ASHESI UNIVERSITY COLLEGE

CRYPTOCURRENCY WALLET FOR VIRTUAL CURRENCIES

UNDERGRADUATE APPLIED PROJECT

B.Sc. Computer Science

RYAN YOOFI MOUJALED
2017

ASHESI UNIVERSITY COLLEGE

CRYPTOCURRENCY WALLET FOR VIRTUAL CURRENCIES

APPLIED PROJECT

Applied Project submitted to the Department of Computer Science, Ashesi University College in partial fulfilment of the requirements for the award of Bachelor of Science degree in Computer Science

Ryan Moujaled

DECLARATION

I hereby declare that this applied project is the result of my own original work and that no part of it has been presented for another degree in this university or elsewhere.

Candidate's Signature:

.....

Candidate's Name:

.....

Date:

.....

I hereby declare that preparation and presentation of this applied project were supervised in accordance with the guidelines on supervision of applied project laid down by Ashesi University College.

Supervisor's Signature:

.....

Supervisor's Name:

.....

Date:

.....

Acknowledgement

I am tremendously grateful to God, my supervisor, my family and friends for the strength, wisdom, advice and patience given me to compile, write and build this project. I would not have in any way been able to complete this project successfully without their selfless assistance. This challenging and new found understanding of digital finance has given me a better appreciation for everyday technology and the hard-working men and women who have made it possible. My profound gratitude goes out to them as well.

Abstract

This project seeks to demonstrate the use of virtual and cryptocurrencies. In recent years, they have taken on various forms; from mediums of exchange to stores of investment value. This unique function is made secure through the security and reliability of Blockchain technology. This project employs Stellar, a Blockchain implemented network to construct a payment and money exchange technology that has the capability of easing transactions and lowering the cost of money transaction across global distances. The project has demonstrated that virtual currencies held on digital ledger networks on the Blockchain can be practically implemented as a money wallet for every day use.

Table of Content

List of Figures

List of Tables

CHAPTER 1: INTRODUCTION

1.1 Background

In the centuries prior to the 21st Century, people historically exchanged goods and services through barter. One desirable good was exchanged for another in return. This mode of transaction ended, when civilization discovered the melting of precious metals. Consequently, they were used to form coins to enable transactions as a better representation of the value of a good or service. These precious metals represented stored values. As time progressed, these precious metals were replaced with paper version of money which represented these stored values (Hurlburt, & Bojanova, 2014). In more recent years, technology has promoted a near real-time exchange of money without the need for physical paper. Hard plastic cards, otherwise known as the credit and debit cards took the place of physical cash. Point of sale devices took hold all over the globe and it promoted a safer and more convenient mode of money exchange. Ubiquitous computing is a current rising trend that shows a lot of promise. This encouraging rise creates the avenue for a new phase of money transactions amongst devices. As such, virtual currencies have also started to gain traction – through Bitcoin, Litecoin, Ethereum etc. These virtual currencies have provided a brand-new mode of transaction that must be explored further to identify its possibility to be the next phase of money.

1.2 Problems and Opportunities

Money is an essential feature in everyday life. While money is also a means to commit crime, it is also a means to solve the world's most interesting problems. The creation of virtual and encrypted currencies – along with Blockchain technology provides a means to

cheaper, ready and more visible money. Contrary to public belief about cryptocurrencies, it could be harnessed to combat fraudulent behavior than support it (European Parliament, 2017). This is because of its traceable nature as opposed to physical money. Furthermore, crime and corruption is still a huge problem on the African continent, and virtual and encrypted currencies have the capabilities of curbing this conundrum.

Another pertinent issue on the African continent and the world at large with regard to money transfer into the continent is high remittance charges. Virtual and Encrypted currencies completely drop remittance charges to a fraction of what they are under companies like Western Union and MoneyGram. If a user wanted to send $1000 to family in Ghana through Western Union, the user would have to pay a fee of roughly $76. Under the current rate, as at this write-up, of 4.23:1(USDGHS) that would be a GHc 321.48 fee. That much money would greatly impact the life of any Ghanaian. Virtual and encrypted currencies could cut this down to as low as $0.01.

Much development has been done to create the framework for virtual currency exchange to be possible, but there are still no fully implemented applications that take advantage of these new currencies and their possibilities in Ghana. While new companies and developers are studying the trend, and attempting to build solutions, no concrete solution has been built for the Ghanaian market. Therefore, it is very vital to build a usable software for virtual currencies and cryptocurrencies in Ghana – more preferably one that takes advantage of the lower remittance overhead, and the traceability of money to support future banking and business solutions.

1.3 Related Work

This section gives a brief description and explanation of various technologies and backgrounds used within this documentation. The goal is to give a better understanding of the opportunities therein.

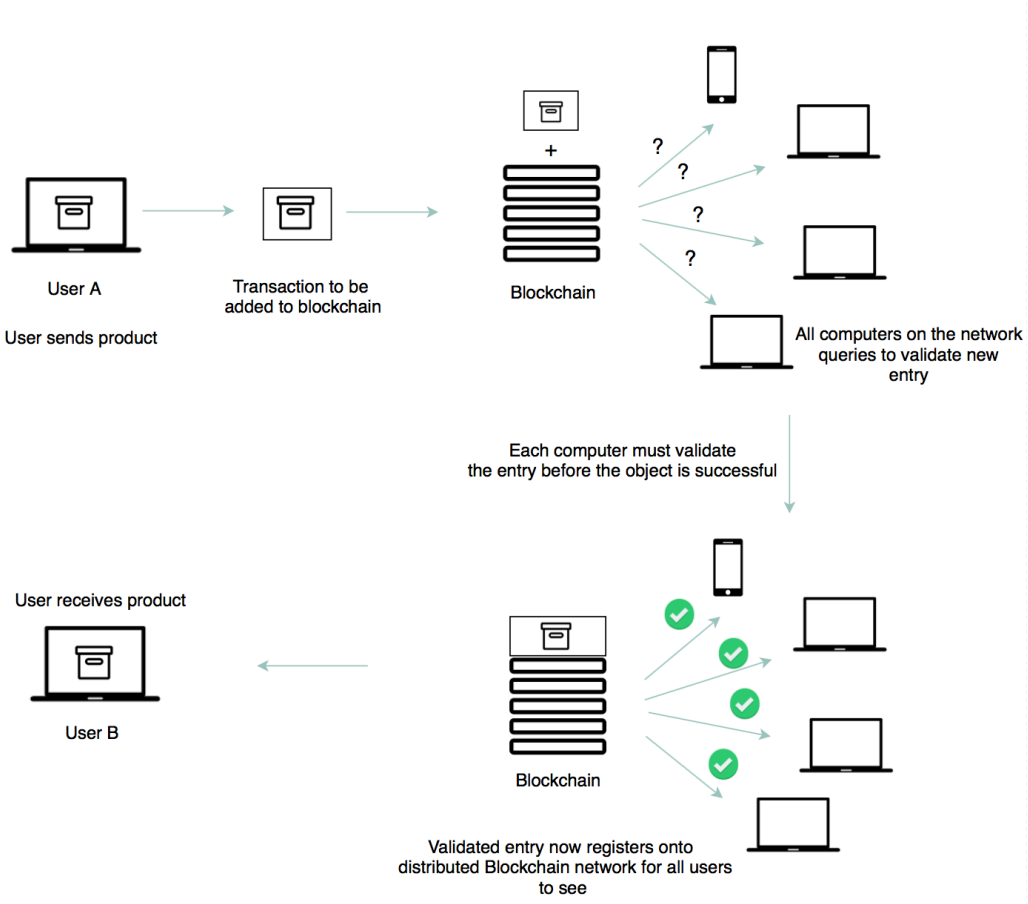

Figure 1.1: The nature of Blockchain technology

Block chain: This is a revolutionary new technology that is most popularly known for supporting virtual and encrypted currencies. It is sometimes referred to as a digital ledger (Nguyen, 2016). Figure 1.1 gives a descriptive view of how the technology works. The transaction object is first sent to a distributed ledger on a network. All computers connected on the network have the very same ledger – including the various details. This

may be likened to every student in the class writing words on each line on a piece of paper. Once a new transaction is added. All the computers verify that the incoming object does not violate any pre-defined rules and is from a trustworthy source (Underwood, 2016). In this analogy, the teacher dictates a new word to be written on a new line by the students. Since the teacher is a trustworthy source, all students write the word down. Once the object is registered on the public ledger, which all computers on the network can view, it is officially publicly recorded and cannot necessarily be changed. In the case of a situation where one computer on the network seems to try to alter this object, all other computers on the network would flag it as erroneous and reject this change into the public distributed ledger. This is one of the major ways the Blockchain secures itself. Notwithstanding, the Blockchain is not just for currencies but for any transaction of value to the user, including money, property, stocks, agreements. These could all be registered and monitored using Blockchain technology, which is why it is exciting technology for keeping fraud in check.

Cryptocurrencies: These are exchange mediums designed for digital information. This process, however, is enabled through cryptography. It promotes security of transactions and regulation or control of the creation of new currencies (Underwood, 2016) . Furthermore, it works using Blockchain technology which ensures even more security (Shehhi, Oudah, & Aung, 2014).

Virtual Currencies: This is a type of unregulated, digital money which is issued and controlled by developers, while accepted and used among members of a virtual community (European Central Bank, 2012).

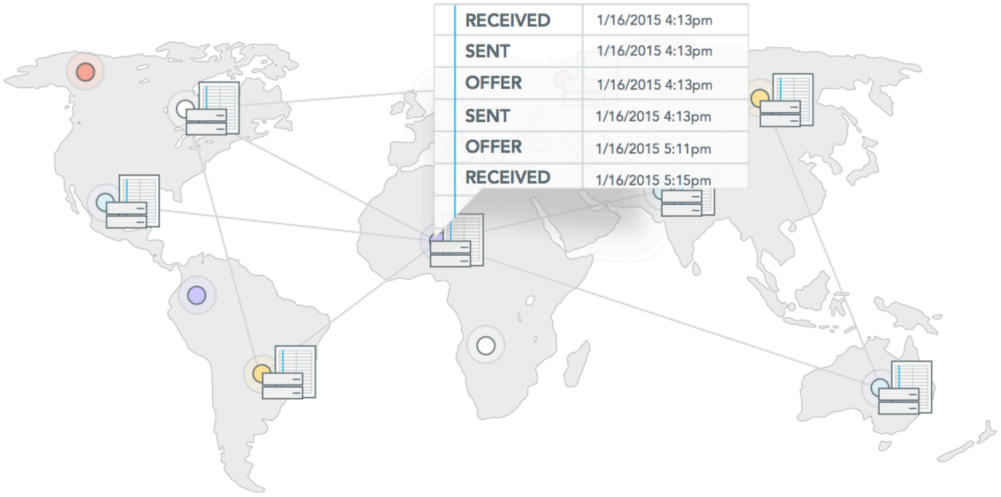

Figure 1.2: The Stellar ledger for recording transactions
Picture credit: stellar.org

Stellar Test-net: The Stellar network is a new technological implementation for processing financial transactions. The technology is open source, distributed, and community owned. It is a network of decentralized, distributed servers that, like the internet, are powered by a distributed ledger (block chain) as shown in Figure 1.2. These servers communicate with each other every 2-5 seconds to verify transactions in a mechanism known as 'consensus.' It should, however, be noted that Stellar is a way to handle virtual currencies. As such, in a real-world use case, it requires application developers to implement the actual wiring of cash to the end-users account, otherwise it would fail in real world cases when dealing with digitized fiat currencies. However, that

implementation is not required when using the Stellar Test-net (Stellar organization, 2017).

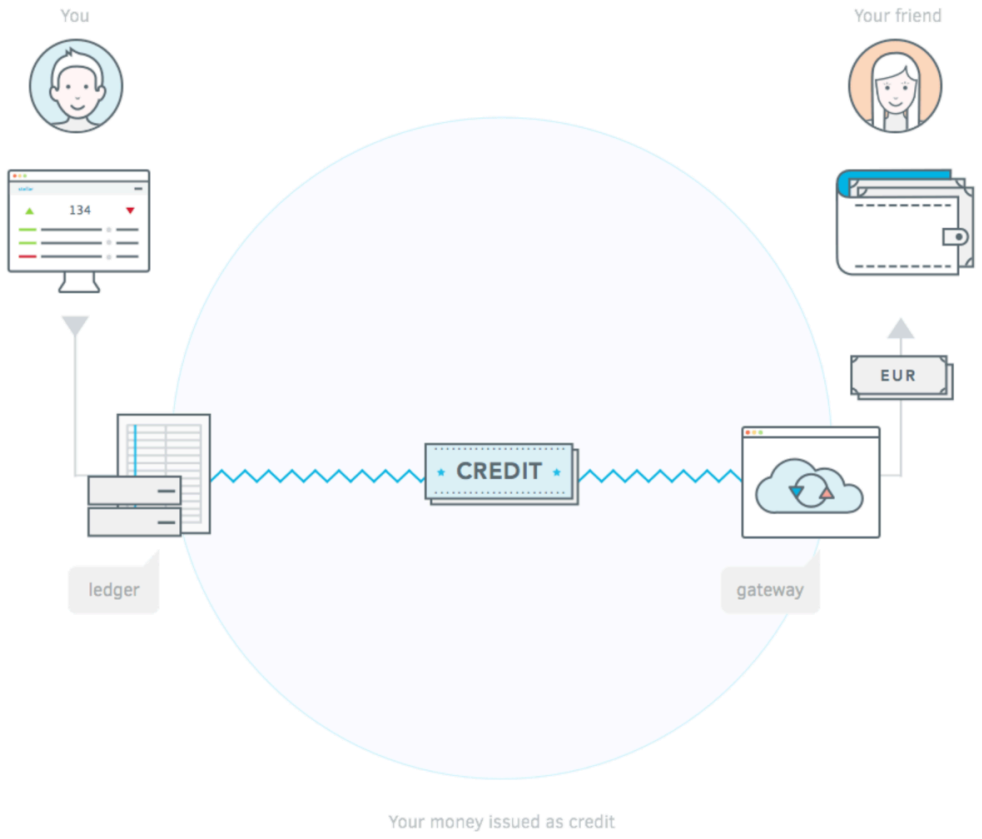

Figure 1.3: The Stellar ledger-gateway diagram for virtual currency transactions
Picture credit: stellar.org

Figure 1.3 demonstrates how the Stellar network executes a transaction from an implemented application utilizing its ledger to an end-user's wallet. The application must utilize public keys and secret seeds to determine the end-user's wallet to complete transactions. Once is it recorded on the ledger, the virtual credit is wired to the end-user.

This happens through a gateway that determines what currency the end-user should receive from implementation details.

