Truxi: A Software System for Enhancing The Communication Between Truck Drivers and Retailers

Bertalan Vad
Babeș–Bolyai University
Cluj–Napoca, Romania
vadbertalan@yahoo.com

Anna Kiss
Codespring
Cluj–Napoca, Romania
kiss.anna@codespring.ro

Roland Nagy
Codespring
Cluj–Napoca, Romania
nagy.roland@codespring.ro

Ákos Orbán
Codespring
Cluj–Napoca, Romania
orban.akos@codespring.ro

Károly Simon
Babeș–Bolyai University
Cluj–Napoca, Romania
ksimon@cs.ubbcluj.ro

***Abstract*—The purpose of the Truxi system is to connect freelancer transporters with users, who wish to deliver their goods.**

The software system consists of a mobile and a web application, which communicate with a central server. Suppliers are able to post their transportable goods through the web page. Drivers use the mobile app to make offers on cargoes and to get in touch with their clients. Furthermore, the system can suggest them an optimal route, that consists of the pickup and delivery locations of the transportable cargoes. By covering the suggested route, drivers can deliver the objects in the shortest way possible.

I. Introduction

It is a common problem, that someone would like to deliver valuables to a specific location, but can not realize it on their own. Nor the advertisement section of the newspaper, nor the internet will be able to help, since these kinds of services are rare and harder to find.

There are not so many freelancer truck drivers, because the number of jobs is relatively low and it is harder to earn a decent living out of it. One reason behind this is the already mentioned problem, the absence of a competitive platform. Most of the freelancer drivers are advertising their service on e-commerce sites, such as OLX [1]. The problem with these sites is, that they were not developed for this kind of matter and they do not provide a decent and clear communication channel between the drivers and their suppliers.

The software system behind the Truxi application brings a solution to the mentioned problems. It provides an online web platform for the suppliers, who own the transportable goods, where they can post so called cargoes, which will be available for the drivers to transport. Truxi also provides a mobile application for the drivers (available both on Android and iOS), where they can browse the available cargoes, and they can place offers on them. Since not only one driver can place a bid on a specific cargo, there will be a rivalry among them and the winner is selected by the cargo owner. After being selected, the driver receives detailed information about the cargo, including the contact information of the cargo owner. After accumulating more accepted cargo jobs through the mobile application, the drivers are able to generate the shortest possible route, which, if taken accordingly, takes the least amount of time to transport every cargo from the pickup location to the delivery location.

The project consists of three components: a web client, a mobile client and a central server responsible for data management and communication between the client applications.

[1] www.olx.ro

II. Functionalities and roles

There are two main user roles within the system: *supplier* and *driver*. The suppliers can operate on the web client, while drivers operate on the mobile application exclusively.

A. Supplier

In order to access the functionalities of the web application, one has to register into the system as a *supplier*. After a successful authentication, suppliers can create new cargoes, manage already listed cargoes, see the incoming offers, choose one of the offers, get in touch with the winner. When creating new cargoes, suppliers can upload an image, choose the pickup and delivery locations using an interactive map or specify other details related to the cargo.

B. Driver

Driver users have to use the mobile application to access the system. After authenticating themselves, they will be able to browse the available cargoes, check their details, place an offer on them, manage the offers, register vehicles into the system and manage these vehicles. After delivering the cargo in real life, they can mark it as *delivered*, which will enable the rating possibility for the cargo owner. The placed ratings are modifiable later. In case a driver has multiple accepted offers, they can choose to deliver all of them together. Since the transporter vehicle can be loaded with multiple cargoes at the same time, there are multiple combinations for the pickup and delivery locations. Using the mobile client, drivers have the possibility to generate a route, which contains these locations in an optimal order, so that the driver can transport the cargoes in the least amount of time.

C. Guest

In the system the scope of guests is minimal, as this role serves only as a way for users, that are new to the platform to look into the benefits of the application. Although guests are able to browse the available cargoes, nothing more. However, the possibility of creating a new user or login into their account always exists.

III. The Software System

The three main components (see Fig. 1) of the system are: a server developed with Spring framework [6], a *React.js* [7] web client application and a mobile application created with *React Native* [8]. The data of the system are persisted into a *MySql* relational database.

