Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

HOCHSCHULE FÜR TECHNIK UND WIRTSCHAFT

PROJEKTARRBEIT

Erfassung, Auswertung und Visualisierung von routenbasierten Positionsdaten anhand der Omnibusse der BVG

Technik mobiler Systeme
Ausgewählte Kapitel mobiler Anwendungen

Pascal Dettmers (551733)
Stefan Neuberger (553849)
Tobias Ullerich (553746)

Betreuende Dozenten:

Prof. Dr. Alexander Huhn
Prof. Dr.-Ing. Thomas Schwotzer

4. Januar 2018

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ADB Android Debug Bridge

BVG Berliner Verkehrsbetriebe

JDBC Java Database Connector

ORM Object Relational Mapping

OSM Open Street Map

RBL rechnerbasiertes Leitsystem

Glossar

Linie ist ein Sammlung von Routen und beschreibt im öffentlichen Nahverkehr in der Regel eine Strecke von einem Startpunkt A nach einem Endpunkt B und umgedreht.

Journey ist eine Fahrt auf einer Route, um Zeitmessungen für die Berechnung zu machen.

Route ist ein Begriff, der eine konkrete Strecke von einem Startpunkt A nach einem Endpunkt B beschreibt.

1 Dokumentengeschichte

Zeitraum	TPL/Autor(en)	Änderungen
Wintersemester 2017/18 (09.11.2017)	Tobias Ullerich	Initiale Dokumentenstruktur Entwurf Aufgabenstellung
Wintersemester 2017/18 (01.12.2017)	Tobias Ullerich	Analyse BVG Datenbank 1/2
Wintersemester 2017/18 (05.12.2017)	Tobias Ullerich	Analyse BVG Datenbank 2/2
Wintersemester 2017/18 (19.12.2017)	Tobias Ullerich	Update Dokumentenstruktur Update Aufgabenstellung
Wintersemester 2017/18 (23.12.2017)	Tobias Ullerich	Dokumentation Rest API (Route, Journey)
Wintersemester 2017/18 (24.12.2017)	Tobias Ullerich	Dokumentation Rest API (Vehicle)
Wintersemester 2017/18 (27.12.2017)	Tobias Ullerich	Dokumentation Server Datenbank
Wintersemester 2017/18 (30.12.2017)	Tobias Ullerich	Dokumentation Funktionsweise
Wintersemester 2017/18 (31.12.2017)	Tobias Ullerich	Dokumentation Programmiersprache & Bibliotheken
Wintersemester 2017/18 (02.01.2018)	Tobias Ullerich	Dokumentation Rest API (Vehicle History)
Wintersemester 2017/18 (03.01.2018)	Tobias Ullerich	Dokumentation Deployment/Runtime
Wintersemester 2017/18 (04.01.2018)	Stefan Neuberger	Dokumentation OSM, GeoJSON, Problemstellung, Komponenten
Wintersemester 2017/18 (04.01.2018)	Pascal Dettmers	Dokumentation Ausblick, Bibliotheken

Tabelle 1.1: Dokumentengeschichte

2 Problemstellung

Ziel des Projektes ist es ein System zu entwerfen um Bus bunching (Pulkbildung) im öffentlichen Nahverkehr, genauer bei den Bussen der Berliner Verkehrsbetrieben, zu verhindern.

Bus bunching ist ein Phänomen im öffentlichen Personennahverkehr bei dem es durch Beeinträchtigungen im Verkehrsfluss zu einer zusammenhängenden Fahrzeugreihe kommt. Daraus resultieren starke Schwankungen in den Folgezeiten der Busse einer Linie welche wiederum eine unregelmäßige Belastung der Fahrzeuge mit Fahrgästen nach sich zieht.

Die hier angestrebt Lösung besteht darin, jedem Busfahrer eine Android Applikation zur verfügung zu stellen, auf der der relative Zeitabstand zu Bussen direkt vor und hinter ihm visualisiert wird. Der eigene Bus wird durch eine Fahrtennummer im Vorfeld festgelegt. Der Fahrer kann, entsprechend der angezeigten Informationen, sein Fahrverhalten so anpassen das der Takt zwischen den Fahrzeugen den Vorgaben der Leitzentrale entspricht. Er kann beispielsweise, wenn der Bus vor ihm durch eine Beeinträchtigung auf der Strecke verspätet an einer Haltestelle ankommt und sich somit der relative Zeitabstand verringert, langsamer fahren oder an der nächsten Haltestelle so lange warten bis der Zeitabstand wieder der Strecken Taktung entspricht.

Grundlage des Systems ist das Senden der Koordinaten jedes Buses aus der App heraus. Diese Koordinaten und die Fahrtdauer zwischen den Koordinaten werden in einer Datenbank gespeichert. Aus der Menge der Datensätze einer Route ist es möglich durchschnittliche Fahrzeiten für die Route zu unterschiedlichen Tageszeiten zu berechnen. Diese Berechnungen erlauben eine Vorhersage der relativen zeitlichen Distanz zwischen zwei Bussen einer Linie zu berechnen.

3 Aufgabenstellung

Aus der Problemstellung von Abschnitt 2 ergeben sich für das Projekt folgende Anforderungen. Das zu entwickelnde System hat die primäre Aufgabe, den zeitlichen Abstand von zwei Objekten auf einer Route zu bestimmen. Im konkreten Anwendungsfall handelt es sich bei diesen Objekten um Omnibusse der Berliner Verkehrsbetriebe. Zu entwickeln ist eine Android Applikation, die die Positionsdaten der Omnibusse erhebt, und Abstand der Objekte in Weg und Zeit visualisiert darstellt. Speziell ist der Abstand zum Vorgänger und Nachfolger von Interesse. Zudem ist eine Persistierung der Daten in eine Datenbank gefordert.

4 Architektur

4.1 Grundlagen

4.1.1 Analyse Datenbank BVG

Die BVG nutzt für die Persistierung der Daten, inklusive der Prozessdaten, ein Datenbanksystem der Firma Oracle. Es werden bei der BVG zwischen zwei verschiedenen Systemen unterschieden. Zum Einen gibt es die sogenannte SC05 Schnittstelle. Diese enthält Prozessdaten der aktuellen Betriebslage. Dazu zählen unter anderem Positionen von Bussen und deren Verspätung (vgl.: „Die Prozessdatenschnittstelle (SC05) spiegelt die aktuelle Situation im RBL wider." [1, S. 4]). Zum Anderen gibt es die SC51 Datenbank, entwickelt von der Firma Alcatel. Diese Schnittstelle enthält unterschiedlichste Daten für die Durchführung des öffentlichen Nahverkehrs der BVG. Darunter fallen Informationen zu Linien (Bus und Bahn), Informationen über deren Routen mittels geografischer Koordinaten und vieles mehr. Für die Analyse dieser relationalen Datenbanken waren jeweils Dokumentationen und ein Dump der Datenbank zur Verfügung.

