

Hochschule Ravensburg-Weingarten

Fakultät Elektrotechnik und Informatik

Studiengang Informatik

GTFS-basierte Live-Visualisierung des Öffentlichen Nahverkehrs Stuttgart



Master-Thesis

zur Erlangung des akademischen Grades

Master of Science

Andreas Lorer

September, 2017

Prof. Klemens Ehret

Prof. Benedikt Groß

Zusammenfassung

In einer Zeit, in der Städte aufgrund von zunehmendem Individualverkehr und damit einhergehender Umweltverschmutzung mehr und mehr an ihre Grenzen stoßen, wird die Verbesserung der Öffentlichen Verkehrsinfrastruktur immer wichtiger. Die Visualisierung von Daten des Öffentlichen Nahverkehrs kann hierbei einen Beitrag leisten, indem sie bspw. Pendlern und Reisenden als anschauliche Informationsquelle oder Städteplanern und Verkehrsunternehmen als Analyseinstrument und Planungsgrundlage dient. Im Rahmen dieser Arbeit wurde ein Ansatz zur Visualisierung von Nahverkehrsdaten im GTFS-Format auf einer interaktiven Karte entwickelt. Das Ergebnis ist eine Live-Karte für den Raum Stuttgart, mit der auf einfache und praktikable Weise die Daten aller Fahrpläne des Verkehrsverbunds einsehbar werden.

Inhaltsverzeichnis

1 Einleitung	4
1.1 Hinführung zum Thema	4
1.2 Gliederung der Arbeit	6
2 Discover	8
2.1 Related Work	8
2.2 Visualisierungsmöglichkeiten und Zielgruppe	9
2.3 Mögliche Datengrundlage und Format	11
2.3.1 Einsatz von GPS-Daten	11
2.3.2 GTFS-Fahrplandaten	12
2.3.3 GTFS-Realtime	16
3 Define	18
3.1 Kartenart	18
3.2 Wahl der Datengrundlage	18
3.3 Gewählte Technologien	19
3.3.1 Datenbank	19
3.3.2 Serverwahl	19
3.3.3 Kartenmaterial	20
3.3.4 Frameworks	20
3.4 Zielsetzung	21
3.5 Begriffe und Definitionen	21
3.5.1 Time	21
3.5.2 Vehicle	22
3.5.3 Polyline	22
3.5.4 Station	22
3.5.5 Trip	22
3.5.6 Route	23
4 Develop	24
4.1 Anzeigen einer Polyline	24
4.2 Hinzufügen der Stationen	25
4.3 Animieren eines Vehicles durch Interpolation	28
4.4 Zeichnen aller Polylines	30
4.4.1 Ramer–Douglas–Peucker	30
4.4.2 Aggregieren der Shape-Tabelle	32
4.4.3 Polyline Encoding	32
4.5 Anzeigen aller Stationen	33
4.5.1 GTFS-Optimierungen	33
4.5.2 Denormalisierung der Datenbank	34

4.6	Animieren der aktiven Trips	37
5	Deliver	41
5.1	Funktionsprinzip	41
5.2	Funktionen der Webanwendung	44
5.3	Performance-Ergebnisse	48
6	Fazit	51
7	Anhang	53
	Abbildungsverzeichnis	63
	Tabellenverzeichnis	64
	Listingverzeichnis	64
	Algorithmenverzeichnis	64
	Literatur	65

1 Einleitung

Diese Einleitung gliedert sich in zwei Abschnitte. Während der erste Abschnitt der Einführung in das Thema dient, sollen im zweiten Abschnitt die Gliederung und Vorgehensweise der Ausarbeitung beschrieben werden.

1.1 Hinführung zum Thema

Datenvisualisierung ist ein Thema, das nicht nur in jüngster Zeit sehr viel Zuwendung fand, sondern auch für die Analyse verschiedenster Sachverhalte immer wichtiger wird. So lassen sich komplexe Zusammenhänge eines Systems oftmals erst dann richtig begreifen, wenn wir alle möglichen Zustände davon erfassen können. In „Up and Down the Ladder of Abstraction“, beschreibt Bret Victor, wie sich Systeme in ihrer Ganzheit besser begreifen und gestalten lassen.

„When designing [a system], the challenge lies not in constructing the system, but in understanding it. In the absence of theory, we must develop an intuition to guide our decisions. The design process is thus one of exploration and discovery.“ [14]

Auch eine Ansammlung an Daten ist erst einmal sehr abstrakt. In einer Datenbank in Relation gebracht, bleiben die meisten Erkenntnisse und Stories hinter diesen Daten verborgen. Ein tieferes Verständnis erhalten wir erst dann, wenn wir sie auswerten. Eine mögliche Form einer solchen Auswertung stellt die Datenvisualisierung dar. Die Art der Datenvisualisierung hat sich dabei in den letzten Jahren stark gewandelt. Während Daten anfangs vor allem in Form von Häufigkeitsanalysen ausgewertet und als Bar- oder Linechart visualisiert wurden, finden wir heute vermehrt interaktive Karten, mit denen raumbezogene Zusammenhänge veranschaulicht werden können. Dieser Trend von dynamischen Live-Visualisierungen fand in verschiedenen Bereichen Einzug. Zum Beispiel der erst kürzlich erschienene Flight-Planner des Dubai Airports¹, die Marine-Traffic-Map² in der Schifffahrt oder auch für die Live-Simulation von Wetterdaten³.

Auch hier in Deutschland gibt es verschiedene Produkte, die veröffentlicht wurden. Beispielsweise der Zugradar⁴ (2014) und der Busradar⁵ der Deutschen Bahn oder eine Karte für das S-Bahn Netz in München⁶ (2009).

Eine gesamte Erfassung des Öffentlichen Verkehrs ging ebenfalls 2014 mit Travic⁷ online und bietet mit über 650 integrierten Fahrplänen eine enorme Abdeckung. Da die Live-Karten der Deutschen Bahn damals nur die Visualisierung der eigenen Bus- und Bahnlinien ermöglichte,

¹<http://www.dubaiairports.ae/flight-planner>

²<https://www.marinetraffic.com/>

³<https://www.windy.com/>

⁴<http://bahn.de/zugradar>

⁵<https://play.google.com/store/apps/details?id=de.hafas.android.dbbusradar>

⁶<http://s-bahn-muenchen.hafas.de/>

⁷<http://tracker.geops.de/>

war Travic bestrebt, diese Lücke zu schließen und den gesamten Öffentlichen Verkehr darzustellen. Zusätzlich sei noch LiveMap24⁸ von Verdicht erwähnt, deren Veröffentlichungsdatum allerdings nicht bekannt ist.

Der Vorteil einer digitalen Karte besteht in seiner ständigen Aktualität. Ein statischer Fahrplan kann keine Informationen zu Störungen, Verspätungen oder Ausfall eines Zuges geben. Auf einer Live-Karte lassen sich solche Informationen visuell aufbereiten und dem Anwender über die Benutzeroberfläche vermitteln. Der Betrachter kann einsehen, wie viele Fahrzeuge gerade aktiv sind und kann zusätzlich zum statischen Fahrplan auch visuell erleben, wo sich sein Zug oder Bus befindet. Da gleichzeitig Fahrplan als auch Karte vorhanden sind, ist auch eine geographische Orientierung möglich. Dem Satz „Der Zug hat 5 Minuten Verspätung“ wird ein visueller Kontext gegeben und ist dadurch nicht mehr nur eine Aussage, sondern wird visuell erfahrbar. Den Mehrwert einer Live-Karte sehen wohl aber nicht nur Pendler oder Reisende, sondern auch Verkehrsunternehmen, Städteplaner oder Verkehrsforcher.

Diese Arbeit befasst sich umfassend mit der Entwicklung einer Live-Karte für den Öffentlichen Nahverkehr im Raum Stuttgart als Webanwendung. Der Fokus liegt dabei auf der Exploration verschiedener Visualisierungen und Funktionen, welche die Karte interaktiv gestalten und gleichzeitig die User-Experience erhöhen. Das Ergebnis dieses Vorhabens soll bereits vorab vorgestellt werden und ist deshalb als Screenshot in Abbildung 1 zu sehen.

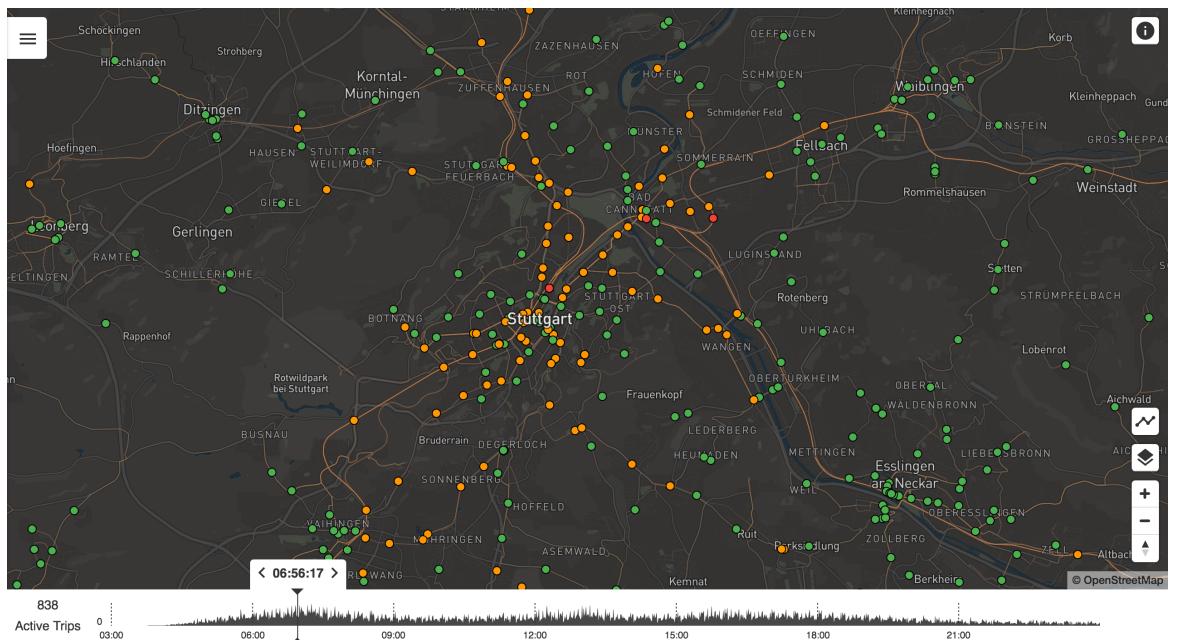


Abbildung 1: Screenshot der fertigen Webanwendung

⁸<https://www.livemap24.com/>

1.2 Gliederung der Arbeit

Das Projekt wurde nach dem Design-Prinzip „The Double Diamond“ bearbeitet, welcher der vorliegenden schriftlichen Ausarbeitung auch als Gliederungsgrundlage dient. Der Double Diamond wurde vom British Design Council 2005 entwickelt und soll im Folgenden kurz beschrieben werden.[2]

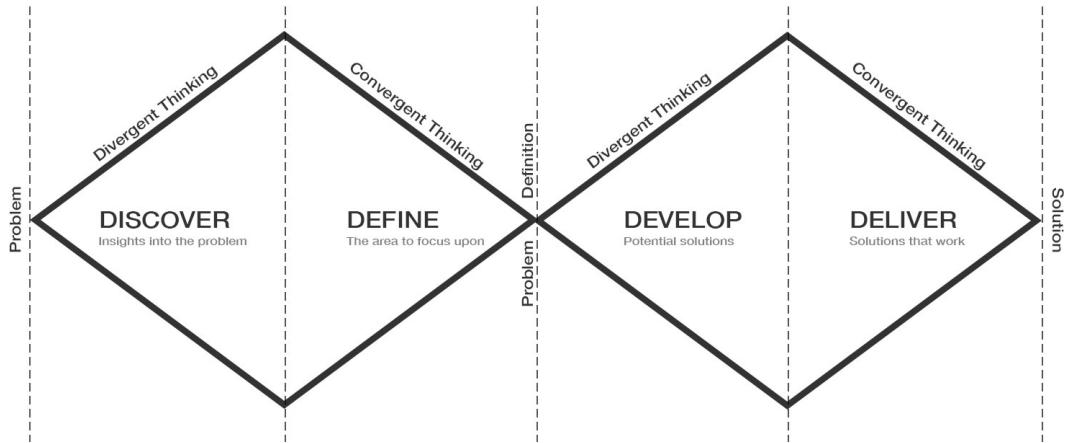


Abbildung 2: „The Double Diamond“ eigene Abbildung nach [2]

Der Double Diamond beschreibt einen iterativen Prozess. Wie in allen kreativen Prozessen werden dabei eine Reihe von möglichen Ideen geschaffen („divergentes Denken“), bevor sie verfeinert und auf die beste Idee reduziert werden („konvergentes Denken“). Der Double Diamond zeigt jedoch an, dass dies zweimal geschieht - einmal zur Bestätigung der Problemdefinition und einmal zur Erstellung der Problemlösung. Einer der größten Fehler wäre es, den linken Diamanten und damit auch die Zieldimension zu vernachlässigen und am Ende womöglich das falsche Problem zu lösen.

Discover: Der erste Teil des Double Diamond steht am Anfang des Projektes. Hier wird versucht, die Welt neu zu sehen, Neues wahrzunehmen und Einsichten in das zu lösende Problem zu sammeln.

Define: Der zweite Teil stellt die Definitionsphase dar. Dabei wird versucht, alle in der Entdeckungsphase identifizierten Möglichkeiten zu verstehen. Ziel ist es, ein klares Briefing zu entwickeln, das die grundsätzlichen Herausforderungen umreißt.

Develop: Der dritte Teil markiert eine Entwicklungsphase, in der Lösungen oder Konzepte erstellt, prototypisiert, getestet und iteriert werden. Dieser Prozess des Ausprobierens hilft, Ideen zu verbessern und zu verfeinern.

Deliver: Der letzte Teil des Double Diamond ist die Lieferphase, in der das daraus resultierende Projekt (z. B. ein Produkt, eine Dienstleistung oder eine Umwelt) abgeschlossen, produziert und in Betrieb genommen wird.

Auf den Aufbau dieser Arbeit bezogen, werden diese Phasen auch für die Bezeichnung der Kapitel verwendet, in denen der Problemlöseprozess für die Entwicklung der Webanwendung beschrieben wird. Im ersten Kapitel sollen die grundlegenden Recherchearbeiten beschrieben sowie mögliche Lösungszenarien diskutiert werden, bevor im zweiten Kapitel die zu erreichen- den Ziele und zu lösenden Probleme vor dem Hintergrund bereits getroffener Entscheidungen festgelegt werden. Das Kapitel Develop soll der Darstellung des konkreten technischen Ent- wicklungsprozesses und der Problemlösung dienen. Das letzte Kapitel Deliver gibt schließlich einen Gesamtüberblick über das erstellte Produkt, seinen Funktionen und Leistungen. Im Fa- zit werden die Ergebnisse des Projekts vor allem im Hinblick auf mögliche Nutzer diskutiert.

2 Discover

In der Discover Phase soll der Blickwinkel möglichst weit geöffnet werden. Dafür erfolgte eine weitreichende Recherche zu relevanten Themenbereichen wie Live-Visualisierung, Visualisierung von Öffentlichem Nahverkehr sowie Tools zum Erstellen von Karten. Darüber hinaus wurden mögliche Datenformate auf ihre Vor- und Nachteile untersucht. Die wichtigste Frage stellt sich allerdings hinsichtlich der möglichen Zielgruppe, die von einer Visualisierung von Echtzeitdaten profitieren könnte. Nach dieser müssen sich schließlich die zu implementierenden Funktionen und Features richten. Die wichtigsten Überlegungen und Rechercheergebnisse zu diesen Aspekten sollen in den folgenden Abschnitten beschrieben werden.

2.1 Related Work

Es existieren mehrere Publikationen, welche für diese Arbeit Relevanz haben oder die für dieses Thema wertvolle Informationen bereitgestellt haben. Die wichtigsten werden in diesem Abschnitt vorgestellt.

Einen ersten guten Überblick zum momentanen Stand von Transportvisualisierungen bietet das Paper „*Visualizing Public Transport Systems: State-of-the-Art and Future Challenges*“[9] von Massimo De Marchi. In seiner Arbeit wird dargelegt, welche visuellen Lösung bereits bestehen und welche Stärken & Schwächen sie haben. Dabei wird auf zwei unterschiedliche Benutzertypen eingegangen. Namentlich als „Traveler“ und „Transportation Researcher“ benannt. Er beschreibt dabei, welche verschiedenen Eigenschaften die jeweilige Nutzerrolle besitzt und welche Bedürfnisse diese hat.

[10] beschreibt sehr gut die Vorgehensweise und verwendeten Technologien für die Umsetzung der Visualisierung von „*An interactive exploration of Boston's subway system*“⁹. Es werden dabei sowohl die verwendeten Tools beschrieben als auch Mockups gezeigt, die Einblicke in den Arbeitsprozess gewähren. Die resultierenden Visualisierungen und deren Detailgrad sind sehr eindrucksvoll und lassen einen tiefen Einblick in das Verkehrsnetz zu.

Eine weitere wertvolle Quelle ist die Arbeit von Patrick Brosi „*Real-Time Movement Visualization of Public Transit Data*“[1]. Brosi stellt dabei die Entwicklung von Travic vor, der bereits eingangs¹⁰ erwähnt wurde. Seine Arbeit zeigt einen Lösungsweg für die Visualisierung von tausenden Fahrzeugen auf einer interaktiven Karte. Interessant sind dabei die diskutierten Vor- und Nachteile von verschiedenen Lösungsansätzen.

⁹<http://mbtaviz.github.io/>

¹⁰siehe Kapitel 1.1

2.2 Visualisierungsmöglichkeiten und Zielgruppe

Die Recherchearbeit führte zu einer Sammlung an verschiedensten Materialien. Live-Karten, Plugins, Tube-Maps aber auch wissenschaftliche Arbeiten die an das eigene Thema angrenzen wurden gesucht und in verschiedene Listen übernommen. In Abbildung 3 ist ein Teil dieser Materialsammlung zu sehen.

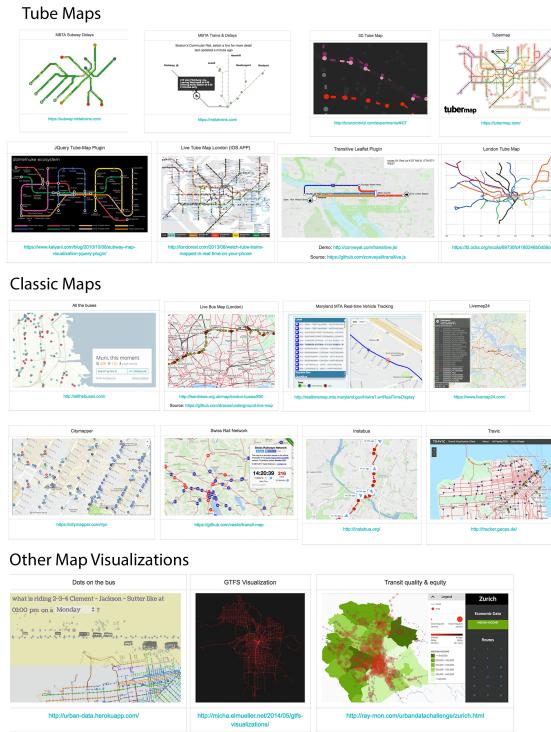


Abbildung 3: Überblick über bestehende Tools und Visualisierungen

Das gesammelte Material zu den möglichen Visualisierungsarten wurde in einer 2x2 Matrix (Abbildung 4) in die Unterkategorien „Live-Karten, Künstlerische Visualisierung, Plugin / Software / Tool, Tube-Map“ eingeordnet, um einen sortierten Gesamtüberblick zu bekommen.

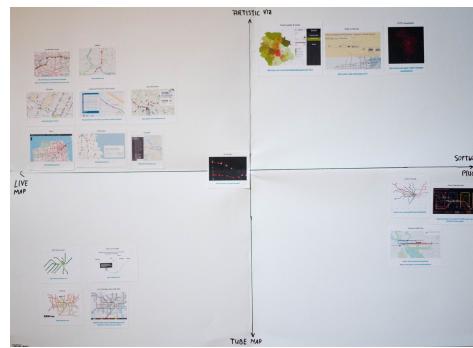


Abbildung 4: 2x2 Matrik Methode auf DIN A2

Durch diese Ansicht wurde die Erkenntnis gewonnen, dass es schon viele Live-Visualisierungen

auf interaktiven Karten gibt, aber nur sehr wenig so genannte Tube-Maps. Auch sind bereits verschiedenste Tools zum Generieren von GTFS-basierten Visualisierungen vorhanden. Eine sehr ausführliche, aber bei weitem nicht vollständige Liste über das Thema „Transit“ wurde auf Github von der Community zusammengetragen <https://github.com/luqmaan/awesome-transit>.

Damit die verschiedenen existierenden Zielgruppen besser verstanden werden können, wurden sogenannte **Personas** eingesetzt[3]. Personas sind fiktive Charaktere, die auf der Grundlage von Recherche erstellt werden, um die verschiedenen Benutzertypen eines Produktes oder Service zu simulieren. Sie können dabei helfen, einen Perspektivenwechsel zu vollziehen und sich in verschiedene Nutzer hineinzuversetzen, um ihre unterschiedlichen Bedürfnisse, Ziele und Erwartungen besser zu verstehen. Für die Charakterisierung werden typische Merkmale, Stereotype oder Trends genutzt. Wie in Abbildung 5 zu sehen ist, wurden in der Recherche-Phase drei mögliche Nutzergruppen identifiziert.

