CRYPTO ABODE

Project Members:

Name : Sayali Madhusudan Martal
Email : smartal@buffalo.edu
Person No : 50367344

Name: Vaishnavi Ruppa Gangadharan
Email : vruppaga@buffalo.edu
Person No : 50418483

The main goal of our project is to implement a Blockchain platform for the Real Estate Marketplace that transacts using custom RealT tokens. Our goal is to tokenize real estate assets which can be traded much like stocks on an exchange, thus allowing digital real estate transactions. In real estate, the trust of a website, listing or an agent is most important. Using Distributed Ledger Technology (DLT) will lead to increased trust and transparency on the usage of funds as this information is registered on the blockchain and publicly available. The RealT token will be the standard currency for buying, selling or renting real estate. Once a deal is finalized, an agreement (smart contract) would take place between the entities involved along with the transfer of the amount in ethereum currency.

Uniqueness of our Marketplace:

Our platform is not just a simple buying or selling application but it’s an auction for the real estate properties where the seller posts the estate and the buyers can enter the bid amount.
The seller can stop the bidding at any time according to the bidding amount.
By this way, the auction benefits both the buyer and the seller.

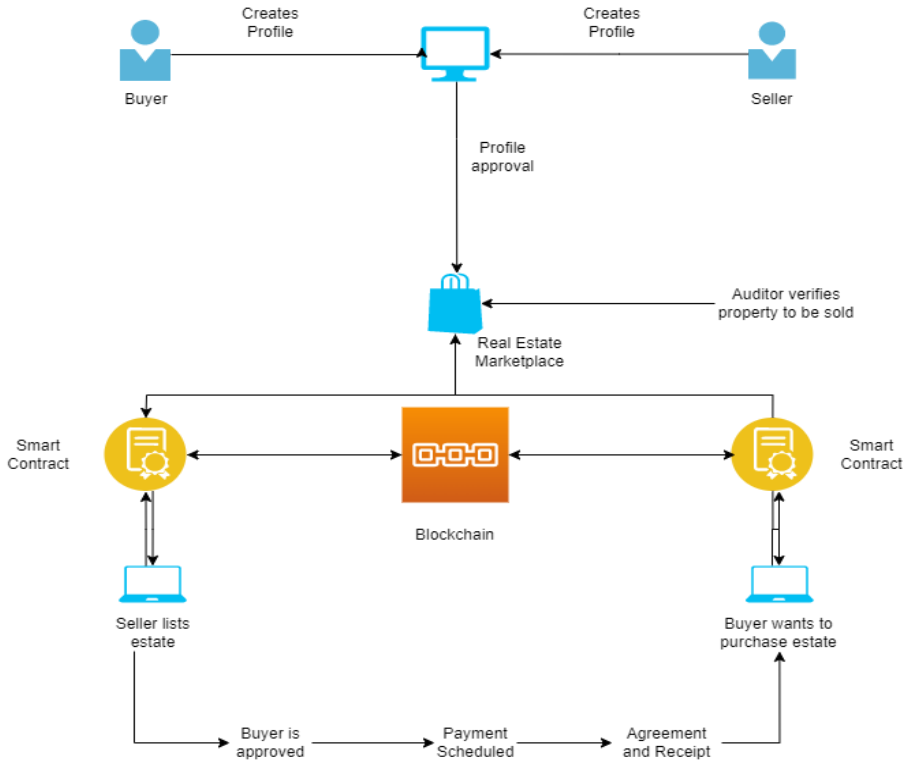
Token Symbol and Reasoning:

Token Name: RealT (The Real Estate Token)
Marketplace Name: Crypto Abode

This logo symbolizes the secure way of buying and selling of assets using the ERC20 token. Buyers can purchase RealT tokens in exchange for ETH. These tokens can be used to purchase homes, lands or rentals and to pay the cost of the real estate to the seller.

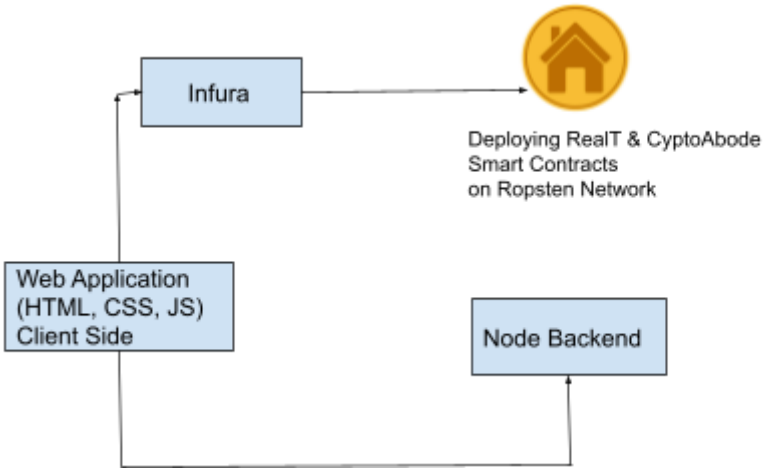
UML diagram:

Initial distribution of the RealT tokens take place while the admin approves the registration of the users. While bidding is taking place over the real estate properties, transfer of tokens takes place between the buyers and the seller.



Using RealT tokens

Architectural Diagram:



Explanation for data structures, modifiers, functions:

Structures:

struct UserDetails{} → This structure is used for storing the user's details and it contains information which includes id, usertype(1: Admin, 2: Auditor, 3: Buyer / Seller), name, contact, location, estates owned and a boolean value for checking if the user is registered or not.

struct EstateListing{} → This structure is used for storing the estate's details and it contains information which includes estimated minimum required price, image url, features, a boolean value for checking if the property can be sold or not, owner's address, a boolean value for checking if the property is verified by the auditor or not, highest bidder's address and highest bid amount value.

Mappings:

mapping(address => UserDetails) public users; → This mapping stores the address of all the users added with their details in the form of UserDetails structure.

mapping (uint => EstateListing) public listedEstates; → This mapping stores the

estate's details of all the estates posted in the form of EstateListings structure.

Variables:
uint public numEstates; → This is used for tracking the number of estates posted.
uint public numUsers; → This is used for tracking the number of users registered in this particular application.

Modifiers:

- modifier onlyAdmin {
  require(msg.sender == _admin);
  _;
}

→ This modifier can be used to check if the specified functions are performed only by the admin. Eg: Registering the users can be done only by the admin.

- modifier onlyAuditor {
  require(users[msg.sender].user_type == 2);
  _;
}

→ This modifier can be used to check if the specified functions are performed only by the auditor. Eg: Verifying the properties posted by the users can be done only by the auditor.

- modifier onlyBuyerOrSeller{
  require(users[msg.sender].user_type == 3);
  _;
}

→ This modifier can be used to check if the specified functions are performed only by the users. Eg: Placing the bid, Stopping the bid, Adding the estate details can be done by the users.

Transferring RealT tokens:

```
function transferRealTToken(address to, uint amount) public{
    realT.approve(msg.sender,amount);
    realT.transferFrom(msg.sender, to, amount);
}
```

This function has been used for transferring the tokens which we created using ERC20.sol from openzeppelin within the users in the marketplace.

Airdrop Functionality:

```
function transferRealTTokenAirDrop(address to, uint amount) onlyAdmin public{
    realT.approve(msg.sender,amount);
    realT.transferFrom(msg.sender, to, amount);
}
```

This function has been used for Air drop functionality where the contract's deployer will be able to send any number of tokens to the users in the marketplace when provided with their account address at any time.

Function for adding estate details:

```
function addEstate(string memory _location, uint _cost,string memory imageURL, string memory feat) public returns(bool)
{
    numEstates = numEstates + 1;
    //address payable highBidder = address(0);
    address highBidder = address(0);
    EstateListing memory myEstate = EstateListing(
        {
            owner: msg.sender,
            estateAddress: _location,
            requiredPrice: _cost,
            imgUrl: imageURL,
            features: feat,
            toBeSold: true,
            isVerified: false,
            months: 0,
            estateType: 1,
            highestBid: _cost,
            highestBidder: highBidder
        });
    listedEstates[numEstates] = myEstate;
    users[msg.sender].estates.push(numEstates);

    return true;
}
```

A person who is a seller (User Type 3) generally posts the estate details. The details of the estate are passed as parameters in this function. The parameters include location of the estate, estimated minimum price of the estate, url of the estate image and features of the posted property.

Whenever an estate is posted by someone, the number of estate variable gets incremented and the details get stored in the form of EstateListings's structure. This structure which contains the details of this estate is added to the listed estates mapping in order to have a track. This event is emitted finally so that the event logs can be recorded on the blockchain.

