University of Sheffield International Faculty,
CITY College
Department of Computer Science

FINAL YEAR PROJECT

Secure Decentralized Marketplace for Microstocks

This report is submitted in partial fulfillment of the requirement for the degree of Bachelor of Science with Honours in Computer Science by

Vijon Baraku

June, 2022

Approved

Dr. Simeon Veloudis

Secure Decentralized Marketplace for Microstocks

by
Vijon Baraku

Dr. Simeon Veloudis

Abstract

This dissertation aims to present and implement a web-based, secure, decentralized marketplace for microstocks. The rationale behind developing and implementing a decentralized marketplace where intangible 'creations of mind' are traded is that through blockchain technology the need for intermediaries is eliminated, hence resulting in a transparent, cheaper, and unbiased market, albeit a secure one that guarantees integrity of transactions and confidentiality of traded artefacts. The report will present an overview of blockchain technology fundamentals, demonstrate smart contract implementations, and employ multiple encryption schemes to provide security.

DECLARATION

All sentences or passages quoted in this thesis from other people's work have been specifically acknowledged by clear cross referencing to author, work and page(s). I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this thesis and the degree examination as a whole.

Name : **Vijon Baraku**

Singed: ________________ Date: ________________

Acknowledgements

First and foremost I am extremely grateful to my supervisor Dr. Simeon Veloudis for his invaluable advice, continuous support, and patience throughout the project. I'm proud of, and grateful for, my time working with Dr. Veloudis, and I hope that I get the opportunity to work with you again in the future.

I am also grateful to my classmates, friends, and loved ones for their late-night feedback sessions, as well as much-needed distraction, entertainment, and moral support.

Most importantly, I am grateful for my parents, Agon and Trandelina, and my brothers, Tron and Rezon. Their belief in me has kept my spirits and motivation high and I am forever thankful for the unconditional love and support throughout the entire thesis process and every day. To my family, I give everything, including this. Thank you for always being there when I needed you the most.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The Internet has always been a driving factor for ongoing development in the business sector since its inception. Organizations are constantly evolving to stay current and gain a competitive edge as new and improved solutions for how people utilize the World Wide Web are introduced. However, one of the most significant "evolutionary breakthroughs" has emerged on the web in the form of Web 3.0.

The term Web 3.0 was first introduced in 2014 and depicts the next generation of the internet and, potentially, of society's organization [1]. Web 1.0 was the period of decentralized, open protocols, when most internet activity required going to individual static pages [1]. Web 2.0 is the period of centralization, wherein a significant proportion of communication and business occurs on controlled platforms run by a group of mega corporations (Amazon, Google, Meta) that are nominally subject to the authority of centralized government authorities [1].

Web 3.0 is designed to liberate the internet from monopolistic dominance and may be summarized as "less trust, more truth". At its most fundamental, Web 3.0 corresponds to a blockchain-based decentralized online environment. Systems and apps created on Web 3.0 will be owned by users, who will earn their ownership stake by contributing to the development and maintenance of such applications [1]. The most important shift will be in data collection and management. It would provide the user more control over who obtains their data and how they utilize it. In the next chapters, the Web 3.0 concept and the blockchain technology that enables decentralized applications will be extensively researched and discussed.

1.1 Project motivation

The benefits of the new web could be put to use in the development of a decentralized marketplace. Currently, intermediaries that facilitate transactions between users exhibit biases (seller-bias, buyer-bias) by design, collect user data for profit, and charge fees to both sellers and buyers for the services they provide. These are the concerns that the project intends to address by developing and implementing a web-based secure, decentralized microstock marketplace. To be more descriptive, the project will develop a marketplace in which clients will be able to browse through and purchase various microstock in the form of pictures, while simultaneously providing a way for merchants to upload the before-mentioned artefacts and present the microstock to interested clients.

The project will be built utilizing blockchain technology, which implies that the system is decentralized and that the entire operations of the system are dispersed rather than performed on a central server. The rationale behind developing and implementing a decentralized marketplace where intangible 'creations of mind' are traded is that through blockchain technology the need for intermediaries is eliminated, hence resulting in a transparent, cheaper, and unbiased market, albeit a secure one that guarantees integrity of transactions and confidentiality of traded artefacts.

Because the artefacts are intangible it is critical that they are encrypted and are not viewed by unauthorized users, other than the client purchasing the microstock. This market will be beneficial to both clients, who will be able to acquire artefacts at a lesser cost, and merchants, who will receive the full share of earnings from each transaction. Furthermore, a driving factor in selecting the project is the opportunity to learn new technologies such as blockchains and smart contract. These technologies are on the rise and are extremely significant in today's environment, with great potential in the future.

1.2 Aim

With the problems described earlier, as well as the rationale for this project, the underlying aim has been devised:

"The aim of the project is to design and develop a decentralized web-application that allows users to purchase and sell microstock in a

transparent, cheaper, non-biased, and non-reputable way."

1.3 Objectives

A list of objectives has been compiled in order to work toward the project's aim. These objectives are the foundations around which the system will be constructed, and they are critical to the platform's architecture.

1. **Decentralization.**

   The developed system should be a true decentralized application that operates entirely on blockchain technology through smart contracts. The application will not collect data and will not have a central database, instead storing data on the IPFS peer-to-peer network. Users will have complete control over their data and will only share what they choose. The system will be completely transparent and permissionless, with no requirement for approval from a controlling and centralized authority, allowing all users to engage on an equal basis.

2. **Offer sampling, purchasing, and delivering of microstock.**

   This is the core functionality of the system. Buyers should be able to sample and purchase products. These products must be delivered when purchased, and funds must be transferred from the buyer's account to the seller's account.

3. **Security.**

   The system must ensure that the second objective is secure for both buyers and sellers. Centralized platforms offer security through intermediaries facilitating each transaction, but their elimination shifts this responsibility to the system being developed.

In the following chapters, each of the three objectives will be extended into more fine-grained functional and non-functional requirements. These requirements will then be analyzed, designed, implemented, and tested in order to develop the system that fulfills the project's aim.

1.4 Report Structure

A project of this scale necessitates significant documentation to support the choices made throughout development. Each chapter is specialized on documenting a certain aspect of the project. The overall number of chapters, including the introduction and conclusion, is ten, with a list of appendices offering supplementary resources. Each chapter's content include:

1. Introduction

The description of what the idea behind the system is and the stages of development start with an introduction to Web 3.0 and blockchain technology. The rationale for developing a decentralized marketplace and the project aim are also presented.

2. Literature review

A literature review is required to comprehend the environment, technologies, and tools with which the project is going to be developed. The blockchain, smart contracts, and electronic marketplaces will be thoroughly examined and discussed, including the benefits and drawbacks of each technology.

3. System description & Requirements

The aim of this section is to relate the theoretical results from the literature review with the project's motivation, understanding the decision to utilize blockchain technology to address the issues expressed in Chapter 1. A detailed description of the system and the requirements to be addresses by the system is provided in this chapter.

4. Project management

Since the amount of time and effort necessary to effectively complete the dissertation is substantial, additional procedures were adopted to organize the project as per the time allocated and as a preventative measure for any deviations from the plan. This chapter discusses the selected software development and the reasoning behind the selection, as well as the project's risks and mitigation strategies for each risk.

5-8. Sprints 1-4

Each sprint chapter consists of an **analysis, design, implementation, testing, and evaluation** section. At the start of the sprint, requirements are chosen, thoroughly analyzed, and stated how they fit into the system being developed. Then, the design is constructed, and the implementation is described in detail. Finally, each implemented requirement is tested, and the sprint is evaluated.

9. Evaluation

Chapter 9 presents a final detailed evaluation describing the achievements and the overall success of the project compared to the aims and objectives that were set initially.

10. Conclusion

The final chapter concludes the system that was developed and lists the challenges faced and how they were overcome. Lastly, a number of prospective improvements for the system are suggested as future works.

Chapter 2

Literature Review

2.1 Blockchain technology

At their core, blockchains are digitized ledgers that track every transaction. They are tamper-resistant, because any tampering is evident, and they are distributed without relying on a central authority [2], [3]. Blockchains allow a group of people to log transactions to a ledger that is distributed among them, in a manner that no transaction can be modified once logged [2]. The blockchain concept forms the basis of contemporary cryptocurrencies: electronic currency safeguarded by cryptographic processes rather than a central repository or government [4].

At a high level, blockchain-based cryptocurrency connects data describing electronic currency transactions with a digital address. Users may digitally sign their transactions, which is a mathematical approach used to authenticate a message's integrity [5], and transfer the rights to the traded assets to someone else. The blockchain publicly documents this transaction, permitting every network member to independently confirm its legitimacy [4]. A dispersed set of individuals stores, maintains, and manages the blockchain jointly. That, together with appropriate cryptographic techniques, helps make the blockchain resistant to efforts to change the ledger [6].

More closely, the blockchain, as the name implies, is made up of a series of chained individual blocks. The blocks themselves are separated into two parts; the header and the body. In the header of each block, the hash of the previous block, a timestamp, the hash of the block data, and the nonce are stored. The hash of the previous block is what turns blocks into blockchains and if a previously published block were to be altered, the hash of that block would be altered as well.

As a result, all following blocks will have different hashes since they incorporate the prior block's hash [7]. All nodes in the network use these hashes to validate transactions and this allows for the easy detection and rejection of changed blocks. The nonce is the number for which blockchain miners must solve in order to get rewarded [7]. More on this under 2.2. Below is the architecture diagram of a basic blockchain.

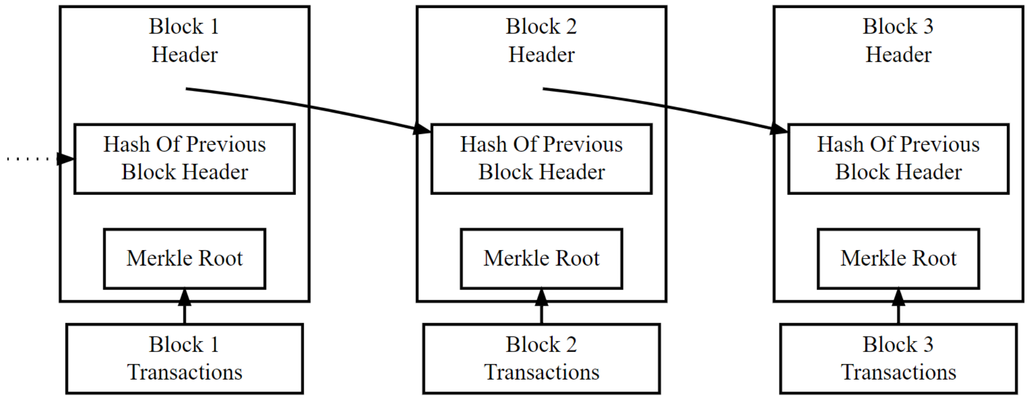


Figure 2.1: Blockchain architecture

To be acknowledged as being completed or legitimate, every transaction must be recorded to the blockchain. Firstly, a user approves a transaction via their digital wallet, seeking to transmit a specific cryptocurrency or token to some other user. The wallet program broadcasts the transaction, which is now pending to be picked up by a miner on the relevant blockchain. Miners receive their name from the fact that they are volunteers who put in a lot of effort to verify transactions in the hopes of discovering 'gold' and being rewarded [8]. More about this processes under section 2.2.1.1.

While the transaction is not picked up and placed in a block, it floats in a 'pool of unvalidated transactions'. Because the number of transactions that can be included in any particular block is restricted to an average of 1MB[1], once there is a high volume of transactions pending approval, there may be more transactions pending approval than there is capacity in a block [9]. As a result, users compete with one another to get their transaction included in the next block by ever-increasing the transaction fee until an equilibrium with a fee that users are prepared to pay is reached [10].

[1]Not always the case, check section 2.3.2.1 for more details

The transaction fee is a transaction cost that users must pay when they do transactions on the blockchain. Although the fee is not determined by the amount users transfer, it is determined by network circumstances at the time and the size of the transaction. Miners are incentivized to choose transactions with greater fees since they have a higher financial gain for each transaction they include in a block [11].

Blockchain technology is becoming increasingly popular because of its unrivalled security and capacity to give a comprehensive answer to digital identification challenges and in the next five years, the blockchain market is anticipated to be worth more than 3 trillion dollars [6].

2.1.1 Categorization of blockchains

Blockchains can be generally categorized into permissionless, permissioned, and consortium blockchains [9].

2.1.1.1 Permissionless blockchain

In permissionless blockchains there are no restrictions on membership. Anyone may use a digital wallet, which allows users to store and manage their cryptocurrencies, and write data to transactions as long as they obey the blockchain's regulations [12]. A node, or even a mining program, discussed further in section 2.2, can be run by anybody. These are often known as public blockchains, and permissionless blockchains are powering the majority of the digital money on the market. Bitcoin is the most illustrative example of a permissionless blockchain [13].

2.1.1.2 Permissioned blockchain

They are sometimes referred to as private blockchains. They function as a closed environment in which users cannot join the blockchain network unless they are authorized. It is owned by a private person or company with a centralized authority in charge of granting permissions such as who can view the transactions or who can transact [13]. The consensus process might be the same as the public blockchain, or it could be different.

Consortium or federated blockchain: Similarly to the permissioned blockchain, a consortium blockchain is also controlled. Unlike permissioned blockchains, however, consortium blockchains are owned by an association of different entities thus not relying on a single entity with centralised authority [13].

2.2 Consensus Algorithms

Identifying which miner gets to publish the next block in the blockchain, is an important feature of the technology. The problem resides in how to reach consensus, or a general agreement, among nodes that are untrustworthy of one another.

This is resolved by applying one of several different consensus models. Consensus protocols are a collection of rules used by network nodes to determine if a transaction recorded is valid. This assures that all parties have the same transaction ledger copy. In most cases, numerous publishing nodes compete simultaneously to publish the next block. Users typically do this in order to earn cryptocurrencies [14].

Consensus protocols are the foundation of a blockchain network because they guarantee integrity and consistency, resulting in an immutable and non-reputiable system.

In this section, various well-known consensus protocols will be analyzed and reviewed. These protocols are divided into two categories: proof-based protocols and voting-based protocols. Table 2.1 represents these two categories and lists the protocols within each.

Proof-based protocols	**Voting-based protocols**
Proof of Work (PoW)	Proof of Stake (PoS)
delayed Proof of Work (dPoW)	Delegated Proof of Stake (DPoS)
Proof of Importance (PoI)	Ripple Protocol (RP)
Proof of Elapsed Time (PoET)	Practical Byzantine Fault Tolerance (PBFT)
Proof of Capacity (PoC)	Delegated Byzantine Fault Tolerance (DBFT)
Proof of Authority (PoA)	Federated Byzantine Agreement (FBA)
Proof of Burn (PoBr)	

Table 2.1: Consensus protocols table

2.2.1 Proof-Based protocols

Proof-based protocols require users to display some sort of proof of effort or resource expenditure. They are generally computationally intensive, environmentally unfriendly, and less scalable than voting-based algorithms [15]. However, proof-based protocols are generally more secure and are free of a centralized entity controlling the blockchain [15].

2.2.1.1 Proof of Work (PoW)

In this model, a user publishes the next block by being the first to solve a computationally expensive problem. The solution to this riddle is the "proof" that the user has done work. The puzzle is structured in such a way that solving it is tough, yet verifying that a solution is legitimate is simple [16]. All other nodes may effortlessly validate any suggested following blocks, and any recommended block that does not solve the puzzle is discarded.

Nodes trying to solve the puzzle must generate a nonce (number only used once), attach it to the current header's hash, rehash the value, and evaluate it to the target hash. If the resultant hash value fulfils the requirements, explained under 2.2 and 2.3, the miner has solved the problem and is rewarded with the block [14].

A typical puzzle approach is to demand that a block header's hash digest be smaller than a certain number. In order to obtain a hash digest that fits the criterion, publishing nodes make several small modifications to the nonce. The publishing node must calculate the hash for the whole block, resulting in a computationally demanding procedure [16]. The goal value can be changed over time to vary the difficulty, such as obtaining a hash with 13 zeros rather than 18, and hence control how frequently blocks are issued.

The diagrams 2.2 and 2.3 show an example in which the goal of the puzzle is to obtain a hash string containing 18 zeros. The nonce picked in 2.2 was 1, and the results were unsatisfactory; however, when the nonce was adjusted to 2 in 2.3, the puzzle was solved. SHA-256 is the cryptographic hash function used most commonly. A computer generates nonces and evaluates them to determine whether they are the solution to the puzzle.

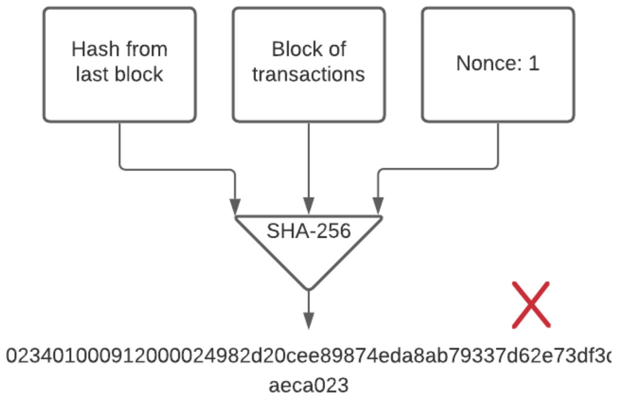


Figure 2.2: PoW puzzle not solved

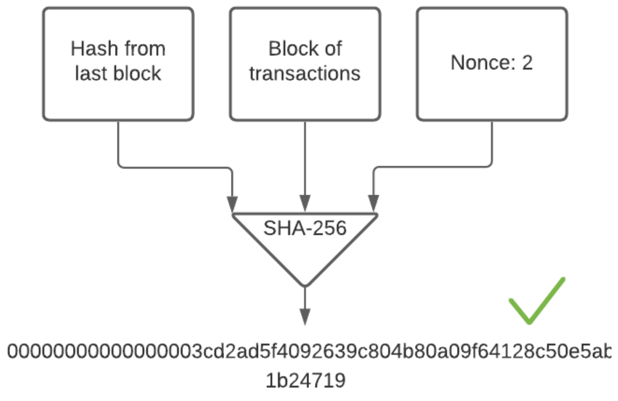


Figure 2.3: PoW puzzle solved

Modifications to the difficulty objective attempt to guarantee that no unit can control block production. That is, if the puzzle is too difficult to solve, it is inconceivable that the same unit will always solve it. However, as a result of this, the puzzle-solving calculations use a substantial amount of resources. Because PoW blockchain networks consume a substantial amount of resources, there is a push to build publishing nodes in locations where there is an abundant supply of inexpensive electricity [16].

Because the puzzles are fully independent of one another, the time spent on a current puzzle has no influence on the odds of finishing future ones, which is a crucial component of this paradigm. This implies whenever a user gets a finished and legitimate block from some other user who has solved the puzzle before them, they are encouraged to abandon their present work and begin building off from the newly received block instead, knowing that the other nodes will be doing the same [16].

Besides being environmentally concerning, PoW protocol has another issue. To evade the PoW protocol and disturb the consensus, an adversary must acquire superior computer assets when compared to the aggregate computing assets of all honest mining organizations. This is known as the 51 percent attack. Many people feel that executing this assault over a blockchain network is not economically possible; however, the introduction of mining pools and the usage of application-specific integrated circuits (ASICs) has demonstrated otherwise [14].

2.2.1.2 Delayed Proof of Work

Komodo, a multi-chain platform, employs the delayed PoW consensus technique [17]. The multi-chain platform, as the name indicates, uses the security offered by a secondary blockchain (in this example, the security supplied by solving hash puzzles in PoW) to protect blocks in the primary blockchain. This is accomplished because dPoW uses the PoW network of choice as a storage location for "backups" of dPoW transactions.

This means that transactions get stored in the Komodo (main) blockchain following the dPoW consensus, but periodically the data is backed up in another blockchain (Bitcoin) which uses the PoW consensus. In the case of a catastrophic assault, a single undamaged copy of the main chain may be used to recreate the whole Komodo ecosystem. This procedure is aided by 64 notary nodes, who are elected yearly and whose sole job is to write to the PoW blockchain. By doing so dPoW eliminates additional energy use and administrative expenses [17]. Aside from being cost-effective, dPoW is also resistant to the 51 percent problem, since an attacker must target both the main and secondary chains in order to be successful.

2.2.2 Voting-Based protocols

Voting-based protocols require network participants to share their findings of validating new blocks or transactions before making a final judgment on which node is permitted to commit a new block [18]. They are more scalable and do not suffer from the disadvantages of being computationally intensive and environmentally concerning, but they are more prone to a central figure forming and controlling the network. Thus, diminishing the decentralized property and all the benefits that come with it [18].

2.2.2.1 Proof of Stake (PoS)

This model is founded on the premise that the more stake a user has, the more inclined they are to want the network to succeed and the less probable they are to want to undermine it. Stake is frequently defined as the amount of cryptocurrency that a user has invested in the system. When a cryptocurrency is staked, it is typically no longer usable. PoS blockchain networks utilize the level of stake a member has to determine when new blocks are published [19]. As a result, the probability of a blockchain network user publishing a new block is proportional to their stake in relation to the total amount of cryptocurrency staked in the blockchain network.

The need to do resource-intensive calculations as in the POW is eliminated in the PoS model. Because this consensus mechanism consumes fewer resources, some blockchain networks have opted to abandon a block creation incentive; these systems are built in a manner that all cryptocurrency has already been dispersed among users rather than new cryptocurrency getting created at a steady rate. In these kinds of systems, the incentive for block publication is often the collection of user-provided transaction fees [19].

Regardless of the specific method, people with a higher stake are more prone to publish new blocks. This leads to an issue where the “wealthy” can more easily stake digital assets, resulting in them gaining more digital assets in the process; but, obtaining the majority of digital assets inside a system with the idea to “control” that system, is generally cost impractical [13].

2.2.2.2 Proof of Authority

PoA is a PoS protocol variation [14]. To validate transactions and blocks in PoA, trusted nodes known as validators are chosen. The network has a high throughput, is extremely scalable, and has almost zero processing fees due to the small number of validators required. In addition, unlike PoS, a validator does not need to stake any of its assets, but only its own reputation. By staking reputation (something that must be gained over time by participating in a network), this solves the core problem of PoS, which is that staking wealth results in affluent players gaining the majority of the network's incentives. However, because the whole network is in the control of a limited number of individuals, PoA leads toward strong centralization.

2.3 Smart Contracts

In 1994 the phrase "smart contract" was coined, being defined as "a computerized transaction protocol that performs the conditions of a contract" [20]. The proposal was to encode contractual clauses in code and integrate them in software or hardware that can enforce these clauses automatically. This reduces the need for traditional intermediaries among transacting parties while also having the benefit of removing occurrences of malicious or accidental exceptions in a contract [21].

Blockchain technology is extended and used by smart contracts. A smart contract is a set of code and data that is distributed on the blockchain network via cryptographically signed transactions. The smart contract is run by nodes in the blockchain network and each node that executes this contract must provide the same outcome, which is then stored on the blockchain.

Users must decide on the "if/when...then..." rules surrounding such transactions, investigate any conceivable exceptions, and create a framework for resolving conflicts in order to set the terms. Because the code is on the blockchain, it is also tamper obvious and resistant, and may therefore be utilized as a trustworthy third party. A smart contract may execute computations, store information, reveal properties to represent a publicly available state, and, if necessary, transmit payments to other accounts automatically. It should be noted that not all blockchains can support smart contracts [21].

Smart contracts have to be deterministic, which means that given an input, they must always create the same outcome. Furthermore, all nodes processing the smart contract have to agree on the new state gained upon execution [22].

2.3.1 Benefits of smart contracts

Benefits of using smart contracts to automate the execution of an agreement include:

Efficiency, accuracy and speed : When a condition is satisfied, the contract is instantly executed. Since smart contracts are digitized and automated, there is no paperwork to sort and no time wasted correcting errors that frequently occur when filling out forms manually.

Transparency and trust : There is no need to investigate if information has been manipulated for personal gain since there is no third party engaged and encrypted recordings of transactions are transmitted between parties involved.

Security : Because they are blockchain technology, smart contracts have the same security advantages of integrity, consistency and non-reputability.

Savings : Smart contracts eliminate the necessity for intermediaries to conduct transactions, as well as the time delays and fees that come with them.

2.3.2 Ethereum

Essentially, the Ethereum blockchain is state machine based on transactions. In computer science, a state machine has some internal state which in response to an external event can be changed [23].

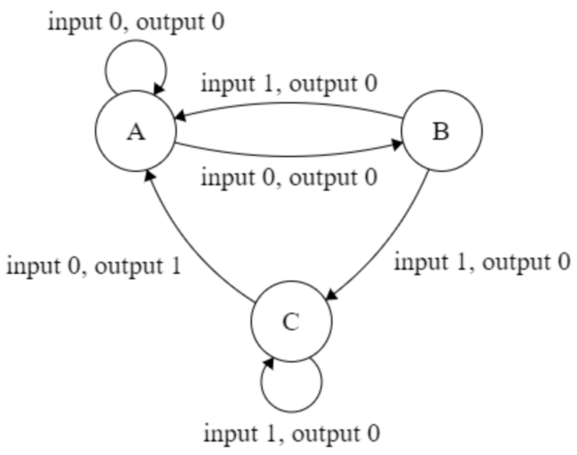


Figure 2.4: Example state machine

In the Ethereum's state machine, the initial state is the genesis state. This state is analogous to a blank paper prior to any transactions. With transactions taking place, the state transitions until the final state which represents the current status of the Ethereum blockchain [23].

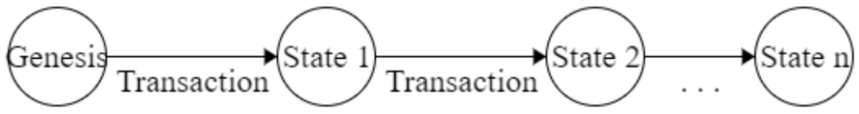


Figure 2.5: Ethereum state machine

Ethereum was designed to allow programmers to construct and deploy smart contracts and distributed apps (dApps) which can be utilized without the danger of failure, theft, or third-party intervention [24]. DApps communicate with smart contracts on the blockchain, therefore dApps integrate the front-end UI with the back-end smart contract which processes the data and send it to the blockchain.

Ethereum has been called “The world’s programmable blockchain” and it acts as a marketplace for financial services and various applications. These services are paid for using the Ethereum cryptocurrency called Ether and are free of fraud, theft, or censorship [24].

One of the primary benefits of utilizing Ethereum is the ability to develop self-running programs and distribute them in the blockchain without dealing with the high complexity of blockchain. This is feasible due to the Ethereum blockchain’s built-in programming language called Solidity which is an object-oriented programming language comparable to Java and C++. Ethereum provides a virtual machine called Ethereum Virtual Machine (EVM), which runs locally in the computer and is segregated from the main blockchain network, allowing developers to build, deploy, and test programs referred to as smart contracts in the Ethereum blockchain.

In the Ethereum blockchain, a cost is charged for each transaction or smart contract that is performed; this fee is known as ‘Gas’ [25]. This charge is used to compensate miners who offer processing power for the EVM’s execution. The quantity of gas is decided by the size of the contract code and data supplied in the transaction [25]. For instance, addition and subtraction operations (performed as part of the execution of a smart contract) cost 3 gas, multiplying and dividing operations cost 5 gas, and storing a 256-bit value costs 20,000 gas [26].

Gas costs are expressed in gwei, which is an Ether denomination and one gwei equals 0.000000001 ETH or 10^{-9} ETH. The term ‘gwei’ denotes ‘giga-wei’, and it is equivalent to 1,000,000,000 wei, which is the smallest unit of Ether [27].

Ether is a currency, and it must have inherent value in order to serve the purpose of monetary unit. Whereas, Gas is a commodity and is simply the cost associated with using the system. Because the Ether-to-Gas conversion rate is variable, the computational cost in the Ethereum is constant (when measured in Gas) but fluctuates (when measured in Ether); this rate is used as a financial instrument to adjust both computational (and hence transaction) cost and miner reimbursement. Thus, gas is essentially an interface over which transactions are carried out.

To illustrate, consider the following analogy: when calculating the cost of driving from destination A to destination B, the amount (litres) of petrol consumed is constant (assuming that driving conditions - speed, traffic, weather - are always the same); this cost measured in petrol litres has an equivalent value in Euro that dynamically varies depending on the current petrol price. Because the two are separated, the price of Gas may change without impacting the price of Ether, and vice versa [28].

The Ethereum blockchain, unlike the Bitcoin blockchain, was not designed with the main purpose of supporting a cryptocurrency. Merely, the Ether currency was established to serve as an internal currency for Ethereum blockchain-based apps [29]. That is, Ethereum has higher ambitions. It aspires to be a platform for all types of applications that want to securely store data.

2.3.2.1 Ethereum fees and block size

To understand the fees and the block size in the ethereum blockchain, it is important to mention the London upgrade. The London upgrade was an update to the Ethereum blockchain on August 5th 2021, which modified the transaction fee market and altered the way gas refunds are processed [27].

Base fee : As of the London upgrade, each block in the ethereum blockchain includes a base fee, which is the lowest price per unit of gas for insertion in a block [27]. The base fee is generated irrespective of the current block but is instead calculated from blocks preceding it. This allows for more predictable transactions to the users. Finally, the base fee is burned in order to lower the circulating amount of ether, increasing the coin's scarcity and therefore value [27].

