BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
Faculty of Economics and Business Administration
Business Informations Systems

Bachelor's Thesis

Supervisor,
Associate Prof. Dénes CSALA, PhD

Graduate,
Etele TORÓ

2024

BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
Faculty of Economics and Business Administration
Business Informations Systems

Bachelor's Thesis

Decentralized Software Solution for Digital Student Identification

Supervisor,
Associate Prof. Dénes CSALA, PhD

Graduate,
Etele TORÓ

2024

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
Facultatea de Ştiinţe Economice şi
Gestiunea Afacerilor
Informatica Economică

Lucrare de Licență

Soluție Software Decentralizată pentru Identificarea Digitală a Studenților

Coordonator științific,
Lect. univ. dr. Dénes CSALA

Absolvent,
Etele TORÓ

2024

BABEŞ-BOLYAI TUDOMÁNYEGYETEM
KOLOZSVÁR
Közgazdaság- és Gazdálkodástudományi Kar
Gazdaság Informatika

Szakdolgozat

Decentralizált Szoftvermegoldás Digitális Diák Igazolásra

Témavezető,
Dr. Dénes CSALA egyetemi adjunktus

Végzős hallgató,
Etele TORÓ

2024

Abstract

The topic of my thesis is the implementation and presentation of a decentralized, digital application for student verification, utilizing the possibilities offered by blockchain technology. This document encompasses the advantages of this revolutionary technology that emerged just a few years ago, contrasting them with the characteristics of traditional, centralized software systems. The outcome of my research is a standardized smart contract on the Ethereum blockchain, which can be implemented by various universities to verify their own students through inheritance. Additionally, I deemed it important to realize this implementation for my own university, resulting in the UBB smart contract derived from the aforementioned Standard Identification Contract, along with the accompanying React JavaScript web application that provides a user interface for simplifying interactions. The implementation of smart contracts was done in Solidity, which was later deployed on Ethereum-based public and local test networks for cost savings and ease of testing. Throughout the writing of this thesis, I placed great emphasis on clear expression and explanation of technical terms to make it easily understandable and useful for a wide range of readers.

Összefoglaló

Dolgozatom témája egy decentralizált, digitális, diákokat igazoló alkalmazás megvalósítása és bemutatása, a blokklánc technológia által nyújtott lehetőségek kihasználásával. Ez a dokumentum magába foglalja ennek a forradalmi, csupán pár éve megjelent technológiának az előnyeit, szembeállítva a hagyományos, centralizált szoftver rendszerek karakterisztikáival. Kutatásom végterméke standardizált okos szerződés Ethereum blokkláncon, melyet különböző egyetemek implementálhatnak, öröklés réven, saját diákjaiknak igazolására. Mindemellett fontosnak tartottam ezt az implementációt megvalósítani saját egyetemem számára, így végül megszületett az előbbiekben említett Standart Identification Contract-ból származó UBB okos szerződés, és a hozzá tartozó React JavaScript webalkalmazás, mely felhasználói felületet biztosít az interakciók egyszerűsítése érdekében. Az okos szerződések implementátása Solidity-ben történt, melyeket a költségek elkerülése, és a könnyű tesztelhetőségért Ethereum alapú publikus és lokális teszt hálózatokon telepítettem. A dolgozat írása során nagy hangsúlyt fektettem az érthető fogalmazásra és a technikai kifejezések magyarázatára, hogy minél több olvasó számára könnyen befogadható és hasznos legyen.

Contents

Abbreviations

P2P Peer-to-peer

ETH Ethereum

EVM Ethereum Virtual Machine

eWASM Ethereum flavoured WebAssembly

EOA Externally owned accounts

EIP Ethereum Improvement Proposal

ERC Ethereum Request for Comment

PoA Proof-of-authority

DOM Document Object Model

IPFS Inter Planetary File System

CID Content Identifier

API Application Programming Interface

JSON JavaScript Object Notation

SIC Standard Identification Contract

SPDX Software Package Data Exchange

GPL General Public License

BDD Behavior-driven Development

TDD Test-driven Development

npm Node package manager

sol Solidity file extension

CLI Command Line Interface

SDK Software Development Kit

Chapter 1

Introduction: Blockchains in general

In the past few years, Blockchain technology has disrupted industries and sparked a revolution in the way we think about trust and transactions. With a market size projected to reach over $163 billion by 2029 (*The global blockchain market is projected to grow from $7.18 billion in 2022 to $163.83 billion by 2029, at a CAGR of 56.3% in forecast period, 2022-2029...*, 2022), it's no surprise that Blockchain technology is being hailed as one of the most significant technological innovations since the internet. The technology's potential to transform industries such as finance, healthcare, and education, has led to an increased interest in the development of new applications, including decentralized identification systems. In this thesis, we will explore the potential benefits and challenges of this technology and its impact on the future of education and beyond.

1.1 What is a Blockchain?

At its core, a Blockchain is a distributed database or ledger that allows multiple parties to access and maintain a single version of the truth. Think of a reliable global notary public. A Blockchain gives digital transactions a stamp of approval, just like a notary public does for formal documents. But instead of relying on a single notary, Blockchain uses a network of computers to verify and validate each transaction. Each transaction is recorded in a secure, tamper-proof way, and everyone on the network can see and trust the same information. It's similar to having a large network of notaries collaborating to guarantee the accuracy of every transaction without the need for a centralized authority.

But this technology is much more than just a digital ledger. It has the potential to revolutionize how data is stored, shared, and verified, making outmoded conventional middlemen like banks, governments, and businesses. Blockchain's decentralized and unchangeable structure offers a degree of trust and transparency that has

never been seen before, creating new opportunities for efficient and safe business transactions across a range of industries. We will go into the underlying ideas that make Blockchain such a ground-breaking technology in this section (Kube, 2018).

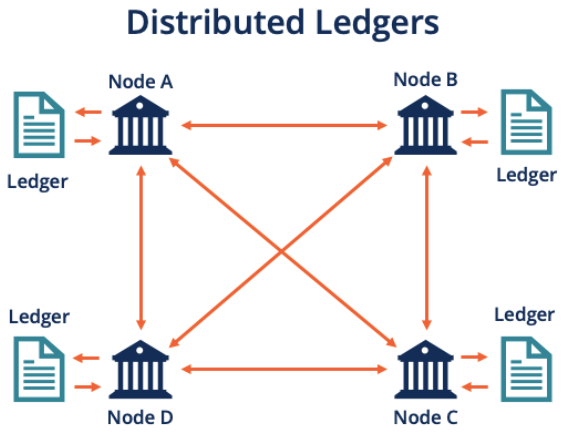


Figure 1.1: Blockchain as a distributed ledger

1.2 Concept of distribution

Even if you are not a mechanican, you probably know that cars comes with different types of engines. Two cars may appear identical on the surface, yet they can differ considerably in terms of performance, security, and so on. Your choice of the engine will have an impact on other characteristics of the car. Understanding the importance of blockchain in the larger picture will be lot easier with this picture in mind (Kube, 2018).

Before implementing a software system, designing it's architecture is a fundamental concern. The two major software architectures are centralized and distributed approaches. In a centralized system the components are connected to a central location or a server. All the data and control are managed by this entity, which has several disadvantages: single point of failure (if the server goes down, the system is unavailable), limited scalability, security risks and lack of flexibility (Tapscott & Tapscott, 2016).

Blockchains and other decentralized systems are intended to disperse control and decision-making among a network of computers or nodes. Each node in a blockchain has a copy of the whole ledger or database, and all nodes collaborate to validate and record transactions. Decentralization is accomplished by a consensus method that will be detailed shortly. Decentralization several advantages, including improved transparency, security, and resilience. It also eliminates the need for

intermediaries or central authorities, which can lower costs and boost efficiency in a variety of industries (Kube, 2018).

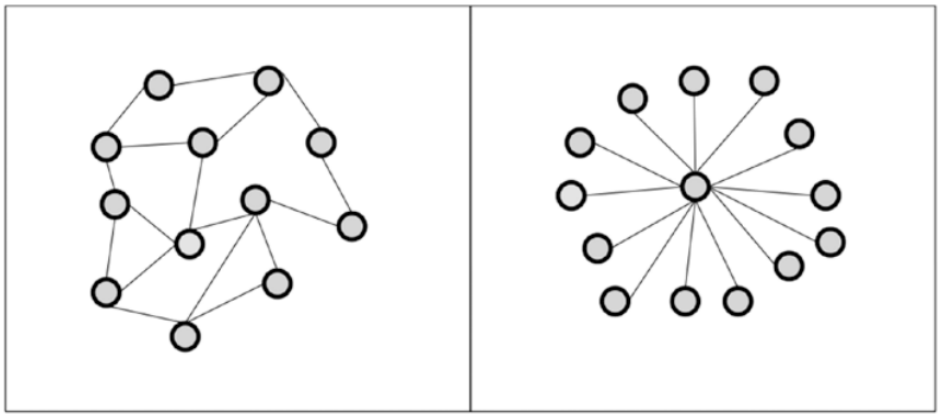


Figure 1.2: Distributed (left) vs. centralized (right) system architecture

To be exact, blockchains are distributed peer-to-peer networks. In a peer-to-peer (P2P) system each node can act both as a client and a server, and communicate directly with other nodes without the need for a central server or authority. In a blockchain, each node communicates with other nodes to validate and record transactions, and each node has an identical copy of the entire ledger. The peer-to-peer nature of a blockchain also allows for a high degree of fault tolerance and resilience, as the failure of one node does not necessarily affect the operation of the network as a whole. Since there is no central point of control, it becomes significantly more challenging for malicious actors to manipulate or tamper with the data stored on the blockchain, ensuring the integrity and immutability of the network(Bambara & Allen, 2018).

As mentioned before, blockchains use a consensus mechanism to maintain and distribute the one true version of the truth. This mechanism simply consists of a collection of rules that specify how the nodes in the network can agree on the current state. Bitcoin blockchain uses Proof-of-Work to achieve consensus on the network. The first node to successfully complete a difficult mathematical puzzle is awarded the right to add the following block to the chain as the nodes compete to validate the upcoming block of transactions. All nodes in the network update their copies to reflect the addition of a new block to the ledger, guaranteeing that everyone is working from the same version of the truth. This procedure is repeated with each new block added to the chain, and the consensus mechanism assures that the ledger is immutable and tamper-proof. In contrast, the Ethereum blockchain currently utilizes a consensus mechanism known as Proof-of-Stake, which will be discussed in the next chapter(Lashkari & Musilek, 2021).

1.3 Immutability and Security

Another remarkable characteristic of blockchain technology is its immutability, meaning that the distributed ledger cannot be altered or tampered with in any way by anyone. Blockchain is an open system, available to anyone and everyone. But like any other public platform, there are a few bad actors who might try to bend the rules to their advantage(Kube, 2018). "The challenge is to keep the system open to everyone yet protect the history of transaction data from being forged or manipulated."(Kube, 2018, p. 136)

On the blockchain, new data is simply added to the existing data rather than being changed. Blockchain transactions are never modified,they are only ever written. We won't have to worry about data manipulation at all if it's impossible to alter the transaction history for every actor, dishonest or not(Kube, 2018).

1.4 Transparency

Transparency is one of the most crucial aspects of blockchain technology. Blockchains are public ledgers that record every transaction and event in real time, and anybody may access the data recorded on the blockchain. For many companies, particularly those that depend on trust and verification, this degree of transparency is revolutionary. All participants can trust one another without the need for intermediaries since blockchain is transparent. Anyone may investigate the transactions that are occurring on the Ethereum blockchain thanks to the availability of numerous tools, like Etherscan, which enables browser-based inspection of all blocks and accompanying transactions. Users can additionally inspect transactions by running their own Ethereum node on their computer. Because of this, customers may independently verify information rather than relying on outside organizations. To assure trust and responsibility, this kind of transparency can be very helpful in industries like supply chain management and healthcare (Solorio, Kanna, & Hoover, 2019).

1.5 Use-cases

Due to its potential to completely revolutionize a number of industries, blockchain technology has recently gained a lot of attention and adoption. Blockchains provide a safe, open, and decentralized platform for recording and validating transactions in a variety of industries, such as finance, healthcare, supply chain management, and voting systems. This section will discuss some of the most exciting applications of blockchain technology and how they are being used to address current issues present in the industries.

Simplified, we can think of the blockchain as a specific kind of box that serves as a general data repository for all types of information. As I mentioned above it is purely distributed peer-to-peer, immutable, append-only, ordered, time-stamped, transparent, secure, and consistent data store, which characteristics open a huge variety of applications. In general, blockchains offer various use cases including proof of existence, proof of nonexistence, proof of time, proof of order, proof of identity, proof of authorship, and proof of ownership (Kube, 2018).

One of the most promising use cases for blockchain technology is record management, using the technology as a secure storage of sensitive data, such as healthcare records. As healthcare data is highly personal and confidential, its storage and transmission pose significant challenges in terms of privacy and security. Blockchain offers a solution by providing a decentralized and immutable ledger that can securely store and manage healthcare records. With blockchain, patients can have more control over their own data, granting access to specific healthcare providers while maintaining privacy and ensuring data integrity. Another area where blockchain technology can be valuable is identity management. Traditional identity systems often rely on centralized databases that are vulnerable to hacks and data breaches. By utilizing blockchain, users can store their identity information in a secure and tamper-proof manner. The decentralized nature of blockchain ensures that no single entity has complete control over the user's identity, reducing the risk of identity theft and fraud. Furthermore, blockchain-based identity systems can enable individuals to have more control and ownership over their personal information, selectively sharing it with trusted parties when needed, all while maintaining privacy and security.

