UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Vladyslav Kopylash

An Ethereum-based Real Estate Application with Tampering-resilient Document Storage

Master's Thesis (30 ECTS)

Supervisor(s): Luciano García-Bañuelos

Tartu 2018

An Ethereum-based Real Estate Application with Tampering-resilient Document Storage

Abstract:

Blockchain and smart contracts technology enables changes in many industries providing a distributed platform for running decentralized applications. Many companies want to adopt smart contracts technology and use it in their business processes to boost the performance. In this work we present the case study of the real estate company in Singapore that partially integrated blockchain into one of its processes, but wants to move the whole process to the smart contract. After modelling and analyzing their business processes, we create a proof-of-concept of a hybrid system that integrates Ethereum smart contract and traditional web application. Also, we introduce the concept of tampering-resilient document storage and extend the baseline solution to add support for such storage that is based on IPFS. Finally, we summarize and discuss the potential problems that can be met during the development of a blockchain-based application. We provide potential solutions and describe their implications.

Keywords:

blockchain, real estate business, Ethereum, smart contracts, IPFS, Ethereum Swarm

CERCS: P170 – Computer science, numerical analysis, systems, control

Ethereumi baasil kinnisvara rakendus koos võltsimiskindla dokumendihoidlaga

Lühikokkuvõte:

Plokiahela ja nutilepingute tehnoloogia on võimelised muutma mitmeid tööstusharusid pakkudes hajutatud platvormi detsentraliseeritud rakenduste arendamiseks. Seejuures soovivad mitmed ettevõtted nutilepinguid kasutada äriprotsesside tõhustamiseks. Käesolevas töös me esitleme juhtumiuuringut Singapuris tegutseva kinnisvara rendiga tegeleva ettevõtte kohta, mis integreeris plokiahela ühte oma protsessidest, kuid soovib kogu protsessi nutilepingusse tõsta. Pärast ettevõtte äriprotsesside modelleerimist ning analüüsimist loome me pilоottarkvara, mille arhitektuur on hübriidne - Ethereumi nutileping integreeritakse traditsioonilisse tsentraliseeritud veebirakendusse. Peale selle tutvustame me võltsimiskindla dokumendihoidla põhimõtet ning lisame selle IPFS näitel pilootprojekti lahendusse. Viimaseks me arutleme potentsiaalsete tüüpprobleemide üle, mis võivad plokiahela rakenduse arendamisel tekkida, pakume võimalikke lahendusi ning kaalume nende tagajärgi.

Võtmesõnad:

plokiahel, kinnisvara ettevõte, Ethereum, nutilepingud, IPFS, Ethereum Swarm

CERCS: P170 – Arvutiteadus, arvanalüüs, süsteemid, kontroll

Table of Contents

List of figures

List of Abbreviations

Abbreviation	**Meaning**
IPFS	Inter Planetary File System
DAO	Decentralized Autonomous Organization
API	Application Programming Interface
BPMN	Business Process Model and Notation
EVM	Ethereum Virtual Machine
ABI	Application Binary Interface
JSON	JavaScript Object Notation
UML	Unified Modelling Language
CEO	Chief Executive Officer
HDB	Housing and Development Board
DIY	Do It Yourself
POC	Proof of Concept
UI	User Interface
UX	User Experience
ORM	Object Relational Mapping
SQL	Structured Query Language
PDF	Portable Document Format
WWW	World Wide Web
DDOS	Distributed Denial of Service
HTTP	HyperText Transfer Protocol
DHT	Distributed Hash Table
ICO	Initial Coin Offering
ADT	Abstract Data Types

1 Introduction

Blockchain is said to be one of the most significant technologies of the last decade, enabling business transformations in almost every domain. But what is blockchain? It is referred to as a distributed ledger, append-only database shared between different nodes in the network. Ledgers were used a long time ago before blockchain appeared. Each sheet had a unique number, so one could not remove or replace the data in the ledger. Periodical checks of the ledger state enabled transparency in operations preventing fraud and tampering. Besides classical ledgers there were also digital ones. Nevertheless, blockchain brought everything to a new level. Traditional and even digital ledgers could still be corrupted or lost. Blockchain technology in its turn presented a shared distributed ledger that is fault tolerant due to its distributed nature and immutability (what is satisfied by cryptographic algorithms built into blockchain). It means that everybody in the network holds own copy of the ledger and nobody could change the data without notifying the rest of the participants.

Blockchain, as we know it today, seen the world in 2008 when Satoshi Nakamoto published a paper "Bitcoin: A Peer-to-Peer Electronic Cash System[6] Therefore, many people associate blockchain with Bitcoin as two equal things. It is not true. On the one hand it is a cryptocurrency, on the other it is a software that implements a blockchain using cryptographic protocol to implement a decentralized consensus algorithm on the top of a peer-peer system. Bitcoin allows almost instant worldwide transfers of money with low commission. With time people started thinking about some automation or contracts over Bitcoin platform. For example, releasing money deposits only after specific action happened. Unfortunately, Bitcoin was initially designed as a cash transfer system, limiting the possibilities of users to a specific use-case.

In 2015 an alternative solution was launched - Ethereum - a decentralized platform built on top of a custom blockchain that runs decentralized applications (smart contracts). Ethereum provided a Turing complete language, called Solidity, to write smart contracts. It means that with Ethereum people can achieve results that were extremely complicated or impossible to implement with Bitcoin. It is because these platforms were created for different purposes: Bitcoin - for money transfer, while Ethereum - to create and run decentralized applications, even considering the fact that Ethereum has its token traded as cryptocurrency (tokens are used to incentivize the computation power needed to run smart contracts). Nowadays, Ethereum is a leading platform for decentralized applications. Hence, in this paper, we will mainly focus on its ecosystem and Solidity to create smart contracts for our application.

There is a misconception among many people that smart contracts are entirely self-driven, automated applications that can create so-called decentralized organizations (DAOs) as they are running on top of fault tolerant, fraudless blockchain. In the current state of the art smart contracts have a lot of limitations and potential at the same moment. It is a challenge to apply them to particular use cases and fully use the benefits they can provide.

One can perceive smart contracts as applications that can read information from blockchain and store some data in it. Smart contracts cannot perform external calls. For instance, if smart contract operations depend on the weather data in a certain region, it is not possible for it to call some external weather API and retrieve the needed information. This operation is not persistent. Other nodes that will try to verify this transaction in blockchain must call the same API and check that the data is valid, but it may happen that data will be different between calls. Therefore, such options are restricted for smart contracts. This problem is solved by oracles[1] in Ethereum - special trusted smart contracts that represent a data feed

[1] https://blockchainhub.net/blockchain-oracles/

from the external world (e.g., posting a weather data into blockchain every hour), so any other contract can rely on it.

There is another big challenge on the way towards decentralized organizations. Many businesses are regulated by governments and one need to do many checks to perform an action. Moreover, those checks cannot be automated because authorities do not digitalize them. Problem resolutions require manual actions and a significant amount of legal information is still on paper or in internal electronic systems not exposed to the world.

Challenges mentioned above can be tackled with hybrid solutions that benefit from both approaches, giving the possibility to overcome the limitations of smart contracts and maintaining their key features like fault tolerance, fraud prevention and autonomy.

1.1 Problem Statement

In this paper, we will focus on one particular domain, real estate business, given a case of an undisclosed real estate company located in Singapore. The company claims to be the first blockchain driven real estate platform in Singapore. With a trend of blockchain in 2017, they have already integrated rental contract signing with blockchain, storing hashes of signed documents in it. However, now they want to entirely migrate some of their core business processes to blockchain. Therefore, we are presented with a problem to apply technology to a case study.

1.2 Contribution

We performed a case study about real estate company in Singapore by modelling and analyzing its business processes. Based on the analysis we decided which of them can be migrated to the blockchain and which should stay outside of it. We developed a proof-of-concept hybrid application that reflects the modeled business processes. It can be referred to as a baseline solution. In addition, we extended the baseline solution with an integration of tampering-resilient document storage to store property rental agreements. We analyzed and compared the IPFS and Ethereum Swarm as a basis for the document storage. Finally, we discussed problems and challenges discovered during the development phase. We presented possible solutions, patterns and described their implications.

1.3 Content

This paper is divided into five chapters. Chapter 2 covers all necessary background needed to read this paper: blockchain, Ethereum, smart contracts and BPMN. Chapter 3 describes the business processes of a real estate company and presents the domain model that will be used in the proof-of-concept application. In Chapter 4 we discuss and define which parts of the modeled business processes can be moved to the blockchain. Then, we design and implement a proof-of-concept hybrid application that reflects the business processes modeled in Chapter 3. We cover the technicalities related to smart contracts, their deployment and integration with a traditional web application. Additionally, as a part of Chapter 4, we discuss a tampering-resilient document storage. We consider IPFS and Ethereum Swarm as two possible options, compare them and integrate one of this solutions into our hybrid application. In Chapter 5 we discuss the challenges we faced building a hybrid application, our decisions, possible solutions and their implications.

2 Background

In this chapter, we will provide all the necessary background needed for further reading of this work. We will briefly describe blockchain and Ethereum. Then we will cover Solidity as a smart contract programming language and provide an example of a smart contract. Finally, we will introduce Business Process Model and Notation as a tool for capturing the business processes.

2.1 Blockchain

Blockchain is a distributed append-only database that operates in a peer-to-peer network. It means that every peer in the network owns the full or partial copy of the database [2, 3]. The distributed architecture of the blockchain increases fault tolerance, even if some nodes will be removed from the network, it still can operate in a usual way. If somebody wants to change something in blockchain he creates a transaction that is broadcasted to all nodes and in the network, is verified and only then appended to the blockchain. It takes some time, so transactions do not come in the order they were generated by users. It may happen that in two different points in the network the same transaction is created twice, which is called "double-spend attack" in Bitcoin [5]. Due to distributed nature of blockchain it should have a mechanism that arranges the order of incoming transactions. Usually transactions are grouped in blocks and considered to happen at the same moment of time. However, in this setting different nodes can create blocks consisting of different set of transactions and add it to blockchain, so which block should be considered as the last one and valid? Hence, blockchain protocol introduces a concept called "Proof of Work" - a special mathematical puzzle one need to solve to append the block to the blockchain [4]. It is computationally hard and requires a node to spend some time and resources to solve it. It is a "right" to participate in blockchain. Since not every user is interested in solving the mathematical puzzle to add the transaction to the blockchain, but network requires such "workers" to operate, an incentive was introduced to engage those "workers". In general, this process is called mining and nodes that do a Proof of Work to add block to the chain are called miners. Miners are the crucial part of the blockchain ecosystem as they create a capability of processing certain number of transactions per minute.

Another important aspect of blockchain system is that every user maintains a pair of keys that verify ownership and allow to transfer units of account or digital tokens to other users. This is accomplished through asymmetric cryptography, when one key is public and second is private, known only to its holder. The former is shared with the network and is used as a personal address to which other users can send digital assets, whereas the latter one is used to create a unique digital signature and create transaction on the blockchain. Public-private key schema ensures the validity of ownership, because only the person who holds the private key can create a valid digital signature and create transaction to transfer value over the network. In addition, this schema allows users to operate as anonymous entities, not exposing their real identities. High fault tolerance and consensus mechanism enable blockchain to operate in anonymized untrusted environment where ownership and value transfer is verified by the system.

2.2 Ethereum

Ethereum is a decentralized platform that runs smart contracts: applications that run exactly as programmed without any possibility of downtime, censorship, fraud or third-party interference [7]. These apps run on a custom built blockchain, an enormously powerful shared global infrastructure that can move value around and represent the ownership of property.

The runtime environment for smart contract is Ethereum Virtual Machine. It is completely isolated, that means that smart contract code has no access to network, file system or any other process. Smart contracts have limited access to other contracts running in Ethereum.

Accounts

There are two types of Ethereum accounts: external accounts and contract accounts. Both types share the same address space and are treated equally by Ethereum Virtual Machine. External accounts are controlled the public-private key pairs used by humans to access the accounts. Contract accounts are controlled by the smart contract code that is stored in this address. The address of external account is determined from the public key whereas contract address is created upon deployment and is derived from the creator address. Every account has a persistent key-value store. Also, every account has balance that can be changed by sending transactions that include Ether - a cryptocurrency backed by Ethereum platform.

Transactions

A transaction is a message sent from one account to another. It can include payload and Ether. If the target account stores smart contract code, that code is executed with the payload provided as input parameters. If the target account is the zero-account (has address 0), the transaction triggers the creation of new contract. The payload of such transaction should be a EVM bytecode and is executed. The execution output is permanently stored as the smart contract code. Therefore, to create a smart contract transaction should contain not the actual code, but the code that will return the smart contract code.

Gas

Gas is a special unit of measure which purpose is to limit the resources needed to execute the smart contract code. While the EVM executes the code, the gas is gradually depleted according to specific rules. To send a transaction, certain amount of Ether must be paid for the gas consumed by the execution. The creator of the transaction defines the gas price, because he needs to pay the following amount in Ether: gas amount multiplied by gas price. Transactions with higher gas price get higher priority in the execution queue. It is interesting to note that this payment should be made upfront with the transaction itself. If there is any gas left, its cost will be refunded. If the amount of consumed gas exceeded, then the exception is triggered and all modifications are reverted to the previous state. Therefore, it is necessary to wisely plan the amount of gas consumed by a smart contract, because in the latter case gas will not be refunded.

Storage, memory and stack

Every account has a persistent memory area which is called **storage**. Storage is a key-value store. Both key and value is 256-bit word. It is not possible to enumerate storage from the contract. It is costly to read from the storage. Write operations to storage are more expensive than read operations. A contract has access to only its own storage.

Another memory area is called **memory**. For each message call smart contract receives a clear instance of memory. Memory is linear and can be addressed in a byte level. Read operations are limited to the size of 256 bits. Write operations can be either 8 bits or 256 bits wide. When memory is expanded it adds and offset equal to 256 bits and the gas for extension must be paid. Memory cost increases quadratically.

The Ethereum Virtual Machine is a stack machine, so all computations are performed on a memory area called **stack**. It has maximum size of 1024 elements and stores words of 256 bits. Access to stack is limited to 16 top elements. It is not possible to access arbitrary elements without first removing the top of the stack.

Message calls

Smart contracts can call other contracts or send Ether to external accounts with help of message calls. Message calls are similar to transactions: they have source, target, payload, Ether, cost gas and return data. A called contract will receive cleared memory instance and a access the payload - it is provided in a separate memory area called **calldata**. Call are limited to the depth of 1024 because of the EVM stack peculiarities.

There is a specific type of a message call - a **delegatecall**. It allows to execute target address code in the context of the calling contract. Using delegate calls contracts can dynamically load code from a different address at runtime. It makes possible to implement libraries for smart contracts.

Logs

Ethereum provides a way to store logs about transactions. The information is stored in a specially indexed data structure. Smart contracts cannot access the logs. However, logs can be accessed from outside the blockchain in an efficient and cryptographically secure way. Therefore, peers in the network can check the logs without downloading the whole blockchain.

2.3 Solidity

Smart contracts are developed using language called Solidity[2]. It is statically typed, supports inheritance, libraries and complex user-defined types. Solidity was influenced by C++, Python and JavaScript and is designed specifically for the Ethereum Virtual Machine. An example of a smart contract written in Solidity is given in Appendix I.

Each file is structured in the following way. It is obligatory to define pragma version that restricts the version of the compiler to be used. Pragma version supports semantic versioning. Solidity provides means for importing other source files. Syntax of import statement is very similar to JavaScript ES6. Imports, if present, are followed by the smart contract declaration. After compilation compiler creates Application Binary Interface for a contract. ABI is a set of metadata that describes how to interact with a contract. ABI is usually stored in a JSON format.

2.3.1 Contract Structure

Contracts in Solidity are similar to classes in object-oriented languages. Each contract can contain declarations of state variables, functions, function modifiers, events, struct types and enum types. Contracts can also extend other contracts.

State variables

State variables are values that permanently stored in contract storage.

```
pragma solidity ^0.4.23;

contract Ballot {
    address public chairperson; // State variable
    // ...
}
```

[2] https://solidity.readthedocs.io/en/v0.4.23/

Functions

Functions are the executable units of code within a contract.

```
pragma solidity ^0.4.23;

contract Ballot {
  function vote(uint proposal) public { // Function
    // ...
  }
}
```

There are four visibility types for functions and state variables: external, public, internal, private. Public is default. For state variables external is not possible, default is internal. External functions are part of the contract interface. They can be called from other contracts and via transactions, they cannot be called internally. Public functions can be called both externally and internally, by default getter is generated for public state variable. Internal functions and state variables can only be used within a contract and its successors. Private functions and state variables are visible only to the contract where they are defined.