 <i>Icon credits: CC BY 3.0 http://www.freepik.com</i>	Silvia - Commuter <ul style="list-style-type: none"> • 24 years old • Single • Studies Environmental Science • Is a commuter who takes the U- / S-lines to university • She doesn't need a timetable because the frequency of her line is ever so regularly. However she sometimes misses her tram because it's too early 	Questions to ask: <ul style="list-style-type: none"> • What happens when she needs to take another line than usual and maybe to a late or unusual time? • What information would be beneficial for Silvia to increase her trust into public transit services? • How does she get schedule information? <ul style="list-style-type: none"> • Traditional timetables at a station? • Does she use one of the most popular Apps (e.g. DB-Navigator or Öff) or does she use an app from one of the local service vendors (e.g. VVS) • Where would she see a live map that informs her about delays and the state of the transport system? Where could it fit into her life? • Are temporary transportation issues a problem? How is that information displayed?
 <i>Icon credits: CC BY 3.0 http://www.freepik.com</i>	Tim - Traveler <ul style="list-style-type: none"> • 39 years old • Lives in Augsburg with his wife and 2 kids • He and his family often takes the train to visit his parents in Karlsruhe. To not get into any surprises he plans trips carefully via a popular web platform. Printing out the tickets together with the schedule and Tim is good to go. 	Questions to ask: <ul style="list-style-type: none"> • How would a live transit map be beneficial for Tim? • How would he like to use it? • Which benefits does the live map mean for him? • While there is a good amount of information on his train travel, the real-time information suddenly gets more thin the moment Tim and his family are arriving on their target location. What would help to solve this information gap? • Bad signal or low bandwidth are effecting travelers like Tim. What would this mean for the App? • Using the DB-Navigator App he will get push notifications about delays and issues affecting his route. What other information could he need?
 <i>Icon credits: CC BY 3.0 http://www.freepik.com</i>	Paul - Transportation Researcher <ul style="list-style-type: none"> • 32 years old • Works for a city in a planning and research role. • Uses data driven decision making • Tries to understand systems in its whole • Needs to present and argue his findings 	Questions to ask: <ul style="list-style-type: none"> • What data does he need for his work? • Which data would he find interesting? • What are the key control elements he might need to solve his problems? • What are the insights that he could take away from a live map? • What data is not just interesting for him, but would also interest other users?

Abbildung 5: Personas

2.3 Mögliche Datengrundlage und Format

Für eine Visualisierung von Öffentlichen Verkehrsmittel wären mehrere Ansätze denkbar. Einige davon sollen nachfolgend beschrieben werden.

2.3.1 Einsatz von GPS-Daten

Eine Möglichkeit ist eine GPS-basierte Visualisierung der Echtzeitpositionen der Vehicle. Dies ist eine exakte Abbildung des momentanen Zustandes des Verkehrsnetzes und entspricht am ehesten einer Abbildung des realen „Ist“-Zustands. Die einzelnen Vehicle würden in einem festen Intervall von n-Sekunden ihre Position senden, welche sich dann auf einer Karte anzeigen ließe. Je nach Frequenz des Aktualisierungsintervalls, wäre das Ergebnis der Position genauer oder ungenauer.

Der Einstaz von GPS hat aber allerlei Schwachstellen, die auf den ersten Blick vielleicht nicht offensichtlich sind.

Fehlende Datenverfügbarkeit: Ein Weg, um Echtzeitdaten zu visualisieren, wäre die Verarbeitung von GPS-Daten, die von den jeweiligen Verkehrsverbünden zur Verfügung gestellt werden müssten. Dies ist allerdings nicht der Fall. Zwar gibt es durchaus eine Erfassung der Öffentlichen Verkehrsmittel, allerdings werden diese nicht für Dritte zur Verfügung gestellt. Die HaCon GmbH sammelt beispielsweise solche Daten, indem sie diese durch in den Fahrzeugen integrierte Software berechnet.[5]. Es wären also GPS-Daten vorhanden, da sie unter anderem im Bus-Radar der Deutschen Bahn (entwickelt von HaCon) verwendet werden, sie sind allerdings weder über eine API noch anderweitig für die Öffentlichkeit erhältlich. Über die rechtlichen Belange und ob solche Daten in Deutschland überhaupt öffentlich gemacht werden dürften, soll an dieser Stelle nicht diskutiert werden¹¹.

Aktualisierungsintervall: Abseits der fehlenden Beschaffung von GPS-Daten haben diese noch einen weiteren Nachteil. Vehicle, die mit einer GPS-Lokalisierung ausgestattet sind, senden keinen kontinuierlichen Strom an Daten, sondern nur in einem gewissen Aktualisierungsintervall. Zwar preist HaCon seinen Busradar durch folgende Aussage an:

„Der neue Busradar eignet sich hervorragend, um die eigene Fahrt zu visualisieren und Anschlussfahrzeuge zu verfolgen. Erstmals geschieht dies GPS-basiert und nicht durch interpolierte Echtzeitdaten, was eine noch höhere Genauigkeit zur Folge hat.“[5]

Nähme man aber nur die GPS-Daten als Basis für eine Visualisierung, so würde der Bus bei jeder Aktualisierung von der jeweils vorherigen Position zur nächsten springen.

¹¹In „Opening Public Transit Data in Germany“ von Stefan Kaufmann[7] wird dieses Thema der Rechtslage näher betrachtet.

Dieses „Springen“ kann beim Busradar dann auch dazu führen, dass der Nutzer einen Bus auf seiner App verfolgen will, aber dieser nach dem nächsten GPS-Update nicht mehr auf dem Display zu sehen ist, da er nun außerhalb des Viewports liegt. Dieses Verhalten kann den Nutzer durchaus verwirren, da nicht klar ist, in welche Richtung sich das Vehicle bewegt hat, sodass in alle Richtung gesucht werden muss.

Verlässlichkeit & Verfügbarkeit: Zudem sind GPS-Signale nicht immer verlässlich. Sie können oftmals gestört werden oder die Verbindung zum Satelliten verlieren. Wie würde sich in einem solchen Fall des Signalverlusts die Live-Visualisierung verhalten? Verschwindet das Vehicle von der Karte oder bleibt es für längere Zeit auf der Stelle stehen? Beide Möglichkeiten erscheinen als nicht optimal.

Zuletzt sei erwähnt, dass ein GPS-basiertes System für U- und S-Bahn erst gar nicht infrage käme, da diese unterirdisch verlaufen und andere Technologien für deren Erfassung eingesetzt werden müssten. Für eine Live-Karte, die nicht nur Busse, sondern auch andere Verkehrsmittel abbilden möchte, ist die GPS-basierte Lokalisierung folglich nicht zielführend.

2.3.2 GTFS-Fahrplandaten

Das GTFS (General Transit Feed Specification) ist eine Datenstandardisierung, die von Google im Jahr 2006 entwickelt wurde. Vor dessen Einführung gab es weder eine einheitliche Standardisierung, noch ein „de facto Standard“ für die Fahrpläne des Öffentlichen Nahverkehrs in den USA. Unter anderem aus diesem Grund erfolgte die Adaption an das neue GTFS-Format sehr schnell. Vor allem für digitale Produkte, wie zum Beispiel Trip-Planer, die viele verschiedene Fahrpläne von unterschiedlichen Unternehmen in ihren Service integrieren müssen, ist ein standardisiertes Datenformat unbedingt notwendig. Heute sind die meisten öffentlich verfügbaren Fahrpläne im GTFS-Format auf Plattformen wie: <http://transitfeeds.com> oder <https://transit.land/> frei verfügbar.¹². GTFS ermöglichte es Transit Organisationen, ihre Daten für Dritte zu öffnen und ist heute das weit verbreitetste offene Datenformat für den Öffentlichen Nahverkehr¹³.[12, S. 2]

Ein GTFS-Feed besteht aus mindestens 6 und maximal 13 csv-Dateien, die im .txt Format vorliegen müssen. Die Struktur eines Feeds lässt sich in Worten wie folgt beschreiben:

Ein GTFS-Feed besteht aus einer oder mehreren Routen. Jede Route (`routes.txt`) hat einen oder mehrere Trips (`trips.txt`). Jeder Trip besucht eine Abfolge von Stops (`stops.txt`) zu einer bestimmten Zeit (`stop_times.txt`). Trips und Stop-Zeit beinhalten nur die Tageszeitinformationen. Der Kalender (`calendar.txt` und `calendar_dates.txt`) bestimmt dann, an welchen Tagen ein bestimmter Trip stattfindet. [16, S. 8]

¹²Momentan besitzt Transitfeeds 535 Feeds (Stand 18.08.2017)

¹³Dies gilt allerdings nicht für Deutschland, hier bieten zum jetzigen Zeitpunkt nur 3 Verkehrsunternehmen ein öffentliches GTFS-Feed an

Um sich den Inhalt einer GTFS-Datei besser vorstellen zu können, ist nachfolgend ein Auszug der `stops.txt` Tabelle abgebildet.

```

1      stop_id,stop_name,stop_lat,stop_lon
2      668,Moetzingen Bruehlstrasse,48.53249,8.775416
3      2840,Ludwigsburg Mainzer Allee,48.9018,9.21601
4      6409,Burgholzhof,48.81742,9.191285

```

Listing 1: Auszug der ersten Zeilen von `stops.txt`

Nachfolgend sind kurz die wichtigsten Tabellen beschrieben.

- `agency.txt` beinhaltet Informationen über die Verkehrsunternehmen, welche das Feed und die Daten bereitstellen.
- `routes.txt`: Eine Route ist eine Gruppierung von Trips. Die verschiedenen Eigenschaften einer Route werden in dieser Tabelle gespeichert.
- `trips.txt`: Ein Trip gehört zu einer Route. Eine Route kann dabei beliebig viele Trips haben. Welche Trips aktiv sind, wird durch den Kalender festgelegt.
- `calendar.txt` bestimmt, an welchen Tagen ein Trip aktiv ist.
- `stop_times.txt`: Diese Tabelle beschreibt, welche Stationen nacheinander angefahren werden. Für jede Station beinhaltet sie die Ankunfts- und Abfahrtszeiten.
- `stops.txt` stellt nähere Informationen für jede Station zur Verfügung wie zum Beispiel den Stationsnamen und deren Position.
- `shapes.txt`: Jeder Trip hat eine dazugehörige Polyline (Begriffsdefinition siehe 3.5.3)

Ein UML-Diagramm, welches die in Relation stehenden Dateien aufzeigt, existiert unter folgender Adresse: <https://developers.google.com/transit/gtfs/reference/>

Fehlende Echtzeitkomponente

Aber auch GTFS besitzt Nachteile. Für eine Live-Visualisierung fehlt ihr die Echtzeitkomponente. Die Fahrplandaten stellen nur einen **Soll-Zustand** dar, der erheblich vom **Ist-Zustand** abweichen kann. Die reale Position eines Vehicles kann somit nicht wiedergegeben werden, da die tatsächliche Verkehrslage nicht berücksichtigt wird. Dies wirkt sich besonders dann nachteilig aus, wenn Verkehrsmittel sich aufgrund von Staus oder sonstigen Verzögerungen verspäten, Verkehrsmittel ganz ausfallen oder aufgrund von Baustellen etc. umgeleitet werden müssen. Solche außerfahrplanmäßigen Vorkommnisse kann die Visualisierung ohne Echtzeitkomponente nicht bewältigen. Auch die Geschwindigkeit eines Vehicles entspricht bei einer Interpolation lediglich der Durchschnittsgeschwindigkeit, die sich anhand der Fahrplandaten ausrechnen lässt. Benötigt ein Vehicle V von Station A nach B 3 Minuten für eine Strecke von 1.2 Kilometer, so würde die Animation eine durchschnittliche Geschwindigkeit von $v = \frac{s}{t} = \frac{1.2 \cdot 1000}{3 \cdot 60} = 6.6 \frac{m}{s} = 23.76 \frac{km}{h}$ errechnen. Diese kann zwar in vielen Fällen wie bspw.

dem Schienen- oder Überlandsverkehr zutreffen, berücksichtigt aber im Stadtverkehr weder Geschwindigkeitsbegrenzungen noch Wartezeiten an Kreuzungen.

Uneinheitliche Gestaltung von GTFS

Trotz der Standardisierung durch GTFS gibt es immer noch diverse Freiräume in der Umsetzung des Formats. Wie anfangs erwähnt wurde, beträgt die Anzahl der Dateien, die für ein gültiges GTFS-Feed benötigt werden, nur 6. Es sind allerdings bis zu 13 Dateien möglich. Dies zeigt, wie viele unterschiedliche Informationen ein GTFS-Feed bereitstellen kann, aber nicht muss. Auch innerhalb der Dateien gibt es Felder, die vorhanden sein „müssen“ oder nur „dürfen“. Beispielsweise muss das Feld `route_short_name` in `routes.txt` vorhanden sein, aber `route_desc` (Route Description) nicht. Der Interpretationsspielraum lässt sich aber noch weiter veranschaulichen, wenn wir uns Tabelle 1 ansehen. In dieser Tabelle sind zwei Einträge aus unterschiedlichen GTFS-Feeds aufgelistet. Wir sehen, dass die Spalte `route_id` bei Stuttgart-VVS als Zahlenwert angegeben wird, wohingegen Boston-MBTA einen Text verwendet.

Tabelle 1: Unterschiede innerhalb GTFS

	route_id	route_short_name	route_long_name
Stuttgart-VVS	379	U1	Fellbach - Hauptbahnhof - Vaihingen
Boston-MBTA	Blue Line	Blue	Bowdoin - Wonderland

„Blue Line“ ist dabei die Bezeichnung der U-Bahnlinie[15]. Wir sehen also, dass Stuttgart-VVS die `route_id` zur eindeutigen Identifizierung mittels Zahlenwert verwendet, wohingegen Boston-MBTA dieses Feld nutzt, um den Namen der Linie zu beschreiben. Angenommen, wir verwenden die `route_id` in einer Benutzeroberfläche wie in Abbildung 6.



Abbildung 6: UI Element mit GTFS Informationen

Links in der Abbildung ist die korrekte Bezeichnung der Route zu sehen, nämlich `Blue Line`, wohingegen rechts nur eine numerische ID zu sehen ist, die nicht für den Nutzer vorgesehen und damit unbrauchbar ist. Damit die rechte Seite korrekt wäre, müsste dort `U1` abgebildet sein. Die fehlende, beziehungsweise nicht gegebene Übereinstimmung der beiden Feeds führt also zu Problemen bei der Darstellung, die auch durch die Verwendung eines anderen Feldes wie zum Beispiel `route_short_name` nicht behoben werden können.

Dies ist nur ein Beispiel, bei dem Abweichungen in der Ausführung der Spezifikation ei-

ne Auswirkung auf die Programmierung haben. Aus diesem Grund ist im März 2017 auf <http://gtfs.org/> eine neue **Best-Practices** Sektion erschienen. Dabei handelt es sich um Empfehlungen an die Hersteller von GTFS-Feeds, um eine Verwendung der Feeds möglichst zu vereinheitlichen. Würden sich alle Hersteller an solch eine striktere Implementierung der Spezifikation halten, müssten Programmierer weniger „Edge Cases“¹⁴ abfangen und Anwendungen würden in Qualität und Zuverlässigkeit noch besser werden.

Probleme beim Auslesen von Daten

Ein weiteres Problem in GTFS ist das Auslesen von Daten. GTFS lässt sich zwar sehr einfach in eine relationale Datenbank überführen, aber das Auslesen der Daten kann schnell sehr komplex werden, sodass die Geschwindigkeit für eine Webanwendung nicht mehr schnell genug ist. Bereits 1993 stellte **Jakob Nielsen** einen Richtwert für die responsive Wahrnehmung einer Webanwendung vor:

„1.0 second is about the limit for the user’s flow of thought to stay uninterrupted, even though the user will notice the delay. Normally, no special feedback is necessary during delays of more than 0.1 but less than 1.0 second, but the user does lose the feeling of operating directly on the data.“ [11]

Ob oder wie die Geschwindigkeit erreicht werden kann, soll hier nicht vertieft werden, denn diesem Thema habe ich bereits meine Bachelorarbeit gewidmet[8].

Damit eine Webanwendung aber überhaupt eine Chance hat, diese Geschwindigkeitsmarke zu erreichen, ist eine schnelle Antwortzeit des Backends sehr wichtig. Dabei sind Antwortzeiten innerhalb von 1 bis 200 Millisekunden ein sehr guter Wert. Natürlich gilt: Je weniger umso besser.

Das GTFS-Format hat den entscheidenden Nachteil, dass es eine hohe Komplexität aufweist, sobald Daten aus verschiedenen Tabellen benötigt werden. Für eine Live-Visualisierung, sind Daten aus nahezu allen Tabellen relevant. Abbildung 7 zeigt, welche davon benötigt - beziehungsweise nicht benötigt werden (grau).

Das UML-Diagramm ist auf den ersten Blick relativ simpel zu verstehen und die Grundlagen der verschiedenen Relationen wurde bereits in Kapitel 2.3.2 beschrieben. Wo liegt also das Problem? In Worten ließe sich diese Datenbankabfrage mit folgendem Statement beschreiben:

„Gib uns alle aktiven Trips mit deren Linienverlauf, die am heutigen Tag aktiv sind und in einer Zeitspanne zwischen t_a und t_b liegen.“

Das große Problem dieses Satzes liegt in der Zeitkomponente „Trips die am heutigen Tag aktiv sind zwischen ...“. Die Trip-Tabelle selbst (bezogen auf Abbildung 7) bietet diesbezüg-

¹⁴Ein „Edge Case“ ist ein Problem oder eine Situation, die nur bei einem extremen (maximalen oder minimalen) Betriebsparameter auftritt.

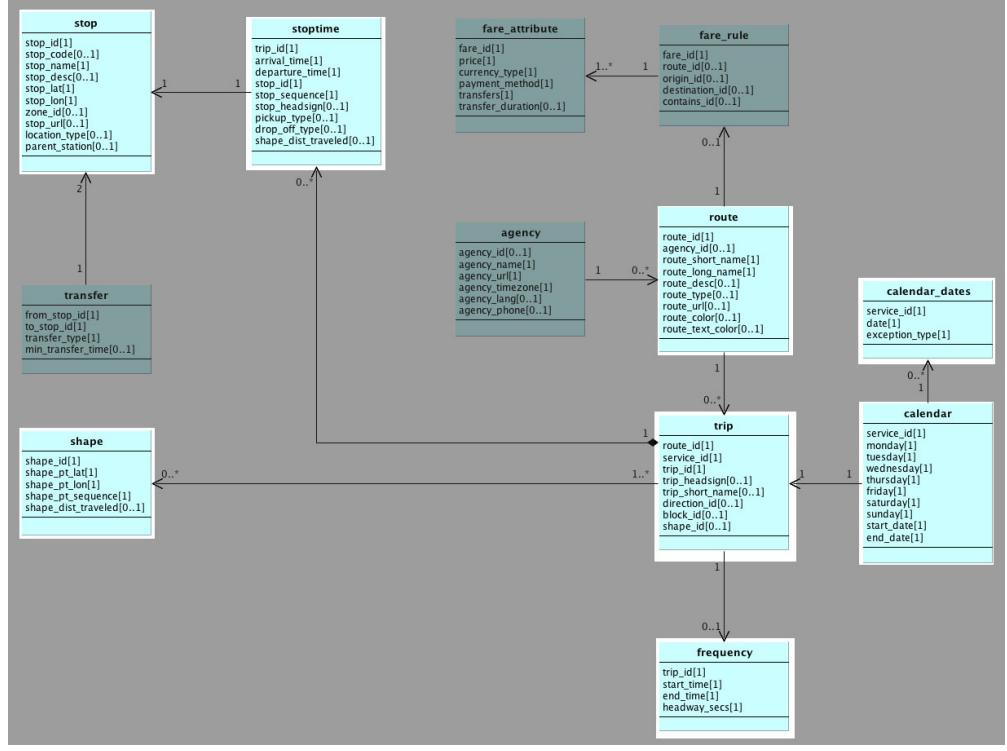


Abbildung 7: Benötigte GTFS Tabellen[6]

lich keinerlei Informationen. Auch die Calendar- und Calendar-Dates-Tabelle beinhaltet nur Informationen darüber, an welchem Datum ein Trip stattfindet, nicht aber um welche Uhrzeit. Erst die Stoptime-Tabelle ermöglicht es uns, eine Aussage zu treffen, wann ein Trip aktiv ist. Über die zwei Felder `arrival_time` und `departure_time` lässt sich sagen, zu welchem Zeitpunkt ein Vehicle an einer Station anhält. Die erste und letzte Station (S_1 und S_n) geben uns also einen zeitlichen Rahmen, in dem der Trip aktiv ist. Hierbei wird klar, dass allein die Beantwortung der Frage zur zeitlichen Komponente bereits sehr viele Daten aus verschiedenen Tabellen benötigt. Die anderen Tabellen wie `shape`, `route`, `stop` und `frequency` würden für weitere Informationen wie Vehicle-Farbe, Stop-Position (Längen- und Breitengrad) oder für die Polyline benötigt werden. Angesichts dieser komplexen Datenstruktur kann angenommen werden, dass einige technische Optimierungen notwendig wären, um die Zeit der Datenbank-abfragen zu minimieren.

