CSE 4/526 Handicraft NFT Marketplace (BitCrafty) using Blockchain

Manya Singh
manyasin@buffalo.edu
Person number: 50367314

Farhan Chowdhury
ftchowdh@buffalo.edu
Person number: 50304468

(Design Report - Phase 3)

Introduction:

We believe everyone in this world is unique in their own way and we also believe that the things they create by themselves also have unique character. To embrace that we have created a marketplace named “BitCrafty” which can serve the people to showcase talent in their hands and get a fair and secure worldwide trade of their creation.

About our idea:

We have seen that there are many people who create great artistic and utility handmade products and try to get some money in exchange for their work. Unfortunately, there are not many widespread platforms which are able to get fair prices for their creation and artists/collectors get disappointed. Our marketplace “BitCrafty” is all about unique identity and fair trade.

Bitcrafty is a Ethereum based NFT marketplace which ensures uniqueness of a product by defining its ownership and current pricing state on the blockchain and can be traded only on the platform by transferring the listed price.

Blockchain being an immutable ledger we can ensure instant transactions worldwide and ensure the certainty of the ownership of the products when they are bought.

What is a Non-Fungible Token?

A Non-Fungible Token (NFT) is used to identify something or someone in a unique way. This type of Token is perfect to be used on platforms that offer collectible items, access keys, lottery tickets, numbered seats for concerts and sports matches, etc. This special type of Token has amazing possibilities so it deserves a proper Standard, the ERC-721 came to solve that!

In our contract we are inspired by the ERC-721 standard and we are writing our own implementation of the transactions on the blockchain.

What is ERC-721?

The ERC-721 introduces a standard for NFT, in other words, this type of Token is unique and can have different value than another Token from the same Smart Contract, maybe due to its age, rarity or even something else like its visual. Wait, visual? Yes! All NFTs have a uint256 variable called tokenId, so for any ERC-721 Contract, the pair contract address, uint256 tokenId must be globally unique. That said, a Dapp can have a "converter" that uses the tokenId as input and outputs an image of something cool, like zombies, weapons, skills or amazing kitties!

For the this phase a custom Fungible Token (UNIQ) based on ERC20 standard will be introduced.

What is a Token?

Tokens can represent virtually anything in Ethereum:

- reputation points in online platform
- skills of a character in a game
- lottery tickets
- financial assets like a share in a company
- a fiat currency like USD
- an ounce of gold

Such a powerful feature of Ethereum must be handled by a robust standard, right? That's exactly where the ERC-20 plays its role! This standard allows developers to build token applications that are interoperable with other products and services.

What is ERC-20?

The ERC-20 introduces a standard for Fungible Tokens, in other words, they have a property that makes each Token be the same (in type and value) of another Token.

For example, an ERC-20 Token acts just like the ETH, meaning that 1 Token is and will always be equal to all the other Tokens.

Issue Addressed:

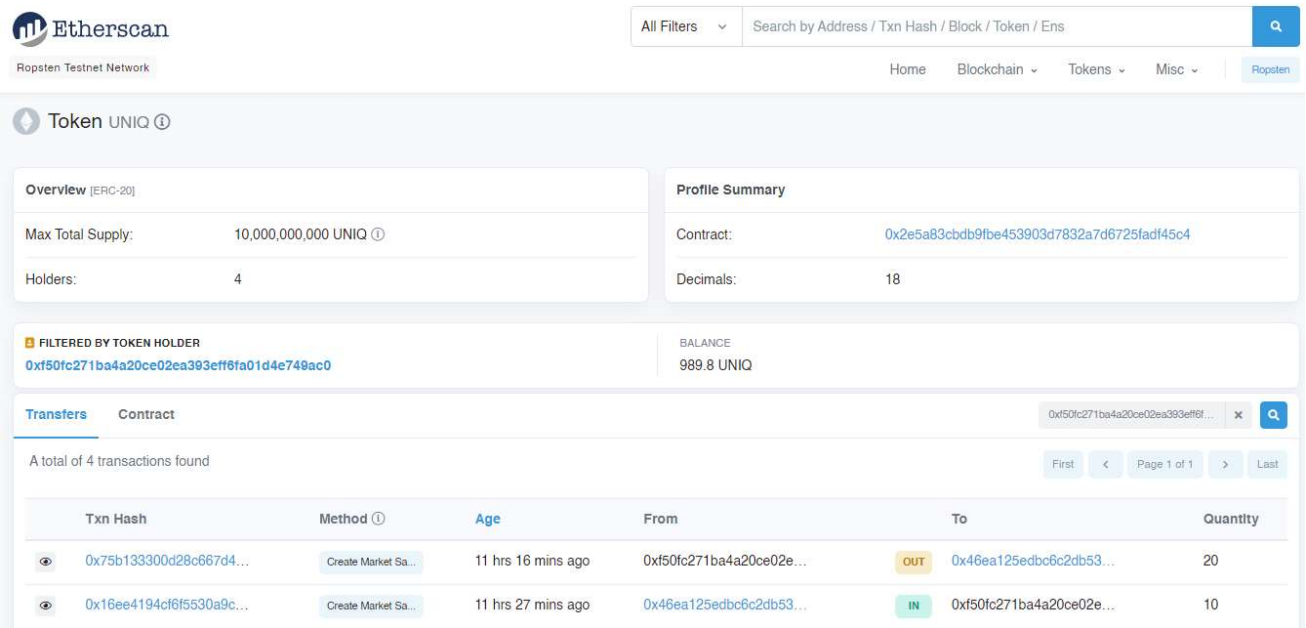
Currently, there is no worldwide marketplace for people to express their individuality and creativity and get a fair price for their creation. There are a few very localized non-distributed solutions available but the artists do not get a broader audience and they do not get fair compensation for their products.

Instruction to Deploy and Start: test and Interact:

1. Go-To: Project Root Directory and Open Command Window/ Terminal.
2. Run: "cd bitcrafty-app"
3. Run: "npm install --force"
4. Run: "cd .."
5. Run: "cd bitcrafty-contract"
6. Run: "npm install --force"
7. Make sure metamask is connected to ropsten network and have the accounts added.
8. Go to the Open metamask and Go To: Security and Privacy in settings and click on “Reveal Secret Recovery Phrase” to get your 12-word security phrase if you don’t have it already.
9. Go To: truffle-config.js file to paste the MNEMONIC on line 4 to the 12-word secret phrase you copied in previous step.
10. Run: "truffle migerate --network ropsten"
11. Copy both Contract Address and token address after file 2_deploy_contracts.js is executed from previous step and paste the address in config.js file located at BitCrafty-Dapp\bitcrafty-app\src\config.js
12. Run: "cd .."
13. Run: "cd bitcrafty-app"
14. Run: "npm start"
15. Visit http://localhost:3000/ .

Note: If you have the contract already deployed to Ropsten network there is no need to redeploy and, in that case, follow the following steps. (In our case contracts are already deployed to Ropsten by developers.)

1. Go-To: Project Root Directory and Open Command Window/ Terminal.
2. Run: "cd bitcrafty-app"
3. Run: "npm install --force"
4. Run: "npm start"
5. Visit http://localhost:3000/ .
6. Make sure your account have some ropsten Ether to pay for the gas price.
7. New Users visiting above homepage will be given option to avail 1000 UNIQ joining bonus and avail using clicking on button “Get your token now” using some gas.
8. Go to your account in metamask and click on “View Account on Etherscan” to get detailed overview of the account.
9. On the etherscan homepage (For example: https://ropsten.etherscan.io/address/0xf50fc271ba4a20ce02ea393eff6fa01d4e749ac0) , change the token to UNIQ and see the following information about our ERC 20 token, and your own balance.
10. Token information like total supply, number of holders can be found here along with all the transactions on the chain for that account.



Used node Dependencies:

- **ethers**: A complete Ethereum wallet implementation and utilities in JavaScript (and TypeScript).
- **web3modal**: It is an easy-to-use library to help developers add support for multiple providers in their apps with a simple customizable configuration.
- **ipfs-http-client**: Used to save files (images in our case) in a safe way and relies on hash code for verification.
- **@openzeppelin/contracts:** node dependency which contains standard token implementation for ERC-20 as well as some other utilities.
- **truffle-hdwallet-provider:** HD Wallet-enabled Web3 provider. Use it to sign transactions for addresses derived from a 12- or 24-word mnemonic.

Requirements for Phase 3:

- Application Smart Contract.
- Complete code implementation for the marketplace.
- Transactions using Custom ERC-20 based token. (UNIQ in our project).
- ERC-20 functionality and methods.
- Code implementation screenshots.
- Architectural diagrams.