Function modifiers

Function modifiers are very similar to the concept of decorator. They wrap up the function and change its behavior. Form the implementation point of view function modifiers are very similar to macros. Wrapped function code is pasted into a placeholder in function modifier.

```
pragma solidity ^0.4.23;

contract Purchase {
    address public seller;

    modifier onlySeller() { // Modifier
        require(
            msg.sender == seller,
            "Only seller can call this."
        );
        _;
    }

    function abort() public onlySeller { // Modifier usage
        // ...
    }
}
```

Events

Events provide an interface to the EVM logging feature. Each event is logged and can be accessed outside of blockchain later.

```
pragma solidity ^0.4.23;

contract SimpleAuction {
    event HighestBidIncreased(address bidder, uint amount); // Event

    function bid() public payable {
        // ...
        emit HighestBidIncreased(msg.sender, msg.value); // Triggering event
    }
}
```

Struct types

Structs allow to define custom type variables by grouping different fields.

```
pragma solidity ^0.4.23;

contract Ballot {
    struct Voter { // Struct
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }
}
```

Enum types

Enumerations allow to create custom types with finite set of predefined values.

```
pragma solidity ^0.4.23;

contract Purchase {
    enum State { Created, Locked, Inactive } // Enum
}
```

2.3.2 Types

Solidity is a statically typed language. Solidity has a rich type system, providing basic types which can be combined to create complex types.

Value types

The variables of the value types are always passed by value, i.e. are copied when they passed as function arguments or in variable assignments. The following types are value types:

- Booleans
- Integers - signed and unsigned integers of different sizes from `int8` to `int256`
- Fixed point numbers - not fully supported in version 0.4.23, can be declared, but not assigned
- Address - holds 20 bytes' value
- Fixed-size byte arrays
- Address literals
- Rational and Integer literals
- String literals
- Hexadecimal literals
- Enums
- Function types

Reference types

Complex types that do not always fit into 256 bits need more precision in operations because copying them is expensive. Also, reference types have an additional attribute called data location. It defines whether variable of this type is stored in memory or storage. Data locations influence the assignments behavior (creating independent copy or not). There are several reference types in Solidity: dynamic arrays, structs and mappings.

Mapping type can be considered as a hashtable that is built in the way that every possible key exists and points to the default value which byte representation is zeros. The key in the mapping can be of any type except other mapping, dynamic array, contract, enum or struct, while value can be of any type including other mappings.

```
pragma solidity ^0.4.23;

contract PropertyEnlistment {
    mapping(string => Offer) tenantOfferMap;
}
```

For further reading and advanced topics, the official Solidity reference[2] should be checked.

2.4 Business Process Model and Notation

Business Process Model and Notation is the standard notation to capture business processes [12]. As well as UML, which can be more familiar to the reader, BPMN is developed and standardized by Object Management Group. While UML is usually used for the modelling and analysis of information systems, BPMN is used to analyze organization from the process-oriented perspective. BPMN composed of many different elements which makes it a powerful and flexible tool for visualizing processes. The notation elements in BPMN can be separated into four groups.

Events

There are different types of events: Start event, End event, and intermediate events as shown on Figure 1. BPMN event types

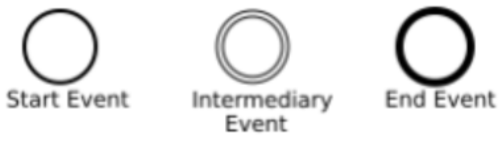


Figure 1. BPMN event types

Start event represents the beginning of the process or message that triggers a new process instance. There can be only one *Start event* in the process model. *End event* refers to the end of the process instance. There also can be only one *End event* in the process model. Intermediate events occur during process execution and can change the execution flow. Different types of intermediate events are presented on Figure 2.

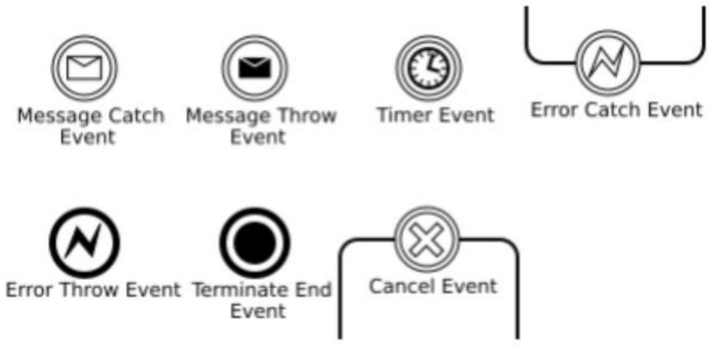


Figure 2. BPMN intermediate events

Most commonly used types of intermediate events are *Message events*, *Timer events*, and *Error events*. *Message events* represent information exchange between entities. *Timer events* can either add a delay or denote that something happens at a specific point in time. *Error events* change the usual execution flow of the process. Also, events can be divided into two categories: throwing and catching. Throwing events are emitted from the process and should be caught at a higher level. Catching events work like handlers for thrown events.

Activities

Activities can be divided into two types: simple and compound. *Task* is a simple atomic activity which cannot be broken down. It is a core element of BPMN and represents an action that should be performed. *Sub-process* shows a compound activity that consists of a group of smaller tasks related to each other. Usually, *Sub-process* encapsulates a certain part of business logic either to reuse it in another place or to make the main process cleaner and simpler to read. As shown on Figure 3, activities, both *Tasks* and *Sub-processes,* can have such attributes as multi-instance and loop. Multi-instance attribute means that several instances of same activity can be executed in parallel. Loop attribute means that activity will be executed in a sequential loop until the exit conditions will be met.

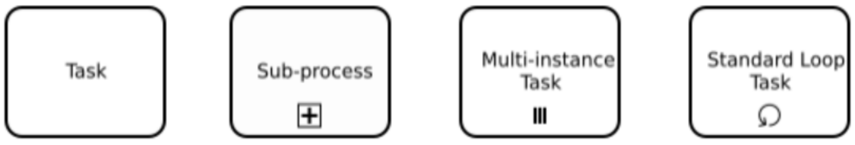


Figure 3. BPMN activities

Gateways

Gateways are used when the process is not executed sequentially. It is possible to implement branching and merging of the process execution paths with the help of gateways. Based on the number of incoming and outgoing sequences, gateways can be divided into two types: split gateways and join gateways. Split gateways divide the execution path into multiple subsequent alternative or concurrent paths. Join gateways, in their turn, merge and synchronize several execution paths into a single one. BPMN gateways are presented on Figure 4.

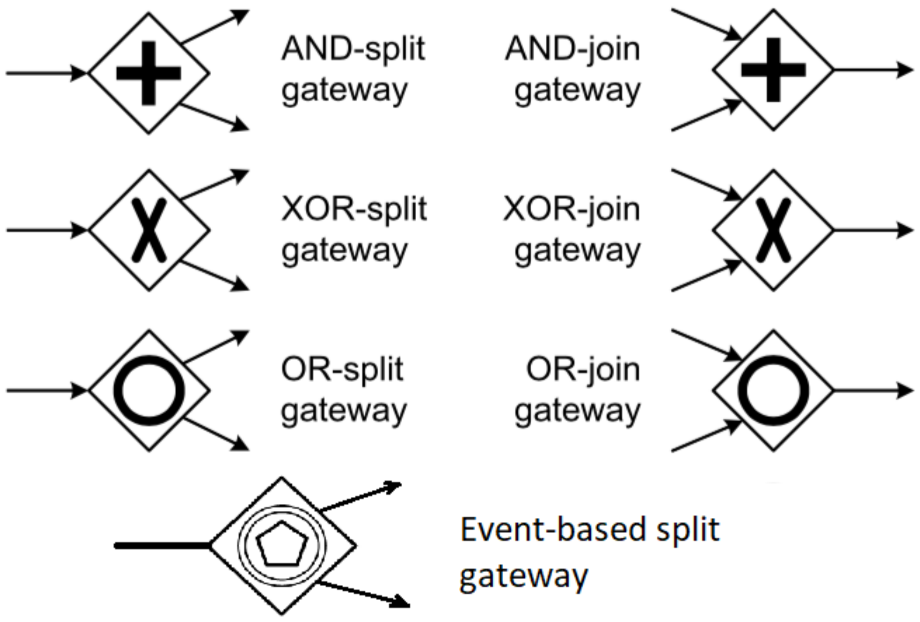


Figure 4. BPMN gateways

XOR-gateway selects one and only one path from all possible alternatives. *AND-gateway* allows multiple paths to be executed simultaneously and synchronizes the execution flow, so all of them should be executed eventually. *Event-based gateway* is very similar to *XOR-gateway*. The only difference is that after the split the next element should be an event. Also, this split gateway does not have a corresponding join gateway. *OR-gateway* allows executing one or more outgoing paths non-exclusively. It will wait until all active paths will finish execution and then merge them. It is not necessary for all outgoing paths to be completed.

Other types of BPMN elements

The process in BPMN is a sequence of elements that are connected by *Sequence flows*. They define the execution order. Sometimes, *Sequence flow* may have a condition attached, after an *XOR-gateway*, for example. This condition determines which path should be followed during the execution of the process. It is necessary to specify a default sequence flow for the case when all conditions return false. Visual representation of *Sequence flow* is shown on Figure 5.

Message flows show the information streams between entities in the model. They can connect separate processes or process and a black-boxed actor. Usually represented as a dashed line with the arrow at the end pointing the direction in information exchange, as shown on Figure 5.

Data artifacts show the information objects required and produced by each activity on each step of the process execution. The most common data artifact is *Data object*. It represents a particular file or piece of information that is used or produced by an activity during process execution. Another data artifact is *Data store* that points out the storage for data objects. Activities can read and write objects to the *Data store*.

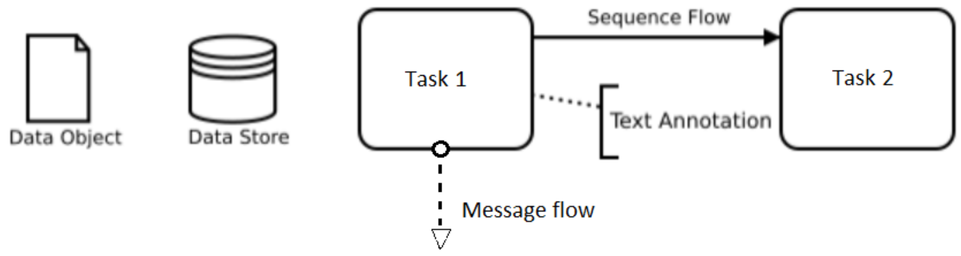


Figure 5. Other BPMN elements

Swimlanes represent the entities and participants in the business process. There are two types of swimlane objects: *Pools* and *Lanes*. *Pools* are usually used to represent an entire organization, whereas *Lanes* are used to denote particular actors inside an organization. *Lanes* can be further divided into smaller lanes as shown on Figure 6.

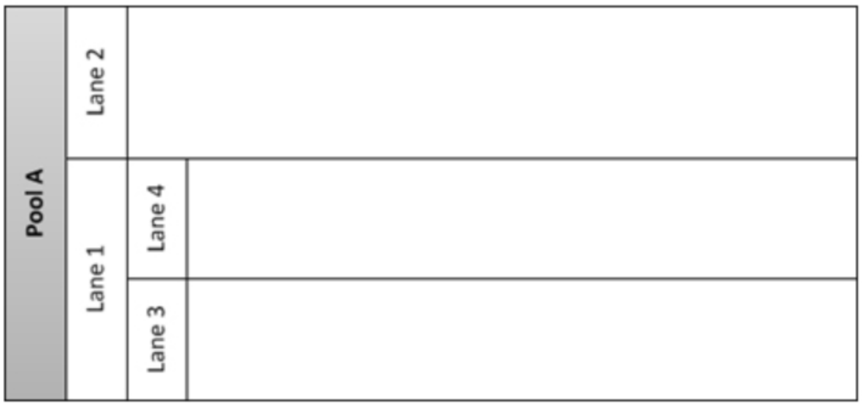


Figure 6. BPMN swimlanes

3 Case Study

In this chapter, we will talk about real estate business and its peculiarities in Singapore. We will introduce Singaporean real estate company, and present its case. Also, we will describe the core processes of the company modeled with the help of BPMN after the interview sessions with the CEO of the company. Eventually, we will present a domain model for modeled rental process.

3.1 Real Estate Business in Singapore

Before start talking about real estate business in Singapore, we would like to give a general overview of real estate. Real estate refers to a property consisting of land, buildings and natural resources surrounding it [13]. Based on the usage real estate can be divided into next categories: residential real estate (homes, condominiums), commercial real estate (properties used in business to generate revenue: shopping malls, offices, hotels), industrial (properties used for manufacturing, storage). Commercial operations related to the real estate are called real estate business. There are a lot of different branches and directions in real estate business, but we can point out three key ones - property trading, investment and ownership transfers (rentals and leasing).

Singapore is territorially a small country. Government regulates the real estate market and owns 80% of the property. For example, due to the high population density, short-term rentals (less than six months) of government-owned property are forbidden in Singapore, and companies such as Airbnb are considered illegal[3].

There are several middlemen involved in processes of property sale and rentals: Housing and Development Board of Singapore (HDB), real estate agents, banks. All parties significantly complicate the process of renting or selling a property adding extra verifications and information handovers. On average, a single property trading transaction can take 4 months[4].

Regarding the renting of a real estate property in Singapore it is interesting to note that most of transactions are performed by real estate agents. Every agent must register himself and receive a license from HDB. According to the information retrieved from the representative of a real estate company, real estate agents as they are now preventing industry from adopting the technologies and changes. Expectation from a real estate agent is such that he takes the role of a marketer, deal-maker and trusted advisor. However, in the current state of the market real estate agents do not satisfy those expectations. Adopting the technology and letting industry to change will create more DIY (“do-it-yourself”) transactions, when landlords and tenants meet and transact on their own. Blockchain and distributed ledger technologies promise to disintermediate and disrupt real estate industry [1, 10]. A research made by Averspace Pte. Ltd. in January 2017 showed that with digital automation and elimination of information handovers it is possible to reduce waiting time for an HDB resale application by at least 2 months [11]. Creating a platform that will be able to remove intermediaries from the real estate processes would significantly decrease the transaction price and time.

[3] Based on Skype call with the CEO of the real estate company on 28.09.2017

[4] Based on Skype call with the CEO of the real estate company on 21.09.2017

3.2 Introducing the company

The case study is performed with an undisclosed industry partner – a Singaporean real estate company that operates in a property rental sector. It claims to be the first blockchain enabled real estate platform in Singapore. It allows landlords and prospective tenants to meet each other and facilitates real estate transactions. The company managed to integrate blockchain into one of their core processes - property contract signing, storing hashes of signed documents in the blockchain. Currently, the company is using in-house solution for digital signatures and a third-party provider to store records in the blockchain. However, now they want to migrate some of core business processes to the blockchain. Below, we present and describe property rental process modeled after a series of interviews with CEO of the company.

3.3 Rental Process

Speaking about rental process in the real estate company there are basically two parties involved: tenant and landlord. The company provides a platform for their collaboration, adding extra services like validation, moderation and money transactions on top of it. We captured a high-level perspective of the rental process in the real estate company with a value chain diagram. It is presented on Figure 7.

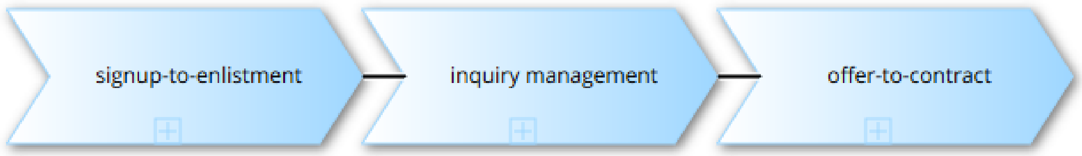


Figure 7. Rental value chain

As it can be seen from the diagram value chain is divided into three parts. “Signup-to-enlistment” part of the value chain refers to the process when landlord registers himself on the real estate platform and creates property enlistment that is reviewed and published after eligibility checks done by the company employee. Having published the property enlistment landlord starts receiving different inquiries about the property and schedules property visits. This stage is represented by “inquiry management” part of the value chain. “Offer-to-contract” part describes process after a successful offer was received. Landlord issues a rental agreement draft that passes through several reviews by both parties and is eventually signed. We will cover in details all three parts of the value chain below.

