Mobility Service Based on Blockchain Technology for SEAT MO

Degree Thesis
submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by

Adrián Soria Montoya

In partial fulfillment of the requirements for the degree in
Telecommunications Technologies and Services Engineering

Advisors: Jose Luis Muñoz Tapia and Antonio Jimenez Viguer
Barcelona, Date 2022

Contents

List of Figures

Listings

List of Tables

Abbreviations

ABI Application Binary Interface

API Application Programming Interface

AWS KMS Key Management System of Amazon Web Services

CORS Cross Origin Resource Sharing

CPU Central Processing Unit

dApp Decentralized application

DB Data Base

DNS Domain Name System

ECDSA Elliptic Curve Digital Signature Algorithm

EIP Ethereum Improvement Proposal

ERC Ethereum Request for Comment

EVM Ethereum Virtual Machine

FIAT Type of currency that is declared legal by some entity as the European Union

GSN Gas Station Network

JWT Json Web Token

MATIC Base Coin used in the Polygon network

MBTN Mumbai Test Net

NFT Non-Fungible Token

PoC Prove of Concept

PoS Proof Of Stake

PoW Proof of Work

RAM Random Access Memory

RHEL Red Hat Enterprise Linux

RLP Recursive Length Prefix

RPC Remote Procedure Call

SETK Seat ERC-20 Token

UTC Central Universal Time

VM Virtual Machine

Special Thanks

I would like to thank SEAT and all the IT department for the opportunity of doing the internship.
Also, I am very gratefull for the treatment received by all the FS-A3 team; Elena, Paco, Toni, Alvaro, Marcos, Crisanto, Sara, Carlos, Alan and Cathi. For all the things that I learned and to make me feel part of the team since the first day.
Specially I would like to thank Antonio Jimenez, my supervisor during my intership, for the opportunity of doing this thesis and for including me in the Metahype project sessions to learn how a project is developed in a big company.
To conclude, thanks to my project supervisors, Jose Luis Muñoz and Antonio Jimenez for the advising provided to the project.

Abstract

Security and privacy are vital nowadays, with all the leaks and tracking that every service uses. Because of that it is important to gain the trust of the users in our application. This thesis presents a PoC of a web dApp for managing the flow of the reservation of a vehicle. The technologies used are, React for the front-end, NodeJS for the back-end and HardHat suite with ganache to deploy and manage the contracts in a local blockchain and in the main blockchain. As for the blockchain, a layer 2 solution from the ethereum net is used, the Polygon network, in particular MBTN for testing. The dApp provides a transparent and secure solution that allows the user to truck all his movements in the app in an anonymous way and works perfectly as a propose for an utility case for the announced Metahype by Cupra [19].

Revision history and approval record

Revision	Date	Purpose
0	08/03/2022	Document creation
1	05/05/2022	Document revision
2	24/05/2022	Document revision
3	10/06/2022	Document revision
4	16/06/2022	Document revision
5	20/06/2022	Document validation

DOCUMENT DISTRIBUTION LIST

Name	e-mail
[Student name]: Adrián Soria	adrian.soria@estudiantat.upc.edu
[Project Supervisor 1]: Jose Luis Muñoz Tapia	jose.luis.munoz@upc.edu
[Project Supervisor 2]: Antonio Jimenez Viguer	antonio.jimenez@seat.es

Written by:		Reviewed and approved by:	
Date	08/03/2022	Date	20/06/2022
Name	Adrián Soria Montoya	Name	Jose Luis Muñoz Tapia and Antonio Jimenez Viguer
Position	Project Author	Position	Project Supervisor

1 Introduction

1.1 Statement of purpose

The purpose of this project is to implement a valid, transparent, and scalable solution to the mobility service that SEAT MO offers based on blockchain technology. The proposal should be functional, with good latency and response, and economically viable for a future implementation in the market. As it is a PoC, calls to the SEATMO API or payments with FIAT are simulated and the database used is local.

1.2 Requirements and Specifications

1.2.1 Requirements

Requirements are what the project must be in order to get completed. The requirements of my projects are:

1. Develop a scalable, transparent solution.
2. Secure communication between the different modules.
3. The project must be economically viable.
4. The contract should be written using the net standard ERC-20, to be used across new services.
5. The solution should be as decentralized as possible.

1.2.2 Specifications

Specifications make reference to the expected measured in order to see if the measures have been accomplished:

1. Metatransactions in the MBTN a layer 2 solution, so the user don't pay the gas fees.
2. A secure key management using Metamask or AWS KMS.
3. Make sure that the user and the company can track all the movements.
4. System has to have low latency in the user side, taking into account that transactions in the blockchain are not instantaneous and the user may take longer to release a vehicle than a transaction to being processed.

1.3 Methods and procedures

This project starts as a service proposal for the upcoming Metahype [19] anounced by Cupra. This project is a metaverse, a virtual environment where the user can be immersed and perform all kind of tasks, being the most famous, Meta by Facebook [12].
The blockchain technology is constantly evolving, and there are multiple programs and standards that this project will use. The most important one is the EIP-20 [24] which introduces the ERC-20, a fungible token that is used as virtual currency. Also, the EIP-712 [2] is the structure for the meta transactions that we are going to use. For the contracts, the project will be based in the Openzeppelin [3] contracts and use his Defender Relay [20] to use meta-transactions and allow the user to make transactions without paying any fee. The technology used in the front-end is React, and in the back-end are NodeJS and MongoDB. For the node, to have a participation in the MBTN, I am going to use Ansible, a Red Hat product to make the deployment and Heimdall and Bor are the core elements of the blockchain node.
The developing of the smarts contracts are going to be made in the Solidity programming language, and the for the deployment and testing, Hardhat and Ganache are going to be used respectively. All the structure is deployed in the EC2 of the Amazon Web Services, using internal communication when needed, and HTTPS when the communication is done in the internet.

1.4 Workplan, milestones and Gantt Diagram

1.4.1 Workplan

Project:	WP ref: 1
Major constituent: Documentation	Sheet 1 of 6
Short description: Decide which is the best way to proceed	Start event: 14/02/2022 End event: 01/06/2022
	Deliverables: - Presentation for SEAT MO Schematic of the system
Internal task T1: Decide the structure of the system Internal task T2: Plan the future tasks Internal task T3: Research about all the solutions and parts of the system and about the state-of-the-art current solutions.	Milestones: - Validation of SEAT MO - Defined structure

Project:	WP ref: 2
Major constituent: Software	Sheet 2 of 6
Short description: Configure the structure in AWS KMS	Start event: 16/02/2022 End event: 11/03/2022
	Deliverables: No
Internal task T1: Deploy the Polygon Node Internal task T2: Deploy the API machine Internal task T3: Deploy the Frontend Machine	Milestone: - Have access to the machines. - Configure the network to proper functions.

Project:	WP ref: 3
Major constituent: Software	Sheet 3 of 6
Short description: Configure the Mumbai Testnet	Start event: 14/03/2022 End event: 10/06/2022
	Deliverables: Proofs of the proper function
Internal task T1: Set up a node Internal task T2: Do the hello world smart contract Internal task T3: Setup an Amazon KMS signature to valid the transactions. Internal task T4: Do a test transaction	Milestones: - Get the public key of the KMS. - Sign with KMS. - Do a full transaction using KMS.

Project:	WP ref: 4
Major constituent: Software	Sheet 4 of 6
Short description: Configure an API	Start event: 11/04/2022 End event: 20/05/2022
	Deliverables: No
Internal task T1: Connect the API with the blockchain net, with the needed security. Internal task T2: Set up a database to emulate the SEAT MO database. Internal task T3: Make the API adaptative to the most common errors.	Milestones: - Transactions can be made. - We can make a transaction with a smart contract. - The database response to transactions.

Project:	WP ref: 5
Major constituent: Software and testing	Sheet 5 of 6
Short description: Test the system and study scalability in the number of users.	Start event:16/05/2022 End event: 17/06/2022
	Deliverables: The scalability proposal
Internal task T1: Create a frontend to make the petitions. Internal task T2: Test the system and correct errors Internal task T3: Make a scalability proposal for future implementation in the market.	Milestones: - Frontend is easy to use. - The system works properly

Project:	WP ref: 6
Major constituent: Documentation	Sheet 6 of 6
Short description: Make all the documentation for the project	Start event: 14/02/2022 End event: 21/06/2022
	Deliverables: - Work Plan - Critical Review - Final report
Internal task T1: Documentation of all the steps Internal task T2: Deliver the workplan Internal task T3: Deliver the Critical Review Internal task T4: Deliver the final report.	Milestones: - Be up to date with the deliverables

1.4.2 Gantt Diagram

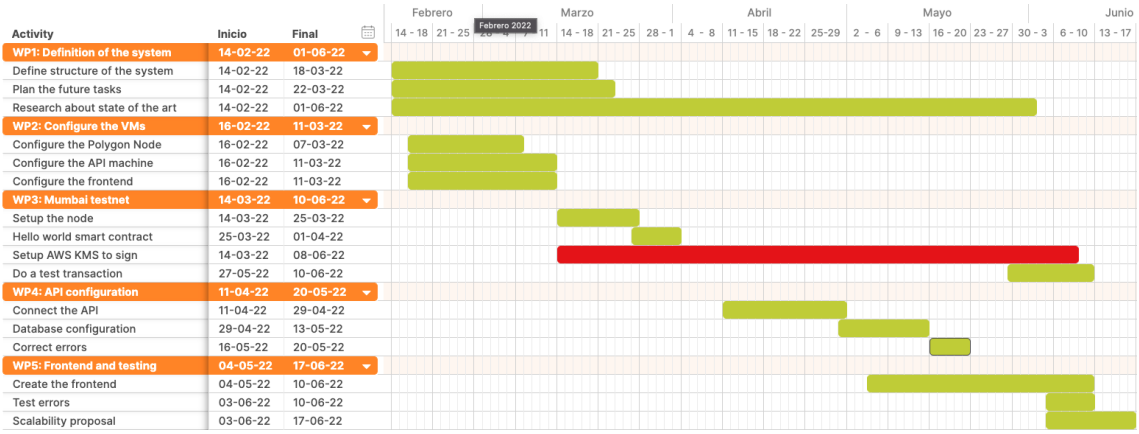


Figure 1: Final Gantt Diagram of the project

1.5 Deviations from the initial plan

One of the goals is to make the dApp as user friendly as possible, and to work with the workflow that SEAT MO has. For that being possible, a creation of a wallet for the user was necessary. That means that the final user didn't have to know anything about blockchain to use the service. Because the libraries for the AWS KMS doesn't actually support the signature of a meta-transaction, and the work around will take several weeks of research and implementation that will delay the thesis, an alternative solution, involving metamask is implemented.
Nevertheless, as we can verify which address belongs to which real person, since we will have the email and phone number of the user, the potential loss of tokens are avoided by transferring the tokens from the old account to the new one.

2 State of the art of the technology used

The purpose of this section is to introduce the reader to all the technologies that this project is going to use, and why are they used instead other alternatives. This section will be used as introduction to the blockchain technology including metatrasnactions, industry standards, the second layers solutions and Smart contracts, adding an explanation of the Amazon Web Services used.

2.1 Blockchain

When it is time to store information the majority of systems are centralized with a database in which the users put trust on it. But what happens if the source cannot be trusted? To solve this trust issue, *Satoshi Nakamoto* propose a decentralized system of transactions, which is basically information. The Bitcoin [17] transaction protocol.
It has properties that makes very easy to the users to put trust in them:

1. **Robustness**
   Blockchain Technology stores information in blocks which are chronologically synchronised. And these blocks cannot be controlled by an individual, making blockchain highly secured and trustable.
2. **Decentralised Ledger**
   Decentralised Ledger operates on peer to peer basis. Because every node has a copy of the blockchain and the longest match rule, makes that the blockchain is owned by everyone so there is nobody to trust.
3. **Immutable**
   Once a block is sealed cryptographically or added to main chain, it is impossible to delete or edit, ensuring the immutability of the digital ledger, making it perfect for storing permanent data such as transactions or other registers of use of any service.
4. **Transparency**
   It is possible for a user to verify and track their transactions, with the entire record of the transactions available in the blockchain in every node. As it is public and immutable, everybody can see what it has been written in the blockchain and by who. Regardless, the user anonymity is ensured by a *nickname* which corresponds to the address of the wallet.

So considering all the benefits that the blockchain provides, it is the proper technology to implement the mobility service that SEAT MO offers. Booking the vehicle, tracking it and the implementation of tokens that can be used as engagement to other products and offers of the company.

2.1.1 The blocks in the blockchain

As intuitive as it sounds, a blockchain is composed by blocks that stores the information. You can think of it as a page of notebook where you can write, but everyone can look at it and nobody can edit anything. The complete notebook will be the blockchain. To link the pages we will use the previous hash of the previous block as a field in the next block

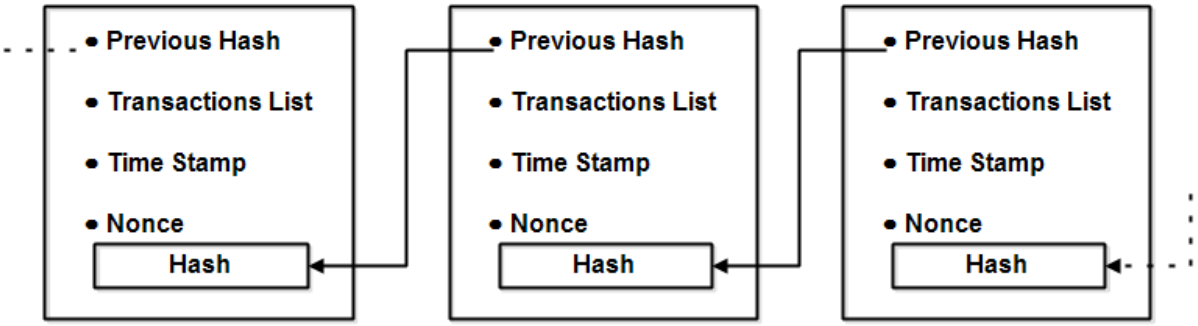


Figure 2: Chain of blocks

2.1.2 Transactions

Writing in a blockchain is what we call a transaction, and has to be differentiated from the concept of transferring a token. We are going to transfer an ERC-20 token, but the transaction will transfer a cryptocurrency, for instance transferring SETK costs some gas as MATIC which is the equivalent to Polygon as Bitcoin is to the Bitcoin blockchain network. The equivalent will be that, when you want to transfer euros to someone, the bank may charge you a fee for the transfer.
Reading the blockchain it is free, since all the content is public but it does not mean that there is no privacy because every user has an address but it is not linked to a physical person. However, writing in the blockchain has costs. As the standard says, this payment of writing in the blockchain is called **gas fee**, where the miner who validates the transaction gets a reward that varies between consensus mechanisms. There are notes about transactions that need to be clarified for this thesis.

Metatransactions The concept of Metatransaction is to put a transaction inside another transaction. This means you can pay the gas for another person transaction. This is essential to the development of a dApp oriented to the general users that do not know how a blockchain works and is reluctant to spend any money in a system that does not trust.

Relayer The function of a relayer in a blockchain is to act as a buffer and wait to validation of the block that contains the transaction. This technology allows more reliability because if a transaction is not validated it can be retried any time in the future, or if a dApp that uses a relayer goes down, it prevents the transaction lost. Working with metatransactions, a relayer can sign the main transaction, allowing gas-less transactions since the relayer would pay for the gas. For this function the relayer may be deployed in a GSN, but for the project the Defender Relayer of Openzeppelin[20] will be used as it funtions as a buffer and signer.

2.1.3 Smart Contracts

Smart Contracts are scripts that are written in the blockchain and have variables, methods and can inherit other classes like other languages. Mapping is an important part of a smart contract, which is how we store information from multiple users and allows to control the actual nonce of an address. Nonce stands for, number once, and helps to prevent replay attacks in a blockchain. A replay attack occurs when a blockchain forks and then transactions can be duplicated and the attacker get benefited twice. The most extended language to code is Solidity.
To get everyone to work in the same basis and make the blockchains to understand each other there are standards that are followed, such the ERC-721[9] also known as NFT. For this project the main standard is the ERC-20.

ERC-20 [24] An ERC-20 is an standardized token that provides an API for calling methods. It has a mapping with the balance of all address that holds the token and defined methods to be called such as transfer or mint. This standard is extended with more functions: the burn method uses the original transfer function to send tokens from the requester address to an empty address, or the pause method that calls all the allowance transfer of the original ERC to stop all the transactions if any change has to be made.

ERC-777 [6] The ERC-777 starts as an intention to upgrade the ERC-20. It integrates the burn and allowance function without the necessity of any modification, and adds to main functions. The send function, that allows to send data in a token transfer and works with operators. This functions add to a mapping the address that are allowed to mint and burn tokens, which is restricted to the owner in the ERC-20.

2.1.4 Consensus mechanisms

When a transaction is submitted to the blockchain, we need a method to make sure that the block is valid and no block has been corrupted. Those are the consensus mechanisms, which need to pass the 51% proof to maintain the blockchain secure by definition.

Proof of Work The PoW[15] is a consensus mechanism that establish a difficulty in the block. The miners have to proof that some computational work has been expended by resolving a cryptographic puzzle. For that, when generating a transaction, a partial hash is sent. Then the miner has to complete this hash in order to validate a block. The main problem of this consensus validation mechanism is the power consumption[13] that can be as high as an entire country, such Switzerland[14]. To valid a corrupted block, someone has to have the 51% of computational power of all the blockchain, that makes the consensus secure by definition.

