University of Basel

52354-01 Smart Contracts and Decentralized Finance

Token Basket Smart Contract

Group 16:

Abeeshan Rasadurai 2018-054-601

Elena Parshina 22-061-485

Sebastian Strickfaden 5102336 (University of Freiburg)

Yvan Tercier 95-106-555

Professor:

Dr. Fabian Schär

Submission date: 18.11.2022

Table of contents

1. Introduction. Smart contract - Token basket

This paper gives you an overview report of our smart contract solution that we have built. Our general idea is the creation of a token basket built on the Ethereum blockchain. This allows our customers to diversify their token holdings in a single basket. Making it easier, cheaper,and less time intense than buying each of these tokens separately. The plan of including multiple assets in a basket is already done in the stock market with ETFs with great success, allowing customers to dollar cost average into these ETFs, with predetermined monthly orders. In order to achieve this in an efficient manner we interacted with the decentralized exchange uniswap. We imported the code to interact with uniswap in order to be able to exchange assets and build our smart contract using it.

Two tokens (USDC/USDT) are included in our basket. The basket would be perfect for someone who wants to dollar cost average in multiple tokens because they would save on gas fees (USDC and USDT are used here as an example). There are no limits on how many tokens can be assigned to one basket. We designed the tokenBasket contract in the way that each customer can add themselves as a customer, deposit funds in ETH to the individual balance in the smart contract and then swap the funds added for 50% USDC and 50% USDT tokens. The tokens are being held and stored for the customer inside the contract. The customer can at any time swap USDC and USDT tokens back to ETH and then withdraw them to their wallet (no withdrawal of the tokens). The owner of the contract (added as the first customer) receives 1% fee of the amount sent in ETH every time a customer deposits funds to the contract.

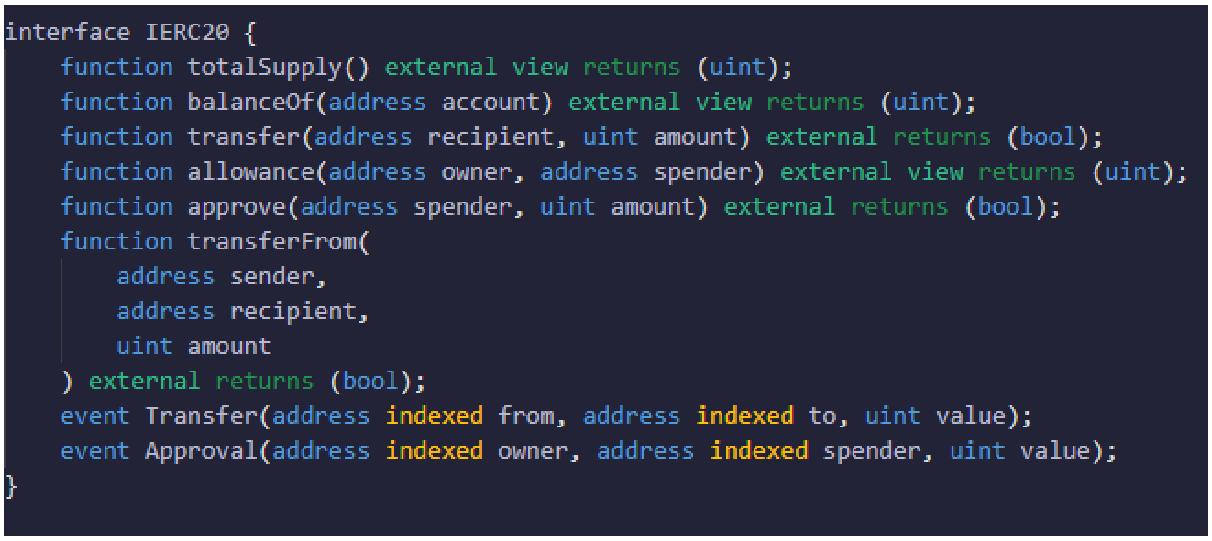
Future developments to group the orders and make one daily exchange for all the customers will lead to a further reduction in gas and exchange fees.

