Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Optimierung und Erweiterung einer Applikation zur automatisierten Erfassung eines elektronischen Fahrtenbuches

Tolunay Yüksel

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel, Prof. Dr.-Ing. H.-C. Reuss
Betreuer/in:	Dipl.-Ing. Chris Auer
Beginn am:	1. August 2018
Beendet am:	1. Februar 2019

Kurzfassung

In einer vorangegangenen Studienarbeit wurde eine Node.js Applikation zur automatisierten Erfassung eines elektronischen Fahrtenbuches entwickelt. Diese Applikation besteht aus einer Fahrzeug- und einer Serveranwendung. Die Fahrzeuganwendung erfasst die für das Fahrtenbuch relevanten Daten, wie zum Beispiel GPS-Koordinaten und Kilometerstände. Diese Daten erhält sie aus dem Controller Area Network (CAN) Bus der in den Fahrzeugen verwendeten Messtechnik. Danach werden die Daten in einer Datenbank abgelegt und mittels mobilem Hotspot an einen Message Queuing Telemetry Transport (MQTT) Broker geschickt. Aufgabe der Serveranwendung ist es, die Daten vom MQTT Broker zu holen und in die eigene Datenbank zu speichern. Die GPS-Koordinaten werden an Nominatims Reverse Geocoding Server gesendet, um aus den Daten eine menschenlesbare Routenbeschreibung generieren zu können. Auf Wunsch stellt der Server dann einen Excel Export zur Verfügung, in der die Fahrten vermerkt sind.

Das Ziel dieser Bachelorarbeit ist es diese Applikation weiterzuentwickeln. Aus einer Analyse der bisherige Applikation ist auszugehen, dass folgende Anforderungen an die Applikation erfüllt werden müssen:

- Aufgrund von Verbindungsabbrüchen muss gewährleistet sein, dass die Daten vom Fahrzeug dennoch in absehbarer Zeit beim Server ankommen.
- Da die GPS-Daten nur zu erfassen nicht ausreichend ist, um die gefahrene Strecke zu rekonstruieren, müssen diese Daten aufbereitet werden können.
- Die Routenerkennung, die Start, Ziel und die drei meist befahrenen Straßen ermittelt, muss optimiert werden.
- Die Anwendung muss in der Lage sein, mit mehreren Fahrzeugen umgehen zu können, da sie bisher nur ein Fahrzeug unterstützt.

Um die Synchronisation der Datenbanken zu gewährleisten, wird ein Synchronisationsablauf implementiert, der immer zu Beginn einer Fahrt stattfindet. Somit können die Daten, auch bei Verbindungsabbrüchen während des Synchronisationsvorgangs, dennoch in absehbarer Zeit beim Server ankommen.

Darüberhinaus wird eine tägliche Routine eingeführt, die unbehandelte GPS-Daten aufbereitet, in dem sie nicht verwertbare Daten eliminiert und fehlerhafte Daten durch eine Interpolation korrigiert.

Die Routenerkennung wird durch eine genauere Berechnung der drei meist befahrenen Straßen optimiert.

Bei der Serveranwendung werden sowohl Datenbank als auch alle Funktionen so angepasst, sodass mehrere Fahrzeuge unterstützt werden können. Dabei ist gewährleistet, dass alle Daten in der Datenbank und alle Nachrichten an und vom Fahrzeug korrekt gekennzeichnet sind, damit es zu keiner Verwechslung kommen kann.

Inhaltsverzeichnis

1 Einleitung **17**

2 Grundlagen **19**
2.1 Hardware 19
2.1.1 Raspberry Pi 19
2.1.2 Fahreridentifizierung (NFC) 20
2.2 Node.js 21
2.2.1 Node Package Manager (NPM) 22
2.2.2 Express 22
2.2.3 AngularJS 23
2.3 Controller Area Network 23
2.3.1 CAN-Utils 24
2.4 Datenübertragung (MQTT) 25
2.5 Kartendienst und Reverse Geocoding 26

3 Analyse der bisherigen Applikation **29**
3.1 Konzept 29
3.2 Benutzeroberfläche 30
3.3 Kommunikation innerhalb der Fahrzeuganwendung 31
3.4 Erfassung der CAN-Daten 33
3.5 Synchronisation der Datenbank 33
3.6 Routenerkennung 34

4 Anforderungen an die optimierte und erweiterte Applikation **35**
4.1 Synchronisation der Datenbank auf dem Server 35
4.2 Aufbereitung der GPS-Daten 36
4.3 Routenerkennung 37
4.4 Erweiterung zur Unterstützung mehrerer Fahrzeuge 38

5 Implementierung **39**
5.1 Fahrzeug-Backend 39
5.1.1 Datenbank 42
5.1.2 Unveränderte Module 44
5.1.3 routes.js 46
5.1.4 db.js 47
5.1.5 mqtt.js 49

5.1.6 can.js . . . 50
5.2 Fahrzeug-Frontend . . . 53
5.2.1 index.html . . . 53
5.2.2 map.html . . . 54
5.2.3 factory.js . . . 54
5.2.4 app.js . . . 54
5.3 Server-Backend . . . 55
5.3.1 Datenbank . . . 58
5.3.2 server.js . . . 61
5.3.3 msdb.js . . . 61
5.3.4 nominatim.js . . . 63
5.3.5 utility.js . . . 63
5.3.6 daily.js . . . 65
5.3.7 excel.js . . . 67
5.4 Synchronisation . . . 69

6 Test der Applikation **73**

7 Fazit & Ausblick **77**

Literaturverzeichnis **79**

Abbildungsverzeichnis

2.1 Raspberry Pi 3 Model B 20
2.2 ACR122U NFC Reader [Ide] 21
2.3 Publish/Subscribe-Funktionsweise von MQTT [Obe18] 26
2.4 Antwort auf eine Reverse Geocoding Anfrage an Nominatim 27

3.1 Gesamtkonzept (in Anlehnung an [Gro18]) 30
3.2 Benutzeroberfläche nach Beendigung der Fahrt per Stop-Button . . . 31
3.3 Kommunikation zwischen Fahrzeug-Frontend und -Backend (in Anlehnung an [Gro18]) 32

4.1 Kreuzung mit einem Beispiel für eine ungenaue GPS-Position (rot) (in Anlehnung an [Gro18]) 36
4.2 Reverse Geocoding Antwort von Nominatim ohne Straße 37
4.3 Globus mit Längen- und Breitengraden 38

5.1 Aufbau der Fahrzeuganwendung [Gro18] 40
5.2 Grobe Übersicht über den Ablauf der Fahrzeuganwendung 41
5.3 Ablaufdiagramm nfc.js [Gro18] 45
5.4 Ablaufdiagramm gpio.js (in Anlehnung an [Gro18]) 46
5.5 Klassendiagramm von routes.js 47
5.6 Klassendiagramm von db.js 48
5.7 Ablaufdiagramm mqtt.js (in Anlehnung an [Gro18]) 50
5.8 Ablaufdiagramm can.js 51
5.9 Aufbau der Serveranwendung (in Anlehnung an [Gro18]) 56
5.10 Grobe Übersicht über den Ablauf der Serveranwendung 57
5.11 Klassendiagramm von msdb.js 62
5.12 Klassendiagramm von utility.js 63
5.13 Cron Format [NPM] 65
5.14 Synchronisationsvorgang zwischen Fahrzeug und Server 70

Tabellenverzeichnis

5.1 Spalteneigenschaften der Tabelle Users im Fahrzeug 42
5.2 Spalteneigenschaften der Tabelle Trips im Fahrzeug 43
5.3 Spalteneigenschaften der Tabelle GPSLookups im Fahrzeug 43
5.4 Spalteneigenschaften der Tabelle Trips beim Server 60
5.5 Spalteneigenschaften der Tabelle GPSLookups beim Server 60

Verzeichnis der Listings

2.1 Asynchrone Funktion mit Callback um Datei auszulesen 22
2.2 Aktivieren des virtuellen CAN-Busses 24
2.3 Konvertierung der Log-Dateien 24
2.4 Abspielen der Log-Datei auf dem virtuellen CAN-Bus 25
2.5 Konsolenausgabe der Daten des virtuellen CAN-Busses 25
5.1 Inhalt einer CAN Nachricht 52
5.2 Asynchrone for-Schleife 66

Akronyme

CAN Controller Area Network. 5, 19, 23

CSS Cascading Style Sheets. 53

GPIO General Purpose Input Output. 19

GPS Global Positioning System. 5, 23

HAT Hardware Attached on Top. 45

HTML Hypertext Markup Language. 22, 30

I/O Input/Output. 21

ID Identifier. 32

IoT Internet of Things. 25

IVK Institut für Verbrennungsmotoren und Kraftfahrwesen. 17

JSON JavaScript Object Notation. 46

MQTT Message Queuing Telemetry Transport. 5, 19, 25

NFC Near Field Communication. 20

NPM Node Package Manager. 22

ODbL Open Data Commons Open Database Lizenz. 26

OSM OpenStreetMap. 26

QoS Quality of Service. 25

RAM Random-access memory. 19

RFID Radio-Frequency Identification. 20

SQL Structured Query Language. 29

UID Unique Identifier. 20

URL Uniform Resource Locator. 63

USB Universal Serial Bus. 19

VIN Vehicle Identification Number. 49

1 Einleitung

Für die Elektrofahrzeuge der Fahrzeugflotte des Instituts für Verbrennungsmotoren und Kraftfahrwesen (IVK) ergibt sich aufgrund der Zulassung als Erprobungsfahrzeug die Pflicht, für jedes Fahrzeug ein Fahrtenbuch zu führen. Diese wurden bisher immer mit Stift und Papier, und anschließender Übertragung in eine Excel Tabelle manuell geführt, was jedoch sowohl aufwändig als auch fehleranfällig ist. Aus diesem Grund entstand der Wunsch eine Applikation zu entwickeln, die diese Aufgabe automatisiert.

In einer vorangegangenen Studienarbeit wurde bereits eine Node.js Applikation entwickelt, die diese Aufgabe übernehmen soll. Die Applikation wurde jedoch nicht vervollständigt und ist teilweise fehlerhaft. Das Ziel dieser Arbeit ist es diese Applikation so weit zu optimieren und zu erweitern, sodass der IVK sie während ihrer Fahrten in Betrieb nehmen kann.

Aufgrund diesen Voraussetzungen gilt es, im Rahmen dieser Bachelorarbeit zuerst die bisherige Applikation zu analysieren, um herauszufinden, auf welchem Stand sie sich befindet und welche Optimierungen bzw. Erweiterungen benötigt werden.

Auf der Basis dieser Analyse werden im Anschluss Anforderungen an die Applikation gestellt, die am Ende dieser Arbeit erfüllt sein sollen.

In Kapitel 5 werden die Änderungen, die sowohl in der Fahrzeug- als auch Serveranwendung im Laufe dieser Arbeit stattgefunden haben, detailliert beschrieben.

Am Ende dieser Bachelorarbeit wird die gesamte Applikation durch verschiedene Szenarios in Testfahrten und einen Ausblick für die Zukunft dieses elektronischen Fahrtenbuches abgeschlossen.

2 Grundlagen

Um ein Verständnis für die in dieser Arbeit behandelten Themen zu vermitteln, verschafft das Kapitel Grundlagen zuerst einen Überblick über die verwendete Hardware und dessen Benutzung. Im darauf folgenden Abschnitt werden Node.js und dessen Komponenten vorgestellt. In Abschnitt 2.3 wird das Controller Area Network (CAN) beschrieben, welches eine wichtige Rolle in der Fahrzeuganwendung spielt. Außerdem wird hierzu die Verwendung von einer Reihe von Werkzeugen, die als *CAN-Utils* zusammengefasst werden, erklärt. Der Abschnitt danach handelt vom Message Queuing Telemetry Transport (MQTT). Dieser dient der Kommunikation zwischen der Fahrzeug- und Serveranwendung. Im Anschluss wird dieses Kapitel mit der Vorstellung des verwendeten Kartendienstes abgeschlossen.

2.1 Hardware

2.1.1 Raspberry Pi

Raspberry Pi ist ein Einplatinenrechner in Größe einer Kreditkarte. Für die Entwicklung der Fahrzeuganwendung steht ein Raspberry Pi 3 Model B (Abbildung 2.1) zur Verfügung. Dieses Model besitzt einen 1024 Megabyte großen Random-access memory (RAM) und vier Kerne mit einer Taktfrequenz von 1200 MHz. Zudem sind auf der Platine vier Universal Serial Bus (USB) Ports (Abbildung 2.1 - 1) befestigt, die dadurch eine Benutzung von Peripheriegeräten wie zum Beispiel einer Maus oder einer Tastatur ermöglichen. Wie in Abbildung 2.1 - 2 zu sehen ist, besitzt der Raspberry Pi außerdem noch eine Netzwerkschnittstelle, die Datenübertragung sowohl über Bluetooth als auch WLAN anbietet. [RaspF]

In dieser Arbeit wird der Rechner über dessen Display Serial Interface (Abbildung 2.1 - 3) um ein 7 Zoll Touchscreen Display erweitert. Damit können die Fahrer durch Berührung beispielsweise ihre Fahrt manuell beenden oder anfangen, eine neue Fahrt aufzunehmen.

Da jedoch das Erfassen des elektronischen Fahrtenbuches weitestgehend automatisch ablaufen soll, wird die Inbetriebnahme des Raspberry Pi vom Fahrzeug gesteuert. Um das zu realisieren wird die sogenannte Klemme 15 des Fahrzeugs mit einem der General Purpose Input Output (GPIO) Pins, die in Abbildung 2.1 - 4 zu sehen sind, verbunden. Ein GPIO Pin ist eine programmierbare Schnittstelle, die Signale von

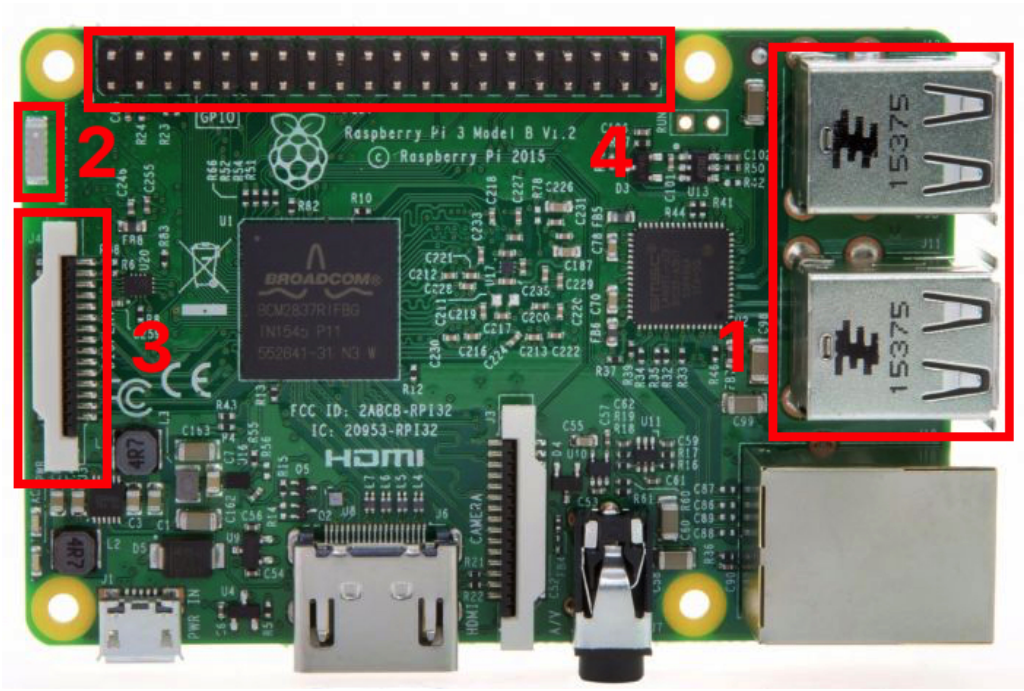

Abbildung 2.1: Raspberry Pi 3 Model B

Sensoren oder anderen elektrischen Geräten entgegennimmt. In diesem Fall nimmt der angeschlossene GPIO Pin die Signale von der Klemme 15 entgegen, wobei ein HIGH/LOW-Signal dafür steht, dass die Zündung des Fahrzeugs ein-/ausgeschaltet ist.

2.1.2 Fahreridentifizierung (NFC)

Die Fahreridentifizierung findet mittels Near Field Communication (NFC) statt, das auf der ausgereiften Radio-Frequency Identification (RFID) Technik basiert. Diese Technik ist der internationale Übertragungsstandard zum kontaktlosen Austausch von Daten per elektromagnetischer Induktion mittels loser gekoppelter Spulen über kurze Strecken von wenigen Zentimetern. NFC findet heutzutage immer mehr Verwendung, zum Beispiel beim bargeldlosen und kontaktlosen Zahlen mittels Kreditkarte.

Ein NFC System besteht aus einem aktiven Lesegerät und einem passiven Transponder, auch *NFC Tag* genannt. Jeder NFC Tag besitzt eine vom Hersteller vergebene eindeutige Kennnummer, auch *Unique Identifier (UID)* genannt. Das in dieser Arbeit verwendete Lesegerät (Abbildung 2.2) ist ein ACR122U NFC Reader vom Hersteller Advanced Card Systems Ltd. Es basiert auf der 13,56 MHz Contactless (RFID) Technologie, wodurch auch eine Kompatibilität zu älteren Smartcardsystemen gegeben

ist. Die Reichweite des Reader ist bis zu 5 cm, abhängig vom Typ des NFC-Tags. Da der verwendete Reader die plug-and-play Fähigkeit und einen USB Anschluss besitzt, lässt er sich unkompliziert mit dem Raspberry Pi verbinden. [ACS]

Abbildung 2.2: ACR122U NFC Reader [Ide]

2.2 Node.js

Node.js ist eine asynchrone, event-basierte Laufzeitumgebung zur Entwicklung von plattformunabhängigen, serverseitigen Webanwendungen in JavaScript [Nod]. Ryan Dahl hat Node.js entwickelt und erstmals im Jahr 2009 veröffentlicht [Rod12].

Node.js basiert auf Googles V8 Engine, dem derzeit schnellsten verfügbaren JavaScript-Compiler. Dadurch sind die entwickelten Webanwendungen hochperformant und ermöglichen ausgezeichnete Parallelisierung [Rod12]. Dabei arbeitet Node.js anstelle von Threads mit untergeordneten Prozessen, mit denen man leicht kommunizieren und Lastenverteilung über Kerne hinweg ermöglichen kann [Nod].

Da Websockets und Streaming fundamentale Konzepte von Node.js sind, lassen sich schon mit wenigen Zeilen JavaScript-Code echtzeitfähige Webanwendungen entwickeln, die Desktopanwendungen in nichts nachstehen. [Rod12]

Weil Node.js blockierungsfrei ist, lassen sich hiermit skalierbare Webanwendungen entwickeln. Grund hierfür ist, dass fast jede Funktion in Node.js Input/Output (I/O)-Operationen asynchron ausführt. [Nod]

Listing 2.1 Asynchrone Funktion mit Callback um Datei auszulesen

```
1 readFile('file.txt', function(err, text) {
2         if (err) throw err;
3         console.log('Inhalt von file.txt: ');
4         console.log(text);
5 });
6 console.log('Dieser Text erscheint zuerst.');
```

Bei der asynchronen Abarbeitung einer I/O-Operation, wie zum Beispiel beim Auslesen einer Datei, wird intern eine Anfrage an das Betriebssystem weitergegeben. Dabei wird zur Behandlung der Antwort eine sogenannte *Callback*-Funktion registriert. In Listing 2.1 wird veranschaulicht, wie eine Datei asynchron ausgelesen und dessen Inhalt auf der Konsole ausgegeben wird. Die asynchrone Funktion `readFile` bekommt als Parameter den Dateipfad und die Callback-Funktion übergeben. Nach Aufruf der Funktion `readFile` kümmert sich das Betriebssystem um das Auslesen der Datei. Daraufhin wird schon das nächste Kommando in Zeile 6 ausgeführt, das einen Text auf der Konsole ausgibt, noch bevor das Auslesen beendet ist. Das bedeutet, dass die Abarbeitung nicht durch die Leseoperation blockiert wird. Ist das Auslesen der Datei schließlich beendet, erhält Node.js den Inhalt und ruft die Callback-Funktion auf, dessen Verhalten bei der Übergabe als Parameter an die Funktion `readFile` definiert wurde. Hierfür muss die Abarbeitung der Applikation kurz unterbrochen werden, um die Callback-Funktion ausführen zu können. [Spr18, S. 40 f.]

2.2.1 Node Package Manager (NPM)

Der Node Package Manager (NPM) ist ein Paketverwalter für JavaScript, der in der Plattform von Node.js direkt integriert ist und somit nicht zusätzlich installiert werden muss. NPM besitzt die meisten Pakete unter den Package Managern. Mit ihm können selbsterstellte Module anderen Entwicklern zur Verfügung gestellt werden. Ebenso lassen sich damit Applikationen mit Paketen anderer Entwickler erweitern. Kernfunktionen des NPM sind somit das Suchen, Installieren, Aktualisieren und Löschen von Paketen, die sich über die Kommandozeile mit verschiedenen Optionen steuern lassen. [Spr18, S. 535]

2.2.2 Express

Express ist ein serverseitiges Webframework für Node.js, das den Funktionsumfang von Webservern erweitert und somit den Umgang damit vereinfacht. Hierbei handelt es sich um fundamentale Erweiterungen wie zum Beispiel das *Routing* (Weiterleitung), *Sessions* und *Coockies*. [Yaa13]

Bei einer Anfrage eines Clients an den Webserver entscheidet Express in Abhängigkeit von der Routing Konfiguration, welche Funktion der Anwendung ausgeführt werden soll. Dabei wird zum Beispiel mit einer Benutzeroberfläche in Form von HTML, CSS und Javascript Dateien an den Client geantwortet, die anschließend durch den Browser des Clients dargestellt wird.