Function for placing the bid:

```
function placeBid (uint estateIndex,uint bidValue) onlyVerifiedProperty(estateIndex) onlyBuyerOrSeller public payable{
    //require(bidValue >= listedEstates[estateIndex].requiredPrice,"User does not have enough balance to buy this estate");
    checkRealTAndBid(bidValue); ///******* */
    require(bidValue >= listedEstates[estateIndex].highestBid,"User's bid is lower than the highest bid.");
    require(msg.sender != listedEstates[estateIndex].owner ,"Seller itself cannot be the buyer");
    require(listedEstates[estateIndex].toBeSold == true, "Not for sale");
    require(listedEstates[estateIndex].estateType == 1, "Estate type is not sellable");

    if(listedEstates[estateIndex].highestBidder == address(0)){
        // HighestBidder is the firstBidder;
        listedEstates[estateIndex].highestBid = bidValue;
        listedEstates[estateIndex].highestBidder = msg.sender;
    }else{
        // Revert amount to previous bidder logic.
        address _previousHighestBidder = listedEstates[estateIndex].highestBidder;
        uint _previousHighestBid = listedEstates[estateIndex].highestBid;

        listedEstates[estateIndex].highestBid = bidValue;
        listedEstates[estateIndex].highestBidder = msg.sender;
        //(_previousHighestBidder).transfer(_previousHighestBid);
        //payable(_previousHighestBidder).transfer(_previousHighestBid);
        // transferRealTToken(_previousHighestBidder, _previousHighestBid);
        realT.approve(listedEstates[estateIndex].owner,_previousHighestBid);
        realT.transferFrom(listedEstates[estateIndex].owner, _previousHighestBidder, _previousHighestBid);

    }
    transferRealTToken(listedEstates[estateIndex].owner, bidValue);

    // emit Transfer(seller,msg.sender, estateIndex);
}
```

Only the buyer is eligible to place the bid which is checked with the help of modifiers because these kinds of users are assigned the user type 3.

For the particular estate, if the buyer has enough minimum required balance to place the bid and if the amount which the buyer bids is greater than the previously highest amount bid and also if the buyer is also not the owner, the buyer will finally be able to successfully bid the amount which he enters after checking the prescribed conditions in the require function.

The amount which the buyer bids also gets deducted from his account. Using transferRealTToken, the tokens are transferred between the users in the marketplace.

Function for stopping the bid:

```
function stopBidding (uint estateIndex) onlyVerifiedProperty(estateIndex) onlyBuyerOrSeller payable public {
    require(listedEstates[estateIndex].owner == msg.sender, "Only owner can stop a bid.");
    require(listedEstates[estateIndex].toBeSold == true, "Cannot stop a bid if estate is not to be sold.");
    require(listedEstates[estateIndex].estateType == 1, "Estate type is not sellable");

    listedEstates[estateIndex].toBeSold = false;
    listedEstates[estateIndex].owner = listedEstates[estateIndex].highestBidder;
    listedEstates[estateIndex].highestBidder = address(0);

    // emit Transfer(seller,msg.sender, estateIndex);
}
```

Only the seller is eligible to stop the bid. Once the seller finds the highest bid amount is fair enough according to the current property value, the seller has all rights to stop the bid and sell the property to the person who placed the highest bid value.

This function helps the seller to stop the bid and the estate is sold to the highest bidder. The estate's owner is updated to the account address of the buyer that won the auction.

RealEstateMarketplace QUAD chart: use case, issues, blockchain solution, and benefits

Use case: Real Estate Marketplace Problem statement: a decentralized blockchain-based Real estate marketplace	**Issues with existing centralized model:** ● Lengthy and expensive buying and selling procedure. ● There may be many brokers who are not trustable. Either the property might be fake or the broker might be selling the property at a very high price. Using blockchain will result in saving commissions and fees charged by them and will also make the process much quicker. ● Lack of transparency in the process which leads to increased fraud. ● No option to review and sign documents online unless maintained by the third parties. ● Relatively decreased liquidity of real estate assets.

Proposed Blockchain Solution:	**Benefits:**
● Users can be of three types in our blockchain application. → Admin → Auditor → Buyers/Sellers ● The constructor which is defined in our contract is useful for assigning the person who deploys the application as admin. ● Only the admin has the rights to register the auditor and others who are the buyers and sellers. ● Only the auditor has the rights to verify the property before it gets posted in the user interface. ● Any user who is a buyer or seller will be able to post the estate details for sale. The details of the estate must be location, estimated minimum price, features and the estate's image URL. ● Our platform is not just a simple buying or selling application but it's an auction for the real estate properties where the seller posts the estate and the buyers can enter the bid amount. ● The seller can stop the bidding at any time according to the bidding amount. ● By this way, the auction benefits both the buyer and the seller.	This auction benefits both the buyer and seller in various ways. ● Investments done in this way are considered as smart as the properties are usually purchased at fair market value through competitive bidding. ● Buyers know they are competing fairly and on the same terms as all other buyers. ● This results in increased market efficiency by acceleration in sales. ● There will be an assurance that property will be sold at true market value. ● This platform exposes the property to a large number of people. ● Regarding negotiation: ● The seller is eliminated from the negotiation process. ● Long negotiation periods are avoided.

Contract Diagram:

This contains the contract name, Data, Modifier details, function details.

For CryptoAbode.sol

CryptoAbode
struct UserDetails{} struct EstateListing{}

```
mapping(address => UserDetails) public users;
mapping (uint => EstateListing) public listedEstates;
uint public numEstates;
uint public numUsers;
address _admin;
RealT realT
```

```
modifier onlyAdmin {
    require(msg.sender == _admin);
    _;
}

modifier onlyAuditor {
    require(users[msg.sender].user_type == 2);
    _;
}

modifier onlyBuyerOrSeller{
    require(users[msg.sender].user_type == 3);
    _;
}

modifier onlyVerifiedProperty (uint estateIndex){
    require(listedEstates[estateIndex].isVerified == true);
    _;
}
```

```
constructor() payable{
    _admin = msg.sender;
    users[msg.sender].user_type = 1;
}

addUser (string memory name, string memory contact, string memory location)
public{}

registerUser (address userAddr, uint user_type) public onlyAdmin{}

verifyProperty(uint estateIndex) public onlyAuditor{}

addEstate(string memory _location, uint _cost,string memory imageURL, string
```

memory feat) public onlyBuyerOrSeller{}

getEstateDetails(uint _index) onlyBuyerOrSeller public view returns (string memory, uint,string memory,string memory, bool, address, bool) {}

placeBid (uint estateIndex) onlyVerifiedProperty(estateIndex) onlyBuyerOrSeller payable public {}

stopBidding (uint estateIndex) onlyVerifiedProperty(estateIndex) onlyBuyerOrSeller payable public {}

function getUser (address userAddr) onlyAdmin public view returns (uint, uint,string memory,string memory, string memory, uint[] memory) {}

function checkIfAdmin() external view returns(bool){}

function checkIfAuditor() external view returns(bool){}

function getUserAddresses() public view returns (address[] memory){}

function getEstateCount() public view returns (uint){}
function checkRealTAndBid(uint bid) internal view{ }

function transferRealTToken(address to, uint amount) public{}

function transferRealTTokenAirDrop(address to, uint amount) onlyAdmin public{}

function getRealTBalance() public view returns(uint){}

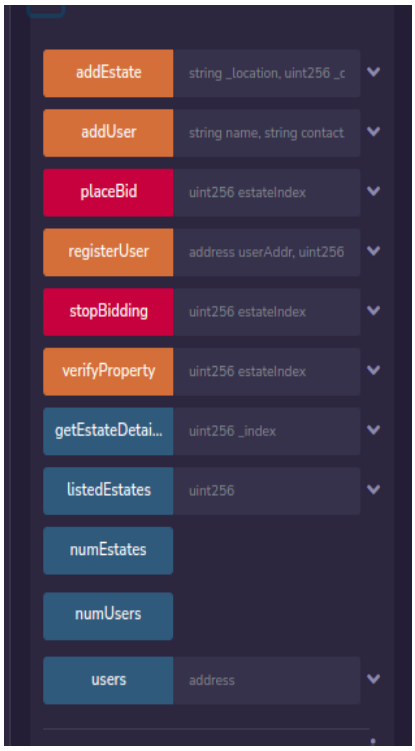
For RealT.sol

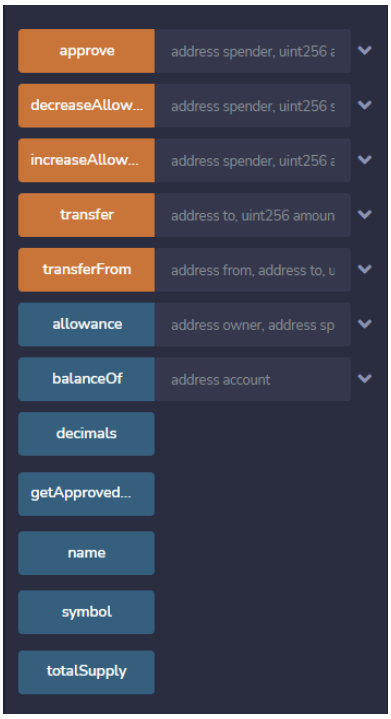
RealT
address _owner; address approvedMarket;

