



DEPARTMENT OF COMPUTING

BENG, JOINT MATHEMATICS AND COMPUTER SCIENCE

C301J - FINAL YEAR INDIVIDUAL PROJECT

---

## Digipound: A Proof-of-Concept Stablecoin Audited in Real Time

---

*Author:*

Shravan Nageswaran  
sn2316  
CID: 01193565

*Supervisor:*

Prof. William Knottenbelt

*Second Marker:*

Prof. Kin Leung

June 19, 2019

## Abstract

The twenty-first century calls for an innovative form of money. Cryptocurrencies traded on blockchains offer a glimpse into the future of currency, but their immense volatility makes them a risky choice to invest in both the short and long terms. A cryptocurrency with a defined value can address these concerns. Stablecoins, such as Tether, are cryptocurrencies whose values are tied to other assets - namely, fiat currencies such as the US Dollar.

Yet, without a regularly-updating auditing system in place to track both the amount of Tether issued and the amount of US Dollars held in the Tether reserve account, the value of Tether, and other stablecoins, can be compromised. Attacks on its smart contract can go unnoticed, and if discovered that it truly does not have a one-to-one backing by the US Dollar, the Tether coin and its three billion dollar market cap can crash just as easily as Bitcoin did in 2018.

This project started out by investigating several types of stablecoins and other assets to determine that there is a need for a stablecoin with a regularly-updating audit. Its main outcome was a model of a functional stablecoin, complete with the ability to issue, trade, and redeem this digital coin, with an audit updating in real time.

First, a standard smart contract for an ERC-20 token, Digipound, was written and deployed to a test Ethereum blockchain. Next, three main functions were created to implement each of the stages of the issue-trade-redeem cycle of a stablecoin. In the issue stage, payments of GBP are accepted using the Stripe platform, triggering a call to an API that is connected to the Ethereum blockchain, web3, to issue the aforementioned token. Digipound tokens can be traded through both the project or independently. In the redeem stage, a call to the web3 API writes a transaction to burn the Digipound tokens before calling the Stripe API to issue a payout of the appropriate amount of pounds back to the user. These functions were transformed into a user experience via the Digipound website, which also kept track of the balance of GBP in a reserve account connected to Stripe as well as a link to a block explorer, a blockchain search engine, to verify the amount of the Digipound tokens that were issued. These quantities were tracked in real time and displayed in a clear manner on the web application's landing page, representing the auditing system.

Overall, this project resulted in a proof-of-concept for a stablecoin whose value is explicit and can be easily verified, Digipound. It is currently a Minimum Viable Product for an application to trade a real stablecoin token on the Ethereum blockchain, having the potential to serve as a widely-used and secure platform for digital transactions.



---

## Acknowledgments

This project marks the culmination of an enriching and challenging three years at Imperial College London. Throughout my university degree, and ending with my final project, there have been numerous individuals and groups I would like to recognise that have contributed to my university experience:

- My project supervisor, Professor William Knottenbelt, for his endless support, insight, and - most importantly - enthusiasm throughout this year.
- My personal tutor, Professor Christopher Hankin, for his guidance and support throughout my Imperial education.
- Dr. Robert Chatley, whose course, Software Engineering: Design, taught me many principles for good design of the Digipound website.
- My entire family, for their unrelenting support, particularly:
  - My mother, Vidhya, and father, Parthasarathy, for showing me by example what it means to work hard, and my brother, Shreyan.
  - G. Natarajan, to whom this project is dedicated to, for teaching me that hard work can be accompanied by a smile.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Objectives . . . . .	1
1.3	Contributions . . . . .	2
1.4	Organisation . . . . .	2
<b>2</b>	<b>Background</b>	<b>4</b>
2.1	Cryptocurrency . . . . .	4
2.1.1	Examples . . . . .	4
2.1.2	Volatility . . . . .	5
2.1.3	Blockchain . . . . .	6
2.1.4	Smart Contracts . . . . .	8
2.1.5	Storage . . . . .	10
2.2	Stablecoins . . . . .	10
2.2.1	Collateralisation . . . . .	11
2.2.2	Examples . . . . .	11
2.2.3	Reserve Accounts . . . . .	12
2.3	Tokenisation . . . . .	13
2.3.1	Examples . . . . .	13
2.3.2	Issue-Trade-Redeem . . . . .	15
2.4	Summary . . . . .	17
2.5	Vision . . . . .	17
<b>3</b>	<b>Design</b>	<b>18</b>
3.1	Components . . . . .	18
3.1.1	Digipound . . . . .	18
3.1.2	web3 . . . . .	21
3.1.3	Stripe . . . . .	25
3.1.4	Audit . . . . .	30
<b>4</b>	<b>Implementation</b>	<b>34</b>
4.1	MVC . . . . .	35
4.2	Laravel . . . . .	37

4.2.1 Issue-Trade-Redeem . . . . .	37
4.2.2 Signing Transactions . . . . .	42
4.2.3 Audit . . . . .	43
4.3 Ethereum Node . . . . .	45
<b>5 Security</b>	<b>47</b>
5.1 Block Reorganisation Attacks . . . . .	47
5.2 Private Keys . . . . .	51
5.3 Breach of Smart Contracts . . . . .	52
5.4 Losing Keys . . . . .	52
<b>6 Testing and Edge Cases</b>	<b>54</b>
6.1 Issue . . . . .	54
6.2 Trade . . . . .	59
6.3 Redeem . . . . .	60
<b>7 Evaluation</b>	<b>64</b>
7.1 Cryptocurrency . . . . .	64
7.1.1 Cycle . . . . .	64
7.1.2 Valuation and Audit . . . . .	65
7.2 Application . . . . .	67
7.2.1 Website . . . . .	67
<b>8 Conclusion</b>	<b>71</b>
8.1 Takeaways and Lessons Learned . . . . .	71
8.2 Future Implementation . . . . .	72
8.2.1 Live Application . . . . .	72
8.2.2 Acceptance Among Merchants . . . . .	74
8.2.3 Payment Channel Network . . . . .	75
<b>A Additional Functions</b>	<b>77</b>
<b>B User Guide</b>	<b>80</b>

# **Chapter 1**

## **Introduction**

### **1.1 Motivation**

Bitcoin, a peer-to-peer payment network, is considered to have revolutionized currency. It is the first currency to be transferred online, and is considered much safer than traditional money due to a network called blockchain. This manages forms of cryptocurrency by securely recording every one of its transactions.

Yet, a major issue with Bitcoin, and other cryptocurrencies, is volatility. Its rapid changes in value cause doubts of its ability to serve as a currency.

A possible solution to avoid the volatility of cryptocurrencies is the use of stablecoins. Stablecoins are cryptocurrencies as well. However, their values are tied to the value of other assets, namely fiat currencies. For example, Tether is the stablecoin with the highest market cap, and its value is tied to that of the US Dollar.

The current problem that exists with stablecoins is the ambiguity of its tokens actually being backed by real units of currency. While tying a value to a tangible asset is a safe concept in practice, the value of a stablecoin is compromised without a concise and regularly-updating audit system in place. For example, even the Gemini Dollar, the first regulated stablecoin, only releases examinations on its holdings once per month. Without regulation in real time, stablecoins face a potential consequence of under-collateralization. This will not help assuage investors' concerns on volatility or safety.

### **1.2 Objectives**

A potential solution is the creation of a stablecoin which has real time auditing. In this scenario, a token is only sold and its transaction on the blockchain is only cre-

ated when the adequate funds have been received, and a refund of such payment is made only when the tokens have been burned. Moreover, both the amount of tokens in circulation and the corresponding reserve of fiat currency are tracked in real time.

The objectives of this project were to design and implement a new stablecoin, whose value would be tied to that of the GBP, and connect it to an audit to ensure its true value would never be compromised. A system to issue stablecoin tokens for fiat currency, provide a medium to trade these tokens, and then allow for them to be redeemed back into fiat currency, combined with an audit updating in real time, was a desired outcome of the project.

## 1.3 Contributions

Setting up this system, in the form of a web application, required both a connection to a payment API, Stripe, and a connection to a blockchain-based platform, Ethereum. By deploying a smart contract for an ERC-20 token, the stablecoin token itself was created. Integrating the system with the Stripe API allowed for creation of a payment method within the application so users can purchase tokens online, leading to the following flow. Once the money is received, a transaction is created on a blockchain, by calling another API, web3. Then, the corresponding amount of ‘digital pounds’ are issued to the user. The ensuing update of the audit shows an equal amount of fiat pounds and digital pounds in circulation. These digital pounds are able to be traded between users, through a local node connected to the Ethereum blockchain, as well. Finally, in a similar manner, these stablecoin tokens can be redeemed for fiat currency at any time.

The main result of this project was a proof-of-concept of a stablecoin that is both accurately backed by real currency and transparent and secure for investors. It can ultimately serve as an effective decentralized management of digital currency - a ‘digital pound’, or Digipound.

## 1.4 Organisation

This report is organised into the following chapters:

- **Background:** I identify the need for such a stablecoin by comparing a variety of current assets, from cryptocurrencies to wrapped currencies.
- **Design:** At a high level, I analyse the components of implementing a stablecoin trading service within a web application framework. I also delve into

the creation and deployment of the Digipound token on the Ethereum (test) blockchain.

- **Implementation:** I detail the specifics of coding the Digipound web application, from issuing, trading, and redeeming the Digipound token to updating and displaying the realtime audit. I explain this in the context of the Model-View-Controller architecture present in the Laravel web framework. I identify and describe prerequisites for running a blockchain-based application locally.
- **Security:** I identify potential security concerns that are present with a stable-coin of this potential and explain how I mitigate these.
- **Testing and Edge Cases:** I detail the completeness of my application by running a series of tests on all of its flows and components and explain how I respond to potential edge cases and potential system mishaps.
- **Evaluation:** I critically analyse my project against my initial objectives of it, as both a new cryptocurrency and a new web application.
- **Conclusion:** I describe the takeaways from my project. I present my vision for Digipound as the future of decentralised assets and detail how to build upon my current work to achieve such a vision.

# **Chapter 2**

## **Background**

### **2.1 Cryptocurrency**

Cryptocurrencies have revolutionised the way money has been perceived. As digital assets, they are essentially a medium of exchange. They have recently gained much popularity. For example, the world's top-trading cryptocurrency, Bitcoin, has a market cap of nearly one-hundred and fifty billion dollars [1].

Their benefits lie in their security, as cryptography secures financial transactions, relative anonymity, and decentralisation. To elaborate on the latter, a cryptocurrency records all financial transactions on a distributed and decentralised ledger called a blockchain. Therefore, one can send cryptocurrencies directly to another user without the use of a third party, such as a centralised banking system.

#### **2.1.1 Examples**

As stated above, the most widely used cryptocurrency is Bitcoin. However, there are many other types of cryptocurrencies, whose implementations differ in multiple ways. The main differences between cryptocurrencies are in their methods of issue and places to spend them.

For example, Bitcoin and Dash, among others, are issued in a process called ‘mining.’ Mining a block is the process of adding transaction records to a blockchain. It is performed whenever a new coin has been issued. A hash is created for the newly-created block, with requirements of it being unique and hard to find arbitrarily. Finding the hash is essentially a cryptographic puzzle, and the system gives coins to the ‘miner’ that can solve the puzzle [3].

Cryptocurrencies are traded on platforms, and can be versatile in the way they are exchanged. To elaborate, one intent to invest on the Ripple blockchain is to transfer

money between countries. Users can convert their fiat currency (foreign) to currency on the Ripple platform, and after requesting a transaction on the Ripple blockchain, the recipient of the transaction can convert these new coins into whichever currency desired [5].

The markets and purposes of spending differs among cryptocurrencies, as well. For example, Bitcoin is the most widely accepted cryptocurrency among merchants. It is not uncommon for some retailers to accept purchases in Bitcoin, including Microsoft.

The screenshot shows a Microsoft webpage with a blue header bar containing links for 'Microsoft account support', 'Manage account & billing', 'Community', and 'Contact us'. Below the header, the main content area has a title 'How to use Bitcoin to add money to your Microsoft account' and a subtitle 'Applies to: Microsoft Account, Microsoft Store'. The main text reads: 'Use Bitcoin to add money to your Microsoft account.' followed by a numbered list of steps:

1. Go to [account.microsoft.com](https://account.microsoft.com) and sign in to your Microsoft account.
2. Under **Payment & billing**, select **Payment options**.
3. Select **Redeem bitcoin**, select the amount you want to add, then select **Next**.
4. Review the amount of Bitcoin needed and use your digital wallet to complete the transaction within 15 minutes.
  - On your PC: select **Open in wallet** and pay from your Bitcoin wallet on the same device.
  - On your phone: scan the QR code on the page to pay from your mobile wallet app.
  - If your wallet is on another device, you can copy the receiving address and BTC amount to the wallet on your device or the web and then make your purchase.

**Figure 2.1:** Microsoft allows customers to pay for services, such as games, apps, and movies, using Bitcoin [6].

Some cryptocurrencies, however, have a very niche use. In particular, Ether is a currency that is only used when one requests a service on the Ethereum platform. Ethereum is a blockchain-based platform, as will be elaborated below, and when one makes a transaction on it or uses an application on it, Ether is the currency which is paid to facilitate these functions. Overall, cryptocurrencies have gained immense popularity and a wide variety of uses.

### 2.1.2 Volatility

High volatility is currently one of the biggest problems that cryptocurrency faces. For example, Bitcoin's historic run in 2017 saw it become one of the most popular cryptocurrencies [4]. With sudden ascensions come crashes as well. In early 2018, Bitcoin's price fell by as much as 65 percent, and the cryptocurrency market suffered a 342 billion (US) dollar loss [15].

A simple reason why cryptocurrencies are so volatile is the fact that they have no intrinsic value. As they do not sell anything tangible, or rarely invest their profits



**Figure 2.2:** The history of Bitcoin's price highlights its volatility [4].

back into the cryptocurrency market, it is hard to place a value on many of these online currencies. Additionally, banks and hedge funds have yet to commit to cryptocurrency as a viable investment, which means that cryptocurrency faces a lack of institutional capital [14].

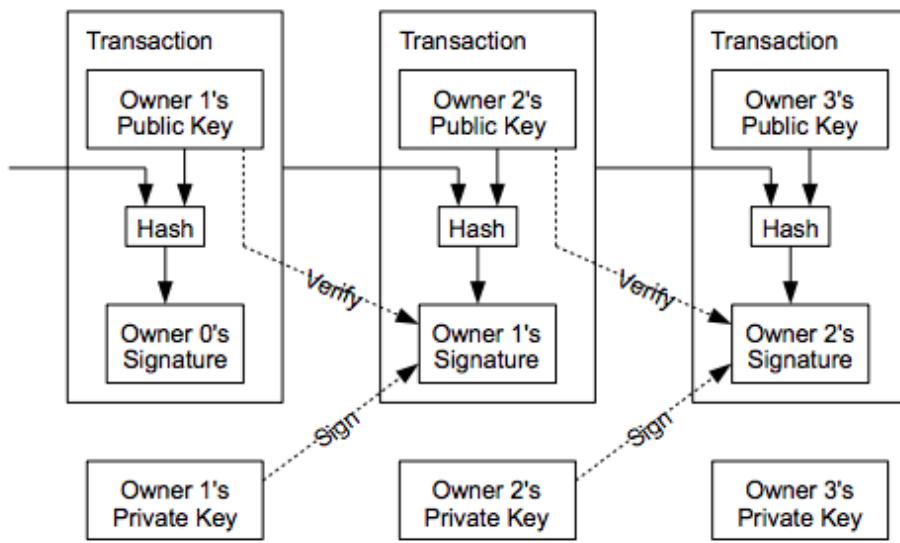
The reason for this is that, at its source, a cryptocurrency is just that - a currency. In most cases, its value is only determined by the cost people are willing to forfeit for it. Perhaps, with more regulation, diverse investors, and a defined value, the cryptocurrency market may decrease in volatility, but this seems rather unlikely in the short term.

### 2.1.3 Blockchain

Blockchain, a decentralized ledger, is important to cryptocurrency transactions as it eliminates the need for a centralized third party to validate digital peer-to-peer exchanges. An electronic coin is often defined as a chain of digital signatures [2]. After each transaction of an electronic coin, payees can verify their signatures to the recipients of the transactions to continue the chain of ownership.

For reference, a blockchain can be set up as a linked list or an array in most programming languages, so long as each block has at least the following essential attributes:

- The **hash** of the previous block.
- The **current hash** of the block.
- The **timestamp** when the block was created.
- The **nonce**, whose function is detailed below.



**Figure 2.3:** When transferring a Bitcoin, for example, the current owner digitally signs a hash of the previous transaction and specifies the public key of the next owner. The chain is then extended to reflect the transaction [2].

- The list of **transactions** represented by the block.

In order to ensure that coins cannot be spent multiple times, the hash of each block is made publicly accessible, along with a timestamp corresponding to the respective transaction. Every transaction in the chain has a timestamp which reinforces the order of transactions and shows that the data was present in order to create the hash of the following transaction. To setup a block in order to add a transaction to it, Bitcoin increments its *nonce* until it finds a valid hash for that block [2].

There are a few established blockchain-based platforms already in place. Blockchain implementations can be broken into two types. There are enterprise blockchains such as Hyperledger where blocks have to be authorised before entering the network. The other alternative, which this project is concerned with, is open public blockchains, such as Bitcoin and Ethereum [7].

Ethereum is one of the most popular blockchains, and is particularly useful in its ability to incorporate a scripting language, Solidity, to create applications and new currencies on a shared blockchain. The Ethereum platform acts as a medium to deploy *smart contracts*, which can be written in Solidity.

These smart contracts can reflect any kind of transaction, and they can be easily accessed on the Ethereum platform. The platform also has its own currency,

as well, called Ether. As mentioned before, Ether is the fundamental token of Ethereum which facilitates transactions of all tokens and applications operating on the Ethereum network.

An additional benefit of the Ethereum network, in context of application or currency developers, is the presence of numerous test networks, such as the Ropsten test network. These give developers a testing environment to create their applications and make transactions without spending real currency such as Ether, before they bring their applications to the main network. The Ropsten test network was used to develop a proof-of-concept for the objectives of this project.

#### **2.1.4 Smart Contracts**

As stated, one method to deploy a cryptocurrency is the creation of a smart contract on a Blockchain-based platform, such as Ethereum. The smart contract represents a token, which can represent any tradable, tangible product.

In essence, most smart contracts on a particular blockchain have identical attributes in their codebases. The code, itself, is not unique, rather it is what the token represents that distinguishes it. For example, in the case of a stablecoin traded on the Ethereum network, one token can represent a unit of fiat currency. Main functions that a smart contract must include are a constructor for the token and also a function that details the transfer of a token from one address to another. Each address represents a user that has holdings on a blockchain.

The screenshot shows the Etherscan interface for the WBTC token. At the top, there's a navigation bar with links for Home, Blockchain, Tokens, Resources, More, Sign In, and a search bar. Below the header, the token details are shown:

- Overview [ERC-20]**
- PRICE**: \$8,700.5365 @ 32.433223 Eth (+ 2.20%)
- FULLY DILUTED MARKET CAP**: \$2,251,763.17
- Total Supply**: 258,807,9236 WBTC
- Holders**: 586 addresses
- Transfers**: 11,976

On the right, the **Profile Summary** includes:

- Contract: 0x2260fac5e5542a773aa44fbcef7c193bc2c599
- Decimals: 8
- Official Site: <https://www.wbtc.network/>
- Social Profiles: (links to email, website, and Twitter)

Below the overview, there are tabs for Transfers, Holders, Info, Exchange, DEX Trades, Read Contract, Write Contract, Analytics, and Comments. The **Transfers** tab is selected, showing a table of recent transactions:

Txn Hash	Age	From	To	Quantity
0x4ca05e637a119c...	2 mins ago	0x9ae49c0d7f8f9ef4...	0x63825c174ab367...	0.059509
0x4ca05e637a119c...	2 mins ago	0x381758cbe8e8dc...	0x9ae49c0d7f8f9ef4...	0.059509
0x5351fe6cdd05ddf...	2 hrs 10 mins ago	0x9ae49c0d7f8f9ef4...	0x63825c174ab367...	0.01267592

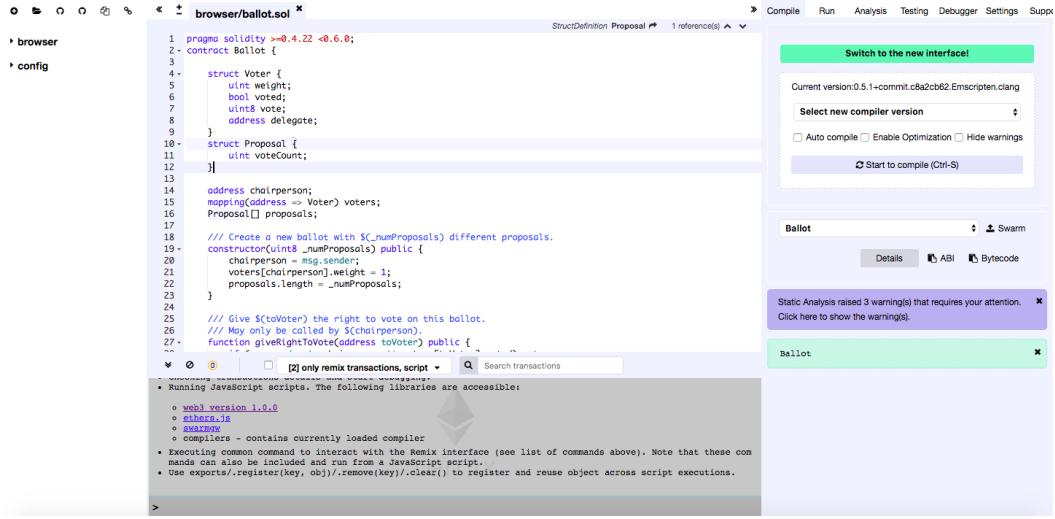
**Figure 2.4:** WBTC is a token deployed on the Ethereum blockchain that serves as a wrapper for Bitcoin, which will be explained below. Its contract address, token holders, and list of previous transactions are displayed using Etherscan, which is essentially a search engine, or block explorer, for the Ethereum network [8].

