

## Blockchain Project Phase 3

Title of the project: **Trading in Fractionalized Realty**

Project Members:

Team member 1: **Diksha Saxena**, [dikshasa@buffalo.edu](mailto:dikshasa@buffalo.edu), 50478011

Team member 2: **Shubhankar Kumar**, [skumar45@buffalo.edu](mailto:skumar45@buffalo.edu), 50476119

*(P.S -> Phase 3 details from page 9.)*

### **Issue(s) addressed:**

Peer to peer decentralized, disintermediate, distributed Trading: Trading has always been regulated by a centralized authority keeping a record of every transaction. Through our platform we are hoping to decentralize this and make it possible for end users to directly interact with one another. This would enable us to add the trust layer between the end users directly. The traders have full control on the amount and fraction of real estate asset that they would like to buy and when they would like to exit the trading investment. This would be securely built on the Ethereum protocol and would ensure transparency and traceability amongst the community. Because there is no central authority, this would incur reduced costs for the traders and the property owners eliminating the middlemen and using the distributed ledger technology. It is immutable in nature and would ensure no fraud by a single entity or individual.

The ability to invest in real estate with small chunks of money. Whenever someone wants to buy or sell a piece of property, it has traditionally been done in whole quantities, with discrete values. Real estate is a wonderful form of long-term investment which is bound to yield good returns for the investors. **However, because it needs to be bought in whole, most of the lower middle-class population can't invest in it. Through this model, we would be able to make it available for a larger chunk each owning smaller stakes.** The owner will list his desired fraction of the property that he wants to enlist and traders who want to invest in it by buying and selling fractions of it, will have to buy our crypto token and use that to trade the property.

**Abstract:** a 100-word description of the problem: what, why, when, who, where and how etc. Think of a problem that cannot be solved using traditional means.

Investment in real estate which may yield long-term profitable returns has so far been available to only those who can afford to buy the whole piece of property and hasn't been open to a huge chunk of the lower pyramid population. By enabling trusted peer-to-peer trading of fractionalized real estate, we are opening the market to a larger base. This model focuses on the inclusion of the lower income groups. The users will connect their Ethereum wallet to our app and once an owner lists his desired fraction of the property on the app, it is bid by the buyers, and upon successful buyout which verifies its trade status, it is then open for trade. Since the trade could be quantified in continuous real numbers, it is open for exchange for those who want to trade smaller quantities of it, even down to a dollar in worth. For example, if one wanted to buy a plot in Downtown, Buffalo which was 10000 sqft big, traditionally, they would have to buy the whole plot to invest in it which would have discouraged a lot of people from venturing into it, but this app would take care of that.

→ **What is your application? What can you do?**

- Our application aims at enabling masses from lower economy section to buy properties in chunks which they won't be able to afford as a whole. The owner of the property lists the property which required details which they think will be good enough to attract users. They also state the number of fractions in which they want to fractionalize their property. We do the math's and calculate the rate of each fraction by dividing the total price with the number of fractions.
- Now after the property is listed by the owner, it is not available to trade yet. First it will go in a bidding phase, where users will bid on it based in the listing details and if the total number of bid reached 5, then only it is available for the users to trade. This step is just a verification step which will eliminate wrongly listed properties from our system.
- After the property is available to trade, the users can buy or sell. If they are trying to buy then first it will check whether any other user has placed a request to sell it. If request is matched the property will be bought, if there is no selling request, and there is still at least a fraction available, the property will be bought from the owner directly. If both the case doesn't match, a new request will be raised, which will be fulfilled if someone tries to sell it. Similarly for the selling side, the buying request will be checked, if not available then a new selling request will be created. Once bought, you can't sell it back to the owner.

→ **Instructions to deploy, test, and interact.**

- First you need to open your ganache then go inside the **contract** folder and run **truffle compile** and **truffle migrate** which will deploy the smart contract.
- Open your Meta mask and log in based on the Ganache credentials.
- To run the application, open **app** folder and **run node app.js**
- Your application will be up.

→ **Your token symbol and reasoning**

- Our token symbol will be PROP -> since it is related to Property Trade and Exchange. To buy and sell property you need to have PROP token in your account.

