Renther, a Smart Contract auctioneering platform on the Ethereum Blockchain.

Ross Halpin

Final Year Project – 2018/2019

B.Sc. Single Honours Multimedia, Mobile and Web Development

Department of Computer Science

Maynooth University

Maynooth, Co. Kildare

Ireland

A thesis submitted in partial fulfilment of the requirements for the B.Sc. Single Honours in Multimedia, Mobile and Web Development.

Supervisor: Dr Phil Maguire

Table of Contents

List of Figures

Appendices

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study as part of BSc. Multimedia, Mobile and Web Development qualification, is *entirely* my own work and has not been taken from the work of others - save and to the extent that such work has been cited and acknowledged within the text of my work.

I hereby acknowledge and accept that this thesis may be distributed to future final year students, as an example of the standard expected of final year projects.

Signed: [illegible] Date: 27th March 2019

Acknowledgements

Special thanks to my supervisor Dr Phil Maguire for providing an invaluable source of discussion, encouragement and brainstorming throughout this dissertation project.

A big thank you to my peers and family for encouragement and support throughout the project.

Abstract

In recent years, the rental crisis in Ireland has become a growing problem across communities. While we are currently in a weaker economic climate, the cost of renting is rising far above that of previous economic boom periods. Many desperate and inexperienced renters are now vulnerable to predatory and fraudulent tactics by landlords and scam artists attempting to take advantage of this current market. In this thesis, I propose an auctioneering platform that will allow renters to bid for rental properties, while also attempting to protect them from potential abuses. I will discuss the Ethereum blockchain's smart contracts and how they can act as the backend for auction functionality. Briefly, I will explain the concept of the blockchain, along with my use of the InterPlanetary File System (IPFS) network as a decentralized store of multimedia for each auction. I will describe how smart contracts can be used to create an incorruptible ledger of traceable bids and bidders, and how smart contracts can act as a trustless and unbiased third party along with how contract-based solutions could potentially discourage fraudulent behaviour. I also propose a simple user interface to make using the blockchain as simple as possible. This platform also aims to display powerfulness of the blockchain, and how we can create always-on, secure and transparent applications which cannot be taken down.

1. Introduction

1.1 Topic

For this thesis, I will present the concept of smart contracts on the Ethereum blockchain written in the Solidity contract-oriented language. I will explain how they operate on the blockchain itself and how they could be utilized for an auction in this use case. I will further discuss the creation of the user interface using React.js and some other libraries, and how I utilized the Metamask [1] plugin and Web3.js [2] to deploy and interact with smart contracts on the Ethereum blockchain.

1.2 Motivation

Ireland is currently experiencing a rental crisis affecting many within our country, this is especially relevant to those in University as they are mostly young and inexperienced students and soon to be professionals. With a scarcity of rental options, it is increasingly difficult to haggle and barter with landlords to secure accommodation, as renting is in my experience, based on first come first served or biased selection. By creating an auction scenario for renting, the platform can please both the landlord who can guarantee interest and the prospective renter, as they would have a clear set period where the rental property is up for auction and bidders will collectively decide how much the property is worth through bidding. By placing the price of the rent in the hands of renters, this means the final price will be based on market sentiment rather than Landlords artificially raising the price in order to profit under false pretences. Through the locking in of the actual value of the winning bid with a smart contract, the landlord can be certain the winner is not going to waste their time, and in the event of the renter withdrawing, it could be possible to automatically compensate the landlord.

I cannot wholly solve the issue of inflating prices of rental properties, but I can attempt to make them transparent in order to fight against scams and greed. As there is currently large profit to be made in the market due to prices increasing to demand, this gives rise to opportunistic landlords and scam artists looking to increase their profit margins through fraudulent and greedy tactics. In general, these tactics include downright fraudulent advertisements where no property is actually available. While in legitimate cases such as in an auction scenario where the auction is not held digitally, the Landlord/Auctioneer can announce bids which either do not exist, or they can use a fake bidder who purposely drives up the price. These can be discouraged through Ethereum smart contracts. As the blockchain is immutable, all bids and bidding accounts can be made transparent, permanent and public. This traceability allows for potential investigation into fraudulent behaviour that could not be avoided through the solution I present but tracked due to its implementation. As centralized digital auctions accessed through the web can be vulnerable to attacks such as a denial-of-service (DoS) attack, where an excessive number of requests are made to a server to temporarily disrupt functionality, the distributed nature of the blockchain means there is no single attack point to disrupt the auction service.

As crypto-currency is still a relatively foreign concept to consumers, an effort must be made to make interacting with smart contracts as painless and easy as possible. By creating an easy to use platform this can work towards making cryptocurrency more appealing to the average consumer.

1.3 Problem Statement

To create a transparent, secure way to create and participate in auctions for rental properties through trust-less and immutable smart contracts on the Ethereum blockchain. Bidders and their bids should be listed publicly to deter fake bids and bidders. Other financial obstacles should be implemented to deter any fraudulent behaviours. A user-friendly interface should be provided to allow for easy interaction with blockchain technology in order to work towards mass adoption of crypto-currencies. The auction service should be secure and decentralized, so any manipulation of auction services can be avoided.

1.4 Approach

In approaching this problem, it was crucial that I learn how to utilize smart contracts. I set about learning what smart contracts were, what they were used for and how they were made. To expedite this process, I chose to partake in an Udemy [3] online learning course. The 24-hour-long course provided the basics of Solidity, common pitfalls, and patterns along with interacting with the blockchain via a dynamic interface. I compiled and deployed smart contracts easily and quickly through the Solidity Remix IDE [4] as I began to learn the ins and outs of Solidity and its somewhat limited features and boundaries. In research, I found the best option for interacting with the blockchain was to use a third-party API as handling transactions were outside of the project scope. I was already savvy on how an English Auction operates but took inspiration from the provided English Auction smart contract within the Solidity documentation [5]. I performed unit tests to verify the operation of the smart contracts through Remix.

From visual inspiration based on the interfaces of commonly used Irish rental websites, I planned a simple interface that would make use of a third-party framework for styling and interface functionality. As the Udemy course covered some basic use of React.js I found this easy to pick up and implement as the framework upon which the user interface would sit. In finding a solution for storing data off-chain I found a peer to peer solution that suits perfectly for easily storing and retrieving data. The API and interface functionality were tested utilizing mocking and unit testing.

1.5 Metrics

1.5.1 Smart Contracts

I plan to evaluate smart contracts built using Solidity by performing unit tests through the Remix IDE. This will allow me to quickly and effectively test the contracts due to the lack of any verification time on transactions and the easy to use contract interface provided by the development environment. The tests will be written using Remix's '*Assert()*' exception handling.

1.5.2 User interface and frontend functionality

The React.js frontend will be tested through Mocking API calls and responses along with unit testing functions using the Jest JavaScript testing framework. The front-end elements and components of the platform will be evaluated using the Enzyme testing utility that will allow me to test the rendering and operation of the user interface. The interface will also be tested using Selenium to carry out use case tests.

1.6 Project Achievements

I greatly familiarised myself with the blockchain and successfully deployed several working smart contracts to the network. The platform I created is a fully decentralized "always on" application. There is no single point or avenue for this application to be shut down, or to even experience temporary malicious downtime.

I believe this project expands into currently unfilled bleeding edge space within the blockchain area of research within Computer Science. Although the methods are proven, very little exists in terms of implementation. I believe this thesis pushes past the already plentiful conceptual space of auctions on the blockchain into the practical space.

2. Technical Background

2.1 Topic Material

A blockchain, in this case, Ethereum, “can be viewed as a transaction-based state machine: we begin with a genesis state and incrementally execute transactions to morph it into some final state.” [5] where the state can contain information, which is described by Wood (2014) as “account balances, reputations, trust arrangements, data pertaining to information of the physical world; in short, anything that can currently be represented by a computer is admissible.”. All of this information will exist within transactions on the blockchain, these transactions are then grouped together as blocks. These blocks are then “chained together using a cryptographic hash” [5] of the previous block, with each block acting as a ledger for a number of transactions. Each transaction has its own hash identifying it on the network, known as its transaction hash.

