Andrine Celine Flatby
Ole Bjørn Gran
Marius Lillevik

Blockchain-based Ticketing

Bachelor's project in Bachelor of Science in Engineering - Computer Science
Supervisor: Hao Wang
May 2019

Bachelor's project

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Kunnskap for en bedre verden

Andrine Celine Flatby
Ole Bjørn Gran
Marius Lillevik

Blockchain-based Ticketing

Bachelor's project in Bachelor of Science in Engineering - Computer Science
Supervisor: Hao Wang
May 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Kunnskap for en bedre verden

Sammendrag av Bacheloroppgaven

Tittel:	**Billettsystem på Blockchain**
Dato:	May 19, 2019
Deltakere:	Andrine Celine Flatby Ole Bjørn Gran Marius Lillevik
Veiledere:	Hao Wang
Oppdragsgiver:	Innit AS
Kontaktperson:	Hao Wang, hawa@ntnu.no, +47 70 16 15 34
Nøkkelord:	Blockchain, Billettsystem, Ethereum, Hyperledger Fabric, Solidity, JavaScript
Antall sider:	43
Antall vedlegg:	6
Tilgjengelighet:	Åpen

Sammendrag:

Innit AS har flere lisensierte applikasjoner for billettsalg for ishockey lag i den norske hockey ligaen. De forbedrer kontinuerlig sine billettsystemer, og de vil se om de kan forbedre sin eksisterende løsning ved hjelp av blockchain-teknologi.
Blockchain er en relativt ny teknologi med mange bruksområder. Det gjør det mulig for medlemmer av et nettverk å dele informasjon sikkert og har forbedret loggegenskaper sammenlignet med tradisjonelle databaser.

Denne oppgaven undersøker bruken av blockchain i billett systemer. Først presenteres en bakgrunnsstudie av blockchain-teknologien og dens utviklingsplattformer. Deretter en undersøkelse av dagens marked og eksisterende løsninger bygget på blockchain. Basert på disse funnene ble en prototype av et billettsystem utviklet som et "proof of concept" for å fremheve mulighetene for disse systemene.

Summary of Graduate Project

Title:	**Blockchain-based Ticketing**
Date:	May 19, 2019
Authors:	Andrine Celine Flatby Ole Bjørn Gran Marius Lillevik
Supervisor:	Hao Wang
Employer:	Innit AS
Contact Person:	Hao Wang, hawa@ntnu.no, +47 70 16 15 34
Keywords:	Blockchain, Ticket systems, Ethereum, Hyperledger Fabric, Solidity, JavaScript
Pages:	43
Attachments:	6
Availability:	Open

Abstract:

Innit AS has multiple licensed applications for ticket sales for ice hockey teams in the Norwegian hockey league. They are continuously improving their ticketing systems, and they want to see if they could improve their existing solution by using blockchain technology.
Blockchain is a relatively new technology with a wide variety of applications. It allows members of a network to share information securely and has improved logging capabilities compared to traditional databases.

This thesis investigates the applications of blockchain in ticketing systems. First, a background study of the blockchain technology and its development platforms are presented. Then an investigation into the current market and the existing solutions using blockchain. Based on these findings, a prototype ticketing system was developed as a proof-of-concept to showcase the feasibility of these systems.

Preface

We want to thank Øyvind Tangen and Olafur Trollebø at Innit for the interesting bachelor thesis and for their continuous support during this period.
We also want to thank our supervisor Hao Wang for his input on the report; he helped structure and proofread our report from day one.
We also want to thank Mariusz Nowostawski for his help during the development of our prototype. He helped us with most of the questions we had about developing the prototype on a blockchain.

Contents

List of Figures

List of Tables

Listings

Abbreviations

P2P, peer-to-peer

DLT, distributed ledger technology

Geth, Go-Ethereum

PoW, Proof of Work

PoS, Proof of Stake

PBFT, Practical Byzantine Faul Tolerance

VM, Virtual Machine

EVM, Ethereum Virtual Machine

Dapp, Decentralized application

DAO, Decentralized Autonomous Organizations

MVP, Minimum Viable Product

1 Introduction

Throughout the last decades, it has become more and more common to attend events such as music concerts, festivals and sporting events [1]. Moreover, in recent years the number and variety of events have grown tremendously, this has prompted the concept of tickets to become ingrained in our modern society. However, despite tickets prevalence in our modern lives, there have been relatively few technological advancements related to them. In recent years tickets have been made digital, but that has been more related to the digitization of paper. When someone purchases a ticket digitally, they might for example receive a PDF of the ticket in a mail. Hence, a digital ticket is not much different from a paper one. There is room for improvement in the ticket industry and blockchain could be a way forward.

1.1 Background

Innit AS is a technology company located in the city of Hamar, Norway and functions as the parent corporation for two child entities; Innit Utvikling AS and Innit Drift AS. Innit Utvikling AS is a software development company with a wide range of products and services, ranging from out-of-house client projects to in-house products. One of Innit's core products is a ticketing system used by several ice hockey clubs in Norway. Innit has over two years of ticketing experience and on a theoretical level the blockchain technology seems to have several properties that are potentially beneficial to a ticketing system, such as traceability of transactions.

1.2 Problem area

In Innit's existing system they sometimes experience a glitch, where a ticket will get falsely marked as spent or invalid. This can cause confusion, not only for the person whose ticket was randomly invalidated but also for Innit since they have yet to find the cause of this issue. Blockchain should prevent issues like this, and this is one of the reasons why Innit is interested in this technology. Innit is also interested in the logging capabilities of blockchain, since it can make management and troubleshooting easier.

1.3 Project description

The initial version of the project, as described to us, were to develop a prototype ticketing system based on blockchain technology. Where a web-based administration panel, accompanying mobile applications; one for purchasing and one for validation of the tickets would be developed. Additionally, a report discussing the pros, cons, and viability of developing a blockchain-based ticketing system was to be provided. However, during the planning phase, we agreed to shift the focus of the project and make it mainly about the report. Innit has no plans of actually using any part of the prototype for future work, they were only interested in learning about the potential of blockchain. They still wanted a prototype, but mainly as a proof of concept, so the requirements list were cut down, and the development phase delayed until after we finished the main bulk of the report.
For the report to make a compelling case, either for or against blockchain-based ticketing, we first need to build an understanding of what a blockchain is, how it works and what features are useful for this scenario. We then need to look at the market, are there already existing systems based on blockchain out there and is the market ready for these types of solutions. Once all this is clear, we will feel comfortable concluding whether or not blockchain is a technology worth pursuing. Various development platforms will be compared with each other in order to find the most suitable for our use-case and our employer's needs. After finishing the central part of the report, we shift our attention to the prototype. However, the prototype now being so simplified will only serve as a simulation of an actual system and serves only to demonstrate the core functionalities of a real system.

1.4 Delimitations

For the report, the only delimitation is that we are only focusing on blockchain's potential for ticket systems. However, for the prototype, there are multiple delimitations; in general, the prototype will be entirely separate from any existing system, so there is no compatibility issues to be had. We are only testing for one event, therefore only one set of tickets. Finally, the system only needs the core functionality of a ticket system and only needs to support the purchase and validation of a ticket.

1.5 Target audience

The report is meant to inform our employer, about blockchain technology and its potential use in the ticketing industry. The report is also targeting the academic community here at NTNU and others with similar technical knowledge. We decided to write the report in English with respect to a multilingual audience. Our supervisor being a non-Norwegian speaker made this decision easy.

1.6 Academic background

We are three students that have studied computer engineering at NTNU in Gjøvik, and during our three years, we have gone through mostly the same courses. We all have experience with C++, Java and general system development, but neither of us has any experience with blockchain development. This lack of knowledge meant that in order to develop a prototype we had to dedicate a portion of the project period to learning the basics.

1.7 Project organization

Since the project had distinct phases, how we worked changed throughout the process, and although we had assigned roles in our pre-project (see appendix E) these were not set and did change throughout the project. In the beginning, there was a lot of research to be done. The lack of proper working space at the school meant that we spent a good portion of the time working from home, but still communicating digitally. We had regular meetings at school to discuss and compare what each member had done since the last meeting and to plan for what to do next. During the last weeks of the project, all days were spent together at school; this was to discuss changes and make decisions as a group more efficiently. As we were also still working on the prototype, sitting together when developing helped to speed up the progress.
We had regular meetings with our supervisor, and continues contact through email. We also talked with our employer as much as possible, these meetings were mostly to inform them about our progress with the research, with us also sending them weekly drafts of the report.
During the development phase, we started working with Marius Nowostawski, a teacher at NTNU, since he had previous experience with blockchain development. He was able to help us troubleshot more efficiently and was extremely helpful throughout the development phase.

1.8 Document structure

This report is written in LaTeX on `overleaf.com` with a template provided by NTNU. We document all sources in Vancouver style at the end of the report. Afterwards are our various attachments; Definitions A, Source Code B, Installation guide C, Meeting logs D, Project Plan E and Project Agreement F.
This report is divided into seven main chapters:

1. **Introduction** 1: Contains an introduction to the thesis and the report, information about the authors and their academic background, organization of the project, and finally, the structure of the report.
2. **Blockchain** 2: Here we analyze blockchain technology, its history, functionalities and use cases.
3. **Existing Services** 3: An exploration of existing ticket services based on blockchain and how they benefit from using it.
4. **Platforms and Protocols** 4: A comparison and discussion of what platforms and protocols are best suited for our use-case.
5. **Design** 5: The requirements and technical design for the prototype.
6. **Implementation** 6: Contains the development process and a showcase of the finished prototype.
7. **Conclusion** 8: Conclusion about the project and our findings.
8. **Bibliography**: A list of all sources.

2 Blockchain

Towards the end of 2017, Bitcoin got extremely popular, peeking December 17th 2017, with the price of one single bitcoin at $19,783.06 [2]. With the popularity of bitcoin so high many actors started gaining interest in the technology behind it and its many potential use cases. Despite the popularity of blockchain increasing, there are still many within the technology industry that do not know how blockchains work.

2.1 What is blockchain?

In the simplest terms, blockchain is a growing list of records called blocks that get generated and linked using cryptographic hash functions [3]. Each new block gets generated by hashing the current newest block in the chain and adding that hash as a header in the proposed newest block. For this new block and its changes to be approved, a mathematical puzzle involving the hash, has to be solved. Across the network, so-called "miners" attempt to solve this puzzle. When a miner eventually comes up with a solution all the nodes on the network has to check and confirm the change before updating the blockchain [4].

The first blockchain was created in 1991, a Merkle Tree was used to create a “Secured chain of blocks”, where each block contains data, a timestamp and a cryptographic hash of the previous block. This is the “chain” in Blockchain [5]. A Merkle tree is a data structure where each non-leafe node is a hash of its child nodes, where the leaf nodes are the lowest tier of nodes [6].

Late in 2008, an anonymous programmer or a group called Satoshi conceptualized the design of the first blockchain [5]. Then in early 2009, Satoshi created and published the distributed blockchain to be used as a public transaction ledger for a cryptocurrency called Bitcoin [7]. This blockchain would contain a secure history of data exchanges, timestamps and each exchange is verified by utilizing a peer-to-peer network. All of this would be done without a central figure of authority, autonomously.
Later that same year, Satoshi Nakamoto released the open-source, decentralized cryptocurrency, Bitcoin, the first cryptocurrency [7]. Since the release of Bitcoin, there have been many so-called Altcoins, different cryptocurrencies with many different names, but Bitcoin is still the most valuable and popular.
A cryptocurrency is a digital asset or virtual currency designed to be used for exchanges in a blockchain network [7]. Cryptography is used to secure and verify transactions, and control the creation of new units of a cryptocurrency.

Cryptography in blockchain technology is used for transactions, wallets, privacy and security [8]. Blockchain uses a Public-key cryptography system. This system uses a pair of keys for interactions between users in the chain, a public key, and a private key. For example, if Kimo wants to send Missy a message, he would need Missy’s public key to encrypt the message and send it to her. Missy would then get an encrypted message which she would be able to decrypt with her private key and then read the message.

Cryptographic hashing is another fundamental part of blockchain technology [8]. Hashing takes a string as an input, for example, a password of any length, and turns it into a fix length output. A cryptographic hash function has three properties [8]:

- Deterministic: A specific input will always have the same output.
- Irreversible: The output cannot be used to determine input.
- Collision resistance: Two different inputs will always have different outputs.

2.1.1 Decentralization

All blockchain networks are decentralized; this means no single entity owns the blockchain [3]. A decentralized network of many nodes is similar to a supercomputer, but instead of one computer doing all the computing, each computer on the network do a part of a given task. This method is both faster and cheaper than a supercomputer. A User can submit a task, like big data analysis, to the network, and the task will then be divided, processed and reassembled after all the computers are done with their given task [9].

2.1.2 Different types

There are three different types of blockchain networks: Public, Private, and Federated/-Consortium. All these types originated from the Bitcoin blockchain when people realized that it could be used for any value transaction or agreement. Private institutions started using the core idea of blockchain as a distributed ledger, which spawned the creation of permissioned, private and federated blockchains [10].

Public Blockchain

The most common type of blockchain is a public network, which is based on a Proof of Work (i.e.,PoW) consensus algorithm. The Public blockchain is open source and not permissioned, meaning anyone can download the code and run a public node on their local computer. Users can make, validate, and see transactions with the public block explorer [10]. Every transaction is transparent, but every user is anonymous. Examples of public blockchains are Bitcoin[11] and Ethereum[12].

Private Blockchain

In a private blockchain, write permissions are kept locally within an organization, while read permissions may be public or restricted. Groups take advantage of blockchain technology by using it internally within, i.e. a company and use it to verify transactions. Private blockchains are faster and more scalable [10]. Examples of private blockchains are MONAX[13] and Multichain[14].

Federated/Consortium Blockchain

A Federated blockchain is controlled by a group of people. While public blockchains allow anyone with an internet connection to participate, Federated blockchains allow only certain people to participate. These blockchains are faster and more private than public blockchains [10]. A Consortium blockchain is used most often by banks. The consensus process is controlled by a selected number of nodes, where a number of financial institutions must sign every block in order for the block to be valid [10]. An example of this type of blockchain is R3 Corda [15].

2.1.3 Frameworks and platforms

Framework and platforms are generally loosely defined in the blockchain community [16]. Since they are often used interchangeably, it can lead to confusion about what a framework and a platform is. In this bachelor thesis, we define them as: frameworks are tools necessary to develop and deploy smart contracts, and platforms are where the smart contracts get deployed. An example of a use of our definition: Ethereum is a platform and Truffle is a framework [16].

Two of the most popular blockchain platforms currently are Ethereum and Hyperledger Fabric [17]. These are examples of a public blockchain and an enterprise blockchain platform respectively. Also, an example of a popular framework is Embark, which is used on Ethereum [16].

2.2 How it works

To explain how blockchain works it is essential to know that it is a peer-to-peer (i.e.,P2P) system and distributed ledger [3]. It allows participants to use a service, for example; an online store, without involving a third-party, like a broker or a bank. It can reduce cost and increases trust between participants. The increase of trust comes from it being a decentralized, (see section 2.1.1), transparent and tamper-proof database. This means that information is stored across all participants instead of in a single database. All participants in a blockchain can see, change and take part in the decision of whether or not to approve changes to the ledger, but ultimatly, it is down to the decision of the majority.

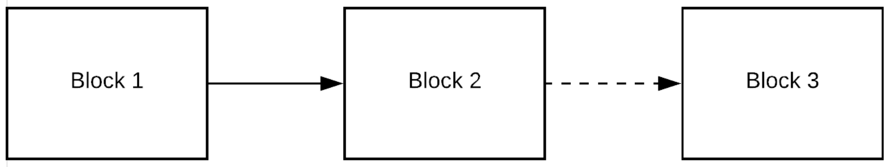


Figure 1: Example of how blockchain works

Figure 1 show a simplified illustration of how blocks and chains works. The first block in the chain is the Genesis block, where gas limit, chain id, and more is determined. When creating data, a second block containing the data is mined and added to the chain. This process repeats for every additional block mined. These blocks are now holders of the data in the system and cannot be changed. If a user wants to change the data, it will create a new block that now holds the updated data.

An example of a simple transaction is, Missy wants to send money to Kimo. The transaction contains the address of the sender (Missy) and receiver (Kimo), the amount of money to be traded and a unique hash. This data forms a block that will be added to the chain if the majority approve it. Since blockchain is unchangeable, participants cannot change the data of a block, so if Missy or Kimo wants to change the transaction, a new block will be created with the proposed changes.

2.2.1 Distributed ledger

The distributed ledger technology (i.e, DLT) is a vital component in the blockchain. It provides a shared database where participants store their identical copy of the ledger [18][19]. Unlike a traditional database, the ledger is stored across the participants, because of this, blockchain archives are decentralized. One of the benefits of DLT is that without a centralized authority, the level of trust between participants is higher. It is also crucial to understand that a blockchain ledger is a special kind of distributed ledger. The difference is that a blockchain DLT stores the data in blocks that only exists on the blockchain.

The append-only structure of blockchain means that every addition to the ledger is permanent [20]. Entries cannot be removed or altered; all changes are stored in history logs, which makes managing and tracking records easy. These features make the ledger tamper-proof, since nothing can be changed without other participants knowing.