A. Server

The server application follows the *N-tier architecture*, hence the data management, the business logic and data exposure had been separated. The components are structured into separated layers, therefore the architecture is flexible, the code became robust and clear. The layered architecture reduces dependencies between different tiers, hence improves their testability and enables them to hide their implementation.

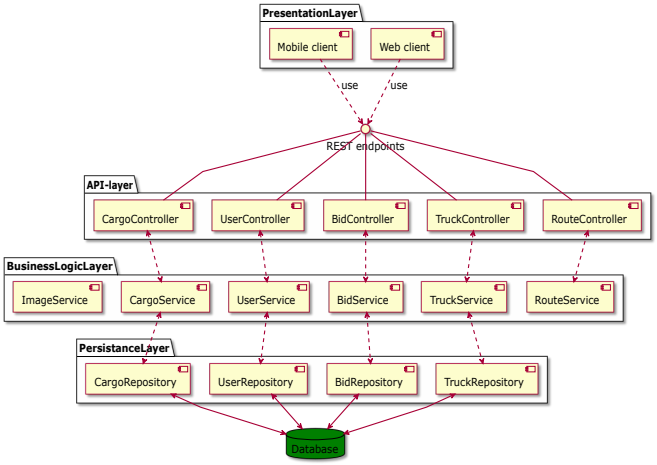

Fig. 1. The multilayer architecture used by the server.

There are three different layers, the API layer, which receives the incoming requests and builds the response, the service layer, where the business logic happens and the persistence layer, where the data management is implemented. The layers are *closed*: each layer communicates only with adjacent layers.

1) Data model and persistence layer: The domain model classes used in the system and their properties are visible on the 2nd image. The main entities are the `Cargo`, `User`, `Bid` and `Truck`. They all inherit from `BaseEntity` and `AbstractModel` abstract classes an ID and a *Universally unique identifier* (UUID). The ID is used as a primary key by the *Object Relational Mapping* (ORM) framework to identify the entity, while the UUID is used as a global identifier to the entity.

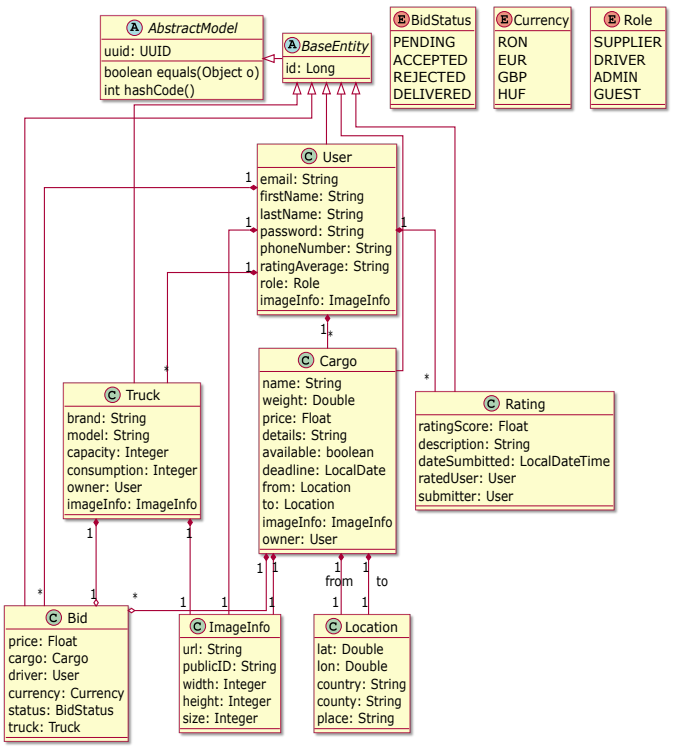

Fig. 2. The Domain Model used by the system.

Users are distinguished by their role. They may have the following roles: *Supplier*, *Driver* or *Guest*. Users with *Driver* role can create offers, which will be wrapped in `Bid` objects. Every bid contains the following data: the offered price, its currency, the cargo object, the driver that made the offer and the truck the driver intends to use to transport the cargo. A bid also has a `BidStatus` property of enumeration type, that can have one of the values: *pending*, *accepted*, *rejected* and *delivered*. The status of the bid plays an important role in the application and it is determined by user actions.