3.3.1 Signup-to-enlistment

In “signup-to-enlistment” part of the value chain landlord creates an account on the rental platform and submits the property enlistment, by entering all necessary data. In Singapore, a landlord must have permission to rent his property. Thus, real estate companies always perform extra eligibility checks before publishing the property enlistment. BPMN representation of “signup-to-enlistment” part of the value chain is shown on Figure 8.

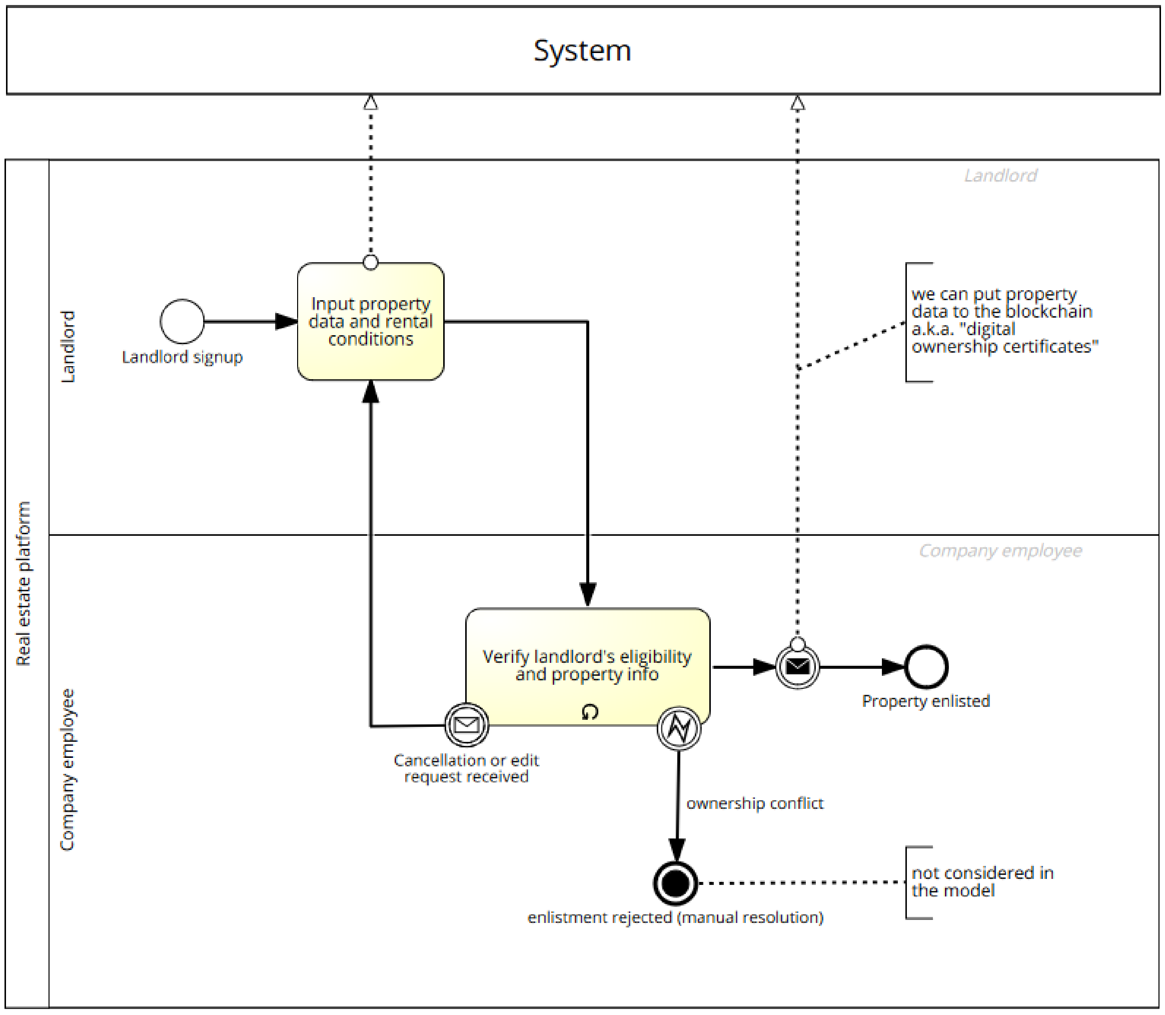

System
Landlord
Company employee
Real estate platform
Landlord
Landlord signup
Input property data and rental conditions
we can put property data to the blockchain a.k.a. "digital ownership certificates"
Company employee
Verify landlord's eligibility and property info
Property enlisted
Cancellation or edit request received
ownership conflict
not considered in the model
enlistment rejected (manual resolution)

Figure 8. "Signup-to-enlistment" part of rental value chain

After creation of property enlistment, one of the company employees performs manual eligibility checks. It is a repetitive process that can include several contacts with the landlord. If the landlord decides to cancel the property enlistment or modify the data in it, he needs to pass eligibility checks again. If there is an ownership conflict, then the employee marks this enlistment as rejected and does not publish it. In the model we do not consider what happens next, we agreed that ownership conflict leads to the termination of the process. After passing all the checks, property enlistment is published. During the modeling phase, we were considering the case where it is possible to retrieve digital certificates that verify landlord ownership from the government and store them in the system. However, it is not possible yet.

3.3.2 Inquiry Management

The next important part of the rental value chain is inquiry management. Usually, after property enlistment is published landlord receives extra questions about the property or arranges property visits. From the interview with the company representative, we received the information that it is managed via chat engine embedded into the platform[5]. It allows both parties to communicate and resolve all pending questions, negotiate and arrange the appointment. So, when tenant searches for property and finds a suitable one, he can contact landlord in chat if he has any questions or concerns. This part of the value chain can be considered as a

[5] Based on Skype call with the CEO of the real estate company. on 05.10.2017

repetitive exchange of messages and the high-level representation of this process can be found on Figure 9.

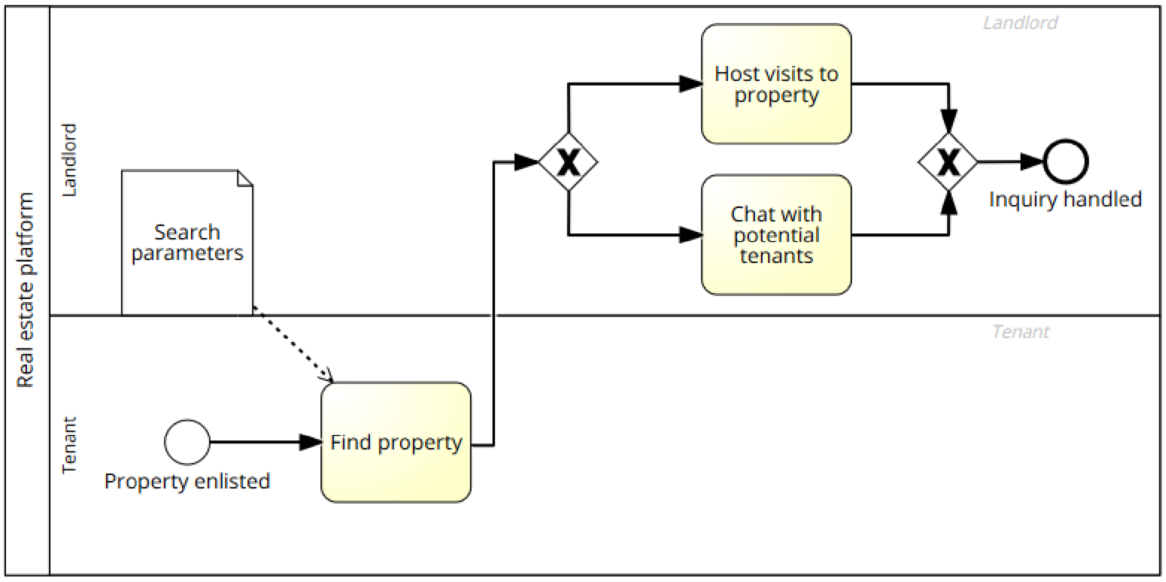


Figure 9. “Inquiry management” part of rental value chain

3.3.3 Offer-to-contract

The last and the most important part of the rental value chain in the real estate company is “offer-to-contract” process. The BPMN diagram of this process is shown on Figure 10.

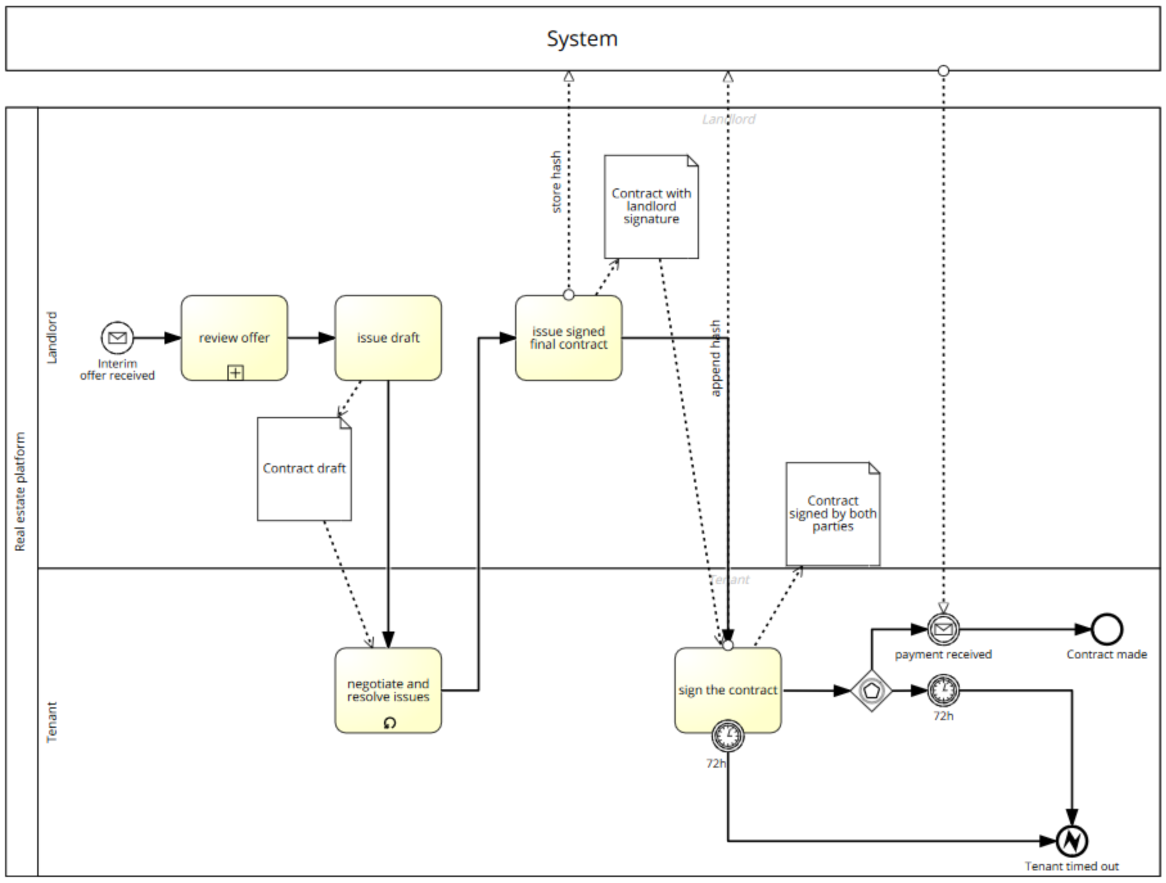


Figure 10. “Offer-to-contract” part of rental value chain

When a tenant makes an offer for a specific enlistment, the landlord should review it and either accept or reject. In a successful case, landlord issues a tenancy agreement draft. The system allows parties to exchange drafts until they find a consensus. Also, it is possible to resolve issues in chat. Once landlord and tenant agree on the rental agreement conditions, landlord issues and signs the final version. The notable part is that a hash is calculated from the signed document and is stored in the blockchain. From this moment tenant has 72 hours to sign the contract and then the resulting hash will be calculated and stored in the blockchain.

Initially, during the interviews with the representative of the real estate company, we were also discussing the money flow. The tenant had to pay an escrow that would be returned in case of successful contract signature and split between parties if the tenant fails to sign the agreement. Also, we introduced a timer event for the first month payment. The tenant had 72 hours to make the payment. Otherwise, the agreement would not be considered as finished. However, building the blockchain proof-of-concept, we considered neither the money flow nor timers because they add extra complexity and can be researched separately. As a result of such decision, we simplified our business process models and split "offer-to-contract" process into three sub-processes presented below.

The first sub-process describes the flow how tenant and landlord agree on offer. When a tenant decides to make an offer, he sends it the landlord with a proposed amount of monthly rent. The system saves a pending offer. The landlord either accepts or rejects the offer. At any point in time, the tenant can cancel the offer. This sub-process is presented on Figure 11.

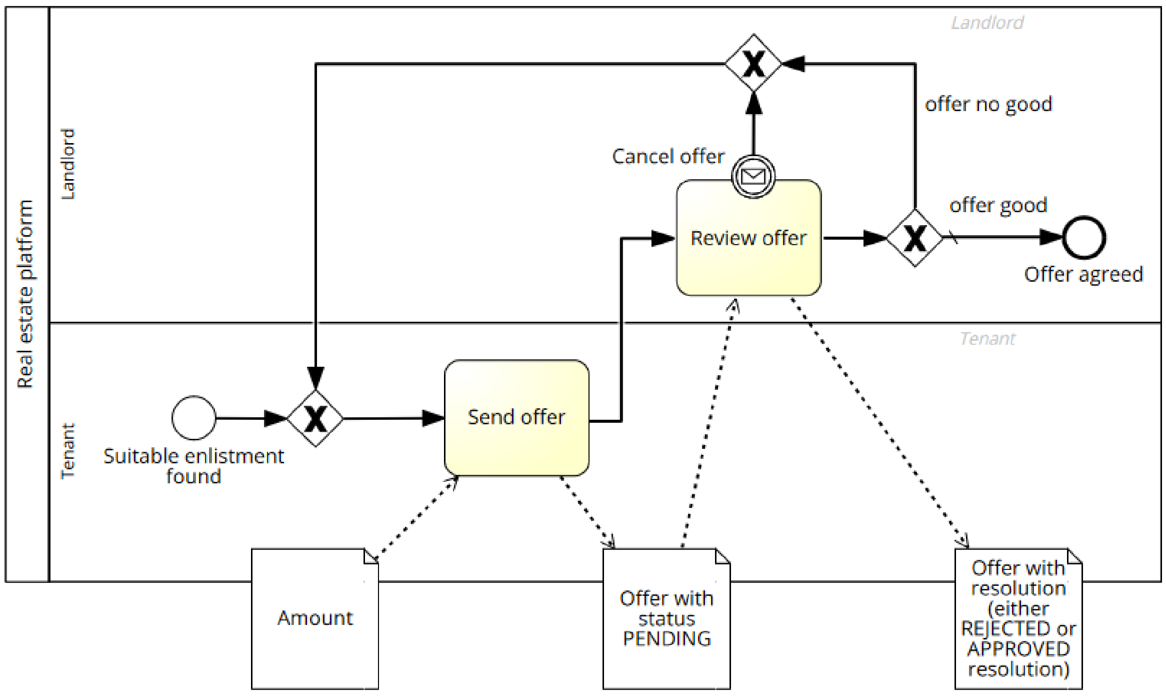


Figure 11. Receive and review offer

The second sub-process is presented on Figure 12. It covers the exchange of agreement drafts between tenant and landlord. Again, at any point of time agreement can be canceled that returns process to the stage when the offer is accepted by the landlord. It is possible to resolve any issues via chat, but it is not captured in sub-process model as this option exists in the background and can be used at any time.

