An End-User App for On-Demand Train Service

Nig Rambow

Bachelor's thesis
April 2025

Prof. Reinhard von Hanxleden
Real-Time and Embedded Systems Group
Department of Computer Science
Kiel University

Advised by
Dr.-Ing. Alexander Schulz-Rosengarten

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Weiterhin erkläre ich, dass die digitale Fassung dieser Arbeit, die dem Prüfungsamt per E-Mail zugegangen ist, der vorliegenden schriftlichen Fassung entspricht.

Kiel,

Abstract

Public transport is an important aspect to achieve a sustainable and environmentally friendly future. Specifically rural areas are difficult to connect with traditional public rail transport. Train lines have a high upkeep and drive most of the time empty outside of cities and moments of high demand, such as work traffic.

One way to improve rural train traffic, would be the usage of an on-demand system, that enables users to request a train. Other on-demand services, like Uber, provide their service via an app. Apps have the advantage that they are used on portable devices, which increases the availability to the user. Further more an app can keep the user updated about relevant information and display them for easier access.

This thesis will develop an app, that provides an on-demand train service, presents concepts for the app and communication with other systems, and give insight in the user-flow and use-cases of the app. Additionally, a backend will be implemented to provide an API for the app and provide communication between systems.

After the development the app will be evaluated, by testing the app together with the other projects. For this each student in the project tried the app and gave feedback.

Acknowledgements

First, I would like to thank Prof. Dr. Reinhard von Hanxleden as the Head of the Real-Time and Embedded Systems Group for making it possible to write this thesis.

Further, I would like to thank my advisor Dr.-Ing. Alexander Schulz-Rosengarten for the guidance and feedback he has provided for this thesis.

I would also like to thank Sven Ratjens for granting us access to the railway track Malente Lütjenburg and for his support during the on-site data collection.

Finally, I would like to thank Kevin Ebsen, Lorenz Tiedemann, Simon Jürgensen, and Rasmus Janssen for the good cooperation and joint work in the REAKT Project.

Contents

List of Figures

List of Tables

Acronyms

AMoD	autonomous mobility-on-demand
CLI	Command Line Interface
API	Application Programming Interface
GNSS	Global Navigation Satellite System
JSON	JavaScript Object Notation
ATO	Automatic Train Operation
TMS	Traffic Management Systems
REST-API	Representational State Transfer Application Programming Interface

Chapter 1

Introduction

Trains are a crucial component of modern public transportation, offering a unique advantage by efficiently transporting large numbers of passengers quickly to their destinations. To achieve a comparable efficiency with vehicles like cars or buses, more vehicles and personnel are needed. However, in rural areas where demand for train service is lower, it becomes unprofitable for companies to operate regular train lines, due to high labor and operating costs. In addition to these costs, the train would mostly be empty and waste fuel. As a result, residents often rely on cars for transportation, which presents challenges for those who cannot afford a car or lack access to one. Trains driving empty and more cars also lead to environmental problems, such as increasing carbon emissions, or in case of electric cars, the destruction of habitats by mining rare metals.

In contrast to rural areas, urban areas or inter-city traffic functions well with traditional scheduled transportation systems. In rural areas, however, trains operate less frequently due to lower demand, leading to long gaps between available trains. This reduced availability makes trains less attractive as a transportation option for rural residents.

One example for tracks in rural areas is the track between Malente and Lütjenburg. It is currently used as a tourist attraction providing tourist the option to rent a trolley and explore the old track. Besides that, the track serves as a testing and development environment for the REAKT initiative[1]. The goal of the REAKT initiative is to revitalize these rural tracks with on-demand autonomous train traffic. A concept that is currently developed at Kiel University in the context of the REAKTOR initiative is the "Single-Track Transfer Traffic" as seen in Figure 1.1. The idea is that two train cars on one track connect with each other to allow passenger to switch the vehicle to continue to their destination. After the passenger transfer concludes, both vehicles drive back in the direction they came from.

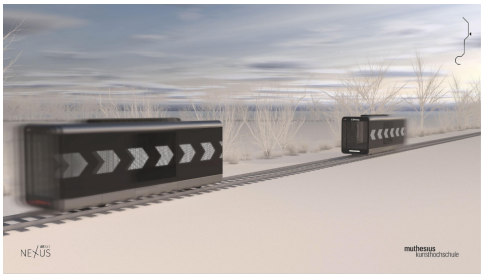


Figure 1.1. Single-Track Transfer Traffic (©Muthesius Kunsthochschule)

The goal of this concept is to increase availability of trains on rural lines with only one track. A normal train would drive constantly from one end to the other and could only pick up passengers at one location simultaneously. Even an on-demand system, like Uber or taxis already use, would not change this problem, because the train could still only be at one place

[1] https://reakt.sh/

at the time. By splitting the train in multiple vehicles, each vehicle could pick up passengers at different locations. Then each vehicle would drive in the direction of the given destination. When two vehicles meet each other, a passenger transfer happens on the track. Adding an on-demand system to this concept, would reduce fuel consumption, because the vehicles would only drive when they are needed. By using autonomous vehicles the operating costs can be reduced even further, by eliminating labor costs.

1.1 Problem Statement

The work in this thesis contributes to a project, that is part of the REAKT initiative, and includes five other students. The goal of the project is to develop an autonomous on-demand rail vehicle, the REAKTOR.

The goal of this thesis is to create a concept for an app, that provides an on-demand train service for reactivated rural train tracks, then implement and evaluate it. The app should assist users to easily interact with the REAKTOR and use it for comfortable transportation. To help the user navigate to his destination, the app needs to display relevant information, without filling the display with too much information so that the user becomes overwhelmed. For this thesis the scope of the project was limited to the Malente-Lütjenburg railway track.

1.2 Outline

The next chapter will cover related work on the topic of on-demand-traffic. Chapter 3 introduces technologies used in this project and gives a short insight in why these were chosen. Chapter 4 discusses concepts for the on-demand app, what information must be communicated with the backend and what functionalities a train must implement to work with this system. Chapter 5 presents the implementation of the app, the backend, and the client for the train. Chapter 6 evaluates the developed application by discussing the feedback of testers and how the currently provided service compares to a service provided by a fully functional management system. Chapter 7 ends this thesis with a conclusion, identifying areas for future improvements and summarizing important findings.

Chapter 2

Related Work

The concept of on-demand-service in public transport is not new. Projects similar to the REAKT Project are already deployed in some areas. These projects utilize on-demand-traffic in different areas, mostly using cars with drivers like taxis.

2.1 NAHSHUTTLE

NAHSHUTTLE[1] is an on-demand transportation service that is provided by the transport provider NAH.SH[2]. The shuttle currently operates in three service areas in Schleswig-Holstein, REMO in Rendsburg, Smartes DorfSHUTTLE in Amt Süderbrarup and Lüttbus in Amt Mittleres Nordfriesland. Depending on the service area the shuttle can be used at different times. NAH.SH provides the service with a mobile app.

Users can download the app and select a service area, next the user chooses a time, start and destination. As seen in Figure 2.1 start and destination can either be selected by providing an address or by moving a pin over a map. The app searches for physical and virtual stops near the selected locations and provides, if possible, multiple connections. In a seminar at the faculty for economics and social sciences at the university Hamburg, a group of students analyzed different aspects of the REMO project [Dül]. Dülmen and Manderscheid conclude in some recommendations on how to improve the project. One important recommendation is the limited service area and time, which limits the usage for users who want to travel into or out of the area.

The NAH.SH app provides a similar service in a different area, that helps with finding solutions for similar problems, such as how the input of locations functions. Solutions specific to on-demand car traffic are less relevant, but can be helpful with understanding why specific solutions where choosen.