Priority fee : Prior to the London Upgrade, miners received the whole gas cost from all transactions contained in a block. But because the base fee is now burned, users are expected to include a tip within each transaction. This tip is what miners get, hence the higher the tip, the higher the incentive for the transaction to be picked up. That is why this tip is also called the priority fee [27].

Max fee : Users can designate a maximum limit they are prepared to pay for the transaction to be executed on the network. Following the transaction, the sender receives a reimbursement for the difference between the maximum fee and the total of the base fee plus tip [27].

Block size : Ethereum had set block sizes prior to the London Upgrade. During times of peak network demand, blocks ran at full capacity. Because of this, consumers frequently simply had to wait for the demand to subside before being included in a block, resulting in a bad user experience.

After the London upgrade, Ethereum now has blocks that can vary in size. Every block has a target value of 15,000,000 gas, although the size of blocks will vary depending on network demand and it can go up to the maximum of 30,000,000 gas. Through the tâtonnement[2] concept, block sizes obtain an equilibrium of 15,000,000 on average [27]. This implies that if the block size exceeds the targeted block size, the protocol will raise the base fee for the next block. Whereas, if the opposite happens and the block size is smaller than the target, the protocol reduces the base fee for the next block. The degree whereby the base fee is changed is proportional to the difference between the current block size and the target [27].

2.3.2.2 Non-fungible tokens (NFTs)

A non-fungible token is a unit of data that is held on the blockchain. “Non-fungible” signifies that it is one-of-a-kind and cannot be substituted with anything else. NFTs are usually coupled with readily replicable assets like as images, videos, music, and blockchain technology is utilized to provide the NFT owner with public evidence of ownership [30].

NFTs work similarly to cryptographic tokens, however unlike Bitcoin or any other cryptocurrency, they are not directly interchangeable. While all cryptocurrency tokens are equivalent, every NFT represents a distinct underlying asset and hence has a different value from another NFT [30]. Despite the fact that they are not directly interchangeable, they are still an asset, therefore they may be exchanged or sold through smart contracts.

[2]Tâtonnement is a trial-and-error method that achieves equilibrium pricing and stability in competitive marketplaces

The art provided with an NFT can be freely reproduced and shared by anybody, but only one individual owns the piece. In comparison to conventional art, anybody may purchase a da Vinci print, but only one person can own the original. At the time of writing, the highest price an NFT has been sold for is $69.3 million [31].

Because they are secure, accessible, and transparent, NFTs have become financially attractive. They are similiar to comic book, baseball card, or Pokémon card collecting, however because NFTs are on the blockchain, their real scarcity (and worth) is far less uncertain since an undeniable record of each token exists.

2.3.3 Hyperledger

Hyperledger is an open source initiative that was designed to aid in the development of blockchain-based ledgers. Hyperledger is a collective approach to provide the frameworks, protocols, algorithms, and libraries required to create applications related to blockchains [32].

From its inception by the Linux Foundation in 2016, the Hyperledger project has benefited from donations from companies such as IBM & Intel, as well as Microsoft, American Express, Samsung, and Visa. Overall, the cooperation spans finance, manufacturing, supply chain management, production-related disciplines, and internet of things [32]. Hyperledger seeks to develop blockchain technology by implementing an open standard platform for distributed ledgers across the industry.

Hyperledger serves as a central point for several distributed ledger systems and libraries. Through Hyperledger, a company may, for instance, employ one of its frameworks to maximize the productivity, speed, and transactions in their business operations [33].

Hyperledger facilitates the development of blockchain applications and systems by ensuring the adequate infrastructures and protocols. Hyperledger Greenhouse (the frameworks and services that comprise Hyperledger) is used by developers to create commercial blockchain initiatives. Users in the network are all aware of one another and may engage in consensus-making procedures [33].

The following layers are used by hyperledger-based software:

- Consensus algorithm layer

- Smart contract layer

- Communication layer (handles peer-to-peer communication)
- Application programming interface (API) layer (enables other apps to connect to the blockchain)
- Identification management layer (verify users' and systems' credentials)

Hyperledger Fabric is the most well-known and used framework.

2.3.3.1 Hyperledger Fabric

Hyperledger Fabric is the most prominent Hyperledger project. It is a permissioned blockchain architecture that may be utilized to develop blockchain-based applications and services.

The project was created in collaboration with IBM and Digital Asset and it offers a modular design that specifies node responsibilities, smart contract enforcement, and adjustable consensus algorithms. Through module installation the project supports various programming languages, enabling a higher variety for developers. It is best utilized in integration initiatives that necessitate the deployment of a distributed ledger.

2.4 Blockchain Limitations

It is often the case that emerging technology is overhyped and overutilized. Many initiatives will strive to include emerging technology, even if it is not required. The blockchain technology has not been immune to this trend. This section discusses some of the limits and disadvantages of blockchain technology.

2.4.1 Scalability issues

The greater the number of transactions, the larger the blockchain that each node must store and process. Furthermore, due to limitations in the size of the blocks and the time necessary to build new blocks, it is hard to process a large number of records in a timely manner. Increasing the size of blocks in an attempt to remedy this issue may result in the blockchain being too huge for the ordinary node to handle, resulting in greater centralization and, as a result, less trust [34]. The elimination of outdated transaction data is one potential solution to the size issue. Another proposal argues for a complete redesign of the blockchain architecture, such as scale-out blockchains in which miners do not need to know and confirm all transactions [34].

2.4.2 Privacy issues

Because users may perform transactions with created addresses rather than a real identity, blockchain is thought to guarantee security and anonymity for sensitive personal data. This is however not completely true, since the public key used to initiate a transaction is accessible to network peers [35]. The key along with the transaction history of a user can be connected to expose a member's actual identity.

2.4.3 Blockchain attacks

Another disadvantage of blockchain technology is the variety of attacks that may be done on it. The most prevalent attacks are outlined and discussed here.

2.4.3.1 51% attack

The 51 percent assault on consensus entails an attacker or group of attackers possessing more than 50 percent of the network's compute capacity, as the perpetrators can double spend in this instance. This refers to a problem in which a single digital token can be spent more than once [36]. This is possible because if an attacker has a hashing power greater than the rest of the network combined (51%), then they can spend money on the valid blockchain and then eventually build their fraudulent version of the blockchain (in which they did not spend the money) to be the longest chain. Due to the longest chain rule, which specifies that the longest chain is what individual nodes acknowledge as the valid version of the blockchain [37], the fraudulent version is now considered the valid one.

Although a 51 percent assault on Bitcoin has never been accomplished, the four largest mining pools on the Bitcoin network currently make up more than half of its processing power [38]. Only four separate entities working together may utterly destabilize the system. Therefore, opposite to the original plan for Bitcoin's decentralization, four actors could centralize the network's authority [39].

2.4.3.2 Selfish mining

An adversary with less than 51 percent of the network's mining power can use the selfish mining approach to acquire monetary advantages or conduct double-spending assaults. Initially, the selfish miner does what miners are meant to do: they strive to grow the longest chain. However, after creating a block by solving a puzzle, they keep the block hidden rather than release it, and then attempt to expand it further, constructing a secret branch.

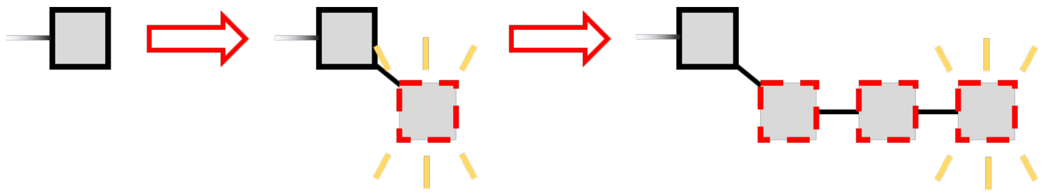


Figure 2.6: Selfish miner expanding a private blockchain

At the same time, the other miners lengthen the public chain by finding solutions to puzzles. The selfish miner keeps extending their hidden branch until the public chain falls one step behind. Once this has happened, they reveal their secret chain.

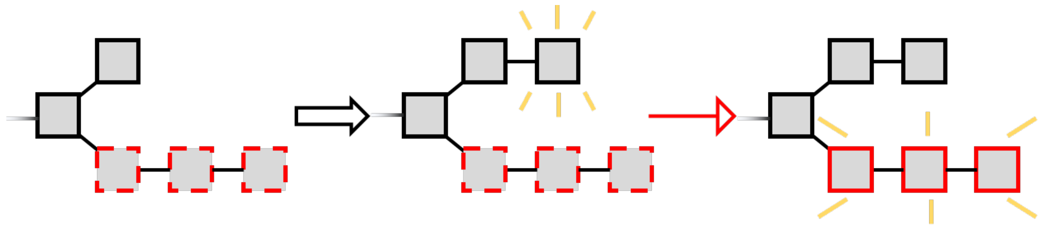


Figure 2.7: Selfish miner publishes the private blockchain

Because the hidden chain is longer, the other parties regard it as the main chain, and everyone is therefore now following the selfish miner's blocks. The blocks created by the other miners are therefore pruned - disregarded, and their producers receive no reward.

The malevolent participant leads the network to converge on its state by producing a fork that is longer than that of honest miners. As a result, if the attacker controls at least 25% of the network's total processing capacity, they can successfully conduct double-spending assaults [39].

2.5 Electronic Marketplaces

With the diffusion of internet use, electronic markets (EMs) have become one of the most profitable types of businesses. In 2020, 47 percent of total e-commerce sales were realized through EMs, reaching over two trillion dollars [40]. Granted a significant push by the COVID-19 outbreak, EM popularity is expected to increase considerably over the next five years as more businesses use marketplaces as the ideal venue for promoting online sales [41].

Although many definitions exist, an EM is a type of e-commerce that is usually defined as a venue on the internet where companies or individuals are able to purchase or trade products and services [42]. As a result, sellers/providers have more business options, and customers have a wider range of choice and value.

There are several approaches to categorize an EM, but the most commonly utilized [43], [44], considers the buyer-seller relationship to produce four types. This classification can be seen under table 2.2.

Seller/Buyer	**Consumer**	**Business**
Consumer	C2C	C2B
Business	B2C	B2B

Table 2.2: EMs Classification

Consumer to consumer : A market setting in which consumers purchase and sell goods to other consumers. Online auctions are a common form of this category, in which users may post products for bidding and other users can bid on the items for a certain period of time. Most popular C2C marketplaces include eBay, Amazon Marketplace, and Mercari.

Consumer to business : The least common business paradigm in which customers sell good or services to businesses. Example of this model include marketplaces where customers sell their old cars to car dealerships.

Business to consumer : The most common business model where businesses sell goods to consumers. This EM model often replicates physical commerce in an online context. Examples include the biggest marketplace Amazon.

Business to business : This model allows businesses to interact with other organizations and do business in a single location. Here, items are often sold in bulk, and businesses acquire materials in order to further process and resell them at a larger profit. Examples include Alibaba.

2.5.1 Benefits of EMs

The benefits of participating in an EM will differ depending on the industry and business, as well as between sellers and buyers. A few of the potential advantages are listed below [45].

General benefits

- There are more options for sellers and customers to form new commercial partnerships, either inside or across supply chains.
- Considering supply, pricing, and stock levels are all available in an open setting, EMs can give increased transparency in the shopping experience.
- Time limits and issues with varied business hours for foreign trade are no longer an issue because it is feasible to work around the clock.

Buyer benefits

- Pricing and availability details that is constantly updated makes it easy to get the optimal offer.
- Instead of visiting each particular supplier, e-marketplaces provide an easy method to evaluate products and prices from a single source. This is accomplished automatically by sites that aggregate/compare material from different sites. More alternatives lead to more competition, which leads to reduced prices.
- Because they only deal with suppliers that are trusted, established EMs guarantee a degree of confidence for the buyer.

Seller benefits

- Adds a large customer base, otherwise not accessible if the shop was only brick and mortar.
- If compared to traditional sales channels, EMs provide lower marketing expenditures through removing the need for a brick and mortar shop, thus lowering costs.
- The utilization of foreign EMs can open up prospects for international sales, that would be impossible otherwise.

2.5.2 Trust factor in EMs

For commerce (electronic or conventional) to be successful, sellers and buyers must have a level of trust. As a byproduct of the trust problem, centralisation has come as a solution. In conventional business, this type of trust is usually established through 3rd parties (Banks) that remove the need for people to trust one another directly. Besides this solution, with blockchain technology advancements, a new approach of a decentralized way of handling the trust problem has emerged [46]. Both the centralized and the decentralized EMs will be analyzed and their advantages and disadvantages will be discussed below.

2.5.2.1 Centralized EMs

In a centralized EM, every trust issue is addressed by depending on a reliable third party. They take accountability for the marketplace's overall stability. The other participants rely on them to regulate the platform (for example, by eliminating any problematic participants), secure it with up to date technology, and arbitrate any conflicts. However, depending only on reputation as a trust-building element might create entry barriers to new vendors, making customers less likely to purchase from them [46].

Additionally, the centralized authority that facilitates the transactions takes a fee for doing so. Both the seller and the buyer are obliged to pay this fee resulting in higher prices for buyers and less profit for sellers in each transaction. Depending if the EM is seller biased or buyer biased, the fee might favor one or the other [46]. This is another issue that centralized EMs are always biased. Moreover, centralized EMs can also be biased among the sellers utilizing the site (favoring some and promoting them for any reasons).

Lastly, centralized EMs suffer from in the privacy aspect. The data of buyers and sellers are all owned by the company monitoring the transactions, and not by the users themselves. This data is then subject to be processed, abused, or sold for profit [46].

2.5.2.2 Decentralized EMs

With the advent of blockchain technology, a solution to the problems associated with centralized EMs has become possible.

Decentralized marketplaces leverage distributed systems technology to overcome the majority of the drawbacks of the old centralized approach. A decentralized EMs is a marketplace on the blockchain network that allows merchants or investors to transact with one another directly and do not need the use of middlemen to facilitate deals [47].

There is no need for trust among the traders, since each trader has a duplicate of the identical data, and if the transaction requirements are not satisfied or the data is manipulated or damaged, the transaction will not be completed [47].

Centralized markets frequently charge greater fees, offer low transparency, and impose regulations that consumers may not want to follow. Centralized EMs also pose increased security concerns since the network is based on a single source, increasing the likelihood of failure or attack. Through smart contracts, described under section 2.3, users are able to trade directly with one another mitigating all of the flaws that arise in centralized marketplaces.

Decentralized EMs offer higher transparency since traders all voluntarily agree on the information and data that is exchanged and no important data is shared with third parties. Because traders are responsible for their own data, there is no possibility of a central figure exploiting this data or a data breach occurring, thus preserving the privacy of the users.

Lastly, intermediaries that oversee transactions require a fee as a tax for facilitating each transaction. Because there is no intermediary monitoring transactions in decentralized marketplaces, this charge is eliminated altogether, resulting in lower transaction costs for both vendors and buyers [47].

2.6 Similar Projects

Decentralized marketplaces have been attempted in a variety of formats, with varying degrees of success. This section examines two general paths of decentralized markets, the first and most popular of which are NFT marketplaces, and the second, the curious example of OpenBazaar.

2.6.1 NFT Marketplaces

Although few people appear to recognize what non-fungible tokens are, this does not seem to have been an issue for the NFT market capitalization which reached \$23 billion in 2021 [48]. Consequently, the most popular form of decentralized electronic marketplaces has been marketplaces where users can buy and sell NFTs.

At the time of writing, the number of active decentralized marketplaces for NFT trading reaches a little over 40, with the most popular being: **OpenSea, Magic Eden, CryptoPunks, Solanart, Rarible,** and **Foundation**.

These platforms enable users to create NFTs for free and then auction or sell these NFTs for cryptocurrency. Although they all follow the same fundamental concept, they differ in the blockchain platform on which they are implemented. For example, OpenSea (the most popular NFT marketplace currently) is built on the Ethereum blockchain, whereas Magic Eden (the second most popular) is built on the Solana blockchain.

Although they employ smart-contracts, therefore mitigating the need to facilitate transactions, these markets stay lucrative by charging either a listing charge or a selling fee [49]. A listing fee is a cost charged to users for listing their NFTs on the platform, however this is not the most prevalent solution because the marketplace would no longer be 'free' for consumers. As a result, charging users for each sale they make has become the norm. NFT marketplaces may charge either sellers or buyers, with a seller's fee deducted from the overall cost and a buyer's fee added to the total cost. For instance, if a user sells an NFT for 100 ETH with a 5% seller's fee, they will receive 95 ETH, whereas with a buyer's fee the buyer will pay 105 ETH and the user will receive the requested 100. The most popular NFT marketplace, OpenSea, collects a 2.5% commission on each transaction, which is paid by the seller [49].

2.6.2 The interesting case of OpenBazaar

After the FBI closed down the infamous darknet marketplace 'Silk Road' which facilitated the sales of illegal products, a team of developers created a new marketplace called OpenBazaar [50]. OpenBazaar is an open source project that aimed to create a completely decentralized C2C marketplace where users could sell and buy anything without extra fees or intermediaries.

On the surface, OpenBazaar appeared to be similar to Silk Road, but with one major difference: it was completely decentralized, which means it wasn't hosted on servers in a single centralized spot. Rather, the application was hosted on each of its customers' machines, which is referred to as a peer-to-peer network. Although the default OpenBazaar search engine excludes out undesired products like narcotics, because the project is opensource, the developers have no control over sellers displaying such items on the marketplace [50].

Although there is evident network activity and a rather significant number of products accessible on the OpenBazaar platform, economic activity remains small and seems to be primarily driven by illegal sales revenue [50]. Moreover, over two-thirds of customers remained on the platform for less than a day before leaving. Sellers stayed a little longer, particularly if they recorded at least one purchase. Approximately three-quarters of all users utilized the platform for less than a week [50].

Furthermore, because OpenBazaar runs on a peer-to-peer infrastructure, each market participant must install specific software to enter the market rather than accessing the marketplace on the web like in centralized markets. The network's limited number of vendors offering genuine items discourages legitimate users from joining, while the network's low number of legitimate consumers discourages vendors from selling on it. Until some changes are made to the network, OpenBazaar may continue to be the ghost town that it is now.

Chapter 3

System Description & Requirements

The system being developed is a web application that allows users to purchase microstock (in the form of pictures) in a secure and decentralized manner. Because the system will leverage blockchain technology, the necessity for an intermediary to facilitate transactions is eliminated, resulting in lower-cost products for consumers and larger revenues for sellers. Furthermore the system enables for transparent transactions that dispense the need to place trust upon a third party intermediary, thus also avoiding biases (seller-bias, buyer-bias) that such intermediaries may exhibit by design.

Conventional web applications are usually comprised of a frontend and a backend. On the frontend, decentralized applications and web applications use the same technologies to render a page. However, on the backend, decentralized applications interact with their respective blockchain networks through a wallet which serves as a gateway towards the blockchain ecosystem. Wallets store the blockchain address and the cryptographic keys required to identify and validate users. Rather than using the HTTP protocol, decentralized application wallets invoke smart contracts which communicate with the blockchain and perform transactions. Although a dApps's user interface may look comparable to that of a web application, it varies in that it excludes servers, HTTP communications, and possible censorship.

Section 3.1 presents a theoretical model of how the intended system will operate. Whereas, in section 3.2, a complete list of the requirements that this system must meet will be analyzed and discussed.

3.1 Theoretical model

The following Figure 3.1 provides an overview of how users interact with the system. The system's business logic and how users can sell and buy microstock securely through blockchain will be discussed and explained in depth in the following chapters.

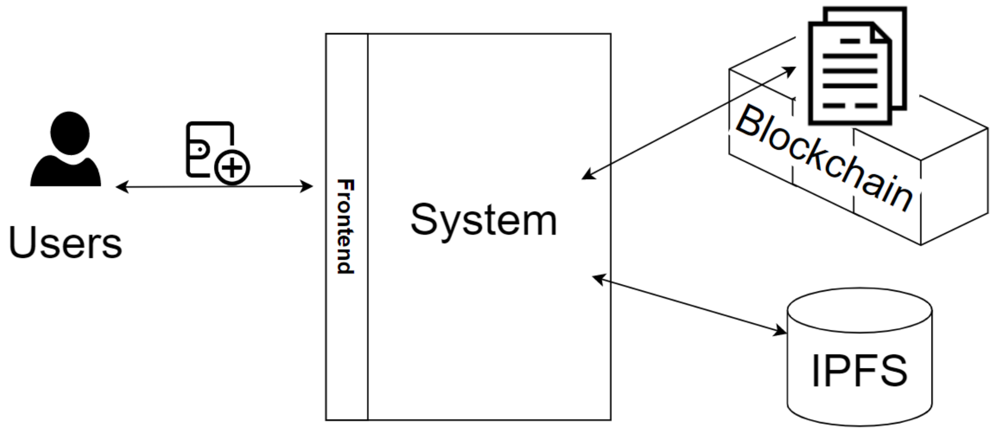


Figure 3.1: Overview of the system components

- To interact with the system new users connect their blockchain wallets.
- Users make requests to the frontend after connecting. Depending on the kind of request, it is either delegated to the blockchain's smart contracts, IPFS, or handled directly on the frontend.
  - File uploading and retrieval requests are delegated to the Inter Planetary File System (IPFS), which serves as a decentralized database and will be covered in depth in later chapters.
  - Any request pertaining to the system's business logic is delegated from the frontend to the smart contracts deployed on the blockchain.
  - The final form of request occurs when users perform encryption and decryption of files or keys. This request is handled directly on the frontend, and no data is stored.

3.2 Requirements and Analysis

The system requirements are a specification of the intended system's characteristics and functions. They convey the users' expectations of the system [51]. Having a clear set of requirements is a necessity for developing any application as they provide a vision of the final goal for the system. The system being designed for this project must assure that its functions are not only appropriate and adequate, but also deliver a pleasant user experience.

Requirements are classed as functional or non-functional [51], and this section will provide a high-level overview for each requirement, with more detailed descriptions in subsequent sections.

3.2.1 Functional requirements

Functional Requirements represent the criteria that the end user expects the system to provide as functionalities [52]. Functional requirements typically specify the interactions that a system offers to its users, as well as the system's reactions to these interactions.

Concerning the marketplace, the system supports two types of users: buyers and sellers, each with their own set of user needs. The list below depicts these requirements[1]:

E - **Connect wallet:** Users will need to connect their blockchain wallet in order to interact with the smart contracts deployed on the blockchain.

E - **View purchase history:** For transparency, users should be able to view the entire purchase history.

B - **View available products:** Buyers should be able to view a list available products that they can purchase.

B - **Request a sample:** Buyers should be able to request a sample product.

B - **Purchase products:** Buyers should be able to purchase products that they desire through the system.

B - **Receive products:** After purchasing a product, buyers should receive the entire product securely.

[1]E = Every user; B = Buyer; S = Seller

S - List products: In order for buyers to have products they can purchase, sellers should be able to upload the products they want to sell.

S - Deliver sample: When sellers have interested buyers, they should be able to provide sample products.

S - Receive payment: When sellers provide the full product, they should receive the amount they requested for the product.

3.2.2 Non-functional requirements

Non-Functional Requirements (NFR) provide quality attributes to software systems [53]. They evaluate the software application depending on its security, portability, usability, availability, and other non-functional characteristics that are crucial to its success [53]. "How quickly does the webpage load?" and "Does the system offer confidentiality" are instances of nonfunctional requirements. If these requirements are not met, the overall system will not meet the demands of the users. The detailed description of NFRs is just as crucial as the description of functional requirements.

-Security : The system security is realised through three parts: access control, confidentiality and integrity of traded artefacts, and non-repudiation of transactions. To maintain system stability and prevent any compromises between the two designated user categories (buyers and sellers) the authentication and authorization processes will be employed through the identification of wallet addresses, thus ensuring proper access control.

Furthermore, the system should provide confidentiality and integrity. The term confidentiality refers to the practice of ensuring that the artifacts being sold, excluding the samples, are only viewed by authorized legitimate buyers and no one else. System integrity relates to the correctness and quality of the exchanged artifacts in the system [54].

Finally, the system must guarantee non-repudiation. Meaning the system ensures that a user cannot refute the actions they performed. More specifically, a user cannot deny actions taken to purchase artifacts.

-Usability : The goal of this project is to produce a user-friendly system that is unobtrusive and appealing for individuals to utilize. Substantial attention will be put into the user interface design in order to meet the usability requirements for performance and quality. Furthermore, the functionality offered to the user should be readily available and the situation in which the user gets caught in menus and links with the possibility of hitting dead ends should be strictly avoided. Having a poor user experience hinders even the best systems, therefore usability is a strong NFR. To ensure the usability requirement is met this project will follow the principles on Nielsen's usability criteria [55]. Finally, in the event of an error, the system will offer enough feedback to its users in order to assist them in recovering from these errors.

-Maintainability : To lessen the complexity of resolving errors and problems, the system should be easily maintainable. The ideal technique is to have well-organized code that has been thoroughly refactored, so that locating the needed piece of code is easier and there is more adequate time for correcting errors and increasing functionality. This implies that the code must be modular, with each functionality divided into distinct modules.

-Extensibility : The system will be built to be modular, with the expectation of future additions. This project may be expanded in a variety of ways to improve the process of open and secure trading, and the requisite infrastructure should be in place to enable it. Ideas for future developments will be discussed in subsequent sections.

Chapter 4

Project Management

Spending a small amount of effort in project management may save a significant amount of time, money, or other resources that would otherwise be lost if a preventable crisis occurred and no recovery plan was devised. A good practice is to establish a risk management strategy from the beginning of the project, examining potential circumstances and issues that may arise throughout the development period. Additionally, developing a thorough timetable and adhering to it as much as possible minimizes delays and guarantees that targets are met. Lastly, and most crucially for a software development project, the software development process should be carefully chosen, thoroughly explained and well understood.

These techniques will be beneficial in enhancing both the development procedure and the end result, and in the case of this project, substantial effort will be taken to ensure that the project adheres to appropriate management principles throughout all development phases.

4.1 Software Development Process

Software development is often a challenging and lengthy endeavor, making it difficult for developers to approach projects and assess their progress at various phases. A software development process (SDP) is necessary to address these challenges. SDPs provide a process for project planning and thorough scheduling of project stages [56].

Depending on whether these objectives are firmly defined or subject to modifications and revisions, different software development processes are more suited than others. The waterfall, incremental, and iterative development processes will be examined in this section as they are the most influential, each providing a basis for various schools of thought on software development.

Waterfall model - One of the earliest SDPs is the waterfall model. It proposes a method for streamlining work phases such that each step reflects a distinct part of the project [57]. It has the benefit of using a clear and straightforward structure, placing the most emphasis on a well-defined set of stages. These phases include obtaining requirements, analyzing, designing, implementing, and testing [57]. Because of this linear approach, the waterfall model necessitates specific requirements which will not alter throughout the development phase. As a result, it is an extremely rigid approach that does not allow for any flexibility or adjustment [57]. Furthermore, postponing testing until after completion is dangerous because if errors are discovered, it may be too late to correct them. Given the unfamiliarity with new technology and the fact that requirements are prone to change, this model will not be employed for this project.

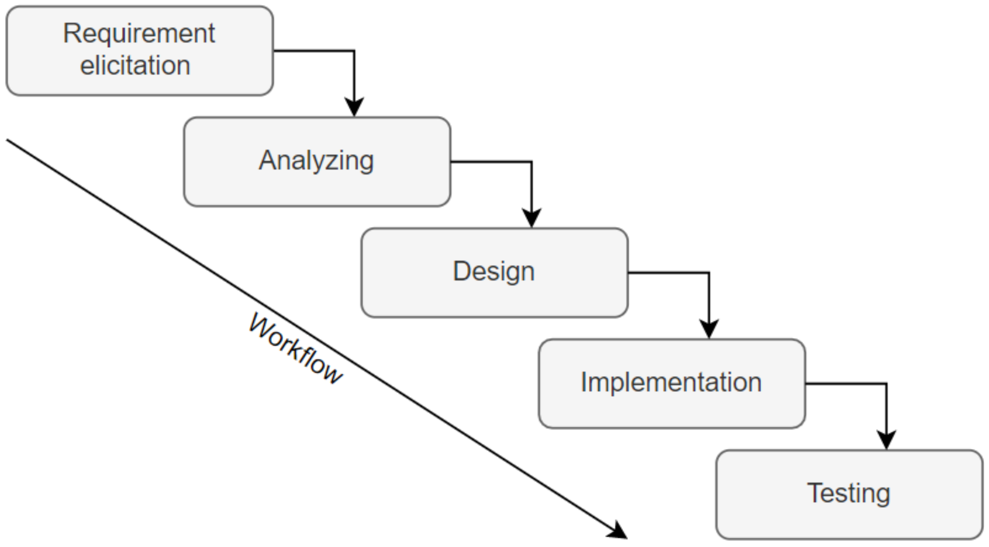


Figure 4.1: Waterfall model diagram

Incremental model - The incremental model shifts its focus, applying the many characteristics outlined in the waterfall methodology to each functionality. That means that with every new group of features added to the project, there is an analysis, design, implementation, and testing phase [58]. Because new requirements may be introduced throughout development, it is far more adaptable than the waterfall approach. Furthermore, time will not be lost on discarded requirements in case these requirements have not yet been selected to be worked on. Early and repeated testing, maintenance and extensibility achieves better time and cost savings [58]. Through testing and maintenance, issues are discovered early in development. Early detection of issues enables for easier management of these problems. This is accomplished because solutions to smaller iterations are simpler than issues discovered at the end of a full project (like in the waterfall model), and because it is discovered early, it has more time to be solved.

