DELOCK TECHNICAL SPECIFICATION

Project Title: Delock – decentralized rental platform

Student Name: Mark McAdam

Student Number: 14566803

Date Finished: 20th May 2018

Table of Contents

1. Introduction
    a. Overview
    b. Motivation
    c. Glossary
2. Research
    a. Blockchain
    b. Smart Contracts
    c. Platform
    d. Storage
3. System Architecture
    a. *High-Level design*
    b. *User Implementation*
    c. *Android Implementation*
    d. Smart Contract Implementation
4. Sample Code
5. Problems and Resolutions
6. Testing and Feedback
    a. Unit testing
    b. User testing

1. Introduction

A. Overview

Delock is a decentralized rental application for Android, the goal of Delock is to provide a trustless, secure and transparent platform where individuals or businesses can rent their property.

In recent years, decentralized technologies and their potential benefits have become the subject of massive hype and anticipation. The benefits of a system where users do not need to know nor trust each-other, with zero-downtime and immutable records of all transactions is undeniable, yet it is still to be seen whether blockchain technology can become mainstream and scale to meet huge demand.

Crypto-currencies like Bitcoin and Ether have garnered a lot of attention lately with people adamant that they cannot fail and vice-versa that they are destined too. The underlying technology of these coins blockchain has also been front and center of late, the amount of blockchain projects on GitHub and ICO's has risen exponentially in the last few years and most of them have failed or been abandoned.

It is important to look back at past events such as during the early years of this century and remember the emergence of social media platforms such as MySpace and Friendster. No doubt there was much speculation over whether social media would take off, and in the end these companies failed. However, their existence paved the way for platforms such as Facebook and Twitter to completely over-run the world and become super-powers in the technology industry. The same may be said of Bitcoin and Ethereum, they are the foundation on which better technology will be built.

B. Motivation

The original idea for the project stemmed from an article I read last year. It described how a company was developing a blockchain based system for the rental of autonomous cars, the idea being that autonomous car owners could instruct their vehicles to act as taxi's when they would usually be idle i.e. when the owner goes to bed at night the car would go off to ferry people around and earn the owner money.

This seemed like a very outlandish yet achievable system considering the speed of advancements in recent years, I wanted to incorporate this idea into my project and concluded that there is a countless number of items, maybe services too that people could list on such a system and earn money from.

The goal of Delock was to take the previous idea and apply it to virtually anything a user wants to rent and monetize.

C. Glossary

Blockchain – is a continuously growing list of records, bundled into blocks, the blocks are linked and secured using cryptography. Each block containing a pointer to the previous block. By design, blockchains are immutable and secure, the only exception to this is when one entity controls more than half the network, giving them the ability to re-write the chain but this is extremely unlikely.

ICO – "Initial Coin Offering" the launch of a new coin that can be used to interact with a certain system or platform. Users buy these coins to use in the system.

Decentralization – is the process of distributing or dispersing functions, powers, people or things away from a central location, entity or authority.

Decentralized-**App** (DApp) - is a piece of software consisting of a user interface (UI) and a decentralized backend; typically making use of a blockchain and smart contracts.

Ethereum – is an open source, public, blockchain-based computing platform featuring smart contract functionality.

Infura – provides secure, stable, and scalable Ethereum nodes. Access to the blockchain without local node.

Miner – A node on the network who performs hashing operations to verify new transactions and create new blocks.

Proof-of-work – An economic measure to deter denial of service attacks and other abuses such as network spam, usually requires high resources and processing time from a computer.

Genesis block – The very first block in a blockchain, the beginning of the chain.

Smart Contract – is a computer protocol intended to facilitate, verify and/or enforce the negotiations of a contract.

Solidity – scripting language for smarts contracts on the Ethereum platform.

Wallet – is a software program that stores private and public keys for an account and uses these to sign transactions for the blockchain.

Signed transaction – When a user's private keys is used to verify a transaction.

Etherscan – A searchable block explorer, API and analytics provider for Ethereum and its associated test nets.

Web3j – Lightweight java and Android library for integration with Ethereum clients, namely Infura in this case.

Truffle – development framework for testing, deploying and interacting with smart contracts on the network.

IPFS - "Interplanetary File System" is a protocol designed to create a permanent and decentralized method of storing and sharing files. It is a content-addressable, peer-to-peer hypermedia distribution protocol.

JSON - "JavaScript Object Notation" is a human readable file format for transmitting data.

NFC – "Near field Communication" for exchanging data between devices within close proximity.

2. Research

A. Blockchain

The relative immaturity of blockchain technology and the lack of standards and detailed documentation makes it a bit more challenging a technology to adopt, it is an open-source, community driven movement rather than a project undertaken by a large company with plentiful resources and time to spend on development. The likes of Java and Angular were developed by companies to fill an in-house need.

These are disadvantages in one respect but on the flip side, the fact that it is an emerging technology and the sheer amount hype surrounding it has contributed to the congregation of many interested and knowledgeable people into very active and helpful forums and groups. People are excited about the future and want to help push forward.

When undertaking this project, I needed to consider several fundamental questions about the needs of the system,

"What benefit will the use of blockchain add to this application?"

The use of a blockchain provides an immutable and secure record of all the transactions that have occurred from the creation of the genesis block. This is a valuable feature in any system to have one true, immutable and secure source of truth for the system.

"What kind of blockchain would be most suitable?"

There is more than one method of implementing a system using a blockchain, the three prominent methods are as follows,

- Un-permissioned Blockchain:

  Bitcoin, Ether, Lite-coin and countless others are un-permissioned blockchains. In this type of chain, no overarching authority exists to verify transactions. Anyone can read from and write to these blockchains.

- Public-Permissioned Blockchain:

  Are operated by known entitles such as stakeholders of a given industry. They value immutability and efficiency over anonymity and transparency. The financial industry would use this to reduce time of international payments from days to seconds.

- Private-Permissioned Blockchain:

  Operated by one entity. These value efficiency over anonymity, immutability and transparency. I wouldn't say that they are useless, but they are limited in their applications.

I decided to use a public un-permissioned blockchain i.e. Ethereum, this provides me with all the infrastructure I need to create a working version of the system and keeps the system completely open-source at the same time.

"What platform suits my needs best?"

At present, the most dominant platform with the largest community is Ethereum so it made sense to use, in recent months another platform called EOS has gained momentum.

"Will I need a dedicated token?"

In the beginning, I was under the impression that I would need to create a dedicated coin for the system. However, I learned that the reason that developers behind a project launch coins in what is called an ICO is to finance the development, the developers pre-mine some coins before the launch as payment and when the value of the coin increases they make money. This would only be necessary if I wanted to launch the project officially.

"Where will I store sensitive information?"

The only sensitive information the system handles is the passcodes for the physical locks, an encrypted messaging protocol called Whisper runs alongside the Ethereum blockchain and can be used to share these securely.

"How will it implement storage?"

The system uses IPFS to store data, this is comparable to a single peer-to-peer BitTorrent swarm exchanging versioned objects. IPFS is in active development and can be unreliable especially working from a mobile device, however, I am willing to tolerate some instability for this prototype system to decouple the it from a central database.

"Is it even possible to interact with a blockchain from a mobile device?"

It took some research to figure out how to achieve this, the prominent library for working with the blockchain from Web Apps is Web3. The creator ported it to Java in the form of Web3j, this is perfect for my application.

"How will users rent / return / lock / unlock items?"

NFC seemed like a feasible option whereby items would be secured with NFC enabled locks that a renter could unlock with their phone. For the period of their booking the lock would be openable and closable without need of NFC.

