



Smart contract based decentralized voting system

Luka Dimnik

Haaga-Helia University of Applied Sciences

Bachelor's Thesis

2022

Bachelor of Business Administration

Abstract

Author(s) Luka Dimnik
Degree Bachelor of Business Administration
Report/thesis title Smart contract based decentralized voting system
Number of pages and appendix pages 38
<p>This project has started out of the need for secure voting system which minimizes the need for trust regarding the party conducting the vote operation as well as other participants. The outcome of the project is intended as a proof of concept for the application of blockchain voting system on a small organizational level with minimal overhead. This new voting system should also eliminate the need for external database dependency for authentication.</p> <p>In the theoretical part of the thesis reader can get an understanding of the concepts that make blockchain development possible. Concepts such as blockchain, smart contracts, NFTs, Ethereum, cryptocurrencies and more are explained in moderate detail.</p> <p>Thesis goes in to details and describes the technologies that were chosen to complete the project and what was the rationale behind the decision.</p> <p>Practical part of the thesis describes the architecture of the application and describes the details of setting up the local development environment. It continues with describing the process of implementing the smart contracts and user interface. Smart contracts were implemented with solidity, Openzeppelin, NodeJS, Hardhat and ethers and user interface was implemented with ReactJS, ethers and Metamask wallet.</p> <p>Results and outcome of the project are discussed at the end of the thesis. Project provided a good outlet for the author to gain knowledge about the emerging field of blockchain development. Project achieved the initial objectives of the thesis. Outcome of the project was a functional decentralized application deployed on the Ethereum testnet with a functional UI and an access token NFT as a form of authentication. Smart contracts were never deployed to the mainnet due to high Ethereum fees which were also stated as one of the reasons why Ethereum smart contract platform is not yet ready for mass adoption.</p>
Keywords ethereum, smart contract, blockchain, solidity, hardhat

Table of contents

1	Glossary	1
2	Introduction	2
2.1	Objectives	3
2.2	Scope.....	3
3	Blockchain and cryptocurrencies: an overview	4
3.1	Block.....	4
3.2	Consensus algorithm.....	5
3.2.1	Proof of work.....	5
3.2.2	Proof of stake	5
3.3	Blockchain use cases.....	6
3.4	Smart contracts	7
3.5	Wallet.....	7
3.5.1	Wallet address	8
3.5.2	Transactions.....	8
3.6	Solidity	8
3.7	Ethereum and cryptocurrencies.....	9
3.7.1	EIP – Ethereum improvement proposals	10
3.7.2	Ethereum virtual machine.....	10
3.8	NFTs.....	11
4	Methods	12
4.1	ReactJS	12
4.1.1	Virtual DOM.....	12
4.1.2	One way data flow.....	12
4.1.3	JSX	13
4.1.4	Props and state	13
4.2	Typescript	14
4.3	NodeJS	14
4.4	Hardhat	14
4.5	Ethers.js.....	15
4.6	MetaMask	15
5	Implementation.....	16
5.1	Local development environment.....	17
5.2	Access NFT smart contract	19
5.3	Voting smart contract	22
5.4	Smart contract deployment and verification.....	25

5.5 UI	30
6 Results and discussion.....	33
7 Conclusion	35
8 References.....	36
Appendix 1. Ballot smart contract Etherscan address.....	39
Appendix 2. Access token smart contract Etherscan address.....	39
Appendix 3. Smart contract source code	39
Appendix 4. Front end source code	39

1 Glossary

- RPC – Remote procedure call. Protocol that enables communication with Ethereum blockchain and outside applications. It provides programmatic way for developers to interact with Ethereum network such as querying the blockchain, executing smart contract code and signing transactions.
- NFT – Non-fungible token. Unique digital token that exists on the blockchain. It can be used to represent various types of digital media, virtual real-estate etc. They are created with smart contracts and provide a secure and transparent way to verify ownership and transfer of the assets.
- IPFS – Inter-planetary files system. Protocol and network for decentralized storage and access of files. It can be integrated with Ethereum blockchain to provide storage for decentralized applications.
- EVM – Ethereum virtual machine. Runtime environment for executing smart contracts on Ethereum blockchain. It runs on every node in the Ethereum network and is capable of storage, executing compiled code and therefore running decentralized applications.

2 Introduction

There has been a lot of interest and speculation in recent years about this new and emerging technology called blockchain. There are areas where blockchain has been proven useful and has been adopted on a massive scale such as cryptocurrency and online gaming. Cryptocurrency has been around since 2008 and its popularity is only increasing with each passing year but in the last couple of years blockchain games gained a lot of adoption. Blockchain games allow players to earn cryptocurrency by playing the game and it also enables virtual assets to become truly unique or rare. Other areas where adoption is in early stages but there is a lot of potential are supply chain management for tracking items, identity verification, real estate for recording property transfers, healthcare and voting.

The best way to find out about the potential viability of the technology for solving a problem is by experimenting and testing. Because of its transparency, censorship resistance and other qualities blockchain lends itself as a very convenient tool to help us bring about transparent, trustless, and convenient voting system. In this project I won't be attempting to tackle political voting on a mass scale. For that kind of feat this technology is perhaps too new and not yet widely accepted or understood. Instead, I will rather try to see if I can use this technology to bring about voting system on a smaller scale.

This project has started out of the need for secure voting system which minimizes the need for trust regarding the party conducting the vote operation as well as other participants. The idea is to use this new emerging technology to bring about system with minimal overhead, high degree of security and anonymity that can be used in organizations of various sizes and orientations.

The thesis will be beneficial to all students and non-students who have interest in blockchain, smart contracts, and security. It may also serve as proof of concept and inspire some organizations in to adopting this approach. As a professional software developer, it is a part of my job to stay educated about new technologies and their applications and this project will serve as a good starting deep dive into the topic of blockchain.

The task of this project is to produce the smart contract voting application with its own identity verification system that would serve as proof of concept for organizations who want to set up a secure, trustless, anonymous voting system in a short time with small overhead.

corrections

2.1 Objectives

Objective of this project is to gain understanding and explain the theory behind the core concepts that power blockchain application and to develop an open source blockchain voting application with a functional user interface that serves as a proof of concept. Voting application should not depend on any centralized database to manage access or internal state. Research and experience from practical work on the project should provide me with sufficient knowledge to critically assess the technical viability of using blockchain voting system in small to medium sized organizations.

To reduce any dependency outside of the blockchain I will use NFTs (non-fungible tokens) for verifying identity on the blockchain. Each participant in the voting will have the possibility to mint their own NFT which will be used to verify themselves to the smart contract.

Application will consist of several components. UI will be built with typescript and ReactJS and the backend will consist of several smart contracts deployed to the blockchain written with Solidity.

This project will provide insight in blockchain theory, how to programmatically interact with the blockchain and how to write smart contracts in solidity programming language and deploy them. If the schedule permits, I might also investigate the decentralized deployment of the frontend with technology like IPFS.

2.2 Scope

The scope of the project does not include a very detailed and advanced styling of the UI. Highly likely is that because of the time constraint end to end integration tests will be missing.

correction

3 Blockchain and cryptocurrencies: an overview

Blockchain as we know and understand it today was invented in 2008 by one person or a group of people under a pseudonym called Satoshi Nakamoto. This technology is only possible due to the previous work done by Dave Bayer, W. Scott Stornetta and Stuart Haber. What made blockchain stand out from the rest of technology with similar goal is that it solved the double spend problem in a decentralized way, meaning that suddenly there was no need for trust in a central authority. The system itself was capable of self-regulation.

In the practical part of the project, I will utilize Ethereum blockchain to build voting application which is both cryptocurrency and smart contract platform therefore theoretical overview will be focused primarily on Ethereum.

3.1 Block

Blockchain is a chain of transaction record collections that are linked together with cryptography. It holds a complete record of all the transactions that were processed (Chuen, 2015). Each individual collection of records is called a block and because they are linked together, it's called blockchain. In general, every block is composed of the header and the body. Figure 1 provides high level overview of the blocks in the blockchain being linked together with the hashes and the structure of individual block.

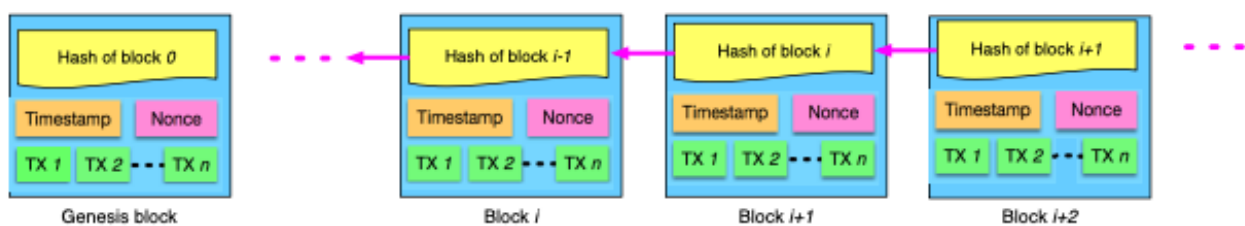


Figure 1. Example of connected blocks that form a blockchain (Zheng et al., 2018)

Block header contains important data about the block itself. It contains timestamp, parent block hash that serves as a link to the previous block, Merkle tree root hash that is a hash of all transactions in the block, block version that specifies what are the block validation rules, nBits and Nonce. Block body contains transactions and counter of transactions. Block size and the size of the transactions dictates how many transactions can be stored in a single block (Zheng et al., 2018).