The incremental model necessitates well defined interfaces between different components and does not provide a comprehensive picture of the system before a considerable number of increments are accomplished [58]. As a result, the incremental method is unsuitable for projects that require an early working prototype, which this project expects to have before progressing with further development.

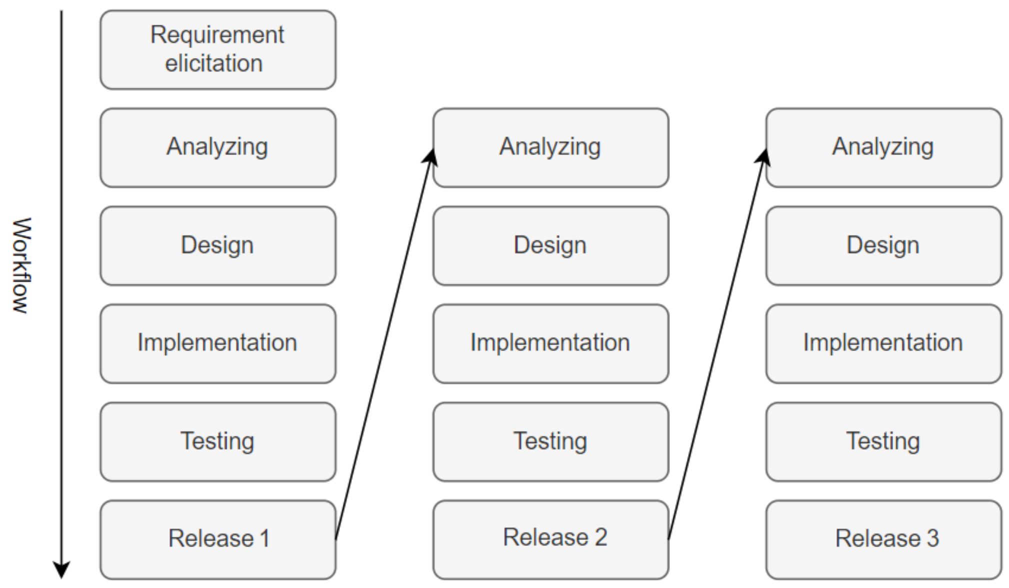


Figure 4.2: Incremental model diagram

Iterative model - The iterative model, like the incremental approach, divides the work into cycles of analyzing, designing, implementing, and testing. What distinguishes it is the deliverables released at the conclusion of each iteration [51]. The incremental model shows incremental steps that will ultimately result in a full system, whereas the iterative approach develops a more conceptual implementation of the entire system and afterwards tries to enhance this implementation throughout each cycle. This implies that an early proof of concept will be developed, and it will iteratively be transformed into a comprehensive system across the subsequent iterations.

To illustrate, figure 4.3 depicts the difference between incremental and iterative models.

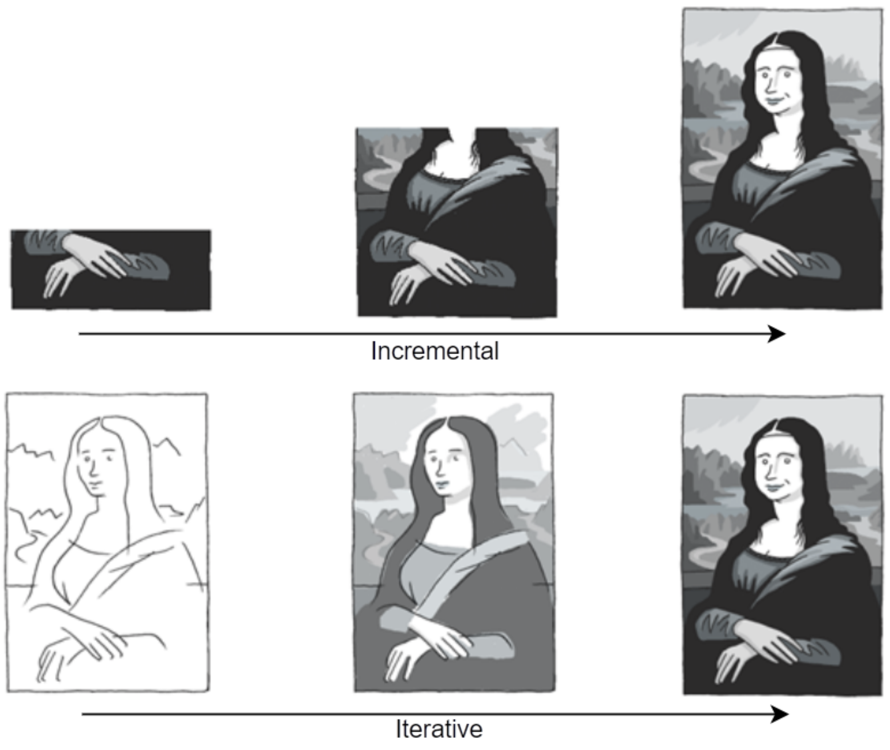


Figure 4.3: Incremental and Iterative difference

Agile methodology / Scrum - The capability to develop and adapt to change is referred to as agility. It is a method of coping with, and ultimately prevailing in, an unpredictable and chaotic environment. Agile development is a hybrid of the iterative and incremental techniques. Work is divided into iterations in agile development, but each cycle must deliver something of significance [59]. Meaning at the conclusion of each cycle, something functional, such as additional feature, should be produced. However, these functions, or increments, are not required to be flawless at the conclusion of each iteration. Agile development refers to a collection of frameworks and techniques and it is more of a mindset or set of standards, and it is left to other models adopt this philosophy [59]. Scrum is one such example, where iterations are divided into sprints. Sprints, as the name implies, are short-term iterations that generally last 3-5 weeks that culminate with a self reflection in case of a solo developer (solo-scrum) or a meeting in case of multiple developers [60]. This meeting / self reflection serves as a tool to reflect on the work done during the sprint, challenges faced, and to devise a plan for the next sprint. Scrum is a highly adaptable work style that incorporates the advantages of both the iterative and incremental methods [60].

Final thoughts / Decision - At first, when taking into consideration that for this project:

- There are functional requirements that have been obtained, but there is a strong likelihood that they may change or that additional ones will be introduced later during the development process;
- The software must be examined on regular intervals to ensure that it is meeting the project goals;
- The software is modular, built by combining components with defined interfaces;
- The iterative approach allows for an early implementation, with an early functioning version of the product to be available;
- Running tests in each iteration allows for easier error detection;

The iterative approach appears to be the best choice; however, because it is difficult to judge the final goal of the project at the start, the incremental technique may result in increments of dramatically different proportions. Additionally, the increments have a high probability of needing to be revisited, which is more analogous to an iterative strategy. By integrating the two models together, workflow gets extremely flexible while remaining ordered. Thus, the technique chosen for this system will be the Scrum model.

The solo-scrum paradigm necessitates a high-level specification of the sprints that will be included, with each sprint having a particular theme they must adhere to. Themes can include many features as long as they are closely connected. Prior to Spring 1, an inception stage of tasks that must be finished before any sprints is also included.

Inception (October 21 - December 21) : The inception step is where the majority of the research is done. Throughout this period, the developer writes a literature review on concepts relevant to the project. Furthermore, a risk analysis and management plan will be devised, and the themes of each sprint will be decided on. Lastly, a conceptual model and the collection of a preliminary set of requirements will be completed.

Sprint 1 (January 22 - February 22) : The first sprint will set up the blockchain environment and serve as an opportunity to get familiar with the new technologies. At the end an evaluation will reveal if the requirements were met, and the plans for the following sprints will be revised.

Sprint 2 (February 22 - March 22) : During Sprint 2, the factory pattern approach will be introduced to the system. Additionally, the Inter Planetary File System network will be implemented to serve as a decentralized database.

Sprint 3 (March 22 - April 22) : During Sprint 3, a detailed analysis, design, and implementation of the security system will be introduced. At the end of this sprint, all major requirements will be fulfilled.

Sprint 4 (April 22 - May 22) : The wrap-up sprint will serve as a way to improve the user experience. The frontend will be redesigned, and different encryption tools will be added directly to it for users to utilize in encrypting and decrypting files and keys.

4.2 Risk Management

Risk management is a vital component to consider given the project's workload and complexity, since there is room for a variety of issues to occur. Its purpose is to prepare developers for any problematic scenarios that may arise throughout the development phases, so that they can deal with them effectively.

Under this section table 4.1 depicts the project-related hazards that have been recognized. There are five sorts of hazards that the development process may encounter, including requirements, technologies, estimating, and hardware. Additionally, each risk is documented in further depth in the form of tables, with a definition and a mitigation plan for how those risks will be avoided or addressed. In the table 4.1 the columns represent:

Probability – this column reflects the likelihood of a risk occurring before or after a certain activity; probabilities are given as percentages ranging from 0% to 100%.

Impact – this column shows the severity of the repercussions that a certain risk may have when it occurs; the values are given on a scale of 1 to 5, with 1 indicating minor consequences and 5 indicating significant ones.

Exposure - this column represents the project's exposure to particular hazards. Its worth is determined by the product of the first two columns (probability and impact). This number is a measure that is used to rank threats. A risk can be less harmful over another even though it is of greater severity, if the likelihood of it occurring is low. As a result, exposure attempts to balance impact and likelihood in order to effectively prioritize risks.

Risk	**Description**	**Impact**	**Probability**	**Exposure**
1	Unfamiliarity to new technology	4	80%	3.2
2	Time management issues	4	70%	2.8
3	Changing requirements	4	30%	1.2
4	Project size underestimation	3	40%	1.2
5	Underestimated number of errors	3	30%	0.9
6	Hardware breakdown	4	10%	0.4

Table 4.1: Risk table

A mitigation plan, which represents an action to to reduce the severity or probability of the risks, is developed for each threat.

Unfamiliarity to new technology	
Risk Description	**Mitigation strategy**
The technologies employed during development are unfamiliar to the developer, and it may take some time to become acclimated to them.	Allocate adequate time for research and reading documentation on relevant technologies. Develop a functionality that makes use of all of the proposed technologies early.

Table 4.2: Risk 1 and the mitigation strategy

Time management issues	
Risk Description	**Mitigation strategy**
The scope of the offered functionalities is not accurately measured, therefore time is not correctly allocated.	At the end of each iteration, time is evaluated, and the most critical features are prioritized in order to detect problematic areas early on.

Table 4.3: Risk 2 and the mitigation strategy

Changing requirements	
Risk Description	**Mitigation strategy**
During development the requirements may change, resulting in additional effort or lost development time.	Choosing an iterative software development process allows for better adaptability towards changing requirements.

Table 4.4: Risk 3 and the mitigation strategy

Project size underestimation	
Risk Description	**Mitigation strategy**
The amount of work required at each iteration is not accurately anticipated and is underestimated.	Evaluate the completed work and the remaining work on a regular basis to adapt the schedule and timeframe.

Table 4.5: Risk 4 and the mitigation strategy

Underestimated number of errors	
Risk Description	**Mitigation strategy**
The number of errors is understated, and a significant amount of effort is spent on resolving them.	At the completion of each developed functionality, do both manual and automated testing. Ensure that the tests cover unexpected cases.

Table 4.6: Risk 5 and the mitigation strategy

Hardware breakdown	
Risk Description	**Mitigation strategy**
The hardware malfunctions, resulting in the loss of progress or crucial files.	Backups should be performed on a regular basis using version control systems and external/cloud storage.

Table 4.7: Risk 6 and the mitigation strategy

Chapter 5

Sprint 1

Sprint 1 is the initial development sprint, divided in four phases: analysis, design, implementation, and testing. The stages will be executed in that sequence, and after the conclusion of the testing stage, the sprint as a whole will be assessed.

Once completed, the assessment will be used to define the scope of the following sprint. These assessments will also act as indications of the system's progress over time.

5.1 Analysis

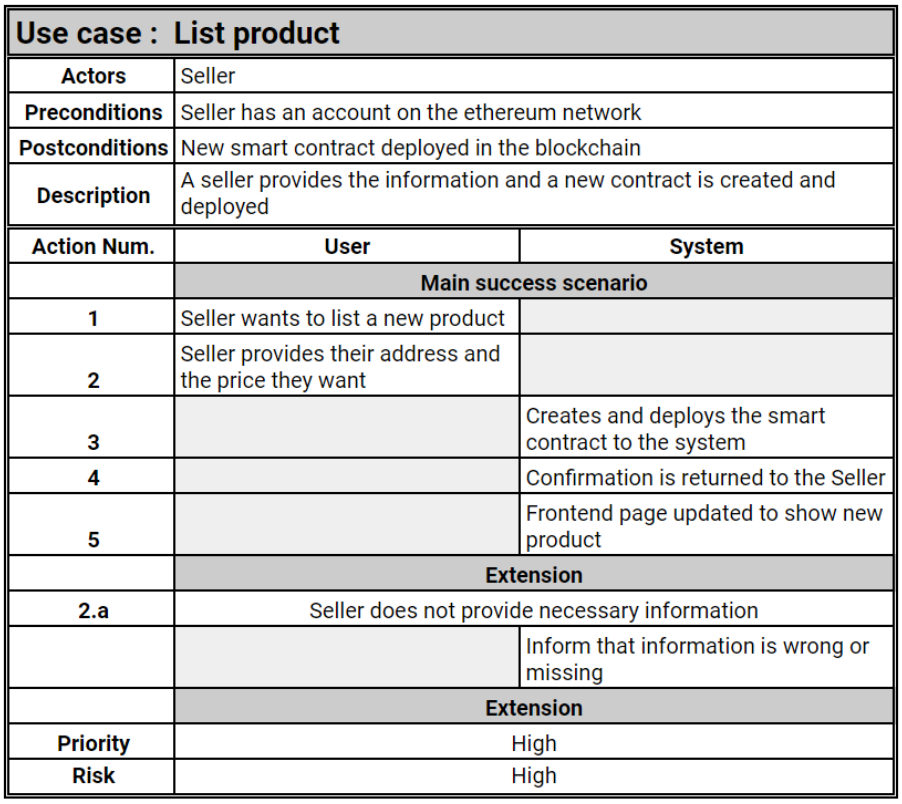
The initial requirements that are critical to the system and need to be addressed first are:

- List new products
- Purchase products

These requirements form the core of the system's functionality, and each subsequent requirement will include, expand on, or be connected to these requirements. More precisely, the focus of the sprint will be on the development of the smart contract responsible for transactions between buyers and sellers. This contract will be deployed to a test blockchain and have a functional frontend allowing for interaction with the contracts logic.

To further analyze the requirements, use cases are a valuable tool that help illustrate the process flow of the chosen requirements. Two use cases have been designed for the two requirements that are going to be tackled under this sprint.

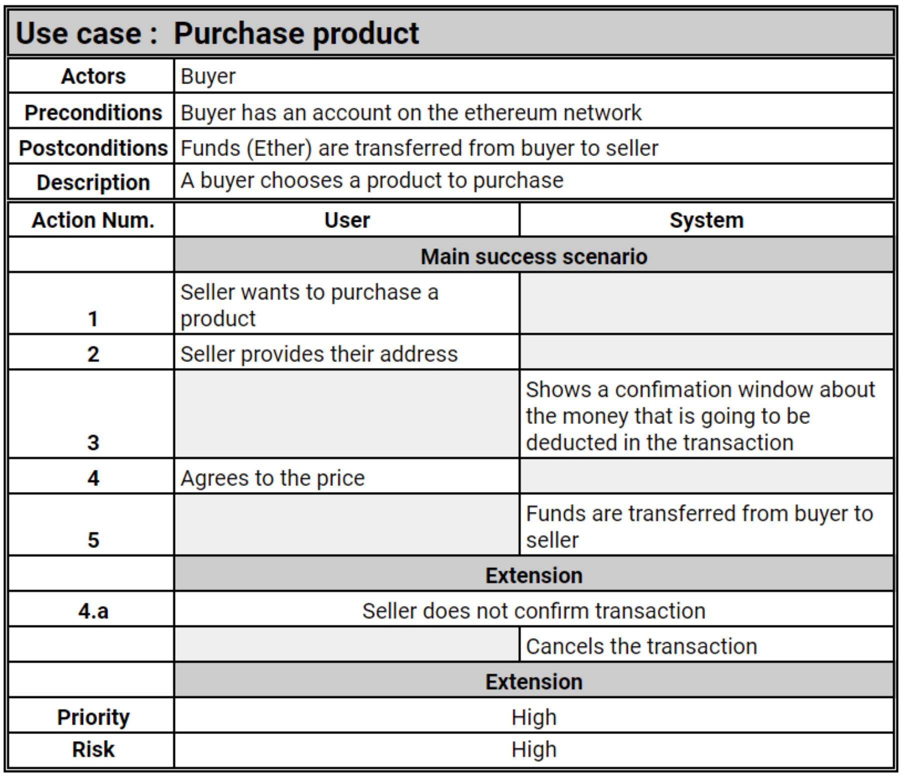
Use cases 5.1 and 5.2 depict the actor, preconditions, postconditions, description, priority, risk, and overall process flow for each requirement.



Use case : List product		
Actors	Seller	
Preconditions	Seller has an account on the ethereum network	
Postconditions	New smart contract deployed in the blockchain	
Description	A seller provides the information and a new contract is created and deployed	
Action Num.	**User**	**System**
	Main success scenario	
1	Seller wants to list a new product	
2	Seller provides their address and the price they want	
3		Creates and deploys the smart contract to the system
4		Confirmation is returned to the Seller
5		Frontend page updated to show new product
	Extension	
2.a	Seller does not provide necessary information	
		Inform that information is wrong or missing
	Extension	
Priority	High	
Risk	High	

Figure 5.1: Use case for listing a product on the blockchain

Use case 5.1 covers the **list product** requirement. The seller is the actor, and they must have an account in order to use the system. The primary postcondition of the requirement is the deployment of a new smart contract based on the seller's information. This process is depicted in the main success scenario, in which users supply their address and the price they desire, and the system uses this information to create and deploy a new smart contract. The system also addresses sellers that provide incorrect or missing information by informing them that they are doing so. The priority is high since it is a backbone requirement of the system, and the risk is high because the technology is new to the developer.



Use case : Purchase product		
Actors	Buyer	
Preconditions	Buyer has an account on the ethereum network	
Postconditions	Funds (Ether) are transferred from buyer to seller	
Description	A buyer chooses a product to purchase	
Action Num.	**User**	**System**
	Main success scenario	
1	Seller wants to purchase a product	
2	Seller provides their address	
3		Shows a confimation window about the money that is going to be deducted in the transaction
4	Agrees to the price	
5		Funds are transferred from buyer to seller
	Extension	
4.a	Seller does not confirm transaction	
		Cancels the transaction
	Extension	
Priority	High	
Risk	High	

Figure 5.2: Use case for purchasing a product on the blockchain

Use case 5.2 follows a pattern similar to use case 5.1. In this situation, the actor is the buyer, and the main success scenario focuses on the activities taken in response to the requirement to purchase products. For the same reasons as in use case 5.1, the risk and priority remain high.

Lastly, because smart contracts are the most complex component in the system, and implemented in a new technology (Solidity), they should be addressed first. Tackling the most complex issues first, provides for greater time for resolving any issues that may arise.

5.2 Design

To illustrate the design of how users interact with smart contracts in the system diagram 5.3 and diagram 5.4 have been designed and described.

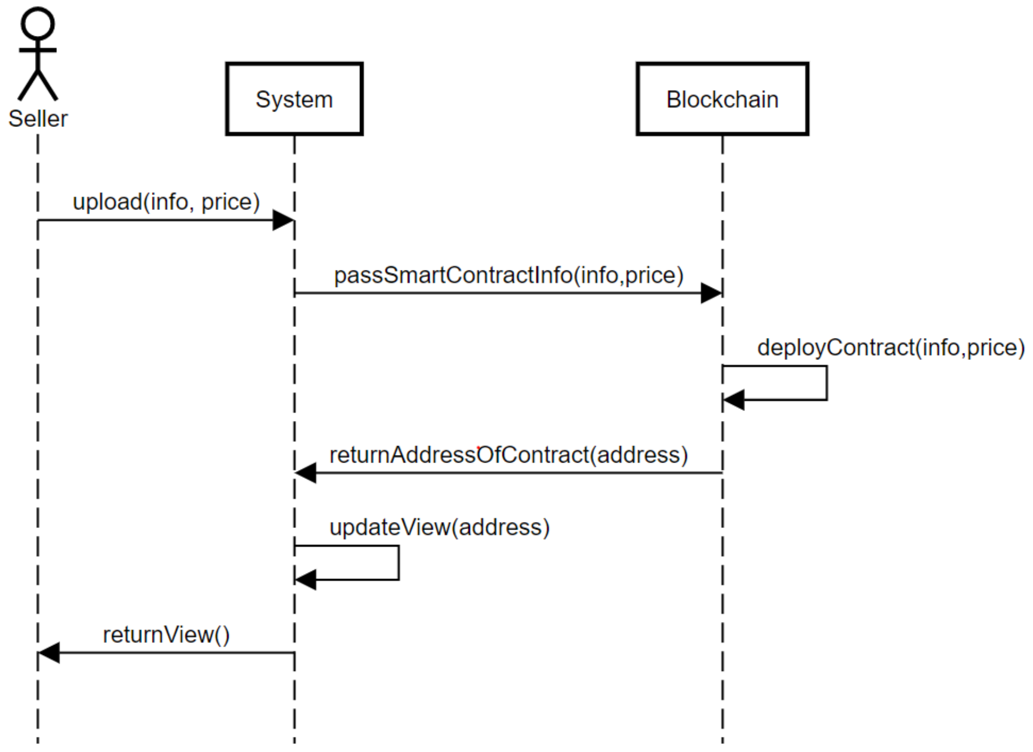

Seller
System
Blockchain
upload(info, price)
passSmartContractInfo(info,price)
deployContract(info,price)
returnAddressOfContract(address)
updateView(address)
returnView()

Figure 5.3: Sequence diagram for listing a product in the blockchain

The user interacts with the system to pass the needed information for the smart contract. This information is address of the creator (owner) and the price they would like to receive. On following sprints, the seller will also upload the data (encrypted) that they want to sell.

This information is passed by the system to the blockchain, which deploys a new contract based on the sellers public address and the price requested. After deployment, an address of the contract is returned and used to update the view for the seller.

When a buyer requests to purchase a product, the following sequence is taken:

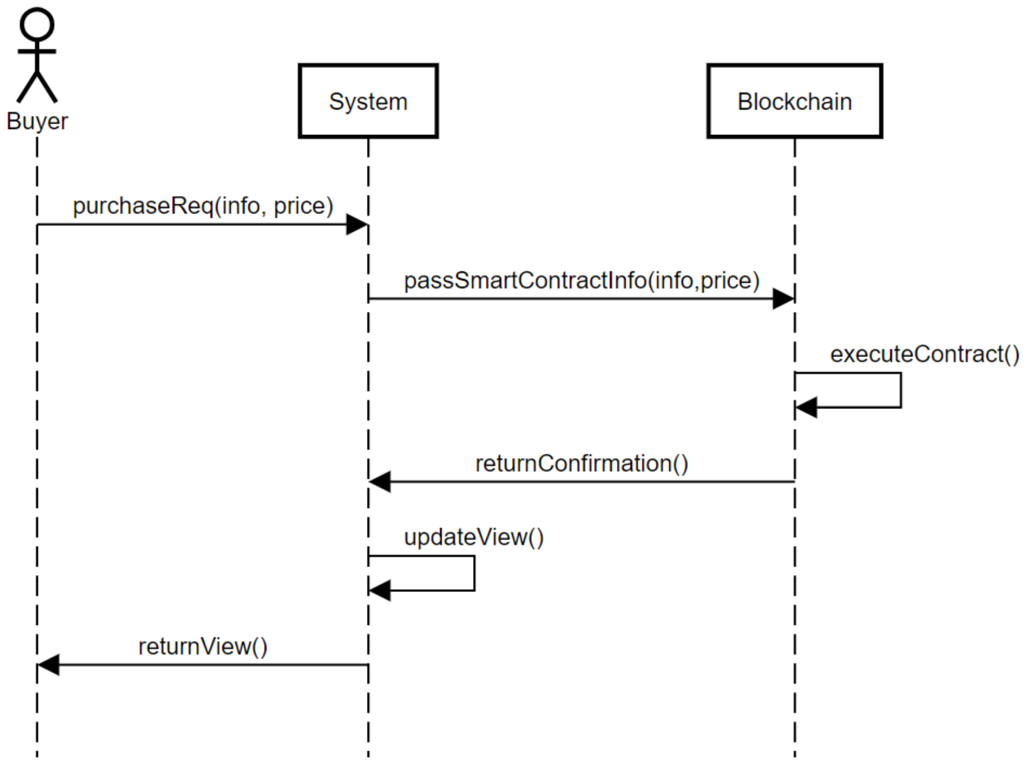


Figure 5.4: Sequence diagram for purchasing

The buyer sends a purchase request alongside with their address to the system. The smart contract then verifies that all terms (described in detail under 5.3.2) have been satisfied and executes the transaction, delivering funds to the seller. Finally, the buyer is shown a view verifying the transaction's success.

5.3 Implementation

This section focuses on the implementation of the smart contracts created during Sprint 1. It begins with an overview of the technologies and tools used to implement, compile, and deploy the smart contract before delving into how the smart contract operates and interacts with the frontend.

5.3.1 Technologies and tools

This section describes the technologies and the rationale for their use, however it does not include the definitive list of technologies, merely those that were significant during the sprint's development.

5.3.1.1 Solidity

Solidity is a high-level, object-oriented programming language facilitating smart contract implementation [61]. These contracts control the behavior of accounts in the Ethereum state.

A curly bracket language, Solidity takes influence from C++, Python, and JavaScript, and it's built for the Ethereum Virtual Machine. Statically typed, Solidity includes features like inheritance, libraries, and elaborate user-defined types [61].

Figure 5.5: Solidity logo

5.3.1.2 Hardhat

Hardhat is an Ethereum development environment that allows users to compile, deploy, test, and debug their application [62]. It allows developers to conveniently organize and automate the recurrent activities that come with designing smart contracts and decentralized applications, as well as adds extra features to this workflow.

Additionally, Hardhat comes with its own local Ethereum development network. This network simulates an Ethereum blockchain for testing purposes, and the transition to the real network can be achieved with ease [62].

Another similar tool which was released prior to Hardhat is the Truffle Suite. However, Hardhat with its extensive documentation seems to be the more popular choice currently.

Figure 5.6: Hardhat logo

5.3.1.3 React

React is a free and open-source front-end JavaScript framework for creating interfaces based on UI components [63]. Because it is component-based, React reduces both time and effort during development. An interface can be broken down into reusable components which can be used to create dynamic user interfaces [63].

Finally, because of its popularity and usage in the industry, React has developed a large and devoted developer community. This community continuously contributes by maintaining and expanding the framework with new features.

Figure 5.7: React logo

5.3.1.4 Metamask

MetaMask is a browser plugin that makes it simpler to access Ethereum's Dapp ecosystem. It also functions as a wallet for cryptocurrency tokens, letting users utilize the wallet to access network-based services [64]. Metamask will be used to store the different addresses of the accounts that want to use the system, as well as their private keys.

Figure 5.8: Metamask logo

5.3.1.5 Ethers.js - Chai.js - MUI

Ethers.js library provides a comprehensive and lightweight library for interfacing with the Ethereum Blockchain and environment. It allows users to interact with a remote or local ethereum node [65]. In the system developed, Ethers connects the React frontend with the local blockchain of Hardhat.

Chai.js is an assertion library for node which provides a variety of assertion result for tests [66]. This library will be useful for running automated testing on the smart contracts developed.

Lastly, MUI is a front end library that offers a rich, flexible, and accessible collection of basic and advanced React components, allowing users to construct React apps and design system faster [67]. This library will be utilized to implement react components using Google's Material Design, thus saving time on CSS code writing.

Figure 5.9: Ethers.js - Chai.js - MUI logos

5.3.2 Smart contract implementation

The following listing shows the implementation of the smart contract under Sprint 1.

```
pragma solidity 0.8.12;
contract Purchase {
    address payable public beneficiary;
4      uint256 public requestedAmount;
5      constructor(uint256 wantedAmount, address payable creator) {
6          beneficiary = creator;
7          requestedAmount = wantedAmount;
8      }
9      modifier SellerCantBuy() {
10         require(
11             msg.sender != beneficiary,
12             "You are the seller you can't purchase your own product"
13         );
14         _;
15     }
16     modifier OnlyOwnerCanSetPrice(){
17         require(
18             msg.sender == beneficiary,
19             "Only the owner can change the price"
20         );
21         _;
22     }
23     function getPrice() public view returns (uint256 price) {
24         return requestedAmount;
25     }
26     function setPrice(uint256 newPrice) public OnlyOwnerCanSetPrice {
27         requestedAmount = newPrice;
28     }
29     function getOwner() public view returns (address owner) {
30         return beneficiary;
31     }
32     function buy() public payable SellerCantBuy {
33         require(requestedAmount == msg.value, "invalid amount");
34         beneficiary.transfer(msg.value);
35     }
36 }
```

Listing 5.1: Smart contract implementation

`1.` The initial line of code in each Solidity file is usually referred to as the pragma. The pragma directive provides the compiler version for the given Solidity file.