Contracts can be composed according to a specific technical standard. In the Ethereum blockchain, the most popular standard is ERC-20, which defines a shared set of rules for Ethereum tokens to use. As of May 2019, there are over one hundred ninety thousand ERC-20 compatible tokens on the main Ethereum network [8].

Functions that tokens following the ERC-20 standard must have include [10]:

- `totalSupply()`, used to access the total supply of the token,
- `balanceOf(address tokenOwner)`, which shows the balance of an account holder of that token,
- `transferFrom(address from, address to, uint256 value)`, which is especially essential for a cryptocurrency.

After composing a contract, the next step is to deploy it. In the case of deploying it on the Ethereum network or its subsidiary test networks, one such IDE that can be used is called Remix. Remix runs a local Ethereum node to interact with its blockchain and can compile and deploy the contract for any application.



**Figure 2.5:** The Remix IDE can be used to deploy smart contracts to create cryptocurrencies or any applications on the Ethereum blockchain - in this case, a default application to resemble a voting ballot [11].

Once deployed, a contract is assigned a hexadecimal address and transactions such as issuing tokens can be written to the blockchain by specifying the address of the contract of the token to be transferred or the application to be used. These are the essential steps needed to create a basic cryptocurrency that can be transferred between users.

### 2.1.5 Storage

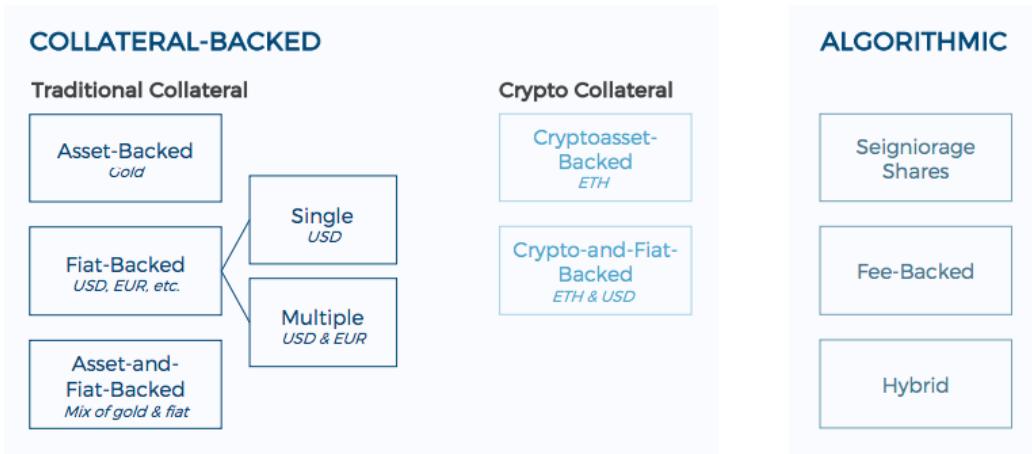
The mechanism where users can store and access their cryptocurrency holdings is a wallet. Users have access to a public key and a private key; the public key is hashed and the result makes up a hexadecimal address where cryptocurrency can be sent to [12]. The private key is what users require to sign each transaction, to ensure additional security. An example of a wallet for cryptocurrencies on the main Ethereum network and its test networks is MyEtherWallet.

## 2.2 Stablecoins

According to the *2018 State of Stablecoins* report, there are fifty-seven stablecoins, of which thirty-nine percent are live [13]. Stablecoins pose a natural solution to the volatility of cryptocurrency as their values are always backed by some form of asset. Stablecoins can be traded on the same platforms as other cryptocurrencies. The most popular platform on which stablecoins are traded is Ethereum, where fourteen stablecoins are traded [13].

### 2.2.1 Collateralisation

Stablecoins are very versatile in terms of the asset their values are tied to. For example, stablecoins can be backed by collateral or algorithms which determine their worth. Collateral-backed stablecoins can be valued by traditional collateral or cryptocurrency. Traditional collateral can range from fiat currency to gold.



**Figure 2.6:** Stablecoins can be backed by many entities [13].

This project is concerned with traditional collateral-backed stablecoins. These particular stablecoins have recently gained much popularity, and their relation to fiat currencies results in them being called ‘digital currencies.’ In fact, the head of the International Monetary Fund, Christine Lagarde, suggested that a change to digital currencies was imminent for central banks of several countries, namely Canada, China, Sweden, and Uruguay [16].

### 2.2.2 Examples

By market cap, Tether is the most popular stablecoin, trading at over three billion US Dollars [1]. It had been originally claimed that each Tether coin was backed by one US dollar, although that claim seems to be rather tenuous. To counter this uncertainty, Cameron Winklevoss released the Gemini Dollar, which claims to be the world’s first regulated stablecoin that can verify its holding claims [17].

Other examples of stablecoins include the MakerDAI stablecoin, which is actually backed by the Ether cryptocurrency. Though crypto-based stablecoins are outside the scope of this project, it is interesting to note that MakerDAI aims to stabilise the

#	Name	Market Cap	Price	Volume (24h)	Circulating Supply	Change (24h)	Price Graph (7d)	...
1	Bitcoin	\$154,572,646,234	\$8,714.77	\$22,134,811,634	17,736,862 BTC	2.10%		...
2	Ethereum	\$28,668,043,543	\$269.63	\$9,681,569,688	106,324,038 ETH	1.16%		...
3	XRP	\$18,556,634,650	\$0.439918	\$1,701,925,154	42,181,995,112 XRP *	2.14%		...
4	Bitcoin Cash	\$7,895,493,211	\$443.17	\$1,824,584,329	17,815,950 BCH	1.19%		...
5	EOS	\$7,126,975,228	\$7.77	\$5,203,619,819	917,451,190 EOS *	-5.32%		...
6	Litecoin	\$7,095,746,332	\$114.36	\$4,181,385,344	62,046,301 LTC	1.28%		...
7	Binance Coin	\$4,681,182,051	\$33.16	\$492,747,310	141,175,490 BNB *	0.38%		...
8	Bitcoin SV	\$3,402,778,017	\$191.02	\$426,830,109	17,813,761 BSV	0.59%		...
9	Tether	\$3,146,936,538	\$1.00	\$21,782,924,785	3,131,993,375 USDT *	0.18%		...

**Figure 2.7:** The stablecoin with the highest market cap, Tether, has the ninth highest market cap amongst all other cryptocurrencies, as of June 2019 [1].

value of the Dai cryptocurrency by, through using smart contract technology, issuing users Dai in exchange for Ethereum. This is essentially a loan of Dai, and the idea is that if the value of Ethereum decreases, MakerDAI will be returned. Surprisingly, it has relatively low short-term volatility, but naturally, given that it is tied to a cryptocurrency, long term volatility is a drawback [18].

### 2.2.3 Reserve Accounts

Most stablecoins go hand-in-hand with a reserve account, which supposedly holds the appropriate amount of collateral relative to the number of stablecoins issued. In principle, if every holder of a particular stablecoin wanted to sell, or redeem, their stablecoin holdings for exactly what they paid for, this would be feasible.

However, there is often much doubt among stablecoin investors that their digital currencies are appropriately backed up by fiat currency. There have been instances where stablecoins are under-collateralized - or, in the odd case of the USD Coin by Circle, over-collateralized [19]. Recently, much doubt was cast on Tether's ability to be fully redeemed by the US dollar, as investors do not believe that there are sufficient USD reserves in place, the fiat currency which Tether is tied to. What made matters more concerning was the fact that in March 2019, Tether qualified their claim that their tokens were backed up by one USD each by stating that their backings included loans from affiliated parties [20]. To date, there is no consumer with

absolute certainty with regards to Tether's dollar reserves.

To remedy this doubt, certain stablecoins have issued auditing systems which document the amount of digital currency in place along with its corresponding reserve of fiat currency. Yet, no such system is in place to automatically update the audit after each transaction. The Gemini Dollar, for example, only releases audits monthly [21].

Gemini dollars issued and in circulation <sup>1</sup>	90,995,076.75
U.S. dollar balance of Gemini dollar Accounts <sup>2</sup>	\$90,995,076.75

**Figure 2.8:** As of December 31st, 2018, the amount of Gemini dollars in circulation was the same as the U.S. Dollar balance in the respective accounts. The previous examination, however, took place on November 30th, 2018 [21]. There is no information present to show if there were any fluctuations in collateral during the month of December.

## 2.3 Tokenisation

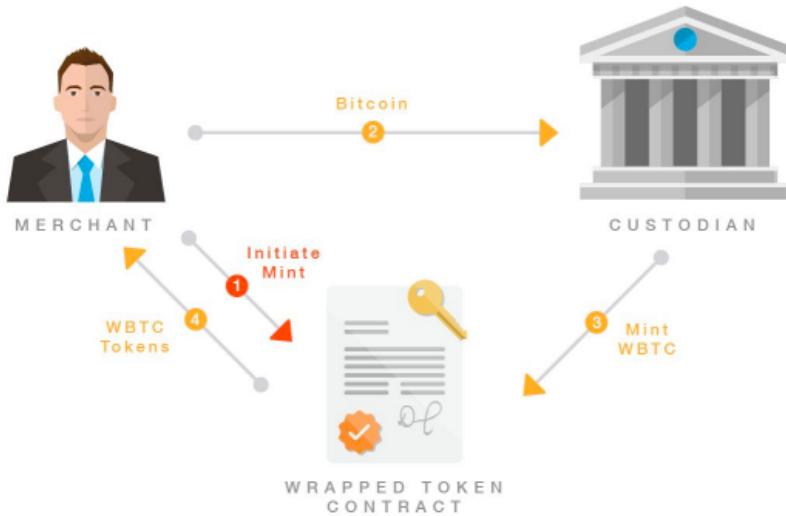
Some popular cryptocurrencies serve as ‘wrappers’ for other coins. This is referred to as tokenizing assets, and provides many advantages:

- Faster and more secure transactions on the blockchain.
- Less intermediaries for trading assets on the blockchain.
- Higher security as users can have control of their asset’s private keys.
- Greater transparency as total tokens, token creation, and removal transactions, etc. are universally known.

A stablecoin, as designed in this project, can be considered as a ‘wrapper’ for a fiat currency.

### 2.3.1 Examples

An example is the recent ERC-20 token called Wrapped Bitcoin (WBTC), which is backed by Bitcoin [21]. The following sequence of events shows how one can mint WBTC on the Ethereum chain:



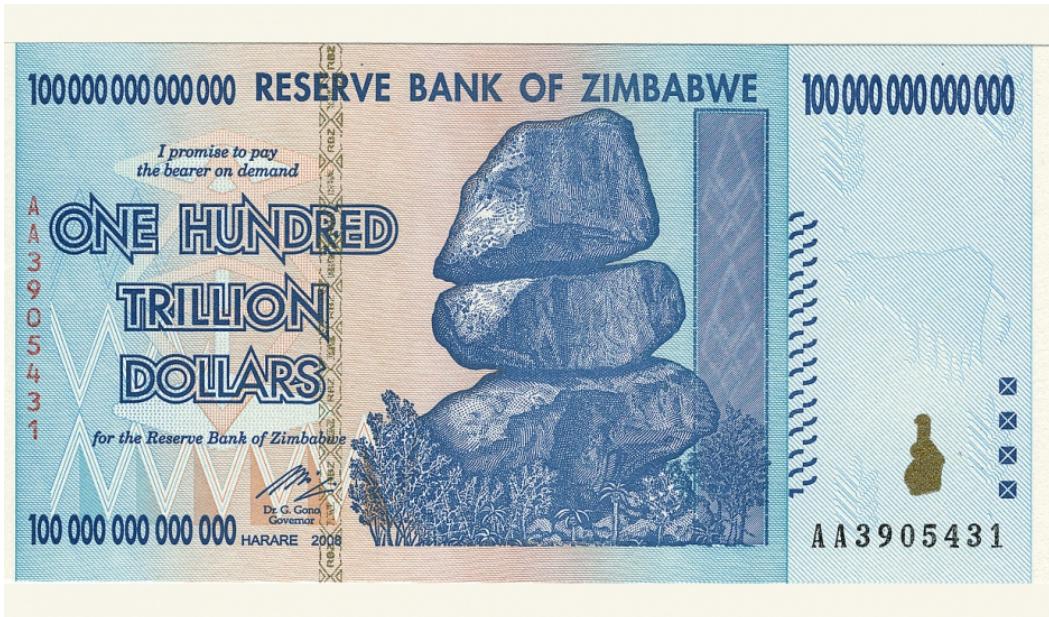
**Figure 2.9:** A merchant can send the custodian, who exchanges assets for wrapped tokens, Bitcoin, to which the custodian can issue WBTC that is stored on the Ethereum blockchain [21].

Another example of this type of token is Dogethereum. Dogethereum is a bridge between Dogecoins, a type of cryptocurrency, and the Ethereum market. Dogethereum is very similar to WBTC in its manner of issuing tokens. Unlike WBTC, however, where burning tokens results in sending a block to an unrecoverable address, each DGD token contains a lock which is activated while the token is in use. Only when the token is unlocked can the remaining Dogecoins be received [24].

Such ‘wrapping’ does not have to only involve cryptocurrencies. The ensuing example of the Zimbabwean dollar, and its eventual hyperinflation, demonstrates physical tokenisation of a fiat asset, along with its potential downsides if not audited correctly.

The Zimbabwean dollar was released in 1980 to replace the Rhodesian Dollar, and the government intended for its value to be on par (1:1) with the US dollar. Zimbabwe would require its citizens to exchange USD for ZWD upon entry to Zimbabwe, which they could theoretically redeem at any point for the same amount of USD. To summarise, this claim soon proved invalid. After time, Zimbabwe started requesting more Zimbabwe dollars to return US dollars, and consumers were unhappy as they could not use their Zimbabwe dollars outside of Zimbabwe [22]. This led to hyperinflation of the ZWD, and Zimbabwe’s currency was suspended in 2009 [23].

The example of the Zimbabwe dollar offers a cautionary tale in terms of tokenisation of both fiat currencies and cryptocurrencies. It highlights the need for a sta-



**Figure 2.10:** A one hundred trillion dollar Zimbabwe note seemed unfathomable before 2009, when the value of the Zimbabwe dollar was apparently equal to the US dollar. However, a secret deficit of USD as well as a lack of willingness from the Zimbabwe government to redeem the ZWD for USD at any time led to the devaluation of the currency, hence its massive inflation. The ‘wrapped’ nature of the Zimbabwe dollar strikes much similarity to stablecoins, highlighting the necessity of a strong auditing system, which is currently not present in any stablecoin[23].

blecoin which can consistently maintain a cycle of issuing, trading, and redeeming tokens, and whose reserves are always sufficient and verified, potentially by third-party sources.

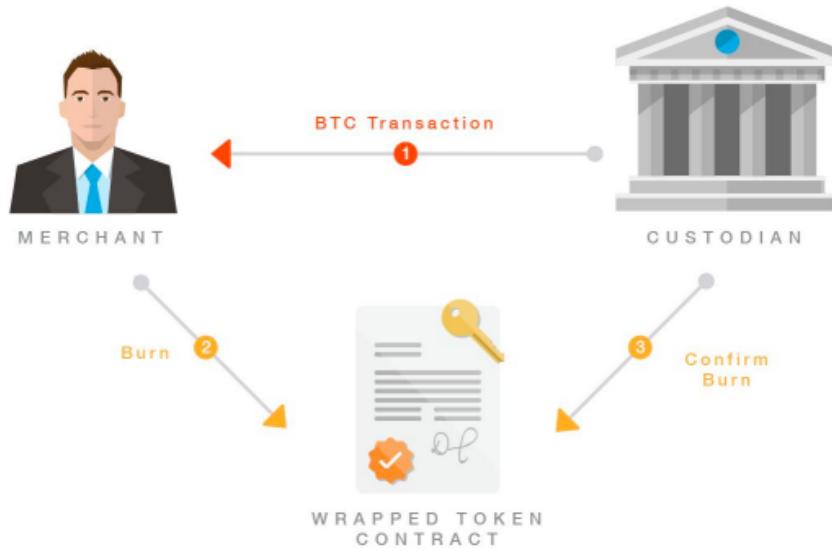
### 2.3.2 Issue-Trade-Redeem

Wrapped tokens allow users the ability to take part in the ‘issue-trade-redeem’ cycle that is vital to cryptocurrencies.

In the case of Wrapped Bitcoin, the cycle starts with a merchant sending the custodian Bitcoin, which the custodian then receives and proceeds to issue the appropriate amount of WBTC. Now, users can freely exchange this token on the Ethereum network as they wish, and can supposedly redeem it at any time and receive the corresponding amount of Bitcoin in return.

It is important to note that the custodian only creates a transaction on the blockchain to issue WBTC once the appropriate BTC transaction has been confirmed - multiple

times, in fact. Redeeming BTC for WBTC tokens, referred to as ‘burning’ is also only carried through after the custodian receives multiple confirmations from the Ethereum chain.



**Figure 2.11:** Burning or redeeming BTC reduces the supply of WBTC and the merchant’s WBTC balance [21].

Ultimately, this cycle, which is the basis of all tokens, can be summarised as follows:

1. The ability to **issue** tokens on a particular blockchain-based platform.
2. The ability to **trade** tokens to any user, with only the obvious restrictions (enough tokens to trade, etc.)
3. The ability to **redeem** tokens for the corresponding asset at any given time.

## 2.4 Summary

The following table summarises the background knowledge and existing solution by identifying the positives and negatives of the aforementioned methods of exchange:

	Benefits	Drawbacks
Cryptocurrency	Anonymity, Security, and Independence of P2P transactions.	Volatility, Lack of Regulation.
Stablecoins	Perceived Lower Volatility.	Can be Misleading if Improper Reserves are in Place.
Wrappers	Faster and More Secure Due to Less Intermediaries along the Blockchain.	Can Lead to Inflation if Not Backed Up Properly.

The aim is to create a new method of exchange which captures all the benefits of the previous three methods of exchange whilst minimising the drawbacks.

## 2.5 Vision

WBTC, Dogethereum, and the concept of tokenization inspire a similar model for stablecoins, where tokens are issued only when the appropriate fiat currency has been received and redeemed once the corresponding transaction has been created to relinquish ownership of the token. Moreover, any issued tokens can be exchanged in a similar manner to that detailed in the above section. This can successfully achieve a safe and effective ‘issue-trade-redeem’ cycle and decentralized management of a stablecoin. Additionally, as long as the coins are audited in real time, customers will have complete control of their digital currency and have little doubt of the value of their assets.

This vision led to Digipound: my final year project.

# **Chapter 3**

## **Design**

Given that such a project is quite time-constrained and intense, especially for a BEng thesis, below are the components which composed a Minimum Viable Product for the application Digipound, aptly sharing the name of the stablecoin it trades. This application was designed during my final year at Imperial College. Digipound aims to facilitate the three major parts of the issue-trade-redeem cycle for a custom ERC-20 token, Digipound, encapsulated within a website.

### **3.1 Components**

As alluded to above, the MVP of Digipound involved three main components to satisfy the preceding requirements of functionality:

- A custom token on a blockchain-based platform.
- A web application to facilitate the issue-trade-redeem cycle of such a token, involving both an API to interact with the blockchain and an API to handle payments.
- An auditing mechanism that displays both the current amount of fiat currency (GBP) in the Digipound reserve account and the amount of Digipound tokens that have been issued, which can be verified by an external source.

#### **3.1.1 Digipound**

The Digipound (symbol: DGP) represents the first component: a custom token that can be traded on a blockchain. The blockchain chosen for this project was Ethereum, given its popularity amongst developers, its size and hence security, its speed, and its compatibility with various standards of tokens. Since the specifics of the token itself were not the focus of the project, Digipound was created as a standard ERC-20 token.

Nonetheless, this process was quite time-intensive, requiring multiple weeks to compose and deploy a smart contract. Another reason why Ethereum was chosen was because of its multiple test networks. For the Minimum Viable Product of the application, the Digipound token was deployed on the Ropsten test network. This meant that Digipound still exhibits the same functionality as a token traded on the Ethereum network, without incurring a cost of transactions in Ether.

