



Studiengang Informationstechnik

Studienarbeit

KFZ-Spritverbrauchsanalyse für eine effiziente Routenfindung

1. Oktober 2014 - 24. Dezember 2014

30. März 2015 - 19. Juni 2015

Autoren	Matrikelnummer	Kurs
Rico Fritzsche	5050390	TINF12ITIN
Christoph Prinz	2537478	TINF12ITIN
Simon Sander	11116840	TINF12ITIN

Eigenständigkeitserklärung

Hiermit versichern wir, dass die vorliegende Arbeit von uns persönlich ohne Hilfe Dritter verfasst wurde und dass wir keine weiteren als die angegebenen Hilfsmittel und Quellen benutzt haben. Wörtliche oder sinngemäße Übernahmen aus anderen Schriften und Veröffentlichungen sind entsprechend gekennzeichnet. Sämtliche Quellen sind nachgewiesen und in dem Literaturverzeichnis aufgeführt.

Die hier vorgelegte Arbeit ist weder ganz noch in Teilen einer anderen Prüfungsbehörde vorgelegt worden.

Rico Fritzsche

Christoph Prinz

Simon Sander

Mannheim, den 19. Juni 2015

Zusammenfassung

Aktuelle Bestrebungen der Automobilindustrie zeigen einen deutlichen Trend zu verbrauchsarmen Fahrzeugen bis hin zu Elektroautos. Diese Entwicklung ist durch Umweltschutzgesetze bedingt und folgt dem Wunsch der Verbraucher nach sparsamen Modellen. Mit der vorliegenden Studienarbeit wird ein neuer Ansatz zur Reduzierung des Kraftstoffverbrauchs bei regelmäßig gefahrenen Strecken umgesetzt und dessen Tragfähigkeit untersucht. Durch die Verknüpfung von Kraftfahrzeug (KFZ)-Parametern und Ortsinformationen wird eine gefahrene Route bewertet und mit Alternativstrecken verglichen. Unter Einbeziehung des Fahrstils, der Zeit und des gefahrenen Automodells kann die CO₂-effizienteste Route ermittelt werden. Neben positiven ökonomischen Effekten für den Verbraucher kann die Umwelt entlastet und wachsenden Verkehrsproblemen in Großstädten begegnet werden. Im Rahmen dieser Arbeit wurde ein funktionsstüchtiger Prototyp entwickelt, welcher das Potenzial für weitere Forschung in sich trägt.

Abstract

Current efforts in the motor industry show a clear trend towards fuel-efficient vehicles or even electric cars. This development is the result of environment protection laws as well as the consumer's demand for more fuel-efficient models. This study presents a new approach to reduce fuel consumption for highly frequented routes, which is implemented and examined based on its sustainability. By linking car parameters and location information the route will be evaluated and compared to alternatives. Taking into account driving style, time and car model, the most CO₂-efficient route can be determined. In addition to positive economic effects on consumers the environmental impact and growing traffic problems can be reduced. As part of this study a working prototype has been developed, which carries the potential for further research.

Inhaltsverzeichnis

Abbildungsverzeichnis	IX
Tabellenverzeichnis	X
Quellcodeverzeichnis	XII
Abkürzungsverzeichnis	XIII
1 Einleitung	1
1.1 Motivation	1
1.2 These und Entwurf	4
1.3 Ziele und Aufgabenstellung	5
1.4 Stand der Technik	5
1.5 Aufbau der Arbeit	6
2 Konzept	8
2.1 Messdatenerfassung	8
2.1.1 Funktionsumfang des Frontends	9
2.1.2 Wahl der Smartphoneplattform	10
2.2 Messdatenverarbeitung	12
2.2.1 Webserver	12
2.2.2 Definition von Kreuzungen	13
2.2.3 Algorithmus zur Erfassung von Kreuzungen	13
2.2.4 Wahl des Geocoding Frameworks	16
2.3 Messdatenspeicherung	17
2.3.1 Technische Daten des Backendservers	17
2.3.2 Repräsentation der Daten	19
2.3.3 Zusammenfassung von Strecken zu Routen mittels Tags	21
2.4 Messdatenfluss und Infrastruktur	22
3 Grundlagen	24
3.1 Kommunikation mit dem Fahrzeug	24
3.1.1 On-Board-Diagnose	25
3.1.2 ELM327-Mikrocontroller	26
3.1.3 Zugriff auf den ELM327 Bluetooth RFCOMM	27

3.2 Runtime-App-Entwicklung für Windows Phone 8.1	28
3.2.1 Komponenten einer Windows Runtime App	29
3.2.2 APIs	32
3.2.3 Debugging	34
3.2.4 Deployment	35
3.3 Server-Client-Modell	35
3.4 Representational State Transfer	36
3.5 Node.js	37
3.5.1 Struktur und Aufbau	37
3.5.2 Entwicklung mit Webstorm und REST Client Add-On	39
3.5.3 Node.js und JavaScript	40
3.6 Graphen und Graphdatenbanken	43
3.6.1 Graphdatenbank Neo4J	44
3.6.2 Abfragesprache Cypher	45
3.7 Openstreetmap	46
3.7.1 Geokodierung	48
3.7.2 Geocoding-Framework Gisgraphy	49
3.7.3 Openstreetmap-Schnittstelle Overpass	49
3.8 Lizenzen	50
4 Umsetzung des Frontends	53
4.1 Anforderungen	53
4.2 Architektur der Windows Phone-Applikation	54
4.3 Kommunikation mit dem Server	55
4.3.1 Verarbeitung von Objekten in JSON-Notation	55
4.3.2 Nutzerverwaltung	57
4.4 Bluetooth-Kommunikation	59
4.4.1 Aufbau der Bluetooth-Verbindung und Initialisierung der Kommunikation	60
4.4.2 Zugriff auf den Kommunikationssocket	62
4.5 Messdatenerfassung im Tracking-Prozess	62
4.5.1 Auslesen und Interpretieren von OBD-Parametern	63
4.5.2 Bestimmung des Momentanverbrauchs	65
4.5.3 Aufzeichnen einer Strecke	67
4.6 Vergleichen von Strecken	70

4.7 Erfüllung nicht-funktionaler Anforderungen	71
4.7.1 Bedienungskonzept und Nutzerergonomie	71
4.7.2 Anpassung auf verschiedene Displaygrößen	72
4.7.3 Reduzierter Stromverbrauch	73
4.7.4 Robustheit gegenüber schwacher Datenverbindung	73
4.7.5 Reaktion auf Displayrotationen	74
4.8 Evaluation	75
5 Umsetzung der Serveranwendung mit Node.js	77
5.1 Anforderungen	77
5.2 Webserver	77
5.2.1 Express als Framework	78
5.2.2 Routing eingehender Nachrichten	78
5.2.3 Schnittstelle zur App	79
5.2.4 Benutzerauthentifikation	81
5.3 Algorithmen zur Datenerfassung und -auslieferung	85
5.3.1 Komprimierung der GPS-Punkte zu Kreuzungspunkten zur Speicherung	85
5.3.2 Zusammenführen vorhandener Daten mit einer neuen Strecke	90
5.3.3 Vorberechnung der Daten zur Auslieferung an den Client	93
5.4 Evaluation	98
6 Umsetzung des Backend-Servers	100
6.1 Anforderungen	100
6.2 Absicherung gegen digitale Angriffe	100
6.3 Mapping einer GPS-Koordinate zu einer Straßen-ID mit Gisgraphy	102
6.4 Abfragen an die Openstreetmap-Datenbank mittels Overpass	105
6.5 Evaluation	107
7 Fazit und Ausblick	108
7.1 Fazit	108
7.2 Ausblick	109
Literatur- und Quellenverzeichnis	XII

Abbildungsverzeichnis

1	Übersicht zur Entwicklung des Fahrzeugbestandes.	2
2	Entwicklung der CO ₂ -Emissionen bei Neuwagen in Deutschland.	3
3	Schematische Darstellung des Graphdatenbank-Designs.	20
4	Darstellung einer Route mit drei Strecken.	21
5	Skizze der Infrastruktur für Streckenabhängige Verbrauchsmessung und -analayse (SAM).	22
6	OBD-Schnittstelle eines Seat Ibiza 6J.	25
7	XAML-Markup und Simulation der SAM-Einstellungen.	30
8	Simulation des Standorts im Windows Phone 8.1-Emulator auf einem 4 Zoll Smartphone.	34
9	Schema der 3-Tier-Architektur.	35
10	Node Event Schleife	38
11	Webstorm-Arbeitsumgebung	39
12	Schematische Darstellung eines Property-Graphen.	43
13	Schematische Darstellung der Elementtypen von Openstreetmap.	47
14	Geocoding Beispiel.	48
15	Inverse Geocoding Beispiel	49
16	Übersicht der verschiedenen Views der App.	54
17	Willkommensbildschirm der Windows Phone-App (MainPage).	58
18	View zum Verbinden des OBD-Adapters (ConnectOBDPPage).	61
19	Klassendiagramm für die OBD-Kommunikation (unvollständig).	64
20	Regression zwischen Momentanverbrauch und OBD-Parametern zur Kalibrierung der Verbrauchsberechnung.	67
21	Hauptmenü der App zur Auswahl einer Strecke (MainPage).	68

22 Anzeige von aktuellen Fahrtdataen sowie einer Zusammenfassung während des Trackings (TrackingPage).	69
23 Zusammenfassung-View der App (SummaryPage).	70
24 Vergleich von Strecken zu einem Tag (ComparisonPage).	71
25 Rotation am Beispiel des Willkommensbildschirms (StartPage).	74
26 Schema der eingehenden JSON-Daten.	80
28 Ablaufdiagramm des Algorithmus	86
29 Ablaufdiagramm des Algorithmus in dem Adapter zur Overpass-Anfrage.	87
30 Minimale Distanzen zwischen aufgezeichneten GPS-Punkten und Kreuzungen.	89
31 Ablaufdiagramm des Algorithmus zum Zusammenführen von Daten.	91
32 Vier Szenarien, welche der Merge-Algorithmus behandelt.	92
33 Schema zur Bestimmung der HitIDs	94
34 Schema der Overpass-Anfrage.	105

Tabellenverzeichnis

1 Konzepte zur Messdatenerfassung.	9
2 Übersicht der Marktanteile mobiler Betriebssysteme in Deutschland (Stand Februar 2015) [6]	11
3 Vergleich der Algorithmen zur Kreuzzungserfassung.	15
4 Vergleich von Geocoding Frameworks.	16
5 Vergleich von Serverumgebungen.	18
6 Importierte Module für die Benutzeroauthentifikation und deren Funktion. .	84
7 Mögliche Kombinationen der Zusammenführung von Daten in der Datenbank.	95

Quellcodeverzeichnis

1	Algorithmus: Straßennamenänderung mit Reverse Geocoding	14
2	Algorithmus: Web-API (GeoNames)	14
3	Eigener Algorithmus zur Kreuzungsfindung	15
4	Kommunikation mit dem ELM327 zur Initialisierung des Mikrocontrollers	27
5	Kommunikation mit dem ELM327 zum Abfragen der aktuellen Motordrehzahl	27
6	Gegenüberstellung einer Eigenschaft und eines Getters in C#	31
7	Zugriff auf eine Eigenschaft in C#	32
8	Beispiel für asynchrone Ausführung einer Methode	33
9	Beispiel einer package.json-Datei	38
10	Anonyme Funktionen	42
11	Beispiel eines Cypher Statements	45
12	Nachbildung der JSON-Objekte als C#-Eigenschaft	56
13	Serialisierung der Trip-Daten als Beispiel für die Generierung eines Objektes in JSON-Notation	56
14	Deserialisierung der Strecken zu einem Tag als Beispiel für das Parsen eines Objektes in JSON-Notation	57
15	Übernahme eines Cookies aus einem HTTP-Paket in die Kommunikationsinstanz	59
16	Eintragung in das App-Manifest zur Autorisierung der Verwendung eines RFCOMM-Bluetooth-Sockets	59
17	Anzeige von bereits gekoppelten Bluetooth-Geräten, welche das RFCOMM-Protokoll unterstützen	60
18	Aufbau des Kommunikationssockets mit dem OBD-Adapter	61
19	Senden und Empfangen einer Nachricht mithilfe des Kommunikationssockets	62

20 Senden, Empfangen und Interpretieren eines OBD-Kommandos.	65
21 Abfrage aller Tags eines Nutzers.	81
22 Pseudocode für die Methode addRelation.	93
23 Generierung der Antwortliste aus Strecken für das Frontend.	96
24 Sortierung des Results einer Cypher-Anfrage nach dem gesetzten Zeitstempel.	96
25 Zusammengefasste Routen im JSON-Format für das Frontend.	98
26 Auszug aus dem sshd-Log.	101
27 Auszug aus der Konfigurationsdatei sshd_config.	101
28 Statusausgabe der ufw-Firewall.	102
29 Request einer <i>Street Search</i> -Anfrage.	103
30 Response einer Anfrage an den <i>Street Search</i> -Endpoint.	104
31 Overpass-Query zur Kreuzungsfindung.	105
32 Response eines Overpass-Query zur Kreuzungsfindung.	106

Abkürzungsverzeichnis

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Program Interface
ASCII	American Standard Code for Information Interchange
DHBW	Duale Hochschule Baden-Württemberg
DOM	Document Object Model
EOBD	European On-Board-Diagnose
GiST	Generalized Search Tree
GPS	Global Positioning System
GSM	Global System for Mobile Communications
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protokoll
IC	Integrated Circuit
IDE	Integrated Development Environment
IO	Input/Output (Eingang/Ausgang)
JOBD	Japanese On-Board-Diagnose
JS	Javascript
JTA	Java Transaction API
JSON	JavaScript Object Notation
KFZ	Kraftfahrzeug
NPM	Node Package Manager
NIO	Non-blocking I/O
OBD	On-Board-Diagnose
OSM	Openstreetmap
PID	Permanent Identifier
REST	Representational State Transfer
RDF	Resource Description Framework
SAM	Streckenabhängige Verbrauchsmessung und -analyse

SSH	Secure Shell
SDK	Software Development Kit
UART	Universal Asynchronous Receiver Transmitter
UI	User Interface
USB	Universal Serial Bus
VCDS	Volkswagen COM Diagnose-System
VPN	Virtual Private Network
WLAN	Wireless Local Area Network
XAPI	Extended API
XML	Extensible Markup Language

1 Einleitung

Im Rahmen dieser Arbeit wird eine Anwendung entwickelt, die eine effiziente Routenfindung auf Basis des Kraftstoffverbrauches ermöglicht.

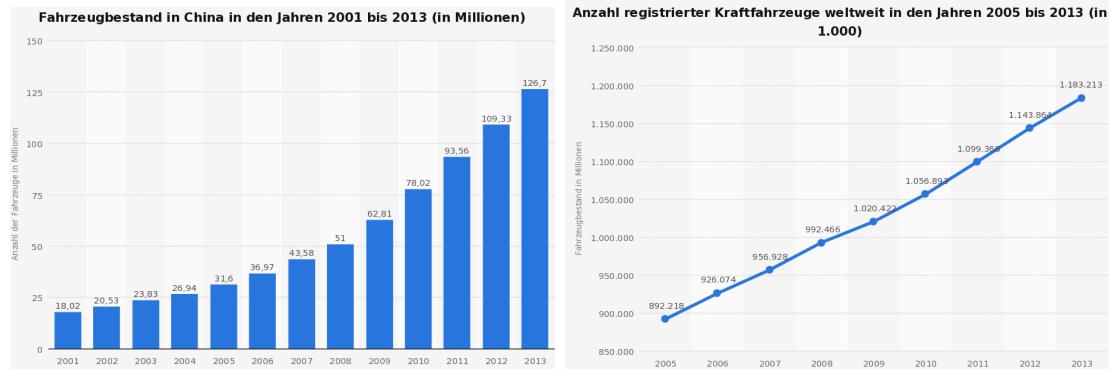
Dieses Kapitel erläutert die Motivation hierfür, erarbeitet eine These und definiert auf dieser Basis eine Aufgabenstellung und Ziele. Ferner wird der aktuelle Stand der Technik vorgestellt und ein Ausblick auf die Kapitel dieser Arbeit gegeben.

1.1 Motivation

Mit dem Werbeslogan „*Freude am Fahren*¹“ verbindet BMW Spaß und Mobilität mit KFZs. Im städtischen Verkehr sind jedoch oft gegensätzliche Szenen zu beobachten, im Besonderen zu Hauptverkehrszeiten. Staus, lange Ampelwartezeiten und Baustellen beeinträchtigen die Freude am Fahren. Dennoch möchte ein Großteil der Verkehrsteilnehmer nicht auf die Vorteile der Mobilität verzichten. Die ständige Bereitschaft, große Distanzen zu überbrücken, eine individuelle Beladung sowie die Freiheit der Routenwahl gehören zu diesen Vorteilen. Das Kraftfahrzeug wird als wichtiger, aber auch selbstverständlicher Teil der persönlichen Entfaltung gesehen und ist in der Gesellschaft anerkannt. Dennoch eröffnet diese Art der Mobilität Problemfelder.

In Europa gehören Staus zu den Hauptverkehrszeiten zum Alltag. Mit steigendem Lebensstandard in anderen Teilen der Welt steigt auch dort das Bestreben nach einem Kraftfahrzeug. Als Beispiel kann hier die Volksrepublik China angeführt werden. Eine wachsende Wirtschaftskraft mit zunehmender Lebensqualität geht einher mit steigendem Fahrzeugbestand. Am chinesischen Beispiel in Abbildung 1a ist zu erkennen, wie sich der Fahrzeugbestand zwischen den Jahren 2001 und 2013 um 85,78 % gesteigert hat. Die Abbildung 1b verdeutlicht, dass diese Entwicklung nicht nur ein regionales Phänomen ist, sondern eine globale Entwicklung darstellt. Zu erkennen ist, dass sich die Anzahl der registrierten Fahrzeuge in sieben Jahren zwischen 2005 und 2013 um fast 291 Millionen erhöht hat.

¹BMW Werbeslogan seit 1969



- (a) Entwicklung des Fahrzeugbestandes in China zwischen 2001 und 2013 [1]. (b) Anstieg der Anzahl an KFZs weltweit zwischen 2005 und 2013 [2].

Abbildung 1: Übersicht zur Entwicklung des Fahrzeugbestandes.

Dieser Entwicklung stellen sich nun Probleme entgegen, welche durch eine zielführende Diskussion erörtert und benannt werden müssen. Dazu gehört zum einen die Überstrapazierung der Verkehrsinfrastruktur. Durch die hohe Auslastung der Straßen kommt es regelmäßig zu Stau und langen Wartezeiten an Kreuzungen. Des Weiteren beschädigt die Auslastung die Verkehrswege, was zu Baustellen führt, welche den Verkehr zusätzlich belasten, und zum anderen finanzielle Investitionen notwendig macht. Ebenfalls als Problem zu adressieren ist der steigende CO₂-Ausstoß durch die wachsende Anzahl von Verkehrsteilnehmern weltweit. In Europa macht der Anteil der CO₂-Abgase 26 % aus, wobei Automobile 12 % beitragen [3].

Die Entwicklung der CO₂-Emissionen bei Neuwagen ist seit 2009 bis ins Jahr 2014 stetig gesunken, wie die Abbildung 2 für Deutschland zeigt. Jedoch ist die Verbesserung nicht ausreichend, daher reagiert die Europäische Union unter anderem mit einer Verordnung, die von den Automobilherstellern verlangt, dass die Effizienz der Fahrzeuge erhöht und die CO₂-Emission vermindert wird. Konkret ist vorgesehen, dass ab 2020 die Gesamtflotte der Neuwagen einen Grenzwert von 95 $\frac{g}{km}$ CO₂ nicht überschreiten darf [3].

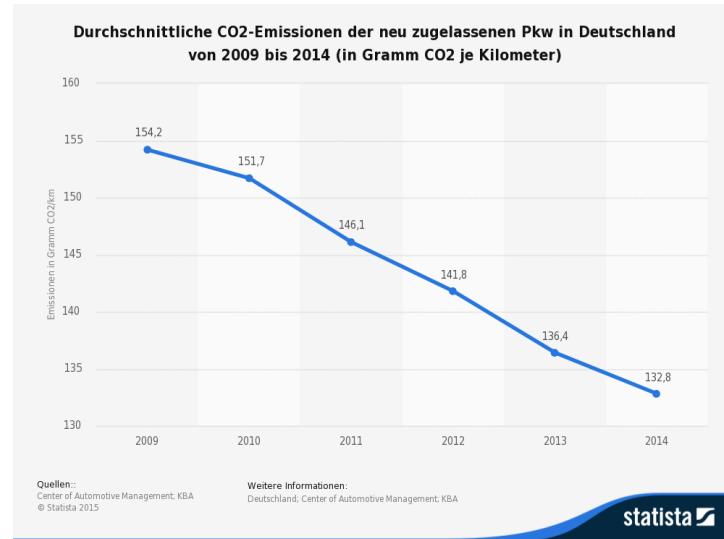


Abbildung 2: Entwicklung der CO₂-Emissionen bei Neuwagen in Deutschland.

Die genannten Probleme sind vielfältig und benötigen zahlreiche Lösungsansätze, die miteinander kooperieren müssen, um erfolgreiche Lösungen zu generieren. Diesen Weg verfolgt auch die vorliegende Studienarbeit „Streckenabhängige Verbrauchsmessung und -analyse“ (SAM). Ziel ist es, einen Beitrag zur effizienteren Gestaltung des Verkehrswesens beisteuern zu können und einen Lösungsvorschlag für aktuelle Probleme zu liefern. Die Reduzierung des Kraftstoffverbrauches dient zum einen der Einsparung von Energie und der Minderung der CO₂-Emissionen. Dadurch kann der Einfluss von KFZs auf das Klima verringert und die Umwelt geschont werden. Letzteres erhöht besonders in Städten unmittelbar die Lebensqualität der Menschen. Des Weiteren ermöglicht ein reduzierter Kraftstoffverbrauch positive ökonomische Effekte. Dies ist umso wichtiger, wenn man aktuelle Preisentwicklungen für Kraftstoffe vergleicht. Daher ist es erstrebenswert, einen verminderten Verbrauch der Fahrzeuge zu ermöglichen.

1.2 These und Entwurf

Das Projekt SAM soll einen neuen Weg zur Kraftstoffeinsparung aufzeigen und somit eine Teillösung für aktuelle Probleme anbieten, wie sie in Abschnitt 1.1 dargelegt sind. Ziel ist es, dass nicht mehr eine Route im Mittelpunkt steht, sondern der Fahrer und seine Umwelt. Dazu werden die Fahrzeugdaten ausgelesen, die über eine On-Board-Diagnose (OBD)-Schnittstelle zur Verfügung stehen. Auf Basis dieser Daten kann der Verbrauch des Fahrzeuges ermittelt werden. Zur Auswertung einer gefahrenen Strecke genügt es, die Informationen über den Kraftstoffverbrauch zu analysieren. Der Verbrauch reflektiert Verkehrssituationen, Wettereinflüsse, Fahrstil und Streckencharakteristiken. Wenn es zum Beispiel sehr warm ist und die Klimaanlage benutzt wird, steigt direkt proportional der Verbrauch. Ebenfalls proportional verhält sich der Verbrauch, wenn eine gefahrene Route Steigungen enthält. Auch lange Wartezeiten in Staus oder an Ampelkreuzungen, häufige Bremsvorgänge oder schnelle Anfahrten wirken sich auf den Verbrauch aus.

Fährt ein Fahrer eine Strecke regelmäßig, kann diese Strecke bewertet werden. Neben den genannten Einflussfaktoren ist die Tageszeit zu berücksichtigen. Dazu werden die Kraftstoffverbrauchsdaten mit einem Zeitstempel markiert und mit GPS-Daten verknüpft. Wird eine hoch frequentierte Strecke zu Hauptverkehrszeiten gefahren, ist es wahrscheinlich, in einen Stau zu geraten. Wird diese Strecke jedoch vorrangig nachts gefahren, entfallen Wartezeiten. Dies führt dazu, dass Strecken tageszeitabhängig, fahrstilabhängig und fahrzeugabhängig unterschiedlich zu bewerten sind. Damit die aufgezeichneten Informationen möglichst repräsentativ sind, wird bei mehrfacher Befahrung das arithmetische Mittel der Parameter „Verbrauch“ und „Benötigte Zeit“ gebildet. Als Konsequenz ist es denkbar, dass eine Strecke A im Mittel weniger Kraftstoff verbraucht als eine Strecke B, obwohl Strecke A eine längere Distanz aufweist.

Durch diese Optimierung wird nicht nur der Verbrauch eines einzelnen Fahrzeuges gesenkt. Wenn lange Wartezeiten in Staus den Verbrauch erhöhen, wird der Verkehrsfluss

geändert. Wenn dieses Konzept im Rahmen von Big Data² Anwendung findet (zum Beispiel in Navigationssystemen integriert) und weiter erforscht wird, könnte eine dezentrale Verkehrsführung entstehen.

Die Auswertung der aufgezeichneten Daten soll nicht die Bewertung des Fahrstils zum Inhalt haben. Das Konzept sieht lediglich vor, dass der Fahrstil bei der Effizienzanalyse berücksichtigt wird.

1.3 Ziele und Aufgabenstellung

Ziele der Studienarbeit „SAM“ sind der Entwurf und die Entwicklung einer Anwendung als Prototyp. Dabei soll die konzeptionelle Arbeit in eine funktionsfähige Anwendung überführt werden. Hierfür wird aktuelle Technik genutzt, um die Tragfähigkeit und Anwendungsmöglichkeit derselben zu erforschen. Neben dem Aufzeichnen von Sensor-signalen ist es das Ziel, erfasste Daten zu persistieren und analysieren zu können. Zur Realisierung wird eine Infrastruktur geplant und umgesetzt. Der Funktionsumfang des Prototypen wird in folgender Übersicht zusammengefasst.

- Erfassung von Positions- und KFZ-Sensordaten
- Datentransfer
- Verarbeitung und Persistierung der nutzerbezogenen Daten
- Auswertung persistierter Daten
- Plattform zur Darstellung der ausgewerteten Daten

1.4 Stand der Technik

Aktuelle Anwendungen Mit Anwendungen wie Google Maps und NOKIA here ist es möglich, Routen für KFZs zu planen. Die Anbieter ermöglichen die Berechnung der Verbrauchskosten für die jeweilige Strecke. Diese Berechnung basiert auf Schätzungen zur Entfernung, dem allgemeinen, durchschnittlichen Verbrauch und dem aktuellen

²Mit Big Data wird das Sammeln, Auswerten und Verarbeiten von Daten bezeichnet, deren Menge es nicht ermöglicht, Aktionen manuell durchzuführen.

Kraftstoffpreis. Unter Umständen werden Informationen zur Verkehrslage hinzugezogen. Die Dienste verfolgen dabei meist das Ziel, die kürzeste oder schnellste Route aufzuzeigen. Dabei werden jedem Nutzer ähnliche oder gleiche Ergebnisse geliefert. Die Berücksichtigung der fahrzeugrelevanten Parameter sowie der Fahrweise des Fahrers entfällt. Somit können nur grobe und verallgemeinernde Schätzungen getroffen werden.

Patente Das Patent „Verfahren und Vorrichtung zum Bestimmen einer Route mit einem geschätzten minimalen Kraftstoffverbrauch für Fahrzeuge“ aus [4] verfolgt ein vergleichbares Konzept. Es werden Segmente gebildet, welchen bestimmte Eigenschaften zugeordnet werden. Diese Parameter sind zum einen die geschätzte Anzahl an Stopps und zum anderen die geschätzte Anzahl an Abbiegungen. Die dabei jeweils auftretende Geschwindigkeitsverringerung wird zur Kalkulation des Kraftstoffverbrauches genutzt und mit anderen Routen verglichen.

Bemerkenswert ist, dass Stopps (die einer Geschwindigkeitsverringerung entsprechen) als Kraftstoffeinsparung angesehen werden. Dabei wird nicht berücksichtigt, dass beim Startvorgang deutlich mehr Kraftstoff verbraucht wird, als wenn mit konstanter Geschwindigkeit die gleiche Strecke gefahren wird. Des Weiteren beruhen sämtliche Kalkulationen auf Schätzungen, was sie ungenau macht.

Weitere Konzepte Aus [5] geht hervor, dass es weitere Ansätze gibt, um den Kraftstoffverbrauch zu optimieren, wie die BMW EcoChallenge, das Porsche ACC InnoDrive, das GreenGPS und die Garmin Eco Route. Das dabei vielversprechendste System ist das GreenGPS, welches jedoch die Anschaffung mehrerer Geräte verlangt. Dennoch konnten hier nennenswerte Einsparungen erreicht werden.

1.5 Aufbau der Arbeit

Die vorliegende Arbeit beinhaltet die Beschreibung von der Entwicklung des Konzeptes bis zur fertigen Umsetzung eines Prototypen. Der Bericht soll Entscheidungsfindungen

darlegen und verwendete Technologien vorstellen. Dabei wird erklärt, wie die eingesetzte Technik zusammenwirkt, und die theoretischen Überlegungen im Praktischen implementiert.

Kapitel 2 erörtert die Messdatenerfassung sowie die Messdatenspeicherung. Dabei werden mögliche Implementierungen diskutiert und verglichen. Darüber hinaus wird die Infrastruktur der Anwendung dargestellt. Daraus leiten sich Entscheidungen zur Umsetzung ab.

In Kapitel 3 werden grundlegende Technologien näher betrachtet, welche für das Projekt relevant sind.

Kapitel 4 befasst sich mit der Implementierung des Frontends, welches die Messdatenerfassung und Repräsentation der optimalen Route umfasst.

Dem schließt sich Kapitel 5 an, in dem die Umsetzung der Serveranwendung ausführt wird. Hierbei wird zwischen dem Webserver und den Algorithmen zur Auswertung und Analyse der Daten unterschieden.

Kapitel 6 beinhaltet die Darstellung der Implementierung des Backend Servers. Dazu werden die installierten Dienste erläutert und die Funktionalität dargestellt.

Abschließend wird ein Fazit der Arbeit gezogen und ein Ausblick erörtert.

2 Konzept

Im folgenden Kapitel sollen technische Möglichkeiten und deren Umsetzbarkeit im Rahmen von SAM diskutiert werden. SAM lässt sich in die drei Themen Messdatenerfassung, Messerarbeitung und Messdatenspeicherung aufteilen. Zu Beginn müssen die Messdaten einer Route erfasst werden. Danach erfolgt die Analyse und Verarbeitung der erfassten Daten. Die extrahierten Daten müssen nun persistent gespeichert werden. In den folgenden Kapiteln werden konzeptionelle Entscheidungen für die einzelnen Themenbereiche getroffen, die als Grundlage für die technische Umsetzung dienen sollen.

2.1 Messdatenerfassung

Ein zentrales Konzept von SAM ist es, eine optimale Route zu bestimmen, die auf den entsprechenden Nutzer und sein Fahrzeug abgestimmt ist. Das Konzept für die Datensammlung wird im folgenden Abschnitt vorgestellt.

Hervorzuheben ist, dass neben der Ortung KFZ-Sensordaten aufgezeichnet werden sollen, um die Effizienz eines Streckenabschnittes zu bewerten. Die Sensordaten sollen über die OBD-Schnittstelle moderner Fahrzeuge ausgelesen werden.

Zur Umsetzung der Messdatenerfassung in einer für den Nutzer komfortablen Art und Weise kommt grundsätzlich eine Embedded-Lösung oder die Verwendung eines Smartphones in Frage. In Tabelle 1 werden diese anhand einer Arduino und einer Windows Phone basierten Lösung gegenübergestellt.

Das Konzept der Verwendung eines Mikrocontrollers³ hat den Hauptvorteil, dass im Praxisbetrieb keine Interaktion mit dem Nutzer notwendig ist. Dieser würde automatisch mit der Zündung des Fahrzeuges aktiviert werden und müsste sicherstellen, dass die Daten vor Deaktivierung der Zündungpersistiert werden. Eine Smartphone-Anwendung muss dagegen manuell gestartet werden. Dafür wäre sie von der Gesamtkostenbilanz günstiger, verfügt über eine Datenverbindung und könnte gleichzeitig auch zur Darstellung der optimalen Strecken genutzt werden.