User information:

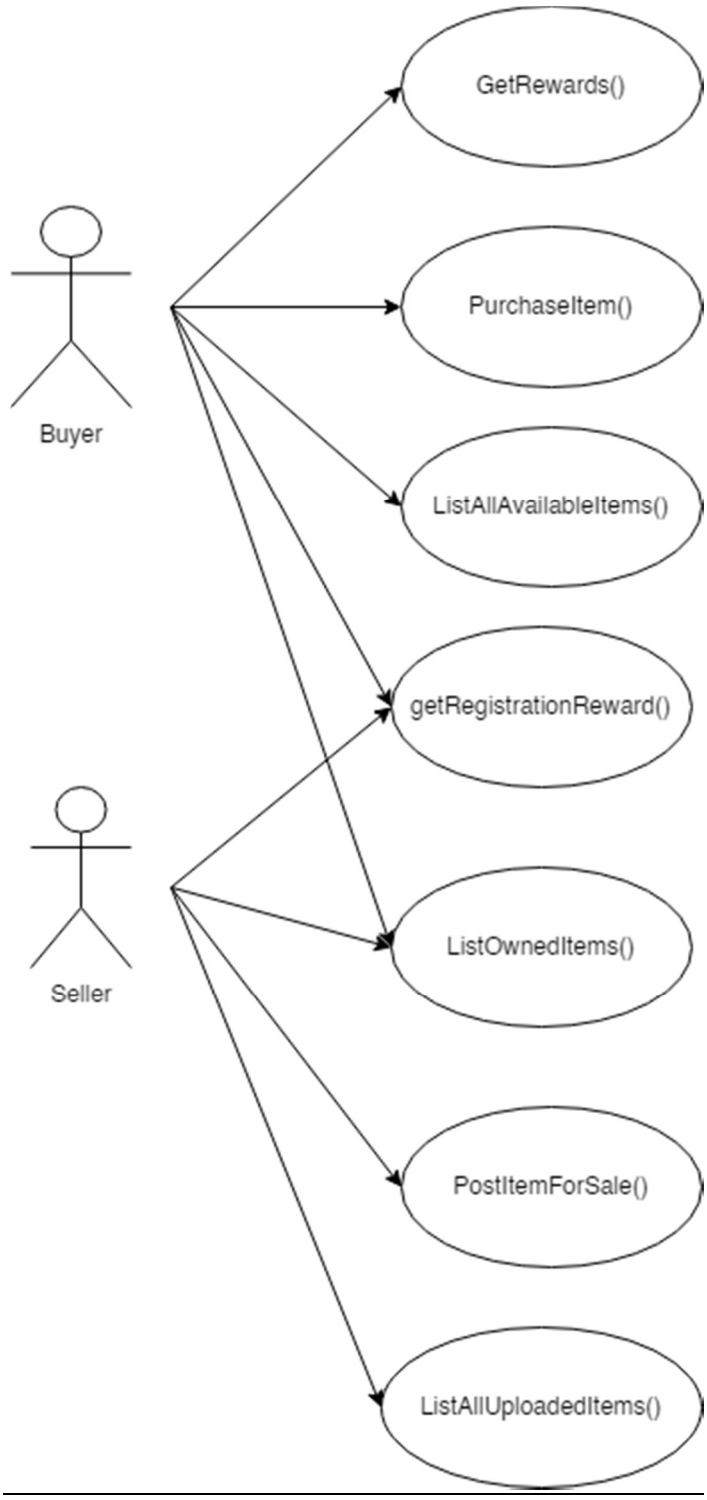
Our system has different kinds of users and they are interchangeable based on their roles:

- Buyer: Can buy any listed item and relist their bought item at different prices as well, also get rewarded by 1% for the purchases of more than 10 UNIQ.
- Seller: They can list a new unique item and buy any existing item to keep or resell.

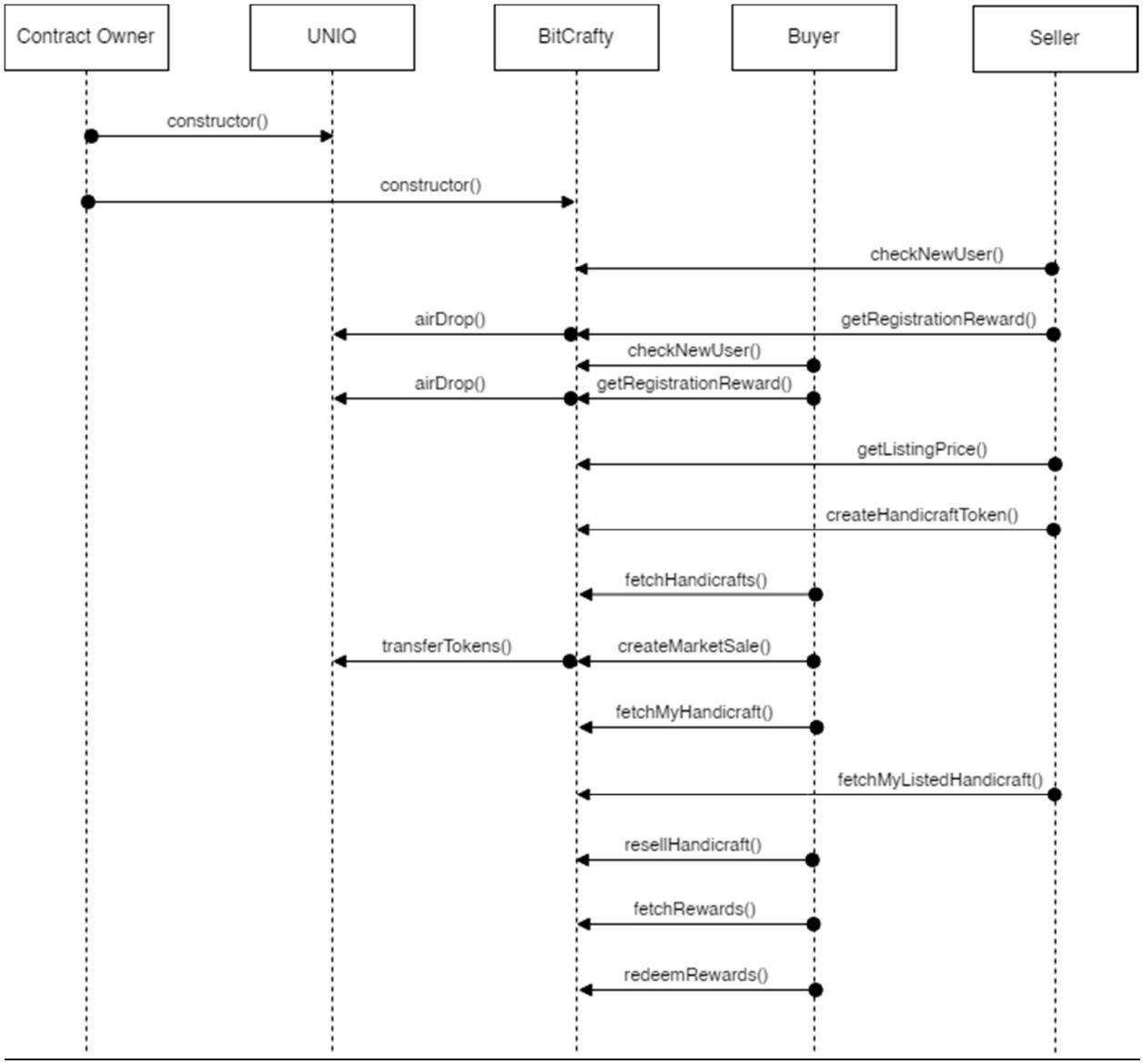
The Buyer and Seller roles are interchangeable based on their actions for a given transaction, a buyer can be a seller of their owned products and a seller can be a buyer of someone else's product. This is here to ensure a fair marketplace for everyone, not just some fixed sellers can monopolize the market.