`2-8.` The Purchase `contract` (similar to a class in Java) encapsulated the data and the functions of the contract. Because Solidity is a statically typed language, each variable's type (state and local) must be declared. Furthermore, because each data element of a contract is kept on the blockchain which incurs gas fees (described under 2.3.2.1), each data declaration must be efficient. The `address` type contains a 20-byte value (the size of an Ethereum address), and the addition of `payable` indicates that this is an address to which Ether can be transferred. `Uint` type represent an unsigned integer, in this case the price the seller wants for their product. Finally, the `constructor` is a function that is run once when the contract is deployed on the blockchain. It sets the beneficiary to be the creator of the contract, and sets the requested amount based on the parameter given.

`9-22.` Function Modifiers are used to alter a function's behavior. The function body is injected through the placeholder symbol `_;`. Two modifiers can be noticed in this implementation. The first verifies that the individual requesting to purchase the product is not the owner, and the second verifies that only the owner has the authority to change the product's pricing. The term `msg.sender` represents the person (or contract) who is currently connecting and communicating with the contract.

`23-36.` These lines contain the functions of the contract. Functions `getPrice`, `setPrice`, and `getOwner` represent getters and setters to the data elements (it is assumed that the owner of the contract cannot be changed). The `buy` function verifies if the amount matches the requested amount and transfers funds from one account to the other. In the function declaration the `SellerCantBuy` modifier can also be noticed.

5.3.3 Setting up the Ethereum environment using Hardhat

After developing a smart contract, an Ethereum environment where this contract can be compiled and deployed needs to be created. The utilization of Hardhat will be used to create this environment. To install hardhat:

```
npm install|save-dev hardhat
```

To initialize a hardhat project:

```
npx hardhat
```

These commands provide the project setup with the structure:

- `contracts/` this directory contains the source files for the contracts.
- `scripts/` this directory contains the scripts used to deploy the smart contracts to the blockchain.

- `test/` this directory contains tests for the contracts.
- `artifacts/` this directory contains compiled artifacts of the contracts.
- `hardhat.config.js` this file is responsible for configurations of the hardhat project, for example which blockchain to connect to.

5.3.3.1 Compiling the contract and deploying it

After setting up the environment and writing the smart contract, the contract needs to be compiled and deployed. To compile the contracts in a Hardhat project, the compile task is utilized:

```
npx hardhat compile
```

This produces a compiled artifact which contains a lot of useful information related to a contract like the contract bytecode, the application binary interface (ABI), the compiler version, and the deployment details. This artifact will be saved in the artifacts/ directory.

To deploy, a blockchain network is needed. This can be the real Ethereum blockchain network but for testing purposes Hardhat provides a test net which can be run with the command:

```
npx hardhat node
```

Besides running a simulation of the blockchain, the hardhat node also provides the developer with 20 accounts full of ether that can be used to test, deploy and interact with contracts. One of these accounts is represented below:

```
Account #0: 0xf39fd6e51aad88f6f4ce6ab8827279cfffb92266
(10000 ETH)
Private Key: 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed
5efcae784d7bf4f2ff80
```

Once the contract is compiled and there exists a destination blockchain, a script is used to deploy the contract. In subsequent sprints, a factory contract will allow for deployment directly from the frontend, but for Sprint 1 the deployment can only be done through a script. The `deploy.js` script is represented below:

```javascript
1  const hre = require("hardhat");
2  const deploy = require("../lib/deploy");
3  const { ethers } = require("hardhat");
4
5  async function main() {
6    let accounts;
7    accounts = await ethers.getSigners();
8    const purchase = await deploy(
9      ethers.utils.parseEther("10"),
10     accounts[0].address
11   );
12   console.log("Purchase deployed to:", purchase.address);
13 }
14
15 main()
16   .then(() => process.exit(0))
17   .catch((error) => {
18     console.error(error);
19     process.exit(1);
20   });
```

Listing 5.2: Deploy contract script

On line 8, the parameters that were necessary for the deployment of the smart contract can be seen being passed. The owner of this contract will be address 0 (this is an example account out of 20 given by running a local blockchain) and the price they would want is in this example 10 ethers. To run the script:

```
npx hardhat run --network localhost scripts/deploy.js
```

This deploys the contract in localhost (test net), and if later the system is desired to be deployed on the real Ethereum network the localhost has to be exchanged with the appropriate network.

5.3.4 Interacting with the contract

Until now, a smart contract was implemented in solidity, an Ethereum environment was set up, and the contract was compiled and deployed. There is now a requirement to communicate with the deployed contract in a simple and straightforward manner.

5.3.4.1 Setting up Metamask

In order to communicate with the smart contract, an account with an Ethereum address is needed. The private key of the account serves as the password. To set up the account, this system will utilize Metamask.

Once installed, Metamask includes an import account feature that asks for a private key to an account. In this case, the private key can be any of the 20 accounts issued to my Hardhat when the local blockchain was set up. If deployed in the real network, users would utilize their genuine Ethereum main net accounts.

5.3.4.2 The React frontend

In order to connect the accounts being stored in Metamask to the deployed contract a frontend using React and the Ethers.js library was implemented.

```
function App() {
  const [price, setPrice] = useState();
  const [priceToChange, setPriceToChange] = useState();
  const [owner, setOwner] = useState();
  async function requestAccount() {
    await window.ethereum.request({ method: "eth_requestAccounts" });
  }
  useEffect(() => {
    const fetchData = async function fetchContractDetails() {
      if (typeof window.ethereum !== "undefined") {
        const provider = new ethers.providers.Web3Provider(window.
ethereum);
        const contract = new ethers.Contract(
          purchaseAddress,
          Purchase.abi,
          provider
        );
        try {
          const data = await ethers.utils.formatEther(
            await contract.getPrice()
          );
          setPrice(data);
          const owner = await contract.getOwner();
          setOwner(owner);
          console.log("price ", data);
          console.log("owner ", owner);
        } catch (err) {
          console.log("error ", err);
        }
      }
    };
    fetchData();
  }, []);
  async function setPriceOnContract() {
    if (!priceToChange) return;
    if (typeof window.ethereum !== "undefined") {
      await requestAccount();
      const provider = new ethers.providers.Web3Provider(window.
    ethereum);
      const signer = provider.getSigner();
      const contract = new ethers.Contract(
        purchaseAddress,
        Purchase.abi,
        signer
      );
      try {
        const transaction = await contract.setPrice(
          ethers.utils.parseEther(priceToChange)
        );
        await transaction.wait();
        setPrice(priceToChange);
      } catch (error) {
        alert("Only the owner can change the price");
        console.log(error);
      }
    }
  }
  async function purchaseProduct() {
    if (typeof window.ethereum !== "undefined") {
      await requestAccount();
      const provider = new ethers.providers.Web3Provider(window.
    ethereum);
      const signer = provider.getSigner();
      const contract = new ethers.Contract(
        purchaseAddress,
        Purchase.abi,
        signer
      );
      try {
        const transaction = await contract.buy({
          value: ethers.utils.parseEther(price),
69         });
70         await transaction.wait();
71       } catch (error) {
72         alert("You are the owner you cannot buy your product");
73         console.log(error);
74       }
75     }
76   }
77 }
```

Listing 5.3: Frontend functionalities implementation

`2-4.` The state variables are declared using the React hook `useState()`. These variables are expected to be updated based on the price and owner of the contract.

`5-7.` The request account function connects to the Metamask wallet of the user whenever a transaction is required to happen.

`8-32.` These lines create a `useEffect` hook that wraps the function fetching the data. The react hook runs the function first when loading the component and also each time the dependencies to the hook change. This feature is utilized to fetch the data of the smart contract when the frontend initially loads. The `fetchData()` function first checks for Metamask to inject an object to the window (line `10`). This injection will be automatic if the user has a Metamask wallet.

After checking for the ethereum window, instances of a provider and a contract are initialized. A Provider is an abstraction of an Ethereum network connection that provides a clear, consistent interface to typical Ethereum node functions. Once the provider is initialized, an instance of the contract can be created. The parameters the contract takes are the address of the contract, the contract's ABI, and the provider that was initialized. Using the contract, the function reads the data from the blockchain to get the price (lines `18-20.`) and the owner (line `22.`) and sets the state variables for each information. Any error is caught and logged.

`33-55.` This function is responsible for changing the price on the contract. It follows a similar approach to the previous function, first checking for an injected Ethereum window, then initializing the provider, the contract, and the signer (user), and finally running the function `setPrice()` with the catch ready to catch any errors thrown.

`56-74.` The final function is responsible for purchasing and transferring the funds from one user to another. Similarly, a provider, a signer, and the contract are needed for the function to be called. Once called the parameter is given to the function as `value: ethers.ParseEther(price)`. This ensures that the input of the user reflects as Ethers and not Wei.

5.4 Testing

Testing is essential to the success of every project, regardless of the development method or programming language used. It is critical to do tests at the conclusion of each sprint to verify that everything that was implemented is operating properly before moving on to the next sprint. This sprint's primary testing approach will be functional and black-box testing, in which the developer provides input to the system and expects the appropriate output without independently studying the inner workings of the functions.

After testing that the logic of the contract developed is correct, some end-to-end manual testing was also performed to ensure that application's end-to-end workflow behaves as expected.

5.4.1 Testing the smart contract with Chai.js

When the Hardhat project was initialized, a separate test directory was also set up. This is where the automated tests will be written. These tests aim to investigate all situations and interactions with the smart contract. To write the tests, the library Chai.js was utilized.

```
it("Should not be able to buy own product", async function () {
    const purchase = await deploy(
      ethers.utils.parseEther("10"),
      accounts[0].address
    );

    await expect(purchase.buy()).to.be.revertedWith(
      "You are the seller you can't purchase your own product"
    );
  });
```

Listing 5.4: Testing that sellers are not allowed to buy their own product

Listing 5.4 shows the automated test for the situation where owners of the contracts try to purchase their own product. In Chai.js, `it` represents a test case, and the description of the case follows each test case (line `1.`). In the test function, an instance of the purchase smart contract is deployed (lines `2-5.`). The Ethers.js function to `parseEther()` converts the input users give from Wei into Ether. After deploying, the function `buy()` is called, and since it is called from the account that deployed the contract (owner) it is expected to be reverted with the message "*You are the seller.*."

```
it("Should not be able to buy without right funds", async function () {
    const purchase = await deploy(
      ethers.utils.parseEther("10"),
      accounts[0].address
    );
    const buyer = purchase.connect(accounts[1]);
    await expect(buyer.buy()).to.be.revertedWith("invalid amount");
  });
```

Listing 5.5: Testing that buyers can not buy with insufficient funds

Listing 5.5 tests for the case where a user tries to purchase a product with insufficient funds. Similarly to listing 5.4, the purchase contract is first deployed with adequate parameters. After deployment, another account is used to represent a buyer and to distinguish from the owner. This buyer tries to call the function `buy()` of the smart contract, but does not provide the required ethers (10), therefore the contract is expected to revert with "*Invalid amount*".

```
it("Should not be able to change price if not owner", async function(){
    const purchase = await deploy(
      ethers.utils.parseEther("10"),
      accounts[0].address
    )
    const buyer = purchase.connect(accounts[1]);
    await expect(buyer.setPrice(5)).to.be.revertedWith("Only the owner can change the price")
  });
```

Listing 5.6: Testing that buyers can not change the price of the product

Another feature of the smart contract is the ability to change the price requested by the owner. The test case in listing 5.6 depicts a situation where another user and not the owner tries to change the price of the contract. This test case follows the logic of test cases under listing 5.4 and listing 5.5.

```
it("Should be able to change price if owner", async function(){
    const purchase = await deploy(
      ethers.utils.parseEther("10"),
      accounts[0].address
    )
    const owner = purchase.connect(accounts[0]);
    await owner.setPrice(ethers.utils.parseEther("11"));
    await expect(await ethers.utils.formatEther(await owner.getPrice())
    ).to.be.equal("11.0")
  });
```

Listing 5.7: Testing for owners ability to change the price in the contract

In listing 5.6 it was tested for the situation where the contract should prevent an unauthorized user trying to change the price. To make the testing complete for the function, the scenario where the user is authorized (owner) to change the price is seen under listing 5.7. Because this is a case where the contract should execute, the function is not expected to be reverted. To check that the price was changed the contract function `setPrice()` is called and awaited to be executed. Once executed the test checks that `getPrice()` matches the new price and not the originally set price, in this case '10' and '11' respectively. One thing to notice is that the `getPrice()` function will return the price in Wei and this needs to be converted to Ether before comparing.

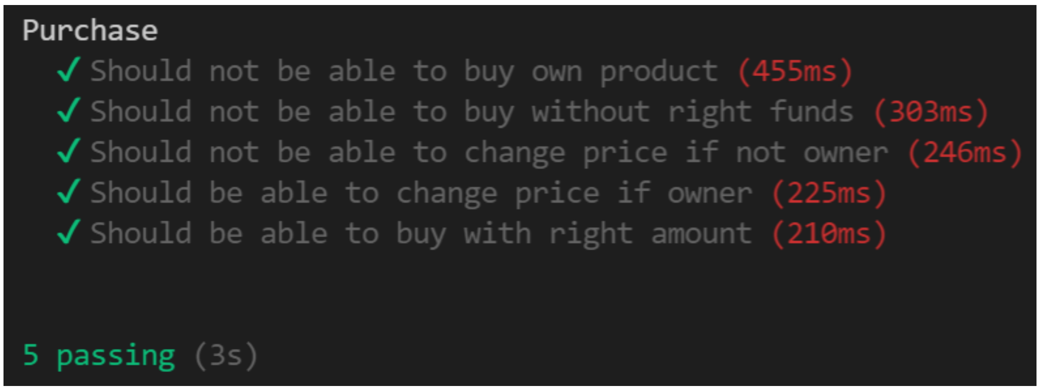
```
it("Should be able to buy with right amount", async function () {
    const price = ethers.utils.parseEther("10");
    const purchase = await deploy(price, accounts[0].address);
    const buyer = purchase.connect(accounts[1]);
    const buyerInitialBalance = await ethers.provider.getBalance(
      accounts[1].address
    );
    const sellerInitialBalance = await ethers.provider.getBalance(
      accounts[0].address
    );
    const transaction = await buyer.buy({
      value: price,
    });
    const receipt = await transaction.wait();
    const gasUsed = receipt.cumulativeGasUsed.mul(receipt.
    effectiveGasPrice);

    await expect(
      await ethers.provider.getBalance(accounts[1].address)
    ).to.be.equal(buyerInitialBalance.sub(price).sub(gasUsed));

21     await expect(
22       await ethers.provider.getBalance(accounts[0].address)
23     ).to.be.equal(sellerInitialBalance.add(price));
24   });
```

Listing 5.8: Testing that buyers can buy with sufficient funds

The final test case in listing 5.8 tests for the situation where buyers complete the transaction and purchase the product. Lines `2-4.` initialize the variables of the transaction similarly to previous test cases. Lines `2-10.` initialize the initial balances of the seller and the buyer using the Ethers.js function `getBalance()`. Lines `11-13.` initialize a transaction in which the buyer calls the `buy()` function and passes the correct parameters. Lines `14-15.` are responsible for initializing a receipt of the transaction and calculate the gas used from the transaction. This is important because every transaction of funds uses additional gas that needs to be taken into calculation. Finally lines `17-24.` is what the test case expects to find. The first expectation is that the balance of the buyer is reduced by the price of the product and the price of the gas used for the transaction. The second expectation checks that the initial balance of the seller has increased by the amount of the product.

5.4.2 Automated testing results

To run the test file with all the test cases, the command `npx hardhat test` is run. This command executes the test file and produces the following outcome in the terminal:



```
Purchase
  ✓ Should not be able to buy own product (455ms)
  ✓ Should not be able to buy without right funds (303ms)
  ✓ Should not be able to change price if not owner (246ms)
  ✓ Should be able to change price if owner (225ms)
  ✓ Should be able to buy with right amount (210ms)


5 passing (3s)
```

Figure 5.10: Testing outcome

Figure 5.10 shows that each test was successful (passed) and that the implementation of the smart contract was correct.

5.4.3 Testing the frontend

Testing each frontend functionality and their respective interactions with the smart contract was achieved through manual testing. The react server was run and the contract was displayed, then each functionality was tested and the outcome was the expected one.

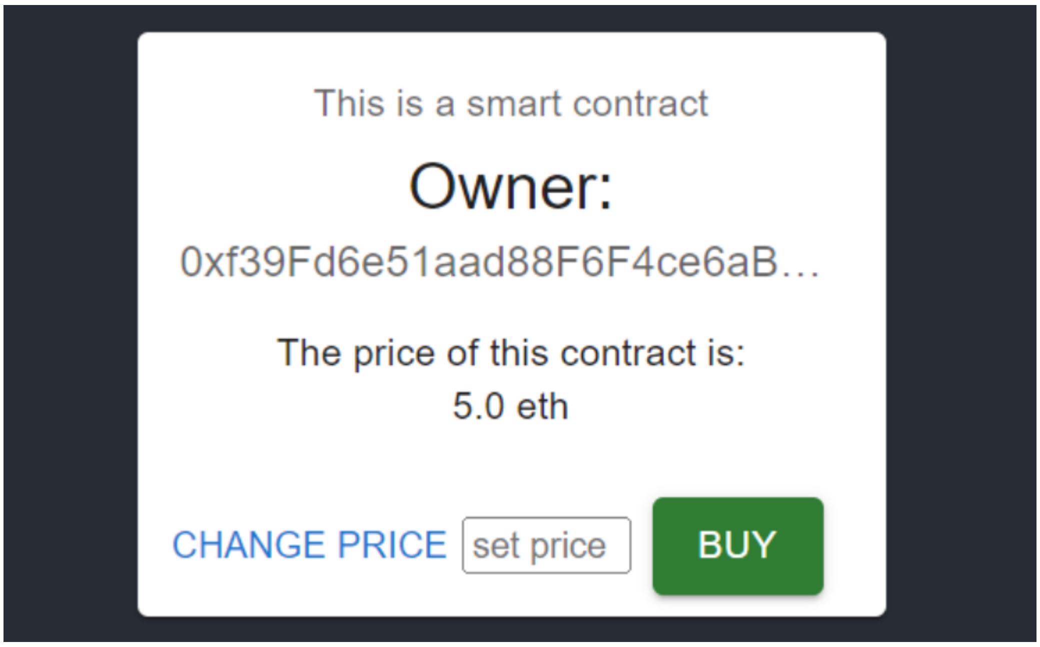


Figure 5.11: Frontend testing

Figure 5.11 shows the frontend view of the contract. The owner is the account that deployed the contract, the price reflects the price given in eth.

To test the change price functionality, the function was tested with both the owner account and another account. Only the owner could perform the operation.

Similarly, the buy function was tested with both accounts and only account 2 (not the owner) could transfer the funds to the creator of the account. Metamask provided the functionality to switch between and also view the balance in each account.

5.5 Sprint 1 evaluation

At a high level, Sprint 1 was a success despite the fact that not many requirements were addressed. Each technology and concept utilized to develop the system was new at the start of the sprint, and this sprint provided as a foundation for learning and applying each new technology and concept. The foundation was carefully designed and thoroughly tested to ensure that it is implemented correctly and solidly. This foundation facilitates the opportunity to expand the system and project on further sprints.

On a more detailed level, the basis of the smart contract responsible for the operation of the system was implemented. This contract should be expanded to contain more information (hashes of items in IPFS) and new functionalities (choosing a random sample), but the overall principle should remain the same throughout subsequent sprints. Furthermore, the smart contract was compiled and deployed in a test blockchain, and a frontend was developed to interface with each contract functionality. This end-to-end capability resulted in the first functioning prototype of a system that utilizes smart contracts and blockchain technology to perform secure transactions.

Chapter 6

Sprint 2

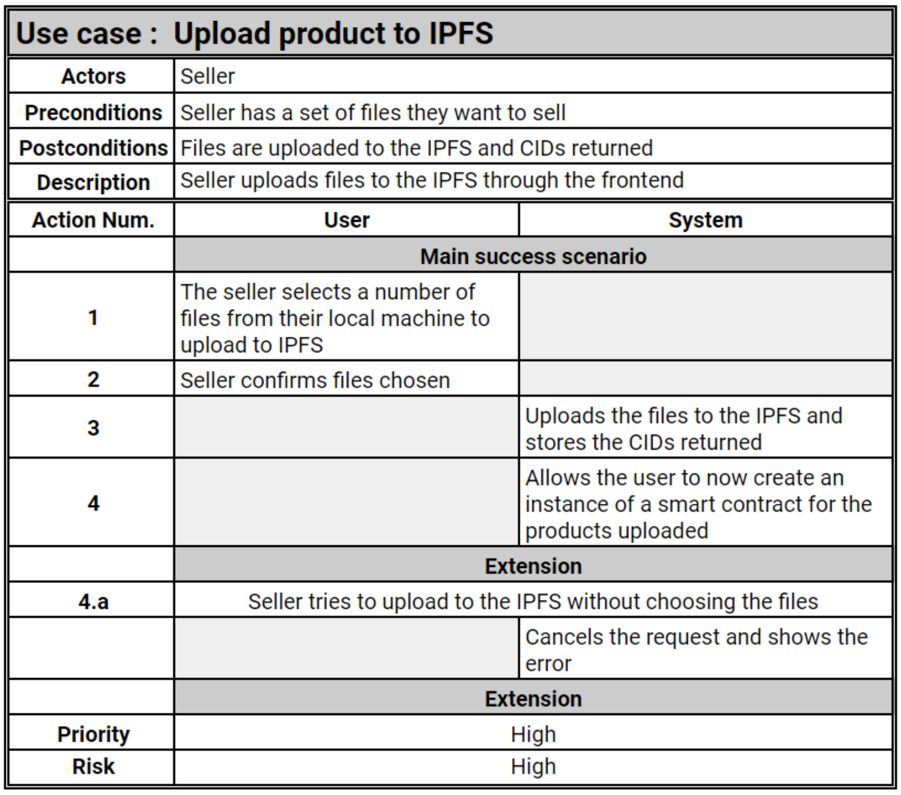
Sprint 2 will focus on extending and developing on the foundation established in Sprint 1. The implemented prototype smart contract must be extended to provide more functionality related to additional requirements. In the following sprint sections, these new features and the requirements they relate to will be analyzed, designed, implemented, and tested.

6.1 Analysis

The underlying smart contract was developed during Sprint 1, however it had to be deployed via a script, and the user was limited to only one contract. The objective of this sprint is to enable system users to create multiple instances of contracts and deploy them directly from the React frontend. In addition, the requirement for users to be able to upload their data (microstock) to the IPFS and connect this data to the smart contract will be addressed.

These requirements expand on the established smart contract, enhancing and adding functionality to the system and bringing it closer to the intended system.

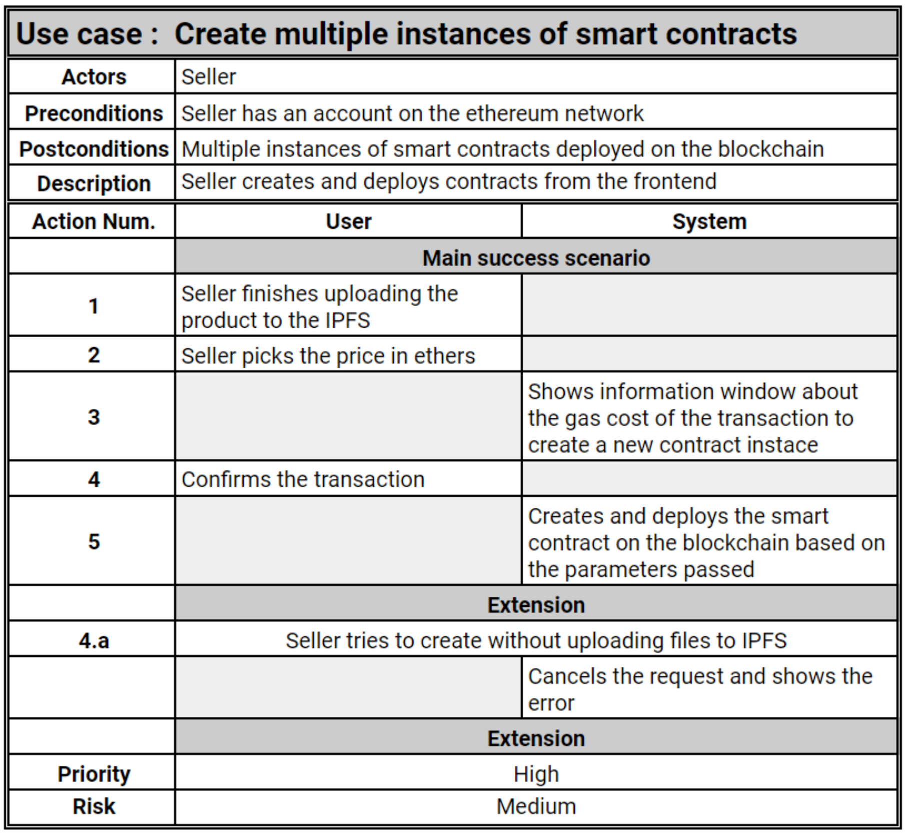
For a visualisation of the requirements chosen for Sprint 2, two use cases have been created. The use case for uploading products to the IPFS is depicted in Figure 6.1, and the use case for creating multiple instances of smart contracts is represented in Figure 6.2.



Use case : Upload product to IPFS		
Actors	Seller	
Preconditions	Seller has a set of files they want to sell	
Postconditions	Files are uploaded to the IPFS and CIDs returned	
Description	Seller uploads files to the IPFS through the frontend	
Action Num.	**User**	**System**
	Main success scenario	
1	The seller selects a number of files from their local machine to upload to IPFS	
2	Seller confirms files chosen	
3		Uploads the files to the IPFS and stores the CIDs returned
4		Allows the user to now create an instance of a smart contract for the products uploaded
	Extension	
4.a	Seller tries to upload to the IPFS without choosing the files	
		Cancels the request and shows the error
	Extension	
Priority	High	
Risk	High	

Figure 6.1: Use case for uploading products on the IPFS

In use case 6.1, the actor is the seller, and they must have a set of files they would like to put up for sale. The key postcondition of the requirement is that each file be uploaded to the IPFS network. In the main success scenario, sellers select and confirm the files they wish to upload to IPFS, and the system uploads those files and returns the CIDs (explained further in section 6.3.1.1). With the CIDs obtained, users are permitted to create a smart contract instance around the files uploaded. If a user attempts to upload to the IPFS network without first selecting any files, the system will deny the upload and notify the user. Both the priority and the risk remain high since the requirement is vital to the system, and the IPFS technology for implementation is unfamiliar.



Use case : Create multiple instances of smart contracts		
Actors	Seller	
Preconditions	Seller has an account on the ethereum network	
Postconditions	Multiple instances of smart contracts deployed on the blockchain	
Description	Seller creates and deploys contracts from the frontend	
Action Num.	**User**	**System**
	Main success scenario	
1	Seller finishes uploading the product to the IPFS	
2	Seller picks the price in ethers	
3		Shows information window about the gas cost of the transaction to create a new contract instace
4	Confirms the transaction	
5		Creates and deploys the smart contract on the blockchain based on the parameters passed
	Extension	
4.a	Seller tries to create without uploading files to IPFS	
		Cancels the request and shows the error
	Extension	
Priority	High	
Risk	Medium	

Figure 6.2: Use case for listing a product on the blockchain

Use case 6.2 follows a similar approach to use case 6.1. In this scenario, after obtaining the CIDs, the seller creates multiple instances of smart contracts. This depicts the primary success scenario and users are informed about the amount of gas the transaction will consume. Lastly, the system prevents any user from creating a contract instance without having uploaded any files to the IPFS.

6.2 Design

The description of the design of the chosen requirements begins with the factory pattern design and is followed by the IPFS design.

6.2.1 Factory pattern design

A factory pattern design was chosen to allow users to create and deploy multiple instances of contracts. This programming pattern is based on the notion of one object (the factory) creating instances of other objects. Because smart contracts are objects, this paradigm can be utilized in the system being developed [68]. Furthermore, the factory model is advantageous when it is necessary to quickly construct several instances of a smart contract at runtime, as well as when dealing with many contracts that all accomplish the same functionality. Both points are relevant to the system under development since users will deploy from the frontend, and the instances that are deployed have the same functionalities. Figure 6.3 depicts the factory pattern where the source code of the factory is deployed once and then utilized to create instances of other contracts.