2.2.3 AngularJS

AngularJS ist ein, von Google entwickeltes, clientseitiges Webframework für JavaScript, das Entwicklern ermöglicht, unkompliziert Single-Page-Webanwendungen zu erstellen. Dadurch bildet AngularJS das passende Gegenstück zu Express. [Höl17]

Dabei baut AngularJS eine Zwei-Wege-Datenbindung zwischen der Benutzeroberfläche und der Anwendung auf, die in einer Richtung dafür sorgt, dass sich die Änderungen der Daten von der Anwendung in der Ansicht automatisch auswirken. In die andere Richtung wird ebenfalls gewährleistet, dass sich Benutzerinteraktionen innerhalb der Ansicht automatisch in den Daten widerspiegeln. Somit entsteht durch die Zwei-Wege-Datenbindung eine Brücke zwischen der Anwendung und den Elementen der Benutzeroberfläche. [TB14]

2.3 Controller Area Network

Das Controller Area Network (CAN) ist derzeit das in der Automobilindustrie am häufigsten eingesetzte Bussystem. Es wurde 1985 von Bosch entwickelt und dient zur Kommunikation zwischen einzelnen Fahrzeugkomponenten. [ZS14]

Der CAN-Bus hat sich als Standard etabliert und wird heutzutage in nahezu jedem Fahrzeug mit unterschiedlicher Anzahl an Busteilnehmern verwendet. Dabei hat diese Technologie zwei große Vorteile. Zum einen entfällt eine Vielzahl an schwer diagnostizierbaren Kabeln und Verbindungsstellen, da die Busteilnehmer nicht direkt miteinander Verbunden sind, zum anderen wird dadurch die Ausfallrate deutlich gesenkt. [GG06]

Durch die in den Elektrofahrzeugen des IVK vorhandene Messtechnik lassen sich, mithilfe des Zugangs vom Raspberry Pi zum CAN-Bus, Fahrzeugdaten wie z.B. Global Positioning System (GPS)-Koordinaten und Kilometerstände auslesen.

2.3.1 CAN-Utils

Um während der Entwicklung der Applikation die Software testen zu können, ist es ausgesprochen wichtig, nicht immer auf ein reales Fahrzeug angewiesen zu sein. Deshalb wurden vom IVK im Vorfeld mehrere CAN-Log-Dateien von unterschiedlichen Fahrten aufgenommen und zur Verfügung gestellt.

CAN-Utils sind Werkzeuge für *SocketCAN*. SocketCAN ist eine Sammlung an CAN-Treibern für das Betriebssystem Linux. Die Treiber wurden von der Konzernforschung der Volkswagen AG als Open Source bereitgestellt und sind bereits in den Linux Kernel integriert, wodurch eine zusätzliche Installation nicht vonnöten ist.

Mit Hilfe dieser Werkzeuge lässt sich die Entwicklung der Fahrtenbuch-Applikation deutlich einfacher gestalten. Dabei wird durch die CAN-Log-Dateien ein CAN-Bus während einer Fahrt simuliert.

Starten des virtuellen CAN-Busses

Bevor mit den CAN-Utils gearbeitet werden kann, muss der virtuelle CAN-Bus initialisiert werden. Dies geschieht durch folgende Befehle in der Kommandozeile, die aus Listing 2.2 zu entnehmen sind:

Listing 2.2 Aktivieren des virtuellen CAN-Busses

```
1 $ sudo modprobe vcan
2 $ sudo ip link add dev vcan0 type vcan
3 $ sudo ip link set up vcan0
```

asc2log

Da sich die zur Verfügung stehenden CAN-Log-Dateien zunächst im ASCII-Dateiformat *.asc* befinden, müssen sie mittels `asc2log` (Listing 2.3) in das für den `canplayer` passenden Format konvertiert werden.

Listing 2.3 Konvertierung der Log-Dateien

```
1 $ ./asc2log -I 20170405_S-Zuffenhausen1.asc -O 20170405_S-Zuffenhausen1.log
```

Dabei wird hier durch `-I` die Input Datei und durch `-O` die Output Datei spezifiziert.

canplayer

Der `canplayer` dient dazu, die CAN-Log-Dateien, die zuvor durch `asc2log` konvertiert wurde, auf dem virtuellen CAN-Bus abzuspielen. In Listing 2.4 ist die Option `-l i` zu sehen, die den `canplayer` auffordert, die Log-Datei unendlich oft abspielen zu lassen.

Listing 2.4 Abspielen der Log-Datei auf dem virtuellen CAN-Bus

```
1 $ ./canplayer -l i vcan0=can0 -I 20170405_S-Zuffenhausen1.log
```

candump

Für Testzwecke lassen sich mit dem Werkzeug `candump` (Listing 2.5) die Daten, die sich auf dem virtuellen CAN-Bus befinden, auf der Konsole ausgeben.

Listing 2.5 Konsolenausgabe der Daten des virtuellen CAN-Busses

```
1 $ ./ candump vcan0
```

2.4 Datenübertragung (MQTT)

Die Kommunikation der Fahrzeuge mit dem Server findet mittels Message Queuing Telemetry Transport (MQTT) statt. MQTT ist ein Messaging Protokoll für das Internet of Things (IoT). Der Datenaustausch wird über das Senden von Nachrichten über einen zentralen Verteiler, auch *Broker* genannt, an interessierte Teilnehmer realisiert. Das MQTT Protokoll wurde für ressourcenarme Geräte entwickelt, die über schlechten Internetempfang verfügen [Obe18]. Dadurch ist die Benutzung dieses Protokolls ideal geeignet für den Raspberry Pi und dessen Senden der Daten vom Fahrzeug zum Server via mobilem Internet.

MQTT implementiert die Publish/Subscribe-Funktionsweise (Abbildung 2.3). Nachrichten, die ein Client sendet (publish), enthalten neben den eigentlichen Nutzdaten einen so genannten *Topic*. Jeder Client, der Nachrichten von diesem Topic empfangen möchte, abonniert (subscribe) es beim Broker. Dabei benachrichtigt der Broker bei neu veröffentlichten Nachrichten alle Abonnenten des Topics. Ist ein Abonnent aufgrund eines Verbindungsabbruchs nicht erreichbar gewährleistet der Broker, dass die Nachricht zugestellt wird sobald wieder eine Verbindung zu dem Client besteht. Dabei bietet MQTT unterschiedliche Quality of Service (QoS) Level an, um sicherzustellen, dass eine gesendete Nachricht beim Empfänger ankommt [Obe18]:

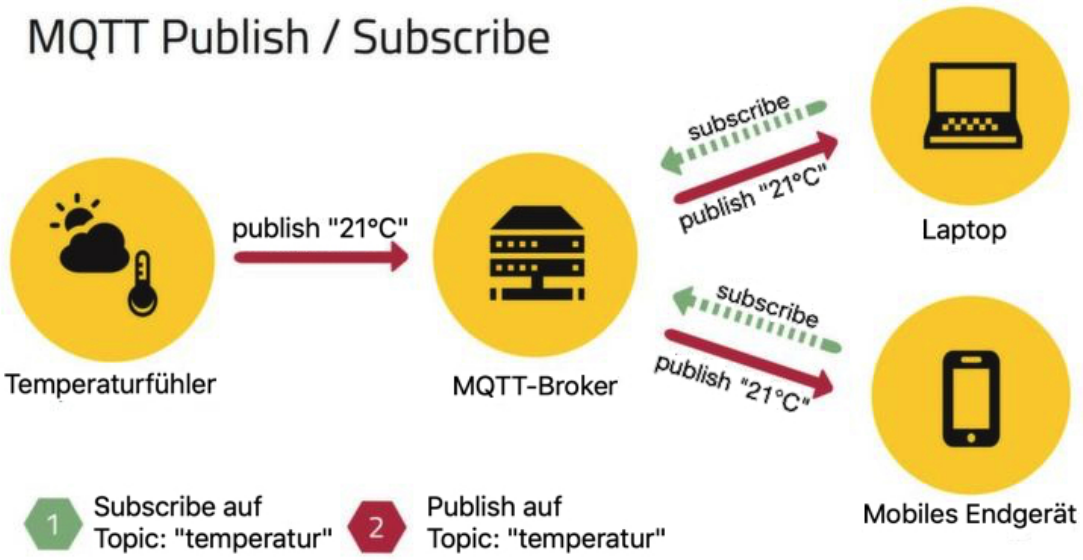

Abbildung 2.3: Publish/Subscribe-Funktionsweise von MQTT [Obe18]

- 0: at most once
- 1: at least once
- 2: exactly once

In dieser Arbeit wird ausschließlich das QoS Level 2 verwendet, da aufgrund der Benutzung von mobilen Daten mit Verbindungsabbrüchen zu rechnen ist und nur dieses Level sicherstellt, dass eine Nachricht genau ein mal beim Empfänger ankommt.

2.5 Kartendienst und Reverse Geocoding

Damit der Fahrer seine gefahrene Strecke nachvollziehen kann, wird auf dem Display des verwendeten Raspberry Pi eine Karte angezeigt auf dem die Strecke eingezeichnet ist. Hierfür wird OpenStreetMap (OSM) verwendet. OSM ist ein Open-Source-Kartendienst mit der größten freien Geodatenbank der Welt [Opeb]. Die Kartendaten werden von Freiwilligen, die mit GPS-Geräten unterwegs sind und ihre Wegstrecke aufzeichnen, aufgenommen [RT10, S. 3]. Diese Daten unterliegen der Open Data Commons Open Database Lizenz (ODbL), wodurch es jedem gestattet ist die Daten zu bearbeiten, kopieren, weiterzugeben sowie für kommerzielle Zwecke zu verwenden [Opea].

Die Serveranwendung hingegen verwendet *Nominatim*. Nominatim ist ein OSM-Tool, das die Fähigkeit *Reverse Geocoding* (auch *Inverse Geocodierung* genannt) besitzt, die anhand von GPS-Koordinaten nach Objekten in der Datenbank von OSM sucht. Alle von den Fahrzeugen empfangenen GPS-Koordinaten werden an Nominatim gesendet, um eine Straße und eine Stadt zu diesen Punkten zu erhalten und sie somit menschenlesbar zu machen. In Abbildung 2.4 ist die Antwort auf eine Anfrage an Nominatim für die Koordinate 48°N, 9°O zu sehen:

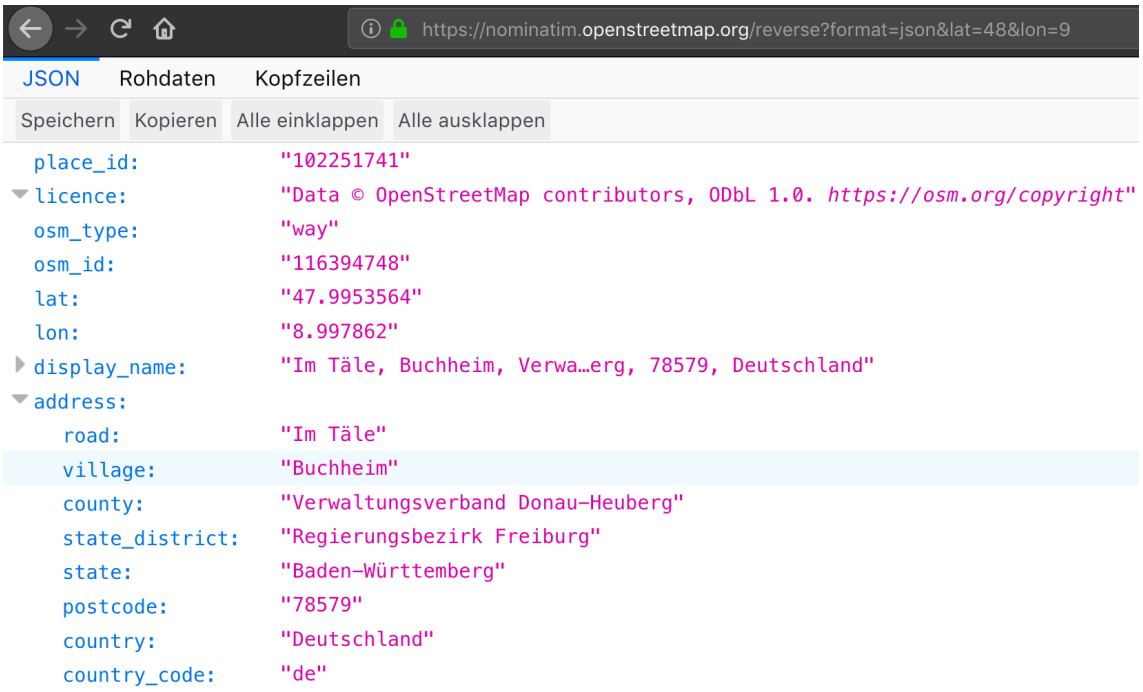

Abbildung 2.4: Antwort auf eine Reverse Geocoding Anfrage an Nominatim

Aufgrund der Nutzungsbedingungen von Nominatim ist nur eine Anfrage pro Sekunde gestattet. Außerdem muss jeder Punkt einzeln angefragt werden, da Nominatim keine Massenauswertung anbietet. [OSMF]

3 Analyse der bisherigen Applikation

Dieses Kapitel beschäftigt sich mit der Analyse der bisherigen Applikation, die aus einer vorangegangenen Studienarbeit übernommen wurde. Dabei wird zuerst auf das Gesamtkonzept der Applikation eingegangen. Im Abschnitt darauf ist ein Einblick auf die Benutzeroberfläche gegeben, die im Fahrzeug auf dem Touchscreen-Display des Raspberry Pi zu sehen ist. Daraufhin wird im nächsten Abschnitt die Erfassung der CAN-Daten untersucht. In Abschnitt 3.5 wird analysiert, inwiefern die Synchronisation der Microsoft Structured Query Language (SQL) Datenbank auf dem Server mit der MySQL Datenbank im Fahrzeug gewährleistet ist. Der letzte Abschnitt beschreibt, wie aus den CAN-Daten für jede Fahrt eine menschenlesbare Routenbeschreibung erstellt wird.

3.1 Konzept

Abbildung 3.1 beschreibt das Gesamtkonzept der bisherigen Applikation.

Sobald der Fahrer die Zündung des Fahrzeuges einschaltet, beginnt der Bootvorgang des Raspberry Pi und anschließend die Fahrzeuganwendung. Die Aufzeichnung der Fahrt startet automatisch. Während der Aufnahme der Fahrt kann sich der Fahrer zu jeder Zeit mit seinem NFC Tag als Fahrer identifizieren. Sollte kein Fahrer identifiziert werden, ist das Fahrzeug und dessen Aufzeichnung der Fahrt trotzdem im vollen Umfang nutzbar.

Zuerst wird eine neue Fahrt mit dem aktuellen Zeitstempel als Startzeit in der MySQL Datenbank angelegt. Durch den Zugang zum CAN-Bus erhält die Anwendung regelmäßig die GPS-Koordinaten und Kilometerstände des Fahrzeuges und legt sie in der MySQL Datenbank ab. Die GPS-Koordinaten werden für die Karte in der Benutzeroberfläche verwendet, um die gefahrene Route einzuzeichnen, die Kilometerstände und Fahrtzeiten hingegen, um die Fahrtdaten einzutragen.

Zeitgleich werden die gespeicherten Daten an den MQTT Broker gesendet, sofern eine Internetverbindung besteht. Sollte dies nicht der Fall sein, warten die nicht gesendeten Nachrichten in einer Warteschlange bis sie gesendet werden können. Beim MQTT Broker angekommen, werden die Nachrichten an den Server weitergeleitet.

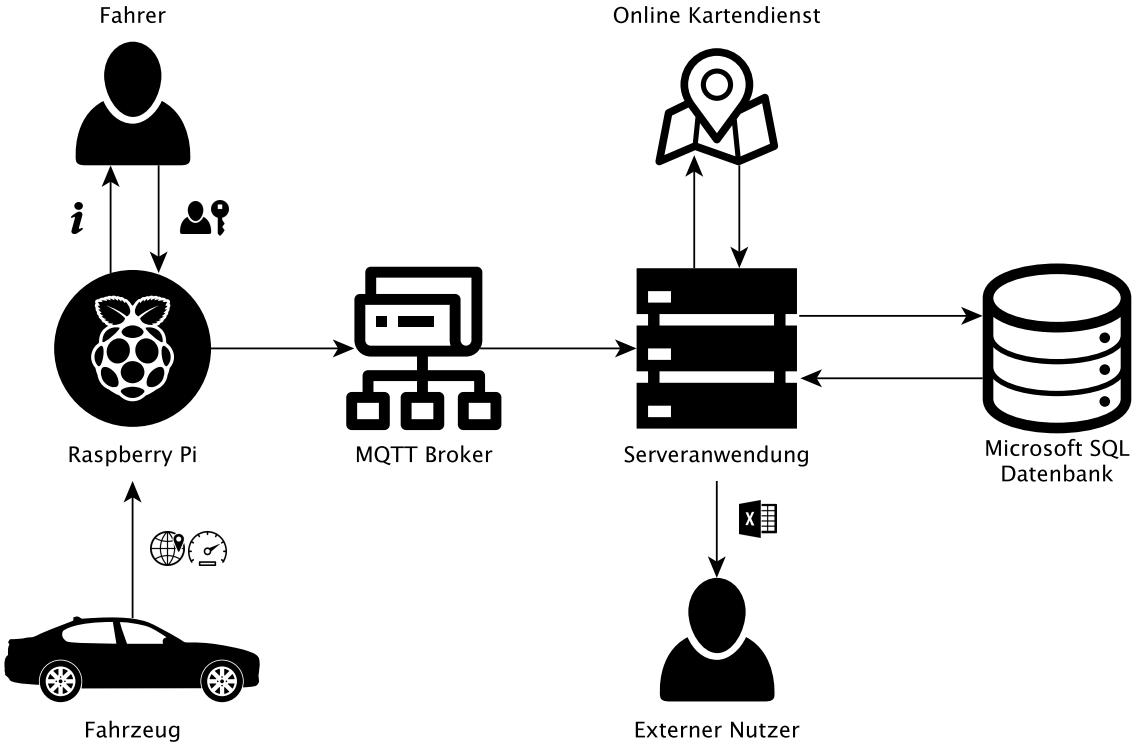

Abbildung 3.1: Gesamtkonzept (in Anlehnung an [Gro18])
Icons lizensiert nach Creative Commons 3.0, www.iconfinder.com

Kilometerstände und Fahrtzeiten werden von der Serveranwendung in der Microsoft SQL Datenbank gespeichert. Die GPS-Koordinaten hingegen, werden zuerst an den Nominatim Server geschickt, um die Straße und die Stadt der GPS-Punkte zu ermitteln und dann erst in die Datenbank aufgenommen.

Mitarbeiter des IVK können jeder Zeit beim Server das in der Datenbank gespeicherte Fahrtenbuch in Form einer Excel-Datei exportieren lassen. Dabei generiert ein Algorithmus für jede Fahrt eine Routenbeschreibung in Textform, um die Fahrten auf diese Weise menschenlesbar darzustellen.

3.2 Benutzeroberfläche

In Abbildung 3.2 ist die Benutzeroberfläche zu sehen, die dem Fahrer auf dem 7 Zoll Touchscreen-Display des Raspberry Pi angezeigt wird. Die Benutzeroberfläche wurde mit Hilfe der Hypertext Markup Language (HTML) beschrieben und wird durch den Webbrowser Midori dargestellt. Auf der linken Seite sind die Daten der Fahrt aufgezeichnet. Diese bestehen aus dem Vor- und Nachnamen des Fahrers, der durch sein NFC Tag erkannt wurde, der Startzeit und dem Kilometerstand des Fahrzeuges zu Beginn der Fahrt. Nach Beendigung der Fahrt werden zusätzlich noch die Endzeit

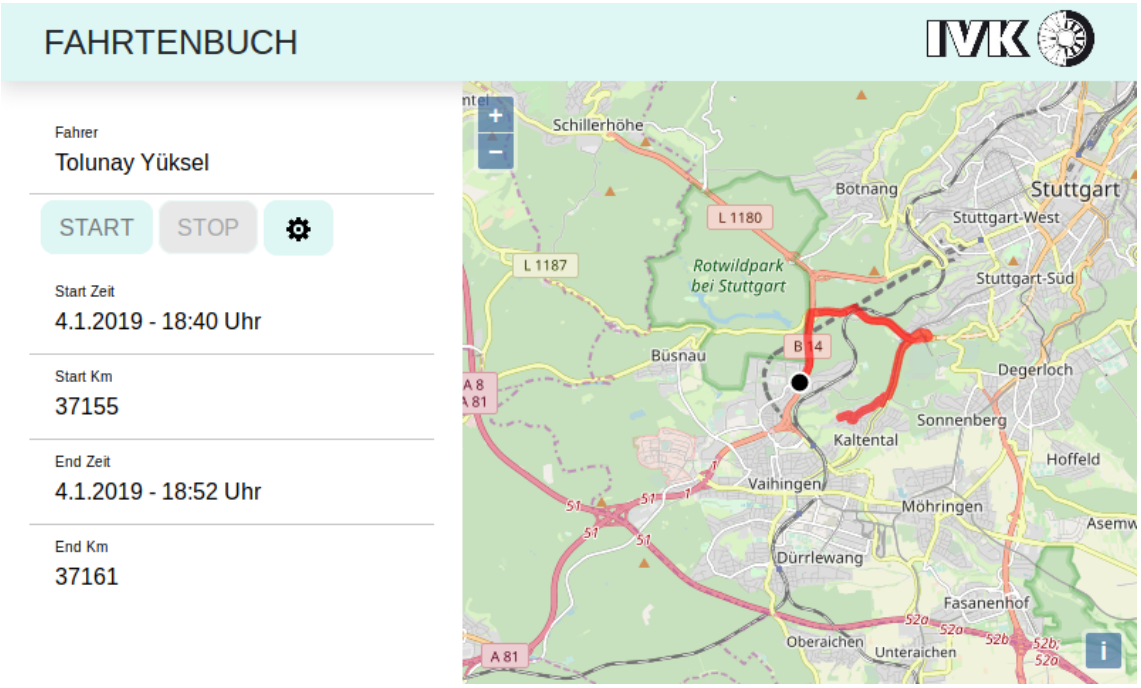

Abbildung 3.2: Benutzeroberfläche nach Beendigung der Fahrt per Stop-Button

und der momentane Kilometerstand angezeigt. Darüber hinaus sind dem Fahrer zwei Buttons gegeben, um die Aufnahme einer Fahrt manuell zu starten oder zu beenden.