2. Uniswap - Swap functions

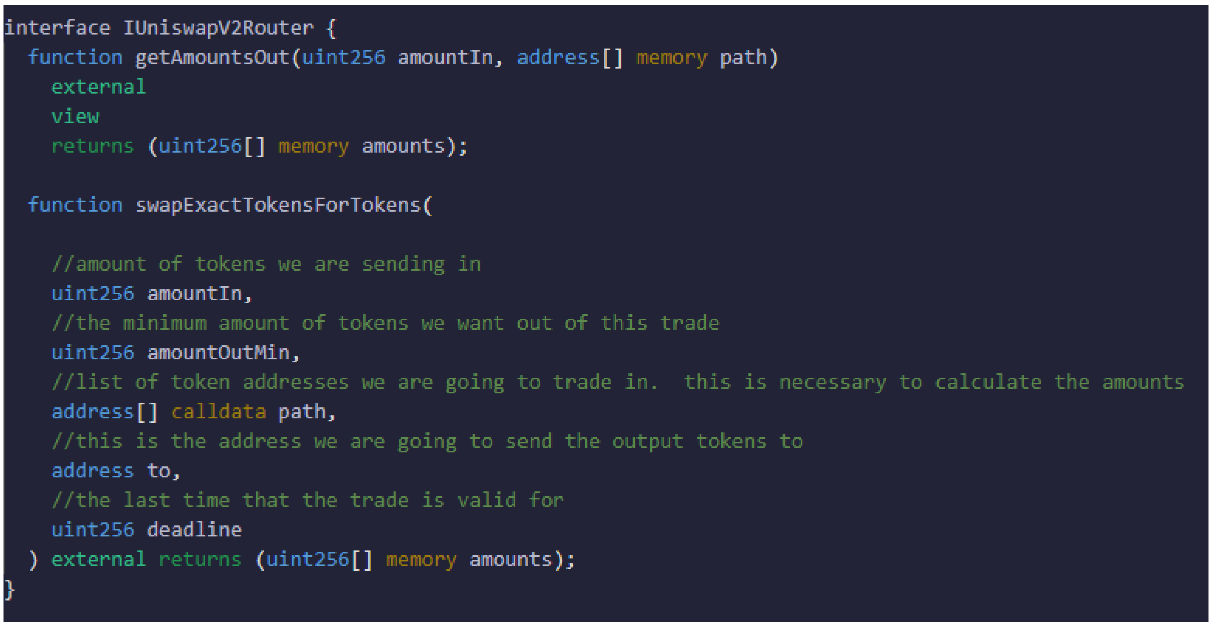
All the codes about uniswap are imported from :
https://cryptomarketpool.com/how-to-swap-tokens-on-uniswap-using-a-smart-contract/
We added clarifying comments to illustrate the function of it in our use case.

This code creates the interface for our smart contract to interact with, making it possible to make swaps with the decentralized exchange uniswap.

Wrapped ETH (WETH) is a token that represents Ethereum, including it allows us to check for price differences between WETH and ETH as the inputs for the swaps. This can create opportunities for a better exchange price. We were not able to obtain the addresses for the uniswap and the corresponding addresses for the tokens in it. This made it impossible for us to test the code relating to the swap.

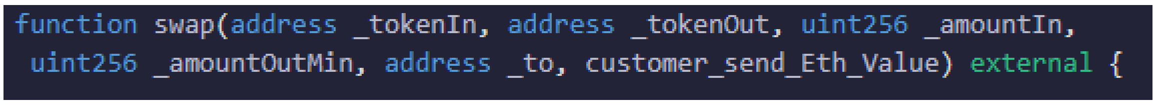


```
interface IERC20 {
    function totalSupply() external view returns (uint);
    function balanceOf(address account) external view returns (uint);
    function transfer(address recipient, uint amount) external returns (bool);
    function allowance(address owner, address spender) external view returns (uint);
    function approve(address spender, uint amount) external returns (bool);
    function transferFrom(
        address sender,
        address recipient,
        uint amount
    ) external returns (bool);
    event Transfer(address indexed from, address indexed to, uint value);
    event Approval(address indexed owner, address indexed spender, uint value);
}
```



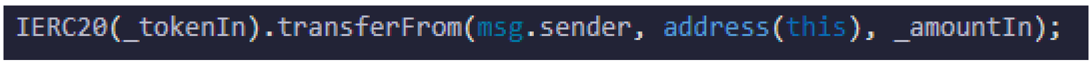
```
interface IUniswapV2Router {
  function getAmountsOut(uint256 amountIn, address[] memory path)
    external
    view
    returns (uint256[] memory amounts);

  function swapExactTokensForTokens(

    //amount of tokens we are sending in
    uint256 amountIn,
    //the minimum amount of tokens we want out of this trade
    uint256 amountOutMin,
    //list of token addresses we are going to trade in.  this is necessary to calculate the amounts
    address[] calldata path,
    //this is the address we are going to send the output tokens to
    address to,
    //the last time that the trade is valid for
    uint256 deadline
  ) external returns (uint256[] memory amounts);
}
```

The token address we want to trade out of (tokenIn), and the token address that is the output of this trade (tokenOut) are defined in the code beneath. AmountIn stands for the amount of tokens that we are sending in, while amountOutMin signifies the minimum amount of tokens that we want out of this trade. Address to is the designated address that we want the newly exchanged tokens to be sent to.



