University College London

Faculty of Engineering Sciences

COMPGC99 - Individual Project

MSc Computer Science

A Peer-to-Peer Option Trading Platform

Author:
Filippos A. Zofakis

Author's Email:
filippos.zofakis.17@ucl.ac.uk

5th September 2018

This dissertation is submitted in partial fulfillment of the requirements for the MSc Computer Science degree at UCL.

The work presented is my own. Where information has been derived from other sources, I confirm that this has been indicated.

Abstract

The purchase, execution and settlement of financial options is still to a large extent a manual and inefficient process [1], [2]. Financial institutions are already undertaking efforts to try and automate at least some parts of it using distributed ledger technology. This project aims to investigate to what extent such a system can be built on a fully decentralised public peer-to-peer blockchain network, with open access for everyone. The project report describes a prototype option trading platform, which is built as a decentralised application (DApp) consisting of a website and smart contracts written and deployed on the public Ethereum blockchain network [3]. This proposed configuration investigates the viability of a public distributed ledger as a means of data storage and automated execution, contrasted with more traditional setups, such as a central server and database.

The application created allows option writers to sell option contracts online by creating a customised option factory. The buyers can view spot prices for the underlying assets, which are retrieved securely to the blockchain from external market data feeds via oracles. They can input the desired characteristics of the financial option they wish to purchase and receive offers, with premiums calculated on-chain. Once they decide to make a purchase, the system selects the best offer and stores the relevant information in the blockchain's state. The buyers can also view their balance and option portfolio, as well as exercise options past their expiration date, with balances adjusted automatically. To coordinate all this, a master registry smart contract was created that acts as an agent keeping track of the sellers and their respective factories. The registry also communicates with the committee of deployed oracles and calculates an average price for assets through a mock weighted voting scheme.

Owing to the consensus protocols (algorithms) currently employed by most public blockchain networks such as Bitcoin and Ethereum, where every full node has to verify all transactions for the integrity of the network, transaction processing times are very long [4]–[6]. The system created did work in the experiments that used the permission-less (open) Ethereum network, but it was quite slow and even occasionally unresponsive. The takeaway from this is that achieving full decentralisation and financial disintermediation is still very challenging. For the purposes of comparison, a second set of experiments were carried out, this time with the platform reconfigured to use a private blockchain network simulated on a local machine, rather than distributed around the world. The private blockchain enabled instant transaction processing and illustrates how the technology would work when utilised in a controlled setting, as would be the case with enterprise chains, which are purpose-built institutional platforms.

Supervisor:
Dr Christopher D. Clack

Supervisor's Email:
c.clack@cs.ucl.ac.uk

Contents

List of Tables

List of Figures

1 Introduction

1.1 Motivation

The option trading and settlement process is an inefficient endeavor, in terms of processes, maintenance and communication [1], [2]. It needs to be brought to the modern age and opened up to a wider market by capitalising on recent technological advances. The implementation and use of distributed ledgers holds potential for reducing complexity and increasing efficiency because they possess desirable properties, such as disintermediation and automatic reconciliation between accounts [7]–[9]. Blockchain-based implementations, in particular, are also characterised by an immutable history of states, meaning that there is a single source of truth about transactions that have happened in the past and that history can be independently verified by everyone participating in the network [10]–[12]. This feature can aid in dispute resolution, which is another area where blockchains show promise [13]–[16].

A traditional system makes use of the three-tier web architecture, consisting of a client layer that runs the front-end for users to interact with the system, a middle-ware application server handling the business logic and a data layer/source [17]. Such a web-based application relies on centralised servers to run the code handling the business logic. A decentralised application (DApp), in contrast, runs (at least partly) on a decentralised peer-to-peer network of nodes [18], thus removing the need for a central party. This is accomplished via the use of smart contracts, which offer automatic enforcement of agreed upon transactions. They typically run on a public network that is distributed in nature and, to a certain extent, decentralised (depending on the distribution of full nodes that mine/process transactions versus those who do not). As far as the front-end client for users to access the application is concerned, any programming language or framework that would normally be used can be also be utilised, as long as functionality exists to make calls to a node running an Ethereum client. A DApp thus consists primarily of the front-end layer acting as a user interface (the website) and the smart contract layer containing both the business logic and the data.

1.2 Project Objectives

The aim of this project was to investigate possible mechanisms to enable the secure online trading of financial option contracts on a public platform, while at the same time automating as much of the process as possible. To achieve those goals, an online system was created that enables a user to interact with code running independently and verifiably on a blockchain network. Building the system was accomplished by creating a web application, writing smart contracts in the Solidity programming language [19] and deploying them on the public Ethereum blockchain network [3], as well as implementing the connection between the two parts.

The final outcome is a hybrid application that has some centralised features, characteristic of traditional Web 2.0, but combines them with decentralised ones, taking advantage of the advances in blockchain technology. At its core, the system presented in this technical report features an interplay between a web-facing client and smart contracts deployed and running on the Rinkeby version of the Ethereum test networks [20], which are intended for development and experimentation. The application logic frequently incorporates this exchange between the two parts of the system, starting with the end user performing some action on the front-end client, followed by either a call of a function residing in a smart contract or a centralised server response, depending on the use case. Use of a server, together with a central database, is necessary for the storage and retrieval of large pieces of data, where it is not economical to put

them on the blockchain. An example of this are the files making up the front-end application. Expensive computation could also be delegated to the server (such as the calculation of option premiums), as well as triggering routine tasks that cannot yet be scheduled on-chain, such as periodic price updates based on incoming live market data or automatic option exercise at expiration.

The project's objectives can thus be defined as follows:

1. Create an interactive web application that retrieves and displays market data from online data feeds, via use of their Application Programming Interfaces (APIs). APIs are tools that allow the application to communicate with remote servers in a standardised way and retrieve data from them [21]. The user requesting the information might act as a writer or a buyer of a financial option contract. The application should also connect to a central server.
2. Create smart contracts modeling financial options, more specifically call options, and deploy them to one of the public Ethereum peer-to-peer blockchain networks (main or test).
3. Implement the components enabling the interaction between the two parts of the system, thus creating a minimum viable product.

This report describes the process of creating the decentralised platform and the outcomes with regards to achieving these objectives. It starts with a background discussion to provide the reader with the necessary context and continues with an analysis of the project requirements and the use cases envisioned. The report then proceeds to the design part, where the architecture of the system is illustrated, the design challenges encountered are examined and the approach taken is elaborated on. The implementation section comes next, where a thorough analysis of the code written for the project is presented, as well as a comparison with a private version of the same system. A section on testing, validation and a security audit that was conducted follows. Finally, the conclusion section draws a comparison of this emerging technology and its feasibility versus traditional centralised systems. Avenues for future work are also discussed. The appendices and references have been placed at the end of the document.

2 Background

2.1 Ethereum Blockchain Application Platform

To begin with, a brief introduction to the concept of a blockchain will be presented. As the name suggests, a blockchain is essentially a chain of blocks that keeps on growing over time. Each block contains a set of transactions from a peer-to-peer network that have been pooled together, along with a time-stamp (in Unix time) [22]. These transactions are passed through a cryptographic hash function to form a record that cannot be altered or reversed, which is then publicised to the network, thereby proving that they have indeed taken place at that specific time [23]. A reference to the previous block is included in the next one created (mined), thus resulting in the aforementioned chain of blocks and ensuring that they are irreversible [24]. A blockchain's shared memory of transactions across all network participants is intended to be used as a single source of truth, as an adjudicator of disputes [25] over who owns what (for example, how much of a unit of value such as Bitcoin), which of those transactions are legitimate and have actually occurred and in what specific order (to prevent being able to spend the same money twice, the famous double-spending problem [23], [26], [27]). Reaching agreement on those things among a public group of participants who do not know and do not trust each other is a very difficult task, which is the reason regulated financial intermediaries exist in our socioeconomic systems. The important innovation that Bitcoin brought to the world and was presented in the original whitepaper [23] was a new algorithm to reach agreement on which of those transactions are actually valid and should be accepted, thus enabling peers in a public network to enter into transactions with each other directly and securely. These types of algorithms enabling agreement and coordination among a distributed group of autonomous agents are commonly known as consensus mechanisms or protocols in distributed computing (for example, see [28], [29]).

Blockchain design features at its core an interplay between scalability (measured, for example, by transaction throughput) and public auditability/verifiability of the outputs of each transaction. Networks running protocols like Bitcoin and Ethereum are secure from attacks because of the network effect, meaning that a large number of anonymous participants verify all transactions independently, but that makes them very slow to reach consensus [30]. This is especially true when the networks are public and open for anyone around the world to join, in a peer-to-peer fashion (similar to the BitTorrent protocol used for file sharing [31]). When some code (for example, a function) is executed, the output has to be verifiable and auditable by all the nodes comprising the network. Full nodes (or miners or validators) are the ones who actually process transactions and run the protocol that the network uses, rather than just reading the current state. Every single one of them has to run the code of the smart contract to verify its output, when a transaction calling one of the contract's functions is executed [32]. This makes the system decentralised and the data immutable, providing cryptographic guarantees that the users have the value they are transferring or that the smart contract runs the way it is supposed to, however it comes at the cost of speed/efficiency. There are also inherent restrictions on the amount of data that can be stored in a public blockchain. While theoretically infinite, in reality data storage carries prohibitively high processing fees and storage costs [33]. The reason is that other participants in such a network have to be incentivised and rewarded for storing data and processing transactions. These constraints feed into the design process and the decisions that have to be made, when implementing a distributed system utilising an open-access model.

Ethereum is a successor to Bitcoin that modified the consensus protocol and enabled participants to per-

form computation on a distributed network using a shared blockchain [34], instead of just sending transactions. Its defining characteristic is the Ethereum Virtual Machine, which is a mechanism designed to run the same coded instructions with exactly the same outputs on every single node in the network. This enables the creation of decentralised applications that can do more than simply record balances securely on a distributed ledger. These decentralised applications are called smart contracts and are explained in the following subsection.

2.2 Smart Contracts

The term 'smart contract' dates as far back as 1994, when it was coined by Nick Szabo [35]. Szabo has also been argued by many to be the actual creator of Bitcoin, hiding behind the moniker 'Satoshi Nakamoto' [36]. He described a smart contract as a computer-based transaction protocol executing the terms of a legal contract. The motivation behind smart contracts is to satisfy common contractual terms, while at the same time minimising exceptions and dispensing with the need for intermediaries. He also envisioned additional potential benefits, such as a reduction in enforcement and transaction costs. All this is possible because the contract as a program is able to exert direct control over digital assets, meaning it can transfer funds, tokens, or other assets (such as digitised financial options) it has access to by itself [37].

A framework that paved the way conceptually for the digitalisation of financial instrument issuance is the Ricardian Contract [38]. It describes financial instruments as contracts between their issuers and the buyers, meaning that the issue of an instrument is the same thing as the contract itself. This enables the issuance of value to be digitised and integrated with an online payment system, as part of a smart contract's encoded functionality. The concept does not have to be confined to finance, however; it can also be extended to cover applications in various other domains.

The economist Williamson describes how human relations of a calculative nature are governed by institutions facilitating trust, both socio-political and economic ones [39]. The example most pertinent to this project is that of financial institutions. Smart contracts aim to facilitate trust by being, at least in theory, guaranteed to execute precisely as defined. Besides safety failures that can take place, though, one can also not be certain that the definition is as intended, to begin with. This reinforces the need for verification and validation of the software system and its code, particularly when it comes to critical applications in heavily regulated areas like financial services. An example of a typical use case for smart contracts can be found in one of Alibaba's latest patent filings for a cross-border payments system [40]. As described in the regulatory filing, the process commences with a transaction request. The next step consists of checking whether the user has enough of a balance in their current account with the contract, as well as taking into consideration all relevant legal/compliance matters. Finally, the transfer is made automatically by the contract. This process has to be codified as part of a smart contract that is then deployed on the blockchain network. Users can subsequently trigger that contract's functionality, whenever they wish to make a payment.

2.3 Privacy

A blockchain network can be configured to allow a varying degree of access, based on the requirements for its use. There are two main access paradigms, open versus closed. In an open access system, a so-called public or permission-less one, everyone is free to read the chain or create transactions, as long as they follow a predefined set of rules (the consensus protocol) [41]. The most prominent example of such a network based on the blockchain data structure is Bitcoin [23]. Bitcoin is decentralised by design, which

enhances its resistance to censorship (even though there has been an inadvertent centralisation of power [42], [43]). This public nature of networks like Bitcoin and Ethereum means they suffer from privacy issues, though. Anyone can join the network and receive transactions that are broadcast by others and there are no 'holy grail' solutions to fix that [44].

A private or permissioned chain, on the contrary, is a closed system, where you have to be authorised to run a node and participate in the network. This is handled by a layer controlling network access [45]. While on public blockchain networks anyone could become a miner or validator processing transactions anonymously, on private ones the validators have to be approved by the network's owner. A miner is a node that receives, processes and validates transactions, using a specific consensus protocol [46]. The consensus protocol is the mechanism which ensures that all nodes in the network agree on the current state, meaning they agree on the transactions that have taken place and the account balances of each participant (in the case of an electronic cash implementation). Examples of permissioned blockchains are Hyperledger [47] and R3 Corda [48], which are tailored towards enterprise consortiums. There can also be combinations of public and private networks, which are the so-called hybrid chains.

It is worth mentioning that even in a private network, the participants that have been authorised to join will still be able to read all transactions. Solutions that have been suggested are centered around encryption and aim to take advantage of cryptographic techniques like code obfuscation and zero-knowledge proofs, by combining them with the blockchain and only using the latter to prove the authenticity of the encrypted transactions [44]. Ethereum, in particular, has been described as having an ingrained transparency, that is aimed at ensuring all transactions are legitimate; as a result, attempts to increase privacy have met serious challenges [49]. There are other well-known public platforms such as Monero, however, that claim to offer untraceability [50]. When it comes to enterprise blockchains, J.P. Morgan is an example of a financial institution working on such an experimental project: their implementation is called Quorum [51], is a version of Ethereum that requires permission (so it is not public) and it aims to support higher transaction throughput and different levels of privacy.

2.4 Governance

There are different views and approaches towards governance of a public blockchain. One view is to simply attempt to reproduce existing legal structures, while another is to strive to regulate every aspect of behaviour [52]. Code may have the capability to enforce rules in a more efficient manner than legal documents, however it is usually not straightforward to convert complex agreements and organisational structures into a format fit for machine execution [53], [54]. Smart contract code does hold potential for regulating people's interactions, though, especially as we move from "code is law" to "law is code", meaning that law will increasingly be defined as code [55].

Governance of a network depends heavily on its construction. Permission-less networks are censorship resistant by nature and that has been touted as one of their great advantages, even part of their reason to exist. However, they also have disadvantages, such as the relative loss of privacy, given that anyone can join the network, receive transactions and read the accompanying data in their payload. Another negative byproduct of their structural design is that it is more difficult to take action, when necessary. To illustrate, a bad actor could set up and run a malicious node and it would be hard to identify them as such, so it might not be possible to block their transactions. It is thus difficult to stop attackers from infiltrating the network. A permissioned network, on the other hand, can have procedures in place to remove offenders.

For example, a denial of service attack can be dealt with much more easily in a permissioned network [56].

In the field of corporate finance, plenty has been written about voting rights and rules. The one share-one vote rule has been claimed to be socially optimal [57]. In the world of blockchains, though, shares have been substituted by tokens, which may or may not grant the their holders various rights. This has given rise to the field of *cryptoeconomics*, which studies the economic incentives embedded in systems built using cryptographic primitives [58].

2.5 Oracles

The concept of an oracle is an important one, both in a general complexity and computability sense, as well as with regards to applications in cryptography and blockchains that are more relevant to this project. An oracle machine is a standard Turing machine, but with an added connection to an oracle that is able to solve certain decision or function problems [59]. The machine queries the oracle to get a solution to a computational problem. In a blockchain context, the oracle serves as an agent that can retrieve information existing outside the realm of the blockchain/smart contracts and then submit that information in a provably secure manner to the caller [60]. These properties enable the use of oracles as intermediaries between the smart contracts running on the blockchain and the outside world, thus facilitating the exchange of information with public APIs, for instance, and ensuring that the results computed by the contracts are identical. Another typical illustration of the need for oracles is the secure generation of random numbers, for instance when designing hash function outputs [61].

The use of oracles in a blockchain system, however, does not alleviate certain security challenges. Even when using TLS-based 'proofs of honesty' to ensure that the data were not tampered with [62], [63], the source of the data still needs to be a trusted authority/third-party. To mitigate that risk to a certain extent, this project implemented a voting scheme, whereby a committee of oracles each submit their view on the current price in a secure manner and the system then aggregates the information. In this case, a weighted average of the different prices is calculated, similar to a weighted-voting scheme [64]. The weights are meant to represent the degree of confidence in the integrity of the data source of each oracle. For the purposes of this project, mock values have been used, but ideally there would be a reputation metric for each data source that could be used to calculated the relative trustworthiness weight. If the data sources were to also transition into the blockchain network and operate their own nodes, then a metric could be devised that applies to all nodes, perhaps in the fashion of a network credit system.

2.6 Common Patterns

The subsection that follows describes common patterns that can be used, when writing smart contracts for the Ethereum blockchain application platform.

2.6.1 Authentication

A pattern that has emerged as the *de facto* standard for implementing "user permissions" is the Ownable smart contract template (see Implementation section 5.2.3). The template can be integrated with any smart contract, by means of the latter inheriting from the former. It stores the address of the current owner and provides basic functions to control authorization. It has an owner address property that is set to the message sender upon creation (in the constructor) and provides a function to transfer ownership to a new owner that is different to the old one. This is implemented by the use of the require function. Additionally, the function can only be called by the current owner, as specified by a modifier. Finally, an event is defined

and emitted, to indicate the successful completion of the ownership transfer. Events enable clients to subscribe to them and wait for confirmation (or a thrown error), before proceeding [65]. In that sense, they serve a similar purpose as an HTTP Status Response Code returned by an API.

2.6.2 Inheritance

In object-oriented programming languages, the focus is on classes and objects. In contrast, Solidity revolves around the concept of a contract. Consequently, a logical step is to think about how conventional concepts such as inheritance apply to this new paradigm. Solidity does support both inheritance and polymorphism, via the mechanisms of code copying and virtual function calls [66]. Multiple inheritance is possible and the implementation is not too dissimilar to the way it has been implemented in Python. Besides allowing the creation of maintainable modular code, inheritance provides for additional use cases. One contract can inherit from another, in order to be able to use its library functions or modifiers. For example, such a use is evident in the Registry smart contract of this project (see Implementation section 5.2.5), which inherits from both the Ownable and Oracle contracts.

2.6.3 Upgradability

Contractual agreements have to account for occasional changes in their terms [53]. Even laws are not fixed *ad eternum*. Smart contracts cannot be modified post deployment, though. This can sometimes be remedied via setter functions that allow (only) the owner of a contract to modify certain parameters stored in the state of the smart contract. However, there can be times when the changes are too drastic, thus necessitating functionality to upgrade contracts with a newer version. One way to deal with this is to simply shut them down via the self-destruct or the (now deprecated) suicide function [67]. These functions are only callable by the owner of the contract and the remaining ether balance is returned to their address. While useful under exceptional circumstances, for example as protection against a hack, killing the contract and deploying it anew to a different address is far from ideal, since all the references pointing to that address have to now be updated one by one, both within the internals of the system in question, as well as across all other systems using it that are developed and maintained by third parties, such as exchanges. Another major disadvantage is the loss of all data stored within the contract, since there is no built-in provision for back-ups or data recoverability.

A solution that has been implemented in the Ethereum development ecosystem is the use of proxies [68]. The state (data) is stored in the proxy, which does not change. The function calls, however, make use of the *delegate call* opcode, which allows them to be forwarded to another contract, the target [69]. This mechanism allows the swapping out of target contracts, while the proxy, along with all references to it, remains intact. The proxy contract acts as the storage layer and sits in-between the Ethereum client application the user is running and the contract containing the application logic (the behaviour contract) [70]. The issue with this approach is that all new versions need to conform to the storage structure used by the proxy. An approach that has been proposed is to utilise unstructured storage mechanisms, more specifically fixed slots of storage, and access them via in-line assembly [71]. This is reminiscent of memory organisation, for example the run-time stack/dynamic data segment [72].

3 Analysis

3.1 Requirements Listing

The goal of the analysis phase is to create a conceptual model of the system [73]. This process starts with requirements modelling, which aims to produce a full specification of the problem. That specification mainly includes the functions that the system is required to perform, performance and acceptance criteria and environmental constraints. The technical requirements are the most critical part of the software engineering process because it is quite challenging to change them later on [74].

The system analysed is named the Option Trading Platform (OPT). Sellers registering with the OPT and creating shops to offer their services to customers are called Option Writers. The customers are termed Option Buyers, correspondingly. The online shops are coded as smart contracts deployed on a public version of the Ethereum blockchain application platform (the Rinkeby test network [20], in this case). The shops themselves are called option factories, as they are used to create option contracts. The creation of new option contracts takes into account the specific features provided by the Writers and the parameters requested by the Buyers.

With regards to project management, in any project there is a need for prioritisation because time and other resources are scarce. The importance of requirements relative to one another has to be taken into account, in order to ensure that the development effort adheres to (often strict) deadlines. A technique that is commonly used in software engineering is MoSCoW prioritisation [75]. It is an abbreviation for the following labels:

- Must have
- Should have
- Could have
- Won't have

Thanks to using this technique, the implications of including a feature or opting to exclude it are made much more explicit than simply denoting it as 'high' or 'low' importance. As a result, the planning process is more organised and effective.

The requirements specifications, together with their MoSCoW characterisations, are presented in Figure 3.1, on the next page.

ID	Specification	Type (F/N)*	Priority (MoSCoW)
1. Option Writer Functionality			
RQ1	The Option Trading Platform shall allow an Option Writer to register and create a smart contract representing their services and selling to Buyers	F	M
RQ2	The Option Trading Platform shall record and keep track of the options sold to Buyers from each Writer	F	M
RQ3	The Option Trading Platform shall automatically calculate the amount to pay out for each exercised option	F	M
RQ4	The Option Trading Platform shall keep track of internal balances	F	M
RQ5	The Option Trading Platform shall allow an Option Writer to view a list of past transactions	F	M
RQ6	The Option Trading Platform shall allow an Option Writer to modify their credentials and/or contract features	F	M
RQ7	The Option Trading Platform shall send alerts to the Option Writer	F	M
2. Option Buyer Functionality			
RQ8	The Option Trading Platform shall allow an Option Buyer to register	F	M
RQ9	The Option Trading Platform shall allow an Option Buyer to search for and view relevant market information	F	M
RQ10	The Option Trading Platform shall allow an Option Buyer to retrieve and view a list of different Option Writers	F	M
RQ11	The Option Trading Platform shall allow an Option Buyer to input the required parameters and view the premium rates offered by different Option Writers	F	M
RQ12	The Option Trading Platform shall allow an Option Buyer to buy options and have the parameters stored in the smart contracts	F	M
RQ13	The Option Trading Platform shall allow an Option Buyer to view a list of options bought, along with expiration dates	F	M
RQ14	The Option Trading Platform shall allow an Option Buyer to exercise their options that are past expiration date	F	M
RQ15	The Option Trading Platform shall send alerts to the Option Buyer	F	M
3. Maintenance			
RQ16	The Option Trading Platform shall store data in the smart contracts running on the public Ethereum peer-to-peer blockchain network	F	M
RQ17	The Option Trading Platform shall allow a System Administrator to terminate a running contract	F	S
4. Login/Security			
RQ18	The Option Trading Platform shall distinguish between different types of logged-in users: Option Writer, Option Buyer, System Administrator	F	M
RQ19	The Option Trading Platform shall allow a User to register as an Option Writer or Option Buyer	F	M
RQ20	The Option Trading Platform shall check if the supplied email address has already been registered	F	M
RQ21	The Option Trading Platform shall validate Option Writer/Option Buyer credentials upon logging in and before granting access	F	M
RQ22	The Option Trading Platform shall assess User password strength and enforce a recommended level	F	S
5. Performance			
RQ23	The Option Trading Platform shall be operational 24/7	N	S
RQ24	The Option Trading Platform shall support a load of 100 concurrent users under standard operating conditions	N	S
RQ25	The Option Trading Platform shall validate User login credentials and process all other transactions in an amount of time appropriate for the public Ethereum blockchain network	N	S
RQ26	The Option Trading Platform's sync time to the latest confirmed block shall not exceed 1 hour	N	S
6. Accessibility			
RQ27	The Option Trading Platform shall use a suitably configured web browser as its User interface client and support the most popular options (Mist, Chrome with the MetaMask plug-in)	N	M
RQ28	The Option Trading Platform's client application shall support and adapt to different screen sizes	N	S
RQ29	The Option Trading Platform could support multiple languages	N	C
	* (F) Functional requirements	**22**	
	* (N) Non-functional requirements	**7**	
	All requirements	**29**	

Figure 3.1: Requirements of the Option Trading Platform

3.2 Use Case Analysis

Following on from the analysis of requirements, the next step is use case modelling. Use case modelling starts with defining the system and its boundaries, along with the distinct actors who use it [76]. Each use case describes an interaction between the actors and the system. Here the critical interaction happens between a user (via the front-end application running in a web browser) and the smart contracts deployed and running on the nodes making up the blockchain network (the back-end). Each use case may have one or more alternative scenarios that could unfold; for example, a user could attempt to login and the credentials could either be correct or invalid. Use cases were consulted and reevaluated throughout the development process, to ensure that they are a realistic representation of the system and that it performs the functions intended.

The use case analysis begins with a Domain Model Diagram (Figure 3.2), then continues with a Use Case Listing (Table 3.1) and finally illustrates the outcome through a Use Case Diagram (Figure 3.3). It is subsequently validated using a Requirements - Use Cases Traceability Matrix (Figure 3.4), which checks the use cases against the requirements they correspond to.

3.3 Domain Model Diagram

Domain modelling is the creation of an abstract model of a system, depicting the entities and their relationships [76]. The diagram produced is refined after a few iterations and then feeds into the use case listing and specifications.

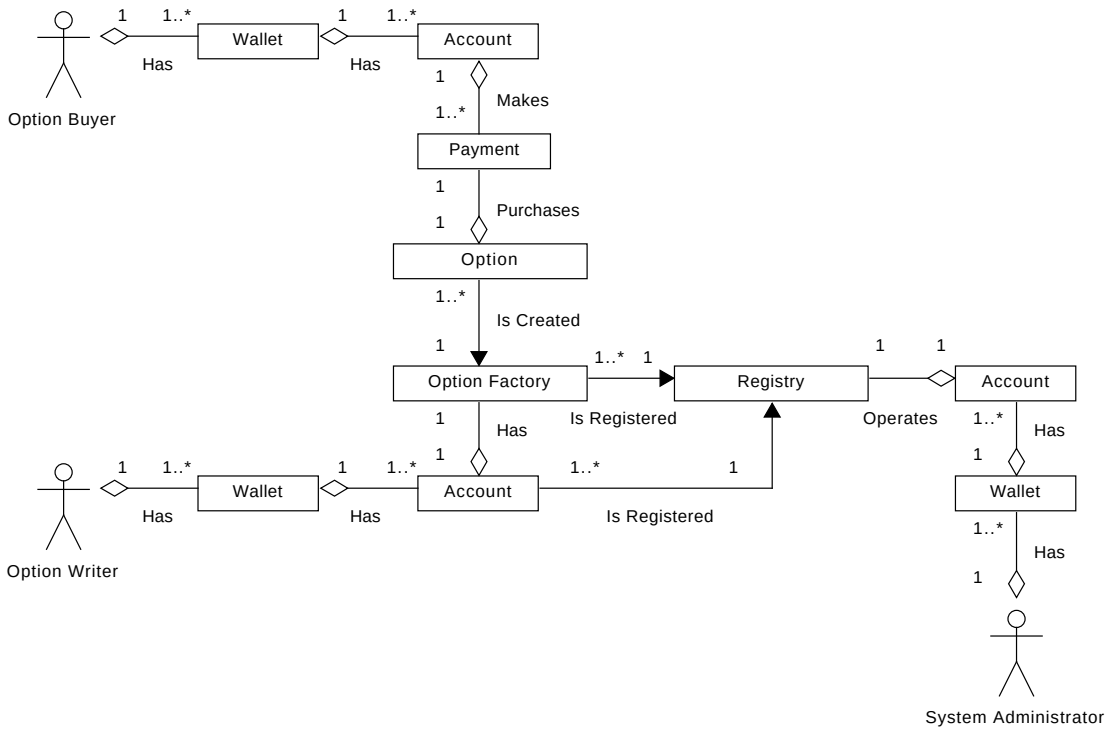

Figure 3.2: Domain model diagram

As a side note and as is shown in the diagram above, each user can have many wallets and each wallet can in turn have many accounts, but the option factory is controlled by only one of those accounts, since it is tied to (owned by) a single network address that belongs to that account. The same is true for the registry. Alternative configurations (utilising multisignature schemes, for example) are possible, but are beyond the purposes and the scope of this project.