In case of the `Cargo`, `User` and `Truck` entities there is a possibility to upload images. Images are uploaded using *Multipart Form Data* encoding type, the server stores them in the Cloudinary [10] cloud service. When someone wants to access an image, an access link to the image will be emitted. This link and a few other attributes of the image are stored in the database.

Since the server application is written with the help of the Spring framework, *Spring Data* is responsible for the persistence. It automatically provides implementation to the *Create-Read-Update-Delete* operations on system entities, but also gives the possibility of implementing custom database operations.

2) Business logic layer: The business logic layer (or service layer) is the part of the code, where the more complex operations are done. Every entity in the system has its own service, which performs the entity related business logic. An entity service consists of a *Java interface*, which describes

the operations, and its implementation. Separating the interface from its implementation was necessary because of the *Inversion-of-control* (IoC) mechanism [2]. The separation also made the code more testable and allowed the hiding of the implementations. The functions of the service layer are complex, because they contain a lot of validation and calls towards the persistence layer. These functions are running as transactions in order to ensure the valid state of the database.

3) API layer: The communication between the server and the clients is realized through an *Application Programming Interface* (API), which is published by the server. This API follows the *REpresentational State Transfer* (REST) architectural style [1]. The API of the server inherits the attributes of the REST architecture: uniform interface, statelessness, client-server communication, layered system. In order to access the resources served by the REST API of the server, the clients have to use *Uniform Resource Identifier*s (URIs).

The API layer (or controller layer) is the outermost layer of the server. It is the layer that communicates with the client applications. During this communication the data transfer happens through the *Data Transfer Object* (DTO) pattern [3]. An entity could be translated into one or more DTO classes, that embody different representations of it. The same DTO classes are used both on server side and client side. When sent from the server to the client or vice-versa, the DTOs are being converted to JSON format. The API layer always calls the business logic layer, which works only with model objects, so before passing on the data, the API layer has to convert the DTOs to model objects using assembler classes.

The various requests are being handled by controller classes, following the *Front Controller Pattern* [16]. There is a separate controller class for each entity, thus there are separate base URLs too. In concordance with the REST conventions, the most important base URLs are: `/api/cargoes`, `/api/users`, `/api/bids`, `/api/trucks`.

B. Mobile and web client

The mobile and web clients are part of the presentation layer. They provide an easy to use and user friendly graphical interface, while they are communicating with the server in the background. The web client is using the *React.js* framework, the mobile client is using the *React Native* framework. React Native enables the use of React.js in a native mobile environment, thus the same declarative UI framework could be used both on the mobile and the web client. As a result the architecture of the client applications (see Fig. 3) is very similar.

1) Components: *React* and *React Native* are both component-based libraries, which means that UI elements are structured into *React component*s which are organized into a *component-hierarchy*. Due to an application level convention, the React components are class components.

Each component in the application has a single task (*single-responsibility principle* [4]), thus, they are always further divided into child components, as long as it makes sense.

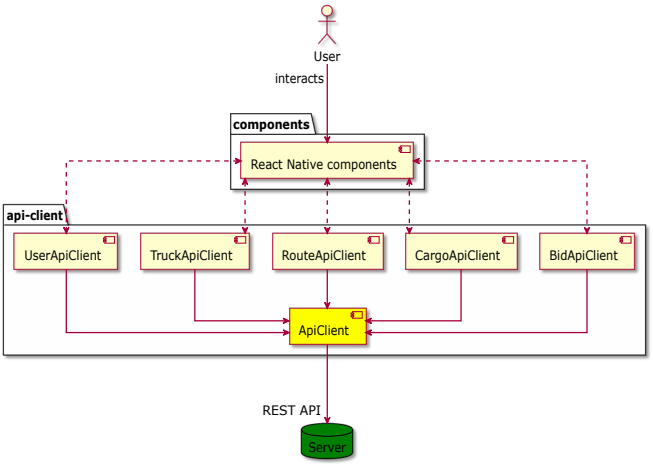

Fig. 3. The architecture of the client applications.

