Technische Universität Berlin

Information Systems Engineering

Fakultät IV

Einsteinufer 17

10587 Berlin

www.ise.tu-berlin.de

Project Report

Praxisprojekt Anwendungssysteme Daimler Autonomous Van

Domenic Bosin, Alexander Dittmann,
Daniel Sebastian Lienau, Marius Möck,
Philipp Ratz, Antonio Schürer,
Christoph Markus Witzko

Under the guidance of

Prof. Dr.-Ing. Stefan Tai

Supervised by

Dominik Ernst

March 17, 2019

Contents

List of Figures

List of Tables

Acronyms

API	Application programming interface
CD	Continuous Deployment
CI	Continuous Integration
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
MVP	Minimum Viable Product
REST	Representational state transfer
SDK	Software Development Kit
VBS	Virtual Bus Stop

1 Introduction

Author: D. Bosin, S. Lienau
Reviewed by: A. Schürer

This project refers to a new kind of mobility service made possible by autonomous vehicles, which is one of the megatrends in the automotive industry[1].

Climate change is becoming ever more present and its consequences are already clearly noticeable in many countries[2]. Especially in urban areas, where driving bans for combustion engines are already being imposed in some cases because the overall air quality is so poor that it is harmful to health[3]. In addition, there is the fact that during many car journeys within the city only one or two persons sit in the car, which does not make ecological sense, as the unused seats could be used for people who want to travel to the same direction in order to reduce the overall traffic volume and thus avoid traffic jam[4]. Moreover, private cars are by far not used as much as public transportation vehicles and take up valuable space in the city while they park unused on the roadside.

In order to counteract this, it is necessary to move in a more environmentally friendly manner in the future. Because autonomous driving vans are electrically operated, can carry more people than most common cars and can operate 24 hours per day which makes them very flexible, autonomous vehicles will become indispensable in the future. Autonomous vehicles are the future not only because they are flexible, but also cost-effective because no driver is needed. In addition, autonomous vehicles are also ecologically advantageous because they are electrically operated and thus protect the environment as long as renewable energies are used to charge the batteries. Even if no renewable energies

[1] https://www.automobil-industrie.vogel.de/autonomes-fahren-definition-level-grundlagen-a-786184/ (accessed 15.03.2019)
[2] https://climate.nasa.gov/effects/ (accessed 15.03.2019)
[3] https://www.berlin.de/special/auto-und-motor/nachrichten/4947848-2301467-drohende-fahrverbote-was-dieselfahrer-wi.html (accessed 15.03.2019)
[4] https://www.umweltbundesamt.de/umwelttipps-fuer-den-alltag/mobilitaet/fahrgemeinschaften (accessed 15.03.2019)

are used, the air quality in the city is improved because the van itself does not emit any pollutants.

Daimler has developed such an autonomous driving van, which is already in use on a test area, but the goal is to use the van on public roads to offer mobility services to their end customers. However, there is still no mobile app that can be used to order that mobility service and to guide the user through the whole journey from start to destination.

Based on the developed user stories described more in detail in section 2.4, different interests are pursued. One of our target groups are business people, among other things, it was important for us to develop a solution that helps them to get from one meeting to another quickly and easily, so that they always arrive on time at their appointments. Other interests include social aspects like using the app to connect with other like-minded users of the service. In order to address even older people with our mobility service, who are not exactly enthusiastic about the new technology, it is important to keep the app as simple as possible.

The goal for this project was to develop a mobile application for ordering a ride with an autonomous driving van. Therefore, a concept for a mobility service including innovative features like loyalty program and ride sharing has been developed first before we realized this concept in form of a prototype smartphone application for the end user. The project was realized in cooperation with Daimler, which in this case is the supplier of the mobility service. Daimler commissioned our project team to design and realize that service, hence Daimler was our customer and we were the contractor. Although those vans developed by Daimler exists, they are not yet used in public transport. Thus, it is not possible to book a real van ride with our app. Instead, the vans and the rides are simulated by our self-developed backend.

This project report is structured as follows. In chapter 2 the requirement engineering is discussed in order to provide a better overview on how this project was organized, what kind of assumptions were made and how we solved them and finally all resulting user stories and the quality management will be presented. In chapter 3 the conceptual design will be discussed with the focus on the Minimum Viable Product (MVP), the architecture and the used technologies. In chapter 4 the implementation of our application is described. The evaluation of our application is presented in chapter 5. Finally, chapter 6 contains a summary including an outlook for future work.

2 Requirements Engineering

Author: M. Möck
Reviewed by: A. Schürer

Daimler Vans offers mobility services with an autonomous van. These services include a mobile application for its users. Among other things, this application should be able to order the mobility service for personal rides. In this chapter the assumptions that were made in advance of the requirement engineering process, a vivid and relatable user journey, as well as the Personas and their particular user stories that are used to build a base for the requirements for this mobile application are described.

The requirement engineering process[1] is shown in Figure 2.1. First, assumptions were made, to create a basis for the ensuing development process. After that, a generic user was constructed, which covers the fundamental needs of an average user. Based on this generic user, two important key elements were extracted from this generic user, the customer journey and generic user stories. The customer journey (section 2.3) provides a description of the user's path as he engages and interacts with the product. The generic user stories (section 2.4) summarize and elaborate the central needs of the generic user, corresponding to the customer journey.

Furthermore, three personas were created, to cover the specific needs of a user. From this personas custom user stories were evaluated. Together with the generic user stories and the customer journey, a feature mix consisting of basic and advanced features was distilled. Afterwards the features served to create specific tasks, which were processed and tested during the project lifecycle.

[1] D. Pandey, U. Suman, A.K. Ramani- (2010). An Effective Requirement Engineering Process Model for Software Development and Requirements Management. 2010 International Conference on Advances in Recent Technologies in Communication and Computing. P. 287 - 291.

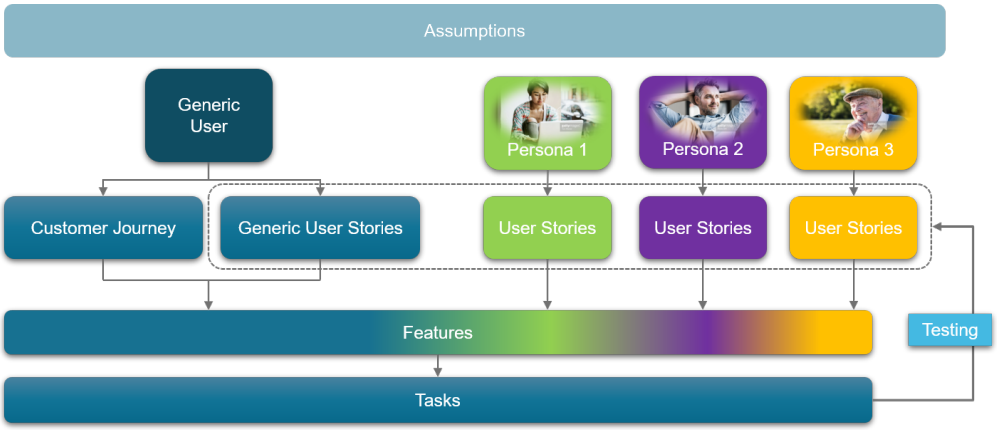


Figure 2.1: Requirement Engineering Process

2.1 Agile Project Management

Author: M. Möck
Reviewed by: D. Bosin

The requirement engineering process, pictured in the last section, disembogued in a variety of tasks. These tasks provided the basis for the product backlog, which was adjusted before every Scrum sprint. The tasks in the current product backlog, were dispensed between the team members, regarding their individual roles, shown in Figure 2.2. For efficient managing and progress measuring of these tasks the online tool Trello was used. After completing a sprint, the progress was summarized in an excel chart, including all tasks and corresponding user stories.

Scrum

Scrum is an agile project management procedure, which was selected for this project. This decision was made, because Scrum is suitable for small teams and fast development. In consideration of the short amount of development time (four month) and our team size of seven students, Scrum was the most fitting choice for these circumstances.
Scrum requires the assignment of the roles, which were allocated as shown in Table 2.1.

Project Owner	Scrum Master	Development Team
Daimler Vans:	Marius Möck	Domenic Bosin
Bernd Woltermann		Alexander Dittmann
Dr. Ingo Melzer		Sebastian Lienau
		Marius Möck
		Philipp Ratz
		Antonio Schürer
		Christoph Witzko

Table 2.1: Scrum Roles

The sprints took place in two week cycles and were completed with a sprint review. Once a week a meeting was held with the project owner Daimler, to settle and justify the procedure for the following week, based on the product backlog.
Besides the actual product in form of a mobile application, the outcomes of the sprints were the sprint reviews, documentation and testing reports.

Functional Team Organization

This section continues with the roles and responsibilities of the development team members, which are shown in Figure 2.2. The team organization follows a functional structure. The major contact partner for the project owner Daimler was the Scrum master, to ensure an efficient communication. The superscripted sections in the figure are indicating the responsibilities of the respective person. Bold printed names indicate the person, who hold the main responsibility for one scope. Some areas are overlapping because there are interfaces between them.

2.2 Assumptions

Author: P. Ratz
Reviewed by: D. Bosin

The following assumptions were made to ensure that the developed application is operable in a production-grade environment. If an assumption results in a requirement that is not yet in place and is not expected to be in place in the near future, the expected behaviour will be imitated or mocked exemplary.

- A van has a unique identification so that customers know which van they need to

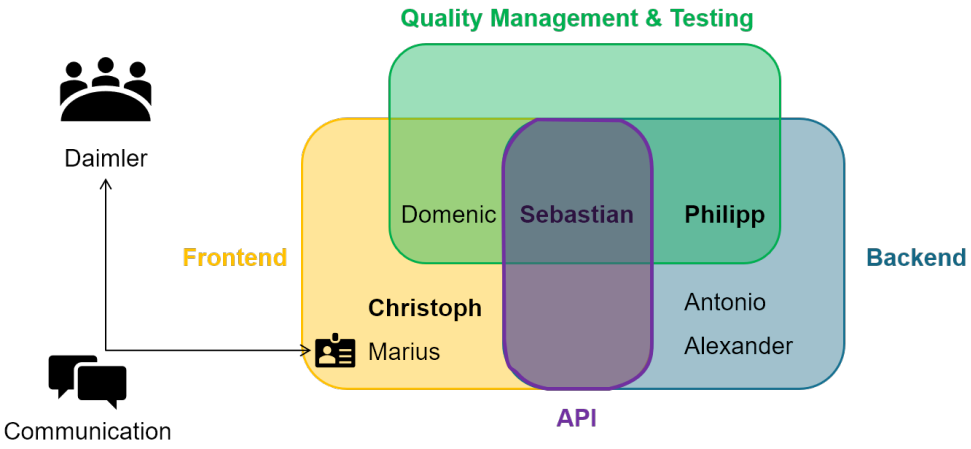


Figure 2.2: Team Organization

get into.

- The maximum capacity of a van is eight passengers.
- The van can stop at any designated point in the city to let passengers enter or exit
- A van and the management system (MS) are in place and working. A van is connected to the MS and can share its location at all times while receiving destinations to go to.
- Each van takes care of granting access to its authorized passengers.
- The MS can be accessed by our backend application to request rides and to retrieve the estimated time of arrival.

Vocabulary

Author: P. Ratz
Reviewed by: C. Witzko

A common vocabulary is vital for any customer-focused product. It defines the central terms related to the product and ensures that all involved parties share a common understanding of all of the product's components.

User	A user is a single natural person that uses any part of the offered transportation services or the corresponding application.
Starting point	The location from which a user starts their journey.
Destination	The location at which a user will end their journey.
Application (App)	The customer-facing part of the product that acts as a platform to request journeys and place orders. It can be accessed from mobile devices.
Order	An order is a virtual representation of a user's wish to ride a van and the corresponding parameters. It needs to be placed and confirmed for a van to actually come.
Account	An account is a virtual representation of one user. It is created by the user and is required to use the application.
Virtual bus stop (VBS)	A location at which a van can and will stop eventually to let passengers alight and enter.
Van	A van represents one self-driving vehicle that can carry passenger from one virtual bus stop to another.
Route	A potential geographic track from the user‘s start to their destination
Ride	The part of the journey between two VBSs that involves interaction with the vehicle

Table 2.2: Vocabulary

2.3 Customer Journey

Author: D. Bosin
Reviewed by: A. Schürer

A potential user hears from a friend that there are autonomous driving vans in the city. They have already read about it in social networks and on advertising billboards. One day they randomly pass one of these autonomous driving vans and see the advertisement for the app on the passenger door. In the app store, the user searches for the app and reads the user reviews. After reading mostly positive reviews, they download the app, create an account, and log in. Afterwards, the user receives a confirmation by email. Before they leave for work the next day, they open the app and order a ride with an autonomous van to take them to their destination. A few seconds later, the user will receive a confirmation of the booked van, as well as information about where and when the van will pick them up. Consequently, the user goes to the indicated location. Once they arrive there, they enter the booked van and are taken to the virtual bus stop that

is closest to his desired destination. While riding the van, the user has the opportunity to play games with other passengers through the mobile application, receive interesting facts about the van, and get information about their surroundings. It is also possible for the user to retrieve information on the expected arrival time or any potential delay. Shortly before the van reaches the virtual bus stop closest to the user's destination, the app informs them about leaving the van at the upcoming virtual bus stop. As soon as the van stops, the user alights from the van and walks to his desired destination. Because they reach their workplace on time and are delighted about the simple process and the seamless journey experience, they leave a five-star rating on the app store.

2.4 User Stories

Author: P. Ratz
Reviewed by: S. Lienau

To reach realizable and necessary requirements for the application we followed a user-oriented process that includes multiple steps. We followed an approach outlined by Roman Pichler on his website[2]. We first defined Personas that represent potential app users. This includes modeling the Persona's characters, their goals with respect to the application and what they expect from it. During this step it is important to ensure that Personas have different skills, intentions, and needs to cover a wide range of the application's user base and their interests.
Based on these Personas we derived various User Stories that define specific needs or wishes that the users have in relation to the application. These User Stories are meant to be collectively exhaustive, which means that they should cover all possible user interaction scenarios that are planned for implementation. In our case, we split up in groups and brainstormed for user stories. Each group came up with user stories for a different functional area of the application. In a group meeting we consolidated all user stories into one document. This process included rephrasing them to a shared standard, selection of relevant user stories, as well as prioritizing them. This document then became, after further consultation with the product owner, the base for the tasks that were defined in the project backlogs.
While the user stories that have been specified in the user story document are the common basis for the later implementation and should therefore not be subject to much change, we decided to make changes to it after each implementation phase (sprint) to adapt to

[2]Roman Pichler: https://www.romanpichler.com/blog/personas-epics-user-stories/ (accessed 15.03.19)

arising challenges or include new ideas.

Prioritization

Author: P. Ratz
Reviewed by: S. Lienau

The user stories were prioritized with a priority score that ranges from 1 to 3. A priority score of 1 means that the user story has a high priority and is a definite requirement, whereas a score of 3 translates to a low priority and means the features described in the respective user story is not of high importance. All user stories were prioritized according to their relevance for the overall service experience, their feasibility within the limits of time and resources and our personal preferences. The priority score directly corresponds to the order of implementation. User stories with a priority score of 1 are to be focused on and will be the first to be implemented. User stories with a priority score of 2 are planned to be implemented but will not be focused on if complications arise. User stories with a priority score of 3 can be seen as "nice to have" and are not planned in the main backlog. They represent ideas that could be implemented if the project has time and space for them.

Generic User Stories

Author: All members

The following user stories represent the very basic app requirements which are relevant to all users. They consequently all have a priority score of 1 and must be fulfilled.

1. As a user, I can get from the Tegel Airport to the Daimler office in Berlin.
2. As a user, I can get a suggested route from my current location to Daimler office in Berlin.
3. As a user, I can see the current estimated time of arrival of my suggested route at any time.
4. As a user, I can see my current location, while I'm on my way to the Daimler office in Berlin.
5. As a user, I can access all required information in an intuitive way.

6. As a user, I can see the waiting time before I order a van, so that I can decide whether I want to place an order or not.
7. As a user, I can order a van, so that it will pick me up at a specific virtual bus stop.
8. As a user, I can access the shortest route to get to the virtual bus stop of the recommended van.
9. As a user, I can see the time of arrival and the current location of the van.
10. As a user, I can unmistakably identify the recommended van assisted by the given information of my current route.
11. As a user, I can get all relevant information, after entering the van.
12. As a user, I can see at what time I must leave the van.
13. As a user, I can see the shortest route to the Daimler office in Berlin, after leaving the van.
14. As a user, I can cancel my order before the van picks me up.

Custom User Stories and Personas

Author: All members

In this section we introduce three fictive personas that we created to generate user stories that focus on different needs of groups of user's with individual requirement for our app. Each of the personas has some typical characteristics that help finding desired features for these special needs. For each of the personas we derived an individual epic and individual user stories, which are based on the persona's characteristics. To prevent confusion: The numeration in the listed user stories is the numeration from our whole list of user stories.

David Summer

Description David Summer is a 35 year old businessman working in management. He is competitive and does not have much time, wherefore he is always stressed out. He also has a bad sense of direction. David likes to be efficient and really dislikes waiting as well as being late for appointments.

Epic As David, I usually have a very busy schedule. That's why I need a solution which helps me to get from one meeting to another meeting quickly and easily so that I am always on time for my appointments. I also enjoy being able to get data about my past activities to help me with future decisions.

User Stories

15. As David, I can access all information about my past rides to optimize my future usage. (1)
16. As David, I can get an electronic receipt for my ride, so that I can get a refund from my company. (1)
17. As David, I can access all features of the app in an intuitive and fast way, so that I save even more time. (2)
18. As David, I can have the app remember my credentials and keep me logged-in, so that I do not have to enter them every time I open the app. (2)
19. As David, I can receive suggestions about when and where to go next based on my calendar and the location information stored in my calendar. (2)
20. As David, I can configure what kind of notifications/emails I get, so that I only receive the information that is relevant to me. (3)
21. As David, I can set the office as my place of work, so that I have a shortcut for navigating to the office (3).
22. As David, I can pre-order the van, so that the van is waiting at e.g. 2 pm after my meeting to take me directly to another appointment at 2.30pm. (3)

Sarah Green

Description Sarah Green is 22 years old and a digitial native and influencer. She cares about the environment and has a love for the detail. She is also an athletic person. Sarah likes sharing and comparing her app usage with others and enjoys interesting side information. In addition, she likes traveling and being in the nature. Sarah dislikes the pollution of the environment and being bored.

Epic As Sarah, I always need to keep my social media followers entertained. Using and talking about the newest and trendiest things is my daily business. By using the Daimler Van and the associated smartphone app, I can show my social media audience that I am up-to-date with the newest technology and that I am using an environment-friendly transportation service. Furthermore, I want to use the app to get in touch with other like-minded users.

User Stories

23. As Sarah, I can share my current location with others during the ride. (1)
24. As Sarah, I can see interesting facts about my current environment, so that I will be entertained in the form of podcasts, information texts or videos during the ride. (2)
25. As Sarah, I can see how much CO2 I'm saving by taking the autonomous van, so that I feel more eco-conscious and share this with others. (2)
26. As Sarah, I can get bonus points for my driven distance and get rewards for that. (2)
27. As Sarah, I can sign in with my social media account, so that my personal data and pictures can be taken from there and don't have to be re-entered. (2)
28. As Sarah, I can see information about other passengers in the same van, so that I can determine similarities and come together in conversation. (3)
29. As Sarah, I can set which information is visible to other passengers. (3)
30. As Sarah, I can customize my routing preferences, so that I can incorporate some walking time into my trip. (3)

Wilbert Kettlestone

Description Wilbert, 62 years old, is a (former?) car mechanic. Since he is not the youngest anymore he is not very confident with new technologies. Wilbert is a food lover and likes playing board games with his family. He is not very spontaneous and dislikes canceling plans. He is also quickly overwhelmed by too much information at once.

Epic As Wilbert, I am easily confused by too much information. While I own a mobile phone, I am not very proficient using it. This is why I need applications like the Daimler Van mobility services to guide me through their processes. I would also enjoy it if the app helped me pass the time, for instance with interesting information about the vehicle and its technology or simple games I could play against other users.

User Stories

31. As Wilbert, I can get an instructive tour through the application, so that I am being introduced to its main features. (2)
32. As Wilbert, I can be guided through the ordering process, so that I don't forget any relevant points. (2)
33. As Wilbert, I can get suggestions for future rides based on information of my past rides because I often take the same route. (2)
34. As Wilbert, I can play some simple games all by myself or against other people during the ride, so that I can enjoy the ride even more. (2)
35. As Wilbert, I can access the game results of my past games and see a ranking of other users based on that points. (3)

Added User Stories

Author: M. Möck

During the development process three new user stories were added. Furthermore the ride sharing concept brought into focus. The decision was made to specify this concept as the unique feature of the application. To cover this functionality two new user stories were created.

36. As a user, I will be asked to confirm that I entered the van once I'm close enough to the van.
37. As a user, I can share my ride with other passengers, so that I reduce pollution and traffic on the streets.
38. As a user, I can order a van for my friends and me, so that we take a ride together.

2.5 Quality Management

Author: C. Witzko
Reviewed by: P. Ratz

The quality management process during our project was an essential tool to detect unforeseeable issues early and monitor the implementation progress of our project. It was designed to be simple but efficient and turned out to be very helpful for our weekly status updates for the product owner.
As it is infeasible to test user stories directly for their integrity and correctness of implementation we decided to create one or more acceptance tests for each user story. These acceptance tests usually cover a very high-level test scenario and are broken down into one or more test cases.[3]
The following example demonstrates the procedure:

First, we select a **user story**. For this example we choose user story 14:
As a user, I can cancel my order before the van picks me up.

Next, we create an **acceptance test**:
Given that I am logged in as a customer and have ordered a van, then clicking the button 'Cancel Order' takes me to a view showing that my order was successfully canceled.

Finally, the acceptance test is broken down into several **test cases**:

1. After logging in and booking a van the 'Cancel Order' Button exists.
2. Pressing the 'Cancel Order' Button goes to the order cancel confirmation view.
3. The order should be marked as canceled in the database.

This procedure was repeated for every single user story. The test cases were performed manually during and after the sprints to check whether a user story had been implemented successfully. The results were documented in a spreadsheet. This constant user story mirroring process greatly facilitated the actual implementation of the application.

[3] Software Quality Assurance and Testing / How can I test a user story? by Kate Paulk: https://sqa.stackexchange.com/questions/6450/how-can-i-test-a-user-story-examples-please/64586458 (accessed 07.03.2019)

3 Conceptual Design

Author: D. Bosin, A. Dittmann

In this chapter we describe the overall concept of our product. We begin with an explanation of our minimum viable product (MVP), which contains the most important features for a working prototype. Based on the MVP we will continue with focusing on two additional features beyond the MVP: Ride sharing and a comprehensive loyalty program. These sections are then followed by the architecture and the communication of our whole application. Afterwards, the technologies that we used in our application are discussed.

3.1 Minimum Viable Product

Author: D. Bosin, A. Dittmann
Reviewed by: P.Ratz

This section focuses on the Minimum Viable Product (MVP) to provide insight into what we chose to be the minimum requirements for application to be a sound product. The minimum requirements mainly consist of the generic user stories that have been prioritized with a score of one. Our MVP covers all of the 14 generic user stories. Based on the requirement engineering process outlined earlier, a catalog of necessary user stories was derived. The following section does not cover all of these user stories but highlights some of the important ones exemplarily.

One of the minimum requirements that the app must provide is a ride history to get an overview of the past rides including some loyalty program information. Constant showing of the vans' location and the user's waiting time is another important minimum requirement. Before the customer is able to order a van, the app must be able to display the current vans on a map and navigate the customer to the virtual bus stop and indicate the waiting time if he orders a van. The most important part of the MVP is the ordering process. The user must be able to order a van via the app. The order process contains

the following steps. First, the user enters his start and destination position. Subsequently, his route will be displayed on a map including estimated time of arrival. Now the user has the opportunity to order this route within a minute until it expires. After the route has been ordered, their journey begins.

The development of the MVP was completed in January as planned. Hence, we had one last sprint remaining to implement features on top of the MVP. We decided to implement a *ride sharing* feature and a comprehensive and sophisticated loyalty program, the loyalty program being also a requirement of the product. Both features as well as the motivation behind them are explained in the following two sections.

3.2 Ride Sharing

Author: A. Dittmann
Reviewed by: A. Schürer

A key idea of the mobility service is the concept of ride sharing. This feature not only makes the van service and also our app unique compared to many other mobility services, but we were also very interested in approaching the problem of ride sharing and seeing what challenges may arise. That is why we decided on integrating this feature into our app. Since ride sharing was not covered in our original list of user stories, we had to extent it, as already mentioned in user story 38 in the last chapter.

Two passengers who ordered a van can end up in the same van and thus share their rides, if certain circumstances match. The circumstances under which this happens are now going to be explained. Therefore, we distinguish between two scenarios:

- Scenario A: Two passengers, Alice and Bob, have the **same start** virtual bus stop.
- Scenario B: Two passengers, Alice and Bob, have the **same end** virtual bus stop.

Scenario A Alice has ordered a van to an arbitrary destination. A van was assigned and is on its way to the VBS where Alice will enter the van. Now, Bob orders a van to some other arbitrary destination. If Bob's start VBS is the same as Alice' start VBS, we could possibly pick them both up with the same van and then bring them to their respective destinations. Ride sharing is possible, if there is a route which first brings Alice to her destination and then Bob to his destination (or vice versa, depending on which

route is faster) without neither Alice nor Bob having a delay of more than ten minutes compared to straight routes to their destinations. If this condition is fulfilled, their rides will be shared and both Alice and Bob can meet up in the van, eventually having a nice conversation.

Scenario B Alice again has ordered a van to some destination. Bob orders a van, too, but this time he wants to get to the same destination as Alice. Again, this would be a good opportunity to bring them to their common destination together with the same van. But this time we have two different cases depending on whether Alice has already entered the van or not. If Alice is not in the van yet, then the van will always first pick up Alice and then Bob, and then bring them both to their common destination. We do so because we do not want that Alice, who ordered the van first, has to wait longer. If Alice has already entered the van, then we can pick up Bob next. Therefore, we determine a way point that the van will pass in at least 90 seconds in the future. We take this way point for calculating a route via Bob's start VBS to Alice' and Bob's common destination. This makes sure that Bob has enough time to check the route and confirm the order. After Bob confirms the order the van's route will be updated accordingly to pick him up. In both of the cases only Alice' route changes compared to her original route. Again, we will only allow ride sharing, if Alice experiences no delay longer then ten minutes.

To conclude this, ride sharing is possible, if both passengers have the same start or destination VBS and neither of the two passengers would experience a delay of more than ten minutes due to the modified route. We focused on the two mentioned scenarios because, as already stated, ride sharing was just planned in the last sprint and it was not clear how complex the overall development of this feature would be. This way we could keep the complexity under control and make sure to successfully implement this feature in the end. At the end of the sprint we had both scenarios implemented and extensively tested, to ensure that it is working properly. This also shows that we planned the ride sharing feature just right for our schedule. The user receives loyalty points for sharing the van, which is described in the next section.

3.3 Loyalty Program

Author: A. Schürer
Reviewed by: D. Bosin

The main goal of the loyalty program is to establish a long term relation to the user

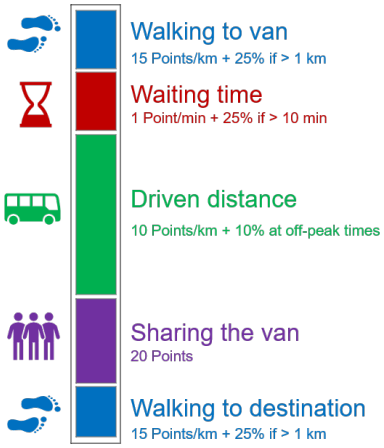


Figure 3.1: Loyalty Program

and motivate the user to use the mobile app and the van service. Moreover, the loyalty program rewards the user for a behaviour which enables the van service provider to operate efficiently. We identified different ways to reward the user as shown in Figure 3.1. If the user is willing to walk a longer distance to the van, then the van service provider can pool different users with different routes together with a smaller detour. The user is rewarded for walking to the virtual bus stop where the van picks him up with 15 loyalty points per km and 25% extra points if the walking distance is greater than 1 km. The same applies to the walking distance from where the van drops the user off to the final destination of the user. The user is also rewarded for waiting for the van. When the user is willing to wait longer for the van, this enables the van service provider to manage the routes of the vans and pooling of different users more flexible. The user gets one point for each minute of waiting for the van and 25% extra points if the waiting time is greater than 10 minutes. The waiting time starts to count when the users arrives at the virtual bus stop. The core of the loyalty program is the driven distance. The users is rewarded with 10 loyalty points per km and 10% extra points if he is using the van at off-peak times. Additionally, the user gets 20 loyalty points when he shares the van with other passenger because of pooling.

The user earns loyalty points with each order. Those loyalty points are accumulated and are presented in the account screen of the app. Collecting loyalty points leads to the achievement of a loyalty status which is presented in both the account screen and leader board screen. The leader board presents the top 10 users with the most loyalty points. Since payment for the van service was out of scope for our project, we didn't implement

any material incentive and focused on gamification features.

In the next section we describe the architecture of our application.

3.4 Architecture

Author: A. Schürer
Reviewed by: S. Lienau

In this section we provide an overview of the architecture as shown in Figure 3.2 and the communication between the frontend and the backend.

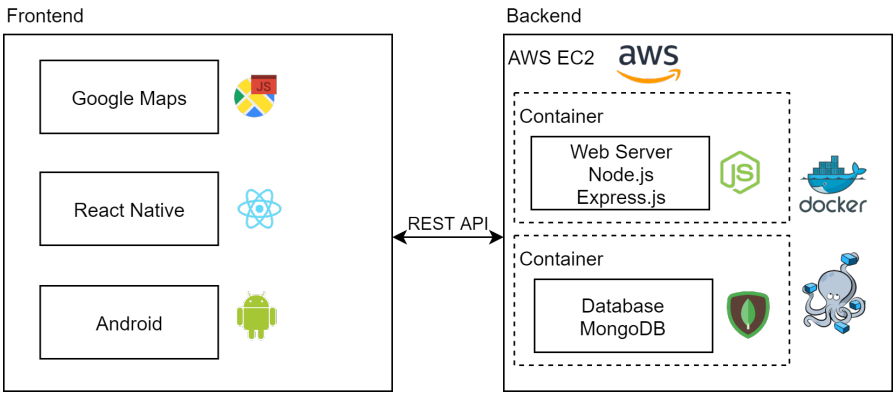


Figure 3.2: Architecture

In the frontend we are using Android, React Native and Google Maps. In the backend we are using a Node.js Express.js web server and a mongoDB which are deployed in their own container and managed by Docker Compose. The backend is hosted on a Ubuntu Linux VM on AWS EC2. For communication between the frontend and the backend we are using HTTP and a REST API. The frontend sends the username and password upon user login and receives a JSON Web Token (JWT) from the backend. For all following requests from the frontend to the backend the user is authenticated by the JWT.
In the next section we explain our decisions on the individual frontend and backend components.

3.5 Technologies

This section will discuss which technologies for the frontend as well as for the backend were used and why.

3.5.1 Map Service

Author: D. Bosin
Reviewed by: C. Witzko

Since the application requires a map service to display van locations, routes and virtual bus stops, it is necessary to familiarize with the currently available map services. Therefore we have analyzed four prominent map services: Google Maps, OpenStreetMap, MapBox and HERE Maps. The comparison is based on the most important features with which the service should come with and can be found in the Table 3.1.

Feature	Google	OpenStreet Map	MapBox	HERE Maps
Detailed API documentation	✓	✓	✓	✗
Location Tracking	✓	(-)	✗	✗
Turn by turn navigation	✓	✗	✓	✗
Public transport	✓	✗	✗	(-)
Gated areas	✗	✗	✗	✓
Free usage	(-)	✗	✗	✓
Customization options	✓	✓	✓	✓

Table 3.1: Map Services

We have chosen Google Maps as map service for our application, because it covers the most important features regarding our needs.

3.5.2 Frontend Technologies

Authors: S. Lienau, C. Witzko

Our frontend application is mainly build with the `React Native`[1] framework explained in the following.

React Native

At the beginning of the project we had the choice to develop an app for iOS or Android. We decided to develop an Android app using React Native. The following is a brief comparison of the technologies we had to choose between.

- **Objective-C or Swift (iOS)**

[1] React Native: https://facebook.github.io/react-native/ (accessed on 15.03.2019)

  - The basic requirement for iOS development is a Mac with macOS. As we are using our private computers for this project and not everyone on our team is equipped with a Mac we decided against Objective-C / Swift.

- **Java (Android)**
  - All project members have experience with Java, but not with the native development of native Android apps. Since the user interface design in React Native is based on concepts of web technologies (e.g. `flexbox` is used for positioning) and all project members had experiences in web development and some even with React, this was a better option than to deal with Android's own user interface design concepts which would have been completely new to all of us. In addition, the aim of the project was to develop a good-looking prototype with as many functions as possible in a relatively short time. Android Studio, unlike React Native, doesn't allow something like hot- and live-reloading, which makes changes to the code immediately visible in the app. With native Android app development the code has to be recompiled after every change to the code, which takes time, even if only small things have been changed.

- **JavaScript / React Native (Android / iOS)**
  - **JavaScript** gained a lot of popularity in the last years and is today the most popular programming language in the open source community[2]. In our team, JavaScript was the language everyone had experience with and therefore using it as our primary programming language made a lot of sense. Another big advantage is to have one programming language for our frontend application and our backend services.
  - **Reusability** A big advantage of React Native is that this technology and the learned knowledge can also be used for the development of web applications with React. This also works in the other direction. Some of us already had experience developing web applications with React and could reuse that knowledge for developing our app using React Native. Furthermore, React Native is a very popular framework and is used in well-known and widely used apps such as Instagram or SoundCloud[3]. So it is likely that the learned knowledge can also be used in future (web or mobile app) projects.

[2]Octoverse: https://octoverse.github.com/projectslanguages (accessed on 26.11.2018)
[3]https://brainhub.eu/blog/react-native-apps/ (accessed 15.03.2019)

What is React Native? React Native is a JavaScript framework for writing real, natively rendering mobile applications for iOS and Android. It is based on React, Facebook's JavaScript library for building user interfaces, but instead of targeting the browser, it targets mobile platforms. [4]

Redux

Since React provides just the view layer (the `V` of the `MVC` pattern) for our application, we need a layer to store the data. In this project, we use `Redux`[5] to provide this store. Redux provides a *central* data store which can be used in any component. The only thing we have to do is to `connect` a component which relies on a global application state to the redux store. If we would not use redux and manage the state just using react instead, the whole state (at least those fields that are used by multiple, independent screens) must be maintained in the root-component and passed down the component-tree via props.
For instance, if a component on the 5th hierarchy level depends on a global application state, we would have to pass down the data from the 1st through the 2nd, 3rd, 4th and finally to the 5th component, even though components on hierarchy level 1 till 4 don't need that data for themselves.
The same applies in the other direction. Not only must data be passed further down, but also upwards if a component wants to change data in the store.
Because this is very inconvenient and error-prone, redux helps us to write cleaner code. With Redux you can connect any component to the store, no matter on which hierarchy level it is located.

3.5.3 Backend Technologies

Authors: A. Schürer, P. Ratz

The following section elaborates the backend technologies and explains decisions on the individual backend components. Since the project goal was to implement a prototype with innovative features in a very short time period, we chose a simple solution with only few tools and frameworks.

The backend is powered by a Node.js Express web server, which we chose because it's widely adopted and has many tools and frameworks for authentication, database

[4] Learning React Native by Bonnie Eisenman: https://www.oreilly.com/library/view/learning-react-native/9781491929049/ch01.html (accessed 11.03.2019)
[5] Redux: https://redux.js.org/ (accessed 15.03.2019)

connection and server middleware. Additionally, we hoped for synergies when using Node.js in the frontend, the backend and also the native data representation JavaScript Object Notation (JSON) in the API. The JWT are managed by Passport.js[6]. We chose MongoDB as database management system, because it offers JSON as native data format, a JSON based query language, existing Node.js frameworks, and JavaScript integration. Having the same programming language and data representation in the frontend, API, backend and database gives us the flexibility to move developers between the frontend and backend based on how many frontend and backend task are in the sprint backlog. We did not choose MongoDB for its non-relational data structures. In fact, the NoSQL-nature of MongoDB is arguably not optimal for all of the project's data storage needs. While we were able to use MongoDB for all its intended purposes we acknowledge that other database system might perform better. This point is therefore certainly one to be revisited.

The Node.js Express.js web server and the MongoDB are each deployed in their own container. The containers are based on Docker[7] and the deployment is managed by Docker Compose[8]. These technologies offer multiple advantages. Docker enables an easy deployment of the backend with all dependencies documented in the docker file and included in the docker image. It also helps to avoid vendor lock-in effects since our backend can run on any system that has docker installed. This provides the opportunity to move our backend to other cloud platforms. Moreover, we can create a local development and testing environment which is the exactly like the production environment. An external requirement given by the project owner was to use Amazon Web Services[9] (AWS) to host the backend. The backend is hosted on an AWS Elastic Compute Cloud[10] (EC2) running Ubuntu Linux virtual machine.

[6]Passport.js: http://www.passportjs.org/ (accessed 06.03.2019)
[7]Docker: www.docker.com (accessed 28.02.2019)
[8]Docker Compose: https://docs.docker.com/compose (accessed 28.02.2019)
[9]Amazon Web Service: https://aws.amazon.com (accessed 28.02.2019)
[10]AWS Elastic Compute Cloud: https://aws.amazon.com/ec2 (accessed 28.02.2019)

4 Implementation

Author: P. Ratz
Reviewed by: C. Witzko

This section of the report will give an overview of how the concepts described in the previous section were realized and built. It does not only include a description of the development approach but will also touch on some of the specific technical solutions that were used to build the application as well as the backend.
The first subsection will briefly talk about the guidelines that were followed during the development process. It will also introduce some of the technologies and tools that were made use of such as continuous integration. The second and the third subsection will focus on the implementation of the frontend and the backend respectively. Here, we will give a short overview of how the source code is structured. Furthermore, we will use some specific examples to demonstrate how some of the core features have been implemented. While the frontend subsection will talk about the main app components, the backend subsection will introduce its main components within a tour through the actual ordering process. We cover the backend subsection a little differently for three reasons. Firstly, the order in which the backend components are usually used or called is mirrored by the ordering process, which makes their introduction within a tour through the ordering process easier to understand. Secondly, the frontend subsection will already introduce some of the API components and, thirdly, the mark-down API documentation that can be found in the appendix of this report also gives an extensive overview of how the backend was implemented.

4.1 Development Approach

Author: C. Witzko
Reviewed by: A. Schürer

To simplify and unify our whole software development process we agreed on certain standards and procedures on how programming code is written, reviewed and deployed.

In the sprint planning process, we decided which user stories we wanted to implement during the sprint. As a single user story is a very high-level specification of the planned implementation, it was broken down into multiple small development tasks. An example of such a task: "Create a new API endpoint to list all vans of the van management system". After creating the tasks they were assigned to the different development teams (frontend and backend) and later on assigned to individual engineers inside the teams.

We wanted to have a unified code style. When it comes to implementing a functionality there are plenty of different ways to write the corresponding source code. As every engineer has their own personal preference in this matter, the code style of a project would end up to be very inconsistent. This was the main reason why we decided to use a fixed set of rules for the code style and comply with the JavaScript Standard Style.[1]

We used Git as a version-control system to track all changes on the source code historically. To host our repositories centrally we used the software development platform GitHub. To introduce as few bugs as possible, every change in the code base was required to be peer-reviewed by at least one other engineer. We accomplished that by using pull-requests.[2].

To continuously integrate (CI) developed software components, every push to a branch triggered a build on our Travis CI[3] platform. As GitHub and Travis CI are directly connected, a passing build is another necessary requirement to be able to merge a pull request. During the build the CI clones the repository, checks out the pushed branch and runs the linter and the test suit. If there were no linting errors and all tests have passed the build was successful. With that process, we ensured that our development branch is always in a working state and the code base complies with the code style rules. The manual deployment process for the backend service was not trivial and forgetting to deploy the latest version manually can lead to inconsistencies. That is why we decided to automate the whole deployment process. After the build for the development branch has passed the code is automatically deployed to our infrastructure.

[1] JavaScript Standard Style by Feross Aboukhadijeh: https://standardjs.com/ (accessed 11.03.2019)
[2] About pull requests: https://help.github.com/en/articles/about-pull-requests (accessed 11.03.2019)
[3] Travis CI: https://travis-ci.org/ (accessed 16.03.2019)

4.2 Frontend

Authors: D. Bosin, S. Lienau
Reviewed by: A. Dittmann

This section describes how the mobile app is fundamentally built, how the state management, screens and components work. Code snippets are used for explanation purposes.

4.2.1 UI components

Author: S. Lienau

We have created some user interface components which are used in different screens and can also be used for further development. These can be found in the `/src/components/UI` folder. As a starting point we use the JavaScript library `Native Base`[4], which provides ready to use UI components.
For instance, the component `DefaultScreenHeader` is used in all screens that have a header, e.g. `AccountScreen` or `LeaderboardScreen`. The appearance and behavior of the UI components can be determined by setting their respective props. Thus, `title` determines the display text in the header and `onPress` the function which is to be executed when the icon is clicked. Additionally, the visible icon can be set by the `icon` prop, which is set to `arrow-back` by default.
The file `colors.js` contains several color codes which are used by different components. If you want to change the color scheme, this only has to be done at this single point.
We don't want to describe every single UI component here, we just wanted to explain what they are and what they are used for. Use-cases can be found in the different screens of the app.

4.2.2 Navigation

Author: S. Lienau

For the navigation between the screens we use the JavaScript library `React Navigation`[5]. The setup takes place in the file `/src/init/App.js`, where all screens are registered with the navigator and rendered afterwards. First of all, our root navigator is a

[4]Native Base: https://nativebase.io (accessed 12.03.2019)
[5]React Navigation: https://reactnavigation.org/ (accessed 12.03.2019)

`SwitchNavigator`, which can only show one screen at the time and is not able to handle `back` actions. It is kept by the variable `AppNavigator` and is rendered as the root component later on.

```
const AppNavigator = createAppContainer(
  createSwitchNavigator(
    {
      Loading: LoadingScreen,
      Login: LoginScreen,
      MainAppStack: MainAppStack,
    },
    {
      initialRouteName: 'Loading',
    }
  )
)
```

The initial screen here is the `LoadingScreen`, which will do nothing more than redirecting to either the `LoginScreen`, if no valid token exists, or to the `MainAppStack`, if a valid token is present meaning the user is logged in.

```
if (!(await isTokenValid())) return this.props.navigation.navigate('Login')
this.props.navigation.navigate('MainAppStack')
```

Once the user is logged in, the `MainAppStack` is shown, which holds all "main" screens like `MapScreen` or `AccountScreen`. The `StackNavigator` is able to handle `back` actions, so it is possible to `push` the view from one screen to another (put a screen on top of the stack), and `goBack` later (drop the top element from the stack).
Additionally, `navigationOptions` can be set for each screen individually. For instance, for the `AccountScreen` we set the `header` to the `DefaultScreenHeader` component, which is described in the previous section. The screen header's `onPress` function is set to `navigation.goBack()` here, so it will drop the current screen from the stack once the back-Button is pressed.

```
const MainAppStack = createStackNavigator(
  {
    Account: {
      screen: AccountScreen,
      navigationOptions: ({navigation}) => ({
        header: (
          <DefaultScreenHeader
```

```
            title="Account"
            onPress={() => navigation.goBack()}
          />
        ),
      }),
    },
    Leaderboard: { ... },
    Map: { ... },
    ...
  },
  {
    initialRouteName: 'Map',
  }
)
```

Here, the initial screen is the `MapScreen`. We removed the header on this screen since this is the first screen of the stack. Furthermore, this screen plays an important role in our application because it is always mounted. Therefore, it contains a lot of functionality that needs to be constantly executed in the background (e.g. periodically get the active order status from the backend).
The navigator can be accessed via `props.navigation` in any screen. For instance, to navigate from the `AccountScreen` to the `OrderHistory` screen, the `push` function is called, when the list item is pressed:

```
onPress={() => this.props.navigation.push('OrderHistory')}
```

4.2.3 API Calls

Author: S. Lienau

The API endpoint is set in the file `/src/lib/config.js` via a default export of the property `apiEndpoint`. This IP is then used for all API calls. The API specification can be found in Appendix C.

axios

For the API calls we use the `axios`[6] JavaScript library. The API axios object is configured and exported in the `/src/lib/api.js` file, so it can be imported and used in other parts

[6] axios: https://github.com/axios/axios (accessed 12.03.2019)

of the application. The IP from the previous section is included in the `baseURL`. Requests are made via `HTTP` on port `8080`. `Content-Type` is set to `application/json` and the default `timeout` is set to `10 seconds`.
Furthermore, on successful login, the function `setAuthToken` is executed, which sets the default `Authorization` header, so that it is send with every request according to our API specification.

axios interceptors

In axios one can use so-called interceptors. With interceptors it is possible to manipulate every request and every response before it is processed further.

Our `response` interceptor is responsible for error handling. If any response contains an error, a JSON string containing the `error` object specified in our REST API will be returned. The interceptor then reads the two values `response.data.code` and `response.data.message` before it `throws` a new error object containing those values, so that other components of the app, which are using the API, can just read the error code and message from that object.
The procedure just described will only be used, if the HTTP status code is not `401 (Unauthorized)`. If this is the case, the app will first try to log in with the stored credentials and then send the request again. If this is also not successful, an error will be thrown.
The relevant code can be found in Appendix A, Listing 7.1.

4.2.4 State Management

Author: S. Lienau

Redux uses `actions` and `reducers` to handle the state. There can be multiple reducers to logically separate states and therefore make the app cleaner. Our global redux state consists of three reducers, which can be found inside the folder `/src/ducks`:

1. `account` for all account relevant data as well as leaderboard and order history
2. `map` for all data required for the MapScreen as well as current route information such as arrival time or van position
3. `orders` for the active order (if there is one) and for the active order status

It should be mentioned here that the separation could be improved. Initially, the `orders` state was intended for all past and active orders, including route information. Now, that the Order History has become a sub-screen of the `AccountScreen`, we decided to move the `pastOrders` field to the account state. Thus, the orders state contains only the active order.
At the same time the booking process takes place on the `MapScreen`, which also shows a suggested route including departure and arrival times. Hence, these data belong to the map state first, but as soon as this route is booked, it is part of an active order. In our current (final) implementation some data will be available in the map as well as in the orders state, although we are only reading data from the redux map state for displaying the route information like arrival time. For further development of the project, it might be a good idea to clearly separate this. One approach could be to handle the entire booking and ride process within an own state. We started with a refactoring on this but quickly realized that it was too complex and time consuming for our prototype app. The map state is used in many places in the app and the risk of introducing bugs during refactoring is too high.

Map State

Each redux state needs an initial state, which exists if no data has been changed by actions yet. We will only explain the map state here, since the map state is the most important and most used state in our application. The other two states are less complex and easy to understand. The initial map state is defined in `src/ducks/map.js` and can also be found in Appendix A, Listing 2.2.
First of all, the `orderState` field is used a lot in the `MapScreen` for conditional rendering (render a component or not, depending on the state). It is an *enumerated type* and can hold the following values:

```
export const OrderState = {
  INIT: 'INIT',
  SEARCH_ROUTES: 'SEARCH_ROUTES',
  ROUTE_SEARCHED: 'ROUTE_SEARCHED',
  ROUTE_ORDERED: 'ROUTE_ORDERED',
  VAN_RIDE: 'VAN_RIDE',
  EXIT_VAN: 'EXIT_VAN',
}
```

Value	Description
`INIT`	The inital state of the map, where no destination location is set (start location is set to current used location by default).
`SEARCH_ROUTES`	The state when a destination location is set. The `SearchForm` component and the *Search Routes* Button are visible.
`ROUTE_SEARCHED`	The state when a route has been searched and we got a suggested route from the backend. The route is shown on the map and can be ordered.
`ROUTE_ORDERED`	The state after a route has been ordered, but the van has not been entered yet. The van's live position and departure time is shown. It is possible to cancel the order or *Hop On* the van if user and van are both at the starting virtual bus stop.
`VAN_RIDE`	The state during the ride, after the user has entered the van. The `RideScreen` is shown.
`EXIT_VAN`	The state after the user left the van. `MapScreen` is shown again and guides the user from the ending virtual bus stop to the final user destination.

Next, a brief explanation of the other fields of the `map` state follow.
`userStartLocation` and `userDestinationLocation` hold the values entered on the search form on the `MapScreen`.
`currentUserLocation` holds the live user location and is constantly updated and sent to the backend once an order has been placed.
`routeInfo` is an object and holds route relevant information. The objects' field names match those of the API specification, where they are well described.
`visibleCoordinates` and `edgePadding` control the displayed map section (e.g. the whole route including the footpath, or only the footpath to the starting virtual bus stop).
`personCount` holds the number of passengers for the order.
`vans` is an array and holds the position of each available van. These are displayed on the map before an order has been placed.

Manipulating Map State

To change one or more fields of the state, an `action` needs to be *dispatched*. Actions are defined as constant Strings. For instance, to change the actual order state, we have defined an `action` called `CHANGE_ORDER_STATE`:

```
export const CHANGE_ORDER_STATE = 'map/CHANGE_ORDER_STATE'
```

This action is then used in a dispatch function, which in this case gets the new `OrderState` as input (payload). This function is imported and used in different parts of the application, wherever the order state needs to be changed.

```
export const changeOrderState = payload => {
  return {
    type: CHANGE_ORDER_STATE,
    payload: payload,
  }
}
```

After this function is executed, the `action` and the `payload` is forwarded to the `reducer` function. In the `reducer` we define how the redux state should be changed. In this case, we just want to set the `orderState` field of the new state to the `payload` value.

```
const map = (state = initialState, action) => {
  const newState = _.cloneDeep(state)
  switch (action.type) {
    case CHANGE_ORDER_STATE:
      newState.orderState = action.payload
      return newState
    default:
      return state
  }
}
```

A more complex example is the `fetchRoutes` dispatch function, which executes an API call first and dispatches three actions afterwards. This function is executed when the `SearchRoutesButton` is pressed. Because Redux doesn't support asynchronous calls natively, we use the JavaScript middleware *Redux Thunk*[7].

```
export const fetchRoutes = () => {
  return async (dispatch, getState) => {
    const {map} = getState()
    const {data} = await api.post('/routes', {
      start: map.userStartLocation.location,
      destination: map.userDestinationLocation.location,
      passengers: map.personCount,
    })
```

[7] Redux Thunk: https://github.com/reduxjs/redux-thunk (accessed 12.03.2019)

```
    dispatch({
      type: FETCH_ROUTES,
    })
    dispatch({
      type: UPDATE_ROUTE_INFO,
      payload: data[0],
    })
    dispatch(changeOrderState(OrderState.ROUTE_SEARCHED))
  }
}
```

Here, we first read values from the actual `map` state and send an `HTTP POST` request to the `/routes` endpoint of the backend. The `api` object is imported from `/src/lib/api.js`, which has been described earlier. The request contains the `userStartLocation`, `userDestinationLocation` and `personCount` values of the map state. After the response has been received, several actions are dispatched.

1. `FETCH_ROUTES` changes nothing in the state, it's just there to signal that routes have been fetched.
2. `UPDATE_ROUTE_INFO` updates the state's `routeInfo` value according to the HTTP response.
3. Finally, the `CHANGE_ORDER_STATE` action is dispatched to set the `orderState` value to `ROUTE_SEARCHED`, so that the `MapScreen` changes its appearance and shows the `PlaceOrderButton`.

Using the map state and dispatch functions

The redux state can be read by any component using the `connect` function coming with the `react-redux` package, but before we can do this, the React Native root component has to be wrapped inside the `Provider` component provided by `redux`. This happens in `/src/init/App.js`:

```
<Provider store={store}>
    <Root>
      <AppNavigator />
    </Root>
</Provider>
```

The `store` variable in turn contains the `rootReducer` which is exported from `/src/init/rootReducer.js` and combines our three reducers (see Appendix A, Listing 7.3). With this setup, it is possible to read values from the redux store and execute dispatch functions from any React component, no matter on which hierarchy level these components are. For instance, the `SearchRoutesButton` connects to the store as follows:

```
import {connect} from 'react-redux'
...
export default connect(
  state => ({
    userStartLocation: state.map.userStartLocation,
    userDestinationLocation: state.map.userDestinationLocation,
  }),
  dispatch => ({
    fetchRoutes: () => dispatch(fetchRoutes()),
  })
)(SearchRoutesButton)
```

The first parameter of the `connect` function maps a redux `state` to React component props. In this case, `userStartLocation` from the redux `map` state is mapped to the components' prop `userStartLocation`, so it can be accessed via `props.userStartLocation`. The same applies to `userDestinationLocation`. If the redux state is changed from anywhere else in the application (by any other component), this `SearchRoutesButton` component will automatically receive those changes.
The second parameter of the `connect` function maps `dispatch` functions to React component props. In this case, the earlier described `fetchRoutes()` function is imported and mapped to the component's prop `fetchRoutes`, so it can be executed via `props.fetchRoutes()`, which is done when the button is pressed. The relevant code can be found in Appendix A, Listing 7.4.
In our case, `fetchRoutes` is not accessed via `props.fetchRoutes` directly. This is because the functionality has been outsourced into another internal function which then executes `props.fetchRoutes()`. It has been outsourced to implement additional error handling, which will be explained in the next section.

4.2.5 Error Handling

Author: S. Lienau

As earlier described in the API section, the `response interceptor` is used to prepare and throw an `error` object, which can then be used by the calling function. For instance,

the `SearchRoutesButton` component calls the `fetchRoutes()` function, which executes a HTTP request. An error is thrown in case the request was not successful. The relevant code can be found in Appendix A, Listing 7.4.
If the HTTP request fails, the `catch` block will be executed. There is one of two possible error messages shown, depending on the `error.code`. `404` means that no routes could be found (this is specified in our API). If this is the case, the message *No routes found.* concatenated with `error.message` will be shown in a `Toast`. A `Toast` is a small notification which appears at the bottom of the screen by default. For error messages a `danger Toast` is used, which has a red background. In contrast to this, a `success Toast` with a green background is displayed if the action is successful, e.g. if a route is successfully booked or canceled. Default toasts are set in `/src/lib/toasts.js`. Default settings such as duration or position are set there, so the behaviour could easily be changed at this single place and would apply for all default toasts.

4.2.6 Screens

Author: D. Bosin

In this subsection the individual screens as well as their structure and responsibility are discussed in greater detail. Each view is represented by the `index.js` which contains all the necessary components and functionalities. Views that involve a little more logic and are more complex were outsourced in sub folders, such as components or assets for images, to keep the code as clean as possible. Each screen is structured as follows. Internal class states will be defined at the beginning of the class, followed by the functions. Then, the screen is rendered via the render function. Finally, the screen (or the class) is exported via the export function, so that it can be reused by other classes, if necessary.

LoginScreen

As the name suggests, the `LoginScreen` is responsible for logging in a user. Each registered user can log in using their username and password. After successful login, they will be redirected from this view to the `MapScreen`, which will be discussed in the following in more detail.

MapScreen

The `MapScreen` is the heart of the app and the first screen you see right after logging in. Starting from this screen, all other screens can be reached via buttons or input fields.

This makes the map screen the most complex of them all. Because of this complexity several sub folders have been created for different components. Part of the final results are shown in the Tables 5.1 and 5.2 in chapter five.
The `MapScreen` folder contains the sub folder `BottomButtons`. In it, all bottom buttons have been outsourced as an independent component. A look into the `index.js` shows that, depending on the current `MapState`, different buttons are loaded and displayed. As a result, the `BottomButtons` class is responsible to display corresponding buttons including its functionality, depending on the current `MapState`. For instance, if the current `MapState` is `SEARCH_ROUTES`, then the `BackButton` and the `SearchRouteButton` will be displayed. By pressing the `SearchRouteButton` the `MapState` will switch to `ROUTE_SEARCHED` and the `ClearRouteButton` and the `PlaceOrderButton` will be display, as shown in Listing 7.5. For the `TopButtons` the behavior is similar.
Another subfolder is `Info`. Both ordering information and route information have been defined in this folder. While the route information includes the route from the current location to the start VBS, the route from the start VBS to the destination VBS, and the route from the destination VBS to the destination, the order information only includes the route from the current location to the start VBS and from the destination VBS to the final destination itself. The route information should offer the user the opportunity to gain an insight into the upcoming journey. The route information is only visible and selectable prior to ordering, whereas the order information is visible after the order has been confirmed and after leaving the van, see Figures 4.1 and 4.2.
The last sub folder is `SearchForm` and represents the input form for the start and end point. In addition, the probable departure time and arrival time at the respective virtual bus stops as well the duration of the ride are displayed, see top of Figure 4.1. When entering start and destination, you will be forwarded to the `SearchScreen`.

SearchScreen

The `SearchScreen` is responsible for the search of the start and destination positions. The search uses the Google Places API to give the user the possibility of searching for the names of the locations instead of specifying latitude and longitude.

AccountScreen

The `AccountScreen` is set up as a list and uses `ListItems` for the content. The `AccountScreen` contains address information as well as account details, such as the username and email address. In addition, you can see your current score in the loyalty

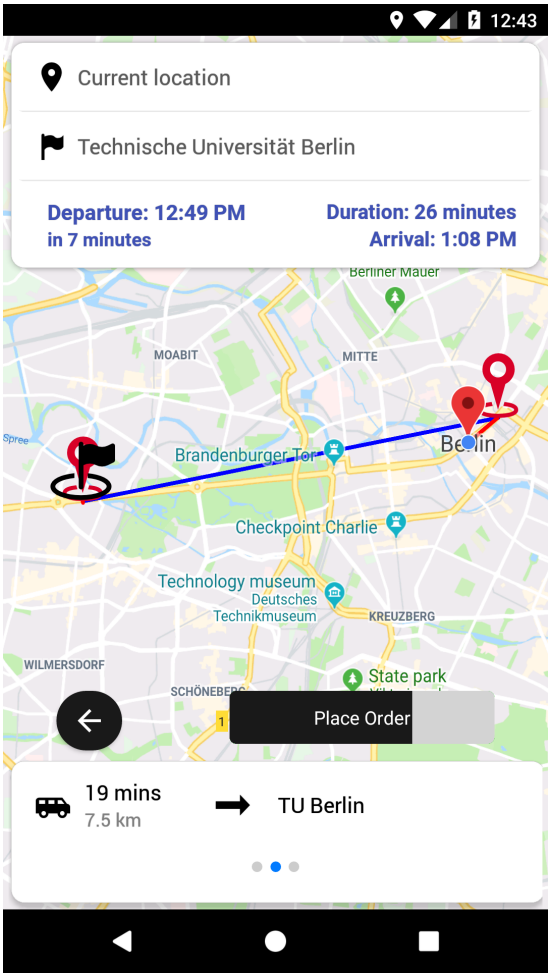


Figure 4.1: Route Information

Figure 4.2: Order Information

program and can access the loyalty leader board. You can also immediately see in the order section how many kilometers you have already driven with the van and how much CO2 you have saved in total due to using the van service. An overview of the last rides is accessable via the order history menu item. You will be redirected to the `OrderHistory` screen.

OrderHistoryScreen and OrderDetailsScreen

The `OrderHistoryScreen` consists of a list of recent rides. Further details on the individual past journeys, such as the distance covered, loyalty points, CO2 savings and distance, are obtained by pressing on one of these rides as shown in Table 5.7.

RideScreen

The `RideScreen` is primarily intended for entertainment while riding the van. For example, it shows fun facts, as depicted in Table 5.4. So that the user always knows where he is with the van, he has the option of using the map button to display a current map section with the current position.

4.3 Backend

Authors: P. Ratz, A. Dittmann

The following section describes the structure of the backend and its specific components. Please refer to the API document in the Appendix 7.2 section of this report for all the API endpoints that are given by the backend as well as their descriptions.
The project folder structure follows a rather standard structure of server-side NodeJS projects and can be viewed in the appendix of this report. The core idea behind the structure is to have minimal logic in within the routes. If need arises for any slightly more complex operation the logic for this operation is then outsourced to a respective service file.

4.3.1 Ordering Process

As mentioned in the introduction of section 4, rather than giving an in detail explanations of the different backend components we will instead introduce them as part of the ordering process. The ordering process is implemented in a series of many smaller processes which are outlined in the following and additionally depicted in Figure 4.3.

Route Search The ordering process starts with the user sending an HTTP GET/POST request to the backend via the mobile application. The request contains various information needed for the ordering process, the most important being the desired start and destination location of the user. When the backend receives the request, it will begin with the processing. The first step is to find the two virtual bus stops that are closest to the user's desired start location (start VBS) and destination location (end VBS). The distance measure we use is therefore the raw linear distance. Having determined both the start and end VBS we continue with finding the best possible van.

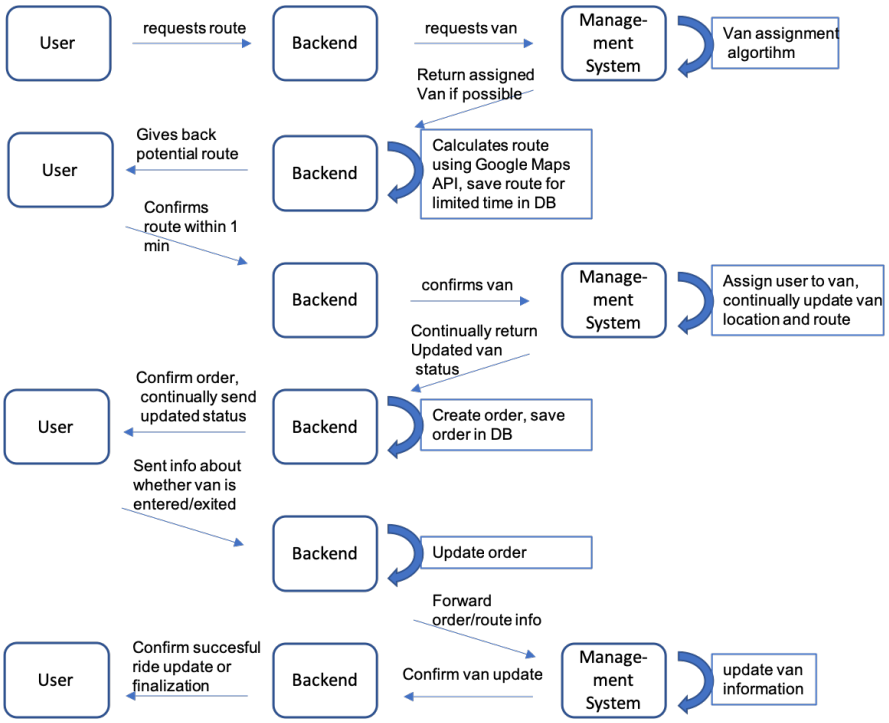


Figure 4.3: Ordering Process

Van selection For simplicity we calculate the van that reaches the start VBS the fastest as the best possible van. This way we also minimize the user's waiting time. For each available van we determine the duration it needs to reach the start VBS. A van is available if it is not currently *locked* and has no passenger. For determining the duration we use the Google Maps Directions API: For each van we issue a request for a route from the current van position to the start VBS. From the result we obtain the duration of the route. We then select the van with the lowest duration and *lock* it in order to prevent that the same van is assigned to more than one order. The actual van assignment algorithm can also be viewed in Appendix 7.3.

Route calculation Having selected the best possible van we continue with the route calculation. We determine the route from the van's current position to the start VBS and the route from the start VBS to the end VBS. We also determine the footpath the user has to take to reach the start VBS. These routes are all determined by using the Google Maps Directions API. The routes are then packed into a *Route* object and sent back to the user as an HTTP response. The response not only contains the routes but also other data, e.g. the the start time of the ride, the duration of the ride or the length

of the footpath. See the API documentation for a full list. The start time of the ride is particularly important because it acts as the van's departure time. The ride start time is equal to the later of either the van's arrival time at the start VBS or the user's arrival time at the start VBS.

Order confirmation After receiving the HTTP response from the backend the user has one minute to confirm the order. If the user does not do so the *locked* van will be released. To confirm the order the app sends another HTTP request to the backend. On receiving it the backend will check for the one minute threshold. If it is not exceeded the order will be confirmend by the backend. The backend indicates this to the user by responding with an *Order* object. The *Order* object is linked to the *Route* object and contains additional data like an updated start time. This order object is then the user's active order.

Order cancellation After the user has confirmed an order the user is still able to cancel the order as long as the user has not entered the van yet. We wanted to give the user some flexibility and also think that an option to cancel the order provides a positive user experience.

Entering and leaving the van As soon as a user confirms an order the assigned van will begin driving to the start VBS. When the van reaches the start VBS the van can be entered by the user, if the user is allowed to. For this purpose we make use of geofencing: The user has to be in a perimeter of 15 meters around the start VBS to be allowed to enter the van. If the user does not enter the van within ten minutes after the planned ride start time, the user's order will be automatically cancelled. After entering the van the van will continue driving to the destination VBS where the user can then exit the van and terminate the user's active order.

4.3.2 Management System

Since we were developing a prototype for an autonomous van service but did not have access to real vans we decided to develop a mock-up to simulate these vans. The management system can be seen as a test utility giving us the opportunity to test many of our app's most important functionalities, e.g. the ordering process, the ride sharing or the loyalty program. The management system acts as a service that provides us with all information that in a real-world scenario real vans would provide us with. This includes plain data like the current van position and also whole functionalities like simulating

the van rides or entering and leaving a van. We separated our management system into three components, each of them providing different functionalities: `SimulatorService`, `HandlerService` and `RequestService`.

The `SimulatorService` is responsible for simulating the van rides. When an order is placed by a user, the routes that were calculated in the ordering process (see subsection 4.3.1 for details) are assigned to the respective van. The `SimulatorService` then simulates the van driving these routes by continuously updating the position of the van. This is done with the help of the responses received from the Google Maps Directions API. Each response contains various information about the route, e.g. the duration of the route or the length of the route. The response also contains an array of so called *steps* which represent single way points along the route. Each step contains the following information:

- The start location of the step (or way point), as latitude and longitude
- The end location of the step (or way point), as latitude and longitude
- The duration of the step (or way point) in seconds
- The length of the step (or way point) in meters

We use this information to subsequently move the van along the steps. Since the steps only contain a start and end location we interpolate the locations within the steps to provide a more fluid simulation.

The `HandlerService` takes care of all actions that handle the van's state: users entering or leaving the van, users cancelling their current orders (which may result in a need to recalculate the route the van has to drive) or resetting the van after a ride is completed. The `RequestService` provides the functionality to request the best van for a user's route request. This involves basically two steps: Finding all vans that are potential candidates for executing the ride and then determining which is best of these vans. As already stated in subsection 4.3.1, the best van is always the one that is fastest at the user's starting virtual bus stop. Finding the potential vans in the first place is the more complex part of requesting a van. For example, we have to calculate how long the vans would need to pick up the user and how long the ride would take. We have to manage that a user's best van cannot be used by others as long as the user is in the ordering process. Our ride sharing approach is also integrated into that part. Hence, we also have to take care of the conditions and limitations of our ride sharing concept when finding potential vans.

4.3.3 Logging

The backend uses various logging channels. to manage this we use the package *winston*. Winston enables us to specify the logging output of our application depending on the environment in the service `WinstonLogger.js`: If the backend is run locally (development), the logging output is expressed in the console. If the backend is run from the designated production environment the logging output is saved as json in the MongoDB. While this can sometimes make the logs in the database hard to read, this is excellent preparation for further monitoring services such as Logstash in combination with ElasticSearch that will facilitate finding errors and solving problems in the future. Winston also allows us to group logs by level and use diffferent database stores for these levels. We mostly use the two main levels which are Info and Error.

5 Outcome

Author: M. Möck
Reviewed by: S. Lienau

The following chapter addresses the product in form of a mobile application, which was developed. The user stories were particularly focused, because of their important role in the development process.
The following shows the user stories that have been completed during the project and the corresponding features, which fulfill the customer's needs.
The user stories *5. Intuitive Access* and *14. Access Features Intuitively* are not listed down below, because these user stories can not be presented based on an explicit feature, but were considered as important guidelines for the user experience.

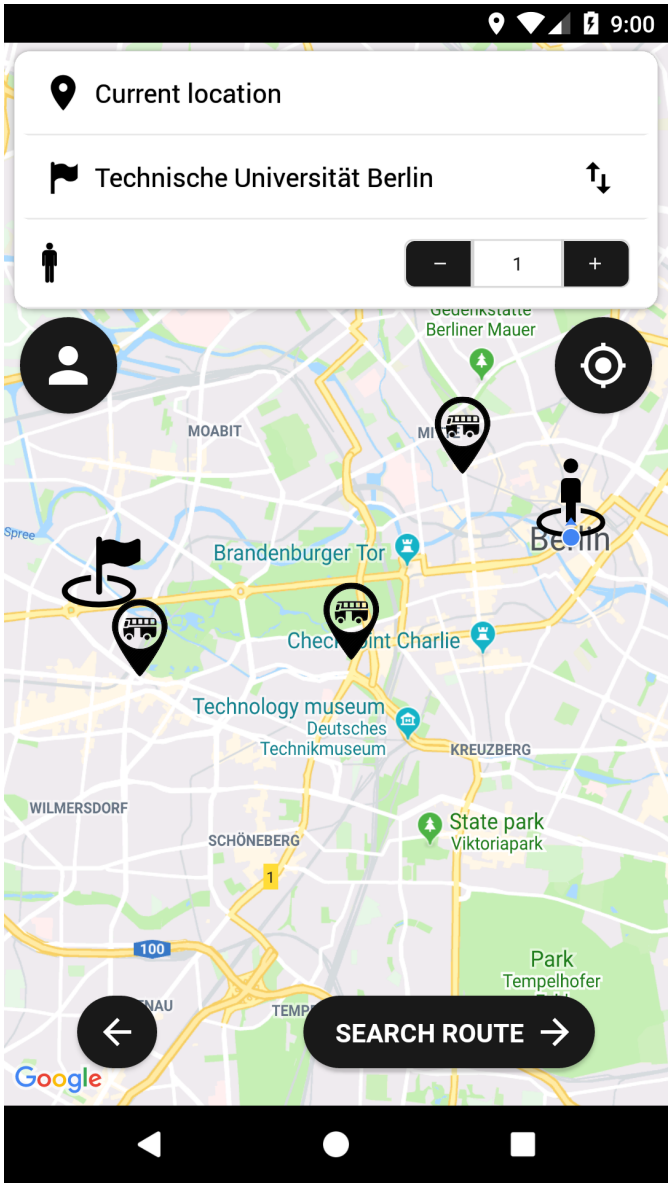
User Stories and Feature	Application View
2. Suggested Route • Search mask for searching a route • By tapping the arrows start and destination can be swapped 38. Ride together • A passenger can book a van for himself or for additional passengers by increasing the passenger counter	

Table 5.1: Suggested Route, Ride Sharing

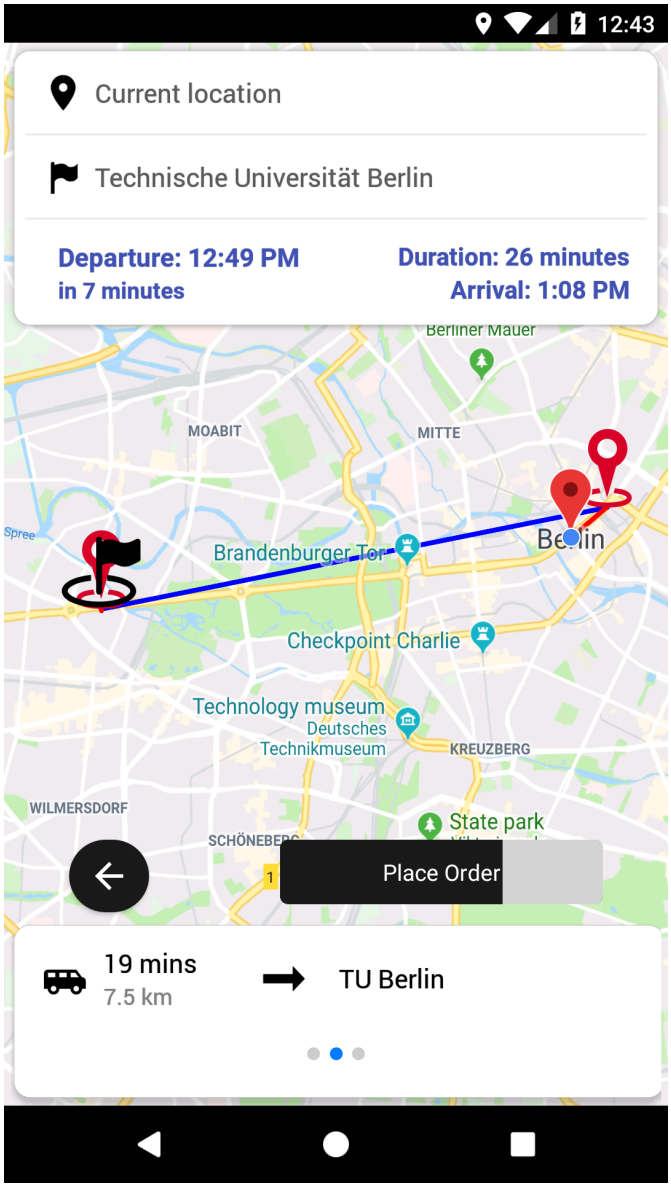
User Stories and Feature	Application View
2. Suggested Route • Route suggestion based on the searched route 3. Time of Arrival • Time of arrival will displayed along with the duration of the whole journey 6. Waiting Time • Time at which the van will pick the passenger up (departure) is displayed before the passenger places the order 7. Order and VBS • `Place Order` button for confirming the order • Passenger has one minute to place the order before the order expires	

Table 5.2: Time of Arrival, Waiting Time

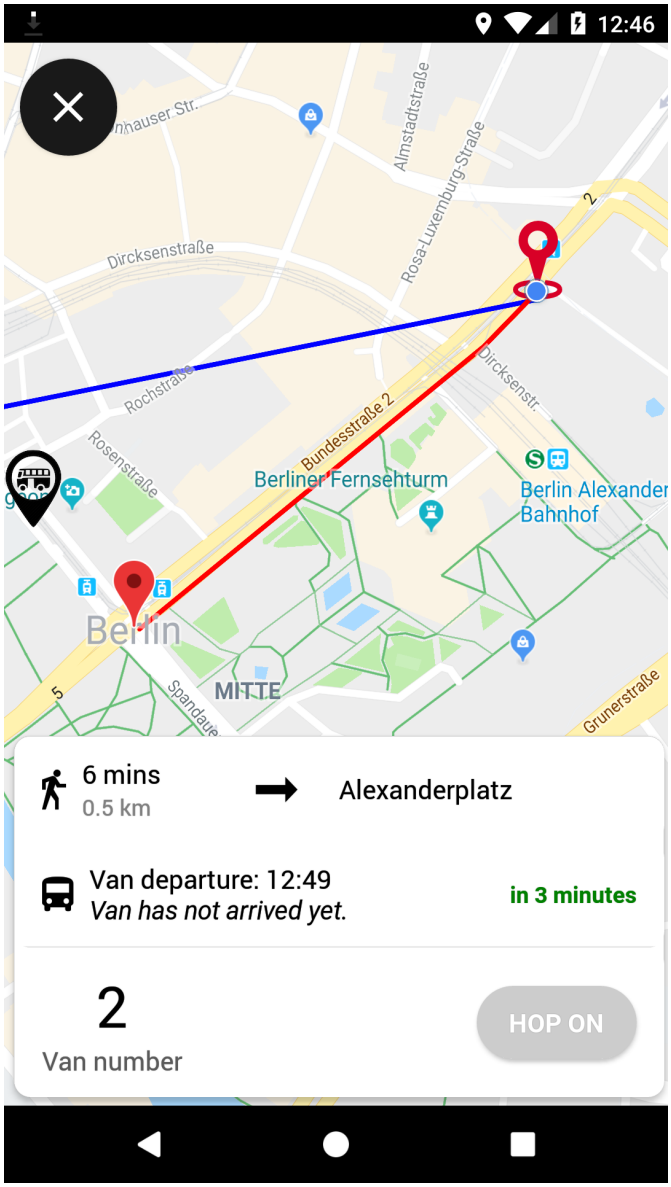
User Stories and Feature	Application View
4. Current Location • The current location of the user is displayed on the map at any time 8. Shortest Footpath to VBS • The shortest route to the VBS is displayed 9. Van Location • Live van location is displayed at any time 10. Identify Van • Van can be identified by individual van number 14. Cancel Active Order • By pressing the `X` button in the upper left corner, the passenger can cancel his active order 36. Confirm Boarding • Geo-fence: `Hop On` button is selectable by the time the passenger and van are close to the VBS	

Table 5.3: Shortest Footpath to VBS, Van Location, Identify Van

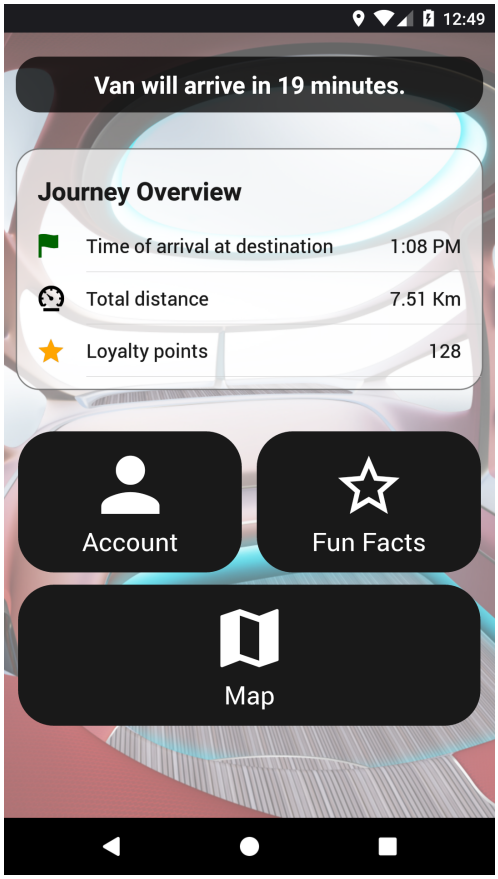
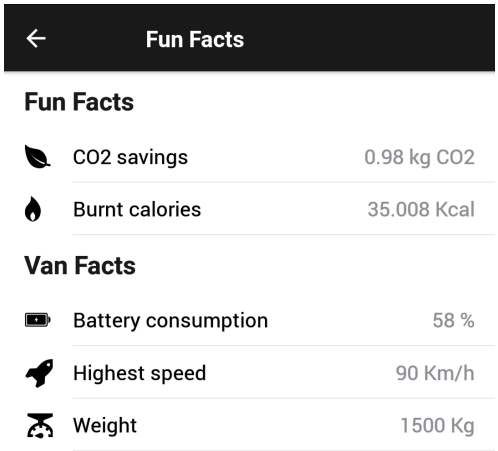
User Stories and Feature	Application View
11. Get Information • All relevant information for the passenger are displayed during the ride 12. Exit Time • Live countdown of van arrival time	
25. CO2 Savings • The Fun Facts section displays additional information and the CO2 savings for this ride	

Table 5.4: Get Information, Exit Time, CO2 Savings

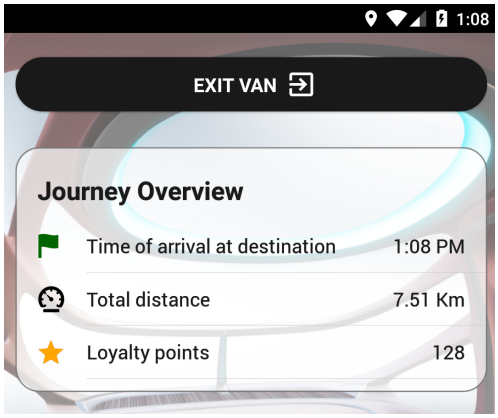
User Stories and Feature	Application View
12. Exit Time • `Exit Van` button appears after the van reaches the destination VBS	
13. Shortest Footpath (Exit) • The shortest route from the VBS to the final destination is displayed • The Journey is completed after pressing manually the `Stop Journey` button or automatically after reaching the final destination (geo-fence)	

Table 5.5: Exit Time, Shortest Footpath

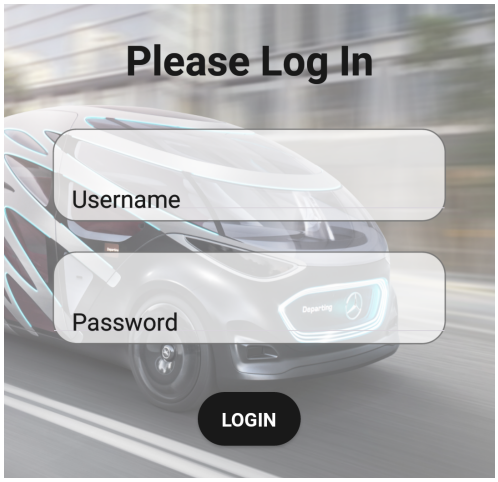
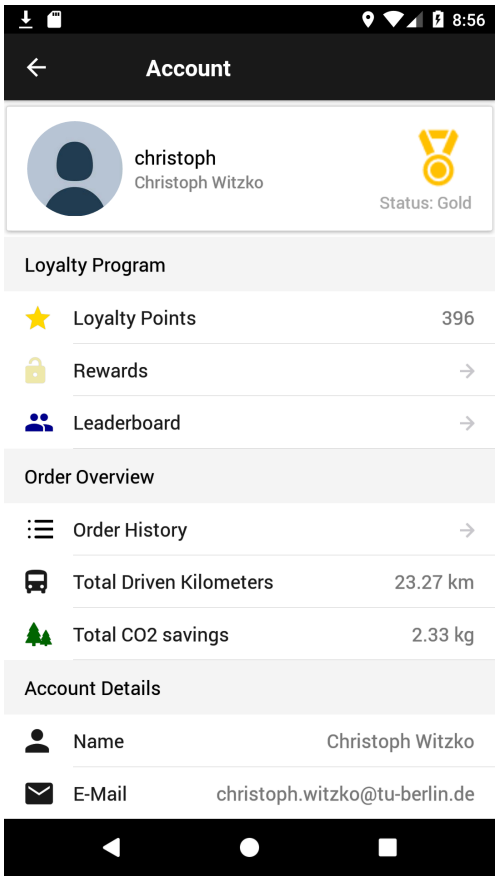
User Stories and Feature	Application View
18. Permanent Login • After the passenger has logged in once, he is permanently logged in	
Account • The account view shows all relevant information regarding the user	

Table 5.6: Permanent Login, Account View

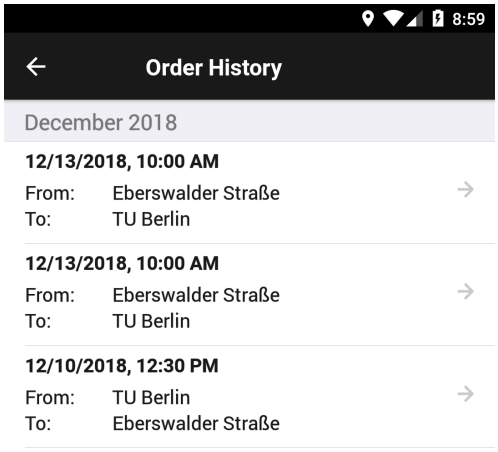
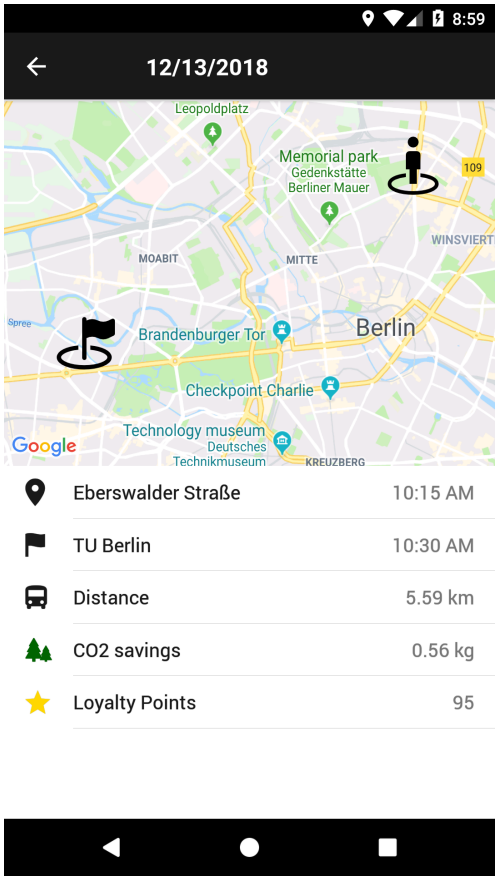
User Stories and Feature	Application View
15. Past Rides • The passenger can access his order history over the account view	
• The passenger can also access detailed information about the recent orders by tapping on them	

Table 5.7: Past Rides

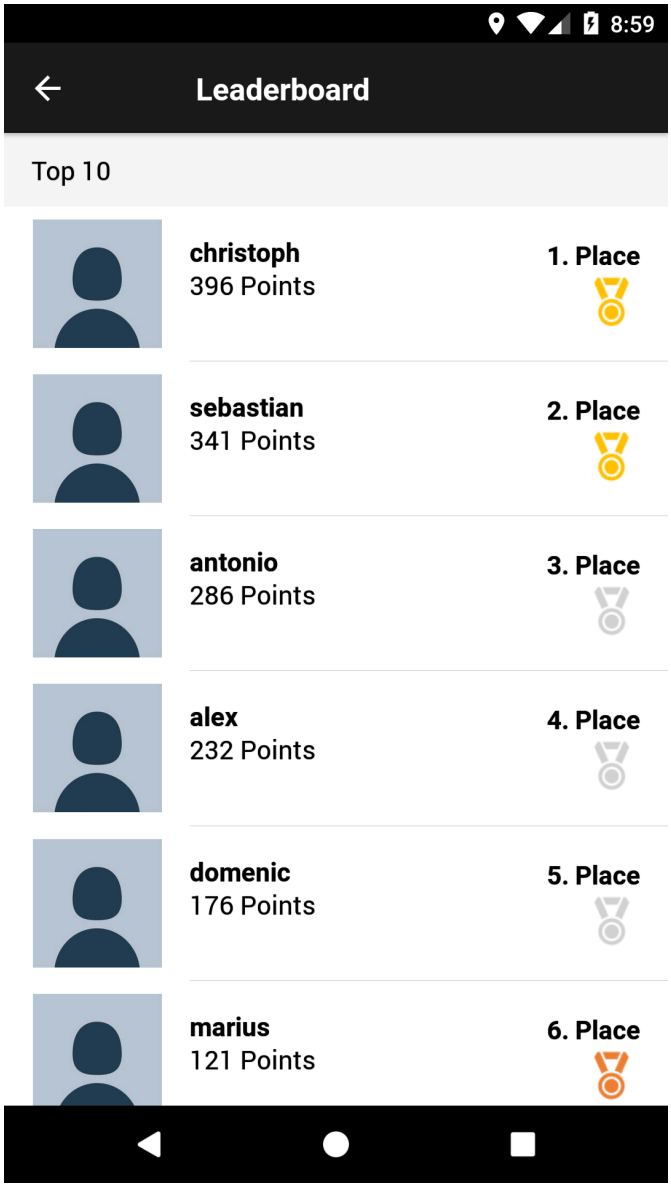
User Stories and Feature	Application View
26. Loyalty Points • The passenger can access the loyalty points leaderboard over the account view • The golden, silver or bronze medal shows the loyalty status of the users • The loyalty points for each ride are declared in the in ride view during riding the van • Further information regarding the loyalty program are described in section 3.3	

Table 5.8: Loyalty Points

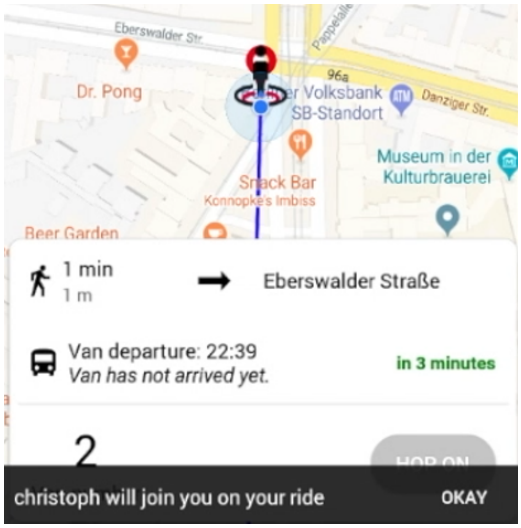
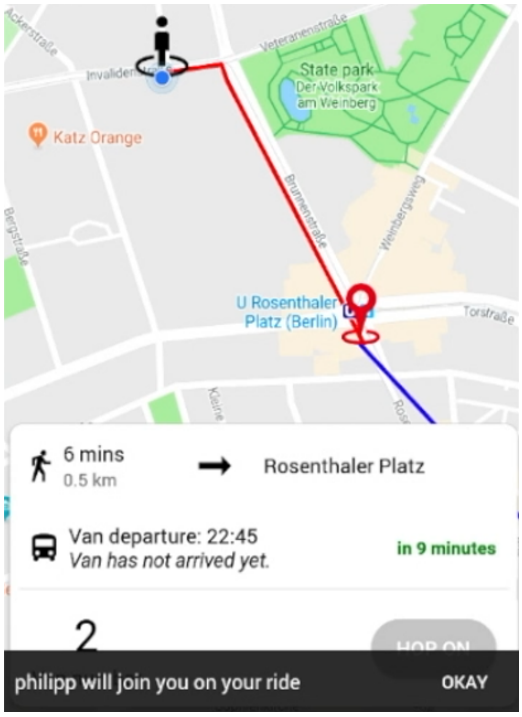
User Stories and Feature	Application View
37. Ride Sharing • Multiple passengers share a ride, if their route resemble one another • The passengers get a notification as soon as their ride will be shared with an other passenger • A detailed description of the ride sharing concept is shown in section 3.2	Passenger Philipp:  Passenger Christoph: 

Table 5.9: Ride Sharing

User Story Coverage

Author: M. Möck
Reviewed by: P. Ratz

After presentation of the completed user stories, this section continues with the evaluation of the overall user story coverage in two subsections. The first shows the already described completed user stories in Table 5.10. The second table captures the aborted user stories as well as the ones that were never started in Table 5.11.

Covered User Stories

The following paragraph considers the user stories, which are completed. The determined MVP, consisting of the user stories with the highest priority, was entirely covered. Table 5.10 shows the whole list of completed abbreviated user stories. The uncovered user stories are discussed in the next section.

User Stories	**User Stories**
1. From airport to office	12. Exit information
2. Suggested route	13. Shortest footpath (exit)
3. Time of arrival	14. Cancel active order
4. Current location	15. Past rides
5. Intuitive access	17. Access features intuitively
6. Waiting time	18. Permanent login
7. Order and VBS	25. CO2 savings
8. Shortest Footpath (VBS)	26. Loyalty points
9. Van location	36. Confirm boarding
10. Identify van	37. Pooling
11. Get information	38. Ride together

Table 5.10: Completed User Stories

Uncovered User Stories

The following section considers the user stories, which have not been completed. These user stories were rejected during the development process based on their priority and based on whether they relate to any important feature. After examination of the uncompleted user stories we found that they can be grouped into three functional

categories. Table 5.11 lists the uncompleted user stories within their corresponding categories. These categories subdivide the referenced user stories and are further explained in the sections *Productivity, Social, Customizability.*

After the MVP was finished, further options that could be added to the application were discussed. After consultation with the project owner the decision was made to focus on development and implementation of a ride sharing concept. Consequently, the user stories, which did not match with the determined realignment, were rejected. The main reason for the rejection of these user stories was the time pressure that came with the given deadline for implementation. We identified three categories of user stories that were not covered.

Productivity The productivity features were prioritized lower, because they only create additional value for the application, if an advanced basis is already existing. Furthermore, the *instructive tour (31)* became obsolete, since the ordering procedure as well as the whole application flow is self-explanatory. The *route suggestions (33)* user story was sorted out, because the implementation would require integration of machine learning algorithms, which are highly complex and would require a lot of resources.

Social The features that are related to social aspects were created with the intention to bring people together during the ride. This idea was based on the intention of long-term customer loyalty. The customer will keep using the application if they kept a positive memory of their van ride. Nevertheless, the social features were ultimately rejected, because the decision was made to achieve long-term customer loyalty by means of a loyalty program. The detailed specification of the loyalty program is shown in section 3.3.

Customizability The customizability features enable the customer to customize the application and their routing preferences according to their needs. Like the productivity features, the customizability features only create additional value, when the application is already in a mature state. Furthermore, the additional value that these feature would yield was assumed to be even less than the one of productivity features. This assumption was made, because most customers would prefer to save time, which provide the productivity features, to the possibility to customize the application.

Productivity	**Social**	**Customizability**
16. Electronic receipt	23. Share ride on social media	20. Notification settings
19. Calender conjunction	27. Social media conjunction	21. Route shortcut
22. Pre-order van	28. Fellow passenger	29. Visibility of profile info
31. Instructive tour	34. Games	30. Route preferences
33. Route suggestions	35. Game Results	

Table 5.11: Categories of sorted out User Stories

The rejected features mentioned above were evaluated based on the weighting of criteria, which also reflect the opinion of the project owner Daimler Vans, the ISE Chair at TU Berlin, the perspective of the van riding customer as well as our personal preferences. Because it was not possible to carry out a customer survey in context of this project, the van riding customers were portrayed by the Personas *Sarah, David* and *Wilbert* mentioned in section 2.4. Despite the reasons to eliminate these features, all of them are suitable additions for the application and can be considered possible starting points for future projects.

6 Summary

Author: A. Dittmann
Reviewed by: P. Ratz, A. Schürer

Within this project we successfully developed a mobile application for an autonomous van. We began with an extensive requirements engineering process where we developed a set of general user stories to identify the fundamental requirements our product has to fulfill. By creating personas we derived further user stories that are serving special target audiences. This procedure allowed us to view on the service from different directions and, thus, to provide a comprehensive list of user stories that fulfill the needs of most of the users. From the set of general user stories we identified the ones that were crucial for a minimal app usage. To these belong the account management including the log in functionality, the whole ordering process as well as the integration of the Google Maps service into our product. The remaining user stories were then prioritized according to the importance. This allowed us to efficiently plan our sprints in the implementation phase since we could directly determine which user stories were the most important to be done next. In retrospect, the requirements engineering was successful but to some extent also troublesome. None of us had great experience in creating user stories and, hence, we all did independent research on this field. In that research we could not identify one valid definition of how user stories have to be phrased, which resulted in different approaches to formulating them. It took some effort to find a common approach and phrase homogeneous and satisfying user stories.

The requirements engineering process was followed by the implementation phase. We scheduled our available time into sprints of a length of two weeks. In each sprint we selected the most important user stories that we planned to finish in that particular sprint due to our prioritization. This was an overall successful approach since we always had an appropriate workload during the sprints and almost always finished all of the planned tasks: In one sprint we did not accomplish to solve all tasks in the scheduled time. But since these remaining tasks were of only small extent, we were able to integrate them into our next sprint without affecting our overall project progress.

We had finished the MVP in our planned schedule in mid January. The user of the app could log in with his own account and access his account information in a separate view. Furthermore, the whole ordering process was implemented and ready to use. A user could search for a route with an arbitrary start and destination location. Search suggestions were also implemented in the route searching process. The resulting route was then displayed on a Google Maps view along with the time of arrival and departure, the duration of the route as well as the length of the footpath to the VBS where the van would depart. Based on that information the user can order the ride. The app's state also included the possibility to cancel the order as long as the user had not yet entered the van. With this feature we wanted to give flexibility to the user of the app. Here, we not only want to appreciate the "visible" parts in the mobile app, but also mention the more "hidden" parts. We developed a sophisticated, detailed and well-documented API for communication between the frontend and backend that was much helpful during the development process. We also put effort into a good manageable project structure. For that, we always reflected about our folder and file structure and we included linters, that automatically checked for a homogeneous, consistent coding style before committing to the version control. We integrated automatic deployment and unit tests that would check the backend's functionality before deploying it to prevent an error-prone server state.

Having finished the MVP in our planned schedule, we had one sprint left to implement further features that go beyond the MVP. During our requirements engineering process we discussed various features that could be implemented at that state. Some examples include social features like a Facebook integration or gamification approaches, e.g. being able to play a round of Tic Tac Toe with other passengers in the van, but see section 2.4 for a full list. We developed a comprehensive and sophisticated loyalty program, since this was also one key requirement for the product. As another feature we decided to integrate a ride sharing approach. This was not initially required but we thought that this kind of feature would be really interesting to develop. Besides it is a feature that is unique about our product compared to many other mobility services. Although the development of the ride sharing concept had it pitfalls we still accomplished to integrate it into our final product along with the loyalty program. To further test our mobile app we also built a management system that simulates real vans. With its help we could simulate whole rides from the start to the beginning. This was especially useful since we discovered some bugs that otherwise would have been much harder to find. The management system also confirmed to us that our developed product indeed worked as we

wanted it to work and made it possible to show the working prototype to the project owner.

To sum this up we built a mobile application comprising a well-tested MVP extended with two meaningful features, namely our loyalty program and ride sharing concept. This state also constitutes a good basis to work on further features. We could on the one hand further extend our loyalty program and ride sharing concept. On the other hand there is still room to integrate other, new features into the app. Our list of user stories suggests integrating social features, e.g. integration of social networks, or features for customization and productivity. Since our loyalty program is very flexible many of these features could be integrated into the loyalty program to further motivate the app usage. The last thing we want to suggest is Augmented Reality (AR). Augmented Reality, which was not covered in our user stories, could depict another interesting option to give the user additional information: With AR one could provide an alternative way of visualizing virtual bus stops, which are not present in the real world.

7 Appendix

7.1 Appendix A - Code Listings Frontend

```
api.interceptors.response.use(null, async error => {
  ...
    if (error.message.startsWith('timeout␣of␣')) {
      error.code = 408
      error.message = 'Timeout␣exceeded.␣Please␣check␣your␣internet␣connection.'
      throw error
    }
    // set error message to response body error message if possible
    error.code = _.get(
      error,
      'response.data.code',
      _.get(error, 'response.status', 400)
    )
    error.message = _.get(error, 'response.data.message', error.message)
    throw error
  ...
})
```

Listing 7.1: API response interceptor for error handling

```
const initialState = {
  orderState: OrderState.INIT,
  userStartLocation: null,
  userDestinationLocation: null,
  currentUserLocation: null,
  routeInfo: {
    id: null,
    vanStartVBS: null,
    vanEndVBS: null,
    toDestinationRoute: null,
    toStartRoute: null,
```

```
    vanRoute: null,
    vanDepartureTime: null,
    vanArrivalTime: null,
    validUntil: null,
    vanId: null,
    guaranteedVanArrivalTime: null,
    toDestinationWalkingTime: null,
    userArrivalTime: null,
    guaranteedUserArrivalTime: null,
  },
  visibleCoordinates: [],
  edgePadding: {top: 0.2, right: 0.1, left: 0.1, bottom: 0.2},
  hasVisibleCoordinatesUpdate: false,
  vans: [],
  personCount: 1,
}
```

Listing 7.2: Initial redux map state

```
import {combineReducers} from 'redux'

import account from '../ducks/account'
import map from '../ducks/map'
import orders from '../ducks/orders'

const rootReducer = combineReducers({
  account,
  map,
  orders,
})

export default rootReducer
```

Listing 7.3: Redux root reducer

```
const SearchRoutesButton = props => {
  const fetchRoutes = async () => {
    try {
      await props.fetchRoutes()
    } catch (error) {
      if (error.code === 404)
        Toast.show(defaultDangerToast('No routes found. ' + error.message, 0))
```

```
      else
        Toast.show(defaultDangerToast('Error getting routes. ' + error.message))
    }
  }

  return (
    <DefaultButton
      onPress={() => fetchRoutes()}
      disabled={!props.userDestinationLocation || !props.userStartLocation}
      text="Search Route"
      iconRight="arrow-forward"
    />
  )
}
```

Listing 7.4: SearchRoutesButton component

```
render() {
    let toReturn
    switch (this.props.mapState) {
      case MapState.INIT:
        toReturn = <DestinationButton {...this.props} />
        break
      case MapState.SEARCH_ROUTES:
        toReturn = (
          <>
            <BackButton onPress={() => this.props.resetMapState()} />
            <SearchRoutesButton />
          </>
        )
        break
      case MapState.ROUTE_SEARCHED:
        toReturn = (
          <>
            <ClearRoutesButton
              onPress={() => {
                this.props.clearRoutes()
                this.zoomToMarkers()
              }}
            />
            <View>
```

```
                <PlaceOrderButton
                  routeExpireProgress={this.state.expireProgress}
                  onPress={() => this.placeOrder()}
                />
              </View>
            </>
          )
          break
        case MapState.ROUTE_ORDERED:
          return null
        default:
          return null
      }
      return (
        <>
          <View style={styles.bottomButtons}>{toReturn}</View>
          {[MapState.INIT, MapState.SEARCH_ROUTES].includes(
            this.props.mapState
          ) && <View style={styles.bottomPadding} />}
        </>
      )
    }
```

Listing 7.5: Index.js of BottomButtons shows, that depending of the MapState different buttons are loaded.

7.2 Appendix B - Project Folder Structure

Project Folder Structure - Frontend

Author: S. Lienau

Folder	Description
`/android`	Android native project (Java) which compiles the code into an android native application
`/ios`	iOS native project (Objective-C) which compiles the code into an iOS native application
`/src`	JavaScript source files
`/src/archive`	Source files that are no longer needed, but might be useful for further developments
`/src/components`	React Native components which are used across different screens
`/src/ducks`	Redux reducers and action creators
`/src/init`	Application initialization files
`/src/lib`	Library functions which are needed in different parts of the application
`/src/screens`	The different screens of the App, e.g. Login, Map, Account

In addition, screens can have their own subcomponents which are only used by that single screen and not shared with other screens. In this case those subcomponents are located in the `components/` subfolder, relative to the screen root folder (e.g. `/src/screens/RideScreen/components/`).
Furthermore, screens can also contain non-JavaScript files, such as images. These can then be found in the `assets/` subfolder (e.g. `/src/screens/RideScreen/assets/`).

Project Folder Structure - Backend

Author: Philipp Ratz

Folder	Description
`/bin`	Contains only one file which creates and starts the server and prepares the database
`/models`	Contains the four MongoDB models that describe the database entities and their properties: `Account.js`, `Order.js`, `Route.js` and `VirtualBusStop.js`
`/routes`	Contains all of the routes that can be accessed via HTTP as specified in the API: `/account`, `/activeorder`, `/auth`, `/index`, `/leaderboard`, `/orders`, `/routes`, `/vans` and `/virtualBusStops`
`/scripts`	Contains the two shell scripts `deploy.sh` and `publish.sh` that run as part of the continuous integration
`/services`	Contains the main services that are needed to operate the backend such as the ones creating orders or accounts. The services are structured by functionality in a total of nine files (excluding `/vanServices`).
`/services/vanServices`	Contains the services in relation to all of the van operations such as assigning a van to a ride or recalculating the location of all vans.
`/tests`	Contains all of the various test that were written to test either parts of the backend (unit tests) or it's full functionality.(integration tests)

The main folder level not only includes these folders but also a variety of other service providing files such as `app.js`, which is the entrypoint for clients accessing the backend or the `Dockerfile` which manages containerization.

7.3 Appendix C - Van Assignment Algorithm

Author: Philipp Ratz

Algorithm 1: Van assignment algorithm

Result: Either one specific van or none

Initialize possibleVans = [];

Input: Pool of Vans; passengerRoute

foreach *van in Pool of Vans* **do**
  if *van.waiting = false AND van.potentialUnconfirmedRoute = NULL AND van.currentlyPooling = false* **then**
    if *van.routes.lenght = 0* **then**
      possibleVans.add(van);
    end
    if *(passengerRoute.startVBS = van.routes[0].startVBS OR passengerRoute.endVBS = van.routes[0].endVBS) AND passengerRoute and van's currentRoute would not elongate by more than 10 min* **then**
      possibleVans.add(van);
    end
  end
end

foreach *van possibleVans* **do**
  Calculate time t of van to passengerRoute.startVBS
end

return argmin in relation to t of possibleVans

7.4 Appendix D - REST API

The following pages contain a printout of our REST API. This can also be found in markdown format in the code repository.

API

Author: Sebastian Lienau

Table of Content

Endpoint

The base URL for this APIs endpoint is

`http://3.120.249.73:8080`

Error handling

This API uses standard HTTP status codes to indicate the status of a response.

Error types

Name	Code	Description
Bad request	`400`	The request was unacceptable
Unauthorized	`401`	The request has not been applied because it lacks valid authentication credentials for the target resource.
Forbidden	`403`	The server understood the request, but is refusing to fulfill it
Not Found	`404`	The server has not found anything matching the request URI
Server error	`500`	A technical error occured in the Cloud

Error object

This general error structure is used throughout this API.

Field	Type	Required	Description	Examples
`code`	`Integer`	yes	The error code.	`400`
`message`	`String`	No	(Long) description of the error.	`The server understood the request, but is refusing to fulfill it`

Example

```
{
  "code": 400,
  "message": "Bad query parameter [$size]: Invalid integer value [abc]"
}
```

Objects

Location

This object represents a geographical location.

Field	Type	Required	Description	Examples
`latitude`	`Number`	yes	The north coordinate.	`52.478442`
`longitude`	`Number`	yes	The east coordinate.	`13.405938`

Example

```
{
  "latitude": 52.478442,
  "longitude": 13.405938
}
```

VirtualBusStop

This object represents a virtual bus stop.

Field	Type	Required	Description	Examples
`id`	`String`	yes	The ID.	`0e8cedd0-ad98-11e6-bf2e-47644ada7c0b`
`name`	`String`	Yes	Name of the station, e.g. street name or point of interest name	`Straße des 17. Juni 135`
`accessible`	`Boolean`	Yes	True if the virtual bus stop is currently accessible, false if not.	`true`, `false`
`location`	`Location`	Yes	The location of the VirtualBusStop.	See Location

Example

```
{
  "id": "0e8cedd0-ad98-11e6-bf2e-47644ada7c0b",
  "name": "Straße des 17. Juni 135",
  "accessible": true,
  "location": {
    "latitude": 52.478442,
    "longitude": 13.405938
  }
}
```

Account

This object represents a user account.

Field	Type	Required	Description	Examples
`id`	`String`	yes	The ID.	`0e8cedd0-ad98-11e6-bf2e-47644ada7c0f`
`firstName`	`String`	Yes	The users first name.	`Max`
`lastName`	`String`	Yes	The users last name.	`Müller`
`email`	`String`	Yes	The users email.	`maxmueller@tu-berlin.de`
`username`	`String`	Yes	The username.	`maxiboy123`
`loyaltyPoints`	`Number`	Yes	The users total loyalty points.	`234`
`loyaltyStatus`	`String`	yes	The loyalty program status.	`gold`
`address`	`Object`	No	The users address. Contains `street`, `zipcode` and `city`	*See account example.*
`distance`	`Number`	yes	Total amount of kilometres driven by the user.	`423.34`
`co2savings`	`Number`	yes	Total amount of CO2 savings (in kilogram) for all van rides.	`70.4`

Example

```
{
  "id": "0e8cedd0-ad98-11e6-bf2e-47644ada7c0f",
  "firstName": "Max",
  "lastName": "Müller",
  "email": "maxmueller@tu-berlin.de",
  "username": "maxiboy123",
  "loyaltyPoints": 234,
  "loyaltyStatus": "gold",
  "address": {
    "street": "Salzufer 1",
    "zipcode": "10587",
    "city": "Berlin"
  },
  "distance": 423.34,
  "co2savings": 70.4,
}
```


Order

This object represents an order. It can be created by a user using the `POST /orders` endpoint.

Field	Type	Required	Description	Examples
`id`	`String`	yes	The order ID.	`13cf81ee-8898-4b7a-a96e-8b5f675deb3c`
`accountId`	`String`	yes	User account ID which placed this order.	`0e8cedd0-ad98-11e6-bf2e-47644ada7c0f`
`orderTime`	`Datetime`	yes	Time at which the order was created.	`2018-11-23T18:25:43.511Z`
`active`	`Boolean`	yes	`True` if this order is active (user is still travelling to the ending virtual bus stop). `False` if the user reached the ending virtual bus stop (finished the journey), or if the order was canceled.	`true`

`canceled`	`Boolean`	yes	True if the user canceled the order.	`false`
`vanStartVBS`	`VirtualBusStop`	yes	Virtual Bus Stop where the van departs.	See VirtualBusStop
`vanEndVBS`	`VirtualBusStop`	yes	Virtual Bus Stop where the van arrives.	See VirtualBusStop
`vanEnterTime`	`Datetime`	no	Time at which the user entered the van. `Null` if the order was canceled.	`2018-11-23T18:30:25.000Z`
`vanExitTime`	`Datetime`	no	Time at which the user left the van. `Null` if the order was canceled.	`2018-11-23T18:45:48.000Z`
`vanId`	`Number`	yes	The van which carries the user.	`3`
`loyaltyPoints`	`Number`	yes	The amount of loyalty points for this order.	`200`
`distance`	`Number`	yes	Amount of kilometres the van drives.	`12.4`
`co2savings`	`Number`	yes	Amount of CO2 savings (in kilogram) for this ride .	`2.2`
`route`	`Route`	only if `active` is `true`	The route object for this order.	See Route

Example

```
{
  "id": "13cf81ee-8898-4b7a-a96e-8b5f675deb3c",
  "accountId": "0e8cedd0-ad98-11e6-bf2e-47644ada7c0f",
  "orderTime": "2018-11-23T18:25:43.511Z",
  "active": true,
  "canceled": false,
  "vanStartVBS": {
    "id": "7416550b-d47d-4947-b7ec-423c9fade07f",
    "name": "Straße des 17. Juni 135",
    "accessible": true,
    "location": {
      "latitude": 52.515598,
      "longitude": 13.326860
    }
  },
  "vanEndVBS": {
    "id": "76d7fb2f-c264-45a0-ad65-b21c5cf4b532",
    "name": "Straße des 17. Juni 120",
    "accessible": true,
    "location": {
      "latitude": 52.512974,
      "longitude": 13.329145
    }
  },
  "vanEnterTime": "2018-11-23T18:30:25.000Z",
  "vanExitTime": "2018-11-23T18:45:48.000Z",
  "vanId": 3,
  "loyaltyPoints": 3980,
  "distance": 12.4,
  "co2savings": 2.2,
  "route": {}
}
```


Route

This object represents a travel route from journey start to the final destination of the journey.

Field	Type	Required	Description	Examples
`id`	`String`	yes	The route ID.	`13cf81ee-8898-4b7a-a96e-8b5f675deb3c`
`userStartLocation`	`Location`	yes	Location where the journey starts.	See Location
`userDestinationLocation`	`Location`	yes	Final destination of the journey.	See Location
`vanStartVBS`	`VirtualBusStop`	yes	The VirtualBusStop where the user should enter the van.	See VirtualBusStop
`vanEndVBS`	`VirtualBusStop`	yes	The VirtualBusStop where the user should exit the van.	See VirtualBusStop
`vanDepartureTime`	`Datetime`	Yes	The time the van will depart from `vanStartVBS` .	`2018-12-17T17:20:00.000Z`
`vanArrivalTime`	`Datetime`	Yes	The (expected) time the van will arrive at `vanEndVBS` .	`2018-12-17T17:35:00.000Z`
`guaranteedVanArrivalTime`	`Datetime`	Yes	The guaranteed arrival time of the van at `vanEndVBS` if the ride is shared with other passengers.	`2018-12-17T17:45:00.000Z`
`toDestinationWalkingTime`	`Number`	Yes	The time in seconds the user needs from `vanEndVBS` to `userDestinationLocation` by foot.	`274`
`toStartRoute`	`Object`	Yes	Route from `userStartLocation` to `vanStartVBS` . This is an object returned by the Google	

			Maps API.	
`vanRoute`	`Object`	Yes	Van route from `vanStartVBS` to `vanEndVBS`. This is an object returned by the Google Maps API.	
`toDestinationRoute`	`Object`	Yes	Route from `vanEndVBS` to `userDestinationLocation`. This is an object returned by the Google Maps API.	
`vanId`	`Number`	yes	The ID of the van which drives this route.	`3`
`validUntil`	`Datetime`	Only for `POST /routes` request	Time until this route can be confirmed (create an order out of this route). After this time has elapsed, a new route must be requested, otherwise no order can be created.	`2018-12-17T17:10:00.000Z`

Example


```
{
  "id": "13cf81ee-8898-4b7a-a96e-8b5f675deb3c",
  "userStartLocation": {
    "latitude": 52.516639,
    "longitude": 13.331985
  },
  "userDestinationLocation": {
    "latitude": 52.513245,
    "longitude": 13.332684
  },
```

```
  "vanStartVBS": {
    "id": "7416550b-d47d-4947-b7ec-423c9fade07f",
    "name": "Straße des 17. Juni 135",
    "accessible": true,
    "location": {
      "latitude": 52.515598,
      "longitude": 13.326860
    }
  },
  "vanEndVBS": {
    "id": "76d7fb2f-c264-45a0-ad65-b21c5cf4b532",
    "name": "Straße des 17. Juni 120",
    "accessible": true,
    "location": {
      "latitude": 52.512974,
      "longitude": 13.329145
    }
  },
  "vanDepartureTime": "2018-12-17T17:20:00.000Z",
  "vanArrivalTime": "2018-12-17T17:35:00.000Z",
  "guaranteedVanArrivalTime": "2018-12-17T17:45:00.000Z",
  "toDestinationWalkingTime": 274,
  "toStartRoute": {},
  "vanRoute": {},
  "toDestinationRoute": {},
  "vanId": 3,
  "validUntil": "2018-12-17T17:10:00.000Z"
}
```


ActiveOrderStatus

All fields are *not* required. Only modified values since the last request must be included in the response.

Field	Type	Description
`vanId`	`Number`	The van ID.
`userAllowedToEnter`	`Boolean`	`true` if the user can enter the van (if the user is close enough to the van), `false` otherwise.
`userAllowedToExit`	`Boolean`	`true` if the user can exit the van, `false` otherwise.
`vanLocation`	`Location`	The current position of the van.
`vanDepartureTime`	`Datetime`	The time the van will depart from `vanStartVBS`
`vanArrivalTime`	`Datetime`	The (expected) time the van will arrive at the `vanEndVBS`
`guaranteedVanArrivalTime`	`Datetime`	The guaranteed arrival time of the van at `vanEndVBS` if the ride is shared with other passengers.
`otherPassengers`	`Array`	Array containing usernames (Strings) of other passengers.
`message`	`String`	The status message.
`nextStops`	Array of `VirtualBusStop`	This array contains the next virtual bus stops that the van will drive up in case of pooling.
`nextRoutes`	Array of `Object`	This array contains Google Maps Route objects. Every element of the array contains the route from one virtual bus stop to the next.

Example

```
{
  "vanId": 3,
  "userAllowedToEnter": false,
  "userAllowedToExit": false,
  "vanLocation": {
    "latitude": 52.515598,
    "longitude": 13.32686
  },
  "vanDepartureTime": "2018-11-23T18:20:25.000Z",
  "vanArrivalTime": "2018-11-23T18:30:25.000Z",
  "guaranteedVanArrivalTime": "2018-11-23T18:37:25.000Z",
  "otherPassengers": [
    "sarah",
    "wilbert"
  ],
  "message": "Van has not arrived yet",
  "nextStops": [],
  "nextRoutes": []
}
```


Resources

All request and response bodys are of type `application/json`.

Authorization

Authorization is required for all requests except to the `/login` endpoint. If not authorized, the server will respond with `HTTP 401`.

Authorization Header

Parameter	Value	Required	Description
`Authorization`	`Bearer <TOKEN>`	Yes	The JWT token from the `POST /login` response.

/login

POST /login

Request Body

```
{
  "username": "admin",
  "password": "xyz"
}
```

Response Body

```
{
  "userId": "64e0d993-2867-44cf-872d-9a78a5c212a0",
  "token": "xxxxx.yyyyy.zzzzz"
}
```

/account

GET /account

Get the user account information.

Responses

Code	Body Type	Description
`200`	`Account`	See Account.
`400`	`Error`	Bad request. See Error.
`401`	`Error`	If a wrong username or password has been entered. See Error.

PUT /account

To update user account information.

Body

Type	Required	Description
`Account`	Yes	Containing the updated user account information. See Account.

Responses

Code	Body Type	Description
`200`	`Account`	Containing the updated user account information. See Account.
`400`	`Error`	See Error.

/orders

GET /orders

Get the orders of a user.

Request Query Parameters

Property	Type	Required	Description
`fromDate`	`Datetime`	No	If set, the response will only contain orders which were placed *after* this date.
`toDate`	`Datetime`	No	If set, the response will only contain orders which were placed *before* this date.

Example

- `/orders`
- `/orders?fromDate=2018-01-01T00:00:00.000Z&toDate=2018-10-01T00:00:00.000Z`

Responses

Code	Body Type	Description
`200`	Array of `Order`	Orders which match the request parameters. See Order.
`400`	`Error`	See Error.

Example

```
[
  {
    "id": "13cf81ee-8898-4b7a-a96e-8b5f675deb3c",
    "accountId": "0e8cedd0-ad98-11e6-bf2e-47644ada7c0f",
    "orderTime": "2018-02-23T18:25:43.511Z",
    "active": false,
    "canceled": false,
    ...,
  },
  {
    "id": "13cf81ee-4b7a-8898-a96e-8b5f675deb3c",
    "accountId": "0e8cedd0-ad98-11e6-bf2e-47644ada7c0f",
    "orderTime": "2018-02-24T18:25:43.511Z",
    "active": false,
    "canceled": false,
    ...,
  }
]
```

POST /orders

To create (place) a new van order.

Request

Body

Property	Type	Required	Description
`routeId`	`String`	Yes	ID of the route.

Example

```
{
  "routeId": "d79ab15d-39e8-4817-83d0-ed21d395dded",
}
```

Responses

Code	Body Type	Description
`200`	`Order`	The new created order. See Order.
`400`	`Error`	If the order couldn't be created. See Error.
`404`	`Error`	If the `routeId` wasn't found or if the route has expired. See Error.

/activeorder

GET /activeorder

Get the current active order.

Responses

Code	Body Type	Description
`200`	`Order`	The current active order. See Order.
`400`	`Error`	See Error.
`404`	`Error`	If there is no active order at the moment. See Error.

PUT /activeorder

Update an active order.

Request Body

The request body contains a JSON object with the properties `action`, which describes what should be done/changed in the active order, and `userLocation`, which contains the users current location coordinates (See Location).
Possible `action`-types are:

Value	Description
`startride`	User has entered the van. After this request has been sent, the van should lock the doors and start the ride. `403` Error if the user isn't close enough to the van, so he didn't enter the van.
`endride`	User left the van. `403` error if the van hasn't arrived at the destination virtual bus stop yet.
`cancel`	User wants to cancel the order. `403` error if the order can not be canceled anymore (ride already started or ended).

Example

```
{
  "action": "startride",
  "userLocation": {
    "latitude": 52.123456,
    "longitude": 13.123456
  }
}
```

Responses

Code	Body Type	Description
`200`	`Order`	The updated Order. See Order.
`400`	`Error`	See Error.
`403`	`Error`	If the `action` requested by the user is not allowed. See request body and Error.

GET /activeorder/status

Get information about the current active order.

Request Query Parameters

Property	Type	Required	Description
`passengerLatitude`	`Number`	Yes	The users current latitude.
`passengerLongitude`	`Number`	Yes	The users current longitude.

Example

```
GET /activeorder/status?passengerLatitude=52.123456&passengerLongitude=13.123456
```

Response

Code	Body Type	Description
`200`	`ActiveOrderStatus`	See ActiveOrderStatus.
`400`	`Error`	See Error.

/routes

POST /routes

Get suggested routes from starting point to destination.

Request

Body

Property	Type	Required	Description
`start`	`Location`	Yes	Journey start location. See Location.
`destination`	`Location`	Yes	Journey destination. See Location.
`passengers`	`Number`	No	The amount of passengers that want to take the ride. Default: `1`
`startTime`	`Datetime`	No	Route start time. Current time if not specified.

Example

```
{
  "start": {
    "latitude": 52.512974,
    "longitude": 13.329145
  },
  "destination": {
    "latitude": 52.285946,
    "longitude": 13.317390
  },
  "startTime": "2018-12-15T18:30:00.000Z"
}
```

Responses

Code	Body Type	Description
`200`	Array of `Route`	Array of possible routes from desired start location to destination. See Route.
`400`	`Error`	See Error
`404`	`Error`	If no route was found. See Error

Example


```
[
  {
    "startLocation": {
      "latitude": 52.516639,
      "longitude": 13.331985
    },
    "startStation": ...
    ...
  },{
    "startLocation": {
      "latitude": 52.516639,
      "longitude": 13.331985
    },
    "startStation": ...
    ...
  }
]
```


/virtualbusstops

GET /virtualbusstops

Get nearby virtual bus stops.

- Will only return virtual bus stops whose `accessible` property is `true`.
- Returns an array of virtual bus stops which are *inside of the radius* from the given location.
- Returns an empty array if no virtual bus stops are found inside that area.

Request

Query Parameters

Property	Type	Required	Description
`latitude`	`Number`	Yes	The latitude of the center position
`longitude`	`Number`	Yes	The longitude of the center position
`radius`	`Number`	No	Maximum distance of the virtual bus stops to the location parameter. *Unit: meter* Min: `100`; Max: `10000`; Default: `1000`

Examples

`/virtualbusstops?radius=2000&latitude=52.512974&longitude=13.329145`

`/virtualbusstops?latitude=52.512974&longitude=13.329145`

Responses

Code	Body Type	Description
`200`	Array of `VirtualBusStop`	Array of virtual bus stops which are inside of the radius from the given location. See VirtualBusStop
`400`	`Error`	See Error

Examples

```
[]
```

```
[
  {
    "id": "d79ab15d-39e8-4817-83d0-ed21d395dded",
    "name": "Straße des 17. Juni",
    "accessible": true,
    "location": {
      "latitude": 52.515729,
      "longitude": 13.323373
    }
  },
  {
    "id": "7416550b-d47d-4947-b7ec-423c9fade07f",
    "name": "Straße des 17. Juni",
    "accessible": true,
    "location": {
      "latitude": 52.515598,
      "longitude": 13.326860
    }
  }
]
```

/leaderboard

GET /leaderboard

Get the top 10 users with the most loyalty points.

Response

Code	Body Type	Description
`200`	Array of `Objects`	Every object contains `username`, `loyaltyPoints` and `status` (see example). The array is sorted by the property `loyaltyPoints` (highest first).
`400`	`Error`	See Error

Example


```
[
    {
        "loyaltyPoints": 175,
        "username": "antonio",
        "status": "platin"
    },
    {
        "loyaltyPoints": 125,
        "username": "philipp",
        "status": "gold"
    },
    {
        "loyaltyPoints": 75,
        "username": "alex",
        "status": "silver"
    }
]
```


/vans

GET /vans

Get all available vans and their positions.

Response

Code	Body Type	Description
`200`	Array of `Objects`	Every object contains the properties `vanId` and `location` (see example) .
`400`	`Error`	See Error

Example

```
[
  {
    "vanId": 1,
    "location": {
      "latitude": 52.5197098,
      "longitude": 13.3882533
    }
  },
  {
    "vanId": 2,
    "location": {
      "latitude": 52.50712919999999,
      "longitude": 13.3305731
    }
  }
]
```