3.2 Consensus algorithm

In a system of distributed nodes without a central authority we run in to a problem of making the nodes reach a consensus on what is the truth. In computer science this is commonly called a byzantine general's problem. Consensus algorithm is a way of addressing this problem by setting the protocol, incentives, and ideas by which nodes come to a consensus in a trustless way (Zheng et al., 2018). The most well-known consensus algorithms are proof of work and proof of stake.

3.2.1 Proof of work

Proof of work relies on computational power. Nodes in the network are calculating hashes of the block header. Calculated value also called nonce must be smaller or equal than a given value. Nodes are competing by trying to calculate the right value until the target is reached. As soon as one node reaches the target other nodes must verify it. When the value is accepted by other nodes, new block is added to the blockchain and the node who added the new block is rewarded with some cryptocurrency. Nodes that are calculating hashes are called miners (Nakamoto, 2008).

Nakamoto in his whitepaper explains in principle how the proof of work is working but the explanation lacks nuance, context and the negatives that come with it. It turns out that proof of work is terrible for environment due to energy consumption and the consumption of hardware for the computations. There is also a danger of emerging mining pools and centralization. If power is concentrated among small number of mining pools, it can present a real danger to decentralization and security of the whole network.

3.2.2 Proof of stake

Central premise of proof of stake algorithm is that people with a lot of currency are less likely to attack the network. In PoS the validators are staking their capital if they want to participate in block validation. In the example of Ethereum the validator deposits 32 Ethereum. If the validator behaves in bad faith, they are penalized by destroying their staked capital. (Ethereum documentation, 2022). One of the goals of PoS is to eliminate energy consumption problem of PoW in which it is quite successful as it reduces energy consumption to trivial level. In PoS competition between validator nodes is eliminated with random selection of stakeholders that append the block to the blockchain. Selected validator is given the chance to add a block to the blockchain. By doing that the validator is rewarded with the reward in the form of blockchain native cryptocurrency (Saleh, 2021).

3.3 Blockchain use cases

The core functionality of what makes blockchain so useful is that once the data on the block is linked to the previous block it is impossible to reverse it. Blockchain can be viewed as a database that is hosted on a decentralized network of nodes from which you are only able to read and write data but never delete or alter said data.

Most common form of blockchains are hosted and managed on a peer-to-peer network of computers where every node holds a “copy” of the whole blockchain. How those computers or nodes agree on the current state of the blockchain is determined by the consensus protocol. Consensus protocols are a set of rules that nodes can use to determine if the new block is valid.

Blockchain technology is already being used in supply chain management to bring transparency and security to the tracking of goods (Queiroz et al., 2020). One example of this is a startup called Everledger which applies blockchain to diamond supply chain to ensure higher trust in the diamond's features (Lu & Xu, 2017).

Gaming has found a good use case for blockchain because it enables real ownership of virtual items and earning of cryptocurrency through reaching game objectives. In recent years there was a big increase in the popularity of the blockchain games such as Gods Unchained. As participants of these games are investing real currency in these items and there is a high degree of chance involved it is easy to draw parallels between blockchain gaming and gambling (Scholten et al., 2019). While the comparison with gambling has a lot of merit there are numerous examples of mainstream non blockchain games using the same features as loot boxes and in games items that can be purchased with real money.

Blockchain technology has potentials to disrupt online verification. Because of its decentralized nature it enables individuals to verify their identity without a centralized authority (Jamal et al., 2019). In healthcare blockchain could be potentially used to create a transparent and secure system for storing and sharing patient data. This can in turn reduce healthcare costs and improve patient outcomes (McGhin et al., 2019). Blockchain has the potential to bring about increased confidence to voting due to its security and transparency. All voting data can be stored indefinitely and securely. There is an option to protect user anonymity and privacy without sacrificing transparency because individual voters can be represented by an encrypted key. These features could help increase voter turnout and decrease voter fraud (Kshetri & Voas, 2018).

Real estate system for recording property ownership and transfers can be improved with the introduction of blockchain. Security and transparency can simplify buying and selling process and reduce the risk of fraud. While the blockchain shows a lot of promise in the real estate sector the current obstacle is unstandardized data. When real estate data is standardized verification and automation can be greatly increased (Wouda & Opdenakker, 2019).

3.4 Smart contracts

First person to come use the term smart contract was Nick Szabo in the 1990s and the original meaning of the term related to the automation of legal contracts (Szabo, 1997). Four purposes we can think of smart contracts as pieces of computer code that are hosted on the blockchain. Code will execute by itself once certain predefined conditions are met. Since the code is on the blockchain it means that everyone has access to the source code, verify it and check how to interact with the contract.

Smart contracts development promises a lot of potential but currently there are still challenges for developers. They naturally require high degree of security but currently there is little else for developers to ensure safe code than with code reviews and testing. There is a lack of good debugging tools which makes development slow. Solidity the main development language and EVM has numerous limitations. Overall, there is a lack of standards, community support and best practices (Zou et al., 2019).

3.5 Wallet

Cryptocurrency wallet is a software application that enables access to the blockchain. Its two primary use cases are signing the transactions and keeping track of the blockchain assets related to its address. A wallet also contains private keys which are used to sign transactions and as the name implies it is vitally important that private keys are not exposed to unwanted parties. The term wallet is misleading because the wallet does not hold the cryptocurrency itself, it only holds the private keys which enable signing of transactions which in practice means sending and receiving blockchain assets and interacting with smart contracts in some cases (Suratkar et al., 2020). If the user loses access to the private keys, they also lose access to the assets in the wallet indefinitely. Assets cannot be spent without the private keys therefore they are considered lost.

We can differentiate between several different wallet types (Suratkar et al., 2020):

- Desktop wallet is installed on the computer and is relatively secure if the computer does not get hacked.

- Online wallet is a software managed by a third party and can be easily accessed from multiple different devices. It is a very insecure option because third party holds the user's private keys therefore the users put their trust in to third party not to mismanage or steal their private keys and assets in the wallet.
- Mobile wallet is a software app conveniently installed on the mobile device.
- Hardware wallet provides the maximum level of security because the private keys are safely stored on the hardware device and never get exposed to the outside world. It is the most secure wallet option but not the most convenient because every transaction needs to be confirmed with the hardware device. If the hardware device gets stolen or damaged it is possible to restore the wallet instance on another device if the user has the backup of the private keys.

As of the time of this writing the most widely used wallet is MetaMask. MetaMask is installed as a browser extension. It can be used as desktop wallet where users manage their own private keys, or it can also serve as a user interface for the hardware wallet. This wallet will be used throughout the practical phase of the project.

3.5.1 Wallet address

Every wallet has an address associated with it in the form of a long string that is created with cryptography (Suratkar et al., 2020). Address in a blockchain context is a unique alphanumeric identifier of fixed length on the blockchain which is used for sending and receiving cryptocurrencies or other blockchain assets. It is normally created by a cryptocurrency wallet from a public and private key. It works very similar to email address or bank account number.

3.5.2 Transactions

The only way to alter the state of the blockchain is with transactions and the only way to execute transactions is by signing it and paying a network fee which can vary depending on many factors. Transaction is signed with the private key, it is then included in a block and distributed in the network. The receiver can verify the transaction by comparing decrypted hash and the hashed value of the received data (Zheng et al., 2018). On Ethereum transactions are enabled by ERC-777 standardized smart contracts (Dafflon, 2017).

3.6 Solidity

Solidity is a programming language designed for writing smart contracts on Ethereum. It's a high level, statically typed, object-oriented programming language that resembles JavaScript and C++

the most. Solidity code is compiled into machine readable bytecode that is executed in the EVM (Ethereum virtual machine) instance that is run on Ethereum nodes.

Solidity was proposed by Gavin Wood in 2014 who was one of the core Ethereum founders and its first CTO. It was also heavily developed by Christian Reitwiessner and his group. The language is actively being developed and sponsored by Ethereum Foundation and its community.

Language was designed from the beginning to be as user friendly as possible and incorporated syntax that is very familiar to JavaScript developers but with time it became apparent that security is the primary concern of smart contracts and became more verbose.

Solidity is built explicitly for Ethereum virtual machine and is not multi-purpose language like JavaScript or Python. It contains many built-in functions that help with smart contract development.

3.7 Ethereum and cryptocurrencies

Cryptocurrencies emerged on the scene in 2008 with Bitcoin. They are a form of digital money built on top of cryptography. The main selling point of cryptocurrencies is that they remove the need for the centralized authority that can enable, censor, or even revert transactions.

Ethereum like Bitcoin is a base layer decentralized blockchain. It used to use proof of work consensus mechanism but on September 15th, 2022, it already transitioned to proof of stake which reduces its energy consumption by 99.95%. It was invented by Vitalik Buterin and his team in 2013 and it went live in 30th of July 2015. Motivation for Ethereum inception was born out of Bitcoins constraints. Vitalik saw the need for general purpose programmable blockchain and in his view Bitcoin couldn't fulfil the requirements because of its limited scripting language, transaction types, data types and storage.

