



Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

HOCHSCHULE FÜR TECHNIK UND WIRTSCHAFT

PROJEKTARRBEIT

Visualisierung und Auswertung von Positionsdaten der Omnibusse der BVG

Technik mobiler Systeme
Ausgewählte Kapitel mobiler Anwendungen

Pascal Dettmers (551733)

Stefan Neuberger (553849)

Tobias Ullerich (553746)

27. Dezember 2017

Inhaltsverzeichnis

1	Dokumentengeschichte	1
2	Problemstellung	2
3	Aufgabenstellung	3
4	Architektur	4
4.1	Überblick	4
4.1.1	Analyse Datenbank BVG	4
4.1.2	OSM	5
4.2	Komponenten	5
4.3	Rest API Schnittstellendefinition	5
4.3.1	Request für Routen einer Linie	5
4.3.2	Request für GeoJson einer Route mit ID	7
4.3.3	Hinzufügen einer Route	8
4.3.4	Request für eine Journey	10
4.3.5	Hinzufügen einer neuen Journey	11
4.3.6	Hinzufügen einer Messposition für die Zeitmessung	12
4.3.7	Request für ein Fahrzeug	13
4.3.8	Hinzufügen eines Fahrzeuges	14
4.3.9	Update eines Fahrzeuges	15
4.4	Datenbankschema Server	16
5	Nutzung	18
5.1	Code	18
5.1.1	Programmiersprache	18
5.1.2	Bibliotheken	18
5.2	Funktionsweise	18
5.3	Deployment / Runtime	18
6	Vorschläge / Ausblick	19
7	Literaturverzeichnis	20

Abbildungsverzeichnis

4.1 Datenbank Schema aus SC05 und SC51	5
--	---

Tabellenverzeichnis

1.1 Dokumentengeschichte	1
------------------------------------	---

Abkürzungsverzeichnis

RBL rechnerbasiertes Leitsystem

OSM Open Street Map

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

Tabelle 1.1: Dokumentengeschichte

2 Problemstellung

Problemstellung Busbunching, BVG, Ist-Zustand, Ziel

3 Aufgabenstellung

Das zu entwickelnde System ist eine Komponente, die einem Busfahrer Informationen über die Busse einer Buslinie zur Verfügung stellen soll. Dabei soll das System nicht nur auf Busse beschränkt sein. Denkbar sind hier auch andere Arten von öffentlichen Transportmitteln. Dabei wird die Komponente den zeitlichen Abstand zum vorherigen und nachfolgenden Bus einer Linie visualisieren. Die Visualisierung geschieht durch eine Android Applikation.

4 Architektur

4.1 Überblick

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 anderen Positionen von Bussen und deren Verspätung. (vgl.: „Die Prozessdatenschnittstelle (SC05) spiegelt die aktuelle Situation im RBL wider.“) Zum Anderen gibt es die SC51 Datenbank, entwickelt von der Firma Alcatel. Diese Schnittstelle enthält unterschiedlichste Daten für 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 relationaler Datenbanken waren jeweils deren Dokumentationen und ein Dump zur Verfügung.

Der erste Schritt der Analyse bestand darin, die Dumps der Oracle Datenbank 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“. Mit diesem Tool ist es möglich ein Oracle Datenbank Dump in eine MySQL Datenbank zu importieren. Vorteil dieser Methode ist, dass auf bestehende Kenntnisse mit dem Umgang von MySQL zurückgegriffen werden kann. Im folgenden wurde mittels der Schnittstellen Dokumentation 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, sowie eine Einordnung in die Route. Um ein 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 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 `PointsOnRoute` einem Fahrzeug zugeordnet werden.

Die Datenbank SC51 beinhaltet Tabellen für die Linien (Lines). Eine Linie ist im Kontext der BVG zum Beispiel die Buslinie X11. Jede Linie besteht aus mehreren 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: Falkensee, Bahnhof->S+U Rathaus Spandau). Die Tabelle `POINTS_ON_ROUTE` ordnet 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 eigentliche geographischen Punkte in der Form Latitude und Longitude.