→ **Pick 2-3 smart contract functions and explain how they work in detail. Include Screenshots where applicable.**

- **Vote Function:** - This function is used in the bidding phase, where we get the propertyID in the argument and check which if the vote count is already 5 and if the particular user has already voted for that property. If all the checks are successful, it adds the user for that property and increase the vote count by 1.

```

function vote(string memory propertyId) public returns (bool) {
    voteDetails storage vd = propertyVoteDetails[propertyId];
    require(vd.voteCount < 5, "Bidding is Over");
    mapping (address => bool) storage userIds = vd.userIds;
    require(!userIds[msg.sender], "User has already voted");
    vd.voteCount++;
    vd.userIds[msg.sender] = true;
    propertyVoteDetails[propertyId] = vd;
    return vd.voteCount >= 5;
}

```

→ Returns the total number of votes for that property.

```

function getVoteCount(string memory propertyId) public view returns (uint) {
    voteDetails storage vd = propertyVoteDetails[propertyId];
    return vd.voteCount;
}

```

→ Transfer the ether from buyer to the seller

```

function sendMoney(address buyer) public payable {
    address payable to = address(uint160(address(buyer)));
    address(to).transfer(msg.value);
}

```

→ Buy Property: Here we check if there is any pending selling request, if yes then the fund is transferred from buyer to seller

```

function buy(address useraddress, string memory propertyId) payable public {
    request storage buyingRequest = buyingRequests[propertyId];
    require(!buyingRequest.users[useraddress], "Buying Request already exists");
    request memory sellingRequest = sellingRequests[propertyId];
    if(sellingRequest.exists){
        address payable seller = sellingRequest.usersaddress[0];
        trade.sendMoney.value(msg.value)(seller);
    }
    else{
        buyingRequest.exists = true;
        buyingRequest.users[useraddress] = true;
        address payable buyer = address(uint160(address(useraddress)));
        buyingRequest.usersaddress.push(buyer);
    }
}
}

```

→ Sell Property: Here we check if there is any pending buying request, if yes then the fund is transferred from buyer to seller

```

function sell(address useraddress, string memory propertyId) payable public {
    request storage sellingRequest = sellingRequests[propertyId];
    require(!sellingRequest.users[useraddress], "Selling Request already exists");
    request memory buyingRequest = buyingRequests[propertyId];
    if(buyingRequest.exists){
        address payable buyer = buyingRequest.usersaddress[0];
        trade.sendMoney.value(msg.value)(useraddress);
    }
    else{
        sellingRequest.exists = true;
        sellingRequest.users[useraddress] = true;
        address payable seller = address(uint160(address(useraddress)));
        sellingRequest.usersaddress.push(seller);
    }
}