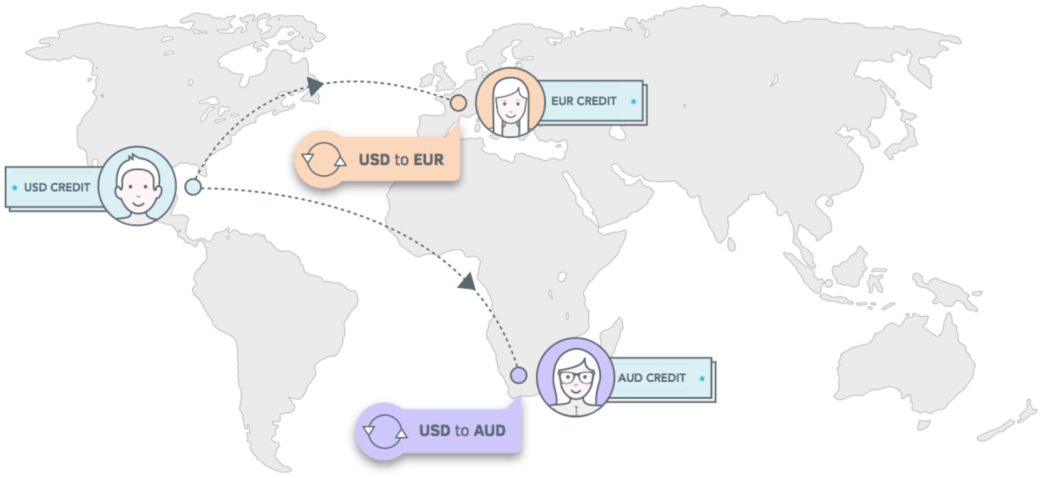

Figure 1.4: Stellar distributed exchange for currency issuing around the world
Picture credit: stellar.org

Lumens: Lumens is the native currency of the stellar network and is assists with the Stellar network's distributed exchange. They contribute to the ability to move money around the world and conduct transactions between different currencies quickly and securely. Figure 1.4 depicts the stellar distributed exchange behavior. Stellar supports credit currencies, which when sent from a USD creditor to a EUR creditor must be converted to lumens (Stellar Organization, 2017). This then goes through the gateway described in Figure 1.3 through the distributed exchange and is received on the other end as EUR. This scenario also requires implementation details and collaboration between developers of both accounts that should exist on the stellar network. i.e. Developer A creates TablePay Company in the United States and Developer B creates ChairPay

Company in the United Kingdom. Developers A and B, should both build their application on the stellar network to enable the Stellar distributed exchange to work.

1.4 Scope

This project would take into considerations the opportunity that is available to create a foundation for the implementation of other wallets that can use the Stellar Network.

1.4.1 Existing Solutions

Further research shows that not many companies in Ghana utilize the stellar network. However numerous companies do accept bitcoin both in Ghana and outside of Ghana. Bitcoin is also a decentralized virtual currency that is currently leading the digital currency market. In April 2013, one bitcoin was worth $150. Currently on April 14, 2017, one bitcoin is worth $1180.99. This shows the increasing attraction for digital currencies. Companies like Coinbase in the US, BitPesa in Kenya and ANX is China amongst others are huge names in the bitcoin industry. However, none exists for the Ghanaian market. The only source of bitcoin exchange is via localbitcoins.com as shown in Figure 1.5.

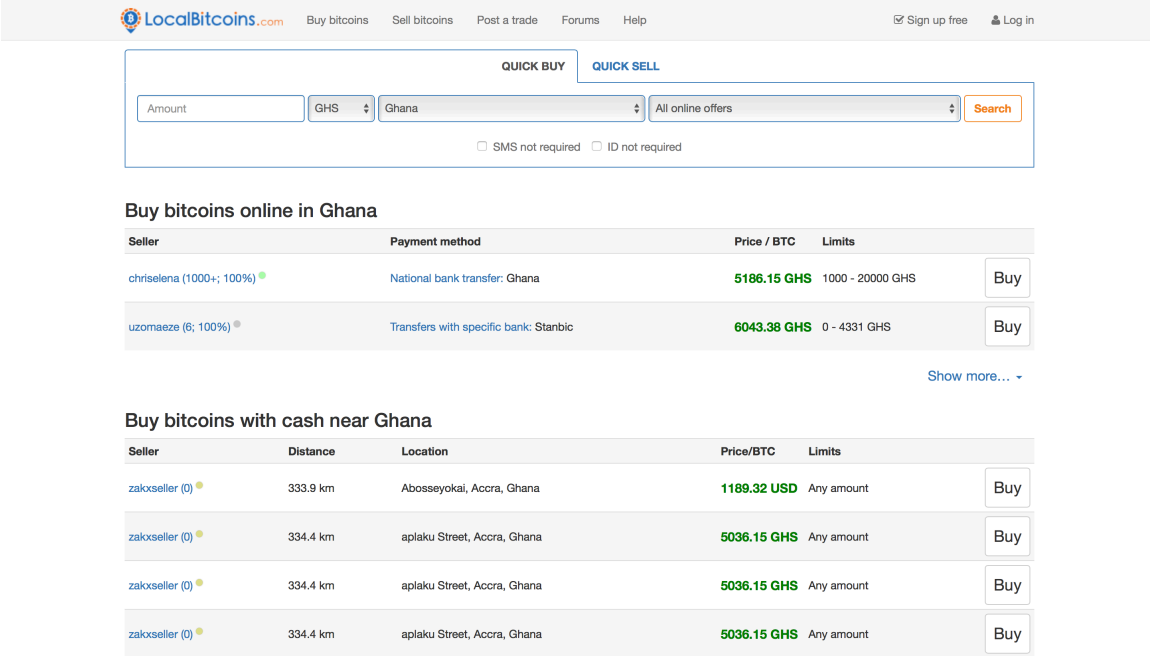

Figure 1.5: the localbitcoins.com website

However, this existing way of acquiring and utilize Bitcoin does not help the average Ghanaian very much. This is because, it still cannot be easily transacted between users, and people buy and sell based on an auction model rather than a currency exchange model. This makes things very unclear and haphazard.

However, outside of Ghana, Deloitte and Tempo Money Transfer amongst others have successfully implemented prototypes using the Stellar network. Deloitte reported to have reduced their transaction costs for banking solutions outside North America by up to 40% (Stellar Organization, 2017). The truly admirable thing about the Stellar network is that it allows for developers to make accounts that hold any kind of digital asset, including Bitcoin credits. This makes users capable of trading in Bitcoin. This is also dependent on implementation details.

1.4.2 Intended Solution

This project intends to provide a solution for acquiring digital currencies and spending small amounts between each user in the form of wallets. This project would focus on Ghana Cedis, because it would make functionality more understandable. Although we could have used lumens, bitcoin or some other digital currency. In Figure 1.6 below, we see that each user through a device can access his or her wallet, register transactions on the ledger and exchange digital currency between other users through the wallet.

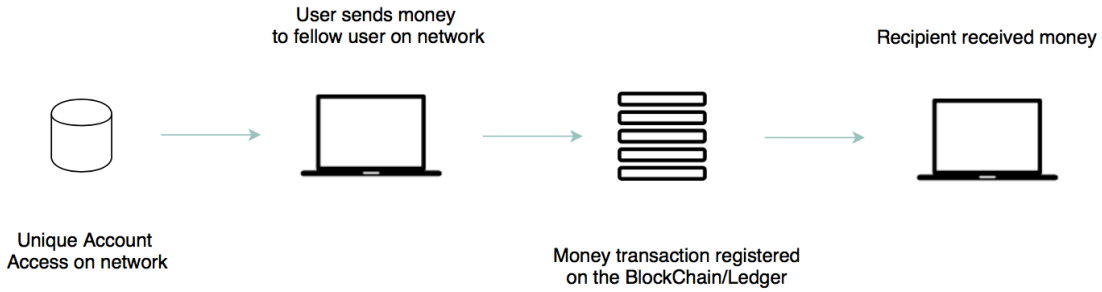

Figure 1.6: Simple system relationship for virtual currency transactions

The focus of the framework is to firstly consider security. It will consist of strong security protocols. The first phase of security would be to utilize password hashing. The next phase of authentication is through the stellar network. Every stellar-created account has a public key and secret seed. Stellar utilizes this public key cryptography to validate that all transactions are secure. The public key, unlike the secret seed can be shared during implementation with other wallets. This would require it to identify accounts and verify that a user has authorized a transaction. The secret seed on the other hand is private to a user and should not be shared. The secret seed is used to encrypt and decrypt data.

The project involves the ability to request and make payments directly into a wallet. The wallet would enable two distinct actions; to request for money and to send/pay money. Firstly, the stellar network requires that accounts exist before a pay or request can be made to that account through its public key. All key activities that would be addressed are shown in Figure 1.7.

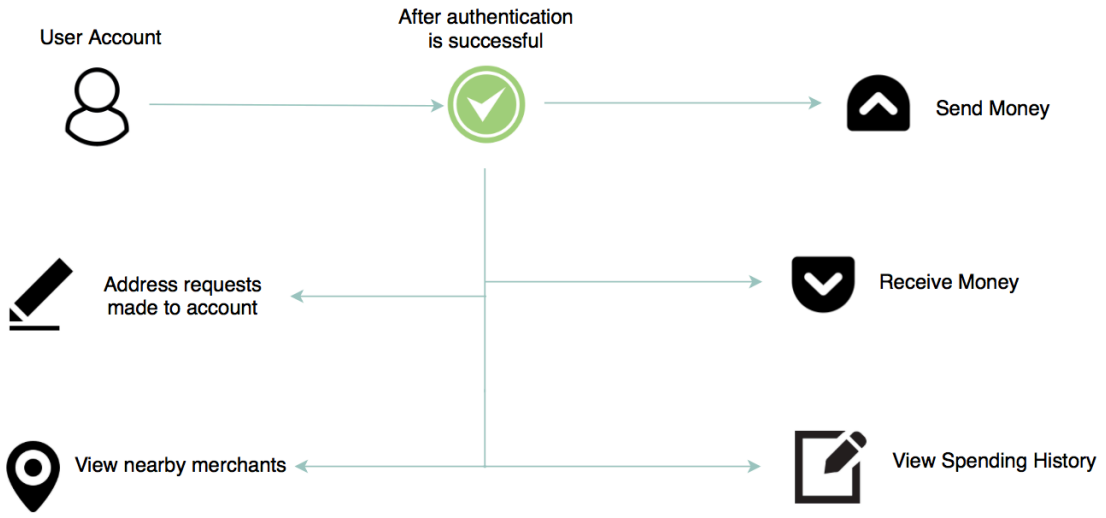

Figure 1.7: Overview of project

Actions executed on the stellar network require an operation element and a transactional element. Operation elements are essentially manipulative - such as, making payments, altering account details and trading various currencies on the stellar network. Transactional elements on the other hand is 'a group of operations with extra information.' If a user had a wallet balance of GHc50, but tried to send GHc50 to two accounts within one transaction but 2 operations, it would fail. Consequently, the request and pay implementation details utilize operations and transactional models of stellar to

create a pay and request module. Furthermore, the use of the request and pay module should be made through a phone number. This assists with keeping the person's wallet mobile. The account is tied to a particular mobile number - this is not a component of stellar, but a component of this projects' implementation detail as shown in Figure 1.8.

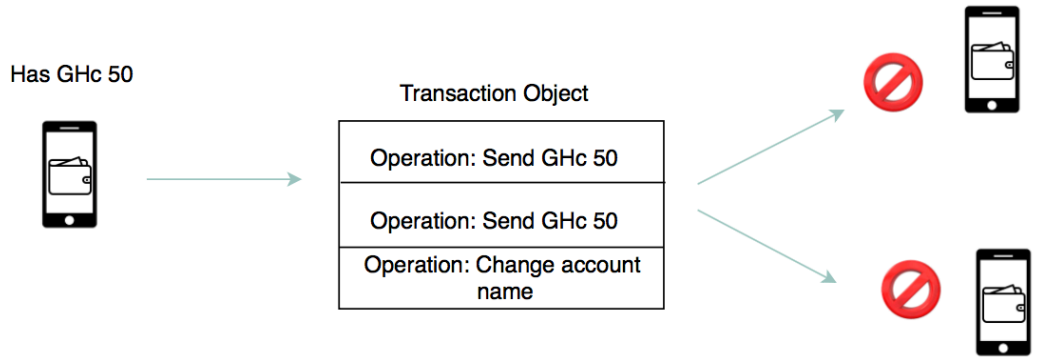

Figure 1.8: Implementation detail should prevent transaction object from being tricked.

This project also takes into consideration the need for logging user's money. This is done by maintaining a steady log or memo of particular payments. Users can, therefore, identify who they have made particular payments to and from, along with the date and attached message by the requestor or sender of the money.

The project also considers logging transactional history, approving and declining of requested money. The project furthermore addresses merchants and the communication between a user and a merchant. In a more realistic implementation, the users would have access to phone numbers through a contact list that is integrated with the phone. A wallet account cannot send money or request money towards uncreated accounts. This attempt would be met with an error. The issue here is that a transactional element cannot be

created for that operation element. Lastly, this project is limited to a sandbox implementation of stellar test-net. Therefore, the exchange of currencies and use of real word currencies was not added to the application. The goal is to create a framework upon which a government backed currency and other cryptocurrencies can be built on top of.

1.5 Benefit and Objective

A prevalent problem within the Ghanaian society is the existence of point of sale devices for accepting Visa and MasterCard credit and debit cards. Frequently, these machines are either broken down or out of battery at various restaurants and result in the reduction of customer satisfaction. One benefit of a wallet, with both merchant and client side is to solve this problem and eliminate the middleman (Robert, Kubler, & Traon, 2016).

Furthermore, digital money means digital solutions and services that require particular currencies are readily accessible. Modern software as a service companies are increasingly accepting digital currencies for their services – such as bitcoin and litecoin. A wallet framework tailored to reaching out to these software frameworks serves to bring these services closer to a consumer.

The outstanding benefit is geared toward solving the problems mentioned in section 1.2, where money becomes more traceable and remittances are drastically reduced for users around the world (Maxwell, Speed, & Campbell, 2015).

The objective of this project is to set up a base wallet framework for future work on virtual currency wallet implementation for real world use. This base implementation was

highlighted in the scope section, and leaves room for addition of API's. A prime objective in a real-world scenario would be to encourage small-scale transactions between users.

1.6 Chapters Overview

The next phase of this documentation is to outline the requirements of the software as well as design considerations and specifications. Subsequently in this documentation, an outline of implementation details is outlined. The next chapter after implementation details would outline various tests that were done on the software as well as the accompanying results. The documentation concludes with a specification of limitations and relevant additions and future work that would serve to improve the virtual currency wallet.

CHAPTER 2: REQUIREMENTS

2.1 Requirements Overview

This section discusses factors and decisions that go into requirement specifications.

2.2 Product Perspective

The product will exist on distinct platforms that would work together in a single software container. However, the transaction and operational elements would be accomplished using the stellar network and requires its existence.

Essentially the software, will consist of:

a. **The Web Application**

b. **Heroku: Cloud Platform as A Service (PaaS)**

c. **Stellar Network's Test-Net**

2.3 Product Functions

The software functions that run the core of the application are as follows:

a. **Send or request and receive digital currency:** This function of the software enables users to send and receive virtual currencies through their virtual currency wallets held on the stellar network. Additionally, the function enables a user to make a request to a different user to be approve or decline the request for money.

b. **Track and monitor transactional and request history:** Here, the software functions to monitor history of each wallet. When a user executes an operation element to send money, and the transaction element completes, it is logged

c. **Fetch and transact with merchants:** The software has the capability of fetching

registered merchants. This is done through geolocation and communicated to the application accordingly.