2.2 HVV hop

HVV hop[3], previously known as ioki Hamburg, provides an on-demand service in Hamburg. The idea behind HVV hop is to provide a transportation opportunity for the last/first mile, that means it is mainly used in city districts where the coverage of bus and train is not

[1] https://www.nahshuttle.sh/
[2] https://www.nah.sh/
[3] https://ioki.com/project/hvv-hop/

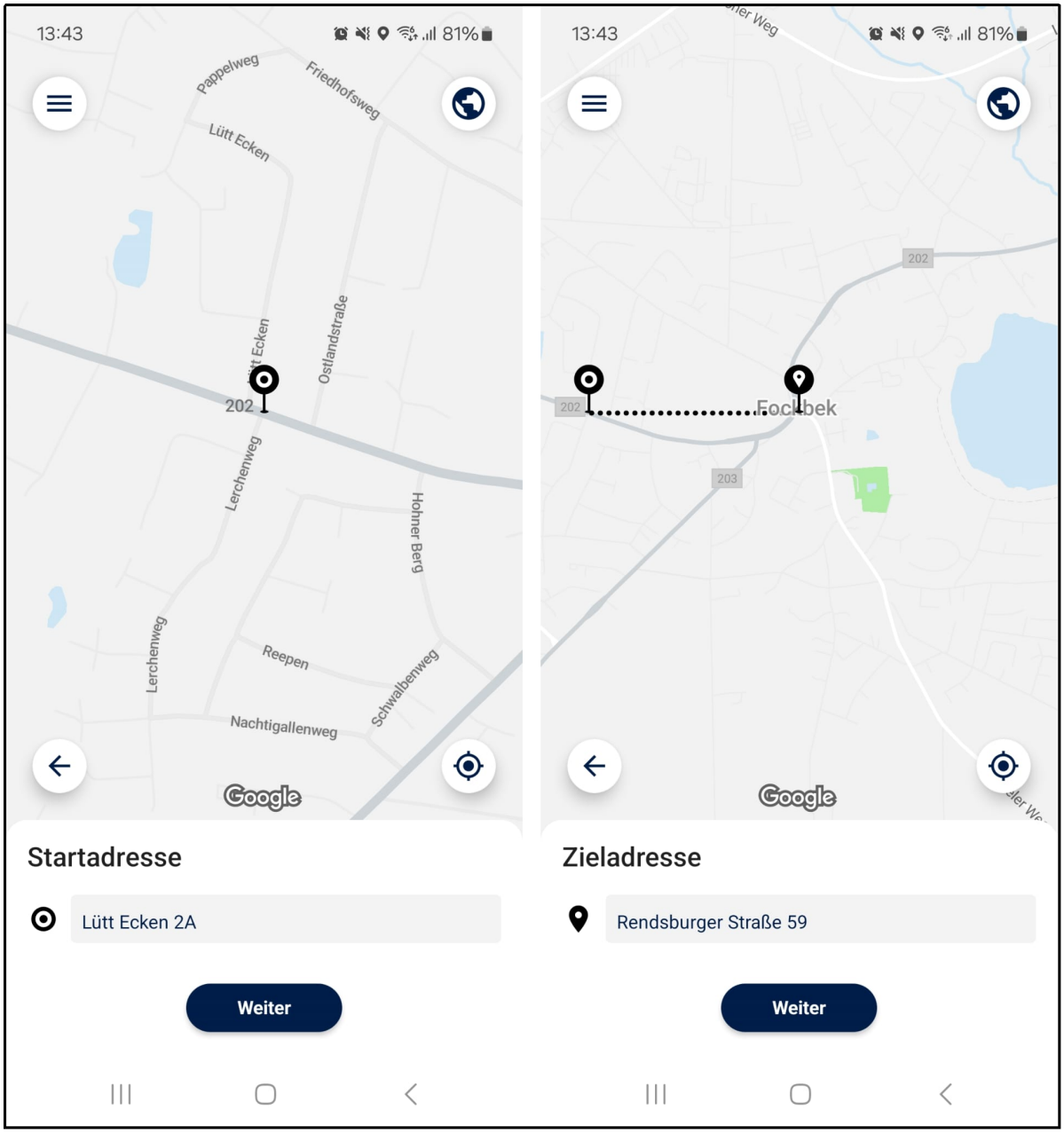


Figure 2.1. NAHSHUTTLE start and destination selection screens.

optimal and residents need to take long walks to get to a bus or train station. Residents can request a pickup from a desired address and travel to bus, train or shuttle stop, or the other way around. Diebold et al. from the TU Hamburg analyzed ioki Hamburg from 2018 to 2020 [DCG21]. Passengers were able to fill out a survey and answer questions about who they are and what the reason is why they use ioki Hamburg. Based on this survey the main reason for using the new on-demand traffic were time-saving, more comfort and a better connection, compared to the local transport. In 2025 the first autonomous vehicles will be added to the fleet of HVV hop and will drive in Hamburg Harburg. Planned is to add 20 of these vehicles to the fleet. These vehicles will still have a driver for security reasons.

Diebold et al. show how an on-demand service improve user experience and usage of public transport. The REAKT Project tries to achieve similar goals. To do so it is helpful to analyze the improvements for public transport, which HVV hop achieved and how their solutions can be implemented in the REAKT Project.

Both examples show that an on-demand service can improve the service quality of transportation in rural areas, as well as in cities with areas, that are not well-connected to the network. Particularly the NAHSHUTTLE shows that, with on-demand concepts, it is possible to provide good transportation in rural areas, without the need for expensive infrastructure and high operating costs.

2.3 Swiss on-demand train lines

In rural regions, low population density and long travel distances to urban areas make mobility a critical factor for economic and social well-being. Due to their flexibility, private owned cars are the primary mode of transportation, but they are not affordable for everyone. Public transportation services, such as buses and trains, provide alternatives to the car, but face challenges due to infrequent schedules, low demand and high operational costs compared top public transport in city areas. These challenges lead to the requirement of government subsidies for rural areas to make the operation viable. A study at the ETH Zurich [SRH+20] evaluated the potential of mobility-on-demand systems, including conventional and autonomous mobility-on-demand (AMoD), to replace existing rural public transport lines. The study analyzed four Swiss rural train lines, three showed that AMoD systems could provide better service at lower costs. Especially in rural areas the implementation of AMoD systems could lead to economic benefits and better service quality.

2.4 Uber

Uber[4] is a widespread on-demand transportation service from the US. In contrast to a normal taxi company, anybody who wants to drive for Uber can apply for a license and transport passengers with their personal vehicle. The Uber app currently provides two different services,

[4]https://www.uber.com/

Uber Eats is a food delivery service, while the main usage is an on-demand transportation service. Presented in Figure 2.2 are screenshots from the Uber app. The user can select a pickup location, via address input or with a pin on a map. To help the user, Uber uses the location of the user as a default pick up location. Then the app switches to the destination screen, where the user can pick a destination out of a suggestion list or provide a specific destination via pin marker on a map or by providing the address of the destination. The app supports the user through the process of requesting a vehicle, by defaulting to the user's location as a start location, the user can most of the time skip the location selection, which reduces the time that is needed to request a vehicle. By providing a list with popular destinations, the user can quickly pick one of these. This further reduces the time that is spent requesting a vehicle. When a specific destination is required, the map helps by selecting the destination in an intuitive way.

By analyzing how the Uber app interacts with the user and assist them in requesting vehicles, important features can be found that are helpful for user and improve user experience. These features can then be implemented to assist users using the REAKTOR.

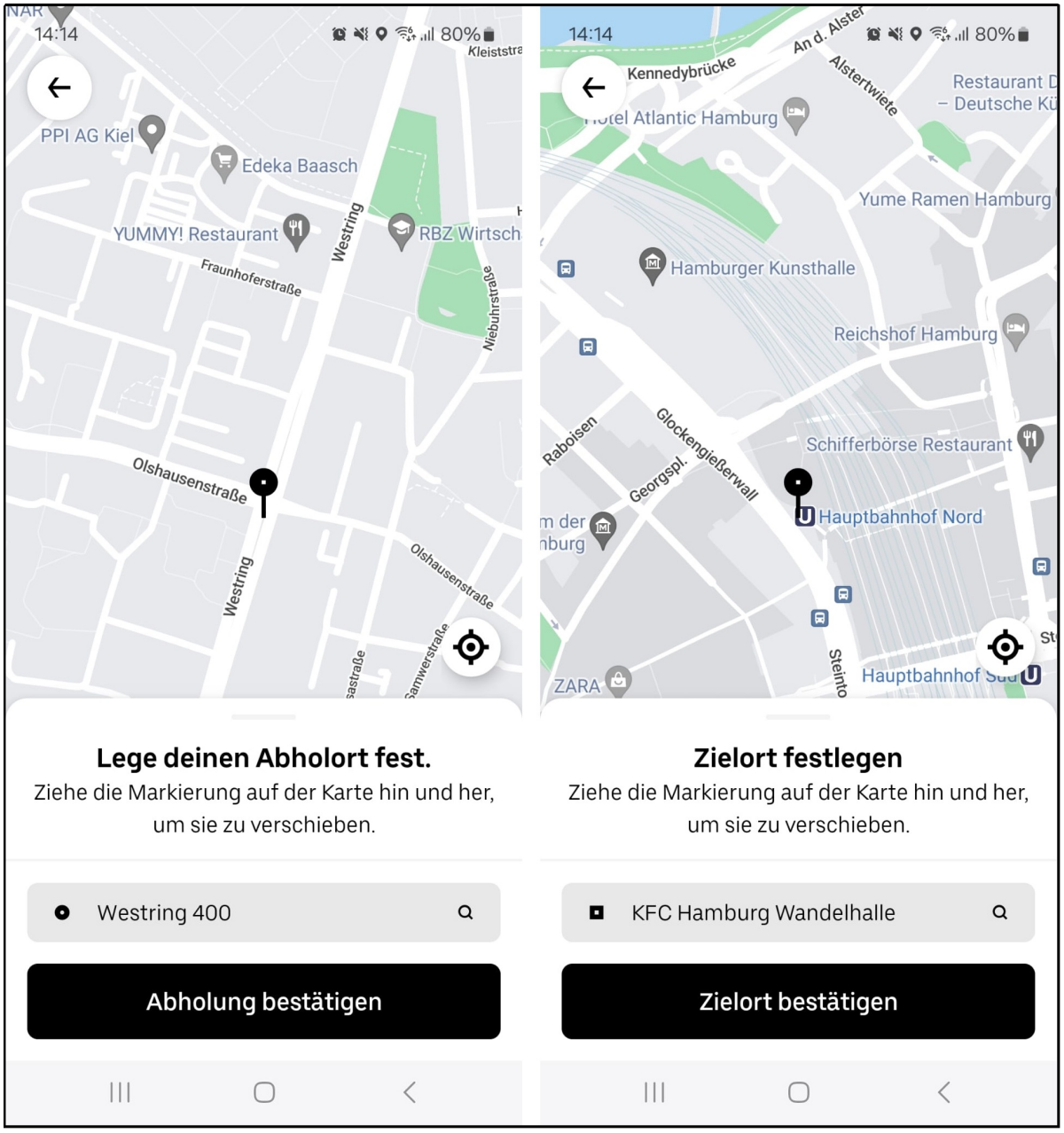


Figure 2.2. Uber start and destination selection.