Architecture Diagrams:

- **Use case diagram (UML):**

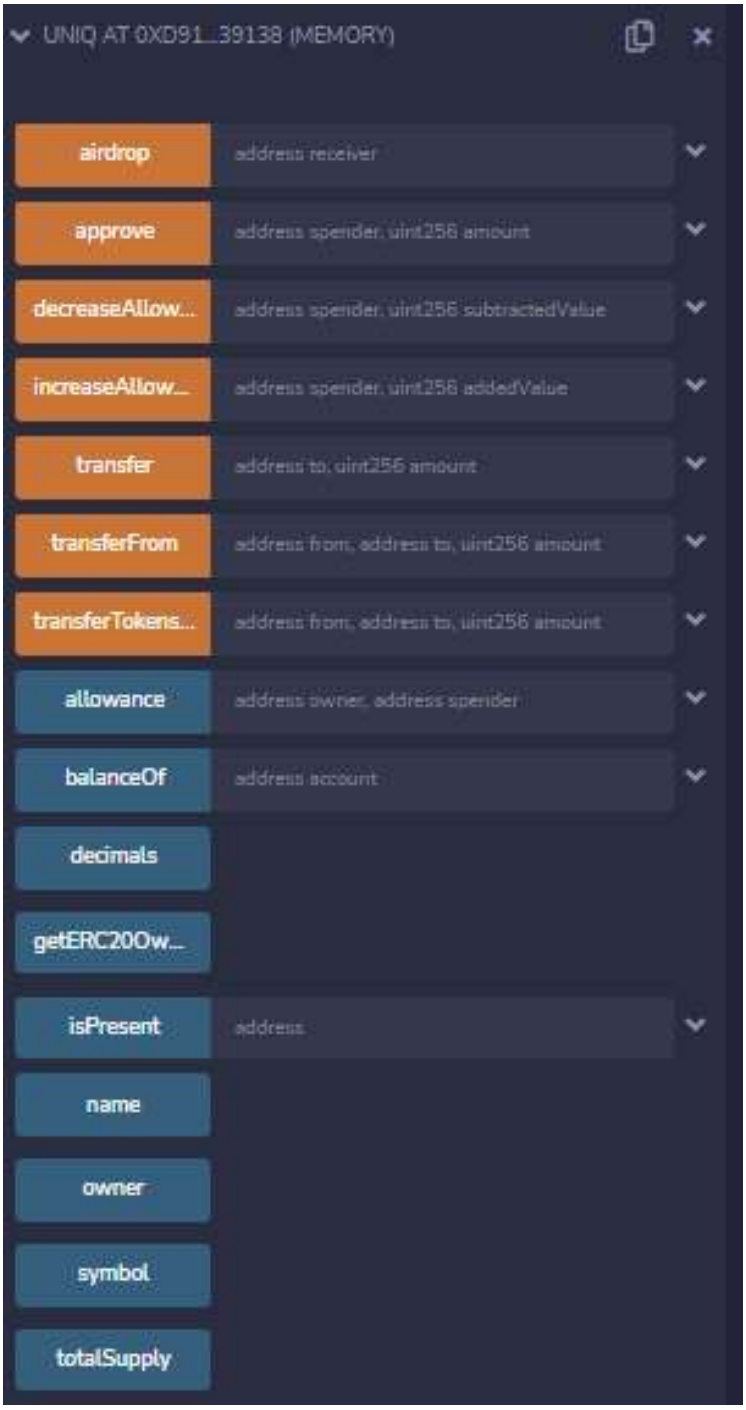


Sequence Diagram:

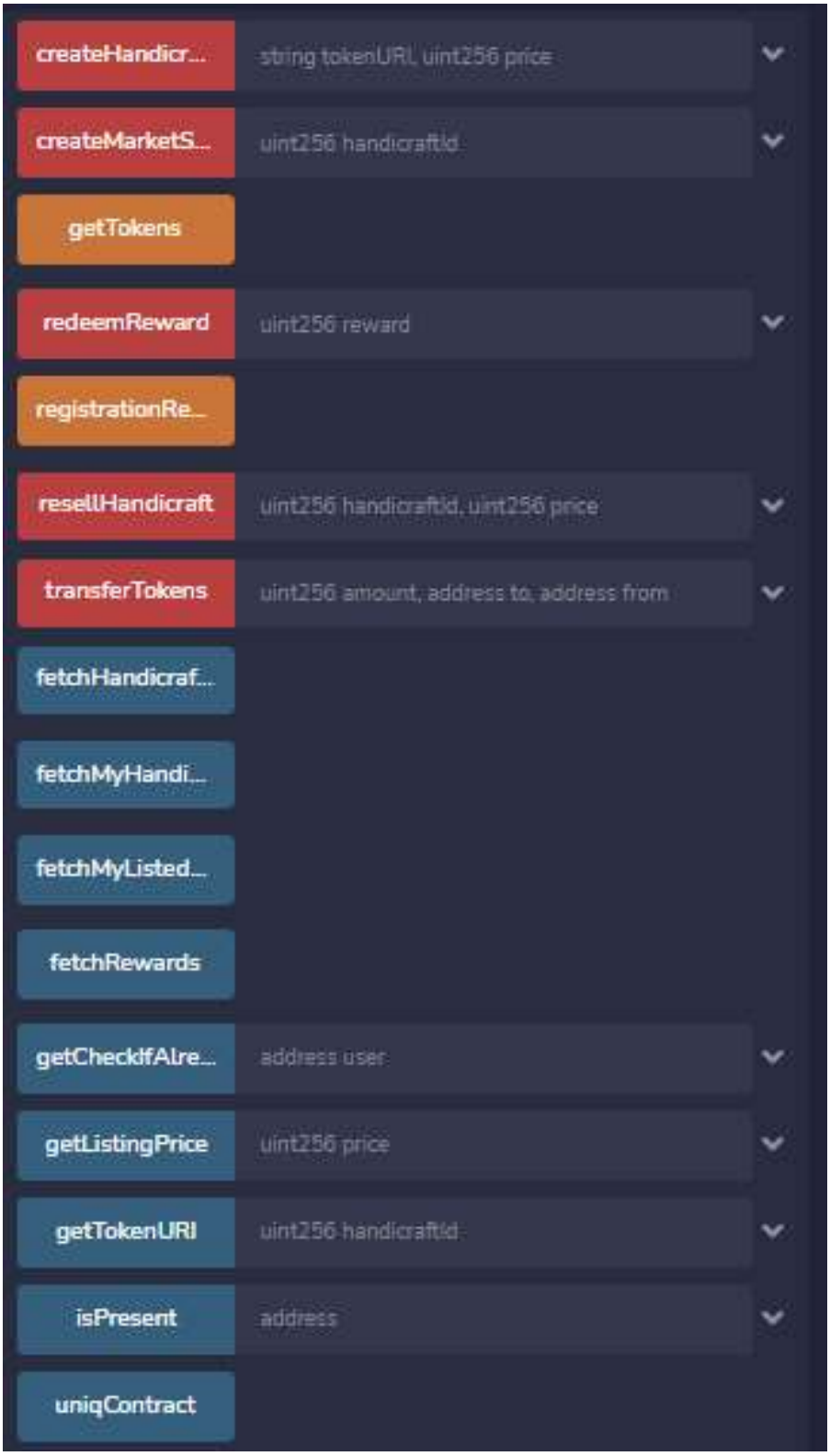


- **Contract Functions:**

UNIQ:



BitCrafty_Contract:



- **Contract Diagram:**

BitCrafty_Contract (import "./UNIQ.sol")

mapping(uint256 => Handicraft) idToHandicraft
mapping(uint256 => string) tokenUriMapping;
mapping(address => uint256) totalPurchaseValue;
address[] private listOfAddress;
address private owner;
UNIQ private uniq;
address public uniqContract;
mapping(address => bool) public isPresent;
struct Handicraft {uint256 handicraftId;address payable seller;address payable owner;uint256 price;bool sold;}
uint private handicraftsSold;
uint private handicraftIds;

event HandicraftCreated (uint256 indexed handicraftId,address seller,address owner,uint256 price,bool sold);
event Transfer(address seller, address owner, uint256 handicraftId);
event Mint(address to, uint256 handicraftId);

modifier priceGreaterThanZero(uint256 price) modifier onlyItemOwner(uint256 handicraftId) modifier amountSpentIsGreaterThan10() modifier ownerBalanceIsGreaterThanReward()
constructor(address token) function registrationReward() public function getCheckIfAlreadyHaveBonus(address user) public view returns (bool) function transferTokens(uint256 amount, address to, address from) public payable function createHandicraftToken(string memory tokenURI, uint256 price) payable (uint) function getTokenURI(uint256 handicraftId) view (string memory) function getListingPrice(uint256 price) pure (uint256) function createMarketSale(uint256 handicraftId) payable function createHandicraft(uint256 handicraftId,uint256 price) function resellHandicraft(uint256 handicraftId, uint256 price) payable function fetchHandicrafts() view (Handicraft[] memory) function fetchMyHandicrafts() view (Handicraft[] memory) function fetchMyListedHandicrafts() view (Handicraft[] memory) function fetchRewards() view (uint256) function redeemReward(uint reward) payable

UNIQ Contract:

UNIQ
address public owner;
constructor() ERC20("UNIQ", "UNIQ") function airdrop(address receiver) public

Deployment Script:

```
const BitCrafty_Contract = artifacts.require("BitCrafty_Contract");
const Uniq = artifacts.require("UNIQ");


module.exports = async function(deployer) {
    await deployer.deploy(Uniq);
    const token = await Uniq.deployed();
    await deployer.deploy(BitCrafty_Contract, token.address);
};
```

First the contract UNIQ is deployed to the network and truffle waits until the first contract is deployed.

Then the contract address of UNIQ is fetched and passed in parameterized constructor of BitCrafty_Contract as address of ERC 20 token.