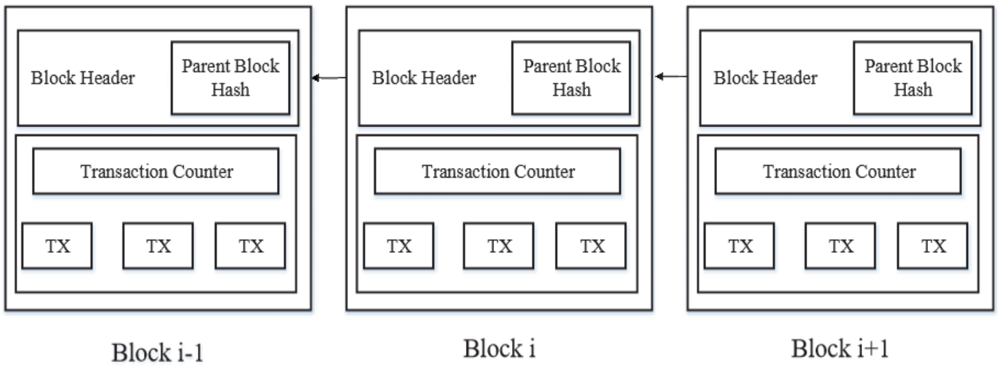


Fig 1. An example of blockchain which consists of a continuous sequence of blocks. [6]

Although Ethereum builds further upon these concepts, the basic architecture of a blockchain can be seen in Fig. 1. Each block is a “collection of relevant pieces of information (known as the block header), H, together with information corresponding to the comprised transactions, T” [5]. A decentralized blockchain exists on a “distributed network” made up of nodes running the blockchain’s software along with a copy of the entire network. “Consensus algorithms in blockchain are used to maintain data consistency” [6] this is to ensure that no incorrect transactions exist on the network. Each block contains a nonce, in Fig 1. this is the transaction counter value. Transactions fit into the architecture seen above in Fig. 1. as “TX”. For a block to be added to the blockchain, “miners” calculate a special hash value based on the given nonce. This difficult calculation is known as “Proof of Work” [7], and it is designed so that the “miner” must put forward their computing power to calculate this hash value, which “must be equal to or smaller than a certain given value”. Zheng et al. (2017) state that when the target value is reached, the miner broadcasts the block to other nodes, who then must come to a consensus on the correctness of the hash value. If everything is verified to be correct, all nodes then add this block to their copy of the blockchain. The miner is then rewarded the transaction fees.

Due to the persistence of the network, “it is nearly impossible to delete or rollback transactions once they are included in the blockchain.” [6]. On the Ethereum Blockchain, smart contracts exist as bytecode within transactions contained within blocks. Zheng et al. (2017) describe how smart contracts are “executed by miners automatically once the contract has been deployed on the blockchain.”. Ethereum provides the Turing complete Solidity language for creating these smart contracts.

There are currently a rising number of ‘Decentralised Applications’ built with Solidity smart contracts on the Ethereum blockchain. The majority of these exist in concept stages as mass adoption of both blockchain Cryptocurrencies and the use of smart contracts has yet to reach its potential. As transaction based smart contracts require security and are directly related to the use of blockchain as a financial network, there are many examples and previous works to draw from. In my investigation, I could find no previous research articles in terms of auctioning of property or rental properties via smart contracts, but I found many examples of auctions being conceptualized for other markets.

Researchers demonstrated their work [7] on an auction-based smart contract system for Energy consumption. In this system, there are energy "Seller Agents" and "Bidder Agents". The energy vendor will create an implementation of a "Vickrey" auction for available energy. The energy consumer will then bid via this smart contract. When the auction is completed, the winning bidder is then given the energy in exchange for their winning bid. This smart contract demonstration assumes some implementation of a smart "Meter Agent" that interacts with the blockchain while monitoring energy usage. This would essentially be a cryptocurrency based 'pay-as-you-go' meter, but with the ability to choose different vendors in a decentralised energy market.

Another example of an auction-based smart contract is found in an article [8] where an e-procurement platform is proposed utilising a reverse auction. The researcher defines procurement as "The act of obtaining or buying goods and services" which "includes preparation and processing of demand" [8]. This research puts forward an auction smart contract interface where multiple Buyers put forward proposals lower than the Suppliers pre-defined high offer. In a reverse auction, the price is gradually brought down from an arbitrarily high price until an agreeable price is met between prospective Buyer and Supplier. This proposal assumes some user interface on top of the smart contract but does not provide any working concept.

In a conference on 'Collaboration and Internet Computing', researchers offered a 'Decentralised Marketplace Application' [9] incepted as a solution to the centralised power that marketplaces such as eBay wield over merchants. They described one drawback of a centralised market as eBay's "ability to block merchants at their own whim", with their solution being a free market. Their implementation utilised the "Truffle Suite" development environment as a framework for the creation of both front-end and back-end features. In testing, they experimented with using a centralised MongoDB database for storing website data but resorted to using the BigChainDB decentralised database service in order to avoid a single point of failure in their design. Their decentralised application also makes use of their own node on the IPFS network for storing media off-chain, which was implemented via Node.js. This design also utilises Ethereum's Web3.js library provided through Metamask to allow users to interact with the blockchain through the Chrome web browser.

2.2 Technical Material

2.2.1 Solidity Auction Smart Contract

In crafting the auction contract, it was necessary to understand the underlying algorithm of an auction. Luckily, within the 'Solidity by Example' documentation [10] there exists an "Open Auction" implementation of a solidity smart contract. I would easily be able to adapt and develop upon this example to further enhance it to become a fully transparent and fraud-resistant contract.

2.2.2 Web3.js and Metamask

The Ethereum JavaScript API, Web3.js, makes up the backbone of this platform for interacting with the blockchain. Web3.js provides an API for connecting to smart contracts, sending transactions and retrieving data. The plugin I will be using that provides its own Web3.js connected to their own Ethereum node, is Metamask [1]. Metamask is a secure in-browser cryptocurrency wallet that allows for handling users and their accounts in a safe manner. As stated in the Metamask compatibility guide [11], the plugin injects the Web3.js library into the browser page upon request and approval by the user. The platform can then use this library to interact with the blockchain through the user's accounts.

2.2.3 React.js

React [12] is a Javascript library that allows for quick, simple creation of user interfaces, the framework "will efficiently update and render just the right components when your data changes.". It implements a virtual Document Object Model (DOM) which is a node tree that makes up a standard web page. Re-rendering and updating components individually in the DOM can be slow and inefficient, instead React's virtual DOM "abstracts out the attribute manipulation, event handling, and manual DOM updating" [13] into a much quicker and smoother process. Data flows in a top-down fashion through React components, this is done through "props" or properties that are included in the element's tag when declared in the parent, and then accessed through this.props in the child. HTML components are declared through React.js tag syntax, JSX [14] which is syntactically like HTML.

2.2.4 Smart Contract Security Patterns

When dealing with monetary value within smart contracts, there are some attacks that take advantage of the behaviour of smart contracts and the Ethereum network. Fortunately, there are patterns to avoid some of these pitfalls. It is not secure to have a user triggered function that transfers assets. So, in a platform such as this where there is no backend that could be given permission to perform automatic transfer of funds, it is necessary to allow the bidders to withdraw their bids manually. One attack on a user called function is a "re-entrancy" [15], where the function is called multiple time consecutively before the first call has finished. In a withdrawal function, if we simply transfer the users balance out of the contract, the user could call the withdrawal function multiple times thus withdrawing their balance multiple times over, effectively clearing out the contract. This can be avoided using the "Checks-Effects-Interaction pattern" which "defines a certain order of actions: First, check all the preconditions, then make changes to the contract's state, and finally interact with other contracts" [15]. (See Appendix 1 for code example)

2.2.5 InterPlanetary File System

In order to store media in a decentralised way, this project makes use of the 'ipfs-http-client' [25], which will allow the platform to send HTTP requests to the IPFS network, which "is a peer-to-peer distributed file system" [16]. This makes it the perfect candidate for storing the media related to each auction. Upon uploading media to the IPFS network, it responds with a unique hash identifying it on the network. This can be stored as plain text within each auction smart contract, to be retrieved later.

2.2.6 Fuze.js

To implement a search functionality within the platform I sought an easy and lightweight way to implement search queries and results so that users can search for and interact with desired auctions. Fuze.js [17] solved this problem by allowing me to input the data making up each auction and then filter it based on my search query.