Im rechten Teil der Benutzeroberfläche ist die von OpenStreetMap zur Verfügung gestellte Karte. Die gefahrene Route wird auf der Karte anhand der erfassten GPS Koordinaten eingezeichnet. Außerdem lässt sich die Karte bewegen und sowohl hinein als auch heraus zoomen.

3.3 Kommunikation innerhalb der Fahrzeuganwendung

Um dem Fahrer die Benutzeroberfläche anbieten zu können, werden Daten aus der Fahrzeugdatenbank benötigt. Da Node.js für seine Ausführung von serverseitigem JavaScript Code bekannt ist, wird mithilfe von Express ein Webserver errichtet, wodurch die Fahrzeuganwendung in zwei Teile getrennt wird (Front- und Backend). Dabei ist das Backend nur für das Erfassen der CAN-Daten und für die Datenübertragung zum MQTT Broker zuständig. Das Frontend hingegen hat die einzige Aufgabe darin, dem Fahrer eine Benutzeroberfläche anzubieten.

In Abbildung 3.3 ist zu sehen, dass das Fronend eine Anfrage für die Benutzeroberfläche an den Webserver des Backends richtet. Dies geschieht durch das Aufrufen der lokalen Seite *http://localhost:8080* im Browser Midori. Als Antwort erhält das Frontend die Benutzeroberfläche, die zur Darstellung auf das Framework Angular zurückgreift.

Der Webserver des Fahrzeug-Backends wird durch die Routing Fähigkeit von Express erweitert. Dabei werden Unterseiten, sogenannte *Anwendungsendpunkte*, erstellt. Je nach Routing Konfiguration wird bei einem Aufruf dieser Unterseiten entschieden, welche Funktion des Backends ausgeführt wird, um dem Frontend beispielsweise die Erfassten GPS-Koordinaten zugänglich zu machen, damit sie in die Karte eingezeichnet werden können.

Darüber hinaus wird die Kommunikation um eine WebSocket Verbindung (in Abbildung 3.3 rot markiert) ergänzt. Dabei handelt es sich um eine bidirektionale Verbindung, die Push-Nachrichten zeitnah zustellen kann, wodurch dem Backend ermöglicht wird, die Benutzeroberfläche zu steuern. Somit kann das Backend bei neu erfassten Daten dem Frontend mitteilen, dass die Benutzeroberfläche aktualisiert werden soll.

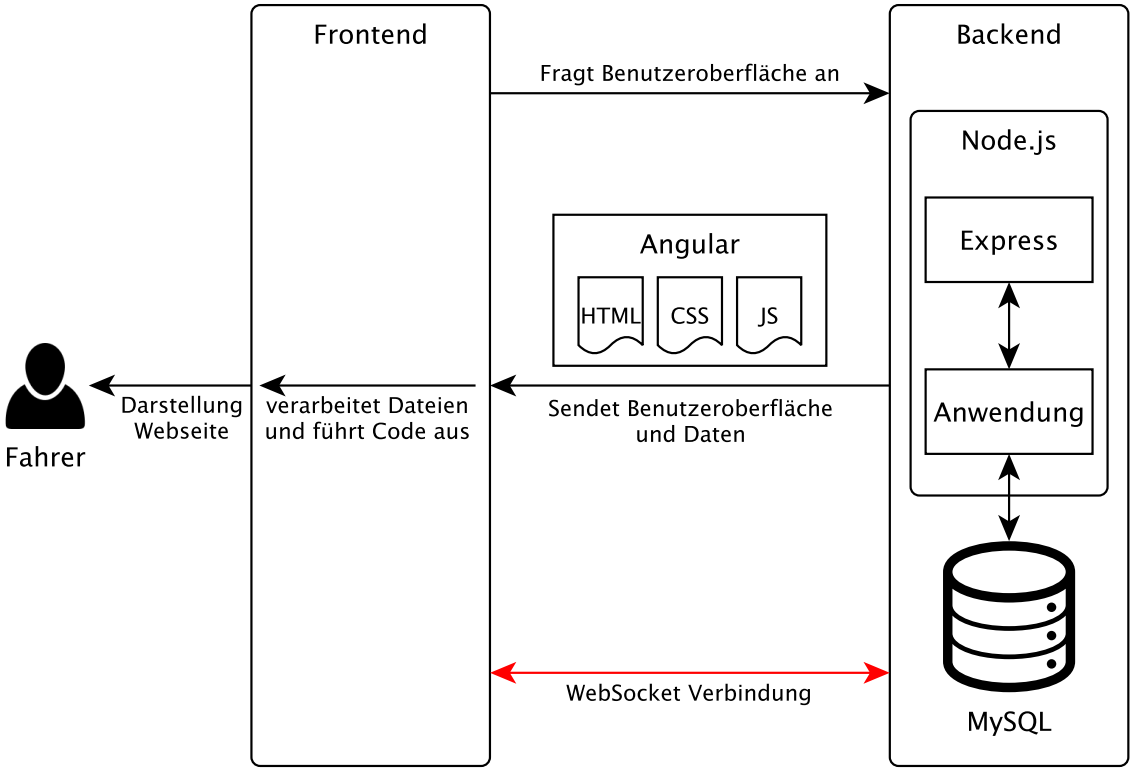

Abbildung 3.3: Kommunikation zwischen Fahrzeug-Frontend und -Backend (in Anlehnung an [Gro18])
Icons lizensiert nach Creative Commons 3.0, www.iconfinder.com

3.4 Erfassung der CAN-Daten

Die Fahrzeuganwendung liest alle fünf Sekunden die CAN Nachricht mit dem Message Identifier 0x500 ein, in dem die GPS-Koordinaten des Fahrzeugs enthalten sind. Dies geschieht ebenfalls wenn das Fahrzeug zum Beispiel durch einen Tunnel fährt und dort seine GPS-Position nicht orten kann. In diesem Fall liegt am CAN-Bus eine GPS-Koordinate an, die sowohl im Breiten- als auch Längengrad einen Wert von höher als 180 hat, was somit eine ungültige GPS-Position darstellt. Zusammen mit dem Identifier (ID) der aktuellen Fahrt werden die erfassten GPS-Koordinaten in die MySQL Datenbank aufgenommen.

Alle 60 Sekunden wird über die CAN Nachricht mit dem Message Identifier 0x624 der Kilometerstand des Fahrzeuges ermittelt. Dabei wird beim ersten Auslesen der Kilometerstand als Startwert zum Beginn der Fahrt in die Datenbank aufgenommen. In jedem weiteren Durchlauf wird dann der Endkilometerstand aktualisiert. Nach jeder Aufnahme des Kilometerstandes als entweder Start- oder Endkilometerstand wird die Benutzeroberfläche aktualisiert, um die in dieser Periode neu erfassten GPS-Punkte in der Karte einzeichnen zu lassen.

Sobald eine Fahrt durch Ausschalten der Zündung des Fahrzeuges oder Betätigung des Stop Buttons am Touchscreen-Display des Raspberry Pi beendet wurde, wird in der Datenbank ein aktueller Zeitstempel als Endzeit der Fahrt vermerkt.

3.5 Synchronisation der Datenbank

Sobald die Fahrzeuganwendung gestartet wurde und sich mit dem MQTT Broker verbunden hat, wird an den Server eine Benachrichtigung geschickt, dass das Fahrzeug online ist. Daraufhin sendet der Server die Userdaten aus seiner Datenbank an das Fahrzeug, damit das Fahrzeug diese in dessen eigene Datenbank speichern kann und für die Identifizierung des Fahrers immer aktuelle Daten zur Verfügung hat.

Zeitgleich wird in der Fahrzeugdatenbank ein neuer Trip angelegt, der zu diesem Zeitpunkt nur eine Startzeit besitzt. Danach wird dieser Trip an den Server gesendet.

Die GPS-Daten, die alle fünf Sekunden erfasst werden, werden anschließend an den MQTT Broker übermittelt. Kilometerstände und Zeitstempel hingegen, werden nur nach Beenden der Fahrt an den Server geschickt.

Da es bei dem im Fahrzeug mitgeführten mobilen Hotspot zu Verbindungsabbrüchen kommen kann, werden alle 60 Sekunden, nachdem die Anwendung den Kilometerstand für die Aufzeichnung der aktuellen Fahrt ermittelt hat, die Tripdaten der letzten drei Fahrten an den Server geschickt.

Um fehlende GPS-Daten nachzuholen, ermittelt der Server anhand der alle 60 Sekunden nachgelieferten Tripdaten, ob eine Lücke in den GPS-Daten zu diesen drei Trips existiert und beschafft sich dann die fehlenden Daten vom Fahrzeug. Dabei sendet der Server die IDs der fehlenden GPS-Einträge einzeln an den MQTT Broker. Für jede ID, die beim Fahrzeug angekommen ist, wird der GPS-Eintrag aus der Datenbank gelesen und wiederum einzeln an den MQTT Broker geschickt.

3.6 Routenerkennung

Für den Excel Export gilt es für jede Fahrt anhand der erfassten GPS-Daten eine menschenlesbare Routenbeschreibung zu generieren. Folgendes ist ein Muster für eine Routenbeschreibung:

```
Von Mühlstraße, Tübingen
nach Torstraße, Stuttgart
(über L1208 - B464 - A81)
```

Da mehrere Wege von A nach B führen können, sind in einer derartigen Routenbeschreibung neben der Start- und Zieladresse zusätzlich noch die drei am meisten befahrenen Straßen vermerkt. Mit „am meisten befahren“ sind die Straßen gemeint, auf denen in Kilometern am längsten gefahren wurde. Einfach nur die GPS-Punkte einer Fahrt zu zählen, würde die Zeit, die man auf der jeweiligen Straße verbracht hat, repräsentieren. Dies würde jedoch bei roten Ampeln und Staus zu Problemen führen.

Um die drei am meisten befahrenen Straßen zu ermitteln, berechnet ein Algorithmus für jede Straße in einer Fahrt, die weder die Start- noch die Zielstraße ist, die Strecke in Kilometern. Dabei wird der Abstand von jedem erfassten GPS-Punkt einer Straße zum Nächsten berechnet. In der Summe ergibt sich dann die gefahrene Distanz dieser Straße. Die Berechnung der Abstände erfolgt mithilfe des euklidischen Abstandes:

$$Distanz = \sqrt{(Längengrad_1 - Längengrad_2)^2 + (Breitengrad_1 - Breitengrad_2)^2}$$

4 Anforderungen an die optimierte und erweiterte Applikation

In diesem Kapitel wird untersucht, welche Anforderungen an die Applikation zu stellen sind auf Basis der Analyse aus Kapitel 3.

4.1 Synchronisation der Datenbank auf dem Server

Da während der Aufzeichnung einer Fahrt immer wieder mit Verbindungsabbrüchen zu rechnen ist, müssen die Daten, die nicht gesendet werden konnten, zu einem späteren Zeitpunkt nachgesendet werden können.

MQTT ermöglicht während Verbindungsabbrüchen die Daten, die nicht gesendet werden können, in einer Warteschlange zwischen zu speichern. Sobald die Verbindung wieder steht, werden die Nachrichten an den MQTT Broker übermittelt. Dies ist jedoch nur während der Fahrt, also wenn die Fahrzeuganwendung auf dem Raspberry Pi in Betrieb ist, möglich. Deshalb ergibt sich daraus der Anlass gewisse Synchronisationsroutinen einzuführen.

Bisher werden die Tripdaten der letzten drei Fahrten alle 60 Sekunden an den Server geschickt. Deren nicht versendete GPS-Daten hingegen werden von der Serveranwendung ermittelt. Dabei wird anhand der mitgesendeten GPS-IDs der erfolgreich gesendeten GPS-Daten überprüft, ob die IDs konsekutiv fortlaufend sind. Dabei werden jedoch nur die fehlenden GPS-Punkte ermittelt, deren ID zwischen der ID der ersten und letzten erfolgreich gesendeten GPS-Punkten liegt.

Das Problem dabei ist, dass die fehlenden Fahrt- und GPS-Daten, die länger als drei Fahrten zurückliegen, niemals beim Server ankommen werden. Das gleiche gilt für die GPS-Daten einer Fahrt, die nach dem letzten erfolgreich gesendeten GPS-Eintrag erfasst wurden. Deshalb gilt es eine Strategie zu finden, die gewährleistet, dass die fehlenden Daten vom Server korrekt ermittelt und beim Fahrzeug angefragt werden. Für den Fall, dass die neu angefragten Daten wieder nicht gesendet werden können, muss die Synchronisation in einer gewissen Routine stattfinden, um somit die fehlenden Daten während einer bestehenden Internetverbindung des Fahrzeuges letztendlich beschaffen zu können.

4.2 Aufbereitung der GPS-Daten

Im Zeitraum der Entwicklung dieser Applikation wurde oftmals die Erfahrung gemacht, dass es nicht reicht, die GPS-Koordinaten des Fahrzeugs zu ermittelt, um somit die Fahrt rekonstruieren zu können. Immer wieder schleichen sich „schlechte“ GPS-Einträge in die Datenbank, die entweder fehlerhaft oder sogar nicht verwertbar sind.

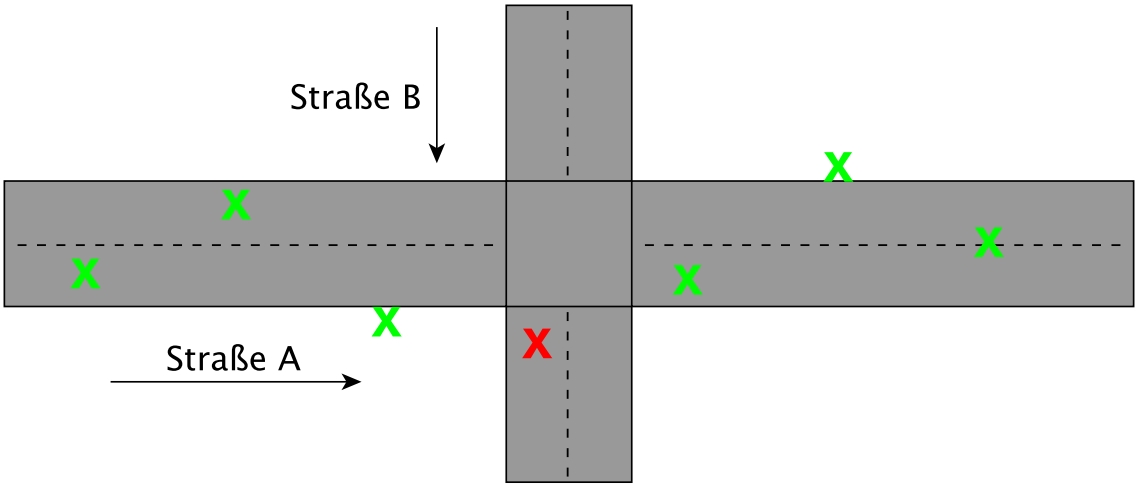

Abbildung 4.1: Kreuzung mit einem Beispiel für eine ungenaue GPS-Position (rot) (in Anlehnung an [Gro18])

Grund für solche GPS-Einträge in der Datenbank ist die Genauigkeit des GPS Systems. Diese liegt bei über 10 m. Das kann bei der Reverse Geocoding Anfrage an den Nominatim Server zu falschen Werten führen. Ein Beispiel hierfür ist in der Kreuzung in Abbildung 4.1 dargestellt. Wie hier zu sehen ist liegen bei einer Fahrt auf der Straße A von links nach rechts die erfassten Positionen nicht immer auf der gefahrenen Strecke. Im Fall einer Überquerung der Seitenstraße B, kann es dazu führen, dass das Reverse Geocoding von Nominatim aufgrund der ungenauen Ortung folgendes Ergebnis liefern könnte:

[A A A B A A A]

Ein weiterer Grund für schlechte GPS-Einträge ist die Qualität der Daten aus der Datenbank von OpenStreetMap. Da die Geodaten von Freiwilligen aufgezeichnet und dann mit Zusatzinformationen wie zum Beispiel Straßennamen versehen in die Datenbank aufgenommen werden, fehlen hin und wieder einige Informationen. Deshalb kann es dazu führen, dass die Antwort auf Nominatims Reverse Geocoding Anfrage keinen Straßennamen liefert. Wie man aus Abbildung 4.2 entnehmen kann, tritt dieser Fall bei einer Anfrage der Koordinate 48.7453011°N, 9.1345099°O ein.

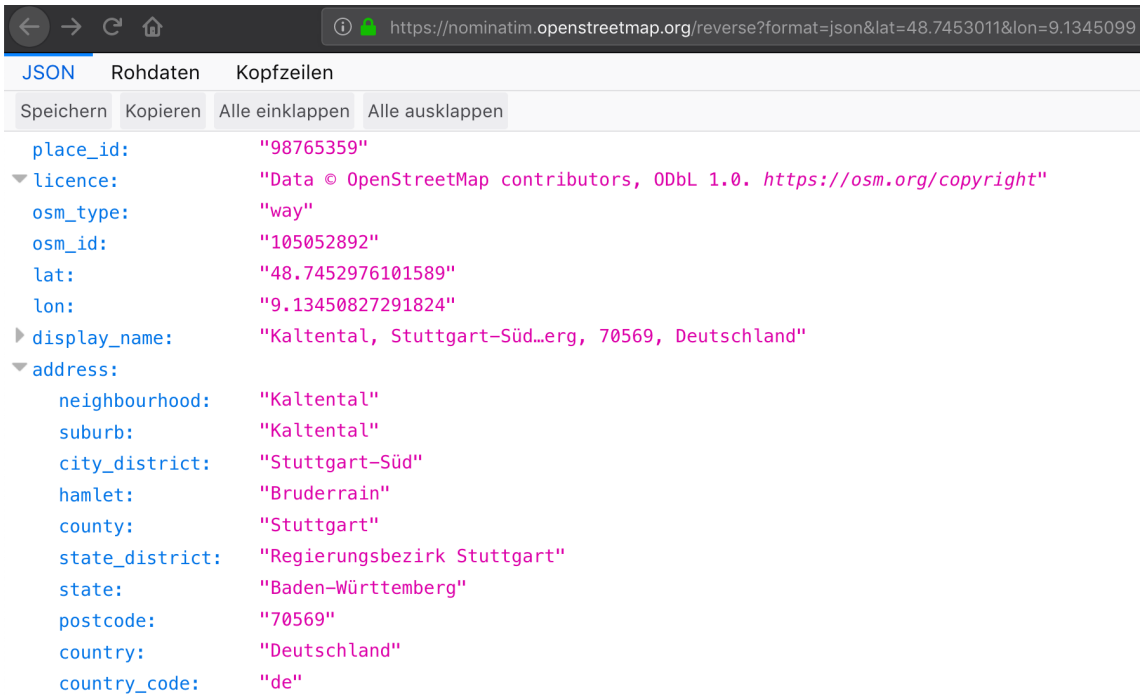

Abbildung 4.2: Reverse Geocoding Antwort von Nominatim ohne Straße

Um für das Fahrtenbuch eine korrekte Route erkennen und darauf eine korrekte Routenbeschreibung generieren zu können, ist es nötig die erfassten GPS-Einträge aufzubereiten und die oben genannten Probleme zu eliminieren.

4.3 Routenerkennung

Für die Generierung der Routenbeschreibung wird bisher bei der Ermittlung der drei am meisten befahrenen Straßen der euklidische Abstand verwendet. Da jedoch GPS-Koordinaten in Grad sind, lässt sich der Abstand zwischen zwei GPS-Koordinaten nicht mit der bisherigen Formel berechnen.

Wie in Abbildung 4.3 zu sehen ist, verlaufen Breitengrade parallel zueinander. Der Abstand zweier Breitengrade beträgt somit immer 111,3 Kilometer. Längengrade hingegen verlaufen nicht parallel, sondern treffen sich alle sowohl in Nord- als auch Südpol. Aus diesem Grund variiert der Abstand zwischen zwei Längengraden in Abhängigkeit von der geografischen Breite. Während der Abstand zweier Längengraden am Äquator ebenfalls 111,3 Kilometer beträgt, ist der Abstand an den Polen hingegen 0. [Kom]

Somit gilt es eine Lösung zu finden, die es ermöglicht, den Abstand zweier GPS-Koordinaten korrekt zu berechnen.

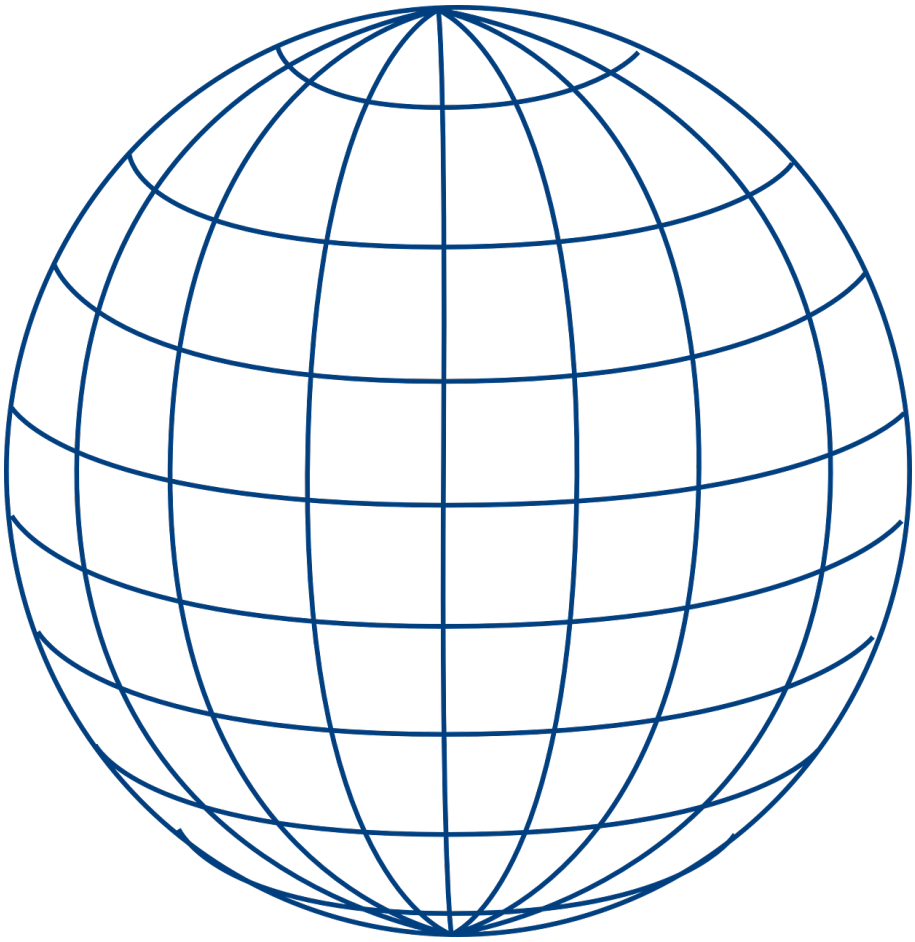

Abbildung 4.3: Globus mit Längen- und Breitengraden

4.4 Erweiterung zur Unterstützung mehrerer Fahrzeuge

Die bisherige Applikation wurde nur so weit entwickelt, dass nur ein Fahrzeug unterstützt wird. Da dem IVK jedoch mehrere Elektrofahrzeuge zur Verfügung stehen, die im Besitz einer Zulassung als Erprobungsfahrzeug sind, müssen mehrere Fahrtenbücher geführt werden. Deshalb gilt es die Applikation auf mehrere Fahrzeuge zu erweitern.