Der erste Schritt bei der Analyse bestand darin, die Dumps der Oracle Datenbanken zu importieren, um anschließend Zugriff auf die Tabellen und deren Daten zu erlangen. Für den Import fiel die Entscheidung für das Tool „OraDump to MySQL" (vgl. [2]). Mit diesem Tool ist es möglich ein Oracle Datenbank Dump in eine MySQL Datenbank zu importieren. Vorteil dieser Methode ist, das auf bestehende Kenntnisse im Umgang mit MySQL zurückgegriffen werden kann. Im folgenden wurde mittels der Schnittstellendokumentation die Struktur der Datenbank analysiert. Im Fokus dieser Analyse stehen die Routen Information aus der SC51 und die Positionsdaten der Fahrzeuge aus der SC05 Schnittstelle. Bei der Analyse haben sich folgende Datenbanktabellen als wertvoll gezeigt.

Die Tabelle `CM_VEHICLE_POSITION` aus der SC05 Datenbank enthält Informationen zu der aktuellen geografischen Position mittels Latitude und Longitude, der Abweichung vom Sollfahrplan in Sekunden, sowie eine Zuordnung zu einer Route. Um einen Omnibus auf einer Route einzuordnen, gibt es eine endliche Menge von geografischen Punkten. Zu all diesen Punkten ist ein zeitlicher und örtlicher Abstand bekannt (siehe SC51). Zu jedem Fahrzeug ist der letzte passierte Punkt der Route referenziert (`LAST_POR_ORDER`). Der prozentuale Abstand zum Folgepunkt auf einer Route ist ebenfalls in der Relation durch die Spalte `REL_LNK_DISTANCE` gegeben.

Für die Zuordnung der Fahrzeuge aus der Tabelle `CM_VEHICLE_POSITION` zu einer Route und einem Kurs gibt es in der SC05 Datenbank zwei Tabellen. Zum Einen hat die Tabelle `CM_ACCT_COURSES` die Aufgabe, ein Fahrzeug einem Kurs zuzuordnen. Zum Anderen wird durch `CM_ACCT_JOURNEY` ein Bus einer Route zugeordnet. Somit können die `POINTS_ON_ROUTE` einem Fahrzeug zugeordnet werden.

Die Datenbank SC51 beinhaltet Tabellen für die Linien (`LINES`). Eine Linie ist im Kontext der BVG zum Beispiel die konkrete Buslinie X11. Jede Linie besteht aus mehren Fahrten, hier `COURSES_ON_JOURNEY` genannt. Zu einem Kurs gehören Informationen wie Startzeit, Endzeit und eine Kursnummer, die nur im Kontext einer Linie eindeutig ist.

Die geografischen Informationen zum Routenverlauf werden in den Tabellen `ROUTE`, `POINTS_ON_ROUTE` und `NETWORK_POINTS` verwaltet. Zu einer Buslinie können verschiedene Routen gehören. Diese Routen sind in der Tabelle `ROUTE` zu finden. Dafür enthält auch diese Relation zusätzlich ein Feld `Description` (Beispieldaten: Falken-

see, Bahnhof->S+U Rathaus Spandau). Die Tabelle `POINTS_ON_ROUTE` koordiniert die Punkte einer Route, indem jeder Punkt einen Laufnummer hat (`POR_ORDER`). Um den zeitlichen und örtlichen Abstand zwischen zwei Punkten zu ermitteln, wird die Relation `LINKS` verwendet. Die Tabelle `NETWORK_POINTS` enthält abschließend die eigentlichen geografischen Punkte in der Form Latitude und Longitude.

In der Abbildung 4.1 sind die Zusammenhänge der einzelnen Datenbanktabelle von SC05 und SC51 zu sehen. Dabei handelt es sich lediglich um einen Auszug der relevanten Daten für das zu entwickelnde System.

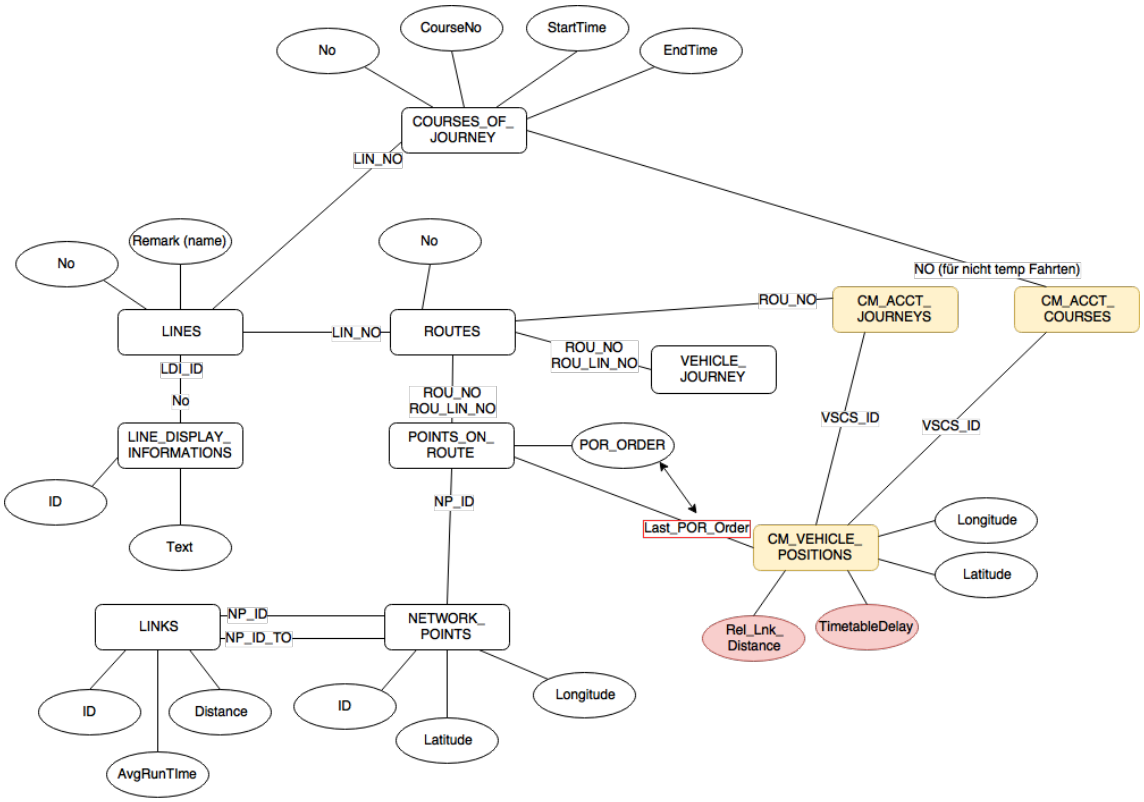

Abbildung 4.1: Datenbank Schema aus SC05 und SC51

4.1.2 Open Street Map

OpenStreetMap ist ein freies Projekt unter der Open Database License welches Geodaten sammelt und zur freien Nutzung in einer Datenbank speichert (vgl.: [3]). Das Projekt wurde im Juli 2004 ins Leben gerufen. Seit April 2006 ist ein Gremium, die OpenStreetMap Foundation, für das Projekt verantwortlich. Ziel ist das „Erzeugen, Verteilen und Vergrößern eines geographischen Datenbestandes sowie dessen freie Bereitstellung zum allgemeinen Gebrauch?"(vgl.: [4]).