B. Smart Contracts

Smart contracts allow for the execution of code on the Ethereum Virtual Machine (EVM), the contracts themselves are immutable and secure. This is what allows for everyday objects to have the capacity to "execute code" because they have a contract on the network to represent them.

Contracts can receive and send payments autonomously, for example when a potential renter pays a deposit for an item. The contract keeps that payment and when the rental period ends it will calculate the costs and distribute the funds to the appropriate parties.

C. Platform

Anything other than a mobile device for this kind of system would be inconvenient for users, they need to be able to access the system on the go. It would be feasible to develop a web app and use passwords for the locks in theory but for sheer convenience, a mobile app seemed the best choice.

In terms of choosing between Android or IOS, initially I decided to make the app cross-platform and use React Native. Through my research I found that NFC functionality is not available to developers on the IOS platform, therefore I decided to focus on Android alone.

D. Storage

Having answered the previous questions and in keeping with the idea of the system being open-source and decentralized. I had to find a suitable solution for storage and that came in the form of IPFS, a peer-to-peer file sharing protocol.

This seemed like the perfect solution, however, after extensive research I failed to find any sure way to avail of it from a mobile device. Eventually I came across a side project someone had built on GitHub, he had implemented an IPFS daemon on the Kotlin language for Android. I decided to use his solution and credit him. The daemon can be unreliable and at times may fail to retrieve information.

[IPFSDroid project - https://github.com/ligi/IPFSDroid]

3. System Architecture

A. High-Level Design

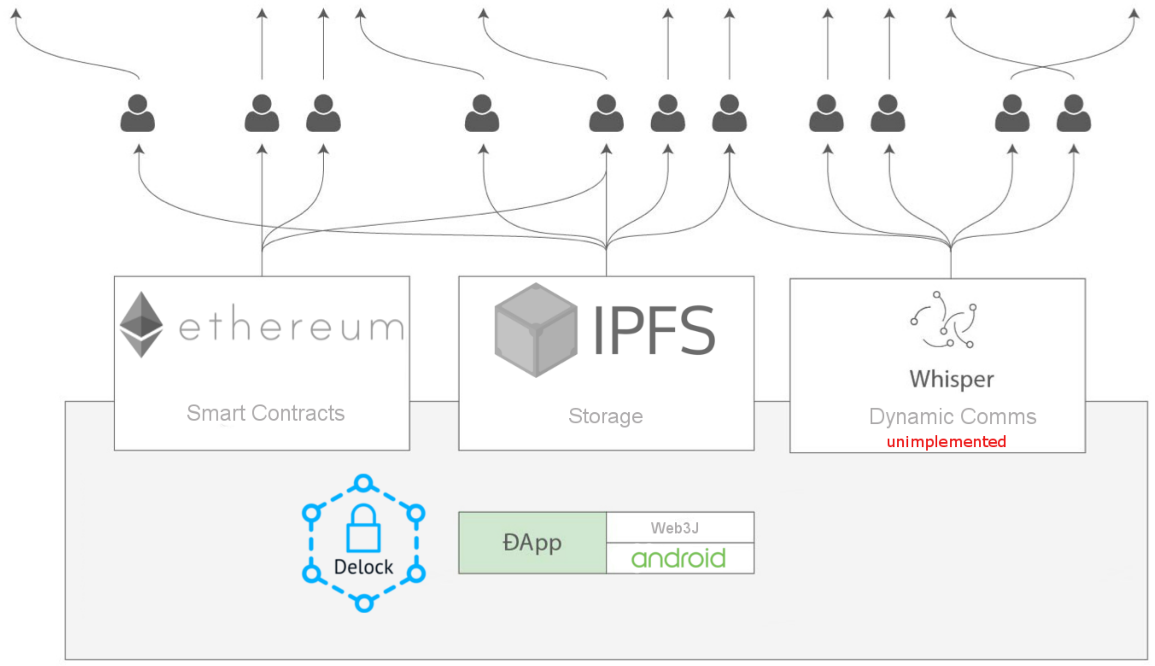


Fig. Above is an updated view of the system design as seen in the Functional Specification.

The above diagram illustrates the overall architecture of the system, the Delock application run on the Android platform and utilizes the Webj library for communication with the Ethereum network.

It would also use web3j for communication with peers via the Whisper network, however as is highlighted in the diagram, this functionality is not yet implemented.

The silhouettes of people along the top of the diagram represent peers on each network who would be both Delock users and miners who verify transactions. For IPFS they would be peers that store and serve files.

B. User Implementation

Types of user

The application has two types of user, there is no explicit division between the two as any user can rent their property as the owner and simultaneously be the renter of another listing.

Owner:

These are users who want to rent their property to others, they fill out all the appropriate details and provide images of the item and then publish the listing. Prior to this they must credit their account with enough funds to cover the cost of publishing a new listing.

Renter:

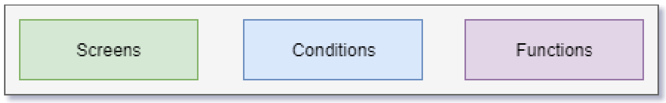
These are users who wish to rent something from another user for a fee. They can browse the listings and open a listing's details for more information regarding price and availability. They can then choose to rent the item and pay the necessary deposit amount.

Delock Admin

This is not a user per say but another entity involved in the system. Delock retains some control over the deployed contracts, to publish a contract, you need an account to own said contract. This applies to contracts that handle the organization and tracking of other contracts, the Rental Directory contract keeps track of all listings on the system and is owned by Delock.

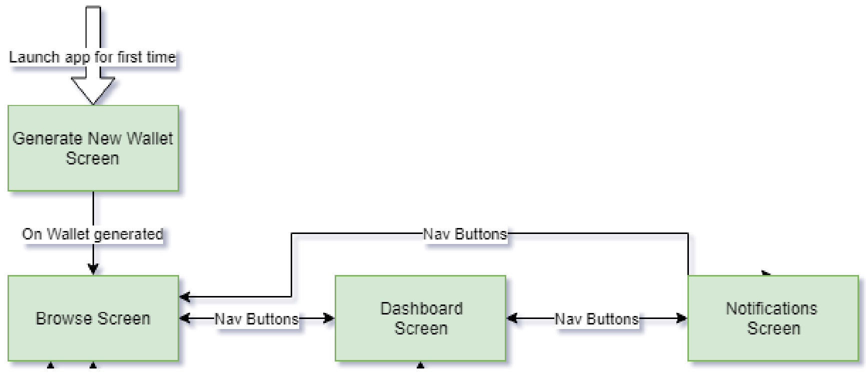
C. Android Implementation

The following diagrams illustrate the flow of the user interface under varying circumstances:



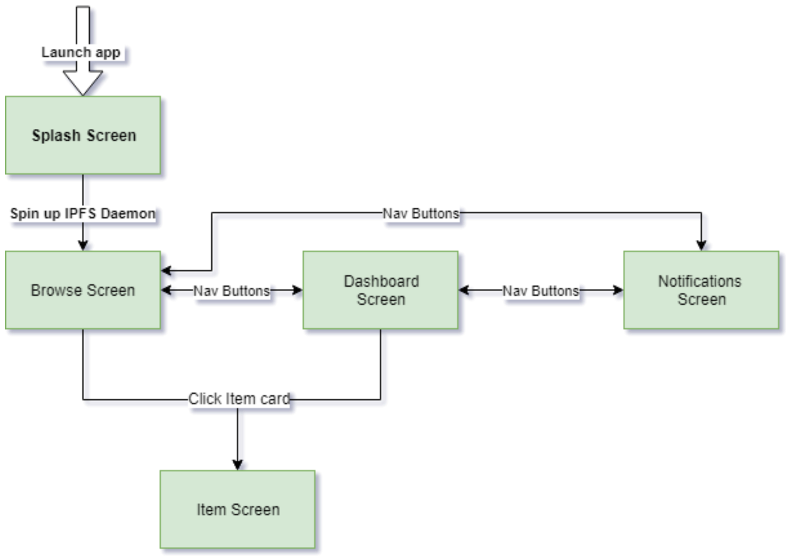
Common Functionality

First launch



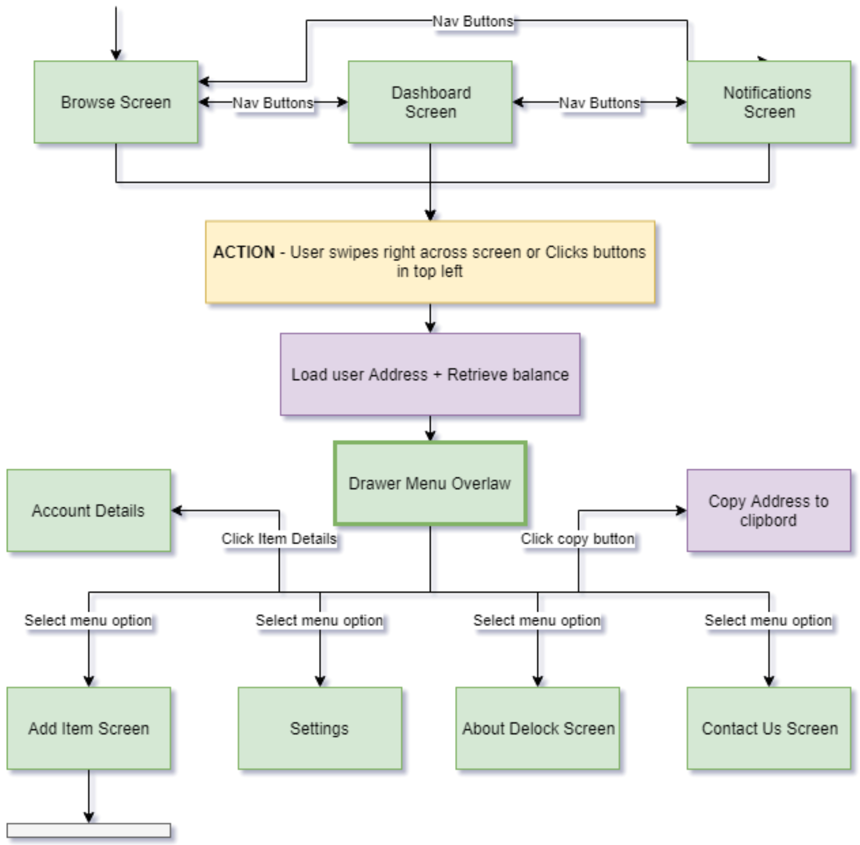
- First time launch of the application
- New wallet file and Ethereum address generated
- Setup IPFS Daemon in background on first start

Not first launch



- Wallet file and address already exist.
- IPFS daemon start-up on Splash screen.

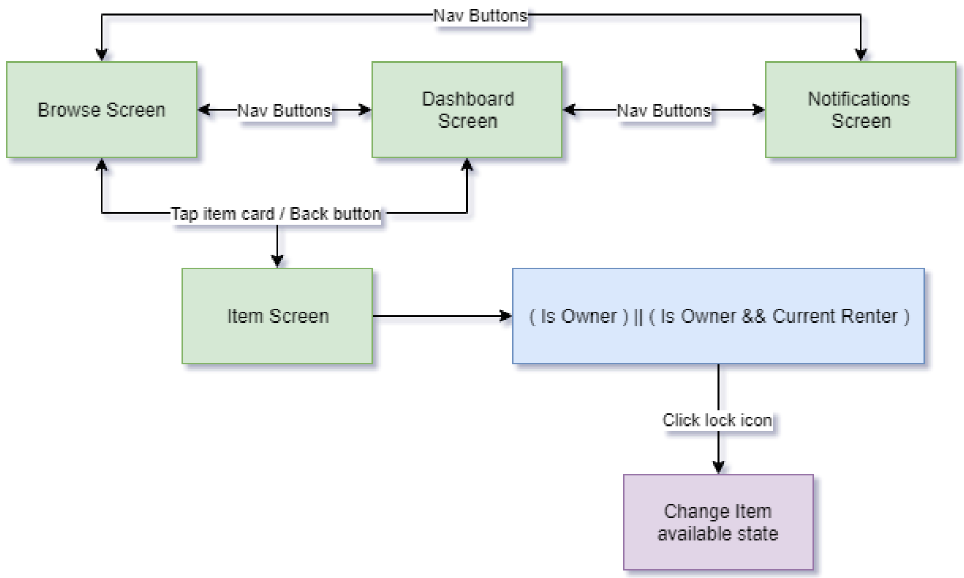
Open Drawer



- User can swipe right on the three main activities or click the menu button in the top-left to open the menu drawer.
- Here they can choose from several option as well as view their address and current balance.
- When opened, the drawer will asynchronously send a Web3j request for the user's current balance, at the same time it will retrieve the most up to date exchange rate for (EUR-ETH) and show the user their balance in euro also.

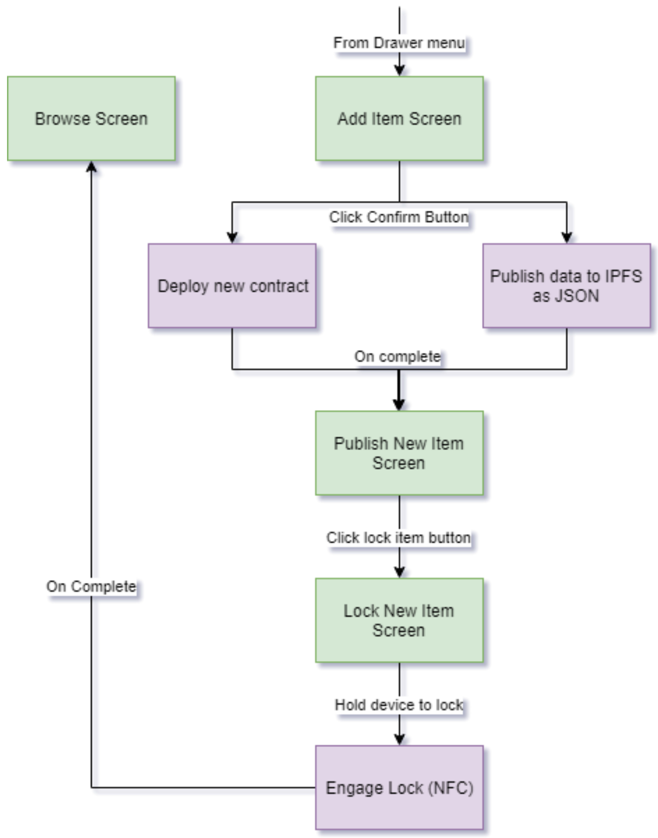
Owner Functionality

Toggle Owned Item Availability



- When the item card is opened, the system checks whether the user owns the item, who the current renter is and the availability state of the item.
- When the owner toggles the state of the item it triggers a new transaction on the blockchain to update the contract.
- When an owner sets their Item as unavailable – they are set as the current renter, this is how we know it is disabled I.e. "(isOwner && isCurrentRenter)"
- This screen behaves differently for others who do not own the item, shown in a later diagram.