3.4 Use Case Listing

The domain model analysis resulted in a list of actors and use cases, mapping out the entirety of the system's functionality. Three actors were identified, each with their corresponding use cases: Option Writer, Option Buyer and System Administrator. A subset of activities were shared between actors. The list of use cases was reviewed throughout the project's life-cycle, in order to maintain faithfulness and for amendments, as needed. A detailed description of each use case is given in Section 3.6.

Table 3.1: List of use cases and associated actors

ID	Use Case	Actors
UC1	Login	Option Writer, Option Buyer, System Administrator
UC2	Register Seller	Option Writer
UC3	Register Buyer	Option Buyer
UC4	View Market Rates	Option Writer, Option Buyer
UC5	Find Option Writer	Option Buyer
UC6	View Option Writer Page	Option Buyer
UC7	Purchase Option	Option Buyer
UC8	View Personal Profile	Option Writer, Option Buyer, System Administrator
UC9	Edit Profile	Option Writer, Option Buyer
UC10	View Past Transactions	Option Writer, Option Buyer
UC11	Edit Services	Option Writer
UC12	View Administrator Dashboard	System Administrator

3.5 Use Case Diagram

The use case diagram is a UML diagram depicting a system, its boundaries and actors who interact with it to accomplish certain tasks, described via use cases [77]. The diagram clarifies the associations between the actors and the use cases, as well as among different use cases. It presents a static view of the system and helps in the modelling of relationships and behaviours. Use case diagrams are one of the five main types of UML diagrams used to model and contextualise a system's dynamic behaviour and aspects.

The use case diagram created for this project is presented below, in Figure 3.3.

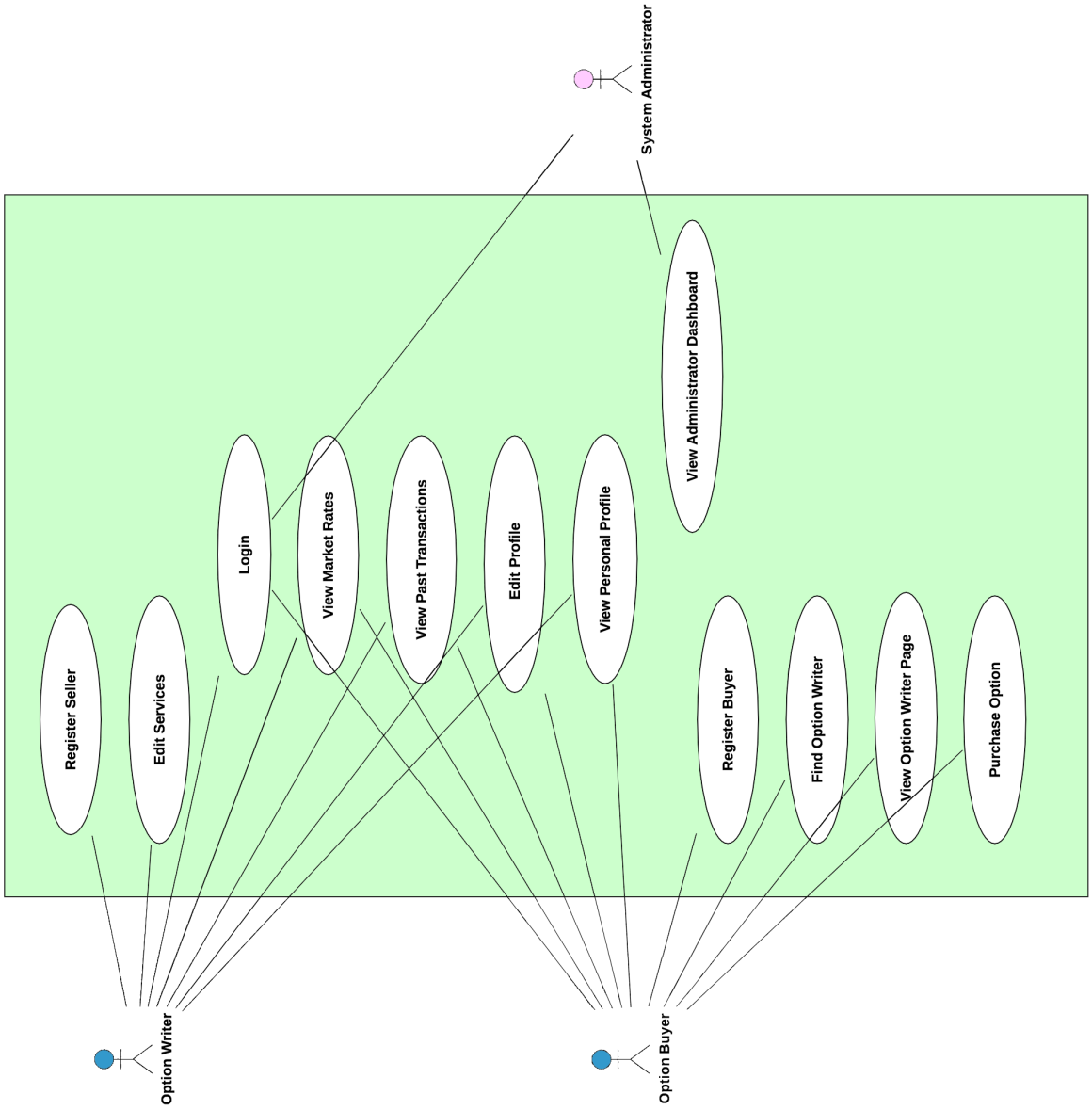

Figure 3.3: Use case diagram

3.6 Use Case Specifications

What follows is a textual description of each use case, detailing an interaction between an actor and the system, in order to perform a certain task or accomplish an objective. In some cases, alternative or exception flows have to be accounted for, in order to provide a full picture of the possible paths or outcomes. One example of this is the exception that occurs when a user enters invalid login credentials, as part of Use Case 1 (Login). The exception flow has the precondition of the user already being registered with the system and having attempted to login with invalid credentials. The steps taken are that the system triggers an alert to notify the user of the authentication error, access to the main pages of the application is withheld and the user is prompted to try entering the correct username and password again. These alternate flows demonstrate the various scenarios that might take place, for a given general use case.

It can be observed that Register Seller and Register Buyer share certain elements of the main flow logic, however it was a conscious design decision to have them as separate use cases, rather than alternative flows. Besides the difference in details required, the main underlying reason is the difference between the scope of a Seller, who is able to create an option factory and offer their services, versus that of a Buyer, who is able to simply register and browse the marketplace. Each type of account is thus meant to have different features and functionalities associated with them, even though a user could theoretically act as both Buyer and Seller. As a consequence of the aforementioned, it was decided to enforce this separation between types of accounts in the use cases, as well.

The use case specification tables follow, from the next page on.

Table 3.2: Use case: Login

Use Case	Login
ID	UC1
Brief Description	A User enters login credentials to gain access to the application
Primary Actors	Option Writer / Option Buyer
Secondary Actors	None
Preconditions	A User is already registered as an Option Writer, Option Buyer, or System Administrator
Main Flow	1. A User navigates to the Login Page 2. A User enters their login credentials (username, password) 3. The System checks the login credentials provided against the ones stored in the database *(See exception flow)* 4. The System allows access
Postconditions	The logged-in User can now access the main pages of the application

Table 3.3: Use case exception flow: Login: invalid credentials

Exception Flow	Login: Invalid Credentials
ID	UC1.1
Brief Description	A User is not successful in logging in, due to incorrect credentials
Primary Actors	Option Writer / Option Buyer
Secondary Actors	None
Preconditions	A User has already registered an account as Option Writer or Option Buyer
Main Flow	1. This exception flow begins after the third step of the main use case flow 2. The System alerts the User of the unsuccessful login attempt and denies access to the main application pages
Postconditions	The User remains at the Login Page and is given the option to try again

Table 3.4: Use case: Register Seller

Use Case	Register Seller
ID	UC2
Brief Description	A User registers as a Seller
Primary Actors	User
Secondary Actors	None
Preconditions	A User has navigated to the Registration Page
Main Flow	1. A User selects the option to register as a Seller 2. A User inputs First Name, Last Name, Email Address, Username, Country of Residence *(See exception flow)* 3. The System creates an Option Writer with the supplied information in the database 4. The System notifies the User of the success of the registration via an on-screen confirmation message 5. The System gives the User the opportunity to customise and deploy their Option Factory
Postconditions	The User receives an on-screen notification of their successful registration as an Option Writer

Table 3.5: Use case exception flow: Register Seller: invalid input

Exception Flow	Register Seller: Invalid Input
ID	UC2.1
Brief Description	The User is unsuccessful in registering as an Option Writer, due to invalid input
Primary Actors	User
Secondary Actors	None
Preconditions	The User has navigated to the Registration Page
Main Flow	1. The exception flow begins after the third step of the main flow 2. The System notifies the User that they have entered invalid registration details via an on-screen message 3. The System gives the User the option to try again
Postconditions	The User is given the opportunity to enter valid registration details

Table 3.6: Use case: Register Buyer

Use Case	Register Buyer
ID	UC3
Brief Description	A User registers as an Option Buyer
Primary Actors	User
Secondary Actors	None
Preconditions	A User has navigated to the Registration Page
Main Flow	1. A User selects the option to register as an Option Buyer 2. A User inputs First Name, Last Name, Email Address, Username, Country of Residence *(See exception flow)* 3. The System creates an Option Buyer with the supplied information in the database 4. The System notifies the User of the success of the registration via an on-screen confirmation message
Postconditions	The User receives an on-screen notification of their successful registration as an Option Buyer

Table 3.7: Use case exception flow: Register Buyer: invalid input

Exception Flow	Register Buyer: Invalid Input
ID	UC3.1
Brief Description	The User is unsuccessful in registering as an Option Buyer, due to invalid input
Primary Actors	User
Secondary Actors	None
Preconditions	The User has navigated to the Registration Page
Main Flow	1. The exception flow begins after the third step of the main flow 2. The System notifies the User that they have entered invalid registration details via an on-screen message 3. The System gives the User the option to try again
Postconditions	The User is given the opportunity to enter valid registration details

Table 3.8: Use case: View Market Rates

Use Case	View Market Rates
ID	UC4
Brief Description	An Option Writer/Buyer navigates to the Dashboard Page and views market rates for various securities
Primary Actors	Option Writer / Option Buyer
Secondary Actors	None
Preconditions	A User has registered as an Option Writer/Buyer *(Register Seller/Buyer Use Case)* An Option Writer/Buyer is logged-in *(Login Use Case)*
Main Flow	1. The Option Writer/Buyer navigates to the Dashboard Page 2. The Option Writer/Buyer selects different rates to view 3. The System retrieves and displays the requested information
Postconditions	None

Table 3.9: Use case: Find Option Writer

Use Case	Find Option Writer
ID	UC5
Brief Description	An Option Buyer searches through the list of Option Writers, filtering based on types of options and instruments provided
Primary Actors	Option Buyer
Secondary Actors	None
Preconditions	An Option Buyer has navigated to the main page of the application
Main Flow	1. The Option Buyer selects search options 2. The System retrieves the results matching the search terms from storage on the blockchain 4. The System displays the retrieved results
Postconditions	The Option Buyer can now choose an Option Writer that offers the requested instruments and view their page

Table 3.10: Use case: View Option Writer Page

Use Case	View Option Writer Page
ID	UC6
Brief Description	An Option Buyer views an Option Writer's page, which includes information regarding the types of financial instruments available and the rates offered
Primary Actors	Option Buyer
Secondary Actors	None
Preconditions	The Option Buyer is logged-in *(Login Use Case)* The Option Buyer has searched for Option Writers *(Find Option Writer Use Case)*
Main Flow	1. The Option Buyer selects an Option Writer from the results of *Find Option Writer* 2. The System navigates to the selected Option Writer's page 3. The System retrieves the relevant information from the storage on the blockchain and displays them
Postconditions	None

Table 3.11: Use case: Purchase Option

Use Case	Purchase Option
ID	UC7
Brief Description	An Option Buyer selects an instrument to buy from an Option Writer
Primary Actors	Option Buyer
Secondary Actors	None
Preconditions	An Option Writer has registered successfully with the System and filled in their offerings information *(Register Seller Use Case)* An Option Buyer has searched for an Option Writer *(Find Option Writer Use Case)* The Option Buyer has selected the specific Option Writer *(View Option Writer Page Use Case)*
Main Flow	1. The Option Buyer selects a financial instrument from the *View Option Writer Page* 2. The Option Buyer enters the necessary input and purchases the instrument 3. The System stores the purchase information on the blockchain 4. The System notifies the Option Buyer of the success of their purchase via an on-screen confirmation message
Postconditions	The System sends an on-screen message to the Option Writer as well, notifying them of the sale

Table 3.12: Use case: View Personal Profile

Use Case	View Personal Profile
ID	UC8
Brief Description	Option Writers, Buyers and System Administrators can view their own profile
Primary Actors	Option Writer / Option Buyer / System Administrator
Secondary Actors	None
Preconditions	An Option Writer/Option Buyer/System Administrator has logged into the System
Main Flow	1. An Option Writer/Option Buyer/System Administrator navigates to their personal Profile Page 2. The System retrieves the relevant profile information from the blockchain and displays them
Postconditions	None

Table 3.13: Use case: Edit Profile

Use Case	Edit Profile
ID	UC9
Brief Description	An Option Writer/Option Buyer updates their details
Primary Actors	Option Writer / Option Buyer
Secondary Actors	None
Preconditions	An Option Writer/Option Buyer has already registered successfully with the System *(Register Seller/Buyer Use Case)* The Option Writer/Option Buyer has logged into the System *(Login Use Case)*
Main Flow	1. The Option Writer/Option Buyer navigates to their Profile Page 2. The System retrieves and displays their information 3. The Option Writer/Option Buyer selects the option to edit their details 4. The System displays an editable version of the Page and awaits input 5. The Option Writer/Option Buyer enters the updated information and selects the option to save the changes 6. The System stores the new information to the blockchain and displays the updated Page
Postconditions	The Option Writer/Option Buyer profile information is updated

Table 3.14: Use case: View Past Transactions

Use Case	View Past Transactions
ID	UC10
Brief Description	An Option Writer/Option Buyer views their transaction history
Primary Actors	Option Writer / Option Buyer
Secondary Actors	None
Preconditions	An Option Writer/Option Buyer has logged into the System *(Login Use Case)*
Main Flow	1. An Option Writer/Option Buyer navigates to the Home Page 2. The System retrieves all past transactions involving that Option Writer/Buyer 3. The System displays the transaction history
Postconditions	None

Table 3.15: Use case: Edit Services

Use Case	Edit Services
ID	UC11
Brief Description	An Option Writer updates the option contracts they are offering
Primary Actors	Option Writer
Secondary Actors	None
Preconditions	An Option Writer has successfully registered with the System *(Register Seller Use Case)* An Option Writer has logged in *(Login Use Case)*
Main Flow	1. The Option Writer selects the option to edit the contracts they offer 2. The System enables editing mode 3. The Option Writer selects the contracts that they would like to offer, along with corresponding details 4. The Option Writer selects the option to save the changes 5. The System stores the changes in the blockchain and displays the updated Page
Postconditions	The Option Writer's contract offering is updated with the changes

Table 3.16: Use case: View Administrator Dashboard

Use Case	View Administrator Dashboard
ID	UC12
Brief Description	A System administrator views the Administrator Dashboard
Primary Actors	System Administrator
Secondary Actors	None
Preconditions	A System Administrator has logged into the System
Main Flow	1. The System Administrator selects the option to view the Administrator Dashboard, which is only visible when logged-in as an Administrator 2. The System retrieves the information relevant for an Administrator and displays the Dashboard
Postconditions	The Administrator Dashboard is now visible

3.7 Traceability Matrix

The traceability matrix maps requirements to use cases and helps ensure that they have all been taken into account, thus guaranteeing completeness of the analysis process [78]. It is used as a tracking tool throughout the progression of the project, but could also be used for subsequent testing towards the conclusion of the project's life-cycle [79]. Through the creation of the traceability matrix, it was confirmed that there were no missing requirements. It is presented in Figure 3.4 below.

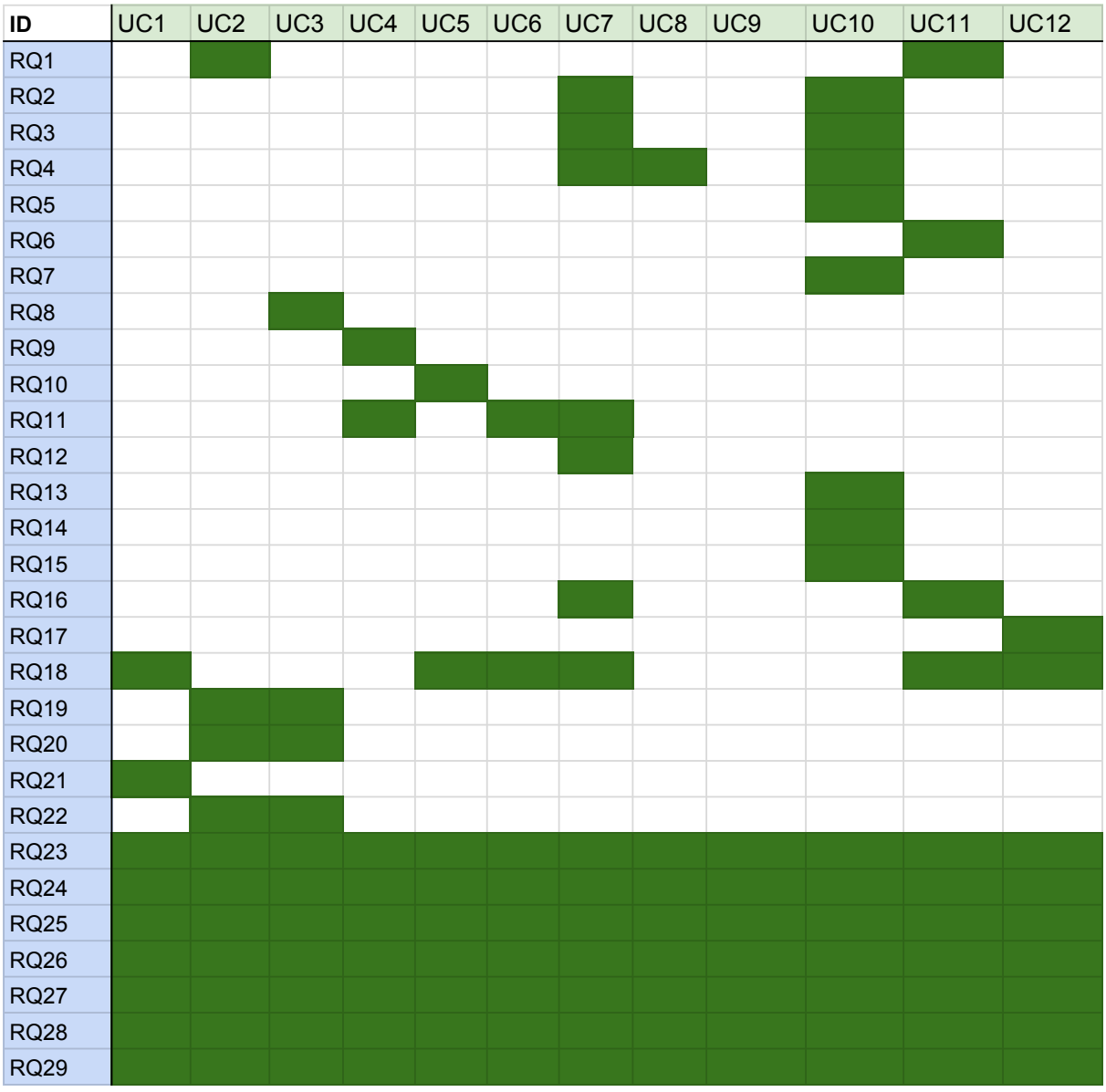

Figure 3.4: Requirements / use cases traceability matrix

4 System Design

4.1 Typical Architecture

This section describes the standard architecture of a decentralised application [80]. The end user of the system is running the front-end client in their web browser. The client is used to interact with a server application (usually written in Express/Node.js) and download the HTML and JavaScript code to display and run the web application. The client also interfaces with the Ethereum node via the injected web3 provider [81], [82]. This provider is self-contained in the browser (or the server [83]) and encrypted for security reasons. It acts as the gateway enabling access to a node of the peer-to-peer Ethereum blockchain network, where the Solidity smart contract code is deployed. The smart contract code is executed by instances of the Ethereum Virtual Machine (EVM) that all network nodes are running [84], [85].

An high-level overview of the architectural components of the system and their interactions is presented in Figure 4.1[a].

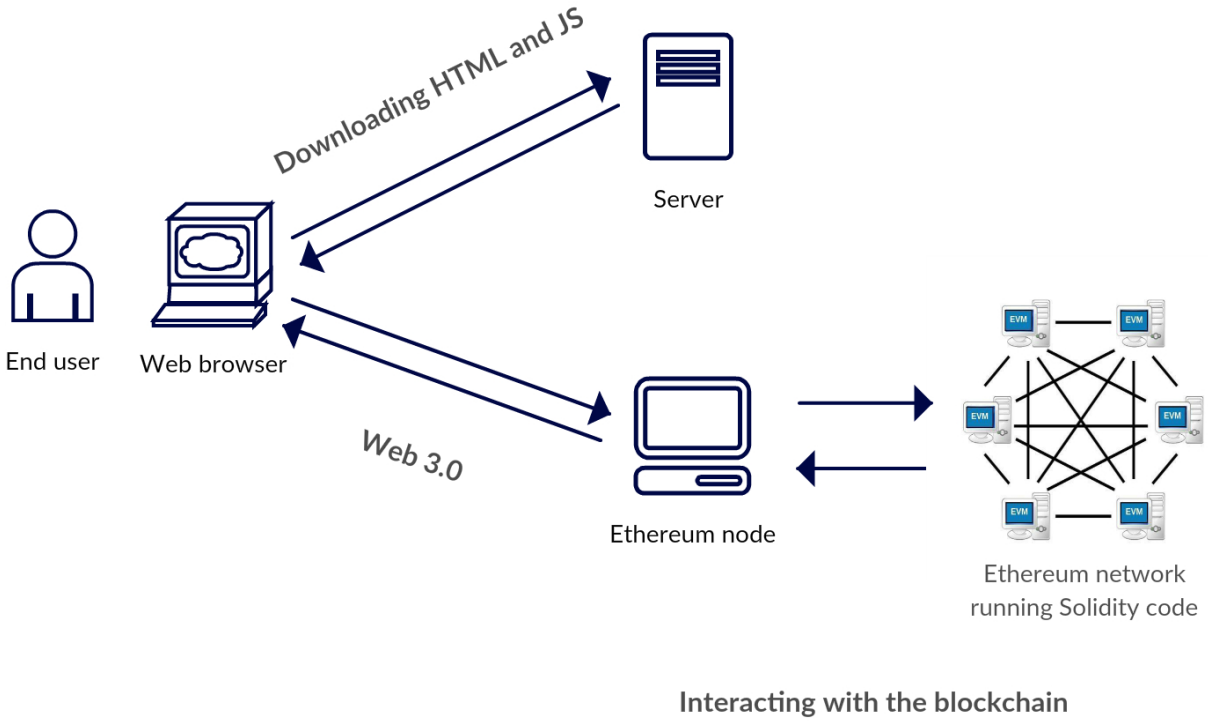

Figure 4.1: Decentralised application architecture

The critical aspect that differentiates the structure of a decentralised application from that of a traditional one is interaction with the Ethereum blockchain, which is achieved via the web3 module. The front-end

[a]The illustration of the Ethereum peer-to-peer network of computers running the EVM, which makes up the bottom-right hand part of the image, was incorporated from [86].

itself can be written in a combination of plain HTML/CSS/JS, or any framework implementing the Model-View-Controller architectural pattern.

The smart contracts are similar to the back-end part of an application in that they serve as data storage, but they also, more importantly, encompass the application's business logic. After the contracts are written, compiled and deployed on the network, an Application Binary Interface (ABI) JSON object is returned. This ABI contains the instructions regarding the interface via which interaction with the smart contract can take place. Utilising the ABI enables access to the smart contract, either to read from its permanent storage or call its public/external functions and write new information that is being sent as part of the payload of transactions. The unit of currency (Ether in this case), the code of the smart contract and the data transmitted as the payload all reside in the blockchain, affording thus the possibility of combining function calls with transfers of value and executing both with a single transaction.

4.2 Deployment Diagram

A deployment diagram in the Unified Modeling Language depicts a view of the hardware nodes of the system and the software artifacts that are deployed and running on them at run-time [87]. It also includes the middle-ware that connect those separate devices. Software artifacts refer to the definite results of the development process, such as executable files or libraries [88]. Deployment diagrams are particularly useful for distributed systems with a complex architecture, since they elucidate how the different components interoperate.

Figure 4.2 presents the deployment diagram for the Option Trading Platform. As stated in the key, the two main building blocks are hardware nodes (or alternative execution environments) and software components. Physical nodes are labeled as *device*, to indicate their material status. Connections are illustrated via lines and their accompanying stereotypes. The web browser serves as the client interface and connects via HTTP to the server, which delivers the web page content of the application. There is a messaging connection to a central application server, with support for accounts and other utilities. The *User* model implements the Façade design pattern [89], meaning that it implements the public interface of the User component. This helps control access to it from other external components [90].

More interestingly, the web browser communicates with the Ethereum node via JSON IPC. The use of IPC rather than RPC in the diagram is worth elaborating. IPC stands for Inter-process Communication and it indicates a local connection, while RPC a remote one [91]. This means that the web application process connects to a local node (via the IPC pipe the Ethereum client geth creates or via the injected web3 provider). The Ethereum client takes care of all account functionality and also acts as a wallet enabling the signing of transactions and transfer of funds. Because creating transactions by writing their information in JSON format is not that user-friendly, web3 provides helper functions that abstract away the minutiae and enable signing and publishing a transaction with relative ease. The transaction is then broadcasted to the Ethereum blockchain network, which consists of all the nodes running an instance of the Ethereum Virtual Machine (EVM). As an example, the source code of a smart contract is first compiled to byte-code and deployed to the peer-to-peer network. The contract is then assigned a fixed address to reside at. From that point on, the nodes running an instance of the EVM can execute the code in a deterministic manner.