_mint: This ERC20 function is being called in constructor where it creates a total supply of tokens for the contract owner.

```
function _mint(address account, uint256 amount) internal virtual {
    require(account != address(0), "ERC20: mint to the zero address");

    _beforeTokenTransfer(address(0), account, amount);

    _totalSupply += amount;
    _balances[account] += amount;
    emit Transfer(address(0), account, amount);

    _afterTokenTransfer(address(0), account, amount);
}
```

- **Quad Diagram:**

Use Case: Handicraft Market Place (BitCrafty) Problem Statement: People who create great artistic work are not paid fairly and their products are not considered unique.	**Issues with centralized marketplace:** ● The marketplaces are not centralized enough. They are usually localized and are confined in their specific places. ● There is a middleman usually interacting between buyer and seller, taking up a huge profit margin from the craftsmen. ● Fair Trading is not an option for the seller, without considering their opinion of pricing.
Proposed blockchain-based Solution: ● We are creating a worldwide blockchain based on a standard Ethereum token. ● Creating a non-fungible token which is transferred from seller to buyer, making the item/handicraft unique, and proposing the authenticity and ensuring ownership. ● All transactions recorded on blockchain for dispute resolution and for business analytics.	**Benefits:** ● All the users are treated equally be it buyer or seller i.e., all the rules in the contract are same for every actor. ● Fair Pricing is enabled when the seller is adding the handicraft to marketplace with the prices, they feel is appropriate ● This platform provides a seamless way to resell items, to gain profit from interested entities. ● The buyer is also eligible to get reward of 1% of the amount they have spent on our platform to encourage more usage of the platform.

Important Functions and Their Descriptions: (Added ERC-20 related functions)

Using ERC-20 standard to implement UNIQ:

1. A solidity file with the token name UNIQ.sol is created as shown:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract UNIQ is ERC20 {mapping(address => bool) public isPresent;
    address public owner;
    constructor() ERC20("UNIQ", "UNIQ") {owner = address(this);
        _mint(owner, 10000000000000000000000000000);}

    function airdrop(address receiver) public {_transfer(owner, receiver, 10000000000000000000);}

}
```

2. In the file ERC20 standard is imported from @openzeppline/contracts library and the contract file class is extended with ERC 20.
3. In order to instantiate name and symbol of new token ERC20 constructor is called with name and symbol in argument (both UNIQ in our case).
4. Owner is set to contract address along with that _mint function of ERC20 is called to invoke totalSupply internally and make contract the owner of the total supply.
5. Function airdrop is implemented to transfer 1000 UNIQ tokens to new user from contract owner.

Interaction of UNIQ contract with BitCrafty:

1. Because the marketplace is supposed to use new type of fungible token the constructor of BitCrafty must have a instance of UNIQ where tokens are minted to the owner already as shown:

```
constructor(address token){owner = msg.sender;
    uniq = UNIQ(token);
    uniqContract = token;}

function getTokens() public {registrationReward();}
```

2. The deployer gives the “token” address (which is deployed UNIQ address) as input.
3. The instance of UNIQ is obtained using that address and all the transaction-based functions like transfer will be called which is from UNIQ.

Important ERC-20 based functions:

- **registrationReward():**

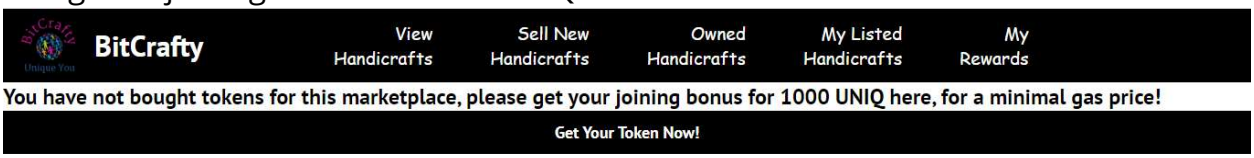
This is a key function when it comes to marketplace operation using custom tokens. It allows new users to get 1000 UNIQ tokens as a joining bonus. Just like any other economy every buyer/seller need some starting money and this provides that.

As shown below the following is new default landing page of the BitCrafty application where an internal call to function(shown below) happens which keeps track if a user has already availed the joining bonus:

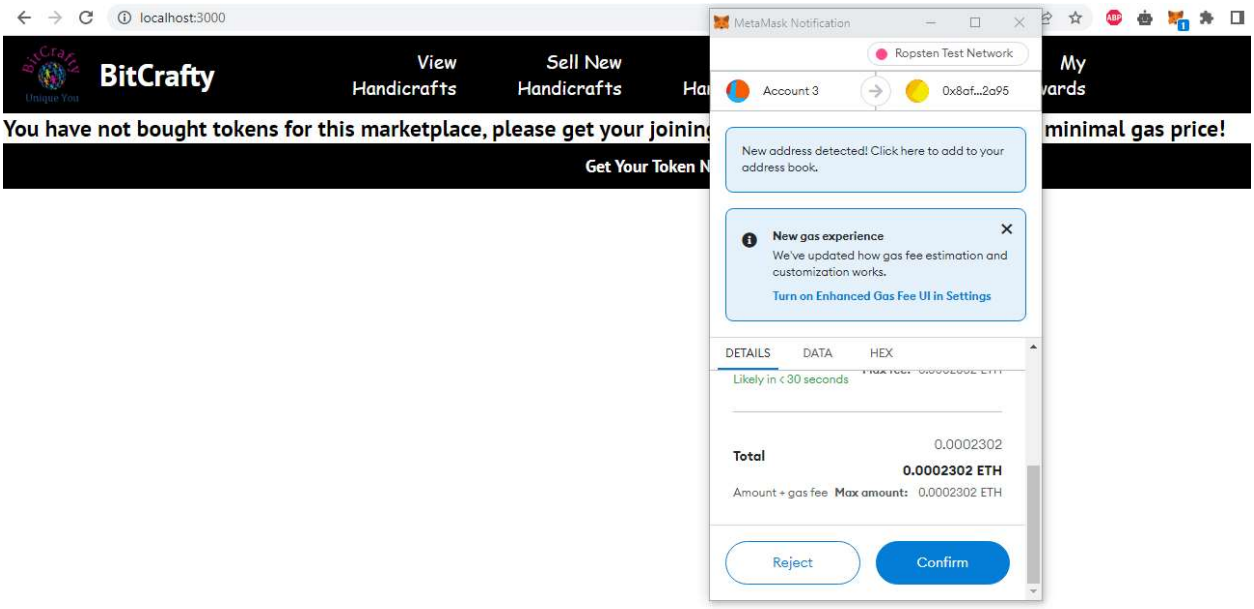
```
function getCheckIfAlreadyHaveBonus(address user) public view returns (bool)  {
    return isPresent[user];}
```

If a new user is detected the following page is shown where user can click the button

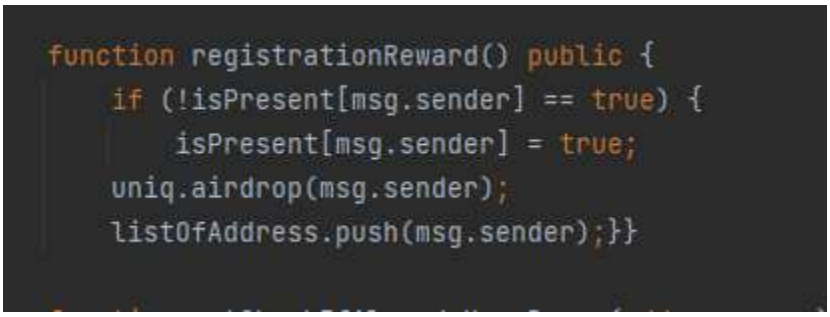
and get its joining bonus of 1000 UNIQ in their metamask wallet account.



After the user clicks the button, they are prompted to approve some ropsten gas price and avail bonus as shown:



The following function is invoked when the user clicks on “Get your Tokens”:



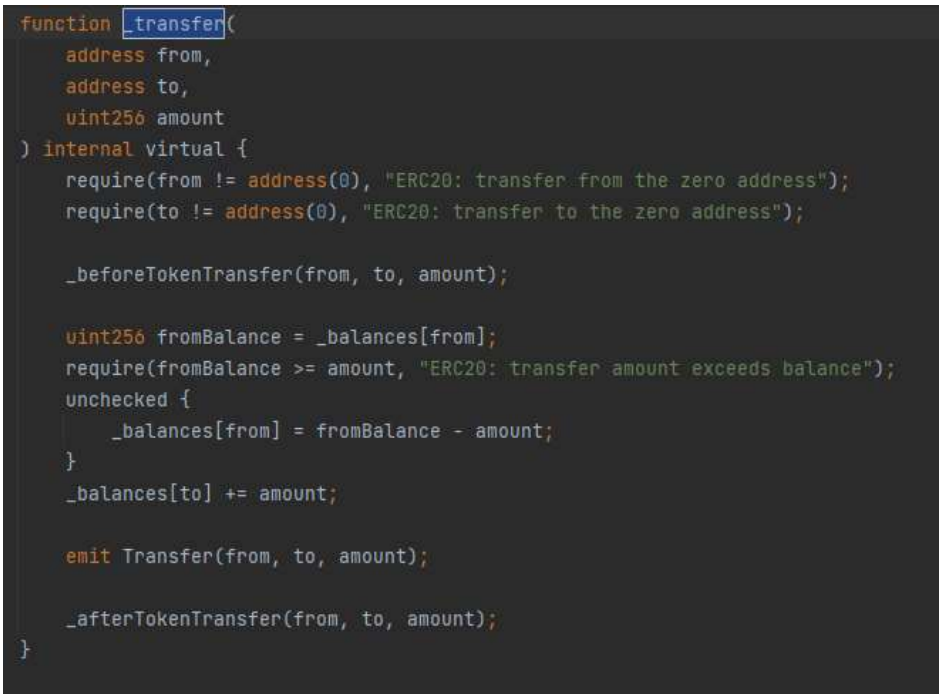
```
function registrationReward() public {
    if (!isPresent[msg.sender] == true) {
        isPresent[msg.sender] = true;
    uniq.airdrop(msg.sender);
    listOfAddress.push(msg.sender);}}
```

This function calls airdrop function of UNIQ and transfers 1000 UNIQ from contract to the user keeping total supply of UNIQ intact as instantiated only once.

The following function is being called which is internal to ERC20.

_transfer:

This function is called to transfer tokens from one address to another. In our case from owner to user during airdrop. We are not calling approve here since we don't need any approval from the smart contract itself.

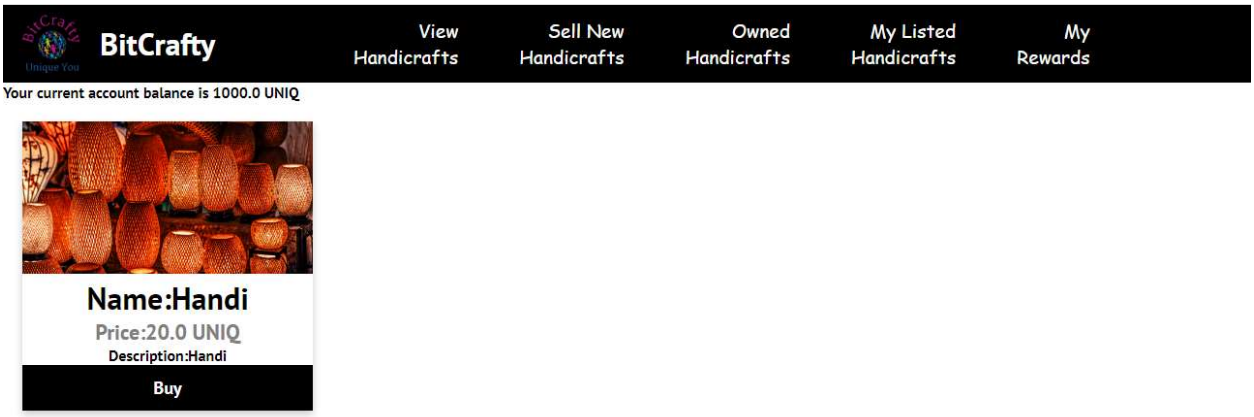