2) Data model: The client applications mirror the domain model of the server, and they also have the same DTO classes. The use of DTO classes imply the need for assembler classes that convert the DTOs to model classes used by the client app.

3) API client: In the client applications the only responsibility of components is handling the user interface. State mutations and updates are accomplished by the *MobX state management library* [9]. Tasks that require complex logic are performed in the MobX store classes. Networking and communication with the server are done by the API client layer. The API client layer is the direct consumer of the REST API of the server.

The API client layer can be divided into two parts: the `BaseApiClient` class and the ApiClient classes per entity. The `BaseApiClient` class makes the API calls towards the server. Using this base class every configuration operation can be made in one place. It performs the API calls through its functions, which comply to the used HTTP methods (GET, POST, PUT, PATCH, DELETE). The functions expect the relative URI path as parameter and in some cases (e.g. POST, PUT) the request body. Each entity in the client applications has its own ApiClient class, which contains methods for entity-specific operations. These methods prepare the API calls, perform the necessary DTO conversions and call the suiting `BaseApiClient` function. This behaviour is implemented using composition, the specific ApiClient classes have a reference to the `BaseApiClient` class. The ApiClient class methods are called by the components.

IV. Optimal route generation

The optimal route is comprised of locations, that are either pickup or delivery locations of the transportable cargoes. The algorithm expects a few parameters to be passed: the cargo the driver wants to begin the route with, the truck, which will be used throughout the route and a Boolean value denoting whether the driver wants to return to the starting point after the end of the route.

The generation of the optimal route is a variation of the Traveling Salesman Problem [17], which makes it an *NP-hard* problem. In order to solve the problem, an evolutionary algorithm is used: a custom version of the *2-opt* algorithm [5]. As distance function the *Euclidean distance* function is used.

In case of the Truxi application, there are some *constraints* the algorithm needs to satisfy.

- The geographical points of a cargo can be of two types: pickup point and delivery point. The delivery point can not be visited sooner than the pickup point, because one can not deliver a cargo without picking it up first.
- The vehicle has an upper limit of capacity, thus can not be overloaded at any moment of the route.

Taking these into consideration, the algorithm creates a list, which contains the pickup and delivery points of the cargoes in the order they should be visited. If the driver transports the cargoes according to the order imposed by the list, they will cover the shortest possible total distance during their tour.

Algorithm 1: The 2-opt swap function

```
Function TwoOptSwap(pointList, leftIndex,
 rightIndex):
    newPointList ← ∅
    leftOuts ← emptyStack
    for i ← 1 to leftIndex do
        newPointList.append(pointList_i)
    end
    for i ← rightIndex to leftIndex by − 1 do
        point ← pointList_i
        if point.type = DESTINATION &
         point.pair ∉ newPointList then
            leftOuts.push(point)
        else
            newPointList.append(point)
        end
    end
    while leftOuts stack not empty do
        point ← leftOuts.pop()
        newPointList.append(point)
    end
    for i ← rightIndex + 1 to pointList.size do
        newPointList.append(pointList_i)
    end
    return newPointList
```

The algorithm starts with a randomized sequence of the input points and keeps performing the *2-opt swap* on it. The 2-opt swap is implemented in a separate function (see algorithm 1). It has three parameters: the 2-opt-swappable list, a left index and a right index. The 2-opt swap consists of copying the list items from the beginning of the list to the left index, then reversing the subarray between the left and right indices, and lastly, copying the remaining items from the right index until the end. When reversing the subarray, in case of every delivery point it is checked if its pair, the pickup point of the same cargo, has already been placed in the list. If not, instead of being copied into the output list, they will be left-out and pushed into a stack. After the reversed subarray was added to the output list, the stack entries will be added next. Because of the stack data structure, the left-out elements can keep their original order in the output list. This ensures that the first constraint of the problem remains satisfied. The left index and the right index parameters take the values (i, j), where $i \in [0, n-1]$ and $j \in [i+1, n)$, and n is the number of points in the list. After each swap, the new total travel distance is determined, and in case it is lower than the previous total distance, the modified list takes its place. Before a new route takes over the place of the old route, it has to pass the capacity test, which checks whether an overload will occur during the route, so that the second constraint stays intact. The swapping stops, when the total distance does not improve anymore. After the stop of the cycle, the resulting list will contain the optimal route.