```

**References: List all the resources used and reading material to support your project idea.**

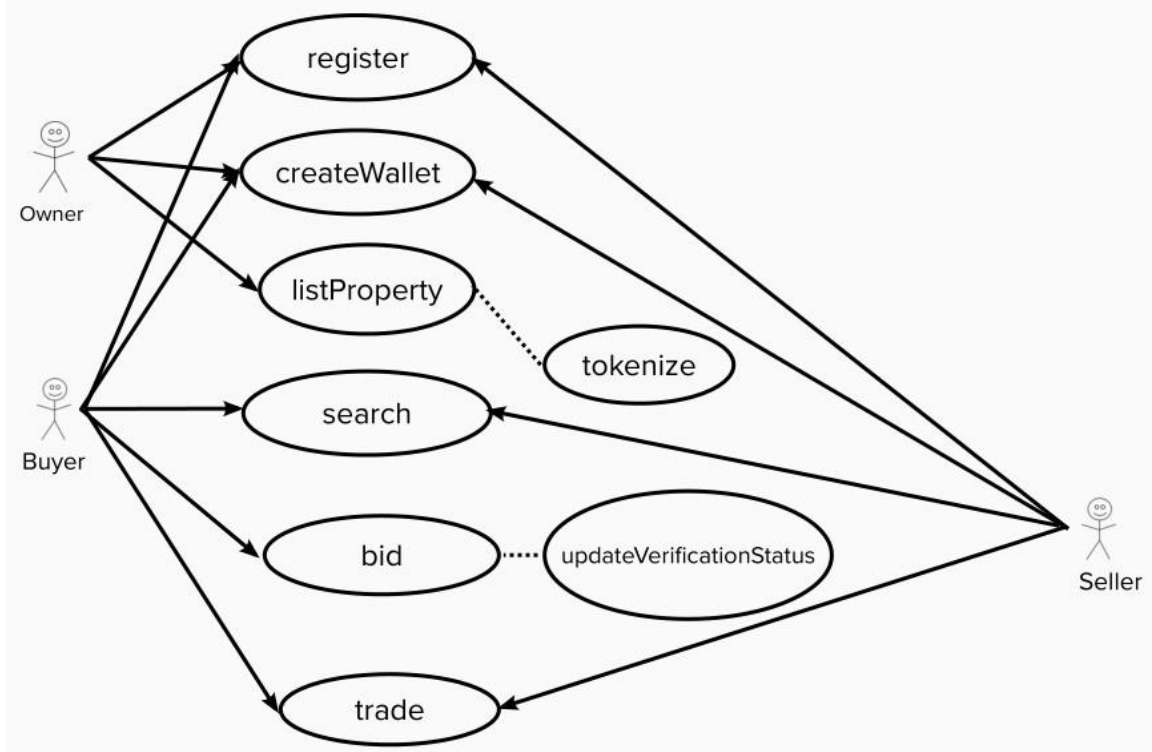
<https://ethereum.stackexchange.com/>  
<https://livebook.manning.com/book/blockchain-in-action/>  
<https://app.mural.co/t/universityatbuffalo9946/m/universityatbuffalo9946/1664048348685/8cc1c6c014d0e8f5ca34069c33d1ae28cea2e101?sender=u0e66c96e1cfac133aaae7208>  
<https://www.parcl.co/blog/digital-real-estate-investing-what-is-tokenized-ownership>  
<https://www.dappuniversity.com/articles/the-ultimate-ethereum-dapp-tutorial>  
<https://reali.com/resources/how-does-trading-a-house-work/>  
<https://bernardmarr.com/the-5-big-problems-with-blockchain-everyone-should-beaware-of/>  
<https://learn.arrivedhomes.com/fractional-real-estate-investing/>  
<https://www.benzinga.com/real-estate/22/07/28017011/a-beginners-guide-tofractionalized-commercial-real-estate-investing>  
<https://medium.com/coinmonks/real-estate-tokenization-vs-fractionalization-whichone-is-best-for-me-300131df9633>

## → Design diagrams

### 1. Quad Chart diagram

<p><b>Use Case:</b> Enable masses from lower economic background to decide which property should be listed and buy those properties in fractions and participate in their trading.</p>	<p><b>Existing Issues:</b></p> <ol style="list-style-type: none"> <li>1. Buying a property in whole is not possible for everyone.</li> <li>2. Selling the entire property may not be ideal for the owner.</li> <li>3. Centralized Trading platform with not much power to the users and traders.</li> </ol>
<p><b>Proposed Solution:</b></p> <ol style="list-style-type: none"> <li>1. Allow the owners to fractionalize the property and sell whatever part of the property they want to different persons and not in entirety to one person.</li> <li>2. The user have the power to verify which property they want to be listed and which they don't.</li> <li>3. Users buy the property whatever fraction of it they can afford.</li> </ol>	<p><b>Benefits:</b></p> <ol style="list-style-type: none"> <li>1. Listing of the property is unbiased. All the voting details is present on Blockchain.</li> <li>2. Transfer of funds doesn't happen through any intermediary and all the transactions are again recorded on blockchain.</li> <li>3. The fractionalized property can be stored as NFTs</li> </ol>

### 2. Use Case Diagram



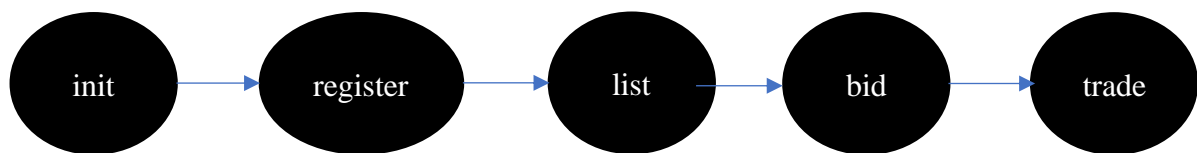
### 3. Contract Diagram

Trade
sendMoney()