```
function _transfer(
    address from,
    address to,
    uint256 amount
) internal virtual {
    require(from != address(0), "ERC20: transfer from the zero address");
    require(to != address(0), "ERC20: transfer to the zero address");

    _beforeTokenTransfer(from, to, amount);

    uint256 fromBalance = _balances[from];
    require(fromBalance >= amount, "ERC20: transfer amount exceeds balance");
    unchecked {
        _balances[from] = fromBalance - amount;
    }
    _balances[to] += amount;

    emit Transfer(from, to, amount);

    _afterTokenTransfer(from, to, amount);
}
```

- **fetchHandicrafts():**

This is one of the key functions in the application which shows all the listed handicrafts from all the sellers. The user can buy any handicraft from this page if they have enough tokens and gas in their wallet to pay.

As shown below the following page is the default landing page for our marketplace and the function to fetch all the handicrafts gets invoked automatically as soon as the user visits the homepage of the application.




```
function fetchMyHandicrafts() public view returns (Handicraft[] memory) {
    uint totalItemCount = handicraftIds;
    uint itemCount = 0;
    uint currentIndex = 0;
    for (uint i = 0; i < totalItemCount; i++) {
        if (idToHandicraft[i + 1].owner == msg.sender) {
            itemCount += 1;
        }
    }
    Handicraft[] memory items = new Handicraft[](itemCount);
    for (uint i = 0; i < totalItemCount; i++) {
        if (idToHandicraft[i + 1].owner == msg.sender) {
            uint currentId = i + 1;
            Handicraft storage currentItem = idToHandicraft[currentId];
            items[currentIndex] = currentItem;
            currentIndex += 1;
        }
    }
    return items;
}
```

The front-end JavaScript file invokes the function in the contract and fetches all the listed handicrafts available for sale and then show handicraft image, price, and description as a card on the webpage. There is also option to buy any handicraft using Buy button.

1. In contract, first the number of unsold items is calculated by subtracting the number of all the handicrafts posted till date with number of handicrafts which are already sold.
2. All the unsold handcraft map is maintained in the map “idToHandicraft” which takes id as key and returns Handicraft Object as value.

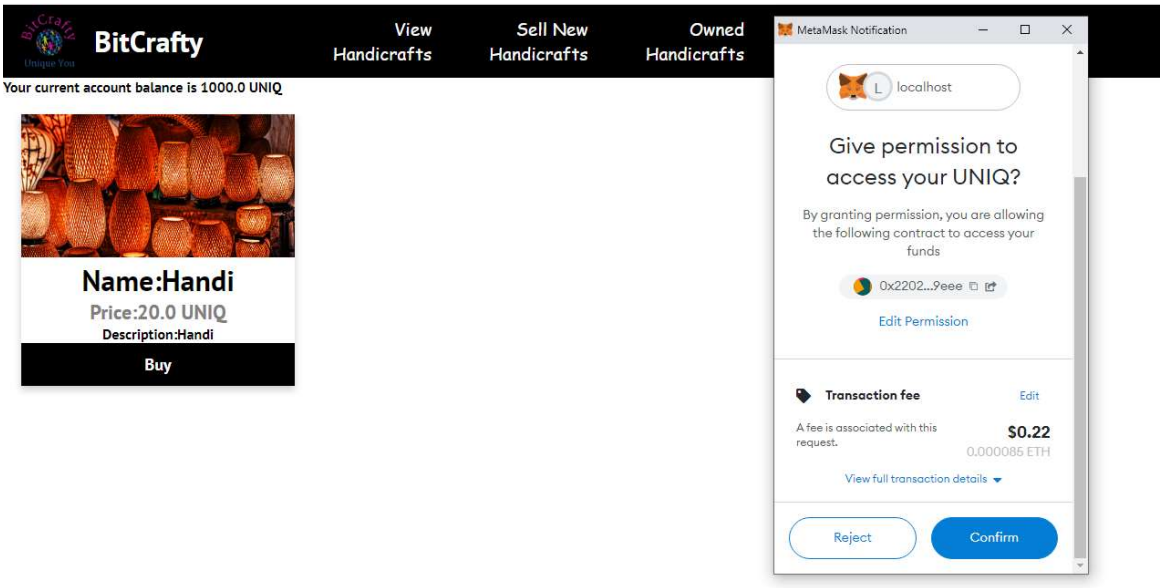
3. A loop for number obtained in step 1 above is executed and all the Handicrafts obtained is saved into an array.
4. The function checks if the handicrafts are owned by the contract and then the list of items is returned.

The array is then processed by webpage to show all the items.

- **createMarketSale(handicraftId):**

This is crucial feature of the marketplace where it lets user to buy one handicraft at a time and debiting transaction tokens from their wallet with totaling amount of product and gas price.

As shown below, as soon as a user clicks on the button buy for a handicraft the metamask window pops up and if user has enough money in wallet, they can click the button confirm and the ownership of the handicraft will be transferred to that account.



approve: This function is called from the session, to approve the smart contract to spend the approved amount on user's behalf. This will in turn increase the allowance between the user and the smart contract address.

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

```
function createMarketSale(uint256 handicraftId) public payable {uint price = idToHandicraft[handicraftId].price;
    address seller = idToHandicraft[handicraftId].seller;
    idToHandicraft[handicraftId].owner = payable(msg.sender);
    idToHandicraft[handicraftId].sold = true;
    idToHandicraft[handicraftId].seller = payable(address(0));
    handicraftsSold = handicraftsSold + 1;
    emit Transfer(address(this), msg.sender, handicraftId);
    transferTokens(price, seller, msg.sender);
    totalPurchaseValue[msg.sender] = totalPurchaseValue[msg.sender] + idToHandicraft[handicraftId].price;}
```

Once allowance amount between the user and smart contract is increased, the front-end JavaScript file invokes the function in the contract with handicraftId and price value and if all the validations are passed the transaction is successful and an entry on the block is created.

1. First the handicraft is fetched from the contract from the map idToHandicraft using handicraftId and the price and seller on the blockchain is fetched.
2. Then the validation checks if the price passed by the front end matches the price of handicraft matches on the map.
3. In the map for that particular handicraft the owner is reassigned to the msg.sender i.e.: the user buying the handicraft.
4. The flag sold is set to true and seller is set to none (address(0)). Then the counter to contain the number of sold handicrafts is incremented.
5. This transaction is emitted and a block in the chain is created stating the transaction with owner of handicraft id to be the current user.
6. The desired amount is then transferred to the seller from buyer using transferTokens() function.
7. Then the total purchase amount of that user is updated to be used for reward function which is discussed later.

transferTokens() function calls the following ERC20 functions and they work as follows:

- **transferFrom:** This is used to transfer token from one user to another which also involves approval.

