Imperial College London
Department of Computing

Individual Project Report

Blockchain-based Smart Tenancy Agreements

Author:
Henry Cuttell

Supervisor:
Prof. William Knottenbelt

June 2017

Abstract

This project proposes a solution to reduce the bureaucracy involved when renting a property, aiming to improve the speed and ease of completing actions throughout the tenancy, from contract creation to deposit arbitration. Smart contracts and the Ethereum blockchain are utilised, enabling the tenancy agreement terms to be encoded and deployed to the blockchain, with signatures and payments sent and processed securely over the Ethereum network. The blockchain provides an immutable record of these interactions providing irrevocable proof of tenants signing a contract, along with the terms that were agreed upon. I have developed a responsive and easy-to-use web application to enable contract participants to seamlessly interact with the smart contracts representing tenancy agreements. Tenants, landlords, and arbitrators are able to perform a wide range of features via this application including issue reporting, notice creation, and dispute resolution respectively. I have presented the application described in this report to several leading companies in the blockchain industry, a property letting consultant, as well as at the Imperial College London Blockchain Forum, and have received very positive feedback. I have also obtained valuable suggestions from a user study and am continuing to further develop the application, whilst investigating the potential of taking it to market.

Acknowledgements

First of all, I would like to thank my supervisor, Professor William Knottenbelt, for his enthusiasm and invaluable advice; I have thoroughly enjoyed our project meetings.

I would also like to thank my second supervisor, Dr David Birch, who gave me excellent guidance during this project.

I am grateful to Gary Feger, of Woodward Estate Agents, for taking the time to meet and explain the technicalities of tenancy agreements at the start of this project, as well as his feedback and suggestions upon later demonstrating my application to him.

Finally I would like to thank my parents and sister, who have supported me not only during this assignment, but throughout my studies at Imperial College London.

Contents

Chapter 1

Introduction

1.1 Motivation

Renting a property often involves a great deal of bureaucracy and estate agent fees for both the tenant and landlord. When a property is let, the agreed terms such as rental duration and monthly payments due are laid out in tenancy agreements; a predominantly paper-based and hand-signed form of contract. Problems with the current system involve the excess of paper documents required, lengthy bank transfer times, and high estate agent fees. According to Barclays it can take up to a day when transferring funds between different banks or longer for larger amounts, which can incur extra charges [3]. Foxtons Estate Agents also currently charges £96 to renew a tenancy [33], which involves re-signing the existing tenancy agreement updated with any new terms. Furthermore, at the end of a tenancy, it can take up to 10 days for a tenant to receive back their deposit, with the potential for disputes to end up in court, taking even longer to resolve [83].

1.2 Objectives

The aim of this project is to create a system which tackles these issues, allowing for easy creation, signing, and execution of contract terms of tenancy agreements. This project will focus on the England and Wales *assured shorthold tenancy* (AST) contract, but should be suitable, with minimal adaptation, for any type of tenancy agreement. The goal is to implement this system using smart contract and blockchain technology, providing access to the platform for landlords and tenants through an online web application.

The key project objectives are as follows:

- Develop smart contracts to enable execution of tenancy agreement terms such as rent payment and deposit handling over the blockchain.
- Create a transparent system that will enable verification of a contract being signed and the terms that were agreed upon.

- Allow simple creation of tenancy agreements to be stored privately with the ability to be enforced as legal documents.
- Enable swift resolution of issues between the landlord and tenant, such as the boiler breaking down.
- Design and implement an easy-to-use and responsive user interface for both landlord and tenant interaction with the system.

1.3 Contributions

The main project contributions are as follows:

- A novel smart contract representing a tenancy agreement supporting a range of user interactions such as signing a contract, paying a deposit, and transferring rent payments. The smart contract removes the need for a trusted third party to store the deposit and securely holds the funds, allowing for end of tenancy arbitration via a designated arbitrator where the funds are released.
- Utilisation of the Ethereum blockchain in order to provide irrevocable proof of the contract clauses that were signed.
- Creation of a secure server to store confidential details about tenancy agreements whose hash value is recorded on the blockchain.
- A system allowing tenants to report issues to the landlord, the information and timestamp of which are logged on the blockchain as proof, with the ability for the landlord to confirm and resolve these issues.
- A prototype application to allow tenants, landlords and arbitrators to interact seamlessly with a smart tenancy agreement via a responsive and user-friendly interface.

Figure 1.1: 'To Let' signage in Harrow.

Chapter 2

Background Research

This section will explore the technology behind Bitcoin and the blockchain, followed by an in-depth look at smart contracts. I will also describe the background research on tenancy agreements and how the system will aim to implement this core functionality.

2.1 Cryptography

I will begin by giving a brief overview of hash functions and public key cryptography, and how they apply to cryptocurrencies such as Bitcoin and Ethereum.

2.1.1 Hash Functions

Hash functions are widely used in blockchain technology. They provide a way to deterministically produce a fixed length random value from arbitrary length input data. This process is computationally infeasible to reverse, so given a hash value it is practically impossible to obtain the original data. Current widely used hash functions such as SHA-2 and SHA-3 have extremely low probabilities of collisions occurring (where two different pieces of data map to the same hash value). A common use case of hash functions is for data integrity checks which identify whether some data has been modified since its corresponding hash value was generated, as slightly different input data will cause a completely different hash value to be output by the function. Figure 2.1 shows how a user's Bitcoin public key is hashed to obtain a Bitcoin address (the hashing function on the diagram actually consists of 2 hash functions: SHA-256 followed by the RIPEMD-160 hash function [88, Chapter 4]).

2.1.2 Public Key Cryptography

Public key cryptography is the technology that underpins the way currency is secured and transactions executed in many cryptocurrencies, including Bitcoin and Ethereum. The principle is that a user has a public key (available to anyone) and

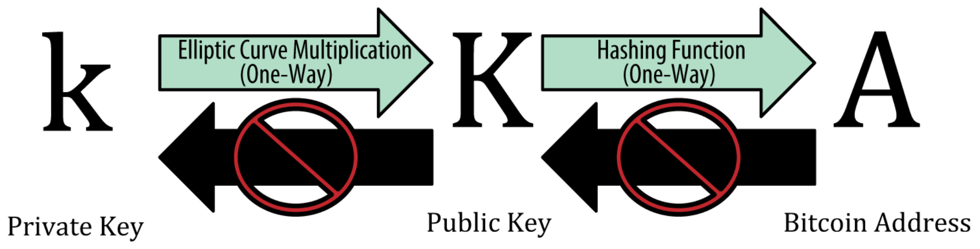

k
Elliptic Curve Multiplication (One-Way)
K
Hashing Function (One-Way)
A
Private Key
Public Key
Bitcoin Address

Figure 2.1: Generation of a Bitcoin address and public key from a private key [88, Chapter 4].

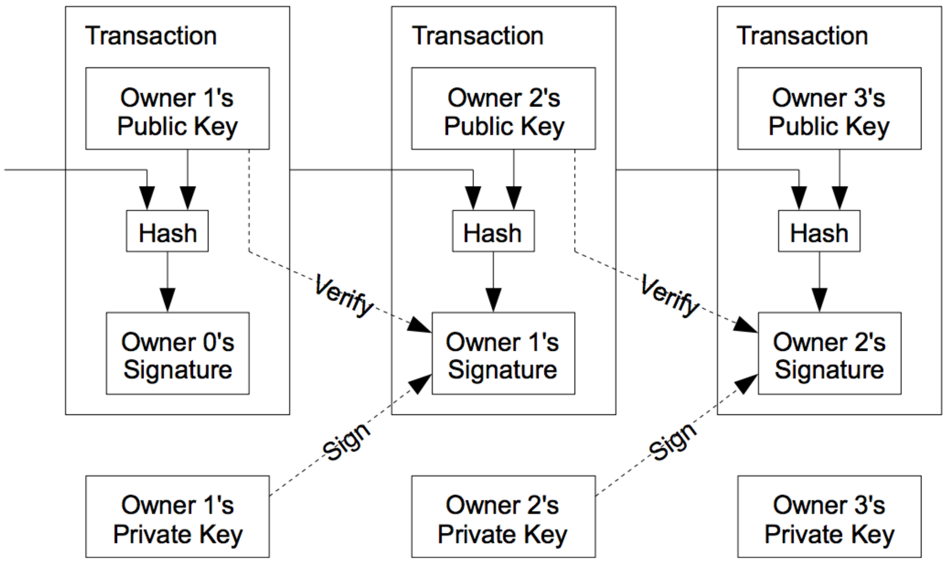


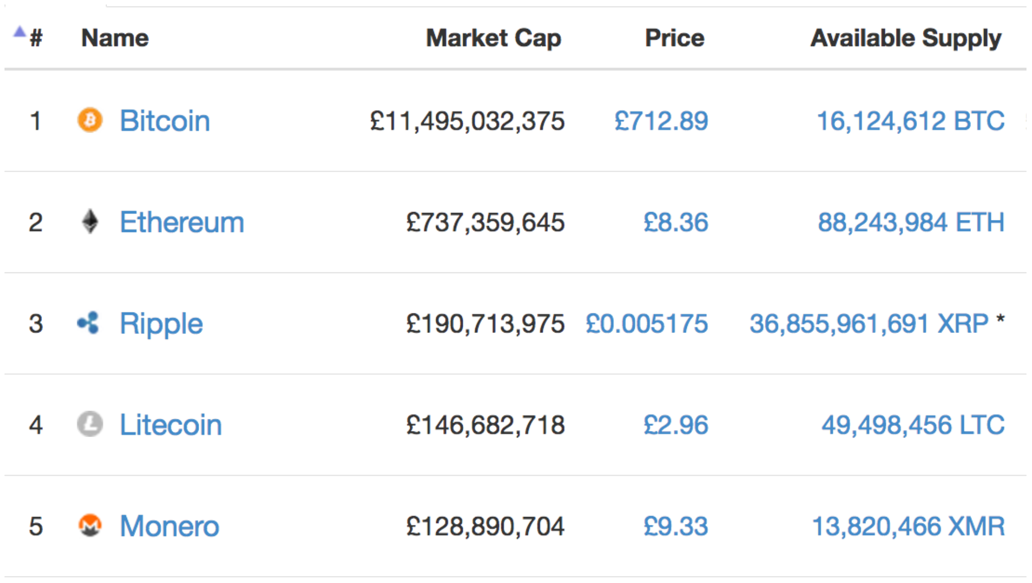
Figure 2.2: Chain of digital signatures [96].

a private key (must be kept secret). The private key is used to 'sign' transactions from accounts they own, with their public key being used to verify that they were the true sender for these transactions (without the private key being needed for the verification). Figure 2.2, from Satoshi Nakamoto's Bitcoin white paper, illustrates this concept showing a chain of transactions where each one can be verified to prove the chain of ownership.

Bitcoin uses elliptic curve cryptography to generate a user's public key from their private key; details can be found in *Mastering Bitcoin* by Andreas M. Antonopoulos [88, Chapter 4].

2.2 Bitcoin

Bitcoin was proposed in a white paper in 2008 by Satoshi Nakamoto and has been gaining in popularity ever since, with a market cap of £28 Billion as of May 2017 (see



#	Name	Market Cap	Price	Available Supply
1	Bitcoin	£11,495,032,375	£712.89	16,124,612 BTC
2	Ethereum	£737,359,645	£8.36	88,243,984 ETH
3	Ripple	£190,713,975	£0.005175	36,855,961,691 XRP *
4	Litecoin	£146,682,718	£2.96	49,498,456 LTC
5	Monero	£128,890,704	£9.33	13,820,466 XMR

Figure 2.3: Top 5 cryptocurrencies based on market capitalisations (as of January 2017) [10].

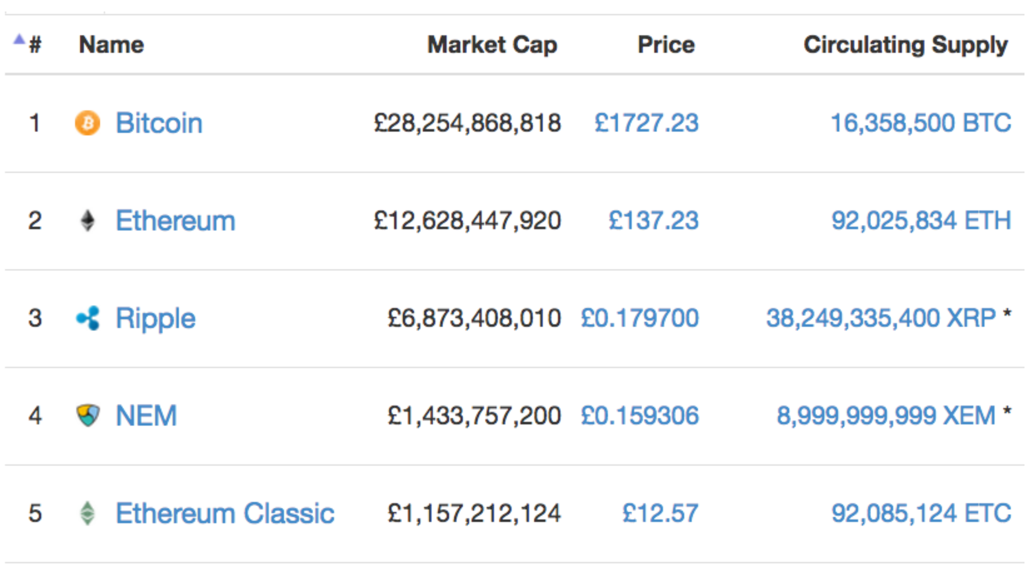
Figure 2.4). Many other cryptocurrencies, referred to as *Altcoins* have also recently surged in popularity, with ether rising in unit price from £8 in January 2017 to £137 in May 2017 (the Ethereum platform is described in section 2.3.1). Refer to Figures 2.3 and 2.4 for a comparison in the prices and market capitalisations of the top 5 cryptocurrencies for January 2017 and May 2017.

Satoshi Nakamoto's paper, titled "Bitcoin: A Peer-to-Peer Electronic Cash System" [96], describes the fundamental concepts of Bitcoin including transactions, proof of work, and the blockchain. In this section I will explain these in more detail as many of the technologies utilised in Bitcoin are also used in other cryptocurrencies such as Ethereum.

2.2.1 Transactions

Bitcoin transactions are flows of value, they have one or more inputs and one or more outputs. The value in the system is comprised of many *unspent transaction outputs* (UTXOs). The users that these UTXOs belong to are able to sign a transaction with their private key in order to transfer funds. These transactions consume UTXOs and output more UTXOs. Figure 2.5 provides an illustration of this.

A user's balance in Bitcoin is the sum of all of their scattered UTXOs. A Bitcoin wallet [5] handles all of the administration for these UTXOs and decides which ones to use when a new transaction is created [88, Chapter 5].



#	Name	Market Cap	Price	Circulating Supply
1	Bitcoin	£28,254,868,818	£1727.23	16,358,500 BTC
2	Ethereum	£12,628,447,920	£137.23	92,025,834 ETH
3	Ripple	£6,873,408,010	£0.179700	38,249,335,400 XRP *
4	NEM	£1,433,757,200	£0.159306	8,999,999,999 XEM *
5	Ethereum Classic	£1,157,212,124	£12.57	92,085,124 ETC

Figure 2.4: Top 5 cryptocurrencies based on market capitalisations (as of May 2017) [10].

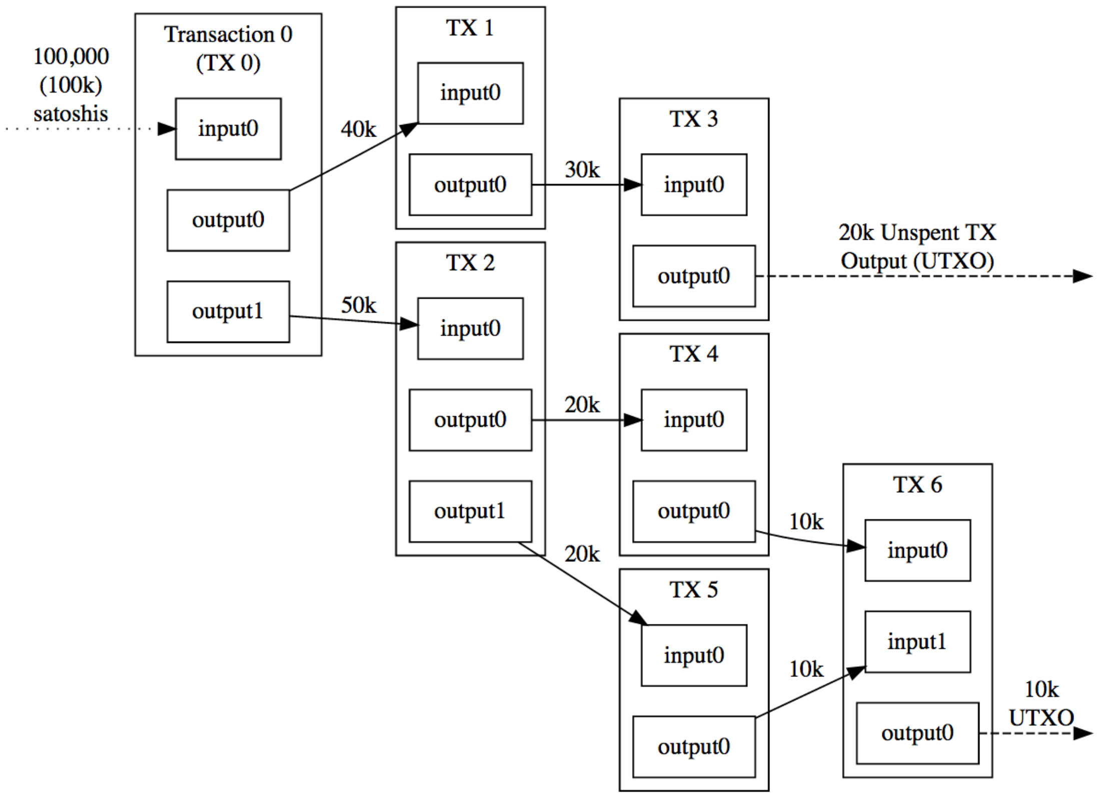


Figure 2.5: A group of transactions showing inputs, outputs, and UTXOs [4].

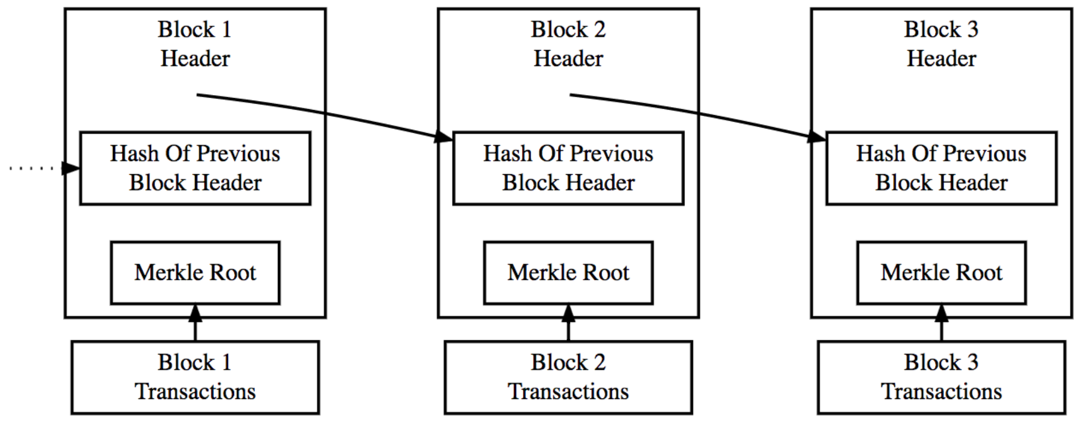


Figure 2.6: Simple visualisation of the Bitcoin blockchain [4].

2.2.2 The Blockchain

The blockchain is an immutable distributed ledger to which groups of valid Bitcoin transactions are added in ‘blocks’. Each block header contains a hash of the previous block’s header; this is the crucial part that means the blocks form a ‘chain’. This is also what makes the blockchain immutable, because if any part of a block’s header is changed, the value of its hash will change, thereby changing the value of its hash referenced in the next block, as well as all blocks that follow (see Figure 2.6 for a diagram of this). The way the protocol stops an attacker from re-writing chains of blocks is the concept of *proof of work* which will be explained in the following section.

Each block header also has a ‘Merkle Root’ field which provides an efficient way of hashing all of the transactions included in the block. A Merkle tree works by pairing the transactions and hashing them, followed by pairing these hash values and performing another hash, and so on forming a tree-like structure. This continues until there is a single hash, the Merkle root. The transactions therefore cannot be modified as the alteration would change all of the hash values propagating up the tree.

2.2.3 Proof of Work

The proof of work algorithm helps ensure that the blockchain cannot be rewritten, by requiring that all blocks have undergone significant computation in order to find a ‘nonce’ value for their block header. This nonce must have a value that causes the hash of the block header to be below a certain threshold. Consequently it takes many attempts of incrementing the nonce and using the SHA-256 hash function to calculate the hash of the block header, followed by checking whether or not the resulting value is below the threshold. The difficulty of this process correlates to how low this threshold value is, being dynamically adjusted so that a successful nonce is found approximately every 10 minutes (the frequency of a new block being added to the blockchain). This computation is very time-consuming to calculate but trivial to verify. The likelihood of blocks being overwritten by malicious actors decreases as successive blocks are added to the blockchain. Satoshi Nakamoto calculated the probabilities of this happening in the Bitcoin white paper and showed the probability drop off is exponential, so long as the majority of the network's computational power is controlled by honest nodes [96].

The network can be attacked if more than 51% of the nodes are controlled by colluding malicious actors. However, the potential attacker is incentivised to remain truthful because compromising the integrity of the network would cause the price of Bitcoin to fall, devaluing their own Bitcoin stake.

While Bitcoin features many desirable properties of an electronic currency, it would not be suitable for use in this project because of the limitations of its scripting language not being Turing complete. This will be required to encode the complex conditions and state transitions necessary in tenancy agreements.

2.3 Smart Contracts

2.3.1 Ethereum

Ethereum is a decentralised platform that shares many of the same core concepts as Bitcoin, such as digital transactions, the blockchain, and a proof of work consensus algorithm. However, the key difference is that Ethereum allows smart contracts to be executed on the nodes within the network. These smart contracts can be written by anyone and allow logic to be coded into the blockchain which is executed whenever the block containing the code is downloaded and validated by a node [89]. This technology enables the creation of *decentralised applications* or *ĐApps* whose contract code can be trusted to execute deterministically across all nodes in the Ethereum network. Example applications include crowdfunding for a product without a trusted third party, or a *decentralised autonomous organisation* (DAO) which carries out proposals democratically voted for by its members [18]. This section will explore more in-depth details about Ethereum.

Ether and Gas

The currency of the system is *ether* which is used to pay for the transaction fees and the computation involved when the smart contracts are run [19]. The units of ether are as follows:

- 1 - wei
- 10^{12} - szabo
- 10^{15} - finney
- 10^{18} - ether

Gas is the unit of computation with an associated price in ether, and so different operations in smart contracts have different gas costs, with more expensive operations requiring more gas to complete. The individual costs for the all of the possible operations can be seen in the Ethereum Yellow Paper by Dr Gavin Wood [99].

Account Types

There are two types of accounts that are available in the Ethereum network: *externally owned accounts* and *contract accounts*. Externally owned accounts are owned by individuals and are controlled by their associated private key, whereas contract accounts are controlled by their contract code. Both types of account are referred to by their 20-byte address and have an ether balance associated with them [89]. The accounts also store a *nonce*, to prevent double spending, as well as a storage field [14]. Externally owned accounts are able to send transactions whereas contract accounts can only send messages. Only contract accounts can have corresponding code which is executed upon receiving a transaction or message.

Transactions and Messages

Transactions sent by externally owned accounts must be signed with their corresponding private key and can transfer ether or call code in contract accounts. In addition they can also create new contracts on the blockchain. The transactions are written to the blockchain and can trigger further message calls to occur in contract code.

Messages are only sent by contract accounts and are not written to the blockchain; they are internal to the Ethereum network [89]. In addition to ether, data can also be sent with messages or transactions.

Ethereum smart contracts are Turing complete and so the code has the potential to infinite loop. This would be a waste of network resources and so the way Ethereum prevents this from occurring is by having a gas limit `STARTGAS` for every transaction and message. This means gas will be used up during the computation and if it runs out before completion, the state is rolled back to before the transaction or message was sent (minus the fees which are paid to the miner) [89]. The sender of a transaction can specify the fee paid with the `GASPRICE` field (a higher fee will generally mean the transaction is executed by miners faster [92]).

State Transitions

The Ethereum state transition function, displayed in Figure 2.7, shows a transaction applied to an initial state; it is executed by the network miners and is as follows:

1. The incoming transaction must be correctly formed. Checks are done to ensure the field count, signature (checked against the sender's public key), and nonce (must match the value of the sender's account nonce) are correct.

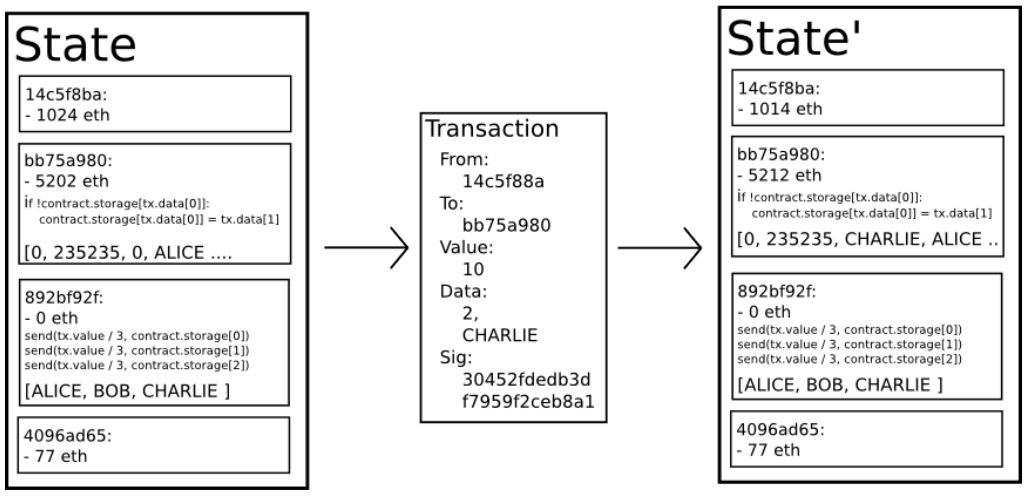

State
14c5f8ba:
- 1024 eth
bb75a980:
- 5202 eth
if !contract.storage[tx.data[0]]:
contract.storage[tx.data[0]] = tx.data[1]
[0, 235235, 0, ALICE ....
892bf92f:
- 0 eth
send(tx.value / 3, contract.storage[0])
send(tx.value / 3, contract.storage[1])
send(tx.value / 3, contract.storage[2])
[ALICE, BOB, CHARLIE]
4096ad65:
- 77 eth
Transaction
From:
14c5f88a
To:
bb75a980
Value:
10
Data:
2,
CHARLIE
Sig:
30452fdedb3d
f7959f2ceb8a1
State'
14c5f8ba:
- 1014 eth
bb75a980:
- 5212 eth
if !contract.storage[tx.data[0]]:
contract.storage[tx.data[0]] = tx.data[1]
[0, 235235, CHARLIE, ALICE ..
892bf92f:
- 0 eth
send(tx.value / 3, contract.storage[0])
send(tx.value / 3, contract.storage[1])
send(tx.value / 3, contract.storage[2])
[ALICE, BOB, CHARLIE]
4096ad65:
- 77 eth

Figure 2.7: An example Ethereum state transition [89].

2. The transaction fee maximum amount is calculated to be `STARTGAS * GASPRICE`, the cost of unused gas will be refunded to the sender in a subsequent stage. This fee is subtracted from the sender's ether balance and their account nonce is incremented.

3. The current gas amount is initialised to the value of `STARTGAS` and the gas cost for the bytes representing the transaction is subtracted from the gas value.

4. Any ether sent with the transaction is subtracted from the sending account balance and added to the the balance of the receiving account. The contract code of the receiving account is also run if it is a contract account. The code is executed, depleting the gas value per computation step, until completion or once the gas supply has been exhausted. If the receiving account does not exist, it is created in this step.

5. If the sender had an insufficient ether balance, or code execution did not complete due to lack of gas, the transaction is rolled back. All changes to the state are restored apart from the payment of fees, which are credited to the miner of the transaction. Alternatively, on success the sender is refunded the surplus fees for unused gas.

In addition, if any of the checks for a sufficient balance or valid signature fail at any stage, an error is returned to the sender of the transaction.

Smart Contract Execution

Smart contracts are generally written in higher level languages such as *Solidity* or *Serpent* which compile to low-level "Ethereum virtual machine code" (EVM code)

[89]. This is a bytecode language which is stack-based where each byte corresponds to an operation that is executed by the Ethereum virtual machine (see [99] for more details). EVM code has access to the following three storage locations:

- Stack: last-in-first-out data structure with a maximum depth of 1024.
- Memory: A byte array which can be infinitely expanded. It is cleared between external function calls.
- Storage: Persistent key-value store. Data stored here remains in the Ethereum state even after a contract finishes execution.

EVM code is executed in the Ethereum network whenever a transaction is received causing a contract to execute. This execution is part of the state transition function described in the previous section. The state transition function is part of the block validation algorithm (described in the next section) and so the EVM code is executed whenever blocks are validated by the miners. This is a point about Ethereum which can be hard to grasp at first - the fact that specific contract code is executed on *every* node in the network that downloads and validates the block containing the transaction that calls the code [89].

Mining

The job of the miners is to secure the network and agree on a single history of the blockchain. Their computational power is used to ensure nobody can 'rewrite' the blockchain history, by computing the proof of work for each new block that is added. The miner that successfully computes the proof of work for a block at a given difficulty is rewarded with 5 ether.

The latest version of the proof of work algorithm is *Ethash* which involves computing a "directed acyclic graph" (DAG), requiring a large amount of memory [16]. This aims to be ASIC resistant (application-specific integrated circuit) to help reduce centralisation of the network (when the majority of the mining is done by few entities). This reduces the likelihood of a 51% attack, whereby the blockchain history could be manipulated. Hence more users are able to run this algorithm on conventional CPUs and GPUs, promoting decentralisation.

In the Ethereum network the miners produce blocks which are then verified by others using the block validation algorithm (currently approximately every 14-18s [17]). The algorithm summary is described as follows (the specification is defined in the Ethereum Yellow Paper [99] and implementation details can be found in the Ethereum wiki [21]):

1. Check the previous block is valid. This involves checking the current block's "previous block hash" field matches the hash of the previous block.
2. The following fields of the current block are then checked for validity:

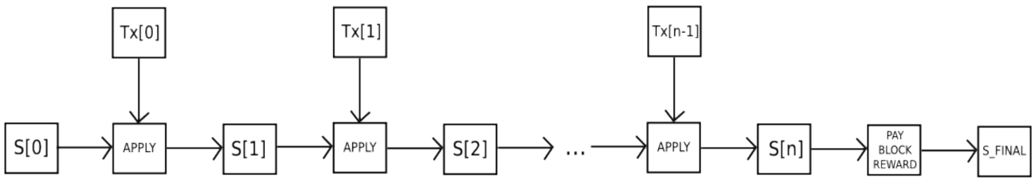


Figure 2.8: Validation of a block in the Ethereum blockchain [89].

- **Timestamp** - must be greater than the previous block timestamp and less than 15 minutes ahead of the current time.
- **Block number** - must be one greater than the previous block number.
- **Difficulty, Transaction Hash, Uncle Hash, and Gas limit** - must all satisfy validity checks (details can be found in the Ethereum Yellow Paper [99]).

3. The block proof of work must be valid.
4. The transactions in the block's transactions list are applied on top of the previous block end state `S[0]` one at a time - see Figure 2.8. If any of the state transitions return an error, or if at any point the cumulative gas consumed exceeds the `GASLIMIT`, an error is returned.
5. In Figure 2.8 the state `S_FINAL` is the end state `S[n]` but with the miner's reward paid.
6. The block is only valid if the Merkle tree root of `S_FINAL` matches the final state root in the block header.

To learn more about mining and to provide a source of ether funds for this project, I started mining with my desktop computer consisting of an AMD 3GB GPU (useful for the parallel and memory hard mining algorithm). I joined a mining pool (see Figure 2.9) which removes the luck associated with being the winner to mine a block. The pool combines all of the mining power and distributes the funds proportionally according to the computational power each miner contributes, leading to a lower but consistent income of ether. In the event of a pool's mining power becoming 51% of the network, miners should leave and join an alternative pool.

The Ethereum Blockchain

Ethereum uses specialised Merkle trees called Merkle Patricia trees to store the state and transactions, allowing efficient insertion and deletion of items [89]. Figure 2.10 shows how the state is shared between adjacent blocks using pointers to data which remained the same. This massively reduces the amount of state data which needs to be stored. Full implementation details can be found on the Ethereum wiki [24].

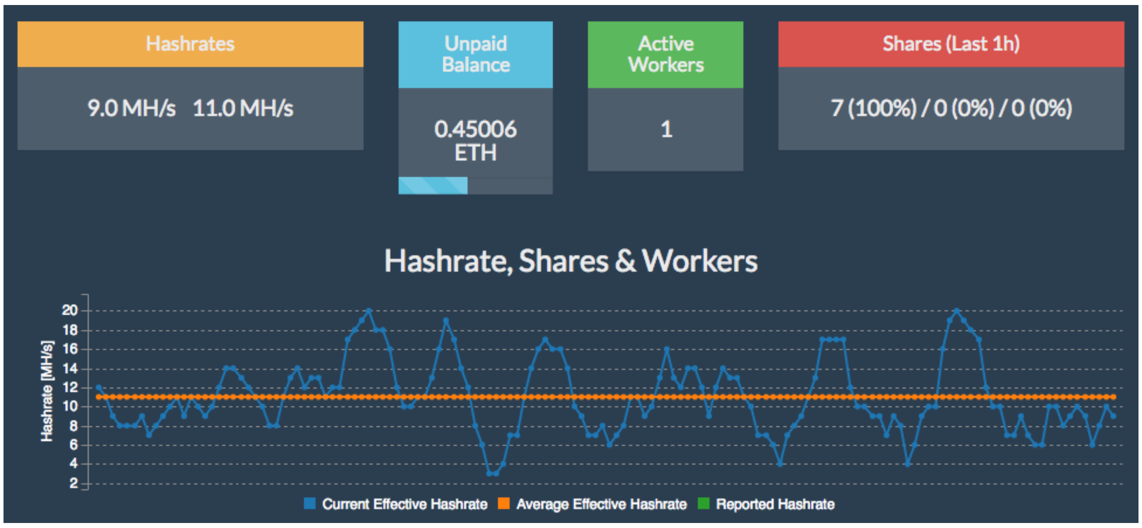


Figure 2.9: ethermine.org dashboard [28]

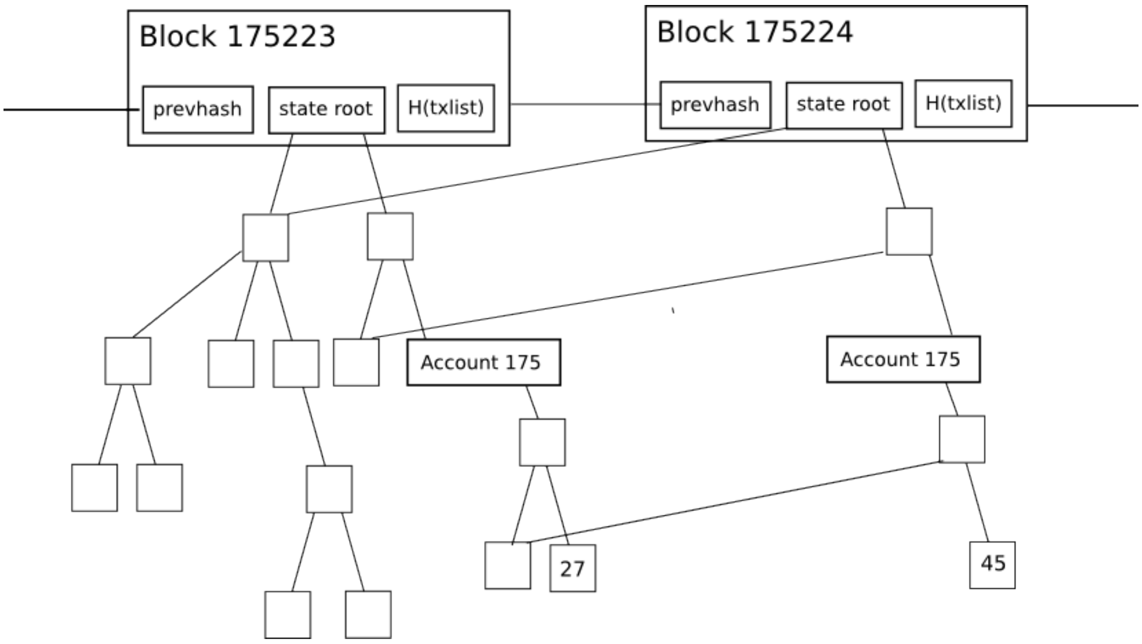


Figure 2.10: Shared state between adjacent blocks in the Ethereum blockchain [91].

