SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF COMPUTER SCIENCE
MASTER DEGREE IN CYBERSECURITY

# Remora: a decentralized carpooling application based on the Ethereum blockchain

## BLOCKCHAIN AND DISTRIBUTED LEDGER TECHNOLOGIES

**Professor:**
Claudio Di Ciccio

**Students:**
Alessandro Calandrelli
Leonardo Cataldi
Francesco Goberti
Martino Porceddu
Matteo Vicari

Academic Year 2023/2024

# Contents

# 1 Introduction

## 1.1 Preface

Remora is a distributed car-pooling application created to make as many people as possible to contribute polluting the environment less and less. The application provides a registration mechanism and, after that, a user will be able to provide a car ride to other users of the application in exchange for particular tokens (Gears) that have a corresponding Ethereum value. When a user completes a certain amount of trips as a driver, he can self-mint tokens as an incentive to use the application. Furthermore, the application allows users to sell and buy tokens in exchange for Ethereum.
This project has been developed by 5 people divided into two teams:

- **Smart contract and solidity team**:
  Alessandro Calandrelli, Leonardo Cataldi, Francesco Goberti;
- **Front-end and back-end team**:
  Martino Porceddu, Matteo Vicari.

The team's sub-division was made just for work and time splitting, but in the end all 5 participants worked on the entire project completely.

## 1.2 Report outline

The document first of all will cover some Blockchain theory topics. Afterward, it will discuss the Remora application context and its aim in more detail, in particular, it will explain why the use of the blockchain was necessary and how it has improved the project itself. At this point, we will be in the report core, where it will discuss all the software architecture and the implementation details by using also some diagrams and some pictures to make it clearer.
Finally, it will talk about some limitations we have found during the application development, possible solutions to those limitations, and future remarks.

# 2 Background

## 2.1 What is Blockchain?

Blockchain can be defined as an "*open, distributed ledger that can record transactions between two parties efficiently and in a verifiable and permanent way*"[1]. In other words, blockchain is a protocol that allows multiple, distributed nodes to keep track of transactions maintaining a decentralized, immutable ledger that stores them in an ordered and permanent way. A **ledger** is an ordered, append-only, list of transactions. More precisely, a ledger is a totally ordered set without complement. Modern ledgers are typically digital, centralized ledgers, such as online banking accounts. However, centralization creates a problem of **trust**. Any user of a centralized ledger has to trust who manages it, hoping that the ledger is *valid*, *complete*, *unaltered*, and that *it won't be destroyed*. The blockchain aims at solving all those issues through decentralization. Each node can transact with the others, and stores a local copy of the entire ledger. In this way, there is no single point of failure, which makes the ledger almost impossible to be lost. The validity, completeness, and integrity of the ledger are ensured by the protocol features and can be verified by every user.

## 2.2 Blockchain history

The idea of a digital currency arose since the diffusion of the internet to the public. The first attempts to produce a protocol for anonymous electronic payments, however, failed due to the usage of a centralized intermediary. At the same time, the usage of decentralization was hindered by severe issues, the main one being having to deal with potentially byzantine users. In 2008, however, Satoshi Nakamoto pseudonymously published the paper "*Bitcoin: A Peer-to-Peer Electronic Cash System*"[2]. In this article, he described the first real blockchain, solving the issues of decentralization with a brilliant usage of digital signatures and Hal Finney's Reusable Proof of Work (RPoW) algorithm, ensuring the validity of transactions and allowing the protocol to keep functioning and reach consensus despite the presence of some byzantine nodes in the network. Since Nakamoto's article, the protocol has been expanded and improved, leading to the creation of many other chains. In particular, modern blockchains tend to use the more efficient Proof of Stake instead of Proof of Work, and typically allow, aside from the transacting functionality, the execution of code and the memorization of a state in a distributed environment.

## 2.3 Blockchain functioning

In this paragraph, we want to summarize which are the main concepts behind blockchain functioning. As already said, blockchain is a protocol that allows to maintain a

distributed ledger of transactions. The ledger is updated thanks to the periodic publication of blocks, i.e. collections of transactions, by specific nodes, called mining nodes. For their effort, mining nodes receive a reward. To keep the order of the ledger, both the transactions in the blocks and the blocks themselves are ordered. To keep the order of the blocks, every block refers to the previous one via its hash. Each transaction is a transfer of coins (Ether, Bitcoin, etc...) between a sender and a receiver. Digital signatures ensure the validity of transactions, allowing only those who know an account's secret key to spend that account's assets. To decide who will publish the next block, and to agree on the view of the blockchain, the nodes that participate in the protocol run a **consensus** algorithm. Blockchains are mainly divided into:

- **Proof of Work** blockchains: the right to publish a new block is earned by solving a complex computational puzzle, and the valid chain is the one with the highest computational effort put into it.
- **Proof of Stake** blockchains: the right to participate in the consensus algorithm is earned by putting at stake a certain amount of crypto-fuel (coins). The right to publish is assigned randomly and the consensus is reached via votes issued by all the participants to the algorithm (validators).

## 2.4 Blockchain application domain

We can identify more or less two types of blockchains in terms of functionality. There are blockchains, such as Bitcoin, that have been only thought of as a decentralized alternative to the current banking system. Bitcoin, as well as similar chains, does not provide programmability, but its features are limited to ensuring a secure and decentralized way to perform and store transactions.
New generation blockchains instead, such as Ethereum, are not only ordered collections of transactions. They can be used as a decentralized computer to execute programs. In Ethereum, programs living in the blockchain are called *Smart Contracts*, they are deployed by Externally Owned Accounts or by other contracts, and have their own address. Smart Contracts expose functions that, if called via a transaction, will be executed by the entire blockchain network. Thus, blockchains like Ethereum can be used not only as simple ledgers but also as a decentralized backend for distributed applications (DApps). A typical concept in programmable blockchains is the concept of ***token***. There are different types of tokens, that can be *fungible*, *non-fungible*, or *semi-fungible*. In particular, fungible tokens can be considered another full-fledged type of *cryptocurrency* created and managed by the Smart Contract that defines it.

# 3 Presentation of the context

## 3.1 Aim of the DApp

First of all, we have to explain why we have chosen this project name since it is representative of the aim of the DApp. So, what is a Remora? Remora is a fish that travels in symbiosis with other fishes and attaches itself to boats and divers. It symbolizes companionship and mutual benefit, which are in our mind the best elements for humans to solve the car pollution problem.

Now, let's suppose that there is a particular event in Milan and a lot of people are moving there from the south of Italy: all of them would go by themselves, each one taking their car. This is happening daily, causing a low average occupancy of a vehicle. According to data from different sources, we have observed that only 1.7 people travel in the same car[3], in particular Italy is representative of this problem since it is the third country with the most cars per inhabitant in Europe. On the other hand, across the Atlantic Ocean, the situation isn't any better.

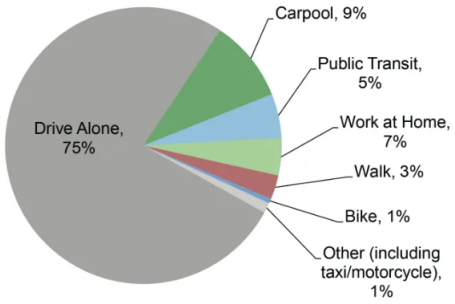



Figure 1: *U.S. Modes of Transportation to Work 2020*

The results of this problem are both high costs for every single driver and a huge amount of traffic pollution and *Co2* emissions. The app aims to raise the average occupancy of cars to 2.8 people per car and therefore reduce costs and *Co2* emissions by at least 30%.