2.4 User Classes and Characteristics

The intended users of the product varies widely. The system should be usable by everyone. The wallet account holder and merchant and the two important entities of the software.

- **Wallet Account Holder:** The wallet account holder would require an individual with literacy and competency with technological devices. The software contains many cues and directions for software use, as well as many descriptive and colour coded elements to give users ease of use. Other characteristics, such as educational level and experience would play no decisive role in the use of the software.

- **Merchant:** The merchant would require little to no expertise as money would be wired straight into the merchants account. Practically, if linked with a bank account – the merchant would be able to tap a button to withdraw his real cash to third party application like mobile money. Characteristics, such as educational level and experience would play no decisive role in the use of the software.

2.5 Operating Environment

All parts of the software will execute exclusively on a browser. This includes all browsers on any operating system.

2.6 Design and Implementation Constraints

- Particular conditions that provide limitation on implementation options include regulation in implementing a wallet that involve testing currency trading. (i.e trading GHc for dollars. Exchanging of currency requires an agreement between the two developers of the respective wallets.

- Regulations for real world testing also include having a registered bank account from which transaction would go through as well as legal documentation, which are out of the scope of this project.

2.7 Assumptions and Dependencies

The following factors influence and affect the performance of the software, and are required or not required for its functioning smoothly:

- All devices sending, requesting and accessing the wallet must be connected to the internet.
- Sending and requesting money must be made to and via an existing wallet hosted on Stellar network's test-net.
- Merchants exclusively show when the user is in the region of the merchant using a triangulation calculation.
- Merchants must be exclusively linked to a phone number and name for successful payments (transactional elements) to be completed and for effective transactional history logging.

2.8. Specific Requirements

2.8.1.1 Description and Priority

This feature provides the user with capabilities to execute transfer of money. This feature is a high priority one and is the essential to the effective use of the application. The feature will ensure correct exchange of money between wallets on the stellar test-net.

2.8.1.2 Use Case: Send, Request and Receive Money

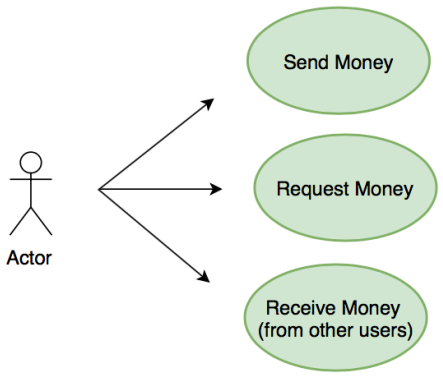

Figure 2.1: Use case diagram for sending, requesting and receiving money

2.8.1.3 Response Sequence: Send, Request and Receive Money

Mr. Gary would like to pay Mr. Ash for paying for lunch yesterday at a diner that does not accept payment through this projects implementation. Mr. Ash left his physical wallet in a hurry because they had to grab a late lunch. Mr. Gary foots the bill and makes a request for Mr. Ash to pay him back using the request module. Mr. Gary would then have to approve the request and Mr. Gary would receive his money back.

2.8.1.4 Functional Requirement

REQ-SRR-1: A user should be able to send money to another users' wallet

Scenario:	A wallet account holder would want to send GHS to another person account to increase the beneficiary's account balance.
Input:	Input the beneficiary's phone number, value to pay and message attached to the pay module.
Output:	The wallet account holder will receive a feedback on the transaction being completed.
Action:	The account holders' wallet would have to decipher the beneficiary's wallet for delivery on the stellar test-net.
Pre-condition:	The wallet account holder needs to be logged in.
Post-condition:	N/A
Side-effects:	Transactional element will not be completed if the specified beneficiary's number does not exist.

Table 2.1: Table of requirements for REQ-SRR-1

REQ-SRR-2: A user should be able to request money from another user

Scenario:	A wallet account holder would want to request money from another

	wallet account holder to buy groceries the next day.
Input:	Input the 'requestee's' phone number, value to request and message attached to the request module.
Output:	The wallet account holder will receive a feedback on the request being completed.
Action:	The wallet account of the requestee's public key would have to discover the beneficiary's wallet public key for delivery using the stellar test-net upon approval
Pre-condition:	The wallet account of the 'requestee' needs to be logged in to approve the request.
Post-condition:	The requestor needs to log in to view the changes made to their account.
Side-effects:	Transactions would not complete if the requestor specifies a wrong requestee phone number.

Table 2.2: Table of requirements for REQ-SRR-2

REQ-SRR-3: A user should have his total balance affected upon money sent or requested

Scenario:	A wallet account holder would like to send the user some amount of money and ensure the money has been sent.
Input:	Input the beneficiary's phone number, value to pay and message attached to the pay module
Output:	The wallet account holder will receive a feedback on the transaction being completed for the payment.
Action:	A transactional element is sent to the end-users account wallet. The users balance update when the wallet balance is loads.
Pre-condition:	N/A
Post-condition:	User account wallets is online
Side-effects:	Stagnated pages may not have updated until some manner of refreshing is done.

Table 2.3: Table of requirements for REQ-SRR-3

2.7.2 Track and Monitor Transactional History

This section outlines the tracking and monitoring of transactional and request history. It gives a brief description, followed by use cases of the requirement. Subsequently, this is followed by a response sequence and an extensively discussed functional requirement.

2.8.2.1 Description and Priority

This feature provides the account holder the ability to view changes to his wallet. Additionally, the user has the capability to see the memo attached to the transaction as a way of keeping tabs on previous dealings. This feature is of medium priority.

2.8.2.2 Use Case: Track and Monitor Transactional History

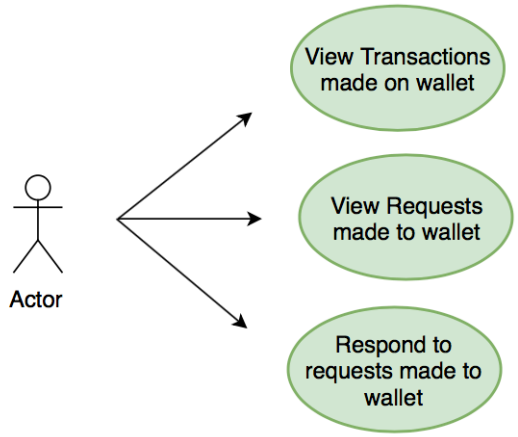

Figure 2.2: Use case diagram from viewing and responding to transactions and requests

2.8.2.3 Requirement Scenario: Track and Monitor Transactional History

Mr. Oak has made a lot of transactions in the past few months but did not note them down in his budgeting application. As such, he would need a reference to recent transaction that he has made to keep abreast with his current expenditure.

2.8.2.4 Functional Requirement

REQ-TTR-1: A user should be able to view transactional history

Scenario:	A wallet account holder would want to view his transactional history.
Input:	The user would have to navigate to the transactional history tab.
Output:	Transactional history of the wallet name, date, amount, type of payment and reason for the transaction are displayed.
Action:	All the wallet account holder' transactional elements based on the logged in ID are fetched for the user to view.
Pre-condition:	User logged in
Post-condition:	User logged in
Side-effects:	Transactional elements cannot be fetched without the user being logged in.

Table 2.4: Table of requirements for REQ-TTR-1

2.8.3 Fetch and Transact with Nearby Merchants

This section outlines the tracking and monitoring of transactional and request history. It gives a brief description, followed by use cases of the requirement. Subsequently, this is followed by a response sequence and an extensively discussed functional requirement.

2.8.3.1 Description and Priority

This feature enables the user's software to retrieve nearby merchants as well as execute specified payments to the merchants. Consequently, these eliminates the need for the user to physically ask, or search for a sign to decide whether they accept the currency or not. This is also of medium priority since it is an added benefit of the software and promotes a special case use of the software.

2.8.3.2 Use Case: Fetch and Transact with Nearby Merchants

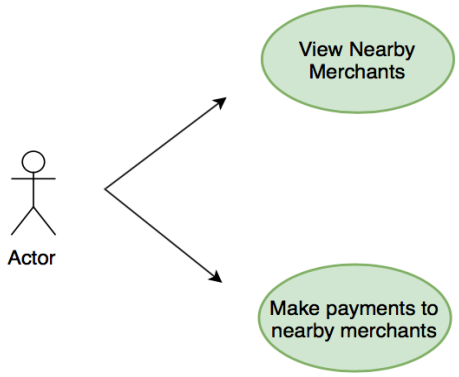

Figure 2.3 Use case diagram for fetching and transacting with nearby merchants

2.8.3.3 Requirement Scenario: Fetch and Transact with Nearby Merchants

Mr. Birch sits in his car and realizes he forgot to buy his kids a new game the asked of him in the morning. However, he has spent all his physical cash and only has money left on his wallet. Nonetheless, Mr. Birch can fire up the software application and discover nearby merchants that would potentially be selling the game his kids desperately want.

2.8.3.4 Functional Requirement

REQ-FTM-1: A user should be able to view all merchants around a location

Scenario:	A wallet account holder would want to view nearby merchants while in his car.
Input:	User selects merchant tab with mouse click.
Output:	A list of nearby merchants are displayed on the users dashboard.
Action:	User pans across map to view all merchants with mouse click.
Pre-condition:	User logged in
Post-condition:	User logged in
Side-effects:	No merchants may be discovered if there are not registered merchants in the area or no internet connection.

Table 2.5: Table of requirements for REQ-FTM-1

REQ-FTM-2: A user should be able to make payments to a nearby merchant after rendered service.

Scenario:	A wallet account holder would want to pay a merchant after a service rendered.

Input:	The user would have to select pay and specify the requisite amount to be transferred.
Output:	Verification of the payment once complete is delivered to the sender to confirm that the transaction element to the merchant was complete.
Action:	Transactional element from the users public key to the merchants public key are successfully routed and GHS delivered and registered.
Pre-condition:	User logged in
Post-condition:	User logged in
Side-effects:	User cannot request from merchants.

Table 2.6: Table of requirements for REQ-FTM-2

REQ-FTM-3: A user should be able to view merchant payments in the transactional history.

Scenario:	A wallet account holder would want to view his transactional history.
Input:	The user would have to navigate to the transactional history tab.
Output:	Transactional history of the wallet name, date, amount, type of payment and reason for the transaction are displayed – Merchant

	name would be shown instead of the wallet name.
Action:	All the wallet account holder' transactional elements based on the logged in ID are fetched for the user to view.
Pre-condition:	User logged in
Post-condition:	User logged in
Side-effects:	Transactional elements cannot be fetched without the user being logged in.

Table 2.7: Table of requirements for REQ-FTM-3

2.9 External Interface Requirements

External interface requirements refer to the structural approach regarding interfacing of the application software. It consists of the user interface, hardware interface, software interface and communication interface.

2.9.1 User Interfaces

The interface should first provide information that allows the user to know the state of his wallet. A web interface would be required to enable it to run on multiple platforms. This includes desktop, mobile and other devices with browser capabilities.

Web Application: Once the user opens the application, the first point of contact should be authentication. Once authentication is completed, the user should gain access to the wallet. The user then has options in tabs available for further viewing and action options.

2.9.2 Hardware Interfaces

User accounts require phone numbers to access them. This access is powered my sms verification codes sent to a mobile number. This must work in the following manner: user should enter mobile details. Subsequently, the user should receive a code via sms for verified access. Therefore, a handheld mobile device that can receive sms is required to retrieve this verification code to confirm the users' identity.

2.9.3 Software Interfaces

The application software would integrate with a plethora of API and service solutions such as the Stellar network, Postgres database, Heroku's cloud platform as a service, Node.JS, Google API's and Auth0.

Stellar Core: Using this backbone of the stellar network and their API Documentation's the software should implement the requisite parts to enable the requirements to be met. These comprise of building the client endpoint, fetch info callback, client end point for making payments and Sanctions, ask user, fetch info and receive callbacks for receiving payments.

Database: All requisite date should be held in a database for storage and retrieval accordingly using the corresponding database API.

Cloud Service PaaS: A cloud service hosts will be required to host the requisite post, get and put API's for manipulating data within the user interface.

Server-side: Server-side language would be required is used in creating the backend with the stellar core. It involves a list of libraries. This comprises of express, bodyParser, moment, Chance and sequelize, which would be discussed later in this documentation.

Google API: Google maps API for geolocation is utilized for merchant triangulation with Wi-Fi and is required for displaying the merchants.

2.9.4 Communications Interfaces

The system requires effective communication to carry out its tasks. Although it is not communication heavy, every step requires a degree of communication requirements. For most activities within the software, the stellar core, auth0 or Google's API would need to be called. Furthermore, they need to be delivered within an acceptable amount of time that makes it a practical and viable tool to use. The communication interfaces used within the application are:

- **Asynchronous JavaScript and XML:** This is required to ensure that the application operates quicker and mitigates the effects of a slower connections
- **Hypertext Transfer Protocol:** This is to enable communication between media on the application.

2.10 Nonfunctional Requirements

Performance Requirements: The system must be quick and convenient for the user. The user needs to find it fast so it would not be a challenge to send money out through the virtual currency wallet.

Standard Compliance: Given the application utilizes virtual currency, it must conform to the laws of the country according to the Payment System and Service Bill Act of Ghana. It is currently, still within parliament and pending approval. However, this application would conform the details outlined in the drafted bill accordingly – such as the Anti-Money Laundering paragraphs and the requisite information that must be gathered from a user.

Availability: The user should have the capability to make payments once they have an internet connection. In the case where the user is not, the user should have the availability to still view wallet details.

Security: The system would require authentication, authorization and accounting (AAA) user access before making any payments. This would prevent users from having access to particular wallet accounts without confirmation of the user's credibility to prevent thievery and payments inflations.

Usability: The user should be constantly communicated with, throughout the user's engagement with the software, to ensure the user is aware of communicating parts of the software. This includes transactions and message passing confirmation and refusal amongst others.

Scalability: The user should not experience lag or experience the software slowing down when the user has numerous past transactions.

CHAPTER 3: ARCHITECTURAL DESIGN

3.1 Architectural Overview

This section discusses the architectural decisions that influence the design of the system implemented.

3.2 Design Reflection

The continuing sections below reflect and discuss the architectural models and design for the system implementation.

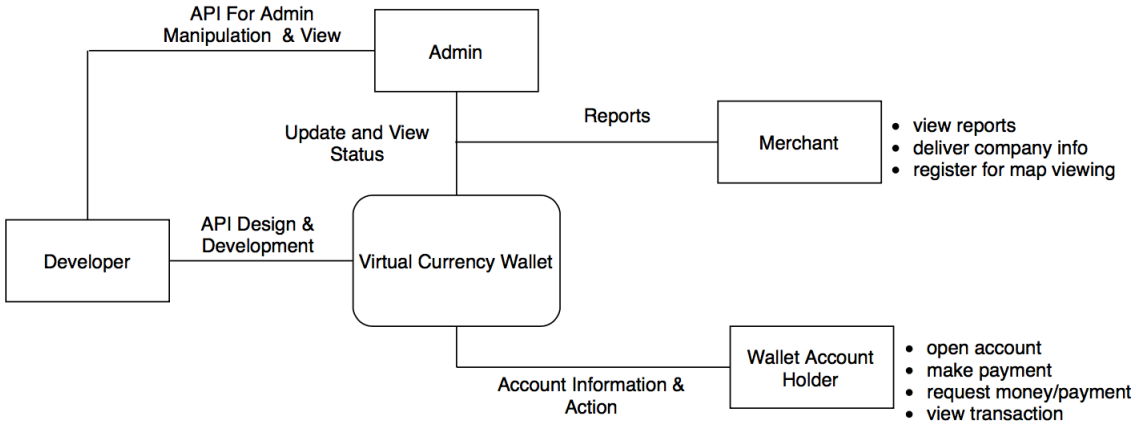

Figure 3.1: Context diagram showing system interrelations

Figure 3.1 describes the context of the system. In this design, the developer is responsible for designing and implementing the application programming interface for the required modules of the wallet. The three other users; the administrator, merchant and wallet account holders all have their activity between each other through the wallet and do not have direct interaction between each other. It should be noted that the internet is a central part of the architecture and relies on it for stability.