Der Zugang zu den Daten erfolgt über die OSM API. Zum Zeitpunkt des Erstellen dieses Dokumentes in der Version 0.6 (vgl.: [5]). Eine Ausführliche Dokumentation wird auf openstreetmap.org zur Verfügung gestellt (vgl.: [6]). Jedem auf OSM gespeichertem Objekt wird eine Basiseigenschaft (Attribut) zugeordnet welche wiederum durch eine festgelegte Datenrepräsentation ausgedrückt wird. Die Liste der Attribute (Map Features) findet sich auf (vgl.: [7]).

Für dieses Projekt ist insbesondere das Map Feature „Route"von interesse (vgl.: [8]). Die spezifikation für bestimmte Routen erfolgt über eine Sammlung von Key/Value paaren. So sind Busrouten der Berliner Verkehrsbetriebe Relationen mit folgenden Eigenschaften:

```
["type"="route"]["route"="bus"]["operator"="Berliner␣
    Verkehrsbetriebe"].
```

Für das Suchen, Anzeigen und downloaden von bestimmten Relationen empfehlen wir Overpass turbo (vgl.: [9]), ein webbasiertes Datensammelwerkzeug für OSM. Overpass turbo zeigt Resultate einer Anfrage auf einer interaktiven Karte und erlaubt den Download der Daten in unterschiedlichen Formaten. In diesem Projekt wurde GeoJSON benutzt (siehe 4.1.3). Eine Ausführliche Beschreibung der Overpass API findet sich unter (vgl.: [10]).

Listing 4.1: Overpass Query für alle Busrouten der Berliner Verkehrsbetriebe

```
[out:json][timeout:25];
(
  // query part
  relation["type"="route"]["route"="bus"]["operator"="
      Berliner␣Verkehrsbetriebe"];
);
// print results
out body;
>;
out skel qt;
```

Listing 4.2: Overpass Query für die Straßenbahnroute der Linie 67 der Berliner Verkehrsbetriebe

```
[out:json][timeout:25];
// gather results
(
  // query part for: ?type=route and route=bus?
  relation["type"="route"]["route"="tram"]["ref"=67]["
      operator"="Berliner␣Verkehrsbetriebe"];
);
// print results
out body;
>;
out skel qt;
```

4.1.3 GeoJson

Die Daten zu den Buslinien wurden über die OSM API im GeoJSON Format gedownloadet. GeoJSON ist ein offenes Format um geographische Daten zu repräsentieren. Es basiert auf JSON, der JavaScript Object Notation, und wurde im August 2016 als RFC 7946 veröffentlicht (vlg.: [11]) . GeoJSON erlaubt folgende geometrische Typen:

- Point: definiert durch Coordinaten (Lat, Long). Beschreiben beispielsweise Adressen
- LineString: Sammlung von Punkten welche verbunden beispielsweise Straßen und Flüsse repräsentieren Polygon: mehrere miteinander verbundene Punkte wobei der erste und letzte Punkt identisch sind. Beschreiben z.B. Ländergrenzen
- MultiPoint: mehrere nicht verbundene Punkte

- MultiLineString: bestehen aus 1 bis n LineStrings. Werden benutzt wenn zu einer Relation mehrere LineStrings existieren die dieser zugeordnet sind. z.B. unterschiedliche Fahrtstrecken für eine Buslinie.
- MultiPolygon: zur Darstellung von komplexen Flächen benützt zB bei geographisch getrennt liegenden Flächenstücken welche einer Relation angehören.

Die in diesem Projekt benutzte Daten zum repräsentieren der Routen bestehen aus MultiLineStrings. Ein GeoJSON hat ein eltern Objekt welches üblicherweiser ein collection Objekt ist. Listing 4.3 ist ein verkürztes Beispiel der Daten für die Straßenbahn Linie 27. Der Ausschnitt zeigt ein feature Objekt des eltern Objektes (FeatureCollection). Das Feature besitzt mehrere properties (Eigenschaften) und eine Geometry. Diese ist ein MutliLineString, also eine Sammlung von Punkten welche verbunden die Route der Tramlinie 27 repräsentieren.

Listing 4.3: Beispiel Relation Route 27

```
{ "type": "Feature",
      "properties": {
        "@id": "relation/2077646",
        "from": "Krankenhaus␣Köpenick",
        "name": "Straßenbahnlinie␣27:␣Krankenhaus␣Köpenick␣=>
           ␣Weißensee,␣Pasedagplatz",
        "network": "Verkehrsverbund␣Berlin-Brandenburg",
        "operator": "Berliner␣Verkehrsbetriebe",
        "ref": "27",
        "route": "tram",
        "to": "Weißensee,␣Pasedagplatz",
        "type": "route"
      },
      "geometry": {
        "type": "MultiLineString",
        "coordinates": [
          [
            [13.5939995, 52.4385062], ..., [13.5921062,
               52.4388869]
          ]
        ]
      }
}
```

4.2 Komponenten

Bestandteile des Systems beschreiben (Server, App, ...)

4.3 Datenbankschema Server

Für die Persistierung der Daten des Servers wurde eine relationale Datenbank gewählt. Speziell für die Implementierung wird eine MySQL Datenbank gewählt. Um auf die Datenbank aus Java zuzugreifen, wird der Java Database Connector (JDBC) verwendet.

In der Datenbank werden Daten der Route und deren Geometrien aus OSM gespeichert. Dafür gibt es die Tabellen `Route`, `MultiLineString` und `LineString`. Die Re-

lation `Route` enthält Informationen über die Liniennummern (`rel`), Start- und Zielhaltestelle (`from`, `to`). Zudem sind Informationen über das Verkehrsunternehmen (`operator`, Beispiel: BVG) und den Verkehrsverbund (`network`, Beispiel: VBB) vorhanden. Die Eigenschaft `type` entscheidet über den Typ der Geometrie einer Route. Die eigentliche Information über die Geometrie befindet sich als String in der Tabelle `MultiLineString` oder `LineString`. Ein solcher MultiLineString kann beispielsweise wie folgt aussehen:

```
1 MultiLineString((13.6920278 52.4516269, 13.6923877
      52.4515733, 13.6926666 52.4515393, ...))
```

Um Aussagen über die Geschwindigkeit von Fahrzeugen auf einer Route treffen zu können, wurden im Vorfeld Testfahrten gemacht. Die gesammelten Daten während dieser Testfahrten sind in den Tabellen `Journey` und `MeasurePoint` persistiert. Die Tabelle `Journey` beschreibt eine spezielle Fahrt zu einer Uhrzeit auf einer Route. Die Messwerte, also die GPS Position eines Fahrzeugs und Zeitstempel, werden zu einer Fahrt in der Relation `MeasurePoint` gespeichert. Bei den GPS Daten handelt es sich um die ungeglätteten Daten, das heißt also, dass die GPS Daten nicht zwingend auf der Route liegen.

In der Tabelle `Vehicle` werden die aktuellen Informationen zu den Fahrzeugen gespeichert. Dafür enthält die Tabelle Informationen zu GPS Position (geglättet), einen Zeitstempel und die Route eines Fahrzeuges. Jedes Fahrzeug wird über einen Unique Identifier identifiziert (`ref`).