modifier onlyMyMarket(){
require(msg.sender == approvedMarket);
_;
}
 	
modifier onlyRealTAdmin(){
require(msg.sender == _owner);
_;
}

constructor() ERC20("The Real Estate Token", "RealT") {
_owner = msg.sender;
_mint(msg.sender, 1000 * 10 ** 18);
}
function fetchApprovedMarket() public view onlyMyMarket returns(address){}
 	
function updateApprovedMarket(address _approvedMarket) public onlyRealTAdmin{}
 	
function approve(address spender, uint256 amount) public virtual override returns (bool){}

Remix: CryptoAbode.sol RealT.sol

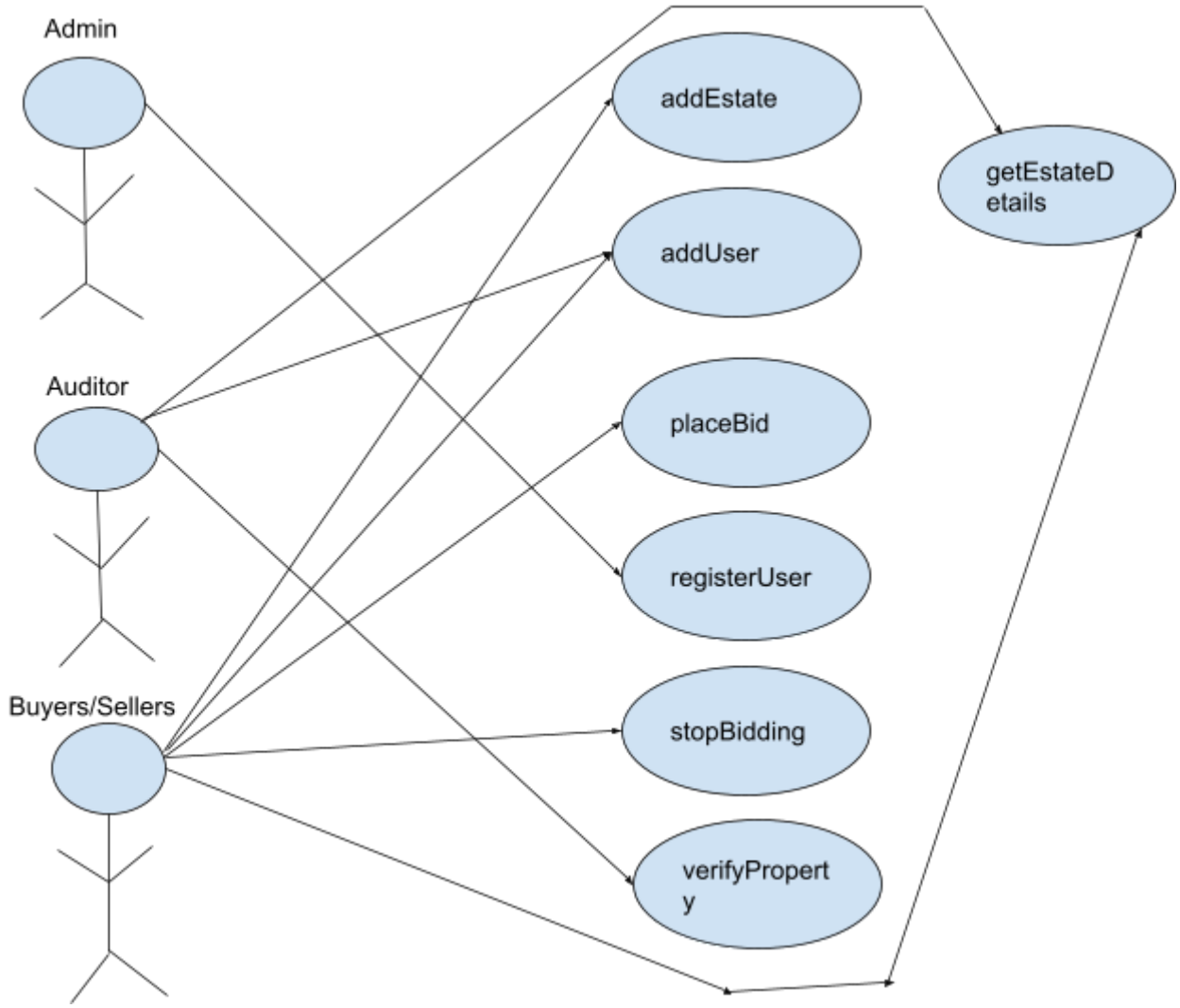




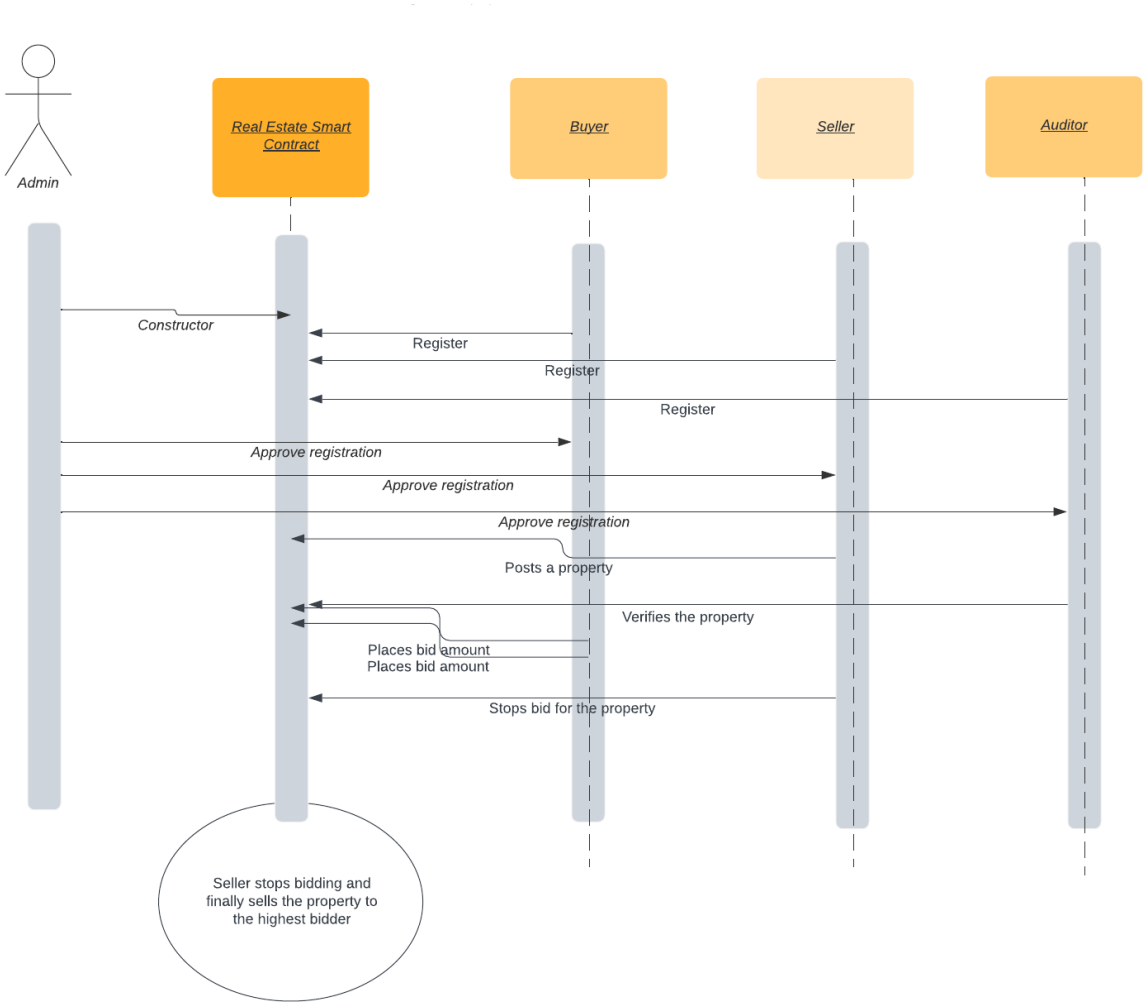
These are the main functions which we have used to implement our marketplace.