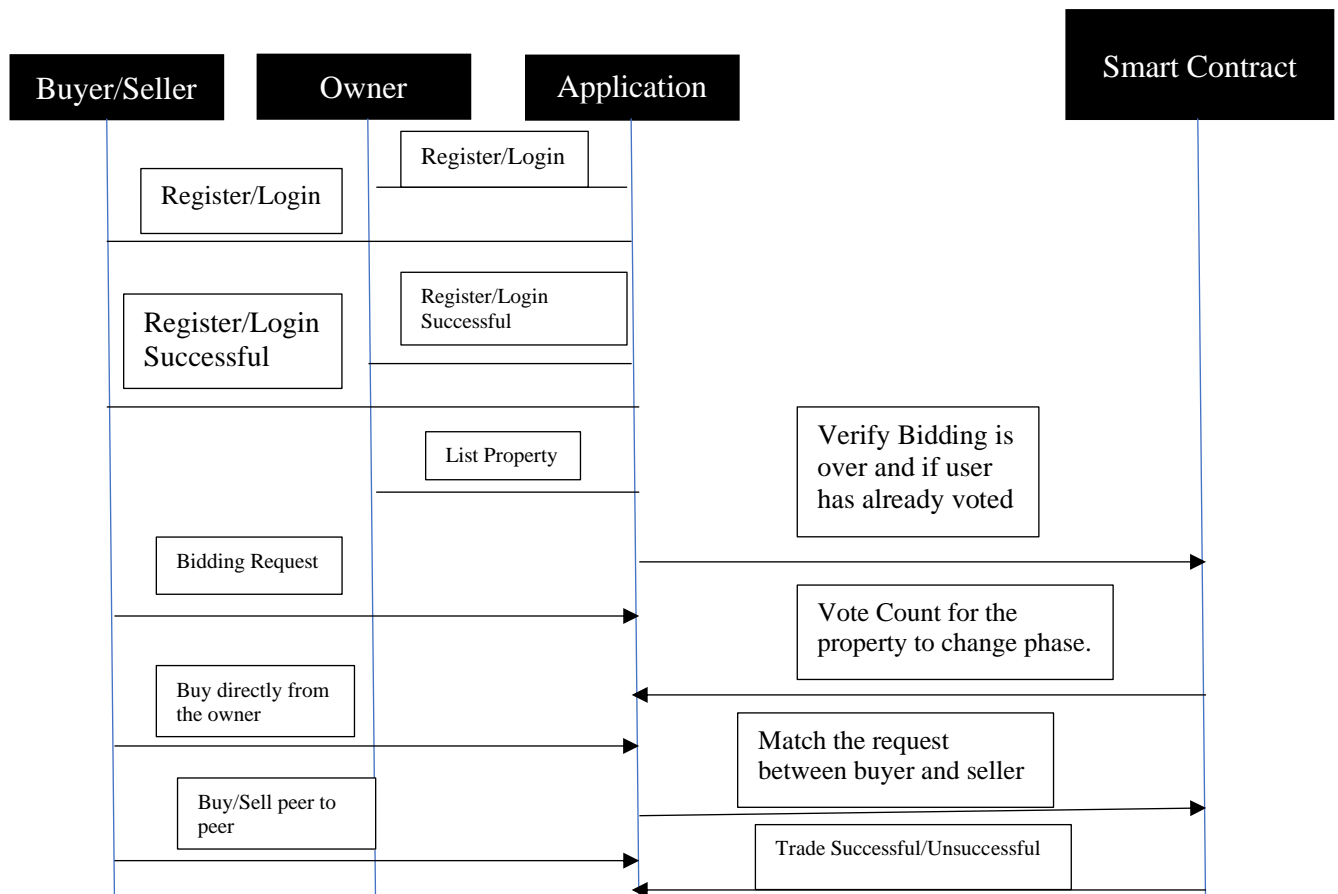
Bid
struct voteDetails mapping propertyVoteDetails
voteCount <5 !userIds[msg.sender]
vote() getVoteCount()

Trading
struct request mapping buyingRequests mapping sellingRequests
!buyingRequest.users[useraddress] !sellingRequest.users[useraddress]
buy() sell()

### 4. FSM Diagram



### 5. Sequence Diagram



→ Solidity Code:

1. Trade.sol

```
contract Trade{  
    function sendMoney(address seller, string memory propertyId) public payable{  
        address payable to = address(uint160(address(seller)));  
        address(to).transfer(msg.value);  
    }  
}
```

1. sendMoney function transfer fund from buyer to the seller.

