UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

Development of a DAPP

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

Gerard Castell Ferreres

In partial fulfillment
of the requirements for the master in
Advanced Telecommunication Technologies **ENGINEERING**

Advisor: Jose Luis Muñoz Tapia
Barcelona, October 20, 2023

Contents

List of Figures

List of Tables

Acronyms

ABI Application Binary Interface. 26, 70

ETH Ether. 18–20, 22, 25, 27–31, 46, 50, 53, 54, 70

EVM Ethereum Virtual Machine. 18–23, 26, 27, 31, 70

PoS Proof of Stake. 16, 23, 57, 66, 67, 69

PoW Proof of Work. 16, 57, 63, 66, 67

UI User Interface. 12

Revision history and approval record

Revision	Date	Purpose
0	01/09/2023	Document creation
1	18/10/2023	Document revision

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Gerard Castell Ferreres	gerardcastell97@gmail.com
Jose Luis Muñoz Tapia	jose.munoz@entel.upc.edu

Written by:		Reviewed and approved by:	
Date	01/09/2023	Date	18/10/2023
Name	Gerard Castell Ferres	Name	Jose Luis Muñoz Tapia
Position	Project Author	Position	Project Supervisor

Abstract

This thesis explores the development of a Decentralized Application (dapp). The fast growth of blockchain technology has provided an opportunity to revolutionize lots of industries by enhancing transparency, security, and accessibility. This research investigates the architectural design, implementation, and deployment of a dapp that leverages smart contracts to facilitate car insurance policy creation and management.

Key objectives include the development of a user-friendly interface for policyholders, where they can obtain policy proposals and when purchased ensure a secure, tamper-resistant policy ledger. The study assesses the technical and practical aspects of blockchain-based car insurance, emphasizing the advantages of decentralization in reducing fraud, optimizing administrative processes, and enhancing policyholder trust.

The findings highlight the potential of dapps to disrupt traditional insurance models, emphasizing their role in encouraging trust and transparency while streamlining policy management processes.

1 Introduction

In today's highly digitized world, everyone and everything is connected and nearly every routine task we carry out relies on a piece of technology to be accomplished. Unfortunately, many of these processes are complex, have a lack of transparency or involve intermediaries which makes them slow and tedious. Companies and third parties have in their hands the governance of our information, which currently is the most powerful asset.

The Internet plays a crucial role in these interactions since it has become the main connectivity channel. However, it is characterized by a fundamental centralization of data, power, and control. Corporations have monopolized user data, platform access, and content distribution, raising concerns about privacy, censorship, and fairness. In response to these challenges, the next generation of the Internet, well-known as Web 3.0, along with decentralized applications (dapps) have emerged in order to offer a more decentralized, open, and user-centric digital experience.

The emergence of Web 3.0 and dapps represents a huge shift in the way we interact with and perceive the digital world. This transformation is driven by a combination of technological innovation, a growing desire for transparency and user control, and a fundamental rethinking of the traditional web's limitations, which opens up new possibilities for innovation, collaboration, and trust on the internet, redefining the way we engage with the digital world.

1.1 Statement of purpose

Given the emergence of dapps, this thesis aims to figure out how a traditional digital product can adapt its tech stack to harness the capabilities and advantages provided. The main objective is to develop and implement a decentralized application for a real case scenario in order to understand how it achieves to make apps user-centric and learn from the bowels which are the protocols and software that have evolved to implement such applications.

1.1.1 What dapps offer over usual web applications?

Continuously, we need to interact with companies in a lot of scenarios such as checking where a delivery is in real-time, complaining about a service, canceling a purchase, making a reservation, and so on. For nearly all of those cases, we use their web application to carry out such communication. This way, the companies use their technology, such as web applications, to provide just the data that they want to their consumers. In this process, we should be concerned about two worrying topics:

1. **Data Ownership.** The information exchanged between consumers and companies is often sensitive and personal. When we interact with these corporate web applications, we must consider the extent to which our personal data is collected, utilized, and potentially monetized by these companies.
2. **Centralized Control and Trust.** The centralization of technology platforms in

the hands of companies introduces a reliance on trust. We entrust these entities with our data, often without direct insight into how it is managed and secured. Centralized systems are susceptible to data breaches or manipulation, which can leave consumers at a disadvantage.

Decentralized applications appear as an attempt to solve these issues. Dapps are software applications built over a decentralized network, usually blockchain technology such as smart contracts to automate and enforce rules within the application. Smart contracts are self-executing code that runs on a blockchain and automatically runs a set of predefined rules without the need for intermediaries. Thanks to these features, the dapps can address the data privacy and ownership problem:

- **User-controlled data.** Users are able to see the code that rules the smart contracts, what is to say, users can know what companies would be able to do before and after accepting the contract.
- **Blockchain security.** Data stored is cryptographically secured, making it really challenging for anyone to corrupt it. Hence, it ensures the integrity of user data.
- **Transparency.** The transparent nature of blockchain allows you to verify how your data is used. All data interactions, known as transactions, are recorded on the public ledger.

Besides that, dapps take advantage of blockchain consensus philosophy to erase the centralized control and trust problem:

- **Decentralization.** Dapps operate on decentralized networks, typically based on blockchain technology. This decentralization means there is no central authority or intermediary that holds control over the application or your data. Trust is distributed across the network rather than being concentrated in a single entity.
- **Trustless Environment.** In a dapp, trust is not vested in a central authority. Instead, trust is established through the consensus mechanisms of the blockchain. Transactions are verified by a network of nodes, making it more challenging for any single entity to manipulate data or transactions.
- **Censorship Resistance.** Dapps are resistant to censorship, ensuring that access to the application remains open and uncontrolled. The code that contains a smart contract is accessible to everyone always and cannot be changed so companies cannot make unilateral decisions if the contract does not allow it.

By releasing the management and ownership of data to the users, dapps provide a more user-centric, accessible, secure, and transparent model. They erase the need for trust in a single entity and instead share the trust across a network. Therefore, users leverage the power of Web 3.0, the power to own their data on the Internet.

1.1.2 Business case: Car insurance

Once the benefits of dapps are introduced, we may think about a huge amount of real scenarios where this technology could improve significantly the user experience. For this

experiment, I have chosen a regular task that almost everyone has to do at least once in his life: contract car insurance.

The world of insurance has long been characterized by cumbersome policies, complex intermediaries, and, at times, a lack of transparency. However, the advent of blockchain technology has presented an opportunity to disrupt and revolutionize this industry, offering a new era of decentralized insurance applications. In this thesis, I explore the development and implications of a dapp designed for managing car insurance policies.

As we look into the idea of building this dapp, there are a lot of processes that can be automated and trusted in the smart contract such as policy cancellation with corresponding payment refund, claims processing with third parties or policy renewal. I will compare how these regular procedures are streamlined thanks to the dapp approach and how it leaves behind the traditional insurance model.

The motivation of this thesis is to highlight the contribution to the ongoing discourse about the transformation of the Internet through blockchain technology. By focusing on car insurance, an area with a wide audience, I aim to demonstrate the potential of dapps, not just as technological innovations but as a game changer for an industry that, for too long, has been defined by complexity, opaqueness, and a lack of understanding with intermediaries.

1.2 Requirements and specifications

In order to evaluate properly if the use of dapps truly can replace the current model I have developed an end-to-end scenario. To do so, the experiment requires the development of a frontend and a backend that shapes the whole dapp.

The frontend part consists of a web application in order to make the dapp accessible to the policyholders. This platform acts as a bridge to offer the users the features of the backend.

On the other side, the backend is divided into two parts. One consists of developing the smart contracts that are in charge of handling the policy business logic. In addition, I have developed a traditional backend server with a database to store all policy proposals before being purchased.

This whole scenario runs locally on my computer through a tool that simulates the blockchain network. Therefore, smart contracts can be widely tested and executed without spending real money.

Although the whole experiment is developed on a standard laptop, there is a deep dive into the hardware requirements for deploying the dapp comparing the local and production scenarios in the appendix A.

1.3 Work plan

1.3.1 Tasks and milestones

The development of this project has been divided into three stages: development of the

smart contracts, development of the off-blockchain backend and database, and finally development of the frontend web application.

The first step was to implement the smart contracts. To do so, I took a course about Ethereum, Solidity, and smart contracts development. Once I acquired the basis of these technologies, I was able to start coding the contracts for policy creation and management. At this point, the first milestone was achieved: Smart contracts were coded (*Feature 1*) and I was able to deploy them locally and perform several operations:

- The contracts were ready to receive transactions to create or cancel policies, which refunds instantly to the user.
- We could retrieve the policy data just with the company and the client accounts.
- We could report a claim as a client, and an external claim evaluator could approve or decline. If approved, it automatically pays the costs of the sinister.
- Policies can perform all actions while they are within the activation period. Once the end date arrives, the user has to pay the renewal to reactive the policy and continue enjoying the policy coverage.

Once the smart contracts were developed, I took another course about how to build a backend server with a database. Afterward, I was able to develop the backend app that allowed me to generate the proposals that could be turned into policies on the blockchain:

- The app was able to, given car specifications and driver data, offer a bunch of coverage types with the associated pricing per month (*Feature 2*).
- When the user wants to save a proposal with the selected coverage types it must be authenticated through his Ethereum address, this way, the proposals generated could be bound to the account and saved within the database (*Feature 3*).
- Once the user purchases a proposal, the address of the policy smart contract generated can be sent to the backend to identify which proposals have been finally converted (*Feature 4*).

Eventually, I developed the frontend application, where I could combine the app interface with both, the backend and the smart contracts and give sense to the whole decentralized app. These were the features released:

- The user deals with an interface where it can be logged with his Ethereum account (*Feature 5*).
- The user can introduce his car specifications to obtain a list of coverage types with their price. He can select the desired configuration to include in the policy and generate a proposal (*Feature 6*).
- The user can access a dashboard where he can see the stored proposals. The proposal view displays all data stored in the database and served through the backend. The user can purchase the policy through his wallet (*Feature 7*).
- The user can access the dashboard with all policies purchased. At every policy page,

all data is retrieved from the smart contracts. Users can cancel the policy and the proportional part of the premium paid (*Feature 8*).

- The last feature, which I did not have enough time to finish, was to connect the renewal and claim features from the UI app to the smart contract (*Feature 9*).

1.3.2 Gantt Diagram

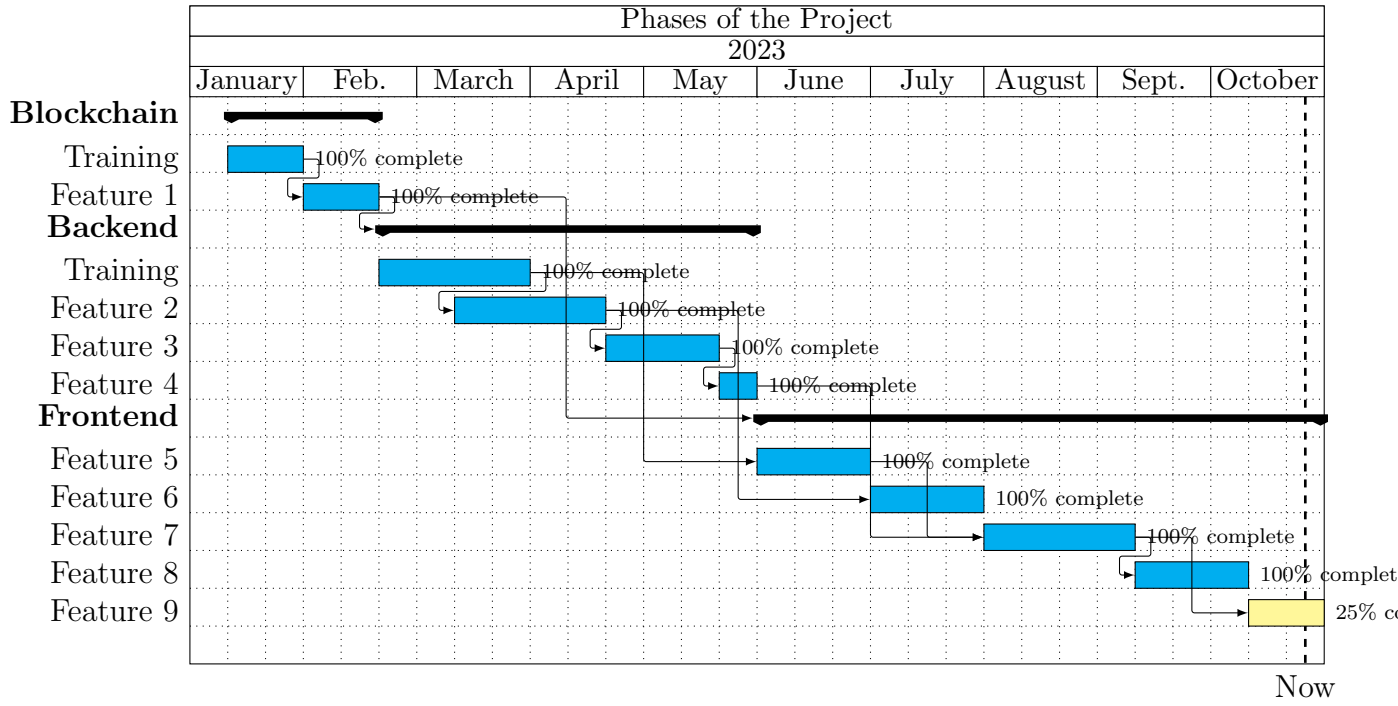

Figure 1: Gantt diagram of the project

1.3.3 Deviations from the initial plan

There were two features that I did not have enough time to implement from the frontend web application even though the smart contracts were ready to be consumed:

- The first one is the policy renewal. The renewal logic on the smart contracts was implemented in the first stage. However, there was not any backend logic implemented to propose a renewal premium taking into account the claims amount and the depreciation of the vehicle, which is a more realistic feature. Therefore, I consider it an incomplete feature without the backend part which I had no time to develop.
- The other one is the report of a claim. A sinister must have a third-party evaluator who qualifies the level of the incident and approves or declines the amount to refund. As I just have the client web app, I consider I would have to implement another app for the claim evaluators that also accesses the smart contracts and visualizes a list of

claims. There, they can decide if the claim is approved or declined. This would be a great example of how smart contracts streamline communication with intermediaries since once the claim is approved, the smart contract immediately compensates the client with the appropriate amount which is implemented. Unfortunately, I consider developing this second web application requires too much extra time and the features cannot be closed.

2 State of the art of the technology used or applied in this thesis

2.1 Blockchain

A blockchain is a way of building a distributed ledger where the transactions that modify the ledger's state are sequentially and immutably ordered by consensus. Therefore, Blockchain allows us to create a shared system to record what has happened. What has been recorded cannot be changed: it acts like a digital notary providing trust among parties. Note also that the information is shared because is built by different people, which is to say it is transparent and public. In addition, Blockchain technology, mainly the one based on smart contracts provides an extremely wide capability of programming our money.

In a blockchain, there are three main concepts: blocks, chains, and nodes. Block refers to data and state being stored in sequential groups of these blocks. To modify this state we have to perform a transaction of data, which needs to be added to a block to be finalized successfully. Furthermore, "chain" refers to each block referencing its parent. The data in each block cannot change without changing all subsequent blocks and would require the consensus of the whole network. Finally, the nodes refer to the computers which shape the network. They must agree upon each new block and the chain as a totality.

You can hear about a lot of benefits of using this technology such as data integrity, anonymity, reliability, etc. If we think about how it achieves all these properties we have to consider a combination of several ideas. Nodes are in charge of sharing the same state for everyone in the network. To achieve this, they negotiate through the consensus algorithm.

2.1.1 Source authentication, data integrity, and non-repudiation

We can start with how blockchain guarantees authentication and integrity. The public key cryptography plays a crucial role here. This algorithm is based on obtaining two keys, one public which is known by everybody and belongs to the user, and another private, which is known just by the owner.

The private one is used to cipher a message, whereas the public one is used to decipher the message. This way, where someone decyphers the message with the public key it can verify that the message belongs to a certain owner since only he has the private key to do so, hence we get the **source authentication**.

Moreover, this data already signed cannot be modified, because I would need the private key to decipher and cypher again the message but the people just know the public key. So, just the private key owner can do this and we have achieved **data integrity** as well.

Note also that consequently, we are obtaining a very interesting property known as "**non-repudiation**", which merely refers to a situation where a statement's author cannot dispute its authorship.

2.1.2 One-way hash functions

Once we know how to protect data we have to think about the blockchain network. The objective of this network is to store data. This data can only be modified through transactions. Somehow, we are talking about a database. There is a basic requirement for a database which is a fast search capability. To achieve a great performance, it is applied one-way hash functions to the inputs, so the outputs are used as keys of the database. To decrease the capacity of that two different inputs generate a collision, i.e. generate the same hash, we are going to use really huge output space with 256 bits. Therefore, thanks to "one-way hash functions", given a hash output, it is computationally unfeasible to find the input. More details about hashing are explained more deeply in appendix B.

Figure 2: One way functions.

2.1.3 Transactions and consensus

Most of the ledgers that we usually use are centralized. All data is stored in a single computer and that makes it simple and it could be fast or convenient for an end user too. However, centralization has a drawback to consider: the power is centralized as well, and that is a risk if you cannot trust the owner. Banks are the typical example of this, but blockchain is attempting to offer a better solution. To do so, it relies on distributed systems, which are models in which components located on networked computers, **nodes**, communicate and coordinate their actions by passing messages. They have separated memories consolidating a shared state, where the ledger can be stored.

Now we are able to introduce how the ledger handles money. Users of the blockchain can create money movements, called **transactions**. All transactions are authenticated by their owners, even though we need **consensus**. The consensus is in charge of providing an order. This is fundamental in a ledger with the aim of avoiding negative balances. In a distributed system the consensus plays a main role since it has to ensure that the network avoids stuck states, stops if it is asynchronous, and progresses when the network recovers. Note that this means to force determinism, so external sources can just be injected as a transaction.

Another interesting property of money is to keep anonymity. Bank transfers are directly bound to the implied persons, whilst pocket money is not, so current digital transactions are not keeping this property. Blockchain pursues to resolve this too by creating an account ID that cannot be linked to a person. Users create a pair of public and private keys, sometimes the address is a derivation of the public key, and use that to sign the transactions on the network. Hence, everyone can have more than one account, and public

keys have absolutely not any relation to the person's ID.

Every blockchain has a different way of achieving consensus, in the case of Ethereum was accomplished initially thanks to Proof of Work (PoW) which was introduced by Bitcoin. Nevertheless, due to several reasons, Ethereum eventually migrated to Proof of Stake (PoS) in 2022, which nowadays is a more common alternative and is increasing in popularity.

Both Proof of Work and Proof of Stake are complex algorithms that are explained thoroughly in appendices C and D respectively. It is a recommended reading in order to understand better the sense of consensus.

2.1.4 Cryptocurrencies

A cryptocurrency is an asset to exchange secured and trusted by a blockchain ledger. It represents a medium of exchange that is accepted as payment for goods and services. Blockchain allows users to make transactions over the network that would remain registered in the ledger without needing a third party to maintain them.

These cryptocurrencies are what the miners receive in compensation for the effort of mining new blocks and behaving honestly in the blockchain. This is the usual way the blockchain creates more tokens.

The first cryptocurrency was Bitcoin, created by Satoshi Nakamoto[10]. Nowadays, thousands of blockchains and cryptocurrencies have appeared and evolved from Bitcoin.