³Hier evaluierte Vor- und Nachteile sind konzeptbedingt, weshalb andere Mikrocontrollerplattformen nicht berücksichtigt werden.

Tabelle 1: Konzepte zur Messdatenerfassung.

Hardware	Embedded	Smartphone
Device	Arduino Uno v3 (20 €)	Microsoft Lumia 530 (75 €)
Momentanverbrauch via OBD	OBD-Shield (40 €)	ELM327 Bluetooth Adapter (15 €)
Position via GPS	Adafruit GPS-Logger (40 €)	eingebaut
Datenübertragung via Mobilfunk	Arduino GSM Shield (69 €)	eingebaut
technische Daten	Embedded	Smartphone
CPU	ATmega328 (16 MHz)	Qualcomm Snapdragon 200 (4 x 1.2 Ghz)
RAM	2 KByte	512 MByte
Betriebssystem	nicht vorhanden	Windows Phone 8.1 (Lumia Denim)
Stromversorgung	über OBD-Bus	über eingebauten Akku
Eigenschaften	Embedded	Smartphone
Entwicklung	Arduino-Bibliotheken	Windows Runtime
Leistungsfähigkeit	gering	vielseitig
Updatemöglichkeit	manuelles Flashen	über Windows Store
Geocoding	begrenzt möglich	Here.com API
Nutzerinteraktion	nicht erforderlich	Starten der App
Interoperabilität	Arduino Mikrocontroller mit Shield-Unterstützung	für PC, Tablets und Smartphones mit Windows 8.1

Aufgrund der dargestellten Vorteile und der Gegenüberstellung aus Tabelle 1 ergibt sich, dass die smartphonebasierte Implementierung der Messdatenerfassung zu bevorzugen ist.

2.1.1 Funktionsumfang des Frontends

Um den Entwicklungsaufwand für das Frontend möglichst niedrig zu halten, wird ein Thin-Client-Ansatz verfolgt. Das bedeutet, dass möglichst viele Funktionen vom Ba-

ckend bereitgestellt werden und im Frontend so wenig Funktionalität wie nötig implementiert wird. Das Backend kann über standardisierte Schnittstellen von verschiedenen Plattformen aus angesprochen werden. Um den Anforderungen an den SAM-Prototypen gerecht zu werden, werden diesem Ansatz gemäß folgende Funktionen im Frontend implementiert:

- User-Authentifizierung
- Messdatenerfassung
 - GPS-Koordinaten
 - Momentanverbrauch aus OBD-Werten
- Anzeigen der optimalen Route

2.1.2 Wahl der Smartphoneplattform

Damit möglichst viele Nutzer SAM verwenden können, ist es langfristig notwendig, mobile Anwendungen für alle Smartphoneplattformen anzubieten. Um die Anwendung nicht für verschiedene Plattformen entwickeln zu müssen, gibt es die Möglichkeit, eine Web-Applikation zu entwickeln. Diese hätte jedoch nur eingeschränkten Zugriff auf die Hardware-APIs, wie zum Beispiel der GPS-API, sodass dies für SAM ungeeignet ist. Eine anderer Ansatz, eine einmal entwickelte Anwendung auf allen gängigen mobilen Betriebssystemen einsetzen zu können, ist die Entwicklung mithilfe eines „Cross-Plattform-Frameworks“ wie Xamarin⁴. Dieses ermöglicht die Verwendung des gleichen Quellcodes für unterschiedliche Plattformen. Bei Xamarin wird der universelle Quellcode mit der Entwicklung eines nativen Designs verknüpft. Somit können Designrichtlinien der unterschiedlichen Betriebssysteme umgesetzt und die Vorteile der nativen Performance genutzt werden.

Es ist im Rahmen dieser Arbeit nicht vorgesehen, ein solches Framework zu nutzen, da lediglich die Umsetzbarkeit des gesetzten Ziels demonstriert werden soll. Dazu genügt es, eine native Anwendung zu implementieren.

In Deutschland spielen dabei die in Tabelle 2 gegenübergestellten Plattformen eine

⁴<http://xamarin.com/>

Rolle. Da auf Blackberry Smartphones Android Apps ausgeführt werden können, wird das Blackberry OS 10 im Rahmen der Plattformwahl nicht separat betrachtet.

Tabelle 2: Übersicht der Marktanteile mobiler Betriebssysteme in Deutschland (Stand Februar 2015) [6]

Betriebssystem	Marktanteil	Aktuelle Version (Anteil)
Google Android	72,7 %	Lollipop 5.0 (1,6 %) KitKat 4.4 (39,7 %)
Apple iOS	17,4 %	iOS 8 (78 %)
Microsoft Windows Phone	8,2 %	Windows Phone 8.1 (70,9 %)

Die Entwicklerlizenz für das Apple Developerportal ist kostenpflichtig und ferner wird neben dem iPhone ein Mac zur Entwicklung benötigt. Im Rahmen von SAM kann iOS mangels der Verfügbarkeit von Apple Hardware nicht als Smartphone-Plattform genutzt werden.

Gemessen an aktuellen Marktanteilen erreicht die Entwicklung einer Android App die meisten Nutzer. Die Entwicklungsumgebung kann ferner auf allen Betriebssystemen genutzt werden und eine Lizenz wird nicht benötigt. Problematisch ist jedoch die Fragmentierung der verschiedenen Betriebssystemversionen, sodass eine Anwendung separat für die Versionen 4.1-4.4 und 5.0 optimiert werden müsste. Eine exemplarische Recherche zeigt, dass alle für SAM benötigten APIs zwischen Android 4.1 und 5.1⁵ geändert wurden, was einen erhöhten Entwicklungsaufwand bedeuten würde, um vom hohen Android-Marktanteil profitieren zu können.

Da im Rahmen von SAM die Umsetzbarkeit der Idee demonstriert werden soll, ist bei der Plattformwahl der voraussichtliche Entwicklungsaufwand wichtiger als ökonomische Faktoren. Aus folgenden Gründen wird daher eine Entwicklung einer nativen App auf der Microsoft Windows Phone 8.1-Plattform umgesetzt:

⁵78 % der Android-Geräte nutzen eine Android-Version zwischen 4.1 und 5.0.

- Vorhandensein eines Nokia Lumia 930 als Testgerät
- keine Gebühren für Developer-Account erforderlich
- native Unterstützung von verwendeten Standards:
 - Bluetooth mit dem RFCOMM-Protokoll
 - HTTP/REST/JSON
- Offline-Nutzbarkeit von Geocoding APIs
- Offline-Verfügbarkeit von weltweitem Kartenmaterial

2.2 Messdatenverarbeitung

2.2.1 Webserver

Der Webserver ist die Schnittstelle zwischen der mobilen Anwendung auf dem Smartphone und der Datenbank. Informationen des Smartphones werden entgegengenommen und mit einer passenden Antwort quittiert. Ebenso erhält der Webserver Zugriff auf die Datenbank, wenn persistierte Daten angefordert werden müssen.

SAM soll eine Serveranwendung implementieren, welche im Rahmen von Big Data Anwendung finden kann. Das bedeutet, dass es mit großen Datenmengen und vielen Anfragen umgehen können muss.

Die Anforderung, einen Webserver mit integrierter Anwendungslogik zu implementieren, wird durch Node.js erfüllt. Es kann eine Software umgesetzt werden, die eine Schnittstelle für die mobile Anwendung anbietet und Zugriff auf die Datenbank erhält. Zudem bietet Node.js die Möglichkeit, nicht blockierende Aufrufe zu realisieren, wodurch eine hohe Erreichbarkeit garantiert wird. Callbacks erlauben es, Datenbankaufrufe zu delegieren und erst nach Abarbeitung darauf zu reagieren. Die modulare Architektur von Node.js bietet einen flexiblen Einsatz von Komponenten und Frameworks. Aus den genannten Gründen wird Node.js als Plattform für die Implementierung des Webservers eingesetzt. Es bietet alle notwendigen Methoden, um einen Webserver und die Serverlogik integriert in einer Anwendung implementieren zu können.

2.2.2 Definition von Kreuzungen

Die Kreuzungen sind ein wichtiger Bestandteil dieser Applikation. Um die Anzahl von aufgezeichneten GPS-Punkten zu reduzieren, sollen nur die markanten Punkte auf dem Streckenverlauf als Kontenpunkte persistent gespeichert und für die Auswertung eingesetzt werden. Somit ergeben sich folgende Anforderungen an eine Kreuzung:

1. aussagekräftige Position im Streckenverlauf
2. deterministisch feststellbar
3. eindeutiger Datenbestand

Beim ersten Punkt sollte beachtet werden, dass die Kreuzung das Fahrverhalten des Benutzers beeinflusst. Beispielsweise verlangt eine einfache Kreuzung von zwei Straßen ein langsames Abbremsen, um die Kreuzung einzusehen. Einfache T-Kreuzungen mit Fuß-, Fahrrad- oder Servicewegen sollen nicht als Kreuzung definiert werden, da diese wenig Einfluss auf das Fahrverhalten haben. Weitere Punkte, welche das Fahrverhalten beeinflussen, sind Ampeln, Bahnübergänge und Straßen mit Geschwindigkeitsbegrenzungen. Diese sollten auch als Kreuzungen definiert werden.

Mithilfe eines Algorithmus zur Erfassung von Kreuzungen müssen die Kreuzungen deterministisch feststellbar sein. Das bedeutet, dass bei mehrmaliger Fahrt die gleichen Kreuzungen extrahiert werden. Um dies zu ermöglichen, wird ein eindeutiger Datenbestand von Kreuzungen vorausgesetzt. Diese Eigenschaften bietet der Kartenbestand von Openstreetmap (OSM). Die Veränderungen im Datenbestand der OSM müssen jedoch beachtet werden und in die Applikation übertragen werden.

2.2.3 Algorithmus zur Erfassung von Kreuzungen

Die Kreuzungserfassung beschreibt das Extrahieren von bestimmten GPS-Punkten aus einer Liste von GPS-Punkten. Die extrahierten GPS-Punkte sollen sich auf Kreuzungen befinden. Diese GPS-Punkte werden anschließend zu einem Graphen verknüpft, um einfache Analyseoperationen darauf anzuwenden. Zur Lösung des Problems werden drei unterschiedliche Algorithmen entwickelt und getestet.

Straßennamenänderung mit Reverse Geocoding Mithilfe von Reverse Geocoding (siehe Abschnitt 3.7.1) wird der Name der nächstliegenden Straßen relativ zu dem aktuellen GPS-Punkt gefunden. Dieser wird mit dem letzten ermittelten Straßennamen verglichen. Wenn sich der Name ändert, dann wurde in eine neue Straße eingebogen. Nicht befahrene Abzweigungen von der aktuellen Straße werden somit nicht zuverlässig erkannt. Bei Überschreiten einer Kreuzung wird diese zweimal erkannt, sowohl beim Einfahren als auch beim Verlassen (vgl. Quellcode 1).

```
1 streetname = reverseGeocoding(GPS);
2 if (lastStreetname != streetname)
3 {
4     lastStreetname = streetname;
5     addCrossing(GPS);
6 }
```

Quellcode 1: Algorithmus: Straßennamenänderung mit Reverse Geocoding.

Web-API (GeoNames) Unter Verwendung der Web-API *findNearestIntersectionOSM* von GeoNames kann die nächstliegende Straßenkreuzung relativ zu dem aktuellen GPS-Punkt ermittelt werden. Hierbei werden alle Arten von Kreuzungen in der Nähe zurückgegeben, unter anderem Kreuzungen, die nicht die aktuelle Straße betreffen, sowie Kreuzungen von Fahrradwegen oder Fußwegen. Des Weiteren ist der kostenlose Service auf 30.000 Anfragen an einem Tag beschränkt. Bei 30 Usern können nur noch 1000 Punkte pro User und Tag überprüft werden (vgl. Quellcode 2).

```
1 crossingGPS = findNearestIntersectionOSM(GPS);
2 if (lastCrossingGPS != crossingGPS)
3 {
4     lastCrossingGPS = crossingGPS;
5     addCrossing(crossingGPS);
6 }
```

Quellcode 2: Algorithmus: Web-API (GeoNames).

Eigener Algorithmus Unter Benutzung eines Reverse Geocoding-Services und einer OSM-API wird ein eigener Algorithmus entwickelt. Mithilfe von Reverse Geocoding

wird der Name der nächstliegenden Straßen relativ zu dem aktuellen GPS-Punkt ermittelt. Mit einer OSM-API werden die einzelnen Nodes auf dieser Straße angefragt. Diese Nodes werden in einem weiteren Schritt analysiert und sortiert, um die relevanten Kreuzungen zu extrahieren. Aus der Liste der analysierten Kreuzungen wird die nächstliegende Kreuzung gefunden. Die umliegenden Kreuzungen können zur weiteren Analyse verwendet werden (vgl. Quellcode 10).

```

1 streetname = reverseGeocoding(GPS);
2 nodes = getAllNodes(streetname);
3 crossings = checkNodes(nodes);
4 crossingGPS = findNearestCrossing(crossings);
5 addCrossing(crossingGPS);

```

Quellcode 3: Eigener Algorithmus zur Kreuzungsfindung.

In der Tabelle 3 werden die drei erläuterten Algorithmen verglichen.

Tabelle 3: Vergleich der Algorithmen zur Kreuzzungserfassung.

Algorithmus	Straßennamenänderung	Web-API	Eigener Algorithmus
Erkennungsrate	gering	gut mit Fehlern	sehr gut
Programmieraufwand	normal	gering	hoch
Kosten	keine	kostenpflichtig	keine

Der Algorithmus „Straßennamenänderung“ löst das Problem mit einer einfachen Implementierung. Die dadurch erzeugten Fehlinformationen müssen danach jedoch noch bereinigt werden. Der Algorithmus „Web-API Geonames“ liefert gute Ergebnisse. Jedoch macht man sich von dem Web-Service abhängig, den man selbst nicht anpassen und optimieren kann. Des Weiteren sind die Anfragen an diesen Web-Service kostenpflichtig. Mit der Implementierung eines eigenen Algorithmus auf der Grundlage eines Geocoding-Service und einer OSM-API können alle relevanten Kreuzungen erfasst werden. Des Weiteren bietet der eigene Algorithmus die Möglichkeit, weitere Kreuzungen zwischenzuspeichern, um die Datenlast zu vermindern.

2.2.4 Wahl des Geocoding Frameworks

Für das inverse Mapping des Straßennamen zu einem GPS-Punkt soll ein Geocoding Framework verwendet werden. Die bekannten Online-Kartendienste wie Google Maps, Bing Maps, Here Maps und Yahoo! Maps bieten jeweils eine Geocoding-API an. Diese sind jedoch in der Benutzung eingeschränkt oder bieten nur einen kostenpflichtigen Service an.

Des Weiteren gibt es im Open-Source-Bereich zwei Geocoding Frameworks: Zum einen das weitverbreitete Geocoding Framework Nominatim, welches bei mehreren freien Webservices verwendet wird und außerdem als Such-Engine bei openstreetmap.com eingesetzt wird. Zum anderen das Geocoding Framework Gisgraphy, welches von David Masclet entwickelt wurde. Um eine optimale Performance beim Mapping der Straßennamen zu den GPS-Punkten zu erzielen, soll eine lokale Installation umgesetzt werden.

Im Folgenden sollen die beiden freien Geocoding Frameworks verglichen werden.

Tabelle 4: Vergleich von Geocoding Frameworks.

Geocoding Framework	Nominatim	Gisgraphy
Kosten	kostenlos (Open Source)	kostenlos (Open Source)
REST-Endpoint	vorhanden	vorhanden
Geocoding	vorhanden	vorhanden
Reverse Geocoding	vorhanden	vorhanden
OSM ID	Gebäudefläche	Straßen

Wie in der Tabelle 4 ersichtlich, unterscheiden sich die Geocoding Frameworks nur marginal. Beide Frameworks stehen kostenlos als Open-Source-Software zur Verfügung. Sowohl bei Nominatim als auch bei Gisgraphy sind REST-Endpoints für das Geocoding und das inverse Geocoding implementiert. Der entscheidende Unterschied liegt in der Funktionsweise des „Reverse Geocoding“. Nominatim liefert auf eine Reverse-Geocoding-Anfrage mit Angabe einer Postion als Antwort die dazugehörige Adresse. Für die Adressenfindung wird die kartografisch nächstliegende Adresse verwendet, wel-

che teilweise nicht mit der geografisch nächstliegenden Straße übereinstimmt. Folglich gibt Nominatim auch eine OSM-Way-ID zurück, welche die Fläche der Adresse identifiziert. Gisgraphy hingegen liefert als Antwort die geografisch nächstliegende Straße zurück (siehe Quellcode 30). Als eindeutiges Identifikationsmerkmal wird die OSM-ID der Straße mitgeliefert. Die Straße muss geografisch richtig und eindeutig festgestellt werden, um die einzelnen Kreuzungen auf einer Straße bestimmen zu können. Infolgedessen wird Gisgraphy als Geocoding Framework gewählt.

2.3 Messdatenspeicherung

2.3.1 Technische Daten des Backendservers

Um alle Anforderungen des Projekts erfüllen zu können, ist eine leistungsstarke Hardware notwendig. Parallel zu dem Webserver müssen verschiedene Dienste wie Geocoding Frameworks und eine Datenbank gehostet werden. Des Weiteren müssen die OSM-Daten lokal gespeichert werden. Daraus lassen sich folgende Spezifikationen für den Server ableiten:

1. ausreichend Arbeitsspeicher für den Cache der einzelnen Dienste
2. ausreichend Festplattenspeicher für OSM-Daten
3. Wahlmöglichkeit verschiedener Betriebssysteme für höchste Kompatibilität
4. Lesegeschwindigkeit des Festplattenspeichers nimmt zu Beginn eine untergeordnete Rolle ein

In Tabelle 5 werden verschiedene Optionen miteinander verglichen. Die erste Option stellt der Einsatz eines privaten Rechners dar. Damit sind jedoch zahlreiche Nachteile verbunden. Rechnungen für die Internetverbindung, Strom sowie die Hardware müssen beglichen werden. Investitionen für einen PC mit den geforderten Anforderungen sind nicht mit dem geplanten Geldrahmen zu ermöglichen. Des Weiteren muss ein Dynamic DNS⁶ eingerichtet werden, um über das Internet Zugriff auf den Server zu bekommen.

⁶Server zur Namensauflösung

Tabelle 5: Vergleich von Serverumgebungen.

Serverumgebung	privater PC	Server an der DHBW	virtueller Server
Kosten	hoch (Umrüstung, Betrieb)	kostenlos	25 €\Monat
Verfügbarkeit	90 % (eigene Verantwortung)	keine Erfahrung	99,6 % [7]
Erreichbarkeit	Dynamic DNS	VPN notwendig	eigene IP-Adresse
Geschwindigkeit	schwankend	keine Erfahrung	hoch

Ein Server der Dualen Hochschule Baden-Württemberg (DHBW) wird ebenso als Option ausgeschlossen, weil dieser nur im Universitätsnetz erreichbar ist und von außerhalb Virtual Private Network (VPN)⁷ notwendig ist. Dadurch ginge Flexibilität bei der Arbeit verloren, da die Entwicklungsgeräte diese Technik unterstützen müssen. Zudem soll die App auch ohne VPN mit dem Server Daten austauschen können.

Die für unsere Anforderungen passendste Lösung stellt ein vServer⁸ der netCup GmbH dar. Dieser bietet neben der geforderten Hardware günstige Konditionen. Die relevanten Merkmale des Servers zeigt folgende Übersicht auf [8]:

- RAM: 16 GB
- Festplatte: 1TB
- Betriebssysteme: u.a. Ubuntu und CentOS 7
- Prozessor: Intel Xeon E5-2670V2 mit 4 Kernen
- eigene IP⁹-Adresse

⁷Engl. für „virtuelles privates Netzwerk“; stellt ein Netzwerk-Interface dar, welches es ermöglicht, eine gesicherte Verbindung von einem privaten Netzwerk in ein entferntes Netzwerk aufzubauen.

⁸Abk. engl. für „virtueller Server“; ist ein Server neben Weitern, welche auf der selben Hardware arbeiten.

⁹Normierte Ziffernfolge, über die jeder Rechner in einem Netzwerk eindeutig identifiziert wird.

2.3.2 Repräsentation der Daten

Wie in Kapitel 3.7 beschrieben wird das Straßennetz in Openstreetmap aus miteinander verbundenen Punkten und Linien aufgebaut. Die Verbrauchsdaten in SAM sollen jeweils zu den gefahrenen Streckenabschnitten gespeichert werden. Infolgedessen entsteht ein System von einzelnen Punkten, zwischen denen sich Verbindungen bilden, welchen jeweils die Verbrauchsdaten zugeordnet werden. Die stark verknüpften Daten stellen einen Graphen¹⁰ dar.

Für die Speicherung eines solchen Graphen eignen sich besonders Graphdatenbanken. In Graphdatenbanken werden die Konten und Kanten optimiert gespeichert, um schnelle Schreib- und Zugriffszeiten zu ermöglichen. Solche Graphdatenbanken werden auch in anderen Bereichen eingesetzt. Wie in [9] beschrieben, kann ein soziales Netzwerk aus befreundeten Nutzern mithilfe eines Graphen dargestellt werden. In [10] wird der Stoffwechsel im menschlichen Körper als Graph dargestellt. Die stark verknüpften Daten können in Graphdatenbanken optimal dargestellt und gespeichert werden. Des Weiteren vereinfachen sich die Abfragen bei einer Graphdatenbank. Im Gegensatz zu relationalen Datenbanken müssen keine umständlichen Join-Queries durchgeführt werden, welche den ganzen Graphen durchlaufen. Zudem bieten Graphdatenbanken ein hohes Maß an Flexibilität. Mathematische Graphen bieten von Natur aus die Möglichkeit, neue Kanten, Konten oder auch Subgraphen hinzuzufügen. Die additiven Möglichkeiten bieten auch Graphdatenbanken und können somit schnell an veränderte Anforderungen angepasst werden. Außerdem können auch die spezialisierten Graphalgorithmen wie Dijkstra- und A*-Algorithmus in Graphdatenbanken optimal eingesetzt werden, um die kürzesten Pfade zu finden [11] [12].

Als Graphdatenbank soll Neo4J von Neo Technology verwendet werden. Da es eine Open-Source-Graphdatenbank ist und sich seit mehreren Jahren in der Entwicklung befindet, bietet diese Graphdatenbank einige Vorteile. Neben einer ausführlichen Dokumentation wurde auch schon verschiedene Literatur [13], [14] hierzu veröffentlicht.

¹⁰Grafische Darstellung in Form von Knoten und verbindenden Linien (Kanten).

Außerdem spricht der Einsatz bei umsatzstarken Unternehmen wie Walmart, hp, TOM-TOM und ebay für die Entscheidung für Neo4J in der Studienarbeit SAM [15].

In der folgenden Abbildung 3 ist das Design für die in SAM verwendete Graphdatenbank illustriert.

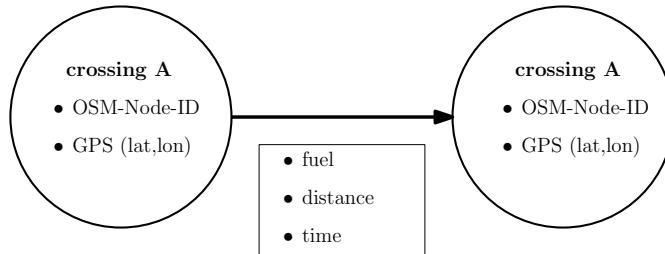


Abbildung 3: Schematische Darstellung des Graphdatenbank-Designs.

Ein Knoten in diesem Graphen soll eine Kreuzung darstellen. Diese Kreuzungen sind eindeutig über die OSM-Node-ID identifiziert. Die Integrität der Kreuzungs-Knoten kann auch noch mit einem Constraint überprüft werden.

Die Fahrstrecke zwischen einzelnen Kreuzungen soll als Kante in diesem Graphen umgesetzt werden. Der Kante werden die Eigenschaften der Fahrstrecke wie der Kraftstoffverbrauch (fuel), die Länge (distance) und die Zeitdauer (time) zugeordnet. Außerdem sollen gerichtete Kanten verwendet werden, da die Streckencharakteristiken sich mit der Fahrtrichtung unterscheiden können. Eine Steigung wirkt sich zum Beispiel beim Abwärtsfahren positiv auf den Kraftstoffverbrauch aus.

In diesem Graphdatenbank-Design werden die Fahrinformationen vor der ersten überfahrenen Kreuzung vernachlässigt. Diese Strecke beschreibt meistens die Fahrt von einem Parkplatz oder ähnlichem und liefert somit keine wichtigen Informationen. Ebenso verhält es sich auch mit den Fahrinformationen nach der letzten überfahrenen Kreuzung.

Die Kommunikation mit der Graphdatenbank soll abgekoppelt von der Implementierung der Algorithmen geschehen. Hierfür soll eine eigene Klasse implementiert werden, die die Kommunikation mit der Graphdatenbank regelt.

2.3.3 Zusammenfassung von Strecken zu Routen mittels Tags

Ziel der Arbeit ist die Bewertung einer Strecke im Vergleich zu alternativen Strecken. In der entwickelten Software des Prototypen werden dem Nutzer die von ihm gefahrenen Routen aufgelistet. Routen fassen Strecken mit gemeinsamem Start- und Zielpunkt zusammen. Das bedeutet, dass einer Route zahlreiche Strecken zugeordnet werden können. Um für den Auswertungsprozess diese Zuordnung zu garantieren, wird zu jeder Fahrt die Eigenschaft Tag eingeführt. Dieser Tag wird vom Nutzer selbst definiert. Dadurch kann zum einen eine Strecke einer Route zugeordnet werden. Zum anderen vereinfacht die Verwendung einer Zeichenkette dem Nutzer, eine Zusammenfassung seiner Routen auszuwählen.

In Abbildung 4 sind die Ausführungen visualisiert. Der Nutzer hat eine Route ausgewählt, zum Beispiel „zu Hause - Arbeit“. Dazu bekommt dieser alle gefahrenen Strecken, die nicht identisch sind, in einer Karte dargestellt.

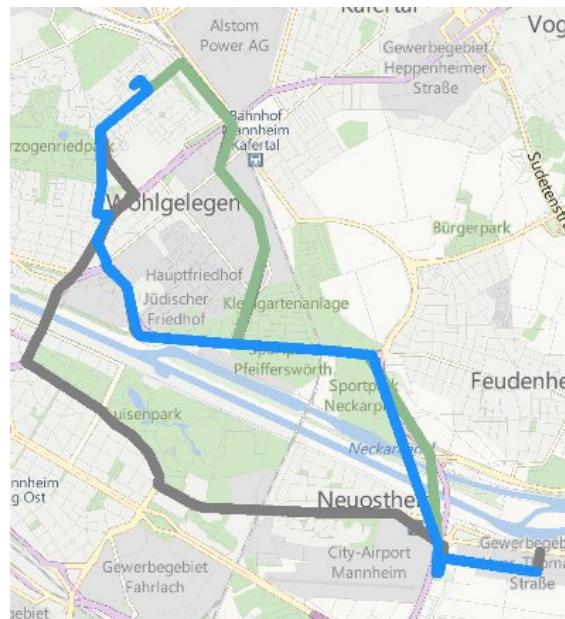


Abbildung 4: Darstellung einer Route mit drei Strecken.

Jede gefahrene Strecke wird separat in der Datenbank gespeichert. Das ist im Besonderen für die Weiterentwicklung von Bedeutung, um die Daten auch nach anderen Parametern auswerten zu können. Sind zwei oder mehr Strecken einer Route identisch,

werden diese vor dem Senden an den Client zusammengefasst. Dazu werden die Eigenschaften, die auf der Kante zwischen zwei Kreuzungen gespeichert sind, gemittelt. Der dazugehörige Algorithmus wird in Abschnitt 5.3.3 näher erläutert.

2.4 Messdatenfluss und Infrastruktur

Die Implementation der Anwendung erfordert den Zugriff auf Dienste wie Neo4J, Gisgraphy und Overpass (vgl. Abschnitt 3.7.3). Diese sind in den vorliegenden Fällen über eine öffentliche Application Program Interface (API) zugänglich. Dabei gelten jedoch Einschränkungen, wenn die Schnittstellen kostenlos genutzt werden sollen. Es wird zum Beispiel die Anzahl der Zugriffe pro Tag limitiert. Andere Dienste erlauben nur die Nutzung bis zu einer definierten Kapazität. Die erhobenen Einschränkungen sind für die Entwicklung und den produktiven Einsatz nicht zu akzeptieren. Aus diesem Grund werden die benötigten Drittdienste auf dem Backendserver gehostet. Die Einrichtung der Dienste auf einem systemeigenen Server erlaubt die uneingeschränkte Nutzung im Rahmen der Studienarbeit.

Die Abbildung 5 skizziert den Datenfluss innerhalb der SAM-Anwendung. Dazu wird im Folgenden das Konzept des Datenflusses beschrieben. Die Abbildung verdeutlicht den Aufbau und setzt die genannten Komponenten in Beziehung. Die Pfeile symbolisieren den Datenfluss der Sensorsignale und Informationen.

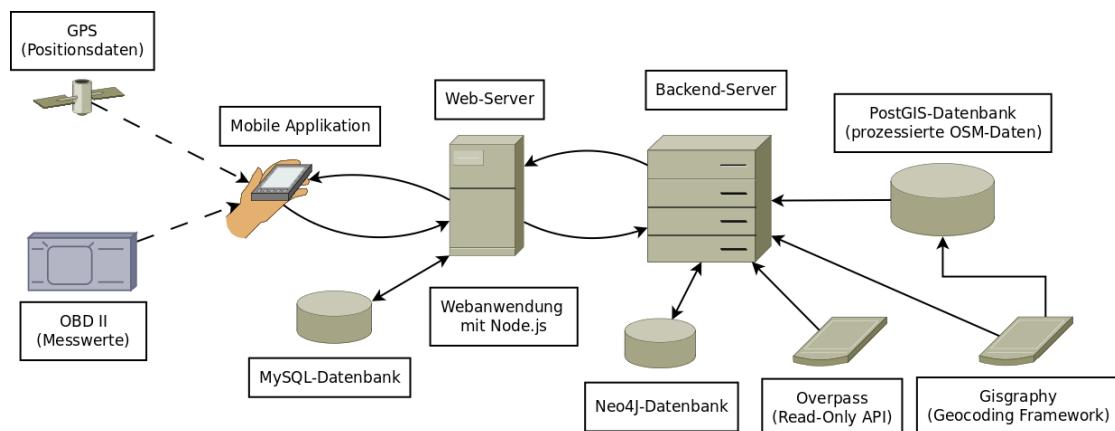


Abbildung 5: Skizze der Infrastruktur für SAM.

Zur Generierung der Messdaten werden zwei Signalquellen verwendet. Die aktuelle Position wird mittels GPS-Modul im Smartphone ermittelt. Mithilfe eines OBD-Adapters werden die Sensordaten des Fahrzeuges ausgelesen.

Ein Webserver implementiert die Serveranwendung. Diese nimmt Messdaten entgegen und ist für die Verarbeitung verantwortlich. Ebenso werden eingehende Anfragen mit angeforderten Daten beantwortet. Damit Informationen benutzerbezogen gespeichert werden können, wird eine MySQL-Datenbank (vgl. Abschnitt 5.2.4) zur Verwaltung der Nutzer verwendet.

Der Backendserver stellt die Plattform für Dienste zur Verfügung, welche zur Auswertung und Umwandlung der aufgezeichneten Daten notwendig sind. Dazu gehören Gisgraphy, Overpass und Neo4J als Datenbank. Diese werden für die Umwandlung von GPS-Punkten, die Abfrage nach OSM Informationen und die Persistierung der Daten benötigt. Gisgraphy greift bei Anfragen auf eine PostGIS-Datenbank zu. Diese Datenbank enthält das gesamte OSM-Kartenmaterial der Bundesrepublik Deutschland.

