

# **An evaluation of the feasibility of a blockchain solution for a food vendor platform.**

CM3203 One Semester Individual Project

Final Report



George Goodall

Supervisor: Dr George Theodoakopoulos

Moderator: Dr David W Walker

# Table of Contents

Introduction.....	3
Background and related work.....	4
What is a Blockchain (Bitcoin).....	4
What is a Smart Contract (Ethereum).....	4
The Current State of ethereum Dapps.....	5
Design.....	5
Development Tools.....	5
Choice of blockchain: public vs private.....	5
Choice of blockchain: ethereum vs EOS.....	6
Language and Framework.....	7
Interfacing with the Blockchain: contract migration.....	7
Interfacing with the Blockchain: client application.....	7
Architecture.....	8
System Flow.....	8
Planned Architecture.....	8
Challenges.....	10
Implementation.....	11
Overview.....	11
Account Creation.....	11
Controller Contract.....	11
Account Contracts.....	11
Customer Contracts.....	12
Rider (Delivery Worker) Contract.....	12
Restaurant Contract.....	12
Order Contract.....	12
User Interface.....	12
Server Code.....	12
Testing and maintenance.....	13
Application Walk-through.....	15
Challenges.....	18
Evaluation.....	20
Transaction fees.....	20
Transaction Times.....	22
Future Work.....	22
Improvements to my system.....	22
Improvements to the ethereum network.....	23
Conclusion.....	24
Reflection.....	24
References.....	25

# Introduction

I will be evaluating the benefits and downfalls of using blockchain technology in the implementation of a food vendor platform similar to just eat or deliveroo when compared to implementation via a more typical approach using a centralised server. My primary motivation for this is to provide a system that removes the high fees charged by “middle man” companies, these companies take a fee for connecting and facilitating transactions between two parties, some examples of these industries include food vendors, peer-to-peer ride sharing (Uber), peer-to-peer e-commerce (eBay), freelancer websites (upwork) and many more.

According to the Just eat website [1] their fees for restaurants are a one-off joining fee of £699 and a commission of 14% on every order, some of these fees will be made back in savings on several of the perks offered by Just Eat such as energy bill savings, cashback from specific wholesale stores among other perks. The fee to the customer is a 50p admin charge on every order.

For this, restaurants get advertised on the just eat website, a tablet and printer for receiving and managing orders, personalised data insights, cashback and savings made on services such as wholesale retailers, energy bills, delivery bikes and scooters.

I hope to replicate the core functionality of a food vendor platform on the blockchain, I will then evaluate how fit it is compared to current centralised systems and make a conclusion on if the blockchain based solution can compete with current centralised systems in respect to fees charged, ease of use among other aspects of the system. Though I will be producing a UI to interface with my blockchain back end I won't be evaluating the aesthetic of the UI as it not only falls outside the scope of my project but also with the blockchain back-end being public to anyone I would expect several user interfaces to be produced by different developers that interface with a blockchain back-end. I will, however, be evaluating the ease of interfacing with a blockchain back end and potential solutions to deal with the asynchronous nature of a blockchain.

## Background and related work

### What is a Blockchain (Bitcoin)

In 2008 a person or group of people using the name Satoshi Nakamoto produced the first blockchain protocol, the bitcoin protocol. This protocol was built on top of the ideas of Stuart Haber and W. Scott Stornetta proposed in a paper from 1991 titled “How to Time-stamp a Digital Document” [2] in which they outline a system that would prevent document timestamps from being tampered with.

Satoshi implemented a decentralized and distributed ledger which can record transactions across several computers without the need for a central server. It does this by keeping a record of all transactions held together in groups also known as blocks that are chained together to create a blockchain. The transactions on the block chain occur between accounts known as addresses, where each address is a public, private key pair, in the case of bitcoin, elliptic curve cryptography is used to generate this key pair. Each transaction within a block is signed using the sender's address' private key to ensure that only the sender can spend their bitcoin. Each new block generated will approve a previous block by including a hash pointer to that block, chaining the blocks together.

## What is a Smart Contract (Ethereum)

In 2013 Vitalik Buterin proposed that bitcoin needed a scripting language for application development, however failing to gain agreement he proposed the development of a new platform with a more general scripting language. ethereum was then announced in early 2014.

ethereum is similar to bitcoin only it has the ability to store and execute smart contracts on the blockchain, where a smart contract is some code that can be run without any human interaction. One of the main uses for a smart contract which I will be taking advantage of in my project is the notion of an automated escrow, this allows code to store value in the form of cryptocurrency until a set of requirements have been met, at this time the funds can be released to the appropriate parties deemed by the publicly viewable smart contract code.

Technically Bitcoin has the ability to store and run smart contracts also, however it is not considered Turing complete due to the halting problem. The issue comes from the fact that any transaction has to be verified by others in the network if a transaction has an infinite loop the whole network could slow down. Bitcoin resolves this by simply removing the programming controls that would allow a program to run forever, for example, loops.

ethereum however in the yellow paper [3] describes itself as “quasi-Turing-complete” they go on to explain that though the ethereum virtual machine (EVM) would be Turing complete if not for each transaction being “intrinsically bounded through a parameter, gas”. Every operation within the ethereum Virtual Machine has a specified gas cost to execute. This gas will then be given to the miners of the network as a reward. Upon sending a transaction to the network the gas usage will be estimated, the sender will then specify their own gas price in wei ( $10^{-18}$  Eth). The Gas sent with the transaction will then be the estimated gas usage multiplied by the specified gas price. Setting a lower gas price will save the sender transaction fees however will also cause the transaction to be picked up and processed slower by the network. According to the website “Eth gas station” the current standard gas price is 3 Gwei or  $3 \times 10^{-15}$  Eth.

## The Current State of ethereum Dapps

A dapp or decentralised application is an application that has its back end code running on a decentralized peer-to-peer network instead of a centralised server.

Currently, most active dapps tend to have purposes that remain virtual and don't extend to the real world, with most of the top dapps being under the category of Games, Wallets, Advertising, Exchanges, Gambling or data storage. [5] I did, however, find some companies that offered some interesting Business models.

Eva is a Canadian company who are working towards building a decentralised uber like platform on the EOS blockchain. [6] They released the beta version of their application for android and ios in October of 2018 in Canada, and though news on its progress since then has been hard to come by, it seems it is mostly liked with a rating of 3.9/5 of google play.[7]

Swarm.city is an ethereum based application that is attempting to build a general peer-to-peer marketplace. It says that it could be used to develop many other applications such as a lift sharing application, a peer to peer bed and breakfast platform or a freelancing platform. However it appears that the use of the project has diminished largely with the recent fall in the value of cryptocurrency, further development to the system doesn't seem to be progressing as fast as it once was.

Most of the drive for development of decentralised applications has subsided during the crash of bitcoin over 2018. On the ethereum blockchain, very few applications that have an impact of the physical world have been developed with most only having virtual utility for example in games and token exchanges.

# Design

## Development Tools

### Choice of blockchain: public vs private

When considering the tools I will use to develop my application, the first and most important decision to make would be the choice of blockchain protocol to use. Firstly I must consider what is a better fit for my application, a public blockchain or a private blockchain.

Private blockchains like hyperledger and ripple restrict who can access the closed network, they are more centralized as only a few miners validate the blocks so trust is needed in those miners however due to the limited number of miners consensus takes less time meaning the network runs faster and is more scalable.

Public blockchains like bitcoin, ethereum or eos are far more decentralized, anyone with sufficient hardware could set up their own node and contribute towards maintaining the network. Because of this the network is considered trustless, more secure and offers complete transparency which can be a good thing but makes the storing of private/personal data on a public blockchain basically impossible. Also, the larger number of nodes cause consensus to take longer, leading to slower network speeds and scalability concerns among higher energy consumption with proof of work based algorithms.

A private blockchain wouldn't fit with what I'm trying to achieve, it offers too much of a centralized model that wouldn't work in the peer-to-peer style of application I'm attempting to make. A public blockchain is completely decentralized so would offer peer-to-peer transactions however the high transaction times and throughput may largely impact the user experience. These issues caused by the high transaction times might be able to be resolved by using intelligent asynchronous UI design among other design patterns

### Choice of blockchain: ethereum vs EOS

For my choice of public blockchain, I'm considering ethereum and EOS. Both of these protocols are geared around the ability to write smart contracts and develop dapps however there are some key differences such as the consensus algorithm, availability of documentation and online support, and the capabilities of the existing frameworks that I should consider before making my decision.

