



**SAPIENZA**  
UNIVERSITÀ DI ROMA

DEPARTMENT OF COMPUTER SCIENCE

**GethDomains: decentralized domain  
names on Ethereum blockchain**  
BLOCKCHAIN AND DISTRIBUTED LEDGER  
TECHNOLOGIES

**Professor:**

Claudio Di Ciccio

**Students:**

Alessandro Scifoni (1948810)

Simone Sestito (1937764)

Andrea Tarricone (1888181)

---

Academic Year 2023/2024

# Preface

## Brief presentation of the protocol

GethDomains is a decentralized platform that allows, through the use of the GETH token, the purchase of Registered Domains on the Ethereum blockchain.

Our motto: discover the new, decentralized domain name system.

## Team members and main responsibilities

From the beginning, the whole team was working on the entire project, trying not to leave anything to chance and confronting themselves when problems or new ideas arise to be implemented or not. Once the idea and the purpose of the project have been chosen, we have moved on to the definition of its architecture and then divided the work, always maintaining a synergy. In particular:

- Alessandro Scifoni: Smart Contract Developer, Pitch and Report, unit tester
- Simone Sestito: Flutter Web Developer, CI/CD pipeline, Web3.js integration
- Andrea Tarricone: Smart Contract Developer, Pitch and Report, Storytelling

## Outline of the report

Geth Domains is a decentralized platform that facilitates the association of TOR and IPFS addresses with more accessible domain names based on Ethereum. Based on blockchain, this system offers immutability and transparency of data, resistance to censorship and elimination of centralized intermediaries. It leverages this blockchain technology to ensure the security and stability of domain names, allowing users to register them as NFT via the GETH token. The platform includes a marketplace for buying and selling domains, giving users the chance to earn royalties for life when selling their domains. Two distinct smart contracts, one for the token and one for the Dapp, work synergistically to support this ecosystem without the need for intermediaries.

# Contents

<b>1</b>	<b>Background</b>	<b>3</b>
1.1	Blockchain's history . . . . .	3
1.2	How the blockchain works . . . . .	3
1.3	Consensus algorithms . . . . .	3
1.4	Smart Contracts . . . . .	4
1.5	IPFS and TOR . . . . .	5
1.6	Encodings . . . . .	5
1.7	Huffman Tree . . . . .	5
<b>2</b>	<b>Presentation of the Context</b>	<b>6</b>
2.1	Aim of the Dapp . . . . .	6
2.2	Immutability . . . . .	6
2.3	Censorship . . . . .	6
2.4	No Third Parties . . . . .	7
2.5	Permissionless Blockchain . . . . .	7
2.6	Use cases . . . . .	7
<b>3</b>	<b>Software Architecture</b>	<b>8</b>
3.1	UML Sequence Diagram . . . . .	9
3.2	UML Class Diagram . . . . .	11
3.3	Smart contracts . . . . .	11
3.4	ERC1155 consideration . . . . .	13
3.5	Reentrancy attack consideration . . . . .	14
3.6	Oracle considerations . . . . .	14
3.7	Data encoding and compression . . . . .	14
<b>4</b>	<b>Implementation</b>	<b>17</b>
4.1	Frontend webapp implementation . . . . .	17
4.2	Smart Contracts . . . . .	21
<b>5</b>	<b>Known Issues and Limitations</b>	<b>23</b>
<b>6</b>	<b>Conclusions</b>	<b>23</b>
	<b>References</b>	<b>24</b>

# 1 Background

## 1.1 Blockchain's history

The first decentralized blockchain was conceptualized in 2008, year of the publication of the Bitcoin white paper by an unknown person or group using the pseudonym Satoshi Nakamoto[1]. The Bitcoin network officially came into existence on January 3, 2009, with the mining of the first block, known as the "Genesis Block." This marked the beginning of the Bitcoin blockchain.

Ethereum, conceived by Vitalik Buterin [2], aimed to expand blockchain's capabilities beyond currency. The Ethereum blockchain went live on July 30, 2015, with its first version, Homestead. Ethereum introduced "smart contracts," enabling developers to create decentralized applications (DApps) on its platform.

## 1.2 How the blockchain works

The blockchain is a shared and an immutable ledger that contains transactions. The principal components are:

- Nodes - participants in the blockchain that through the accounts (Externally owned accounts) can make transactions and with a consensus mechanism can propose a block (miners).
- Transactions are the build-in blocks of a ledger and represent the transfer of an asset between accounts.
- Blocks are the structures that contain transactions and other useful fields like the header or the hash to the previous block.

## 1.3 Consensus algorithms

The aim of consensus algorithms is to reach an agreement through the participants of the blockchain. Given that blockchain is inherently decentralized, it requires a secure mechanism to achieve consensus on determining the latest block (implicitly to the previous ones because the hash of the previous block inside to a field of the header) and the correctness of the transactions inside the block. Different types of mechanisms to reach consensus have been proposed:

- **Proof of Work:** POW is a puzzle miners have to solve to gain the authority to publish a block by finding a nonce that results in a block hash below a difficulty threshold. POW can be solved by different miners at the same time so there can be forks, but the most computational work chain is the one considered valid. For this reason, a block is considered finalized after another 5 blocks. A miner who

solves POW gains a price, which is currently bitcoin plus the transaction fee in the block.

- **Proof of Stake:** In POS the validators are not miners, but they stake a capital of 32 Ethereum(ETH). In Ethereum Proof of Stake, the tempo is fixed. Each epoch is composed by 32 slots and each slot lasts 12 seconds: 6 seconds for block proposal and 6 seconds for block attestations. In each slot, a commit of 128 validators is randomly chosen and one, from them, is randomly selected to be a block proposer and has 6 seconds to forge a new block. Validators vote on checkpoint pairs, justifying the most recent checkpoint with 2/3 of total staked ETH votes, finalizing the oldest justified checkpoint, and employing a sync committee every 256 epochs for block validation. Like in POW, validators can have different views of the head of the chain. To solve this problem POS uses LMD-GHOST algorithm, which chooses as the "right" one, the one that has the **greatest accumulated weight of attestations**. POS introduces slashing mechanisms to deter dishonest behaviors, including double proposals, FFG double votes, and attesters signing a checkpoint attestation that "surrounds" another one.

