches - Dialoge

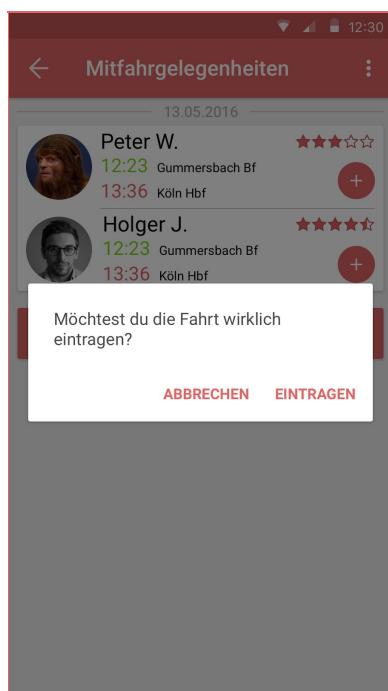


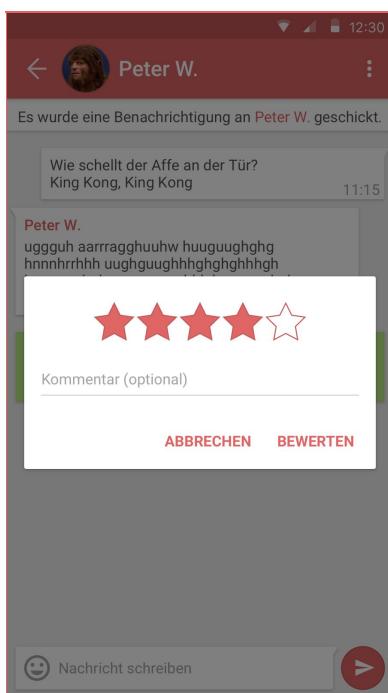
Abbildung 45: Dialog - Eintragen



Abbildung 46: Dialog - Mitfahren

Bei der Gestaltung der Dialoge wird sich stark an der Dialoggestaltung des Google Material Design orientiert. Diese überzeugt durch Schlichtheit bietet durch die explizite Benennung der Flat Buttons jedoch auch Transparenz bezüglich der Aktion die sich hinter diesem verbirgt. Zusätzlich wird auch hier wieder auf ein bereits etabliertes Element zurückgegriffen, was wiederum den Vorteil hat, dass der Benutzer bereits Erfahrung mit diesem hat.

4.3.1.4. Chat - Bewertung



Die Bewertung stellt einen Dialog dar, welcher selbst erstellt werden muss. Auch dieser wurde nach dem Google Material Design erstellt und ist somit sehr schlicht gehalten. Es wurde darauf geachtet, dass der Vorgang einer Bewertung mit so wenig Arbeit wie möglich verbunden wird, da diese Funktion sonst nicht ausreichend benutzt würde, Bewertungen jedoch Ausschlaggebend für die Entscheidungsfindung eines Benutzers sind.

Abbildung 47: Dialog - Bewertungen

4.3.1.5. Menüs

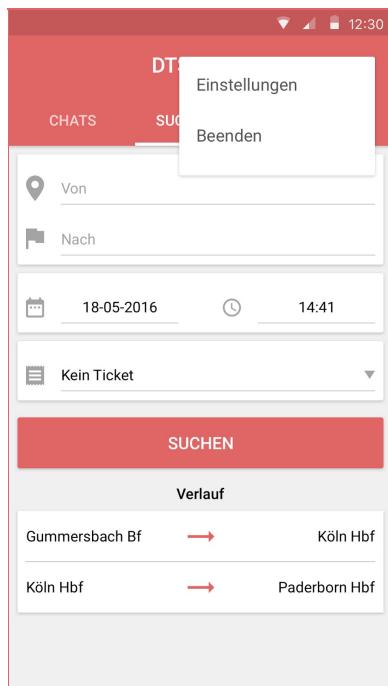


Abbildung 48: Menü - Suchmaske

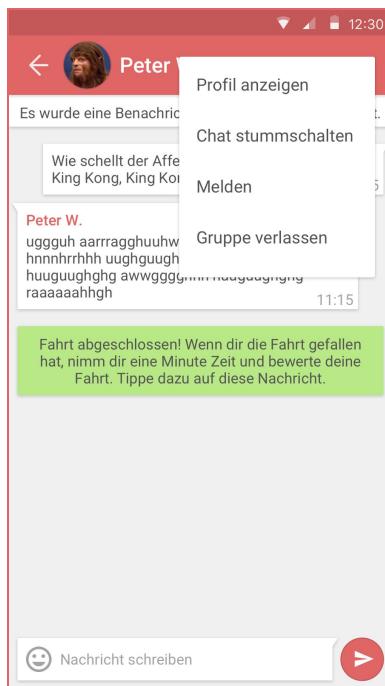


Abbildung 49: Menü - Chat

Es wurden außerdem Dialoge für das Menü entworfen. Bei der Entwicklung dieser Dialoge wurden die Material Design Richtlinien umgesetzt. Abhängig von der Ansicht man sich befindet können unterschiedliche Aktionen innerhalb des Menüs aufgerufen werden. So ist es innerhalb des Chusses möglich sich das Profil seines Mitfahrers anzusehen, den Chat stummzuschalten oder auch den Mitfahrer zu melden falls dieser einen Anlass dazu bietet. Außerdem ist es über diesen Dialog möglich in die Einstellungen zu gelangen oder die Anwendung zu beenden.

