Auction smart contract
Manual

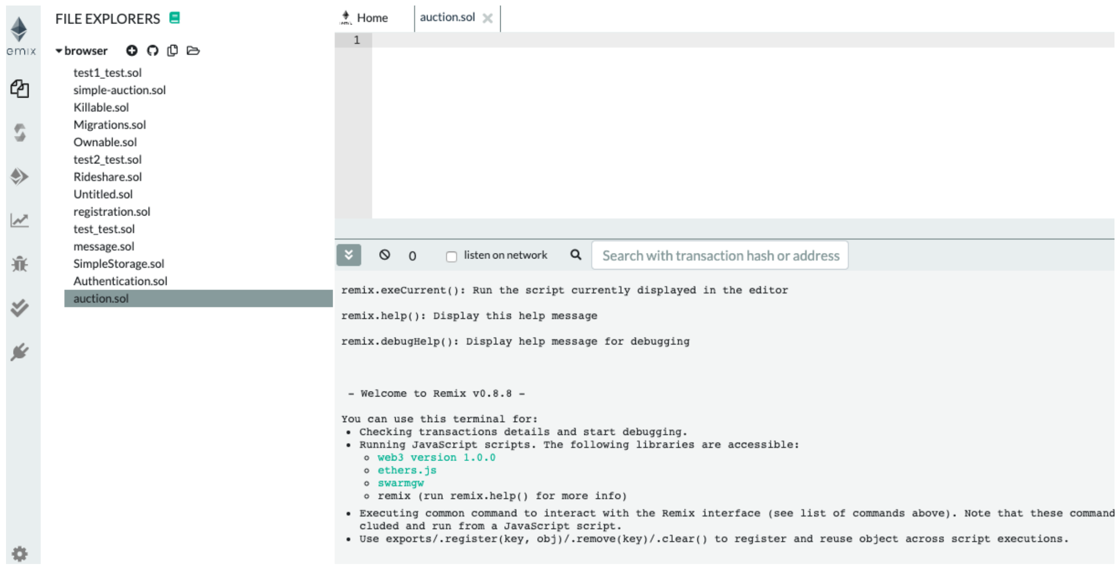
Blockchain based smart contract

Contents

Introduction:

In the previous project, we made a Voting poll Application using Ganache, the Remix IDE, and Solidity as our tools. In this project, we will be developing a simple Auction smart contract only using Remix IDE, Solidity and Ganache. Remix is an online IDE which is located at https://remix.ethereum.org/. The online IDE is used to compile our smart contract. The smart contract resides at a specific address on the Ethereum blockchain.

Ganache - Local in memory blockchain for development purposes. Installed at https://www.trufflesuite.com/ganache. This application gives you test accounts and private keys and can be used with Metamask, which is a way to connect your smart contract with a web application. Each account in Ganache holds 100 fake ether.



```
remix.exeCurrent(): Run the script currently displayed in the editor

remix.help(): Display this help message

remix.debugHelp(): Display help message for debugging



 - Welcome to Remix v0.8.8 -

You can use this terminal for:
 • Checking transactions details and start debugging.
 • Running JavaScript scripts. The following libraries are accessible:
    o web3 version 1.0.0
    o ethers.js
    o swarmgw
    o remix (run remix.help() for more info)
 • Executing common command to interact with the Remix interface (see list of commands above). Note that these command
   cluded and run from a JavaScript script.
 • Use exports/.register(key, obj)/.remove(key)/.clear() to register and reuse object across script executions.
```

Figure 1: the Remix IDE

After going to the website, click on the plus button in the File Explorer and make a new smart contract called “auction.sol”.

The idea of this smart contract is to host a live auction, so any person can send their bids during a certain paying period. Each person’s bid is sent using “Ether” which binds a specific amount of Ether to that bidder’s ID. When the bid gets raised, the previous bid is sent back to the corresponding bidder. When the bidding period ends, the contract is called manually for the beneficiary of the auction to receive the money.