An example of how the distributed ledger works; Missy, Kimo, and Peneline are participants on a blockchain network and each have a copy of the distributed ledger. The transaction for Missy's purchase of Kimo's bike needs to first be verified by Peneline before it can be added to the distributed ledger. When the process is completed, the blockchain network updates each participant's copy permanently with the new block.

2.2.2 Smart contract

A smart contract is like a function in standard programming that only executes when the system meets a predetermined set of conditions [21]. Since it is just a simple computer code, it is crucial to get the logic right for correct operation. Smart contracts are implemented into distributed ledgers to ensure correct operation [3], and mainly operate with an "if this, then that" methodology.

Smart contracts are a transparent, conflict-free way to handle the purchasing of services without third-party intervention [22]. Because of this, there is no need for a lawyer or bank. The benefits of this: it amplifies the trust between the buyer and seller because blockchain is transparent, and can reduce the cost of a transaction [23].

An example of how a smart contract with two participants work: Kimo wants to sell his bike, and Missy wants to buy it[23]. Missy purchases the bike by signing the contract. This is done by using her private key, but the exchange of money happens between her and Kimo's blockchain addresses. Then the smart contract is overseen by the participants in the blockchain network. This system reduces the possibilities for scams, because if the seller or buyer does not meet requirements of the contract, the exchange is not approved. In this scenario, the smart contract is approved, Missy has transferred money to Kimo and has access to the bike's blockchain address.

2.2.3 Consensus protocol vs algorithms

The term consensus is defined as; nodes on a network agreeing on the condition on a block [24]. The terms consensus protocol and consensus algorithm are often used interchangeably, but they are not the same [25]. The simplest explanation of the terms: the protocol provides a set of rules the blockchain follows, and the algorithm determines how the protocol follows these rules.

Consensus protocol keeps the participants synchronized on the network [24]. Participants do not need to trust each other when entering an agreement because they just followed the rules provided by consensus protocol. The rules ensures that blocks don't break protocol when validating them.

Consensus algorithms are a method to make decisions within a group and are used to create equality and fairness in the network [26]. It is essential to know that consensus algorithms do not necessarily only agree with the majority, but also agree with what benefits everyone. The blockchain in itself does not provide a decentralized environment, but consensus algorithms make the system decentralized. Blockchain consensus models are the primary way for participants in a blockchain network to reach agreements. There are many different types of consensus algorithms: PoW, Proof-of-Stake (i.e., PoS) and Practical Byzantine Faul Tolerance (i.e., PBFT).

2.3 Why blockchain in ticketing system?

Blockchain technology can contribute to a safer ticketing system by addressing the two most prominent issues in today's systems; ticket fraud and resale in the secondary market [27]. These issues plague the ticketing market since it reduces the level of trust a consumer has in the system.

2.3.1 Benefits

Blockchains keep a record of every transaction in the system; this makes it easy to identify resellers and, manage tickets and events [27]. Event organizers can benefit from this because it can help increase revenue by removing the need for a broker or bank, by using cryptocurrency. It can also benefit consumers by lowering fees and help mitigate ticket fraud[28].

Cryptography make tickets more secure by having them contain a unique hash, which cannot be changed. By using cryptography tickets become nearly impossible to replicate or double-sell, which leads to a safer buying experience for consumers [28]. Identity management can also help prevent attempts to sell event tickets outside of the blockchain.

A ticket can be programmed with a smart contract to enable specific rules and functions [28]. An example of this would be to enable profit sharing on resales, this makes sure that event organizers do not lose revenue from resellers in the secondary market [29]. It can also link the consumer's blockchain address to the tickets which provides a way to identify the ticket owner.

2.3.2 Challenges

Blockchain technology has the potential to solve many of the current problems in today's ticketing systems, but there are some challenges. Blockchains do not always scale well; it will work great for some users, but at the scale that most ticketing services operate there could be issues. When there are multiple users on a blockchain network, transactions can take a long time to process. This issue can already be seen with Ethereum and Bitcoin, where they are already having trouble solving this [30].

Privacy is another issue because most blockchains use a public ledger structure [30]. Privacy is a necessity in ticketing systems because they contain personal information about users, like; credit card, name, ticket information and more. There are solutions to this problem, but it is dependent on what blockchain platform the system is running on.

Another challenge for blockchain is the lack of awareness about the technology [30]. If the blockchain community wants more industries to adopt blockchain technology, it needs more attention and acceptance from developers and consumers. Otherwise, a user might not trust a system based on this technology, so for blockchain to become more widespread, awareness needs to be raised to inform the average consumer.

3 Existing services

Over the years the ticketing industry have been trying to optimize their systems for efficiency and security. The future of the ticket industry is headed towards an all digital world [31]. Blockchain technology could provide the ticket industry with new ways to adapt, and the use of smart contracts and other blockchain features could be the way. There are multiple ways of using blockchain technology in this area, where the two main approaches are stand-alone services and system protocols.

3.1 Services

Multiple blockchain-based ticketing services have started to appear lately. [32]. These services focuses on being user-friendly both for event organizers and consumers, and all have the goal of being simple and easy to use. These services, like most blockchain-based systems, relay on smart contracts for their operations [31]. The benefits of smart contracts are discussed in more details in section 2.3.1. Additionally, these services do not need an intermediary broker or bank, since it is a decentralized system, see section 2.1.1 for more.
Although there are multiple services currently in development, this thesis will mainly focus on two: Evopass [33] and Blocktix [29].

3.1.1 Evopass

Evopass was a ticketing service system that started in 2016 but shut down and ending all operations in September 2018 [34]. Their application was designed to allow users to buy and resell concert tickets. Evopass initially built their system with blockchain technology, but after latency difficulties when scanning tickets; they concluded that ticketing services were not a good use-case for blockchain technology [35]. On September 7, 2018, they made the decision to shut down Evopass, since they were a small team and could not compete against bigger companies in a growing market [36].

3.1.2 Blocktix

Blocktix started development in late 2016 to showcase the benefits of using blockchain technology to develop a ticketing service [37]. The company aims to a decentralized ticketing service as a counterfeit-resistant solution. By removing the infrastructure of centralized servers, blocktix has fulfilled its goal to allows all parties to experience no downtime and low fees [38].

It has fulfilled the solution by running their smart ticket on the Ethereum [29]. The tickets are linked together and, its including cryptographic signatures and uniquely attributed to preventing fraud and scalping [38]. Fraud is an obvious challenge to prevent, while scalping is a trading strategy where trader profits off small price changes of stock.

The event organizer can use the benefits of the smart ticket by adding additional conditions to their tickets. For example, include giving a predefined percentage of the sales income, to artists or event organizers [29]. The ticket is stored securely and using blocktix 's facial recognition for ID verification.

3.1.3 GUTS

GUTS is a PoW ticket selling application developed in the Netherlands and is the first customer of the GET protocol [39]. GUTS smart tickets are sold via a blockchain network at a fixed price in an attempt to make fraud and profiting impossible by linking each ticket to a user via their phone number. The user can then either spend the ticket on the event it is intended for, or sell it on to another user anonymously via the blockchain at the same fixed price it was bought at. Another feature that makes profiting impossible is the fact that the tickets use a non-fixed QR-code, which updates frequently, so a user cannot send a screenshot of the QR-code and sell it on to someone for a profit [39]. GUTS finally released on ios and android in 2018 after selling over 18,000 tickets on the GET testnet [40].

3.2 Protocols

To be able to offer these types of services on a blockchain, a developer needs a set of rules that allow for the correct operation of the application. In the case of blockchain applications, these rules come in the form of smart contracts and are the core of all functionality in these applications. For a ticket system to function correctly, a developer needs to make sure that they have smart contracts that handle all possible scenarios. This is where protocol providers come in. A protocol is a set of smart contracts designed for a specific use-case. A developer can take a protocol and use it as a base for their application without worrying if they have thought of everything.
For this thesis we have chosen to focus on two protocol providers that are relevant for our use case, Aventus [41] and GET [42].

3.2.1 Aventus

The Aventus protocol started its jurney in late 2017 and is currently in the early stages of development. It is a protocol specially designed to handle the requirements of an online ticketing system. Aventus wants their protocol to help lower the barrier to entry for developers in the ticketing industry. The protocol itself is based entirely on ethereum smart contracts and handles all the ins and outs of a ticketing system. All from the creation of a ticket to the distribution of revenue between all involved parties. This complete control of the lifecycle of a ticket allows a fairer and safer system [43].

3.2.2 GET

The GET protocol is explicitly made for developing ticket selling applications using blockchain technology and is built upon the Ethereum blockchain [39]. GET offers an overview of all tickets from the first ticket is sold until the last ticket is validated at any event. Every smart ticket is unique, traceable on the blockchain and has its properties locked. The protocol uses a cryptocurrency token, GET or Guaranteed Entrance Token as it stands for, which acts as the main asset, holding a stable value. This is necessary for the ticket platform since the tokens can be used to lock the value of a ticket, which adds security and price stability to each event-cycle [39].

3.3 Their solutions

All services mentioned in this chapter have one common goal; make today's systems safer and more efficient for consumers and event organizers. Protocols like Aventus and GET provide tools for developers to more easily create their applications without facing all the challenges with creating these systems. They address these challenges by using the benefits of blockchain (see 2.3.1), with, for example, smart contracts which can be integrated into a ticket element, and is often referred to as a smart ticket [39][43]. Solutions like these are meant to protect consumers and event organizers when they use these ticket services.

3.3.1 Platforms

All services mentioned so far have all developed their solutions using the Ethereum blockchain. Ethereum seems to be the platform of choice for most when developing applications with blockchain; this might be because of Ethereum's focus on decentralized application (i.e., dapps), which we talk more about in section 4.1.1.

3.3.2 Smart tickets

A smart ticket is a programmable ticket that is transferred via a blockchain [29]. Smart tickets are programmed to, for example, have a fixed price which cannot be altered, meaning that there is no profit in the resale of a ticket. By having all tickets linked together in the blockchain, they cannot be duplicated since there is only one copy of each ticket during a transaction. All transactions are done P2P without any third party interference, which makes it easy to resell your ticket to another person, for the same price you bought it for.

Depending on what the event organizer sees fit for their event, the smart tickets can be customized. An example of this is Blocktix, who has personal information connected with each ticket, see more in 3.1.2.

3.3.3 Security

When it comes to security, these solutions have some different security measures to prevent certain misuses or theft. Event organizers that use Blocktix to distribute tickets, stores the identities of attendees in the tickets. This information is stored and secured by using facial recognition for ID verification (See section 3.1.2).
GUTS uses a non-static QR-code, which prevents users from selling their ticket by sending a screenshot of the QR-code and charge a higher price. People can still sell their tickets with GUTS, but the price is fixed, making it impossible to scam other users (see section 3.1.3).
Since these solutions are developed on blockchain, they also inherit some of their core security measures, like cryptography (see section 2.1). This increases security, since every transaction is encrypted.

3.4 Market challenges

The most significant threat to the ticket selling industry is the secondary market [27]. Because of the third-party resale and fraud, the secondary market is a source of lost income for event organizers and service providers.

Before discussing the secondary market, it is essential to know about the primary market. The primary market is mainly the event organizers and their partners, who sell tickets to their events via ticket services, like Ticketmaster [44]. Event organizers often have a difficult time accurately pricing the events, which can lead to underpricing [45]. Another big challenge for the primary market is the use of bots to purchase a large number of tickets to be resold at a higher price in the secondary market. This reduces the number of tickets available legitimately and forces consumers to purchase from third-party sellers at a higher price [39]. A challenge for blockchain-based ticket services in the primary market can be convincing event organizers to switch to their services [39]. The relationship between old ticketing services and event organizers can make it difficult for blockchain-based ticketing services to gain traction.

The secondary market is fueled by supply and demand, which the primary market cannot fulfilled [28][45]. For example, scammers resell tickets on sites like Facebook for a higher price and in some cases list the same ticket on multiples sites. Scammers are a prominent issue in the secondary market because they take advantage of the lack of transparency in pricing and the limited amount of tickets. In Norway, resale of tickets at a higher price is illegal, scammers are committing a crime in terms of the law [46]. According to GUTS old ticket services, like Ticketmaster, is partially to blame for the issues surrounding the secondary market [39]. The reason is that they have stakeholders in secondary market.

If these challenges are not addressed, it could lead to the downfall of ticket services. An example of this scenario is Evopass who got ousted of the market (see 3.1.1). Evopass had problems when it came to the secondary market, which was one of the main reasons for them shutting down all development. The secondary market has grown to become stiff competition, which is dangerous for the primary market. It is worth noting that Evopass allowed resale of tickets, which made them direct competitors with the secondary market. Since Evopass were early adopters of blockchain technology in this area; they also had some issues optimizing their system [47].

Solving these challenges is something the industry is still struggling with. Eliminating the secondary market could solve some of the challenges, if it was possible, but a rational macroeconomic perspective would conclude it is a bad idea [28]. The secondary market is only getting stronger, and the ticket industry need to adapt and use the secondary market to their advantage. As mentioned in section 3.1.2, Blocktix uses smart tickets in an attempt to control the secondary market [29]. When a reseller sells a ticket, it can be programmed to make a certain percentage of the profits go to the artists and event organizers. By using functionality like this, the ticket industry can turn a disadvantage to an advantage.

A blockchain-based ticketing service that has used this technology to solve many of the market challenges and provide a compelling alternative, is GUTS (see 3.2.2). GUTS designed their system to eliminate fraud in the secondary market by using smart tickets and locking the price, which prevents third-parties from profiting on resale [39]. A consumer can still sell their ticket, but it is impossible to charge a higher price (see 3.2.2).

GUTS have been able to convince event organizers to use their services instead of more traditional ones, because of their competitive pricing.

Blockchain can solve many of the challenges ticket services faces in today's market, but it is still trying to build a reputation among event organizers and consumers. Most blockchain-based ticket services are either in alpha or beta, like Aventus (see 3.2.1). The fact that most services are still in development means there are not many usable alternatives to traditional ticket services available today, but GUTS and Blocktix show that blockchain has the potential to change the industry.

4 Platforms and Protocols

Developers first need to consider which platform to use before they can start developing blockchain-based applications. Different platforms have different features and tools, making this decision important for developers. The application's use-case is the most significant factor when deciding on a platform, as most platforms are designed with a particular purpose in mind. Protocols can be a convenient alternative to a platform if one can find one that contains the feature set required for the application (see 3.2 for more). In order to decide on what platform to use for our prototype, we need to compare different platforms to see which fits our use-case the best. The GET protocol will also be considered since it is made specifically for developing ticketing systems.

4.1 Ethereum vs Hyperledger Fabric

In order to determine which platform is better for our prototype, their different features needs to be compared in order to find the best suited for our use case. The use-case is vital since the platform needs to support features that are relevant for ticket services. Considering our use case, platforms like R3 Corda [15] will not be considered since it has a higher focus on the financial industry and lacks some of the more crucial features we are interested in[48]. While R3 Corda is not suited for our application, Ethereum and Hyperledger Fabric are, they focus more on the general development of applications, which is more relevant for our use case.

4.1.1 Ethereum

Vitalik Butuerin proposed the white paper for the concept of Ethereum in late 2013[49]. He created Ethereum as a solution to the problem of developers having to expand the functionalities of Bitcoin, or even design new platforms in order to develop applications with blockchain [50]. Ethereum was a new way of approaching this problem and allowed developers to create applications without having to expand on the Bitcoin platform.

> "I thought [those in the Bitcoin community] weren't approaching the problem in the right way. I thought they were going after individual applications; they were trying to kind of explicitly support each [use case] in a sort of Swiss Army knife protocol."
>
> - *Vitalik Buterin, inventor of Ethereum* [50]

The Ethereum Virtual Machine (i.e., EVM) is the core of the Ethereum platform[50]. The purpose of EVM is to make it easier for developers to develop applications on Ethereum, instead of building a new blockchain-based platform for each application. EVM enables any programs to run by using Turing-complete software to execute and store, for example, smart contracts[49]. This allows developers on Ethereum to implement contracts with more sophisticated logic.

EVM allows developers to build and deploy decentralized applications(i.e., dapp)[50]. Decentralized, as explained in section 2.1.1, means that no single person or entity controls the Ethereum blockchain. Software runs on the Ethereum network instead of being stored in a single location. It can also be used to build Decentralized Autonomous Organizations (i.e., DAO). DAOs can with the help of smart contracts, eliminate the need for people and centralized control within an organization.

Ethereum is a permissionless blockchain, which means that, although not highly confidential it can be both public or private. Ethereum uses a PoW consensus algorithm (see section 2.2.3). With PoW participants having access to every entry recorded and agree on a shared ledger [48]. This is one of the benefits of Ethereum, as the PoW algorithm makes it hard to tamper with the database without anybody noticing.

Ethereum can also be used to create other cryptocurrencies[50]. The ERC20 token is a standard defined by the Ethereum foundation, where other developers can make their own crypto based on an ERC20 token. Ethereum uses the ERC20 standard for their own cryptocurrency, Ether, which is built-in crypto in Ethereum. Another standard token that was recently created is the ERC721 token, which is used for tracking unique digital assets. An example is CryptoKitties, a game built on the new ERC721 token by having kitties as unique digital collectable assets.[51].