The main difference to Bitcoin is that Ethereum uses Turing complete programming language called Solidity which allows complex smart contracts (programs) to be deployed on its blockchain which enable all kinds of use cases most notable of which is decentralized finance. Another big difference is that Bitcoin uses UTXO ledger model which stands for unspent transaction output and Ethereum uses account-based ledger model. Both approaches have its positive and negative sides, but account-based model works better with smart contracts and is easier to rationalize due to its similarity with the bank accounts.

Ethereum also allows other fungible or non-fungible (NFTs) tokens to be launched on top of Ethereum by smart contracts which is quite beneficial for them because they don't need to worry about protocol security because they are secured by Ethereum itself.

On Ethereum the most common tokens are fungible tokens based on ERC-20 standard. There are also non fungible tokens on ERC-721 standard which differ from ERC-20 tokens in that every token produced by the smart contract is unique and ERC-1155 tokens which can be either ERC-721 or ERC-20 or combination of the two.

3.7.1 EIP – Ethereum improvement proposals

EIP is a chronologically numbered document intended for the Ethereum community to provide description of a new feature, process, or environment. EIP document should contain a compact technical description and justification for the feature. It is the responsibility of the author to document the opposing views to the EIP and establish consensus within the community (Becze, 2015).

EIP is intended to be a structured and transparent way of submitting ideas for new features and as a base topic for technical discussion within Ethereum community. EIPs are stored as text files in a versioned repository and as such serve as a historical record of all feature proposals.

There are three main types of EIPs. There is an informational EIP that doesn't propose a new feature but present various information to Ethereum community. Next type is meta or process EIP that propose change to some processes in the Ethereum ecosystem but not the core protocol. The last type is the standards track EIP that deals with changes to the Ethereum protocol itself. There are four subcategories of standards track EIP of which the most relevant is ERC. ERC or Ethereum request for comments introduces standards and conventions on the application level. Obvious example is ERC-20 which is a contract token standard (Becze, 2015).

3.7.2 Ethereum virtual machine

EVM or Ethereum virtual machine is at the core of Ethereum protocol. It is a computation platform for executing machine code written in mostly Solidity programming language. It is quite like the way Java virtual machine works. EVM is decentralized computer that enables smart contract deployment and execution (Antonopoulos & Wood, 2019, 7).

3.8 NFTs

The term NFT stands for non-fungible token. It's a type of cryptocurrency created by smart contract where each token is unique, and it is the only such token on a particular blockchain as opposed to fungible tokens like ERC-20 standard where each token is the same and there is no way to distinguish them from each other. This feature makes it very useful for identification on the blockchain. NFTs open all kinds of possibilities for establishing ownership of digital or physical assets, authentication, identification. One could also argue that anything that is off chain is a potential risk and using NFTs to prove something off chain requires some sort of centralized servers or oracles that feed the data to the blockchain and that can be compromised.

Trading volume of digital arts related to NFTs gained significant traction in 2020 and reached its peak in 2021 and first quarter of 2022 with 17 billion dollars. But by September of 2022 it shrank to 466 million which represents 97% drop (Bloomberg, 2022).

It was first proposed in EIP-721 called non-fungible token standard (Etriken, 2018) and developed fully in EIP-1155 called Multi token standard (Radomski, 2018).

Every NFT token on ERC-721 standard is made unique by a tokenId that is uint256 variable which can be used for creating other identifications (Wang, 2021). EIP-1155 introduced another smart contract standard which enables one contracts to create and manage multiple token types. ERC-1155 token can hold the same functionality as ERC-20, ERC-721 or even a combination of the two (Radomski, 2018).

Wang et al. (2021) summarized several desired properties of NFTs:

- Usability. Current ownership information must be available in such a way that other applications or people can access that information.
- Availability. Token is always available to transact and verify.
- Verifiability. It is possible to publicly verify metadata and historical records.
- Transparent Execution. Every NFT action is recorded and publicly available.
- Tamper resistance. Current metadata and historical data cannot be altered.
- Atomicity. Trading can be done in an ACID transaction.
- Tradability. NFT can be traded at any time.

4 Methods

All the necessary tools and services for completing this project will be described in this section.

4.1 ReactJS

For building the client UI I will choose React JavaScript library because it is currently the industry standard and the familiarity of the library which means I don't need to do additional research when developing user interface.

According to its official documentation React is a JavaScript library for building user interfaces. It was developed by Facebook and first released in 2013. Since then, it has seen a massive rise in popularity, and was the most popular web framework in 2021 (StackOverflow, 2021). Most popular application of React framework is applied when building a UI for the SPA (single page application).

Unlike vanilla JavaScript which uses imperative programming style React uses declarative style which means that we state in the code what we want to happen and react will make that happen behind the scenes. This is most reflected in how React renders html elements using JSX language.

The core building block of React are reusable components which are just functions of JavaScript code that can share data between each other.

4.1.1 Virtual DOM

React utilizes virtual DOM (document object data model) which enables changing only part of the DOM and other optimizations without the need to re-render the whole page which greatly improves performance. Virtual DOM is connected with actual DOM using a diff() algorithm and it makes sure that only the nodes that have been changed in the in memory virtual DOM tree end up being changed in the actual DOM (Bhalla et al., 2020).

4.1.2 One way data flow

React is structured as a tree of components. The main data flow happens from top to bottom, from parent to a child component via props. If the data is to be passed from a child to a parent component, we can achieve this in several ways. One way is to pass functions from parent to child which can return values from child components. Another way is to use a third-party state management library like Redux or even Reacts own state management functionality called Context. When the

component state changes upstream it will trigger re-render and re-evaluation of all child components.

4.1.3 JSX

JSX is a syntax extension to JavaScript. Its syntax looks very similar to HTML but that is misleading since JSX compiles to simple JavaScript. In figure 2 we see a code snippet containing JSX code before compilation and below equivalent code after compilation.

```
const element = (  
  <h1 className="greeting">  
    Hello, world!  
  </h1>  
);
```

```
const element = React.createElement(  
  'h1',  
  {className: 'greeting'},  
  'Hello, world!'  
);
```

Figure 2. JSX code before and after compilation (React documentation, 2022)

Like HTML in JSX elements can also contain children. The purpose of JSX is to help developers reason with the UI structure and to abstract the manual creation of elements as it would need to be done in plain JavaScript. It uses declarative programming as we declare what kind of element we want rendered on screen and React creates those elements behind the screen and updates the DOM. Because JSX is just Syntactically different JavaScript we can conveniently use JavaScript code inside JSX inside curly braces, additionally we can also use JSX in our functions, loops and if statements.

4.1.4 Props and state

In react data is being passed around from one component to another with the use of props. This flow is mainly from the parent to a child component but there is also a possibility for child component to pass data to a parent in the function that the child component received as a prop. Props are used to configure React components externally. They are immutable which means that they can't be changed once they are passed to a child component. This feature prevents child components to effect change on their parent (Wohlgethan, 2018).

Components own data management is called state. It contains data specific to the component and is not shared with others. In React it is not recommended to manipulate state directly. State should

only be mutated with the use of `setState` or with the function returned by the `useState` hook (React documentation, 2022).

4.2 Typescript

Because clean and safe code is especially important in blockchain development I will choose Typescript for my main programming language. In recent years I got to work with Typescript extensively and it has proved to provide good developer experience and code with less errors than pure JavaScript.

Typescript is a strongly typed programming language that extends JavaScript. Typescript is so called superset of JavaScript which means that all JavaScript code is also valid typescript code, and all Typescript code gets transpiled to JavaScript before its run. With its JavaScript gets more features like static typing, compile time checks, code editor support and many more. It was developed by Microsoft in 2012. Because of its powerful features Typescript has become indispensable in bigger JavaScript projects. It is also one of the most loved programming languages ended in third place in 2021 Stack Overflow developer survey (StackOverflow, 2021).

4.3 NodeJS

The chosen scripting language of the project is Typescript therefore we need an engine to execute our code. For this task I will choose NodeJS because majority of the documentation and tooling is built for NodeJS.

NodeJS is a JavaScript runtime engine based on Google Chrome JavaScript V8 engine. It enables execution of JavaScript code outside of the browser. It was developed in 2009 by Ryan Dahl. It is mostly used to create server-side applications. Node has JavaScript's own asynchronous, and event driven functionality that becomes useful in data intensive applications. The feature that makes Node highly scalable is its non-blocking nature. It does not need to wait for the request to resolve before continuing with the next task.

4.4 Hardhat

For writing Ethereum smart contracts we will need a special development environment. Hardhat provides all the needed features such as editing, compiling, debugging, and deploying when developing smart contracts locally. The main component is Hardhat Runner which enables management and automating tasks that come naturally with smart contract development.

Tasks and plugins are the core concept of Hardhat runner. If we want to run a task, we can do that from a command line. If we want to compile our Solidity code, we can run compile task like “`npx hardhat compile`”. Individual tasks can be customized and by combining them we can create workflows (Hardhat documentation, 2022).

4.5 Ethers.js

As of the time of the writing there are two primary libraries for interacting with Ethereum blockchain called Ethers.js and Web3.js. I will choose Ethers.js because it has slightly easier high-level interface, it is more modular, and it provides Typescript support.