System Design:

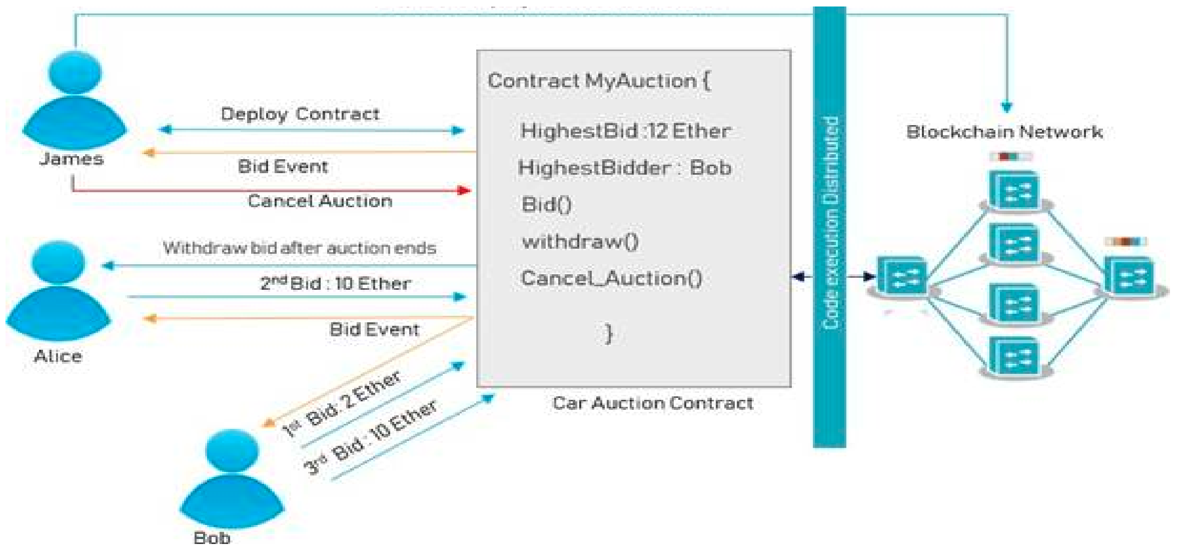


Figure 2: System design for the auction smart contract

From the first project, we already know how the blockchain network connects to your browser and front end application. This user diagram simulates 3 users sending their bid into the blockchain network, and the smart contract figuring out which bid is the highest, so who should recieve the auction prize. The code is distributed between “miners” which are the nodes on the blockchain network, and the transactions are processed and appended to the network.

The complete code for the “auction.sol” contract is below:

```
pragma solidity >=0.4.0 <0.6.0;
// accepts a bid from a bidder and stores the bid into a list,
// the bidder can recall or cancel their bids at anytime. Auctioneer can EndBid.

contract Auction {
    // the one auctioning the item
    address public Auctioneer;
    // the name of the Auction
    string auctionName;

    struct Bid {
        // bid is connected to an address
        address payable bidderAddress;
        uint amount; // value variables in Solidity are always stored as the wei value
        string name;
        bool valid; // is the bid valid
    }
    // array of bids
    Bid[] public bids;
     // set the name of the auction
    constructor(string memory _name)
    public {
    auctionName = _name;
    Auctioneer = msg.sender;
    }
     // returns how many bids have been made on the item
    function getNumberOfBids()
    public
    view
    returns (uint numberOfBids) {
        return bids.length;
    }
     // returns the ID of the highestBID
    function getHighestBidID()
    public
    view
    returns (uint bidID) {
        // there has to be atleast one bid placed
        require(bids.length > 0);

        // finds the highest bid's ID #
        uint highestAmount = 0;
        uint highestID = 0;
        for (uint i=0; i<bids.length; i++) {
            if (bids[i].amount > highestAmount) {
                highestAmount = bids[i].amount;
                highestID = i;
            }
        }
        return highestID;
    }
// call this function make a Bid for the proposed function
    function makeBid(string calldata _name)
    external
    payable
    returns(uint bidIterator) {
        require(msg.value > 0);
        bids.push(Bid(msg.sender, msg.value, _name, true));

        return bids.length - 1;
    }
// returns the value of the bid to the specified address of the msg.sender
    function recallBid(uint bidIterator)
    external {
        require(bidIterator < bids.length, "Invalid bidIterator");
        require(msg.sender == bids[bidIterator].bidderAddress, "msg.sender !=
bidderAddress");
        require(bids[bidIterator].valid, "Bid is invalid; aborting");

        uint amountToSend = bids[bidIterator].amount;
        bids[bidIterator].amount = 0;
        bids[bidIterator].valid = false;
        bids[bidIterator].bidderAddress.transfer(amountToSend);
    }


}
```

Figure 3: complete code for the smart contract

Now I will delve into how the Smart contract works.