4.1.2 Hyperledger

Hyperledger is a global collaborative effort between companies in the finance, healthcare, and supply chain industries. Founded in 2016 by The Linux Foundation, the goal of Hyperledger is to create enterprise-grade, open-source, distributed ledger frameworks designed for business transactions [52]. Hyperledger currently curates more than ten different projects with varying use-cases, but all based on blockchain technology [53].
Two of the most popular Hyperledger projects currently are; Fabric and Sawtooth [54].
Fabric and Sawtooth, are fundamentally very similar but have some key differences; Sawtooth can be configured as public, while Fabric is only permissioned [55]. Fabric is more usefull for creating large industial networks [56]. For this thesis, we will be focusing mainly on Fabric, as it matches our use-case more closely then Sawtooth.

Hyperledger Fabric

Fabric is a joint venture between IBM[57] and Digital Asset[58] and is a permissioned DLT platform designed to handle more complex transactions than traditional platforms. Fabric, like all Hyperledger projects, does not operate with any form of cryptocurrency. It uses a highly modular and configurable architecture that allows more versatility and optimization for industry use cases in sectors like; finance, insurance, healthcare, and supply chain[59].

Fabric is one of the first DLT platforms to support smart contracts, or "chaincode" as Fabric calls it, and are programmed in more traditional languages like Java, Go and Node.js. By not having its own domain-specific language, like Ethereum with Solidity, most enterprises can develop smart contracts without learning a new language [59].

As mentioned, Fabric operates on a private and permissioned network, but even on a private network, all participants share a common ledger. To combat this, Fabric has Channels; channels function as a private subnet between two network participants, allowing them to communicate privately [59].

One of Fabrics most important features is that it supports pluggable consensus protocols. Which allows developers to pick and choose what protocol they want, or even change it midway. For example, starting with PBFT and then switching to Proof of Elapsed Time, later on. This makes the platform more customizable for particular use cases and trust models. With this flexibility, an enterprise can choose more appropriate consensus protocols for their use case and potentially save on performance[59].

All the features mentioned above combined make Fabric one of the best performing platforms that are available today. Both in terms of transaction processing and transaction confirmation latency. Fabric also enables better privacy and confidentiality of transactions and the smart contracts that implement them[59].

4.1.3 Comparing Ethereum and Hyperledger Fabric

When comparing Ethereum and Fabric, it is crucial to understand the platforms has a different target audience. Ethereum focuses more on the general developers, while Fabric focuses on the enterprise side of the market[60] . Both platforms have desired features for the prototype; therefore, we have a review of features sorted by importance.

Table 1: Comparing Ethereum and Hyperleger Fabric

Feature	**Ethereum**	**Hyperledger Fabric**	**Importance**
Use Case	General applications [60]	Enterprise focused, business to business transactions [60]	High
Confidentiality	Transparent [60]	Highly private [60]	High
Mode of Operation	Permissionless, public or private [60]	Permissioned, private [60]	Medium
Language support	Solidity, JavaScript [48]	Go, Java [48]	Medium
Governance	Ethereum [48]	The Linux Foundation [48]	Low
Consensus mechanism	Proof of Work [60]	Customizable [60]	Low
Cryptocurrency	Ether [48]	None [48]	Low

As seen in the table 1, Fabric concentrating use case for the enterprise by aiming to be a private blockchain. It only allows participants who are part of the transaction to partake. One downfall with Ethereum is transparent, that can be adverse to a company when handling sensitive information. As shown in chapter, many services used Ethereum; therefore, it can ensure privacy. Fabric uses more common language like Go and Java, while Ethereum uses Solidity and Javascript. A developer may have to learn a new language, but Solidity is language specific for a smart contract. Cryptocurrency and consensus mechanism has low importance because of none relevance to the prototype.

4.2 Comparing GET and Ethereum

There are already several existing solutions for blockchain-based ticket services, as mentioned in chapter 3. However, we will only be focusing on the GET protocol for this part, as it is the only complete protocol and because it is in use at the moment (see section 3.2).

We assume that since the GET protocol is built on Ethereum and is not a stand-alone platform, it will share most of its general attributes with Ethereum[39]. While Ethereum is designed for general application development, GET is designed explicitly for developing ticket applications and does not require any knowledge of Ethereum to use.

Another big difference is the platform protocol comparison; since GET is a protocol and not a platform; there is less flexibility when developing. It also means that a developer will be reliant on the GET developers for future updates or even to continue to exist, as GET shutting down would make systems based on them inoperative.
Table 2 makes it seem like GET is the optimal choice for our use-case. Given that it is custom made for ticketing systems and does not require any prior knowledge of Ethereum or blockchain in general to use. However, a significant difference is who holds the copyright for the system. Copyright is more important to our employer than how difficult it is to develop, since they want to own their solution. Using GET would be more fitting if, for example, an event organizer needed an application to sell tickets for their events.

Table 2: Comparing GET and Ethereum

Features	**GET**	**Ethereum**	**Importance**
Use case	Only used to create a ticket service with already finished protocol [39]	Can be used to develop another application bedside ticket services 4.1.1	High
Copyrights	GET foundation [39]	Developer	High
Type	Protocol	Platform	Medium
Usability	Easy, made not to need dircet contact with Ethereum [39]	Difficult, requires programming-skills on multiple languages 4.1.1	Medium
Tools	GET provided the developer with everthing them to GET protocol [39]	Mostly blockchain based platform which requires extra tools 4.1.1	Low

4.3 Choice of platform

When deciding on the most suitable development platform for this thesis, we focus on our employer's requirements and their use-case. Since they provide a service for multiple different event organizers, we can quickly conclude that the GET protocol becomes to narrowly focused and lacking in flexibility. The reason is that our employer can not further develop the GET protocol. If they were an event organizer organizing an event, the GET protocol could be a great solution. Additionally, our employer would like to own and fully control their service; therefore by using a protocol made by a third party; they lose this control and have no ownership, as shown in table 2 and mentioned in section 4.2.

With GET not begin an option; it is either between Ethereum or Fabric. When comparing the platforms, table 1 shows that Fabric is more focused on the enterprise market. It has features that are more geared towards handling more massive and more crucial business-to-business transactions. Thus Fabric was overly complicated for our use-case since the prototype is a simple demonstration of a blockchain-based ticketing system. Lastly, Ethereum, seems to be the best option in this scenario. Ethereum positions itself as more of an all-purpose development platform, and we find that the flexibility, large community, and the greater focus on application development (see section 4.1.1). It is also a popular choice in blockchain-based ticketing services, as shown in the chapter. Hence, Ethereum seems to be the most reasonable choice.

5 Design

In the project description (see section 1.3), we mention that our employer wants a prototype to showcase blockchain technology in a ticketing system. When we made the choices regarding the prototype's design, we took into account the requirements, blockchain technology, and our delimitations (see section 1.4).

5.1 Requirements

Our employer's requirements are relatively simple. The system should be capable of assigning a ticket to a user when purchased. It is crucial that a ticket can be validated only once when a user enters a game as it cannot be reused. Every ticket should have a history, that includes time of purchase and validation, and who used it at which time and at which event. The event organizer and our employer should be available to see the history of each ticket.

The tickets need a unique id to make them identifiable. Optimally, it could include a QR code, but this is optional for the prototype. Every ticket needs to connect to a user when purchased, either by name or user id. Lastly, it needs a status, active, or used.
A simple web UI is required for testing, where the following is possible:

- Create a ticket
- Assign ticket to a user
- Validate the user's ticket and change the status from active to used
- Withdraw the user's ticket and show its history over assignments and its status

5.2 Technical Design

In the design phase for the prototype, we decide to add some additional functionality besides the ones that already were in the requirements. The reason for this decision is that we wanted the prototype to be more similar to a real-life ticketing system.

5.2.1 Use case model

Figure 2 shows the use cases for the system. The system only contains the main functionality of the prototype: create games, create tickets, purchase tickets, validate and invalidate tickets and, starting and ending games. Thus there are only two actors: user and event organizer. The use cases serve as a demonstration of the system flow between the actors and the functionality and include detailed descriptions of scenarios.

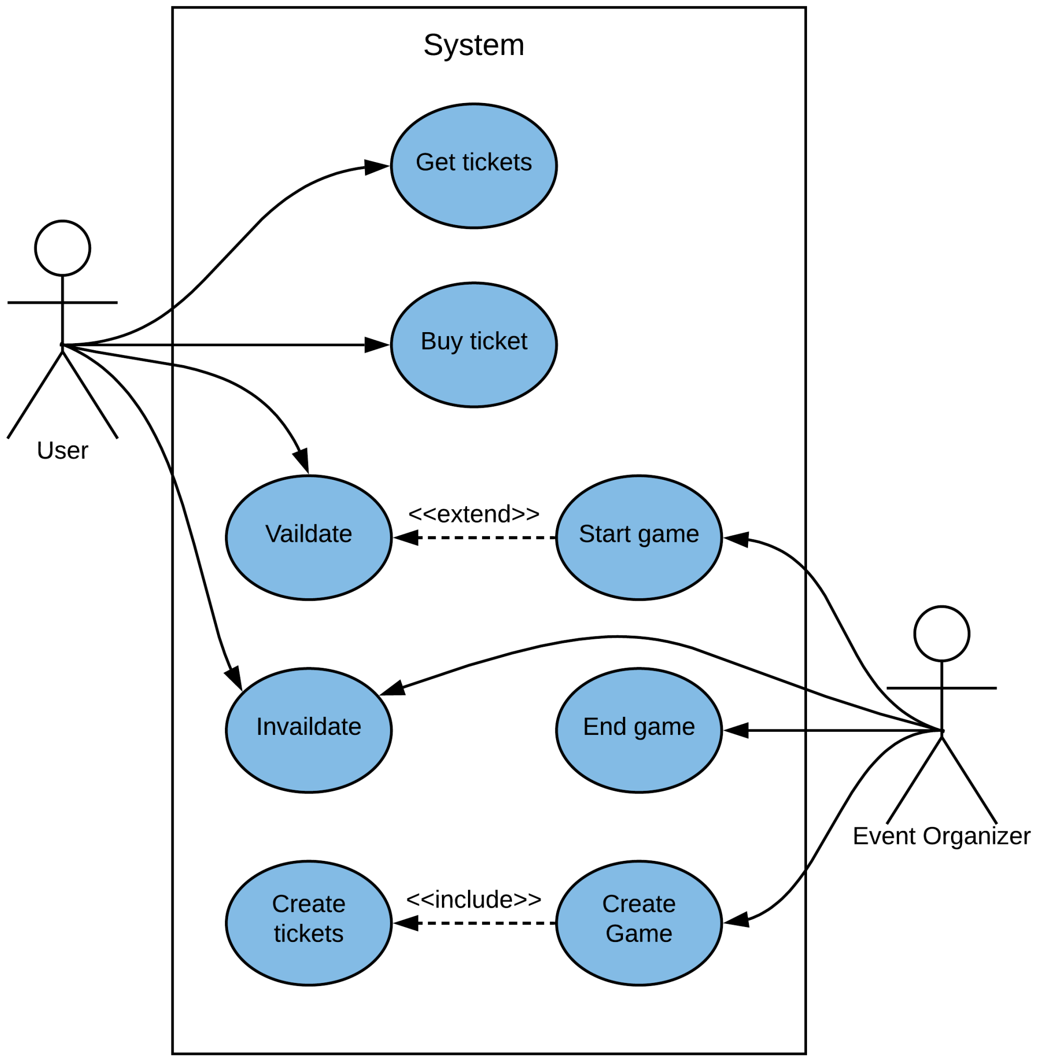


Figure 2: Use Case Diagram

Use case	**Create game**
Actor(s)	Event organizer
Description	Allow an event organizer to create a game
Goal	Create a game
Post-Conditon	Event Organizer has successfully created a game

Use case	**Create tickets**
Actor(s)	Event Organizer
Description	Allow an event organizer to create tickets to a game
Goal	Create tickets to a game
Pre-condition	A game must already exists
Post-condition	Event Organizer has successfully created tickets to a game

Use case	**Buy ticket**
Actor	User
Description	A user buys ticket(s) to a game from the event organizer
Goal	The user can purchase ticket(s)
Pre-condition(s)	The game exists, and there are ticket(s) available to purchase
Post-condition	The user owns ticket(s) to the game

Use case	**Start game**
Actor	Event organizer
Description	The event organizer starts a game
Goal	The event organizer has a successful started a game
Pre-condition	The event organizer must be organizing the game
Post-condition	The game has started

Use case	**End game**
Actor	Event Organizer
Description	The event organizer can end a game
Goal	Allow an event organizer to end a game
Pre-condition(s)	The event organizer must be organizing the game, and the game must be ongoing
Post-condition	The game has ended

Use case	**Vaildate**
Actor(s)	Event organizer and user
Description	An event organizer validates a user(s) ticket(s) for a game
Goal	The ticket(s) was successful validated
Pre-condition	The user must own ticket(s) to the game, and the game must be ongoing
Post-condition	The user's ticket(s) are validated

Use case	**Invaildate**
Actor(s)	Event Organizer and user
Description	The event organizer can invalidate a user's ticket(s)
Goal	Invalidate a user's ticket(s)
Pre-condition	The user must own at least one ticket
Post-condition	The user's ticket(s) are invalidated

5.2.2 System structure

As observed in figure 3, the system is relatively basic in its design. The system consists of four structs: `event organizer`, `game`, `ticket`, and `user`. An event organizer can host games; these games have a state derived from `gameState` and have tickets associated with them. A user can purchase tickets from a game; this copies the ticket object from the game to the user. The ticket derives its state from `States`, for example when it is purchased.

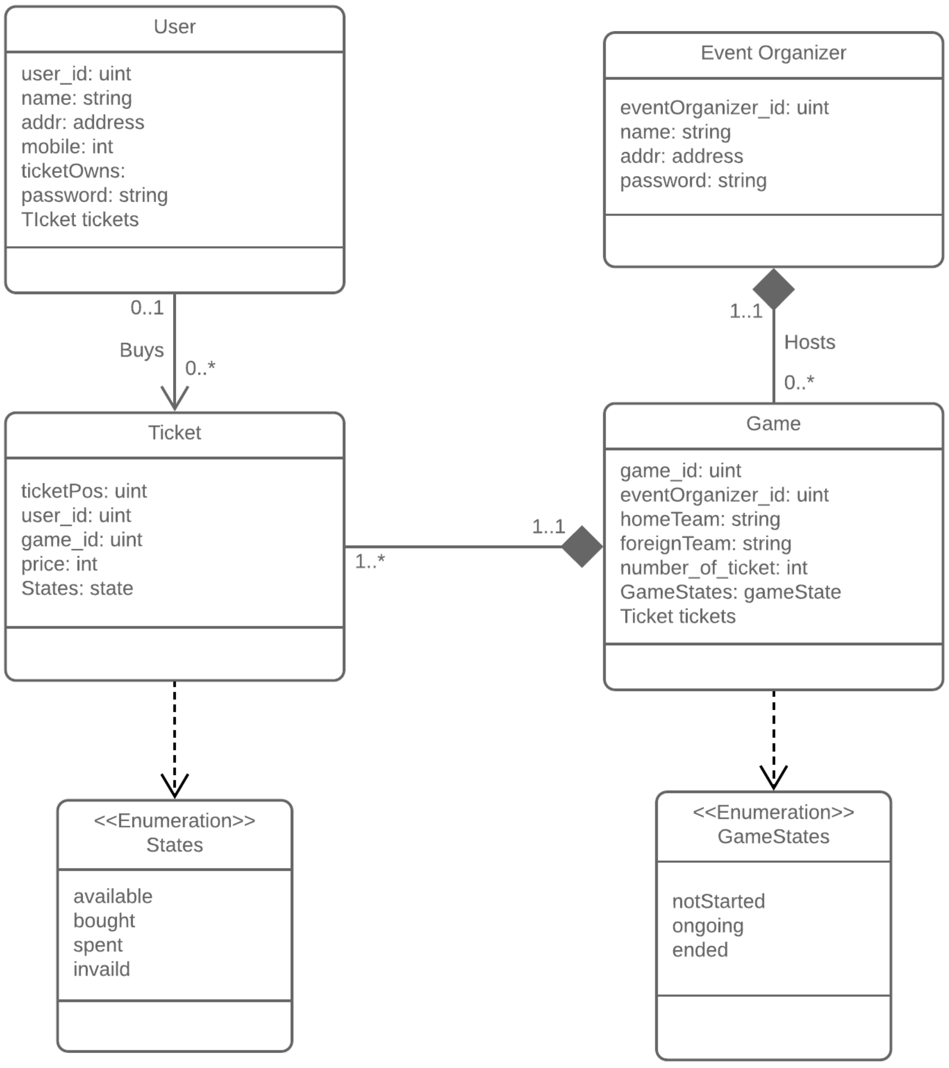


Figure 3: Shows the system structure and dependencies

5.2.3 GUI

Our employer's requirements for the GUI were reasonably straightforward. They wanted a website with buttons for each functionality listed in the requirements section 5.1. They only requested one button for purchase, one for validation, and one for withdrawal of the history of a ticket.

Figure 4 shows our proposed design for the website. Its a simple layout with a navigation bar that help users navigate between the different parts of the system. The site is divided into the three main parts of the system; the user tab, the game tab, and the buy tab.