Chapter 3

Technologies

This chapter introduces technologies used in the development process of the app. The technologies were chosen to help with the implementation and later simplify maintenance by using industry standard frameworks. The following sections will describe each technology and list some advantages, divided by the area they were used in.

3.1 Backend

The backend should provide required functionalities like providing an Application Programming Interface (API) for the frontend, managing multiple requests simultaneously, sending orders to the autonomous vehicles and, in the future, functioning as an API to a database. Based on these requirements the following technologies were chosen.

3.1.1 Python

Python[1] is a programming language that is widely used and comes with numerous well maintained libraries, that improves the development process and helps to implement parts of projects. The large community around Python is filled with guides and solutions for specific problems, which is helpful while developing such a project. The compatibility of Python with different operating systems allows easy deployment on different kinds of systems, either direct on a server or wrapped in a docker container.

3.1.2 FastAPI

FastAPI[2] is a fast and lightweight web framework for API development in Python. The usage of async/await allows faster processing of requests and makes it easier to scale the system for more requests. Django and Flask are frameworks for different use-cases. Flask is used for small API's with a small overhead and Django is used for full web applications. Both frameworks come with a large overhead compared to FastAPI, for the goal to create a lightweight API.

FastAPI comes with an automatically generated documentation, that can be used to debug and test the API interface, while other frameworks do not have this feature and need additional tools to provide this functionality.

[1] https://www.python.org/
[2] https://fastapi.tiangolo.com/

3.1.3 Pydantic

Pydantic[3] is a core library for FastAPI. It is used to automatically validate input data and makes the API type secure. That allows a typified communication and prevents requests that are not sending data in a specific form. The functionality to parse JavaScript Object Notation (JSON) data into types that can be used by the system and serialize these types back into JSON data is ideal for APIs. Pydantic gives detailed error messages when an input is invalid, which helps with debugging the API in development. The types that are generated with Pydantic also help with communication with databases, by validating the data that gets sent to the database. In combination with SQLAlchemy[4], Pydantic types can be used to define the schema of database models, allowing easy storage of data without the need to validate it repeatedly.

3.1.4 JSON Schema

JSON Schema[5] is JSON-format for exchanging data. It brings the vocabulary that enables consistency, validity, and interoperability for communication with JSON data. Using JSON Schema it enables the developer to create a common language for data exchange. By creating precise rules for types, these schemas can be used to create types in different languages and systems. This enables a save communication between these systems and prevents type mismatches.

3.2 Frontend

The frontend will be a mobile app, that an end-user uses on his phone. To allow the end-user to request a vehicle on-demand, the app needs a way to get data about the request from the end-user. The collected data must then be processed and send to the backend. After the backend processing is complete, the app receives data about the requested vehicle and is required to display the given information for the end-user.

3.2.1 TypeScript

TypeScript[6] is a language that is build on JavaScript, it adds static types to JavaScript, which prevents runtime errors and helps with detecting errors in the development process. TypeScript compiles into JavaScript code, that allows the usage of TypeScript's improvements without losing compatibility. The strong type system improves the development process in IDEs by providing extensive auto-completion and error highlighting.

[3] https://pydantic.dev/
[4] https://www.sqlalchemy.org/
[5] https://json-schema.org/
[6] https://www.typescriptlang.org/

3.2.2 Angular

Angular[7] is a framework for the development of modern web applications. It is fully built with TypeScript, that is why these two work really well together. A clear project structure, which is based on a modular system, helps with working as a team on a project and keeping an overview. Two-way-binding is a helpful feature that updates parts of the UI without updating the site. It works with variables that are bound to values in the UI, when the variable changes, it propagates the new value to the UI and updates it. This helps with the development and makes it easy to dynamically change the UI. Angular comes with a powerful Command Line Interface (CLI). It allows the developer to easily create new components via simple commands.

3.2.3 Capacitor

Capacitor[8] is a multi-platform runtime- and plugin system, that allows the conversion of web-apps into native Android, iOS and desktop apps. It works by running a web-project in a native environment and translates CLI calls to native device CLIs, like camera, Global Navigation Satellite System (GNSS), file system and geolocation.

3.2.4 MapLibre

The MapLibre Organization[9] manages multiple, useful open-source mapping libraries. MapLibre-GL-JS is used to display maps on websites. It runs with a good performance due to GPU-accelerated vector tile rendering. MapLibre Native is a library for mobile applications, desktop applications and embedded systems.

[7] https://angular.dev/
[8] https://capacitorjs.com/
[9] https://maplibre.org/

Chapter 4

Concept

The concept of on demand services in public transportation is not new. Many small and large companies provide on demand transportation in different areas. Some even started to use autonomous vehicles to transport passengers, still accompanied by a driver, who can intervene, as mentioned in 2.2. Or on a larger scale as Pavone et al. discusses in [Pav15]. Pavone et al. concludes that an autonomous mobility-on-demand system, as shown in case studies of New York City and Singapore, would be more affordable and convenient compared to traditional mobility systems.

This chapter provides a concept of an app, that applies mobility-on-demand to autonomous train transportation and discusses what use cases the app has and what features were used to provide these use cases.

4.1 Overview of this Project

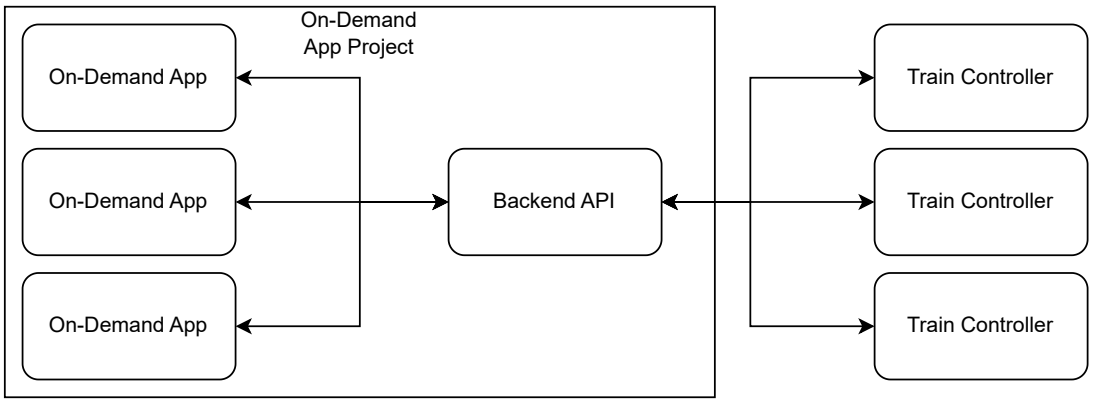


Figure 4.1. Systems in this Project.

Figure 4.1 shows three distinct parts, that are relevant for the app to function. On the left are three boxes labeled with On-Demand App, these boxes model devices that use the app. Each device connects with a server to communicate with the backend API. The box in the center, labeled with Backend API, models a server, that runs an API, which provides necessary features to the app and functions as a manager for trains that connect to it. On the right are

three boxes, labeled Train Controller. These emulate trains, that connect to the backend. These trains then are used to provide the transportation service to users.

To ensure that the app can provide its service, a connection to the backend is needed. For that reason, the concept follows a client server structure, where the backend runs on a separate server and does not run on the trains. Another possibility would be to run the backend on the train, thus providing a stable connection between train and backend. If the deployment area does not provide a good and stable internet connection, the app can not connect to the backend and provide its service. Deploying the backend on a separate server allows a stable connection for the app and enables the backend to provide a service even when the train lost connection.

4.2 Use Cases

To provide the described service, a requirement analysis was conducted, in which use cases the app needs to provide its service. Different use cases emitted during the analysis.