2.1.5 Wallets

If a blockchain aims to process payments, every user will expect some software or hardware to interact with the network. Since there are many non-trivial concepts to take into account when interacting with the network, such a technology has to fit several requirements, some of which are the following:

- Show account balance, even in different currencies and blockchains.
- Manage multiple accounts.
- Manage user private keys: generate, backup, and so on.
- Send transactions and display transactions history.

This item is what a **wallet** does.

Since wallets can manage a handful of accounts, and each one has its key pairs, they end up with a simple idea: Remember a single secret, which is the master, and derive all the key pairs from it. These types of wallets are called **Hierarchical Deterministic Wallets** [11], or HD wallets.

HD wallets ensure that all addresses needed are generated from a single seed instead of being randomly generated on demand, as you may see in figure 3. The seed is expressed in the form of a twelve-word seed-word phrase called **mnemonic**. Different public keys of the same HD wallet cannot be correlated unless you know the seed.

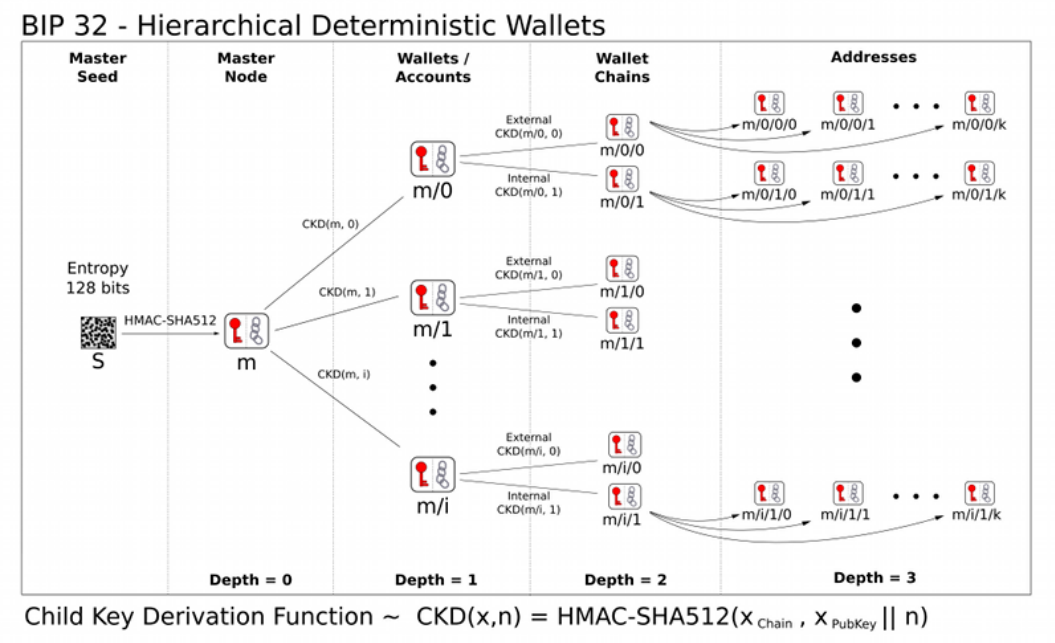

Figure 3: Hierarchical deterministic wallet.

One of the most popular software wallets is Metamask [12]. Metamask is an HD wallet built as a browser extension that provides access to blockchain networks. When a user installs Metamask, it generates a random mnemonic and prompts the user to save it. Then, Metamask asks the user to introduce a password that will be used to encrypt the master secret and store it in the browser's local storage, which is known as *vault*.

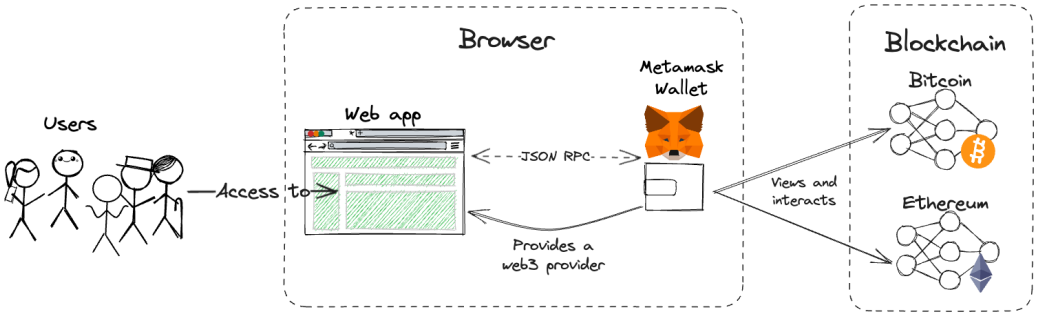

Figure 4: Metamask as web3 provider.

As is illustrated in figure 4, Metamask is designed to be a browser wallet. Wallets act as **web3 provider** to connect the different blockchains. The analogy is to think about this web3 provider as a telecom company offering you access to the cell network. Without such a provider, web applications would not be able to access the blockchains. This communication is carried out through the JSON-RPC protocol.

2.2 Ethereum

Ethereum motivation was to leverage the first blockchain networks, such as Bitcoin, and overcome their constraints. In a nutshell, Ethereum is a blockchain that works like a computer embedded, so is prepared to run apps and organizations in a decentralized, permissionless and censorship-resistant way.

In Ethereum, we can talk about a single computer called Ethereum Virtual Machine (EVM) that establishes which is the state that every node on the network should agree on. Any participant can broadcast a request for EVM to perform a computation over the state. When it happens, other participants on the network validate and ensure the request is signed by the address that sends it. If verified, then they check if that address has permission to execute the state change, if so they do it. Consequently, the state changes in the EVM and propagates throughout the network. Such requests for computations are called transaction requests and are stored on the blockchain by order along with the current state of the EVM.

A "node" is any instance of Ethereum client software that is connected to other computers also running Ethereum software, creating a network. To read more information about the Ethereum client you can find it in appendix F.

Ethereum gathers a lot of concepts and the core ones are explained in the following sections. For more information, you can start by reading the original Ethereum Whitepaper[21].

2.2.1 Ethers

Ether (ETH) is the native cryptocurrency of Ethereum and its purpose is to boost a market for computation. ETH represents an economic incentive for participants to provide computational resources to the network, which allows transactions to be validated and executed.

In Ethereum, when users want to make a transaction, they must pay some amount of ETH to the network as a bounty. These usage costs are known as gas fees. The network will award to the participant that eventually verifies the transaction, executes it and commits it to the blockchain.

This "tip" that senders pay, is equivalent to the amount required by the resources to carry out the computation and depends on the demand of computation power at that moment. Furthermore, paying an amount also prevents malicious participants from blocking the network by requesting infinite computation or other intensive tasks since they must pay for computation resources. If a task runs out of all the ether fees paid, the transaction gets terminated and the network returns to normal.

Ether can just be created on the creation of a new block process and is performed by the protocol definition, so any user can create. About 1/8 of the total issuance goes to the block proposer, whilst the remainder is distributed across the other validators. By contrast, Ether is destroyed on every transaction. This process is called "burning" and the removal is permanent. When users pay for their transactions a base gas fee is destroyed.

This amount is defined by the blockchain according to the computation power demand.

The EVM stores the state, and this state is a ledger of all the accounts that have executed a transaction on the network. The final state provides us the current balance in ETH of all the accounts. Such accounts are able to send, hold and receive ETH from other accounts and interact with deployed smart contracts. We may find two types of accounts:

- Externally-owned accounts (EOA), controlled by a person with the private key.
- Contract account, which is a smart contract deployed to the network and controlled by the code.

2.2.2 Gas

Gas refers to the unit that measures the amount of computational effort required to execute specific operations on the Ethereum network. Each transaction to the network requires computation resources to execute. Such resources must be paid for to ensure Ethereum cannot get stuck in infinite computational loops. Note that regardless the transaction fails or succeeds the computation could have been carried out so the fee is paid in any case. Therefore, this gas fee appears to justify the payment for computation as is reflected in figure 5.

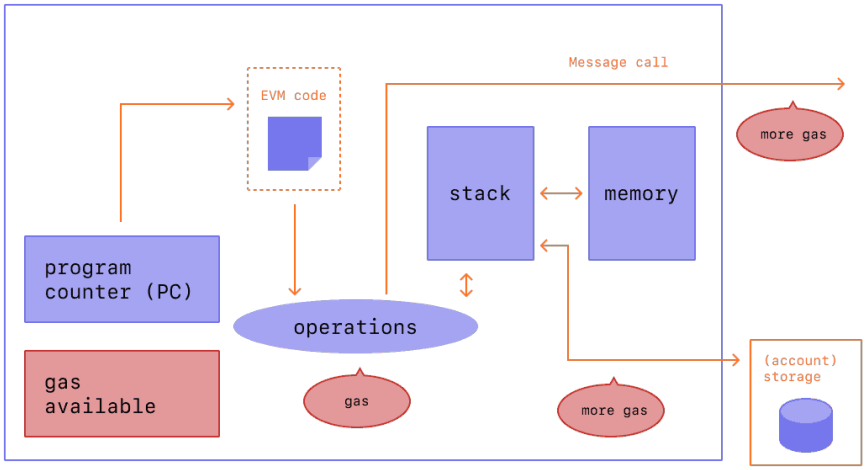

Figure 5: Gas consumed per computation.

If we want to calculate the gas fee that a transaction requires we have to define it as the amount of gas used to do some operation, multiplied by the cost per unit gas. The EVM specifies the units of gas required by each computational operation [22]. On the one hand, the base fee is set by the protocol depending on the number of blocks before it so it is easily predictable for users. On the other hand, the priority fee incentivizes validators to include a transaction on the block. The minimum amount of gas to pay is the base fee, but without a tip validators most likely will choose other transactions over yours or even create blocks with no transactions since they would receive the same award. So it

is important to make this gas estimation predictable so users that offer too much waste some ETH or, by contrast, users that offer too little will not get a validator that chooses the transaction.

Gas fees have to be paid in ETH and it is divided in two components: the **base fee** and the **priority fee** or tip.

It is important to mention that users can define the maximum limit they are willing to pay for their transactions. This parameter is known as *maxFeePerGas* and it must be greater than the sum of the base fee and the tip. The transaction sender is refunded with the difference between the max fee and the sum of the base fee and tip. This parameter is crucial because gas fees can get so high due to too much computation power demand or even complex smart contracts apps doing lots of operations.

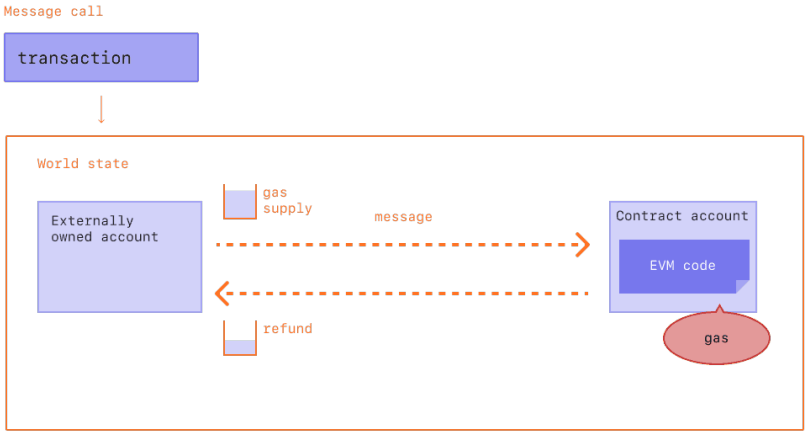

Figure 6: Gas refund procedure.

In brief, gas fees help to keep the Ethereum network secure. By requiring these bounties for computation it prevents bad actors from clogging the network. In addition, it also helps to avoid accidental loops or other intensive computational wastage in code.

2.2.3 Ethereum Virtual Machine (EVM)

The Ethereum Virtual Machine is present at any block in the Ethereum's chain and it is in charge of defining the rules for computing a new valid state from block to block.

Ethereum is in essence a **distributed state machine**, in which state is a large database that holds not only all accounts and their respective balances but also a state machine. This state machine can execute arbitrary machine code and change between blocks according to a pre-defined set of rules defined by the EVM.

The state data structure that contains the large database is called a *modified Merkle Patricia Trie*, which points to all addresses by hashes based on a single root hash such

as the Merkle tree idea explained thoroughly in appendix E. This system lets to verify transactions in the network really fast and without the need for knowing all the content.

Ethereum is like a **state transition function** where given an input, the EVM returns a deterministic output. In terms of state, the expression would be $Y(S,T) = S'$, where S is a valid old state, T a handful of new valid transactions, $Y(S,T)$ is the transition function and S' is the new valid state.

Transactions are instructions from accounts signed beforehand which can be classified into two types: **message calls** and **contract creation**. The last type results in the creation of a new contract account containing compiled smart contract bytecode. Whenever a message call is addressed to this contract account the bytecode is executed.

EVM contains several pieces as you may see in figure 8. One of them is a **memory** which lives during execution by is no persisted between transactions. Nevertheless, contracts contain also a Merkle Patricia **storage** associated with the contract account as part of the global state.

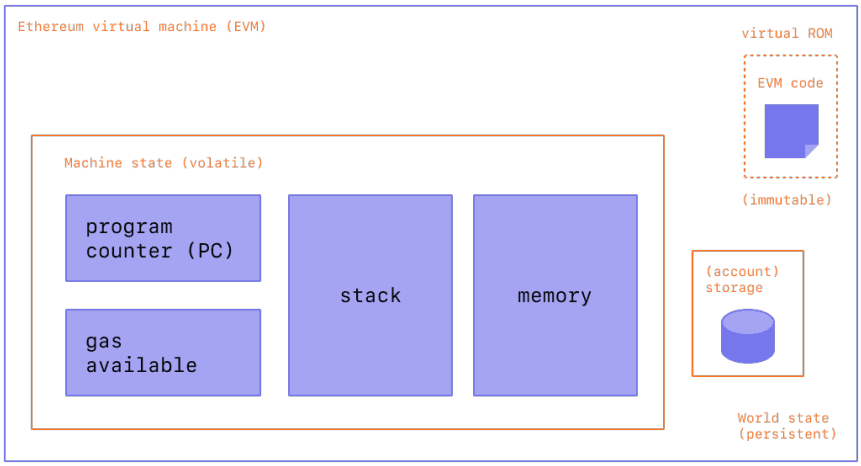

Figure 7: Diagram of EVM internal components.

Compiled smart contracts can perform standard stack operations such as *XOR, AND, ADD or SUB*. In addition, EVM also implements some operations for blockchain purposes like *ADDRESS, BALANCE, BLOCKHASH*, etc.

2.2.4 Smart Contract

At this point, the power and potential of Ethereum is quite clear. The capability to add executable code within the state machine streamlines many recurrent tasks to the extent that developers no longer need to write new code each time they want to request a computation to the EVM. Instead, they can upload programs into the EVM state and make a request to them in order to execute those lines of code which may be configurable with varying parameters of the call. Those scripts uploaded and executed by the network are smart contracts.

One of the most explained analogies about a smart contract is a vendor machine. In a vendor machine, if you introduce correct inputs (money and snack selection), a certain output is guaranteed (snack is dispensed).

Deploying smart contracts is public. To do so, we just need to code a smart contract with some Ethereum developer language such as Solidity, the one chosen for this experiment. Such languages are compiled before being deployed so the EVM can understand the bytecode. Moreover, we need some ETH since deploy means to request a transaction, so we have to pay gas fees.

It is important to highlight a big constraint. They cannot get information about real-world events because the state machine needs to be deterministic and such events are happening off-blockchain. If we inject external data in the middle of a computation it will pollute security and decentralization so determinism could not be achieved. Likewise, using such external data really enhances the scope and value of smart contracts. So to solve it, data is introduced and stored in the blockchain by what is known as **oracles**. Once the data is recorded on the blockchain it can be consumed by smart contracts, but cannot be changed which ensures determinism is achieved.

In a nutshell, smart contracts are simply programs that run on the Ethereum blockchain. They can perform a set of actions, their functions, and handle data, their state, that is located to an Ethereum address, hence they have a balance and can receive and send transactions to any other address. This allows to call of other smart contracts that encourage composability.

2.2.5 Decentralized Applications (dapps)

What really empowers Web 3.0 are the decentralized applications. To define what is a decentralized application, we need to gather all the aforementioned concepts. A dapp is an application built on a decentralized network that combines smart contracts and a frontend user interface. Ethereum defines every smart contract as public, meaning transparent and accessible, hence any frontend application can interact with any smart contract, acting as an open API, even if you are not the owner of the smart contract.

If we think about how standard apps are implemented, we may define two isolated domains: frontend and backend. In the case of a dapp, the backend code runs on a decentralized peer-to-peer network like a blockchain instead of a centralized server. About the frontend, it could be written in any code like the apps do. If we focus on the hosting, the backend code is also hosted in the blockchain, whilst the frontend is not required, so it can be served from any centralized server. In practice, developers use web libraries such as "*ethers.js*"[14] which will trigger and interact with the browser wallets like **Metamask**. Such wallets use **JSON RPC** protocol to connect through a public node, such as "*infura*"[15] or *"geth"*[16], to Ethereum and therefore, to the smart contract. This tech stack illustrated in figure 8 is what conforms to a decentralized application.

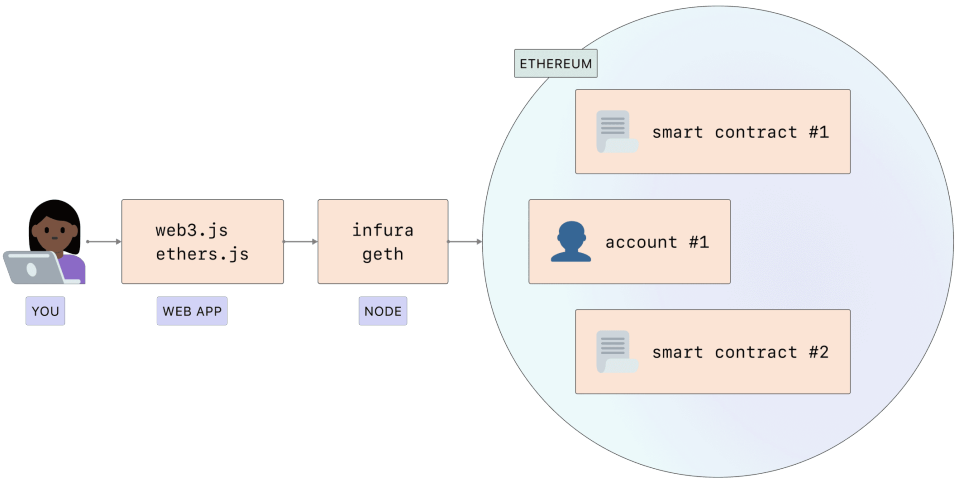

Figure 8: Ethereum connection from web applications

Dapps rely on their business logic in the network through the smart contracts, which run computations on the Ethereum Virtual Machine. Such a code is always deterministic, meaning that given the same network state and the same input transaction, it is going to return the same output. It is also important to note that what is sent to the network cannot be changed so we must be very careful about the code that we write. It has bugs such as infinite loops or exhaustive processes that waste so much gas or even in the worst case security glitches, we could be able to update the code to fix them, so they must be designed and tested thoroughly. Opting for dapps provides some interesting benefits over usual applications but also has drawbacks that we must take into account.