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

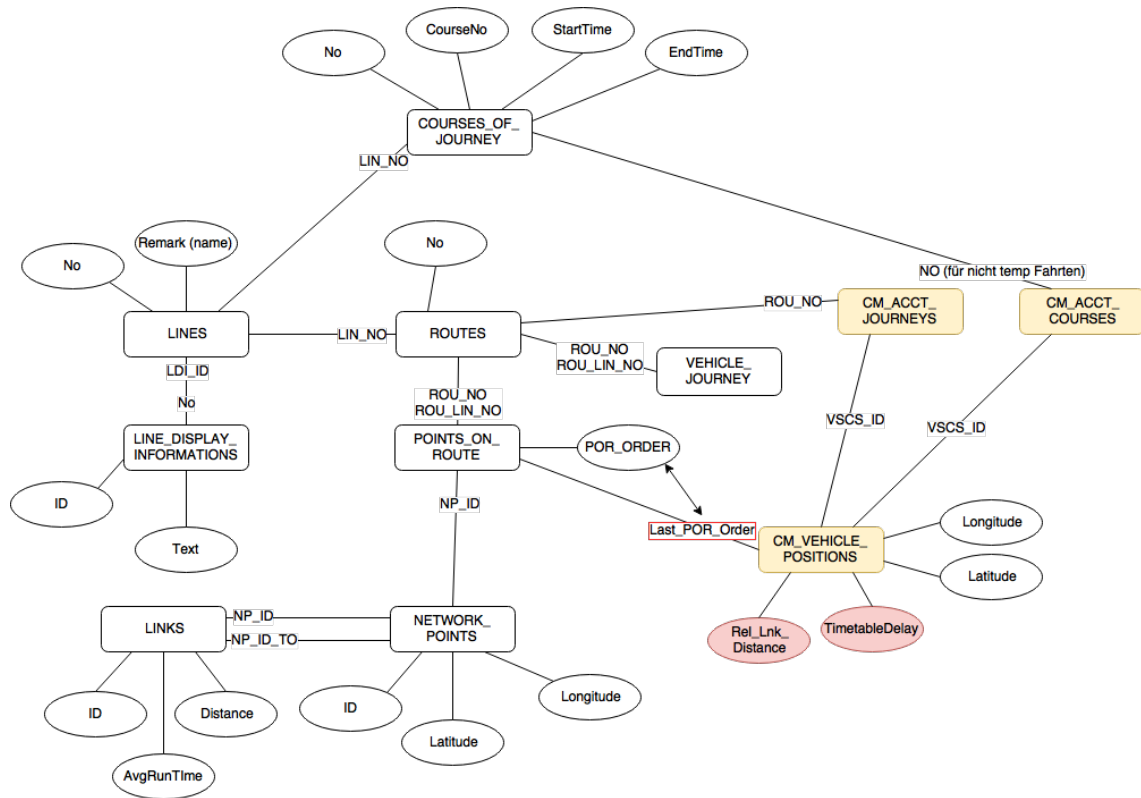


Abbildung 4.1: Datenbank Schema aus SC05 und SC51

4.1.2 OSM

Wie kommt man an die Routen Infos zu den Bus-/Tram Linien. GeoJson.

4.2 Komponenten

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

4.3 Rest API Schnittstellendefinition

Der Server für die Datenverwaltung basiert auf einer Rest API. Im folgenden Abschnitt werden alle möglichen Request 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.

4.3.1 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

```
1 [ {  
2   "id" : <id>,  
3   "osmId" : "relation/<osmId>",  
4   "ref" : "<line_number>",  
5   "name" : "<description_of_the_route>",  
6   "type" : "<transport_type>",  
7   "network" : "<full_name_of_network>",  
8   "operator" : "<full_name_of_operator>",  
9   "from" : "<name_of_start_station>",  
10  "to" : "<name_of_end_station>",  
11  "routeType" : "<MULTILINE_|_LINE>"  
12 } ]
```

HTTP status codes

- 200: Request erfolgreich
- 404: Keine Routen gefunden

Beispiel Request

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

Beispiel Response

```
1 [ {  
2   "id" : 217,  
3   "osmId" : "relation/2088816",  
4   "ref" : "M11",  
5   "name" : "Buslinie_M11:_S_Schöneweide_=>_U_Dahlem_Dorf",  
6   "type" : "bus",  
7   "network" : "Verkehrsverbund_Berlin-Brandenburg",  
8   "operator" : "Berliner_Verkehrsbetriebe",  
9   "from" : "S_Schöneweide",  
10  "to" : "U_Dahlem_Dorf",  
11  "routeType" : "MULTILINE"  
12 }, {
```

```

13   "id" : 218,
14   "osmId" : "relation/2088817",
15   "ref" : "M11",
16   "name" : "Buslinie_M11:_U_Dahlem_Dorf_=>_S_Schöneweide",
17   "type" : "bus",
18   "network" : "Verkehrsverbund_Berlin-Brandenburg",
19   "operator" : "Berliner_Verkehrsbetriebe",
20   "from" : "U_Dahlem_Dorf",
21   "to" : "S_Schöneweide",
22   "routeType" : "MULTILINE"
23 } ]