V. Technologies

A. Tools

The whole source code of the Truxi application is under version control and the used version control system is *Git*. The remote repositories are provided by *GitLab*. The development of the application happened by respecting the rules of the *Scrum framework*.

Using the *GitLab CI/CD* DevOps pipeline alongside *Docker* and *Docker-compose* virtualization tools [11], the server and the web client applications are continuously deployed to a server machine. Therefore, the containerized server and web applications are publicly available on the internet.

B. Server-side technologies

The server application is based on the *Spring framework* and it is written in the *Java* programming language. The configuration and start-up are supported by *Spring Boot* [6]. The controller classes of the API layer use the *Spring Web MVC* package. The persistence layer depends on the services of the *Spring Data JPA* subproject. As JPA implementation, the *Hibernate Object-Relational Mapping* (ORM) framework is used. The data of the server is persisted using a *MySql* relational database. The database schema changes are managed by the *Liquibase* library [13]. As build and dependency management tool, *Gradle* [12] is used.

The images uploaded to the server are not stored in the database, but in the *Cloudinary* cloud service. It provides a Java integration [10], which is being used on the business logic layer to forward the uploaded images from the server to the cloud.

C. Mobile and web clients

The web and mobile clients were written using *React.js* [7] and *React Native* [8], which are very similar libraries, thus the mobile and web clients share a lot of similarities. Both React and React Native are *JavaScript* libraries, developed by Facebook. The React components, that are responsible for the

UI, were written in *JavaScriptXML* (JSX) language, which is a JavaScript syntactical extension. Both the mobile and web clients were written using the *TypeScript* language, which is a JavaScript superset and makes the JavaScript code strongly typed, making it more errorproof. As state management library, the *MobX* framework [9] is used. The *dependency injection* in the client application is done with the *InfersifyJs* library [14]. Forms throughout the client applications are made easier to handle by the handy *Formik* library [15].

There are a few technologies, that differ between the client applications. For routing, on mobile *React Native Router Flux*, whilst on web the *React Router DOM* library is used. *Semantic UI React* and *React Native Elements* were used as component libraries.

VI. Using the Truxi Application

One can use the Truxi application only through the web or mobile clients. Many of the features the application offers are protected, so users have to register and authenticate themselves first. The registration process is the same in both of the client applications. The role of the user depends on which client application the registration was made from. Suppliers can register from the web client, drivers from the mobile client.

A. Web client

Suppliers are required to sign in into the application. After the login, suppliers will be able to create new cargoes. They have to specify the name, weight, and price of the cargo. They have to enter the pickup and delivery locations (using an interactive map). Optionally, suppliers can attach extra details to the cargo, a transport deadline and an image.

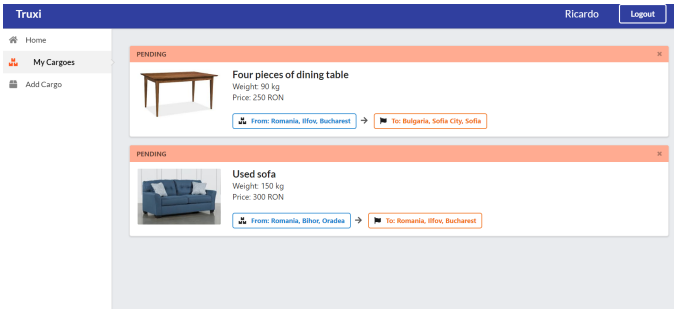

Fig. 4. The web page that lets suppliers manage their own cargoes.

Suppliers can see their own cargoes (see Fig. 4), they can modify or delete them. There are two types of cargoes in the cargo list of the user: available and unavailable cargoes. Available cargoes have no accepted offer yet. In case of available cargoes, the owner can see the incoming offers, whose price can not be higher than the original cargo price, and can accept one of them. From then on, that cargo will be considered as a not available cargo. The supplier can access the private information of the selected driver by clicking on a not available cargo. The private information was not available for the supplier, just from the moment the offer was accepted and they became business partners. With the help of this private information (a phone number) the driver can be contacted.