Proof of Stake The PoS is a consensus mechanism to keep a decentralized system, in this case the blockchain, secure. It was a created as an alternative for the PoW consensus mechanism that consumes a lot of computational power and, therefore a lot of electricity. This method uses the cryptocurrency (as Ethereum) as collateral to validate the block. Everyone that puts some cryptocurrency in a deposit becomes a validator. Then blocks

are validated by the validators, known as miners in PoW. There are many consensus on how to choose the validator. One is to choose randomly between the validators. Another one is to let all the validators vote if the block is valid. If a percentage votes that a block is valid then it is incorporated to the blockchain. By running this validation process, the network gives rewards as an incentive to become a validator which may vary between consensus. The gas fees that the block generates, and other rewards are examples on how to engage the participants to stack his cryptocurrency and become validators. To valid a corrupted block, someone has to stack the 51% of the stacked cryptocurrency of all the blockchain, that makes the consensus secure by definition.

2.1.5 Wallets

A Wallet is the software that allows to interact with the cryptographic assets that a person owns. One wallet has multiple accounts that are identified by an Address and can interact with multiple networks. This account is created with a asymmetrical key pair. Ethereum addresses are hexadecimal numbers, identifiers derived from the last 20 bytes of the Keccak-256 (explained in 2.1.6) hash of the public key. There are several wallets. Arkane is one of the most popular and works as an API and a web version to interact with. However, the easiest one for a user to interact is Metamask because it works as an expansion in any browser and allows an easy interaction with the different blockchain networks and is very user friendly to make transactions and view the assets.

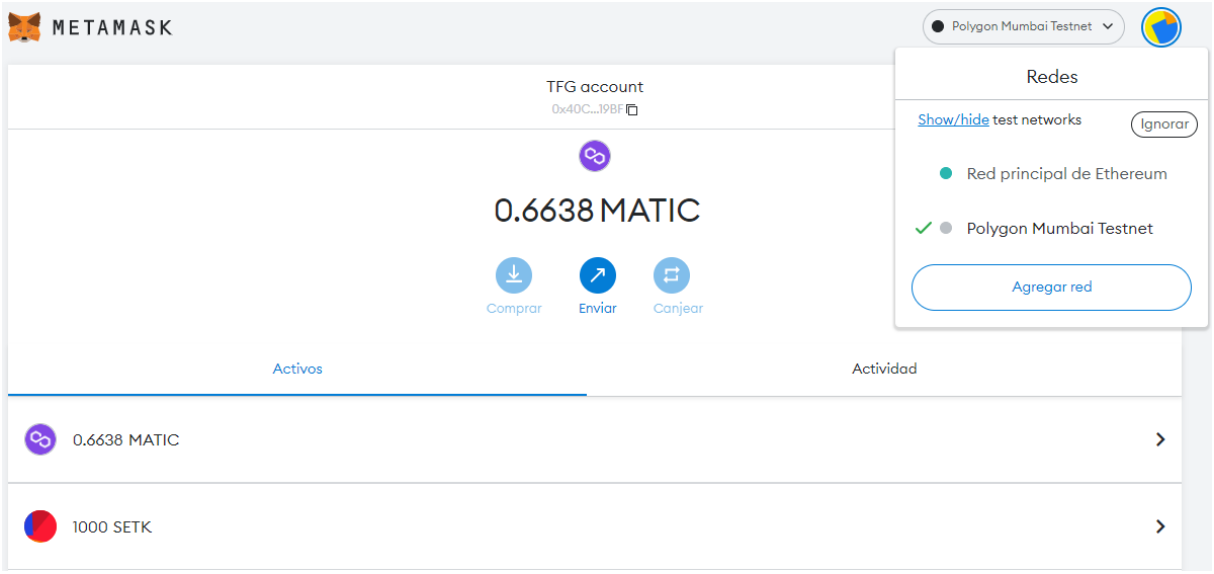


Figure 3: Metamask wallet showing the assets of SEAT

Metamask allows the user to enter his wallet with a password, but a mnemonic (a combination of twelve words) is provided at the creation in case it is necessary to recover the account. In a wallet is vital to have the private keys of the different accounts, and the mnemonic stored in a secure place to prevent the robbery of information.

2.1.6 ECDSA and address recovering

When working with transactions we have to differentiate between hashing and signing a transaction. Hashing provides a method to verify the input integrity, and it is used to link the blocks. By providing the hash of the previous block (see Figure 2), you can verify that this block is valid, so if any change is done in the blockchain, the hash of the block will change and will not correspond the hash in the next block. Ethereum and Polygon uses the Keccak-256 hashing algorithm with corresponds to the standardized SHA-3[18].With that logic we can secure by definition that a blockchain is immutable. When a block is valid you have to make sure that the transaction inside the block is valid. This is the signature, that consist in a key pair where some data is encrypted but only the person that encrypted can decrypted and it is used to verify the sender. To sum up, hashing is for checking that data is valid, and signing is a the course of action to check where the data comes.
The algorithm for signing is ECDSA[1], in specific the secp256k1[23].
The curve is defined as $y2modp = (x3+7)mod(p)$ where **p** is $p = 2256–232–29–28–27–26–24–1$

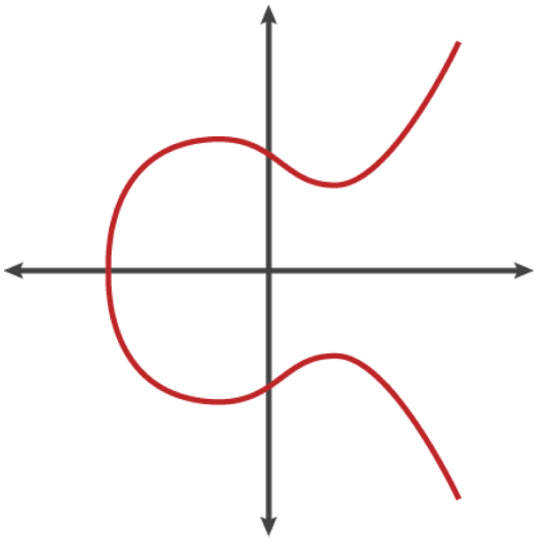


Figure 4: Visualization of an ECDSA

The process of signing with the ECDSA secp256k1 curve are the following:

1. Hashing the transaction.
2. Generate a random number k between [1, n - 1] (n is the order of the curve).
3. Calculate a random point in the curve $K = k * G$ being G(x,y) the generator base point. Then r is defined as the x component of the K point.
4. Calculate the signature proof as: $s = k(^{-}1) * (h + r * privateKey)(modn)$

Now that we have the r and s, we have the signature, but we have to recover the address to verify that is the correct user that signed the transaction. We can recover the address by finding one of the two possible values of v. v is the recovery id and it can be one of

two possible values: 27 or 28. With one of this values the recovered message will be the original message and we will have recovered the address and the recovery id.

2.2 Polygon

Because writing in a blockchain has costs, the **gas fees** might make the dApp non economical viable. For solving this problem, a second layer solution is used. Polygon is a second layer blockchain solution that provides hybrid Proof-of-Stake and Plasma-enabled sidechains and it is fully integrated with Ethereum, that operates as the main chain. It has public Heimdall nodes that works between the Ethereum main chain and only publishes the **hash root** of a Merkle Tree, making that the blocks with the transactions are in a side chain, which is more affordable, but the proof that all the blocks are valid are in the Ethereum blockchain so the gas fees are heavily reduced.

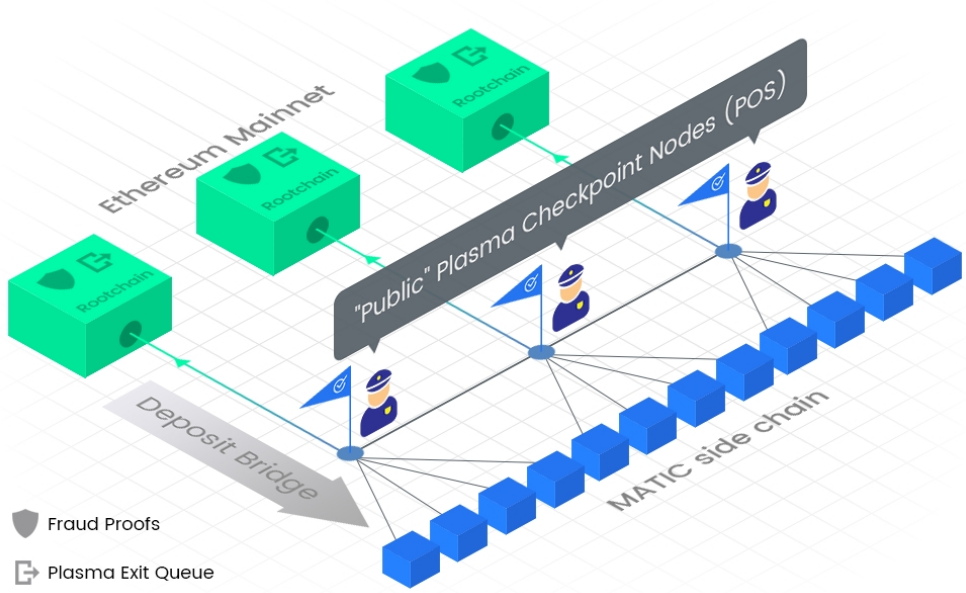


Figure 5: Polygon architecture

2.2.1 Heimdall

Heimdall works as the governance layer between the Polygon and Ethereum net, authenticates with the net using Proof-Of-Stake, controls the block production and the state-sync mechanism. It uses **Pulp** to verify the transactions based on the RLP[10] encoding. RLP is the actual standard for encode binary data, and it is used to encode certain types of data, such strings, integers or floats among others, treating them as unique objects to encode.
The bridge works as the following. It validates all the blocks provided by Bor and creates a Merkle Tree of the block hashes and then writes the root of the tree in the Ethereum chain, saving to the user a lot of gas fees in transactions.

Merkle Tree The Merkle Tree contains the hashes of all the data blocks produced by the Bor layer. Then we reduce it all to one hash, the root hash. With only this hash, we can have a proof that all the blockchain is valid and by only doing one transaction to the main chain, Ethereum.

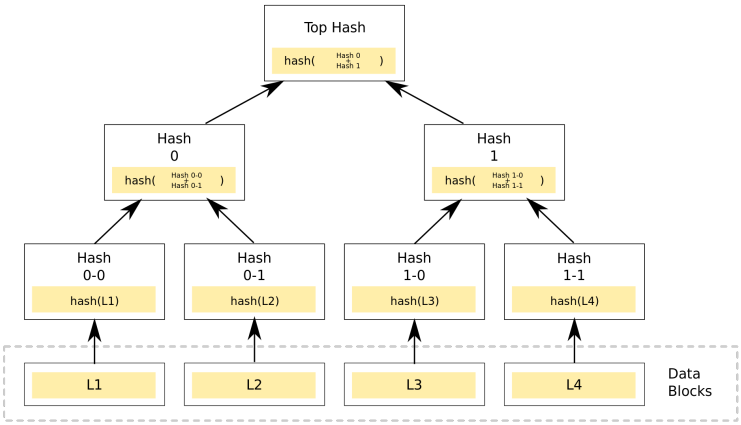


Figure 6: Structure of the Merkle Tree

2.2.2 Bor

Bor is the Polygon's block productor. It assembles and generates all the incoming transactions into blocks. As seen in the architecture figure [7] it contains an EVM that contains a side chain with the blocks. The hashes of this blocks are the ones that are going to be pulled to the Heimdall layer.

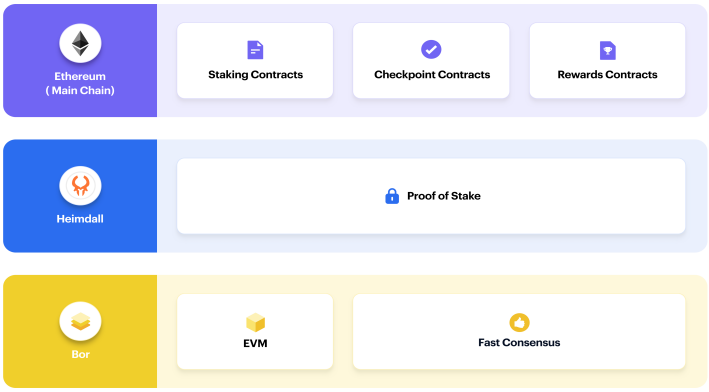


Figure 7: Architecture of the deployed node

2.3 Amazon Web Services

When developping the dApp and simulating and environment, Amazon Web Services offers a wide variety of functionalities and it is the cloud solution selected because the company is currently developing projects using this services and they can be reliable as they use high-availability across all services. Some examples are: **Cognito** to user credentials management, **Pinpoint** as a notification of event to user via email or sms or **Aurora** as a high-availability DB.

2.3.1 Lambda

The **Lambda** service lets you execute code inside a VPC. This service interacts with other services that are in the same account and region and provides the logs of the functions executed. It has a similar function as a Docker container, it only has the interpreter and the libraries installed and It does not contain or virtualize any operating system.

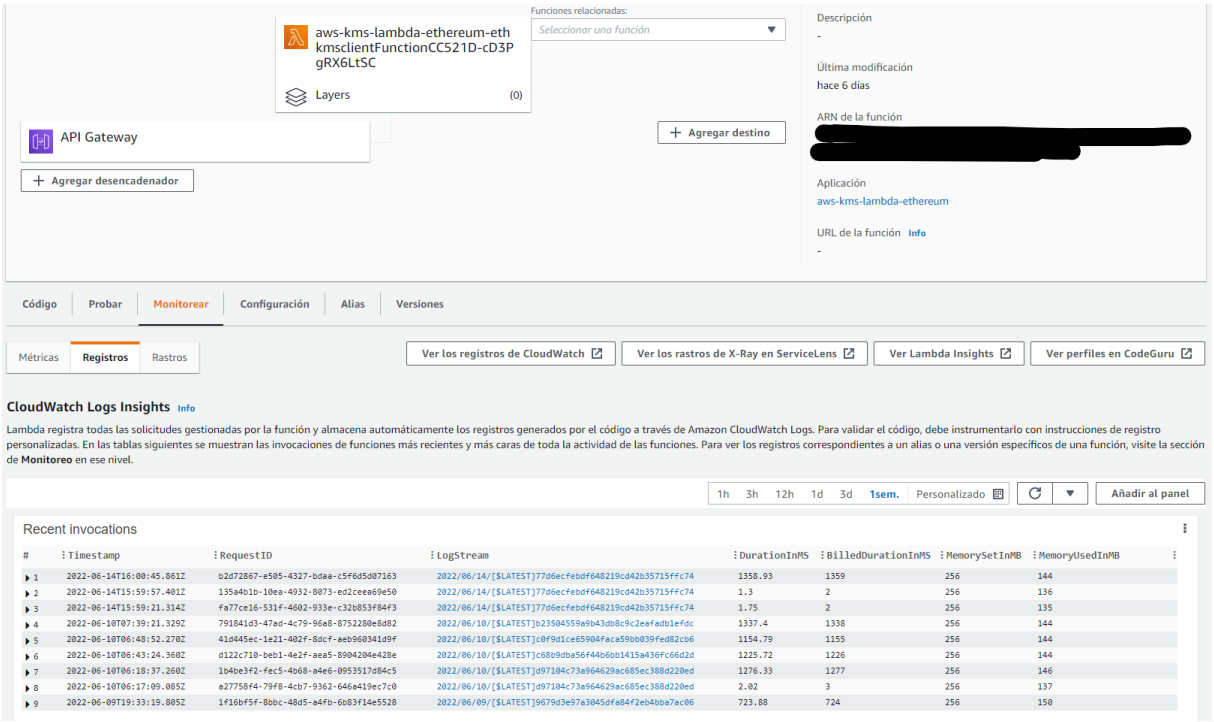


Figure 8: Amazon Lambda with logs of previous executions

The lambda function is connected to an API Gateway, this service is the one that when receives a call, that can be configured as a POST, GET, DELETE, PATCH, HEAD PUT or ANY call to handle every method in the endpoint. In the case it can be linked to a other Amazon Services such as Aurora to interact with the DB, or to Lambda, to execute the function that has to read an HTTPS method.

2.3.2 EC2

EC2 stands for Elastic Compute Cloud, and it is the server renting service that Amazon

offers. By paying for CPU usage, you can deploy a VM of any kind of operating system with a different types of instances. The larger the instance, the more that cost per CPU usage. For the project, the node is in a t3.xlarge instance with 4 virtual virtual CPU and 16GB of RAM and the front-end and backend in a t2.micro instance with 1 vritual CPU and 1GB of RAM. Other characteristics that change between instances types are the speed of write/lecture of the storage and the maximum network performance. This service is used because it provides security group management to accept specific IPs, and they have internal communication to emulate connection of two servers in the same network and offers a public DNS to host the front-end.

2.3.3 AWS KMS

The Key Management System allows to create a symmetric or an asymmetric key pair to use it to cipher or sign a transaction depending on the necessity. The private key is stored and cannot be copied, retrieved or checked in any way, that is why it is secures and decentralizes the dApp even more because SEAT will not hold any key, only a reference to the KMS ID which can only be called by the user. Also, if the user loses his credentials, they can be restored because the key is secure and external to the system.