EOS doesn't charge fees for any computation you request on the network. Instead, you are entitled to a proportion of computational power from the network based on the proportion of total EOS tokens you hold. So if you own 1% of all tokens you are entitled to 1% of the networks computational power, this holds true for CPU and GPU usage, however, ram is considered far more of a valuable resource and is therefore treated differently. Users of the network can buy ram from a ram pool smart contract. The price of ram is automatically set by the smart contract which uses the Bancor algorithm, this pricing algorithm basically ensures that there is always ram available for purchase because as the amount of available ram approaches 0 the cost of ram approaches infinity. Currently, new accounts need 4 to 8 Kb of ram to use the network costing at this time about 0.82 – 1.64 dollars.

EOS uses delegated proof of stake as the consensus protocol. What this entails is a protocol in which participants can cast votes to elect the 21 block producers. If a block producer is honest and earns the trust of the network they accumulate enough votes to remain a block producer. If they act dishonest and abuse their position the network's trust in them will fall and another block producer will be voted in. Each staked token corresponds to one vote for the network. Currently, EOS has an annual inflation rate of 5%, 1% of this goes towards rewarding the block producers while 4% goes into a pool that is allocated as payment to developers who wish to improve the network protocol. Because the EOS network only has 21 block producers, consensus takes a lot less time to be

achieved. This leads it to have the largest operations per second rate of above 19000000 operations per second. [13]

Ethereum's protocol requires a fee (known as gas) to be paid, this fee is proportional to the computational power needed for the transaction. While you could argue that EOS offers a cheaper service to any dapp user due to no fees, the cost has to be paid by the dapp developers which will eventually be passed on to the end user.

Ethereum also differs from EOS in its consensus algorithm. Ethereum is currently using a proof of work algorithm similar to Bitcoin, this offers some positives such as better security due to a higher number of block validators, meaning an attacker would need to control more nodes to accomplish a 51% attack, an attack where the attacker controls 51% of the network allowing them to decide what blocks are added to the blockchain. However, the higher total block validators cause network consensus to take longer. Meaning transaction time is higher, throughput is lower at about 680000 operations per second [13] and the total network energy consumption is higher.

Ethereum is far more established than EOS, with Ethereum being released July of 2015 and EOS being released January of 2018, Ethereum has far better support and developer tools than EOS. Both blockchains have front end libraries that enable interfacing with the blockchains, however again due to the extended time Ethereum has been out for, support, documentation and functionality of these libraries are better than that of EOS.

Though EOS has a higher rate of transactions per second, I feel that the longer track record, better developer support and improved security sway me to choose Ethereum as the platform on top of which I will develop my application.

## Language and Framework

Ethereum offers several languages and compilers that are able to compile code into bytecode that can be run by the Ethereum virtual machine, Solidity is currently the most popular language, it is a JavaScript based language and offers the most support. Other languages exist such as Serpent, which is based on Python however due to Serpent being less popular, documentation is harder to come by. For this reason among the fact that I am more comfortable developing in JavaScript compared to Python. I will be using Solidity to develop my smart contract based back end.

Initially, I used the online IDE Remix [14] to begin developing my smart contracts, Remix offered a very easy entry into the development of smart contracts as you can compile and run tests all in the browser without having to install any other software. I was able to prototype several of the basic systems I would be employing, while just getting a feel for Solidity. Eventually, my project began to outgrow Remix however, the IDE didn't offer the ability to develop GUIs and working on several larger contracts was becoming more troublesome.

I then moved over to Truffle, a framework that was developed to assist Ethereum developers to build and deploy smart contracts, they offer tools to compile, test and deploy your contracts to either the Ethereum main blockchain, any of the test blockchains or a local running blockchain. Initially, I would deploy my contracts to a local running blockchain, this was supplied by a tool released with the Truffle framework known as Ganache that sets up a local blockchain in memory. I went on to deploy my contracts to the Ropsten test network eventually however to evaluate the effectiveness of my solution on a public facing chain.

## Interfacing with the Blockchain: contract migration

Initially, when attempting to migrate my contract to the Ropsten test network I tried syncing my PC with the Ropsten blockchain. This proved to be quite difficult due to the large size of the chain. I opted to use Infura instead to resolve this. Infura is a collection of Ethereum full nodes that can be used to send transactions to the blockchain without the need for a local full node [18]. First, a transaction is signed on by the user, then sent to Infura who then using their full node broadcast the transaction to the blockchain.

## Interfacing with the Blockchain: client application

There are only a few ways someone can send a transaction to the blockchain:

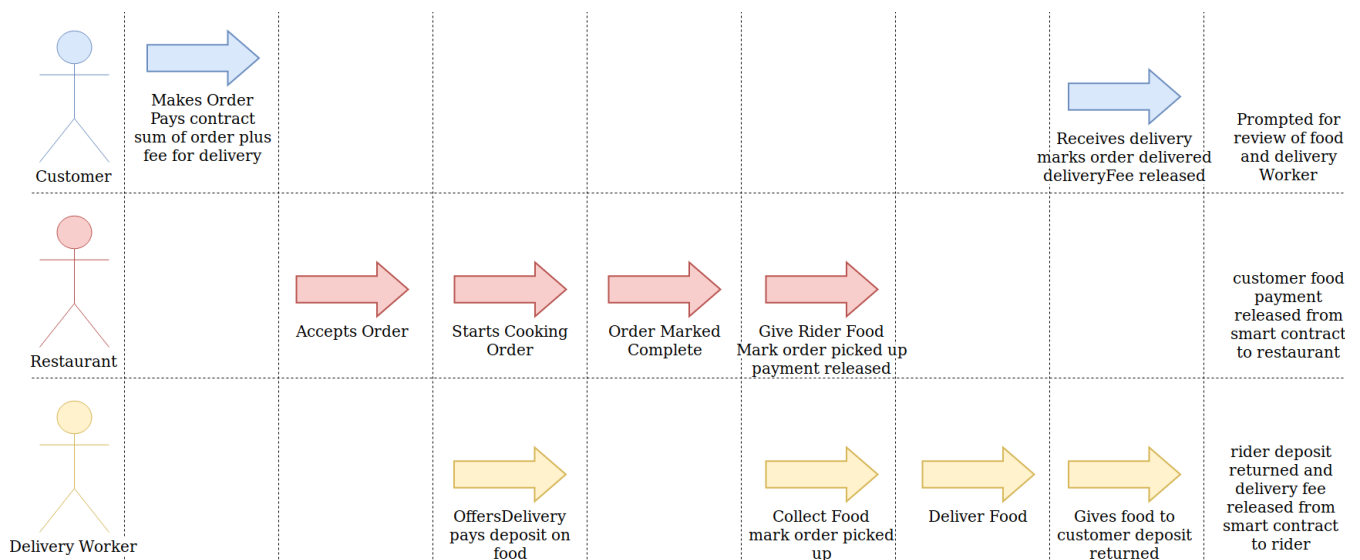
1. Running a full node: this means that you have to have a copy of the whole blockchain saved on your device. Constantly updating and verifying transactions, this method is very secure however requires good hardware and a good network connection.
2. Running a light node: a light node unlike a full node doesn't download the whole blockchain, but instead just the block headers so they can validate the authenticity of each transaction. They are dependent on full nodes to serve them information, and without full nodes could not function.
3. Connection with a 3<sup>rd</sup> party node: a service such as infura is used to interface the user with the blockchain. This is the least secure as no validation is done by you, so any false data received that was sent by the full node or by an attacker who intercepts and changes the contents of a block will be trusted. Though they are easy to use as they require no syncing with the blockchain.

For my application I have decided to use metamask as a method of storing keys, metamask is a browser-based wallet that connects to a full node using infura. I can then use Web3, a javascript API that provides several libraries for interfacing with a blockchain, metamask will automatically create a web3 connection that I can access to make transactions using the keys stored in metamask.

## Architecture

### System Flow

When beginning to design my systems architecture I thought it best to first try to describe the flow of my system, that being all the primary ideal actions of all the actors. This would help me start to construct my contracts and the interactions between them.