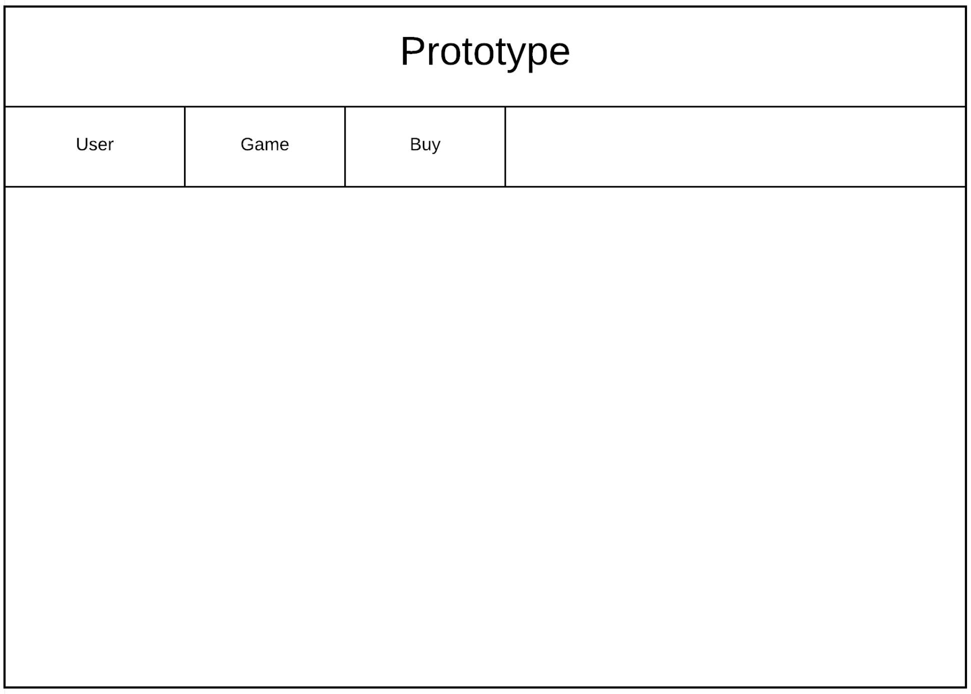


Figure 4: The proposed design for the web interface

6 Implementation

6.1 Development Environment

6.1.1 Software

- **IDE:** Visual Studio Code [61]
- **Language:** Solidity, JavaScript, HTML and CSS
- **Libraries:** Web3 [62], Bootstrap [63]
- **Framework:** Truffle [64]
- **Network Interface:** Geth [65]
- **Network Interface:** Ganache [66]
- **Server Environment** Node.js [67]
- **Version Control:** Github [68]

When developing on Ethereum, we will be coding in their proprietary language solidity, which is influenced by C++, Python, and Javascript but designed specifically for smart contract development. Node.js and JavaScript are used to interface with the contracts from the web UI and run scripts. HTML, CSS and Bootstrap were used to design the web interface. Geth is a command line interface used for running a full Ethereum node and is needed to connect to an Ethereum network. Ganache, on the other hand, creates a personal blockchain that can be used for testing and developing without connecting to an actual network and makes testing contracts more straightforward and user-friendly, we used this for all of our development. To interface with the network created by Geth or Ganache, we used Truffle. Truffle is a development framework for Ethereum that supports among other things; smart contract compilation, automated testing, debugging tools, script support, and network management, where compilation, testing and debugging is most relevant for us. When it comes to writing the code, we decided to go with Visual Studio Code, as it has support for both Solidity and JavaScript linting, and is also a familiar IDE for everyone on the team.

6.1.2 Skyhigh server

NTNU has an internal server hosting service called Skyhigh, where students can borrow server time to run their code. We got allocated a Linux VM where we configured Geth and Truffle to host an Ethereum network that we can deploy our smart contracts on. A setup like this is necessary to be able to test the system with more than one node. But, due to time constraints and it being easier to develop locally with Ganache, we ended up not using the server at all.

6.2 Development

In the development process, we used Ganache for testing the smart contract locally on our computer, since Ganache allows us to see transactions and event logs while testing. The reason for developing on a test network first is to not deploy anything on a main net without testing to make sure everything works properly first.

We decided to only have one smart contract because it was simpler to control functions and variables. If we were to realize the prototype in a real-life scenario; tickets, users, event organizers, and games would have their own smart contracts and inherit from each other. The smart contract is the backend code, while the website is the frontend. We are using the web3 library, which is a javascript library for communication between the back-end and the ethereum network. We are also using Bootstrap for the design of the website.

Initially, in the development process, we first planned out the logic of the smart contract. After that, we realized that we needed four actors: ticket, user, game and event organizer. All actors have their own structs which includes their local variables. To make sure each struct is unique, we used a uniqe id. Everything except ticket has an id; ticket does not need id to be unique since it has the game id and tickets position in the mapping for that game, making the ticket unique. For example, if a ticket has game id 15 and position 0; another ticket can still have position 0, but not in the same game. Therefore a ticket is always unique by the combination of ticket position and game id.

All structs bedsides ticket are declared as global arrays. These are dynamic, not fixed, which in theory, means that the system can hold as many games, event organizers and users as needed. It also provides a more realistic scenario where the ticket system is not limited by how much it can hold in the database.

```
    //all Users
    user[] Users;

    //all Games
    game[] Games;

    //all event Organizer
    eventOrganizer[] EventOrganizers;
```

Listing 6.1: global arrays

User and event organizer are independent, while games need an event organizer to exist, which makes Games dependent on the event organizer. We did not make a global array for tickets, but instead, we are mapping it in the game and user as a way to connect them to the ticket. This is because a ticket should not exist without a game; therefore, it must be connected. Users also have their own set of the tickets, which only includes tickets they own. While the user can only see the tickets they own, games can see all tickets with the same game_id. Each game is connected to an event organizer with the eventOrganizer_id in both structs. We could have mapped the game in the event organizer's struct but decided it was not necessary since by not using mapping, it was easier to control the different functions.

```
    function createEventOrganizer(string memory _name, address _addr, uint
        _evnetOrganizerId, string memory _password) public returns(bool){

        //check if event organizer already exits
        if(checkEventOrganizer(_evnetOrganizerId) && checkAddr(_addr)){
            eventOrganizer memory e = eventOrganizer(_evnetOrganizerId,
                _name, _addr, _password);      //Create a temporary memory
                of struct eventOrganizer
            EventOrganizers.push(e);

            emit CreateEventOrganizer(_evnetOrganizerId, _name, _addr,
                _password);

            return (true);

        } else{

            return(false);
        }
    }
```

Listing 6.2: createEventOrganizer function

The code in 6.2 shows how the system creates an event organizer. The function receives the required parameter for the event organizer. The address needs to be unique; we therefore have to make sure there are no matching addresses in the system. This is done trough the `checkaddr(address_addr)` function.

The code for `createUser` is similar to `createEventOrganizer`. This is because both are treated the same way in the system. Another thing similar to `createEventOrganizer` is the use of `checkaddr(address_addr)` function to make sure there are not any duplicates.

```
    function createGame(uint _gameId, uint _evnetOrganizerId, string memory
        _homeTeam,
    string memory _foreignTeam, uint _tickets, int _price)public returns(
        bool){

        //check if game already exists and event organizer exists
        if(!checkEventOrganizer(_evnetOrganizerId) && checkGame(_gameId)){

            game memory g = game(_gameId, _evnetOrganizerId, _homeTeam,
                _awayTeam, _tickets, GameStates.notStarted);       //Create
                 a temporary memory of struct game
            Games.push(g);

            emit CreateGame(_gameId,_evnetOrganizerId, _homeTeam,
                _foreignTeam, _tickets, GameStates.notStarted);

            createTicket(_gameId, _tickets, _price);
                                              //call the function
                createTicket for creating the tickets in a game

            return (true);

        } else{
            return(false);
        }
    }
```

Listing 6.3: createGame function

The function `createGame` 6.3 creates a game and then adds it to Games array. It requires six parameters to run; gameId, event organizer Id, home team, foreign team, amount of tickets and the price for each ticket. The function makes sure that the game id is unique and that the event organizer id exists in the system. If both the checks pass we create a local game inside the function with all our parameters, and we set the game's state to notStarted. Then we push the local game to the Games array and emit the event, which logs the creation in the blockchain. After that createTicket is called with game id, number of tickets and price.

```
    function createTicket(uint _gameId, uint _tickets, int _price)public
        returns(bool){
        uint pos = findPosGame(_gameId);

        //check if the _gameId and _tickets is same in Games[_gameId]
        if(Games[pos].game_id == _gameId && Games[pos].number_of_tickets ==
             _tickets){
            //creating the number of ticket in a game
                for(uint i = 0; i < _tickets; i++){

                    Games[pos].tickets[i] = ticket(i, 0, _gameId, _price,
                        States.available);    //add tickets for tickets in
                        struct game

                    emit CreateTicket(i, 0, _gameId, _price, States.
                        available);
                }

                return (true);

            } else {

                return (false);
            }
         }
```

Listing 6.4: createTicket function

`createTicket` in 6.4 uses the parameters sent from the `createGame` function. It starts by finding the position of the game in the Games array, then checks if the game id and the number of tickets match its parameters. Then it goes through and adds all the tickets to the game with the state as available and the owner set to 0, since no one owns ticket yet.

```
    function buy(uint _posU, uint _posG)public returns(bool){
        uint posT = Users[_posU].ticketOwns;

        //go throught very ticket in the game
        for(uint i = 0; i < Games[_posG].number_of_tickets; i++){

            //check if the ticket are available
            if(Games[_posG].tickets[i].state == States.available){

                Games[_posG].tickets[i].user_id = Users[_posU].user_id;
                          //Sett the user id

                Games[_posG].tickets[i].state = States.bought;
                                  //change the state to bought

                Users[_posU].tickets[posT] = Games[_posG].tickets[i];
                            //cop ticket to Users[].tickets[]

                emit BuyTicket(Users[_posU].user_id,
                Users[_posU].tickets[posT].game_id,
                posT, Users[_posU].tickets[posT].price,
                Users[_posU].tickets[posT].state);

                Users[_posU].ticketOwns++;

                return (true);
            }
        }
        return (false);
    }
```

Listing 6.5: buy function

When a user purchases a ticket, the `buyTickets` function is called with the parameters; _gameId, _userId and _tickets. This function checks if a game has enough available tickets and then calls the `buy` function once for each ticket the user wants to purchase. This function runs trough all tickets in a game and finds the first available one. It then sets the user_id to the same as the buyer's, sets the ticket's state to bought and adds the ticket to the buyer's ticket mapping.

```
    function vaildateTicket(uint _userId, uint _gameId) public returns(bool
        ){
        uint posG = findPosGame(_gameId);
        uint posU = findPosUser(_userId);

        if(!checkGame(_gameId) && Games[posG].gameState == GameStates.
            ongoing){
            for(uint i = 0; i < Users[posU].ticketOwns; i++){
                if(Users[posU].tickets[i].game_id == _gameId && Users[posU
                    ].tickets[i].state == States.bought){

                uint tickPos = Users[posU].tickets[i].ticketPos;

                Games[posG].tickets[tickPos].state = States.spent;
                                              //change state to spent

                Users[posU].tickets[i] = Games[posG].tickets[tickPos];
                                   //copy the ticket info to Games[]

                emit TicketState(_userId, _gameId, i, Users[posU].tickets[i
                    ].state);
                }
            }
            return (true);
        }
         return (false);

             //return false
    }
```

Listing 6.6: validateTicket function

`ValidateTicket` 6.6 takes two parameters, user Id and game Id. It uses these to find the position of both the game and user in their respective arrays. Then it checks if the game exists and that the game's state is set to ongoing. Then a for loop runs through all the tickets owned by a user and checks if they have a ticket with a matching game Id and that the ticket's state is set to bought. Then the ticket's position is used to set the ticket's state to spent. Similarly, invalidateTicket is mostly the same as validateTicket.

6.3 Finished Product

6.3.1 Web interface

For the web interface, we decided to make a simple layout that a user can easily navigate (see section 5.2.3 for more). By using the web3.js library we could connect the frontend to the backend, which allowed us to call functions in the smart contract from the web interface. We also included a frontend component library, Bootstrap, to make the design neater. Other assets like Innit's logo were provided to us by them.

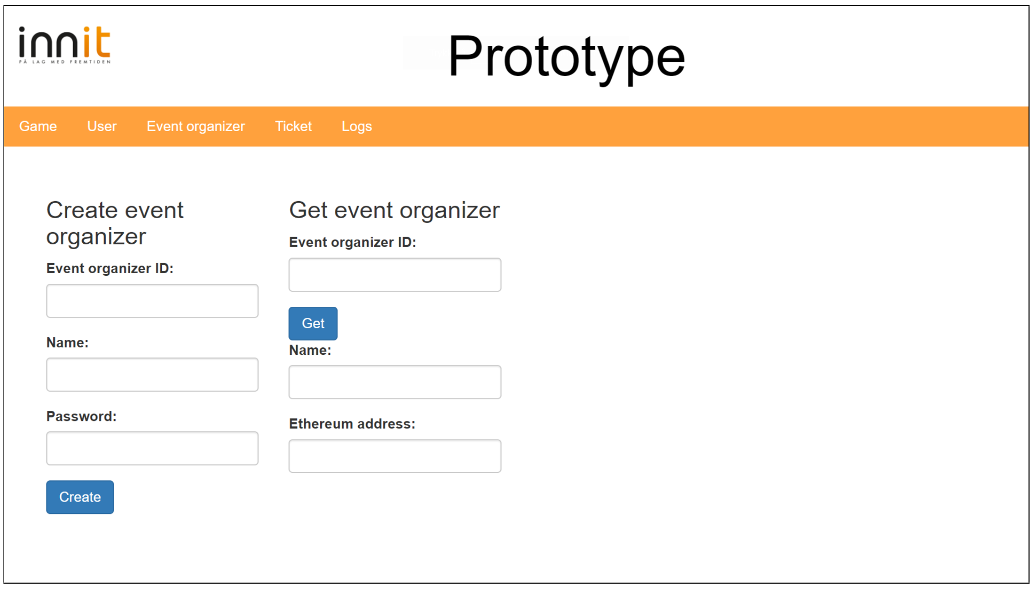


Figure 5: Page to event organizer

Figure 5 shows the event organizer page. Under "create event organizer" the necessary fields needs to be filled out in order to create an event organizer. There is no way for an organizer to select an ethereum address because when creating an event organizer, the system will automatically give the event organizer one. As shown in listing 6.2, the system does not accept an id that is already being used. If the event organizer wonders what their ethereum address is or how many games they are organizing, they can get that information under "get event organizer" by typing their id.

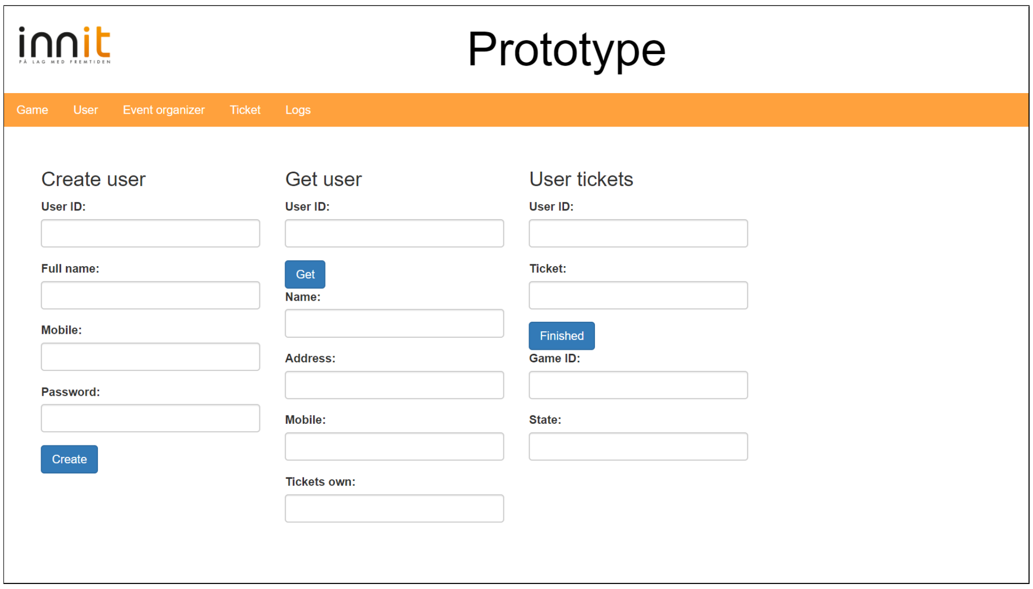


Figure 6: Page to user

Figure 6 show the user page, which is similar to the event organizer's page. The reason is as mentioned in section 6.2, as the create functions are very similar, the interfaces become too. The only difference in the web interface is "User ticket" which is used to get information regarding the user's tickets. In order for the function to get a ticket's information, the user's id and the ticket's number is needed as input. The ticket number is the ticket's position in the mapping which is connected to a user. For example, if a user inputs ticket number 10, but only owns 5 tickets, he will receive an error.