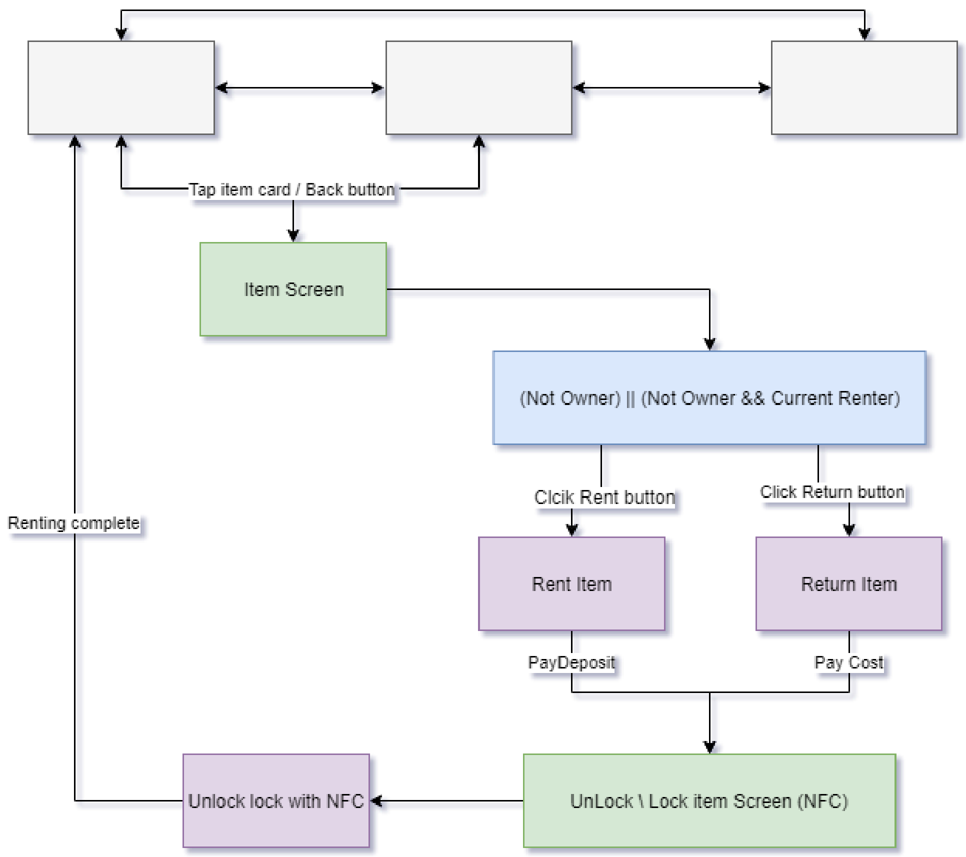
Add new Item



- A user wishing to rent their property navigates to the Add item screen.]
- Here they fill out the necessary details regarding the item they wish to rent
  - Provide images
  - Title
  - Deposit Amount
  - Cost per hour
  - Description
- An asynchronous task is triggered that will bundle the item data into JSON format and publish that file to IPFS along with the provided images.
- The resultant IPFS hashes that point to the files are returned and passed to the contract deployment function.
- The IPFS hashes are recorded in the contract and the new contract is deployed to the network.
- When the new contract has been successfully deployed and the transaction “mined” on the Ethereum network, the deployed contracts address will be returned.
- This address is then sent to the Rental Directory contract which keeps track of all rentals in the system.

Renter Functionality

Rent Item / Return Item

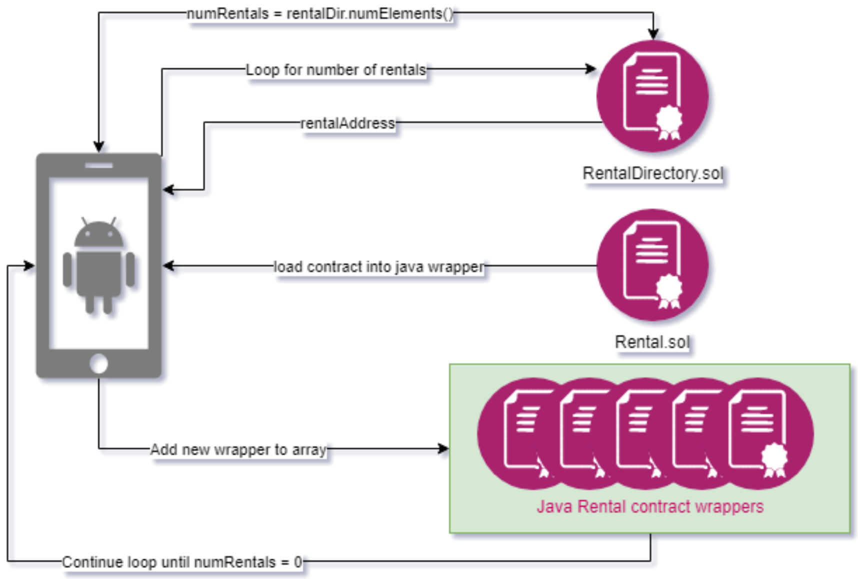


- Upon opening the item screen, the system will check whether the user is renting this item.
- Item is not being rented and is available
  - Lock Icon will be green.
  - Rent button will be visible and active, user can click and pay deposit to rent.
  - User is taken to NFC screen, when device is held to lock the lock will disengage.
- Item is being rented by another user
  - Lock icon will be red.
  - No option to rent or return item
- Item has been disabled by owner
  - Lock icon will be red.
  - No option to rent or return.
- User is the current renter of this item
  - Lock icon will be red.
  - Return button will be visible and active; user can click to end rental and pay the accumulated cost.
  - User is taken to NFC screen, when device is held to lock the lock will re-engage.

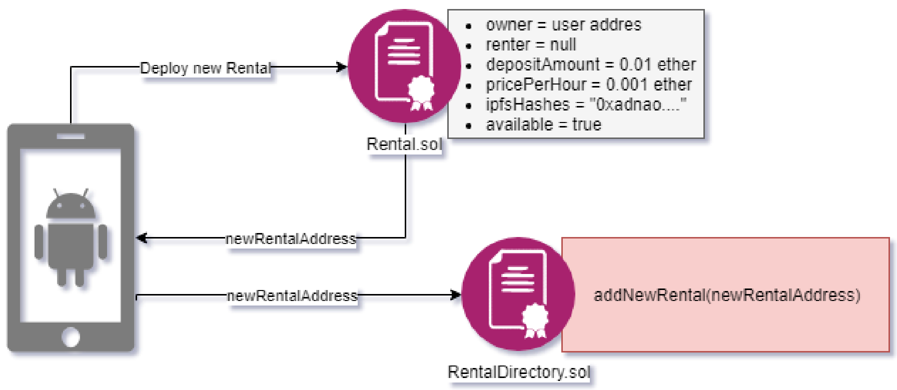
D. Smart Contract Implementation

This section will illustrate how the application communicate with and deploys smart contracts.