4.4. Fazit

Nachdem nun die Dialoge implementiert wurden wurde eine weitere Evaluation, in Form von Gesprächen, mit Stakeholdern durchgeführt. Es konnten allerdings keine weiteren Probleme identifiziert werden, weshalb die zweite Phase des Lifecycles abgeschlossen werden kann. Der aktuelle Stand des interaktiven Prototypes wurde in einer HTML-Datei, welche im Prototyp-Ordner des zweiten Meilensteines zu finden ist, festgehalten. Die verwendeten Anomationen zum wechseln zwischen einzelnen Screens sind dabei nicht die final, da die während der Entwicklung in Flinto genutzten Animationen nicht in die HTML-Datei übernommen werden konnten. Durch die direkte Arbeit mit dem Material Design wurde ein Stimmiger Ergebnis erzielt und das Entwicklerteam hat viel Erfahrung gewonnen. Der Entwickelte Prototyp wird nicht bereits das Perfekte User Interface darstellen und es wird mit dem Auftreten weiterer Probleme gerechnet. In der nun folgenden dritten Phase, der Implementierung, wird

deshalb weiterhin eng mit den Stakeholdern zusammengearbeitet um weitere Probleme möglichst frühzeitig zu identifizieren und effektiv zu lösen.

5. Datenstrukturen

Wie im Architekturdiagramm (7.4.) ersichtlich, findet der Datenaustausch größtenteils zwischen dem Client und dem Server statt. Der Client ist dabei eine Anwendung die auf einem Android-Smartphone läuft. Auf die Nutzung einer Fahrplan-API wurde verzichtet, da keine Echtzeitdaten benötigt werden. Die für die Anwendung benötigten Fahrplandaten werden im *GTFS-Format*¹⁷ in einer Datenbank, auf dem Server, gespeichert. Zusätzlich zu den GTFS Daten werden auf dem Server noch Informationen zu den einzelnen Benutzern und zu Angeboteten Fahrten gespeichert. Auf dem Client werden Daten wie der Suchverlauf oder die Position aller Haltestellen gespeichert. Zusätzlich werden auf dem Client auch Informationen zu einigen Benutzern und Chatnachrichten gespeichert.

5.1. Datenstruktur - Client

Die Clientseitige Datenstruktur ist in dem ER-Diagramm unter Punkt 7.6. ersichtlich. In den Entitäten "chats" und "messages" werden alle Chaträume denen der Benutzer angehört bzw alle Nachrichten dieser Chaträume gespeichert. Beim öffnen eines Chatraumes müssen somit nur neue Nachrichten vom Server abgerufen werden und das Ansehen von Nachrichten funktioniert auch ohne eine aktive Internetverbindung. In der Entität "users" werden alle Benutzerprofile gespeichert die regelmäßig benötigt werden, also z.B. Chatpartner des Benutzers. Die Profilbilder der Benutzer werden in einer separaten Entität gespeichert, sodass bei einem geupdatetem Profil nicht das gesamte Bild übermittelt wird, sondern erst die ID der Bilder miteinander verglichen werden kann und danach, wenn nötig, ein neues Bild heruntergeladen wird. Durch eine Clientseitige Speicherung ist es ebenfalls möglich Benutzerprofile ohne eine aktive Internetverbindung aufzurufen und die Menge an übertragenen Daten wird, da die Profile nicht jedes mal übermittelt werden müssen, minimiert. Zusätzlich sind im Client die Entitäten "stops", eine Auflistung aller Haltestellen inklusive ihrer Position, die für die Umkreissuche bzw. den Autocomplete benötigt werden (siehe 6.2.1.1.), und "history", eine Auflistung aller Suchanfragen die für den Verlauf (siehe 6.2.1.2.) benötigt werden, vorhanden.

¹⁷ <https://developers.google.com/transit/gtfs/reference> (29.05.2016)

5.2. Datenstruktur - Server

Die Serverseitige Datenstruktur wird durch das ER-Diagramm im Anhang (Punkt 7.7.) beschrieben. Dabei kann die Serverseitige Datenstruktur in zwei Bereiche unterteilt werden. Der erste Bereich besteht aus einer Datenbank die mit GTFS Daten gefüllt wurde, der zweite Bereich ist für die Speicherung von Benutzerdaten und eingetragenen Fahrten zuständig.