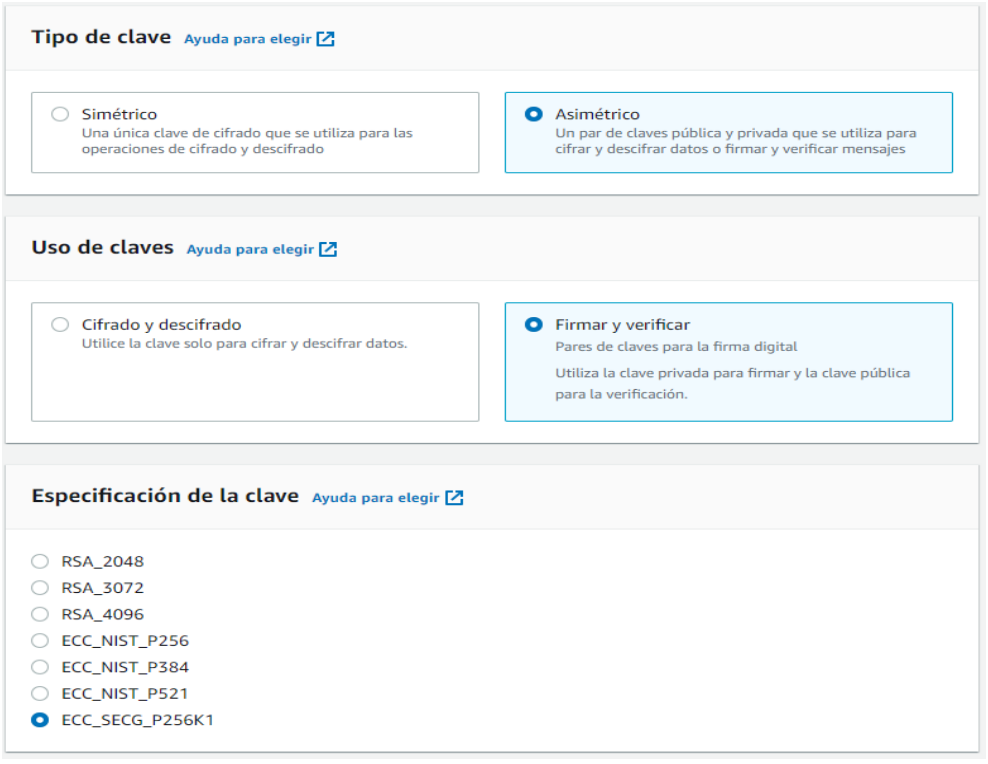


Figure 9: Creation Key option in AWS KMS

As we can specify a *ECC_SECG_P256K1* key type, it will be compatible with the standard in the Ethereum network transactions.

3 Development

This section will go through the development process followed and the structure of the project. The project has 3 instances. The Front, that will contain a react code to view the dApp in a web environment emulating the basic functions of the SEAT MO application and Web3 to interact with contracts, the API with NodeJS, HardHat and Ganache that will manage all the blockchain interaction and deployment of the contracts, and a Polygon Node to be part of the blockchain. The signature is going to be using Metamask, but an alternative with AWS KMS is developed and proposed but not implemented.

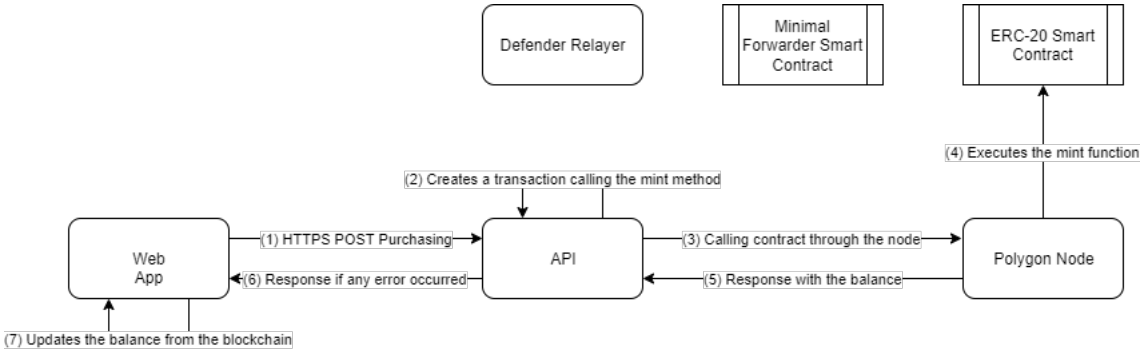


Figure 10: Purchasing block diagram

When purchasing tokens, the transaction is signed in the node and calls directly the mint method in the SETK contract, so the gas fee is payed by the SEAT account.

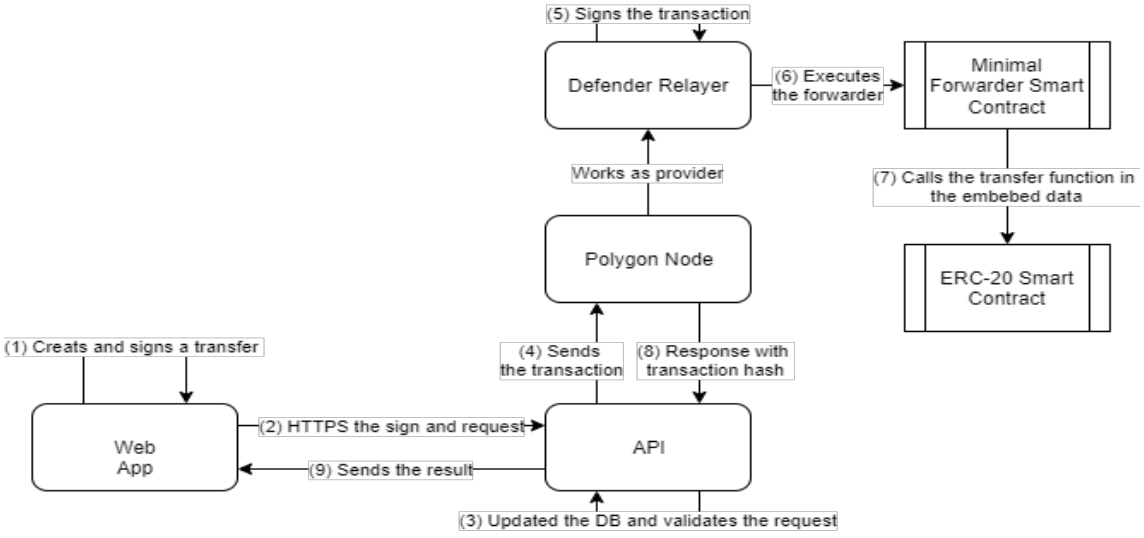


Figure 11: Booking and releasing block diagram

When creating a metatransaction, the gas is now payed by the relayer who signs the transaction that calls the Forwarder contract, the cryptocurrency of the relayer has also to be maintained by SEAT.

3.1 Deployment of a Full Node in the mumbai testnet

In order to get access to the MBTN network we are going to deploy a node in it. This node will be our gateway to the net, to validate transactions and to push blocks. The solution uses two main software components, *Heimdall and Bor*.

3.1.1 Configuring the node files

For the node deployment [22], the software used is Ansible Tower, an automation tool that allows to perform commands in a remote machine. The destination machine is a EC2 **tx3.large** instance with 16GB of ram, 8 core CPU and 750 GB of storage. As a pre-requisite, it is important to notice that the playbook is implemented in a Debian machine, because it uses the package manager *apt* to obtain the packages. This required a redeploying of the instance node, because all the instances were initially RHEL that uses *dnf* package manager and the ansible playbook will not be executed correctly.
From the Ansible Tower that SEAT uses for the automation tasks, we clone the repository and edit the inventory.yml file putting our public node IP address. In the config file we add the seeds for the Heimdall service to fetch the data and a public RPC provided by Polygon to sync the state of the blocks of the current blockchain network. Finally, in the start file we add the bootnodes to indicate where the bor service will sync his EVM, consensus and block structure.

Ansible playbooks A playbook is a list of tasks that uses commands that invokes roles to install all the necessary elements, deploy all the services and starts them. Here is an example of the main playbook to execute, that calls other roles that are the ones that executes the tasks with its own variables.

Listing 1: Main playbook of the node deployment.

```
- name: Network node management
  hosts: all
  tasks:
    - name: Install dependencies
      include_role:
        name: roles/install-dependencies
        apply:
          tags: always
      tags:
        - install_dependencies
    - name: Build Heimdall
      include_role:
        name: roles/install-heimdall
        apply:
          tags: always
      tags:
        - build
    - name: Build Bor
      include_role:
        name: roles/install-bor
        apply:
          tags: always
      tags:
```

```
        - build
    - name: Setup and deploy network
      include_role:
        name: roles/setup-network
        apply:
          tags: always
      tags:
        - deploy
```

Now we have to wait to make the node fully synced with the MBTN and we can check it using *curl localhost:26657/status* command, which returns:

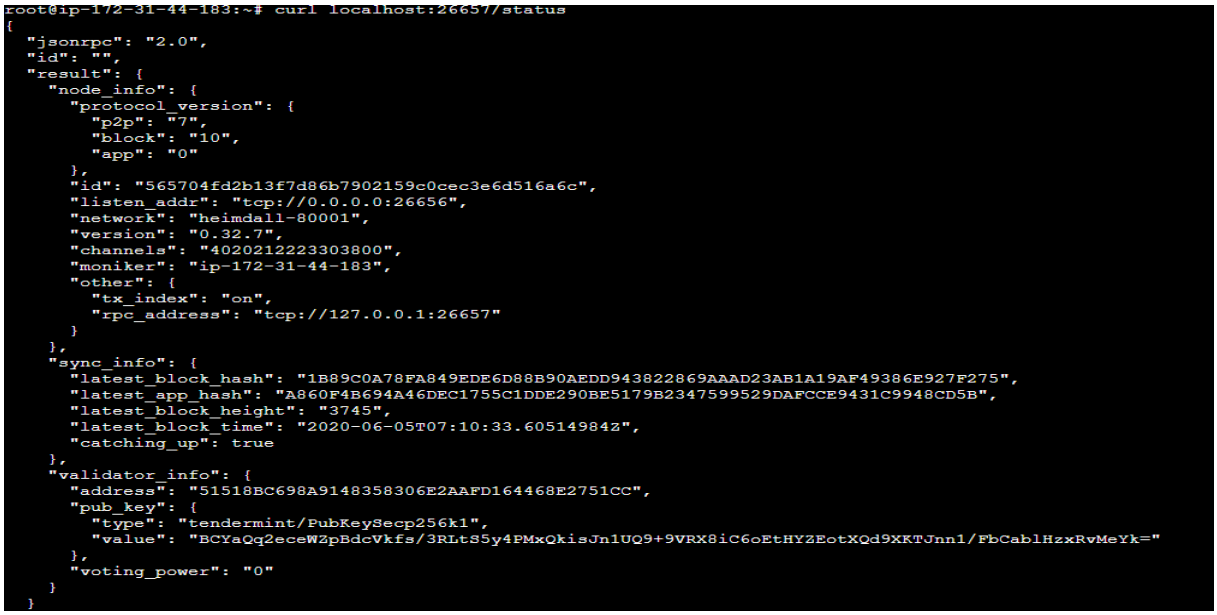


Figure 12: Node still catching up

Meanwhile the node is synchronizing with the blockchain we cannot vote to validate any block. After 3 days the node is finally catched up with the MBTN blockchain, and we can check it by rerunning the previous command.

Listing 2: Sync status of the Polygon Node

```
"sync_info": {
  "latest_block_hash": "FA9D2BDEA84853595D0DC281E26685A41CB8A1DCE489CB5C
     9C5CC56B10A9CAC5",
  "latest_app_hash": "90716D6243A75873BDC9C7A7D1C48915DC254A3D2CE10255F5
     4BD214DCFC0FCA",
  "latest_block_height": "11370598",
  "latest_block_time": "2022-06-14T09:21:59.034846299Z",
  "catching_up": false
}
```

The "catching_up" variable is set up to False. That means that our node is fully sync with the chain and all the blocks take 358GB of storage of the machine, which gradually augments when new blocks are validated.

3.2 Registering and Signing with Key Management System of Amazon Web Services

This implementation is based in an easy boarding paper[8], but could not be done due the lack of time and research for adapting this solution to the metatransaction necessity and make the validation with the Relayer.
Amazon Web Services offers a very wide variety of services, but we are going to use three of them to implement the registration and signing of a transaction[7]: the Lambda service to execute the code, the AWS KMS to sign and create new Ethereum address and API gateway to manage the HTTPS petitions. The signature is going to be made following the ECDSA with the keccak-256 curve.

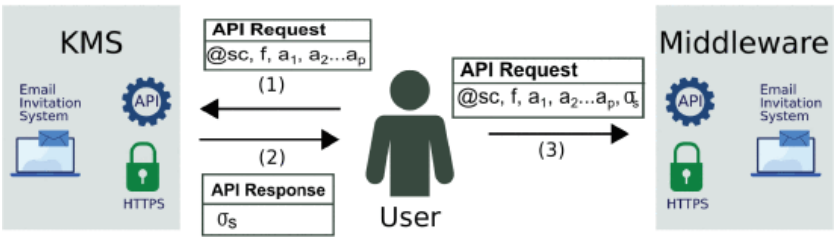


Figure 13: Structure of the service with KMS. Source [8]

The user will call an API Gateway and recover the signature. Then it is passed to the back-end to verify the request and process the transaction. This system it is not finally implemented due to the complexity and the time limitation but a work around will be developed and implemented in the final application.

3.2.1 Setting up the environment

We are going to use the CDK library that amazon provides to interact with the services. Firstly we execute a *cdk bootstrap* with the credentials of the account to link it, and create a App.py to deploy the lambda service.
It creats a lambda instance with the following permissions:

Listing 3: Set up of the *cdk deploy* command to deploy the scripts.

```
cmk = aws_kms.Key(self, "eth-cmk-identity",
                        removal_policy=core.RemovalPolicy.DESTROY)
cfn_cmk = cmk.node.default_child
cfn_cmk.key_spec = 'ECC_SECG_P256K1'
cfn_cmk.key_usage = 'SIGN_VERIFY'

eth_client = EthLambda(self, "eth-kms-client",
                dir="aws_kms_lambda/_lambda/functions/eth_client",
                env={"LOG_LEVEL": "DEBUG","KMS_KEY_ID": cmk.key_id,"
                    ETH_NETWORK":eth_network})
cmk.grant(eth_client.lf, 'kms:GetPublicKey')
cmk.grant(eth_client.lf, 'kms:Sign')
cmk.grant(eth_client.lf, 'kms:CreateKey')
cmk.grant(eth_client.lf, 'kms:ScheduleKeyDeletion')
```

We grant the permissions to the lambda code to Create a Key, sign with the key, retrive the public key and delete one key if a user leaves the dApp. Once this is done we can deploy the code with *cdk deploy* with the requirements.txt file included.

3.2.2 Programming the workflow

Now we are going to create a file that reads the income function, and another one that will handle the functions depending on what we need. We expect to receive an HTTPS POST so an operation field is required. The first function is to calculate the Ethereum Address:

Listing 4: Status function in the lambda service.

```
if operation == 'status':
    if not (event.get('kms_id')):
        return {'operation': 'status',
                'error': 'Not_kms_key_provided'}
    key_id = event.get('kms_id')
    pub_key = get_kms_public_key(key_id)
    eth_checksum_address = calc_eth_address(pub_key)

    return {'eth_checksum_address': eth_checksum_address, 'KMS_KEY_ID':
        key_id}
```

This operation calculates the Ethereum Address with the public key. The public key is a native AWS KMS function so we only need to calculate the Wallet Address.

Listing 5: Funtion that returns the Ethereum Address

```
def calc_eth_address(pub_key) -> str:
    key = asn1tools.compile_string(SUBJECT_ASN)
    key_decoded = key.decode('SubjectPublicKeyInfo', pub_key)
    pub_key_raw = key_decoded['subjectPublicKey'][0]
    pub_key = pub_key_raw[1:len(pub_key_raw)]
    hex_address = w3.keccak(bytes(pub_key)).hex()
    eth_address = '0x{}'.format(hex_address[-40:])
    eth_checksum_addr = w3.toChecksumAddress(eth_address)

    return eth_checksum_addr
```

We use the library asn1tools. The SUBJECT_ASN [5] is a asn1 Schema that describes the type of the algorithm. Then we decode the public key and with the keccak function, that is the signing algorithm, and obtain the hex address. Now the only thing we have to do is format the in a valid Ethereum Address using the web3 library.
For registering the user in the system we should have to create a new AWS KMS key, we do this by using a boto client, and the kms createKey function.

Listing 6: Function that the registers the user in AWS KMS

```
def register(user: str) -> dict:
    client = boto3.client("kms")

    response = client.create_key(
        Description= user,
```

```
6           KeyUsage='SIGN_VERIFY',
7           CustomerMasterKeySpec='ECC_SECG_P256K1',
8           Origin='AWS_KMS',
9           BypassPolicyLockoutSafetyCheck=False,
10          MultiRegion=False)
11
12      return response
```

It is important to notice that we use the ECC_SECG_P256K1, which is the signing algorithm that follows the ECDSA, to create the key, because it is the standard used in the Ethereum Network and therefore in the Polygon Network.
In order to sign a transaction it is mandatory to provide the KMS ID of the user and all the data fields necessary to make a transaction. When lambda receives the call, creates a python dictionary with the following fields: nonce, to, value, data, gas (which corresponds to the gas limit) and gasPrice. Once it is assembled it is passed to a function that signs it:

Listing 7: Main steps for signing a transaction with AWS KMS

```
1  def assemble_tx(tx_params: dict, params: EthKmsParams, eth_checksum_addr:
       str) -> bytes:
2      tx_unsigned = serializable_unsigned_transaction_from_dict(
           transaction_dict=tx_params)
3      tx_hash = tx_unsigned.hash()
4
5      tx_sig = find_eth_signature(params=params,
6                                  plaintext=tx_hash)
7
8      tx_eth_recovered_pub_addr = get_recovery_id(msg_hash=tx_hash,
9                                                  r=tx_sig['r'],
10                                                 s=tx_sig['s'],
11                                                 eth_checksum_addr=
                                                       eth_checksum_addr)
12
13     tx_encoded = encode_transaction(unsigned_transaction=tx_unsigned,
14                                     vrs=(tx_eth_recovered_pub_addr['v'],
                                            tx_sig['r'], tx_sig['s']))
15
16     return w3.toHex(tx_encoded)
```

Firstly it transforms the dictionary into a serializable transaction to the ethereum library to understand. Here is where the main problem became. This library is not yet prepared to handle metatransactions that requires the "from" field of the original sender. After hashing the transaction we find the signature, and obtain the **r** and **s** of the signature using the following:

Listing 8: Function that finds the r and s parameter of a transaction.

```
1  def find_eth_signature(params: EthKmsParams, plaintext: bytes) -> dict:
2      signature_schema = asn1tools.compile_string(SIGNATURE_ASN)
3
4      signature = sign_kms(params.get_kms_key_id(), plaintext)
5
6      signature_decoded = signature_schema.decode('Ecdsa-Sig-Value',
           signature['Signature'])
```

```
7       s = signature_decoded['s']
8       r = signature_decoded['r']
9
10      secp256_k1_n_half = SECP256_K1_N / 2
11
12      if s > secp256_k1_n_half:
13          s = SECP256_K1_N - s
14
15      return {'r': r, 's': s}
```

The sign_kms it is done in the register function, but instead of **create_key** it calls the function **sign**. When we obtain the s, according to the RFC, if it is superior than the half of the elliptic curve, we have to invert it. Once we have the r and s we can check if we can recover the original ethereum address.