Development and Future Releases

The Ethereum Project has a four stage roadmap: Frontier, Homestead, Metropolis, and Serenity [94]. Frontier was the initial beta release which introduced the core Ethereum functionality such as the ability to deploy smart contracts and mine ether. We are currently in the stable Homestead release which launched in March 2016 and has included several big updates to the network, including security improvements.

The upcoming Metropolis release will be targeted towards non-technical users of Ethereum and will see the production release of the Mist browser. This is how users will interact with the ÐApps which will be accessible via a ÐApp store [94].

The final release milestone will be Serenity which will come with several big changes to Ethereum, including changing of the proof of work consensus algorithm to *proof of stake*. This will reduce the electricity consumption of the network as it will no longer require huge amounts of computational power. Users will instead have a 'stake' in the network proportional to the amount of ether they hold, and will be selected to be the next block validator according to this value [25].

ÐApps

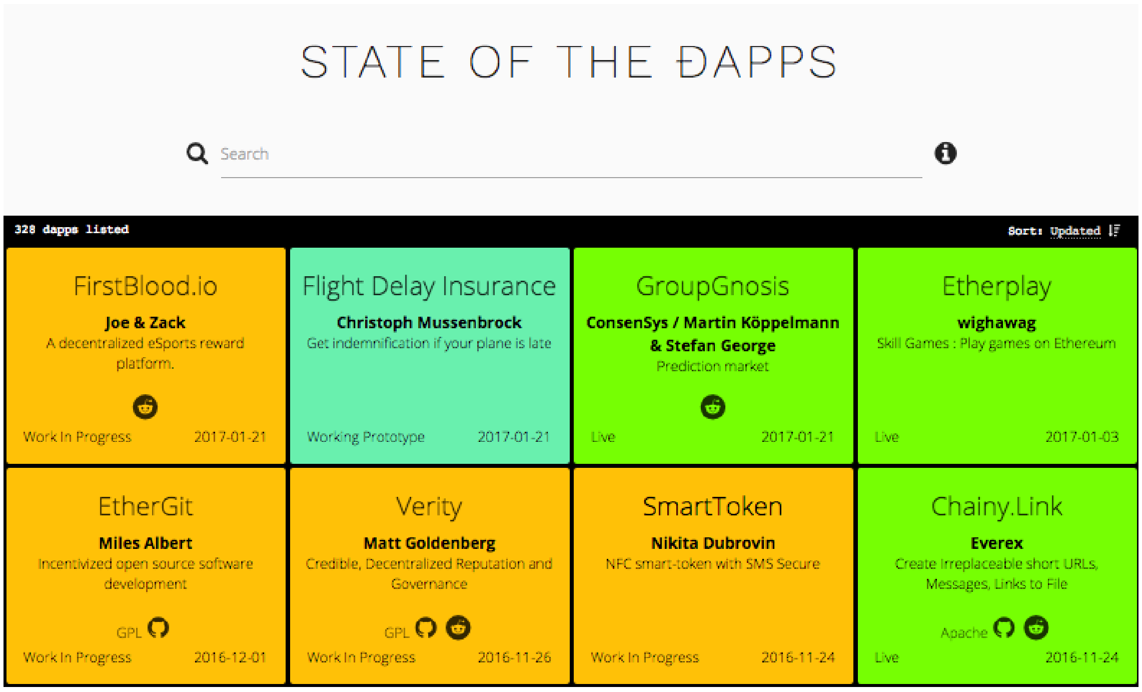
Decentralised applications or ÐApps are the way the average user will interact with the Ethereum network. There are already many ÐApps that are online (or still in development) and can be viewed on the "State of the ÐApps" web page [82]. These applications include marketplaces, DAOs, exchanges, and betting platforms - see Figure 2.11.

In order access the intended features of these decentralised applications, the user needs to be connected to an Ethereum node. There are a wide range of Ethereum clients available which can be downloaded and run on the user's computer. The main implementation is the Go version (geth / go-ethereum) but there are versions written in Rust (Parity), C++ (cpp-ethereum), and many others [15]. It is the job of the client to download the blockchain and verify new blocks, as well as providing a command-line interface for managing the user's Ethereum accounts, sending ether, or deploying smart contracts. The user also has the option of performing mining with the client.

The browser interacts with the client through a JSON RPC API [23], with most ÐApps using the JavaScript wrapper API [27]. However, these clients are more aimed at developers and end users will use the Mist Browser which has the Go client bundled with it [36].

The Mist Browser

The Mist browser is currently available as a pre-release version, able to browse ÐApps, sync to the Ethereum blockchain, and access the Ethereum Wallet. It


STATE OF THE ĐAPPS

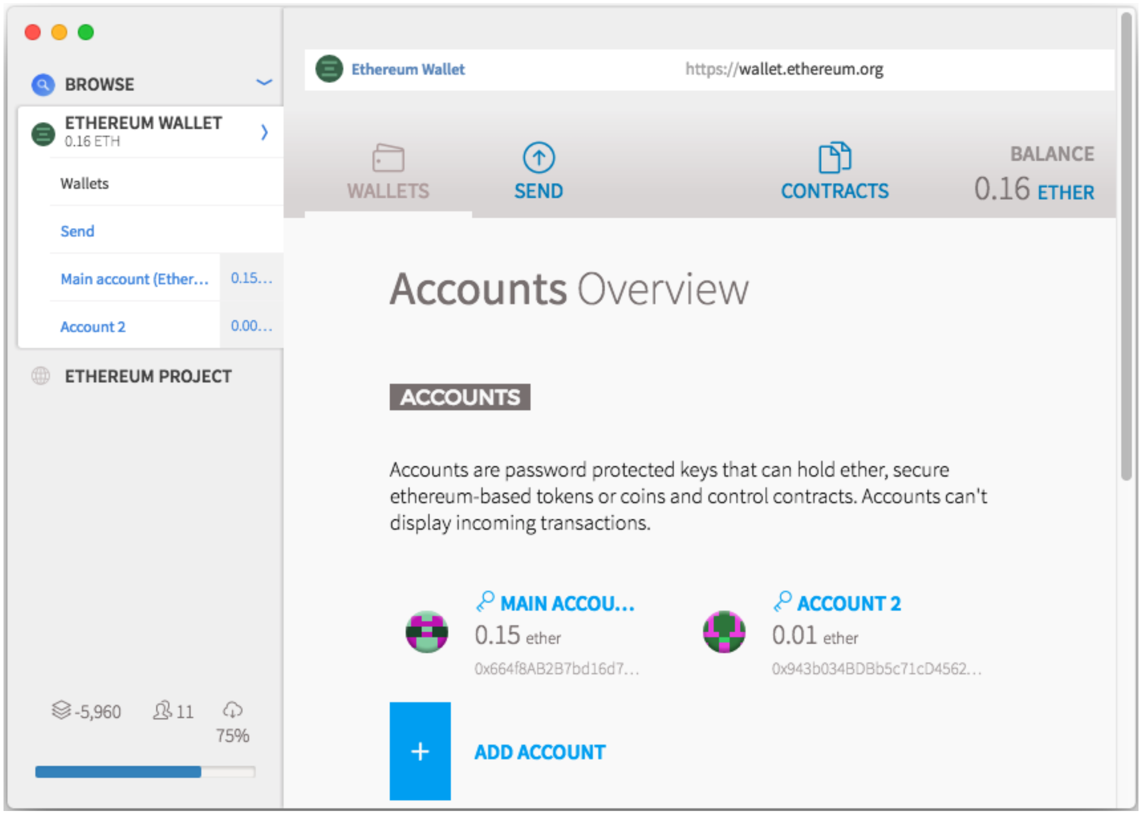
Search

328 dapps listed

Sort: Updated

FirstBlood.io Joe & Zack A decentralized eSports reward platform. Work In Progress 2017-01-21	Flight Delay Insurance Christoph Mussenbrock Get indemnification if your plane is late Working Prototype 2017-01-21	GroupGnosis ConsenSys / Martin Köppelmann & Stefan George Prediction market Live 2017-01-21	Etherplay wighawag Skill Games : Play games on Ethereum Live 2017-01-03
EtherGit Miles Albert Incentivized open source software development GPL Work In Progress 2016-12-01	Verity Matt Goldenberg Credible, Decentralized Reputation and Governance GPL Work In Progress 2016-11-26	SmartToken Nikita Dubrovin NFC smart-token with SMS Secure Work In Progress 2016-11-24	Chainy.Link Everex Create Irreplaceable short URLs, Messages, Links to File Apache Live 2016-11-24

Figure 2.11: ĐApps currently available or in development [82].



Accounts Overview

ACCOUNTS

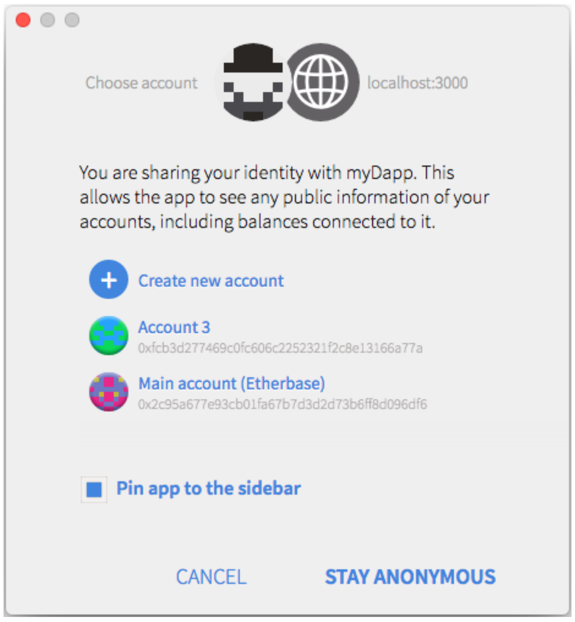
Accounts are password protected keys that can hold ether, secure ethereum-based tokens or coins and control contracts. Accounts can't display incoming transactions.

MAIN ACCOU…
0.15 ether
0x664f8AB2B7bd16d7…

ACCOUNT 2
0.01 ether
0x943b034BDBb5c71cD4562…

ADD ACCOUNT

Figure 2.12: The Mist browser showing the Ethereum Wallet ĐApp [36].



You are sharing your identity with myDapp. This allows the app to see any public information of your accounts, including balances connected to it.

Account 3
0xfcb3d277469c0fc606c2252321f2c8e13166a77a

Main account (Etherbase)
0x2c95a677e93cb01fa67b7d3d2d73b6ff8d096df6

Figure 2.13: The ‘connect account’ dialogue box in the Mist browser [36].

provides a fluid user interface, similar to a regular browser with page navigation and tabs, but enables users to interact directly with the Ethereum network. When accessing a ÐApp, the user has the option of allowing it access to their account’s public information (including balance) for a more tailored experience. The dialogue box is displayed in Figure 2.13.

Mist can be run with the main Ethereum network or a test network which is used for developers to test their applications without using real ether (ether on the testnet has no monetary value).

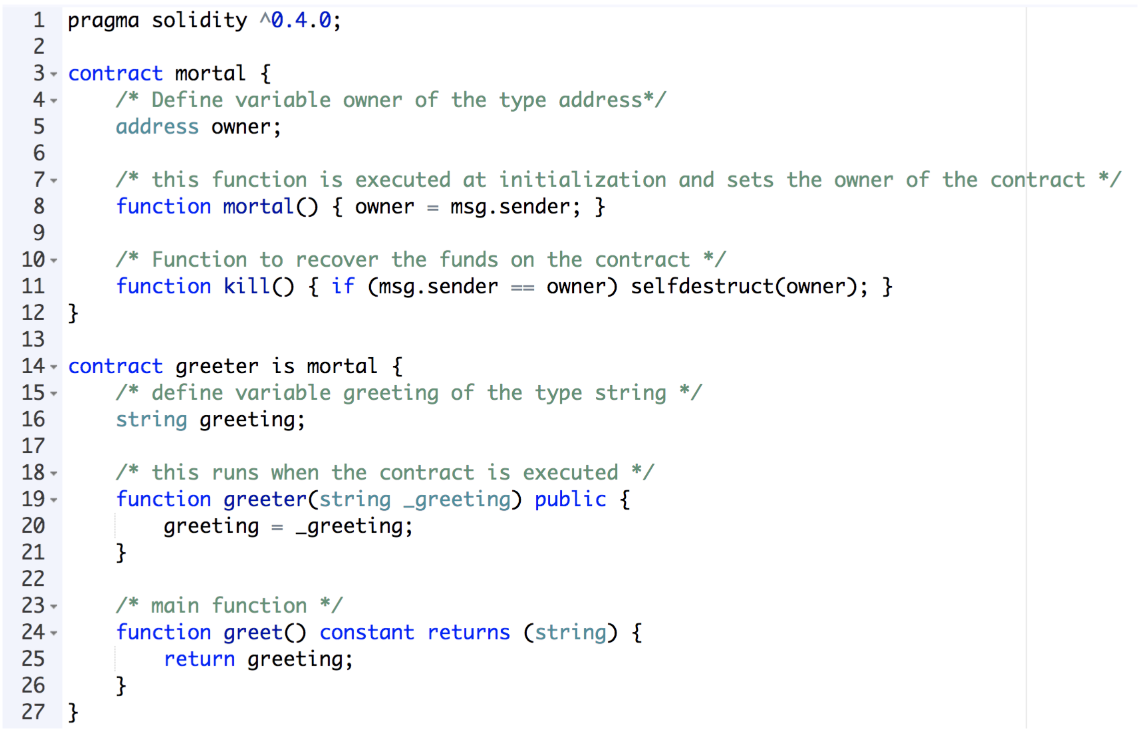
The Ethereum Wallet

The Ethereum Wallet ÐApp comes bundled within the Mist browser and provides a graphical user interface for users to manage their Ethereum accounts and ether funds. It provides interfaces for users to create new accounts, send ether, and deploy smart contracts. The main accounts overview screen can be seen in Figure 2.12.

Users can also download the Ethereum Wallet as a standalone application without downloading the browser, although this is likely to change with the Ethereum Metropolis milestone when Mist will see a production release [94].

Solidity

Solidity is the main high-level JavaScript-like programming language used to code the smart contracts for decentralised applications [75]. Figure 2.14 shows an example ‘greeter’ contract which inherits from the ‘mortal’ contract (code from a tutorial on



```
 1 pragma solidity ^0.4.0;
 2
 3 contract mortal {
 4     /* Define variable owner of the type address*/
 5     address owner;
 6
 7     /* this function is executed at initialization and sets the owner of the contract */
 8     function mortal() { owner = msg.sender; }
 9
10     /* Function to recover the funds on the contract */
11     function kill() { if (msg.sender == owner) selfdestruct(owner); }
12 }
13
14 contract greeter is mortal {
15     /* define variable greeting of the type string */
16     string greeting;
17
18     /* this runs when the contract is executed */
19     function greeter(string _greeting) public {
20         greeting = _greeting;
21     }
22
23     /* main function */
24     function greet() constant returns (string) {
25         return greeting;
26     }
27 }
```

Figure 2.14: Example Solidity contract code in the browser compiler [20, 74].

the Ethereum Foundation website [20] compiled using the browser-based compiler [74]). The greeter contract allows anyone to call `greet()` (line 24) which returns the greeting that was set when the contract was created. In addition, the mortal contract enables the creator to destroy to the contract using `kill()` (line 11), removing the contract from the blockchain and returning any held funds to the owner.

Solidity provides users with the means to create diverse and sophisticated smart contracts which can be programmed to execute different logic upon receipt of different transactions from different senders. This deterministic execution can remove the need for trusted third parties in various scenarios, which, combined with the fact that all transactions are recorded in the immutable blockchain, can provide proof of transactions being received and corresponding logic executed.

2.3.2 Rootstock

Rootstock [73] is a sidechain of the Bitcoin blockchain and, like Ethereum, allows Turing complete smart contracts to be run on its network. It uses 2-way peg technology, whereby the value of one Rootstock coin is equivalent to one Bitcoin, being created by effectively locking up Bitcoin, with the reverse also being true. The Rootstock Virtual Machine (RVM) executes the smart contracts and is compatible with the Ethereum Virtual Machine so Ethereum smart contracts are able to be run [95]. The Rootstock network is secured by 'merge-mining' with the Bitcoin network, where miners can simultaneously mine both chains and receive rewards for both. Rootstock's goal is also to support a high throughput of transactions and currently supports 100 transactions per second [73].

2.3.3 Smart Contract Discussion

There are also other smart contract platforms besides Rootstock and Ethereum, but they are either immature in development progress, or do not possess substantial advantages over Ethereum or Rootstock. Rootstock was not released in time to be used with this project but I will continue to investigate using it in the future as it is compatible with smart contracts written in Solidity for Ethereum. I have therefore decided to use the Ethereum network and its smart contracts as the basis for this project. Not only is it the most mature smart contract platform, but it has excellent development frameworks, documentation, and a thriving user base. As this project will involve users paying their rent and signing tenancy agreements, it is vital that the underlying technology is widely adopted and trusted amongst the community. Users will have the peace of mind that the project is being constantly developed and improved [90], having already had numerous successful security audits [98].

2.4 Related Work

There are a few related projects that have been developed which share some common features of this project. In this section I will discuss the similarities they have with my project requirements and objectives.

2.4.1 Express Agreement

Express Agreement [100] was developed in 2015 by Tina Zhang, an Imperial College London MSc student. The project uses the Bitcoin blockchain to record the signing of non-disclosure agreements and to store the hash value of the NDA clauses. The method uses a multi-signature Bitcoin address generated from the public keys of the parties involved in the agreement. The proof that all parties have signed the contract is represented by a transaction from this multi-signature address (which must have been signed with the private keys of all parties). The transaction uses a Bitcoin 'OP_RETURN' opcode which can be used to encode arbitrary data into the blockchain, in this case the hash of the clauses in the contract. Therefore this transaction can be looked up in the blockchain to verify the Bitcoin addresses involved in signing the NDA, with the hash stored in the transaction proving that a given set of clauses were agreed upon.

This project is similar in the fact that I will also be using the blockchain as a way of proving that an agreement has been signed by multiple parties, but it will instead be on the Ethereum blockchain using smart contracts to provide the means of signing the agreement.

2.4.2 Midasium

Midasium: The Blockchain of Real Estate [52] offers a service which uses smart contracts to represent real estate agreements (mortgage agreements, contracts of sale, and tenancy agreements). These are stored and automatically executed on the private Midasium Blockchain [53]. This enables tenancy agreements to be created which automatically transfer fiat currency from the tenant's account to the landlord's account.

The smart tenancy product offered by Midasium is similar to this project as it uses smart contracts to encode tenancy agreements and the blockchain to provide immutable proof of the contract agreed upon. However, one of the goals of this project is to develop a system on a public blockchain, allowing anyone to check for the existence and validity of a given contract. It is much harder to prove that a private blockchain is operating in a trustworthy manner because you have to trust the party operating it.

2.4.3 Replacing Paper Contracts with Ethereum Smart Contracts - Proof of Concept

This paper [93] aimed to provide an answer as to whether paper contracts such as tenancy agreements could be encoded on the Ethereum blockchain. The capabilities of Ethereum were investigated and a proof of concept for a tenancy agreement smart contract was created. The paper concluded that it is not advisable to place these agreements on the blockchain citing the reason that all of the data is public, thereby compromising the privacy of the contract.

This raises an important question about the privacy implications of storing contract logic on the Ethereum blockchain. However, my project aims to encode the basic conditions of the contract in the publicly visible smart contract, which should not contain any information which enables the parties involved to be de-anonymised. The available data will be rental payments to and from Ethereum addresses, which does not easily enable users to be mapped to the account address used in the contract. The sensitive contract information will be stored encrypted off the blockchain with the hash of this contract stored in the smart contract (using the same method as Express Agreement [100]).

2.4.4 Proof of existence

Proof of existence [70] allows users to store a hash of a document on the Bitcoin blockchain via an 'OP_RETURN' transaction. The website allows users to select a file, with its SHA-256 hash value being calculated client-side and subsequently stored on the immutable blockchain. This proves that the document existed at the time that its hash was recorded on the blockchain, due to the second preimage resistance of the SHA-256 hash function. Second preimage resistance means that it is computationally infeasible to find alternative data to the given original data, where

Figure 2.15: Government-backed tenancy deposit scheme logos (left to right): the Deposit Protection Service, MyDeposits, and the Tenancy Deposit Scheme [13, 54, 84].

the alternative data is different and has an identical hash value. I plan to utilise this general concept of storing a hash value on the blockchain in this project, applied to the case of recording the hash value of tenancy agreement contract clauses.

2.5 Tenancy Agreements

In order to learn more about tenancy agreements, and to gain an understanding of the requirements that the project will need to fulfil, my project supervisor and I met with Gary Feger, a Property Letting Consultant at Woodward Estate Agents in Harrow (Figure 2.17) [1]. We had an in-depth discussion about many aspects of the renting process, including discussing the most common types of tenancy agreements, problems that can arise between the landlord and tenant, and the checks that need to be done before a property can be let. I will outline the points that we discussed in this section.

Gary told us that the most common type of tenancy agreement is the ‘assured shorthold tenancy’ (AST). This agreement must be written in plain English and be of mutual benefit to both the tenant and the landlord. He also explained the differences between the different types of deposit schemes such as insurance backed (landlord keeps the deposit and pays a premium) versus custodial schemes (deposit is transferred to the trusted third party scheme). There are three government-backed deposit schemes [83]: the Deposit Protection Service (DPS), MyDeposits, and the Tenancy Deposit Scheme (TDS) [13, 54, 84]. The landlord must ensure they comply with the law and put the deposit in one of the approved protection schemes within 30 days of receiving it [83]. At the end of the tenancy, the deposit (minus any agreed deductions) must be returned within 10 days to the tenant [83], but can take longer in the case of disputes which can end up in court. Handling the deposit is something I will be looking to integrate into the smart contracts, because they provide a perfect method of holding funds without the need for a trusted third party. The main tenancy agreement stages are illustrated in Figure 2.16.

Issues that can arise with tenancies often involve the tenant failing to pay the rent. If the tenant cannot afford the rent, they can resolve the issue by giving the landlord notice to end the tenancy agreement and leave the property (generally 2 months notice). However, this is not always the case and tenants can stay in the property

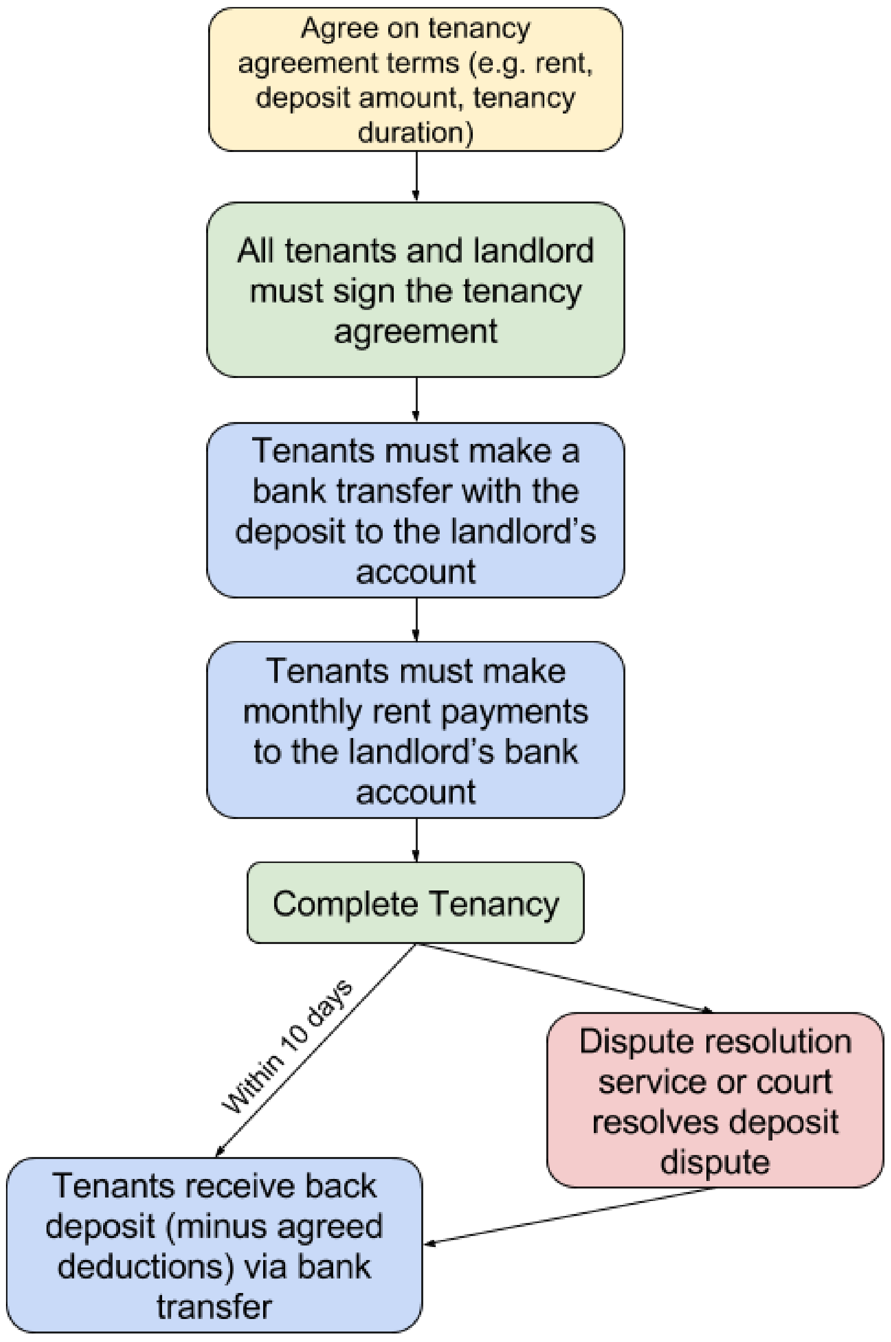

Agree on tenancy agreement terms (e.g. rent, deposit amount, tenancy duration)
All tenants and landlord must sign the tenancy agreement
Tenants must make a bank transfer with the deposit to the landlord's account
Tenants must make monthly rent payments to the landlord's bank account
Complete Tenancy
Within 10 days
Dispute resolution service or court resolves deposit dispute
Tenants receive back deposit (minus agreed deductions) via bank transfer

Figure 2.16: Diagram to show the main stages involved in the standard tenancy agreement process.

without paying rent, leading to them being served an eviction notice from the estate agent. This notice must be hand delivered or sent by recorded delivery to obtain a proof of sending the document (it cannot be sent by email for a number of legal reasons).

There are also several checks that the estate agent must undertake to verify the tenant is suitable for the tenancy and has a good record. They obtain an employer's reference, a bank status inquiry (to ensure they can afford the rent), and a previous landlord reference. These help to ensure the tenant has a good record and will pay the agreed rent on time. The tenant must also have a valid UK/EU passport, or have a valid visa which guarantees their legal right to stay in the country for the duration of the tenancy. I believe these checks would still need to be conducted by the estate agent to work alongside my application, as they have years of experience in this field and I would not likely be able to ensure as accurate of an assessment (especially as a lot of the checks carried out are done with the knowledge of what fraudulent documents look like).

Following this meeting I have obtained the following application requirements:

- The application must be very robust as it will be handling other people's money.
- Application must not show any users information that they are not authorised to see, due to tenancies being private matters.
- The user interface must be simple to use for non-technical users.

THE OLD FIRE STATION 1888 - 1963
HONE
woodward Estate Agents
Reception & Estate Office
woodward Estate Agents
"Without doubt the most professional agents that we have ever dealt with" Mr & Mrs W

Figure 2.17: Woodward Estate Agents in Harrow.

Chapter 3

System Design

This section will look at the requirements that the system aims to fulfil and will discuss the different ways they can be implemented with smart contracts and application logic. In addition, the other architectural aspects of the system such as the contract storage medium and client user interface will be researched, and design decisions justified.

3.1 Tenancy Agreements as Smart Contracts

Smart contracts are perfectly suited to being adapted to all kinds of contractual agreements, particularly those which involve some kind of value transfer where there are multiple, potentially distrusting, parties. As the smart contracts are independent entities which live on the decentralised Ethereum blockchain, they can be publicly inspected, verified, and providing their logic is sound, are completely tamper-proof. Once created, the data they store is immutable, unless explicitly modified by valid transactions sent by contract participants. In the case of tenancy agreements, these transactions can map to operations such as signing the contract or transferring a rental payment. The following subsections will look at these tenancy agreement actions in more detail and will assess the possible design options.

This project will not deal with the phase during which the tenancy agreement terms such as rental amount and tenancy duration are agreed upon, as these are generally known in advance or negotiated in person prior to the contract being created. If it was deemed necessary to add this functionality in the future, the interface would likely resemble that of the Express Agreement project detailed in the background research section [100]. Instead this project will handle the stages during which the tenancy agreement is active, from contract creation until lease completion.

3.1.1 Contract Creation and Signing

The tenancy agreement data will be stored in a smart contract which is stored on the blockchain in order to capitalise on the logic guarantees that smart contract code provides. This means information such as the contract start time and duration need to be stored on-chain in order to, for example, disallow payments being made before the tenancy has started.

The first question to consider is who deploys the smart contract representing the tenancy agreement terms to the blockchain and should this occur before or after the contract has been signed? There are three feasible ways the contract creation and signing process could be handled and are as follows:

1. All parties sign the contract online and once all signatures have been received the contract is deployed to the blockchain by an independent party (such as the web application server).
2. All parties sign the contract online and the last person to sign also undertakes the role of deploying the contract to the blockchain.
3. The landlord deploys the smart contract to the blockchain and all of the tenants then sign the contract via Ethereum transactions digitally signed with their respective private keys.

The first two methods are similar in the fact that the tenancy agreement is signed online. The process would involve the users creating accounts, agreeing to the terms and conditions, and clicking a sign button which would register to the server that the contract has been signed. With the first option the server would have an associated Ethereum account and would deploy the tenancy agreement to the blockchain once the landlord and tenants have signed the contract. This is beneficial because the deployment process is handled automatically so the users do not need to interact with the blockchain. However, after deployment, subsequent actions would either be carried out by the users directly interacting with the blockchain, somewhat nullifying the benefit of not having to perform the initial deployment, or they would also be carried out by the central server on behalf of the contract participants. A major benefit of Ethereum is the decentralisation that it supports when users interact with smart contracts and so having a central server perform these actions is a step backwards when decentralisation is favoured. Furthermore, as the contract participants are relying on the server to transfer all of the contract terms correctly to the smart contract when it is deployed, if the server's security was compromised or there was a logic error, the smart contract terms would not match the initial agreement terms. Therefore I have chosen to reject the first method.

The second option is similar to the first and is also advantageous because the smart contract is not deployed until all parties have signed the agreement. However, by making the person that signs it last have to deploy the contract, it could make the application more confusing for users as their user journey changes depending on the time they sign. Furthermore, if the last user deploys the contract locally from their machine (as opposed to from the server) they would be able to edit the details after it has been signed but before deploying it to the blockchain which is clearly not ideal. This method, like the first, also requires the server to keep track of who has signed the tenancy agreement in a centralised fashion, and so I have decided not to proceed with this implementation either.

The third method eliminates the need for the server to keep track of who has signed the tenancy agreement as the landlord creates and deploys the contract to the Ethereum network immediately. This means the smart contract needs functionality to handle each tenant's signature directly. These signatures translate nicely to Ethereum transactions on the blockchain, as they must be signed with the private key of the sending account. This means all of the users must create Ethereum accounts to perform these contract interactions, which also provides a more secure signing method because only the individual is in charge of storing their account credentials (private key and password). They are never sent to the server but the transactions representing their signatures can still be verified as authentic thanks to public key cryptography (discussed in background research section 2.1.2). This method is ideal because the tenants are signing the contract information stored on the blockchain, which cannot later be altered due to the Ethereum blockchain's immutability.

In addition, the landlord does not need to sign the contract separately as they are in charge of deploying it in the first place, so they should already agree with it and there is no need to send a further signing transaction to reiterate this. A disadvantage with this approach is that the contract is created before the tenants have signed, so if any of them for any reason do not agree with the contract terms that the landlord has laid out, the contract must be adjusted and redeployed. However I believe that this is an acceptable trade-off in return for the guarantee that the contract information agreed upon is immutable.

So to summarise, I will be using the process described in option three for the contract creation and signing aspects of this project. Keeping all contract interactions within the Ethereum network will not only improve security, but also transparency, enabling anyone with a blockchain explorer (such as Etherscan [29]) and knowledge of the contract's location to view its entire transaction history. This would be particularly useful if any legal disputes arise with the tenancy and proof is required in court, for example the exact time an issue is reported (see system design section 3.1.4).

3.1.2 Deposit Payment and Arbitration

The payment of the deposit during a tenancy ensures that the landlord has some funds to compensate them for any unpaid rent or if the property is damaged at the end of the tenancy. As discussed in background research section 2.5, the two types of deposit scheme are insurance backed and custodial. If the landlord holds the deposit in the insurance backed scheme, they pay a fee to insure it, otherwise the deposit is held free of charge by the custodial scheme. By utilising smart contracts, I am looking to reduce the time it takes the tenant to receive back their deposit and to reduce the amount of trust needing to be placed in the system. This section will discuss the possible design decisions for integrating the deposit payment and arbitration processes into smart contract form. Each option below involves the deposit being paid in ether (sent as a transaction over the Ethereum network) with an arbitrator required to decide an appropriate deduction amount in the case of a dispute.