3. The Problem

3.1 Analysis of Problem

This thesis aims to provide a transparent, secure and open platform for both renters and landlords. The project goal is to achieve this by utilising blockchain technology using Ethereum smart contracts. A simple yet effective solution should be achieved through a combination of both a smart contract interface and front-end graphical user interface to break down the technological barrier found with the use of cryptocurrencies. The smart contracts will encapsulate all the standard functionality of an auction while maintaining security and providing a solution to discourage fraudulent behaviour.

A simple and easy to operate graphical user interface should bridge the gap between users and smart contracts, providing a front-end for the auction deployment along with methods for interacting with other pre-existing auctions. The interface should use a third party to handle transactions as this out of the scope of the project. An interface walking through a tutorial for getting set up with cryptocurrency and using the interface should be provided to make cryptocurrency as user-friendly as possible.

The auction platform should be always-on, with no single point of failure, thus a solution to storing auction media off-chain should also be implemented as no centralised database will be used. The auction platform should be able to display and interact with all auctions created with it, this means there must be a linking factor spanning from each auction back to the platform, a list of all auctions so that they can be accessed from the platform.

3.2 System and User Requirements

An initial overview of my system design can be seen in Fig 2. This design focuses on 5 main components:

- Front-end interface for creating, deploying and interacting with smart contracts.
- Storage device for storing images related to the auction.
- A connection to the Ethereum blockchain.
- An address, or wallet, containing sufficient funds on the blockchain which is owned by the user.
- Smart contract acting as the auction back-end functionality.

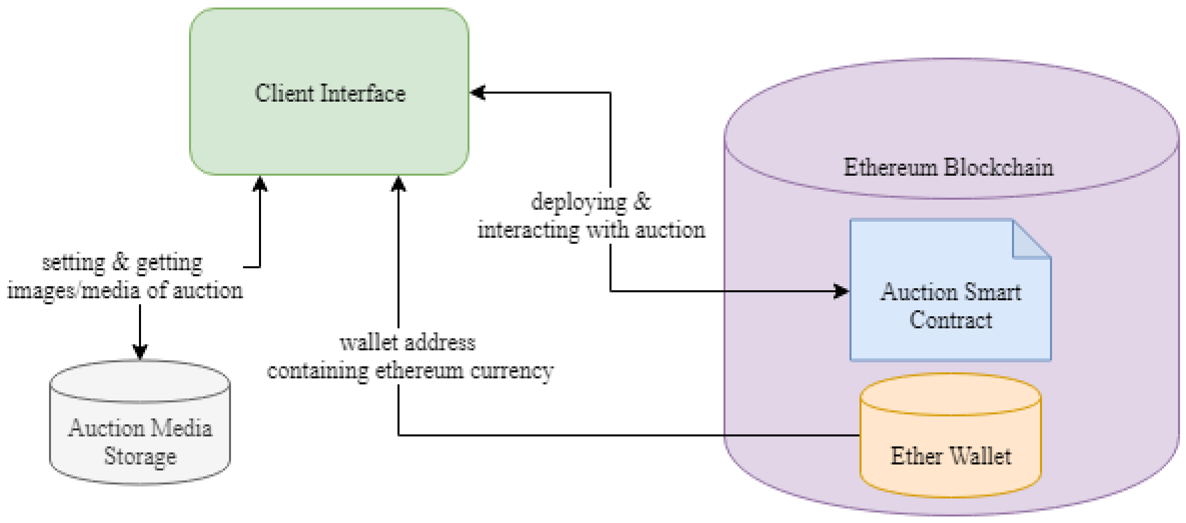


Fig 2. System design overview.

The functional requirements of the auction itself is an implementation of the standard English auction, the bidders offer subsequently higher bids in competition with each other and upon the culmination of the auction, the bidder with the highest bid wins. As per the project specification, all bids and bidder accounts to be transparent, thus, the auction smart contract must have the following functionality:

- Place bid
- Log Bidder/Bid
- Track Highest Bidder
- Withdraw bid
- End Auction

The 'Client Interface' will contain functionality for any user to act as both a landlord or a renter. For a user acting as a landlord, the interface will allow the user to define the terms of the auction, details about the property and select images to allow further description of the offering. The text data should then be compiled into a suitable format to be added as data to the smart contract. The auction media, i.e. images, should be linked between their location on a storage service and the smart contract that they belong to. The interface will then allow the user to deploy their auction smart contract to the Ethereum blockchain.

For a user visiting the platform as a renter, the interface will allow them to interact with the auction's smart contract functionality. They will be able to place bids and withdraw them if they are outbid, while also being able to see a history of previous and subsequent bids on the property. The basic design of the front-end platform can be seen in Fig 3. below. Mock-up wireframes of the user interface can be seen in Appendix 2, giving an overview of the simplistic user interface.

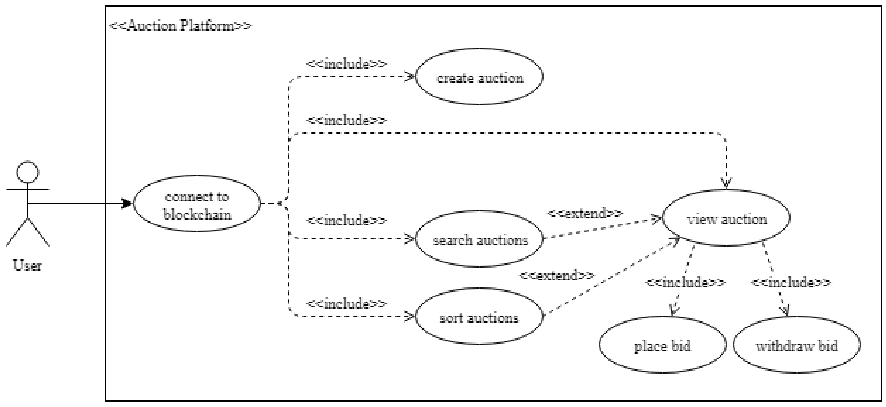


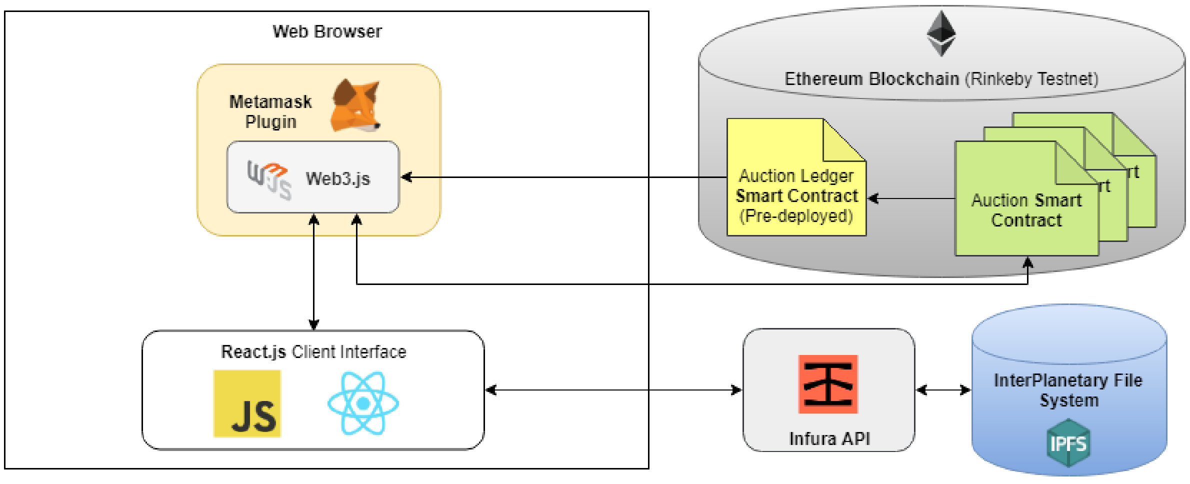
Fig 3. Use case diagram of user interface design.

4. The Solution

In this chapter, I will outline my final design of the auction platform and the decisions which culminated the structure illustrated in Fig 4. below.

4.1 Connecting to the Blockchain

As sending transactions on the Ethereum network will cost actual monetary value, I will be using an Ethereum test network known as the Rinkeby Testnet. The software for this testnet is exactly the same as the real Ethereum network, thus interacting with it is the same, and hereinafter I will still refer to the Rinkeby testnet as the Ethereum network/blockchain. The advantage of this testnet is that I can give myself large amounts of valueless Ethereum cryptocurrency to use for sending transactions.