We first declare variables which are parameters of the auction. We track the state of the auction, so we can find the highest bidder and the highest bid amount. We use a mapping variable to keep track of the withdrawls of the previous bids which aren't the highest bid. The Event variables are used when the state of the auction is changed.

```
Pragma solidity >=0.4.0 < 0.6.0;
Contract Auction {
// the one auctioning the item
    address public Auctioneer;
    // the name of the Auction
    string auctionName;

    struct Bid {
        // bid is connected to an address
        address payable bidderAddress;
        uint amount; // value variables in Solidity are always stored as the wei value
        string name;
        bool valid; // is the bid valid
    }
    // array of bids
    Bid[] public bids;
}
```

Figure 4: Smart contract declarations and variables

Now we will define the methods of the smart contract.

The algorithm for the bid function is:

1. Check if the bidding time is elapsed, if so then throw an error. Check for the highest bid by checking each and every amount for each bid in a for loop.
2. If the bidding amount is less than the highest bid then throw an error, which returns the funds back to the bidder
3. We update the pending returns for the last highest bidder to send the money back to the previous bidders
4. Lastly, we update the highest bidder and highest bid.

```
      // returns the ID of the highestBID
    function getHighestBidID()
    public
    view
    returns (uint bidID) {
        // there has to be atleast one bid placed
        require(bids.length > 0);

        // finds the highest bid's ID #
        uint highestAmount = 0;
        uint highestID = 0;
        for (uint i=0; i<bids.length; i++) {
            if (bids[i].amount > highestAmount) {
                highestAmount = bids[i].amount;
                highestID = i;
            }
        }
        return highestID;
    }
```

Figure 5: the highestBid contract

The algorithm for the withdraw function is:

1. Check whether the bid count is less than the bids array length
2. If there's no entry in the returns then return false.

3. If there is an entry then return true and transfer the amount of the bid to the bidder who called the function.

```
   // returns the value of the bid to the specified address of the msg.sender
    function recallBid(uint bidIterator)
    external {
        require(bidIterator < bids.length, "Invalid bidIterator");
        require(msg.sender == bids[bidIterator].bidderAddress, "msg.sender !=
bidderAddress");
        require(bids[bidIterator].valid, "Bid is invalid; aborting");

        uint amountToSend = bids[bidIterator].amount;
        bids[bidIterator].amount = 0;
        bids[bidIterator].valid = false;
        bids[bidIterator].bidderAddress.transfer(amountToSend);
    }
```

Figure 6: to recall the bid back to the bidder

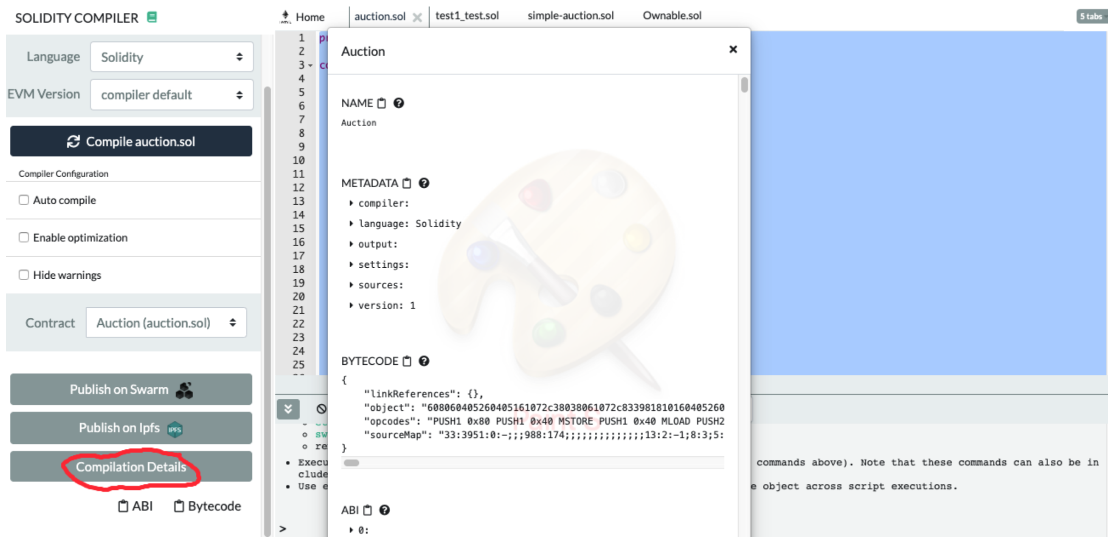
The last method of the smart contract allows us to finish the bid and send then money to the bidder of the auction.