Listing 9: Function that recovers the address and the v parameter.

```
1 def get_recovery_id(msg_hash, r, s, eth_checksum_addr) -> dict:
2 for v in [27, 28]:
3     recovered_addr = Account.recoverHash(message_hash=msg_hash,
4                                          vrs=(v, r, s))
5     if recovered_addr == eth_checksum_addr:
6         return {'recovered_addr': recovered_addr, 'v': v}
7 return {}
```

By using the recoveredHash function we can check if **v** corresponds to one of the two possible **[27,28]** and return the v and the recovered address if it is the same as the one that calls the function. The final step is to encode the transaction with all the parameters and return it as a raw hex transaction.

3.2.3 Calls to the API Gateway

We can know create a key pair by a call to the API GATEWAY, that is previously configured in the lambda service.

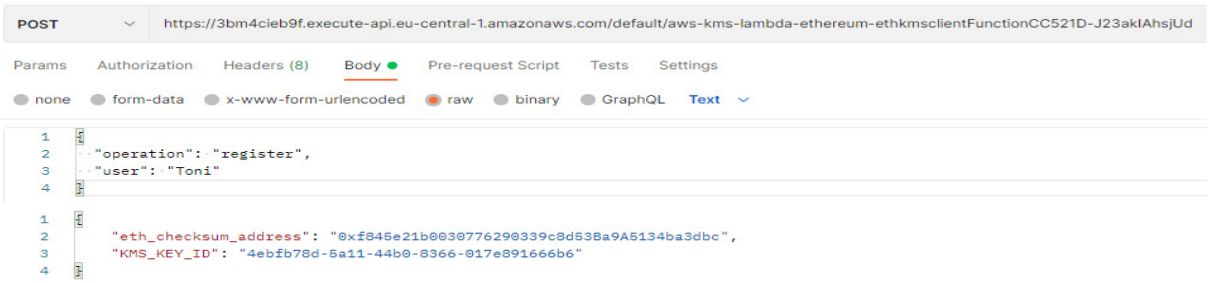


Figure 14: Calling the register function

And the key is properly created in the AWS KMS and can be seen in the web console. Also a delete function is implemented, that will delete the selected key in 7 days to allow the user to cancel the deletion if he wants.

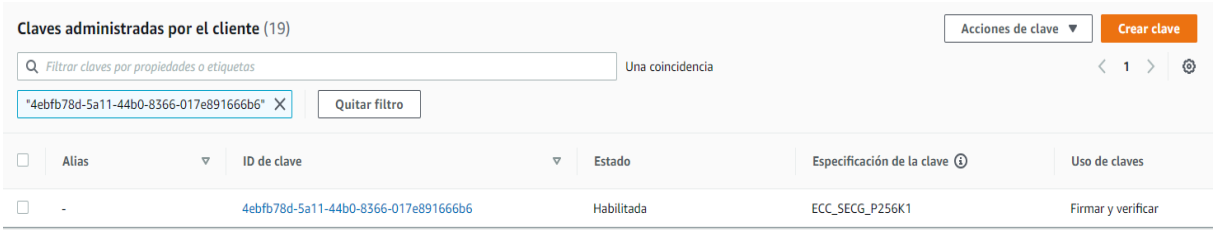


Figure 15: Key in the AWS KMS

Now if we make a call with the operation status and the AWS KMS ID it will calculate and return our Ethereum Address, and finally we can sign a transaction.

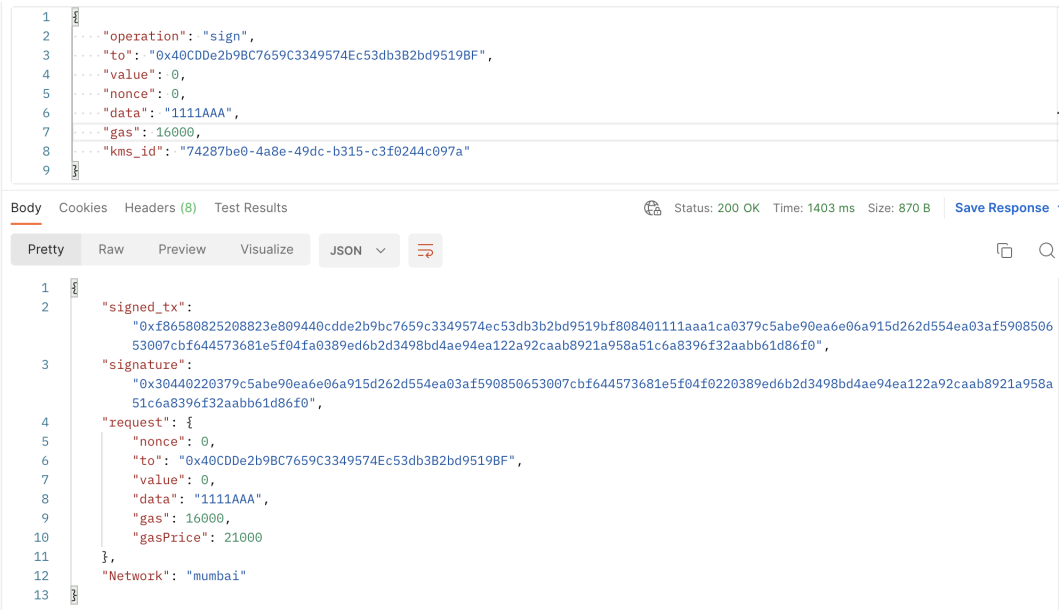


Figure 16: Calling the sign function

The **signed_tx** field is the raw transaction, the signature is the field that should be checked to recover the address, and the request is the transaction processed by the AWS KMS service, that does not match the sent request because some fields are included in the function when creating the complete transaction inside the lambda code.

3.3 Writing and deploying the contracts

The contracts are written in Solidity and have a forwarder that interacts with the contract. The provider of the forwarder it is Defender Relay by Openzeppelin [20].

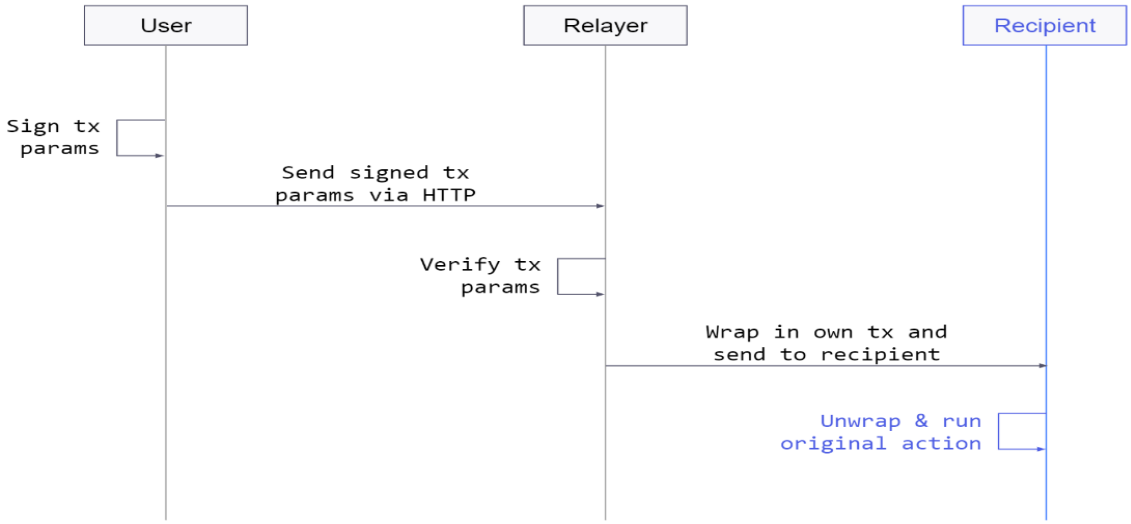


Figure 17: Structure of sending a metatransaction. Source Openzeppelin[21].

The user will wrap and sign the metatransaction and send it to the relayer. The signed transaction will be a signed Type Structured Transaction (EIP-712) and will be signed by the relayer and transmitted to the Forwarder Contract. Finally, the forwarder contract will execute the ERC-20 that contains the SETK.

3.3.1 The Minimal Forwarder Contract

The minimal forwarder follows the structure of the EIP-712[2] a signed typed data, with the from field that is the original sender. With this structure we have three main functions, getting the actual nonce to prevent a replay attack, the verify function to recover the original address and the execute function to use the relayer to sign and send the main transaction.

Listing 10: Verify function in the fowarder contract

```
1 function verify(ForwardRequest calldata req, bytes calldata signature)
      public view returns (bool) {
2     address signer = _hashTypedDataV4(keccak256(abi.encode(_TYPEHASH, req.
          from, req.to, req.value, req.gas, req.nonce, keccak256(req.data)))).
          recover(signature);
3     return _nonces[req.from] == req.nonce && signer == req.from;
4     }
```

The verify function uses the signature to recover the sender address. It uses the same hashing algorithm *keccak256* that is used to hash the transaction, so by passing the encoded abi of the transaction we can recover the address. If it is the same that the incoming address and the nonce also coincide, we return *true* as the transaction is verified. The other function of the contract is to execute the transaction:

Listing 11: Execute function to send the request to the Token Contract.

```
1 function execute(ForwardRequest calldata req, bytes calldata signature)
      public payable
```

```
    returns (bool, bytes memory)
    {
        require(verify(req, signature), "MinimalForwarder: signature does
            not match request");
        _nonces[req.from] = req.nonce + 1;
        (bool success, bytes memory returndata) = req.to.call{gas: req.gas,
             value: req.value}(abi.encodePacked(req.data, req.from));
        // Validate that the relayer has enough gas for the call, if not
            send invalid
        if (gasleft() <= req.gas / 63) {
            assembly {
                invalid()
            }
        }

        return (success, returndata);
    }
```

In the function we require that the function verifies the transaction, we increment the nonce to get a valid transaction and after that we call the relayer to process the transaction, but we invalid it if the relayer do not has enough gas left to pay for the transaction.

3.3.2 Token contract

The token contract is a burnable ERC-20 that requires a forwarder to allow metatransactions. We will also allow to pause all the transactions if any change has to be made in the system.

Listing 12: Token contract with the basic functions.

```
contract Seatoken is ERC2771Context, ERC20, ERC20Burnable, Pausable,
    Ownable {
    constructor(MinimalForwarder forwarder) ERC2771Context(address(
        forwarder)) ERC20("Seatoken", "SETK") {}
    function mint(address to, uint256 amount) public onlyOwner {_mint(to,
        amount);}
    function _msgSender() internal view override(Context, ERC2771Context)
        returns(address) {return ERC2771Context._msgSender();}
    function _msgData() internal view override(Context, ERC2771Context)
        returns(bytes memory)
    {return ERC2771Context._msgData();}
}
```

The contract inherits the ERC2771Context that is the recipient where the contract will accept metatransactions. This ERC2771Context allow to get the original caller from the transaction. Normally the sender is called by *msg.sender*, but if we use this parameter the address will belong to the forwarder and not to the user. Then we will use the *_msgSender()* that returns the original sender when it is called.

Listing 13: ERC2711Context contract retriving the original sender.

```
function isTrustedForwarder(address forwarder) public view virtual returns
    (bool) {
        return forwarder == _trustedForwarder;
```

```
    }
function _msgSender() internal view virtual override returns (address
    sender) {
    if (isTrustedForwarder(msg.sender)) {
        assembly {sender := shr(96, calldataload(sub(calldatasize(), 20)))}
    } else {return super._msgSender();}
}
```

When we created the ERC2771Context we passed the address of the forwarder as the only trusted forwarder, so when we retrieve the original sender, if the forwarder is trusted we assembly the original sender and return it. This function has to be overwritten since the ERC-20 base contract has a function that has the same nomenclature, so our SETK contract cannot compile.
The complete contracts can be found in Appendix D

3.3.3 Compiling and deploying

For the deploying we are going to use a simple script in hardhat. The hardhat suite needs a configuration file. All this is going to be done in a local machine, because Ganache with an interface and not in the command line, is more user friendly and easier to document.

Listing 14: Hardhat config file

```
require('dotenv').config();
require("@nomiclabs/hardhat-waffle");
require("@nomiclabs/hardhat-ethers");
/**
 * @type import('hardhat/config').HardhatUserConfig
 */
module.exports = {
  solidity: "0.8.13",
  networks: {
    local: {
      url: 'http://localhost:8545'
    },
    mumbai: {
      url: 'https://matic-mumbai.chainstacklabs.com',
      accounts: [process.env.PRIVATE_KEY],
    }
  }
};
```

As the mumbai url we use an RPC public endpoint to deploy the contract. Later, when we interact with the contract we are going to use the Polygon Node that we deployed. But first we are going to start a Ganache chain that uses the RPC endpoint as the localhost in the port 8545 and uses the first account in the list so it does not require an account field.
First we compile the contracts by running *hardhat compile*. This command creates the artifacts folder. The artifacts contains the name of the contract, the ABI that is a description of the functions and fields needed by the contract, a bytecode of the undeployed contract and a deployedBytecode referencing the deployed contract. You can see a the

artifact of the SETK in Appendix E For the deploying we are going to use a very simple script that runs the deployment and store the addresses in a JSON file.

Listing 15: Script that deploy the contract in a chain.

```
const { ethers } = require('hardhat');
const { writeFileSync } = require('fs');
async function deploy(name, ...params) {
  const Contract = await ethers.getContractFactory(name);
  return await Contract.deploy(...params).then(f => f.deployed());
}
async function main() {
  const forwarder = await deploy('MinimalForwarder');
  const token = await deploy("Seatoken", forwarder.address);
  writeFileSync('deploy.json', JSON.stringify({
    MinimalForwarder: forwarder.address,
    Seatoken: token.address,
  }, null, 2));
  console.log(`MinimalForwarder: ${forwarder.address}\nRegistry: ${token.
      address}`);
}
```

By running the script we obtain the contracts address as an output:

```
adrian@MacBook-Pro-de-Adrian hardhat_polygon % hardhat run --network local scripts/deploy.js
MinimalForwarder: 0xec8Ba9319D4700826E89346cCC710B4294a93be2
Token: 0x7B038800d36f2a7bD6EBB4C53a4ce7495b95be00
```

Figure 18: Result of the deploy script

Now in the block explorer the transaction must appear. Since is a local blockchain, the transactions are validated immediately.

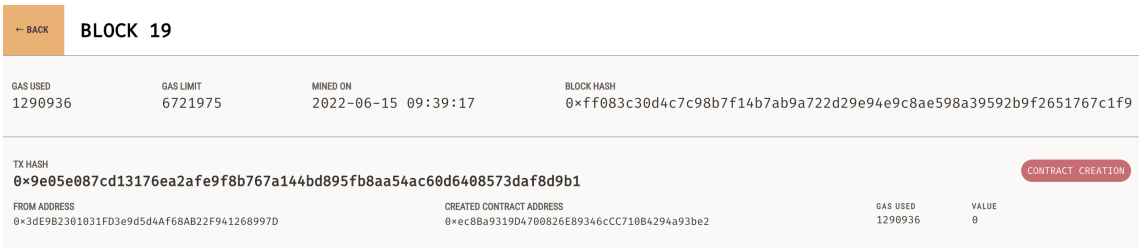


Figure 19: Block of the creation of the forwarder contract

Figure 20: Block of the creation of the token contract

Now that in the local chain is compiled and deployed we can replicate the process and save the contract addresses in the API instance. In the MBTN the contracts are going to be deployed with the SEAT account using the private key. This will make SEAT the owner of the contracts.

3.4 Developing the API

This API is who will handle all the logic and operations and it is written in NodeJS and uses HTTPS as a secure protocol. It also contains a MongoDB to store the users, vehicles and the logs of using it, and uses the libraries of ethers and web3 to interact with the contracts.

3.4.1 Database

We are going to emulate the SEAT MO database storing the vehicles, users and logs. For that we declare the moongose model with the type of information we are going to store. For instance here is the user information.

Listing 16: Model of the user fields in the DB

```
const mongoose = require("mongoose");
const User = mongoose.model("User",new mongoose.Schema({
    username: String,
    email: String,
    password: String,
    address: String,
    vehicle: String,
    lastPicked: String,
    lastReleased: String,
  })
);
module.exports = User;
```

For the vehicles, we will store the plate, the last user that picked up, if it is currently booked and when it was last reserved. For the logs, every time someone reserves a vehicle we create a new instance with the plate of the vehicle, the user and both picked and released times.

3.4.2 Authentication

Now that we know what are we going to store it is time to handle the sign up and the login process.