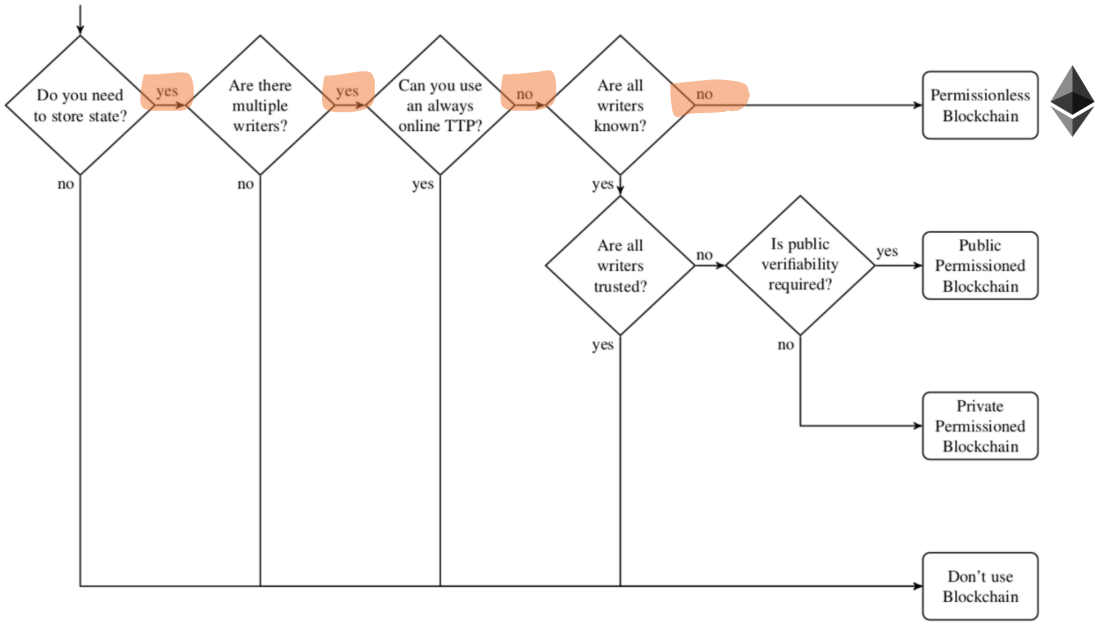
Then, to induce people to share their rides, we have developed the **Gear** Token. A *Gear* is a *fungible* token used on the application as a currency. It can be purchased by anyone through our Ethereum smart contract. Furthermore, drivers will decide the cost of the trip in terms of Gears, receive them as a payment, and as a reward for their commitment.

To facilitate user interaction with the smart contract, we have developed a website in which users can post, book, and pay for trips, but also sell and buy Gears in exchange for Ethereum. If the user decides to use the website functionalities, everything must

be performed using the Metamask browser extension. This implies that he needs to have a personal wallet with some Ethereum inside it. The DApp core functionalities are still accessible by directly interacting with the smart contract.

## 3.2 Why do we need to use the blockchain?

The blockchain allows us to immediately spread the application all over the world since anyone can join without any restriction.




- *Do we need to store state?*
  **yes**, we need to store trips state and other information to certify them;

- *Are there multiple writers?*
  **yes**, multiple untrusted users should be able to post and book trips by modifying the state or directly interacting with the smart contract;

- *Can we use an always online TTP?*
  **no**, because the application system cannot rely on a central entity. In fact, if this central control point becomes inaccessible or compromised, the entire system could be compromised;

- *Are all writers known?*
  **no**, because anyone could be both a driver or a passenger in our application.

This lead us to a **public permissionless** blockchain, which is Ethereum.

# 4 Software Architecture

## 4.1 DApp Infrastructure

The figure 2 below represents the infrastructure of our DApp. The website is locally hosted through an *Apache* web server that interacts with a *php* back-end, used to communicate with a database implemented on a *MySQL* service. To manage the database more easily, we have used the *MySQL Workbench* suite.
The hosted website was developed using *html*, *css* and *javascript*. As a consequence of this, the blockchain connection was executed using *web3.js* modules of *javascript*. The link between *web3.js* and our private Ethereum blockchain (*Ganache*) was performed with *Metamask*.
Our Smart Contracts (*RemoraGears* and *GearsVendor*) have been developed using *Solidity* and some particular *OpenZeppelin* modules (*ERC20.sol*, *Ownable.sol* and *ReentrancyGuard.sol*). Finally, both Smart Contracts have been deployed on our *Ganache* blockchain through *Truffle*.

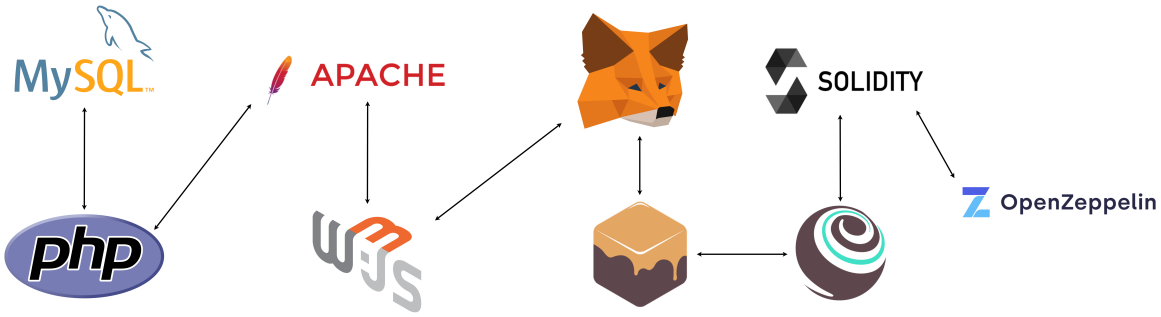



Figure 2: *Schematic representation of the DApp Infrastructure*

## 4.2 Database Infrastructure

The database contains 5 tables, each one representing a key element of our DApp service.

- **users**: This table contains the user information used to manage the login and registration mechanism, but also relevant statistics about his trips and minted Gears.
- **trips**: This table contains all the necessary information about trips posted by users on the website but not stored on the blockchain yet.
- **ongoing_trips**: When a trip is booked by at least one passenger and accepted as such by the driver, it will be entered in this table. This table will have a single entry for the driver but multiple entries for passengers (one for each) that share the same trip.

- **notifications**: This table manages all the communications between website users.
- **history_trips**: When a trip is finally ported into the blockchain by the driver, it will be inserted into this table. This table contains all the information necessary to eventually certify the trip.

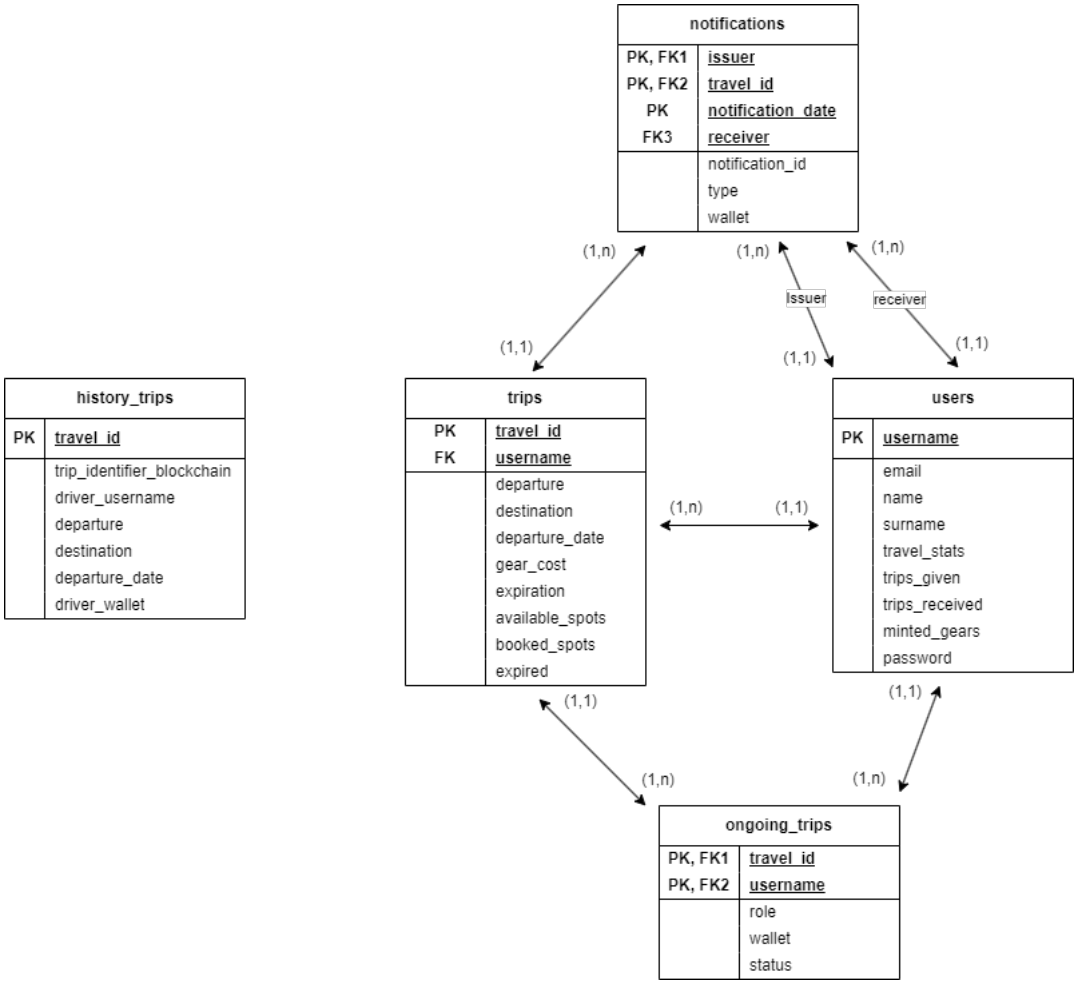



Figure 3: *Database class diagram*

## 4.3 Smart Contracts Infrastructure

The DApp uses two different Smart Contracts: *RemoraGears* and *GearsVendor*. Both the contracts have been written in Solidity, and each contract manages one of the two main aspects of the application: the management of the trips, handled by *RemoraGears*, and the token marketplace, handled by *GearsVendor*.