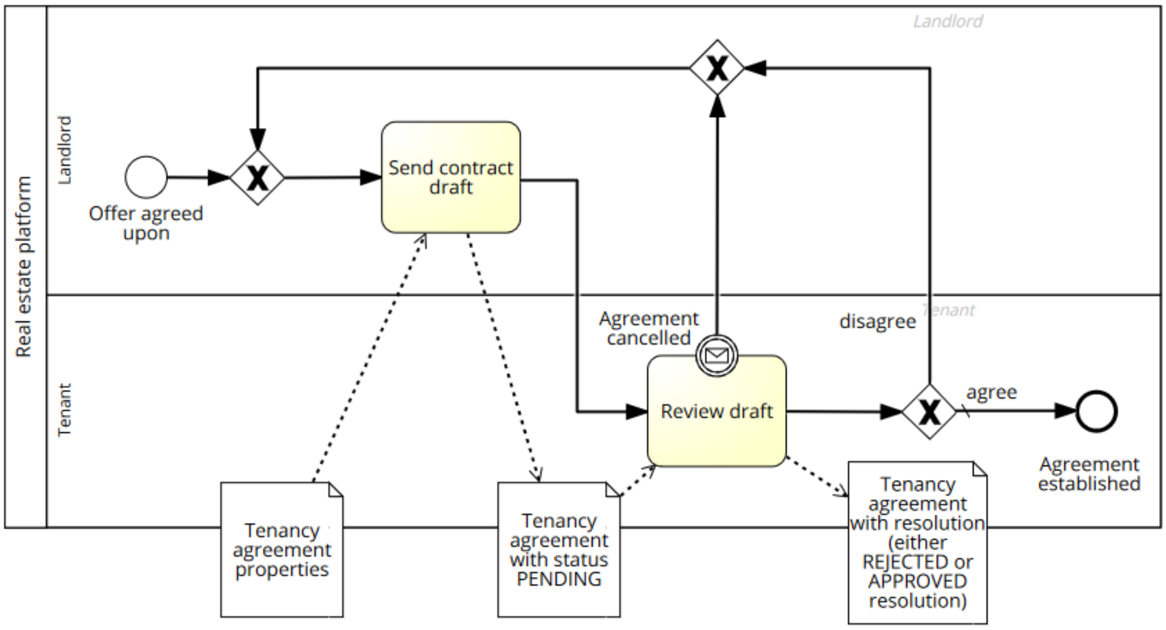


Figure 12. Establish rental agreement

The third sub-process covers signing the rental agreement and is presented on Figure 13. When tenant and landlord established the agreement, landlord signs the final draft and then tenant signs it. From this moment rental agreement is valid. We do not consider any payments and timeouts.

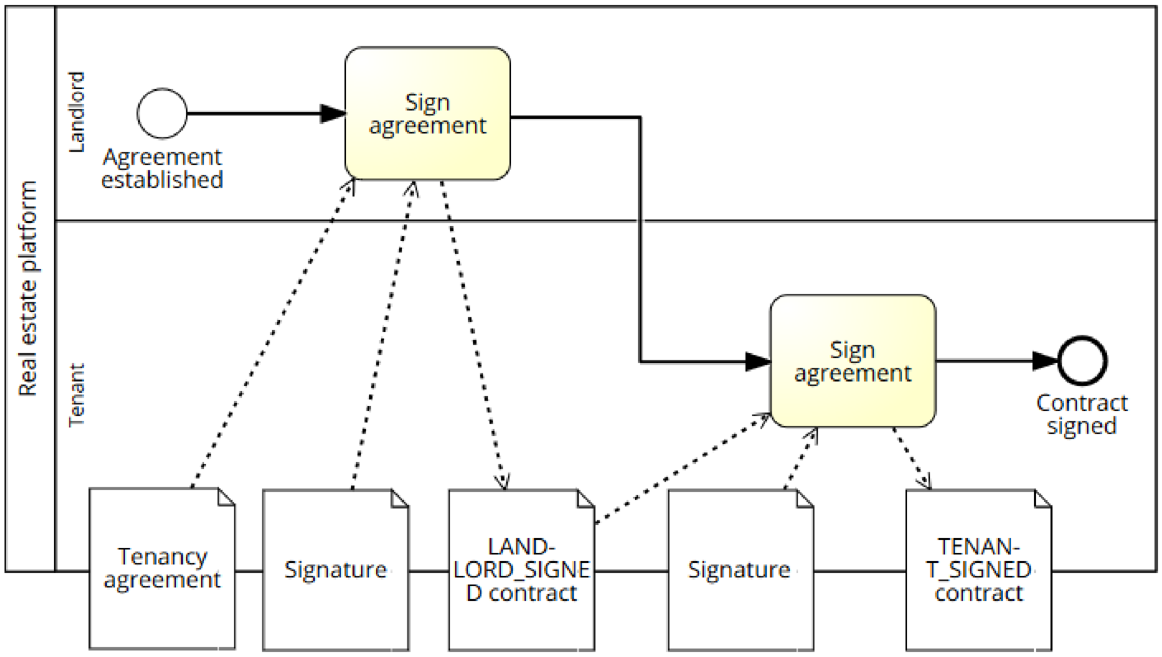


Figure 13. Rental agreement signing

We also annotated steps in each business process with necessary inputs and resulting outputs. Later, we will use this information for defining what should be stored in a smart contract.

3.4 Domain Model

Up to this moment we were discussing the case of the real estate company from the process perspective. Now we want to cover the domain model. Schematic representation of the domain model is shown on Figure 14.

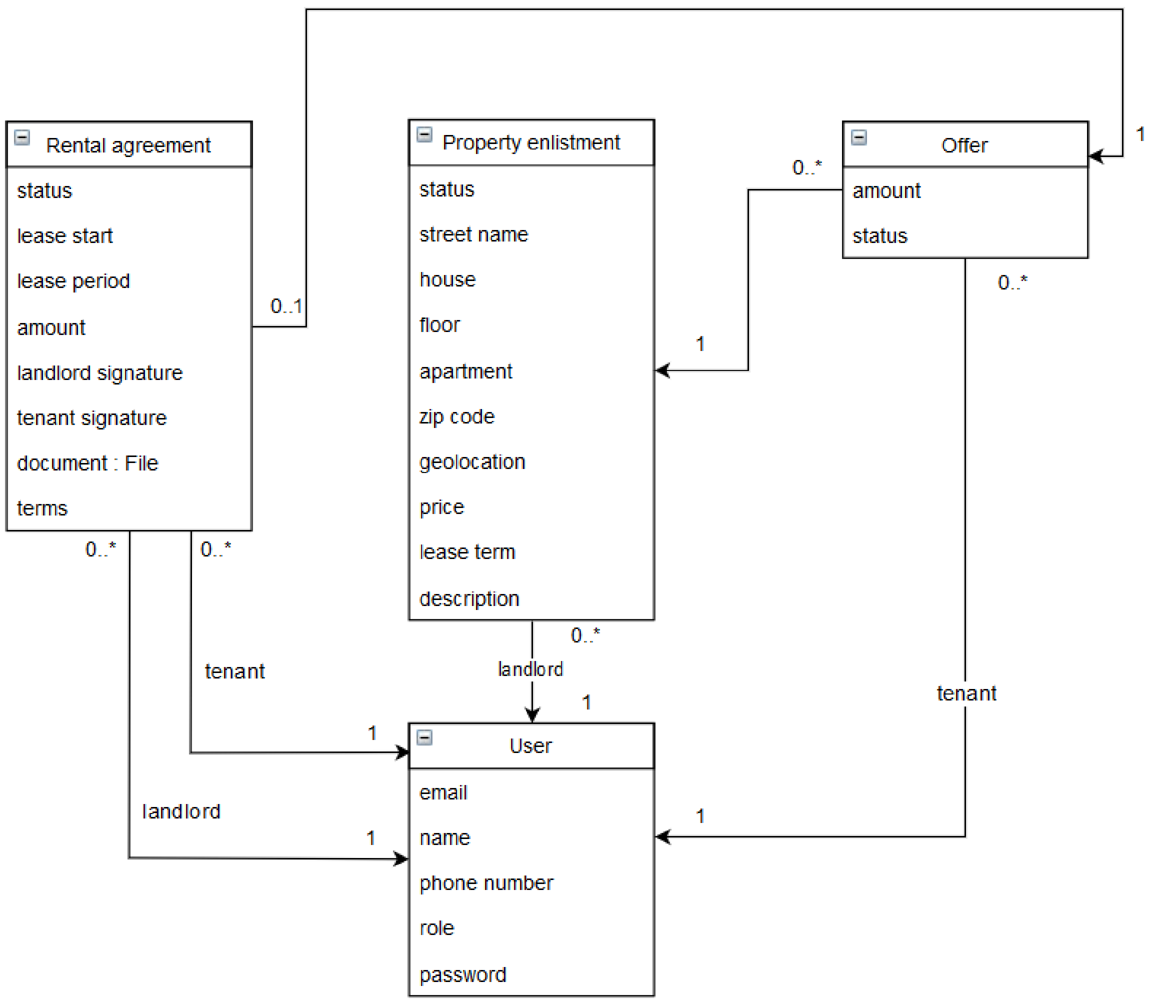


Figure 14. Domain model

There are four primary entities in the domain: *User*, *Property enlistment*, *Offer* and *Rental agreement*. *User* entity is used to represent different actors in the system: landlord and tenant respectively. After registration landlord creates a property listing that is represented by *Property enlistment* entity. It captures all necessary attributes: status, address fields, geolocation, price, lease term and description. A tenant can create an offer for enlistment showing his interest in it. In the domain model we have a corresponding *Offer* entity that is linked to *Property enlistment* and *User* (in this case tenant). If landlord and tenant agreed on the offer, the landlord creates a rental agreement draft and starts the signing process. It the domain model it is captured by *Rental agreement* entity. This entity encapsulates agreement attributes, stores signatures of each party and has a reference to the latest agreement document.

3.5 Summary

In this chapter we described the real estate business in Singapore, where government owns 80% of property and significantly regulates the real estate market. We introduced a real estate company from Singapore and explained its goals regarding the integration of block-chain into its processes. Also, we modelled the business processes of the company with help of BPMN and described them in details. Finally, we presented and discussed the domain model.

4 System design

In this chapter, we will map the case study described in Chapter 3 into a technical solution. We will build a proof-of-concept hybrid application that reflects business processes of a rental platform. Chapter 4 consists of three sections. In the first one, we will discuss what parts of business processes will be moved to the blockchain and why. In the second section, we will discuss actual implementation of the baseline application, covering the architecture, database design, smart contract implementation and integration of the latter one with our web application. In the last section, we will extend the baseline solution adding a tampering-resilient document storage. We will compare Ethereum Swarm and IPFS, choose one of them and integrate into our application. The complete implementation can be found in the GitHub repository[6].

4.1 Design Decisions

Since the company we are collaborating with wants to move the core business logic to the blockchain, we need to define what exactly should be moved to the smart contract based on the performed case study. In this section we heavily use the next terms: “on-chain” and “off-chain”. On-chain transaction modifies the blockchain and depends on it to determine the validity. Any other action that is performed outside of blockchain can be considered as an off-chain transaction.

After series of discussions we decided to entirely move “offer-to-contract” part of the value chain to the smart contract. This part of the value chain is the most complex as property enlistment has many states. It is already partially integrated with a blockchain and the company representative emphasized that it has a potential for tokenization in future. We decided not to move inquiry management to blockchain for several reasons. It requires instant processing and is already perfectly served by chat engine. Providing similar experience with a smart contract is not possible due to high transaction time. It is not rational to move chatty interfaces to the smart contract because it will consume a lot of resources (by paying money for each message). Regarding the creation of property enlistment, we decided to leave it off-chain as this part of the rental process has a lot of manual eligibility checks. Thus, it can kick the process off from its usual flow at early stages. Having it on the blockchain will require tight integration with an off-chain part of the system. Also, there may be a waste of the resources, because deployment of a smart contract costs a certain amount of gas. If the landlord is not eligible for renting his property, business process will not proceed further in this case and money spent to deploy a smart contract will be wasted. It is better to start interacting with smart contracts after all manual checks were performed and the process can be safely continued, though it depends on the architecture of the application and actual implementation of the smart contract.

4.2 Baseline Solution

As discussed in previous section the whole rental process should not be executed on-chain. Therefore, we decided to build a hybrid application that will have a traditional backend and a part of logic implemented in an Ethereum smart contract. On the Figure 15 we present the general architecture of the system.

[6] https://github.com/kopylash/blockchain-real-estate-v1

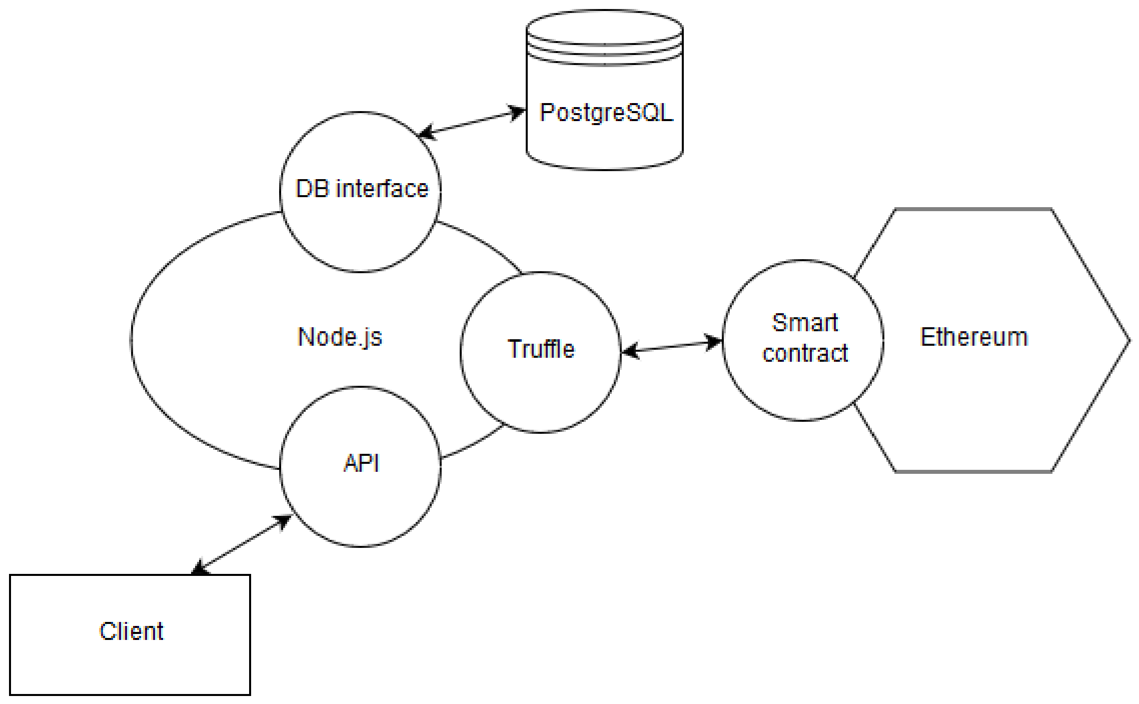


Figure 15. Application architecture

Node.js[7] was used as a platform for the development of the application. Several factors reason this choice. First of all, it was a preferred option for the partner company because it is their main backend platform. Another reason is the big support of Ethereum JavaScript API via web3.js[8] and a huge community around it. Using JavaScript, it is possible to have Ethereum calls both on backend and frontend. It gives a lot of flexibility for creating a proof-of-concept application. PostgreSQL[9] was used as an off-chain database. The application has no UI. It only provides an API for the consumers. In our point of view, UI does not add a sufficient value right now but can be added and explored in further research. Regarding the on-chain part, we have an Ethereum smart contract that captures "offer-to-contract" part of the business process. We used Truffle[10] library to create an abstraction over a smart contract and wrap up the communication with Ethereum. We will cover all these parts in details below.

4.2.1 API

As it can be seen on Figure 15 the off-chain part of the application includes a web server that exposes API and a PostgreSQL database. The web server was created using Express[11] – a minimalistic web framework for Node.js. Express uses middleware concept to serve requests, so we built a middleware stack mapping routes to a list of controllers. To explain the off-chain part of the application better, we will describe the API and request flows. We have three controllers that handle requests for three different contexts: property enlistment, offer, and rental agreement. Those controllers expose the following methods.

[7] https://nodejs.org/en/
[8] https://github.com/ethereum/web3.js/
[9] https://www.postgresql.org/
[10] http://truffleframework.com/
[11] http://expressjs.com/

PropertyEnlistment Controller

- `POST /enlistments` – create property enlistment
- `GET /enlistments` – find property enlistment in area
- `POST /enlistments/:id/approve` – approve property enlistment
- `POST /enlistments/:id/reject` – reject property enlistment

Offer Controller

- `POST /enlistments/:id/offers` – send offer for enlistment
- `GET /enlistments/:id/offers` – get last offer from tenant
- `POST /enlistments/:id/offers/cancel` – cancel offer
- `POST /enlistments/:id/offers/review` – accept or reject offer

AgreementContract Controller

- `POST /enlistments/:id/agreements` – create agreement draft
- `GET /enlistments/:id/agreements` – get agreement for tenant
- `POST /enlistments/:id/agreements/review` – accept or reject agreement
- `POST /enlistments/:id/agreements/sign` – sign agreement
- `POST /enlistments/:id/agreements/cancel` – cancel agreement before it is signed

Complete API reference can be found on this page. It can be imported as a Postman[12] collection to simplify environment setup and testing. The successful API flow is presented on Figure 16.