3.2.1 System Architecture

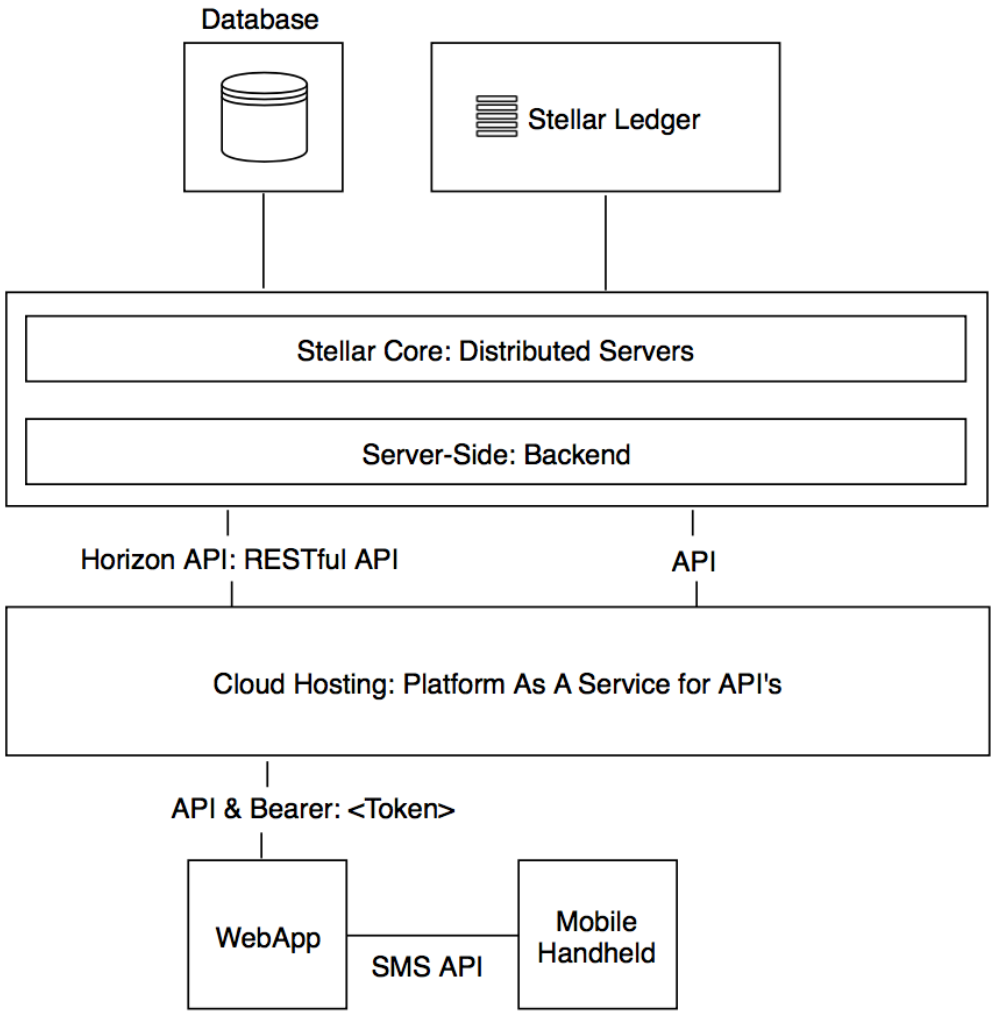

Figure 3.2: System Architecture Block Diagram Showing Application Systems

Figure 3.2 is a diagram that displays the architecture of the entire system as well as how the various blocks communicate. The stellar core network facilitates the functions of the database server and the web application. Additionally, the Heroku Cloud Service hosts the API for use on the web application. Furthermore, the mobile handheld receives communication using an SMS API for authentication. The interfacing for the various

blocks include API connections as well as token maintenance and transfer to facilitate communication between models. The token needs to be constantly maintained and in memory cache to ensure continued access of the application and the stellar core.

3.3 Design Specifications

This section discusses the steps that constitute the design considerations of the software. Possible steps that the user can take are contained within this section and serves as a building block for how the system would be implemented. Figure 3.3 below shows the activity diagram from actions the user can take once the application is booted.

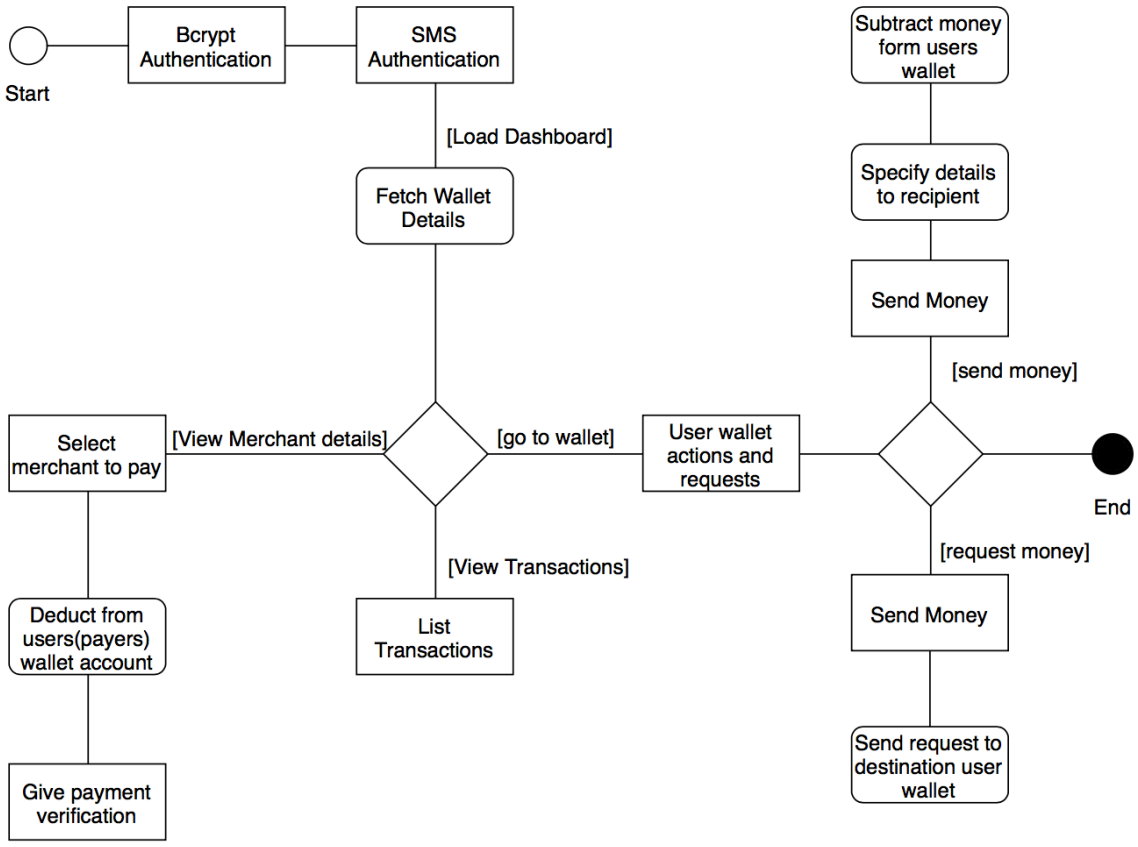

Figure 3.3 Activity Diagram of Software Architecture

3.4 Logical view: Sequence diagrams

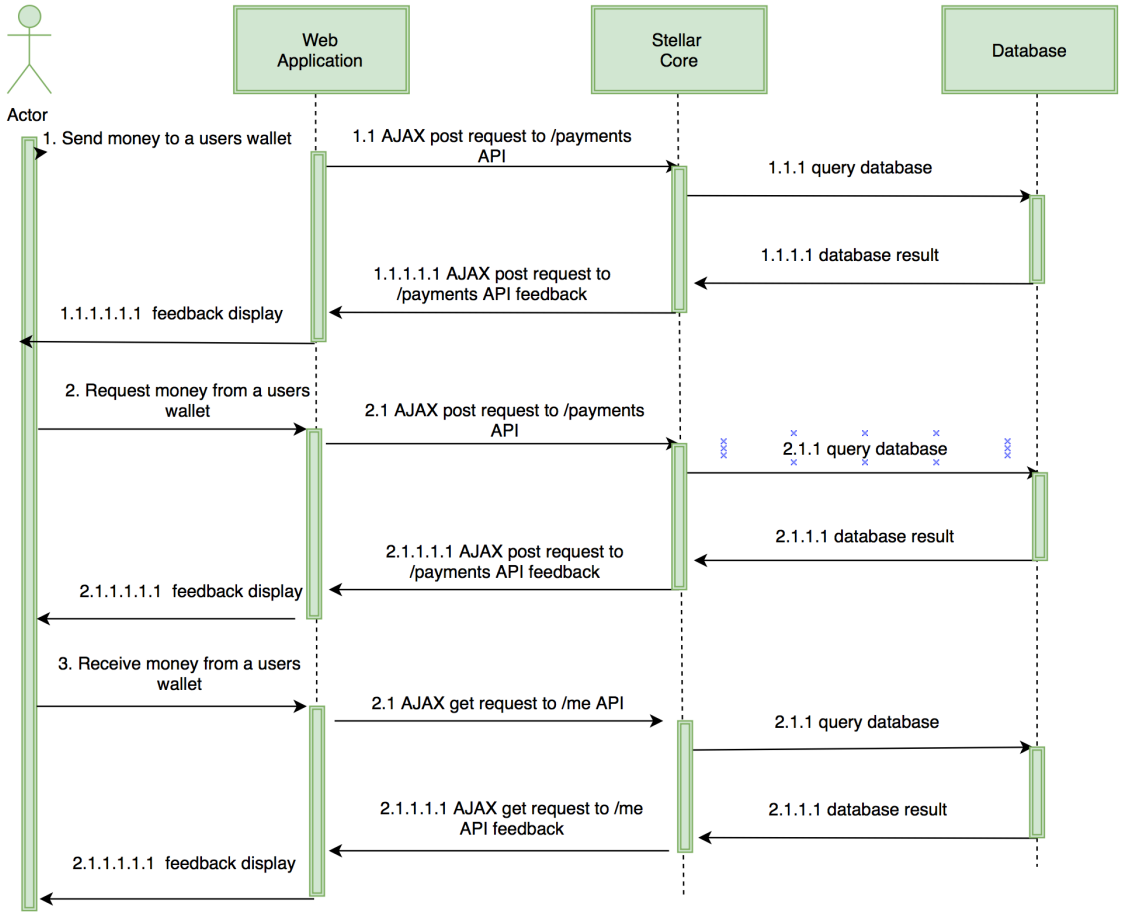

Figure 3.4: Sequence Diagram for Sending, Requesting and Receiving Payments

Figure 3.4 gives a display of the sequence diagram that delineates the communication and interrelation between a user and the application software. It shows the sequences of communication between the user, the web application, the stellar core hosted on a cloud service through the backend, as well as the database system. It shows the process of sending, requesting and receiving money.

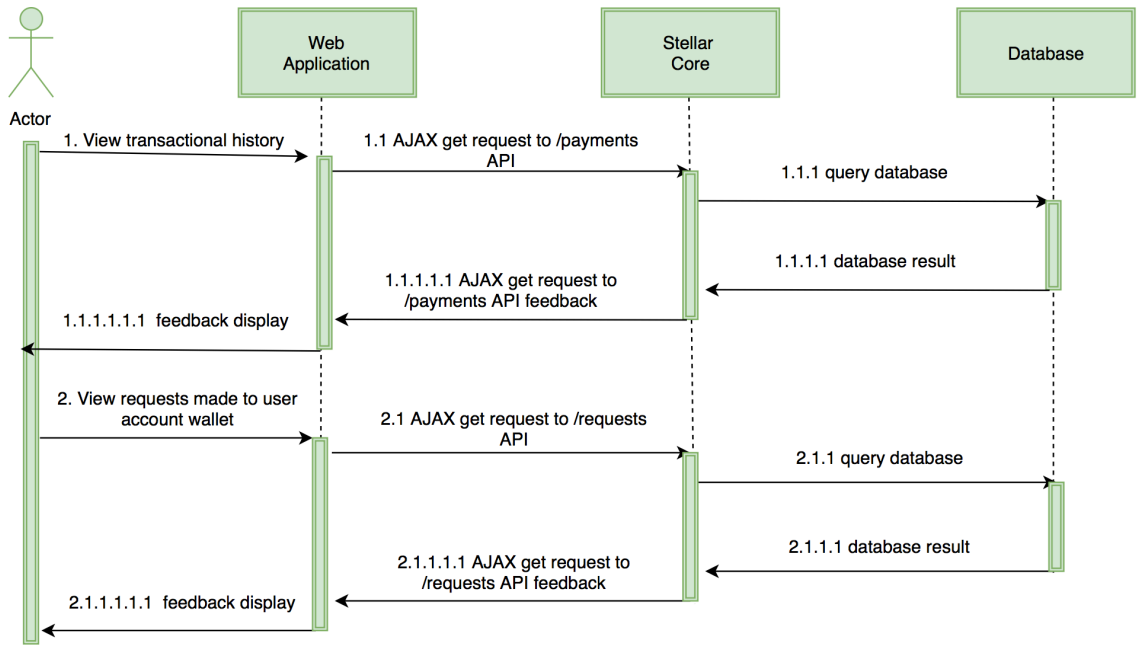

Figure 3.5: Sequence Diagram for Transactional and Request History

Figure 3.5 above demonstrates the sequence of actions the system would progress through. It shows the sequences of communication of the system –which would be hosted on a cloud service, as well as the database system. It shows the process of viewing transactional history as well has viewing requests made to the users account.