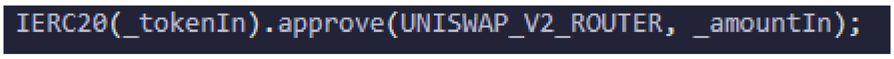
```
function swap(address _tokenIn, address _tokenOut, uint256 _amountIn,
 uint256 _amountOutMin, address _to, customer_send_Eth_Value) external {
```

We facilitate this by first sending the amount of tokens from the msg.sender to this contract :



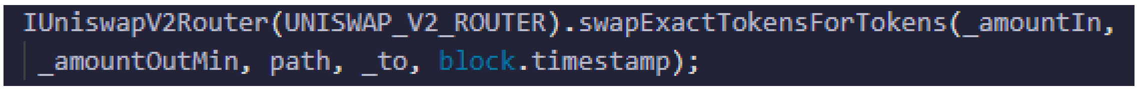
```
IERC20(_tokenIn).transferFrom(msg.sender, address(this), _amountIn);
```

By calling IERC20 approve we allow the uniswap to spend the tokens in the contract:



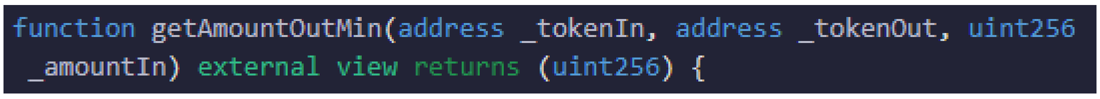
```
IERC20(_tokenIn).approve(UNISWAP_V2_ROUTER, _amountIn);
```

After deciding whether WETH or Native ETH offers better pricing swapExactTokensForTokens will be called :



```
IUniswapV2Router(UNISWAP_V2_ROUTER).swapExactTokensForTokens(_amountIn,
  _amountOutMin, path, _to, block.timestamp);
```

We will pass the deadline in block.timestamp which is the latest time that the trade is valid for. The three inputs below will deliver us the minimum amount required from the swap. This is needed for the swap function above.



```
function getAmountOutMin(address _tokenIn, address _tokenOut, uint256
 _amountIn) external view returns (uint256) {
```

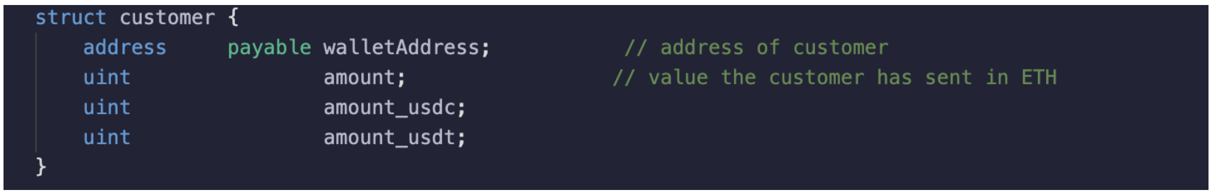
3. Self-Written tokenBasket Contract

3.1. Defining the customer and making deposits/withdrawals available

After the uniswap code is imported, we can proceed with writing our own tokenBasket contract that will interact with the uniswap code.

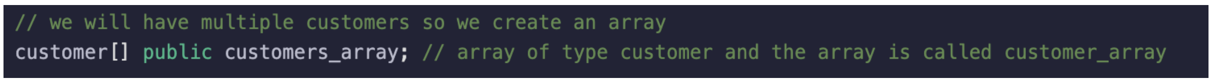
First, we define state variables, in our case: owner (type: address), customer_send_Eth_Value (type: uint, character: public), half_of_customer_send_Eth_Value (type: uint, character: public). We define the owner of the contract once the contract is deployed by using a constructor. We store the main information about the users of our smart contract and their balances in ETH and in tokens in a dynamic array (not a fixed size array, we can always add new customers into it using the command “array.push command”).