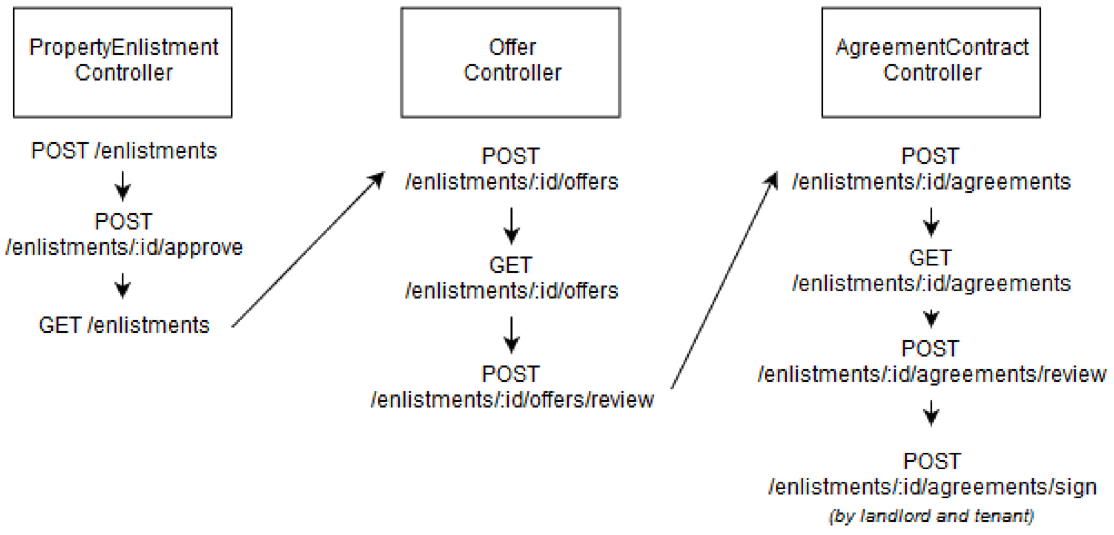

PropertyEnlistment Controller
POST /enlistments
POST /enlistments/:id/approve
GET /enlistments
Offer Controller
POST /enlistments/:id/offers
GET /enlistments/:id/offers
POST /enlistments/:id/offers/review
AgreementContract Controller
POST /enlistments/:id/agreements
GET /enlistments/:id/agreements
POST /enlistments/:id/agreements/review
POST /enlistments/:id/agreements/sign
(by landlord and tenant)

Figure 16. API flow of a successful rental process

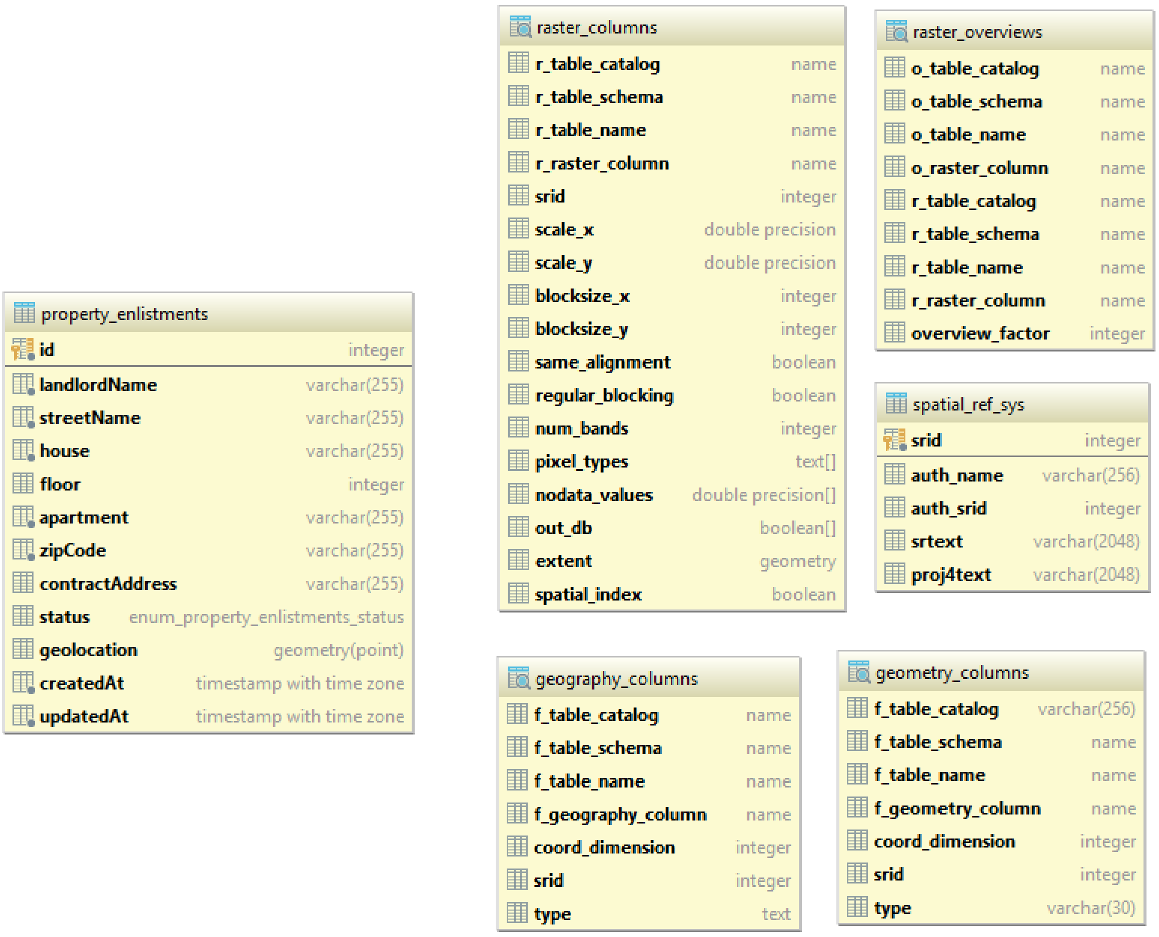
Also, we introduced services layer in the application structure. This layer allows to store logic in the single place and leaves controllers very simple and flat. Moreover, using services enforces loose coupling in the code so that controllers can reuse service functionality. For instance, all three controllers use *PropertyEnlistmentService* that provides necessary functions and hides the implementation details. Therefore, controllers are agnostic, and at any

[12] https://www.getpostman.com/

time we can change service implementation from smart contract proxy to an off-chain implementation.

4.2.2 Database

As an off-chain storage we used PostgreSQL database. To connect it to our Node.js application we used an ORM framework called Sequelize[13]. It adds support of database models providing bindings from database table rows to JavaScript objects. Integrating a smart contract into our application will also influence the data model - it will be partitioned between on-chain and off-chain sides. As we are moving the whole "offer-to-contact" part of the value chain to the blockchain, we decided to split the data model in the following way. In the off-chain database we store only necessary info about property enlistment before it passes manual eligibility checks and is approved by the company employees. Once the property enlistment receives an approval, the smart contract is deployed to Ethereum and its address is linked to the corresponding row in the off-chain database. Since that moment smart contract becomes the only source of truth and we no longer store any data in the off-chain database. Off-chain storage is also used to implement geolocation-based search. PostgreSQL has an extension called PostGIS[14] that adds support for spatial objects and enables location queries to be run in SQL. The schema of the off-chain database is shown on Figure 17.



property_enlistments	
id	integer
landlordName	varchar(255)
streetName	varchar(255)
house	varchar(255)
floor	integer
apartment	varchar(255)
zipCode	varchar(255)
contractAddress	varchar(255)
status	enum_property_enlistments_status
geolocation	geometry(point)
createdAt	timestamp with time zone
updatedAt	timestamp with time zone

raster_columns	
r_table_catalog	name
r_table_schema	name
r_table_name	name
r_raster_column	name
srid	integer
scale_x	double precision
scale_y	double precision
blocksize_x	integer
blocksize_y	integer
same_alignment	boolean
regular_blocking	boolean
num_bands	integer
pixel_types	text[]
nodata_values	double precision[]
out_db	boolean[]
extent	geometry
spatial_index	boolean

raster_overviews	
o_table_catalog	name
o_table_schema	name
o_table_name	name
o_raster_column	name
r_table_catalog	name
r_table_schema	name
r_table_name	name
r_raster_column	name
overview_factor	integer

spatial_ref_sys	
srid	integer
auth_name	varchar(256)
auth_srid	integer
srtext	varchar(2048)
proj4text	varchar(2048)

geography_columns	
f_table_catalog	name
f_table_schema	name
f_table_name	name
f_geography_column	name
coord_dimension	integer
srid	integer
type	text

geometry_columns	
f_table_catalog	varchar(256)
f_table_schema	name
f_table_name	name
f_geometry_column	name
coord_dimension	integer
srid	integer
type	varchar(30)

Figure 17. Off-chain database schema

[13] http://sequelize.readthedocs.io/en/v3/
[14] https://postgis.net/

As it can be seen from the figure, there is only one table represented on the left side: *property_enlistments*. It captures the enlistment data together with geolocation before the smart contract is instantiated. The rest of the tables that are grouped on the right side are created by PostGIS to add spatial objects support.

4.2.3 Smart contract

At first stages of implementation there were discussions what to put inside of the smart contract. Conceptual decision of capturing the "offer-to-contract" sub-process in a smart contract gave boundaries, but the actual implementation was a big question. Should the contract be modeled in a way that landlord and tenant will have own Ethereum wallets and interact directly with a contract? Should the platform be only one interactor with a smart contract? Should everything be captured within one contract or several?

It was decided that actual implementation will consist of one smart contract that captures all the logic related to the property enlistment: storing data about property, handling offers and agreements from different tenants. We will provide snippets form the smart contract code, but the full implementation can be found in the GitHub repository[15].

In the code listing below contract state variables are presented. It was decided that the system will deploy and create smart contracts and only system will have access to them. It is handled with `owner` variable. Upon creation the address form which contract was created is stored in this variable and later all accesses to functions are restricted by corresponding modifier.

```
pragma solidity ^0.4.18;

contract EnlistmentToContract {
    address owner;
    string landlord;
    bool public locked = false;
    Enlistment enlistment;
    mapping(string => Offer) tenantOfferMap;
    mapping(string => AgreementDraft) tenantAgreementMap;

    modifier ownerOnly() {
            require(msg.sender == owner);
            _;
    }
      //...
}
```

Information about property enlistment is stored in a separate struct, while `landlord` variable stores the email of the landlord for easier access. To manage offers from different tenants we introduced a mapping called `tenantOfferMap` that maps tenant email to a specific offer. Such implementation allows to store only one offer from each tenant, but there is no need in offer history, landlord is interested only in the most relevant offer. To capture offer information we defined a special struct.

```
struct Offer {
        bool initialized;
        int amount;
```

[15] https://github.com/kopylash/blockchain-real-estate-v1/blob/master/ethereum/contracts/EnlistmentToContract.sol

```
        string tenantName;
        string tenantEmail;
        OfferStatus status;
}

enum OfferStatus {PENDING, REJECTED, CANCELLED, ACCEPTED}
```

The same work was done to handle rental agreements. A mapping was introduced that maps tenant emails to the agreement drafts. Despite the fact that only one agreement in progress is possible corresponding to the process description, previous agreements from other tenants are stored in the mapping as well.

```
struct AgreementDraft {
       // for simplicity, there is only one landlord
    string landlordName;
       // for simplicity, there is only one tenant and occupants are omitted
    string tenantName;
       string tenantEmail;
    int amount;
    uint leaseStart;
    uint handoverDate;
    uint leasePeriod;
    string otherTerms;
    string hash;
    string landlordSignedHash;
    string tenantSignedHash;
    AgreementStatus status;
}

enum AgreementStatus {
       UNINITIALIZED, // internal
        PENDING, REJECTED, CONFIRMED, CANCELLED,
       LANDLORD_SIGNED, TENANT_SIGNED, COMPLETED
}
```

As it can be seen from the smart contract code, each entity has a *status* enumeration. It allows to implement a complex lifecycle of the property enlistment. To a certain extent our smart contract can be considered a state machine. Therefore, it is natural to represent the lifecycle of the property enlistment with a statechart as shown on Figure 18.

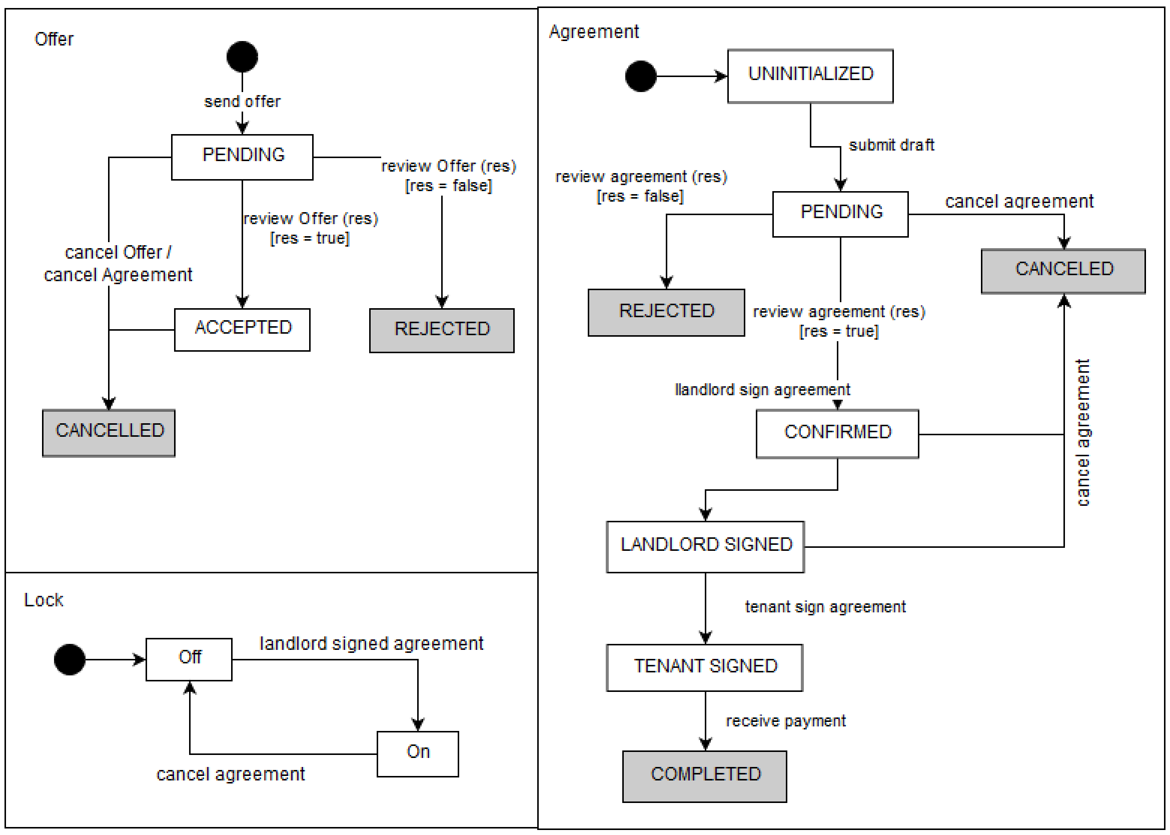


Figure 18. Statechart: property enlistment lifecycle

Property enlistment state consists of several interconnected regions: Offer, Agreement and Lock. When the landlord accepted offer (Offer region is in state ACCEPTED), he issues an agreement draft, so Agreement region switches to state PENDING. Only after final agreement was signed by the landlord, the Lock region changes the state to ON and this smart contract cannot receive offers anymore. Lock region was added to simplify the implementation. Instead of checking three states (LANDLORD_SIGNED, TENANT_SIGNED, COMPLETED) and create guards on their base, we introduced a state variable and use it as a guard. Cancellation of offer also triggers the cancellation of the corresponding agreement if such exists. On the Figure 18 guards for transitions between states are omitted for better visibility. However, there are a lot of guards that ensure that transition between states is allowed and is valid. In the smart contract guards are implemented using modifiers. For example, `sendOffer` function has two modifiers that restrict receiving of the offer if there is any active offer or the property enlistment is locked, because landlord signed the rental agreement.

```
modifier noActiveOffer(string tenantEmail) {
    require(
            tenantOfferMap[tenantEmail].initialized == false
            || tenantOfferMap[tenantEmail].status == OfferStatus.REJECTED
            || tenantOfferMap[tenantEmail].status == OfferStatus.CANCELLED
    );
    _;
}

modifier notLocked() {
    require(!locked);
    _;
}

function sendOffer(int amount, string tenantName, string tenantEmail) payable
public
    ownerOnly()
    noActiveOffer(tenantEmail)
    notLocked()
{
    var offer = Offer({
            initialized: true,
            amount: amount,
            tenantName: tenantName,
            tenantEmail: tenantEmail,
            status: OfferStatus.PENDING
    });
    tenantOfferMap[tenantEmail] = offer;
}
```

In Solidity there is a limitation for the size of the tuple returned from a function. Currently it is allowed to return only tuples with maximum of 7 elements. Therefore, it was necessary to create multiple getter functions that return information about the single agreement. To sum up, the developed smart contract captures the property enlistment information and the lifecycle of property enlistment by handling offers and agreements. Function modifiers enforce the transitions between complex states.