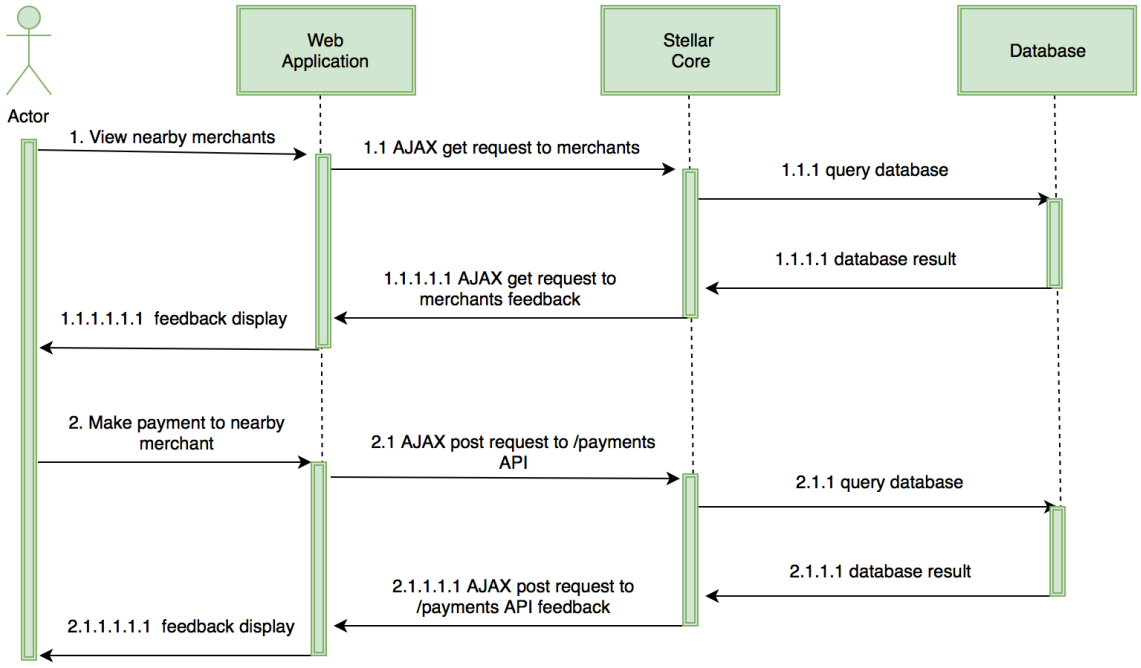

Figure 3.6: Sequence Diagram for Fetching Merchants and Transacting with Merchants

Figure 3.6 above also shows another sequence diagram for fetching and transacting with merchants. The database would always return a result. Error handling should also be handled from the backend depending on the database result.

3.5 Logic Layers

The architecture of the software is additionally segregated into three layers to highlight and expose core functionalities. This particular layered architecture enables the software application to be separated to allow for modular programming (Three-Layered Services Application, 2017). This way the impact of bugs and changes are limited. The layers are shown below:

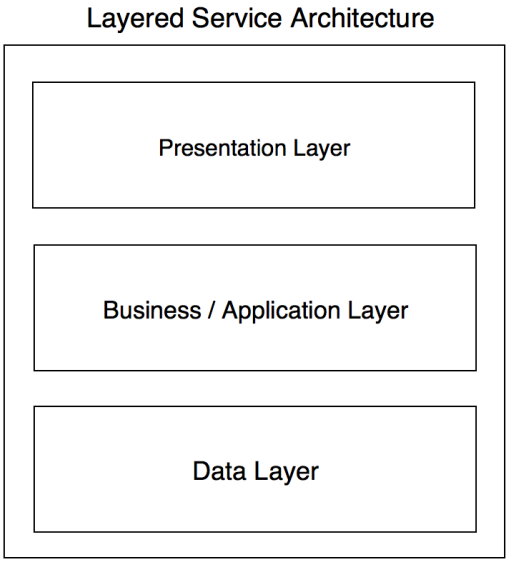

Figure 3.7: Three-Layer software application architecture

Presentation Layer: The presentation layer would be separated constructed using web frameworks and scripting languages. This would specifically allow all factors that rely on display and presentation of information to be delivered specifically by this layer. The backend makes no input in this regard, and strictly presents the frontend with the information for it to display accordingly.

Business / Application Layer: This layer deals with the required functions of the software application. This include the backend implementation to make and request payments, view transactional history and wallet information. This layer deals with the churning of data into a software that can serve use to the end-user.

Data Layer: This layer concerns itself with the retrieving and manipulation of data, as well as the impact of change in the data to the effectiveness of the software. This section

is also implemented in the backend architecture to streamline the software building process.

CHAPTER 4: IMPLEMENTATION

4.1 Overview

The application works using various facets of software implementation. To recap the requirements; send, request and receive money requirements, transactional history and requested money and merchant presentation and payment. This chapter discusses implementation details of the software – including the stellar core, horizon API, the backend, database and web application.

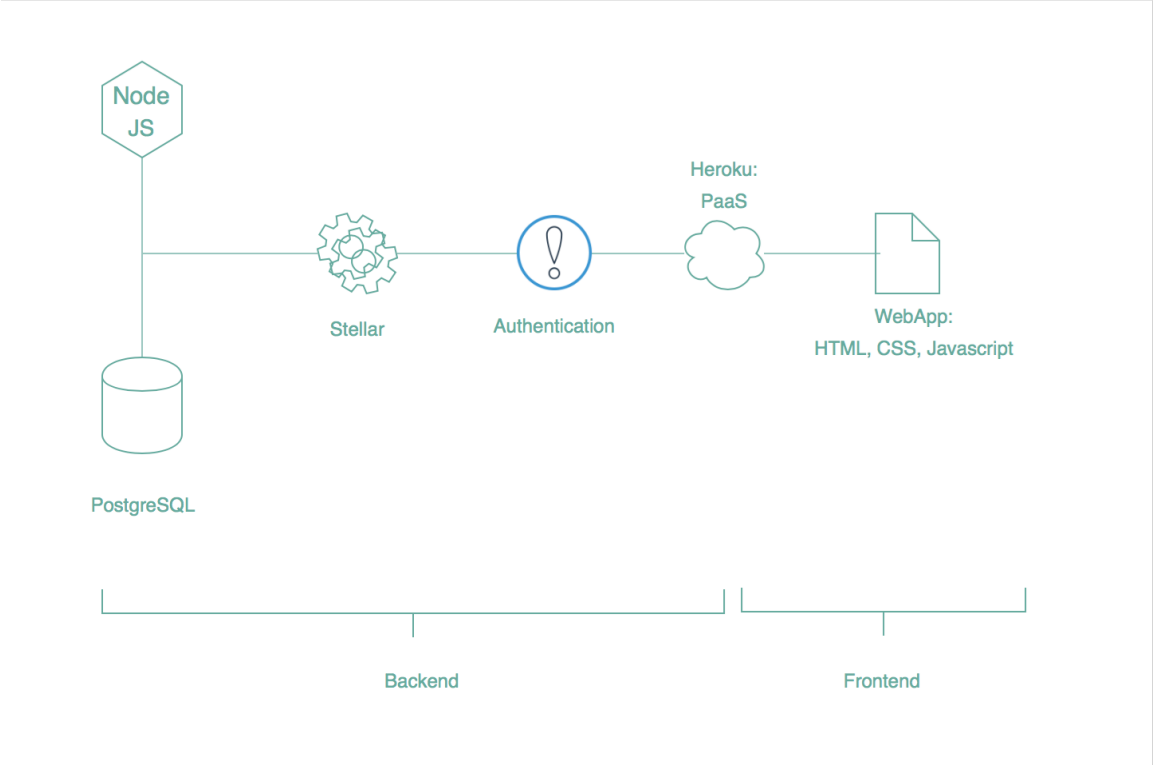

Figure 4.1: Implementation Overview

Figure 4.1 shows the relationships between different components of the software are how they would be connected.

4.2 Implementation Setup

The next section of this project focuses on how the software product is setup. This section also discusses the rationale behind implementation choices and decisions made, as well as brief descriptions of the various layers of implementation.

4.2.1 Stellar core

The stellar core is the backbone of the stellar network. Essentially, it keeps a local copy of the digital ledger to keep the user's wallet application synchronized with the public ledger and thus other wallets on the network.

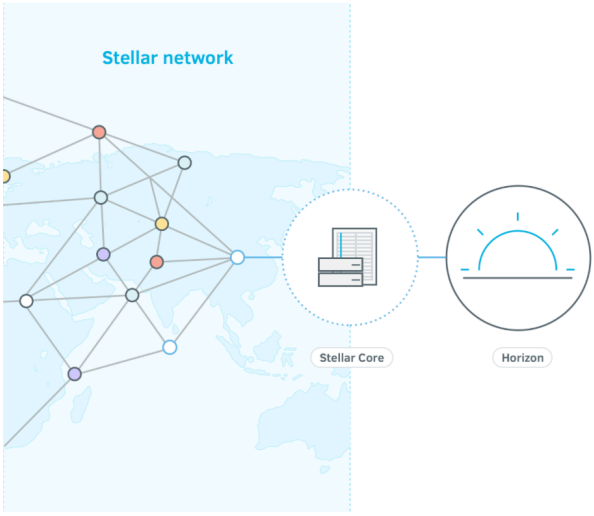

Figure 4.2 The position of the stellar core with respect to the stellar system
Picture credit: stellar.org

Figure 4.2 describes where the ledger/stellar core functions with respect to the architecture. To set up, it is important to install and import the stellar sdk into your project. This project uses JavaScript and its libraries and variations. We would also be using the stellar test-net, as this allows us to bypass some of the Stellar's regulations and gain access to test credit. The Stellar test-net periodically faces wipes and therefore, it

should not be alarming when some previously created accounts cease to function. The Stellar API discusses how to connect the test-net server, along with what port and the 3 public key's for validation to reach the server.

4.2.2 Horizon API

This is a RESTful API provided by stellar for submitting transactions to the stellar network. It is important to note however that transactions for our implementation refers to money. As discussed in chapter 1, transactions could be money, properties, securities etc. It supports the construction of user applications against the stellar network. This provides the HTTP communication methods for submitting transactions, check accounts etc. This API strictly works only once the Stellar core has been set up.

```
server.loadAccount(destinationId)
  // If the account is not found, surface a nicer error
message for logging.
  .catch(StellarSdk.NotFoundError, function (error) {
    throw new Error('The destination account does not
exist!');
  })
  // If there was no error, load up-to-date information on
your account.
  .then(function() {
    return server.loadAccount(sourceKeys.publicKey());
  })
  .then(function(sourceAccount) {
    // Start building the transaction.
    var transaction = new
StellarSdk.TransactionBuilder(sourceAccount)
      .addOperation(StellarSdk.Operation.payment({
        destination: destinationId,
        // Because Stellar allows transaction in many
currencies, you must
        // specify the asset type. The special "native"
asset represents Lumens.
        asset: StellarSdk.Asset.native(),
        amount: "10"
      }))
```

```
      // A memo allows you to add your own metadata to a
transaction. It's
      // optional and does not affect how Stellar treats
the transaction.
      .addMemo(StellarSdk.Memo.text('Test Transaction'))
      .build();
    // Sign the transaction to prove you are actually the
person sending it.
    transaction.sign(sourceKeys);
    // And finally, send it off to Stellar!
    return server.submitTransaction(transaction);
  })
  .then(function(result) {
    console.log('Success! Results:', result);
  })
  .catch(function(error) {
    console.error('Something went wrong!', error);
  });
```

Listing 4.1 Stellar SDK snippet of making a transaction

In listing 4.1, we see in more detail what is meant by an operation exists within a transaction. The instruction is as follows: The first step is to fetch an account on the stellar test-net. Next, the code creates the transaction object, and specify what type of asset(currency) is being issued. In this case it used lumens since it is using the base implementation. The next step is to add a message that describes the transaction being created. Finally, it is signed with account keys to verify the transaction object. Similarly to listing 4.1, we can build a request module but making alterations to the code to check tokens and adjust how the money is sent. Stellar provides a supportive means by which many operations can be built depending on the requirements of the software. These include: creating simple accounts, making payment, managing offers, changing trust, managing data amongst others. All the provided operations are lean implementations. More security and details for added meaning and communications for application use are left to the developer to build.

4.2.4 Database

The database that we would use for this project would be PostgreSQL v9.6. This is because the stellar core used for this implementation has better support for a PostgreSQL database.

4.2.3 Backend

The backend implementation would be developed using Node.JS v6.9.5. There is great support for Node.JS for all the requisite functionality. The project implementation involves authentication using bcrypt. Once a user logs in, a Java Web Token is issued as well. The backend involves password hashing and comparison. The payments, token verifications to make actions possible, creating, accessing and retrieving user accounts, viewing transactions and requests, and responding to them are largely implemented through abstraction of the Horizon API form the stellar SDK. They are our API endpoints, which we utilize AJAX to make requests to and from. Figure 4.3 below lists the requisite libraries that have been added to create a usable level software application.

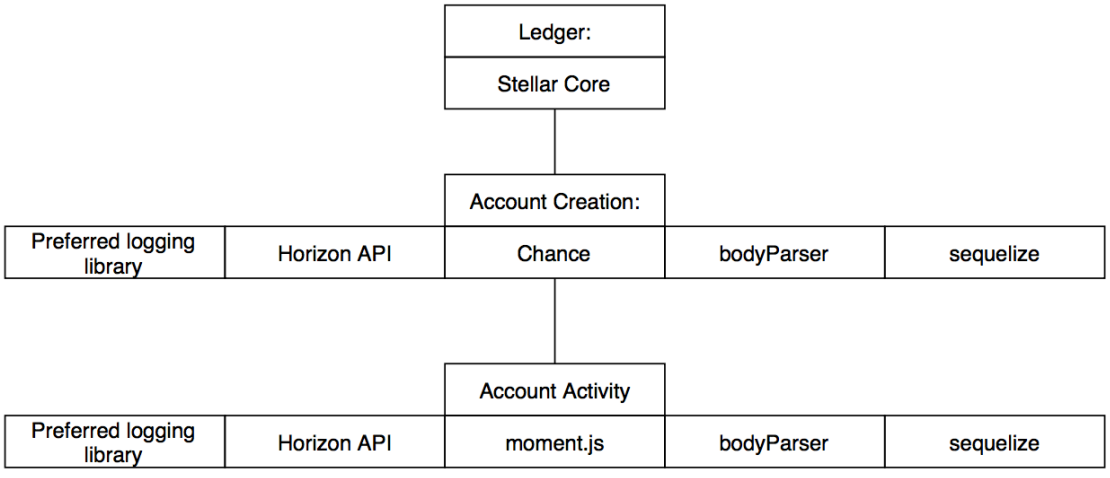

Figure 4.3: Requisite JavaScript libraries to particular activities

The libraries below can be installed using `[npm --install [library name]]` and imported into the target application:

Chance npm: Random generator of Strings, numbers etc. used in wallet account creation to give personalized and real details for each users' wallet. This randomized information can be edited within application interface.

Body Parser: The creation of transactions and data results is a lot of JSON objects for accounts, users, etc. being created and passed for information transfer. This library assists with parsing json.

Sequelize: This is an object relational mapping language for Node.JS which the implementation uses to communicate with the PostgreSQL database.

Moment: this is for parsing, validating and displaying dates for the JSON objects that we create. A time log of actions is crucial in any application, and even more crucial in a financial application.