Hierzu muss jedes Fahrzeug vom Server eindeutig erkennbar sein ohne dass es zu einer Verwechslung kommen kann. Das heißt, dass der Server bei jeder Nachricht, die er empfängt, wissen muss, von welchem Fahrzeug die gesendeten Daten stammen und dementsprechend dem gleichen Fahrzeug gegebenenfalls antworten.

Außerdem gilt es die Fahrten und deren GPS-Punkte in der Datenbank des Servers verwechslungsfrei einem Fahrzeug zuordnen zu können.

5 Implementierung

5.1 Fahrzeug-Backend

Die automatisierte Erfassung der Trip Daten erfolgt durch das Fahrzeug-Backend. Sie ist aus mehreren Modulen aufgebaut. Ein Überblick über die Verwendung der Module ist in Abbildung 5.1 dargestellt. Dieses Kapitel beschreibt die in den Fahrzeugen verwendete Datenbank und geht anschließend auf die einzelnen Module ein. In Abbildung 5.2 auf Seite 41 wird der Ablauf des Fahrzeug-Backends erläutert. Zur Übersicht sind die Anweisungen den zugehörigen Modulen zugeordnet.

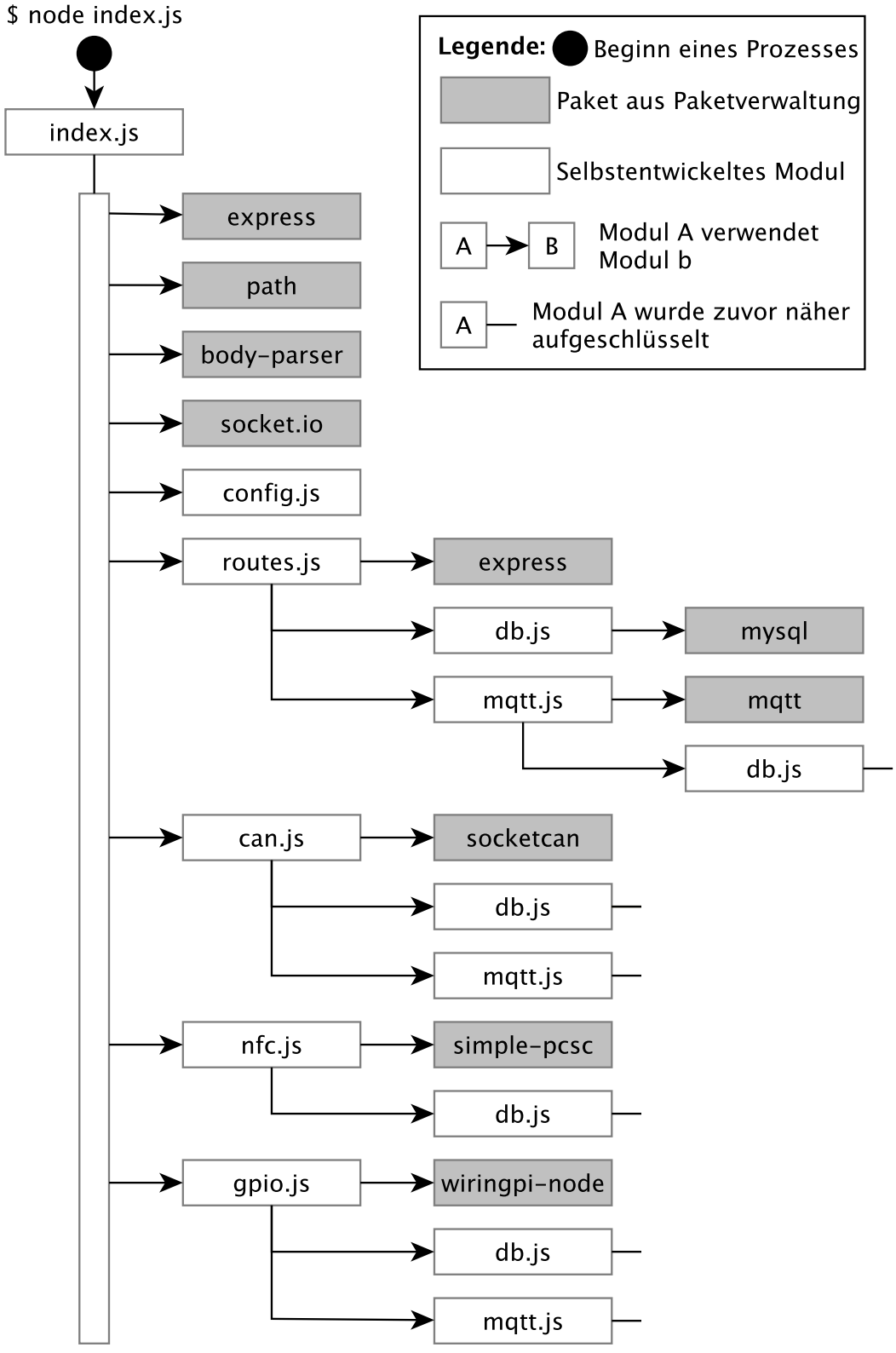

Abbildung 5.1: Aufbau der Fahrzeuganwendung [Gro18]

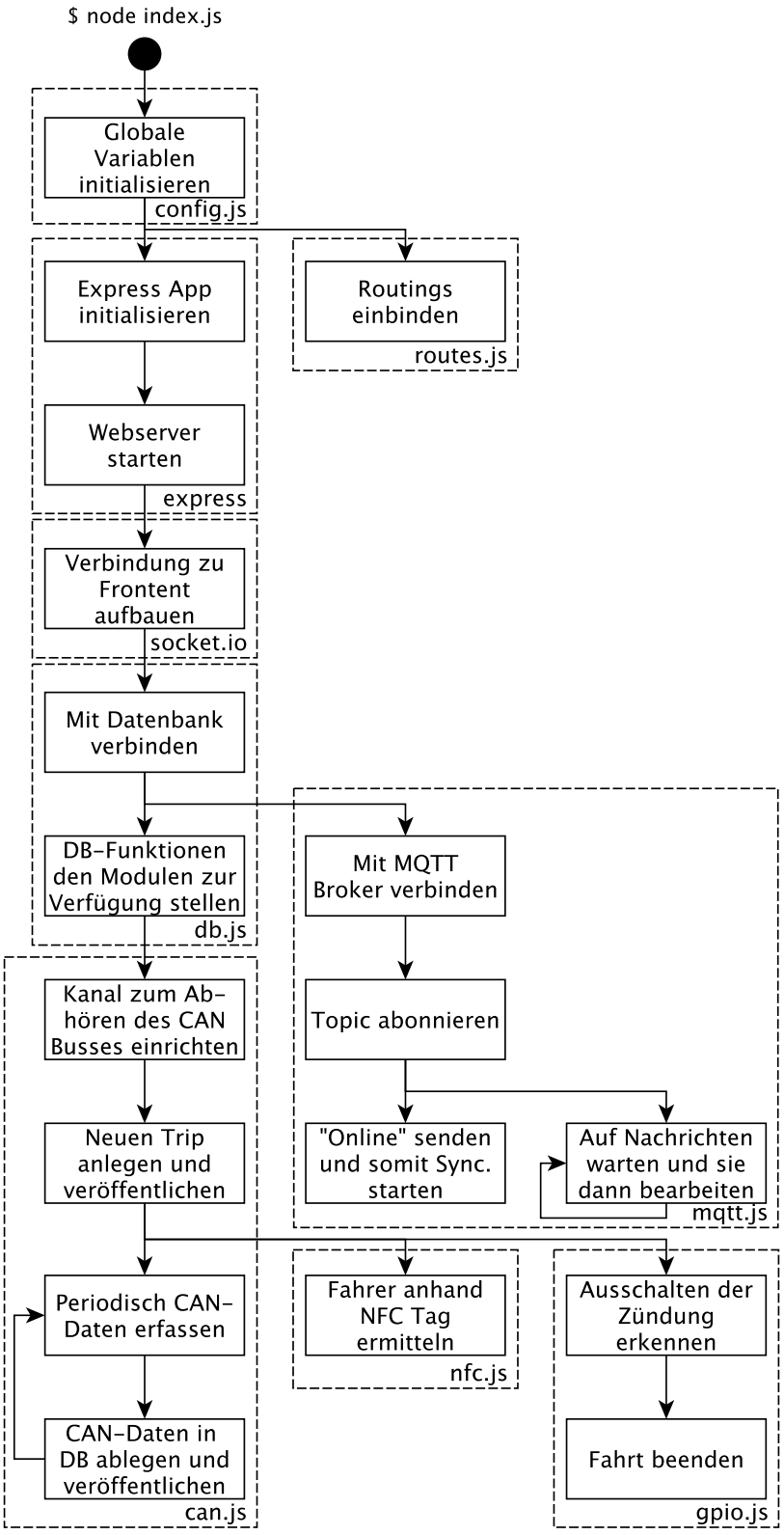

Abbildung 5.2: Grobe Übersicht über den Ablauf der Fahrzeuganwendung

5.1.1 Datenbank

Für den Fall, dass die Serveranwendung nicht in Betrieb ist oder der mobile Hotspot im Fahrzeug keine Internetverbindung hat, legt die Fahrzeuganwendung die erfassten Daten in dessen MySQL Datenbank ab. Die Datenbank besteht aus den Tabellen *Users*, *Trips* und *GPSLookups*. Im folgenden wird beschrieben, wie die drei Tabellen aufgebaut sind und welche Eigenschaften ihre Spalten ausweisen.

Users

Die Users Tabelle wird verwendet, um die Fahrer aus der Serverdatenbank hier abzulegen und bei der Erkennung eines NFC Tags anhand dessen UID den Fahrer zu ermitteln und ihn der Fahrt zuzuordnen.

In Tabelle 5.1 sind die Spalten und deren Eigenschaften der Tabelle Users aufgelistet. Primärschlüssel der Tabelle ist `id`, wodurch jeder Fahrer eindeutig identifizierbar ist. Wichtig hierfür ist, dass die ID der Fahrer mit den User IDs bei der Datenbank des Server übereinstimmen, damit beim Server jede User ID dem richtigen Fahrer zugeordnet werden kann.

Da jeder Fahrer durch seinen eigenen NFC Tag eindeutig erkannt werden muss und es zu keiner Verwechslung kommen darf, ist die Spalte `uid` auf Unique (einzigartig) gesetzt. Dadurch ist gewährleistet, dass nicht mehrere Fahrer den selben NFC Tag benutzen können.

Die Spalten `first_name` und `last_name` werden verwendet, um den Namen des Fahrers auf der Benutzeroberfläche anzuzeigen.

Keines dieser Spalten erlaubt leere Eingaben für einen Eintrag. Dadurch muss jeder Fahrer, der in diese Tabelle aufgenommen wird, eine ID, Vor- und Nachnamen und eine eigene UID durch seinen NFC Tag besitzen.

Tabelle 5.1: Spalteneigenschaften der Tabelle Users im Fahrzeug

Spalte	Datentyp	Nullable	Unique	Schlüssel
id	int	nein	ja	primär
first_name	varchar	nein	nein	-
last_name	varchar	nein	nein	-
uid	varchar	nein	ja	-

Trips

In der Tabelle Trips sind Trip ID, Start- und Endzeit, Start- und Endkilometerstand und Fahrer vermerkt (Tabelle 5.2). Der Primärschlüssel `id` wird bei jedem neuen Eintrag einer Fahrt automatisch inkrementiert. In Bezug auf den Primärschlüssel `id` aus der User Tabelle agiert `user_id` hier als Fremdschlüssel.

Die letzten vier Spalten aus Tabelle 5.2 können alle den Wert `null` annehmen, da es aufgrund der Asynchronität der Anwendung sein könnte, dass diese Daten zu Beginn der Anwendung noch nicht erfasst wurden. Im Laufe der Fahrt werden diese Daten dann regelmäßig aktualisiert.

Tabelle 5.2: Spalteneigenschaften der Tabelle Trips im Fahrzeug

Spalte	Datentyp	Nullable	Autoincrement	Unique	Schlüssel
id	int	nein	ja	ja	primär
created_at	datetime	nein	nein	nein	-
ended_at	datetime	ja	nein	nein	-
start_km	int	ja	nein	nein	-
end_km	int	ja	nein	nein	-
user_id	int	ja	nein	nein	fremd

GPSLookups

Die Tabelle GPSLookups besteht aus den in Tabelle 5.3 aufgelisteten Spalten. Wie auch in der Tabelle Users, sind hier keine der Spalten in der Lage den Wert `null` anzunehmen, wodurch gewährleistet ist, dass jeder Positionseintrag in der Datenbank vollständig vermerkt ist.

Ebenso wie die Tabelle Trips hat GPSLookups auch einen Fremdschlüssel (`trip_id`). Durch ihn lassen sich die GPS-Einträge eindeutig einer Fahrt zuordnen.

Tabelle 5.3: Spalteneigenschaften der Tabelle GPSLookups im Fahrzeug

Spalte	Datentyp	Nullable	Autoincrement	Unique	Schlüssel
id	int	nein	ja	ja	primär
created_at	datetime	nein	nein	nein	-
latitude	char	nein	nein	nein	-
longitude	char	nein	nein	nein	-
trip_id	int	nein	nein	nein	fremd

5.1.2 Unveränderte Module

index.js

Die Fahrzeuganwendung wird durch dieses Modul mittels `$ node index.js` gestartet. Dabei werden für den Betrieb die Module *express*, *path*, *body-parser* und *socket.io* geladen.

Für die Erläuterung von Express siehe Abschnitt 2.2.2.

Path ist ein Standardmodul von Node.js, welches nicht extra installiert werden muss. Es stellt Funktionen zum Arbeiten mit Datei- und Verzeichnispfaden zur Verfügung.

Das Modul body-parser ist eine Erweiterung für express. Es stellt weitere Funktionen zur Verarbeitung von Anfragen zur Verfugung.

Socket-IO ist für die Herstellung der WebSocket Verbindung zwischen Backend und Frontend zuständig. Die WebSocket Verbindung wurde in einer globalen Variable initialisiert, um sie innerhalb des Fahrzeug-Backends überall zugänglich zu machen.

config.js

Die Datei `config.js` ist eine reine Konfigurationsdatei, in der statische Variablen mit `global.variablenName = "Wert"` definiert sind, worauf die gesamte Anwendung Zugriff hat. Darin befinden sich die Zugangsdaten zur MySQL Datenbank und zum MQTT Broker. Darüber hinaus lässt sich hier das Zeitintervall für das Aktualisieren der Benutzeroberfläche und Erfassen des Kilometerstandes oder der Position des Fahrzeuges konfigurieren. Außerdem ist es in dieser Datei möglich, den für die Klemme 15 (Zündung) verwendeten GPIO Pin auf dem Raspberry Pi zu wählen.

Das Ziel dieser Datei ist es, alle nötigen Einstellungen an einem Ort zu haben und somit den Aufwand bei der Portierung auf ein neues Gerät oder ein neues Fahrzeug möglichst gering zu halten.

nfc.js

Dieses Modul ist zuständig für die Fahreridentifizierung. Um mit dem NFC Reader kommunizieren zu können, wird das Paket *simple-pcsc* verwendet.

Zuerst wird der NFC Reader initialisiert und gewartet bis ein NFC Tag erkannt wird (Abbildung 5.3). Sobald dies geschieht, wird die Funktion `reader.on('card')` aufgerufen. Die vom Lesegerät erkannte UID befindet sich in der Variable `card.uid`. Anhand dieser UID wird in der Datenbank die ID des Fahrers ermittelt und in

der Trips Tabelle als User ID vermerkt. Anschließend wird über die aufgebaute WebSocket Verbindung des Moduls `socket.io.js` durch Aufrufen der Funktion `io.emit(„updateTrip“)` die Benutzeroberfläche aktualisiert.

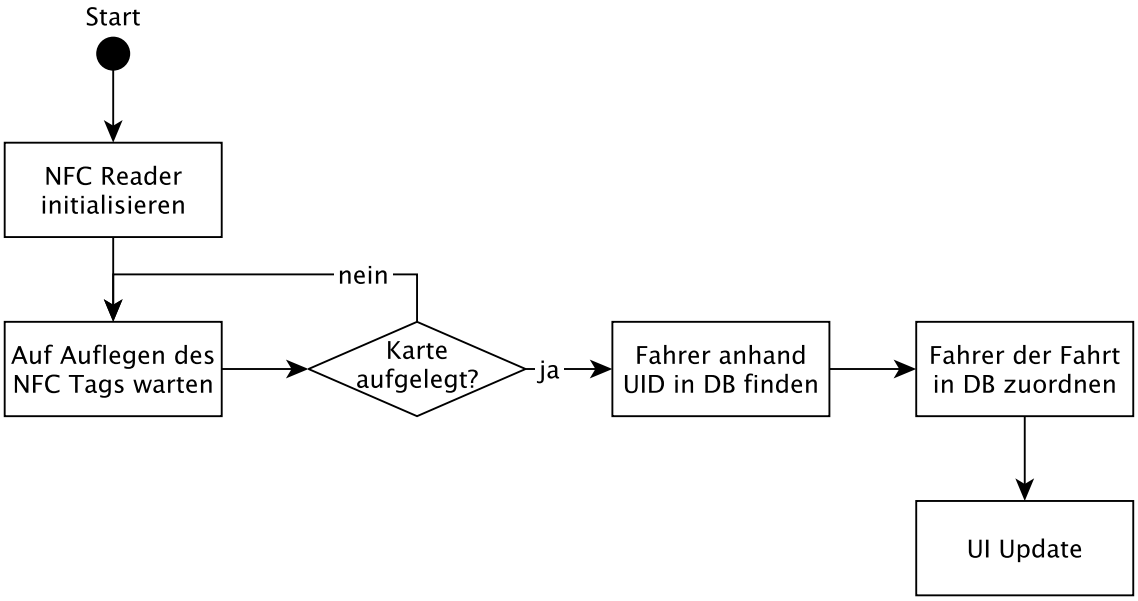

Abbildung 5.3: Ablaufdiagramm nfc.js [Gro18]

gpio.js

Das Modul `gpio.js` hat die Aufgabe, automatisch das Fahrtende durch Ausschalten der Zündung zu detektieren, bevor das System heruntergefahren wird. Die Stromversorgung des Raspberry Pi kommt im Fahrzeug durch ein Energiemanagement Hardware Attached on Top (HAT)-Modul zustande. Das HAT-Modul setzt den digitalen GPIO Pin 4 des Raspberry Pis zu Beginn der Fahrt auf LOW. Sobald der Fahrer die Zündung des Fahrzeuges ausschaltet, wird durch das HAT-Modul der Pegel des GPIO Pins auf HIGH geändert, um den Raspberry Pi herunterzufahren und anschließend herunterzufahren. Dies geschieht jedoch nicht sofort, sondern erst nach Ablauf eines Timers, damit die Anwendung noch Zeit hat, um die Fahrt zu beenden, Daten an den Broker zu übermittelt und anschließend zu terminieren.

Zur Erkennung der Pegel Änderung wird der Node.js Wrapper *wiringpinode* verwendet, der auf der GPIO Library *WiringPi* aufbaut. Dabei wird zuerst mit `wpi.pinMode(pin, wpi.INPUT)` der Modus des Pins als Input Pin registriert (Abbildung 5.4). Der Wechsel des Pegels von LOW zu HIGH wird durch die Funktion `wpi.wiringPiISR(pin, wpi.INT_EDGE_RISING, ...)` erkannt. In so einem Fall wird sofort dessen Callback-Funktion aufgerufen, die dafür sorgt, dass die Endzeit der Fahrt in der Datenbank aktualisiert wird und die Trip-Daten an den MQTT Broker

gesendet werden. Außerdem wird die Benutzeroberfläche aktualisiert, damit der Fahrer noch die Gelegenheit hat, die aktuellen Daten einzusehen bevor das System herunterfährt.

Während der Entwicklung der Applikation wird dieses Modul für Testzwecke ohne Fahrzeug auskommentiert, da keine Simulation der Zündungserkennung vorhanden ist.