Digipound's smart contract was written in Solidity, the scripting language mentioned earlier, identically implementing the same functionality of most ERC-20 tokens. As previously stated, the specifics of the contract are irrelevant when compared amongst tokens, as the defining attribute of a token is what asset it *represents*, not how it was deployed on the blockchain platform. The following components of the application detail how the Digipound token was set up to represent the British pound, which it is fittingly named after.

The smart contract for Digipound defined the basic ERC-20 functions that were mentioned above. Alongside the constructor for the smart contract, the functions that are frequently called in the web application include `balanceOf` and `transferFrom`, whose definitions are also below.

```
constructor() public
{
    symbol = "DGP";
    name = "Digipound";
    decimals = 18;
    totalSupply = 10000000000000000000000000000000;
    balances[0x0B5e6646f3665E5132FDdAAbF1aF95386E70148f]
        = totalSupply;
    emit Transfer(address(0),
        0x0B5e6646f3665E5132FDdAAbF1aF95386E70148f,
        totalSupply);
}
```

**Listing 3.1:** The above code constructs the DGP smart contract.

Here, `balances` is defined to be a map from address to uint and the next function, `balancesOf`, represents a simple lookup function. The map `balances` is initialised by assigning the total supply of Digipound to a single address. That address represents the owner of the contract. I created an arbitrary address to serve as the holder and issuer of Digipound.

The `totalSupply` and `decimals` variables were arbitrarily defined, with the latter referring to the precision at which an amount of Digipound can be issued. For exam-

ple, the decimals of most fiat currencies are 2, as it is possible to issue £6.45 but not £6.451. Finally, the command `emit Transfer(...)` safely creates a Transfer event which is what is interfaced with the Ethereum blockchain.

The second primary function in the Digipound contract is the `balanceOf` function, returning the holdings of Digipound at a certain address on the network.

```
function balanceOf(address tokenOwner)
    public constant returns (uint balance)
{
    return balances[tokenOwner];
}
```

**Listing 3.2:** Retrieving the balance in DGP of a certain address is implemented as a simple lookup function given a global map of balances in the contract.

Finally, transferring a token from one address to another calls the `transferFrom` function, defined below:

```
function transferFrom(address from, address to, uint tokens)
    public returns (bool success)
{
    require(tokens <= balances[from]);
    balances[from] = tokens - balances[from];
    require(tokens <= allowed[from][msg.sender]);
    allowed[from][msg.sender]
        = allowed[from][msg.sender] - tokens;
    new_balance = balances[to] + tokens;
    require(new_balance >= balance[to]);
    balance[to] = new_balance;
    emit Transfer(from, to, tokens);
    return true;
}
```

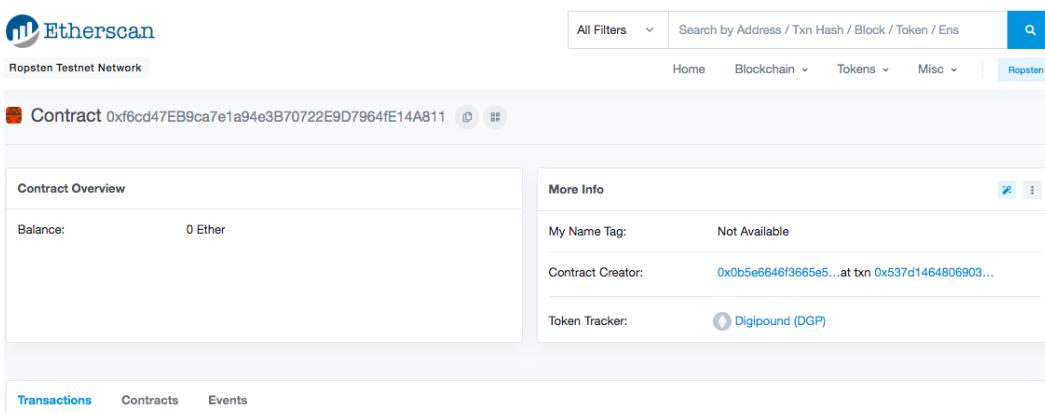
**Listing 3.3:** A transfer updates the balances of the users and then is sent to the blockchain.

This function essentially updates the balances map and then calls the `Transfer` function which interacts with Ethereum. The require checks are just to ensure that the account transferring the tokens has enough balance to make the transfer and is not transferring a negative number or a number outside its total allowance to send.

These three functions comprised the basic functionality of the Digipound smart contract, among others which were taken from the Tenzorum blockchain initiative [25].

Once the contract was written, the Remix IDE was used to deploy the contract to the Ethereum blockchain. Within the IDE, I first compiled the Solidity code and then deployed the smart contract. I chose to deploy the contract on the Ropsten test network, and after passing it through the Remix IDE, I had officially created a token on an Ethereum test network!

My contract can easily be viewed by using the block explorer Etherscan set to the Ropsten network. The following image shows the contract on the blockchain, along with its address, token holders, and list of transactions.



**Figure 3.1:** The address of the Digipound contract along with a list of transactions written to it is publicly accessible [9].

The address of the contract is what is used to identify it and write transactions to it.

Once the Digipound contract was created, my next step was to set up a web application with two components that required different APIs:

- An API to interact with Ethereum and call the token defined in the contract I just deployed to the network (web3).
- An API to handle payments, which can confirm receipt of payments before issuing tokens, and deposit funds back to a user when redemption of tokens is requested (Stripe).

### 3.1.2 web3

*web3* is a collection of JavaScript libraries which can interact with the Ethereum blockchain through an HTTP connection with a local Ethereum node [26]. It was an ideal choice given that JavaScript is a primary language used to code much of the

website's functionality. web3 can be initialised with a service provider, whether that is a local Ethereum node running on a computer (see Implementation section), or a third party provider such as Infura, which is called if no local service provider is present.

```
var web3;
if (typeof web3 !== 'undefined') {
    web3 = new Web3(web3.currentProvider);
} else {
    web3 = new Web3(new Web3.providers.HttpProvider(
        'https://ropsten.infura.io/'
        + "<?=>getenv('INFURA_PROJECT_ID')?>"));
}
```

**Listing 3.4:** Upon opening of the landing page, the instance of web3 is initialised using a service provider and can be called throughout the application.

In the application, the web3 libraries that interact with Ethereum will be used in multiple ways:

- To access the token through the Digipound contract created earlier.
- To transfer instances of the token, whether it is from my reserve account to a user, from a user to another user, or from a user back to the reserve account (issue-trade-redeem cycle).

Accessing the token requires outside information about the contract, including the contract address, as previously explained, and an *ABI*. An ABI is short for Application Binary Interface. With it, it is possible to interact with smart contracts from outside the blockchain, by encoding data about a contract according to its type [27].

The Remix IDE automatically generates an ABI upon compilation of the contract code in Solidity, so I was able to pass this result, represented as an array of datatype symbols, to the web3 library to decode. For the contract I created, the following ABI representation can be found in the Appendix. Both the ABI and contract address can be passed as arguments into web3 to be able to work with the Digipound token that I defined.

```
var ERC-20abi = ...; // see appendix
var contractAddress
= "0xf6cd47EB9ca7e1a94e3B70722E9D7964fE14A811";
var contract = web3.eth.contract(ERC-20abi);
```

```
var token = contract.at(contractAddress);
```

**Listing 3.5:** Once initialised, the `token` object can be used to perform the required functions in JavaScript that are associated with its contract.

The `token` object can now be used to perform the three stages of the issue-trade-redeem cycle of Digipound via its web application. Given that the total supply of DGP is currently held in my single ERC-20 reserve account, as was initialised in the constructor of the contract, the `transfer` method in the contract can be used for each of the stages in the cycle. The main differences are in the sender and recipient address to be defined in the parameters of the function.

### 1. Issue:

```
var mainAddress
    = "0x0b5e6646f3665e5132fddaabf1af95386e70148f";
var accountAddress = "0x" + "{$account}";
var decimals = web3.toBigNumber(18);
var amount
    = web3.toBigNumber(<?php echo $receive_amount ?>);
var value = amount.times(web3.toBigNumber(10).pow(decimals));
ethereum.enable();
web3.eth.defaultAccount = mainAddress;
token.transfer(accountAddress, value, (error, txHash)
    => {
    document.getElementById("eth_transaction").innerText
    = "Ethereum Transaction ID:" + txHash;
});
```

**Listing 3.6:** Issuing tokens is equivalent to writing a transfer on the Digipound contract from the reserve account to the user's account.

The idea is that whenever a payment of fiat currency (GBP) has been received, with confirmation from the payment API, a transfer from the reserve account to the user's account is made with the corresponding amount of Digipounds. As will be detailed later, the function obtains the information of the user's account and the amount paid from PHP, and updates the website with the ID of the transaction on the Ethereum blockchain as proof to the users that they have received the tokens. The `transfer` function has the same functionality as the `transferFrom` function, except the sender's address is set as the default account, which is specified with the line `web3.eth.defaultAccount = mainAddress`.

The next two functions utilise nearly identical code to the previous one with the subtle differences in the changes of the `mainAddress` and the `accountAddress` variables.

## 2. Trade:

```
var mainAddress = “0x” + “{{ $account }}”;
var accountAddress = “0x” + “{{ $receive_account }}”;
var decimals = web3.toBigNumber(18);
var amount
= web3.toBigNumber(<?php echo $receive_amount ?>);
var value
= amount.times(web3.toBigNumber(10).pow(decimals));
ethereum.enable();
web3.eth.defaultAccount = mainAddress;
token.transfer(accountAddress, value, (error, txHash)
=> {
document.getElementById ‘‘eth_transaction’’.innerText
=“Ethereum Transaction ID:” + txHash;
});
```

**Listing 3.7:** Writing a transfer to the network via the contract achieves the trading of DGP.

Again, information about the user’s account, the account to receive the trade, and the amount of the trade are passed in through a controller written in PHP, which will be further explained. The `transfer` function still achieves the desired affect by once again changing the default address in `web3`.

## 3. Redeem:

```
var mainAddress = “0x” + “{{ $account }}”;
var accountAddress
= “0xb5e6646f3665e5132fddaabf1af95386e70148f”;
var decimals = web3.toBigNumber(18);
var amount = web3.toBigNumber(<?php echo $receive_amount ?>);
var value = amount.times(web3.toBigNumber(10).pow(decimals));
ethereum.enable();
web3.eth.defaultAccount = mainAddress;
token.transfer(accountAddress, value, (error, txHash)
=> {
```

```

document.getElementById('eth_transaction').innerText
= "Ethereum Transaction ID :" + txHash;
});

```

**Listing 3.8:** Burning Digipound via the redeem function essentially mimics the issue function with the `mainAddress` and `accountAddress` switched.

Now that the web application was connected with the Ethereum blockchain through the `web3 API` and specified how to transfer tokens according to the issue-trade-redeem cycle, the next step was to link the preceding functions with a payment API that could accept payments before issuing tokens and return payments after redeeming tokens.

### 3.1.3 Stripe

When choosing an API to handle payments, I considered multiple options, including the `Paypal API` and `Stripe API`. When making my decision, I considered the two cases where I would require the API. These cases occurred in the issue and redeem stages of the issue-trade-redeem cycle and would require the following functionalities:

- The ability to accept any bank card from users and process the payment, giving confirmation within the web application that the payment has been received to instruct it to call the Digipound issue function defined earlier.
- The ability to send back money to users' bank accounts after they transfer back the Digipounds into the holder account, completing the redeem stage.

In both of these cases, Stripe proved to be a major update over `Paypal`. In fact, in my first version of `Digipound`, I had made a demo using the `Paypal API`. However, it was analysed with respect to the preceding goals and I realised that a change had to be made to the API, for the following reasons.

In the first example, Stripe was an improvement over `Paypal`, as when accepting payments, `Paypal` takes users to their website to complete the payment and then updates the balance of the application's account. However, this can be problematic if a payment does not go through, because then the application will be unsure of whether or not to issue DGP to the user.

On the other hand, Stripe can create bank transactions through the application, and in turn, give local confirmation if the payment has gone through. In the first draft of my application with `Paypal`, I had mistakenly assumed success of a payment and issued the tokens automatically; however, as will be explained in the Testing section,

it is important to first validate the success of the payment within the application before issuing the tokens, which is why Stripe was used in the ultimate version.

```
public function submit(Request $request)
{
    Stripe::setApiKey(getenv('STRIPE_SECRET'));
    try {
        $charge = Charge::create([
            'amount' => $_POST['amount']*100,
            'currency' => 'gbp',
            'source' => $_POST['token'],
            'description' => "test"
        ]);
        $new_balance = Auth::user()->balance;
        $new_balance += $_POST['amount'];
        Auth::user()->update
            ([['balance' => $new_balance]]);
        if ($charge->status == "succeeded") {
            $transaction = Transaction::create([
                'user_id' => Auth::user()->id,
                'amount' => $_POST['amount'],
                'stripe_transaction' => $charge->id,
                'account' => Auth::user()->account
            ]);
            $request->session()->flash('status',
                'Transaction Successful');
        }
        else {
            $request->session()->flash('error',
                'Transaction failed');
        }
    } catch (\Stripe\Error\Card $e) {
        $request->session()->flash('error',
            'Transaction failed');
    }
    return redirect()->route('home');
}
```

**Listing 3.9:** When accepting payments, I create a Charge object through Stripe which contains a status field detailing if the payment was received successfully [28].

This is an excerpt from a *controller* in PHP - the model-view-controller architecture of my application will be elaborated upon later - which handles payments. With the Stripe API, users can enter their card details (the `token` variable) and an amount (the `amount` variable) to an HTML front end page. Submission routes to the above function, where a `Charge` object is created in Stripe that routes the payments into a reserve bank account that I created. The middle parts of the function are concerned with updating the local user database to keep a backup record of the DGP balance of each user and a list of all transactions for display purposes on a user's home page.



**Figure 3.2:** Stripe provides a widget to collect card details from the front end of the Digipound web application.

As for the second scenario where the API would be used, both Stripe and Paypal had functionality for returns. In principle, when a user wanted to redeem Digipounds for pounds, since both APIs kept a list of transactions, they could return some or all of the pounds from a combination of transactions back to the user depending on the amount of Digipounds to redeem. This seemed acceptable, but when considering edge cases, I realised that traditional returns did not allow total freedom of redeeming Digipounds in all cases.

The preceding implementation works fine if users want to redeem an amount of Digipounds less than or equal to the total amount of Digipounds they have paid for, but consider this situation, which is unable to be solved with the Paypal API:

- User 1 purchases £15 worth of DGP (15DGP).
- User 2 purchases £10 worth of DGP (10DGP).
- User 2 trades 5 DGP to User 1.
- User 1 redeems 19 DGP for £19.

This would be problematic, as when processing the last transaction, there is not £19 worth of transactions User 1 has made through Paypal or Stripe that can be refunded. Hence, an alternative to traditional refunds was desired. The Paypal API has no such alternative. Stripe, on the other hand, offers a feature called Payout, which can directly deposit money from the bank account I created to serve as the reserve account to any user's bank account.

The following code details how payouts can be issued through Stripe, when called from another controller class in PHP after requesting a redemption of DGP in the front end of the website.

```

public function submit(Request $request) {
    if($_POST[‘amount’] <= Auth::user()->balance) {
        if ($_POST[‘amount’] >= 0) {
            $new_balance = Auth::user()->balance;
            $new_balance -= $_POST[‘amount’];
            Auth::user()->update([‘balance’ => $new_balance]);
            Stripe::setApiKey(getenv(‘STRIPE_SECRET’));
            $transfer = Transfer::create([
                ‘amount’ => $_POST[‘amount’]*100,
                ‘currency’ => “gbp”,
                ‘destination’ => Auth::user()->account_no
            ]);
            Payout::create([
                ‘amount’ => $_POST[‘amount’]*100,
                ‘currency’ => “gbp”,
                ‘description’ => “Digipound Return”
            ], [‘stripe_account’ => Auth::user()->account_no]);
            $transaction = Transaction::create([
                ‘user_id’ => Auth::user()->id,
                ‘amount’ => $_POST[‘amount’],
                ‘stripe_transaction’ => ‘Refund’,
                ‘account’ => Auth::user()->account
            ]);
            $request->session()->flash(‘refund_status’,
                ‘Refund Successful’);
            return redirect()->route(‘home’);
        }
        else {
            $request->session()->flash(‘error’,
                ‘Transaction failed: Invalid number.’);
        }
    }
    else {
        $request->session()->flash(‘error’,
            ‘Transaction failed: Not enough funds.’);
    }
    return redirect()->route(‘home’);
}

```

**Listing 3.10:** A Payout object is created in Stripe to facilitate transfers of GBP in the redeem phase.

The function first makes the necessary checks before processing the payout. First, it

determines if the user has enough DGP to redeem, which can be accessed through the backup record of user balances I store locally. Then, it ensures that the user is redeeming a positive amount.

Making a payout to a user's bank account in Stripe requires two steps. The first step is to create a Transfer object, which gives permission to transfer funds from the bank account associated with Stripe (the reserve account) to a connected account, the user's account. Then, a Payout object can be created using similar fields as the Transfer object which completes the transaction.

Activity					
Payments	Transfers	Payouts	Collected fees		
AMOUNT	EXTERNAL ACCOUNT		DESCRIPTION	INITIATED	EST. ARRIVAL
£2.00 GBP	Paid	STRIPE TEST BANK ... 2345	Test Payment	May 31, 2019, 12:14 PM	May 31, 2019
£5.00 GBP	Paid	STRIPE TEST BANK ... 2345	Test Payment	May 31, 2019, 12:10 PM	May 31, 2019
£1.00 GBP	Paid	STRIPE TEST BANK ... 2345	Test Payment	May 29, 2019, 2:43 PM	May 29, 2019
£10.00 GBP	Paid	STRIPE TEST BANK ... 2345	Test Payment	May 29, 2019, 2:31 PM	May 29, 2019
£20.00 GBP	Paid	STRIPE TEST BANK ... 2345	Test Payment	May 29, 2019, 2:29 PM	May 29, 2019

[View all payouts](#)

**Figure 3.3:** Stripe can issue payouts to external bank accounts so long as there are sufficient funds in the reserve bank account; this means that after they redeem a valid amount of Digipounds, users will be directly deposited GBP into their bank accounts, completing the issue-trade-redeem cycle of the Digipound stablecoin.

In order to ensure that a user's bank account can be paid via Stripe, it is set up as a connected account in Stripe. This is done upon their registration for the Digipound application. The function is defined in the Appendix. A user enters a bank sortcode, referred to as routing in Stripe, and account number, along with other relevant personal information such as date of birth. Thus, upon registering for Digipound, the user's bank account details are also registered as a connected account in Stripe; Stripe then assigns the connected account an account ID that is stored locally and accessed when making payments, as seen in the line 'account' => Auth::user()->account.

Now, the issue-trade-redeem cycle has been extended to include payments, ensuring the completeness of the Digipound stablecoin. The web application consists of a connection to the Ethereum blockchain (test network) and a connection to a service that can handle and confirm payments. The final component of the web application is tying this all together in an audit that updates in real time.

The screenshot shows the Stripe account dashboard for a user named Michael Smith. At the top, it displays a summary of payments and payouts, showing a total balance of £0.00 GBP and a lifetime total volume of £50.00 GBP. Below this, the 'Identity' section contains personal information such as name, date of birth, and address, with some fields marked as verified. The 'Balance' section shows the total balance, future payouts, and in-transit bank amounts. The 'Activity' section lists recent payments made to an external account (Stripe Test Bank) with amounts of £30.00, £10.00, and £10.00, all marked as paid.

AMOUNT	EXTERNAL ACCOUNT	DESCRIPTION	INITIATED	EST. ARRIVAL
£30.00 GBP Paid	STRIPE TEST BANK ... 2345	Test Payment	May 22, 2019, 4:43 PM	May 22, 2019
£10.00 GBP Paid	STRIPE TEST BANK ... 2345	Test Payment	May 22, 2019, 4:38 PM	May 22, 2019
£10.00 GBP Paid	STRIPE TEST BANK ... 2345	Test Payment	May 22, 2019, 4:28 PM	May 22, 2019

**Figure 3.4:** Stripe allows the creation of a connected account, which can verify personal information and keep record of all payouts made to the user. Information for all accounts and payments can be viewed on the dashboard in the Stripe website.