Um diese Fahrzeugdaten historisch zu persistieren, gibt es zusätzlich zu `Vehicle` die Relation `VehicleHistory`. In dieser Tabelle existiert die gleiche Struktur wie `Vehicle`. Zusätzlich gibt es noch einen künstlichen Primärschlüssel, um jede Zeile eindeutig zu identifizieren. Somit ist es möglich zu jedem Zeitpunkt herauszufinden, auf welcher Route und an welcher Stelle ein Fahrzeug zu einem Zeitpunkt war.

Die Abbildung 4.2 visualisiert die erstellte Datenbankstruktur mit den Relationen und deren Beziehungen für den Server.

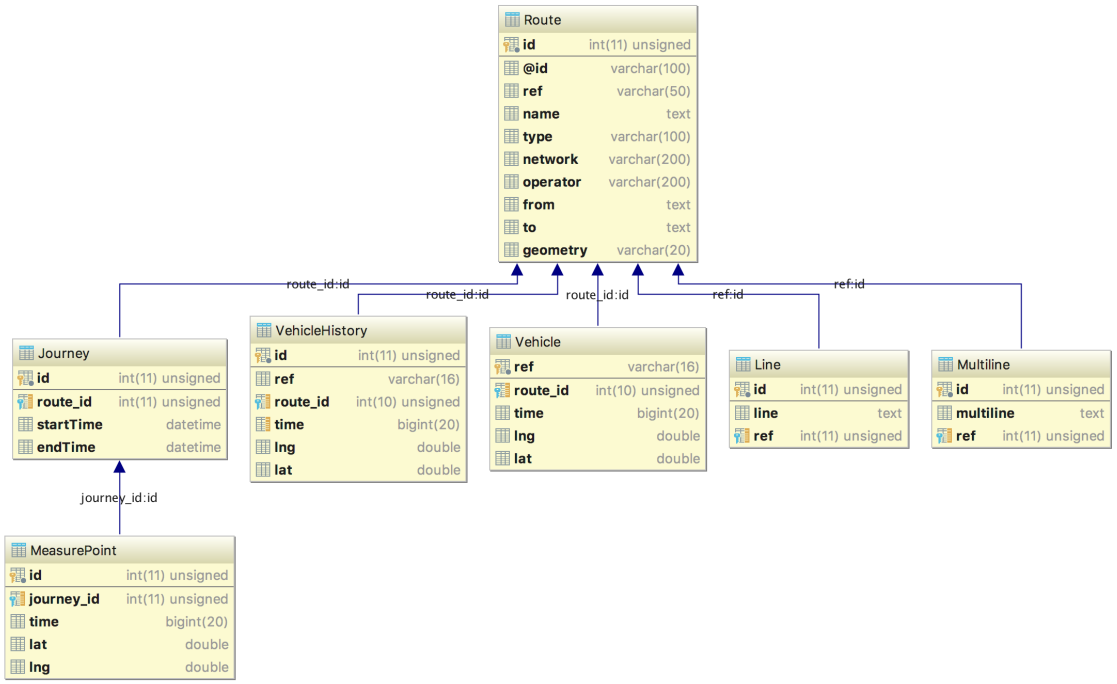

Abbildung 4.2: Datenbank Schema Server

5 Nutzung

5.1 Code

5.1.1 Programmiersprache

Um die optimale Programmiersprache für das Projekt finden zu können, spielen mehrere Faktoren eine Rolle. Zum Einen muss die Zielplattform berücksichtig werden, aber auch die eigene Fähigkeit, mit der Sprache und den damit verbunden Bibliotheken umzugehen. Im Fall der Android Anwendung, ist Java die Programmiersprache der Wahl. Eine Alternative wäre hier beispielsweise Kotlin. Die Vorteile von Kotlin gegenüber Java überliegen klar, da Kotlin sicherer und moderner ist (vgl. [12]). Jedoch ist die eigene Erfahrung der Autoren in Java gegenüber Kotlin mehr vorhanden, was schlussendlich die Entscheidung beeinflusst hat.

Für den Server ist die Wahl der Programmiersprachen deutlich umfangreicher. Die eigene einzige Vorgabe ist, das es sich bei dem Server um eine Rest API mit unterliegendem HTTP-Server handeln soll. Bei dem Server wird auf eine Mischung aus Java und Scala gesetzt. Auch hier spielt die Erfahrung eine entscheidende Rolle. Die Mischung aus Scala und Java funktioniert nur deshalb, da Scala Code zu Java Byte Code compiliert wird. Somit ist es möglich, Java Klassen und Methoden einschränkungsfrei in Scala zu nutzen. Die andere Richtung funktioniert nicht einschränkungsfrei. Aus diesem Grund sind die Modellklassen, die HTTP Handler und Datenbankklassen alle in Java geschrieben. Somit müssen keine Kompromisse zwischen Scala und Java gemacht werden. sämtliche Berechnungen sind in Scala implementiert. Der größte Vorteil von Scala, der auch die Entscheidung beeinflusst hat, ist die funktionale Programmierung, welches sich beim Auswerten von Liste und Arrays als hilfreiches Paradigma darstellt (vgl. [13]).

5.1.2 Bibliotheken

Gson

Gson ist eine Java Bibliothek mithilfe derer man Java Objekte leicht in JSON Objekte konvertieren kann und umgekehrt. So erspart man sich viel Zeit die Java Objekte per Hand zu mappen. Eine Alternative zu dieser Bibliothek wäre Jackson gewesen. Ein großer Vorteil von Jackson ist es, nicht-JSON Formate Dekodieren kann was jedoch bei diesem Projekt nicht von Nöten gewesen ist. Aufgrund der größeren Erfahrung wurde Gson verwendet.

Google Maps API

Da aus Übersichtsgründen eine Karte benötigt wurde die Google Maps API verwendet. Jedoch standen einige Alternativen zur Verfügung, wie zum Beispiel OpenStreetMaps oder auch MapQuest. Obwohl die Auswahl recht groß war, fiel die Wahl auf die Google Maps API, da sie eine sehr ausführliche und leicht verständliche Dokumentation besitzt. Darüber hinaus unterstützt Android Studios die API, beispielsweise bei der Erstellung einer Activity. Dort hat existiert die Option, direkt eine Map Activity samt Karten-layout und Beispiel Code zu generieren. Auch die Implementierung von GeoJSON wurde durch Google Maps unterstützt.

GeoJson

GeoJSON ist eine erweiterte Form des JSON-Formates, dass geografische Daten auf Karten wie OpenStreetMaps oder Google Maps repräsentieren kann. Mithilfe vom GeoJSON konnten Bus und Tram Strecken gut und Nachvollziehbar auf der Karte angezeigt werden. Die Entscheidung GeoJSON zu verwenden wurde deshalb getroffen, weil auch hier wieder eine von Android gut formulierte Dokumentation und dieses Format Google Maps unterstützt.

Asynchronous http client for Android by James Smith

Dieser Asynchrone „callback-based" http client für Android ermöglicht es schnell und einfach Anfragen zum Server zu schicken bzw. Antworten vom Server zu erhalten. Dies ersparte eine Menge Zeit und Mühe selber passende Funktionen zu schreiben und darüber hinaus wird diese Bibliothek auch von großen Apps wie Instagram oder Heyzep verwendet.