Location Input

A user must be able to input a desired location for the start and destination. This can take place in different ways. One possibility is the direct input of the name of a station, that allows the user to freely decide where to start. For those unfamiliar with the track, a feature is needed to assist them with the decision. Another option would be using location data to select a position. The way to do that, must be adapted to what the user wants to select, for example, the user can not use their current position to select their destination, because usually a user wants to travel away from their current location.

Request Transportation

To request a train to the desired location and travel with it, the app needs to send the collected data, about start and destination, to the backend and request updates about the status of the current transportation request. Additionally, the app must be able to receive these updates and display the provided information to the user.

Navigation

Users, that do not know the area around the track, might not know where they are and where their starting location is, just based on the name of the station. For that a map can be used to display the position of the user, train and the selected starting location. The user then can navigate to the starting location and see where the train is currently on the track. After entering the vehicle, the map can provide the next destination and the location of the train. The visual representation can help the user to estimate how long their journey will take.

ETA

After a user requested a vehicle, a timer should display the estimated time of arrival. In combination with the map the user can see how long the vehicle will take to arrive and where it is at the moment. This timer should be updated regularly to adjust to changes on the track and vehicle speed.

4.3 Structure

This section describes the structure of frontend, backend and train client, and how these parts communicate with each other. Figure 4.1 shows the overall structure of the app, backend and train. The following figures show each component in closer detail.

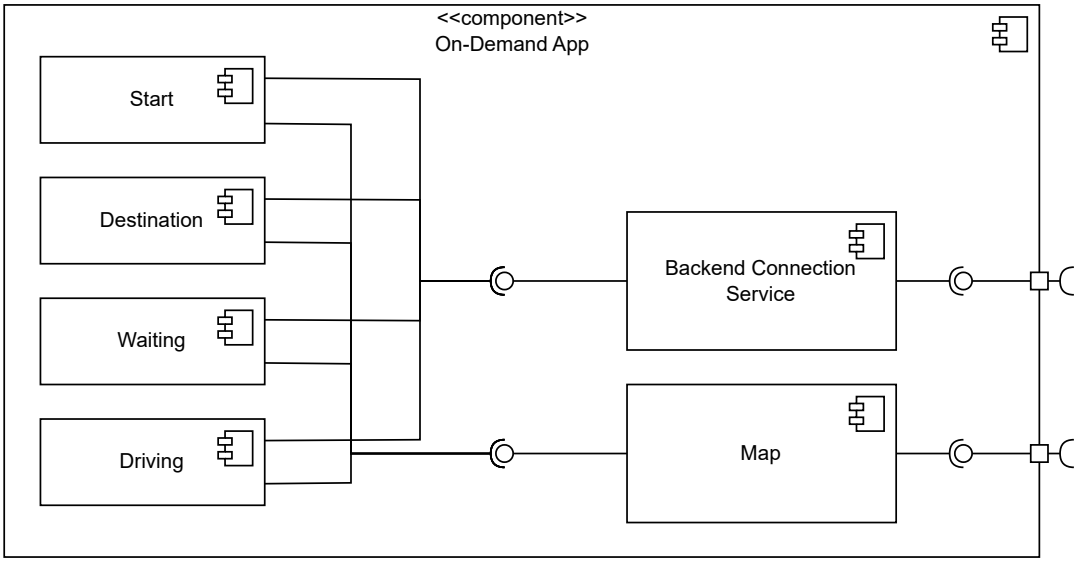


Figure 4.2. Component diagram of the app.

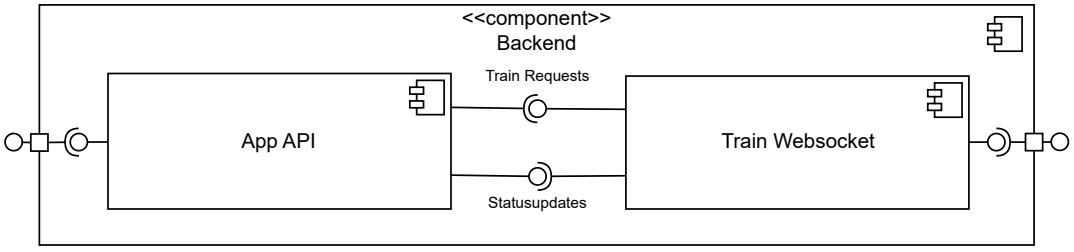


Figure 4.3. Component diagram of the backend server.

4.3.1 Components

Figure 4.2 presents a component diagram of the on-demand app. On the left are four components each representing one page of the app. On the right is one component called the Backend Connection Service, its function is to send and receive data to the backend and store it for later usage. Below that is the Map component. Each page displays a map, that is generated and maintained by this component.

Separating the UI components from the logic components helps with keeping the code easier to read and increases maintainability. Additionally, it enables future developers to easily change UI components without the need to rewrite the logic for the app. To add new features, future developers can add new components and build on the already implemented logic.

Figure 4.3 presents a component diagram of the backend. Within the backend are two components, the App API component provides the API for the on-demand app and processes all incoming requests. The Train Websocket component provides an endpoint for trains to connect with the system. When the App API receives a request, a message is sent to the Train Websocket. The message then gets forwarded to the desired train and the response is returned to the API. The API processes the response and sends a response back to the app.

Separating the communication with the frontend and the trains, enables easy exchange of both components. The communication with the train can easily be modified, without the need to rewrite the API. On the other hand the API can be easily modified to add new features or change already implemented ones.

4.3.2 Communication

For a train to connect to the websocket, the train must implement a client, that has the required functionalities. The client needs to be able to give the train controller a new task and get status updates about its speed and position on the track.

A websocket connection was chosen to allow bidirectional communication between backend server and train client. At the moment only the backend sends request to the train and the train answers them. In the future, when multiple trains need to be managed, communication initiated by trains can be used to start communication with other trains in the docking phase of the single track transfer traffic. A Representational State Transfer Application Programming Interface (REST-API), running on the train, would only allow communication in one direction and blur the server-client structure, by giving the backend server characteristics of a server and a client.

After a user has selected a start and destination, the data is sent to the backend API. Next the backend creates a new job based on the selected locations. This job is stored in the backend and contains every information that is needed about this job. Then the backend sends the selected start location as a new job to the train. These jobs are different from the jobs that are used in the communication between frontend and backend. The train only receives the next location, where a passenger wants to enter or leave the vehicle. Splitting jobs in start

and destination for the trains is needed for a management system, that can manage multiple trains and passenger in parallel. This enables the management system to pick other passenger up while driving a user to its selected destination, by connecting different starting and end locations to one big tour.

The train must be able to receive update requests and answer them with its speed, position data, and if it already arrived at its current destination. The backend then sends the data back to the app to inform the user about the changes. When the train arrives at its destination the user can enter the vehicle. The user then notifies the backend after entering the vehicle and the management system then decides where the next destination for the train is. After the train arrives at the selected destination, the user exits the vehicle and notifies the backend about it. Lastly the management system selects a new destination for the train or if no jobs are available lets the train wait.

4.4 Summary

The concept contains three parts, on-demand app, backend server, and train client, all three together provide the desired service, that enables users to request trains and drive with them to desired location. The app enables the user to request transportation and displays relevant information about the transport. The backend provides required functionalities to enable the apps service and works as a man in the middle, which processes requests and manages information for the app and train. The train client provides an interface for trains to connect with the backend server and receive requests from it, to drive to desired locations and pick up or drop off passengers.

Chapter 5

Implementation

This chapter describes how the frontend implemented the previous described use cases and how the backend provides its functionalities for the app and the train.

5.1 User Flow

Figure 5.1 shows a diagram of the user flow through the app. When the user opens the application, it first checks if a job ID is stored. If a job ID is stored, then an update on this job is requested from the backend API and, in correspondence to the answer, the app loads the waiting or driving screen (Figure 5.2c and 5.2e). If no job ID is stored, then the app loads the start page (Figure 5.2a).

5.1.1 Start and Destination Selection

On the start page, the user gets asked to allow geolocation service for this app, which is needed to locate the user and place a marker on the map accordingly. Next the user gets to choose, if they want to use their GPS location and let the system find the closest point on the track, or if they want to select a station out of a predefined list, which is available in a dropdown menu. In the center of the screen, the user can interact with a map, that is centered on the user's location. The GPS button at the bottom of the screen enables the selection via GNSS and places a marker on the user's location. To leave the start selection and move to the next screen, the user needs to press the "Select Location" button, which is covered by the dropdown menu in 5.2a. After the button is pressed, the app stores the selected option and opens the next page.

The destination selection (Figure 5.2b) works similarly to the previous page. Once again the user can select a station out of a dropdown menu. In contrast to the GPS option, here the user can move a red flag over the map and place it at a desired location. To get a location on the track as a destination, the location of the flag gets sent to the backend and the nearest location on track will be calculated, then this location becomes the destination. After the user pressed the "Select Location" button, the app stores the selected option and sends start and destination to the backend.

The backend creates a new job out of the data and returns a job, which contains an ID, to identify the job and request updates, the selected start and destination, whether the job is currently worked on, an estimated arrival time and whether the train has already picked up the user.