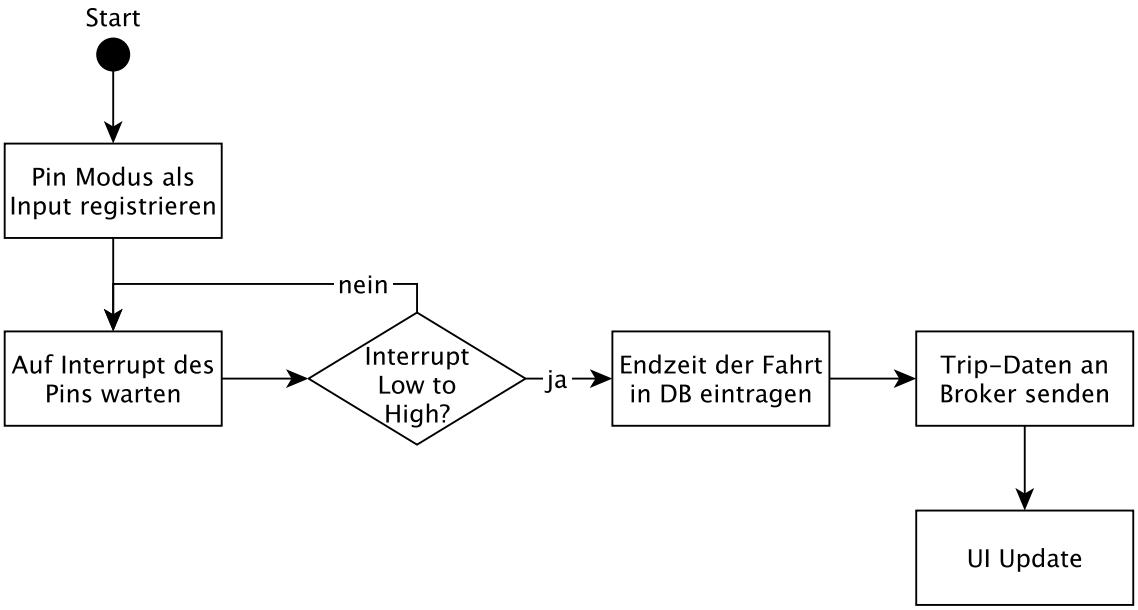

Abbildung 5.4: Ablaufdiagramm gpio.js (in Anlehnung an [Gro18])

5.1.3 routes.js

Um dem Frontend für die Gestaltung der Benutzeroberfläche gewisse Daten aus der Datenbank zugänglich zu machen, wird durch das Modul `routes.js` eine Anwendungsschnittstelle geschaffen. Dabei werden mit der *Routing* Komponente von Express `HTTP GET` Anfragen zu sogenannten Anwendungsendpunkten definiert, die bei Aufruf des Pfades die Daten aus der Datenbank in Form eines JavaScript Object Notation (JSON) Arrays zurückgeben.

Die verschiedenen Pfade zu den Anwendungsendpunkten sind aus Abbildung 5.5 zu entnehmen. Die `HTTP GET` Anfragen werden durch die Funktion `router.get('path')` definiert. Innerhalb der mitgegebenen Callback-Funktion werden die Daten aus der Datenbank mit den Funktionen aus `db.js` zur Verfügung gestellt. Diese werden anschließend mit `res.send(data)` zurückgegeben.

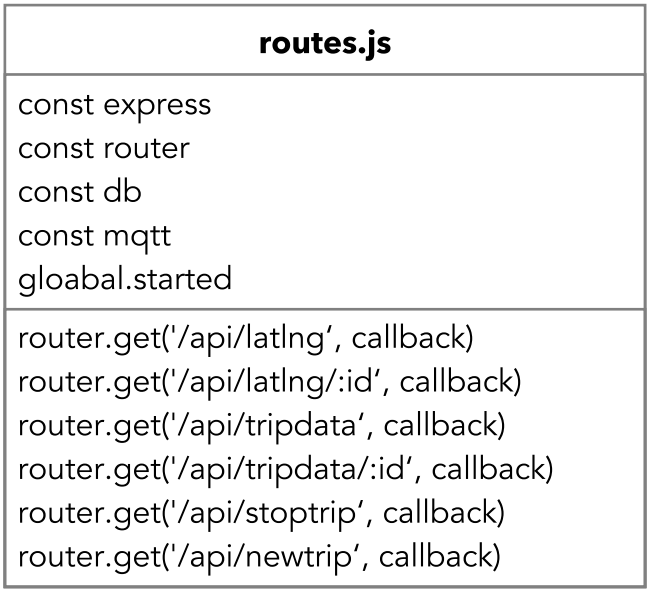

Abbildung 5.5: Klassendiagramm von routes.js

Für Testzwecke lassen sich die Daten durch den Aufruf *localhost:8080/path* des jeweiligen Pfades im Browser anzeigen. Die spätere programmtechnische Einbindung in das Frontend erfolgt mit einer so genannten Factory in Abschnitt 5.2.3.

Die Pfade `/api/newtrip` und `/api/stoptrip` werden durch eine Betätigung des Start- bzw. Stop-Buttons am Touchscreen Display aufgerufen. Dabei wird der Boolean `started` dementsprechend angepasst und entweder ein neuer Trip angelegt oder die Aufnahme der aktuellen Fahrt beendet. Anschließend wird in beiden Fällen der Trip an den MQTT Broker veröffentlicht. Außerdem wird die Benutzeroberfläche durch die WebSocket Verbindung aufgefordert, sich zu aktualisieren.

5.1.4 db.js

Das Datenbankmodul `db.js` dient als Schnittstelle zur MySQL Datenbank und stellt verschiedene Funktionen zum Einfügen, Aktualisieren und Auslesen von Daten zur Verfügung.

Zunächst wird mithilfe von den Zugangsdaten aus dem Konfigurationsmodul eine Verbindung zur internen MySQL Datenbank aufgebaut. Anschließend folgen die Definitionen der in Abbildung 5.6 gelisteten Funktionen. Durch den Zusatz von `exports.Funktionsname` werden die Funktionen zugänglich für die anderen Module gemacht, was mit einer `public` Funktion der Programmiersprache Java zu vergleichen ist.

Bisher wurde für das Einfügen, Aktualisieren oder Auslesen einer Zeile immer eine eigene Anfrage an die Datenbank gestellt. Da jedoch I/O-Operationen wie das Stellen einer Anfrage an die Datenbank mehr Zeit benötigen als normale Operationen innerhalb Node.js, sind die Funktionen nun so konzipiert, dass mehrere Zeilen innerhalb einer Anfrage behandelt werden. Wenn die Serveranwendung beispielsweise alle GPS-Einträge einer bestimmten Fahrt vom Fahrzeug verlangt, wird nicht mehr jeder Eintrag mittels einer `for`-Schleife einzeln ausgelesen, sondern über eine einzige Abfrage bestehend aus mehreren `SELECT` Statements, die mit dem logischen Operator `AND` miteinander Verknüpft sind.

Abbildung 5.6: Klassendiagramm von db.js

5.1.5 mqtt.js

Das Modul `mqtt.js` ist für die Datenübertragung zwischen Fahrzeug- und Serveranwendung zuständig, die über den MQTT Broker läuft. Wie in Abbildung 5.7 wird auch hier zunächst mithilfe der Zugangsdaten aus `config.js` versucht eine Verbindung zum MQTT Broker aufzubauen.

Da jedes Kraftfahrzeug eine Vehicle Identification Number (VIN) besitzt, womit es eindeutig identifiziert werden kann, ist in jeder `config.js` Datei die VIN des Fahrzeuges als Topic vermerkt. Sobald eine Verbindung aufgebaut wurde, wird die Methode `client.on('connect')` aufgerufen und darin das Topic abonniert. Danach wird eine Online-Bestätigung veröffentlicht, die die ID der zuletzt erfassten Fahrt beinhaltet, um beim Server den Synchronisationsvorgang einzuleiten.

Jede Nachricht, die an den MQTT Broker veröffentlicht wird, enthält neben dem Topic noch einen sogenannten *Subtopic*. Dieser dient dazu, den Kontext einer Nachricht klarzustellen, vergleichsweise wie ein Betreff bei E-Mails. Bevor eine Nachricht mit MQTT gesendet werden kann, muss sie zuerst mittels client.on(message) von einem JSON Array in einen ByteBuffer umgewandelt werden. Analog dazu existiert zum Empfang von Nachrichten auch die Funktion `toJSON(data)`, die einen ByteBuffer wieder in einen JSON-Array konvertiert.

Sobald eine MQTT Nachricht in einem abonniertem Topic empfangen wird, wird die Funktion `client.on('message')` ausgelöst. Hier wird anhand des Subtopics klargestellt, um was für eine Nachricht es sich handelt und wie sie behandelt wird. Ist der Betreff der Nachricht `'Users'`, weiß die Anwendung, dass es sich hierbei um Userdaten handelt und aktualisiert die User Tabelle in der Datenbank.

Alle anderen Subtopics und deren Behandlungen dienen der Synchronisation und werden in Abschnitt 5.4 behandelt.

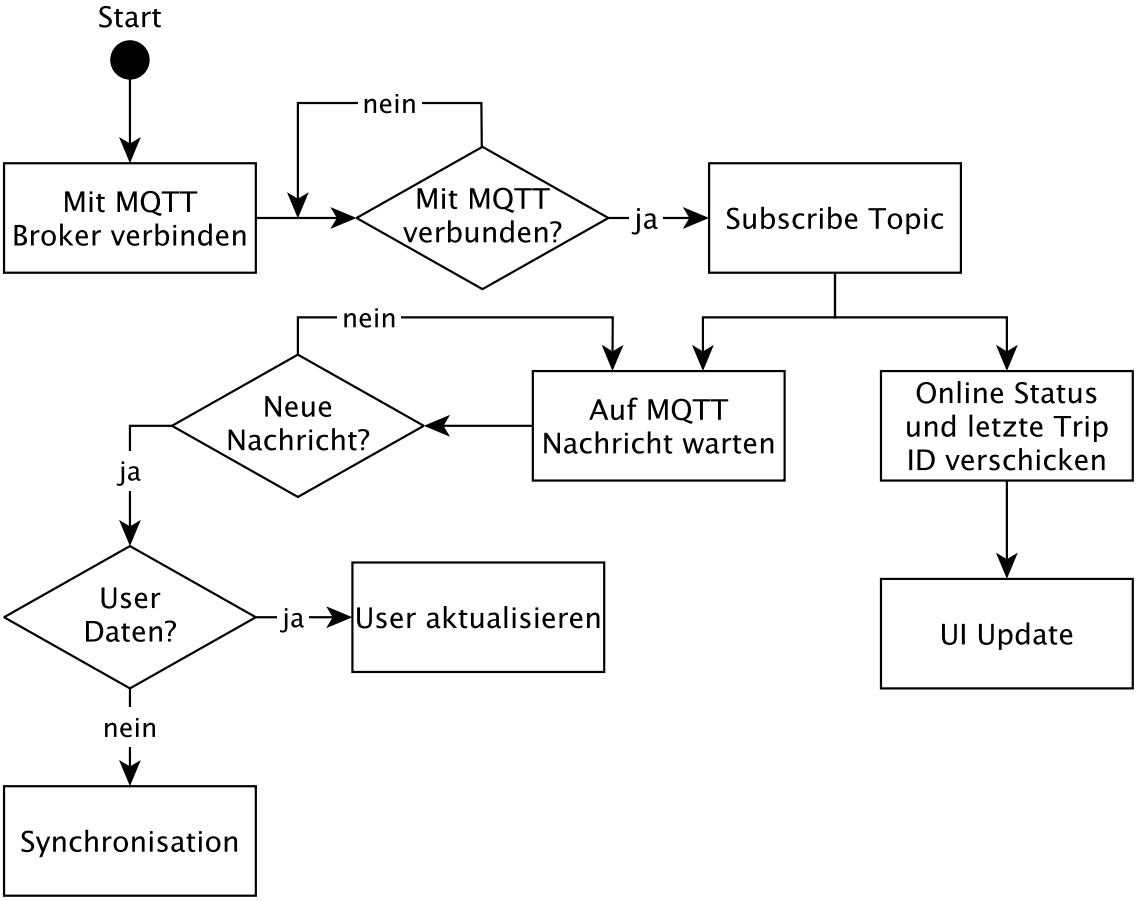

Abbildung 5.7: Ablaufdiagramm mqtt.js (in Anlehnung an [Gro18])

5.1.6 can.js

Das `can.js` Modul liest die Position und den Kilometerstand des Fahrzeuges vom CAN-Bus aus. Dabei wird das Paket *socketcan* geladen, welches dem in Abschnitt 2.3.1 beschriebenen Linux Treiber SocketCAN dient. Der Ablauf des Moduls wird in Abbildung 5.8 grafisch dargestellt.

Zuerst wird versucht durch die Funktion `createRawChannel('can0', true)` einen Kanal zum CAN-Bus zu erstellen. Sollte der CAN-Bus der Anwendung nicht verfügbar sein, wird die Fahrzeuganwendung mit einem Error terminiert, da ohne den CAN-Bus keine Daten zu der Fahrt aufgezeichnet werden können. Falls der Versuch jedoch erfolgreich ist, wird in der Datenbank asynchron ein neuer Trip mit aktueller Startzeit hinterlegt und dieser anschließend an den MQTT Broker veröffentlicht.

Zeitgleich wird in der Callback von `channel.addListener('onMessage', ...)` das Verhalten beim Erhalt von neuen CAN Nachrichten definiert. Im Anschluss wird der Kanal mit `channel.start()` und somit das Abhören des CAN-Busses gestartet.

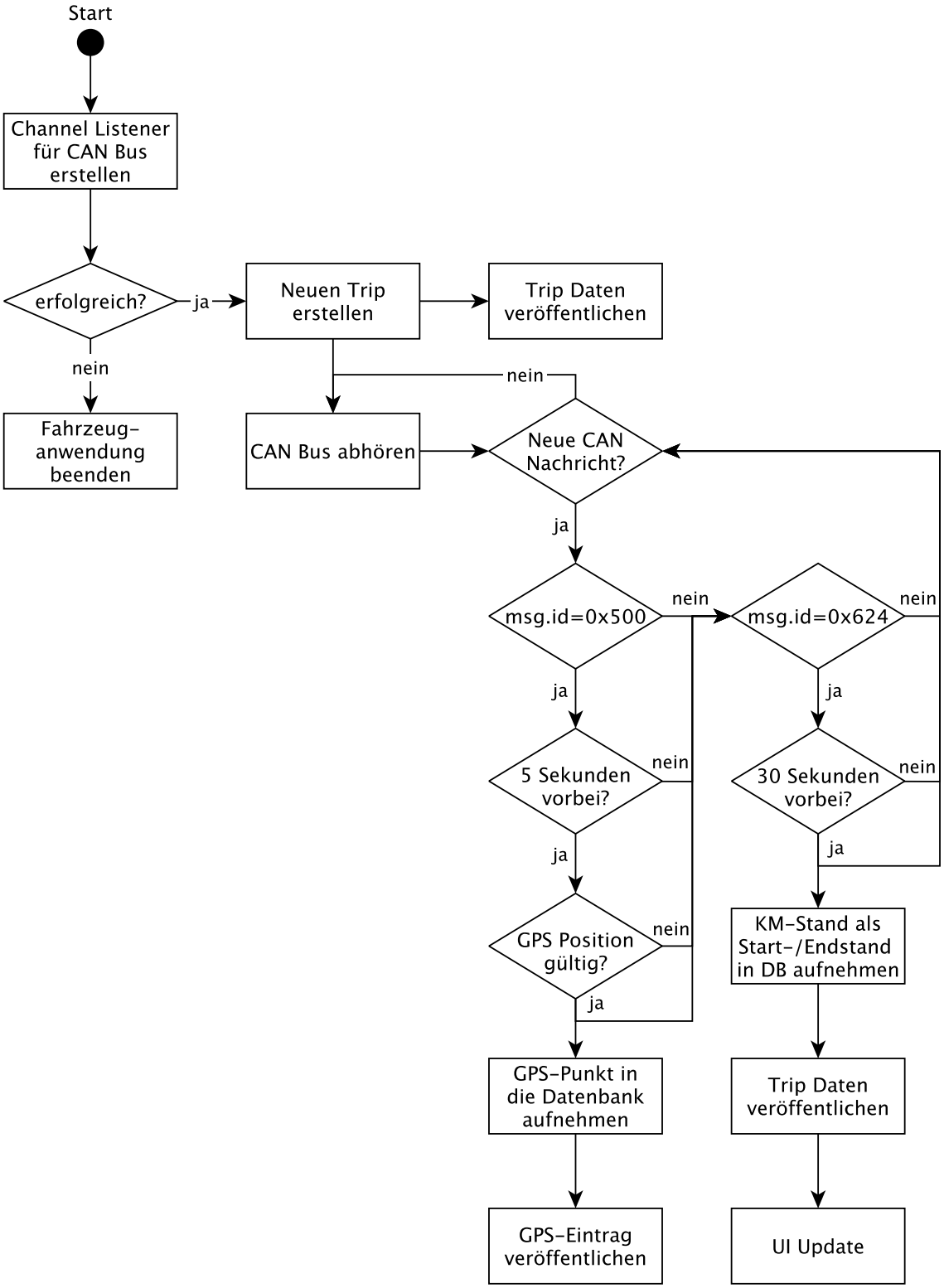

Abbildung 5.8: Ablaufdiagramm can.js

Jede CAN Nachricht ist nach dem Schema aus Listing 5.1 aufgebaut. Dabei geben `ts_sec` und `ts_usec` den Zeitstempel der Nachricht in Sekunden und Mikrosekunden an. Damit kann berechnet werden, ob genug Zeit vergangen ist, um nach 5 Sekunden die Position oder nach 30 Sekunden den Kilometerstand des Fahrzeuges wieder erfassen zu können.

Listing 5.1 Inhalt einer CAN Nachricht

```
msg = {ts_sec:1523895612,
        ts_usec:331082,
        id:1280,
        data:<Buffer d3 00 d6 00 d6 00 00 00>}
```

Da am CAN-Bus eine Vielzahl an verschiedenen Nachrichten anliegen, wird mit `id` gekennzeichnet, um was für eine Nachricht es sich handelt. Die ID für GPS-Koordinaten ist 0x500 und für Kilometerstände 0x624.

Handelt es sich um GPS-Koordinaten, müssen die eigentlich Daten in `data` mithilfe der Node.js Funktion `readUInt32LE(offset)` konvertiert werden, da sie in Form eines ByteBuffers anliegen. Da GPS-Koordinaten aus Längen- und Breitengraden bestehen, muss die Konvertierung jeweils mit dem `offset` 0 für Längen- und 4 für Breitengrad erfolgen. Bei Kilometerständen hingegen findet die Konvertierung durch die Funktion `readUInt32BE(offset)` mit dem `offset` 0 statt.

Sobald eine neue Nachricht am CAN-Bus anliegt, wird zuerst überprüft, ob es sich um GPS-Koordinaten handelt. Falls dies der Fall ist und fünf Sekunden zum letzten Auslesen einer GPS-Koordinate vergangen sind, wird überprüft, ob die Koordinaten legitim sind. Dabei wird die aktuelle Koordinate mit der letzten legitimen GPS-Koordinate verglichen. Unterscheiden sich dabei die beiden Längen- oder Breitengrade um mindestens einen Grad, wird die aktuelle GPS-Koordinate verworfen. Dadurch werden Messungenauigkeiten bis zu einem Grad und ungültige Koordinaten, die bei Verlust des GPS-Empfangs (z.B. in einem Tunnel) am CAN-Bus anliegen, eliminiert.

Sollte die GPS-Koordinate legitim sein, wird sie samt Zeitstempel asynchron in die Datenbank aufgenommen und an den MQTT Broker übermittelt.

Für den Fall, dass die CAN Nachricht den Kilometerstand beinhaltet, wird nach der Konvertierung überprüft, ob bereits ein Kilometerstand für den Start der Fahrt eingetragen wurde. Falls nicht wird der Kilometerstand für den Start in die Datenbank aufgenommen, ansonsten für das Ende. Dabei wird jedes mal mit einem aktuellen Zeitstempel die Endzeit der Fahrt aktualisiert, obwohl die Fahrt noch nicht zu Ende ist. Grund hierfür ist, weil beim alten Stand der Anwendung bei Abstürzen keine Endzeit vorhanden war, da sie immer nur am Ende der Fahrt eingetragen wurde. Durch diese Maßnahme wird bei einem Absturz eine Endzeit hinterlegt, die höchstens 30 Sekunden vor dem Absturz stattgefunden hat und gewährleistet, dass jede Fahrt, in der es dazu kam, dass der Kilometerstand ausgelesen wurde, eine Endzeit besitzt.

Nachdem Kilometerstand und Endzeit in die Datenbank aufgenommen wurden, werden die Trip Daten im Anschluss an den MQTT Broker gesendet und das Frontend aufgefordert, die Benutzeroberfläche zu aktualisieren.

5.2 Fahrzeug-Frontend

Die Aufgabe des Fahrzeug-Frontends ist es, dem Fahrer eine Benutzeroberfläche auf dem Touchscreen Display des Raspberry Pi anzuzeigen. Dabei entsteht die Gestaltung der Oberfläche mithilfe des Angular Frameworks. Die hierfür benötigten Daten der Fahrt werden mithilfe von Express vom Fahrzeug-Backend geholt. Durch die zusätzliche WebSocket Verbindung wird dem Backend ermöglicht, sich mit dem Frontend zu verständigen und somit Einfluss auf die Benutzeroberfläche zu haben.

5.2.1 index.html

Die Benutzeroberfläche wird in der Datei `index.html` beschrieben. Dabei werden im `<head>`-Bereich weitere für die Darstellung benötigten Dateien wie das Angular Framework definiert. Im `<body>`-Bereich hingegen befinden sich die in der Benutzeroberfläche dargestellten HTML Elemente.

Bootstrap

Bei den Dateien `bootstrap.min.css` und `bootstrap.min.js` handelt es sich um Gestaltungsvorlagen für HTML Elemente, durch deren Verwendung der Aufwand für die Erstellung einer benutzerfreundlichen Webseite minimiert wird.

style.css

Die Cascading Style Sheets (CSS)-Datei `style.css` enthält alle benutzerdefinierten Style Vorgaben der verwendeten Elemente und passt Bootstrap auf die notwendigen Vorgaben an.

socket.io.js

Die Datei `socket.io.js` ist die clientseitige Einbindung der bereits beschriebenen WebSocket Schnittstelle zum Fahrzeug-Backend.

5.2.2 map.html

Diese HTML Datei wurde speziell für die Karte auf der rechten Hälfte der Benutzeroberfläche erstellt. Zuvor war die Karte in `index.html` platziert. Um jedoch die Karte separat aktualisieren zu können, wird `map.html` als Frame in `index.html` eingebettet.

Openlayers

Zur Darstellung der Karte wird die JavaScript Bibliothek *Openlayers* verwendet. Hierzu sind die beiden Dateien `ol.css` und `ol.js` im `<head>`-Bereich einzubinden. Die Konfiguration der Bibliothek erfolgt in app.js (Abschnitt 5.2.4).