1. The tenant pays the deposit to a trusted arbitrator at the beginning of the tenancy, who pays back the agreed deposit and deductions to the tenant and landlord respectively at the end of the tenancy.

2. The tenant pays the deposit to the smart contract which is storing the logic for the tenancy agreement. The funds are locked in until the end of the tenancy when the landlord can decide the appropriate deduction amount. An arbitrator makes the final decision for the withdrawal split in the case of a dispute.

Paying the deposit in ether, as is the case in both methods, ensures a fast payment which is sent over the blockchain, and therefore not subject to long bank processing times and high fees in the case of international payments. The disadvantage however is due to the current volatility of the ether unit value in GBP, the ether deposit could be worth a very different amount at the end of the tenancy. This doesn't matter if the value of ether goes up, because the landlord will simply have additional GBP deposit funds, but if it goes down the landlord will have less GBP than the initial amount which in extreme cases may not cover the necessary deductions. However, I believe this is acceptable for the deposit as the landlord could charge a higher initial amount to help compensate for this. Looking to the future, the problem will probably be mitigated as crytocurrencies are likely to become more mainstream and could stabilise in price. Furthermore if cryptocurrency exchanges introduce the ability to place put options for ether in GBP (a put option gives the holder the ability to sell an asset at a certain price within a given time frame), the landlord could simply buy a put option to protect against the price of ether falling.

The arbitrator for both methods would be agreed upon at the beginning of the tenancy by both the landlord and tenants, functioning similarly to the dispute resolution service offered by the three deposit protection schemes listed in section 2.5. The contract participants would agree to go with the decision made by the independent arbitrator so the dispute does not need to go to court.

The first method is similar to the custodial type of deposit scheme with the deposit sent to a trusted arbitrator. The payment would be sent back to the parties at the end of the tenancy, and would be arbitrated accordingly in the case of a dispute. The benefit of this is that the arbitrator could immediately convert the received ether to GBP so it is not subject to cryptocurrency volatility. However, it requires trusting the arbitrator with the funds and requires action from them to transfer the ether back to the tenant even if there are no deposit deductions made by the landlord.

The second option stores the ether in the balance of the smart contract which can be encoded to lock the funds until the end of the contract. This eliminates the need for a trusted party to store the funds as they cannot be accessed or spent by anyone until the contract duration is complete. A further advantage with this method is that the arbitrator does not need to do anything if there is no dispute, as the tenant can simply accept the landlord's specified deductions and the funds will be unlocked to be withdrawn immediately by the respective parties. However the arbitrator still needs to resolve any disputes that arise and must specify the final amount to be deducted from the deposit, but there is no way around this requirement.

Therefore I will proceed with the second option, as the fewer components that depend on a trusted party, the more secure the application will be. It should also speed up the time for the deposit to be returned and the fees involved will only be the small transactions fees necessary to interact with the Ethereum network.

3.1.3 Rent Payments

In addition to the deposit being paid in ether, using smart contracts facilitates the rent instalments to also be paid to the landlord in ether. The smart contract can then enforce logic on these payments, such as preventing the rent from being overpaid or transferred after the tenancy is complete.

Manual vs Automatic Payments

An important decision to make regarding these payments is whether the rent should be paid manually or automatically before the due date.

- Manual payments require the tenant to manually enter their Ethereum account password to approve and trigger each transaction.
- Automatic payments would involve the user granting a program access to their account password in order to transfer rent on their behalf.

While it would be desirable for the tenant to have payments triggered automatically, this is also less secure and the account password should ideally not be entrusted with a third party. A program always has the potential to have subtle programming errors or become compromised by a malicious actor or malware, risking the disclosure of the user's password. As the tenant's Ethereum account is effectively their bank account, I believe it is necessary to have more stringent security at the cost of additional user interaction being required. Furthermore, if the user has to make the payments themselves, the scenario of the account having insufficient funds prior to making a payment is no longer an issue. The tenant can simply purchase more ether instantly with GBP (possible on Coinbase [9]), something that would not be possible with an automated service. Hence for these reasons I have opted for the manual payments design.

Fixed Ether vs Fixed GBP Weekly Rent

My initial plan (and smart contract implementation) was for the landlord to specify a fixed amount of rent in ether per week for the duration of a tenancy, for example a contract specifying 10 ether due per week for a total of 52 weeks. This worked well in January 2017 when the price of ether was stable at roughly £8, equating to around £80 due each week to be paid in ether. However since then, the price of ether has become extremely volatile and rose to £180 at the end of May 2017 (an increase of 2150%) [87]. This means a tenant paying 10 ether per week as described above would have gone from paying the equivalent of £80 per week in January to £1800 per week by the end of May. Clearly this is no longer an acceptable design regardless of whether the tenant had purchased the total amount of ether up front at the lower price, as this would never be expected to be the case. The reverse could also occur where the price of ether plummets, and so with a fixed amount being paid each week, the landlord would be very much worse off.

As the rent must be paid in ether to be transferred and recorded on the Ethereum blockchain, there is one solution to this problem, and that is to use an oracle service to access the current ether to GBP exchange rate (as access to the web is not possible directly from a smart contract). This works by querying the oracle for the price of ether in GBP on each rent payment and requiring the ether paid each week to be equivalent to a fixed amount in GBP, such as £80 per week for 52 weeks. This means the amount of ether paid would be variable depending on its current price, therefore the tenant would pay exactly £80 per week, which would have equated to approximately 10 ether in January and around 0.4 ether at the end of May 2017. This is a much more robust solution due to the generally much lower volatility of GBP. Tenants will also prefer the familiarity of paying an equivalent value in GBP each month, with the goal being to make this project as simple to understand for non-technical users as possible.

Oraclize

I have decided to use the oracle service offered by Oraclize [65] in this project as it is the most mature oracle service around for Ethereum and features a range of security measures to guarantee that the result delivered to the blockchain by the oracle has not been tampered with (including a TLSNotary proof [67]). An alternative oracle service I considered using is the Town Crier project [85], but it is much earlier on in development and doesn't offer any big advantages over Oraclize.

The tenancy agreement smart contract can obtain the ether to GBP exchange rate from a price feed web API (such as the one provided by CryptoCompare [11]) by querying the Oraclize smart contract deployed to the Ethereum network, with the API results being returned via a callback function (see Figure 3.1 for a diagram of the Oraclize architecture [66]). The implementation details of Oraclize in the tenancy agreement smart contract code will be discussed in implementation section 5.2.2.

There are however some disadvantages with using an oracle service. It requires trust in the centralised Oraclize service, so if the Oraclize infrastructure outside of the Ethereum blockchain went offline, the price feed would become inaccessible. In addition it also requires the price feed web API to remain accessible, in order to obtain a value for the price of ether in GBP. This trust could be minimised by querying a second oracle service in the tenancy agreement smart contract and using a different price feed API to reduce the risk of not receiving a response. However, this would also increase the transaction fee when a tenant pays rent due to the larger

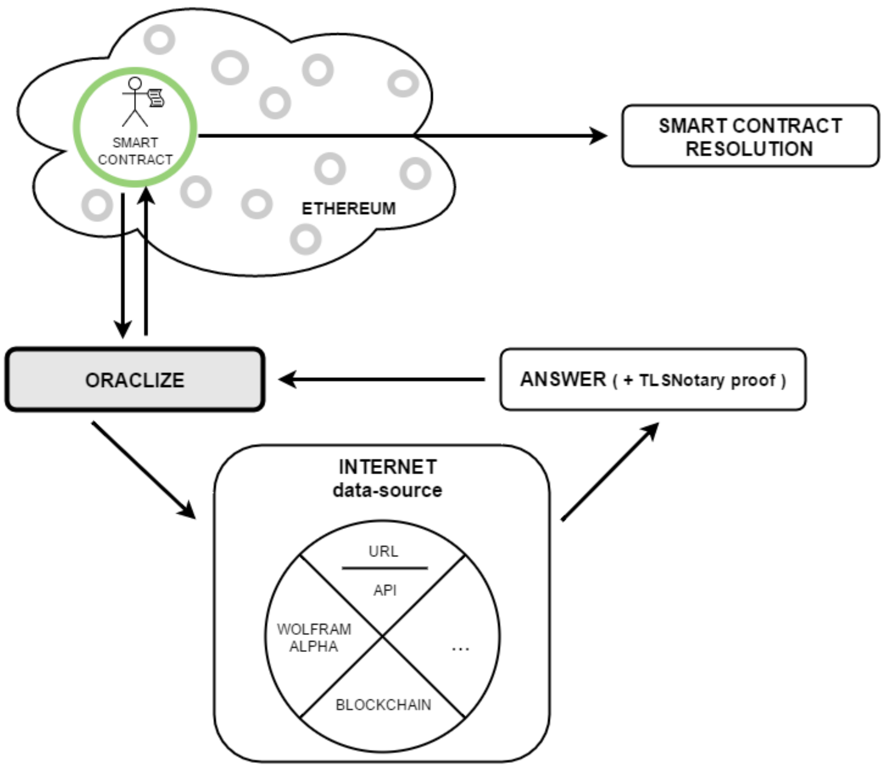


Figure 3.1: Oraclize query flowchart from the Oraclize documentation [66].

amount of smart contract computation required. So for this project I will only be using Oraclize but the above suggestion could be implemented in future work.

3.1.4 Issue Reporting and Resolution

The Ethereum blockchain doesn't just support signed transactions and payments of ether, but also allows for arbitrary data to be stored inside the smart contract. This allows for issues with the tenancy to be recorded on the blockchain, together with the account that reported the issue and corresponding timestamp. These issues can be anything from the tenant reporting that the boiler has broken to alerting the landlord of a mouse infestation. Recording these on the blockchain will enable seamless integration with the front-end web application to alert the landlord of these issues, as well as to provide irrevocable proof of the time that an issue was reported. Although reporting these issues will incur transaction fees on the blockchain, I think it makes sense to keep all of the pieces of information relating to the tenancy agreement in one place and stored together. I will discuss the issues surrounding storing sensitive data on the blockchain in section 3.1.6.

Once an issue has been reported, a mechanism for issue resolution is needed to determine when the issue can be declared resolved. My first approach for this was to just allow either the landlord or tenant to mark an issue as being resolved once the issue has been fixed. However, the problem with this is that the landlord could mark the issue as resolved when it may not be completely resolved from the tenant's point of view. There is no way around this he-said-she-said scenario without an impartial arbitrator, and so the arbitrator agreed upon at the beginning of the tenancy to deal with deposit arbitration should also be able to rule in favour of the landlord or tenant in the case of issue dispute. By allowing the arbitrator to settle these conflicts, the aim is to reduce the need for cases like these needing to be settled in court. In the case of court still being required, for example if the arbitrator is unsure who to side with, the presence of all the tenancy issue interactions on the blockchain would act as tamper-proof evidence which could be used in court.

3.1.5 Notices

The issue system mentioned in the previous section is important to enable tenants to reports issues that need to be fixed by the landlord. However, they may also need to bring information to the landlords attention that does not require any action to be taken, for example they could be going on holiday for an extended period and want to alert the landlord that the property will be empty during a given time period. Moreover, the landlord may wish to send notices to the tenants alerting them of upcoming maintenance, or if the tenancy is nearing completion, new tenant viewings of the property may be scheduled.

These notices will have a similar design to issues, but instead of having to be resolved, they need to be acknowledged by the recipient. The notices will also be stored on the blockchain to provide an immutable record of information sent between the contract participants.

3.1.6 Information Stored on the Blockchain

When designing the smart contract, careful thought must be given to the information that is stored on the Ethereum blockchain as it is a public ledger which anyone can inspect. As tenancy agreements are confidential documents, information such as the names of the tenants, the address of the property, and the contract clauses are not suitable to be stored as plaintext strings in the smart contract. Tenants would not want their Ethereum accounts linked with their name because this is effectively revealing to the world the size of their bank balance. This leaves two options:

1. Store the confidential information on the blockchain but encrypt it first.
2. Do not store any confidential information in the smart contract, instead store it off-chain in another form of storage.

The first option solves the problem of preventing the confidential information from being publicly viewable, however it creates another problem of preventing the relevant parties (landlord, tenants, and arbitrator) from being able to view the contract details without a decryption key. Key management would therefore be required between these authorised parties which would need a central server. Furthermore, storing these large encrypted strings inside the smart contract would incur a high gas cost for their storage (operation gas costs are detailed in the Ethereum Yellow Paper [99]). These values would also be of no use to the smart contract and are simply there so the holders of the keys can verify who the contract participants are and view the contract clauses agreed upon.

Therefore a better solution is the second option where this information is stored on a server outside of the Ethereum network where the contract participants must first authenticate to access (discussed further in section 3.2). Unfortunately this means the contract clauses are stored on a centralised server and so there is potential they could be tampered with. To avoid this, a solution is to calculate a hash of the clauses using a hash function and instead store this value in the smart contract. As discussed in background research section 2.1.1, hash functions provide a way to generate a deterministic fixed length digest for a piece of data, which combined with the original data, can act as an integrity check to detect if the data has been modified since the hash value was generated. This technique is utilised by the website "Proof of Existence" [70], discussed in background research section 2.4.4. The Express Agreement project discussed in background research section 2.4.1 also uses this mechanism of storing a hash value to prove a set of contract clauses were agreed upon. Separate hashes are not required for the contract participant names and the property address because this information can also be detailed in the contract clauses, so just a single hash value of the contract clauses is necessary.

A range of additional, non-personally identifying information must also be stored in the smart contract so it can enforce logic on the tenancy agreement. Below I will list the main values and explain for each one why it is required. Note that the smart contract language I will be using, Solidity, does not currently support storing decimal numbers and so all fields that store ether record 'wei', and fields recording GBP store values in pence.

- **Current contract stage**: Records the current state that the tenancy agreement is in, such as unsigned, signed, active and completed. This ensures certain actions can only be executed when the contract is in a given state, e.g. a tenant can only sign the contract when it is in the 'unsigned' stage.

- **Contract clauses hash**: SHA-256 hash of the tenancy agreement contract clauses (as described in the previous section).

- **Landlord, tenant, and arbitrator Ethereum addresses**: Store the addresses belonging to each contract participant so custom logic can be executed depending on who sent a transaction to the smart contract.

- **Tenants that have signed the contract**: Records which tenants have signed the contract so the contract moves to the 'signed' stage once all tenants have signed.

- **Start time**: The timestamp that the tenancy period begins; rent cannot be paid until the tenancy period has begun.
- **Contract duration**: The tenancy duration in weeks so the contract knows when the tenancy is complete (by adding the total number of seconds in the contract to the start timestamp).
- **Deposit paid / total deposit**: The amount of ether the tenants have paid to the contract and the total deposit amount required respectively. Once the total deposit amount has been paid, the smart contract will begin accepting rent to be paid.
- **Deposit deductions**: Set by the landlord at the end of the tenancy to specify the amount of ether to be deducted (if any) to pay for any damage caused by the tenants. This amount must be accepted by the tenant or arbitrator before the deposit can be withdrawn, which the smart contract logic will enforce.
- **Rent paid in wei**: The amount of rent paid by the tenant in wei (1 ether equates to 10^{18} wei).
- **Rent per week in pence**: Stores the amount of rent due to be paid by the tenant each week in pence.
- **Rent paid / total rent amount due in pence**: The amount of rent paid by the tenant in pence and the total amount of rent due for the duration of the tenancy in pence.
- **Issues reported**: All issues reported to the smart contract need to be stored. The fields recorded for each issue include sender, timestamp reported, issue ID, hash value of issue text, issue severity, and the timestamp of resolution.
- **Notices sent**: All details of notices sent to the smart contract must be stored, including sender, recipient, timestamp sent, notice ID, hash value of notice text, and the timestamp of acknowledgement.

Details of how this data is stored in the smart tenancy agreement and its Solidity code will be covered in implementation section 5.2.2.

3.2 Tenancy Agreement Storage

As discussed in the previous section, the sensitive tenancy agreement information such as the contract clauses must be kept private and so won't be stored on the public Ethereum blockchain. This means they must be stored on either centralised or decentralised storage. This section will briefly discuss the relative merits and disadvantages of each approach.

3.2.1 Centralised Server vs Decentralised Storage

Centralised storage methods typically involve a server which manages access control to the private data, only allowing access for authenticated parties (such as via a session cookie). A centralised server could receive requests to upload new tenancy agreements, or update user details. This provides a simple and fairly secure way to store the confidential data, and is how the vast majority of websites today handle private user data. However, a centralised service introduces the following problems:

1. The server must remain online 24/7 in order for users to have access to their tenancy agreements.
2. A centralised server can potentially be compromised by a malicious actor, with tenancy agreements being modified or erased.
3. It could be costly to host this program on multiple server instances, especially if the number of users increases significantly.

However, as I will be storing the hash of the tenancy agreement clauses on the blockchain, an alteration of the contract clauses stored on the server would be detected as its hash value would not match. Therefore as long as the contract participants save a copy of the clauses offline, the second issue is mitigated.

Decentralised storage on the other hand would mean storing the contract clauses encrypted on many nodes being accessed in a peer-to-peer fashion (encryption is needed because the data is publicly accessible). While this solves issues 1 and 3 above, it makes it much harder to handle the necessary key management required to ensure all contract participants can access the encrypted data. A centralised key management server would be required to access the decentralised storage thereby offsetting the benefit of using it in the first place, because if the server in charge of providing keys and the locations of the encrypted data went down, the tenancy agreements would be rendered unreadable.

Therefore for these reasons I will stick with a centralised server to store the tenancy agreement clauses, but will continue to investigate possible ways to move this to a more decentralised approach in the future. I will still use decentralised storage to host the client-only front-end of the web application, which will make requests to the server to download the necessary contract clauses.

3.3 Web Application User Interface

During the entirety of the user interface development phase I will keep several key requirements at the forefront of the decision process. I have discussed these with a range of potential users and have summarised the most important points which are as follows:

- Ensure the user journey throughout the application is simple and that the users can complete the task at hand with ease and minimal button clicks.
- Keep the instructions clear and minimise the amount of technical jargon presented to the user, as they may not be very familiar with cryptocurrency terminology.
- The user should know the state of the application at all times, they should never be left unsure whether their request has been successful or have to refresh the page for updates.

To evaluate whether the application fulfils these front-end requirements, I will conduct a user survey at the end of the project to ascertain whether a sample of users agree that the above points have been achieved (see section 6.2).

3.3.1 User Journey

The user interface of the web application should enable the user to seamlessly deploy transactions into the Ethereum network, such as the signing of tenancy agreements or payment of rent in ether (as described in section 3.1). The web interface should make it simple for the user to perform these actions, with it being clear which task they need to perform next. At times it may be necessary to display some technical details to the user, such as their Ethereum account address and balance, but I am assuming the user has a basic knowledge about Ethereum and knows how to create an account and purchase ether. No actions on the application can be performed without an Ethereum account funded with some ether anyway, so I am treating this as a hard prerequisite.

A design that could work well to ensure the user always knows what task to perform next is to have an overview screen for each tenancy agreement that shows any action currently required. I have validated this with several potential users, who agreed that using a traffic light system to show an at a glance summary of the current tenancy agreement's standing would work well.

The web application should also update without requiring any page refreshes, ensuring the user always sees the most up to date information.

3.3.2 Name and Logo

To give the application an identity it requires a name and logo, so users can refer to it succinctly and recognise it easily. It should be eye-catching and aim to integrate a triangular aspect to echo its use of Ethereum.

I have chosen the name *Acropolis* which literally means 'high city'. It was the place where contracts were formed in ancient times, with the records of the contracts inscribed on markers at the acropolis in Athens [97].

The logo was created using Boxy SVG [8], with the ‘A’ designed manually and the ‘Abel’ font used for the rest of the name. Figure 3.2 shows the main logo and Figure 3.3 pictures the favicon.

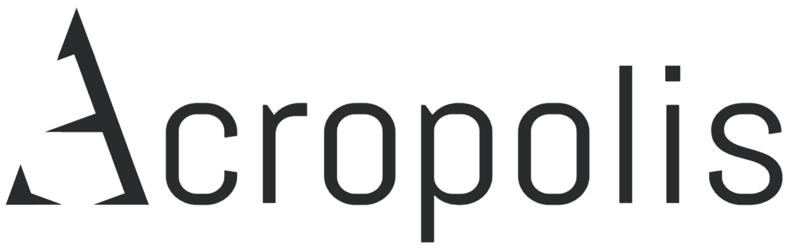


Figure 3.2: Application main logo.

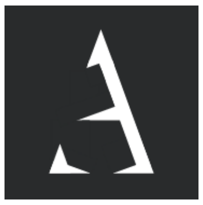


Figure 3.3: Application favicon.

Chapter 4

The Application: Acropolis

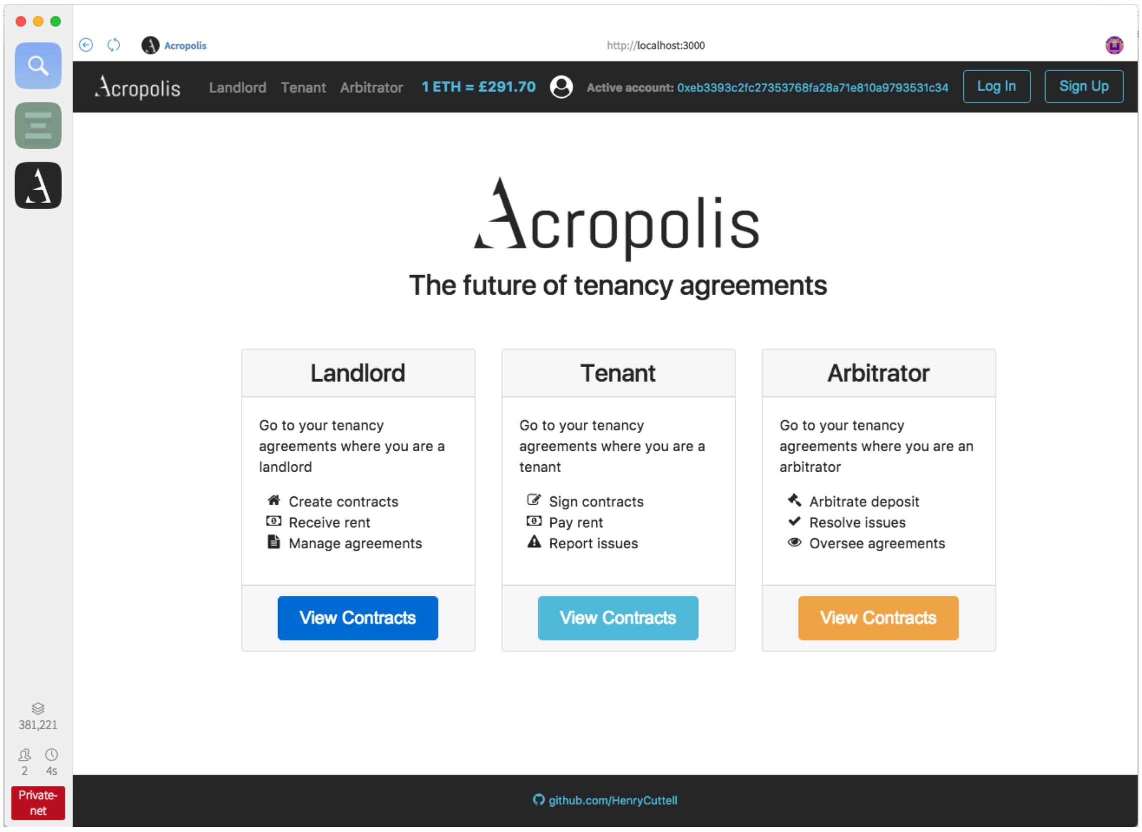
4.1 User Journeys

This section will explain the roles of the different users of Acropolis and will include screenshots to illustrate the steps they can take through the application. The three types of user are as follows:

- **Landlord**: The landlord is in charge of creating the tenancy agreements and overseeing all stages of the contract. They can withdraw the rent paid to them, confirm and resolve issues, as well as send notices to the tenants or arbitrator. The landlord is also in charge of specifying the amount of the deposit to be deducted at the end of the tenancy which must then be approved by the tenant.
- **Tenant**: The tenant is able to sign tenancy agreements, pay the deposit, and transfer rent via the application. In addition they can report issues that need attention to the landlord, as well as send notices to the landlord or arbitrator for things that do not need resolving directly.
- **Arbitrator**: It is the role of the arbitrator to resolve any issues that arise between the landlord and tenants. They are also in charge of resolving deposit disputes and will make the final decision on the amount of the deposit to be deducted at the end of the tenancy, should the tenant reject the landlord's suggested deductions. The arbitrator can also oversee all stages of the contract so they are informed about the history of the tenancy if any disputes arise.

The homepage of the web application within the Mist browser is shown in Figure 4.1, which is the page first presented to the user where they can choose a role to view contracts for. It gives them a brief description of the website and tells them the roles of the three different types of users. Each user may have more than one different role as it is possible for a landlord to be renting a property themself, or an arbitrator to be the landlord of another property. Users can then sign up to create an account shown in Figure 4.2, followed by using the log in modal as shown in Figure 4.3 to log in.

The following subsections will explain in more detail the user journey for each role.


Acropolis
http://localhost:3000
Acropolis Landlord Tenant Arbitrator 1 ETH = £291.70 Active account: 0xeb3393c2fc27353768fa28a71e810a9793531c34 Log In Sign Up
381,221
2 4s
Private-net
github.com/HenryCuttell

Acropolis

The future of tenancy agreements

Landlord

Go to your tenancy agreements where you are a landlord

- Create contracts
- Receive rent
- Manage agreements

View Contracts

Tenant

Go to your tenancy agreements where you are a tenant

- Sign contracts
- Pay rent
- Report issues

View Contracts

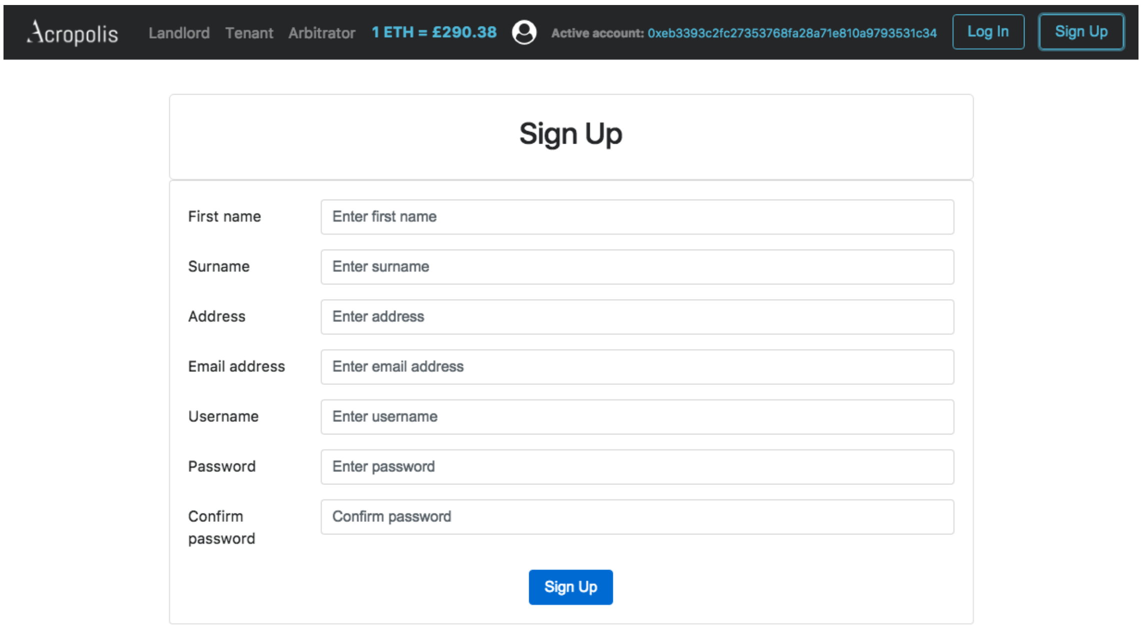
Arbitrator

Go to your tenancy agreements where you are an arbitrator

- Arbitrate deposit
- Resolve issues
- Oversee agreements

View Contracts

Figure 4.1: Home page displayed within the Mist browser

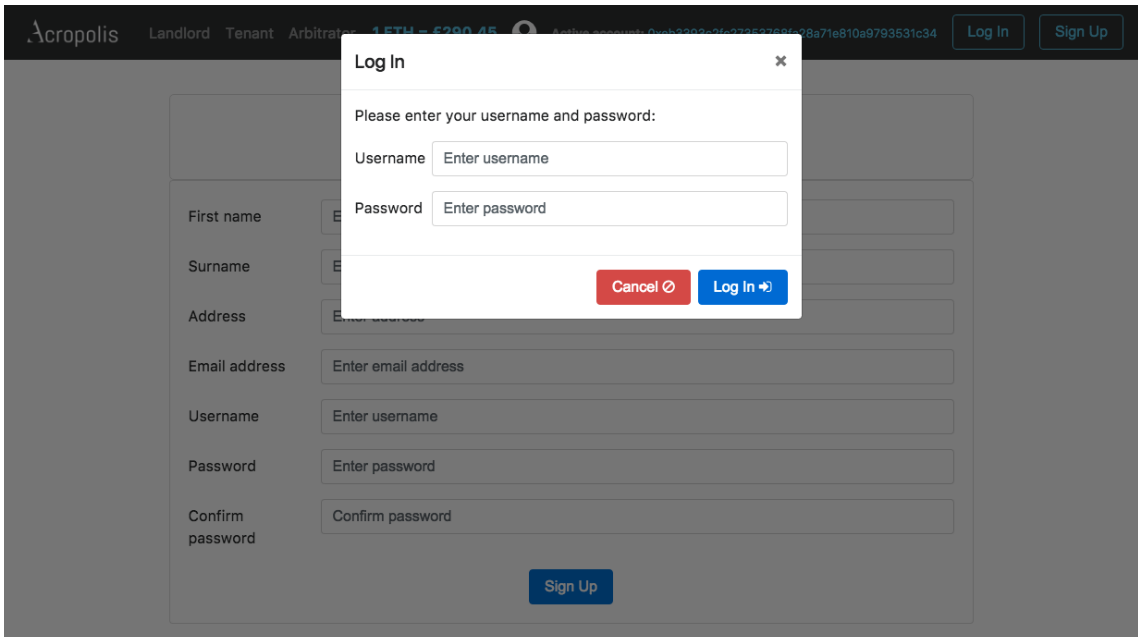

Acropolis Landlord Tenant Arbitrator 1 ETH = £290.38 Active account: 0xeb3393c2fc27353768fa28a71e810a9793531c34 Log In Sign Up

Sign Up

First name	Enter first name
Surname	Enter surname
Address	Enter address
Email address	Enter email address
Username	Enter username
Password	Enter password
Confirm password	Confirm password

Sign Up

Figure 4.2: Sign up page



Log In

Please enter your username and password:

Username	Enter username
Password	Enter password

First name	E
Surname	E
Address	E
Email address	Enter email address
Username	Enter username
Password	Enter password
Confirm password	Confirm password

Figure 4.3: Log in modal

4.1.1 Landlord

One of the main goals for the application is to make it easy for landlords to see overview statistics for their portfolio of properties. As shown in Figure 4.4, the landlord can clearly see at the top of the page how many contracts they currently have, and how much rent they have available to withdraw, which is displayed in GBP but corresponds to ether paid by the tenant (the ether conversion is shown in a tooltip if the user hovers over the GBP value shown in Figure 4.5). The dashboard also shows the total number of unresolved issues aggregated over all of their properties, as well as the total number of new notices sent to them.

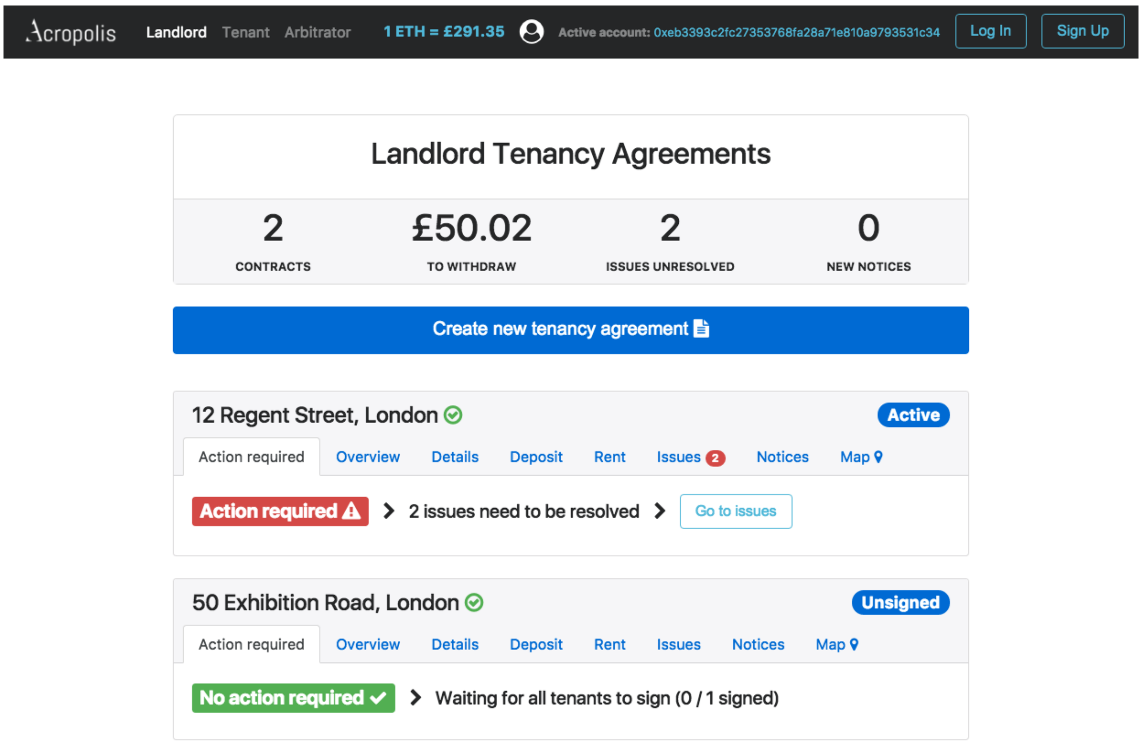
Tenancy agreements are displayed on the page one below the other, with the default tab for each being ‘action required’, which shows a traffic light indicator of the contract’s current standing. The green badge means there is no action required, whereas orange and red badges indicate that there are items requiring attention. This is a key area of the user interface design and clearly gives the user an at a glance way of checking the state of their contracts. The tab system for each tenancy agreement ensures a minimal amount of information is presented to the user at any one time, to try and make navigating the application as simple as possible, and avoid overloading the landlord with too much information.