First, let us define which properties a customer should have. For that we use struct which is a collection of variables under a single name:



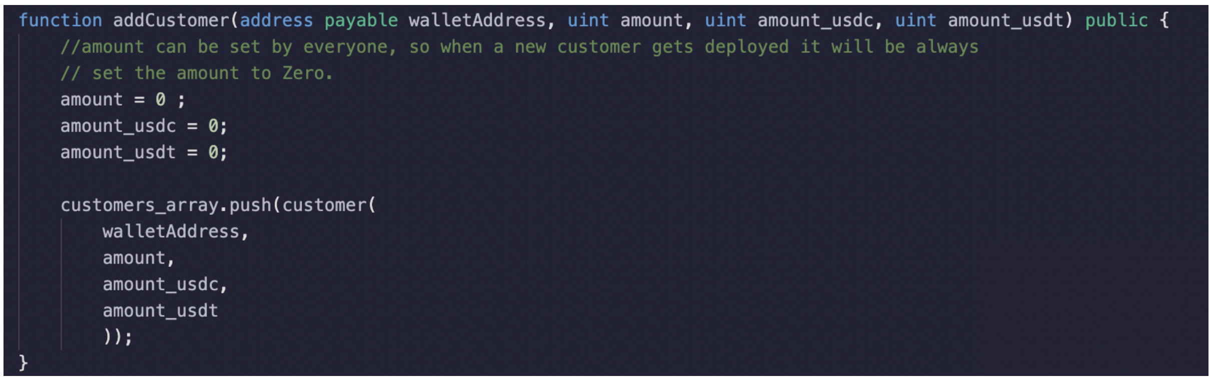
```
struct customer {
    address     payable walletAddress;          // address of customer
    uint                amount;                // value the customer has sent in ETH
    uint                amount_usdc;
    uint                amount_usdt;
}
```

Once we define the parameters of a customer, we can create a dynamic array which is called customers_array.



```
// we will have multiple customers so we create an array
customer[] public customers_array; // array of type customer and the array is called customer_array
```

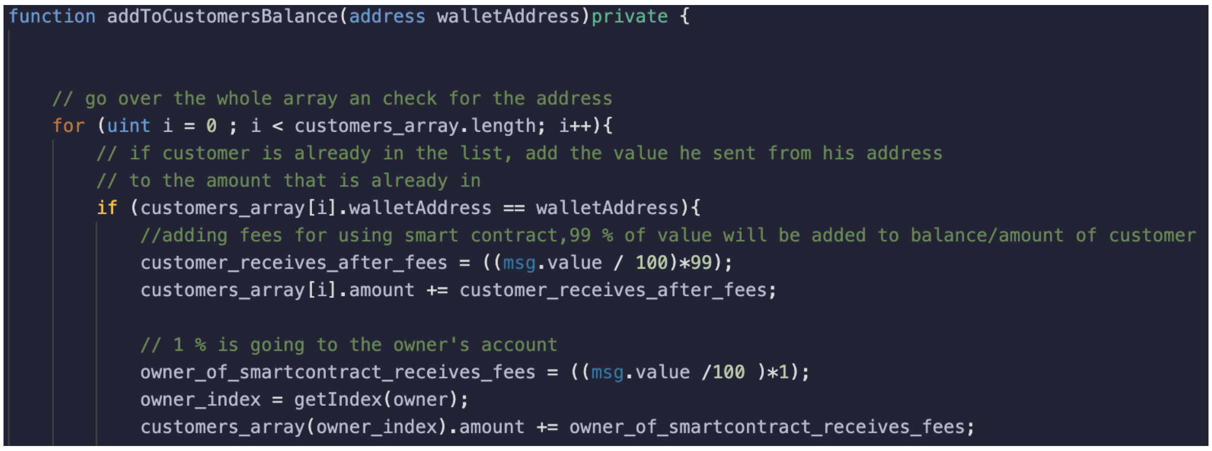
In order to be able to receive money from the customer and swap received ETH for tokens and vice versa we need to add customer to our array first. For that we use function AddCustomer:



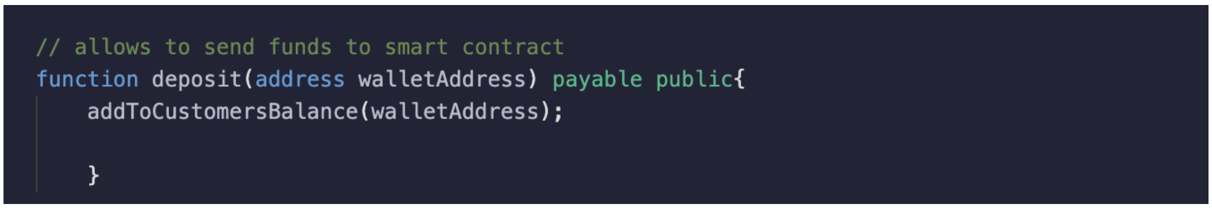
```
function addCustomer(address payable walletAddress, uint amount, uint amount_usdc, uint amount_usdt) public {
    //amount can be set by everyone, so when a new customer gets deployed it will be always
    // set the amount to Zero.
    amount = 0 ;
    amount_usdc = 0;
    amount_usdt = 0;

    customers_array.push(customer(
        walletAddress,
        amount,
        amount_usdc,
        amount_usdt
        ));
}
```

There are no access restrictions, so anyone including the customer can add a new customer to the array. In order to avoid speculations and amount set in the array, we define the default values of the amount of ETH and tokens USDC and USDT.

Now we need to make it possible to deposit funds to the contract, specifically to a concrete customer's account. We do it with function addToCustomersBalance and deposit:



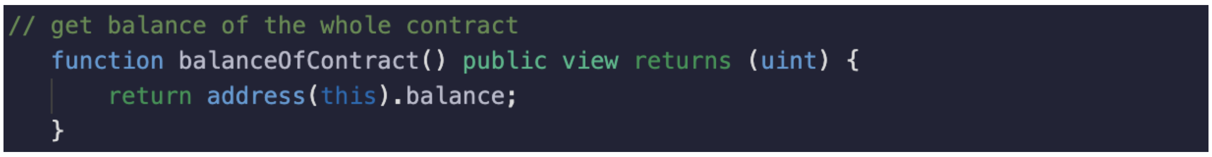
```
function addToCustomersBalance(address walletAddress)private {


    // go over the whole array an check for the address
    for (uint i = 0 ; i < customers_array.length; i++){
        // if customer is already in the list, add the value he sent from his address
        // to the amount that is already in
        if (customers_array[i].walletAddress == walletAddress){
            //adding fees for using smart contract,99 % of value will be added to balance/amount of customer
            customer_receives_after_fees = ((msg.value / 100)*99);
            customers_array[i].amount += customer_receives_after_fees;

            // 1 % is going to the owner's account
            owner_of_smartcontract_receives_fees = ((msg.value /100 )*1);
            owner_index = getIndex(owner);
            customers_array(owner_index).amount += owner_of_smartcontract_receives_fees;
```



```
// allows to send funds to smart contract
function deposit(address walletAddress) payable public{
    addToCustomersBalance(walletAddress);

    }
```

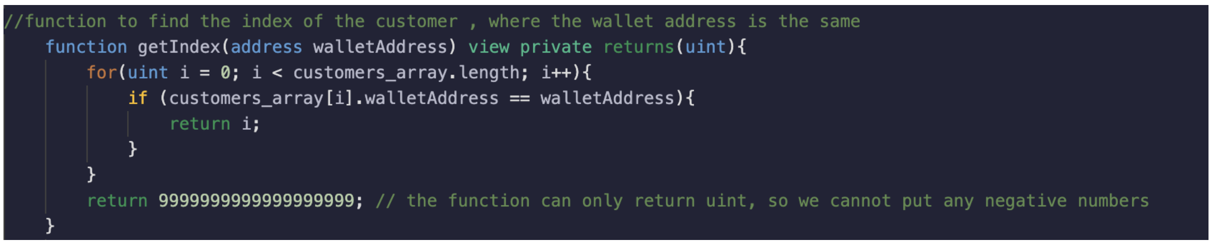
At this stage we charge 1% fee for using our smart contract, so we divide the funds sent and send 99% to the customers balance and 1% to the owners account. The first customer of the smart contract should be the owner so he/she can collect the fees.