Web Browser
Metamask Plugin
Web3.js
Ethereum Blockchain (Rinkeby Testnet)
Auction Ledger Smart Contract (Pre-deployed)
Auction Smart Contract
React.js Client Interface
JS
Infura API
InterPlanetary File System
IPFS

Fig 4. Final design overview for auction platform.

Firstly, in order to connect to the Ethereum blockchain, the platform must connect to one of the networks many nodes. A node is a complete copy of the Ethereum blockchain along with software for providing transaction functionality and consensus with other nodes. This is an intensive service to run and would require that I handle the sending of transactions to the network through this node using my own configuration of Web3.js, where Web3.js is the JavaScript Ethereum API. Fortunately, there are some free alternatives for connecting to the Ethereum network. The solution I will be using is ‘Metamask’, which is a Google Chrome plugin which provides a connection to Metamask’s own Ethereum node through a pre-configured Web3.js library. Metamask, upon login and permission request, injects this Web3.js library into the browser page. Any accounts you have set up in Metamask will also be securely injected into the browser page for access by the front-end. Handling Ethereum accounts and transactions securely through my own implementation would have been an entire project on its own, so I believe using a trusted third-party alternative is the best decision.

The Javascript Ethereum API is the most important component of this platform, as from the pre-configured Web3.js I can now send transactions to the Ethereum Blockchain. The role of Web3.js as the mediator between the blockchain and the front-end can be seen in Fig 4. above. This library gives me several functions for deploying smart contracts, and for connecting to already deployed contracts on the blockchain. Through this API, I can also call functions or variables within contracts at no cost, provided that doing so does not involve changing the state of the contract. If I modify the state, there will be fees involved.

4.2 Storing Media Off-Chain

To keep this platform decentralised for both security and freedom, I must implement a way to store images relating to each auction. Since changing the state of a smart contract costs monetary value, which increases significantly by the size of the data you try to store on the blockchain, it is not practical to store even 1 Megabyte on the blockchain. I also can’t use any centralised database/storage services as this creates a point of failure within the platform. The solution to this problem is InterPlanetary File System (IPFS), like the Ethereum network, IPFS will also require a connection to one of its nodes so that the platform can interact with it. I could set up my own node, but this would involve storing my fair share of the IPFS network, which isn’t within the scope of this project. To connect to an IPFS gateway I will be using the node provided by Infura. This structure can be seen in Fig 4. above. I can connect to the IPFS network through the Infura gateway using the ‘ipfs-http-client’[25] library, which is essentially a wrapper for HTTP POST and GET requests to IPFS nodes.

I can use IPFS by performing a HTTP POST with the data, to the network, which then responds with a hash string, essentially an identification of the data on the network. The data can then be accessed subsequently by performing a HTTP GET on an IPFS gateway with the identifying hash string. In order to somehow link this data, and it’s identifying hash to the contract, the solution is to collect the data to be stored, submit it to the IPFS network, then send the returned hash as an argument with the deployment of the auction smart contract. This IPFS hash can then be accessed and called later by the platform via Web3.js as it is now stored in the contract itself.

This solution replaces storing potentially large amounts of data on the blockchain, with a single hash string. Thus, through this method, the platform now stores its data off-chain, but in a satisfactory decentralised manner.

4.2 Smart Contracts

4.2.1 Ledger

For the platform to be able to access all of the smart contracts deployed from it, I will be creating a smart contract that acts as a ledger of all the addresses of each auction smart contract. Each auction smart contract will automatically send their address to this ledger, which the front-end will use to grab all the auction addresses so their details and data can be pulled from the blockchain. This smart contract is pre-compiled and pre-deployed before the platform. The structure and interaction of the ledger and auction contracts within the platform can be seen in Fig 4. while the class structure can be seen below in Fig 5. (see Appendix 3 for ledger code and outline.).

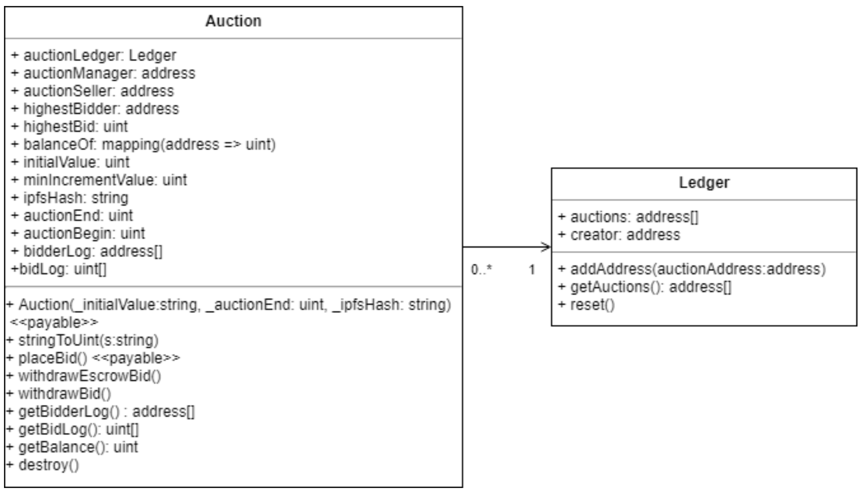


Fig 5. Class diagram for auction and ledger smart contracts.

4.2.2 Auction Smart Contract and Logic

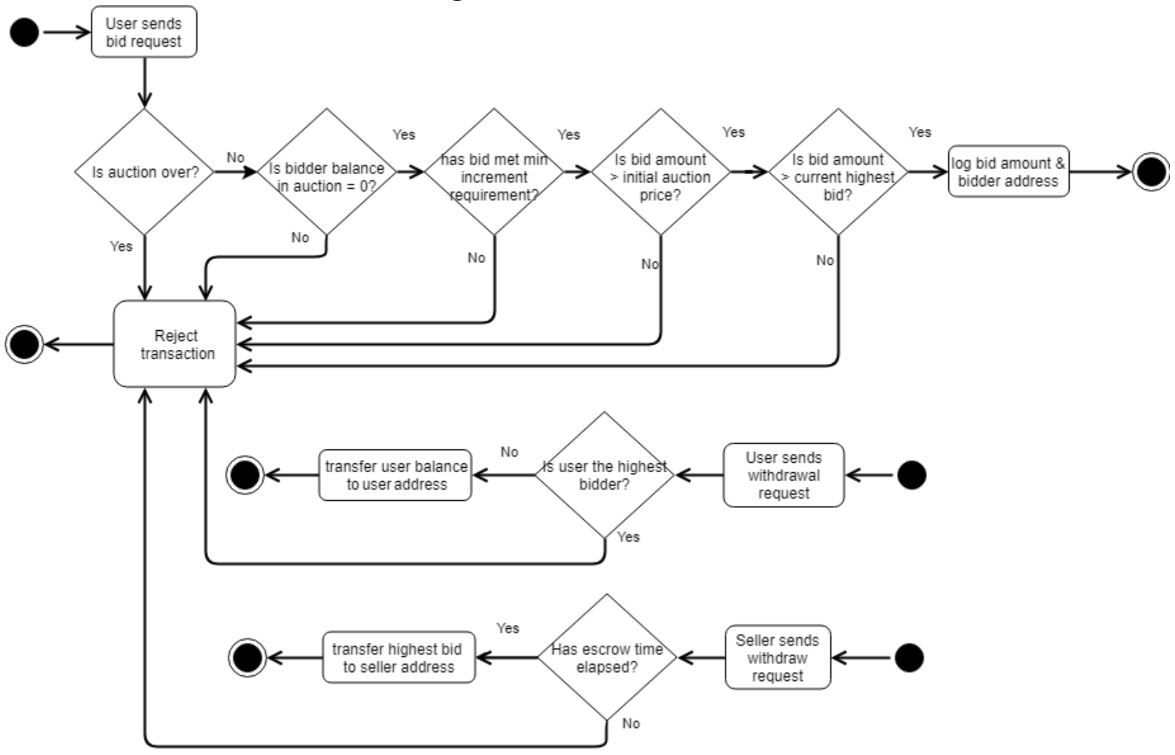
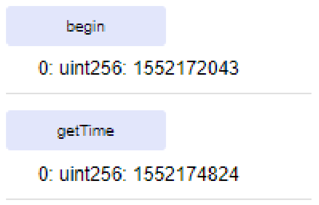

User sends bid request
Is auction over?
No
Yes
Is bidder balance in auction = 0?
Yes
No
has bid met min increment requirement?
Yes
No
Is bid amount > initial auction price?
Yes
No
Is bid amount > current highest bid?
Yes
No
log bid amount & bidder address
Reject transaction
transfer user balance to user address
No
Is user the highest bidder?
Yes
User sends withdrawal request
transfer highest bid to seller address
Yes
Has escrow time elapsed?
No
Seller sends withdraw request

Fig 6. Activity diagram for auction smart contract.