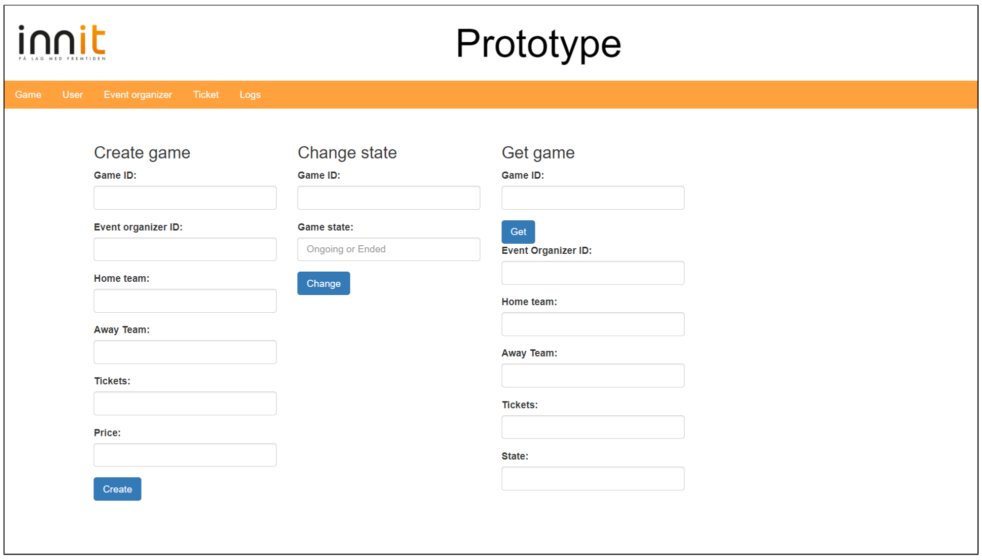


Figure 7: Page to game

The game page again share some of the same features as the user and event organizer pages. This page is meant for event organizers to create games, shown under "Create game", which needs an event organizer id. "Get game" is again similar to, for example, event organizer's "Get event organizer", and needs a valid id, if an invalid id is used, an error message will be displayed. Another difference is the ability to change the game's state to ongoing or ended. Something worth noting is that when a game is created, it will automatically have its state set to not started.

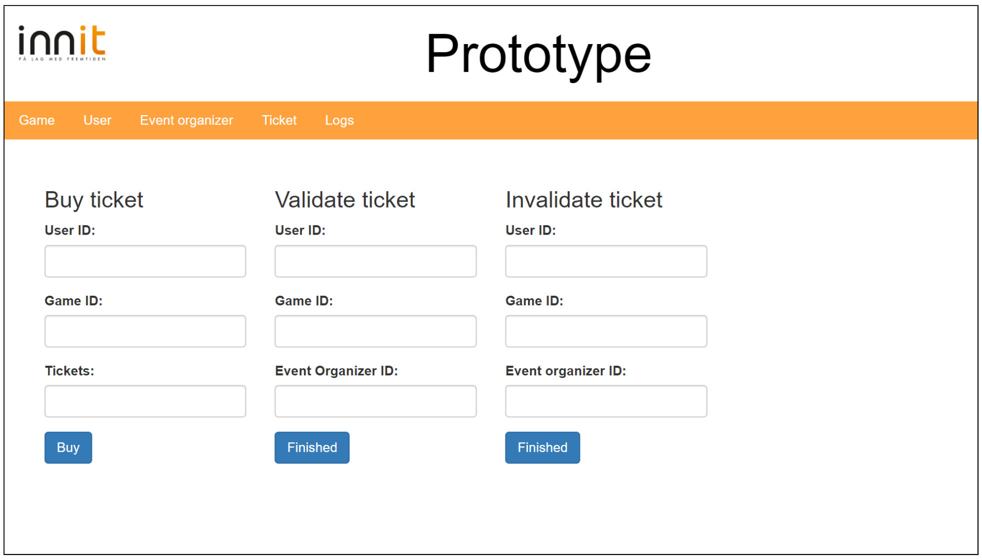


Figure 8: Page to ticket

The ticket's page contains the most important functions in the prototype, the buy, validate and invalidate ticket functions. "Buy ticket" needs a user id, game id and how many tickets a user want to buy. The tickets bought will be added to the user's ticket mapping. In order to withdraw tickets in the user's page, a user will first have to buy tickets. A event organizer can validate or invalidate a user's ticket using their respective functions.

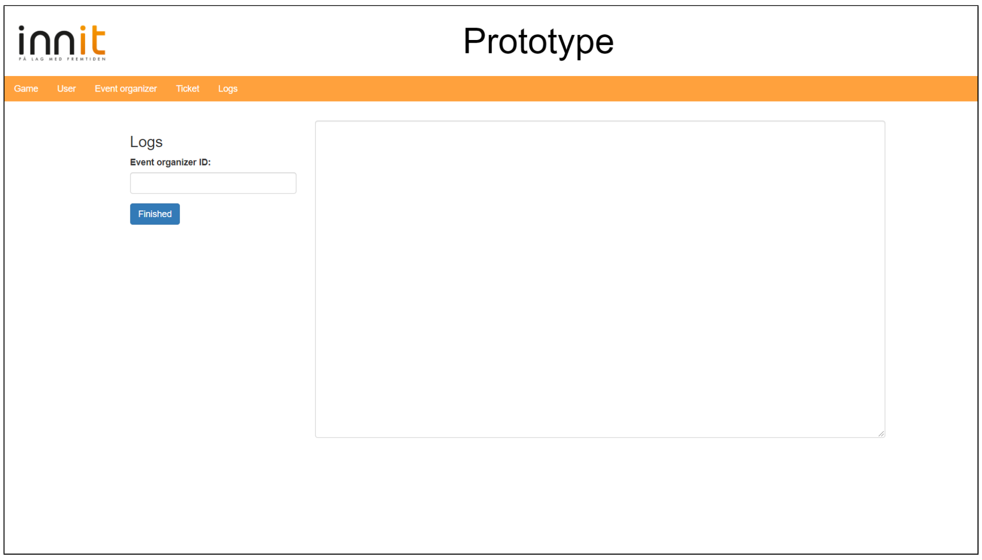


Figure 9: Page to log

The logs page does not include any specific functionality from the smart contract, but is used to withdraw the logs from the blockchain. This is a way for an event organizer to keep track of changes in the blockchain. This page uses the function `getPastEvents` from the web3.js library, unlike the other pages, who call functions from the smart contract.

The web interface is meant to illustrate how we wanted the prototype to work. It includes not only smart contract functionality but also web3's functionalities in order to communicate with the blockchain's data. Generally, the web interface simply showcases the prototype's functionality; it does not showcase how the website should look, there is room for improvement when it comes to a fully functional ticketing system.

6.3.2 Testing

To test the prototype, we made scripts that create a test scenario, to simulate how the system would operate in reality. When running the scripts, they will create users, event organizers, and games. They will allocate tickets to users, change games states and, validate and invalidate user's tickets. All scripts that change or add data to the system, creates new blocks in the network.

We first run the scripts: `createEventOrganizers.js`, `createUsers.js` and `createGames.js` to create the necessary data for the system. It is important to run `createEventOrganizers.js` before `createGames.js` since a game requires an event organizer to exists.

After running the necessary scripts, we can run `getEventOrganizers.js`, `getUsers.js` and `getGames.js`. None of these scripts will create any blocks since they only withdraw and show data. These scripts are not required to run for the system to work, but allows us to check if everything was added correctly.

Users can now buy tickets to a game by running the `user1Buys.js`, `user2Buys.js` and `user3Buys.js` scrpts. Since these scripts change the blockchain, they will create new blocks of data that includes information about the transaction. These scripts can be run multiple times until there are no more tickets available. After the user has bought a ticket, they can run `getUsersTickets.js` to see the information about their tickets.

Before validating a user's tickets, a game has to be ongoing, this can be done by running `startGames.js`. An event organizer can then validate a ticket by running the `vaildate.js` which will validate all tickets bought for that game. They can also end games by running `endedGames.js`, preventing more tickets from being validated. If they need to invalidate tickets, if for example, the game was canceled, they can run `invalid.js` which changes the ticket's state to "invalid".

7 Discussion

7.1 Discussion

We initially set out to discover whether or not a blockchain-based ticketing system would be feasible and a worthwhile investment for our employer. Now at the end of the project, we are comfortable saying that blockchain-based ticketing systems are not only feasible and practical but also relatively easy to develop compared to our initial expectations. We have delivered to our employer, not only a report highlighting all of this but also a prototype system that exceeds the initial requirements.
The main advantage of using blockchain is security. It is tamper-proof, therefore a company can be safe that their data would not be tampered with. Blockchain can also provide logs of all the system changes which is practical when, for example, troubleshooting issues in the system. Another thing is that a blockchain can be transparent; therefore if a ticket changes, it would need the approval of all nodes in the blockchain network (see section 2.2). The company can add features like facial verification or non-fixed QR (see section 3.3.3), in order to make the system safe as possible for the consumer. The safety in the prototype is the identity management to user's and event organizer's Ethereum address as it is unique. This can be used as a way to identify.

There are however downsides to using blockchain technology for these types of applications. A traditional database have certain advantages over blockchain technology. It can remove garbage data; meanwhile, on a blockchain, everything is permanent. Therefore it cannot remove old data from the system as it is ever growing a list of records(see section 2.1). For example, if the blockchain holds a ten-year backlog of tickets, removing the oldest to free up space, is not possible. Hence, the blockchain multiplies in size, and you have to account for an ever-expanding database. This was something we experienced when testing the prototype, the list grew prominently and over-time started to include garbage data. That means there were blocks that no longer held relevant data and could have been removed. While this was not an issue with the scale of the prototype, in lager systems, this could be an issue.
Scalability can also be a challenge with blockchain; for example, the prototype experienced issues when redeploying our smart contracts where the system did not recognize the updated contract. Additionally, when expanding a contract's functionality, the network will require more resources to run the expanded contract. This can lead to system failure if not enough resources are available which experienced in the prototype. Issues like this would be critical in an operational system.

Blockchain can be used as a ticketing system (see chapter 3). As mention in section 3.2.2, GUTS is a functional ticketing service powered by blockchain. Evopass was a company that decided to step away from blockchain because of latency issues caused by blockchain. When comparing these, it shows that blockchain works as a ticketing system, but challenges in the market and technical issues can provide a challenge.

7.2 Group Evaluation

7.2.1 Initially

We have known each other for three years and had worked together numerous times before, so we knew what to expect from each other. Daily discussions were held to make sure we were on the same page when it came to the report and work to be done. There were few internal problems during the project period, the only issue we had were a few illnesses.

7.2.2 Distribution of work

We original split responsibility into three parts, but found during the project that it was easier to work to together. We distributed the different sections between us and went through everyone's work weekly. Initially, we all started reading about blockchain, what it is, how it works and its different uses. Then, we split up and began researching the different frameworks and protocols that we could use. One team member was later assigned to the prototype, while the other two kept working on the report, but were available to help with development.

8 Conclusion

8.1 Future work

As this thesis is mostly a theoretical report, with a practical demonstration, it is meant to inform our employer about the potential of blockchain technology in ticketing systems. For future work, we would recommend looking more in-depth at security, and potentially do some more market investigation, surveys could be run to see if there is consumer interest in this technology, or contact one or more of the existing service providers and talk with them about their experiences. When talking about the prototype, since it lacks a lot of the more complex features of a production-ready system, we would recommend using it for inspiration rather than building upon it. However, there are potential changes that could be made to make the prototype more complete. The prototype should be deployed on a server to allow multiple users to connect. There could also be a login system, removing the need for users to remember their user_id.

8.2 Conclusion

In this thesis, we have researched blockchain technology in order to find out if a ticketing system based on this technology is feasible and practical. We discovered that there were already multiple existing solutions, not only applications but also protocols based entirely on making ticketing systems. These solutions show there is interest in the technology, but as most of these are still in early development, it is hard to conclude whether or not this is the future of ticketing.
Based on our research, we found that the Ethereum platform was the best choice for our use-case. During development it became clear that these systems are not as hard to develop as we initially expected. This lead to the prototype far exceeding the initial requirements set by our employer.
Given the improved prototype and our research prior to development, we can conclude that it is both feasible and straight forward to develop blockchain-based ticketing systems.

Bibliography

[1] October. 5 2017. Live performance australia. http://members.liveperformance.com.au/uploads/files/LPAMRTicketing%20SurveyOct2017%20(all%20media)%20FINAL-1507074102.pdf.

[2] Morris, D. Z. 2017. Bitcoin hits a new record high, but stops short of $20,000. http://fortune.com/2017/12/17/bitcoin-record-high-short-of-20000/. (Visited March. 22 2019).

[3] 2017. What is blockchain technology? a step-by-step guide for beginners. https://blockgeeks.com/guides/what-is-blockchain-technology. (Visited March. 13 2019).

[4] 2015. The great chain of being sure about things. https://www.economist.com/briefing/2015/10/31/the-great-chain-of-being-sure-about-things. (Visited March. 25 2019).

[5] Lafaille, C. 2018. What is blockchain technology? a beginner's guide. https://www.investinblockchain.com/what-is-blockchain-technology/. (Visited March. 18 2019).

[6] Curran, B. 2018. What is a merkle tree? beginner's guide to this blockchain component. https://blockonomi.com/merkle-tree/. (Visited May. 14 2019).

[7] What is cryptocurrency. guide for beginners. https://cointelegraph.com/bitcoin-for-beginners/what-are-cryptocurrencies#history. (Visited March. 18 2019).

[8] Lai, V. 2018. Introduction to cryptography in blockchain technology. https://crushcrypto.com/cryptography-in-blockchain/. (Visited April. 24 2019).

[9] Decentralized computing & storage: Building a global supercomputer for all. https://www.skalex.io/decentralized-computing/. (Visited April. 24 2019).

[10] Blockchains & distributed ledger technologies. https://blockchainhub.net/blockchains-and-distributed-ledger-technologies-in-general/. (Visited March. 18 2019).

[11] Bitcoin. https://www.bitcoin.com/. (Visited May. 19 2019).

[12] ethereum blockchain app platform. https://www.ethereum.org/. (Visited April. 17 2019).

[13] Monax helps businesses focus on value creation by simplifying paperwork. https://monax.io/. (Visited May. 14 2019).

[14] Open platform for building blockchains. https://www.multichain.com/. (Visited May. 14 2019).

[15] r3. 2018. The corda platform. https://www.r3.com/corda-platform/.

[16] Iyer, K. & Dannen, C. 2018. *Building Games with Ethereum Smart Contracts*. Apress, (Visited April. 1 2019).

[17] 02.02.2019. Understanding blockchain frameworks: Ethereum and hyperledger fabric. https://medium.com/coinmonks/understanding-blockchain-frameworks-ethereum-and-hyperledger-fabric-48a57082903e. (Visited March. 19 2019).

[18] Bauerle, N. What is a distributed ledger? https://www.coindesk.com/information/what-is-a-distributed-ledger. (Visited March. 13 2019).

[19] Brakeville, S. & Perepa, B. 2018. Blockchain basics: Introduction to distributed ledgers. https://developer.ibm.com/tutorials/cl-blockchain-basics-intro-bluemix-trs/. (Visited March. 15 2019).

[20] Ray, S. 2018. The difference between blockchains & distributed ledger technology. https://towardsdatascience.com/the-difference-between-blockchains-distributed-ledger-technology-42715a0fa92. (Visited March. 15 2019).

[21] Smart contracts & dapps. https://lisk.io/academy/blockchain-basics/use-cases/smart-contracts-and-decentralized-platforms. (Visited March. 15 2019).

[22] 2017. Smart contracts: The blockchain technology that will replace lawyers. https://blockgeeks.com/guides/smart-contracts/. (Visited March. 13 2019).

[23] Smart contracts. https://blockchainhub.net/smart-contracts/. (Visited March. 18 2019).

[24] Consensus protocols. https://lisk.io/academy/blockchain-basics/how-does-blockchain-work/consensus-protocols. (Visited March. 25 2019).

[25] 2018. What is a blockchain consensus algorithm? https://www.binance.vision/blockchain/what-is-a-blockchain-consensus-algorithm. (Visited March. 25 2019).

[26] Anwar, H. 2018. Consensus algorithms: The root of the blockchain technology. https://101blockchains.com/consensus-algorithms-blockchain/. (Visited March. 25 2019).

[27] Softjourn. 2018. Blockchain revolutionizes ticketing. https://softjourn.com/blog/article/blockchain-revolutionizes-ticketing. (Visited March. 22 2019).

[28] Tirkakis, J. 2018. Solutions in event ticketing according to hellosugoi. https://medium.com/hello-sugoi/solutions-in-event-ticketing-according-to-hellosugoi-ef333395c724. (Visited March. 25 2019).

[29] 2017. A chain of custody ticketing system. https://blocktix.io/. (Visited April. 3 2019).

[30] Anwar, H. 2018. Top 10 blockchain adoption challenges. https://101blockchains.com/blockchain-adoption-challenges/. (Visited April. 1 2019).

[31] 2017. The future of tickets. https://media.consensys.net/the-future-of-tickets-a729ea4e9c95. (Visited April. 5 2019).

[32] Pandya, N. 2018. How blockchain can boost the ticket booking industry. https://medium.com/coinmonks/how-blockchain-can-boost-the-ticket-booking-industry-79ea56fbfab. (Visited April. 8 2019).

[33] 2018. Secure fan-to-fan ticket resale. https://www.evopass.io/. (Visited April. 7 2019).