4.2.4 Interacting with a Smart Contract from Node.js

We have already discussed the web server and the API that is exposed to the world, the database and Ethereum smart contract. However, there is still one part of the system missing. As it was mentioned before, users will have no access to the smart contract and will not interact with it directly. It was agreed that the application will be responsible for interaction with Ethereum smart contract. The question is how to integrate web server with Ethereum smart contract? For these purposes we used Truffle – a JavaScript framework for Ethereum development. Truffle provides a set of tools for the development, testing, compilation and deployment of Ethereum smart contracts. It is well documented and has many examples.
To add Truffle support to the project and have separate contexts for the web server and the smart contract, we added a separate folder and initiated a Truffle project inside of it as shown on Figure 19.

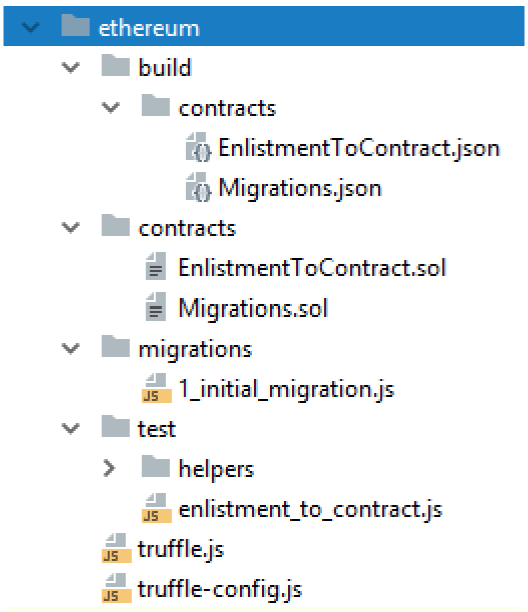


Figure 19. Truffle project structure

Typical Truffle project consists of four folders - *build*, *contracts*, *migrations*, and *test* - and Truffle configuration. In configuration one need to define the Ethereum networks to which Truffle can connect and deploy smart contracts.

Source code of smart contracts written in Solidity is stored in folder */contracts*. By convention, Truffle will look for the source code in that folder and compile it producing contract artifacts and storing them in the */build* folder. Build artifact for each contract contains the name of the contract, ABI, bytecode and other metadata like compiler version. Truffle uses this information for deploying and interacting with smart contracts.

Also, Truffle provides a migration mechanism out of the box. It allows to write custom deployments scripts and enables staged deployments. For example, if one contract depends on another, and it must be deployed before first one, there is a possibility to resolve such dependency in migration. Migrations are stored in the */migrations* folder. Truffle has a track of all applied migrations in a special smart contract, so only new migrations will be applied. The last folder contains tests. It is possible to write automated tests in both JavaScript and Solidity. In our project we covered the smart contract with usual JavaScript tests.

Ethereum provides a JavaScript API via web3.js library. There is an option to interact with the developed smart contract via web3.js. Using web3.js one need to send raw requests to Ethereum network, handle connection and check for transaction status manually. It is complicated comparing to the possibilities provided by Truffle. Truffle allows to create abstractions over the smart contract and to hide the implementation details. All is needed is to provide a build artifact to the *truffle-contract* constructor. Truffle still uses web3.js, but it significantly simplifies the interaction.

```
const Web3 = require('web3');
const contract = require('truffle-contract');

const config = require('../../config/ethereum');
const artifact=require('../../ethereum/build/contracts/EnlistmentToCon-
tract.json');

const provider = new Web3.providers.HttpProvider(config.provider);
const PropertyEnlistmentContract = contract(artifact); // create abstraction

PropertyEnlistmentContract.setProvider(provider);
```

Having such an abstraction, we can easily deploy a new contract on demand and receive the contract address upon successful deployment.

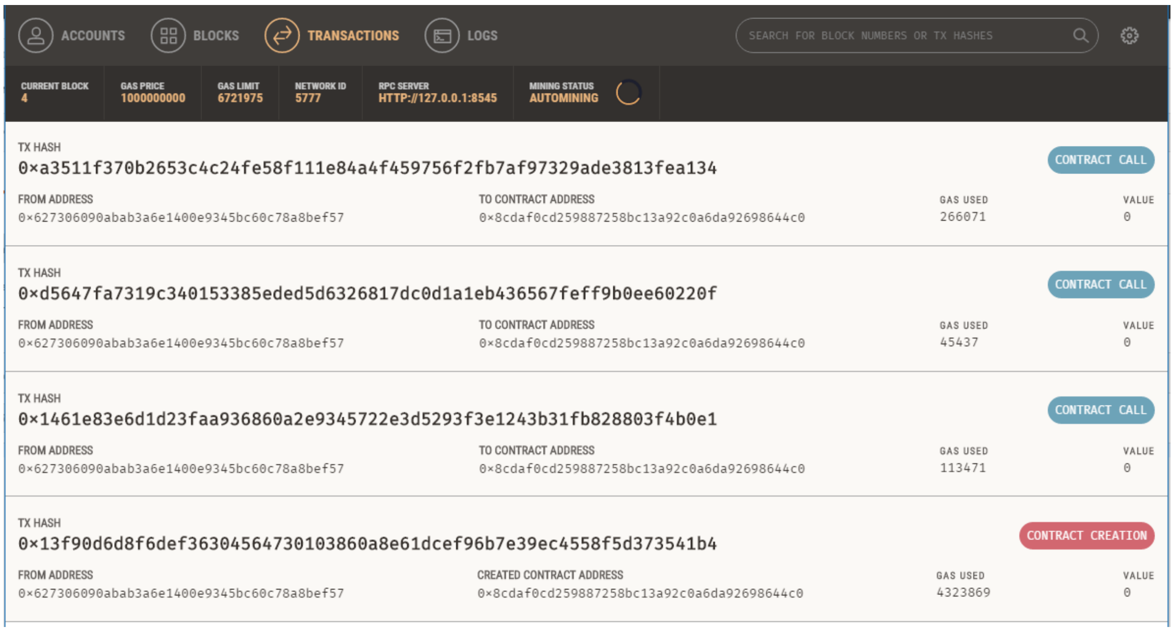
```
createEnlistment(landlordName, streetName, floor, apartment, house, zipCode)
{
    return PropertyEnlistmentContract.new(
        landlordName, streetName,
        floor, apartment, house, zipCode
    ).then(contract => {
      log.info(`Contract created on address: ${contract.address}`);

      return contract.address;
    });
}
```

It is also possible to send a transaction without checking whether it was added to blockchain or not. JavaScript promise will be automatically resolved on a successful transaction processing.

Services layer in the architecture of a web server allowed to introduce a special service that encapsulates interaction with a smart contract. *PropertyEnlistmentContractService* connects to the Ethereum network, creates an abstraction using the contract build artifact and exposes the functions available in a smart contract doing necessary conversions between JavaScript and Solidity. Turning to the API flow presented in section 4.2.1, after property enlistment is validated and accepted, the smart contract is deployed. Then, all requests are proxied to the smart contract on a service level. Hence, it is possible to change the implementation when needed without breaking the API.

Truffle uses a special library to setup the private Ethereum blockchain on a local machine. It is called Ganache[16] and is a continuation of a well-known Ethereum TestRPC project. Ganache has UI client and command line interface. Both were used during the project development for testing purposes. Ganache UI client representing mined transactions is shown on Figure 20.

[16] https://github.com/trufflesuite/ganache-cli



```
CURRENT BLOCK  GAS PRICE   GAS LIMIT  NETWORK ID  RPC SERVER             MINING STATUS
4              1000000000  6721975    5777        HTTP://127.0.0.1:8545  AUTOMINING

TX HASH                                                                                            CONTRACT CALL
0×a3511f370b2653c4c24fe58f111e84a4f459756f2fb7af97329ade3813fea134
FROM ADDRESS                                TO CONTRACT ADDRESS                         GAS USED   VALUE
0×627306090abab3a6e1400e9345bc60c78a8bef57  0×8cdaf0cd259887258bc13a92c0a6da92698644c0  266071     0

TX HASH                                                                                            CONTRACT CALL
0×d5647fa7319c340153385eded5d6326817dc0d1a1eb436567feff9b0ee60220f
FROM ADDRESS                                TO CONTRACT ADDRESS                         GAS USED   VALUE
0×627306090abab3a6e1400e9345bc60c78a8bef57  0×8cdaf0cd259887258bc13a92c0a6da92698644c0  45437      0

TX HASH                                                                                            CONTRACT CALL
0×1461e83e6d1d23faa936860a2e9345722e3d5293f3e1243b31fb828803f4b0e1
FROM ADDRESS                                TO CONTRACT ADDRESS                         GAS USED   VALUE
0×627306090abab3a6e1400e9345bc60c78a8bef57  0×8cdaf0cd259887258bc13a92c0a6da92698644c0  113471     0

TX HASH                                                                                            CONTRACT CREATION
0×13f90d6d8f6def36304564730103860a8e61dcef96b7e39ec4558f5d373541b4
FROM ADDRESS                                CREATED CONTRACT ADDRESS                    GAS USED   VALUE
0×627306090abab3a6e1400e9345bc60c78a8bef57  0×8cdaf0cd259887258bc13a92c0a6da92698644c0  4323869    0
```

Figure 20. Transactions in Ganache UI

4.2.5 Summary

We created a proof-of-concept of a hybrid application that consists of Node.js web server and Ethereum smart contract. Our baseline solution separates the business process into two parts: off-chain and on-chain. An off-chain part of the system handles the process until the moment when property enlistment is approved by the company employee. Also, location-based search is implemented off-chain. The rest of the process is handled by on-chain part of the system, where a separate smart contract instance is created per each approved property enlistment. The application exposes an API and proxies the requests to the specific smart contract. We used Truffle framework to integrate Ethereum smart contract with our web server and Ganache client to setup a local Ethereum network for testing purposes.

4.3 Tampering-resilient Document Storage

The negotiation of a real estate leasing is considered closed when the rental contract between tenant and landlord is signed. Usually it is done on paper. However, modern systems offer digital signature possibilities over an electronic document like PDF. A common practice in this context, is to store hash values computed on the PDF file, which can be used as a unique identifier for the document. The latter is possible because the cryptographic hashes are highly unlikely to collide, when computed on different documents. Going with this option in our real estate application we need to store signed documents. Ethereum is not suitable for storing large files. Blockchain has a limitation of the block size, so files cannot fit the block and be stored in blockchain. Therefore, we should look for other storage solutions that can satisfy our needs. Ideally, we want our storage to inherit the properties of blockchain: distributed, tampering-resilient, fault tolerant. There are two possible candidates for our storage solution: IPFS[17] and Ethereum Swarm[18].

[17] https://ipfs.io/
[18] https://github.com/ethersphere/swarm

4.3.1 IPFS

IPFS (“interplanetary file system”) is a peer-to-peer distributed file system. At the same time, it is content delivery protocol. It implements a content-addressed block storage model with content-addressed hyperlinks [14]. Since IPFS is a peer-to-peer protocol, no nodes are privileged. Each node store IPFS objects in local storage. Nodes connect to each other and transfer objects (files and other data structures). Each file is split into blocks and receives a unique cryptographic fingerprint called multihash. In such way IPFS deduplicates content across the network. Combining distributed hash table and Git (especially its Merkle Directed Acyclic Graph for storing changes) IPFS provides content versioning. Each node in IPFS stores only content it is interested in. Additional indexing over distributed hash table can provide information what is stored on each node. Despite the fact that files are referenced via multihashes, IPFS also provides a human readable names through a distributed naming system called IPNS (“interplanetary file system”). While each node storing only files it used, it is impossible to publish some data to the IPFS, when node is connected it just mounts itself to the global file system exposing desired content. It means that files cannot be backed up to the IPFS and node can be turned off later, if no other node will request the data, it will stay on the local machine and will not be replicated over the network. To meet this requirement IPFS creators introduced Filecoin[19] – an alternative blockchain to incentivize data storing. Combination of IPFS and Filecoin can result into a reliable distributed file system with incentive mechanism that emerges into competitive storage market.

4.3.2 Ethereum Swarm

Ethereum Swarm is a distributed storage platform and content distribution service, a native base layer service of the Ethereum *web3* stack. The primary objective of Swarm is to provide a sufficiently decentralized and redundant store of Ethereum’s public record, in particular to store and distribute decentralized application code and data as well as blockchain data. From an economic point of view, it allows participants to efficiently pool their storage and bandwidth resources in order to provide the aforementioned services to all participants.

From the end user’s perspective, Swarm is not that different from WWW, except that uploads are not to a specific server. The objective is to offer a peer-to-peer storage and serving solution that is DDOS-resistant, zero-downtime, fault-tolerant and censorship-resistant as well as self-sustaining due to a built-in incentive system which uses peer-to-peer accounting and allows trading resources for payment. Swarm is designed to deeply integrate with the devp2p multiprotocol network layer of Ethereum as well as with the Ethereum blockchain for domain name resolution, service payments and content availability insurance [16] (the latter is to be implemented in POC 0.4 by Q2 2018).

4.3.3 Comparison

There are a lot of disputes on the topic how Ethereum Swarm is different from IPFS[20]. Somebody may even say that Ethereum Swarm is reinventing the wheel while community should proceed with IPFS and Filecoin. Here we will point out similarities and differences of these two systems. The comparison is inspired by the article[21] written by Viktor Trón, one of the creators of Ethereum Swarm.

[19] https://filecoin.io/

[20] https://ethereum.stackexchange.com/questions/2138/what-is-the-difference-between-swarm-and-ipfs

[21] https://github.com/ethersphere/go-ethereum/wiki/IPFS-&-SWARM

Similarities

Both projects offer a solution for an efficient distributed file storage. Their high level goals are very similar as each tries to provide an alternative to existing centralized and obsolete HTTP. IPFS and Ethereum Swarm include incentivization layer to encourage file replication by network nodes, though IPFS does it in combination with Filecoin. Both of them use content addressing in data delivery protocol and provide decentralized domain resolution. Each project ensures:

- zero downtime
- low latency retrieval
- resistance to censorship
- content versioning
- efficient autoscaling via content caching

Differences

IPFS is more mature solution in terms of scaling, adoption, community and code maturity. While IPFS has proven itself serving real production use cases, Swarm is being tested on larger scale development networks, though Swarm is built on top of devp2p protocol of Ethereum which has a proven real world usage. However, both projects considered to be in alpha stage. IPFS has a bigger user base and larger community, but Swarm benefits from its tight integration into Ethereum ecosystem and inherits its infrastructural advantage. IPFS has client library implementations in Go and JavaScript, Swarm has only Go version, but web3.js provides API bindings (unstable), so from this point of view, it can be said that both projects are similar. IPFS has better documentation: videos, papers, references; Swarm has only two presentations from conferences, two papers and an uncompleted guide.

Ethereum Swarm is created to become one of three pillars of the new internet vision of the team standing behind Ethereum, guided and inspired by Ethereum needs. At the same time IPFS tries to become a unifying system for integrating different existing protocols. IPFS team has perception that wider adoption is worth compromising censorship with tools for blacklisting, source-filtering (however their usage is optional). Swarm, though, has strong anti-censorship position. Its incentive mechanism ensures content agnostic storage.