Next, I must write the auction logic within the smart contract and compile it so that I can deploy it to the network via Web3.js. For this I utilised the Solidity smart contract language. The flow of this system can be seen in Fig 6. which has been illustrated through an activity diagram. This diagram displays how the auction logic will perform after its deployment to the network, focusing on when a user places or withdraws a bid. Auctions are of course time-based, and they will purposefully exist long after they have expired in order to keep transparency on this platform. To prevent interactions with the auction after it has ended, I must track the time from when the auction begins to when the auction ends. Every time the state of the auction is changed, fees must be paid, thus there is no implementation of a running timer, because as the timer increases the value it will change the state, and the state cannot be changed outside of a transaction.

Solidity offers a solution to keeping time without changing state, by allowing access to the block timestamp. The creator of Ethereum defines the block timestamp as “a scalar value equal to the reasonable output of Unix’s *time()* at this block’s inception” [5] with the block’s inception being when the block containing the smart contract is mined and added to the network. The smart contract itself exists within this block as bytecode contained within the data of a transaction, this transaction and many other transactions are stored in a block. When you interact with the smart contract in a way that modifies its state, this must occur through a new transaction on the blockchain, which will be stored in a new block. I can then get the block timestamp of this new transaction and compare it to when the block containing the contract was incepted, this then gives me a timing component for the auction.

```
uint public begin;
constructor() public{
    begin = block.timestamp;
}
function getTime() external view returns (uint){
    return block.timestamp;
}
```



```
0: uint256: 1552172043
```

```
0: uint256: 1552174824
```

Fig 7. Smart contract time keeping example.

An example Solidity contract implementation of timekeeping can be seen above in Fig 7., note that the ‘*block.timestamp*’ in the ‘*constructor()*’ function is defined when the smart contract is created, and the ‘*block.timestamp*’ in the ‘*getTime()*’ function is defined when a user sends a call request to that function. The resulting difference in the Unix timestamps can be seen on the right which was accessed using the Remix IDE. Also, note that the ‘begin’ variable can be accessed externally without a getter function, this is a specific feature of Solidity contracts.

Table 1. Auction smart contract variables.

Variable Name	**Meaning**
auctionManager	Address of auction manager
auctionSeller	Address of auction seller
auctionLedger	Contract interface for sending addresses to auction ledger contract
highestBidder	Address of highest bidder
balanceOf[]	Mapping of bidder address to contract’s Ether balance of bidder
initialValue	Min price set by the seller
minIncrementValue	Min increment value of bids
ipfsHash	IPFS hash address of auction media
bidderLog[]	Array of bidder addresses
bidLog[]	Array of bid amounts
auctionBegin	Block timestamp defined when the contract is initialised.
auctionEnd	Unix timestamp initialised by auction creator

Table 2. Auction smart contract functions.

Function Name	**Use**
constructor()	Initialise smart contract with auction parameters and details.
placeBid()	Performs requirement checks, either rejecting or allowing bid depending on if it passes all checks. Updates relevant parameters if the bid is allowed.
withdrawBid()	Implementation of the Checks-Effects-Interaction pattern checks msg sender is not the highest bidder, allows the bid to be released if not.

withdrawEscrowBid()	Implementation of the Checks-Effects-Interaction pattern, checks if auction + escrow time has elapsed, checks if msg sender is the seller, allows contract manager to bypass checks.
stringToUint()	Converts string to integer.
getBidderLog()	Returns bidderLog[]
gtBidLog()	Returns bidLog[]

As the auction algorithm depends on several checks before placing a bid, for example, to do this I will be making use of Solidity's '*require()*' exception handling, in order to handle all state changes that do not meet the conditions of the auction algorithm. '*require()*' is a 'state-reverting exception', if the function's require statement fails, any state changes which occurred when the user called the parent function will not take effect and the transaction will fail. The '*require()*' needs to evaluate to true in order to proceed, for e.g. '*uint foo = 1; uint bar = 2; require(foo == bar);*', where the requirement would evaluate to false and an exception would be thrown.

Now I must implement the standard English auction in a smart contract. A contract in Solidity acts like classes in other OOP languages. The contract has a '*constructor()*' function that initialises the auction with 3 variables. These variables and their explanations can be seen in Table 1. An interesting limitation of Web3.js and Solidity appeared here, as floating-point numbers are not yet supported except in the form of ether. Although Web3.js can convert integers to Ether values it will throw an error when given a decimal or on the opposite scale a "BigNumber". For this reason, the '*initialValue*' is given as a string value. The string is then converted back to uint within the contract [18].

The '*constructor()*' checks through a '*require()*' that '*auctionEnd'* is greater than the current block timestamp plus one day this is to ensure the auction date does not end before the current time. '*auctionBegin*' will be used for the user interface and is initialised within the constructor as the current block timestamp. The auction will also require payment sent with the deployment, which allows for an advertisement fee, pre-defined within the contract code. This is done by making the constructor '*payable*' and using a '*require()*' statement which only accepts a msg value greater than the predefined fee. In Solidity a function is defined as payable to enable it to receive Ether value, e.g. '*constructor() public payable{ require(msg.value > 1 ether);}*'. The '*constructor()*' also initialises the '*Ledger*' contract interface and adds the address of the auction to it. The '*auctionManager*' is initialised with a pre-defined address, the advertisement fee is then transferred to this address. this address would theoretically be that of a trusted third party, perhaps the operator of the platform. This address gives power over the contract to another third party, an optional safety measure which is outside the scope of this thesis. This address holder could bypass the contract's requirements to recover funds in escrow if the auction was discovered the be a scam. The implementation of an auctionManager is used conceptually, but no user interface is defined for this functionality.

The other functions necessary for an auction can be seen in Table 2. In the '*placebid()'* function, I must implement 5 requirement checks as defined in Fig 6. These included checking if the auction has ended, ensuring the bidder has an empty balance on the contract, is the bid higher than the current highest bid, and ensuring the bidder is not already the highest bidder. If these '*require()*' statements are passed, the '*balanceOf[]*' is updated with the bidders address and bid amount, the '*highestBidder'* is then set to the bidder's address, and the '*highestBid*' is set to the bid amount. These values are then pushed to '*bidderLog[]*' and '*bidLog[]*'.

As all bidders require the ability to withdraw their bids at any time if they are not the highest bidder, this is implemented with a single ‘*require()*’ statement simply checking the user’s address is not that of the highest bidder. The Checks-Effects-Interaction pattern [15] is implemented here, ensuring a local temp variable is first set to the users ‘*balanceOf[]*’ value, then the ‘*balanceOf[]*’ value of the user is set to zero. The value within the local temp variable is then transferred to the user's address. This avoids the re-entrancy attack [15] on the auction contract. The ‘*withdrawEscrowBid()*’ function operates with the same Checks-Effects-Interaction pattern but requires the escrow time period has elapsed, and the caller is the ‘*auctionSeller*’ upon this time. The escrow is incorporated as a safety feature and deterrent to any potential fake auction scams. Choosing an arbitrary length of time such as the current escrow period of 26 weeks means there is no way for a seller to have quick access to any monetary value. It is assumed that the winning bid will take the place of the deposit and in a legitimate use case, it’s lack of immediate availability should not be problematic to a landlord. The withdrawal of escrow funds also allows the ‘*auctionManager*’ to bypass these checks in the event of a scam or auction. If a user attempts to withdraw the escrow balance, it checks if the user is the auction seller, it then checks if the escrow time has elapsed. If the auction manager attempts to withdraw they will be allowed to without impedance.