## 1.4 Smart Contracts

Smart contracts are pieces of code that run on the blockchain. As transactions, once posted on the blockchain, their code runs uncensorably and cannot be changed. For this reason, they are considered trustworthy and secure, so participants in a smart contract do not need to trust each other or a third party. Smart contracts run on the Ethereum Virtual Machine and they are not for free. Every single bit or execution has a cost in terms of gas.

ERC are standards for smart contracts, so they are guidelines for programming on blockchain.

### 1.4.1 ERC-20

ERC-20[3] is the standard used to represent fungible tokens, meaning that **each token is identical** and can be exchanged on a one-to-one basis with any other token of the same type. These tokens are typically used to represent units of a certain resource, like coins used for accessing features or loyalty points.

### 1.4.2 ERC-721

ERC721 is the standard used to represent non-fungible tokens[4]. Unlike ERC-20 tokens, they are not interchangeable on a one-to-one basis. Each ERC-721 token is distinct and possesses unique characteristics, making them ideal for representing

ownership of unique items. They cannot be divided into smaller units like fungible tokens so they are traded as whole units.

### **1.4.3 ERC-1155**

ERC1155[5] is the combination of ERC-20 and ERC-721, so it allows the management of both fungible and non-fungible tokens within a single contract. These tokens are very useful in DApps that use batch operations, which are more efficient than operations on a single token.

## **1.5 IPFS and TOR**

IPFS is a distributed file system designed to create a more efficient and resilient way of storing and sharing content on the internet.[6] The IPFS system associates a link to the files stored.

TOR is a privacy-focused network that aims to enhance anonymity and security for users on the internet. To access websites with ".onion" domains, users need to use the Tor browser, which is designed to connect to the Tor network. This browser allows users to navigate and access content within the Tor ecosystem. [7]

## **1.6 Encodings**

Encoding is the process of converting data from one form to another, typically for the purpose of efficient storage, transmission, or representation. Some time happen that an encoding such as base64 is not efficient for the aim of the DApp, so finding one is crucial to reduce the amount of data and their cost.

## **1.7 Huffman Tree**

Huffman coding is a compression algorithm that assigns variable-length codes to input characters, with shorter codes assigned to more frequently occurring characters. The algorithm begins by analyzing the frequency of each character in the input data (e.g., a text file). The algorithm builds a binary tree known as the Huffman tree. Each leaf node of the tree represents a character along with its frequency. Starting from the root of the tree, traversing left and right corresponds to adding '0' or '1' to the code. The codes assigned to the characters are determined by the path from the root to the respective leaf nodes. The original data is then encoded using the generated Huffman codes. The resulting compressed data typically occupies fewer bits than the original data, especially if there are frequently occurring characters. To decode the data the code itself is used, crossing through the nodes up to the character's leaf associated.[8]

## 2 Presentation of the Context

Tor and IPFS are born and developed to face challenges related to privacy, security and decentralization of information on the internet.

They also allow access to online content that could be censored or blocked in certain regions.

Websites hosted on TOR are often more difficult to censor or block, thus helping to ensure freedom of information.

Similar speech for IPFS where people can publish content without having to rely on a centralized server. This way, content remains accessible even if some nodes in the network are disconnected or censored.

### 2.1 Aim of the Dapp

Accessing such data can be inconvenient as resource addresses are represented with poorly manageable hash keys.

In this regard, Geth Domains is born, a decentralized application created with the aim of encouraging a platform to associate complex TOR and IPFS addresses to more "friendly" .geth addresses.

This helps create a more accessible and user-friendly ecosystem on the Ethereum blockchain.

We decided to rely on a decentralised system as it offers several advantages including immutability and transparency of data, the absence of an intermediary and resistance to censorship.

### 2.2 Immutability

The data on a blockchain is immutable once it has been confirmed. This means that once a domain name is registered on GethDomains, it is difficult to change or delete it without the network's consent. This contributes to the stability and security of the system.

The only one able to perform operations on the domain is the possessor.

### 2.3 Censorship

No single central authority has complete control over the domain name registry.

This makes it more difficult for government authorities or other entities to censor, block or confiscate specific domain names.

## 2.4 No Third Parties

It is not necessary to rely on central intermediaries such as registers of traditional domains.

This reduces the risk of manipulation, censorship or other interference by centralized third parties.

The lack of an intermediary also leads to cost savings and greater efficiency as well as an increase in ownership control by the user as they are not dependent on anyone.

## 2.5 Permissionless Blockchain

We decided to use the blockchain as there is the need to save all transactions linked to a domain in a clear and transparent way.

In this way anyone who makes purchases can register the new ownership by writing on the blockchain.

The blockchain, in this context, plays the role of impartial intermediary within this ecosystem without knowing a priori who will use it.



## 2.6 Use cases

Our platform will allow you to register your IPFS or TOR domain by associating an address with it. Geth, put it on sale if you do not want to have more possession and therefore offer a marketplace to buy domains to users.

The domains will be registered on the ethereum blockchain via NFT and all operations carried out thanks to the native currency of the Dapp: GETH Token.

In addition, when a user sells his domain you will see a system of royalties that will allow him to earn a lifetime on each future sale and subsequent transfer of ownership.

All the operations mentioned here will be seen in the next section and in particular we will go into detail of the two smart-contracts, one for the token and one for the Dapp, which allow the existence of this ecosystem and the absence of intermediaries.