We can also check the balance of the whole contract by creating the corresponding function:



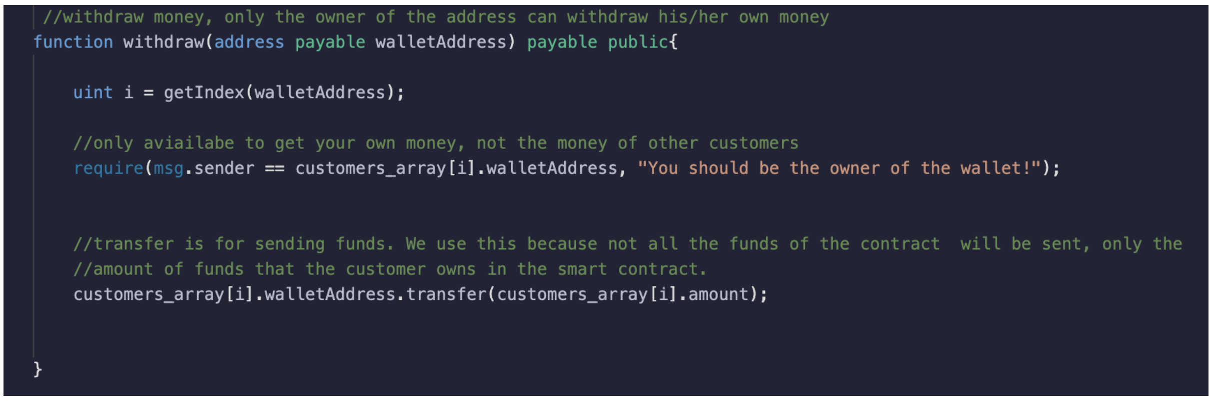
```
// get balance of the whole contract
   function balanceOfContract() public view returns (uint) {
       return address(this).balance;
   }
```

We want to make sure customers can withdraw their own funds from the contract whenever they want. We will check that the account mentioned by the message sender is really their account. For that we use the function getIndex that only can read the state variables (view) and does not consume any gas.



```
//function to find the index of the customer , where the wallet address is the same
    function getIndex(address walletAddress) view private returns(uint){
        for(uint i = 0; i < customers_array.length; i++){
            if (customers_array[i].walletAddress == walletAddress){
                return i;
            }
        }
        return 9999999999999999999; // the function can only return uint, so we cannot put any negative numbers
    }
```

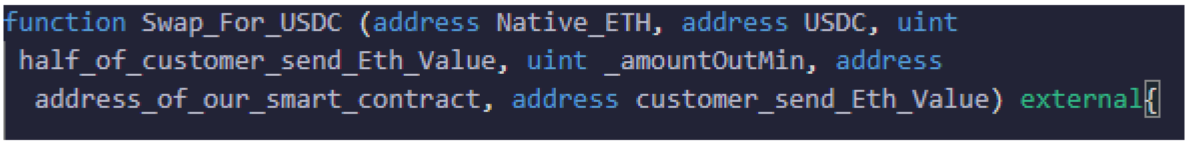
We use an access restriction to require msg.sender == customers_array[i].walletAddress.



```
   //withdraw money, only the owner of the address can withdraw his/her own money
   function withdraw(address payable walletAddress) payable public{

       uint i = getIndex(walletAddress);

       //only aviailabe to get your own money, not the money of other customers
       require(msg.sender == customers_array[i].walletAddress, "You should be the owner of the wallet!");


       //transfer is for sending funds. We use this because not all the funds of the contract  will be sent, only the
       //amount of funds that the customer owns in the smart contract.
       customers_array[i].walletAddress.transfer(customers_array[i].amount);


   }
```

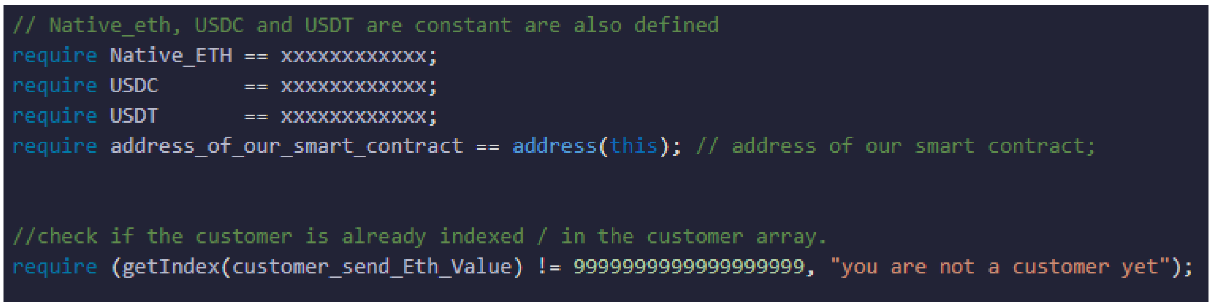
3.2. Swap ETH for USDC/USDT

We define the tokens in our basket as 50% USDC and 50% USDT. We will show you the swap for USDC. Here we swap half of the ETH value into USDC, The USDC goes into the smart contract but is added to the balance of the original customer.