There are many other areas where blockchain can be used to improve the solutions for serious problems, like proving time, order, authorship or ownership. It can serve as a decentralized and trustworthy source for proving the time at which a specific event occurred. For example, in legal proceedings, blockchain can provide a verifiable timestamp for the submission of evidence or the occurrence of an action. In supply chain management, blockchain can be used to establish a verifiable and immutable record of the order in which goods or products were received or shipped. This can help prevent fraud or disputes by providing a transparent and auditable order history. Blockchain can be employed to establish proof of original authorship for creative works such as art, music, or written content. Artists or creators can timestamp their work on a blockchain, providing a publicly verifiable record of their authorship and creation. Blockchain can enable secure and transparent proof of ownership for assets such as property, vehicles, or digital assets. For instance, a digital certificate of ownership for a unique digital artwork can be stored on a blockchain, ensuring the authenticity and ownership rights of the digital asset (Solorio et al., 2019).

Chapter 2

Ethereum Blockchain and Ecosystem

Ethereum is a decentralized blockchain platform founded in 2015 by a young programmer named Vitalik Buterin. The main goal of the platform was to make it possible for programmers to create decentralized applications and smart contracts, which are self-executing contracts in which the conditions of the contract between the buyer and seller are explicitly written in lines of code. With a growing developer community, Ethereum is pushing the boundaries of what is possible with blockchain technology (Solorio et al., 2019).

2.1 Introduction into Ethereum and its key features

The Ethereum platform is based on the same concepts as Bitcoin, but it allows for more than just cryptocurrency transfers. Unlike Bitcoin, which is primarily used as a digital currency, Ethereum offers a programmable platform that enables creators to develop a variety of decentralized applications and services on top of its blockchain. The idea of gas was also introduced by Ethereum, which is a fee paid by users to compensate for the computational resources required to perform transactions and smart contracts on the network. Ethereum also has its own money, Ether (ETH), which is used to pay for gas prices and to motivate miners to process transactions (Antonopoulos & Wood, 2018).

Initially, Ethereum's consensus protocol was proof-of-work, but switching to proof-of-steak has always been on the table due to the significant environmental benefits and the potential to punish malevolent actors with "slashing" (taking their staked ether away). Under the project name "Serenity", Ethereum tries to address the fundamental scalability difficulties by providing many upgrades such as proof-of-stake, sharding, and others. One of the most significant upgrades is the replacement of the present EVM with eWASM (Ethereum flavored WebAssembly). These improvements will enhance the Ethereum network's overall performance and assist

it in overcoming its present scalability difficulties.

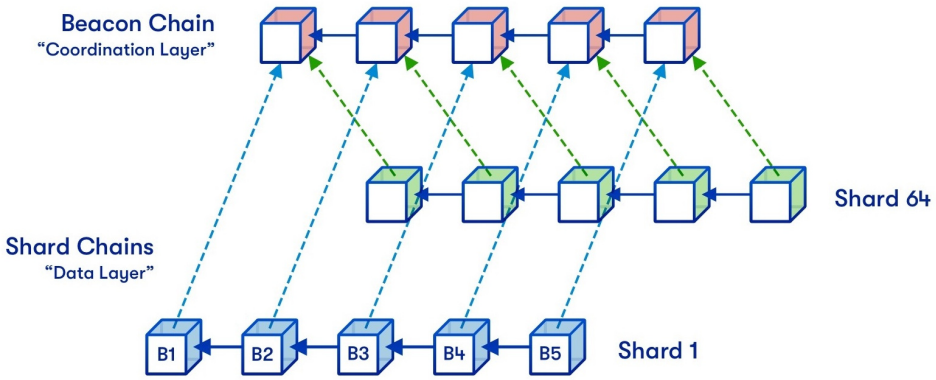


Figure 2.1: Sharding

2.2 Ethereum Virtual Machine (EVM) and its role in executing smart contracts

The Ethereum Virtual Machine, which is meant to be a smart contract execution engine, is the key element of the Ethereum blockchain. All Ethereum nodes run this virtual machine, which processes transactions by running EVM bytecode. The machine code that is executed on a conventional computer CPU is similar to this bytecode. In essence, the EVM makes it possible for smart contracts to be executed in a decentralized manner on the Ethereum Network (Antonopoulos & Wood, 2018).

This virtual machine is like a decentralized world computer, built up with millions of executable objects with each of it's persistent warehouse. These objects are essentially smart contracts that are deployed on the Ethereum network and may be accessed and interacted with by any Ethereum node once they are released. The EVM is a quasi-Turing-complete state machine, which means that it can handle a wide range of computing tasks despite not being really Turing-complete because to gas restrictions. The amount of computational power required to execute a specific smart contract is measured in gas. It is used to avoid spam attacks and to ensure that nodes are compensated for the processing power they supply to the network (Antonopoulos & Wood, 2018).

2.3 Ether and Gas

The native coin of the Ethereum blockchain is called ether (ETH). It's role is to serve as the fuel for the network, the main use of the currency is to pay block creators to

include transactions in blocks, but it is also used to pay for transaction fees, smart contract execution, and other network services. Ether is subdivided into smaller units, the smallest unit is wei, which is a quintillionth of an ether (Solorio et al., 2019).

The price of computational resources needed to execute a transaction or a smart contract on the Ethereum blockchain is known as "gas." It is paid by the transaction sender and is expressed in tiny fractions of ETH. The market demand for computational resources on the network determines gas prices, with greater fees being paid for quicker transaction processing times. The complexity of the operation and the volume of data involved determine how much gas is needed to complete a transaction or smart contract. Gas charges are another method of encouraging miners to complete transactions and protect the network (Antonopoulos & Wood, 2018).

In Ethereum, the term "gas" is used to distinguish between the cost of ether and transaction fees. The separation is essential since it guarantees that transaction fees won't be impacted by ether's price volatility. For the purpose of calculating the price of one unit of gas in wei, each transaction establishes its own gas price. The transactions that give the highest gas price for their computations are those that the block makers, who choose which transactions to include in a block, are most likely to select. This method maintains the network's stability and security while encouraging the effective use of computational resources (Solorio et al., 2019).

2.4 Wallets and Accounts

The management of network interactions depends heavily on Ethereum accounts. Externally owned accounts (EOAs) and contract accounts are the two different categories of accounts. Contract accounts carry the code and storage for smart contracts, whereas EOAs are managed by private keys and can send transactions to other accounts or other contract accounts. Transactions are sent and received using the distinct address assigned to each account. Ethereum accounts contain data storage space in addition to a balance of ether. In general, accounts are crucial elements of the Ethereum network that let users engage with smart contracts and execute transactions.

Ethereum wallets are applications that allow users to securely store, manage, and interact with their Ethereum accounts. Basically this applications are the gateway for users to the Ethereum system. Ethereum wallets come in different forms, including hardware wallets, software wallets, and web wallets. One of the most common wallets is MetaMask, which is a web-based wallet. It is a simple browser extension that runs in your browser, it is easy to use and convenient for testing (Antonopoulos & Wood, 2018).

2.5 Application use cases on Ethereum blockchain

Let's dive into the world of Ethereum and explore some of the most compelling use cases for building decentralized applications on this blockchain. One of Ethereum's most appealing aspects is its ability to ensure confidence in contexts without centralized authority. Ethereum is a great option for trustless environments because it can confirm that all state changes are being made by authorized people using cryptographically signed and validated transactions. The Ethereum blockchain is also public, enabling total transparency in all transactions. This means that all data is open to auditing, which is particularly helpful for applications that call for transparency for customers or business partners. Furthermore, each node contains a complete duplicate of the blockchain data, ensuring that no data is ever lost or destroyed. This adds to Ethereum's unmatched resilience.

Tokens have become one of the most popular use cases for the Ethereum network. These digital assets represent ownership of a particular asset, utility or privilege and can be used to transfer ownership or access to these assets. Tokens are stored on the Ethereum blockchain, which provides a secure and transparent ledger of ownership. To simplify the creation and management of tokens, the Ethereum community has developed various token standards such as ERC-20 and ERC-721 via the Ethereum Improvement Proposal (EIP) process (Solorio et al., 2019). These standards provide developers with guidelines for creating tokens that are interoperable and compatible with other tokens on the network. In the upcoming chapter, I will delve deeper into OpenZeppelin's reusable smart contracts, which are considered to be industrial standard patterns and best practices. These contracts have inspired me to create a standard smart contract for universities to track the identities and states of their students. Now lets discuss the two most common patterns ERC-20 and ERC-721 (Antonopoulos & Wood, 2018).

The ERC-20 standard is widely used for creating fungible tokens on the Ethereum blockchain. These tokens are the alternative solutions for traditional assets like rewards points, airline miles, or currency. ERC-20 tokens are mutually interchangeable, which means that all tokens created from an ERC-20 contract are identical . Therefore, the primary responsibility of an ERC-20 contract is to track balances of these tokens. For example, if a user transfers 100 tokens to another user, the ERC-20 contract would update both users' balances accordingly. Overall, the ERC-20 standard is a powerful tool for creating a wide variety of fungible assets that can be traded between the accounts on the Ethereum network (Antonopoulos & Wood, 2018).

The ERC-721 standard, on the other hand, offers a method for producing non-fungible tokens on the Ethereum network. ERC-721 tokens are distinct from each

other and are not interchangeable like ERC-20 tokens. ERC-721 tokens, for instance, might be utilized to represent assets like digital artwork, collectibles, or even real estate. Each token has a distinct identifier, and the contract is required to keep track of who owns each token. ERC-721 contracts must therefore track the state of each individual token, which makes them more complicated than ERC-20 contracts. ERC-721 is a crucial standard for developing distinctive and valuable digital assets since this increased complexity permits a wide range of new use cases and innovations.

The Ethereum blockchain can also be used for supply chain management. The supply chain, in general, consists of many different elements, including organizations, individuals, processes, and so on. The genesis and the steps that the product through can be tracked in an open and uncorruptible manner using the Ethereum blockchain. All parties participating in the process can have real-time insight over the movement of commodities by logging every stage of the supply chain on the blockchain, from the product's origin to its eventual destination. This can help increase efficiency, reduce costs, and improve consumer safety by quickly identifying the source of any contamination or other issues. Additionally, the use of smart contracts can automate certain processes, such as payment, reducing the need for intermediaries. The immutability and transparency of the Ethereum blockchain make it an ideal solution for creating a secure and efficient supply chain system (Antonopoulos & Wood, 2018).

Chapter 3

Smart Contracts on Ethereum Blockchain

3.1 Bytecode and Opcode

In Ethereum, smart contract code from a higher-level programming language is compiled to bytecode, which consists of a series of opcodes that are interpreted by the Ethereum Virtual Machine. Opcodes are instructions that tell the EVM what to do, such as store or retrieve data. Each opcode instruction has an assigned gas cost, that corresponds to the amount of computing power needed to execute the request. This gas cost is paid by the user who initiates the transaction and is used to compensate the nodes on the network that execute the code. Although as a Solidity developer, it's not essential to understand the contents of the bytecode or opcodes, it can be useful to have a deeper understanding of how the EVM executes smart contract code. Tools like Remix and Etherscan can help developers inspect the bytecode and opcodes of their contracts to optimize gas usage and ensure correct functionality (Solorio et al., 2019).

3.2 Contracts

On the Ethereum network, smart contracts have their own addresses, similar to externally owned accounts have the ability to send and receive ether. However, when a transaction has a smart contract address as its destination, the EVM executes the smart contract using the transaction and its data as input. These transactions can also include which function needs to be executed with the corresponding parameters. Since a contract account lacks a private key, only EOAs can start transactions. However, since contracts can respond to transactions by calling other contracts, complex execution paths can be created. Thanks to this, with the use of smart

contracts developers can create decentralized applications with shared state across the users, where different contracts on the Ethereum blockchain can call each other to maintain the shared state (Antonopoulos & Wood, 2018).

3.3 Smart Contract Languages

Although it's possible to write smart contracts directly in EVM bytecode, the majority of developers find it impractical to construct smart contracts directly in EVM bytecode since it is challenging to read and comprehend. Because of this, the majority of Ethereum developers choose to write their contracts in high-level languages and then translate them into bytecode using a compiler. The writing, testing, and maintenance of developers' code is made much easier by this.

The EVM is a very restricted and fundamental execution environment, where smart contracts function. Additionally, the EVM must be able to access a particular set of system variables and functions. As a result, creating a smart contract language from the ground up is simpler than adapting a general-purpose language for smart contract development. In order to develop smart contracts, many special-purpose languages have been designed. Ethereum offers a variety of languages which are suitable to create smart contracts, along with the necessary compilers for producing bytecode that can be executed by the EVM. Among these languages, Solidity is the most widely used language for developing smart contracts, and also the language of my choice to create my thesis project. Solidity was designed by Dr. Gavin Wood, who is also a co-author of the book "Mastering Ethereum", which is one of the literature from where this thesis was sourced. Other popular languages for Ethereum development are Vyper and Serpent (Antonopoulos & Wood, 2018).