Server

Da der Server eine Rest API implementieren soll, wird dafür ein entsprechendes Framework benötigt. Für die Programmiersprache Java gibt es zahlreiche Frameworks für die Umsetzung von Rest APIs. In diesem Projekt wurde sich für die Bibliothek Spark in Version 2.7.1 entschieden (vgl. [14]). Andere Frameworks wie Beispielsweise Spring WebMVC wären für dieses Projekt zu umfangreich gewesen.Spark ist ein einfaches Framework um Webservices zu erstellen (vgl. [14]). Im Hintergrund arbeitet für die HTTP Anfragen ein Jetty HTTP Server der Eclipse Foundation.

Datenbankanfragen werden in Java mittels der Schnittstelle JDBC gesendet. Als Datenbank kommt ein MySQL Server zum Einsatz. Dementsprechend wird die Bibliothek „mysql-connector-java" in der Version 6.0.5 verwendet. Auch wird bewusst auf eine ORM Bibliothek verzichtet, um maximale Kontrolle über die Datenbankstruktur und Abfragen zu haben.

Als Übertragungsformat für den Payload in den HTTP Anfragen, wird JSON als Format gewählt. Für die serverseitige Verarbeitung wird die Bibliothek jackson und die Erweiterung geojson-jackson genutzt. Jackson ermöglicht es, JSON Daten direkt auf Java Objekte zu mappen. Dafür sind die Abhängigkeiten jackson-annotations und jackson-databind nötig. Jackson und die dazugehörigen Module werden in Version 2.9.2 verwendet.

5.2 Funktionsweise

Um den zeitlichen Abstand von zwei Fahrzeugen auf einer Route zu ermitteln, sind vorab mehrere Schritte notwendig. Grundlage der Berechnung sind die GPS Koordinaten der Fahrzeuge und der Routenverlauf. Zudem werden sogenannte Messfahrten (Journey genannt) auf der Route benötigt, um die zeitliche Entfernung näherungsweise zu bestimmen.

Als erstes müssen alle Fahrzeuge, die auf der gleichen Route fahren, aus der Datenbank ermittelt werden. Bei den GPS Daten zu den einzelnen Fahrzeugen handelt es sich um bereits bearbeitete GPS Koordinaten. Die Koordinaten werden vorab im PUT Request (A.9) bearbeitet, das heißt die ungenaue Position des Handys wird auf den dichtesten Punkt der Route verschoben. Mit diesen Koordinaten wird anschließend für jedes Fahrzeug die Entfernung zum Startpunkt der Route berechnet.

Für die Berechnung der Entfernung zwischen zwei Koordinaten auf der Route (LineString), wird jeweils den Abstand zwischen zwei Punkten des LineStrings ermittelt und

aufsummiert. Für diese Berechnung wird die Haversine Formel verwendet. Die Methode zum Berechnen nach der Haversine Formel ist in Scala implementiert (Grundlage der Implementierung ist die Java Methode vgl. [15]). Bei der Berechnung von Entfernungen auf der Erde besteht der Problem, das die Erde keine flache Ebene ist, sondern ein Geoid. Die Haversine Formel berücksichtig bei der Berechnung diesen Fakt in Ansätzen, jedoch unter der Annahme, das die Erde eine Kugel ist. Dementsprechend findet man in der Formel eine Konstante für den Erdradius wieder. Die Formel 5.3 demonstriert, wie der Abstand zwischen zwei Punkten x und y in Kilometern berechnet werden kann.

$$a = (sin(\Delta lat/2))^2 + cos(lat_1) * cos(lat_2) * (sin(\Delta lng/2))^2 \tag{5.1}$$

$$c = 2 * atan2(sqrt(a), sqrt(1-a)) \tag{5.2}$$

$$distance = 6367[km] * c \tag{5.3}$$

Dieses Verfahren ist nötig, da die zwei Punkte mit der Route im seltensten Fall eine Gerade bilden (siehe Abbildung 5.1). Somit muss für die Berechnung der Entfernung zwischen zwei Punkten immer die Route als Hilfsmittel verwendet werden. Als nächstes wird der Abstand zwischen den Fahrzeugen in Metern bestimmt. Für diese Berechnung wird ein Fahrzeug ausgewählt (siehe A.12), woraufhin die Entfernungen zu den anderen Fahrzeugen ermittelt wird.

Der letzte Schritt ist die Berechnung des zeitlichen Abstandes. Hierfür sind die Messfahrten notwendig. Bei einer Messfahrt wird im Vorfeld die komplette Route abgefahren und zu regelmäßigen Zeitpunkten die GPS Position gemessen. Diese Position wird zusammen mit der Uhrzeit in der Datenbank persistiert. Somit ist es möglich zu bestimmen, wie lange ein Fahrzeug von Position A zu Position B braucht. Im besten Fall existieren mehrere Messfahrten zu einer Route, damit die Aussage genauer wird. Um nun die zeitliche Entfernung von Fahrzeug A zu Fahrzeug B zu ermitteln, wird der dichteste Punkt von einer Messfahrt zum Fahrzeug A ermittelt. Das gleiche Prinzip wird auf das Fahrzeug B angewandt. Anschließend wird die Zeitdifferenz gebildet. Dieses Verfahren wird iterativ für jede Messfahrt durchgeführt. Abschließend wird der Mittelwert aller Zeitdifferenzen ermittelt.

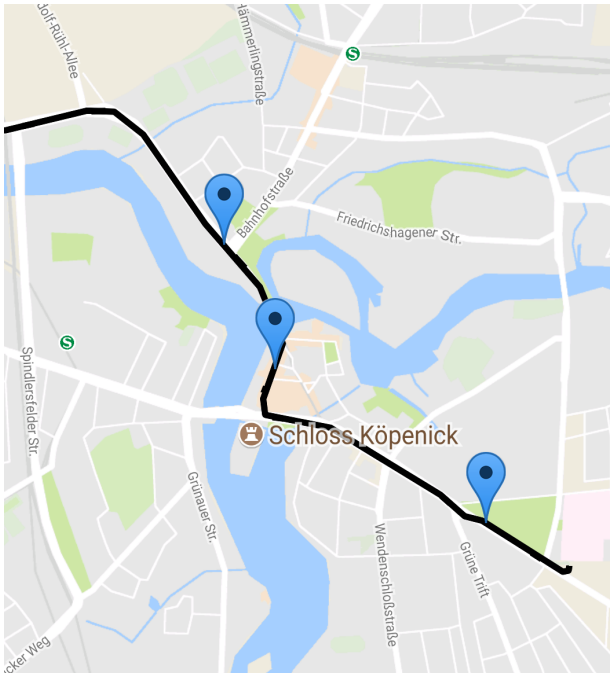

Abbildung 5.1: Beispiel für Route mit drei Fahrzeugen

5.3 Deployment / Runtime

Bei dem entwickelten System, handelt es sich um ein verteiltes System, mit unterschiedlichen Zielplattformen. Für den Client ist das gewählte System Android. Für das Deployment werden Standardtools aus der Androidentwicklung gewählt. Für das erstellen einer APK wird gradle benötigt. Um die fertige APK auf einem Android Mobiltelefon zu installieren können beispielsweise die ADB tools verwendet werden. Als Entwicklungsumgebung wird Android Studio von der Firma Google verwendet.