[34] 2019. Evopass, about us. https://www.evopass.io/about-us/.

[35] 2018. How blockchain can help musicians survive. https://www.longhash.com/news/how-blockchain-can-help-musicians-survive. (Visited April. 3 2019).

[36] 2018. Evopass: End of the road. https://medium.com/evopass/evopass-end-of-the-road-1ec62799cb74. (Visited April. 1 2019).

[37] Blocktix. 2019. Blocktix review of 2018 / 2019 q1. https://blog.blocktix.io/blocktix-review-of-2018-2019-q1-24254cecc06d. (Visited Mai. 14 2019).

[38] Mathieu, F. & Mathee, R. 2017. Blocktix: Decentralized event hosting and ticket distributionnetwork. https://blog.blocktix.io/blocktix-review-of-2018-2019-q1-24254cecc06d. (Visited Mai. 14 2019).

[39] Team, G. F. 2017. Guaranteed entrance token smart event ticketing protocol. https://guts.tickets/files/GET-Whitepaper-GUTS-Tickets-latest.pdf. (Visited April. 4 2019).

[40] 2019. Our progress & roadmap. https://get-protocol.io/about/roadmap/. (Visited April. 15 2019).

[41] Foundation, A. P. 2019. The aventus protocol. https://aventus.io/.

[42] Foundation, G. P. 2019. Honest ticketing. https://guts.tickets/.

[43] 2018. Aventus whitepaper. https://aventus.io/doc/whitepaper.pdf. (Published June 2018, Version 4).

[44] 2019. Ticketmaster. https://www.ticketmaster.com/.

[45] Trikakis, J. 2018. Problems in event ticketing according to hellosugoi. https://medium.com/hello-sugoi/problems-in-event-ticketing-according-to-hellosugoi-8edbbce28815. (Visited April. 1 2019).

[46] 2007. Lov om forbud mot prispåslag ved videresalg av billetter til kultur- og idrettsarrangementer. https://lovdata.no/dokument/NL/lov/2007-06-29-86. (Visited April. 4 2019).

[47] Evopass. 2017. Introducing evopass. https://medium.com/evopass/introducing-evopass-47e0ed574a50. (Visited April. 3 2019).

[48] Sandner, P. 2017. Comparison of ethereum, hyperledger fabric and corda. https://medium.com/@philippsandner/comparison-of-ethereum-hyperledger-fabric-and-corda-21c1bb9442f6. (Visited April. 13 2019).

[49] Curran, B. 2018. What is ethereum? beginner's guide to this decentralized computing platform. https://blockonomi.com/ethereum-guide/. (Visited April. 15 2019).

[50] 2017. What is ethereum? the most comprehensive guide ever! https://blockgeeks.com/guides/ethereum/. (Visited April. 15 2019).

[51] 2019. Cryptokitties: Collectible and breedable cats empowered by blockchain technology. https://drive.google.com/file/d/1soo-eAaJHzhw_XhFGMJp3VNcQoM43byS/view. (Visited May. 15 2019).

[52] 2018. About hyperledger. https://www.hyperledger.org/about. (Visited April 17 2019).

[53] 2019. Hyperledger projects. https://www.hyperledger.org/projects. (Visited 15.05.19).

[54] Maltseva, D. 26.07.18. 10 most popular& promising blockchain platforms. https://dev.to/dianamaltseva8/10-most-popular--promising-blockchain-platforms-djo. (Visited 15.05.19).

[55] Sitoh, P. 04.12.18. What are the differences between ethereum, hyperledger fabric and hyperledger sawtooth. https://medium.com/coinmonks/what-are-the-differences-between-ethereum-hyperledger-fabric-and-hyperledger-sawtooth-5d (Visited 15.05.19).

[56] Suprunov, P. Sep 21, 2018. 5 hyperldeger projects in depth. https://medium.com/practical-blockchain/5-hyperledger-projects-in-depth-3d14c41f902b. (Visited 18.04.2019).

[57] 2019. Ibm. https://www.ibm.com.

[58] 2019. https://digitalasset.com/.

[59] 2019. Hyperledger fabric. https://hyperledger-fabric.readthedocs.io.

[60] Goyal, S. 2018. Hyperledger vs corda r3 vs ethereum: The ultimate guide. https://101blockchains.com/hyperledger-vs-corda-r3-vs-ethereum/. (Visited April. 11 2019).

[61] 2019. Visual studio code. https://code.visualstudio.com/.

[62] 2019. Web3 documentation. https://web3js.readthedocs.io/en/1.0/.

[63] 2018. Bootstrap. https://getbootstrap.com/.

[64] 2019. Truffle. https://truffleframework.com/truffle.

[65] 2019. Go ethereum. https://geth.ethereum.org/.

[66] 2019. Ganache. https://truffleframework.com/ganache.

[67] 2019. Node.js. https://nodejs.org.

[68] 2019. Github. https://github.com/.

A Definitions

Node is a participant in the blockchain network. We defined all participants as limited to one physical or virtual machine.

Miner is a node in charge of solving the cryptographic hashes that determine wether a new block should be added to the chain or not.

Ledger is a record of all transactions in a system.

B Source Code

Attached you will find the source code for the prototype.

```
pragma solidity >=0.4.22 <0.6.0;

contract Services{

    struct ticket{
        //the position of a ticket
        uint ticketPos;
        //the id of a user
        uint user_id;
        //the id of game to the ticket
        uint game_id;
        //the price of the ticket, inn kr
        int price;
        //What state the ticket is in
        States state;
    }

    struct game{
        //the id of game
        uint game_id;
        //the id of event organizer
        uint evnetOrganizer_id;
        //name of the home team
        string homeTeam;
        //the name of foreign team
        string foreignTeam;
        //the number of ticket in a game
        uint number_of_tickets;
        //the state of a Games
        GameStates gameState;
        //create a mapping to struct tickets
        mapping(uint => ticket) tickets;
    }

    struct user{
        //the id of the user
        uint user_id;
        //the name og onwer
        string name;
        //the addr of user in ethereum
        address addr;
        //mobile number to user
        int mobile;
        //number of ticket own
        uint ticketOwns;
        //password to the user
        string password;
        //which ticket the user owns
        mapping(uint => ticket) tickets;
    }

    struct eventOrganizer{
        //the id of event organizer
        uint evnetOrganizer_id;
        //the name of event
        string name;
        //the address of event organizer
        address addr;
        //password to the event organizer
        string password;
    }

    //enum which state the ticket is in
    enum States {available, bought, spent, invaild}

    //enum for which state a game is
    enum GameStates{notStarted, ongoing, ended}

    //all Users
    user[] Users;

    //all Games
    game[] Games;

    //all event Organizer
    eventOrganizer[] EventOrganizers;

    //Create event for createGame
    event CreateGame(uint _gameId, uint _evnetOrganizerId, string _homeTeam, string _foreignTeam, uint _tickets, GameStates state);

    //Create event for createTicket
    event CreateTicket(uint _ticketPos, uint _userId,  uint _gameId, int _price, States state);

    //Create event for createUser
    event CreateUser(uint _userId, string  _name, address _addr, int _mobile, uint _ticketOwns, string _password);

    //Create event for createEvent
    event CreateEventOrganizer(uint _evnetOrganizerId, string _name, address _addr, string _password);

    //Create event for buyTickets
    event BuyTickets(uint _userId, uint _gameId, uint _tickets);

    //Create event for buyTicket
    event BuyTicket(uint _userId, uint _gameId, uint _ticketpos, int _price, States state);

    //Create event for TicketState
    event TicketState(uint _userId, uint _gameId, uint _ticketId, States state);

    //Create event for change gameState
    event GameSate(uint _gameId, uint _evnetOrganizerId, string _homeTeam, string _foreignTeam, GameStates state);

    //createEventOrganizer function create a event organizer in EventOrganizers[]
    //@param string memory _name is the name of a event organizer
    //@param address _addr is teh ethereum address to event organizer
    //@return bool, true if was successfull created, false if failed
    function createEventOrganizer(string memory _name, address _addr, uint _evnetOrganizerId, string memory _password) public returns(bool){

        //check if event organizer already exits
        if(checkEventOrganizer(_evnetOrganizerId) && checkAddr(_addr)){
            eventOrganizer memory e = eventOrganizer(_evnetOrganizerId, _name, _addr, _password);      //Create a temporary memory of struct eventOrganizer
            EventOrganizers.push(e);

            emit CreateEventOrganizer(_evnetOrganizerId, _name, _addr, _password);

            return (true);

        } else{

            return(false);
        }
    }

    //checkEventOrganizer function makes sure that address aren't begin used more then one time
    //@param address _addr is the address begin check
    //@returns bool, return true if not begin used, and false if are already used
    function checkEventOrganizer(uint _evnetOrganizerId)public view returns(bool){

        //check if EventOrganizers[] hold something
        if(getCounterEventOrganizer() != 0){

            //go throught all EventOrganizers
            for(uint i = 0; i < getCounterEventOrganizer(); i ++){

                //check if the address is used
                if(EventOrganizers[i].evnetOrganizer_id == _evnetOrganizerId){

                    return(false);
                }
            }
        }
        return (true);
    }

    //checkAddr function makes sure that address aren't begin used more then one time
    //@param address _addr is the address begin check
    //@returns bool, return true if not begin used, and false if are already used
    function checkAddr(address _addr)public view returns(bool){

        //check if EventOrganizers[] hold something
        if(getCounterEventOrganizer() != 0){

            //go throught all EventOrganizers
            for(uint i = 0; i < getCounterEventOrganizer(); i ++){

                //check if the address is used
                if(EventOrganizers[i].addr == _addr){

                    return(false);                    //return false if address is already in used
                }
            }
        }

        //check if Users[] holds something
        if(getCounterUsers() != 0){

            //check every Users[]
            for(uint i = 0; i < getCounterUsers(); i++){

                //if address is the same
                if(Users[i].addr == _addr){

                    return (false);                    //return false, id and addr already begin used
                }
            }
        }

        return (true);                    //return true if address is not begin used
    }

    //getCounterEventOrganizer function counts how many object is in the EventOrganizers[]
    //@return uint of how many object is in the EventOrganizers[]
    function getCounterEventOrganizer() public view returns(uint){
        return EventOrganizers.length;
    }


    //findPosEventOrganizer function finds the position of a event organizer
    //@param uint  _evnetOrganizerId is the the id of a event organizer
    //@return uint of position to evnetOrganizerId in the EventOrganizers[]
    function findPosEventOrganizer(uint _evnetOrganizerId) public view returns(uint){

        //go through ever object in EventOrganizers[]
        for(uint i = 0; i < getCounterEventOrganizer(); i++){

            //check if the userId matches
            if(EventOrganizers[i].evnetOrganizer_id == _evnetOrganizerId){
                return (i);                                                  //return the position
            }
        }
        return 0;                                                            //return 0 if there are no object in EventOrganizers[]
    }

    //getEventOrganizer function gets all the information about a event organizer
    //@param uint _evnetOrganizerId is the id of evnet organizer
    //@return uint of the evnet Organizer Id, string of evnet Organizer name, address of evnet Organizer and password
    function getEventOrganizer(uint _evnetOrganizerId)public view returns(uint, string memory, address, string memory){
        uint pos = findPosEventOrganizer(_evnetOrganizerId);

        return(EventOrganizers[pos].evnetOrganizer_id,
        EventOrganizers[pos].name,
        EventOrganizers[pos].addr,
        EventOrganizers[pos].password);
    }

    //createUser function create a user in user[]
    //@param uint _userId setting the id to user
    //@param string memory _name setting full name
    //@param address _addr is the ethereum address in the network to user
    //@param int _mobile setting mobile number to user
    //@returns bool, return true if the function was successfull
    function createUser(uint _userId, string memory _name, address _addr, int _mobile, string memory _password) public returns(bool){

        //call the function checkUser to check _userId and _addr
        if(checkUser(_userId) && checkAddr(_addr)){

            user memory u = user(_userId, _name, _addr, _mobile, 0, _password); //Create a temporary memory of struct user
            Users.push(u);

            emit CreateUser(_userId, _name, _addr, _mobile, 0, _password);

            return (true);

        } else {

            return (false);
        }

    }

    //checkUser function make sure that id and address aren't begin used more then one time
    //@param unit _userId is the id begin check
    //@param address _addr is the address begin check
    //@returns bool, return true if not begin used, and false if are already used
    function checkUser(uint _userId) public view returns(bool){

        //check if Users[] holds something
        if(getCounterUsers() != 0){

            //check every Users[]
            for(uint i = 0; i < getCounterUsers(); i++){

                //if userId is the same
                if(Users[i].user_id == _userId){

                    return (false);
                }
            }
        }

        return (true);
    }

    //getCounterUser function counts how many object is in the Users[]
    //@return uint of how many object is in the Users[]
    function getCounterUsers() public view returns(uint){
        return (Users.length);
    }

    //findPosUser function finds the position of a user
    //@param uint _userId is the the id of a user
    //@return uint of position to userId in the Users[]
    function findPosUser(uint _userId) public view returns(uint){

        //go through ever object in Users[]
        for(uint i = 0; i < getCounterUsers(); i++){

            //check if the userId matches
            if(Users[i].user_id == _userId){
                return (i);                                         //return the position
            }
        }
        return 0;                                                   //return 0 if there are no object in Users[]
    }

    //Create a game and tickets in Game[]
    //@param uint _gameId is the the id of a game
    //@param string memory _homeTeam are the name of hometeam
    //@param string memory _foreignTeam is the name of the foreignteam
    //@param unit _ticket is the number of tickets in a game, it start on 0
    //@param int _price is the price of a ticket
    //@return true if function was successful
    function createGame(uint _gameId, uint _evnetOrganizerId, string memory _homeTeam,
    string memory _foreignTeam, uint _tickets, int _price)public returns(bool){

        //check if game already exists and event organizer exists
        if(!checkEventOrganizer(_evnetOrganizerId) && checkGame(_gameId)){

            game memory g = game(_gameId, _evnetOrganizerId, _homeTeam, _foreignTeam, _tickets, GameStates.notStarted);       //Create a temporary memory of struct game
            Games.push(g);

            emit CreateGame(_gameId,_evnetOrganizerId, _homeTeam, _foreignTeam, _tickets, GameStates.notStarted);

            createTicket(_gameId, _tickets, _price);                //call the function createTicket for creating the tickets in a game

            return (true);

        } else{
            return(false);
        }
    }

    //checkGame function check if gameId already exits
    //@param uint _gameId is the id of a game
    //@return bool, true if the gameId dont already exits and false if begin used
    function checkGame(uint _gameId) public view returns(bool){

        //check if Games[] hold something
        if(getCountGame() != 0){

            //go thought all the games
            for(uint i = 0; i < getCountGame(); i++){

                //check if gameId already exists
                if(Games[i].game_id == _gameId){
                    return (false);
                }
            }
        }
        return (true);
    }

    //getCounterGame function counts how many object is in the Games[]
    //@return uint of how many object is in the Games[]
    function getCountGame() public view returns(uint){
        return(Games.length);
    }

    //createTicket function create ticket to a game
    //@param uint _gameId is the id of game we want create ticket for
    //@param uint _tickets is the number of tickets creating
    //@param int _price is the price of a ticket
    //@return bool, true if tickets was successfull created, false if failed
    function createTicket(uint _gameId, uint _tickets, int _price)public
    returns(bool){
        uint pos = findPosGame(_gameId);

        //check if the _gameId and _tickets is same in Games[_gameId]
        if(Games[pos].game_id == _gameId && Games[pos].number_of_tickets == _tickets){
            //creating the number of ticket in a game
                for(uint i = 0; i < _tickets; i++){

                    Games[pos].tickets[i] = ticket(i, 0, _gameId, _price,
                    States.available);   //add tickets for tickets in struct game

                    emit CreateTicket(i, 0, _gameId, _price, States.available);
                }

                return (true);

            } else {

                return (false);
            }
        }

    //findPosGames function finds the position of a game
    //@param uint _gameId is the the id of a game
    //@return uint of position to gameId in the Games[]
    function findPosGame(uint _gameId) public view returns(uint){

        //go through ever object in Games[]
        for(uint i = 0; i < getCountGame(); i++){

            //if gameId  is the same
            if(Games[i].game_id == _gameId){
                return (i);                                     //return uint of
                the position
            }
        }
        return 0;                                                // retunr 0,
        there are no object in Games[]
    }

    //getTicket function gets a ticket by it's gameId and ticketPos
    //@param uint _gameId is the id of a game
    //@param uint _ticketId is the id of ticket in the game
    //@return gameId, ticketPos and state to a ticket
    function getTicket(uint _gameId, uint _ticketPos) public view returns(uint,
    uint, uint, States state){
        uint pos = findPosGame(_gameId);

        return(Games[pos].tickets[_ticketPos].ticketPos,
        Games[pos].tickets[_ticketPos].game_id,
        Games[pos].tickets[_ticketPos].user_id,
        Games[pos].tickets[_ticketPos].state);
    }

    //getGame function gets all the information to a game
    //@param uint _gameId is the id of a game
    //@return gameId, eventOrganizersId, homeTeam, foreignTeam, number of ticket and
    gameState
    function getGame(uint _gameId) public view returns(uint, uint, string memory,
    string memory, uint, GameStates state){
        uint pos = findPosGame(_gameId);

        return(Games[pos].game_id,
        Games[pos].evnetOrganizer_id,
        Games[pos].homeTeam,
        Games[pos].foreignTeam,
        Games[pos].number_of_tickets,
        Games[pos].gameState);
    }

    //getEventOrganizerGame function gets all the information about game to evnt organizer
    //@param uint _eventOrganizerId is the id of a event organizer
    //@param uint _gameId is the id of a game
    //@return eventOrganizersId, name, gameId, homeTeam, foreignTeam, number of ticket and gameState
    function getEventOrganizerGame(uint _eventOrganizerId, uint _gameId) public view returns
    (uint,string memory, uint, string memory, string memory, uint, GameStates state){
        uint posG = findPosGame(_gameId);
        uint posU = findPosEventOrganizer(_eventOrganizerId);

        if(EventOrganizers[posU].evnetOrganizer_id == Games[posG].evnetOrganizer_id){
            return(EventOrganizers[posU].evnetOrganizer_id,
            EventOrganizers[posU].name,
            Games[posG].game_id,
            Games[posG].homeTeam,
            Games[posG].foreignTeam,
            Games[posG].number_of_tickets,
            Games[posG].gameState);
        }
    }

    //countEventOrganizerGame function count how many game a evnet organizer is organizing
    //@param uint _eventOrganizerId is the id of a event organizer
    //@param uint _gameId is the id of a game
    //@return uint of many many game a event organizer is organizing
    function countEventOrganizerGame(uint _eventOrganizerId) public view returns(uint){
        uint posU = findPosEventOrganizer(_eventOrganizerId);
        uint counter = 0;

        for(uint i = 0; i < getCountGame(); i++){

            if(EventOrganizers[posU].evnetOrganizer_id == Games[i].evnetOrganizer_id){
                counter++;
            }
        }
        return(counter);
    }

    //getEventOrganizerAddr get the ethereum address to a event organizer
    //@param uint _evnetOrganizerId is the id of event organizer
    //@return address to event organizer
    function getEventOrganizerAddr(uint _eventOrganizerId) public view returns(address){
       uint posU = findPosEventOrganizer(_eventOrganizerId);

       if(EventOrganizers[posU].evnetOrganizer_id == _eventOrganizerId){
           return(EventOrganizers[posU].addr);
       }
    }

    //getTicketAvailable gets number of tickets available in a game
    //@param uint _gameId is the id of a game
    //@return uint of number available in a game
    function getTicketAvailable(uint _gameId) public view returns(uint){
        uint counter = 0;
        uint pos = findPosGame(_gameId);

        //check all tickets in a game
        for(uint i = 0; i < Games[pos].number_of_tickets; i++){

            //if is successful, will add to the counter
            if(Games[pos].tickets[i].state == States.available && Games[pos].tickets[i].game_id == Games[pos].game_id){
                counter++;
            }
        }
        return (counter);
    }

    //buyTicket let a user buy several tickets in game
    //@param uint _userId is the id of the user
    //@param uint _gameId is the the id of game the user want buy ticket
    //@return bool, true if ticket was buyed, false if no ticket available
    function buyTickets(uint _userId, uint _gameId, uint _tickets) public returns(bool){
        uint posG = findPosGame(_gameId);
        uint posU = findPosUser(_userId);


        //check if there any ticket available
        if(getTicketAvailable(_gameId) >= _tickets){

            //go throught number of ticket user wish to buy
            for(uint i = 0; i < _tickets; i++){

                buy(posU, posG);

                }

                emit BuyTickets(_userId, _gameId, _tickets);

                return(true);
        }
        return (false);
    }

    //buy functio is go through all ticket in game and sett user to a tciket (let user buy it)
    //@param uint _posU is the position of the user
    //@param uint _posG is the position of the game
    //@return bool, true if a ticket was successfull bought, false if no ticket available
    function buy(uint _posU, uint _posG)public returns(bool){
        uint posT = Users[_posU].ticketOwns;

        //go throught very ticket in the game
        for(uint i = 0; i < Games[_posG].number_of_tickets; i++){

            //check if the ticket are available
            if(Games[_posG].tickets[i].state == States.available){

                Games[_posG].tickets[i].user_id = Users[_posU].user_id;       //Sett the user id

                Games[_posG].tickets[i].state = States.bought;                //change the state to bought

                Users[_posU].tickets[posT] = Games[_posG].tickets[i];         //cop ticket to Users[].tickets[]

                emit BuyTicket(Users[_posU].user_id,
                Users[_posU].tickets[posT].game_id,
                posT, Users[_posU].tickets[posT].price,
                Users[_posU].tickets[posT].state);

                Users[_posU].ticketOwns++;

                return (true);
            }
        }
        return (false);
    }

    //getUser function gets a ticket from user
    //@param uint _userId is the id of user
    //@param uint _ticketId is the id of ticket
    //@return uint and uint, the userId, ticketId and passord
    function getUser(uint _userId) public view returns(uint, string memory, address, int, uint, string memory){
        uint pos = findPosUser(_userId);

        return(Users[pos].user_id,
        Users[pos].name,
        Users[pos].addr,
        Users[pos].mobile,
        Users[pos].ticketOwns,
        Users[pos].password);
    }

    //getUser_ticket function gets a ticket for user
    //@param uint _userId is the id of user
    //@param uint _ticket is the ticket we get
    //@return uint of userId, string memory of user name, uint ticketId own
    function getUser_ticket(uint _userId, uint _ticket) public view returns(uint, string memory, uint, uint, States State){
        uint pos = findPosUser(_userId);
        return(Users[pos].user_id,
        Users[pos].name,
        Users[pos].tickets[_ticket].user_id,
        Users[pos].tickets[_ticket].game_id,
        Users[pos].tickets[_ticket].state);
    }


    //getTicketsOwn function check how many tickets user owns to a game
    //@param uint _userId is the id of a user
    //@param uint _gameId is the id of a game
    //@return uint counter with how many ticket own to a game
    function get_number_Of_ticket_own(uint _userId, uint _gameId) public view returns(uint){
        uint pos = findPosUser(_userId);
        uint counter = 0;

        //go throught ever own ticket
        for(uint i = 0; i < Users[pos].ticketOwns; i++){

            //check ticket gameId
            if(Users[pos].tickets[i].game_id == _gameId){
                counter++;
                //add to counter if gamedId is same
            }
        }
        return(counter);
        //return counter of how many ticket own in a game
    }

    //gameStart function is the change the state of a game to ongoing
    //@param uint _gameId is the id of a game
    //@return bool, true if was successful, false if failed
    function gameStart(uint _gameId) public returns(bool) {
        uint pos = findPosGame(_gameId);

        if(Games[pos].gameState == GameStates.notStarted){

            Games[pos].gameState = GameStates.ongoing;

            emit GameSate(Games[pos].game_id, Games[pos].evnetOrganizer_id, Games[pos].homeTeam, Games[pos].foreignTeam, Games[pos].gameState);

            return (true);
        }
        return (false);
    }

    //gameEnded function changes the state of a game to ënded
    //@param uint _gameId is the id of a game
    //@return bool, true if was successful, false if failed
    function gameEnded(uint _gameId) public returns(bool) {
        uint pos = findPosGame(_gameId);

        if(Games[pos].gameState == GameStates.ongoing){

            Games[pos].gameState = GameStates.ended;
```

```
                emit GameSate(Games[pos].game_id, Games[pos].evnetOrganizer_id,
                Games[pos].homeTeam, Games[pos].foreignTeam, Games[pos].gameState);

                return (true);
            }
            return (false);
        }

        //vaildateTicket function changes the state of å ticket to spent
        //@param uint _userId is the id of a user
        //@param uint _gameId is the id of a game
        //@return bool, true if ticket was successfull vaildate and change state spent,
        false if failed
        function vaildateTicket(uint _userId, uint _gameId) public returns(bool){
            uint posG = findPosGame(_gameId);
            uint posU = findPosUser(_userId);

            if(!checkGame(_gameId) && Games[posG].gameState == GameStates.ongoing){
                for(uint i = 0; i < Users[posU].ticketOwns; i++){
                    if(Users[posU].tickets[i].game_id == _gameId &&
                    Users[posU].tickets[i].state == States.bought){

                    uint tickPos = Users[posU].tickets[i].ticketPos;

                    Games[posG].tickets[tickPos].state =
                    States.spent;                          //change state to spent

                    Users[posU].tickets[i] = Games[posG].tickets[tickPos];
                    //copy the ticket info to Games[]

                    emit TicketState(_userId, _gameId, i, Users[posU].tickets[i].state);

                    }
                }
                return (true);
            }
             return
             (false);
             //return false

        }

        //invaildTicket function changes the state of å ticket to spent
        //@param uint _userId is the id of a user
        //@param uint _gameId is the id of a game
        //@return bool, true if ticket was successfull invalid and change state invalid,
        false if failed
        function invalidTicket(uint _userId, uint _gameId) public returns(bool){
            uint posG = findPosGame(_gameId);
            uint posU = findPosUser(_userId);

            if(!checkGame(_gameId)){
                for(uint i = 0; i < Users[posU].ticketOwns; i++){
                    if(Users[posU].tickets[i].game_id == _gameId){

                    uint tickPos = Users[posU].tickets[i].ticketPos;

                    Games[posG].tickets[tickPos].state =
                    States.invaild;                          //change state to invalid

                    Users[posU].tickets[i] = Games[posG].tickets[tickPos];
                    //copy the ticket info to Games[]

                    emit TicketState(_userId, _gameId, i, Users[posU].tickets[i].state);

                    }
                }
                return (true);
            }
             return
             (false);
             //return false

        }
    }
```