3.4 Solidity

Solidity is a special-purpose language designed specifically for writing smart contracts that can be executed on the Ethereum world computer in a decentralized environment. While its initial development was for Ethereum, its general attributes have led it to be adopted for coding smart contracts on other blockchain platforms as well. The Solidity compiler, solc, is the primary output of the Solidity project, which converts Solidity language programs into EVM bytecode. The project also manages the application binary interface standard for Ethereum smart contracts, which is an essential component that we will explore in detail in this chapter. Every version of the Solidity compiler is accompanied by an ABI specification that defines how contracts are expected to behave and communicate with each other (Antonopoulos & Wood,

2018).

Solidity contracts in Ethereum are structured in an object-oriented manner like Java programming language, where a contract is analogous to a class comprising state variables and functions. Object-oriented programming languages enable reuse of common functionalities through a mechanism called inheritance, where a class can inherit from another class. In Solidity, a contract can also inherit from another contract to reuse its code and functionality. Solidity functions are classified into two types: read-only and write-only. Read-only functions, designated by keywords pure and view, can take input data, read contract data, operate on that data, and return data but cannot change the state of the contract or emit events. Read-only functions in Solidity are similar to web API calls, specifically GET requests, as they do not require any updates on the blockchain. As a result, they are instantaneous and do not incur any gas costs or create transactions. It is essential to note that the ability to skip on-chain updates allows for the efficient execution of read-only functions. Write-only functions, which are the default, require a transaction to execute and can modify the contract's state and emit events. Write-only functions in Ethereum are considered the workhorses of the platform because they are responsible for modifying the state of the contract. However, unlike read-only functions, write-only functions cannot be executed instantaneously and require that the function's data be sent via transaction and included in a block in order for the function to be executed. Additionally, successful write-only methods don't have to change anything, but they typically modify the contract state and often emit one or more events. In the event of an unsuccessful write-only method, the function will revert due to running out of gas or reaching an invalid EVM state, or as a result of an explicit statement in the contract, such as failing a require statement (Solorio et al., 2019).

3.5 The Ethereum Contract ABI

An application binary interface (ABI) is an interface between two software modules that defines how data structures and functions are accessed in machine code. Unlike an API, which defines access in high-level formats as source code, the ABI defines access in low-level formats for machine code (Antonopoulos & Wood, 2018).

In Ethereum, the ABI is used to encode contract calls for the EVM and to read data out of transactions. It is essential for applications to interact with smart contracts. The ABI specifies the functions in the contract that can be invoked, and it describes how each function will accept arguments and return its result. The ABI is specified as a JSON array of function descriptions (Solorio et al., 2019).

Each function description is a JSON object with fields such as type, name, inputs, outputs, constant, and payable. Type refers to the type of the function, such as

function or constructor. Name is the name of the function. Inputs describe the arguments that the function accepts, and outputs describe the values that the function returns. Constant is a Boolean value that indicates whether the function is read-only, and payable is a Boolean value that indicates whether the function can receive Ether.

Event description objects also have fields such as type, name, inputs, and anonymous. The ABI JSON can be used by any application that wants to access the contract once it is deployed. To interact with a contract, an application only needs the ABI and the address where the contract has been deployed (Antonopoulos & Wood, 2018).

3.6 Test chains and local blockchains

The Ethereum protocol is a single standard that can be implemented across multiple networks. Private networks can be created using the Ethereum protocol, similar to how private Internets are called intranets. While private networks deal in ether, the cryptocurrency of the Ethereum network, their ether isn't worth anything on the open market. On the other hand, the ether on the public Ethereum network, has real-world value. The Public Ethereum Network, also called the Ethereum Main Network is the primary network for Ethereum developers. In addition to the main network, there are public test networks that developers use as staging environments. Test networks provide developers with free ether, typically by platforms called faucets, to test their smart contracts (Solorio et al., 2019).

To deploy and execute smart contracts on main network, developers need an account that holds ETH. Acquiring ETH can be done by purchasing it via an exchange, receiving it from a friend, earning it from a business, or mining it themselves. It is important to note that executing smart contracts can be costly in terms of gas, especially when writing to the blockchain. In contrast, reading data from the blockchain is free. Therefore, developers use public or local test networks to develop their applications before deploying them to the Ethereum Main Network (Solorio et al., 2019). In my project, I used the Goerli Public Test Network and a local Ganache test network for testing and development purposes.

3.7 Inspiration: Open-Zeppelin

In this section, we will discuss OpenZeppelin's reusable smart contracts, which helps in developing secure and reliable smart contracts on Ethereum. While exploring these contracts, I was inspired to develop a similar standard contract for universities to identify their students on the blockchain. Similar to ERC contracts,

this standard contract can be implemented by any university, making it a versatile and useful tool for the education industry. Let's dive into the details of OpenZeppelin's smart contracts and explore how they have revolutionized the development of secure smart contracts on the Ethereum blockchain.

OpenZeppelin is a community-driven open-source framework, composed of secure and reusable smart contracts, written in Solidity language. Its primary focus is on security, and it achieves this through industry-standard security patterns and best practices. The framework is led by the Zeppelin team and has over a hundred external contributors. The Zeppelin team maintains the security of the contracts using its experience auditing a large number of contracts, while the community continuously tests and audits real-world applications that use the framework as their foundation (Antonopoulos & Wood, 2018).

OpenZeppelin is the go-to framework for Ethereum smart contracts, boasting an extensive library of contracts that includes implementations of ERC20 and ERC721 tokens, different crowdsale models, and basic contract behaviors like Ownable, Pausable, or LimitBalance. This repository contains some contracts that are officially recognized as standard implementations. Under an MIT license, the framework has been designed with a modular approach for easy reuse and extension. These are clean and fundamental building blocks, perfect for almost any kind of Ethereum project (Antonopoulos & Wood, 2018).

Chapter 4

Concept of an industrial standard for Decentralized Identification System

4.1 The main idea

As I mentioned, the OppenZeppelin's reusable standard smart contracts inspired the project. The aim of the project is to facilitate universities if they decide to take advantage of the benefits offered by blockchain technology for keeping track, verifying, and maintaining the status of their students. To better understand how such a reusable contract works, let's take the ERC-20 token standard implementation offered by OppenZeppelin as an example.

Assuming a person wants to create their own cryptocurrency, has not dealt with this before, and does not want to delve deeper into how to implement such a tool. It is also important for them that there are no errors or security vulnerabilities in the end product. To implement it securely and optimally, they would need expertise, experience, or significant financial resources to trust someone with it. The simplest thing they can do is to implement the ERC-20 fungible token interface, which was introduced by Fabian Vogelsteller in November 2015 as Ethereum Request for Comments (ERC). This interface consists of the required functions that we need in order for the token to perform the minimum expected functions. These functions are: totalSupply, balanceOf, transfer, transferFrom, approve, allowance. Most functions do not require explanation, their task can be inferred from their name, but this is not relevant to us now, so I will not go into more detail about them. In addition to these required functions, the ERC-20 contract also includes the optional name, symbol, decimals functions. There are several implementations of this interface available on the internet, but the two most recognized ones are Consensys EIP20 and OppenZeppelin StandardToken. So there is no need for us to recreate a functioning token, and allocate significant resources to avoid security vulnerabilities and optimization,

as we can simply inherit the functions defined in a smart contract implementing an ERC-20 token interface. In the constructor of our own contract, we must call the constructor of the inherited class and provide the necessary parameters. The ERC-20 constructor expects the token's name, symbol, and the number of decimals.

The easy, secure, and optimal implementation is just one advantage of the ERC-20 standard. Consider that all ERC-20 tokens have the same standard functions and properties such as name, symbol, and total supply. This means that we can perform operations with all similar tokens using the same function calls, enabling us to trade with any similar token on different platforms or perform other interactions. ERC-20 tokens are compatible with all supported Ethereum wallets and exchanges, making it easy for users to manage and exchange them. Tokens can be easily transferred to other Ethereum addresses and can be used easily in various applications.

The digital student ID card system that I have imagined would have similar benefits in terms of implementation. The functions required for the system to operate are implemented in the contract, which universities can inherit to develop their own implementations. This would not only reduce development costs but also provide an opportunity for universities to develop common applications outside their own domain, similar to how decentralized exchanges or wallets work in the case of ERC-20. In the execution of my graduation thesis, I have compiled the Student Identification Contract (SIC), which includes all the functionality I deemed necessary to verify student identities. However, the thesis also includes the UBB Contract implemented for the Babeș-Bolyai University, along with the web application developed for it, which enables students, administrators, and third parties to easily interact with the data stored on the blockchain. But before we delve into the details of the implementation, let's explore why I found it necessary and useful to introduce blockchain technology into the education system.

4.2 Why is Blockchain needed?

Digital identity systems play an important role in both the business world and our personal lives. The use of identity documents can arise in many areas, such as proving one's identity, for banking transactions, when traveling, when starting a new job, and last but not least, when verifying our university status. However, traditional identity systems are usually centralized and often prone to fraud and data loss. To address these challenges, blockchain technology offers the opportunity to securely and decentralize the storage of data. Blockchain is a distributed database system that organizes data into blocks and then links these blocks together to create a constantly growing chain that contains all the data. In the blockchain system, blocks are validated and encrypted using cryptographic algorithms, which makes

the storage and transmission of data secure and reliable. Let's see which properties make this type of student identification system advantageous.

One of the most important features of blockchain technology is security. In the blockchain system, data is authenticated and encrypted, which means that blocks cannot be modified or deleted. The authenticity and integrity of the data are therefore guaranteed, and the data cannot be lost or destroyed. Student verification may be necessary in many situations, such as when applying to universities, submitting applications, using public and private institutions, and many more. In these situations, trust in the presented document is important, and this trust is significantly increased when data is stored in a decentralized, encrypted, and authenticated manner. The stored data in the blockchain cannot be retroactively modified, but the system can include the latest state in the next block. This makes it possible to trace every change in the system. Only the administrator with the account that was used to deploy the smart contract can modify the student's data.

One of the key properties of blockchain technology is decentralization. In a blockchain system, data is stored on multiple computers, which means that there is no central system or database that hackers could attack. The data is verified and confirmed on every block by the computers participating in the blockchain, which collaborate in storing and validating the data. This ensures the reliability and security of the system. Due to decentralization, the system is continuously available and never goes down, in contrast to a central server which, in case of errors it's almost useless. Additionally, it provides faster verification and authentication.

4.3 How would this technology revolutionize education?

Nowadays, more and more foreign companies are using blockchain technology to identify various groups in different areas. One example is the South African company Gravity Training, who have decided to issue their diplomas on the blockchain to individuals who have completed their courses. Gravity Training is a leading provider of commercial work-at-height solutions in South Africa with an international presence in Europe, Africa, the Middle East, and India. They offer various training courses, from Fall Arrest to Fall Protection Plan Development. According to Gravity, fake Health Safety certificates are becoming a growing problem in South Africa, as many people are photoshopping certifications and presenting them to employers. Hiring unqualified professionals with fake credentials for high-risk construction work involving heights, heavy machinery, gas, or mining can have serious consequences, putting lives at risk and making companies liable for accidents. To combat certificate fraud in the Health Safety space, Gravity has started using Dock's decentralized certification platform. Gravity is now able to issue fraud-proof, high-

security digital certificates that can be instantly verified and automate and digitize the issuance process of thousands of certificates (*Gravity eliminates Health Safety certificate fraud with Dock*, n.d.).

Digital diplomas have emerged in recent years as the leading example of how blockchain might be applied to education. The implementation of blockchain transcripts and certificates by Maryville University in 2019 made it one of the first universities to give students and graduates ownership over their academic credentials. The smartphone app Blockcerts Wallet now allows students to maintain their digital credentials. The program uses blockchain technology to offer a verifiable, tamper-proof diploma that is simple to distribute with other institutions and potential employers. The use of blockchain technology to store and share academic credentials, particularly diplomas, benefits students, institutions, and employers because to the permanence, convenience, and security connected with it. (*Maryville University: Digital Diploma*, n.d.).

As we can see, there are several examples of educational institutions leveraging the advantages offered by blockchain technology, giving them significant benefits. But how would my envisioned project differ from these implementations? Primarily, it involves a standard smart contract that is publicly accessible to anyone. However, this contract only defines the necessary functions and is not sufficient for use. The institution must inherit the properties of the SIC, specifying their own university parameters in the constructor. They can then expand the implementation to meet their own objectives, and potentially override the functions found in the base contract.

Secondly, the blockchain solutions currently in use in the education system mostly involve issuing university diplomas on the blockchain. With the contract I have created, universities and institutions can identify not only graduated students but also those who are currently enrolled or suspended. Every user can connect to the application with their Ethereum account or personal ID, and on the homepage, their student cards (a student can study multiple majors/faculties) are displayed with their university status. On the Verify page, third parties such as public institutions can verify students by wallet address or personal ID. They simply need to enter the address or ID into the search bar, and if there is an active, graduated, or suspended student associated with it, the diploma certifying their university status will appear.