5.2.1 GTFS-Daten

Der erste Bereich des Serverseitigen Datenstruktur besteht aus den *GTFS Daten*¹⁸. In der Entität "trips" sind Informationen zu allen Fahrten gespeichert. Jeder Trip findet dabei maximal einmal täglich statt. Über die "service_id" können in den Entitäten "calendar_dates" und "calendar" Informationen zu den Tagen, an denen ein Trip stattfindet, abgerufen werden. Über die Zuordnungstabelle "stop_time" kann auf die Haltestellen eines Trips, sowie den Abfahrts und Ankunftszeiten an diesen, zugegriffen werden. Jede Haltestelle wird dabei eine "stop_sequence" zugeordnet, die angibt um wie viele Haltestelle auf dem Trip es sich bei dieser handelt. In der Entität "routes" können Informationen zum Namen eines Trips abgerufen werden, also z.B. die Zugnummer.

5.2.2 Benutzerdaten/eingetragene Fahrten

Der zweite Bereich der Serverseitigen Datenstruktur besteht aus den Benutzerdaten, Chatdaten, den Bewertungsdaten und den Daten über eingetragene Fahrten. In den Entitäten "users" und "pictures" werden alle Benutzerdaten sowie ihre Profilbilder gespeichert. In den Entitäten "chats" und "messages" werden Chatnachrichten und Informationen zu den Chats in denen sie geschrieben wurden gespeichert. Die Entität "ratings" speichert alle Informationen die zu einer Bewertung eines Benutzers gehören. In der Entität "dt_trips" werden alle Informationen zu eingetragenen Fahrten gespeichert. Die "dt_trips_id" ist dabei die Einzigartige ID eines "dt_trips", die "unique_trip_id" ist die ID die fürs Matching benötigt wird (siehe 2.1.1. und 6.2.2.2.) und die trip_id verweist auf einen Eintrag in der "trips" Entität der GTFS-Daten.

¹⁸<https://developers.google.com/transit/gtfs/reference> (29.05.2016)

5.3. Datenübertragung

Die Datenübertragung zwischen Server und Client funktioniert über ein https Protokoll, mit Hilfe dessen JSON Objekte übermittelt werden. Da das versenden von Bildern in JSON Objekten nicht möglich ist wurde sich dafür entschieden diese Bilder mit der *Base64*¹⁹ Encoding Methode in Strings umzuwandeln. Diese können dann mithilfe von JSON Objekten übermittelt werden. Durch die Benutzung dieser Encoding Methode steigt die Datengröße eines Bildes um etwa 33% an. Dies wird nicht als problematisch angesehen, da Bilder von Profilen auf die oft zugegriffen wird Clientseitig gespeichert werden und somit selten übermittelt werden. Nachfolgend sind Beispielhaft zwei JSON-Schemata welche die Struktur von übermittelten Daten aufzeigen.

5.3.1. Fahrten zwischen zwei Bahnhöfen ermitteln

Dieses Schema beschreibt ein JSON-Array das der Client als Antwort auf eine get Anfrage an die URI /trips erhält. In diesem JSON-Array sind alle Daten enthalten die benötigt werden um eine Liste der gesuchten Fahrten anzuzeigen.

```
{
  "type": "array",
  "title": "Trips",
  "description": "Eine Sammlung von Trips die von Bahnhof a nach Bahnhof b fährt",
  "items": {
    "type": "object",
    "title": "Trip",
    "description": "Ein einzelner Trip",
    "properties": {
      "trip_id": {
        "type": "integer",
        "description": "Die ID des Trips"
      },
      "departure_time": {
        "type": "integer",
        "description": "Die Abfahrtszeit des Trips"
      },
      "arrival_time": {
        "type": "integer",
        "description": "Die Ankunftszeit des Trips"
      },
      "sequence_id_departure_station": {
        "type": "integer",
        "description": "Die Sequence ID des Abfahrtsbahnhofs des Trips"
      }
    }
  }
}
```

¹⁹ <https://tools.ietf.org/html/rfc3548> (29.05.2016)

```
"sequence_id_target_station": {  
    "type": "integer",  
    "description": "Die Sequence ID des Zielbahnhofs des Trips"  
},  
"departure_date": {  
    "type": "integer",  
    "description": "Das Datum des Abfahrttages des Trips"  
},  
"number_matches": {  
    "type": "integer",  
    "description": "Die Anzahl an Matches des Trips"  
}  
},  
"required": [  
    "trip_id",  
    "departure_time",  
    "arrival_time",  
    "sequence_id_departure_station",  
    "sequence_id_target_station",  
    "departure_date",  
    "number_matches"  
]  
}  
}
```