5.2.3 factory.js

Mit Hilfe dieser Factory werden bestimmte Funktionen des Fahrzeug-Backends in die Angular App ausgelagert. Dabei erstellt die Factory einen sogenannten `_dataService`, in der verschiedene Funktionen angegeben sind, die die Anwendungsendpunkte des Moduls `routes.js` mit einer `HTTP GET` Anfrage aufruft. Im Anschluss werden die erhaltenen Rückgabewerte an die Angular App weitergeleitet.

5.2.4 app.js

In der JavaScript Datei `app.js` wird ein Controller zum Laden und zur Steuerung der Elemente der Benutzeroberfläche implementiert und das Angular Framework erweitert.

Zunächst wird mit `angular.module` ein neues Angular Modul erstellt. Um vollen Zugriff auf die Inhalte der Benutzeroberfläche zu erlangen, wird durch das Einbinden mit `<body ng-controller="mainController">` in beiden HTML Dateien und aufrufen der `logbook.controller` Funktion in `app.js` ein Controller definiert.

Über `$scope.nameElement` bekommen die Labels in der linken Hälfte der Benutzeroberfläche zunächst einen Initialtext. Die Funktion `updateData()` beschafft sich über die Anwendungsendpunkte des Fahrzeug-Backends die benötigten Daten für die Anzeige aus der Datenbank und aktualisiert die Labels. Dabei sollen Endzeit und Endkilometerstand nur am Ende einer Fahrt angezeigt werden. Durch `updateMap()` hingegen werden für die Karte die benötigten Daten geholt, wodurch die aktuelle Route eingezeichnet wird.

Eingehende Nachrichten der WebSocket Verbindung werden je nach Kontext über die Funktionen `socket.on(topic, ...)` empfangen. Innerhalb der Callbacks wird das Verhalten im Anschluss definiert. Bei einer Aufforderung des Fahrzeug-Backend

die Benutzeroberfläche zu aktualisieren, werden die HTML Dateien und somit auch `app.js` zusammen mit den anderen Komponenten aus dem `<head>`-Bereich neu geladen. Aus diesem Grund ist es `app.js` nicht möglich, sich Zustände dauerhaft zu merken, da der Controller und sämtliche Variablen in diesem Vorgang neu initialisiert werden.

In der bisherigen Applikation wurde das Ende einer Fahrt durch den Datenbankeintrag der Endzeit der Fahrt erkannt. Wie bereits in Abschnitt 5.1.6 erwähnt, besitzt die Fahrt von nun an nicht nur am Ende eine Endzeit, sondern auch schon währenddessen. Dies führte dazu, dass Endzeit und Endkilometerstand auch während der Fahrt angezeigt wurden. Da entweder der Start- oder der Stop-Button deaktiviert sein muss, wurde aufgrund dieses Problems oft der falsche Button ausgeschalten. Um dieses Problem zu lösen, reicht es nicht einen Boolean in `app.js` anzulegen, der sich merkt, ob gerade eine Fahrt aufgezeichnet wird, da, wie bereits erwähnt, `app.js` sich nichts dauerhaft merken kann.

Aus diesem Grund kommt hier die WebSocket Verbindung ins Spiel. Sowohl Beginn als auch Ende der Aufzeichnung einer Fahrt wird vom Backend ermittelt. Zu Beginn der Fahrzeuganwendung wird bereits die Fahrt automatisch aufgezeichnet. Fahrtenden werden entweder über den GPIO Pin oder über die Betätigung des Stop-Buttons registriert. In beiden Fällen wird das Frontend zur Aktualisierung der Benutzeroberfläche aufgefordert. Beim Starten der Aufzeichnung über den Start-Button wird ebenfalls aktualisiert. Deshalb wird jedes mal wenn `app.js` geladen wird, über die WebSocket Verbindung mit dem Backend kommuniziert, um den aktuellen Status der Fahrt zu erfahren und die Benutzeroberfläche dementsprechend zu gestalten.

5.3 Server-Backend

Der Server-Backend besteht aus zwei unterschiedlichen Anwendungen. Die Anwendung `index.js` läuft dauerhaft auf dem Server und dient dazu, die erfassten Daten der Fahrzeuganwendung mit den Daten von Nominatim zu ergänzen und in die Datenbank aufzunehmen. In einem separaten Prozess wird täglich eine Überprüfung und gegebenenfalls eine Korrektur der Daten vollzogen.

Die Anwendung `excel.js` hingegen ist dann aufzurufen, wenn ausgewählte Fahrten in eine Excel-Datei exportiert werden sollen. Dabei werden die Daten der ausgewählten Fahrten aus der Datenbank gelesen und die Routenbeschreibungen generiert. Im Anschluss werden die Daten in eine Excel Tabelle mit vordefiniertem Layout eingetragen.

Aus Abbildung 5.9 sind die verwendeten Module zu entnehmen. Eine Übersicht über den gesamten Ablauf des Server-Backends wird in Abbildung 5.10 auf Seite 57 veranschaulicht.

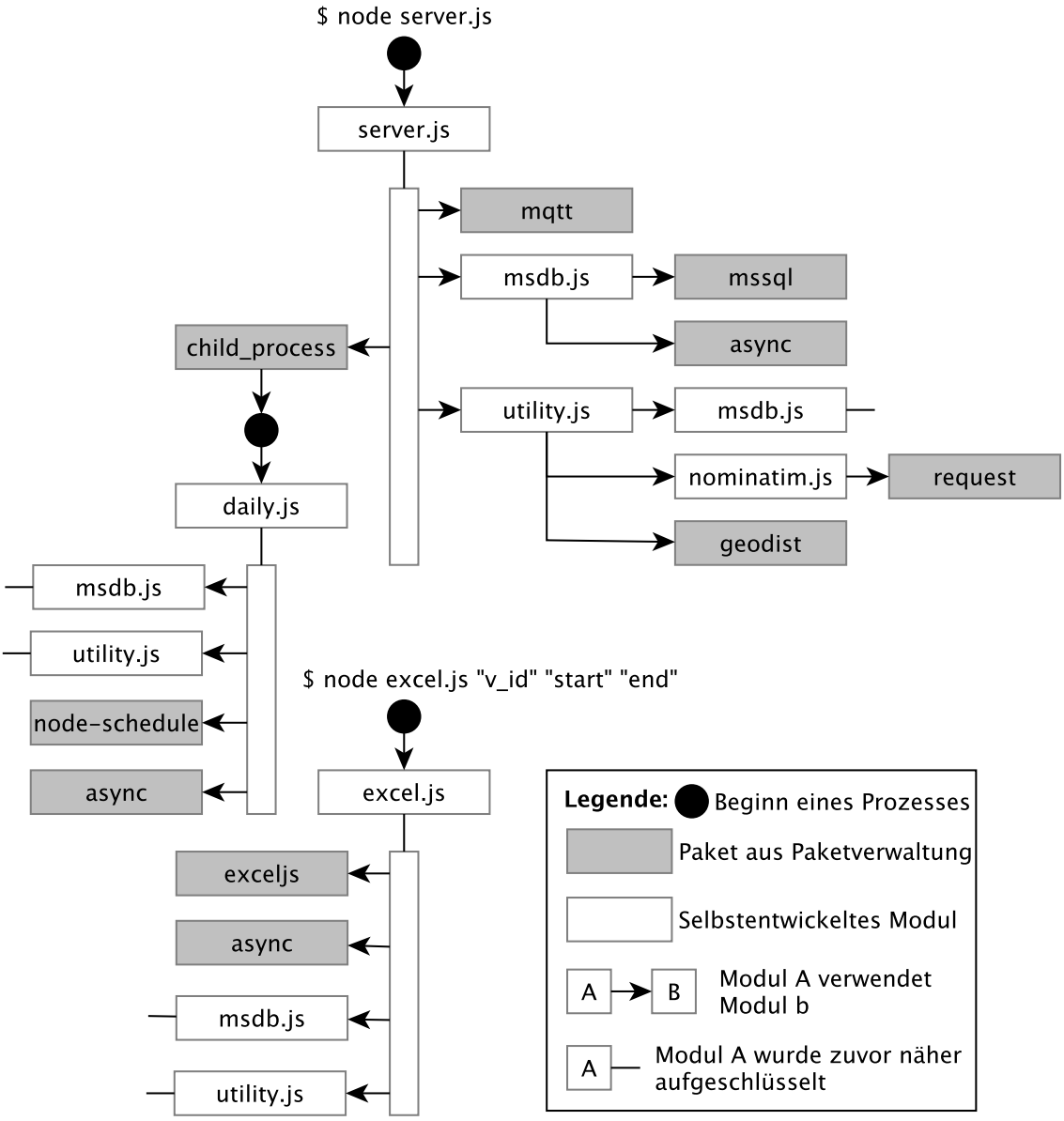

Abbildung 5.9: Aufbau der Serveranwendung (in Anlehnung an [Gro18])

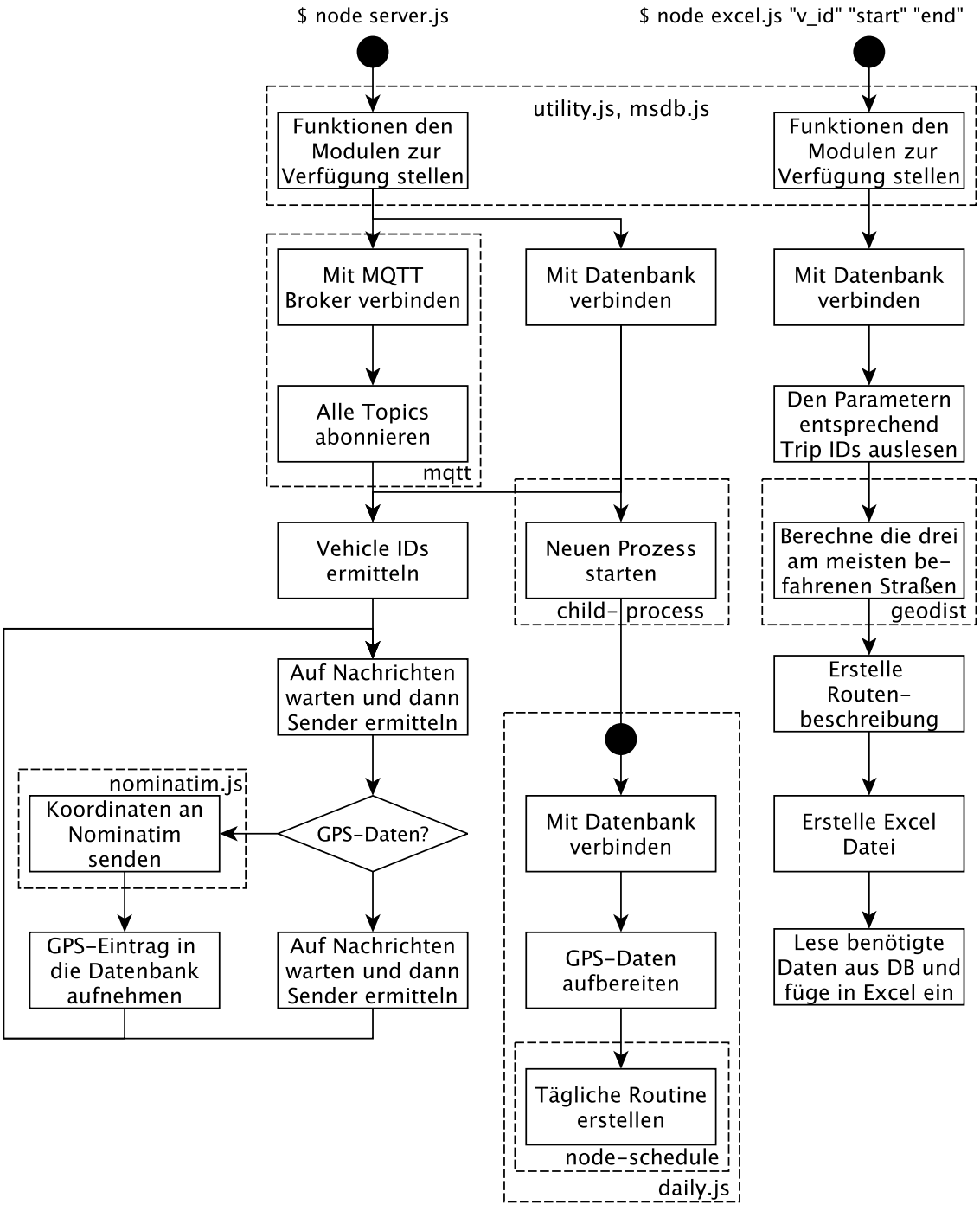

Abbildung 5.10: Grobe Übersicht über den Ablauf der Serveranwendung

5.3.1 Datenbank

Die von den Fahrzeugen erfassten Daten werden alle in einer Microsoft SQL Datenbank gesammelt. Darüber hinaus befinden sich in der Datenbank noch weitere Daten, die für die Fahrtenbücher benötigt werden, wie zum Beispiel Kennzeichen, Marke und Modell der Fahrzeuge. Diese Daten standen schon bereits vor dieser Arbeit seitens des IVK zur Verfügung.

Im folgenden werden die für die Serveranwendung relevanten Tabellen dieser Datenbank beschrieben, wie sie aufgebaut sind und welche Eigenschaften ihre Spalten ausweisen.

Users

Die Users Tabelle der Serverdatenbank ist ähnlich aufgebaut wie im Fahrzeug aus Abschnitt 5.1.1. Alle Spalten, die sich in der Tabelle der MySQL Datenbank befinden, sind auch hier vorhanden. Darüber hinaus sind hier noch zusätzliche Daten zu den Fahrern, wie zum Beispiel Adresse und Geburtsdatum, enthalten, die ebenfalls für das Fahrtenbuch benötigt werden.

TripStatus

Diese Tabelle beinhaltet vier verschiedene Status, die ein Trip in der Datenbank haben kann. Jeder Trip, der beim Server ankommt und in die Datenbank aufgenommen wird, durchläuft jeden Status von 0 bis 3. Dies sind die möglichen Status und deren IDs:

- 0: Missing Trip
- 1: Missing GPS
- 2: Complete
- 3: Finished

Ein Trip, der den Status „Missing Trip“ besitzt, hat eine Lücke innerhalb der Trips Tabelle. Für die Serveranwendung heißt das, dass diesem Eintrag noch Daten für die Trips Tabelle fehlen und diese beim richtigen Fahrzeug angefragt werden müssen.

Für den Fall, dass ein Trip mit dem Status „Missing GPS“ versehen ist, weiß die Serveranwendung, dass diesem Trip keine Tripdaten, aber jedoch GPS-Einträge fehlen.

Erst wenn alle Daten zu einer Fahrt beim Server anliegen, wird der Trip mit der Status ID 2 (Complete) und somit als vollständig angekommen gekennzeichnet.

Der Status „Finished“ wird für die Trips verwendet, deren GPS-Daten durch die tägliche Routine aufbereitet wurden. Erst wenn ein Trip den Status 3 erlangt, zählt er als fertig und ist in der Lage in die Excel-Datei exportiert und somit in das Fahrtenbuch aufgenommen zu werden.

Trips

In Tabelle 5.4 wird der Aufbau der Trips Tabelle der Serverdatenbank dargestellt. Neben den Spalten, die ebenfalls in der Fahrzeugdatenbank vorkommen, sind hier noch zusätzliche Spalten vorhanden.

Beim erstellen eines Trips wird der aktuelle Zeitstempel in die Spalten `created_at` und `updated_at` geschrieben. Für jede Änderung der Trip Daten einer Fahrt, wird `updated_at` aktualisiert.

In den Spalten `start_time`, `end_time`, `start_km`, `end_km` und `user_id` werden die Daten aus den Trips Tabellen der Fahrzeuge eingetragen.

Der Fremdschlüssel `status_id` ist im Bezug auf die TripStatus Tabelle und dient als Status für die Trips.

Die Spalte `vehicle_id` repräsentiert die ID eines Fahrzeuges. Dabei handelt es sich nicht um die VIN, sondern um die Fahrzeug ID innerhalb einer *Vehicles* Tabelle, in der Daten zu den Fahrzeugen hinterlegt sind. Aus diesem Grund agiert `vehicle_id` hier als Fremdschlüssel.

Da `id` benötigt wird, um alle Trips zu nummerieren, werden die IDs der Fahrten aus der Fahrzeugdatenbank in dieser Tabelle nicht in `id` gespeichert, sondern in `vehicle_trip_id`. Damit lässt sich rekonstruieren, welche Trip ID des Fahrzeuges durch die Trip ID des Servers darstellt wird, um bei der Synchronisation gezielt Daten beschaffen werden können.

Obwohl alle Fahrzeuge ihre Trips von eins aufzählend nummerieren, sind `vehicle_id` und `vehicle_trip_id` als Paar einzigartig. Durch das Setzen auf Unique verhindert die Datenbank, dass es zu doppelten Einträgen kommen kann.

Die Routenbeschreibung für einen Trip wird in der Spalte `description` hinterlegt. Da zu Beginn noch keine Routenbeschreibung existiert, ist diese Spalte in der Lage `null` Werte anzunehmen.

GPSLookups

Bei der Serverdatenbank wird die GPSLookups Tabelle um die Spalten `created_at`, `street`, `city` und `vehicle_gps_id` erweitert.

Tabelle 5.4: Spalteneigenschaften der Tabelle Trips beim Server

Spalte	Datentyp	Nullable	Autoincrement	Unique	Schlüssel
id	int	nein	ja	ja	primär
created_at	datetime	nein	nein	nein	-
updated_at	datetime	nein	nein	nein	-
description	varchar	ja	nein	nein	-
start_time	datetime	nein	nein	nein	-
end_time	datetime	nein	nein	nein	-
start_km	int	nein	nein	nein	-
end_km	int	nein	nein	nein	-
user_id	int	nein	nein	nein	fremd
vehicle_id	int	nein	nein	} als Paar	fremd
vehicle_trip_id	int	nein	nein		fremd
status_id	int	nein	nein	nein	fremd

Die Spalten `created_at`, `id` und `vehicle_gps_id` werden analog zu der Trips Tabelle verwendet.

Die durch das Reverse Geocoding ermittelten Straßen und Städte werden in `street` und `city` gespeichert. Da es bei einer Anfrage an Nominatim zu einem Misserfolg kommen kann, sind beide Spalten in der Lage den Wert `null` zu akzeptieren.

Weil in der Trips Tabelle in der Datenbank des Server die Trip IDs von den Trip IDs der Fahrzeuge unterschieden werden, ändert sich der Wert in der Spalte `trip_id`, da sie nun eine Referenz an die Spalte ID aus der Trip Tabelle des Servers ist.

Aus dem selben Grund wie bei Trips, sind hier `trip_id` und `vehicle_gps_id`, ebenfalls als Paar betrachtet, einzigartig.

Tabelle 5.5: Spalteneigenschaften der Tabelle GPSLookups beim Server

Spalte	Datentyp	Nullable	Autoincrement	Unique	Schlüssel
id	int	nein	ja	ja	primär
created_at	datetime	nein	nein	nein	-
latitude	nchar	nein	nein	nein	-
longitude	nchar	nein	nein	nein	-
street	varchar	ja	nein	nein	-
city	varchar	ja	nein	nein	-
trip_id	int	nein	nein	} als Paar	fremd
vehicle_gps_id	int	nein	nein		fremd

5.3.2 server.js

Die Serveranwendung wird durch dieses Modul mittels `$ node server.js` gestartet. Zunächst werden die für den Betrieb benötigten Module geladen. Aufgabe dieses Moduls ist es, sich mit dem MQTT Broker und der MS SQL Datenbank zu verbinden. Im Anschluss wird das Verhalten für eingehende MQTT Nachrichten definiert. Aus diesem Grund ist `server.js` ähnlich wie das Fahrzeugmodul `mqtt.js` aufgebaut und agiert somit als der serverseitige Teil der Synchronisation. Genaueres über die Synchronisation wird in Abschnitt 5.4 behandelt.

Zur Unterstützung von mehreren Fahrzeugen stehen der Serveranwendung für jedes Fahrzeug jeweils ein Topic in Form des VINs des Fahrzeuges zur Verfügung. Nach erfolgreichem Verbindungsaufbau zum MQTT Broker gilt es alle Topics zu abonnieren. Außerdem wird anhand der VIN im Topic die Vehicle ID des Fahrzeuges ermittelt, um empfangene Daten mit der richtigen Fahrzeug ID in die Datenbank aufnehmen zu können.

Da sich der Sender beim Empfangen einer Nachricht durch den Topic zurückverfolgen lässt, wird jede Antwort, die für eine Nachricht benötigt wird, mit dem selben Topic veröffentlicht. Dadurch ist gewährleistet, dass der Server immer mit dem richtigen Fahrzeug kommuniziert.

Beim Empfangen der Online-Benachrichtigung eines Fahrzeuges antwortet der Server mit den Userdaten der Fahrer und weiß, dass das Fahrzeug die Synchronisation einleiten möchte.

Wird eine Nachricht mit dem Subtopic `'updateTrip'` erhalten, weiß die Serveranwendung, dass es sich hier um Trip Daten der aktuellen Fahrt handelt und nimmt diese mit der richtigen Fahrzeug ID in die Datenbank auf.

GPS-Daten der aktuellen Fahrt hingegen haben den Subtopic `'GPS'`. Die in der Nachricht enthaltenen Koordinaten werden an Nominatim gesendet. Erhält die Serveranwendung innerhalb eines Timeouts von fünf Sekunden eine Antwort, wird diese zusammen mit den GPS-Daten des Fahrzeugs in die Datenbank aufgenommen. Falls der Timeout überschritten wird erfolgt die Aufnahme mit dem Wert 0 in den Spalten `street` und `city`.

5.3.3 msdb.js

Das Modul `msdb.js` ist für dasselbe zuständig wie das andere Datenbankmodul `db.js` im Fahrzeug aus Abschnitt 5.1.4 mit dem Unterschied, dass hier das Packet `mssql` verwendet wird, da beim Server eine Microsoft SQL Datenbank im Einsatz ist.