3 Grundlagen

Das Studium der Informationstechnik beinhaltet eine Vielzahl von Bereichen, die für die Informatik von Bedeutung sind. Für die Entwicklung einer Anwendung ist es notwendig, grundlegende Techniken anzuwenden sowie diese zu kombinieren. Um das Projekt besser nachvollziehen zu können, werden im Folgenden die Grundlagen erklärt, die für das Projekt SAM relevant sind.

Zum einen wird auf grundlegende Techniken eingegangen, welche die Kommunikation über das Internet ermöglichen. Zum anderen werden Grundlagen zur Interaktion mit dem Fahrzeugsensor und der App-Entwicklung beschrieben. Ebenso werden die im Backend verwendeten Technologien wie Node.js, Neo4J, Openstreetmap und Geocoding vorgestellt.

3.1 Kommunikation mit dem Fahrzeug

Neben den Global Positioning System (GPS)-Rohdaten werden während des Datenerfassungsprozesses Informationen zum aktuellen Kraftstoffverbrauch aufgenommen. Da dieser in vielen modernen Kraftfahrzeugen vom Bordcomputer erfasst wird, liegt es nahe, einen Weg zu finden, auf die im Fahrzeug vorliegenden Messdaten zuzugreifen. Dazu existieren viele herstellerspezifische Ansätze wie zum Beispiel das in Fahrzeugen der Volkswagen-Group zur Anwendung kommende Volkswagen COM Diagnose-System (VCDS). Die Nutzung einer proprietären Schnittstelle wie dieser würde sehr exakte Daten liefern, ist jedoch aufgrund hoher Lizenzgebühren und der Notwendigkeit, viele Herstellerstandards zu implementieren, nicht praktikabel.

Als herstellerübergreifender Standard für den Zugriff auf fahrzeuginterne Parameter existiert dagegen OBD, welches in Abschnitt 3.1.1 erläutert und schließlich in SAM implementiert wird [16].

3.1.1 On-Board-Diagnose

OBD ist ein Oberbegriff für eine Reihe von Standards für Fahrzeugdiagnosesysteme, welche seit 2001/2004 in allen PKW mit Ottomotor/Dieselmotor verbaut sind und Fahrzeugparameter in Echtzeit über die serielle Schnittstelle verfügbar machen.

In derzeit auf dem Markt erhältlichen Fahrzeugen kommen die Spezifikationen OBD-II, European On-Board-Diagnose (EOBD) oder Japanese On-Board-Diagnose (JOBD) zum Einsatz, deren Kommunikationsprotokolle sich auf Hardwareebene unterscheiden.

Das im Rahmen von SAM zur Verfügung stehende Testfahrzeug, ein Seat Ibiza 6J 1.6 TDI (Baujahr 2010), verwendet das ISO15765-4-Protokoll und erfüllt die EOBD-Spezifikation. Abbildung 6 zeigt die Platzierung der OBD-Schnittstelle im Testfahrzeug links unterhalb des Lenkrads im Sicherungskasten.



Abbildung 6: OBD-Schnittstelle eines Seat Ibiza 6J.

Die von OBD-Systemen bereitgestellten Funktionen werden in neun Modi unterteilt und umfassen unter anderem:

- aktuelle Sensormesswerte wie Drehzahl, Geschwindigkeit und Öltemperatur (Modus 1)
- Auslesen von Motorfehlercodes sowie Fehlerumgebungsdaten
- Status von überwachten Systemen
- Fahrzeugidentifizierung auf Basis von zum Beispiel der Fahrgestellnummer

Für SAM ist ein Zugriff auf die über den Modus 1 erreichbaren Daten notwendig. Die Kommunikation mit der OBD-Schnittstelle wird durch den in Abschnitt 3.1.2 beschriebenen ELM327-Mikrocontroller abstrahiert, sodass im Rahmen dieses Projektes technische Parameter der OBD-Schnittstelle sowie die EOBD-Kommunikationsprotokolle nicht bekannt sein müssen [17].

3.1.2 ELM327-Mikrocontroller

ELM327 ist ein OBD-Interpreter in Form eines Mikrocontrollers, der vom Unternehmen ELM Electronics produziert wird. Er unterstützt alle gängigen OBD-Protokolle und stellt deren Funktionen über eine einheitliche Schnittstelle bereit. ELM Electronics liefert den Controller in Form eines Integrated Circuit (IC)-Glieds aus, welches von anderen Herstellern mit einer seriellen Schnittstelle wie Universal Asynchronous Receiver Transmitter (UART) oder Universal Serial Bus (USB) ausgestattet und als OBD-Adapter auf dem Endkundenmarkt angeboten wird. Ferner existieren auch Versionen mit den kabellosen Standards Bluetooth oder Wireless Local Area Network (WLAN). Unabhängig von der Schnittstelle wird die Kommunikation mit dem Adapter über einen seriellen Socket abgewickelt und basiert auf dem American Standard Code for Information Interchange (ASCII). Sobald der Controller empfangsbereit ist, sendet er das Zeichen >. ELM-spezifische Kommandos, wie zum Beispiel das automatische Konfigurieren des OBD-Protokolls, sind als AT-Kommandos in der entsprechenden ELM327-Referenz dokumentiert [18].

Für SAM wird der Adapter mit folgenden AT-Kommandos initialisiert:

```

1 ELM:  >
2 SAM: AT Z      // Adapter zuruecksetzen
3 ELM: ELM327v1.5 >
4 SAM: AT SPO    // Automatische Protokollauswahl
5 ELM: OK >

```

Quellcode 4: Kommunikation mit dem ELM327 zur Initialisierung des Mikrocontrollers.

Zum Auslesen von Fahrzeugparametern werden OBD-PIDs genutzt. Das folgende Beispiel zeigt das Abfragen der aktuellen Motordrehzahl (Modi 01; OBD-PID 0C).

```

1 ELM:  >
2 User:   01 0C
3 ELM: 01 0C 0D 1B
4   >

```

Quellcode 5: Kommunikation mit dem ELM327 zum Abfragen der aktuellen Motordrehzahl.

In der Standardkonfiguration spiegelt der ELM327 zunächst die Anfrage und fügt die entsprechende Antwort darauffolgend ein. In der Dokumentation ist folgende Formel zum Ermitteln der Drehzahl aus der Antwort spezifiziert:

$$ByteA = 0D_{16} = 13_{10}$$

$$\begin{aligned} ByteB &= 1B_{16} = 27_{10} \\ Drehzahl &= \frac{(ByteA * 256) + ByteB}{4} \\ &= 839 \frac{U}{min} \end{aligned}$$

3.1.3 Zugriff auf den ELM327 Bluetooth RFCOMM

Die Datenerfassung soll bei SAM möglichst komfortabel für den Nutzer ermöglicht werden. Aus diesem Grund wird in Abschnitt 2.1 evaluiert, dass ein ELM327-Adapter mit Bluetooth zum Einsatz kommen soll, welcher mit einem Smartphone gekoppelt wird. Innerhalb von Bluetooth ist das Protokoll RFCOMM definiert, welches die Bluetooth-

Kommunikation auf eine serielle Schnittstelle abstrahiert. Bei der App-Entwicklung kann die Bluetoothverbindung somit wie eine RS232-Schnittstelle angesprochen werden. Folglich hat die Verwendung von Bluetooth keinen Einfluss auf die Nutzbarkeit von verfügbaren Programmiersbibliotheken, welche in der Regel auf eine serielle Kommunikation ausgelegt sind.

3.2 Runtime-App-Entwicklung für Windows Phone 8.1

Eine *Windows Runtime App* ist grundsätzlich eine Anwendung, die auf einem Microsoft Windows 8-System ausgeführt werden kann und über den Microsoft Store installiert und gewartet wird. Dabei werden folgende Typen unterschieden [19]:

- *Windows Store App*: Lauffähig auf Windows 8 Pro/RT für Computer und Tablets
- *Windows Phone Store App*: Lauffähig auf Windows Phone 8.1 für Smartphones
- *Windows Universal App*: Lauffähig auf allen Windows 8-Systemen, Graphical User Interface (GUI) wird gerätespezifisch entwickelt
- *Windows Phone Silverlight App*: Nutzt das Windows Phone Silverlight-Framework und ist lauffähig ab Windows Phone 7.1

Für SAM kommt die Verwendung der *Universal App* oder der *Phone Store App* in Frage, da nur diese auf die in Abschnitt 3.2.2 benötigten APIs zugreifen können.

Die Entwicklung setzt ein kostenloses Entwicklerkonto bei Microsoft sowie die Verwendung von Windows 8.1 voraus. Als Integrated Development Environment (IDE) wird Visual Studio Express 2013 Update 4 verwendet. Dieses enthält das Windows Phone Software Development Kit (SDK) und bietet ferner Werkzeuge zum Entwerfen, Testen und Debuggen der Anwendung. Verfügt der Entwicklungsrechner über eine Hyper-V-kompatible CPU, kann außerdem der Windows Phone Emulator zum Testen der Anwendung verwendet werden.

Für die Entwicklung können abhängig vom Anwendungsszenario folgende Programmiersprachen genutzt werden:

- C++/DirectX mit Direct3D für Spieleentwicklung
- Javascript mit WinJS oder HTML5 für Web-Frontendentwicklung
- .Net: C++, C# oder VB mit XAML für eine *Store/Universal App*
- .Net: C#, VP mit Silverlight für eine *Silverlight App*

Im Rahmen von SAM wird eine *Windows Universal App* in C# mit einem XAML-Markup verwendet, da dieser Typ von Microsoft empfohlen wird, wenn es sich bei der zu entwickelnden Anwendung nicht um ein Spiel oder das Frontend einer Webapplikation handelt. Für eine Universal-Anwendung stehen im Vergleich zu den anderen Applikationstypen alle .Net APIs zur Verfügung. Die Kompatibilität aller *Universal Apps* mit der zukünftigen Windows Phone-Version „Windows 10 Mobile“ wurde ferner angekündigt.

3.2.1 Komponenten einer Windows Runtime App

Eine *Windows Phone Universal App* setzt sich aus folgenden Komponenten und Konzepten zusammen.

Manifest Das Manifest ist eine XML-Datei, welche für die Veröffentlichung der Anwendung relevante Informationen enthält. Sie enthält folgende Merkmale:

- Herausgeber, Name, Beschreibung, Versionsnummer
- Unterstützte Bildschirmauflösungen und Betriebssystemversionen
- Benötigte Hardwarekomponenten (zum Beispiel Gyroskop)
- Erforderliche Zugriffsberechtigung auf andere Teile des Systems (zum Beispiel Kontakte, Medien, Ortung)

Bei der Erstellung eines neuen Projektes legt Visual Studio das Manifest an und konfiguriert es den Einstellungen entsprechend im Entwicklerkonto vor. Ferner ist es möglich, neben einer manuellen Anpassung der XML-Datei einen Assistenten zur Anpassung des Manifests zu nutzen.

Design Die GUI einer *Windows Phone Runtime*-Anwendung wird mithilfe von XAML, einer deklarativen Markup-Sprache, umgesetzt. Sie definiert die User Interface (UI)-Elemente wie Labels, Buttons und Textboxen, welche sich in die Windows Phone-Designsprache einfügen. Die GUI kann aus diesen Komponenten unter Verwendung bekannter Layout-Optionen¹¹ über den Grafikdesigner erstellt werden.

Dynamischere Ergebnisse lassen sich jedoch durch eine manuelle Bearbeitung des Markups erzielen. Für Änderungen am Markup wird wie in Abbildung 7 links neben dem Code eine Vorschau der GUI erstellt.

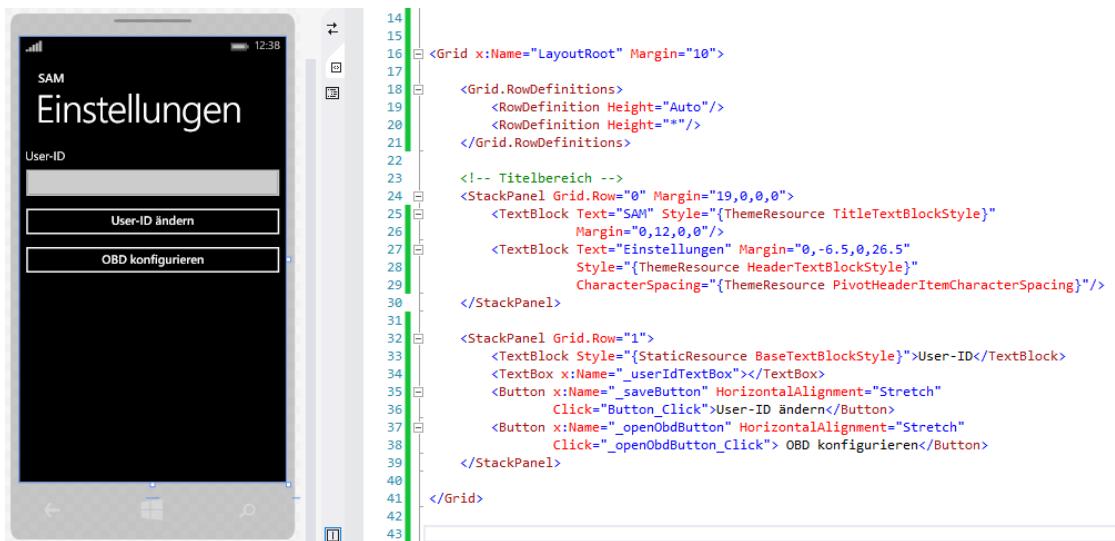


Abbildung 7: XAML-Markup und Simulation der SAM-Einstellungen.

Um ein systemübergreifendes Erscheinungsbild zu erhalten, können jedem GUI-Element Windows Phone-Designressourcen¹² zugeordnet werden.

Programmcode Jedem View wird automatisch eine gleichnamige C# Klasse zugeordnet, welche Aufgaben, die im Zusammenhang mit dem View stehen, implementiert. Dazu gehört der Umgang mit folgenden Standardsituationen:

¹¹Zum Beispiel Gridlayout oder Stacklayout

¹²Vordefinierte und anwendungsübergreifende Designkonfigurationen (zum Beispiel für Überschriften)

- Generieren und Initialisieren eines Views
- Verarbeitung von ausgelösten Events (zum Beispiel Reaktion auf das Klicken einer Schaltfläche)
- Organisation der Steuerelemente bei einer Bildschirmrotation
- Reaktion auf die Windows Phone-Zurücktaste
- Verhalten bei Minimierung und Fortsetzung der App

Die anwendungsspezifische Funktionalität wird darüber hinaus in viewunabhängigen C#-Klassen implementiert.

Datenbindung Microsoft empfiehlt für die Umsetzung des Zusammenspiels von Anzeigeelementen und den zu visualisierenden Daten die Verwendung der Datenbindung. Bei der Datenbindung handelt es sich mit dem *Model-View-ViewModel*-Muster um eine Variante des *Model-View-Controller*-Musters. Dieses sieht die Verknüpfung eines UI-Anzeigeelements mit einer Datenquelle bzw. einer Datensenke vor. Durch diese wird ein Steuerelement automatisch angepasst, sobald sich die verknüpften Daten ändern. Die Aktualisierung eines Views muss somit nicht manuell implementiert werden.

Neben Methoden und Attributen kann eine Klasse in C# sogenannte Eigenschaften enthalten. Diese bieten die Funktionalität von Gettern und Settern, können aber von anderen Klassen wie ein öffentliches Attribut verwendet werden. Folgendes Beispiel zeigt die Implementation einer Eigenschaft:

```

1  class Example{
2      private double _value;           // privates Attribut
3
4      public double Value {          // Eigenschaft
5          get {
6              return _value;          // Anpassungsschritte
7          }
8      }
9
10     public double getSeconds(){    // Getter
11         return _seconds;          // Anpassungsschritte
12     }
13 }
```

Quellcode 6: Gegenüberstellung einer Eigenschaft und eines Getters in C#.

Der Zugriff auf diese Eigenschaft kann von einer anderen Klasse wie folgt realisiert werden:

```

1 // Zugriff ueber Eigenschaft
2 var externValue = exampleInstance.Value;
3
4 // Zugriff ueber Getter
5 var externValue = exampleInstance.getValue();
```

Quellcode 7: Zugriff auf eine Eigenschaft in C#.

Neben einer besseren Codelesbarkeit ist der Hauptvorteil einer Eigenschaft, dass sie im Gegensatz zu einem Getter/Setter als Datenquelle/Datensenke für ein UI-Element dienen kann.

3.2.2 APIs

Die App wird als *Windows Universal App* in C# entwickelt und läuft somit in der Windows-Runtimeumgebung. Neben der Laufzeitumgebung CLR, in welcher C#-Code ausgeführt wird, besteht diese aus der Windows-API, welche einen abstrahierten Zugriff auf alle Systemfunktionen¹³ bietet. Einige Implementationen dieser API werden nativ als Maschinencode (*unmanaged*) anstatt in der Laufzeitumgebung (*managed*) ausgeführt.

In einer solchen Applikation kann ferner auf das .NET Framework (*managed*)¹⁴ zugegriffen werden. Es stellt über den System-Namensraum C#-Grundfunktionen wie Threading oder Collections bereit.

Im Rahmen von SAM kann die Windows-Runtime-API für folgende Funktionen genutzt werden:

¹³Dokumentation, siehe <https://msdn.microsoft.com/de-de/library/windows/apps/bb211377.aspx>

¹⁴Dokumentation, siehe <https://msdn.microsoft.com/de-DE/library/windows/apps/bb230232.aspx>

- Netzwerkfunktionen (Windows.Networking)
- Bluetooth-Kommunikation (Windows.Devices.Bluetooth)
- Ortung (Windows.Devices.Geolocation)
- Manipulation der GUI über C#-Sourcecode (Windows.UI)
- Zugriff auf Speicher (Windows.Storage)

Viele dieser APIs nutzen die C#-Unterstützung asynchroner Methodenausführungen.

Eine asynchrone Methode ist mit dem `async`-Operator gekennzeichnet und sollte Quelltext enthalten, dessen Ausführung nicht CPU-lastig ist¹⁵, aber bei einer synchronen Ausführung den restlichen Programmablauf blockieren würde. Eine solche asynchrone Methode gibt immer einen Task zurück, der mithilfe des `await`-Operators nach Ablauf der asynchronen Ausführung in den Rückgabewert der Funktion umgewandelt wird. Dabei wird der dem asynchronen Methodenaufruf folgende sequentielle Code so lange ausgeführt, bis auf das Ergebnis der asynchronen Methode referenziert wird [20]. Folgendes Beispiel soll dies verdeutlichen: Die GET-Funktion eines Servers wird asynchron aufgerufen und liefert nach Ausführung die Serverantwort als Rückgabewert. Währenddessen kann die Methode `doSomeStuff()` ausgeführt werden. Erst wenn eine Abhängigkeit zum Rückgabewert der asynchronen Ausführung entsteht, wird die Ausführung der Methode blockiert. Dies wäre bei der Zuweisung an `name` der Fall.

```

1 private async Task<String> getUserName(int userid)
2 {
3     HttpResponseMessage response = await HttpServer.GET(userid.ToString())
4         );
5     doSomeStuff();
6     String name = response.content.body;
7     return name;
}
```

Quellcode 8: Beispiel für asynchrone Ausführung einer Methode.

¹⁵Für CPU-lastige Aufgaben können Threads genutzt werden.

3.2.3 Debugging

Zum Debuggen der *Windows Phone Runtime App* muss entweder der Windows Phone-Emulator ausgeführt werden, oder ein als Entwicklergerät freigeschaltetes Windows Phone mit dem PC verbunden sein.

Der Emulator nutzt die Hyper-V-Virtualisierungsumgebung und stellt somit native Anwendungsperformance bereit. Er ermöglicht es ferner, alle Sensoren des Mobiltelefons anzusprechen. In Abbildung 8 ist dargestellt, wie ein GPS-Signal mit dem Windows Phone-Emulator simuliert werden kann.

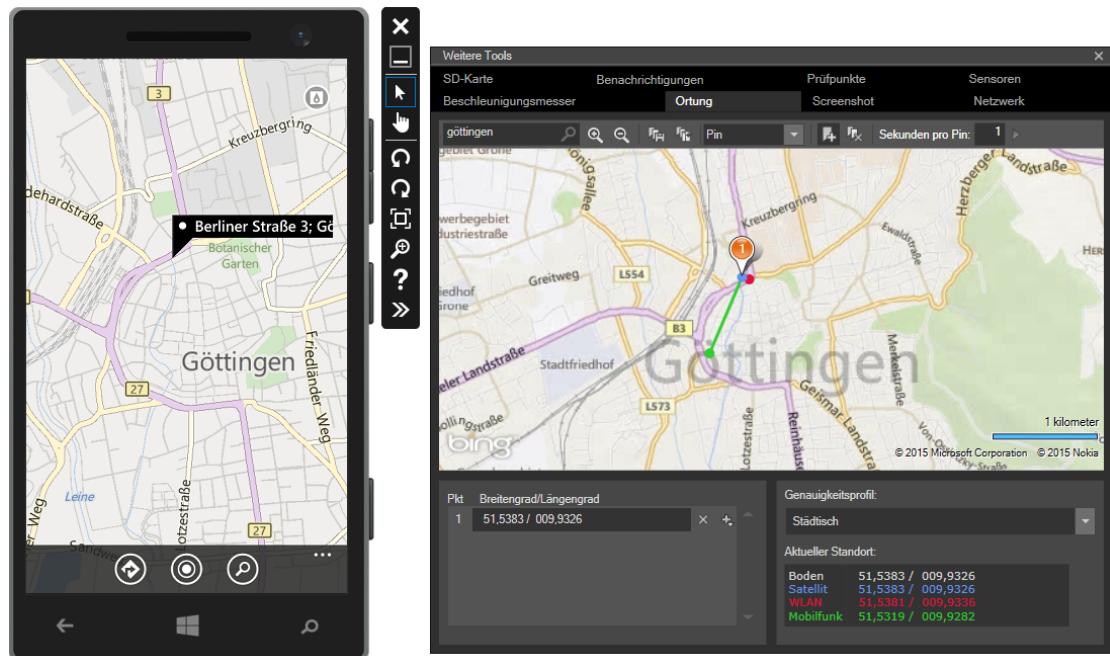


Abbildung 8: Simulation des Standorts im Windows Phone 8.1-Emulator auf einem 4 Zoll Smartphone.

Ferner bietet die IDE übliche Debugging-Werkzeuge wie Haltepunkte, Einzelschrittausführung und Speicheranalyse. Die Gesamtperformance der App lässt sich anhand von Analysewerkzeugen für Stromverbrauch, CPU-Auslastung und die Datenverbindung evaluieren.

3.2.4 Deployment

Windows Phone Runtime-Anwendungen im Produktivbetrieb können nur über den Windows Store installiert werden. In der Testphase ist es mit dem Windows Phone Deployment Tool möglich, ein App-Package (.appx) auf einem als Entwicklergerät registrierten Windows Phone bereitzustellen. Ein valides Package, welches die in Abschnitt 3.2.1 beschriebenen Komponenten beinhaltet, kann mit Visual Studio aus einem Projekt erzeugt werden.

3.3 Server-Client-Modell

Damit Daten von einem mobilen Gerät an einem anderen Ort gespeichert und verarbeitet werden können, wird die 3-Tier¹⁶-Architektur des Server-Client-Modells angewandt. Hierbei werden ein Client und ein Server definiert, die jeweils eine Schicht darstellen. Je nach Anwendung übernehmen sie unterschiedliche Aufgaben. In der Regel wird dabei eine Anfrage vom Client an den Server gesendet, der auf die Anfrage reagiert und antwortet.

Große Datenmengen müssen für ihre spätere Verwendung persistent gesichert werden. Dazu wird eine dritte Schicht eingeführt, welche für die dauerhafte Sicherung der Daten verantwortlich ist. Auf die Datenbank kann über eine Schnittstelle des Servers zugegriffen werden. Das in Abbildung 9 abgebildete Schema verdeutlicht die Architektur.

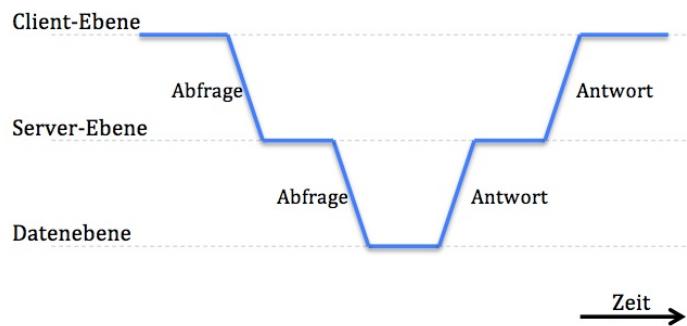


Abbildung 9: Schema der 3-Tier-Architektur.

¹⁶Engl. für „Schichten“

Entsprechend diesem Modell ist der Prototyp realisiert. Ein Smartphone fungiert als Client. Es sendet Anfragen oder Daten an den Webserver, welcher diese je nach empfangener Nachricht verarbeitet. Der Webserver kann dem Client (Smartphone) antworten oder selbst als Client auftreten, wenn er eine Anfrage an die Datenbank stellt. Letztere reicht die Antwort an den Server weiter, der erneut entscheiden kann, wie er mit diesen Informationen umgeht. Das ermöglicht die Kommunikation über mehrere Geräte und die schrittweise Verarbeitung der Daten.

3.4 Representational State Transfer

Folgender Absatz beinhaltet Informationen aus [21] und [22].

Representational State Transfer (REST) schafft die Möglichkeit, Web-Services zu realisieren. Dabei handelt es sich um ein Architektur-Modell, welches die Funktionsweise von Diensten im Internet beschreibt. Dabei trifft das vorgegebene Prinzip keine Aussagen über zu verwendende Protokolle. Auch die Formatierung (Repräsentation) einer Anfrage oder Antwort wird nicht vorgegeben, sie muss jedoch von Server und Client interpretiert werden können.

Das Modell definiert Ressourcen (d.h. Dateien oder Informationen), welche adressierbar und zustandslos sein sollen. Letzteres sagt aus, dass mit jeder Anfrage alle notwendigen Daten oder Parameter gesendet werden und es keine Abhängigkeit von vorherigen Anfragen gibt (*stateless*). Um die Adressierbarkeit einer Ressource zu ermöglichen, muss ihr eine eindeutige Adresse (Unique Resource Identifier (URI)) zugeordnet werden. Über den URI ist die Ressource zu erreichen. Des Weiteren wird definiert, dass der Zugriff auf Ressourcen mit festgelegten Methoden erfolgt. Dazu dienen im Besonderen die bekannten HTTP-Methoden GET, POST, PUT und DELETE, aber auch weitere. Ebenso beschreibt das Architekturnuster, dass eine Ressource, abhängig von der Anfrage des Clients, auf verschiedene Art und Weise dargestellt werden kann.

3.5 Node.js

Node.js soll einen Webserver mit integrierter Anwendungslogik implementieren. Dazu wird im folgenden Abschnitt auf die Modularität eingegangen. Des Weiteren wird der technische Hintergrund der Serverplattform dargestellt.

3.5.1 Struktur und Aufbau

Die Informationen für folgenden Absatz stammen aus [23].

Node.js wurde 2009 veröffentlicht und ermöglicht die Entwicklung von serverseitigen Anwendungen mit JavaScript. Ausgeführt werden die Anwendungen mit der Node.js-Runtime¹⁷, die für die geläufigen Betriebssysteme zur Verfügung steht. Aufgrund von integrierten Programm-Bibliotheken kann eine Anwendung einen Webserver implementieren. Damit wird für eine serverseitige Anwendung kein Webserver wie Apache HTTP benötigt. Der Quellcode einer Node.js-Anwendung wird mit der von Google entwickelten V8 Engine ausgeführt. Dabei handelt es sich um eine JavaScript-Implementierung (Umsetzung des Standards nach Ecma¹⁸), die die Anwendung zur Laufzeit übersetzt. Die Node.js-Umgebung bietet für die Entwicklung den integrierten Node Package Manager (NPM) an. Dieser ermöglicht es, Module für das Projekt einzubinden, zum Beispiel für den Zugriff auf eine Datenbank. Dadurch kann der Server um Funktionen erweitert werden, die sonst eine aufwendige Eigenentwicklung erforderlich gemacht hätten. Letzteres lässt sich im Zweifelsfall dennoch nachholen [24]. Der NPM verwaltet die Abhängigkeiten für SAM. Alle in der package.json-Datei definierten Abhängigkeiten werden mittels Paketnamen und Versionsnummer aufgelöst. Der Manager sucht auf der NPM Registry-Seite nach dem Modul und installiert es. Die Module stehen dem gesamten Projekt in dem Unterordner `npm_modules` zur Verfügung. Abhängigkeiten unter den Modulen werden ebenfalls durch den NPM aufgelöst. Folgender Sourcecode-Ausschnitt zeigt den Aufbau der package.json-Datei, welche die Module für das Projekt lädt.

¹⁷Engl. für „Laufzeitumgebung“

¹⁸Ecma ist eine Organisation zur Normung von Informationssystemen.

```

1 {
2   "name": "sam",
3   "description": "SAM Streckenabhaenige Fahranalyse",
4   "version": "0.0.1",
5   "dependencies": {
6     "log4js": "0.6.21",
7     "gps-distance": "0.0.3",
8     ...
9   }
10 }
```

Quellcode 9: Beispiel einer package.json-Datei.

Node.js setzt einen asynchronen Programmierstil um. Darunter ist zu verstehen, dass die Befehlsketten der Anwendung nicht nacheinander abgearbeitet werden. Wird eine Funktion aufgerufen, kehrt das Programm aus dieser Funktion direkt zurück und führt die Anwendung weiter aus. Die aufgerufene Funktion wird durch Node.js in einer Event-Reihe (*event queue*) verwaltet. Zusätzlich wird in der Regel der Funktion eine Callback-Funktion mitgegeben. Ist eine Methode abgearbeitet, wird die Callback-Funktion ausgeführt und die Funktion aus der Event-Reihe entfernt. Damit wird die Serveranwendung durch zeitaufwendige blockierende Funktionen (zum Beispiel Datenbankzugriffe) nicht blockiert (*non-blocking*).

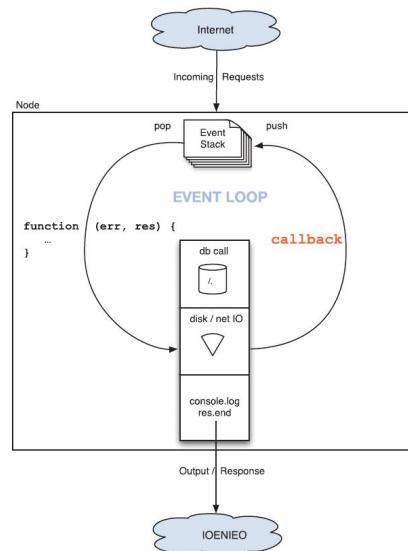


Abbildung 10: Event-Schleife (*event loop*) einer Node.js-Anwendung [23, Seite 53].

3.5.2 Entwicklung mit Webstorm und REST Client Add-On

Die Entwicklung der Serveranwendung erfolgt mit der IDE Webstorm von Jetbrains [25]. Diese wird für Studenten kostenfrei angeboten und bietet zur Entwicklung zahlreiche Vorteile. Zum einen kann die Anwendung direkt aus Webstorm gestartet werden. Zum anderen bietet sie die Möglichkeit, den Server in einem Debugging-Modus zu starten (in Abbildung 11 obere Symbolleiste). So kann zum Beispiel der Informationsfluss einer eingehenden Nachricht verfolgt und ausgewertet werden. Des Weiteren werden Änderungen am Sourcecode direkt übernommen, auch wenn die Anwendung gestartet ist (in Abbildung 11 Hauptfenster). Zudem wird ein Terminalfenster bereitgestellt, mit dem zum Beispiel der npm gesteuert werden kann. Ebenso werden Konsoleninformationen ausgegeben (in Abbildung 11 mittig unten). Darüber hinaus wird eine Dateiexploreransicht angezeigt, die das Navigieren durch Projektdateien vereinfacht (in Abbildung 11 links).