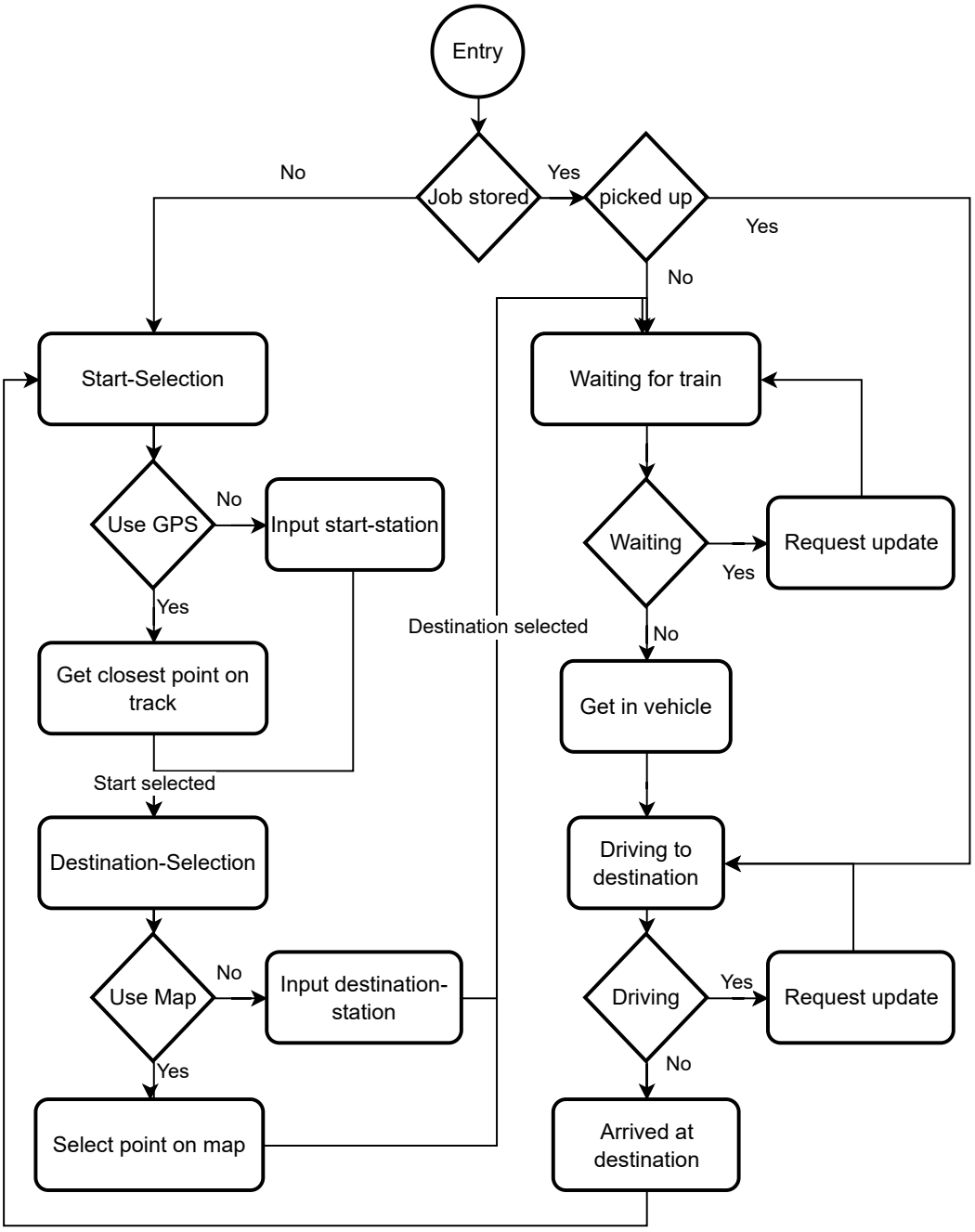
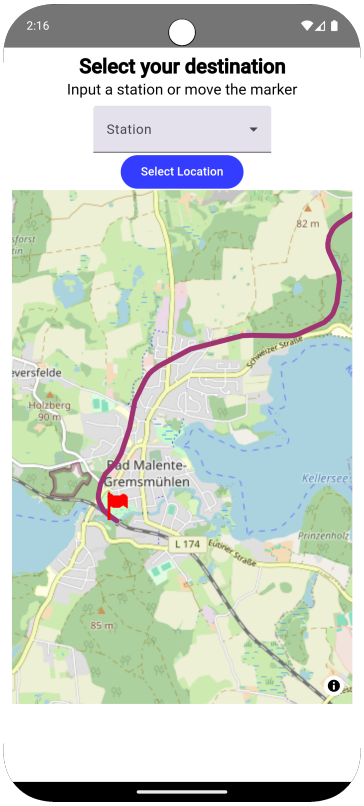
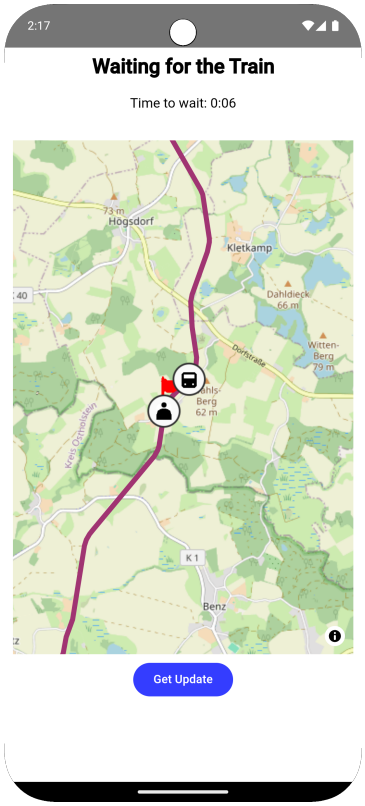


Figure 5.1. User Flow diagram of the app.

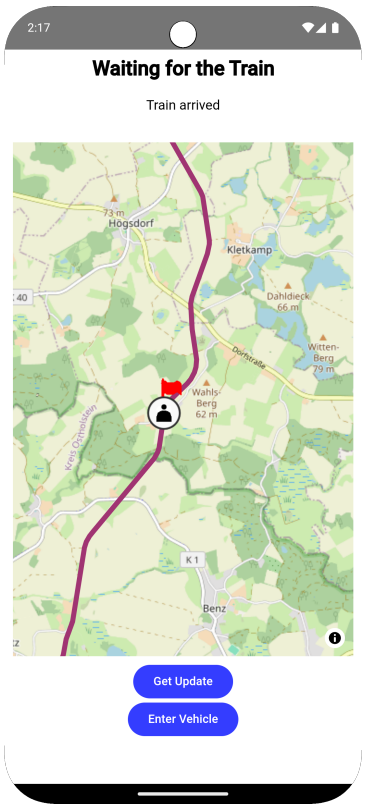
(a) Start Selection.



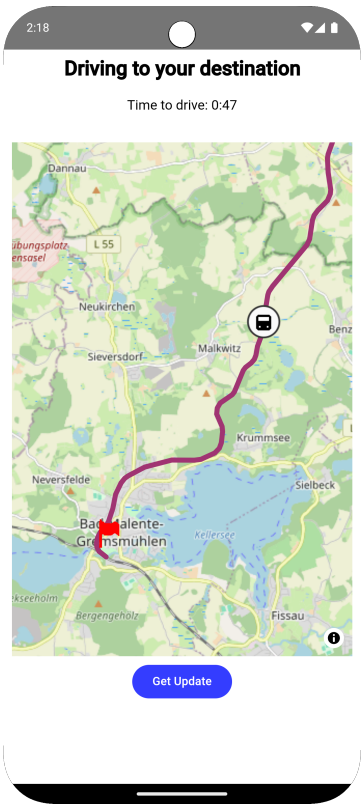
(b) Destination Selection.



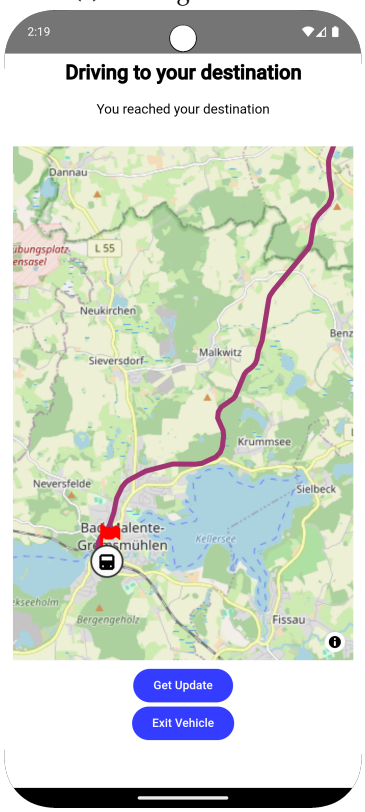
(c) Waiting for train.