The completed Ledger and Auction smart contracts are then pre-compiled. This is done using the Node.js Solidity Compiler ‘solc’. Upon compilation of a smart contract, the application binary interface (ABI) and bytecode are returned. The bytecode, which is unreadable machine code, is the actual code which will be deployed to the blockchain. The ABI, similar in readability to JSON formatted data, is the bridge between the machine code and a user or user interface interacting with the contract, such as for allowing the Web3.js API to interpret the machine code in order to access the smart contracts functions and variables. The Ledger is deployed once before the platform is launched, while the auction smart contract will be deployed multiple times via the platform interface.

4.3 Platform Front-End

The user interface was built with pre-made Material UI components in a React.js environment, using HTML5, CSS3, JSX and Javascript. Using wireframe mock-ups (see Appendix 2) I began to construct the platform. As I wanted to make things as simple as possible, this meant creating a single page application which contains all the functionality of the platform (see Appendix 5 for screenshots).

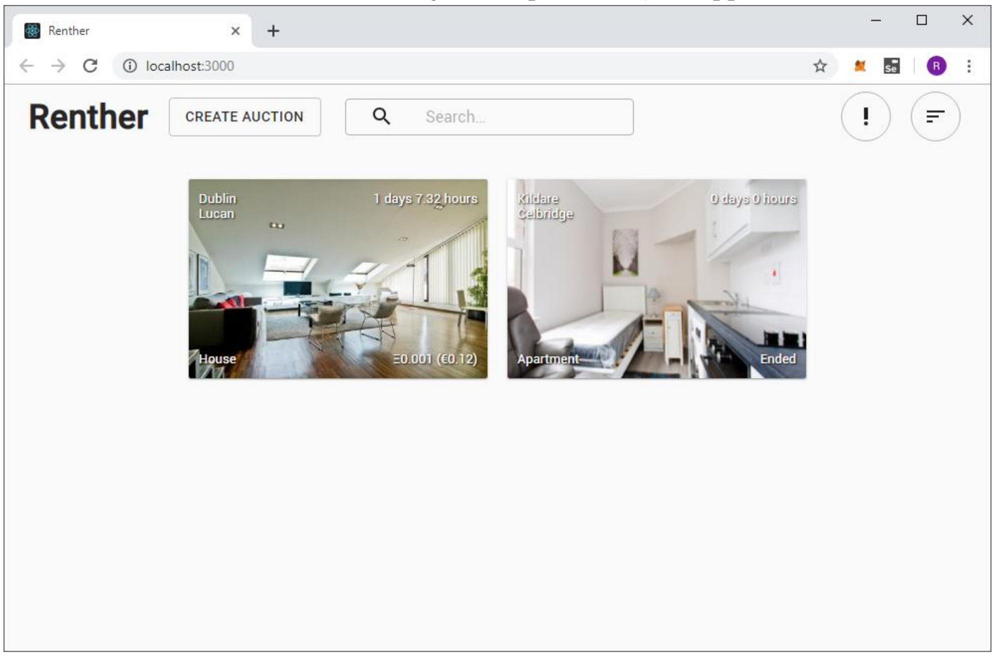


Fig 8. Main user interface view with example auctions.

Upon connecting to the platform, a Metamask notification will appear requesting access to enable its use within the interface. Once approved, the interface will then send a request through Metamask's provided Web3.js API to the Ledger smart contract, grabbing all the auction addresses stored in it. In the background the interface will loop through these addresses and send requests to each Auction smart contract, grabbing all the relevant data stored in the contract including the IPFS hash. This is done every 1000ms and any updates to the array of addresses will be automatically rendered by React.js. The script loops through the received data, sends a HTTP GET request to an IPFS gateway to retrieve the stored media, and then maps an '*AuctionCard*' component to the user interface passing the auction data to the component, which then automatically renders the relevant data.

The '*AuctionCard*' component contains all the functionality of each auction, it generates the HTML, in the form of React.js' JSX, that is displayed as the advert when the platform is loaded up, but also contains a modal component in order to keep with the single page design. This modal dialogue will be the window into each auction and its functionality. The data sent to the '*AuctionCard*' component through props is propagated to the '*BidTable*' and '*Carousel*'. The '*BidTable*' will return the log of bids and bidders as a Material UI table, sorted in descending order of bids, while the '*Carousel*' component will return the Material UI image carousel component, which I have filled with the images passed down through props. The modal also contains buttons for both placing bids and withdrawing bids. A request is sent through Web3.js to the auction to check if the current Metamask user's address has any balance in the auction or if they are the highest bidder. The bid button will be automatically disabled if they do not meet the criteria to bid based on the contract's response. The same functionality is used for withdrawal.

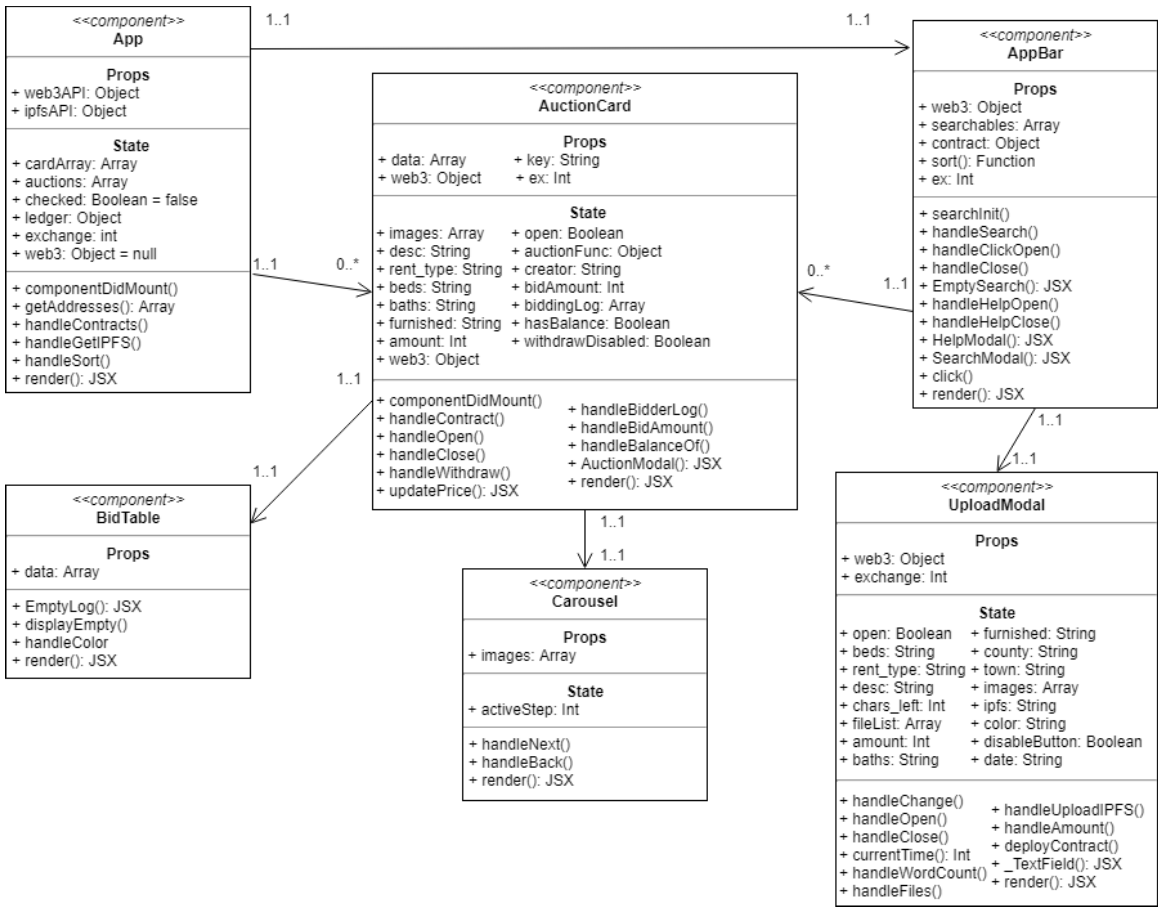


Fig 9. User interface React.js 'component' class diagram.

The ‘*AppBar*’ component of the platform handles the search, contract creation, and implements a button for calling the ‘*handleSort()*’ function in the App component, a “create auction” button to render the ‘*UploadModal*’ component and a “help” button to render a modal dialogue with help information. The “*UploadModal*’ component acts as a form handling the user input of all the property and auction details, along with background functionality for uploading the data to the IPFS network, then deploying the auction smart contract with the IPFS hash response as an argument. The new smart contract will then show up automatically on the page once the transaction has been verified on the blockchain and its address is pulled from the Ledger.