The following figure contains the UML class diagram of the *RemoraGears* Smart Contract:

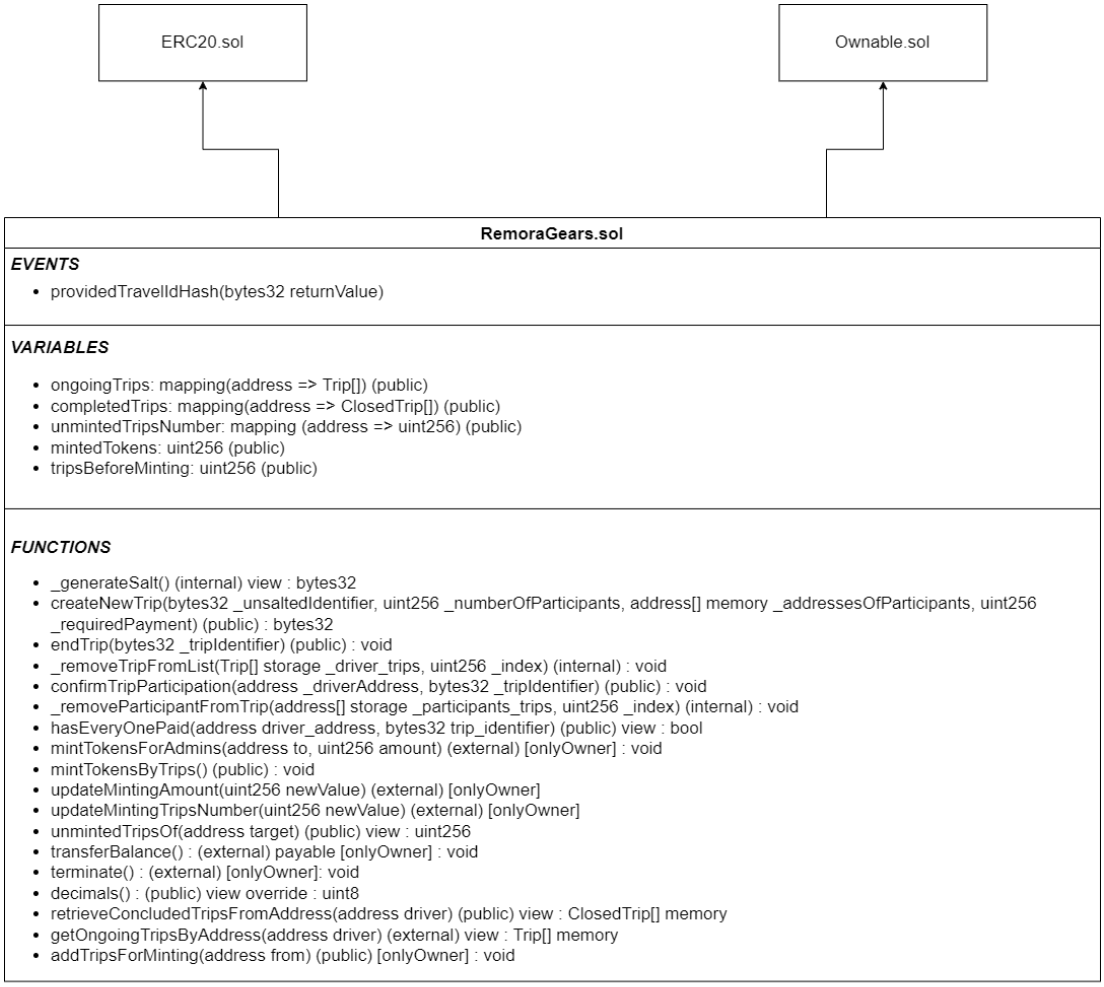



Figure 4: *RemoraGears smart contract diagram*

As it can be seen by the diagram, the contract inherits from two *OpenZeppelin* modules:

- *Ownable.sol*, which allows us to define the concept of ownership of the contract. The owner of RemoraGear, by default, is the address that deployed it. *Ownable* also defines the modifier *onlyOwner* that, when used in a function, makes it invokable only by the contract's owner.

- *ERC20.sol*, which implements the *ERC20* standard for fungible tokens. The *Gear* token is an ERC20 token with 0 decimals.

The contract defines many variables and functions for the handling of trips; their usage will be detailed later. Of particular interest are the two structures used for the management of the trips: *Trip* and *ClosedTrip*.

```
1 struct Trip {
2     bytes32 tripIdentifier;
3     uint256 numberOfParticipants;
4     address[] addressesOfParticipants;
```

```
5       uint256 numberOfConfirmations;
6       uint256 requiredPayment;
7       bytes32 salt;
8   }
```

Listing 1: Trip structure definition

```
1   struct ClosedTrip{
2       bytes32 tripIdentifier;
3       bytes32 salt;
4   }
```

Listing 2: ClosedTrip structure definition

Notice that the *Trip* structure contains an identifier, which is a 32 bytes hash created by the website and by the *createNewTrip* function, the number of participants to the trip, a list of addresses to identify the participants, the number of participants that have confirmed their participation to the trip, the payment required by the driver, and the salt used to create the identifier. A new instance of this structure is created every time a user decides to store a trip on the blockchain (via the *createNewTrip* function), and is destroyed when the user concludes the trip (via the *endTrip* function).

Notice that, instead, the *ClosedTrip* structure contains much less information. A new *ClosedTrip* is created when a *Trip* is destroyed and remains permanently on the blockchain for verification purposes. Since its only purpose is, as said, verification, the only data that are kept are the identifier, which as already said is obtained via a hash, and the salt used, to be able to verify the hash.

### 4.3.1 Trip verification

The verification of a trip is made possible because the trip identifier is computed as:

$$keccak256(sha256(DriverUsername|Departure|Destination|DepartureDate), salt)$$

and all those data are retrievable, to recompute the identifier and compare it with the one stored on-chain. Further details on the verification will be given in section 5.

The following figure contains, instead, the UML class diagram of the *GearsVendor* Smart Contract:

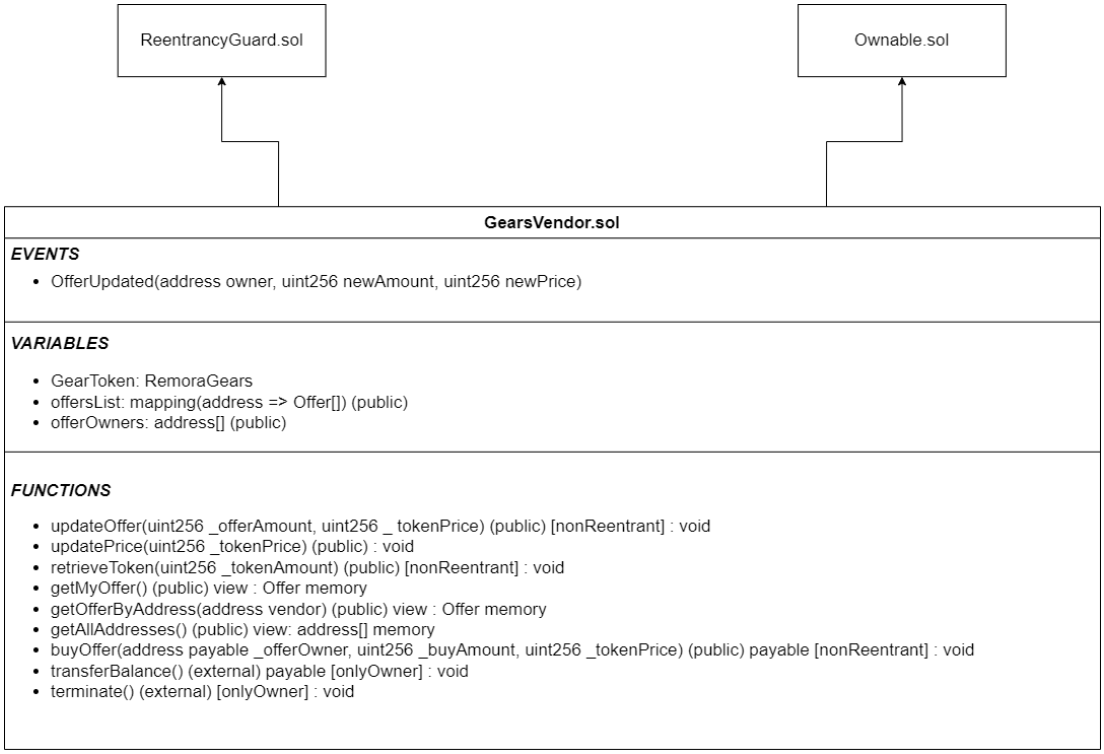



Figure 5: *GearsVendor smart contract diagram*

As it can be seen by the diagram, also this contract inherits from two *OpenZeppelin* modules:

- *Ownable.sol*, which is used for the same purpose for which it is used in the RemoraGears contract.
- *ReentrancyGuard.sol*, which implements the *nonReentrant* modifier. This is a modifier that is useful to avoid reentrancy attacks when calling other Smart Contracts.

As already specified, this contract manages the token marketplace and allows users to buy and sell tokens from and to other users. Those functionalities are mainly achieved thanks to the following *Offer* structure.

```
1 struct Offer{
2     bool updated;
3     uint256 offerAmount;
4     uint256 tokenPrice;
5 }
```

Listing 3: Offer structure definition

Each offer is associated with an address and has a boolean value, which is just needed to keep track of which addresses have actually created an offer and is modified only at the creation of the offer itself, an amount of Gears offered, and a price per Gear.

Every time a user wants to sell some of its Gears, he will transfer them to the vendor's balance. The vendor, upon request, will send those gears back to the original owner, or sell them to another user, giving the owner the corresponding price in Ether. All those operations update the Offer structure associated with the original seller. Further details will be given later.

### 4.3.2 Interaction among the two contracts

The following UML class diagram describes the connection between the two contracts.

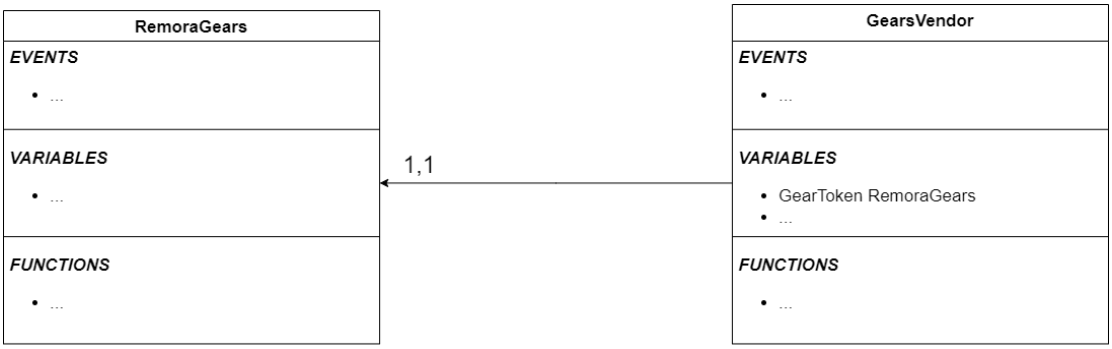



Figure 6: *Connection between RemoraGears and GearsVendor*

As it can be seen by the figure, GearsVendor includes, among its variables, an instance of RemoraGears, to be able to call it and thus perform the various Gears transfers.

## 4.4 Use Case Diagrams

As already mentioned, the two main app functionalities consist of the management of the trips and the Gear token marketplace. Regarding the management of the trips, each user can organize a new trip, and store it on the blockchain to finalize it. Regarding the token marketplace, instead, each user can post an offer and manage its amount and its price. Other users can buy tokens from the available offers.

In the following paragraphs are shown some use case diagrams, which describe the interaction among the users and the application in the main use cases.

### 4.4.1 Trip life-cycle diagram

The first use case diagram to be considered is the one regarding the main functionality of the DApp: the **trip life-cycle** diagram.

First of all, a trip is posted on the website by a user. As a consequence of this, the trip becomes bookable, and the user that has posted that trip will assume the *driver* role in the context of that trip.

Instead, another user who wants to book a trip will have to check the available trips, by searching them on the website. Once he has decided which trip he wants to be part of, he has to book it. As a consequence, a trip request notification will be sent to the trip driver, who will be able to accept or reject the passenger. Assuming that the driver accepts the passenger, the trip will remain available for other passengers until there are remaining spots and the expiration date for the trip doesn't pass.

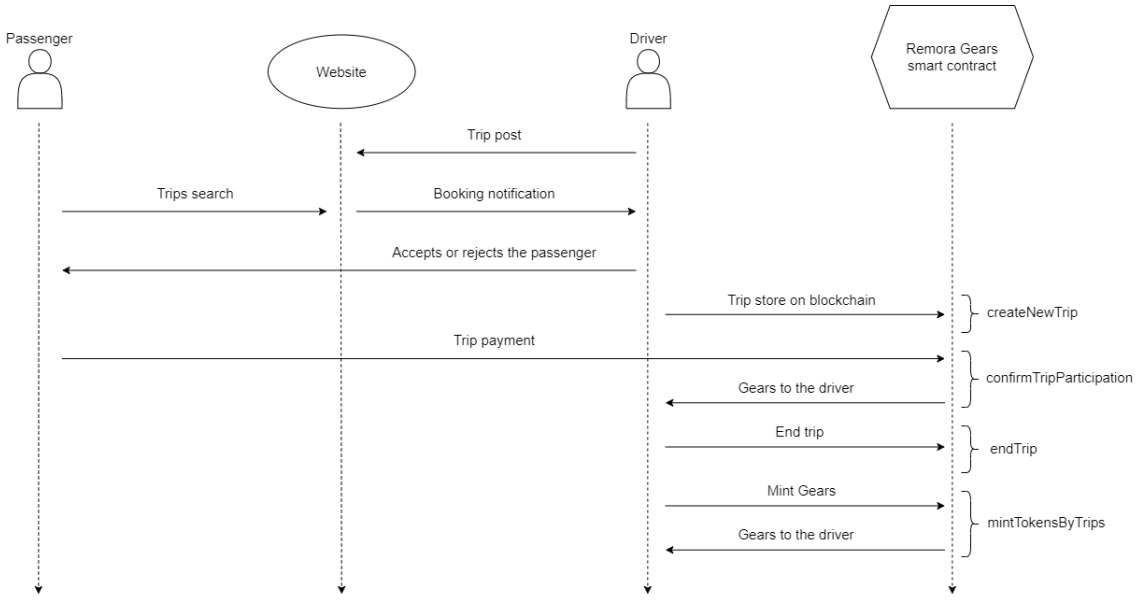



Figure 7: *Trip life-cycle*

Once having received a satisfying amount of bookings, the driver can store the trip on the blockchain, calling the *createNewTrip* method of the *RemoraGears* Smart Contract. After this crucial step, and hopefully, after the trip has been concluded, all the passengers involved in the stored trip will be able to pay the driver, calling the method *confirmTripParticipation* of the *RemoraGears* Smart Contract. In this function, the expected Gears will be transferred from the passenger account to the driver account. Once all passengers have paid, the driver will be able to end the trip, which will be permanently stored on the blockchain for an eventual future certification of the fact that the trip has happened. This is done by calling the method *endTrip* of the *RemoraGears* Smart Contract.

Finally, if a user has completed a specific amount of trips as a driver, he will be able to self-mint some Gears to his account by using the method *mintTokensByTrip* of the *RemoraGears* Smart Contract.

Notice that the roles of *driver* and *passenger* are related only to a specific trip. The driver of a trip can be a passenger in another one and vice-versa.