4.2.3.1 Making a payment backend

Stellar SDK's tutorial method of making a payment was discussed above in Listing 4.1. In this section, we would discuss how this project executes the same action to see the distinction between the two.

```
app.use('/payments', jwtCheck);
app.post('/payments', function(req, res) {

    logger.profile("send_payments");
    const _user = req.user;
    const { phone, amount, note, action, currency } = req.body;

    if (!(action === 'pay' || action === 'request')) {
        res.status(401).send({'error': "Invalid action parameter.
```

```
Pass in 'pay' or 'request' only"});
        return;
    }

    let fetchedUser;

    // Finding the User that sent this request
    User.findById(_user.id)
    .then(user => {
        fetchedUser = user;
        return validateTargetNumber(phone);
    }).then(target => {
        console.log("Target validated: ", target);

        // If the target is the current user, return early.
        if (target.user.id === _user.id) {
            if (action === 'pay') {
                throw new Error("You cannot pay money to
yourself");
            } else {
                throw new Error("You cannot request money from
yourself");
            }
        }
        return createTransactionObject(action, currency, amount,
note, fetchedUser, target);
    })
    .then(transaction => {
        // Should we process this transaction...
        // If this was a pay request, attempt to process right
away.
        if (transaction.action == 'pay') {
            return processTransaction(transaction);
        } else {
            return Promise.resolve(transaction);
        }
    })
    .then(result => {
        logger.profile("send_payments");
        res.status(200).send({ data: result });
    })
    .catch(error => {
        res.status(400).send({ error: error.message });
    });
});
```

Listing 4.2: Project implementation of payment module in backend – API endpoint

In the above code snippet, we have the declaration for the front end to know what URL to request the payment from as:

```
app.use('/payments', jwtCheck);
app.post('/payments', function(req, res) {
```

Listing 4.3: Node.js format of making function an end-point

Once this happens, the parameter of the AJAX call from the front end goes through the following:

```
if (!(action === 'pay' || action === 'request')) {
        res.status(401).send({'error': "Invalid action parameter.
Pass in 'pay' or 'request' only"});
        return;
    }

    let fetchedUser;

    // Finding the User that sent this request
    User.findById(_user.id)
    .then(user => {
        fetchedUser = user;
        return validateTargetNumber(phone);
    }).then(target => {
        console.log("Target validated: ", target);

        // If the target is the current user, return early.
        if (target.user.id === _user.id) {
            if (action === 'pay') {
                throw new Error("You cannot pay money to
yourself");
            } else {
                throw new Error("You cannot request money from
yourself");
            }
        }
        return createTransactionObject(action, currency, amount,
note, fetchedUser, target);
    })
```

Listing 4.4 Project implementation of payment module in backend – Deciphering actions

The code snippet above in Listing 4.4 deciphers which action to take. It checks whether the user intended to make a payment or to make a request for money. Previously as discussed, the software allows the user to make requests as well, and must be implemented as such. Once the user to send to has been fetched along with other details about the transaction, it must then be processed. This is done in the back-end in the figure

below:

```
function processTransaction(transaction) {
    return new Promise((resolve, reject) => {

        const status = transaction.status;
        if (status == 'settled' || status == 'declined' || status
== 'cancelled') {
            throw new Error(`Invalid tx status - transaction
already ${status}`);
        }

        const currency = transaction.currency;
        const amount = transaction.amount;
        const note = transaction.note;

        let payer;
        let destinationId;

        fattenTransaction(transaction)
        .then(tx => {

            if (tx.action === 'pay') {
                payer = tx.actor;
                destinationId = tx.target.user.purse.accountId;
            } else if (tx.action == 'request') {
                payer = tx.target.user;
                destinationId = tx.actor.purse.accountId;
            }

            return _processTransaction(transaction, payer,
destinationId, currency, amount, note);
        })
        .then(result => {
            resolve(result);
        })
        .catch(error => {
            reject(error);
        });
    });
}
```

Listing 4.5: Project implementation of payment module in backend – Processing payment

The code snippet above is involved in processing payments from one account to another. The transaction object created in the /payments API endpoint, is passed into the above function. The transaction is given a status and ‘fattened.’ The fatten method simply pulls

user objects into the transaction and returns that back into the API request. Once the destination and source is also verified it enters another function to process called the _processTransaction(), where the transaction is processed utilizing the necessary keys and stellar SDK elements in completing the payment.

```
function _processTransaction(transaction, payer, destinationId,
currency, amount, note) {
    return new Promise((resolve, reject) => {
        const payerSeed = payer.seed;
        if (!payerSeed) {
            throw new Error("No payer seed provided");
        }

        const payerAccountId = payer.purse.accountId;
        if (!payerAccountId) {
            throw new Error("No payer accountId provided");
        }

        let balance;

        // Get all wallet keypairs for the payer
        const sourceKeys = Stellar.Keypair.fromSeed(payerSeed);

        // Make sure destinationId can be loaded properly
        stellar.loadAccount(destinationId)
        .then(destinationAccount => {
            // TODO: Do some regulatory checks right here, but
not in this implementation.

            // Then load up to date information on the account
that's sending
            return stellar.loadAccount(sourceKeys.accountId());
        })
        .then(sourceAccount => {
            // Check if this account has enough funds
            let _balance = _.find(sourceAccount.balances,
(balance) => {
                return balance.asset_type === 'native';
            });

            let present = parseInt(_balance.balance);
            let required = parseInt(amount);
            if (present < required) {
                throw new Error("Insufficient Funds");
            }
```

```
            // Start building the transaction
            // This requires the account object proper, not just
the ID. Hence we need to have the sourceKeys.
            // It will increment the accounts sequence number
            const stellarTransaction = new
Stellar.TransactionBuilder(sourceAccount)
            .addOperation(
                Stellar.Operation.payment({
                    destination: destinationId,
                    asset: Stellar.Asset.native(), // currency
                    amount: amount
                })
            )
            .addMemo(Stellar.Memo.text(`chp_${transaction.id}`))
            .build();

            // Sign the transaction to prove you're actually the
one doing this.

            stellarTransaction.sign(sourceKeys);

            return stellar.submitTransaction(stellarTransaction);
        })
        .then(result => {
            console.log("Sucessful Stellar transaction: ",
result);
            return fetchWalletBalance(payer);
        })
        .then(_balance => {
            console.log("Balance: ", _balance);
            balance = _balance;
            return transaction.update({
                status: "settled",
                completedAt: Date.now()
            });
        })
        .then(transaction => {
            resolve({transaction, balance});
        })
        .catch(error => {
            console.error("Error processing this transaction: ",
error);
            reject(error);
        });
    });
}
```

Listing 4.6: Project implementation backend – Final payment processing

Listing 4.6 is the final stage of the payment process. Here, the software checks to see that

the payment is coming from an existing wallet by checking the seed. Next, the requisite account details are acquired, and the transaction object to be built on stellar are sent through to the Stellar core.

4.2.5 Web Application

This is the front-end of the application and the side the user would be presented. It makes AJAX requests to our backend. The Heroku cloud service hosts this API endpoints from where we make these AJAX requests. This project uses heroku-cli/5.6.14-b0cc983 (darwin-amd64) go1.7.4

4.2.5.1 Pay and request

```
function Sender()
          {
            alert("Please wait while your transaction is
processing.");

            var receiver=$("#receiver").val();
            var amount=$("#amount").val();
            var note=$("#note").val();

            $.ajax({
                    async: false,
                    url: "http://chipper.herokuapp.com/payments",
                    data: { 'phone': receiver, 'amount': amount,
'currency':'GHS', 'note': note, 'action': 'pay'},
                    type: "POST",
                    beforeSend:
function(xhr){xhr.setRequestHeader('Authorization', 'Bearer
'+sessionStorage.token);},
                    success: function(response)
                     {
                       console.log('Success, ',  response);
                       alert("Money has been sent successfully
to: "+ receiver);
    //Handle how you want the payment completion to be done!
                     },
                     error: function(response)
```

```
                        {
                          var obj =
$.parseJSON(response.responseText);
                          alert(obj.error);

                        }
                    });

                    $("#receiver").val("");
                    $("#amount").val("");
                    $("#note").val("");

                    updateBalance();
            }
```

Listing 4.7: Front-end AJAX request to /payments

The above code snippet shows the process used for making an API post request to Heroku, which hosts our backend, for making a payment to a beneficiary user account. Using a sender's bearer token, formatted data, user account information money can be sent through a public key. Figure 4.7, combined with past included code gives a sense of how the implementation travels from the front-end to the backend and processes payments accordingly.

```
  function Requester()
               {

         alert("Please wait while your transaction is
processing.");

                 var receiver=$("#Rreceiver").val();
                 var amount=$("#Ramount").val();
                 var note=$("#Rnote").val();

                  $.ajax({
                         async: false,
                         url:
"http://chipper.herokuapp.com/payments",
                         data: { 'phone': receiver, 'amount':
amount, 'currency':'GHS', 'note': note, 'action': 'request'},
                         type: "POST",
                         beforeSend:
function(xhr){xhr.setRequestHeader('Authorization', 'Bearer
'+sessionStorage.token);},
                         success: function(response)
```

```
                        {
                  //Handle success JSON object
                  console.log('Success, ',  response);
                   alert("Money has been requested successfully
to: "+ receiver);
                        },
                        error: function(response)
                        {
          var obj = $.parseJSON(response.responseText);
                          alert(obj.error);

                        }
                    });


                    $("#Rreceiver").val("");
                    $("#Ramount").val("");
                    $("#Rnote").val("");

                    updateBalance();

               }
```

Listing 4.8: Front-end AJAX request to payments alternative

Listing 4.8 demonstrates when a request action is sent instead of a pay action. It goes through similar processes. However, the difference would happen in the processTransaction() in the backend, where the request would be processed accordingly.

4.2.5.2 Transaction history

```
    function ShowMyTransactions()
      {
        var amount;
        var completedAt;
        var currency;
        var description;
        var note;

        // alert("fetching your transactions...");

        $.ajax({

                // async: false,
                url: "http://chipper.herokuapp.com/requests",
                // data: { 'phone': receiver, 'amount': amount,
'currency': 'GHS', 'note': note, 'action': 'pay'},
```

```
                type: "GET",
                beforeSend:
function(xhr){xhr.setRequestHeader('Authorization', 'Bearer
'+sessionStorage.token);},
                success: function(response)
                {
                  console.log('response', response);
                  requestdisplay = "<center><ul style='list-
style-type:none'><br>";

                                    let transactions =
response.data.transactions.length;

                                    if(transactions == 0){
                                      requestdisplay =
requestdisplay + "<li>No pending transactions</li>";
                                    }
                                    else
                                    {
                                      for(i=0; i< transactions;
i++)
                                      {

                                        requestdisplay =
requestdisplay + "<li
id="+response.data.transactions[i].id+">"+response.data.transacti
ons[i].description+
                                        " <div
style='position:relative'> <a style='padding-left: 10px; display:
inline;' id="+response.data.transactions[i].id+" href=#
onclick='payme(this)'><img src='approve.png' height='40'></a><a
style='padding-left: 10px;'href=# onclick='cancelme(this)'
id="+response.data.transactions[i].id+"><img src='disapprove.png'
height='40'></a> </li> </div> <br>";

                                      }
                                    }

                    requestdisplay = requestdisplay +
"</ul></center>";

document.getElementById('pendingcontent').innerHTML =
requestdisplay;
                  },
                  error: function(response)
                  {
                    var obj = $.parseJSON(response.responseText);
                    alert(obj.error);

                  }
              });
```

```
        }
```

Listing 4.9: Front-end AJAX request to /requests

The above figure was included to give another sense of the thinness of the client. Every action is made by simply making AJAX requests to the server. Appendix A holds all built API's to the server that were implemented. This particular code snippet in Figure 4.10 allows a user to view requests made to the users wallet. The token is what is used to maintain the current end-user and fetch the requisite transaction list through the backend.

4.2.6 Database

The database is queried using sequelize in Node.JS. Additionally, PostgreSQL and Postico, a Mac OSX is used for database viewing, hosting, creation and management. This makes for ease of software building. The database information is fed into the settings to generate a good UI for ease of viewing; as shown in Figure 4.4

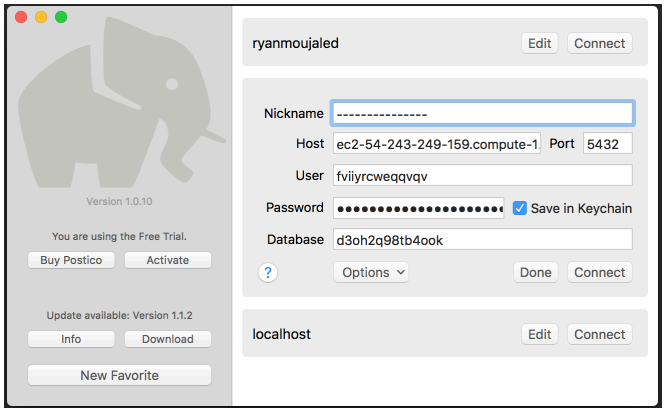

Figure 4.4: Postico PostgreSQL DB connection

4.2.7 Software

The user interface is built based off a web dashboard template from Creative-Tim.

However, modifications have been made to suit the needs of the wallets' implementation with HTML for markup, CSS for design and style, JavaScript and JQuery for functionality.

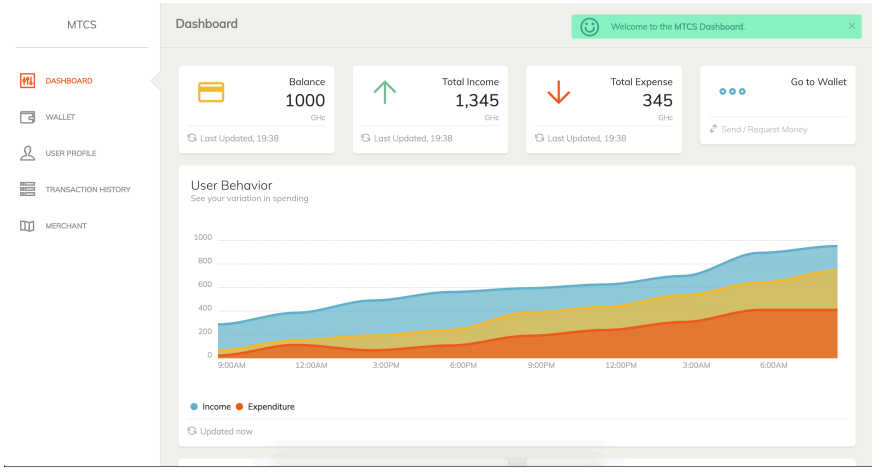

Figure 4.5: Wallet Dashboard User Interface – Part A

In Figure 4.5 above shows the user interface that the user is presented with after authorization is granted. The user is shown their total amount of money, total expenditure and total money that has come into the wallet. As well as graphs and a quick card button to access the wallet, with a navigation pane on the left.