### 3 Software Architecture

The overall DApp consists of various components, summarized in figure 1:

- The code is stored in a GitHub repository, publicly accessible from the date of the submission of this project: each team member is set as a collaborator and has full access to it
- Every time a team member pushes his commits, a CI/CD pipeline on GitHub Actions will run, deploying the webapp and making the new version reachable from the associated website
- In addition to it, the HTML documentation of the latest Solidity smart contracts is generated and published ([https://gethdomains.best/smart\\_contract\\_docs.html](https://gethdomains.best/smart_contract_docs.html))
- The webapp will interact with the code stored and running on the blockchain through a Web3 provider like Metamask
- The **DomainMarketplace** smart contract (which manages the domain names selling and registration) interacts with the Geth smart contract to manipulate the users' balance
- Currently, both smart contracts are running on Sepolia testnet (a semi-production environment), after being tested and developed using Truffle and Ganache

It's noticeable that no centralized backend exists, and that's the core of Decentralized Applications, like GethDomains.

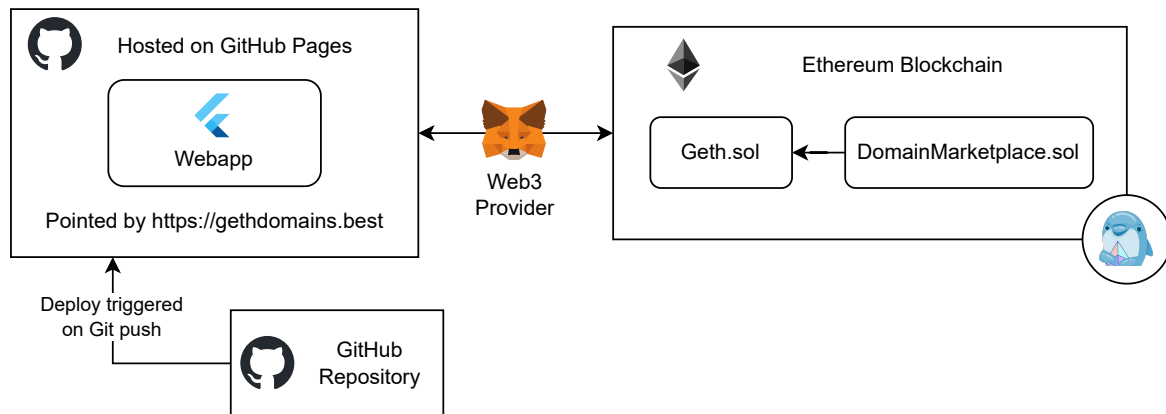
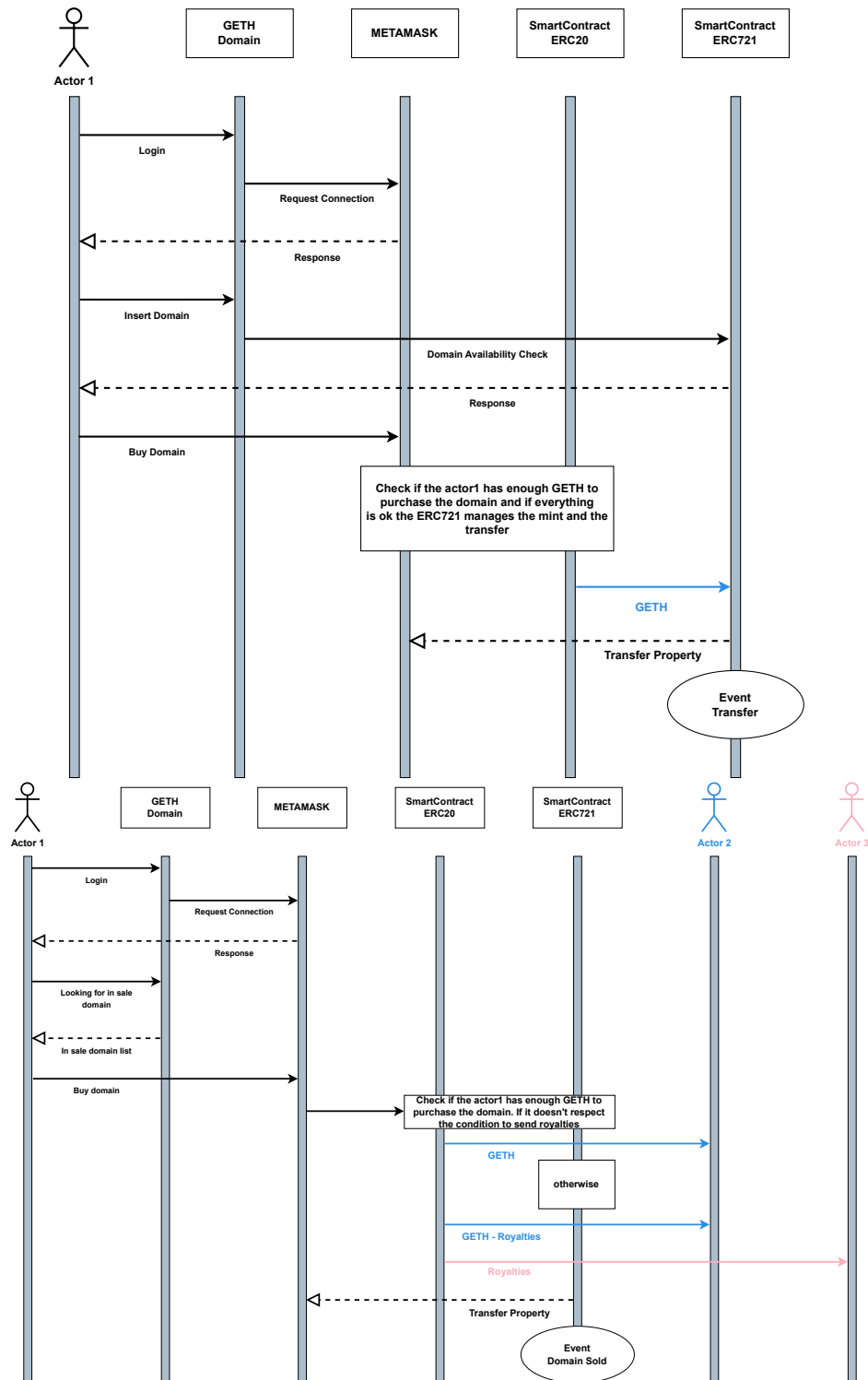
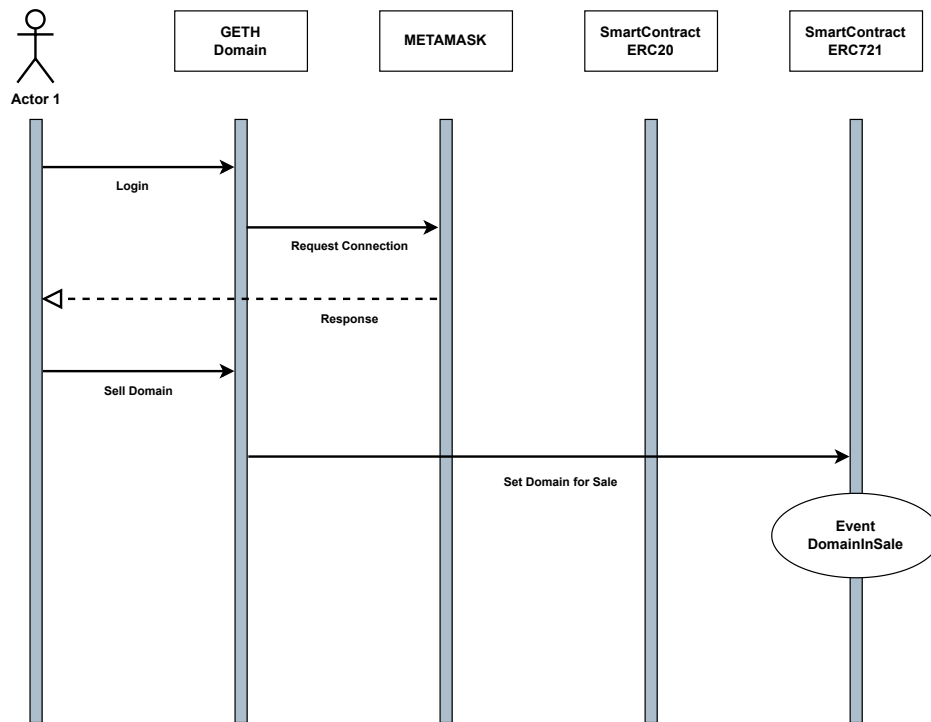