The above diagram shows my system flow. It shows the three agents that will be involved within any one order, the ideal actions taken by them and then the resulting actions issued by my smart contract once all conditions are met.

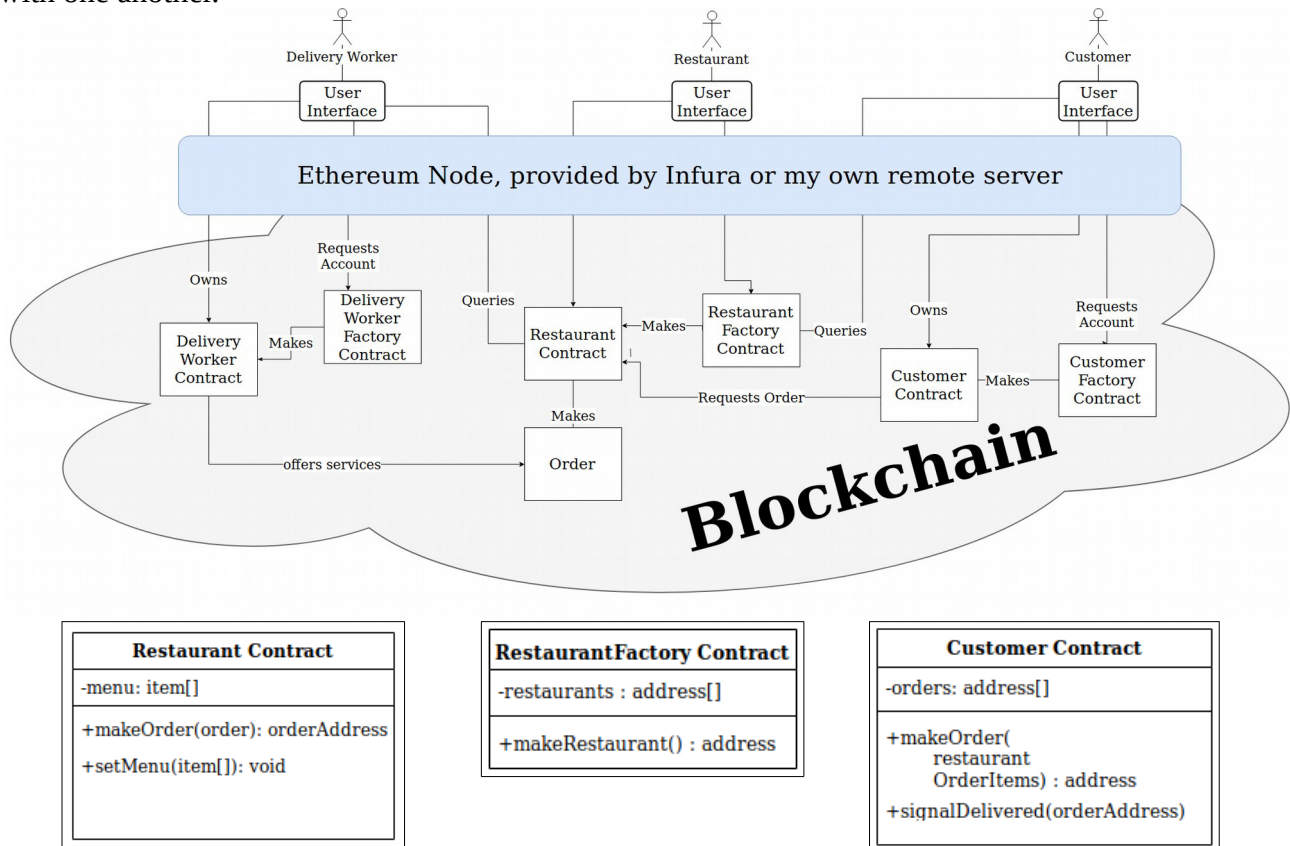
The considering of my system flow has highlighted several design features that I should implement. For one, I realised I needed a way to deter delivery workers against bad practices such as keeping the order for themselves when they should deliver it. I will do this by enforcing a deposit system in which the delivery worker must pay a deposit at least equal to the cost of the order.

Another design feature revolves around the need to create a system for which there are three separate agent types, all of which may have an abundance of new users wishing to create an account of that type. For this reason, I will design my system to make use of the factory contract design

pattern. This pattern involves deploying a factory that is responsible for the creation of other contracts. In my case, I will have a customer factory, a restaurant factory and a delivery worker factory. Using this pattern will allow for easy creation of new accounts while also providing a place to store information such as the addresses of the smart contracts created.

## Planned Architecture

After understanding the basic flow of my system I felt it would now be best to draw up a basic system diagram, specifying the contracts, their variables and methods and how they would interact with one another.



As the management and creation of user accounts would play quite a large role in my program I decided to use the factory contract design pattern, a new customer, delivery worker or restaurant could create an account by calling the respective factory contracts make factory function. Also, I can use these factories to store references to active user contracts for querying so, for example, a user or delivery worker could get all restaurants by checking the list of restaurant addresses within the restaurant factory.

Once a customer has created their account they would query the restaurant factory to find all restaurants, they would then make an order to the restaurant by making a transaction to their own contract, sending a payment for the order including the delivery fee and referring to the restaurant address and items they wish to order. The customer contract would then pass on the payment and order information to the restaurant contract at the address specified which would then create an order smart contract and store the customer's payment within this contract.

Delivery workers can query in turn first the restaurant factory to obtain all restaurants, then each restaurant to obtain all orders. The delivery worker can then offer to deliver the order by depositing an amount equal to that of the order value into the order contract.

The restaurant will update the status of the food preparation so that the delivery worker knows when to arrive for collection, at this point both delivery worker and restaurant would notify the order contract that the food delivery has been handed over. The delivery worker will then deliver the food to the customer's address and then again both parties, delivery worker and customer will



signal the food has been handed over. At this point, funds will be released into the appropriate accounts.

## Challenges

Though I believe my architecture can provide a system that offers the core functionality of a food delivery service, there are some Challenges that need to be resolved to largely improve the quality of my end product. The main two being the high transaction times for the ethereum network, as well as the storage of private user data.

All write operations to a blockchain take time to be processed by the network, for ethereum the block time is around 14 seconds [15] however this isn't consistent and blocks can take longer to process. A user may want to wait for several confirmations also to ensure double spending doesn't occur increasing the transaction time further. Considering all the transactions that will be made by users of my application not all of them will require an immediate response, for example, creating an account, making an order or updating an order status doesn't require an immediate response. Though admittedly an immediate response would increase the user-friendliness of my system. Some other transactions exist where the delay in transaction time could pose a serious problem. These include the transactions that occur during cargo hand over, such as when the restaurant gives the food to the delivery worker and when the delivery worker gives the food to the customer. Having the restaurant and delivery worker wait for the transaction to process and be confirmed before handing over the food will significantly reduce the appeal of my application against other applications that use more traditional application back ends with no waiting times. Both parties can't risk handing over the food prior to this transactions confirmation due to the risk that the other party simply won't notify the system that they have the order causing no payment to be released. Another transaction that might cause problems due to the high transaction times is the offer made by a delivery worker to deliver an order. If two delivery workers were to offer delivery of an order at a similar time, one of them is going to be the first to get delivery while the other is going to have their deposit locked up until their transaction inevitably fails due to a rider already being assigned, at which time they may lose some funds for the gas paid for any computation that occurred prior to their transaction failing.

# Implementation

In this section I will describe my actual implemented program architecture, giving examples of key design choices and explaining the motivations behind them.

## Overview

### Account Creation

new users can create accounts by invoking the make account function from one of the three factory smart contracts, for example, if a new customer wishes to make a new account, they will invoke the makeCustomer function from the customer Factory contract. This function first checks to make sure they don't already have a smart contract linked with their address, if they don't my factory creates a new customer contract and saves the address of this contract. It then emits an event indicating an account has been made that my frontend can detect to cause the UI to update.

## Controller Contract

After some initial testing on my factory contracts and the account contracts they produced, I noticed each contract was building an increasing number of addresses referencing other contracts in my system. I decided to implement the name registry design pattern, the name registry design pattern involves making use of a contract specifically for referencing other contracts, this meant adding a contract that I named controller, that stores references to the factories within my system. This way contracts could simply query the controller to get the factory addresses. The controller contract also acted as a good place to store some system-wide variables and methods such as a minimum delivery fee variable and a hashing function.