Chapter 5

Implementation

We have come to the final and most technical chapter. In this chapter, I will unfold the details of the implementation, starting from the application's foundations and ending with a comprehensive picture of the entire operation in the reader's mind. The chapter is divided into two sections based on the various components of the application, and each section is preceded by a brief introduction to the development environment in which the particular component was implemented. In the first section, I present the development environment I used, the tech stack with which we can implement Ethereum-based smart contracts. Following this, I introduce the SIC and UBB contracts: the Standard Identification Contract and the contract implemented for the Babes-Bolyai University. This concludes the back-end portion of the application, and we move on to the tech stack used in implementing the front-end. I demonstrate how we can communicate with smart contracts deployed on the Ethereum blockchain from node using the contract ABI, and then how we can present this information to users, while focusing on both the simplicity of the user interface and the user experience.

5.1 Ethereum Development Environment

To write smart contracts on the Ethereum blockchain, an EVM-compatible language must be chosen. EVM stands for Ethereum Virtual Machine and is a run-time environment for executing smart contracts on the Ethereum blockchain. The most widely used and supported EVM-compatible language is Solidity, but other languages such as Vyper or Serpent also exist. For my project implementation, I chose the Solidity language because reusable, standard contracts offered by OpenZeppelin, which have become industry standards due to their popularity, are written in this language. To be able to run the smart contract written in Solidity on the Ethereum virtual machine, two steps are required after code writing. We need a

compiler that compiles Solidity into bytecode, which we then need to deploy to an Ethereum client.

For any application development, developers usually use frameworks that greatly simplify the development process. There is a wide range of frameworks available for Solidity development, with the most popular being Truffle Framework, Hardhat, Brownie and Embark. I use Hardhat for project development because it is the most widely used and has the greatest support. When choosing such tools, it is worth choosing from the popular ones because it is easier to find help on forums or developer portals in case of any problems or questions. Hardhat has an internal compiler module, so there is no need for an external tool. As mentioned, the compiled code needs to be deployed to an Ethereum client.

We could also deploy to the Ethereum Main Network at once, but this is strongly discouraged during development. Every interaction that is modifying the blockchain has a gas amount that we need to pay in Ether. Ethereum Main Network's Ether has a value on the open market, so we would need to buy Ether just to deploy the contract. Various other Ethereum clients have emerged for developers. During my project development, I will use two Ethereum clients, so debugging will not incur real costs. I chose the Goerli Public Test network and a Ganache Local network for variety. For testing framework, I chose Chai, which allows comprehensive testing of my smart contract.

5.1.1 Solidity

To write Solidity smart contracts, we need a code editor and the Solidity compiler. It is recommended to use a code editor that has syntax highlighting for Solidity, such as Visual Studio Code with the Solidity extension. The Solidity compiler is used to compile your Solidity code into bytecode that can be executed by the Ethereum Virtual Machine.

When writing Solidity code, it is also recommended to include a "SPDX-License-Identifier" in the source code header. This is a standard way of specifying the license under which the code is released. This is typically included in a comment at the top of the Solidity file and can be used to specify a variety of licenses, such as the MIT license, Apache 2.0, or GPL. Including the "SPDX-License-Identifier" helps to make your code more transparent and encourages reuse and collaboration. After this, the solidity version has to be defined in which the code was written.

```
1 pragma solidity ^8.0.0;
```

5.1.2 Hardhat Framework

Hardhat is an open-source development environment for building smart contracts on the Ethereum blockchain. It offers a wide range of features that make the development process easier and more efficient, including built-in tasks for testing, debugging, and deploying contracts. It has built-in integration with popular testing frameworks like Mocha and Chai, allowing developers to easily write and run automated tests for their contracts. One of the key benefits of using Hardhat is its flexibility and extensibility. It can be easily customized and integrated with other tools and frameworks, making it a powerful tool for building complex decentralized applications. Additionally, Hardhat has a large and active community of developers, which means that there is plenty of support available if you run into any issues or have questions. Overall, using Hardhat can help streamline the development process and make it easier to build high-quality, secure smart contracts.

5.1.3 Goerli public test network

The Goerli Public Test Network is one of the Ethereum networks that developers can use to test their applications and smart contracts before deploying them to the main network. It is a proof-of-authority (PoA) test network, meaning that validators are not required to solve complex mathematical problems to create new blocks and validate transactions. Instead, a group of trusted validators is responsible for adding new blocks to the chain. Test networks are widely used by developers because they provide a stable and reliable environment for testing, without the risk of spending real Ether on the main network. Additionally, the network is supported by several Ethereum clients and has a block time of 15 seconds, making it a fast and efficient option for testing smart contracts and applications. Developers can require Goerli ETH by going to their faucet. The daily limit is 0.02 ETH/day (Solorio et al., 2019).

5.1.4 Ganache local network

When the daily amount of Goerli ETH received by the faucet was no longer sufficient, I turned to using Ganache, a local Ethereum client, for testing my smart contracts. Ganache enables users to create their own Ethereum blockchain on their computer with a user-friendly graphical interface that simplifies the interaction with the client. In addition, Ganache provides 10 default accounts, each with a balance of 100 ETH, which is more than enough for testing purposes. Another benefit is that the blockchain can be easily reset with a single click, making it ideal for testing and debugging. Overall, Ganache is a valuable tool for developers looking to test their smart contracts locally before deploying them to a public network.

5.1.5 Chai

When it comes to developing smart contracts, automation of testing is crucial, just like with any other sort of application development. Without programmatic testing, every time we make minor or major changes to the code, we would have to manually test every functionality of our system to ensure that everything works as expected and produces the correct results. With automated testing, we can test the system with a single command. If there are no errors, all tests run fine, and if there are any issues, the relevant tests will fail.

Chai is a popular assertion library that allows developers to write clear and concise tests for their smart contracts. With Chai, developers can easily assert the expected behavior of their contracts, which helps to ensure that they function as intended. Chai provides a simple and intuitive interface for writing tests and offers a variety of assertion styles, including BDD, TDD style, and even a more expressive should-style. By using Chai to test smart contracts, developers can catch issues and bugs early in the development process, saving time and effort in the long run. Additionally, Chai is compatible with a wide range of testing frameworks, making it easy to integrate into any development workflow (Solorio et al., 2019).

5.2 Standard Identification Contract

This smart contract serves as the backbone of the project. As previously mentioned, the implementation language will be Solidity. In order to make compilation, deployment, and testing easier, we will use the Hardhat framework, which we will install on our computer using the node package manager. Once installed, we can create a new project by running the command ”npx hardhat”. This will create a basic project structure that includes a sample contract, tests, and a configuration file. The project structure contains several folders, such as the contracts folder for Solidity contracts, the tests folder for writing tests using frameworks like Mocha and Chai, and the artifacts folder where the compiled versions of contracts are stored. The scripts folder contains JavaScript files for deploying and interacting with contracts. Lastly, the hardhat.config file is used to configure the Hardhat environment, such as network settings and other project-specific settings.

```
1    npm install -G hardhat
```

Listing 5.1: Install Hardhat framework

```
npx hardhat
```

Listing 5.2: Initialize Hardhat environment

5.2.1 SIC.sol

As this section is the essence of my project, I will present the implementation I have created in detail. Connecting a smart contract with a web application and creating a user interface using the React framework are not particularly innovative topics, so I will not go into detail about their implementation. However, a standard smart contract suitable for verifying student credentials on the Ethereum blockchain deserves several pages so that the reader can fully understand its operation.

Since our contract will have functions that can only be called by the university, who is the owner of the smart contract, we will inherit the Ownable smart contract offered by OpenZeppelin. OpenZeppelin Ownable contracts are a popular choice for implementing ownership functionality in smart contracts on the Ethereum blockchain. The Ownable contract provides a modifier that can be applied to specific contract functions to ensure that they can only be executed by the owner of the contract. This is often used to restrict access to certain functions or to allow for easy transfer of ownership. This contract has been audited and widely used in production, making it a reliable and secure choice for smart contract development. To use this contract we have to import it from OpenZeppelin.

```
// SPDX-License-Identifier: MIT

import "@openzeppelin/contracts/access/Ownable.sol";

pragma solidity ^0.8.0;

contract SIC is Ownable {
    ...
}
```

In addition to this condition, several other conditions must be met when calling certain functions. To avoid repetition in the code, Solidity provides the ability to declare function modifiers, which are typically placed at the beginning of the contract. Function modifiers are used to perform checks and validations before executing a function, such as checking if the caller is authorized to call the function or if certain conditions are met before executing the function. This helps to ensure that the function is executed correctly and securely. Function modifiers are declared using the modifier keyword, followed by the name of the modifier and its logic, and then they can be applied to any function that requires them (Chittoda, 2019). We will need three modifiers: one to check if the student not exists, another to check the

opposite of this, and lastly a modifier which checks if the specialization exists. It's important to mention that modifiers can receive parameters, in this cases the id of the entities.

```
modifier studentNotExists(string calldata _id, address _address) {
    require(studentByID[_id].walletAddress == address(0));
    require(bytes(studentIDByAddress[_address]).length == (0));
    _;
}

modifier studentExists(string calldata _id) {
    require(studentByID[_id].walletAddress != address(0));
    _;
}

modifier specializationExists(string calldata _specialization) {
    require(specializations[_specialization]);
    _;
}
```

Let's see what data structures we need. Like in almost every programming language, Solidity also allows declaring enumerations and structures. We will create an enumeration to determine the student status, which can take three values: finished, active, and suspended. In addition to this, we will need structures: one is representing the student's data, one for the study's data, and an input student structure for adding students.

```
enum StudentState {
    ACTIVE,
    SUSPENDED,
    FINISHED
}

struct Learning {
    string specialization;
    string faculty;
    uint256 started;
    StudentState state;
}

struct Student {
    string name;
    string ipfsUrl;
    address walletAddress;
    Learning[] learnings;
}

```

```
21 struct InputStudent {
22     string name;
23     string id;
24     string ipfsUrl;
25     address walletAddress;
26     string specialization;
27 }
```

In order to store list like data, we will need some kind of array, fortunately Solidity provides us mappings. Mappings are a way to associate a key with a value, similar to a dictionary or hash table in other programming languages. Mappings are a very useful data structure in smart contract development because they provide an efficient way to store and retrieve data. Mappings can be declared with various data types as both keys and values. For example, you can create a mapping with an address type as a key and a integer type as a value, which could be used to store balances of users. Additionally, mappings in Solidity are always initialized to their default value, which is usually zero or false, so there's no need to initialize them explicitly. However, it's important to note that mappings cannot be iterated over like arrays or structures, so you need to know the key in order to retrieve the value (Chittoda, 2019). The contract will also need two string variables to store the name of the University and the symbol.

```
1 mapping(string => Student) studentByID;
2 mapping(address => string) studentIDByAddress;
3
4 mapping(string => bool) specializations;
5 mapping(string => bool) faculties;
6 mapping(string => string) specializationToFaculty;
7
8 string _name;
9 string _symbol;
```

The following lines in the contract makes up the constructor. Solidity contracts have constructors, which are special functions that are called only once during the contract's creation. Constructors are used to initialize the contract's state variables, which are variables that store the contract's data. Without a constructor, a contract's state variables would be uninitialized. Constructors can also be used to perform any necessary setup logic, such as setting default values for variables or initializing other contracts that the current contract depends on. Additionally, constructors can take arguments, which allows for more flexibility in initializing the contract's state (Chittoda, 2019). In our case the constructor will initialize the name and symbol variables, which are representing this properties about the university respectively. Because this contract will be extended by the universities in their implementations, the constructor will be called inside the children's constructor.

```
constructor(string memory name_, string memory symbol_) {
    _name = name_;
    _symbol = symbol_;
}
```

Now that we have looked at everything we will need in this contract to store the information and initialize the contract with the help of the constructor, we can start defining the necessary functions to satisfy the expected functionality.

Solidity functions are a fundamental part of smart contracts, enabling the contract to interact with the outside world and execute its intended behavior. The order of keywords in the function header should be ”function”, followed by the name of the function, then the input parameters enclosed in parentheses. The next keyword is the ”public”, ”private”, ”internal”, or ”external” visibility modifier, which determines who can call the function. Public functions can be called by anyone, including other contracts and external accounts, while private functions can only be accessed within the contract itself. Internal functions can be called by the contract and its derived contracts, and external functions can only be accessed by external accounts and contracts. The ”view” or ”pure” keywords indicate that the function only reads the contract's state or doesn't read or write state, respectively. The ”payable” keyword allows the function to receive ether. The return type of the function is specified after this, or ”void” is used if the function does not return a value. The function body, enclosed in curly braces, contains the code to be executed when the function is called. Proper use of these keywords is essential to ensure the contract behaves as intended and follows best practices in terms of security and efficiency (Chittoda, 2019).