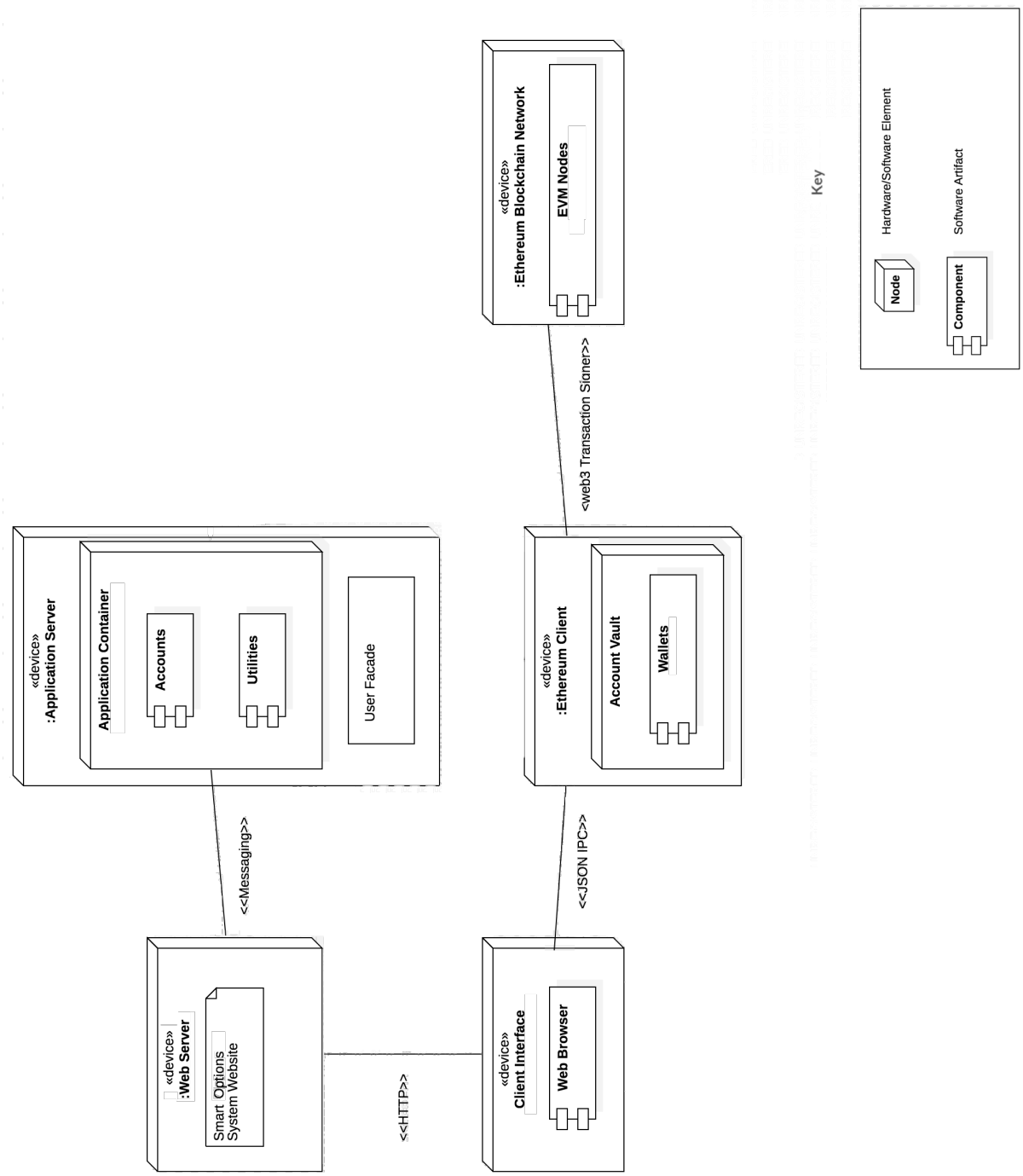

Figure 4.2: Deployment diagram illustrating the nodes in the system, their links and the software artifacts running on them

4.3 Component Diagram

The component diagram organises and displays the parts of the system as logical components [92]. Each part relates to a different aspect of the system's functionality. When put together, they form the architectural specification of the whole system. The structure is aligned with the typical three-tier model, which draws a distinction between boundary classes (views), controllers (application layer) and back-end infrastructure components [93].

The components of the Option Trading Platform can therefore be grouped into three categories, as presented in the Component Diagram in Figure 4.3:

- *The presentation layer*, which consists of the various boundary classes (views) of the application. They are responsible for receiving the output from the application layer and communicating that to the user via the rendering of the different views making up the user interface. They are also in charge of receiving user input/commands and passing that on as actions to the other layers. Acting as a bridge between the controllers and the boundary classes is an Observer Interface.
- *The application layer*, which contains the components (controllers) relating to the business/domain logic of the system [93]. The application layer sits in-between the presentation and the data source, mediating their interaction either partially or (sometimes) completely. The Accounts, Option Contracts and Utilities components represent the packages encapsulating the functionality of the application.
- *The data access layer*, which involves those components carrying out the back-end tasks that the system requires. This category includes other applications that provide data (such as the oracles interacting with APIs in this case). In most applications it would typically also include the database component responsible for persistent storage [93]. This project includes a blockchain for data access (storage and retrieval), rather than a central database. However, a database would come in handy, should one wish to extend the system with more functionality and/or data that would be difficult to have solely on-chain.

The diagram also displays the components' interfaces, using common UML 2 notation. The interfaces represent a collection of attributes and methods making up the behaviour of that respective part of the system [94]. Furthermore, the dependencies between individual components and each other's interfaces are illustrated in the diagram, as well.

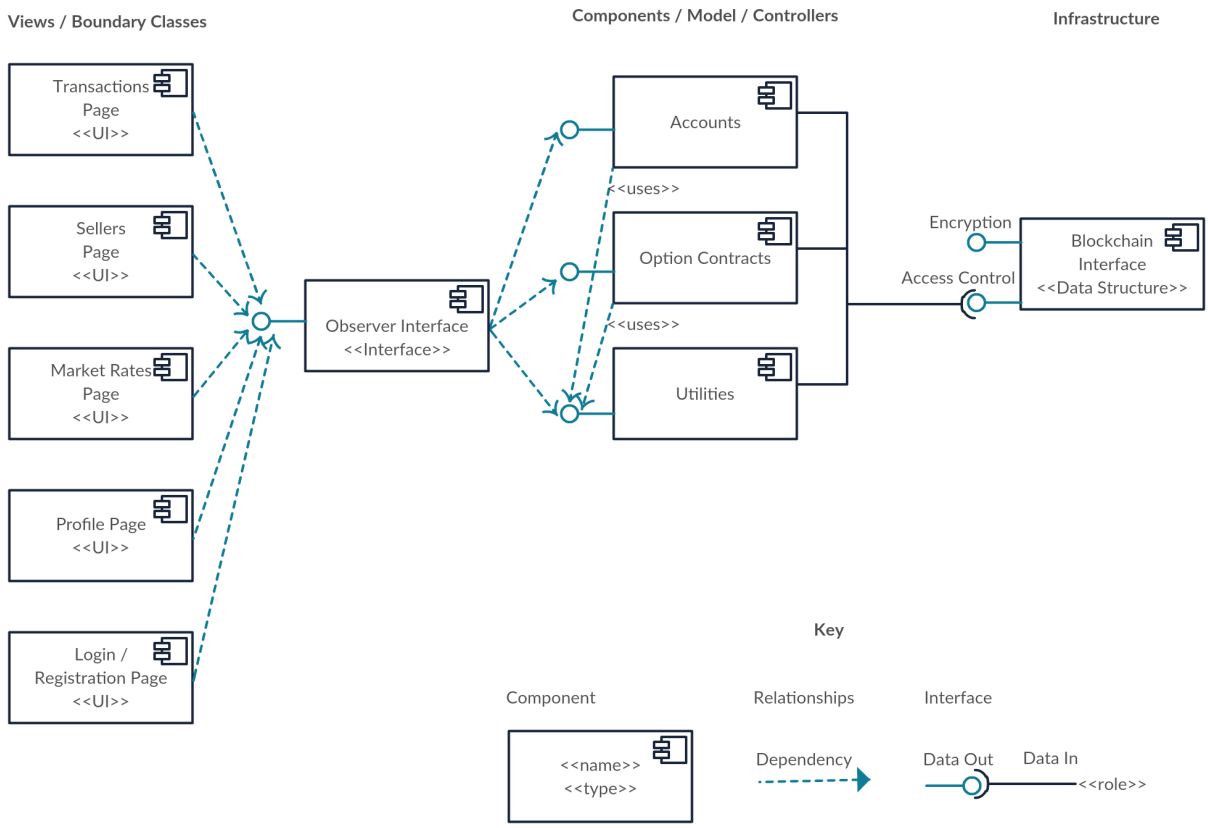

Figure 4.3: Component diagram

4.4 Discussion of Design Decisions

4.4.1 Class Diagram

The class diagram is a representation of the static elements of a model, together with what they contain and how they relate to each other [92]. Dynamic aspects such as operations are included, but not elaborated upon in this diagram. The main elements depicted are classes, with their associations and dependencies. However, for the purposes of this project, classes have been substituted with contracts. As a result, the class diagram is essentially a contract diagram.

Figure 4.4 illustrates how the contracts contain attributes and operations, but also modifiers and events, which are language-specific concepts. A modifier can be attached to any function to alter its behaviour, for example restricting access [95]. It is thus not merely a peculiarity of implementation, but a strategic functionality-related mechanism. Events are of similar importance because they determine what transaction logging information can be passed on the user via callbacks [65].

The diagram shows how all main contracts inherit from the *Ownable* template. The registry smart contract contains a number of sellers, with their respective information. It interfaces and makes use of one or more oracles, in order to retrieve price information from them. The oracles thus act as data feeds, querying the APIs and updating a record of the price. There could simply be a single oracle retrieving prices from all APIs, but having separate contracts deployed for each data source more closely resembles how the system would ideally be, if the data providers were actually nodes in the network, as well. This design also helps modularise the system and makes it more resistant to attack, which is why a committee of oracles with a weighted voting scheme to determine the aggregate price was introduced. If one data feed started acting in a malicious manner or they got hacked, their relative weight could be reduced (or even brought down to zero), which would protect the integrity of the platform to some extent. When it comes to the sellers, each one of them is associated with one option factory, thus creating an one-to-one relationship. The option factory is essentially the seller's own shop and it is responsible for the most fundamental functions within the system, which explains its larger size compared to the other elements. It contains attributes, modifiers, functions and events related to option purchasing behaviour, as well as utility ones like *deposit*, *withdraw* and *getBalance*. Deposits, withdrawals and balances help buyers do business with sellers that they often interact with, so that they can simply settle balances within the contract every time they exercise an option and only conduct a transfer across the network (as a withdrawal) once in a while. This helps them save on fees and reduces the amount of slow transactions that have to take place, thus eliminating some of the friction associated with using a public blockchain network. Finally, each option factory stores the options it has sold to buyers, which contain data members with information about the underlying asset, the exercise price, the expiration date and whether the option has already been exercised or not (to prevent double-exercising).

The diagram is presented below, in Figure 4.4.

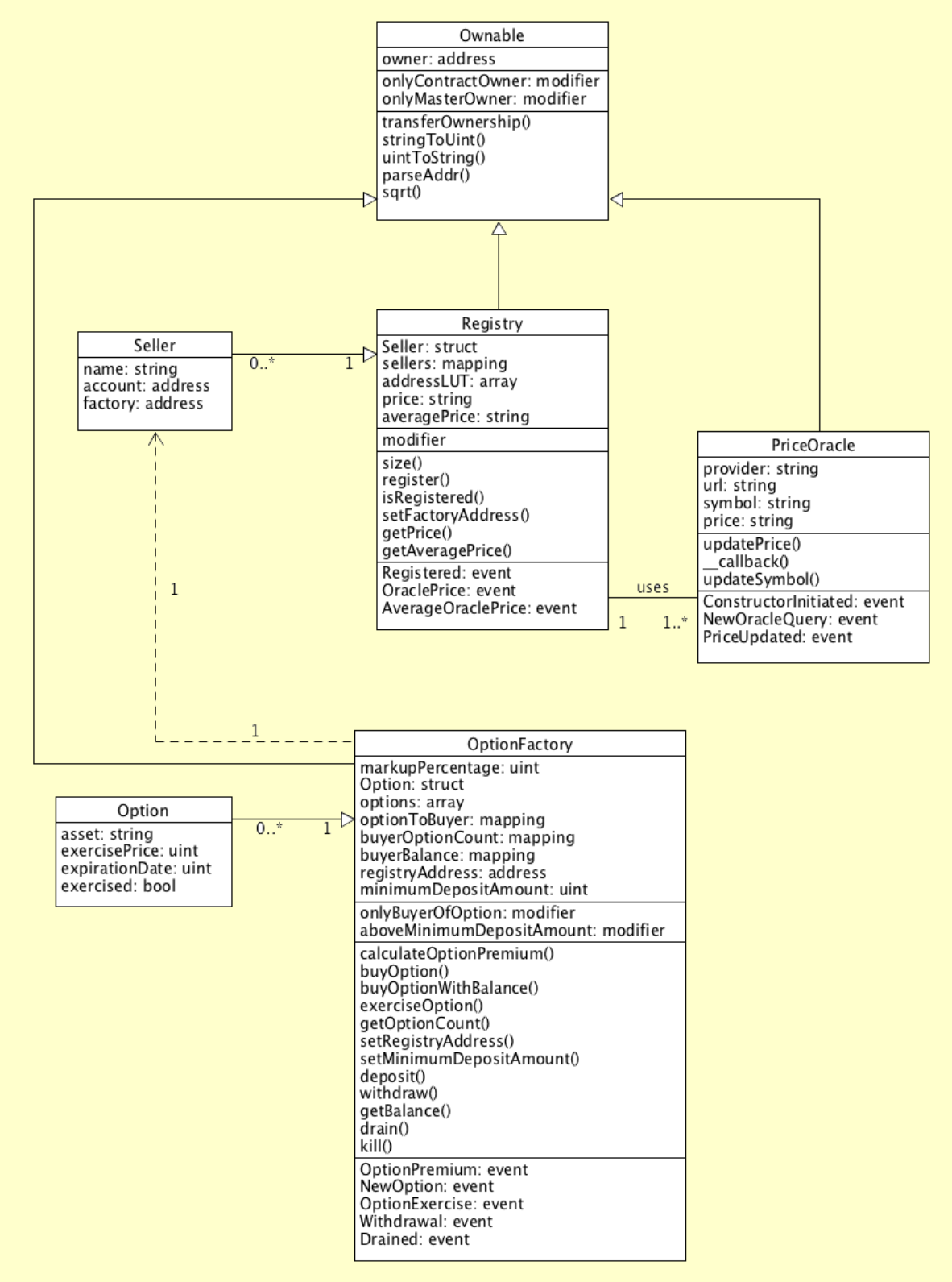

Figure 4.4: Analysis class diagram, depicting the relationships between contracts in the Option Trading Platform

4.4.2 Ethereum Client

The implementation of an Ethereum client capable of interacting with the blockchain network presents significant modularity and security challenges. If the network consisted only of users with permissioned (authorised) access, meaning that each user had been authenticated and was at the same time an actual node in the network, then it would be safe to assume that each one of them would have their own node set up, synchronised and running locally on their machine. They would then be able to connect, sign and send transactions to the network by themselves, not requiring outside assistance and also potentially not having to pay any fees to external miners or validators to process their transactions.

However, for the purposes of an application accessible from any (properly configured) web browser and one that connects to a public network, expecting all users to be running their own nodes before attempting access is impractical and not realistically feasible. In a world where many transactions and sales are still agreed upon and finalised over the phone, requiring that users go through a highly technical process of installation and configuration beforehand would introduce unnecessary friction. The outcome would be a drastic reduction in the user-friendliness of the application.

The solution of choice is to use a light client-side wallet, such as MetaMask [82], which is an approach that promotes modularity. The light wallet can interact either with a full node hosted locally or with a third party provider [96]. It is also designed to be more secure because the private information is encrypted and stored in the browser and all sensitive functions, for example hashing or signing, take place on the client's side (using JSON and IPC channels). The implication of this behaviour is that the seed and the private keys used to sign transactions never get transmitted outside the client machine. Therefore, the balances of the account in the wallet cannot easily be stolen by the providers.

The following diagram in [97, Fig. 4.5] illustrates how the web application written in JavaScript, the web3 library components and the MetaMask wallet extension all reside on the client side, for example running within a web browser.

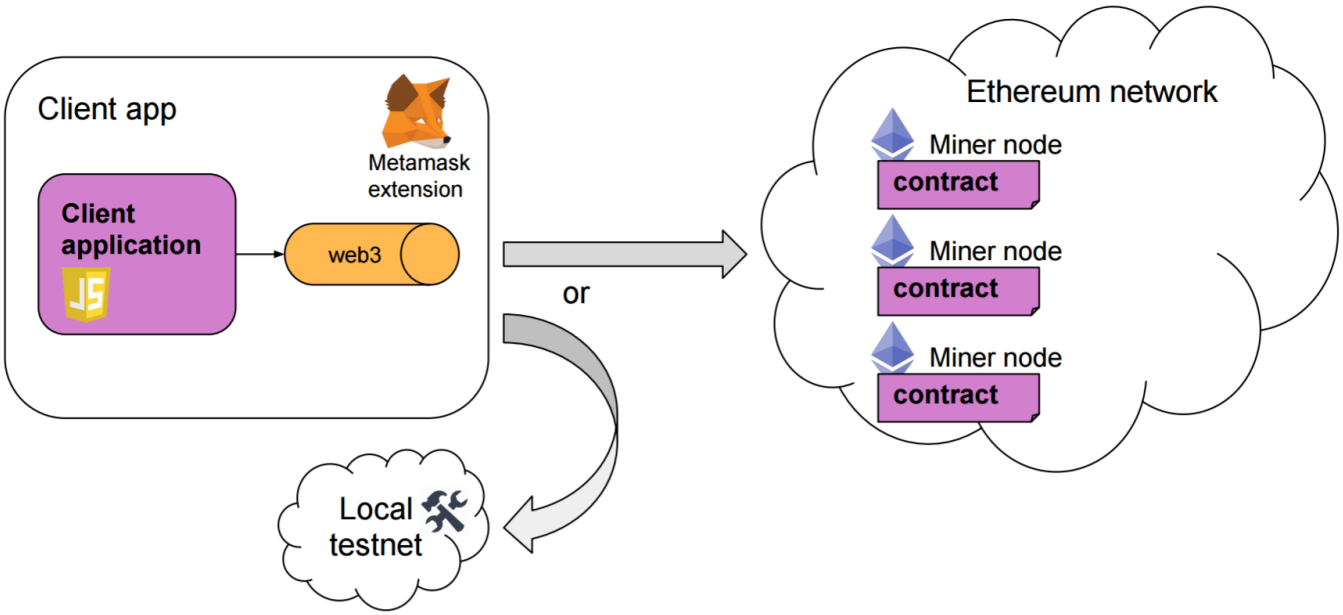

Figure 4.5: A diagram illustrating the typical architecture of a DApp [97]

4.4.3 Connection to Ethereum Clients

An Ethereum client is an implementation of the protocol in a certain programming language [98]. According to the Ethereum documentation [99], connection to an Ethereum client is possible over the JSON-RPC remote procedure call protocol. Various methods are exposed that provide an application with access. This direct interaction, however, presents a number of challenges. To begin with, it requires an implementation of the JSON-RPC protocol and the use of binary format. The documentation states additional limitations, such as restrictions with regards to the size of numeric types (256 bits) [99]. Finally, there has to be support for administrative commands, such as creating, storing and handling addresses, as well as incorporating the ability to sign transactions – wallet functionality, in general. As a result, it is preferable to use a library such as web3 [81] that already takes care of a lot of these issues in a secure manner, to enable the development effort to be concentrated on the actual application.

4.4.4 Data Storage

When it comes to data storage, a dilemma arises. One option is to store everything on a blockchain data structure. The advantage of such a decentralised architecture would be greatly enhanced modularity. According to that paradigm, each subgroup of nodes in the peer-to-peer network could act differently by running a modified version of the protocol, since they would be responsible for a different part of the stack. The types of nodes in the network could range from ones dedicated to serving the end users (client applications) all the way to purpose-built machines for data storage. In such a set-up, the entirety of the data logic would be taken care of by smart contracts deployed and running on the blockchain.

However, storing data in that manner is still far from being cost-efficient and scalable enough to allow for a termination of the use of centralised databases and a full-scale switch to a decentralised paradigm. That being said, there are decentralised platform implementations under development that ultimately aim to solve the storage and content distribution issue, such as IPFS [100] and Swarm [101]. Swarm aims to be the layer that serves static assets, while another protocol, Whisper [102], is aiming to be the solution for handling real-time communication. The user machine would then be tasked with simply running the client interface and handling authentication. These protocols are still in a nascent and experimental phase, though. Consequently, the decentralised application stack usually has to be supplemented by a traditional back-end, at least for the time being. One ramification of this conundrum is that at least some of the data are still in a central location and are often unencrypted, resulting in higher levels of data breach risk, as well as heightened susceptibility to attacks. Furthermore, the load is placed on a single or merely a few nodes, instead of being distributed across the entire network. Finally, a user cannot interact directly with their neighbours, but has to go through the centralised node.

Since the system presented here is a prototype and the data stored is mostly textual or numeric in nature, without being excessively large, there was no need for a central database. All the information necessary about option writers, buyers, asset prices and option contracts is thus stored within the smart contracts, processed by their functions internally and read via calls or events.

4.4.5 Contract Creation

The *raison d'être* of smart contracts is providing a solution to the issue of a lack of trust in business and financial dealings [103]. They enable legal agreements to be enshrined in code in an impartial manner and to be executed automatically. There is thus a guarantee of execution according to a predefined set of instructions and a range of potential outcomes [104]. When it comes to privacy guarantees, however,

a public blockchain network has no inherent support for that. Any node can query to see all the transactions that have taken place or the entire state of a contract. In financial applications, this raises problems, especially given the emphasis on confidentiality in the sector.

A typical DApp would have one major smart contract containing information for all users. For the purposes of this system, it would mean a single smart contract deployed on the blockchain for all option writers. However, in the interest of modularity and customisation a new smart contract is created every time a new seller registers with the system, customised with the information, products and functionality pertaining to them. When it comes to the actual option contracts, generating a new and separate smart contract for each one would significantly hinder usage of the application and render its maintenance cost-prohibitive. Drawing inspiration from discussions about potential implementations of social networks/blogging sites on the Ethereum blockchain [105], the approach chosen was to have a main option factory contract for each seller, which in turns holds an array of option contract objects. To accomplish that, the option factory also defines a struct representing an option, along with declaring and storing other record-keeping information (in mappings) to facilitate search and retrieval. Contracts can hold pieces of information about multiple items [106], meaning that sellers and options could have been stored in the same contract, contrary to tables in a relational database system, which only contain information about related data in a structured format [107]. The reason sellers and options were separated was to enable each seller to have control over their own contract. The system was also developed in this manner for modularity purposes, since having one contract with all of the code would be difficult to maintain.

4.5 Design Patterns

Design patterns in programming can be likened to recipes in cooking, in that they serve as instructions for a particular way of solving a commonly found problem [108]. Throughout the design and development of the system, the use of design patterns was not preplanned or mandated in a rigid top-down way. They emerged naturally and were considered and implemented on the basis of their merits, within the specific context of the problem at hand. Design patterns can be broken down in three main categories [109]:

1. Creational
2. Behavioural
3. Structural

Creational patterns concern the methods for creating objects from classes. A very well-known one is the singleton pattern, which is meant to ensure that no more than a single instance is created. All references then point to that one instance globally. This was applied with regards to the registry smart contract (discussed in Section 5.2.5), which was intended as a single master contract coordinating and interfacing with multiple other ones, both option factories and price oracles. An example of its use is acting as an agent that collects the prices the shared oracle resources have already retrieved and then makes them available to the option factories for calculations.

Another creational pattern that was utilised is the factory one. It is a modern variation of the Factory Method pattern first described by Gamma, Helm, Johnson and Vlissides [89], [110]. The factory pattern enables the instantiation of objects without laying bare the underlying logic. This is applied in the system's option factory smart contract, which holds responsibility for creating objects representing financial options. The options can then be referred to using a shared interface, without having to delve deep into the implementation details or individual parameters of each one.

The final pattern regarding object creation that can be identified in the system is that of an object pool. An object pool helps with efficiency by enabling the reuse of objects that are expensive in terms of cost to create [111]. It thus improves performance, while each user still receives the services that they need. This paradigm is evident in the implementation within the system of a shared pool of oracles, which are then used to provide asset prices for calculations requested by all clients. As an approach, it is much preferable to the naive approach of simply creating a new oracle for each user that wants to read a price from a certain feed provided by an API.

The second class of design patterns are the behavioural ones. A version of the command pattern can be seen in the way that prices are requested from the three different oracles in the Market Rates page [112]. The transaction requests for a price update can be thought of as commands that are executed by the oracle agents, which function as invokers. The requests then reach the respective APIs, which work as a receiver executing them and returning the result. The difference is that this pattern typically involves a queue, whereas these requests can be handled simultaneously and asynchronously, given the fact that there is a separate deployed oracle for each API data source.

The final class of design patterns are the ones concerning structure. The price oracles were designed with the adapter design pattern [113] in mind because they enable the smart contracts to talk to the interfaces of the online APIs, which would not be possible directly. Additionally, the decorator structural pattern is

essentially the same as the concept of a modifier in Solidity, which enables the addition to or modification of a method's functionality, without changing the actual source code contained within its body [95].

4.6 State Machine Diagrams

An object can be broken down into the methods comprising its behaviour and the member variables indicating its state [114]. That state is an analogous concept to a smart contract's state, which encompasses the data that are stored on the blockchain as values of the member variables within that contract [115]. Transitions from one state to another take place via functions, which in turn have to be called by means of a transaction on the Ethereum platform, since they require writing to the blockchain and not just reading from it. State machine diagrams help visualise this sequence of potential states and transitions between them.

An illustrative example that is pertinent to the system presented in this report is that of an option factory. After a seller has registered successfully with the system, they create their own option factory. The factory is instantiated via its constructor and then the function *setFactoryAddress* in the Registry is called, in order to assign the newly created factory's address to the seller to whom it belongs. The factory is then considered registered with the system; buyers can view its calculated option premiums in the *Sellers Page* of the application and purchase options from it. If the seller who owns the factory wishes to draw the balance of the contract, for example to withdraw profits, they can call the *drain* function. After the execution of that function the factory's state is considered to be drained. To conclude the life-cycle of the option factory, the owner can call the *kill* function, which orders the factory to destroy itself – after transferring its balance, obviously. Examples of cases where that would be necessary are if the seller wishes to make drastic changes and needs to redeploy the smart contract anew, if a hacker has breached the contract's security and it needs to be urgently shut down as a precautionary measure or in case the seller has decided to cease writing option contracts.

The state machine diagram showing the entirety of the life-cycle of a seller's option factory is displayed in Figure 4.6 below.

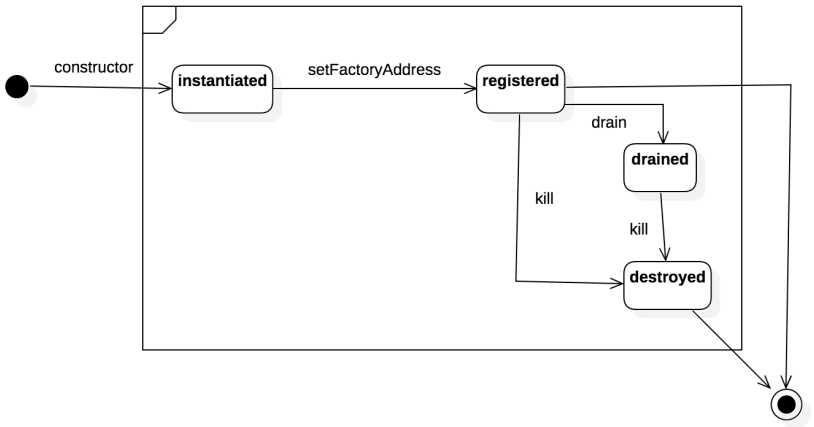

Figure 4.6: State machine diagram for an option factory

The state machine diagram demonstrating the transition of a financial option from creation to exercise is a much more straightforward one and it is shown in Figure 4.7. It is worth noting that the buyer can opt to not exercise the option, in which case the transition is straight from creation to the final absorbing state. If the execution of the financial option is expected to carry a negative settlement amount for the buyer/holder, then they would obviously opt to not exercise (as they have the right to do it, but not the obligation). The inverse does not hold true for the seller writing the option contract, who has to accept the settlement result, once the option has been created. Equivalently, the seller cannot opt to exercise the option, even if that would bring them significant financial gain to the detriment of the buyer. The modifier *onlyBuyerOfOption*, which is discussed in the Implementation part of the report, in Section 5.2.6, ensures that. This is an example of enshrining common legal terms of financial contracts in code.

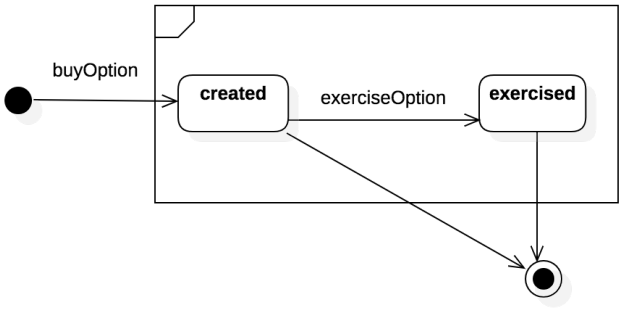

Figure 4.7: State machine diagram for an option

5 Implementation

5.1 Overview

This section discusses particulars of the implementation of the system. It analyses the challenges presented and possible solutions to them, along with their advantages and disadvantages. Finally, it describes the approach taken in each situation and the rationale behind the choices made.