### 3.1.4 Audit

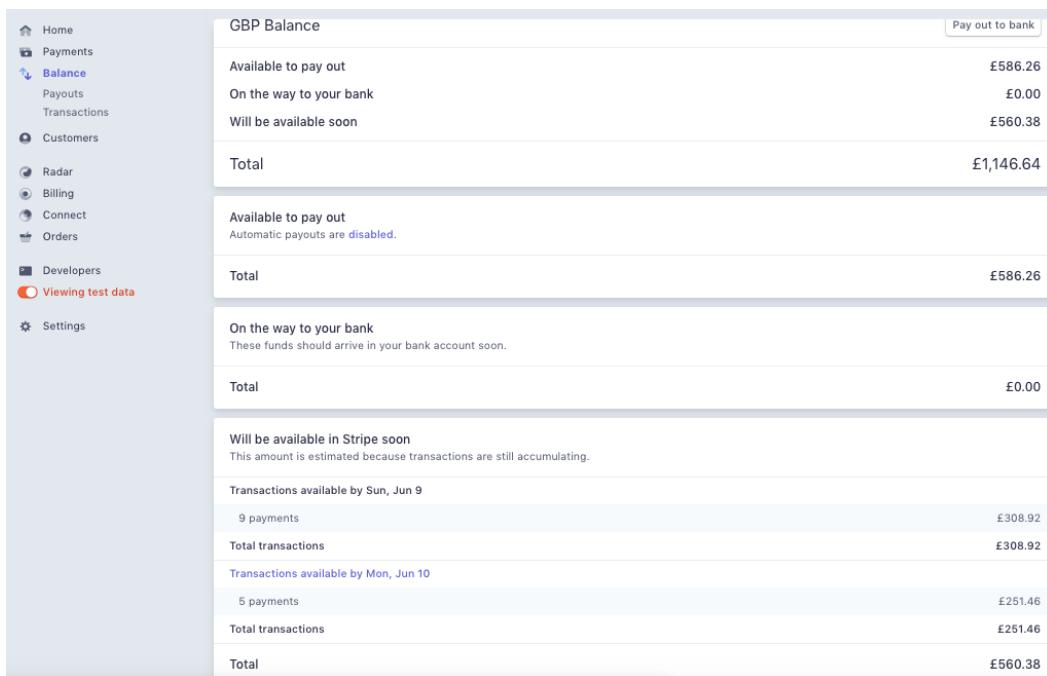
A reputable and frequently updating auditing system is one of the main components that distinguishes Digipound from other stablecoins and cryptocurrencies such as Tether and the Gemini Dollar. For the auditing system, there are two final quantities that it must report:

1. The total amount of pounds in the reserve account (which was created when setting up the Stripe connection).
2. The total amount of Digipounds that have been issued.

In both cases, it is necessary to have a third party be able to provide some verification to the claims made in the audit. Take for example the Gemini Dollar, mentioned earlier. According to their website, the US dollar holdings of the Gemini Dollar are ‘examined monthly by BPM, LLP, a registered public accounting firm, in order to

verify the 1:1 peg [17].'

Similarly, Stripe can verify the GBP holdings of the reserve account. To accomplish this, every time Stripe verifies a payment, I not only issue tokens through the web3 API, but also keep a backup record of account payments that is directly tied to the Stripe platform. This is because, every time a transaction is created, I record a Stripe transaction key, a string, which can be looked up. Using this key, I can analyse the amount of each transaction, whether into the reserve account or from the reserve account. Now, there is a record of transactions which can be accessed locally, which are verifiable by a third party.



The screenshot shows the Stripe dashboard with the 'Balance' tab selected. The main area displays the 'GBP Balance' section. It includes a summary table with three rows: 'Available to pay out' (£586.26), 'On the way to your bank' (£0.00), and 'Will be available soon' (£560.38). Below this is a 'Total' row (£1,146.64). A note below the first row states: 'Available to pay out Automatic payouts are disabled.' The 'On the way to your bank' section notes: 'These funds should arrive in your bank account soon.' The 'Will be available soon' section notes: 'This amount is estimated because transactions are still accumulating.' At the bottom, there are two sections for 'Transactions available by Sun, Jun 9' and 'Transactions available by Mon, Jun 10', each with a table of 9 payments and a total of £308.92 and £251.46 respectively. The final 'Total' for the day is £560.38.

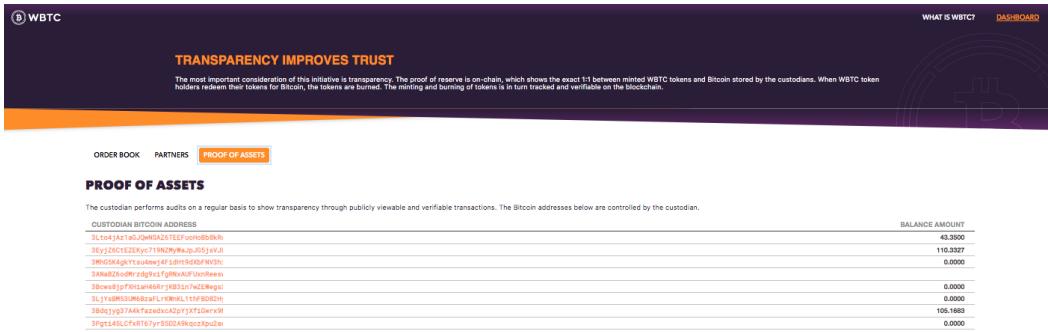
GBP Balance		Pay out to bank
Available to pay out	£586.26	
On the way to your bank	£0.00	
Will be available soon	£560.38	
Total		£1,146.64
Available to pay out Automatic payouts are disabled.		
Total		£586.26
On the way to your bank These funds should arrive in your bank account soon.		
Total		£0.00
Will be available in Stripe soon This amount is estimated because transactions are still accumulating.		
Transactions available by Sun, Jun 9		
9 payments	£308.92	
Total transactions	£308.92	
Transactions available by Mon, Jun 10		
5 payments	£251.46	
Total transactions	£251.46	
Total		£560.38

**Figure 3.5:** Stripe keeps a running total of the balance in an account. Although this view is not publicly accessible to users, by keeping track of the Stripe transaction key each time a successful purchase is made, the application can effectively track and display the balance of the reserve account by showing individual transactions.

For the second case, two options were considered. Clearly, using the web3 API, it is simple to add up the DGP balance of all accounts using the `balanceOf` function to display a final DGP total in the audit. I had actually implemented this in the first iteration of the audit, and presented it to my supervisor, who voiced some concerns. For, although web3 is a third party, there is no indication of its presence in the front end of the audit, and there is no way to verify this information via an external link. Therefore, the first implementation of the audit looked exactly like the audit of the

Gemini Dollar. As seen in the Background section, the audit of the Gemini Dollar is not descriptive at all, just showing two numbers representing each of the previous quantities. After strong consideration, I resolved it was necessary to have a more thorough auditing system.

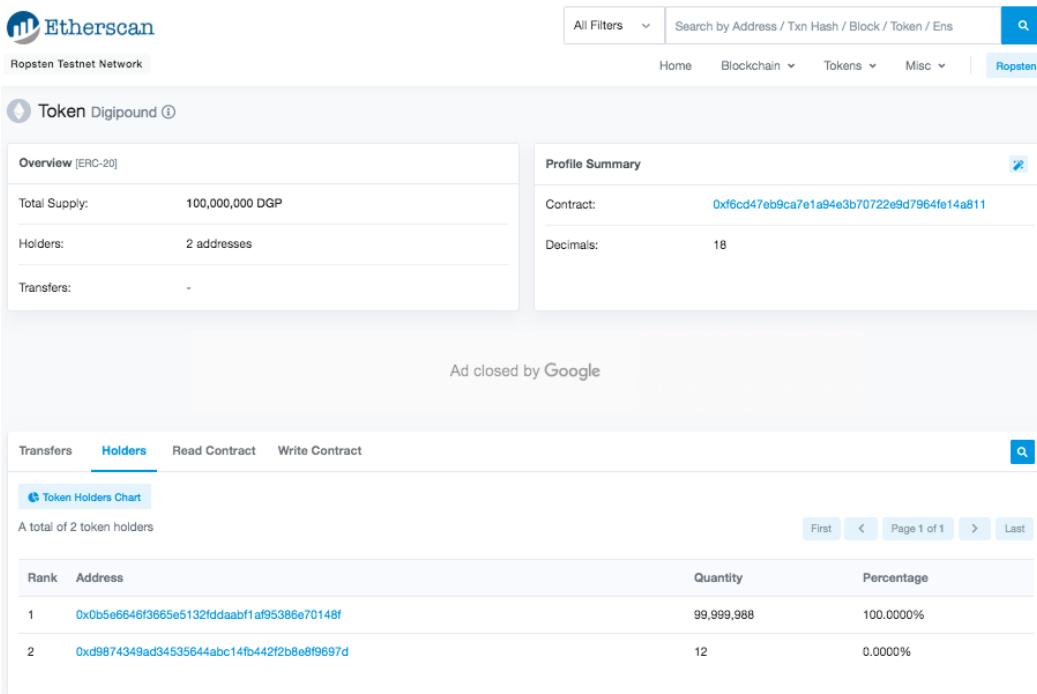
I consulted other auditing systems to inspire Digipound's auditing system, one of which was Wrapped Bitcoin (WBTC). In this particular section of their audit, titled “Proof of Assets,” WBTC details how their token, Bitcoin, is spread across multiple custodian addresses, providing full transparency of its assets to users. My supervisor and I agreed on a more detailed “Proof of Digipound” section that displayed the balances and addresses of all accounts and then linked a third party to validate this. Thus, the first option of using the web3 API to verify the Digipound holdings was removed in favour of a second option: a block explorer.



**Figure 3.6:** WBTC displays the balances of all of its custodian addresses, which serve as the intermediary in transactions. Since one of Digipound's aims is to eliminate this intermediary, I used a block explorer to find and display a similarly-formatted audit.

A block explorer is an online application that is essentially a search engine for the blockchain. It displays the contents of individual blocks, smart contracts, transactions, and, most importantly, balances of addresses. It is also publicly accessible. For example, Bitcoin has a block explorer, called Bitcoin Block Explorer. For the purposes of this project, the Ropsten Etherscan block explorer was used, an Ethereum block explorer specifically on the Ropsten test network.

Thus, the second part of the audit was implemented by including a link to the Etherscan page of the Digipound token. In the ‘Holders’ tab, a list of addresses and balances is made explicit, so using a third party definitely results in a more transparent audit for users. In terms of adding to the front end of the webpage outside of just displaying a link to the Etherscan block explorer, I use the local database of balances and address previously mentioned and display these values to mimic the WBTC layout. Another possibility was to also reference the balances from web3, but given that



**Figure 3.7:** Etherscan can show the balance of all account holders of DGP; excluding the address “0x0b5e....”, the main token holder, the sum of the balances of the other addresses satisfies the validation of the second part of the audit.

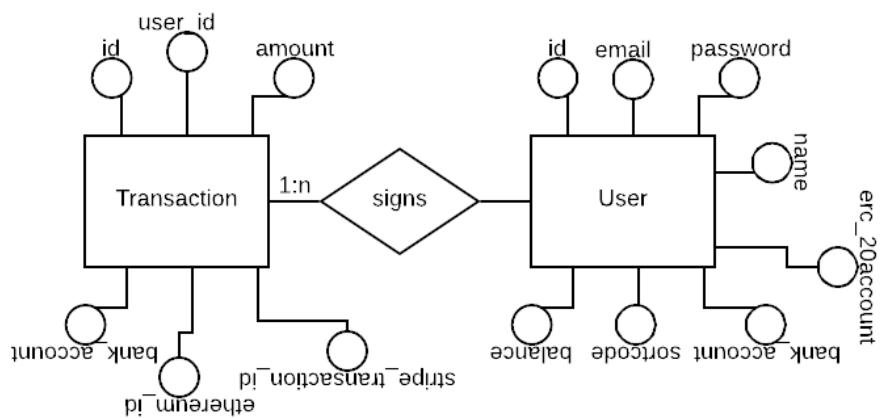
I added a link to Etherscan to verify it [9], I did not want to waste computation time through additional API calls as this is still a complete way to display and validate the total amount of DGP issued for the audit.

With both components carefully thought out, the design of the audit was completed. Now, a token was present, the issue-trade-redeem infrastructure was in place, and the functionality of the audit was finished. The next step was to incorporate these into a web application to which users could easily navigate.

# Chapter 4

## Implementation

When designing the Digipound platform, multiple considerations for the infrastructure of the website were considered. One was the generic client-server pattern, yet, with a complex database and connections to multiple APIs, I desired an architectural pattern with additional tiers.



**Figure 4.1:** The database consists of two main tables, users and transactions, that are connected and hold various fields relating to both Stripe, which refer to the payment of fiat currency, and Ethereum, which refer to blockchain holdings of DGP. These tables were abstracted as models in the model-view-controller architecture of the Digipound application.

## 4.1 MVC

Using the insight I gained in the second year of my degree, especially in the Software Engineering: Design module, I chose to incorporate the Model-View-Controller (MVC) pattern in my website [29]. From the course, and from additional background research, I realised that this pattern is suitable for my application given its three parts:

1. Model: contains the data and specifies methods for each object in the database.
2. View: displays data to user.
3. Controller: handles user input to modify data.

It is important to understand that throughout a web application like Digipound, there are numerous models, views, and controllers that all communicate whilst maintaining their core tasks. In particular, there are two models specifying the relationship between Users and Transactions, with additional functions added to each.

To give one example of how the MVC architecture works within my web application, take the process of registration. Upon entry to the website, after the landing page, a user is presented with a form (written in HTML) to register. This is a view:



The image shows a registration form titled "Welcome to Digipound." at the top. Below the title is a sub-headline "Join a community of intelligent investors in seconds!". A descriptive text below that says "Please provide us with your ERC20 address so we can issue you digipounds, and your bank details so we can redeem them at any time!" The form consists of six input fields arranged in a 3x2 grid. The first row contains "First Name." and "Last Name.". The second row contains "Email." and "ERC20 Account.". The third row contains "Sortcode (no hyphens!)." and "Bank Account Number.". At the bottom of the form is a "Register" button.

**Figure 4.2:** Registering to trade Digipound requires both an Ethereum address where DGP can be issued to and also a traditional bank account where funds in GBP can be delivered to after DGP are redeemed by the user.

The user enters this information into the form, and upon submission (clicking the ‘Register’ button), a POST request is made to a controller called the *Register Controller* in PHP. This references the model, which is another class in PHP called User. The POST request calls the function `create`, fully shown in the Appendix, which not only creates a connected account for future payout purposes, as described above, but also updates the model by creating an object of type User, which is stored in the database.

```
return User::create([
    'name' => $data['name'] . ' ' . $data['surname'],
    'account' => $data['account'],
    'email' => $data['email'],
    'password' => Hash::make($data['password']),
    'routing' => $data['routing'],
    'account_no' => $acct->id,
    'balance' => 0
]);
```

**Listing 4.1:** The function `create`, when applied to the User model, creates and returns a new user in the database.

After the controller references the model and updates the database, the user is taken to another view, the home page. The user’s login details have been successfully verified and the user can access the home page where information such as the user’s name and the list of transactions made will be presented.

```
class User extends Authenticatable
{
    use Notifiable;

    protected $fillable = [
        'name', 'email', 'account', 'routing',
        'account_no', 'password', 'balance'
    ];

    protected $hidden = [
        'password', 'remember_token',
    ];

    public function transactions()
    {
        return $this->hasMany('App\Transaction');
    }
}
```

```
}
```

**Listing 4.2:** The User model class specifies what qualities in the database can be modified and publicly viewed. Additionally, it also specifies a function to return a collection of transactions. Given that Transaction model has a foreign key which points to a User, the `hasMany` relationship can be called to find all transactions associated with each user's primary key, `id`.

This is one of the many interactions between models, views, and controllers in Digipound. With a sound architectural pattern for the website in place, the next step was to utilise a web framework to add more structure to the codebase. Web frameworks are vital to use in a time-intensive project as this, in order to expedite tedious processes such as creating controller classes, organizing code into folders, etc. By using a web framework, I could ensure a coherent and organised codebase while being able to focus on the creative element I was bringing to the website: implementing the issue-trade-redeem cycle and the real time audit of Digipound.

## 4.2 Laravel

There are multiple frameworks that use the MVC framework, such as Django, Ruby-on-Rails, and Laravel. Laravel was chosen because I had the most experience with it, having learnt about it during my second year Web Applications group project at Imperial [30]. Reasons why Laravel was chosen over other web frameworks included the following:

- Built-in models, views, and controllers for authentication, which I was able to modify to achieve the above registration process and extend to allow users to login.
- Strong community with extensive documentation, from running local servers to updating the database through running migrations.

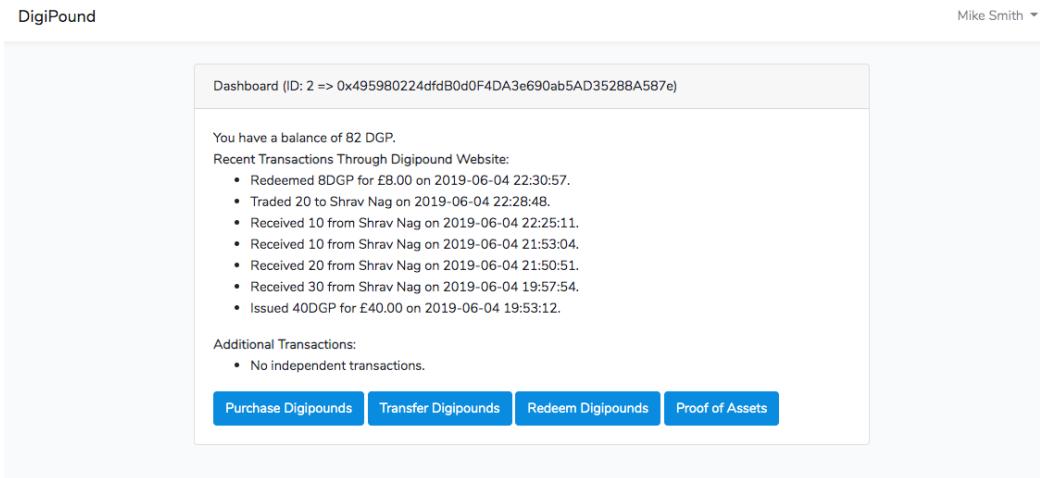
To manage Laravel's dependencies, the `composer` command is called in the terminal. To start the local server, I run the command `php artisan serve`. This deploys my website to the localhost and this is what hosts it for the MVP.

The following shows the flow of how both the issue-trade-redeem cycle and the real time audit were achieved using Laravel and the components mentioned in the Design section.

### 4.2.1 Issue-Trade-Redeem

**Issue:**

1. From the home page, if users select ‘Purchase Digipounds,’ they will be directed to the payment view. This page consists of the Stripe widget, and a form to enter the amount of Digipounds to purchase.
2. After submitting the form, a Payment Controller is called which processes the bank card details that the user entered, using the Stripe API. The controller contains the function to create a Charge object in Stripe, as seen in the earlier listing. Once created, this object contains an element status, to inform me if the payment has been received.
3. If the status field returns *success*, then, before displaying the ensuing the HTML view, I call the issue function in web3 to transfer the amount of tokens posted in the form to the address of the user, which is accessed through the User model.
4. Additionally, in the Payment Controller, I then reference the Transaction model and update the database by creating another transaction which contains the current user’s id as a foreign key. I then add the stripe\_transaction id generated from creating the Charge object, so this payment can be referenced in the audit.
5. After the back end tasks have been completed, I refresh the home view. The home view lists all the transactions whose foreign keys link to the current user’s id, so the view is accordingly updated by the controller as it references the Transaction model.



**Figure 4.3:** From the home view, the user can see all previous transactions made through the DigiPound website, consisting of all three elements from the issue-trade-redeem cycle.

**Trade:**

The trade and redeem cycles in terms of the MVC architecture pattern are nearly identical with slight discrepancies of the functions. Again, a user starts from the home page before calling the Transfer Controller by clicking the ‘Transfer Digipounds’ button. They are presented with another form with two options: the user to transfer Digipounds to and the amount of Digipounds to transfer.

Multiple fields were considered when deciding how a user would select another user to transfer Digipounds to. The most natural one was the id field, the primary key of the user. However, after coding this form and presenting it to my supervisor, I realised that it seemed a bit ambiguous to transfer significant assets to someone who was abstracted as just a number. Another consideration was using the user’s name, but given that there is a strong possibility of multiple users having the same name, this approach was turned down. A natural alternative was to input the ERC-20 address of the recipient user, yet I personally disliked this option. My vision is for Digipound to serve as an easy-to-use bridge between fiat currency and the decentralised web and if users have to type in a long hexadecimal string to send money, they may as well make the transfer over a more complicated third-party blockchain platform.

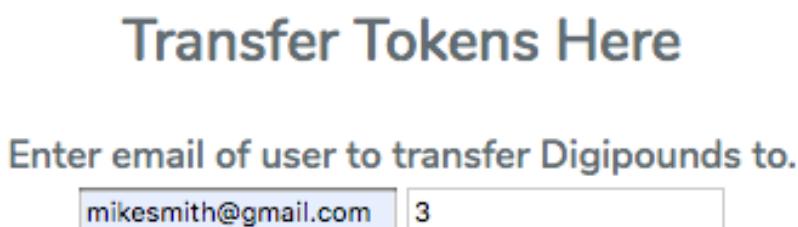
Thus, a compromise was made with a field whose length was in between that of the user’s id and the user’s ERC-20 address: the user’s email address. Therefore, in the transfer view, a user would insert the email address of the recipient user along with the amount to transfer. Before commencing with this feature, I first had to ensure that the email address field was unique amongst users. This led me to modify the form in the registration controller mentioned above. Before a user is created in the registration controller, the fields inputted in the registration form are passed through a Validator object, which verifies factors such as the type of data submitted into a form or the length. In the case of ensuring that email addresses are unique among registered users, all that is necessary is adding a field to the email input to the validator.

```
return Validator::make($data, [
    'name' => ['required', 'string', 'max:255'],
    'surname' => ['required', 'string', 'max:255'],
    'account' => ['required', 'string', 'max:255',
                   'unique:users'],
    'routing' => ['required', 'string', 'max:255'],
    'account_no' => ['required', 'string', 'max:255'],
    'email' => ['required', 'string', 'email', 'max:255',
                 'unique:users'],
    'password' => ['required', 'string', 'min:6',
                    'confirmed'],
]);
```

]);

**Listing 4.3:** Adding ‘unique:users’ to the email field in the validator ensures that every email passed through the registration form is unique in the context of the User model. This allows me to use the email as a way for users to identify recipients of Digipound transfers. This validator is created everytime the Registration Controller is called.