Die Funktionen, die von dem Modul `msdb.js` zur Verfügung gestellt werden, sind in dem Klassendiagramm aus Abbildung 5.11 aufgelistet. Für die Erweiterung zur Unterstützung von mehreren Fahrzeugen wurden alle nötigen Funktionen um den Parameter `v_id` erweitert.

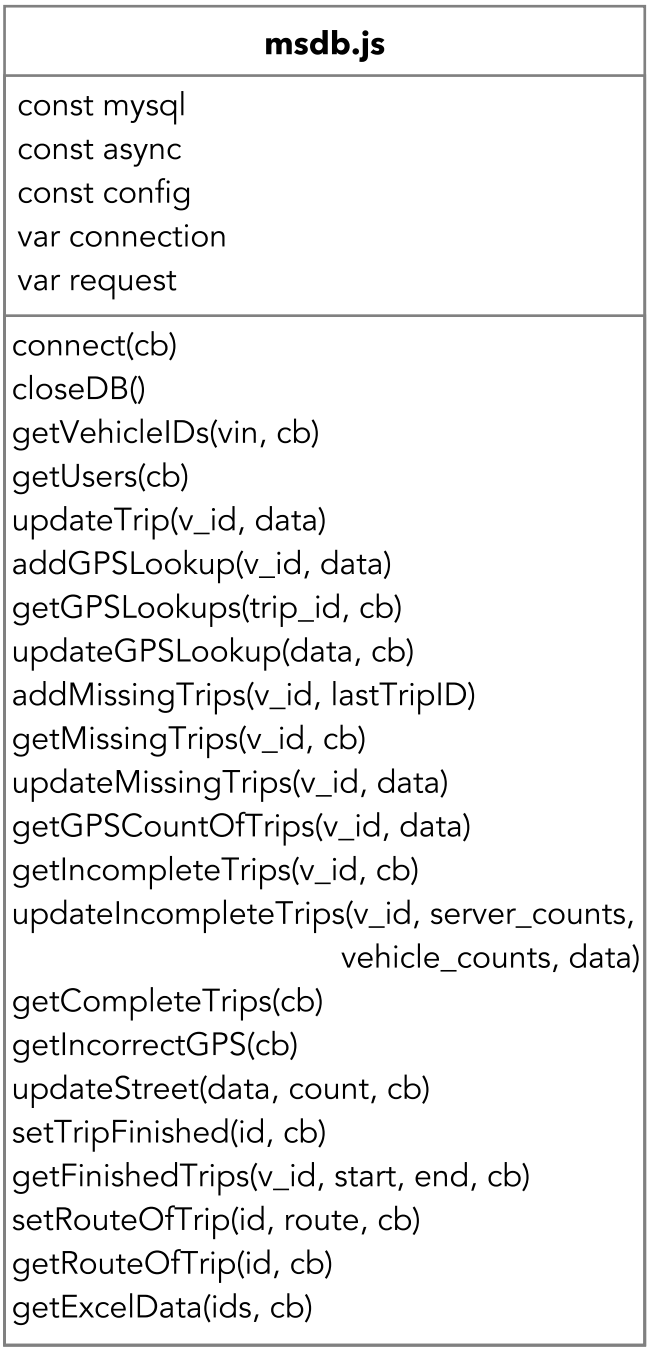

Abbildung 5.11: Klassendiagramm von msdb.js

5.3.4 nominatim.js

Die einzige Aufgabe dieses Moduls ist es, das gleichnamige Werkzeug Nominatim von OSM zum Einsatz zu bringen. Hierfür wird mit dem vorinstallierten Packet *request* eine `HTTP GET` Anfrage an folgende Adresse geschickt:

```
http://nominatim.openstreetmap.org/reverse?format=jsonv2&lat=48&lon=9
```

Das `reverse` innerhalb des Uniform Resource Locators (URL) wählt den Reverse Geocoding Dienst von Nominatim aus. Für die Bearbeitung müssen dem Link Breiten- und Längengrad über die Parameter `lat` und `lon` übergeben werden. Durch `format=jsonv2` werden die zu der Koordinate gehörenden OSM Daten als Antwort im JSON Format übermittelt.

Als Timeout für die Anfrage sind fünf Sekunden festgelegt. Sollte innerhalb dieser Zeitspanne keine Antwort vom Nominatim Server ankommen, wird ein Error geworfen, der jedoch nicht zu einem Absturz der Anwendung führt.

5.3.5 utility.js

In diesem Modul sind die Funktionen aus Abbildung 5.12 definiert, die den anderen Modulen zur Verfügung gestellt werden.

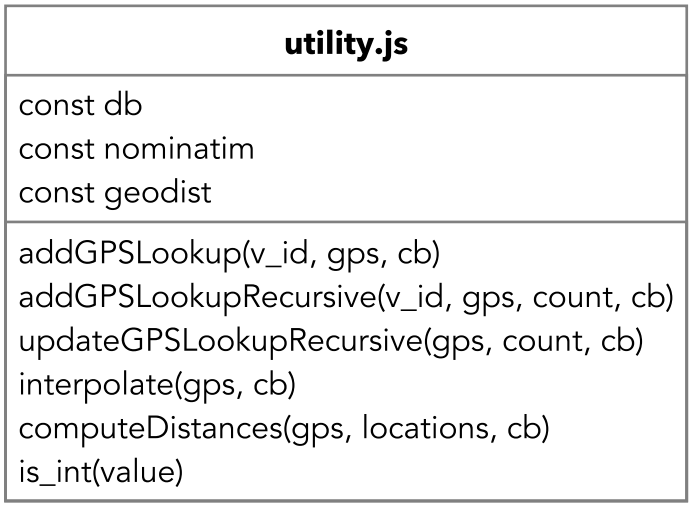

Abbildung 5.12: Klassendiagramm von utility.js

Die Funktionen `addGPSLookup`, `addGPSLookupRecursive` und `updateGPSLookup` senden mithilfe des `nominatim.js` Moduls eine Reverse Geocoding Anfrage an den Nominatim Server und fügen die Antwort in die Datenbank ein.

Die GPS-Koordinaten, die die Serveranwendung alle 5 Sekunden von einem Fahrzeug erhält, werden durch die Funktion `addGPSLookup` bearbeitet.

Durch die Synchronisation, die bei jedem Start der Fahrzeuganwendung zu Stande kommt, werden mehrere GPS-Daten auf einem empfangen. Der erste Gedankte wäre `addGPSLookup` mehrmals innerhalb einer `for`-Schleife aufzurufen. Dabei entsteht jedoch ein Problem beim Event Loop von Node.js, der eingehende Events managt und somit den Programmablauf steuert. Bei einer `for`-Schleife ist der Event Loop in erster Linie damit beschäftigt, die Schleife durchzulaufen, was zu Komplikationen bezüglich der Asynchronität führt. [Stov]

In so einem Fall kommen rekursive Funktionen wie `addGPSLookupRecursive` und `updateGPSLookupRecursive` ins Spiel. Da Nominatims Usage Policy nur eine Anfrage pro Sekunde erlaubt, wird ein Mechanismus implementiert, der nach jeder Anfrage an Nominatim asynchron eine Sekunde wartet. Da dies innerhalb eines Callbacks der Funktion von `nominatim.js` geschehen muss, würde eine `for`- Schleife sofort alle Anfragen an Nominatim senden und anschließend erst in den Callbacks anfangen jeweils eine Sekunde zu warten. Darüber hinaus könnte es währenddessen andere Abläufe der Anwendung blockieren, die parallel laufen, da sie ihre Callbacks nicht ausführen können, weil der Hauptprozess beschäftigt ist. Um dieses Problem zu entgehen wird anstelle von einer `for`-Schleife Rekursion angewandt, in der sich die Funktion innerhalb seines Callbacks selbst aufruft. Dadurch ist gewährleistet, dass die Anfragen an Nominatim untereinander sequenziell mit einer Sekunde Wartezeit stattfinden, aber im Gesamten asynchron zum Programmverlauf beziehungsweise Hauptprozess sind.

Wie bereits in Abschnitt 4.2 erwähnt, versehen Freiwillige Koordinaten mit Zusatzinformationen, wie z.B. Straße oder Stadt, bevor die Daten in der Geodatenbank von OSM hinterlegt wird. Aus diesem Grund sind Straßen von einigen Freiwilligen in `path` und von anderen in `road` gespeichert. Es kommt auch zu Fällen, in denen die Straße nur in `address27` oder `junction` vorkommen. Das selbe Problem ergibt sich auch für die Städte. Sie wurden von einigen Freiwilligen in `city` oder `town` eingetragen. In einigen Fällen kommt die Stadt nur in `county` vor.

Da diese Daten unter willkürlichen Namen gespeichert sind, lässt sich nicht immer herausfinden, worunter die benötigten Daten eingetragen sind. Dieses Problem wurde in der bisherigen Applikation nicht erkannt, was dazu führte, dass die Straßen nur aus `road` und Städte nur aus `city` ausgelesen wurden. Aus diesem Grund kam es zu vielen Lücken in der Serverdatenbank.

Um mit diesem Problem bestmöglich auszukommen, lesen die drei oben genannten Funktionen die Straßen und die Städte nicht nur in `road` und `city` aus, sondern überprüfen ebenfalls die anderen bekannten Felder.

5.3.6 daily.js

Für die Aufbereitung der GPS-Daten wird eine tägliche Routine entworfen. Das Modul `server.js` erstellt zum Beginn der Serveranwendung einen Unterprozess mithilfe des vorinstallierten Pakets *child_process*. Durch die Funktion `fork('daily.js')` wird der Prozess gestartet und `daily.js` als Einstiegsmodul deklariert.

Da `daily.js` sich in einem separaten Prozess befindet, wird, nach dem Laden der benötigten Module, eine weitere Verbindung zur Datenbank aufgebaut. Die Aufbereitung der GPS-Daten findet ein mal zum Beginn der Serveranwendung und täglich als Routine um 1 Uhr statt.

Um eine tägliche Routine einzusetzen, wird das Paket *node-schedule* verwendet. Damit lassen sich zeitbasierte Ausführungen von Prozessen implementieren, um wiederkehrende Aufgaben zu automatisieren. Dabei werden vom Prozess jedoch keine sogenannten *Cronjobs* erstellen, da diese auch nach Terminierung des Prozesses weiterhin bestehen. Deshalb ist node-schedule nur auf zeitbasierte Ausführungen, die prozessintern sind, ausgelegt. [NPM]

Mit der Funktion `schedule.scheduleJob('* * * * * *', ...)` lässt sich die Routine erstellen. Im Callback bekommt die Funktion die eigentliche Aufgabe mitgegeben, die periodisch ablaufen soll. Anhand des ersten Parameters wird durch einen String im Cron Format der Funktion mitgeteilt, wann die Ausführung stattfinden soll. In Abbildung 5.13 wird das Cron Format veranschaulicht. Daraus ergibt sich die Konfiguration `'0 0 1 * * *'`, um die tägliche Routine auf 1 Uhr zu setzen. [NPM]

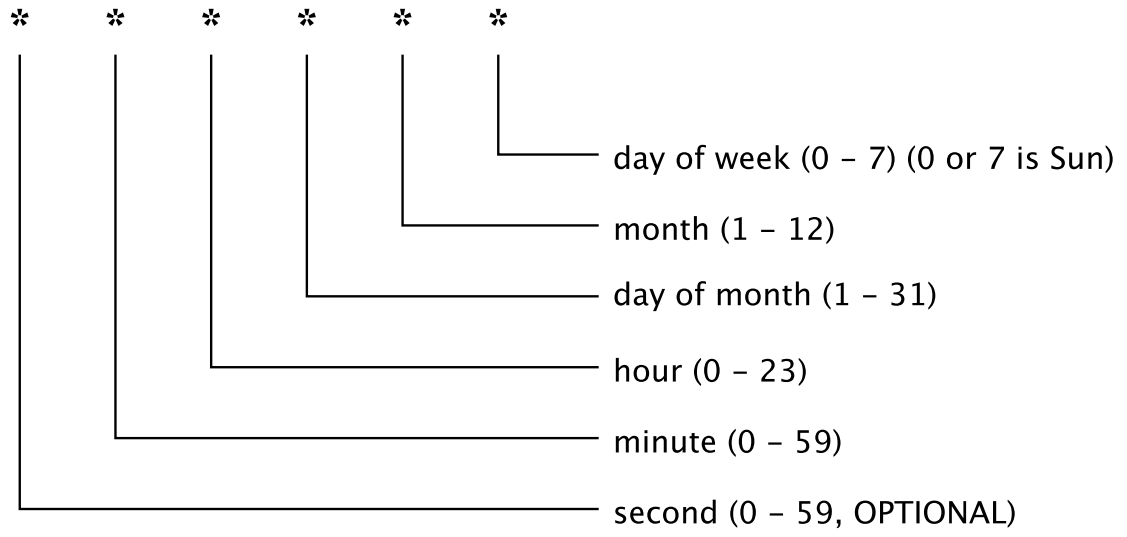

Abbildung 5.13: Cron Format [NPM]

Bei der Aufbereitung wird auschließlich mit GPS-Daten von Trips gearbeitet, die den Status „Completed“ besitzen. Hierzu werden aus der Datenbank zuerst die Koordinaten der GPS-Einträge, die keine Straße oder Stadt besitzen, ausgelesen, um diese erneut bei Nominatim anzufragen. Sollte es zu einem Error bei der Anfrage kommen, wird der Vorgang so lange wiederholt, bis eine Antwort erhalten wird. Aufgrund der Usage Policy von Nominatim wird auch hier zwischen den Anfragen jeweils eine Sekunde gewartet. Sollte in der Antwort immer noch die Straße oder die Stadt fehlen, wird dieser GPS-Eintrag aus der Datenbank gelöscht, ansonsten aktualisiert. Am Ende des Vorgangs besitzt jeder komplette Trip nur noch Einträge, die sowohl Straße und Stadt vorweisen. Erst dann kann die Interpolation beginnen.

In Abschnitt 5.3.5 wurde erläutert wie es zu Problemen mit der Asynchronität bei `for`-Schleifen kommen kann, da in Abhängigkeit von der Anzahl an GPS-Koordinaten asynchron Anfragen an den Nominatim Server stattfinden sollen. Diese Anfragen sind gemeinsam asynchron zum Hauptverlauf des Prozesses, aber untereinander, aufgrund der Usage Policy von Nominatim, sequentiell. Die Interpolation hingegen findet, ebenso wie die Anfragen, in Abhängigkeit von einer Anzahl und asynchron zum Hauptverlauf Prozesses statt. Der Unterschied dabei ist jedoch, dass die Interpolationen untereinander auch asynchron verlaufen sollen.

Hierzu wird die Funktion `forEachOf(array,function(value,key,cb),cb)` aus dem vorinstallierten Paket *async* verwendet. Diese besondere `for`-Schleife ermöglicht es, Anweisungen asynchron zueinander zu bearbeiten und, im Gegensatz zur normalen `for`-Schleife, die Callback direkt im Anschluss auszuführen. Eine Beispielanwendung für diese Funktion befindet sich in Listing 5.2. Hier beinhaltet der erste Parameter ein Array oder ein Objekt mit Dateipfaden, worüber iteriert werden soll. Der zweite Parameter ist eine Funktion, die selber als Parameter den jeweiligen Wert und Zähler der aktuellen Iteration besitzt. Darin wird definiert, dass in die Dateien aus dem Array asynchron geschrieben werden soll. Am Ende des Callbacks der asynchronen Funktion `fs.writeFile` wird der Callback des zweiten Parameters von `forEachOf` ausgeführt. Dadurch wird `forEachOf` signalisiert, dass die jeweilige Iteration in seine Datei asynchron geschrieben und seine Konsolenausgabe abgeschlossen hat. Erst wenn alle Iterationen ihre Callbacks vollständig ausgeführt haben, wird die Callback von `forEachOf` ausgelöst.

Listing 5.2 Asynchrone for-Schleife

```
async.forEachOf(array, function (value, key, callback) {
        fs.writeFile(__dirname + value, 'Hello World!', 'utf8', function () {
                let count = key + 1;
                console.log('Iteration ' + count + ' finished.');
                callback();
        });
}, function () {
        console.log('All iterations finished.' );
});
```

Für die asynchrone Ausführung der Interpolation werden der Funktion `forEachOf` IDs von allen kompletten Trips zum Iterieren übergeben. Bei der Interpolation werden zunächst alle GPS-Einträge eines Trips aus der Datenbank ausgelesen. Als nächstes wird nach Einträgen gesucht, bei denen, aufgrund der GPS-Ungenauigkeit, beim Überqueren von Kreuzungen eine Straße erfasst wurde, die nicht befahren worden ist. Dabei werden die Vorkommnisse von Straßen, die nur ein mal hintereinander vorkommen, mit dem Straßennamen des vorherigen Eintrages interpoliert. Sobald die Änderungen in die Datenbank aufgenommen wurden, werden die Trips mit dem Status 3 („Finished“) gekennzeichnet.

5.3.7 excel.js

Das Modul `excel.js` ist für das Generieren der Routenbeschreibung und den Excel Export des Fahrtenbuches zuständig. Durch Eingabe von `$ node excel.js 'v_id' 'start' 'end'` in der Konsole wird der Excel Export separat von der Serveranwendung gestartet.

Dabei werden dem Kommando drei Parameter übergeben. Durch `v_id` wird angegeben, von welchem Fahrzeug ein Fahrtenbuch generiert werden soll. Hierbei handelt es sich um die ID des Fahrzeuges aus der Vehicles Tabelle in der Datenbank.

Mit `start` und `end` erhält das Modul einen Zeitraum für die Auswahl der Fahrten. Wird der Excel Export beispielsweise mit den Argumenten `4 01.11.2018 30.11.2018` gestartet, erhält man alle Fahrten des Fahrzeuges mit der ID, die November 2018 stattgefunden haben. Beide Parameter können ebenfalls den Wert `0` annehmen. Sollte `start` den Wert `0` und `end` ein Datum haben, werden alle Fahrten eines Fahrzeuges exportiert, die bis einschließlich `end` stattgefunden haben. Im umgekehrten Fall werden analog hierzu alle Fahrten eines Fahrzeuges ausgewählt, die einschließlich ab `start` aufgezeichnet wurden. Falls sowohl `start` als auch `end` den Wert `0` bekommen, werden alle Fahrten eines Fahrzeuges in das Fahrtenbuch geschrieben.

Nachdem eine Verbindung zur Datenbank aufgestellt wurde, wird zunächst mithilfe der in Abschnitt 5.3.6 beschriebenen speziellen `for`-Schleife für jeden Trip, der mit dem Status „Finished“ gekennzeichnet ist und die Auswahlkriterien der Eingabe erfüllt, asynchron eine Routenbeschreibung generiert.

Start- und Zieladresse der Route können direkt über die niedrigste und höchste `vehicle_gps_id` ermittelt werden.

Für die Bestimmung der drei meist befahrenen Straßen müssen die Straßen zunächst korrekt unterschieden werden, da zwei Problemfälle eintreten könnten. Der erste Fall beinhaltet zwei verschiedene Straßen in unterschiedlichen Orten, die den selben Straßennamen teilen. Diese können anhand von `city` unterschieden werden. Im zweiten Fall, ist dies jedoch nicht möglich, da man für große Straßen den gleichen

Straßennamen mit unterschiedlich Orten hat, obwohl es die selbe Straßen ist. Dabei handelt es um Autobahnen (A), Europastraßen (E), Bundesstraßen (B), Landtraßen (L) und Kreisstraßen (K). Die Bezeichnung von solchen Straßen besteht immer aus dem jeweiligen Anfangsbuchstaben und einer Zahl (z.B. A 81), wodurch sie sich von anderen Straßennamen leicht unterscheiden lassen können. Sollte auf diese Weise eine Straße gefunden werden mit unterschiedlichen Orten in `city`, so werden die Einträge unter einem Ortsnamen zusammengefasst.

Wie bereits in Abschnitt 4.3 beschrieben, kann für die Berechnung der gefahrenen Distanz auf einer Straße nicht der euklidische Abstand benutzt werden. Aus diesem Grund wird die Funktion `geodist(p1, p2, exact: true, unit: 'km')` aus dem gleichnamigen Paket verwendet, in der von der *Haversine Formel* Gebrauch gemacht wird. Diese Formel ist darauf ausgelegt, den kürzesten Abstand zweier Punkte, gegeben in Breiten- und Längengrad, auf einer Kugeloberfläche (die Erde) zu berechnen [Ven].

Nachdem anhand der Distanzberechnung für jede Straßen, die nicht Start- oder Zieladresse ist, die drei meist befahrenen Straßen ermittelt wurden, wird eine Routenbeschreibung in Form von

```
Von Mühlstraße, Tübingen
nach Torstraße, Stuttgart
(über L1208 - B464 - A81)
```

generiert und in die Datenbank geschrieben. Sollte eine Fahrt aus weniger als fünf verschiedenen Straßen bestehen, wird dementsprechend die Anzahl der meist befahrenen Straßen in der Beschreibung gesenkt. Sollten nur zwei Straßen vorhanden sein, besteht die Beschreibung nur aus Start- und Zieladresse. Falls nur eine Straße aufgezeichnet wurde, beinhaltet die Routenbeschreibung nur den Straßen- und Ortsnamen dieser Straße.

Sobald die `for`-Schleife von allen asynchronen Generierungen der Routenbeschreibung weiß, dass sie fertig sind, wird mithilfe des Pakets *exceljs* zunächst durch `new excel.Workbook` und `xlsx.readFile` eine Vorlage eingelesen. Im Anschluss werden, für die auserwählten Trips, alle für das Fahrtenbuch nötigen Daten aus der Datenbank gelesen und in die Zellen der Excel Vorlage eingetragen. Am Ende wird die Excel-Datei mit `xlsx.writeFile` unter dem Namen `Fahrzeug_v_id.xlsx` gespeichert und anschließend die Datenbankverbindung beendet.

5.4 Synchronisation

Die Synchronisation zwischen der Microsoft SQL Datenbank des Servers und der MySQL Datenbank eine Fahrzeuges wird immer zu Beginn der Fahrzeuganwendung durchgeführt. Dabei ermittelt der Server, welche Daten ihm fehlen und fragt diese im Anschluss beim Fahrzeug an. Der genaue Ablauf wird in Abbildung 5.14 beschrieben.