From technical point of view there are several key differences between IPFS and Ethereum Swarm. While IPFS uses generic distributed hash table, Swarm's core storage is an immutable content addressed chunkstore, though IPFS structure is modular and DHT is a default option, any other solution can be plugged in instead of it. Two systems use different peer management protocol. IPFS is based on libp2p - an evolved variant of bittorrent implementation with modern optimizations. Ethereum Swarm heavily relies on devp2p protocol that used in the Ethereum core and has proven its capabilities. Historically devp2p creation was inspired by libp2p. In Swarm file can be uploaded to the storage as it could be uploaded to any cloud storage solution, but in IPFS a content from a local node is mounted to the global file system and is not replicated over the network unless somebody will request it. This means that local node cannot be turned off, otherwise it may lead to the case when content is unavailable to the network.

Swarm has a deep integration with the Ethereum ecosystem, so its built-in incentive system benefits from smart contracts capabilities. Devp2p protocol allows efficient accounting off-chain, that will be used for fair bandwidth incentivization and cost reduction due to lower usage of blockchain. IPFS has no built-in incentive layer, but its sister project, Filecoin, adds incentivization and relies on its own alternative blockchain. It uses proof of retrievability and random audits to ensure that content is stored and can be delivered. Such system relies on collective responsibility and can only handle positive incentive. Thus, Swarm incentivizing layer has more potential, considered to be mature and more efficient. One note that should be also added: as two projects are in alpha version, not all of the described features are currently implemented and are subjects to change.

4.3.4 Integration and Implementation

Having considered IPFS and Ethereum Swarm and made their comparison, we chose IPFS as a tampering-resilient document storage for our application. There are several arguments in favor of this decision. First of all, both projects are in alpha stage, but IPFS is more mature and already serves real world use-cases, while Ethereum Swarm is still a proof of concept of version 0.2 which is unstable and lacks many features claimed in a roadmap. Stable version 0.4 should be released by the end of Q2 2018, it is still not released at the time of writing this work. IPFS has better documentation and references. Also, IPFS provides a client library written in JavaScript, that ideally suits our ecosystem with Node.js. In other circumstances we would like to try Ethereum Swarm as in our point of view it has better integration into Ethereum, where we run our smart contracts, its incentive system considered to be more efficient and flexible. Downsides of choosing IPFS are that we will not have guaranteed persistence without using Filecoin, that will require additional efforts, but we exclude storage incentivization from the scope of this work, thus it can be researched in future.

To outline integration of IPFS into our Node.js application we will provide a high-level sequence diagram of rental agreement signing process.

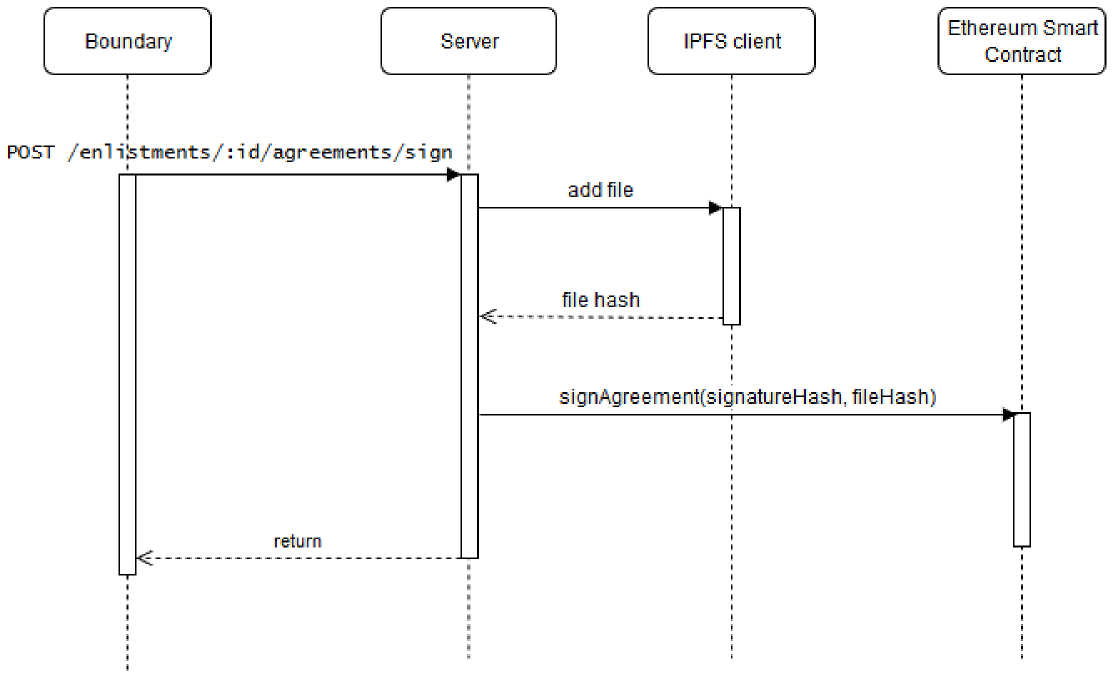

Boundary
Server
IPFS client
Ethereum Smart Contract
POST /enlistments/:id/agreements/sign
add file
file hash
signAgreement(signatureHash, fileHash)
return

Figure 21. Agreement signing sequence diagram

We exposed API with a POST method. When called it will expect a multipart form data with attached PDF file and metadata: tenant email, signature hash and the signing party, either landlord or tenant. Next, server will save the file on a local machine and add it to IPFS via JavaScript IPFS client that will return file multihash. Now, we can use this multihash to reference the file in our smart contract. To finish the flow, server will call smart contract proxy service and invoke an appropriate method depending on the signing party. The design of a smart contract already included a document hash, but instead we will store the IPFS multihash. With a reference to the file in IPFS stored in a smart contract it is impossible to tamper the document because each transaction in Ethereum is verified and IPFS is content addressed (modified file will have another location). On the Figure 22 we present the modified architecture of the system.

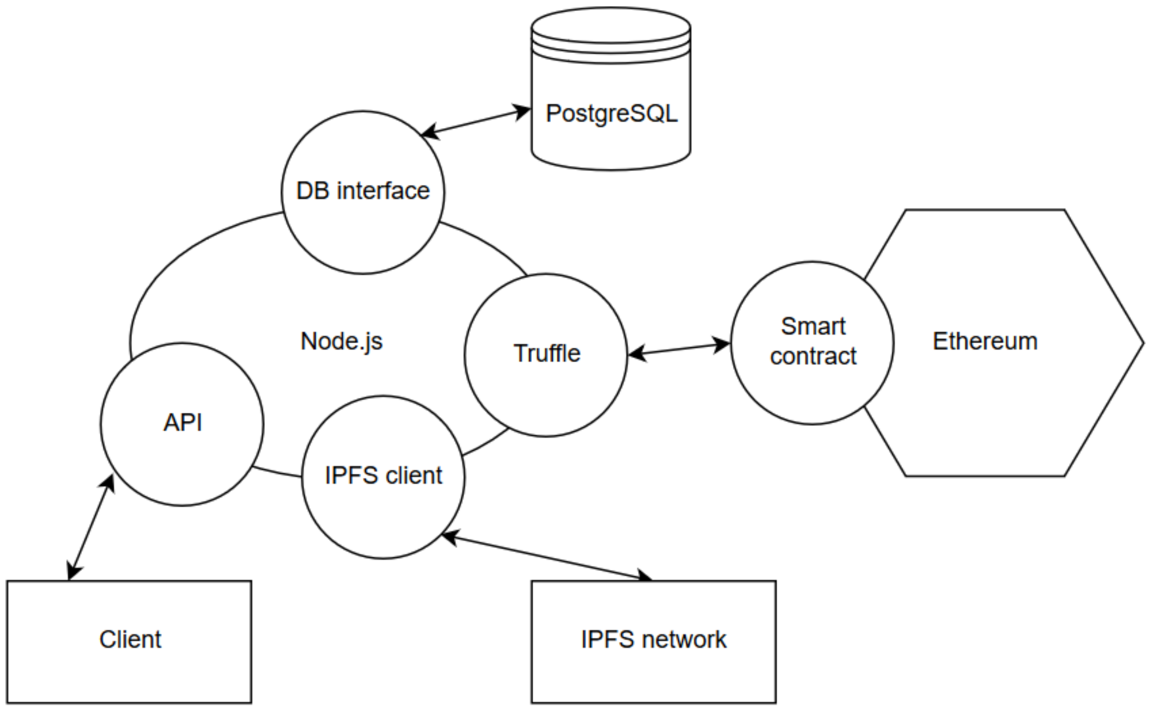


Figure 22. Modified system architecture

To encapsulate the connection and work with IPFS we introduced an interface that hides the implementation details and acts as a mediator between the system and IPFS network. All this is done on a services level, which means that the implementation can be replaced at any time without breaking the contract with the rest of the application.

4.3.5 Summary

We discovered a need for a tampering-resilient document storage and defined the requirements for it. We analyzed and compared the IPFS and Ethereum Swarm as potential basis for such storage. Having done the analysis, we concluded that IPFS is more suitable solution for the tampering-resilient document storage. The reasons for this choice were code maturity, serving the production use cases, strong community, better documentation and existing libraries to work with IPFS. Ethereum Swarm has more potential in its design and tight integration into Ethereum ecosystem, but it is in alpha version and many declared features are not implemented yet. We extended the baseline solution with a tampering-resilient document storage that is based on IPFS and described the implementation. Code can be found in the separate branch in the GitHub repository[22].

[22] https://github.com/kopylash/blockchain-real-estate-v1/tree/ipfs

5 Discussion

Any technology brings advantages and disadvantages and nothing could be achieved without cost. Therefore, in this chapter we will discuss the influence of blockchain integration in the developed proof-of-concept application from an engineering point of view. We will go through each step of the development and cover the problems, decisions and their impact.

5.1 On-chain vs Off-chain

Before proceeding to the actual design decisions we would like to discuss the blockchain integration in general. Encapsulating a part of the business logic in a smart contract can have many benefits. Ethereum provides a distributed platform for running decentralized applications. Therefore, the part of business logic encoded within a smart contract will be resilient and fault tolerant. Data stored in Ethereum blockchain cannot be deleted nor tampered, that means that it can become a source of truth in the system. Due to specifics of smart contract's deployment to the Ethereum network, one can be sure that the business rules encoded in a smart contract will be executed without mistakes and unpredicted situations[23]. Blockchain as underlying technology in Ethereum platform has certain limitations, like transaction speed and system scalability, because each transaction should be distributed to and validated by each peer in the network. Hence, sometimes it is not rational to handle everything with smart contracts. Let's consider a situation where we moved all the logic of our rental process to the smart contract and blockchain. By doing this, we will remove any intermediary party from the process. It will be possible for users to interact with each other directly using Ethereum wallet and Metamask to run decentralized applications in the browser. Money transfers can be handled using tokens of a custom cryptocurrency. Moreover, same tokens can be used to handle offer initiation and agreement signing. The result of such transformation could be faster real estate transactions at a lower price. However, it is not possible to cover all cases with help of blockchain. In the context of the real estate business in Singapore, where government owns 80% of the property and regulates the real estate market, it is hard to overcome the legal restrictions. Property owners must perform many eligibility checks on their own before entering into a real estate transaction with other parties. Besides, blockchain transaction representing a rental agreement, for instance, cannot be considered legal by Singaporean law[24]. It means that if two people agreed on a property renting in a decentralized peer-to-peer platform, with a corresponding transaction stored in the blockchain, and in the real world one of the parties broke the agreement, another party cannot send a claim to the court and bring the former one to the responsibility. Consequently, digital signing of PDF documents is used instead where it is possible to identify parties of the contract and value exchange takes place.

Another serious limitation is a speed of transaction. To provide the user with an appropriate user experience some operations should be very fast, within range of several seconds. For example, it is a common use case in a real estate platform to search for enlistments, filtering and ordering them. Having everything implemented in a smart contract will increase latency in this case, even considering the fact that performing a call in Ethereum does not cost any gas and it is processed immediately (it still takes some time, but much less than transaction) compared to a transaction that should be mined and verified by the network.

One more frequent use case is a search in a particular area, which requires complex spatial calculations with geographical coordinates. It may be too complicated to implement these

[23] During development people can make errors and introduce bugs, so deployed smart contract will have incorrect behavior. Usually, it is prevented with static code analysis and smart contract validation.
[24] Personal communication with the CEO of the real estate company on 26.04.2018

calculations in a smart contract comparing to other options usually provided within frameworks or databases, like PostGIS for PostgreSQL. Even if it is possible to implement needed functionality, it can significantly influence the architecture of the system and the operational costs, because deploying each smart contract we need to pay gas.

To recap we should note that considering moving a part of the business logic into a smart contract we should remember that there can be performance, complexity, cost and legal implications. Therefore, it is rational to leave certain parts of a business process off-chain.

5.2 Smart Contract Challenges

While capturing the business process in a smart contract it is necessary to define boundaries. Is it possible to handle everything in one smart contract or it is better to separate contexts? We faced this problem during the development of our *PropertyEnlistmentContract*. Having everything in a single smart contract may seem simpler. However, there is a gas limit per block in Ethereum. The network of peers defines the maximum number of transactions (and the amount of gas respectively) that can be fit into one block and appended to the blockchain. This number changes with time. The amount of gas required to deploy the smart contract cannot exceed block gas limit, otherwise it will not be possible to deploy the contract. If the contract is too big, it must be split into several ones. In the case of the developed rental application, there were several options: store users, enlistments, offers and agreements in a separate contracts or handle everything from the context of property enlistment in a single contract. First approach gave a lot of flexibility and independence to each entity, but it required much more interactions between different contracts. There are a lot of restrictions in the process regarding receiving offers and agreements as discussed in section 4.2.3. Following this approach would require those checks to be implemented between different contracts adding much overhead. Alternative option of handling everything in a single smart contract was simple and straightforward. It handles different aspects of the business process, therefore, may be hard for understanding. The only concern was about the size of a smart contract. Fortunately, current block gas limit is around 8 000 000 Gwei[25] and the created smart contract fits into this size. If there are doubts regarding the organization of smart contracts, our suggestion is to start from a single smart contract and then split having clear boundaries in mind. Preliminary separation will add more complexity to the development and at early stages boundaries can change a lot.

Another design decision that was made about the architecture of our rental application is that only system will interact with the smart contract. Alternative solution to this was to allow landlords and tenants interact in a peer-to-peer manner with each other. However, for that they needed an Ethereum wallet. Considering the state of Singaporean real estate market, such decision would create obstacles for people in adopting the platform because for them creating the wallet and managing Ether on their account would be an overhead. A possible way to leave the peer-to-peer approach and not to complicate the platform was to manage the Ethereum wallets on behalf of customers and gradually let them to do it themselves. Nevertheless, we were building the proof-of-concept of a hybrid application and it was a tradeoff to make the system only one interactor with a smart contract. In this case smart contract works as a fault tolerant, distributed automation for the part of business process.

One more challenge we already slightly touched is contract deployment. Once contract is created and deployed to the network it cannot be modified. Moreover, the gas is paid for the deployment of the contract and for its instantiation. There may be different architectural

[25] https://ethstats.net/ and https://etherscan.io/chart/gaslimit

approaches to the deployment of smart contracts. For instance, there can be only one contract deployed, like for Ethereum-based cryptocurrencies, when the deployed contract stores the balances of all addresses and manages the transfers between them. This approach is the simplest and does not require any additional efforts and costs. Another approach is a contract per entity instance, like in our rental application. We deploy a new contract for each verified property enlistment. In previous approach smart contract acts more like a registry mapping and handling everything. This approach allows to reduce the scope of smart contract to a single entity instance. The drawback of contract per instance is cost. We need to pay for each deployment and instantiation. If the contract is complex and big, this cost will be high. There is a pattern that can help to tackle this issue. It is called contract factory. The idea is very similar to the factory method pattern [15]. Instead of deploying the smart contract each time, the factory is deployed once. Then factory will be used to instantiate a contract for a specific property enlistment. This approach allows to reduce the deployment cost because it is paid only once when the factory is deployed. Then we need to pay only for instantiation of the new contract, but this cost is smaller than the deployment cost. Also, factory contract can be a registry of all created contracts and provide an interface to retrieve the address of the needed contract for further interactions.