Signing up First, when a request comes in and the headers and CORS are checked, it is decided where to go. This is done in the route file. You can see all the routing in the Appendix C.1

Listing 17: Post function to the sign up API endpoint

```
app.post("/api/auth/signup",[verifySignUp.checkDuplicateUsernameOrEmail],
    controller.signup);
```

Here we accept the post method and called a function that checks if there is a duplicated user, email or wallet address already in the database. If not, we can sign up the user by adding it to the database, with the actual vehicle, lastPicked and lastReleased empty. You can see the code in Appendix C.2.

Signing in When logging in the user and password is provided via secure communication, and the user is found in the database. When we found the user we handle the password requirement as follows:

Listing 18: Verification of the password received.

```
var passwordIsValid = bcrypt.compareSync(req.body.password, user.password);
      if (!passwordIsValid) {
        return res.status(401).send({accessToken: null, message: "Invalid␣
            Password!"});
      }
```

With the bcrypt module we compare the passwords, so the database never has the password as plain text. If the password is incorrect we do not return an access token. If the password is correct we create a JWT to the user that expires in 24 hours and this token must come in every other call that the front-end makes in order to keep logged in. You can see all the code in Appendix C.3.

3.4.3 User functions

The user must be able to book a vehicle and release it while paying with SETK but paying without gas and to purchase tokens in exchange of a FIAT currency, euros in this particular case.

Booking a vehicle When the API receives a call to the *reserve* endpoint it updates the status of the vehicle of the plate received only if it is not booked, saves it in the log collection and update the users as it has reserved the vehicle in the time. The system uses the UTC+0 hour. Finally it is send to the handler to send the transaction with a fixed price as a *reserve fee* to the relay in order to get signed and processed in the MBTN blockchain.

Listing 19: Reserving vehicle function in the API

```
exports.bookVeichle = async (req, res) => {
    const payload = req.query.transaction;
    console.log('Booking: ', payload);
    const { RELAYER_API_KEY: apiKey, RELAYER_API_SECRET: apiSecret } =
        process.env;
    Vehicle.updateOne({plate: req.query.plate, booked: 0}, {$set:{booked:
        1, lastUser: req.query.username, reservedTime: Date().toLocaleString
        ()}}, function(err){
        if (err){
            res.status(500).send(err);
        }
    });
    Logs.create({plate: req.query.plate, User: req.query.username,
        releasedTime: "",reservedTime: Date().toLocaleString()}, function(
        err){
        if (err){
            res.status(500).send(err);
        }
    });
    User.updateOne({username: req.query.username}, {$set:{vehicle: req.
        query.plate, lastPicked: Date().toLocaleString()}}, function(err){
        if (err){
            res.status(500).send(err);
        }
    });
    handler({ apiKey, apiSecret, request: { body: JSON.parse(payload) } }).
        then(rsp => {res.status(200).send(rsp.txHash);})
        .catch(error => {
            console.log('Error', error);
            res.status(500).send(error);
        });
};
```

Releasing a vehicle This function is a mirror one of the previous one and follows the same structure. Now, the price in SETK of the transaction is calculated in the front-end taking into account the minutes that the vehicle has been used with a price of 0.02 SETK per minute. For development and testing purposes the price is fixed to 0.99 SETK to complete 1 SETK between reserving and releasing.

Listing 20: Release function in the API

```
exports.releaseVeichle = async (req, res) => {
    const payload = req.query.transaction;
    console.log('Releasing: ', payload);
    const { RELAYER_API_KEY: apiKey, RELAYER_API_SECRET: apiSecret } =
        process.env;
    console.log('plate', req.query.plate);
    Vehicle.updateOne(...);
    Logs.updateOne(...);
    User.updateOne(...);
    handler(...);
};
```

Minting SETK Minting is the process of creating tokens into an account and can only be called by the owner, SEAT. That is why this transaction is not process by the relayer. Because the project do not have access to the Volkswagen Payment Services, the part of the code that should handle the bank transaction is omitted, but commented where should it go and return a 500 status if any error occurs.

Listing 21: Purchase function in the API

```
exports.purchaseToken = async (req, res) => {
    //Check bank transaction and return state 500 if error
    const minted = await token.mintTokens(req.query.address, req.query.
        amount);
    res.status(200).send(minted);
};
```

The completed user functions can be found in Appendix C.4.

3.4.4 Interacting with the smart contracts

We have three main functions that interacts with the contract. One is a call so it does not cost any gas, the *balanceOf* function that return the actual balance of each user so we don't have to store it. The other ones are the *transfer* and *mint* function. The first one is going to be relayed, but the second one is going to be send by SEAT since it is a function that is only callable by the owner.

Listing 22: Creating the providers to interact with the contracts in the chain.

```
const { PRIVATE_KEY: privateKey, RPC: rpc, TOKEN_ADDRESS: tokenAddr,
    SEAT_ADDR: seatAddr } = process.env;
const provider = new Provider(privateKey,rpc);
const web3 = new Web3(provider);
const token = new web3.eth.Contract(ContractAbi, tokenAddr)
```

When we call the functions from the SEAT account, the structure is the same. Using the library **web3** and the Seat Private Key and the RPC, we connect with the contract to execute a function. The connection with the blockchain network is defined as the **provider**.

Listing 23: Different calls of the methods in the SETK contract.

```
const balance = token.methods.balanceOf(address).call({from: seatAddr}).
    then(balStr => balStr.toString()).then(realBal => web3.utils.fromWei(
    realBal));
const balance = await token.methods.mint(address, amountWei).send({from:
    seatAddr});
```

We execute one method or the other depending in the context, and the only difference is that we call the *balanceOf* function, that does not consume any gas as it is a lecture from the blockchain, or sending the function *mint* that cost gas.

Note in decimals in the SETK The ERC-20 deployed uses a 18 decimals notation. That means that 1 token in the blockchain has to be translated as ***1e18***. We call the *fromWei* function to make this transformation readable to the user. In the *mint* function the amount has been previously converted, to be align with the contract requirements.

When we have to relay a transaction the procedure is similar and it is used when a metatransaction is necessary. Now the provider is not the SEAT account, but the relayer that is connected to the MBTN

Listing 24: Connecting to the Defender Relayer Service

```
1 const provider = new DefenderRelayProvider(credentials);
2 const signer = new DefenderRelaySigner(credentials, provider, { speed: '
    fast' });
3 const forwarder = new ethers.Contract(forwAddr, ForwarderAbi,  signer)
```

We create the forwarder by passing the signer, who is the relay with the credentials and connect with the contract passing the Forwarder address and his ABI allowing the relayer to call the Forwarder contract. Finally, we relay the transaction by validating the request and executing to the relayer.

Listing 25: Validation and processing of a transaction by the Minimal Forwarder Contract.

```
1 // Validate request on the forwarder contract
2 const valid = await forwarder.verify(request, signature);
3 if (!valid) throw new Error('Invalid request');
4 // Send meta-tx through relayer to the forwarder contract
5 const gasLimit = (parseInt(request.gas) + 50000).toString();
6 return await forwarder.execute(request, signature, { gasLimit });
```

The transaction is signed and sent by the relayer and the user does not have to pay for any gas. The complete handling process is of a metatransaction is in the Appendix C.5.

3.5 The Front-end

The Front-End will establish the main functions but as an existing app is already developed, this part will be centered in the interaction with the smart contracts. Despite of that, the web dApp will have the basic functions excluding, for example, the location of the vehicles. In this section you will not find any interface that will be found in 4

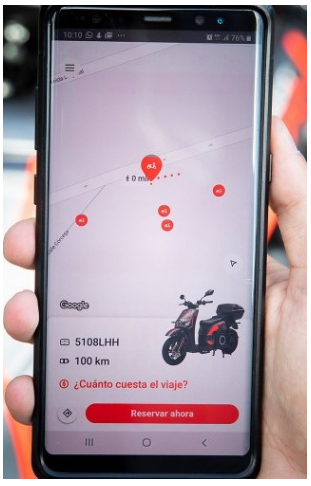


Figure 21: Capture of the mobility app by SEAT code. Source: SEAT MO web

3.5.1 Sing Up and Login Up

When singing in we create a user in the database, sending the username, the email, our wallet address, and when signing up we store the information in the cache the token and all the necessary data.

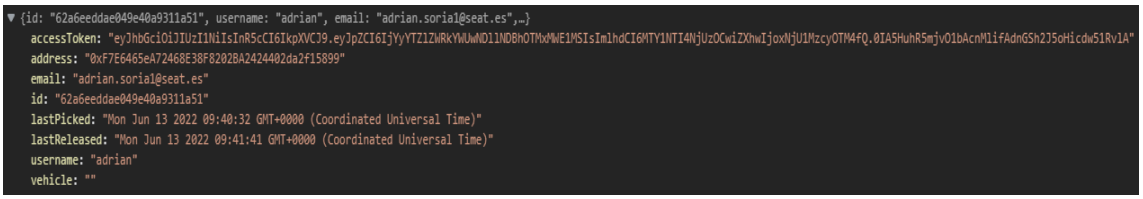


Figure 22: Cache in the web when signing in

With this information we are ready to start using the service.

3.5.2 Buying tokens

When charging the site, we need to set the current state. We put the actual price at 0, the SETK will be an input from the user and by clicking the button the *getUserBoard* function is executed if the introduced value is valid.

Listing 26: Button that executes the function purchase tokens

```
<button onClick={() => {
  if(this.state.priceEur > 0){
      UserService.purchaseTokens(currentUser.address,this.state.priceEur).
          then(window.location.reload()).catch(function(e){console.log('
          Error', e);});
  }else{
      this.notify('Invalid number');
  }
}} className="btn btn-primary">Buy</button>
```

We consider any positive value as a valid one.

Listing 27: GET method to the purchase function in the API instance

```
purchaseTokens(address, amount){
    return axios.get(API_URL + 'purchase', { headers: authHeader(), params
        :{address, amount}});
}
```

This functions makes a get to the purchase endpoint with the authHeader that contains the token. The mint function is explained in the 3.4.4.
The user must wait to the transaction to be processed and refresh the web page to obtain the new balance.

3.5.3 Using the service

As the purchasing tab, we update the local storage by calling the *getUserBoard* function. This function calls the *userBoard* function of the line 13 in the Appendix C.4, who updates the actual balance and returns some updated data. The vehicles field is an array of the vehicles available to choose between.
To understand the logic we might understand the following function:

Listing 28: Button that depending of the current state and signs and process a transaction.

```
<button onClick={() => {
  if (this.state.booked === "Reserve"){
      signTx(currentUser, 0.01, provider, token, this.state.booked,
          document.getElementById('plate').value).then(rsp => this.setState(
      {booked: "Release", pickTime: Date.now(), response: JSON.
          stringify(rsp), balance:(this.state.balance - 0.01),
          currentVehicle: document.getElementById('plate').value})).
          catch(function(e){
       console.log(e);
      });
  }else if(this.state.booked === "Release"){
      let currentprice = (((Date.now() - this.state.pickTime)/(6e4)) *
          0.02).toFixed(2);
      this.setState({price: "for" + currentprice.toString()});
      signTx(currentUser, 0.99, provider, token, this.state.booked, this.
          state.currentVehicle).then(rsp_tx => this.setState(
       {response: rsp_tx, booked: "Reload", balance:(this.state.balance
          - 0.99)})).catch(function(e){
       console.log(e);
      });
  }else{
      window.location.reload();
  }
}} class="btn btn-primary">
  {this.state.booked}
</button>
```

We have the initial state to reserve a vehicle (if there user has no vehicle already reserved), and call the sign function with an initial fee of 0.01. When the function finishes we change the actual state to the release mode, and we save the time when the function is called. When it it time to release, the price is calculated with a fee of 0.02 tokens per minute and the sign function it is called again to release the vehicle. Finally, the button can reload the page and it is a requirement that the user waits some seconds to get the balance updated. You can notice that a 0.99 fee is paid in the release vehicle for testing purposes.

Signing the transaction The provider for the frontend is Metamask, we will activate the service, and get the current network and check is in the MBTN and the Metamask account corresponds to the user.

Listing 29: Connecting and checking Metamask is operative.

```
if (!window.ethereum) throw new Error('User wallet not found');
```

```
await window.ethereum.enable();
const userProvider = new ethers.providers.Web3Provider(window.ethereum);
const userNetwork = await userProvider.getNetwork();
if (userNetwork.chainId !== 80001) throw new Error('Please switch to Mumbai Network for signing');
const signer = userProvider.getSigner();

const forwarder = createInstance(provider);
const from = await signer.getAddress();
if (from !== currentUser.address) throw new Error('Please switch to your account');
const data = token.interface.encodeFunctionData('transfer', ["0x40CDDe2b9BC7659C3349574Ec53db3B2bd9519BF", ethers.utils.parseEther(value.toString())]);
const to = token.address;
const request = await signMetaTxRequest(signer.provider, forwarder, { to, from, data });
```

The forwarding instance is created as a Minimal Forwarder (See *createInsance* function in Appendix F.1) and the address is passed to sign the meta transaction. This function (See in Appendix C.3) gets the nonce of the Forwarder contract, gets the network and parses the data as an EIP-712 and signs it.

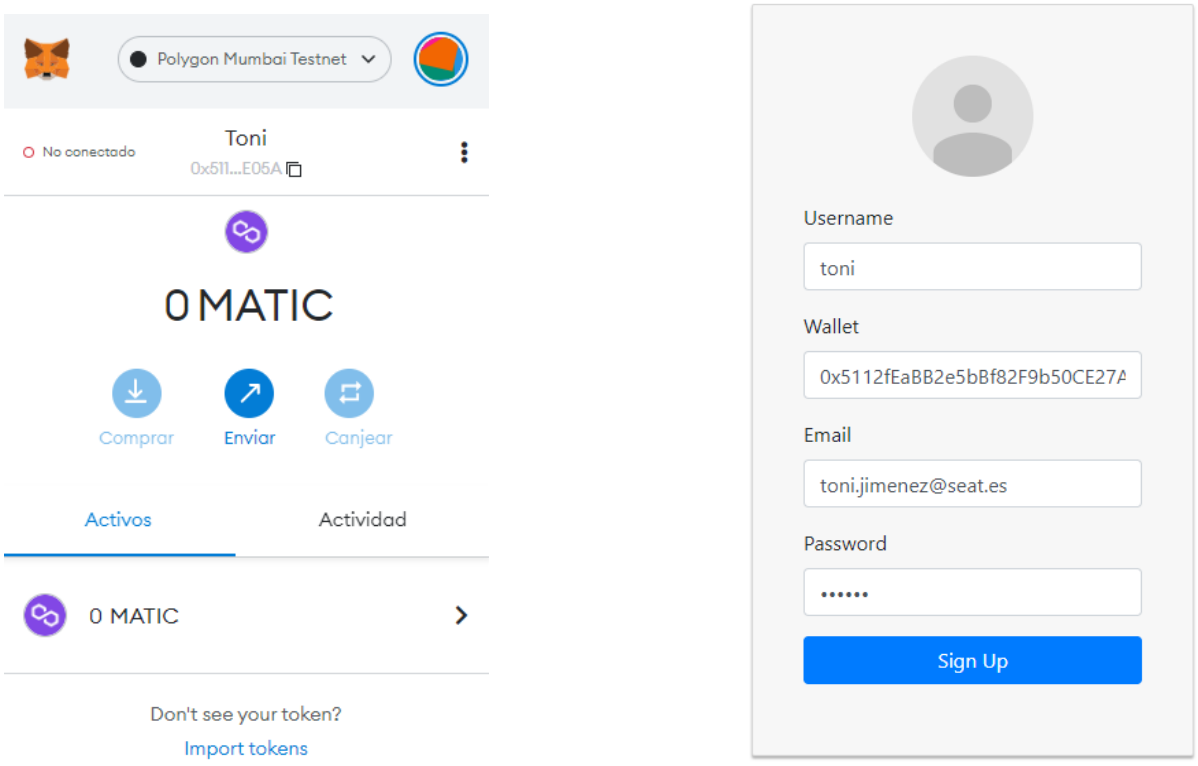
Listing 30: Example of a signature and transaction.

```
{
  "signature": "0xf107ddb8e431e5b6517867f1dc52261ebcb8de5677dfdbe0aabffaa06d0b472f5e8d2e50c944cbc9748e10488e1146e7de6e26c55c5f2737fe343453ced24af51b",
  "request": {
    "value": 0,
    "gas": 1000000,
    "nonce": "1",
    "to": "0x86d269c9a6c3C9410228c10c8b933dEad4ab40e9",
    "from": "0xF7E6465eA72468E38F8202BA2424402da2f15899",
    "data": "0xa9059cbb000000000000000000000005d4f37e5b030fadb3876f0608ab40e95027a85040000000000000000000000000000000000000000000000000de0b6b3a7640000"
  }
}
```

When the signature and request are returned, depending on the action that we want to perform, we call the *reserveVehicle* or *releaseVehicle* functions that send to the corresponding endpoint to call the corresponding function in the API (See Appendix C.4).

4 Final result

We are going to begin with the all the process that the user should see. The project can be found here[1] using the public DNS that Amazon offers and it is only for testing. It is also possible that the service is no longer available because of the developing stage that is at the moment of writting this thesis.
To use the dApp it is mandatory to have a Metamask account in the MBTN.



(a) Metamask account

(b) Register fields

Figure 23: Sign Up process

Because the AWS KMS service is not used we need to import the Ethereum Address. If any field (except the password) is already in the database, the Sign Up will fail.