Ethers.js is an open-source library via MIT license designed for interacting with Ethereum blockchain and its ecosystem. It provides crucial features such as private key security on the client and others (Ethers documentation, 2022).

Ethers API is divided in to four core modules (Moralis, 2021):

- Ethers.Contract: used for deploying and interacting with the smart contracts. Module allows us to call smart contract functions and retrieve information from the smart contracts.
- Ethers.Provider: for changing blockchain state we need signed transactions. Provider module is used for sending transactions and issuing queries.
- Ethers.Utils: used for formatting data and processing use inputs
- Ethers.Wallet: module allows us to connect to a wallet, create new wallets and sign transactions.

4.6 MetaMask

MetaMask is one of the most widely used Ethereum wallets for storing and transacting Ethereum, ERC-20 tokens, additionally it provides user interface when connecting with decentralized applications. It is both a browser plugin and a mobile app. In 2022 one of the most widely used blockchain applications is Uniswap, a decentralized exchange where you connect with MetaMask to exchange Ethereum tokens. Main advantages of MetaMask are simplicity and ease of use, its lightweight since it's not storing Ethereum blockchain, and good Dapp support.

5 Implementation

Entire application will consist of four main components:

- NFT smart contract – enables minting of NFT access tokens which give access to voting function in the voting contract
- Voting smart contract – tracks the voting state and authenticates user.
- MetaMask wallet – holds the access token, Ethereum tokens and enables signing of the transactions
- React application – user interface that connects to MetaMask wallet and interacts with smart contracts

Implementation of the thesis is a decentralized application built on top of the Ethereum blockchain. Application does not have a traditional backend where the bulk of the logic resides instead smart contracts will provide the logic of the application. User of the application can access the React client-side application through any device that runs a web browser. Client-side application will provide user friendly access to the smart application functions. To interact with the voting smart contract, the user needs to be connected to their own MetaMask wallet with which he will sign transactions to execute functions on the smart contract e.g., cast a vote. Before the user can cast a vote in the voting contract NFT access token has to be minted first from the NFT contract. When the user wants to cast a vote the code in the voting contract will check if the MetaMask wallet holds the NFT access token from the NFT contract that was deployed alongside voting contract. If the wallet does not hold the access token the voting operation will be rejected.

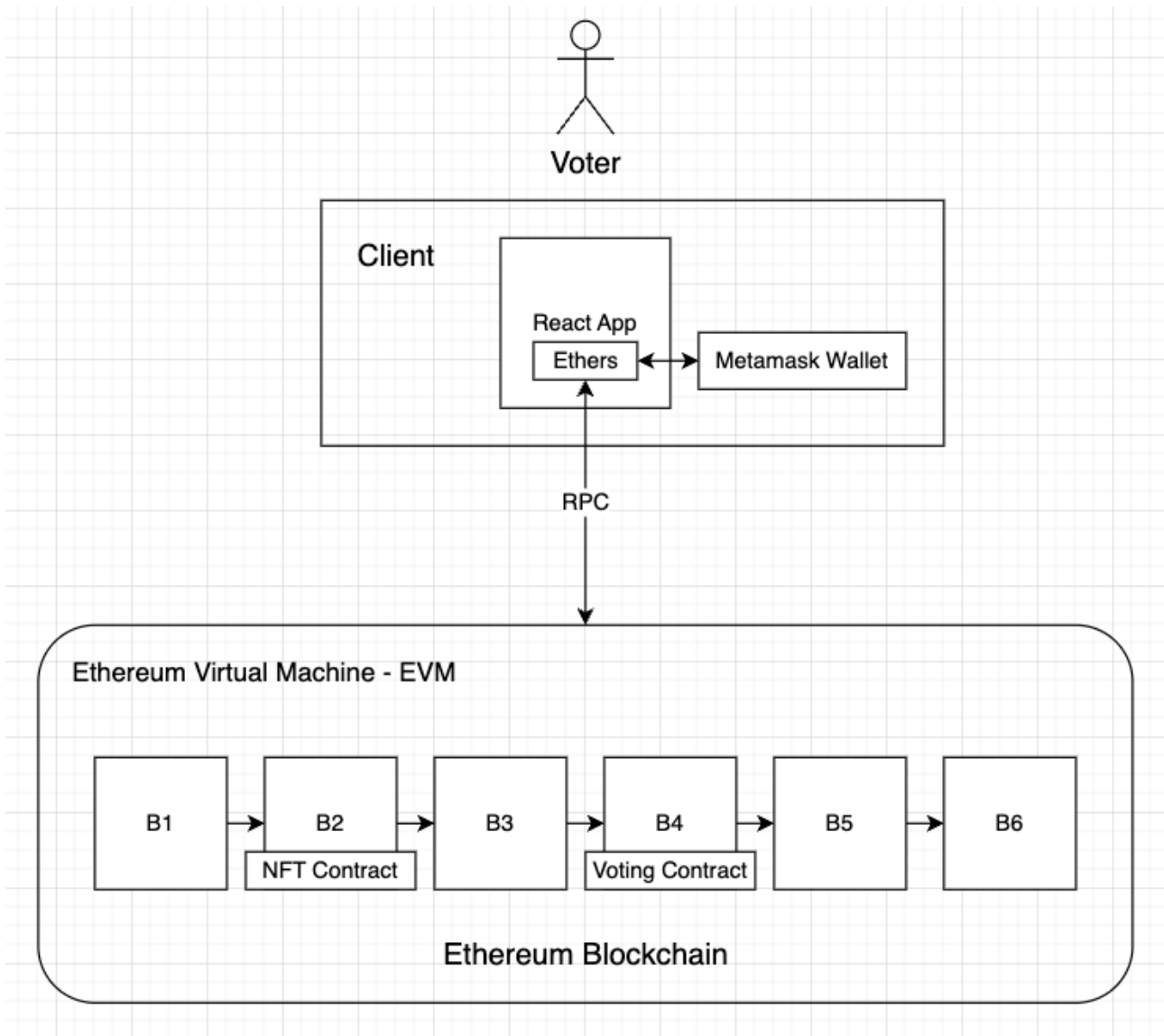


Figure 3. Application architecture

In the Figure 3 we can see a simplified version of Ethereum blockchain composed of linked blocks. NFT and voting contract are deployed separately to different blocks. All the communication between the client-side application and the EVM (Ethereum virtual machine) is done through the RPC (remote procedure call). Ethers.js library is used by the React application to act as a bridge to facilitate communication between the MetaMask wallet and Ethereum blockchain.

5.1 Local development environment

For local development we will use visual studio code editor as it provides a lot of extensions that together make a great blockchain development environment. First, we will install node.js version 18 as this will be a node project. We will initialize node project by running `npm init`. For vs code we will

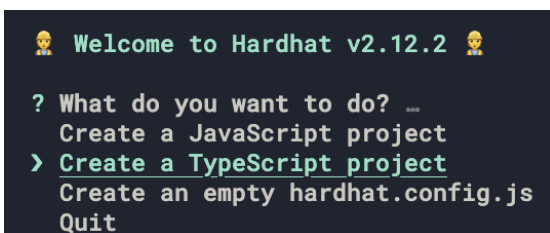
install extension called Solidity + Hardhat as it provides solidity language support with syntax highlighting, code completion and editor integration for hardhat projects.

To communicate with the Ethereum blockchain we must use RPC (remote procedure call) to connect to an Ethereum node. We can construct and send http requests to RPC endpoint directly or we can abstract some of the complexity and use library like Ethers.js that provides far easier interface.

During development we will use wallet private key and API keys which shouldn't be exposed to the public therefore we will create a few local environment variables, store them in .env file and access them in our project with the help of "dotenv" npm package.

To interact with Ethereum test or main network we will need a real EVM wallet with some test Ethereum. For this purpose, we will create a fresh MetaMask wallet and export wallet private key and store it in the .env file.

To complete our development environment, we will also install hardhat local development environment "npm install --save-dev hardhat". Now we can initialize hardhat project by running "npx hardhat". We will be presented by a prompt about what kind of project we want to initialize as seen in Figure 4. We will pick typescript project and we will also agree to install "@nomicfoundation/hardhat-toolbox" which is a bundle of most used hardhat extensions. Hardhat initialization will create hardhat.config.ts file at the root of the project. All the hardhat configuration like extensions, tasks, networks, etc. is done through this file.



```
👤 Welcome to Hardhat v2.12.2 👤  
  
? What do you want to do? ...  
  Create a JavaScript project  
  > Create a TypeScript project  
  Create an empty hardhat.config.js  
  Quit
```

Figure 4. Hardhat initialization prompt

Some of the most helpful things Hardhat does is that it prepares test setup for us, it enables us to run our own node on local machine and offers us a JavaScript console which we can use to send requests to the local node. This is convenient for quickly testing features on the fly without having to write a script. Hardhat toolbox also includes type-chain extension that generates typescript types for smart contracts once they are compiled. We can see how the hardhat.config.ts and the project structure look like at the end of our configuration in Figure 5.