After entering the email of the recipient and the amount to transfer, in the same fashion as the issue cycle, another controller is called which first lookups the user from the email via the User model and the following command: \$recipient = User::where('email', \$\_POST['email'])  
->first(); Here, \$\_POST['email'] refers to the email address submitted by the form in the initial view, which sends a POST request to the current controller through HTTP.



**Figure 4.4:** Transferring DGP through the Digipound platform is aimed to be as straightforward as sending an email.

After looking up the recipient, the local values of both users’ balances are incremented or decremented by the amount in the POST request. Following that, the Transaction model is called to document this transaction in order to update the home view in the user.

Because there are no Stripe API calls in the trade part of the cycle, as it only deals with digital currency, it was important to make my own checks to determine if a transaction is successful. Since no API calls are made yet, before the application determines to call the web3 API and transfer Digipounds, I have to run checks at the onset of calling the controller. Two checks are necessary before commencing a trade:

1. The user has entered a valid email address.
2. The user has entered a valid amount to transfer.

These can just be simple if checks in the controller; in the first case, I have defined \$recipient to be the first user whose email field matches the email field in the POST request. I just have to determine if this value is null; if so, then this is not a valid trade, and the application returns to the home view immediately and flashes an error message. Otherwise, the second check is run, calling two if statements (if the amount is positive and less than the user's balance). If this passes, the local database is updated, as described above, then the trade function in JavaScript using web3 is called, which is the same as the issue function with the difference of the sender and return address input. Lastly, the home view is shown and the transfer is complete.

### **Redeem:**

Much like the preceding stages, the redeem phase starts off at the home view and calls another controller, which runs the same check on the validity of the amount to redeem as in the trade stage. Then, a Payout is created in Stripe accordingly. Much like a Charge object in Stripe, a payout contains a success field that determines whether the web3-backed redeem function is called, burns the DGP and restores the balance of the original ERC-20 address, and a transaction is added to the local database and the home view is updated.

As such, the issue-trade-redeem cycle was implemented in the website. The final step was to effectively explain this cycle to users, which was carried out through a front end redesign on the Digipound landing page.



**Figure 4.5:** The issue-trade-redeem cycle is made clear to users upon their entry to the Digipound website. Digipound ensures easy and effective maintenance of users' digital currencies.

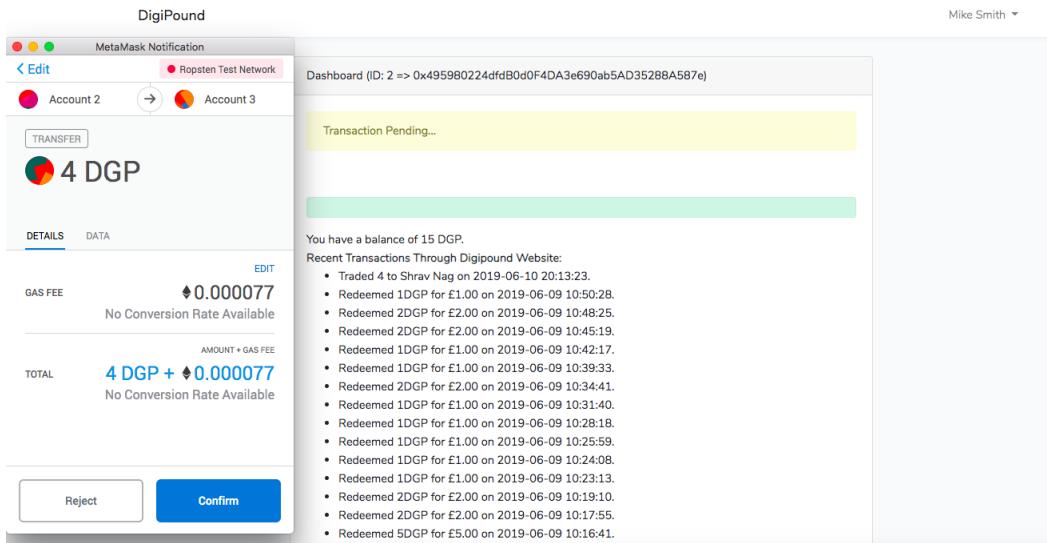
### 4.2.2 Signing Transactions

As is protocol, after issuing any transaction, a user must sign it with a private key before committing it for secure purposes. The web3 API does include a function for signing a transaction, but implementing this within the Digipound application would require users to submit their private keys into the application's database upon registration. In this case, when a user makes a transaction along the issue-trade-redeem cycle, the associated private key can be called from the User model and the transaction can be automatically signed.

This seems sensible in practice. But in reality, users will not feel comfortable divulging their private keys. In the chance that the application is hacked, a hacker will have access to all of the Digipounds that have been issued, given that the public key is stored in the database as the from of the user's ERC-20 address.

An option could be to hash the private key when storing it, just as is applied to the password in the User database. However, this still causes one major drawback: users will not have to enter their private keys when sending a transaction. This is very unsafe, as a scenario could unfold where one user's username and password can be stolen, in which case Digipounds can be directly accessed through the application even if the private key is hashed.

As such, implementation of Digipound called upon a third-party browser extension that interacts with Ethereum to be able to manage private keys and sign transactions. One of the most popular browser extensions is Metamask, which works via the web3 API, as well. It allows users to access the decentralised web and securely store their private keys when they create an account. This is an excellent second line of defense for Digipound transactions, as a user can initiate all of the issue, trade, and redeem functions directly through the Digipound platform, which will communicate with Metamask via the web3 API. Metamask implements the signing function in web3 that I contemplated including the application, but chose not to given the above explanation. Hence, users can sign their transactions through Metamask because Digipound is an ERC-20 token.



**Figure 4.6:** Through Digipound and Metamask, this user is able to separate both the application needed to trade and redeem Digipound and the user's private key. When attempting to make a transfer, the Metamask browser extension is called and prompts the user to confirm, adding security and completion to the issue-trade-redeem cycle in the Digipound application.

### 4.2.3 Audit

Given that the audit is the one of the main selling points of Digipound, I decided that it would be best to display the audit directly on the landing page. Proof of assets should be transparent to everyone and at anytime, differing from the views of current market stablecoins such as Tether and the Gemini Dollar, who either have no auditing system or a vague report hidden deep in their website.

Thus, to display the landing page, a controller for the audit is actually called and passes to the view multiple parameters:

- The current DGP balance of all the users, which is achieved by accessing the User model through calling `User::all()`, iterating through this collection, and adding up the balance of each individual user which was recorded locally.
- The list of all users, called by `User::all()`. This will help verify the preceding declaration because I can display individually the users' ERC-20 addresses and the balance at those addresses, which additionally elaborates on the proof of assets to give users a more thorough audit. This feature will be supported by a link to Etherscan to confirm the balances at each address.
- The list of all elements in the Transaction model, called with `Transaction::all()`.

To the list, I filter out all transactions that were part of the trade section of the cycle (i.e. ones that involved no transfer of fiat currency). I then identify these by the `stripe_transaction` field in the model. Fields beginning with ‘ch’ refer to charges (money entering the reserve account) and fields beginning with ‘po’ refer to payouts (money leaving the reserve account).

This collection of transactions is used to show the fluctuation of money in the reserve account and how it aligns with the amount of Digipounds being issued. To link to a third party, both the inbound and outbound transactions of GBP (charges and payout) are listed, adding a hyperlink to each element to the Stripe website, where one can look up the transaction based on the transaction ID associated with the Transaction model. The last use of the Transaction collection is to identify the date at which the last transaction was made. Although the audit is updated in real time, it would be good for users to identify the time at which the last transaction was made as an extra point of reference to the accuracy of the audit.

## 1 : 1

Always transparent. Digipounds are digital representations of GBP, currently trading at an equal valuation. Here's proof.

The figure consists of two side-by-side screenshots of a web page titled "158".

**Left Screenshot:**

- Header:** 158
- Section:** GBP in the reserve account.
- Table:**

Current Inbound Payments:	Current Outbound Payouts:
£6.00 -> 2019-06-04 21:54:36.	£8.00 -> 2019-06-04 22:30:57.
£10.00 -> 2019-06-04 21:49:40.	£70.00 -> 2019-06-04 19:59:39.
£30.00 -> 2019-06-04 21:48:32.	
£50.00 -> 2019-06-04 21:47:22.	
£100.00 -> 2019-06-04 19:57:01.	
£40.00 -> 2019-06-04 19:53:12.	
- Footer:** Verified by

**Right Screenshot:**

- Header:** 158
- Section:** DGP issued.
- Table:**

Current Holders:
0xD9874349Ad34535644ABc14fb442f2B8e8f9697d -> 76 DGP.
0x495980224dfdBodoF4DAje690ab5AD35288A587e -> 82 DGP.
- Text:** 99999842 DGP remaining in stock.  
Last transaction on 2019-06-04 22:30:57.  
Verified by [Etherscan](#).
- Note:** Exact totals of users may be subject to independent transactions, but total amount of DGP issued always correct.

**Figure 4.7:** The audit, as displayed publicly on the Digipound website, ensures the ‘stability’ of the DGP stablecoin at all times.

The controller interacts with the necessary models and sends these parameters to the landing page view, which updates the audit section accordingly. This is the main feature of the project, so I made sure to format the view in a clear and unique way. Features were added to the front end such as an odometer of the final balances to enhance the display. Ultimately, creating the audit in Digipound was the culmination of intense research and imagination; it is the main element that sets Digipound apart from the current stablecoins and cryptocurrencies.

## 4.3 Ethereum Node

With the main components of the web application created, it was important to find a way to connect it to the Ethereum blockchain to start trading Digipound. In order to use the functions in the web3 API and actually connect to Ethereum (in this case, the Ropsten test network), a local Ethereum node has to be set up, in this case, on my local server.

A node is essentially a device that is connected to the blockchain network that facilitates a variety of tasks, including creating transactions. In essence, the node is a client which establishes a peer-to-peer (P2P) connection with the Ethereum blockchain. Its tasks include signing and broadcasting transactions, working with smart contracts, and mining blocks. It is necessary to setup an Ethereum node to fully demonstrate the MVP.

The Ethereum node will interact with the smart contract for Digipound. Since the token maintains the ERC-20 standard as per the inclusion of functions in the smart contract, the node can directly determine if transactions sent from the application are valid before validating that block. Whenever a transaction is called that uses the smart contract (essentially, every potential transaction in the issue-trade-redeem cycle of Digipound, as this involves sending ERC-20 tokens), a *full node* keeps track of the current state in the blockchain and performs all of the instructions in the contract necessary to ensure it reaches the correct ensuing state [31].

An example of such a node - the most popular blockchain client - is called **Geth**. Geth is run on an RPC server, which stands for remote procedure call, executing the Ethereum node in the port 8545 [32]. For context, the application for the MVP is running on the localhost:8000 server. I can connect Geth to the Digipound application via the calls made by the web3 API in Digipound. The call in the terminal which sets up the Ethereum node is as follows:

```
sudo geth --testnet --rpc --rpcapi  
"db, net, eth, web3, personal" --rpccorsdomain "*"
```

```
--rpcaddr 127.0.0.1 --rpcport 8545 console
```

I will now break this command down. The sudo command allows me to run the command with the security privileges of a superuser, which is necessary because I have been running this application on my local profile on my laptop. After Geth is called, I specify that I want to access a full node of the testnet. I specify the address 127.0.0.1:8545 for the remote procedure call, which I then connect the web3 API with. Finally, the command `-rpcapi "db, net, eth, web3, personal"` specifies API calls that can be made, such as those in the web3 library (web3), enables trading over the ethereum network (eth), and states that it is modifying the local database (db) to store the confirmation strings of the Ethereum transactions, among others.

```
c26821fdb1555 ip=127.0.0.1 udp=30303 tcp=30303
INFO [06-06|22:54:53.456] Started P2P networking self=enode://
b8e30c838d2f338c8a5d0fbe7824dda0c82cc7f8b8d383aa5cba588782a7c786af54958a01cab1db
49f44b6f6b335bea30d917914e8685eaa7de8a3497bcfbeb@127.0.0.1:30303
INFO [06-06|22:54:53.465] IPC endpoint opened url=/Users/sh
ravannageswaran/Library/Ethereum/testnet/geth.ipc
INFO [06-06|22:54:53.469] HTTP endpoint opened url=http://12
7.0.0.1:8545 cors== vhosts=localhost
Welcome to the Geth JavaScript console!

instance: Geth/v1.8.26-stable/darwin-amd64/go1.12.3
modules: admin:1.0 debug:1.0 eth:1.0 ethash:1.0 miner:1.0 net:1.0 personal:1.0
rpc:1.0 txpool:1.0 web3:1.0

> INFO [06-06|22:54:54.526] Finished upgrading chain index type=bloomb
its
INFO [06-06|22:55:12.679] New local node record seq=59 id=9c1
c26821fdb1555 ip=146.169.197.65 udp=30303 tcp=30303
```

**Figure 4.8:** Geth makes a connection to the testnet blockchain and contains a comprehensive history of blocks. It allows me to write transactions to the Ethereum (Ropsten) blockchain from a local device, where the MVP of the application is running.

Now that a local node is running, the next step is to identify its address as the provider for the web3 API. The web3 API can be run from the same server as geth. This is accomplished through a declaration at the beginning of the application: `web3 = new Web3(new Web3.providers.HttpProvider('http://localhost:8545'));` I use the web3 object throughout to access and modify the full local node at the server specified, which corresponds to where Geth is running.

At this stage, everything was setup. The capabilities of the decentralised web were able to be utilised through the Digipound application. The next step was verifying that the application worked as intended through a series of tests.

# Chapter 5

## Security

At this point, the main objectives of my project had been completed. The issue-trade-redeem cycle for my ERC-20 token was complete, as was a functioning audit system which set it apart from other cryptocurrencies and stablecoins because it is updated in real time and verified by two reputed third parties. When presenting it to my supervisor, I realised with him the potential of Digipound and understood that it could one day potentially store a large amount of assets. Therefore, a natural extension was to address the security concerns associated with an application of this potential.

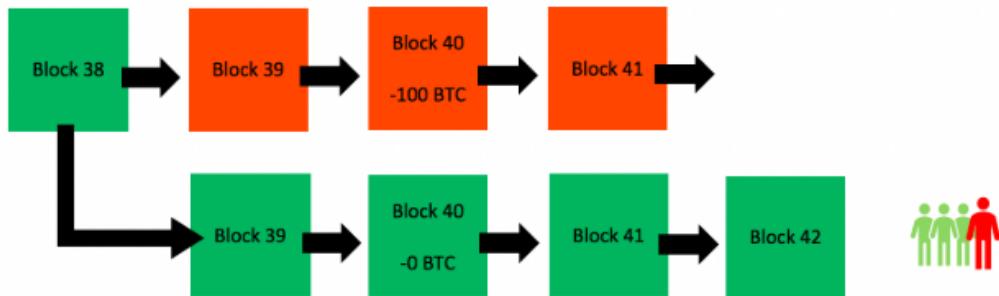
In terms of security, there were four main concerns which my supervisor and I recognised. The following are defenses or explanations that are used to assess each breach.

### 5.1 Block Reorganisation Attacks

**Problem.** A block reorganisation attack occurs when a malicious miner makes a public transaction to a merchant on the blockchain, but it gets overridden after the merchant releases the product to the hacker. This is because the attacker also privately mines along a blockchain fork from the initial current block [34]. The attacker generally has an extensive amount of hashpower, with some attacks being referred to as *51% attacks*, because the attacker has acquired a majority of hashpower on the blockchain.

The theory is that when the attacker makes a transaction to a merchant over the blockchain, the merchant will see it at send the particular product as expected. However, at the exact same time, the attacker has created a ‘fork’ in the blockchain. The attacker mines blocks on a private chain, starting at the original current block (before the public payment was made), at a much faster rate than public blocks can be added to the attacker’s public transaction, due to the attacker’s overwhelming

amount of hashpower. Given that the attacker has a majority of hashpower, at some point, the attacker is able to mine a private chain along one side of the fork that is longer than the other side where the original transaction was made. Here, the attacker will release this chain, override the other side of the fork because the private chain is larger, and regain the coins used to make the payment of the product that has already been received. The blocks on the other side of the chain are now referred to as ‘uncle’ blocks, and are now uncommitted [35].



**Figure 5.1:** The orange chain represents the public chain. On the other side, an attacker has double-spent the amount of cryptocurrency used in Block 39 and 40, mining blocks on a private chain at a faster rate than blocks are being mined on the public chain. Releasing the private chain allows the miner to regain the coins used for the original transaction, as the shorter public chain is abandoned [35].

**How can this affect Digipound?** On all stages of the issue-trade-redeem cycle, a 51% attack can theoretically occur. The stage which could be the most vulnerable to it, and also impact Digipound the most, is the redeem phase. The scenario is as follows: a user makes a request to redeem Digipounds, and a transaction block gets created, with a valid transaction hash, and added to the blockchain. Naturally, the application issues the payout via Stripe, and the user receives GBP. However, if this user performs a 51% attack, the block burning the user’s Digipounds will ultimately be abandoned, and the Digipound user will end up with both DGP and GBP in return.

This will be disastrous for the audit, as there are now more DGP that have been issued than GBP in the Digipound reserve account, so the value of DGP is compromised for all users, and the miner ultimately gets fiat currency for free.

**How can this be avoided?** Most attacks like these can be avoided through waiting for confirmations after a block has been mined. These are confirmations that the block has been added to the chain and other blocks have been added after it. In the current redeem function in the Digipound application, no such confirmations were made, as I originally called for the payout after the block was created in memory.

After talking to my supervisor, I realised that requesting confirmations that the block has been added should precede the creation a payout in Stripe. In theory, since an attacker would have a majority of hashpower in the blockchain, there is no minimum number of confirmations that can fully prevent such an attack. However, having to wait for multiple confirmations before receiving GBP can be very expensive for the attackers, and could cause them to abandon the attack especially if the number of confirmations they have to wait through is quite large.

As such, I implemented this feature in my redeem function. One confirmation means that one block has been added on after the transaction that the Digipound user made, two confirmations means that the application waits until two blocks have been added on before creating the payout in Stripe, and so on. The way I implemented waiting for confirmations used two web3 functions: `getTransaction` and `currentBlock` [26].

When the code in the original redeem function is run, a transaction with a transaction hash is created which represents the burning of a user's Digipounds as they return back into my original address. This transaction hash can be passed into the first function and get information about the transaction, the most important of which is the index of its block along the blockchain. The latter function allows me to find the index of the current block on the Ethereum (Ropsten) blockchain; hence, the number of confirmations received is equivalent to the current block's index minus the index of the block of the user's transaction at the point of creating the payout in Stripe. In order to wait a certain amount of confirmations before preceding, I placed the above calculations in a while loop which would run until the previous value equalled a certain number. Finally, I did not want to block the execution of other tasks in my application, such as other users investing in Digipound, while waiting for confirmations, so I wrote asynchronous calls to both the web3 functions by adding error first callbacks.

```
var transaction;
web3.eth.getTransaction(txHash, function(err, result) {
    if (!err) {
        transaction = result;
    }
});
while (confirmations < 3) {
    var currentBlock;
    web3.eth.getBlockNumber(function(err, result) {
        if (!err) {
            currentBlock = result;
        }
    })
}
```

```

});  

confirmations =  

    transaction.blockNumber === null ? 0 :  

        currentBlock - transaction.blockNumber;  

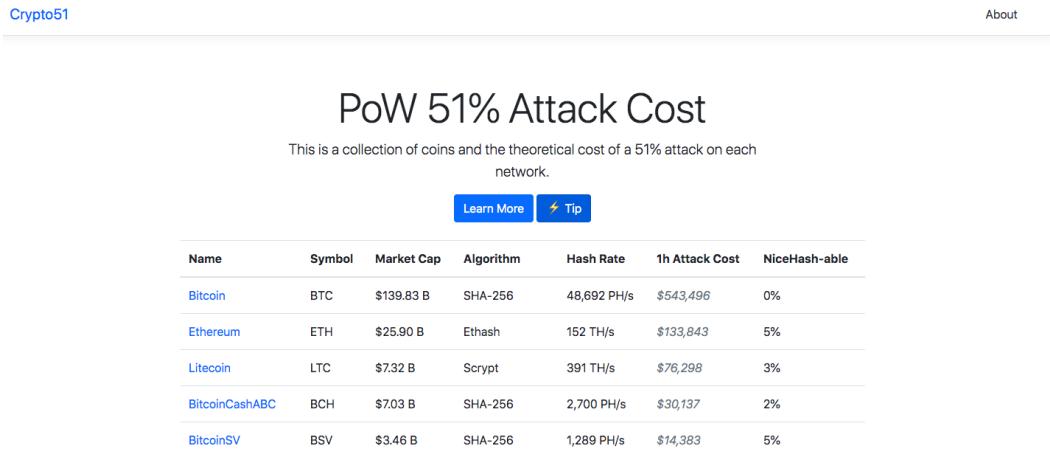
}

```

**Listing 5.1:** After a redeem transaction is created, to ensure it is added to the blockchain and the appropriate amount of GBP can be safely paid out to the user, the function waits for 3 confirmations before proceeding.