The endAuction method, ideally first checks conditions of whether the start time is before the bidding time or not. Next we perform actions, which are things like setting the “ended” variable to true, and call the endAuction event, if the auction is indeed over.

```
function endAuction() public {
        selfdestruct(msg.sender);
    }
```

Figure 7: the “endAuction” method

You should have Copied and pasted the smart contracts code into your remix IDE, leave all the default values and press on the compile button and then press on the “compilation details” button, which is compiled below.



Auction

NAME

Auction

METADATA

```
▸ compiler:
▸ language: Solidity
▸ output:
▸ settings:
▸ sources:
▸ version: 1
```

BYTECODE

```
{
    "linkReferences": {},
    "object": "608060405260405161072c38038061072c833981810160405260
    "opcodes": "PUSH1 0x80 PUSH1 0x40 MSTORE PUSH1 0x40 MLOAD PUSH2
    "sourceMap": "33:3951:0:-;;;988:174;;;;;;;;;;;;;13:2:-1;8:3;5:
}
```

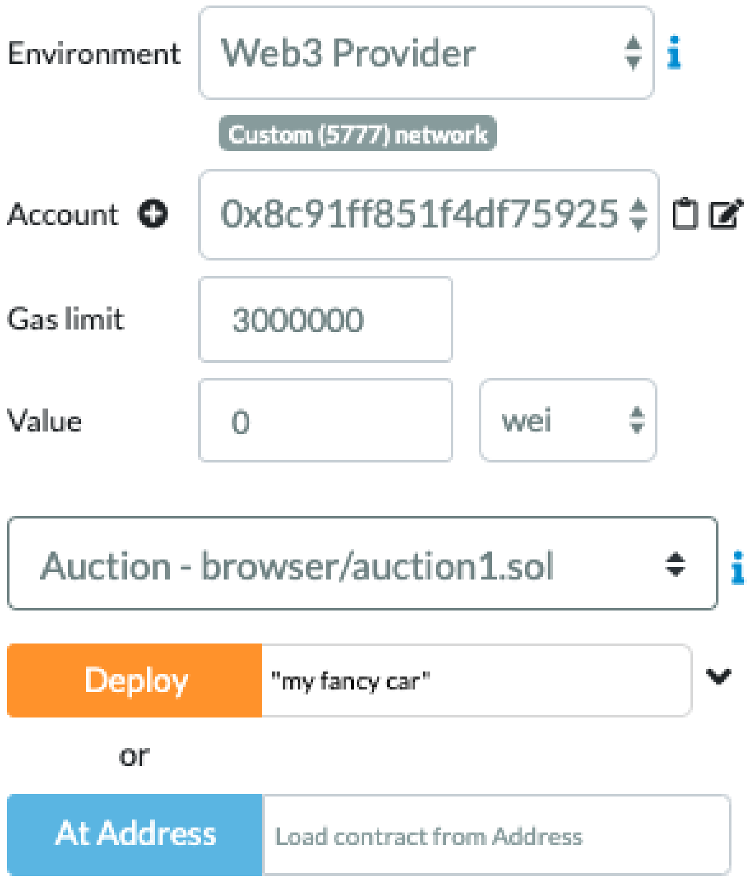
ABI

```
▸ 0:
```

Figure 8: the Remix IDE compiler

The Compilation details contain metadata about the defined smart contract. It has details about the smart contract, whether it is payable, the gas cost estimates, the version of the compiler, and even shows how the compiler translates solidity to assembly code. This information is useful to understand how Remix IDE works and know more about Solidity and how your smart contract works.

Next we actually deploy the smart contract to our local Ganache server. The details for how to do it are described below as well as included in the picture below. Make sure your Ganache is running with your 10 accounts, and switch the Environment to “web3 Provider”. Select yes to connect to your ethereum node and make sure to connect to the same port that Ganache is running on, usually “7545”. This menu is shown in Figure 9.


Environment Web3 Provider

Custom (5777) network

Account 0x8c91ff851f4df75925

Gas limit 3000000

Value 0 wei

Auction - browser/auction1.sol

Deploy "my fancy car"

or

At Address Load contract from Address

Figure 9: the Remix IDE deploy and run transactions

Next enter a name for what you are auctioning and press the “Deploy” button, or copy the account address and deploy it with the “At Address” button. Press on “transact” and you will see the auction contract is created and deployed in the console log.

When we look below, we see the functions that our smart contract contains. You can now simulate your own auction by putting in values for the methods, start with typing a name and pressing bids.