5.3.2. Eine Fahrt eintragen

Dieses JSON-Schemata beschreibt ein JSON-Object das der Client als Payload bei einer post Anfrage auf die URI `/users/:user_id/dt_trips` versendet. In ihm sind alle Informationen enthalten die zum Anlegen eines neuen Datenbankeintrages in der Tabelle `dt_trips` benötigt werden.

```
{
  "type": "object",
  "title": "Submit Trip",
  "description": "Beinhaltet alle Informationen die zum anlegen einer Fahrt notwendig sind",
  "properties": {
    "user": {
      "type": "object",
      "description": "Die Benutzerspezifischen Daten",
      "properties": {
        "session_id": {
          "type": "integer",
          "description": "Die Session ID des Benutzers"
        },
        "user_id": {
          "type": "integer",
          "description": "Die ID des Benutzers"
        }
      },
      "required": [
        "session_id",
        "user_id"
      ]
    },
    "trip": {
      "type": "object",
      "description": "Die Daten des einzutragenen Trips",
      "properties": {
        "unique_trip_id": {
          "type": "integer",
          "description": "Die unique Trip ID, also die die zum matchen benutzt wird."
        },
        "trip_id": {
          "type": "integer",
          "title": "Trip_id schema.",
          "description": "Die ID des Trips, also die ID über die Informationen über den Trip in den GTFS Daten abgerufen werden können"
        },
        "date": {
          "type": "integer",
          "description": "Das Abfahrtsdatum"
        },
        "sequence_id_target_station": {
          "type": "integer",
          "description": "Die Sequence ID des Startbahnhofes"
        },
        "sequence_id_departure_station": {
          "type": "integer",
          "description": "Die Sequence ID des Zielbahnhofes"
        },
        "destination_station_name": {
          "type": "string",
          "description": "Der Name des Zielbahnhofs"
        }
      }
    }
  }
}
```

```

    "type": "string",
    "description": "Der Name des Startbahnhofes"
},
"target_station_name": {
    "type": "string",
    "description": "Der Name des Zielbahnhofes"
},
"has_season_ticket": {
    "type": "boolean",
    "description": "Der Ticketstatus des Benutzers"
}
},
"required": [
    "unique_trip_id",
    "trip_id",
    "date",
    "sequence_id_target_station",
    "sequence_id_departure_station",
    "destination_station_name",
    "target_station_name",
    "has_season_ticket"
]
}
},
"required": [
    "Benutzer",
    "Fahrt"
]
}
}

```

5.4. Datenschutz

Das Thema Datenschutz spielte bei der Entwicklung eine große Rolle. Um das Abhören der Kommunikation zwischen Client und Server zu erschweren wurde als Übertragungsprotokoll https ausgewählt. Um den Chat zwischen den Benutzern zu schützen wird eine End-to-End Verschlüsselung implementiert, dass bedeutet das Nachrichten direkt beim Sender verschlüsselt und erst beim Empfänger entschlüsselt werden. Dazu wird Tozny²⁰, eine Open Source Bibliothek, verwendet (siehe 2.1.4.). Zum Schutz von Passwörtern werden diese noch auf dem Client gehashed und auch in dieser Form gespeichert. Um unerlaubten Zugriff auf Daten zu verhindern wird jedem eingeloggtem Nutzer eine Session ID zugewiesen, über welche er sich bei einem Zugriff authentifizieren muss. Für die asynchrone Kommunikation wird der externe Dienst Firebase Cloud Messaging²¹ genutzt. Um sicherzustellen das keine sensiblen Daten bei der Nutzung dieses Dienstes von dem Anbieter dieses Dienstes abgefangen werden wird nur mit sogenannten Light Pings gearbeitet, welche den Client dazu veranlassen eine Verbindung mit dem Server aufzubauen und sich die benötigten Daten auf direktem Weg abzuholen.

²⁰ <https://github.com/tozny/java-aes-crypto> (29.05.2016)

²¹ <https://firebase.google.com/docs/cloud-messaging/> (29.05.2016)

6. Modellierung der Architekturmerkmale

6.1 Ressourcen und Topics

Wie in dem Kommunikationsmodell (Punkt 7.5.) ersichtlich ist die Kommunikation zwischen den einzelnen Komponenten größtenteils synchron, es gibt allerdings auch einige Daten die mittels einer asynchronen Kommunikation ausgetauscht werden. Für die synchrone Kommunikation wird eine API nach dem REST-Paradigma erstellt. Für die asynchrone Kommunikation soll der Firebase Cloud Messaging Dienst, eine neuere Version des Google Cloud Messaging Dienstes, verwendet werden. Die Kommunikation zwischen dem Client und Server erfolgt über den Zugriff auf folgende Ressourcen:

Ressource	Beschreibung
users	Informationen zu Nutzern
users/sessions	Die Sitzungen von eingeloggten Nutzern
users/ratings	Die Bewertungen von Nutzern
users/pictures	Die Profilbilder von Nutzern
users/chats	Die Chaträume von Nutzern
users/chats/messages	Die Nachrichten in diesen Chaträumen
users/dt_trips	Alle eingetragenen Fahrten von Nutzern
trips	Details zu allen Fahrten die von Bahnhof a zu Bahnhof b fahren, inklusive der Anzahl an Matches für diese Fahrten
matches	Alle zu den Paramteren passenden Matches

Tabelle 3: Ressourcen Übersicht

6.1.1 synchrone Kommunikation

Die synchrone Kommunikation verläuft über eine API im Sinne des REST-Paradigmas. Im Richardson Maturity Model ist diese auf Level Zwei angeordnet, da es verschiedene URI's gibt auf welche mit verschiedenen HTTP-Methoden zugegriffen wird, der Server allerdings keine URI's welche auf die nächste Ressource verweisen übermittelt. Es wurde sich gegen das Implementieren einer Level drei Kommunikation entschieden, da aufgrund der verschiedenen Tabs nicht immer alle zur Navigation benötigten URI's auf dem Server ermittelt werden können bzw. nicht immer direkt mit der Anfrage in Verbindung stehen. Als Datenübertragungsprotokoll wird dabei https verwendet, mithilfe dessen JSON Objekte zwischen Client und Server verschickt werden (siehe Punkt 7.4.). Die synchrone Kommunikation findet durch den Zugriff auf die unter Punkt 7.9. spezifizierten URIs statt.