5.2 Smart Contract Code

5.2.1 The Solidity Programming Language

Solidity is a high-level language for implementing smart contracts [116]. It is thus contract-oriented, rather than class- or object-oriented, which has the implication that everything resides inside the contract [117]. A contract is uniquely identified via its address in the Ethereum peer-to-peer network. Solidity's main influences were C++, Python and JavaScript. As a language, it is a statically-typed one that supports inheritance, libraries, as well as complex user-defined types. Code is written in files with the extension *.sol* and compiles to byte-code targeting the Ethereum Virtual Machine (EVM). The rationale behind selecting it for this project was mainly its current popularity and robust supporting community, which render it the default development choice. There are alternative options, such as the assembly-like Serpent [118], the LLL language that is similar to the Lisp family [119] and the inspired by Python Vyper [120]. Nevertheless, those are either no longer actively pursued or still of a rather experimental nature, so Solidity remains the *de facto* choice.

Table 5.1: Comparison of Solidity with common programming languages

Solidity	Other languages
struct	object
mapping	hash table

As can be seen from the table above, a struct is essentially the definition for an object in Solidity. Mappings act in a similar manner to hash tables, with the exception that the keys are not themselves stored, only their hashes [121]. The key type can be an unsigned integer, acting as a sort of identification, but it can also be an address corresponding to a certain user. This enables the establishment of a connection between a user and information pertaining to them that is stored on the blockchain, such as deposit balance information in this system.

5.2.2 Smart Contract Structure

The following diagram illustrates the way a smart contract is structured. It is specific to the Solidity programming language, so native to the Ethereum blockchain. The contract is the main unit containing the code and the data. Rather than a class serving as a template for the creation of objects, in Solidity structs play that role and they are defined within the contract. Structs declare the member variables of the objects that will be instantiated using their types, such as exercise price in the case of an option. The actual objects created, however, are stored in the contract, typically in arrays or mappings. As a result, data storage in Solidity typically consists of the contract containing an array of objects that are created from a struct. Additionally, the contract includes functions with their headers and bodies, as well as modifiers that can

be applied to those functions to augment their behaviour (similar to decorators in other programming languages).

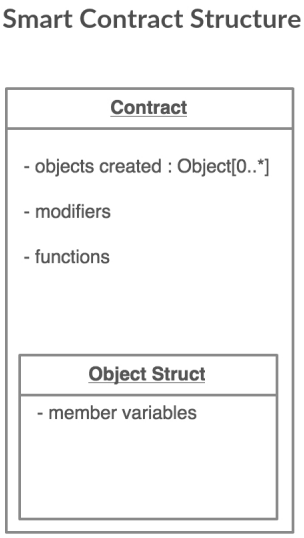

Figure 5.1: Diagram illustrating the typical structure of a smart contract written for the Ethereum blockchain network

5.2.3 Ownable Smart Contract

An important concept in smart contract development is that of the contract owner. The owner is typically the user (address) who created and deployed the contract, but ownership can subsequently be transferred to another user. A contract can also be owned by another contract, given all that is needed for reference is a valid address on the Ethereum network. Setting this up enables the use of a function modifier (*onlyContractOwner* in this case) to restrict access to certain functions of critical importance for the functioning of the system to only the owner. This essentially grants the owner a set of administrative privileges over all other users who can potentially interact with the publicly available contract.

The following snippet of code in Listing 5.2.3 demonstrates the implementation of basic user permissions in Solidity. It has been adapted from one of the prototypes [122]. A small selection of utility functions that were frequently needed have been added, since all other contracts inherit from it. Functions to convert between unsigned integers and strings have been included, since this was necessitated by the fact that the external APIs return strings as responses. On the other hand, integers representing the parts of a fixed-point number before and after the separator needed to be converted to a string representation, so that they could be transmitted to the end user. Finally, a function to parse strings to addresses was added, due to the inability to assign an address variable in Solidity directly to the hex value representing it.

```
pragma solidity ^0.4.24;

import "./safemath.sol";

/**
 * @title Ownable
 * @dev The Ownable contract has an owner address
 * and provides basic authorization control functions.
 * This simplifies the implementation of "user permissions".
 */
contract Ownable {
    // using the safe math library to prevent over/underflows
    using SafeMath for uint256;
    using SafeMath32 for uint32;
    using SafeMath16 for uint16;

    address public owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev The Ownable constructor sets the original `owner` of the contract to the sender
     * account.
     */
    constructor() public {
        owner = msg.sender;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyContractOwner() {
        require(msg.sender == owner, "Only the contract owner can call this function!");
        _;
    }

    /**
     * @dev Throws if called by any account other than the master (registry) owner.
     */
    modifier onlyMasterOwner() {
        require(msg.sender == parseAddr("0xC18AD6E102905fb84B0447077497956c407E6e79"), "Only
    the registry owner can call this function!");
        _;
    }

    /**
     * @dev Allows the current owner to transfer control of the contract to a newOwner.
     * @param newOwner The address to transfer ownership to.
     */
    function transferOwnership(address newOwner) public onlyContractOwner {
        require(newOwner != address(0), "The new owner must be different to the previous one!"
```

```
    );
51      emit OwnershipTransferred(owner, newOwner);
52      owner = newOwner;
53  }
54
55  /**
56   * Adapted from Oraclize library
57   */
58  function stringToUint(string s) public pure returns (uint) {
59      bytes memory b = bytes(s);
60      uint result = 0;
61
62      for (uint i = 0; i < b.length; i++) { // c = b[i] was not needed
63          if (b[i] >= 48 && b[i] <= 57) {
64              result = result * 10 + (uint(b[i]) - 48); // bytes and int are not compatible
    with the operator -.
65          }
66      }
67
68      return result;
69  }
70
71 /**
72   * Adapted from Oraclize library
73   */
74  function uintToString(uint value) public pure returns (string) {
75      uint v = value; // reassigning for security purposes
76      uint maxlength = 100;
77      bytes memory reversed = new bytes(maxlength);
78      uint i = 0;
79
80      while (v != 0) {
81          uint remainder = v % 10;
82          v = v / 10;
83          reversed[i++] = byte(48 + remainder);
84      }
85
86      bytes memory s = new bytes(i); // i + 1 is inefficient
87
88      for (uint j = 0; j < i; j++) {
89          s[j] = reversed[i - j - 1]; // to avoid the off-by-one error
90      }
91
92      string memory str = string(s);  // memory isn't implicitly convertible to storage
93
94      return str;
95  }
96
97  /**
98   * From Oraclize library
99   */
```

```
100     function parseAddr(string _a) internal pure returns (address) {
101         bytes memory tmp = bytes(_a);
102         uint160 iaddr = 0;
103         uint160 b1;
104         uint160 b2;
105 
106         for (uint i = 2; i < 2 + 2 * 20; i += 2) {
107             iaddr *= 256;
108             b1 = uint160(tmp[i]);
109             b2 = uint160(tmp[i + 1]);
110             if ((b1 >= 97) && (b1 <= 102)) b1 -= 87;
111             else if ((b1 >= 65) && (b1 <= 70)) b1 -= 55;
112             else if ((b1 >= 48) && (b1 <= 57)) b1 -= 48;
113             if ((b2 >= 97) && (b2 <= 102)) b2 -= 87;
114             else if ((b2 >= 65) && (b2 <= 70)) b2 -= 55;
115             else if ((b2 >= 48) && (b2 <= 57)) b2 -= 48;
116             iaddr += (b1 * 16 + b2);
117         }
118 
119         return address(iaddr);
120     }
121 
122     /** Babylonian method of calculating square root
123      * from https://ethereum.stackexchange.com/questions/2910/can-i-square-root-in-solidity
124      */
125     function sqrt(uint x) public pure returns (uint y) {
126         uint z = (x + 1) / 2;
127         y = x;
128 
129         while (z < y) {
130             y = z;
131             z = (x / z + z) / 2;
132         }
133     }
134 }
```

Listing 1: Ownable smart contract

5.2.4 Oracle Smart Contract

The oracle smart contract provides the template for each oracle retrieving price data from a different source. This is the reason why it contains state variables holding the name of the data provider and the URL of the API, along with the price to be retrieved. The URL has been split into three parts to allow interchangeability of symbols, so that the same oracle can be configured to fetch prices of various securities, rather than being confined to a single one. Obviously, the owner of the oracle contract (the administrator and also owner of the registry for the purposes of this project) has to look up the specific symbol used by the API for a certain asset from the API's documentation, as that may be a common one, but may also be a custom numeric value.

The smart contract serving as an oracle that retrieves prices inherits from two other ones: the Ownable contract described earlier and the Oraclize library contract. Oraclize provides functions to connect with web APIs in a manner secured by cryptography, so that the data received is provably the same as the data that the API provides [123]. The way the mechanism works is that the data is returned together with an authenticity proof, generated within a Trusted Execution Environment, which in this case is an auditable virtual machine hosted on Amazon Web Services. The authenticity proof is based on the TLSNotary cryptographic scheme [63], [124]. The libraries from Oraclize contain many other useful utility/helper functions. The latest implementation available [125] is targeted towards the older 0.4.18 version of Solidity, so it had to be adapted for 0.4.24, which is the one used in this project.

The main function of the contract is the one updating the price, which first checks that there is enough in the balance of the contract to cover the gas costs of the transactions required and then proceeds to call the query function asking for the price from the given URL. It also emits an event to notify the client listening that the query has been sent. Since the query is asynchronous, the response is received via the callback function. The function checks that the response is from the legitimate address and then proceeds to set the price to the result returned. Finally, an event is emitted, as notification that the price has been successfully updated.

This template contract that was created for the project allows the deployment of customised oracles that retrieve the latest price data for a specified asset, from a certain API. The code is presented below.

```
1  pragma solidity ^0.4.24;
2
3  import "./ownable.sol";
4  import "./oracle.sol";
5
6  contract PriceOracle is usingOracle, Ownable {
7
8      string public provider;
9      string public urlPart1;
10     string public symbol;
11     string public urlPart2;
12     string public price;
13
14     event ConstructorInitiated(string nextStep);
15     event NewOracleQuery(string description);
16     event PriceUpdated(bytes32 id, string price);
17
```

```
18      constructor (string _provider, string _urlPart1, string _symbol, string _urlPart2) public
        payable {
19          emit ConstructorInitiated(
20              strConcat(_provider, " oracle constructor was initiated. Call 'updatePrice()' to
        send the query to the oracle."));
21          provider = _provider;
22          urlPart1 = _urlPart1;
23          symbol = _symbol;
24          urlPart2 = _urlPart2;
25      }
26
27      function updatePrice() public payable {
28          if (oraclize_getPrice("URL") > address(this).balance) {
29              emit NewOracleQuery(strConcat(provider, " oracle query was NOT sent, please add
        some ETH for the query fee!"));
30          } else {
31              emit NewOracleQuery(strConcat(provider, " oracle query was sent, waiting for the
        response..."));
32              oraclize_query("URL", strConcat("json(", urlPart1, symbol, urlPart2));
33          }
34      }
35
36      function __callback(bytes32 id, string result) public {
37          if (msg.sender != oraclize_cbAddress()) revert("Callback from unauthorised address!");
38          price = result;
39          emit PriceUpdated(id, result);
40      }
41
42      function updateSymbol(string _symbol) public payable onlyContractOwner {
43          symbol = _symbol;
44      }
45  }
```

Listing 2: Oracle smart contract retrieving prices

5.2.5 Registry Smart Contract

The registry smart contract was conceived as the master contract recording and interacting with all others. Its source code begins by defining a seller as a struct, containing the name, account address on the network, as well as address where the option factory of said seller will be deployed and eventually reside. To function as a record keeping entity, the contract holds a mapping of addresses to the sellers they are associated with, along with a list of all addresses registered. The latter is required to facilitate discovery and prevent loss of access to any specific record. A getter function named *size* has been added, so that a client reading the data can iterate through the list of registered sellers.

Following on from the initial declarations, the main functions are the ones related to registration. The function to register a new seller checks first that they are not already registered, before proceeding to create a new seller object and store all the necessary information. Since it is a function that stores data, it changes the state of the contract and thus has to be declared *payable* to accept the ether sent to it. It is also declared *external*, given that there is no conceivable use case for a user registration function to be called internally. Finally, the registration function emits an event as notification that it has completed successfully and also returns the boolean value *true*. An event is necessary here because the function is called via transactions, not just calls to view the current state of the contract or pure actions that do not alter it. Transactions, however, return a hash, which is why the emission of an event is necessary to signify to the end user that the process has been brought to a successful close. The additional use of the boolean return value is helpful in terms of extensibility, meaning that another contract in the future could call the registration function externally and it would be able to receive the return value, but not the event. This approach lets the Solidity developer cover both bases. A utility function checking if a certain account has been registered is added in the contract. It could be declared *internal*, since it is used primarily in the registration function, but it is possible that it might be useful for a client, so it is *public* instead. Furthermore, it is declared as a *view* function, given it does not make any changes to storage, but merely reads the state. The first part of the registry's functionality concludes with a setter method, which allows a specific seller who has already registered to declare their factory address at a later time, for modularity and maintainability purposes.

The second part of the registry concerns the functionality to retrieve prices from oracles. As can be seen at the top of the code listed below, the price oracle smart contract is imported, so that the registry is able to create oracle objects and knows their Application Binary Interface (ABI) to interact with their functions. There are methods to retrieve the price from a single oracle at a provided address or to retrieve prices from three different oracles and calculate their weighted average, according to a set of weights representing the relative reputation of each oracle in the network. The prices are stored respectively and events are emitted, so they can be listened for and received by the client libraries. Even an operation as simple as calculating a weighted average on chain posed a significant challenge during development, given the fact that Solidity does not support decimals or fixed point arithmetic yet [126]. Prices have to be stored as strings, to be able to display the floating point and the decimal part, but they have to be split and converted to unsigned integers to perform the necessary calculations. Another approach would be to multiply with a fixed multiplier, for example 10^6, and store the location of the floating point. Each API returned price information with a different degree of precision, though, meaning that of the three prices received, the first one was rounded to the hundredths, the second had a decimal part comprised of eight digits and the last one of nine. These functions to retrieve and calculate prices from oracles conclude the internals of the

registry. The code for the registry smart contract is displayed in its entirety below.

```
pragma solidity ^0.4.24;

import "./ownable.sol";
import "./price_oracle.sol";

/**
 * @title Registry
 * @dev The registry for sellers (master contract)
 */
contract Registry is Ownable, usingOracle {

    struct Seller {
        string name;
        address account;
        address factory;
    }

    mapping (address => Seller) public sellers;
  // lookup table as pattern for key storage to enable total discoverability of data
  // inspired by:
  // https://ethereum.stackexchange.com/questions/15337/can-we-get-all-elements-stored-in-a-mapping-in-the-contract
    address[] public addressLUT;

    /**
   * Public getter function that returns size of LUT containing addresses
   */
    function size() public view returns (uint) {
        return addressLUT.length;
    }

    event Registered(string name, address indexed account);

  /**
   * Registration function
   */
    function register(string _name) external payable returns (bool success) {
    // require that the name does not already exist
        require(!isRegistered(msg.sender), "This account is already registered!");
        addressLUT.push(msg.sender);
        sellers[msg.sender] = Seller(_name, msg.sender, 0);
        emit Registered(_name, msg.sender);
        return true;
    }

    function isRegistered(address _account) public view returns (bool) {
        return sellers[_account].account != 0;
    }

    function setFactoryAddress(address _factoryAddress) public {
```

```
50          sellers[msg.sender].factory = _factoryAddress;
51      }
52
53      string public price;
54      string public averagePrice;
55
56      event OraclePrice(string price);
57      event AverageOraclePrice(string price);
58
59      /**
60     * Getting price from a given oracle at a specified address
61     */
62      function getPrice(address _oracleAddress) public payable returns (string) {
63          PriceOracle oracleContract = PriceOracle(_oracleAddress);
64          price = oracleContract.price();
65          emit OraclePrice(price);
66          return price;
67      }
68
69    /**
70     * Getting prices from all three oracle addresses
71     * Solidity does not support fixed point numbers yet,
72     * hence the extra calculations
73     */
74      function getAveragePrice(
75          address _oracleAddress1, address _oracleAddress2,
76          address _oracleAddress3) public payable returns (string) {
77
78          PriceOracle oracleContract1 = PriceOracle(_oracleAddress1);
79          PriceOracle oracleContract2 = PriceOracle(_oracleAddress2);
80          PriceOracle oracleContract3 = PriceOracle(_oracleAddress3);
81          uint price1 = stringToUint(oracleContract1.price());
82          uint price1_decimal = (price1 % 100000000) / 1000000;
83          price1 /= 100000000;
84          uint price2 = stringToUint(oracleContract2.price());
85          uint price2_decimal = (price2 % 1000000000 ) / 10000000;
86          price2 /= 1000000000;
87          uint price3 = stringToUint(oracleContract3.price());
88          uint price3_decimal = price3 % 100;
89          price3 /= 100;
90          averagePrice = strConcat(
91              uintToString((price1 * 4 + price2 * 3 + price3 * 3) / 10),
92              ".",
93              uintToString((price1_decimal * 4 + price2_decimal * 3 + price3_decimal * 3) / 10))
    ;
94          emit AverageOraclePrice(averagePrice);
95          return averagePrice;
96      }
97  }
```

Listing 3: Registry smart contract

5.2.6 Option Factory Smart Contract

The option factory smart contract is the one responsible for creating options, when users buy them. It also provides functionality to calculate the corresponding premiums, based on given parameters, as well as storing all information required and settling the balances upon execution. Additionally, helper functions to deposit and withdraw ether, check the balance of a buyer with a particular seller and retrieve information about options they have bought are included in the contract.

There is initial provision for customisation of the option factory template. For the purposes of this project, a simplified markup percentage rate is supplied by the seller and stored upon creation of the contract. The rate is the extra charge on top of the premium calculated for a certain option, which is meant to vary by seller and allows the comparison of prices between them.

An option struct is declared that contains essential information about the option, such as the underlying asset, exercise price, expiration date (as a Unix timestamp, so number of seconds since the beginning of the current epoch [127]) and a boolean value denoting whether the option has been exercised yet or not. That allows us to store an array of option objects (as *public*, so that a getter function is automatically created and users can read from it). Each option is identified by its position in the array, a unique unsigned integer. That integer is also stored in a mapping from integers to Ethereum network addresses, so that when a new option is created, its identification number is mapped to the address of the user who bought it. This is how options are linked to their respective buyers. For ease of record-keeping, there is an additional mapping storing a count of the options that each user has bought. That serves to alleviate the need to loop through the entire array of options and referring to the mapping every single time, to check if that particular option is linked to the buyer in question. Finally, a balance mapping is declared in storage, for the purposes of keeping a record of buyers' deposit balances with sellers, enabling the purchase of options without having to sent ether for each purchase, as well as easily settling balances. This provides an additional avenue of functionality for the contract; one that carries lower gas fees for frequent purchasers.

The primary function of the contract is the one to buy an option. It is declared *external* and *payable*, since it is meant to only be called by customers looking to purchase an option and it has to be able to receive amounts equivalent to the premium. The inputs of the function are the characteristics of the option to be bought. Inside the function, an internal call is made to the method that calculates the option premium and this was designed to prevent abuse of the system, which could conceivably occur if the premium was passed on from the outside. The current exchange rate for ether to USD is also retrieved to be used in calculations. After the necessary conversions, a check is made to ensure that the buyer sent enough to cover the premium. Following that, a new option object is instantiated using the supplied information and is assigned to the sender of the message, the option buyer. Finally, the option count of the buyer is incremented and an event is emitted as notification of the option's successful purchase. There is an additional function provided, which is identical, except for the fact that the buyer does not send any ether, but uses their deposit balance instead. As a result, this function is declared *non-payable*, which is the default setting and can thus be omitted from the header. A check is performed respectively in the body of the function to ensure that the balance is sufficient and the user is notified of their remaining balance in the end.

The second most important functionality of the option factory is the one relating to the calculation of the premium to be paid for an option. The function retrieves the latest price from the registry and calculates the

difference to the exercise price requested to arrive at the intrinsic value of the option. The time value was not accounted for because of the fact that Solidity does not even support arithmetic with decimals yet [126]. To find the square root of a number, for example, there is no function in Solidity that can do that presently, so one would have to use methods like the Babylonian root-finding algorithm to try to approximate an answer or a digit-by-digit method, in cases where precision is of the essence. This highlights that it is not yet realistically possible to run (complex) calculations on-chain, which renders formulas like the Black-Scholes option pricing model unusable, especially given their need for probability densities. One solution would be to run the calculations on a server and deliver the prices from a central location, however that goes against the experimental purposes of this project, in terms of determining the exact extent to which functionality can be placed on the blockchain. As such, the intrinsic value was deemed to suffice, along with the application of a markup to differentiate the prices provided by sellers. The option premium is passed on both as a return value (together with the spot price) and through an event. The event uses the keyword *indexed* for the buyer address, so emissions can be filtered by the libraries in the front-end for the current user only.

The option-related functionality includes a function that option buyers can call to exercise one of the options in their portfolio, along with the corresponding event to be emitted. The function could be declared *external*, however it was declared *public* in the interest of possible future extensions, where it could be called internally from another function, perhaps a periodically-scheduled one (when and if Solidity and Ethereum permit that). Since the *exerciseOption* function is meant to be called by the buyer, within the scope of this system, the modifier *onlyBuyerOfOption* is attached. Inside the body, the code checks the current block's timestamp to compare it with the option's stored expiration date (line 83 in Listing 5.2.6 below). If the expiration date has not yet passed, then execution of the rest of the code is skipped and a boolean value of *false* is returned. If, on the other hand, the date has passed and the option has not already been exercised (line 84), then the latest (average) spot price is retrieved from the registry and the settlement amount is calculated, as the difference of the spot price from the exercise price. That amount is subsequently converted from USD back to wei (a subdivision of Ethereum [128]), since wei is the internal unit of measurement that Solidity smart contracts use. Finally, the balance of the buyer is adjusted and the *OptionExercise* event is emitted, which inserts log entries in the transaction receipt containing the settlement amount and the new account balance, along with their indexed address that can be filtered by.

Helper functions returning a count of the options a buyer has purchased and their identification numbers (as an array of unsigned integers) are also provided. These functions are declared *external* because they are meant to be called from the outside and utilise the *message sender* as the buyer address for privacy purposes, meaning they will only return the information pertaining to the specific user who called them. They are also designated as *view* functions, meaning that they merely read from the smart contract's state and do not change it in any way. This is also the reason why the *result* variable of the *getOptionsByBuyer* function is declared as *memory*, indicating to the compiler that it does not have to preallocate space for it in the virtual structure making up the contract's storage, which is immutable at runtime [129].

The rest of the option factory contract consists of utility functions. The registry's address is fixed and stored in a state variable, so that it can be referred to when the factory needs to retrieve the latest price for the underlying asset. It is worth noting that solidity does not provide for parsing strings to addresses, which is why a custom *parseAddr* function from the Oraclize library is used. Despite the registry being predetermined, under extreme circumstances there might be a need for it to change drastically, meaning

that a new contract has to be redeployed. In the interest of being able to update the reference in such a case, a *setRegistryAddress* function has been included. That function is only callable externally by the master registry's owner, to prevent unauthorised modification. It is also declared *payable* because it alters the state.

Following the registry-related methods are the ones providing account/wallet functionality for the customers of the seller who owns the option factory. A minimum deposit amount is set, which could be increased to allow only investors with at least a certain amount of assets to interact with the contract, for instance. The complementary modifier and setter function are subsequently defined. The *require* functions include a message to be logged, if the check fails, which is a relatively new feature of Solidity [130]. Additionally, *deposit* and *withdraw* functions are declared, which are part of a seller's customer accounts offering. The *deposit* function is declared *external* and *payable*, for obvious reasons. The function allowing the customer to withdraw ether from their balance with the seller performs the required checks, prior to doing so. It then emits a *Withdrawal* event to notify the customer of the success of their request. The deposit method, on the other hand, does not need an event because the customer can simply refresh their balance to view the updated one (even though one could make an argument for adding a *Deposit* event for the sake of consistency). Finally, there is a *view* method that reads the message sender's balance from the *buyerBalance* mapping and returns it as an integer. It has merit mentioning that there is no need for an event here to return a value to the caller, since the function does not alter the state and thus does not need to be called via a transaction. Due to the fact that it is read-only, a simple return statement is enough. Another detail that should be paid attention to is that *buyerBalance* is declared as *public*. That would allow any called to view the balance of another, as long as they knew the address to look for, since Solidity automatically creates a getter function for each member variable of a contract that is declared *public* [131], [132]. Mappings do not permit iterative search, however retrieval based on the hash of the key value is easily doable. By restricting access to the mapping and wrapping it in a function that only returns the balance of the current *sender*, privacy is greatly enhanced.

To conclude the analysis of this contract's code, a *drain* and a *kill* function are included. The *drain* function can only be called externally by the factory's owner (the seller) and allows them to withdraw the balance of the contract itself. It also emits an event to notify them of the exact amount withdrawn, as it creates a transaction to transfer the balance, and returns the boolean value *true*. The *kill* function, on the other hand, provides a way to destroy the contract and stop the issuance of any new options. This can only be called by the seller who owns the contract, for obvious reasons. It is meant to be a so-called *nuclear option*, only to be used in extreme cases, for example as protection against a hack.

The source code for the option factory smart contract follows in Listing 5.2.6, on the next page.

```
pragma solidity ^0.4.24;

import "./ownable.sol";
import "./registry.sol"; // importing so the contract knows the ABI to interact with

contract OptionFactory is Ownable {

    uint markupPercentage;

    function setMarkupPercentage(uint _percentage) external onlyContractOwner {
        markupPercentage = _percentage;
    }

    /**
     * Constructor setting the markup percentage
     */
    constructor (uint _percentage) public {
        markupPercentage = _percentage;
    }

    modifier onlyBuyerOfOption(uint _id) {
        require(msg.sender == optionToBuyer[_id], "Only the option buyer can execute this function!");
        _;
    }

    struct Option {
        string asset;
        uint exercisePrice;
        uint expirationDate;
        bool exercised;
    }

    Option[] public options;

    mapping (uint => address) optionToBuyer;
    mapping (address => uint) buyerOptionCount;
    mapping (address => uint) buyerBalance;

    event OptionPremium(address indexed buyer, uint premium);

    /**
     * @dev Calculating intrinsic value, but not time value
     * because of no support for floating arithmetic
     * and probability distributions in Solidity
     * (yet)
     */
    function calculateOptionPremium(uint _exercisePrice) public payable returns(uint, uint) {
        Registry registry = Registry(registryAddress);
        uint spotPrice = stringToUint(registry.averagePrice()) / 100;
        uint intrinsicValue = _exercisePrice - spotPrice;
```

```
51          uint premium = intrinsicValue + ((intrinsicValue * markupPercentage) / 100);
52          emit OptionPremium(msg.sender, premium);
53          return (premium, spotPrice);
54      }
55
56      event NewOption(address indexed _buyer, uint indexed _id, uint _balanceLeft);
57
58      function buyOption(string _asset, uint _exercisePrice, uint _expirationDate) external
    payable {
59          // calculating and retrieving the option premium
60          (uint premium, uint ethToUSD) = calculateOptionPremium(_exercisePrice);
61          // converting the premium from USD to eth and then wei
62          // checking to see that the user sent enough to buy the option
63          require(msg.value >= ((premium * 1000000000000000000) / ethToUSD), "Not enough ether
    sent to buy the option!");
64          uint id = options.push(Option(_asset, _exercisePrice, _expirationDate, false)) - 1;
65          optionToBuyer[id] = msg.sender;
66          buyerOptionCount[msg.sender] = buyerOptionCount[msg.sender].add(1);
67          emit NewOption(msg.sender, id, buyerBalance[msg.sender]);
68      }
69
70      function buyOptionWithBalance(string _asset, uint _exercisePrice, uint _expirationDate)
    external {
71          (uint premium, uint ethToUSD) = calculateOptionPremium(_exercisePrice);
72          require(buyerBalance[msg.sender] >= ((premium * 1000000000000000000) / ethToUSD), "Not
     enough balance!");
73          buyerBalance[msg.sender] -= (premium * 1000000000000000000) / ethToUSD;
74          uint id = options.push(Option(_asset, _exercisePrice, _expirationDate, false)) - 1;
75          optionToBuyer[id] = msg.sender;
76          buyerOptionCount[msg.sender] = buyerOptionCount[msg.sender].add(1);
77          emit NewOption(msg.sender, id, buyerBalance[msg.sender]);
78      }
79
80      event OptionExercise(address indexed _buyer, uint _settlementAmount, uint _balanceLeft);
81
82      function exerciseOption(uint _id) public onlyBuyerOfOption(_id) returns(bool) {
83          if (now >= options[_id].expirationDate) {
84              if (!options[_id].exercised) {
85                  options[_id].exercised = true;
86                  Registry registry = Registry(registryAddress);
87                  uint spotPrice = stringToUint(registry.averagePrice()) / 100;
88                  uint settlementAmount = options[_id].exercisePrice - spotPrice;
89                  // converting settlement amount from USD back to wei to add to buyer balance
90                  buyerBalance[msg.sender] += (settlementAmount * 1000000000000000000) /
    spotPrice;
91                  emit OptionExercise(msg.sender, settlementAmount, buyerBalance[msg.sender]);
92              }
93              return true;
94          } else {
95              return false;
96          }
```

```
97      }
98
99      function getOptionCount() external view returns(uint) {
100         uint result = buyerOptionCount[msg.sender];
101         return result;
102     }
103
104     function getOptionsByBuyer() external view returns(uint[]) {
105         uint[] memory optionIDs = new uint[](buyerOptionCount[msg.sender]);
106         uint pointer = 0;
107         for (uint i = 0; i < options.length; i++) {
108             if (optionToBuyer[i] == msg.sender) {
109                 optionIDs[pointer] = i;
110                 pointer++;
111             }
112         }
113
114         return optionIDs;
115     }
116
117     // one fixed master registry
118     address registryAddress = parseAddr("0xd7303fafe84917a550834ea01e43db473a5e71c9");
119
120     /**
121      * @dev Only allowing the master administrator to change the registry's address,
122      * for security purposes.
123      */
124     function setRegistryAddress(address _newAddress) external payable onlyMasterOwner {
125         registryAddress = _newAddress;
126     }
127
128     uint minimumDepositAmount = 0.01 ether;
129
130     modifier aboveMinimumDepositAmount() {
131         require(msg.value >= minimumDepositAmount, "The amount sent is below the minimum
    threshold!");
132         _;
133     }
134
135     function setMinimumDepositAmount(uint _amount) external onlyContractOwner {
136         minimumDepositAmount = _amount;
137     }
138
139     function deposit() external payable aboveMinimumDepositAmount {
140         buyerBalance[msg.sender] += msg.value;
141     }
142
143     event Withdrawal(address indexed buyer, uint amount, uint balanceLeft);
144
145     function withdraw(uint _amount) external {
146         require(buyerBalance[msg.sender] >= _amount, "Not enough balance!");
```

```
147         buyerBalance[msg.sender] -= _amount;
148         msg.sender.transfer(_amount);
149         emit Withdrawal(msg.sender, _amount, buyerBalance[msg.sender]);
150     }
151 
152     function getBalance() external view returns(uint) {
153         return buyerBalance[msg.sender];
154     }
155 
156     event Drained(uint balance);
157 
158     function drain() external onlyContractOwner returns (bool) {
159         emit Drained(address(this).balance);
160         msg.sender.transfer(address(this).balance);
161         return true;
162     }
163 
164     function kill() public onlyContractOwner {
165         selfdestruct(msg.sender);
166     }
167 }
```

Listing 4: Option factory smart contract

5.3 Private Blockchain Experiment

The issues with regards to the low transaction throughput and high sync times of the public Ethereum blockchain network prompted the author to investigate how the system would perform, when it used a private blockchain instead. To accomplish that, the tool Ganache [133] was used to create a local blockchain and simulate a group of nodes running the same Ethereum consensus protocol as the public network. Ten nodes were created, each with 100 ether as a starting account balance, since those were the default values. The Oraclize Ethereum bridge [134] was configured and deployed to the private network, to make the oracles work. Slight changes in the source code of the project also had to be implemented, in order to make it compatible with the private blockchain.