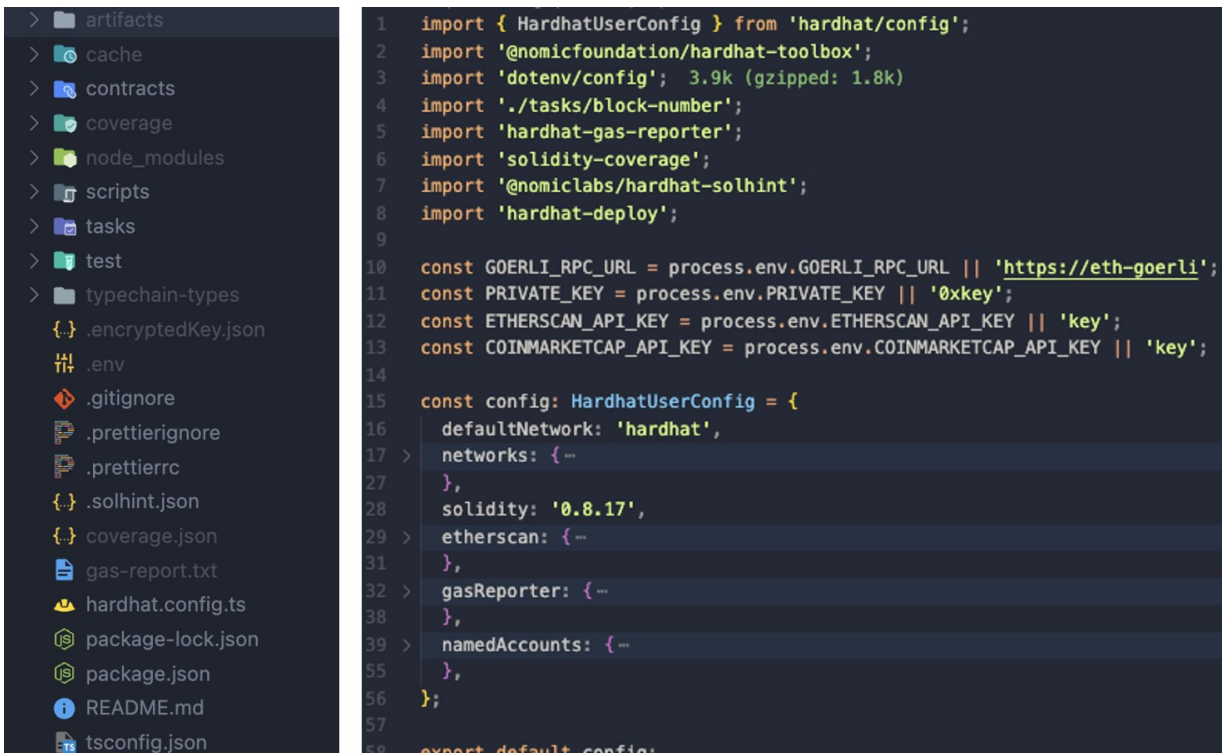


Figure 5. Project folder structure (left) and hardhat configuration file (right)

5.2 Access NFT smart contract

We will use Openzeppelin to give us the basis for our smart contract so that we don't need to write boilerplate. `npm install @openzeppelin/contracts`. By interacting with this smart contract user will be able to mint or create a non-fungible token ownership of which will give the user the right to vote.

First, we specify which solidity version this contract will be using. Then we import the contract template from open zeppelin. When we define the contract, we will extend the ERC721 from the open zeppelin template so that we inherit all the basic functionalities from the template as seen in Figure 6.



Figure 6. Utilizing OpenZeppelin ERC721 contract

Figure 7 shows initialization of “s_tokenCounter” property. It will be a variable that will give a unique token identifier to every minted NFT. We will go with the simple version and will make a token identifier incremented number of every mint execution.

```
1 uint256 private s_tokenCounter;
```

Figure 7. Initialization of s_tokenCounter

In the constructor we specify that this is a 721 constructor. Constructor will take two arguments. First string argument will specify what will be the NFT collection name and the second argument tells the contract what the acronym for the nft collection will be. In the constructor we will initialize s_tokenCounter private variable with zero (Figure 8).

```
1 constructor() ERC721('AccessNft', 'ANFT') {  
2     s_tokenCounter = 0;  
3 }
```

Figure 8. Access token constructor function

Next, we will define a function called mintNft() (Figure 9) which will return uint256 integer. Inside this function we will call _safeMint() function which comes from OpenZeppelin template. _safeMint function will take first argument which will be the address of whoever called the function so that they will receive the nft and the second argument will represent the unique token id which will in our case be s_tokenCounter. The second action of this mintNft function will be the incrementing s_tokenCounter variable by 1 and returning the incremented s_tokenCounter.

```
1 function mintNft() public returns (uint256) {  
2     _safeMint(msg.sender, s_tokenCounter);  
3     s_tokenCounter = s_tokenCounter + 1;  
4     return s_tokenCounter;  
5 }
```

Figure 9. Function for minting access tokens

Next function in this contract is getTokenCounter (Figure 10) which is a public function meaning everyone can access it and it returns current s_tokenCounter.


```

1 function getTokenCounter() public view returns (uint256) {
2     return s_tokenCounter;
3 }

```

Figure 10. Access token counter function

TokenUri is an important function that tells us how this token is going to look like. Token URI (universal resource identifier) turns into URL that returns JSON object like the figure below. We will define the tokenURI function explicitly in our contract to override the same function from the template as seen in Figure 11.

```

1 function tokenURI(uint256 /*tokenId*/) public pure override returns (string memory) {
2     return TOKEN_URI;
3 }

```

Figure 11. Function providing token uri

Every ERC721 token should have the same URI metadata schema. Schema can be seen in Figure 12.

```

1 {
2     "title": "Asset Metadata",
3     "type": "object",
4     "properties": {
5         "name": {
6             "type": "string",
7             "description": "Identifies the asset to which this NFT represents"
8         },
9         "description": {
10            "type": "string",
11            "description": "Describes the asset to which this NFT represents"
12        },
13        "image": {
14            "type": "string",
15            "description": "A URI pointing to a resource with mime type image/*
16            representing the asset to which this NFT represents. Consider making
17            any images at a width between 320 and 1080 pixels and aspect ratio
18            between 1.91:1 and 4:5 inclusive."
19        }
20    }
21 }

```

Figure 12. ERC721 Metadata JSON schema (EntriKen, 2018)

Next, we will write some basic tests to be sure that our contract works correctly. We start with the basic describe block where we will initialize variables `deployer` and `basicNft` because we want them

to be available in all our tests. To prepare for our tests we will use `beforeEach()` function to get the deployer address and the contract object as depicted in Figure 13.

```

1 import { expect } from 'chai';
2 import { deployments, ethers } from 'hardhat';
3 import { BasicNft } from '../typechain-types';
4
5 describe('BasicNft', () => {
6   let deployer;
7   let basicNft: BasicNft;
8
9   beforeEach(async () => {
10    const accounts = await ethers.getSigners();
11    deployer = accounts[0];
12    await deployments.fixture(['basicnft']);
13    basicNft = await ethers.getContract('BasicNft');
14  });

```

Figure 13. Access token test preparation

In our first test we will first mint our nft and then wait for one block confirmation then we get the tokenURI of the minted token and the token counter. First, we want to make sure that tokenCounter increments when we mint a token, second, we want to make sure that minted token has the correct tokenUri as seen in Figure 14.

```

1 it('Allows users to mint and NFT and update counter correctly', async () => {
2   const txRes = await basicNft.mintNft();
3   await txRes.wait(1);
4   const tokenURI = await basicNft.tokenURI(0);
5   const tokenCounter = await basicNft.getTokenCounter();
6
7   expect(tokenCounter).to.equal(1);
8   expect(tokenURI).to.equal(await basicNft.TOKEN_URI());
9 });

```

Figure 14. Testing access token minting functionality

5.3 Voting smart contract

Now that we have nft contract that mints nft access tokens in place we can move on and create a voting smart contract. This contract will contain the logic for creating proposals and following the score and determining who can vote or not.

```

1  contract Ballot {
2      struct Voter {
3          bool voted;
4          uint256 vote;
5      }
6
7      struct Proposal {
8          bytes32 name;
9          uint256 voteCount;
10     }
11
12     address nftContractAddress;
13
14     mapping(address => Voter) public voters;
15
16     Proposal[] public proposals;

```

Figure 15. Construct and properties of Ballot contract

Voting contract will contain two constructs and three properties as depicted in Figure 15. Two constructs are called “Voter” and “Proposal”. We can think of constructs in Solidity as interfaces in TypeScript. It tells us the structure of the object and the types of properties it holds. Voter construct will have property “voted” which is a Boolean and we will simply use it to indicate if a particular voter already cast their vote. “vote” property indicates which voting option voter selected. Proposal construct will contain name property which will be bytes32 type as byte32 type takes less space in memory and we can save a little on gas cost and “voteCount” integer for tracking the amount of votes for the proposal. Below constructs we a property “nftContractAddress” with type “address” and it will store the address of access token contract upon contract initialization. “voters” public property has a type mapping which is equivalent to an object in Java or JavaScript, and we specify that this object will have key object pairs composed of addresses and “Voter” objects. Last property is “proposals” which is an array of “Proposal” objects.

```

1  constructor(bytes32[] memory proposalNames, address _nftContractAddress) {
2      nftContractAddress = _nftContractAddress;
3
4      for (uint256 i = 0; i < proposalNames.length; i++) {
5          proposals.push(Proposal({name: proposalNames[i], voteCount: 0}));
6      }
7  }