Our first function will be responsible for adding a new student to the blockchain. The argument of the function will be a variable of type Input student, which will carry the data of the new student. Between the variable type and name, an important keyword is included that specifies where the variable can be found. In this case, it is calldata, and I will explain its details soon, as it is important for optimization purposes.

```
function addStudent(InputStudent calldata _student) public onlyOwner
    studentNotExists(_student.id, _student.walletAddress)
    specializationExists(_student.specialization)
{
    studentByID[_student.id].name = _student.name;
    studentByID[_student.id].ipfsUrl = _student.ipfsUrl;
    studentByID[_student.id].walletAddress = _student.walletAddress;

    studentIDByAddress[_student.walletAddress] = _student.id;
    studenStartsSpecialization(_student.id, _student.specialization);
}
```

This function is public, so it can be called from anywhere, but only if the caller is the owner of the contract, the student with the given ID does not exist yet, and the specialization given in the student data exists. We can apply these criteria by adding modifiers to the function header.

In Solidity, there are three locations where function arguments can be stored: memory, storage, and calldata. Calldata is a special data location that is used for function arguments that are passed when a contract is called externally. The calldata keyword is used to specify that the function argument should be read from calldata. The reason for using calldata instead of memory or storage is that calldata is much cheaper in terms of gas cost. When a function is called externally, the arguments are passed in calldata and are not stored on the blockchain. This means that using calldata can save a significant amount of gas cost compared to using memory or storage for function arguments. However, it's important to note that calldata is read-only, which means that the function cannot modify its arguments (Chittoda, 2019).

Whenever a student starts a new specialization, there must be a way to add this to the blockchain. For this purpose I defined the "studentStartsSpecialization" function. This function pushes the data of the new specialization to the student's learning's array. This function can be called under the same conditions as the one adding the students, so the same modifiers are added to the header.

```
function studenStartsSpecialization(
    string calldata _id,
    string calldata _specialization
)
    public
    onlyOwner
    studentExists(_id)
    specializationExists(_specialization)
{
    studentByID[_id].learnings.push(
        Learning({
            specialization: _specialization,
            faculty: specializationToFaculty[_specialization],
            started: block.timestamp,
            state: StudentState.ACTIVE
        })
    );
}
```

Every student in a specialization can have three types of status. When they start their training, their status becomes active, but if they want or are forced to suspend their studies for any reason, it is important that this is visible on the blockchain as well. The "studentSuspended" function serves this purpose. Similarly, we need

two other state-changing functions, one for reactivating the student's status after the suspension period has expired and another for setting the finished status when the studies are completed. Since a student can participate multiple trainings, we need a helper function that returns the index in the student's learnings array where the specialization is available based on the specialization identifier. These three functions differ only minimally, so I will only present the last one in more detail.

```
function getLearningIndex(
    string memory _id,
    string memory _specialization
) internal view returns (uint32) {
    uint32 index = 0;
    for (index; index < studentByID[_id].learnings.length; index++) {
        if (keccak256(abi.encodePacked(studentByID[_id].learnings[index].
            specialization)) == keccak256(abi.encodePacked(_specialization
            ))) {
             return index;
        }
    }
    revert("This student does not started the given specialization!");
}

function studentFinished(
    string calldata _id,
    string calldata _specialization
)
    external
    onlyOwner
    studentExists(_id)
    specializationExists(_specialization)
{
    uint32 index = getLearningIndex(_id, _specialization);
    require(
        studentByID[_id].learnings[index].state == StudentState.ACTIVE,
        "Only active students state can be changed to finished"
    );
    studentByID[_id].learnings[index].state = StudentState.FINISHED;
}
```

There are only two pieces of data left regarding the student that we might want to modify. These are the student's Ethereum address and the IPFS URL for their image. Just like the previous methods, only the contract owner can call these methods. The addresses of the students should be changeable, as it can happen that someone loses access to their wallet and doesn't have the proper keys to regain access. It is also advisable to periodically change the IPFS-stored image of the student, as students appearance can change significantly over a few years, and it's important for third

parties to be able to verify the identity of the rightful owner of the certificate based on the image.

```
function studentChangeAddress(string calldata _id, address
    _walletAddress)
    external
    onlyOwner
    studentExists(_id)
{
    delete studentIDByAddress[studentByID[_id].walletAddress];
    studentByID[_id].walletAddress = _walletAddress;
    studentIDByAddress[_walletAddress] = _id;
}

function studentChangeImageIpfsUrl(
    string calldata _id,
    string calldata _ipfsUrl
) external onlyOwner studentExists(_id) {
    studentByID[_id].ipfsUrl = _ipfsUrl;
}
```

As the number of students continuously grows, the university can also launch new courses. In order for students to enroll in these new specializations, they must appear on the blockchain. However, due to low interest, the university may decide not to launch unpopular courses in a given year. For this functionality, we will need three functions: one that adds a new specialization to the array, another that sets the availability status to false, and the last one that makes it available again.

```
function addSpecialization(string calldata _specialization,
    string calldata _faculty)
    external
    onlyOwner
{
    require(
        specializations[_specialization] == false,
        "Already added specialization"
    );
    require(
        faculties[_faculty] == true,
        "Invalid faculty"
    );
    specializations[_specialization] = true;
    specializationToFaculty[_specialization] = _faculty;
}
```

Since the ”enableSpecialization” function only differs in the value from the ”disableSpecialization” function, I will not include the code explicitly.

```
function disableSpecialization(string calldata _specialization)
    external
    onlyOwner
{
    require(
        specializations[_specialization] == true,
        "Specialization is not available"
    );
    specializations[_specialization] = false;
}
```

With the following method the owner of the contract can add a new faculty to the blockchain.

```
function addFaculty(string calldata _faculty)
    external
    onlyOwner
{
    require(
        faculties[_faculty] == false,
        "Already added faculty"
    );
    faculties[_faculty] = true;
}
```

The basic functionalities needed for the system are almost complete. Now the contract owner can add new faculties, specializations, and assign students to them. Also, they can change their availability and some other data. However, there is still something missing. Currently, third parties do not benefit from this system at all. What happens if a student wants to buy a train ticket at a discounted rate, but it is only available to students? They could enter their Ethereum wallet on their phone and show their student ID through the university's web3 application. Yes, but this carries the risk of fraud, which third parties can only avoid in a cumbersome way. For usability, we want third parties to be able to verify a student's university status on their own devices using the student's identifiers. What are these identifiers? One of the most obvious is the student's personal identification number, which can be shown using their identity card and does not change over time. Another option is to identify the student based on their wallet address, which can also be shown as a QR code using a wallet. The third party enters one of these pieces of information into the search bar on the university's web3 application and immediately receives the necessary information. The chance of fraud is minimal. Finally, I decided to offer both options to users, as this way I can cover those cases when someone doesn't have their Ethereum wallet with them, only their ID card, or vice versa. This pair

of functions is public and can be called by anyone, but it can only verify someone's university status if they have one of the two identifiers.

```
function verifyByID(string calldata _id)
    public
    view
    studentExists(_id)
    returns (Student memory, string memory id)
{
    return (studentByID[_id], _id);
}

function verifyByAddress(address _walletAddress)
    public
    view
    returns (Student memory, string memory id)
{
    require(bytes(studentIDByAddress[_walletAddress]).length!=0, "This
        address is not registered as a students address");
    return (studentByID[studentIDByAddress[_walletAddress]],
        studentIDByAddress[_walletAddress]);
}
```

5.3 UBB contract

So we have finished the Student Identification Contract which, being a standard contract, cannot be used on its own. In order for a university to start using it, it needs to be extended through inheritance. The minimal extension required for it to be usable is calling the contract constructor with the name and symbol of the university. Therefore, the implementer does not need to add new functions, but they have the option to introduce custom functions and variables in the children contract according to their needs.

Great! Let's implement the system for our own university. We will create the UBB.sol contract, which inherits the properties of the SIC.sol contract. At the same time, we want to extend the basic contract functionality with a function that allows adding multiple students to the blockchain with a single function call.

```

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

import "./SIC.sol";

```

```
contract UBB is SIC {
    constructor() SIC("Universitatea Babes Bolyai", "UBB") {}
    function addMultipleStudents(InputStudent[] calldata _students)
        external
        onlyOwner
    {
        uint256 studentArrayLength = _students.length;
        for (uint256 i = 0; i < studentArrayLength; i++) {
            addStudent(_students[i]);
        }
    }
}
```

Testing

As I mentioned, Hardhat offers us the opportunity to automate testing with the Mocha and Chai libraries. I have decided to use Chai for now. The tests folder can be found within the folder structure. The tests should have a JavaScript extension. We can create as many files as we want, and we can run them all at once or one by one. The tests are structured using the describe and it keywords. The describe function is used to group the tests, while it represents an individual test.

```
const faculty = "Facultatea de Stiinte Economice si Gestiunea Afacerilor"
describe("Student tests:", function () {
  async function deployContract() {
    const UBB = await ethers.getContractFactory("UBB");
    const ubb = await UBB.deploy();
    await ubb.addFaculty(faculty);
    await ubb.addSpecialization("Informatica Economica", faculty);
    await ubb.addSpecialization("Management", faculty);
    await ubb.addSpecialization("Marketing", faculty);
    await ubb.addSpecialization("Finante si banci", faculty);
    return ubb;
    }
  it("Should be possible to add a student", async function () {
    const ubb = await loadFixture(deployContract);
    await ubb.addStudent({
      name: name,
      id: id,
      ipfsUrl: ipfsUrl,
      walletAddress: address,
      specialization: specialization,
    });
    const res = await ubb.verifyByID(id);
  });
});
```

The "deployContract" function, as its name suggests, deploys our code and also inputs some initial data for testing purposes. When testing smart contracts in Hardhat with Chai, the contract is deployed to a local Hardhat network. Hardhat provides a built-in development network that runs on your local machine, allowing to test and debug smart contracts before deploying them to a public blockchain network. This local network is created automatically when you run npx hardhat node or npx hardhat test command, and it simulates the behavior of a real blockchain network, including contract deployment and execution. Developers can interact with the deployed contracts on this local network using the contract instance returned by the deployment function in your test file.

Deploying

To specify whether the code should be deployed to the Ethereum Main Network, test networks, or a Local Ethereum Blockchain, we can use a command invoked in the CLI. First, we need to define the different networks in the hardhat configuration file. When deploying, we can then specify which network to choose from in the command.

```
module.exports = {
  solidity: "0.8.9",
  networks: {
    mainnet: {
      url: `https://mainnet.infura.io/v3/${PROJECT_ID}`,
      accounts: [MAINNET_PRIVATE_KEY]
    },
    goerli: {
      url: `https://eth-goerli.alchemyapi.io/v2/${ALCHEMY_API_KEY}`,
      accounts: [GOERLI_PRIVATE_KEY]
    },
    ganache: {
      url: 'http://127.0.0.1:8545',
      accounts: ["5485
          aaccb72957e8682e1131f1c1cfdc369099edbba15582df585baca99788ff"],
    }
  }
};
```

5.4 Node.js environment for Web3 Development

Node.js is a popular open-source JavaScript runtime environment that allows developers to build scalable and high-performance applications. When it comes to Web3 development, Node.js is an essential tool for interacting with the Ethereum blockchain. It also provides a range of tools and libraries that can be used to build decentralized applications on top of Web3, such as IPFS for decentralized storage and communication.

5.4.1 Wagmi

WAGMI is a JavaScript library for web3 development that aims to simplify the interaction between web applications and the Ethereum blockchain. It provides an easy-to-use interface for interacting with Ethereum smart contracts, including sending transactions, reading data from the blockchain, and listening for events. WAGMI abstracts away some of the complexities of interacting with the Ethereum blockchain, allowing developers to focus on building the functionality of their applications by having a collection of React Hooks containing everything you need to start working with Etherium. The library is open source and actively maintained, with a growing community of contributors and users.

5.4.2 Ethers

Ethers is a popular, powerful, flexible JavaScript library used for building Ethereum-based decentralized applications. It provides a simple and comprehensive API for interacting with the Ethereum blockchain and supports various features, such as sending and receiving transactions, signing messages, and deploying smart contracts. Ethers also supports multiple networks, including Ethereum Main Network and various test networks, making it easy for developers to test their decentralized applications in different environments. Additionally, the library provides several useful utilities, such as the ability to parse Ethereum addresses and convert between different units of ether.

5.4.3 Rainbow-kit

Rainbow is a JavaScript library designed for web3 development, specifically for Ethereum-based applications. It provides a set of tools and utilities that simplify working with Ethereum, such as handling account management, transaction signing, and contract interactions. Rainbow supports a wide range of networks, including the Ethereum Main Network and various test networks. It also provides built-in

support for popular wallets like MetaMask and WalletConnect. Rainbow is a relatively new library, but it has gained popularity in the web3 community due to its ease of use and comprehensive documentation.

5.4.4 Moralis SDK

Moralis SDK is a back-end infrastructure provider that simplifies the process of developing Web3 applications. It offers a range of features that include serverless functions, a fully-managed database, user authentication, and more. Moralis provides a simple and straightforward way for developers to integrate with the Ethereum blockchain and other decentralized technologies without the need to worry about complex infrastructure and security. With Moralis, developers can quickly build and deploy decentralized applications that are scalable, secure, and highly available. The SDK is designed to work seamlessly with popular Web3 development libraries such as Web3.js and Ethers.js, making it an excellent choice for developers looking to streamline their development process.

5.4.5 React Front-End

React is a widely popular JavaScript library for building user interfaces, and it is a fantastic choice for creating web applications, especially for web3 development. React's core philosophy revolves around the concept of reusable and modular components, which greatly simplifies the process of developing complex web applications.