On the one hand, one of the most important advantages is the **zero downtime**. Since the code is deployed as a smart contract on the blockchain, the network will always be able to serve client requests so the service is never suspended. The blockchain wrapping also offers more **privacy** because the users will use address accounts to interact that are not linked to persons. In addition, dapps ensure **data integrity** on the blockchain. Such data is immutable and malicious actors cannot corrupt persisted information. Eventually, the more significant game changer properties are **censorship resistance** and **trustless behavior**. Smart contracts can be analyzed and predict what they will compute and return in a deterministic way, which completely replaces the need for trust in third parties with opaque processes. Moreover, no single entity can unilaterally block users from submitting transactions to change or read the state. This does not mean that we cannot define rules that allow just certain accounts to perform some actions but it must be coded in the moment of deployment.

On the other hand, we may find some disadvantages to take into account, starting with **maintenance**, which requires complex solutions since we cannot modify the code once deployed. There is also a huge **performance overhead** because Ethereum demands that every node runs and stores each transaction and, in addition, Proof of Stake takes its time as well. Furthermore, there is a possibility of the network getting congested since it

can only process 10-15 transactions per second, if the rate of transactions is greater than the capacity to process them or the computation required increases the mempools will be full of unconfirmed transactions. Eventually, if we take a look at the user experience consuming the dapp it can be penalized as well. Most end-users might find it difficult to understand the Ethereum basis and use this kind of tools that really leverages blockchain power. In favor of reducing this effect, software solutions built on top of the base layer of Ethereum might end up trusting part of the logic and data managed in centralized services which could eliminate partially the benefits of blockchain.

There is more detail about the Web evolution until the Web 3.0 model in the appendix H.

3 Project development

In order to develop the insurance dapp, the process is divided into two parts as explained previously: backend and frontend.

3.1 Backend partition

For the backend, we will talk about the cohabitation of a traditional centralized backend app with smart contracts on the blockchain. This is due to the business model selected for the experiment. In an effort to represent a realistic scenario we have to note that all data stored in the blockchain will suppose a transaction, hence it will consume ETH, i.e. money. This raises the first decision of the project: select which data have to be stored on the blockchain in order to not exclude all its benefits. When thinking about insurance, we have to keep in mind a mental model about the flow a user carries out in order to contract a policy in the off-blockchain model:

1. User accesses a web platform where he is asked for his car specifications.
2. Such information is sent to the backend as a "quote". The backend executes some computations to generate pricing according to the risk assumed.
3. Backend returns to frontend app a policy proposal that includes the coverage types available and their respective price per month.
4. The web app displays the pricing by coverage, so the user may decide if the price convinces him and select which coverage types to include in his proposal. If the proposal is attractive for the user he can save it in the backend.
5. Once the user decides to purchase the proposal, he introduces his credit card and confirms. Then, the payment proceeds at the backend, usually along with third-party entities that are specialists in bank interactions. If the whole process succeeds, the policy is bound to the user in a centralized database. The backend notifies the frontend that the process has been terminated successfully.
6. The frontend notifies the user as well and for the rest of the "life of the policy" the user will interact with the company through the web app to modify or read data from his policy. We will consider a couple of actions that a user can perform with an active policy:
    - Make a claim: The user suffers a sinister which is covered by his policy so he uses the web app to report it and claim a refund for the possible damages. The backend receives the claim and the claim evaluator analyzes the details. If proceeds, the evaluator notifies the insurance company, which pays the third-party payment intermediary to refund the client or externals involved. Otherwise, if the claim is declined the company notifies the user through the app directly and no refund is carried out.
    - Cancel the policy: The user decides to finish the policy unilaterally. The company gets notified and then it has to notify the third-party intermediary to refund the client.

7. If the end date of the policy arrives and the user has not canceled, the policy turns inactive and no operation can be performed to modify the state. The company offers a renewal proposal or not to the user with a new price taking into account depreciation and claims occurred. If the user accepts we go back to the first point.

We may identify a main entity in the process which is the **policy**. The policy is what defines the contract with the user so it would collect all the data related to what the user has paid, what has been insured and which action could perform. We can make the analogy of a user purchasing the proposal with a transaction to the smart contract because in both processes we are paying and amount to execute a computation. This computation results in storing the policy with its corresponding data.

Nevertheless, there is a previous process where the user is still not paying and requires computation, which is the **quote**. When the user quotes in the traditional model, the backend is computing an estimated price to assume the risk. If we translate this model to the blockchain world, we would have to pay for every possible quote. In terms of economics, this is absolutely not profitable since we do not know yet if the user will eventually purchase the policy. So even in the best scenario where the user confirms, we would have to increase the price considering a rough fee based on the quote per purchase ratio. If we observe usual businesses this purchase rate per price discovered, equivalent to a quote, is usually quite low. Therefore I have decided to avoid this computation on blockchain.

With the aim of overcoming this quote computation constraint, I designed a mixed on-blockchain/off-blockchain solution that relies on the quote computation and storage of the backend logic into a centralized server, whilst the policy management and storage keep in the smart contract in favor of blockchain advantages. Henceforth, we will consider centralized backend as **off-blockchain** and smart contract backend **on-blockchain**.

3.2 On-blockchain backend

I started developing the smart contract because it is in charge of managing the policy entity which is the core of the business case. To develop the smart contract I chose Solidity[4] as one of the most popular languages for writing smart contracts.

The code written with Solidity is compiled to generate at least two files: an Application Binary Interface (ABI) and a Bytecode. The bytecode is the compiled code of the smart contract at low-level and readable by the EVM, whilst the ABI is a JSON file that describes the smart contract functions and data structures and how to interact with them. Such ABI is going to be crucial for the frontend part so after deploying it, the web must be able to access this file in order to be able to prepare the transactions according to the smart contract's expected input. To read more about how is the smart contract code you will find more information in the appendix G.

3.2.1 Policy smart contract

Regarding the car insurance flow commented at 3.1, the policy smart contract is in charge of storing the data. To create this smart contract we need to collect the following data:

- *riskData*. A JSON object with the specific details of the drive and car covered.
- *premium*. The total amount of ETH paid by the user for the insurance service.
- *owner*. The account address of the owner of the policy, which references the user.
- *factoryAddress*. Sets the factory smart contract address. This value is not pre-defined for security reasons. We have to be able to propose new versions of the contracts.
- *renewalDate*. The date the user defines as the end date of the policy. On that date a renewal

Besides this data, there are some more properties that are settled on contract deployment. The first one is the *endDate*, which initially is equivalent to the renewal date. It will remain the same unless the user or company cancels the policy before the renewal date for some justified reason. The other property is the *startDate* to consider the policy effective, which is set to the block timestamp of the EVM that computes the operation. Eventually, it also stores an empty array of *claims*.

If we think about the functions we can group them into three types depending on the purpose. One of the more interesting is the *modifiers*. They look after to meet certain conditions and if not they terminate the transaction. These functions are used by other functions to check conditions before executing their code. An example from the experiment is a modifier that checks whether the sender of a transaction corresponds to the address of the insurance company. Then, the function **renew**, which renovates the policy, can just be executed by the insurance company address. Furthermore, there is also another modifier that ensures the transaction sender is the owner of the policy which is the client and it is used in the method to make claims. There are four modifiers in this smart contract:

- *onlyOwner*. A modifier to check if the sender is the owner of the policy.
- *onlyFactory*. A modifier to check if the sender is the factory smart contract.
- *onlyFactoryOrOwner*. Combining the previous two we obtain another one that ensures the send is either the owner or the insurance address.
- *isActive*. A modifier that ensures the policy end date is future to the execution time so it checks the policy to be active.

In addition, there are a handful of functions that are simple *getters* to retrieve the data of the contract. These getters are really useful for the web application to consume and display the information in a more understandable and fancy way. All the getters of the contract are protected with modifiers that ensure the data can just be called by the insurance company or the owner.

The remaining functions are the ones that perform the computations to modify the state. A brief explanation about then:

- *cancelPolicy*. It can just be executed if the policy is activated and sets the *endDate* to the block timestamp.

- *makeClaim*. It can be called by the owner if the policy is active. Adds a new claim data structure to the list.
- *approveClaim*. It can be called by the factory. Look for the claim in the list and set the expenses of the sinister. Then calls *resolveClaim* accepting the refund.
- *declineClaim*. It can be called by the factory. Calls *resolveClaim* declining the refund.
- *resolveClaim*. It is a private method. It stores on the claim the block timestamp and if it is accepted or declined.
- *renew*. It can just be executed by the factory. Updates the *endDate*, *renewalDate* and *premium* to the values suggested by the company and executed by the client.

Note that some of the functions do not make sense without the Factory contract explained in the section 3.2.2.

The entire code of this smart contract may be found in the appendix I.

3.2.2 Factory smart contract

Once the policy smart contract is defined we have translated the policy management on the blockchain. Nevertheless, the objective of the dapp is to offer a real car insurance service and to achieve this we need to put in the insurance company's shoes. As an insurance company, it pursues to convince thousands of clients, and each one of them can own more than one policy. That implies a smart contract deployed for every client policy. Every time a user enters the platform, he expects to visualize all his policies and be able to operate with them. To do so, the company needs to store all the smart contract addresses bound to each client. If that data is collected in a centralized server then the user ownership and censorship resistance get lost, so somehow that data should be saved in the blockchain.

Therefore, another smart contract is needed to collect all policy contracts deployed and the user to whom they belong. Fortunately, the smart contracts can be composable, therefore we can code make this smart contract, henceforth called **factory**, be the entry point of the clients to deploy their policy smart contract, i.e. create new policies.

In addition, the factory acts also as the payer since all the transactions to create policies will fetch it. This streamlines the way the insurance company transfers money and has better control of the overall balance.

If we evaluate this smart contract code, we may see how it completes the Policy smart contract. Since the section 3.2.1 explains in detail the basis of code this explanation is more straightforward.

Starting with the deployment, it is expected to exist just one version unless some fix or improvement has to be applied. In such a case, the new smart contract will replace this one partially or completely. Then, to create this contract we will need a minimum amount of ETH. To be realistic, every insurance needs an initial amount in the bank to start. This symbolic amount is used to get the ball rolling in order to create policies and be able to pay claims greater than the amount collected. This way the insurance economy can start

working. In addition, the sender of the deployment transaction is stored as the insurance company address, *insuranceAddress*.

The contract stores just the insurance company address and two mapping arrays:

- *policiesMapping*. Where the keys are the client addresses and the value is an array of their policy smart contract addresses.
- *claimEvaluators*. Where the keys are the evaluator account addresses and the value is a boolean that represents whether the evaluator is enabled or not.

We can find two modifiers in the code. *companyOnly* is used to validate that the sender is the insurance address. The other modifier is *claimEvaluatorOnly* which ensures the transaction sender is the company or an enabled evaluator.

Regarding the functions, the smart contract contains the following set:

- *addBudget*. It is the function to inject ETH when needed and just the company address is allowed.
- *withdrawBudget*. It is only actionable by the company and withdraws the amount received as a parameter.
- *createPolicy*. It receives the proposal data and the end date of the policy as parameters. These parameters are used along with the transaction ETH and the sender address, which is the policyholder, to deploy a new Policy smart contract. The resulting smart contract address is pushed to the list of policy addresses of the policyholder in the factory contract.
- *cancelPolicy*. It receives a policy smart contract address and checks if the transaction sender is the corresponding policyholder or the insurance address which are the only ones allowed. If so, the Policy Smart contract function *cancelPolicy* is called. Afterward, the factory computes the remaining time from the block timestamp until the end date and pays the proportional premium not enjoyed. Then, the refund is processed to the policyholder.
- *renewPolicy*. Ensures the just the policyholder is the sender and calls the specific policy contract *renew* method forwarding the new end date and premium received.
- *getPolicies* Retrieve the list of policy smart contract addresses stored bound to the transaction sender.
- *addEvaluator*. Adds an evaluator address to the list of claim evaluators. Just the company can do this.
- *changeEvaluatorValue*. Change the value of the evaluator address deciding if is enabled or not. Just the company can do this.
- *approveClaim*. It can be called just by an enabled evaluator or the insurance company. First of all, it checks if it has the necessary amount to pay the claim. If so, it calls the policy smart contract address received and calls the method *approveClaim*. Finally, it transfers the amount to the owner.

- *declineClaim*. It can be called just by an enabled evaluator or the company. If this is met, it calls the policy smart contract function *declineClaim*.
- *isClaimEvaluatorKnown* It received an address and it checks if it is an enabled claim evaluator.

As we can see from analyzing this smart contract, it is in charge of handling the balance and sending the ETH when required. It also acts as a facade for intermediaries like the claim evaluators, defining which are enabled or not to validate the truthfulness of the claims.

In a nutshell, when some action has to make a change in the state you must address the factory smart contract. It will update the policy smart contract state and execute the transactions when necessary. Moreover, the policy factory is restricted to the factory contract and policyholder use, whereas the factory smart contract deals with all possible consumers and restricts the actions accordingly to their domain. In figure 9, we may observe a map of the possible transaction directions that can carry out the policyholders, claim evaluators and the smart contracts between them. Note the transactions are not ordered chronologically and each arrow represents a transaction where the origin is the sender and it points to the receiver.

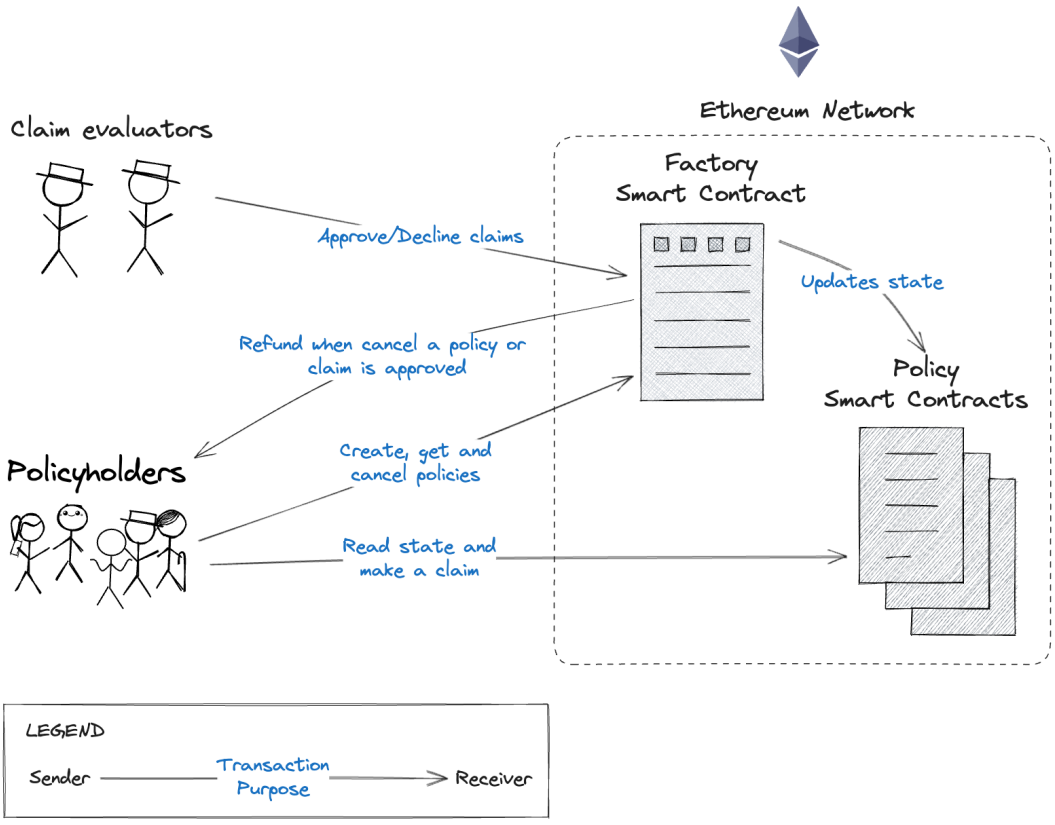

Figure 9: Factory and policy smart contracts transaction flow

The entire code of the Factory smart contract can be found in appendix J.

3.2.3 Ethereum developer environment

To develop and test smart contracts a developer uses what is well-known as **testnet**, short for test network. The primary public network is called **Mainnet** and testing smart contracts here would imply consuming Mainnet ETH on transactions, which requires real money. Therefore, testnets are really useful to test potential smart contracts in a production-like environment before deployment to Mainnet, so it is the analogy to production and staging servers. For the experiment, there were two possibilities:

- Use a local testnet, which requires selecting a feature that simulates the Ethereum network locally.
- Use a public testnet, which requires asking for testnet ETH to "faucet entities", which distribute it sometimes free but with daily limitations.

Since the public testnet option requires external dependencies to obtain ETH and the process can be challenging, I decided to use a local testnet through the tool **Hardhat**.

Hardhat is a tool that lets developers easily deploy contracts, run tests and debut Solidity code without dealing with live environments. Hardhat comes out-of-the-box with **Hardhat Network**[2]. It basically runs a process that implements the EVM and serves JSON-RPC and WebSocket requests. Moreover, it mines a block with each transaction that it receives so this really streamlines the developer experience. On server launch, it provides accounts with the public-private key pairs and a balance of 10.000 Hardhat ETH.

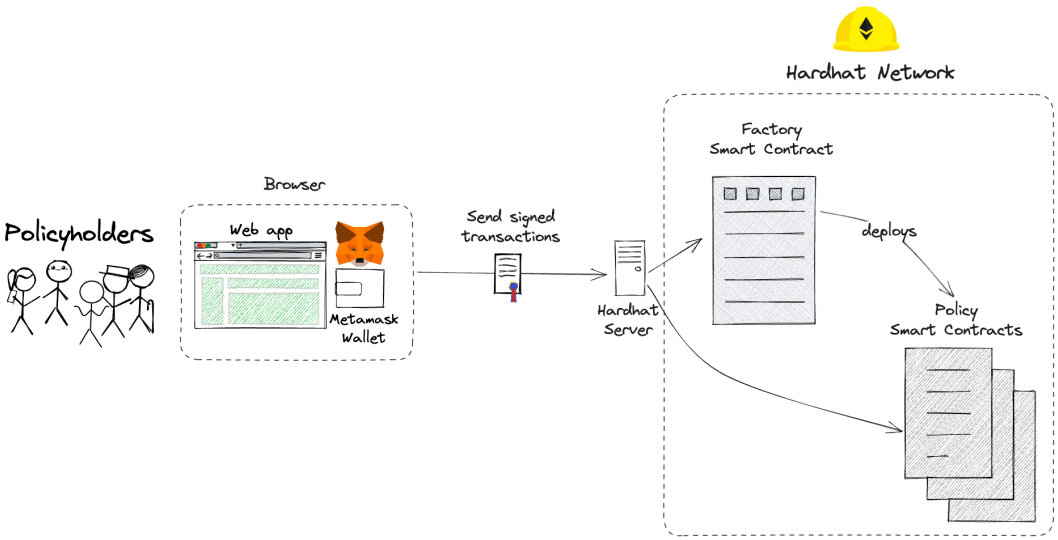

Figure 10: Web app consuming Hardhat network

In the experiment, the Hardhat network is served at a certain port, where the dapp will deploy the smart contract factory first, and then store the address of such contract into the web app application to make it accessible. As you may observe in figure 10, the web app and the Metamask wallet are configured to reach the factory smart contract and at this point, we are able to request transactions.