B. Mobile client

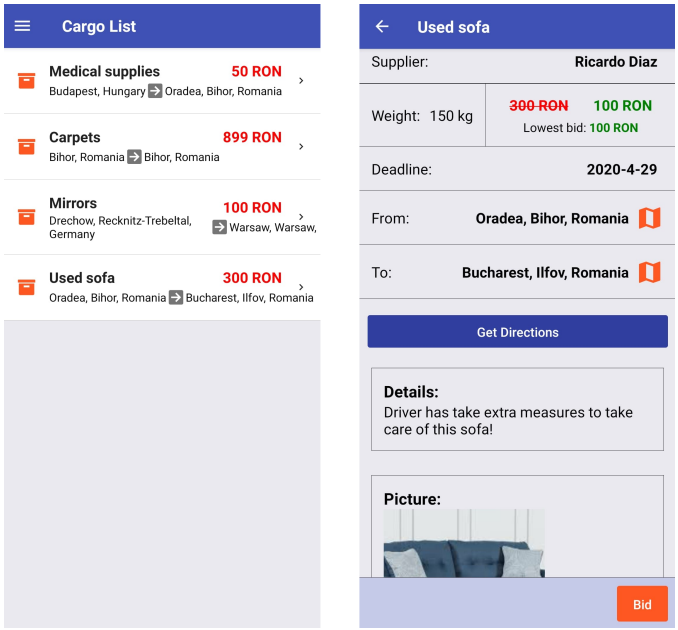

Fig. 5. The list of available cargoes and the detailed cargo view.

When the driver opens the mobile application, a list view of available cargoes becomes visible (see Fig. 5). The user can inspect the details of every cargo. The cargo details also unveil the lowest bid price to that particular cargo, even if it does not belong to the logged in user. Nevertheless, on that view, one can place bids on the cargo or can modify the already placed bid. The locations can be inspected in the local map application (Google Maps or Apple Maps in case of iOS). However, the locations are not precise, just of indicative nature. The precise GPS coordinates will be available for the driver only in case his offer is selected by the cargo owner.

The navigation in the application is made easier with the help of a drawer, which contains entries to majority of the routes in the application. The bid manager screen also has an entry in the drawer, where the driver can see every bid he made. The bids are filtered by their status. There are different actions available based on the status of the bid. In case of pending bids, the driver can change the bid amount or can remove the offer. In case of accepted bids, the driver can mark the cargo as delivered and also rate the proficiency of the supplier. In case of delivered bids, one can modify the rating given. In case of rejected offers, only the removal of the offer is possible.

Under the *"My Trucks"* entry, drivers can access the vehicle manager interface, where they can add new vehicles and remove existing ones.

The *"Route finder"* menu option takes the user to the optimal route generation interface (see Fig. 6). Before accessing the route, the driver has to specify the input parameters, by selecting the first cargo of the route, the vehicle to be used and whether he intends to return to the starting point of the

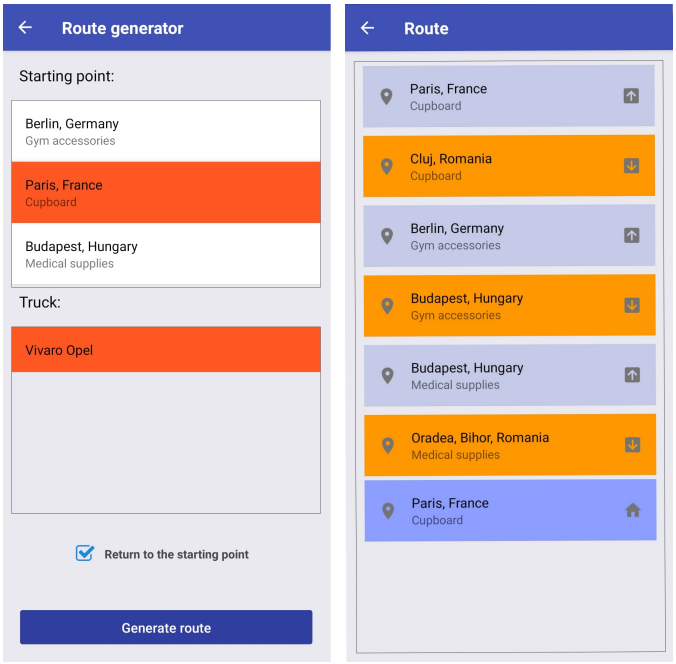

Fig. 6. The interface of the optimal route generator feature.