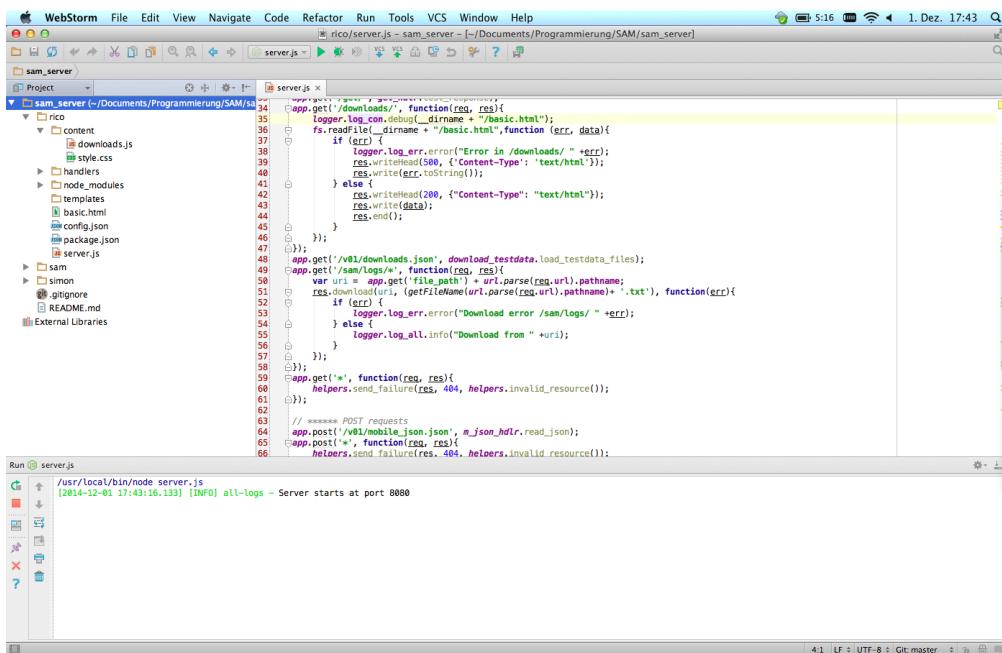


Abbildung 11: Webstorm-Arbeitsumgebung.

Zum Testen der Anwendung wurde das für den Browser Firefox erhältliche Add-On RESTClient [26] verwendet. Dieses stellt ein einfaches Interface zur Verfügung, welches

es erlaubt, REST-Methoden an eine angegebene URL zu senden. Für POST-Anfragen kann der Body der Nachricht mit Inhalt gefüllt werden. Die Antwort der Webanwendung wird in einem Antwortfenster dargestellt. Des Weiteren können die Header-Werte der Nachrichten verändert werden, was für das Testen der Anmeldung eines Nutzers wichtig ist.

3.5.3 Node.js und JavaScript

Javascript (JS) ist eine Skriptsprache, welche entwickelt wurde, um HTML-Seiten dynamisch und interaktiv gestalten zu können. Sie wird mit dem in Webbrowsersn integrierten Interpreter ausgeführt. Mit JS können verschiedene Aufgaben realisiert werden, wie die Manipulation des Document Object Model (DOM) und die Überprüfung von Formulareingaben. Im Laufe der Zeit hat die Sprache auch außerhalb des Webbrowsers Anwendung gefunden, zum Beispiel kann sie zur Entwicklung von Serveranwendungen eingesetzt werden. Node.js stellt eine Laufzeitumgebung von JS auf Servern dar. Dabei wird diese um weitere Funktionalitäten wie den Zugriff auf das Dateisystem und den Netzwerkverkehr erweitert. Im Folgenden soll auf einige Besonderheiten von JS eingegangen werden.

Datentypen Node.js bietet die Grunddatentypen `number`, `boolean`, `string` und `object`. Letzteres nimmt eine besondere Stellung ein, da die Datentypen `function`, `array`, `null` und `undefined` Sonderformen dieses Typs sind. Zwischen `null` und `undefined` wird dahingehend unterschieden, dass mit `null` ein Wert als „*is no value*“ definiert wird. `Undefined` zeigt an, dass der Wert einer Variablen nicht gesetzt ist.

Alle Zahlen sind 64 bit Gleitkommazahlen mit doppelter Genauigkeit, welche acht Byte Speicher belegen und im Dezimalsystem knapp 16 Stellen Genauigkeit erreichen. Bei Operationen mit kleinen Werten ist darauf zu achten, dass die Ergebnisse nur approximiert werden können. Daher muss bei diesen Rechnungen überprüft werden, ob das Ergebnis in einem erwarteten Bereich liegt. Für die Division mit 0 wird `infinity` als Ergebnis geliefert, welches im Gegensatz zu anderen Hochsprachen ein valides Ergebnis repräsentiert.

Booleans können den Wert `true` (wahr) oder `false` (falsch) annehmen. Die Werte `false`, `0`, „“ (leerer String), `NaN` (not-a-number), `null`, und `undefined` werden als `false` interpretiert. Alle anderen Werte gelten als `true`.

Mit Strings werden Unicode-Zeichen umgesetzt. Dass es keinen `char`-Typ gibt, wird dieser mit einem String der Länge eins realisiert. Des Weiteren können Strings mit einem ‘+’-Operator verbunden werden und die Länge ausgegeben werden (`.length`). Strings bieten darüber hinaus weitere Funktionen an.

In JS werden Objekte vielseitig eingesetzt. Sie bieten eine hohe Flexibilität, da Eigenschaften nachträglich ergänzt beziehungsweise entfernt werden können. Um Objekte zu erzeugen, gibt es verschiedene Möglichkeiten. Neben der literalen Syntax für Objekte (`var obj1 = new Object();` und `var obj2 = {};`) können Eigenschaften und Attribute definiert werden. Zusätzlich ist es möglich, ein Objekt über eine Konstruktorfunktion zu deklarieren.

Objekte in JS können durch die Verwendung von Anführungszeichen für jedes Attribut zum textbasierten Austausch genutzt werden. Liegt ein Objekt in beschriebenem Format vor, wird es als JavaScript Object Notation (JSON) bezeichnet.

Typenvergleich und -umwandlung Variablenwerte können mit dem `==`-Operator verglichen werden. Zusätzlich gibt der Operator `===` die Möglichkeit, neben dem Wert auch zu kontrollieren, ob der Datentyp identisch ist. Folgendes Beispiel soll das Verhalten verdeutlichen:

```
123 == '123'    true
123 === '123'   false
```

Eine Besonderheit gilt es zu beachten: Wenn *Primitives* mit Konstruktoren angelegt werden (zum Beispiel `var x = new Number(123)`), gibt die Funktion `typeof` den Wert `object` zurück. Das hat zur Folge, dass die Überprüfung mit dem `==` stets `false` liefert.

Funktionen JS setzt eine funktionale Programmierung um. Dieses Programmierparadigma gibt vor, dass das Programm ausschließlich mit Funktionen realisiert wird. Funktionen können wie in anderen Programmiersprachen umgesetzt werden, müssen jedoch keinen Rückgabetypen bestimmen. Es sollte dabei auf die korrekte Übergabe der Parameter geachtet werden, da diese zur Laufzeit nicht validiert werden.

Es ist möglich, Funktionen ohne Namen zu definieren, wie folgendes Beispiel zeigt. Diese werden auch „anonyme Funktionen“ genannt.

```
1 var calc = function (a, b) {  
2     return a + b;  
3 }  
4 > calc(3,7);  
5 10
```

Quellcode 10: Anonyme Funktionen.

Klassen, Prototypen und Vererbung Objektorientierte Programmierung wird in JS mit `functions` realisiert, eine explizite `class`-Anweisung wird nicht angeboten. Dennoch lassen sich zu objektorientierten Sprachen ähnliche Klassenkonzepte umsetzen.

Ein neues Objekt der Klasse kann mit dem `new`-Operator erzeugt werden. Im Anschluss stehen die in der Funktion definierten Methoden und Attribute zur Verfügung. JS hat die Eigenschaft, dass der Klasse nachträglich beliebige Eigenschaften und Methoden hinzugefügt werden können. Zu beachten ist, dass für jede neue Instanz sämtliche Funktionen neu angelegt werden.

In JS angelegte Objekte erben Eigenschaften und Methoden mittels eines Zeigers von einem Objekt-Prototypen. Dieser besitzt Standardmethoden, welche bei der Erzeugung eines neuen Objektes um die angegebenen Parameter (Schlüssel, engl. `keys`) ergänzt werden. Wird ein Key des Objektes nicht gefunden, wird durch den JS-Interpreter im Prototyp-Objekt gesucht. Wenn es hier ebenfalls nicht vorhanden ist und das Ende der Prototyp-Kette erreicht ist, wird `undefined` zurückgegeben.

Ein Objekt muss jedoch nicht direkt auf das Prototyp-Objekt zeigen, sondern kann von einem anderen Objekt erben. Somit kann es auch dessen Methoden verwenden. Dabei besteht der Unterschied zum Erzeugen neuer Instanzen darin, dass diese auf die selben Methoden ihrer Prototyp-Objekte referenzieren und nicht neu anlegen. Dadurch wird das Programm effizienter [27].

3.6 Graphen und Graphdatenbanken

In der realen Welt ist ein großer Anteil der Informationen miteinander verknüpft. Zum Beispiel führen wir Freundschaften mit verschiedenen Personen. Hierbei besteht zwischen diesen Personen eine symbolische Verbindung. Aus mathematischer Sicht entsteht ein *Property-Graph*¹⁹. Ein Property-Graph besteht aus Knoten, welche hier die Personen darstellen, und Kanten, welche die Freundschaften symbolisieren. Sowohl Knoten als auch Kanten können Schlüssel-Wert-Paare zugeordnet werden, um weitere Informationen zu speichern (vgl. Abbildung 12). Eine Kante ist eine gerichtete Verbindung zwischen einem Start- und einem Endknoten. Eine solche gerichtete Verbindung beschreibt die Beziehung in einer bestimmten Richtung zwischen zwei Knoten. Zusätzlich können die Knoten und Kanten mit einem beschreibenden Namen, einem so genannten Label, versehen werden. Diese Labels strukturieren den Graphen und geben den einzelnen Knoten und Kanten einen semantischen Kontext. Viele Abfragesprachen können diese Labels verwenden, um nicht den ganzen Graphen durchlaufen zu müssen und somit die Effizienz der Abfragen zu steigern.

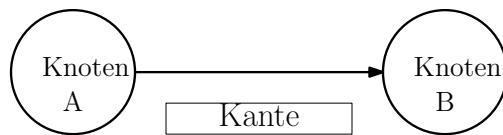


Abbildung 12: Schematische Darstellung eines Property-Graphen.

Eine Graphdatenbank ist ein Datenbank-Management-System mit den CRUD-Methoden (Create, Read, Update, and Delete), welches die Daten in Form eines Property-

¹⁹Gerichteter Multigraph mit Schlüssel-Wert-Paaren

Graphen speichert. Viele Graphdatenbanksysteme speichern die Graphen nativ, um eine höhere Performance zu erreichen. Die Daten eines Graphen können auch serialisiert werden und danach in einer relationalen oder objektorientierten Datenbank gespeichert werden [14].

3.6.1 Graphdatenbank Neo4J

Wie in Abschnitt 2.3.2 evaluiert, wird die Graphdatenbank *Neo4J* von Neo Technology zur Speicherung der gesammelten Daten eingesetzt. Die Graphdatenbank Neo4J wurde in Java entwickelt und ist eine der ältesten Graphdatenbanken nach dem Resource Description Framework (RDF). Zur Optimierung der Schreibvorgänge wurde eine eigene, optimierte Persistenzschicht entwickelt. Dieser Persistenzmechanismus verwendet die Java Non-blocking I/O (NIO)-Schnittstelle, um die Knoten- und Kanteninformationen in Blöcken mit einer festen Größe auf der Festplatte zu speichern. Somit wird die Speicherung in den unteren Schichten optimiert.

Des Weiteren verfügt die Graphdatenbank Neo4J über eine Transaktionssicht. Hierfür entwickelte Neo4J eine eigene Transaktionsinfrastruktur, welche die Schnittstellen der Java Transaction API (JTA) implementiert. Somit können die bekannten ACID-Eigenschaften garantiert werden, welche atomare Datenbankoperationen gewährleisten. Diese werden benötigt, da Änderungen an dem Graphen in einem bestimmten Kontext gesehenen und somit ganz oder gar nicht durchgeführt werden müssen [13].

Zur Kommunikation mit der Graphdatenbank steht eine REST-API zur Verfügung. Mit Hilfe von verschiedenen REST-Endpoints können Änderungen an der Graphdatenbank durchgeführt werden. Außerdem dient die Abfragesprache *Cypher* zur Manipulation der Datenbank (siehe Abschnitt 3.6.2). Für die Benutzung der Abfragesprache Cypher stehen zwei verschiedene Endpoints zur Verfügung. Der „Legacy Cypher HTTP endpoint“ kann nur eine Transaktion pro Hypertext Transfer Protokoll (HTTP)-Request durchführen. Mit dem „Transactional Cypher HTTP endpoint“ kann eine Transaktion auf mehrere HTTP-Requests aufgeteilt werden. Dies erlaubt innerhalb einer Transaktion einzelne Änderungen an der Graphdatenbank, die von den anderen Benutzern isoliert werden.

3.6.2 Abfragesprache Cypher

Cypher ist als eine deklarative Abfragesprache für die Graphdatenbank Neo4J entwickelt worden. Die Abfragesprache dient sowohl zum Abfragen als auch zum Aktualisieren von Daten in der Graphdatenbank. Die Abfragesprache ist jedoch auf das Lesen von Daten optimiert. Mithilfe von Cypher kann der Graph nach Mustern durchsucht werden. Diese Muster werden für Menschen verständlich und für Maschinen lesbar in ASCII-Art abgebildet.

```
1 MATCH (s)-[r]-(e)
2 RETURN s;
```

Quellcode 11: Beispiel eines Cypher Statements.

Der Quelltext 11 zeigt eine einfaches Cypher Statement. Die Knoten werden mit geschlossenen, runden Klammern dargestellt. Die Kante zwischen zwei Knoten wird mit einem Pfeil „ \rightarrow “ symbolisiert. Die Informationen zu den Kanten stehen in geschlossenen, eckigen Klammern. In diesem Cypher Statement stehen auch die beiden Schlüsselworte MATCH und RETURN. Im Folgenden werden die meist verwendeten Schlüsselworte aufgezählt und erläutert:

- MATCH – Angabe von Mustern, nach denen gesucht wird
- WHERE – Einschränkung der Ergebnismenge
- RETURN – Projektion der Ergebnismenge
- DELETE – Löschen der Ergebnismenge
- CREATE, MERGE – Erzeugen von Knoten und Kanten
- ORDER BY – Sortierung der Ergebnismenge
- SKIP, LIMIT – Reduzieren der Ergebnismenge

Des Weiteren bietet die Abfragesprache Cypher auch imperative Operationen, wie zum Beispiel EXTRACT, FILTER oder REDUCE, an. Mit solchen Operationen können Funktionen auf der Graphdatenbank ausgeführt werden.

Eine weiterer Aspekt bei der Benutzung der Abfragesprache Cypher ist die Verwendung von Parametern. Durch den Einsatz von Parametern können bereits getätigte Anfragen in einem Cache gespeichert werden. Außerdem unterbindet die Benutzung von Parametern die Injektion von Abfragen durch den Nutzer.

3.7 Openstreetmap

Openstreetmap ist eine internationale Non-Profit-Organisation, die das Erzeugen, Verteilen und Vergrößern eines frei nutzbaren Geodatenbestandes verwaltet. Gegründet wurde das OSM-Projekt im Jahr 2004 von Steve Coast. Die Geodaten, aus denen sich die Karten generieren lassen, werden mithilfe von drei verschiedenen Elementtypen (node, way, relation) gespeichert. Durch eine eindeutige ID können die jeweiligen Elemente identifiziert werden. Jedem Elementtyp können weitere Zusatzinformationen (tags) in Form von Schlüssel-Wert-Paaren zugeordnet werden. Als Datenbank verwendet OSM eine PostgreSQL-Datenbank. Hierbei werden jedoch weder die geometrischen Typen von PostgreSQL noch die Erweiterung PostGIS, die geografische Funktionen und Objekte hinzufügt, verwendet. Die Repräsentation der Geodaten in der Datenbank findet mit den oben erwähnten Elementtypen statt [28]. In der folgenden Abbildung 13 werden die Elementtypen schematisch dargestellt.

Ein Punkt (engl. *node*) beschreibt in OSM eine geografische Koordinate. Die Eigenschaften dieses Punktes werden mit Attributen beschrieben. Eine OSM-Node kann einen speziellen geografischen Punkt wie eine Sehenswürdigkeit beschreiben. Außerdem stellen die OSM-Nodes die Zwischen- sowie Start- und Endpunkte von Linien dar. Diese Eigenschaft wird im weiteren Verlauf dieser Arbeit zur Beschreibung von Kreuzungen verwendet.

Die Linien (engl. *ways*) in OSM stellen einen Linienzug dar, der aus einzelnen Teilssegmenten besteht. Die Verbindung der einzelnen Teilssegmente findet mit den eben erwähnten OSM-Nodes statt. Mit diesen OSM-Ways werden alle möglichen geografischen Verläufe, wie zum Beispiel Eisenbahnen, Flüsse sowie alle Arten von Straßen,

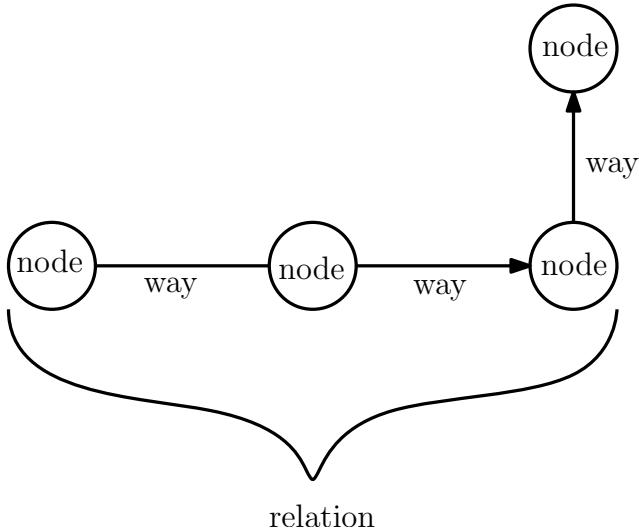


Abbildung 13: Schematische Darstellung der Elementtypen von Openstreetmap.

beschrieben. In dieser Anwendung werden die OSM-Ways verwendet, um befahrene Straßen eindeutig zu identifizieren. Außerdem gibt es noch den Elementtyp Relation, welcher eine Gruppierung von OSM-Nodes oder OSM-Ways beschreibt. Mithilfe von Relations sollen Beziehungen zwischen den Objekten beschrieben werden. Die Gruppierung von Autobahnteilstücken ist ein Beispiel für eine Relation. Zur Erzeugung wurden die Geodaten aus GPS-Spuren extrahiert, aus Luftbildern abgezeichnet oder von anderen Geodatenbanken importiert. Diese Geodaten können danach zur Karten erstellungen, Routenberechnung und zur Navigation verwendet werden. Verschiedene APIs stehen zur Verfügung, um die Geodaten abfragen und verändern zu können. Im Folgenden werden einige Schnittstellen beschrieben.

Die OSM Foundation stellt verschiedene APIs zur Benutzung der OSM-Daten zur Verfügung. Die *APIv6* stellt eine Schnittstelle zur einfachen Bearbeitung der OSM-Daten in der Live-Datenbank dar. Diese API limitiert die geografische Größe der angefragten OSM-Daten.

Des Weiteren bietet OSM eine Extended API (XAPI) an. Diese Schnittstelle stellt eine nur lesende Version der Standardschnittstelle dar und verwendet eine Spiegel-Datenbank der primären OSM-Datenbank. Neben erweiterten Abfrage- und Suchmöglichkeiten lassen sich auch größere Gebiete abfragen. Die angefragten Daten werden über eine REST-Schnittstelle im XML-Format ausgeliefert.

Außerdem besteht die Schnittstelle *Overpass API*. Diese Schnittstelle wird von Roland Olbricht entwickelt und ermöglicht selektive Anfragen an die OSM-Daten. Mithilfe einer eigenen Abfragesprache kann man die OSM-Daten nach Informationen wie Keys, Tags, Objekttypen oder kombinierten Informationen selektieren. Somit stellt diese API eine Möglichkeit dar, komplexe Anfragen an die OSM-Datenbank zu stellen [29].

3.7.1 Geokodierung

Bei der Geokodierung wird einer Lokationsangabe eine Georeferenz zugeordnet. Das bedeutet, dass einer postalischen Adresse eine geografische Koordinate zugeordnet wird. Dieser Zusammenhang wird verwendet, um textuelle Lokationsangaben in einer Karte darstellen zu können (vgl. Abb. 14). Der eingegebene Text wird geparsst, in eine standardisierte Form umgewandelt, um dann eine Anfrage an einen Geocoding-Service zu stellen. Als Antwort erhält man eine GPS-Koordinate.

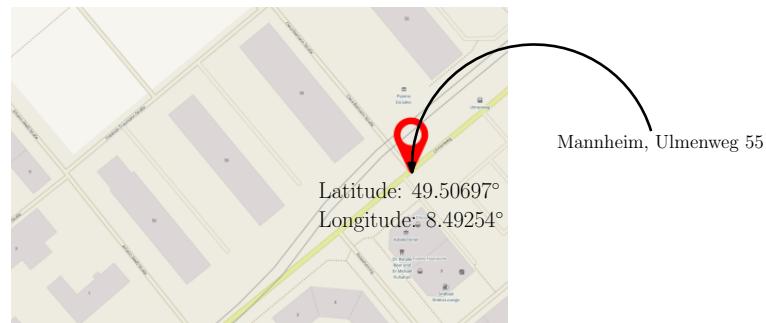


Abbildung 14: Geocoding Beispiel: Eingabe „Mannheim, Ulmenweg 55“ wird zu einer geografischen Koordinate umgewandelt.

Im Sinne von OSM findet ein Geotagging (Geocoding) statt. Hierbei wird die geografische Koordinate mit weiteren Attributen ergänzt. Infolgedessen können diese Daten räumlich dargestellt und analysiert werden. Außerdem werden Georeferenzen benutzt, um geometrische Verzerrungen in Datensätzen zu entfernen oder um unterschiedlich skalierte bzw. orientierte Datensätze aneinander anzupassen [30].

Inverse Geokodierung Die inverse Geokodierung beschreibt den gegenteiligen Vorgang. Hierbei wird einer geografischen Koordinate eine textuelle Lokationsangabe zugeordnet. Somit können zu einer gegebenen Geokoordinate der Straßename, die Stadt

sowie das Land bestimmt werden. Diese Zuordnung wird verwendet, um eine geografische Koordinate für den Anwender lesbar und verständlich darzustellen (siehe Abbildung 15).

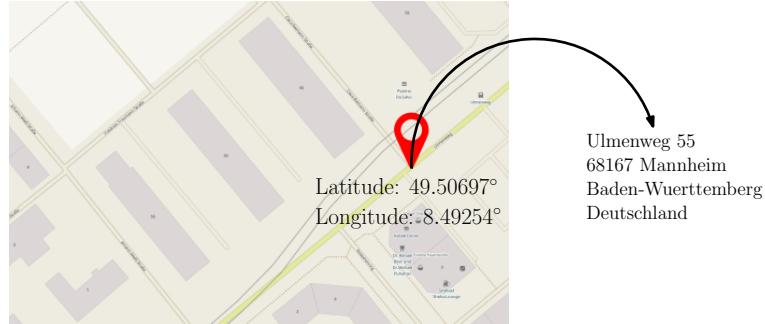


Abbildung 15: Inverses Geocodierungsbeispiel: Der geografischen Koordinate wird eine Adresse zugeordnet.

3.7.2 Geocoding-Framework Gisgraphy

Zur inversen Geokodierung wird das freie Geocoding-Framework *Gisgraphy* verwendet. Das in Java implementierte Framework bietet verschiedene REST-Webservices für die Geokodierung. Neben der inversen Geokodierung und der Geokodierung stehen auch noch ein Adressen-Parser und eine Volltextsuche zur Verfügung. Die Rohdaten werden aus den Datenbanken von OSM und Geonames extrahiert und prozessiert. Diese prozessierten OSM-Daten werden in einer PostGIS-Datenbank gespeichert. Die Volltextsuche wird mithilfe von SolR umgesetzt. Die Entscheidung zu diesem Framework wird in Abschnitt 2.2.4 beschrieben.

3.7.3 Openstreetmap-Schnittstelle Overpass

Für Anfragen an die OSM-Daten wird die Overpass-API verwendet. Mithilfe der eigenen Abfragesprache können auch komplexe Abfragen, wie zum Beispiel die Kreuzungsfindung, umgesetzt werden. Overpass speichert die OSM-Daten in seiner eigenen Datenbank *Template DB*. Hierbei werden die geografisch nah beieinander liegenden

Nodes in einem Datenblock auf der Festplatte gespeichert. Bei Anfragen werden meist nur OSM-Daten aus einer bestimmten geografischen Region angefragt, sodass sich mit dem technischen Aufbau der Template DB die Antwortzeiten stark minimieren.

Die in C++ implementierte API bietet eine Schnittstelle für selektive Anfragen auf die OSM-Daten. Des Weiteren besitzt Overpass eine eigene Abfragesprache *Overpass QL*. Mit dieser Abfragesprache können die OSM-Daten nach Informationen wie Keys, Tags und Objekttypen selektiert werden. Ein Abfrage in der Overpass QL ist in einzelne Statements aufgeteilt. Die Zwischenspeicherung von einzelnen Datensätzen findet in sogenannten Sets statt, welche mit einem Punkt eingeleitet werden. Ein Statement liest ein solches Set ein und schreibt das generierte Ergebnis wieder in ein Set. Die Statements lassen sich in fünf Gruppen einteilen:

1. Query Statements – Anfragen an die Datenbank
2. Eigenständige Statements – Ohne Anfrage an die Datenbank
3. Filter Statements – Filtern von Ergebnissen
4. Block Statements – Gruppieren von Statements und Schleifen
5. Einstellung Statements – Setzen bestimmter Einstellungen

3.8 Lizenzen

Openstreetmap (OSM) Die OSM wird unter der Lizenz „Open Database Licence (ODbL) 1.0“ verteilt. Produkte, die aus einer Datenbank mit einer solchen Lizenz entstehen, müssen mit einem Vermerk zu dieser Lizenz gekennzeichnet werden. Die abgeleiteten Daten dürfen verkauft werden. Jedoch müssen diese Daten mit der „Open Database Licence (ODbL) 1.0“ lizenziert sein. Infolgedessen dürfen die Benutzer ihre Daten ohne Bezahlung weiter veröffentlichen [31].

Overpass Der Sourcecode von Overpass ist mit der „GNU Affero General Public License“ lizenziert. Die „GNU Affero General Public License“ stellt eine modifizierte Version der GNU General Public License (GPL) Version 3 dar. In der Affero-Version wurde das ASP-Schlupfloch geschlossen. Dadurch konnte man seine veränderte GLP-Software im Hosting anbieten und musste keinen Quellcode veröffentlichen [32]. Die Ergebnisse von Overpass beruhen auf den OSM-Daten und sind somit unter der Lizenz *ODbL* lizenziert.

Gisgraphy Das Projekt Gisgraphy wird mit der Lizenz GNU Lesser General Public License (LGPL) veröffentlicht. Diese Lizenz erlaubt das Einbinden der Software, ohne den gesamten Quellcode der eigenen Software zu veröffentlichen. Lediglich der LGPL-lizenzierte Teil muss den Anwendern zur Verfügung gestellt werden. Weiterhin darf LGPL-lizenzierte Software auch in proprietärer Software verwendet werden [33].

Node.js Die Serverplattform Node.js wird unter der MIT-Lizenz veröffentlicht, welche am Massachusetts Institute of Technology (MIT) entstanden ist. Diese Lizenz schränkt die Benutzung und den Gebrauch der lizenzierten Software kaum ein. Nur der Urhebervermerk muss erhalten bleiben und es findet ein Haftungsausschluss statt [34].

Neo4J Die Graphdatenbank ist sowohl mit der GNU General Public License version 3 (GPLv3) als auch mit der GNU Affero General Public License lizenziert. Die GPLv3 erlaubt das kostenlose Nutzen, Studieren, Ändern und Verbreiten. Außerdem ist es eine Copyleft-Lizenz. Das bedeutet, dass Ableitungen von GPL-lizenzierten Werken unter der gleichen Lizenz veröffentlicht werden müssen [35]. Die GNU Affero General Public License modifiziert die GPLv3, um das ASP-Schlupfloch zu schließen.

Aus dem Studium der Lizenzen der einzelnen Komponenten und Services kann man festhalten, dass bei einem Verkauf des Services die gesammelten Daten der Open Database Licence unterliegen. Das bedeutet, dass Daten für den Anwender zum Download bereitstehen müssen. Des Weiteren muss der Service einen Urhebervermerk auf Node.js aufweisen. Alle weiteren Lizenzen beziehen sich nur auf Änderungen am Quellcode der jeweiligen Komponenten und Services. Da die Services ohne Veränderung genutzt werden, finden diese Lizenzen keine Anwendung.

4 Umsetzung des Frontends

Das Frontend von SAM wird, wie in Abschnitt 2.1 erörtert, als Smartphone-Anwendung auf der Microsoft Windows Phone 8.1-Plattform umgesetzt. Im folgenden Kapitel wird die Implementation dieser Smartphone-Anwendung (im Folgenden als „App“ bezeichnet) vorgestellt.

4.1 Anforderungen

Bei der Umsetzung des Frontends wird der in Abschnitt 2.1.1 vorgeschlagene Ansatz des Thin-Clients verfolgt. Abgeleitet aus dem Gesamtkonzept von SAM muss die App folgende funktionale Anforderungen erfüllen:

- Nutzerverwaltung:
 - Erstellen eines Accounts
 - Login/Logout zur Authentifizierung des Datenaustauschs
- Tracking:
 - Positionserfassung (GPS)
 - Ermitteln des Momentanverbrauchs (OBD)
- Auswertung:
 - Zusammenfassung einer Strecke
 - Ermitteln der optimalen Route

Ferner werden im Allgemeinen an eine App die folgenden nicht-funktionalen Anforderungen gestellt:

- Unterstützung mehrerer Displayauflösungen, Bildschirmgrößen und Seitenverhältnisse
- ergonomische und effiziente Bedienungsabläufe
- geringer Stromverbrauch
- Robustheit gegenüber schwacher Datenverbindung
- Reaktion auf Displayrotation

Die App wird zunächst als Prototyp entwickelt, mithilfe dessen die Funktionsfähigkeit des Gesamtkonzepts demonstriert werden soll. Bei der Entwicklung liegt der Fokus daher zunächst auf der Funktionalität. Die Erfüllung der nicht-funktionalen Anforderungen wird im Rahmen dieser Arbeit als optional angesehen.

4.2 Architektur der Windows Phone-Applikation

Aus den in Abschnitt 4.1 genannten Anforderungen werden die in Abbildung 16 dargestellten Views entwickelt und daraus eine logische Bedienführung abgeleitet. Das Design der Views ist durch XAML-Markup definiert, während die Funktionalität in einer gleichnamigen C#-Klasse implementiert wird. Die App besteht darüber hinaus aus den in Abschnitt 3.2.1 beschriebenen Grundkomponenten.

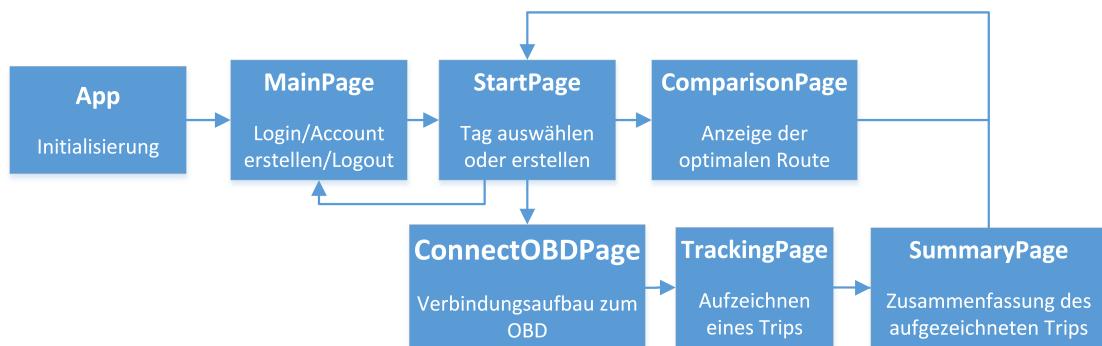


Abbildung 16: Übersicht der verschiedenen Views der App.