The wallet is in charge of asking for user confirmation on each transaction, so they reach the network signed with the policyholder address. Note that every policy contract deployed through the factory creation process will be in the Hardhat network as well, so they are reachable by the web app too.

3.2.4 Security consideration

There is an important security issue in this system. The data stored in the smart contracts is public so if it contains sensitive data anyone can see it. We can protect data making private or the getters with modifiers as explained above although, the miners can see all data in the nodes anyway. For our experiment, it does not represent an inconvenience but for a real case this problem could be solved by storing the data encrypted, for example using the public key of the user account to encrypt and the private key to decrypt. This is just a need to protect sensitive data but not necessary for changing the state since function executions can be protected, for instance by the address of the transaction sender.

3.3 Off-blockchain backend

The off-blockchain application consists of a backend app that handles proposal generation by providing pricing and storage of such proposals. There are many possible solutions to implement these requirements but the one selected for me is **NestJS**[3].

NestJS is a framework for building efficient, scalable Node.js server-side applications. It is developed with Javascript[1] and provides an out-of-the-box application architecture that allows developers to create highly testable, scalable, loosely coupled and easily maintainable applications.

Usually, a backend uses an *ORM*, Object-Relational mapping, which provides a way of mapping the data schemas with the backend models and eases the connection and comprehension. For the experiment, I selected **Prisma**, a new kind of ORM which apart from offering really comprehensive documentation and easy implementation with NestJS, recently has grown in popularity because of the way it declares just a data schema as a single source of truth and generates the database and backend types according to this agreement. For the experiment, a **PostgreSQL**[9] database has been selected, which is a powerful, open-source object-relational database more than enough for storing some proposals.

Thanks to these technologies, I was able to implement the two expected features.

3.3.1 Proposals module

The goal of the module is to offer insurance coverage and the corresponding price given a specific car and driver data. In addition, when the user likes the proposal he can save it to purchase it later on. To achieve this the app accepts several requests:

1. The first request is through the endpoint **POST "/api/proposals/quote"**. The body of this request basically consists of a risk subject and a risk object. The risk subject contains the driver data such as name, document number, and birth date,

whilst the risk object collects all detailed specifications about the car. Just a few of this data has been required in favor of the real scenario. Some of these fields are maker and model, plate, fuel type, etc. The response contains the list of coverage services available with their monthly price respectively.

2. Once the request wants to be saved, the app offers the endpoint **POST "/api/proposals/quote/save"**. The body for this case collects the same attributes as before, risk subject and risk object, but also the coverage configuration the user wants to include on his policy.

3. When a user wants to retrieve his saved proposals he can use the endpoint **GET "/api/proposals/quote"**. To develop this feature the authentication module has to be implemented, which is explained in the section 3.3.2.

With the previous endpoints, we enable the web app to ask for the data from the user and request the backend for a quote pricing. Once the backend responds, the coverage possibilities may be displayed to the user who then selects the coverage services that he would like to include. Eventually, if the proposal sounds good to him, he could save the proposal. The following step was to implement the authentication module to bind the proposals to the clients.

3.3.2 Authentication module

The authentication flow proposed for the dapp is a little bit different from the standard models since it uses the client wallets instead of a social media login or the usual email/password combination to identify a user.

Since we are using blockchain and we can assume each of our clients has an address, we can leverage these authentication credentials to identify them in our backend. The concept is called **nonce-based authentication** and is quite simple:

1. A user wants to be authenticated so he asks the backend to generate a nonce. The backend app generates the nonce and stores it for some time. Theoretically, this nonce cannot be generated again so each time is requested the app generates a different one. To generate this nonce it was used the library *Siwe, Sign in with Ethereum*[23].

2. Then, the user receives the nonce and signs it using his private key. Once signed he sends to backend the signed message along with the public key.

3. Backend verifies that the message has been signed with the public key received and it contains the nonce provided previously, so it guarantees that the sender is the owner of the private key, the unique capable of signing the message, hence it is authenticated. The nonce gets removed from the backend storage and a token for authenticating the account is returned to the user in order to attach it to the headers and access protected endpoints.

This behavior is achieved with two endpoints: **GET "/api/auth/generate-nonce"** and **POST "/api/auth/authenticate"** as you may observe in figure 11. At this point, the backend app can protect the endpoint for saving proposals, so the proposals will be always

sent with an authentication header including the token that identifies the user and we can bind the proposal to him.

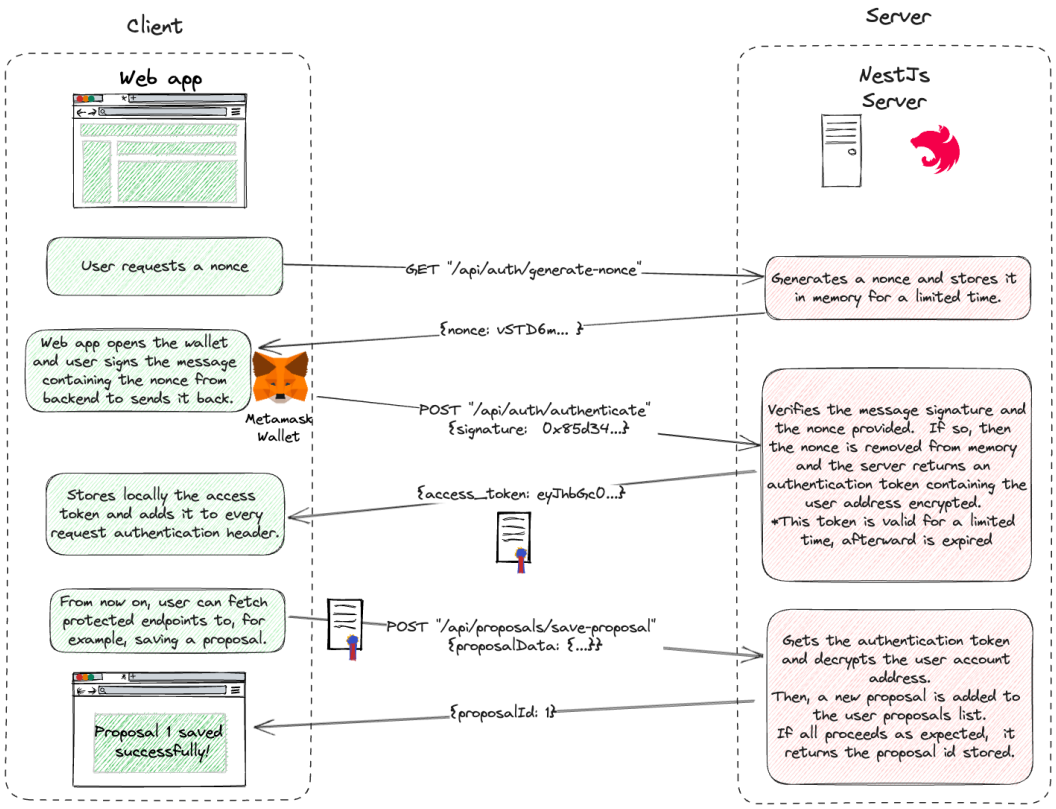

Figure 11: Authentication flow

In addition, the endpoint to get user proposals must be protected as well so it requires the authentication token. Therefore, when the backend app receives a request, it checks the token, gets the account and retrieves the list of proposals for that user. This idea of authenticating through the Ethereum account is based on the articles [23] and [24].

3.4 Frontend

The frontend app is the last piece to complete the system. It aims to provide a platform for the users that want to interact with the dapp. Moreover, it will be the unique software to interact with both, off-blockhain and on-blockchain backends.

There are many technologies to develop a frontend web application. For the experiment, I selected **Next.js**. Next.js is a **React**[6] framework to create full-stack web applications that include out-of-the-box features to compile, bundle and build the application which reduces widely the time spent in configuration. This is one of the most common solutions adopted by the frontend community.

React, like most of the libraries to develop web applications, is based on the idea of building User Interface (UI) components and composing them like a tree with nodes in

the DOM in order to eventually create interactive pages as is represented in the figure 12. Such components can be compared to math functions, which receive input data and return visual elements that are displayed in the browser.

In addition, components can pass data to other components and react to user interactions such as clicking a button or filling out a form. Every event or change triggers a render which means the component is computed again to display the change in the layout and it propogates the change to the child components. React counts on a really efficient algorithm called "*Virtual DOM*" to compute the changes between two renders and display the new UI immediately in the browser Dom.

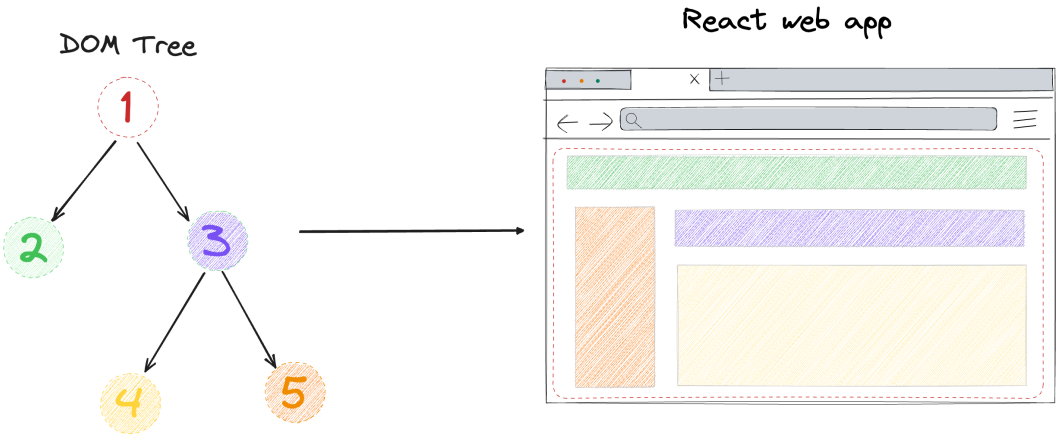

Figure 12: React tree to browser DOM.

Next.js provides a powerful layer which is known as **Server side rendering (SSR)**. As you may see in figure 13, when the user requests a web page SSR server pre-renders the content in the server and responds with the first version rendered to the client. This significantly enhances the user first-load experience and page indexation in browser engines. Once the app is displayed on the client, the next renders happen on the client side.

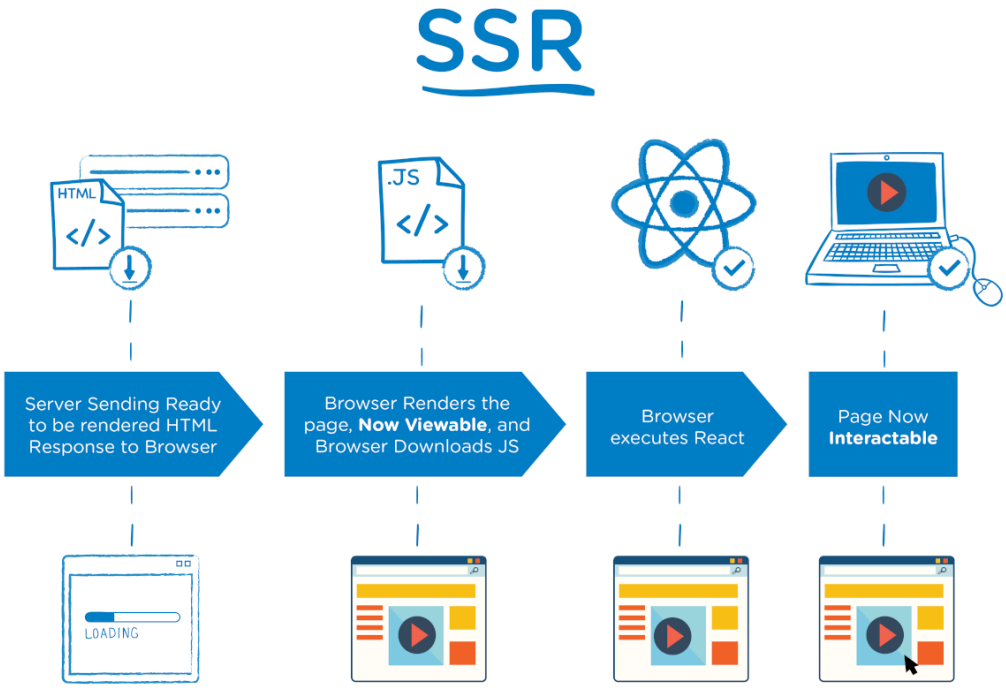

Figure 13: Server side rendering of web applications.

The project has used libraries like *Material UI*[17], which provide a huge amount of stylish and pretty components that can be used to ease the design and development process resulting in a more fancy layout. About the blockchain part, it consumes the ABI's generated by the contracts compilation and along with the Javascript libraries *wagmi.sh* and *ethers.js*[14], the communication with the smart contracts and the wallet becomes pretty straightforward.

3.5 Version control

All the code of the project has been stored in **Github**[8], a cloud version control storage that allows us to host our **Git** repository. Git[7] is an open-source version control system worldwide used by developers to work with code in a traceable, organized, and clean way. It lets the developers access all the codebase history, creating incremental versions and forking to try different ideas but also merge when necessary.

Having all the code centralized in a single source of truth, which let me create different versions or approaches of the experiment was fundamental to not blocking the progress of the project and working on several features without breaking a stable incremental version.

4 Results

Eventually, the dapp took shape thanks to the combination of the technology explained in the previous section. By using the backend on-blockchain, the backend off-blockchain, and the frontend web application, I have built an accessible web app that lets standard users the possibility to contract a car insurance policy with blockchain empowered capabilities, such as ownership and censorship-resistance. The clients can read the policy contract before confirming the purchase and make it clear what the company can and cannot do. All the operations are transparent and there is even a trace of all computations carried out so the blockchain model enhances the situation of the traditional model. The dapp project has been called **Insurechain**, mixing the words "blockchain" and "insurance".

4.1 Dapp software overview

The resulting software stack of the dapp may be cumbersome. It relies on every piece in charge of some specific tasks that combined with the other pieces deliver a comprehensive and valuable product. To have a clear idea of the elements involved we can take a look at the figure 14.

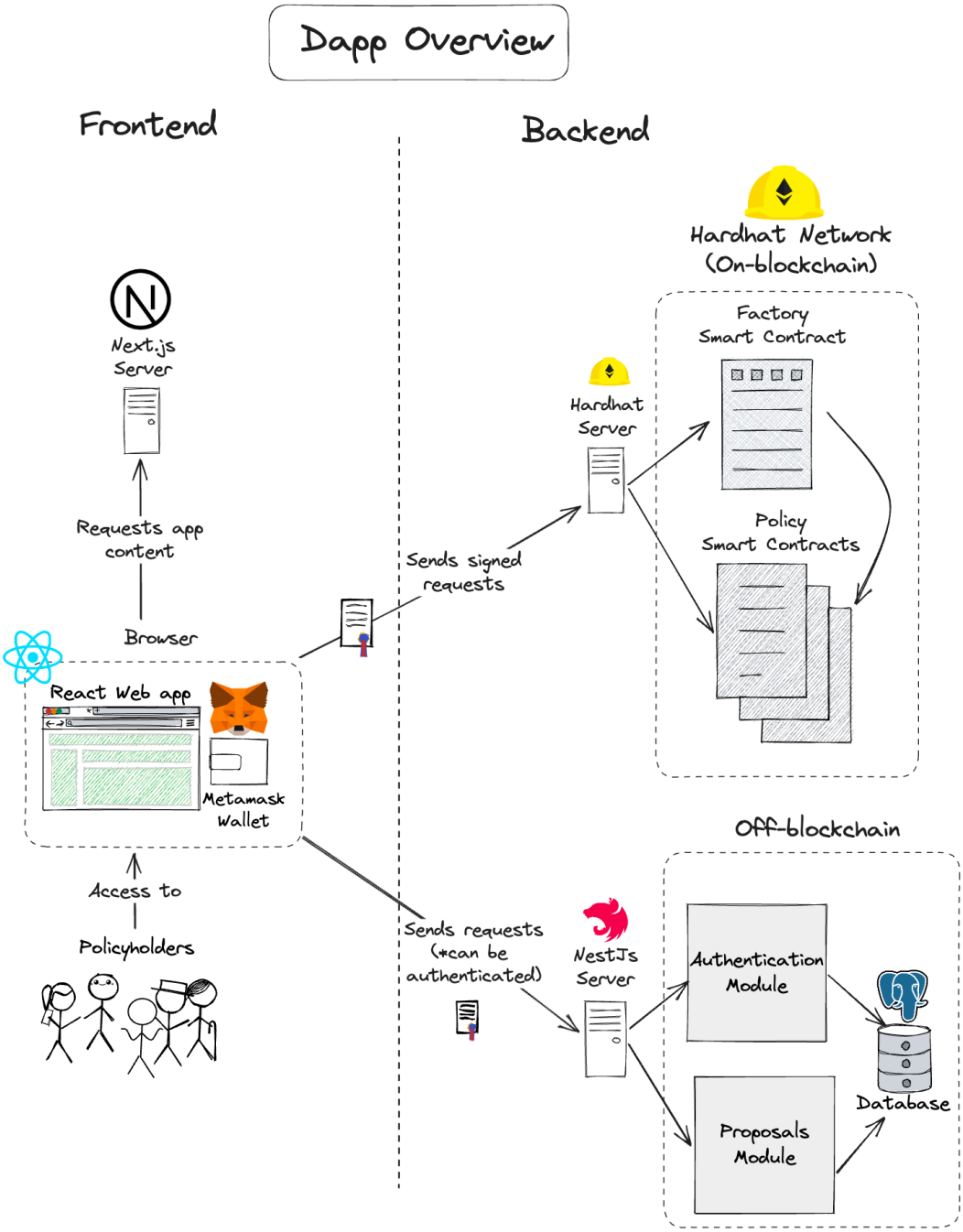

Figure 14: Dapp software overview.

There are some insights to highlight in this breakdown:

- The policyholders can easily access the dapp through the web app technology and a wallet such as Metamask. The Next.js server is responsible for delivering the UI that the web app demands to display the data properly.
- The communication with the backend can be authenticated or not depending on the moment the user signs in using the wallet. This way, the identification of a

user is the same for both backend pieces: the user address. Although the Ethereum transactions get signed by protocol automatically, the NestJs server verifies as well the user by the authentication flow based on "Siwe".

- The Nestjs server protects some endpoints according to the need to create data bound to a specific user address.
- The Nestjs modules rely on the relational database to store all the proposals list of the users, where the primary key is their account address. In this manner, we prevent the user from consuming gas, i.e. spending money, just to create a proposal, which from the product point of view does not make any sense. Hence, creating proposals does not have any blockchain effect.
- The web application can display all proposals of a user. If a user wants to purchase one, this transaction is emitted to the Hardhat network when the Factory smart contract is deployed. There, it creates and deploys a Policy smart contract and returns the deployment address. Metamask helps the user by displaying the transaction details and prompting them to confirm the execution.
- Since the Factory smart contract stores data from all the policies created for each user, every user can get their policy details on the web application.
- The policyholders can perform actions on the policy smart contract. The web app application has implemented the cancel method which requires the minimum computational gas to confirm the transaction.

I would like to emphasize that the backend parts do not communicate between them intentionally. Each part is in charge of storing a certain part of the business logic that has been thought for. It is beyond the scope of this experiment to make an interface for the company and evaluators, because it is a matter of fact that converting the policyholder flow proves that the conversion to the blockchain model is feasible from the other parts too since it is streamlining the communication for all parts concerned. Furthermore, from the point of view of the company, it also offers a lot of value since users have to pay to operate over the policy which represents a first barrier for reducing fraud but also there are incredible possibilities to rely on the blockchain such as trust the risk computation to the network so the policyholders cover themselves, therefore the proposal quote price and renewals regulates automatically depending on historical accident rate.