## Account Contracts

As my customer, restaurant and delivery worker contacts share a lot in common I will discuss the key parts of their design here and then go into each in more depth. Firstly, each of the account contracts implements the ownership design pattern where they all have a variable specifying the owner's address. This is then used across several of the functions within the contract to ensure that only the owner is able to call these functions. The owner address is also the recipient of any payments the contract may receive.

The account contracts also store references to the order contracts that they are involved with. This allows the retrieval of relevant orders.

## Customer Contracts

The customer contract stores some non-private information on the customer and provides functionality for that customer to request an order from a restaurant by paying an amount of eth into the new order contract. Customer contracts are also able to interact directly with the order contract in some ways by specifying that they have received the order.

## Rider (Delivery Worker) Contract

The rider contract is very similar to the customer contract however instead of being able to invoke a restaurant to make an order, it can interact with an existing order and offer to deliver the food from restaurant to customer. This function call requires a deposit of eth larger than the value of the order. The delivery worker can also update the order to specify their status. For example, if they have paid the deposit, have picked up the food or have finished with the delivery.

## Restaurant Contract

The restaurant contract is a restaurants interface with my system. It stores information on the restaurant including the menu and prices. It also provides the functionality to update the menu as the restaurant sees fit. The most important function in this contract is the makeOrder function. It can only be called from a customer contract and leads the creation of a new order contract.

## Order Contract

The order contract is the largest contract within my system, it stores and handles the information on the order, acts as an intermediary between users by providing the functionality for different agents involved in this order to specify their current progress towards the goal state, and also acts as an escrow, storing all agents funds and then releasing these funds to the appropriate parties when the goal state is met. (the goal state being that the food has been delivered to the customer)

## User Interface

The user interface was developed as a simple website, using html, css and javascript. I spent very little time working on the aesthetic of my UI. Instead, opting to prioritise how best for my UI to cope with the asynchronous nature of my blockchain back end. This was done using promises and the await keyword for method calls. Programming the user interface in a way to optimise data acquisition was important, calling asynchronous methods in parallel significantly reduced the loading time for my application.

## Server Code

for my server I use a nodejs express server, it was responsible for serving website content, handling requests to read or update personal information and obtaining the price of eth for price conversions.

## Testing and maintenance

During the development process, it is important to know that the changes I make to one part of my software doesn't have a knock on effect and break other areas of my software. Even more so as the size of my application grew, and spotting new bugs became more difficult. Implementing whatever development practices I could to reduce the maintenance of my codebase would help me save time in fixing my software, and give me more time to work on pushing the quality and features of my application.

Truffle offers the ability to write smart contract tests, that have the ability to run transactions and contract calls. Transactions can be made and then variables in the contract checked to ensure the contract state changed as expected, for example:

```
it("Can make an order if enough ether is sent", async function(){
  var random = makeid(12);
  customerKey = random;
  var hash = await controllerInstance.getHash(web3.utils.fromAscii(random));
  return customerInstance.methods.makeOrder(restaurantInstance.options.address, [0,1], 2000000000000000, hash)
    .send({from: theAccounts[6], gas: 3000000, value: 2000000000000000})
    .then(function(){
      return customerInstance.methods.getTotalOrders().call({from: theAccounts[6]}).then(function(totalOrders){
        assert.equal(totalOrders, 1);
      });
    });
});
```

The above example is a test that shows the customer can make an order. First, the customer makes a transaction that creates an order via the makeOrder function, this is followed by a “.send()” to indicate that this is a transaction. Then after this transaction has been confirmed the total number of orders is read, from the customer smart contract, and checked to make sure that it is now equal to 1 as an order has now been made. The “.call()” informs the compiler that this should be a read of data and not a transaction to the blockchain.

## Application Walk-through

Here I will walk through the process of using my application, also describing any additional steps you must take to prepare your system for interfacing with the ethereum blockchain.

Firstly, you will need to download the metamask browser extension which can be found here: <https://metamask.io/>, this extension allows you to interact with a blockchain without running a full node. They achieve this by sending all your transactions via infura, a remote full node. After this is installed, click the fox icon in your extensions, create an account and ensure that privacy mode under the security settings is off. This will allow my application to detect your wallet and make requests to you for transactions.

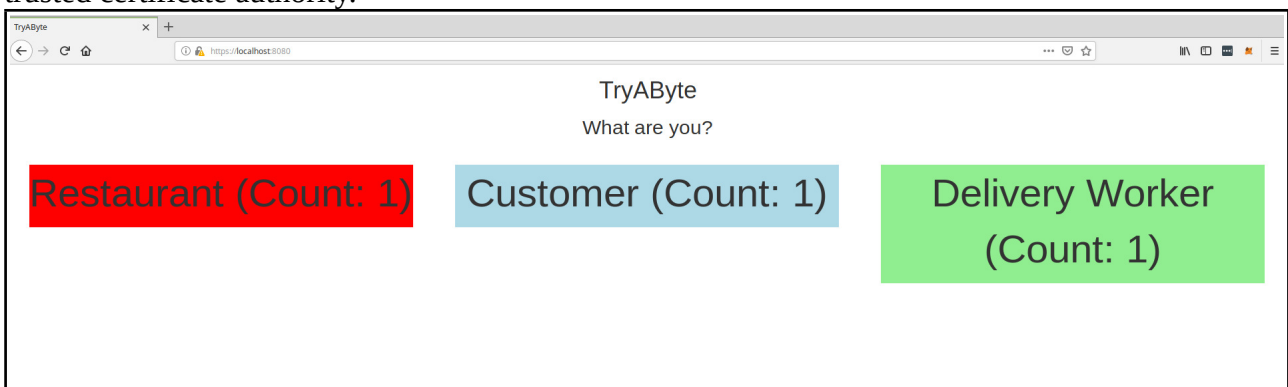
You will then need to obtain some test eth from what is known as a faucet, My decentralised application is currently running on the ropsten test network, so you will need to request eth from the ropsten faucet: <https://faucet.ropsten.be/>. after navigating to the faucet and entering the address of your wallet click “send me test eth” and you will shortly find one eth has been deposited within your ropsten account. Note by default metamask connects to the main ethereum network so you will need to change this connection to ropsten by clicking the dropdown at the top of the metamask window.

Now your browser is ready you will need to install npm, node and mogodb. Use the command npm install from within my project file to install my project dependencies, start a mongodb instance and finally have node run my App.js file in the src folder to start the server.

```
george@george-P6689-MD60969:~/Documents/TryAByte$ node src/App.js
listening on port: 8080
hosting
Current Eth Price: 134.85
Connection to database made
record count: 0
```