```

4.3.2 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

```

1  {
2    "type": "Feature",
3    "properties": {
4      "ref": "<id>",
5      "name": "<description_of_the_route>",
6      "@id": "relation/<osmId>",
7      "from": "<name_of_start_station>",
8      "to": "<name_of_end_station>",
9      "type": "<transport_type>",
10     "operator": "<full_name_of_operator>",
11     "network": "<full_name_of_network>"
12   },
13   "geometry": {
14     "type": "<MultiLineString_|_LineString>",
15     "coordinates": []
16   }
17 }

```

HTTP status codes

- 200: Request erfolgreich
- 404: Keine Route gefunden

Beispiel Request

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

Beispiel Response

```
1 {
2   "type": "Feature",
3   "properties": {
4     "ref": "67",
5     "name": "Straßenbahnlinie_67:_Krankenhaus_Köpenick_=>_S_
        Schöneeweide",
6     "from": "Krankenhaus_Köpenick",
7     "@id": "relation/2084473",
8     "to": "S_Schöneeweide",
9     "type": "tram",
10    "operator": "Berliner_Verkehrsbetriebe",
11    "network": "Verkehrsverbund_Berlin-Brandenburg"
12  },
13  "geometry": {
14    "type": "MultiLineString",
15    "coordinates": [
16      [
17        [
18          13.5939995,
19          52.4385062
20        ],
21        [
22          13.5939794,
23          52.438533
24        ], ....
25      ]
26    ]
27  }
```

4.3.3 Hinzufügen einer Route

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

HTTP-Method

POST

Resource

/api/v1/route

Payload

```
1 {
2   "type": "FeatureCollection",
3   "features": [
4     {
5       "type": "Feature",
6       "properties": {
7         "@id": "relation/<osmId>",
8         "name": "<description_of_the_route>",
9         "network": "<full_name_of_network>",
10        "operator": "<full_name_of_operator>",
11        "from": "<name_of_start_station>",
12        "to": "<name_of_end_station>",
13        "ref": "<id>",
14        "route": "<transport_type>",
15        "type": "route",
16      },
17      "geometry": {
18        "type": "<MultiLineString_|_LineString>",
19        "coordinates": [
20          ]
21      }
22    }
23  }
```

HTTP status codes

- 201: Resource erstellt
- 500: Bad payload

Beispiel Request

POST <http://domain.com/api/v1/route>

```
1 {
2   "type": "Feature",
3   "properties": {
4     "ref": "67",
5     "name": "Straßenbahnlinie_67:_Krankenhaus_Köpenick_=>_S_
        Schöneweide",
6     "from": "Krankenhaus_Köpenick",
7     "@id": "relation/2084473",
8     "to": "S_Schöneweide",
9     "type": "tram",
10    "operator": "Berliner_Verkehrsbetriebe",
11    "network": "Verkehrsverbund_Berlin-Brandenburg"
12  },
13  "geometry": {
14    "type": "MultiLineString",
```

```

15     "coordinates": [
16         [
17             [
18                 13.5939995,
19                 52.4385062
20             ],
21             [
22                 13.5939794,
23                 52.438533
24             ], ....
25         ]
26     }
27 }

```

4.3.4 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

```

1  {
2      "id": <journeyId>,
3      "routeId": <routeId>,
4      "startTime": <timeStamp>,
5      "endTime": <timeStamp>,
6      "points": [
7          {
8              "id": <id>,
9              "journeyId": <journeyId>,
10             "time": <timeStamp>,
11             "lngLat": {
12                 "lng": <longitude>,
13                 "lat": <latitude>
14             }
15         }, ...
16     ]
17 }

```

HTTP status codes

- 200: Anfrage erfolgreich
- 404: Keine Journey gefunden

Beispiel Request

GET <http://domain.com/api/v1/journey/2>

Beispiel Response

```
1 {
2   "id": 2,
3   "routeId": 67,
4   "startTime": 1513596120000,
5   "endTime": 1513597500000,
6   "points": [
7     {
8       "id": 93,
9       "journeyId": 2,
10      "time": 1513596185100,
11      "lngLat": {
12        "lng": 13.5916396,
13        "lat": 52.4390415
14      }
15    }, ...
16  ]
17 }
```

4.3.5 Hinzufügen einer neuen Journey

Erstellt eine neue Journey.