As intended, this provided a much more secure way to redeem Digipounds. Three confirmations are now required before proceeding, and the number was chosen under the influence of Ethereum founder Vitalik Buterin. He stated “...only a small number of extra confirmations (to be precise, around two to five) on [a fast blockchain (such as Ethereum)] is required to bridge the gap” between an attackers privately-mined fork and the public chain in a block reorganisation attack [36].

The Ethereum blockchain, itself, represents a particularly safe and established way to store Digipound-related blocks. As stated above, it is quite a fast-moving chain, currently supporting 15 transactions per second, providing good defense against 51% attacks given three confirmations of a transaction. Another statistic that supports this is that, according to Crypto51, Ethereum has the second highest cost of achieving 51% hashpower at \$133,843 [37]. In fact, the only blockchain with a higher cost, Bitcoin, has never been subject to a block reorganisation attack. Thus, the fact that Digipound is an ERC-20 token traded on the Ethereum (Ropsten) blockchain also means that is it naturally less prone to such attacks compared to less established and smaller tokens traded on other blockchain-based platforms.



**Figure 5.2:** The above chart implies that it would only be worthwhile for an attacker to invest in attacking the Ethereum blockchain if the assets the attacker is trying to double-spend are worth over \$130,000. The previous statement also assumes zero confirmations for each stage, and the value is much greater if the attacker has to wait for confirmation [37].

## 5.2 Private Keys

**Problem.** When a user sets up a wallet address to store ERC-20 tokens, both a public key and a private key are generated which the user must hold onto. The public key allows transactions to be sent to a user, and the private key provides security when users send a transaction as they are required to sign it with their private keys. If gained access to, these keys allow a hacker complete control of one's cryptocurrency assets.

**How can this affect Digipound?** This can affect users of Digipound as they could lose their tokens if their keys are not held securely.

**How can this be avoided?** In the current demonstration of Digipound, users can use a browser extension to store their private keys, such as Metamask, which stores a user's private keys in the browser. As such, Metamask also offers an extra line of confirmation to sign a transaction when a user sends Digipounds, whether in the trade or redeem phase of the cycle. Given that Digipound is an ERC-20 token, it can be stored across a variety of wallets, including *hardware wallets*. These are physical devices that are used to store private keys in an encrypted offline environment. An example is Ledger Nano S, which supports thousands of cryptocurrencies across multiple blockchains including Bitcoin and Ethereum, meaning it can support Digipound as well.

## 5.3 Breach of Smart Contracts

**Problem.** There have been many recent, high-profile smart contract hacks, such as the attack on the DAO, a Decentralised Autonomous Organization comprised of numerous smart contracts to democratise Ethereum ICOs. A hacker stole 3.6 million Ether through identifying the possibility of using the ‘recursive Ethereum send exploit’ in the code of the smart contracts. Whilst analysing a piece of the Solidity codebase, it became clear to the hacker that when calling a particular function, `split`, which sends Ether to the user, the hacker would recall the function again before the reward went through and updated the hacker’s balance. This results in transferring more tokens to the hacker than is permitted. These hacks went untracked for a lengthy period of time. Similar hacks have occurred by exploiting vulnerabilities in smart contracts [33].

**How can this affect Digipound?** In a staggering development, it was estimated that 45% of contracts in the Solidity language that are deployed to the Ethereum network are vulnerable to similar attacks. Given that the average smart contract in Ethereum holds about \$4,531 worth of assets, it is clear that a breach of the Digipound contract in the long run could not only disrupt a large quantity of money, but could also threaten the stability of Digipound [38]. This is because tokens will be incorrectly issued without having received the correct amount of fiat currency. Especially if Digipound gains a high market share among stablecoins, if a surplus of DGP to GBP would be identified, this would be disastrous as it disproves the valuation of the cryptocurrency.

**How can this be avoided?** There are practices to make the code of a smart contract more secure, but this can be examined in an extension of this project and can be implemented before deploying the Digipound contract to the live Ethereum blockchain. This project was not particularly focused on changing the scope of the ERC-20 contract, as a feature which secures the valuation of Digipound is the real time auditing system. The audit provides a great defense against attacks to the smart contract, as the instant a transaction issuing Digipounds is made that does not send GBP into the Digipound reserve account, the audit will reflect the lack of fiat currency backing. I can handle this situation immediately and restore Digipound’s valuation, as explained in the following chapter of the report.

## 5.4 Losing Keys

**Problem.** Sometimes, users can outright lose their private keys, meaning all access to their tokens is lost forever.

**How can this affect Digipound?** This can affect users of Digipound as their DGP

---

holdings could be lost forever.

**How can this be avoided?** In truth, this sparked a great deal of thought. I considered adding support for this through Digipound, whether through a social network storing of private keys or a backup database where users could store their private keys if they desired (which many would understandably not desire). However, I realised that both of these options truly diluted the concept of Digipound. This is because Digipound is not an application meant for users to manage their keys, like a hardware wallet is, for example.

Digipound is first and foremost a one-to-one bridge from one's wallet of fiat currencies to one's decentralised wallet of cryptocurrency. In this mode of thinking, my supervisor likened users losing their private key to them "throwing their physical wallets off a bridge." In this manner, Digipounds can be thought of as "digital cash." It is a valuable and unique asset to have, but a rule of thumb is this: people should not invest in Digipound more than the maximum amount of cash they would feel comfortable carrying in their physical wallets.

# Chapter 6

## Testing and Edge Cases

Now that functionality of both the audit and the issue-trade-redeem cycle had been implemented in the website, it was important to run tests and determine how to handle edge cases: potential problems that could occur on each of the stages in the stablecoin cycle. These tests were manually performed to ensure completeness of the application.

### 6.1 Issue

The proper cycle of issuing Digipounds is the following:

1. The user requests a positive amount of tokens to enter that is less than the total supply of tokens.
2. The user inputs a valid bank card and hits submit.
3. A charge is created in Stripe which returns a successful status.
4. Upon loading the home view, the ensuing amount of DGP are transferred on the blockchain, returning the ID of the Ethereum transaction as confirmation.

Testing on a variety of bank cards that were provided by Stripe and with numerous acceptable amounts always resulted in the correct outcome.

However, in the case of issuing Digipounds, there are multiple edge cases present in a transaction:

1. A user enters an invalid bank card and the payment cannot be processed.
2. A user enters a negative amount to pay.



**Figure 6.1:** Whenever a transaction is successful (in this case, issuing 2 DGP), its ID on the Ethereum blockchain is flashed on the home view so a user can look it up on a block explorer such as Etherscan.

3. The program crashes after the payment has been processed but before the transaction can be sent to the Ethereum blockchain, or the Ethereum transaction does not go through.

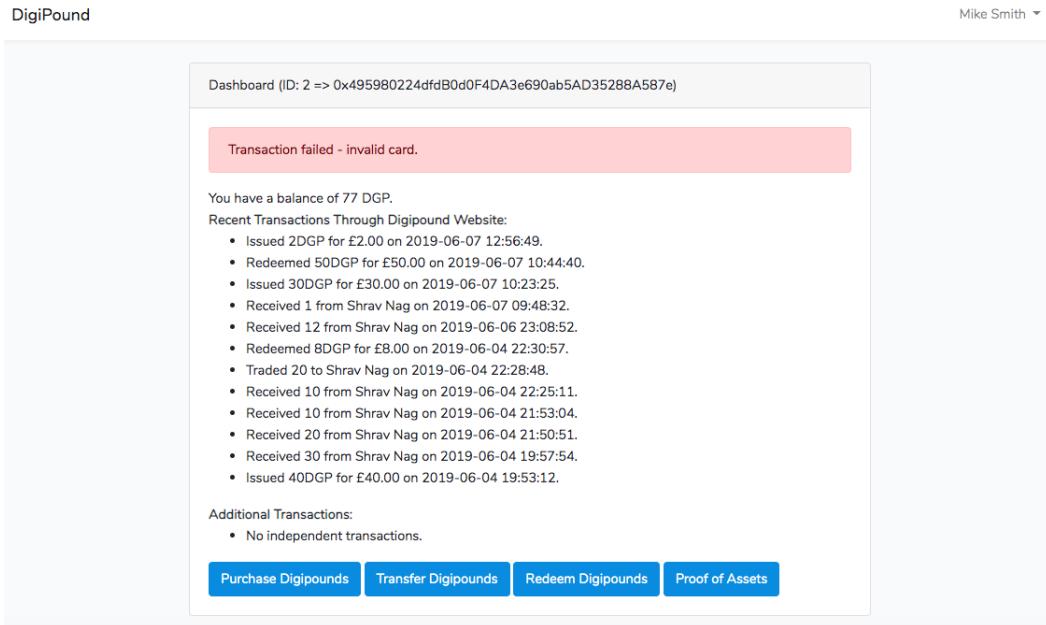
Each one of these cases was tested. The first case was particularly straight-forward to test manually, given Stripe has extensive documentation related to testing. Stripe provides developers with numerous fake cards for testing, some of which are successful and others which intentionally fail for specific reasons.

4000 0000 0000 0002	Charge is declined with a <code>card_declined</code> code.
4000 0000 0000 9995	Charge is declined with a <code>card_declined</code> code. The <code>decline_code</code> attribute is <code>insufficient_funds</code> .
4000 0000 0000 9987	Charge is declined with a <code>card_declined</code> code. The <code>decline_code</code> attribute is <code>lost_card</code> .
4000 0000 0000 9979	Charge is declined with a <code>card_declined</code> code. The <code>decline_code</code> attribute is <code>stolen_card</code> .
4000 0000 0000 0069	Charge is declined with an <code>expired_card</code> code.
4000 0000 0000 0127	Charge is declined with an <code>incorrect_cvc</code> code.
4000 0000 0000 0119	Charge is declined with a <code>processing_error</code> code.
4242 4242 4242 4241	Charge is declined with an <code>incorrect_number</code> code as the card number fails the <a href="#">Luhn check</a> .

**Figure 6.2:** There are a series of test cards Stripe provides to return errors in creating charges, hence used to test that Digipounds do not get transferred if the GBP payment does not succeed.

In the second case, all that was required to test was to enter an invalid number of tokens to issue, such as -1. Since both of these cases would return errors even before a request to the web3 API and the Ethereum blockchain was made, I was testing to ensure the home view was returned immediately with an appropriate error message and the audit remained untouched.

Both these cases were extensively tested with a variety of failed cards and amounts and passed each time.



**Figure 6.3:** When inputting the bank card 4000 0000 0000 9987 to purchase Digi-pounds, Stripe throws a `lost_card` error and the home view is immediately returned, flashing the appropriate error message.

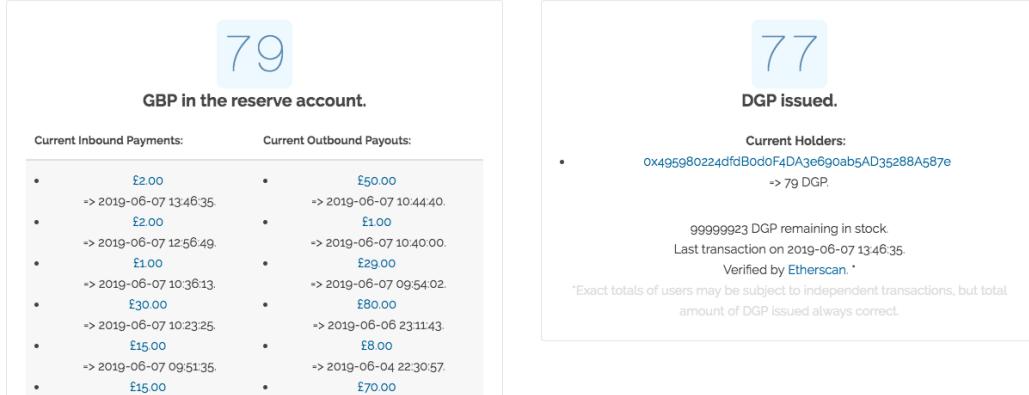
The final case is harder to test manually as it stems from a system error rather than a user error. A system error could come from the application itself, or perhaps result from an attack on the blockchain that prevents Digipounds from being issued.

To model this, I wanted to see that the pounds have been received into the account but the Digipounds have not been transferred, due to an error uploading the transaction to the Ethereum blockchain. To test this, I targeted my local Ethereum node, Geth, and aimed to quickly terminate the process using `Ctrl + C` on the terminal after a successful payment had been made through the DigiPound application.

```
[> exit
INFO [06-07|14:50:03.780] HTTP endpoint closed url=http://12
7.0.0.1:8545
INFO [06-07|14:50:03.783] IPC endpoint closed url=/Users/sh
ravannageswaran/Library/Ethereum/testnet/geth.ipc
INFO [06-07|14:50:03.787] Blockchain manager stopped
INFO [06-07|14:50:03.787] Stopping Ethereum protocol
INFO [06-07|14:50:03.788] Ethereum protocol stopped
WARN [06-07|14:50:03.791] Node data write error err="state no
de 5c452f...21b066 failed with all peers (0 tries, 0 peers)"
INFO [06-07|14:50:03.789] Transaction pool stopped
CRIT [06-07|14:50:03.792] Failed to store last header's hash err="leveldb:
closed"
INFO [06-07|14:50:03.793] Imported new state entries count=134 el
apsed=1.082ms processed=23296627 pending=4431 retry=0 duplicate=208 unexp
ected=895
CRIT [06-07|14:50:03.793] Failed to store fast sync trie progress err="leveldb:
closed"
```

**Figure 6.4:** Terminating Geth through the exit command (Ctrl + C) relinquishes the connection to the Ethereum blockchain and prevents transactions such as issuing Digipounds to be written locally.

I ran a test case purchasing 2 Digipounds with the correct card details, waiting for the payment to arrive, and then immediately closing the connection to Ethereum (it took a many tries because the call to the web3 API occurs nearly instantaneously after a payment has been submitted). As expected, it sent the audit into a frenzy.



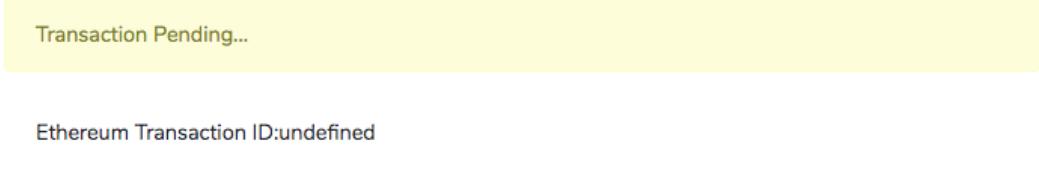
**Figure 6.5:** The audit no longer displays a 1:1 ratio, given that the recent £2.00 transaction was submitted and 2DGP were not issued.

One option to deal with this failure is to shut down the DGP servers to assess the situation. However, it is not necessary to do that because there is now a state of surplus of GBP to DGP, so technically, Digipound has more than enough backing than is expected. Its value is hence not compromised, as all users can theoretically

redeem all of their tokens at the same time, so this situation does not warrant a shut down yet. Ultimately, there are two types of errors that the audit can sense:

1. Surplus: more GBP than DGP (as in this scenario).
2. Deficit: more DGP than GBP (can be observed if a redeem fails).

Clearly, the latter error warrants a shutdown, because the valuation of Digipound has now been compromised. However, in the first scenario, given that it is equally rare, all that is required to do is a quick investigation on which users did not receive the Digipounds they purchased. Then, these Digipounds can be manually transferred from a wallet connected to the DGP reserve account. This is made simple with the models already set up. The transaction database records the Stripe transaction IDs of both charges and payouts; now it is also necessary to record the ID of the Ethereum transaction which transfers DGP, as mentioned above.



Transaction Pending...

Ethereum Transaction ID:undefined

**Figure 6.6:** web3 returns ‘undefined’ when a transaction cannot be written to the blockchain due to the termination of Geth.

This error case can be resolved as follows. Given a surplus - which can be referred to as ‘Code Yellow’ - I can go through the local database of Digipound transactions and identify the ones with an ID beginning with ‘ch’, or charges. Then I can determine the associating ID of the Ethereum transaction, and if it is undefined, it means that it pertains to this error case. I am able to identify the amount of this transaction in the model, as well, and manually send that amount of Digipounds over to the user who made the transaction. Although there is not an automatic solution to this, given that this is a rare error, the previous implementation resolves the issue and restores the audit.

In this manner, all edge and error cases were tested and passed when issuing Digipounds.

## 6.2 Trade

There are three main edge cases when considering trading Digipounds via my application:

1. A user enters an invalid email address.
2. A user enters an invalid amount to transfer (negative or above their current balance).
3. The transaction is not correctly written to the Ethereum blockchain.

Again, the first two cases will throw errors before the request to write a transaction to the Ethereum blockchain is made. As stated in the Implementation section, multiple if checks are made before calling the web3 API and updating the local Transaction model. Therefore, just like in the issue phase, if one of the checks fails, the home page is reloaded immediately and flashes an error message.

```
$recipient = User::where('email', $_POST['email'])->first();
if (!$recipient) {
    $request->session()->flash(
        'error', 'Transaction failed: Invalid recipient.');
}
```

**Listing 6.1:** The controller class handling the transfer of Digipounds checks the User model to ensure that a valid recipient has been entered based on the email address. If not, the home view is returned immediately, flashing an error message.

As such, all invalid test transfers according to the first two cases failed correctly.

For the third case, when the actual transaction itself is not written to the Ethereum blockchain, I considered whether or not to interfere. However, I realised that if a trade fails, since transfers currently only occur between two registered Digipound users and do not involve any fiat currency being moved, no additional DGP or GBP will enter or leave the audit. This error has no ramifications to the rest of the Digipound platform, hence executive action does not need to be taken. The only action the website does is display the Ethereum transaction ID as ‘undefined’ and suggest the user tries the trade again. To reiterate, the previous error is harmless because no GBP or DGP has left or entered the reserve accounts untracked.

## 6.3 Redeem

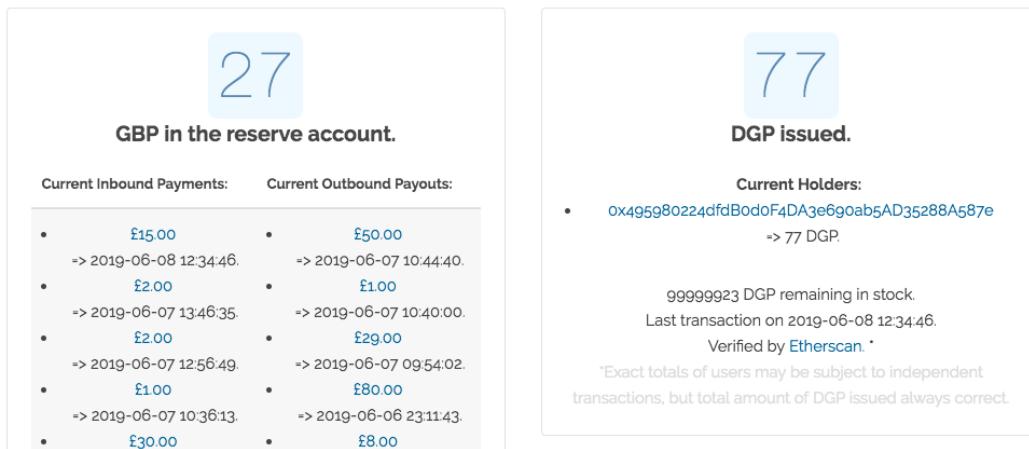
In order to test the redeem phase of the cycle, three edge cases were considered:

1. A user enters an amount to redeem greater than the user’s current balance.
2. A user enters a negative amount to redeem.

3. A user receives a payout from Stripe, but the transaction of burning the user's Digipounds does not go through.

In the first two cases, just as in the previous stages of the cycle, manual tests were made to ensure the application did not proceed further should an invalid amount to redeem be entered.

The final edge case required some thought on how to respond. In this situation, a deficit of GBP to DGP would be present, which is quite a pressing error. The value of DGP is compromised. One of the main causes for this case is a block reorganisation attack, as mentioned earlier. However, given the limited hash power I had, along with the aforementioned feature I added to wait for confirmations of the block being added before processing the return, I was unable to attempt a block reorganisation attack to manually test this edge case. So, as another way to model a deficit, I manually issued an arbitrary payout to a test user's bank account on the Digipound system. Afterwards, the audit reflected the deficit. This models the outcome of a blockchain attack that creates a transaction but surprisingly does not get committed, to which the payment of GBP still gets issued in response.



**Figure 6.7:** The third edge case results in the audit showing a deficit, which, although extremely rare, can be a major concern for users.