Figure 1: GethDomains DApp general architecture

### 3.1 UML Sequence Diagram

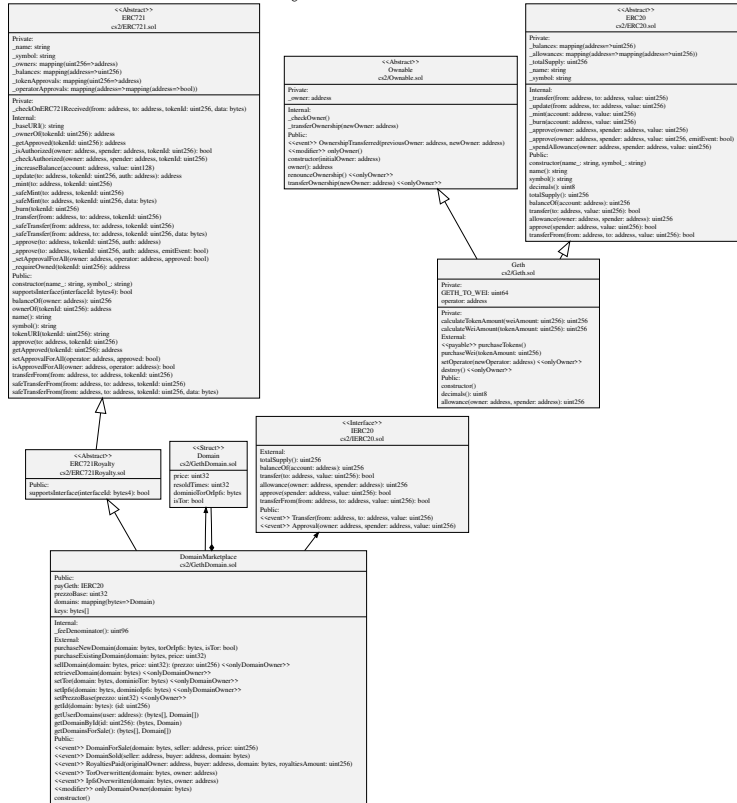
Sequence Diagrams are used to show interactions between the entities of our Dapp. They show the interactions in a time sequence exchange messages. Below there are the sequence diagrams of the most important operation in our Dapp.





### 3.2 UML Class Diagram

A class diagram represents the static structure of a system by depicting the classes, their attributes, methods, and the relationships among classes. It provides a visual representation of the essential components and their **relationships** in the design or structure of a software system.



### 3.3 Smart contracts

The Geth Domain Dapp is based on fungible and non-fungible tokens.

The Geth contract manages the fungible token of the Dapp, used to buy a new or an existing domain. The most important field of the smart contract is the operator, the address of another smart contract that will be allowed to spend geth from the balance of domain buyers. The main functions are:

- **purchaseTokens** is used to purchase Geth tokens with ETH, according to the established rate.
- **purchaseWei** is used to withdraw ETH with Geth tokens, according to the established rate
- **allowance** is used to establish the amount that the operator can spend from the owner balance. This function has been overwritten to comply with the specification that states that the operator should be allowed to spend geth from the balance of domain buyers.

The DomainMarketplace contract regulates the NFT Domain name marketplace and registration. It follows ERC721 interface for interoperability and extends the abstract contract Ownable to manage operation security and ERC721Royalties to manage the royalties mechanism. Within the GethDomain.sol we define a struct that represents a Domain in our Dapp. The struct is composed of the following fields:

- the **price** of the domain if it is for sale, or zero if not.
- the **resoldTimes** attribute that keeps how many times the domain is sold, to know when royalties should be payable.
- **pointedAddress** is a field containing a bytes pointer.
- **isTor** is a boolean to know which type of pointer is **pointedAddress**.

Within the GethDomain.sol we define some events to notify the clients to update the UI.