### 4.4.2 Gears marketplace diagram

The second use case diagram that has to be analyzed is the **token-trading system**. This diagram describes how users can buy and sell Gears on the platform.

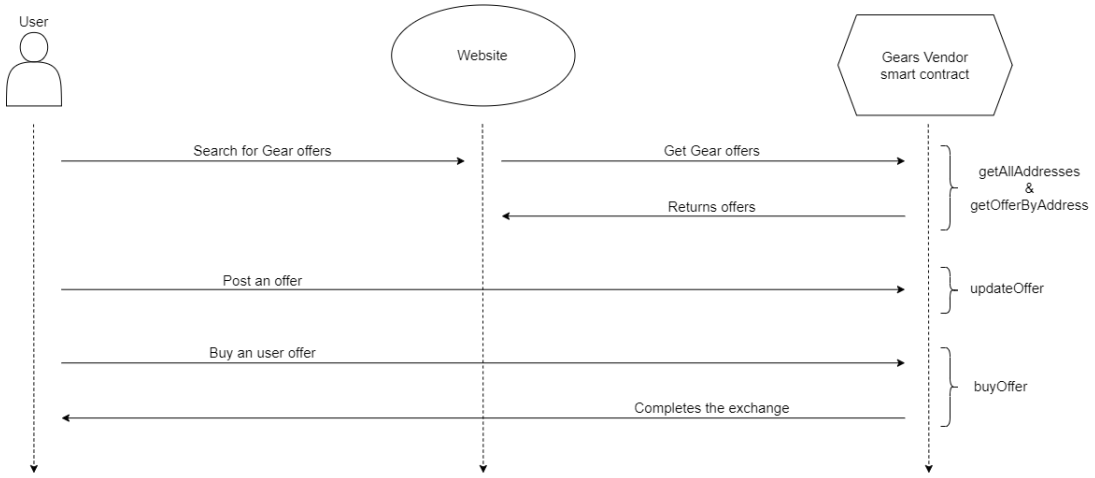



Figure 8: *Token-trading system*

Visiting a specific page on the website, a user can check the Gears offers that are currently available. Since they are stored on the Smart Contract, to show them to the user the website calls the methods *getAllAddresses* and, for each address retrieved, *getOfferByAddress* of the *GearsVendor* Smart Contract. Those calls allow the website to obtain a list of all the available offers. Notice that the calls are performed via the *call()* method of *web3.js*, and not via *send()*, since both the functions are *view* and thus we don't need to send transactions.

From the same page, a user can post a Gears offer on the blockchain, calling the *updateOffer* method of *GearsVendor*, and can buy other user's Gears, using the method *buyOffer*. Notice that before calling *updateOffer*, the user must call the *approve* function on *RemoraGears* Smart Contract to approve the transfer of Gears from his balance to the contract's one.

### 4.4.3 Update an offer diagram

The third use case diagram that has to be analyzed is the **offer update mechanism**. It describes all the possible ways through which an offer can be updated by the owner. Visiting the *Profile* page, each user can manage his offer:

- **Update ETH Price**: this feature allows the user to update the Ethereum price of an offer that is already on the market.
- **Retrieve Gears**: this feature allows the user to retrieve a specific amount of Gears from the offer that has already been posted on the market.

- **Add Gears**: this feature allows the user to add a specific amount of Gears to the offer that he has already posted on the market.

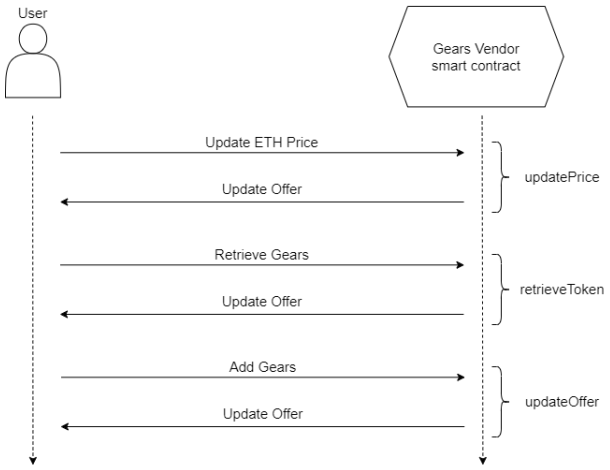



Figure 9: *Offer system*

Notice that all these actions are performed by calling respectively the *updatePrice*, *retrieveToken*, and *updateOffer* methods of the *GearsVendor* Smart Contract.

Notice that, even if the use case diagrams illustrate the Smart Contract interactions to be performed directly, those actions are performed using the website GUI and *web3.js*.

## 4.5 Interaction among application components

The following diagram describes the interaction among the application components. Since the interaction is similar for all the functionalities of the application, the diagram is generic.

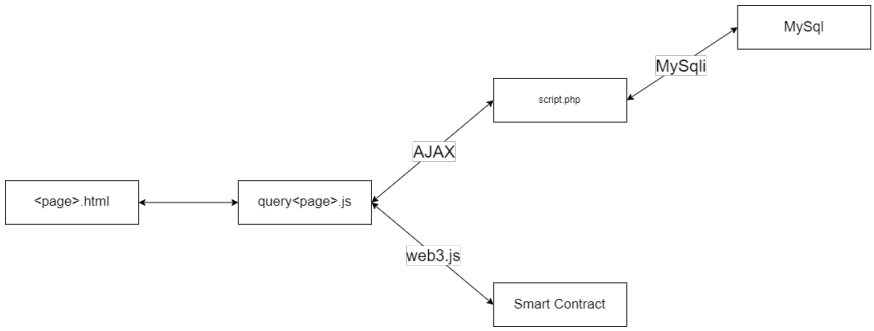



Figure 10: *Interaction among the application components*

As it can be seen in the figure, the *html* of the visualized page interacts with a *Javascript* file associated with it. According to what the user performs, the *Javascript* file may interact with the *php* backend through *AJAX* calls, or with the *Smart Contracts* via *web3.js*. The interaction between the *php* backend and the *MySql* server is performed via the *MySqli* library.

# 5 Implementation

Before delving further into this section, it should be emphasized that the entire explanation of the implementation details must be divided between the two main aspects of our DApp: the *trip life-cycle* and the *Gear market mechanism*.

## 5.1 Trip life-cycle implementation

Let's suppose that Alice and Bob decided to use our distributed application. The first thing that each user has to do is to register himself through the apposite registration page. Once a user completes this phase, he will be able to login to the website, through the login page, with his own authenticated session.

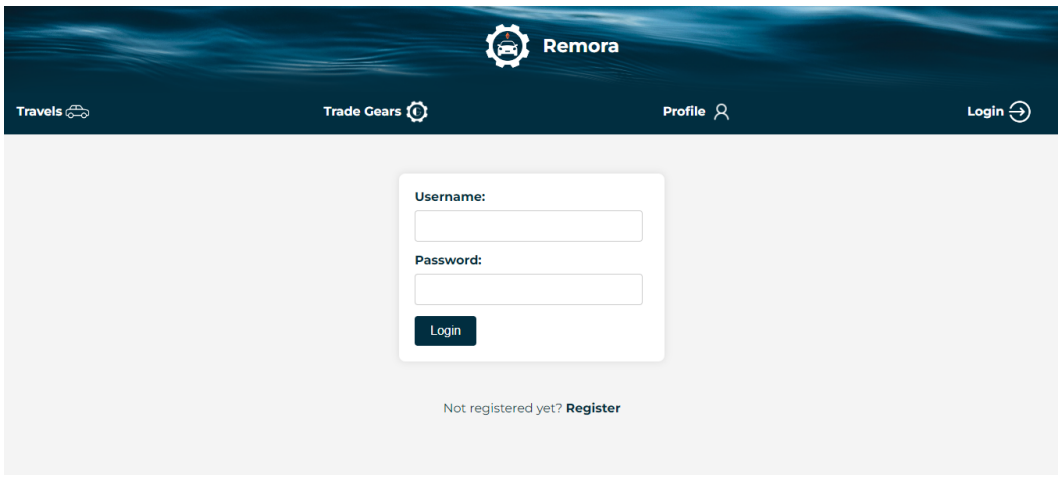



Figure 11: *Login page*

Notice that the database stores users' passwords in a hashed format as a security measure.

The principal page of the website is the *Travels* page, where a user can search for all the available trips and post new ones.