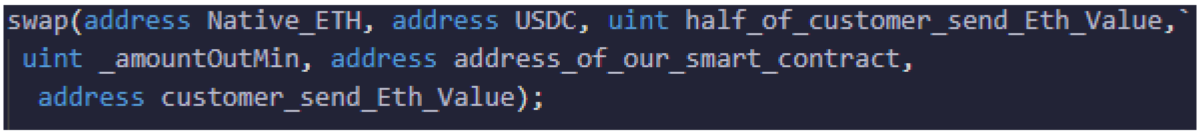
```
function Swap_For_USDC (address Native_ETH, address USDC, uint
 half_of_customer_send_Eth_Value, uint _amountOutMin, address
  address_of_our_smart_contract, address customer_send_Eth_Value) external{
```

Furthermore a check is needed in order to make sure that the customer is already indexed. Otherwise the process is canceled and an error message will appear.



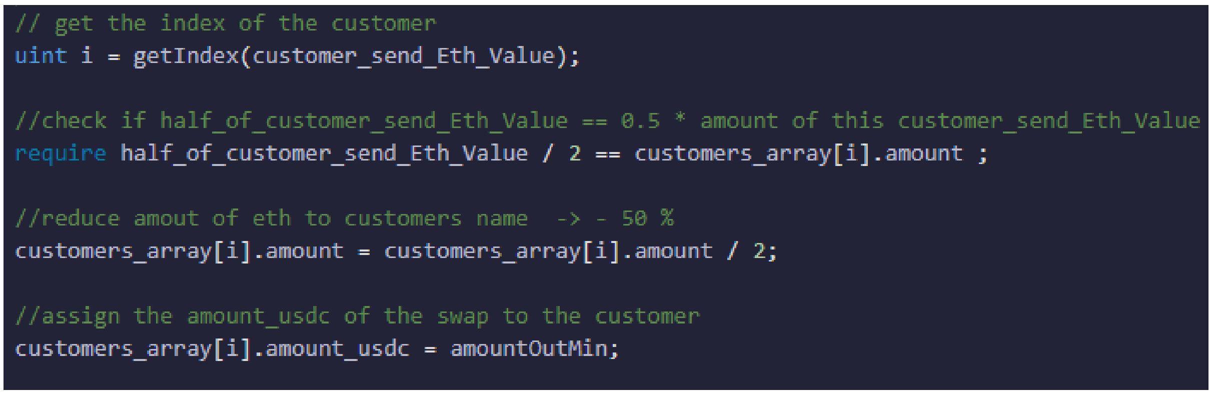
```
// Native_eth, USDC and USDT are constant are also defined
require Native_ETH == xxxxxxxxxxxx;
require USDC       == xxxxxxxxxxxx;
require USDT       == xxxxxxxxxxxx;
require address_of_our_smart_contract == address(this); // address of our smart contract;


//check if the customer is already indexed / in the customer array.
require (getIndex(customer_send_Eth_Value) != 9999999999999999999, "you are not a customer yet");
```

After ensuring that the requirements are followed the swap takes place :



```
swap(address Native_ETH, address USDC, uint half_of_customer_send_Eth_Value,`
 uint _amountOutMin, address address_of_our_smart_contract,
  address customer_send_Eth_Value);
```

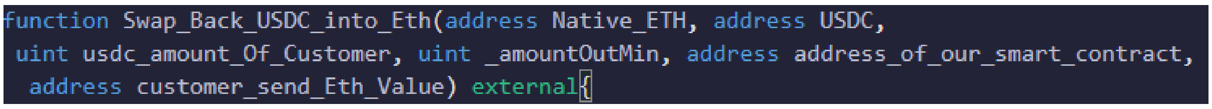
This leaves the customer with 50% ETH and 50% USDC being held by the contract. By doing the same procedure now for USDT the Swap of ETH to 50% USDC and 50% USDT is finished.



```
// get the index of the customer
uint i = getIndex(customer_send_Eth_Value);

//check if half_of_customer_send_Eth_Value == 0.5 * amount of this customer_send_Eth_Value
require half_of_customer_send_Eth_Value / 2 == customers_array[i].amount ;

//reduce amout of eth to customers name  -> - 50 %
customers_array[i].amount = customers_array[i].amount / 2;

//assign the amount_usdc of the swap to the customer
customers_array[i].amount_usdc = amountOutMin;
```