Retrieve Listings

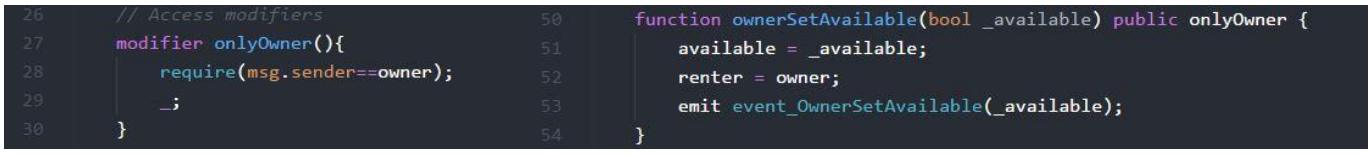

numRentals = rentalDir.numElements()
Loop for number of rentals
rentalAddress
RentalDirectory.sol
load contract into java wrapper
Rental.sol
Add new wrapper to array
Java Rental contract wrappers
Continue loop until numRentals = 0

Add new Item



(Owner) Set Availability

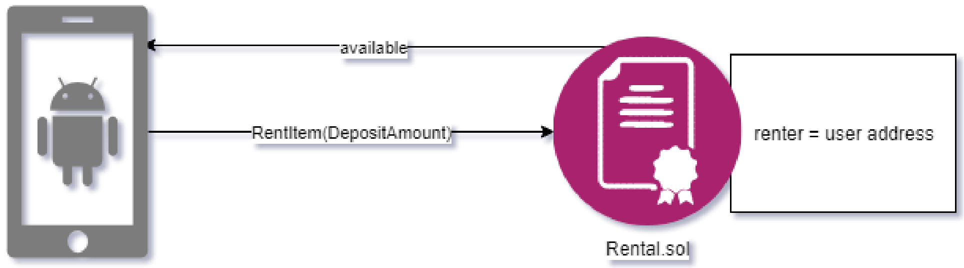
Contract is loaded into java wrapper, the user calls the 'ownerSetAvailibilty()' which has an access modifier applied to ensure only the owner of that contract can call that method. The access modifier is shown below in code:



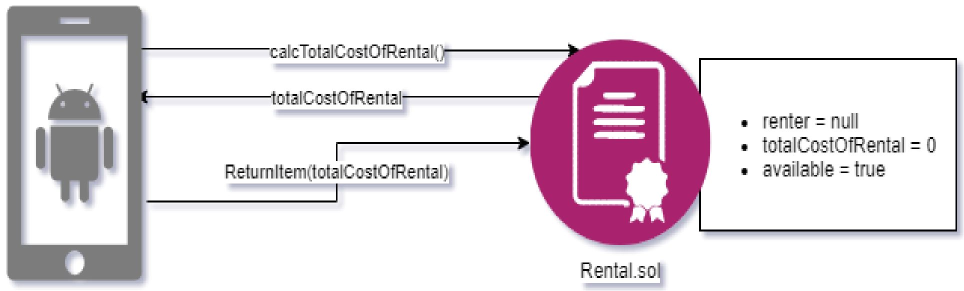
```
// Access modifiers
modifier onlyOwner(){
    require(msg.sender==owner);
    _;
}
```

```
function ownerSetAvailable(bool _available) public onlyOwner {
    available = _available;
    renter = owner;
    emit event_OwnerSetAvailable(_available);
}
```

(Renter) Rent Item



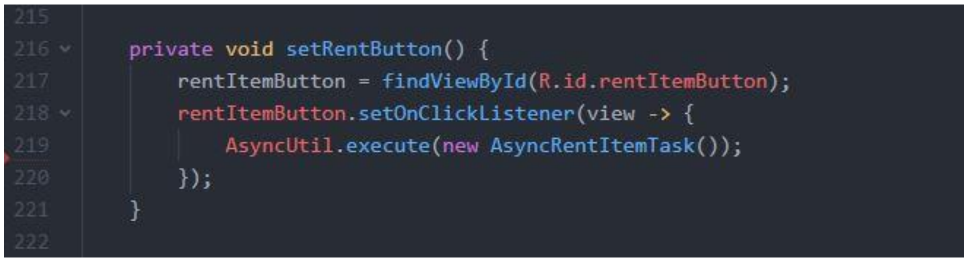
(Renter) Return Item



4. Sample Code

The following series of code snippets will provide a look at the process of renting an item. We assume that the user has navigated to an item they wish to rent and is currently on the Item screen.

ItemActivity.java



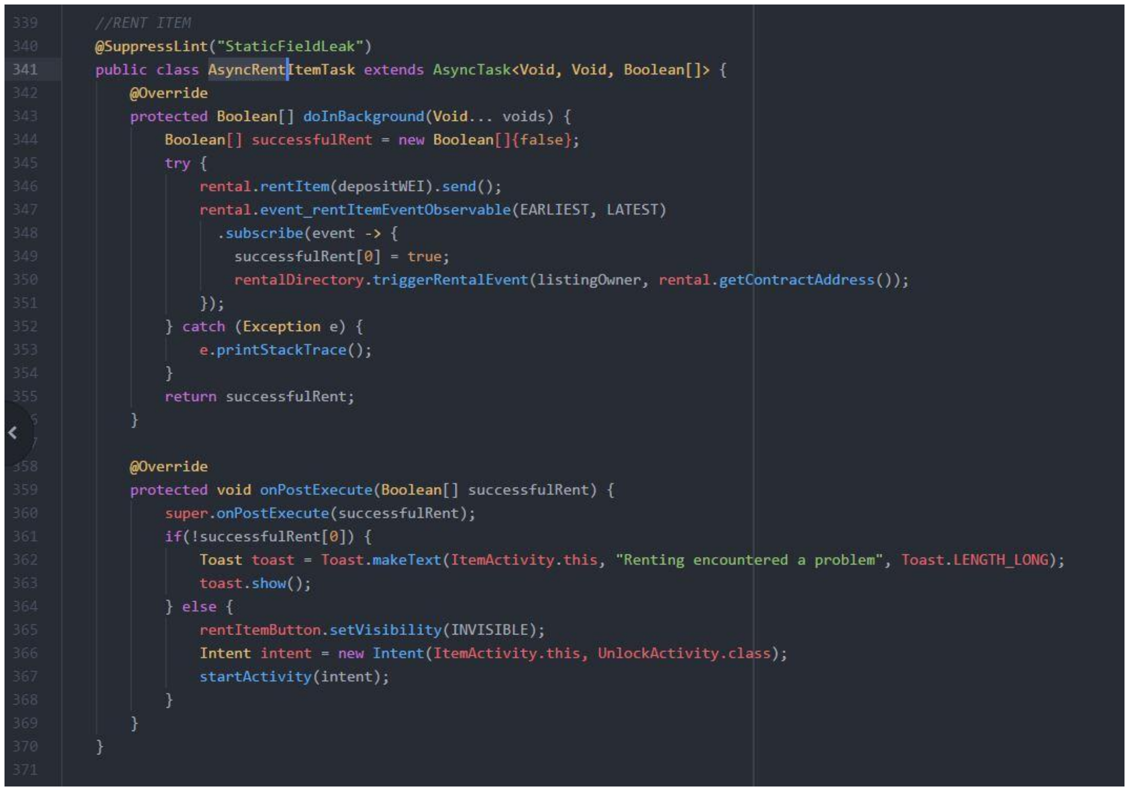
```java

    private void setRentButton() {
        rentItemButton = findViewById(R.id.rentItemButton);
        rentItemButton.setOnClickListener(view -> {
            AsyncUtil.execute(new AsyncRentItemTask());
        });
    }

```

This is a very straight forward snippet showing how an 'onClick' listener is attached the Rental Button. When the button is clicked by the user, an asynchronous task will be executed that will try to rent the item.