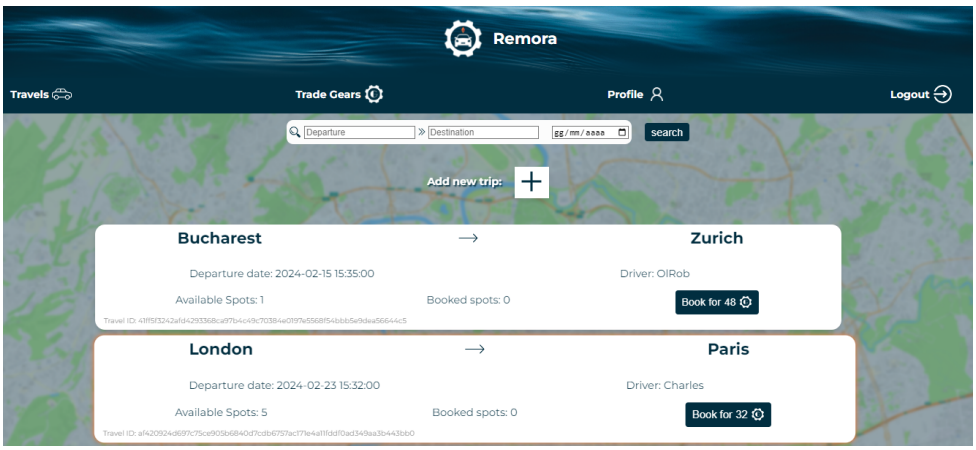



Figure 12: *Travels page*

In our demonstrative case, Bob will assume the *driver* role and Alice will be the *passenger*. Bob decides to post a new trip on the website using the apposite post button "*Add new trips*".

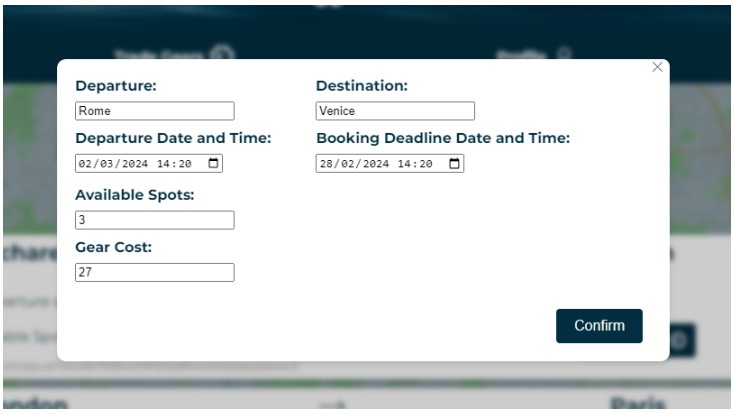



Figure 13: *Trip posting*

Alice decides to search for a trip departing from Rome and arriving in Venice.

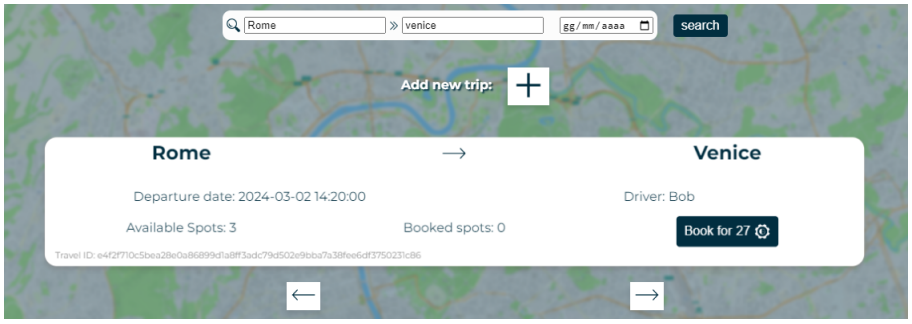



Figure 14: *Bob's trip*

Once she has found the trip she is interested in, by using the search bar to speed up the search itself, she can book it. Notice that Alice must be logged in on *Metamask* with the wallet that she wants to use in the future to pay for this trip. This is an implementation choice that will be explained later. If the trip booking is successful, a *trip request notification* will be sent to our driver Bob, who can check it inside the notifications section on his profile page. Recall that notifications are implemented through the database *notifications* table.


Figure 15: *Bob's trip request notification*

Even though they are not specified, there are a lot of controls put in place, while users perform the actions described above, to avoid misbehaviors. For example, a driver cannot book his trips.

Once Alice has requested to join a trip, Bob is free to accept or reject her by clicking the apposite notification button. If the driver decides to reject a passenger, a "*refused trip*" notification will be sent to him:

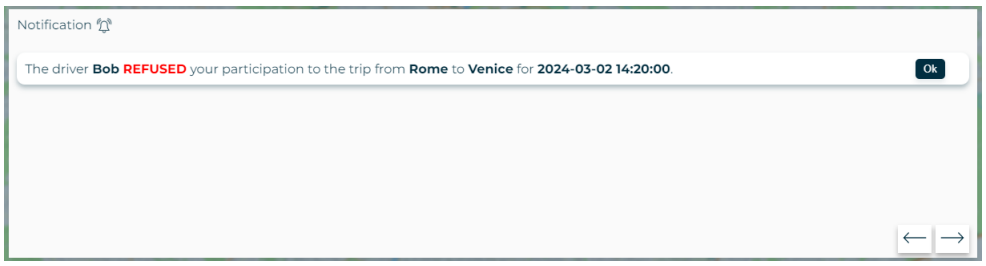



Figure 16: *Alice's participation refused notification*

Otherwise, the passenger will receive an "*accepted trip*" notification:

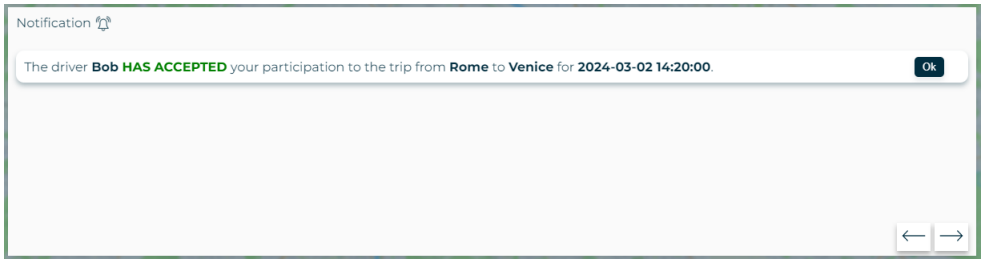



Figure 17: *Alice's participation accepted notification*

Assuming that Bob accepts Alice as a passenger, two new entries are added inside the *ongoing_trips* database table, one for the passenger and one for the driver. Notice that the entry related to the driver is added only once, the first time a passenger is accepted, to avoid redundancy.

Notice also that, when a passenger books a trip, the wallet he was using at that moment is added to the *notifications* table. If the passenger is accepted, this wallet is stored in the *ongoing_trips* table, in the row associated with the passenger.

Since the driver Bob has accepted Alice as a passenger for his trip, both the users can visualize the trip in the *OnGoing Trips* section of the profile page, which contains their active ongoing trips.

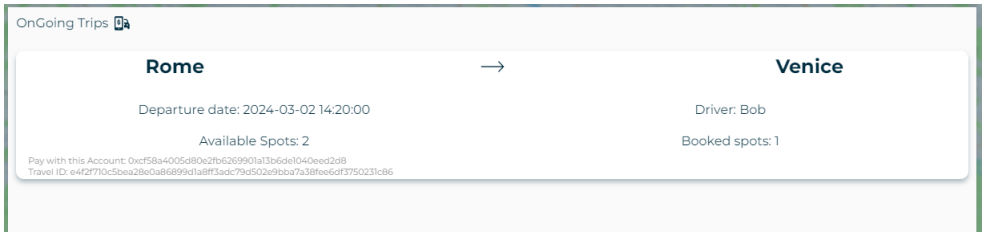



Figure 18: *Alice's ongoing trip card*

Notice that since Alice is a passenger for this trip, at the beginning her ongoing trip card has no buttons. Instead, at the beginning, the ongoing trip card of the driver Bob contains the *Store on Blockchain* button.