6.1.2 asynchrone Kommunikation

Für die asynchrone Kommunikation wird der Firebase Cloud Messaging Dienst²² genutzt. Über diesen Dienst können Benutzer direkt über eine ID oder über Topics die sie abonnieren angesprochen werden. Um die Daten des Benutzers zu schützen soll nur mit Lightpings gearbeitet werden, welche den Zugriff auf eine URI durch den Benutzer initiieren, über welche dann die aktualisierten Daten abgerufen werden. Die asynchrone Kommunikation erfolgt durch das Abonnieren von folgenden Topics.

URI	Zweck der Operation	response Datenformat	response Payload
/users/:user_id/ dt_trips/:dt_trip _id	trip wird gelöscht, partner trägt sich aus, neuer Partner trägt sich ein, Suchagent findet einen zum trip passenden Match	JSON	light-ping, dt_trip_id
/users/:user_id/ chats	neuer Chatraum wird für den Nutzer erstellt	JSON	light-ping, user_id
/users/:user_id/ chats/:chat_id/ messages	neue Chatnachricht wird in einem Chatraum des Nutzers erstellt	JSON	light-ping, user_id, chat_id

Tabelle 4: Topics

²² <https://firebase.google.com/docs/cloud-messaging/> (29.05.2016)

6.2. Skizzierung der Anwendungslogik

6.2.1. Client

6.2.1.1. Umkreissuche

Um dem Benutzer die Eingabe der Starthaltestelle zu erleichtern sollen Haltestellen die sich in seiner Nähe befinden beim Autocomplete an erster Stelle angezeigt werden. Die in den PoC's (2.1.2/3.) vorgestellte Implementation von Geolib ist dafür nicht optimal, da die Distanz zwischen den Punkten nicht zwischengespeichert wird und somit die Distanz bei einem Funktionsaufruf einmal bei der Filterung und einmal zum Sortieren berechnet wird. Um eine schnellere Ermittlung von Haltestellen in der Nähe zu implementieren wird nur noch die Distanz zwischen Haltestelle und Benutzer mittels Geolib berechnet, die Sortierung und Filterung von Haltestellen wird dann anhand dieser Distanz von Hand ausgeführt.

```
#Variablen
#area                               -der Radius in dem Haltestellen ermittelt werden
sollen
#maxNumberStations                 -die Maximale Anzahl an Haltestellen die ermittelt werden
soll
#stations[Name, lat, lon]           -ein Array das alle Bahnhöfe inklusive Koordinaten
enthält

findStationsInArea(area, maxNumberStations, stations)
    #eigene Position auslesen
    lon = position.lon
    lat = position.lat

    #Stationen im Raidus area ermitteln und in stationInArea speichern
    stationsInArea[Name][distance]
    überprüfe jede station in stations
        distance = geolib.getDistance({lon, lat},{station.lon, station.lat})
        wenn distance <= area ist
            stationsInArea.add(station)

    #Sortiere stationInArea nach distance
    stationsInArea.sort(distance)

    #Entferne die letzte station bis die gewünschte Anzahl an stations erreicht
    ist
    solange in stationsInArea mehr stations als maxNumberStations sind
        entferne die letzte station aus stationsInArea

return stationsInArea
```

6.2.1.2. Verlauf

Eine weitere Hilfe beim Ausfüllen der Suchmaske soll ein interaktiver Verlauf bilden. Damit der Benutzer immer die wichtigste Fahrt angezeigt bekommt wird mit einem Rating-System gearbeitet welches Suchanfragen ein Rating zuweist das im Laufe der Zeit abnimmt. Dabei werden Suchanfragen, welche mehrmals am selben Tag ausgeführt wurden, stärker bewertet, als Suchanfragen, welche nur einmal ausgeführt wurden. Allerdings nicht doppelt oder dreimal so stark. Dies wird durch die Benutzung einer Logarithmusfunktion erreicht. Wenn während der Laufzeit festgestellt wird, dass dem Benutzer nicht immer die gewünschten Fahrten angezeigt werden, können die Variablen innerhalb der Funktion angepasst werden. Des Weiteren ist es auch vorstellbar den Verlauf anhand einer Mischung aus Rating und der letzten gestellten Suchanfrage zu erstellen.