Use case diagram:

Users



Sequence diagram:



ERC-20 functionality and methods:

Some of the functions which we used to deploy our marketplace are

approve(address spender, uint256 amount) → This is the default approve function used to set the amount as the allowance of spender over the caller's tokens.

Overridden approve function: function approve(address spender, uint256 amount) public virtual override returns (bool){} → We have overridden the approve function in order to approve the marketplace and the spender.

transferFrom(address from, address to, uint256 amount) → This function moves the tokens from the sender to the receiver using the allowance mechanism. The amount is then deducted from the caller's allowance. This emits a transfer event.

Transfer(address from, address to, uint256 value) → This is an event emitted to indicate the transfer of tokens from the sender to the receiver.

allowance(address owner, address spender) → This function returns the remaining number of tokens that spender will be allowed to spend on behalf of the owner through transferFrom. This is zero by default. This value changes when approve or transferFrom are called.

balanceOf(address account) → This function returns the amount of tokens owned by the account.

The balanceOf function is utilized when a user clicks the 'Check My Balance' button from the UI. The approve and transferFrom functions are utilized when the Admin airdrops tokens to other users, as well as when a Buyer places a bid for a particular listed estate.

ERC-20 contract and CryptoAbode contract integration:

We are importing the RealT.sol (token solidity file) in our CryptoAbode.sol file. Then, we are integrating both the sol files by minting the tokens in RealT.sol and then using it in the marketplace. In order to ensure that the tokens are spent only by the users in the marketplace, we are using the approve function for approving the marketplace as well as the user. This is how we are integrating our marketplace with the ERC-20 contract.

Screenshots of the Solidity code:

CryptoAbode.sol

```
//SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;
import "./RealT.sol";///******** */
contract CryptoAbode{
    address _admin;
    RealT realT;///********* */

    struct UserDetails {
        uint id;
        uint user_type; // 1: Admin, 2: Auditor, 3: Buyer / Seller
        string name;
        string contact;
        string location;
        uint[] estates;
    }
    struct EstateListing {
        string estateAddress;
        uint requiredPrice;
        string imgUrl;
        string features;
        bool toBeSold;
        address owner;
        bool isVerified;
        uint estateType; // 1:Buy/sell, 2:Rent
        uint months;
        //address payable highestBidder;
        address highestBidder;
        uint highestBid;
    }

    mapping(address => UserDetails) public users;
    mapping (uint => EstateListing) public listedEstates;
    uint public numEstates=0; //total no of estates via this contract at any time
    uint public numUsers=0; //total no of users via this contract at any time
    address[] public userAddrs;

    event InitUser(address _user,RealT realT); //constructor event
```

```
    function addContractBalance() public payable{}

    function addUser (string memory name, string memory contact, string memory location) public {
        numUsers = numUsers + 1;
        UserDetails memory user = UserDetails(
            {
                id: numUsers,
                name: name,
                contact: contact,
                location: location,
                user_type: 0,
                estates: new uint[](0)
            });
        users[msg.sender] = user;
        userAddrs.push(msg.sender);

        emit AddUser(msg.sender, numUsers);

    }

    function registerUser (address userAddr, uint user_type) public onlyAdmin{
        require((user_type == 2 || user_type == 3), "User type must be either auditor or buyer or seller");
        users[userAddr].user_type = user_type;
        // realT.transferFrom(_admin, msg.sender, 100*10**18) ;
        transferRealTToken(userAddr, 100*10**18);
    }

    function verifyProperty(uint estateIndex) public onlyAuditor{
        listedEstates[estateIndex].isVerified = true;
    }

    function getContractBalance() public view returns(uint){
        return address(this).balance;
    }
```

```
function addEstate(string memory _location, uint _cost,string memory imageURL, string memory feat) public returns(bool)
{
    numEstates = numEstates + 1;
    //address payable highBidder = address(0);
    address highBidder = address(0);
    EstateListing memory myEstate = EstateListing(
        {
            owner: msg.sender,
            estateAddress: _location,
            requiredPrice: _cost,
            imgUrl: imageURL,
            features: feat,
            toBeSold: true,
            isVerified: false,
            months: 0,
            estateType: 1,
            highestBid: _cost,
            highestBidder: highBidder
        });
    listedEstates[numEstates] = myEstate;
    users[msg.sender].estates.push(numEstates);

    return true;
}

function getEstateDetails(uint _index) public view returns (string memory, uint,string memory,string memory, bool, address, boo
    return (
        listedEstates[_index].estateAddress,
        listedEstates[_index].requiredPrice,
        listedEstates[_index].imgUrl,
        listedEstates[_index].features,
        listedEstates[_index].toBeSold,
        listedEstates[_index].owner,
        listedEstates[_index].isVerified
    );
}
```

```
function getEstateCount() public view returns (uint){
    return numEstates;
}

function getUserAddresses() public view returns (address[] memory){
    return userAddrs;
}

function checkIfAdmin() external view returns(bool){
    if(_admin == msg.sender){
        return true;
    }
    return false;
}

function checkIfAuditor() external view returns(bool){
    if(users[msg.sender].user_type == 2){
        return true;
    }
    return false;
}

function getUser (address userAddr) onlyAdmin public view returns (uint, uint,string memory,string memory, string memory, uint[
    return (
    users[userAddr].id,
    users[userAddr].user_type,
    users[userAddr].name,
    users[userAddr].contact,
    users[userAddr].location,
    users[userAddr].estates
    );
}
```

```
function placeBid (uint estateIndex,uint bidValue) onlyVerifiedProperty(estateIndex) onlyBuyerOrSeller public payable{
    //require(bidValue >= listedEstates[estateIndex].requiredPrice,"User does not have enough balance to buy this estate");
    checkRealTAndBid(bidValue); ///******* */
    require(bidValue >= listedEstates[estateIndex].highestBid,"User's bid is lower than the highest bid.");
    require(msg.sender != listedEstates[estateIndex].owner ,"Seller itself cannot be the buyer");
    require(listedEstates[estateIndex].toBeSold == true, "Not for sale");
    require(listedEstates[estateIndex].estateType == 1, "Estate type is not sellable");

    if(listedEstates[estateIndex].highestBidder == address(0)){
        // HighestBidder is the firstBidder;
        listedEstates[estateIndex].highestBid = bidValue;
        listedEstates[estateIndex].highestBidder = msg.sender;
    }else{
        // Revert amount to previous bidder logic.
        address _previousHighestBidder = listedEstates[estateIndex].highestBidder;
        uint _previousHighestBid = listedEstates[estateIndex].highestBid;

        listedEstates[estateIndex].highestBid = bidValue;
        listedEstates[estateIndex].highestBidder = msg.sender;
        //(_previousHighestBidder).transfer(_previousHighestBid);
        //payable(_previousHighestBidder).transfer(_previousHighestBid);
        // transferRealTToken(_previousHighestBidder, _previousHighestBid);
        realT.approve(listedEstates[estateIndex].owner,_previousHighestBid);
        realT.transferFrom(listedEstates[estateIndex].owner, _previousHighestBidder, _previousHighestBid);

    }
    transferRealTToken(listedEstates[estateIndex].owner, bidValue);

    // emit Transfer(seller,msg.sender, estateIndex);
}
```