```

Figure 16. Voting contract constructor function

Constructor function has two parameters “proposalNames” array of bytes32 and “_nftContractAddress”. Memory keyword indicates that the argument value is only needed for the constructor

execution and can be destroyed afterwards to save storage space. We will loop through “proposalNames” and create an array of proposals and give each proposal a name and a “voteCount” starting at zero. Second parameter “_nftContractAddress” will simply be assigned to “nftContractAddress” property (Figure 16).

```
1 function vote(uint256 proposal) external {
2     require(
3         IERC721(nftContractAddress).balanceOf(msg.sender) > 0,
4         'Does not hold the right access NFT'
5     );
6     require(!voters[msg.sender].voted, 'The voter already voted.');
```

Voter storage sender = voters[msg.sender];

```
8     sender.voted = true;
9     sender.vote = proposal;
10
11     proposals[proposal].voteCount += 1;
12 }
```

Figure 17. Voting function

Our main function for voting will take a proposal represented by integer as an argument and we will use it as an index for accessing the desired proposal in the “proposals” array and incrementing its vote count by one. There are two checks before the proposal vote count can be incremented. First, we will import an interface IERC721 and its function “balanceOf” from OpenZeppelin contracts to check if the address that is calling the function holds at least one token from the access token contract we deployed earlier. If the caller of the function does not hold the token the function will throw an error. Next condition will check if the voter has already voted if it did it will again throw an error. If the checks pass, we initialize “sender” variable with the type of voter and assign it to the newly created property in the voter’s object. Storage keyword here indicates that we want to persist the “sender” variable even after the function execution. Finally, we mark that “sender” has voted and for which proposal (Figure 17).

```

1  function winningProposal() public view returns (uint256 winningProposal_) {
2      uint256 winningVoteCount = 0;
3      for (uint256 p = 0; p < proposals.length; p++) {
4          if (proposals[p].voteCount > winningVoteCount) {
5              winningVoteCount = proposals[p].voteCount;
6              winningProposal_ = p;
7          }
8      }
9  }

```

Figure 18. Function for declaring winning proposal

“winningProposal” is a public view function that can be called, and it will return the index of proposal with the highest vote count. Code can be seen in Figure 18.

```

1  function winnerName() external view returns (bytes32 winnerName_) {
2      winnerName_ = proposals[winningProposal()].name;
3  }
4
5  // will check if the caller has the right to vote
6  function isEligibleToVote() public view returns (bool) {
7      return IERC721(nftContractAddress).balanceOf(msg.sender) > 0;
8  }

```

Figure 19. Function for declaring the winner of the vote (above) and function for checking eligibility to vote (below)

If instead of just the index of the proposal, we want the name we can call a public view function “winnerName” and it will return the name of the proposal in byte32 format. Last function on this contract is “isEligibleToVote” that returns a Boolean value based on the fact if the caller holds the access nft token or not (Figure 19).

5.4 Smart contract deployment and verification

Since the contracts are ready, we can begin implementing the deployment script. We will be deploying to Goerli Ethereum test network. To help us with deployment we will use another hardhat plugin called “hardhat-deploy”. We will install it with the command “npm install -D hardhat-deploy”. This will give our hardhat environment another command called deploy which we will use later.

```

1  import {
2    developmentChains,
3    VERIFICATION_BLOCK_CONFIRMATIONS,
4  } from '../helper-hardhat-config';
5  import verify from '../utils/verify';
6  import { stringArrToByte32Arr } from '../utils/toByte32';
7  import { DeployFunction } from 'hardhat-deploy/types';
8  import { HardhatRuntimeEnvironment } from 'hardhat/types';
9  import { BasicNft } from '../typechain-types/contracts/BasicNFT.sol/BasicNft';
10 import { Ballot } from '../typechain-types/contracts/Ballot';

```

Figure 20. Deployment script imports

For the deploy scripts we will need to import a few packages. Most of the imports are for various types but there are a few utility functions and constants we will be using as can be seen in Figure 20.

```

1  const deployNftAndBallot: DeployFunction = async function (
2    hre: HardhatRuntimeEnvironment
3  ) {
4    // @ts-ignore
5    const { deployments, getNamedAccounts, network, ethers } = hre;
6    const { deploy, log } = deployments;
7    const { deployer } = await getNamedAccounts();
8    const waitBlockConfirmations = developmentChains.includes(network.name)
9      ? 1
10     : VERIFICATION_BLOCK_CONFIRMATIONS;

```

Figure 21. Hardhat runtime environment de-structuring

Deploy function will be an async function that will be responsible for deploying both contracts consecutively. It takes “hre” (hardhat runtime environment) as an argument. We don’t need to worry about passing arguments to deploy function because hardhat-deploy will pass it the whole hardhat development environment for us in the background. From “hre” we will de-structure some properties and methods we will use (Figure 21).

```

1  log('Deploying BasicNft contract...');
2  const basicNftArgs: any[] = [];
3  const basicNft: BasicNft = await deploy('BasicNft', {
4    from: deployer,
5    args: basicNftArgs,
6    log: true,
7    waitConfirmations: waitBlockConfirmations || 1,
8  });
9  log('-----');
10 log('Deploying Ballot contract...');
11 const ballotArgs: any[] = [
12   await stringArrtoByte32Arr(['John', 'Fred']),
13   basicNft.address,
14 ];
15 const ballot: Ballot = await deploy('Ballot', {
16   from: deployer,
17   args: ballotArgs,
18   log: true,
19   waitConfirmations: waitBlockConfirmations || 1,
20 });

```

Figure 22. Defining arguments and executing deployment

First contract to be deployed is the access nft contract. This contract does not need any arguments when deployed so we will leave the args as an empty array. Next, we will use the deploy function which we de-structured from “hre” argument earlier. Deploy will first take a contract name, and a deploy options object as arguments. In the options argument we will specify the address of the deployer, arguments for the contract, how many block confirmations to wait before proceeding and that we want to log results of the deployment. We will repeat similar set of instructions for deploying the Voting contract with one important difference. We want to pass two arguments to the Ballot contract. First will be the array of proposal options and the second will be the contract address of the nft access token contract (Figure 22).

```

1  if (
2    !developmentChains.includes(network.name) &&
3    process.env.ETHERSCAN_API_KEY
4  ) {
5    log('Verifying BasicNft contract...');
6    await verify(basicNft.address, basicNftArgs);
7    log('Verifying Ballot contract...');
8    await verify(ballot.address, ballotArgs);
9  }
10 };
11
12 export default deployNftAndBallot;
13 deployNftAndBallot.tags = ['all', 'nftandballot', 'main'];

```

Figure 23. Conditional Etherscan verification

Before finishing the script, we want to decide if we want to verify the contracts on Etherscan or not. Verified contracts instill more trust as it makes the code publicly available and verifiable. We only want to verify if we are deploying to Ethereum test or main net therefore we want to skip this step if we are running node locally. Verify is a utility function that was created just for this scenario. In the end we use the default export to the function and add the tags. Tags are very useful when we want to customize our deployment. By adding the flag “--tags nftandballot” to deploy command we tell hardhat that we only want to run deploy scripts that have that same tag defined (Figure 23).

```

1  import { run } from 'hardhat';
2
3  const verify = async (contractAddress: string, args: any[]) => {
4    console.log('Verifying contract...');
5    try {
6      await run('verify:verify', {
7        address: contractAddress,
8        constructorArguments: args,
9      });
10   } catch (e: any) {
11     if (e.message.toLowerCase().includes('already verified')) {
12       console.log('Already verified!');
13     } else {
14       console.log(e);
15     }
16   }
17 };

```

Figure 24. Etherscan verification function

We will pass contract address and constructor arguments to verify function. It will run a common hardhat command verify and handle any potential errors (Figure 24).

We can use Etherscan which is a visual tool for browsing the blockchain to get an overview of our contracts (Figure 25).

Contract Overview

Balance: 0 Ether

More Info

My Name Tag: Not Available

Contract Creator: [0xa3137e51614e747daf...](#) at txn [0x3e66af93fc863abe43e...](#)

Transactions Internal Txns Erc20 Token Txns Contract Events

Latest 2 from a total of 2 transactions

Txn Hash	Method	Block	Age	From	To	Value	Txn Fee
0x50dee6f70c17387040...	Vote	7989008	63 days 10 hrs ago	0xa3137e51614e747daf...	0x12901a6217fa4f91341...	0 Ether	0.0001522
0x3e66af93fc863abe43e...	0x60806040	7987859	63 days 15 hrs ago	0xa3137e51614e747daf...	Create: Ballot	0 Ether	0.00637546

[Download CSV Export]

Figure 25. Contract overview on Etherscan

In the contract tab of the we can also see that our verification process was successful (Figure 26). We now ensured that we deployed the exact intended code of the contract to the blockchain. This action also gives the users confidence in the contract because they can read and audit the contract.