#Variablen:

```
#trips[           -Eine Sammlung aller bereits gesuchten Fahrten,  
                 bestehen aus  
#departureStationName   -Name des Abfahrtsbahnhofs  
#targetStationName      -Name des Zielbahnhofs  
#date                  -Suchdatum  
#rating                -Rating  
#lastCalculated]       -dem Datum an dem das Rating der gesuchten Fahrt das  
                         letzte mal errechnet wurde
```

#updated das Rating aller jemals gesuchten Fahrten

determineHistory

durchlufe jeden trip in trips

#Fahrten die seit dem letzten Verlauf dazugekommen sind

wenn lastCalculated null ist

#Suchen die öfters an einem Tag ausgeführt wurden sollen stärker als Suchen die nur einmal ausgeführt wurden gerated werden, allerdings nicht zu stark. Die Formel

$\ln(\text{anzahlIdentischerFahrten})+1$ ergibt folgende Ratingtabelle:

anzahlIdentischerFahrten	rating
1	1
2	1,693147181
3	2,098612289
4	2,386294361
5	2,609437912
7	2,945910149
10	3,302585093
15	3,708050201
20	3,995732274

für alle Fahrten mit identischem date, targetStationName und departureStationName

behalte nur einen Eintrag dieser Fahrt

rating = $1,1^{(\text{anzahlIdentischerFahrten}-1)}$

```
        lastCalculated = date

        #das Rating einer Fahrt soll jeden Tag um 5% fallen
        #solange wie lastCalculated < des heutigen Datums ist
        #erhöhe lastCalculated um einen Tag
        rating*=0.95

#Fasse alle Suchen mit gleichem Start und Zielbahnhof zusammen
für alle Fahrten mit identischem targetStationName und departureStationName
    summiere das Rating dieser Fahrten
    behalte nur einen Eintrag für diese Fahrten, mit dem summierterem Rating

#Die Suchen sollen so sortiert werden das die wichtigste am Anfang steht
sortiere trips absteigend nach rating

#Die trips sollen inklusive des neu errechneten Ratings gespeichert werden
speichere trips, überschreibe dabei die alten Einträge

return die ersten 5 Elemente von trips
```

6.2.1.3. Durchschnittliche Bewertung

Da das Serverseitige speichern und aktualisieren einer durchschnittlichen Bewertung als sehr Aufwendig eingeschätzt wurde soll die durchschnittliche Bewertung Clientseitig anhand der übermittelten Bewertungen stattfinden. Diese Implementation bietet den Vorteil, dass ein speichern der durchschnittlichen Bewertung nicht nötig ist. Da der Client zum Errechnen der Durchschnittsbewertung alle Bewertungen des zu bewertenden Benutzers braucht, könnte dies bei vielen Bewertungen problematisch werden. Es wird allerdings nicht erwartet, dass diese Anzahl an Bewertungen eines Nutzers in absehbarer Zeit erreicht wird. Des Weiteren müsste dann auch das System so umstrukturiert werden dass nicht mehr alle Bewertungen an den Client übermittelt werden.

```
#Variablen:  
#ratings[           -Eine Sammlung aller zu berücksichtigen Ratings  
#stars]            bestehen aus  
                     -der Anzahl an Sternen die jede Bewertung bekommen  
                     hat  
  
#Berechnet den durchschnitt alle Bewertungen  
averageRating(ratings[starts])  
    #Variable in der das durchschnittliche Rating gespeichert wird  
    average  
    #Summe aller Bewertungen bestimmen  
    durchlaufe jedes rating in ratings  
        average += rating.stars  
  
    #den Durchschnitt bestimmen  
    teile average durch die Anzahl an Bewertungen  
  
return average
```

6.2.2 Server

6.2.2.1. Trips ermitteln

Bevor ein Benutzer eine Fahrt erstellen kann muss ermittelt werden, welche Fahrten von seinem Start zu seinem Zielbahnhof fahren. Dabei wurde die Möglichkeit eines Umstieges nicht beachtet, um die Anzahl an potentiellen Matches zu maximieren. Zur Ermittlung der Fahrten werden erst alle Fahrten ermittelt die von seinem Start zum Zielbahnhof fahren. Daraufhin wird ermittelt welche Fahrten am Abfahrtsdatum als nächstes, nach der Abfahrtszeit, am Startbahnhof losfahren. Dazu wird mithilfe der ServiceID einer Fahrt nachgeschaut ob diese überhaupt am Abfahrtstag fährt. Wenn nicht genügend Fahrten gefunden wurden, wird diese Suche für den Nächsten Tag wiederholt. Zum Schluss wird die Anzahl an Matches für jede Fahrt ermittelt, damit diese dem Benutzer sofort angezeigt werden können.