The first change that had to be made was that the web3 provider used had to be set to the one running on localhost, as shown below. MetaMask was thus no longer needed (even though it could be configured to work with local accounts, too).

```
this.web3 = new Web3(new Web3.providers.HttpProvider('http://127.0.0.1:8545'));
```

Listing 5: Provider on localhost

Another issue was that the transaction message sender was now not the current account of the logged-in user interacting with the application by default, but rather the first one created in the test blockchain. One solution implemented was that the code involving the *msg.sender* property (for instance, the sender address being used as a key to a mapping like *buyerBalance*) had to be replaced with an address being passed to the smart contract's function as a parameter. Another one was that the *from* property had to be set in the front-end, to specify the message sender explicitly. In a few cases, the headers of functions in the contract service of the front-end had to be altered to incorporate an extra parameter, the account from which to sign and send the transaction. An example snippet of code that had to be added in a few places is shown in the following listing.

```
from: this.web3.eth.accounts[account],
```

Listing 6: Specifying the from field explicitly

A more challenging issue was that the *memory* keyword indicating that a variable should just be stored in memory and not in storage did not work in the virtual machine where the private blockchain network was running, so the code to retrieve the options indexes for a specific buyer (in the *getOptionsByBuyer* function) in the option factories did not work anymore. After experimenting with different potential solutions that proved unsatisfactory, such as declaring the variable as a *storage* one or concatenating indexes discovered in a string and logging an event with that string, a fix was found. The command to emit a *BuyerOptions* event was placed within the loop searching for options bought by a specific account, so each time an option was discovered an event was emitted containing the option's information. The following code was then added to the *Transactions Page* of the front-end application, to listen for those events and read the information emitted about each option found.

```
let optionIDsEvent = this.contractService.optionFactories[
  __this.contractService.selectedOptionFactoryId].BuyerOptions(function(error, eventInfo) {
  if (error) {
        return;
    }
    console.log(eventInfo.args._optionID.c[0]);
    // retrieving a specific option via its id
```

```
    __this.contractService.getOption(__this.contractService.selectedOptionFactoryId,
        eventInfo.args._optionID.c[0]).then(optionInfo => {
            const option: Option = {
              id: eventInfo.args._optionID.c[0],
              asset: optionInfo[0],
              exercisePrice: optionInfo[1].c[0],
              expirationDate: new Date(optionInfo[2].c[0] * 1000),
              exercised: optionInfo[3]
            };

      __this.options.push(option);
    });
  }
);
```

Listing 7: New method of retrieving options bought by a specific buyer

Furthermore, the old way of calling the *getOptionCount view* method from the web application did not work, so it was adapted to be able to read from the local blockchain. The modified code is presented below.

```
async getOptionCount(id: number): Promise<number> {
  return new Promise((resolve, reject) => {
    this.optionFactories[id].getOptionCount.call(this.account, function (
      error,
      result
    ) {
      if (error) {
        alert(error);
        return;
      } else {
        resolve(result);
      }
    });
  }) as Promise<number>;
}
```

Listing 8: Modified getOptionCount method

After those changes, the registry and the oracle smart contracts were deployed as usual and the web application was served to the browser.

For purposes of comparison, an experiment was conducted to investigate the speed difference between transaction confirmations in the public network versus those in the private one running locally. The time measured was from the start of the transaction initiation to the moment the output of the function being called is received successfully (for example, via an event). The function executions were timed using the *console.time()* method [135] and results are presented in Table 5.2. As a side note, the number of milliseconds each action took has been rounded to the second decimal point.

Table 5.2: Comparison of transaction confirmation times between the public and private network

Action	Public Network	Private Network
Registering	58053.98ms	71.55ms
Deploying factory	13039.27ms	635.02ms
Retrieving prices from all three deployed oracles	84884.75ms	13782.01ms
Calculating weighted average price	23242.25ms	118.74ms
Getting option premiums from two sellers	31098.15ms	584.80ms
Buying option	23194.82ms	88.70ms
Retrieving options bought in transactions page	327.97ms	171.59ms
Exercising option	22364.79ms	72.98ms

As is evident from the results of the experiment above, the private network enables transaction speeds that are orders of magnitude faster than the public one, even though they are both running the same consensus protocol. The retrieval of prices from the three different data feeds via the oracles takes the longest in both cases, which makes sense intuitively, given the number of transactions required for that whole sequence to conclude. Furthermore, it is worth observing that there is no need to pay fees in the private network, since there are no public nodes that have to be incentivised to process the transactions, and also that transactions get signed automatically. Based on all of the above, a permissioned blockchain would be the natural choice for enterprise purposes, where speed and efficiency are of the essence. Further improvements could be made, for example with regards to the consensus mechanism employed. Given that all nodes in a private network have been authenticated and public verifiability and auditability are not an issue, there is less of a need to run a computationally expensive scheme such as proof-of-work to artificially slow down the network and protect it from double-spending and other attacks [23]. Network design could even go as far as designating a single validator node, representing a central counterparty such as a stock exchange or an option clearing house, which would be responsible for processing and validating all transactions. The speed of the network would then be dependent on that node's capacity.

6 Testing

6.1 Overview

A piece of software is expected to produce a certain output for a given input. Evaluating whether it indeed produces the correct output is known as software testing [136]. Testing as a process allows developers to judge the quality of the software. It can be split into verification and validation. Verification evaluates whether the software behaves in the manner intended by its design, whereas validation is concerned with whether the software satisfies the requirements of the end users [136]. Given the nature of the project, a security audit of the smart contracts was also conducted and is detailed in Section 6.3.

6.2 Verification

Software verification is the undertaking of ascertaining whether a system works without bugs [137]. Copeland describes testing as the process of comparing outputs with what they should be, given certain inputs and under specific conditions, which are encoded as test cases [138]. He then proceeds to identify the four classical levels of testing: unit, integration, system and acceptance. Acceptance testing concerns the end users of the system, so it was not applicable given the scope of this project, as the system created was more of an experimental proof of concept.

6.2.1 Unit Testing

Unit testing involves testing software at a low level, down to the individual pieces of code like classes or functions [138]. The Angular framework used for the front-end supports unit testing by incorporating the well-known Jasmine test framework for JavaScript [139] within Angular's Karma testing environment [140]. The Angular command line interface sets them both up, runs the tests and displays the output in the console. To write unit tests for each component, a file with the same name as the component, but ending in *.spec.ts* was created, which enables it to get picked up by the Karma test runner [141]. The initial suite of tests, as can be seen in Appendix C, merely picked up Angular component and provider declaration issues within the tests themselves. To dive deeper, the Angular team recommends looking at services, which can be tested directly just with Jasmine [141]. They provide examples for methods that should return real values, values from observables and from promises. The *contract service* is the service that contains most of the web3-related functionality of the project, so that one was focused on the most. It makes heavy use of promises, given the asynchronous nature of transactions in the Ethereum network [142]. An example of a unit test written for the *register* function can be seen in the listing below.

```
1 it('#register should return value from a promise', (done: DoneFn) => {
2   service = TestBed.get(ContractsService);
3   expect(service.register('name')).toBe('promise value');
4   done();
5 });
```

Listing 9: Unit test for register function returning a Promise

The problem that was encountered was that the test environment did not recognise the MetaMask plug-in. Another issue was that calls to a node in the blockchain network are asynchronous, so TypeScript requires them to return a promise-like object [143], rather than a promise value. Furthermore, the responses returned are external to the web application, so they should not be tested as part of it [144]. The Angular documentation suggests that the preferred solution to overcome these issues is to create a *spy* on the service method being tested, to mock the injected dependency [141]. The code for this, as implemented for

the *register* method in the project's *ContractsService* Angular service can be found in the following listing. The *describe* keyword indicates a new test suite and the *it* keyword a test case [145].

```
describe('ContractsService without Angular testing support', () => {
  it('#register should return transaction hash from a spy', () => {
    // create 'register' spy on an object representing the ContractsService
    const contractServiceSpy =
      jasmine.createSpyObj('ContractsService', ['register']);

    // set the value to return when the 'register' spy is called.
    const transactionHash = '0
    x4f05b82a13a47851f4062d634128b4f798a1d7c1e32a3485a3190884d60130ca';

    contractServiceSpy.register.and.returnValue(transactionHash);

    expect(contractServiceSpy.register(4, 'name'))
      .toBe(transactionHash, 'service returned hash value');
    expect(contractServiceSpy.register.calls.count())
      .toBe(1, 'spy method was called once');
    expect(contractServiceSpy.register.calls.mostRecent().returnValue)
      .toBe(transactionHash);
  });
});
```

Listing 10: Jasmine unit test for register function using a spy

Similar unit tests were written for the rest of the functions. The output can be found in Appendix C.

6.2.2 Integration Testing

According to Copeland [138], integration testing is the phase of the process where units are combined to form a whole system. The motivation he provides behind it is that units of code might perform properly when they are isolated, but display erroneous behaviour when integrated. The library used for integration testing was Protractor [146], since it is compatible with Angular. An example of testing how the injected contract service functions end-to-end is shown in the code listing below.

```
it('Should retrieve account', done => {
  inject([ContractsService], (service) => {
    service.getAccount(0).then(account => {
      console.log('account:', account); // Print the account if a response was received
      done(); // informs runner that the asynchronous code has finished
    });
  });
});
```

Listing 11: Example integration test using Protractor

6.3 Security Audit

Smart contract development necessitates a novel approach of thinking, which feeds into the testing process, too. Besides encouraging good user behaviour and healthy network effects via economic incentives, the prospect of adversaries attacking the contract also needs to be taken seriously into consideration [147]. Furthermore, the governance mechanisms employed present another challenge in terms of security. An owner/central authority can ensure that the system recovers from bugs or hacks, for example via reversal of transactions or activation of a fork. However, this reintroduces a single point of failure, upon which the entirety of the system is dependent.

Consequently, security auditing of smart contracts is a critical part of the code review process and should be accompanied by formal verification. Events such as the DAO hack [148], which took place even with the smart contract having passed a professional security audit, highlight the risks. Security is particularly pertinent when it comes to financial applications, due to the stakes in question.

A basic methodology has been suggested as a blueprint for the security audit process of smart contracts [149]. The first step is to interview the developers, in order to understand their thought process with regards to the design of the contracts. This was supplemented by self-reflection for the purposes of this project, since the author serves as both developer and auditor. The next step in the process is to set up an environment to test the execution of the contracts. It goes without saying that contracts should first be reviewed on test blockchains, such as the various test networks that the Ethereum platform provides [20]. This has the added benefit of permitting both the developer and the auditor to save on costs during the development and testing phases, which might otherwise prove prohibitively expensive. If the chain is part of a private or permissioned environment, then access to that network would have to be granted.

Following that, the source code contained in the *.sol* files should obviously be reviewed, together with the comments within it, so that potential threats are modelled in the context of the application under examination. What is important to look out for and break down is the flow of the code. A graph of the control flow can optionally be created using tools such as the solgraph parser [150] or the parser by Consensys [151]. The main step of the recommended analysis consists of analysing the smart contract code using a tool called Oyente, which will be discussed in Section 6.3.1. In addition to Oyente checking for certain known vulnerabilities, they should also be verified manually and additional ones should also be tested, for thoroughness. Examples of such vulnerabilities are the call stack attack (that has been patched), the concurrency bug, attacks based on time dependency (race conditions) and recursive flaws. A final measure that could be taken would be to attempt to decompile the EVM byte-code using a tool like Porosity [152]; the source code is readily available in this case, though, so this is not deemed necessary.

6.3.1 Symbolic Execution Testing

Symbolic execution is a technique that lies between program testing and proving [153]. It replaces sample inputs with symbols representing sets of input classes, thus allowing each test result to cover a range of test cases. Oyente [154] is a tool using symbolic execution to analyse smart contracts for known security vulnerabilities, either straight from their source code or even contracts that are already deployed and running on the blockchain. It was developed as a research project by a team from the National University of Singapore and its first version accompanied their published paper [155]. It checks for a range of potential hacks, including over/underflows, the Parity [156] multisig wallet bug that led to a loss of $155 million [157] and the re-entrancy vulnerability [158] that took down the first DAO [159]. The tool is now maintained by

the team from Melonport [160], however it still lags behind the most recent version of Solidity, as it only works with *0.4.19*. As a result, features (both minor syntactic and more substantial ones) that are used in the utility functions of the smart contracts of this project are not supported by the latest compatible compiler, so a few of them had to be removed and the code slightly altered to allow its analysis by the tool. Furthermore, the dynamically retrieved oracle prices within the bodies of the functions of the registry smart contract were not supported using the tool, as was the retrieval of the average price from the registry within the option factory contract. The analysis of contract-to-contract interaction that can be accomplished using this tool is thus limited. Consequently, those retrieved values were altered to fixed ones.

The analysis performed uncovered mostly integer under/overflow issues, which arise primarily out of Solidity using just unsigned integers. It also unearthed a transaction-order dependency issue. A closer look at each of the issues discovered follows here, whereas the full log can be found in Appendix B.

In the option factory smart contract, an integer underflow was possible in a few places. In the *exerciseOption* function, the *spotPrice* is deducted from the *exercisePrice*, to calculate the settlement amount. Since the calculation concerns call option holders having the right to exercise, they would only do so in the case where the exercise price of the option they had bought is higher than the spot price at expiration. As a fix to the issue, the following check was added to ensure that the subtraction only takes place under the correct circumstances:

```
if (options[_id].exercisePrice > spotPrice)
```

Listing 12: Fix in the source code of the exerciseOption function in the option factory smart contract

Another integer underflow issue was in the *calculateOptionPremium* function.

```
uint intrinsicValue = _exercisePrice - spotPrice
```

Listing 13: Problematic line in calculateOptionPremium function

The first attempted solution was to check for the validity of the *_exercisePrice* argument, but that did not remove the warning. As a side note, a declaration of simply *uint* in Solidity is equivalent to *uint256* [161], so the maximum value for a variable of that type is $2^{256} - 1$.

```
require(_exercisePrice >= 0 && _exercisePrice <= 2**256 - 1);
```

Listing 14: First attempted solution

The next attempt consisted of comparing the two parts of the subtraction equation, the subtrahend and the minuend, prior to actually going ahead and performing the arithmetic operation. The *intrinsicValue* variable was declared and initialised prior to the *if-else* statements to prevent compiler warnings. This implementation removed the error about the integer underflow vulnerability. It essentially could just be encapsulated in an absolute value utility function that is called whenever a similar subtraction has to take place.

```
uint intrinsicValue = 0;
if(_exercisePrice >= spotPrice) {
    intrinsicValue = _exercisePrice - spotPrice;
} else {
    intrinsicValue = spotPrice - _exercisePrice;
}
```

Listing 15: Second attempted solution that worked

However, these subtraction underflows should have been prevented by the use of the *SafeMath* library, as indicated in the following piece of code in the *Ownable* smart contract, from which all others inherit:

```
// using the Safe Math library to prevent over/underflows
using SafeMath for uint256;
using SafeMath32 for uint32;
using SafeMath16 for uint16;
```

Listing 16: Importing the SafeMath library

After some investigation, it emerged that it is not simply enough to import and declare the use of the libraries, but the developer also has to substitute their functions (*add, sub, div, mul*) for the standard arithmetic operators, in each situation where a potential vulnerability issue could arise [162]. The solution then becomes the following:

```
intrinsicValue = exercisePrice.sub(spotPrice);
...
settlementAmount = options[_id].exercisePrice.sub(spotPrice);
```

Listing 17: Using the functions of the SafeMath library

In addition to those mathematical operations, the tool gave underflow warnings for state variables, such as the array storing the options and the mapping storing the sellers. This indicates that the public getter functions automatically created by Solidity for those variables are also vulnerable to integer underflows, due to the indexes used. The following is an example of code that triggered warnings:

```
options[_id]
```

Listing 18: Integer underflow warnings for public list-based variables

A solution would be to declare those variables as *private* and explicitly create getter functions for them, so that access can be controlled (for example, via modifiers, *require* statements or assertions). There are issues with that approach, for example not being able to return a struct. However, workarounds can be found, such as returning a tuple to remedy the aforementioned problem [163].

Strings also threw warnings for potential integer overflow. The reason is that they are not a real value type in Solidity, but rather dynamically-sized UTF-8-encoded arrays of bytes [164]. It is thus problematic to prevent overflows, especially when the strings come from outside the contract's environment, for instance as results returned to the oracles from API callbacks.

```
public provider
public urlPart1
...
function __callback(bytes32 id, string result) public {
^
Spanning multiple lines.
Integer Overflow occurs if:
    result = 1157920892373161954235709850086879078532699846656405640394575840079131296399 35
```

Listing 19: Integer overflow warnings for public string variables

The final vulnerability discovered is a transaction-ordering dependence one in the option factory smart contract. Transaction-ordering dependence is a type of race condition, which are attacks that involve a function calling an external contract and that contract then taking control of the execution flow [165]. The

outside contract can alter the calling contract's data in unexpected ways and this type of attack can manifest itself through many different forms. As can be seen in the code listing below, a sender can request to have their balance transferred to them via a withdrawal (*Flow1*). If, however, the option factory owner creates another transaction calling the *selfdestruct* (*kill*) method of the contract (*Flow2*) and that one gets mined first, then the total remaining balance from all deposit accounts goes to the contract owner. Possible solutions would be removing the account balance functionality from the contract, so that no ether belonging to customers are stored, or taking away the functionality to destroy the contract and drain its remaining funds from the contract owner. Alternatively, a reputation system could be implemented, which would help mitigate fraudulent seller activity.

```
INFO:symExec:Flow1
Warning: Transaction-Ordering Dependency.
    msg.sender.transfer(_amount)
Flow2
Warning: Transaction-Ordering Dependency.
    selfdestruct(msg.sender)
```

Listing 20: Transaction-ordering dependence (TOD) vulnerability

6.4 Validation

Validation is the part of testing that evaluates whether the user requirements were correctly understood and if the system created meets them to a satisfactory extent [138]. Each one of the original requirements was evaluated and the results are presented in Figure 6.1. As is evident from the figure, almost all the requirements were achieved. Only the last one that concerned the application supporting a multilingual interface was deemed superfluous and was thus not pursued in the end. With regards to those implemented partially, the implementation of requirement 6 could have included more customisation options within the option factories of each seller. Requirement 24 concerning the number of concurrent users supported could not be examined thoroughly, due to constraints imposed by the network on funding new accounts created and on transaction throughput.

ID	Specification	Accomplished	Evaluation
1. Option Writer Functionality			
RQ1	Registering as seller	Yes	Created registry, option factories
RQ2	Keeping track of options sold	Yes	Storing options in arrays and mappings within option factories
RQ3	Automatically calculating settlement amount	Yes	Adjusting *buyerBalance* in *exerciseOption* function
RQ4	Keeping track of internal balances	Yes	Implemented by use of *buyerBalance* mapping
RQ5	Viewing past transactions	Yes	Using *options* array, *Option* struct in option factory, *Transactions Page* in front-end
RQ6	Editing contracts offered	Partially	*markupPercentage* variable in option factory, could have added further customisation parameters and more option contract types
RQ7	Notifying seller	Yes	Events in registry (like *Registered*) and in option factory (like *NewOption*), code to subscribe to events in front-end
2. Option Buyer Functionality			
RQ8	Registering as buyer	Yes	*Register Page* in front-end
RQ9	Viewing market rates	Yes	*Market Rates Page* in front-end, use of APIs to retrieve financial market data
RQ10	Viewing different sellers	Yes	Retrieving and displaying list of sellers in *Sellers Page*
RQ11	Inputting parameters and viewing premiums	Yes	*Sellers Page* in front-end, *calculateOptionPremium* function and *OptionPremium* event in option factory
RQ12	Buying options and storing them	Yes	*buyOption*, *buyOptionWithBalance* functions, *NewOption* event in option factory, *Sellers Page* in front-end
RQ13	Viewing options bought with expiration dates	Yes	*getOptionCount*, *getOptionsByBuyer* functions in option factory, *Transactions Page* in front-end
RQ14	Exercising options	Yes	*exerciseOption* function, *OptionExercise* event in option factory
RQ15	Notifying buyer	Yes	Events in option factory, such as *NewOption*, *OptionExercise* and *Withdrawal*, code to subscribe to events in front-end
3. Maintenance			
RQ16	Using the public Ethereum blockchain network	Yes	Used the public *Rinkeby* test network
RQ17	Terminating a running contract	Yes	*kill* function in option factory
4. Login/Security			
RQ18	Distinguishing between different user types	Yes	In the *Profile Page* sellers are offered the opportunity to customise and deploy their option factory
RQ19	Registering as either buyer or seller	Yes	*Register Page* offers the option to register as a seller
RQ20	Checking if email address is already in use	Yes	*checkEmail* function in *register* front-end component that uses authentication service that queries server
RQ21	Authenticating users upon login	Yes	Authentication guard implemented in front-end
RQ22	Enforcing minimum password strength level	Yes	*validatePassword* function using regular expression
5. Performance			
RQ23	Operational 24/7	Yes	The public Ethereum blockchain network is running 24/7, the web application can run 24/7 on a server
RQ24	Supporting a reasonable number of users concurrently	Partially	The system does not have any limits on the number of users it supports, however there are limits (scalability constraints) with regards to the capacity of the public network and the number of accounts that can be created and funded, which is why this could not be thoroughly evaluated
RQ25	Processing transactions in an appropriate amount of time	Yes	Actions that involve transactions take an amount of time that is within expectations, given the current state of the public Ethereum blockchain network
RQ26	Syncing time should be reasonable	Yes	The application takes a reasonable time to sync to the latest block, given the typical sync times for nodes in the Ethereum network
6. Accessibility			
RQ27	Supporting popular web browser options for DApps	Yes	Application works with both Google Chrome (with the MetaMask extension installed) and the Mist browser for DApps
RQ28	Adapting to different screen sizes	Yes	Angular supports responsive web applications, used Material components, interactive Chart.js for graph
RQ29	Supporting multiple languages	No	Not deemed necessary, was an optional requirement from the start

Figure 6.1: Requirements evaluation

7 Conclusion & Future Work

7.1 Conclusion

A range of simplifying assumptions had to be made in order to create this proof of concept, but the work presented in this report demonstrates that a live deployment of an option trading platform open to the public and secured by a blockchain can be constructed, given adequate resources. However, given the current state of the technology, the systems that can realistically be built on a public Ethereum blockchain running a computationally intensive consensus algorithm like proof-of-work are nowhere near scalable enough and cannot encompass all functionality necessary. As demonstrated through this project, the proper functioning of the system is still dependent on outside entities, for example schedulers or data providers, which are frequently centralised. There also remain challenges with regards to user-friendliness and running complex calculations on-chain. A hybrid system would probably prove more successful, rather than a completely decentralised one. With regards to permissioned blockchains, a shared institutional platform that does not have to rely on public verifiability and network effects for its integrity and security could definitely replace and automate large parts of today's technological infrastructure of the financial system.

When it comes to blockchain development, it is significantly more challenging than standard software development in mature programming languages and ecosystems. One reason is that every tiny change in the code requires first recompiling the contract, then redeploying it to a public test network and subsequently waiting for anywhere from half a minute to a few minutes for a response (due to the low throughput and the current consensus protocols employed). After the contract has been deployed successfully on the network, transactions might still fail, due to a number of reasons, such as insufficient gas amount, gas price set or internal balance of the contract. Other potential issues include network congestion, incorrect parameters or address or simply the miners not picking the specific transaction to include in a block. When a transaction does fail, it is hard to figure out the exact reason without having to rely excessively on experimentation.

With regards to the specifics of the Solidity language used for Ethereum smart contract development, there are again a number of significant limitations. For example, the only logging that can be used for debugging a smart contract written in Solidity is via the use of events. Events, however, are not emitted instantaneously; one needs to wait and listen for them, in a similar manner to an asynchronous API call. Furthermore, even commonplace language constructs like fixed/floating point numbers or string concatenation are not supported yet [126]. Another drawback is the inability to listen for events in a contract-to-contract interaction, even though calls between contracts and their functions work seamlessly [166]. This means that function calls using hashed transactions not only are not able to return values directly, when called from another contract, but using events as the typical workaround solution is also out of the question. The way the author overcame this is by storing the return value in a public variable and then reading it from the other contract.

The aforementioned are a selection of the challenges encountered, however it is definitely worth remembering that the technologies used are still in a nascent phase. Solidity is still in version 0.4 [116], after all, so there is definitely plenty of scope for improvement.

All objectives listed in the introduction were achieved:

1. The first objective was to create an interactive web application to view financial data. The application had to connect to APIs and a central server and database. This has been achieved via the development of a MEAN stack web application, comprised of a Node.js server written using Express.js and an Angular client for the front-end. The application includes features like an authentication guard. It also retrieves financial data from the Alpha Vantage API [167] and displays graphic visualisations using Chart.js [168].
2. The second objective was to model financial call options as smart contracts deployed on a blockchain. This was accomplished by writing a customisable template for a factory smart contract that can be used to create options according to provided parameters. Each seller wishing to register with the system and begin selling financial instruments is given the opportunity to create a bespoke option factory from the template (for example, by customising the fees they want to charge) and subsequently to deploy it live to the Ethereum blockchain. That factory then becomes responsible for keeping records of user balances and bought options, calculating option premiums based on external price data and settling account balances upon execution of a certain option.
3. The third objective consisted of implementing the connection between the two parts of the system, by enabling the web application to make calls to the smart contract code running on the blockchain. This was attained via the use of the web3 JavaScript library collection for Ethereum [81], which provides functionality to enable interaction with a node of the Ethereum peer-to-peer blockchain network in a programmatic manner. More specifically, an Angular service [169] was written in TypeScript that is responsible for data access to and from the blockchain. The components of the front-end web application focus on data presentation [169], whereas the service takes care of retrieving data from the blockchain and sending transactions to be signed and broad-casted by the network node. This design takes advantage of the Dependency Injection pattern [170], as implemented in Angular [171], which essentially means that the contract service gets injected in all the pages that require interaction with the blockchain.