One of the key advantages of React is its component-based architecture. With React, you can break down your user interface into smaller, self-contained components that can be reused across your application. This promotes code reusability, maintainability, and makes it easier to manage and update different parts of your application.

React's virtual DOM (Document Object Model) is another powerful feature that contributes to its efficiency and performance. Instead of directly manipulating the actual DOM, React uses a virtual representation of it. This allows React to efficiently update only the necessary components when there are changes in the application's state, resulting in faster rendering and improved user experience.

When it comes to web3 development, React's component-based approach aligns well with the decentralized nature of blockchain applications. React's flexibility and modularity make it easier to integrate web3 libraries, such as Web3.js or Ethers.js, for interacting with blockchain networks and smart contracts. Additionally, React's ecosystem offers numerous third-party libraries and tools specifically designed for web3 development, making it even more convenient to build decentralized applications.

React's large and active community is another advantage. It means there are abundant resources, tutorials, and community support available, which can help developers learn and solve problems more efficiently. React also has excellent developer tools, such as React DevTools, that aid in debugging, inspecting component hierarchies, and optimizing performance.

5.5 Connecting to the Ethereum Blockchain

Now that we have finished developing the smart contracts, tested them, and deployed them to the test network, it's time to move on. As I mentioned earlier, a UI is not necessary to interact with smart contracts deployed on Ethereum. However, since this system will have many users who are not experts in this field, we will need a UI that allows them to communicate with the blockchain without code. In this documentation, I will not discuss the implementation of the React front-end, because it is not the purpose of this thesis. Instead, I will show you how the connection between the blockchain and the user interface is established using popular JavaScript libraries. We will look at the implementation of some important functions to better understand how this connection works. I would also like to showcase how we can store images in a decentralized manner using IPFS (InterPlanetary File System).

5.5.1 Log-In to Ethereum Wallet

In order to use a decentralized application running on the Ethereum blockchain, we need a wallet. Any type of Ethereum wallet will be suitable for our goal, whether it is browser-based or a mobile application. To communicate with the blockchain, we need to sign in to our wallet so that the application can identify us based on our public key. The implementation of this sign-in process will be the first thing I would like to introduce. To achieve this, we will need a component that allows the user to choose from different wallets upon clicking. The "ConnectButton" component from the "Rainbowkit" package is a perfect choice for this. First, we import the necessary dependencies and hooks from various libraries such as "next-auth/react", "@moralisweb3/next", and "wagmi". These dependencies provide the functionalities required for authentication, web3 interaction, and network handling. Inside the Connect component, we utilize several hooks to access and manage various states and actions. The "useAccount" hook from the "wagmi" library provides access to the connected account's information, including whether it is connected and the account address. The "useNetwork" hook also from "wagmi" provides access to the current network information. The "useSession" hook from "next-auth/react" retrieves the authentication session status. The "useSignMessage" hook from "wagmi" allows

signing of messages. The ”useAuthRequestChallengeEvm” hook from ”@moralisweb3/next” is used for requesting an authentication challenge from the Ethereum Virtual Machine.

```
const Connect = () => {
  const { isConnected, address } = useAccount();
  const { chain } = useNetwork();
  const { status } = useSession();
  const { signMessageAsync } = useSignMessage();
  const { requestChallengeAsync } = useAuthRequestChallengeEvm();

  useEffect(() => {
    const handleAuth = async () => {
      const { message } = await requestChallengeAsync({
        address: address,
        chainId: chain.id,
      });

      const signature = await signMessageAsync({ message });

      await signIn("moralis-auth", {
        message,
        signature,
        redirect: false,
        callbackUrl: "/",
      });

      console.log("Authentication finished.");
    };

    if (status === "unauthenticated" && isConnected) {
      console.log(
        "Authentication called because: status=" +
          status +
          ", isConnected=" +
          isConnected
      );
      handleAuth();
    }
  }, [status, isConnected]);

  return (
    <div className={styles.connect__container}>
      <ConnectButton />
    </div>
  );
};
```

The component utilizes the ”useEffect” hook to handle the authentication process. It listens for changes in the ”status” (authentication status) and ”isConnected” (account connection status) variables. When the status is ”unauthenticated” and ”isConnected” is true, it triggers the authentication process.

The authentication process involves requesting an authentication challenge from the EVM using the ”requestChallengeAsync” function. It then signs the challenge message using the ”signMessageAsync” function. Finally, it calls the ”signIn” function from ”next-auth/react” to authenticate the user with the signed message. The ”signIn” function receives parameters such as the authentication provider (”moralis-auth” in this case), the challenge message, the signature, and other options.

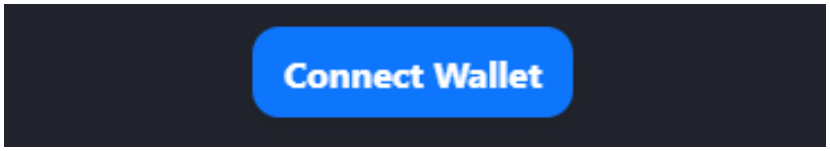


Figure 5.1: Connect to wallet button

When we click on the Connect wallet, a modal pops out where users can choose between wallet applications, or also can read about what is actually an Ethereum wallet, and create their first wallet if needed.

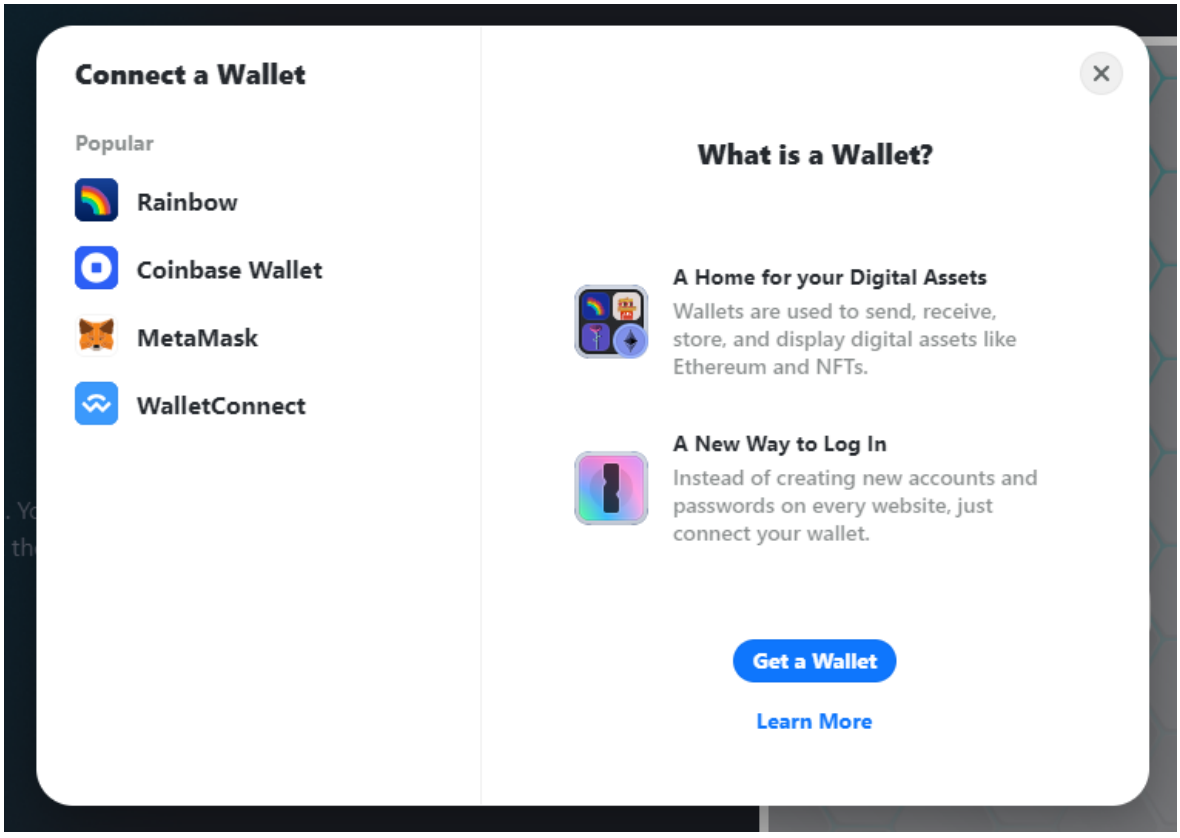


Figure 5.2: Connect to wallet button

As I mentioned, for ease of testing, I'm using the MetaMask browser-based wallet. This wallet can be installed as an extension in our browser. After clicking on the MetaMask button in the modal, the extension pops up in a window, and we need to confirm the login. Since I am already logged into the extension, there is no need to enter a password in this case.

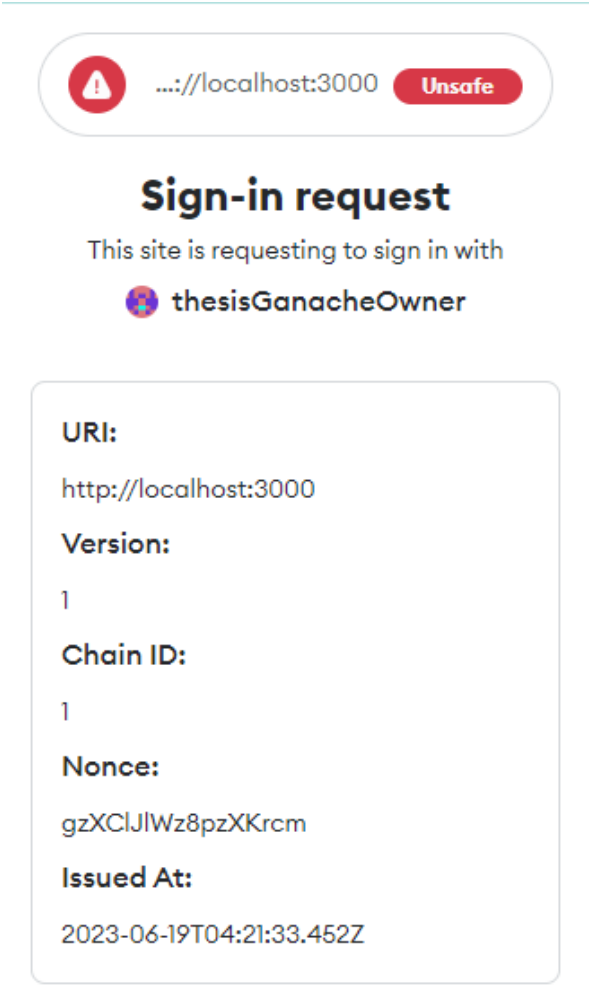


Figure 5.3: Confirm sign-in request in MetaMask

After clicking Sign-In button, the pop-out disappears, and we are successfully signed-in to our wallet. This means that the application is connected to the blockchain, we can perform actions, and interact with the smart contract. The Connect buttom from the navigation bar is replaced with two buttons, one for changing network, and the other one indicating the user's balance ad public address.

Figure 5.4: On successful connection

5.5.2 IPFS Upload

Inter Planetary File System, is a distributed and decentralized protocol designed for storing and sharing files on the internet. By adopting content-based addressing rather than the more conventional location-based addressing, it provides a alternative, more secure method of file storing. Each file in IPFS is given a special identification number called a content identifier (CID), which is derived from the file's actual content. The file's integrity is retrieved and verified using this CID, making it resistant to manipulation and guaranteeing data consistency. By effectively caching and distributing data, content-based addressing lowers duplication and boosts network speed.

One of the key advantages of IPFS is its decentralized nature. Instead of relying on a central server or a single point of failure, IPFS utilizes a peer-to-peer network of interconnected nodes. Each node in the network stores and serves pieces of the files, creating redundancy and ensuring high availability. This decentralized architecture not only enhances data resilience but also enables censorship resistance and promotes a more open and democratic internet. Additionally, IPFS supports offline and intermittent connectivity. Files can be accessed and shared even in disconnected or low-bandwidth environments. When two nodes come into proximity, they can exchange files directly without relying on a centralized server.