#Variablen:

#departureStationName	-Names des Startbahnhofs
#targetStationName	-Name des Zielbahnhofs
#departureTime	-Abfahrtenzeit
#departureDate	-Abfahrtendatum
#hasSeasonticket	-Boolean der Auskunft über den Ticketstatus des
#userID	-Die eindeutige ID des Benutzers

#Ermittelt die nächsten Zehn Fahrten von Bahnhof a nach Bahnhof b ab einer spezifischen Zeit und einem spezifischem Datum

```
findTrips(departureStationName, targetStationName, departureTime, departureDate,
hasSeasonticket, userID)
```

```
    departureStationID = stops-Datenbankabfrage(departureStationName)
    targetStationID = stops-Datenbankabfrage(targetStationName)
```

```
#alle Fahrten die an einem der Bahnhof halten ermitteln
departureTrips[tripID][departureTime][sequenceID] =
stop_times-DBabfrage(departureStationID)
targetTrips[tripID][arrivalTime][sequenceID] =
stop_times-DBabfrage(targetStationID)
```

```
#alle Fahrten die an beiden Bahnhöfen halten ermitteln
trips[tripID][departureTime][arrivalTime][sequenceID][DepartureStation][sequenc
eID][TargetStation][serviceID]
für jede tripID die in departureTrips und in targetTrips vorkommt
#Die Fahrt fährt in die richtige Richtung
wenn departureTrip.sequenceID < targetTrip.sequenceID ist
#Fahrten die an beiden Bahnhöfen vorkommen speichern
```

```

        trips.add(departureTrip.tripID, departureTrip.departureTime,
targetTrip.arrivalTime, departureTrip.sequenceID,
targetTrip.sequenceID, trips-DBabfrage(departureTrip.tripID)

#hier werden die nächsten 10 Fahrten von Startbahnhof zum Zielbahnhof
gespeichert
uniqueTrips[uniqueTripID][tripID][departureTime][arrivalTime][sequenceIDDepartureStation][sequenceIDTargetStation][departureDate][numberMatches]

#es gibt keine Linie die an beiden Bahnhöfen hält
wenn trips leer ist
    return Error(Es gibt keine Linie, welche die beiden Bahnhöfe
    miteinander verbindet)
#es gibt wenigstens eine Linie die an beiden Bahnhöfen hält
wenn trips nicht leer ist
    sortiere trips nach arrivalTime
    #Fahrten die am selben Tag stattfinden
    überprüfe jeden trip in trips
        wenn trip.departureTime >= departureTime ist und
            calendar-Datenbankabfrage(trip.serviceID) am Wochentag von
            departureDate == 1 ist und
            calendar_dates-Datenbankabfrage(trip.serviceID) an departureDate
            == 1 oder null ist
                wenn die departureTime des trips >= 24:00:00 ist
                    #erstellen einer Einzigartigen ID, Zug ist am
                    Vortag losgefahren
                    uniqueTripID = trip.tripID + (departureDate-1)
                ansonsten
                    #erstellen einer Einzigartigen ID, Zug ist am
                    selben Tag losgefahren
                    uniqueTripID = trip.tripID + departureDate

    #Fahrten inklusive der uniqueTripID speichern
    uniqueTrips.add(uniqueTripID, trip.tripID,
trip.departureTime, trip.arrivalTime,
trip.sequenceIDDepartureStation,
trip.sequenceIDTargetStation, trip.departureDate, null)

#suche weitere Fahrten bis insgesamt 10 gefunden wurden
currentDepartureDate = departureDate
solange noch keine 10 Fahrten in uniqueTrips gespeichert wurden
    erhöhe currentDepartureDate um einen Tag

```

```

#keine Fahrten die weiter als eine Woche vom Startdatum entfernt
sind
wenn currenDepartureDate <= departureDate + 7 ist
    breche die Suche nach weiteren Fahrten ab
ansonsten
    überprüfe jeden trip in trips
        wenn calendar-Datenbankabfrage(trip.serviceID) am
        Wochentag von departureDate = 1 und
        calendar_dates-Datenbankabfrage(trip.serviceID) am
        departureDate = 1 oder null ist
            wenn die departureTime des trips >= 24:00:00 ist
                #erstellen einer Einzigartigen ID, Zug ist am
                Vortag losgefahren
                uniqueTripID = trip.tripID + (departureDate-1)
            ansonsten
                #erstellen einer Einzigartigen ID, Zug ist am
                selben Tag losgefahren
                uniqueTripID = trip.tripID + departureDate

#Fahrten inklusive der uniqueTripID speichern
uniqueTrips.add(uniqueTripID, trip.tripID,
trip.departureTime, trip.arrivalTime,
trip.sequenceIDDepartureStation,
trip.sequenceIDTargetStation, trip.departureDate,
null)

#Für die Anzeige im Client müssen die Anzahl an Matches für eine Fahrt
ermittelt werden
überprüfe jeden uniqueTrip in uniqueTrips
    uniqueTrip.numberMatches =
    findNumberMatches(trip.sequenceIDDepartureStation,
    trip.sequenceIDTargetStation, trip.uniqueTripID, userID,
    hasSeasonticket)

return uniqueTrips

```

6.2.2.2. Erweitertes Matching

Das erweiterte Matching funktioniert mithilfe der uniqueTripID. Diese ID setzt sich aus der TripID der Fahrt und dem Datum an dem die Fahrt beginnt zusammen. Jede Fahrt hat innerhalb der GTFS Datenbank diese einzigartige TripID, mit welcher diese Fahrt identifiziert wird. Da eine Fahrt allerdings einmal pro Tag stattfinden kann, muss diese ID durch das Datum der Fahrt erweitert werden, um eine ID zu erhalten, die für alle Personen auf dieser Fahrt, also z.B. alle Personen die an einer beliebigen Haltestelle in die RB 25 einsteigen, welche am 28.05.2016 um 15:18 in Meinerzhagen losfährt, identisch ist. Nachdem alle Personen mit der selben uniqueTripID ermittelt wurden muss nur überprüft werden ob diese einen zueinander passenden Ticketstatus haben und ob die Reihenfolge der Haltestellen für ein Matching geeignet ist, also der Dauerticketbesitzer die gesamte Strecke seines Mitfahrers im Zug mitfährt.