- **DomainForSale** is emitted when a user sets his domain for sale by adding a price.
- **DomainSold** is emitted when a user buys an existing domain, previously settled for sale.
- **RoyaltiesPaid** is emitted to notify the creator of the domain that he received royalties by the sale.
- **DomainPointerEdited** is emitted when an user overwrites the field **pointedAddress**.

In the smart contract is defined the **onlyDomainOwner** to check that the caller of a method is the owner of the domain concerned. The DomainMarketplace has some fields such as:

- **gethTokensContract** is an instance of the erc20 used to pay the purchases.
- **newDomainPrice** used to keep a standard prize for purchasing a new domain in Geth.
- **domains** is the mapping of domains' information, given the bytes of an encoded name.
- **domainsKeys** is the list of all the domains' names, used to iterate over the mapping.
- **feeNumerator** is the numerator of the royalty fees fraction.

The constructor of the smart contract is used to set the instance of the `erc20` used to pay the purchases. The main methods of the smart contract are:

- **`purchaseNewDomain`** registers a new domain, given its name and the address it points to. The domain name is encoded as bytes, and then hashed to get the ID of the NFT. The domain is minted to the creator only after checking that the domain is not already registered and that the purchaser has enough Geth tokens to pay the registration fee. The price is set to 0 since it is not for sale and the gets paid are added to the contract balance. The domain is also set to point to the given address.
- **`purchaseExistingDomain`** is used to purchase an existing domain, given its name and price. The domain is transferred to the buyer, and the seller receives the price. The creator of the domain receives its royalties, if the domain has been resold. The function checks that the buyer is not the seller, that he must have enough Geth tokens to pay the price and the consistency between the price displayed in the UI and the actual cost. The domain name must already be for sale, otherwise the transaction will revert.
- **`setNewDomainPrice`** set the price of a new domain, in Geth. This function can be called only by the contract owner thanks to the `Ownable` interface, because of its security implications.

Some functions have the `onlyDomainOwner` because they can be called only by the owner of the domain:

- **`sellDomain`** puts a domain for sale, given its name and the price. The price must be greater than 0, otherwise the transaction will revert.
- **`retrieveDomain`** removes a domain from sale, given its name. The domain must be for sale, otherwise the transaction will revert.
- **`setTor`** and **`setIPFS`** edit a domain, given its name and the Tor or IPFS address it will point to.

It's worth noticing that the smart contracts Solidity source code is documented following the **NatSpec** format, following a standard for documentation.

### 3.4 ERC1155 consideration

We decided to use two smart contracts to manage our tokens instead of using only one implementing `erc1155` because despite it allows grouping multiple transfers of different assets into a single transaction, thereby reducing the overall cost of operations, our functions don't transfer different assets from one user to another. So we preferred this approach to improve readability and reusability.

### 3.5 Reentrancy attack consideration

A reentrancy attack[9] occurs when a function makes an external call to another untrusted contract. Then the untrusted contract makes a recursive call back to the original function in an attempt to drain funds. When the contract fails to update its state before sending funds, the attacker can continuously call the withdraw function to drain the contract's funds. `PurchaseWei` function in `geth.sol` contract could be vulnerable to it but it's not, because it never makes a call to an external contract and most importantly, it follows the Check Effects Interactions pattern [10].

### 3.6 Oracle considerations

In our Dapp a domain can point to an address that may not exist. To avoid the user's mistakes, which means additional cost, we thought to use an oracle that checks the correctness of the pointer. After careful evaluation of our needs and the costs associated with using an external oracle, we have decided not to adopt this approach. Our decision is based on the fact that both the public key extracted by an onion address and the multi-base hash of IPFS address have their checksum. So in Tor is nearly impossible to make a mistake on a character that allows a match with another address. Instead in IPFS the hash is encoded through some base, so the decodification fails if some bytes don't match. In summary, our Dapp user is safe from this point of view, but an external user can still set an invalid TOR or IPFS address and that's on him.

### 3.7 Data encoding and compression

In the blockchain, we must save several types of string data, but reserving as few bytes as possible, in order to save costs. In this section, we'll understand the different encoding strategies applied to achieve this result.

#### 3.7.1 Naive Domain Encoding

The domain name is a string containing all lowercase ASCII letters from 'a' to 'z', ending in `.geth`. Our goal is to shrink it as much as possible, in order to save the smallest amount of bytes possible on the blockchain.

The suffix can of course be removed since it's constant.

The first naive approach that comes into mind is to use 5 bits to encode each lowercase letter. The calculation to reach this conclusion is simple:

$$BitsPerChar = \lceil \log_2(ord('z') - ord('a') + 1) \rceil = 5$$

Every 5 bits, a character is encoded so it's basically a task of reading 5 bits at a time from a Byte stream.

### 3.7.2 Huffman Coding

This coding, discussed theoretically in section 1.7, it's practically applied to compress domains. Typically, Huffman guarantees the best compression possible, chunking the input stream every single byte, but it must also calculate the Huffman tree on the input to compress, and it must be stored. Storing this additional piece of data is absolutely counterproductive for our goal, but the intuition comes: the prior probability (or frequency) of a character appearing in the input text (i.e.: the domain name to compress) may be assumed to be very similar with respect to the average domain name already existing on the web.

We have collected data about the domain names registered on the standard web on 6 days, using a third-party service. More than 1.3 million domain names have been collected. They have finally been concatenated and the Huffman tree generation algorithm was run on that text. The resulting tree is the one that is actually used for Gethdomains compression.

However, it works because is based on the strong assumption that the characters in the user input domain name follow the same probability distribution. This is not always the case, so in those cases, we fall back to the 5 bits encoding we have seen earlier. The first bit (not byte) in the resulting byte array indicates the encoding used.

### 3.7.3 Tor address encoding

An Onion v3 address is basically a concatenation of the 32 bytes ECDSA public key of the hidden service, with its checksum and address version number.[11]

We can simply decode and extract the relevant 32 bytes, storing just them.