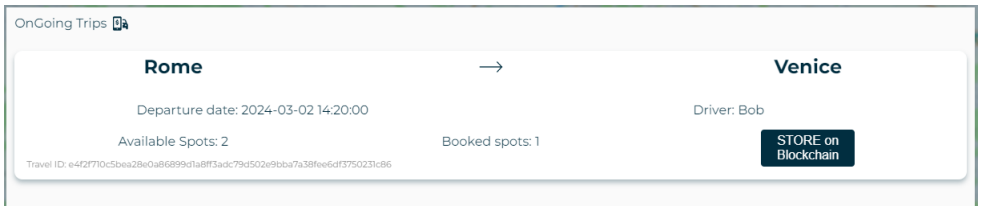



Figure 19: *Bob's ongoing trip card*

Since a passenger has booked the trip, Bob can store this trip on the blockchain. This is the first interaction with the *RemoraGears* Smart Contract: all the actions described before this one are handled off-chain. Storing a trip on the blockchain has a lot of consequences:

- From now on, the trip is stored on the blockchain as an *ongoing trip*, associated with the Ethereum account that Bob used to perform the *store* operation;
- No one will be able to book the trip anymore, because the trip won't be visualized anymore on the *Travels* page;
- The driver entry in the *ongoing_trips* table of the database will be updated with the address that the driver used to perform the *store* operation;
- The trip is added inside a specific database table called *history_trips*, which stores some important information about the trip itself, that will be necessary in the future if someone wants to certify that this trip has happened.
- Each passenger participating in the trip will visualize, in the ongoing trip card related to the stored trip, a *pay* button. This allows passengers to pay for the trip;
- The ongoing trip card of the driver will now contain the button *End Trip*.

After the *Store on Blockchain* action, these will be the two new ongoing trip cards:

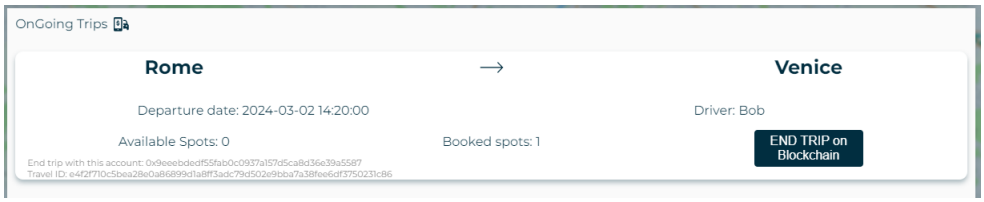



Figure 20: *Bob's updated ongoing trip card*

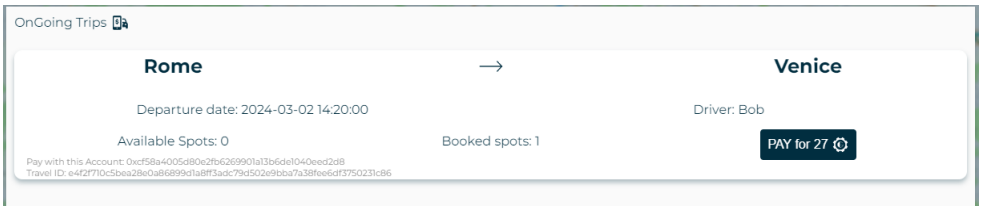



Figure 21: *Alice's updated ongoing trip card*

Alice, and all the other passengers that may be part of this trip, are now able to perform the trip payment. Of course, before this, the trip should physically happen. To pay, the passengers cannot use an account different from the one that they have used to book the trip, because the Smart Contract memorizes all the passengers' addresses to avoid someone else paying in their place. The website facilitates this action by specifying, in the card, the address that has to be used for the payment, and blocking any payment made with the wrong address before the call to the smart contract (to prevent forgetful users from spending useless fees). The receiver of the payment will be the driver's address, retrieved by the *ongoing_trips* table.


Figure 22: *Alice's confirming payment*

Thus, only if the passenger is paying with the correct address and has enough Gears to pay the driver, a *RemoraGears* Smart Contract function will be invoked to perform the payment. As a consequence of this, the ongoing trip card of the passenger will be updated again and a payment notification will be sent to the driver:

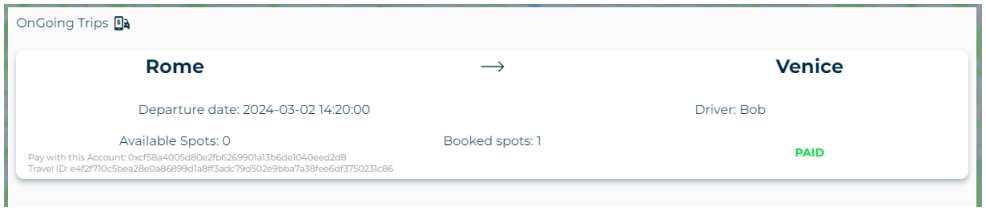



Figure 23: *Alice successfully paid*


Figure 24: *Bob's payment received notification*

At this point, once all passengers have paid for the trip, the driver can conclude the trip by using the *End Trip on Blockchain* button on his ongoing trip card. This button will invoke a *RemoraGears* Smart Contract function that will permanently close this trip on the blockchain. The Smart Contract function will work only if all the passengers have paid, that's why if the driver tries to press it before everyone has paid, the website will not call the Contract but instead will notify the driver that not everyone has paid.

Once the driver has performed the *End Trip on Blockchain* action, the trips will be deleted from the *OnGoing Trips* section on the profile page of both the passengers and the driver. This is because the trip is removed directly by the *ongoing_trips* table and by the *trips* table of the database. The trip remains in *history_trips* for future certification.

### 5.1.1 Some details about the trips

The above paragraph describes the implementation of a trip life-cycle. However, some details have not been covered and instead have to be explained to have a full understanding of the application's functioning.

Every time a new trip is created on the database by a driver, a *travel_id* is computed. The presence of a *travel_id* in the table can also be seen inspecting the database class diagram in figure 3. This ID is generated as a hash (*sha256*) of the driver's username, the departure, the destination, and the departure date, and is a primary key (in the *trips* table). This prevents a user from creating multiple identical trips.

When a trip is stored on the blockchain, the RemoraGears Smart Contract function that handles the store (*createNewTrip*) takes as input the *travel_id* taken from the

database to uniquely identify the trip among the ongoing trips (and, once the trip will be completed, among the completed trips) associated to the driver's address. To avoid a user who knows how the ID is computed pushing two identical trips on the contract, tampering with the certifiability of the application, *createNewTrip* generates some salt and uses it to hash again the *travel_id*, this time using *keccak256*. The salt and the final ID obtained with this operation are saved, and the final ID is returned to the website through an event. The website function that posts the trip awaits the transaction receipt before terminating, and retrieves from it the event and thus the final *trip identifier*. This identifier is saved inside the *history_trips* database table.

If someone wants to certify that a specific trip has happened, he can inspect the *history_trips* table and retrieve the trip identifier and the driver's address. He can then use the address to call the function *retrieveConcludedTripsFromAddress* of the *RemoraGears* Smart Contract, and the *trip identifier* to verify that the trip is in the list of concluded trips. Since the *ClosedTrip* structure includes the salt used to hash the original *travel_id*, the verifier can compute from scratch the identifier and verify that it matches the one stored on-chain.

## 5.2 Gears minting for drivers

Once a driver has performed a specific number of trips, he will be able to self-mint some Gears to his Ethereum account. In particular, we have decided that the number of trips that a driver has to perform is exactly 10. Instead, the number of Gears that he will be able to mint every 10 given trips is 5.

At the beginning, the *Mint Tokens* button is not visible to the user:

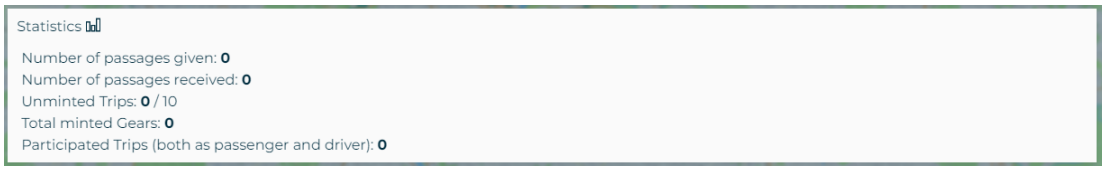



Figure 25: *Bob's statistics at the registration*

However, once the user has performed at least 10 trips as a driver, he will be able to see the button, that performs a call to the *RemoraGears* Smart Contract to perform the minting. The button is visible in the figure below.