In addition to these initial objectives, the system presented has exceeded the requirements and accomplished two additional achievements.

4. The implementation includes and makes use of oracles for the secure retrieval of external data from within the smart contracts running on the blockchain. A committee of oracles has been created, consisting of three oracles in this case, which retrieve price/market data from different sources. This retrieval can be held to a very high standard of security and honesty, due to its use of a mechanism incorporating TLS notary secrets and Amazon Web Services Virtual Machines [62]. A mock weighted voting scheme has also been designed, as part of an enhanced incentive system to prevent malfeasance. Each price for an underlying asset received from a different data source is assigned a weight, based upon the reputation of the data source, before they are all aggregated to produce the final average price. The weights could be adjusted in the future to reflect renewed views on the trustworthiness of each data provider.
5. A master registry smart contract has been created, which is responsible for registering new sellers and keeping a list of their details, such as name and addresses. This registry is also responsible for interfacing with the oracles to retrieve the price data they have received, calculate the average price and store it, so that it can then be securely passed on to the option factories. In this way, it is guaranteed that the price used by the factories to calculate premiums and account settlements

is the actual one that the oracles have returned, thus facilitating trust in the decentralised system. Therefore, the registry acts as an agent between the oracles and the option factories, enabling the user to get accurate prices through the front-end client. The user then sees the premiums offered by the different sellers and the cheapest one to buy from is selected automatically.

As demonstrated by the system presented, smart contracts can be quite useful in automating things. However, critics have a valid point when they claim that they cannot serve as a panacea [172]. The reason is that there are many eventualities that are unpredictable by nature. Especially when it comes to long-term agreements, a common understanding is necessary as an initial foundation, but what is also needed is ongoing communication, rather than automated devices. The reason is that arguments arising unexpectedly can distort incentives down the road. At the end of the day, computers cannot solve everything by themselves, at least not yet.

7.2 Future Work

Future work could aim towards standardisation and the creation of a library of common types of financial contracts, to serve as templates. These could then be implemented for funding or hedging purposes by different parties. They could even be modular in the sense that elements could be combined to structure more complicated instruments. In general, there is wide scope in investigating and implementing smart contract templates for common use cases that do not necessarily have to lie within the financial domain. Areas that spring to mind include authentication patterns, governance mechanisms and proxy functionality.

Another avenue for improvement relates to the ongoing effort within the broader blockchain community to design systems/mechanisms that are more scalable (in terms of throughput and transaction processing speeds). The system presented in this report does work, but it is limited by the choice of consensus protocols of the public blockchain network (mining, proof-of-work), which put a cap on the speed with which actions can be completed (at least in the current incarnation of the network). Active research areas in public blockchain design that could be taken advantage of in the future include sharding and state channels. On the other hand, a private (consortium) blockchain, secured by a less computationally intensive consensus mechanism (such as proof-of-stake or even just one central validator node authenticating transactions), would perform much better at the task at hand.

In conclusion, there remain usability challenges that are embedded within this technology at its current stage of evolution. However, there is potential as the ecosystem slowly heads towards maturity, especially given wider acceptance and adoption. The discovery and implementation of more efficient algorithms and the development of better tools and frameworks should be highly beneficial, too.

Appendices

A Glossary

Name	Description
blockchain	A blockchain is a public register of transactions between users in a network [173]. The transactions are stored in a way that is secure and cannot be altered. They are encapsulated in blocks, which are in turn linked to each other cryptographically, forming a hierarchical chain. The main benefit of a blockchain is the ensuing traceability and verifiability of the data.
smart contract	A smart contract is a computer protocol intended to digitally facilitate, verify, or enforce the negotiation or performance of a contract [174]. Smart contracts remove the need for intermediaries to verify the credibility of transactions between two parties and allow them to be conducted autonomously.
Ethereum	Ethereum is a decentralized smart contract platform that enables programs to run without censorship [3]. It runs on a custom blockchain that is shared globally.
DApp	An abbreviated form for decentralised application [18]. A DApp consists of the front-end and the smart contracts.

B Tools Used

Rinkeby Ethereum Network

The blockchain instance used is the public Rinkeby test network on the Ethereum platform [20]. The motivation behind this choice of a so-called *testnet* is to enable experimentation on a cutting-edge platform that is exactly the same as the main network, without incurring costs. Furthermore, it allowed the development process to not be limited by licensing/pricing considerations, as would be the case when using other (supposedly) open-source implementations that do still require a license to operate fully.

Remix

Remix is an online browser-based IDE that allows the compilation and debugging of smart contracts written in Solidity [175].

Front-end/client-side

Vanilla JavaScript would suffice, but the author opted to use the Angular web application framework [176], due to familiarity with the development process. The library web3.js [81] for JavaScript provides functionality for programmatic interactions with nodes of an Ethereum blockchain network via the generic JSON RPC specification [177].

NodeJS (Express)

The development server was implemented using the Express application framework [178] for Node.js [179]. This was also compatible with web3.js, however almost all of the functionality was implemented in the Angular application, for ease of development and use.

Truffle

Truffle [180] is an alternate suite of tools for smart contract development, compilation and publishing. It can also be helpful when testing the functionality of the contracts.

Ganache

Ganache [133] used to be called TestRPC, but it has now been renamed and integrated with the Truffle Suite development framework. It is a tool that allows developers to set up a virtual Ethereum blockchain and generate accounts for local development and testing [133].

MetaMask

MetaMask [181] simplifies the use of an application as the end user, by providing account management and wallet functionality. It essentially removes the need for the user to configure and run their own node, in order to interact with the Ethereum blockchain network.

Geth

Geth [182] is the implementation of the Ethereum protocol in the Golang programming language. The application provides a command-line interface that allows the user to create a local blockchain, spin up a new node and join the live Ethereum network or act as a miner, among others.

Mist

Mist [183] is a browser for decentralised web applications. Mist aims to be the equivalent of Chrome or Firefox, but for DApps [183]. However, it is still a work in progress, meaning that there might be security issues. The Ethereum Wallet is similar to Mist, but only works as a wallet, not with every DApp. They are both simply clients/browsers on the front-end side, handling accounts and transactions, so the implementation of a bridge to the blockchain is still necessary. What they essentially do is allow the user to connect to an Ethereum node, create and fund accounts and send transactions. A user has to go through them to connect to a remote node because the official *personal geth API* [184] containing account management functionality is not exposed over Remote Procedure Calls, primarily for security reasons [185], [186].

Atom

There are plug-ins that provide the Atom text editor with functionality to handle Solidity and Serpent files. One such example is etheratom [187], which enables compilation and deployment. Another is language-ethereum [188], which features editing enhancements.

User Interface Design

The inspiration behind the design of the user interface of the system came from Google's Material Design guidelines and principles. The reason is that they are flexible, cross-platform and of an intuitive nature [189]. They have also been developed, tried and tested by experts at Google, according to best practices. The specific incarnation that was utilised was Angular Material, which contains Material Design components optimised for Angular [190].

C Test Logs

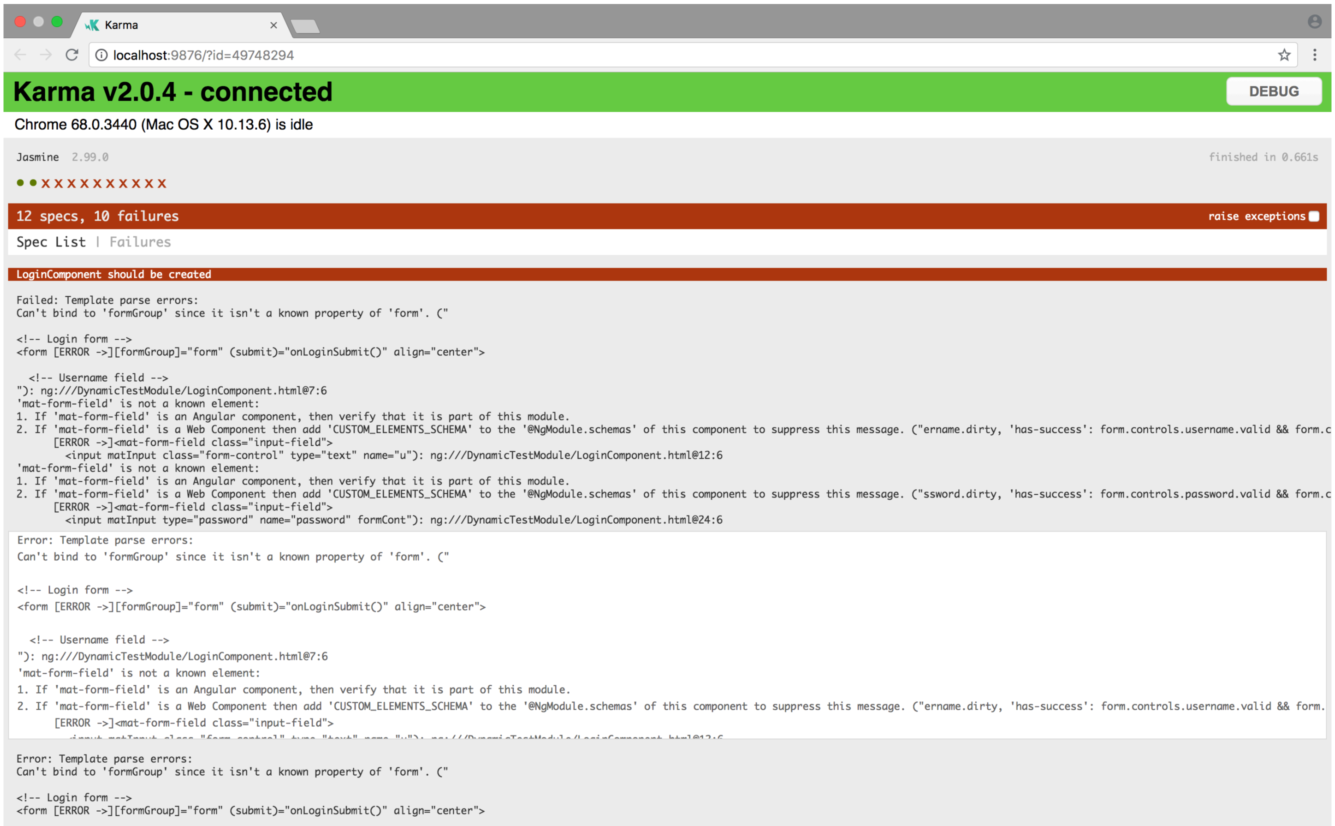

Figure 7.1: Initial output of Karma tests written using the Jasmine framework

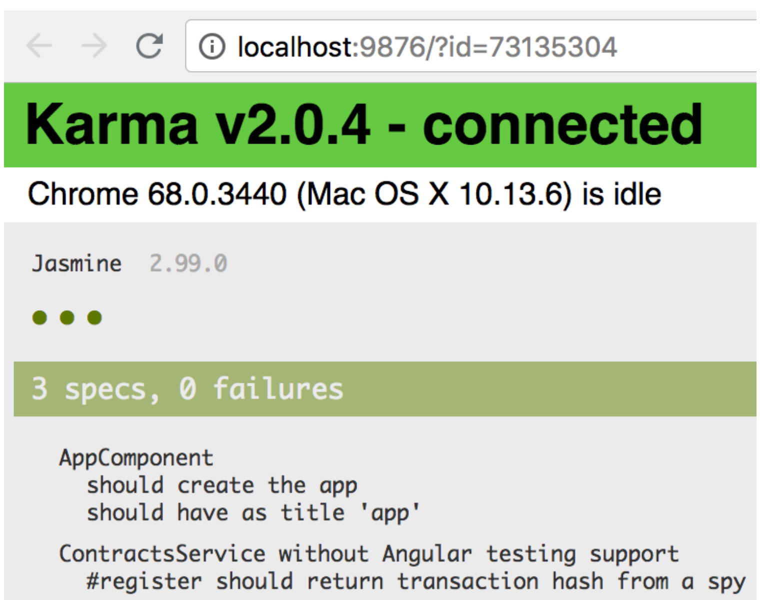

Figure 7.2: Unit testing the register function in the contract service using Jasmine spies

```
Chrome 68.0.3440 (Mac OS X 10.13.6): Executed 6 of 23 SUCCESS (0 secs / 0.098 se
Chrome 68.0.3440 (Mac OS X 10.13.6): Executed 7 of 23 SUCCESS (0 secs / 0.099 se
Chrome 68.0.3440 (Mac OS X 10.13.6): Executed 8 of 23 SUCCESS (0 secs / 0.099 se
Chrome 68.0.3440 (Mac OS X 10.13.6): Executed 9 of 23 SUCCESS (0 secs / 0.101 se
Chrome 68.0.3440 (Mac OS X 10.13.6): Executed 10 of 23 SUCCESS (0 secs / 0.102 s
Chrome 68.0.3440 (Mac OS X 10.13.6): Executed 11 of 23 SUCCESS (0 secs / 0.102 s
Chrome 68.0.3440 (Mac OS X 10.13.6): Executed 12 of 23 SUCCESS (0 secs / 0.102 s
Chrome 68.0.3440 (Mac OS X 10.13.6): Executed 13 of 23 SUCCESS (0 secs / 0.103 s
Chrome 68.0.3440 (Mac OS X 10.13.6): Executed 14 of 23 SUCCESS (0 secs / 0.103 s
Chrome 68.0.3440 (Mac OS X 10.13.6): Executed 15 of 23 SUCCESS (0 secs / 0.103 s
Chrome 68.0.3440 (Mac OS X 10.13.6): Executed 16 of 23 SUCCESS (0 secs / 0.103 s
Chrome 68.0.3440 (Mac OS X 10.13.6): Executed 17 of 23 SUCCESS (0 secs / 0.103 s
Chrome 68.0.3440 (Mac OS X 10.13.6): Executed 18 of 23 SUCCESS (0 secs / 0.103 s
Chrome 68.0.3440 (Mac OS X 10.13.6): Executed 19 of 23 SUCCESS (0 secs / 0.105 s
Chrome 68.0.3440 (Mac OS X 10.13.6): Executed 20 of 23 SUCCESS (0 secs / 0.106 s
Chrome 68.0.3440 (Mac OS X 10.13.6): Executed 21 of 23 SUCCESS (0 secs / 0.106 s
Chrome 68.0.3440 (Mac OS X 10.13.6): Executed 22 of 23 SUCCESS (0 secs / 0.107 s
Chrome 68.0.3440 (Mac OS X 10.13.6): Executed 23 of 23 SUCCESS (0 secs / 0.107 s
Chrome 68.0.3440 (Mac OS X 10.13.6): Executed 23 of 23 SUCCESS (0.116 secs / 0.1
07 secs)
TOTAL: 23 SUCCESS
TOTAL: 23 SUCCESS
TOTAL: 23 SUCCESS
```

Figure 7.3: Final console output of unit tests written in Jasmine and run by Karma

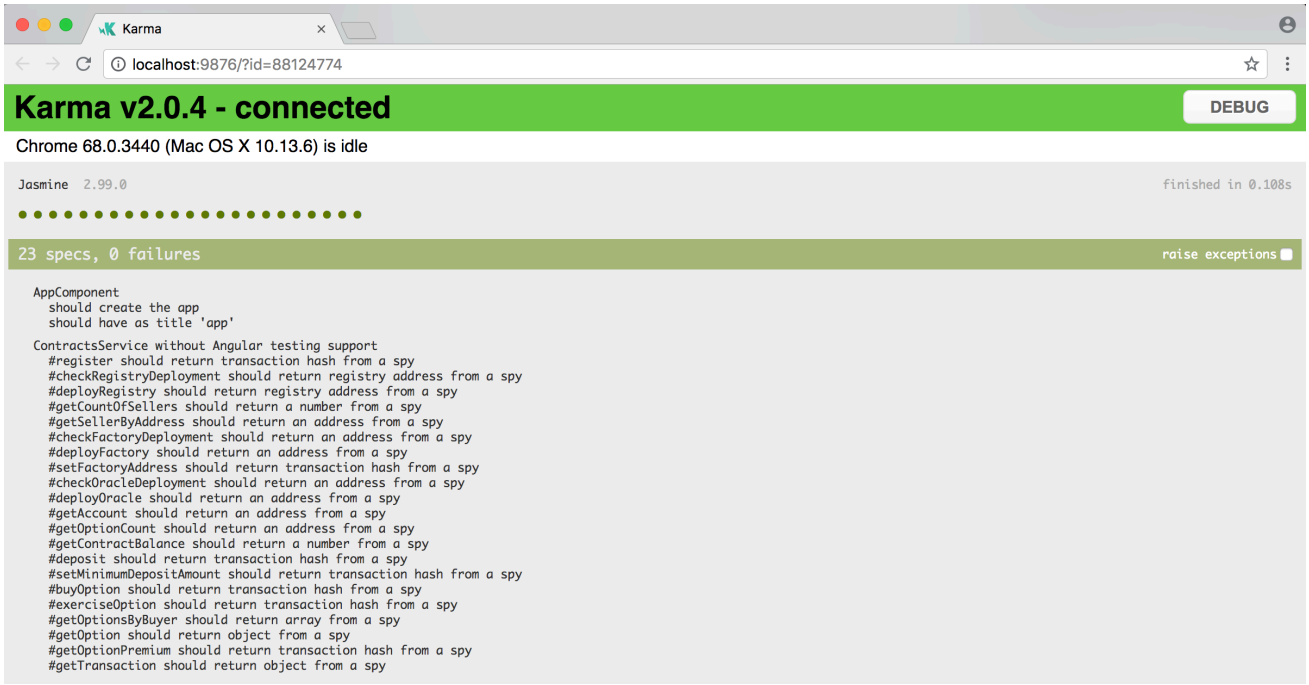

Figure 7.4: Final Karma output

D Security Audit Output

The following is the output of the execution analysis for security vulnerabilities using the Oyente tool [154].

```
root@37b0d1f4789e:/oyente/oyente# python oyente.py -s price_oracle.sol -ce
WARNING:root:You are using evm version 1.8.2. The supported version is 1.7.3
WARNING:root:You are using solc version 0.4.21, The latest supported version is 0.4.19
INFO:root:contract price_oracle.sol:Buffer:
INFO:symExec: ============ Results ===========
INFO:symExec:   EVM Code Coverage:          100.0%
INFO:symExec:   Integer Underflow:          False
INFO:symExec:   Integer Overflow:           False
INFO:symExec:   Parity Multisig Bug 2:      False
INFO:symExec:   Callstack Depth Attack Vulnerability:   False
INFO:symExec:   Transaction-Ordering Dependence (TOD): False
INFO:symExec:   Timestamp Dependency:       False
INFO:symExec:   Re-Entrancy Vulnerability:      False
INFO:symExec: ====== Analysis Completed ======
INFO:root:contract price_oracle.sol:CBOR:
INFO:symExec: ============ Results ===========
INFO:symExec:   EVM Code Coverage:          100.0%
INFO:symExec:   Integer Underflow:          False
INFO:symExec:   Integer Overflow:           False
INFO:symExec:   Parity Multisig Bug 2:      False
INFO:symExec:   Callstack Depth Attack Vulnerability:   False
INFO:symExec:   Transaction-Ordering Dependence (TOD): False
INFO:symExec:   Timestamp Dependency:       False
INFO:symExec:   Re-Entrancy Vulnerability:      False
INFO:symExec: ====== Analysis Completed ======
INFO:root:contract price_oracle.sol:OptionFactory:
INFO:symExec: ============ Results ===========
INFO:symExec:   EVM Code Coverage:          51.0%
INFO:symExec:   Integer Underflow:          True
INFO:symExec:   Integer Overflow:           True
INFO:symExec:   Parity Multisig Bug 2:      False
INFO:symExec:   Callstack Depth Attack Vulnerability:   False
INFO:symExec:   Transaction-Ordering Dependence (TOD): True
INFO:symExec:   Timestamp Dependency:       False
INFO:symExec:   Re-Entrancy Vulnerability:      False
INFO:symExec:price_oracle.sol:1675:41: Warning: Integer Underflow.
                uint settlementAmount = options[_id].exercisePrice - spotPrice
Integer Underflow occurs if:
    options[_id].exercisePrice = 2
    options[_id].exercised = 0
    options[_id].expirationDate = 0
    options[_id] = 1
    optionToBuyer[_id] = 0
price_oracle.sol:1620:5: Warning: Integer Underflow.
    Option[] public options
price_oracle.sol:1637:31: Warning: Integer Underflow.
        uint intrinsicValue = _exercisePrice - spotPrice
INFO:symExec:price_oracle.sol:1672:17: Warning: Integer Overflow.
```

```
                options[_id]
Integer Overflow occurs if:
    options[_id].exercised = 0
    options[_id].expirationDate = 0
    options[_id] =
    72370055773322622139731865630429942408293740416025352524660990004945706024 97
    optionToBuyer[_id] = 0
price_oracle.sol:1645:5: Warning: Integer Overflow.
    function buyOption(string _asset, uint _exercisePrice, uint _expirationDate) external
    payable {
    ^
Spanning multiple lines.
price_oracle.sol:1670:20: Warning: Integer Overflow.
        if (now >= options[_id].expirationDate
Integer Overflow occurs if:
    options[_id] = 1
    optionToBuyer[_id] = 0
price_oracle.sol:1675:41: Warning: Integer Overflow.
                uint settlementAmount = options[_id].exercisePrice
Integer Overflow occurs if:
    options[_id].exercised = 0
    options[_id].expirationDate = 0
    options[_id] = 1
    optionToBuyer[_id] = 0
price_oracle.sol:1671:18: Warning: Integer Overflow.
            if (!options[_id].exercised
Integer Overflow occurs if:
    options[_id].expirationDate = 0
    options[_id] = 1
    optionToBuyer[_id] = 0
price_oracle.sol:1657:5: Warning: Integer Overflow.
    function buyOptionWithBalance(string _asset, uint _exercisePrice, uint _expirationDate)
    external {
    ^
Spanning multiple lines.
price_oracle.sol:1671:18: Warning: Integer Overflow.
            if (!options[_id]
Integer Overflow occurs if:
    options[_id].expirationDate = 0
    options[_id] =
    72370055773322622139731865630429942408293740416025352524660990004945706024 97
    optionToBuyer[_id] = 0
price_oracle.sol:1675:41: Warning: Integer Overflow.
                uint settlementAmount = options[_id]
Integer Overflow occurs if:
    options[_id].exercised = 0
    options[_id].expirationDate = 0
    options[_id] =
    72370055773322622139731865630429942408293740416025352524660990004945706024 97
    optionToBuyer[_id] = 0
price_oracle.sol:159:5: Warning: Integer Overflow.
```

```
    function stringToUint(string s) public pure returns (uint) {
    ^
Spanning multiple lines.
price_oracle.sol:1620:5: Warning: Integer Overflow.
    Option[] public options
price_oracle.sol:1672:17: Warning: Integer Overflow.
                options[_id].exercised
Integer Overflow occurs if:
    options[_id].exercised = 0
    options[_id].expirationDate = 0
    options[_id] = 1
    optionToBuyer[_id] = 0
price_oracle.sol:1670:20: Warning: Integer Overflow.
        if (now >= options[_id]
Integer Overflow occurs if:
    options[_id] =
    72370055773322622139731865630429942408293740416025352524660990004945706024 97
    optionToBuyer[_id] = 0
price_oracle.sol:1727:9: Warning: Integer Overflow.
        buyerBalance[msg.sender] += msg.value
Integer Overflow occurs if:
    buyerBalance[msg.sender] =
    70036696666780064463973016200043117429077326574545407627672066683247125486415
    minimumDepositAmount =
    69722843986276191112704802276904042686523943500539225692285358676886601247599
INFO:symExec:Flow1
price_oracle.sol:1735:9: Warning: Transaction-Ordering Dependency.
        msg.sender.transfer(_amount)
Flow2
price_oracle.sol:1752:9: Warning: Transaction-Ordering Dependency.
        selfdestruct(msg.sender)
INFO:symExec: ====== Analysis Completed ======
INFO:root:contract price_oracle.sol:Ownable:
INFO:symExec: ============ Results ===========
INFO:symExec:   EVM Code Coverage:         57.2%
INFO:symExec:   Integer Underflow:         False
INFO:symExec:   Integer Overflow:          True
INFO:symExec:   Parity Multisig Bug 2:     False
INFO:symExec:   Callstack Depth Attack Vulnerability:  False
INFO:symExec:   Transaction-Ordering Dependence (TOD): False
INFO:symExec:   Timestamp Dependency:      False
INFO:symExec:   Re-Entrancy Vulnerability:     False
INFO:symExec:price_oracle.sol:159:5: Warning: Integer Overflow.
    function stringToUint(string s) public pure returns (uint) {
    ^
Spanning multiple lines.
Integer Overflow occurs if:
    s = 1157920892373161954235709850086879078532699846656405640394575840079131296399 32
INFO:symExec: ====== Analysis Completed ======
INFO:root:contract price_oracle.sol:PriceOracle:
INFO:symExec: ============ Results ===========
```

```
INFO:symExec:    EVM Code Coverage:          28.1%
INFO:symExec:    Integer Underflow:          True
INFO:symExec:    Integer Overflow:           True
INFO:symExec:    Parity Multisig Bug 2:      False
INFO:symExec:    Callstack Depth Attack Vulnerability:  False
INFO:symExec:    Transaction-Ordering Dependence (TOD): False
INFO:symExec:    Timestamp Dependency:       False
INFO:symExec:    Re-Entrancy Vulnerability:      False
INFO:symExec:price_oracle.sol:1443:4: Warning: Integer Underflow.
   string public urlPart2
price_oracle.sol:1441:4: Warning: Integer Underflow.
   string public urlPart1
price_oracle.sol:1472:32: Warning: Integer Underflow.
           emit NewOracleQuery(strConcat(provider, " oracle query was sent, waiting for the
    response...")
Integer Underflow occurs if:
    oraclize = 0
    OAR = 0
price_oracle.sol:1440:4: Warning: Integer Underflow.
   string public provider
price_oracle.sol:1438:1: Warning: Integer Underflow.
contract PriceOracle is usingOraclize, Ownable {
^
Spanning multiple lines.
Integer Underflow occurs if:
    OAR = 0
price_oracle.sol:1442:4: Warning: Integer Underflow.
   string public symbol
price_oracle.sol:1444:4: Warning: Integer Underflow.
   string public price
price_oracle.sol:1470:32: Warning: Integer Underflow.
           emit NewOracleQuery(strConcat(provider, " oracle query was NOT sent, please add
    some ETH for the query fee!")
Integer Underflow occurs if:
    oraclize = 0
    OAR = 0
INFO:symExec:price_oracle.sol:1458:4: Warning: Integer Overflow.
   function __callback(bytes32 id, string result) public {
   ^
Spanning multiple lines.
Integer Overflow occurs if:
    result = 1157920892373161954235709850086879078532699846656405640394575840079131296399 35
price_oracle.sol:1464:4: Warning: Integer Overflow.
   function updateSymbol(string _symbol) public payable onlyContractOwner {
   ^
Spanning multiple lines.
Integer Overflow occurs if:
    _symbol = 1157920892373161954235709850086879078532699846656405640394575840079131296399 35
price_oracle.sol:568:5: Warning: Integer Overflow.
    function __callback(bytes32 myid, string result, bytes proof) public {
    ^
```

```
Spanning multiple lines.
Integer Overflow occurs if:
    result = 1157920892373161954235709850086879078532699846656405640394575840079131296399 35
price_oracle.sol:159:5: Warning: Integer Overflow.
    function stringToUint(string s) public pure returns (uint) {
    ^
Spanning multiple lines.
INFO:symExec: ====== Analysis Completed ======
INFO:root:contract price_oracle.sol:Registry:
INFO:symExec: ============ Results ===========
INFO:symExec:   EVM Code Coverage:         34.0%
INFO:symExec:   Integer Underflow:         True
INFO:symExec:   Integer Overflow:          True
INFO:symExec:   Parity Multisig Bug 2:     False
INFO:symExec:   Callstack Depth Attack Vulnerability:  False
INFO:symExec:   Transaction-Ordering Dependence (TOD): False
INFO:symExec:   Timestamp Dependency:      False
INFO:symExec:   Re-Entrancy Vulnerability:     False
INFO:symExec:price_oracle.sol:1491:2: Warning: Integer Underflow.
  mapping (address => Seller) public sellers
price_oracle.sol:1522:5: Warning: Integer Underflow.
    string public price
price_oracle.sol:1523:5: Warning: Integer Underflow.
    string public averagePrice
INFO:symExec:price_oracle.sol:1483:1: Warning: Integer Overflow.
contract Registry is Ownable, usingOraclize {
^
Spanning multiple lines.
Integer Overflow occurs if:
    addressLUT.push(msg.sender) =
    1057828986581143626399982365905880584115175661133146654754335527488885610065 58
    sellers[_account].account = 0
price_oracle.sol:568:5: Warning: Integer Overflow.
    function __callback(bytes32 myid, string result, bytes proof) public {
    ^
Spanning multiple lines.
price_oracle.sol:565:5: Warning: Integer Overflow.
    function __callback(bytes32 myid, string result) public {
    ^
Spanning multiple lines.
price_oracle.sol:159:5: Warning: Integer Overflow.
    function stringToUint(string s) public pure returns (uint) {
    ^
Spanning multiple lines.
price_oracle.sol:1519:9: Warning: Integer Overflow.
        sellers[msg.sender].factory
price_oracle.sol:1508:3: Warning: Integer Overflow.
    addressLUT.push(msg.sender)
Integer Overflow occurs if:
    addressLUT.push(msg.sender) =
    1157920892373161954235709850086879078532699846656405640394575840079131296399 35
```

```
    sellers[_account].account = 0
price_oracle.sol:1495:5: Warning: Integer Overflow.
    address[] public addressLUT
INFO:symExec: ====== Analysis Completed ======
INFO:root:contract price_oracle.sol:SafeMath:
INFO:symExec: ============ Results ===========
INFO:symExec:   EVM Code Coverage:        100.0%
INFO:symExec:   Integer Underflow:        False
INFO:symExec:   Integer Overflow:         False
INFO:symExec:   Parity Multisig Bug 2:    False
INFO:symExec:   Callstack Depth Attack Vulnerability:  False
INFO:symExec:   Transaction-Ordering Dependence (TOD): False
INFO:symExec:   Timestamp Dependency:     False
INFO:symExec:   Re-Entrancy Vulnerability:    False
INFO:symExec: ====== Analysis Completed ======
INFO:root:contract price_oracle.sol:SafeMath16:
INFO:symExec: ============ Results ===========
INFO:symExec:   EVM Code Coverage:        100.0%
INFO:symExec:   Integer Underflow:        False
INFO:symExec:   Integer Overflow:         False
INFO:symExec:   Parity Multisig Bug 2:    False
INFO:symExec:   Callstack Depth Attack Vulnerability:  False
INFO:symExec:   Transaction-Ordering Dependence (TOD): False
INFO:symExec:   Timestamp Dependency:     False
INFO:symExec:   Re-Entrancy Vulnerability:    False
INFO:symExec: ====== Analysis Completed ======
INFO:root:contract price_oracle.sol:SafeMath32:
INFO:symExec: ============ Results ===========
INFO:symExec:   EVM Code Coverage:        100.0%
INFO:symExec:   Integer Underflow:        False
INFO:symExec:   Integer Overflow:         False
INFO:symExec:   Parity Multisig Bug 2:    False
INFO:symExec:   Callstack Depth Attack Vulnerability:  False
INFO:symExec:   Transaction-Ordering Dependence (TOD): False
INFO:symExec:   Timestamp Dependency:     False
INFO:symExec:   Re-Entrancy Vulnerability:    False
INFO:symExec: ====== Analysis Completed ======
INFO:root:contract price_oracle.sol:usingOraclize:
INFO:symExec: ============ Results ===========
INFO:symExec:   EVM Code Coverage:        40.6%
INFO:symExec:   Integer Underflow:        False
INFO:symExec:   Integer Overflow:         True
INFO:symExec:   Parity Multisig Bug 2:    False
INFO:symExec:   Callstack Depth Attack Vulnerability:  False
INFO:symExec:   Transaction-Ordering Dependence (TOD): False
INFO:symExec:   Timestamp Dependency:     False
INFO:symExec:   Re-Entrancy Vulnerability:    False
INFO:symExec:price_oracle.sol:565:5: Warning: Integer Overflow.
    function __callback(bytes32 myid, string result) public {
    ^
Spanning multiple lines.
```

```
292 Integer Overflow occurs if:
293     result = 1157920892373161954235709850086879078532699846656405640394575840079131296399 32
294 price_oracle.sol:568:5: Warning: Integer Overflow.
295     function __callback(bytes32 myid, string result, bytes proof) public {
296     ^
297 Spanning multiple lines.
298 Integer Overflow occurs if:
299     result = 1157920892373161954235709850086879078532699846656405640394575840079131296399 35
300 INFO:symExec: ====== Analysis Completed ======
```