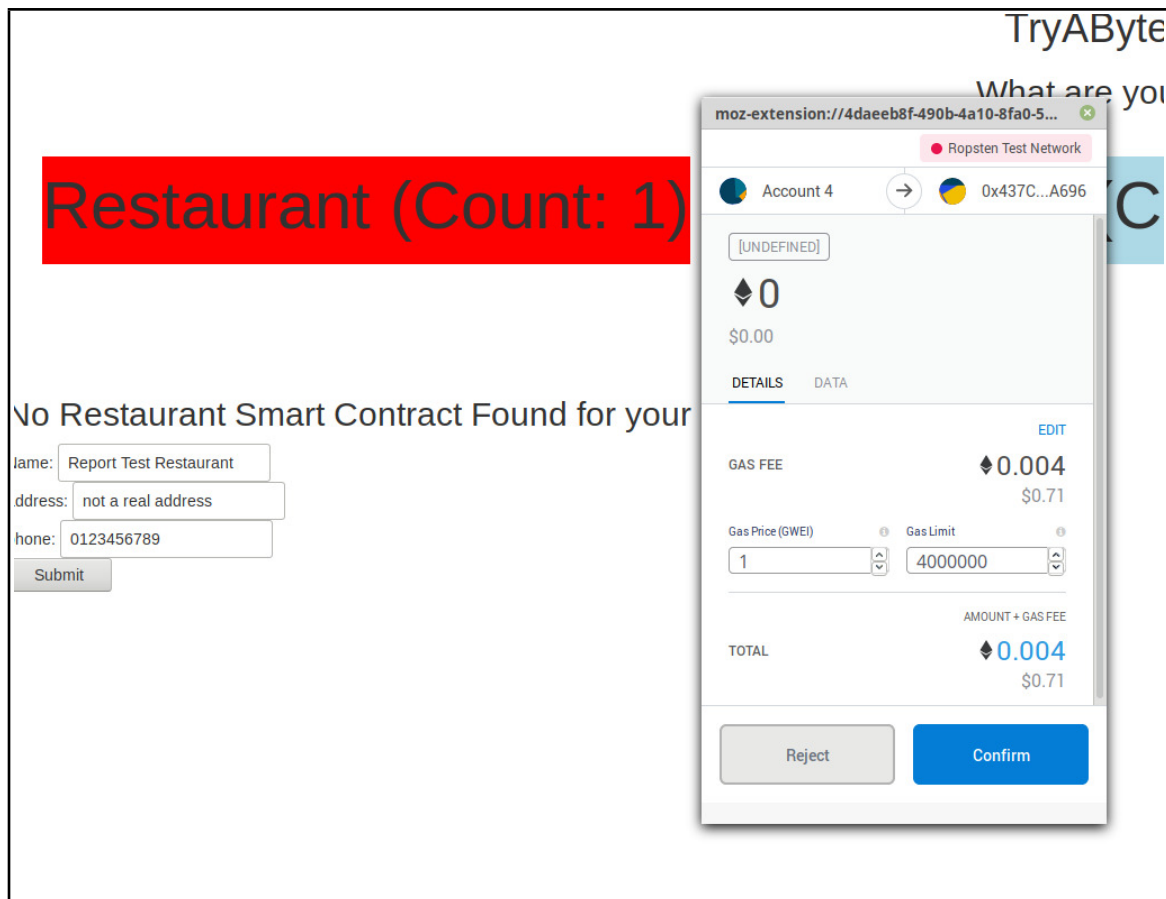
The console will output some basic information, stating the port of the server, the current price of eth, the status of the database connection and the number of records detected in the database.

Then from within your browser connect to the url “<https://localhost:8080/>”, as I’m connecting using https to ensure secure communications between the client and server a server certificate exception is required to continue. This could be resolved by having my certificate placed on a trusted certificate authority.



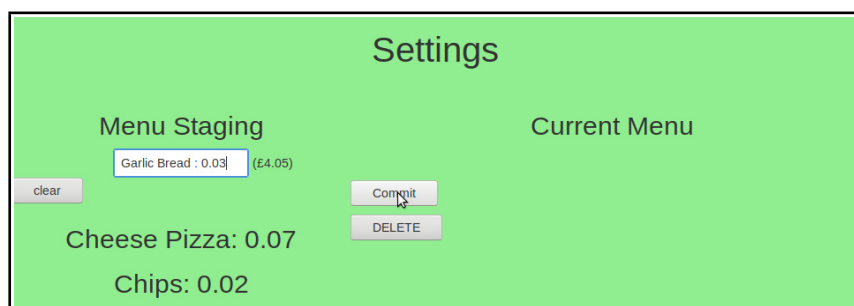
You will be navigated to the login screen, from here you can select what type of account you wish to log in as. If you already have a smart contract linked with your wallet address when clicking on an account you will be logged in straight away, otherwise, you will be prompted to make a new account. No passwords are required as authentication is done using your wallet address.

We are first going to check out the restaurant's view, for this, I will create a new restaurant contract linked with my address.



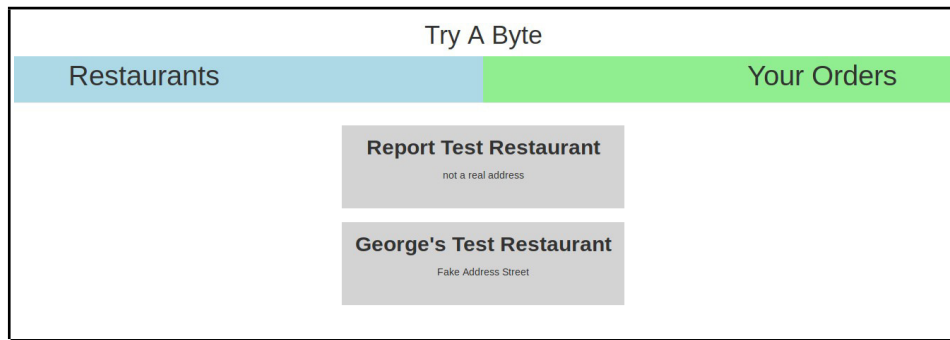
After clicking submit, metamask will open a prompt asking you to confirm the transaction that is about to take place. It gives you the gas fee you are expected to pay, and you can change this by modifying the gas price. Gas prices normally reside between 1 and 2 Gwei but can be much higher if the network is congested. Offering a gas price of 1 Gwei causes this transaction of creating a restaurant smart contract to cost about \$0.71. after clicking confirm, the UI updates to inform you the transaction is taking place. And about 20 seconds later the transaction is recorded and the user is navigated to the restaurants home page.

The restaurant home page has two tabs, a list of all orders known as the orders tab and a settings tab which allows for the editing of the restaurant's menu, the settings tab is shown below.

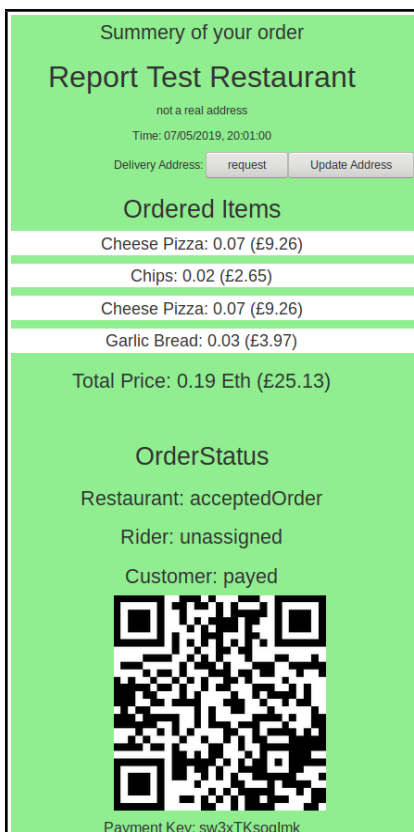
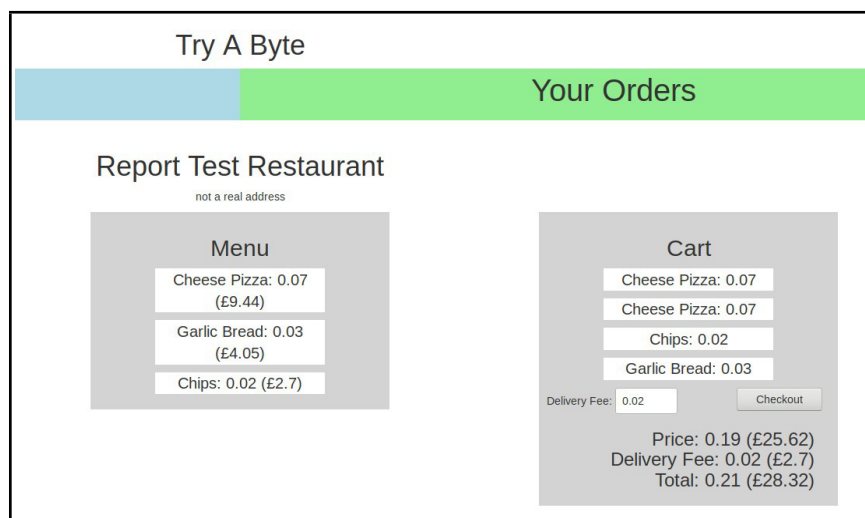


Here users can add menu items to a staging area and then commit them to their restaurant smart contract once they are done, the price for each item is specified in eth however conversion to pounds is shown to the right of the input box so a user knows how much they are pricing their items at. After confirming this transaction in metamask, the UI is updated to indicate that the transaction is being processed and after the transaction is confirmed the menu updates appropriately.