Diese Klassen werden in der Mutterklasse der Applikation (`App.cs`) instanziert und können über den `NavigationHandler`, welcher global in allen Klassen zur Verfügung steht, angezeigt werden. Er ist ebenso in der Lage, Daten zwischen verschiedenen Views zu transferieren und kann die Behandlung der Systemfunktion *Zurück*²⁰ beeinflussen. Funktionen, die im direkten Zusammenhang mit dem View stehen, werden in der zum View gehörenden Klasse implementiert. Für Funktionen, die von mehreren Views aus angesprochen werden sollen, muss dagegen eine eigene Klassenstruktur geschaffen werden. Die Implementierung der App umfasst folgende Bereiche:

²⁰Alle Windows Phones haben eine Hardwaretaste für die *Zurück*-Funktion.

- Kommunikation mit dem Server (siehe Abschnitt 4.3)
- Kommunikation mit der OBD-Schnittstelle (siehe Abschnitt 4.4)
- Nachbildung der für die Kommunikation mit dem Server benötigten Objekte
- Tracking-Prozess (siehe Abschnitt 4.5)

Damit die App genutzt werden kann, muss im App-Manifest die Autorisierung für die Datenverbindung sowie die Ortungsfunktion gefordert werden.

4.3 Kommunikation mit dem Server

Das Frontend ist über einen Webserver mit dem Backend verbunden. Dieser stellt die in Abschnitt 5.2 beschriebene Funktionalität bereit, welche von der App angesprochen werden muss. Dies umfasst folgende Funktionen:

- Nutzerauthentifizierung
- Übertragen von aufgezeichneten Messdaten
- Empfangen von Strecken zu einem Tag

Dazu werden auf dem Webserver Endpoints geschaffen, sodass dieser über das HTTP-Protokoll Daten als JSON-String oder URL-Parameter empfangen bzw. senden kann. Die für die Implementation der Serverkommunikation notwendigen Funktionen sind im .NET Framework im Namespace `Windows.Web.Http` zusammengefasst [36].

Da aus verschiedenen Views mit dem Server kommuniziert werden muss, wird im Namespace der App die Klasse `ServerCommunication` geschaffen, welche die oben genannte Funktionalität global bereitstellt.

4.3.1 Verarbeitung von Objekten in JSON-Notation

Der Austausch von Messdaten und die Zuordnung von Strecken zu einem Tag geschieht, indem Objekte im Format eines JSON-Strings mit dem Webserver ausgetauscht werden. Da Windows Phone keine native JSON-Unterstützung bietet, wird zum

Erstellen und Parsen von JSON-Strings das weitverbreitete Framework „JSON.NET“²¹ verwendet. Zum Serialisieren und Deserialisieren müssen auszutauschende Objekte in der Klassenstruktur der App als Eigenschaften einer C#-Klasse nachgebildet werden. Die JSON-Notation für das Objekt zur Kommunikation mit dem Server wird in Abschnitt 5.2.3 vorgestellt. Die entsprechende C#-Implementation sieht wie folgt aus.

```
1 public class Trip
2 {
3     public int userId { get; set; }
4     public String tag { get; set; }
5     public List<Point> points;
6 }
7
8 public class Point
9 {
10    public double lon { get; set; }
11    public double lat { get; set; }
12    public double time { get; set; }
13    public double distance { get; set; }
14    public double fuel { get; set; }
15 }
```

Quellcode 12: Nachbildung der JSON-Objekte als C#-Eigenschaft.

In der Instanz für die Serverkommunikation können die getrackten Daten mithilfe des folgenden Aufrufs als JSON-String an den entsprechenden Endpoint des Webservers übertragen werden.

```
1 using namespace JSON.NET;
2 using namespace Windows.Web.HTTP;
3
4 private HttpClient _client;
5 private string _url;
6
7 public async void transmitTrip(Trip trip)
8 {
9     String message = JsonConvert.SerializeObject(trip);
10    await _client.PostAsync(_url, new HttpStringContent(message));
11 }
```

Quellcode 13: Serialisierung der Trip-Daten als Beispiel für die Generierung eines Objektes in JSON-Notation.

²¹Siehe <http://www.newtonsoft.com/json>

Das folgende Beispiel zeigt, wie man ein in JSON-Notation vorliegendes Objekt in die in Quellcode 20 implementierte C#-Klasse deserialisieren kann.

```
1 //Herunterladen der Trips zu einem Tag
2 public async Task <List<Trip>> downloadTrips(String tag)
3 {
4     HttpResponseMessage response = await _client.PostAsync(_url, new
5         HttpStringContent(tag));
6     return await JsonConvert.DeserializeObject<List<Trip>>(response.
7         Content.ToString());
8 }
```

Quellcode 14: Deserialisierung der Strecken zu einem Tag als Beispiel für das Parsen eines Objektes in JSON-Notation.

4.3.2 Nutzerverwaltung

Um Daten mit dem Server austauschen zu können, muss sich ein Nutzer authentifizieren. Da die App zurzeit das einzige Frontend ist, muss sie die Möglichkeit bieten, ein Nutzerkonto zu erstellen. Dies ist beim ersten Start der App auf dem in Abbildung 17a dargestellten View möglich. Bei allen nachfolgenden Starts wird als MainPage der in Abbildung 17b gezeigte View geladen.

Der Server stellt, wie in Abschnitt 5.2.4 beschrieben, einen Endpunkt zum Erstellen eines neuen Nutzerkontos bereit. Dieser muss mit einem HTTP-Post angesprochen werden, welcher den gewünschten Nutzernamen sowie das Passwort beinhaltet. Als Antwort erhält das Frontend einen JSON-String mit einer Fehlermeldung, falls der Nutzernname bereits vergeben ist. Andernfalls wird die neu erstellte User-ID zurückgeliefert. Das empfangene Objekt wird analog zum in Abschnitt 4.3.1 vorgestellten Verfahren deserialisiert und anschließend ausgewertet. Die User-ID wird in der globalen Instanz für die Serverkommunikation gespeichert, sodass sie der darauffolgenden Kommunikation beigefügt werden kann.

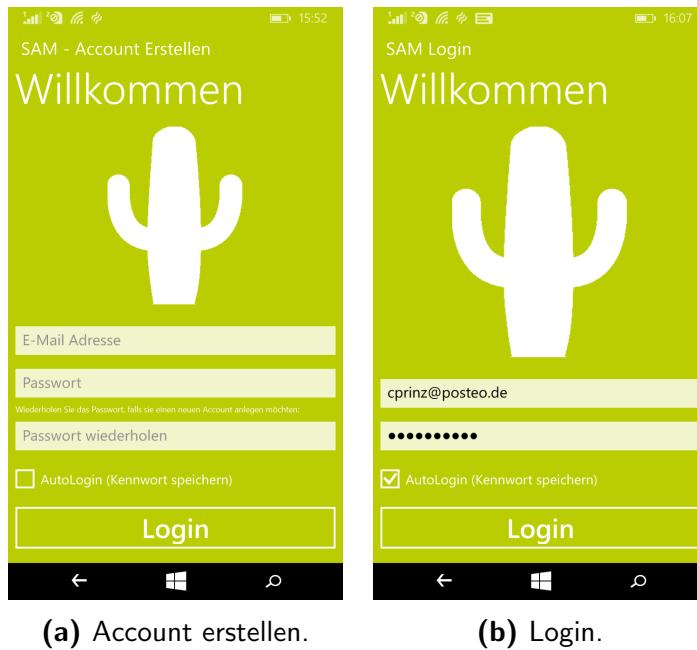


Abbildung 17: Willkommensbildschirm der Windows Phone-App (MainPage).

Konnte ein Konto erstellt werden, werden die User-ID und der Nutzernname im ApplicationData-Space²² persistent gespeichert, um später erkennen zu können, ob in der App bereits ein Nutzer registriert wurde. In Abhängigkeit davon wird entschieden, welche der in Abbildung 17 gezeigten MainPages geladen wird.

Gemäß den in Abschnitt 4.7.1 vorgestellten Ergonomierichtlinien soll die App mit einem Minimum an Nutzerinteraktionen einsatzbereit sein. Daher wird ein AutoLogin-Modus implementiert, der durch das Setzen der entsprechenden Checkbox auf der MainPage aktiviert wird. Der Login wird nun beim Start der App ohne Nutzereingabe vorgenommen. Hierfür wird das Passwort ebenfalls im ApplicationData-Space persistiert und die Login-Routine beim Start automatisch aufgerufen. Damit ein Nutzer über die App Daten an das Backend übertragen oder abrufen kann, muss er authentifiziert sein. Die Authentifizierung gegenüber dem Server wird über einen Cookie realisiert, der dem Frontend beim Login ausgeliefert wird. Der Cookie wird mit folgendem Code aus der Antwort des HTTP-Posts für den Login extrahiert und in der globalen Instanz für die Server-Kommunikation gespeichert.

²²API der Windows-Runtime zur Persistierung von Daten mittels eines eindeutigen Schlüssels.

```
1 HttpBaseProtocolFilter filter = new HttpBaseProtocolFilter();
2 HttpCookieCollection cookieCollection = filter.CookieManager.
    GetCookies(new Uri("http://rif.atria.uberspace.de/signup/"));
3 HttpCookie receivedCookie = cookieCollection[0];
4 _sessionCookie = new HttpCookie(receivedCookie.Name, "http://rif.
    atria.uberspace.de/", receivedCookie.Path);
```

Quellcode 15: Übernahme eines Cookies aus einem HTTP-Paket in die Kommunikationsinstanz.

Der Cookie muss bei jeder Anfrage enthalten sein und kann einem HTTP-Paket über den Windows.Web.Http-Namespace hinzugefügt werden. Die Kommunikation, die über die Anmeldung hinaus mit dem Server stattfindet, ist Teil der Abschnitte 4.5.3 und 4.6.

4.4 Bluetooth-Kommunikation

Für die Umsetzung des in Abschnitt 2.1 beschriebenen Konzeptes muss die Bluetooth-Schnittstelle des Smartphones verwendet werden. Das Konzept sieht vor, dass der Momentanverbrauch des Fahrzeugs über einen ELM327-Mikrocontroller, der über eine Bluetooth Schnittstelle verfügt, ausgelesen wird.

Dieser wurde in Abschnitt 3.1.2 charakterisiert und kann über den Bluetooth-Socket unter Verwendung des RFCOMM-Protokolls angesprochen werden. Damit die App die Bluetooth-API der Windows-Runtime für RFCOMM nutzen kann, muss dem App-Manifest zunächst folgender Code beigefügt werden:

```
1 <Capabilities>
2   <Capability Name="internetClientServer"/>
3   <DeviceCapability Name="location" />
4     <m2:DeviceCapability Name="bluetooth.rfcomm">
5       <m2:Device Id="any">
6         <m2:Function Type="serviceId:00001101-0000-1000-8000-00805
          F9B34FB" />
7       </m2:Device>
8     </m2:DeviceCapability>
9   </Capabilities>
```

Quellcode 16: Eintragung in das App-Manifest zur Autorisierung der Verwendung eines RFCOMM-Bluetooth-Sockets.

Die Bluetooth-Kommunikation wird in der Klasse OBDCommunication implementiert, welche global instanziert und damit von allen Views aus erreichbar ist.

4.4.1 Aufbau der Bluetooth-Verbindung und Initialisierung der Kommunikation

Bevor die Kommunikation mit dem Adapter beginnen kann, muss eine Bluetooth-Kopplung durchgeführt werden. Bei Windows Phone muss ein Bluetooth-Gerät einmalig über die Systemsteuerung gekoppelt werden, bevor eine App die Verbindung automatisiert aufbauen kann. In der App kann die Windows-Runtime-API durch folgenden Code alle Geräte, welche das RFCOMM-Protokoll nutzen, auflisten:

```
1 private static readonly Guid RfcommChatServiceUuid = Guid.Parse("00001101-0000-1000-8000-00805F9B34FB");
2 private DeviceInformationCollection _rfcommDevices;
3
4 //RFCOMM kompatible Peers in Auswahlbox laden
5 private void findPeers()
6 {
7     _rfcommDevices = await DeviceInformation.FindAllAsync(
8         RfcommDeviceService.GetDeviceSelector(RfcommServiceId.FromUuid(
9             RfcommChatServiceUuid)));
10    _selectBluetoothPeerBox.ItemsSource = _rfcommDevices.Names;
}
```

Quellcode 17: Anzeige von bereits gekoppelten Bluetooth-Geräten, welche das RFCOMM-Protokoll unterstützen.

Wenn der Nutzer auf der StartPage einen Tag ausgewählt hat und die Messdatenerfassung starten möchte, werden auf diese Weise im in Abbildung 18 dargestellten View (ConnectOBDPage) kompatible Bluetooth-Peers aufgelistet. Ist die Verbindung erfolgreich aufgebaut, wird die Device-ID des OBD-Adapters im ApplicationData-Space persistent gespeichert. Gemäß den in Abschnitt 4.7.1 vorgestellten Ergonomierichtlinien soll die App mit einem Minimum an Nutzerinteraktion einsatzbereit sein. Dies kann erfüllt werden, indem die App automatisch einen Verbindungsaufbau mit dem zuletzt gekoppelten Adapter initiiert, welcher über die gespeicherte Device-ID zugreifbar sein sollte.



Abbildung 18: View zum Verbinden des OBD-Adapters (ConnectOBDPPage).

Die Bluetooth-Kommunikation wird durch die API auf eine StreamSocket-Instanz abstrahiert, auf die Lese- und Schreibzugriffe über DataReader und DataWriter organisiert werden. Diese repräsentieren im .NET-Framework Buffer, auf die byteweise zugegriffen werden kann. Der Socket kann über folgenden Code aufgebaut werden und bleibt geöffnet, solange der StreamSocket instanziert ist. Service, Socket, Writer und Reader sind Member der OBDCommunication-Klasse, welche wie folgt instanziert werden, sobald der Button „Verbindung aufbauen“ getippt wird.

```

1 //Verbindung zum OBD Adapter
2 private void connectPeer(String deviceID)
3 {
4     _obdService = await RfcommDeviceService.FromIdAsync(deviceId);
5     _obdSocket = new StreamSocket();
6     await _chatSocket.ConnectAsync(_chatService.ConnectionHostName,
7         _chatService.ConnectionServiceName);
8     _writer = new DataWriter(_obdSocket.OutputStream);
9     _reader = new DataReader(_obdSocket.InputStream);
}

```

Quellcode 18: Aufbau des Kommunikationssockets mit dem OBD-Adapter.

4.4.2 Zugriff auf den Kommunikationssocket

Nachdem die Verbindung aufgebaut ist, können mit dem Writer und Reader Nachrichten über den Bluetooth-Socket versendet und empfangen werden. Beim Lesen von Nachrichten aus dem Eingangsbuffer muss die Länge der Nachricht bekannt sein, da der Aufruf blockiert, falls zu viele Zeichen aus dem Buffer gelesen werden. Da die Länge einer OBD-Antwort nicht zuverlässig definiert ist, scannt die App eine empfangene Nachricht byteweise, bis das in Abschnitt 3.1.2 beschriebene Trennzeichen „>“ erkannt wird. Folgendes Beispiel macht das Versenden einer Nachricht über die Windows-Runtime-API deutlich.

```
1 _writer.WriteString("testmessage");
2 await _writer.StoreAsync();
3 //auf Antwort warten
4 await _reader.LoadAsync(1);
5 while (_reader.UnconsumedBufferLength > 0)
6 {
7     String byteString = _reader.ReadString(1);
8     response += byteString;
9     if (byteString == ">") return response;
10 }
```

Quellcode 19: Senden und Empfangen einer Nachricht mithilfe des Kommunikationssockets.

Durch die asynchrone Implementierung erhält die aufrufende Instanz die Antwort, sobald diese verfügbar ist. Auf diese Weise wird automatisch die maximal mögliche Kommunikationsfrequenz erreicht. Im folgenden Abschnitt 4.5 wird beschrieben, wie die Bluetooth-Kommunikation mit dem OBD-Adapter realisiert ist.

4.5 Messdatenerfassung im Tracking-Prozess

Im Trackingprozess sollen die Momentanverbrauchsinformationen sowie die aktuelle Position erfasst und abhängig von der Zeit gespeichert werden, damit Streckenabschnitte auf dem Server hinsichtlich ihrer Effizienz charakterisiert werden können. Die aktuelle Position kann über das im Smartphone eingebaute GPS-Modul ermittelt werden. Der Momentanverbrauch kann über den Bluetooth-OBD-Adapter, welcher an die Diagno-

seschnittstelle im Fahrzeug angeschlossen ist, ermittelt werden. Im folgenden Abschnitt wird beschrieben, wie das Erfassen der Messdaten implementiert ist.

4.5.1 Auslesen und Interpretieren von OBD-Parametern

Um den Momentanverbrauch vom Auto auslesen zu können, müssen, wie in Abschnitt 4.5.2 beschrieben, mehrere Parameter vom OBD-Adapter abgefragt werden. Jedem OBD-Parameter ist ein sogenannter PID²³ zugeordnet, mit dessen Hilfe der entsprechende Wert abgefragt werden kann.

Zum Auslesen eines Parameters muss die App den OBD-PID als ASCII-Code gefolgt von einem Linefeed in den DataWriter des Bluetooth-Sockets schreiben. Das OBD-Kommando ist immer 4 Zeichen lang, während der zu interpretierende Teil der Antwort aus 1, 2 oder 4 Zeichen besteht. Die Interpretation der Werte ist abhängig vom OBD-PID und in der ELM327-Spezifikation dokumentiert.

Neben den Nutzdaten besteht die Antwort des Adapters aus einem Echo des OBD-PIDs und anderem Overhead. Das Echo kann mithilfe des Steuerungskommandos (AT-Commands) „ATE0“ deaktiviert werden, sodass der Overhead auf ein Minimum reduziert wird. Dies erhöht ferner die Kommunikationsfrequenz und damit die Genauigkeit der getrackten Parameter.

Die Interpretation eines gesendeten Kommandos wird zustandsabhängig implementiert, da die Antworten in der Reihenfolge der Anfragen erfolgen. Damit die konkreten Messwerte aus einer Antwort extrahiert werden, wird sie von der App in folgenden Teilschritten bearbeitet:

1. **cleanupResponse** (für alle Kommandos gleich): entfernt Freizeichen und anderen Kommunikationsoverhead aus der Antwort
2. **getResponseBody** (abhängig von Länge der Antwort): extrahiert die Nutzdaten aus der Antwort und stellt sie als hexadezimalen Wert (unverändert) bereit
3. **interpretResponse** (abhängig vom OBD-PID): konvertiert den Wert in das gewünschte Zielformat

²³Zugriffscodes

In Anlehnung daran wird die in Abbildung 19 gezeigte Klassenstruktur implementiert, welche aus folgenden Komponenten besteht:

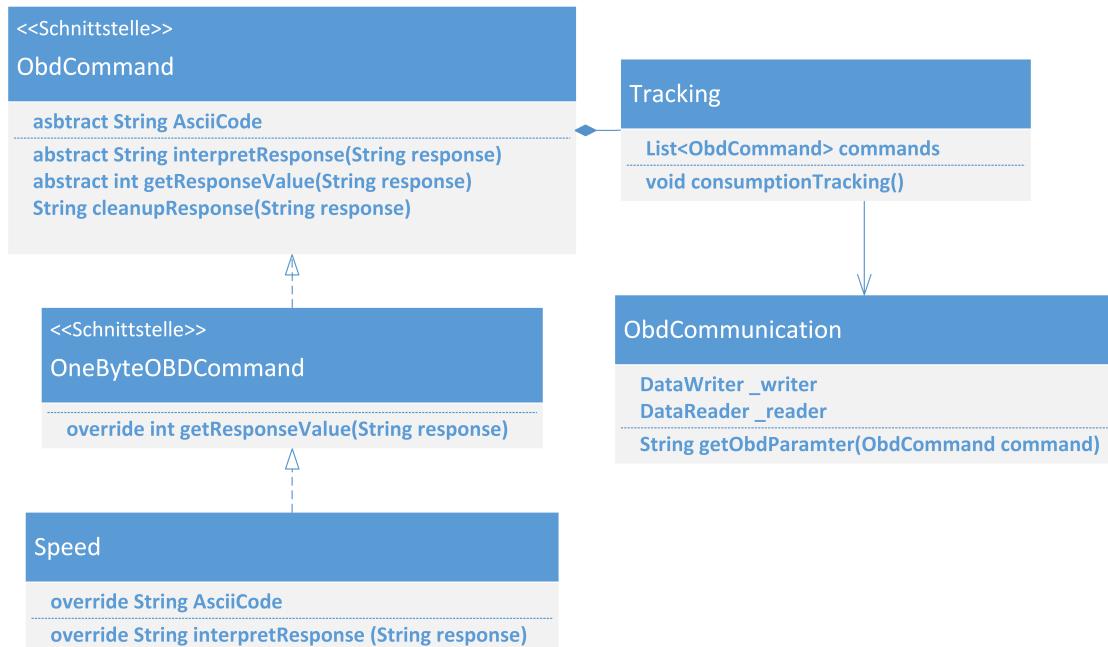


Abbildung 19: Klassendiagramm für die OBD-Kommunikation (unvollständig).

OBDCmd: Schnittstelle für OBD-PID-spezifische Implementierungen. Enthält die `cleanupResponse`-Routine zur Entfernung des Kommunikationsoverheads.

OneByteOBDCmd: Implementiert die Extrahierung der Nutzdaten bei einer 1 Byte langen Antwort. Es existieren parallel hierzu die Klassen `TwoByteOBDCmd` und `FourByteOBDCmd` für 2 und 4 Byte lange Antworten.

Speed: Implementiert den ASCII-Code zur Generierung des OBD-Kommandos und verfügt über die Routine zur Interpretation der zuvor extrahierten Nutzdaten. Alle in SAM genutzten OBD-PIDs existieren analog zu dieser Klasse und implementieren die jeweils passenden Schnittstellen.

Tracking: Instanziert die für das Berechnen des Verbrauches notwendigen OBD-Kommandos und hat Zugriff auf die Instanz zur OBD-Kommunikation. Die Messdatenerfassung während des Trackings findet in einem separaten Thread statt und erfolgt mit der maximal möglichen Frequenz.

OBDCommunication: Reguliert die Kommunikation mit dem verbundenen OBD-Adapter. Während des Trackings kann der Methode `getParameters` eine Liste an OBDCommand-Instanzen übergeben werden, welche durch folgenden Programmcode abgearbeitet werden.

```

1 public async Task<List<String>> getParameters(List<ObdCommand>
2     parameters)
3 {
4     List<String> results = new List<String>();
5     foreach (ObdCommand parameter in parameters)
6         results.Add(await getParameter(parameter));
7     return results;
8 }
9
10 public async Task<String> getParameter(ObdCommand parameter)
11 {
12     return parameter.interpretResponse(await getValue(parameter.
13         AsciiCode));
14 }
15
16 private async Task<String> getValue(String asciiCode)
17 {
18     String command = asciiCode + "\r\n";
19     _writer.WriteString(command);
20     await _writer.StoreAsync();
21     return await receiveMessage();
22 }
```

Quellcode 20: Senden, Empfangen und Interpretieren eines OBD-Kommandos.

Die Methode liefert das interpretierte Ergebnis zurück, sodass dieses in der Tracking-Klasse weiterverwendet werden kann.

4.5.2 Bestimmung des Momentanverbrauchs

Nachdem nun die Möglichkeit besteht, beliebige OBD-Parameter auszulesen, muss als Teil des Trackings auf diese Weise der Momentanverbrauch ermittelt werden. Die Spezifikation für die OBD-PIDs enthält den Parameter „Engine Fuel Rate“ (015E), welcher laut Dokumentation Auskunft über den Momentanverbrauch gibt. Dieser wird jedoch nicht von den zur Verfügung stehenden Fahrzeugen unterstützt, sodass er im Rahmen von SAM nicht genutzt werden kann.

Grundsätzlich ist es möglich, den Momentanverbrauch aus anderen OBD-PIDs näherungsweise zu berechnen. Um die Kompatibilität der App mit verschiedenen Fahrzeugtypen zu fördern, sollten OBD-Parameter verwendet werden, die von möglichst vielen Fahrzeugen unterstützt werden.

Im Rahmen von SAM konzentriert sich das Ermitteln des Verbrauches jedoch zunächst auf das zur Verfügung stehende Testfahrzeug²⁴. In [37] wird beschrieben, wie der Momentanverbrauch mithilfe von OBD-Parametern bestimmt werden kann. Die verwendeten OBD-Parameter werden von einer Vielzahl an Fahrzeugen unterstützt. Demnach besteht für Dieselfahrzeuge ein linearer Zusammenhang zwischen dem Produkt aus dem Parameter „MAF air flow rate“ (0110) und „Calculated engine load value“ (0104) mit dem Momentanverbrauch. Durch eine lineare Regression des Produktes mit den Momentanverbrauchswerten kann die Formel für eine praxisnahe Kalkulation des Momentanverbrauchs genutzt werden. Es werden die Momentanverbrauchswerte, die der Bordcomputer des Fahrzeugs anzeigt, verwendet. Für die Generierung der Messwerte werden Testfahrten mit einer Mindestdauer von 2 Minuten durchgeführt, um die Genauigkeit zu erhöhen. Die Messungen werden in unterschiedlichen Lastbereichen durchgeführt. Die mit dem Testfahrzeug durchgeführte Kalibrierung ist in Abbildung 20 dargestellt.

Im Praxiseinsatz ließ sich unter Verwendung der auf das Fahrzeug kalibrierten Funktionsparameter eine Genauigkeit des Verbrauches von etwa 20 % Prozent erreichen. Ein genaueres Ergebnis wird verhindert, da sich mit der ermittelten Funktionsgleichung nur unter Last ein korrekter Momentanverbrauch errechnen lässt. Sowohl im Leerlauf als auch im Betrieb mit Schubabschaltung liefert die Regression falsche Werte.

Mit diesem Wissen lässt sich die Genauigkeit der Messung erhöhen. Im Leerlauf beträgt der Offset zwischen dem Momentanverbrauch und dem berechneten Wert $0.5 \frac{l}{h}$. Wenn die über die OBD-Schnittstelle ermittelte Drehzahl unter $950 \frac{U}{min}$ fällt, wird dieser Offset von der App korrigiert.

²⁴Seat Ibiza 6J mit 1.6 TDI Dieselmotor

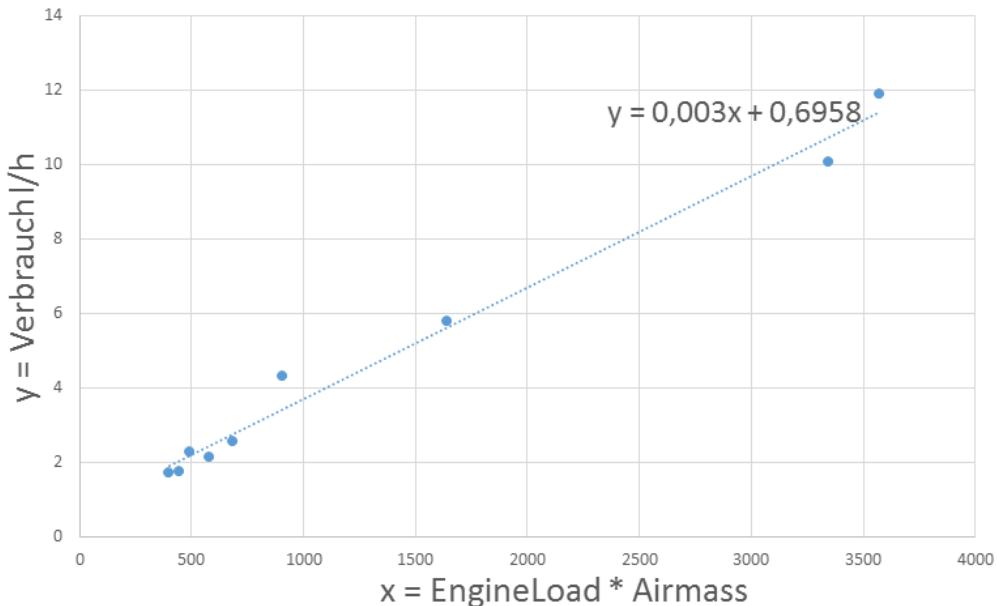


Abbildung 20: Regression zwischen Momentanverbrauch und OBD-Parametern zur Kalibrierung der Verbrauchsberechnung.

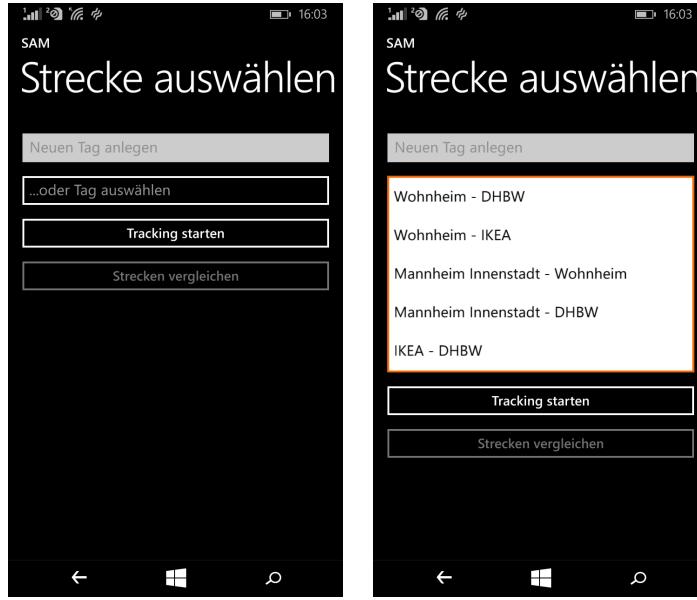
Wenn die Schubabschaltung des Motors aktiv ist, liegt der Momentanverbrauch bei $0 \frac{l}{h}$. In diesem Fall liegt der über das OBD ermittelte Wert, der „Calculated engine load value“, ebenfalls bei 0. Die App kann somit detektieren, wann die Schubabschaltung aktiv ist und den Momentanverbrauch entsprechend korrigieren.

Diese Verbesserungen erhöhen die Genauigkeit der Formel auf eine Abweichung von etwa 10 %. Die Kalibrierung sollte bei konstanter Fahrt und mit möglichst vielen Phasen, in denen die Schubabschaltung oder der Leerlauf aktiv ist, durchgeführt werden. Auch wenn der Verbrauch mithilfe der kalibrierten Formel nicht mit vollkommener Zuverlässigkeit ermittelt werden kann, reicht die Genauigkeit, um festzustellen, ob eine Strecke effizienter ist als eine andere, womit das Ziel dieser Studienarbeit erreicht werden kann.

4.5.3 Aufzeichnen einer Strecke

Alle Fahrten mit gleichem Start- und Zielpunkt kann der Nutzer unter einem Tag zusammenfassen. Zu jedem Tag kann er sich ferner, basierend auf den aufgezeichneten Daten aller Fahrten, eine Auswertung anzeigen lassen. Abbildung 21 zeigt das Haupt-

menü (StartPage) von SAM nach dem Login, in dem eine Strecke zu einem neuen oder bestehenden Tag aufgezeichnet sowie ein bereits befahrener Tag ausgewertet werden kann. Die Tags werden über einen vom Webserver bereitgestellten Endpoint abgerufen.



(a) Neuen Tag erstellen.

(b) Vorhandene Tags.

Abbildung 21: Hauptmenü der App zur Auswahl einer Strecke (MainPage).