Unlike an earlier situation described where issuing Digipounds fails on the blockchain, this situation requires immediate action. Should the audit show a deficit, I will now be pressed to quickly and temporarily shut the system down. Rectifying this situation, however, has a similar approach. In this case, the ID of the payout has been added to the local database as a part of the Transaction model, and hence is also associated with the ID of an Ethereum transaction. Though this ID will not be 'undefined,' it will not correspond to a committed transaction, which can be confirmed

through the following asynchronous call to the web3 API:

```
web3.eth.getTransaction(txHash, function(error, result) {
  if (!error) {
    console.log(result.blockNumber)
  }
});
```

**Listing 6.2:** The `getTransaction` method can be run with an Ethereum Transaction Hash (`txHash`) to determine if the associated transaction hash corresponds to a committed transaction. When calling `blockNumber` on the result, a `null` value ensures that this transaction was not committed to the blockchain and has contributed to the deficit in the audit.

Now that the payout transaction associated with the uncommitted Ethereum transaction has been identified, there are two ways to proceed with this:

**Cancel the payout via Stripe.** Through Stripe, most payouts aim to be delivered into the user's bank account by the end of the day they were issued. However, Stripe offers a feature called *rolling payouts*, which delays the deposit of a payout. Per Stripe's website, "Most banks deposit payouts into your bank account as soon as they are received, though some may take a few extra days to make them available [28]."

An option can be to institute a *2-day rolling phase*, meaning that although the payout will be immediately created in Stripe when a Digipound user requests to redeem DGP, the payment will only be sent after two days. This provides a forty-eight hour buffer in case the corresponding Ethereum transaction did not commit. Here, I can identify the payout to be cancelled, and directly notify the bank to cancel the payment, thus restoring the audit. I can also notify users via their email addresses, saying that the previous transaction was unsuccessful and prompting them to try again.

This has the benefit of reversing invalid redeem transactions, but also the drawback of having users wait for two days to receive their Digipounds, which is not practical in the long run, when the number of users and transactions on the platform will increase exponentially. Then, users would demand their transactions to be instant. Yet, this option is practical in completing the proof-of-concept of the real time auditing system, since both Digipounds and the GBP in the reserve account are still moved at the call of the redeem transaction.

**Try to restore the transaction.** Currently, the block that has not been added to the Ethereum blockchain, containing the unconfirmed transaction, represents an uncle block. In the Bitcoin network, these uncle blocks are removed from the database;

however, Ethereum keeps a pool of uncle blocks and incentivises miners to mine these unconfirmed blocks. Thus, although quite difficult, there is still an ability to ultimately commit the transaction. However, this option is outside the scope of this project, given the time constraints and lack of experience with uncle blocks.

At this stage, the issue-trade-redeem cycle and its connection to the auditing system were extensively tested. Along with qualitative tests that were run on potential users with regards to the UI and ease of use (see Evaluation section), this ensured that Digipound is a complete, tightly-valued, and unique stablecoin, contributing to the goals of my final year project.

# Chapter 7

## Evaluation

As the nature of my final year project is very applied, building upon lots of current technologies, I believe honest evaluation is necessary to understand if Digipound can be a widely-used stablecoin in the future. Given the dual elements of my project: the Digipound cryptocurrency (DGP) and the Digipound web application, I felt that each element should be evaluated accordingly, under the premise that the main objectives for the project were successfully worked towards.

To start, I shall reiterate some of the objectives of this project and evaluate its components against it.

- To design a secure and accessible infrastructure for a cryptocurrency to be issued, traded, and redeemed, whose value is tied to that of a fiat currency.
- To guarantee this valuation by creating an auditing system that updates in real time and is backed by third parties.

### 7.1 Cryptocurrency

The Digipound (DGP) token itself is a standard ERC-20 token, yet my project aimed to satisfy the above ambitious objectives using it.

#### 7.1.1 Cycle

**Positives.** As the creator of the Digipound token, I own an Ethereum address which initially held the entirety of the Digipound supply. As such, I was able to handle the issue and redeem stages of the cycle by using the smart contract written. In terms of what was in between, the fact that Digipound is an ERC-20 token means that it is very versatile on where it can be traded and spent, whether within the Digipound application or in a variety of potential exchanges.

For my project, I was able to model the issue-trade-redeem cycle of Digipound on the Ropsten test network of the Ethereum blockchain. This was very important as it was essential for my token to be used just like any other cryptocurrency, such as Bitcoin or Tether. The issue-trade-redeem cycle was the foundation for my project, as it allowed me to set up a fully-functioning cryptocurrency which led me to break new ground in the real time auditing feature.

**Drawbacks.** As stated, my smart contract was deployed on the Ropsten test network, not the Ethereum main network. This meant that all transactions were not subject to Ether cost, as previously described. The Ether cost is attributed to the *gas* of a function, or the computational effort required to write a transaction. Ether is the token which is used as a fee and will be charged on all stages in the issue-trade-redeem cycle. This means that users will have to hold investments in Ether before investing in Digipound.

It is better, however, that the transaction fees are taken in Ether and not Digipound, as transaction fees are variable based on the gas and the amount of the transaction. Varying DGP costs would cause the value of Digipound to be compromised. For example, processing a 10DGP transaction and a 2DGP transaction should not come with a fee in Digipounds, as the 10DGP transaction would require a greater fee of Digipounds. Hence, the value of DGP will also be related to how much DGP was exchanged, not just the value of the GBP.

To resolve this drawback, once Digipound is brought to the main network, I will encourage users to invest in a one-time purchase of Ether that is enough to cover a certain amount of transactions (such as 0.15ETH, approximately 40USD). For context, the average transaction value on Ethereum is about 0.0007ETH [40]. Hence, 40USD is enough for about two-hundred transactions. Users can liken this one-time cost to the cost of a physical wallet to hold cash, which, in a way, further models Digipound against fiat currency.

### 7.1.2 Valuation and Audit

**Positives.** One of the main motivations for this project was to improve upon stablecoins such as Tether and the Gemini Dollar for their untrustworthy valuation of their tokens. In the context of the previous examples, this is because Tether does not document any of its USD holdings and Gemini Dollar only releases one nondescript audit every month. In fact, a quick look at the Tether market chart reads that it is currently trading for \$1.01, which should worry investors given that its \$3B market cap relies on having 1:1 reserves of USD [39].

Thus, the creation of a thorough auditing system backed by two third-parties serves

---

as an upgrade to potential investors. Additionally, the fact that the audit is updated in real time adds tremendous value, as potential users of Digipound can feel comfortable about investing at any time, not just the beginning of each month in the case of the Gemini Dollar! The audit is what sets Digipound apart from current stablecoin cryptocurrencies, allowing it to be freely issued and traded with the confidence of being able to redeem it at any time at a one-to-one ratio of GBP.

**Drawbacks.** Clearly, the value of DGP will change slightly if moved from a testing phase to a live phase - but this does not relate to the Ethereum transaction fee previously mentioned, as costs on transactions are charged in Ether, not Digipound. What will change is the cost of receiving GBP payments, as the Stripe platform takes approximately a 2.9% commission on all incoming payments made with Charge objects [28]. Moreover, thinking in advance, if I were to make the Digipound application and token live, I would have to consider holding onto a percent of incoming GBP, as well, in order to maintain Digipound as a sustainable business. Naturally, there are many costs associated with taking Digipound live, such as maintaining public servers. Assuming a total of 5% of GBP in the reserve account has to be spent between Stripe and application maintenance, this sets the Digipound token at a valuation of 0.95, or ninety-five pence.

At first glance, it does seem that the previous valuation is a drawback for my project's future. However, the important aspect of valuation is whether it is fixed or fluctuating. Since I intend to take 5% on all Digipound transactions, I can ensure that the valuation of 95 DGP to 100 GBP will remain constant, which is still supported through the auditing system. As long as the audit shows 95 DGP issued for every 100 GBP in the Digipound reserve account, users will still be confident that they can redeem their tokens anytime at the 95% valuation. Thus, the value of DGP is always tied to the GBP, maintaining its identity as a stablecoin which will not be subject to the volatility of other cryptocurrencies.

One particular drawback in the audit is its state in the infinitesimal time in between when a payment is received and DGP are issued in the issue phase and when DGP are burned and a payout is made in the redeem phase. Given the security concerns expressed earlier, there are natural, albeit small, delays, such as when running confirmation checks in the redeem phase. Yet, one of the main features of the audit is its real time updating capability. However, for example, if users somehow view the audit at the moment when a process is running that is accepting confirmations from a block to burn Digipounds pushed by another user, they will witness a misconstrued surplus, as the payout is still waiting to be issued but the Digipounds have already been burned. Although a surplus will not be viewed as unfavourably as a deficit, it will still call into question the value of Digipound.

Though this seems like a rare scenario at the moment, it will be more likely as Digipound scales and has to process a large amount of transactions. A suggestion could be just to update the audit after each transaction, but that would compromise the most unique feature of Digipound: the real time audit. Ultimately, the response to this scenario will have to be carefully thought through as the users on the Digipound platform increase.

To summarise, the ambitious efforts to make DGP a potentially widely-used, versatile, complete, and stable cryptocurrency were mostly successful. Moreover, I am conscious on some of the drawbacks associated with my efforts, which bodes well as I attempt to make this cryptocurrency live in the near future.

## 7.2 Application

Although my project is centered around developing a new cryptocurrency, since Digipound is a stablecoin, the application used to interact with it is an incredibly important component, as well. When evaluating the Digipound website, I was able to use a number of qualitative and quantitative metrics, when testing it not only by myself but with a group of random users, as well as my supervisor.

### 7.2.1 Website

In terms of the website's objectives, I required a platform to allow users to issue, trade, and redeem Digipounds in a straight-forward manner, and to display the audit in an appealing and concise manner.

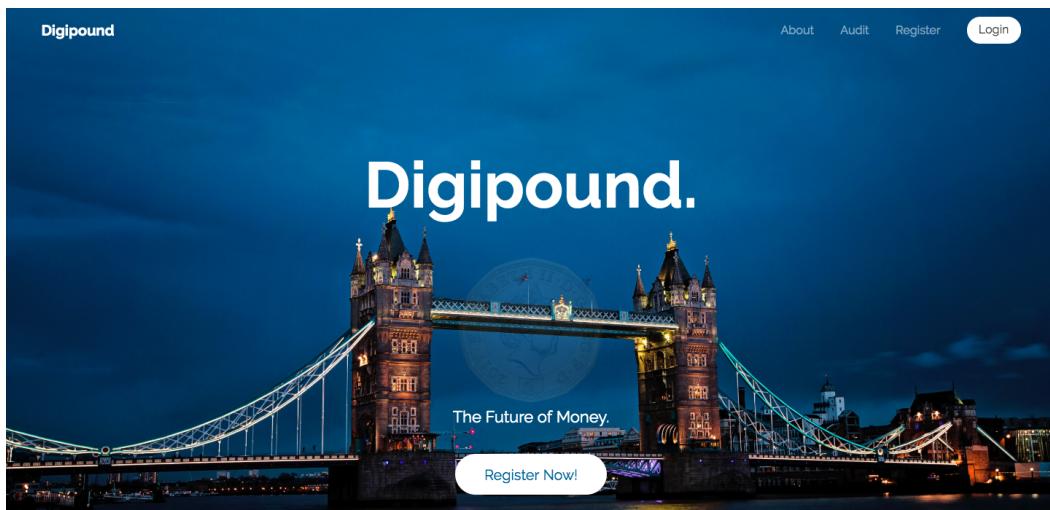
**Positives.** One element of my project I am particularly proud of is the ease-of-use in investing in Digipound. In order to determine this, I used a variety of metrics and consulted numerous potential test users. One of the most successful metrics in my project was the minimum amount of clicks users required to complete functions such as issuing, trading, and redeeming Digipounds and viewing the audit. It is important to have a relatively low number of clicks per task so to not confuse users.

From the second year Web Applications project lectures, I am familiar with the three-click rule, which is that users should be able to perform most tasks in three or fewer clicks. For the aforementioned tasks, the metric ultimately reflected favourably on the ease-of-use of investing in Digipound via the website:

Task	Number of Clicks Needed <sup>1</sup>
Issuing Digipounds.	3
Trading Digipounds <sup>2</sup>	3
Redeeming Digipounds.	3
Viewing the Audit.	0

It was promising to see that the functionality of the Digipound token was able to be used through the Digipound website. Moreover, having spent much effort on the landing page, I made sure to evaluate it based on test user experiences. The landing page contained the following components:

- An introduction banner to Digipound.
- An explanation of the issue-trade-redeem cycle.
- The audit, updating in real time.
- A form to register.



**Figure 7.1:** The landing page describes Digipound as the “future of money” and describes to users how to use the DGP token and Digipound web application.

I consulted ten final year students at Imperial College London, presented them with the landing page, and received feedback for the following questions:

1. **Is the issue-trade-redeem cycle made clear?** .... 90% said yes.
2. **Is the landing page visually appealing?** .... 100% said yes.

<sup>1</sup>from the landing page to the completion of the task.

<sup>2</sup>using the Digipound website; amounts may vary when trading token directly on other platforms.

3. Does the auditing system make sense? .... 90% said yes.
4. Does using the Digipound platform seem easy and beneficial? .... 90% said yes.

Although ten students is a particularly small sample size, I gleaned that the content on my landing page was enriching and truly educated people on the benefits of Digipound, serving as a positive evaluation which strongly influences the next step of making the application live. Along with the tests that were run, the group evaluation made me feel confident in Digipound as an application that users can safely, informatively, and easily trade the stablecoin.

**Drawbacks.** The main drawback actually occurs in the use of the web3 API. Since the API is written in Javascript, admittedly all of its calls are client-facing. This is because JavaScript is primarily used for front-end development. Additionally, there are currently no APIs in place to interact with Ethereum via PHP, the language used in the backend. This would have been ideal as, in my project, when coding the issue-trade-redeem cycle, the function definitions are facing the client.

I had considered using web3 within Node.js, a JavaScript runtime environment, to run the issue, trade, and redeem functions outside of the browser. However, I would have had to run an additional server along with my local PHP server, which is impractical in both the short term and the long term. Although functionally, it makes no difference whether or not the code is run in JavaScript or PHP, it can be slightly unsettling that users can see the code when clicking ‘Inspect Element’ on the website.

At first, this seems like a security hazard. However, because of the fact that I did not automate the signing of a private key, rather giving users an extra layer of protection by allowing them to confirm the transaction of Digipound via a browser extension like Metamask, no transaction can be written without a follow-up confirmation. Thus, functionality and security of transactions are not compromised by this drawback. Yet, it is important to be conscious of it. I intend on scaling the application to reach a large number of users so I would soon prefer to keep the interaction with Ethereum on the back end of my website as is the case with the interaction with Stripe.

At a high level, another drawback of my application is its inability to handle a large amount of users trading at one time. This is to be expected, since the application is being run on a local PHP server on a Macbook laptop that is approximately five years old. I mention this drawback, however, because I realise another area to consider when bringing the application live are the overhead costs of running it, such as hosting and data backup. Admittedly, the latter is not as much of a concern given

all the live transaction records will be stored in Ethereum, as well.

Ultimately, given that the objective of my project was to develop a proof of concept for a revolutionary, functional, and secure stablecoin, I believe the purely application-centered drawbacks do not undermine my progress in the project. Rather, they give me perspective of the realistic challenges that will be faced when trying to scale Digipound, which I believe is a natural extension of my efforts this year.

# Chapter 8

## Conclusion

The culmination of this project resulted in both reflection on the many concepts learned and optimism for the real-world implementations of it.

### 8.1 Takeaways and Lessons Learned

Throughout my project, I learnt a variety of essential skills and unique concepts in computer science.

**Lesson: Blockchain is permanent.** This is a particularly important cautionary tale for any blockchain developer. What happens to be a defining feature of blockchain can also cause great frustration to engineers who are not careful. I admittedly experienced the downside of this in the early stages of my project, when, before creating the official Digipound token, I created a token to understand the process of deploying a smart contract. I had created an ERC-20 account and experimented with transferring tokens to the account, directly through a decentralised wallet, MyEtherWallet. The total supply of the token was quite low, so I experimented sending the entire total supply. However, when entering the recipient address, I accidentally inputted the wrong address by one character - after which I proceeded to lose all of the supply of the test token! Since the transaction was then written to the Ropsten test blockchain, and I did not have access to the mistaken recipient, the tokens were essentially out of my reach forever.

Ultimately, there are many positives of the permanency attribute to the blockchain. Especially in Ethereum, it adds security and completely removes the need for a third party to facilitate and record transactions of assets. So luckily in this case, I was able to create another token, with a new and improved name (Digipound), and was very careful with it throughout the rest of my project. Additionally, when I started to automate and verify transactions through the Digipound website, the permanency of blockchain became less daunting. This was, however, a valuable lesson and will

hold much more weight when I am dealing with live assets on a live blockchain such as Ethereum, as I intend to when I take Digipound live.

**Takeaway: The issue-trade-redeem cycle of stablecoin can be safely and effectively automated in a web framework.** By connecting both the Stripe API and web3 API in my Laravel project, and calling them in the correct order at each phase (for example, issuing Digipound requests the payment first, verifies that it has reached the reseve account, and then issues the tokens), I have effectively implemented the issue-trade-redeem cycle, showing a proof of concept of a new stablecoin, as the title of my project suggests.

Understanding that this was possible set the foundation for Digipound to be assessed on the same standard as stablecoins such as Tether and the Gemini Dollar, which I feel it stands out due to the following conclusion:

**Takeaway: A real time and third-party supported auditing system to verify the value of a stablecoin is feasible in practice.** Digipound is distinguished by the auditing feature I implemented. Being able to call third-parties, both through the application when making the backend calculations of the audit and by listing websites such as Etherscan on the front end to provide users additional proof of assets, I was able to create an effective and thorough auditing system. This is a feature which both Tether and Gemini Dollar, along with all other stablecoins, lacked. Although the audit may have some aforementioned drawbacks in terms of scaling, being able to develop a proof of concept of this is very promising for Digipound's ability to serve as the future of decentralised assets.

## 8.2 Future Implementation

While most projects were designed to be able to be followed up by a PhD or another intensive research programme, the applied nature of mine means that a natural extension would involve demonstrating the application's impact in the real-world. Thus, as my Bachelor's degree comes to a close, and I am proud - and constructively critical - of my achievements in Digipound, I believe that the only way to demonstrate that Digipound is the future of money is, as hinted earlier, to make the application live.

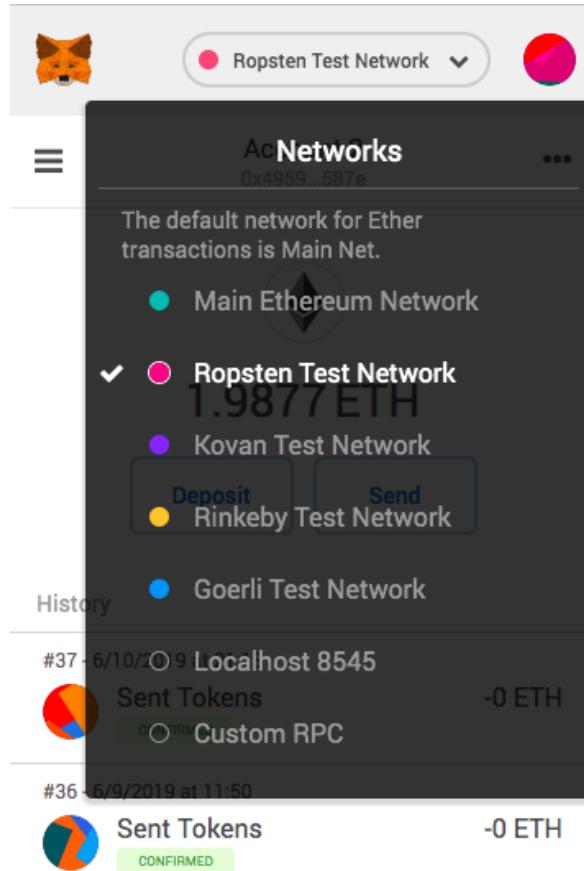
### 8.2.1 Live Application

Making Digipound live is quite simple. It will start with removing both the test restrictions from the payment API and the Ethereum blockchain. This is because the rest of the infrastructure is already in place to have an ERC-20 token trade on the

real Ethereum blockchain, backed up by real fiat currency and verified through an audit in real-time.

In the case of the Stripe API, all that is required is to change the API keys in the environment file. When setting up a Stripe account, a developer is provided with both test keys and live keys, which can be switched at any time to turn the application into a live payment handler.

Putting the Digipound token on Ethereum instead of Ropsten would be just as straight-forward. The exact same contract can be used given that Digipound was specified as an ERC-20 token, even including the name and symbol (a quick check on Etherscan confirms that it is unique). All that is needed is to recompile the contract using Remix, but this time specify the blockchain to be deployed to as Ethereum, not Ropsten. If using Remix with a digital wallet which will hold the full supply of the token, the network can be changed via the browser extension used to connect the wallet, such as Metamask, directly.



**Figure 8.1:** Changing the network from Ropsten to Ethereum requires one click and will allow the Remix IDE to deploy the same Digipound contract to the Ethereum blockchain, to remove the test restrictions on the token.