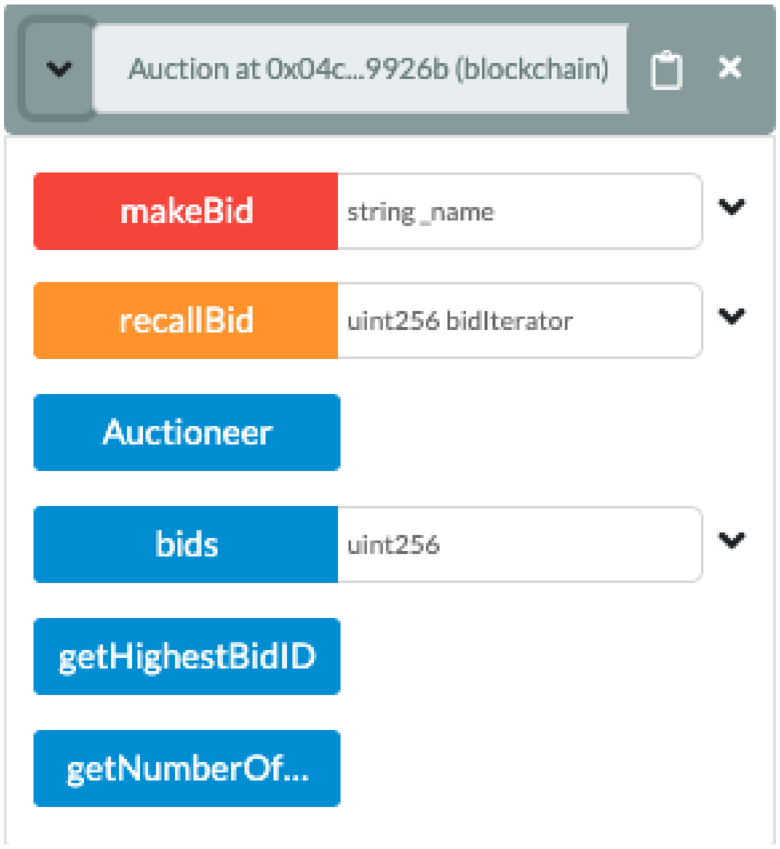
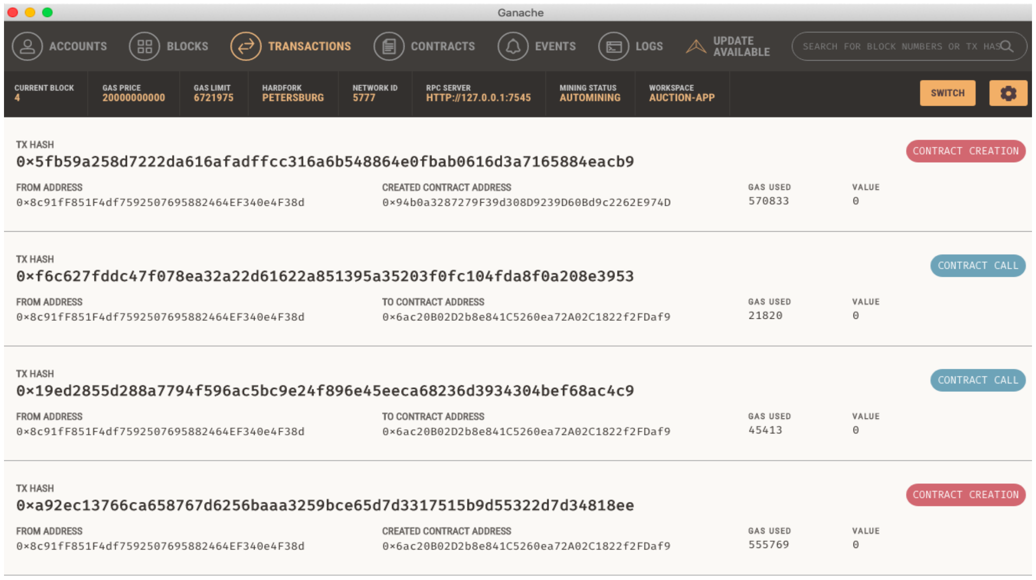


Figure 10: the Remix IDE smart contract deployed and opened

Also don't forget to change the value section under the "Gas limit" add a value to this section, which is the amount of ether that you are bidding, with the account that is selected. You should have an good understanding of the contract's code by now to understand how to simulate the Auction using the figure below.

In Ganache it will also show you the transactions that were appended to the blockchain network through the Remix IDE, if you head over to the “transactions” section of Ganache.