Neben dem Momentanverbrauch wird die aktuelle GPS-Position gespeichert. Da die Geolocation-API zur Bestimmung der Position eine maximale Aktualisierungsrate von 1 Hz hat, während die OBD-Schnittstelle bei der Aktualisierung eines Parameters 15 Hz erreicht, dienen neue GPS-Positionen als Taktgeber für den Trackingprozess. Sobald eine GPS-Genauigkeit von 10 m erreicht ist, welche der maximalen Genauigkeit des Smartphones entspricht, kann der Nutzer den Trackingvorgang starten. Das OBD ist zu diesem Zeitpunkt bereits verbunden und sammelt in einem separaten Thread Momentanverbrauchsdaten.

Die Geolocation-API wird so konfiguriert, dass sie bei einer Positionsänderung von ebenfalls 10 m ein Event auslöst, welches den neuen GPS-Punkt zum aktuellen Trip hinzufügt. Die Zwischenwerte für den Momentanverbrauch werden in einen Buffer geschrieben und bei Erstellung eines neuen Punktes mit einem Mutex zugriffsgeschützt ausgelesen und anschließend gelöscht. Der Mittelwert der Zwischenwerte wird gebildet

und mithilfe der Zeitdaten auf den absoluten Spritverbrauch umgerechnet. Dieser wird dem GPS-Punkt zusätzlich zu einem Zeitstempel und dem Abstand zum letzten Punkt hinzugefügt. Die einzelnen Messpunkte werden in einer indizierten Liste gesammelt. Während des Trackingprozesses zeigt die App dem Nutzer die in Abbildung 22 gezeigten Informationen an (TrackingPage). Die Zusammenfassung der aktuellen Strecke wird bei jedem neuen GPS-Punkt oder spätestens nach einer Sekunde aktualisiert, während der Momentanverbrauch sowie die Geschwindigkeit immer aktualisiert werden, sobald neue Werte verfügbar sind. Ferner wird über die Geocoding-API die aktuelle Straße eingeblendet.



Abbildung 22: Anzeige von aktuellen Fahrtdaten sowie einer Zusammenfassung während des Trackings (TrackingPage).

Das Tracking kann jederzeit durch den Nutzer abgebrochen werden. Die Instanz, welche die Punkte sowie die Zusammenfassung der Strecken enthält, wird unter Verwendung des Frame-Navigators als Übergabeparameter an den Zusammenfassung-View übergeben. Dieser zeigt die gefahrene Strecke auf einer Karte an und fasst die Eckdaten der Strecke wie in Abbildung 23 zusammen (SummaryPage).



Abbildung 23: Zusammenfassung-View der App (SummaryPage).

Der Nutzer kann die Strecke mit dem zuvor gewählten Tag auf den Server übertragen und erhält eine Bestätigung, sobald die Daten erfolgreich übertragen wurden. Die Übertragung findet mithilfe der globalen `ServerCommunication`-Instanz in Form des in Abschnitt 5.2.1 beschriebenen JSON-Objektes statt.

4.6 Vergleichen von Strecken

Neben der Erfassung von Messdaten besteht eine Kernfunktion der App darin, dem Nutzer eine optimale Route zu einem gewählten Tag anzuzeigen (ComparisonPage). Wie in Abbildung 24 deutlich wird, liegt der Fokus dabei auf der effizientesten und der schnellsten Route.

Die verschiedenen Fahrten zu einem Tag können über einen Endpunkt abgerufen werden. Das JSON-Objekt ist analog zum Tracking aufgebaut, enthält allerdings nicht alle Punkte, sondern nur jene, welche auf einer Kreuzung liegen. Die gefahrenen Routen werden in die Karte eingezeichnet und anschließend werden die optimalen Routen farblich hervorgehoben. Zu diesen werden ferner die Distanz, der Durchschnitt des absoluten Verbrauchs sowie die Fahrzeit angezeigt.

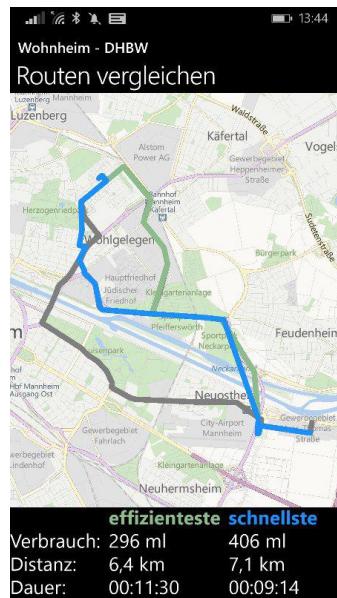


Abbildung 24: Vergleich von Strecken zu einem Tag (ComparisonPage).

4.7 Erfüllung nicht-funktionaler Anforderungen

Im folgenden Abschnitt wird beschrieben, welche Maßnahmen ergriffen werden, um die nicht-funktionalen Anforderungen zu erfüllen.

4.7.1 Bedienungskonzept und Nutzerergonomie

Die App soll dem Nutzer einen ergonomischen Zugriff auf die Funktionen von SAM bieten. Grundlegende Anforderungen an eine ergonomische Bedienung werden durch die Verwendung der Windows-Runtime-Designelemente erreicht. Diese stellen sicher, dass das „Look & Feel“ der App vergleichbar mit anderen Systemanwendungen sowie Windows Phone-Apps ist. Der Nutzer erkennt Bedienelemente wieder und kann basierend auf Erfahrungswerten erkennen, wie diese benutzt werden.

Da ein Smartphone in der Regel mit einer Hand bedient wird, ist es einer ergonomischen Bedienung förderlich, Bedienelemente im unteren Teil der Nutzeroberfläche zu platzieren. Außerdem werden Buttons, die der Nutzer mit hoher Wahrscheinlichkeit verwendet, größer dargestellt als andere.

Ein weiterer Grundsatz ist es, die Nutzerinteraktion auf ein Minimum zu reduzieren.

Erreicht wird dies durch das Speichern von wiederkehrenden Eingaben wie der Nutzer-E-Mail-Adresse sowie der Auswahl des OBD-Adapters. Ferner wird der „AutoLogin“ sowie der „AutoConnect“ zur automatisierten Anmeldung beziehungsweise dem automatischen Verbinden mit einem bekannten OBD-Adapter implementiert. Beim Start der Anwendung muss der Nutzer somit nur einen Tag auswählen und anschließend das Tracking oder den Vergleich starten.

Mit diesen Maßnahmen wird die Ergonomie der App verbessert, sodass die nicht-funktionale Anforderung eines effektiven Bedienkonzeptes erfüllt ist.

4.7.2 Anpassung auf verschiedene Displaygrößen

Microsoft spezifiziert Hardwarevoraussetzungen für Smartphones, auf denen Windows Phone als Betriebssystem eingesetzt werden soll. Bei der Entwicklung einer App muss beachtet werden, dass Windows Phones unterschiedliche Auflösungen im 16:9 oder 16:10 Seitenverhältnis haben. Das Windows Phone SDK skaliert Bedienelemente mit absoluten Größenangaben entsprechend der Displaygröße. Auf einem 4-Zoll-Display werden Bedienelemente daher kleiner dargestellt als auf einem 6 Zoll großen Display. Die Anzahl der darstellbaren Inhalte ist jedoch im Standardfall identisch. Unterschiedliche Pixeldichten haben ebenfalls keinen Einfluss auf die Größe der dargestellten Elemente. Da im Rahmen von SAM nicht mehr Elemente auf einem größerem Display angezeigt werden sollen, wurde das Standardverhalten der Bedienelemente nicht beeinflusst.

Durch die Nutzung des Grid-Layouts mit Pufferzonen zwischen zwei Elementen kann sichergestellt werden, dass in allen Seitenverhältnissen Bedienelemente korrekt dargestellt werden. Die App kann somit auf allen Windows Phones ohne Einschränkung ausgeführt werden, sodass die nicht-funktionale Anforderung, alle Displaygrößen und -auflösungen zu unterstützen, erfüllt ist.

4.7.3 Reduzierter Stromverbrauch

Visual Studio bietet Debug-Tools an, welche den Stromverbrauch sowie die CPU-Auslastung der App zur Laufzeit anzeigen können. Folgende Maßnahmen dienen dazu, die Ressourcennutzung der App möglichst effizient zu halten:

- GPS-Genauigkeit auf 10 Meter anstatt auf maximale Genauigkeit
- Straßename in Tracking-View wird nur alle 10 Punkte aktualisiert
- während der Trackings wird keine dynamische Karte angezeigt
- schwarze Hintergründe (vorteilhaft bei OLED-Displays)

Während des Trackingvorgangs liegt die CPU-Auslastung des Testgerätes bei etwa 10 %. Dies ist deutlich weniger als bei einer aktiven Navigationsanwendung, sodass die funktionale Anforderung eines geringen Stromverbrauchs erfüllt ist.

4.7.4 Robustheit gegenüber schwacher Datenverbindung

Da SAM in Kraftfahrzeugen eingesetzt wird, ist die App zur Datenübertragung an den Server auf die mobile Datenverbindung des Smartphones angewiesen. Auf diese ist jedoch nicht immer Verlass, sodass die App auch offline funktionieren muss. Erreicht werden kann das, indem die App eine lokale Kopie der Datenbank verwaltet, auf deren Basis Daten und Strecken ausgewertet werden können. Ferner muss die Möglichkeit bestehen, einen aufgezeichneten Trip lokal zu speichern und ihn zu einem späteren Zeitpunkt hochzuladen.

Durch ein einstündiges Tracking entstehen etwa 300 KByte Daten, die an den Server gesendet werden müssen. Ein Praxistest im Netz der Deutschen Telekom ergibt, dass meist auf das LTE- oder 3G-Netz zugegriffen werden kann, sodass ein Transfer der Daten in weniger als einer Sekunde möglich ist. Mit dem langsamsten Netzstandard EDGE dauert der Transfer etwa 14 s, was einer vertretbaren Wartezeit entspricht. Die App zeigt nach dem erfolgreichen Transfer eine Bestätigung an.

Beim Download von Strecken fallen weniger Daten an, da hier nicht alle GPS-Punkte, sondern nur die Kreuzungen übertragen werden. Auf die Implementation des oben

beschriebenen Offline-Modus wird daher verzichtet, da für den Testeinsatz der App keine Nachteile durch eine schwache Datenverbindung zu erwarten sind. Die nicht-funktionale Anforderung kann somit nicht erfüllt werden.

4.7.5 Reaktion auf Displayrotationen

In Fahrzeugen sind Smartphones meist im Querformat montiert. Folglich muss die Nutzeroberfläche der App für Hoch- und Querformat optimiert werden. Durch den Ausrichtungssensor des Smartphones rotiert der Bildschirm automatisch, sobald der Nutzer das Gerät dreht. Dies löst das `OrientationChanged`-Event aus, welches in der Klasse des entsprechenden Views behandelt werden kann.

Alle Nutzeroberflächen der SAM-App verwenden das Grid-Layout. Durch eine Neupositionierung der einzelnen Grid-Elemente als Reaktion auf das ausgelöste Event kann der Bildschirm in jeder Ausrichtung optimal ausgenutzt werden. Dies soll anhand des Anmeldebildschirms (StartPage) verdeutlicht werden. Der Anmeldebildschirm ist als 3x2 Grid organisiert. Die horizontalen Grenzen passen sich dynamisch an die Größe der drei Elemente an, während die vertikale Grenze in der Mitte des Views liegt. In Abbildung 25a wird deutlich, dass im Hochformat alle Elemente über beide Spalten gehen.



(a) Hochformat.

(b) Querformat.

Abbildung 25: Rotation am Beispiel des Willkommensbildschirms (StartPage).

Wird das Event für die Bildschirmrotation ausgelöst, werden die drei gruppierten Elemente neu positioniert, sodass der Bildschirm wie in Abbildung 25b vollständig genutzt

werden kann. Auf die Displayrotation wird in allen Views auf diese Weise reagiert, so dass der Bildschirm sowohl im Hochformat als auch im Querformat optimal ausgenutzt wird. Die nicht-funktionale Anforderung, auf Bildschirmrotationen zu reagieren, ist damit erfüllt.

4.8 Evaluation

Die Windows Phone-App ist als Prototyp in der Lage, Messdaten zu erfassen und diese nutzergespeist an den Server zu übertragen. Außerdem können mit ihr verschiedene Strecken eines Tags verglichen werden, was dem Ziel von SAM entspricht. Die in Abschnitt 4.1 definierten funktionalen Anforderungen sind damit vollständig erfüllt. Um das gesamte Nutzererlebnis zu verbessern, sind die ebenfalls in Abschnitt 4.1 beschriebenen nicht-funktionalen Anforderungen umgesetzt.

Die Funktionalität und Robustheit der App wird im Praxistest validiert. Dabei entspricht das Verhalten der geplanten Funktionalität und erweist sich ferner als zuverlässig. Lediglich beim Vergleich von verschiedenen Strecken kann es auf der Karte zu Darstellungsfehlern kommen, da die Kreuzungen direkt miteinander verbunden werden und somit der eingezeichnete Weg nicht immer auf Straßen verläuft.

Der Entwicklungsprozess der App kann dahingehend verbessert werden, dass die für die OBD-Kommunikation benötigten Komponenten mithilfe eines OBD-Simulators entwickelt werden. Aktuell kann die Kommunikation mit dem Fahrzeug nur im Fahrzeug getestet werden. Zurzeit steht ein derartiger Simulator nicht öffentlich zur Verfügung. An der Windows Phone-App können in Zukunft folgende konkrete Verbesserungen vorgenommen werden:

- Nutzung der Geofencing-API zur Reduzierung des Strombedarfs für die Ortung
- Lauffähigkeit im Hintergrund
- Konsequente Verwendung der Datenbindung
- Nutzeroberfläche für Tablets und PCs, sodass die App im gesamten Windows 8-Ökosystem verwendet werden kann

Um die Qualität der Verbrauchserfassung zu verbessern, sind in Zukunft folgende Schritte denkbar:

- automatische Kalibrierung der Formel für den Momentanverbrauch
- verschiedene Formeln zur Berechnung des Momentanverbrauchs in Abhängigkeit von verfügbaren OBD-Parametern
- Sammeln von Fahrzeugkalibrierungen in einer Datenbank

Sobald SAM über den Testbetrieb hinaus verwendet werden soll, wird es notwendig, die App auch für andere Plattformen zu entwickeln. Bei dem Betriebssystem iOS von Apple muss dabei beachtet werden, dass dies nur mit einem WLAN-OBD-Adapter funktionieren wird, da das verwendete Bluetooth-Protokoll RFCOMM nicht unterstützt wird. Für die Entwicklung einer Google Android-App kann auf Softwarebibliotheken für die OBD-Kommunikation zurückgegriffen werden.

In die App können ferner in Zukunft folgende Features integriert werden:

- Offline-Modus
- automatischer Start des Trackings, wenn Adapter über Bluetooth verbunden ist oder mithilfe eines NFC-Tag
- Aufzeichnen eines Trips parallel zur Anzeige der optimalen Route
- Export einer Route in andere Navigationsanwendungen

5 Umsetzung der Serveranwendung mit Node.js

Wie in Abschnitt 2.2 dargestellt, wird die Serveranwendung mittels Node.js umgesetzt und bei uberspace.de [38] gehostet. Im folgenden Abschnitt soll auf die Implementierung des Webservers und die Algorithmen zur Datenerfassung und -auslieferung eingegangen werden.

5.1 Anforderungen

Die Serveranwendung setzt definierte Anforderungen um. In Anlehnung an das verlangte Thin-Client-Modell implementiert die Backendanwendung wesentliche Teile der Anwendungslogik. Die folgende Übersicht soll die Anforderungen an die Serveranwendung darstellen:

- Webserver:
 - Routing eingehender Nachrichten
 - REST-Endpunkte für das Frontend bereitstellen
 - Benutzeroauthentifikation
- Algorithmen zur Datenerfassung und -auslieferung:
 - Komprimierung der GPS-Punkte in Kreuzungspunkte zur Speicherung
 - Zusammenführen vorhandener Daten mit einer neuen Strecke
 - Vorberechnung der Daten zur Auslieferung an den Client

5.2 Webserver

Der Webserver stellt REST-Endpunkte zur Kommunikation mit dem Frontend bereit. Das bedeutet, dass der Webserver die Anfragen des Clients annimmt, verwaltet, verarbeitet und beantwortet. Aufgaben des Webservers sind dabei das Verteilen (Routing) eingehender Nachrichten innerhalb der Anwendung, die Bereitstellung von Schnittstellen für das Frontend und die Datenbank sowie die Benutzeroauthentifikation und das Logging.

5.2.1 Express als Framework

Für zahlreiche Problemstellungen bei der Entwicklung einer Anwendung mit Node.js stehen fertige Implementierungen (Module) via NPM zur Verfügung. Zu den Standardmodulen gehört dabei Express, welches eine Middleware für einen Webserver bereitstellt. Dieses erleichtert den Umgang mit Anfragen und vereinfacht die Entwicklung. Node.js bietet mit dem `http`-Modul die Möglichkeit, einen Webserver zu implementieren, und stellt mit der Klasse `server` die Funktion `listen(port[, hostname][, backlog][, callback])` bereit. Express ermöglicht es, mit dem Befehl `app.listen(port, callback)` die Anwendung an einen Port zu binden und nach Verbindungen abhören zu lassen. Eine eingehende Anfrage wird von dem Framework entgegengenommen und gibt dem Node.js-`http`-Modul eine Funktion zurück, welche es dem Server ermöglicht, die Anfrage anzunehmen. Der eingehenden Verbindung wird nach der Verarbeitung durch den Webserver geantwortet [39].

Eine weitere Option erlaubt es, Key-Value-Paare mittels Express zu verwalten. Die gesetzten Werte können mit Getter-Methoden abgefragt und durch den Webserver genutzt werden. Des Weiteren stellt die Middleware einen Funktions-Stack zur Verfügung, welcher bei eingehenden HTTP-Nachrichten durchlaufen wird. Somit werden die Anfragen durch die im Stack definierten Funktionen bearbeitet, bevor sie die eigentliche Anwendung erreichen.

5.2.2 Routing eingehender Nachrichten

Der Clientanwendung werden durch den Server Endpunkte angeboten. Eine reduzierte Weboberfläche ermöglicht zudem, vereinfacht auf Daten des Servers zugreifen zu können. Damit die Funktionalität und Logik, die hinter einem Endpunkt steht, verteilt und wieder verwendet werden kann, wird ein Router eingesetzt. Dieser hat die Aufgabe, eingehende Anfragen je nach angefragtem Endpunkt an eine Funktion weiterzuleiten. Diese Funktionalität beinhaltet die Datei `server.js`. Das Routing selbst übernimmt die Middleware Express, welche dazu über das Konstrukt `app.METHOD(PATH, HANDLER)`

das Routing bereitstellt. Mit `app` wird auf die Instanz der Middleware zugegriffen. Die `METHOD` legt fest, welche HTTP-Methoden (POST oder GET) bearbeitet werden. Innerhalb der Klammern wird mittels `PATH` die Endpunktadresse definiert, welche die Funktionalität des Händlers (`HANDLER`) erreichbar macht. Demnach stellt der Händler den Zugriff auf die gewünschte Funktion bereit, die über den Endpunkt erreichbar sein soll.

Für das Aufzeichnen (Loggen) der auf dem Webserver eingehenden Anfragen wird der Aufruf `app.all('*', HANDLER);` verwendet. Mittels `all` wird Express angewiesen, auf eingehende Anfragen mit der im Händler verbundenen Funktion zu reagieren.

5.2.3 Schnittstelle zur App

Die entwickelte Windows Phone-App erreicht die Serveranwendung unter der URL²⁵ `http://rif.atria.uberspace.de/`. Unter dieser Adresse werden der App Endpunkte zur Verfügung gestellt, um mit dem Server interagieren zu können. Der Webserver erwartet POST-Anfragen für die Endpunkte `/v01/mobile_json.json`, `/usertags/`, `/usertrack/`, `/signin/` und `/signup/`. Zusätzlich wird die GET-Methode `/signout/` bereitgestellt. Auf die drei zuletzt Genannten wird im folgenden Abschnitt 5.2.4 näher eingegangen.

Der Endpunkt `/v01/mobile_json.json` stellt eine Funktion bereit, die JSON-Nachrichten von der mobilen Anwendung annimmt. Erreicht den Server eine Nachricht unter dieser Adresse, leitet er die Anfrage an die Funktion `read_json` weiter. Dabei wird erwartet, dass im Body der gesendeten HTTP-Nachricht Daten im JSON-Format enthalten sind. Die aufgerufene Funktion kann die in der Nachricht enthaltenen Daten auslesen und zur weiteren Verarbeitung nutzen. Dazu werden diese in einem Objekt gespeichert. Sind alle Zeichen gelesen, wird der Inhalt zusätzlich in einer Datei auf dem Server persistiert. Die Datei kann über den Browser heruntergeladen und ausgewertet werden. Diese Option erfüllt den Anspruch, Fehler in der Nachrichtenübermittlung besser erkennen zu können. Des Weiteren ist es möglich, Testdaten zu nutzen, welche

²⁵Uniform Resource Locator, engl. für „einheitlicher Ressourcenanzeiger“

bei einem frühen Entwicklungsstand noch nicht in der Datenbank persistiert werden können.

Die von der App zu übermittelnden Daten müssen ein definiertes Format aufweisen. Dieses wird in Abbildung 26 dargestellt.

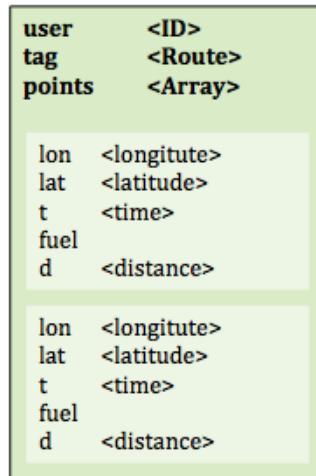


Abbildung 26: Schema der eingehenden JSON-Daten.

Das Attribut `user` hält die Nutzer-ID aus der SQL-Datenbank (vgl. Abschnitt 5.2.4), um die gesendeten Daten einem Nutzer zuordnen zu können. Des Weiteren wird ein `tag` gesendet. Diese Zeichenkette ermöglicht die Zuordnung einer gefahrenen Strecke zu einer Route, wie es in Abschnitt 2.3.3 beschrieben ist. Anschließend folgt ein Array `points`, welches die aufgezeichneten GPS-Punkte mit den dazugehörigen Parametern enthält. Folgende Eigenschaften werden durch ein Array-Element repräsentiert: Längengrad (`longitude lon`), Breitengrad (`latitude lat`), vergangene Zeit zum vorherigen Punkt (`time t`), Kraftstoffverbrauch zwischen dem aktuellen und vorherigen Punkt (`fuel`) und der Abstand zum vorherigen Punkt (`distance d`).

Die Daten werden beim Eintreffen mithilfe der in Abschnitt 5.3.1 und 5.3.2 beschriebenen Algorithmen verarbeitet. Anschließend können sie in der Graphdatenbankpersistiert werden.

Um Daten vom Server abzufragen, werden dem Client die Endpunkte `/usertags/` und `/usertrack/` angeboten. Ersterer liefert dem Frontend eine Liste der vom Nut-

zer angelegten Tags. Dazu ruft der Webserver die Funktion `getTagsById` auf, welche eine Cypher-Abfrage gegen die Neo4J-Datenbank stellt, wie in Quellcode 21 dargestellt ist. Es werden alle Knoten (`n`) mit einer Verbindung (`r:DRIVE`) zu einem weiteren Knoten gesucht. Die Verbindung muss ein Attribut besitzen, welches mit der Nutzer-ID übereinstimmt (Zeile 2). Für das zurückgelieferte Ergebnis werden gleiche Werte mittels `DISTINCT` zusammengefasst.

```
1 "MATCH (n)-[r:DRIVE]->() " +
2 "WHERE r.userID = {userID} " +
3 "RETURN DISTINCT r.tag AS tag";
```

Quellcode 21: Abfrage aller Tags eines Nutzers.

Der Endpunkt `/usertrack/` erhält eine Anfrage mit einem Tag als Parameter. Dieser muss vom Nutzer der App vorher ausgewählt werden. Mit dem empfangenen Attribut werden die zu dem Tag gehörenden Strecken aus der Datenbank ausgelesen. Dabei ist zu beachten, dass in der erzeugten Liste identische Strecken enthalten sein können. Um diese zusammenzufassen, werden deckungsgleiche Wege mit dem in Abschnitt 5.3.3 beschriebenen Algorithmus verrechnet. Die berechneten Strecken sind so miteinander vergleichbar. Die aufbereiten Daten werden an das Frontend gesendet, welches die Informationen visualisiert.

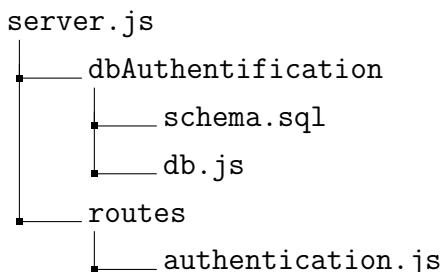
5.2.4 Benutzeroauthentifikation

Die Umsetzung wurde von [40] ausgehend entwickelt, welche im Wesentlichen auf das Node.js-Modul Passport [41] zurückgreift.

Dem Nutzer der Anwendung soll nur der Zugriff auf die von ihm erzeugten Daten gewährt werden und die Berechtigung mit einem Passwort abgesichert sein. Des Weiteren muss eine Zuordnung zwischen dem Nutzer und dessen gefahrenen Strecken erfolgen können. Daher ist serverseitig eine Benutzeroauthentifikation implementiert, welche die dargestellten Anforderungen für ein solches System erfüllt.

Der aktuelle Stand der Software ermöglicht dem Nutzer die Registrierung, die An-

meldung und das Abmelden vom System. Ist ein Nutzer nicht angemeldet, muss er sich registrieren. Dazu steht die Funktion `signUpPost` zur Verfügung. Nach einer erfolgreichen Registrierung kann sich der Nutzer mit der Methode `signOut` abmelden. Um sich erneut anmelden zu können, wird die Funktion `signInPost` bereitgestellt. Der Zugriff auf die Funktionen wird durch Endpunkte serverseitig und die entsprechende clientseitige Implementierung umgesetzt. In der App wird zusätzlich eine Plausibilitätsprüfung der Anmelddaten durchgeführt. Des Weiteren nutzt der Server das Modul `express-session`, um mittels Cookies²⁶ einen angemeldeten Nutzer bei weiteren Anfragen zu erkennen (Aufbau einer Session). Die Middleware Passport kann aus einem einmaligen Cookie auf die Nutzerdaten schließen, sodass nicht mit jeder durch den Client gesendeten Nachricht die Nutzerinformationen mitgesendet werden müssen. Dieses Vorgehen erhöht die Datensicherheit bei der Kommunikation. Eine umfangreichere Benutzerverwaltung wird zunächst nicht implementiert, da die Aufmerksamkeit auf der Bestätigung der zu Beginn gestellten These liegt.



Der dargestellte Dateibaumausschnitt soll im Folgenden zur Orientierung dienen und die für die Nutzerverwaltung notwendigen Dateien angeben.

Die Datei `schema.sql` definiert die MySQL-Datenbank. Zur Einrichtung der Nutzerdatenbank wird diese Datei mit dem MySQL-Befehl `source` importiert. Dabei wird zunächst überprüft, ob bereits eine Datenbank mit dem Namen existiert. Ist das der Fall, wird diese gelöscht. Anschließend werden die Datenbank und die Tabelle mit den Spalten `username`, `password`, `userID`, `trackID` und `hitID` erzeugt. Für die Nut-

²⁶Ist ein Textinformation, die im Browser durch einen Webserver gesetzt wird, der diese zu einem späteren Zeitpunkt auswerten kann.

zerverwaltung werden die Nutzer-ID, der Nutzernname und das Passwort benötigt. Die Track-ID wird zur eindeutigen Markierung von gefahrenen Strecken eines Nutzers eingesetzt. Die Hit-ID ermöglicht eine Zuordnung von gleichen Strecken. Diese beiden IDs werden vom Webserver gelesen und inkrementiert. Somit erfolgt eine direkte Zuordnung der IDs zu einem Nutzer, jedoch nicht zu den Fahrten. Es wird eine Datentrennung erreicht, die den Datenschutz erhöht. Da die Zuordnung auf unterschiedlicher Hardware erfolgt, kann im Fall eines Datenlecks keine Zuordnung zwischen Nutzern und deren Routen erfolgen. Die eigentliche Information geht für Dritte verloren.

In weiteren Schritten sollten die IDs in eigene SQL-Tabellen ausgelagert werden, um die Funktion einer relationalen Datenbank besser auszunutzen. Das aktuelle Datenbankmodell für die Nutzerverwaltung umfasst eine Tabelle (Abbildung 27).

The screenshot shows the MySQL Workbench interface with a table named 'tblUsers'. The table has the following columns:

	userID INT(11)	username VARCHAR(1...	password VARCHAR(100)	trackID INT(11)	hitID INT(11)
Indexes	▶				

Abbildung 27: Datenbanktabelle für die Nutzerverwaltung.

Mit db.js kann das Model erstellt werden, welches den Zugriff auf die Datenbankinhalte als Objekte erlaubt. Dazu wird zunächst das Modul knex initialisiert, welches die Parameter zum Verbindungsaufbau mit der MySQL-Datenbank erhält. Des Weiteren wird das darauf aufbauende Modul Bookshelfjs eingebunden, um Abfragen an die Datenbank erstellen zu können.

Die Datei server.js bindet alle weiteren für die Implementierung der Benutzeroauthentifizierung benötigten Module ein, wie sie in der Auflistung 6 dargestellt sind.

Die Module werden anschließend mittels app.use() der Express-Middleware übergeben. Dadurch kann Express eingehende Nachrichten an die Module weiterreichen, ein separater Aufruf durch den Webserver ist somit obsolet.

Tabelle 6: Importierte Module für die Benutzeroauthentifikation und deren Funktion.

Modul	Beschreibung
bodyParser	Middleware zum parsen eines HTTP-Body
cookieParser	parsen von Cookies in HTTP Anfragen
express-session	erstellen einer Session-Middleware
bcrypt	native JS-Implementierung einer bcrypt-Verschlüsselung
passport	Middleware für Nutzerauthentication

Des Weiteren wird in der server.js-Datei das Modul Passport initialisiert, welches verschiedene Anmeldemöglichkeiten anbietet. Unter anderem kann eine sogenannte „Locale Strategie“ (LocalStrategy) genutzt werden, welche eine Benutzeranmeldung mit selbst verwalteten Logindaten implementiert. Hierfür werden Nutzernname und Passwort des Nutzers bei der Anmeldung mittels einer Datenbank verifiziert. Mit der durch das Modul Bookshelfjs bereitgestellten Schnittstelle wird der gesuchte Nutzer aus der MySQL-Datenbank geladen. Wurde eine Übereinstimmung gefunden, wird mit dem Modul bcrypt das Passwort gehasht²⁷ und mit dem gehaschten Passwort des Nutzers aus der Datenbank verglichen. Bei einer Übereinstimmung ist der Nutzer korrekt authentifiziert. Die Strategie wird Passport mittels `passport.use()` zugewiesen.

Der Zugriff auf die Authentifikation wird durch die Endpunkte `/signup/`, `/signin/` und `/signout/` ermöglicht. Die Händler der Endpunkte leiten die Anfrage an die jeweilige Funktion in `routes/authentication.js` weiter.

In der Datei `authentication.js` werden die Funktionen `signInPost` (POST-Methode), `signUpPost` (POST-Methode) und `signOut` (GET-Methode) bereitgestellt. Letztere meldet den Nutzer vom System ab und löscht die Session des Nutzers, der damit nicht mehr authentifiziert ist.

Zum Login wird die Funktion `signInPost` aufgerufen. Sie erhält vom Client die Parameter für den Nutzernamen und das Passwort. Das Modul Passport verarbeitet eine

²⁷Eine Hashfunktion bildet eine beliebig lange Zeichenkette auf eine Zeichenfolge mit fester Länge ab, dabei handelt es sich um eine sogenannte „Einwegfunktion“. Ist eine Zeichenfolge durch diese Funktion abgebildet, lässt sich der ursprüngliche Wert praktisch nicht mehr errechnen.

entsprechende Anfrage mit der Methode `passport.authenticate()`. Nachdem der Inhalt auf Fehler überprüft wurde, kann mit `logIn()` eine Session für den Nutzer angelegt werden. Damit sich ein Nutzer anmelden kann, muss dieser sich zuvor registriert haben. Dafür wird die Funktion `signUpPost` verwendet. Diese erstellt eine Datenbankabfrage, um ein Nutzer-Modell mit entsprechenden Logindaten bereitzustellen. Ist das möglich, wird die Nachricht zurückgegeben, dass der Nutzer bereits registriert ist. Andernfalls wird das gesendete Passwort mit `bcrypt` gehasht und mit dem Nutzernamen in der Datenbankpersistiert. Anschließend ist der neu registrierte Nutzer direkt angemeldet.