These changes are aimed to be implemented shortly after my final presentation, where the benefits of Digipound will be demonstrated as a proof-of-concept. Once implemented, Stripe will also start taking commission, as may the Digipound business, hence the valuation will be redefined slightly as described above. Despite that, Digipound can then serve as a platform for people to exchange a guaranteed amount of money for a guaranteed amount of digital assets, which can be redeemed at any time and traded at will. It can serve as the model of “digital cash,” which is an exciting development. Therefore, as “digital cash,” it would be useful to trade it in various places, leading to a natural second extension of my project.

### 8.2.2 Acceptance Among Merchants

Making the Digipound a widely accepted token among merchants will be a natural next step. One of the ways to truly harness the benefits of a guaranteed-collateralised

token like Digipound is to exchange it via merchants with products that have defined values. This gives new meaning to the term e-commerce.

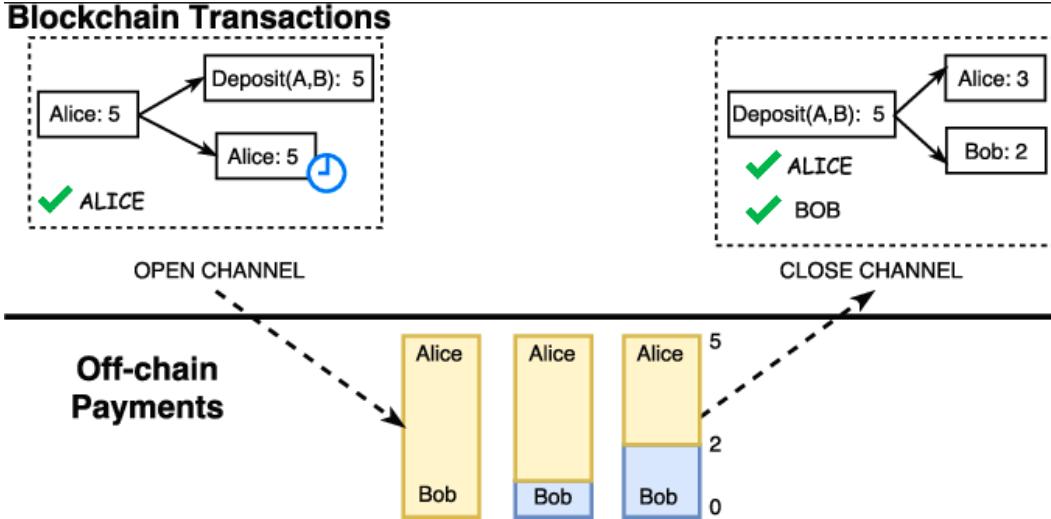
Since Digipound is an ERC-20 token (and will be live at this stage), it already has the capability to be used as payment in a few services. For example, the website management system Wordpress has a plugin called Woocommerce which enables payment in websites. Woocommerce can also be enhanced with a feature called the *Ether and ERC-20 tokens WooCommerce Payment Gateway*, a plugin which accepts payments of ERC-20 tokens [41]. Other infrastructures and features, such as CoinPayments, are available to process online payments of ERC-20 tokens.

Hence, as Digipound gains more users, there is quite a compelling case to convince merchants to accept it as a form of payment, because, compared to other ERC-20 tokens and cryptocurrencies, it is not as volatile. Since merchants know the exact value of a Digipound, they do not even need to haphazardly modify their prices and can be comfortable with the assets they receive, as it is essentially a form of decentralised cash.

### 8.2.3 Payment Channel Network

Ethereum currently supports 15 transactions per second. In this context, when thinking about the future of my project, I also have to think about the future of the decentralised environment. Given its increasing popularity - the market cap of Ether is now 30 billion USD - one can only imagine its market share in the upcoming years (to which Digipound will hopefully contribute). This will potentially mean an increase in transaction fees, measured in Ether, on the Ethereum blockchain.

In turn, the price of transactions could deter users from investing in Digipound. One solution to this is the presence of offline payment channels for cryptocurrencies on the blockchain. A Payment Channel Network (PCN), is an off-chain solution to address scalability of the blockchain. The idea is that PCNs allow two users to make an unlimited amount of transactions offline. They do not have to access the blockchain, except at their first transaction and their last transaction. To explain at a high level, the network runs parallel to the blockchain, containing a special address, where transactions can be written without broadcasting to the blockchain. Before starting the channel, all parties in the channel will input a certain amount of tokens, and when the channel starts, all users in the channel are free to distribute and trade the total amount of tokens as they please. The end state of the payment channel is what is written back to the blockchain, meaning the succession of transactions in the payment channel was not held accountable to the blockchain transaction rate and is recorded privately [43].



**Figure 8.2:** In the context of Digipound, if user A inputs 5 DGP into a channel with user B, then A can make as many off-chain transactions as desired to B involving the 5 DGP, and the only time they interact with the blockchain is when writing the final state of the channel back to the network [42].

The ability to use PCNs would be vital for Digipound's ability to scale. Implementing them is outside the scope of the project but is stated as a valuable resource to have. This is to ensure Digipound users do not get deterred by high transaction fees on the Ethereum blockchain, and can continue to trade DGP in high volume, mimicking fiat currency. This would be another revolutionary development, of which the present work in this project lays a stable foundation for achieving.

Ultimately, the future of Digipound has been thoroughly planned. To be able to turn the knowledge and skills I have gained at Imperial College London into a tangible proof-of-concept that can potentially lead to a business that breaks new ground is quite satisfying. In conclusion, the final result of this project is a working application for a fully-functioning stablecoin test cryptocurrency whose value can be verified by third parties in real time. As described above, the impact, market, and possibilities of this proof-of-concept are very promising and I look forward to developing them further.

# Appendix A

## Additional Functions

```
[{"constant":true,"inputs":[],"name":"name","outputs":[]}, {"name":"","type":"string"}], {"payable":false,"stateMutability":"view","type":"function"}, {"constant":false,"inputs":[{"name":"spender","type":"address"}, {"name":"tokens","type":"uint256"}],"name":"approve","outputs":[]}, {"name":"success","type":"bool"}], {"payable":false,"stateMutability":"nonpayable","type":"function"}, {"constant":true,"inputs":[],"name":"totalSupply","outputs":[]}, {"name":"","type":"uint256"}], {"payable":false,"stateMutability":"view","type":"function"}, ... ... {"indexed":true,"name":"to","type":"address"}, {"indexed":false,"name":"tokens","type":"uint256"}], {"name":"Transfer","type":"event"}, {"anonymous":false,"inputs":[{"indexed":true,"name":"tokenOwner", "type":"address"}, {"indexed":true,"name":"spender","type":"address"}, {"indexed":false,"name":"tokens","type":"uint256"}], {"name":"Approval","type":"event"}];
```

**Listing A.1:** Excerpt of the ABI for the Digipound contract. The entire value is much larger, but the excerpt shows how the contract is represented when calling the web3 API in the Digipound application.

```

protected function create(array \$data)
{
    Stripe::setApiKey(getenv('STRIPE_SECRET'));
    try {
        $bank = Account::create([
            'type' => 'custom',
            'country' => 'GB',
            'email' => $data['email'],
            'business_type' => 'individual',
            'individual' => [
                'first_name' => $data['name'],
                'last_name' => $data['surname'],
                ... /* Additional relevant fields
                      such as address and DOB */ ...
            ],
            'external_account' => [
                'object' => 'bank_account',
                'country' => 'GB',
                'currency' => 'gbp',
                'account_holder_name' => $data['name']
                    . " " . $data['surname'],
                'routing_number' => $data['routing'],
                'account_number' => $data['account_no']
            ],
            'tos_acceptance' => [
                'date' => time(),
                'ip' => $_SERVER['REMOTE_ADDR'],
            ]
        ]);
        return User::create([
            'name' => $data['name']
                . " " . $data['surname'],
            'account' => $data['account'],
            'email' => $data['email'],
            'password' => Hash::make($data['password']),
            'routing' => $data['routing'],
            'account_no' => $acct->id,
            'balance' => 0
        ]);
    }
    catch (\Stripe\Error\Card $e) {
        $request->session()->flash('error',

```

```
        'Transaction failed');
    }
}
catch (\Stripe\Error\Card $e) {
    $request->session()->flash('error',
        'Transaction failed');
}
}
```

**Listing A.2:** Upon registration, a user's bank details are encapsulated as a connected account in Stripe so payouts can be sent to them if they choose to redeem Digipounds.

# Appendix B

## User Guide

### Setting up the Application

The full source code of the Digipound application, including the Stripe API keys, are submitted on behalf of the project. To run the application, install and save the source code. Next, open a new terminal window to install the web3 API to make the function calls. It can be done with the following command:

```
$ npm install web3
```

To run the application locally, ensure that the local port 8545 is unused as it is specified as the provider for web3. Then, a local Ethereum node can be connected by the following command:

```
$ sudo geth --testnet --rpc --rpcapi ‘‘db, net, eth, web3, personal’’  
……—rpccorsdomain ‘‘*’’—rpcaddr 127.0.0.1 —rpcport 8545 console
```

The Digipound application can now be accessed locally. To run a local PHP server, use the following command:

```
$ php artisan serve
```

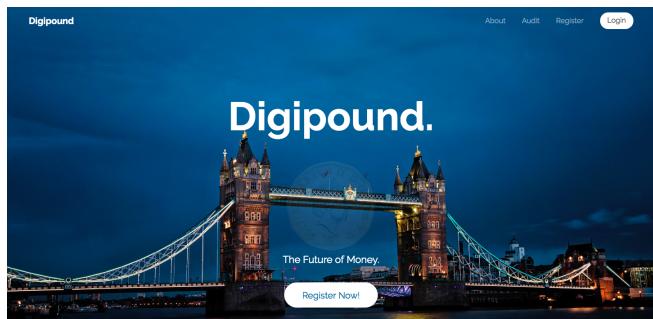
At this stage, Digipounds can be issued, but not traded or redeemed on behalf of the user account. This is because the user has not specified the private key of the ERC-20 account used to sign up to Digipound, so the user cannot yet sign transactions to the blockchain. To proceed, one course of action is to install a browser extension like Metamask, which will save private keys and ask for confirmation anytime a request to trade Digipounds or redeem Digipounds is made from the user account. More information about Metamask and its installation can be found at <https://metamask.io/>.

## **Customisation**

It is also worth noting that the Digipound application can be thought to implement the functionality of a fully-audited stablecoin to an ERC-20 token. In theory, this token can be any ERC-20 token. For example, when making Digipound live, all that is needed is to change the specified contract address variable in the Digipound source code and it will run functionally with a live Ethereum token. Similarly, a developer can set up any ERC-20 token as a stablecoin using the Digipound application code. Simply replace the variable `contractAddress` in the issue, trade, and redeem functions with the address of the contract specifying the token to trade, as well as the `erc20-abi` variable with the new ABI.

Additionally, the developer can also change the keys of the Stripe account to process payments. These keys are located in the `.env` file in the directory. Setting up a Stripe account to handle payments associated with Digipound can be learned at <https://stripe.com/>, and a public and secret key are required to call the Stripe API from an application.

## Using the Application



Welcome to Digipound!  
Join the decentralised  
web in seconds.



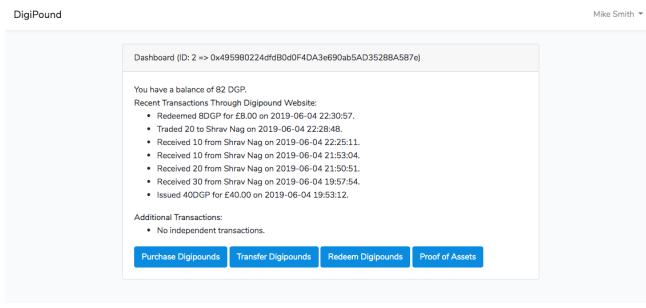
The Digipound application gives users capability to invest in the fully-collateralised Digipound (DGP) stablecoin, which can be issued, traded, and redeemed like any other cryptocurrency.

## Chapter B. User Guide

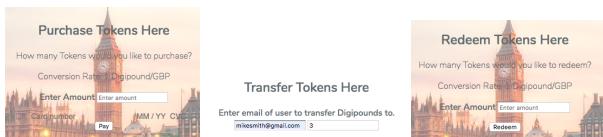
---



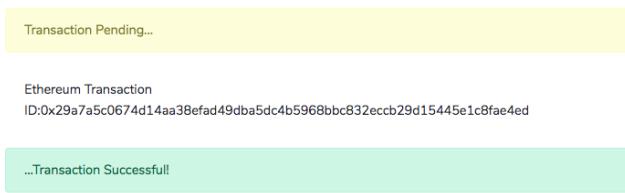
To begin, register with the following details, including an email address and password for login and trading purposes, an ERC-20 address to store Digi-pounds, and a bank address to receive GBP after redeeming DGP.



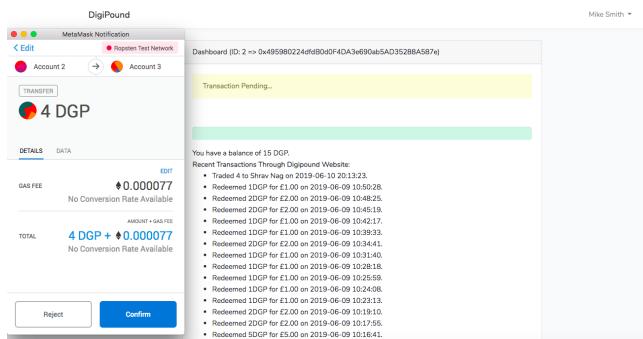
The home page lists a user's current holdings, a list of previous transactions, a link to the audit, and options to issue, trade, and redeem Digi-pounds.



Issuing, trading, and redeeming tokens is very simple with the Digipound application.



You have a balance of 75 DGP.  
Recent Transactions Through DigiPound Website:  
• Issued 2DGP for £2.00 on 2019-06-07 12:56:49.

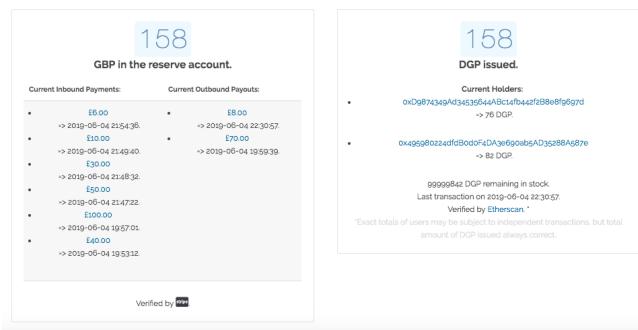


After a successful transaction, the Ethereum ID of the transaction is displayed for easy lookup.

In order to trade and redeem Digipounds, a browser extension to store private keys can be used to sign transactions, such as Metamask.

**1 : 1**

Always transparent. Digipounds are digital representations of GBP, currently trading at an equal valuation. Here's proof:



To validate the value of Digipound, the audit shows a list of all previous transactions written to the DigiPound smart contract, with a link to Etherscan to verify, and a list of all inbound and outbound payments, verified by Stripe.

# Bibliography

- [1] Cryptocurrency Market Capitalizations. <https://coinmarketcap.com/>. Accessed 15 Jan 2019. pages 4, 11, 12
- [2] Nakamoto, Satoshi. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Web. Accessed 20 Jan 2019. pages 6, 7
- [3] Fortney, Luke: Investopedia. *Bitcoin Mining, Explained*. Web. Accessed 20 Jan 2019. pages 4
- [4] Bitcoin Wiki. *Bitcoin History and Genesis Block*. Web. Accessed 5 Feb 2019. pages 5, 6
- [5] Ripple. <https://ripple.com/>. Accessed 19 Mar 2019. pages 5
- [6] Microsoft Support. *Add Money to your Microsoft account*. Web. Accessed 20 May 2019. pages 5
- [7] Hyperledger. <https://www.hyperledger.org/>. Accessed 20 May 2019. pages 7
- [8] Etherscan Block Explorer. <https://etherscan.io>. Accessed 10 Apr 2019. pages 9
- [9] Ropsten Block Explorer. <https://ropsten.etherscan.io>. Accessed 10 Apr 2019. pages 21, 33
- [10] The Ethereum. *ERC20 Token Standard*. [https://theethereum.wiki/w/index.php/ERC20\\_Token\\_Standard](https://theethereum.wiki/w/index.php/ERC20_Token_Standard). Accessed 11 Apr 2019. pages 9
- [11] Remix - Ethereum IDE. <https://remix.ethereum.org>. Accessed 10 Apr 2019. pages 10
- [12] Houben, Robby, et. al. *Cryptocurrencies and Blockchain* - Section 2.1.2. Accessed 19 May 2019. pages 10
- [13] Blockchain. *2018 State of Stablecoins*. Web. Accessed 1 Feb 2019. pages 10, 11

- [14] Iiunuma, Arthur. *Why Is the Cryptocurrency Market So Volatile: Expert Take.* Web. Accessed 1 Feb 2019. pages 6
- [15] Wikipedia. *2018 Cryptocurrency Crash.* Web. Accessed 21 Jan 2019. pages 5
- [16] BBC News. *IMF's Lagarde says central banks could issue digital money.* Web. Accessed 14 Nov 2018. pages 11
- [17] Gemini Dollar: U.S. Dollars on the Blockchain. <https://gemini.com/dollar/>. Examinations section. Accessed 28 Jan 2019. pages 11, 31
- [18] Dai Stablecoin. <https://makerdao.com/>. Examinations section. Accessed 28 Jan 2019. pages 12
- [19] Berreman, Allison: ETH News. *Circle's USD Coin Audit Reports Full Fiat Backing.* Web. Accessed 18 Jan 2019. pages 12
- [20] Khatri, Yogita: Coin Desk. *Tether Says Its USDT Stablecoin May Not Be Backed By Fiat Alone.* Web. Accessed 3 May 2019. pages 12
- [21] Kyber Network et. al. *Wrapped Tokens.* Web. Accessed 25 Jan 2019. pages 13, 14, 16
- [22] Chen, James: Investopedia. *ZWD (Zimbabwe Dollar).* Web. Accessed 25 Mar 2019. pages 14
- [23] Federal Reserve Bank of Dallas. *Hyperinflation in Zimbabwe.* Web. Accessed 30 Mar 2019. pages 14, 15
- [24] Odera, Lujan. *Why the release of Dogethereum bridge is a big deal for Dogecoin.* Web. Accessed 1 Feb 2019. pages 14
- [25] Neto, Moritz: The Tenzorum Blockchain Initiative. *How to issue your own token on Ethereum in less than 20 minutes.* Web. Accessed 5 Apr 2019. pages 20
- [26] web3.eth. <https://web3js.readthedocs.io/en/1.0/>. Accessed 19 Apr 2019. pages 21, 49
- [27] web3j. Application Binary Interface. <https://web3j.readthedocs.io/en/latest/abi.html>. Accessed 19 Apr 2019. pages 22
- [28] web3.eth. <https://stripe.com/docs/>. Accessed 1 Feb 2019. pages 26, 62, 66
- [29] Chatley, Robert. Imperial College London, Software Engineering: Design (CO 220). *Interactive Applications.* Accessed 20 Dec 2018. pages 35
- [30] Laravel. <https://laravel.com/docs/>. Accessed 21 Dec 2019. pages 37

- [31] EthHub. *Running an Ethereum Node*.  
<https://docs.ethhub.io/using-ethereum/running-an-ethereum-node/>. Accessed 19 Apr 2019. pages 45
- [32] Mislav, Javor. *An Introduction to Geth and Running Ethereum Nodes*. Web. Accessed 19 Apr 2019. pages 45
- [33] Atzei, Nicola, et. al. *A Survey of Attacks on Ethereum Smart Contracts*. 2018, pp. 10. pages 52
- [34] Frankenfield, Jake: Investopedia. *51% Attack*. Web. Accessed 8 Jun 2019. pages 47
- [35] Coinmonks. *How a 51% Attack Works*.  
<https://medium.com/coinmonks/what-is-a-51-attack-or-double-spend-attack>. Accessed 8 Jun 2019. pages 48
- [36] Buterin, Vitalik. *On Slow and Fast Block Times*. Web. Accessed 9 Jun 2019. pages 50
- [37] Crypto51. <https://www.crypto51.app/>. Accessed 7 Jun 2019. pages 50, 51
- [38] Luu, Loi, et. al. *Making Smart Contracts Smarter*. 2016, Chapter 6. pages 52
- [39] Tether Market Cap. <https://coinmarketcap.com/currencies/tether/>. Accessed 10 Jun 2019. pages 65
- [40] Ethereum Average Transaction Fee Chart.  
[https://bitinfocharts.com/comparison/ethereum-transactionfees.html/](https://bitinfocharts.com/comparison/ethereum-transactionfees.html). Accessed 12 Jun 2019. pages 65
- [41] WordPress. *Ether and ERC20 tokens WooCommerce Payment Gateway*.  
<https://wordpress.org/plugins/ether-and-erc20-tokens-woocommerce-payment-gateway/>. Accessed 7 Jun 2019. pages 75
- [42] Malavolta, Giulio, et. al. *Concurrency and Privacy with Payment-Channel Networks*. Accessed 9 Jun 2019. pages 76
- [43] Xue, Guoliang. Arizona State University. *Payment Channel Networks for Blockchain-based Cryptocurrencies*. Accessed 9 Jun 2019. pages 75