2. Bid.sol

```
contract Bid{  
    struct voteDetails {  
        mapping (address => bool) userIds;  
        uint voteCount;  
    }  
    mapping(string=>voteDetails) public propertyVoteDetails;  
  
    function vote(string memory propertyId) public returns(bool){  
        voteDetails storage vd = propertyVoteDetails[propertyId];  
        require(vd.voteCount <5, "Bidding is Over");  
        mapping (address => bool) storage userIds = vd.userIds;  
        require(!userIds[msg.sender], "User has already voted");  
        vd.voteCount++;  
        vd.userIds[msg.sender] = true;  
        propertyVoteDetails[propertyId] = vd;  
        return vd.voteCount>=5;  
    }  
  
    function getVoteCount(string memory propertyId) public view returns (uint){  
        voteDetails storage vd = propertyVoteDetails[propertyId];  
        return vd.voteCount;  
    }  
}
```

1. struct voteDetails contains detail about the voting for a property.
2. propertyVoteDetails contains mapping of a particular property with its voting details

3. Function vote check if the voting is eligible or not. Then it increases the vote count by 1 and add the voting details in that particular property mapping
4. Function getVoteCount returns the number of vote for that property

### 3. Trading.sol

```
contract Trading{
    Trade trade;
    struct request{
        mapping(address=>bool) users;
        address payable[] usersaddress;
        bool exists;
    }
    mapping(string=>request) buyingRequests;
    mapping(string=>request) sellingRequests;

    function buy(address useraddress, string memory propertyId) payable public {
        request storage buyingRequest = buyingRequests[propertyId];
        require(!buyingRequest.users[useraddress], "Buying Request already exists");
        request memory sellingRequest = sellingRequests[propertyId];
        if(sellingRequest.exists){
            address payable seller = sellingRequest.usersaddress[0];
            trade.sendMoney.value(msg.value)(seller);
        }
        else{
            buyingRequest.exists = true;
            buyingRequest.users[useraddress] = true;
            address payable buyer = address(uint160(address(useraddress)));
            buyingRequest.usersaddress.push(buyer);
        }
    }

    function sell(address useraddress, string memory propertyId) payable public {
        request storage sellingRequest = sellingRequests[propertyId];
```



```

require(!sellingRequest.users[useraddress], "Selling Request already exists");
request memory buyingRequest = buyingRequests[propertyId];
if(buyingRequest.exists){
    address payable buyer = buyingRequest.usersaddress[0];
    trade.sendMoney.value(msg.value)(useraddress);
}
else{
    sellingRequest.exists = true;
    sellingRequest.users[useraddress] = true;
    address payable seller = address(uint160(address(useraddress)));
    sellingRequest.usersaddress.push(seller);
}
}
}

```

1. buyingRequest and sellingRequest contains details about the request raised by a user for a particular property.
2. Buy function takes care of the buying part, where it checks if the buying request for that property already exists by the user, if yes then return, otherwise it finds the corresponding selling request and match it, or register a new buying request.
3. Sell function takes care of the selling part, where it checks if the selling request for that property already exists by the seller, if yes then return, otherwise it finds the corresponding buying request and match it, or register a new selling request.

## → ERC721 Token Details

We have created an ERC721 Token/NFT to correctly verify the owner of the property.