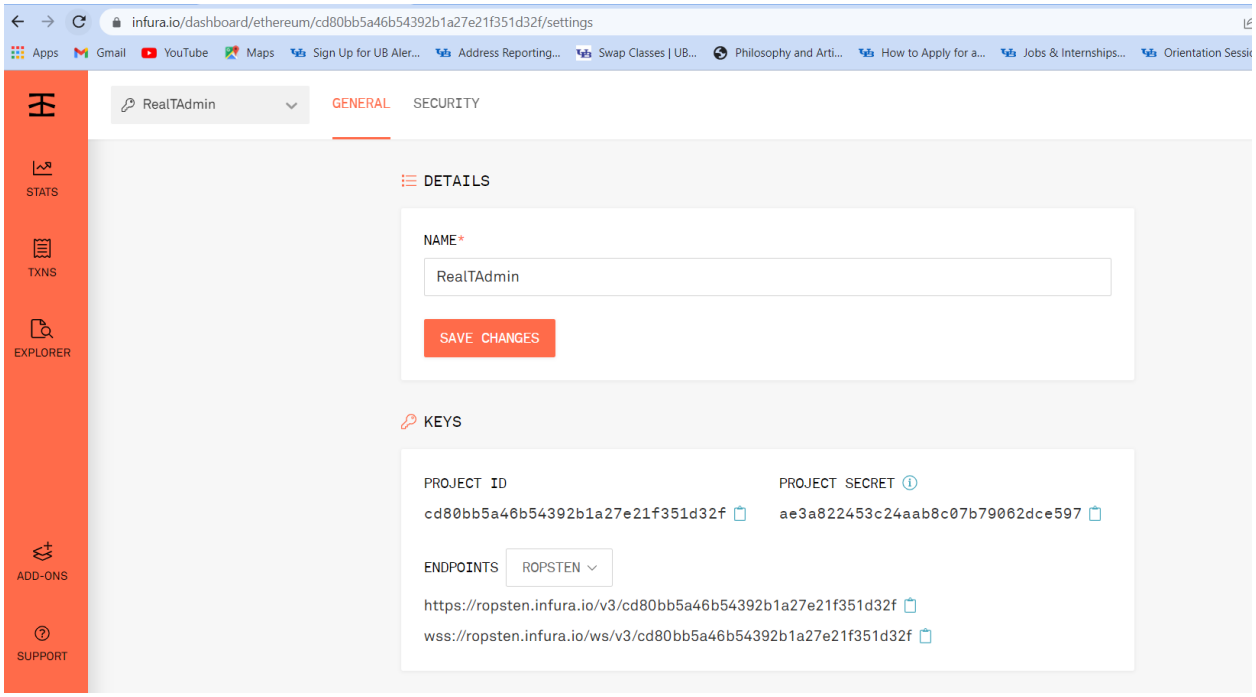
```
function stopBidding (uint estateIndex) onlyVerifiedProperty(estateIndex) onlyBuyerOrSeller payable public {
    require(listedEstates[estateIndex].owner == msg.sender, "Only owner can stop a bid.");
    require(listedEstates[estateIndex].toBeSold == true, "Cannot stop a bid if estate is not to be sold.");
    require(listedEstates[estateIndex].estateType == 1, "Estate type is not sellable");

    listedEstates[estateIndex].toBeSold = false;
    listedEstates[estateIndex].owner = listedEstates[estateIndex].highestBidder;
    listedEstates[estateIndex].highestBidder = address(0);

    // emit Transfer(seller,msg.sender, estateIndex);
}

function getRealTBalance() public view returns(uint){
    return realT.balanceOf(msg.sender);
}
// function getThisAddressTokenBalance() public view returns (uint256) {
//     return balances[address(this)];
// }
function checkRealTAndBid(uint bid) internal view{
    if(bid > realT.balanceOf(msg.sender)){
        revert("Not Enough RealT Tokens");
    }
}

function transferRealTToken(address to, uint amount) public{
    realT.approve(msg.sender,amount);
    realT.transferFrom(msg.sender, to, amount);
}

function transferRealTTokenAirDrop(address to, uint amount) onlyAdmin public{
    realT.approve(msg.sender,amount);
    realT.transferFrom(msg.sender, to, amount);
}
```

RealT.sol

```sol
//SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract RealT is ERC20 {
    address _owner;
    address approvedMarket;

    constructor() ERC20("The Real Estate Token", "RealT") {
        _owner = msg.sender;
        _mint(msg.sender, 1000 * 10 ** 18);
    }

    modifier onlyMyMarket(){
        require(msg.sender == approvedMarket);
        _;
    }

    modifier onlyRealTAdmin(){
        require(msg.sender == _owner);
        _;
    }

    function fetchApprovedMarket() public view onlyMyMarket returns(address){
        return approvedMarket;
    }

    function updateApprovedMarket(address _approvedMarket) public onlyRealTAdmin{
        approvedMarket = _approvedMarket;
    }

    function approve(address spender, uint256 amount) public virtual override returns (bool){
        _approve(spender, approvedMarket, amount);
        return true;
    }
}
```

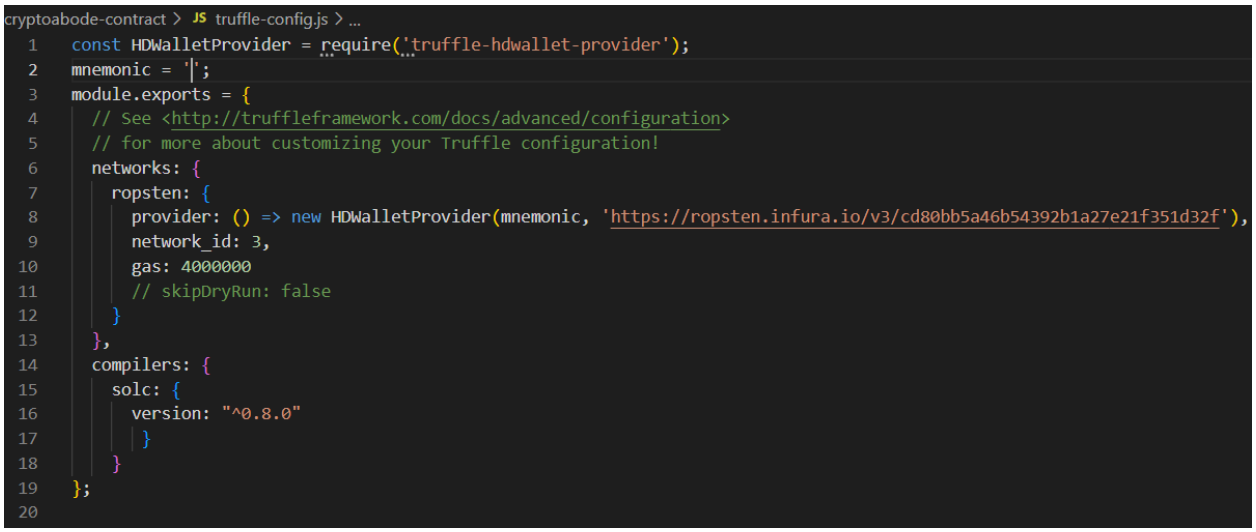
Deploying the application in Ropsten network using Infura (Phase 3):

To deploy the smart contracts to the Ropsten Test network using Infura, we first create an Infura account and then an Infura Project with product as 'Ethereum' as shown below:



We also need to install the Truffle hardware wallet provider for using Infura using npm.

The truffle-config.js needs to be updated to contain the 12 word mnemonic for the Admin account and the Infura endpoint URL as shown below:



The Infura URL also needs to be updated while using the Web3 provider in the app.js file.

To deploy the contract to Ropsten, run the following command:

Commands to be run from cryptoabode-contract/contracts/:

- truffle compile
- truffle migrate –network ropsten

Command to be run from cryptoabode-app/ to start the UI:

- npm start

Deploying the application in Ganache network (Phase 2):

Requirements: Web3, Solidity, Metamask extension, truffle, nodejs, ganache, remix-ethereum IDE.

The project code can be downloaded and extracted to a specific source.