The ‘*AppBar*’ is passed the same data via props as the ‘*AuctionCard*’, but for use with the lightweight Fuze.js [17] in order to perform a fuzzy search on all the auctions via queries taken from the search bar. Fuze.js will return relevant auctions to the search queries, which are then mapped into ‘*AuctionCard*’ and rendered. I chose to implement it this way in a scrollable modal so that the search results are separate to the constantly updating auction view, this way the search results won’t update with any newly added auctions until the next search. It also allows the user to quickly withdraw a search by simply clicking away from the modal.

5. Evaluation

5.1 Software Design Verification

To verify the design of the platform this was done by checking several parts of the architecture. Checks carried out to ensure the smart contract was deployed successfully to the blockchain. This was verified by outputting the contract address and then verifying its existence on the network by inputting the address to EtherScan [19]. The functionality of the smart contracts was verified using the activity diagram as seen in Fig. 6., and the class diagram as seen in Fig 5., which allowed me to evaluate the use case scenarios of both bidders and the seller in all 3 scenarios, such as when the user bids, when the user withdraws and when the seller withdraws. The contract methods were tested using unit tests in the Remix IDE as it allowed me to quickly compile and deploy the contracts in a virtual blockchain environment. Unit tests and use case tests were implemented to verify the React.js user interface and functionality.

5.2 Software Verification/Solution Verification

5.2.1 Smart Contract Unit Testing

Unit tests were performed on all methods within the Auction and Ledger smart contracts. Remix provides the “remix_tests.sol” [20] library that allows for testing with assertions, meaning I can check the correct behaviour has occurred relative to the function executed. Remix, unfortunately, does not contain the functionality to automatically run tests where there are payable functions. This meant each test had to be run manually and the results checked manually. First, the Auction contract was initialised with mock values, these initial values were then asserted to verify they were updated. Checks were done on all other methods to ensure they correctly updated the contract variables. An example of an assertion unit test and the result can be seen below in Table 3. The assertion is checking that the value in the first position of the bidderLog is equal to address of tester contract after the tester contract has already placed a bid on the auction. A relevant message describing the test is also defined in the assertion. As these tests must be run manually, the response is in a JSON form, which gives a ‘*“passed”:true*’ message indicating the test has indeed passed.

Table 3. Assertion and Result

```
function check_3_Balance_Logs() public {
   Assert.equal(auctionToTest.bidderLog(0), address(this), "check bidderLog has been updated");
}
```

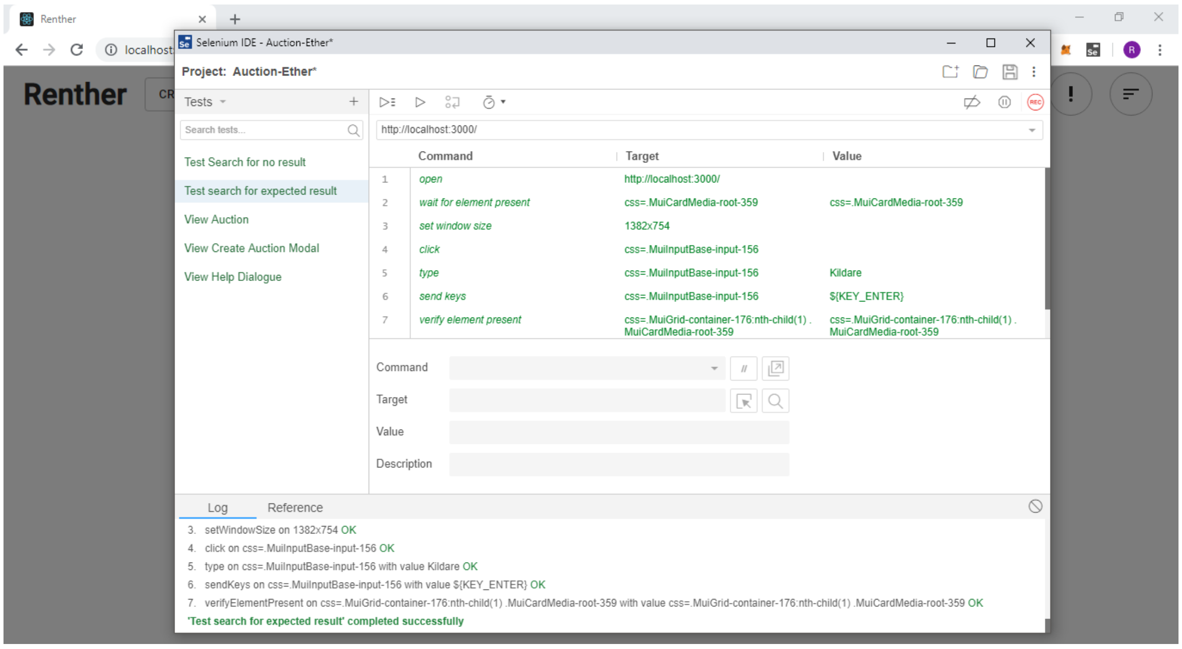
```
"from": "0x8609a0806279c94bcc5432e36b57281b3d524b9b",
"topic": "0xe81a864f5996ae49db556bf6540209c15b8077395a85ede9dfa17ad07d9ff366",
"event": "AssertionEvent",
"args": {
  "0": true,
  "1": "check bidderLog has been updated",
  "passed": true,
  "message": "check bidderLog has been updated",
  "length": 2
}
```

5.2.2 React.js and API Unit Testing

The front-end was verified with both unit tests and automated use case testing, this was done using several testing suites. As Javascript is a scripting language, more function focused than Object focused, it was necessary to test that all functions were operating as expected. Unit tests were carried out using Jest.js [21] which allowed me to mock API calls within the platform to validate they behave correctly. Jest provides rich tools for doing so allowing for mocked modules to replace those used for any network calls. Mocking involves creating functionally similar local modules to their API counterparts, allowing for testing how the platform behaves rather than how the API operates. Interactions with the blockchain via Web3.js, along with storing and retrieving media through IPFS were mocked and tested through this framework. Enzyme testing utility [22] was utilised for rendering React.js components in the test environment.

5.3 Validation and Use Case Testing

It would not have been practical to carry out a user study for this platform, due to needing to take several steps before using the platform, including installing Metamask, acquiring an Ether wallet and acquiring Ether on the testnet. To validate the platform, it was necessary to carry out use case testing to validate the overall behaviour of the platform along with its simplicity. These use cases were defined and then carried out automatically using the Selenium IDE [23] browser plugin. Selenium allows for web browser automation through set instructions. The tests covered a range of interactions within the user interface (see Appendix 4 for use case examples) such as viewing auctions and searching. Manual use case tests were done for placing bids and withdrawing bids due to the involvement of multiple users required for the functionality to be evident.



	Command	Target	Value
1	open	http://localhost:3000/	
2	wait for element present	css=.MuiCardMedia-root-359	css=.MuiCardMedia-root-359
3	set window size	1382x754	
4	click	css=.MuiInputBase-input-156	
5	type	css=.MuiInputBase-input-156	Kildare
6	send keys	css=.MuiInputBase-input-156	${KEY_ENTER}
7	verify element present	css=.MuiGrid-container-176:nth-child(1) .MuiCardMedia-root-359	css=.MuiGrid-container-176:nth-child(1) .MuiCardMedia-root-359

```
3.  setWindowSize on 1382x754 OK
4.  click on css=.MuiInputBase-input-156 OK
5.  type on css=.MuiInputBase-input-156 with value Kildare OK
6.  sendKeys on css=.MuiInputBase-input-156 with value ${KEY_ENTER} OK
7.  verifyElementPresent on css=.MuiGrid-container-176:nth-child(1) .MuiCardMedia-root-359 with value css=.MuiGrid-container-176:nth-child(1) .MuiCardMedia-root-359 OK
'Test search for expected result' completed successfully
```

Fig 10. Selenium tests running in browser.

6. Conclusions

In conclusion, I will discuss the results of this project and the contribution towards decentralised applications utilising smart contracts on the Ethereum blockchain.