(d) Train arrived at start.



(e) Driving with Train.



(f) Train arrived at destination.

Figure 5.2. Screenshots of the app.

These two pages implement the first and second use case, they enable the user to input two desired locations to request a transport and the inserted data is sent to the backend to request a new job.

5.1.2 Waiting and Driving

After the app received an answer from the backend, the waiting screen (Figure 5.2c) opens. On this screen a timer, a map, and a button are being displayed. The timer shows when the train arrives at the starting location. The map displays three markers, one at the user's location, one for the train, that moves along the track and a flag, that marks the pickup point, where the user can enter the vehicle after both have arrived. With the "Get Update" button at the bottom, the user can actively request updates from the backend. After the train arrives (Figure 5.2d) at the pickup point, the timer changes to a message, that signals that the train arrived. At the bottom of the screen a button appears, that, after being pressed by the user, signalizes the backend, that the user has entered the vehicle and the train can move on to its next destination.

Here would a management system take over and check where the train should drive next to pick up or deliver other passengers, or if that is not necessary at the moment, task the train with driving the passenger to the desired destination. The management system is not implemented in this thesis and should be implemented in the future. How the current implementation functions compared to a management system, that implemented the single track transfer traffic, is discussed in Chapter 6.

Next a new screen appears (Figure 5.2e), which is shown, when the user has entered the vehicle and drives to their destination. This screen is very similar to the previous screen. The two relevant differences are, that the map only shows markers for the destination and the train vehicle, while no longer showing the users marker. After the train arrives at its destination, the button, at the bottom (Figure 5.2f), signalizes the backend that the user has left the vehicle and that the train can accept new jobs.

The waiting and driving pages implement the last two use cases. The map assists the user on their way to the train, by providing an overview of the surrounding area, also the location of the track, the train, and the user. The timer provides the user with an estimated time of arrival, when the train will arrive at its next location, helping the user to estimate when they will arrive or when they need to be at the starting location.

5.2 Backend API

To enable the app to provide its functionalities, a backend API is needed to help with calculations and communication with other systems. This API is implemented in Python using FastAPI and Pydantic.

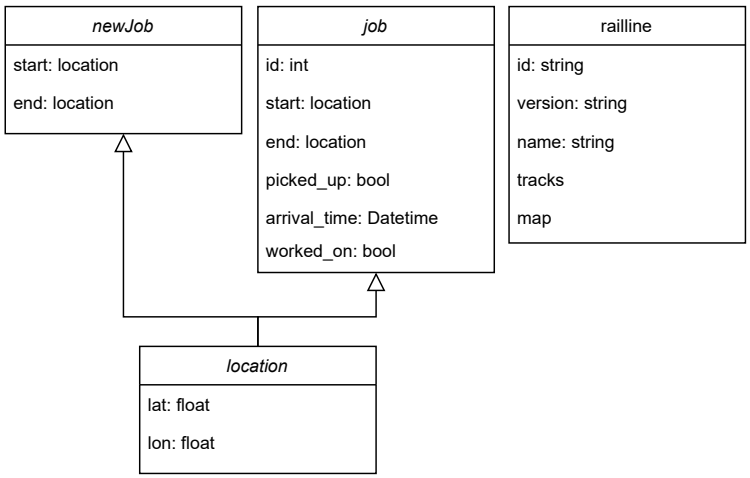


Figure 5.3. Class diagram for communication.

5.2.1 Communication

To ensure a type secure communication, JSON schema is used to create classes for the backend and frontend. These classes are described in Figure 5.3.

The class railline describes how a rail line is modeled. It contains an ID, a version, a name, tracks, and a map with settings. ID, version and name are self-explanatory, tracks stores an array filled all tracks on a line, each track contains geojson data of the track. Map stores the start configuration for the representation of the map, it contains latitude and longitude for the center of the map, and a value for the zoom.

The class location represents a GNSS location for the position data of the train, start, and destination. It contains two numbers representing latitude and longitude.

The class newJob is used when the app sends a new request for transportation. It contains the start and end location of the selected stations or locations. The backend uses this data to generate a new job that stores all the needed data.

The class job represents a complete job and will be generated from newJob. It contains an ID for identification, the start and end location, a flag that stores whether the user has entered the vehicle, a point in time when the train arrives at the next location and a flag that stores if the job is currently being worked on by a train or not.

5.2.2 API

The API provides its service with some routes. Each route is used for a different task, that is required for the app to provide its service. This section describes these routes and gives a description on what they are used for.

Get Stations

This route is used to transfer a list of stations to the app. The list is requested when the app opens and stored in memory. It contains all stations on the track Malente-Lütjenburg with IDs, names, and location data. The app then utilizes the location data, when the user selects a station out of the list, and uses it to request a new job.

New Job

The app sends a request containing a previously described newJob object, the backend generates a job object and stores it. Next, the server sends the current job to the train using a websocket connection. This connection is described in detail later. After the train answered with location and velocity data, an estimated time of arrival is calculated, and the job object is sent back to the app.

Get Update

To request updates about a specific job, the app sends a request with a job ID. If the ID is valid and matches a stored job, the backend requests location and velocity data from the train to calculate the estimated time of arrival and sends the update back to the app.

Pick up

This route is used when the user presses the "Enter Vehicle" button in the app. It signalizes the backend that the user has entered the vehicle and that the vehicle can move to the next location. The backend sends a new destination to the train and changes the flag in the stored job object.

Find nearest Location

This route calculates the nearest point on the track from a given location. The point is calculated by finding the two closest points on the track and interpolating a point between these two points. This route is needed when the user uses their GNSS location or chooses their destination with the flag marker.

Finish Job

The finish_job route is used, when the user exits the vehicle and presses the "Exit Vehicle" button. By doing so the user signalizes the backend that the job is done, and the train can get a new job if one is available.

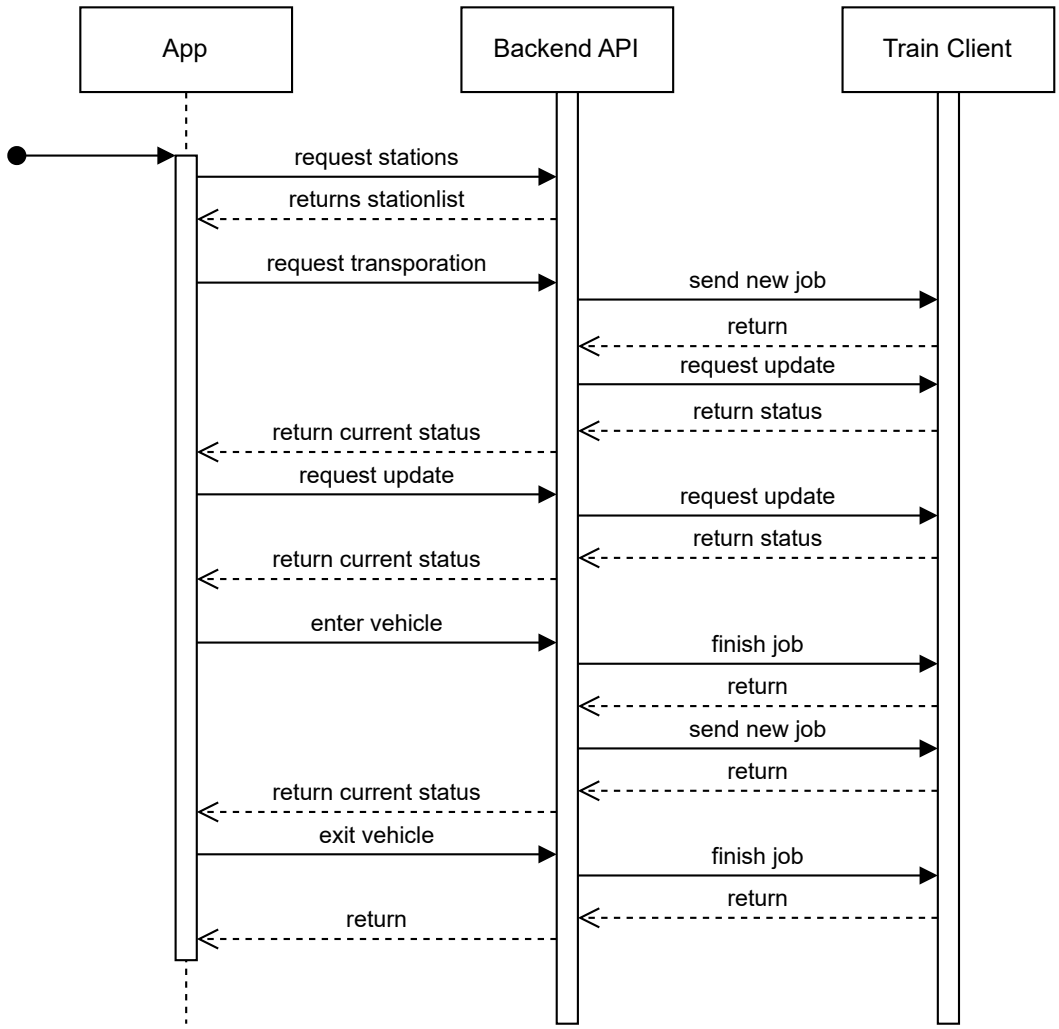


Figure 5.4. Sequence diagram of communication between each system.