C Installation guide

Attached you will find an installation guide to get set up for blockchain development.

Guide

Visual Studio Code

The guide uses Visual Studio Code to write code files.

1. Download Visual code on https://code.visualstudio.com/
2. Install visual code on your computer

Extension:

The extensions must be download on Visual Studio Code

- Download **solidity** extension by Juan Blanco

Extra:

This is extra, setting and preferences the guide is using.

- Set spaces and tab to 4, this do by click on bar to down of Visual Studio Code window.
- Open Preferences->setting, then open setting.json and put this code in:

```
{
    "editor.dragAndDrop": false,
    "editor.fontSize": 12,
    "editor.formatOnSave": true,
    "editor.formatOnType": true,
    "editor.formatOnPaste": true,
    "editor.wordWrap": "on",
    "editor.quickSuggestions": {
        "other": true,
        "comments": true,
        "strings": true
    },
    "explorer.confirmDragAndDrop": false,
    "explorer.confirmDelete": false,
    "files.autoSave": "onFocusChange",
    "workbench.colorTheme": "Default Light+",
    "solidity.enabledAsYouTypeCompilationErrorCheck": true,
    "solidity.linter": "solium",
    "solidity.packageDefaultDependenciesContractsDirectory": "",
    "solidity.packageDefaultDependenciesDirectory": "",
    "solidity.validationDelay": 1500
}
```

Go

First requirement to run Go-ethereum is to download Go.

1. Download Go version go1.12 on https://golang.org/dl/
2. Run download package

For Windows:

1. Create GOPATH in environment variables to your go working directory, for example set **GOPATH=c:\Users\%USERNAME%\go** in command (i.e cmd) window or direct in environment variables window
2. Then create GOPATH\bin by set **PATH=%PATH%;%GOPATH%\bin** in cmd window or direct in environment variables window, if bin folders is not in go working directory, create one
3. Create other necessary folders in your go working directory, pkg and src.
4. Structure for your go directory should look like:
    - C:\GOPATH\bin
    - C:\GOPATH\pkg
    - C:`GOPATH\src

For Mac OS:

1. Set GOPATH in environment variables to your go working directory, for example set **export GOPATH=$HOME/go** in cmd window
2. Then create GOPATH\bin by set **export PATH=$PATH:$GOPATH/bin** in cmd window, if bin folders is not in go working directory, create one
3. Create other necessary folders in your go working directory, pkg and src.
4. Structure for your go directory should look like:
    a. C:\GOPATH\bin
    b. C:\GOPATH\pkg
    c. C:`GOPATH\src

Testing go workings correct:

- First test if version is correct, write **go version** in cmd window and output should be package you download, for example go version go1.12 windows/amd64 for windows
- In src folder, create hello.go with the following code:

```
package main

import "fmt"

func main() {
        fmt.Printf("hello, world\n")
}
```

after creating the file, build with go tool by writing **go build** in cmd window while being in the path to GOPATH\src. It will build hello.exe in src folder which to run write simply **hello** in cmd window and output should be: **hello, world**

Go's own tutorial for install: https://nats.io/documentation/tutorials/go-install/

Geth

Download Go Ethereum (i.e geth):

- Download geth version 1.8.23 on https://geth.ethereum.org/downloads/
- Then run package

For Mac OS:

The easiest way and one describe in this guide, is to use Homebrew to install geth

- First download Hombrew by https://brew.sh/ and paste **-e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"** in cmd window
- Then write these to lines for install geth:
    - **brew tap ethereum/ethereum**
    - **brew install ethereum**

Testing go workings correct:

- First test if version is correct, write **geth version** in cmd window and output should be package you download, for example windows should get:

```
Geth
Version: 1.8.23-stable
Git Commit: c942700427557e3ff6de3aaf6b916e2f056c1ec2
Architecture: amd64
Protocol Versions: [63 62]
Network Id: 1
Go Version: go1.11.5
Operating System: windows
GOPATH=C:\Users\%USERNAME%\go
GOROOT=C:\Go\
```

Geth's own tutorial for Mac OS x:
https://github.com/ethereum/go-ethereum/wiki/Installation-Instructions-for-Mac

Ganache

Download Ganache

- Download the installation package on https://truffleframework.com/ganache
- Then run the package

Testing go workings correct:

- Open ganache and create a test blockchain network

Node.Js

Node.Js is need to download other programs the guide uses.

- Download Node.Js on https://nodejs.org/en/
- Run the install

For windows:

- Write **npm install -g windows-build-tools** in cmd window for download the tools

Package:

This packages must be download

- Solidity is code smart contract uses. Download solidity on Node.js by writing **npm install -g solc** in cmd window
- Truffle runs the code for programs in the geth network. Download Truffle by writing **npm install -g truffle** in cmd window
- Web3 is to communicate with the blockchain. Download web3 by writing **npm install -g web3** in cmd window

D Meeting Logs

Since we have had constant communication with both our supervisor and employer, keeping logs of all of this became difficult, but we have logs for most of the physical meetings.

11.01.2019

First meeting with Hao Wang

- Introducing ourselves, talk about our background.
- Quick talk about the project.

28.01.2019

Meeting with Hao Wang

- Discussion about the project plan
- Talk about Gantt-scheme, risk analysis, Work Breakdown Structure

18.02.2019

Meeting with Hao Wang

- Discussed what we should focus on about frameworks
- The structure of the report
- What frameworks we should consider developing on

05.03.2019

Meeting with Hao Wang

- Asked for recommendations regarding setup for Ethereum and Fabric
- Discussed what we should talk to Innit about

20.03.2019

Meeting with Hao Wang

- Talked about structure of chapter 2 and 3

27.03.2019

Meeting with Hao Wang

- Talked about the prototype and what we needed from Innit to start developing
- Planned a meeting with Mariusz Nowostawski for help with developing

05.04.2019

Meeting with Mariusz Nowostawski

- Talked about using a testnet
- Discussed Sky-High and deploying there

- Talked about the prototypes structure
- Talked about Solidity

24.04.2019

Short meeting with Mariusz Nowostawski

- Needed help to connect to Sky-High

02.05.2019

Meeting with Hao Wang

- Talked about report structure
- Discussed tables

14.05.19

Last meeting with Hao Wang

- Questions regarding the use-case diagram and chapter 5 i general
- CoDiscussed correct use of citation
-

E Project Plan

Attached you will find the project plan from the planing period.

Project plan

Andrine Celine Flatby, Marius Lillevik and Ole Bjørn Gran

May 4, 2019

Contents

List of Figures

List of Tables

1 Goals and Boundaries

1.1 Background

Innit is an IT consultant and hosting company. Innit have developed several ticket applications for the Norwegian hockey league, but there are separate applications for each hockey team. They are now interested in renewing this system and developing a common solution for all types of events. Because of this they want to know if this could be developed using blockchain technology and want us to explore this possibility. They want a detailed theoretical report of the topic and a simple prototype as a proof of concept. They are interested in the feasibility of such a system and if there are any existing solutions you could base the system on.

Blockchain is a database like technology consisting of two components[1]: "blocks", representing a number of transnational records and "chain", connecting the blocks together with hash functions. It is best associated with bitcoin, but has been used in different frameworks like[2]: Ripple[3], Ethereum[4], Corda[5] and Hyperledger[6].

1.2 Project goals

- Research Blockchain technology and discuss the feasibility of a ticket system based on this technology.
- Develop a simple prototype to test the system.

1.3 Boundaries

- Time frame: 2.5 months for research and development.
- 3 people working 30 hours a week makes for approximately 1000 man hours.
- No resources for licensing software, will be relying on open source software and frameworks.
- The group is responsible for their own equipment.

2 Scope

2.1 Subject

As shown in the the Work Breakdown Structure Figure 2 on page 9, this project consists of two main goals. The first goal will be researching the blockchain technology, it's capabilities, pros and cons, and different frameworks. The second goal is to develop a prototype system for handling tickets. The prototype consists of two core functionalities; purchase and validate tickets.