4.2 User experience flows

In this section, we can find the resulting user experience flows of what a policyholder is facing when he opens the Insurechain web application to obtain a car policy. To make explanations more clear and accurate, during this whole section the on-blockchain part is referred to as smart contracts, the off-blockchain part as the backend, and the frontend part as the web app.

4.2.1 Landing page

Every web application has a landing page. This page is where a new user usually "lands". There, he may find all the information about what the product does and which are its benefits. Moreover, the company is introduced to the client. In so doing, the user can make a quick idea about what the company offers to him. For Insurechain, I simulated what could be a landing page, as you may see in figures 15, 16, 17 and 18.

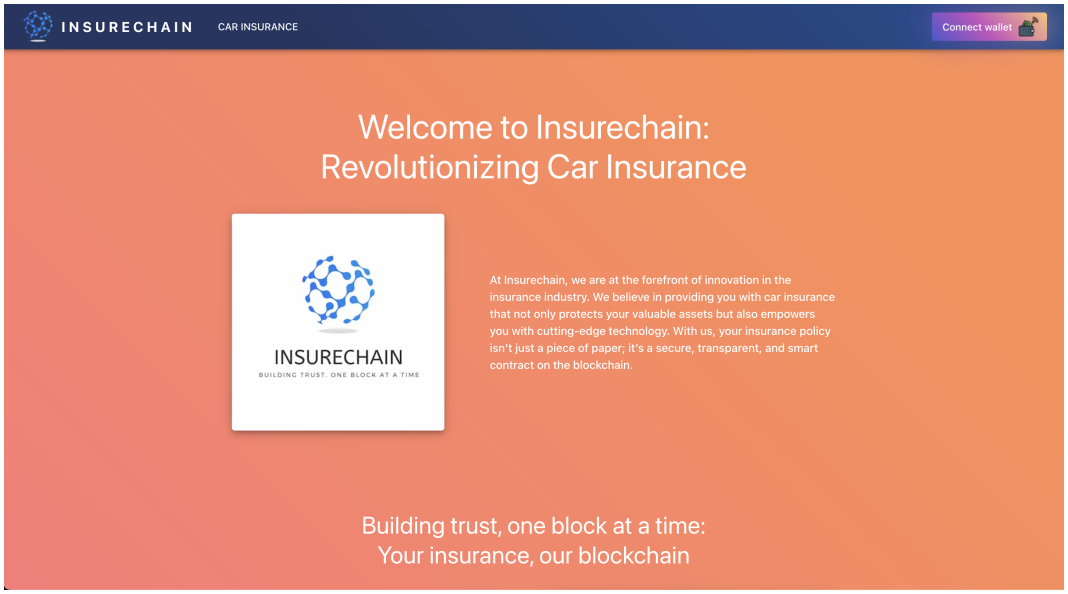

Figure 15: Landing page - part 1.

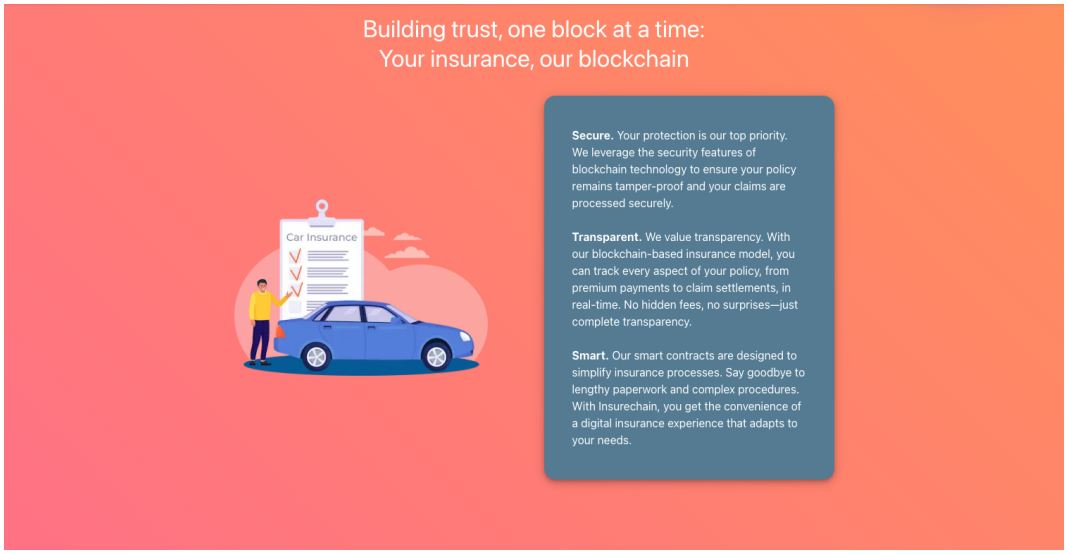

Figure 16: Landing page - part 2.

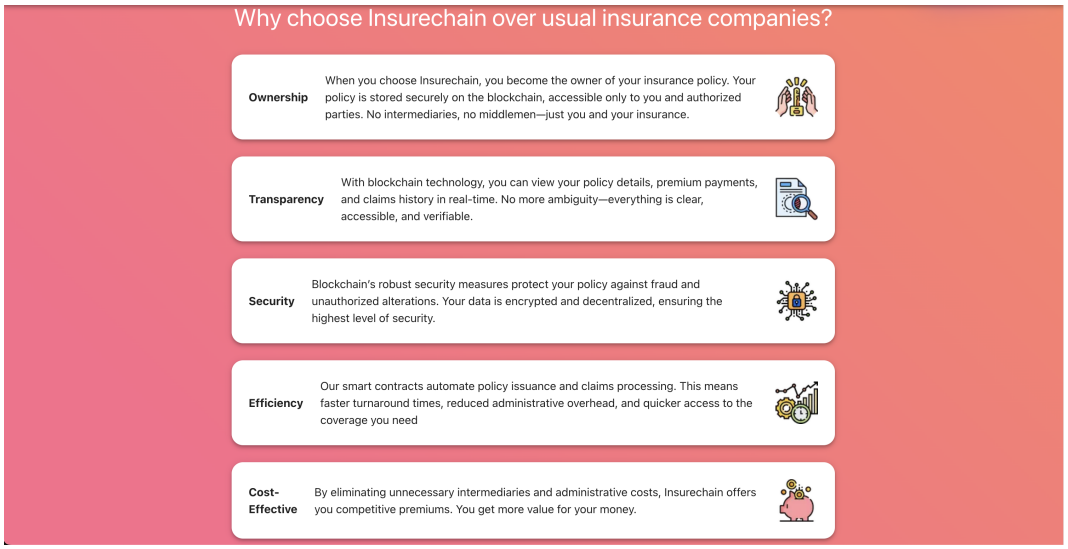

Figure 17: Landing page - part 3.

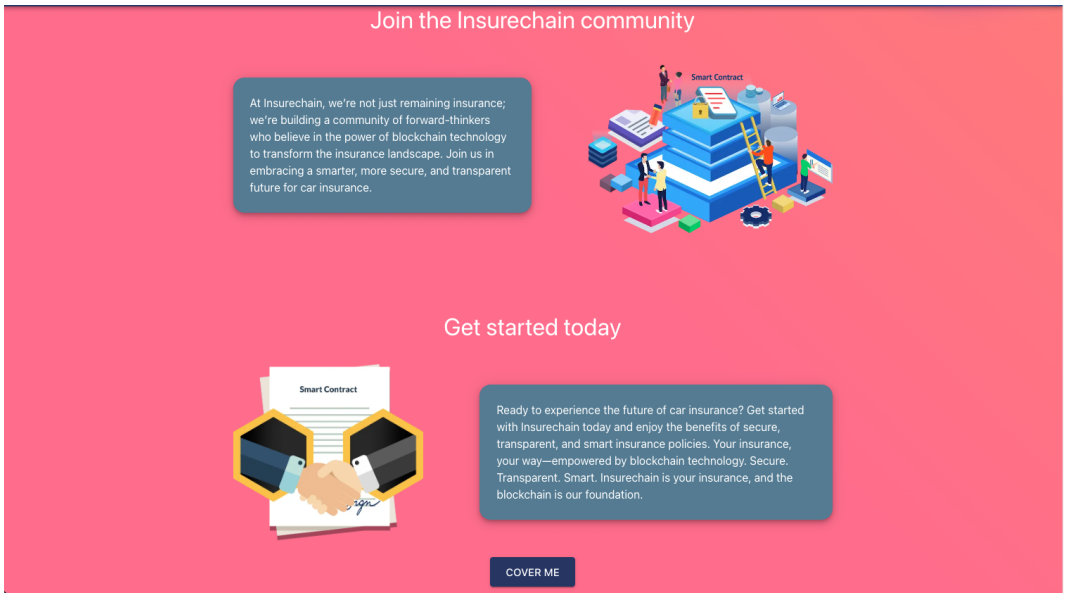

Figure 18: Landing page - part 4.

As you may see, there is a basic but comprehensive explanation of what the product achieves and the advantages for the user. Finally, the user can find a "Call to action" button to begin the process of creating a proposal.

4.2.2 Login

In the header, which is the fixed toolbar located at the top of the page that you can see in figure 19, we may find the Insurechain logo, a link "Car Insurance" which navigates

to the Car insurance proposal enrollment. In addition, on the right side, we may find a button that says "Connect Wallet".

Figure 19: Header toolbar.

Clicking on the "Connect to wallet" displays the available wallets installed in your browser. Each client is supposed to choose one. In the experiment, we have used Metamask as you may see in figure 20.

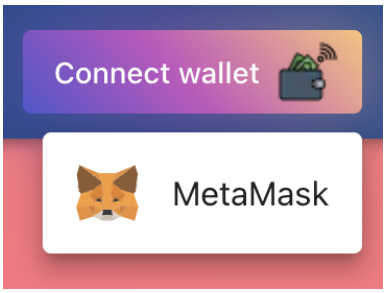

Figure 20: Available wallets.

When selecting Metamask, a new pop-up will raise to provide permissions to the web app for accessing the wallet data as shown in figure 21.

Figure 21: Metamask granting access to selected accounts

Once the user has granted the permissions to the app to consume the wallet account data, the frontend immediately requests a nonce to the backend and displays a modal where the user is asked to confirm the signing message as you can see in figure 22.

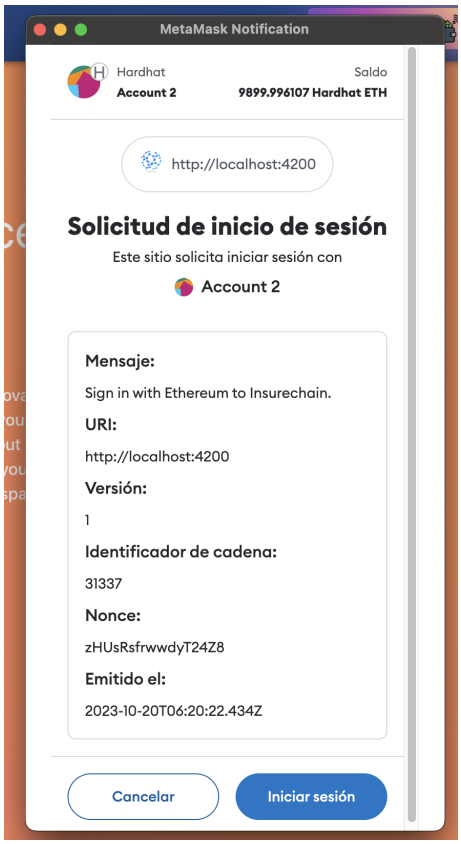

Figure 22: Metamask prompting to sign the sign-in message.

If the user accepts, the message signed is sent to the backend where it verifies it, and if all proceeds as expected, the user gets authenticated so the web displays a notification toast, like the one in figure 23.

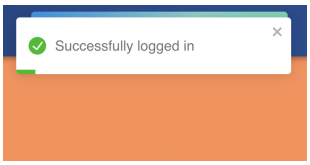

Figure 23: Notification when login succeeds.

Now, as is represented in figure 24, the header toolbar updates its layout in order to display the first and last characters of the account signed in. This is useful since a user may have several accounts.

Figure 24: Toolbar when user sign in succeeds.

When clicking this new element, a modal appears to display some information retrieved from the connected wallet. In addition, a button to get disconnected when the user desires and another button to copy the full account address is rendered. You may find it in figure 25.

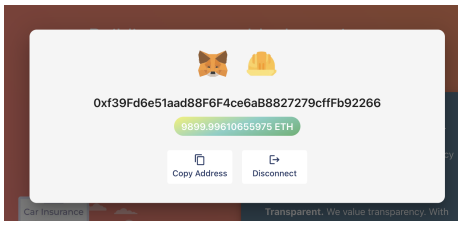

Figure 25: User account modal.

4.2.3 Creating a proposal

The user can access the signed-in or not to the insurance contraction page. There, he may find a first form where the user has to introduce all the data about the car to be covered as presented in figure 26.

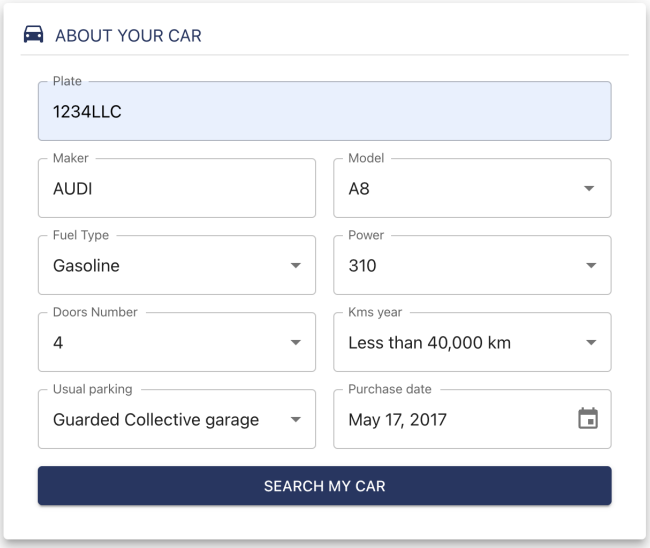

Figure 26: Car form insurance enrollment.

Then, the user must select the specific version for the car detailed as is demonstrated in figure 27. These car models are consumed for a public API.

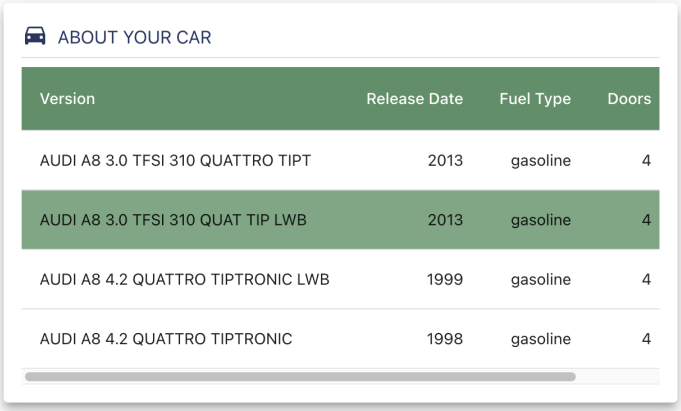

Figure 27: Car version selection in insurance enrollment.

With the specific version and model of the car defined, the app displays a form to ask for the driver's information. Note in figure 28 that the car information is collected and displayed in the first block above the driver's form.

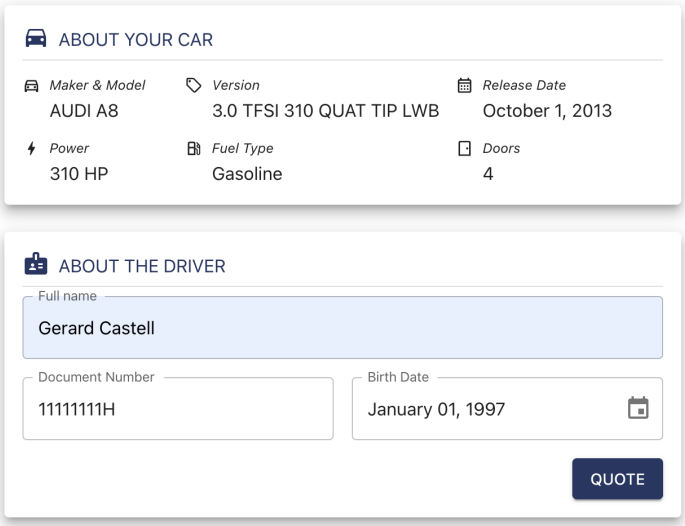

Figure 28: Driver form in insurance enrollment.

With all this information introduced, a summary of such risk figures is displayed on top as shown in figure 29, the web app makes a quote and the backend calculates a price for each coverage type. This price is random since it is out of the scope of the project. Thus, the backend server returns all the coverage products with a monthly price associated and the web displays it. Note that the price is reflected in euros and ETH. This conversion is obtained in real time to a public API.

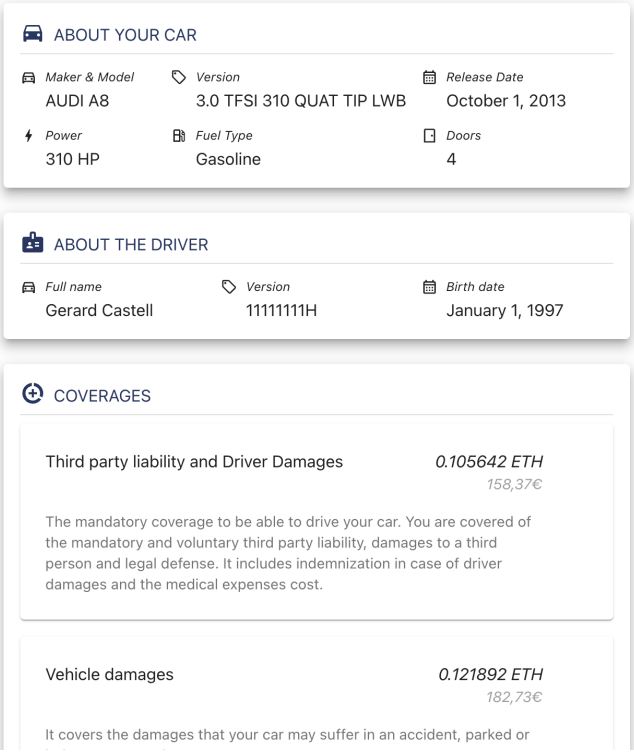

Figure 29: Risk figures summary in insurance enrollment.

The user can select the coverage types in which he is interested and then save the proposal when he is ready as illustrated in figure 30.

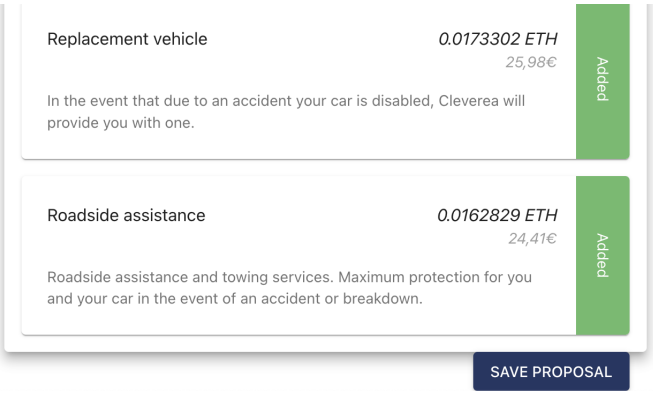

Figure 30: Coverage types selection in insurance enrollment.