2.3.3 GTFS-Realtime

Eine andere Möglichkeit für die Erfassung von Echtzeitpositionen und Verspätungen bietet GTFS-realtime. GTFS-realtime ist ein von Google entwickelter Standard, der Verkehrsunternehmen das Bereitstellen von Echtzeitinformationen ermöglicht. Dabei gibt es 3 verschiedene Feeds, die GTFS-realtime zur Verfügung stellt:[17][S. 6]

1. Vehicle positions
2. Trip updates

3. Service alerts

GTFS-realtime wäre für diese Arbeit deshalb interessant, da diese Spezifikation Updates zu Trips und Vehicle-Positionen ermöglicht. Beispielsweise kann die Interpolation anhand der Verspätung eines Vehicles angepasst werden. Ein Auszug eines Trip-Updates ist in Listing 2 zu sehen.

```
1  {
2    id: 25732950
3    trip_update {
4      trip {
5        trip_id: 25732950,
6        start_date: 20150120,
7      }
8      stop_time_update {
9        arrival {
10          delay: 240
11        }
12        stop_id: 135
13        ...
14      }
15      ...
16    }
17  }
```

Listing 2: Auszug eines GTFS-realtime Trip Updates von MBTA

Vehicle-Positionen können über ein ähnliches Format bezogen werden Listing 3.

```
1  {
2    id: "v121",
3    vehicle {
4      trip {
5        trip_id: 2590683,
6        start_date: 2017017
7      },
8      position {
9        latitude: 42.267967,
10       longitude: -71.093834
11     },
12     ...
13   }
14 }
```

Listing 3: Auszug eines GTFS-realtime Vehicle-Position Updates von MBTA

Für eine ausführlichere Beschreibung hilft das Buch: „*The Definitive Guide to GTFS-realtime - How to consume and produce real-time public transportation data with the GTFS-rt specification.*“[17] von Quentin Zervaas.

3 Define

Während in der **Discover-Phase** nur Ideen gesammelt und aufgelistet wurden, soll in diesem Kapitel die **Define-Phase** betrachtet werden. Dabei geht es darum, die getroffenen Entscheidungen zu begründen und zu konkretisieren als auch die begrifflichen Grundlagen dieser Arbeit zu definieren.

3.1 Kartenart

Aus der Discover-Phase sind verschiedene Ideen bezüglich der Darstellungsarten entstanden. Zum Beispiel sah eine mögliche Lösung vor, die interaktive Karte mit anderen Visualisierungsformen zu kombinieren. So könnten die einzelnen Trips auch als Balkendiagramm dargestellt werden, welches den Fortschritt der zurückzulegenden Strecke verbildlicht. Dabei war angedacht, dass zwischen diesen verschiedenen Visualisierungsformen hin- und hergeschalten werden kann. Auch der Ansatz, dies mit einer Tube-Map zu verbinden, wurde als Idee notiert und war zeitweise als mögliches Ziel definiert. Da sich für Tube-Maps allerdings keine Kartenanbieter fanden, konnte diese Kartenart nicht umgesetzt werden. Letztendlich wird angestrebt, eine Classic-Map zu entwerfen, bei welcher die optische Karte subtil gestaltet ist, sodass die animierten Vehicle im Vordergrund stehen. Darüber hinaus sollen dem Anwender einige Standardfunktionen wie Zooming oder Panning zur Interaktion bereitgestellt werden. Es wurde entschieden, bei der Entwicklung der Live-Karte zunächst keine spezifische Nutzergruppe anzuvisieren, sondern einen möglichst universellen Prototypen zu entwerfen, welcher vielseitig genutzt und in unterschiedliche Richtungen weiterentwickelt werden könnte.

3.2 Wahl der Datengrundlage

Im Abschnitt „Einsatz von GPS-Daten“ wurde bereits erklärt, wie der Einsatz von GPS-Daten eine Live-Visualisierung ermöglichen könnte. Die einhergehenden Probleme sind allerdings nicht unerheblich und das Fehlen zugänglicher Daten macht eine GPS-basierte Visualisierung zu diesem Zeitpunkt unmöglich.

Aus diesem Grund wird ein GTFS-Feed für die Visualisierung des Stuttgarter Nahverkehrs verwendet und dabei in Kauf genommen, dass eine Echtzeitdarstellung unter Berücksichtigung von Verspätungen, Ausfällen und Umfahrungen nicht möglich ist. Stattdessen wird die Visualisierung auf der Grundlage des Fahrplans und durchschnittlichen Geschwindigkeiten entwickelt.

Eine genauere Erfassung der Geschwindigkeit wäre zwar wünschenswert, bringt allerdings andere Schwierigkeiten mit sich. Die Erfassung der Geschwindigkeit von jedem Vehicle würde eine hohe Menge an Daten bedeuten, die zwischen Server und Client ausgetauscht werden müssen. Ähnlich wie bei einer GPS-basierten Animation, wäre der Client komplett davon abhängig, ständig Daten zu erhalten. Stelle man sich vor, dass mehrere hundert Anwender eine App benutzen, wäre dies eine enorme Menge an Anfragen & Antworten. Für Smartphones

mit schlechter Verbindung ist dieser Umstand ein großes Problem. Ebenso wie die verwendete Bandbreite und der erhöhte Batterieverbrauch durch das ständige Stellen von Anfragen und der Verarbeitung der Antwort.

Bei einer Interpolation des Fahrplans mit GTFS-Daten ist hingegen keine ständige Verbindung zum Server nötig. Existiert der relevante Teil des Fahrplans auf dem Gerät des Endnutzers, so kann die Animation anhand dieser Daten erfolgen. Zudem wird das Problem des „Springens“ umgangen, welches vor allem bei GPS-basierter Animation einen Nachteil darstellt. Durch die Interpolation sind glatte Animationen der Vehicle auf der Karte möglich, sodass die Bewegung von A nach B der realen Fahrbewegung eher entspricht. Dadurch kann der Anwender besser nachvollziehen, was geschieht. Eine Lösung für das Problem der fehlenden Echtzeiterfassung ließe sich über den Einsatz von GTFS-realtime erreichen. In dieser Arbeit kann GTFS-realtime allerdings nicht verwendet werden, da zum jetzigen Zeitpunkt¹⁵, der Verkehrsverbund Stuttgart-VVS dies nicht (auch nicht durch ein anderes Format) öffentlich anbietet.

3.3 Gewählte Technologien

Die auszuwählenden Technologien für ein solches Projekt sind zahlreich. Da es sich bei dieser Arbeit um keine Produktentwicklung mit eingeschränktem Nutzerkreis handelt, besteht bei der Technologieauswahl uneingeschränkte Freiheit. Um diesen Umstand auszunutzen und dem Projekt einen experimentellen Charakter zu verleihen, sollen vor allem zukunftsweisende Technologien Verwendung finden.

3.3.1 Datenbank

Da der GTFS-Standard eine fertige relationale Beziehung der einzelnen Dateien festlegt, ist der Einsatz einer relationalen Datenbank sehr naheliegend. Dabei gibt es eine breite Palette an Auswahl. Damit die Anwendung möglichst zugänglich bleibt, liegt der Fokus auf Datenbanken, die unter einer Open-source-Lizenz kostenfrei zur Verfügung stehen. Die zwei populärsten sind MySQL und PostgreSQL[4]. Beide haben ihre Vor- und Nachteile und die Entscheidung ist mehr eine persönliche Präferenz, als ein großer Vorteil des Einen über den Anderen. Einen kleinen Vorteil bietet PostgreSQL's Unterstützung für Array-Types, welche sehr hilfreich beim Speichern und Abfragen von Daten ist. So fiel die Entscheidung auf die PostgreSQL Datenbank, wobei eine Realisierung auch mit MySQL möglich gewesen wäre.

3.3.2 Serverwahl

Für das Backend soll `Nodejs` verwendet werden. Nodejs ist nicht nur einfach aufzusetzen, sondern auch sehr performant und effizient für Web Applikationen einsetzbar. Zudem lässt es sich sehr einfach mittels Docker in der AWS (Amazon Web Services) Cloud veröffentlichen. Da Nodejs dynamisch typisiert, lassen sich vor allem auch Prototypen sehr schnell erstellen.

¹⁵September, 2017

Zusätzlich können sowohl Server und Client in JavaScript programmiert werden, wodurch die meisten Frameworks sowohl für den Server als auch für den Client zur Verfügung stehen.

3.3.3 Kartenmaterial

Bereits zu Beginn wurde von einer hohen zu bewältigenden Datenmenge ausgegangen. Um die Voraussetzungen dafür zu schaffen, wurde nach Softwarelösungen gesucht, die für solche Datenmengen ausgelegt sind. Für die Karte wird dafür Mapbox eingesetzt. Mapbox verwendet Web-GL (basierend auf OpenGL) und bietet damit die Möglichkeit, ein GPU-unterstütztes Rendering im Browser zu ermöglichen. Zusätzlich bietet Mapbox gegenüber Google-Maps den Vorteil von eigenen Karten-Styles. Diese können über Mapbox-Studio voll umfänglich auf die eigenen Bedürfnisse angepasst werden. Parks, Straßen, Schriftzüge, nahezu alle Elemente der Karte, lassen sich ändern und anpassen. Zusätzlich können eigene Daten in die Karte integriert werden, was Bandbreite und Rechenleistung spart. Damit können sämtliche Routen, die das Stuttgart-VVS Feed beinhaltet, in das Kartenmaterial gezeichnet werden (siehe orangene Linien in Abbildung 8).

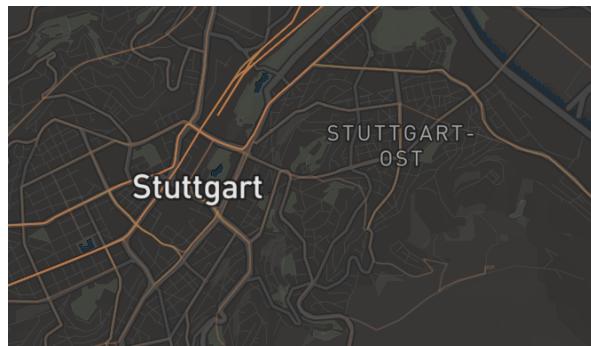


Abbildung 8: Karte mit integrierten Stuttgart-VVS Daten

Die Wahl der richtigen Tools ist dabei nur die Grundlage, um die Datenmenge zu bewältigen. Viele weitere Schritte sind notwendig, um eine performante Webanwendung zu erstellen und werden im nächsten Kapitel Develop noch ausführlicher aufgeführt.

3.3.4 Frameworks

Für die Programmierung wurden folgende Bibliotheken ausgewählt. Diese bieten verschiedene Erleichterungen bei der Programmierung.

Turf¹⁶ stellt eine ganze Reihe an Funktionen für die raumbezogene Verarbeitung von Daten zur Verfügung. Beispielsweise lassen sich mittels Turf unter anderem Distanzen, Flächen oder Schnittpunkte berechnen.

¹⁶<http://turfjs.org/docs/>

Mapbox-gl-js¹⁷ wird benötigt, um das Kartenmaterial von Mapbox im Client zu verwenden. Dafür stellt es eine eigene API zur Verfügung.

Lodash¹⁸ ist eine Hilfsbibliothek, die verschiedene Funktionen zur Verfügung stellt, die das Arbeiten mit JavaScript vereinfachen.

Express¹⁹ ist ein minimalistisches Node.js-Framework für moderne Web-Applikationen. Es vereinfacht die Erstellung von API-Endpunkten durch das Bereitstellen hilfreicher Methoden zur Erstellung des Routing. Routing bezieht sich dabei auf die Bestimmung, wie eine Anwendung auf eine Client-Anfrage an einen bestimmten Endpunkt reagiert, also auf eine URI (oder einen Pfad) und eine bestimmte HTTP-Request-Methode (GET, POST usw.).

3.4 Zielsetzung

Das Hauptziel der Arbeit besteht darin, eine interaktive Karte zu entwickeln, die den Öffentlichen Nahverkehr des Verkehrsverbunds Stuttgart-VVS auf einer Live-Karte visualisiert. Dafür soll ein Prototyp entwickelt werden, der für Demonstrationszwecke eingesetzt werden kann. Weitere visuelle Ziele sollen sich im Prozess durch das Erkunden verschiedener Lösungsansätze ergeben.

Außerdem werden Ziele in Bezug auf die Performance der einzelnen Komponenten (Datenbank, Server, Client) gesetzt. Die Datenbank soll ein GTFS-Feed der Stuttgart-VVS aufnehmen und dessen Daten in 0 bis 200ms bereitstellen können. Des Weiteren soll der Nodejs-Server die Daten innerhalb von maximal 100ms verarbeitet haben. Das Frontend soll die Vehicle mit 60 FPS rendern können.

3.5 Begriffe und Definitionen

Bevor im nächsten Kapitel „Develop“ die Umsetzung des Projekts beschrieben wird, sollen zur Sicherstellung eines gemeinsamen Verständnisses zuerst die verwendeten Begrifflichkeiten geklärt bzw. definiert werden.

3.5.1 Time

In verschiedenen Formeln wird immer wieder eine Time t_{cur} referenziert. Diese beschreibt die globale (aktuelle) Zeit des Systems in Sekunden. Die Sekunden lassen sich durch das Addieren der Stunden, Minuten und Sekunden errechnen.

¹⁷<https://www.mapbox.com/mapbox-gl-js/api/>

¹⁸<https://lodash.com/>

¹⁹<https://expressjs.com/en/starter/basic-routing.html>

Bsp: 17:04:59 Uhr

$$t_{cur} = \text{Stunden} * 3600 + \text{Minuten} * 60 + \text{Sekunden}$$
$$\Rightarrow t_{cur} = 17 * 3600 + 4 * 60 + 59 = 61499 \text{ sec}$$

3.5.2 Vehicle

Ein Vehicle V beschreibt in dieser Arbeit ein Fahrzeug, welches im Dienste der öffentlichen Verkehrsbeförderung steht. Dies sind beispielsweise Bus, U- & S-Bahn, Interrail Züge aber auch Zahnradbahn oder gar Funicular-Services²⁰. Das private Automobil fällt folglich nicht unter diese Definition.

3.5.3 Polyline

Eine Polyline²¹ P ist eine Kurve, die sich durch eine Sequenz an Punkten $\{p_1, \dots, p_n \mid n \in \mathbb{N}\}$ definiert. Sie beschreibt den zurückzulegenden Verlauf eines Vehicles.

3.5.4 Station

Eine Station S ist eine Haltestelle, die von einem Vehicle V während eines Trips T angefahren wird und sich entlang einer Polyline befindet. Die Station definiert dabei die Ankunfts- und Abfahrtszeiten, wann ein Vehicle an dieser Station anhält und wann es diese zur Weiterfahrt wieder verlässt. Ankunfts- und Abfahrtszeit seien wie folgt definiert: `arrival time := t_{ari}` und `departure time := t_{dep}` .

3.5.5 Trip

In dieser Arbeit wird immer wieder der Begriff „Trip“ Verwendung finden. Ein Trip T sei mit folgenden Eigenschaften definiert:

- T besteht aus einer Anzahl an Stationen: $T = \{S_1, \dots, S_n \mid n \in \mathbb{N}, n \geq 2\}$
- Ein Trip T wird dabei von genau einem Vehicle V bedient. Daraus folgt T mit dem Mapping: $T \mapsto V$ ist Injektiv zu V .
- Ein Trip T besitzt genau eine Polyline P . $T \mapsto P \Rightarrow T$ ist injektiv zu P .
- Die Bewältigung der Strecke $\{A, B \mid A, B \in S\}$ entlang einer Polyline P durch ein Vehicle V gilt als ein einziger Trip.
- Ein Trip beginnt genau dann, wenn die momentane Zeit t_{cur} mit der Abfahrtszeit t_{dep} der ersten Station übereinstimmt $\Rightarrow t_{cur} = t_{dep}$.
- Ein Trip endet genau dann, wenn die momentane Zeit t_{cur} mit der Ankunftszeit t_{ari} der letzten Station übereinstimmt $\Rightarrow t_{cur} = t_{ari}$.

²⁰<https://en.wikipedia.org/wiki/Funicular>

²¹Linienverlauf bzw. auch Shape genannt

- Der Rückweg $\{B, A \ni T \mid A, B \in S\}$ ist nicht in einem Trip T enthalten, sondern wird als ein neuer Trip erfasst.

3.5.6 Route

Eine Route R besteht aus einer Anzahl an Trips $T \geq 1$. Eine Route vereint alle vorherigen Relationen in sich. Abbildung 9 veranschaulicht diese.

- $R = \{T_1, \dots, T_n \mid n \in \mathbb{N}, n \geq 1\}$
- R mit dem Mapping: $R \mapsto T$ ist surjektiv²²

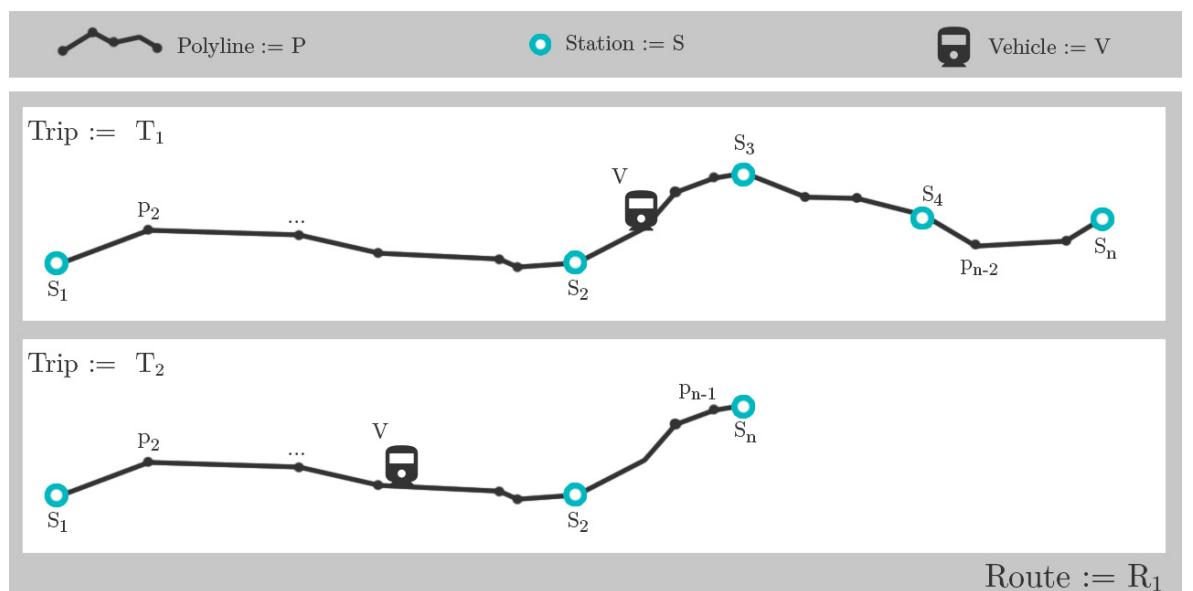


Abbildung 9: Grafische Veranschaulichung einer Route

Abbildung 9 zeigt, dass eine Route zum Beispiel alle Trips einer U-Bahn-Linie erfasst. Eine U-Bahn-Linie muss dabei nicht immer an allen Stationen halten, sondern kann beispielsweise im Nachtbetrieb auch Stationen auslassen (zu sehen bei Trip T_2). Trotzdem werden nur diejenigen Trips in einer Route vereint, die dem selben Routenverlauf folgen.

²²Eine Route kann mehrere Trips besitzen, wohingegen ein Trip nur einer Route zugehörig sein kann.

4 Develop

In diesem Kapitel soll der Entwicklungsprozess zum besseren Verständnis einigermaßen linear beschrieben werden, obwohl der eigentliche Ablauf nicht immer dieser Linearität entsprach. Zuerst soll erläutert werden, welches Prinzip hinter der Animation einzelner Vehicle entlang einer Polyline steht. Anschließend wird dieser Ansatz weiterentwickelt, bis die Animation von allen aktiven Trips auf einer Karte möglich ist. Dabei wurden verschiedene Optimierungsmaßnahmen ergriffen, um Probleme bezüglich der Performance zu beseitigen. Vor allem die zu verarbeitende Datenmenge, als auch das Verarbeiten und Optimieren von GTFS-Feeds, sind die Kernpunkte in diesem Kapitel.

4.1 Anzeigen einer Polyline

Zu Beginn stellte sich die Frage, wie sich in kleinen Schritten an das komplexe Thema einer Live-Visualisierung herangetastet werden kann. Die erste Hürde ist die Animation von nur **einem** Vehicle entlang einer Polyline. Die Umsetzung dieses ersten Schrittes soll in diesem und den nächsten zwei Abschnitten 4.2 und 4.3 erklärt werden.

Um eine Datengrundlage zu haben, wurde ein möglichst vollständiges GTFS-Feed ausgewählt²³ und in die Datenbank importiert. Die Wahl fiel dabei auf das Boston-MBTA Feed. Die Herausforderung bestand nun darin, erste Daten aus der Datenbank an den Client zu senden und sie dort darzustellen. Fast trivial ist das Abfragen der Polyline:

```
SELECT * FROM gtfs_shapes WHERE shape_id = 12345
```

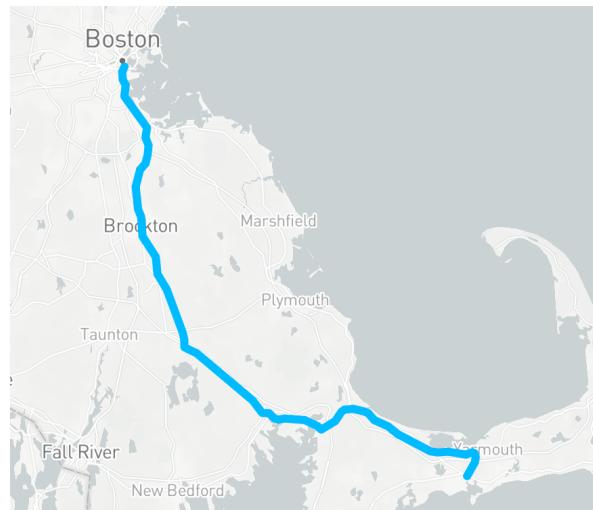


Abbildung 10: Anzeigen einer einzigen Polyline in Boston

Die Daten der Polyline werden als GeoJSON übertragen und lassen sich mittels Mapbox-gl.js auf der Karte anzeigen. Abbildung 10 zeigt das Ergebnis dieser ersten Iteration. Zu sehen ist bereits die Karte mit der Polyline (hier in blau). Die abgefragten Daten können nun also

²³<http://TransitFeeds.com>

bereits verarbeitet und angezeigt werden. Damit ein Vehicle entlang dieser Linie animiert werden kann, werden die einzelnen Stationen des Trips benötigt. Dieser ständige Wechsel zwischen der Arbeit am Backend, um neue Datenabfragen zu ermöglichen und dem Frontend, um diese anschließend anzuzeigen, stellte sich als sehr effektiv heraus und zog sich durch das gesamte Projekt hinweg durch.