2.2 Delimitation

- The solution will not be integrated with any of Innit's existing systems.
- The system does not need to handle multiple events.
- The prototype only needs to include the core features of the system, purchase and validation of tickets.
- The prototype will be completely standalone, no connection to other systems or solutions.

2.3 Question

What is blockchain, how does it work and what can it be used for? How feasible is a ticket system based on blockchain?

3 Project organization

3.1 Responsibilities and roles

Project leader
Marius Lillevik
The Project leader has the responsibility to oversee planed meetings and call for additional meetings if needed. He also has the responsibility to keep the group motivated and help other members who might be struggling throughout the project period.

Head of development
Ole Bjørn Gran
Head of development have the main responsibility to oversee the development phase of the project. Make sure the development process keeps moving forward at all times, make sure that tests are being made for every function and that all source code is being commented.

Head of research
Andrine Celine Flatby
Head of research have the main responsibility to oversee the research phase of project. Ensure each member have a different topic to research and is progressing forward. Also check if the source is credible and can be using in research.

Supervisor
Hao Wang
Supervisor's tasks is to mentoring the group through project. He will also provide assistance with bachelor's thesis.

Employer
Innit
The Employer provides the task for the project. They will also assist our group if needed.

3.2 Routines in the group

Routines

- Regular weekly meetings with both the supervisor and Innit, especially Mondays and Fridays.
- Transcript is to be written after every meeting with supervisor and Innit.
- All research is to be well documented, with detailed notes, links to the references and time stamps for all information gathered.
- The team members will track their time using the free tool Toggl [7].

Rules

- Warn ahead of time if you are unable to meet on time or at all.
- Decisions will be determined as a group, if we're not able to come to an agreement then the Supervisor will be contacted to give his opinion on the issue, then the group will make a final decision.
- The group members commit to finish tasks to agreed time. If a delay should arise, member shall notify the rest of the group.

4 Planning, follow-up and reporting

4.1 Main division of the project

For the first part of the project, all resources will be dedicated to researching the blockchain technology and experimenting with the different frameworks. After the research period we'll pick the framework we find most suitable for the project and start digging deeper into that framework and it's strengths and weaknesses. We'll go into detail about the framework's functionality and capabilities and how it can be utilized as a base for a ticket system. Once the initial technical report is almost finished, we'll start developing a small prototype as a "proof of concept" for the system. The prototype's main focus will be the back-end functionality, with a very basic UI. The main goal of the prototype is just to show that this type of system would work in practice. Once the prototype is finished, only the last touches on the report remains.

Since the majority of this project will focus on researching, and then testing out different solutions, all team members will be actively involved in each step. Therefore we find that applying the scrum model, with 1-2 week sprints will be the most effective. Since each member will get to present their findings once per sprint, we can discuss how to further focus the research based on each member's findings. The prototype portion of the project is more likely to change depending on what we find during the research period, but since the prototype only needs to test the basic parts of the system, it is very likely that it won't take very long to develop it. There is also a good chance that we will find frameworks that we can build the prototype on. As we are using scrum during the research period, and it is suitable for small projects, it will be simplest to utilize scrum for developing the prototype as well. This choice might change depending on what we discover during the research phase or what framework we decide upon, but that is unlikely since scrum is so versatile.

4.2 Plan for status meetings and decision points during the period

Internal group status meeting once per sprint. Plan on meeting with supervisor and Innit once per sprint also, but this might change depending on necessity.

5 Organization of quality assurance

5.1 Documentation, standard use and source code

During the research phase, all sources will be documented using Vancouver style referencing, and for the prototype the code will be commented with a description of the functionality. We will also follow universal standards within coding, since Innit had no standard requirements for us to follow during the project. During development the code will be tracked using Bitbucket [8].

5.2 Tools and equipment

We will be using our own private equipment for this project. This means we will be using the OS that are accessible to us, most likely Windows and Linux. The framework we will been using will be determined in the research phase of project as one of the topics to examine. This also includes which blockchain we are going to been using, but that will be determined later in the project.

Tools beside equipment and information provide in research phase are Bitbucket and Google drive. Bitbucket is the repository we will be using for storing and backup the source code[8] and Google drive will be used to store all of our findings during the research phase.

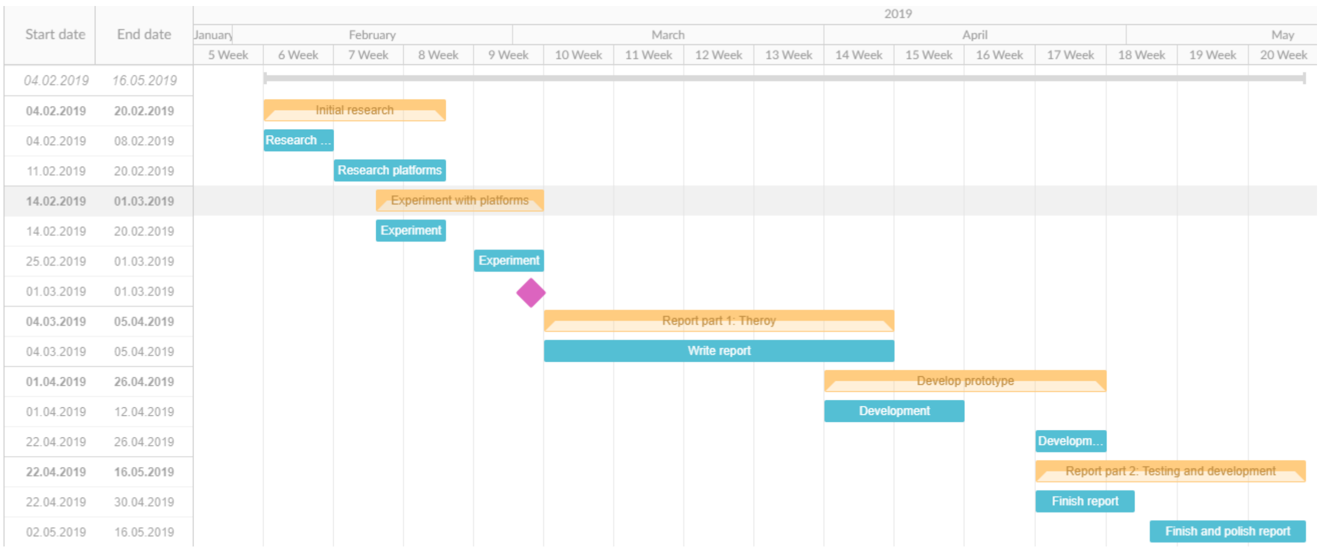
5.3 Risk analysis

Nr	Issue	Description	Consequences	Probability	Actions
1	Loss of Personnel	A team member falls ill	High	Low	Be prepared for periods of higher workloads
2	Failure to find a viable framework	Unable to find a suitable framework to base the system on	Very high	Very low	Change the scope of the project
3	Unfinished prototype	The prototype is not finished by the deadline	Medium	Medium	Allocate more time for development
4	Failure to deliver within deadlines	Unable to keep deadlines can lead to an unfinished report	High	Low	Work overtime if we start to fall behind schema
5	Loss of data	Valuable data is lost	High	Low	Take regular backups

Table 1: Risk analysis table

6 Plan

6.1 Gantt



Start date	End date	
04.02.2019	*16.05.2019*	
04.02.2019	**20.02.2019**	Initial research
04.02.2019	08.02.2019	Research ...
11.02.2019	20.02.2019	Research platforms
14.02.2019	**01.03.2019**	Experiment with platforms
14.02.2019	20.02.2019	Experiment
25.02.2019	01.03.2019	Experiment
01.03.2019	01.03.2019	
04.03.2019	**05.04.2019**	Report part 1: Theroy
04.03.2019	05.04.2019	Write report
01.04.2019	**26.04.2019**	Develop prototype
01.04.2019	12.04.2019	Development
22.04.2019	26.04.2019	Developm...
22.04.2019	**16.05.2019**	Report part 2: Testing and development
22.04.2019	30.04.2019	Finish report
02.05.2019	16.05.2019	Finish and polish report

Figure 1: Gantt diagram of project

The gantt diagram, figure 1 shows how much time we have dedicated to the different steps of the project.

6.2 Work Breakdown Structure

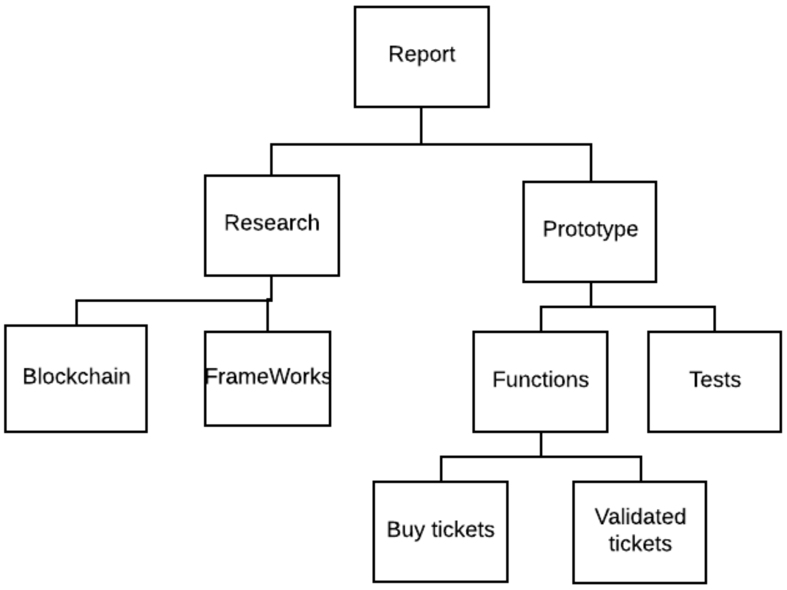


Figure 2: WBS model of project

In figure 2, we have broken down the elements of the report; research and prototype. These topics is main elements of report, and the sub-elements are tasks/topics we need to do.

7 References

[1] Jon Martindale. What is a blockchain[Internett] 08.3.18 [Accessed 28.01.2019] Available from: https://www.digitaltrends.com/computing/what-is-a-blockchain/

[2] Chaonian Guo, Shenglan Ma, Hao Wang, Shuhan Cheng, Tongsen Wang. LoC: Poverty Alleviation Loan Management System based on Smart Contracts. Division of Sci. anf Tech, Department of ICT and Natural Sci. [Accessed 28 January 2019]

[3] Ripple[Internett]. [Accessed 2 January 2019]. Available from: https://ripple.com/

[4] G. Wood, Ethereum: A secure decentralised generalised transaction ledger, ethereum Project Yellow Paper.

[5] M. Hearn, Corda - a distributed ledger, corda Technical White Paper.

[6] Hyperledger[Internett].[Accessed 28. January 2019]
Available from: https://www.hyperledger.org/

[7] Toggl[Internett]. [22. January 2019] Available from: https://toggl.com/

[8] Bitbucket[Internett]. [28. January 2019] Available from: https://bitbucket.org

F Project Agreement

Attached you will find the signed project agreement.

NTNU

Norges teknisk-naturvitenskapelige universitet

Vår dato | Vår referanse

Prosjektavtale

mellom NTNU Fakultet for informasjonsteknologi og elektroteknikk (IE) på Gjøvik (utdanningsinstitusjon), og

Innit ved Øyvind B Tangen

(oppdragsgiver), og

Andrine celine flatby, Ole Bjørn Gran
marius lillevik

(student(er))

Avtalen angir avtalepartenes plikter vedrørende gjennomføring av prosjektet og rettigheter til anvendelse av de resultater som prosjektet frembringer:

1. Studenten(e) skal gjennomføre prosjektet i perioden fra 4/02/19 til 20/05/19

Studentene skal i denne perioden følge en oppsatt fremdriftsplan der NTNU IE på Gjøvik yter veiledning. Oppdragsgiver yter avtalt prosjektbistand til fastsatte tider. Oppdragsgiver stiller til rådighet kunnskap og materiale som er nødvendig for å få gjennomført prosjektet. Det forutsettes at de gitte problemstillinger det arbeides med er aktuelle og på et nivå tilpasset studentenes faglige kunnskaper. Oppdragsgiver plikter på forespørsel fra NTNU å gi en vurdering av prosjektet vederlagsfritt.

2. Kostnadene ved gjennomføringen av prosjektet dekkes på følgende måte:
   - Oppdragsgiver dekker selv gjennomføring av prosjektet når det gjelder f.eks. materiell, telefon/fax, reiser og nødvendig overnatting på steder langt fra NTNU på Gjøvik. Studentene dekker utgifter for ferdigstillelse av prosjektmateriell.
   - Eiendomsretten til eventuell prototyp tilfaller den som har betalt komponenter og materiell mv. som er brukt til prototypen. Dersom det er nødvendig med større og/eller spesielle investeringer for å få gjennomført prosjektet, må det gjøres en egen avtale mellom partene om eventuell kostnadsfordeling og eiendomsrett.

3. NTNU IE på Gjøvik står ikke som garantist for at det oppdragsgiver har bestilt fungerer etter hensikten, ei heller at prosjektet blir fullført. Prosjektet må anses som en eksamensrelatert oppgave som blir bedømt av intern og ekstern sensor. Likevel er det en forpliktelse for utøverne av prosjektet å fullføre dette til avtalte spesifikasjoner, funksjonsnivå og tider.

4. Alle bacheloroppgaver som ikke er klausulert og hvor forfatteren(e) har gitt sitt samtykke til publisering, kan gjøres tilgjengelig via NTNUs institusjonelle arkiv hvis de har skriftlig karakter A, B eller C.

Tilgjengeliggjøring i det åpne arkivet forutsetter avtale om delvis overdragelse av opphavsrett, se «avtale om publisering» (jfr Lov om opphavsrett). Oppdragsgiver og veileder godtar slik offentliggjøring når de signerer denne prosjektavtalen, og må evt. gi skriftlig melding til studenter og instituttleder/fagenhetsleder om de i løpet av prosjektet endrer syn på slik offentliggjøring.

Den totale besvarelsen med tegninger, modeller og apparatur så vel som programlisting, kildekode mv. som inngår som del av eller vedlegg til besvarelsen, kan vederlagsfritt benyttes til undervisnings- og forskningsformål. Besvarelsen, eller vedlegg til den, må ikke nyttes av NTNU til andre formål, og ikke overlates til utenforstående uten etter avtale med de øvrige parter i denne avtalen. Dette gjelder også firmaer hvor ansatte ved NTNU og/eller studenter har interesser.

5. Besvarelsens spesifikasjoner og resultat kan anvendes i oppdragsgivers egen virksomhet. Gjør studenten(e) i sin besvarelse, eller under arbeidet med den, en patentbar oppfinnelse, gjelder i forholdet mellom oppdragsgiver og student(er) bestemmelsene i Lov om retten til oppfinnelser av 17. april 1970, §§ 4-10.

6. Ut over den offentliggjøring som er nevnt i punkt 4 har studenten(e) ikke rett til å publisere sin besvarelse, det være seg helt eller delvis eller som del i annet arbeide, uten samtykke fra oppdragsgiver. Tilsvarende samtykke må foreligge i forholdet mellom student(er) og faglærer/veileder for det materialet som faglærer/veileder stiller til disposisjon.

7. Studenten(e) leverer oppgavebesvarelsen med vedlegg (pdf) i NTNUs elektroniske eksamenssystem. I tillegg leveres ett eksemplar til oppdragsgiver.

8. Denne avtalen utferdiges med ett eksemplar til hver av partene. På vegne av NTNU, IE er det instituttleder/faggruppeleder som godkjenner avtalen.

9. I det enkelte tilfelle kan det inngås egen avtale mellom oppdragsgiver, student(er) og NTNU som regulerer nærmere forhold vedrørende bl.a. eiendomsrett, videre bruk, konfidensialitet, kostnadsdekning og økonomisk utnyttelse av resultatene. Dersom oppdragsgiver og student(er) ønsker en videre eller ny avtale med oppdragsgiver, skjer dette uten NTNU som partner.

10. Når NTNU også opptrer som oppdragsgiver, trer NTNU inn i kontrakten både som utdanningsinstitusjon og som oppdragsgiver.

11. Eventuell uenighet vedrørende forståelse av denne avtale løses ved forhandlinger avtalepartene imellom. Dersom det ikke oppnås enighet, er partene enige om at tvisten løses av voldgift, etter bestemmelsene i tvistemålsloven av 13.8.1915 nr. 6, kapittel 32.

12. Deltakende personer ved prosjektgjennomføringen:

NTNUs veileder (navn): Hao Wang

Oppdragsgivers kontaktperson (navn): Øyvind R. Tangen

Student(er) (signatur): Andrine Celine Flatby dato 29/01/19

ole Bjørn Gran dato 29/01/19

marius Lillevik dato 29/01/19

________________ dato ________

Oppdragsgiver (signatur): [illegible] dato 29/01/19

Signert avtale leveres digitalt i Blackboard, rom for bacheloroppgaven.
Godkjennes digitalt av instituttleder/faggruppeleder.

Om papirversjon med signatur er ønskelig, må papirversjon leveres til instituttet i tillegg.
Plass for evt sign:

Instituttleder/faggruppeleder (signatur): ________________ dato ________

NTNU
Kunnskap for en bedre verden