This is how our NFT looks



We have used openzeppelin for base code and created 2 additional functions and mappings for our special use case. This is how our contract looks like.

```
pragma solidity >=0.7.21 <0.9.0;
```

```
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
```

```
import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
```

```
import "@openzeppelin/contracts/access/Ownable.sol";
```

```
import "@openzeppelin/contracts/utils/Counters.sol";
```

```
contract MyPROP is ERC721, ERC721URIStorage, Ownable {
```

```
    using Counters for Counters.Counter;
```

```
    Counters.Counter private _tokenIdCounter;
```

```
    constructor() ERC721("MyPROP", "PRP") {}
```

```
    string uri = "https://gateway.pinata.cloud/ipfs/QmRDnkaHEEV4wk2vNnyK8cHf7iD6tTzH9BfH5x5Ca9sCQC";
```

```
    mapping(uint=>string) tokenURIs;
```

```
    function safeMint(address to) public payable {
```

```
        uint256 tokenId = _tokenIdCounter.current();
```

```
        _tokenIdCounter.increment();
```

```
        _safeMint(to, tokenId);
```

```
        _setTokenURI(tokenId, uri);
```

```
        tokenURIs[tokenId] = uri;
```

```
    }
```

```
    mapping(address => mapping(string=>uint[])) public ownerDetails;
```

```
    function safeMint(address to, string memory propertyId) public payable {
```

```
        uint256 tokenId = _tokenIdCounter.current();
```

```
        _tokenIdCounter.increment();
```

```
        _safeMint(to, tokenId);
```

```
        _setTokenURI(tokenId, uri);
```

```
        tokenURIs[tokenId] = uri;
```

```
        ownerDetails[to][propertyId].push(tokenId);
```

```
    }
```

```
    function transfer(address from, address to, string memory propertyId) public payable{
```

```

uint tknId = ownerDetails[from][propertyId][0];
require(tknId>0, "Seller doesnt have the corresponding token");
safeTransferFrom(from, to, tknId);
ownerDetails[to][propertyId].push(tknId);
uint[] storage tokens = ownerDetails[from][propertyId];
for(uint i = 1;i<tokens.length;i++){
    tokens[i]=tokens[i++];
}
tokens.pop();
}

// The following functions are overrides required by Solidity.
function _burn(uint256 tokenId) internal override(ERC721, ERC721URIStorage) {
    super._burn(tokenId);
}

function tokenURI(uint256 tokenId)
    public
    view
    override(ERC721, ERC721URIStorage)
    returns (string memory)
{
    return tokenURIs[tokenId];
}
}

```

Here are the 2 mappings that we are using

1. `mapping(uint=>string) tokenURIs =>` It contains mapping of tokenURIs with the tokenId.
2. `mapping(address => mapping(string=>uint[])) public ownerDetails =>` It contains the token id for a particular property NFT that the owner has. PropertyId is very important for us and so we are using it in addition with tokenId.

Here are the 2 functions that we created:

```
1. function safeMint(address to, string memory propertyId) public payable {
    uint256 tokenId = _tokenIdCounter.current();
    _tokenIdCounter.increment();
    _safeMint(to, tokenId);
    _setTokenURI(tokenId, uri);
    tokenURIs[tokenId] = uri;
    ownerDetails[to][propertyId].push(tokenId);
}
```

It accepts address and propertyId as the argument and mints the NFT for that user. It also accepts propertyId which we can add to the metadata of our NFT to distinguish different NFTs. And then it updates our mappings accordingly.

```
2. function transfer(address from, address to, string memory propertyId) public payable{
    uint tokenId = ownerDetails[from][propertyId][0];
    require(tokenId>0, "Seller doesnt have the corresponding token");
    safeTransferFrom(from, to, tokenId);
    ownerDetails[to][propertyId].push(tokenId);
    uint[] storage tokens = ownerDetails[from][propertyId];
    for(uint i = 1;i<tokens.length;i++){
        tokens[i]=tokens[i++];
    }
    tokens.pop();
}
```

It accepts from and to addresses and also the propertyId. Using the owner address and the propertyId we fetch the tokenId belonging to that user for that property and then call `safeTransferFrom`(from, to, tokenId) function internally.