4.2 Hinzufügen der Stationen

Nachdem die Polyline auf die Karte gebracht wurde, werden die Stationen des Trips benötigt (Abbildung 11). Diese beinhalten für die Animation essentielle Daten wie zum Beispiel Abfahrts- und Ankunftszeit eines Vehicles. Die genaue SQL-Abfrage dafür soll an dieser Stelle der Einfachheit wegen ausgespart bleiben.

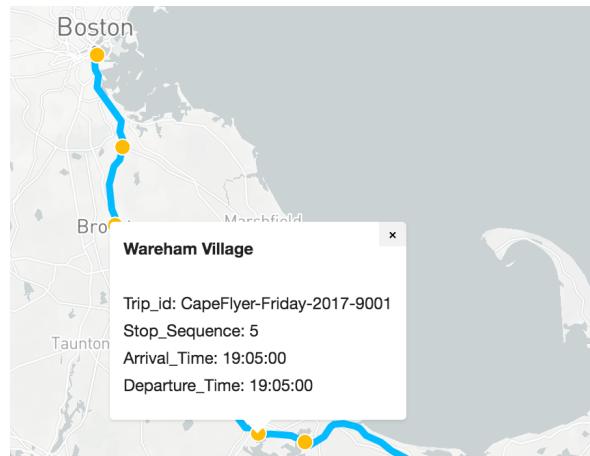


Abbildung 11: Hinzufügen von Stationen entlang der Polyline

Für die Animation der Vehicle (Abschnitt 4.3) wird außerdem der Wert für die Differenz der `distance_traveled`, also die Distanz zwischen den einzelnen Stationen, benötigt. Diese kann aus den `distance_traveled`-Werten der einzelnen Stationen über Subtraktion berechnet werden. Nun hat zwar das Stuttgart-VVS-Feed ein `distance_traveled`-Feld, welches die benötigten Distanz-Informationen direkt aus der Datenbank liefert; für andere GTFS-Feeds, die getestet wurden, ist dieses Feld jedoch oftmals nicht vorhanden. Aus diesem Grund wurde ein Algorithmus entwickelt, welcher über ein sogenanntes **Station-Matching** die Werte der `distance_traveled` und damit die Distanz zwischen den Stationen ermitteln kann. Der Station-Matching-Algorithmus wird von der Applikation nur dann als Fallback-Lösung für die Berechnung verwendet, wenn das `distance_traveled`-Feld nicht im Feed vorhanden ist.

Station-Matching

Das Station-Matching soll folgendes Problem lösen:

Wie in Abbildung 12 zu sehen ist, sind die Stationen (Haltestellen oder Bahnhöfe) nicht exakt auf der Polyline, sondern ein wenig abseits positioniert, da dies auch ihrem realen Standort



Abbildung 12: Stationen liegen nicht direkt auf der Polyline

neben der Fahrbahn entspricht. Um nun die Distanzen zwischen den Stationen zu berechnen, legt der Algorithmus im Sinne des Matchings zunächst die Stationen auf die Polyline.

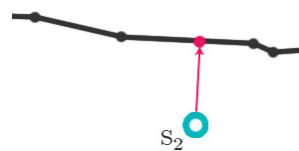


Abbildung 13: Finde den nächstgelegenen Punkt der Station auf der Polyline

Nachdem der entsprechende Punkt auf der Polyline gefunden wurde, berechnet der Algorithmus die jeweiligen Wegstrecken zur ersten Station des Trips (distance_traveled), bevor er die Werte von aufeinanderfolgenden Stationen subtrahiert. Die Distanz zweier Stationen $\{S_i, S_{i+1} \mid i \in \mathbb{N}\}$ sei d_Δ . Diese kann jetzt wie folgt berechnet werden: $d_{\Delta_i} = d_{S_i} - d_{S_{i-1}} \mid d_S := DistanceTraveled, n \in \mathbb{N}, n \geq 2$ In Listing 13 des Anhangs wird dieser Station-Matching-Algorithmus vorgestellt.



Abbildung 14: Berechnen der Distanz

Abbildung 15 stellt eine frühere Implementierung mit der nun aktuellen Version des Algorithmus gegenüber, indem die für das Matching über n -Trips benötigte Zeit der beiden Algorithmen verglichen wird. Um eine durchschnittliche Laufzeit zu erhalten, wurde jeder Algorithmus 10 mal mit der gleichen Anzahl an Trips ausgeführt.

Der alte Algorithmus war sehr simpel und beruhte darauf, die Funktion `pointOnLine` der `Turf.js`-Bibliothek zu verwenden. Diese Funktion hatte den entscheidenden Nachteil, dass sie in 3 `for`-Schleifen über die gesamten Punkte der Polyline iteriert. Hinzu kommt, dass das Matching nicht nur auf eine Station, sondern auf sämtliche Stationen aus allen Trips angewendet werden muss. Das führte dazu, dass insgesamt 5 `for`-Schleifen verwendet wur-

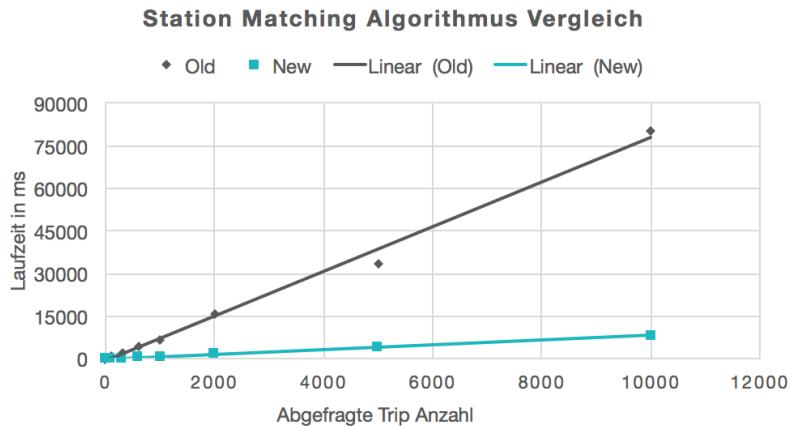


Abbildung 15: Vergleich der zwei Station-Matching-Algorithmen

den. Damit lässt sich der im Vergleich höhere Anstieg der Laufzeit bei steigender Trip-Anzahl erklären. Generell ist der neue Algorithmus sehr viel schneller. Durch die Verwendung eines R-Trees²⁴ und der eigenen Implementierung verschiedener Bibliotheksfunktionen (`turf.lineSlice`, `turf.pointOnLine`), konnte die Laufzeit drastisch reduziert werden (siehe Tabelle 2).

Tabelle 2: Station-Matching Vergleich Old / New

Anz. verarbeiteter Trips	Old (in ms)	New (in ms)
100	712	121
300	2191	305
600	4344	545
1.000	6780	874
2.000	15782	1700
5.000	33708	4161
10.000	80291	8279

Zwar wachsen beide Implementierungen lediglich linear mit steigender Trip-Anzahl, allerdings benötigt der neue Algorithmus für die Verarbeitung von 10.000 Trips anstatt 80.29 nur 8.28 Sekunden.

Im Realbetrieb verarbeitet der Server zwischen 0 – 500 Trips. Bei dieser Anzahl beträgt die Laufzeit des Algorithmus $\approx 80ms - 400ms$. Dadurch kann argumentiert werden, dass der Algorithmus gerade noch schnell genug für eine Webanwendung arbeitet. Auch größere Anzahlen an Trips wären noch in akzeptabler Geschwindigkeit berechenbar. So können 1000 Trips immer noch in unter einer Sekunde berechnet werden. Allerdings wäre es bei einer größeren Anzahl an Trips ein falscher Ansatz, diese bei jeder Serveranfrage neu zu kalkulieren.

²⁴Ein R-Tree (R für Rectangle) bezeichnet eine baumförmige Datenstruktur für das Speichern und Abfragen von raumbezogenen Informationen. In Verwendung ist folgende Bibliothek: <https://github.com/mourner/rbush>

Besser wäre es, einmalig das Matching für alle Trips eines GTFS-Feeds durchzuführen und die Ergebnisse persistent in der Datenbank abzuspeichern. Dies könnte beispielsweise gleich beim Importieren der Daten in die Datenbank geschehen. Dadurch könnte die Berechnung komplett eingespart werden. Da für das Stuttgart-VVS-Feed glücklicherweise die zurückgelegte Distanz bis zu einer Station bereits zur Verfügung steht, muss nur noch die Distanz zwischen den Stationen (d_{Δ}) berechnet werden. Dies geschieht nach dem selben Prinzip wie in der oben genannten Subtraktions-Formel. Diese Berechnung ist trivial und erfolgt bei 10.000 Trips in unter 15 Millisekunden.

4.3 Animieren eines Vehicles durch Interpolation

Der zentrale Kerngedanke für die Animation der Vehicle-Bewegung ist die Interpolation von Distanzen zwischen zwei aufeinanderfolgenden Stationen A und B mit der Distanz d_{Δ} . Um zu verstehen, wie ein Vehicle zwischen den einzelnen Stationen interpoliert werden kann, soll Abbildung 16 helfen.

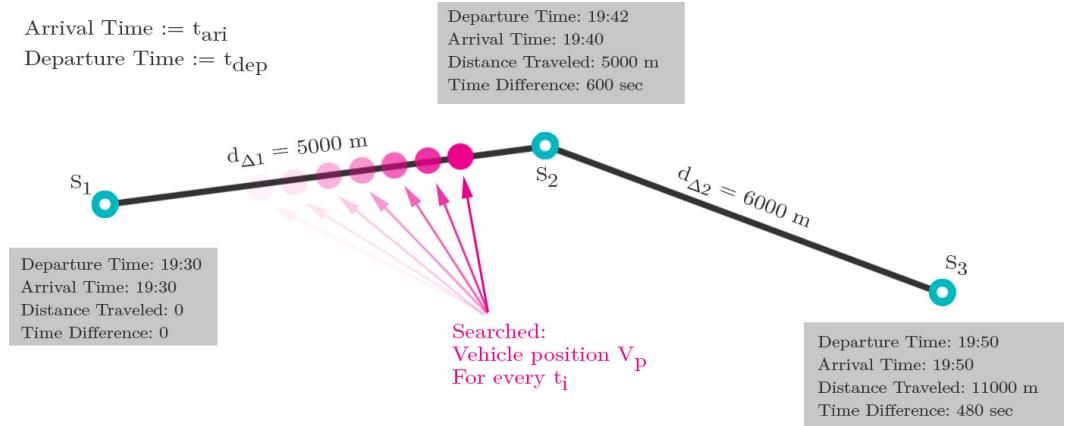


Abbildung 16: Interpolation der Vehicle Position: V_p

In den grauen Kästen der Grafik sind die Daten der einzelnen Stationen zu sehen. Diese kommen direkt aus der Datenbank oder werden vom Server vorberechnet und sind für die Berechnungen der Interpolation zwingend notwendig.

Arrival Time: Ankunftszeit t_{ari} des Vehicles an der Station laut Fahrplan

Departure Time: Abfahrtszeit t_{dep} des Vehicles von der Station laut Fahrplan

Distance Traveled: Bis zu dieser Station zurückzulegende Gesamtdistanz d_S

Difference Distance Traveled: d_{Δ} ist die zurückzulegende Distanz zwischen 2 Stationen A und B²⁵

Time Difference: Zeitdifferenz zwischen Ankunftszeit einer Station und der Abfahrtszeit der vorherigen Station $TimeDifference = t_{aris_n} - t_{deps_{n-1}}$

Gesucht ist nun eine Vehicle Position V_p zwischen den einzelnen Stationen für jede Zeiteinheit t_i . Dazu werden folgende Parameter benötigt:

- Aus der Datenbank wird die Polyline eines Trips benötigt. Auf dieser soll sich das Vehicle fortbewegen. Dieser Schritt wurde in Abschnitt 4.1 bereits erläutert.
- Außerdem werden alle Stationen, die zu diesem Trip gehören, benötigt. Auch der Abruf dieser Informationen wurde im vorherigen Abschnitt bereits behandelt.
- Um die Vehicle-Position ermitteln zu können, wird die Vehicle-Geschwindigkeit v benötigt. Um diese zu berechnen, wird die Formel $v = \frac{d_{\Delta}}{\text{TimeDifference}}$ für gleichförmige Bewegungen verwendet. Sowohl die Zeitdifferenz als auch d_{Δ} lässt sich aus dem GTFS-Feed auf dem Server berechnen. Wie dies geschieht, wird später in Kapitel „4.2 Station-Matching“ beschrieben.
- Mithilfe der berechneten Geschwindigkeit kann eine interpolierte Distanz s_{neu} des Vehicles zu einem bestimmten Zeitpunkt berechnet werden, damit das Vehicle zwischen Station A und B bewegt werden kann. Dazu wird die Formel der gleichförmigen Bewegung $s_{neu} = v*t_i + s_0$ benötigt. s_{neu} ist die interpolierte Distanz zwischen zwei Stationen A und B. v ist die zuvor berechnete Vehicle Geschwindigkeit. s_0 ist die Anfangsdistanz und damit die **Distance Traveled** der vorherigen Station A. t_i stellt eine Zeitdifferenz in Sekunden dar. Die Genauigkeit beträgt dabei Millisekunden, also beispielsweise 1.522sec. Diese wird errechnet, indem die **Departure Time** t_{dep} der Station A von der momentanen Systemzeit der Webanwendung t_{cur} subtrahiert wird. Dadurch lässt sich feststellen, wann ein Trip aktiv oder inaktiv ist.

Für $t_i < 0$ ergeben sich folgende Fälle:

$$t_{cur} < t_{ari_{S_1}} \Rightarrow \text{Trip hat noch nicht begonnen und ist inaktiv}$$

$$t_{ari_{S_i}} < t_{cur} < t_{dep_{S_i}} \Rightarrow \text{Trip ist aktiv, aber Vehicle wartet an der Station auf weiterfahrt.}$$

$$t_{cur} > t_{dep_{S_n}} \Rightarrow \text{Trip ist beendet}$$

Für $t_i > 0 \Rightarrow$ der Trip ist aktiv und das Vehicle befindet sich zwischen zwei Stationen A und B.

Sind all diese Parameter vorhanden, lässt sich die Distanz des Vehicles zwischen den einzelnen Stationen zu jedem Zeitpunkt t_{cur} interpolieren. Dafür kann die Bibliotheksfunktion `turf.along(polyline, sneu)` verwendet werden, die eine Polyline und eine bestimmte Entfernung zum Startpunkt dieser Polyline nimmt und einen Punkt in dieser Distanz zurückgibt. Aus einer Distanz lässt sich also ein Punkt mit Längen und Breitengrad ausrechnen, der anschließend auf der Karte angezeigt werden kann. Erfolgt diese Berechnung eines neuen Punktes pro Sekunde 60 mal (was genau 60 FPS entspricht), so lässt sich durch das Verschieben dieses Punktes eine Animation des Vehicles erreichen.

²⁵Als A und B seien immer zwei direkt aufeinander folgende Stationen S_{n-1}, S_n bezeichnet.

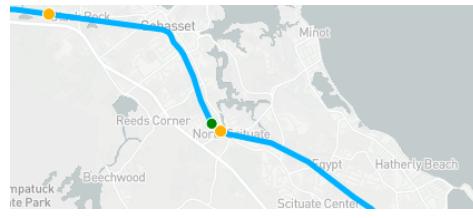


Abbildung 17: Interpolation eines Vehicles entlang seiner Polylines

Das Ergebnis lässt sich in Abbildung 17 betrachten. Das Vehicle wird als grüner und die Stationen als orangene Kreise dargestellt.

4.4 Zeichnen aller Polylines

Nachdem auf der Karte nun ein einzelner Trip angezeigt und animiert werden kann, sollte nun versucht werden, alle Polylines der Trips auf der Karte anzuzeigen. Abbildung 18 zeigt, wie dies für das Boston MBTA-Feed aussieht.



Abbildung 18: Darstellung aller Polylines auf der Karte

Zwar konnten alle Polylines auf der Karte angezeigt werden, allerdings bei sehr langer Rechen- und Ladezeit. Durch die Erhöhung der in der Datenbank abzufragenden Datenmenge, wurden neue Probleme offen gelegt. Die ursprüngliche Datenrepräsentation der Polylines in der Datenbank war nicht optimal und die Daten ließen sich nicht performant genug abfragen. Im Folgenden sollen die Optimierungsmaßnahmen, welche an der Polyline und der Datenbank zur Steigerung der Performanz unternommen wurden, aufgezeigt werden.

4.4.1 Ramer–Douglas–Peucker

Das Problem: Die im Stuttgart-VVS-Feed zur Verfügung gestellten Polylines sind überdefiniert und können aus tausenden Punkten bestehen. Für eine Visualisierung ist eine solche

Genauigkeit nicht notwendig und aufgrund der großen Datenmenge problematisch. Durch die Verwendung des „Ramer–Douglas–Peucker“ (RDP)-Algorithmus kann die Anzahl der Punkte einer Polyline drastisch reduziert werden. Der Vorteil besteht darin, dass dabei nicht der Linienerlauf verändert wird. Abbildung 19 zeigt ein Beispiel einer solchen Vereinfachung mittels einer JavaScript Bibliothek²⁶.

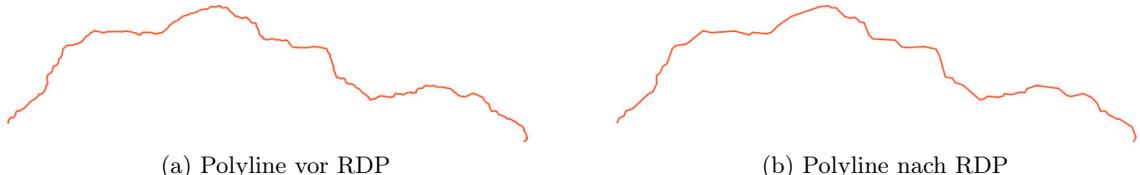


Abbildung 19: Vereinfachung einer Polyline mittels Simplify.js

Ausgangspunkt ist eine Polyline mit ≈ 1000 Punkten (19a). Nach der Vereinfachung (19b) ist die Anzahl auf 100 Punkte reduziert, ohne dabei visuell merklich einzubüßen. Dies ist eine erhebliche Reduzierung der Punkte um 90%. Wie wirkt sich dieser Algorithmus positiv auf das Projekt aus? Die Vorteile sind weitreichend. Sehen wir uns die Shape-Tabelle in Abbildung 20 an. 20a zeigt 394 Reihen vor der Optimierung und nur noch 140 (20b) nach Anwendung des RDP Algorithmus.

agency_key	shape_id	shape_pt_lat	shape_pt_lon	shape_pt_sequence		agency_key	shape_id	shape_pt_lat	shape_pt_lon	shape_pt_sequence	
stuttgart-vvs	3201	48.64096	8.908688	1	⋮	stuttgart-vvs	3201	48.997425	9.137341	1	⋮
stuttgart-vvs	3201	48.641	8.90877	2		stuttgart-vvs	3201	48.997227	9.137504	2	
stuttgart-vvs	3201	48.66781	8.685253	394		stuttgart-vvs	3201	49.041973	9.090726	140	

(a) Shape-Tabelle vor RDP
(b) Shape-Tabelle nach RDP

Abbildung 20: Reduzieren der Anzahl an Tabellen-Einträge via RDP

In seinem Originalzustand hat das verwendete VVS-Feed 1,085,859 Mio. Zeilen. Nach der Anwendung sind diese auf 617,653 Tsd. verringert. Testet man folgende PostgreSQL Abfrage, `SELECT * FROM gtfs_shapes WHERE shape_id = 3201` die alle Punkte einer Polyline ausgeben soll, so ergibt sich für ein optimiertes Feed eine Query-Zeit von $\approx 145ms$ und für das nicht optimierte Feed $\approx 250ms$. Schon durch diese einfache Methode sind bereits erste Performance-Steigerungen wahrnehmbar.

Der RDP-Algorithmus wurde auf das GTFS-Feed angewendet, noch bevor die Daten der Polyline in die Datenbank importiert wurden. Dadurch muss die Polyline nicht während der Laufzeit vereinfacht werden, sodass diese Rechenzeit eingespart werden kann. In Kapitel 4.5.1 wird ein Tool namens `gtfstidy` vorgestellt, das GTFS-Feeds optimieren kann. Dabei ist auch das Vereinfachen von Polylines mittels RDP möglich.