The onion address will be easily recalculated client-side by calculating the checksum and concatenating the various parts, encoding all of them as Base32 without padding. This will make the original 62 characters long address, just 32 bytes.

### 3.7.4 IPFS CID

The CID (Content Identifier) is a special string derived from the hash of the file. It exists in many forms, but since v0 is always convertible to v1, we will focus only on this one.

It is a concatenation of the following elements:[12]

```
<cidv1> ::= <multibase-prefix>
           <multicodec-cidv1><multicodec-content-type>
           <multihash-content-address>
```



It is possible to process it as follows:

1. Transform it from v0 to v1, if not already
2. Decode the v1 string, according to the first character which indicates the Base that encodes the rest (according to the Multibase specification)
3. Throw away the first decoded byte, which is always 1 in the v1 CID version

While it's true that we have lost forever which was the original base that encoded the output or the version of the CID, these pieces of information are actually useless: re-encoding it in any base we want, in v1 version, will make it work just fine because of its interoperability.

## 4 Implementation

### 4.1 Frontend webapp implementation

The frontend of this app has been developed using Flutter, written in the Dart programming language, then compiled to run on the web. Flutter is a multi-platform framework to make apps, both on mobile and desktop, including the web, as in this case.

The software architecture of this app follows a Single Page Application pattern, where the webapp is rendered in the browser. Moreover, in this specific case, no HTML or CSS is used. But instead the content is rendered on a web canvas using CanvasKit.

Finally, its interoperability with JavaScript allowed the use of Web3.js APIs as usual.

It is **currently hosted via GitHub Pages** on <https://gethdomains.best>, compiled using a CI/CD pipeline on GitHub Actions at every push on the repository.

#### 4.1.1 Data flow across software layers

The flow of data when communicating with the Smart Contract is interesting and more focused on design patterns such as BLoC (summarized in figure 2):

- the user interface is made up of Widgets, the core element for the UI
- in most of the cases, when a user input is required, the text field is attached to a Reactive Form, with real-time client-side validations to provide a better UX when the user inputs data that is not in the right format
- input data is then collected in a model object, to be sent to the business logic of the application, implemented using BLoCs
- inside of a BLoC is kept the state of the application, such as the current balance, the list of domains currently for sale, and so on; that allows various parts of the webapp to interact with the global state in a reactive and decoupled way
- then data has a first transformation, from a model or an event to simpler data structures to be sent to the Repository layer which is at a lower abstraction level with respect to the actual Smart Contract interaction code
- the Repository will transform data such as domain names from String to the data type that the Smart Contract requires in input, such as byte arrays (in Dart: Uint8List)
- the invocation is now at the level of the smart contract Dart class: this is responsible for the call of the JavaScript code from Dart; for instance, byte arrays and more complex data types must be simplified in order to be serialized then

sent to a function written in a completely different language such as JavaScript (Uint8List may become base64 strings, structs may be sent or received as JSON strings, and so on)

- finally, the call has reached the JavaScript code, where the web3.js API stands and can be used to make the actual feature requested: whatever we had before is just an abstraction layer required to drastically improve the quality of the code thanks to the decoupling of responsibilities, according to the SRP principle

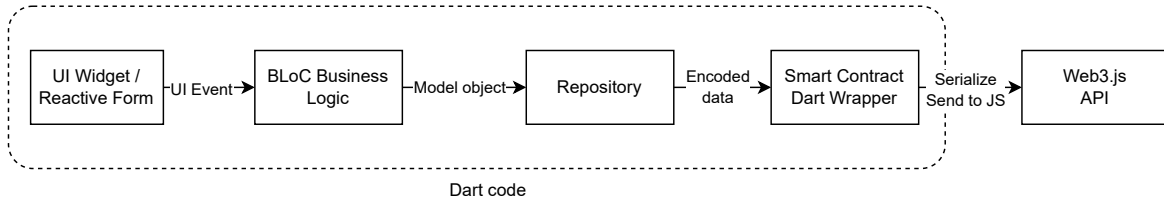


Figure 2: Layered architecture overview

#### 4.1.2 Transaction hash immediately available

As soon as the transaction is sent, a transaction hash is available and it's good practice to inform the user of this result. To do that, on the JavaScript side the asynchronous function call is handled using Promises, which will be translated on the Dart side as Futures, but the conceptual meaning is exactly the same. At this point, a global event of this transaction is propagated and the user sees a nice looking banner, with a link to Etherscan to actually see his transaction. Also, the global state cannot be updated so soon, because the transaction is not mined in a confirmed block yet.

After some time, the transaction will be confirmed and the events emitted will also be received. Only here and now we are able to update the global state informing the user of the success of the action. Updating the global state (contained in a BLoC) will trigger a reactive redraw of elements of the UI affected by the change.

#### 4.1.3 Fees estimation

Right before sending a transaction, a local pre-check is made. The gas fees are estimated calling the corresponding API provided by web3.js (which leverages on the Ethereum provider, often Metamask). The noticeable here is that, in case the smart contract will raise an error and revert its execution given the current local state of the Smart Contract, the user will be notified immediately, without the need to send a transaction that will likely fail.

Moreover, the gas fees estimation is useful and displayed interactively to the user when its input is able to affect the fees required, in order to make it more aware.

#### 4.1.4 Login management

The login procedure, when connected to Metamask or another compatible provider, is basically the request of the permission to read a registered Ethereum account. In case the app can do that right away, the permission was already granted and the user can be considered logged in.

Different story regarding the logout: it is implemented artificially by saving that in the state of the app and making it persistent across refreshes of the page (i.e.: `localStorage` in the browser, but abstracted using `HydratedBLoC`).