I now back out to the login menu and create a new customer smart contract. The customer page has two tabs, a tab listing all restaurants and a tab listing your orders



You can further investigate a restaurant by clicking on it, this will show you a list of items on the menu. I will now click on the restaurant we just made, "Report Test Restaurant"



In this view, you have displayed the Menu of the restaurant, with prices in both eth and pounds. A cart is also displayed, you can add and remove items from the cart by simply clicking the item name. The price is calculated at the bottom again in both eth and pound, you can also specify the delivery fee you are willing to pay however, a minimum delivery fee of 0.02 eth is enforced.

Once you are happy with your order you can click checkout and metamask will prompt you for a confirmation of the transaction. And you will also be asked to enter the delivery address.

After your order has been processed you can view it under the orders tab. Basic information about your order including the restaurant name, address, order time, ordered items, order cost, agent status' and the payment key to present to the delivery worker are displayed. The customer can also request to view their delivery address. This couldn't simply be printed instantly as the delivery address is sensitive information and therefore requires the user to prove they are allowed to view the address by sending a signature to the server.

The restaurant will now have a very similar view of this order, with most of the details remaining the same except the restaurant not having a payment key. The restaurant also has some additional buttons that display based on the restaurants current status with the order. Initially the restaurant has the ability to accept the order, doing this will update the status of the restaurant within the smart contract, giving the restaurant the ability to view the delivery address and also update their status stating that the order is ready for collection.

Creating an account and logging in as a delivery worker, we first get displayed a list of all restaurants, clicking any of these restaurants will give us a list of all orders that require delivery. The delivery worker can select any of these and offer delivery, for this they must deposit an amount equivalent to the order value. They then are given a Key to present to the restaurant which firstly proves that they are the correct delivery worker while also acting as the key that allows payment to be released to the restaurant.

Once the restaurant has notified the order ready for collection and a delivery worker has been chosen the delivery worker will go to the restaurant to collect the order. They must present their key to the restaurant who will then compare its hash to the keyhash onchain, if valid the food is handed over, the restaurant submits the key to the order contract to request payment and the delivery worker can request the delivery address by signing a message that is the contract address and sending it to the server, when the server returns the delivery address the rider can deliver the food to the customers door.

At the door of the customer, the customer presents their key to the rider to validate that they are in fact the customer who made this order, the key is validated instantly by the delivery worker by comparing its hash to the onchain hash, if valid the food is handed over and the delivery worker submits the received key to the order contract, returning their deposit and paying them payment.

## Challenges

working with solidity and ethereum, one of the more general difficulties I've had is the scarcity of documentation and code examples. Solidity, the language used to develop smart contracts, updated to v5.0 in November of 2018 changing the language syntactically, this coupled with the relatively small number of developers working with ethereum meant that by the time I had started development in early 2019 example code and online help wasn't as plentiful as for version 4.0.

solidity doesn't support dynamically sized arrays, an alternative offered is the mapping data structure with which you can store objects in a one to one map format. This is mostly fine and doesn't cause any issues, but does mean extra work is needed to iterate through a mapping or delete a mapping. This is because when deleting items from the middle of a mapping you need to shuffle the other items to ensure that the indexing is still consistent. Bellow is the function in which I do this...

```
function menuRemoveItems(uint[] calldata itemIds) external{
    require(msg.sender == owner);
    require(itemIds.length>0);

    uint totalItemsToKeep = menuLength - uint(itemIds.length);

    uint[] memory itemsToKeep = new uint[](totalItemsToKeep);
    uint counter = 0;
    for(uint i = 0; i<menuLength;i++){
        bool idUndeleted = true;
        for(uint j = 0; j < itemIds.length; j++){
            if(i == itemIds[j]){
                idUndeleted = false;
            }
        }
        if(idUndeleted){
            itemsToKeep[counter] = i;
            counter++;
        }
    }

    for(uint i = 0; i < itemsToKeep.length; i++){
        menu[i] = menu[itemsToKeep[i]];
    }
    menuLength = itemsToKeep.length;
    emit MenuUpdated();
}
```

After enforcing the ownership design pattern by checking that the caller is the owner of the contract, I then create a nested for loop. The outer loop iterating over the old menu and the inner loop iterating over the items to delete. If I find that the current item from the old menu isn't in the array of items to delete I add its index to a new array of items to keep, then using this array of indexes I reassign all my mappings and change the menu length.

One of the major challenges I faced was the storing of private data. With all the data stored on the blockchain being publicly viewable by design storing personal data within my smart contracts isn't an option. I would need to devise a solution that stored data where it couldn't be viewed by anyone, except those people with specified access rights. Additionally, some form of data validation would be useful to ensure that the personal information an end user is shown hasn't been maliciously modified.

At this stage, I removed a lot of previously unused pieces of data until I had only one data element left. This was just to simplify the development of a solution and to allow me to prove my developed solution is fit for purpose before applying it to other pieces of personal data. The data element I decided to use is the delivery address for the order. This is because I felt it was the most crucial piece of personal data to the delivery of the food.

To solve this problem, I decided that an off chain storage solution was the way to proceed. I developed my system to store personal data within a mongodb server. A customer could request to add/update a delivery address for their order by sending to the server a signed copy of the address at which the order contract is stored along with the new delivery address. On the server side, the signature is recovered. The server then queries the contract at the specified address for the customers public key. The recovered signature and the public key of the customer are then compared, if they match then we know the request was sent by the customer and we update the database storing a mapping from the contract address to the delivery address.

A similar process happens for people who wish to request the delivery address, a signed copy of the order address is sent to the server, the recovered signature is compared with the public keys for the delivery worker, restaurant and customer and if any of these comparisons return a match the server retrieves and sends the delivery address from the mongodb database to the end user.

Another of the major challenges is the high transaction times imposed by working with a public block chain. ethiums current consensus algorithm, proof of work, has a block time of about 15 seconds however this is only an average and transactions can on occasion take a lot longer. This is all necessary to ensure the consensus and security of the network, however, it can have a detrimental effect on the end users experience.

Evaluating all transactions that will take place over the typical cycle of my application, I have spotted two transactions where it is far more important to have the transaction happen quickly. These are both during the hand over of the food, more specifically, just before the restaurant hands the order over to the delivery worker, the delivery worker needs to specify that they have revived the order, and just before the delivery worker hands the order over to the customer, the customer has to specify that the food has been delivered. If either the restaurant or delivery worker gave the other party the food before getting confirmation from the network, there is a risk that the other party never informed the network they had received the food, meaning the goal state of the order contract will never be reached and the funds won't be released to the appropriate parties.

I made use of the commit and reveal design pattern to resolve this issue. More specifically, in the case of the transaction prior to the delivery worker handing the food to the customer, the high transaction time issue is solved like so. As the customer requests a new order they are required to also generate a random string as a key, and pass the hash of this key with the request to make a new order. The Key is hashed using the keccak256 hashing algorithm, which is a solidity implementation of sha3. This hashed key is then stored in the order contract. The order contract contains a method that results in the payment being released to the delivery worker, this method takes one parameter which is a key, this parameter is hashed and then compared to the hash of the key submitted by the customer. If the two hashes match, the payment is released. Due to the first pre-image resistance property of hash functions we can assume that the delivery worker won't be able to obtain the original key from the hash. Therefore, the only way they can receive the correct key is directly from the customer. Also, as you can read from the blockchain instantly, the delivery worker can validate the given key instantly by simply reading the hash from the order contract, hashing the key they have and comparing the two. To the customer and delivery worker, this transaction would play out like so. The delivery worker would arrive at the customer's house and



request from the customer the key is presented. The customer would then present the key, and the delivery worker would enter this into their client application. The client would quickly retrieve the hash stored on chain and compare it with the hash of the key they were just given. The delivery worker would be notified that the key is the correct one, they can hand over the food and leave knowing that they have the ability to release the payments at any time, this is done straight away anyway for convenience.

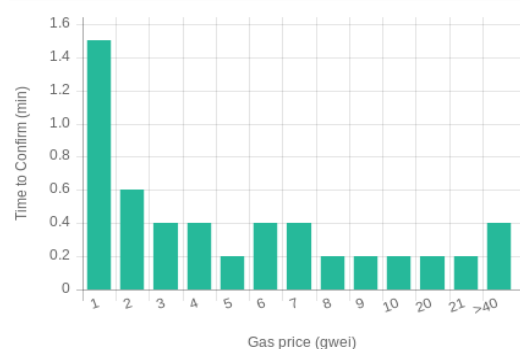
## Evaluation

In this section I will evaluate my final product, comparing the solution with a more traditional solution built around a centralised server. After this, I will go on to describe further improvements that I wish I could have implemented or explored but haven't been able to due to time restrictions. Finally, I will include an additional section on the planned upgrades to the ethereum network and how these upgrades could potentially change, or improve my end product.