The landlord can also create new contracts from this page by clicking the large blue button which reveals a dropdown input box where they can input details of a new tenancy agreement (see Figure 4.6). This allows them to specify the tenants in the agreement, with multiple input sections supported by clicking the green ‘plus’ button. The landlord can then enter the arbitrator details, as well as the weekly rent to be paid in GBP, the total deposit amount, and finally the tenancy agreement clauses which should include everything in the current paper-based tenancy agreements. The agreement should also include the necessary clauses for using this system and the legal specifics for transferring the rent in ether (such as payments cannot be reversed once sent). When the landlord submits the contract, it will be deployed to the blockchain and, upon successful deployment, added to the back-end database storing the tenancy agreement contract clauses, as well as confidential information such as the names of tenants and the property address.

As a tenancy agreement transitions through its various stages as actions are completed (for example from 'unsigned' to 'signed'), the 'action required' tab the landlord will see for each contract will change. After contract deployment to the blockchain there is no immediate action required by the landlord; they must wait for the tenant to sign the contract (Figure 4.7), pay the deposit (Figure 4.8), and transfer rent payments throughout the tenancy (Figure 4.9). However, the landlord must take action if the tenant reports issues with the property, and as shown in Figures 4.10 and 4.11, the 'action required' tab is updated to alert them of this. A notification showing the number of unresolved issues for a tenancy agreement is also displayed next to the 'issues' tab and can be seen in Figure 4.4. The landlord can confirm the issue within the 'issues' tab to let the tenant know they acknowledge the issue is a problem (Figure 4.13). They can also resolve the issue once they have dealt with it, though this doesn't remove the issue until either the tenant or the arbitrator also says the issue has been resolved, ensuring issues cannot be disregarded by the landlord. A mechanism to ensure the landlord must resolve issues truthfully is the fact that the deposit funds cannot be released at the end of the tenancy if there are issues which remain unresolved.

Other actions the landlord can perform include withdrawing rent from the 'rent' tab (Figure 4.12), and sending notices to the tenant or arbitrator via the 'notices' tab (Figure 4.15). Figure 4.14 shows how a landlord can create a notice to send to the tenants to inform them that there is maintenance scheduled for a given date.

At the end of the tenancy the landlord is able to specify the deposit deductions which are to be deducted from the initial deposit paid by the tenant (see Figure 4.16). This action transitions the tenancy agreement to the 'completed' stage and alerts the tenant that they have deposit deductions to approve/reject. In the case when the landlord does not deduct any funds from the deposit (specifying a deduction value of zero), the tenant can immediately withdraw the entire deposit from the smart contract.



Acropolis Landlord Tenant Arbitrator 1 ETH = £291.35 Active account: 0xeb3393c2fc27353768fa28a71e810a9793531c34 Log In Sign Up

Landlord Tenancy Agreements

2	£50.02	2	0
CONTRACTS	TO WITHDRAW	ISSUES UNRESOLVED	NEW NOTICES

Create new tenancy agreement

12 Regent Street, London — Active

Action required | Overview | Details | Deposit | Rent | Issues 2 | Notices | Map

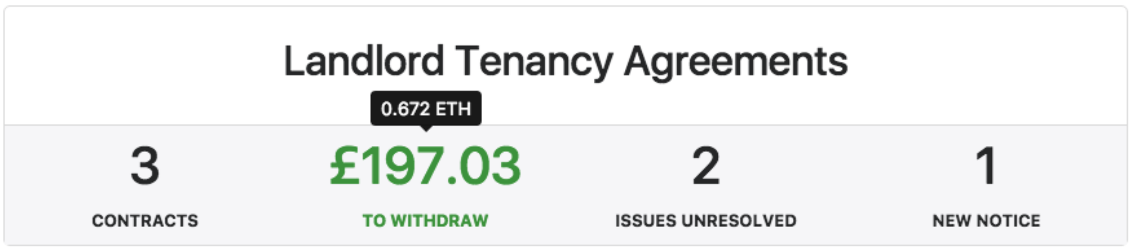
Action required > 2 issues need to be resolved > Go to issues

50 Exhibition Road, London — Unsigned

Action required | Overview | Details | Deposit | Rent | Issues | Notices | Map

No action required ✓ > Waiting for all tenants to sign (0 / 1 signed)

Figure 4.4: Landlord home screen

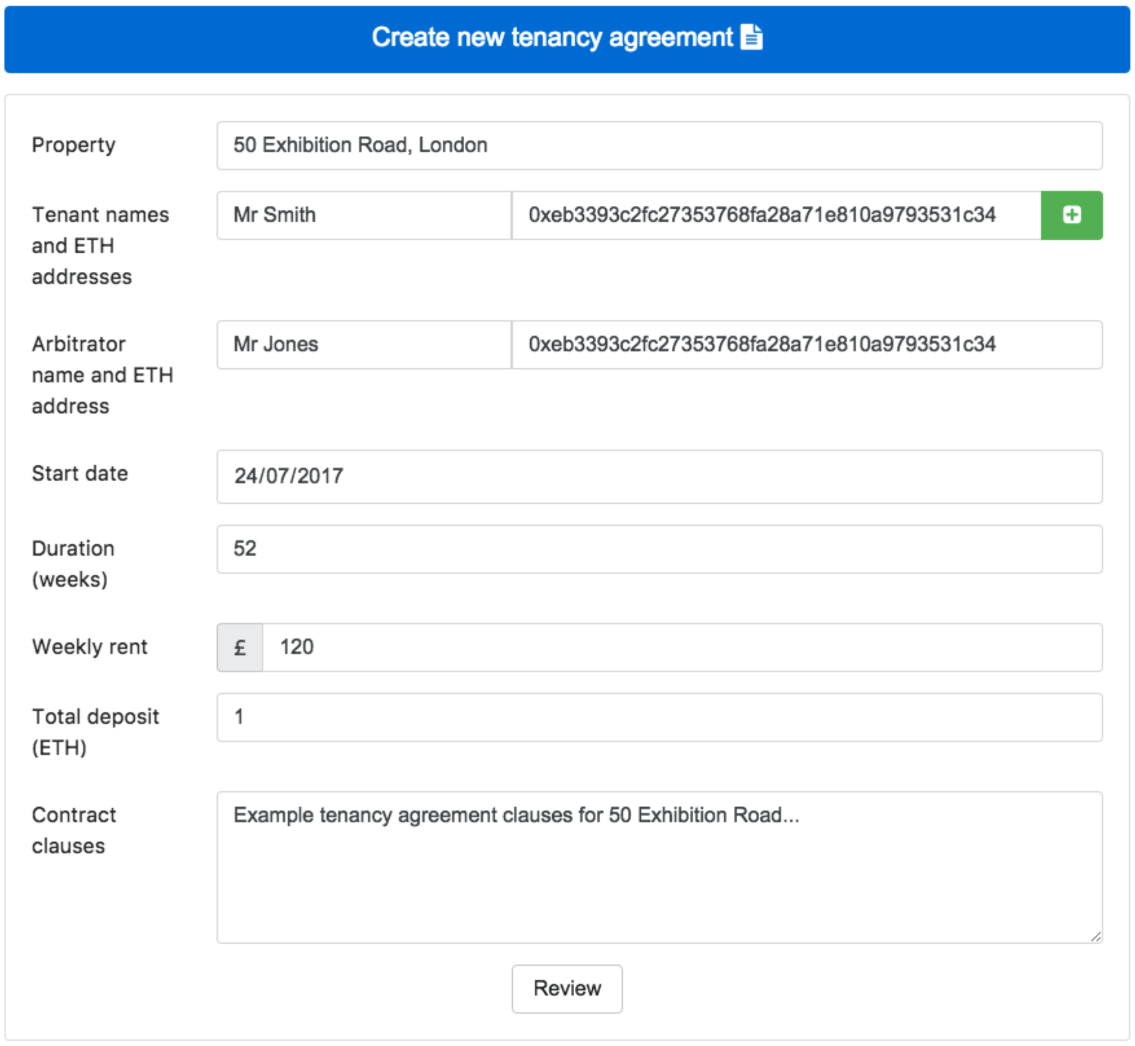


Landlord Tenancy Agreements

0.672 ETH

3	£197.03	2	1
CONTRACTS	TO WITHDRAW	ISSUES UNRESOLVED	NEW NOTICE

Figure 4.5: Landlord statistics panel - GBP to ether conversion tooltip

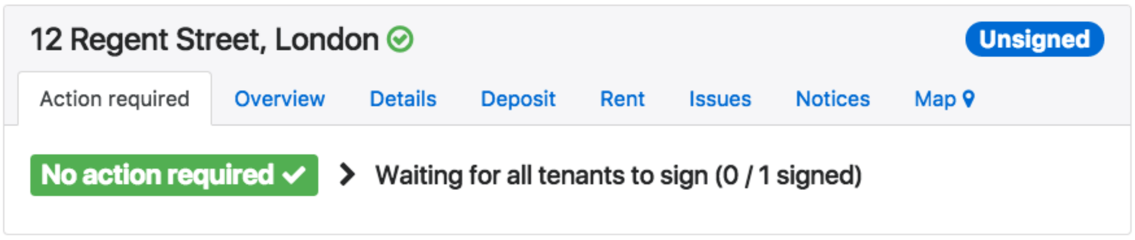


Create new tenancy agreement

Property	50 Exhibition Road, London	
Tenant names and ETH addresses	Mr Smith	0xeb3393c2fc27353768fa28a71e810a9793531c34
Arbitrator name and ETH address	Mr Jones	0xeb3393c2fc27353768fa28a71e810a9793531c34
Start date	24/07/2017	
Duration (weeks)	52	
Weekly rent	£ 120	
Total deposit (ETH)	1	
Contract clauses	Example tenancy agreement clauses for 50 Exhibition Road...	

Review

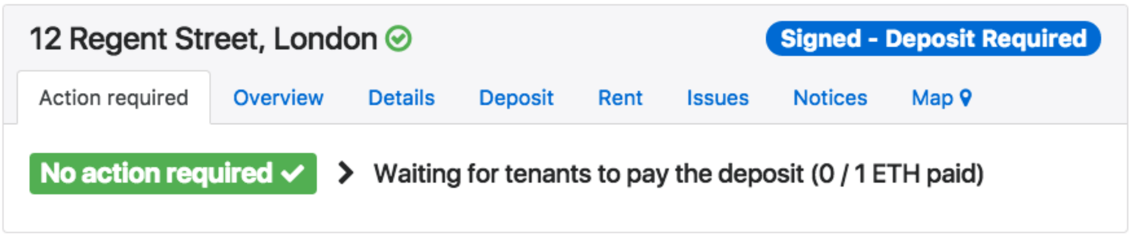
Figure 4.6: Create new tenancy agreement dropdown, with example contract details filled in



12 Regent Street, London ✓ Unsigned

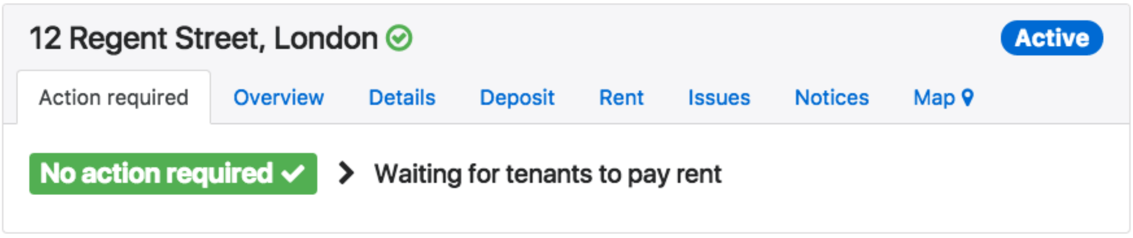
No action required ✓ > Waiting for all tenants to sign (0 / 1 signed)

Figure 4.7: Landlord action required tab - waiting for tenants to sign contract



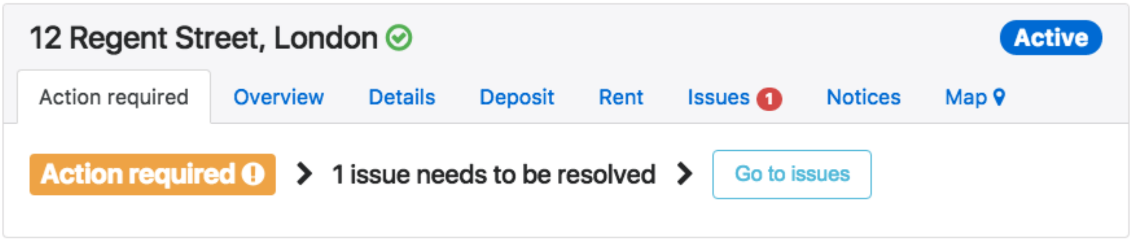
Waiting for tenants to pay the deposit (0 / 1 ETH paid)

Figure 4.8: Landlord action required tab - waiting for tenants to pay deposit



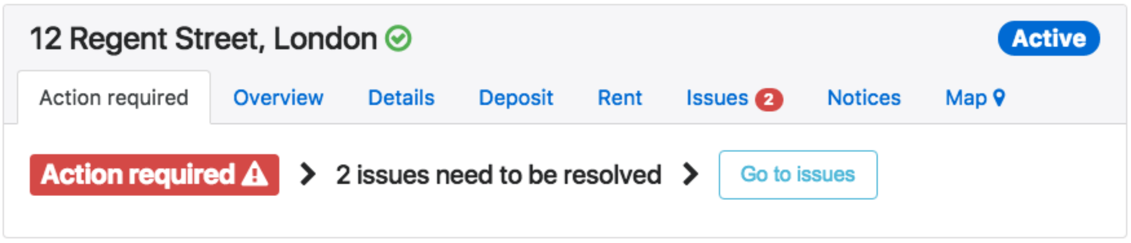
Waiting for tenants to pay rent

Figure 4.9: Landlord action required tab - waiting for tenants to pay rent



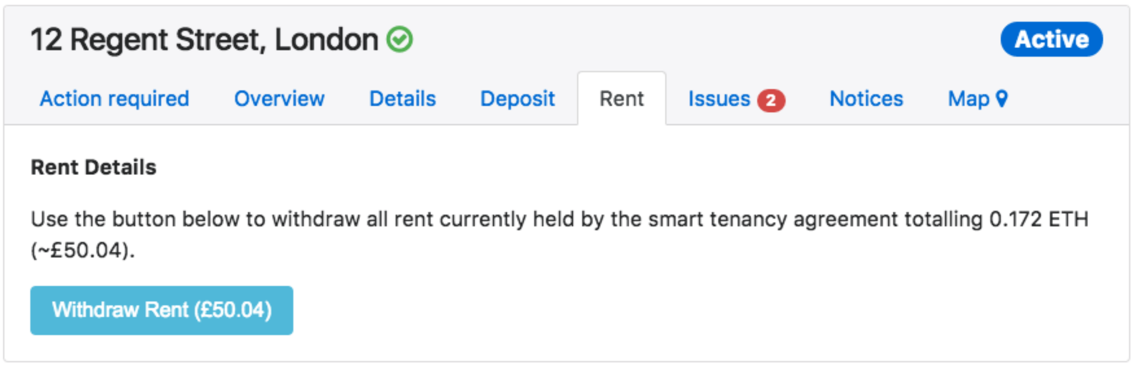
1 issue needs to be resolved

Figure 4.10: Landlord action required tab - 1 issue needs resolving



2 issues need to be resolved

Figure 4.11: Landlord action required tab - 2 issues need resolving



12 Regent Street, London

Active

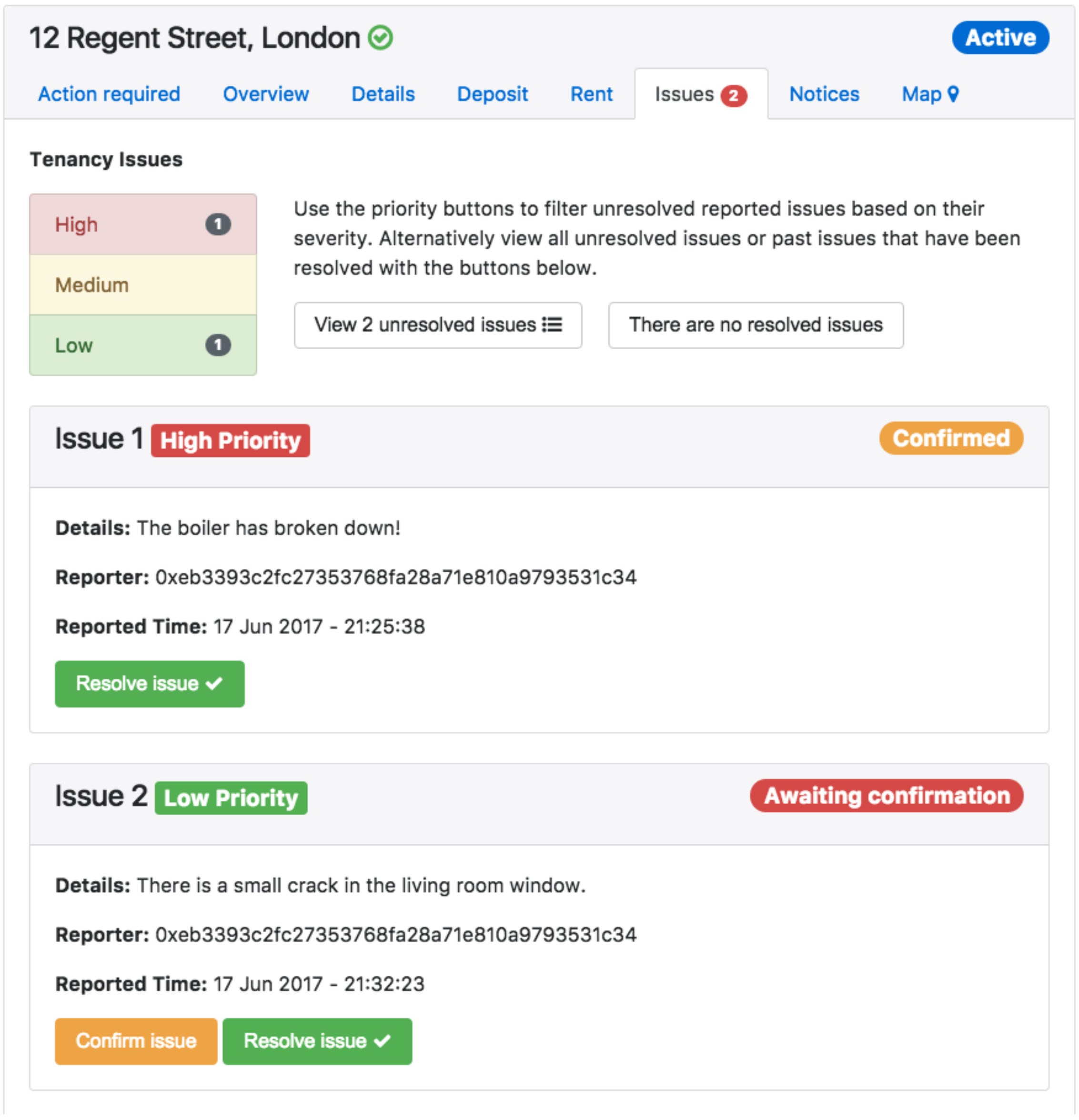
Action required | Overview | Details | Deposit | Rent | Issues 2 | Notices | Map

Rent Details

Use the button below to withdraw all rent currently held by the smart tenancy agreement totalling 0.172 ETH (~£50.04).

Withdraw Rent (£50.04)

Figure 4.12: Landlord rent tab - rent needs withdrawing



12 Regent Street, London

Active

Action required | Overview | Details | Deposit | Rent | Issues 2 | Notices | Map

Tenancy Issues

High	1
Medium	
Low	1

Use the priority buttons to filter unresolved reported issues based on their severity. Alternatively view all unresolved issues or past issues that have been resolved with the buttons below.

View 2 unresolved issues | There are no resolved issues

Issue 1 High Priority — Confirmed

Details: The boiler has broken down!

Reporter: 0xeb3393c2fc27353768fa28a71e810a9793531c34

Reported Time: 17 Jun 2017 - 21:25:38

Resolve issue

Issue 2 Low Priority — Awaiting confirmation

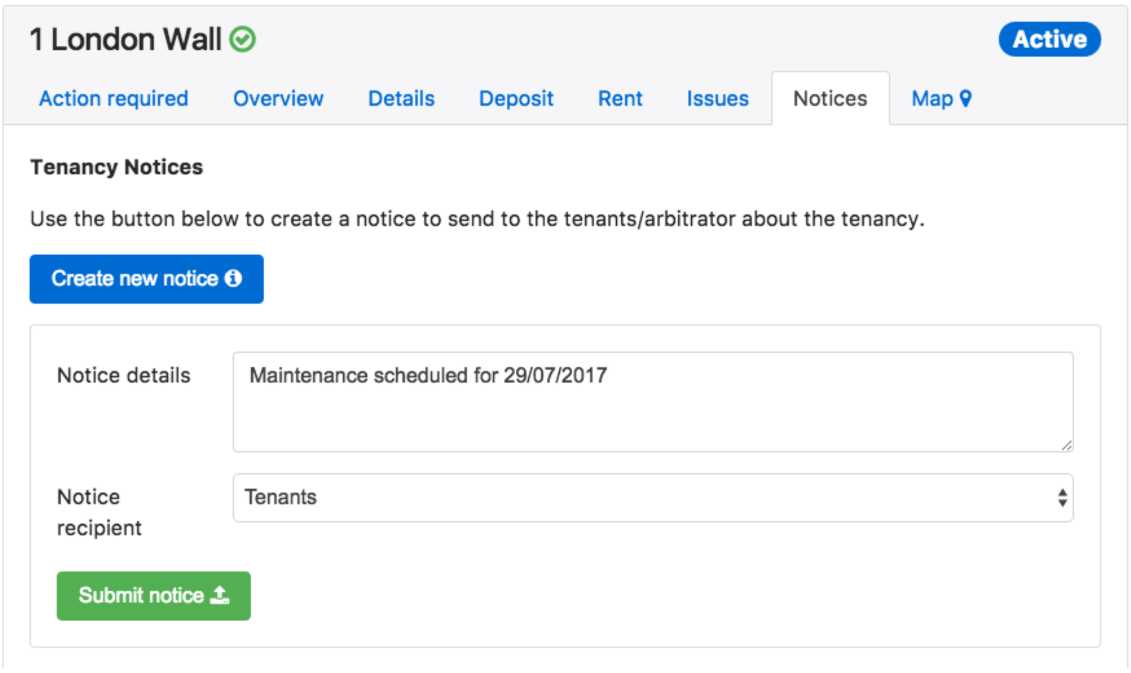
Details: There is a small crack in the living room window.

Reporter: 0xeb3393c2fc27353768fa28a71e810a9793531c34

Reported Time: 17 Jun 2017 - 21:32:23

Confirm issue | Resolve issue

Figure 4.13: Landlord issues tab



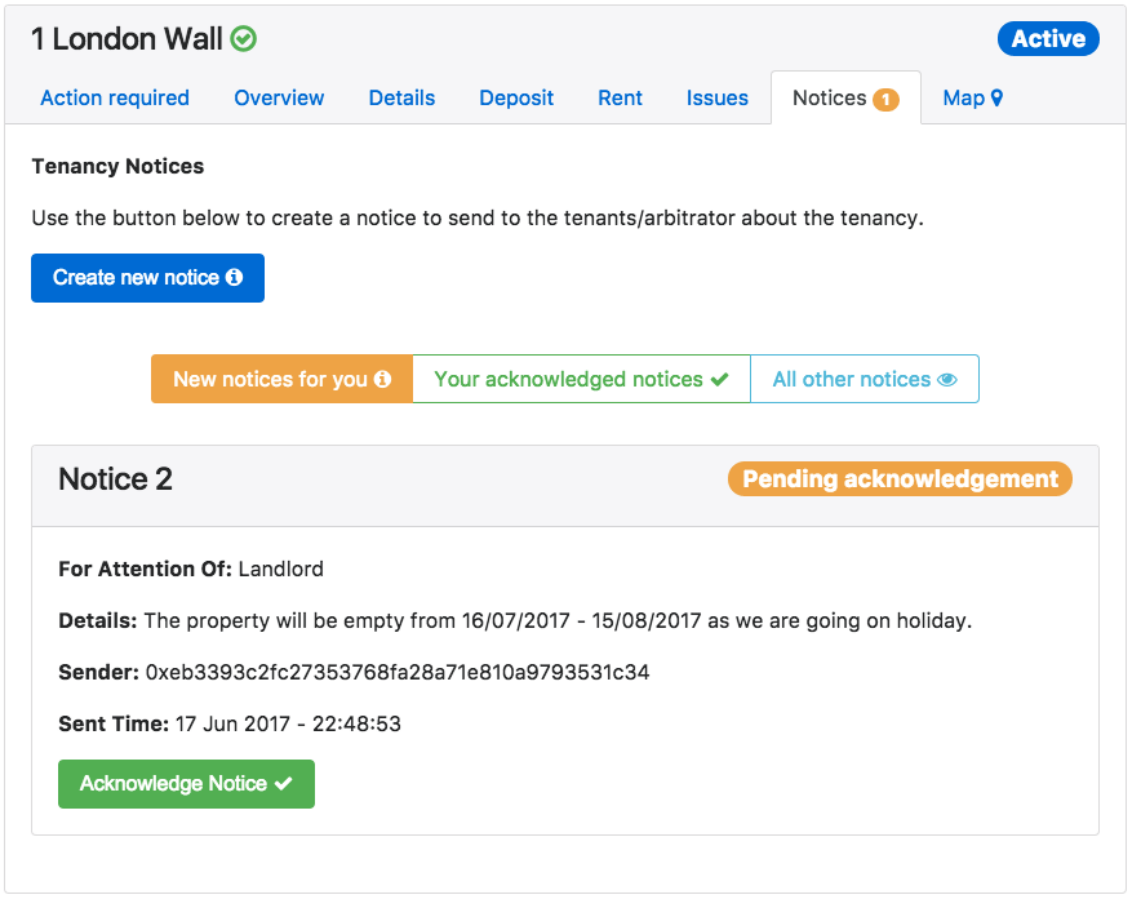
Tenancy Notices

Use the button below to create a notice to send to the tenants/arbitrator about the tenancy.

Notice details: Maintenance scheduled for 29/07/2017

Notice recipient: Tenants

Figure 4.14: Landlord notices tab - create new notice dropdown



Tenancy Notices

Use the button below to create a notice to send to the tenants/arbitrator about the tenancy.

Notice 2

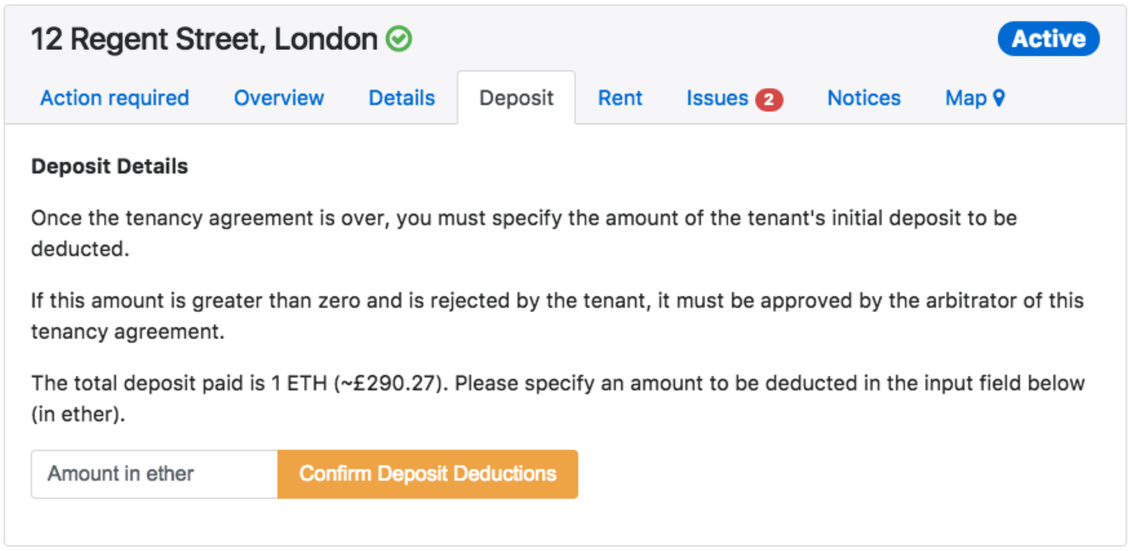
For Attention Of: Landlord

Details: The property will be empty from 16/07/2017 - 15/08/2017 as we are going on holiday.

Sender: 0xeb3393c2fc27353768fa28a71e810a9793531c34

Sent Time: 17 Jun 2017 - 22:48:53

Figure 4.15: Landlord notices tab - view new notices



Deposit Details

Once the tenancy agreement is over, you must specify the amount of the tenant's initial deposit to be deducted.

If this amount is greater than zero and is rejected by the tenant, it must be approved by the arbitrator of this tenancy agreement.

The total deposit paid is 1 ETH (~£290.27). Please specify an amount to be deducted in the input field below (in ether).

Figure 4.16: Landlord deposit deductions tab

4.1.2 Tenant

The tenant dashboard is similar to the landlord dashboard, except that it displays the amount of rent the tenant must pay that week (in GBP) instead of the amount of rent to withdraw (see Figure 4.17). The tenant screen also shows if they have more than one tenancy agreement active, but this is likely to be less common for a tenant than a landlord.

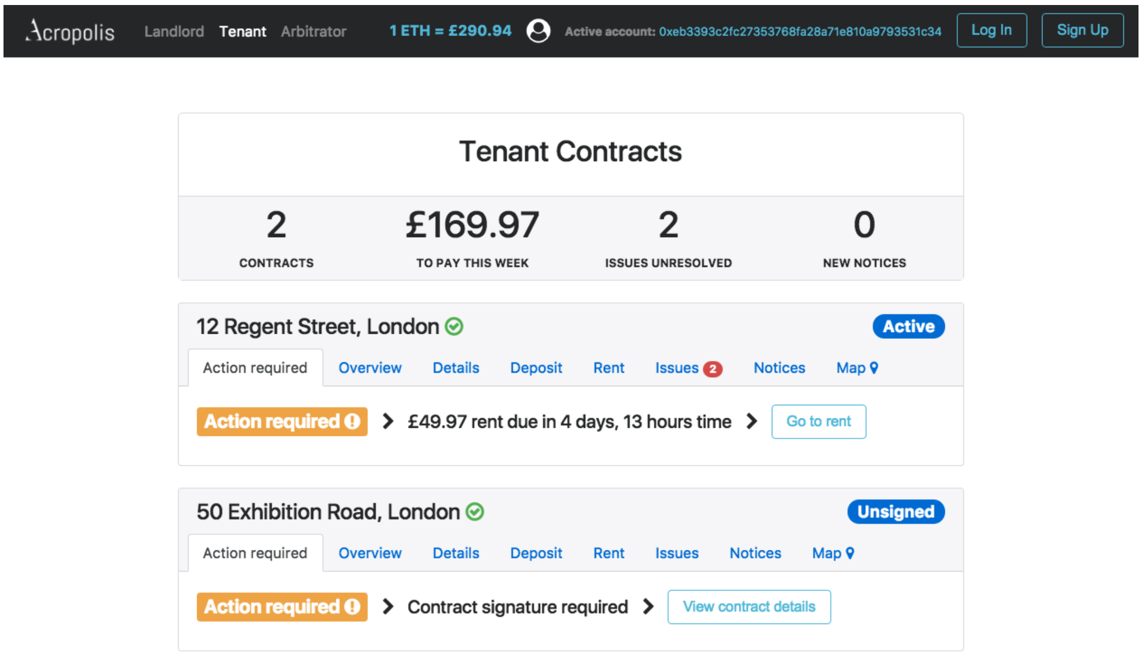
The contracts have similar features to those available to the landlord but they are tailored towards the actions that need to be performed by tenants. Each contract again shows the next step the tenant needs to perform on the 'action required' tab, enabling tenants to easily see which action they must complete next.

The main 'action required' tab will change much like the landlord's equivalent, and will transition through the following phases as actions are completed. The tenant will first be prompted to sign a new tenancy agreement which has been deployed with their account address listed as a tenant (see Figure 4.18). This will take them to the details tab where they can review the contract clauses and general details about the contract (see Figure 4.19). They must agree on the arbitrator chosen at this stage, as this is the party that will resolve disputes should they arise during the tenancy. Once the contract has been signed, the tenant must pay the deposit in ether (see Figure 4.20). This can be paid in a single instalment or multiple tenants can pay smaller amounts each adding up to the total (see Figure 4.21). After the deposit has been paid, the 'action required' tab displays to the tenant how much rent needs to be transferred by a given date (see Figure 4.22); the button shown directs them to the 'rent' tab where they can specify an amount in GBP to pay (see Figure 4.23). This input is converted in real time to a corresponding value in ether displayed next to it so the tenant knows how much ether they will be paying. A warning will also be displayed if tenant tries to enter more ether than is available in their Ethereum account's balance (see Figure 4.24).

At the end of the tenancy, once the landlord has specified any deposit deductions, the tenant has the opportunity to accept or reject these deductions (see Figure 4.26). If they accept them, the deposit and the deductions will be available to be immediately withdrawn by the tenant and landlord respectively. However, if they reject them, the arbitrator for the contract will be notified of a deposit dispute and must perform their arbitration duties before the deposit can be withdrawn (explained in section 4.1.3). Once the deductions have been approved, the tenant can withdraw the remaining deposit (see Figure 4.28).

As was mentioned in section 4.1.1, the tenant can report issues with the property to the landlord, as illustrated in Figure 4.30. The creation box allows details about the issue to be described and the issue severity to be selected from a dropdown list (high, medium, or low) (see Figure 4.29). By clicking 'submit issue', the tenant deploys the issue submission transaction to the blockchain, logging it as evidence that the issue was reported at that timestamp in the case of a future dispute. All users are able to use the filtering buttons to only display certain issues in the case where a lot have been reported. The 'High', 'Medium', and 'Low' buttons when clicked filter unresolved issues by their severity, and the other two buttons filter issues based on whether they are resolved or unresolved. Each issue displays details such as when it was created and who reported it, as well as details of the issue and its current state: 'awaiting confirmation', 'confirmed', or 'resolved'. A tenant is also able to mark an issue as resolved which immediately changes its state to 'resolved', unlike with the arbitrator or landlord, who require both parties to mark it as 'resolved' before it is considered resolved.

Tenants can also create and sent notices to either the landlord or arbitrator via the 'notices' tab, in exactly the same way as the landlord can.