#### 4.1.5 Settings

A few settings are available for the user to change:

- the IPFS gateway is one of the most important ones because supposing the anti-censorship goal of the service, in case one provider gets blocked or simply it is down for a while, another one can be freely chosen and used
- to improve the UX of the app, a setting on Dark and Light theme is made: the default setting is to follow the brightness of the operating system and therefore the current browser

#### 4.1.6 Metamask integration

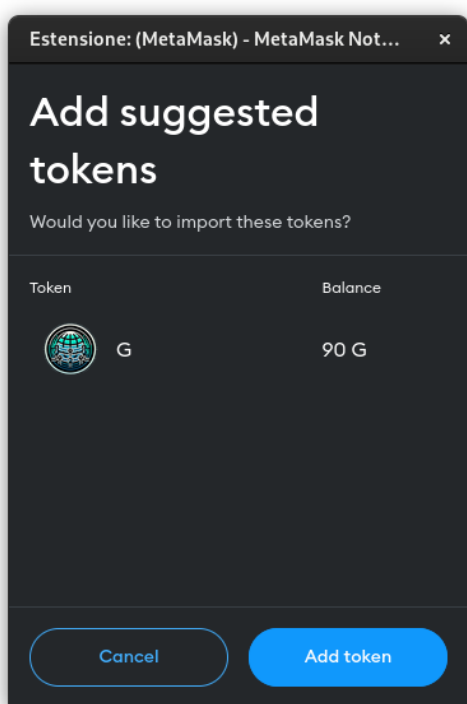
Metamask is considered key in the UX improvement.

After the user buys our token for the first time, a Metamask prompt will pop-up (figure 3) asking to add the ERC-20 token to the wallet directly, via the `wallet_watchAsset` API. It allows the user to have an eye on his balance right from Metamask, but also send and receive tokens directly by interacting with the wallet. Since the smart contract implements the ERC-20 interface and the frontend webapp handles the Transfer events, this approach is totally fine and incentivized by showing the Metamask dialog.

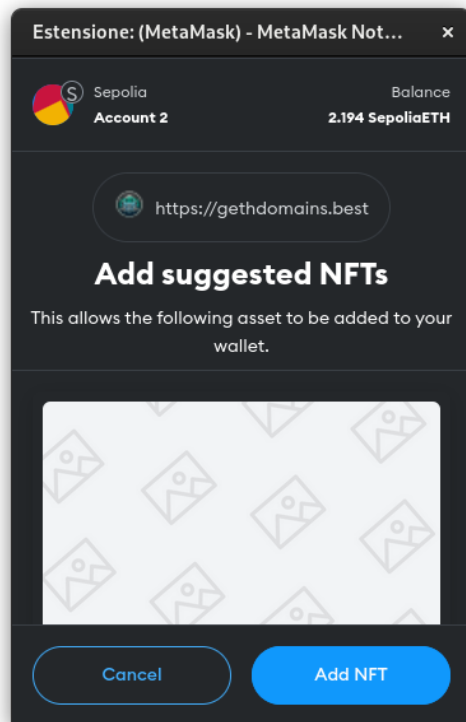
A very similar approach is used for the domain, which is actually an NFT. Its contract implements the ERC-721 interface and the webapp allows the user to add the domain in the Metamask NFT subsection, allowing traditional token exchange outside of the webapp. As before, the webapp handles this situation just fine and in a reactive way listening to the Transfer events emitted by the smart contract.

#### 4.1.7 Encoder library

The logic to encode and decode multiple types of data is considered distinct from the app itself from a code recycle perspective, so it was worth making it a totally separate library.



(a) Prompt for ERC-20 token



(b) Prompt for ERC-721 token

Figure 3: Metamask dialogs

It has the advantage of being reusable and cross-compilable, from desktop apps (on Windows, OS X, Linux) to mobile apps (Android, iOS) but also on the web (both JavaScript or WebAssembly) thanks to the power of the Dart programming language.

In our specific situation, it was also compiled as a CLI app that runs on Linux, for testing purposes when the webapp didn't exist yet (as can be seen in figure 4).

The library offers the logic to encode and decode various data:

- domain names, using a mixed encoding strategy (Huffman trees or using 5 bits to encode each character)
- Tor Onion v3 addresses, extracting the ECDSA public key
- IPFS CID, extracting only the multihash and multicoded parts of the specification

It also provided a command line interface, with a short help page, to make it usable in a simple way.

#### 4.1.8 Future progress for the webapp

As we had the possibility to observe the webapp layered architecture in one of the previous subsections, it's obvious as the only layer in which the actual platform native

```

simone@archlinux:~/Documenti/gethdomains/data_encoder + : - _ x
→ data_encoder git:(main) ✎ ./encoder.exe domain encode claudiodiciccio
Input length: 15
Encoded strings sizes:
- Huffman: 9 bytes
- Five bits encoding: 10 bytes
Compressing with Huffman
Result bytes (hex): e3bac95d4af178c5d0
Result length (bytes): 9
→ data_encoder git:(main) ✎ ./encoder.exe domain decode e3bac95d4af178c5d0
Input length (bytes): 9
Result: claudiodiciccio
Result length: 15
→ data_encoder git:(main) ✎ 

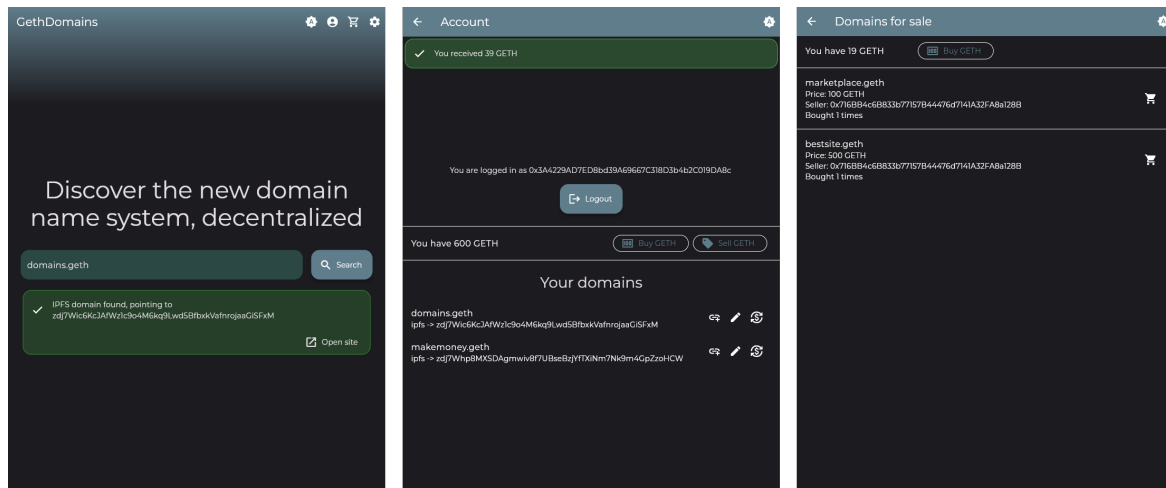
```

Figure 4: Example usage of the encoder library via CLI

implementation shows up is right before calling the JavaScript function. All the business logic and the high level transformations are totally separated and recyclable.