Eventually, when the user wants to confirm it has to be signed in. Otherwise, a modal like the one in figure 22 will appear. Once logged in, the web app sends the proposal configuration with the risk figures data to the backend with the authentication header. The backend stores the proposal bound to that client in the database and returns an OK.

Consequently, the web displays a modal announcing the confirmation as you can see in figure 31.

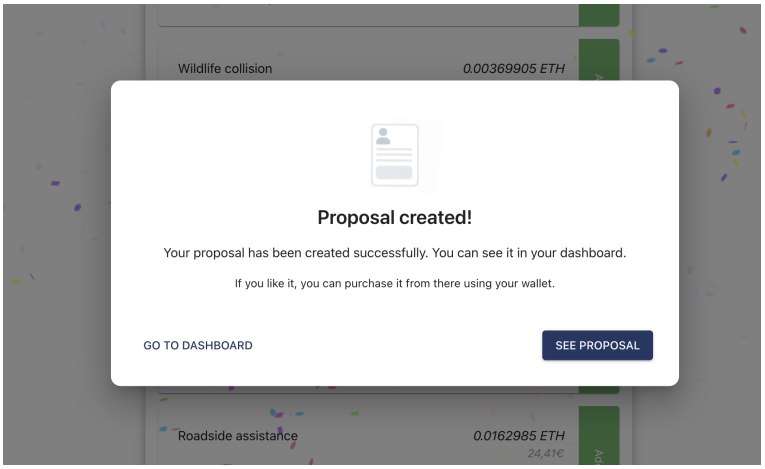

Figure 31: Succeed modal in insurance enrollment.

4.2.4 Show user proposals

If the user wants to see their proposals he can access them on the dashboard page, where he may find two boxes to visit existing proposals or policies as revealed in figure 32.

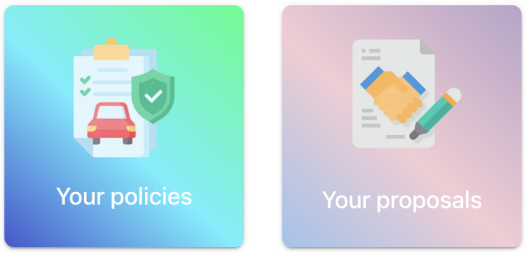

Figure 32: Dashboard page.

If the user selects proposals, the user proposals page will appear as in the figure 33. On this page, the user may find all his proposals, purchased and still not, listed with a summary of this accorded data and price.

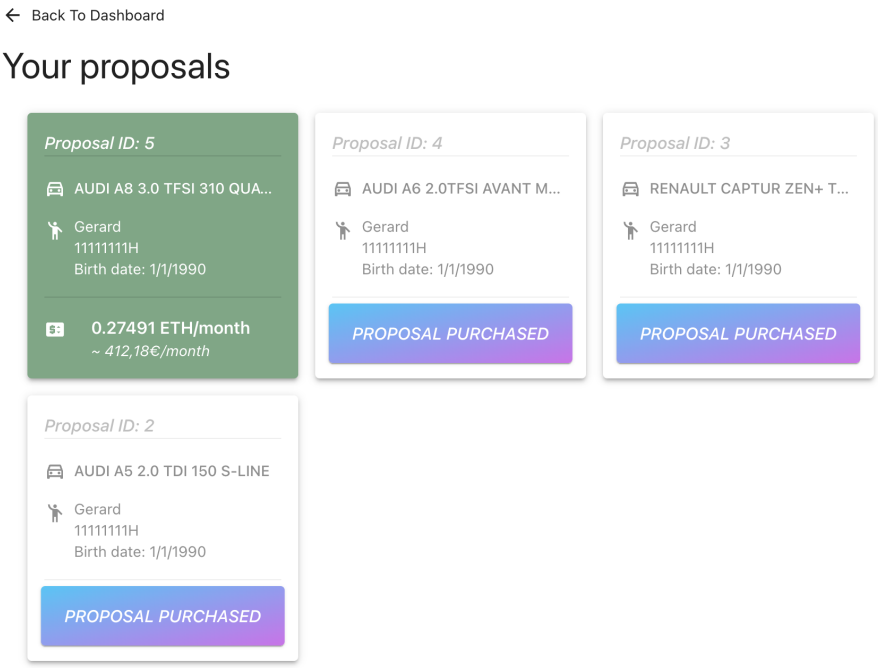

Figure 33: Proposals page.

If the user selects the first one which is the last one created he will navigate to the detail of such proposal as reflected in figure 34.

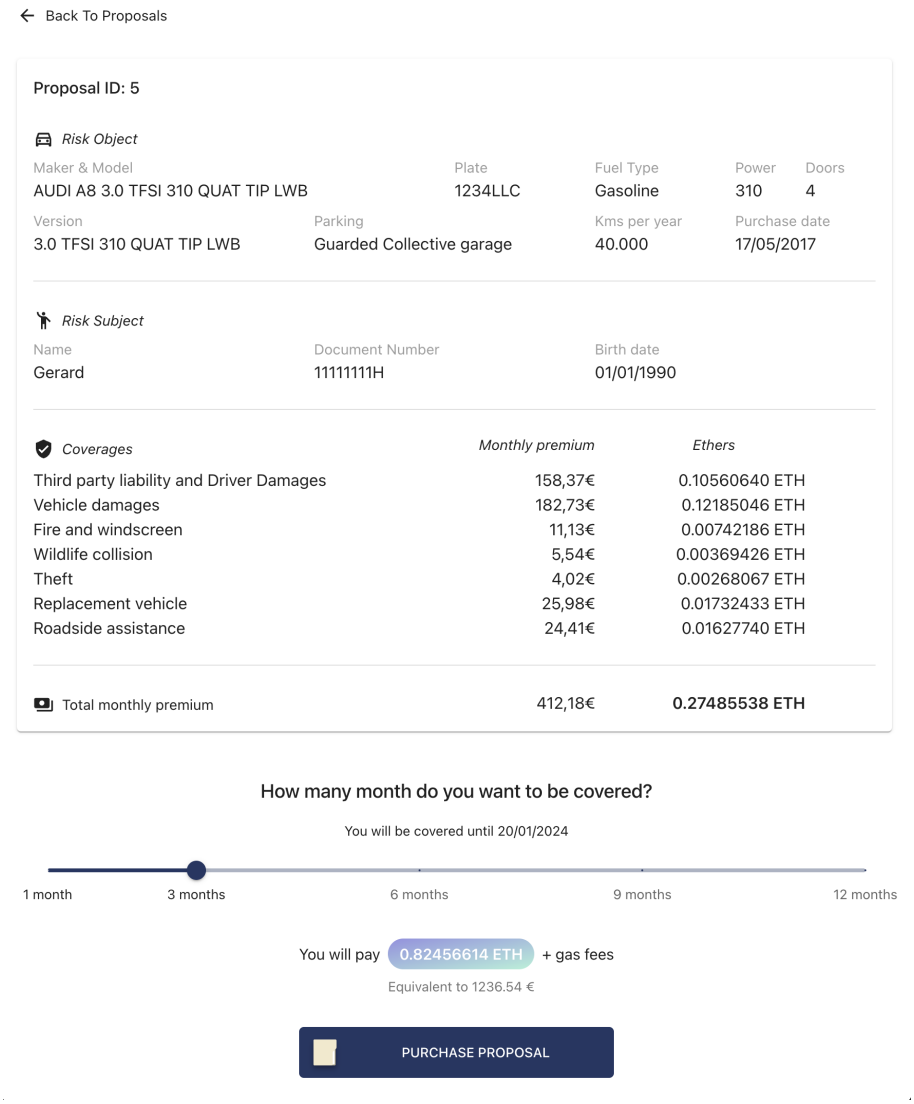

Figure 34: Proposal detail page.

The web requests the backend for the specific proposal and it is returned. Then the user may find on this page all the proposal information collected on the proposal form. It is important to focus that for purchase the user must select the number of months to be covered from the current day. The total monthly amount will be multiplied by the number of months and this total price is indicated at the bottom. Take into account that this is a rough calculation because the ETH cannot be accurate at that moment and the gas needed to purchase the policy is not estimated.

4.2.5 Purchase proposal

In the proposal detail page, when the user clicks the purchase proposal button the wallet will display a confirmation of the according transaction as shown in figure 35.

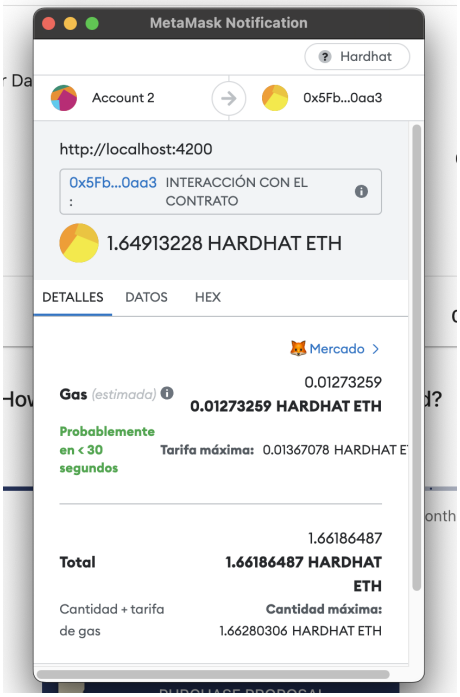

Figure 35: Purchase transaction.

If the user agrees, the transaction will be sent to the factory smart contracts. If the computation succeeds a modal will be displayed as you may see in figure 36.

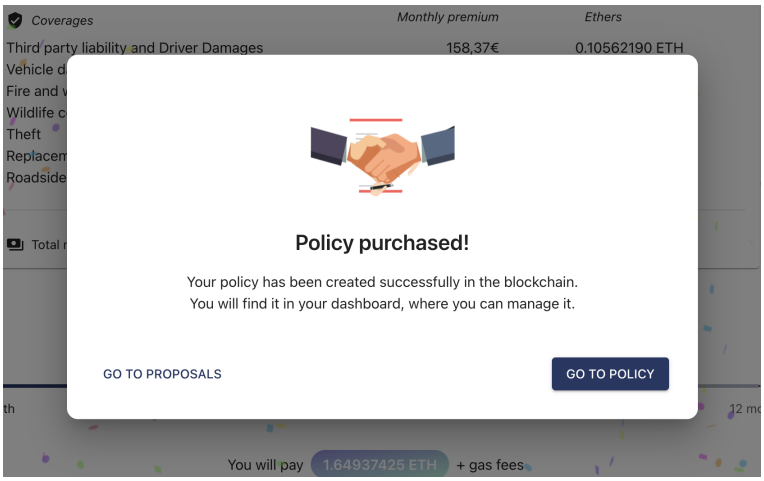

Figure 36: Purchase succeed modal.

Keep in mind, that the transaction has proceeded successfully, however, to consider the transaction as confirmed and make sure the data can be considered immutable in the Ethereum blockchain we must way a few blocks yet. If we evaluate the Metamask within that period of time we will see something similar to the figure 37.

Figure 37: Transaction pending.

Eventually, when the wallet considers that enough blocks have been mined and the state is immutable the transaction turns confirmed as you can see in figure 38.

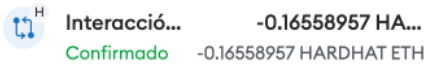

Figure 38: Transaction confirmed.

4.2.6 Show user policies

The user will find all the policies purchased and canceled on the policies page as illustrated in figure 39.

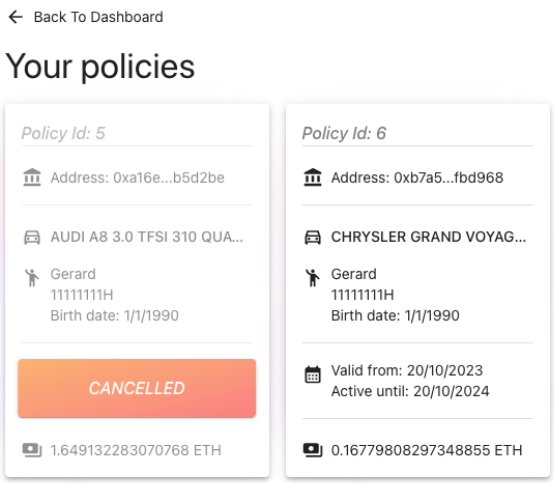

Figure 39: Policies page.

If we visit the policy purchased before, we have to see exactly the same data that was reflected in the proposal page but with the difference that now the source is the Policy smart contract deployed in the purchase and not the backend database. The total amount paid must be different because the price displayed is precisely the paid in the purchase transaction that was stored in the policy smart contract. This data comparison can be performed by comparing figures 34 and 40.

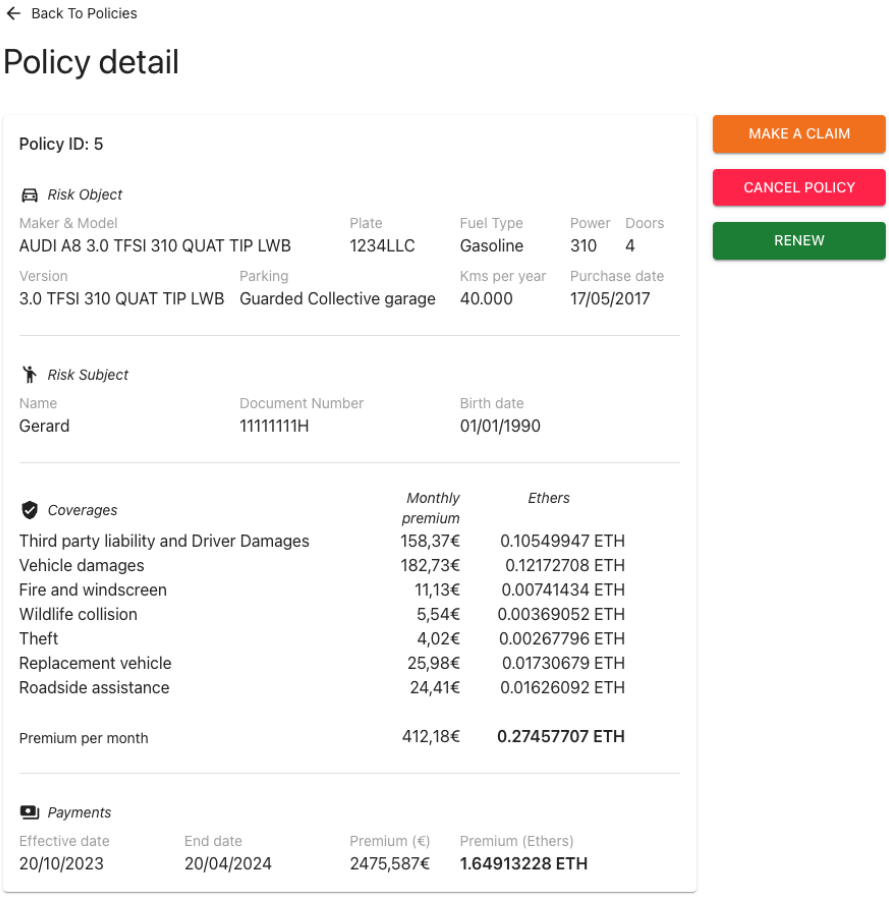

Figure 40: Policy detail page.

4.2.7 Cancel policy

Furthermore, if we look at the right side we find three buttons to make a claim, renew and cancel. Just the cancel button is implemented as explained before. If we press this button, the web app estimates how much ETH would be refunded to the user according to the time not consumed of the policy live as observed in the figure 41.

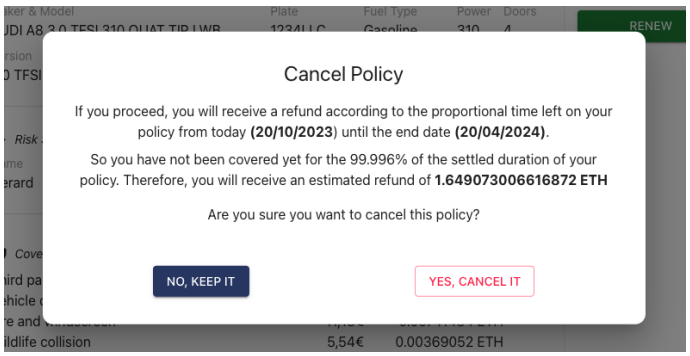

Figure 41: Cancel policy prompt modal.

If the user confirms the modal, the wallet will trigger a transaction modal to confirm the gas spent as you may see in figure 42. This is due to the user is going to modify the state of the policy contract since some store variables are going to change their value such as the end date or the renewal date. The amount of ETH is not too high so it is reasonable that this charge falls on the client side.

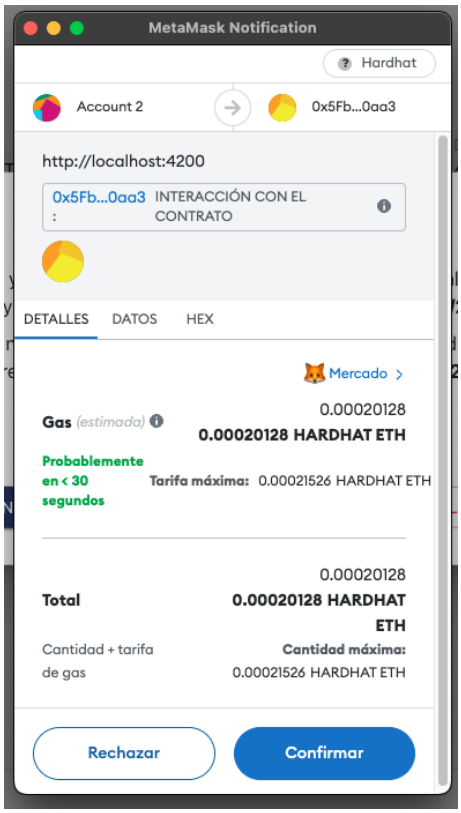

Figure 42: Cancel policy transaction.

When the user confirms the transaction if the operation succeeds the user will see the layout displayed in figure 43. Even though, as explained in the section 4.2.5 the operation has succeeded, nevertheless, to consider the state immutable we must wait for a few transactions.

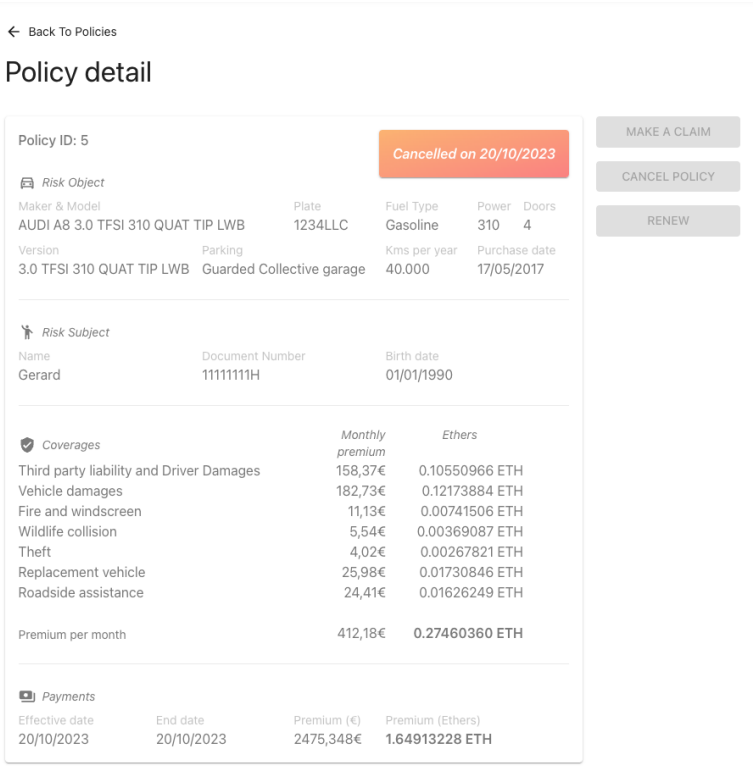

Figure 43: Policy canceled page.