²⁶Simplify.js <http://mourner.github.io/simplify-js/>

4.4.2 Aggregieren der Shape-Tabelle

In GTFS wird für jeden Punkt einer Polyline eine Reihe in der Datenbank belegt. Diese Abfolge ist durch eine sogenannte **Shape Point Sequence** festgelegt, was nichts anderes ist, als eine Zahl von 1 bis n . Dies ist auch bereits in obiger Tabelle 20 zu sehen gewesen. Sehr viel effektiver wäre es allerdings, diese Punkte nicht Reihenweise, sondern alle zusammengehörenden Punkte in nur einem einzigen Feld zu speichern. Dies ist in PostgreSQL durch eine Aggregation möglich. Daraus ergibt sich folgende Shape-Tabelle:

agency_key	shape_id	shape_coords
stuttgart-vvs	3201	[[9.1373409999999927,48.997424999999998],[9.1375039999999985,48.997227000000023],[9.1378179999999933,48.9...

Abbildung 21: Aggregierte Koordinaten der Shape-Tabelle

Wie zu sehen ist, benötigt nun eine Polyline in der Shape-Tabelle nicht mehr 140 Reihen, sondern nur noch eine einzige. Für diese Arbeit ist dies auf alle Polylinien angewendet worden und in einer neuen Tabelle namens `denormalized_shapes` abgespeichert. Dadurch ist die Berechnung der Aggregation nur einmal nötig. Der SQL-Befehl dafür ist dem Anhang unter 7 zu entnehmen. Wir wenden die selbe SQL-Abfrage, die bereits oben Verwendung fand, auf die neue `denormalized_shapes` Tabelle an. Die Query-Zeit ist auf $\approx 1ms$ gesunken. Anstatt hunderte Reihen, muss nur eine einzige Reihe ausgelesen werden, was sehr effizient ist. Durch das Denormalisieren²⁷ der Shape-Tabelle ist auch die Anzahl der Reihen in der Datenbank auf ein Minimum gesunken. Von den früheren 617,653 Tsd. Reihen sind durch die Aggregation nur noch 4,524 Tsd. übrig.

4.4.3 Polyline Encoding

Die letzte Maßnahme zur Optimierung der Polyline stellt das sogenannte Polyline-Encoding dar. Wie dieses Verfahren genau funktioniert, geht an dieser Stelle zu weit. Hier soll nur erklärt werden, was unter Polyline-Encoding verstanden wird und warum es hier angewandt wird.

Das Polyline-Encoding kann in JavaScript beispielsweise durch das Google-Polyline²⁸ Paket eingesetzt werden. Das Encoding wandelt eine Polyline, bestehend aus Punkten, in einen String um. Zum Beispiel die Punkte: (38.5, -120.2), (40.7, -120.95), (43.252, -126.453) werden als `_p~iF~ps|U_ullnnqC_mqNvxq`@` codiert. Dies geschieht in meiner Anwendung immer bevor Daten vom Server in Richtung Client geschickt werden: Encode → Send → Decode. Da eine codierte Polyline weniger Zeichen benötigt, kann damit Datenvolumen bei der Kommunikation zwischen Server und Client gespart werden.

²⁷Denormalisieren beschreibt den Prozess der Relationsauflösung von Datenbanktabellen.

²⁸<https://www.npmjs.com/package/google-polyline>

4.5 Anzeigen aller Stationen

Nachdem zuvor alle Trips mit ihren Polylines in der Karte angezeigt wurden, folgt nun das Anzeigen aller Stationen, die zu einem Trip gehören. Ähnlich wie beim Abfragen und Anzeigen aller Polylines, sollte dieser Schritt mögliche Engpässe oder unvorhergesehene Probleme aufdecken (Abbildung 22).

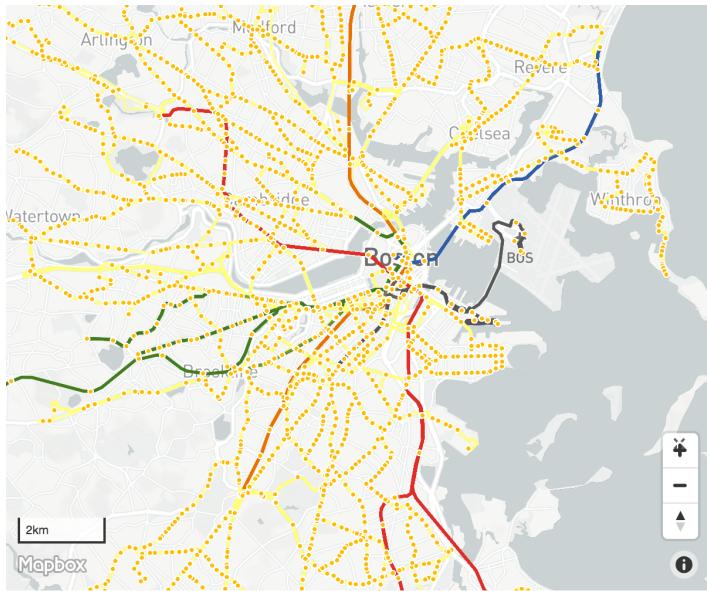


Abbildung 22: Aktive Trips mit ihren dazugehörenden Stationen

Vor allem die Abfrage von aktiven Trips aus der Datenbank stellte ein großes Problem dar. Die Datenbank konnte die Anfragen des Clients nicht effizient genug verarbeiten. An dieser Stelle stand fest, dass für die weitere Arbeit umfassende Optimierungen der Datenbank erfolgen müssen, um eine responsive Webanwendung zu ermöglichen.

4.5.1 GTFS-Optimierungen

Durch eine Optimierung des GTFS-Feeds lässt sich die Datenmenge bereits vor dem Importieren in die Datenbank erheblich verringern. Ein entsprechendes Tool hierfür ist `gtfstidy` <https://github.com/patrickbr/gtfstidy>. Es bietet allerdings nicht nur die Möglichkeit für die Vereinfachung von Polylines, sondern liefert eine ganze Reihe weiterer Optimierungsmöglichkeiten.

Der Kommandozeilenbefehl `$ gtfstidy -sSiRDe0 input.zip output` optimiert das Stuttgart-VVS-Feed wie folgt:

-s reduziert die Punktanzahl einer Polyline.

-S entfernt redundante Polylines.

-i wandelt Zeichen-ID's (String) in Zahlen-ID's (Integer).²⁹

²⁹Aus der String ID '1.T0.10-1-j17-1.16.H' wird 78

- O** entfernt Feed-Einträge, die nicht referenziert werden.
- R** entfernt doppelt vorhandene Routen.
- e** setzt fehlerhafte oder optionale Felder auf einen Standardwert.
- D** entfernt fehlerhafte Einträge aus dem Feed.

Durch Verwendung von gtfstdiy konnte das Feed optimiert werden und die Datengröße der einzelnen Dateien um folgendes Maß verringert werden:

Dateiname	Größe davor	Größe danach
trips.txt	6 MB	2.8 MB
stop_times.txt	103 MB	53 MB
stops.txt	651 KB	355 KB
shapes.txt	77.3 MB	22.4 MB
routes.txt	54 KB	38 KB
calendar_dates.txt	557 KB	463 KB

Tabelle 3: Tabellengröße vor und nach der Anwendung von gtfstdiy

Insgesamt konnte so die Größe des Feeds von 79 MB auf 118 MB um knapp 50% verringert werden. Vor allem die Umwandlung von langen String-ID's in kürzere Integer-ID's trägt maßgeblich zur Verringerung der Dateigröße bei.

4.5.2 Denormalisierung der Datenbank

Um an die Daten der aktiven Trips zu gelangen, müssen alle relevanten Tabellen mittels SQL JOIN über eine SQL-Abfrage³⁰ miteinander verknüpft werden. Die Verknüpfung erfolgt durch die Verbindung der einzelnen Reihen zweier Tabellen (TabelleA und TabelleB) gegen eine Verknüpfungsbedingung. Das Resultat ist eine neue Ergebnistabelle mit den Inhalten der kombinierten Reihen. Solche Verknüpfungen sind besonders dann zeitintensiv, wenn eine große Menge an Daten (siehe Tabelle: 4) kombiniert werden müssen.

Tabelle 4: Tabellen Metriken

Tabellen Name	Anzahl Reihen
trips.txt	71,000
stop_times.txt	1,3000,000
stops.txt	7,900
shapes.txt	1,085,860

Sollen alle Trips in einem Zeitraum von 1 - 15 Minuten gefunden werden, sind bereits Rechenzeiten entstanden, die aufgrund ihrer langen Laufzeit abgebrochen werden mussten. In

³⁰Die dazugehörige SQL-Abfrage ist aufgrund seiner Länge im Anhang unter Listing 6 zu finden.

mehreren Iterationen wurde versucht, die SQL-Abfrage zu optimieren, was allerdings keine Verbesserung herbeiführte. Es sind zu viele JOIN Verknüpfungen und WHERE Bedingungen in dieser Abfrage, als dass sich eine performante Lösung damit finden ließe. Es musste ein neuer Ansatz gefunden werden, um die Abfragezeiten erheblich zu verringern.

Die Denormalisierung ist eine Strategie, die auf eine zuvor normalisierte Datenbank angewendet wird, um die Leistung zu erhöhen. Dabei wird versucht, die Leseperformance einer Datenbank zu verbessern, indem redundante Kopien von Daten hinzugefügt oder Daten gruppiert werden. Damit wird zwar die Schreibleistung verringert, die Lesegeschwindigkeit jedoch beschleunigt.[13] Da die Daten ausschließlich ausgelesen und nicht geschrieben werden, ergeben sich nur Vorteile.

Um diese Methode anwenden zu können, wird eine neue Tabelle generiert, die den Zugriff auf die benötigten Daten vereinfacht. Im Grunde handelt es sich um eine Vorberechnung. Anstatt die Tabellen bei jeder Anfrage an den Server aufwändig über viele SQL-JOINS zu verknüpfen, wird diese Verknüpfung einmalig vorberechnet und in einer Tabelle gespeichert. Die Denormalisierung einer Tabelle wurde bereits im vorherigen Abschnitt „Aggregieren der Shape-Tabelle“ aufgezeigt. Dadurch ließ sich die Polyline über eine einzige Tabellenreihe abfragen, was die Performance signifikant erhöhte. Zum besseren Verständnis der Denormalisierungsmethode soll folgende Abbildung dienen:

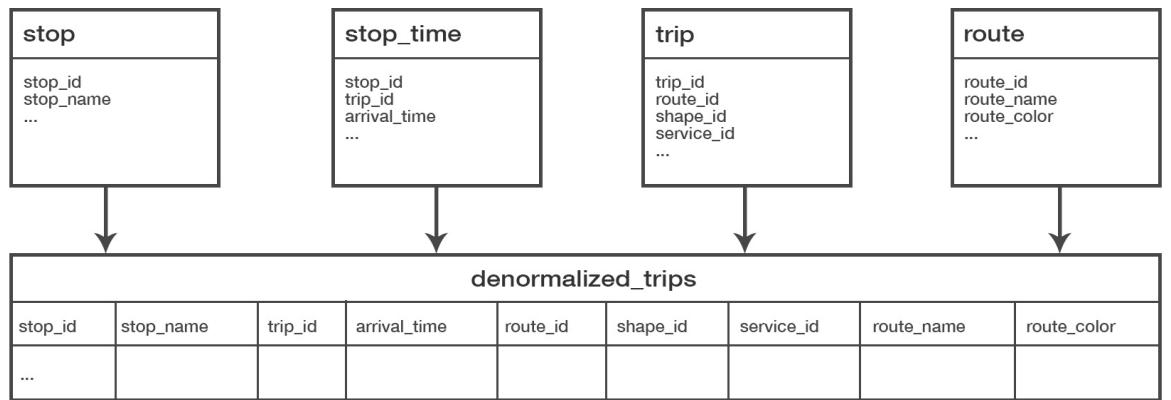


Abbildung 23: Beispiel einer Denormalisierung von Tabellen

Abbildung 23 zeigt, wie aus einer vertikalen Anordnung eine horizontale Anordnung der einzelnen Datenfelder in einer einzigen **denormalized_trips** Tabelle erzeugt wird. Eine Reihe in dieser neuen Tabelle steht für genau einen Eintrag eines Trips. Anstatt also bei jeder Anfrage an den Server die verschiedenen Daten mittels JOIN verknüpfen zu müssen, können diese jetzt per Zugriff auf eine einzige Reihe in nur einer Tabelle abgefragt werden.

Dieses Prinzip der Gruppierung von Daten in einer neuen Tabelle soll nun auch auf die anderen benötigten Tabellen angewendet werden. Die Denormalisierung erfolgt in 3 Schritten:

1. Erstellen der neuen Tabelle `denormalized_trips`
2. Importieren der verschiedenen Daten in diese neue Tabelle
3. Abfragen der Datenn über diese neue Tabelle

Das SQL-Statement ist abermals aufgrund seiner Länge dem Anhang 8 zu entnehmen. Dargestellt in einer Tabelle, sieht das SQL-Statement wie folgt aus:

stops_coords	stops_object	stops_dist_traveled	agency_key	trip_id	route_id	shape_id	trip_start_time	trip_end_time	direction_id	route_long_name	route_color	route_type
[{"48.66864000000000...": "1", "Böbl... Diezern. (Zentrum) S... 0,460,20096000000009,1043,113800000035,9.0050329...}	0000008,1418.86930000000007,1780.8...	stuttgart-vvs	1	224	3418		74700	75300	0	Böbl. Kreiskrankenh.- Thermalbad - ZOB - Diezenhalde	FF0000	700
[{"2129", "1", "Rüdern... 0,491,1166400000000018,874.169999999972,9.2826140...}	9999959,1174.0781999999992,1395.9...	stuttgart-vvs	2	398	3789		84600	85380	1	Rüdern - Sulzgries - Esslingen (N) ZOB	FF0000	700
[{"761", "1", "Kirchheim [0,2610.72310000000018,6404.54439999994,9.4438720...": "1", "Kirchheim 0,2610.72310000000018,6404.54439999994,9.4438720...]	9999977,10355.1110000000008,12990...	stuttgart-vvs	3	78	3967		51660	56580	0	Kirchheim (T) - Plochingen - Stuttgart - Herrenberg	83B23B	109

Abbildung 24: Auszug aus der `denormalized_trips`-Tabelle

Ergebnisse der Denormalisierung

Für die Visualisierung ist eine Abfrage der aktiven Trips am wichtigsten, da diese die Daten für die Darstellung der Vehicle auf der Karte liefert. Wie in Abbildung 25 zu sehen ist, wird auf die `denormalized_trips`-, `denormalized_shapes`- und die `calendar_dates`-Tabellen zugegriffen. Dabei werden die einzelnen Tabellen über die `shape_id` und `service_id` miteinander verknüpft. Das Resultat ist die Ausgabe aller aktiven Trips in einem variabel definierbarem Zeitraum des aktuellen Datums.

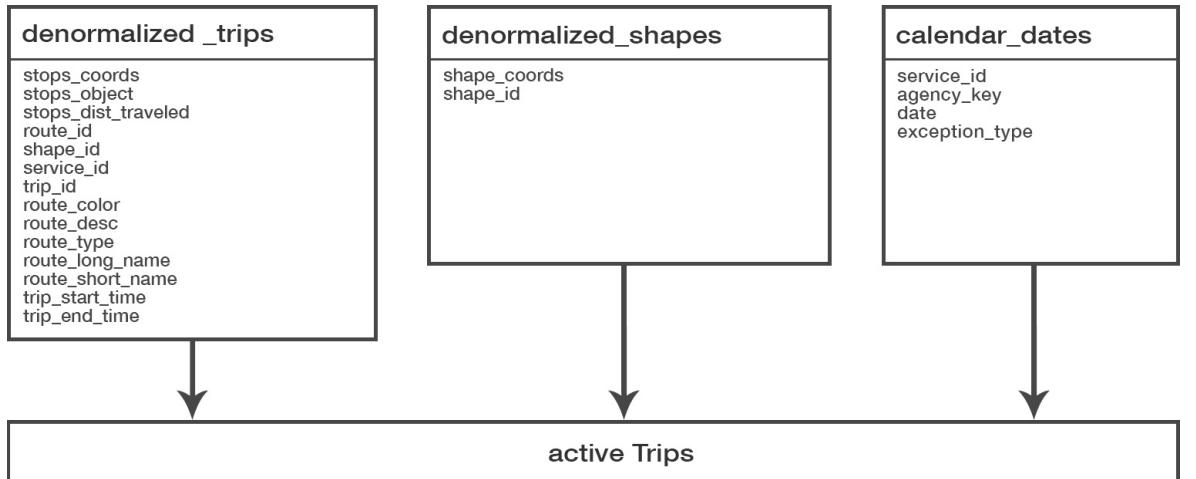


Abbildung 25: Benötigte Tabellen zur Abfrage von Trips

Nachfolgend werden in Tabelle 5 die Ergebnisse der Evaluation für die Abfrage der aktiven Trips in einem wachsenden Zeitraum dargestellt. Die verwendete SQL-Abfrage befindet sich im Anhang unter Listing 9.

Tabelle 5: Evaluierung der Denormalisierung

Zeitraum	Trip Anzahl	Query Zeit
9:00 bis 9:15	88	98 ms
9:00 bis 10:00	1125	154 ms
9:00 bis 12:00	3360	285 ms
9:00 bis 15:00	7070	497 ms
9:00 bis 21:00	14718	900 ms

Die Ergebnisse zeigen, dass die Abfragezeit der Datenbank für die aktiven Trips erheblich gesunken ist. Zu Projektbeginn war eine solch effiziente Abfrage aufgrund der überaus langen Laufzeit ($> 1\text{min}$) nicht möglich.

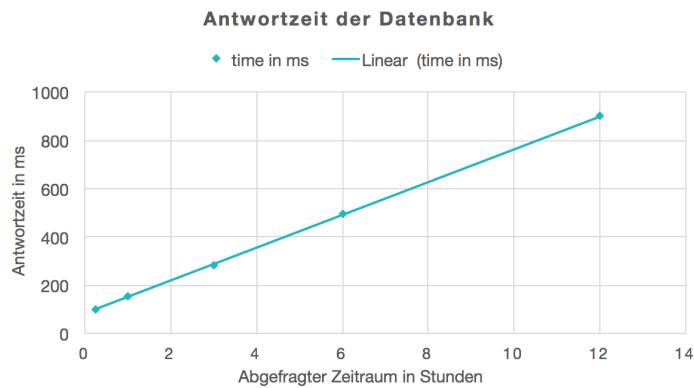


Abbildung 26: Plot der Abfragezeiten

Abbildung 26 zeigt einen Plot der Query-Zeit aus Tabelle 5 als linearen Graphen. Daraus ist ersichtlich, dass die Antwortzeit der Datenbank linear mit dem abgefragten Zeitraum wächst. Für die Visualisierung sind vor allem zwei Abfragen wichtig: Erstens das Abfragen eines größeren Zeitraums von 1-2 Stunden. Dies geschieht beim ersten Aufrufen der Webanwendung wenn die Karte noch keine Trips besitzt und damit leer ist. Zweitens die Abfrage von kleinen Zeiträumen von nur einer Minute, um neue Trips abzufragen. Für diese zwei Abfragen bewegt sich die Antwortzeit des Servers zwischen $\approx 80 - 160\text{ ms}$. Damit wurde das in Kapitel 3.4 gesetzte Ziel von 0 bis 200ms bereits erreicht.

4.6 Animieren der aktiven Trips

Im Client findet die Visualisierung statt. Er stellt die erste Anfrage an den Server, um alle Trips in einem Zeitraum zu bekommen. Die Antwort vom Server ist ein Objekt, bestehend aus einer ID und einer **GeoJSON-FeatureCollection**. Die Verwendung von GeoJSON hat den Vorteil, dass verschiedene Bibliotheken für dieses Format zur Verfügung stehen, die dessen Verarbeitung vereinfacht. Auch Mapbox setzt auf die Verwendung von GeoJSON und ist fest damit verbunden. Der Nachteil von GeoJSON ist seine sehr wortreiche Beschreibung. Dass macht es zwar für Menschen gut lesbar, allerdings auf Kosten der Datengröße.

Algorithm 1 Animate Vehicle

```
1: ServerQueryTimer ← 30 Seconds
2: Vehicles ← Vehicles Inside Bounding Box
3: Trips ← Requested Trips
4: function ANIMATE(timestamp)
5:   for all Vehicles as Vehicle do
6:
7:     if Vehicle started its Trip then
8:       CALCULATEVEHICLEPOSITION(Vehicle)
9:     end if
10:    if Vehicle not started its Trip then
11:      CHECKVEHICLEACTIVITY(Vehicle, Trips)
12:    end if
13:    CHECKIFVEHICLEHASFINISHED(Vehicle)
14:    UPDATERMAPWITHNEWPOSITIONS(Vehicles)
15:  end for
16:  if ServerQueryTimer Expired then
17:    Query Server for New Trips
18:    ServerQueryTimer ← 30 Seconds
19:  end if
20:  ANIMATE(timestamp)
21: end function
```

Nachdem die Daten für alle aktiven Trips im Client ankommen, kann für jeden dieser Trips ein Vehicle erstellt werden. In einem Animation-Loop wird für jedes Vehicle pro Frame eine neue Position berechnet und die Karte wird mit der neuen Position aktualisiert. Algorithmus 1 beschreibt dies in Pseudo-Code.

Innerhalb dieses Animation-Loops passieren mehrere Dinge. Zuerst wird geprüft, ob sich ein Vehicle überhaupt im Sichtbereich des Anwenders befindet. Trifft das zu, werden für eben diese Vehicle die Distanzen berechnet und die Position des Vehicles entlang der Polyline interpoliert. Falls das Vehicle seinen Trip noch nicht begonnen hat, wird überprüft, ob dies immer noch der Fall ist. Anschließend werden alle Vehicle geprüft, ob sie ihren Trip bereits beendet haben. Danach wird die Karte mit den neuen Positionen der Vehicle aktualisiert. Während all dies geschieht, läuft ein Timer mit, der nach dem Ablauen von 30 Sekunden den Server nach neuen Trips abfragt, womit der Abfragekreislauf von vorn beginnt.