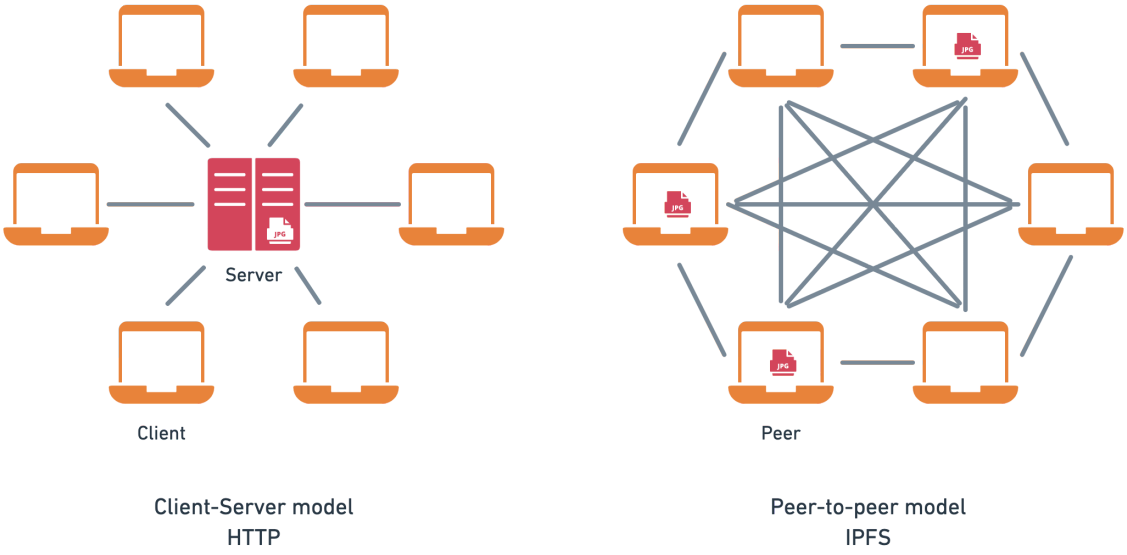


Figure 5.5: IPFS: Peer-to-peer model

In order to upload a file we can use the API provided by Moralis specially for interacting with IPFS. To implement a function with this functionality, first we have to import the Moralis library, which is a development platform that simplifies the process of building decentralized applications.

The "uploadToIpfs" function takes a single parameter, file, which represents the file that the user wants to upload. This function is designed to be used asynchronously, so it returns a Promise that will either resolve with the uploaded file's information or reject with an error.

```
export default async function uploadToIpfs(file) {
  if (!Moralis.Core.isStarted) {
    await Moralis.start({
      apiKey: process.env.moralisApiKey
    });
  }

  return new Promise((resolve, reject) => {
    const reader = new FileReader();
    reader.readAsDataURL(file);
    reader.onload = async () => {
      const fileUploads = [
        {
          path: file.name,
          content: reader.result,
        },
      ];
      try {
        const res = await Moralis.EvmApi.ipfs.uploadFolder({
          abi: fileUploads,
        });
        resolve(res.toJSON());
      } catch (error){
        reject(error)
      }
    };
    reader.onerror = (error) => {
      reject(error)
    };
  })
}
```

Before uploading the file, the function checks if Moralis Core has been started. If it hasn't been started, it means the Moralis SDK has not been initialized. In that case, the function initializes Moralis by calling the "Moralis.start" method and providing an API key for authentication. This step ensures that the necessary resources are set up before proceeding with the file upload.

To read the content of the file, the function creates a "FileReader" object. It reads the contents of the file passed as an argument. The "onload" event handler is defined to handle the successful file read. Within this event handler, the file data is retrieved using reader.result.

Next, the function prepares the file upload data. It creates an array that contains an object representing the file being uploaded. The object has two properties: path represents the name of the file, and content contains the file data obtained. The file upload is initiated using the ”Moralis.EvmApi.ipfs.uploadFolder” method. This method uploads the specified files to IPFS. The ”fileUploads” array is passed as the ABI parameter to provide the necessary information for the upload. If the file upload is successful, the Promise is resolved with the JSON representation of the response. On the other hand, if an error occurs during the upload process, the Promise is rejected with the error object.

Chapter 6

Conclusions

The project I envisioned and implemented fully leverages the proof-of-identity property of blockchains, providing a reliable source for both external and internal parties regarding the university status of students. In the future, I consider the complete digitalization of various identity documents inevitable, along with a revolutionary transformation of data storage procedures for the sake of security. Blockchain technology proves to be a perfect alternative to traditional data storage processes due to its high security, transparency, immutability, and decentralization. However, the technology is still very new, rapidly evolving, and in a few years, it has the potential to completely replace traditional software architectures, becoming the successor to the centralized digital world in a decentralized manner.

Blockchain technology is already being utilized in various industries. As mentioned in the thesis, the issuance of different diplomas is also being carried out on the Ethereum blockchain. However, this is not yet easily accessible to the general public. There is a need for various tools to simplify such processes, which is why I created the standard smart contract for student certification. With the emergence of the ERC-20 Non-Fungible token standard, the number of tokens on the Ethereum blockchain has experienced exponential growth, enabling individuals with limited technical knowledge to create their own tokens for their specific purposes. The ease of token creation is just one aspect. Since these tokens adhere to the same ERC-20 token standard implementation, it has become possible to establish decentralized exchanges. By using the same functions, we can interact with all ERC-20 tokens, making any application built on these tokens compatible with newly emerging tokens without requiring modifications.

A similar approach should be taken for a standard smart contract designed for student certification. The base contract includes all essential methods to fulfill its functionality while also allowing for customization based on individual needs. By leveraging inheritance, various individual implementations can be created, each containing the basic methods, enabling the development of aggregated platforms.

Let's imagine that in this case, with my implementation tailored to my university, the UBB contract, we can only verify the students of that specific university. However, in the future, aggregated platforms could be established where universities can apply, allowing students to be verified even if they are studying at different institutions. For example, it is conceivable to create a nationwide platform that would serve as a reliable source across the entire country.

Although the current version that has been developed is functional, there is still a lot of work to be done before it becomes fully usable in terms of security and gas optimization. If it were to be deployed for actual usage, it should not be deployed on the test network but rather on the official Ethereum Main Network. This would incur real costs, but it would provide a completely reliable network for operation.

Bibliography

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

Bambara, J. J., & Allen, P. R. (2018). Blockchain. *A practical guide to developing business, law and technology solutions. New York City: McGraw-Hill Professional*.

Chittoda, J. (2019). *Mastering blockchain programming with solidity: Write production-ready smart contracts for ethereum blockchain with solidity*. Packt Publishing Ltd.

The global blockchain market is projected to grow from $7.18 billion in 2022 to $163.83 billion by 2029, at a cagr of 56.3% in forecast period, 2022-2029... (2022). `https://www.fortunebusinessinsights.com/industry-reports/blockchain-market-100072`.

Gravity eliminates health safety certificate fraud with dock. (n.d.). `https://www.dock.io/gravity-case-study`. (Accessed on April 30, 2023)

Kube, N. (2018). *Daniel drescher: Blockchain basics: a non-technical introduction in 25 steps: Apress, 2017, 255 pp, isbn: 978-1-4842-2603-2.* Springer.

Lashkari, B., & Musilek, P. (2021). A comprehensive review of blockchain consensus mechanisms. *IEEE Access*, *9*, 43620–43652.

Maryville university: Digital diploma. (n.d.). `https://www.maryville.edu/digitaldiploma`. (Accessed on May 1, 2023)

Solorio, K., Kanna, R., & Hoover, D. H. (2019). *Hands-on smart contract development with solidity and ethereum: From fundamentals to deployment*. O'Reilly Media.

Tapscott, D., & Tapscott, A. (2016). *Blockchain revolution: how the technology behind bitcoin is changing money, business, and the world*. Penguin.