Listing 21: Terminal output of security audit using Oyente

E User Manual

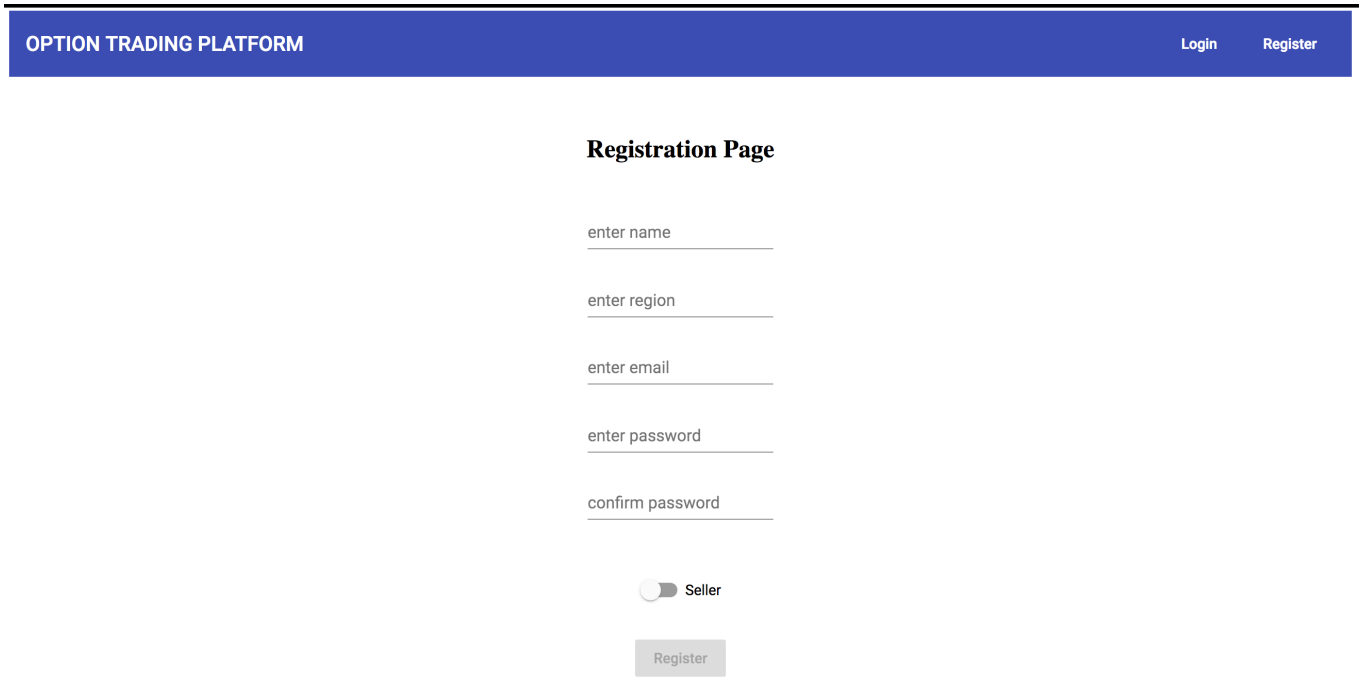

Figure 7.5: Registration page

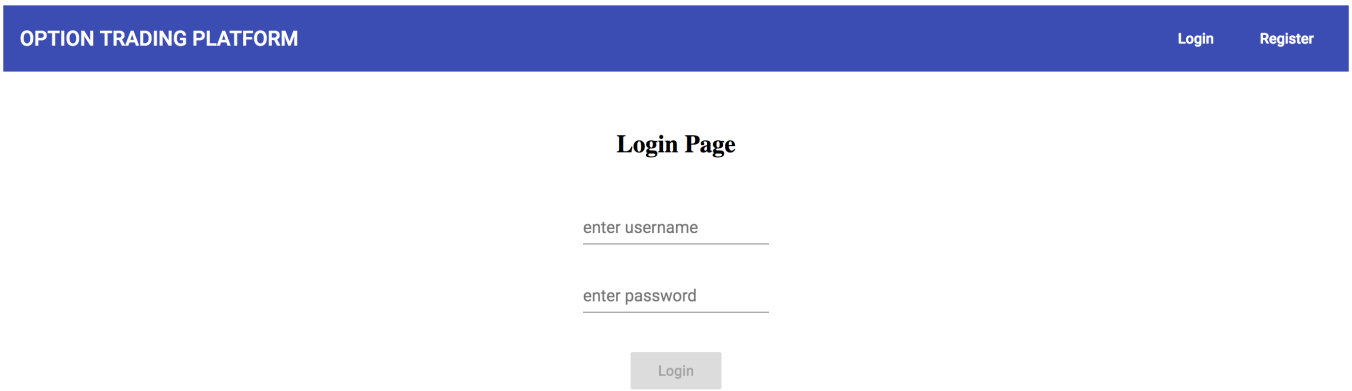

Figure 7.6: Login page

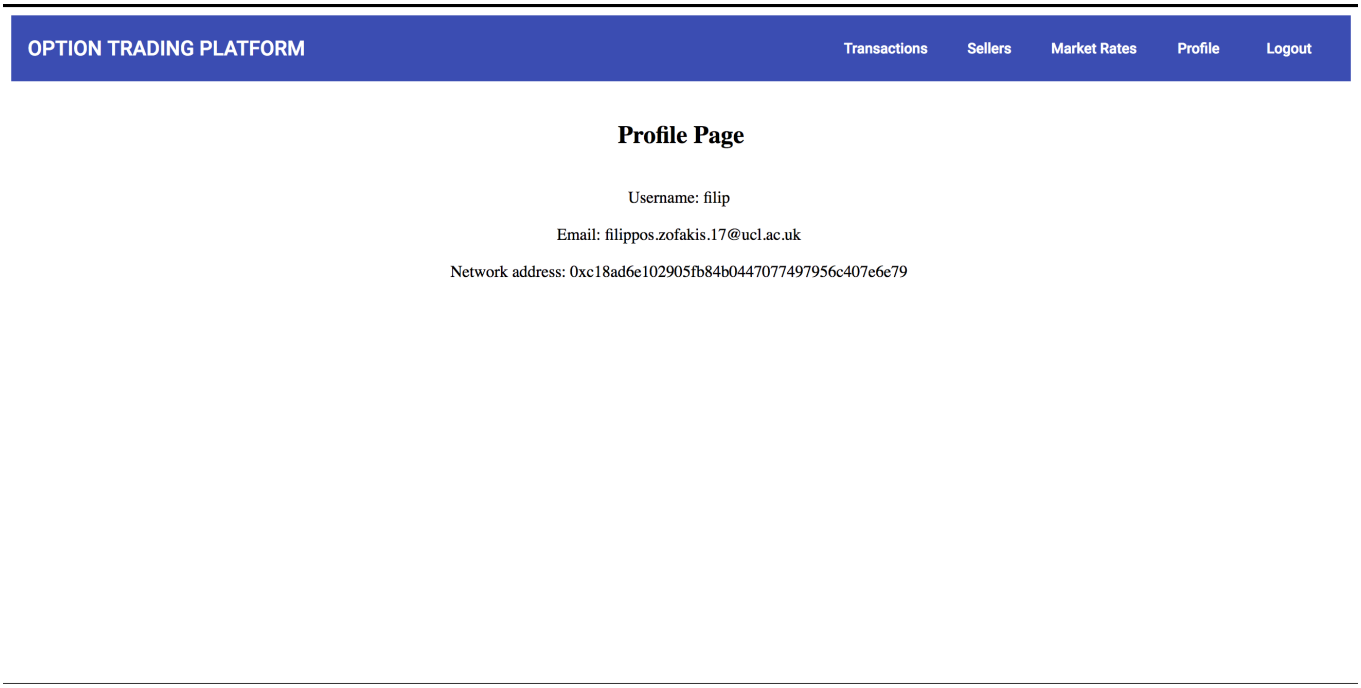

Figure 7.7: Profile page

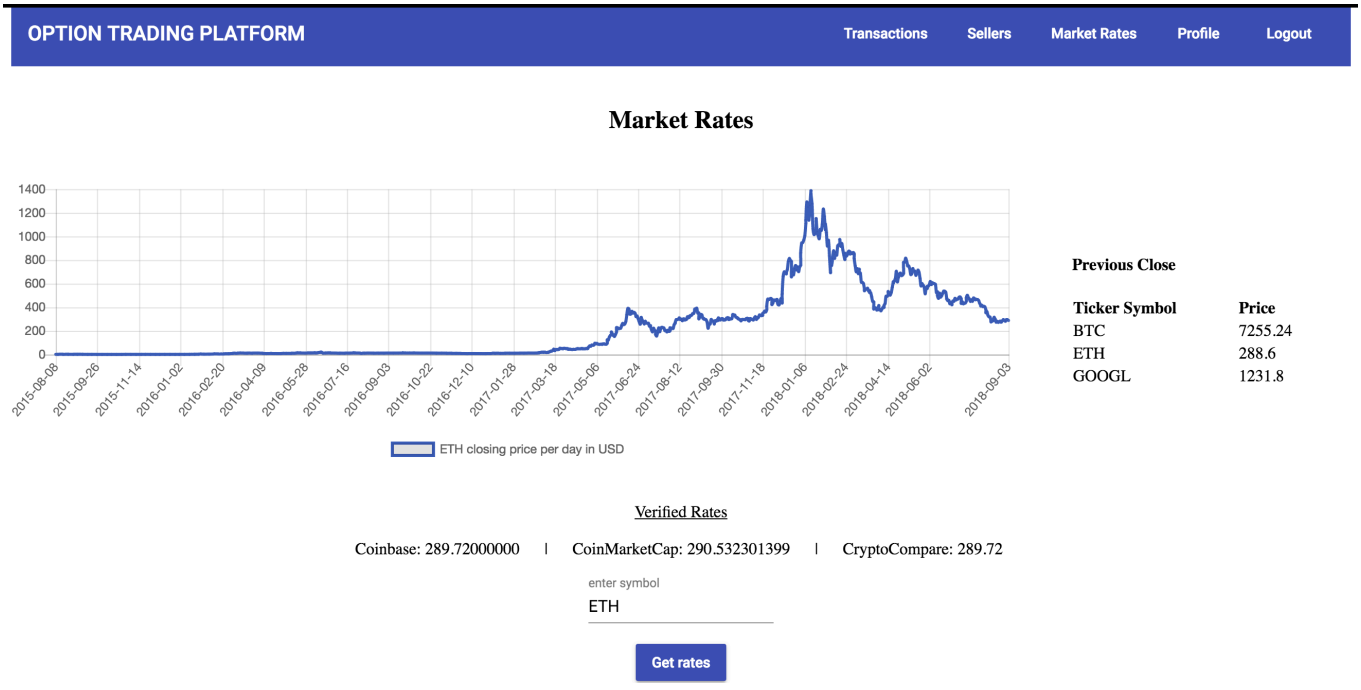

Figure 7.8: Market rates page

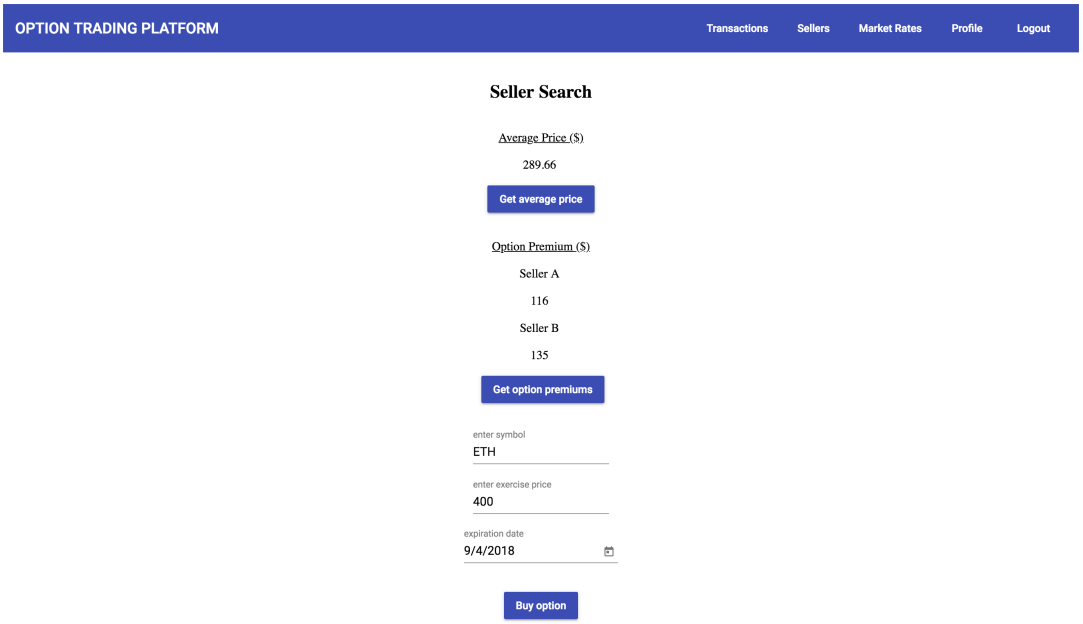

Figure 7.9: Sellers page

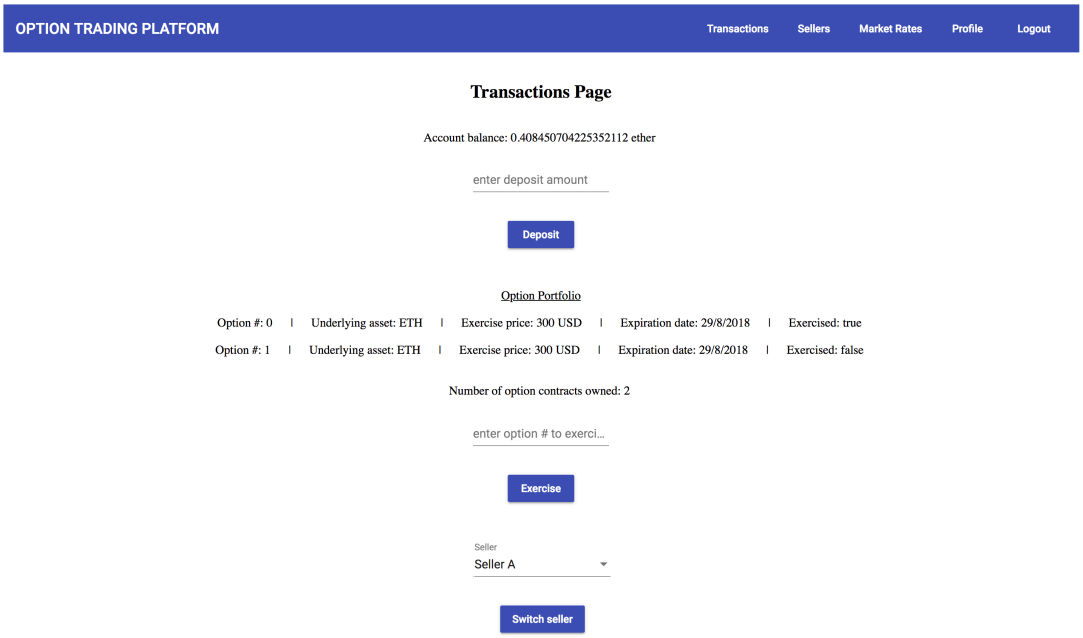

Figure 7.10: Transactions page

F System Manual

General System

The System consists of an Angular (TypeScript) front-end, a NodeJS server, smart contract code and an Ethereum Client.

To set up and host the application locally, the user should install the Express for Node.js and Angular frameworks. They can be installed all together, but it would be advised to split them into two parts, namely client (Angular) and server (Node.js and Express).

- Node.js [179] is an open-source JavaScript run-time environment that is commonly used as a web server. Along with Node.js, it is highly recommended to install npm [191], which is the de facto package manager that handles Node modules written in JavaScript. npm is therefore a useful tool to install and manage project dependencies.
- Express [178] is a minimal web framework for Node.js. It is particularly useful, when it comes to creating APIs.

  After downloading the source code files and navigating to the server folder, the following command has to be run to install all the necessary modules/dependencies.

```
npm install
```

Listing 22: Command to install dependencies

  After both Node.js, Express.js and the other dependencies required have been installed, the server can now be run from the directory where the *server.js* file is located by using the following command:

```
node server
```

Listing 23: Command to run the Node.js server

- Angular [176] is an open-source framework for front-end web applications that was originally created by Google. It is recommended to use the Angular CLI [192], which is a command line interface that speeds up development and follows best practices by default. After downloading the client source code files and navigating to the client folder, the following command has to be run to install all the necessary modules, including Angular and web3.js [81], among others.

```
npm install
```

Listing 24: Command to install dependencies

  To run the client locally (in the browser), the following Angular CLI command can be used:

```
ng serve —o
```

Listing 25: Command to run the Angular client for development and testing

  The *open* option ('–open' or simply '–o') instructs the CLI to open the default browser automatically.

  When building for deployment/production, the command that should be used [193] instead is:

```
ng build
```

Listing 26: Command to build the client for production

  The files to be deployed are then placed in the *dist* folder, inside the client's root directory. The client application can then be deployed using common cloud services, such as Azure Web Apps [194].

- When it comes to the smart contracts, the code contained in the *.sol* files needs to be compiled and deployed to the network. Tools such as the Remix IDE for Solidity [175] or the Truffle Framework [180] can help with compilation, from which we derive both the EVM byte-code (as a hex string) [195] to be sent as transaction data [196], when creating the contract, as well as the Application Binary Interface (ABI) [197]. The ABI is in JSON format and is essentially the encoding schema of the data, thus enabling interaction with a smart contract, both in a contract-to-contract manner, as well as from the outside [197]. For the purposes of this project, it serves as the link between the web3 library and the binary program module containing the smart contract code on the blockchain [198]. The application code contains functionality to deploy code and create contracts programmatically, rather than manually. The relevant code is in the *./client/src/app/services/contract.service.ts* file and the ABI is stored in the assets folder, namely in *./client/src/assets/contractABI.json*.
- The last piece to make everything work is the Ethereum client, which sits in-between the front-end and the smart contract code running on the blockchain [98]. The client is used to perform calls and sign transactions. One option is to run a local node and use a standalone client like Geth [182], the official implementation of Ethereum's protocol in the Go programming language. A more straightforward option recommended for beginners is to use the Mist Browser [183], which already contains a built-in Ethereum Wallet with basic account management and transaction signing functionality. Another very popular alternative is to install the MetaMask add-on [181] for traditional web browsers (Chrome, Firefox, Opera, Brave), in order to turn them into Web 3.0 enabled ones.

Private Blockchain Version

To use the system with a private blockchain network running locally, a few steps have to be taken first. First of all, the Ganache tool [133] from the Truffle Suite has to be installed because it enables developers to set up and spin a local test blockchain in a straightforward manner. The author recommends using the command line interface, rather than the GUI version of the tool, since the former offers more configuration options and parameters. After it is installed, the command to create the genesis (initial) block and start simulating the network locally is the following:

```
ganache-cli --db="./data/blockchain/" -i="5" -d --mnemonic="box fitness lady fever calm crater
    sense excuse flee festival length wagon"
```

Listing 27: Command to create and run local test Ethereum blockchain

The most important parameter to note is the *-db* one, which indicates the path where the files of the blockchain will be stored. If the location supplied is not empty, then the tool starts from the latest block of the chain already stored there, which means that this parameter allows the reuse of test networks created before and continues from the last block saved earlier. The mnemonic parameter, on the other hand, serves as a seed phrase, ensuring that the blockchain created is the same one every single time (same addresses, for example). It thus enables deterministic execution and is ideal for reproducible experiments or testing processes.

The next step in setting up the system to work with a local network is deploying a custom version of the Oraclize Ethereum bridge [134], which enables any Ethereum network to use oracles of the type utilised in this project. The bridge creates a link between the local blockchain network and the outside world, thus acting as a connector. The command to create the oracle instance is the following:

```
ethereum-bridge -H localhost:8545 -a 9
```

Listing 28: Command to create an Ethereum bridge for the network running on localhost

The actual local port where the blockchain can be reached should be substituted for the value *8545* (it is either 7545 or 8545 by default). The parameter *-a* indicates the account to use to deploy the connector, which in this case is the one at position *9* (the 10th one created). After the oracle connector is instantiated and deployed, the tool outputs the name of that specific instance, including a timestamp. To reuse the same one, the following command must be entered:

```
ethereum-bridge --instance oracle_instance_20180829T004315.json -dev
```

Listing 29: Command to run an Ethereum oracle instance created earlier

The actual instance created should obviously be substituted for the *20180829T004315* part. The *-dev* flag instructs the tool that it should run in development mode, which enhances compatibility with test blockchains.

The final step involves modifying the constructor of the oracle smart contract retrieving prices, so that it works with the local connector address. The following line should be added to the constructor of each oracle created:

```
OAR = OraclizeAddrResolverI(0x6f485C8BF6fc43eA212E93BBF8ce046C7f1cb475);
```

Listing 30: Command to be placed within constructor of oracle smart contract

The address in the brackets should be changed to the one indicated by the bridge's console output.

As a closing note, the oracle connector sometimes skips running queries, if the same ones have been executed earlier. It maintains a log, which can be found in the following location:

```
/usr/local/lib/node_modules/ethereum-bridge/database
```

Listing 31: Location of Ethereum bridge log files

If the queries are not getting executed, then the log files can be cleaned (erased) to make the bridge run them again.

References

[1] The Boston Consulting Group, 'Cost benefit analysis of shortening the settlement cycle', The Depository Trust and Clearing Corporation, New York City, New York, United States, Commissioned study, 1st Oct. 2012, p. 18.

[2] T. G. Group, 'Cost benefit analysis of shortening the settlement cycle', European Commission, Brussels, Belgium, Commissioned study, 1st Jan. 2001, p. 2. [Online]. Available: http://ec.europa.eu/internal_market/financial-markets/docs/clearing/first_giovannini_report_en.pdf.

[3] *Ethereum: Blockchain app platform*, https://www.ethereum.org/, (Accessed on 07/26/2018), 2018.

[4] Z. Zheng, S. Xie, H.-N. Dai and H. Wang, 'Blockchain challenges and opportunities: A survey', *Work Pap.–2016*, 2016.

[5] Z. Zheng, S. Xie, H. Dai, X. Chen and H. Wang, 'An overview of blockchain technology: Architecture, consensus, and future trends', in *Big Data (BigData Congress), 2017 IEEE International Congress on*, IEEE, 2017, pp. 557–564.

[6] M. E. Peck, 'Blockchain world-do you need a blockchain? this chart will tell you if the technology can solve your problem', *IEEE Spectrum*, vol. 54, no. 10, pp. 38–60, 2017.

[7] C. Catalini and J. S. Gans, 'Some simple economics of the blockchain', National Bureau of Economic Research, Tech. Rep., 2016.

[8] D. Brandon, 'The blockchain: The future of business information systems', *International Journal of the Academic Business World*, vol. 10, no. 2, pp. 33–40, 2016.

[9] S. Shafer, 'Blockchain and cryptocurrencies', 2018.

[10] J. Plansky, T. O'Donnell and K. Richards, 'A strategist's guide to blockchain', *PwC Report*, 2016.

[11] K. Wüst and A. Gervais, 'Do you need a blockchain?', *IACR Cryptology ePrint Archive*, vol. 2017, p. 375, 2017.

[12] L. DalleMule and T. H. Davenport, 'What's your data strategy?', *Harvard Business Review*, vol. 95, no. 3, pp. 112–121, 2017.

[13] M. Swan, *Blockchain: Blueprint for a new economy*. " O'Reilly Media, Inc.", 2015.