5 Budget

The whole experiment has been carried out with free open source. In addition, I just needed a computer to run all the pieces locally and no more infrastructure was required. Therefore, the experiment was quite cheap.

Nevertheless, we can assume this proof of concept is performed by a company and estimate the budget. To do so, we will have to buy the computer and pay the salary of the developer and training resources.

On the one hand, the computer could be anyone with proper power which is valued at around **€800** with taxes included. On the other hand, the time spent on developing the experiment is around 600 hours in my case, 30 weeks working 20 hours per week. Moreover, we may assume a net salary for a full stack average developer of around €15 per hour. Taking into account that the 30% is subject to the IRPF tax, to maintain such a net salary we need to pay a gross salary of €21.42 per hour. In conclusion, if we multiply the 600 hours of work by the €21.42 per hour we obtain then the total salary sums up to **€12857.15**. Finally, the training to learn how to develop smart contracts and an off-blockchain backend has a price of **€40** altogether. The table 1 collects all the expenses aforementioned.

Table 1: This table gathers the different concepts that a company should pay to carry out the project as a proof of concept

Concepts	Cost
Computer	€800
Wage	€12857.15
Training	€40
Total	€13697.15

6 Environment Impact

The Ethereum consensus was previously based on Proof of Work, which mining process consumes a significant amount of energy to resolve the complex cryptographic puzzle.

Ethereum was concerned about this issue and migrated to Proof of Stake consensus finishing in September 2022, which drastically reduced the energy consumption since now miners do not race to find the hash. The energy used for PoS depends on the sources so it has even less carbon footprint if it comes from renewable energy sources.

The environmental impact of the Ethereum blockchain versus a traditional backend with a database model depends on various factors but are quite similar models in terms of energy consumption and possible solutions. The transition to PoS and the use of renewable energy sources are positive steps for reducing the environmental footprint of blockchain technology. However, it is essential to consider efficiency, scaling, and the energy mix to assess the overall impact accurately.

7 Conclusions and future development

In this thesis, I have converted a traditional insurance model to a blockchain-based approach from end to end. The experiment is considered successful since the resulting dapp gathers all the pieces needed to offer a realistic digital product. These pieces include a frontend web application to provide access to the dapp for the users, a traditional backend server and database to generate and store the policy proposals, and eventually several smart contracts in charge of managing the core business logic.

It is been really gratifying to have developed the entire software with technology ahead of the curve, which has given me a wide knowledge about how to build an MVP of a product, taking always into account the user as the focus of the product. It is been a very large project but without any doubt, I have enjoyed every line of code I have written.

About blockchain technology, I have to admit that the more I read about it the more impressed I was. To figure out how transactions can be programmable brings to my mind a lot of business cases to apply in which all parties involved would benefit from. It truly represents a game changer in a moment where the big companies have our data ownership completely. Without any doubt, I would love to see how companies start migrating toward Web 3.0 models where all transactions and data live in a more secure, traceable and censorship-resistance ecosystem. However, it represents a huge technical investment that without representing a clear benefit for companies I do not believe that they will carry out.

Moreover, it is very satisfactory to know that this exact dapp could be deployed in a real-world scenario without a few modifications. Even though, if I get realistic I consider most end-users might find to understand the Ethereum basis and use this kind of tool that really leverages blockchain power cumbersome. In favor of reducing this effect, software solutions built on top of the Ethereum layer, such as browser wallets like Metamask, are appearing in order to make it more accessible and understandable. Nevertheless, the migration of the Web 2.0 models towards blockchain-based solutions will have to happen slowly and incrementally, trusting part of the logic and data managed in centralized services taking care of not eliminating the benefits of blockchain. It is a good example of this the off-blockchain server that generates and stores proposals in the experiment which is forced to exist in order to not differ too much from the original insurance business model.

From my point of view, we are still far from using daily dapps for our regular tasks but as technology advances, it may represent at some point a crucial player to take advantage of. In the same way that large-language models have nowadays exploded in popularity and demand and most companies are trying to include them in their stack to improve their product value, perhaps at some point, blockchain will be the key to offering these valuable solutions too.

In brief, we cannot force users to use a specific technology but that technology must offer a solution to the users in order to be used.

Glossary

51% attack A 51% attack is an attack on a distributed network by a group of dishonest nodes who control more than 50% of the network's mining node rate. Owning 51% of the nodes on the network theoretically gives the controlling parties the power to alter the state as desired . 67

API API stands for application protocol interface, which is a set of definitions and protocols for building and integrating application software in other to communicate with other services and products without having to know how are they implemented.. 22

dom The Document Object Model (DOM) is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content.. 35

economy of scale A proportionate saving in costs gained by an increased level of production. 67

Gossip protocol The Gossip protocol is a protocol that allows designing highly efficient, secure and low latency distributed communication systems Peer-2-Peer(P2P). 65, 67

JSON A JSON (JavaScript Object Notation) is a simple text formatting used for data communication. 26

nonce A nonce is an arbitrary number used only once in a cryptographic communication created randomly or pseudo-randomly.. 33, 64

ORM Object Relational Mapping (ORM) is a technique that creates a bridge between object-oriented programs and relational databases.. 32

tip The priority fee (tip) incentivizes validators to include a transaction in the block. Without tips, validators would find it economically viable to mine empty blocks, as they would receive the same block reward. Small tips give validators a minimal incentive to include a transaction. For transactions to be preferentially executed ahead of other transactions in the same block, a higher tip can be added to try to outbid competing transactions. 18, 20, 67

References

[1] Javascript. https://developer.mozilla.org/es/docs/Web/JavaScript.

[2] Nomic Foundation. Hardhat network. https://hardhat.org/.

[3] Nest.js. https://nestjs.com/.

[4] Solidity. https://soliditylang.org/.

[5] Next.js. https://nextjs.org/.

[6] React.js. https://react.dev/.

[7] Git. https://git-scm.com/.

[8] Githut. https://github.com/.

[9] Postgresql. https://www.postgresql.org/.

[10] Adam Hayes. Who is satoshi nakamoto? https://www.investopedia.com/terms/s/satoshi-nakamoto.asp, 2023.

[11] Harsha Goli. Hd wallets explained: From high level to nuts and bolts. https://arshbot.medium.com/hd-wallets-explained-from-high-level-to-nuts-and-bolts-9a41545f5b0, 2018.

[12] Metamask. https://metamask.io/.

[13] Proof of stake. https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/.

[14] Ethers.js. https://docs.ethers.org/v5/.

[15] Infura. https://www.infura.io/.

[16] Geth. https://geth.ethereum.org/.

[17] Material ui. https://mui.com/.

[18] Everett Muzzy. What is Proof of Stake. https://consensys.io/blog/what-is-proof-of-stake, 2020.

[19] Vitalik Buterin. What is Proof of Stake and why it matters. https://bitcoinmagazine.com/culture/what-proof-of-stake-is-and-why-it-matters-1377531463, 2013.

[20] CME Group. Defining Ether and Ethereum. https://www.cmegroup.com/education/courses/introduction-to-ether/defining-ether-and-ethereum.html, 2022.

[21] Vitalik Buterin. Ethereum Whitepaper. https://ethereum.org/en/whitepaper/, 2014.

[22] Ethereum Organization. Opcodes for the EVM. https://ethereum.org/en/developers/docs/evm/opcodes/, 2023.

[23] Leon Do. How to use: Sign in with Ethereum. https://leondo.medium.com/how-to-use-sign-in-with-ethereum-ce712e622ee6, 2023.

[24] Rocco. Why Sign-In with Ethereum is a Game-Changer. https://blog.spruceid.com/sign-in-with-ethereum-is-a-game-changer-part-1/, 2022.

[25] Lara Parvinsmith. web3.js vs ethers.js: a comparison of Web3 libraries. https://dev.to/lparvinsmith/web3js-vs-ethersjs-a-comparison-of-web3-libraries-2ap5, 2022.

Appendices

A Deployment scenarios for a dapp

The hardware required to deploy the dapp depends on the use of it, so we can draw two scenarios:

- Development scenario. This case is used by developers to implement the software features before releasing it to a real scenario, well-known as Production. In this scenario, we just want to try the code locally and we do not need to make it public and accessible through the Internet for other users but us. This is the cheapest use case due to we can locally serve all the content on our computer. Assigning a different port for each process we can simulate a network where all the requests and communications happen through the different protocols like a real case would do. Both frontend and backend frameworks offer scripts out of the box to serve it locally. About the blockchain, we can simulate a local Ethereum network thanks to the Hardhat tool [2]. This tool emulates and hosts a real configurable network and, what is more, provides out-of-the-box a bunch of fake Ethereum accounts with Ethers to play around in our sandbox. With these resources at hand, we can create and deploy a fully functional dapp with the capability to test it using simulated assets, all with a standard computer. This entire process can be accomplished without the need for an internet connection.
- Production scenario. If we want to monetize our dapp with real users we will have to deploy it properly in order to make it accessible through the Internet. There are several points to consider:
  - Blockchain network. If we want to work with real Ethers, the Ethereum tokens, we will have to use the main net. To do so, we will have to pay for the deployment of the smart contract.
  - Frontend app, backend off-blockchain and the database. We will have to host them on public servers and connect them. Then, we will need a cloud service provider such as Amazon Web Service[**?**] or Google Cloud Platform[**?**] in order to host such servers. Moreover, we must consider the traffic scale. We should increase the number of machines and/or their power according to the traffic received and evaluate which point represents a bottleneck. If the amount of visits expected is really high, we may consider more cloud pieces to reduce the overload such as CDNs. A whole exercise of cost optimization can be performed here in order to optimize the user experience versus the cost of the infrastructure.

Therefore, to carry out this project I considered the developer scenario, given that the main goal of this project is for academic purposes. However, if at some point we head for monetizing the dapp, we will have to scale the platform and pay for the allocated resources in the cloud according to the traffic volume.

B Hashing

Hashing is the process of transforming any given key or a string of characters into another value. This is usually represented by a shorter, fixed-length value or key that represents and makes it easier to find or employ the original string.

The most popular use for hashing is the implementation of hash tables. A hash table stores key and value pairs in a list that is accessible through its index. Because key and value pairs are unlimited, the hash function will map the keys to the table size. A hash value then becomes the index for a specific element.

A hash function generates new values according to a mathematical hashing algorithm, known as a hash value or simply a hash. To prevent the conversion of hash back into the original key, a good hash always uses a one-way hashing algorithm.

Hashing is relevant to data indexing and retrieval, digital signatures, and cryptography.

C Proof of Work

There is a kind of blockchain, like Ethereum, that is defined as a permissionless distributed ledger, which means there is no need to ask for permission to participate in consensus. If everyone can be a member of the network, we have to ensure that no one corrupts the network. The Bitcoin's creator, Satoshi Nakamoto[10], relates the capacity to influence the transaction order to the work that you do:

- Nodes participating in consensus are called **miners**.
- Miners build consensus using an algorithm called **Proof of Work**.
- Miners participating in PoW spend money on equipment and electricity.

This way if you want to corrupt the network it will cost you a lot of money. Moreover, how does it motivate the miners to behave honestly? It applies game theory: workers get rewarded for their work with digital money known as **crypto**.

Miners are in charge of storing and processing all the state of the blockchain. Such a state is represented by blocks, which are chained and ordered one by one as they are being processed. Each block contains a bunch of transactions and is chained to the previous block with a hash pointer, so if we process all blocks we obtain all the history of committed transactions ordered, which represents the current state of the ledger. Hence, we can define mining as finding a consensus, which is a very costly process.

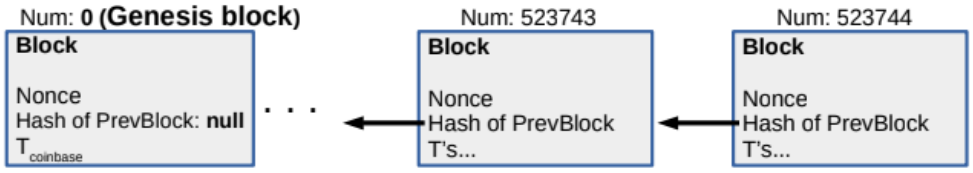

Figure 44: Chaining blocks.

Mining is the process of introducing a new valid block to the network. To do so, they must perform the following steps:

1. **Transaction validation.** When a user initiates a transaction, it is broadcast to the network. Before being added to the blockchain, these transactions must be validated to ensure they adhere to the network's rules and that the sender has sufficient funds to make the transfer.
2. **Candidate block creation.** Miners create the potential block by gathering a collection of validated transactions, known as "mempool". In addition, the new block must include a reference to the previous block identifier, which is a unique hash. This way we ensure a chronological order.
3. **Proof of Work.** Before a miner can add their candidate block to the blockchain, they must solve a complex cryptographic puzzle. The operation consists of finding a **nonce**, which will be the identifier of this new block so must be unique. This nonce has to be generated from hashing the content of the block itself and the challenge is that the hash must be led with a certain number of zeros. This property is known as *"difficulty"*. Finding these hashes supposes a significant computational power.

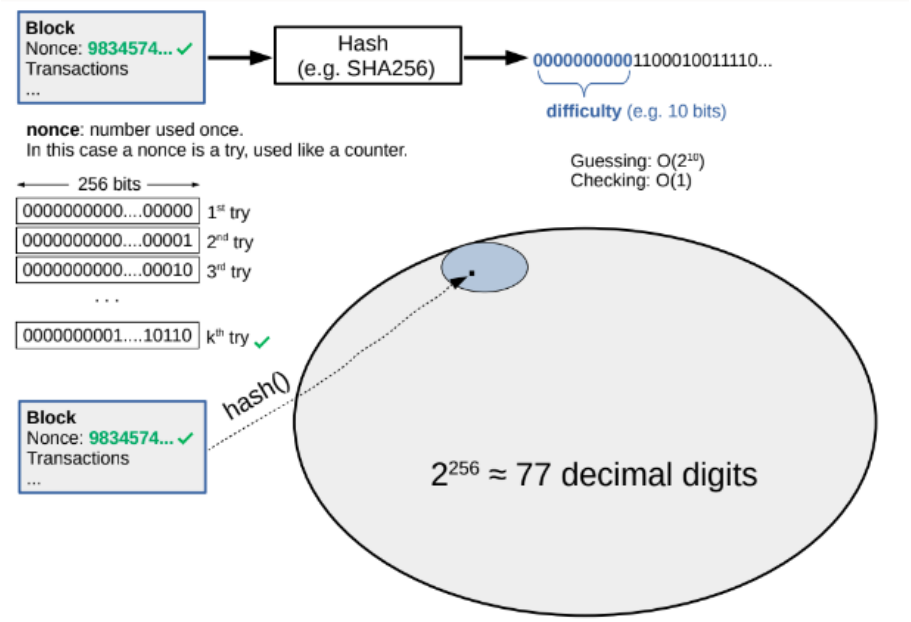

Figure 45: Difficulty in nonce computation.

4. **Competition.** Multiple miners on the network compete to solve the puzzle first like a race. Miners have to continuously calculate different nonce values until one of them successfully fits all the conditions. We can compare it to a lottery, therefore, it guarantees no single entity monopolizes the block creation.
5. **Consensus and block addition.** Once a miner successfully solves the puzzle and adds their candidate to the blockchain it communicates the network through the "*Gossip protocol*". Then, other participants of the network can easily verify that the solution of your block is correct, so the consensus is reached and the new block becomes part of the blockchain.

To compensate the miners after accomplishing the hard task of mining there are two kinds of rewards:

- The first one is the coinbase reward. It is decided by governance consensus and it is processed at the first transaction of the block added. This is the only way of creating money in the system automatically.
- The second one is collecting fees from the transactions included in the block.

Nowadays, the difficulty is already to the point where it requires over a quadrillion (15 zeros) hashes to solve a block. Moreover, the probability of solving the block is fewer than one in a billion. It is also important to mention that the difficulty is adjusted to produce a constant average inter-block time (after a certain number of blocks). Inter-block time is like the *blockchain clock* and allows engineering: halving date, network capacity, etc. In Bitcoin, the time inter-block is 10 minutes, whilst in Ethereum is 15 seconds, which obviously is a benefit to consider for dapps deployed in this network.

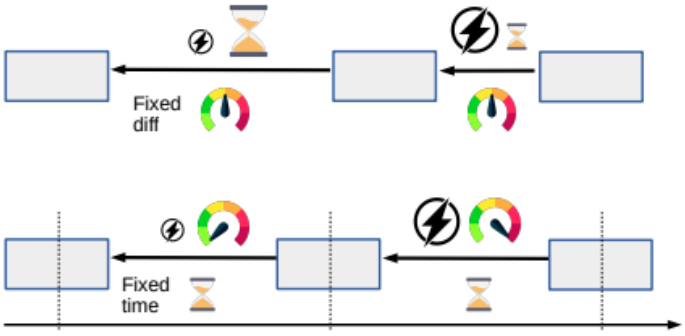

Figure 46: Fixed difficulty compared to fixed time.

History is not immediately consistent in PoW because finality is probabilistic. We have to wait to consider the blocks written in the chain immutable. For instance, Bitcoin recommends waiting 6 blocks, which is equivalent to 1 hour, to confirm a transaction.

In a nutshell, Proof of Work consensus can be summarized with the following ideas:

- Proof of Work consists of solving a cryptographic puzzle built with a hash.
- Each miner is solving its own puzzle, i.e. mining its own block.
- Notice that the nonce is valid for a particular block only.
- No one needs permission to participate in the consensus and no one (banks, companies) has privileges to solve the challenge: To solve the challenge faster you need more hash power but "anyone" can do it by buying more equipment and wasting more electricity.

D Proof of Stake

Ethereum initially was based on Proof of Work, but in 2022 switched the mechanism to Proof of Stake (PoS)[13]. The main reasons were that it was less energy-intensive, more secure and better for implementing more scalable solutions.

In Proof of Stake, the participants, henceforth known as **validators**, put something valuable into the network that can be destroyed if they act dishonestly. In the case of Ethereum, validators stake capital in the form of ETH, into a smart contract on the network. The validators are in charge of checking that new blocks inserted into the network are valid and, occasionally they also create new blocks. If they behave dishonestly, for example by proposing multiple blocks instead of one or sending invalid blocks, then the consensus mechanism will destroy partially or completely their staked ETH.