Das Ergebnis ist die Animation aller Vehicle der momentan aktiven Trips (Abbildung 27).

Verbesserung der Client Performance

Damit die Animationen auf der Karte bei 60 FPS möglich sind, werden mehrere Optimierungsschritte ausgeführt.

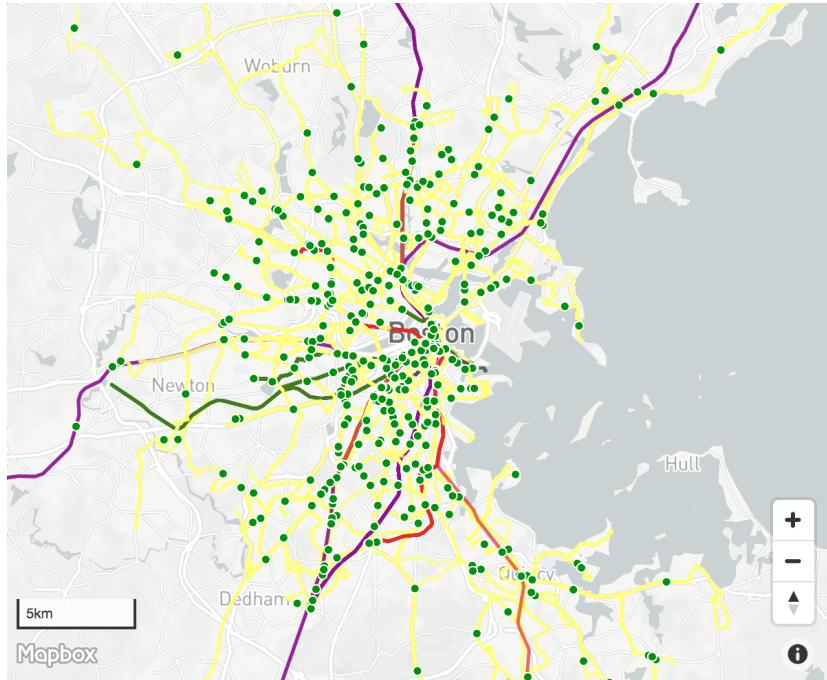


Abbildung 27: Animieren aller Vehicles auf der Karte

- **Manipulieren der FPS:** Je niedriger das Zoom-Level der Karte, umso geringer wird die FPS eingestellt. Das hat den Hintergrund, dass bei niedrigem Zoom die Bewegung der Vehicle fast nicht mehr wahrnehmbar ist. Wohingegen bei höherem Zoom die Animation umso flüssiger sein muss. Folgende Grenzen haben sich bei Tests als gute Werte erwiesen:

- Zoom Level < 12 → 1 FPS
- Zoom Level > 15 → 60 FPS
- Sonst → 8 FPS

Dieses Vorgehen bringt den Vorteil, dass bei niedrigem Zoom viel mehr Vehicle angezeigt werden, aber diese nur noch auf 1 FPS animiert werden müssen. Bei hohem Zoom ist dies genau umgekehrt. Dort sind nur noch wenige Vehicle sichtbar, diese werden aber dafür bei 60 FPS animiert. Somit ist einerseits eine gute Performance bei vielen Vehicles möglich und andererseits die Animation bei genauerer Betrachtung trotzdem sehr flüssig.

- **Speichern von Zuständen:** Um Rechenleistung einzusparen, werden wann immer möglich, ausgerechnete Werte abgespeichert, damit diese nicht nochmals berechnet werden müssen. Zum Beispiel (siehe Abbildung 28) weiß das Vehicle, auf welchem Polyline Segment³¹ es sich befindet.

Das bedeutet, dass die Richtung des Vehicles nicht neu berechnet werden muss, solange es diesem Segment folgt. Erst wenn das Vehicle von Segment *A, B* auf ein neues Segment *B, C* übergeht, muss die Richtung neu berechnet werden.

³¹Ein Segment sei in diesem Kontext ein gerader Linienteil der Polyline, bestehend aus zwei Punkten *A, B*.

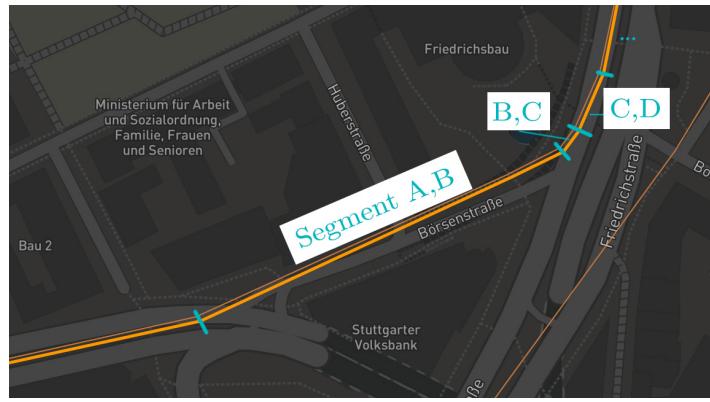


Abbildung 28: Polyline Segmente

- **Aufteilen der Vehicle in zwei Gruppen:** Da der User durch seinen Bildschirm meistens nur einen Teil der Vehicle zu sehen bekommt, sind die Vehicle in die Gruppen unterteilt. Namentlich seien sie als **Innerhalb** und **Außerhalb** benannt. Die Gruppe Innerhalb besitzt all diejenigen Vehicle, die sich im Sichtfeld des Anwenders befinden. Diese werden bei der Animation bevorzugt behandelt und erhalten die volle Rechenleistung. Die Gruppe Außerhalb liegt nicht im Sichtfelds und wird maximal jede Sekunde nach ihrer Aktivität geprüft. Dadurch bleibt auch diese Gruppe immer aktuell.

5 Deliver

In diesem Kapitel sollen das Endprodukt und seine Funktionsweise im Gesamtzusammenhang vorgestellt werden. Der erste Abschnitt widmet sich dem generellen Funktionsprinzip und der Kommunikation zwischen Server und Client, bevor im zweiten Abschnitt die einzelnen Funktionen der fertigen Webanwendung beschrieben werden. Im letzten Abschnitt wird schließlich überprüft, inwiefern die Zielsetzungen bezüglich der Performance erreicht werden konnten.

5.1 Funktionsprinzip

Für einen Datenaustausch zwischen Server und Client, sind in der fertigen Webanwendung folgende API³²-Endpunkte vorhanden.

- `/daily` stellt die Daten für das Zeitstrahldiagramm bereit und wird beim Start der Anwendung einmalig angefragt. Die Antwort enthält XY-Wertpaare. X stellt dabei die Zeit in Sekunden und Y die zu diesem Zeitwert aktiv werdenden Trips dar.

Listing 4: Antwort des Servers zur Anfrage `/daily`

```
1  [
2    {"x":86340, "y": "6"},
3    {"x":86400, "y": "10"},
4    ...
5  ]
```

- `/trips/:from,:to` ermöglicht das Abfragen von Trips, die in einer Zeitspanne `from - to` aktiv sind. Beim initialen Aufruf der Webanwendung wird dieser Endpunkt als Erstes angefragt, um die aktiven Trips innerhalb einer Stunde zu erhalten. Der gewählte Zeitraum ist in Sekunden anzugeben.

Die Antwort des Servers auf einen Endpunkt vom Typ `/trips/` ist ein Objekt mit der `Trip_Id`, dessen Inhalt der `GeoJSON` Spezifikation nach RFC 7946 folgt:

Listing 5: Trip Objekt

```
1  {
2    2498: {
3      "type": "FeatureCollection",
4      "features": [
5        {
6          "type": "Feature",
7          "properties": {
8            "name": "shape",
9            ...
10           },
11          "geometry": {
12            "type": "LineString",
```

³²Application Programming Interface

```

13         "coordinates": [[9.4437, 48.64482], ...]
14     }
15 },
16 {
17     "type": "Feature",
18     "properties": {
19         "name": "station"
20     },
21     "geometry": {
22         "type": "Point",
23         "coordinates": [9.443688, 48.6448]
24     }
25 },
26 ...
27 ]
28 }
29 }
```

Da die Antwort in Listing 5 mittels „...“ gekürzt ist, sind detailliertere Antworten im Anhang unter Listing 10, 11 und 12 zu finden.

- **/trips/:id** antwortet mit den zur ID gehörenden Trip-Informationen. Dieser Endpunkt ermöglicht es, Informationen für nur einen einzigen Trip zu bekommen. Dies ist vor allem dann hilfreich, wenn der Nutzer ein Vehilce anklickt und Informationen über diesen Trip angezeigt bekommen möchte. Beispiel: `/trips/51295`
- **/trips/new/:from,:to,:tripIds** stellt die Abfrage für neue Trips zur Verfügung und exkludiert dabei diejenigen Trips, die in `:tripIds` genannt sind. Damit wird verhindert, dass bereits auf der Karte vorhandene Trips nicht doppelt auftauchen können. Diese Datenbankabfrage wird in einem 30-Sekunden-Intervall vom Client an den Server gesendet, um die neusten Trips zu erhalten. Damit wird die Karte aktuell gehalten. Beispiel: Es ist 10:00 Uhr, hole die in der nächsten Minute aktiv werdenden Trips (Zeitraum 10:00 bis 10:01 Uhr) und schließe die Trips mit der ID 51295,9212,52 vom Ergebnis aus `/trips/new/36000,36060,51295,9212,52`.
- **/trips/new/:from,:to** stellt die gleiche Funktionalität wie der vorherige Endpunkt zur Verfügung, mit der Ausnahme, dass keine Trip-ID's übermittelt werden müssen. Dieser Endpunkt ist beispielsweise dafür da, falls die Karte leer ist und noch keine aktiven Trips enthält.

Das generelle Prinzip der Webanwendung beruht auf einer Client / Server Architektur. Der Nodejs-Server stellt verschiedene Endpunkte mittels Express als ansprechbare Routen dem Client zur Verfügung.

Trifft eine valide Anfrage auf den `/trips/:from,:to` Endpunkt, so wird ein Ablauf nach Abbildung 29 angestoßen. Die eintreffenden Anfragen werden vom Server entgegengenommen,

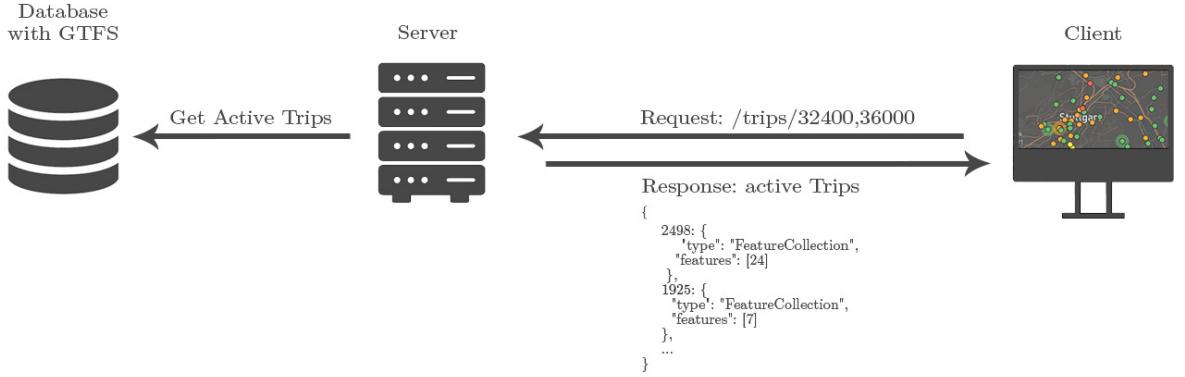


Abbildung 29: Server / Client Relation

validiert, verarbeitet und anschließend die entsprechende Antwort zurückgesendet. Die Validierung prüft die vom Client übergebenen Parameter auf ihre Plausibilität. Schlägt diese Prüfung fehl, wird ein Fehler vom Server zurückgegeben und der Server wartet auf eine neue Anfrage. Die wichtigste Routine des Servers stellt die Abfrage von Trips aus der Datenbank dar. Die Datenbank sucht diejenigen Trips heraus, welche in dem benötigten Zeitraum `from`, `to` aktiv sind. Dabei wird das Datum und der Wochentag zum Zeitpunkt der Anfrage verwendet. Um die Rechenarbeit im Client zu minimieren, werden alle Daten bei denen dies möglich ist, auf dem Server vorberechnet. Die aus der Datenbank abgefragten Daten durchlaufen folgenden Transformationsprozess:

- **Daten Mapping:** Die Trips aus der Datenbank werden in das `GeoJSON`-Format umgewandelt, damit diese im weiteren Programmverlauf einfacher zu verarbeiten sind. Dabei werden die im Kapitel „3.5 Begriffe und Definitionen“ festgelegten Regeln beachtet.
- **Zurückgelegte Distanz:** Damit eine Animation der Vehicle stattfinden kann, ist die Berechnung der Distanzen zwischen den einzelnen Stationen nötig.
Falls das Feld `dist_traveled`³³ in der Datenbank vorhanden ist, kann die zurückgelegte Distanz sehr einfach daraus berechnet werden. Ist dies nicht der Fall, so wird das in Abschnitt 4.2 beschriebene Station-Matching durchgeführt, um die Distanzen berechnen zu können.
- **Feststellen der Richtung:** Für eine Polyline ist es unerheblich, ob die Koordinaten in der Reihenfolge $\{p_1, p_2, \dots, p_n\}$ oder $\{p_n, \dots, p_2, p_1\}$ angeordnet sind. Damit das Vehicle aber in die richtige Richtung von A nach B fährt, ist es wichtig, dass die Koordinaten der Polyline in aufsteigender Reihenfolge festgelegt werden. Falls dies nicht der Fall ist, werden die Koordinaten in ihrer Reihenfolge umgekehrt.
- **Zeit zwischen Stationen:** In diesem Schritt wird die Fahrzeit (in sec) zwischen den einzelnen Stationen der Trips vorberechnet.

³³Die zurückgelegte Distanz bis zu einer Station S

- **Codieren der Polyline:** Hier werden die Koordinaten in einen Polyline-String codiert.
- **Versenden:** Zuletzt wird die Anfrage des Clients vom Server mit einem Response-Paket beantwortet und der Prozess ist damit abgeschlossen bis eine neue Anfrage den Server erreicht.

5.2 Funktionen der Webanwendung

In diesem Abschnitt sollen die verschiedenen UI-Komponenten vorgestellt werden.

Die Karte

Die Karte (30) ist standardmäßig auf den Längengrad 9.244 und Breitengrad 48.757 ausgerichtet. Damit findet sich der Anwender beim Aufrufen der Applikation gleich an der richtigen Stelle wieder. Die Karte verwendet eine abgeänderte Version des Kartenstils **Mapbox-Dark**. Dabei wurden Parks, Grünflächen und Wasser subtil eingefärbt und die Routen des GTFS-Feeds mit einem leichten Orange hervorgehoben.

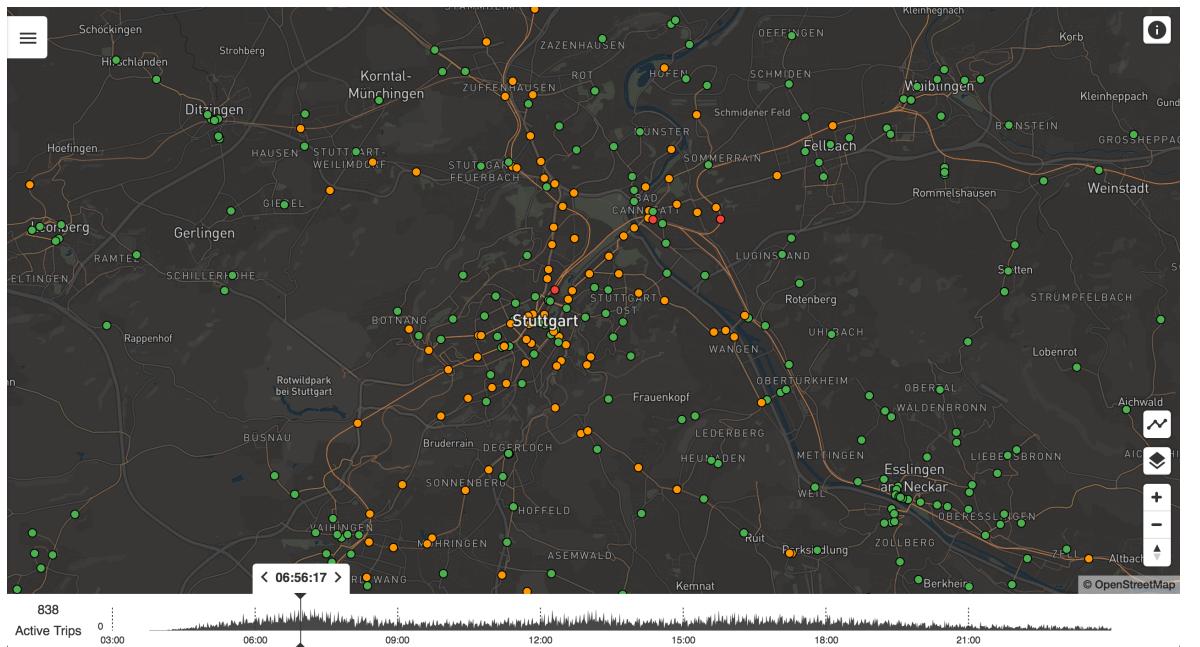


Abbildung 30: Die Karte mit angepasstem Style: Mapbox-Dark

Vehicle

Die Vehicle sind auf der Karte als Kreis dargestellt. Die verwendete Farbe orientiert sich dabei an den offiziellen Farben des jeweiligen Verkehrsunternehmens. Zum Beispiel sind Interrail-Züge im Rot der Deutschen Bahn dargestellt und die U- und S-Bahnen im Orange des Stuttgart-VVS.

Vehicle werden auf der Karte animiert, wenn sie aktiv sind. Abbildung 31 zeigt die zwei verschiedenen Animationen, die ein Vehicle beim Start und Beenden des Trips annehmen kann. Wird ein Trip aktiv, so wird das dazugehörige Vehicle mit vergrößertem Radius auf die Karte platziert. Danach wird der Radius des Vehicles in kurzer Zeit verringert, bis er der Radiusgröße der anderen Vehicle entspricht.



Abbildung 31: Vehicle Status Anzeige

Ist ein Vehicle dabei, seinen Trip innerhalb von 30 Sekunden zu beenden, so wird ein leicht transparentes Pulsieren angezeigt. Nachdem das Vehicle den Trip beendet hat, wird es von der Karte genommen und verschwindet. Technisch betrachtet, werden erst alle Referenzen auf das Vehicle beseitigt und anschließend das Vehicle-Objekt gelöscht.

Zeitstrahl

Die Webanwendung besitzt einen interaktiven Zeitstrahl am unteren Bildrand. Wie in Abbildung 32 zu sehen, besteht der Zeitstrahl aus mehreren Einzelteilen.



Abbildung 32: Zeitstrahl-Komponente

Links unten ist die Anzahl an momentan aktiven Trips zu sehen. Diese Anzahl korreliert mit den Vehicles auf der Karte. Die Anzeige wird immer aktuell gehalten und steigt, falls neue Trips aktiv werden oder fällt, wenn ein Trip beendet ist. Der Zeitstrahl selbst zeigt die Anzahl an aktiv werdenden Trips pro Minute an. Bewegt der Anwender die Maus darüber, so bekommt er die genaue Trip-Anzahl zu einer Uhrzeit als Tooltip angezeigt. Ebenfalls ist es möglich, die Animation zu einem beliebigen Zeitpunkt anzuzeigen. Dafür kann der Anwender einfach auf die gewünschte Zeitmarke im Zeitstrahl klicken und die Animation aktualisiert sich. Damit lässt sich die Karte zu unterschiedlichen Tageszeiten untersuchen. Zuletzt ist auch die gewählte Uhrzeit auf dem Zeitstrahl zu sehen. Diese zeigt dem Anwender, welcher Zeitpunkt momentan auf der Karte angezeigt wird.

Wechseln der Kartendarstellung

Über das **Switch Style**-Element  hat der Anwender die Möglichkeit, zwischen verschiedenen Darstellungsarten der Karte zu wechseln. Auch die Polyline der Routen lassen sich zusätzlich über das Anwählen von **Shape** ein- oder ausblenden. Standardmäßig ist **Dark** ausgewählt.

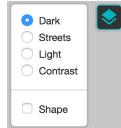


Abbildung 33: Wechseln zwischen verschiedenen Kartendarstellung

Vier verschiedene Kartenstile stehen zur Auswahl: Dark, Streets, Light und Contrast (Abbildung 34).