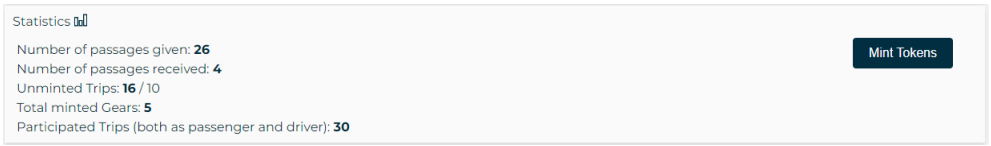



Figure 26: *Mint tokens button*

Once the driver presses the button, a certain amount of gears (in our case, 5) is added to his balance, while the number of *unminted trips* associated with him will be decreased by the number of trips that we want to reward (in our case, 10). The driver will keep seeing the button until his *unminted trips* become lower than the number of trips rewarded by the application, then the *Mint Tokens* button will disappear again.

Notice that the amount of Gears minted to the driver is hard-coded in the Smart Contract, but can be changed via a function that can be called only by the contract's owner. The same holds for the number of trips to perform before being able to mint tokens.

## 5.3 Gears marketplace implementation

The marketplace of our DApp can be reached by accessing the *Trade Gears* website page.

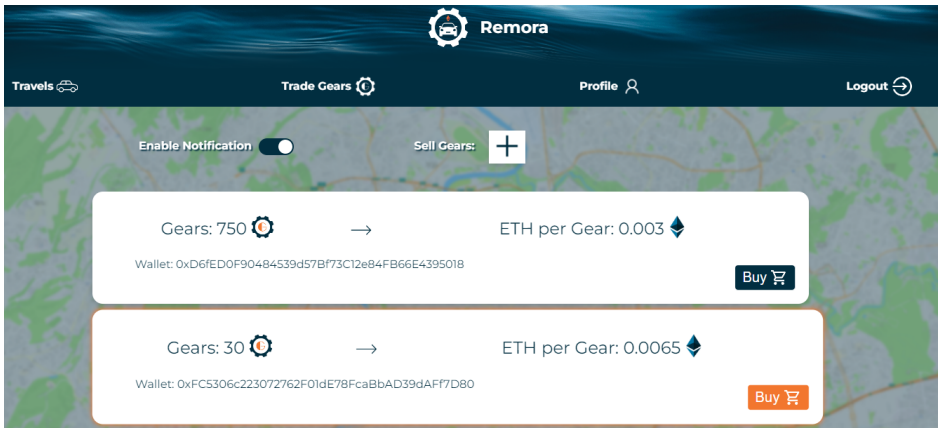



Figure 27: *Trade Gears website page*

Each user that has a wallet can post a new offer. To do this, he can use the "*Sell Gears*" button and compile the apposite form with the number of Gears that he wants to sell and the Ethereum price per Gear. Notice that each Ethereum address is associated with a single offer, thus a user can post multiple offers only if he has multiple Ethereum accounts. Pressing the "*Confirm*" button will push the offer interacting with the *GearsVendor* Smart Contract.

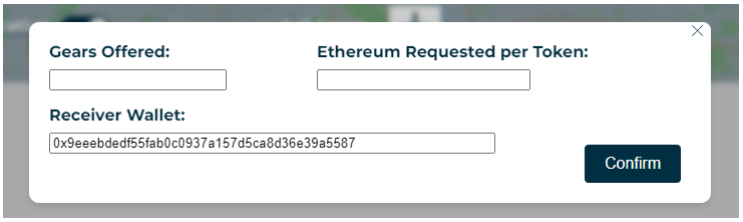



Figure 28: *Sell Gears form*

Once the offer is published on the website, all the users can visualize it on the *Trade Gears* website page. If a user decides to buy an offer, he has to click on the "*Buy*" button, and the apposite form will appear. The user has only to insert the number of Gears that he wants to buy and the total amount will be calculated in real-time in the field next to the Gears amount.

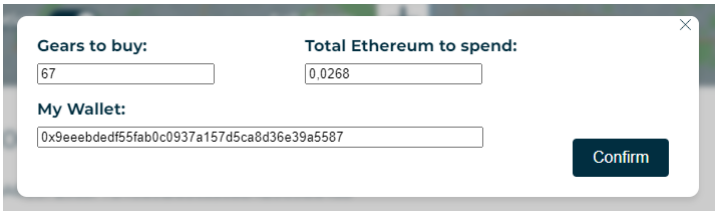



Figure 29: *Buy Gears form*

When the user decides to effectively buy an offer, he presses the "*Confirm*" button: this will cause the interaction with the *GearsVendor* Smart Contract.

Each user can visualize his offer through his *Profile* page.

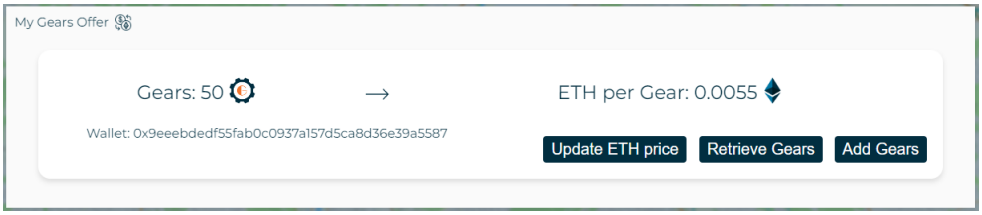



Figure 30: *My Gears Offer profile section*

From this specific card, a user can modify the offer price, retrieve some tokens from the offer, or add other tokens. Each action can be performed with the relative button, that will trigger an interaction with the *GearsVendor* Smart Contract.

Notice that tokens can be released to the public by minting them to the admin account (i.e. to our personal Ethereum account) and selling them, initially, at a low price. This can be done every time the app needs an injection of liquidity.

Finally, notice that when a user is visiting the *Trade Gears* page, he can enable notifications. If enabled, a notification will appear every time an offer is modified by someone who is not ourselves. This notification system is realized by subscribing to the **OfferUpdated** event emitted by *GearsVendor*.


Figure 31: *Updated offer notification*

# 6 Conclusions

## 6.1 Known issues and limitations

As with every kick-starting project, Remora has some issues that would need to be solved before a release to the public. In particular, given that its main aim is to certify that trips have happened, an important addition would be a push-in hardware oracle that physically verifies that a trip succeeded before allowing the driver to conclude the trip and to store it definitely on-chain. Also, to salt the trip IDs, the *RemoraGears* Smart Contract uses some block-related and user-related data. This approach however is not entirely safe, and it would be better to replace this salt with a random number provided by a random oracle.

## 6.2 Future remarks

Remora allows users to perform carpooling in a decentralized way, and to certify that a trip has happened thanks to the way the Trip ID is created. Future developments may include the integration of the website with some geolocalization APIs, the integration with oracles, and the creation of carbon credits based on the estimated *CO2* saved thanks to carpooling. Also, the website may be expanded to include some social networking functionalities that allow drivers and passengers to stay in touch and materially organize the trips.

# References


[1] Marco Iansiti, Karim R Lakhani, et al. The truth about blockchain. *Harvard business review*, 95(1):118–127, 2017.

[2] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, 2008.

[3] Miles Traveled. Personal transportation. *ile*, 33:1–5, 2010.

[4] Metamask website. `https://metamask.io/`. Accessed: 2024-01.

[5] Truffle website. `https://trufflesuite.com/`. Accessed: 2024-01.

[6] Web3.js documentation. `https://web3js.readthedocs.io/en/v1.10.0/`. Accessed: 2024-01.

[7] Erc20 ethereum standard. `https://ethereum.org/en/developers/docs/standards/tokens/erc-20`. Accessed: 2024-01.

[8] Openzeppelin erc20 implementation. `https://docs.openzeppelin.com/contracts/4.x/erc20`. Accessed: 2024-01.

[9] Apache website. `https://httpd.apache.org/`. Accessed: 2023-12 and 2024-01.

[10] Mysql website. `https://dev.mysql.com/`. Accessed: 2023-12 and 2024-01.

[11] Ganache documentation. `https://trufflesuite.com/ganache/`. Accessed: 2024-01.

[12] Solidity documentation. `https://docs.soliditylang.org/en/v0.8.24/`. Accessed: 2024-01.

[13] Claudio Di Ciccio. *Slides for the Blockchain and Distributed Ledger Technologies course*. Sapienza University of Rome, 2023/2024.

[14] Php documentation. `https://www.php.net/docs.php`. Accessed: 2023-12 and 2024-01.

[15] Html, css and javascript documentation. `https://www.w3schools.com/`. Accessed: 2023-12 and 2024-01.