5.3 Algorithmen zur Datenerfassung und -auslieferung

In diesem Abschnitt werden die drei wichtigsten Algorithmen zur Datenerfassung, Speicherung und Auslieferung erläutert. Im Kontext der gesamten Applikation stellen die Algorithmen das Bindeglied zwischen Erfassung und Speicherung der Daten dar. In einem ersten Schritt werden die erfassten Informationen auf Kreuzungspunkte komprimiert. In einem zweiten Schritt werden die komprimierten Daten in der Graphdatenbank gespeichert. Hierbei müssen die neuen Daten mit bereits persistierten Daten zusammengeführt werden. Zur Auslieferung der Daten an den Client müssen diese aufbereitet werden. Dabei werden bestimmte Vorberechnungen durchgeführt, damit der Client die erhaltenen Informationen visualisieren kann.

5.3.1 Komprimierung der GPS-Punkte zu Kreuzungspunkten zur Speicherung

In diesem Algorithmus findet die Komprimierung der Daten statt, indem die Kreuzungspunkte aus den erfassten GPS-Punkten extrahiert werden. Anschließend wird ein Punkt nach dem anderen gewählt und dem folgenden OSM-Kreuzungspunkt zugeordnet. Ferner werden die Parameterwerte der einzelnen GPS-Punkte vom letzten Kreuzungspunkt bis zum Neuen aufsummiert und diesem zugeordnet.

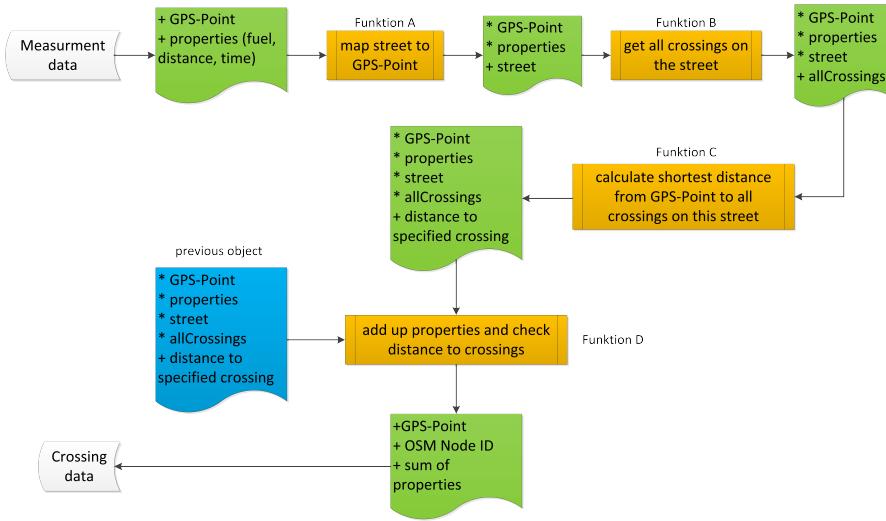


Abbildung 28: Ablaufdiagramm des Algorithmus zur Umrechnung von GPS-Punkten zu Kreuzungspunkten zur Speicherung.

Die folgende Erklärung des Algorithmus bezieht sich auf Abbildung 28, die den Ablauf schematisch darstellt. Die abgerundeten Rechtecke illustrieren die Input- und Outputdaten. Die einzelnen Schritte des Algorithmus sind mit Rechtecken dargestellt und symbolisieren eine Funktion.

Aufbereitung der Daten Im ersten Schritt (Funktion A) werden die vom Client gesendeten GPS-Punkte in einzelne Objekte überführt. Diese enthalten die Parameter Verbrauch, Zeitdifferenz, Distanz, User-ID, Track-ID und Tag sowie die GPS-Koordinate. Das erzeugte Objekt dient als Grundlage und wird im Verlauf des Algorithmus um zusätzliche Attribute erweitert.

Dazu wird zunächst die zum GPS-Punkt gehörende Straße mittels Gisgraphy, auf welches in Abschnitt 3.7.2 näher eingegangen wird, ermittelt. Der Dienst liefert die ID der Straße zurück, welche durch die OSM-Datenbank vergeben ist. Die Straßen-ID identifiziert eine Straße eindeutig und wird zu dem Datenobjekt hinzugefügt (Funktion B).

Wrapper zur Overpass-Anfrage Die gespeicherte Straßen-ID wird zur Detektierung aller Kreuzungen auf dieser Straße genutzt (Funktion B). Dafür wird eine Anfrage an Overpass gestellt, welches sämtliche auf einer Straße liegenden Kreuzungen

zurückliefert. Diese Anfrage wird durch eine Wrapper-Funktion realisiert, welche das Verifizieren und Cachen von Overpass-Anfragen implementiert. Im Folgenden wird diese Adapterfunktion `getCrossingsByGisgraphyStreetIdAdvanced` anhand der Abbildung 29 erläutert.

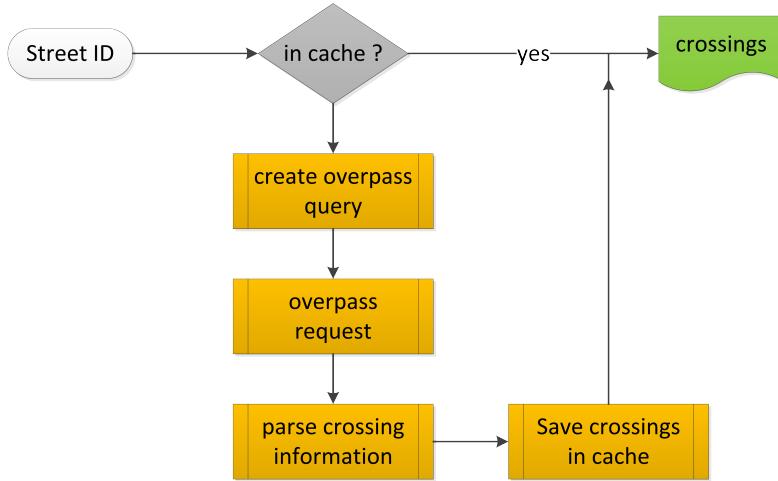


Abbildung 29: Ablaufdiagramm des Algorithmus in dem Adapter zur Overpass-Anfrage.

Im ersten Schritt implementiert diese Funktion das Puffern von Overpass-Anfragen. Dies ist sinnvoll, da für jede GPS-Position die Kreuzung der gerade befahrenen Straße ermittelt wird. Aufgrund dieser Tatsache wiederholt sich eine spezielle Overpass-Anfrage, solange sich das Auto auf einer Straße befindet. Durch die Pufferung wird der Overpass-Webservice entlastet und die Antwortgeschwindigkeit verbessert. Infolgedessen findet vor jeder Overpass-Anfrage eine Abfrage des lokalen Caches statt. Dieser Cache speichert die Antwort einer Overpass-Anfrage für jeweils eine Stunde. Das garantiert, dass der Arbeitsspeicher nicht überlastet wird.

Wenn die Antwort auf die Overpass-Anfrage sich noch nicht in dem lokalen Cache befindet, wird aus der Straßen-ID eine Overpass-Abfrage erzeugt. Diese Anfrage wird an den Webservice gesendet. Der Aufbau der Overpass-Abfrage ist in Abschnitt 3.7.3 dargestellt. Die Antwort auf die Overpass-Anfrage wird in einem zweiten Schritt verifiziert. Hierbei wird der Spezialfall in den OSM-Daten betrachtet, bei dem eine Straße in zwei Teilstücke aufgeteilt ist. Diese werden in OSM mit zwei unterschiedlichen IDs gespei-

chert und somit von Overpass auch als zwei unterschiedliche Straßen zurückgegeben. Dieser Fehler wird durch die Kontrolle der Straßennamen gelöst, indem Straßen mit gleicher Bezeichnung als eine Straße übernommen werden. Nun werden nur noch die Kreuzungen in ein Array übernommen, an denen sich zwei oder mehr Straßen treffen. Dieses Array von Kreuzungen wird sowohl im Cache gespeichert als auch von der Funktion zurückgegeben. Daraufhin wird die Liste von Kreuzungen auch dem Datenobjekt des GPS-Punktes hinzugefügt.

Berechnung der Distanz von GPS-Punkt und nächstgelegener Kreuzung Im Anschluss wird bestimmt, ob ein GPS-Punkt nahe einer Kreuzung liegt und daher als eine solche definiert werden kann (Funktion C). Dafür ist die Methode `nextCrossroad` implementiert. Ihr werden als Parameter eine Liste von Kreuzungen und ein GPS-Punkt übergeben. Dies entspricht einem Teil der Attribute eines Objektes, wie es in Abbildung 28 dargestellt ist. Die Methode sucht die Kreuzung mit der geringsten Distanz zum vorgegebenen GPS-Punkt und gibt die Node-ID der Kreuzung, ihre geografische Position sowie die dazu berechnete Entfernung in Metern zurück. Wird dieses Vorgehen für alle Punkte durchgeführt, lässt sich aus den resultierenden Distanzen ein Plot erstellen (siehe Abbildung 30). Der generierte Plot wird anschließend analysiert. Der Algorithmus vergleicht die Distanz zu einer Kreuzung von aufeinanderfolgenden Punkten. Es wird davon ausgegangen, dass man sich auf der aktuellen Straße einer Kreuzung nähert. Daher muss sich die Entfernung mit jedem GPS-Punkt zur Kreuzung verringern. Wird die berechnete Distanz größer, wurde eine Kreuzung passiert. Das bedeutet, dass der vorherige Punkt als Kreuzungspunkt identifiziert ist und entsprechend markiert wird. Der beschriebene Ablauf führt zu dem im Diagramm in Abbildung 30 dargestellten Ergebnis.

Die lila Kreuze stellen die Entfernung der erfassten GPS-Punkte zur nächsten Kreuzung dar. Mit den türkisen Kreuzen wird die Markierung einer erkannten Kreuzung dargestellt. Wie aus der Abbildung erkenntlich, wird jeweils der richtige GPS-Punkt gefunden.

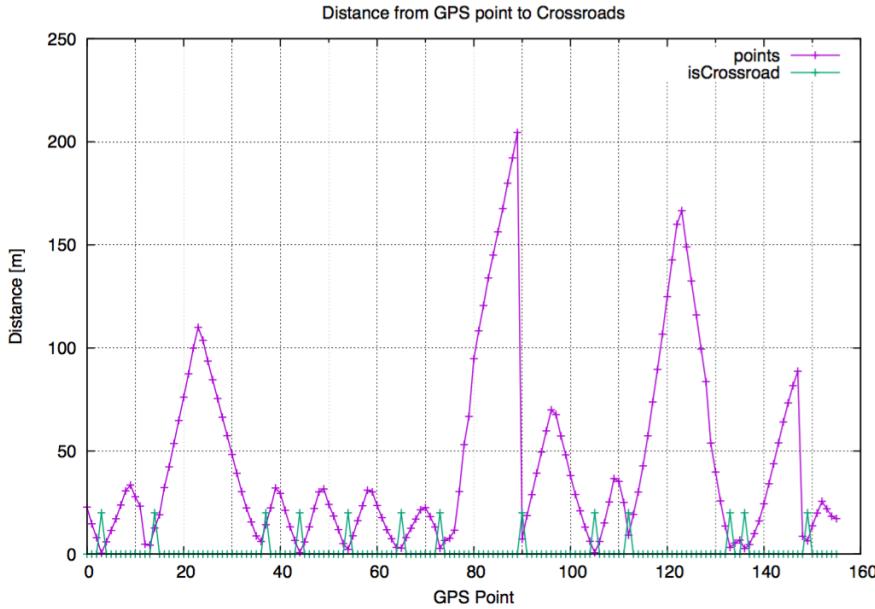


Abbildung 30: Minimale Distanzen zwischen aufgezeichneten GPS-Punkten und Kreuzungen.

Zuordnung der GPS-Punkte zu einer Kreuzung In einem letzten Schritt müssen die Eigenschaften wie der Verbrauch, die Zeitdifferenz und die Distanz zwischen zwei identifizierten Kreuzungen aufsummiert werden. Als Kreuzungspunkt wird der geografische GPS-Punkt aus der Overpass-Antwort verwendet. Das bedeutet, dass in der Datenbank der GPS-Punkt einer Kreuzung gespeichert wird, wie er in der OSM-Datenbank vergeben ist. Dadurch werden nur eindeutige Punkte persistiert. Die Differenz zwischen der berechneten Kreuzung aus den aufgezeichneten Daten des Nutzers und dem vorgegebenen Kreuzungspunkt der OSM-Datenbank kann hierbei vernachlässigt werden. Bevor die Liste von Kreuzungen mit den benutzerspezifischen Parametern User-ID, Track-ID und Tag in einem neuen Objekt persistiert wird, findet eine Validierung der erkannten Kreuzungen statt. In den Standphasen des Autos (zum Beispiel an einer Ampel) können durch Ungenauigkeiten des GPS-Sensors Kreuzungen doppelt aufgezeichnet werden. Dieser Fehler wird bei der Validierung der OSM-Node-IDs der Kreuzungen erkannt. Außerdem werden nahegelegene Kreuzungen zusammengeführt, da diese sich nicht einzeln auf das Fahrverhalten auswirken. Einen solchen Fall kann man in der Abbildung 30 zwischen den GPS-Punkten 130 und 140 erkennen. Dies sind in

der Realität meist große Kreuzungen, die OSM aus mehreren Kreuzungspunkten zusammensetzt. Das Objekt mit den benutzerspezifischen Parametern und der Liste von Kreuzungen steht dem Algorithmus zum Zusammenführen vorhandener Daten mit der neuen Strecke zur Verfügung.

5.3.2 Zusammenführen vorhandener Daten mit einer neuen Strecke

Der in Abschnitt 5.3.1 dargelegte Algorithmus liefert eine Liste von Kreuzungen, die den gefahrenen Weg beschreiben. Diese Objekte sind aufgebaut aus einer GPS-Koordinate, der OSM-Node-ID und den aufsummierten Parametern (Zeit, Verbrauch sowie Distanz) von der vorherigen bis zur aktuellen Kreuzung. Der folgende Algorithmus soll die entstandenen Daten als Graph in der Graphdatenbank persistieren. Wichtig ist hierbei, dass bereits gesetzte Knoten (Kreuzungen) nicht mehrfach angelegt werden, da die Datenbank ein Abbild der Karte aufbaut. Mehrfach gespeicherte Knoten könnten auftreten, wenn mehrere Nutzer dieselbe Strecke fahren oder ein Nutzer eine Strecke mehrfach befährt. Aus diesem Grund muss der Algorithmus gleiche Knoten erkennen und Verbindungen zwischen diesen setzen können. Die Diskussionen zur Implementierung des Algorithmus haben zu dem Ablaufdiagramm geführt, welches in Abbildung 31 dargestellt ist.

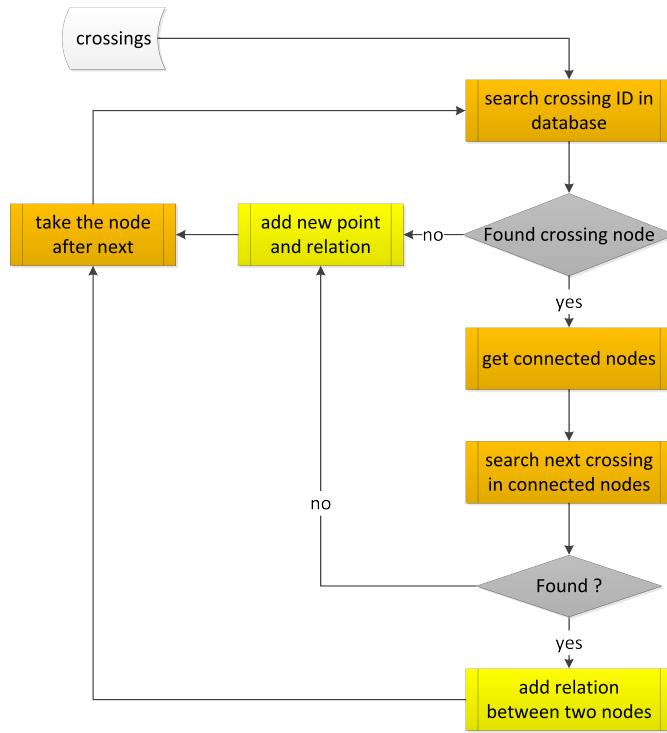


Abbildung 31: Ablaufdiagramm des Algorithmus zum Zusammenführen vorhandener Daten mit einer neuen Strecke.

Das Verfahren unterscheidet vier Szenarien, wie eine Relation zwischen zwei Kreuzungspunkten generiert wird. Diese werden in den Abbildungen 32 visualisiert. Orange eingefärbte Kreise stellen Kreuzungen dar, die schon in der Datenbank persistiert sind. Eine gestrichelte Umrandung deutet die Daten des neuen Datensatzes an. Folglich stellen weiß eingefärbte Kreise mit einer gestrichelten Umrandung Knoten dar, welche nicht in der Datenbank abgelegt sind. Ein orange gefärbter Kreis mit gestrichelter Umrandung gibt an, dass für den neuen Datensatz ein entsprechender Knoten in der Datenbank gefunden werden konnte. Schwarze Pfeile bilden existierende Kanten ab, gestrichelte die Erzeugung einer neuen Verbindung mit den Daten aus dem neuen Datensatz.

Das Ziel des Algorithmus ist es, die Relation der in 5.3.1 bearbeiteten Daten zwischen zwei aufeinander folgenden Kreuzungen zu speichern. Je nach Datenstand in der Datenbank muss dafür eine, zwei oder keine neue Kreuzung angelegt werden. Zu unterscheiden sind die Szenarien für eine Relation mit und ohne gegebenen Startknoten.

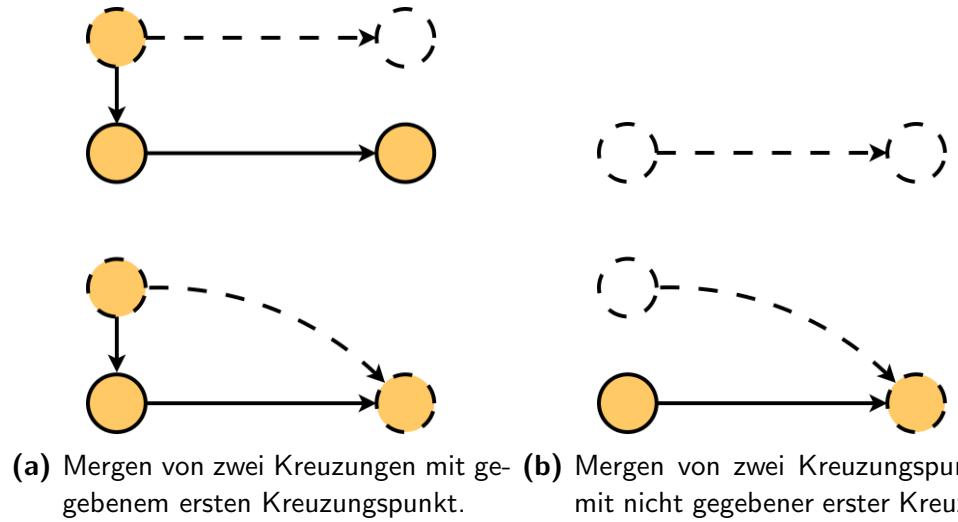


Abbildung 32: Vier Szenarien, welche der Merge-Algorithmus behandelt.

Der Algorithmus iteriert über die im vorangegangen Schritt erzeugte Liste von Kreuzungen. Dabei sucht er in der Datenbank nach einer existierenden Kreuzung für ein Element der Liste. Wird ein existierender Knoten erkannt, wird dem Schema in Abbildung 32a gefolgt. Es ist dann zu unterscheiden, ob die Kante der neuen Daten zu einer in der Datenbank noch nicht vorhandenen Kreuzung führt (Abbildung 32a oben) oder zu einer vorhandenen (Abbildung 32a unten). Im ersten Fall wird ein neuer Punkt angelegt und die Relation gesetzt. Existiert auch der zweite Knoten (Abbildung 32a unten), kann eine Verbindung zwischen den beiden Knoten direkt gesetzt werden (gestrichelte Linie). Existierende Verbindungen werden dabei nicht berücksichtigt. Ziel ist es, möglichst alle Informationen zu erhalten. Eine Zusammenfassung der Daten soll immer zu einem späteren Zeitpunkt geschehen, z.B. zum Zeitpunkt der Auswertung. Relevant sind nur existierende Kreuzungspunkte.

Ist der Startknoten einer Verbindung noch nicht in der Datenbank gespeichert, muss dieser angelegt werden. Wenn der Endknoten der Verbindung in der Datenbank vorhanden ist, wird der neue Startknoten mit diesem Punkt verbunden (Abbildung 32b unten). Andernfalls muss zusätzlich der folgende Kreuzungsknoten angelegt werden (Abbildung 32b oben).

Um eine Relation zu erzeugen, ist die Methode `addRelation` implementiert, deren

Ablauf im folgenden Pseudocode 22 dargestellt wird. Der Methode werden die OSM-Node-IDs für die Straßenkreuzungen sowie die zu der Kante gehörenden Eigenschaften übergeben. Diese werden in einem Objekt gespeichert und der Graphdatenbank-Anfrage als Parameter mit übergeben. Nun wird eine Graphdatenbank-Anfrage erzeugt, welche die Verbindung zwischen den beiden Kreuzungen erzeugt und Eigenschaften setzt. Im vierten Schritt wird die Datenbankanfrage ausgeführt. Dafür sendet das verwendete Framework Seraph.js die Cypher-Anfrage und die Parameter zusammen an den REST-Endpoint der Neo4J-Datenbank. Dort wird die Anfrage zusammengesetzt und ausgeführt. Abschließend wird das Ergebnis von dem Framework Seraph.js geparsert und der aufrufenden Funktion zurückgeliefert.

```
1 take startOSMID, endOSMID and properties
2 set parameter (startOSMID, endOSMID fuel, time, distance, userID,
     trackID, tag)
3 create database instruction
4 make db request
5 return result
```

Quellcode 22: Pseudocode für die Methode addRelation.

Aus der Ausführung geht hervor, dass Relationen immer gerichtete Kanten sind. Als Konsequenz werden entgegengesetzte identische Strecken als verschiedene Wege erkannt. Das ist sinnvoll, da die Streckencharakteristiken unter Umständen nicht zu vergleichen sind. Als Beispiel sei angeführt, dass ein Nutzer auf einem Berg wohnt und zur Arbeit ins Tal muss. Fährt der Nutzer den Rückweg, wird mehr Kraftstoff verbraucht als bei der Fahrt ins Tal.

5.3.3 Vorberechnung der Daten zur Auslieferung an den Client

Die Auswertung der in der Graphdatenbank gespeicherten Daten erfolgt in zwei Phasen. Die erste Phase beinhaltet die Berechnung der Durchschnittswerte gleicher Strecken und bereitet die Daten für die Auslieferung an den Client vor. Dies ist nötig, da jede

gefahrenen Strecke separat gespeichert wird, um einen Datenverlust zu vermeiden. Die zweite Phase erfolgt im Frontend. Dieses Phase vergleicht die berechneten Absolutwerte der Strecken und kann dann das Ergebnis entsprechend den Vorgaben für Zeit und Effizienz darstellen. Im Folgenden wird auf die Implementierung der ersten Phase eingegangen.

Bevor die Daten ausgewertet werden können, müssen sie in entsprechender Form in der Datenbank gespeichert werden. Dabei wird das Konzept angewendet, welches in Abschnitt 2.3.3 erläutert ist. Wie gefahrene Strecken in der Datenbank persistiert werden, skizziert das Schema 33 in reduzierter Form. Jeder Strecke ist eine eindeutige Track-ID zugewiesen. Zur Identifizierung identischer Strecken wird eine Hit-ID eingeführt. Das Attribut Hit-ID wird einer Strecke zugeteilt und kennzeichnet Strecken, deren Kreuzungspunkte durchgehend identisch sind. Die Kennzeichnung wird eingeführt, damit zu einem späteren Zeitpunkt identische Strecken zusammengefasst und die Parameter der Kanten arithmetisch gemittelt werden können.

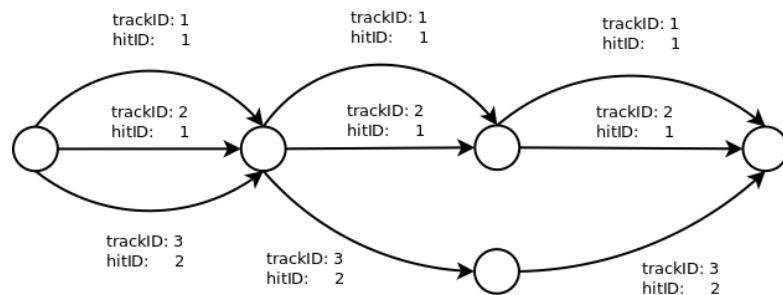


Abbildung 33: Schema zur Bestimmung der HitIDs.

Die Implementation erzeugt ein Objekt, das die Eigenschaften `isHit`, `trackIDs []` und `hitIDs []` besitzt. Bei dem Attribut `isHit` handelt es sich um einen Booleanwert. Dieser Wert bestimmt, ob am Ende des Merge-Prozesses eine neue Hit-ID vergeben wird oder eine bereits vorhandene dem neuen Track zugeordnet werden kann. Die beiden anderen Attribute sind Arrays, können aber als Listen verstanden werden.

Wird während des Mergeprozesses ein neuer GPS-Punkt erkannt, muss ein neuer Knoten in der Datenbank erzeugen werden. In diesem Fall wird der Booleanwert `false`

gesetzt. Existieren die Kreuzungspunkte, sind jedoch noch nicht miteinander verbunden, wird der Variable ebenfalls `false` zugewiesen. Nur wenn eingehende Daten mit einem in der Datenbank persistierten Graph übereinstimmen, bleibt der Ausgangswert `true` des Attributes erhalten. Die Tabelle 7 fasst die genannten Varianten zusammen.

Tabelle 7: Mögliche Kombinationen der Zusammenführung von Daten in der Datenbank.

Modul	Wert von <code>isHit</code>
aktueller Knoten existiert, folgender Knoten existiert nicht	<code>false</code>
aktueller Knoten existiert, folgender Knoten existiert, Knoten sind aber nicht miteinander verbunden	<code>false</code>
aktueller Knoten existiert nicht, folgender Knoten existiert	<code>false</code>
aktueller und folgender Knoten existieren und sind miteinander verbunden	<code>true</code>

Die Aktualisierung des Hit-Objektes wird mit der Funktion `hitObj.updateHits(isHit, newTrackIDs, newHitIDs)` umgesetzt, welche den Booleanwert für das Objekt bestimmt sowie eine aktualisierte Liste der Track-IDs und Hit-IDs übergibt. Die beiden Listen enthalten alle möglichen IDs, die für eine Übereinstimmung relevant sind. Kommt der Mergealgorithmus an eine Verzweigung, werden die Elemente eliminiert, die nicht mit dem neuen Abschnitt identisch sind. Die Auswertung des Hit-Objektes kann erst nach dem Abschluss der Zusammenführung erfolgen. Als Konsequenz müssen alle Verbindungen zwischengespeichert werden, um am Ende dieser Phase die Hit-ID für jede Verbindung zu setzen. Hierfür wird das Array `relationIDs[]` eingeführt und nach jedem Merge-Schritt mit der neu erzeugten Verbindung ergänzt. Nach Abschluss des Mergeprozesses wird der Wert von `isHit` überprüft. Ist dieser `true`, steht in der Liste der Hit-IDs nur noch eine Hit-ID, welche rückwirkend für alle neuen Verbindungen gesetzt werden kann. Ist dies nicht der Fall, wird eine neue Hit-ID durch deren Inkrementierung erzeugt.

Die Auslieferung der Informationen zu Strecken, die unter einem bestimmten Tag zusammengefasst sind (Route), beginnt mit der Anfrage des Clients nach allen Strecken

einer Route. Dieser sendet in der Anfrage an den Endpunkt /usertrack/ einen Tag mit, zu dem er Daten anfordert. Im ersten Schritt werden dazu über die Funktion getHitsByUserIdAndTag alle zu einem Nutzer und dem empfangenen Tag gehörenden Hit-IDs aus der Datenbank abgefragt und in einem Array gespeichert. Die erzeugte Liste von Hit-IDs wird an die nächste Funktion weitergereicht und dazu verwendet, eine Liste der Strecken zu generieren. Der Sourcecode 23 verdeutlicht die Implementation.

```

1  var evaluationTracks = [];
2  async.eachSeries(hitIDs, function(hitID, eachSeriesCallback) {
3    db_con.getTracksWithUserIdAndTagAndHitID(userID, userTag,
4      hitID).then(
5      function (result) {
6        var resultTrack = evaluation.addUpTrackValues(result);
7        evaluationTracks.push(resultTrack);
8        eachSeriesCallback();
9      }
10    )

```

Quellcode 23: Generierung der Antwortliste aus Strecken für das Frontend.

Für die oben beschriebene Funktionalität wird die Funktion getTracksWithUserIdAndTagAndHitID für jedes Element der Hit-ID-Liste angewendet. Mit db_con wird die Anfrage an die Graphdatenbank gestellt, welche mit dem Objekt result eine Liste mit den vollständigen Tracks zurückliefert. Vollständiger Track bedeutet in diesem Fall, dass ein Element mit allen Knotenpunkten (GPS-Koordinaten der Kreuzungen) und den dazugehörigen Kanten (gefahrenen Strecke) mit den jeweiligen Parametern gefüllt ist. Bei der Anfrage ist darauf zu achten, dass die gelieferten Ergebnisse nach deren Zeitstempel (time) sortiert sind, wie es die Cypher-Abfrage 24 formuliert. Das ist notwendig, da die Rückgabewerte standardmäßig nach den Indizes der Datenbank sortiert sind.

```

1 MATCH (s)-[r:DRIVE]->(e)
2 WHERE r.userID = {userID} AND r.tag = {tag} AND r.hitID = {hitID}
3 RETURN s,r,e
4 ORDER BY r.time

```

Quellcode 24: Sortierung des Results einer Cypher-Anfrage nach dem gesetzten Zeitstempel.

Das `result`-Objekt enthält eine Liste mit identischen Strecken. Diese kann der Funktion `addUpTrackValues` übergeben werden. Enthält das Objekt mehr als einen Eintrag (N Einträge), wird innerhalb der Funktion das arithmetische Mittel über die Eigenschaften der Kanten gebildet. Die zu berechnenden Parameter (P) sind Distanz, Zeit und Kraftstoffverbrauch.

$$\frac{\sum_{n=0}^{N-1} (P_n)}{N}$$

Nachdem die Durchschnittsbildung für jede Relation erfolgt ist, werden abschließend über den gesamten Graph die Absolutwerte für die drei Parameter durch Summierung ermittelt. Nach der Berechnung wird das Objekt in dem Array `evaluationTracks` gespeichert und für die restlichen Hit-IDs wiederholt. Nachdem jedes Element der Hit-ID-Liste bearbeitet wurde, enthält das Array `evaluationTracks` eine Liste der gefahrenen Strecken zu einer Route. Die Strecken enthalten alle relevanten Parameter, wobei identische Strecken kombiniert wurden. Das generierte Array wird abschließend an den Client gesendet.