ItemActivity.java



```java
    //RENT ITEM
    @SuppressLint("StaticFieldLeak")
    public class AsyncRentItemTask extends AsyncTask<Void, Void, Boolean[]> {
        @Override
        protected Boolean[] doInBackground(Void... voids) {
            Boolean[] successfulRent = new Boolean[]{false};
            try {
                rental.rentItem(depositWEI).send();
                rental.event_rentItemEventObservable(EARLIEST, LATEST)
                  .subscribe(event -> {
                    successfulRent[0] = true;
                    rentalDirectory.triggerRentalEvent(listingOwner, rental.getContractAddress());
                });
            } catch (Exception e) {
                e.printStackTrace();
            }
            return successfulRent;
        }

        @Override
        protected void onPostExecute(Boolean[] successfulRent) {
            super.onPostExecute(successfulRent);
            if(!successfulRent[0]) {
                Toast toast = Toast.makeText(ItemActivity.this, "Renting encountered a problem", Toast.LENGTH_LONG);
                toast.show();
            } else {
                rentItemButton.setVisibility(INVISIBLE);
                Intent intent = new Intent(ItemActivity.this, UnlockActivity.class);
                startActivity(intent);
            }
        }
    }

```

This code snippet is somewhat more complex than the last, this is an Asynchronous task that spawns a new thread so that it can perform its functions without using the main thread which should only be used by the UI.

It consists of two parts:

DoInBackground():

This is the part of the task that will be executed on the new thread. The Rental wrappers have already been instantiated somewhere e else and values set. We only need call the 'rental.rentItem()' function and pass in the appropriate deposit amount and this will make a call to the deployed contract on the Ethereum Network.

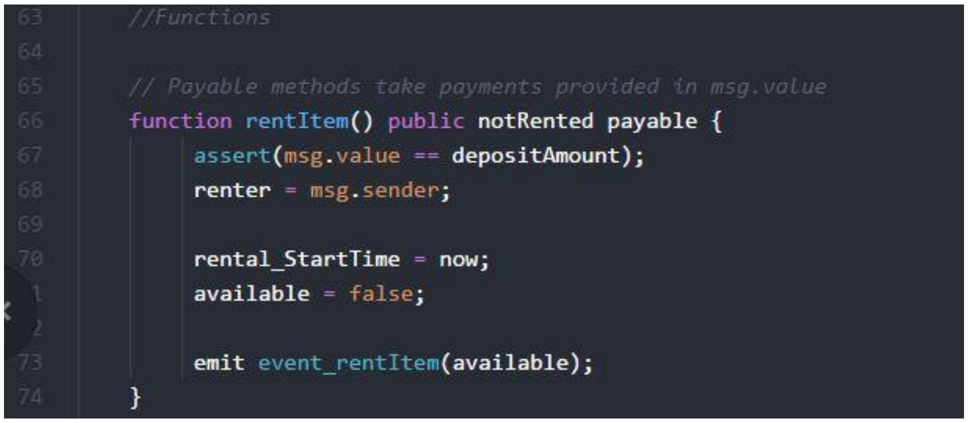
We then wait for the contract to emit an event into the EVM logs and use the information in this event to decide if the rental was executed successfully.

The call to 'rentalDirectory.triggerRentalEvent()' will be used in the future as a trigger for push notifications, this owner's devices will receive these events and alert the owner to rentals or returns pertaining to their items.

OnPostExecute():

This method is executed after the DoInbackground() method has finished, it operates on the UI thread meaning you can take results from the async-task and use them to update the UI. Here we show a Toast message to alert the user that renting was unsuccessful, otherwise we navigate to the Unlock Item activity where the user can unlock the item they just rented.

Rental.sol (Smart contract)



```
    //Functions

    // Payable methods take payments provided in msg.value
    function rentItem() public notRented payable {
        assert(msg.value == depositAmount);
        renter = msg.sender;

        rental_StartTime = now;
        available = false;

        emit event_rentItem(available);
    }
```

The code to rent an item in the smart contract is pretty intuitive to read, first of all there are two access modifiers applied to this function:

NotRented – the item must be available for rental to execute this function.

Payable – this is an explicit declaration that this method can receive and send funds.

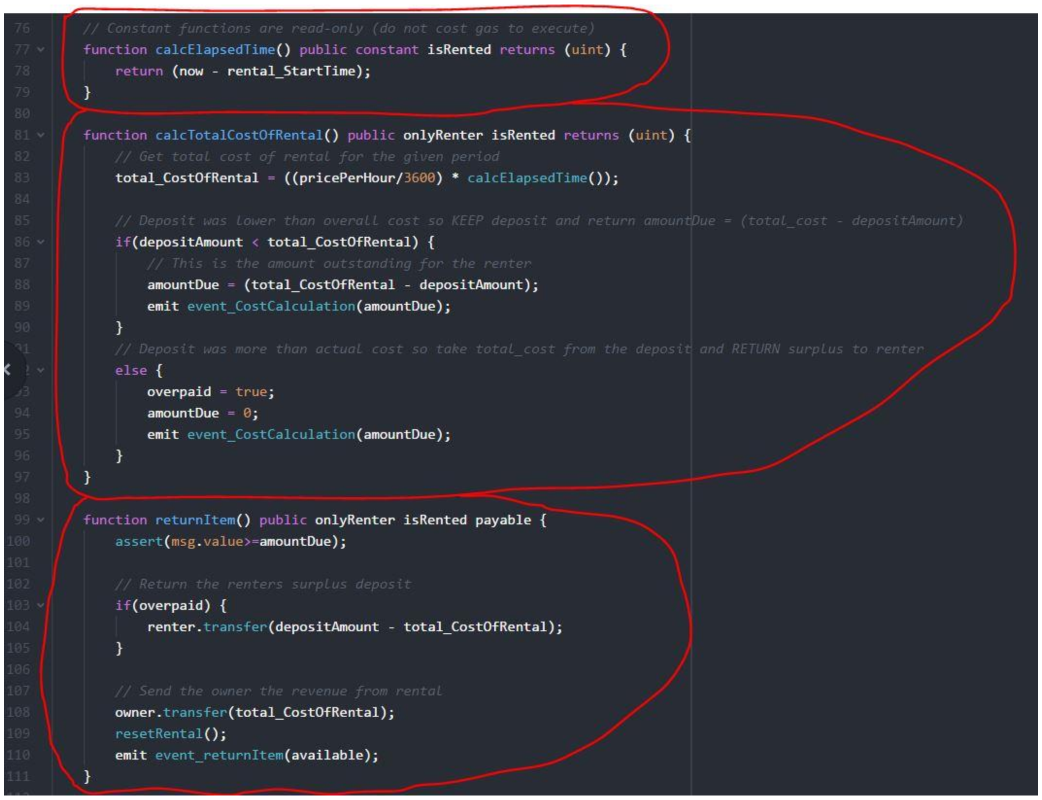
We first assert that the value sent with the function call is enough to pay the deposit.

If the value is correct then we proceed with the rental, the users address that made the call is set as the current renter of the item. The current time is recorded so we can calculate the cost of the rental period upon return of the item.

The item is now rented, so we make it unavailable to anyone else rent. We then emit an event to which the android app can listen for as feedback.

The process of returning an item is similar to what was shown in the previous code snippets, however, to return the item we need to calculate the total cost of the rental, this is shown below:

Rental.sol



```
// Constant functions are read-only (do not cost gas to execute)
function calcElapsedTime() public constant isRented returns (uint) {
    return (now - rental_StartTime);
}

function calcTotalCostOfRental() public onlyRenter isRented returns (uint) {
    // Get total cost of rental for the given period
    total_CostOfRental = ((pricePerHour/3600) * calcElapsedTime());

    // Deposit was lower than overall cost so KEEP deposit and return amountDue = (total_cost - depositAmount)
    if(depositAmount < total_CostOfRental) {
        // This is the amount outstanding for the renter
        amountDue = (total_CostOfRental - depositAmount);
        emit event_CostCalculation(amountDue);
    }
    // Deposit was more than actual cost so take total_cost from the deposit and RETURN surplus to renter
    else {
        overpaid = true;
        amountDue = 0;
        emit event_CostCalculation(amountDue);
    }
}

function returnItem() public onlyRenter isRented payable {
    assert(msg.value>=amountDue);

    // Return the renters surplus deposit
    if(overpaid) {
        renter.transfer(depositAmount - total_CostOfRental);
    }

    // Send the owner the revenue from rental
    owner.transfer(total_CostOfRental);
    resetRental();
    emit event_returnItem(available);
}
```

Fig. Above we can see the three functions involved in returning an item.