In practical terms, this means that it would be relatively easy to just reimplement the calls to the smart contract using another library (for example a wrapper that calls the Metamask mobile app), then the webapp will immediately be compilable for another operating system, making it an Android or iOS app in little time, just as an example.

### 4.1.9 Screenshots



## 4.2 Smart Contracts

### 4.2.1 OpenZeppelin library

To minimize the risk of vulnerabilities we used the `OpenZeppelin` library which provides heavily tested contracts that implement the interfaces defined by the ERC standards. To represent a domain, a non-fungible token, we implemented the `ERC721Royalty`'s

interface in our smart contract DomainMarketplace. Its interface allows us to represent NFTs and manage their transfer and the possible amount of royalties that the creator of a domain would expect. For the Geth token, used for the purchase of a domain, we extended the `erc20` contract of OpenZeppelin which is used to represent fungible tokens.

#### 4.2.2 Tools adopted

The testing of the smart contracts was made using 2 different environments.

The first one is composed by **Truffle** and **Ganache**. Truffle was used to deploy smart contracts and work on them through the truffle console. To optimize the testing we wrote some JavaScript tests. Ganache was used as a workspace where the changes to the state of the blockchain were visible. The JavaScript tests were made on the main functionalities of the smart contracts:

- For the `GethDomain.sol` the functions called to buy a new domain or an existing domain were tested, checking if the state of the contract and the user account were consistent with the amount of Geth spent, and that the data was stored correctly in the chain. Tests were made on the functions to sell and retrieve an owned domain too, checking the data consistency and the specified constraints on the permissions of these.
- For `Geth.sol` the test consists of purchasing and selling the token, checking the consistency of the balances.

The second one was made through the deployment on **Sepolia** and using **remix** with **metamask** to interface with the testnet. Testnets[13] are blockchains designed to mimic the operating environment of a mainnet but exist on a separate ledger. Sepolia ETH is the currency used to pay to complete transactions on the Sepolia testnet, similar to how ETH is used to pay for computation on Ethereum's mainnet.

## 5 Known Issues and Limitations

During the analysis and development of the DApp, we identified two main limitations closely related to the user experience. Currently, the fact of not having common browsers that can support the opening of such domains registered on the Ethereum blockchain increases the complexity of using the platform making it to the hands of only those more experienced users. Thus, although access to resources saved on TOR or IPFS is simplified, which are usually pointed by hash addresses is still difficult for what is explained above to have an easy mass adoption for all those users less accustomed to blockchain technology.

Secondly, there is a dependence on using digital wallets like Metamask, necessary to authenticate the user within the DApp but binding in performing even the simplest operations such as data consultation as they are deposited within the Ethereum blockchain. A possible compromise would be to save some information outside of it in order not to be tied to the use of wallets but you would lose the main purpose of the DApp.

## 6 Conclusions

Finally, we are aware that our success is closely linked to the growth of the blockchain and web3 industry. We hope that the evolution of decentralized technologies will advance rapidly, thus helping to eliminate current barriers and facilitate a seamless transition to a more decentralized and inclusive digital landscape.

GethDomains, like Dapp at the forefront, is a tangible example of how the mission of simplifying the interaction with resources on decentralized systems can be realized in a concrete way. By allowing users to autonomously manage their domains on Ethereum, the platform promotes a user-centered vision, highlighting the fundamental concept of decentralization.

Through the architecture of the Ethereum blockchain, GethDomains provides an environment in which the user has complete control over their domains, ensuring unprecedented freedom in the use of digital resources. This approach eliminates the risk of censorship and reduces dependence on third parties, transforming the user into an autonomous actor within the decentralized ecosystem.



## References

- [1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. May 2009.
- [2] Vitalik Buterin. A next-generation smart contract and decentralized application platform. <https://ethereum.org/en/whitepaper/>, 2014.
- [3] Fungible tokens (ft). <https://ethereum.org/en/developers/docs/standards/tokens/erc-20/>.
- [4] Non-fungible tokens (nft). [https://ethereum.org/en/developers/docs/standards/tokens/erc-721](https://ethereum.org/en/developers/docs/standards/tokens/erc-721/).
- [5] Fungible and non-fungible tokens (ft/nft). <https://ethereum.org/en/developers/docs/standards/tokens/erc-1155/>.
- [6] Juan Benet. IPFS - content addressed, versioned, P2P file system. *CoRR*, abs/1407.3561, 2014.
- [7] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation onion router. In *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA, August 2004. USENIX Association.
- [8] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
- [9] Reentrancy attack. <https://hackernoon.com/hack-solidity-reentrancy-attack>.
- [10] Checks-effects-interactions pattern. <https://docs.soliditylang.org/en/v0.6.11/security-considerations.html>.
- [11] The Tor Project. Tor rendezvous specification - version 3. <https://github.com/torproject/torspec/blob/8961bb4d83fccb2b987f9899ca83aa430f84ab0c/rend-spec-v3.txt>, 2015.
- [12] Protocol Labs Inc. Cid (content identifier) specification. <https://github.com/multiformats/cid/blob/603f4570ba051192224dd1a3b131a6ebdd486b4f/README.md>, 2020.
- [13] What is the sepolia testnet? <https://www.alchemy.com/overviews/sepolia-testnet>.