We also wanted to discuss the use of structs and events in the smart contracts. The decision to handle property enlistment lifecycle in a single smart contract led to a complicated and big smart contract. Structs help to improve the readability of the code and create custom types that are easier to manipulate. However, there are some cost implications and limitations of Solidity regarding structs. When structs are used in events it requires more gas than passing flat values as event parameters. In the version of Solidity that was used during development (v.0.4.18) there is an issue with structs used in events. A struct retrieved from mapping cannot be passed to the event as an argument, a new struct must be created and passed. This led to the following code pattern in the smart contract.

```
event OfferUpdated(Offer offer);

function reviewOffer(bool result, string tenantEmail) payable public
{
    //...
    var offer = tenantOfferMap[tenantEmail];
    var typed = Offer(
           offer.initialized,
           offer.amount,
           offer.tenantName,
           offer.tenantEmail,
           offer.status
    );

    OfferUpdated(typed);
}
```

Recreating of a struct in such cases to pass it to event increased the cost of each operation. It was one of the reason why events were removed from the *PropertyEnlistmentContract*. Solidity is constantly developed and this issue may be already addressed. The last available version of Solidity is 0.4.24.

In first versions of the smart contract there were events to track significant actions like new offers or status updates. Having events in a smart contract can be beneficial, because other parties can listen for them and react. Later, we removed the events with an intention to reduce costs. The integration with a smart contract in our hybrid application does not rely on the events because Truffle hides the implementation of transaction status checks (it uses polling). The listeners attached to a function sending a transaction fire only when it is verified by network. So, there was no need in periodic checking for Ethereum event logs. To measure the impact, we calculated gas consumption at each step of a successful rental flow and used a low-priority mining price to receive the minimum cost of the rental flow. Without events the cost became 5 USD lower. The measurement and corresponding changes can be found in this pull request on GitHub[26].

5.3 Integration Challenges

Performance aspects

Building a hybrid proof-of-concept application we made many decisions with tradeoffs, trying to combine best attributes of on-chain and off-chain parts of the system. Blockchain integration has a significant impact on a traditional web application. Despite the fact we managed to encapsulate integration with Ethereum in a single service, there is one issue. The communication between client and server used to be a synchronous request-response in traditional web application. In the current architecture requests are handled by a web server and are proxied to the smart contract. There is a delay while transaction will be mined and appended to the blockchain. According to Etherescan average block time is around 14 seconds[27], but there is also a queue of transactions from 20 to 40 thousands per minute[28]. Thus, the HTTP request may timeout. To cope with this problem, the request-response communication between client and server should be changed to an asynchronous model, for instance, websockets. Handling part of the business process with a smart contract made it fault tolerant and resilient. However, it also increased the response time of the application. Appending transactions to the blockchain is time consuming. Even with asynchronous communication between client and server, the UX delivered to the customer is much different. Before, customers expected to see the result of interaction within several seconds, and now some action may take five minutes, for example. Scalability is one of the current problems Ethereum is trying to solve by increasing transaction block size and working on a concept of sharding [9].

The web server is a single point of failure in the current architecture of the developed hybrid application. This was a tradeoff for the decision to have an off-chain part of the system. However, it is not a big issue. Many web applications face it every day. To increase the fault tolerance of the system one can apply usual scalability patterns. For instance, horizontal scaling of the web server will remove the single point of failure. Of course, it may bring some complexity of a distributed system, but nothing can be achieved for free.

Data storing aspects

Deciding to move some parts of business logic to the blockchain we should also think how it will change our data model. Traditionally storing data outside of blockchain we had it in one place, and there was one distinct way of manipulating that data. When moving logic to a smart contract, we also need to move some part of our data to the blockchain. Moreover, there are many questions connected with data partition:

- should we store data only in one place or duplicate?
- what is the source of truth?

[26] https://github.com/kopylash/blockchain-real-estate-v1/pull/7
[27] https://etherscan.io/chart/blocktime
[28] https://etherscan.io/chart/pendingtx

- is it right to store that data in blockchain as it will be publicly available?

Because data in blockchain is immutable and is stored there forever, blockchain can be used as a single source of truth for certain parts of the application. Another option is to store data both in blockchain and in an off-chain database. However, there may be issues with synchronization of states between those two storages. Going further, last option can be considered as having the source of truth off-chain and backing up data to the blockchain, though off-chain storages do not benefit from blockchain characteristics. There also may be changes in a data model because smart contracts have certain limitations regarding storage and execution. Solidity implies minimalistic usage of resources providing a strict type system and syntax. Hence, there may be a need to reorganize how data is stored in a smart contract.

In the table below we compare the on-chain and off-chain from the data storing perspective. IPFS is presented as a separate option because it allows to mitigate the weak points of on-chain and off-chain data storing.

Table 1. Comparison of data storing options

Criteria	**Off-chain**	**On-chain**	**IPFS**
Tampering resilience	-	+	+
Availability	Low availability due to centralized storage	High availability via replication	High availability via replication
Structuring flexibility	Rich ADT support	Poor ADT support	Not structured
Volume	High	Low/costly	High

Storing data both on-chain and in IPFS makes it tampering-resilient because in blockchain the network verifies each change to the database, while IPFS is a content addressed storage, so a tampered data will have another address and will not affect the original source. Storing data off-chain is vulnerable to tampering conversely. Regarding the availability of the data, both on-chain and IPFS options provide high availability of the content due to their distributed nature. Off-chain storages are usually centralized or their scalability is relatively low comparing to the previous options. However, storing data off-chain has no limitations in form and structure due to the big number of abstract data types provided by different solutions. Blockchain, in its turn, has many restrictions concerning data structures that can be used within it. The last criteria of comparison is volume. Off-chain storages can handle large amounts of data and there is no explicit restriction for that. IPFS is a distributed file system, hence the aim of the nodes is to store and serve content. As each node stores only the needed files the capacity of the whole network is high. In terms of volume blockchain is very limited. Each peer in the network stores the whole or partial copy of the blockchain. Therefore, block size is restricted. Otherwise, blockchain volume would grow so much, that some peers will not have enough storage capacity to save it. Ethereum also has many memory limitations what is driven by underling blockchain technology: limited stack size, memory and storage size. Each of the aforementioned options has strong and weak sides, but combination of these options can overcome the limitations as shown in this work. In our solution we stored part of the data in PostgreSQL providing a location based search on top of that data. Another part of the data needed for smart contract was stored in the blockchain. IPFS was used to integrate tampering-resilient document storage because Ethereum blockchain and off-chain storage did not satisfy the requirements.

When storing data in a smart contract one should remember that this data will be available to everybody in the network. Blocks of transactions are stored in blockchain and every peer in the network has own copy of the blockchain. A peer cannot read the source of the smart contract because it is compiled into a bytecode, but all the variables and their values can be reverse engineered. Even setting a visibility level of a variable to private will not help because visibility level is used by Ethereum Virtual Machine at runtime and data can still be extracted from blockchain. One of possible solutions is to encrypt stored data. Encryption is computationally expensive process, therefore implementing it in a smart contract is costly. If the data is sensitive, it is much rational to store it off-chain and apply the necessary security protection.

6 Conclusion

In this thesis, we presented a case of Singaporean real estate company that wants to integrate blockchain into their core business processes. The company has already integrated rental contract signing with blockchain, storing hashes of signed document in it. However, the goal of the company is to adopt the smart contracts technology. We performed a descriptive case study, describing the particularities of the real estate business in Singapore, and modeled the company's business processes with the help of BPMN. After series of interviews with company's representative, we discovered that the value chain of the company consists of three parts. We discussed each of them in details and decided that the part covering the offers and rental agreements can be captured within a smart contract. We proposed the domain model for the modeled process and built the proof-of-concept of a hybrid application, integrating traditional web application with Ethereum smart contract. We proved the feasibility of smart contracts to handle complex business processes in a hybrid application. Also, we extended the solution with tampering-resilient document storage. For this, we analyzed and compared IPFS and Ethereum Swarm. We concluded that Ethereum Swarm is not mature yet and the implementation of tampering-resilient document storage on top of it is not feasible. Therefore, we integrated IPFS into our hybrid application. Finally, we discussed the problems and challenges we faced during the development of a hybrid application. We covered tradeoffs between moving a part of the process to the blockchain and leaving it off-chain. We discussed the challenges in smart contract implementation, proposed possible solutions, their implications and reasoned our decisions. Also, we discussed the architectural impact of blockchain integration and talked about its strong and weak sides.

To sum up, the result of this paper is the implementation of a proof-of-concept of a hybrid application that integrates Ethereum smart contracts with a traditional web application. The implementation is based on a case study of a Singaporean real estate company and is done in tight collaboration with the company (from the business side). This work summarizes the experience of applying technology to a case study and gives the overview of potential problems and possible solutions during the development of a blockchain-based real estate application.

Regarding the future work, there are many places for improvement. During the modeling phase, many things were removed from the scope of this work for simplification. Future research can be conducted to include timers and escrow mechanisms in the process. Due to the lack of time, we have not implemented the contract factory approach. However, it will be interesting to measure the actual cost reduction when applying this pattern for the deployment of smart contracts. The current implementation has no UI. It only exposes an API. Another future contribution can be an implementation of the UI for the hybrid application. Also, the real estate company we collaborated with expressed the interest in tokenization of the process handled by the smart contract. Therefore, the future work can address this question and research the ICO feasibility in the context of the case of Singaporean company. We used the IPFS as the basis for the tampering-resilient document storage, but it has a limitation in the replication of content over the network. A potential improvement can be an integration of Filecoin to add an incentivization layer and measure the operational costs for the storage. With the stable release of Ethereum Swarm it will be possible to implement a tampering-resilient document storage on top of it and compare with our IPFS based implementation.

.

7 References

[1] Blockchain in commercial real estate https://www2.deloitte.com/content/dam/Deloitte/us/Documents/financial-services/us-dcfs-blockchain-in-cre-the-future-is-here.pdf, accessed on 20.10.2017

[2] Blockgeeks. What is blockchain technology? https://blockgeeks.com/guides/what-is-blockchain-technology/, accessed on 26.11.2017

[3] Coindesk. What is blockchain technology? https://www.coindesk.com/information/what-is-blockchain-technology/, accessed on 26.11.2017

[4] Understanding the blockchain https://www.oreilly.com/ideas/understanding-the-blockchain, accesed on 26.11.2017

[5] Blockchain technology beyond Bitcoin http://scet.berkeley.edu/wp-content/uploads/BlockchainPaper.pdf, accessed on 26.11.2017

[6] Satoshi Nakamoto “Bitcoin: A Peer-to-Peer Electronic Cash System” https://bitcoin.org/bitcoin.pdf, accessed on 27.11.2017

[7] What is Ethereum Guide? https://blockgeeks.com/guides/ethereum/, accessed on 27.11.2017

[8] Peck, Morgan ""Hard Fork" Coming to Restore Ethereum Funds to Investors of Hacked DAO". *IEEE Spectrum: Technology, Engineering, and Science News*. IEEE. Accessed 27.11.2017

[9] TrustNodes “Vitalik Buterin Lays Roadmap for Ethereum Visa Levels Quadratic Sharding” http://www.trustnodes.com/2017/11/25/vitalik-buterin-lays-roadmap-ethereum-visa-levels-quadratic-sharding, accessed on 26.11.2017

[10] “Is real estate tach disruption the end of the property agent?” https://e27.co/real-estate-tech-disruption-end-property-agent-20170718/, accessed on 22.12.2017

[11] How technology can save you time and money on your next HDB resale transaction. https://e27.co/technology-can-save-time-money-next-hdb-resale-transaction-technology-20171002/, accessed on 22.12.2017

[12] Object Management Group (OMG). Business Process Model and Notation (BPMN) Version 2.0.

[13] "Real estate": Oxford English Dictionary online, accessed on 19.04.2018

[14] Juan Benet “IPFS - Content Addressed, Versioned, P2P File System (Draft 3)”, https://github.com/ipfs/papers/blob/master/ipfs-cap2pfs/ipfs-p2p-file-system.pdf, accessed on 03.04.2018

[15] Gamma, E. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley.

[16] Ethsphere. Introduction — swarm 0.2rc5 documentation. https://swarm-guide.readthedocs.io/en/latest/introduction.html, accessed on 15.04.2018

Appendix

I. Smart Contract Example[29]

```
pragma solidity ^0.4.22;

/// @title Voting with delegation.
contract Ballot {
    // This declares a new complex type which will
    // be used for variables later.
    // It will represent a single voter.
    struct Voter {
        uint weight; // weight is accumulated by delegation
        bool voted;  // if true, that person already voted
        address delegate; // person delegated to
        uint vote;   // index of the voted proposal
    }

    // This is a type for a single proposal.
    struct Proposal {
        bytes32 name;   // short name (up to 32 bytes)
        uint voteCount; // number of accumulated votes
    }

    address public chairperson;

    // This declares a state variable that
    // stores a `Voter` struct for each possible address.
    mapping(address => Voter) public voters;

    // A dynamically-sized array of `Proposal` structs.
    Proposal[] public proposals;

    /// Create a new ballot to choose one of `proposalNames`.
    function Ballot(bytes32[] proposalNames) public {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        // For each of the provided proposal names,
        // create a new proposal object and add it
        // to the end of the array.
        for (uint i = 0; i < proposalNames.length; i++) {
            // `Proposal({...})` creates a temporary
            // Proposal object and `proposals.push(...)`
            // appends it to the end of `proposals`.
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }
```

[29] https://solidity.readthedocs.io/en/v0.4.23/solidity-by-example.html

```
// Give `voter` the right to vote on this ballot.
// May only be called by `chairperson`.
function giveRightToVote(address voter) public {
    // If the first argument of `require` evaluates
    // to `false`, execution terminates and all
    // changes to the state and to Ether balances
    // are reverted.
    // This used to consume all gas in old EVM versions, but
    // not anymore.
    // It is often a good idea to use `require` to check if
    // functions are called correctly.
    // As a second argument, you can also provide an
    // explanation about what went wrong.
    require(
        msg.sender == chairperson,
        "Only chairperson can give right to vote."
    );
    require(
        !voters[voter].voted,
        "The voter already voted."
    );
    require(voters[voter].weight == 0);
    voters[voter].weight = 1;
}

/// Delegate your vote to the voter `to`.
function delegate(address to) public {
    // assigns reference
    Voter storage sender = voters[msg.sender];
    require(!sender.voted, "You already voted.");

    require(to != msg.sender, "Self-delegation is disallowed.");

    // Forward the delegation as long as
    // `to` also delegated.
    // In general, such loops are very dangerous,
    // because if they run too long, they might
    // need more gas than is available in a block.
    // In this case, the delegation will not be executed,
    // but in other situations, such loops might
    // cause a contract to get "stuck" completely.
    while (voters[to].delegate != address(0)) {
        to = voters[to].delegate;

        // We found a loop in the delegation, not allowed.
        require(to != msg.sender, "Found loop in delegation.");
    }

    // Since `sender` is a reference, this
    // modifies `voters[msg.sender].voted`
    sender.voted = true;
    sender.delegate = to;
```

```
        Voter storage delegate_ = voters[to];
        if (delegate_.voted) {
            // If the delegate already voted,
            // directly add to the number of votes
            proposals[delegate_.vote].voteCount += sender.weight;
        } else {
            // If the delegate did not vote yet,
            // add to her weight.
            delegate_.weight += sender.weight;
        }
    }

    /// Give your vote (including votes delegated to you)
    /// to proposal `proposals[proposal].name`.
    function vote(uint proposal) public {
        Voter storage sender = voters[msg.sender];
        require(!sender.voted, "Already voted.");
        sender.voted = true;
        sender.vote = proposal;

        // If `proposal` is out of the range of the array,
        // this will throw automatically and revert all
        // changes.
        proposals[proposal].voteCount += sender.weight;
    }

    /// @dev Computes the winning proposal taking all
    /// previous votes into account.
    function winningProposal() public view
            returns (uint winningProposal_)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }

    // Calls winningProposal() function to get the index
    // of the winner contained in the proposals array and then
    // returns the name of the winner
    function winnerName() public view
            returns (bytes32 winnerName_)
    {
        winnerName_ = proposals[winningProposal()].name;
    }
}
```

II. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Vladyslav Kopylash**,

(*author's name*)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

An Ethereum-based Real Estate Application with Tampering-resilient Document Storage ,

(title of thesis)

supervised by Luciano García-Bañuelos,

(supervisor's name)

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **21.05.2018**