Get Train Location

This route is used to get the current location of the train to mark it on the map in the app. The app requests a location update each time it requests an update. This is implemented separately to enable separate communication when only one is needed. This would be the case when the job is currently not worked on, but the map should still update.

5.2.3 Websocket Connection with Train

For the API to process requests, that require updates from a train, an option for trains to connect with the backend is needed. These trains must implement a client that assist with the communication and wraps required data in predefined patterns. Figure 5.4 displays the communication between app, backend and train, that is happening during an active job. There are four message patterns that are needed for the communication. Firstly, `newJob` is used to send a new job to the train, containing a job ID and a destination translated into track kilometer. Secondly, `update` request the train to send the track kilometer, speed and if it arrived at the current destination. Based on that the backend can then process an update for the app. The `position` request is used to get the current position of the train, which is then used to mark the location on the map. Requesting the location in two different ways is required to get the location when the train either works on a task or has no current task. To signalize the train that the current job has finished, the user presses a button when they enter or leave the vehicle, thus sending a `finish` request After the task is finished the train is free for other tasks.

Chapter 6

Evaluation

This chapter evaluates the feedback that I collected by testing the app my with fellow students. Additionally, this chapter evaluates, how the transportation works with the current implementation and how it should work with a proper implementation for managing passengers and trains.

6.1 Feedback Evaluation

The feedback was collected one month before the end of this thesis, in a project test where all projects, that are currently part of the REAKT Project, were connected and tested together. Three students with an Android device installed the app and did some test runs. After that each student answered some questions and described their experience. The questions were:

- How useful was the displayed information?
- Was the app intuitive to use?
- Which features were missing?
- Other comments?

These questions were asked to find areas where improvements could be made and to evaluate whether the app fulfilled its task in assisting users with the interaction with the REAKTOR. The collected feedback is not representative for the performance of the app, because the number of participants is too low to represent future user groups and all three testers already had experience with the app, which leads to the problem, that they know how to use the app, while a real user does not know that.

6.1.1 Usefulness of Displayed Information

All testers stated, that the displayed information was useful for the service the app provides. But it was commented that map and timer are sometimes inconsistent, particularly after a new page loaded. The map had problems centering on the relevant marker after loading. The timer was inconsistent when the train was driving at low speed or stopping. A reason for that is found in the calculations for the timer, that are based on the speed of the vehicle. The dropdown menu, containing all predefined stations, was described as useful to people, that do not know the track.

Overall the displayed information is useful to users, but the way how the information is presented should be improved by improving the calculations for the timer to result in a more accurate time at low speed and improve the implementation of the map to display the relevant information faster and more precise.

6.1.2 Intuitive Use

The user flow through the app was described as intuitive and easy to follow. It was noted on the start and destination selection screen looked too similar, leading to confusion about which screen was open and what information should be inserted. The "GPS" button on the start selection screen caused some confusion, because it was not directly clear what its use was, whether it just marked the user on the map or selects the users position as the start location.

The overall user flow of the app is straight forward and easy to understand for users, the confusion caused by the start and destination screen can easily be fixed by, for example, using icons to make it clearer what page is currently open. To help the user understand what buttons do, an explanatory page could be added, that provides a guide to all functionalities that the app provides.

6.1.3 Missing Features

Tester provided valuable suggestion for features that should be added:

- Marking selected stations on the map.
- Automatic centering on the vehicle, after the user entered the vehicle.
- Language selection.
- A visual indication for how many jobs are in front of the user, when the user needs to wait.

Most of these suggestions are implementation fixes. this means they can just be added in the implementation and improve the service that way. The last point is interesting in that regard, because its only relevant to know how many other users are in front of you when the train can only work on one user at the time, which it is the case in the current implementation. A future implementation, that also implements the single track transfer traffic, would not need this feature anymore. Section 6.2 compares how the current implementation works compared to how I think the single track transfer traffic would work in the future.

6.1.4 Other Comments

One problem was, that the app's layout was different on different devices, which led to buttons being out of the screen and the map covering a large part of the screen. The buttons were still usable, but the user had to scroll down to the buttons, which decreases the users experience, if they are not aware of that. Another comment was made about the map zoom.

After the destination selection the map zooms to the location of the user, possibly leading to the issue, that the start location marker is out of the map and the user needs to zoom manually. The commenter noted, that this would help the user to navigate.

Both issues should be fixed before the app will be used by users, but are currently not that big of an issue, because the app still functions properly.

6.2 Managing Trains and Passenger

The goal of the REAKT Project is to create an autonomous train that can provide reliable and convenient transportation on reactivated train tracks. To achieve that, the concept of single-track transfer traffic was created. This thesis focused on the frontend development of the app and implemented a minimal backend to provide required functionalities. The current implementation of the backend does not support single-track transfer traffic and works only with one vehicle that transports one passenger at a time. That leads to the problem, that if multiple user request a transport, the system can only pick one of them up and bring them to their destination, then drive to the next user and repeat the process.

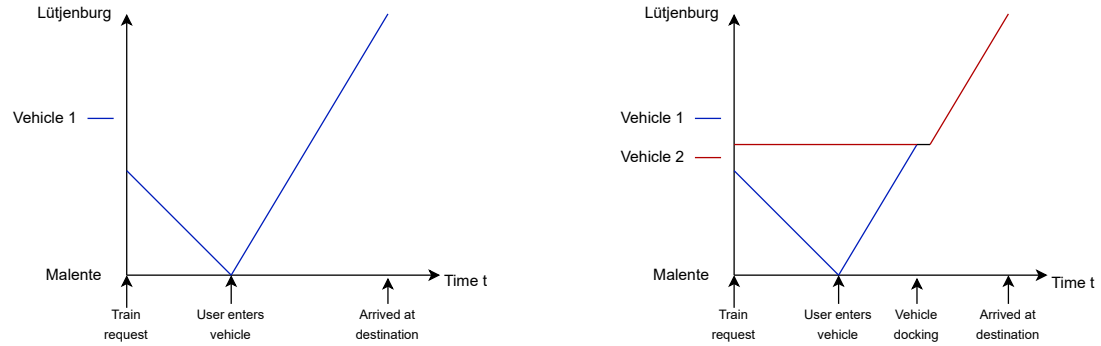
To evaluate the current implementation and compare it to the desired one, the next sections will describe different scenarios from the perspective of a user at the Malente-Lütjenburg test track and compare how the system would work in comparison.

The following scenarios build on the assumption, that two vehicles are deployed on the track and that the vehicles function like the concept of single-track transfer traffic describes them to.

Scenario 1: One Passenger

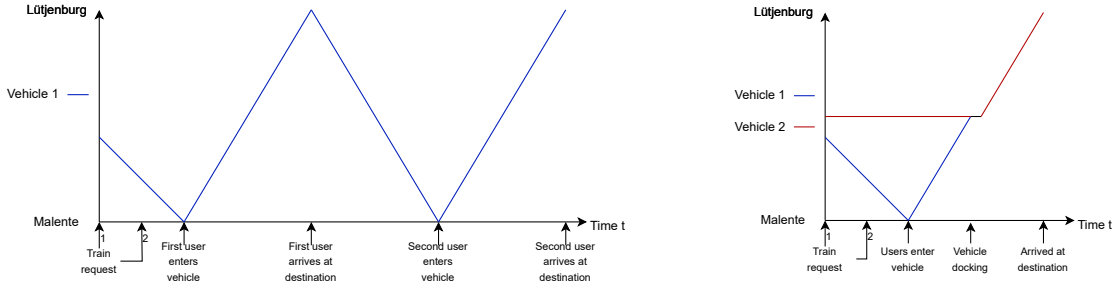
Figure 6.1b shows how a future implementation with single track transfer traffic would work. One person wants to travel from Malente to Lütjenburg. The user orders a transport and the backend selects the closest train to pick up the user. After pickup, the train drives towards Lütjenburg. The second train currently stands at the middle of the track, after some time the first train arrives at the location of the second train and docks with the parked train. The user moves to the second train and drives with it towards Lütjenburg. The first train then waits for a new task and parks at its location. After the arrival in Lütjenburg, the user leaves the vehicle and the train parks until it receives a new task.

The current implementation, shown in figure 6.1a, works similar. The only difference being the docking and moving to the other vehicle, due to the lack of a second train on the track. In scenarios with only one passenger the system already works fine. Even with only one vehicle. The passenger does not have to wait long and arrives as fast as possible at their destination.



(a) Current implementation. **(b)** Future implementation.

Figure 6.1. Path-time-diagram scenario 1.



(a) Current implementation. **(b)** Future implementation.