Acropolis Landlord Tenant Arbitrator 1 ETH = £290.94 Active account: 0xeb3393c2fc27353768fa28a71e810a9793531c34 Log In Sign Up

Tenant Contracts

2	£169.97	2	0
CONTRACTS	TO PAY THIS WEEK	ISSUES UNRESOLVED	NEW NOTICES

12 Regent Street, London — Active

Action required | Overview | Details | Deposit | Rent | Issues 2 | Notices | Map

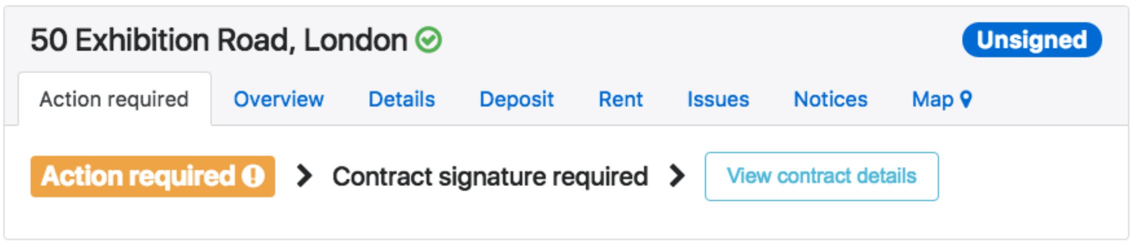
Action required > £49.97 rent due in 4 days, 13 hours time > Go to rent

50 Exhibition Road, London — Unsigned

Action required | Overview | Details | Deposit | Rent | Issues | Notices | Map

Action required > Contract signature required > View contract details

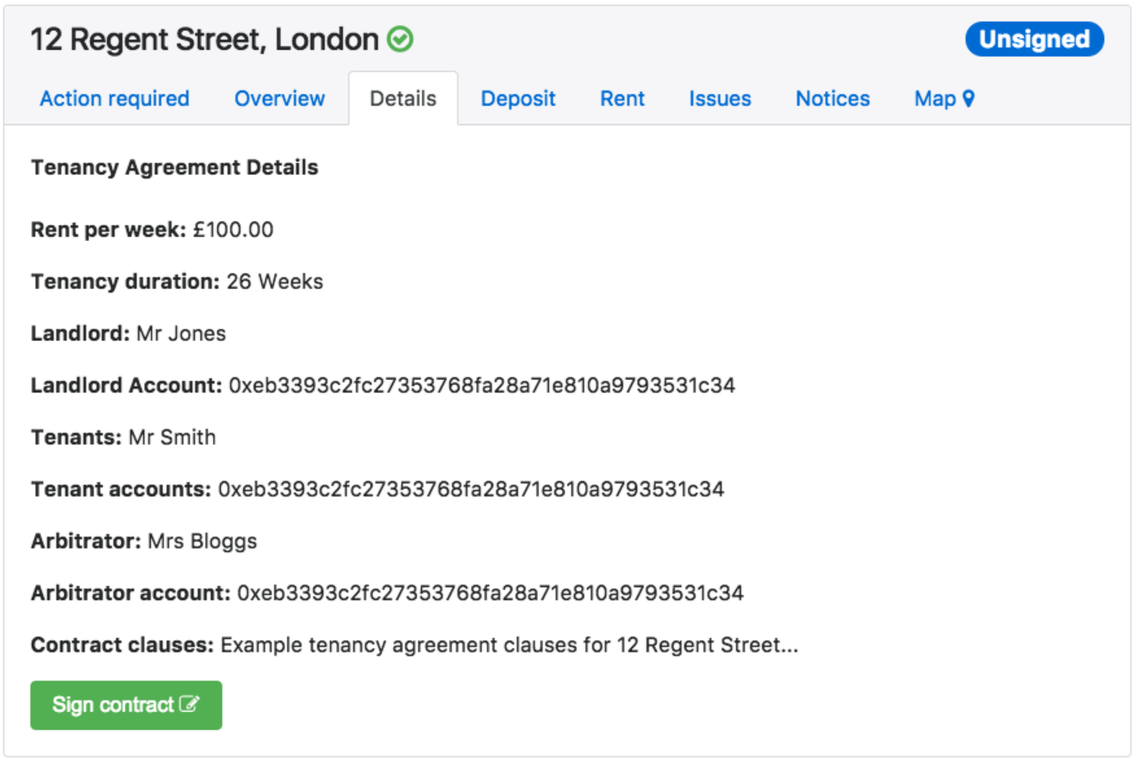
Figure 4.17: Tenant home screen


50 Exhibition Road, London — Unsigned

Action required | Overview | Details | Deposit | Rent | Issues | Notices | Map

Action required > Contract signature required > View contract details

Figure 4.18: Tenant action required tab - sign contract



Tenancy Agreement Details

Rent per week: £100.00

Tenancy duration: 26 Weeks

Landlord: Mr Jones

Landlord Account: 0xeb3393c2fc27353768fa28a71e810a9793531c34

Tenants: Mr Smith

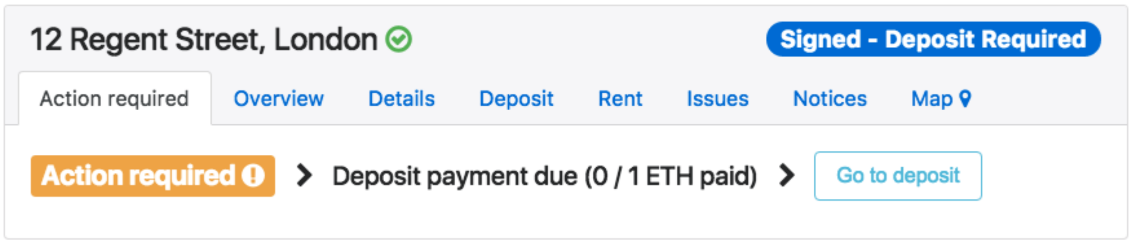
Tenant accounts: 0xeb3393c2fc27353768fa28a71e810a9793531c34

Arbitrator: Mrs Bloggs

Arbitrator account: 0xeb3393c2fc27353768fa28a71e810a9793531c34

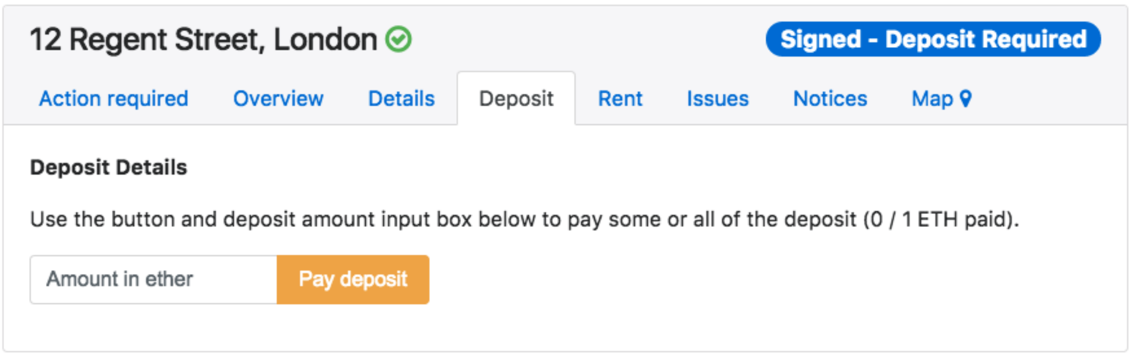
Contract clauses: Example tenancy agreement clauses for 12 Regent Street...

Figure 4.19: Tenant details tab - sign contract



Deposit payment due (0 / 1 ETH paid)

Figure 4.20: Tenant action required tab - pay deposit



Deposit Details

Use the button and deposit amount input box below to pay some or all of the deposit (0 / 1 ETH paid).

Figure 4.21: Tenant deposit tab - pay deposit

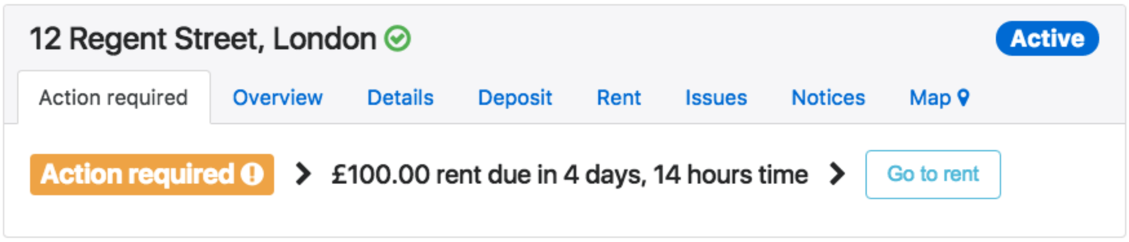
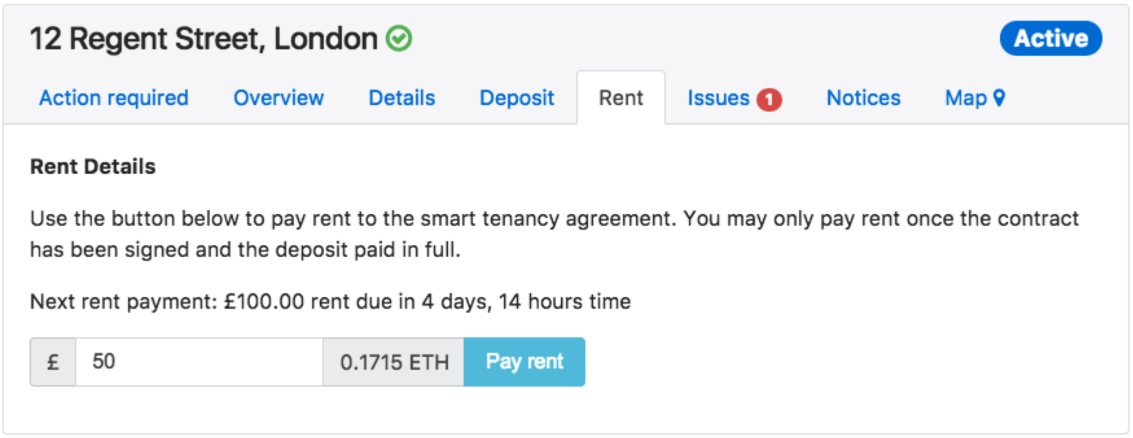


Figure 4.22: Tenant action required tab - rent due

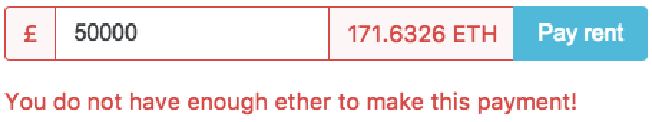


Rent Details

Use the button below to pay rent to the smart tenancy agreement. You may only pay rent once the contract has been signed and the deposit paid in full.

Next rent payment: £100.00 rent due in 4 days, 14 hours time

Figure 4.23: Tenant pay rent tab



You do not have enough ether to make this payment!

Figure 4.24: Tenant pay rent tab - insufficient ether funds warning

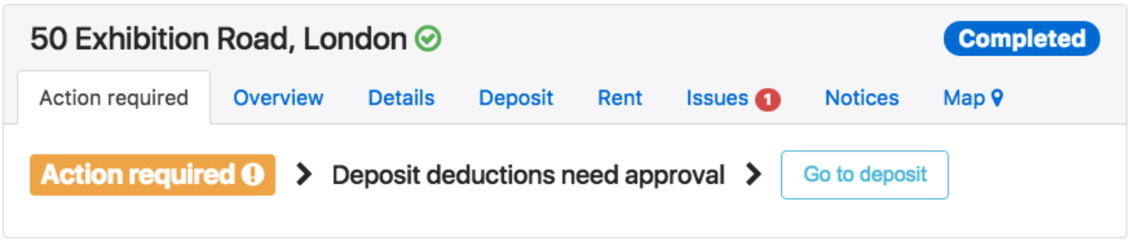
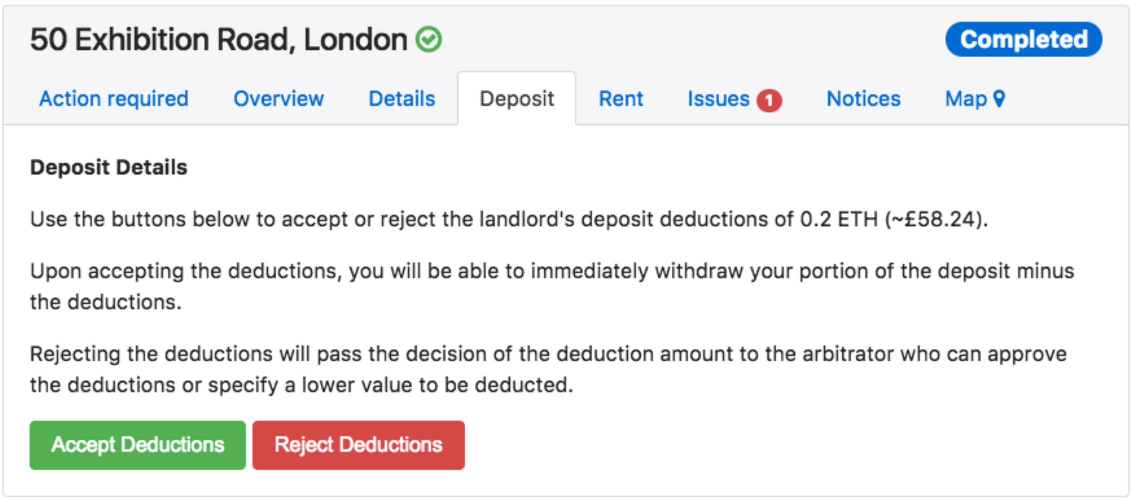


Figure 4.25: Tenant action required tab - deductions decision



Deposit Details

Use the buttons below to accept or reject the landlord's deposit deductions of 0.2 ETH (~£58.24).

Upon accepting the deductions, you will be able to immediately withdraw your portion of the deposit minus the deductions.

Rejecting the deductions will pass the decision of the deduction amount to the arbitrator who can approve the deductions or specify a lower value to be deducted.

Figure 4.26: Tenant deposit tab - deposit deductions need approving or rejecting

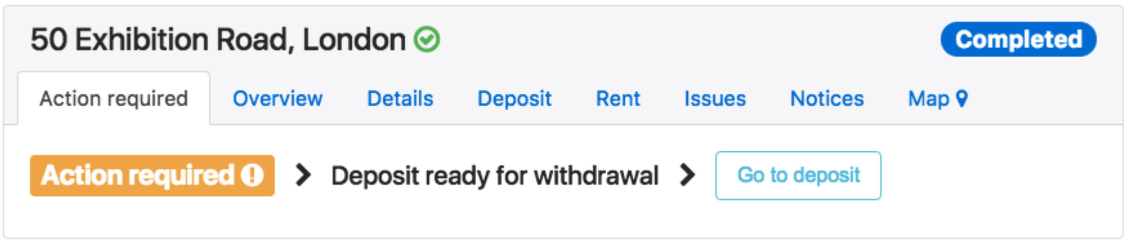
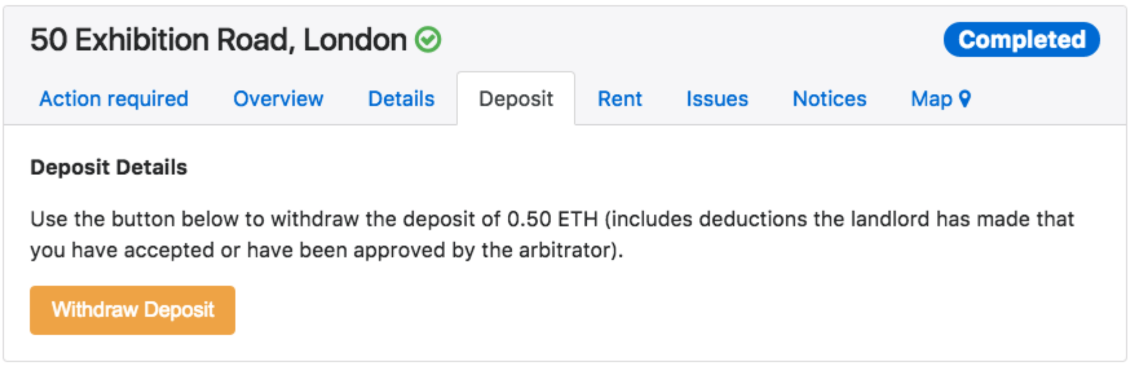


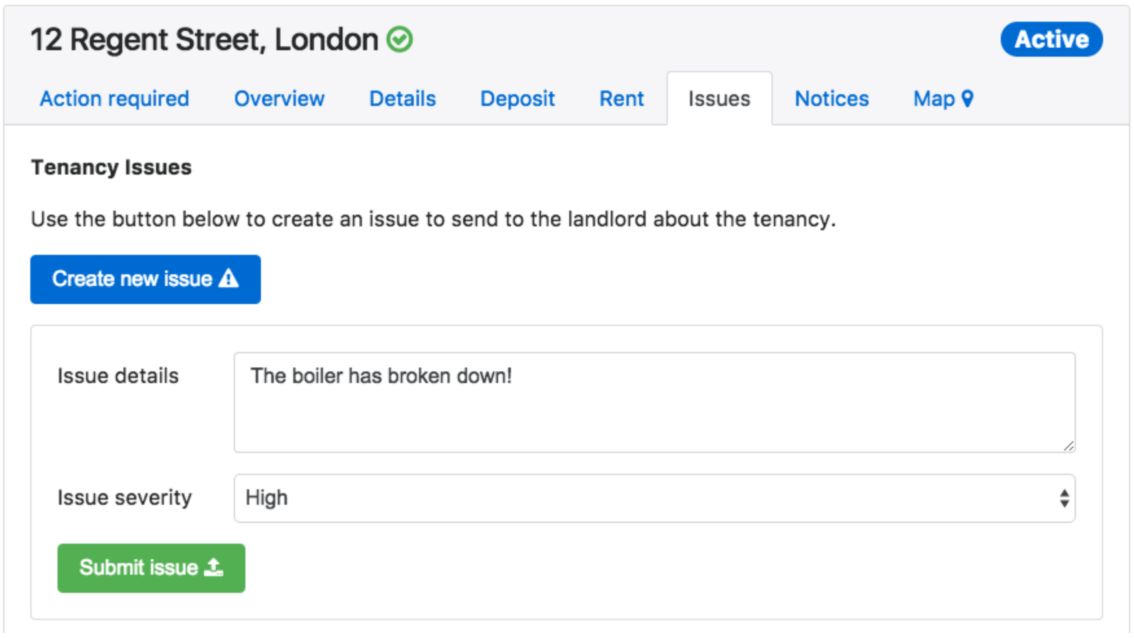
Figure 4.27: Tenant action required tab - withdraw deposit



Deposit Details

Use the button below to withdraw the deposit of 0.50 ETH (includes deductions the landlord has made that you have accepted or have been approved by the arbitrator).

Figure 4.28: Tenant deposit tab - withdraw deposit



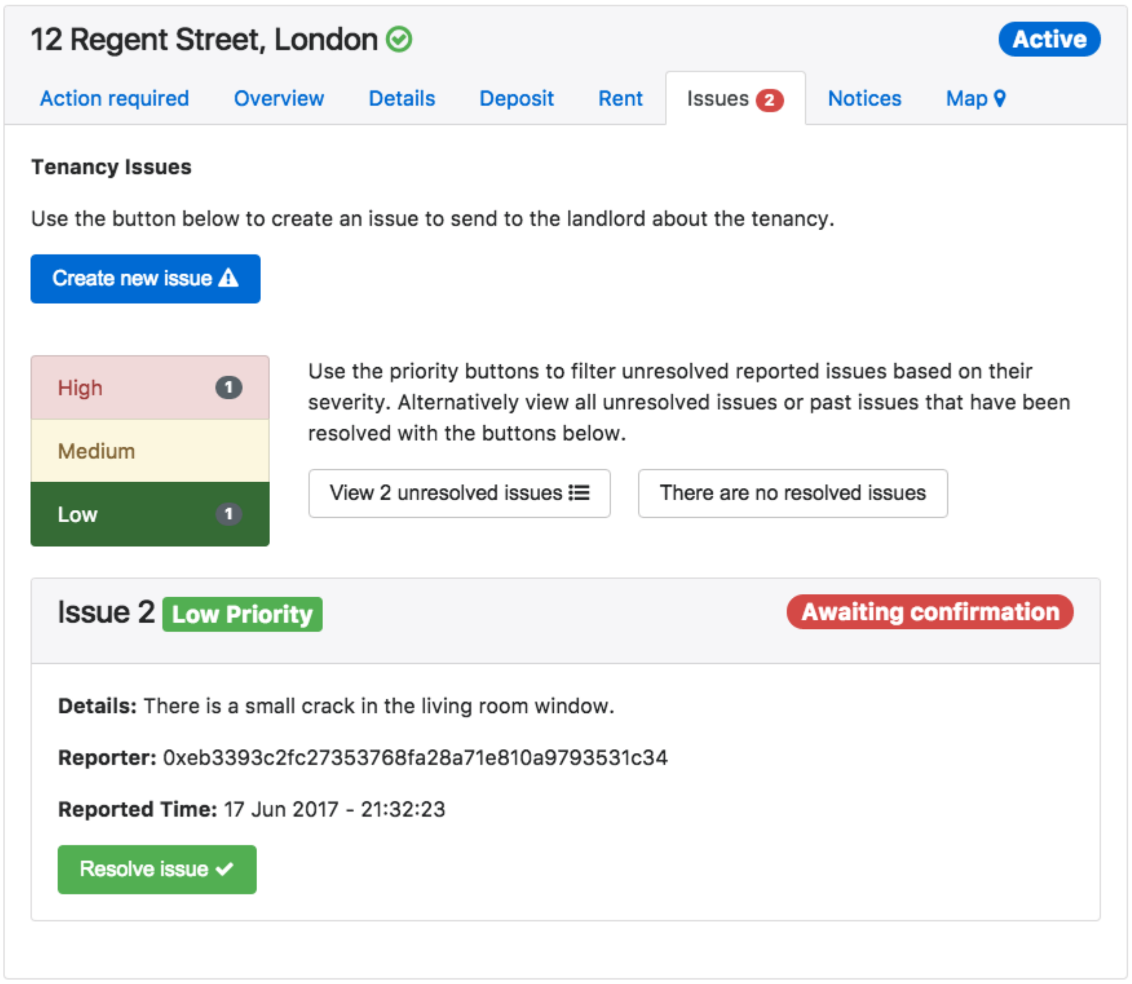
Tenancy Issues

Use the button below to create an issue to send to the landlord about the tenancy.

Issue details: The boiler has broken down!

Issue severity: High

Figure 4.29: Tenant issues tab - create new issue dropdown


12 Regent Street, London

Active

Tenancy Issues

Use the button below to create an issue to send to the landlord about the tenancy.

Use the priority buttons to filter unresolved reported issues based on their severity. Alternatively view all unresolved issues or past issues that have been resolved with the buttons below.

Issue 2 Low Priority

Awaiting confirmation

Details: There is a small crack in the living room window.

Reporter: 0xeb3393c2fc27353768fa28a71e810a9793531c34

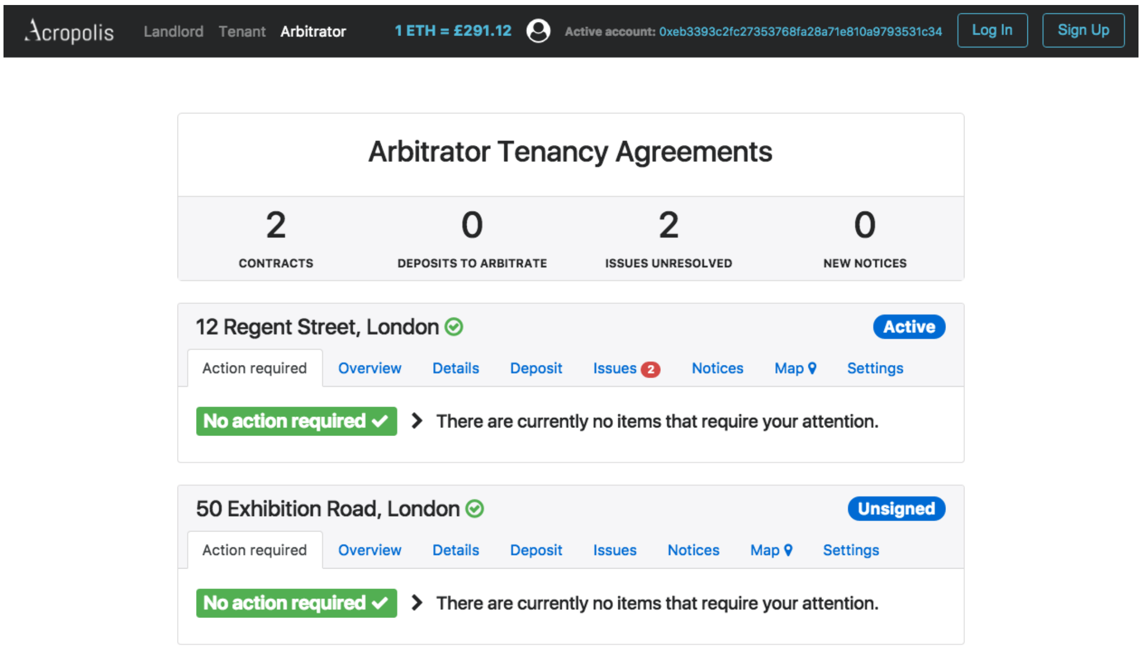
Reported Time: 17 Jun 2017 - 21:32:23

Figure 4.30: Tenant issues tab with a low priority filter applied

4.1.3 Arbitrator

The arbitrator has a similar dashboard to both the landlord and tenant, as shown in Figure 4.31, but there are fewer methods of interaction with the tenancy agreement available. Their purpose is to oversee the contract and resolve issues if the landlord has resolved them but the tenant has not (an issue dispute). However, the most important part of the arbitrator's job is to arbitrate the deposit deductions if the tenant rejects those specified by the landlord. Again, the simple 'action required' tab alerts the arbitrator if they need to perform arbitration (Figure 4.32), otherwise it tells them that no action is required (Figure 4.33).

If the tenant rejects the landlord's deposit deductions, the arbitrator must navigate to the 'deposit' tab, as displayed in Figure 4.34. They must then either accept the deductions the landlord originally specified, overruling the tenant, or they can specify a new amount to be deducted which must be less than the original amount. They cannot set a higher deduction amount than the landlord originally specified however, to ensure the tenant does not become worse off by requiring their services.



Arbitrator Tenancy Agreements

2	0	2	0
CONTRACTS	DEPOSITS TO ARBITRATE	ISSUES UNRESOLVED	NEW NOTICES

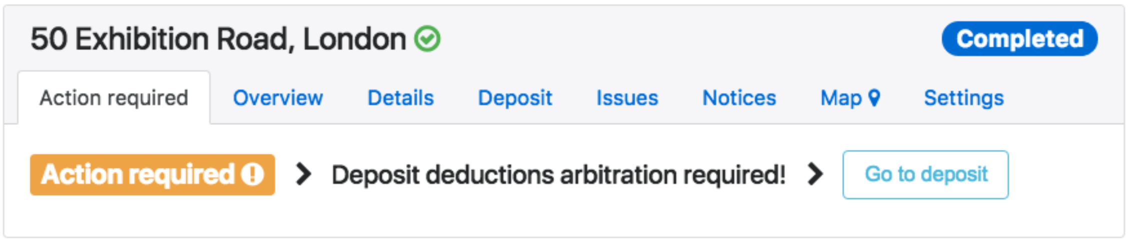
12 Regent Street, London ✓ Active

No action required ✓ > There are currently no items that require your attention.

50 Exhibition Road, London ✓ Unsigned

No action required ✓ > There are currently no items that require your attention.

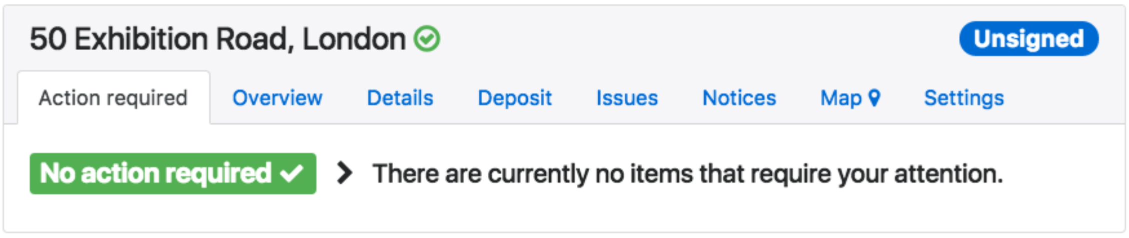
Figure 4.31: Arbitrator home screen



50 Exhibition Road, London ✓ Completed

Action required ! > Deposit deductions arbitration required! > Go to deposit

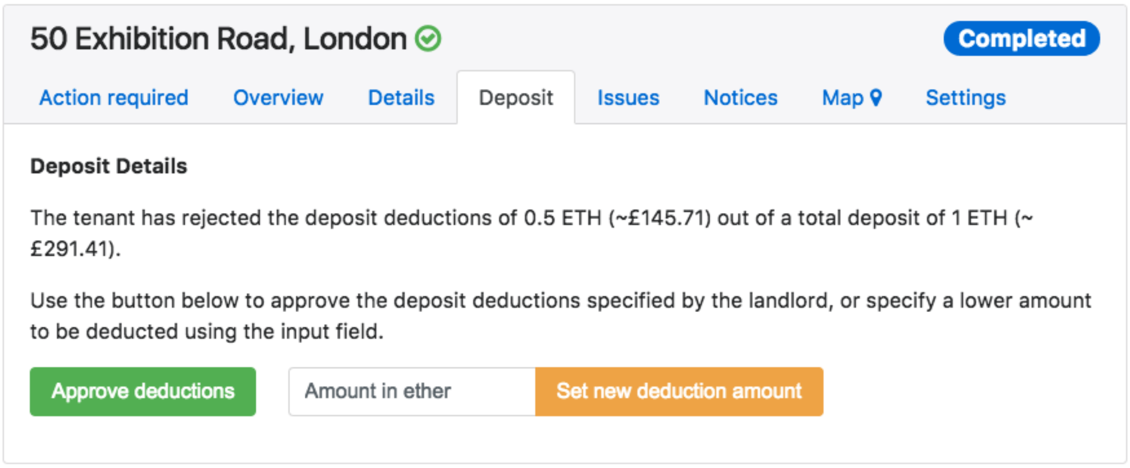
Figure 4.32: Arbitrator action required tab - deposit needs arbitration



50 Exhibition Road, London ✓ Unsigned

No action required ✓ > There are currently no items that require your attention.

Figure 4.33: Arbitrator no action required tab



Deposit Details

The tenant has rejected the deposit deductions of 0.5 ETH (~£145.71) out of a total deposit of 1 ETH (~ £291.41).

Use the button below to approve the deposit deductions specified by the landlord, or specify a lower amount to be deducted using the input field.

Figure 4.34: Arbitrator deposit tab - arbitration

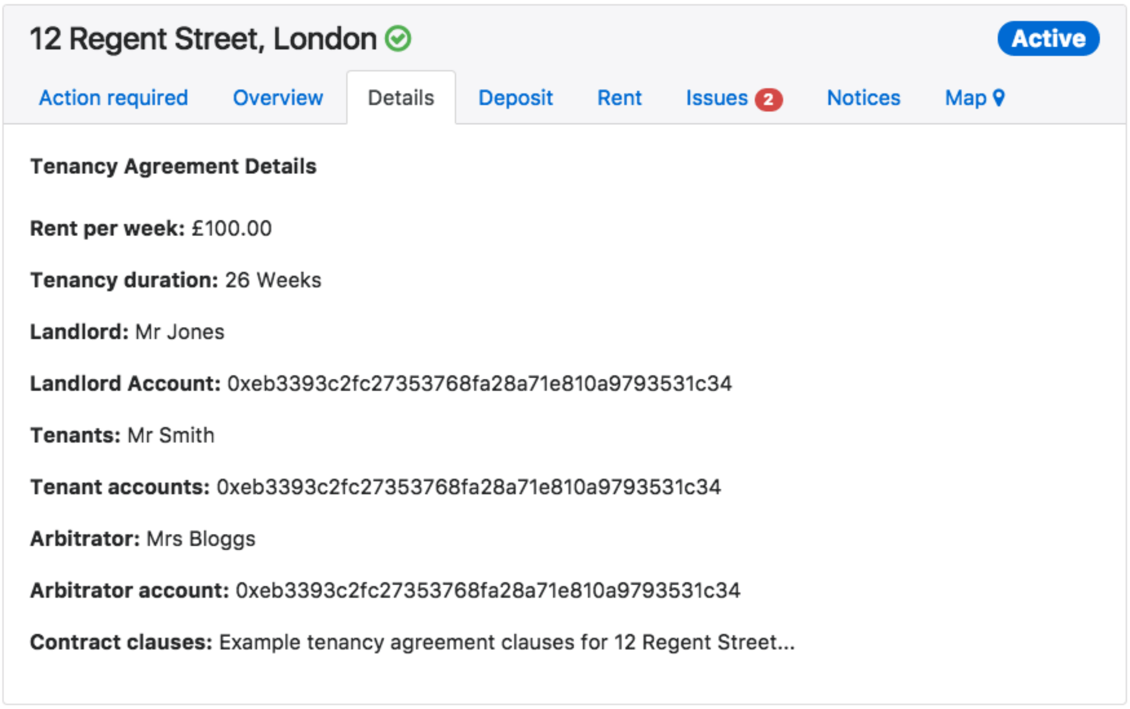
4.2 Features Available to All Users

There is a lot of common functionality between all types of users of Acropolis. All users participating in a tenancy agreement can view details of it on the ‘details’ tab (see Figure 4.35), this includes the contract clauses which are retrieved from the back-end database. The green tick symbol next to the property’s address (shown in Figure 4.36) indicates that the hash of the contract clauses retrieved from the database matches the contract clause hash on the blockchain. Hovering over this symbol with the cursor displays a tooltip explaining to the user that the contract has been deployed successfully (i.e. the database record agrees with the blockchain). A red cross is shown to the user otherwise to alert them of this mismatch (which should never occur if the system is functioning correctly).

The user can view the ether balance of their active address by hovering over it in the navbar as shown in Figure 4.37. This address also acts as a hyperlink to Etherscan [29] where they can use the blockchain explorer to view all transactions to and from their Ethereum account. The navbar also displays the current exchange rate from ether to GBP from the CryptoCompare API [11]; this is also a hyperlink to the CryptoCompare ether to GBP overview page where the user can view market information and historical price charts [12].

All users for a given contract can view all of the notices for that tenancy agreement, regardless of who they were sent to, thus improving the transparency of the system. For a given user, notices can be filtered by buttons which show new or acknowledged notices, or all notices not sent directly to that user (see Figure 4.38).

Finally, there is also a ‘map’ tab which displays an embedded Google map showing the property’s geographical location, useful for landlords or property managers that deal with large property portfolios (see Figure 4.39).


12 Regent Street, London

Active

Tenancy Agreement Details

Rent per week: £100.00

Tenancy duration: 26 Weeks

Landlord: Mr Jones

Landlord Account: 0xeb3393c2fc27353768fa28a71e810a9793531c34

Tenants: Mr Smith

Tenant accounts: 0xeb3393c2fc27353768fa28a71e810a9793531c34

Arbitrator: Mrs Bloggs

Arbitrator account: 0xeb3393c2fc27353768fa28a71e810a9793531c34

Contract clauses: Example tenancy agreement clauses for 12 Regent Street...