Um die Android App auf einem Smartphone zu verwenden, muss dieses einige Voraussetzungen erfüllen. Die App setzt eine konstante Internetverbindung für die Kommunikation mit der Server voraus. Des Weiteren benötigt die Anwendung Zugriff auf das GPS Modul. Als Android Version wird das API Level 22 vorausgesetzt (getestet mit API Version 22 und höher). Im aktuellen Entwicklungstand wird für die Kartenansicht ein eigener Google Maps API Key vorausgesetzt.[1]

Die Serverkomponente des Systems ist ein maven Projekt, da auf Java zurück gegriffen wird. In maven besteht die Möglichkeit Scala zu integrieren, so auch in diesem Projekt. Um ein Executable Jar Archive zu erstellen, wird das JDK 1.8 oder neuer (getestet mit JDK 1.8.0_144) und maven (empfohlen Version 3.5.2) benötigt. Für die Laufzeit wird lediglich das Java JRE 8 vorausgesetzt.

Der Server benötigt für die Persistierung der Daten eine MySQL Datenbank. Zur Einrichtung der Datenbank gibt es eine Konfigurationsdatei. Das Listing 5.1 zeigt eine Beispiel Konfigurationsdatei. Als Betriebsystem für die Ausführung gibt es keine Einschränkungen (getestet Linux Debian 8). Der Server benötigt Zugriff auf TCP Port 4545, um Anfragen zu bearbeiten. Während der Projektlaufzeit läuft eine Serverinstanz auf einem Server der HTW-Berlin mit der Adresse `http://bus.f4.htw-berlin.de:4545`.

Listing 5.1: Server Konfigurationsdatei

```
1 db_host=localhost
2 db_database=bvg
3 db_username=root
4 db_password=password
5 db_port=3306
```

[1] vgl.: BusbunchingFahrer/Busbunching/app/src/debug/res/values/google_maps_api.xml

6 Vorschläge / Ausblick

6.1 Vorschläge

Das Endergebnis dieser Arbeit ist ein funktionsfähiger Prototype, der zeitliche Abstände von Fahrzeuge in Minuten ermitteln kann und diese auf einer Übersichtsgrafik oder Google Maps darstellt. Jedoch läuft sie nicht Fehlerlos. Beim Testen kam es immer wieder vor das, so vermuten wir, einzelne Fahrzeuge von der Route abkamen und so für kurze Zeit die Abschätzung des Abstandes falsch angezeigt wurde. Verbessern könnte man die Darstellung des Vorgängers und Nachfolgers, die aus Übersichtsgründen farblich voneinander getrennt sind. Beispielsweise wäre eine farbliche Veränderung der Bus Icons, sobald ein Fahrzeug entweder zu nah ranfährt oder sich zu weit entfernt. Parallel dazu wäre eine Warnung oder ein kleiner Dialog sehr hilfreich, um mit dem Fahrer kommunizieren zu können und wenn Nötig ihn anzuweisen zu warten. Und falls es überhaupt keinen Vorgänger oder Nachfolger geben sollte, sich die Anzeige passend verändert indem die entsprechenden Bus Icons ausgeblendet werden. Des Weiteren wäre eine Übersicht mit Informationen, die beispielsweise angeben kann, wann das Fahrzeug seine nächste Haltestelle bzw. das Ende der Strecke erreicht, von Vorteil. Nicht berücksichtigt wurde auch ein unvorhersehbare Änderung des Fahrtenverlaufes durch einen Unfall oder ähnliches. Die App könnte eine Alternative Route vorschlagen und entsprechend die Zeitlichen Abstände anpassen.

6.2 Ausblick

Diese Arbeit beschränkt sich darauf, Abstände von Fahrzeugen zu messen, die in einem bestimmten Intervall auf einer vordefinierten Strecke fahren. Dieser Abstand wurde mithilfe von Referenzdaten ermittelt, die vorab gemessen wurden. Leider war die Verwendung der, von der BVG schon ermittelten Daten, nicht möglich. Es wurde zwar eine Kooperation angeboten, jedoch gab es kleinere Komplikationen mit der Einbindung der Datenbank, die erstmals nur einen ?Dump? von ca. 50 Datensätze umfasste die aus unsortierten und zufällig ausgewählten Daten bestand. Ein zeitnaher Zugang zur kompletten Datenbank war leider nicht zeitlich abzusehen und die eher unglücklich formulierte Datenbankdokumentation machte eine Verwendung sehr schwierig. So wurde entschieden, dass die selbst Daten zu generiert werden. Dabei wird die Anzeige der Abstände umso genauer, je mehr Referenzdaten gesammelt werden. Bei einer bestimmten Menge von Daten könnten dann individuelle Berechnung zu verschiedenen Uhrzeiten ermittelt werden. Durch die hohe Genauigkeit, die sich immer wieder steigert, wird die Schaltzentrale entlastet und Koordination des Personals und Einsatz der Fahrzeuge deutlich verbessert und effektiver. Darüber hinaus könnten die Anzeigen an Haltestellen, die dem Fahrgast die Wartezeit anzeigt, deutlich genauer und weniger Fehleranfällig werden. Ein anderer Ansatz, der als Anregung dieser Arbeit diente, war von Matthias Andres der in seiner Masterarbeit Algorithmen beschreibt, mit denen man den Fahrtverlauf vorhersagen kann. Somit könnte man nur anhand der Geodaten der Fahrzeuge prognostizieren, wie lange sie bis zu ihrer nächsten Haltestelle benötigen oder wann das Ende ihrer Fahrt erreicht ist. Kombiniert man beide Verfahren, ist man in der Lage Fahrzeuge via Algorithmus autonom zu steuern. Dies wäre gerade im Öffentlichen Personennahverkehr sehr hilfreich, da Busse so koordiniert werden könnten das gewährleistet wird, dass in einem bestimmten Zeitintervall eine vorgegebene Anzahl an Bussen eine Haltestelle

passieren. Zusammengefasst lässt sich sagen das eine gute Grundlage geschaffen wurde, bus-bunching früh zu erkennen und präventive Maßnahmen einzuleiten.

Anhang

A Rest API Schnittstellendefinition

Der Server für die Datenverwaltung basiert auf einer Rest API. Im folgenden Abschnitt werden alle möglichen Requests zu Ressourcen allgemein und mithilfe eines Beispiels echter Daten beschrieben. Während der Projektarbeit besteht die Möglichkeit, die API über den Server `http://bus.f4.htw-berlin.de:4545` aus dem HTW-Berlin Netz zu nutzen.

A.1 Request für Routen einer Linie

Anfrage aller Routen einer Linie (Beispiel Buslinie). Eine Linie hat mehrere Routen.