6.1 Smart Contract Review

The goal of this solution was to implement a transparent, secure and open standard English Auction within a smart contract on the Ethereum blockchain. I believe this implementation successfully achieves these goals. For transparency, all bids and bidders are logged permanently and made available. There can also be an advertisement fee and escrow period to help towards deterring fraudulent scams. the auction algorithm performs as expected of an auction and useful data is stored within the Auction contract itself such as the IPFS hash. Combining this with the ledger contract, the platform effectively exists outside of the interface. If a user has access to both contract's Application Binary Interface (ABI), users can still access the ledger of auctions, and from the ledger they can access all the auctions. As each auction contract contains all of the necessary information related to the auction, along with the IPFS hash, this means that the auction can still function, and users can still access this information without the front-end platform. This is all due to the decentralised nature of blockchain technology. If a malicious party were to bring down this platform, it would involve taking down the hundreds of nodes within the Ethereum network.

6.1 Front-End User Interface Review

I believe the front-end user interface solves the problem of interacting with smart contracts in a simple and uncomplicated manner. There are very little user interface elements, and the application operates on just one single page. React.js allows for snappy components and automatic updating of data, while Material-UI allows for simplistic and easy to use components. Metamask fully handles all the interactions with the blockchain, allowing for a third-party API to abstract out this process further increases the security of the platform that would not have been as achievable without it. All the functionality of the auction smart contracts is abstracted into the UI elements displayed here.

6.2 Tools and Understanding

In completing this project, I was exposed to several unfamiliar technologies such as React.js and Solidity, along with difficult concepts such as the blockchain. In executing this platform, I believe I have adequately proven my proficiency in both the use and knowledge of these technologies.

6.3 Future Work

In future, this project could be greatly expanded with more functionality. The search functionality could be improved and refined using better search algorithms. The platform could also implement a Map API to locate the properties more effectively. The Application Binary Interface (ABI) of the smart contracts could also be made available publicly so that others can build their own API for interacting with the auctions. There are still many avenues that could be pursued to create a much larger and diverse software package. I do believe more work could be done to adequately stress test the platform, unit tests and use case tests on both the contracts and user interface were carried out. However, fully testing and evaluating the costs involved in deploying contracts would need to be performed along with testing for extreme use cases. Due to time constraints, it was not practical to perform any user studies, I believe this would lead to a much more refined user interface. More work also would need to be done to ensure proper form security.

Further on, a service could be implemented for investigation of reportedly fraudulent auctions and landlords. This would include tracking the movement of Ether between different accounts and analysing whether they had fraudulent involvement in the bidding and auctioning of the property. Although it would implement a centralised party into the platform and would be a massive undertaking it would be necessary for a fully-fledged service with legal responsibility to its users.

6.4 Conclusion

In conclusion, the implementation of this project was successful, fulfilling the requirements for making an open and transparent platform for rental auctions. The smart contracts implement the auction perfectly as anticipated, and the user interface provides a smooth experience for interacting with them. With the platform backed by blockchain technology and other decentralised solutions such as IPFS, this really exemplifies the idea of zero downtime unstoppable software. I believe this makes it extremely powerful especially for microservices who wish to provide smooth and uninterrupted services. A back-end on the blockchain and IPFS proves much more cost effective than current centralised services.

Although I believe this project expands onto a currently relatively barren practical realm of blockchain research, I do not feel that this has made much of a leap in terms of what is potentially possible. However, as this is a form of technology that has not yet fully emerged in the mainstream and is still within its youth/ early period of existence, I believe that by securing basic practical solutions such as this, it makes way for future innovation. There are plenty of foundations to build upon already, the concept of the blockchain is solid but there is, as of yet, no big-name tech companies building wide use decentralised applications.

References

[1] “MetaMask,” [Online]. Available: https://metamask.io/. [Accessed: 25-Mar-2019]

[2] Web3js, “web3.js - Ethereum JavaScript API — web3.js 1.0.0 documentation,” [Online]. Available: https://web3js.readthedocs.io/en/1.0/. [Accessed: 25-Mar-2019]

[3] S. Grider “Ethereum and Solidity: The Complete Developer's Guide | Udemy,” [Online]. Available: https://www.udemy.com/ethereum-and-solidity-the-complete-developers-guide/. [Accessed: 11-Nov-2018]

[4] Remix, “Remix - Solidity IDE,” [Online]. Available: https://remix.ethereum.org/. [Accessed: 23-Mar-2019]

[5] G. Wood, “ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER EIP-150 REVISION,” 2014. [Online]. Available: http://gavwood.com/paper.pdf. [Accessed: 09-Mar-2019]

[6] Z. Zheng, S. Xie, H. Dai, X. Chen and H. Wang, “An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends,” 6 2017. [Online]. Available: http://ieeexplore.ieee.org/document/8029379/. [Accessed: 05-Mar-2019]

[7] A. Hahn, R. Singh, C.-C. Liu and S. Chen, “Smart contract-based campus demonstration of decentralized transactive energy auctions,” 4 2017. [Online]. Available: http://ieeexplore.ieee.org/document/8086092/. [Accessed: 05-Mar-2019]

[8] N. F. Tavares, “Using Blockchain and Smart Contracts in a reverse auction syndicated e-procurement platform,” 2018. [Online]. Available: https://repositorio-aberto.up.pt/bitstream/10216/115625/2/287086.pdf. [Accessed: 05-Mar-2019]

[9] V. P. Ranganthan, R. Dantu, A. Paul, P. Mears and K. Morozov, “A Decentralized Marketplace Application on the Ethereum Blockchain,” 10 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8537821/. [Accessed: 04-Mar-2019]

[10] Solidity, “Solidity by Example — Solidity 0.4.25 documentation,” [Online]. Available: https://solidity.readthedocs.io/en/v0.4.25/solidity-by-example.html#simple-open-auction. [Accessed: 15-Nov-2018]

[11] MetaMask, “MetaMask Compatibility Guide,” [Online]. Available: https://github.com/MetaMask/faq/blob/master/DEVELOPERS.md. [Accessed: 26-Mar-2019]

[12] Reactjs, “React – A JavaScript library for building user interfaces,” [Online]. Available: https://reactjs.org/. [Accessed: 20-Mar-2019]

[13] Reactjs, “Virtual DOM and Internals – React,” [Online]. Available: https://reactjs.org/docs/faq-internals.html. [Accessed: 20-Mar-2019]

[14] Reactjs, “Introducing JSX – React,” [Online]. Available: https://reactjs.org/docs/introducing-jsx.html. [Accessed: 20-Mar-2019]

[15] M. Wöhrer and U. Zdun, “Smart Contracts: Security Patterns in the Ethereum Ecosystem and Solidity,” 20 3 2018. [Online]. Available: http://eprints.cs.univie.ac.at/5433/. [Accessed: 01-Dec-2018]

[16] J. Benet, “IPFS - Content Addressed, Versioned, P2P File System,” 14 7 2014. [Online]. Available: http://arxiv.org/abs/1407.3561. [Accessed: 12-Nov-2018]

[17] Fusejs, “Fuse.js - JavaScript fuzzy-search library,” [Online]. Available: https://fusejs.io/. [Accessed: 05-Mar-2019]

[18] J. Calderon, “solidity - How to convert string to int - Ethereum Stack Exchange,” [Online]. Available: https://ethereum.stackexchange.com/a/18035. [Accessed: 11-Nov-2018]

[19] Etherscan, “TESTNET Rinkeby (ETH) Blockchain Explorer,” [Online]. Available: https://rinkeby.etherscan.io/. [Accessed: 17-Feb-2019]

[20] Remix, “Remix-Tests,” [Online]. Available: https://github.com/ethereum/remix/tree/master/remix-tests. [Accessed: 03-Mar-2019]

[21] Jestjs, “Getting Started · Jest,” [Online]. Available: https://jestjs.io/docs/en/getting-started. [Accessed: 26-Feb-2019]

[22] airbnb, “Introduction · Enzyme,” [Online]. Available: https://airbnb.io/enzyme/. [Accessed: 27-Mar-2019]

[23] Selenium, “Selenium - Web Browser Automation,” [Online]. Available: https://www.seleniumhq.org/. [Accessed: 20-Mar-2019]

[24] IPFS, " The JavaScript HTTP client library for IPFS implementations." [Onlin]. Available: https://github.com/ipfs/js-ipfs-http-client [Accessed: 03-Dec-2018]