Figure 4.35: Contract details tab

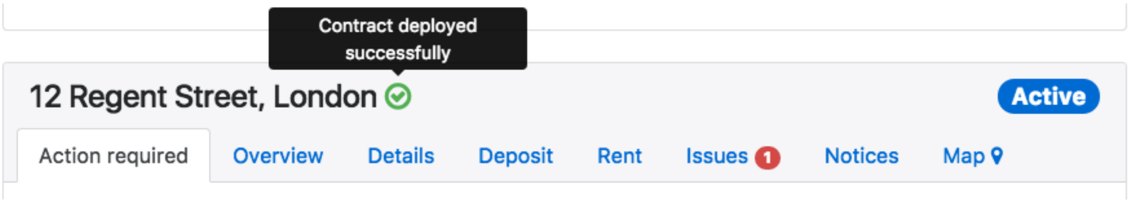


Figure 4.36: Contract clauses hash verified symbol tooltip

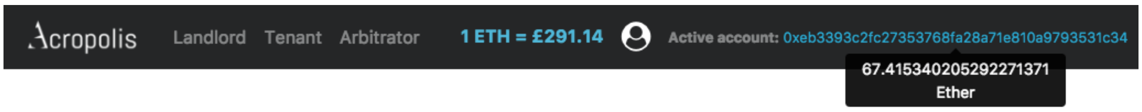
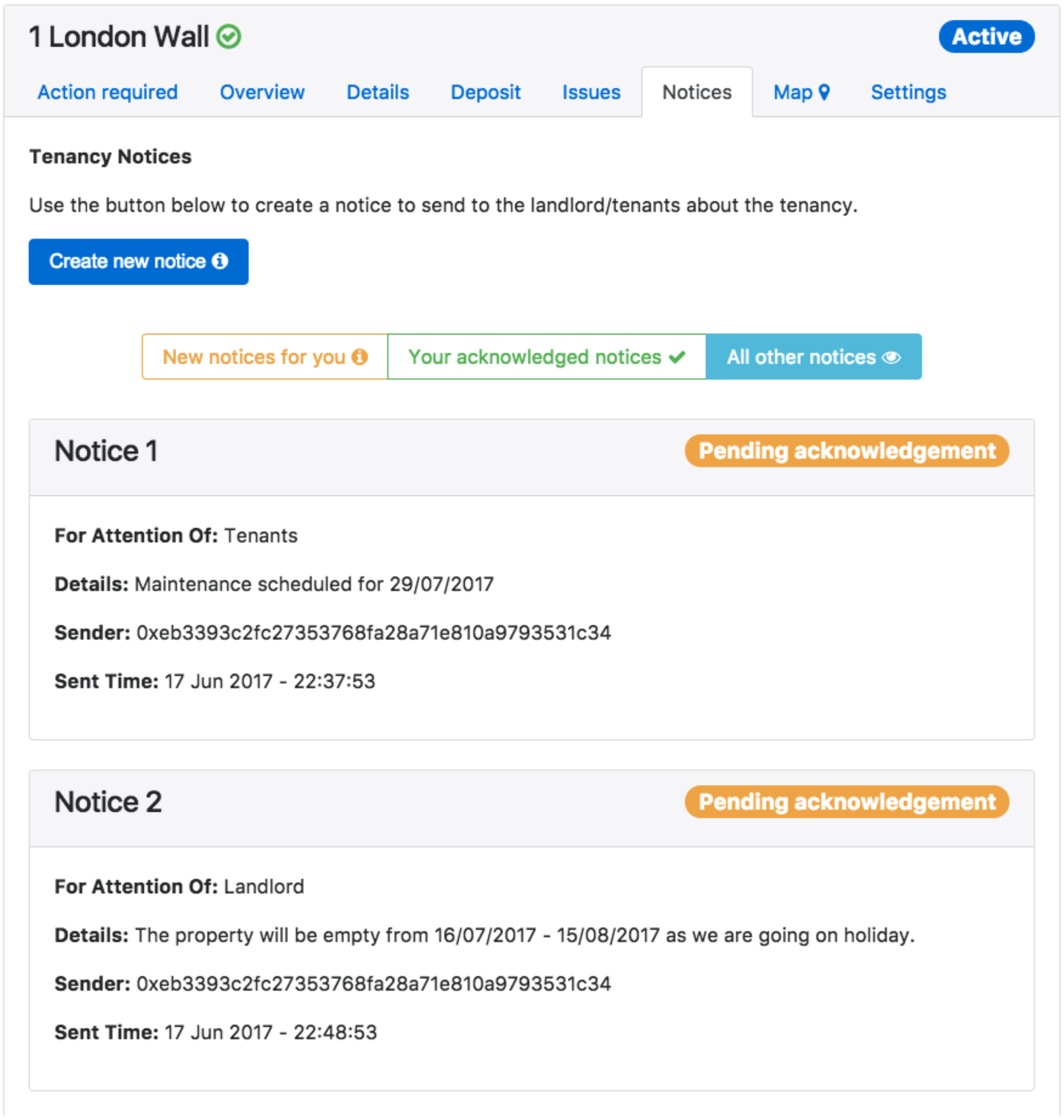

Acropolis
Landlord
Tenant
Arbitrator
1 ETH = £291.14
Active account: 0xeb3393c2fc27353768fa28a71e810a9793531c34
67.415340205292271371 Ether

Figure 4.37: Ether balance navbar tooltip



Tenancy Notices

Use the button below to create a notice to send to the landlord/tenants about the tenancy.

Notice 1 — Pending acknowledgement

For Attention Of: Tenants

Details: Maintenance scheduled for 29/07/2017

Sender: 0xeb3393c2fc27353768fa28a71e810a9793531c34

Sent Time: 17 Jun 2017 - 22:37:53

Notice 2 — Pending acknowledgement

For Attention Of: Landlord

Details: The property will be empty from 16/07/2017 - 15/08/2017 as we are going on holiday.

Sender: 0xeb3393c2fc27353768fa28a71e810a9793531c34

Sent Time: 17 Jun 2017 - 22:48:53

Figure 4.38: Notices tab - all notices sent from the view of the arbitrator

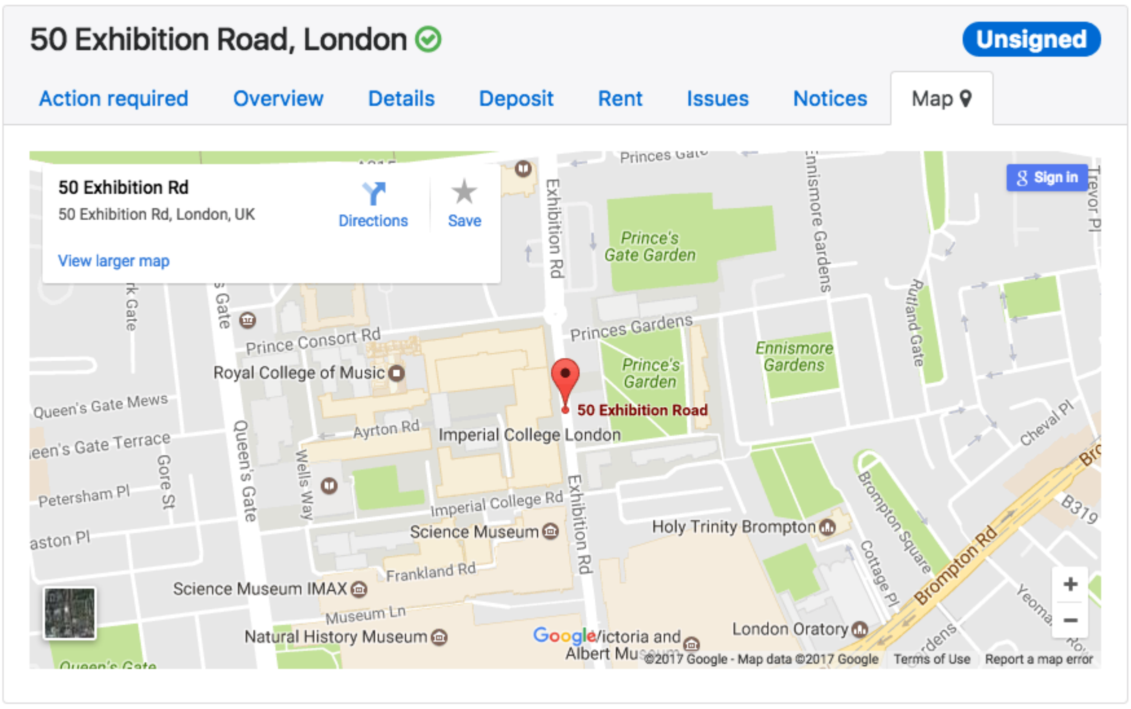


Figure 4.39: Property map tab

Chapter 5

Implementation

This chapter will describe the implementation details of the application and will discuss why certain technologies and techniques were used.

5.1 Architecture

Figure 5.1 shows a high-level architecture diagram of the implementation of the system. The user accesses the client-side Meteor application via the InterPlanetary File System (IPFS) (either locally if they are running the IPFS daemon, or via a public gateway) [43]. The application communicates with the smart contracts deployed to the Ethereum network via an Ethereum node running locally (included with the Mist browser). The smart tenancy agreements also interact with the Oraclize system as shown on the diagram, however this occurs indirectly via the Oraclize smart contract interfaces. Sensitive contract data is obtained from a Node.js API service running on the Microsoft Azure cloud which is connected to a MongoDB instance. This JSON data is queried via an HTTPS AJAX request to the API endpoint.

The following sections will discuss each of these components in more detail.

5.2 Smart Tenancy Agreement

This section will go through the main features of the tenancy agreement smart contract. I have coded the logic in Solidity [75], the flagship smart contract language for Ethereum which is constantly in development with new features regularly being added. There are several other languages available to code smart contracts:

- **Serpent**: Python-like language and was previously the main language used to code smart contracts on Ethereum before Solidity. [26]
- **LLL**: Low-level Lisp-like language.

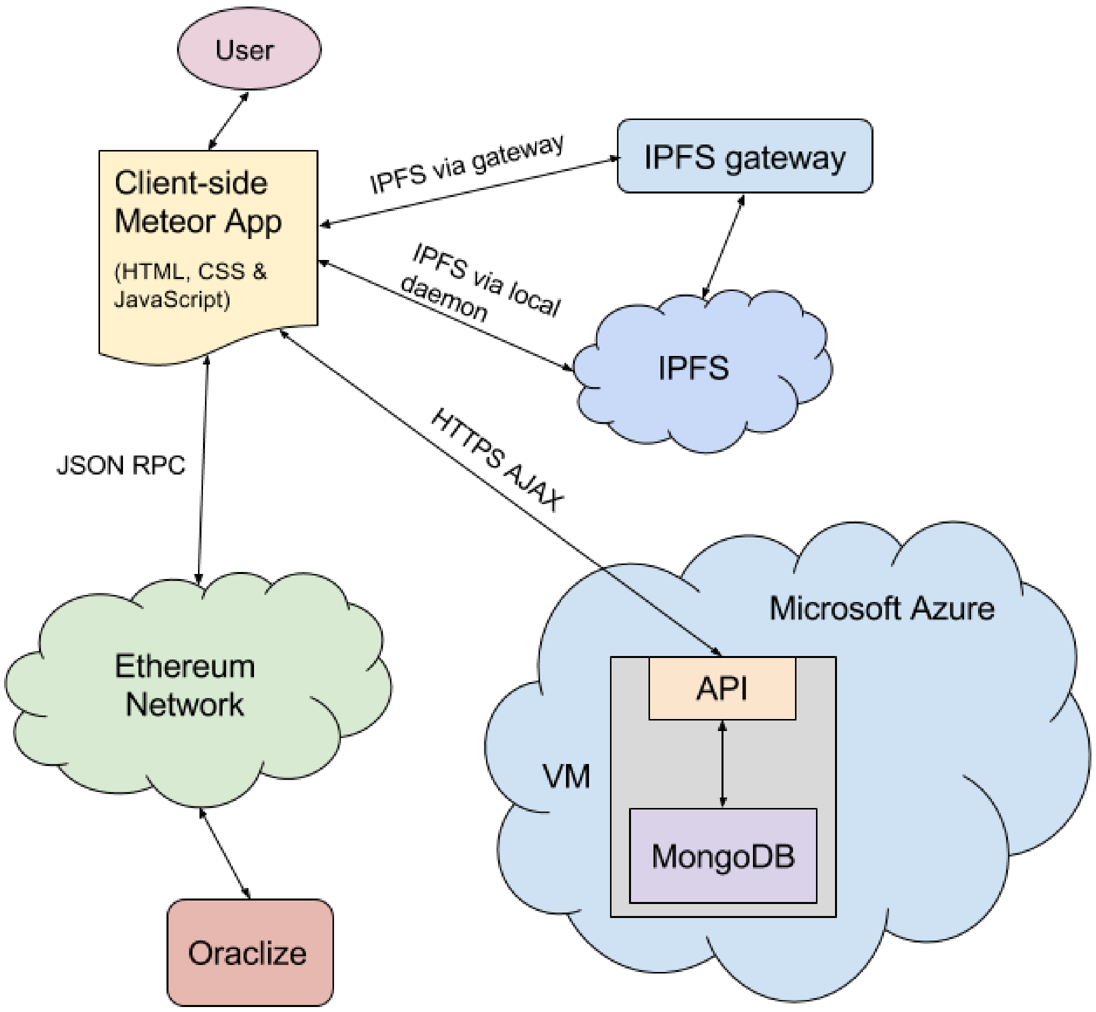


Figure 5.1: Architecture diagram for application

- **Viper**: Experimental Python-like decidable language being developed with features such as strong typing, fixed point number support, and the ability to compute the maximum amount of gas a function call can consume. [34]

However, these are either rarely used any more for new smart contract development or are experimental in the case of Viper, so Solidity was a straightforward choice.

5.2.1 Development

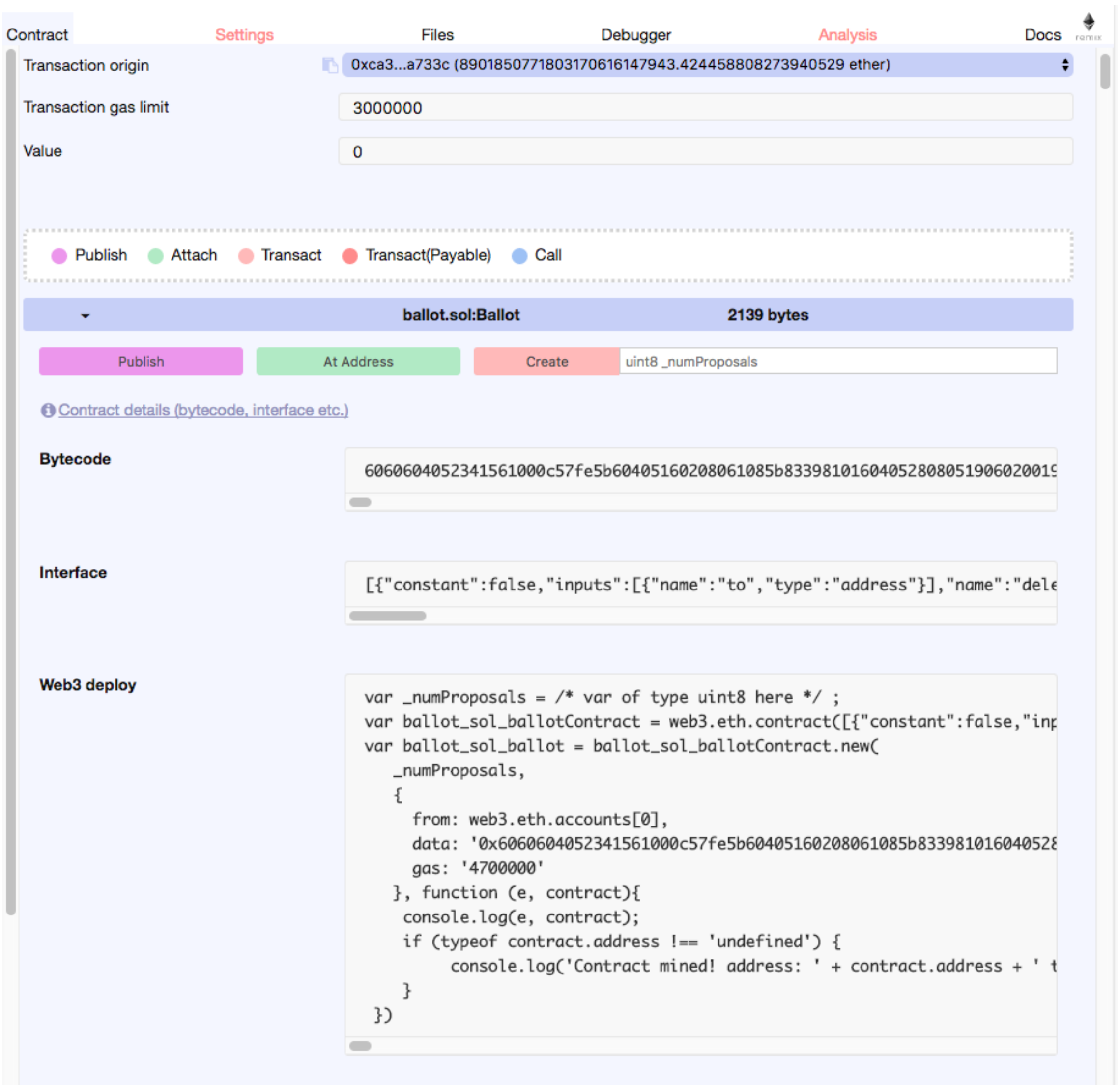
I developed the Solidity code using Remix, the Ethereum Solidity browser compiler and IDE (see Figure 5.2) [74]. This automatically recompiles the code every time a change is made and alerts the user of any syntax errors. It also allows the user to deploy the contract to a JavaScript VM for debugging without deploying it to the Ethereum testnet.

The compiler generates the contract bytecode and interface, which are needed to be able to deploy the contract using the JavaScript Web3 library. This code is also automatically generated by the compiler, and is pictured in Figure 5.2. The screenshot does not include the contract code which is located immediately to the left of the contract details panel. Using this tool greatly speeds up the contract development, debugging, and deployment process.

5.2.2 Solidity Implementation

State Transitions

In the smart contract an 'enum' is used to represent the current stage of the agreement which has possible values `Unsigned`, `SignedDepositRequired`, `Active` and `Completed`. When actions are completed, a variable storing the current stage enum is updated with the new stage. Function 'modifiers' are used in function calls which make them only executable when the contract is in a certain stage, calling `throw` otherwise, reverting any changes the function made. These state transitions make it easier to reason about the smart contract, enforcing an ordering to the functions being called, therefore helping to reduce the likelihood of subtle bugs existing in the code. The Solidity documentation describes this state machine design pattern in further detail [80]. Figure 5.3 illustrates the possible stages that the smart contract transitions through; the Solidity code has multiple checks before a transition can occur, ensuring all of the correct conditions are met beforehand.



Transaction origin 0xca3...a733c (8901850771803170616147943.424458808273940529 ether)

Transaction gas limit 3000000

Value 0

● Publish ● Attach ● Transact ● Transact(Payable) ● Call

ballot.sol:Ballot 2139 bytes

uint8 _numProposals

Contract details (bytecode, interface etc.)

Bytecode

```
6060604052341561000c57fe5b60405160208061085b833981016040528080519060200191
```

Interface

```
[{"constant":false,"inputs":[{"name":"to","type":"address"}],"name":"dele
```

Web3 deploy

```
var _numProposals = /* var of type uint8 here */ ;
var ballot_sol_ballotContract = web3.eth.contract([{"constant":false,"inp
var ballot_sol_ballot = ballot_sol_ballotContract.new(
   _numProposals,
   {
     from: web3.eth.accounts[0],
     data: '0x6060604052341561000c57fe5b60405160208061085b83398101604052
     gas: '4700000'
   }, function (e, contract){
    console.log(e, contract);
    if (typeof contract.address !== 'undefined') {
         console.log('Contract mined! address: ' + contract.address + ' t
    }
 })
```

Figure 5.2: Screenshot of the Solidity browser compiler and IDE displaying the contract details panel [74]

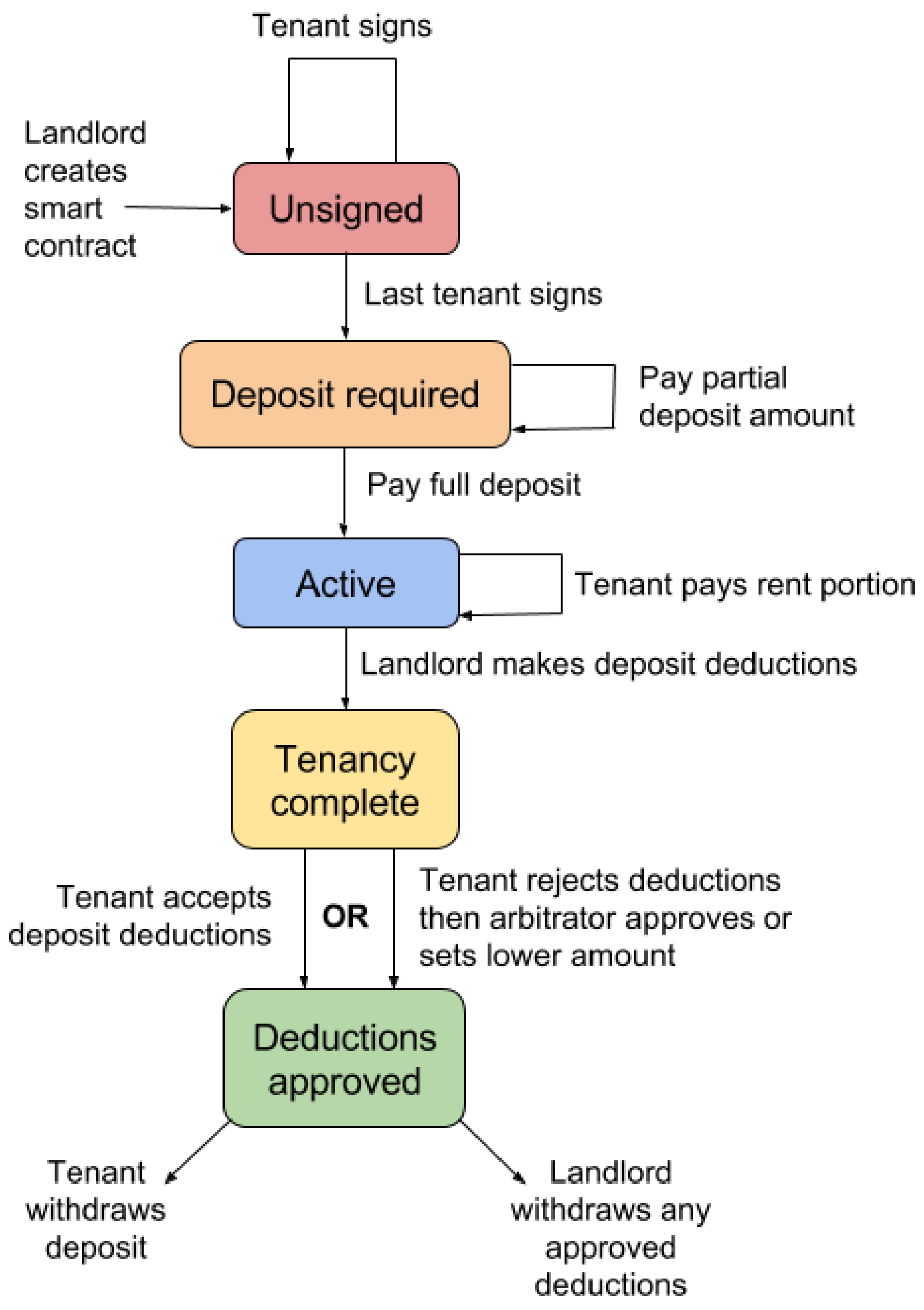

Tenant signs
Landlord creates smart contract
Unsigned
Last tenant signs
Deposit required
Pay partial deposit amount
Pay full deposit
Active
Tenant pays rent portion
Landlord makes deposit deductions
Tenancy complete
Tenant accepts deposit deductions
OR
Tenant rejects deductions then arbitrator approves or sets lower amount
Deductions approved
Tenant withdraws deposit
Landlord withdraws any approved deductions

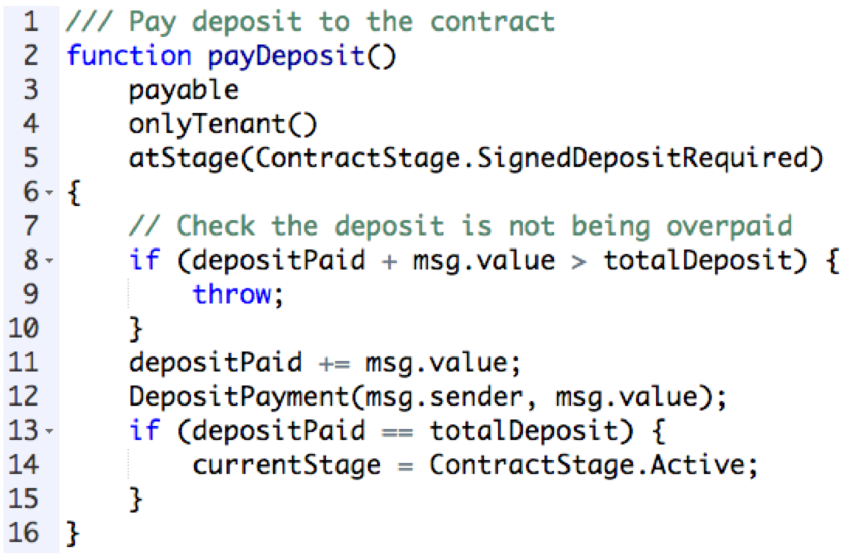
Figure 5.3: Diagram of the smart contract state transitions

Withdraw vs Send Payment Pattern

An important design consideration during development of the contract is the trade-off between sending ether directly to the landlord when making a rent payment, or transferring the rent to the contract and allowing the landlord to withdraw the rent themselves. There are fewer contract interactions required if the payment is sent directly to the landlord, however, the Solidity documentation warns of security risks associated with transferring rent directly because the transaction can always potentially fail (run out of gas) [79]. It advises to use the 'withdrawal' pattern [81] which is much safer; therefore I will be using this method as I believe it is of paramount importance to maximise security in these tenancy agreement contracts.

Figure 5.4 shows the function in the smart contract used for paying the deposit to the balance of the contract. This is the same principle as used for the rent payments to the contract, but these are more complicated as they involve querying the oracle service and so will be discussed in section 5.2.2. The main features of this function are as follows:

- The `payable` keyword on line 3 allows the function to receive ether.
- Line 4 is a function modifier that ensures only a tenant can pay the deposit.
- Line 5 ensures the contract is in the correct transition stage and is currently accepting deposit payments.
- Line 11 is when the `depositPaid` variable is incremented with the amount of the deposit sent with the function call.
- Line 12 creates a `DepositPayment` event that is listened for on the application front-end to update the UI accordingly.
- Line 14 advances the contract stage to `Active` once the deposit has been paid in full.

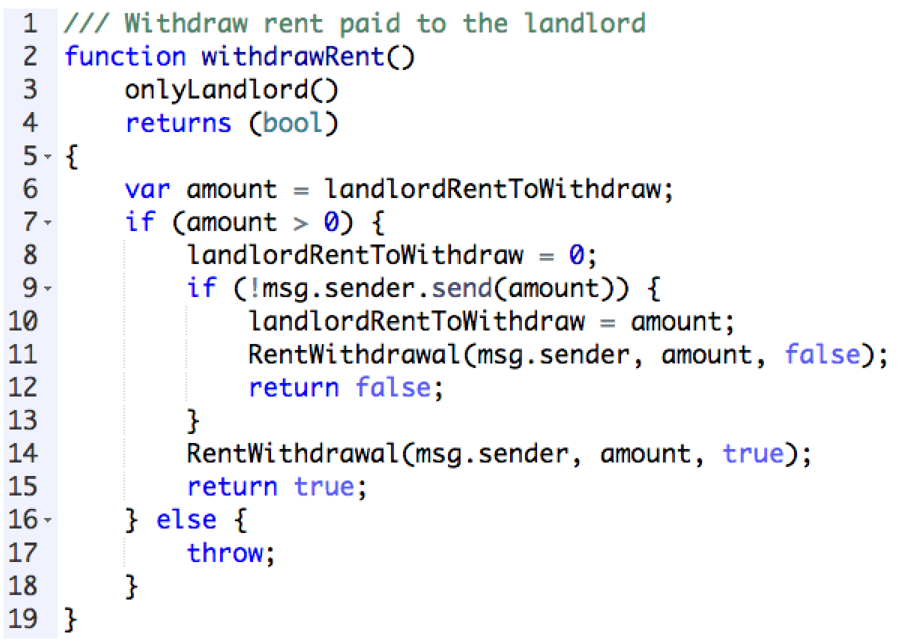


```
 1  /// Pay deposit to the contract
 2  function payDeposit()
 3      payable
 4      onlyTenant()
 5      atStage(ContractStage.SignedDepositRequired)
 6  {
 7      // Check the deposit is not being overpaid
 8      if (depositPaid + msg.value > totalDeposit) {
 9          throw;
10      }
11      depositPaid += msg.value;
12      DepositPayment(msg.sender, msg.value);
13      if (depositPaid == totalDeposit) {
14          currentStage = ContractStage.Active;
15      }
16  }
```

Figure 5.4: Solidity function used by the tenant to pay the tenancy agreement deposit.

Figure 5.5 shows the function used to withdraw the rent stored in the smart contract. This function heeds the security advice in the Solidity documentation to avoid re-entrancy bugs [78] and uses the "Checks-Effects-Interactions" pattern [76]. I will briefly go through the lines in this function that are of note.

- Line 3 ensures that only the landlord can withdraw the rent.
- Line 7 is the check from the "Checks-Effects-Interactions" pattern, ensuring that there is a positive and non-zero amount of rent to withdraw.
- Line 8 is the effect from the "Checks-Effects-Interactions" pattern, and sets the amount of rent to withdraw to zero. This is what prevents re-entrancy bugs where the other contract could call back into the contract before the interaction is complete, potentially enabling the contract to be drained of ether by the malicious actor. By setting this value to zero, calling back into the function would not get past the initial check on line 7.
- Line 9 is the interaction from the "Checks-Effects-Interactions" pattern and sends the ether back to the account that called the function.
- Lines 11 and 14 are `RentWithdrawal` events which are listened for on the client-side code and trigger the application UI to update accordingly.



```
 1 /// Withdraw rent paid to the landlord
 2 function withdrawRent()
 3     onlyLandlord()
 4     returns (bool)
 5 {
 6     var amount = landlordRentToWithdraw;
 7     if (amount > 0) {
 8         landlordRentToWithdraw = 0;
 9         if (!msg.sender.send(amount)) {
10             landlordRentToWithdraw = amount;
11             RentWithdrawal(msg.sender, amount, false);
12             return false;
13         }
14         RentWithdrawal(msg.sender, amount, true);
15         return true;
16     } else {
17         throw;
18     }
19 }
```

Figure 5.5: Solidity function used by the landlord to withdraw the rent currently paid to the contract.

Oraclize Ether to GBP Price Lookup

Rent payments are a little more complicated than the deposit payments explained in the previous section. As discussed in section 3.1.3, each rent payment transaction uses the Oraclize oracle service [66] to obtain the current ether to GBP exchange rate in order to determine the equivalent amount of GBP paid when transferring the ether. The smart contract uses the Oraclize API and inherits from the contract 'usingOraclize' which provides functions that interact with the Oraclize interface contracts [37].

Figure 5.6 shows the function `payRent` which tenants use to transfer rent to the contract. The key parts of this function are as follows:

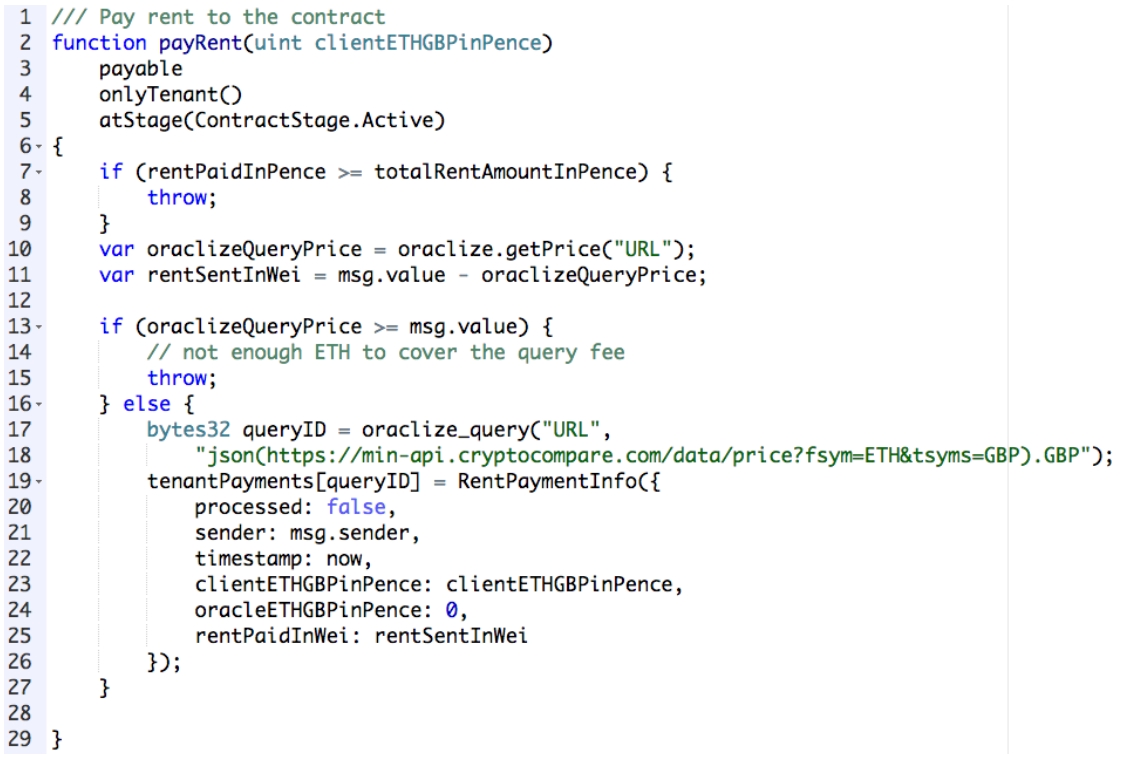
- In line 2 the parameter `clientETHGBPinPence` is passed to the function and represents the exchange rate in pence recorded on the client-side application. This of course can be manipulated by the user, and so the Oraclize value is used for the actual rent calculation. This parameter is simply used as a sanity check to ensure the value received from Oraclize is sensible in the callback function. If there is a difference greater than a certain threshold, the payment is invalidated.

- Line 10 gets the fee that Oraclize charge for a URL query, which is currently \$0.01 plus \$0.04 for a TLSNotary Proof to be included to prove the result has not been tampered with [68].

- Line 17 performs the Oraclize query requesting the current ether to GBP exchange rate from CryptoCompare [11]. The result is returned via a callback to the function `__callback`, which calculates the GBP amount paid towards the rent and updates the contract variables accordingly.

- Line 19 adds a new `RentPaymentInfo` struct to the `tenantPayments` mapping so that the callback function has access to data relating to the Oraclize function call. It is indexed with the query ID obtained on line 17.