HTTP-Method

GET

Resource

/api/v1/route/<ref>

Parameters

- ref: Liniennummer

Response

```
[ {
  "id" : <id>,
  "osmId" : "relation/<osmId>",
  "ref" : "<line number>",
  "name" : "<description of the route>",
  "type" : "<transport type>",
  "network" : "<full name of network>",
  "operator" : "<full name of operator>",
  "from" : "<name of start station>",
  "to" : "<name of end station>",
  "routeType" : "<MULTILINE | LINE>"
} ]
```

HTTP status codes

- 200: Request erfolgreich
- 404: Keine Routen gefunden

Beispiel Request

GET `http://domain.com/api/v1/route/M11`

Beispiel Response

```
[ {
  "id" : 217,
  "osmId" : "relation/2088816",
  "ref" : "M11",
  "name" : "Buslinie M11: S Schöneweide => U Dahlem Dorf",
  "type" : "bus",
  "network" : "Verkehrsverbund Berlin-Brandenburg",
  "operator" : "Berliner Verkehrsbetriebe",
  "from" : "S Schöneweide",
  "to" : "U Dahlem Dorf",
  "routeType" : "MULTILINE"
}, {
  "id" : 218,
  "osmId" : "relation/2088817",
  "ref" : "M11",
  "name" : "Buslinie M11: U Dahlem Dorf => S Schöneweide",
  "type" : "bus",
  "network" : "Verkehrsverbund Berlin-Brandenburg",
  "operator" : "Berliner Verkehrsbetriebe",
  "from" : "U Dahlem Dorf",
  "to" : "S Schöneweide",
  "routeType" : "MULTILINE"
} ]
```

A.2 Request für GeoJson einer Route mit ID

Anfrage einer Route per Id. Das Format der Response ist GeoJson Format.

HTTP-Method

GET

Resource

/api/v1/route/geo/<id>

Parameters

- id: Route ID

Response

```
{
  "type": "Feature",
  "properties": {
    "ref": "<id>",
    "name": "<description of the route>",
    "@id": "relation/<osmId>"
    "from": "<name of start station>",
```

```
    "to": "<name␣of␣end␣station>",
    "type": "<transport␣type>",
    "operator": "<full␣name␣of␣operator>",
    "network": "<full␣name␣of␣network>"
  },
  "geometry": {
    "type": "<MultiLineString␣|␣LineString>",
    "coordinates":[]
  }
}
```

HTTP status codes

- 200: Request erfolgreich
- 400: Ungültiger Parameter
- 404: Keine Route gefunden

Beispiel Request

GET `http://domain.com/api/v1/route/geo/67`

Beispiel Response

```
{
  "type": "Feature",
  "properties": {
    "ref": "67",
    "name": "Straßenbahnlinie␣67:␣Krankenhaus␣Köpenick␣=>␣S␣
       Schöneweide",
    "from": "Krankenhaus␣Köpenick",
    "@id": "relation/2084473",
    "to": "S␣Schöneweide",
    "type": "tram",
    "operator": "Berliner␣Verkehrsbetriebe",
    "network": "Verkehrsverbund␣Berlin-Brandenburg"
  },
  "geometry": {
    "type": "MultiLineString",
    "coordinates": [
      [
        [
          13.5939995,
          52.4385062
        ],
        [
          13.5939794,
          52.438533
        ], ....
    ]
  }
```

```
}
```

A.3 Hinzufügen einer Route

Fügt eine Route (oder mehrere) hinzu. Der Payload muss im GeoJson Format sein.

HTTP-Method

POST

Resource

/api/v1/route

Payload

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {
        "@id": "relation/<osmId>",
        "name": "<description of the route>",
        "network": "<full name of network>",
        "operator": "<full name of operator>",
        "from": "<name of start station>",
        "to": "<name of end station>",
        "ref": "<id>",
        "route": "<transport type>",
        "type": "route",
      },
      "geometry": {
        "type": "<MultiLineString | LineString>",
        "coordinates": [
        ]
     }
  }
}
```

HTTP status codes

- 201: Resource erstellt
- 500: Bad payload

Beispiel Request

POST `http://domain.com/api/v1/route`

```
{
  "type": "Feature",
```

```
  "properties": {
    "ref": "67",
    "name": "Straßenbahnlinie 67: Krankenhaus Köpenick => S
       Schöneweide",
    "from": "Krankenhaus Köpenick",
    "@id": "relation/2084473",
    "to": "S Schöneweide",
    "type": "tram",
    "operator": "Berliner Verkehrsbetriebe",
    "network": "Verkehrsverbund Berlin-Brandenburg"
  },
  "geometry": {
    "type": "MultiLineString",
    "coordinates": [
       [
         [
         13.5939995,
         52.4385062
       ],
       [
         13.5939794,
         52.438533
      ], ....
    ]
  }
}
```

A.4 Request für eine Journey

Anfrage für Messwerte einer Fahrt (Journey) auf einer Route. Die Antwort enthält die bereits geglätteten Koordinaten.

HTTP-Method

GET

Resource

/api/v1/journey/<id>

Parameters

- id: Journey ID

Response

```
{
  "id": <journeyId>,
  "routeId": <routeId>,
  "startTime": <timeStamp>,
  "endTime": <timeStamp>,
  "points": [
```

```
    {
      "id": <id>,
      "journeyId": <journeyId>,
      "time": <timeStamp>,
      "lngLat": {
        "lng": <longitude>,
        "lat": <latidute>
      }
    }, ...
  ]
}
```

HTTP status codes

- 200: Anfrage erfolgreich
- 400: Ungültiger Parameter
- 404: Keine Journey gefunden

Beispiel Request

GET `http://domain.com/api/v1/journey/2`

Beispiel Response

```
{
  "id": 2,
  "routeId": 67,
  "startTime": 1513596120000,
  "endTime": 1513597500000,
  "points": [
    {
      "id": 93,
      "journeyId": 2,
      "time": 1513596185100,
      "lngLat": {
        "lng": 13.5916396,
        "lat": 52.4390415
      }
    }, ...
  ]
}
```

A.5 Hinzufügen einer neuen Journey

Erstellt eine neue Journey.

HTTP-Method

POST

Resource

/api/v1/journery

Payload

```
{
  "routeId" : <routeId>,
  "startTime": <timestamp>,
  "endTime": <timestamp>
}
```

Response

```
<id>
```

HTTP status codes

- 201: Resource erstellt
- 400: Bad payload

Beispiel Request

POST `http://domain.com/api/v1/journey`

```
{
  "routeId" : 67,
  "startTime": 1513596120000,
  "endTime": 1513597500000
}
```

Beispiel Response

```
7
```

A.6 Hinzufügen einer Messposition für die Zeitmessung

Fügt einen neuen Messpunkt für eine Route einer Journey hinzu. Eine Journey beschreibt eine konkrete Fahrt auf einer konkreten Route.

HTTP-Method

POST

Resource

/api/v1/journey/position

Payload

```
{
  "journeyId" : <journeyId>,
  "time": <timestamp>,
  "lngLat" : {
    "lng": <longitude>,
    "lat": <latitude>
  }
}
```

Response

```
<id>
```

HTTP status codes

- 201: Resource erstellt
- 400: Bad payload

Beispiel Request

POST `http://domain.com/api/v1/journey/position`

```
{
  "time":1467888902,
  "journeyId": 3,
  "lngLat": {
    "lat":13.43546,
    "lng":52.32332
  }
}
```

Beispiel Response

```
251
```

A.7 Request für ein Fahrzeug

Anfrage für ein Fahrzeug. Die Antwort enthält Geo Daten und Routeninformationen.

HTTP-Method

GET

Resource

/api/v1/vehicle/<id>

Parameters

- id: Unique Id des Fahrzeuges (Devices)

Response