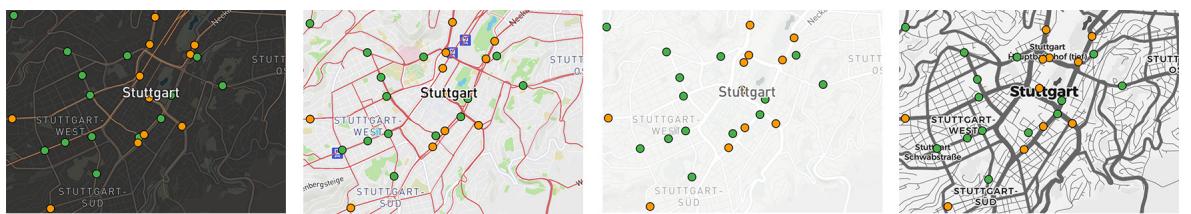


Abbildung 34: Verschiedene Auswahlmöglichkeiten der Kartendarstellung

Anzeigen von Trip-Informationen

Wenn der Anwender ein Vehicle in der Karte durch Klicken auswählt, öffnet sich ein Fenster, welches Informationen für diesen Trip anzeigt (Abbildung 35). Neben dem Namen der Route lässt sich im Kreis (hier in Rot) die Routen-Nummer ablesen. Im unteren Bereich sind die Fahrplaninformationen für den Trip gelistet. Neben dem Namen der Station ist auch die Abfahrtzeit des Vehicles gelistet. Die bereits besuchten Stationen werden in einem Grauton dargestellt, um sie als inaktiv zu markieren.

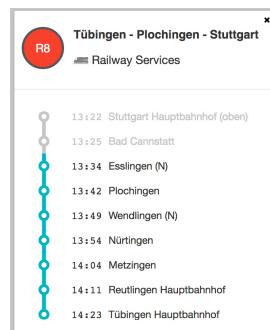


Abbildung 35: Anzeigen von Trip-Informationen

Filter

Über ein ausklappbares Menü lassen sich verschiedene Filter auswählen. Dadurch kann der Anwender zum Beispiel alle Vehicle eines Typs oder einer Linie abrufen. Auch Kombinationen der **Filter Vehicles** und **Filter Lines** sind möglich.

Damit der Anwender die Relation zwischen Filter und Vehicles auf der Karte versteht, sind die Farben einheitlich gestaltet. Nachdem ein Filter ausgewählt ist, kann das Menü entweder wieder zugeklappt oder der Filter abgewählt werden.

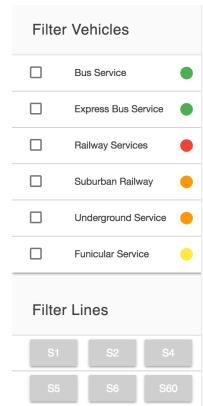


Abbildung 36: Filter-Funktion

Wegfindung

Durch Klicken des **Line Finder**-Buttons lässt sich auf der Karte eine Route finden, die zwei Stationen A, B verbindet. Dafür setzt der Anwender zwei Pins auf die Karte. Danach sucht ein Algorithmus diejenige Route aus, die am besten diese Stationen verbindet. Das Ergebnis sieht dann wie folgt aus:

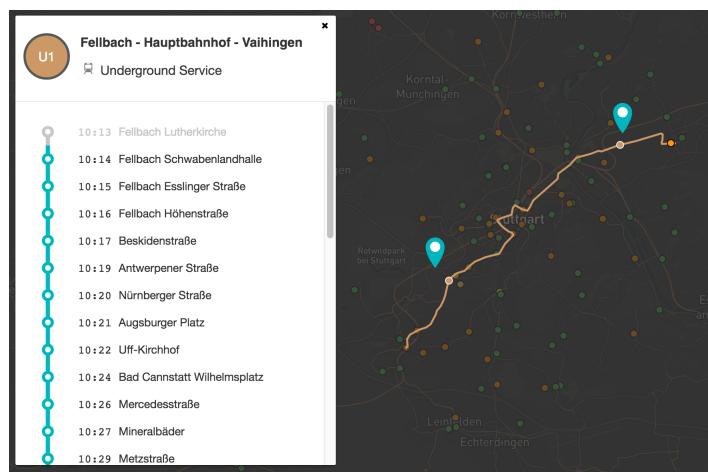


Abbildung 37: Linien Finder

Diese Funktion vereint Visualisierung und Wegfindung in einer Applikation. Während wir von anderen Applikationen gewohnt sind, eine Wegfindung ausschließlich über verschiedene Formularfelder (Von..., Nach..., Datum, Uhrzeit) anzufragen, könnte es für städtische oder regionale Wegfindung auch eine visuelle Lösung geben. Der Vorteil liegt dabei, dass kein Kontextwechsel nötig ist. Die Orientierung, Wegfindung und Fahrplaninformationen ließen sich alle in einer Ansicht vereinen. Ein weiteres Merkmal dieses visuellen Ansatzes ist die Möglichkeit, eine Route zu finden, ohne dass die Namen der Haltestellen bekannt sein müssen. Das auf der Karte dargestellte Ergebnis zeigt dem Nutzer auch sofort, wo sich das zur Route gehörende Vehicle momentan (oder zu einem bestimmten Zeitpunkt) befinden müsste. Damit könnte der Anwender auch gleich entscheiden, ob er dieses Vehicle noch erreichen würde oder nicht. Insbesondere an dieser Stelle wären GTFS-realtime-Informationen von Nutzen, mit welchen beispielsweise Verspätungen mit angezeigt werden können.

Von allen implementierten Komponenten bietet der Linien Finder das größte Potential für verschiedenste Weiterentwicklungen. Angefangen von der Implementierung von Echtzeitinformationen, über eine integrierte Anwender-Navigation (zum Beispiel könnte man den Anwender zu einer Station navigieren), bis zum Anbieten von Verbindungsanschlüssen bestünden Entwicklungsmöglichkeiten. Dariüber hinaus kann der Algorithmus zur Auswahl der empfohlenen Route noch sehr viel weiter verbessert werden. Momentan fließt vor allem die mittlere Distanz zwischen gesetztem Pin und nächster Station in die Entscheidung ein. Weitere Faktoren könnten aber noch berücksichtigt werden. Beispielsweise der Typ des Vehicles (U-Bahn bevorzugt gegenüber Bus), Frequenz der Route, Wartezeit auf das Vehicle, Preis, Reisezeit oder gar das momentane Verkehrsaufkommen.

5.3 Performance-Ergebnisse

In dieser Arbeit wird macOS Sierra 10.12.6 auf einem Macbook Pro 2,9 GHz Intel Core i5 mit 16 GB 1867 MHz DDR3 und Intel Iris Graphics 6100 1536 MB eingesetzt. Alle Messungen werden auf dieser Maschine ausgeführt. Das Projekt ist mittels einer **t2 Medium** AWS-Instanz in der Zone **EU-West** aufgesetzt.

Durch die Anwendung war es möglich, die Daten über einen ganzen Tag hinweg aufzuzeichnen und in einem Diagramm darzustellen. Dabei bilden die Daten die Aktivität des Öffentlichen Nahverkehrs in Stuttgart an einem Montag den 28.08.2017 zwischen 3.00 und 24.00 Uhr ab. Abbildung 38 zeigt, dass nach einem rapiden Anstieg in der Morgenzeit um 07:04 Uhr ein Höhepunkt mit 799 aktiven Trips erreicht wird.

Anschließend flacht mittags die Aktivität leicht ab, um dann zur Rush Hour am Abend wieder auf 767 gleichzeitig aktive Trips anzusteigen. Letztlich flacht die Anzahl immer weiter ab, um am nächsten Tag gegen 3 Uhr wieder anzusteigen und der Kreislauf beginnt von neuem. Insgesamt wurden an diesem Tag knapp 19650 Trips absolviert. Das Maximum betrug dabei $27 \frac{\text{Trips}}{\text{Minute}}$, wohingegen das Minimum bei $0 \frac{\text{Trips}}{\text{Minute}}$ lag. Im Schnitt starten 9 Vehicles pro

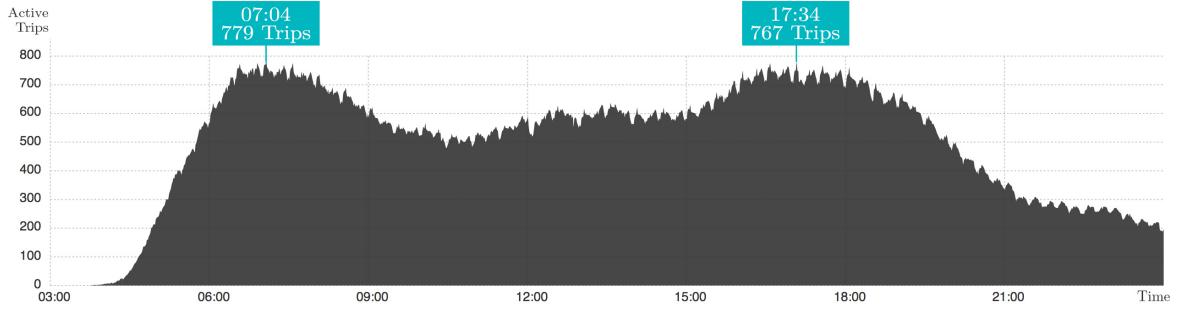


Abbildung 38: Anzahl an aktiven Trips zwischen 3.00 und 24.00 Uhr am 02.08.2017

Minute ihre Fahrt. Für eine interaktive Karte bedeutet dies, dass je nach Tag zwischen 0 und 1000 Trips aktiv sein können. Dies entspricht dann auch der Anzahl an Vehicles, die sich auf der Karte bewegen und animiert werden müssen. Sowohl durch die verschiedenen Frontend-Optimierungen aus Kapitel 4.6, als auch durch die Anpassung von verschiedenen Bibliotheksfunktionen, konnte durchgängig ein Rendering mit 60 FPS im Frontend erreicht werden. Damit wurde dieses gesetzte Teilziel erreicht.

Für die Ergebnisse der Evaluierung des Gesamtsystems wurden die Verarbeitungszeit des Servers, als auch die Antwortzeit der Datenbank in einem Zusammenhang betrachtet. Da das Projekt beim Abschluss dieser Arbeit noch nicht veröffentlicht wurde, konnten nur Messungen auf Localhost durchgeführt werden. Der Zugriff auf den AWS Server / Datenbank konnte nur über VPN erfolgen. Durch das Verbinden auf einen VPN-Server können keine genauen Messungen der Performance des Projektes auf AWS erfolgen, da immer die Netzwerkzeiten bis zum VPN-Server miteingeflossen wären. Damit bilden die getätigten Messungen den optimalen Zustand ab, wohingegen ein veröffentlichtes System etwas schlechter abschneiden würde. Folgende Metriken lassen sich für dieses Gesamtsystem feststellen:

Tabelle 6: Backend Evaluation

Anz. Trips	20	100	500	1000	5000	10000
Query Zeit (ms)	25	88	124	200	855	1631
Verarbeitungszeit (ms)	2	27	40	142	226	435
Summe (ms)	27	115	164	342	1081	2066

In Tabelle 6 sind die Datenwerte für verschiedene Abfragen aufgelistet. Die Werte ergeben sich aus dem Mittelwert der Laufzeit in 10 Durchläufen.

Aus dieser Aussage plus den gemessenen Werten lässt sich folgende Schlussfolgerung ziehen: Bei einer Anzahl zwischen 20 und 100 Trips reagiert der Server innerhalb von 25 bis 120ms. Da die meisten Trips in diese Spanne fallen, ist dieses Ergebnis am bedeutendsten.

Im Bereich von 100 bis 500 Trips ist eine Antwortzeit von 115 bis 164ms immer noch sehr gut. Diese Anzahl an Trips ist dann relevant, wenn die Applikation das erste Mal aufgerufen wird und die Karte noch leer ist. In diesem Fall kann es sein, dass der Server (je nach Datum und Uhrzeit) zwischen 200 - 500 Trips verarbeiten muss. Aber selbst Abfragen von bis zu 10.000 Trips, was ungefähr einer Zeitspanne von einem Tag gleich kommt, sind immer noch innerhalb von 2 Sekunden verarbeitet.

Abschließend kann gesagt werden, dass in dieser Arbeit ein performantes Backendsystem für eine Web-Applikation entwickelt worden ist, welches Serveranfragen effizient be- und verarbeiten kann.

6 Fazit

In dieser Arbeit wurde aufgezeigt, wie durch die Verwendung von GTFS-Daten eine Live-Visualisierung des Öffentlichen Nahverkehrs für die Region Stuttgart erstellt werden konnte. Durch Technologien wie Mapbox, Nodjes und PostgreSQL wurde eine visuell ansprechende Anwendung entwickelt, die neben der Darstellung aktiver Fahrzeuge im zeitlichen Verlauf weitere Funktionen wie die einer visuellen Wegfindung bietet. Trotz hoher Datenmengen konnten die erwünschten Performance-Zielsetzungen erreicht werden. Der Mehrwert einer solchen interaktiven Karte ergibt sich vor allem für den verkehrsanalytischen Bereich. Hier kann die Karte zur überblicksartigen Erfassung des Verkehrsnetzes und dessen Analyse beitragen. Verkehrsdaten werden auf diese Weise also sowohl erfahr- als auch explorierbar gemacht.

Für den gewöhnlichen Nutzer wäre der momentane Ansatz mit einer gesamten Abbildung des Öffentlichen Nahverkehrs einer Region allerdings auch mit Problemen behaftet. So ist die Übersichtlichkeit durch die Darstellung aller zu einem Zeitpunkt aktiven Fahrzeuge deutlich eingeschränkt und die Identifikation eines Fahrzeugs einer spezifischen Linie auf der Karte wird erschwert. Auch das Fehlen von GTFS-realtime in Stuttgart würde sich für ein konsumentenorientiertes Produkt nachteilig auswirken, da ohne Echtzeitkomponente viele Informationen, die für eine echte Live-Visualisierung wichtig wären, nicht angeboten werden können. Die Art der Anwendung suggeriert dem Nutzer zwar einen Ist-Zustand, da jedoch tatsächlich nur der Soll-Zustand übermittelt wird, bleiben Fahrplanänderungen oder -abweichungen unberücksichtigt.

Als Ausblick für dieses Projekt wären diverse Verbesserungen möglich. Die Verknüpfung mit einer Echtzeitkomponente würde am meisten Potenzial für Weiterentwicklungen bieten. Die verschiedenen Stadien, die ein Fahrzeug annehmen kann (verspätet, verfrührt, pünktlich), könnten in die UI-Elemente mit einfließen und auch auf der Karte in kreativer Weise verarbeitet werden. Beispielsweise wären Änderungen in der farblichen Darstellung eines Vehicles je nach Pünktlichkeit oder Verspätung möglich. Aber auch statistische Auswertungen wie beispielsweise der Anteil an Verspätungen im Vergleich zum Vortag oder über einen gewissen Zeitraum (zu 34% um 5 Minuten zu spät) könnten dem Anwender angezeigt werden. Möglich wäre auch, die „Soll“-Position, also wo sich das Fahrzeug laut Fahrplan eigentlich befinden sollte, im Vergleich zur tatsächlichen Position, anzuzeigen.

Darüber hinaus könnte der „Wegfinder“ aus Kapitel 5.2 als Möglichkeit einer visuellen Wegfindung, Potenzial für Optimierungen und Raum für kreative neue Ideen in sich bergen. Hier könnten Tests mit echten Anwendern zeigen, inwieweit diese Möglichkeit anklang findet oder welche Änderungen für eine verbesserte Nutzbarkeit notwendig wären.

Schlussendlich wurde aus dem Projekt noch die Erkenntnis gewonnen, dass Live-Karten im Allgemeinen wohl spezifischer auf die Anwendungsbereiche und Bedürfnisse entsprechender Nutzergruppen zugeschnitten sein müssten. Dies lässt sich aus der Zusammenschau der Vor- und Nachteile der in diesem Rahmen entwickelten Webanwendung, aber auch unter Berück-

sichtigung anderer bestehender Visualisierungen, schlussfolgern. Momentan gibt es zwar bereits verschiedene gute Produkte, die eine Live-Visualisierung des Öffentlichen Nahverkehrs - auch auf breiter Basis - abbilden, aber ich kenne niemanden in meinem Umfeld, der solch eine Anwendung verwendet. Dies kann zum einen daran liegen, dass all diese Anwendungen nicht primär für die native Nutzung per Smartphone - der Hauptnutzerbasis im Bereich Mobilität - entwickelt wurden, sondern eher auf Desktop oder Laptop-Bildschirmen gut darstellbar sind. Zum anderen benötigen unterschiedliche Usergruppen unterschiedliche Informationen. Ein Reisender beispielsweise ist im Zug auf andere Informationen angewiesen als ein Pendler. Ein Besucher in der Stadt findet sich anders zurecht als ein dort ansässiger Einheimischer. An einem öffentlichen Platz sind lokale, stationsbezogene Informationen am meisten relevant. Für Bahnstationen, aber auch für öffentliche Plätze oder gar Cafés, die nahe an einer Haltestelle liegen, könnte sich ein Live-Monitor nach Art von <http://transitscreen.com/live/> anbieten, welcher die nächstliegendsten Stationen und Linien anzeigt und dem Anwender umfassende Echtzeitinformationen rund um seinen Standort bietet. Deshalb wäre es bildlich gesprochen erfolgversprechender, anstatt einem großen Schweizer Taschenmesser, lieber viele kleinere Nieschenprodukte zu entwickeln.

Die im Rahmen dieser Masterarbeit entwickelte Live-Visualisierung stellt demgegenüber also eher ein Experten-Tool als eine Endanwender-Applikation dar. Allerdings könnte der entstandene Prototyp verwendet werden, um Evaluationen mit verschiedenen Endanwendern für eine nutzerspezifische Weiterentwicklung durchzuführen. Dadurch ließen sich eventuell neue spannende Erkenntnisse für die Entwicklung bisher unbekannter Produkte finden lassen.

7 Anhang

Listing 6: Postgresql Datenbankabfrage von allen aktiven Trips mit ihrem dazugehörigen Linienvorlauf

```
1  SELECT
2    array_to_json(ARRAY_AGG(array[stop_lat, stop_lon])) AS stops_coords,
3    array_to_json(ARRAY_AGG(array[
4      CAST ( stops.stop_id AS TEXT ),
5      CAST ( stop_times.stop_sequence AS TEXT ),
6      stops.stop_name,
7      stop_times.departure_time,
8      CAST ( stop_times.departure_time_seconds AS TEXT ),
9      stop_times.arrival_time,
10     CAST ( stop_times.arrival_time_seconds AS TEXT )
11   ] ORDER BY stop_times.stop_sequence)) AS stops_object,
12   routes.route_short_name,
13   routes.route_long_name,
14   routes.route_type,
15   routes.route_color,
16   routes.route_text_color,
17   trips.trip_id,
18   trips.route_id,
19   trips.direction_id,
20   frequency.start_time,
21   frequency.end_time,
22   frequency.start_time_seconds,
23   frequency.end_time_seconds,
24   frequency.headway_secs AS frequency_seconds,
25   calendar.start_date,
26   calendar.end_date,
27   agencies.agency_timezone,
28   agencies.agency_id,
29   agencies.agency_name,
30   array_to_json(ARRAY_AGG(array[shape_pt_lat, shape_pt_lon]
31     ORDER BY shape_pt_sequence ASC)) AS shape_coords
32 FROM gtfs_stop_times AS stop_times
33
34 INNER JOIN gtfs_trips AS trips
35   ON trips.trip_id = stop_times.trip_id
36   AND trips.agency_key = stop_times.agency_key
37
38 INNER JOIN gtfs_routes AS routes
39   ON routes.route_id = trips.route_id
40   AND routes.agency_key = trips.agency_key
41
42 INNER JOIN gtfs_agency AS agencies
43   ON agencies.agency_id = routes.agency_id
44   AND agencies.agency_key = routes.agency_key
45
```

```

46 INNER JOIN gtfs_stops AS stops
47   ON stops.stop_id = stop_times.stop_id
48   AND stops.agency_key = stop_times.agency_key
49   AND NOT EXISTS (
50     SELECT 0
51     FROM denormalized_max_stop_sequence AS max
52     WHERE max.agency_key = stop_times.agency_key
53     AND max.trip_id = stop_times.trip_id
54     AND max.trip_max = stop_times.stop_sequence
55   )
56
57 LEFT JOIN gtfs_calendar_dates AS calendar_dates
58   ON calendar_dates.service_id = trips.service_id
59   AND calendar_dates.agency_key = trips.agency_key
60   AND date = '2017-05-10'
61   AND calendar_dates.exception_type = 1
62
63 LEFT JOIN gtfs_calendar AS calendar
64   ON trips.service_id = calendar.service_id
65   AND calendar.agency_key = trips.agency_key
66   AND calendar.thursday = 1
67   AND '2017-05-10' BETWEEN calendar.start_date
68   AND calendar.end_date OR calendar_dates.date IS NOT NULL
69   AND calendar.start_date <= '2017-05-10'
70   AND calendar.end_date >= '2017-05-10'
71
72 LEFT JOIN gtfs_frequencies AS frequency
73   ON frequency.trip_id = trips.trip_id
74   AND frequency.agency_key = trips.agency_key
75
76 LEFT JOIN gtfs_shapes AS shapes
77   ON trips.shape_id = shapes.shape_id
78   AND trips.agency_key = shapes.agency_key
79
80 WHERE (
81   frequency.start_time_seconds >= 25200
82   AND frequency.end_time_seconds <= 90000
83   AND shapes.shape_id IS NOT NULL
84   AND '2017-05-10' BETWEEN calendar.start_date
85   AND calendar.end_date OR calendar_dates.date IS NOT NULL
86 )
87 AND NOT EXISTS (
88   SELECT 0
89   FROM gtfs_calendar_dates AS date_exceptions
90   WHERE date = '2017-05-10'
91   AND date_exceptions.agency_key = trips.agency_key
92   AND date_exceptions.service_id = trips.service_id
93   AND exception_type = 2
94 )
95 GROUP BY