[1] If you cannot click, the link is the following: https://ec2-18-197-1-4.eu-central-1.compute.amazonaws.com:8081/

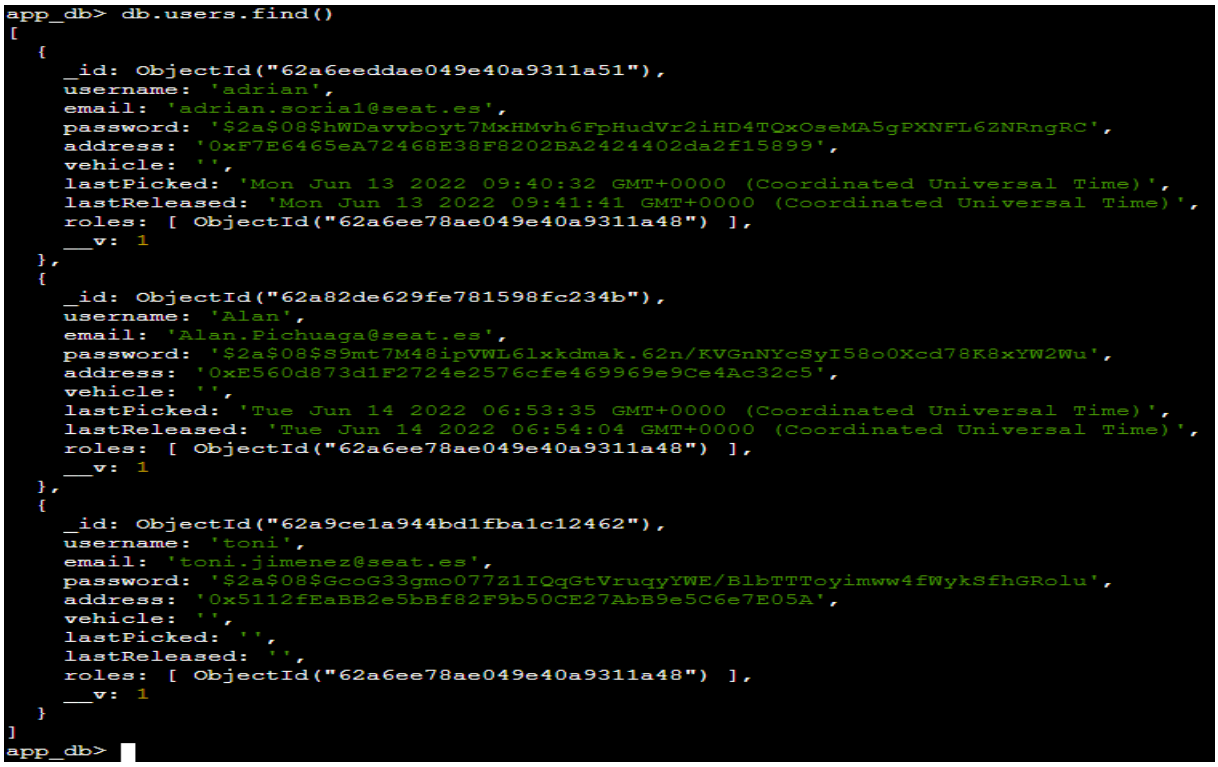


Figure 24: DB with some users

The password is encrypted when stored to prevent data leak. When the user is registered in the DB we can Sign In in the app and go to our profile.

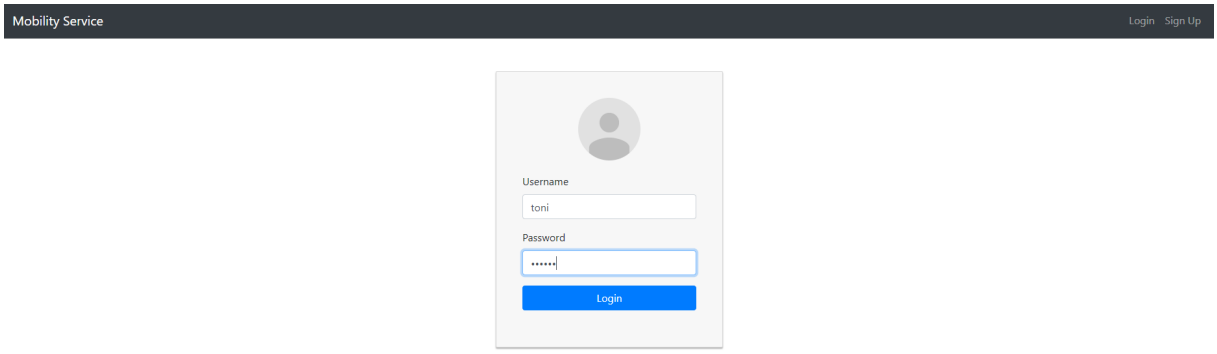


Figure 25: Sign In page

When the user properly signs in, it is redirected to his user page. Notice that now he can navigate to the Purchase token and Book a vehicle tabs at the top of the window.

Figure 26: Profile tab of a user in the web page

We have 0 tokens in our account, so we are going to purchase some tokens.

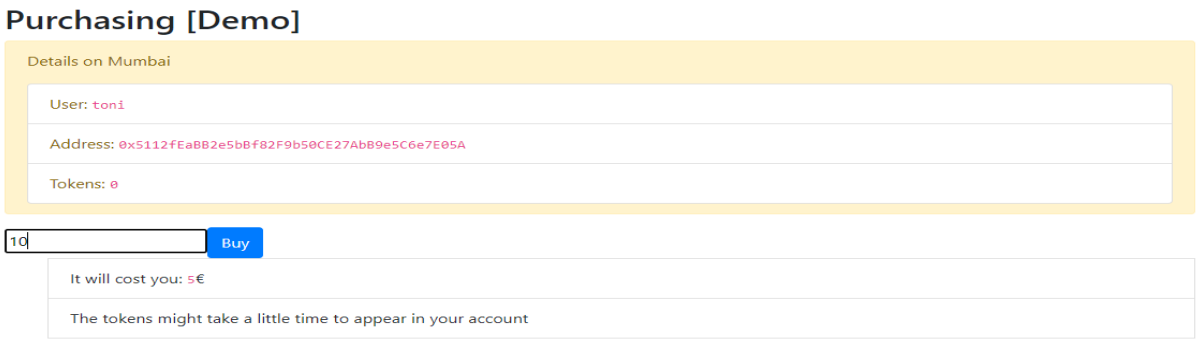


Figure 27: Profile tab of a user in the web page

The ratio is 2 token per euro. According the information provided by SEAT MO, the average cost of a trip is 2.77€, so this ratio must be adjusted by the company to make it economically valid (A brief analysis of this ratio can be found in 5). The converter is dynamically adjusted to acknowledge the user how many euros will the purchase cost. When we press buy a transaction is sent to the API and minted by the SEAT account.

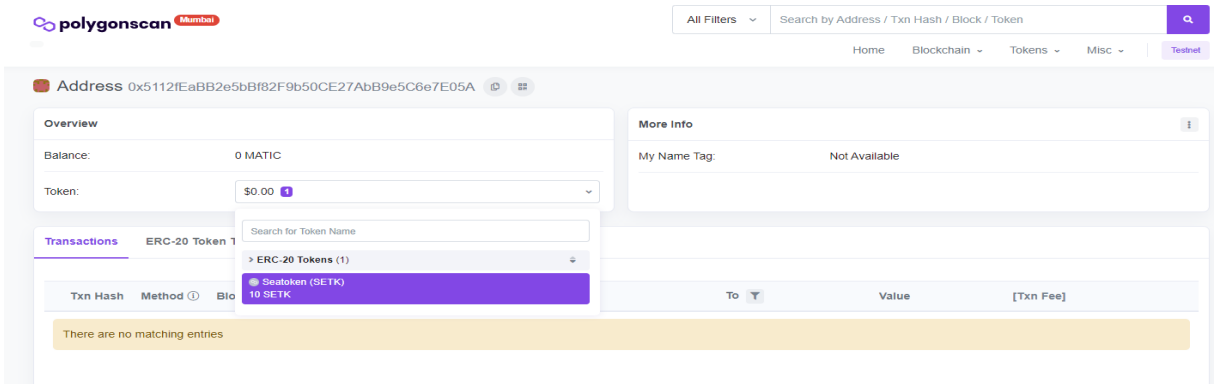


Figure 28: Balance of SETK in Toni's address

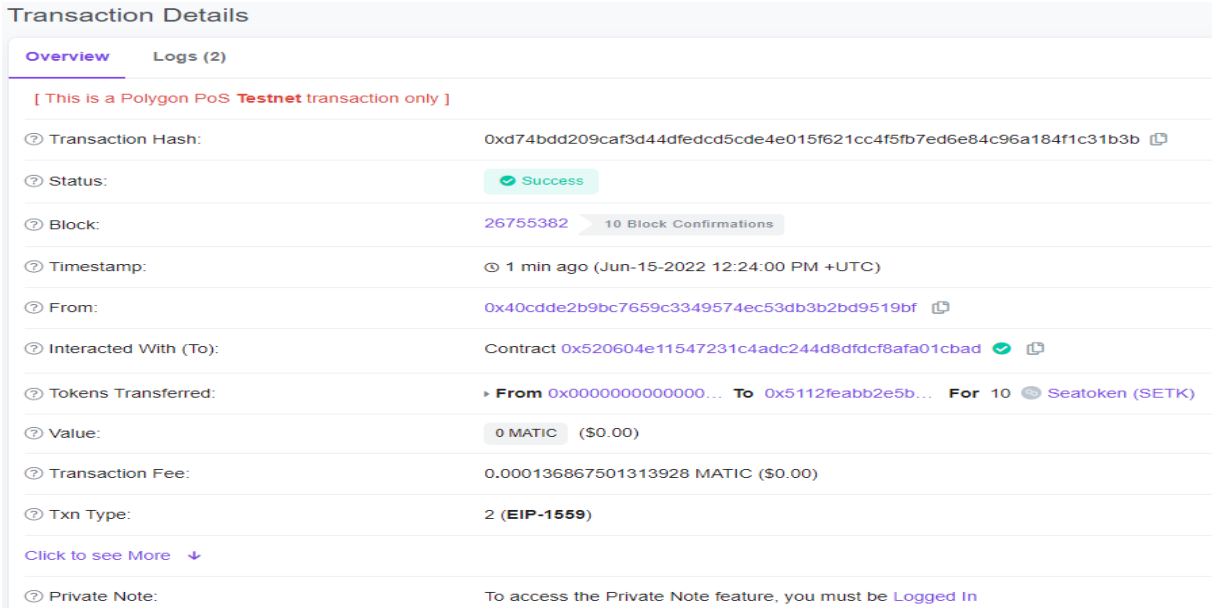


Figure 29: Transaction details

The most important part is that the gas is payed by the SEAT account **(0x40cdd...)** and is who pays for the gas. When the user has tokens, it can use the service.
In the booking page we update the local storage cache every time we refresh the page with the new information, when was last picked vehicle and when it was released and if the user has a current vehicle.

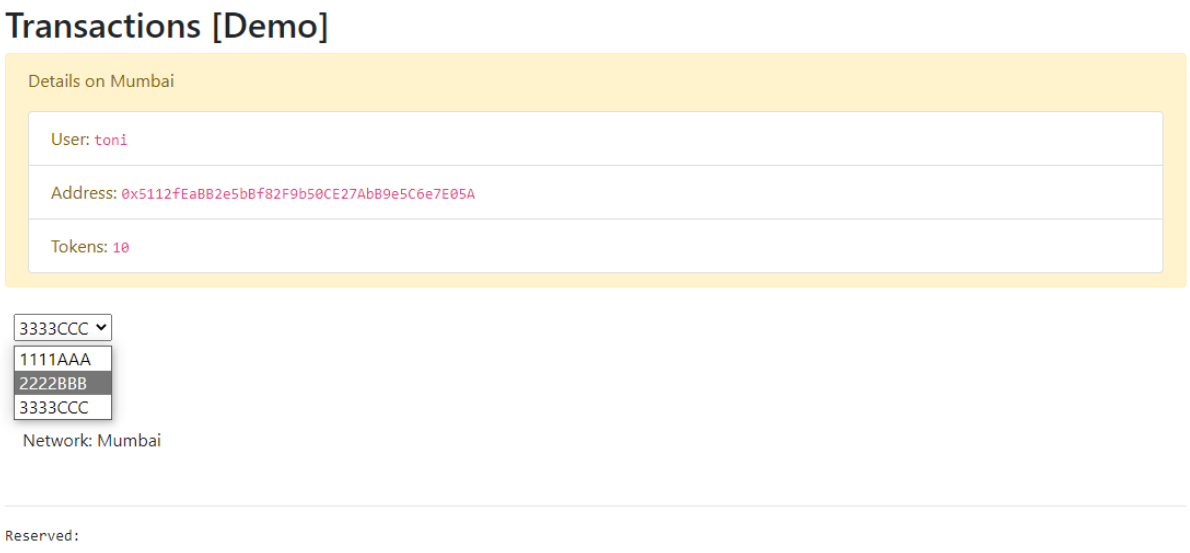


Figure 30: Booking vehicle web page

In the booking web page we can see three vehicles that were previously created in respective collection in the DB. The main identification that we are going to use it the plate of

the vehicle.

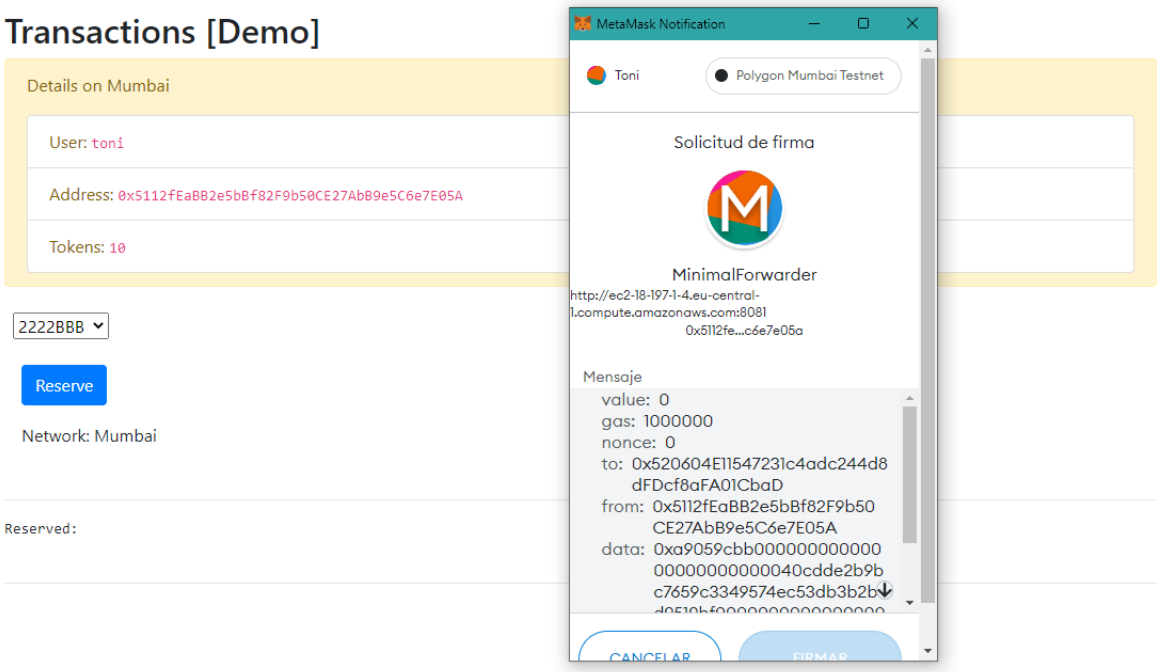


Figure 31: Signature of a vehicle with Metamask.

When pressing the reserve button, the Metamask window pops up to sign the transaction. Once it is mounted and sent, the status changes to Release the vehicle and accessing with another user the vehicle reserved does not appear in the list.

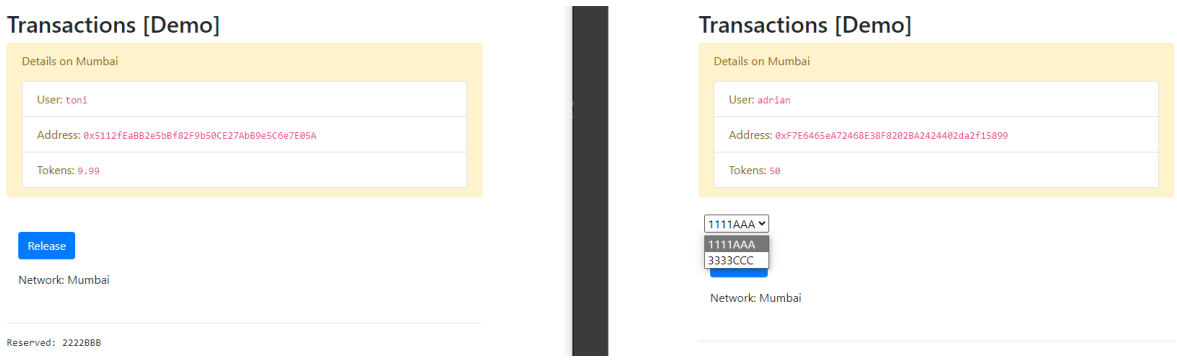


Figure 32: User trying to reserve a vehicle while one is reserved.