### Transaction fees

The Primary benefit that I proposed for a blockchain based solution compared to systems such as just eat is the fee difference. Before I go into the fees used by my system I wish to explain a few things about ethereum fees. As mentioned before, ethereum fees are calculated based on the gas usage and the gas price. The gas usage will mostly stay consistent for any operation performed, only sometimes changing with updates to the network. The gas price, however, fluctuates based on the congestion of the network, as the network becomes more congested the gas price increases which incentivise more miners to join the network, reducing congestion and lowering the gas price. According to the ethereum Gas station [4] the current safe low gas price is 1 Gwei, this will give you a confirmation time of about 90 seconds. The graph to the right was taken from the ethereum gas station [4] on the 8th of May. It shows the average confirmation time by gas price.

Confirmation Time by Gas Price



Firstly, a comparison of account creation costs. Just eat and deliveroo don't require any payment for a customer or a delivery worker to sign up to their service, my blockchain solution does require payment from a delivery worker or customer to sign up, this is shown below.

Action	Gas Used	Cost in eth at a gas price of:			Cost in pounds at a gas price of:		
		(1 Gwei)	(3 Gwei)	(6 Gwei)	(1 Gwei)	(3 Gwei)	(6 Gwei)
Customer Make Account	611958	0.00061	0.00184	0.00367	0.08	0.24	0.47
Rider Make Account	434446	0.00043	0.00130	0.00261	0.06	0.17	0.34

Depending on how quickly you would like the transaction to process, the cost to create an account varies. Paying 3 Gwei would mean account creation would take about 24 seconds depending on network usage. Having to charge customers and workers is a deterrent to new users and would stunt the growth of this application. Therefore it is important some method of offsetting this initial cost. Such as having the customer pay a small fee for each order until the cost has been paid.

Just eat charges a joining fee to restaurants of £699 [1], deliveroo doesn't require any initial sign up fee. The restaurant sign up fee for my system, and the cost to change the menu is as follows...

Action	Gas Used	Cost in eth at a gas price of:			Cost in pounds at a gas price of:		
		(1 Gwei)	(3 Gwei)	(6 Gwei)	(1 Gwei)	(3 Gwei)	(6 Gwei)
Restaurant Creation	2049648	0.00205	0.00615	0.01230	0.27	0.80	1.59
Adding 20 Items to the menu	391949	0.00039	0.00118	0.00235	0.05	0.15	0.30
add a further 3 items to the menu	77494	0.00008	0.00023	0.00046	0.01	0.03	0.06
remove random 3 items from the menu	260120	0.00026	0.00078	0.00156	0.03	0.10	0.20
remove remaining 20 items from the menu	70507	0.00007	0.00021	0.00042	0.01	0.03	0.05
Total	2849718	0.00285	0.00855	0.01710	0.37	1.11	2.21

these fees will vary based on the exact parameters you are sending in the function calls, having a restaurant with a larger menu would also increase the sign-up cost for your restaurant. An interesting point to note here, however, removing only some items from the menu commands a higher gas usage than simply removing the whole menu, this is because all other menu items need to be shifted to ensure that the indexing is correct. Perhaps it would be worth updating this function so to delete a few items the code would clear the whole menu and then add the menu back in without the deleted items.

Now we consider the fees for the actual orders being placed, this fee is the most crucial as it will be charged most often. According to the just eat website they charge 14% commission to the restaurant for every order and a 50p service charge to the customer. Deliveroo doesn't openly disclose their commission rates, each restaurant negotiates its own deal, but several sources have stated that deliveroo charges between 20% and 30% commission on every order.

The bellow table documents the gas usage, and cost of all transactions that would be made in the process of ordering using my system.

Action	Gas Used	Cost in eth at a gas price of:			Cost in pounds at a gas price of:		
		(1 Gwei)	(3 Gwei)	(6 Gwei)	(1 Gwei)	(3 Gwei)	(6 Gwei)
Make order containing 10 Items	1551927	0.00155	0.00466	0.00931	0.20	0.60	1.20
Restaurant Accepts Order	47344	0.00005	0.00014	0.00028	0.01	0.02	0.04
Deliver Worker Offer Delivery	165722	0.00017	0.00050	0.00099	0.02	0.06	0.13
Restaurant Notify ready for collection	32344	0.00003	0.00010	0.00019	0.00	0.01	0.03
Restaurant Submits Delivery Worker's Key	82295	0.00008	0.00025	0.00049	0.01	0.03	0.06
Delivery Worker Submits customer's Key	82630	0.00008	0.00025	0.00050	0.01	0.03	0.06
Total	1962262	0.00196	0.00589	0.01177	0.25380	0.76140	1.52279

Assuming all agents pay the fees at a gas price of 3 Gwei, and 10 items are ordered, an order is cheaper to make using my system than from just eat if the order costs more than £1.85. and my system is cheaper than deliveroo when the order costs more than £2.53. it should be said however that my system is still very bare bones, a lot more would need to be added to produce a system that is ready for real-world use, these additional features would push up the price to make an order. Still, my proof of concept indicates that purely based on fee cost, my system could viable.

## Transaction Times

the biggest worry I had for my system was the high transaction times. Compared to just eat and deliveroo where changing the state of the central server appears to happen seamlessly, the high transaction times of the ethereum blockchain make a seamless experience much harder if not currently impossible to produce. With that being the case I still feel my end solution employed some intelligent design to work around this.

My best workaround for the high transaction times was the use of the commit and reveal design pattern to implement keys that allowed the release of payments. The two transactions where I implemented this require immediate feedback to avoid the delivery worker or restaurant having to wait around. Other transactions don't require such an immediate response, and instead could be made and checked up on later. I still felt it important to develop a UI that compensated for any asynchronous calls by displaying messages indicating that transactions are being processed.

One transaction that I suspect may cause issues is when two or more delivery workers offer to deliver an order at the same time. I imposed a check to evaluate if a delivery worker had been assigned as early as possible.

```
function riderOfferDelivery(bytes32 keyHash) public payable{
    require(riderStatus == uint(riderState.unassigned),"this order already has delivery organised");
    require(RiderFactory(Controller(controller).riderFactoryAddress()).riderExists(msg.sender), "must
```

This would minimise the gas cost of a delivery request to an order that already has a delivery worker assigned, however, delivery workers would have to be willing to pay out large deposits only to have them returned without being added to the contract as a delivery worker.

## Future Work

### Improvements to my system

If time wasn't a constraint there's many upgrades and additional features I would have liked to add to my system, I will now go over some of them, what impact they would have on my system and how I would go about implementing them.

Firstly, I would like to have investigated blockchain based indexing methods. Currently, my system simply returns a list of all restaurants and a list of all orders, this works fine for my proof of concept but if many restaurants signed up it would be important that the user only received the ones relevant to them. This would most probably be some form of spatial indexing, only showing a user the restaurants within a specified radius of their location.

Another upgrade that I feel would be needed is the storage of user data. Currently, only the delivery address is stored off chain and a hash is stored on chain to verify the address is valid. This model should be extended to all personal user details, including names, phone numbers and emails. Access to this information would be restricted to only those who have access rights according to the order contract on the blockchain.

Currently, the value of eth is very volatile, this would cause issues with the pricing of items as most restaurants and customers would want a relatively stable price in fiat currency. It is possible to get the price of ethereum using an existing smart contract on the network, known as the fiat contract. You can simply call this contract and have it return the price of ethereum. This contract is updated by the owner though, and there is no system enforcing them to specify the correct price. Another solution could be to retrieve the data myself using oraclize, an oracle service for blockchain. I could make a request and receive several prices, only trusting the received price if it accumulates enough votes. Over time, as ethereum approaches mass adoption, the price should stabilise reducing the impact of this issue.