Contract Overview

Balance: 0 Ether

More Info

My Name Tag: Not Available

Contract Creator: [0xa3137e51614e747daf...](#) at txn [0x3e66af93fc863abe43e...](#)

Transactions Internal Txns Erc20 Token Txns Contract Events

Code Read Contract Write Contract

Search Source Code

Contract Source Code Verified (Exact Match)

Contract Name: Ballot

Compiler Version: v0.8.17+commit.8d45f5f

Optimization Enabled: No with 200 runs

Other Settings: default evmVersion

Figure 26. Verification successful

5.5 UI

Once the smart contracts are deployed to Ethereum test network we can use the React UI to interact with it. For creating our react app we will use a front-end build tool called Vite. We will initialize our app with the command “npm create vite@latest”. To interact with the blockchain from the client we will use Ethers.js library and install it with “npm install --save ethers”. Final folder structure and UI main page can be seen in Figure 25.

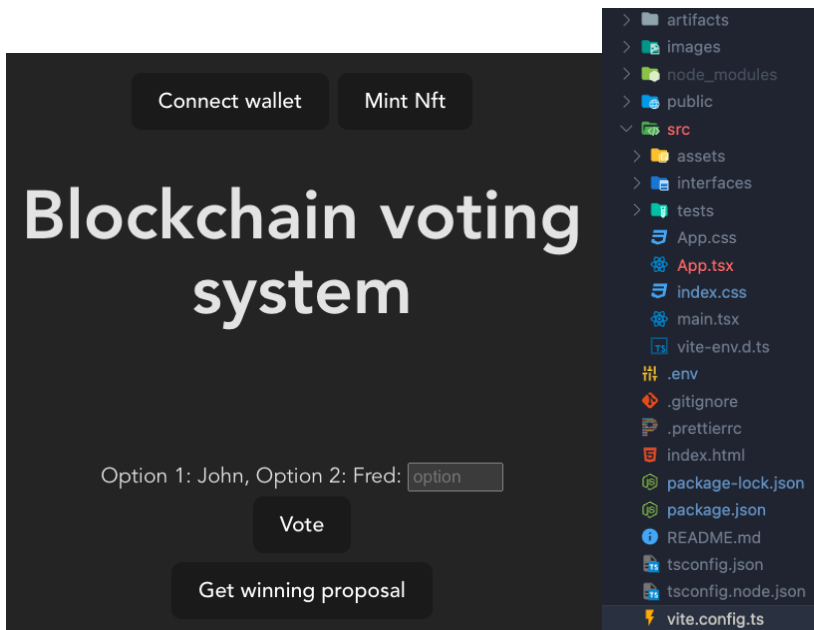


Figure 27. Front-end main page (left) and project structure (right)

To do anything on the Ethereum blockchain we need to sign transactions and to sign transaction we need access to the wallet.

```

1  const requestAccount = async () => {
2    await window.ethereum.request({ method: 'eth_requestAccounts' });
3  };

```

Figure 28. Request client’s wallet account

To connect to the wallet, we can press “Connect wallet” button which will invoke the “requestAccount” function (Figure 26). This function will open the MetaMask wallet in the browser. We can proceed by typing in our password and clicking connect. When connecting we are giving our app the rights to suggest transactions for us to approve.

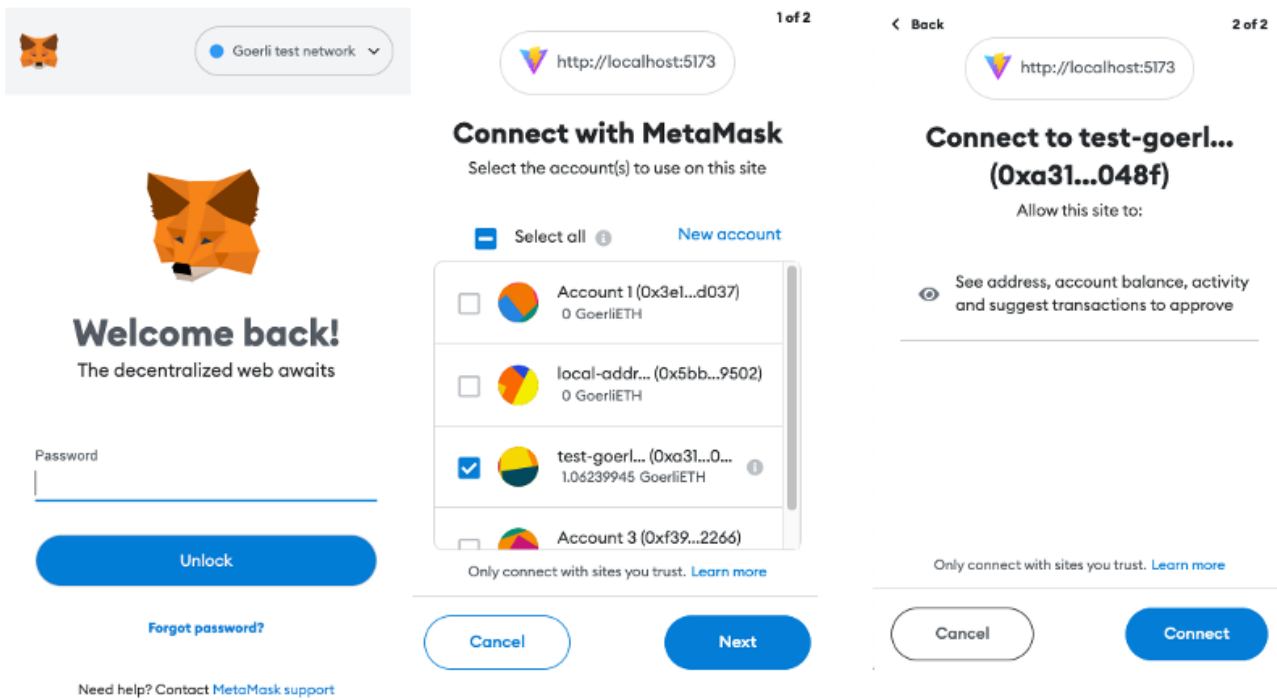


Figure 29. Connecting MetaMask wallet to the browser application

Next step is to mint the access token by clicking on the “Mint NFT” button (Figure 28).

```

1  const mintNft = async () => {
2    console.log('Starting to mint nft...');
3
4    if (typeof window.ethereum !== 'undefined') {
5      await requestAccount();
6      const provider = new ethers.providers.Web3Provider(window.ethereum);
7      const signer = provider.getSigner();
8      const contract: BasicNft = new ethers.Contract(
9        accessNftAddress,
10       BasicNftAbi.abi,
11       signer
12     );
13     try {
14       const transaction = await contract.mintNft();
15       transaction.wait(1);
16     } catch (error) {
17       console.log('Error: ', error);
18     }
19   }
20 };

```

Figure 30. Mint access token from client application

Once the wallet is connected and we have minted the access token we can proceed and vote in the election. We choose the desired option and press vote. Function for voting will look as depicted in Figure 29.

```

1  const vote = async () => {
2    if (!voteOption) return;
3    if (typeof window.ethereum !== 'undefined') {
4      await requestAccount();
5      const provider = new ethers.providers.Web3Provider(window.ethereum);
6      const signer = provider.getSigner();
7      const contract: Ballot = new ethers.Contract(
8        ballotAddress,
9        BallotAbi.abi,
10       signer
11     );
12     try {
13       const transaction = await contract.vote(voteOption);
14       transaction.wait(2);
15       const data = await contract.winningProposal();
16       console.log('Winning proposal data: ', data.toNumber());
17     } catch (error) {
18       console.log('Error: ', error);
19     }
20   }
21 };

```

Figure 31. Voting function from client application

To check the winner of the election we can press the “Get winning proposal” button and the result will be displayed on screen.

```

1  const fetchWinningProposal = async () => {
2    if (typeof window.ethereum !== 'undefined') {
3      const provider = new ethers.providers.Web3Provider(window.ethereum);
4      const contract: Ballot = new ethers.Contract(
5        ballotAddress,
6        BallotAbi.abi,
7        provider
8      );
9      try {
10       const data = await contract.winningProposal();
11       setWinningProposal(data);
12       console.log('Winning proposal data: ', data.toNumber());
13     } catch (error) {
14       console.log('Error: ', error);
15     }
16   }
17 };

```

Figure 32. Fetching winning proposal function from client application

6 Results and discussion

By security we mean that we can be sure that the contract can't be taken down, but we can't be sure that we don't have vulnerabilities in the code itself.

Results of the project are satisfactory. The main goals of the project have been achieved most important of which is delivering smart contract application with a functional UI and authentication based on the access token. The application is secured by the whole weight of the Ethereum blockchain and has no dependance on the external components like databases for example. Smart contracts have been deployed to the Ethereum test network and we can use Etherscan to see all interactions with the smart contracts and that contracts have been verified. Using blockchain hosted access token for voting is a practical solution that removes the need for a database. Potentially that same token can also be used for accessing other contracts of for example certain organization. I can also envision some downsides to this solution. There can potentially be the case of a loss of access to the wallet that holds the token, or the wallet could get compromised and the access token stolen. In this case it would be interesting to explore a possibility to invalidate stolen or lost token and give the user the option to mint a new token.