When pressing the Release button, another Metamask window pops up to sign the Typed Data Transaction, and after waiting until the transaction is processed in the blockchain, the user may reload the web page to see the new token balance. In the defender relayer we can see both transactions payed by the relayer

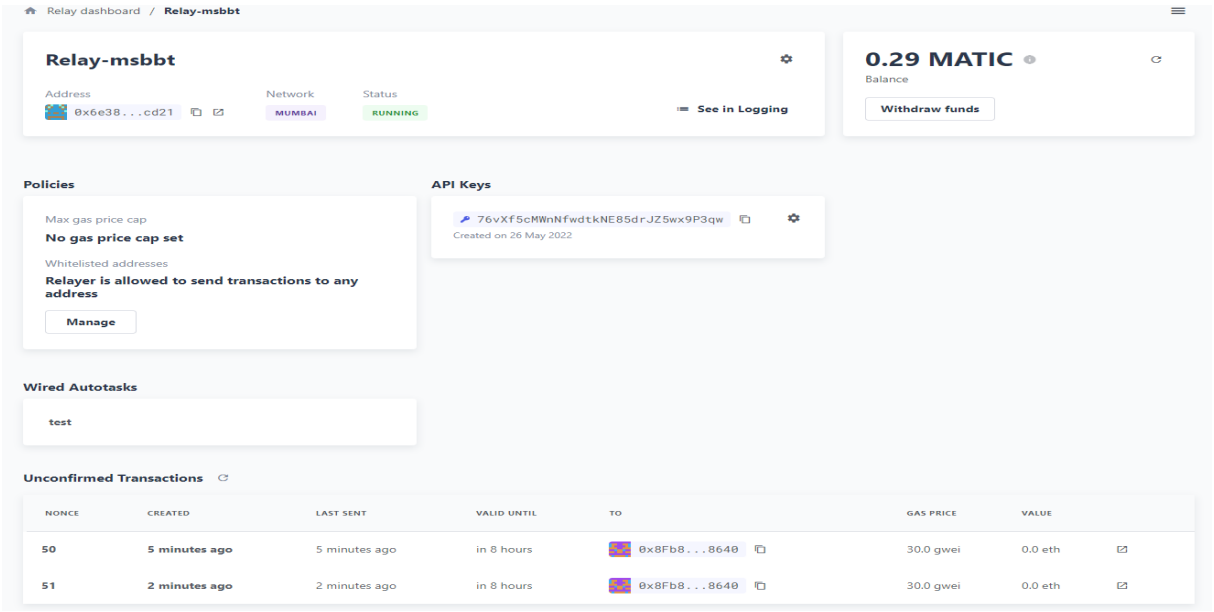


Figure 33: Defender Relayer with the reserve and release transaction

The flow of the application is completed with the two transactions in the relayer that holds the transactions until they are confirmed. All transactions are going to have the Minimal Forwarder contract as a receiver and this forwarder calls the *transfer* function. In the MBTN block scan we can see all the transactions made.

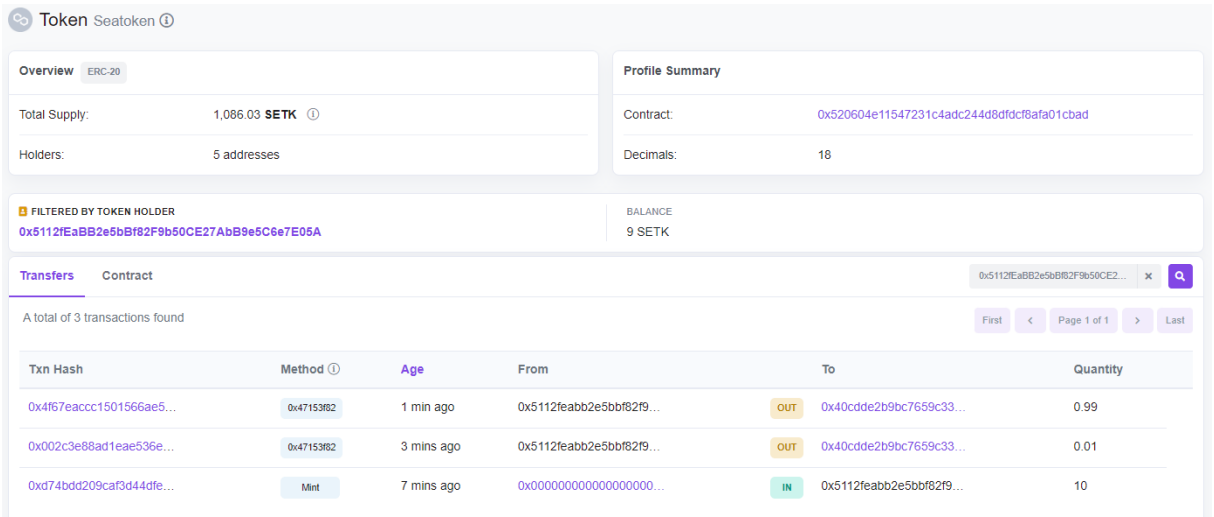


Figure 34: Transactions of the tokens in the blockchain

The address is registered with a timestamp and the method called, with the quantity that allow us to know if a vehicle is reserved or released. With this the flow of the service is finished.

5 Budget

The budget for this project will take into account the instances created in the Amazon Web Service despite of this servers are currently in SEAT so the will not have an extra cost. It also contains the salary of the engineer and the estimated cost of MATIC to pay the gas.

Table 1: Budget for the project

	Time	Transactions	Cost	Total
EC2 Instances	5 Months	None	550€/month	2750€
Junior Engineer	620 hours	None	9€/h	5580€
Senior Engineer	30 hours	None	30€/h	900€
MATIC	3896 TX/day	0.003 MATIC/TX	0,7€/MATIC	1227.24€
			Total	**10457.24€**

The MATIC cost has been calculated as the entire fleet of the 681 vehicles[16] made 2,86 trips per day[2], that makes 1948 trips/day, which are 3896 transactions (reserving and releasing) per day, then the cost has been calculated for the duration of the thesis, 5 months. With the maximum historic price of the euro per MATIC and rounding up the max value of the gas fee payed while developing this project to get the worst case scenario. We get the total cost of the MATIC by multiplying the number of transactions per day with the MATIC/transaction ratio and the cost of one MATIC, then extended for the extending it for 5 months.
In average, 2.77€ are expended by the user in every trip. With a price of 2 SETK per euro, if the company maintains the euro/SETK cost, the user will expend half in the trip, and since minting any amount of tokens costs 0.003 MATIC approximately in gas, SEAT MO will receive more value for every trip.
In addition, a senior engineer is taken into account with a consulting role.

[2]Information of the vehicles and trips provided by SEAT MO

6 Conclusions

Blockchain is a growing technology with lots of potential.The project tries to put trust in a decentralized system. Despite that it has a higher latency than the current system, the user can trust all his movements, make secure payments and use tokens in a new, and every day more popular, virtual world.

The dApp integrates the blockchain technology and the Solidity programming language with the every day standards such as, Java, NodeJS or MongoDB and works with the proper asynchronous processing. The front-end of the project implements the basic functions and works as a PoC with limitations in the number of queries and some minor bugs that the SEAT MOtosharing app does not have due to the application is in a production environment. The contracts are developed in Solidity, which is the most use language to developing smart contracts and can be deployed in any network that uses Ethereum standards which makes it a very scalable solution. The downside is that a completely usability without any knowledge of a Blockchain or Wallet is not possible because the AWS KMS service[8] is not implemented due the time expended working in an implementation from the zero. Nevertheless, it could be implemented with more time.

To conclude, the objective of developing a use case for integrating a sharing mobility service is completed and functional, and with a proper migration to the SEAT MO servers can be quickly implemented, as it uses the standard ERC-20 and EIP-712 and it is able to be implemented in another services and necessities that the company may have in the future.

7 Future of the project

The future of the project will depend on how much the company will invest in the Blockchain technology with other utility cases, such a digital twin or with Metahype[19]. In regard to this specific project, with more dedication and time, some improvements should be done:

- **Integrating the vehicles as an ERC-721[9]**
  Also known as NFT, the ERC-721 defines a non consumable token, a digital asset. The interesting part regarding this project is that you can define a user that can call the reserve and release function inside an ERC-721 contract, and with a mapping, keep track of which person had which vehicle at any time. You can also define the state of the vehicle and any other attribute that the implementation may need. If this is done, the logs in the DB will not be mandatory as all the registers are in the blockchain. The disadvantage is that more calls to the blockchain must be made so the budget will increase, but the fee of reserving a vehicle can be omitted since his only objective is to track that a vehicle is reserved.
- **Integrating properly with the AWS KMS**
  Due to lack of knowledge and time, the author was not able to create a Wallet and sign a metatransaction using the Key Management Service. This will increase the engagement with the general public because they will not have to know that the dApp is working in a blockchain, and eliminates the process of creating and managing a wallet, making the flow more agile.
- **Implementation of the own relayer and gas station[11]**
  In the project, the Defender Relayer free tier is used. For a develop stage works perfectly fine with a maximum of 120 transactions per hour, but in a production environment, the company should integrate his own relayer to not be obligated to pay for a premium tier in the Defender Relayer service. This will increase the cost in short term basis but the decision is more economically viable in a long term basis.

References

[1] Andreas M. Antonopoulos and Gavin Wood. *Mastering Ethereum*. O'Reilly Media, Inc., November 2018. ISBN 9781491971949.

[2] Remco Bloemen, Leonid Logvinov, and Jacob Evans. Eip-712: Ethereum typed structured data hashing and signing, September 2017. URL https://eips.ethereum.org/EIPS/eip-712.

[3] Demian Brener. Openzeppelin, 2017. URL https://docs.openzeppelin.com/.

[4] Vitalik Buterin. Eip-155: Simple replay attack protection, October 2016. URL https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md.

[5] D. Cooper, Trinity College Dublin S. Farrell, S. Boeyen, R. Housley Entrust, Vigil Security, W. Polk, and NIST. Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile, May 2008. URL https://datatracker.ietf.org/doc/html/rfc5280#page-16.

[6] Jacques Dafflon, Jordi Baylina, and Thomas Shababi. Eip-777: Token standard, November 2017. URL https://eips.ethereum.org/EIPS/eip-777.

[7] David Dornseifer. Use key management service (aws kms) to securely manage ethereum accounts, 2021. URL https://aws.amazon.com/es/blogs/database/part1-use-aws-kms-to-securely-manage-ethereum-accounts/.

[8] Rafael Genés Durán, Diana Yarlequé-Ruesta, Marta Bellés-Muñoz, Antonio Jimenez-Viguer, and José L. Muñoz-Tapia. An architecture for easy onboarding and key lifecycle management in blockchain applications. *IEEE Access*, 8:115005–115016, 2020. doi: 10.1109/ACCESS.2020.3003995.

[9] William Entriken, Dieter Shirley, Jacob Evans, and Nastassia Sachs. Eip-721: Non-fungible token standard, January 2018. URL https://eips.ethereum.org/EIPS/eip-721.

[10] Ethereum. Rlp, November 2020. URL https://eth.wiki/fundamentals/rlp.

[11] Ethereum. Gas station network, May 2021. URL https://docs.opengsn.org/.

[12] Meta (Facebook). Metaverse by meta, 2022. URL https://about.facebook.com/en/meta.

[13] Cambridge Center for Alternative Finance. Cambridge bitcoin electricity consumption index, September 2020. URL https://ccaf.io/cbeci/index.

[14] Prof. Dr. Robby HOUBEN and Alexander SNYERS. Cryptocurrencies and blockchain: Legal context and implications for financial crime, money laundering and tax evasion, July 2018. URL https://www.europarl.europa.eu/cmsdata/150761/TAX3%20Study%20on%20cryptocurrencies%20and%20blockchain.pdf.

[15] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols(extended abstract), 1999. URL https://link.springer.com/chapter/10.1007/978-0-387-35568-9_18.

[16] SEAT MO. A day of moto-sharing, November 2020. URL https://www.seat.com/company/news/company/a-day-of-moto-sharing.html.

[17] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, October 2008. URL https://bitcoin.org/bitcoin.pdf.

[18] National Institute of Standards and Technology. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, August 2015. URL https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[19] Cupra Official. Metahype, 2022. URL https://www.cupraofficial.es/metahype.html.

[20] OpenZeppelin. Openzeppelin defender, August 2019. URL https://defender.openzeppelin.com.

[21] Santiago Palladino. Meta transactions powered by openzeppelin defender, March 2021. URL https://github.com/OpenZeppelin/workshops/blob/master/01-defender-meta-txs/slides/20210211%20-%20Defender%20meta-txs%20workshop.pdf.

[22] Polygon. Full node deployment, November 2021. URL https://docs.polygon.technology/docs/develop/network-details/full-node-deployment.

[23] Certicom Research. Sec 2: Recommended elliptic curve domain parameters, January 2010. URL https://www.secg.org/sec2-v2.pdf.

[24] Fabian Vogelsteller and Vitalik Buterin. Eip-20: Token standard, November 2015. URL https://eips.ethereum.org/EIPS/eip-20#token.

Appendices

A Bor and Heimdall logs

Once the execution is finished the Bor and Heimdall services are working and producing blocks.

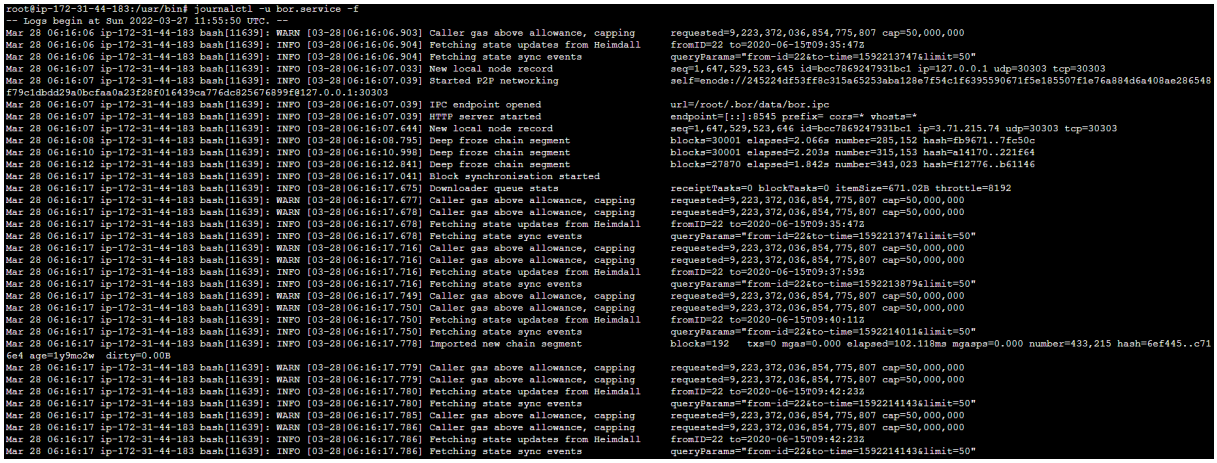


Figure 35: Log Bor when starts running

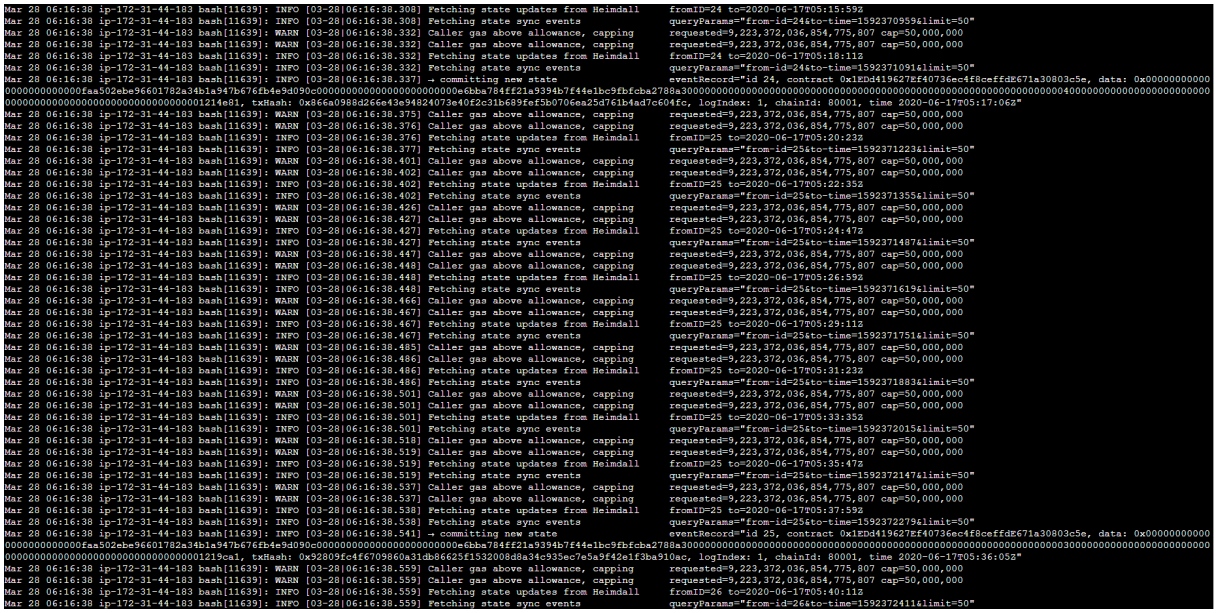


Figure 36: Log Bor syncing with Heimdall

We can see that Bor starts the HTTP server, and sync the current state with Heimdall and looking for the necessary gas. As it generates blocks, it commits new states, that are the hashes of the blocks synced in the side-chain.

```
root@ip-172-31-44-183:~# journalctl -u heimdalld -f
-- Logs begin at Mon 2022-03-21 06:09:55 UTC. --
Mar 21 06:36:59 ip-172-31-44-183 heimdalld[1832]: I[2022-03-21|06:36:59.141] Committed state                    module=state height=726415 txs=0 appHash=A490BE04247FF0714540E8B42FC95198F3E3991794
BF02E60516DAFD59FB2D54
Mar 21 06:36:59 ip-172-31-44-183 heimdalld[1832]: I[2022-03-21|06:36:59.218] Executed block                     module=state height=726416 validTxs=0 invalidTxs=0
Mar 21 06:36:59 ip-172-31-44-183 heimdalld[1832]: I[2022-03-21|06:36:59.636] Committed state                    module=state height=726416 txs=0 appHash=A490BE04247FF0714540E8B42FC95198F3E3991794
BF02E60516DAFD59FB2D54
Mar 21 06:36:59 ip-172-31-44-183 heimdalld[1832]: I[2022-03-21|06:36:59.703] Executed block                     module=state height=726417 validTxs=0 invalidTxs=0
Mar 21 06:37:00 ip-172-31-44-183 heimdalld[1832]: I[2022-03-21|06:37:00.115] Committed state                    module=state height=726417 txs=0 appHash=A490BE04247FF0714540E8B42FC95198F3E3991794
BF02E60516DAFD59FB2D54
Mar 21 06:37:00 ip-172-31-44-183 heimdalld[1832]: I[2022-03-21|06:37:00.185] Executed block                     module=state height=726418 validTxs=0 invalidTxs=0
Mar 21 06:37:00 ip-172-31-44-183 heimdalld[1832]: I[2022-03-21|06:37:00.661] Committed state                    module=state height=726418 txs=0 appHash=A490BE04247FF0714540E8B42FC95198F3E3991794
BF02E60516DAFD59FB2D54
```

Figure 37: Log of Heimdall service running

B The v value

The v value in a blockchain corresponds to the recovery ID. It is calculated as $v = 0,1 + CHAIN_ID * 2 + 35$. To understand this, we have the r value, which is the x component of the point in the curve used for signing. Then we have two candidates for being the other component of the poind and, according to the EIP-155[4], v has two possible values depending on the side of the curve you are, and those values are 27 and 28.