There are many other functions in the tenancy agreement smart contract that have not been described in this section, such as for signing the contract, reporting issues, and submitting notices, but have been left out for brevity.



```
/// Pay rent to the contract
function payRent(uint clientETHGBPinPence)
    payable
    onlyTenant()
    atStage(ContractStage.Active)
{
    if (rentPaidInPence >= totalRentAmountInPence) {
        throw;
    }
    var oraclizeQueryPrice = oraclize.getPrice("URL");
    var rentSentInWei = msg.value - oraclizeQueryPrice;

    if (oraclizeQueryPrice >= msg.value) {
        // not enough ETH to cover the query fee
        throw;
    } else {
        bytes32 queryID = oraclize_query("URL",
            "json(https://min-api.cryptocompare.com/data/price?fsym=ETH&tsyms=GBP).GBP");
        tenantPayments[queryID] = RentPaymentInfo({
            processed: false,
            sender: msg.sender,
            timestamp: now,
            clientETHGBPinPence: clientETHGBPinPence,
            oracleETHGBPinPence: 0,
            rentPaidInWei: rentSentInWei
        });
    }

}
```

Figure 5.6: Solidity function used by the tenant to pay rent.

5.3 Client-side Application

5.3.1 Framework and Libraries

As one of the user interface requirements discussed in section 3.3 was to ensure that the user never has to refresh the page for new updates to be reflected, a good choice for the front-end was the Meteor framework [51]. Meteor supports reactive programming featuring a dependency tracking system, therefore when certain data that is displayed on the UI changes, the relevant front-end templates are automatically re-rendered to reflect the change. For example, in the context of this application, when an event is received signifying that a rent payment has been made, the rent data on the client-side is updated, thus triggering the dependent UI templates to be re-rendered with the new rent amount. This ensures the user interface always displays the latest information.

Furthermore, Meteor is the recommended framework to use when building ĐApps [22], for a number of reasons including the ability to bundle all front-end code into single HTML and JavaScript files so that it can be hosted statically and uploaded to decentralised storage such as IPFS [43]. There is also an abundance of Meteor developer resources and frameworks specifically targeted at Ethereum, making development easier.

The Meteor packages used in the application are as follows:

- **ethereum:web3** [45]: Enables communication with the local Ethereum node via JSON remote procedure calls (RPC). The `web3.eth` object provides access to the Ethereum blockchain allowing information to be accessed such as the addresses of the user's active accounts, as well as transactions to be sent such as transferring ether or deploying new contracts to the blockchain.
- **iron:router** [47]: Provides client-side routing functionality to manage routes for the landlord, tenant, and arbitrator pages.
- **session** [49]: Used for storing variables that reactively update the UI when their value is changed. For example, the current ether to GBP exchange rate is stored in a session variable.
- **tracker** [50]: Provides functionality to manually set up reactive data sources.
- **http** [46]: Provides functionality for making HTTP requests on the application front-end. Used for making GET and POST requests to the contract server.
- **jquery** [48]: Used for many of the front-end animations, such as fading sections in and out to create a fluid user interface.

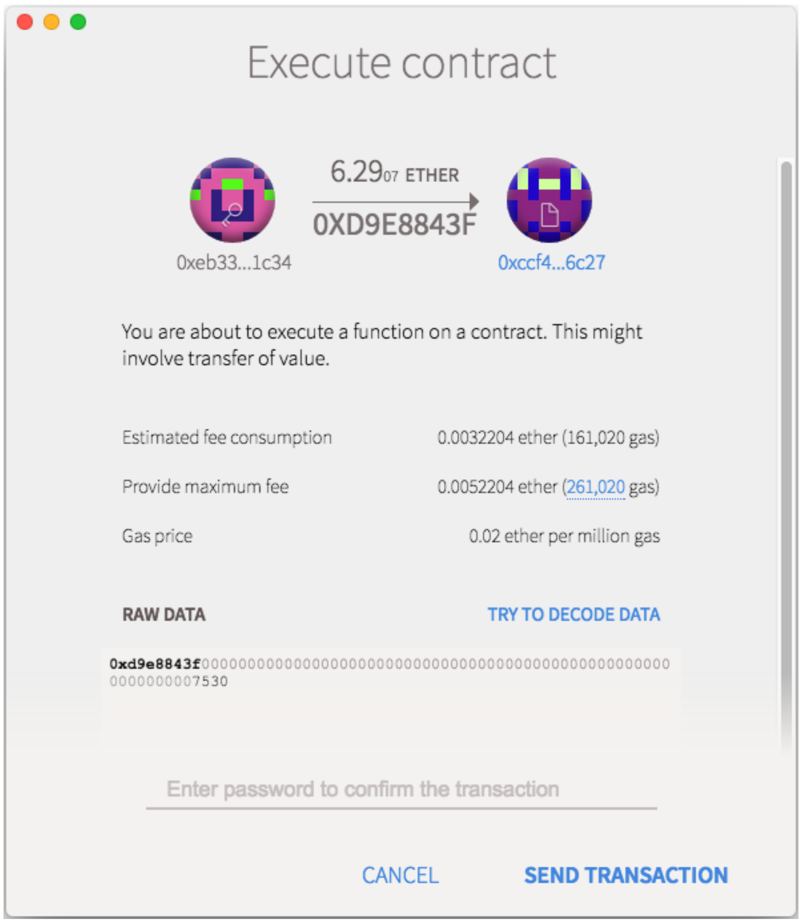
In addition, the following npm packages were used:

- **bootstrap (v4 alpha 6)** [58]: Provides the front-end styling for many components including forms, buttons and the navbar, as well as responsive flexbox features for grid alignment.
- **sha.js** [63]: Used to calculate SHA-256 hash values on the front-end including generating the hash of the contract clauses.
- **tether** [64]: The tether package is used alongside bootstrap to provide the animated tooltip functionality for showing the user more information when they hover over certain elements. One example of this is when the user hovers over their active account address, their balance in ether is displayed in a tooltip.

Finally, the fontawesome [32] JavaScript source code is linked in the HTML head section to provide the icons used on many buttons and components within the user interface.

5.3.2 Ethereum Integration

The JavaScript Web3 API enables the blockchain to be interacted with directly on the client-side of the application provided that there is an Ethereum node running locally. If the user is using the Mist browser, a local Ethereum node comes bundled with it, allowing them to send transactions to the network which will be picked up by the miners and added to the blockchain. The user is presented with a pop-up window before a transaction is sent, where they are able to enter their Ethereum account password to sign the transaction with their private key and deploy it to the network (see Figure 5.7). Sending transactions also requires some 'gas' to be sent as well to provide a fee for the miners to execute the transaction, which also means there is a delay before the block including the transaction is added to the chain (typically around 15 seconds for the Ethereum main network).


Execute contract

6.29 07 ETHER

0XD9E8843F

0xeb33...1c34

0xccf4...6c27

You are about to execute a function on a contract. This might involve transfer of value.

Estimated fee consumption	0.0032204 ether (161,020 gas)
Provide maximum fee	0.0052204 ether (261,020 gas)
Gas price	0.02 ether per million gas

RAW DATA

TRY TO DECODE DATA

```
0xd9e8843f00000000000000000000000000000000000000000000000000
00000000007530
```

Enter password to confirm the transaction

CANCEL

SEND TRANSACTION

Figure 5.7: Mist browser pop-up for transaction deployment confirmation.

Figure 5.8 shows a snippet of JavaScript code used to call the `signContract` method which will alter the state of the smart contract, hence requiring the user to enter their password in a pop-up similar to that shown in Figure 5.7. The variable `contractABI` is the contract's Application Binary Interface, which defines the low level specification for the contract. The variable `contractAddress` is the address of the contract stored on the blockchain.

```
web3.eth.contract(contractABI).at(contractAddress).signContract(
  {from: web3.eth.accounts[0]},
  function(error, result) {
    if (!error) {
      console.log("Contract signed OK");
    } else {
      console.warn("Error signing contract");
    }
  }
);
```

Figure 5.8: JavaScript Web3 code - signContract transaction deployment.

Figure 5.9 shows the code required to send a ‘call’ to the network which does not mutate the contract’s state. Hence this does not require gas to be sent and the method returns immediately with the value read from the blockchain.

The contract data displayed on the UI is retrieved via ‘calls’ to ensure that the user is seeing the true data stored on the blockchain, as opposed to displaying values stored in the database.

```
var contractInstance = web3.eth.contract(contractABI).at(contractAddress);
var numIssues = contractInstance.getIssueCount();
```

Figure 5.9: JavaScript Web3 code - getIssuesCount call.

To guarantee that the UI is updated whenever the data on the blockchain changes, events from the smart contract are fired whenever a user performs a transaction that mutates the contract state. Figure 5.10 shows the front-end JavaScript code that listens for `DepositPayment` events, with `depositPaidDep.changed()` (from the ‘Tracker’ package) being called on line 3 to signify that the dependency is stale (as some of the deposit has just been paid). Everywhere in the code that retrieves and displays data from the blockchain which depends on the amount of deposit paid calls `depositPaidDep.depend()`, setting up the reactive data dependency. Therefore whenever an event is received, all user interface templates that depend on this data will be automatically re-rendered.

```
1  var depositPaymentEvent = contractInstance.DepositPayment(function(error, result) {
2    if (!error) {
3      depositPaidDep.changed();
4      console.log("Deposit Payment Event Received: "
5        + web3.fromWei(result.args.amount, "ether") + " Ether");
6    }
7  });
```

Figure 5.10: JavaScript Web3 code - listening for a DepositPayment event.

5.3.3 Meteor Build Client and Decentralised Storage

The Meteor Build Client [35] developed by Fabian Vogelsteller, enables the client side of a Meteor project to be bundled into single HTML, CSS, and JavaScript files, along with any standalone public files such as images. This means the website is not required to be run from a Meteor server and can be served statically. Therefore these files can be uploaded to decentralised storage so they are not served from a single server which could go offline, improving the decentralisation of the application.

Acropolis has been uploaded to the InterPlanetary File System (IPFS) [43] so it can be accessed at `https://ipfs.io/ipfs/$SITE_HASH` due to the content-addressable nature of IPFS, where `$SITE_HASH` is the IPFS hash of the website directory. However, if the site content changes so will this hash value, and so the InterPlanetary Naming System (IPNS) is used to access the latest website hash via a peer ID which remains constant, such as `https://ipfs.io/ipns/$PEER_ID` [42]. A domain name can also be pointed at this peer ID to make all of the IPFS address resolution happen without users knowing IPFS is being used under the hood.

5.4 Server

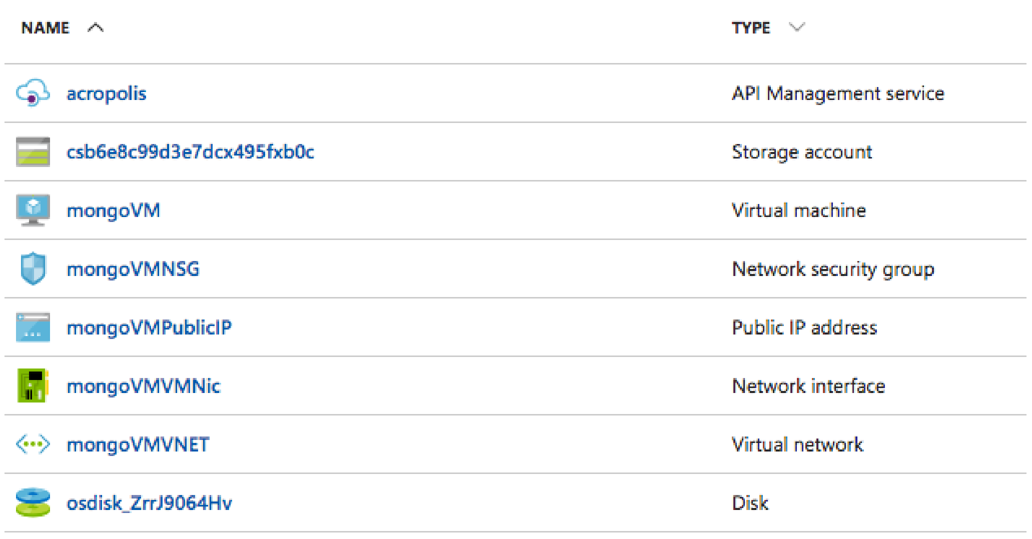
As was discussed in design section 3.2.1, I have decided to utilise a server to store the tenancy agreement clauses containing sensitive data, as they cannot be stored on the blockchain in plaintext. This section will discuss the implementation details of the cloud deployment, API, and the database used.

5.4.1 Deployment

The API server and database are deployed to a Microsoft Azure virtual machine running CentOS. The Azure subscription was provided free to use with this individual project and so it was the most cost-effective cloud solution. Azure provides many tools to make cloud deployment easier, and includes a GUI to easily edit options such as the inbound security rules and DNS name label for the VM. An overview of the deployment can also be observed on the Azure dashboard, which provides monitoring tools and historical metrics for virtual machine CPU usage and disk read/write operations per second. Figure 5.11 shows the items in the cloud deployment for this project.

5.4.2 API

The API server is built with Node.js [55] and Express [30] as these make it straightforward to get a simple server up and running, requiring relatively few lines of JavaScript code. I have also had experience from previous projects working with Node.js and so decided to use it again, having enjoyed using it in the past. An alternative would have been the Flask framework [31] which uses Python, however



NAME	TYPE
acropolis	API Management service
csb6e8c99d3e7dcx495fxb0c	Storage account
mongoVM	Virtual machine
mongoVMNSG	Network security group
mongoVMPublicIP	Public IP address
mongoVMVMNic	Network interface
mongoVMVNET	Virtual network
osdisk_ZrrJ9064Hv	Disk

Figure 5.11: Microsoft Azure VM deployment items, where 'mongoVM' is the name of the virtual machine where the MongoDB database is running.

I decided it was simpler to stick with JavaScript for both the client-side application and the server, enabling faster context switching between the two during development.

The following API GET and POST endpoints are active on the server (note the use of 'v1' in the path, enabling the API to be upgraded in the future and remain backwards compatible as new routes can simply be added with an incremented API version as necessary):

- **POST - /api/v1/users**
  To add a new user to the database via the sign up form. The user details are contained in the body of the POST request.

- **POST - /api/v1/login**
  Used to send the username and password to the server in the body of the POST request (via HTTPS) to log in to the application. The response sets a cookie in the user's browser.

- **GET - /api/v1/contracts/:type/:user**
  To retrieve the contracts relevant to the user of the application. Authentication occurs via a cookie sent with the request. 'type' is the role of the user (landlord, tenant, or arbitrator), and 'user' is the user's username.

- **POST - /api/v1/contracts/:type/:user**
  To add a new contract to the database containing the details which are not stored on the blockchain such as the contract clauses and the names of the contract participants. 'type' is the role of the user (landlord, tenant, or arbitrator), and 'user' is the user's username.

- **POST - /api/v1/issues/:contractAddr**
  To add a new issue for a given contract to the database. 'contractAddr' is the Ethereum address of the contract.

- **POST - /api/v1/notices/:contractAddr**
  To add a new notice for a given contract to the database. 'contractAddr' is the Ethereum address of the contract.

The following npm libraries are used with the server:

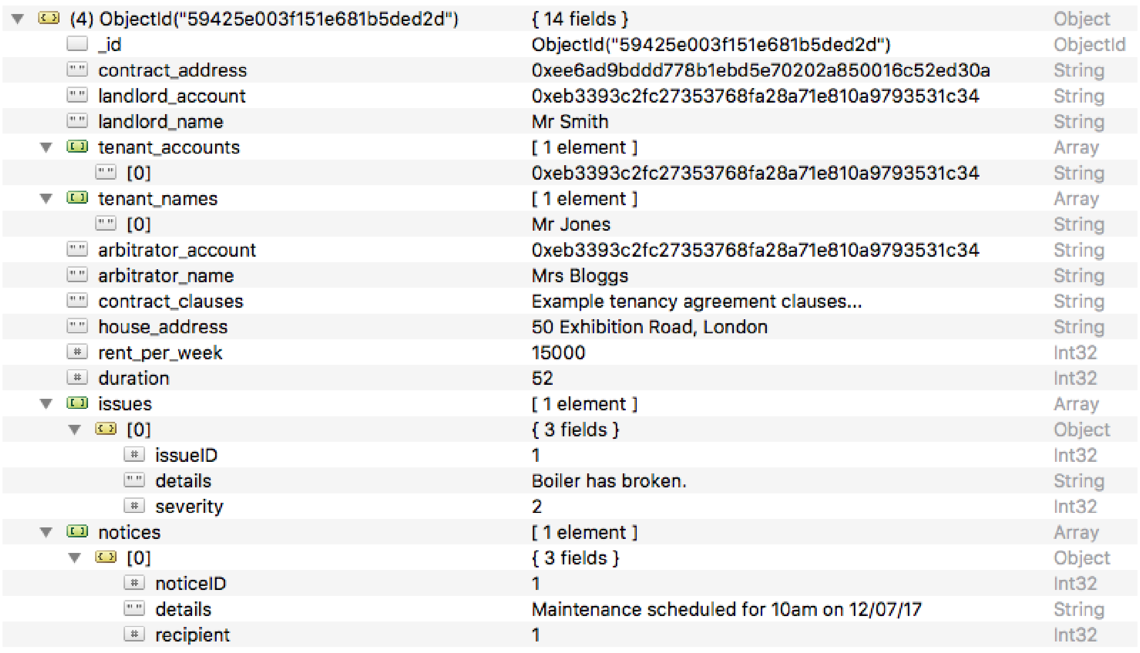
- **bcryptjs** [56]: Used to generate the combined hash and salt stored in the database from the user passwords. BCrypt is a computationally intensive hashing function which aims to slow down brute-force search attacks.
- **express** [60]: High performance web framework for Node.js.
- **body-parser** [57]: Body parsing middleware for parsing JSON bodies of received POST requests.
- **cookie-session** [59]: Used for setting cookies in response bodies with the 'Set-Cookie' header.
- **mongodb** [62]: MongoDB driver for Node.js to interact with the database.
- **https** [61]: Used to set up the HTTPS server.

5.4.3 MongoDB

I chose to use MongoDB [86] as the database for this application because of its flexible document-based storage model. It streamlines the development process with its lack of restrictions on the format of the documents inserted, as the data to be stored in the database is not confirmed at the start of development and is constantly evolving. However, this does mean there is more manual validation that needs to be performed on both document insertion and retrieval, as both data type and existence for a field are not enforced.

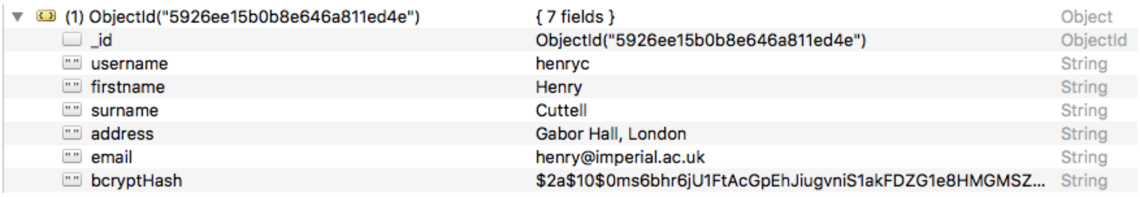
MongoDB also has several frameworks which make it very simple to integrate with a Node.js server. Potentially in the future, the MongoDB database could be switched out for a relational database such as PostgreSQL [69] which enforces a more rigid schema. However, at this stage in prototyping the application, MongoDB is perfectly suited when the application is frequently evolving.

Figure 5.12 shows an example contract document stored in MongoDB, being viewed with the Robo 3T tool [72]. In addition, an example user document is shown in Figure 5.13.



(4) ObjectId("59425e003f151e681b5ded2d")	{ 14 fields }	Object
_id	ObjectId("59425e003f151e681b5ded2d")	ObjectId
contract_address	0xee6ad9bddd778b1ebd5e70202a850016c52ed30a	String
landlord_account	0xeb3393c2fc27353768fa28a71e810a9793531c34	String
landlord_name	Mr Smith	String
tenant_accounts	[1 element]	Array
[0]	0xeb3393c2fc27353768fa28a71e810a9793531c34	String
tenant_names	[1 element]	Array
[0]	Mr Jones	String
arbitrator_account	0xeb3393c2fc27353768fa28a71e810a9793531c34	String
arbitrator_name	Mrs Bloggs	String
contract_clauses	Example tenancy agreement clauses...	String
house_address	50 Exhibition Road, London	String
rent_per_week	15000	Int32
duration	52	Int32
issues	[1 element]	Array
[0]	{ 3 fields }	Object
issueID	1	Int32
details	Boiler has broken.	String
severity	2	Int32
notices	[1 element]	Array
[0]	{ 3 fields }	Object
noticeID	1	Int32
details	Maintenance scheduled for 10am on 12/07/17	String
recipient	1	Int32

Figure 5.12: Screenshot of an example contract document stored in MongoDB being viewed through the Robo 3T tool [72].



(1) ObjectId("5926ee15b0b8e646a811ed4e")	{ 7 fields }	Object
_id	ObjectId("5926ee15b0b8e646a811ed4e")	ObjectId
username	henryc	String
firstname	Henry	String
surname	Cuttell	String
address	Gabor Hall, London	String
email	henry@imperial.ac.uk	String
bcryptHash	$2a$10$0ms6bhr6jU1FtAcGpEhJiugvniS1akFDZG1e8HMGMSZ...	String

Figure 5.13: Screenshot of an example user document stored in MongoDB being viewed through the Robo 3T tool [72].

Chapter 6

Evaluation

6.1 Industry Feedback

In order to receive feedback for the project and evaluate its technical achievement, I demonstrated my application to several CEOs/co-founders of companies in the blockchain space including Blockchain, Aventus, and Oraclize. I also presented the application at the Imperial College London Blockchain Forum June event to receive some wider feedback, and demonstrated it to a property lettings consultant at Woodward Estate Agents.

6.1.1 Blockchain

Midway through the development process I presented the project to the co-founders and team at Blockchain, a leading provider of online Bitcoin wallets and blockchain explorer software [7]. Their feedback was very positive overall and they said the project provided a good solution to the issues associated with current paper-based tenancy agreements. One concern was raised with the rental payments being made in ether with a fixed amount of the currency due for the entirety of the contract. Tenants would end up having to pay vastly different equivalent amounts in GBP each month due to the extremely volatile nature of the cryptocurrency market. This issue is discussed in more detail in section 3.1.3 and the solution of using an oracle service to retrieve the current ether to GBP exchange rate for each payment was implemented.

6.1.2 Aventus

Upon application development completion I spoke to Alan Vey, Co-Founder and Director of Aventus, a blockchain-based event ticketing startup aiming to eliminate unregulated touting [2]. I demonstrated my project to him and he gave me some feedback and invaluable advice from his experience developing Ethereum applications. He suggested some ways to fine-tune my smart contract implementation, which I have taken on board, as well as some ideas for potential future monetisation of the project. He really liked the project concept and said the application has a solid user interface which is clear and easy to navigate. Alan also said the project "has real potential to reduce the extortionate administration fees which are the norm when renting a property".

6.1.3 Oraclize

As I am using the blockchain oracle service provided by Oraclize [65] in this project to obtain the current price of ether in GBP, I decided it would be beneficial to receive feedback from the company for my web application and in particular its integration with Oraclize. Therefore I visited the Oraclize London office to demonstrate the project to Thomas Bertani, the CEO and Founder, as well as the Oraclize team. They were impressed with the project and agreed with the importance of the payments being paid in ether corresponding to a fixed GBP amount for the tenancy. I also gave them a technical explanation of how the smart contract code is implemented and they advised me on some ways to improve the structure of the code in order to reduce the amount of ether required for the initial contract deployment to the blockchain. We also evaluated the overall application design and discussed a theoretical way to enable the confidential tenancy agreement information to be encrypted and stored on decentralised storage in the future, as the technique is not currently possible to implement due to Ethereum node API restrictions.

6.1.4 Palantir Cryptocurrency Mini Hackathon

I was invited by a small group of Software Engineers from Palantir Technologies to give a talk about my project at a mini hackathon they organised to build some novel blockchain-based applications. I demonstrated the functionality of the application and discussed how it utilises the Ethereum network, which was followed by an in-depth discussion about the cryptocurrency ecosystem. They were very positive of the application and praised its level of completeness, in particular the user interface. One piece of advice they gave was about my presentation, which they said needed a brief background introduction of Ethereum and outline the benefits it brings to the application over just using a regular server as a back-end. This is something I had not realised was necessary up until now, as all of the companies I had previously presented to specialise in blockchain technology.

To make the application appeal to a wider audience they also suggested I display every price in ether with an associated value converted to GBP (I had done this for the rental payments but not for the deposit amounts at this stage). They also said in future development work it would be beneficial to completely abstract the use of ether from the application and implement GBP bank payments straight into the website with automatic ether conversion taking place. This is certainly the next big step to take during development in the future and by abstracting the Ethereum accounts, the option for automatic GBP payments could be achieved, though great care would need to be taken to maintain high security.

6.1.5 Woodward Estate Agents

At the end of the project I believed it was important to go back and visit Gary Feger, the Property Letting Consultant who gave me the initial project requirements and advice (detailed in section 2.5) [1]. I gave him a demonstration of the application and explained the various technicalities of using the Ethereum network for smart tenancy agreements, in particular for rent payments and holding the deposit. Gary was really impressed with the project outcome and agreed that this is the way contracts will be interacted with and signed in the future. He praised the simplicity of the user interface and stressed how important it is for the system to be easy to use for users of all technical abilities.

Gary did have some concerns regarding the legality of storing the deposit on the blockchain, as currently it is required to be held by one of the approved schemes by UK law, as discussed in section 2.5. The aim of this project is to provide a prototype of a system which could have the legal restrictions lifted as blockchain becomes more mainstream and UK law catches up with the technology. In Arizona State, USA, a bill was passed in March 2017 making blockchain-based signatures valid electronic signatures, and declared that a “contract relating to a transaction may not be denied legal effect, validity or enforceability solely because that contract contains a smart contract term” [44]. This shows that laws are already changing to adopt this technology and so the legal framework required for a blockchain-based tenancy agreement may be on the horizon.

In addition, Gary suggested some very good ideas to further improve the application which I will detail as follows:

- Ability for the user to input their meter readings when they move in, to be stored on the blockchain as proof. It would also be useful if the tenant could upload a photo as proof, with the hash of the photo stored on the blockchain as irrevocable evidence.
- Text alerts to the landlord when the tenants transfer rent to the smart contract, alerting them that it can be withdrawn. There could also be text alerts to tell the tenant when rent is due.
- The arbitrator tab should be split into two sections: one for the arbitrator and one for the property manager. This makes the separation of roles clearer, with the property manager in charge of resolving issue disputes between the landlord and tenant, and the arbitrator in charge of the final deposit resolution if a dispute arises.
- There should also be an option to upload an image along with the issue description as evidence, as well as to give the landlord an idea of the scale of the problem (something which can often be exaggerated by tenants).

As I have reached the end of application development, I will continue to develop the application and incorporate these ideas in the future.

6.1.6 Imperial College London Blockchain Forum

On the 15th June 2017, I presented my project at the Imperial College London Blockchain Forum to a lecture theatre filled with attendees from a wide range of backgrounds including computer science, mathematics, engineering, and law. I followed a presentation by Iain Stewart which gave an overview of the different types of cryptocurrencies, discussing how blockchains can interact with the real world and introduced the concept of smart contracts. I subsequently discussed the problem my project is trying to solve, explained the smart contract state transitions, and gave a demonstration of the application, showing a rent payment being made and an issue reported.

After the event, I spoke with several attendees to hear their thoughts on the concept and application. The feedback was very positive and people could see the value of the application and the benefits that blockchain technology brings. I spoke with Gabrielle Patrick, a Cryptocurrency lawyer, who also spoke at the Blockchain Forum, and she agreed that in order for laws to evolve to accept blockchain technology, there must first be products built to drive this change. Having discussed the legal issues with storing a deposit on the blockchain with Gary Feger (see section 6.1.5), it is re-assuring to know that developing the application first to show there is a need for the law to change is a good approach. I have also received several emails following the talk to arrange meetings to discuss the possibility of turning the project into a business.

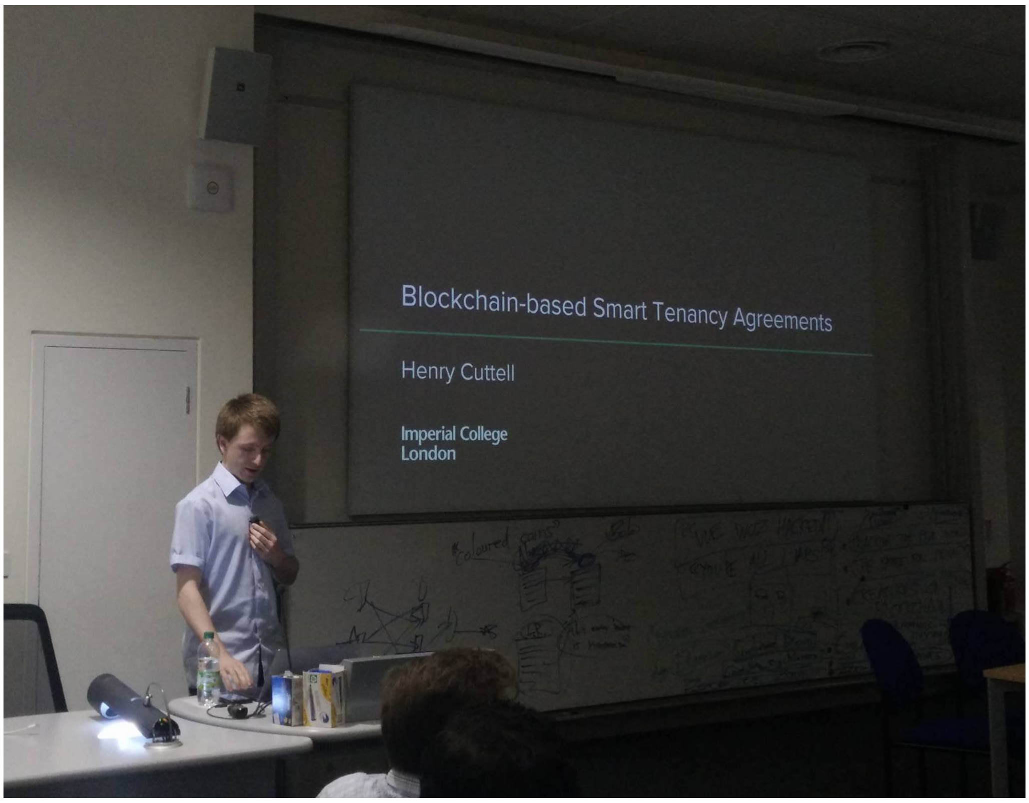


Figure 6.1: Photo from my talk at the Imperial College London Blockchain Forum on the 15th June 2017.

6.2 User Feedback and Survey

In order to gain an idea of how users get on with using the application themselves, I went to an Imperial College London Department of Computing lab demonstration session to let fellow students try out the application. I devised a Google form [41] with questions for them to answer after they had used the application to complete tasks such as signing a tenancy agreement and making a rent payment. I let them navigate the application on their own and provided minimal instructions after telling them an action they needed to complete, so I could see how easy to use it is for someone who has never seen the website before. I asked every user that tried out the application whether they knew much about Ethereum, and they generally had heard of it but didn't know much about the technology. This is useful because I wanted to find out which aspects of the application are confusing for people without much knowledge of Ethereum, as it is likely that the majority of future users will fall into this category.

The questionnaire included the following questions or statements (numbers 1 - 4). In the case of a statement, the possible answers range from 'strongly agree' to 'strongly disagree'. I also added an explanation question for each question where the user could disagree in order for them to provide some feedback as to what could be improved. The full results are displayed in Appendix A including response proportion pie charts generated automatically by Google forms.

0. Have you ever rented a property before and signed a tenancy agreement?
1. The home page of the website makes the purpose of the application clear.
2. The application is easy to navigate.
3. It is easy to see the stage that the tenancy agreement is in and it is obvious which action needs completing next.
4. The benefits of using the Ethereum network are clear.
5. Would you be interested in using the application to sign tenancy agreements?
6. Would you be interested in using the application to pay your rent in ether?
7. Do you have any general feedback to give about the application? Such as things you liked or didn't like?

Nine out of ten of the users had signed a tenancy agreement before, and so were able to compare the application to their experiences with the current paper-based system. There was a fairly mixed response to question 1, with one user suggesting that a more in depth explanation of the application's purpose would be beneficial. I will address this during the next development iteration and will add an 'about' section to the website to give a thorough explanation of the site's function, which would help potential users gain a better understanding of the application before committing to using it.

Questions 2 and 3 were answered very favourably with all users strongly agreeing or agreeing with the statements. I believe the application has succeeded in meeting the goal of being easy to navigate and clearly directs the user to the next action they must complete.

Question 4 highlighted to me that perhaps the application does not explain clearly enough the benefits that using Ethereum brings to users. This question was perhaps slightly misworded, as the benefits of using the application are effectively side-effects of utilising the Ethereum blockchain, such as the swift signing of contracts with irrevocable proof of the signature. As it should not be required for users to know a great deal about Ethereum, apart from being about to purchase ether and have a rough idea of how it is used in the application, the decision statement could be changed to ‘the benefits of using the application are clear’ in future user surveys.

Question 5 was answered very favourably with 80% of users surveyed being interested in using the application to sign tenancy agreements.

Question 6 had a very mixed response, with 4 in 10 users interested in paying their rent in ether, with 3 responding that they might be interested and the remaining 3 saying that they would not be interested. In the feedback section for question 6, they cited reasons why they did not want to use ether as down to currency instability and their lack of knowledge about ether. Hopefully this will become less of an issue as time goes on, when Ethereum will become more widely known about by the general public, as well as ideally the currency would become less volatile. A way to combat this would be to build a layer on top of the Ethereum account system, by managing users’ private keys, abstracting the use of ether away and presenting them directly with GBP payment options. This is of course less secure and more centralised than the current system, but could lead to increased adoption.

For the final question users responded favourably about the application’s user interface, as well as giving some further suggestions of features that would improve the front-end design and functionality. For example, the ability to change the displayed currency from GBP to USD would be crucial if the application is launched outside of England and Wales. Another suggestion was for the user to be able to pay bills such as water and electricity through the application. Further feedback comments can be seen on the questionnaire results in Appendix A. In future, I would like to widen the user test base and get feedback from a larger variety of demographics.

6.3 Strengths and Weaknesses

Overall, this project has aimed to provide a glimpse into the future where it is routine for tenants and landlords to transact over the blockchain; it is not to produce a solution that is totally compliant with the laws of today. The project has demonstrated that it is possible to create an application that streamlines interactions with a legal contract, and is secure and simple to use for participants using Ethereum accounts. Feedback has shown that people can see the benefit of encoding these contracts on the blockchain and would be willing to use the application for signing their contracts and interacting with the agreement. This has the potential to reduce our reliance on paper-based contracts, speeding up multiple aspects of the overall process.

The strength of the Acropolis application, as stressed in the industry presentations and user feedback, lies with its responsive and clear user interface. I believe it has succeeded in simplifying the user journey and making it as easy-to-use as possible for non-technical users. The smart contract code has proven to be robust, but I would like to have a formal security audit completed in future with the source code published, allowing anyone using the application to be able inspect the code themself if they wish.

I have not had time to ensure that the front-end and API server are as thoroughly tested as I would have liked, but this is something that I will focus on in the future. It would be beneficial to have the server undergo penetration testing before being released in a production environment. Furthermore, while I have tried to keep the code as modular in design as possible, there are a few sections which would benefit from some refactoring.