route after finishing it. Upon the answer arrives, the next screen exposes the route. A list will be available with the locations in visiting order. By tapping on them, the app brings up the local map application with the precise coordinates of the selected entry. Entries are differentiated through colors and icons. Pickup locations are shown with a light blue background, while destination locations with orange.

VII. Conclusion and Future Plans

The paper presented the Truxi application, its architecture, the used technologies and the features of the system. The application is meant to solve a real-life problem of two groups of people. It provides a way for cargo suppliers to conveniently transport objects and at the same time ensures that freelancer drivers can continue to make a living out of their job. The service provided by the system links together transporters and their clients. Suppliers can save costs by using the service because drivers will keep on under-bidding each other in order to own the lowest offer made. The lower the price the more favorable the deal is for the supplier. Drivers can benefit from the system by being able to make offers on various available cargoes at the same time. They can also generate an optimal route after they gathered a few cargoes to transport.

There are a few more features that are meant to be implemented in the future.

- The distance function used in the optimal route generator feature should be changed, the real geographical distance between the locations should be used.
- Suppliers could have a feature, which lets them track the whereabouts of the driver, who is transporting their cargo at the moment.
- A price estimator would help the user to establish an initial price for the new cargoes.

References

[1] R. T. Fielding and R. N. Taylor, “Architectural styles and the design of network-based software architectures,” Ph.D. dissertation, University of California, Irvine, 2000. [Online]. Available: https://www.ics.uci.edu/fielding/pubs/dissertation/rest_arch_style.htm

[2] Inversion-of-Control. Ralph E. Johnson & Brian Foote (June–July 1988). ”Designing Reusable Classes”. Journal of Object-Oriented Programming, Volume 1, Number 2. Department of Computer Science University of Illinois at Urbana-Champaign. pp. 22–35. Retrieved 29 April 2014. [Online]. Available: http://www.laputan.org/drc/drc.html

[3] Data-Transfer-Object pattern, Martin Fowler. [Online]. Available: https://martinfowler.com/eaaCatalog/dataTransferObject.html

[4] Single-responsibility principle. Martin, Robert C. (2003). Agile Software Development, Principles, Patterns, and Practices. Prentice Hall. p. 95. ISBN 978-0135974445.

[5] 2-opt algorithm description. [Online]. Available: http://pedrohfsd.com/2017/08/09/2opt-part1.html

[6] Spring Boot documentation. [Online]. Available: https://spring.io/projects/spring-boot

[7] ReactJs official documentation. [Online]. Available: https://reactjs.org/

[8] React Native official documentation. [Online]. Available: https://reactnative.dev/

[9] MobX official documentation. [Online]. Available: https://mobx.js.org/README.html

[10] Cloudinary Java intergration official documentation. [Online]. Available: https://cloudinary.com/documentation/java_integration

[11] Docker virtualization tool, official documentation. [Online]. Available: https://www.docker.com/

[12] Gradle build tool, official website. [Online]. Available: https://gradle.org/

[13] Liquibase database schema manager, official documentation. [Online]. Available: https://www.liquibase.org/documentation/index.html

[14] InversifyJS dependency injection framework, official documentation. [Online]. Available: https://github.com/inversify/InversifyJS

[15] Formik form handler library, official documentation. [Online]. Available: https://jaredpalmer.com/formik/docs/overview

[16] Front Controller design pattern, description. [Online]. Available: https://www.tutorialspoint.com/design_pattern/front_controller_pattern.htm

[17] Applegate, D. L.; Bixby, R. M.; Chvátal, V.; Cook, W. J. (2006), The Traveling Salesman Problem, ISBN 978-0-691-12993-8