[14] F. Idelberger, G. Governatori, R. Riveret and G. Sartor, 'Evaluation of logic-based smart contracts for blockchain systems', in *International Symposium on Rules and Rule Markup Languages for the Semantic Web*, Springer, 2016, pp. 167–183.

[15] R. Koulu, 'Blockchains and online dispute resolution: Smart contracts as an alternative to enforcement', *SCRIPTed*, vol. 13, p. 40, 2016.

[16] G. W. Peters and E. Panayi, 'Understanding modern banking ledgers through blockchain technologies: Future of transaction processing and smart contracts on the internet of money', in *Banking Beyond Banks and Money*, Springer, 2016, pp. 239–278.

[17] G. Roberts, 'Web application basics', University College London, 2018.

[18] *What is a dapp?*, https://ethereum.stackexchange.com/questions/383/what-is-a-dapp, (Accessed on 07/18/2018), 2018.

[19] *Solidity*, https://solidity.readthedocs.io/en/v0.4.24/, (Accessed on 08/29/2018).

[20] *Rinkeby: Ethereum testnet*, https://www.rinkeby.io/#stats, Aug. 2018.

[21] J. Bai, *What's an api? 3 ways to explain it*, 2014. [Online]. Available: http://developer.pearson.com/blog/whats-api-3-ways-explain-it.

[22] *Block timestamp*, https://en.bitcoin.it/wiki/Block_timestamp, (Accessed on 08/08/2018), 2016.

[23] S. Nakamoto, 'Bitcoin: A peer-to-peer electronic cash system', 2008.

[24] *Block*, https://en.bitcoin.it/wiki/Block, (Accessed on 08/09/2018).

[25] J. Poon, 2017. [Online]. Available: https://blog.iqoption.com/en/how-do-casper-plasma-and-other-ethereum-upgrades-works/.

[26] D. Pointcheval and J. Stern, 'Security arguments for digital signatures and blind signatures', *Journal of cryptology*, vol. 13, no. 3, pp. 361–396, 2000.

[27] S. Brands, 'Untraceable off-line cash in wallet with observers', in *Annual international cryptology conference*, Springer, 1993, pp. 302–318.

[28] E. Ephrati and J. S. Rosenschein, 'The clarke tax as a consensus mechanism among automated agents.', in *AAAI*, vol. 91, 1991, pp. 173–178.

[29] M. J. Fischer, N. A. Lynch and M. S. Paterson, 'Impossibility of distributed consensus with one faulty process', *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[30] *Dapp and optimize gas solution*, https://medium.com/@yangnana11/dapp-and-optimize-gas-solution-8c586ae0fdb4, (Accessed on 08/09/2018).

[31] M. Swan, 'Blockchain thinking: The brain as a decentralized autonomous corporation [commentary]', *IEEE Technology and Society Magazine*, vol. 34, no. 4, pp. 41–52, 2015.

[32] *Does every node execute the contract code for each transaction?*, https://ethereum.stackexchange.com/questions/357/does-every-node-execute-the-contract-code-for-each-transaction, (Accessed on 08/30/2018).

[33] *Where and how application data is stored in ethereum?*, https://www.singulargarden.com/blog/storage-and-dapps-on-ethereum-blockchain/, (Accessed on 08/09/2018).

[34] G. Wood, 'Ethereum: A secure decentralised generalised transaction ledger', *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[35] *Smart contracts*, http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html, (Accessed on 08/09/2018).

[36] *Bitcoin speculation*, https://en.wikipedia.org/wiki/Nick_Szabo#Bitcoin_speculation, (Accessed on 08/09/2018).

[37] TechCrunch, *Decentralizing everything with ethereum's vitalik buterin | disrupt sf 2017*, https://www.youtube.com/watch?v=WSN5BaCzsbo, Accessed on 22/08/2018.

[38] *The ricardian contract*, http://iang.org/papers/ricardian_contract.html, (Accessed on 08/09/2018).

[39] O. Williamson, 'Calculativeness, trust, and economic organization', pp. 453–486, Apr. 1993.

[40] *Alibaba cross-border payments patent filing*, https://www.scribd.com/document/382854443/Alibaba-cross-border-payments-patent-filing, (Accessed on 08/10/2018).

[41] *Demystifying blockchain and consensus mechanisms - everything you wanted to know but were never told*, https://blogs.oracle.com/integration/demystifying-blockchain-and-consensus-mechanisms-everything-you-wanted-to-know-but-were-never-told, (Accessed on 08/16/2018).

[42] A. Beikverdi and J. Song, 'Trend of centralization in bitcoin's distributed network', in *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2015 16th IEEE/ACIS International Conference on*, IEEE, 2015, pp. 1–6.

[43] R. Böhme, N. Christin, B. Edelman and T. Moore, 'Bitcoin: Economics, technology, and governance', *Journal of Economic Perspectives*, vol. 29, no. 2, pp. 213–38, 2015.

[44] *Privacy on the blockchain*, https://blog.ethereum.org/2016/01/15/privacy-on-the-blockchain/, (Accessed on 08/29/2018).

[45] *Who has the power in enterprise blockchains?*, https://www.ibm.com/blogs/blockchain/2018/02/who-has-the-power-in-enterprise-blockchains/, (Accessed on 08/16/2018).

[46] L. Wang and Y. Liu, 'Exploring miner evolution in bitcoin network', in *International Conference on Passive and Active Network Measurement*, Springer, 2015, pp. 290–302.

[47] T. L. Foundation. (Aug. 2018). Hyperledger, [Online]. Available: https://www.hyperledger.org/.

[48] R3. (Aug. 2018). The corda platform, [Online]. Available: https://www.r3.com/corda-platform/.

[49] *Four projects seeking to solve ethereum's privacy paradox*, https://www.coindesk.com/four-projects-seek-solve-ethereums-privacy-paradox/, (Accessed on 08/29/2018).

[50] *Monero - private digital currency*, https://getmonero.org/, (Accessed on 08/29/2018).

[51] *Quorum - advancing blockchain technology*, https://www.jpmorgan.com/global/Quorum, (Accessed on 08/31/2018).

[52] *Block.one's proposal for eos constitution v2.0*, https://block.one/news/block-ones-proposal-for-eos-constitution-v2-0/, (Accessed on 08/13/2018).

[53] C. D. Clack, V. A. Bakshi and L. Braine, 'Smart contract templates: Foundations, design landscape and research directions', Barclays Bank PLC, London, United Kingdom, Commissioned study, 4th Aug. 2016, p. 15. [Online]. Available: https://arxiv.org/pdf/1608.00771.pdf.

[54] P. De Filippi and A. Wright, *Object-oriented analysis and design with applications*, 2. Edition, Ed. Addison-Wesley, 1994.

[55] P. De Filippi and S. Hassan, 'Blockchain technology as a regulatory technology: From code is law to law is code', pp. 1–23, Jan. 2018.

[56] *How dos attacks threaten distributed networks and how to fight them*, https://www.radixdlt.com/post/what-is-a-denial-of-service-attack, (Accessed on 08/14/2018), 2018.

[57] M. Harris and A. Raviv, 'Corporate governance: Voting rights and majority rules', pp. 203–235, Aug. 1988.

[58] B. Vitalik, *Introduction to cryptoeconomics*, https://vitalik.ca/files/intro_cryptoeconomics.pdf, (Accessed on 14/08/2018).

[59] Jun. 2018. [Online]. Available: https://en.wikipedia.org/wiki/Oracle_machine.

[60] Aug. 2018. [Online]. Available: https://blockchainhub.net/blockchain-oracles/.

[61] Dec. 2017. [Online]. Available: https://en.wikipedia.org/wiki/Random_oracle.

[62] Apr. 2017. [Online]. Available: https://ethereum.stackexchange.com/questions/201/how-does-oraclize-handle-the-tlsnotary-secret.

[63] TLSnotary, 'Tlsnotary - a mechanism for independently audited https sessions', pp. 1–10, Sep. 2014.

[64] Z. Rjašková, *Electronic voting schemes*, https://people.ksp.sk/~zuzka/elevote.pdf, (Accessed on 14/08/2018).

[65] *Contracts: Events*, https://solidity.readthedocs.io/en/v0.4.24/contracts.html#events, (Accessed on 08/19/2018).

[66] *Contracts - inheritance*, http://solidity.readthedocs.io/en/v0.4.21/contracts.html#inheritance, (Accessed on 08/11/2018), 2017.

[67] *The ultimate end-to-end tutorial to create and deploy a fully decentralized dapp in ethereum*, https://medium.com/@merunasgrincalaitis/the-ultimate-end-to-end-tutorial-to-create-and-deploy-a-fully-descentralized-dapp-in-ethereum-18f0cf6d7e0e, (Accessed on 08/11/2018), 2017.

[68] *Proxy libraries in solidity*, https://blog.zeppelin.solutions/proxy-libraries-in-solidity-79fbe4b970fd, (Accessed on 08/09/2018).

[69] *Summary of ethereum upgradeable smart contract r&d*, https://blog.indorse.io/ethereum-upgradeable-smart-contract-strategies-456350d0557c, (Accessed on 08/11/2018), 2018.

[70] *Proxy patterns*, https://blog.zeppelinos.org/proxy-patterns/, (Accessed on 08/11/2018), 2018.

[71] *Upgradeability using unstructured storage*, https://blog.zeppelinos.org/upgradeability-using-unstructured-storage/, (Accessed on 08/12/2018), 2018.

[72] R. McQuain Feng, 'T25 - mips stack', CS@VT November 2009 Computer Organization I, 2009.

[73] G. Roberts, 'Software engineering', University College London, 2018.

[74] F. P. Brooks Jr., *The mythical man-month: Essays on software engineering*, A. Edition, Ed. Addison-Wesley, 1995.

[75] D. Consortium, *Atern handbook*, 2008. [Online]. Available: https://www.agilebusiness.org/content/moscow-prioritisation-0.

[76] G. Roberts, 'Use cases part i', University College London, 2018.

[77] G. Booch, J. Rumbaugh and I. Jacobson, *The unified modeling language user guide*, F. Edition, Ed. Addison-Wesley, 1998, pp. 31, 197.

[78] G. Roberts, 'Compgc22 - the unified modelling language', University College London, 2018.

[79] Guru99, 2018. [Online]. Available: https://www.guru99.com/traceability-matrix.html.

[80] *Typical architecture of a dapp with a browser client*, https://ethereum.stackexchange.com/questions/12186/typical-architecture-of-a-dapp-with-a-browser-client, (Accessed on 08/19/2018), 2017.

[81] *Javascript api*, https://github.com/ethereum/wiki/wiki/JavaScript-API, (Accessed on 08/19/2018), 2018.

[82] *Web3 provider engine*, https://github.com/MetaMask/provider-engine, (Accessed on 08/19/2018), 2018.

[83] M. Ohtamaa. (Feb. 2017). Web3.js execution - server vs browser, [Online]. Available: https://ethereum.stackexchange.com/questions/11991/web3-js-execution-server-vs-browser.

[84] *Ethereum yellow paper*, https://github.com/ethereum/yellowpaper, (Accessed on 08/19/2018), 2018.

[85] atomh33ls, *Ethereum block architecture*, https://ethereum.stackexchange.com/questions/268/ethereum-block-architecture/6413#6413, (Accessed on 08/19/2018), 2016.

[86] A. Chakravarty. (Sep. 2017). Here's how i built a private blockchain network, and you can too, [Online]. Available: https://hackernoon.com/heres-how-i-built-a-private-blockchain-network-and-you-can-too-62ca7db556c0.

[87] S. W. Ambler, *Uml 2 deployment diagrams: An agile introduction*, http://www.agilemodeling.com/artifacts/deploymentDiagram.htm, (Accessed on 08/25/2018), 2018.

[88] *Deployment diagrams overview*, https://www.uml-diagrams.org/deployment-diagrams-overview.html, (Accessed on 08/25/2018), 2018.

[89] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1994.

[90] S. W. Ambler, *Uml 2 component diagrams: An agile introduction*, 2018. [Online]. Available: http://agilemodeling.com/artifacts/componentDiagram.htm.

[91] T. O. BokkyPooBah, *What are ipc and rpc?*, https://ethereum.stackexchange.com/questions/10681/what-are-ipc-and-rpc, (Accessed on 08/25/2018).

[92] G. Booch, J. Rumbaugh and I. Jacobson, *The unified modeling language reference manual*, S. Edition, Ed. Addison-Wesley, 2004, p. 217.

[93] M. Fowler, *Patterns of enterprise application architecture*. Boston, MA, USA: Pearson Education, Inc., 2003, pp. 19–20, ISBN: 0321127420.

[94] *Uml 2 sequence diagrams: An agile introduction*, http://www.agilemodeling.com/artifacts/sequenceDiagram.htm, (Accessed on 08/20/2018).

[95] *Common patterns: Restricting access*, https://solidity.readthedocs.io/en/v0.4.24/common-patterns.html#restricting-access, (Accessed on 08/19/2018).

[96] *Light vs. full node*, https://iota.readme.io/docs/light-vs-full-node, (Accessed on 08/19/2018).

[97] G. Pandu Rangarao. (Aug. 2018). Dapp architecture, [Online]. Available: https://sites.google.com/site/blockchaintutorial/dapp-architecture.

[98] *Choosing a client*, http://ethdocs.org/en/latest/ethereum-clients/choosing-a-client.html, (Accessed on 07/16/2018), 2016.

[99] *Connecting to ethereum clients*, http://ethdocs.org/en/latest/connecting-to-clients/, (Accessed on 07/16/2018), 2016.

[100] *Ipfs is the distributed web*, https://ipfs.io/, (Accessed on 07/16/2018), 2018.

[101] *1. introduction*, http://swarm-guide.readthedocs.io/en/latest/introduction.html, (Accessed on 07/16/2018), 2018.

[102] *Whisper*, https://github.com/ethereum/wiki/wiki/Whisper, (Accessed on 07/18/2018), 2018.

[103] S. Grybniak, *Advantages and disadvantages of smart contracts in financial blockchain systems*, https://hackernoon.com/advantages-and-disadvantages-of-smart-contracts-in-financial-blockchain-systems-3a443145ae1c, (Accessed on 07/17/2018), 2017.

[104] C. Lim, T. Saw and C. Sargeant, *Smart contracts: Bridging the gap between expectation and reality*, https://www.law.ox.ac.uk/business-law-blog/blog/2016/07/smart-contracts-bridging-gap-between-expectation-and-reality, (Accessed on 07/17/2018), 2016.

[105] pabloruiz55, *The structure of a dapp and contracts compared to a traditional web platform*, https://ethereum.stackexchange.com/questions/28792/the-structure-of-a-dapp-and-contracts-compared-to-a-traditional-web-platform?rq=1, (Accessed on 08/19/2018), 2017.

[106] AusIV, *Dapp storage for data other than transactions?*, https://ethereum.stackexchange.com/questions/25335/dapp-storage-for-data-other-than-tranactions, (Accessed on 08/19/2018), 2017.

[107] E. F. Codd, 'Extending the database relational model to capture more meaning', *ACM Transactions on Database Systems (TODS)*, vol. 4, no. 4, pp. 397–434, 1979.

[108] *Design patterns*, https://refactoring.guru/design-patterns, (Accessed on 08/20/2018).

[109] *Design patterns*, https://www.oodesign.com/, (Accessed on 08/20/2018).

[110] *Factory pattern*, https://www.oodesign.com/factory-pattern.html, (Accessed on 08/20/2018).

[111] *Object pool*, https://www.oodesign.com/object-pool-pattern.html, (Accessed on 08/20/2018).

[112] *Command pattern*, https://www.oodesign.com/command-pattern.html, (Accessed on 08/20/2018).

[113] T. Point, *Design patterns - adapter pattern*, https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm, (Accessed on 08/27/2018), 2018.

[114] S. W. Ambler, *Uml 2 state machine diagrams: An agile introduction*, 2018. [Online]. Available: http://www.agilemodeling.com/artifacts/stateMachineDiagram.htm.

[115] yprateek, *What is meant by smart contract state? where do the data get stored?*, Jun. 2017. [Online]. Available: https://ethereum.stackexchange.com/questions/17456/what-does-it-mean-by-smart-contract-state-where-they-get-stored.

[116] Jul. 2018. [Online]. Available: http://solidity.readthedocs.io/en/v0.4.24/.

[117] pabloruiz55, *The structure of a dapp and contracts compared to a traditional web platform*, https://ethereum.stackexchange.com/questions/28792/the-structure-of-a-dapp-and-contracts-compared-to-a-traditional-web-platform/28797#28797, (Accessed on 08/24/2018), Oct. 2017.

[118] (Oct. 2017). Serpent, [Online]. Available: https://github.com/ethereum/serpent.

[119] B. Edgington. (2017). Lll, [Online]. Available: https://lll-docs.readthedocs.io/en/latest/lll_introduction.html.

[120] (Aug. 2018). Vyper - new experimental programming language, [Online]. Available: https://github.com/ethereum/vyper.

[121] *Types: Mappings*, https://solidity.readthedocs.io/en/v0.4.24/types.html#mappings, (Accessed on 08/20/2018).

[122] OpenZeppelin. (Jun. 2018). Ownable.sol, [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/ownership/Ownable.sol.

[123] *Oraclize - blockchain oracle service, enabling data-rich smart contracts*, http://www.oraclize.it/, (Accessed on 08/17/2018), 2016.

[124] Aug. 2018. [Online]. Available: http://docs.oraclize.it/#ethereum-quick-start-authenticity-proofs.

[125] Oraclize. (Jun. 2018). Oraclize api for ethereum smart contracts, [Online]. Available: https://github.com/oraclize/ethereum-api.

[126] Aug. 2018. [Online]. Available: https://solidity.readthedocs.io/en/develop/types.html#fixed-point-numbers.

[127] D. Tools. (Sep. 2018). Epoch unix time stamp converter, [Online]. Available: https://www.unixtimestamp.com/.

[128] E. community. (2016). Ether - denominations, [Online]. Available: http://ethdocs.org/en/latest/ether.html%5C#denominations.

[129] chriseth, *In ethereum solidity, what is the purpose of the "memory" keyword?*, https://stackoverflow.com/questions/33839154/in-ethereum-solidity-what-is-the-purpose-of-the-memory-keyword, (Accessed on 08/21/2018).

[130] axic and yann300. (Apr. 2018). Display reason for revert (solidity 0.4.22 feature), [Online]. Available: https://github.com/ethereum/remix/pull/760.

[131] *Contracts - getter functions*, https://solidity.readthedocs.io/en/v0.4.24/contracts.html#getter-functions, (Accessed on 08/24/2018), 2018.

[132] *Solidity visibility and getter functions*, https://www.bitdegree.org/learn/solidity-visibility-and-getters/, (Accessed on 08/24/2018), 2018.

[133] Jul. 2018. [Online]. Available: https://truffleframework.com/ganache.

[134] O. Limited. (Jul. 2018). Independent bridge to link any ethereum network with the oraclize engine, [Online]. Available: https://github.com/oraclize/ethereum-bridge.

[135] Mozilla. (Jul. 2018). Console.time(), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Console/time.

[136] R. Zafar. (Mar. 2012). What is software testing? what are the different types of testing?, [Online]. Available: https://www.codeproject.com/Tips/351122/What-is-software-testing-What-are-the-different-ty.

[137] D. R. Wallace and R. U. Fujii, 'Software verification and validation: An overview', *Ieee Software*, vol. 6, no. 3, pp. 10–17, 1989.

[138] L. Copeland, *A practitioner's guide to software test design*. Artech House, 2004.

[139] P. Labs. (Aug. 2018). Jasmine – behavior-driven javascript, [Online]. Available: https://jasmine.github.io/.

[140] G. Testing. (Aug. 2018). Karma, [Online]. Available: https://karma-runner.github.io/2.0/index.html.

[141] Google. (Aug. 2018). Testing, [Online]. Available: https://angular.io/guide/testing.

[142] eth. (Jul. 2016). Are contract calls asynchronous in web3.js?, [Online]. Available: https://ethereum.stackexchange.com/questions/7327/are-contract-calls-asynchronous-in-web3-js.

[143] M. Scherer. (Sep. 2016). Async function return type, [Online]. Available: https://github.com/Microsoft/TypeScript/issues/11097.

[144] A. Zhou. (Oct. 2017). Unit testing async calls and promises with jasmine, [Online]. Available: https://medium.com/dailyjs/unit-testing-async-calls-and-promises-with-jasmine-a20a5d7f051e.

[145] P. Singh. (Sep. 2017). Automate e2e testing of angular 4 apps with protractorjs & jasmine, [Online]. Available: https://medium.com/paramsingh-66174/automate-e2e-testing-of-angular-4-apps-with-protractorjs-jasmine-fcf1dd9524d5.

[146] (Aug. 2018). Protractor - end to end testing for angular, [Online]. Available: https://www.protractortest.org/.

[147] Sep. 2017. [Online]. Available: https://medium.com/@davekaj/how-hard-is-it-to-become-a-smart-contract-developer-f159bafd8018.

[148] G. Wood, *Gavin wood on how $60m hack of dao happened and what to do next*, https://www.youtube.com/watch?v=JzCGRtGyxvY, Accessed on 02/08/2018.

[149] K. Karagiannis, *Def con 25 - - hacking smart contracts*, https://www.youtube.com/watch?v=WIEessi3ntk, (Accessed on 08/23/2018), Oct. 2017.

[150] raineorshine, *Visualize solidity control flow for smart contract security analysis.* https://github.com/raineorshine/solgraph, Aug. 2018.

[151] ConsenSys. (Jul. 2018). Solidity parser in javascript, [Online]. Available: https://github.com/ConsenSys/solidity-parser.

[152] M. Suiche. (Jul. 2017). Def con 25: Porosity, [Online]. Available: https://blog.comae.io/porosity-18790ee42827.

[153] J. C. King, 'Symbolic execution and program testing', in *Communications of the ACM*, B. Wegbreit, Ed., vol. 19, IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, N.Y. 10598: ACM, Jul. 1976, pp. 385–394. [Online]. Available: http://www.cs.umd.edu/class/fall2014/cmsc631/papers/king-symbolic-execution.pdf.

[154] M. AG. (Jul. 2018). An analysis tool for smart contracts, [Online]. Available: https://github.com/melonproject/oyente.

[155] L. Luu, D.-H. Chu, H. Olickel, P. Saxena and A. Hobor, 'Making smart contracts smarter', in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Vienna, Austria: ACM, 2016, pp. 254–269, ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978309. [Online]. Available: https://www.comp.nus.edu.sg/~loiluu/papers/oyente.pdf.

[156] P. Technologies. (Aug. 2018). Parity, [Online]. Available: https://www.parity.io/.

[157] S. Petrov. (Nov. 2017). Another parity wallet hack explained, [Online]. Available: https://medium.com/@Pr0Ger/another-parity-wallet-hack-explained-847ca46a2e1c.

[158] P. Daian. (Jun. 2016). Another parity wallet hack explained, [Online]. Available: http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

[159] (Aug. 2018). The dao (organization), [Online]. Available: https://en.wikipedia.org/wiki/The_DAO_(organization).

[160] M. AG. (Aug. 2018). Melonport - asset management computer, [Online]. Available: https://melonport.com/.

[161] Ethereum. (Aug. 2018). Types - value types - integers, [Online]. Available: https://solidity.readthedocs.io/en/v0.4.24/types.html%5C#integers.

[162] L. Y. Thanh, *Prevent integer overflow in ethereum smart contracts*, https://medium.com/@yenthanh/prevent-integer-overflow-in-ethereum-smart-contracts-a7c84c30de66, (Accessed on 08/25/2018), 2018.

[163] kasceled and goodvibration, *Adding a modifier to an auto-generated variable getter*, https://ethereum.stackexchange.com/questions/46877/adding-a-modifier-to-an-auto-generated-variable-getter, (Accessed on 08/25/2018), Apr. 2018.

[164] Ethereum, *Types - dynamically-sized byte array*, https://solidity.readthedocs.io/en/v0.4.24/types.html#dynamically-sized-byte-array, (Accessed on 08/25/2018), 2018.

[165] *Known attacks - race conditions*, Mar. 2018. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/known_attacks/.

[166] Aug. 2018. [Online]. Available: https://ethereum.stackexchange.com/questions/1895/can-a-contract-listen-to-events-of-another-contract.

[167] A. V. Inc. (Jul. 2018). Alpha vantage, [Online]. Available: https://www.alphavantage.co/.

[168] (Jun. 2018). Chart.js, [Online]. Available: http://www.chartjs.org/docs/latest/.

[169] Google, 2018. [Online]. Available: https://angular.io/tutorial/toh-pt4.

[170] ——, 2018. [Online]. Available: https://angular.io/guide/dependency-injection-pattern.

[171] ——, 2018. [Online]. Available: https://angular.io/guide/dependency-injection.

[172] O. Hart, *Nobel prize-winning economist on smart contracts*, https://www.facebook.com/businessinsider/videos/10155882786044071/UzpfSTIwNDQ2MjU0MDcwOlZLOjEwMTU1ODgyNzg2MDQ0MDcx/, (Accessed on 07/22/2018), 2018.

[173] *Blockchain: What is it and what is it for?*, https://www.forbes.com/sites/forbestechcouncil/2018/03/28/blockchain-what-is-it-and-what-is-it-for/, (Accessed on 07/26/2018), 2018.

[174] *Smart contract*, https://en.wikipedia.org/wiki/Smart_contract, (Accessed on 07/26/2018), 2018.

[175] Ethereum.org. (Jun. 2018). Remix - solidity ide, [Online]. Available: https://remix.ethereum.org/.

[176] Google. (Jun. 2018). Angular, [Online]. Available: https://angular.io/.

[177] E. Foundation. (Aug. 2018). Json rpc, [Online]. Available: https://github.com/ethereum/wiki/wiki/JSON-RPC.

[178] StrongLoop, IBM and expressjs.com. (Jun. 2018). Express - fast, unopinionated, minimalist web framework for node.js, [Online]. Available: https://expressjs.com/.

[179] N. Foundation. (Jun. 2018). Node.js, [Online]. Available: https://nodejs.org/en/.

[180] ConsenSys. (Jun. 2018). Truffle suite - sweet tools for smart contracts, [Online]. Available: https://truffleframework.com/.

[181] (Jun. 2018). Metamask - brings ethereum to your browser, [Online]. Available: https://metamask.io/.

[182] T. go-ethereum Authors. (Dec. 2017). Go ethereum - official go implementation of the ethereum protocol, [Online]. Available: https://github.com/ethereum/go-ethereum/wiki/geth.

[183] Jul. 2018. [Online]. Available: https://github.com/ethereum/mist.

[184] ——, (Nov. 2017). Management apis - personal, [Online]. Available: https://github.com/ethereum/go-ethereum/wiki/Management-APIs%5C#personal.

[185] *Can exposing personal be made safe anyhow?*, Dec. 2016. [Online]. Available: https://ethereum.stackexchange.com/questions/10637/can-exposing-personal-be-made-safe-anyhow.

[186] *Ethereum consortium template- unable to unlock accounts, private api is not exposed*, Aug. 2017. [Online]. Available: https://github.com/Azure/azure-quickstart-templates/issues/3784.

[187] 0mkara. (Aug. 2018). Etheratom - solidity compilation and ethereum contract execution interface for hackable atom editor., [Online]. Available: https://atom.io/packages/etheratom.

[188] caktux. (Jul. 2016). Language-ethereum - ethereum language support in atom, [Online]. Available: https://atom.io/packages/language-ethereum.

[189] Jul. 2018. [Online]. Available: https://material.io/design/.

[190] Jul. 2018. [Online]. Available: https://material.angular.io/.

[191] I. npm. (Jun. 2018). Npm, [Online]. Available: https://www.npmjs.com/.

[192] Google. (Jun. 2018). Angular cli - a command line interface for angular, [Online]. Available: https://cli.angular.io/.

[193] ——, (Jun. 2018). Deployment, [Online]. Available: https://angular.io/guide/deployment.

[194] Microsoft. (Jun. 2018). Web apps, [Online]. Available: https://azure.microsoft.com/en-gb/services/app-service/web/.

[195] E. Foundation. (Aug. 2018). Ethereum development tutorial, [Online]. Available: https://github.com/ethereum/wiki/wiki/Ethereum-Development-Tutorial.

[196] Jul. 2018. [Online]. Available: http://solidity.readthedocs.io/en/v0.4.21/using-the-compiler.html#output-description.

[197] Ethereum. (Jul. 2018). Contract abi specification, [Online]. Available: https://solidity.readthedocs.io/en/develop/abi-spec.html.

[198] Jul. 2018. [Online]. Available: https://en.wikipedia.org/wiki/Application_binary_interface.