Nachdem sich die Fahrzeuganwendung mit seiner Datenbank und dem MQTT Broker verbunden hat, startet sie den Synchronisationsvorgang mit der letzten erfassten Trip ID innerhalb der Online-Benachrichtigung. Da die IDs der Trips automatisch inkrementiert werden, spiegelt die letzte ID gleichzeitig die Anzahl der erfassten Fahrten wieder.

Während der Server die Userdaten aus der Datenbank liest und sie veröffentlicht, werden anhand der erhaltenen letzten Trip ID des Fahrzeuges alle Trips erstellt, deren ID nicht in der Datenbank vorhanden sind. Da beim Server zu diesem Zeitpunkt noch keine Daten zu diesen Trips vorhanden sind, werden sie mit dem Status „Missing Trip“ gekennzeichnet. Sie enthalten nur eine Trip ID, das Erstellungsdatum und die Fahrzeug ID. Daraufhin sendet der Server die IDs der neu erstellten Trips und die deren Status bereits auf 0 (Missing Trip) war an den Broker.

Die Fahrzeuganwendung empfängt die Anfrage des Servers und liest erst die benötigten Daten aus der Trips und anschließend aus der GPSLookups Tabelle aus. Aufgrund von Verbindungsabbrüchen und um Datenvolumen zu sparen, werden alle ausgelesenen Daten zusammen verschickt.

Nach Erhalt der angefragten Daten werden die betroffenen Trips in der Trip Tabelle aktualisiert und deren Zustand in Status 1 (Missing GPS) versetzt. Zur selben Zeit werden bereits die neuen GPS-Koordinaten nacheinander mit einer Sekunde Abstand an den Nominatim Server gesendet und anschließend in die Datenbank aufgenommen. Nun werden alle Trip IDs von den Fahrten ermittelt, denen womöglich GPS-Einträge fehlen. Diese werden mit dem Subtopic „Incomplete Trips“ an den Broker veröffentlicht.

Auf die Nachricht des Servers geht die Fahrzeuganwendung in dessen Datenbank und zählt für jede Trip ID, die sich in der Anfrage befindet, wie viele GPS-Einträge erfasst wurden. Die selben IDs werden im Anschluss, mit ihrer Anzahl an GPS-Einträgen versehen, verschickt.

Um herauszufinden, ob bei den als „Missing GPS“ gekennzeichneten Fahrten GPS-Daten fehlen, zählt die Serveranwendung für die gleichen Trip IDs ebenfalls die Anzahl an GPS-Einträgen. Diese werden mit den erhaltenen Anzahlen des Fahrzeuges verglichen. Stimmen sie für eine Trip ID überein, heißt das für den Server, dass dieser

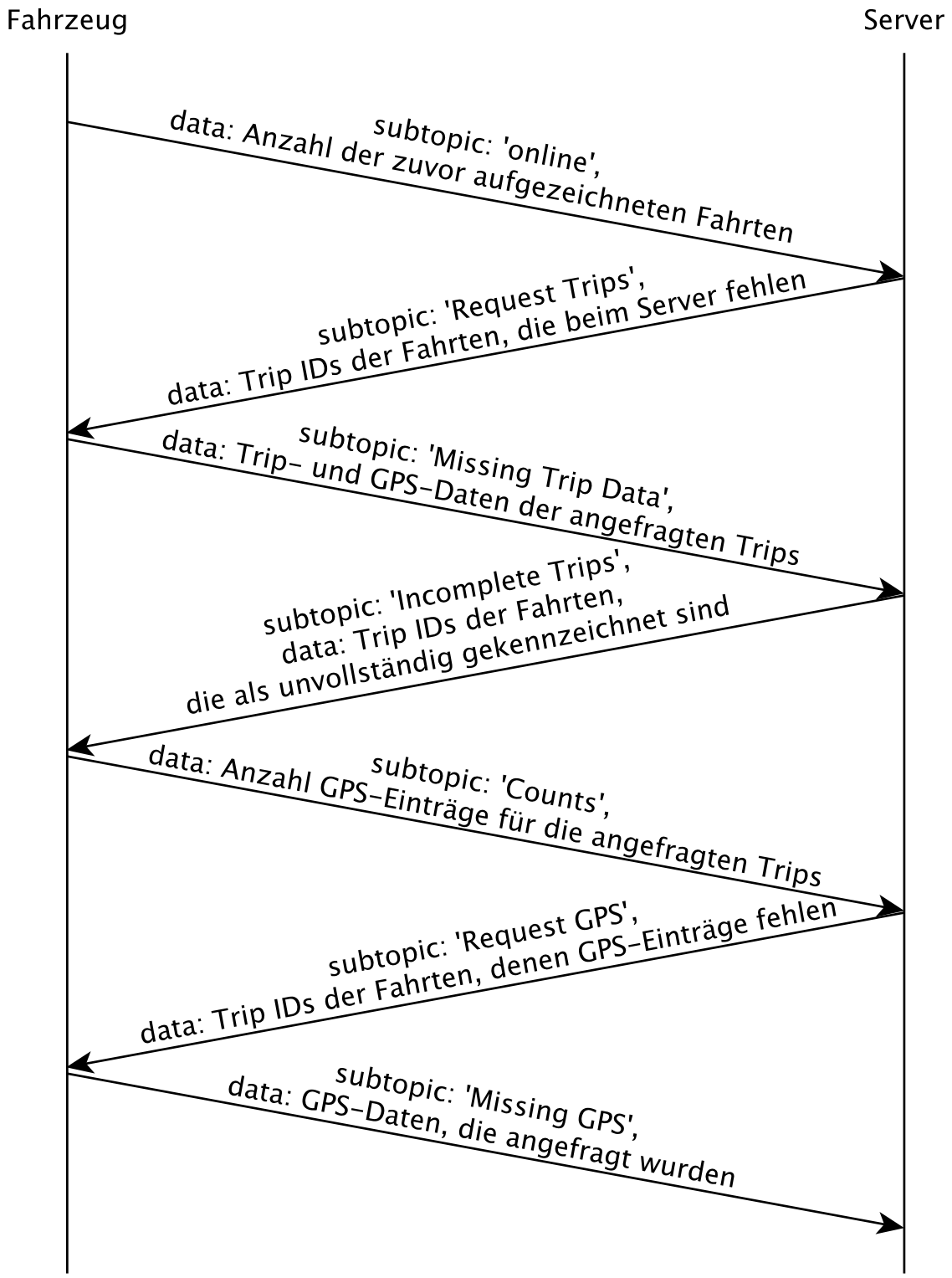

Abbildung 5.14: Synchronisationsvorgang zwischen Fahrzeug und Server

Fahrt keine Daten mehr fehlen und der Status somit auf „Complete“ geändert werden kann. Stimmen sie jedoch nicht überein, heißt das, dass mindestens ein Eintrag fehlt. Aus diesem Grund werden sie wieder beim Fahrzeug angefragt.

Nachdem die Fahrzeuganwendung auch diese GPS-Daten veröffentlicht, werden sie wieder vom Server empfangen, an den Nominatim Server gesendet und in die Datenbank aufgenommen, wodurch die Synchronisation als beendet gilt. Trips ändern jedoch ihren Status von „Missing GPS“ zu „Complete“ nur dann, wenn die Anzahl der GPS-Einträge überprüft wurde.

Durch den Status „Complete“ wird gewährleistet, dass bei großen Datenmassen, die vollständigen Daten bei der Synchronisation ignoriert werden. Somit werden während der Synchronisation nicht zu viele Daten gesendet und Datenvolumen effizient genutzt.

Außerdem ist sichergestellt, dass alles in absehbarer Zeit beim Server ankommt, da die Synchronisation immer zu Beginn der Fahrzeuganwendung stattfindet. Sollte es also während des Vorgangs zu einem Verbindungsabbruch kommen, der bist zum Ende der Fahrt anhält, dann wird versucht, diese Daten beim nächsten Start der Fahrzeuganwendung zu synchronisieren.

6 Test der Applikation

Um die Funktion der Applikation zu überprüfen, wird sie verschiedenen Tests unterzogen. Sowohl die Fahrzeug- als auch die Serveranwendung sollten dabei nicht abstürzen. Ein Absturz der Serveranwendung hätte keine großen Folgen, da die Daten, die währenddessen nicht empfangen werden konnten, noch beim MQTT Broker abgerufen werden können. Außerdem können nachträglich bei Synchronisationsvorgängen fehlende Daten ermittelt und beim jeweiligen Fahrzeug angefragt werden. Bei einem Absturz der Fahrzeuganwendung hingegen wird die aktuelle Fahrt nicht weiter aufgezeichnet.

Das Testen der Applikation hat sowohl mit als auch ohne Fahrzeug stattgefunden. Folgende Testszenarien wurden durchgeführt:

Test 1: Starten ohne laufende Serveranwendung

> *Ereignis:* Verhalten der Fahrzeuganwendung ändert sich nicht. Daten werden nach wie vor in der MySQL Datenbank aufgezeichnet und an den MQTT Broker gesendet.

Test 2: Nachträgliches Starten der Serveranwendung

> *Ereignis:* Serveranwendung alle verpassten Nachrichten vom MQTT Broker und behandelt sie. Aufgrund der Vielzahl an Nominatim Anfragen kommt es zu Timeouts. Die Anfragen werden in der täglichen Routine wiederholt.

Test 3: Manuelles Löschen eines Eintrages aus der Trips Tabelle

> *Ereignis:* Serveranwendung erkennt, dass ein Trip fehlt und beschafft sich alle Daten zu dieser Fahrt bei der nächsten Synchronisation mit dem jeweiligen Fahrzeug. Falls die zugehörigen GPS-Einträge nicht ebenfalls gelöscht wurden, befinden sich nach der Synchronisation Duplikate in der GPSLookups Tabelle, da die alten GPS-Einträge in der `trip_id` Spalte auf die alte ID des gelöschten Trips verweisen und sie somit nicht erkannt oder können.

Test 4: Manuelles Ändern eines Eintrages aus der Trips Tabelle

Ereignis: Falls der betroffene Trip den Status „Missing Trip“ besitzt, wird der Eintrag mit den Daten des Fahrzeuges überschrieben werden. Bei allen anderen Trips bleibt die Änderung bestehen.

Test 5: Manuelles Löschen eines Eintrages aus der GPSLookups Tabelle

Ereignis: Bei einer Status ID von 0 oder 1 wird der fehlende Eintrag erkannt und bei der nächsten Synchronisation angefragt. Bei allen anderen Trips muss die Status ID auf 1 heruntergestuft werden, um das Löschen rückgängig zu machen.

Test 6: Manuelles Ändern eines Eintrages aus der GPSLookups Tabelle

Ereignis: Bei einer Status ID von 0 wird der geänderte Eintrag bei der nächsten Synchronisation überschrieben, bei „Missing GPS“ hingegen nur dann, falls die Anzahl an GPS-Einträgen für den betroffenen Trip nicht mit der Anzahl beim Fahrzeug übereinstimmt. Bei allen anderen Trips bleibt die Änderung bestehen.

Test 7: Verlust der Internetverbindung während der Fahrt

Ereignis: MQTT Client schickt die Daten automatisch an den Broker sobald wieder eine Internetverbindung besteht.

Test 8: Verlust des GPS Signals während der Fahrt (z.B. im Tunnel)

Ereignis: Ungültige Koordinaten werden verworfen. Da währenddessen keine neuen GPS-Daten erfasst werden, wird die gefahrene Route in der Karte nicht weitergezeichnet. Sobald das Signal wieder da ist und neue GPS-Daten erfasst wurden, zeichnet die Karte eine Luftlinie von der letzten erfassten Position vor dem Signalverlust zum ersten danach.

Test 9: Mehrmalige Identifizierung durch NFC Tag

Ereignis: Erkannter Fahrer wird uberschrieben.

Test 10: Ausschalten der Zündung

Ereignis: Fahrt wird beendet. Die Benutzeroberfläche wird mit den aktuellen Daten zu der Fahrt aktualisiert. Nach einigen Sekunden wird der Raspberry Pi heruntergefahren.

Test 11: Manuelles Starten und Stoppen der Aufzeichnung mittels Knopfdruck

Ereignis: Zu Beginn der Fahrt ist der Startknopf automatisch deaktiviert. Somit lässt sich nur der Stop-Button drücken. Sobald dieser getätigt wird, wird die Aufzeichnung der Daten beendet und die Benutzeroberfläche aktualisiert. Dabei wird der Stop-Button deaktiviert und der Start-Button aktiviert. Sollte die Fahrt zu früh beendet werden, kann es dazu führen, dass Endzeit oder -kilometerstand noch nicht erfasst wurden und somit nicht angezeigt werden. Beim Betätigen des Start-Buttons wird ein neuer Trip angelegt, die Aufzeichnung wieder gestartet und die Benutzeroberfläche auf den Anfangszustand gebracht.

Test 12: Überqueren von Querstraßen

Ereignis: Einzelne GPS-Einträge von Querstraßen werden erkannt und interpoliert. Beim Überqueren und von mehreren Querstraßen in kurzer Zeit funktioniert die Interpolation ebenfalls, solange nicht mehrmals hintereinander die gleiche falsche Querstraßen erfasst wurde.

Test 13: Rote Ampel

Ereignis: Durch das Stehen an einer roten Ampel bei einer Kreuzung, kann es dazu kommen, dass aufgrund der GPS Ungenauigkeit mehrmals hintereinander die Querstraße erfasst wird und die Interpolation sie nicht erkennt. Dasselbe Szenario tritt bei Staus und über- oder unterführenden Querstraßen auf.

Während den Tests konnten die beiden Anwendungen nicht zum Absturz gebracht werden.

7 Fazit & Ausblick

Im Rahmen dieser Arbeit wurde das elektronische Fahrtenbuch aus einer vergangenen Studienarbeit optimiert und erweitert. Den Mitarbeitern des IVK ergibt sich dadurch in der Zukunft eine hohe Zeitersparnis. Durch die Unterstützung von mehreren Fahrzeugen und der einfach zu bedienenden Konfiguration der Fahrzeuganwendung, sind sie in der Lage mit wenig Aufwand die Fahrten ihrer ganzen Fahrzeugflotte aufzuzeichnen.

Der Verlust der Internetverbindung spielt aufgrund der eingeführten Synchronisation keine wichtige Rollen mehr, da bei jeder Inbetriebnahme der Fahrzeuganwendung die Datenbanken auf Konsistenz geprüft werden. Die während dieser Arbeit gesammelte Erfahrung zeigt, dass es bei den meisten Fahrten zu Verbindungsabbrüchen des mitgeführten mobilen Hotspots kommt. Sollten diese am Ende der Fahrt stattfinden, werden die Daten erst wieder bei der nächsten Fahrt bezogen, was zu einer Verzögerung führt, weil die Daten der vergangen Fahrt noch nicht exportiert werden können. Aus diesem Grund besteht beim Konzept der Synchronisation ein Verbesserungspotenzial.

Aufgenommene Fahrten lassen sich aufgrund der täglichen Routine erst frühestens ab 1 Uhr exportieren, da die Daten überprüft und gegebenenfalls korrigiert werden müssen. Durch manuelle Eingriffe in die Datenbank kann ein Trip jedoch exportierfähig gemacht werden. Auf die Aufbereitung der Daten kann jedoch nicht vollständig verzichtet werden, da aufgrund der GPS Ungenauigkeit eine Interpolation benötigt wird. Die Interpolation ist jedoch nicht in der Lage rote Ampeln oder Staus zu erkennen, wodurch die falschen Position nicht vollständig bereinigt werden können. Aus diesem Grund besteht der Anlass in der Zukunft eine Erkennung für solche Fälle zu entwickeln.

Die Interpolation für einen Trip könnte direkt nach dem Erhalt des Status „Complete“ ausgeführt werden. Hierzu muss jedoch aufgrund der Qualität der Reverse Geocoding Antworten ein neuer Kartendienst in Betracht gezogen werden.

Es ist sinnvoll, den Mitarbeitern des IVK eine Anwendung mit einer interaktiven Benutzeroberfläche zur Verfügung zu stellen, um die Daten in der Serverdatenbank richtig verwalten zu können. Dabei sollten die Mitarbeiter in der Lage sein, alte Daten sowohl beim Server als auch beim jeweiligen Fahrzeug korrekt löschen zu können. Auch die Bedienung des Excel Exports könnte über eine Auswahl der Fahrten in der Benutzeroberfläche erfolgen. Dadurch sind auch nicht in dieses Projekt eingewiesene Mitarbeiter in der Lage ein Fahrtenbuch exportieren oder die Daten anpassen zu können.

Eine Erweiterung der Benutzeroberfläche, die dem Fahrer während der Fahrt angezeigt wird, könnte zu einer Verbesserung des Nutzererlebnisses führen. Dabei könnte die Oberfläche um Status Icons fur die Qualität des Internetempfangs, des GPS Signals oder sonstiger Fehler und Hinweise ergänzt werden.

Für weitere Optimierungen und Erweiterungen der Applikation wurde sämtlicher Code mit JSDoc Kommentaren versehen, um zukünftigen Entwicklern den Umfang des Einarbeitens in die Applikation zu minimieren und somit ihre Arbeit zu erleichtern.

Literaturverzeichnis

[ACS] A. C. Systems. *ACR122U USB NFC Reader*. Eingesehen am 26.12.2018. URL: https://www.acs.com.hk/en/products/3/acr122u-usb-nfc-reader/ (zitiert auf S. 21).

[GG06] H.-J. Gevatter, U. Grünhaupt. *Handbuch der Mess- und Automatisierungstechnik im Automobil*. Berlin, Heidelberg: Springer, 2006. ISBN: 978-3-540-29980-6. DOI: 10.1007/3-540-29980-7 (zitiert auf S. 23).

[Gro18] K. Grossmann. *Konzeption und Implementierung einer Applikation zur automatisierten Erfassung eines elektronischen Fahrtenbuchs*. Institut für Verbrennungsmotoren und Kraftfahrwesen, Apr. 2018 (zitiert auf S. 30, 32, 36, 40, 45, 46, 50, 56).

[Höl17] C. Höller. *Angular - Das umfassende Handbuch.* 1. Aufl. Bonn: Rheinwerk, 2017. ISBN: 978-3-8362-3914-1 (zitiert auf S. 23).

[Ide] Identible. *RFID-Reader ACS ACR122U NFC*. Eingesehen am 18.01.2019. URL: https://www.identible.de/acs-acr122u-a9-rfid-nfc-reader.html (zitiert auf S. 21).

[Kom] M. Kompf. *Entfernungsberechnung*. Eingesehen am 07.01.2019. URL: https://www.kompf.de/gps/distcalc.html (zitiert auf S. 37).

[Nod] Node.js. *Über Node.js*. Eingesehen am 15.01.2019. URL: https://nodejs.org/de/about/ (zitiert auf S. 21).

[NPM] *Node Schedule*. Eingesehen am 27.01.2019. URL: https://www.npmjs.com/package/node-schedule (zitiert auf S. 65).

[Obe18] D. Obermaier. „Was ist MQTT?“ In: *Embedded Software Engineering - Fachwissen* (Juni 2018). Eingesehen am 27.12.2018. URL: https://www.embedded-software-engineering.de/was-ist-mqtt-a-725485/ (zitiert auf S. 25, 26).

[Opea] OpenStreetMap. *Urheberrecht und Lizenz*. Eingesehen am 28.12.2018. URL: https://www.openstreetmap.org/copyright/de (zitiert auf S. 26).

[Opeb] OpenStreetMap. *Willkommen bei OpenStreetMap*. Eingesehen am 28.12.2018. URL: https://wiki.openstreetmap.org/wiki/Willkommen_bei_OpenStreetMap (zitiert auf S. 26).

[OSMF] O. Foundation. *Nominatim Usage Policy*. Eingesehen am 28.12.2018. URL: https://operations.osmfoundation.org/policies/nominatim/ (zitiert auf S. 27).

[RaspF] R. P. Foundation. *FAQs*. Eingesehen am 28.12.2018. URL: https://www.raspberrypi.org/documentation/faqs/ (zitiert auf S. 19).

[Rod12] G. Roden. *Node.js & Co. (iX Edition): Skalierbare, hochperformante und echtzeitfähige Webanwendungen professionell in JavaScript entwickeln*. dpunkt.verlag, 2012. ISBN: 9783864911897. URL: https://books.google.de/books?id=jmN4DwAAQBAJ (zitiert auf S. 21).

[RT10] F. Ramm, J. Topf. *OpenStreetMap: Die freie Weltkarte nutzen und mitgestalten*. Lehmanns, 2010. ISBN: 9783865416353. URL: https://books.google.de/books?id=UpLjCQAAQBAJ (zitiert auf S. 26).

[Spr18] S. Springer. *Node.js - Das umfassende Handbuch*. 3. Aufl. Bonn: Rheinwerk, 2018. ISBN: 978-3-836-24003-1 (zitiert auf S. 22).

[Stov] *Can't determine any concurrency*. Eingesehen am 23.01.2019. URL: https://stackoverflow.com/questions/52597357/cant-determine-any-concurrency (zitiert auf S. 64).

[TB14] P. Tarasiewicz, R. Böhm. *AngularJS: Eine praktische Einführung in das JavaScript-Framework*. dpunkt.verlag, 2014. ISBN: 9783864915185. URL: https://books.google.de/books?id=InN4DwAAQBAJ (zitiert auf S. 23).

[Ven] C. Veness. *Calculate distance, bearing and more between Latitude/Longitude points*. Eingesehen am 28.12.2018. URL: https://www.movable-type.co.uk/scripts/latlong.html (zitiert auf S. 68).

[Yaa13] H. Yaapa. *Express Web Application Development*. Community experience distilled. Packt Publishing, 2013. ISBN: 9781849696555. URL: https://books.google.de/books?id=nQ_I1yuV_DwC (zitiert auf S. 22).

[ZS14] W. Zimmermann, R. Schmidgall. *Bussysteme in der Fahrzeugtechnik - Protokolle, Standards und Softwarearchitektur*. Wiesbaden: Springer Vieweg, 2014. ISBN: 978-3-658-02419-2. DOI: 10.1007/978-3-658-02419-2 (zitiert auf S. 23).

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift