BodaBoda - Detailed Design

Group 1

December 2018

Aleksandar acimovic
acimovic.alek@gmail.com

Zacharias Leo
zlo16001@student.mdh.se

Stanislas Pedebearn
spn18013@student.mdh.se

Chingiz Esenbaev
tjv15001@student.mdh.se

Jonathan Major
jmr16009@student.mdh.se

Linus Sens Ingels
lis16001@student.mdh.se

Sireesha Kumari Kulari
ski18005@student.mdh.se

Contents

1 Introduction **5**
1.1 Client . . . 5
1.2 Product . . . 5
1.3 Definitions . . . 5

2 Functional Description **6**
2.1 Use Case Model . . . 6
2.1.1 Actors . . . 6
2.1.2 Use Cases . . . 6
2.2 Information about Guest's Use Cases . . . 8
2.2.1 Register account . . . 8
2.2.2 Log in . . . 9
2.3 Information about Customer's Use Cases . . . 10
2.3.1 Request Trip . . . 10
2.3.2 Cancel Trip . . . 11
2.3.3 Decline Trip . . . 12
2.3.4 Accept Trip . . . 13
2.3.5 Pay for Trip . . . 14
2.3.6 Change preferred Payment method . . . 15
2.4 Information about Guest's and Driver's Use Cases . . . 16
2.4.1 Modify Account . . . 16
2.4.2 Log out . . . 17
2.5 Information about Driver's Use Cases . . . 18
2.5.1 Finish Trip . . . 18
2.5.2 Get navigation help . . . 19
2.5.3 Browse pending Trips . . . 20
2.5.4 Accept Trip Request . . . 21
2.5.5 Set base price per distance . . . 22
2.5.6 See transaction history . . . 23
2.5.7 Cancel Trip early . . . 24

3 Software Architecture **25**
3.1 Overview and Rationale . . . 25
3.2 Collaboration between parts . . . 26
3.3 System decomposition . . . 27
3.4 Data persistence . . . 29

4 Detailed Software Design **30**
4.1 Front-end Design . . . 30
4.1.1 Structure . . . 30
4.1.2 Class Structure . . . 31
4.2 Back-end Design . . . 32
4.2.1 Structure . . . 32

4.2.2 Class Structure 33

5 Graphical Interfaces **35**
5.1 User Interface 35
5.1.1 Guest Interface 35
5.1.2 Driver Interface 36
5.1.3 Customer Interface 38

References **40**

6 APPENDIX A: Swagger Documentation **41**

List of Figures

1 Use Case Diagram 6
2 Register account sequence diagram 8
3 Log in sequence diagram 9
4 Request trip sequence diagram 10
5 Cancel Trip sequence diagram 11
6 Decline Trip sequence diagram 12
7 Accept Trip sequence diagram 13
8 Pay for trip sequence diagram 14
9 Change payment method sequence diagram 15
10 Modify account information sequence diagram 16
11 Log out sequence diagram 17
12 Finish trip sequence diagram 18
13 Navigation help sequence diagram 19
14 Browse pending trip sequence diagram 20
15 Accept trip request sequence diagram 21
16 Set base price sequence diagram 22
17 Transaction history sequence diagram 23
18 Cancel Trip early sequence diagram 24
19 Component Diagram 27
20 Database view diagram 29
21 Front-End Class Diagram 31
22 Back-End Class Diagram 33
23 Guest interfaces 35
24 Driver interfaces part one 37
25 Driver interfaces part two 37
26 User interfaces part one 39
27 User interfaces part two 39

1 Introduction

This document covers the detailed design of the BodaBoda project, the application being developed for the company named Okapi Finance. The application provides a taxi service for motorcycle drivers in East Africa.

1.1 Client

Okapi Finance is offering financial services mainly in Africa with a mission to bank the unbanked. Okapi's services increases accessibility for financial services. They believe that everyone should be able to access financial services regardless of where they live and their status. Transparency, security, and compliance with financial regulations are key for their operations and they believe in building the Okapi network through partnership with financial institutions and other existing structures. Their goal is to be the pan-African fintech (financial technology) solution [1].

1.2 Product

The product produced is an application called BodaBoda, an application for motorcycle taxis in East Africa. The customers request trips by stating where they want to go, the drivers then see the pending requests in a list where they can choose a trip they want to take. The app will tell the driver where to go to find the customer and it will track the driven distance and calculate a price for the customer. The app can also help the driver navigate to the final destination. The customer can pay in her app and the driver will be able to track her received payments and see how much she has earned on a daily, weekly or monthly basis.

1.3 Definitions

- **Guest** - An unregistered user of the application.
- **Customer** - The customer that requests a trip by a taxi driver using the application.
- **Driver** - The taxi driver that get assigned to a trip by accepting a request using the application.
- **GPS** - Global position system.
- **API** - Application programming interface. A set of routines, protocols, and tools for building software applications.
- **UC** - Use case
- **JSON** - JavaScript object notation, human readable data format.
- **HTTP** - Hypertext transfer protocol, a protocol to send data.
- **TTL** - Time to live, the amount of time units the owner of the variable exists.

2 Functional Description

2.1 Use Case Model

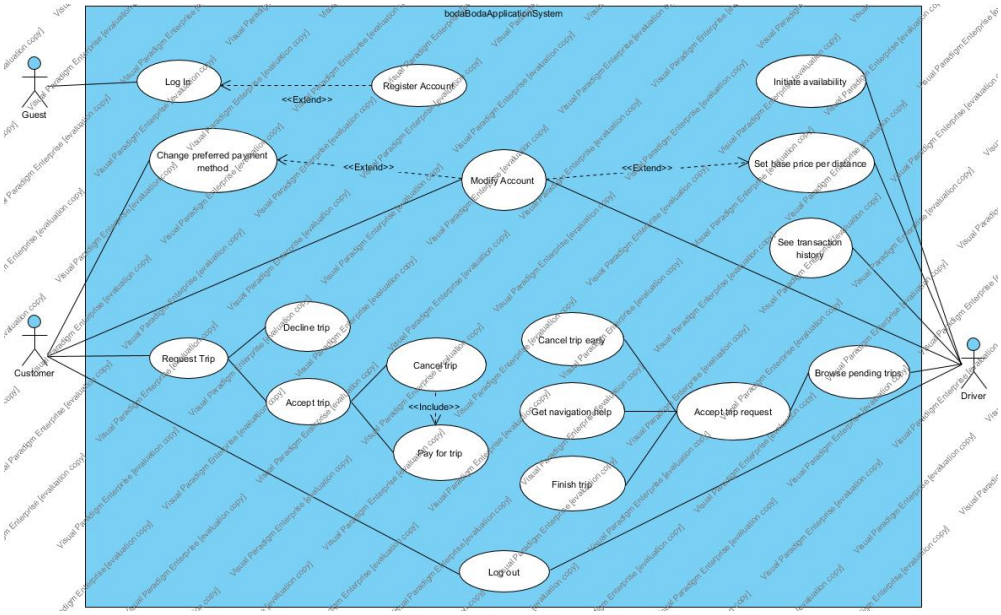

Figure 1: Use Case Diagram

2.1.1 Actors

- **Guest:** This actor represents a user on the application who do not yet have an account or is not logged in to her account. A guest can not request a trip, she just have access to the log in page and the registration page.
- **Customer:** This actor represents a user who is logged in to a customer account. A customer can request trips, modify her account or log out.
- **Driver:** This actor represents a user who provides the drive service. A driver can browse trip requests from customers, modify her account, check earning statistics and log out.

2.1.2 Use Cases

Important points of the Use case diagram : Have an account permit to become a customer after identification. You can identify yourself if you have an account. A customer can log out, request a trip and modify her account. When a trip request is sent, the application broadcast the trip to all nearby drivers, who can see all pending trip requests in a list. The driver can browse pending trip requests, modify her account

and see her earning statistics. The driver can also change her price per kilometer, as well as the starting fee for a trip.

This is the list of each different use cases. The page number refers to a detailed description of the use case :

- UC#01: Register account (page: 8)
- UC#02: Log in (page: 9)
- UC#03: Request Trip (page: 10)
- UC#04: Cancel Trip (page: 11)
- UC#05: Decline Trip (page: 12)
- UC#06: Accept Trip (page: 13)
- UC#07: Pay for Trip (page: 14)
- UC#08: Change preferred Payment method (page: 15)
- UC#09: Modify Account (page: 16)
- UC#10: Log out (page: 17)
- UC#11: Finish Trip (page: 18)
- UC#12: Get navigation help (page: 19)
- UC#13: Browse pending trips (page: 20)
- UC#14: Accept Trip Request (page: 21)
- UC#15: Set base price per distance (page: 22)
- UC#16: See transaction history (page: 23)
- UC#17: Cancel Trip early (page: 24)

2.2 Information about Guest's Use Cases

2.2.1 Register account

Initiator: Guest.

Goal: Create an account to have the possibility to log in.

Main flow Events:

1. The user adds her information to create the account.
2. Tap the register button.
3. The account is added to the database.

Extensions: This use case is described as an extension of the "log in" use case. Indeed, if you want log in, having an account is an obligation. When the account is created, your account is added to the customer list.

Exceptions:

1. If some value is incorrect, an error is returned. Go back to 1 and enter the correct value.

Comments: None.

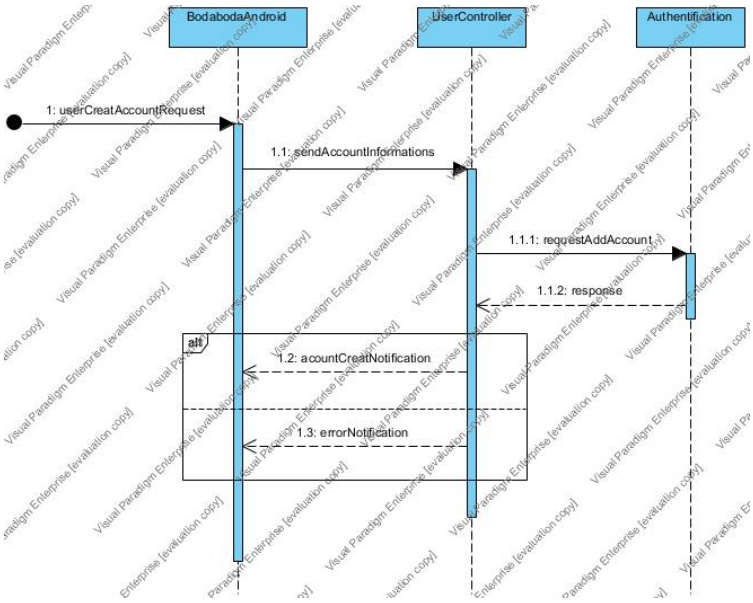

Figure 2: Register account sequence diagram

2.2.2 Log in

Initiator: Guest.

Goal: A user should be able to log in to the mobile application by entering a valid user-name and password.

Main flow Events:

1. Enter a valid user-name and password.
2. Tap the log in button.

Extensions: This use case extends Register account, which means the user should have been registered already in order to log in.

Exceptions:

1. Wrong account credentials. Go back to 2.

Comments: None.

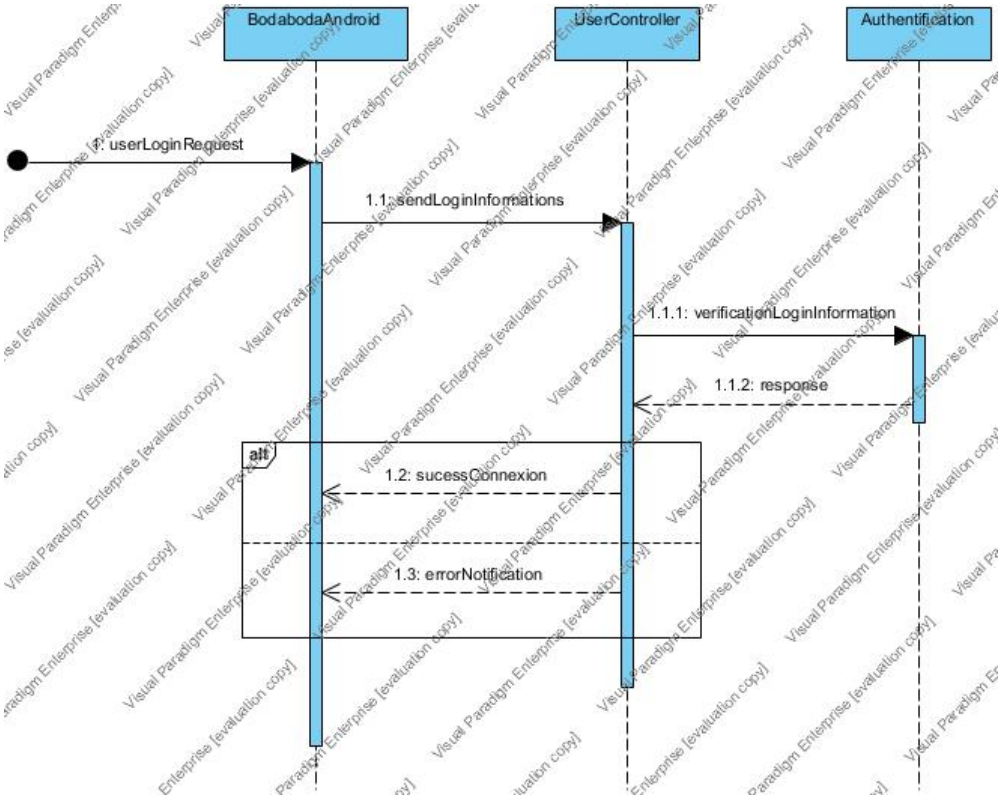

Figure 3: Log in sequence diagram

2.3 Information about Customer's Use Cases

2.3.1 Request Trip

Initiator: Customer.

Goal: A user should be able to request a trip by selecting starting and destination locations.

Main flow Events:

1. Enter starting and destination locations.
2. Tap on the request trip button.

Extensions: None.

Exceptions:

1. Not logged in as a customer. Go back to 1.
2. No drivers are available. Go back to 2.

Comments: None.

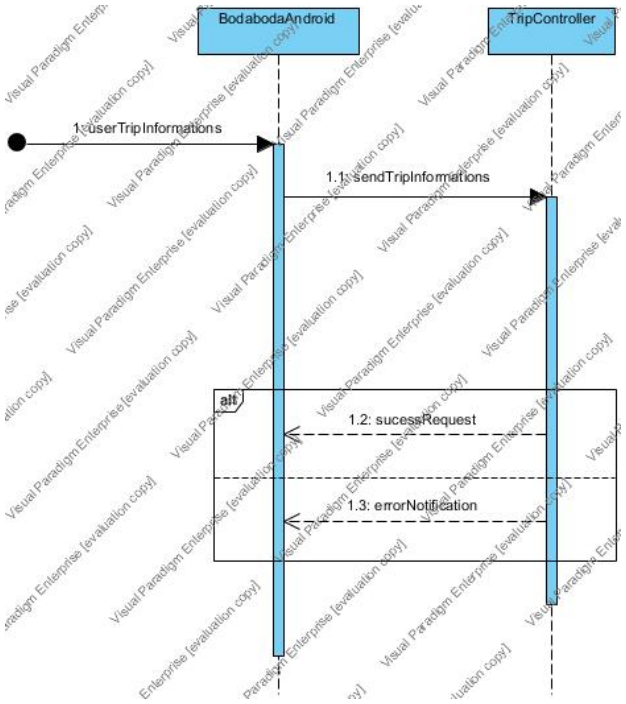

Figure 4: Request trip sequence diagram

2.3.2 Cancel Trip

Initiator: Customer.

Goal: A customer should be able to cancel a trip after the being matched with a driver.

Main flow Events :

1. Tap on the cancel trip when a proposition appear.

Extensions: None.

Exceptions:

1. Communication error.

Comments: If the trip has been accepted by a driver and the driver has not yet reached the customer and then gets canceled by the customer, this results in the customer being charged the driver's starting fee. If the trip has started and the customer wants to cancel it prematurely a price is calculated from the driven distance plus the starting fee.

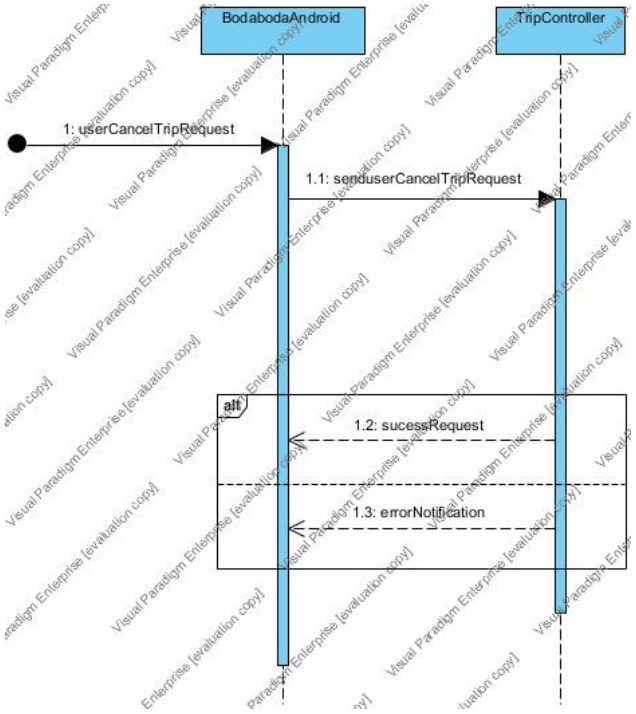

Figure 5: Cancel Trip sequence diagram

2.3.3 Decline Trip

Initiator: Customer.

Goal: A customer should be able to decline a driver match on their trip request.

Main flow Events:

1. Decline the suggested trip match.

Extensions: None.

Exceptions:

1. Wrong account credentials. Go back to 1.
2. Not logged in as a customer. Go back to 1.
3. No available drivers. Go back to 2.

Comments: Customers should have the choice to decline a trip without a fee if they think the price is too high or simply has changed their mind regarding taking the trip.

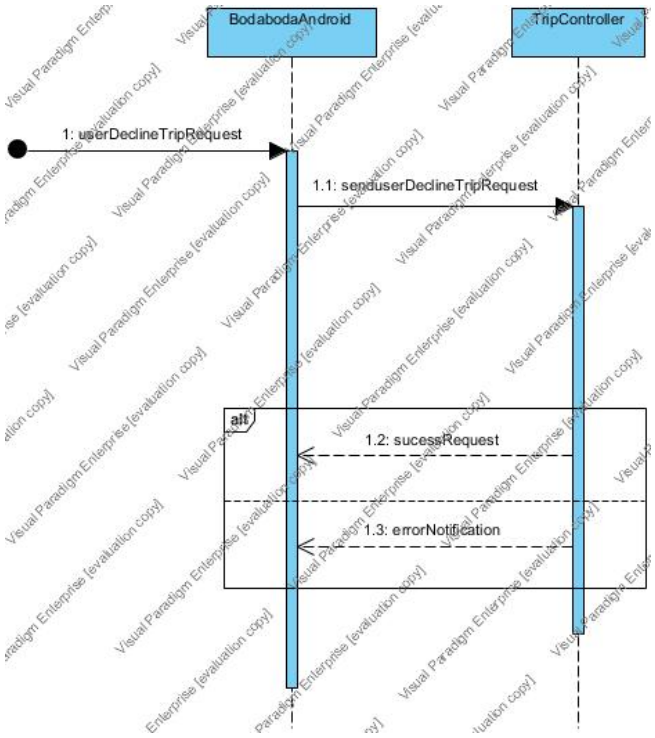

Figure 6: Decline Trip sequence diagram

2.3.4 Accept Trip

Initiator: Customer.

Goal: A customer should be able to accept a trip suggestion.

Main flow Events:

1. Tap on the request trip button.
2. Check the price and the driver's estimated time of arrival to the pickup spot.
3. Accept the trip.

Extensions: None.

Exceptions:

1. Wrong account credentials. Go back to 1.
2. No available drivers. Go back to 2.
3. The price is too high for the customer's liking. Go back to 2.

Comments: None.

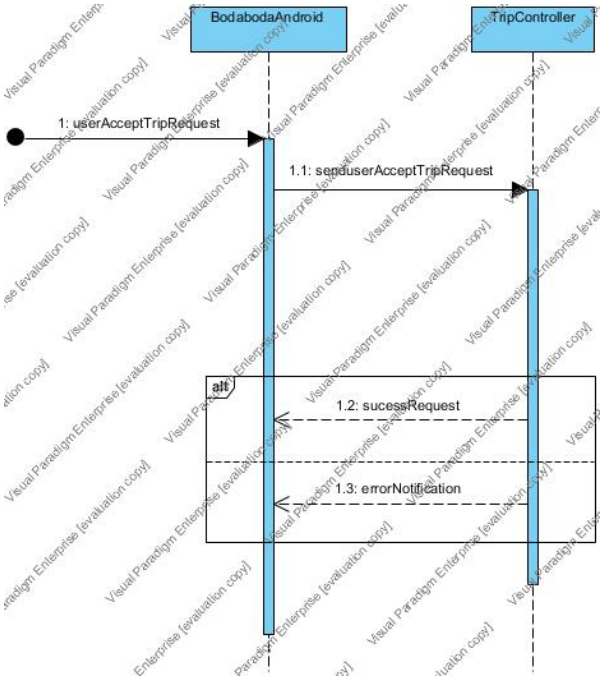

Figure 7: Accept Trip sequence diagram

2.3.5 Pay for Trip

Initiator: Customer.

Goal: A customer should be able to pay for a trip either by cash or card.

Main flow Events:

1. After Accept the trip, the driver initiates the trip.
2. Pay for the trip after finishing the trip.

Extensions: None.

Exceptions:

1. Wrong account credentials. Go back to 1.
2. Not logged in as a customer. Go back to 1.
3. No available drivers. Go back to 2.

Comments: If the customer pays with cash, the company share should be charged from the driver's credit card.

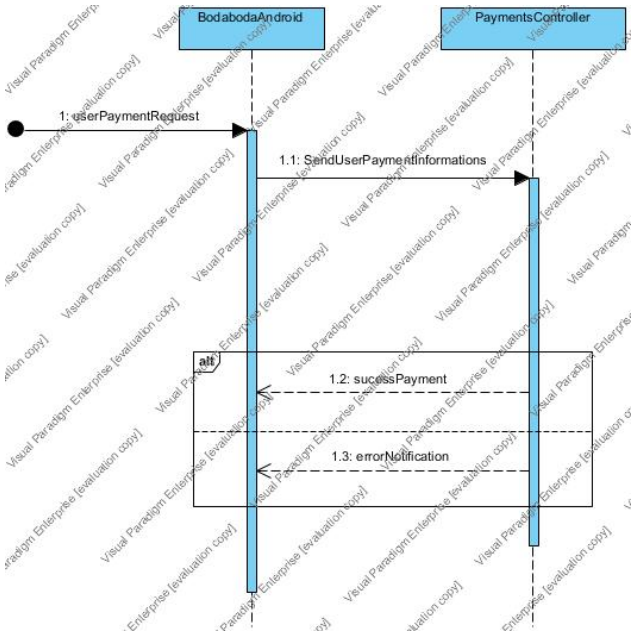

Figure 8: Pay for trip sequence diagram

2.3.6 Change preferred Payment method

Initiator: Customer.

Goal: A customer should be able to change the preferred payment method to either cash or card.

Main flow Events:

1. Tap on the modify account button.
2. Change preferred payment method.

Extensions: This is an extension to "Modify account"

Exceptions:

1. Wrong account credentials. Go back to 1.
2. Not logged in as a customer. Go back to 1.

Comments: This will give the driver a heads up on how the customer wants to pay.

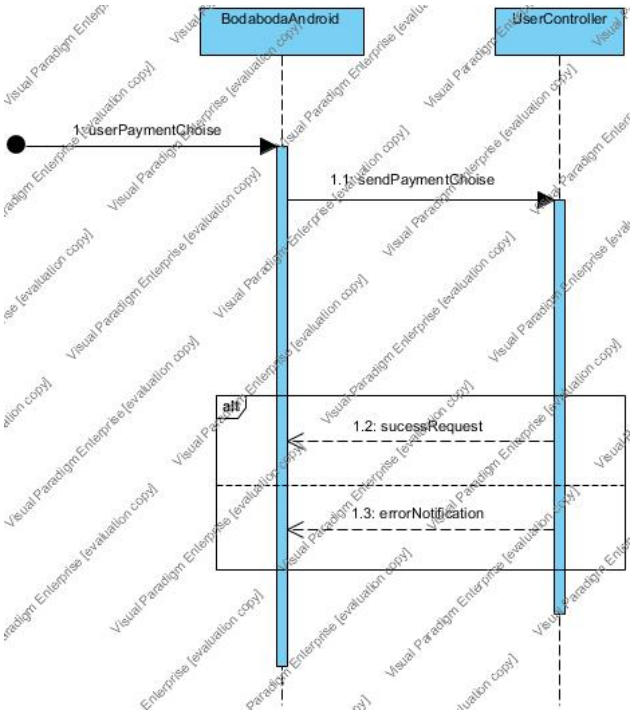

Figure 9: Change payment method sequence diagram

2.4 Information about Guest's and Driver's Use Cases

2.4.1 Modify Account

Initiator: Driver or Customer.

Goal: The user should be able to modify their account information except for the user-name. The user can choose to delete their account in here as well.

Main flow Events:

1. Log in to the application as a customer or driver with valid user name and password.
2. Tap on the modify account button.

Extensions: This use case is extended with optional "Change preferred payment method" for customers and "Set base price per distance" for drivers.

Exceptions:

1. Wrong account credentials. Go back to 1.

Comments: None.

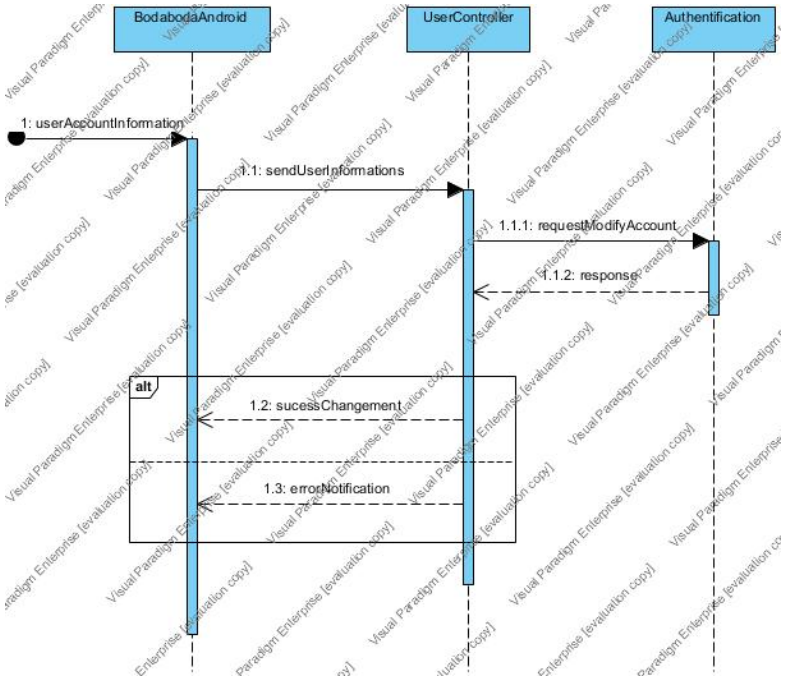

Figure 10: Modify account information sequence diagram

2.4.2 Log out

Initiator: Driver or Customer.

Goal: A logged in user should be able to log out of the application.

Main flow Events:

1. Log in to the application as a customer or driver with valid user name and password.
2. Tap on the log out button.

Extensions: None.

Exceptions:

1. The user is not logged in. Go back to 1.

Comments: None.

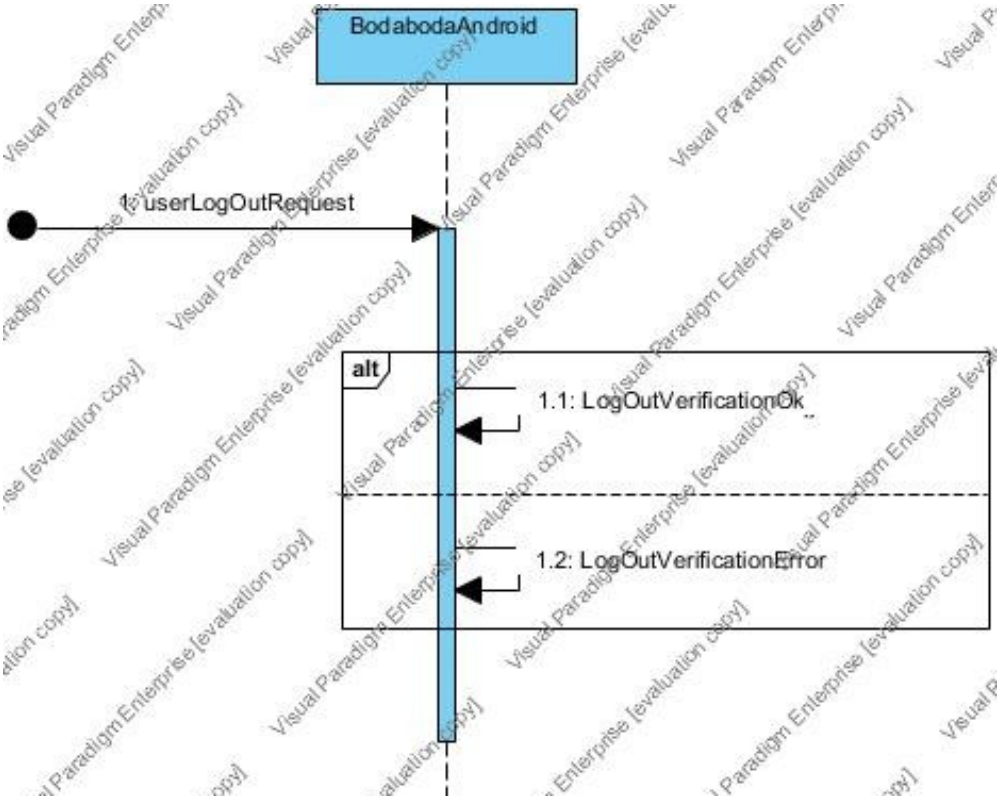

Figure 11: Log out sequence diagram

2.5 Information about Driver's Use Cases

2.5.1 Finish Trip

Initiator: Driver.

Goal: A driver should be able to finish a trip when the destination is reached.

Main flow Events:

1. End the trip when the destination is reached.

Extensions: None.

Exceptions:

1. Not logged in as a driver. Go back to 1.
2. No available trip requests. Go back to 2.
3. Could not find the customer. Go back to 3.
4. Trip is canceled early. Go back to 2.

Comments: None.

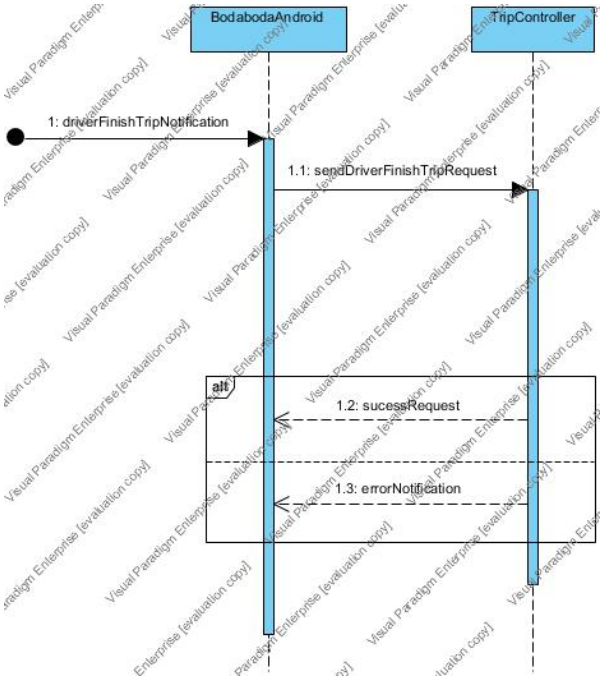

Figure 12: Finish trip sequence diagram

2.5.2 Get navigation help

Initiator: Driver.

Goal: A driver should be able to get help navigating to where they are going.

Main flow Events :

1. After browse and accept a trip.
2. Get navigation help.

Extensions: None.

Exceptions:

1. Wrong account credentials. Go back to 1.
2. Not logged in as a driver. Go back to 1.
3. No available trips. Go back to 2.

Comments: Navigation help should be provided to the driver if she needs help either finding the trip starting location, or the trip destination.

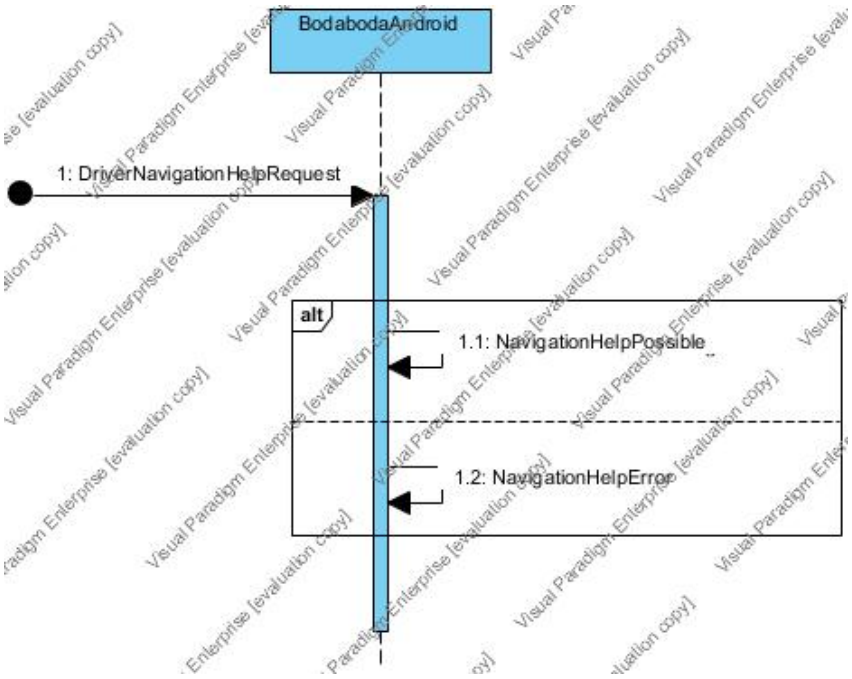

Figure 13: Navigation help sequence diagram

2.5.3 Browse pending Trips

Initiator: Driver.

Goal: A driver should be able to browse pending trip requests.

Main flow Events:

1. Log in to the application as a driver with valid user name and password.
2. Browse the pending Trips.

Extensions: None.

Exceptions:

1. Not logged in as a driver. Go back to 1.
2. Trip list not found

Comments: None.

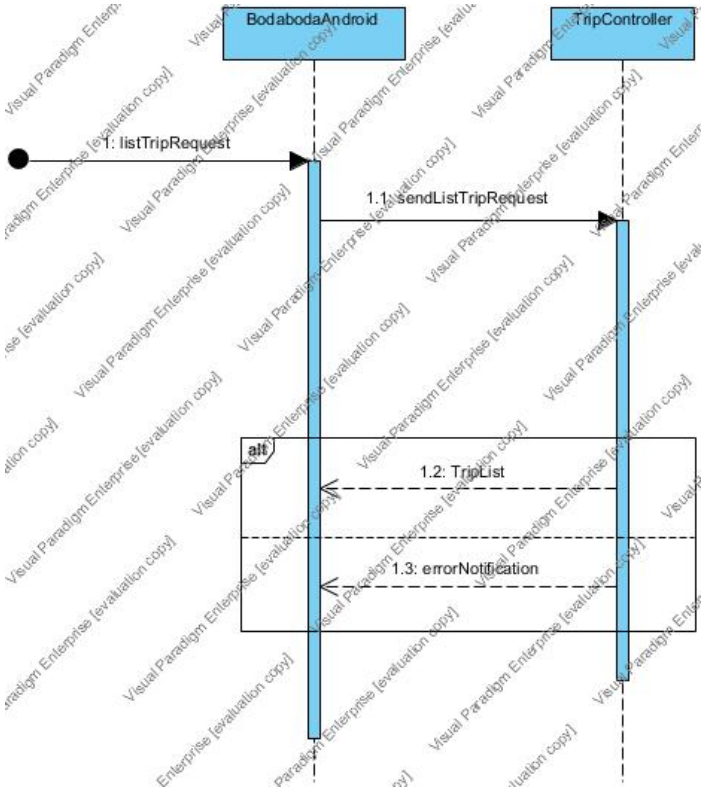

Figure 14: Browse pending trip sequence diagram

2.5.4 Accept Trip Request

Initiator: Driver.

Goal: A driver should be able to accept a trip request.

Main flow Events:

1. Log in to the application as a driver with valid user name and password.
2. Browse the pending trips.
3. Accept the trip of her choice.

Extensions: This use case extends "Browse pending trips".

Exceptions:

1. Not logged in as a driver. Go back to 1.
2. No available trips. Go back to 2.

Comments: None.

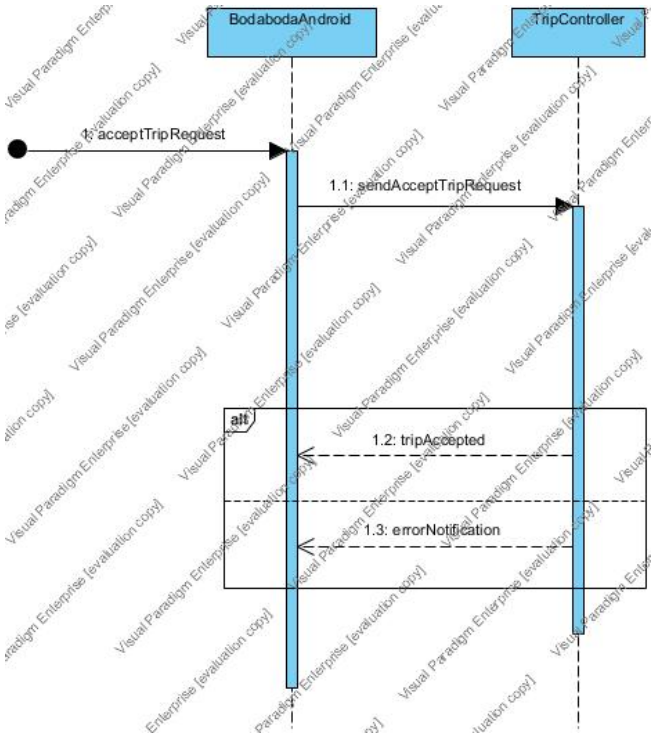

Figure 15: Accept trip request sequence diagram

2.5.5 Set base price per distance

Initiator: Driver.

Goal: A driver should be able to set the base price per kilometer.

Main flow Events:

1. Log in to the application as a driver with valid user name and password.
2. Tap on the modify account button.
3. Set the base price per kilometer.

Extensions: This use case extends "Modify account".

Exceptions:

1. Not logged in as a driver. Go back to 1.
2. Impossible modification.

Comments: There is a default price, but the drivers are free to change it.

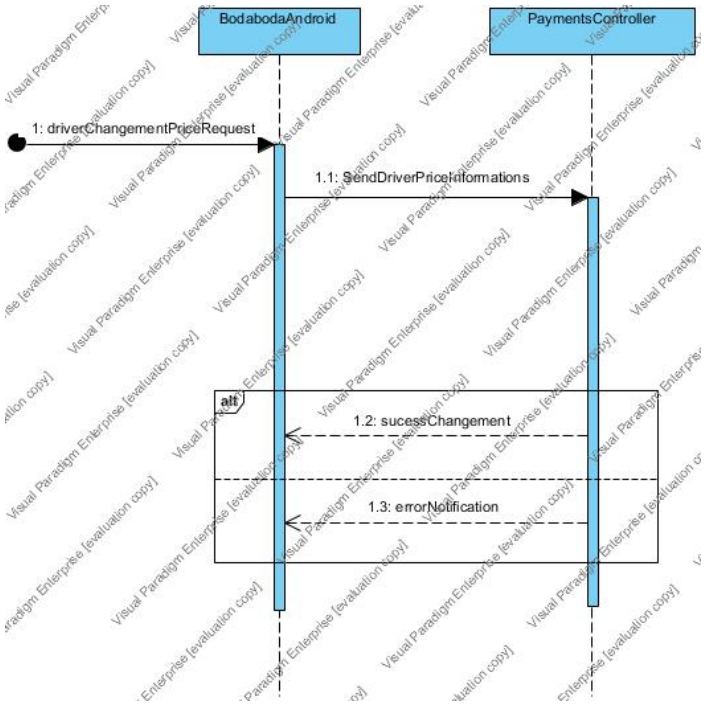

Figure 16: Set base price sequence diagram

2.5.6 See transaction history

Initiator: Driver.

Goal: A driver should be able to track their daily, monthly or yearly earnings.

Main flow Events:

1. Log in to the application as a driver with valid user name and password.
2. Tap on the show earnings button.

Extensions: None.

Exceptions:

1. Wrong account credentials. Go back to 1.
2. Not logged in as a driver. Go back to 1.
3. The driver has not yet completed any trips. No statistics available to show.

Comments: None.

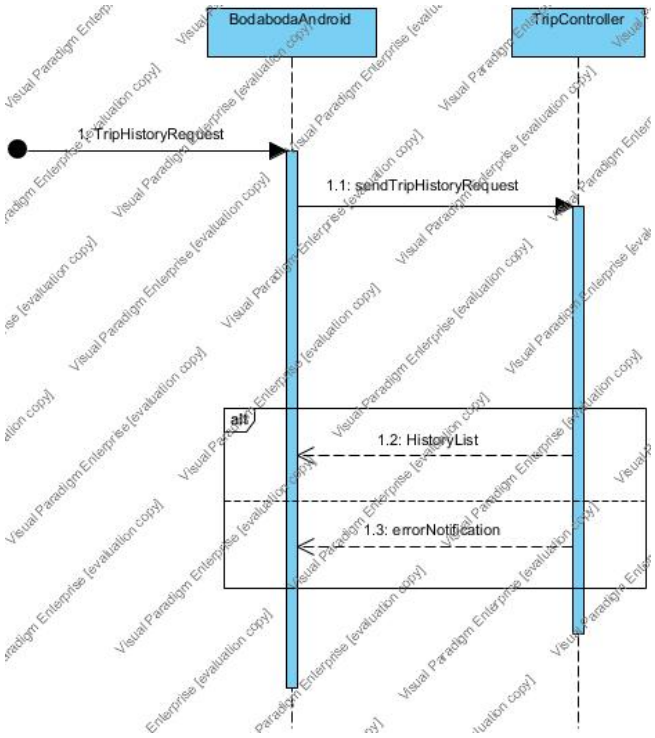

Figure 17: Transaction history sequence diagram

2.5.7 Cancel Trip early

Initiator: Driver.

Goal: A driver should be able to cancel the trip early.

Main flow Events:

1. Log in to the application as a driver with valid user name and password.
2. Browse the pending trip requests.
3. Accept the a trip of her liking.
4. Initiate the trip once the customer has been located.
5. Cancel the trip before reaching the destination.

Extensions: None.

Exceptions:

1. Wrong account credentials. Go back to 1.
2. No available trip requests. Go back to 2.
3. Customer could not be found. Go back to 4.

Comments: The driver should be able to cancel the trip in case the customer can not behave for instance. The price is calculated on the distance driven before cancellation.

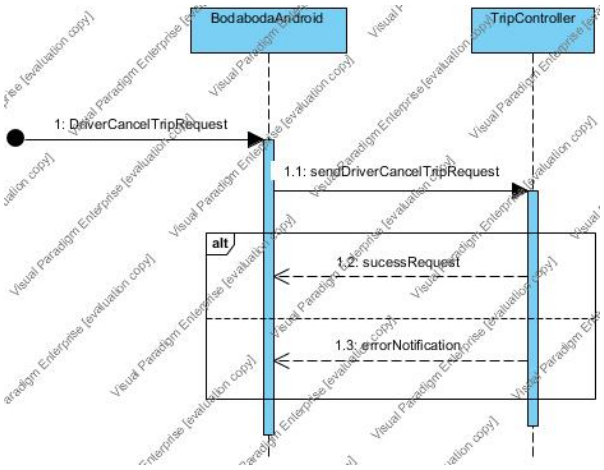

Figure 18: Cancel Trip early sequence diagram

3 Software Architecture

Software architecture describes how the system is built, it can be seen as a blueprint of the system. This section describes the architecture of our system.

3.1 Overview and Rationale

The system is built on a client/server architecture. The main parts of our system are the front-end android application and the back-end .NET Core server as well as the SQL database. The mobile application is simply the interface for the user that sends the data to the server which in turn perform the calculations on the data. The server will have access to the database in order to verify user log in credentials for instance, as well as storing data there. The front-end application will have no direct contact with the database, everything will go through the server.

Our application uses a couple of tools:

- **Google Places API:** This service allows us to work with addresses and locations that are stored in the Google database. With this information available we are able to pinpoint the trip locations, calculate the price for the trip, give estimated times etc. We can also help the driver navigate by using this API. This is the same information that Google uses for their Maps services, so it is easy to understand how it is useful for our application.
- **Google Maps API:** This service allows us to visualize the navigation the driver might need to find the customer. The customer can also use this to see where the driver is.
- **PubNub:** PubNub is a realtime asynchronous publish/subscribe message API. The way it works is that a publisher sends a message to a channel, this channel has subscribers that listen for new messages that they can read. We make use of it by sending the customer trip requests in the form of coordinates to a channel that the server is subscribed to. The server then updates a list with the trip requests making it visible for the drivers to browse.
- **JSON Web Token:** We use this as a way to confirm that the user is who they claim to be. The user is assigned a token when logging in, this token is then checked to verify that the user has permission to perform an action.
- **Retrofit:** We use this with the client application to make communication easier between the client and the web service. It is a type-safe REST client for Android and Java. It uses the OkHttp library for HTTP requests. It can receive or upload data structures such as JSON via the REST based web service.
- **Swagger:** We use Swagger for automating the documentation of REST API on the beck-end side. This tool provides a user interface for describing the HTTP Methods as well as testing them. Initial documentation generated by swagger can be seen in APPENDIX A (page: 41).

3.2 Collaboration between parts

The collaboration between the parts of the system can simply be described as follows: The application either takes input from the user or give output to the user. Input is sent to the server to work on the data. The packets, which follows the HTTP protocol structure, get sent with the data in JSON format. The packet received by the server tells it what to do by the HTTP method, if the packet contains data that needs to be stored in the database the server sends it to the database. If the data in the packet might need some more data from the database in order to be computed correctly, the server send a query to the database for the needed data. It might also be a simple packet that the server can handle on it's own, simply computing it and returning it to the application.

3.3 System decomposition

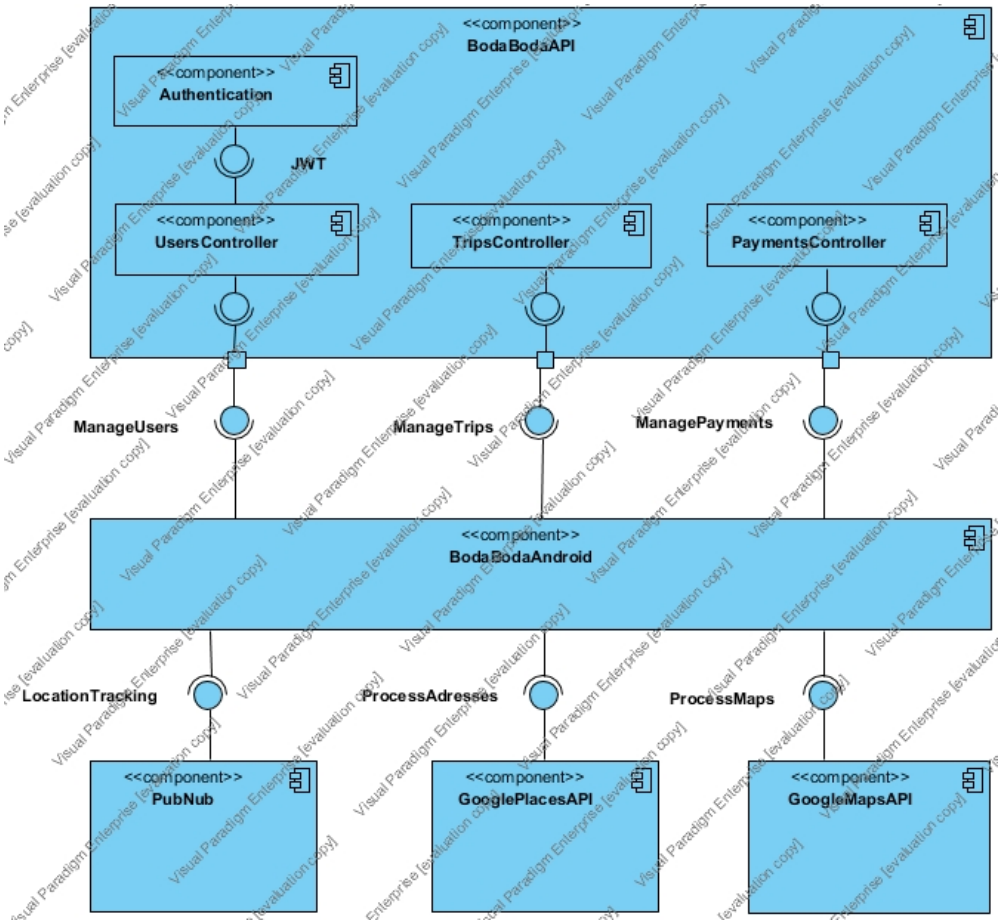

Figure 19: Component Diagram

The BodaBoda Taxi application as a system consists of several components;

- **BodaBodaAPI** - This component represents a main back-end server in charge of managing information about users, trips and payments, as well as a authentication server. Internally, BodaBodaAPI is made out of following sub-components, or parts:
  - Authentication - A component in charge of user authentication. The component provides an interface for acquiring and validating JWTs. JWTs carry important user information in the form of Claims such as their roles inside the system.

  - UsersController - This component manages all the information tied to the users in the systems. Also, it uses the provided interface from Authentication component, these two components work together. UsersController exposes an interface for user managing.
  - TripsController - A component in charge of handling all the trips and information about them. Exposes an interface for trip managing.
  - PaymetsController - A component in charge of handling all the payments, their information and payment logs. Exposes an interface for payment managing.

- **PubNub** - An external tool and a system component in charge of handling asynchronous messages. In BodaBoda Taxi application this component is used for real time geolocational tracking. It exposes an interface for location tracking.
- **GooglePlacesAPI** - An external API provided by Google for providing information about various real world places. It exposes an interface for address processing.
- **GoogleMapsAPI** - An external API provided by Google for maps manipulation. Exposes an interface for processing maps.
- **BodaBodaAndroid** - This component represent a front-end android application and a user interface for the system. It uses all the previously mentioned interfaces to consume and manage data, and it that way provides the functionality to the users.

3.4 Data persistence

The data is stored in an SQL database that is only accessed through the server. Every user account and information about trips etc. is stored in the database.

The database is divided in 6 tables :

1. **LocationI :** With all location's information (longitude, latitude, locationType ...) This, table can be used for save position of a customer, a driver, destination request and for each information related to the travel.
2. **User :** With all User's information. This table permit to save the profile of each users and each account (first name, last nape, password, email ...)
3. **TaxiPrice :** With all information related to a specific driver.
4. **Trip :** This table permit to save each information about a trip : a starting location, an ending location, the user who want a taxi, the driver, the price ...
5. **Payment :** With all information related to the payment.The payment is attached to a trip, a payer and a payee.
6. **Payment option :** This last table is here to save the method payment of an user.

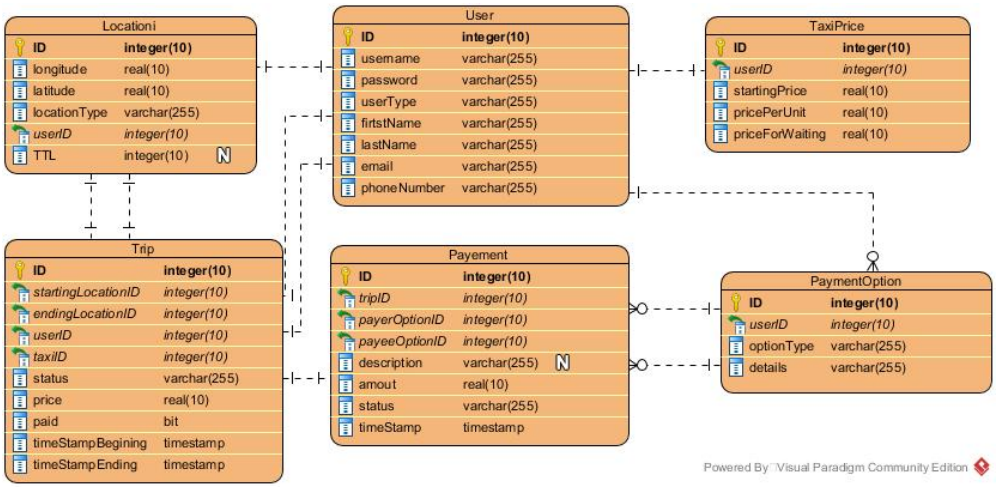

Figure 20: Database view diagram

4 Detailed Software Design

This section describes the front-end and back-end design.

4.1 Front-end Design

The front-end is built as an Android application acting as an interface for the user. The application is designed to be lightweight in order for it to give the phones longer battery time, thus most of the calculations will happen back-end. The front-end basically handles navigation through the application and input and output from the user that gets sent to the server for it to handle.

We use the phone's own GPS to keep track of where the user is, as well as the Google Places API to get the coordinates of the starting location and destination on the trip. This geolocation information is then sent as a message to a PubNub channel, from which the other parts of the system that needs the geolocation information can fetch it.

4.1.1 Structure

The Android Studio project structure is very deep by default. The relevant structure starts at the main directory, which is the root. The file structure from the main directory looks as follows:

- java
  - bodaboda - Project files
    - activities - Directory containing every activity used in the application.
    - classes - The classes used in the application.
    - utils - Different utility files, such as custom widgets and constant variables.
- res - Containing resource files, only the most important are listed.
  - layout - The files containing information about the layout of each activity.
  - drawable - The files used to customize the application layout, for instance images used for the background.
  - values - The files containing variables for strings, colors and styles.
- AndroidManifest.xml - The file containing essential information about the application.

4.1.2 Class Structure

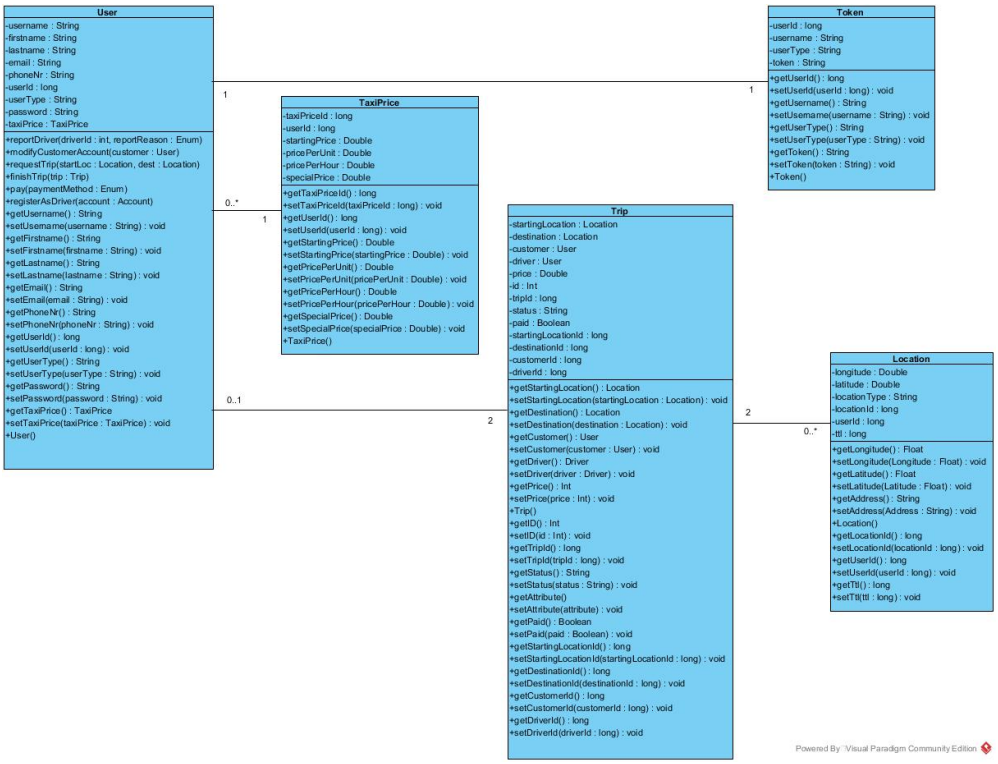

Figure 21: Front-End Class Diagram

- **User:** The user class is a class for both the driver and the customer. The user class contains every attribute that the customer and driver needs.
- **TaxiPrice:** The TaxiPrice class is for the driver's price settings. A driver will be able to hold multiple prices as a driver can have multiple vehicles.
- **Token:** The token class will hold the information received from the server for the user and will be the validation class for the connectivity with the server. The token string will be used to authenticate the user.
- **Trip:** The trip class holds all the information about a trip. This class keeps track of the actors in the trip, where the trip is starting and ending, as well as the price, etc.
- **Location:** The location class contains the coordinates of a location as well as a userId and a TTL. The location class is used as attributes for the trip class.

Above attributes will be changed through get and set functions.

4.2 Back-end Design

The back-end server is built as a .NET Core REST API providing storing, processing and authentication functionality to the system. In more detail the back-end server implements JSON Web Tokens as an authentication standard. Further, it uses Entity Framework as an object relation mapping tool which translates the defined models into database as tables and their relations. The back-end server provides HTTP methods for manipulating and processing user, trip and payment data.

4.2.1 Structure

The .NET Core Project, as the Android Studio project, has a very deep structure. The structure of the project is defined as follows:

- Models - Containing model definitions as classes, view models, and database context. (In the future this item should be moved elsewhere).
- Services - Containing classes for services which are charge of all the business logic and processing. The services are implementing their own interfaces which are further used by controllers, or other parts of back-end server in which they are needed.
- Controllers - Containing classes with definitions of HTTP methods. A controller is in charge of handling HTTP requests and they responses.
- Helpers - Contains classes with all the extra tools and scripts that are helping the functionality of the back-end server.
- Properties - Contains files for defining extra configurations.
- Migrations - Contains classes related to Entity Framework. These classes are auto generated by the Entity Framework and describe in which way Models should be mapped in database.
- BodaBodaServer.csproj - Contains project settings and references to files used by the project.
- Program.cs - Class containing the main() method.
- Startup.cs - Class with the important configurations for starting up the server.
- appsettings.Development.json - Main configuration file for development environment.
- appsettings.json - Main configuration file.

4.2.2 Class Structure

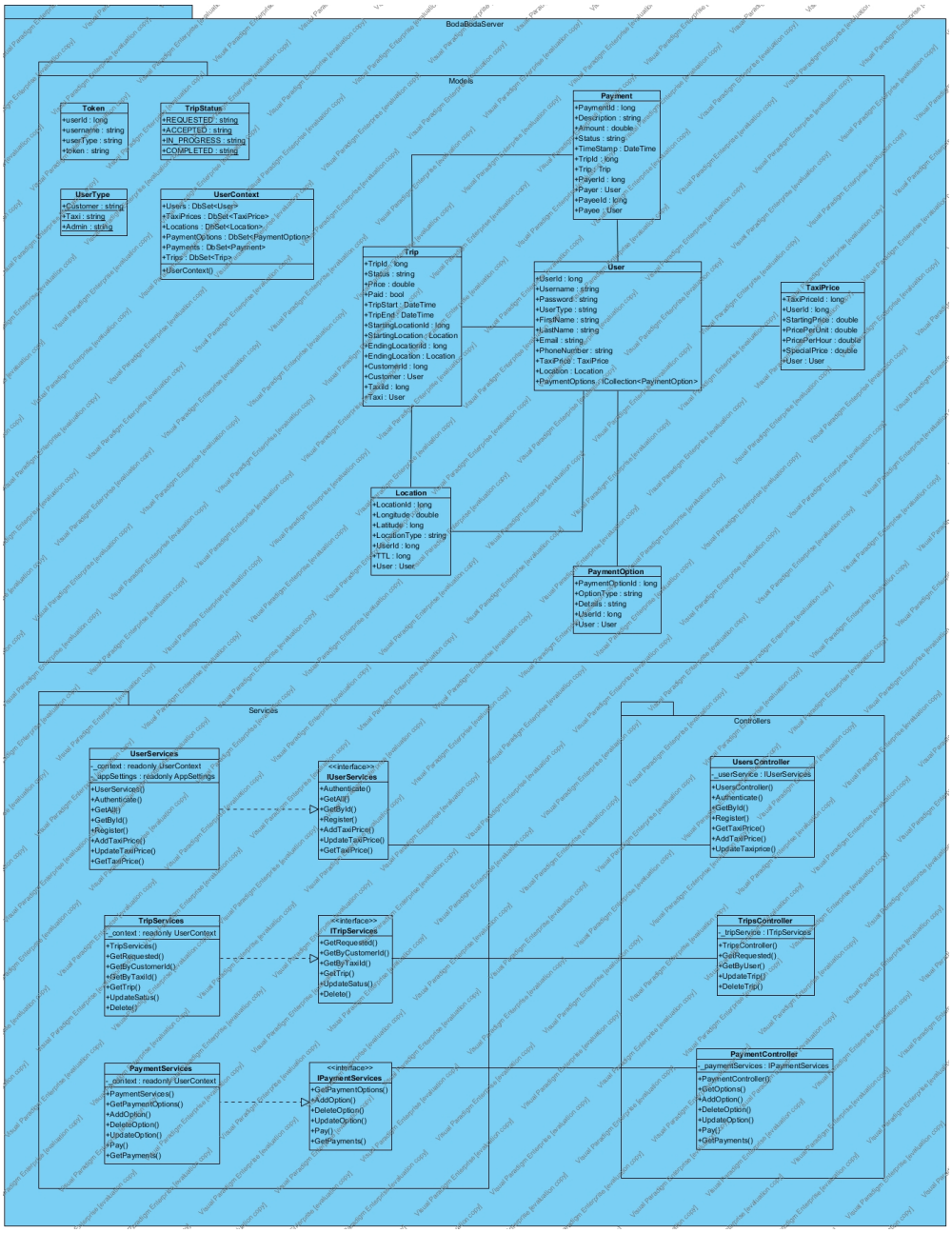

Figure 22: Back-End Class Diagram

- **Models**
  - User - A class containing basic user information for both taxi and customer. This class contains credentials as well.
  - TaxiPrice - A class containing information about taxi prices. It is linked to a taxi driver.
  - PaymentOption - A class containing information about users/taxis payment option information.
  - Location - A class containing location coordinates
  - Trip - A class containing information about the trip including starting and ending location, status, price and more.
  - Payment - A class containing information about payments. This information includes payment options, amount and trip that it's linked to.
  - Token - A view model class for formatting representation of a Token in HTTP requests.
  - UserType - A static class defining user types in the system.
  - TripStatus - A static class defining trip statuses.
  - UserContext - A class defining database contexts that are used as a interface to database. This class contains contexts for all of the models.
- **Services**
  - IUserServices - An interface defining the usage of UserServices.
  - UserServices - A class containing all of the business logic and processing of user data.
  - ITripServices - An interface defining the usage of TripServices.
  - TripServices - A class containing all of the business logic and processing for trips.
  - IPaymentServices - An interface defining the usage of PaymentServices.
  - PaymentServices - A class containing all of the business logic and processing for payments and payment options.
- **Controllers**
  - UsersController - A class defining all of the HTTP Methods and their responses related to user information.
  - TripsController - A class defining all of the HTTP Methods and their responses related to trips.
  - PaymentsController - A class defining all of the HTTP Methods and their responses related to payments and payment options.

5 Graphical Interfaces

The application's interface is structured in a way that favors usability over aesthetics, the customers and drivers are generally not familiar with mobile applications and it will therefore be beneficial for them to have an application that is user friendly. Aesthetics will be present of course, but will be designed with simplicity and usability in mind.

5.1 User Interface

The application is divided into two interfaces depending on what role the user has. The following section describes the different interfaces.

5.1.1 Guest Interface

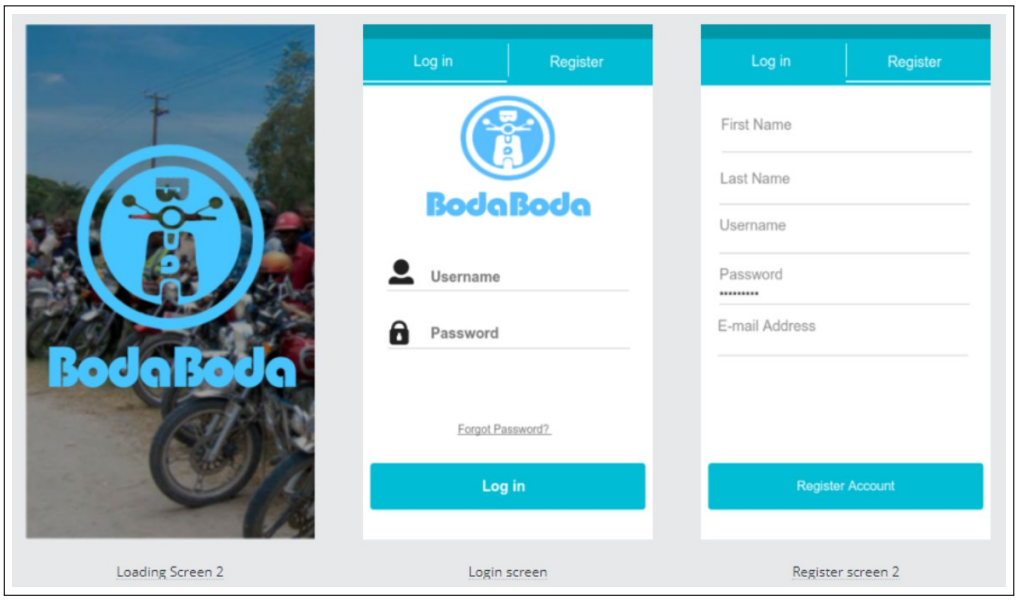

Figure 23: Guest interfaces

These different pages (figure 23) are here to log in and create an account. The first page is the loading page of the application. On the log in page, if the guest have an account she can try to log in. To do this she needs to enter her password and her user name and click on the "log in" button to make the verification.

If the guest clicks on the top right corner ("register" button) she can access the register page to create an account. To create an account, the guest needs to enter her email, first name, last name, phone number, user name and her password. Click on the register button to create the account if all the provided information is correct.

5.1.2 Driver Interface

The driver's main page (figure 24, left) display a list view of all the pending customer requests that are available to the driver, where the most vital information is shown. An item in the list can be expanded in order to show more information regarding that item.

When the driver clicks on the menu button (in the top right corner), a list of choices are displayed (figure 25, left):

- Show Earnings - A page containing transaction history of previous trips and statistics of earnings.
- Modify Account - A page where the driver will be able to change account information. A driver should be able to choose to become a customer instead of a driver, and will also be able to manually modify a booking fee and personal information.
- Log Out - If a driver wants to log out for the day, she can do so with a simple button click.
- Payment - A page to change the payment information.
- Help - A help page to have more information about the application.

Once the driver has logged into the application she will be met with the driver main page (figure 24, left). If the driver chooses a pending trip from a customer, the trip request will be expanded showing more information: The distance to the customer, the estimated price, The first name of the customer, the trip length, the starting location of the trip and the end destination. If a driver accepts the trip a new page will be triggered (figure 25, middle) where the driver will wait for the customer to accept the trip. The customer will be notified that a driver is willing to take the trip and will also receive the estimated price for the trip. If the customer chooses to accept the driver and the estimated price the driver will be directed to a new page which can navigate the driver to the customer (figure 25, right). The driver will initiate the trip when meeting the customer and will end the trip when arriving at the given destination. When the customer has paid the transaction will be visible in the Show Earnings transaction history.

If a driver wants to see their earnings or transaction history they can press the settings button in the driver main menu and navigate to transaction history. This will display the driver's total amount of driven trips (figure 24, right), earnings and hours worked on daily, weekly, monthly or yearly basis. If the driver taps one of the options she can get detailed information about the various options.

If a driver wants to edit their information they can press the Account Settings button in the driver settings menu (figure 25, left). The driver will be able to edit vehicle information, payment options, personal information and can also receive help from support.

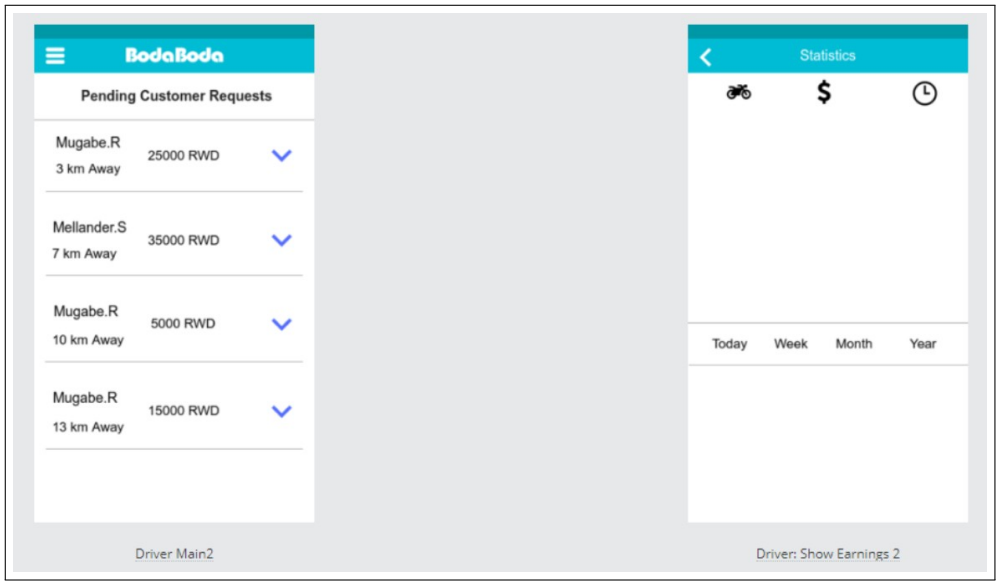

Figure 24: Driver interfaces part one

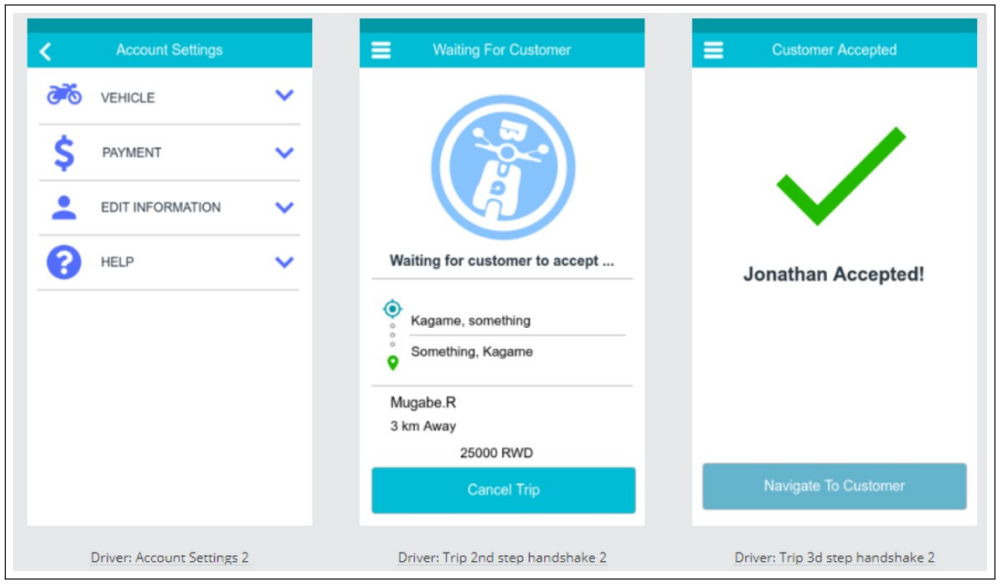

Figure 25: Driver interfaces part two

5.1.3 Customer Interface

The customer's main page (figure 26, left) consist to display a map with the direct possibility to write her desired trip information and request it. This is quick and simple to use. When the user clicks on the menu icon (on the top left corner) a list of possibilities is displayed (figure 26, right):

- Edit Information - A page that will allow the customer to edit their account information and preferred payment option.
- Log out - If a customer wishes to sign out from the application, she can do so with a simple button click.
- Payment - A page to change the payment method
- Help - A help page to have more information about the application, some tutorials ...

Once the customer has logged into the application she will be met with the customer main page (figure 26, left). On this page she can directly request a trip, and she is able to fill out the starting location of the trip and the destination.

Once a request is sent, the trip will be pending on the driver side until a driver chooses to accept the trip (figure 27, left). When a driver accepts the pending trip, the customer will be able to see the name of the driver and the estimated price for the trip (figure 27, middle).

The customer now has two options, either accept the trip or cancel the trip. If the customer chooses to accept the trip (figure 27, right), the driver will be notified and will be able to navigate to the customer's destination. Once the driver is at the customer's location, the driver will start the trip. If a customer wants to cancel an ongoing trip, she can do so by pressing the cancel trip button. The driver will then drop off the customer at a nearby place and the customer will be charged based on the distance traveled. Once the customer has reached her destination and the driver has ended the trip. The customer will be able to pay the driver.

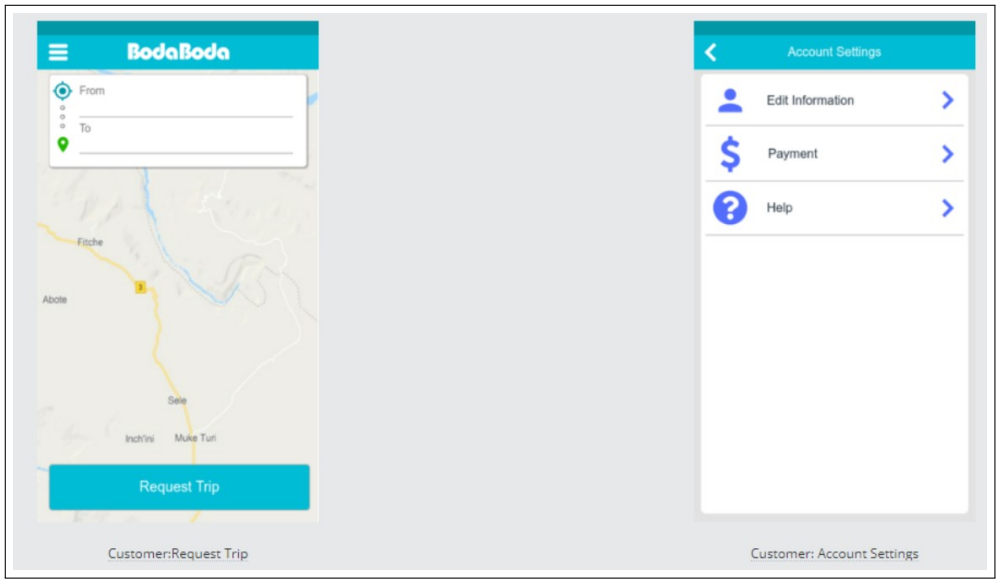

Figure 26: User interfaces part one

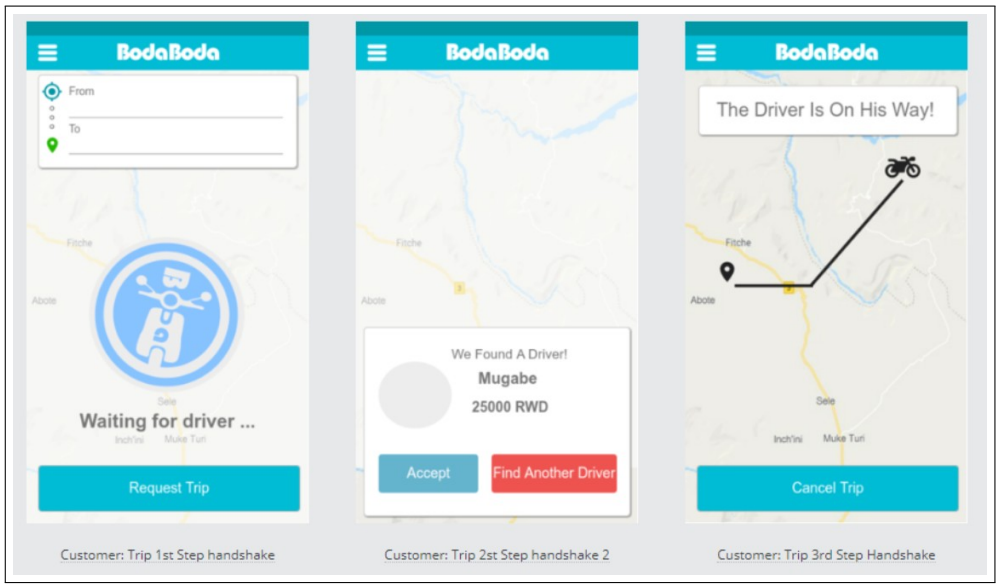

Figure 27: User interfaces part two

References

[1] Okapi. *Okapifinance.com*. 2018. URL: `https://www.okapifinance.com/` (visited on 11/26/2018).

6 APPENDIX A: Swagger Documentation

swagger Select a spec BodaBoda API V1

BodaBoda API v1

/swagger/v1/swagger.json

Payment

GET /api/Payment/options

Parameters Try it out

No parameters

Responses Response content type application/json

Code	Description
200	*Success*

PUT /api/Payment/options

Parameters Try it out

Name	Description

Name	Description
paymentOption ***(body)***	**Example Value** Model


```
"details": "string",
"userId": 0,
"user": {
  "userId": 0,
  "username": "string",
  "password": "string",
  "userType": "string",
  "firstName": "string",
  "lastName": "string",
  "email": "string",
  "phoneNumber": "string",
  "taxiPrice": {
    "taxiPriceId": 0,
    "userId": 0,
    "startingPrice": 0,
    "pricePerUnit": 0,
    "pricePerHour": 0,
    "specialPrice": 0
  },
  "location": {
    "locationId": 0,
    "longitude": 0,
    "latitude": 0,
    "locationType": "string",
    "userId": 0,
    "ttl": 0
  },
  "paymentOptions": [
    null
```


Parameter content type

application/json-patch+json

Responses Response content type application/json

Code	Description
200	*Success*

POST /api/Payment/options

Parameters Try it out

Name	Description

Name	Description

paymentOption

(body)

Example Value Model


```
"details": "string",
"userId": 0,
"user": {
  "userId": 0,
  "username": "string",
  "password": "string",
  "userType": "string",
  "firstName": "string",
  "lastName": "string",
  "email": "string",
  "phoneNumber": "string",
  "taxiPrice": {
    "taxiPriceId": 0,
    "userId": 0,
    "startingPrice": 0,
    "pricePerUnit": 0,
    "pricePerHour": 0,
    "specialPrice": 0
  },
  "location": {
    "locationId": 0,
    "longitude": 0,
    "latitude": 0,
    "locationType": "string",
    "userId": 0,
    "ttl": 0
  },
  "paymentOptions": [
    null
```


Parameter content type

application/json-patch+json

Responses Response content type application/json

Code	Description
200	*Success*

DELETE /api/Payment/options/{id}

Parameters Try it out

Name	Description

Name	Description
id * required integer($int64) *(path)*	

Responses Response content type application/json

Code	Description
200	*Success*

GET /api/Payment

Parameters Try it out

No parameters

Responses Response content type application/json

Code	Description
200	*Success*

POST /api/Payment

Parameters Try it out

Name	Description

Name	Description
payment ***(body)***	**Example Value** Model


```
"amount": 0,
"status": "string",
"timeStamp": "2018-12-06T10:46:56.478Z",
"tripId": 0,
"trip": {
  "tripId": 0,
  "status": "string",
  "price": 0,
  "paid": true,
  "tripStart": "2018-12-06T10:46:56.478Z",
  "tripEnd": "2018-12-06T10:46:56.478Z",
  "startingLocationId": 0,
  "startingLocation": {
    "locationId": 0,
    "longitude": 0,
    "latitude": 0,
    "locationType": "string",
    "userId": 0,
    "ttl": 0
  },
  "endingLocationId": 0,
  "endingLocation": {
    "locationId": 0,
    "longitude": 0,
    "latitude": 0,
    "locationType": "string",
    "userId": 0,
    "ttl": 0
  }
```


Parameter content type

application/json-patch+json

Responses **Response content type** application/json

Code	Description
200	*Success*

Trips

GET `/api/Trips/requested`

Parameters Try it out

No parameters

Responses Response content type application/json

Code	Description
200	*Success*

GET /api/Trips

Parameters Try it out

No parameters

Responses Response content type application/json

Code	Description
200	*Success*

PUT /api/Trips

Parameters Try it out

Name	Description

Name	Description

_trip

(body)

Example Value Model


```
      "ttt": 0,
    "user": {
      "userId": 0,
      "username": "string",
      "password": "string",
      "userType": "string",
      "firstName": "string",
      "lastName": "string",
      "email": "string",
      "phoneNumber": "string",
      "taxiPrice": {
        "taxiPriceId": 0,
        "userId": 0,
        "startingPrice": 0,
        "pricePerUnit": 0,
        "pricePerHour": 0,
        "specialPrice": 0
      },
      "paymentOptions": [
        {
          "paymentOptionId": 0,
          "optionType": "string",
          "details": "string",
          "userId": 0
        }
      ]
    }
  },
  "endingLocationId": 0,
```


Parameter content type

application/json-patch+json

Responses Response content type application/json

Code	Description
200	*Success*

DELETE /api/Trips/{id}

Parameters Try it out

Name	Description

Name	Description
id * required `integer($int64)` *(path)*	

Responses Response content type: application/json

Code	Description
200	*Success*

Users

POST `/api/Users/authenticate`

Parameters Try it out

Name	Description
userParam *(body)*	**Example Value** Model `{ "username": "string", "password": "string" }` **Parameter content type** application/json-patch+json

Responses Response content type: application/json

Code	Description
200	*Success*

GET /api/Users/{id}

Parameters Try it out

Name	Description
id * required integer($int64) *(path)*	

Responses Response content type text/plain

Code	Description
200	*Success* **Example Value** Model


```
  "lastName": "string",
  "email": "string",
  "phoneNumber": "string",
  "taxiPrice": {
    "taxiPriceId": 0,
    "userId": 0,
    "startingPrice": 0,
    "pricePerUnit": 0,
    "pricePerHour": 0,
    "specialPrice": 0
  },
  "location": {
    "locationId": 0,
    "longitude": 0,
    "latitude": 0,
    "locationType": "string",
    "userId": 0,
    "ttl": 0
  },
  "paymentOptions": [
    {
      "paymentOptionId": 0,
      "optionType": "string",
      "details": "string",
      "userId": 0
    }
  ]
}
```


POST /api/Users

Parameters Try it out

Name	Description

_user

(body)

Example Value Model


```
  "lastName": "string",
  "email": "string",
  "phoneNumber": "string",
  "taxiPrice": {
    "taxiPriceId": 0,
    "userId": 0,
    "startingPrice": 0,
    "pricePerUnit": 0,
    "pricePerHour": 0,
    "specialPrice": 0
  },
  "location": {
    "locationId": 0,
    "longitude": 0,
    "latitude": 0,
    "locationType": "string",
    "userId": 0,
    "ttl": 0
  },
  "paymentOptions": [
    {
      "paymentOptionId": 0,
      "optionType": "string",
      "details": "string",
      "userId": 0
    }
  ]
}
```


Parameter content type

application/json-patch+json

Responses

Response content type text/plain

Code	Description

Code	Description
200	*Success*

Example Value Model


```
{
  "userId": 0,
  "username": "string",
  "password": "string",
  "userType": "string",
  "firstName": "string",
  "lastName": "string",
  "email": "string",
  "phoneNumber": "string",
  "taxiPrice": {
    "taxiPriceId": 0,
    "userId": 0,
    "startingPrice": 0,
    "pricePerUnit": 0,
    "pricePerHour": 0,
    "specialPrice": 0
  },
  "location": {
    "locationId": 0,
    "longitude": 0,
    "latitude": 0,
    "locationType": "string",
    "userId": 0,
    "ttl": 0
  },
  "paymentOptions": [
    {
      "paymentOptionId": 0,
```


GET `/api/Users/prices`

Parameters Try it out

No parameters

Responses Response content type application/json

Code	Description
200	*Success*

PUT `/api/Users/prices`

Parameters Try it out

Name	Description
taxiPrice ***(body)***	**Example Value** Model


```
"startingPrice": 0,
"pricePerUnit": 0,
"pricePerHour": 0,
"specialPrice": 0,
"user": {
  "userId": 0,
  "username": "string",
  "password": "string",
  "userType": "string",
  "firstName": "string",
  "lastName": "string",
  "email": "string",
  "phoneNumber": "string",
  "location": {
    "locationId": 0,
    "longitude": 0,
    "latitude": 0,
    "locationType": "string",
    "userId": 0,
    "ttl": 0
  },
  "paymentOptions": [
    {
      "paymentOptionId": 0,
      "optionType": "string",
      "details": "string",
      "userId": 0
    }
  ]
```


Parameter content type application/json-patch+json

Responses Response content type application/json

Code	Description
200	*Success*

POST /api/Users/prices

Parameters Try it out

Name	Description

Name	Description

taxiPrice

(body)

Example Value Model


```
{
  "taxiPriceId": 0,
  "userId": 0,
  "startingPrice": 0,
  "pricePerUnit": 0,
  "pricePerHour": 0,
  "specialPrice": 0,
  "user": {
    "userId": 0,
    "username": "string",
    "password": "string",
    "userType": "string",
    "firstName": "string",
    "lastName": "string",
    "email": "string",
    "phoneNumber": "string",
    "location": {
      "locationId": 0,
      "longitude": 0,
      "latitude": 0,
      "locationType": "string",
      "userId": 0,
      "ttl": 0
    },
    "paymentOptions": [
      {
        "paymentOptionId": 0,
        "optionType": "string",
```


Parameter content type

application/json-patch+json

Responses **Response content type** application/json

Code	Description
200	*Success*

Models

PaymentOption

User

TaxiPrice

Location

Payment

Trip

Login