The system has been designed to maximise security and decentralisation, but this means usability has suffered in some places such as the requirement for manual rent payments (discussed in section 3.1.3). The possibility for payments to be made in GBP is discussed in the future work section 7.2 and could also be combined with automatic payments, reducing the decentralisation but improving the usability of the application. This trade-off will be re-evaluated in the future, but I am confident a good middle ground can be found.

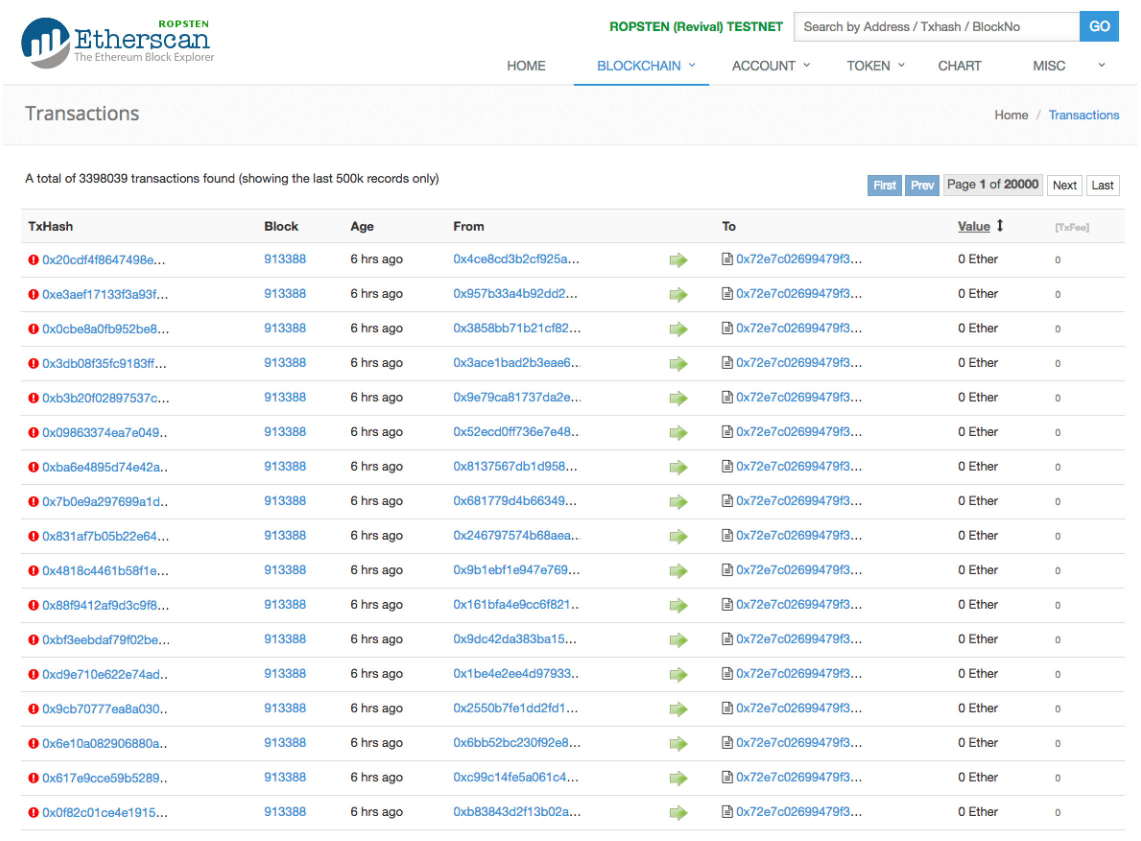
Lastly, as users will hold ether before they pay rent or before the landlord cashes out the ether for GBP, it is subject to currency volatility. As was seen from the user survey, this is a barrier to entry for users unwilling to purchase ether for this reason, therefore the automatic GBP conversions mentioned above should resolve this weakness.

6.4 Project Challenges

Development of this project has led to certain challenges that I have had to overcome. I will describe these in the following subsections and detail the steps I took to overcome them.

6.4.1 Testing Transactions on the Ethereum Testnet

I found testing transactions on various forms of Ethereum test networks to be problematic when using the Mist browser, which is one of the main ways I am intending for users to interact with the Acropolis application. I began by trying to get TestRPC working with Mist, but found an open issue on GitHub to be the cause of it not working [38].



A total of 3398039 transactions found (showing the last 500k records only)

TxHash	Block	Age	From		To	Value	[TxFee]
0x20cdf4f8647498e...	913388	6 hrs ago	0x4ce8cd3b2cf925a...		0x72e7c02699479f3...	0 Ether	0
0xe3aef17133f3a93f...	913388	6 hrs ago	0x957b33a4b92dd2...		0x72e7c02699479f3...	0 Ether	0
0x0cbe8a0fb952be8...	913388	6 hrs ago	0x3858bb71b21cf82...		0x72e7c02699479f3...	0 Ether	0
0x3db08f35fc9183ff...	913388	6 hrs ago	0x3ace1bad2b3eae6...		0x72e7c02699479f3...	0 Ether	0
0xb3b20f02897537c...	913388	6 hrs ago	0x9e79ca81737da2e...		0x72e7c02699479f3...	0 Ether	0
0x09863374ea7e049..	913388	6 hrs ago	0x52ecd0ff736e7e48..		0x72e7c02699479f3...	0 Ether	0
0xba6e4895d74e42a..	913388	6 hrs ago	0x8137567db1d958...		0x72e7c02699479f3...	0 Ether	0
0x7b0e9a297699a1d..	913388	6 hrs ago	0x681779d4b66349...		0x72e7c02699479f3...	0 Ether	0
0x831af7b05b22e64...	913388	6 hrs ago	0x246797574b68aea..		0x72e7c02699479f3...	0 Ether	0
0x4818c4461b58f1e...	913388	6 hrs ago	0x9b1ebf1e947e769...		0x72e7c02699479f3...	0 Ether	0
0x88f9412af9d3c9f8...	913388	6 hrs ago	0x161bfa4e9cc6f821..		0x72e7c02699479f3...	0 Ether	0
0xbf3eebdaf79f02be...	913388	6 hrs ago	0x9dc42da383ba15...		0x72e7c02699479f3...	0 Ether	0
0xd9e710e622e74ad..	913388	6 hrs ago	0x1be4e2ee4d97933..		0x72e7c02699479f3...	0 Ether	0
0x9cb70777ea8a030..	913388	6 hrs ago	0x2550b7fe1dd2fd1...		0x72e7c02699479f3...	0 Ether	0
0x6e10a082906880a..	913388	6 hrs ago	0x6bb52bc230f92e8...		0x72e7c02699479f3...	0 Ether	0
0x617e9cce59b5289..	913388	6 hrs ago	0xc99c14fe5a061c4...		0x72e7c02699479f3...	0 Ether	0
0x0f82c01ce4e1915...	913388	6 hrs ago	0xb83843d2f13b02a...		0x72e7c02699479f3...	0 Ether	0

Figure 6.2: Screenshot of the latest transactions on the Ropsten test network following a spam attack on the 12th May 2017 from Etherscan [29].

I then decided to use the Ropsten testnet, a network practically identical to the Ethereum main network using the proof-of-work system, but the ether tokens transacted on it have no monetary value. This was advantageous because I could be sure my application would also work when switched to using the main network. However, in late February, the Ropsten network suffered a denial-of-service attack, where spam blocks were inserted into the blockchain which were computationally expensive for nodes to process. This made testing at this time impossible, but was later fixed by a donation of GPU power to mine a new heaviest chain which had a higher difficulty than the spammed chain, so nodes switched to this branch, thus resolving the issue [40].

Unfortunately this happened again on the 12th May 2017, where the Ropsten testnet suffered another spam attack (see Figure 6.2). The red warning symbol next to each transaction hash represents an 'out of gas' error, with the most recent transaction occurring 6 hours before the time of this screenshot. Hence this, combined with the fact that blocks were generally taking a proportionally long time to be mined (often over a minute) during regular use on Ropsten, led me to look for other testing solutions.

I subsequently switched to the Rinkeby 'proof-of-authority' test network [39]. This testnet restricts distribution of ether funds via a faucet [71], where it is possible to request a certain amount of ether per day, and prevents the attacks that plagued Ropsten due to its clique proof-of-authority consensus protocol. So far this has proven to be an effective way of testing transactions with a reliable 15 second block time interval.

6.4.2 Breaching the Gas Limit for Contract Deployment

After adding the Oraclize service code to the smart contract, the gas deployment cost raised considerably and became greater than the 4.7 million gas limit in place on the testnet. A quick fix for this was to temporarily remove some of the contract functions which were infrequently used, so I was still able to continue deploying new contracts, however this was not a permanent fix for the problem. I made several optimisations but the gas cost was still too high despite this, as all of the logic is contained within the single monolithic smart contract.

The current solution I have gone with is by editing the inherited code from the ‘usingOraclize’ contract and removing functions that the smart tenancy agreement does not currently use. This has successfully reduced the gas deployment cost to below the gas limit, but it still remains high. It also raises the problem of maintainability of the code, as if the Oraclize team change the code of ‘usingOraclize’ on their GitHub, my smart contract will not automatically inherit this new code on new deployments without manually editing the cut-down version of their contract. However, when I demonstrated my project to the Oraclize team, they told me that they were looking into transitioning their code to a Solidity library [77], which would solve this issue in future as library code is pre-deployed to the blockchain, therefore saving deployment gas. In future work, I will also investigate the viability of splitting up the core smart tenancy agreement code with sections being delegated to library code.

Chapter 7

Conclusion

7.1 Lessons Learned

This project has been an exciting opportunity to dive into the cryptocurrency ecosystem and in particular explore the applications of smart contracts. Blockchain technology is a rapidly evolving field, with huge investments currently being made by companies of all sizes, who see the applications to be particularly disruptive in their respective industries. This project has demonstrated the possibility of applying the power of smart contracts to a predominantly paper-based form of contract, with many of the tenancy agreement interactions being possible to transact in a swift and secure fashion over the blockchain.

Smart contracts not only provide a way to sign the contracts, but they enable a vast array of conditions to be encoded to deal with many eventualities that would otherwise require the use of a trusted third party. A good example of this is storing the tenancy deposit in the smart contract to ensure it cannot be spent by anyone until the tenancy is over and the release conditions satisfied. As this technology simply did not exist a few years ago, it is natural that the current legislation does not allow tenancy deposits to be stored on the blockchain. However, as these smart contract applications become more widespread, laws will recognise this and be updated accordingly, provided that the technology continues to be universally trusted.

As cryptocurrencies become more mainstream, it is likely that users will be more willing to transact in ether in the future, but at present it is an obstacle likely to prevent mass adoption. Applications built on Ethereum will need to have better support for GBP interaction and have seamless currency conversion mechanisms to speed up user adoption.

Smart contracts are here to stay and I believe innovative use cases for transferring paper-based contracts to the blockchain will continue to be realised in the coming years.

7.2 Future Work

Payments and Withdrawals in GBP

Integrating payments in GBP is definitely the next step in developing this application and is one of the main areas that was identified in the application user feedback questionnaire. Developing a front end which accepts GBP payments, perhaps with a similar interface to BitPay [6], but with payments in GBP instead of Bitcoin and sending equivalent funds to a specified Ethereum address. This would of course reduce the decentralisation of the application however, and so careful thought would need to be given to maximise the security of this approach, as well as ways to maintain a high level of trust with the end users.

The same applies to the landlord receiving the rent payments, where an Ethereum account management system could be developed which would automatically withdraw the ether paid to the smart contract if it is given access to their account private key. This ether could then be automatically transferred to their bank account, again using a third party ether to GBP conversion service, or by developing one from scratch to be used specifically with this application.

Further Smart Contract Functionality

Future work could also integrate additional smart contract functionality. This will be easier when sections of the smart contract are abstracted into libraries to reduce the initial contract deployment cost, and will mean more complex features will be able to be added whilst staying within the gas limit.

Some possible additions are as follows:

- **Penalties for late rent payment**: There could be contract terms which automatically deduct funds from the initial deposit paid if a rent payment is made late. It would have to be investigated how much ether is appropriate to deduct from the deposit, with either a single amount deducted if rent is paid late, or an incremental system could be used where ether is deducted for every subsequent day when overdue rent is not paid.
- **Tenant payment split ratio**: A feature which would benefit tenants using the system would be to encode the rent split ratio between the tenants in the smart contract (if there is more than one) where the application would show them the exact split they need to pay individually. This is useful if they are each paying unequal shares, for example if some tenants have larger bedrooms and have agreed to pay a higher share of the rent.
- **Integrate property inventory functionality**: This would allow the tenants to input the inventory when they move in, with it being stored on the blockchain as proof. It could also include logging defects with the property at the start of the tenancy.

Support for Paying Bills

As was mentioned in the user questionnaire feedback, a future addition could be to integrate bill payments into a tab on the application dashboard. This is more complex to achieve as it would mean either the utility companies would need to accept ether payments, or further infrastructure would be required to make payments in GBP when ether is paid to a certain address owned by the application. We are likely some time away from utility companies accepting ether for bill payments so this option is unlikely to be feasible in the foreseeable future. Creating infrastructure to handle ether to GBP payments to the utility companies would require some substantial engineering to ensure the system is robust and delivers the correct amounts in GBP to the correct company accounts. I will continue to investigate the feasibility of this feature in the future.

Bibliography

[1] About Woodward Estate Agents. http://www.woodward.co.uk/about/. Accessed: 11th June 2017.

[2] Aventus. https://aventus.io/. Accessed: 31st May 2017.

[3] Barclays: How long do payments from my account take to clear? https://www.help.barclays.co.uk/faq/payments/payment-information/clear-from-account.html. Accessed: 24th Jan. 2017.

[4] Bitcoin developer guide. https://bitcoin.org/en/developer-guide. Accessed: 25th Jan. 2017.

[5] Bitcoin.org - choosing your bitcoin wallet. https://bitcoin.org/en/choose-your-wallet. Accessed: 25th Jan. 2017.

[6] BitPay. https://bitpay.com/. Accessed: 17th June 2017.

[7] Blockchain.com. https://www.blockchain.com/. Accessed: 27th May 2017.

[8] Boxy svg - scalable vector graphics editor. https://boxy-svg.com/main.html#about. Accessed: 4th June 2017.

[9] Coinbase.com. https://www.coinbase.com/. Accessed: 31st May 2017.

[10] Crypto-currency market capitalizations. https://coinmarketcap.com/. Accessed: 25th Jan. 2017 and 29th May 2017.

[11] Cryptocompare API documentation. https://www.cryptocompare.com/api/. Accessed: 2nd June 2017.

[12] Cryptocompare ether to gbp overview. https://www.cryptocompare.com/coins/eth/overview/GBP. Accessed: 3rd June 2017.

[13] Deposit Protection Service (DPS). https://www.depositprotection.com/. Accessed: 31st May 2017.

[14] Ethereum documentation - account types, gas, and transactions. http://ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html. Accessed: 17th Jan. 2017.

[15] Ethereum documentation - clients. http://www.ethdocs.org/en/latest/ethereum-clients/choosing-a-client.html. Accessed: 22nd Jan. 2017.

[16] Ethereum documentation - mining. http://www.ethdocs.org/en/latest/mining.html. Accessed: 22nd Jan. 2017.

[17] Ethereum network statistics. https://ethstats.net/. Accessed: 21st Jan. 2017.

[18] Ethereum official website. https://www.ethereum.org/. Accessed: 17th Jan. 2017.

[19] Ethereum official website - ether. https://www.ethereum.org/ether. Accessed: 17th Jan. 2017.

[20] Ethereum official website - smart contract tutorial. https://www.ethereum.org/greeter. Accessed: 23rd Jan. 2017.

[21] Ethereum wiki - block protocol 2.0. https://github.com/ethereum/wiki/wiki/Block-Protocol-2.0. Accessed: 21st Jan. 2017.

[22] Ethereum wiki - Dapp using Meteor. https://github.com/ethereum/wiki/wiki/Dapp-using-Meteor. Accessed: 12th June 2017.

[23] Ethereum wiki - JSON RPC API. https://github.com/ethereum/wiki/wiki/JSON-RPC. Accessed: 22nd Jan. 2017.

[24] Ethereum wiki - merkle patricia tree specification. https://github.com/ethereum/wiki/wiki/Patricia-Tree. Accessed: 22nd Jan. 2017.

[25] Ethereum wiki - proof of stake FAQ. https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ. Accessed: 22nd Jan. 2017.

[26] Ethereum wiki - serpent. https://github.com/ethereum/wiki/wiki/Serpent. Accessed: 9th June 2017.

[27] Ethereum wiki - web3 javascript DApp API. https://github.com/ethereum/wiki/wiki/JavaScript-API. Accessed: 22nd Jan. 2017.

[28] Ethermine mining pool. https://ethermine.org/. Accessed: 21st Jan. 2017.

[29] Etherscan: The ethereum block explorer. https://etherscan.io/. Accessed: 30th May 2017.

[30] Express framework. https://expressjs.com/. Accessed: 14th June 2017.

[31] Flask framework. http://flask.pocoo.org/. Accessed: 14th June 2017.

[32] fontawesome icon toolkit. http://fontawesome.io/. Accessed: 12th June 2017.

[33] Foxtons fees. https://www.foxtons.co.uk/help/fees/. Accessed: 24th Jan. 2017.

[34] GitHub - ethereum viper language. https://github.com/ethereum/viper. Accessed: 9th June 2017.

[35] GitHub - Meteor Build Client. Accessed: 13th June 2017.

[36] Github - mist browser. https://github.com/ethereum/mist. Accessed: 22nd Jan. 2017.

[37] GitHub - oraclize API. https://github.com/oraclize/ethereum-api. Accessed: 11th June 2017.

[38] GitHub - testrpc issue. https://github.com/ethereumjs/testrpc/issues/236. Accessed: 15th June 2017.

[39] GitHub ethereum - rinkeby poa testnet. https://github.com/ethereum/EIPs/issues/225. Accessed: 15th June 2017.

[40] GitHub Ethereum - ropsten revival. https://github.com/ethereum/ropsten/blob/master/revival.md. Accessed: 15th June 2017.

[41] Google forms - about. https://www.google.co.uk/forms/about/. Accessed: 15th June 2017.

[42] ipfs for websites. https://ipfs.io/ipfs/QmNZiPk974vDsPmQii3YbrMKfi12KTSNM7XMiYyiea4VYZ/example#/ipfs/QmP8WUPq2braGQ8iZjJ6w9di6mzgoTWyRLayrMRjjDoyGr/websites/README.md. Accessed: 13th June 2017.

[43] Ipfs is the distributed web. https://ipfs.io/. Accessed: 12th June 2017.

[44] Legiscan: Bill text: Az hb2417 | 2017 | fifty-third legislature 1st regular | chaptered. https://legiscan.com/AZ/text/HB2417/id/1588180. Accessed: 11th June 2017.

[45] Meteor atmosphere - ethereum:web3. https://atmospherejs.com/ethereum/web3. Accessed: 12th June 2017.

[46] Meteor atmosphere - http. https://atmospherejs.com/meteor/http. Accessed: 12th June 2017.

[47] Meteor atmosphere - iron:router. https://atmospherejs.com/iron/router. Accessed: 12th June 2017.

[48] Meteor atmosphere - jquery. https://atmospherejs.com/meteor/jquery. Accessed: 12th June 2017.

[49] Meteor atmosphere - session. https://atmospherejs.com/meteor/session. Accessed: 12th June 2017.

[50] Meteor atmosphere - tracker. https://atmospherejs.com/meteor/tracker. Accessed: 12th June 2017.

[51] Meteor documentation. http://docs.meteor.com/#/full/. Accessed: 12th June 2017.

[52] Midasium: The blockchain of real estate. http://midasium.com/. Accessed: 23rd Jan. 2017.

[53] Midasium: The blockchain of real estate - smart contracts. http://midasium.com/smart-contracts. Accessed: 23rd Jan. 2017.

[54] MyDeposits. https://www.mydeposits.co.uk/. Accessed: 31st May 2017.

[55] Node.js - about. https://nodejs.org/en/about/. Accessed: 14th June 2017.

[56] npm - bcryptjs. https://www.npmjs.com/package/bcryptjs. Accessed: 14th June 2017.

[57] npm - bodyparser. https://www.npmjs.com/package/body-parser. Accessed: 14th June 2017.

[58] npm - bootstrap. https://www.npmjs.com/package/bootstrap. Accessed: 12th June 2017.

[59] npm - cookiesession. https://www.npmjs.com/package/cookie-session. Accessed: 14th June 2017.

[60] npm - express. https://www.npmjs.com/package/express. Accessed: 14th June 2017.

[61] npm - https. https://www.npmjs.com/package/https. Accessed: 14th June 2017.

[62] npm - mongodb. https://www.npmjs.com/package/mongodb. Accessed: 14th June 2017.

[63] npm - sha.js. https://www.npmjs.com/package/sha.js. Accessed: 12th June 2017.

[64] npm - tether. https://www.npmjs.com/package/tether. Accessed: 12th June 2017.

[65] Oraclize. http://www.oraclize.it/. Accessed: 2nd June 2017.

[66] Oraclize documentation. https://docs.oraclize.it/. Accessed: 2nd June 2017.

[67] Oraclize documentation: security deep dive. https://docs.oraclize.it/#security-deep-dive. Accessed: 3rd June 2017.

[68] Oraclize documentation: TLSNotary Proof. https://docs.oraclize.it/#security-deep-dive-authenticity-proofs-tlsnotary-proof. Accessed: 11th June 2017.

[69] Postgresql - about. https://www.postgresql.org/about/. Accessed: 15th June 2017.

[70] Proof of existence: what is proof of existence? https://proofofexistence.com/about. Accessed: 3rd June 2017.

[71] Rinkeby.io. https://www.rinkeby.io/. Accessed: 15th June 2017.

[72] Robo 3t: Native and cross-platform mongodb manager (formerly robomongo). https://robomongo.org/. Accessed: 15th June 2017.

[73] Rootstock. http://www.rsk.co/. Accessed: 10th June 2017.

[74] Solidity browser compiler. https://ethereum.github.io/browser-solidity/. Accessed: 23rd Jan. 2017.

[75] Solidity documentation. http://solidity.readthedocs.io/en/develop/index.html. Accessed: 9th June 2017.

[76] Solidity documentation: checks-effects-interactions pattern recommendation. https://solidity.readthedocs.io/en/develop/security-considerations.html#use-the-checks-effects-interactions-pattern. Accessed: 11th June 2017.

[77] Solidity documentation: libraries. http://solidity.readthedocs.io/en/develop/contracts.html#libraries. Accessed: 15th June 2017.

[78] Solidity documentation: re-entrancy pitfalls. https://solidity.readthedocs.io/en/develop/security-considerations.html#re-entrancy. Accessed: 11th June 2017.

[79] Solidity documentation: security considerations. http://solidity.readthedocs.io/en/develop/security-considerations.html#security-considerations. Accessed: 31st May 2017.

[80] Solidity documentation: state machine pattern. http://solidity.readthedocs.io/en/develop/common-patterns.html#state-machine. Accessed: 9th June 2017.

[81] Solidity documentation: withdrawal pattern. http://solidity.readthedocs.io/en/develop/common-patterns.html#withdrawal-pattern. Accessed: 9th June 2017.

[82] State of the DApps. http://dapps.ethercasts.com/. Accessed: 22nd Jan. 2017.

[83] Tenancy deposit protection. https://www.gov.uk/tenancy-deposit-protection/overview. Accessed: 31st May 2017.

[84] Tenancy Deposit Scheme (TDS). https://www.tenancydepositscheme.com/. Accessed: 31st May 2017.

[85] Town crier: An authenticated data feed for smart contracts. http://www.town-crier.org/. Accessed: 3rd June 2017.

[86] What is MongoDB? https://www.mongodb.com/what-is-mongodb. Accessed: 15th June 2017.

[87] World coin index - Ethereum charts. `https://www.worldcoinindex.com/coin/ethereum`. Accessed: 2nd June 2017.

[88] Andreas M. Antonopoulos. *Mastering Bitcoin.* 2014.

[89] Vitalik Buterin. Ethereum white paper. `https://github.com/ethereum/wiki/wiki/White-Paper`. Accessed: 17th Jan. 2017.

[90] Vitalik Buterin. Roundup round III - 24th May 2017. `https://blog.ethereum.org/2017/05/24/roundup-round-iii/`. Accessed: 29th May 2017.

[91] Vitalik Buterin. State tree pruning. `https://blog.ethereum.org/2015/06/26/state-tree-pruning/`. Accessed: 21st Jan. 2017.

[92] Joseph Chow. Ethereum, gas, fuel & fees. `https://media.consensys.net/ethereum-gas-fuel-and-fees-3333e17fe1dc`. Accessed: 19th Jan. 2017.

[93] Wesley Egbertsen, Gerdinand Hardeman, Maarten van den Hoven, Gert van der Kolk, and Arthur van Rijsewijk. Replacing paper contracts with ethereum smart contracts, 2016.

[94] Vinay Gupta. The ethereum launch process. `https://blog.ethereum.org/2015/03/03/ethereum-launch-process/`. Accessed: 22nd Jan. 2017.

[95] Sergio Demian Lerner. Rootstock white paper. 2015.

[96] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[97] M. Sage. *Warfare in Ancient Greece: A Sourcebook.* Routledge Sourcebooks for the Ancient World. Taylor & Francis, 2002.

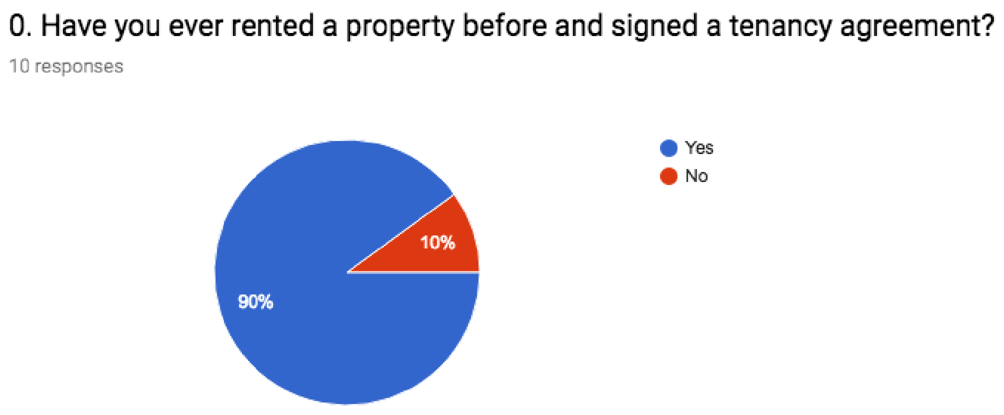
[98] Jutta Steiner. How do you know ethereum is secure? `https://blog.ethereum.org/2015/07/07/know-ethereum-secure/`. Accessed: 29th May 2017.

[99] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014.

[100] Tina Zhang. Express agreement: A framework for efficient contract negotiation and blockchain-based agreement verification, 2015.

Appendix A

Web Application User Feedback Questionnaire

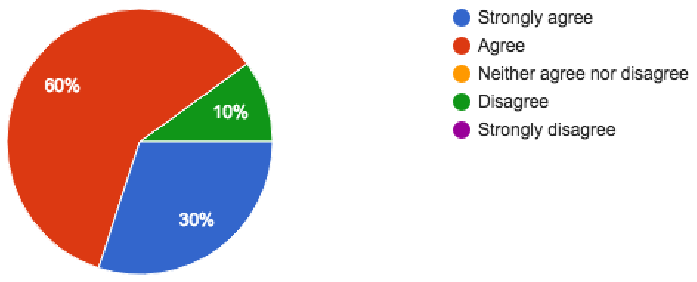

0. Have you ever rented a property before and signed a tenancy agreement?

10 responses

Figure A.1: User survey results: question 0

1. The home page of the website makes the purpose of the application clear.

10 responses



1. If you disagree, how could it be improved?

2 responses

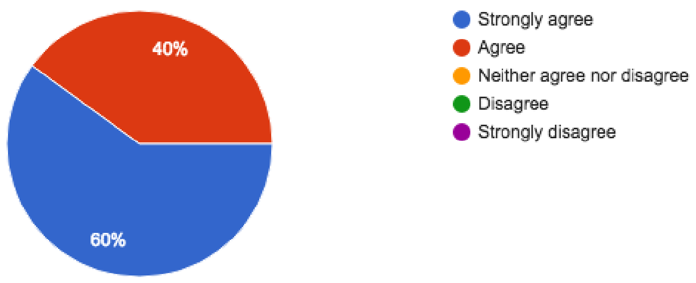
More in depth explanation of the applications purpose. Description of each parties role in the tenancy agreement.

A picture.

Figure A.2: User survey results: question 1

2. The application is easy to navigate.

10 responses



2. If you disagree, how could it be improved?

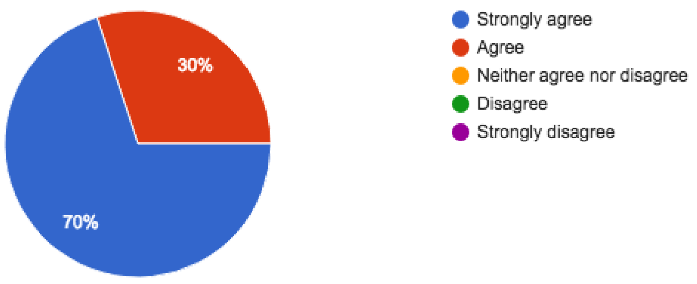
0 responses

No responses yet for this question.

Figure A.3: User survey results: question 2

3. It is easy to see the stage that the tenancy agreement is in and it is obvious which action needs completing next.

10 responses



3. If you disagree, how could it be improved?

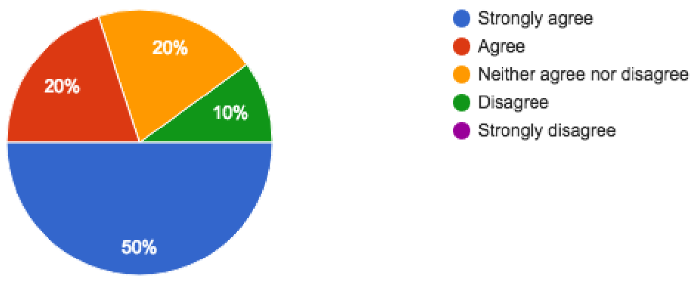
0 responses

No responses yet for this question.

Figure A.4: User survey results: question 3

4. The benefits of using the Ethereum network are clear.

10 responses



4. If you disagree, how could it be improved?

2 responses

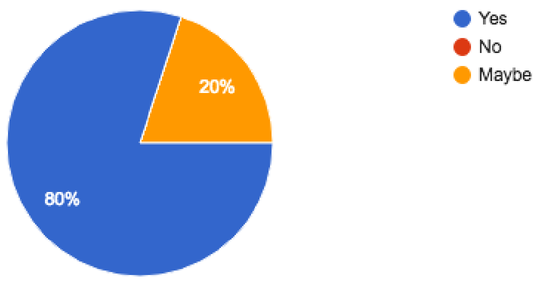
This is not explained anywhere in the application.

Some ethereum background.

Figure A.5: User survey results: question 4

5. Would you be interested in using the application to sign tenancy agreements?

10 responses



5. If not, what would make you more likely to consider using it?

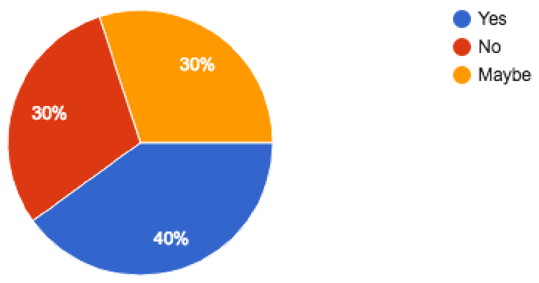
0 responses

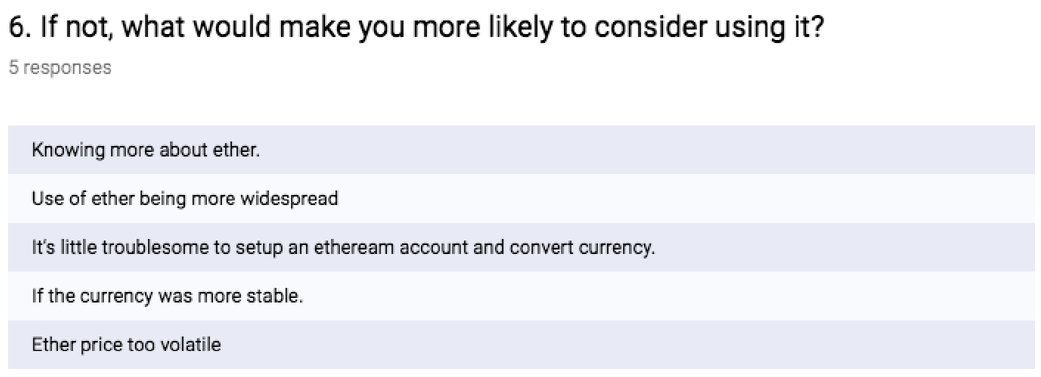
No responses yet for this question.

Figure A.6: User survey results: question 5

6. Would you be interested in using the application to pay your rent in ether?

10 responses



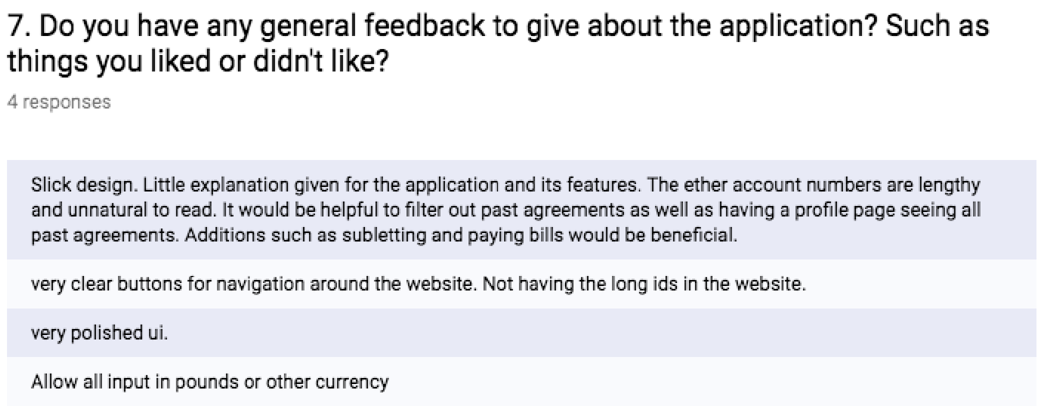


6. If not, what would make you more likely to consider using it?

5 responses

Knowing more about ether.
Use of ether being more widespread
It's little troublesome to setup an ethereum account and convert currency.
If the currency was more stable.
Ether price too volatile

Figure A.7: User survey results: question 6



7. Do you have any general feedback to give about the application? Such as things you liked or didn't like?

4 responses

Slick design. Little explanation given for the application and its features. The ether account numbers are lengthy and unnatural to read. It would be helpful to filter out past agreements as well as having a profile page seeing all past agreements. Additions such as subletting and paying bills would be beneficial.
very clear buttons for navigation around the website. Not having the long ids in the website.
very polished ui.
Allow all input in pounds or other currency

Figure A.8: User survey results: question 7