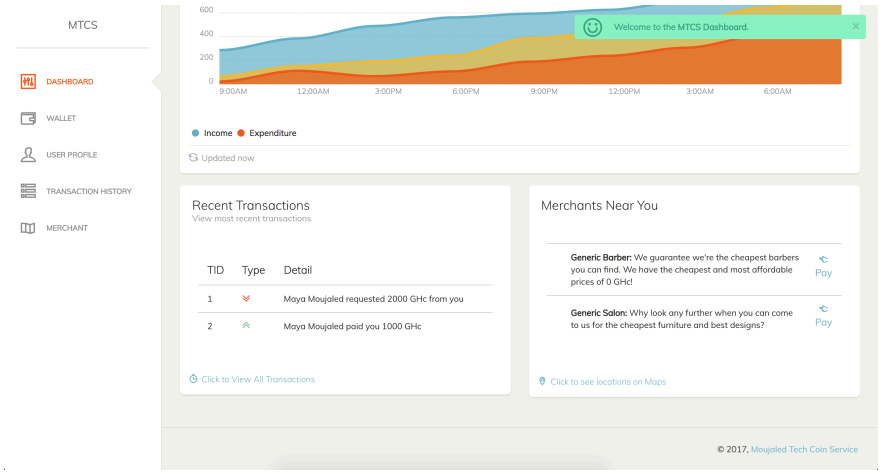

Figure 4.6: Wallet Dashboard User Interface – Part B

The above image shows the lower part of the same page. In figure 4.6 above the page has a limited view of past transactions, with a link to view all transactions. This is the same with the merchant list.

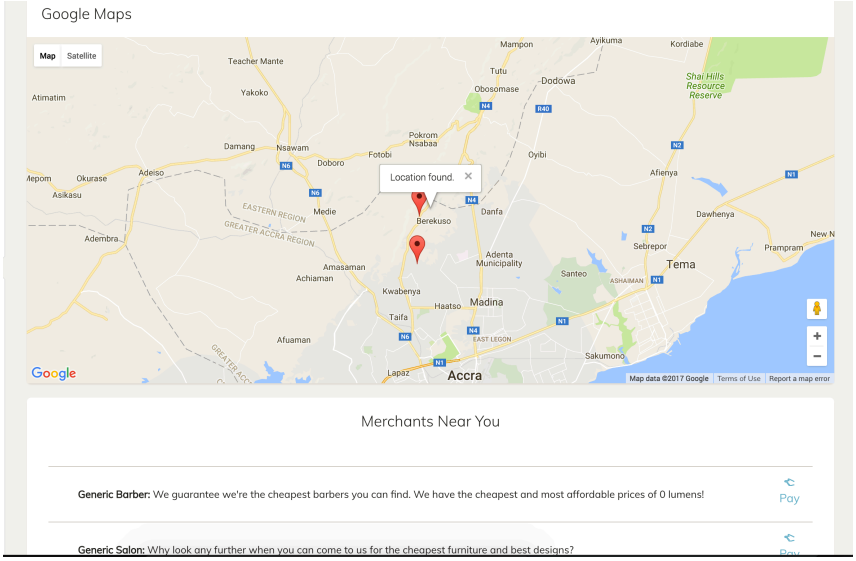

Figure 4.7: Wallet Merchant Page User Interface

Figure 4.7 shows us the merchant page. The merchant page is fitted with the google map API for placing merchants on the map. This is to give the user a sense of where the merchant is located relative to the users' location. Only merchants nearby are shown and not merchants all over the map. All merchants that show up on the map are also displayed down below for payment accessibility.

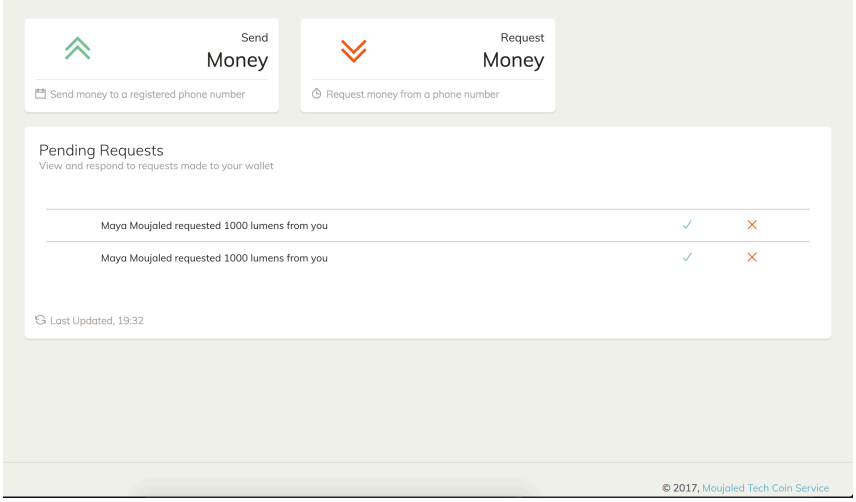

Figure 4.8: Wallet Merchant Page User Interface

Figure 4.8 above shows the page for making payments and requests, as well as accepting or declining incoming requests made to you. The user interface provides the user with the requisite buttons to enact these operations and transactions accordingly.

CHAPTER 5: TESTING

5.1 Web Application Testing

The application was tested with the web browsers console to ensure that quintessential information is passed by the application. This was done by console logging the crucial points where the client communicates with the server in our code. The token is also

console logged to ensure that the server and client is still aware that a user is present. The token has been mentioned prior to this section. However, the token is effectively what the server uses to verify that, indeed a created and validated user is sending this object through. Figure 5.1 shows a success JSON object, followed by the assigned token that the user bears. This token is what allows the wallet to make changes. The response object is what also confirms that out request to the server met a positive result. Figure 5.1 shows the components of a successful AJAX request for the implementation, that we generate within our web browsers console.

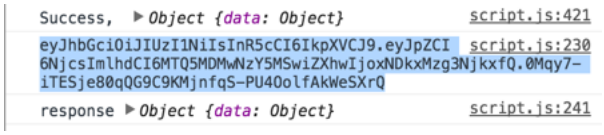

Figure 5.1: console log showing tokens being passed

5.2 System Testing

This section contains various tests enacted on the system to envision its effectiveness

5.2.1 Response Time

The response time of the web application is very important. It is imperative that timely access to the wallet be as small as possible from the users point of view. The access into the wallet completes in under five seconds - dependent on internet connectivity. This was measured using a real-time clock. Similar investigational methods were used in sending and requesting money and provide execute generally in under 10 seconds. A time of under 30 seconds is acceptable, beyond that the action fails.

5.2.2 Conflict Resolution

Tests of the application demonstrate that the system handles conflicts effectively using node.js promises in the backend. Conflicts are essentially mismanagement of payments – such as dual statuses because of duplicate payment creation. This typically happens in the backend, payments must be assigned their relevant status on the backend for all users and not as per the two transacting users. The core idea is to give every transactional and operational element statuses of pending, fulfilled and rejected.

- pending - The initial state of a promise.
- fulfilled - The state of a promise representing a successful operation.
- rejected - The state of a promise representing a failed operation.

5.3 Requirements Testing

The requirements testing was done using Postman v4.9.3. This is a professional tool for developing and testing APIs to verify they function as required.

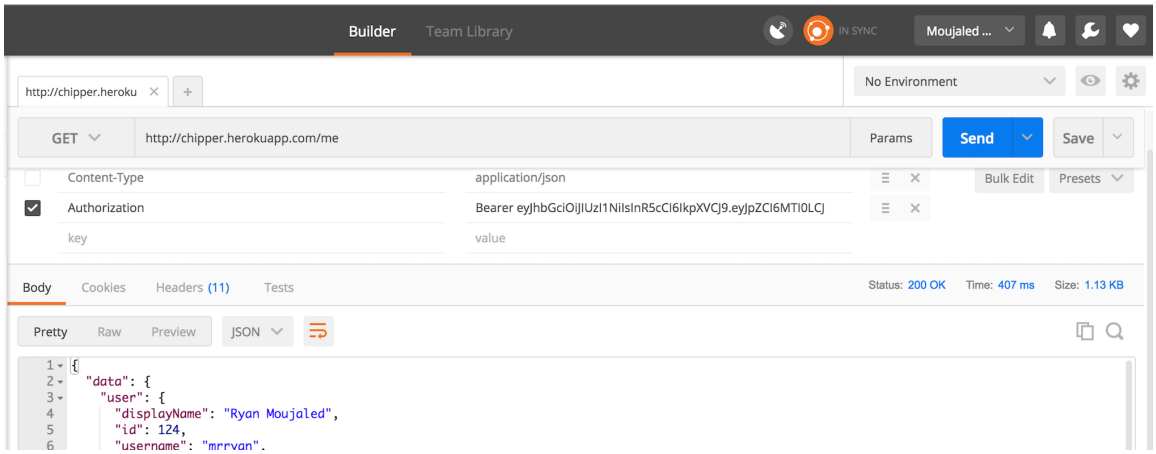

Figure 5.2: Postman test of /me API endpoint

Figure 5.2 shows to retrieve a current user through the GET /me API endpoint. The body below shows a JSON object returning the details of a user. It is important to note that the random String and Integers are sometimes are a result of Chance js – which was discussed in the implementation.

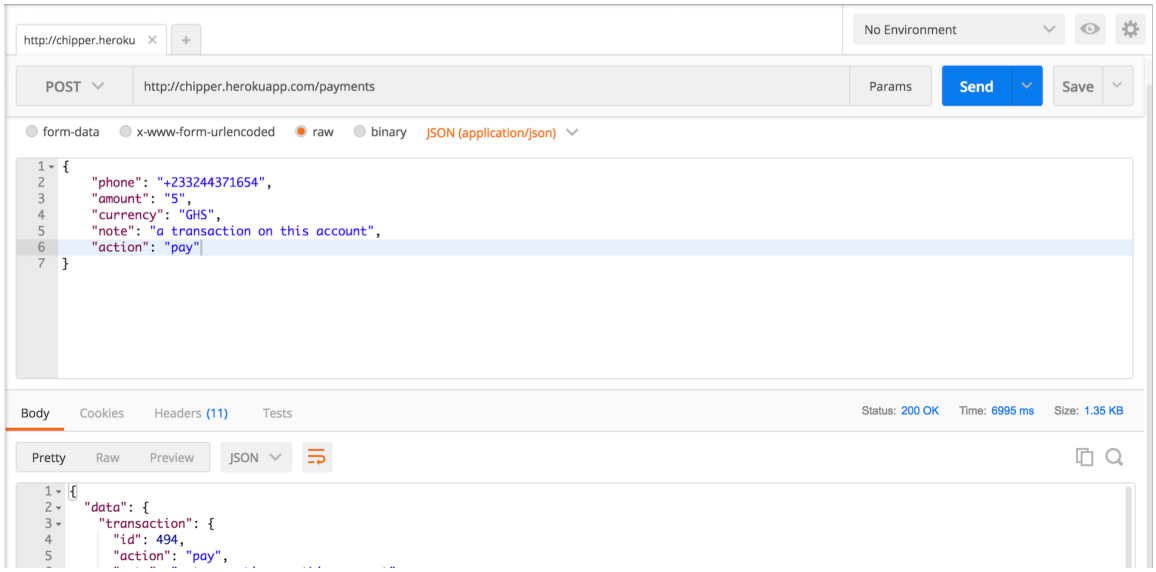

Figure 5.3: Postman test of /payments API endpoint

The figure above shows a test of the POST /payments endpoint. This demonstrates that a user can successfully send money. The JSON Object returned is the created transaction object that is returned. This endpoint is also how merchants are paid.

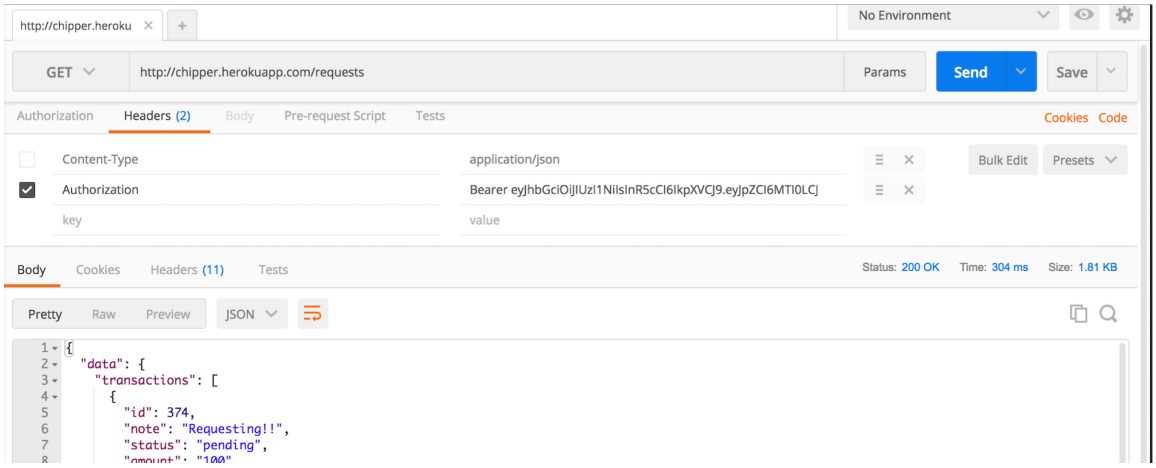

Figure 5.4: Postman test of /requests API endpoint

Figure 5.4 shows the retrieval of requests test. These requests responded to in the wallet by accepting or declining – these requests too are transactional objects. As mentioned in the implementation stage; requests are modified 'send money' payments requests but with locks to support the accept or decline functionality.

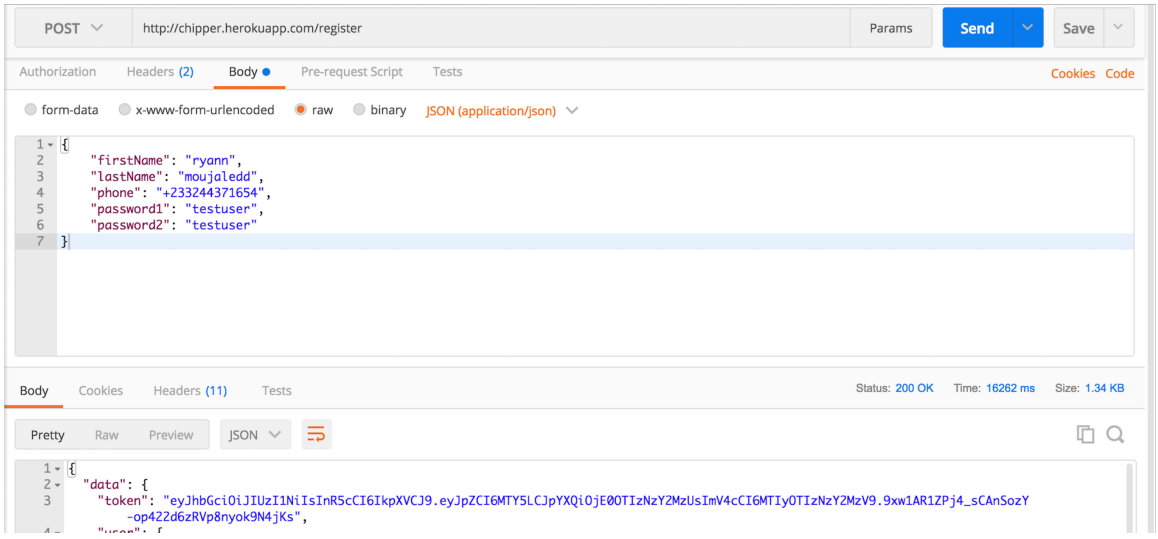

Figure 5.5: Postman test of /register API endpoint

The figure above shows a test for registering a user. The user is given a token, as evident in the first line of the JSON response. That token is what allows a user to continually perform actions within the application. The token has an inactivity timer, at which point it ceases to function. Logging in, sending payments, requesting and viewing payments all require the token. This token recycles. This end-point is also used for registering merchants.