CalcTotalCostOfRental is called from the android app so it will know what value to send when it calls the returnItem function.

CalcTotalCostOfRental begins by breaking the hourly price of the rental down into the price per second, it multiplies this value by the number of seconds that have passed since the rental began I.e. calcElapsedTime().

If the renter has returned the item and the total cost exceeds what the deposit was, (we will keep the deposit) and set amount due to whatever is left when we subtract the deposit rom the total.

Otherwise, the cost of the rental has not exceeded the deposit price, therefore the renter has overpaid, and we must return a portion of their deposit (We take payment from the deposit and return the remainder).

In the returnItem function, we check to see whether the renter has overpaid and if so we transfer the appropriate funds back into their account. Whatever is left after this is revenue for the owner and we then transfer this to them.

Finally, we reset the Item for another rental and emit an event that the app can listen for.

5. Problems and Resolutions

There were many problems when designing and implementing this project, I will focus on some of the worst ones here:

Problem with IPFS timeout – Unresolved

From time to time, the IPFS node will timeout when trying to retrieve listings. I have not been able to get to the bottom of this issue and I believe it is beyond my control for now. I didn’t find a reliable fix but by using regular Http requests to get the files it seems to work more often than when using the IPFS Daemon itself. You would think that this would point to the daemon as the culprit, yet the problem still persists with Http, just to a lesser degree.

It seems to be a problem either with the daemon I am using or with IPFS itself. The content seems to get lost or corrupted, sometimes I can access the files through the standard Chrome browser using an IFPS plugin and sometimes not. This issue was by far the most time-consuming and troublesome to get around and led to quite significant setbacks.

Android Studio Build Time -

Not so much a problem with the project itself but was a major pain for development, the Gradle build framework is old and slow. Especially as the project grew and my laptop came under pressure, build times increased to a point where it was becoming a problem. Having had minor experience with React Native which is almost instantaneous (for small project anyway), the difference is stark in terms of responsiveness and development time.

Authentication -

In terms of authentication for the app, the goal is to keep the application decentralized and open-source. We need to verify a user and link a person an address, anonymity is considered a major benefit of blockchain, however, for this use case we need to identify users, else anyone can create throw away account, pay a deposit and not return an item.

I believe this can be solved by developing the application using a public permissioned blockchain, this way we know who is on the system and the transparency of the system is maintained. The overhead of setting this up was too much for this project but in the future, this would be a viable option.

Slow Ethereum Transactions -

Something I did not foresee when planning the project was the slow speed of transactions on the network. It's can take anywhere between 10 seconds to 4 minutes for a transaction to complete. Most of this is down to the gas price supplied with the transaction (how much are you willing to pay to have this executed?).

To combat this problem and because the application runs on a test net where I have access to an unlimited amount of fake ether, I have set a constant gas price that is used throughout the application. This gas price is extremely high and is not representative of real-world prices. In reality we would use an API such as 'EtherGasStation' to choose a "best-value" gas price for each transaction.

Decentralization -

Something that occurred to me recently was the fact that the application is meant to be decentralized, not tied to any one entity or individual. I realized a major design flaw, who owns and manufactures the locks that users lock their items with. In real-life I think this would be different, Delock would be an official project with its own coin and would manufacture the locks.

Another option would be to use QR codes that people can stick to their items and then have them provide their own means of locking items, for example, when a renter scans the QR code they would be provided with a combination for the owner lock.

The same problem crops up again when we consider the Google maps API that the app currently uses, however, this could easily be solved by emerging technologies that aim to decentralize the area of mapping systems.

6. Testing and Feedback

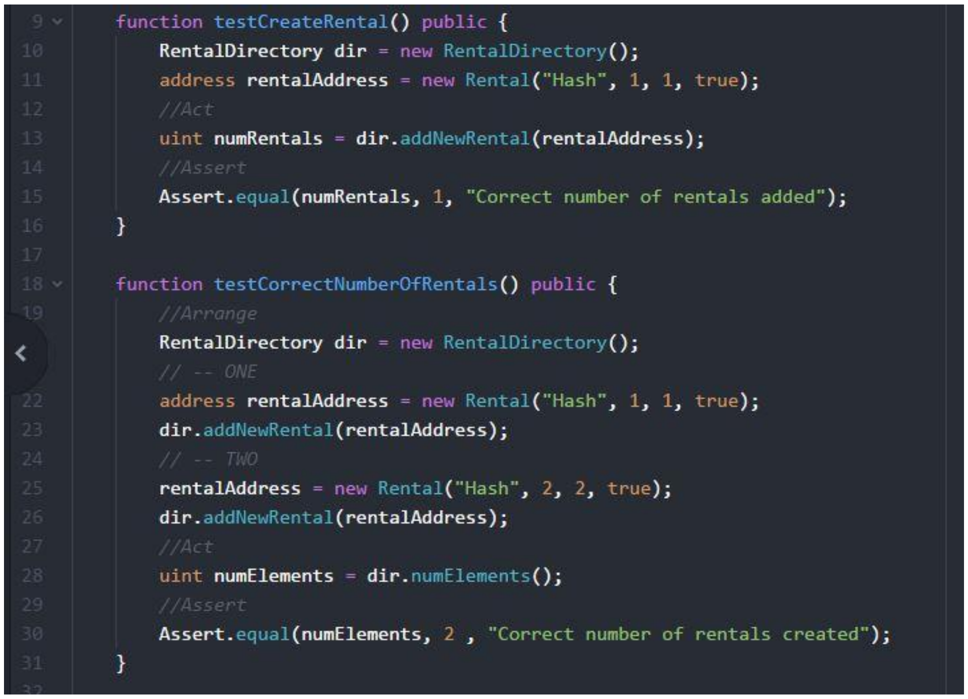
Unit Testing

Unit testing involves testing code and functions on a granular level, for example, unit tests should focus on one function and try to prove that this individual function is either working correctly or not.

Truffle + TestRPC

To develop the smart contract, I used the truffle framework along with testrpc. This allowed me to run frequent tests after every change or new feature introduced. TestRPC creates a dummy network on your local machine with several test accounts, each with a certain amount of ether.