```

```

96     routes.route_short_name,
97     routes.route_long_name,
98     routes.route_type,
99     routes.route_color,
100    routes.route_text_color,
101   trips.trip_id,
102   trips.route_id,
103   trips.direction_id,
104   frequency.start_time,
105   frequency.end_time,
106   frequency.start_time_seconds,
107   frequency.end_time_seconds,
108   frequency_seconds,
109   calendar.start_date,
110   calendar.end_date,
111   agencies.agency_timezone,
112   agencies.agency_id,
113   agencies.agency_name

```

Listing 7: Erstellen einer denormalized_shapes Tabelle

```

1 -- first we creat the new database table
2 CREATE TABLE denormalized_shapes (
3   agency_key text,
4   shape_id INTEGER,
5   shape_coords json,
6   shape_dist_traveled double precision[]
7 );
8
9 -- generating indexes
10 CREATE INDEX denormalized_shapes_unique_index
11   ON denormalized_shapes (agency_key, shape_id);
12
13 -- insert the shape data into new table
14 INSERT INTO denormalized_shapes
15 SELECT
16   agency_key,
17   shape_id,
18   shape_coords,
19   shape_dist_traveled
20 FROM (
21   SELECT
22     shapes.agency_key,
23     shapes.shape_id,
24     array_to_json(ARRAY_AGG(array[shape_pt_lon, shape_pt_lat]
25       ORDER BY shape_pt_sequence ASC))
26     AS shape_coords,
27     ARRAY_AGG(shape_dist_traveled
28       ORDER BY shape_pt_sequence ASC)
29     AS shape_dist_traveled
30

```

```

31   FROM gtfs_shapes AS shapes
32
33   GROUP BY
34     shapes.agency_key,
35     shapes.shape_id
36 ) as shp;

```

Listing 8: Erstellen der denormalized_shapes Tabelle

```

1 -- create denormalized_trips table
2 CREATE TABLE denormalized_trips (
3   stops_coords json NOT NULL,
4   stops_object json NOT NULL,
5   stops_dist_traveled json,
6   agency_key TEXT NOT NULL,
7   trip_id INTEGER NOT NULL,
8   route_id INTEGER NOT NULL,
9   service_id INTEGER NOT NULL,
10  shape_id INTEGER,
11  trip_start_time INTEGER NOT NULL,
12  trip_end_time INTEGER NOT NULL,
13  direction_id INTEGER,
14  route_color TEXT,
15  route_long_name TEXT,
16  route_short_name TEXT,
17  route_desc TEXT,
18  route_type INTEGER
19 );
20 -- create indexes
21 CREATE INDEX denormalized_trips_index
22   ON denormalized_trips (agency_key, trip_id, route_id, shape_id);
23 -- now query the data and insert it into the newly created table
24 INSERT INTO denormalized_trips
25 SELECT
26   stops_coords,
27   stops_object,
28   agency_key,
29   trip_id,
30   route_id,
31   service_id,
32   shape_id,
33   trip_start_time,
34   trip_end_time,
35   route_color,
36   route_long_name,
37   route_short_name,
38   route_desc,
39   route_type
40 FROM (
41   SELECT
42     array_to_json(ARRAY_AGG(array[stop_lat, stop_lon]

```

```

43     ORDER BY stop_times.stop_sequence::int)) AS stops_coords,
44     array_to_json(ARRAY_AGG(array[
45         stops.stop_id,
46         CAST ( stop_times.stop_sequence AS TEXT ),
47         stops.stop_name,
48         stop_times.departure_time,
49         CAST ( stop_times.departure_time_seconds AS TEXT ),
50         stop_times.arrival_time,
51         CAST ( stop_times.arrival_time_seconds AS TEXT )
52     ] ORDER BY stop_times.stop_sequence::int)) AS stops_object,
53     trips.agency_key,
54     trips.trip_id,
55     trips.route_id,
56     trips.service_id,
57     trips.shape_id,
58     min(stop_times.arrival_time_seconds) as trip_start_time,
59     max(stop_times.departure_time_seconds) as trip_end_time,
60     routes.route_color,
61     routes.route_long_name,
62     routes.route_short_name,
63     routes.route_desc,
64     routes.route_type
65 FROM gtfs_stop_times AS stop_times
66
67 INNER JOIN gtfs_trips AS trips
68     ON trips.trip_id = stop_times.trip_id
69     AND trips.agency_key = stop_times.agency_key
70
71 INNER JOIN gtfs_routes AS routes
72     ON trips.agency_key = routes.agency_key
73     AND routes.route_id = trips.route_id
74
75 INNER JOIN gtfs_stops AS stops
76     ON stops.stop_id = stop_times.stop_id
77     AND stops.agency_key = stop_times.agency_key
78
79 GROUP BY
80     trips.agency_key,
81     trips.trip_id,
82     trips.route_id,
83     trips.service_id,
84     trips.shape_id,
85     routes.route_color,
86     routes.route_long_name,
87     routes.route_short_name,
88     routes.route_desc,
89     routes.route_type
90 ) as trps;

```

Listing 9: Abfrage von Trips und dessen Polyline in einer Zeitspanne XY zu einem gegebenen Datum Z

```

1  SELECT
2    trips.stops_coords,
3    trips.stops_object,
4    trips.stops_dist_traveled,
5    shapes.shape_coords,
6    trips.route_id,
7    trips.shape_id,
8    trips.trip_id,
9    trips.route_color,
10   trips.route_desc,
11   trips.route_type,
12   trips.route_long_name,
13   trips.route_short_name,
14   trips.trip_start_time,
15   trips.trip_end_time
16 FROM denormalized_trips AS trips
17
18 INNER JOIN gtfs_calendar_dates AS calendar_dates
19 ON calendar_dates.service_id = trips.service_id
20 AND calendar_dates.agency_key = trips.agency_key
21 AND date = 'YYYY-MM-DD'
22 AND exception_type = 1
23
24 -- GTFS STUTTGART-VVS does not make use of the calendar,
25 -- if you want to process other feeds aswell you might need
26 -- to join this table too
27 --LEFT JOIN gtfs_calendar AS calendar
28 -- ON trips.service_id = calendar.service_id
29 -- AND calendar.agency_key = trips.agency_key
30 -- AND calendar.$date = 1
31
32 LEFT JOIN denormalized_shapes AS shapes
33   ON trips.shape_id = shapes.shape_id
34   AND trips.agency_key = shapes.agency_key
35
36 WHERE (
37   trip_start_time BETWEEN XY AND XY
38   AND shapes.shape_coords IS NOT NULL
39 )
40 --AND (
41 -- 'YYYY-MM-DD' BETWEEN calendar.start_date AND calendar.end_date
42 -- OR calendar_dates.date IS NOT NULL
43 --)
44 AND NOT EXISTS (
45   SELECT 0
46   FROM gtfs_calendar_dates AS date_exceptions
47   WHERE date = 'YYYY-MM-DD'
```

```

48     AND date_exceptions.agency_key = trips.agency_key
49     AND date_exceptions.service_id = trips.service_id
50     AND exception_type = 2
51 )

```

Listing 10: Geojson FeatureCollection als Antwort zu /trips/:from,:to

```

1 {
2   2498: {
3     "type": "FeatureCollection",
4     "features": [24]
5   },
6   1925: {
7     "type": "FeatureCollection",
8     "features": [7]
9   },
10 ...
11 }

```

Listing 11: Shape Feature

```

1 "features": [
2   {
3     "type": "Feature",
4     "properties": {
5       "name": "shape",
6       "route_color": "#FF0000",
7       "route_type": 700,
8       "trip_id": 8784,
9       "route_short_name": "701",
10      "route_long_name": "Sindelf. Eichholz - ZOB - Böbl. ZOB - Diezenhalde",
11      "trip_start": 55560,
12      "trip_end": 57780,
13      "route_type_color": "#f44336",
14      "polyline": "kswv@x`jH_@f@}@?c@UiJ{AQHiBeAqDcCl@yAjAoB|@iAjBsC^cAOi@yCsA
15      _FgAq@MuB@_@YkAa@]A]cDgAUM_FLoa@qBcHq@T{@eCqD_EoDoNiJoJwIaGuHaDqGaCmM?cF|
16      @kFvLoSxDaF1G{EpAkA\mAfCbf@q@vA_C0_B[
17      m@DiB^eANgAG}APuAbAeARYRm@HyA@}ISyBEiAKsGDsCf]aBmBgP]
18      iDDiBNsAtBs@~GiBrAm@tCyEzCqCdDyB~FmC|AWhEY|R]
19      rHAxKDnFAdMFvc@lE~JtA|Ex@bD`@vKrBnG~@nDRdBdfONj]z@jK1A`DNvC@zGS|E_@`Dg@zF
20      sAxHqAlCa@rDKzIB|FLM\?f@|Fv@dBvC|@d@S@H_CjCPjA@pKLHqBF_GD_@"
21    },
22    "geometry": {
23      "type": "LineString",
24      "coordinates": []
25    }
26  }
27 ]

```

Listing 12: Station Feature

```

1 {

```

```

2   "type": "Feature",
3   "properties": {
4     "stop_id": "4637",
5     "stop_sequence": 1,
6     "stop_name": "Sindelf. Eichholz",
7     "departure_time_secs": "55560",
8     "arrival_time_secs": "55560",
9     "secs_between_stops": 60,
10    "dist_traveled": 0,
11    "delta_dist_traveled": 0
12  },
13  "geometry": {
14    "type": "Point",
15    "coordinates": [
16      9.002212,
17      48.73068
18    ]
19  }
20 }
```

Listing 13: Station Matching

```

1  const _ = require('lodash');
2  const knn = require('rbush-knn');
3  const rbush = require('rbush');
4  const turf = require('turf');
5
6 // own implementation of turf.js pointOnLine()
7 // https://github.com/Turfjs/turf/tree/master/packages/turf-point-on-line
8 function getPointOnLine(line, pt, units) {
9   let coords;
10
11  if (line.type === 'Feature') {
12    coords = line.geometry.coordinates;
13  } else if (line.type === 'LineString') {
14    coords = line.coordinates;
15  } else {
16    throw new Error('input must be a LineString Feature or Geometry');
17  }
18
19  let closestPt = turf.point([Infinity, Infinity], { dist: Infinity });
20  const start = turf.point(coords[0]);
21  const stop = turf.point(coords[1]);
22  start.properties.dist = turf.distance(pt, start, units);
23  stop.properties.dist = turf.distance(pt, stop, units);
24 // perpendicular
25  const heightDistance = Math.max(start.properties.dist, stop.properties.dist);
26  const direction = turf.bearing(start, stop);
27  const perpendicularPt1 = turf.destination(pt, heightDistance, direction + 90,
28    units);
28  const perpendicularPt2 = turf.destination(pt, heightDistance, direction - 90,
```

```

        units);
29  const intersect = turf.lineIntersect(
30    turf.lineString([perpendicularPt1.geometry.coordinates,
31      perpendicularPt2.geometry.coordinates]),
32    turf.lineString([start.geometry.coordinates, stop.geometry.coordinates]));
33  let intersectPt = turf.point([Infinity, Infinity], { dist: Infinity });
34  if (intersect.features.length > 0) {
35    intersectPt = intersect.features[0];
36    intersectPt.properties.index = line.properties.index1;
37    intersectPt.properties.dist = turf.distance(pt, intersectPt, units);
38  }
39
40  if (start.properties.dist < closestPt.properties.dist) {
41    closestPt = start;
42    closestPt.properties.index = line.properties.index1;
43  }
44  if (stop.properties.dist < closestPt.properties.dist) {
45    closestPt = stop;
46    closestPt.properties.index = line.properties.index2;
47  }
48  if (intersectPt && intersectPt.properties.dist < closestPt.properties.dist) {
49    closestPt = intersectPt;
50    closestPt.properties.index = line.properties.index1;
51  }
52  return closestPt;
53}
54 // own implementation of turf.js pointOnLine()
55 // https://github.com/Turfjs/turf/tree/master/packages/turf-line-slice
56 function lineSlice(startPt, stopPt, line) {
57  const start = turf.point(startPt, { index: 0 });
58  let coords;
59  if (line.type === 'Feature') {
60    coords = line.geometry.coordinates;
61  } else if (line.type === 'LineString') {
62    coords = line.coordinates;
63  } else {
64    throw new Error('input must be a LineString Feature or Geometry');
65  }
66  const ends = [stopPt, start];
67  const clipCoords = [ends[0].geometry.coordinates];
68  for (let i = ends[0].properties.index + 1; i < ends[1].properties.index + 1; i += 1) {
69    clipCoords.push(coords[i]);
70  }
71  clipCoords.push(ends[1].geometry.coordinates);
72  return turf.lineString(clipCoords, line.properties);
73}
74
75 function fillTree(trips) {

```

```

76  const tree = rbush(200, '[0]', '[1]', '[0]', '[1]', '[2]');
77  const coords = [];
78  for (let k = 0; k < trips.features[0].geometry.coordinates.length; k += 1) {
79    const item = [
80      trips.features[0].geometry.coordinates[k][0],
81      trips.features[0].geometry.coordinates[k][1],
82      { index: k },
83    ];
84    coords.push(item);
85  }
86  tree.load(coords);
87  return tree;
88 }
89
90 function matchStation(geojson) {
91   _.forEach(geojson, (trips) => {
92     const shape = _.first(trips.features);
93     for (let i = 1; i < trips.features.length; i += 1) {
94       const tree = fillTree(trips);
95       const lat = trips.features[i].geometry.coordinates[0];
96       const lng = trips.features[i].geometry.coordinates[1];
97       const neighbours = knn(tree, lat, lng, 2);
98       const line = turf.lineString([
99         [neighbours[0][0], neighbours[0][1]],
100        [neighbours[1][0], neighbours[1][1]]],
101        { index1: neighbours[0][2].index, index2: neighbours[1][2].index });
102       const pointOnLine = getPointOnLine(line, trips.features[i], 'kilometers');
103       const lineDistance =
104         turf.lineDistance(lineSlice(trips.features[0].geometry.coordinates[0],
105           pointOnLine, shape));
106       trips.features[i].properties.dist_traveled = lineDistance * 1000;
107       trips.features[i].geometry.coordinates = pointOnLine.geometry.coordinates;
108       if (i > 1) {
109         const deltaDist = trips.features[i].properties.dist_traveled -
110           trips.features[i - 1].properties.dist_traveled;
111         trips.features[i].properties.delta_dist_traveled = Math.abs(deltaDist);
112       } else {
113         trips.features[i].properties.delta_dist_traveled = 0;
114       }
115     });
116   }

```

Abbildungsverzeichnis

1	Screenshot der fertigen Webanwendung	5
2	„The Double Diamond“ eigene Abbildung nach [2]	6
3	Überblick über bestehende Tools und Visualisierungen	9
4	2x2 Matrik Methode auf DIN A2	9
5	Personas	10
6	UI Element mit GTFS Informationen	14
7	Benötigte GTFS Tabellen[6]	16
8	Karte mit integrierten Stuttgart-VVS Daten	20
9	Grafische Veranschaulichung einer Route	23
10	Anzeigen einer einzigen Polyline in Boston	24
11	Hinzufügen von Stationen entlang der Polyline	25
12	Stationen liegen nicht direkt auf der Polyline	26
13	Finde den nächstgelegenen Punkt der Station auf der Polyline	26
14	Berechnen der Distanz	26
15	Vergleich der zwei Station-Matching-Algorithmen	27
16	Interpolation der Vehicle Position: V_p	28
17	Interpolation eines Vehicles entlang seiner Polyline	30
18	Darstellung aller Polylines auf der Karte	30
19	Vereinfachung einer Polyline mittels Simplify.js	31
20	Reduzieren der Anzahl an Tabellen-Einträge via RDP	31
21	Aggregierte Koordinaten der Shape-Tabelle	32
22	Aktive Trips mit ihren dazugehörenden Stationen	33
23	Beispiel einer Denormalisierung von Tabellen	35
24	Auszug aus der <code>denormalized_trips</code> -Tabelle	36
25	Benötigte Tabellen zur Abfrage von Trips	36
26	Plot der Abfragezeiten	37
27	Animieren aller Vehicles auf der Karte	39
28	Polyline Segmente	40
29	Server / Client Relation	43
30	Die Karte mit angepasstem Style: Mapbox-Dark	44
31	Vehicle Status Anzeige	45
32	Zeitstrahl-Komponente	45
33	Wechseln zwischen verschiedenen Kartendarstellung	46
34	Verschiedene Auswahlmöglichkeiten der Kartendarstellung	46
35	Anzeigen von Trip-Informationen	46
36	Filter-Funktion	47
37	Linien Finder	47
38	Anzahl an aktiven Trips zwischen 3.00 und 24.00 Uhr am 02.08.2017	49

Tabellenverzeichnis

1	Unterschiede innerhalb GTFS	14
2	Station-Matching Vergleich Old / New	27
3	Tabellengröße vor und nach der Anwendung von gftstidy	34
4	Tabellen Metriken	34
5	Evaluierung der Denormalisierung	36
6	Backend Evaluation	49

Listingverzeichnis

1	Auszug der ersten Zeilen von <code>stops.txt</code>	13
2	Auszug eines GTFS-realtime Trip Updates von MBTA	17
3	Auszug eines GTFS-realtime Vehicle-Position Updates von MBTA	17
4	Antwort des Servers zur Anfrage <code>/daily</code>	41
5	Trip Objekt	41
6	Postgresql Datenbankabfrage von allen aktiven Trips mit ihrem dazugehörigen Linienverlauf	53
7	Erstellen einer denormalized_shapes Tabelle	55
8	Erstellen der denormalized_shapes Tabelle	56
9	Abfrage von Trips und dessen Polyline in einer Zeitspanne XY zu einem gegebenen Datum Z	57
10	Geojson FeatureCollection als Antwort zu <code>/trips/:from,:to</code>	59
11	Shape Feature	59
12	Station Feature	59
13	Station Matching	60

Algorithmenverzeichnis

1	Animate Vehicle	38
---	---------------------------	----

Literatur

- [1] Patrick Brosi. „Real-Time Movement Visualization of Public Transit Data“. Master-Thesis. Universität Freiburg, März 2014 (siehe S. 8).
- [2] Design Council. „A study of the design process. Eleven lessons: managing design in eleven global brands“. In: (Nov. 2016). URL: [http://www.designcouncil.org.uk/sites/default/files/asset/document/ElevenLessons_Design_Council%20\(2\).pdf](http://www.designcouncil.org.uk/sites/default/files/asset/document/ElevenLessons_Design_Council%20(2).pdf) (besucht am 13.09.2017) (siehe S. 6).

- [3] Interaction Design. *Personas – Why and How You Should Use Them*. 2017. URL: <https://www.interaction-design.org/literature/article/personas-why-and-how-you-should-use-them> (siehe S. 10).
- [4] db-engines. *DB-Engines Ranking of Relational DBMS*. 2017. URL: <https://db-engines.com/en/ranking/relational+dbms> (besucht am 28.08.2017) (siehe S. 19).
- [5] HaCon GmbH. *Busradar-App zeigt DB-Busse in Echtzeit*. 2014. URL: <http://bit.ly/2wSDphU> (siehe S. 11).
- [6] Google. *GTFS Reference*. 2016. URL: <https://developers.google.com/transit/gtfs/reference/> (besucht am 10.08.2017) (siehe S. 16).
- [7] Stefan Kaufmann. „Opening Public Transit Data in Germany“. Master-Thesis. Universität Ulm, 2014 (siehe S. 11).
- [8] Andreas Lorer. „Web Performance für den mobilen Endanwender“. Deutsch. Bachelor-Thesis. Hochschule Ravensburg-Weingarten, 2015. DOI: 10.13140/RG.2.2.25935.28327. URL: https://www.researchgate.net/publication/313428046_Web_Performance_fuer_den_mobilen_Endanwender (siehe S. 15).
- [9] Massimo De Marchi. *Visualizing Public Transport Systems: State-of-the-Art and Future Challenges*. Techn. Ber. University of Illinois at Chicago, USA, 2015 (siehe S. 8).
- [10] Brian Card Mike Baray. *Visualizing MBTA Data. An interactive exploration of Boston's subway system*. 2014 (siehe S. 8).
- [11] Jakob Nielsen. *Usability Engineering*. 1993. ISBN: 0125184069 (siehe S. 15).
- [12] Wade Roush. *Welcome to Google Transit: How (and Why) the Search Giant is Remapping Public Transportation*. 2012. URL: http://web1.ctaa.org/webmodules/webarticles/articlefiles/Spring_12_DigitalCT_Google_Transit.pdf (besucht am 10.08.2017) (siehe S. 12).
- [13] Seung Kyoon Shin und G. Lawrence Sanders. „Denormalization Strategies for Data Retrieval from Data Warehouses“. In: *Decis. Support Syst.* 42.1 (Okt. 2006), S. 267–282. ISSN: 0167-9236. DOI: 10.1016/j.dss.2004.12.004. URL: <http://dx.doi.org/10.1016/j.dss.2004.12.004> (siehe S. 35).
- [14] Bret Victor. *Up and Down the Ladder of Abstraction*. 2011. URL: <http://worrydream.com/LadderOfAbstraction/> (besucht am 18.08.2017) (siehe S. 4).
- [15] Wikipedia. *Blue Line (MBTA)*. 2017. URL: [https://de.wikipedia.org/wiki/Blue-Line_\(MBTA\)](https://de.wikipedia.org/wiki/Blue-Line_(MBTA)) (siehe S. 14).
- [16] Quentin Zervaas. *The Definitive Guide to GTFS*. 2014 (siehe S. 12).
- [17] Quentin Zervaas. *The Definitive Guide to GTFS-realtime. How to consume and produce real-time public transportation data with the GTFS-rt specification*. 2015 (siehe S. 16, 17).

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben. Die Arbeit wurde bisher, in gleicher oder ähnlicher Form, keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Unterschrift

Ort, Datum