→ We are using OpenSea testnet (<https://testnets.opensea.io/>) to view our NFTs

→ We have deployed our contract on Infura using Infura API key and mnemonics of our account address. We have stored the mnemonics and the API key in an .env file so that it is not exposed to the outside world. Here is how our truffle-config.js looks like.

```

require('dotenv').config();
const HDWalletProvider = require('@truffle/hdwallet-provider');
const { INFURA_API_KEY, MNEMONIC } = process.env;
module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",
      port: 8545,
      network_id: "5"
    },
    goerli: {
      provider: () => new HDWalletProvider(MNEMONIC,
        INFURA_API_KEY),
      network_id: '5',
      gas: 4465030
    }
  },
  compilers: {
    solc: {
      version: "0.8.1",
      settings: {
        optimizer: {
          enabled: false,
          runs: 200
        }
      }
    }
  },
  contracts_directory: 'contracts',
  contracts_build_directory: 'abis'
};

```

→ How to deploy:

Go inside the contract directory : `cd contract/`

Then use `truffle compile` to compile all the files

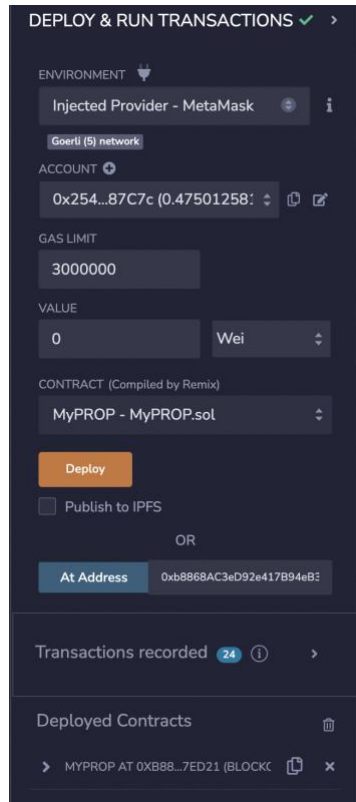
Then use `truffle migrate -- network goerli` to deploy.

→ Using the contract address we tested all the edge cases on Remix.

→ How to Test on Remix:

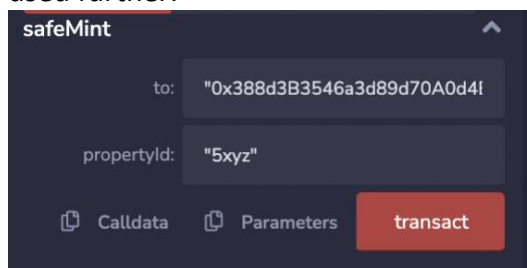
1. Copy the solidity file and paste it in a new remix file and compile.
2. Get the contract address from contract/abi/MyProp.Json → inside network object
3. In Remix deploy option use Metamask as Environment and choose appropriate account
4. Paste the contract address in At Address field at the bottom and click on At Address, it will fetch the contract.

This is how it will look like:



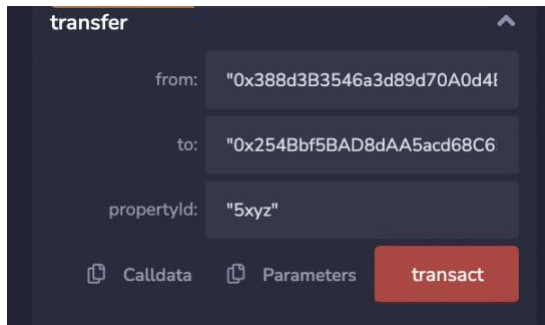
Click on the MyPROP deployed contract and you will be able to see all the functions and public fields.

1. Mint an NFT to get started. Make note of the property Id as it is our main key and will be used further.



2. Go to OpenSea testnet (<https://testnets.opensea.io/>) and verify the minted NFT.

3. Transfer the token to a different user using the propertyId.



The screenshot shows a dark-themed web interface for a 'transfer' transaction. It features three input fields: 'from:' with the value '0x388d3B3546a3d89d70A0d4I', 'to:' with the value '0x254Bbf5BAD8dAA5acd68C6', and 'propertyId:' with the value '5xyz'. Below these fields are two links, 'Calldata' and 'Parameters', each with a document icon. To the right of these links is a prominent red button labeled 'transact'.

4. Check open sea again, the token should be now transferred from old user to the new user.

→ As can be seen in Remix, similarly when a person buys a contract the safe mint function will be called and a new NFT will be created for the user, incase when they are buying directly from the owner of the property. In case they buy it from a seller, the NFT will be transferred from the seller to the buyer and no new NFT will be created.

→ In the NFT Metadata we can add details like propertyId and fractionNo. to uniquely distinguish all the NFTs.

→ Unfortunately, as Heroku was asking for charges we couldn't deploy it to Heroku.