```
function transferFrom(
    address from,
    address to,
    uint256 amount
) public virtual override returns (bool) {
    address spender = _msgSender();
    _spendAllowance(from, spender, amount);
    _transfer(from, to, amount);
    return true;
}
```

UNIQ Token balance:

Token balance will be displayed in the frontend by invoking **balanceOf(address account)** function by passing the current user address.

balanceOf(address account):

Returns the amount of tokens owned by the account.

```
let balanceOfUser = await tokenContract.balanceOf(currentAddress)
let balanceInUniq = ethers.utils.formatUnits(balanceOfUser.toString(), 'ether')
setBalance(balanceInUniq.toString())
```

The owned handicrafts can now be seen in the “Owned Handicraft” tab on the navigation bar as shown with an option to resell at same or different price:

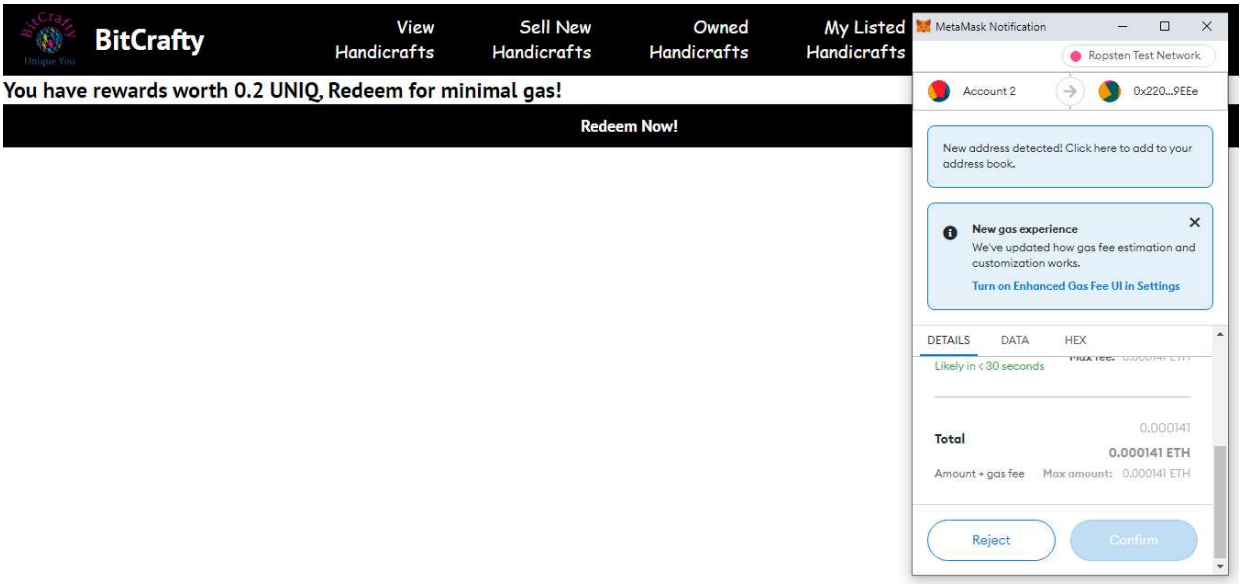
- **fetchRewards() and redeemReward(uint reward):**

This is an additional feature of the marketplace giving them 1% reward for purchases more than 10 UNIQ to encourage more user participation.

As shown below the “My Rewards” tab of the marketplace will be populated with the reward amount and option to redeem if the user is eligible for the reward.

As soon as the user clicks on the button “Redeem Now!” a metamask window will pop up to deduct the gas charges in order to process transaction for the reward redemption.

As shown below the following logic is implemented in the contract and the window to redeem reward:



As soon as a user goes to “My Rewards” window the fetchrewards() function is invoked and if total amount of purchase made by user is more than 10 UNIQ the 1% reward is calculated and shown on screen.

```
function fetchRewards() public view returns (uint256) {if (totalPurchaseValue[msg.sender] >= 10) {uint reward = (totalPurchaseValue[msg.sender] * 1) / 100;
    return reward;}
    return 0;}

function redeemReward(uint reward) public amountSpentIsGreaterThan10 payable {
    require(uniq.balanceOf(address(this)) > msg.value, "Smart contract balance should at least be greater than reward ");
    uniq.approve(address(this), reward);
    transferTokens(reward, msg.sender, address(this));
    totalPurchaseValue[msg.sender] = 0;}
```

After the user invokes the function to redeem rewards with the reward amount the amount is transferred to the user and then total purchase amount is set to 0 to ensure that the user does not claim reward multiple times for same purchase.

Contract Description:

mapping(uint256 => Handicraft) idToHandicraft : Map used to store all the handicrafts with handicraftId as key and Handicraft Object as value.

mapping(uint256 => string) tokenUriMapping : Map used to store all the tokenURI with handicraftId as key and URI string as value.

mapping(address => uint256) totalPurchaseValue: Map used to store total purchase amount of a user with user as key and amount integer as value.

struct Handicraft {uint256 handicraftId;address payable seller;address payable owner;uint256 price;bool sold;} : Handicraft object

uint private handicraftsSold: Counter to count number of sold handicrafts.

uint private handicraftIds: Counter to count number of listed handicrafts.

event HandicraftCreated (uint256 indexed handicraftId,address seller,address owner,uint256 price,bool sold): Event to be stored on the blockchain with all the needed data for the handicraft object.

event Transfer(address seller, address owner, uint256 handicraftId): event to transfer ownership of the NFT from seller to owner with handicraftId and record that transaction on blockchain.

event Mint(address to, uint256 handicraftId): event to initialize ownership of the NFT to seller with handicraftId and record that transaction on blockchain.

modifier priceGreaterThanZero(uint256 price): Modifier to check if the price of listed item is more than 0.

modifier onlyItemOwner(uint256 handicraftId): Modifier to ensure only an item owner can post and item to resell.

modifier amountSpentIsGreaterThan10(): Modifier to ensure that amount spent is more than 10UNIQ to be eligible for reward.

function createHandicraftToken(string memory tokenURI, uint256 price) payable (uint): Creates a token based on tokenURI and price and calls mint function to persist that transaction into blockchain.

function getTokenURI(uint256 handicraftId) view (string memory): Function to get handicraftId as input and return tokenURI as output querying the tokenURI map.

function getListingPrice(uint256 price) pure (uint256): Function to calculate and return the listing price of a Handicraft based on the selling price i.e.: 2% of selling price.

function createMarketSale(uint256 handicraftId) payable: Function to transfer ownership of sold handicraft by calling emit Transfer and record the transaction on blockchain.

function resellHandicraft(uint256 handicraftId, uint256 price) payable: Function to relist an owned handicraft for a different price including the listing price from user.

function fetchHandicrafts() view (Handicraft[] memory): Function to return list of all handicrafts listed by all the users on marketplace which are available to buy.

function fetchMyHandicrafts() view (Handicraft[] memory): Function to return list of all handicrafts owned by the user.

function fetchMyListedHandicrafts() view (Handicraft[] memory): Function to return list of all handicrafts listed by user on marketplace which are available to buy.

function fetchRewards() view (uint256): Function to fetch the amount of reward a user is eligible based on his purchases.

function redeemReward(uint reward) payable: Function to credit reward amount to the user wallet.

function transferTokens(uint256 amount, address to, address from) public payable: Invokes transferFrom function of ERC20 to transfer amount from address to “to” address.

ERC-20 (UNIQ) Solidity Code:

```
pragma solidity ^0.8.4;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract UNIQ is ERC20 {mapping(address => bool) public isPresent;
  address public owner;
  constructor() ERC20("UNIQ", "UNIQ") {owner = address(this);
     _mint(owner, 100000000000000000000000000000);}

  function airdrop(address receiver) public {_transfer(owner, receiver, 1000000000000000000000);}

  function getERC20Owner() view public returns (address) {return owner;}

  function transferTokensTo(address from, address to, uint256 amount) public {_transfer(from, to, amount);}
```

OpenZeppelin ERC20.sol

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v4.6.0) (token/ERC20/ERC20.sol)

pragma solidity ^0.8.0;

import "./IERC20.sol";
import "./extensions/IERC20Metadata.sol";
import "../../utils/Context.sol";

/**
* @dev Implementation of the {IERC20} interface.
*
* This implementation is agnostic to the way tokens are created. This means
* that a supply mechanism has to be added in a derived contract using {_mint}.
* For a generic mechanism see {ERC20PresetMinterPauser}.
*
* TIP: For a detailed writeup see our guide
* https://forum.zeppelin.solutions/t/how-to-implement-erc20-supply-mechanisms/226[How
* to implement supply mechanisms].
*
* We have followed general OpenZeppelin Contracts guidelines: functions revert
* instead returning `false` on failure. This behavior is nonetheless
* conventional and does not conflict with the expectations of ERC20
* applications.
```

```
*
* Additionally, an {Approval} event is emitted on calls to {transferFrom}.
* This allows applications to reconstruct the allowance for all accounts just
* by listening to said events. Other implementations of the EIP may not emit
* these events, as it isn't required by the specification.
*
* Finally, the non-standard {decreaseAllowance} and {increaseAllowance}
* functions have been added to mitigate the well-known issues around setting
* allowances. See {IERC20-approve}.
*/
contract ERC20 is Context, IERC20, IERC20Metadata {
  mapping(address => uint256) private _balances;

  mapping(address => mapping(address => uint256)) private _allowances;

  uint256 private _totalSupply;

  string private _name;
  string private _symbol;

  /**
   * @dev Sets the values for {name} and {symbol}.
   *
   * The default value of {decimals} is 18. To select a different value for
   * {decimals} you should overload it.
   *
   * All two of these values are immutable: they can only be set once during
   * construction.
   */
  constructor(string memory name_, string memory symbol_) {
      _name = name_;
      _symbol = symbol_;
  }

  /**
   * @dev Returns the name of the token.
   */
  function name() public view virtual override returns (string memory) {
      return _name;
  }

  /**
   * @dev Returns the symbol of the token, usually a shorter version of the
   * name.
   */
  function symbol() public view virtual override returns (string memory) {
      return _symbol;
  }
```

```
/**
 * @dev Returns the number of decimals used to get its user representation.
 * For example, if `decimals` equals `2`, a balance of `505` tokens should
 * be displayed to a user as `5.05` (`505 / 10 ** 2`).
 *
 * Tokens usually opt for a value of 18, imitating the relationship between
 * Ether and Wei. This is the value {ERC20} uses, unless this function is
 * overridden;
 *
 * NOTE: This information is only used for _display_ purposes: it in
 * no way affects any of the arithmetic of the contract, including
 * {IERC20-balanceOf} and {IERC20-transfer}.
 */
function decimals() public view virtual override returns (uint8) {
    return 18;
}

/**
 * @dev See {IERC20-totalSupply}.
 */
function totalSupply() public view virtual override returns (uint256) {
    return _totalSupply;
}

/**
 * @dev See {IERC20-balanceOf}.
 */
function balanceOf(address account) public view virtual override returns (uint256) {
    return _balances[account];
}

/**
 * @dev See {IERC20-transfer}.
 *
 * Requirements:
 *
 * - `to` cannot be the zero address.
 * - the caller must have a balance of at least `amount`.
 */
function transfer(address to, uint256 amount) public virtual override returns (bool) {
    address owner = _msgSender();
    _transfer(owner, to, amount);
    return true;
}

/**
 * @dev See {IERC20-allowance}.
 */
function allowance(address owner, address spender) public view virtual override returns (uint256) {
    return _allowances[owner][spender];
```

```
}

/**
 * @dev See {IERC20-approve}.
 *
 * NOTE: If `amount` is the maximum `uint256`, the allowance is not updated on
 * `transferFrom`. This is semantically equivalent to an infinite approval.
 *
 * Requirements:
 *
 * - `spender` cannot be the zero address.
 */
function approve(address spender, uint256 amount) public virtual override returns (bool) {
    address owner = _msgSender();
    _approve(owner, spender, amount);
    return true;
}

/**
 * @dev See {IERC20-transferFrom}.
 *
 * Emits an {Approval} event indicating the updated allowance. This is not
 * required by the EIP. See the note at the beginning of {ERC20}.
 *
 * NOTE: Does not update the allowance if the current allowance
 * is the maximum `uint256`.
 *
 * Requirements:
 *
 * - `from` and `to` cannot be the zero address.
 * - `from` must have a balance of at least `amount`.
 * - the caller must have allowance for ``from``'s tokens of at least
 * `amount`.
 */
function transferFrom(
    address from,
    address to,
    uint256 amount
) public virtual override returns (bool) {
    address spender = _msgSender();
    _spendAllowance(from, spender, amount);
    _transfer(from, to, amount);
    return true;
}

/**
 * @dev Atomically increases the allowance granted to `spender` by the caller.
 *
 * This is an alternative to {approve} that can be used as a mitigation for
 * problems described in {IERC20-approve}.
 *
```

```
     * Emits an {Approval} event indicating the updated allowance.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool) {
        address owner = _msgSender();
        _approve(owner, spender, allowance(owner, spender) + addedValue);
        return true;
    }

    /**
     * @dev Atomically decreases the allowance granted to `spender` by the caller.
     *
     * This is an alternative to {approve} that can be used as a mitigation for
     * problems described in {IERC20-approve}.
     *
     * Emits an {Approval} event indicating the updated allowance.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     * - `spender` must have allowance for the caller of at least
     * `subtractedValue`.
     */
    function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool) {
        address owner = _msgSender();
        uint256 currentAllowance = allowance(owner, spender);
        require(currentAllowance >= subtractedValue, "ERC20: decreased allowance below zero");
        unchecked {
            _approve(owner, spender, currentAllowance - subtractedValue);
        }

        return true;
    }

    /**
     * @dev Moves `amount` of tokens from `sender` to `recipient`.
     *
     * This internal function is equivalent to {transfer}, and can be used to
     * e.g. implement automatic token fees, slashing mechanisms, etc.
     *
     * Emits a {Transfer} event.
     *
     * Requirements:
     *
     * - `from` cannot be the zero address.
     * - `to` cannot be the zero address.
     * - `from` must have a balance of at least `amount`.
     */
```

```
function _transfer(
    address from,
    address to,
    uint256 amount
) internal virtual {
    require(from != address(0), "ERC20: transfer from the zero address");
    require(to != address(0), "ERC20: transfer to the zero address");

    _beforeTokenTransfer(from, to, amount);

    uint256 fromBalance = _balances[from];
    require(fromBalance >= amount, "ERC20: transfer amount exceeds balance");
    unchecked {
        _balances[from] = fromBalance - amount;
    }
    _balances[to] += amount;

    emit Transfer(from, to, amount);

    _afterTokenTransfer(from, to, amount);
}

/** @dev Creates `amount` tokens and assigns them to `account`, increasing
 * the total supply.
 *
 * Emits a {Transfer} event with `from` set to the zero address.
 *
 * Requirements:
 *
 * - `account` cannot be the zero address.
 */
function _mint(address account, uint256 amount) internal virtual {
    require(account != address(0), "ERC20: mint to the zero address");

    _beforeTokenTransfer(address(0), account, amount);

    _totalSupply += amount;
    _balances[account] += amount;
    emit Transfer(address(0), account, amount);

    _afterTokenTransfer(address(0), account, amount);
}

/**
 * @dev Destroys `amount` tokens from `account`, reducing the
 * total supply.
 *
 * Emits a {Transfer} event with `to` set to the zero address.
 *
```

```
     * Requirements:
     *
     * - `account` cannot be the zero address.
     * - `account` must have at least `amount` tokens.
     */
    function _burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: burn from the zero address");

        _beforeTokenTransfer(account, address(0), amount);

        uint256 accountBalance = _balances[account];
        require(accountBalance >= amount, "ERC20: burn amount exceeds balance");
        unchecked {
            _balances[account] = accountBalance - amount;
        }
        _totalSupply -= amount;

        emit Transfer(account, address(0), amount);

        _afterTokenTransfer(account, address(0), amount);
    }

    /**
     * @dev Sets `amount` as the allowance of `spender` over the `owner` s tokens.
     *
     * This internal function is equivalent to `approve`, and can be used to
     * e.g. set automatic allowances for certain subsystems, etc.
     *
     * Emits an {Approval} event.
     *
     * Requirements:
     *
     * - `owner` cannot be the zero address.
     * - `spender` cannot be the zero address.
     */
    function _approve(
        address owner,
        address spender,
        uint256 amount
    ) internal virtual {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }

    /**
     * @dev Updates `owner` s allowance for `spender` based on spent `amount`.
```

```
     *
     * Does not update the allowance amount in case of infinite allowance.
     * Revert if not enough allowance is available.
     *
     * Might emit an {Approval} event.
     */
    function _spendAllowance(
        address owner,
        address spender,
        uint256 amount
    ) internal virtual {
        uint256 currentAllowance = allowance(owner, spender);
        if (currentAllowance != type(uint256).max) {
            require(currentAllowance >= amount, "ERC20: insufficient allowance");
            unchecked {
                _approve(owner, spender, currentAllowance - amount);
            }
        }
    }

    /**
     * @dev Hook that is called before any transfer of tokens. This includes
     * minting and burning.
     *
     * Calling conditions:
     *
     * - when `from` and `to` are both non-zero, `amount` of ``from``'s tokens
     * will be transferred to `to`.
     * - when `from` is zero, `amount` tokens will be minted for `to`.
     * - when `to` is zero, `amount` of ``from``'s tokens will be burned.
     * - `from` and `to` are never both zero.
     *
     * To learn more about hooks, head to xref:ROOT:extending-contracts.adoc#using-hooks[Using Hooks].
     */
    function _beforeTokenTransfer(
        address from,
        address to,
        uint256 amount
    ) internal virtual {}

    /**
     * @dev Hook that is called after any transfer of tokens. This includes
     * minting and burning.
     *
     * Calling conditions:
     *
     * - when `from` and `to` are both non-zero, `amount` of ``from``'s tokens
     * has been transferred to `to`.
     * - when `from` is zero, `amount` tokens have been minted for `to`.
     * - when `to` is zero, `amount` of ``from``'s tokens have been burned.
     * - `from` and `to` are never both zero.
```

```
    *
    * To learn more about hooks, head to xref:ROOT:extending-contracts.adoc#using-hooks[Using Hooks].
    */
   function _afterTokenTransfer(
      address from,
      address to,
      uint256 amount
   ) internal virtual {}
}
```