#Variablen:

#sequenceIDDepartureStation	-Sequence-ID des Startbahnhofs
#sequenceIDTargetStation	-Sequence-ID des Zielbahnhofes
#uniqueTripID	-Trip-ID die Auskunft über die Fahrt und den Tag der Fahrt gibt
#userID	-Eindeutige ID des Benutzers
#hasSeasonticket	-Boolean der Angibt ob der Benutzer ein Dauerticket besitzt

```
#Ermittelt alle Matches zu einer bestimmten Fahrt
findMatches(sequenceIDDepartureStation, sequenceIDTargetStation, uniqueTripID,
userID, hasSeasonticket)
    matches[] = dt_trips-DBabfrage(uniqueTripID)

    #Dauerticket vorhanden
    if(hasSeasonTicket)
        foreach match in matches
            #Überprüfung ob der Dauerticketbesitzer die gesamte Strecke
            #seines Mitfahrers im Zug mitfährt, nicht selber die Fahrt
            #initiiert hat und die Fahrt noch nicht gematcht wurde
            if (sequenceIDDepartureStation >
                match.sequenceIDDepartureStation & sequenceIDTargetStation <
                match.sequenceIDTargetStation & match.userID != userID &
                match.partner.userID = null & match.hasSeasonTicket = false)
                matches.remove(match)

    #kein Dauerticket vorhanden
    if(!hasSeasonTicket)
        foreach match in matches
            if (sequenceIDDepartureStation >
                match.sequenceIDDepartureStation & sequenceIDTargetStation <
                match.sequenceIDTargetStation & match.userID != userID &
                match.partner.userID = null& match.hasSeasonTicket = true)
```

```
        matches.remove(match)
    return matches

#Ermittelt die Anzahl an Matches zu einer bestimmten Fahrt
findNumberMatches(sequenceIDDepartureStation, sequenceIDTargetStation, uniqueTripID,
userID, hasSeasonticket)
    matches[] = dt_trips-DBabfrage(tripID, date)
    numberMatches = matches.length

#Dauerticket vorhanden
if(hasSeasonTicket)
    foreach match in matches
        #überprüfung ob der Dauerticketbesitzer die gesamte Strecke
        #seines Mitfahrers im Zug mitfährt, nicht selber die Fahrt
        #initiiert hat oder die Fahrt noch nicht gematcht wurde
        if (sequenceIDDepartureStation >
            match.sequenceIDDepartureStation & sequenceIDTargetStation <
            match.sequenceIDTargetStation & match.userID != userID &
            match.partner.userID = null & match.hasSeasonTicket = false)
            numberMatches --
#kein Dauerticket vorhanden
if(!hasSeasonTicket)
    foreach match in matches
        if (sequenceIDDepartureStation >
            match.sequenceIDDepartureStation & sequenceIDTargetStation <
            match.sequenceIDTargetStation & match.userID != userID &
            match.partner.userID = null & match.hasSeasonTicket = true)
            numberMatches --
return numberMatches
```

6.2.2.3. Suchagent

Der Suchagent soll Benutzer, die bereits eine Fahrt eingetragen haben, darüber benachrichtigen, dass eine Fahrt die sich mit ihrer Matchen kann, eingetragen wurde. Dazu werden beim Eintragen einer Fahrt in die Datenbank alle zu ihr passenden Matches ermittelt und über eine von ihnen abonnierte URL angepingt.

#Variablen:

#sequenceIDepartureStation	-Sequence-ID des Startbahnhofs
#sequenceIDTargetStation	-Sequence-ID des Zielbahnhofs
#uniqueTripID	-Abfahrtendatum
#userID	-Eindeutige ID des Benutzers der sich eingetragen hat
#hasSeasonticket	-Boolean der Angibt ob der Benutzer ein Dauerticket besitzt

#wird aufgerufen wenn sich ein Benutzer für einen Trip in die Datenbank einträgt

```
noticeMatches(sequenceIDepartureStation, sequenceIDTargetStation, uniqueTripID,
userID, hasSeasonticket)
```

#matches für den neuen Eintrag ermitteln

```
matches[] = findMatch(sequenceIDepartureStation, sequenceIDTargetStation,
uniqueTripID, userID, hasSeasonticket)
```

#Die Matches benachrichtigen das es einen neuen passenden Eintrag gibt
durchlaufe jeden match in matches

```
ping(user/matches.userID/dt_trips/matches.uniqueTripID,
"Es wurde ein neuer Match gefunden")
```