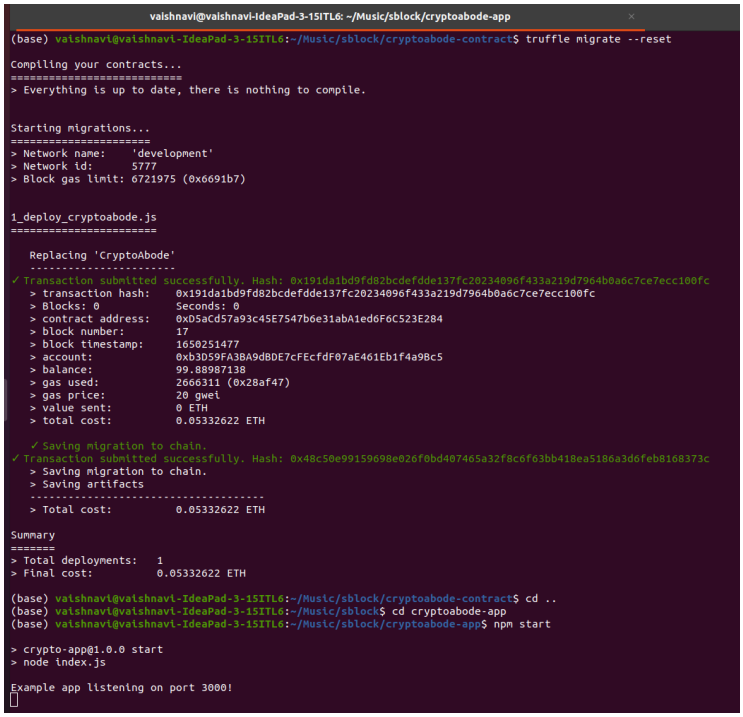
Commands to be run from cryptoabode-contract/contracts/:

- truffle compile
- truffle migrate –reset

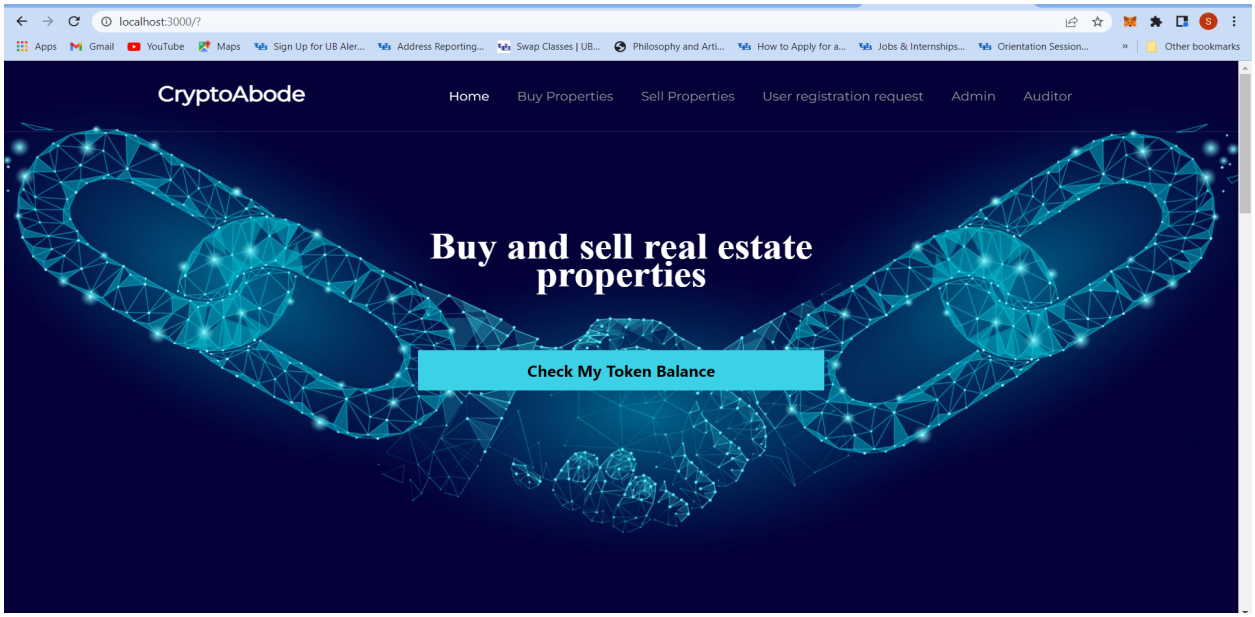
Command to be run from cryptoabode-app/:

- npm start

This is sufficient enough to deploy, test and interact with our built blockchain application.

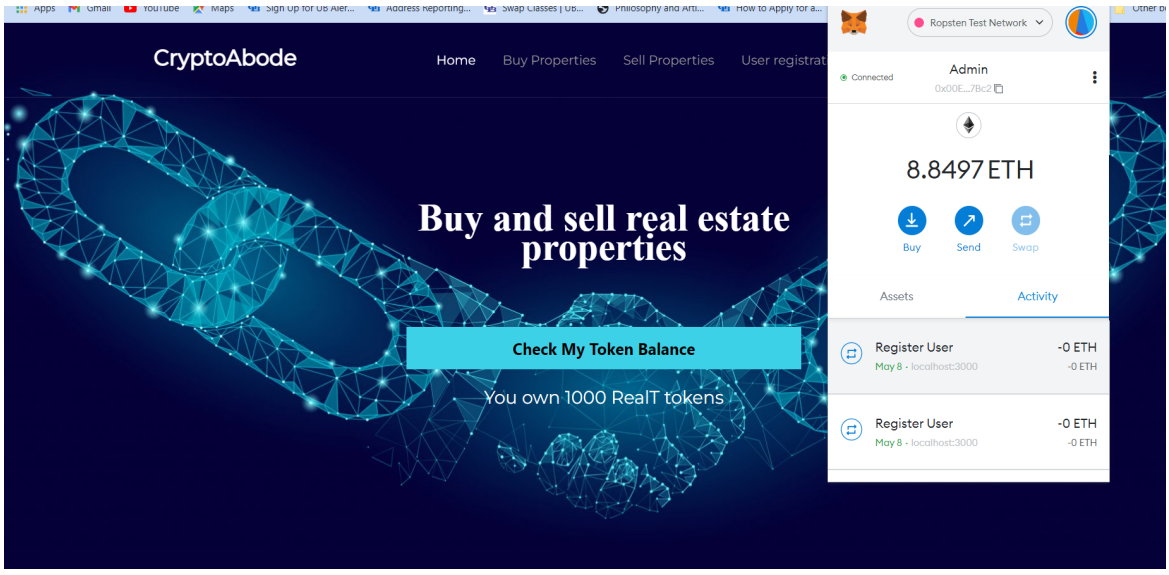


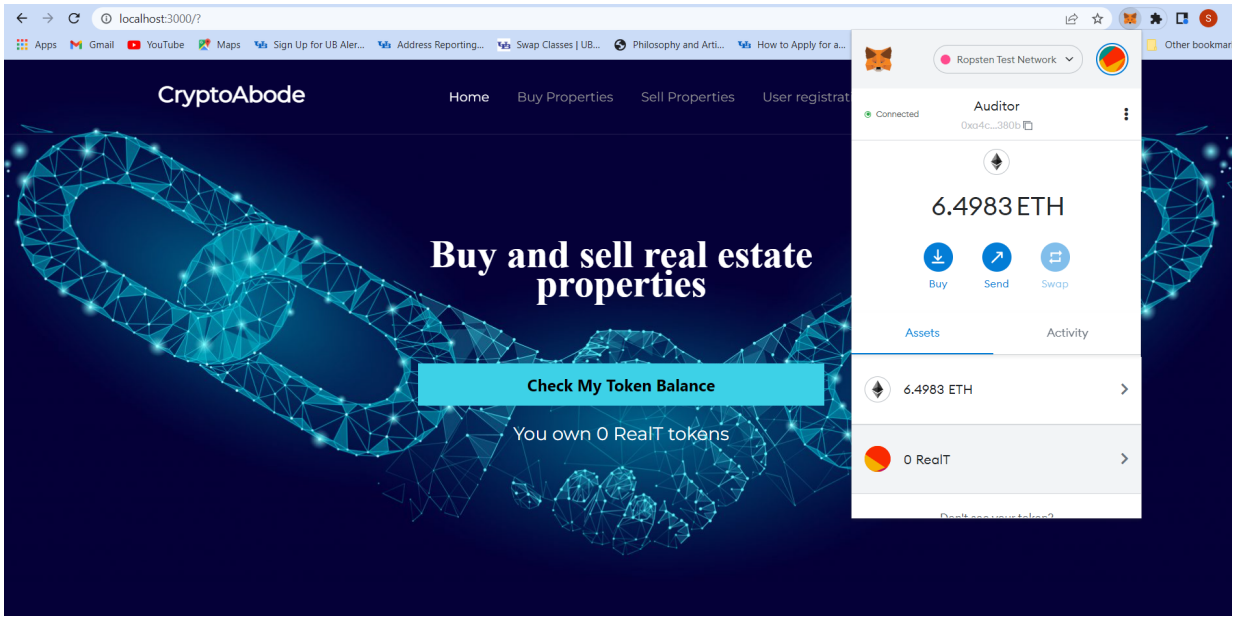
Screenshots of the working application :