Full Marketplace Solidity Code:

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.4;

contract BitCrafty_Contract {


        mapping(uint256 => Handicraft) private idToHandicraft;

        mapping(uint256 => string) private tokenUriMapping;


        mapping(address => uint256) private totalPurchaseValue;


        struct Handicraft {

        uint256 handicraftId;

        address payable seller;

        address payable owner;

        uint256 price;

        bool sold;
```

```
}

event HandicraftCreated (
uint256 indexed handicraftId,
address seller,
address owner,
uint256 price,
bool sold
);

event Transfer(address seller, address owner, uint256 handicraftId);
event Mint(address to, uint256 handicraftId);
uint private handicraftsSold;
uint private handicraftIds;

modifier priceGreaterThanZero(uint256 price){
require(price > 0, "Price must be at least 1 wei");
_;
}

modifier onlyItemOwner(uint256 handicraftId){
```

```
        require(idToHandicraft[handicraftId].owner == msg.sender, "Only item owner can perform this operation");

        _;

        }

        modifier amountSpentIsGreaterThan10{

        require(totalPurchaseValue[msg.sender] >= 10000000000000000000, "Total purchase value should atleast be greater than 10 Ethers");

        _;

        }



        modifier ownerBalanceIsGreaterThanReward{

        require(address(this).balance > msg.value, "Smart contract balance should at least be greater than reward ");

        _;

        }



        function createHandicraftToken(string memory tokenURI, uint256 price) public payable returns (uint) {

        handicraftIds = handicraftIds+1;

        emit Mint(msg.sender, handicraftIds);

        tokenUriMapping[handicraftIds] = tokenURI;

        createHandicraft(handicraftIds, price);

        return handicraftIds;

        }
```

```
    function createHandicraft(
    uint256 handicraftId,
    uint256 price
    ) private priceGreaterThanZero(price) {
idToHandicraft[handicraftId] = Handicraft(
    handicraftId,
    payable(msg.sender),
    payable(address(this)),
    price,
    false
    );
    emit Transfer(msg.sender, address(this), handicraftId);
    emit HandicraftCreated(
    handicraftId,
    msg.sender,
    address(this),
    price,
    false
    );
    }
```

```
    function getTokenURI(uint256 handicraftId) public view returns (string memory) {
    return tokenUriMapping[handicraftId];
    }

    function getListingPrice(uint256 price) public pure returns (uint256) {
    return (price * 2) / 100;
    }

    function createMarketSale(
    uint256 handicraftId
    ) public payable {
    uint price = idToHandicraft[handicraftId].price;
    address seller = idToHandicraft[handicraftId].seller;
    require(msg.value == price, "Please submit the asking price in order to complete the purchase");
    idToHandicraft[handicraftId].owner = payable(msg.sender);
    idToHandicraft[handicraftId].sold = true;
    idToHandicraft[handicraftId].seller = payable(address(0));
    handicraftsSold= handicraftsSold+1;
    emit Transfer(address(this), msg.sender, handicraftId);
    payable(seller).transfer(msg.value);
    totalPurchaseValue[msg.sender] = totalPurchaseValue[msg.sender] + idToHandicraft[handicraftId].price;
```

```
    }

    function resellHandicraft(uint256 handicraftId, uint256 price) public payable
onlyItemOwner(handicraftId) {
    idToHandicraft[handicraftId].price = price;
    idToHandicraft[handicraftId].sold = false;
    idToHandicraft[handicraftId].owner = payable(address(this));
    idToHandicraft[handicraftId].seller = payable(msg.sender);
    emit Transfer(msg.sender, address(this), handicraftId);
    handicraftsSold= handicraftsSold-1;
    }

    function fetchHandicrafts() public view returns (Handicraft[] memory) {
    uint itemCount = handicraftIds;
    uint unsoldItemCount = handicraftIds - handicraftsSold;
    uint currentIndex = 0;
    Handicraft[] memory items = new Handicraft[](unsoldItemCount);
    for (uint i = 0; i < itemCount; i++) {
    if (idToHandicraft[i + 1].owner == address(this)) {
            uint currentId = i + 1;
        Handicraft storage currentItem = idToHandicraft[currentId];
        items[currentIndex] = currentItem;
```

```
    currentIndex += 1;
}
}
return items;
}

function fetchMyHandicrafts() public view returns (Handicraft[] memory) {
uint totalItemCount = handicraftIds;
uint itemCount = 0;
uint currentIndex = 0;
for (uint i = 0; i < totalItemCount; i++) {
if (idToHandicraft[i + 1].owner == msg.sender) {
    itemCount += 1;
}
}
Handicraft[] memory items = new Handicraft[](itemCount);
for (uint i = 0; i < totalItemCount; i++) {
if (idToHandicraft[i + 1].owner == msg.sender) {
        uint currentId = i + 1;
    Handicraft storage currentItem = idToHandicraft[currentId];
    items[currentIndex] = currentItem;
    currentIndex += 1;
```

```
}
}
return items;
}

function fetchMyListedHandicrafts() public view returns (Handicraft[] memory) {
uint totalItemCount = handicraftIds;
uint itemCount = 0;
uint currentIndex = 0;
for (uint i = 0; i < totalItemCount; i++) {
if (idToHandicraft[i + 1].seller == msg.sender) {
    itemCount += 1;
}
}
Handicraft[] memory items = new Handicraft[](itemCount);
for (uint i = 0; i < totalItemCount; i++) {
if (idToHandicraft[i + 1].seller == msg.sender) {
        uint currentId = i + 1;
    Handicraft storage currentItem = idToHandicraft[currentId];
    items[currentIndex] = currentItem;
    currentIndex += 1;
}
```

```
        }
          return items;
        }

        function fetchRewards() public view returns (uint256) {
        if (totalPurchaseValue[msg.sender] >= 10) {
        uint reward = (totalPurchaseValue[msg.sender] * 1) / 100;
        return reward;
        }
        return 0;
        }

        function redeemReward(uint reward) public amountSpentIsGreaterThan10
ownerBalanceIsGreaterThanReward payable{
      payable(msg.sender).transfer(reward);
      totalPurchaseValue[msg.sender] = 0;
        }

}
```

References:

1. Web3: https://web3js.readthedocs.io/en/v1.7.1/getting-started.html
2. Solidity: https://docs.soliditylang.org/en/v0.8.13/introduction-to-smart-contracts.html

3. React: Tutorial: Intro to React – React (reactjs.org)
4. React Tutorial (w3schools.com)
5. Providers — ethers.js 4.0.0 documentation
6. https://docs.openzeppelin.com/contracts/4.x/api/token/erc20
7. https://ethereum.org/en/developers/tutorials/understand-the-erc-20-token-smart-contract/