Figure 6.2. Path-time-diagram scenario 2.

Scenario 2: Two Passengers Same Location

Figure 6.2b shows how a future implementation with single track transfer traffic would work. Two persons want to travel from Malente to Lütjenburg. Both order a transport and the system sends the closest train to their location. In Malente the train picks up both passengers and drives towards Lütjenburg. The second train waits in the middle of the track and after the first train arrived and docked, both passengers can move to the second vehicle. The second vehicle then drives towards Lütjenburg and drops off both passengers.

The current implementation, shown in figure 6.2a, does not work like that. After both passengers have ordered a transport, they are stored with an ID. The backend gets the job with the lowest ID and instructs the train to pick the user up. Then drives the train with only one passenger towards Lütjenburg. After the train has dropped off its passenger in Lütjenburg, the backend finishes the job and deletes it. Next the backend selects the next job with the smallest ID and instructs the train with driving towards Malente. There the train picks up the second passenger and drives back to Lütjenburg to drop the passenger off.

Because of the current limitations of only working on one job at the time and only working with one vehicle, the current implementation does not work as efficiently as desired. The first user gets the best service and has a minimal waiting time, while the second user needs to wait until the first user arrived. The train then moves back to their location, immensely extending their waiting time.

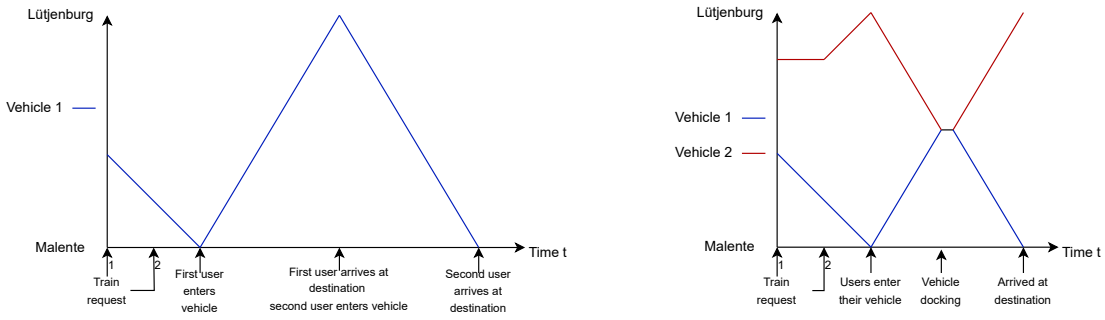
Scenario 3: Two Passengers Different Location

Figure 6.3b shows how a future implementation with single track transfer traffic would work. In this scenario, there are two people who want to travel in the opposite directions. One starts in Malente and wants to drive to Lütjenburg. The other person however wants to travel from Lütjenburg to Malente. After both ordered a transport, the backend sends the closest train to each location. After arrival both trains pick their passenger up and drive towards their destination. In the middle of the track both vehicles meet and dock. The passengers then change to the other train and both drive with their new trains towards their destination.

The current implementation, shown in figure 6.3a, can not deliver the same level of service. After both users have ordered a transport, the backend would prioritize the job that arrived first at the backend, for easier explanation the first job that arrived is the one from Malente to Lütjenburg. The backend tasks the train with driveing to Malente and picking the waiting user up. After pickup the train drives towards Lütjenburg and drops its passenger of. After the backend has marked the job as finished, the job with the lowest ID will be started. The train will be instructed to drive to Lütjenburg and answer that it already is there. Next the waiting user can enter the vehicle and the train drives to Malente to drop its passenger of.

Once again, due to the current limitations of only working on one job at a time and only working with one vehicle, the current implementation does not work as efficiently as desired. The first user gets the best service with minimal waiting time and the second user must wait until the first user has arrived at their destination.

As the described scenarios explain, the management aspect for the backend is not implemented yet and was not part of this thesis. Future works in the REAKT Project need to implement algorithms, that enable the backend to provide the desired service quality.



(a) Current implementation. **(b)** Future implementation.

Figure 6.3. Path-time-diagram scenario 3.

Chapter 7

Conclusion

This thesis has presented the concept and development of an on-demand train service app with the goal of providing an attractive and easy to use service for rural areas. The evaluation has shown the benefits that an on-demand train service app with single-track transfer traffic can provide.

In the future, after some improvements, the app could be used on the railway track Malente-Lütjenburg, to provide an on-demand train service and improve mobility for residents and tourists in the area. For a launch the app and backend need to be further developed to include a functioning backend, that can manage multiple trains and user requests, as well as a data protection policy, that allows users to use the app without the usage of location data. Furthermore, to use the app on different tracks, a dynamic way of providing data about the track must be implemented, such as providing the track data through the backend to improve maintainability.

7.1 Future Work

The current implementation of the app and backend fulfills basic requirements to provide a service, but there is still a lot of room for future work to improve this service. In this section, some possible improvements, are listed below, as inspiration for future developments.

Management System

To improve the availability of trains, multiple trains should be used, as described in 6.2. Multiple trains on one track need to be managed to work efficiently together. For that purpose, a management system must be implemented.

X. Rao et al. discussed in [RMW13] how to combine Traffic Management Systems (TMS) with Automatic Train Operation (ATO). They proposed to combine the rescheduling algorithm of TMS with the multi-objective algorithm of ATO, to reduce costs and improve the performance of railway transportation.

One problem for this system is to manage these trains optimally. For example the question of how a train should move to collect multiple passengers. If there are two passengers with different starting locations and both wanting to travel in the same direction, should the train either pick up both and then drive to the destination or pick them up separately and drive them to their destiantion. Both scenarios have their advantages and disadvantages, the first would decrease waiting time for all users, but increases driving time for the first user and

maybe confuses them because they drive away from the chosen destination. The second option would provide optimal service for the first user, but increases waiting time for the second user. Multiple trains would help in this case but come with other issues, such as managing which train picks up which passenger and drive in what direction it should drive.

One way to manage how trains pick up passengers, would be to apply first come first serve combined with checking if a new transport request occurs between the current position of the train and its destination. The management system can then redirect the train to the new location, where the train picks up the passenger and then continues to drive to the first passenger.

UI Improvements

As mentioned in Section 6.1.4, the layout of the app provides some problems on different devices, which leads to buttons being outside the screen or components being pressed into each other. Another layout issue that was mentioned was that the selection screen for start and destination look really similar, that could be improved by adding more indicator to the screens, helping differentiate between the two.

Displayed Information

In the feedback the testers returned, they missed some information and features in the app. One mentioned missing information about how many jobs are in front of you in the waiting line. This information can be added to this version of the app, but would lose it usefulness in later implementations, when the backend can manage multiple users and trains.

Language Support

Currently, the only language that is supported is English. Testers noted, that this app would be mainly used by residents of the area, that mainly speak German. The English version is beneficial for tourist that don't speak German, but by adding more languages it would allow a larger group of people to use the app.

Timer

The current timer, that displays the time until the train arrives at its next destination, is based on the current speed of the vehicle, leading to large fluctuations when the train drives at low speed. To improve it, a new way to calculate the timer must be implemented, that is based on the average speed. Doing so would decreases fluctuations and smooth out the timer.

Bibliography

[DCG21] Tyll Diebold, Felix Czarnetzki, and Carsten Gertz. "On-demand-angebote als bestandteil des öpnv: nutzungsmuster und auswirkungen auf die verkehrsmittelentscheidung in einem hamburger stadtrandgebiet". In: (Sept. 2021). DOI: 10.15480/882.3870.

[Dül] Christoph van Dülmen und Katharina Manderscheid. *Die nutzer*innenperspektive auf on-demand-mobilität in ländlichen räumen*. Accessed: 12-10-2024.

[Pav15] Marco Pavone. "Autonomous mobility-on-demand systems for future urban mobility". In: *Autonomes Fahren: Technische, rechtliche und gesellschaftliche Aspekte*. Ed. by Markus Maurer, J. Christian Gerdes, Barbara Lenz, and Hermann Winner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 399–416. ISBN: 978-3-662-45854-9. DOI: 10.1007/978-3-662-45854-9_19. URL: https://doi.org/10.1007/978-3-662-45854-9_19.

[RMW13] Xiaolu Rao, Markus Montigel, and Ulrich Weidmann. "Holistic optimization of train traffic by integration of automatic train operation with centralized train management". In: *Computers in Railways XIII: Computer System Design and Operation in the Railway and Other Transit Systems (Wit Transactions on the Built Environment)* 127 (2013), pp. 39–50.

[SRH+20] L. Sieber, C. Ruch, S. Hörl, K.W. Axhausen, and E. Frazzoli. "Improved public transportation in rural areas with self-driving cars: a study on the operation of swiss train lines". In: *Transportation Research Part A: Policy and Practice* 134 (2020), pp. 35–51. ISSN: 0965-8564. DOI: https://doi.org/10.1016/j.tra.2020.01.020. URL: https://www.sciencedirect.com/science/article/pii/S0965856418314083.