```
{
  "id": <id>,
  "ref": "<uniqueId>",
  "routeId": <routeId>,
  "time": <timeStamp>,
  "position": {
    "lng": <longitude>,
    "lat": <latitude>
  },
  "pastedDistance" : <meter>
}
```

HTTP status codes

- 200: Anfrage erfolgreich
- 404: Kein Fahrzeug gefunden

Beispiel Request

GET `http://domain.com/api/v1/journey/2`

Beispiel Response

```
{
  "id": 1,
  "ref": "636c81cc2361acd7",
  "routeId": 67,
  "time": 1513596185100,
  "position": {
    "lng": 52.3453,
    "lat": 13.53234
  },
  "pastedDistance" : 5592.54969445254
}
```

A.8 Hinzufügen eines Fahrzeuges

Fügt ein neues Fahrzeug der Datenbank hinzu.

HTTP-Method

POST

Resource

/api/v1/vehicle

Payload

```
{
  "ref": "<uniqueId>",
  "routeId": <routeId>,
  "time": <timeStamp>,
  "position": {
    "lng": <longitude>,
    "lat": <latitude>
  }
}
```

Response

```
<id>
```

HTTP status codes

- 201: Resource erstellt
- 400: Bad payload

Beispiel Request

POST `http://domain.com/api/v1/vehicle`

```
{
  "ref": "636c81cc2361acd7",
  "routeId": 67,
  "time": 1513596185100,
  "position": {
    "lng": 52.3453,
    "lat": 13.53234
  }
}
```

Beispiel Response

```
2
```

A.9 Update eines Fahrzeuges

Aktualisiert die Daten eines Fahrzeuges in der Datenbank.

HTTP-Method

PUT

Resource

/api/v1/vehicle/<id>

Payload

```
{
  "ref": "<uniqueId>",
  "routeId": <routeId>,
  "time": <timeStamp>,
  "position": {
    "lng": <longitude>,
    "lat": <latitude>
  }
}
```

HTTP status codes

- 201: Resource erstellt
- 400: Bad payload

Beispiel Request

PUT `http://domain.com/api/v1/vehicle/636c81cc2361acd7`

```
{
  "ref": "636c81cc2361acd7",
  "routeId": 67,
  "time": 1513596185100,
  "position": {
    "lng": 52.3453,
    "lat": 13.53234
  }
}
```

A.10 Request für ein Fahrzeug

Anfrage aller Fahrzeuge, die mit einem Fahrzeug in Verbindung stehen. Das bedeutet, die Antwort enthält alle Fahrzeuge auf der Route des angefragten Fahrzeuges.

HTTP-Method

GET

Resource

/api/v1/vehicle/<id>/list

Parameters

- id: Unique Id des Fahrzeuges (Devices)

Response

```
[ {
  "ref" : "<devideId>",
  "geoLngLat" : {
    "lng" : <longitude>,
    "lat" : <latitude>
  },
  "relativeDistance" : <distanceMetersToRequestedVehicle>,
  "relativeTimeDistance": <
     distanceMilliSecondsToRequestedVehicle>
} ]
```

HTTP status codes

- 200: Anfrage erfolgreich
- 404: Kein Fahrzeug gefunden

Beispiel Request

GET `http://domain.com/api/v1/vehicle/636c81cc2361acd7/list`

Beispiel Response

```
[ {
  "ref" : "636c81cc2361acd7",
  "geoLngLat" : {
    "lng" : 13.5395005,
    "lat" : 52.4575977
  },
  "relativeDistance" : 0.0
}, {
  "ref" : "4dcghc4zzc6cghcf",
  "geoLngLat" : {
    "lng" : 13.5716218,
    "lat" : 52.4511964
  },
  "relativeDistance" : 2504.827656693912,
  "relativeTimeDistance": 411008
} ]
```

A.11 Request für die historischen Daten aller Fahrzeuge

Anfrage aller historischen Fahrzeugdaten.

HTTP-Method

GET

Resource

/api/v1/vehicles

Response

```
[ {
  "ref": "<uniqueId>",
  "routeId": <routeId>,
  "time": <timeStamp>,
  "position": {
    "lng": <longitude>,
    "lat": <latitude>
  }
} ]
```

HTTP status codes

- 200: Anfrage erfolgreich
- 404: Kein Fahrzeug gefunden

Beispiel Request

GET `http://domain.com/api/v1/vehicles`

Beispiel Response

```
[ {
  "ref": "636c81cc2361acd7",
  "routeId": 67,
  "time": 1513596185100,
  "position": {
    "lng": 52.3453,
    "lat": 13.53234
  }
} ]
```

A.12 Request für die historischen Daten eines Fahrzeugs

Anfrage aller historischen Fahrzeugdaten.

HTTP-Method

GET

Resource

/api/v1/vehicles/<id>

Parameters

- id: Unique Id des Fahrzeuges (Devices)

Response

```
[ {
  "ref": "<uniqueId>",
  "routeId": <routeId>,
  "time": <timeStamp>,
  "position": {
    "lng": <longitude>,
    "lat": <latitude>
  }
} ]
```

HTTP status codes

- 200: Anfrage erfolgreich
- 404: Kein Fahrzeug gefunden

Beispiel Request

GET `http://domain.com/api/v1/vehicles/636c81cc2361acd7`

Beispiel Response

```
[ {
  "ref": "636c81cc2361acd7",
  "routeId": 67,
  "time": 1513596185100,
  "position": {
    "lng": 52.3453,
    "lat": 13.53234
  }
} ]
```

B Quellenverzeichnis

[1] SPO: *Schnittstellenbeschreibung Prozessdatenaustausch*. 2004

[2] CONVERTERS, Intelligent: *Ora Dump to MySQL*. https://www.convert-in.com/ord2sql.htm

[3] *Open Street Map*. https://www.openstreetmap.de

[4] *Was ist OpenStreetMap?* https://www.openstreetmap.de/faq.html#was_ist_osm

[5] *API v0.6*. http://wiki.openstreetmap.org/wiki/API_v0.6

[6] *OpenStreetMap Wiki*. http://wiki.openstreetmap.org/wiki/Main_Page

[7] *Map Features*. http://wiki.openstreetmap.org/wiki/Map_Features

[8] *Route types (route)*. http://wiki.openstreetmap.org/wiki/Relation:route#Route_types_.28route.29

[9] *overpass turbo*. https://overpass-turbo.eu/

[10] *Overpass API*. http://wiki.openstreetmap.org/wiki/Overpass_API

[11] BUTLER, H.: *The GeoJSON Format*. https://tools.ietf.org/html/rfc7946

[12] HELMBOLD, Christian: *Kotlin - Das bessere Java*. http://helmbold.de/artikel/kotlin/

[13] ODERSKY, Martin ; LEX SPOON, Bill V.: *Programming in Scala*. Third edition. Artima Press, 2016

[14] WENDEL, Per: *Spark*. http://sparkjava.com

[15] GRAVELLE, Rob: *Calculate the Distance between Two Points in your Web Apps*. https://www.htmlgoodies.com/beyond/javascript/calculate-the-distance-between-two-points-in-your-web-apps.html