The project serves as a proof of concept only and it was never deployed to the Ethereum main net due to unacceptable high fees. This illustrates that in its current state Ethereum platform is not yet ready for mass adoption and it is not a good choice for hosting an application such as the voting system. The smart contract platform that will be able to provide speed and security on the same level as Ethereum has currently and is able to scale and keep network fees in check, will be a winner of the future. As of now Ethereum is the market leader for smart contract platforms, and it has a lot of strong features. It has the biggest number of developers, users, development tools, it is environmentally friendly, but it is not able to process large number of transactions with low fees required for mass adoption.

The UI due to the time constraint turned out to be quite simple but that was expected from the beginning. It had the least importance for this proof of concept and the development time dedicated to it was first in line to be sacrificed in case of delays. Useful improvement to the UI could be implementing a feature where the user could select themselves which voting smart contract they want to connect to because currently the address is provided to with the environment variables. Additionally, if the application would be developed for production, some considerable UX improvements would be required.

For the project implementation I always strived to pick the most up to date and popular tools and technologies as that would give me the best opportunity of finding support online once I inevitably hit the roadblock in development. In hindsight this was a good decision since even with using the established tools finding the right information was much harder compared to the traditional web development.

Before the start of the project, I had basic understanding of blockchain technology but no knowledge of blockchain and smart contract development. This led me to spend lots of time researching, reading documentation, and going through tutorials. As a result of this, development was much slower than I originally anticipated. I had to learn how to use certain development tools and a completely new specialized programming language. At the end of the project, I gained a good overall understanding of blockchain development and challenges that come with it. There is also more to blockchain development that was covered during this project such as the use of oracles to feed data outside of blockchain to smart contracts which opens even more possibilities, NFTs with special feature, cross-chain development and more.

Blockchain and cryptocurrency opened the door for a completely new field in software development. This new type of developer is now called blockchain developer and they combine some of the knowledge from traditional front-end web development with a backend that is now powered with blockchain and other tools. This new field also requires a complete shift in the mentality of the developer. The idea of “move fast and break things” that was popularized by the successful Silicon Valley startups must be put aside as it can be dangerous in blockchain space as was demonstrated by multiple examples in the past. More appropriate approach should be more akin to developing critical piece of software like for example software for planes, spaceships, etc. Just like in launching a rocket to space there is very little that can be done once smart contracts are already deployed to the blockchain. Once smart contract is deployed you can't update it. Of course, if vulnerability is found early contract can be scrapped and we could deploy a new one but if vulnerability is found years later and multiple other complex applications are being built on top of the existing one it can be a serious issue.

Suggestion for future development could be development of a DAO (decentralized autonomous organization) based on the unique access token where voting system such as it was presented in this thesis could serve as only one component of a bigger organization.

7 Conclusion

During this thesis, it was demonstrated that it is possible to build a small-scale, transparent voting system quickly with a high degree of security utilizing smart contract platforms like Ethereum where no component of the system is centralized and vulnerable. The biggest obstacle currently for making this kind of applications production ready and widely used are the high Ethereum fees. When Ethereum or some other platform solves the scaling problem applications like the one described in the thesis will become viable.

It is very possible that in the future cryptocurrency will not be the main application of the blockchain but there will be numerous other applications. Number of mainstream smart contract applications will increase dramatically once a couple of big issues are solved. One is the maturity of the smart contract programming languages, Ethereum fees, high degree of oracle reliability and cross-chain communication.

8 References

Antonopoulos, A.M. and Wood, G., 2018. Mastering ethereum: building smart contracts and dapps. O'reilly Media.

Bhalla, A., Garg, S. and Singh, P., 2020. Present day web-development using reactjs. International Research Journal of Engineering and Technology.

Bloomberg. (2022) NFT Trading Volumes Collapse 97% From January Peak. Retrieved October 20, 2022, from <https://www.bloomberg.com/news/articles/2022-09-28/nft-volumes-tumble-97-from-2022-highs-as-frenzy-fades-chart?leadSource=uverify%20wall>.

[Chuen, D.L.K. ed., 2015. Handbook of digital currency: Bitcoin, innovation, financial instruments, and big data. Academic Press.](#)

Ethereum documentation. (2022) The Official Ethereum Documentation. Retrieved October 19, 2022, from <https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/#top>.

Ethers documentation. (2022) The Official Ethers Documentation. Retrieved October 24, 2022, from <https://docs.ethers.org/v5/>.

HardHat documentation. (2022) The Official HardHat Documentation. Retrieved October 22, 2022, from <https://hardhat.org/hardhat-runner/docs/getting-started#overview>.

Jacques Dafflon, Jordi Baylina, Thomas Shababi, "EIP-777: Token Standard," Ethereum Improvement Proposals, no. 777, November 2017. [Online serial]. Available: <https://eips.ethereum.org/EIPS/eip-777>.

Jamal, A., Helmi, R.A.A., Syahirah, A.S.N. and Fatima, M.A., 2019, October. Blockchain-based identity verification system. In 2019 IEEE 9th International Conference on System Engineering and Technology (ICSET) (pp. 253-257). IEEE.

Kshetri, N. and Voas, J., 2018. Blockchain-enabled e-voting. Ieee Software, 35(4), pp.95-99.

Lu, Q. and Xu, X., 2017. Adaptable blockchain-based systems: A case study for product traceability. Ieee Software, 34(6), pp.21-27.

Martin Becze, Hudson Jameson, et al., "EIP-1: EIP Purpose and Guidelines," Ethereum Improvement Proposals, no. 1, October 2015. [Online serial]. Available: <https://eips.ethereum.org/EIPS/eip-1>.

McGhin, T., Choo, K.K.R., Liu, C.Z. and He, D., 2019. Blockchain in healthcare applications: Research challenges and opportunities. *Journal of Network and Computer Applications*, 135, pp.62-75.

Moralis. (2022) JavaScript Libraries – Ethers.js vs Web3.js. Retrieved October 12, 2022, from <https://moralis.io/javascript-libraries-ethers-js-vs-web3-js/>.

Nakamoto, S., 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, p.21260.

Queiroz, M.M., Telles, R. and Bonilla, S.H., 2020. Blockchain and supply chain management integration: a systematic review of the literature. *Supply chain management: An international journal*, 25(2), pp.241-254.

React documentation. (2022) The Official React Documentation. Retrieved October 16, 2022, from <https://reactjs.org/>.

Saleh, F., 2021. Blockchain without waste: Proof-of-stake. *The Review of financial studies*, 34(3), pp.1156-1190.

Scholten, O.J., Hughes, N.G.J., Deterding, S., Drachen, A., Walker, J.A. and Zendle, D., 2019, October. Ethereum crypto-games: Mechanics, prevalence, and gambling similarities. In *Proceedings of the annual symposium on computer-human interaction in play* (pp. 379-389).

StackOverflow. (2022) Stack Overflow developer survey 2021. Retrieved October 15, 2022, from <https://insights.stackoverflow.com/survey/2021#overview>.

Suratkar, S., Shirole, M. and Bhirud, S., 2020, September. Cryptocurrency wallet: A review. In *2020 4th international conference on computer, communication and signal processing (ICCCSP)* (pp. 1-7). IEEE.

Szabo, N., 1997. Formalizing and securing relationships on public networks. *First monday*.

Wang, Q., Li, R., Wang, Q. and Chen, S., 2021. Non-fungible token (NFT): Overview, evaluation, opportunities and challenges. arXiv preprint arXiv:2105.07447.

William Entriken, Dieter Shirley, Jacob Evans, Nastassia Sachs, "EIP-721: Non-Fungible Token Standard," Ethereum Improvement Proposals, no. 721, January 2018. [Online serial]. Available: <https://eips.ethereum.org/EIPS/eip-721>.

Witek Radomski, Andrew Cooke, Philippe Castonguay, James Therien, Eric Binet, Ronan Sandford, "EIP-1155: Multi Token Standard," Ethereum Improvement Proposals, no. 1155, June 2018. [Online serial]. Available: <https://eips.ethereum.org/EIPS/eip-1155>.

Wohlgethan, E., 2018. Supporting web development decisions by comparing three major javascript frameworks: Angular, react and vue.js (Doctoral dissertation, Hochschule für Angewandte Wissenschaften Hamburg).

Wouda, H.P. and Opdenakker, R., 2019. Blockchain technology in commercial real estate transactions. *Journal of property investment & Finance*.

Zheng, Z., Xie, S., Dai, H.N., Chen, X. and Wang, H., 2018. Blockchain challenges and opportunities: A survey. *International journal of web and grid services*, 14(4), pp.352-375.

Zou, W., Lo, D., Kochhar, P.S., Le, X.B.D., Xia, X., Feng, Y., Chen, Z. and Xu, B., 2019. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering*, 47(10), pp.2084-2106.

Appendices

Appendix 1. Ballot smart contract Etherscan address

0x12901a6217Fa4f91341DB6EAeDD5CEd128F8D518 x



Appendix 2. Access token smart contract Etherscan address

0xBe26a0fe741616299C5a75C6E6d20FDD579831AE x



Appendix 3. Smart contract source code

<https://github.com/lukadimnik/voting-system>

Appendix 4. Front end source code

<https://github.com/lukadimnik/voting-system-frontend>