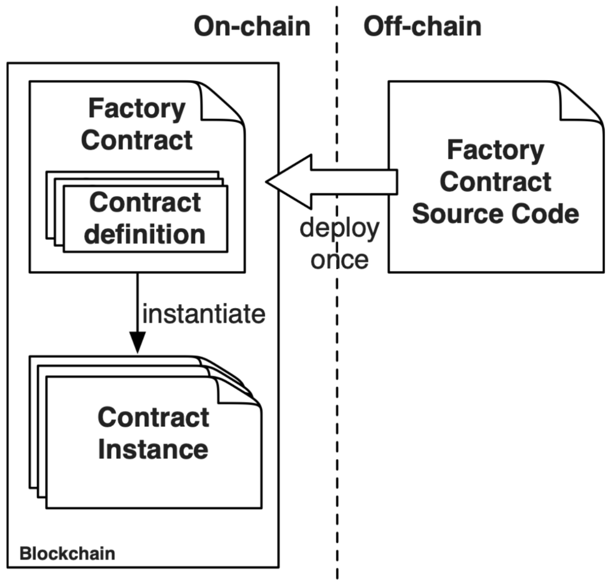


Figure 6.3: Factory contract pattern

Additionally, the benefits of using the factory pattern in Solidity include:

- Ease of track for all deployed contracts
- High gas-efficiency for multiple contract deployment
- Simple manner for user to create and deploy contracts

As a result of all the benefits mentioned, the system will utilize a single factory contract in charge of creating and deploying multiple instances of the same contract, storing these instances on the blockchain, and retrieving them as needed.

6.2.2 IPFS design

To illustrate how the user interacts with the IPFS (further explained under 6.3.1) sequence diagram 6.4 was designed and described.

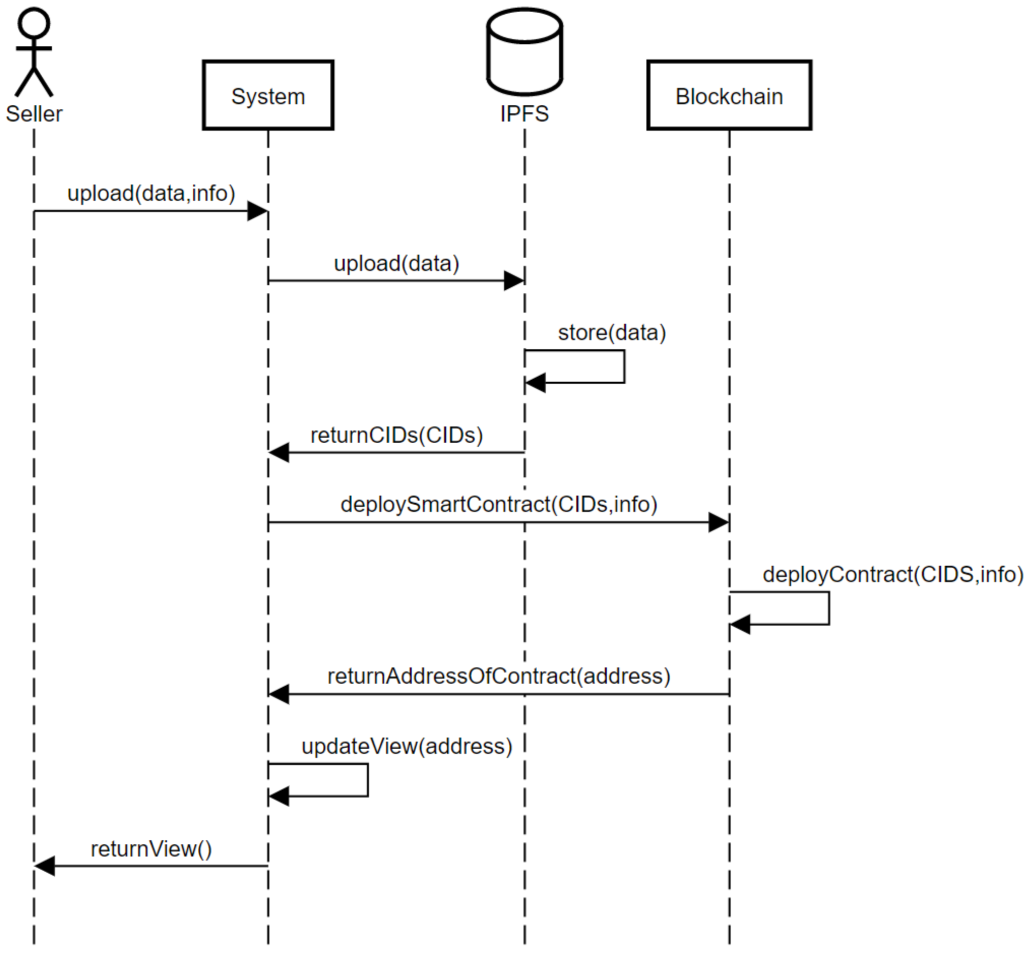


Figure 6.4: Sequence diagram for user interaction with IPFS

Correspondingly to Sprint 1, the seller interacts with the system by providing the amount of Ethers for which they are selling their product (data) as well as their public address (info). In this sprint, alongside the requested amount the users will upload the microstock (pictures) as well.

After user provide their product to the frontend, the system uploads it to the IPFS, where it will be stored. The IPFS returns a content identifier (CID), which is a label pointing to the contents. The system then passes the CIDs, the requested amount, and the seller's public address to the blockchain's deployed factory contract. Based on the information supplied, the factory creates and deploys a new smart contract instance.

Finally, the factory contract returns the address of the deployed contract to the frontend, which is then utilized to update the view displaying the details of the deployed smart contract.

6.3 Implementation

This section provides an overview of the technologies and tools used throughout this sprint as well as a detailed description of the factory contract implementation and IPFS integration within the project.

6.3.1 Technologies and tools

The list of technologies and tools utilized must be expanded in order to execute the current sprint requirements. This list will now include IPFS and js.ipfs.io.

6.3.1.1 IPFS

The InterPlanetary File System (IPFS) is a protocol and peer-to-peer network that allows users to store and exchange data in a distributed manner.

Figure 6.5: IPFS logo

1 - Once the users upload a file to IPFS, it is divided into smaller bits, hashed cryptographically, and assigned a unique fingerprint known as a content identifier (CID). The fragmenting of the product to be sold is an important part of the security protocol (described in the next sprint under section 7.1.1). The assigned CID serves as a permanent record of the data as it existed at the time.

2 - When other nodes search for the file, they query their peer nodes in order to find in which node the material addressed by the file's CID is stored. Once

a node views or downloads the data, they cache a replica and become another distributor of the material until their cache is emptied.

3 - Any node can pin material to store (and distribute) it indefinitely, or it can remove data it hasn't accessed in a while to conserve space. This implies that every node in the network maintains just the information that it is concerned with, as well as some indexing metadata that aids in determining which node is holding what.

4 - When users add a new variant of their content to IPFS, its cryptographic hash changes, and it receives a new CID. This implies that IPFS files are robust to manipulation and censorship, as any modification does not erase the original.

Figure 6.6: Illustration for the IPFS upload steps

6.3.1.2 JS IPFS

JS-IPFS provides a gateway for the IPFS protocol to be implemented in browsers. It is developed purely in JavaScript by the developers behind IPFS and operates in a Browser, a Service Worker, a Web Extension, and Node.js. Although the API is still in its early stages, it was chosen over the alternative Infura since Infura is not completely free and charges for the API it provides to the IPFS protocol. Furthermore, having a centralized provider contradicts the concept of a truly decentralized application.

JS-IPFS is what the system will utilize in order to upload the data from the React frontend to the IPFS nodes.

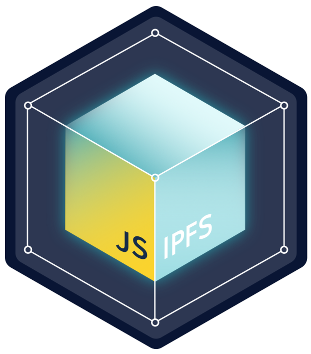


Figure 6.7: JS IPFS logo

6.3.2 Factory contract implementation

Listing 6.1 represents the factory contract that will be deployed once (script 5.2 under Sprint 1) and will be responsible for creating instances of other contracts.

```
pragma solidity 0.8.12;
import "./Purchase.sol";
contract ContractFactory {
    Purchase[] private _purchases;
    event contractCreated(
        address contractAddress,
        uint256 wantedAmount,
        address owner,
        string[] ipfsCIDs
    );
    function createPurchase(uint256 wantedAmount, string[] memory
    ipfsCIDs) public
    {
        Purchase purchase = new Purchase(
            wantedAmount,
            payable(msg.sender),
            ipfsCIDs
        );
        _purchases.push(purchase);
        emit contractCreated(
            address(purchase),
            wantedAmount,
            msg.sender,
            ipfsCIDs
        );
    }
26     function getContracts() public view returns (Purchase[] memory) {
27         return _purchases;
28     }
29 }
```

Listing 6.1: Factory contract implementation

`1-3`. The solidity version is declared, the purchase contract (listing 5.3.2 is imported and the Contract ContractFactory is created.

`4`. Array declaration for storing all instances of the Purchase contracts. The type of the elements is specified and the array has a dynamic size since the number of contracts is not known beforehand.

`5-10`. An event is a contract component that can be inherited. An event is emitted, and the arguments provided are recorded in transaction logs. These logs are preserved on the blockchain and may be accessed using the contract's address as long as the contract is available on the blockchain. Here the created contract's address, owner, amount, and ipfsCIDs are emitted.

`11-12`. The signature of the main function responsible for creating instances of the purchase contract. The function expects a `uint` for the wanted amount and an array of strings for the ipfsCIDs. `Memory` instructs solidity to allocate a block of space for the variable during method execution, ensuring its size and structure for future usage in that method. Lastly, the function has the `public` keyword, meaning it is accessible outside of the contract.

`13-25`. A new instance of the purchase contract is initialized with the wanted amount, `msg.sender` which represents the caller of the contract's public address, and the array of IPFS CIDs. This contract is pushed to the array of purchase contracts, and an event is emitted showing all the information of the contract.

`26-28`. Function for fetching the array of contracts that are currently deployed. This function will be useful to display on the frontend each contract instance that is deployed on the blockchain.

6.3.3 Additions to the purchase contract

The purchase contract developed under Sprint 1 needs to be expanded to include the IPFS CIDs. This development can be seen under listing 6.2

```
contract Purchase {
              ...
    string[] public ipfsCIDs;
4
5       constructor(uint256 wantedAmount, address payable creator, string[] memory cids) {
6           beneficiary = creator;
7           requestedAmount = wantedAmount;
8           ipfsCIDs = cids;
9       }
10              ...
11
12      function getCIDs() public view returns (string[] memory) {
13          return ipfsCIDs;
14      }
15  }
```

Listing 6.2: Purchase contract additions

`3.` Array declaration for a dynamic array responsible for holding the IPFS CIDs. It is important to store the CIDs on the blockchain for security and transparency.

`5-9.` Constructor is altered from Sprint 1 to now accept the parameter of the CIDs array. Sets `ipfsCIDs` declared in line `3` to the given array in the parameter.

`12-14.` Function responsible for returning the array of IPFS CIDs.

6.3.4 Frontend IPFS connection and contract deployment

Users will be presented with a two-part form to upload products to the IPFS and to interact with the factory contract in order to deploy smart contract instances. Listing 6.3 shows the implementation of how users connect to the IPFS to upload their products from the frontend.

```
export default function AddProduct() {
        ...
    const [images, setImages] = React.useState({
        files: [],
    });
    const [ipfsCIDs, setIpfsCIDs] = React.useState([]);

    const retrieveFile = (e) => {
        setImages({ files: [...images.files, ...e.target.files] });
    };
    const handleSubmit2 = (e) => {
        e.preventDefault();
        let tempArray = [];
14          IPFS.create().then((ipfs) => {
15              images.files.forEach((e) => {
16                  ipfs.add(e).then(({ cid }) => {
17                      tempArray.push(cid.toString());
18                      setDisabledButton(false);
19                  });
20              });
21              setIpfsCIDs(tempArray);
22          });
23      };
24          ...
25  }
```

Listing 6.3: IPFS Upload implementation

`1.` Defining the component responsible for getting the products, uploading them to the IPFS, and creating the smart contract.

`3-6.` Two react state components are declared. The former will be updated to hold the images that have been uploaded, while the latter will store the CIDs of the files stored on the IPFS.

`8-10.` Once the user has selected all of the files to upload, this function is invoked. It is in charge of collecting and storing every file. Using the spread operator (**...**), the objects uploaded are destructed, and the state variable stated in line `3.` is set to hold the files.

`11-23.` When the user has decided which files to upload, they will click the submit button. `e.preventDefault()` prevents the event's default action from taking place. In this case, the submit function will be prevented from submitting a form and refreshing the page. This is required because the CIDs of the files must be supplied to the second part of the form that is in charge of creating the smart contract. `IPFS.create()` is an asynchronous function that initializes an IPFS client. Because it communicates with the IPFS network, it is asynchronous and must be resolved with `.then()`. After the client is initialized, `ipfs.add()` uploads each element to IPFS and returns their CID. This CID is converted to a string and added to a temporary array that contains all of the CIDs, one for each file. Lastly, the state variable is set to the array of CIDs obtained by the function.

Once the user's selected files have been uploaded to IPFS and the CIDs have been obtained, the user is permitted to create a new instance of a smart contract. This implementation is depicted in listing 6.4, showing the interaction user $\rightarrow$ contract factory.

```
1  export default function AddProduct() {
2          ...
3      const handleSubmit = async () => {
4          if (!priceOfNew) return;
5          if (typeof window.ethereum !== "undefined") {
6              await requestAccount();
7              const provider = new ethers.providers.Web3Provider(window.
     ethereum);
8              const signer = provider.getSigner();
9              const contract = new ethers.Contract(
10             factoryAddress,
11             ContractFactory.abi,
12             signer
13             );
14             try {
15                 const transaction = await contract.createPurchase(
16                   ethers.utils.parseEther(priceOfNew),
17                   ipfsCIDs
18                 );
19                 await transaction.wait();
20             } catch (error) {
21                 alert("There was an error");
22                 console.log(error);
23             }
24         }
25         setOpen(false);
26     };
27         ...
28 }
```

Listing 6.4: Creating and deploying new contract instances through the factory

`2.` Utilizing the contract factory (listing 6.1) this function is responsible for contract creation. The function is asynchronous since it communicates with the blockchain and awaits responses.

`4.` If the user has not chosen a price for the product the function exists, preventing a request for a contract with no set price.

`5-8.` A provider and a signer are initialized, as explained in the listing 5.3.

`9-13.` An instance of the factory contract is initialized using the factory's address, ABI, and the signer.

`14-23.` With the factory initialized, a transaction to create a new smart contract is made. The factory function `createPurchase()` creates and deploys the new contract taking as parameters the price for the products and the ipfs-CIDs obtained in listing 6.3. The user who invoked the function is automatically designated as the owner.

6.4 Testing

Similarly to Sprint 1, testing will be divided into two phases. The first phase is the automated testing which is performed using Chai.js and tests the factory contract implementation, and the second phase is manual testing which is performed to test the frontend.

6.4.1 Factory Contract testing

Listing 6.5 depicts the automated tests written in order to check if the implementation of the factory contract was correct.

```
it("Should deploy contract correctly", async function () {
    const Factory = await ethers.getContractFactory("ContractFactory");
    const contractFactory = await Factory.deploy();
    const testCIDs = ["abc", "def"];

    await contractFactory
      .connect(accounts[0])
      .createPurchase(ethers.utils.parseEther("10"), testCIDs);
    await contractFactory
      .connect(accounts[1])
      .createPurchase(ethers.utils.parseEther("11"), testCIDs);

    const returnedAddresses = await contractFactory.getContracts();
    const Purchase = await ethers.getContractFactory("Purchase");
    let purchase1 = Purchase.attach(returnedAddresses[0]);
    let purchase2 = Purchase.attach(returnedAddresses[1]);

    await expect(await purchase.getOwner()).to.be.equal(accounts[0].
    address);

    await expect(
      await ethers.utils.formatEther(await purchase.getPrice())
    ).to.be.equal("10.0");

    await expect(await purchase2.getOwner()).to.be.equal(accounts[1].
    address);
25
26      await expect(
27        await ethers.utils.formatEther(await purchase2.getPrice())
28      ).to.be.equal("11.0");
29    });
```

Listing 6.5: Automated testing for the implementation of the factory contract

`1-4.` Declaring the test name and setting up the needed constants necessary for the testing. These constants include the factory which is initialized in line `2` and deployed in line `3`. In line `4` because these tests do not examine the IPFS upload and CID retrieval procedure, the CIDs are hard coded.

`6-11.` The factory contract generates and deploys two purchase contracts. The first contract is owned by the first test account, has a price of 10 ethers, and holds the test CIDs specified in line `4`, whereas the second contract is owned by the second account, has a price of 11 ethers, and holds the the same CIDs for convenience.

`13-16.` With the contracts deployed, the factory function `getContract()` is called to fetch the addresses of each contract. These addresses are attached to purchase contract instances declared in line `14`. The two deployed contracts are given the names `purchase1` and `purchase2`.

`18.` The first anticipated outcome of the test. The owner `purchase1` should be equal to the account address used to deploy the contract (the owner).

`20-22.` The test's second expected outcome. The price of `purchase1` should be 10 ethers. The returned price from the contract is converted to ethers since the amount is stored in wei.

`24-28.` These lines follow the example of the previous two anticipated outcomes but now verify that the second contract `purchase2` has been deployed successfully with adequate parameters.

6.4.2 Automated testing results

The testing was carried out in the same approach as in Sprint 1 (see 5.4.2), using the hardhat command: `npx hardhat test`. The results revealed that all tests were passed and all expectations were reached.

6.4.3 Testing the frontend

Since the functionality for contract deployment and IPFS data upload was provided to users directly on the frontend, it was imperative that each functionality be well tested.

This testing was conducted manually. Different accounts were used to deploy contracts with various parameters and files. These contracts were then correctly displayed on the React frontend and the uploaded files could be found in the IPFS by accessing `ipfs.io/ipfs/(CID).`

The system prevented the user from submitting a request to create a contract that has no price assigned. Furthermore, the system stops the user from creating an instance of a contract that holds no files. Finally, before deployment, all of the information is displayed again in a confirmation window, which also displays the price of the gas for the contract deployment.

6.5 Sprint 2 evaluation

Sprint 2 was successful as the requirements chosen were implemented correctly and on time, despite the fact that the factory design and IPFS technology were unfamiliar initially. The system now supports for the creation and deployment of multiple smart contract instances, each of which now contains the CIDs of the products to be sold. This sprint's development brings the system closer to the goal system.

Since the CIDs are recorded on the blockchain, everyone can view them and fetch the valuable products, hence the system is still not secure. To ensure confidentiality, everything stored on the blockchain must be encrypted. This and other security concerns will be addressed in the forthcoming sprint.

Chapter 7

Sprint 3

Sprint 3 will focus on analyzing, designing, implementing, and testing a security protocol surrounding the system. Since the system developed will handle "creations of mind", each requirement fulfilled in the previous sprints needs to be performed in a secure manner. With every transaction on the blockchain being public, a system that protects the confidentiality of the files from unauthorized access needs to be implemented.

7.1 Analysis

A comprehensive step-by-step study of the security protocol is presented in the analysis section. Additionally, this section includes the rationale behind each decision and how these decisions contribute to the system's security.

7.1.1 Security protocol

1. **Files to sell** - Handling the files that are to be sold is the first stage in the security protocol. The files will be sold in bulk, and the seller must ensure that the product can only be viewed by authorized users (buyers once purchased). The seller achieves this confidentiality through encrypting each file with symmetric encryption, thus generating a set of keys. This set of keys remains with the seller for the time being.

2. **Sampling** - Buyers must ensure that the product they are acquiring meets their expectations and is appropriate. They accomplish this by supplying their public key and requesting a sample product. This sample must not have

been picked by the seller, since they may have chosen a sample that is of greater quality than the rest of the files on purpose. As a result, this selection must be a random file that will be selected by the smart contract deployed in the blockchain. Through blockchain, transparency on the selection is also achieved.

3. **Delivering sample** - Once a potential buyer has been identified and their public key has been received, the seller encrypts the set of keys using the buyer's public key and hashes the results. These hashed results are uploaded and stored on the blockchain for the smart contract to pick one at random. The seller must then provide the non-hashed version of the randomly selected sample in order for the potential buyer to decrypt the file and view the sample. This asymmetric encryption assures that only the buyer, using their private key, can decrypt the sample. Additionally, the stored hashes are utilized to verify that the key sampled and the set of keys to be delivered match and have not been altered.

4. **Delivery of products** - If the buyer is satisfied with the product, they can proceed to purchase the full product. As a result, the remaining keys are provided by the seller, encrypted with the buyer's public key. The smart contract rehashes these keys and compares them to the hashed keys that were initially uploaded. If everything is correct, the buyer receives the set of keys and the seller receives the Ether. The buyer has now acquired the keys to the IPFS files, and they must do a final decryption in order to view the product.

This protocol ensures that the seller maintains confidentiality of their product, even from the system itself. Because a malicious system administrator would not have access to the keys at any moment, the seller's obligation to trust the system itself is eliminated. Buyers, on the other hand, are informed that the product they are about to purchase meets the criteria and is not fraudulent (since the sample is chosen at random), and they are assured that the sample key comes from the set of keys they are about to purchase (through the hash comparison).

7.2 Design

To visualize the security protocol analyzed in 7.1.1, two sequence diagrams have been designed. The sequence diagram 8.3 depicts the actions taken in requesting a sample, whereas the diagram 7.2 represents the steps taken in purchasing the product.

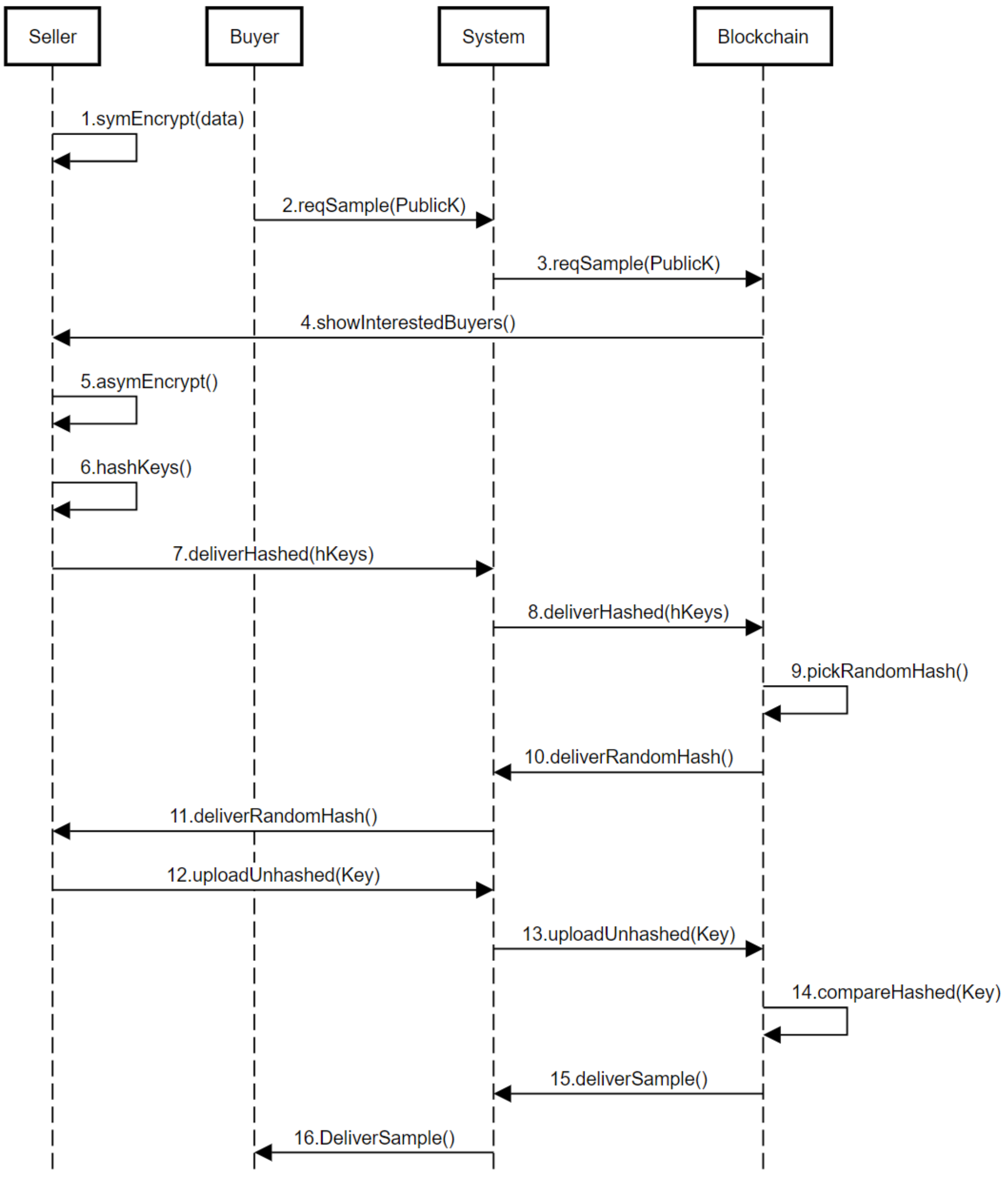


Figure 7.1: Sequence diagram for sample request

`1.` Seller encrypts their data symmetrically, generating a set of keys.

`2-3.` A potential buyer sends a sample request and provides their public key which is stored on the blockchain.

`4-6.` The seller is informed that there is a new potential buyer. They encrypt the set of keys from step 1 with the potential buyer's public key. The seller then hashes the results.

`7-13.` The hashed keys are stored on the blockchain, and the smart contract selects a random hash that corresponds to one key. The seller is presented this random hash, and they must then deliver the original key representing the chosen hash.

`14-16.` The smart contract validates the provided key (by hashing it and comparing the hash to the randomly picked hash) and then provides the key to the potential buyer.

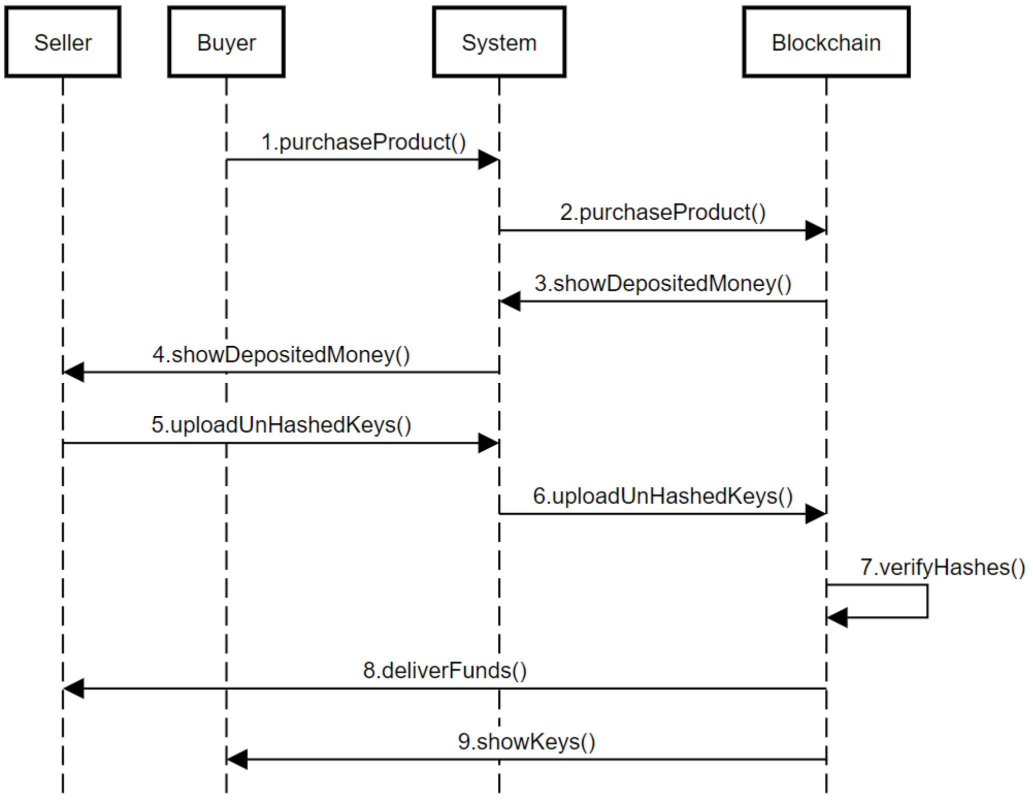


Figure 7.2: Sequence diagram for purchasing request

`1-2.` If the buyer is satisfied with the sample, they submit a purchase request. This request is delegated to the blockchain, which holds the funds until the seller delivers the keys.

`3-6.`The blockchain informs the seller that the buyer has submitted a purchase request and deposited funds. To claim these funds, the seller must upload the non-hashed keys.

`7-9.` The submitted keys are compared to the previously uploaded hashes (in the sampling step), and if they match, the transaction is valid. The blockchain delivers the funds to the seller and provides the buyer the keys.

7.2.1 Converting a public address into a public key

The second step in the security protocol (analysis section 7.1.1) is requesting a sample. In order to request a sample the potential buyer must provide their public key. The issue is that users do not have direct access to their public key; instead, they can only see their public address, which is a hashed form of the public key. In order to recover the public key from the hashed version, a user (Alice) must sign a transaction. This signing provides the signature values r, s which are needed alongside the message m (transaction) to perform the following public key recovery algorithm [69].

1. Validate that r and s are integers in $[1, n-1]$. If otherwise, the signature is incorrect.

2. Compute a curve point $R = (x_1, y_1)$, where x_1 is one of r, $r+n$, $r+2n$, (given x_1 is not too large for a field element) and y_1 is a value that satisfies the curve equation.