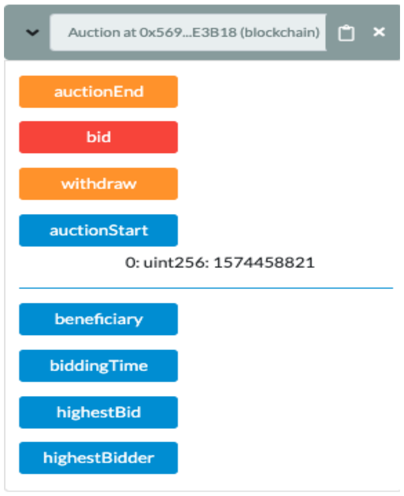


CURRENT BLOCK 4 | GAS PRICE 20000000000 | GAS LIMIT 6721975 | HARDFORK PETERSBURG | NETWORK ID 5777 | RPC SERVER HTTP://127.0.0.1:7545 | MINING STATUS AUTOMINING | WORKSPACE AUCTION-APP

```
TX HASH
0×5fb59a258d7222da616afadffcc316a6b548864e0fbab0616d3a7165884eacb9                CONTRACT CREATION
FROM ADDRESS                                  CREATED CONTRACT ADDRESS                      GAS USED   VALUE
0×8c91fF851F4df7592507695882464EF340e4F38d    0×94b0a3287279F39d308D9239D60Bd9c2262E974D    570833     0

TX HASH
0×f6c627fddc47f078ea32a22d61622a851395a35203f0fc104fda8f0a208e3953                CONTRACT CALL
FROM ADDRESS                                  TO CONTRACT ADDRESS                           GAS USED   VALUE
0×8c91fF851F4df7592507695882464EF340e4F38d    0×6ac20B02D2b8e841C5260ea72A02C1822f2FDaf9    21820      0

TX HASH
0×19ed2855d288a7794f596ac5bc9e24f896e45eeca68236d3934304bef68ac4c9                CONTRACT CALL
FROM ADDRESS                                  TO CONTRACT ADDRESS                           GAS USED   VALUE
0×8c91fF851F4df7592507695882464EF340e4F38d    0×6ac20B02D2b8e841C5260ea72A02C1822f2FDaf9    45413      0

TX HASH
0×a92ec13766ca658767d6256baaa3259bce65d7d3317515b9d55322d7d34818ee                CONTRACT CREATION
FROM ADDRESS                                  CREATED CONTRACT ADDRESS                      GAS USED   VALUE
0×8c91fF851F4df7592507695882464EF340e4F38d    0×6ac20B02D2b8e841C5260ea72A02C1822f2FDaf9    555769     0
```

Figure 11: the Remix IDE smart contract deployed and opened

Each smart contract has its own unique ID, which can be used to interact with the contract as well. My smart contract’s ID is “0x569155E8d43746794584a6CcB958e126392E3B18”.



```
0: uint256: 1574458821
```

Figure 12: Another hypothetical Auction application involving Auction time.

Critical analysis:

The transition from having a full stack application, to simply testing the code in an online IDE was the decision that ultimately took up the most time because of our lack of knowledge. We also used more time switching from Neo to Ethereum when we realised that the support documentation for Neo was mainly in Chinese and because of the installing issues a group member had with their MacBook. Our project management methodology involved more planning than trialling and this ended up affecting our project negatively, if we had trialled more while planning we would've realised sooner that Ethereum was the better option and therefore could've spent more time on the application itself. Each group member brainstormed design options, ultimately we converged the concepts into something simple, yet aesthetically pleasing. We choose to converge our ideas instead of picking the best one because we wanted to focus more time on creating on the application itself rather than perfecting the design, and converging was the simplest solution. Our developed system met the initial requirements that our supervisor had originally set out for us, we weren't however able to finish the extra requirements that were suggested towards the end.

Properties

Confidentiality - the account that we are connecting to the blockchain with is "our account" and is a lot of random numbers and letters that are generated by ganache as the public key. Private key is inputted into metamask to do that.

Integrity - the integrity of the data could be preserved by making use of HMAC function in solidity. Adding sha256 function for integrity of data

Authenticity - HMAC sha 256 function is generated and the key is verified. The authorization of the keys comes from the "Ganache" application.

Why Blockchain?

An auction is a good smart contract Idea because auctions follow a certain amount of rules that can be automatically and effortlessly follow with a smart contract. The smart contract will ensure that a bidder receives ther bid right away from withdrawing. The user's name can be hidden from the auction in total so nobody knows who is auctioning. The address is made from the Ganache node, and is a much safer value to use.