Looking into different wallet providers instead of metamask may improve user experience, currently, metamask asks for confirmation on every transaction or signature you place. This does improve security and assures that users aren't sending transactions they don't wish, however, I would like to look into storing user private keys myself, for the purpose of removing many dialogue boxes that slow down the flow of my system. Eth Light wallet is a wallet that lets you do this, I actually used it for handling the deployment of my contracts to the test network, though due to time restrictions felt it best I prioritise other project aims before attempting to develop a private key storage solution.

Customer and delivery worker fees will certainly motivate users against my platform, therefore, I would like to develop some way to offset these initial fees for the user. However, I would have to be very careful when developing this as a solution. The ability for users to create an account at no charge might lead to a malicious user spamming the creation of user account. If the gas fee for this came from another location that location could quickly be drained of all funds.

I hoped I would have also had time to implement the mortal design pattern, this is basically where the owner of a contract can self destruct this contract once it is no longer required. This could be done by simply restricting access to the functions if a variable was set indicating the contract had

been destroyed. Currently, I have no way to remove unused restaurants so the risk of inactive restaurants cluttering my system is present.

A worry of mine with this system is also how to deal with non-conforming agents, if an agent defects from the typical process of an order, all three parties have the potential to lose out, as funds will simply remain within the contract. Adding time triggered events within the contract to return the funding to the correct agents based on the status of the order when the timed event fires could solve most of these issues. For example, if the delivery worker decided after picking up the order from the restaurant they would eat the food instead of delivering it. After enough time had passed the timed order event would fire, the order contract would see that the last recorded status of the order was that the delivery worker had the food. It would then refund the customer and pay the delivery drivers deposit to the restaurant.

## Improvements to the ethereum network

the ethereum blockchain is an emerging technology and as such has a lot of upgrades that are currently being developed by some major teams with a lot of funding. In this section, I will discuss what these upgrades are and what they could mean for my system.

Proof of stake (POS) is an alternative consensus algorithm to proof of work that is currently being developed for ethereum. POS offers vastly reduced network energy consumption. It does this by instead of selecting miners based on their computational power, it selects them based on their stake of eth. This upgrade wouldn't directly impact my application however it is an important point to mention due to the current high energy consumption of the network. As of the 7th of May, the first proof of stake test network for ethereum has been started. Known as the sapphire testnet [16] currently it is very basic, not offering smart contract compatibility or the ability to use any other client but their own, I wouldn't currently be able to use this as a development platform. However, this is a good milestone for the development of proof of stake.

Currently, any ethereum node has to store the entire state of the blockchain and sync all data. This limits the transactions a blockchain can process to no more than the transactions a single node can process. Sharding is a proposition to basically split the network into parts where any transaction wouldn't have to be verified by all nodes in the network but instead only a subset of nodes. [17] This would greatly improve the transactions per second of the network and also largely reduce the fees.

One of the layer two solutions that has been proposed is state channels. This involves two or more users performing secure transactions off chain, and can reduce the number of transactions made to the blockchain down to just two, The opening of a state channel and the closing of a state channel. How this would apply to my system is the restaurant, customer and delivery worker would all pay a deposit into a state channel smart contract on the blockchain, opening the state channel. Transactions would then be made, signed and sent to one another but not published on the block chain. Once all transactions have been made the state channel can be closed, at this time the state channel looks at all signed transactions that have been made between the parties off chain and releases the funds to the agents based on the transactions that have been made off chain. As the off chain transactions are basically instant and the number of on chain transactions is reduced down to two, the total time waiting for blockchain consensus is reduced and the overall gas fee is reduced as computation is taken off chain.

## Conclusion

Blockchain technology is still an emerging technology, ethereum still has a lot of catching up to do if it ever wants to compete with more centralised systems. I feel this my project has proven that given enough time a solution could be implemented, however, I don't believe currently that the solution would scale up. Using state channels and sharding could push the ethereum network to be able to process enough transactions to cope with the activity that platforms like just eat and deliveroo see. But as it currently stands, the ethereum network would likely become seriously contested if a similar level of activity was seen, this would push gas prices up causing the fees to be

too excessive to provide an affordable service. Even if ethereum reached a state where it could handle the required transaction throughput, It should be noted that some centralisation would still be needed, for storing personal user information.

## Reflection

I have been interested in blockchain technology for the last two years, I'm fascinated by its potential to disrupt a lot of the industries as we know them. I chose to do this project with the aim of gaining more insight into the technical details behind blockchains and to gain a basic understanding of how smart contract development works.

Firstly it should be said that during this experience my knowledge regarding blockchain has improved dramatically, other than gaining a much deeper understanding for the ethereum protocol, I have also developed a good understanding of the solidity programming language for smart contract development. I have also improved my understanding of asynchronous programming, having never explored the topic too deeply I have had to adapt and improve my knowledge to allow for the front end development of my application.

My ability to plan has been tested like never before, this is the largest project I have ever done and therefore required the most planning and background research. I feel that my ability to conceptualise and outline a system has improved, I was effectively able to detect and investigate potential design flaws prior to my design so that I had a plan for ways to solve these challenges when I encountered them in development.

## References

- [1] : Just Eat 2019, Restaurant sign up page, viewed 28 April 2019, <<https://restaurants.just-eat.co.uk/>>.
- [2] : S. Haber and W. Scott Stornetta, How to time-stamp a digital document, 1991: <<https://link-springer-com.abc.cardiff.ac.uk/article/10.1007/BF00196791>>.
- [3] : Dr G. Wood, ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER, 2019, Viewed 28 April 2019, <<https://ethereum.github.io/yellowpaper/paper.pdf>>.
- [4] : Eth Gas Station, n.d, viewed 8 May 2019, <<https://ethgasstation.info/>>.
- [5] : Dapp Rankings, state of the dapps, 2019, viewed 28 April 2019, <<https://www.stateofthedapps.com/rankings>>.
- [6] : Eva, 2019, viewed 28 April 2019, <[https://eva.coop/index.php#about\\_us](https://eva.coop/index.php#about_us)>.
- [7] : Eva, 2019, Google Play, viewed 28<sup>th</sup> April 2019, <<https://play.google.com/store/apps/details?id=com.evaclient>>.
- [8] : swarm city, n.d, viewed 28 April 2019, <<https://thisis.swarm.city/>>.
- [9] : M.Wöhrer and U.Zdun, Design Patterns for Smart Contracts in the Ethereum Ecosystem, n.d, viewed 29 April 2019, <[https://eprints.cs.univie.ac.at/5665/1/bare\\_conf.pdf](https://eprints.cs.univie.ac.at/5665/1/bare_conf.pdf)>.
- [10] : Y. Liu, Q. Lu, X. Xu, L. Zhu and H.Yao, Applying Design Patterns in Smart Contracts, n.d, <[https://www.researchgate.net/profile/Qinghua\\_Lu5/publication/325900304\\_Applying\\_Design\\_Patterns\\_in\\_Smart\\_Contracts/links/5c2ecd8ea6fdccd6b58fa0c4/Applying-Design-Patterns-in-Smart-Contracts.pdf](https://www.researchgate.net/profile/Qinghua_Lu5/publication/325900304_Applying_Design_Patterns_in_Smart_Contracts/links/5c2ecd8ea6fdccd6b58fa0c4/Applying-Design-Patterns-in-Smart-Contracts.pdf)>.
- [11] : Dapp statistics, State Of Daps, 2019, viewed 1 May 2019, <<https://www.stateofthedapps.com/stats>>
- [12] : EOS resource planner, n.d, viewed 1 May 2019, <<https://www.eosrp.io/#calc>>
- [13] : Blockchain activity matrix, Block'tivity, n.d, viewed 1 May 2019, <<https://www.blocktivity.info/>>
- [14] Remix Online Ide, <<https://remix.ethereum.org>>.
- [15] : Ethereum Blocktime History, Etherscan, 2019, viewed 3 May 2019, <<https://ethscan.io/chart/blocktime>>
- [16] : The Sapphire Testnet, Prysm, 2019, viewed 8 May, <<https://alpha.prylabs.net/>>
- [17] : Sharding FAQ, github, 2019, viewed 8 May, <<https://github.com/ethereum/wiki/wiki/Sharding-FAQ#what-is-the-basic-idea-behind-sharding>>.
- [18] : About Infura, Infura, 2019, viewed 8 May, <<https://infura.io/about>>.