3. Compute $e = HASH(m)$, where HASH is the same algorithm that was used to generate the signature.

4. Let z represent the L_n leftmost elements of e.

5. Compute $u_1 = -zr^{-1} \bmod n$ and $u_2 = -sr^{-1} \bmod n$

6. Compute the curve point $Q_A = (x_A, y_A) = u_1 \times G + u_2 \times R$

7. If Q_A matches Alice's public address, the signature is valid and Q_A represents Alice's public key.

7.3 Implementation

This section gives an overview of the encryption schemes and tools utilized during this sprint, as well as a comprehensive breakdown of the smart contract functionalities related to the security protocol analyzed and designed in previous sections.

7.3.1 Encryption schemes

To maintain system security, the system employs a combination of encryption schemes, incorporating the benefits of each scheme.

7.3.1.1 AES

The Advanced Encryption Standard (AES) is a standard for digital data encryption developed by the United States National Institute of Standards and Technology. Despite being more difficult to implement, AES is frequently used today because it is substantially stronger than DES (Data Encryption Standard) [70]. A type of block cipher, AES is a symmetrical encryption scheme with key sizes that can range between 128, 192, or 256 bits. Data is encrypted in 128-bit chunks, meaning AES accepts 128 bits of input and produces 128 bits of encrypted cipher text [70]. AES is based on the substitution-permutation network principle, which implies that it is carried out by a series of connected processes that entail substituting and scrambling the input data [70]. AES has several modes of operations, including CTR (Counter), CBC (Cipher-Block Chaining), and ECB (Electronic Codebook). The AES encryption scheme will be utilized by the users to encrypt the microstock being sold.

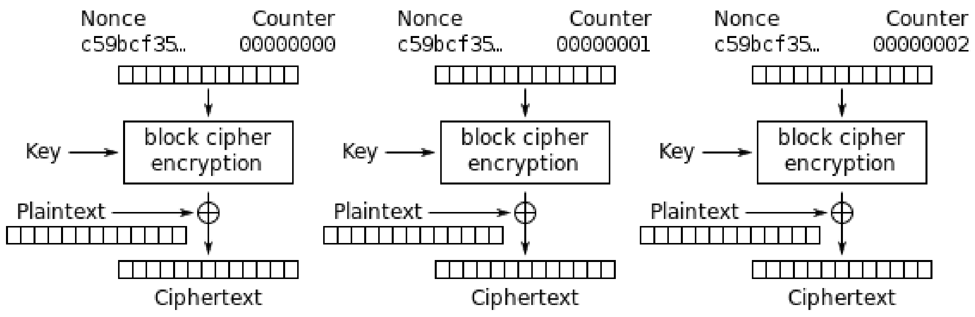


Figure 7.3: Counter (CTR) mode encryption

7.3.1.2 ECDSA

Elliptic Curve Digital Signature Algorithm (ECDSA) is a Digital Signature Algorithm that employs keys generated from elliptic curve cryptography [69]. An asymmetric encryption scheme, ECDSA uses the private key to create a digital signature for a message (or transaction), which can then be confirmed using the signer's associated public key. The digital signature ensures message authentication (the recipient can validate the origin of the message), integrity (the recipient can ensure that the message was not altered), and non-repudiation (the sender cannot deny signing the message) [69]. ECDSA is widely utilized in a variety of security systems, including Bitcoin and Ethereum, where users sign transactions and their addresses serve as the public key.

7.3.1.3 ECIES

To utilize ECDSA to encrypt and decrypt (rather than just sign), the system will employ an integrated encryption scheme known as the Elliptic Curve Integrated Encryption Scheme (ECIES). This allows users to encrypt files with the public key from their Ethereum address and decode them with their private key. This functionality represents a crucial component of the security system analysed and designed in sections 7.1 and 7.2.

7.3.2 Public key recovery implementation

Th algorithm behind recovering the public key was described under section 7.2.1, and listing 7.1 shows this implementation in JavaScript.

```
async function recoverPublicKey() {
    const ethAddress = await signer.getAddress();
    const hash = await ethers.utils.keccak256(ethAddress);
    const sig = await signer.signMessage(ethers.utils.arrayify(hash));
    const pk = ethers.utils.recoverPublicKey(
      ethers.utils.arrayify(
        ethers.utils.hashMessage(ethers.utils.arrayify(hash))
      ),
      sig
    );
    try {
      const transaction = await contract.requestSample(pk);
    } catch (error) {
      alert("The transaction couldn't be completed");
15          console.log(error);
16      }
17  }
```

Listing 7.1: Recovering the public key from the public address

`2-4.` The `ethAddress` represents the Ethereum address of the user. This address is then hashed with the `keccak256` hashing algorithm and stored under the constant `hash`. Now that the address and its hash are both given, the user generates the signature by signing a message of the hash.

`5-10.` To calculate the public key, first the message needs to be converted to binary: `ethers.utils.arrayify(hash)`. Next, the prefixed-message hash needs to be computed: `ethers.util.hashMessage(hashBytes)`. The result needs to be converted to binary: `ethers.utils.arrayify(messageHash)`. These steps provide the digest, which together with the signature (`4.`) are utilized to recover the public key: `ethers.utils.recoverPublicKey(digest ,signature)`.

`11-15.` With the public key recovered, the frontend delivers a sample request with the pk as the parameter.

7.3.3 Smart contract security system implementation

The sample request is the initial security protocol step. The function for this request is shown in listings 7.2 and 7.3.

```
1   contract Purchase{
2       string[] public interestedBuyers;
3           ...
4       function requestSample(string memory pkOfBuyer) public {
5           interestedBuyers.push(pkOfBuyer);
6       }
7
8       function getInterestedBuyers() public view returns (string[] memory
        ) {
9           return interestedBuyers;
10      }
11          ...
12  }
```

Listing 7.2: Sample request implementation 1

`2.` String array variable to hold all the public keys of the interested buyers.

`4-10`. Function for adding the public key (calculated in the frontend) to the array of interested buyers, and a return function to get this variable.

After encrypting the keys to the files with the buyer's public key and hashing them (explained under 7.1.1), the seller uploads these hashes to the blockchain. Listing A.5 shows this function.

```
contract Purchase{
        ...
    string[] public hashedSamples;
    string public randomHashPicked;
    string public unHashedSample;
        ...
    function pickHashedSample(string[] memory hashedKeys) public
    OnlyOwner {
        hashedSamples = hashedKeys;
        uint256 randomIndex = uint256(
            keccak256(abi.encodePacked(block.timestamp, msg.sender))
        ) % hashedKeys.length;
        randomSampleId = randomIndex;
        randomHashPicked = hashedKeys[randomIndex];
    }

    function returnRandomHashPicked() public view returns (string
    memory randHash, uint256 hashId){
        return (randomHashPicked, randomSampleId);
    }

    function putUnhashedSample(string memory unHashedS) public
    OnlyOwner {
        string memory hashingToCompare = Strings.toHexString(
            uint256(keccak256(abi.encodePacked(unHashedS))),
            32
        );
        require(
            keccak256(abi.encodePacked(hashingToCompare)) ==
                keccak256(abi.encodePacked(randomHashPicked)),
            "hashes do not match"
        );
        unHashedSample = unHashedS;
    }
        ...
}
```

Listing 7.3: Sample request implementation 2

`3-5.` Variable declaration for the hashed samples, the random hash picked, and the non-hashed sample.

`7.` When the user has encrypted the keys and hashed them, they upload them to the blockchain with this function.

`8-13.` The provided hashes are first stored in the blockchain (`8.`), then a random sample is picked by the smart contract. Solidity contracts are deterministic, thus it is difficult to obtain a truly random number; however, a pseudo random number is sufficient for this system. To generate the pseudo random index, the timestamp of the block is combined with the msg.sender and hashed. To get an index within the range of the array, result modulo the length of the array is calculated. Lastly, the random sample id (index) and the random hash picked are stored on the blockchain.

`16-18.` The randomly selected hash is returned, along with the hash's id. This is done since the number of hashes is large and the user can get lost attempting to locate the randomly selected hash.

`20.` Once the randomly selected hash has been returned to the seller, they must provide the unhashed version. The function in line `20` is responsible for this operation.

`21-29.` The provided sample key (`unHashedS`) must be hashed and compared to the hash uploaded in the previous step. To hash the input, the keccak256 function (part of SHA-3) is called. Solidity provides the hashing function directly, however it expects bytes as input, therefore to hash a string, the function `encodePacked()` is used. This function returns a bytes32 hash of the unhashed key. To convert from bytes32 to hex string (which is what the `randomHashPicked` is stored as), the function `Strings.toHexString()` is utilized. The final step is comparing `randomHashPicked` (the original hashed sample chosen) with the variable `hashingToCompare`. In Solidity the `"=="` operator is not compatible for string types, but it is for bytes32, therefore the two hex strings are hashed again to produce bytes32 outputs which are comparable.

`30.` If the hashes match, the unhashed sample is stored on the blockchain for the interested buyer to view.

```
contract Purchase{
        ...
```

```
 3      address payable public buyerAddress;
 4      uint256 public depositTime;
 5          ...
 6      function purchaseProducts() public payable SellerCantBuy {
 7          require(requestedAmount == msg.value, "invalid amount");
 8          buyerAddress = payable(msg.sender);
 9          depositTime = block.timestamp;
10      }
11
12      function returnDeposit() public {
13          require(msg.sender == buyerAddress, "You have nothing deposited
        ");
14          require(
15              block.timestamp - depositTime > 86400,
16              "24 hours have not passed yet, please wait"
17          );
18          buyerAddress.transfer(address(this).balance);
19      }
20          ...
21  }
```

Listing 7.4: Purchase request function

`3-6.` If the buyer is satisfied with the sample, they send a purchase request. This request is handled by the `purchaseProducts()` function (line `5`). The Ethereum address of the account purchasing the product is also stored in the variable `buyerAddress` (line `3`).

`7-9.` The purchase function is a `payable` function, meaning the function accepts Ethers that are passed to the it. This Ether will be deposited in the smart contract and must match the requested amount (specified when the contract is constructed) by the seller. If this is the case, the address is updated to hold the `msg.sender` address, which corresponds to the buyer calling the function. Lastly, the deposit time is recorded on the blockchain.

`12-19.` After the funds have been deposited, the seller has 24 hours to provide the rest of the keys. If they fail to do so, the buyer can take back their deposit. The function in line `12.` is responsible determining whether the `msg.sender` (function caller) matches the address of the user that deposited the Ether and whether the required amount of time has passed. In this case, the funds will be transferred back to the buyer's account.

```
1  contract Purchase{
2          ...
3      function withdraw(string[] memory uhashedKeys) public OnlyOwner {
4           require(address(this).balance != 0, "there is no deposited
        money");
5           require(uhashedKeys.length == hashedSamples.length);
6           for (uint256 i = 0; i < uhashedKeys.length; i++) {
7               require(keccak256(abi.encodePacked(Strings.toHexString(
8                           uint256(keccak256(abi.encodePacked(uhashedKeys[
        i]))), 32
9                       ))
10                  ) == keccak256(abi.encodePacked(hashedSamples[i])),
11                  "product does not match the originally uploaded one"
12              );
13          }
14          unHashedKeys = uhashedKeys;
15          beneficiary.transfer(address(this).balance);
16      }
17
18      function getProduct() public view returns (string[] memory) {
19          require(msg.sender == buyerDeposit, "You can not collect this
        product");
20          return unHashedKeys;
21      }
22
23      function finish() public {
24          require(msg.sender == buyerDeposit, "you cannot call this
        function");
25          isFinished = true;
26      }
27          ...
28  }
```

Listing 7.5: Withdraw funds function

`3.` To withdraw the funds deposited, the seller needs to provide the rest of keys, which are encrypted with buyers public key and not hashed. The function `withdraw` takes as input this set of unhashed keys and compares them with the originally uploaded hashed version.

`4-5.` To withdraw, there needs to be funds deposited in the smart contract, and the length of the unhashed keys array must match the length of the hashed keys array.

`5-11.` When comparing the keys, the function follows the same pattern described under listing 7.3, with the exception that it does this comparison for each key in the array.

`14-15`. If each key matches each hash, the unhashed version of the keys is stored on the blockchain, and the Ether is transferred from the smart contract to the seller.

`18-26`. Once the keys are uploaded to the blockchain, the buyer can fetch them with the `getProducts()` function. Additionally, the finish function is called by the buyer to indicate that this product has been purchased (important for the frontend to not load finished contracts).

7.3.4 Frontend implementation

The frontend was updated with additional functionalities to enable users to connect with the smart contract operations. Some of the new functions include:

```javascript
      ...
  async function provideHashedKeys() {
  if (document.getElementById("hashedKeys").value == "") {
    alert("Hashed keys field cannot be empty");
  } else {
    let longString = document.getElementById("hashedKeys").value;
    let arrayString = longString.split(",").map((el) => "0x" + el);
    try {
      const transaction = await contract.pickHashedSample(arrayString);
      const transaction2 = await contract.returnRandomHashPicked();
      alert(
        "The random hash that was picked and needs to be provided unhashed is hash nr " +
          transaction2[1] + ". " + transaction2[0]
      );
    } catch (error) {
      alert("Only the owner can provide keys");
      console.log(error);
    }
  }
}

async function provideUnHashedKeys() {
  if (document.getElementById("unHashedKeys").value == "") {
    alert("Un-Hashed keys field cannot be empty");
  } else {
    let unHashedKey = document.getElementById("unHashedKeys").value;
27       try {
28         const transaction = await contract.putUnhashedSample(
     unHashedKey);
29         alert(
30           "The unhashed key was delivered"
31         );
32       } catch (error) {
33         alert("The unhashed keys do not match the orignially uploaded
     ones");
34         console.log(error);
35       }
36     }
37   }
38         ...
```

Listing 7.6: Frontend implementation

`2-20` The frontend function for providing the hashed keys. Line `7.` converts the hash array to include the 0x in front of each element, as this is the standard in Solidity hash algorithms. It is important to convert in the frontend so that the smart contract can afterward compare hashes. Lines `8-18.` implement the transaction of calling the contract functions. The contract object was described in Sprint 1, under listing 5.3.

`22-37.` The function `provideUnHashedKeys()`, follows a similiar pattern to the `provideHashedKeys()` function. Keys are fetched, the contract calls are conducted, and any errors are caught. Since the remaining frontend functions do not demonstrate any new concepts, they will not be included in the report.

7.4 Testing

Similarly to Sprint 1 and 2, automated testing will be performed utilizing Chai.js and will test the newly implemented security system functionalities of the smart contract.

7.4.1 Smart contract security system testing

```
it("Should return a random hash from an array", async function () {
    const purchase = await deploy(
      testTitle,
      ethers.utils.parseEther("10"),
      accounts[0].address,
6        testCIDs
7      );
8      const testHash1 = "0x9c22ff5f21f0b81b113e63f7db6da94fedef11b2119b4088b89664fb9a3cb658";
9      const testHash2 = "0x6d255fc3390ee6b41191da315958b7d6a1e5b17904cc7683558f98acc57977b4";
10     const testHash3 = "0x4da432f1ecd4c0ac028ebde3a3f78510a21d54087b161590a63080d33b702b8d";
11     const testSampleKeys = [testHash1, testHash2, testHash3];
12     const owner = purchase.connect(accounts[0]);
13
14     await owner.pickHashedSample(testSampleKeys);
15     const pickedSample = await owner.returnRandomHashPicked();
16     const pickedSampleNumber = await owner.returnSampleid();
17     expect(testSampleKeys.includes(pickedSample.randHash)).to.equal(true)
18   });
```

Listing 7.7: Returning random hash test

2-7. Deploying an instance of the Purchase smart contract.

8-11. Three sample hashes and the array of hashes.

12-17. Calling the smart contract functions with the provided test variables. Since the function returns one of the three hashes at random, the `expect` (line 17) cannot know the exact return, but it expects that it will return one of the three.

```
it("Should match unhashed to the initial hash", async function () {
    const purchase = await deploy(
      testTitle,
      ethers.utils.parseEther("10"),
      accounts[0].address,
      testCIDs
    );
    const testUnHashed = "test1";
    const testHash1 = "0x6d255fc3390ee6b41191da315958b7d6a1e5b17904cc7683558f98acc57977b4";
    const testSampleKeys = [testHash1];
    const owner = purchase.connect(accounts[0]);
    await owner.pickHashedSample(testSampleKeys);
    await owner.putUnhashedSample(testUnHashed);
    const unHashReturned = await owner.returnUnHashedSample();

    await expect(testUnHashed).to.be.equal(unHashReturned[0]);
17   });
```

Listing 7.8: Matching initial hash test

`8-10` When the string “test1” is hashed using keccak256 the result is the variable stored under `testHash1`.

`8-16` If the hashed array contains only one element, the randomly chosen sample must be that element. Knowing this, the function is tested that the smart contract can hash “test1” and compare it to the `testHash1`.

```
1  it("Should be able to purchase", async function () {
2      const purchase = await deploy(
3        testTitle,
4        ethers.utils.parseEther("10"),
5        accounts[0].address,
6        testCIDs
7      );
8      const price = ethers.utils.parseEther("10");
9      const buyer = purchase.connect(accounts[1]);
10     const buyerInitialBalance = await ethers.provider.getBalance(
11       accounts[1].address
12     );
13     const transaction = await buyer.purchaseProducts({
14       value: price,
15     });
16     const receipt = await transaction.wait();
17     const gasUsed = receipt.cumulativeGasUsed.mul(receipt.
    effectiveGasPrice);
18     await expect(
19       await ethers.provider.getBalance(accounts[1].address)
20     ).to.be.equal(buyerInitialBalance.sub(price).sub(gasUsed));
21   });
```

Listing 7.9: Purchase request test

`10-20` With the initial requested amount given, the buyer should be able to send a purchase request to the smart contract. To confirm the transaction was successful, the buyer’s balance is compared with the initial balance. If the test is successful, the price of the product and the gas used for the transactions should be deducted from the balance.

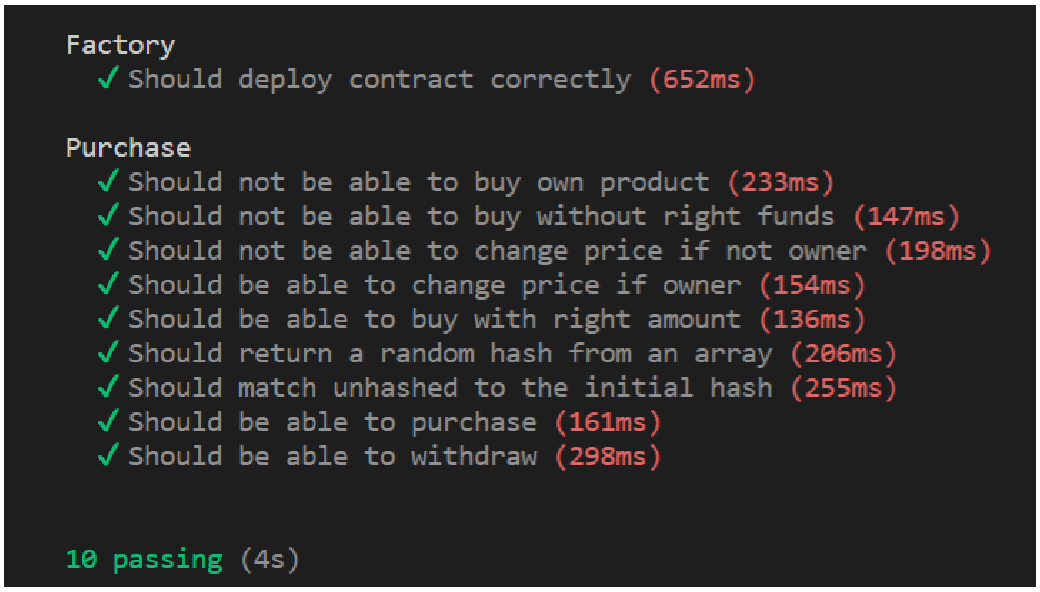
```
1  it("Should be able to withdraw", async function () {
2      const purchase = await deploy(
3        testTitle,
      ethers.utils.parseEther("20"),
      accounts[0].address,
      testCIDs
    );
    const owner = purchase.connect(accounts[0]);
    const ownerInitialBalance = await ethers.provider.getBalance(
      accounts[0].address
    );
    const buyer = purchase.connect(accounts[1]);
    const price = ethers.utils.parseEther("20");
    const testHash1 = "0x9c22ff5f21f0b81b113e63f7db6da94fedef11b2119b4088b89664fb9a3cb658";
    const testHash2 = "0x6d255fc3390ee6b41191da315958b7d6a1e5b17904cc7683558f98acc57977b4";
    const testHash3 = "0x4da432f1ecd4c0ac028ebde3a3f78510a21d54087b161590a63080d33b702b8d";
    const testSampleKeys = [testHash1, testHash2, testHash3];
    const testUnHashedKeys = ["test", "test1", "test2"];
    const transaction1 = await owner.pickHashedSample(testSampleKeys);
    const receipt1 = await transaction1.wait();
    const gasUsed1 = receipt1.cumulativeGasUsed.mul(receipt1.effectiveGasPrice);
    await buyer.purchaseProducts({
      value: price,
    });
    const transaction2 = await owner.withdraw(testUnHashedKeys);
    const receipt2 = await transaction2.wait();
    const gasUsed2 = receipt2.cumulativeGasUsed.mul(receipt2.effectiveGasPrice);

    await expect(
      await ethers.provider.getBalance(accounts[0].address)
    ).to.be.equal(ownerInitialBalance.add(price).sub(gasUsed1).sub(gasUsed2));
  });
```

Listing 7.10: Purchase request test

The final test case determines whether the withdraw function is operating correctly. To verify the withdraw function, the seller must first submit the hashed keys and the sample. When the buyer makes the purchase request, the owner must provide the non hashed keys. If everything works as expected, the seller's balance should increase by the requested amount.

7.4.2 Automated testing results

The testing was conducted in the same approach as in Sprint 1 and Sprint 2 (section 5.4.2 and 6.4.2), with the hardhat command: `npx hardhat test`. According to the results, every test was passed, and all expectations were met.



```
  Factory
    ✓ Should deploy contract correctly (652ms)

  Purchase
    ✓ Should not be able to buy own product (233ms)
    ✓ Should not be able to buy without right funds (147ms)
    ✓ Should not be able to change price if not owner (198ms)
    ✓ Should be able to change price if owner (154ms)
    ✓ Should be able to buy with right amount (136ms)
    ✓ Should return a random hash from an array (206ms)
    ✓ Should match unhashed to the initial hash (255ms)
    ✓ Should be able to purchase (161ms)
    ✓ Should be able to withdraw (298ms)


  10 passing (4s)
```

Figure 7.4: Testing results. Includes tests from Sprints 1-2

7.5 Sprint 3 evaluation

Sprint 3 addressed the system's most challenging requirements. Analysing, designing, implementing, and testing a security system that is truly secure for both sellers and buyers proved to be difficult. The necessity to recover the public key from the public address was not anticipated and only appeared during the analysis phase. Despite the Sprint's high complexity, all requirements were implemented successfully and on schedule. The completion of this sprint takes the system very close to the target system.

Currently, the system is fully operational and functional. Although to properly utilize the system, users need to have access to implementations of the AES and ECIES encryption algorithms. During the forthcoming Sprint 4 these encryption schemes will be provided to users who wish to use it. Lastly, the frontend design needs to be updated to make it more attractive for users.

Chapter 8

Sprint 4

The final sprint, Sprint 4, will focus on analyzing, designing, implementing, and testing encryption mechanisms, adding a history of contract purchases, and re-designing the user interface. Although the system has full blockchain and smart contract functionality, users must still employ various encryption tools to use the system. The final sprint will focus on the requirement of developing these tools within the frontend for users to use. The encryption schemes that will be implemented include: AES, ECIES, and Keccak256.

Finally, the sprint evaluation section will be expanded to its own chapter, where both the sprint and the entire project will be evaluated.

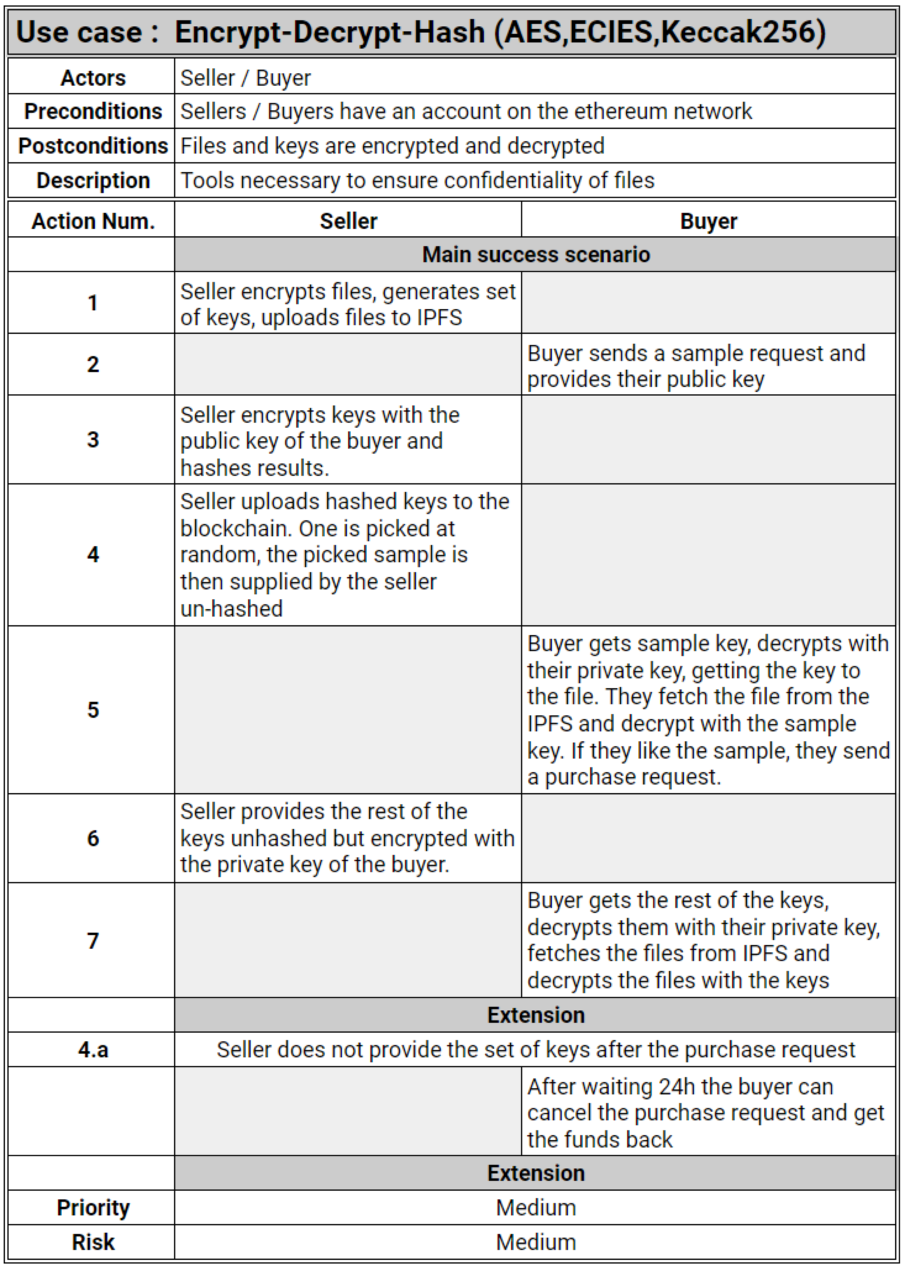
8.1 Analysis

The system implemented in previous sprints is completely functional, but the obligation for users to employ different available encryption tools creates a negative user experience. Furthermore, while there are tools available online for symmetrical file encryption (AES) and for hashing (Keccak256), there are none supporting asymmetrical encryption using the public keys of Ethereum accounts (ECIES). To complete the system and improve the user experience, all the necessary encryption tools will be implemented directly on the frontend for users to utilize.

Moreover, although only active smart contracts (products that can be purchased) are currently displayed, the blockchain keeps records of all transactions, and allowing users to view the history of who purchased what will be a useful addition to the system, further increasing its transparency and immutability.

Finally, considering previous sprints' frontend development prioritized functionality over appearance, on the final sprint the frontend requires improvement to provide a better user experience.