Die Bewertung der Daten erfolgt aktuell nicht im Backend. Dieses sendet die berechneten Werte, gibt jedoch keinen Hinweis auf eine Bewertung der Strecken mit. Mit der Umsetzung einer Website und der Unterstützung verschiedener Smartphone-Plattformen sollte dieses Vorgehen überdacht werden. Wenn das Backend die Aufgabe übernahme, würde redundanter Sourcecode in den Clients entfallen können. Aktuell wurde die Entscheidung getroffen, dass der Prototyp durch die gegebene Implementation vereinfacht wird.

```
1  [{ "totalFuel":176.653,
2    "totalDistance":2489.740,
3    "totalTime":297.270,
4    "points": [{"lon":8.4903842,"lat":49.5069739,"fuel":6.63,
5      "distance":39.07,"time":1432159993.790688},...]
6  },
7  { "totalFuel":167.5413452777778,
8    "totalDistance":2259.190,
9    "totalTime":319.0020594596863,
10   "points": [{"lon":8.4925482,"lat":49.5069758,"fuel":22.463,
11     "distance":116.510,"time":1432160987.0729434},...]
12 }]
```

Quellcode 25: Zusammengefasste Routen im JSON-Format für das Frontend.

Wird die Ausgabe für das Frontend betrachtet, fällt auf, dass die ermittelten Durchschnittswerte an den Client gesendet werden. Jedoch würde es dem Client ausreichen, die Absolutwerte direkt zu vergleichen, um die Route zu bewerten. Für die aktuelle Visualisierung werden nur die GPS-Koordinaten der Kreuzungen benötigt, die Attribute der Teilstrecken werden in der Visualisierung noch nicht verwendet. Dennoch werden diese Daten übermittelt, da die Option erhalten bleiben soll, die einzelnen Abschnitte bewerten und farblich codiert darstellen zu können. Das erlaubt eine differenziertere Betrachtung der Routen.

5.4 Evaluation

Während der Umsetzung wurde erkannt, dass für die Serveranwendung die Node.js-Plattform nicht für alle Anforderungen geeignet ist. Insbesondere für die entworfenen Algorithmen zur Datenauswertung ist festzuhalten, dass eine Umstellung auf eine typisierte, sequentielle Programmiersprache, wie zum Beispiel Java, eine Alternative darstellen kann. Dies könnte den Programmfluss und die vorgestellten Algorithmen vereinfachen. Es ist jedoch auch zu evaluieren, ob die Entwicklung neuer Algorithmen sinnvoll ist, wenn diese es ermöglichen, den asynchronen Charakter von Node.js auszunutzen.

Analog gilt für die Verwendung der Datenbanktechnologie Neo4J, dass die Vorteile einer Graphdatenbank derzeit noch nicht ausgenutzt werden. Diese eignet sich für den

Einsatz mit GPS-Punkten, stellt jedoch eine gewisse Redundanz zu den vorliegenden OSM-Daten dar. Die Erarbeitung eines Konzeptes zur Integration der gewonnenen Daten in eine lokal gehostete PostgreSQL-Datenbank wäre erstrebenswert. Die sich daraus ergebenden Vorteile wären eine Vereinfachung der Infrastruktur und die Elementarisierung von redundanten Datensätzen. Es ist jedoch zu berücksichtigen, dass weitere Entwicklungen der Funktionalität den Einsatz von Neo4J rechtfertigen könnten. Daher sollte dieses Konzept zu einem späteren Zeitpunkt analysiert werden.

Der aktuelle Stand der Software ist für die Streckenerkennung in Städten ausgelegt. Kreuzungen auf Autobahnen und Landstraßen können mit den entwickelten Algorithmen nicht immer korrekt erkannt werden, da sich deren Aufbau im Vergleich zu Kreuzungen innerhalb von Städten in OSM unterscheiden. Daher ist hier ein Konzept zu erarbeiten, welches die Funktionalität auf außerstädtische Straßenformen erweitert.

6 Umsetzung des Backend-Servers

Auf dem Backend-Server werden die in Abschnitt 2.2 erörterten Drittanbieter-Dienste implementiert. Der Backend-Server stellt hiermit eine Schnittstelle zu den OSM-Daten dar.

6.1 Anforderungen

Bei der Realisierung werden die Drittanbieter-Dienste Gisgraphy, Overpass sowie Neo4J auf dem Backend-Server installiert. Hierbei ergeben sich folgende funktionale Anforderungen:

- sichere Serverumgebung
- schnelle Antwortzeiten
- einfache und schnelle Installation

In den folgenden Abschnitten werden die Installation und die Benutzung der Drittanbieter-Dienste erläutert. Overpass erlaubt OSM-Abfragen, um kartografische Informationen aus der Datenbank zu gewinnen. Gisgraphy dient der Umwandlung von aufgezeichneten GPS-Punkten in OSM-Straßen-IDs. Zunächst wird die Absicherung des Backend-Servers beschrieben.

6.2 Absicherung gegen digitale Angriffe

Wie bereits in Abschnitt 2.3.1 beschrieben, handelt es sich bei dem Backend-Server um einen leistungsstarken vServer. Solche öffentlich erreichbaren Server werden immer wieder angegriffen, um sie dann in ein Botnetz zu integrieren oder als Spamverteiler zu nutzen. Folgender Auszug aus dem Log des Secure Shell (SSH)-Deamon zeigt die sekündlichen Anmeldeversuche beim System.

```

1 16:21:37 sshd: pam_unix(sshd:auth): check pass; user unknown
2 16:21:37 sshd: pam_unix(sshd:auth): authentication failure; [...]
3 16:21:39 sshd: Failed password for invalid user virus from
   121.14.5.125 port 45799 ssh2
4 16:21:39 sshd: Received disconnect from 121.14.5.125: 11: Bye Bye
5 16:21:41 sshd: Invalid user virus from 121.14.5.125
6 16:21:41 sshd: input_userauth_request: invalid user virus
7 16:21:41 sshd: pam_unix(sshd:auth): check pass; user unknown
8 16:21:41 sshd: pam_unix(sshd:auth): authentication failure; [...]
9 16:21:44 sshd: Failed password for invalid user virus from
   121.14.5.125 port 49613 ssh2
10 16:21:44 sshd: Received disconnect from 121.14.5.125: 11: Bye Bye
11 16:21:47 sshd: Invalid user windows from 121.14.5.125
12 16:21:47 sshd: input_userauth_request: invalid user windows

```

Quellcode 26: Auszug aus dem sshd-Log.

Als erster Schritt wird die Absicherung von der Secure Shell vorgenommen. Dafür wird die Konfigurationsdatei des SSH-Deamon `/etc/ssh/sshd_config` angepasst.

```

1 LoginGraceTime 30
2 AllowGroups users
3 PermitRootLogin no

```

Quellcode 27: Auszug aus der Konfigurationsdatei `sshd_config`.

Wie in Quellcode 27 ersichtlich, werden drei wichtige Konfigurationsparameter eingestellt. Der Parameter `LoginGraceTime` beschreibt den Zeitraum nach einem Request, in dem die Verbindung offen gehalten wird. Dieser Wert wurde auf 30 Sekunden reduziert, um möglichen Angreifern weniger Zeit für Passworteingaben zu geben. Mithilfe der Parameter `AllowGroups` und `PermitRootLogin` wird eingestellt, dass sich nur die User aus der Gruppe `users` anmelden dürfen. Somit wird die Anzahl der möglichen Loginnamen gering gehalten. Außerdem kann man somit keinen direkten Rootzugriff auf den Server erlangen.

In einem zweiten Schritt wird eine Firewall²⁸ auf dem Server eingerichtet. Hierfür werden mithilfe von `iptables`²⁹ Kommandos bestimmter Filterregeln festgelegt. Zur ein-

²⁸Sicherungssystem, das einen Server vor unerwünschtem Zugriff über Datenleitungen von außen schützt.

²⁹Kommandozeilenprogramm zur Konfiguration von Paketfiltern.

fachen Einrichtung wird das iptables-Frontend *ufw*³⁰ verwendet. Folgender Quellcode zeigt die Konfiguration der iptables mit dem Frontend *ufw*.

To	Action	From
--	-----	---
22/tcp (OpenSSH)	LIMIT IN	Anywhere
80/tcp (Apache)	ALLOW IN	Anywhere
8080/tcp	ALLOW IN	Anywhere
8017/tcp	ALLOW IN	Anywhere

Quellcode 28: Statusausgabe der ufw-Firewall.

Die erste Regel limitiert die Anfragen an Port 22, über den die SSH-Verbindungen stattfinden. Die Limitierung wird mithilfe des iptables-Moduls *recent* umgesetzt. Durch diese Regel wird eine IP-Adresse für eine Minute gesperrt, wenn innerhalb von 30 Sekunden mehr als sechs Anfragen von dieser ausgehen. Die weiteren Regeln öffnen die Ports für die installierten Services auf dem Server. Alle anderen Ports sind grundsätzlich gesperrt.

6.3 Mapping einer GPS-Koordinate zu einer Straßen-ID mit Gisgraphy

Das freie Geocoding-Framework *Gisgraphy* ist auf dem Backend-Server installiert. Es wird die Beta-Version *V4.0beta1* eingesetzt, da hierbei eine Kompatibilität zu der aktuellen PostGIS-Datenbank 2.0 besteht. Des Weiteren stehen für die Version 4.0 aktuell prozessierte OSM-Daten von Deutschland zu Verfügung. Mithilfe des Entwicklers David Masclet konnte die Beta-Version zum ersten Mal extern kompiliert und installiert werden. Bei der initialen Installation treten noch einzelne Bugs auf, welche in Zusammenarbeit mit dem Entwickler gelöst werden können [42]. Die daraus entstandene Version kann im Rahmen von SAM eingesetzt werden.

Während anfänglicher Tests wiesen die Webservices von Gisgraphy hohe Latenzen auf. Durch die Erzeugung eines Generalized Search Tree (GiST)-Index kann die Volltextsuche in der PostGIS-Datenbank beschleunigt werden. Folglich verringern sich die Latenzen.

³⁰Uncomplicated Firewall

Für den Algorithmus zur Umrechnung von GPS-Punkten zu Kreuzungspunkten (vgl. Abschnitt 5.3.1) wird der Webservice *Street Search* verwendet. Eine Anfrage an den *Street Search*-Endpoint besteht aus folgenden Parametern:

- **lat, lon**

Die geografische Position für das Reverse Geocoding wird durch den Längen- und Breitengrad spezifiziert.

- **from, to**

Festlegung der Anzahl der zurückgegebenen Resultate

- **format**

Festlegung der Notation

- **radius**

Festlegung des Suchradius

- **distance**

Hinzufügen der gemessenen Entfernung zu der jeweiligen Straße

- **streettype**

Einschränkung der Typen von zurückgegebenen Straßen

```
1 http://free.gisgraphy.com/street/streetsearch
2 ?lat=49.50637&lng=8.49139
3 &from=1&to=5
4 &format=JSON
5 &radius=500
6 &distance=true
7 &streettype=MOTROWAY | TRUNK | PRIMARY | SECONDARY | TERTIARY | UNCLASSIFIED
   | RESIDENTIAL
```

Quellcode 29: Request einer *Street Search*-Anfrage.

Der Quellcode 29 zeigt die eingesetzte Anfrage an den REST-Webservice *Street Search*. Die geografische Position mit Längen- und Breitengrad wird übergeben. Die Anzahl der zurückgegeben Resultate wird auf fünf reduziert, um die Rechenzeit auf dem Backend-Server zu minimieren. Als Notation für die Antwort wird ein Objekt in JSON-Notation verwendet, damit keine Umwandlung in der Javascript-Applikation stattfinden muss. Der Radius der Straßensuche wird auf 500 Meter gesetzt, damit nur die geografisch

nächstgelegenen Straßen gefunden werden. Durch das Flag *distance* wird die gemessene Entfernung zu der jeweiligen Straße an die Antwort angehängt. Infolgedessen werden die einzelnen Resultate nach der Entfernung aufsteigend sortiert. Somit muss keine aufwendige Sortierung in der Applikation stattfinden. Der Parameter *streettype* schränkt die Typen von Straßen ein, welche von Gisgraphy gefunden werden. Die einzelnen Straßentypen werden durch die Werte des OSM-Tag-Schlüssels *highway* dargestellt. Mit diesem Schlüssel wird in den OSM-Karten die Verkehrsbedeutung der Straßen eingestuft. Für den Algorithmus zur Umrechnung von GPS-Punkten zu Kreuzungspunkten werden nur Straßen gesucht, welche mit dem Auto befahrbar sind. Das bedeutet, dass nur Straßen berücksichtigt werden, die mindestens einer Gemeindestraße entsprechen. Auf eine solche Anfrage liefert der *Street Search*-Endpoint als Antwort die geografisch nächstgelegenen Straßen (siehe Quellcode 30). Für jede einzelne Straße gibt Gisgraphy eine bestimmte Menge von Informationen zurück. Für den Algorithmus zur Umrechnung von GPS-Punkten zu Kreuzungspunkten wird jedoch nur die *openstreetmapId* benötigt. Diese ID identifiziert die Straße eindeutig in den OSM-Daten.

```

1 {
2     "numFound": 5,
3     "QTime": 10,
4     "result": [
5         {
6             "name": "Ulmenweg",
7             "distance": 1.62213278539163,
8             "gid": 116142362,
9             "openstreetmapId": 152412641,
10            "streetType": "TERTIARY",
11            "oneWay": false,
12            "length": 811.856784142129,
13            "lat": 49.50650672988005,
14            "lng": 8.49167638346584,
15            ...
16        },
17    ...]

```

Quellcode 30: Response einer Anfrage an den *Street Search*-Endpoint.

6.4 Abfragen an die Openstreetmap-Datenbank mittels Overpass

Außerdem wird auf dem Backend-Server die *Overpass-API* installiert. Vor der ersten Benutzung müssen die OSM-Daten in die Template-Datenbank von Overpass importiert werden. Hierfür werden die Importskripte von Overpass verwendet.

Für den Algorithmus zur Umrechnung von GPS-Punkten zu Kreuzungspunkten (vgl. Abschnitt 5.3.1) werden die Kreuzungen auf einer Straße benötigt. Für diese Problemstellung wird folgende Overpass-Anfrage verwendet.

```

1 [out:json];
2 way(235125011)->.street;
3 (node(w.street);)->.nodes;
4 foreach.nodes(
5   out body;
6   (
7     way
8       (bn)
9         ["highway"~"motorway|trunk|primary|secondary|tertiary|
10           unclassified|residential"];
10 )->.otherStreets;
11   out count;
12   .otherStreets out body;
13 );

```

Quellcode 31: Overpass-Query zur Kreuzungsfindung.

Der Quellcode 31 zeigt die verwendete Anfrage an die Overpass-API. In der folgenden Abbildung 34 ist die Funktionsweise der Anfrage in einem Schema kurz zusammengefasst.

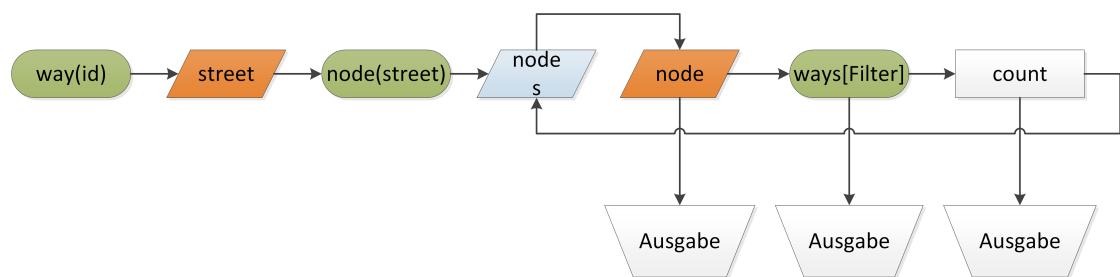


Abbildung 34: Schema der Overpass-Anfrage.

Zum Beginn wird mit dem Query-Statement `way()` die Straße zu einer bestimmten OSM-Way-ID in das Set `.street` geladen. Mit einem zweiten Query-Statement `node()` werden die sich auf dieser Straße befindenden Knoten in das Set `.nodes` geladen. Mithilfe einer Schleife wird über dieses Set von Knoten iteriert. Das Filter-Statement `way(bn)` selektiert die angrenzenden Straßen zu dem aktuellen Knoten. Diese Selektion wird durch einen weiteren Filter eingeschränkt, welcher nur Straßen erlaubt, die einem gewissen Straßentyp entsprechen. Für den Algorithmus zur Umrechnung von GPS-Punkten zu Kreuzungspunkten werden nur Kreuzungen gesucht, die sich auf das Fahrverhalten auswirken. Das Statement `count` zählt die Elemente in dem Eingabeset. In diesem Fall werden die angrenzenden Straßen aufsummiert. Ausgegeben werden in jedem Schleifendurchlauf der aktuelle Knoten, die Anzahl der angrenzenden Straßen sowie weitere Angaben zu den Straßen. Der Knoten stellt hierbei die einzelne Kreuzung mit den sich dort kreuzenden Straßen dar.

Der Quellcode 32 zeigt einen Ausschnitt aus der Antwort auf die vorher beschriebene Overpass-Anfrage. Die drei Ausgabenbereiche sind deutlich zu erkennen. Zuerst werden die Informationen zu dem Knoten, der einer Kreuzung entspricht, geliefert. Hierbei wird die OSM-Node-ID sowie die geografische Position der Kreuzung zurückgegeben. Anschließend kann die Anzahl der angrenzenden Straßen ausgelesen werden. Diese Information wird benötigt, um die nachfolgenden Straßen einfacher parsen zu können. Zuletzt werden die einzelnen Straßen mit ihren Metadaten übertragen.

```

1 { "type": "node", "id": 1656889223,
2   "lat": 49.5076376, "lon": 8.4916121
3 },
4 {
5   "count": { "ways": 2, ...}
6 },
7 {
8   "type": "way", "id": 180312562,
9   "nodes": [...],
10  "tags": {"name": "Clara-Reimann-Strasse", ...} ,
11 {
12   "type": "way", "id": 235125011,
13   "nodes": [...],
14  "tags": {"name": "Friedrich-Traumann-Strasse", ...}}

```

Quellcode 32: Response eines Overpass-Query zur Kreuzungsfindung.

6.5 Evaluation

Auf dem Backend-Server sind die Drittanbieter-Dienste erfolgreich installiert. Durch die implizierten Updates von Gisgraphy ist nun eine einfache Installation möglich. Durch einzelne Modifikationen der Anfragen und das Erzeugen des GiST-Indexes antworten die Dienste mit kurzer Reaktionszeit. Die Anzahl der Angriffe auf den Backend-Server verringert sich durch die Absicherungsmaßnahmen. Die in Abschnitt 6.1 definierten funktionalen Anforderungen sind somit vollständig erfüllt.

Zur effizienteren Nutzung der Dienste ist es erstrebenswert, die Infrastruktur anzupassen. Statt eines kleinen Servers für die Webanwendung und eines großen Backend-Servers könnten alle Dienste auf kleineren Servern laufen, da keine Abhängigkeiten zwischen den einzelnen Diensten vorhanden sind. Wenn die einzelnen Server innerhalb eines Rechenzentrums gemietet sind, würden sich Vorteile wie kurze Laufzeiten, optimierte Ausnutzung von Ressourcen und geringere Kosten ergeben. Zusätzlich sollte die Verwendung von Containerlösungen zur Migration von Diensten auf einen anderen Server diskutiert werden. Dazu zählt unter anderem die Möglichkeit, Docker [43] zu verwenden. Hierzu müsste vorab abgewägt werden, ob eine solche Software die gewünschten Vorteile für den vorliegenden Anwendungsfall mit sich bringt.

Des Weiteren sollte eine regelmäßige Aktualisierung der importierten OSM-Daten stattfinden. Hierfür müssen sinnvolle und effiziente Konzepte entwickelt werden.

7 Fazit und Ausblick

In diesem Kapitel wird ein Fazit zur Entwicklung gezogen und ein Ausblick zu der Frage gegeben, wie die Weiterarbeit an dem Projekt aussehen kann.

7.1 Fazit

Mit der Studienarbeit „Streckenabhängige Verbrauchsmessung und -analyse“ wurden ein Konzept und ein Prototyp entwickelt, welche - im Vergleich zu anderen Techniken - eine effizientere Routenfindung ermöglichen. Dazu wurde zunächst ein Konzept entwickelt, welches die Grundlage für die Arbeit darstellt. Nach der Definition des angestrebten Ziels wurde eine Infrastruktur eingerichtet, welche für Tests, Entwicklung und das Produktivsystem Verwendung findet. Dies beinhaltet die Installation verschiedener Dienste auf dem Backend Server. Darauf aufbauend wurde die Serveranwendung implementiert und über einen Webserver mit dem Backend verbunden. Zur laufenden Entwicklung gehörte zudem die Umsetzung einzelner Unit-, Integrations- sowie Anwendungstests.

Das Frontend ist als App für die Microsoft Windows Phone 8.1-Plattform entwickelt worden und bietet dem Nutzer den Zugriff auf die Schnittstellen der Serveranwendung. Außerdem dient es der Messdatenerfassung und erfüllt somit alle geforderten Anforderungen.

Das Backend wurde mit der Node.js-Plattform serverseitig umgesetzt. Die Persistierung der Daten erfolgt mittels einer Neo4j-Datenbank. Damit wurde das Ziel erreicht, eine Anwendung zu entwickeln, die aktuelle Technik für die Implementation verwendet. Die Tragfähigkeit von Node.js und Neo4j für SAM ist bewiesen. Des Weiteren konnten im Umgang mit den Technologien wertvolle Erfahrungen gesammelt werden. Wie in Abschnitt 5.4 angeführt, werden derzeit nicht alle Vorteile der verwendeten Komponenten ausgenutzt. Ferner bestehen Einschränkungen bei der Software im Hinblick auf die Funktionalität im städtischen Verkehr. Zur Demonstration für den vorliegenden Prototypen ist dieser Entwicklungsstand zufriedenstellend.

Das Ergebnis der Arbeit ist ein Prototyp, der die gestellten Anforderungen umsetzt. Daten werden personenbezogen mittels einer Nutzerverwaltung gesichert. Die Daten basieren auf Messwerten, die mithilfe des OBD-Systems des Fahrzeugs für eine Strecke aufgezeichnet und persistiert werden.

Auf Basis dieser Analyse kann dem Nutzer somit für einen Tag die effizienteste und schnellste Strecke angezeigt werden. Zusammenfassend ist festzuhalten, dass die im Rahmen der Studienarbeit gesetzten Ziele erreicht wurden. Aus diesen Gründen kann die Arbeit mit einem positiven Fazit abgeschlossen werden.

7.2 Ausblick

Für die weitere Entwicklung sollte die Bewertung der Routen auch in Abhängigkeit der Zeit erfolgen. Das bedeutet, dass es ermöglicht werden soll, die voraussichtlich benötigte Zeit für jede Strecke einer Route zu interpolieren. Dabei beruht die Kalkulation auf stochastischen Berechnungen. In einem weiteren Schritt können die ermittelten Daten mit denen anderer Nutzer verglichen werden, um so für jeden Nutzer eine optimale Route und Zeit zu finden. So könnten Kraftstoff- und Zeiteinsparungen zur Vermeidung von Staus führen und der CO₂-Ausstoß noch weiter verringert werden.

Aktuell beruht die Bewertung der Daten auf der Berechnung gefahrener Strecken. Mit einer größeren Nutzerbasis könnten im Rahmen von Big Data fehlende Abschnitte für eine Strecke berechnet werden. Dazu wird die Betrachtung einer Strecke von Kreuzung zu Kreuzung beibehalten. Nun können neue Strecken durch die Verknüpfung von Kreuzungspunkten anderer Nutzer erstellt werden und so neue, effizientere Routen ermittelt werden. Mit diesem Vorgehen könnte die Routenfindung aktiv durch die Software umgesetzt werden. Es ist jedoch darauf zu achten, dass die Eigenschaftswerte eines anderen Nutzers nicht mehr als um ein definiertes Delta abweichen. Dazu könnte eine Klassifizierung eingeführt werden. Hierzu sind jedoch weitreichende Diskussionen bezüglich des Datenschutzes zu führen.

Eine weitere Verbesserung würde die Verwendung der Daten in Echtzeit bedeuten. So könnte die Vorhersage durchgängig aktualisiert und den aktuellen Verkehrsverhältnissen angepasst werden. Dazu könnten erneut die Daten als ähnlich klassifizierter Nutzer verwendet werden. Schlussendlich würde damit eine Anwendung realisiert, welche eine Routenfindung basierend auf dem persönlichen Fahrstil, dem Fahrzeug sowie der Tageszeit und Verkehrslage in Echtzeit ermöglicht. Dabei kann das bereits implementierte Muster der Berechnung der Werte von Kreuzung zu Kreuzung grundsätzlich beibehalten werden. Hier ist das Potenzial zur Einsparung von CO₂ und die Verminderung von Staugefahr im städtischen Verkehr zu erforschen.

Literatur- und Quellenverzeichnis

- [1] National Bureau of Statistics of China. Fahrzeugbestand in China in den Jahren 2001 bis 2013 (in Millionen). <http://de.statista.com/statistik/daten/studie/219921/umfrage/anzahl-der-fahrzeuge-in-china/>, Abruf: 03. Juni 2015.
- [2] OICA. Anzahl registrierter Kraftfahrzeuge weltweit in den Jahren 2005 bis 2013 (in 1.000). <http://de.statista.com/statistik/daten/studie/244999/umfrage/weltweiter-pkw-und-nutzfahrzeugbestand/>, Abruf: 03. Juni 2015.
- [3] Bundesministerium für Umwelt, Naturschutz, Bau und Reaktorsicherheit (BMUB). Die EU-Verordnung zur Verminderung der CO₂-Emissionen von Personenkraftwagen . http://www.bmub.bund.de/fileadmin/bmu-import/files/pdfs/allgemein/application/pdf/eu_verordnung_co2_emissionen_pkw.pdf, Abruf: 04. Juni 2015.
- [4] Prinz & Partner Patentanwälte. Verfahren und Vorrichtung zum Bestimmen einer Route mit einem geschätzten minimalen Kraftstoffverbrauch für Fahrzeuge. 2008.
- [5] Fabian Schmid. Verbrauchsoptimierte Fahrverhalten durch Einsatz mobiler Technologien unter Berücksichtigung sozialer Komponenten. Master's thesis, Fachhochschule Hagenberg, 2013.
- [6] Statista. Marktanteile der mobilen Betriebssysteme am Absatz von Smartphones in Deutschland von Januar 2012 bis April 2015. <http://de.statista.com/statistik/daten/studie/225381/umfrage/marktanteile-der-betriebssysteme-am-smartphone-absatz-in-deutschland-zeitreihe/>, Abruf: 6. Juni 2015.
- [7] Allgemeine Geschäftsbedingungen (AGB). netcup GmbH. <https://www.netcup.de/bestellen/agb.php>, Abruf: 6. Juni 2015).

- [8] Netcup. Bestellseite. <https://www.netcup.de/bestellen/produkt.php?produkt=1020>, Abruf: 20. Januar 2015.
- [9] Neo4j Staff. Social networks in the database: using a graph database. <http://neo4j.com/blog/social-networks-in-the-database-using-a-graph-database/>, Abruf: 10. Juni 2015).
- [10] Patil, Kiran Raosaheb and Nielsen, Jens. Uncovering transcriptional regulation of metabolism by using metabolic network topology. *Proceedings of the National Academy of Sciences of the United States of America*, 102(8):2685–2689, 2005.
- [11] Dijkstra, Edsger W. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.
- [12] Hart, Peter E and Nilsson, Nils J and Raphael, Bertram. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.
- [13] Hunger, M. *Neo4j 2.0: Eine Graphdatenbank für alle*. Schnell + kompakt. Entwickler.Press, 2014.
- [14] Robinson, I. and Webber, J. and Eifrem, E. *Graph Databases*. O'Reilly Media, 2nd edition, 2015.
- [15] Neo Technology, Inc. Neo4j Customers. <http://neo4j.com/customers/>, Abruf: 6. Juni 2015).
- [16] Zimmermann, Werner AND Schmidgall, Ralf. *Bussysteme in der Fahrzeugtechnik* - . Springer-Verlag, Berlin Heidelberg New York, 5. Aufl. edition, 2014.
- [17] ISO. 15031 - Road vehicles – Communication between vehicle and external equipment for emissions-related diagnostics, 2014.

- [18] ELM327 OBD to RS232 Interpreter. <http://elmelectronics.com/DSheets/ELM327DS.pdf>, Abruf: 12. April 2015.
- [19] Microsoft. Was ist eine Windows-Runtime-App? <https://msdn.microsoft.com/library/windows/apps/dn726767.aspx>, Abruf: 27. Januar 2015.
- [20] Microsoft. Asynchrone Programmierung mit Async und Await. <https://msdn.microsoft.com/de-de/library/hh191443.aspx>, Abruf: 2. Juni 2015.
- [21] Thomas Bayer. REST Web Services. <http://www.oio.de/public/xml/rest-webservices.htm>, Abruf: 15. Januar 2015.
- [22] Erik Wilde und Cesare Pautasso. *REST: From Research to Practice*. Springer Verlag, 2011.
- [23] Wandschneider, M. *Learning Node.js: A Hands-On Guide to Building Web Applications in JavaScript*. Learning. Addison-Wesley Verlag, 2013.
- [24] NPM Registry. The JavaScript Package Registry. <https://docs.npmjs.com/misc/registry>, Abruf: 20. Januar 2015.
- [25] Jetbrains. Webstorm The smartest JavaScript IDE. <https://www.jetbrains.com/webstorm/>, Abruf: 20. Januar 2015.
- [26] Chao ZHOU. RESTClient, a debugger for RESTful web services. https://addons.mozilla.org/de/firefox/addon/restclient/?src=collection&collection_id=10146e50-da6f-4a44-8fe4-ee919bd8aa2e, Abruf: 20. Januar 2015.
- [27] Flanagan, D. *JavaScript: The Definitive Guide*. Definitive Guides. O'Reilly Media, 6th edition, 2011.
- [28] OpenStreetMap Wiki. Database. <http://wiki.openstreetmap.org/wiki/PostgreSQL>, Abruf: 18. Januar 2015.

- [29] OpenStreetMap Wiki. Databases and data access APIs. http://wiki.openstreetmap.org/wiki/Databases_and_data_access_APIs, Abruf: 18. Januar 2015.
- [30] Goldberg, Daniel W. A geocoding best practices guide. *Springfield, IL: North American Association of Central Cancer Registries*, 2008.
- [31] Open Knowledge Foundation. Open Data Commons Open Database License (ODbL). <http://opendatacommons.org/licenses/odbl/1.0/>, Abruf: 7. März 2015.
- [32] Free Software Foundation. GNU Affero General Public License. <http://www.gnu.org/licenses/agpl-3.0.html>, Abruf: 7. März 2015.
- [33] Free Software Foundation. GNU Lesser General Public License, version 3. <https://www.gnu.org/licenses/lgpl-3.0.en.html>, Abruf: 14. April 2015.
- [34] Open Source Initiative. The MIT License (MIT). <http://opensource.org/licenses/mit-license.php>, Abruf: 1. Juni 2015.
- [35] Free Software Foundation. GNU General Public License, version 3. <http://www.gnu.org/licenses/gpl-3.0.html>, Abruf: 7. März 2015.
- [36] Microsoft. Windows.Web.Http namespace. <https://msdn.microsoft.com/en-us/library/windows/apps/windows.web.http.aspx>, Abruf: 1. Juni 2015.
- [37] Adriano Alessandrini, Francesco Filippi, Fernando Ortenzi . Consumption calculation of vehicles using OBD data. <http://www.epa.gov/ttnchie1/conference/ei20/session8/aalessandrini.pdf>, Abruf: 4. Juni 2015).
- [38] Uberspace. Offizielle Internetseite. <https://uberspace.de/>, Abruf: 20. Januar 2015.

- [39] StrongLoop. Express ADPI 4.x. <http://expressjs.com/4x/api.html>, Abruf: 20. Januar 2015.
- [40] yifeed. ExpressJs + Passport.js + MySQL Authentication. <http://yifeed.com/passportjs-mysql-expressjs-authentication.html>, Abruf: 20. Januar 2015.
- [41] Jared Hanson. Passport. <http://passportjs.org/>, Abruf: 20. Januar 2015.
- [42] D. Masclet. Persönliche Mitteilung, 2015.
- [43] Docker Website. <https://www.docker.com/>, Abruf: 09. Juni 2015.