Annexes

User interface

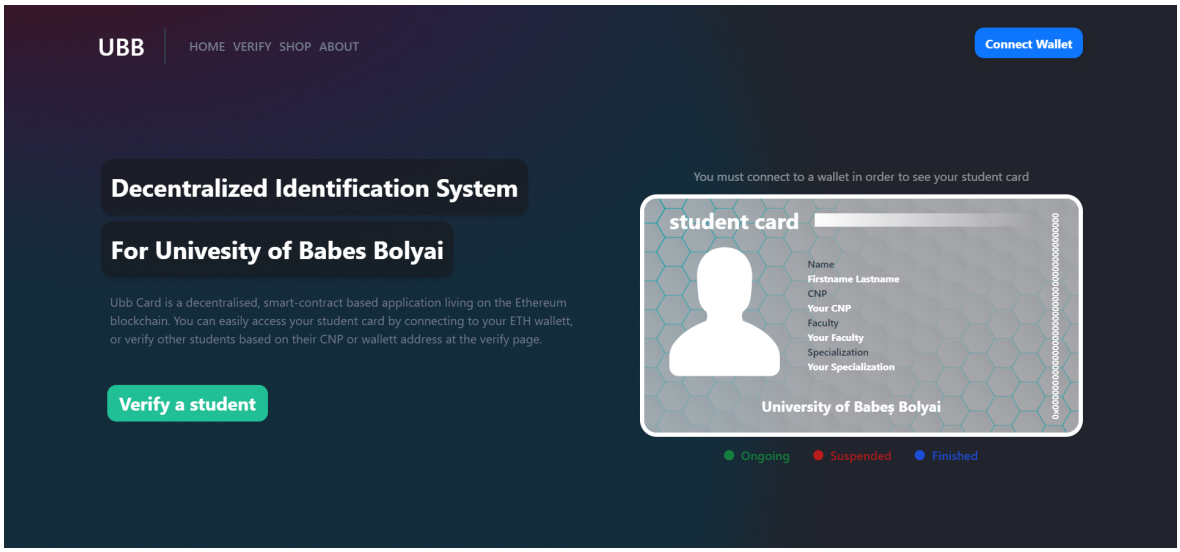


Figure 1: Home Page when no user is connected

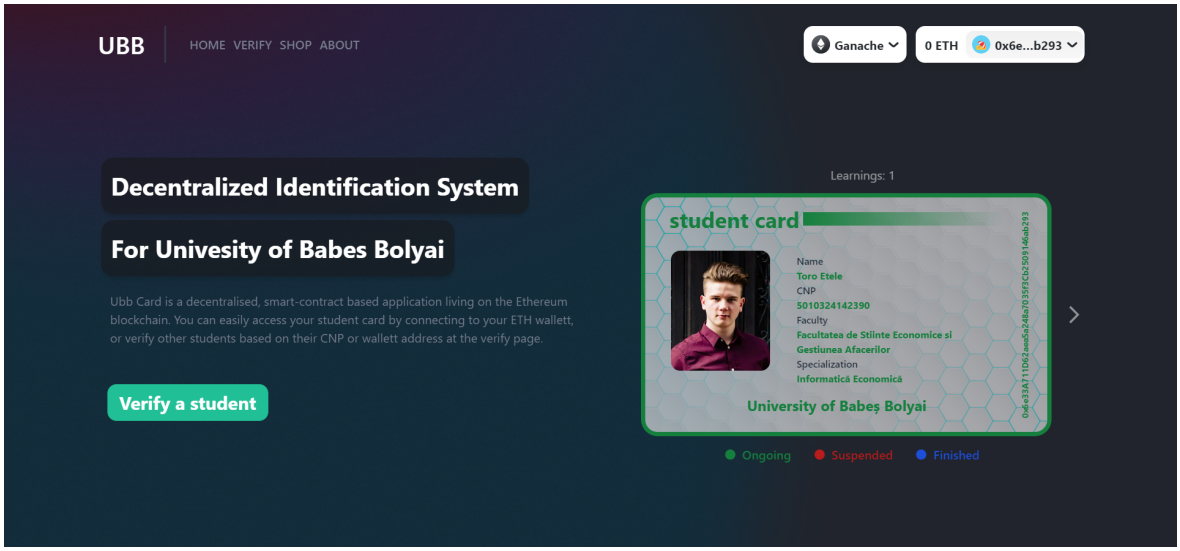


Figure 2: Home Page when user is connected

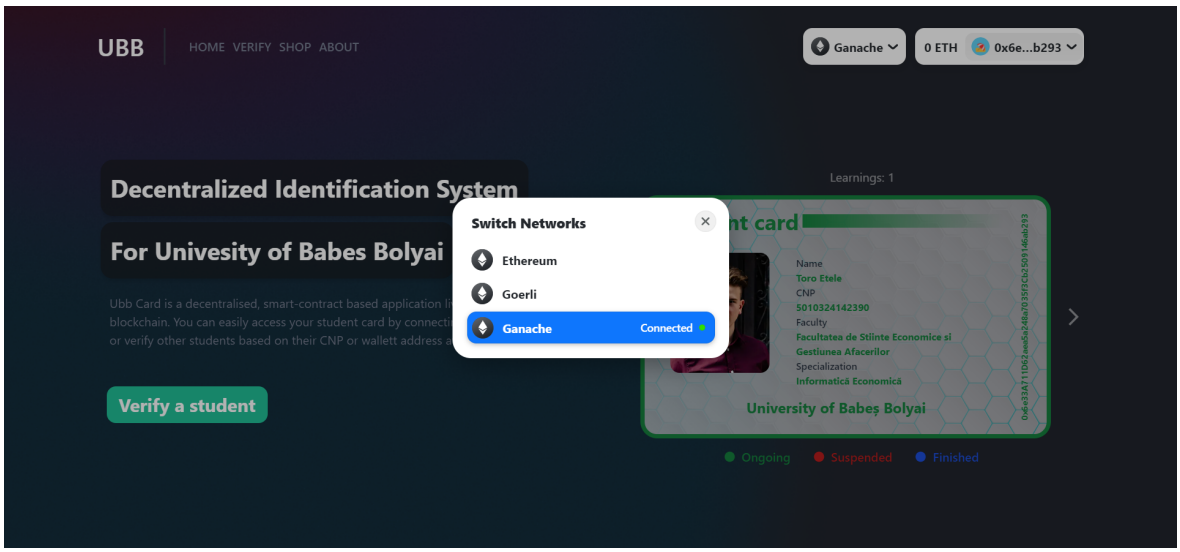
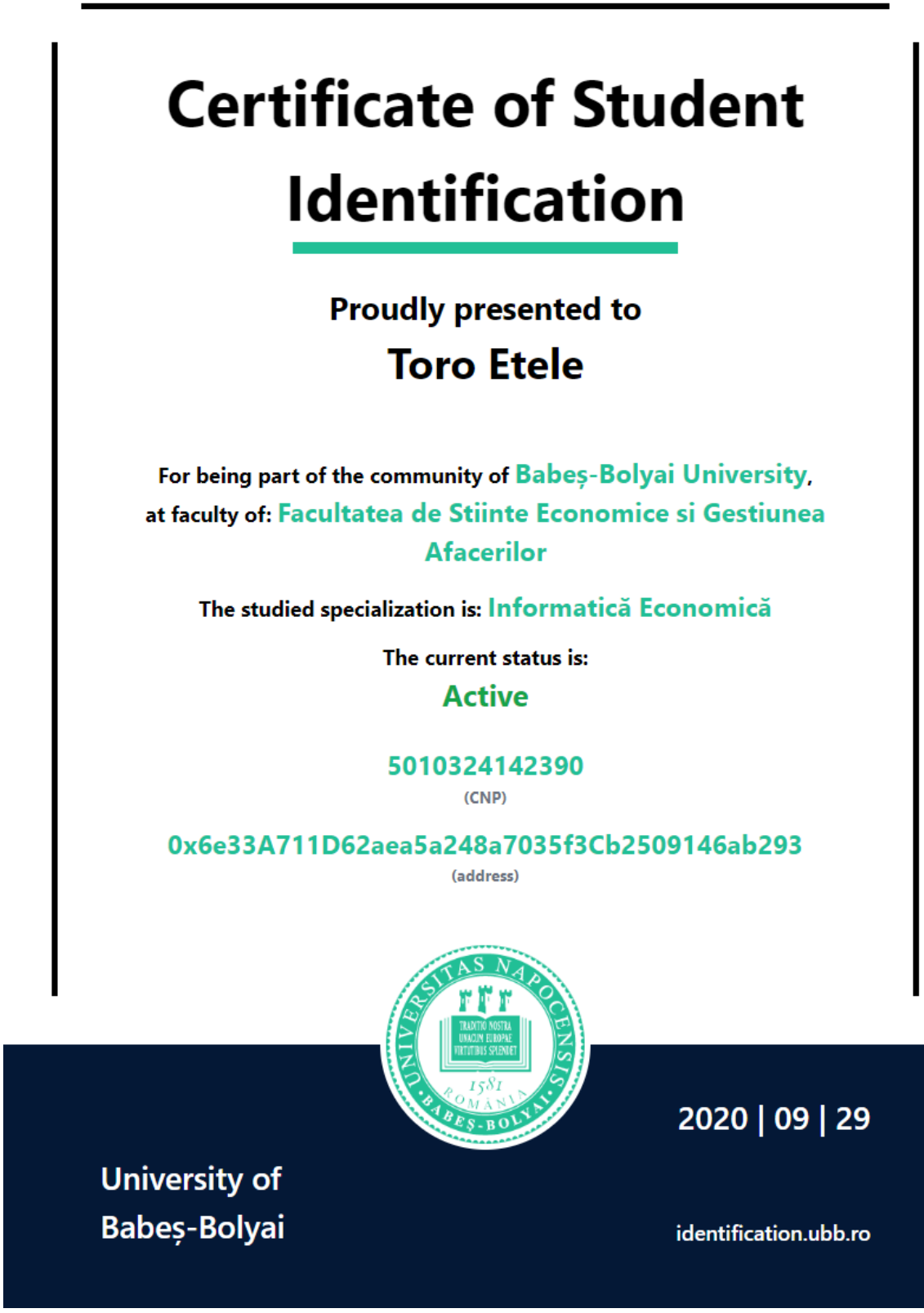


Figure 3: Switch network modal

Figure 4: Search bar on the verify page toggled

Figure 5: Search bar on the verify page address mode



Certificate of Student Identification

Proudly presented to

Toro Etele

For being part of the community of Babeș-Bolyai University, at faculty of: Facultatea de Stiinte Economice si Gestiunea Afacerilor

The studied specialization is: Informatică Economică

The current status is:

Active

5010324142390

(CNP)

0x6e33A711D62aea5a248a7035f3Cb2509146ab293

(address)

UNIVERSITAS NAPOCENSIS
TRADITIO NOSTRA UNACUM EUROPAE VIRTUTIBUS SPLENDET
1581
ROMANIA
BABEȘ-BOLYAI

2020 | 09 | 29

University of Babeș-Bolyai

identification.ubb.ro

Figure 6: Result of the student verification

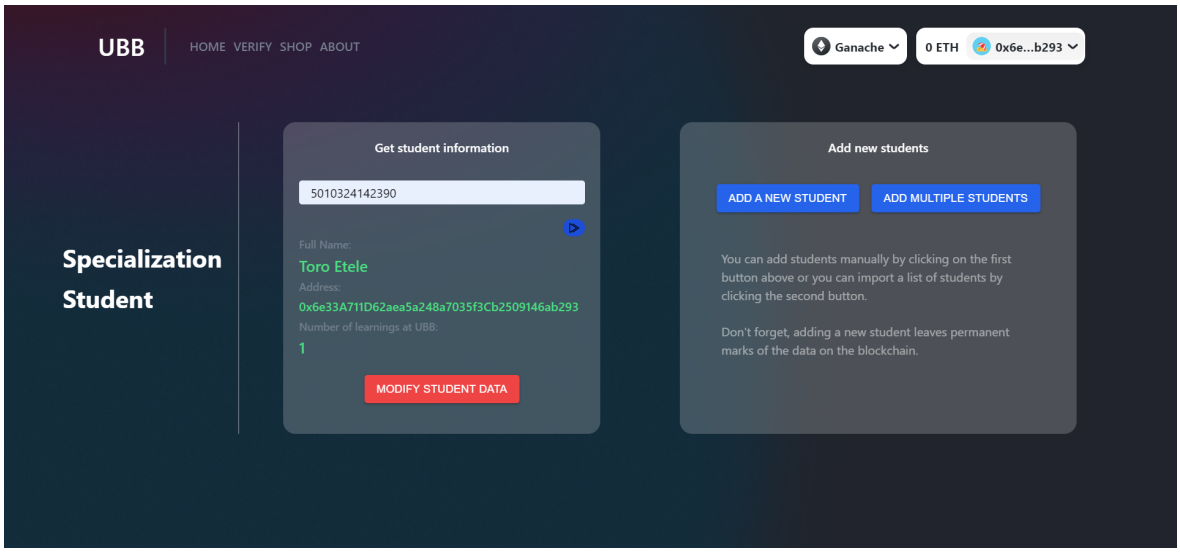


Figure 7: Admin page student section

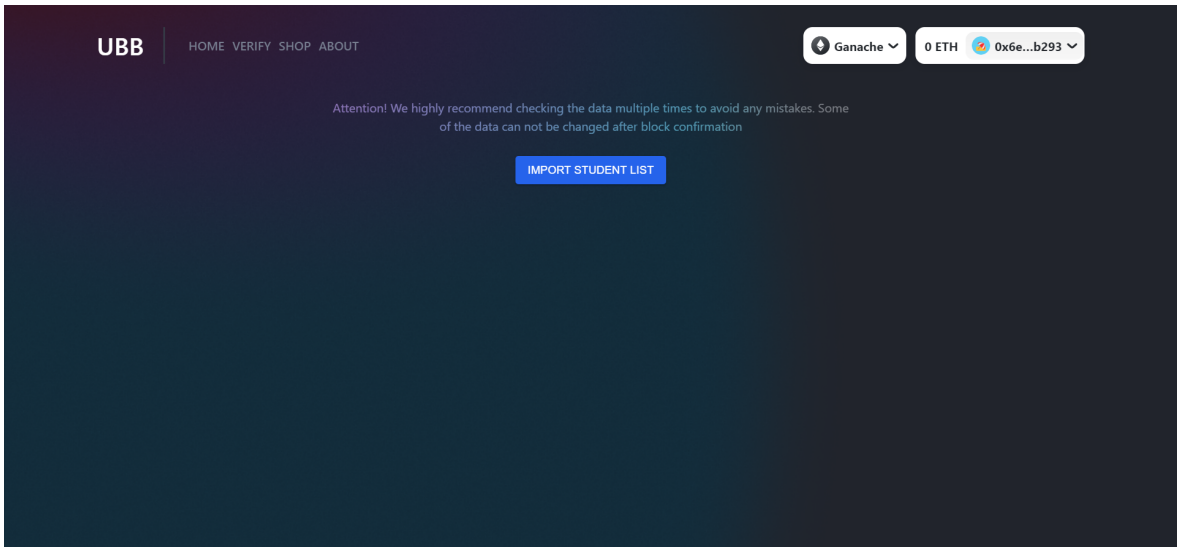


Figure 8: Import multiple students from file

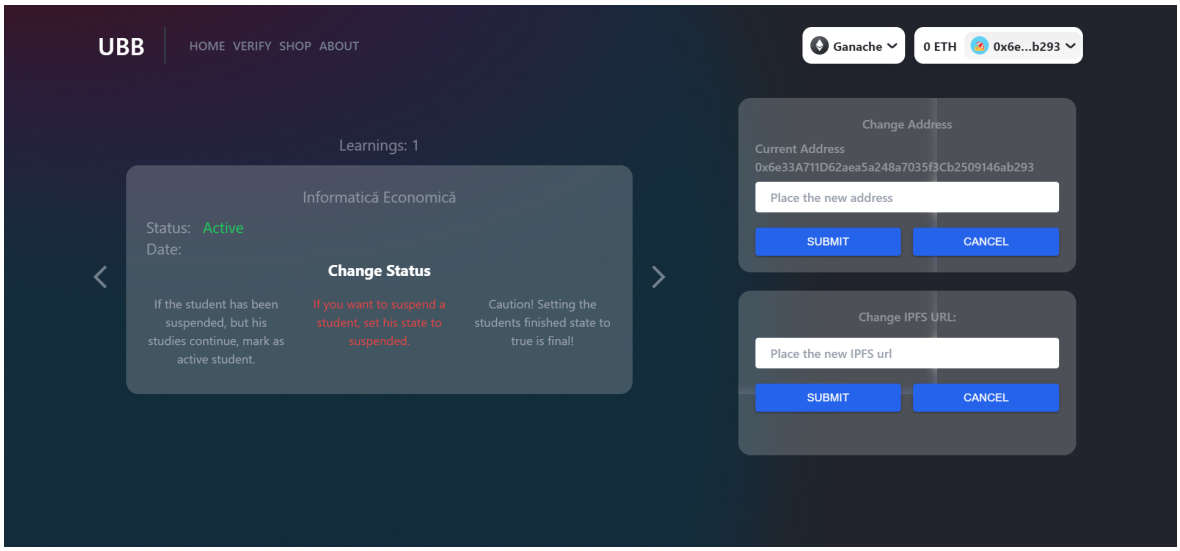


Figure 9: Edit the data of a student

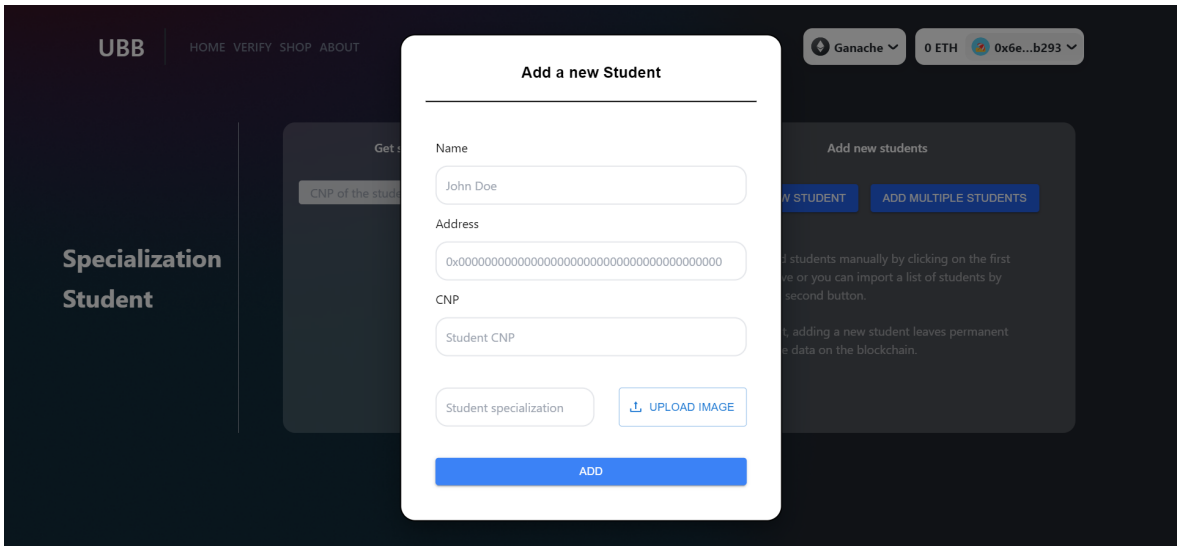


Figure 10: Add a new student