Truffle let you compile, test and deploy contracts. Below we can see an example of a unit test.



```
    function testCreateRental() public {
        RentalDirectory dir = new RentalDirectory();
        address rentalAddress = new Rental("Hash", 1, 1, true);
        //Act
        uint numRentals = dir.addNewRental(rentalAddress);
        //Assert
        Assert.equal(numRentals, 1, "Correct number of rentals added");
    }

    function testCorrectNumberOfRentals() public {
        //Arrange
        RentalDirectory dir = new RentalDirectory();
        // -- ONE
        address rentalAddress = new Rental("Hash", 1, 1, true);
        dir.addNewRental(rentalAddress);
        // -- TWO
        rentalAddress = new Rental("Hash", 2, 2, true);
        dir.addNewRental(rentalAddress);
        //Act
        uint numElements = dir.numElements();
        //Assert
        Assert.equal(numElements, 2 , "Correct number of rentals created");
    }
```

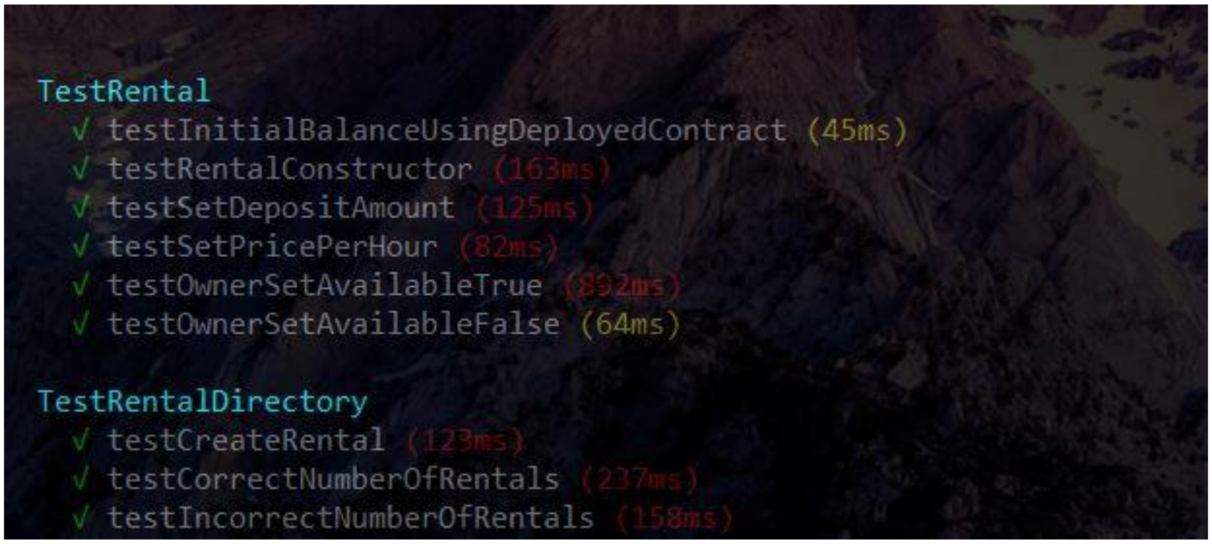
Above we can see examples of unit tests for the Rental Directory, notice that the tests do not resuse any values or objects, they are independent of one another.

They first "Arrange" what they need to be able to run the test.

They then "Act" or run the function they mean to test.

And finally, "Assert" that the value is as expected.

It would be preferable to use random number generator for any values being fed in, this would help make the tests more robust, unfortunately there are no straightforward random number generator functions for solidity yet.

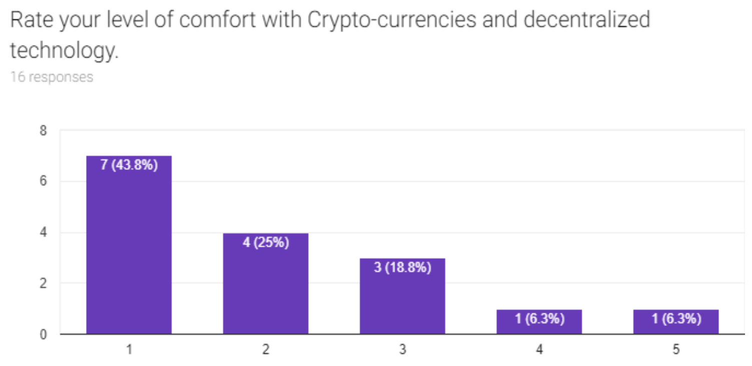


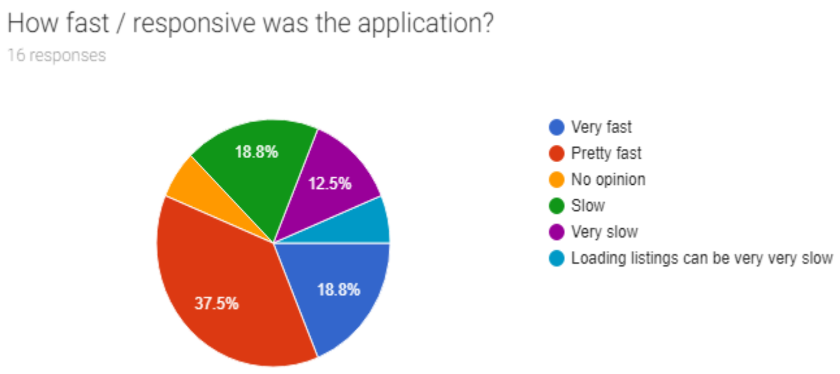
```
TestRental
  √ testInitialBalanceUsingDeployedContract (45ms)
  √ testRentalConstructor (163ms)
  √ testSetDepositAmount (125ms)
  √ testSetPricePerHour (82ms)
  √ testOwnerSetAvailableTrue (892ms)
  √ testOwnerSetAvailableFalse (64ms)

TestRentalDirectory
  √ testCreateRental (123ms)
  √ testCorrectNumberOfRentals (237ms)
  √ testIncorrectNumberOfRentals (158ms)
```

User Testing

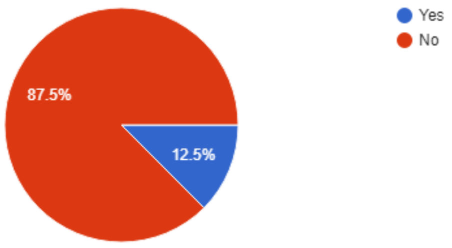
Usability





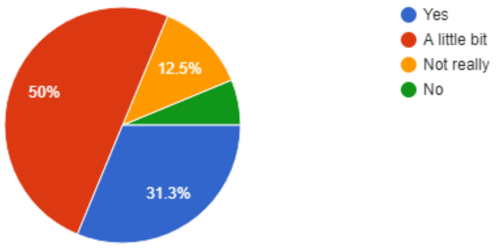
Did the app crash / hang / freeze?

16 responses



Did you feel you needed to understand crypto-currencies and blockchain technology to use the app?

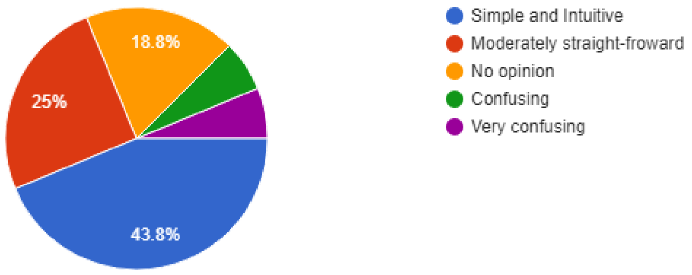
16 responses



User Interface

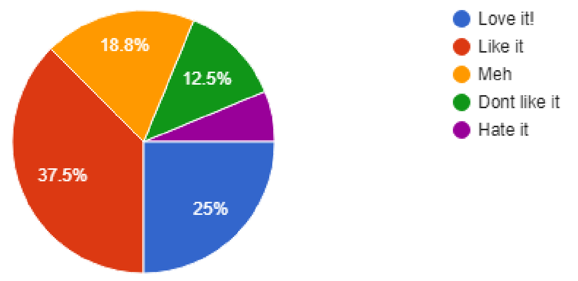
How easy was it to understand and navigate the application?

16 responses



What did you think of the colour scheme?

16 responses



Are there any changes you would like to see?

Nope (2)

None

Would like the ability to book items in advance

Push notifications for when my stuff gets rented or returned would be useful

Should have bookings

Bookings

You should be able to import an existing Ethereum address instead of having the app generate a new wallet