HTTP-Method

POST

Resource

[/api/v1/journery](#)

Payload

```
1 {
2   "routeId" : <routeId>,
3   "startTime": <timestamp>,
4   "endTime": <timestamp>
5 }
```

Response

```
1 <id>
```

HTTP status codes

- 201: Resource erstellt
- 500: Bad payload

Beispiel Request

POST <http://domain.com/api/v1/journey>

```
1 {
2   "routeId" : 67,
3   "startTime": 1513596120000,
4   "endTime": 1513597500000
5 }
```

Beispiel Response

```
1 7
```

4.3.6 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/position>

Payload

```
1 {
2   "journeyId" : <journeyId>,
3   "time": <timestamp>,
4   "lngLat" : {
5     "lng": <longitude>,
6     "lat": <latitude>
7   }
8 }
```


Response

```
1 <id>
```

HTTP status codes

- 201: Resource erstellt
- 500: Bad payload

Beispiel Request

POST <http://domain.com/api/v1/position>

```
1 {
2   "time":1467888902,
3   "journeyId": 3,
4   "lngLat": {
5     "lat":13.43546,
6     "lng":52.32332
7   }
8 }
```

Beispiel Response

```
1 251
```

4.3.7 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

```
1 {
2   "id": <id>,
3   "ref": "<uniqueId>",
4   "routeId": <routeId>,
```

```

5  "time": <timeStamp>,
6  "position": {
7      "lng": <longitude>,
8      "lat": <latitude>
9  }
10 }

```

HTTP status codes

- 200: Anfrage erfolgreich
- 404: Kein Fahrzeug gefunden

Beispiel Request

GET <http://domain.com/api/v1/journey/2>

Beispiel Response

```

1  {
2      "id": 1,
3      "ref": "636c81cc2361acd7",
4      "routeId": 67,
5      "time": 1513596185100,
6      "position": {
7          "lng": 52.3453,
8          "lat": 13.53234
9      }
10 }

```

4.3.8 Hinzufügen eines Fahrzeuges

Fügt ein neues Fahrzeug der Datenbank hinzu.

HTTP-Method

POST

Resource

/api/v1/vehicle

Payload

```

1  {
2      "ref": "<uniqueId>",
3      "routeId": <routeId>,
4      "time": <timeStamp>,
5      "position": {
6          "lng": <longitude>,

```

```
7     "lat": <latitude>
8   }
9 }
```

Response

```
1 <id>
```

HTTP status codes

- 201: Resource erstellt
- 500: Bad payload

Beispiel Request

POST <http://domain.com/api/v1/position>

```
1 {
2   "ref": "636c81cc2361acd7",
3   "routeId": 67,
4   "time": 1513596185100,
5   "position": {
6     "lng": 52.3453,
7     "lat": 13.53234
8   }
9 }
```

Beispiel Response

```
1 2
```

4.3.9 Update eines Fahrzeuges

Aktualisiert die Daten ein neues Fahrzeuges int der Datenbank hinzu.

HTTP-Method

PUT

Resource

</api/v1/vehicle>

Payload

```
1 {
2   "ref": "<uniqueId>",
3   "routeId": <routeId>,
4   "time": <timeStamp>,
5   "position": {
6     "lng": <longitude>,
7     "lat": <latitude>
8   }
9 }
```

HTTP status codes

- 201: Resource erstellt
- 500: Bad payload

Beispiel Request

POST <http://domain.com/api/v1/position>

```
1 {
2   "ref": "636c81cc2361acd7",
3   "routeId": 67,
4   "time": 1513596185100,
5   "position": {
6     "lng": 52.3453,
7     "lat": 13.53234
8   }
9 }
```

4.4 Datenbankschema Server

Für die Persistierung der Daten des Servers wurde eine relationale Datenbank gewählt. Speziell 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 Geometrien aus OSM gespeichert. Dafür gibt es die Tabellen `Route`, `MultiLineString` und `LineString`. Die Relation `Route` enthält Informationen über Liniennummern (`rel`), Start- und Zielhaltestelle (`from`, `to`). Desweiteren sind Informationen über das Verkehrsunternehmen (`operator`, Beispiel: BVG) und den Verkehrsverbund (`network`, Beispiel: VBB) vorhanden. Die Eigenschaft `type` entscheidet über den Typ der Geometrie der 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: `MultiLineString((13.6920278 52.4516269, 13.6923877 52.4515733, 13.6926666 52.4515393, 13.6929464 52.451526, ...))`.

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 Tabel-

le Journey beschreibt eine spezielle Fahrt zu einer Uhrzeit auf einer Route. Die Messwerte, also die GPS Position eines Fahrzeugss 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 befinden müssen.

In der Tabelle `Vehicle` werden die aktuellen Informationen zu den Fahrzeugen gespeichert. Dafür enthält die Tabelle Informationen zu GPS Position (nicht geglättet), ein 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`. Der Primärschlüssel der Relation `VehicleHistory` ist ein Tupel aus `ref` und `time`.

5 Nutzung

5.1 Code

5.1.1 Programmiersprache

5.1.2 Bibliotheken

5.2 Funktionsweise

Wie funktioniert die Berechnung.

5.3 Deployment / Runtime

6 Vorschläge / Ausblick

7 Literaturverzeichnis