C API code

C.1 Routes in the API

```
const { verifySignUp } = require("../middlewares");
const controller = require("../controllers/auth.controller");
module.exports = function(app) {
  app.use(function(req, res, next) {
    res.header("Access-Control-Allow-Headers","x-access-token, Origin, Content-Type, Accept");
    next();});
  app.post("/api/auth/signup",[verifySignUp.checkDuplicateUsernameOrEmail], controller.signup);
  app.post("/api/auth/signin", controller.signin);
};
```

```
const { authJwt } = require("../middlewares");
const controller = require("../controllers/user.controller");
module.exports = function(app) {
  app.use(function(req, res, next) {
    res.header("Access-Control-Allow-Headers","x-access-token, Origin, Content-Type, Accept");
    next();});
  app.get("/api/test/all", controller.allAccess);
  app.get("/api/test/user", [authJwt.verifyToken], controller.userBoard);
  app.get("/api/test/reserve", [authJwt.verifyToken], controller.bookVeichle);
```

```
10   app.get("/api/test/release", [authJwt.verifyToken], controller.
         releaseVeichle);
11   app.get("/api/test/purchase", [authJwt.verifyToken], controller.
         purchaseToken);
12 };
```

C.2 Sign Up Code

```
1  exports.signup = async (req, res) => {
2    console.log("USUARI: " + req.body.username);
3    const user = new User({
4      username: req.body.username,
5      email: req.body.email,
6      password: bcrypt.hashSync(req.body.password, 8),
7      address: req.body.address,
8      vehicle: "",
9      lastPicked: "",
10     lastReleased: ""
11   });
12   user.save((err, user) => {
13     if (err) {
14       res.status(500).send({ message: err });
15       return;
16     }
17     res.send({ message: "User was registered successfully!" });
18   });
19 };
```

C.3 Sign In Code

```
1  exports.signin = (req, res) => {
2    User.findOne({
3      username: req.body.username
4    }).exec((err, user) => {
5        if (err) {
6          res.status(500).send({ message: err });
7          return;
8        }
9        if (!user) {
10         return res.status(404).send({ message: "User Not found." });
11       }
12       var passwordIsValid = bcrypt.compareSync(
13         req.body.password,
14         user.password
15       );
16       if (!passwordIsValid) {
17         return res.status(401).send({
18           accessToken: null,
19           message: "Invalid Password!"
20         });
21       }
22       var token = jwt.sign({ id: user.id }, config.secret, {
```

```
        expiresIn: 86400 // 24 hours
      });
      res.status(200).send({
        id: user._id,
        username: user.username,
        email: user.email,
        accessToken: token,
        address: user.address,
        vehicle: user.vehicle,
        lastPicked: user.lastPicked,
        lastReleased: user.lastReleased,
        lastTransaction: user.lastTransaction
      });
    });
};
```

C.4 User functions

```
const db = require("../models");
const Vehicle = db.vehicles;
const Logs = db.logs;
const User = db.user;
const token = require('../eth/token');
const { handler } = require('../eth/relay');
require('dotenv').config();
exports.allAccess = (req, res) => {
    res.status(200).send("Welcome to the Mobility Service app!");
};
exports.userBoard = async (req, res) => {
    const actBalance = await token.getBalance(req.query['0']);
    User.findOne({address: req.query['0']}).exec((err, user) =>{
        if(err){
            res.status(500).send({message: err});
        }
        Vehicle.find({booked:0}, {_id: 0, plate: 1}).exec((err, data)=>{
            if(err){
                res.status(500).send({message: "No vehicles available"});
            }
                res.status(200).send({
                    balance: actBalance,
                    vehicles: data,
                    vehicle: user.vehicle,
                    lastPicked: user.lastPicked,
                    lastReleased: user.lastReleased
                });
        });
    });
};
exports.bookVeichle = async (req, res) => {
    const payload = req.query.transaction;
    console.log('Booking: ', payload);
    const { RELAYER_API_KEY: apiKey, RELAYER_API_SECRET: apiSecret } =
        process.env;
```

```
35      Vehicle.updateOne({plate: req.query.plate, booked: 0}, {$set:{booked:
            1, lastUser: req.query.username, reservedTime: Date().toLocaleString
            ()}}, function(err){
36          if (err){
37              res.status(500).send(err);
38          }
39      });
40      Logs.create({plate: req.query.plate, User: req.query.username,
            releasedTime: "",reservedTime: Date().toLocaleString()}, function(
            err){
41          if (err){
42              res.status(500).send(err);
43          }
44      });
45      User.updateOne({username: req.query.username}, {$set:{vehicle: req.
            query.plate, lastPicked: Date().toLocaleString()}}, function(err){
46          if (err){
47              res.status(500).send(err);
48          }
49      });
50      handler({ apiKey, apiSecret, request: { body: JSON.parse(payload) } }).
            then(rsp => {res.status(200).send(rsp.txHash);})
51          .catch(error => {
52              console.log('Error', error);
53              res.status(500).send(error);
54          });
55  };
56  exports.releaseVeichle = async (req, res) => {
57      const payload = req.query.transaction;
58      console.log('Releasing:␣', payload);
59      const { RELAYER_API_KEY: apiKey, RELAYER_API_SECRET: apiSecret } =
            process.env;
60      console.log('plate', req.query.plate);
61      Vehicle.updateOne({plate: req.query.plate}, {$set:{booked: 0,
            reservedTime: ""}}, function(err){
62          if (err){
63              res.status(500).send(err);
64          }
65      });
66      Logs.updateOne({plate: req.query.plate, releasedTime: ""}, {$set:{
            releasedTime: Date().toLocaleString()}}, function(err){
67          if (err){
68              res.status(500).send(err);
69          }
70      });
71      User.updateOne({username: req.query.username}, {$set:{vehicle: "",
            lastReleased: Date().toLocaleString()}}, function(err){
72          if (err){
73              res.status(500).send(err);
74          }
75      });
76      handler({ apiKey, apiSecret, request: { body: JSON.parse(payload) } }).
            then(rsp => {res.status(200).send(rsp.txHash);})
77          .catch(error => {
78              console.log('Error', error);
```

```
            res.status(500).send(error);
        });

};
exports.purchaseToken = async (req, res) => {
    //Check bank transaction and return state 500 if error
    console.log('Miniting ' + req.query.amount + ' to ' + req.query.address
        );
    const minted = await token.mintTokens(req.query.address, req.query.
        amount);
    res.status(200).send(minted);
};
```

The user functions are called by the react front-end but processed in the API. The request is signed by the user to verify the identity.

C.5 Handeling the transactions in the blockchain

```
async function relay(forwarder, request, signature) {
    // Validate request on the forwarder contract
    const valid = await forwarder.verify(request, signature);
    if (!valid) throw new Error(`Invalid request`);
    console.log('Valid', valid);

    // Send meta-tx through relayer to the forwarder contract
    const gasLimit = (parseInt(request.gas) + 50000).toString();
    return await forwarder.execute(request, signature, { gasLimit });
}

async function handler(event) {
    require('dotenv').config();
    const { FORWARDER_ADDRESS: forwAddr } = process.env;
    // Parse webhook payload
    if (!event.request || !event.request.body) throw new Error(`Missing
        payload`);
    const { request, signature } = event.request.body;

    // Initialize Relayer provider and signer, and forwarder contract
    const credentials = { ...event };
    const provider = new DefenderRelayProvider(credentials);
    const signer = new DefenderRelaySigner(credentials, provider, { speed:
        'fast' });
    const forwarder = new ethers.Contract(forwAddr, ForwarderAbi, signer)
    console.log('Relaying...');
    // Relay transaction!
    const tx = await relay(forwarder, request, signature);
    console.log(`Sent meta-tx: ${tx.hash}`);
    return { txHash: tx.hash };
}
```

The API handles the event by connecting to the relayer and interacting with the forwarder contract.

D Contracts

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol";
import "@openzeppelin/contracts/security/Pausable.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/metatx/ERC2771Context.sol";
import "@openzeppelin/contracts/metatx/MinimalForwarder.sol";

contract Seatoken is ERC2771Context, ERC20, ERC20Burnable, Pausable,
    Ownable {
    constructor(MinimalForwarder forwarder) ERC2771Context(address(
        forwarder)) ERC20("Seatoken", "SETK") {
    }
    function pause() public onlyOwner {
        _pause();
    }
    function unpause() public onlyOwner {
        _unpause();
    }
    function mint(address to, uint256 amount) public onlyOwner {
        _mint(to, amount);
    }
    function _msgSender() internal view override(Context, ERC2771Context)
        returns(address) {
        return ERC2771Context._msgSender();
    }
    function _msgData() internal view override(Context, ERC2771Context)
        returns(bytes memory)
    {
        return ERC2771Context._msgData();
    }
    function _beforeTokenTransfer(address from, address to, uint256 amount)
    internal
    whenNotPaused
    override
    {
        super._beforeTokenTransfer(from, to, amount);
    }
}
```

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v4.5.0) (metatx/MinimalForwarder.
    sol)
pragma solidity ^0.8.0;
import "../utils/cryptography/ECDSA.sol";
import "../utils/cryptography/draft-EIP712.sol";

contract MinimalForwarder is EIP712 {
    using ECDSA for bytes32;
    struct ForwardRequest {
        address from;
        address to;
```

```
12          uint256 value;
13          uint256 gas;
14          uint256 nonce;
15          bytes data;
16      }
17      bytes32 private constant _TYPEHASH = keccak256("ForwardRequest(address
            from,address to,uint256 value,uint256 gas,uint256 nonce,bytes data)"
            );
18
19      mapping(address => uint256) private _nonces;
20      constructor() EIP712("MinimalForwarder", "0.0.1") {}
21
22      function getNonce(address from) public view returns (uint256) {
23          return _nonces[from];
24      }
25      function verify(ForwardRequest calldata req, bytes calldata signature)
            public view returns (bool) {
26          address signer = _hashTypedDataV4(
27              keccak256(abi.encode(_TYPEHASH, req.from, req.to, req.value,
                    req.gas, req.nonce, keccak256(req.data)))
28          ).recover(signature);
29          return _nonces[req.from] == req.nonce && signer == req.from;
30      }
31      function execute(ForwardRequest calldata req, bytes calldata signature)
            public payable returns (bool, bytes memory){
32          require(verify(req, signature), "MinimalForwarder: signature does
                not match request");
33          _nonces[req.from] = req.nonce + 1;
34          (bool success, bytes memory returndata) = req.to.call{gas: req.gas,
                value: req.value}(
35              abi.encodePacked(req.data, req.from)
36          );
37          // Validate that the relayer has enough gas for the call.
38          if (gasleft() <= req.gas / 63) {
39              assembly {
40                  invalid()
41              }
42          }
43          return (success, returndata);
44      }
45  }
```

```
1   // SPDX-License-Identifier: MIT
2   // OpenZeppelin Contracts (last updated v4.5.0) (metatx/ERC2771Context.sol)
3   pragma solidity ^0.8.9;
4   import "../utils/Context.sol";
5   abstract contract ERC2771Context is Context {
6       address private immutable _trustedForwarder;
7
8       constructor(address trustedForwarder) {
9           _trustedForwarder = trustedForwarder;
10      }
11      function isTrustedForwarder(address forwarder) public view virtual
            returns (bool) {
12          return forwarder == _trustedForwarder;
```

```
13      }
14      function _msgSender() internal view virtual override returns (address
            sender) {
15          if (isTrustedForwarder(msg.sender)) {
16              assembly {
17                  sender := shr(96, calldataload(sub(calldatasize(), 20)))
18              }
19          } else {
20              return super._msgSender();
21          }
22      }
23      function _msgData() internal view virtual override returns (bytes
            calldata) {
24          if (isTrustedForwarder(msg.sender)) {
25              return msg.data[:msg.data.length - 20];
26          } else {
27              return super._msgData();
28      }}
29  }
```

E Seatoken Artifact

```
{
  "_format": "hh-sol-artifact-1",
  "contractName": "Seatoken",
  "sourceName": "contracts/Seatoken.sol",
  "abi": [
    {
      "inputs": [
        {
          "internalType": "contract MinimalForwarder",
          "name": "forwarder",
          "type": "address"
        }
      ],
      "stateMutability": "nonpayable",
      "type": "constructor"
    },
    {
      "anonymous": false,
      "inputs": [
        {
          "indexed": true,
          "internalType": "address",
          "name": "from",
          "type": "address"
        },
        {
          "indexed": true,
          "internalType": "address",
          "name": "to",
          "type": "address"
        },
```

```
        {
          "indexed": false,
          "internalType": "uint256",
          "name": "value",
          "type": "uint256"
        }
      ],
      "name": "Transfer",
      "type": "event"
    }
    ...
  ],
  "byteCode": ...
}
```

The *bytecode* has been deleted because of his longitude and some other functions declarations just to make clear how an artifact and an ABI works. In the ABI we can see the functions. For instance, the transfer function has an address input with the name *to* and an unit256 named *amount*, and returns a boolean.

F Front-end Code

Here is presented the complete necessary code to understand the front-end.

F.1 Signing functions

```
1  export async function signTx(currentUser, value, provider, token, action,
       plate){
2      if (!window.ethereum) throw new Error('User wallet not found');
3      await window.ethereum.enable();
4      const userProvider = new ethers.providers.Web3Provider(window.ethereum)
           ;
5      const userNetwork = await userProvider.getNetwork();
6      if (userNetwork.chainId !== 80001) throw new Error('Please switch to
           Mumbai Network for signing');
7      const signer = userProvider.getSigner();
8      const forwarder = createInstance(provider);
9      const from = await signer.getAddress();
10     if (from !== currentUser.address) throw new Error('Please switch to
           your account');
11     const data = token.interface.encodeFunctionData('transfer', ["0
           x40CDDe2b9BC7659C3349574Ec53db3B2bd9519BF", ethers.utils.parseEther(
           value.toString())]);
12     const to = token.address;
13
14     const request = await signMetaTxRequest(signer.provider, forwarder, {
           to, from, data });
15     if (action === "Reserve"){
16         console.log('Sending reserve transaction...', request);
17         return await UserService.reserveVehicle(plate, currentUser.username
               , request).catch(function(e){
18             console.log(e);
```

```
        });
    }else if (action === "Release"){
        console.log('Sending releasing transaction...', request);
        return await UserService.releaseVehicle(plate, currentUser.username
            , request).catch(function(e){
             console.log(e);
        });
    }
    return "Error";
}
```

```
const EIP712Domain = [
    { name: 'name', type: 'string' },
    { name: 'version', type: 'string' },
    { name: 'chainId', type: 'uint256' },
    { name: 'verifyingContract', type: 'address' }
];
const ForwardRequest = [
    { name: 'from', type: 'address' },
    { name: 'to', type: 'address' },
    { name: 'value', type: 'uint256' },
    { name: 'gas', type: 'uint256' },
    { name: 'nonce', type: 'uint256' },
    { name: 'data', type: 'bytes' },
];
function getMetaTxTypeData(chainId, verifyingContract) {
    return {
        types: {
            EIP712Domain,
            ForwardRequest,
        },
        domain: {
            name: 'MinimalForwarder',
            version: '0.0.1',
            chainId,
            verifyingContract,
        },
        primaryType: 'ForwardRequest',
    }
};
async function signTypedData(signer, from, data) {
    const isHardhat = data.domain.chainId === 31337;
    const [method, argData] = isHardhat
        ? ['eth_signTypedData', data]
        : ['eth_signTypedData_v4', JSON.stringify(data)]
    return await signer.send(method, [from, argData]);
}

async function buildRequest(forwarder, input) {
    const nonce = await forwarder.getNonce(input.from).then(nonce => nonce.
        toString());
    return { value: 0, gas: 1e6, nonce, ...input };
}
async function buildTypedData(forwarder, request) {
    const chainId = await forwarder.provider.getNetwork().then(n => n.
```

```
        chainId);
    const typeData = getMetaTxTypeData(chainId, forwarder.address);
    return { ...typeData, message: request };
}
async function signMetaTxRequest(signer, forwarder, input) {
    const request = await buildRequest(forwarder, input);
    const toSign = await buildTypedData(forwarder, request);
    const signature = await signTypedData(signer, input.from, toSign);
    return { signature, request };
}
```

```
import { ethers } from 'ethers';
import { MinimalForwarder as address } from '../deploy.json';
const abi = [...];
export function createInstance(provider) {
    return new ethers.Contract(address, abi, provider);
}
```

Note: The abi has been removed for longitude purposes. an example of an ABI can be found in Appendix E