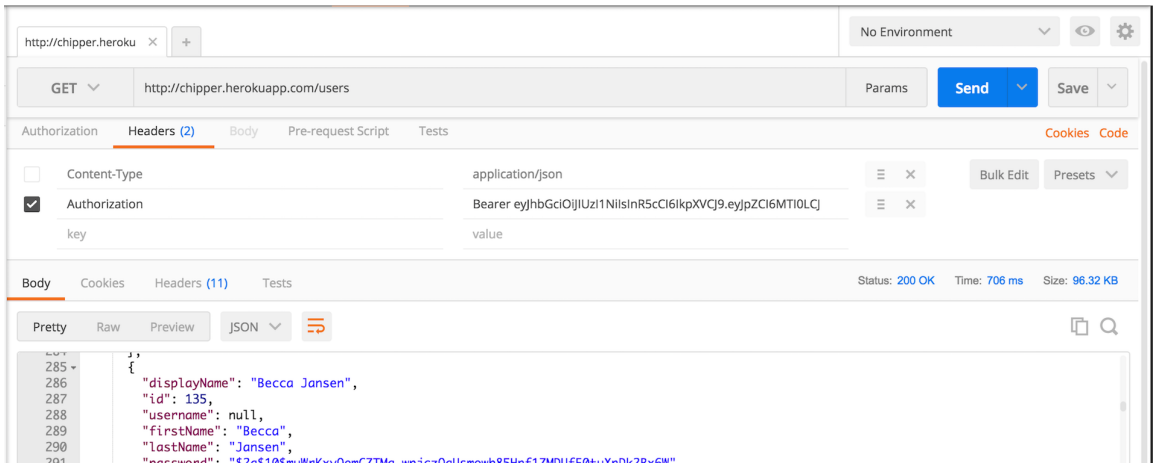

Figure 5.6: Postman test of /users API endpoint

This is testing the admin end-point to retrieve all users (merchants as well) that have a wallet on the database and gain access to view required information.

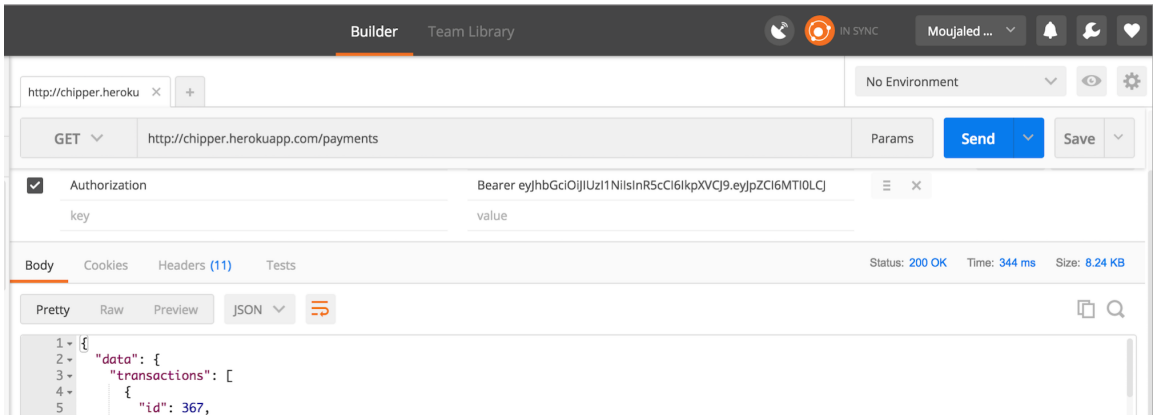

Figure 5.7: Postman test of listing transactions

The above, Figure 5.7 uses the GET /payments API endpoint to fetch all transactions made by the user. This also requires the accounts token.

Requirement	Test Reference	Result (Status / Time)
Register an account	Postman /register test	JSON Object returned successfully (200 OK / 16262 ms)
Send, request and receive money	Postman POST /payments test. Required: Bearer: <token>	JSON Object returned successfully (200 OK) Request: 304 ms Send Money: 6995 ms
View transactions	Postman GET /payments test. Required: Bearer: <token>	JSON Object returned successfully (200 OK/344 ms)
Fetch transactions where the user is the requestor	Postman GET /requests?isrequestor=true test. Required: Bearer: <token>	JSON Object returned successfully (200 OK / 779ms)

Fetch transactions where the user is not the requestor	Postman GET /requests?isrequestor=false test. Required: Bearer: <token>	JSON Object returned successfully (200 OK / 232ms)
Send Money	Postman POST /payments, test. Required: Bearer: <token>	JSON Object returned successfully (200 OK / 304 ms)
Respond to request	Postman PUT /payments/374	JSON Object returned successfully (200 OK / 872 ms)
Fetch self user information	Postman GET /me test. Required: Bearer <token>	JSON Object returned successfully (200 OK / 407 ms)

Table 5.1: Table showing testing of various requirements.

Table 5.1 Shows a list of various end-points that were created with all components of the system. The API tests with Postman show that all API endpoints work satisfactorily with statuses of 200 OK

CHAPTER 6: CONCLUSION

6.1 Summary

The software meets a large part of specified requirements and non-functional requirements. It successfully sends, requests and receives money. Additionally, the software suitably displays a user's wallet activity, along with requests to be confirmed and declined. Finally, it suitably meets merchant searching and merchant payment.

6.2 Future Work

The work has much room for growth and additions. These additions include API's such as Stripe to facilitate debit and credit card additions to the software. This way the application can be taken off test-net and implemented for real world use. Stripe's API in the possible future would enable users to use their credit cards to purchase currency supported by the virtual wallet for further real-word digital use. Furthermore, Flutterwave is also a possible addition API as a gateway to banks and beneficiary payments. Flutterwave presents a new, modern and simple solution to traditional money exchange. They serve as a gate way, eliminating the risk management and processing required by a developer straight to a bank or blockchain dependent on the traded currency, and then directly to the beneficiary. Furthermore, other developers on the stellar network can be liaised with to enable money to be transacted between them to test gateway procedure for stellar distributed exchange use.

Bibliography

Blockgeeks. (2017). *What is Blockchain Technology? A Step-by-Step Guide For Beginners*. Retrieved 24 March 2017, from https://blockgeeks.com/guides/what-is-blockchain-technology/.

Bohr, J., & Bashir, M. (2014). Who Uses Bitcoin? An exploration of the Bitcoin community. *2014 Twelfth Annual International Conference On Privacy, Security And Trust*. http://dx.doi.org/10.1109/pst.2014.6890928.

Decker, C., Seidel, J., & Wattenhofer, R. (2016). Bitcoin meets strong consistency. *Proceedings Of The 17Th International Conference On Distributed Computing And Networking - ICDCN '16*. *http://dx.doi.org/10.1145/2833312.2833321*.

European Central Bank. (2012). Virtual Currency Schemes. *European Central Bank* Retrieved 14 April 2017, from http://www.ecb.europa.eu/pub/pdf/other/virtualcurrencyschemes201210en.pdf.

Gervais, A., Karame, G., Wüst, K., Glykantzis, V., Ritzdorf, H., & Capkun, S. (2016). On the Security and Performance of Proof of Work Blockchains. *Proceedings Of The 2016 ACM SIGSAC Conference On Computer And Communications Security - CCS'16*. http://dx.doi.org/10.1145/2976749.2978341.

Graydon C. (2014). *What is Cryptocurrency?*. Retrieved 24 March 2017, from https://www.cryptocoinsnews.com/cryptocurrency/.

Hari, A., & Lakshman, T. (2016). The Internet Blockchain. *Proceedings Of The 15Th ACM Workshop On Hot Topics In Networks - Hotnets '16*. http://dx.doi.org/10.1145/3005745.3005771.

Hurlburt, G., & Bojanova, I. (2014). Bitcoin: Benefit or Curse?. *IT Professional*, *16*(3), 10-15. http://dx.doi.org/10.1109/mitp.2014.28.

Judmayer, A., & Weippl, E. (2016). Condensed Cryptographic Currencies Crash Course (C5). *Proceedings Of The 2016 ACM SIGSAC Conference On Computer And Communications Security - CCS'16*. http://dx.doi.org/10.1145/2976749.2976754

Karame, G. (2016). On the Security and Scalability of Bitcoin's Blockchain. *Proceedings Of The 2016 ACM SIGSAC Conference On Computer And Communications Security - CCS'16*. http://dx.doi.org/10.1145/2976749.2976756.

Lindesay, F. (n.d.). *Promises*. Retrieved 24 March 2017, from https://www.promisejs.org.

Maesa, D., Marino, A., & Ricci, L. (2016). Uncovering the Bitcoin Blockchain: An Analysis of the Full Users Graph. *2016 IEEE International Conference On Data Science And Advanced Analytics (DSAA)*. http://dx.doi.org/10.1109/dsaa.2016.52.

Maxwell, D., Speed, C., & Campbell, D. (2015). 'Effing' the ineffable. *Proceedings Of The 2015 British HCI Conference On - British HCI '15*. http://dx.doi.org/10.1145/2783446.2783593.

Morrison, A., & Sinha, S. (2017). *A primer on blockchain (infographic)*. *Usblogs.pwc.com*. Retrieved 24 March 2017, from http://usblogs.pwc.com/emerging-technology/a-primer-on-blockchain-infographic/.

Nguyen, Q. (2016). Blockchain - A Financial Technology for Future Sustainable Development. *2016 3Rd International Conference On Green Technology And Sustainable Development (GTSD)*. http://dx.doi.org/10.1109/gtsd.2016.22.

Robert, J., Kubler, S., & Traon, Y. (2016). Micro-billing Framework for IoT: Research & Technological Foundations. *2016 IEEE 4Th International Conference On Future Internet Of Things And Cloud (Ficloud)*. http://dx.doi.org/10.1109/ficloud.2016.50.

Samaniego, M., & Deters, R. (2016). Using Blockchain to push Software-Defined IoT Components onto Edge Hosts. *Proceedings Of The International Conference On Big Data And Advanced Wireless Technologies - BDAW '16*. http://dx.doi.org/10.1145/3010089.3016027.

Shehhi, A., Oudah, M., & Aung, Z. (2014). Investigating factors behind choosing a cryptocurrency. *2014 IEEE International Conference On Industrial Engineering And Engineering Management*. http://dx.doi.org/10.1109/ieem.2014.7058877.

Stellar Organization. (2017). *Lumens FAQ - Stellar*. Retrieved 24 March 2017, from https://www.stellar.org/lumens/.

Stellar Organization. (2017). *Stellar*. Retrieved 24 March 2017, from https://www.stellar.org/how-it-works/stellar-basics/#.

Three-Layered Services Application. (2017). *Msdn.microsoft.com*. Retrieved 24 March 2017, from https://msdn.microsoft.com/en-us/library/ff648105.aspx.

Underwood, S. (2016). Blockchain beyond bitcoin. *Communications Of The ACM*, *59*(11), 15-17. http://dx.doi.org/10.1145/2994581.

European Parliament. (2017) Virtual currencies: what are the risks and benefits*? European Parliament . Retrieved 14 April 2017, from http://www.europarl.europa.eu/news/en/news-room/20160126STO11514/virtual-currencies-what-are-the-risks-and-benefits.*

Watanabe, H., Fujimura, S., Nakadaira, A., Miyazaki, Y., Akutsu, A., & Kishigami, J. (2015). Blockchain contract: A complete consensus using blockchain. *2015 IEEE 4Th Global Conference On Consumer Electronics (GCCE)*. http://dx.doi.org/10.1109/gcce.2015.7398721.

World Economic Forum. (2017). All you need to know about blockchain, explained simply Retrieved 14 April 2017, from https://www.weforum.org/agenda/2016/06/blockchain-explained-simply.

Appendix A – API Documentation

Api Endpoint: http://chipper.herokuapp.com/

POST /register
To register a new account on Chipper
Save <token> on Client
Pass in the "Authorization: Bearer <token>" in the HTTP Header of every single request.

```
Body {
      firstName: <firstName>,
      lastName: <lastName>,
      phone: <phone_number>,
      password1: <password_1>,
      password2: <password_2>
}
```

Password1 and password2 must match each other, server will verify it as well..

Returns

```
data {
      token: <token>,
      user: <user>
      balance: <balance>,
}
```

POST /login
Will log in the user.
Save <token> on Client
Pass in the "Authorization: Bearer <token>" in the HTTP Header of every single request.

```
Body {
      phone: <phone_number>,
      password: <password>,
}
```

Returns

```
data {
      token: <token>,
      user: <user>
      balance: <balance>,
}
```

GET /me

Get details about the user, Eg. Refreshing the user etc. Requires Bearer Token

Returns

```
{
      User: <user>,
      Balance: <balance>
}
```

POST /payments

Endpoint to make a payment to a user. Requires Bearer Token

Body:

```
{
      phone: <string of the phone number to send money to eg.
"23321232583>
      amount: <string of the amount to send money to. Eg. "100"
      currency: <string currency payment is in. e.g "GHS">
      note: <string Just a note given by the user. Eg. "For
buying me top up credit">
      action: <string, Either "pay" or "request". Only "pay is
supported right now>
}
```

Returns

```
data {
      transaction: <transaction> Entire Transaction endpoint,
      balance: <balance>
}
```

GET /payments

Endpoint to get a transaction history of the users completed transactions. Requires Bearer Token

transaction.description contains a description of this payment.

Eg. Ryan sent you 5 GHS.

You paid Maijid 2 GHS etc.

transaction.isCredit is a Boolean. True means, this transaction credited the wallet. False means otherwise.

Returns

```
data {
      transactions: [
            { … }
            { … }
            { id, note, currency, amount, target, actor,
description, isCredit, etc }
```

```
      ]
}
```

GET /requests
Endpoint to return a list of all the payment requests that this user has received. Requires Bearer Token

Pass in a query parameter “isRequestor” to return either payment requests that the user requested, or payment requests that someone else requested from the user.

GET /requests?isRequestor=true
GET /requests?isRequestor=false

Returns.
A list of “pending” transactions in which the user could either “approve” or “decline” if she received the request.
Or cancel if she sent the request.
tx.description will give a description of this request. Eg. “David requested 20 GHS from you”

structure

```
{
Data: {
Transactions:
[
      {transaction_object},
      {transaction_object}
]}
}
```

PUT /payments/{transaction_id}
This is an endpoint to respond to a payment request.
You pass in the transaction.id into the url first.
Eg. /payments/85

Then, pass in the parameters to the end point. Only 3 are allowed.

```
“approve”, “decline”, “cancel”
{
      “action”: “approve”
```

```
}
```

Returns:
If action == cancel or decline. You'll get the updated transaction object back. With the status changed appropriately, and the date_completed timestamp updated.

```
Data: {
      Transaction: {
            Id,
            Status,
            completedAt:
      }
}
```

If action == approve. You'll get the **transaction** object, and the **balance** object showing the user's current balance.

```
data: {
      transaction: { … },
      balance: { … }
}
```

User End points. (They all require the Bearer Token)

GET /users
Returns an array of all users on Chipper.

GET /users/{userId}
Returns a specific user object when you pass in the userId

PUT /users/{userId}
Updates the user's public data information.
Pass in the body these keys:
about, firstName, lastName, username