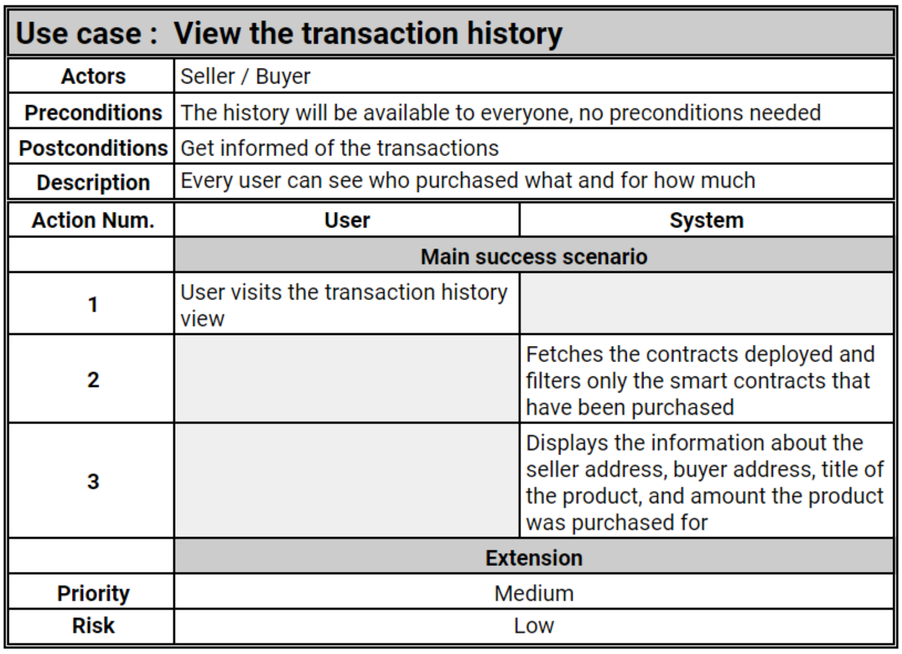
To visualize the requirements chosen to be implemented for Sprint 4, two use cases have been created. Figure 8.1 depicts the use case for sellers and buyers utilizing the encryption tools in order to utilize the system.



Use case : Encrypt-Decrypt-Hash (AES,ECIES,Keccak256)		
Actors	Seller / Buyer	
Preconditions	Sellers / Buyers have an account on the ethereum network	
Postconditions	Files and keys are encrypted and decrypted	
Description	Tools necessary to ensure confidentiality of files	
Action Num.	**Seller**	**Buyer**
	Main success scenario	
1	Seller encrypts files, generates set of keys, uploads files to IPFS	
2		Buyer sends a sample request and provides their public key
3	Seller encrypts keys with the public key of the buyer and hashes results.	
4	Seller uploads hashed keys to the blockchain. One is picked at random, the picked sample is then supplied by the seller un-hashed	
5		Buyer gets sample key, decrypts with their private key, getting the key to the file. They fetch the file from the IPFS and decrypt with the sample key. If they like the sample, they send a purchase request.
6	Seller provides the rest of the keys unhashed but encrypted with the private key of the buyer.	
7		Buyer gets the rest of the keys, decrypts them with their private key, fetches the files from IPFS and decrypts the files with the keys
	Extension	
4.a	Seller does not provide the set of keys after the purchase request	
		After waiting 24h the buyer can cancel the purchase request and get the funds back
	Extension	
Priority	Medium	
Risk	Medium	

Figure 8.1: Use case for encryption and decryption tools

The actors are both the seller and the buyer, as they will both employ encryption techniques to ensure the files' confidentiality. The steps follow the security system analysed, designed, and implemented during Sprint 3 (section 7.1.1). The extension depicts the situation whereby the buyer can cancel the purchase request and reclaim the payments if the seller fails to supply the keys within 24 hours after the request. As the users are free to find other solutions for encryption tools, the use case's priority and risk remain on a medium level.



Use case : View the transaction history		
Actors	Seller / Buyer	
Preconditions	The history will be available to everyone, no preconditions needed	
Postconditions	Get informed of the transactions	
Description	Every user can see who purchased what and for how much	
Action Num.	**User**	**System**
	Main success scenario	
1	User visits the transaction history view	
2		Fetches the contracts deployed and filters only the smart contracts that have been purchased
3		Displays the information about the seller address, buyer address, title of the product, and amount the product was purchased for
	Extension	
Priority	Medium	
Risk	Low	

Figure 8.2: Use case for viewing the transaction history

Although not an essential requirement, viewing the complete transaction history increases the system's transparency and immutability. Once purchased, the contracts are permanently stored on the blockchain and cannot be altered. These contracts make up the transaction history of the application and will be visible to all users. Because it is not fundamental to the system, the priority of the requirement is medium, and the risk is low due to the ease of implementation.

8.2 Design

To illustrate the design of how users interact with the encryption tools of the system diagrams 8.3 and 8.4 have been designed and described.

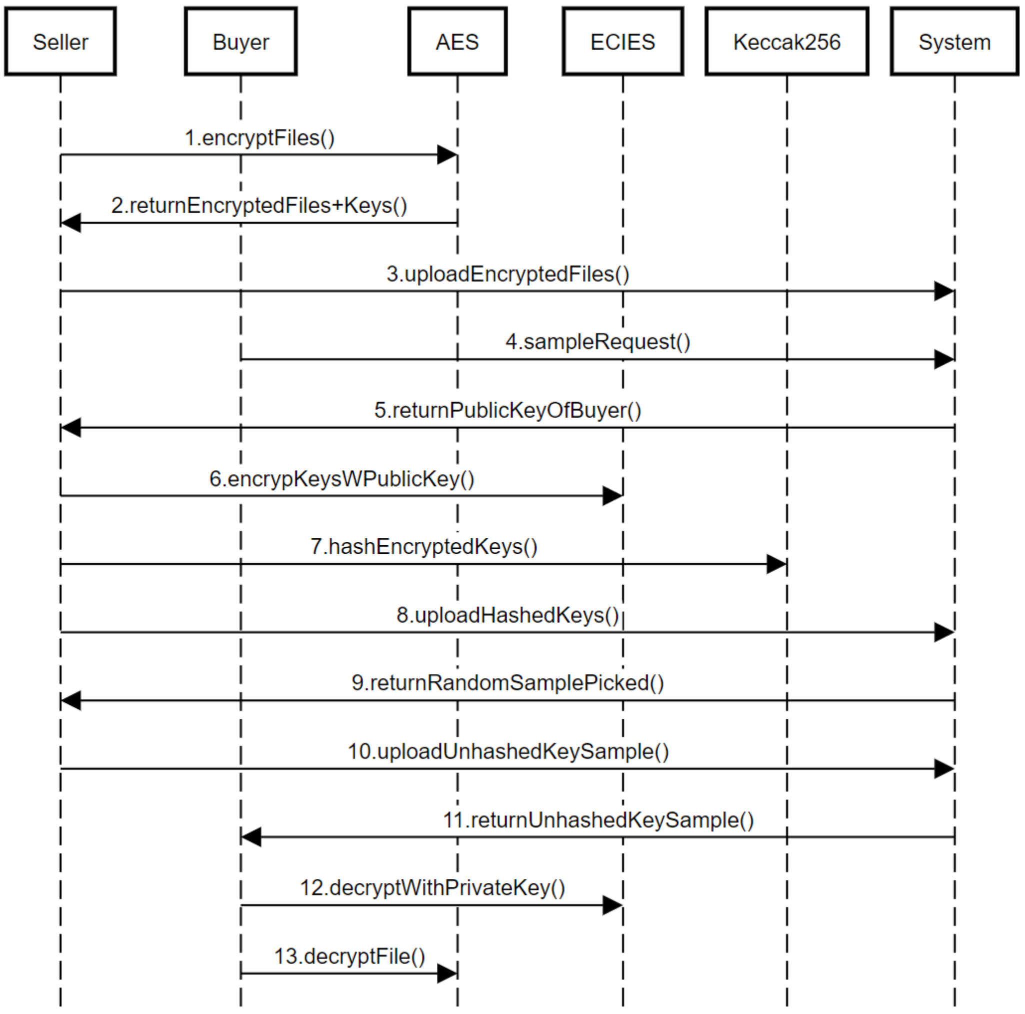


Figure 8.3: Sequence diagram for sampling

The seller initially interacts with the AES encryption tool to encrypt the files they intend to sell. This interaction results in a set of encrypted files as well as a set of keys. After uploading the files to the system and receiving an interested buyer, the seller encrypts the set of keys from step 1 using the interested buyer's public key. This collection of keys is hashed and sent to the system in step 7 so that a random sample key may be chosen. The seller then uploads the unhashed sample key, which is received by the buyer. The buyer then decrypts the key with their private key (ECIES step 12) and uses that key to decrypt the sample file (AES step 13). This concludes the sampling procedure and if the buyer is satisfied with the sample, they send a purchase request (depicted in sequence diagram 8.4).

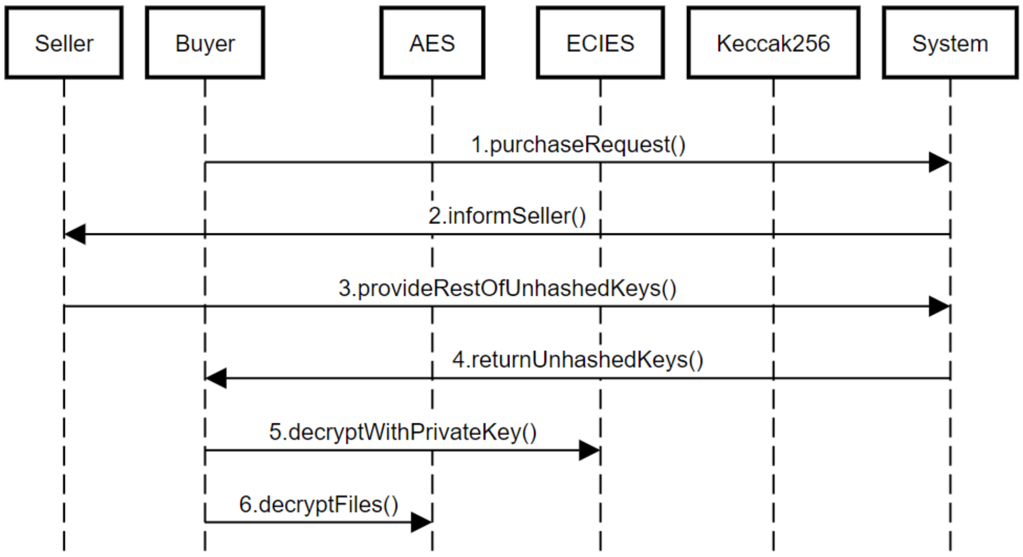


Figure 8.4: Sequence diagram for buying

Once the buyer has submitted a purchase request, the seller is obliged to provide the remaining keys unhashed and encrypted with the buyer's public key. This set of keys is returned to the buyer, which they decrypt using their private key (ECIES). Lastly, the buyer utilizes AES decryption with the decrypted set of keys to decrypt the files sold.

8.3 Implementation

This section provides an overview of the encryption schemes implementation. As each scheme was described and addressed in the previous sprint (section 7.3.1), this section will focus solely on implementation. Finally, while the Keccak256 and IPFS file retrieval tools were implemented during this sprint, they are comparable to previous implementations and will not be addressed in depth. The IPFS implementation is comparable to the one in section 6.3, and Keccak256 implementation was utilized in section 7.1.

8.3.1 AES Encryption - Decryption Implementation

```
async function encryptAES() {
    let encryptedFilesArray = await Promise.all(
      images.files.map(async (element) => {
```

```
4         const key = new Uint8Array(16);
5         window.crypto.getRandomValues(key);
6         const buffer = new Uint8Array(await element.arrayBuffer());
7         const ctr = new aesjs.ModeOfOperation.ctr(key);
8         const encryptedFile = ctr.encrypt(buffer);
9         const encryptedFileHex = aesjs.utils.hex.fromBytes(
    encryptedFile);
10        var keyHex = aesjs.utils.hex.fromBytes(key);
11        const file = {
12          name: element.name,
13          hex: encryptedFileHex,
14          key: keyHex,
15        };
16        return file;
17      })
18    );
19        ...
20    encryptedFilesArray.forEach((file) => {
21      var element = document.createElement("a");
22      element.setAttribute(
23        "href",
24        "data:text/plain;charset=utf-8," + encodeURIComponent(file.hex)
25      );
26      element.setAttribute("download", file.name);
27      element.style.display = "none";
28      document.body.appendChild(element);
29      element.click();
30      document.body.removeChild(element);
31    });
32  }
```

Listing 8.1: AES encryption

`1-3.` Since there are asynchronous operations, the function to encrypt is `async` and the requests are turned into promises and awaited. The array `images` being mapped holds each file uploaded to be encrypted.

`4-5.` A 128-bit (16-byte) key is required to encrypt. AES also works with keys of 192 bits (24 bytes) or 256 bits (32 bytes), but for this system, 16 bytes are adequate. The 16 bytes are stored in the `Uint8Array(16)` variable which represents 16 8-bit unsigned integers, and using `crypto.getRandomValues(key)` the key is generated. This function fills the array with random values, resulting in a random key.

6. The `element` represents each file that will be encrypted, and the function `.arrayBuffer()` transforms the element (image) to an array of raw binary data. This raw binary data is transformed into an `Uint8Array` since this is the type AES expects in order to encrypt.

7. AES operates in several modes, and this is where the desired mode is selected. The CTR mode was chosen for this system because to its high security and ease of implementation.

8-16. The buffer (image) is encrypted and then converted from bytes to a hexadecimal string to save space. Similarly the key is also converted to a hex string and a file object is created with name, hex, and key attributes. This file represents the encrypted file.

20-30. An element is produced for each encrypted file and used to download the encrypted files locally on the user's computer. A similar approach is also used to download the set of keys generated during encryption.

```javascript
async function decryptAES() {
    const key = await keys.text();
    let keyArray = key.split(",");
    let decryptedFilesArray = await Promise.all(
      images.files.map(async (element, index) => {
        const fileBytes = new aesjs.utils.hex.toBytes(await element.text());
        const ctr = new aesjs.ModeOfOperation.ctr(
          aesjs.utils.hex.toBytes(keyArray[index])
        );
        const decryptedFile = ctr.decrypt(fileBytes);
        const file = {
          name: element.name,
          decrypt: decryptedFile,
        };
        return file;
      })
    );

    decryptedFilesArray.forEach((file) => {
      const blob = new Blob([file.decrypt], { type: "image/png" });
      if (window.navigator.msSaveOrOpenBlob) {
        window.navigator.msSaveBlob(blob, file.name);
      } else {
        const elem = window.document.createElement("a");
        elem.href = window.URL.createObjectURL(blob);
26        elem.download = file.name;
27        document.body.appendChild(elem);
28        elem.click();
29        document.body.removeChild(elem);
30      }
31    });
32 }
```

Listing 8.2: AES decryption

`2-3. keys.text()` coverts the keys file into a stream and reads it to completion. It returns a promise that resolves to a string. This string will hold each key separated by a comma, and to split this string into an array of keys the function `key.split()` is used.

`5-15.` The `images` array now holds the encrypted files, and each element is turned into a stream of text. This stream will be the hex string that the file was stored as during encryption, and it must be converted back to bytes before it can be decrypted. This is done with `hex.toBytes()` function. To decrypt, the file bytes are matched with the appropriate key, which is located at the same index as the element. This key, like the files, was stored as a hex string and must be converted to bytes before it can be used for decryption. In line `10`, the file is decrypted using the same AES mode of operation (CTR) and returned as an object to the `decryptedFilesArray`.

`19-30` The decrypted file is now in byte format and needs to be converted to a PNG. To do this, a Blob object is instantiated to retain the data, and the PNG is downloaded to the user's computer.

8.3.2 ECIES Encryption - Decryption Implementation

```
async function encryptEcies() {
    const key = await keys.text();
    let keyArray = key.split(",");
    const pk = ethers.utils.arrayify(
      document.getElementById("publicKey").value
    );
    let encryptedKeys = await Promise.all(
      keyArray.map(async (el) => {
        const encrypted = await EthCrypto.encryptWithPublicKey(pk, el);
        const stringEncrypted = EthCrypto.cipher.stringify(encrypted);
        return stringEncrypted;
      })
13      );
14          ...
15
16      // iv: '02aeac54cb45283b427bd1a5028552c1',
17      // ephemPublicKey: '044
        acf39ed83c304f19f41ea66615d7a6c0068d5fc48ee181f2fb1091...',
18      // ciphertext: '5fbbcc1a44ee19f7499dbc39cfc4ce96',
19      // mac: '96490
        b293763f49a371d3a2040a2d2cb57f246ee88958009fe3c7ef2a38264a1'
20  }
```

Listing 8.3: ECIES encryption

`1-2`. The file of keys is turned into an array of keys.

`4-6`. The public key is provided by the seller but this key must be converted to a `uint 8 byte array` before it can be used for encryption. This is achieved with the `utils.arrayify()` function.

`7-13`. Each key (to the files) is encrypted with the public key (Ethereum account) provided. Lines `16-19` show an example of the object created by the encryption function. This object can be parsed and turned into a single string with the function `cipher.stringify()`. This string can now only be decrypted with the private key of the account. Lastly, similarly to the AES encryption / decryption, the stringified result is downloaded to the user's computer as a file.

```
1   async function decryptEcies() {
2       const key = await keys.text();
3       let keyArray = key.split(",");
4       const pk = document.getElementById("privateKey").value;
5
6       let decryptedKeys = await Promise.all(
7         keyArray.map(async (el) => {
8           const parsed = EthCrypto.cipher.parse(el);
9           const decrypted = await EthCrypto.decryptWithPrivateKey(pk,
        parsed);
10          return decrypted;
11        })
12      );
13          ...
14  }
```

Listing 8.4: ECIES decryption

`1-12`. The ECIES decrypt function is the inverse of the encrypt function described in Listing 8.3. The private key is provided by the buyer, and each file key is parsed to be converted to the object depicted under Listing 8.3, lines `16-19`. After this conversion is complete, the file keys can be decrypted with the Ethereum account's private key.

8.4 Testing

The final testing for each encryption scheme was performed manually, ensuring that each encryption method operated as envisioned in both regular and edge cases.

The AES encryption tool was first tested through encrypting strings, then text files, and once the results were satisfactory, the tool was tested against the main product to be encrypted, pictures. The tool was tested for encrypting single and large quantities of pictures, and in both cases the tool successfully encrypted the files and generated corresponding keys for each file. Similarly, performance was measured dependent on the quantity of files to be encrypted, and despite the increased number of files, the system performed efficiently.

For AES decryption, a similar approach was followed. Initially, string and text file decryption were tested, followed by image decryption testing. Files could only be decrypted with the corresponding key, and no other keys or techniques could be used. This confirmed that the key was unique, and that only the authorized individual with access to it could decrypt the data.

Since hashes are not reversible, the testing for Keccak256 has to be handled differently. Hashes are irreversible, however hashing the same data must result in the same hash. This concept was tested with multiple hashes of the same data, and the results were compared and confirmed to be identical each time. Furthermore, to verify that the hashing algorithm was appropriately implemented, an online Keccak256 tool was found, and when supplied with identical data, both the online and system's implementations produced identical hashes.

Lastly, the ECIES implementation was also tested in similar fashion as the two previous encryption schemes. Encryption was examined using various public keys, and each time the data could only be decrypted by the owner of the Ethereum account who has access to the private key.

Chapter 9

Evaluation

The evaluation chapter is critical in software development since it provides insights on how proximate the developed system is compared to the specified definition, how effectively it satisfies functional and non-functional criteria, and what the general quality level is.

9.1 Evaluation of the system objectives

In the introduction chapter of this project, the rationale behind developing the system was given and three foundational objectives were set. These objectives were:

1. **Decentralization.**
2. **Offer sampling, purchasing, and delivering of microstock.**
3. **Security.**

Regarding the first objective, the system developed adhered to its vision of being entirely decentralized and a true Web 3.0 application. Users have complete control over the data they share, and no information is stored in a central database. Even users' usernames and accounts are not saved, ensuring complete privacy. When data storage was necessary, in the form of microstock uploaded to be sold, the decentralized peer-to-peer IPFS network was utilized, with any data uploaded encrypted beforehand. The system has no central authority to regulate or censor users, and all transactions are completely transparent.

The second objective encompassed the functional requirements of the system. The developed system allows users to sample and purchase microstock. Following the purchase, the system transfers funds to the buyer and delivers the product to the buyer, all without an intermediary facilitating the transactions.

All of the functional requirements of the second objective have to be secure for both sellers and buyers. This was the project's third and final objective. Since there is no intermediary to secure the transactions, security was addressed by employing various cryptographic schemes and combining symmetrical and asymmetrical encryption. With all three objectives met, the final evaluation of the system's requirements is successful.

9.2 Evaluation of the user interface

A user evaluation was performed to determine whether the user interface is user-friendly, unobtrusive, and appealing. The subjects questioned were classmates and friends. This provided a sample of users with and without computer science backgrounds. On all pages, users were surveyed about the clarity, structure, colours, and overall design.

9.2.1 User evaluation results

The results received were mostly positive. The overall quality of each screen was determined using a rating system ranging from 1 to 5. Regarding the main page, where the products are displayed, 73 percent of respondents stated that the products are clear and easily discernible. This page received the highest rating, followed by the transaction history page with 62 percent giving it the highest score, and the encryption tools page ranked in last with 60 percent. Regarding the colors, organizational level, and overall look of the interface, 65 percent of respondents assigned it the highest grade, 25 percent rated it a 4, leaving only 10 percent of the respondents rating the user interface with a 3 or less.

Chapter 10

Conclusion

The final year project provided an opportunity to develop a project of considerable size using multiple technologies and tools that were initially new to the developer. The supervisor's ongoing direction helped this project to continuously stay on course, resulting in a final output that can be regarded as both successful and significant. Many challenges were encountered during the project's development, however the end goal always appeared to be attainable. The final concluding chapter will examine the difficulties encountered and how they were addressed, discuss the present state of the system, as well as mention future improvements to the system.

10.1 Challenges encountered

Although the desired objectives set initially were all met successfully, various issues arose during development. These issues posed challenges to the system's development, and the most significant challenges that had to be overcome are listed and explained below.

Setting up the blockchain environment - Due to the complexity of the blockchain technology and the initial unfamiliarity from the developer with it, the first major challenge showed up early. This challenge was anticipated, and the decision was made during the planning phase to address it in sprint 1. This decision proved to be correct, as setting up a local blockchain, deploying contracts, linking wallets, and integrating these interactions from the React frontend proved difficult. In order to set up the foundational blockchain environment correctly, several papers were examined and significant effort was put in to thoroughly understand how the blockchain operates and how smart contracts execute code on it. The significant effort spent proved effective since the foundation did not need to be changed later in the development process, and understanding the blockchain early on aided in speeding up the development processes throughout each sprint and requirement.

Recovering the public key from the public address - The most challenging issue of the project was encountered during the implementation of the security system in sprint 3. The security system designed was intricate on itself, but materializing it to code proved a significant challenge for the developer, particularly the first step in the system. For the security system to succeed, asymmetrical encryption using the public keys of Ethereum accounts was critical, and it was discovered early on that users do not have direct access to their public key, but only to their public address. This issue was not foreseen since it was assumed that the public address and public key were interchangeable, or that if they weren't, users could find out what their public key was. At the start of the sprint, it was revealed that this was not the case, that the public address could not be used for asymmetrical encryption, and that users had no method of accessing and viewing their public key. To solve the issue, the Elliptic Curve Digital Signature Algorithm was analyzed, and a public key recovery algorithm was discovered. This algorithm was implemented correctly, and the public key was retrieved and successfully utilized for asymmetrical encryption.

Implementing AES encryption - The final challenge occurred during the implementation of the system's encryption schemes (AES, ECIES, Keccak256). More specifically, implementing AES to encrypt and decrypt images proved difficult. Although for strings and text files the encryption was straightforward, the encryption library had no documentation on image encryption. To address this issue, effort and time were invested in understanding the encoding standards and how files are converted from UTF-8 to an array buffer to an encrypted hex string and back to UTF-8 upon decryption.

10.2 Current status

Currently, the application allows users to connect their blockchain wallets and browse through, sample, and purchase microstock. These microstock are then delivered to the buyer and the funds are transferred to the seller. For storage, the system enables users to upload and fetch files from the IPFS. When a user lists a new product, they specify the title, price, and the encrypted photos. For encryption and decryption the system provides three encryption schemes. Advanced Encryption Standard (AES) for encryption of files, Elliptic Curve Integrated Encryption Scheme (ECIES) for asymmetrical encryption, and Secure Hash Algorithm in the form of Keccak256 for hashing. The current system enables users to exchange microstock in a transparent, inexpensive, and unbiased manner by combining encryption techniques and smart contracts deployed on the blockchain.

10.3 Future work

The current system meets the project's objectives and requirements, but there is still plenty of opportunity for future work. Because the project's foundation was meticulously constructed, the project can be built on and extended with ease. Some of the improvements and additions to the system are presented as the following:

Selected sample - Currently all microstock is encrypted before being uploaded to the IPFS. This ensures confidentiality of the files, as the IPFS is a peer-to-peer network, and files cannot be controlled where they are stored. The disadvantage with this is that users browsing to purchase products can only view the product name and must request a sample if they are interested. Although requesting a sample will not be removed, as it provides security as explained in section 7.1.1, allowing sellers to pick one file as their sample and submit it unencrypted would improve the buyer's user experience.

Music and videos - The current system enables users to securely trade microstock in the form of pictures. Music compositions and videos can also be microstock and support to include them in the system could be easily provided. To integrate this functionality, the security system would not need to be changed; instead, while uploading files, the appropriate check would be performed to discern between the different types of files. Finally, if the microstock being sold is one large file rather than a high quantity of small files, the large file would need to be partitioned into fragments and each fragment be uploaded separately.

Search & categorize - Currently, all products are displayed in a single screen, ordered from newest to oldest. A system enhancement to categorize products and provide a search tool to quickly locate what the user is looking for might be a beneficial addition to the system, further improving the user experience.

Deployment - The deployment of the system was the final element that was considered. The system is currently running on a local blockchain, but only minor adjustments are required for the system to be deployed to the Ethereum main network. Although deployment within the scope of the project was conceivable, due to the gas prices associated with the main network, a local blockchain was preferred.

10.4 Final words

A few closing remarks are offered to conclude the project. This project was both demanding and gratifying, requiring extensive study and critical thought to get to where it is now. The project provided an opportunity to develop a large scale project based on emerging technologies. Blockchain technology and Web 3.0 have significant future potential, and this project provided a glimpse of the possibilities and benefits of developing a truly decentralized application.

The skills and knowledge gained on Blockchain, Smart contracts, Solidity, IPFS, symmetrical and asymmetrical encryption, React, problem solving, and project planning will undoubtedly be beneficial in the future and professional career. With every aim and objective fulfilled, the project was major success, and the developer can be proud to have their name attached with it.

Bibliography

[1] GD Ragnedda and Giuseppe Destefanis. *Blockchain and Web 3.0.* London: Routledge, Taylor and Francis Group, 2019.

[2] Pinyaphat Tasatanattakool and Chian Techapanupreeda. Blockchain: Challenges and applications. In *2018 International Conference on Information Networking (ICOIN)*, pages 473–475, 2018.

[3] Dylan Yaga, Peter Mell, Nik Roby, and Karen Scarfone. Blockchain technology overview. Technical report, October 2018.

[4] Ahmed Afif Monrat, Olov Schelén, and Karl Andersson. A survey of blockchain from the perspectives of applications, challenges, and opportunities. *IEEE Access*, 7:117134–117151, 2019.

[5] Ravneet Kaur and Amandeep Kaur. Digital signature. In *2012 International Conference on Computing Sciences*, pages 295–301. IEEE, 2012.

[6] Simanta Shekhar Sarmah. Understanding blockchain technology. *Computer Science and Engineering*, 8(2):23–29, 2018.

[7] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. An overview of blockchain technology: Architecture, consensus, and future trends. 06 2017.

[8] Filipe Calvão. Crypto-miners: Digital labor and the power of blockchain technology. *Economic Anthropology*, 6(1):123–134, 2019.

[9] Johannes Göbel and Anthony E Krzesinski. Increased block size and bitcoin blockchain dynamics. In *2017 27th International Telecommunication Networks and Applications Conference (ITNAC)*, pages 1–6. IEEE, 2017.

[10] Shoji Kasahara and Jun Kawahara. Effect of bitcoin fee on transaction-confirmation process. *arXiv preprint arXiv:1604.00103*, 2016.

[11] Jiawen Kang, Zehui Xiong, Dusit Niyato, Ping Wang, Dongdong Ye, and Dong In Kim. Incentivizing consensus propagation in proof-of-stake based consortium blockchain networks. *IEEE Wireless Communications Letters*, 8(1):157–160, 2018.

[12] Po-Wei Chen, Bo-Sian Jiang, and Chia-Hui Wang. Blockchain-based payment collection supervision system using pervasive bitcoin digital wallet. In *2017 IEEE 13th international conference on wireless and mobile computing, networking and communications (WiMob)*, pages 139–146. IEEE, 2017.

[13] Michael Nofer, Peter Gomber, Oliver Hinz, and Dirk Schiereck. Blockchain. *Business & Information Systems Engineering*, 59(3):183–187, 2017.

[14] Damilare Peter Oyinloye, Je Sen Teh, Norziana Jamil, and Moatsum Alawida. Blockchain consensus: An overview of alternative protocols. *Symmetry*, 13(8), 2021.

[15] Leo Maxim Bach, Branko Mihaljevic, and Mario Zagar. Comparative analysis of blockchain consensus algorithms. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1545–1550. IEEE, 2018.

[16] Michael Crosby, Pradan Pattanayak, Sanjeev Verma, Vignesh Kalyanaraman, et al. Blockchain technology: Beyond bitcoin. *Applied Innovation*, 2(6-10):71, 2016.

[17] Sarwar Sayeed and Hector Marco-Gisbert. Assessing blockchain consensus and security mechanisms against the 51% attack. *Applied Sciences*, 9(9):1788, 2019.