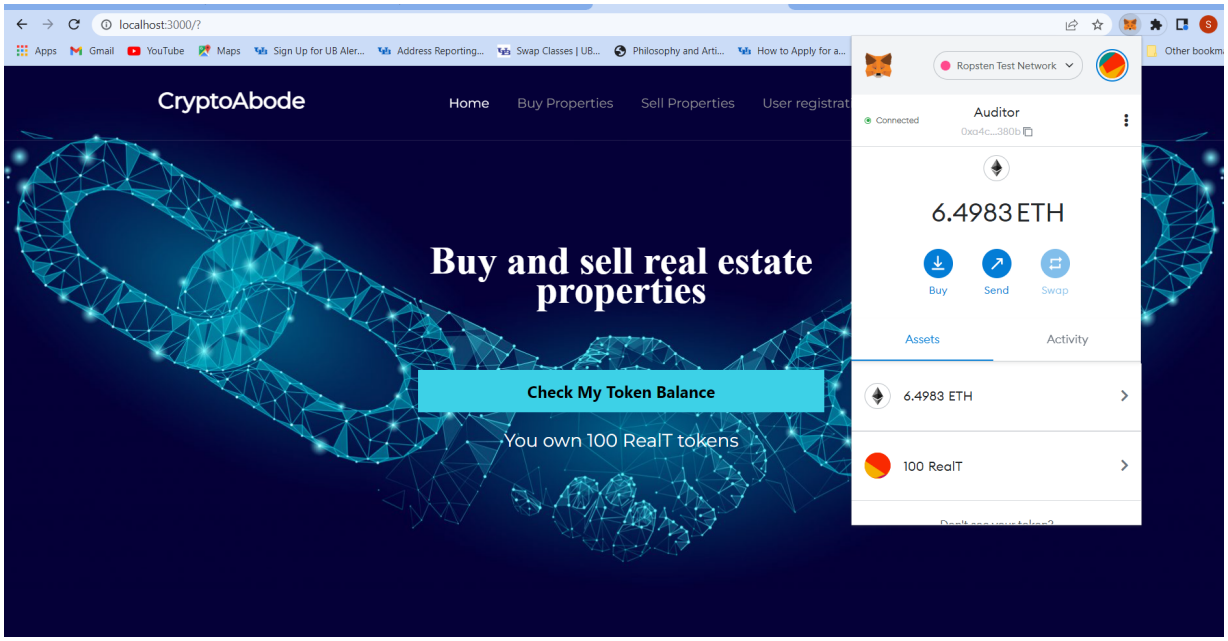
For Checking the Current User's Token Balance:

Initially the Admin has 1000 Tokens and other added users have 0 Tokens as shown below:

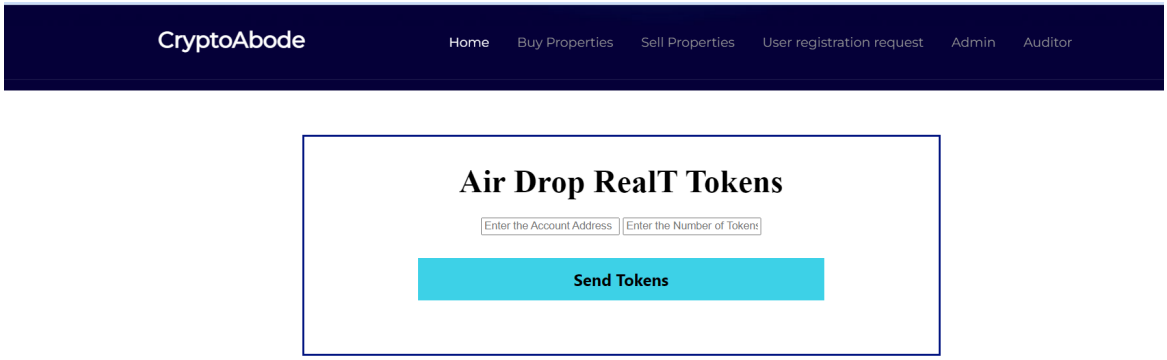




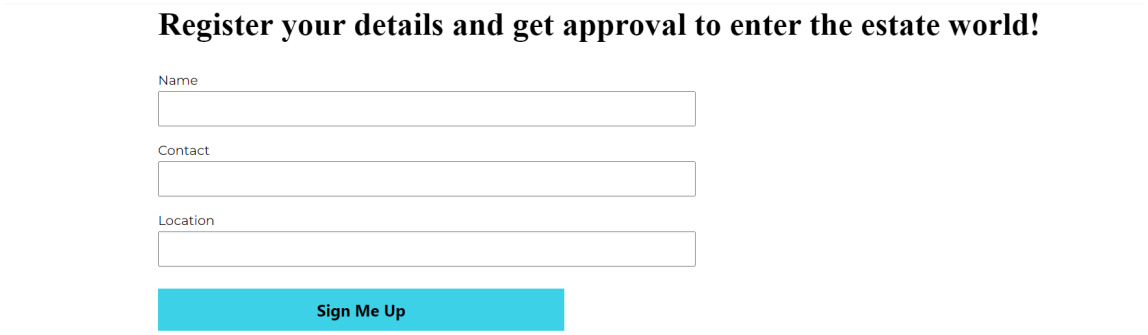
A user is granted 100 Tokens once registered by the Admin as shown below:



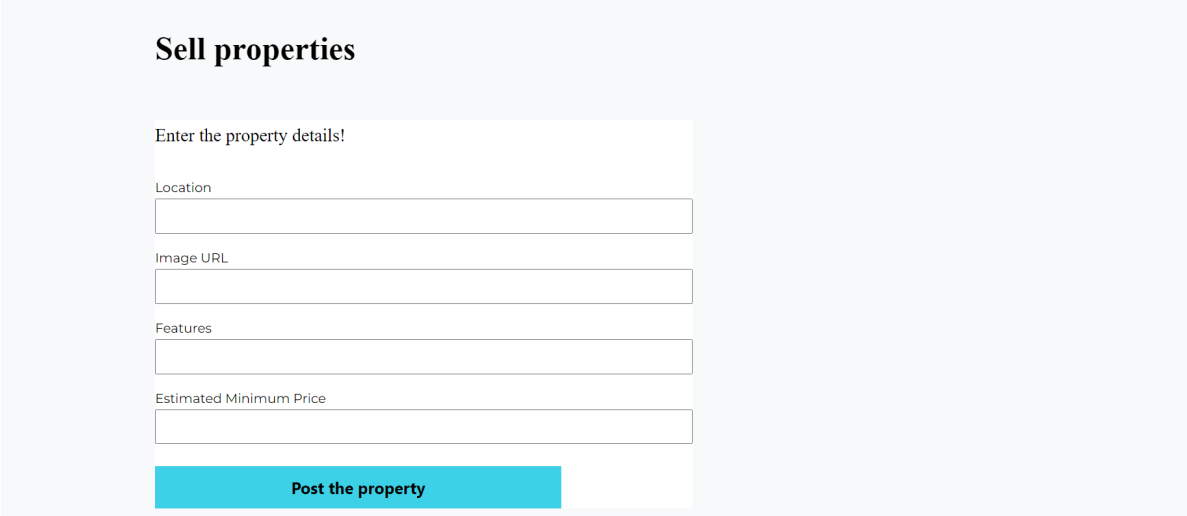
Admin can send additional Tokens to a user using Airdrop functionality if required:



For registration of users:



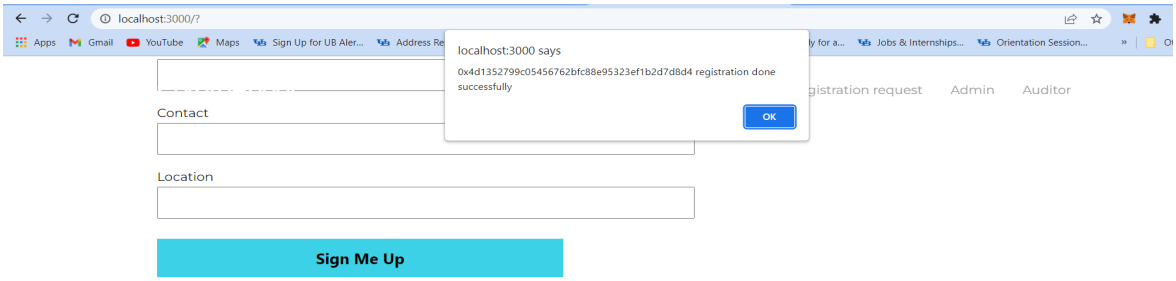
For Posting the property details:



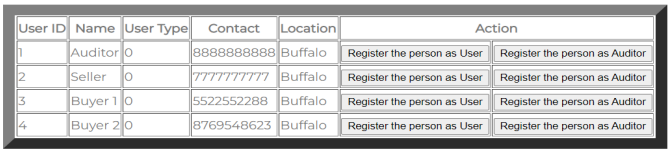
We have used 5 different Metamask accounts for the following roles:

Admin, Auditor, Seller, Buyer 1, Buyer 2

The list of added users is only accessible to the Admin. The Admin can approve the registrations as follows: (can approve the buyers/sellers or auditors)

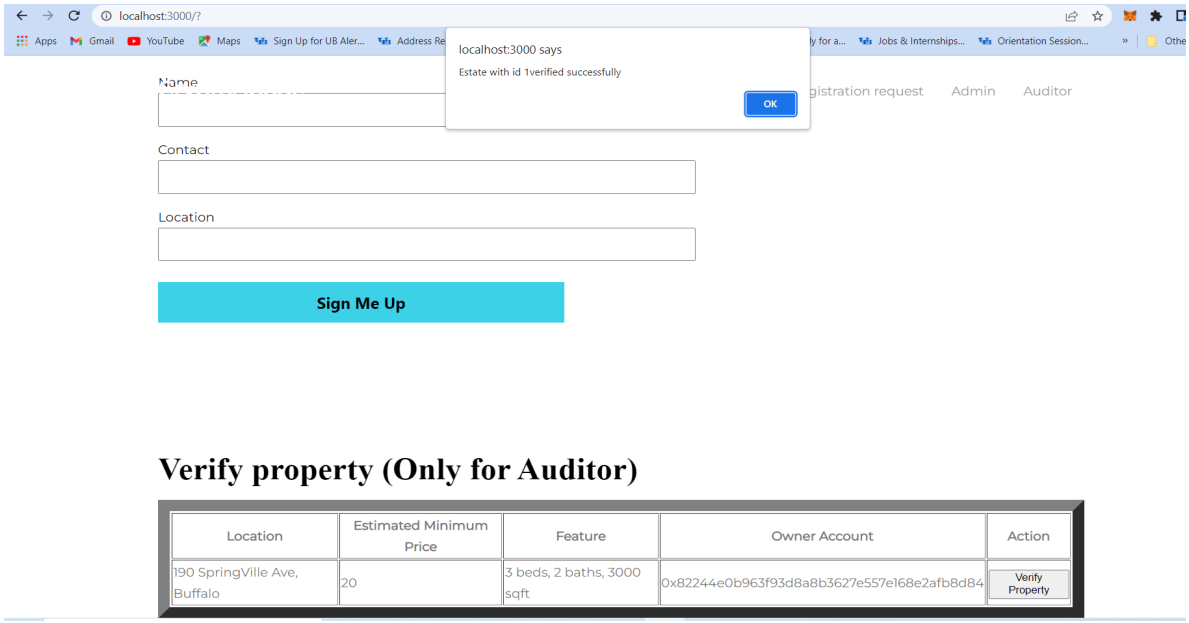


Approve user registration (Only for Admin)

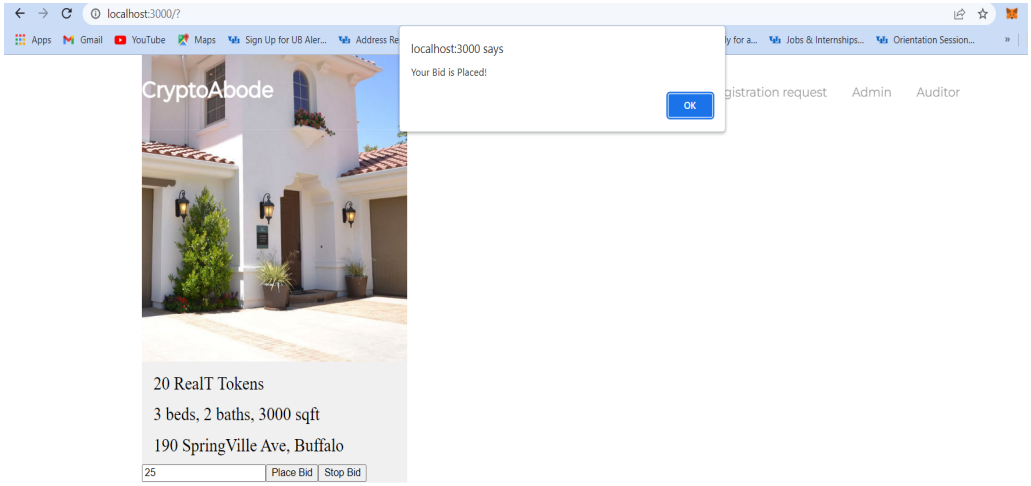


User ID	Name	User Type	Contact	Location	Action	
1	Auditor	0	8888888888	Buffalo	Register the person as User	Register the person as Auditor
2	Seller	0	7777777777	Buffalo	Register the person as User	Register the person as Auditor
3	Buyer 1	0	5522552288	Buffalo	Register the person as User	Register the person as Auditor
4	Buyer 2	0	8769548623	Buffalo	Register the person as User	Register the person as Auditor

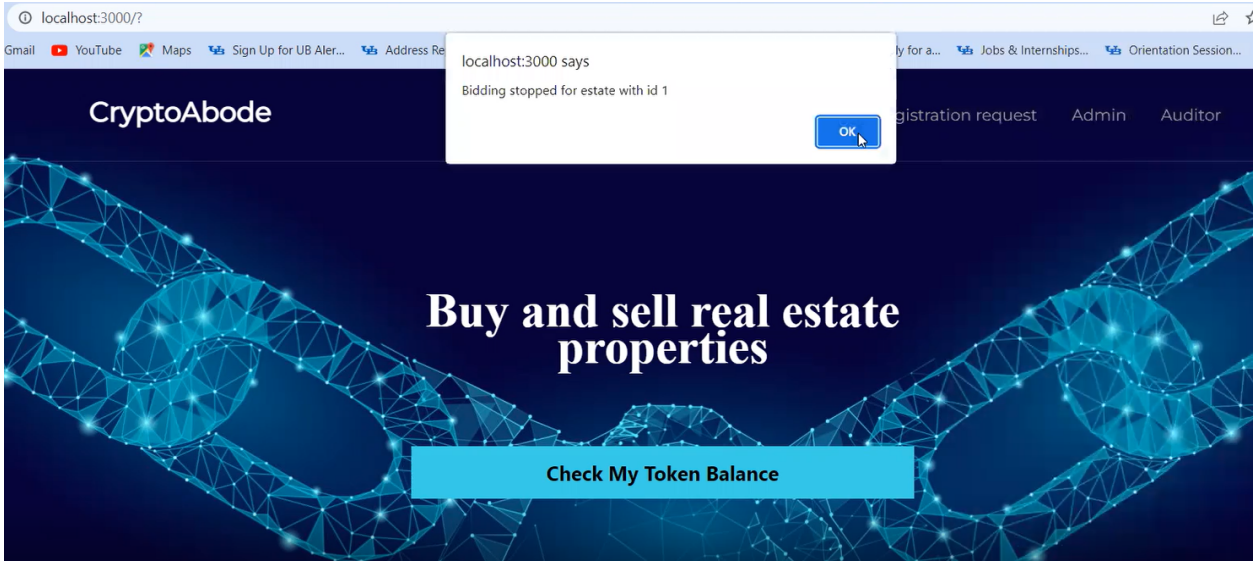
The List of estates to be verified is only visible to the Auditor. The Auditor can verify the listed estates by clicking the 'Verify Property' button across each listed estate as follows (This also shows the initial owner of the estate which is currently the Seller's account address):



The estates are visible to the buyers only after they have been verified. The buyers can then place bids as shown below:



The Seller can stop the bid any time using the 'Stop Bid' Button corresponding to an estate which creates an alert as shown below:



The listed estate disappears from the list of estates to be sold as soon as the bid is stopped. The change in the owner of the estate from the Seller's address to the Buyer's address is shown below:

Verify property (Only for Auditor)

Location	Estimated Minimum Price	Feature	Owner Account	Action
190 SpringVille Ave, Buffalo	20	3 beds, 2 baths, 3000 sqft	0xc697f426611dc85550d4586a701a73230eb7fa55	Verify Property

References:

1. **W3schools.com - For HTML, CSS**
2. **Blockchain in Action by Bina Ramamurthy**
3. **https://web3js.readthedocs.io/en/v1.7.3/**
4. **https://trufflesuite.com/docs/truffle/**
5. **Lucid Charts**
6. **https://api.jquery.com/**
7. **https://docs.openzeppelin.com/contracts/4.x/api/token/erc20#IERC20-approve-address-uint256-**