Whereas under Proof of Work, the timing of blocks is determined by the difficulty as explained in the Proof of Work subsection, in Proof of Stake the timing is fixed. This time is divided into slots of 12 seconds and epochs, composed of 32 slots. Now we can evaluate how this mechanism gets a transaction executed:

1. Users create and sign transactions, where they define the amount of gas they would pay as a tip for validators to encourage them to include the transaction in their blocks.
2. The transaction is then validated by a miner, which also checks whether the sender has enough ETH to pay the transaction.
3. Then the participant adds the transaction to his mempool of pending transactions and broadcasts the network through the Gossip protocol, so other nodes will add it to their local mempool too.
4. A network node is selected pseudo-randomly with the aim of proposing a block for the current slot, which will include such a transaction. This node is responsible for creating and adding the next block to the blockchain. To do so, the node runs in parallel with the execution client and the consensus client: the execution client bundles a bunch of transactions from his mempool into an **execution payload** and computes locally the new state. Such information is sent to the consensus client which wraps the payload into a **beacon block** that includes extra information about rewards, penalties, etc.
5. Then, the nodes of the network receive the new beacon block and the execution client re-executes the transactions locally to ensure the proposed state is valid. Thereupon, the validator client attests that the block is valid and then it represents the next block of the network from its point of view. The block with the greatest weight of attestations is considered the next block of the network and the consensus is achieved.
6. The transaction can be considered finalized whether it is part of a chain with a **supermajority link** between two checkpoints. A checkpoint is set at the beginning of an epoch and this is due to just a subset of validators attesting in each slot, whilst all validators attest during an epoch. Hence, we can only ensure the transaction is finalized between two checkpoints, a supermajority link since the 66% of total staked ETH on the network agrees.

Proof of Stake was created to enhance the conditions of Proof of Work. Staking makes it easier for individuals to participate since a validator node can run on a usual laptop and staking pools allow users to stake without having 32 ETH. Furthermore, it is more decentralized and economy of scales is much better than mining with PoW since it can apply economic penalties for misbehavior, making 51% attacks more costly. Moreover, it is more crypto-economic resilient. If we focus on the negative we have to mention that it is really a new player so it is not as battle-tested compared to Proof of Work and it is much more complex to implement and understand because users need to run three pieces of software to participate in the mining.

E Verify transactions with Merkle proofs

Transactions in blockchain are verified using a cryptographic structure called *Merkel tree*, also known as a binary hash tree. The Merkle tree ensures the integrity of a transaction

and is really efficient to verify. To do so, the tree is constructed by hashing pairs of transactions iteratively until a single root hash is obtained. Such a node is called Merkle Root.

In the interest of verifying that a transaction is included in a block, a *Merkle Proof* is generated. This proof consists of a path from the transaction's leaf node to the Merkle root, including the hash of sibling nodes along the way. As you can observe in figure 47, the root contains a contribution from all the leaves, so for n leaves need $\log_2 n$ levels. Therefore, proofs are size $O(\log_2 n)$.

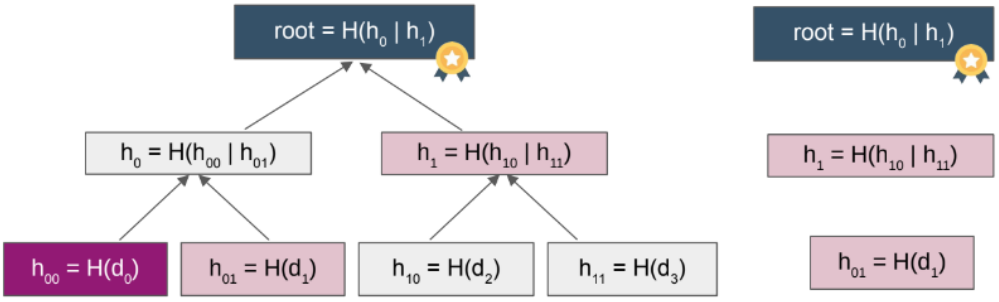

Figure 47: Merkle proof.

This merely represents the way a transaction is connected to the root, so to validate the transaction we just have to hash it with the sibling hashes and following the path indicated we obtain the Merkle root. If the computed root is the same as the one coded in the block header, we can confirm the transaction is included in the block, hence it takes part in the blockchain's history.

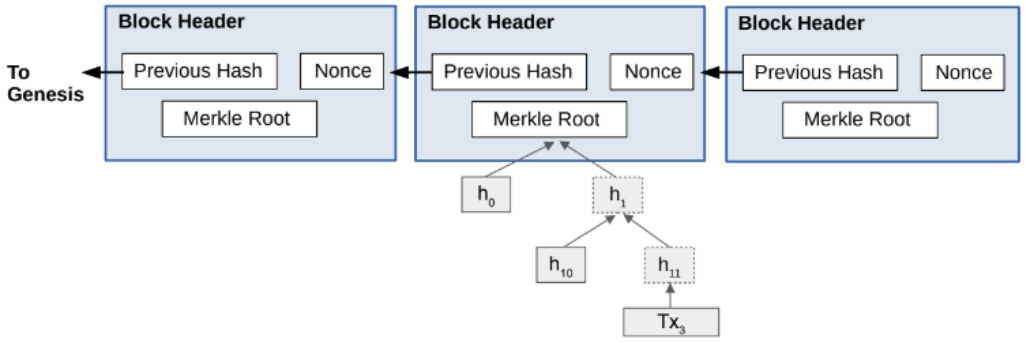

Figure 48: Merkle root in block header.

Merkle proofs enhance the security and efficiency of blockchain networks by providing a way to validate transactions without the need to verify the entire block. This method ensures data integrity and allows network participants to trust the blockchain's history while conserving computational resources. It's a fundamental component of the blockchain's transparency and trustworthiness.

F Ethereum client

Ethereum is, in essence, a group of computers inter-communicated running the same software that forms a distributed network. Such computers are known as nodes, and the software they run consists of two separate clients, the execution client and the consensus client, that communicate with each other and with the network:

- The execution client or Execution Engine listens to the network for new transactions. When it receives one it is executed in the EVM. Moreover, it also stores the latest state and database of all current Ethereum data.
- The consensus client or Beacon Node implements the Proof of Stake algorithm. It basically enables the node to obtain agreement based on validated data from the execution client. Furthermore, there is a third piece of software called "*validator*" that can be added to the consensus client, enabling the node capacity to participate in securing the network.

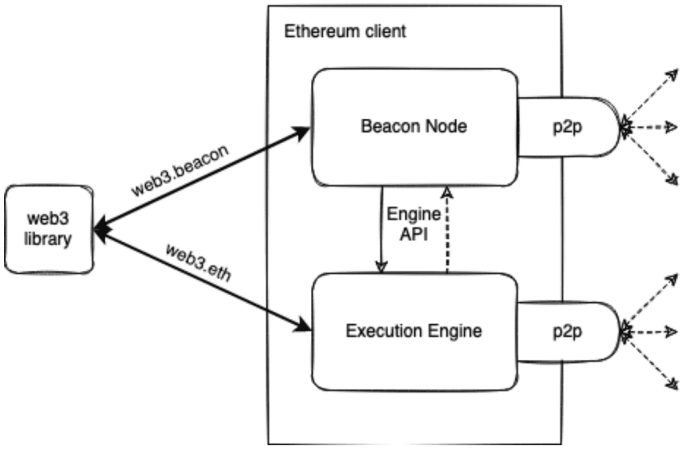

Figure 49: Simplified diagram of how clients of a node work

G Anatomy of smart contracts

A smart contract is basically data and functions that run on Ethereum when it receives a transaction. There are some concepts we have to consider about the data:

- Modify data storage in a smart contract cost gas.
- There two types of data: **memory** and **storage**:
  - Storage is data permanently recorded on the blockchain and needs to be typed so when it compiles the contract can keep track of how much storage it needs.
  - Memory is where the smart contract stores the data that just lives during the contract function's execution. Since these are not permanent, they are much cheaper to use.

- In addition, there are some kind of environment variables that provide data about the blockchain or current transaction, like the sender address or the block timestamp.

If we focus now on the functions, simplifying the idea they can just get information or set information in response to incoming transactions. We can divide the function calls into two types:

- **Internal** functions which do not create an EVM call and can just be called from the current contract or derived ones.
- **External** functions which do create an EVM call and are part of the ABI, so they can be called from other contracts.

Another type of classification which also applies to data is:

- **Public** functions which can be called internally and externally.
- **Private** functions which are only visible for the contract but not derived ones.

The functions of a smart contract can have different purposes:

- **View functions** promise not to alter the data of the contract. For example, we can think of getter functions.
- **Pure functions** promise not to either read or change data of the contract. For example, we can think about an operation function that given two integers returns the sum of them.
- **Payable functions** functions that can receive ethers as input.
- **Constructor functions** are only executed once at contract deployment. Typically, state variables are initialized here.
- **Built-in functions** that the EVM know how to interpret for instance *address.send()* in Solidity, which is use to send ETH from the smart contract to that address.
- **Writing functions** which are the remaining can receive typed parameter variables and return typed value if desired. Moreover, it can be declared as internal/external or pure/view/payable.

H Evolution of the Web

The web that we use daily has been constantly evolving since it was created. The origin is defined at CERN in 1989, when Tim Berners-Lee was developing the protocols that would become the World Wide Web with the aim of creating open and decentralized protocols that allow to sharing of information from anywhere on Earth based on a **read-only model**, like a huge Wikipedia. This first approach is known as "*Web 1.0*" and the idea was maintained until roughly 2004.

With the emergence of social media platforms, the read-only model was insufficient and unable to reach such requirements. This is why the web evolved into a **read-write** model.

Instead of only companies providing content to users they started to provide web applications and platforms to share user content and offer user-to-user interactions. This model supposed a revolution across the world in the sense of communication. Consequently, as more people were engaged to this platform more and more companies began to offer this kind of products and services to attract traffic and therefore, it caused the age of targeted advertising. It is important to highlight the major drawback of this ecosystem, to earn money from user content they need intermediaries, and that implies a forced trust for better or for worse. This model has been the standard until the present time.

As time went by, disruptive and very technical ideas such as blockchain appeared. This kind of technology has unlocked a lot of ideas to implement and eventually, in 2014 Ethereum proposed a solution for a problem that many early crypto adopters realized: the Web granted all the power of the data to private companies, which required too much trust. Thus, the Web is evolving again to offer a new era of a better internet. Blockchain-based, "*Web 3.0*" proposes a **read-write-own** model, where ownership gets distributed amongst content-generators and users. It is also permissionless since everyone has equal access to participate in the Web. In addition, payments and transactions occur straightforwardly without either intermediaries or opaque processes.

I Policy smart contract

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.9;

struct Claim {
    string title;
    address expertAddress;
    uint incidentDate;
    uint expenses;
    bool approved;
    uint resolveDate;
    bool isResolved;
}


contract Policy {
    string riskData;
    uint256 premium;
    address owner;
    address public factoryAddress;
    uint endDate;
    uint renewalDate;
    uint startDate;
    mapping(uint256 => Claim) claims;
    uint256[] claimIdList;
```

```
event Creation(address policyholderAddress);
event Cancelation(uint endDate);
event ClaimDeclaration(uint claimId);
event ClaimApproved(uint claimId);
event ClaimDeclined(uint claimId);
event Renewal(uint endDate);

modifier onlyFactoryOrOwner(){
    require(msg.sender == factoryAddress || msg.sender ==
        owner, "Just the policyholder or the insurance
        company can perform this action");
    _;
}

modifier onlyOwner(){
    require(msg.sender == owner, "Just the policyholder can
        perform this action");
    _;
}

modifier onlyFactory(){
    require(msg.sender == factoryAddress, "Just the
        insurance company can perform this action");
    _;
}

modifier isActive(){
    require(endDate > block.timestamp && renewalDate > block
        .timestamp, "Policy is not active");
    _;
}

constructor(string memory _proposalData, uint256 _premium,
   address _owner, uint256 _endDate){
    require(_endDate > block.timestamp, "Renewal date has to
        be upcoming");
    require(_premium > 0 , "Required a premium to activate
        the policy");
    riskData = _proposalData;
    premium = _premium;
    owner = _owner;
    factoryAddress = msg.sender;
    startDate = block.timestamp;
    endDate = _endDate;
    renewalDate = _endDate;
```

```
        emit Creation(owner);
    }

    function cancelPolicy() onlyFactory() isActive external {
        endDate = block.timestamp;
        emit Cancelation(endDate);
    }

    function makeClaim(uint256 claimId, Claim memory _claim)
       onlyOwner isActive external {
        Claim memory newClaim = _claim;
        claims[claimId] = newClaim;
        claims[claimId].isResolved = false;
        emit ClaimDeclaration(claimId);
    }


    function approveClaim(uint256 claimId , uint256
       claimExpenses) onlyFactory external {
        Claim storage claim = claims[claimId];

        resolveClaim(claimId, true);
        claim.expenses = claimExpenses;
        emit ClaimApproved(claimId);
    }

    function declineClaim(uint256 claimId) onlyFactory external{
        resolveClaim(claimId, false);
        emit ClaimDeclined(claimId);
    }

    function resolveClaim(uint256 claimId, bool isApproved)
       internal {
        Claim storage claim = claims[claimId];
        require(claim.isResolved == false, "Claim is already
           resolved");

        claim.resolveDate = block.timestamp;
        claims[claimId].isResolved = true;
        claim.approved = isApproved;

    }

    function renew(uint newEndDate) onlyFactory external returns
```

```
        (uint){
        require(newEndDate > endDate, "New end date has to be
            after the current one");
        endDate = newEndDate;
        renewalDate = newEndDate;

        emit Renewal(endDate);
        return renewalDate;
    }


    function getOwnerAddress() onlyFactoryOrOwner external view
        returns (address) {
        return owner;
    }

    function getClaim(uint256 claimId) onlyFactoryOrOwner
        external view returns (Claim memory){
        return claims[claimId];
    }

    function getClaimsList() onlyFactoryOrOwner external view
        returns (uint256[] memory){
        return claimIdList;
    }

    function getEndDate() onlyFactoryOrOwner external view
        returns (uint){
        return endDate;
    }

    function getStartDate() onlyFactoryOrOwner external view
        returns (uint){
        return startDate;
    }

    function getRenewalDate() onlyFactoryOrOwner external view
        returns (uint){
        return renewalDate;
    }

    function getRiskData() onlyFactoryOrOwner external view
        returns (string memory){
        return riskData;
    }
```

```
    function getPremium() onlyFactoryOrOwner external view
        returns (uint256){
        return premium;
    }

    function getIsActive()onlyFactoryOrOwner external view
        returns (bool){
        return endDate > block.timestamp;
    }

}
```

J Factory smart contract

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.9;

import "./Policy.sol";


contract Factory {
    address payable public insuranceAddress;
    uint256 minimumBudget = 0.999 ether;
    mapping(address => address[]) policiesMapping;
    mapping(address => bool) private claimEvaluators;
    event PolicyCreated(address owner, uint when);
    event PolicyCanceled(address policy, uint when, uint256
        amount);
    event PolicyRenewal(address policy, uint newDate, uint
        renewalAmount);
    event ClaimApproved(address policyAddress ,uint256 claimId ,
        uint256 claimExpenses);
    event ClaimDeclined(address policyAddress, uint256 claimId);
    event BudgetAdded(uint amount);
    event BudgetWithdrawal(uint amount);

    constructor() payable {
        require(msg.value >= minimumBudget, "Minimum budget is
            not achieved to start an Insurance Smart Contract");
        insuranceAddress = payable(msg.sender);
    }


    modifier companyOnly {
```

```
        require(msg.sender == insuranceAddress, "Just the
            insurance company can perform this action");
        _;
    }

    modifier claimEvaluatorsOnly {
        require(msg.sender == insuranceAddress ||
            isClaimEvaluatorKnown(msg.sender), "Required a valid
            evaluator approved by the company");
        _;
    }

    function addBudget() companyOnly public payable returns (
        uint256){
        emit BudgetAdded(msg.value);
        return address(this).balance;
    }

    function withdrawBudget(uint256 amount) companyOnly public
        returns (uint256){
        require(amount < address(this).balance, "There is not
            enough amount to withdraw");
        insuranceAddress.transfer(amount);
        emit BudgetWithdrawal(amount);
        return address(this).balance;
    }

    function createPolicy(string memory proposalData, uint
        endDate) public payable returns (address){
        require(msg.value > 0, "To create a policy you must pay
            a premium.");
        Policy policyContract = new Policy( proposalData, msg.
            value, msg.sender, endDate);
        address policyAddress = address(policyContract);
        address holderId = msg.sender;
        policiesMapping[holderId].push(policyAddress);

        emit PolicyCreated(msg.sender, block.timestamp);
        return policyAddress;
    }

    // Renews the policy and return the new end date.
    function renewPolicy(address policyAddress, uint newEndDate,
        uint renewalAmount) public payable returns (uint){
        require(msg.value > renewalAmount, "To renew a policy
```

```
            you must pay a premium.");
        Policy policyContract = Policy(policyAddress);
        require(msg.sender == policyContract.getOwnerAddress(),
            "Just the policyholder of the policy is able to renew
            it.");
        emit PolicyRenewal(policyAddress, newEndDate,
            renewalAmount);

        return policyContract.renew(newEndDate);
    }


    function approveClaim(address policyAddress ,uint256 claimId
        , uint256 claimExpenses) claimEvaluatorsOnly public {
        require(claimExpenses < address(this).balance, "
            Insufficient balance to pay the claim.");

        Policy policy = Policy(policyAddress);
        policy.approveClaim(claimId, claimExpenses);

        address payable holderAddress = payable(policy.
            getOwnerAddress());
        emit ClaimApproved( policyAddress, claimId,
            claimExpenses);
        holderAddress.transfer(claimExpenses);
    }

    function declineClaim(address policyAddress, uint256 claimId
        ) claimEvaluatorsOnly public {
        Policy policy = Policy(policyAddress);
        emit ClaimDeclined(policyAddress, claimId);
        policy.declineClaim(claimId);
    }

    function addEvaluator(address newAddress) companyOnly public
        {
        claimEvaluators[newAddress] = true;
    }

    function changeEvaluatorValue(address evaluator, bool value)
        companyOnly public {
        claimEvaluators[evaluator] = value;
    }

    function isClaimEvaluatorKnown(address checkAddress)
```

```
        companyOnly public view returns(bool) {
            return claimEvaluators[checkAddress];
        }

        function getHolderPolicies() public view returns (address[]
            memory){
            return policiesMapping[msg.sender];
        }

        function cancelPolicy(address addressPolicy) public {
            Policy policy = Policy(addressPolicy);
            address owner = policy.getOwnerAddress();

            require(msg.sender == owner || msg.sender ==
                insuranceAddress, "Just the policyholder of the
                policy is able to cancel it.");
            policy.cancelPolicy();

            uint premium = policy.getPremium();
            uint256 startDate = policy.getStartDate();
            uint256 renewalDate = policy.getRenewalDate();
            uint256 cancellationDate = policy.getEndDate();

            uint256 timeNotEnjoyed = renewalDate - cancellationDate;
            uint256 totalTimeSpan = renewalDate - startDate;

            uint256 timePercentage = (timeNotEnjoyed * 100) /
                totalTimeSpan;
            address payable holderAddress = payable(owner);
            uint256 amountToReturn = premium * timePercentage / 100;

            require (amountToReturn < address(this).balance, "
                Insufficient balance to pay the cancellation.");

            holderAddress.transfer(amountToReturn);
            emit PolicyCanceled(addressPolicy, cancellationDate,
                amountToReturn);
        }
}
```