[18] Du Mingxiao, Ma Xiaofeng, Zhang Zhe, Wang Xiangwei, and Chen Qijun. A review on consensus algorithm of blockchain. In *2017 IEEE international conference on systems, man, and cybernetics (SMC)*, pages 2567–2572. IEEE, 2017.

[19] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. Blockchain challenges and opportunities: A survey. *International Journal of Web and Grid Services*, 14(4):352–375, 2018.

[20] Nick Szabo. Smart contracts. 1994. *Virtual School*, 1994.

[21] Konstantinos Christidis and Michael Devetsikiotis. Blockchains and smart contracts for the internet of things. *Ieee Access*, 4:2292–2303, 2016.

[22] Georgios Papadodimas, Georgios Palaiokrasas, Antonios Litke, and Theodora Varvarigou. Implementation of smart contracts for blockchain based iot applications. In *2018 9th International Conference on the Network of the Future (NOF)*, pages 60–67, 2018.

[23] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[24] Sara Rouhani and Ralph Deters. Performance analysis of ethereum transactions in private blockchain. In *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 70–74, 2017.

[25] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[26] Iago S Ochôa, Rafael Piemontez, Lucas Martins, Valderi Reis Quietinho Leithardt, and Cesar Albenes Zeferino. Experimental analysis of the scalability of ethereum blockchain in a private network. In *Anais do II Workshop em Blockchain: Teoria, Tecnologia e Aplicações*. SBC, 2019.

[27] Daniel Sieradski. Gas and fees.

[28] Vitalik Buterin. Ethereum: Platform review. *Opportunities and Challenges for Private and Consortium Blockchains*, 2016.

[29] Preethi Kasireddy. How does ethereum work, anyway. *published on Medium (https://medium. com/@ preethikasireddy/how-does-ethereum-work-anyway-22d1df506369)*, 2017.

[30] Ferdinand Regner, Nils Urbach, and André Schweizer. Nfts in practice–non-fungible tokens as core component of a blockchain-based event ticketing application. 2019.

[31] Abram Brown. Beeple nft sells for $69.3 million, becoming most-expensive ever, Mar 2021.

[32] Christian Cachin et al. Architecture of the hyperledger blockchain fabric. In *Workshop on distributed cryptocurrencies and consensus ledgers*, volume 310. Chicago, IL, 2016.

[33] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[34] Anamika Chauhan, Om Prakash Malviya, Madhav Verma, and Tejinder Singh Mor. Blockchain and scalability. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 122–128. IEEE, 2018.

[35] Jorge Bernal Bernabe, Jose Luis Canovas, Jose L Hernandez-Ramos, Rafael Torres Moreno, and Antonio Skarmeta. Privacy-preserving solutions for blockchain: Review and challenges. *IEEE Access*, 7:164908–164940, 2019.

[36] A Begum, A Tareq, M Sultana, M Sohel, T Rahman, and AH Sarwar. Blockchain attacks analysis and a model to solve double spending attack. *International Journal of Machine Learning and Computing*, 10(2):352–357, 2020.

[37] Nicolas T Courtois. On the longest chain rule and programmed self-destruction of crypto currencies. *arXiv preprint arXiv:1405.0534*, 2014.

[38] Iuon-Chang Lin and Tzu-Chun Liao. A survey of blockchain security issues and challenges. *Int. J. Netw. Secur.*, 19(5):653–659, 2017.

[39] Joanna Moubarak, Eric Filiol, and Maroun Chamoun. On blockchain security and relevant attacks. In *2018 IEEE Middle East and North Africa Communications Conference (MENACOMM)*, pages 1–6. IEEE, 2018.

[40] Anam Bhatti, Hamza Akram, Hafiz Muhammad Basit, Ahmed Usman Khan, Syeda Mahwish Raza, and Muhammad Bilal Naqvi. E-commerce trends during covid-19 pandemic. *International Journal of Future Generation Communication and Networking*, 13(2):1449–1452, 2020.

[41] Petra Jílková and Petra Králová. Digital consumer behaviour and ecommerce trends during the covid-19 crisis. *International Advances in Economic Research*, 27(1):83–85, 2021.

[42] Stuart Feldman. Electronic marketplaces. *IEEE Internet Computing*, 4(4):93–95, 2000.

[43] Henry Chan, Raymond Lee, Tharam Dillon, and Elizabeth Chang. *E-commerce, fundamentals and applications.* John Wiley & Sons, 2007.

[44] Kazim Rifat Ozyilmaz, Mehmet Dogan, and Arda Yurdakul. Idmob: Iot data marketplace on blockchain. *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, 2018.

[45] Petter Gottschalk and Anne Foss Abrahamsen. Plans to utilize electronic marketplaces: the case of b2b procurement markets in norway. *Industrial Management & Data Systems*, 2002.

[46] Hemang Subramanian. Decentralized blockchain-based electronic marketplaces. *Communications of the ACM*, 61(1):78–84, 2017.

[47] Vishnu Prasad Ranganthan, Ram Dantu, Aditya Paul, Paula Mears, and Kirill Morozov. A decentralized marketplace application on the ethereum blockchain. *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*, 2018.

[48] Chris Morris. Nfts are now worth more than nissan and domino's pizza.

[49] Liam Kemp. How nft marketplaces make money, Sep 2021.

[50] James E Arps and Nicolas Christin. Open market or ghost town? the curious case of openbazaar. In *International Conference on Financial Cryptography and Data Security*, pages 561–577. Springer, 2020.

[51] Roger S Pressman. *Software engineering: a practitioner's approach.* Palgrave macmillan, 2005.

[52] Ruth Malan, Dana Bredemeyer, et al. Functional requirements and use cases. *Bredemeyer Consulting*, 2001.

[53] Martin Glinz. On non-functional requirements. In *15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 21–26, 2007.

[54] Chai Kar Yee and Mohamad Fadli Zolkipli. Review on confidentiality, integrity and availability in information security. *Journal of ICT in Education*, 8(2):34–42, 2021.

[55] Jakob Nielsen. *Usability engineering*. Morgan Kaufmann, 1994.

[56] Walt Scacchi. Process models in software engineering. *Encyclopedia of software engineering*, 2002.

[57] Adetokunbo AA Adenowo and Basirat A Adenowo. Software engineering methodologies: A review of the waterfall model and object-oriented approach. *International Journal of Scientific & Engineering Research*, 4(7):427–434, 2013.

[58] Basit Shahzad, Ihsan Ullah, and Naveed Khan. Software risk identification and mitigation in incremental model. In *2009 International Conference on Information and Multimedia Technology*, pages 366–370. IEEE, 2009.

[59] James Shore and Shane Warden. *The art of agile development*. " O'Reilly Media, Inc.", 2021.

[60] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. Agile software development methods: Review and analysis. *arXiv preprint arXiv:1709.08439*, 2017.

[61] Solidity documentation - v0.8.12, 2022.

[62] Ethereum development environment for professionals by nomic foundation, 2022.

[63] React – a javascript library for building user interfaces, 2022.

[64] Metamask - crypto wallet & gateway to blockchain apps, 2022.

[65] Ethers.js documentation, 2022.

[66] Chai.js documentation, 2022.

[67] Mui: The react component library documentation, 2022.

[68] James William Cooper. Java design patterns: a tutorial. 2000.

[69] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.

[70] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. 1999.

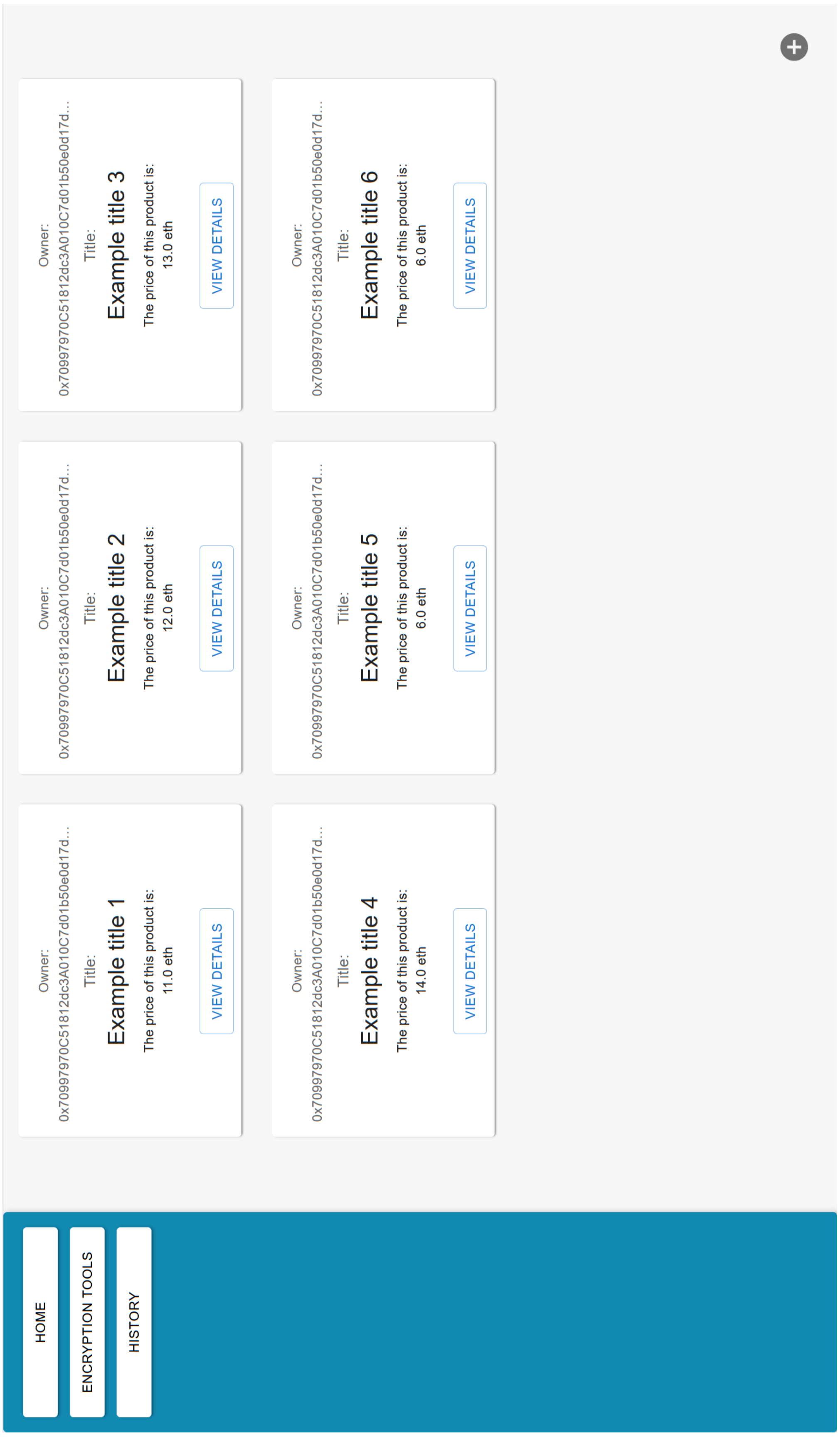
Appendices

Appendix A

User Interface

The user interface is Comprised of three pages:

- **Home Page**. This page displays the available products.
- **Encryption tools Page**. This page provides all the encryption tools to the users.
- **History page**. This page shows a table of all the transactions performed through the system.



Owner:
0x70997970C51812dc3A010C7d01b50e0d17d...
Title:
Example title 1
The price of this product is:
11.0 eth
VIEW DETAILS

Owner:
0x70997970C51812dc3A010C7d01b50e0d17d...
Title:
Example title 2
The price of this product is:
12.0 eth
VIEW DETAILS

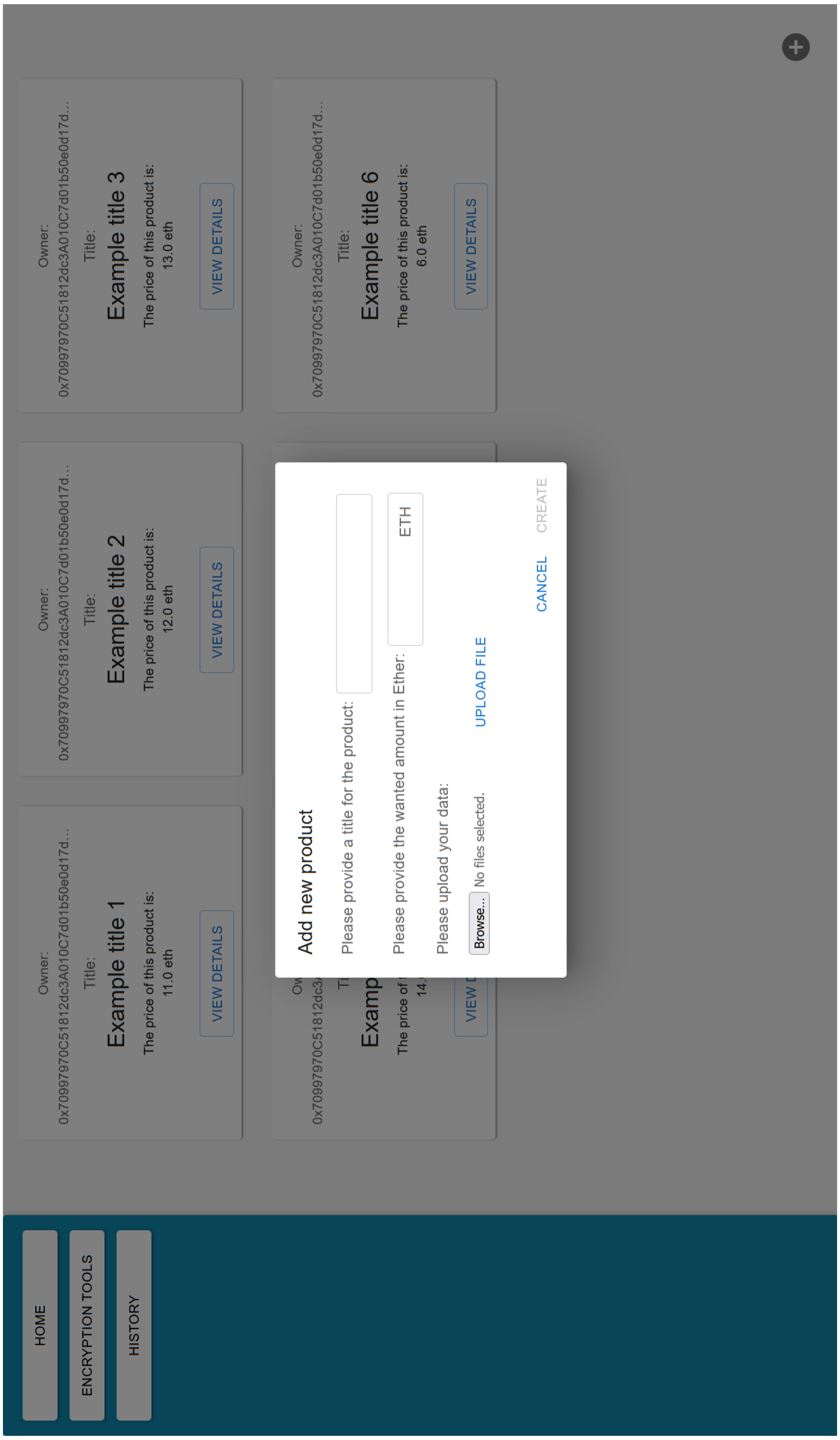
Owner:
0x70997970C51812dc3A010C7d01b50e0d17d...
Title:
Example title 3
The price of this product is:
13.0 eth
VIEW DETAILS

Owner:
0x70997970C51812dc3A010C7d01b50e0d17d...
Title:
Example title 4
The price of this product is:
14.0 eth
VIEW DETAILS

Owner:
0x70997970C51812dc3A010C7d01b50e0d17d...
Title:
Example title 5
The price of this product is:
6.0 eth
VIEW DETAILS

Owner:
0x70997970C51812dc3A010C7d01b50e0d17d...
Title:
Example title 6
The price of this product is:
6.0 eth
VIEW DETAILS

Figure A.1: Home page



Owner:
0x70997970C51812dc3A010C7d01b50e0d17d...
Title:
Example title 1
The price of this product is:
11.0 eth

Owner:
0x70997970C51812dc3A010C7d01b50e0d17d...
Title:
Example title 2
The price of this product is:
12.0 eth

Owner:
0x70997970C51812dc3A010C7d01b50e0d17d...
Title:
Example title 3
The price of this product is:
13.0 eth

Owner:
0x70997970C51812dc3A010C7d01b50e0d17d...
Title:
Example title 6
The price of this product is:
6.0 eth

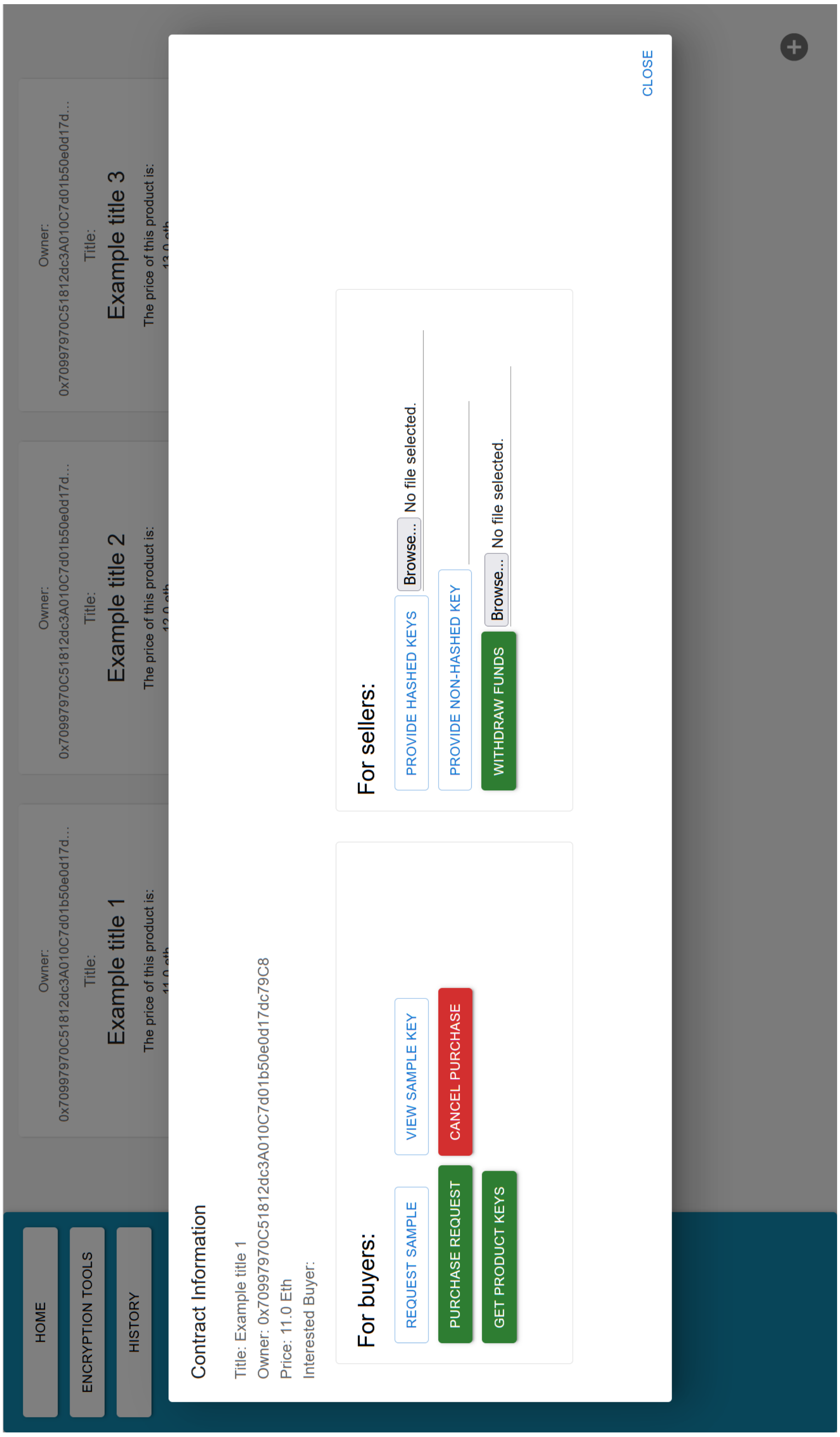
Add new product

Please provide a title for the product:

Please provide the wanted amount in Ether: ETH

Please upload your data:

Figure A.2: Adding new product view



Contract Information

Title: Example title 1
Owner: 0x70997970C51812dc3A010C7d01b50e0d17dc79C8
Price: 11.0 Eth
Interested Buyer:

Figure A.3: Details page of product

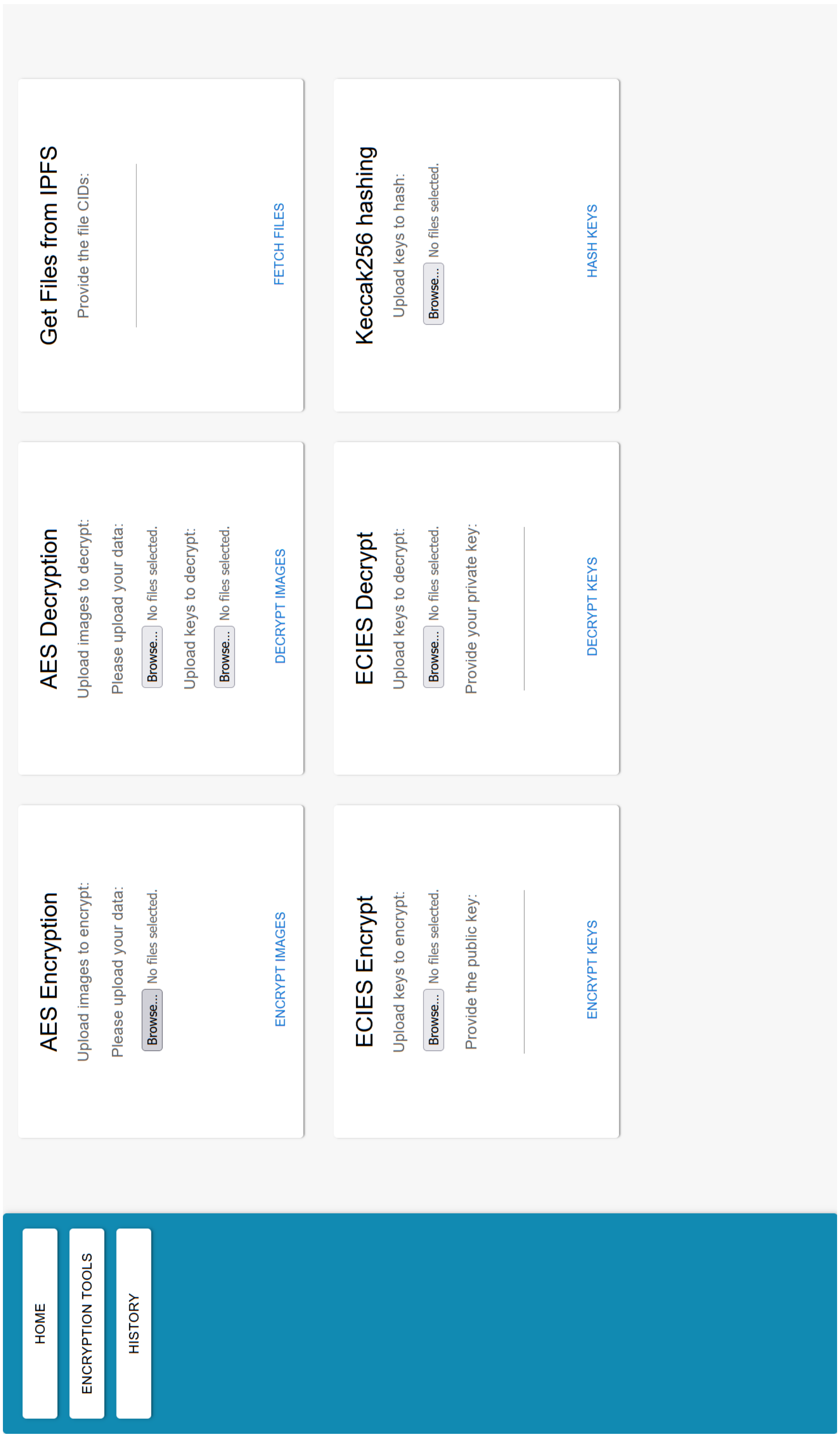
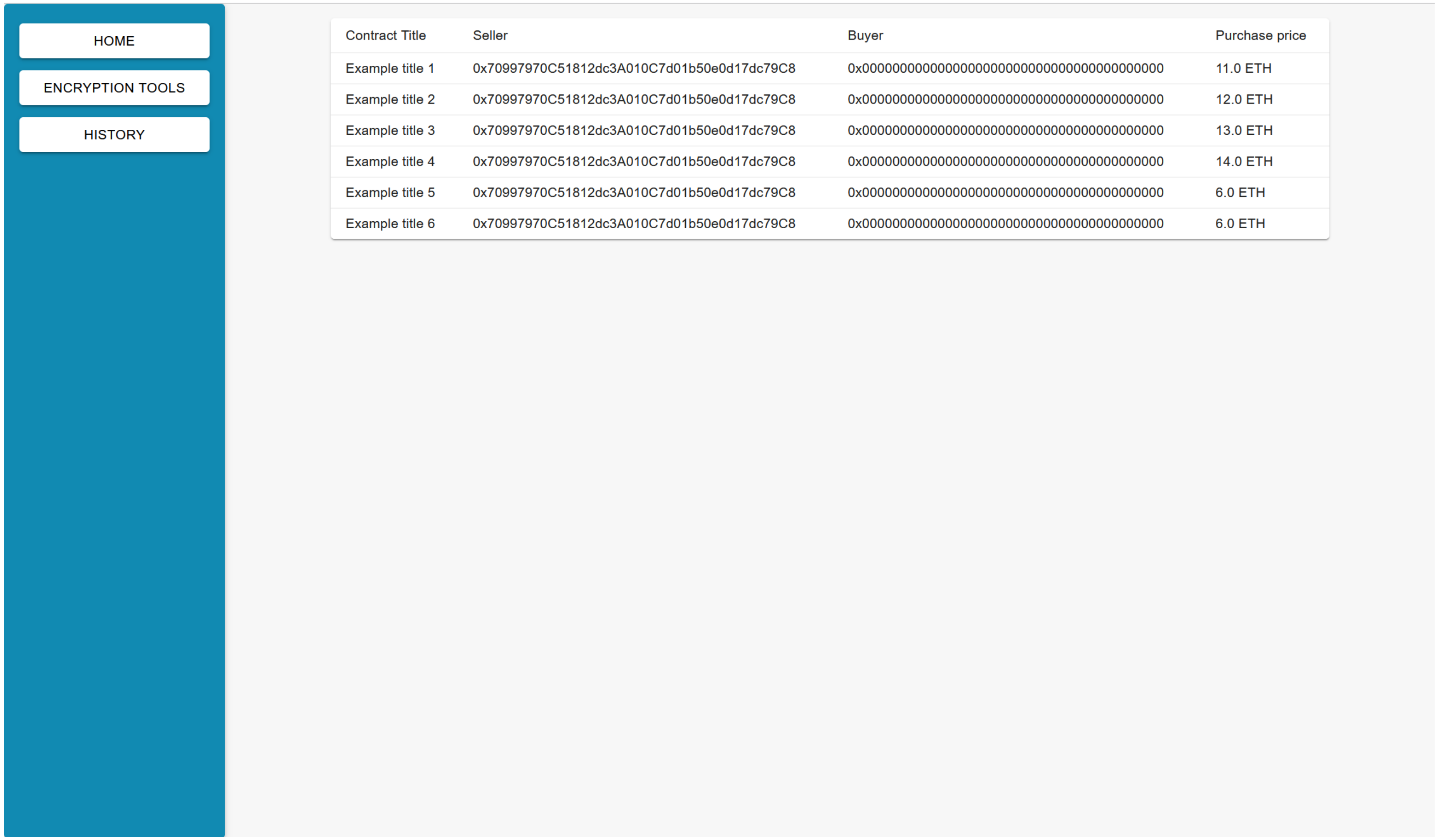


Figure A.4: Encryption tools page



Contract Title	Seller	Buyer	Purchase price
Example title 1	0x70997970C51812dc3A010C7d01b50e0d17dc79C8	0x0000000000000000000000000000000000000000	11.0 ETH
Example title 2	0x70997970C51812dc3A010C7d01b50e0d17dc79C8	0x0000000000000000000000000000000000000000	12.0 ETH
Example title 3	0x70997970C51812dc3A010C7d01b50e0d17dc79C8	0x0000000000000000000000000000000000000000	13.0 ETH
Example title 4	0x70997970C51812dc3A010C7d01b50e0d17dc79C8	0x0000000000000000000000000000000000000000	14.0 ETH
Example title 5	0x70997970C51812dc3A010C7d01b50e0d17dc79C8	0x0000000000000000000000000000000000000000	6.0 ETH
Example title 6	0x70997970C51812dc3A010C7d01b50e0d17dc79C8	0x0000000000000000000000000000000000000000	6.0 ETH

Figure A.5: Transaction history page