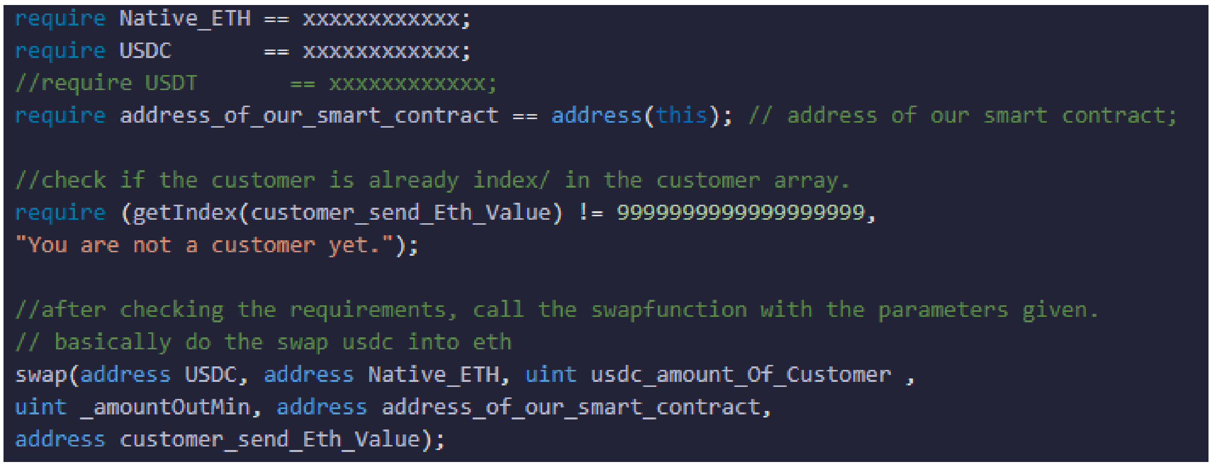
3.3. Swap USDC/USDT for ETH

Here we swap all of the USDC back into ETH. The same can be done with USDT by replacing USDC with USDT in the code.



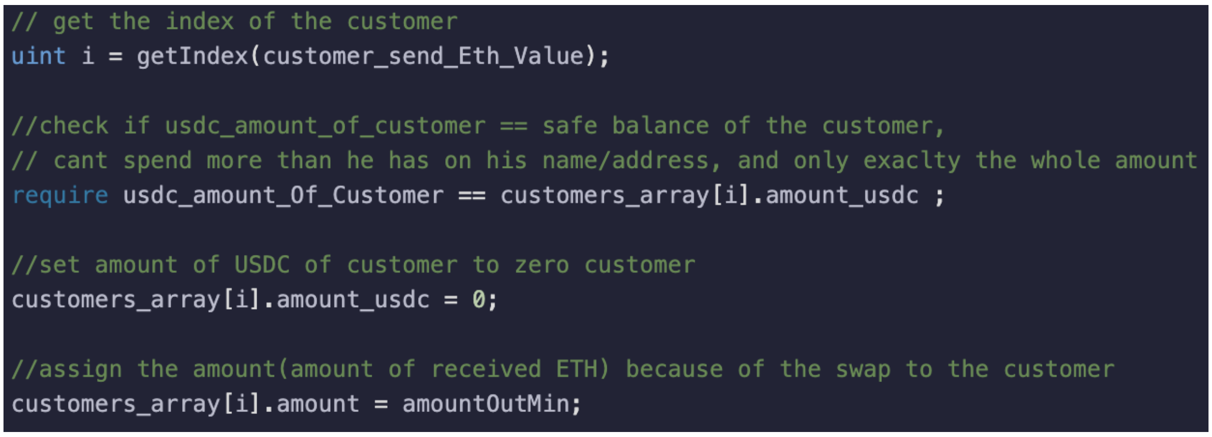
```
function Swap_Back_USDC_into_Eth(address Native_ETH, address USDC,
 uint usdc_amount_Of_Customer, uint _amountOutMin, address address_of_our_smart_contract,
  address customer_send_Eth_Value) external{
```

The same checks as before apply here. We make sure that the customer is indexed, otherwise he'll receive an error message. Adding to that we ensure the requirements given and finally execute the swap function.



```
require Native_ETH == xxxxxxxxxxxx;
require USDC       == xxxxxxxxxxxx;
//require USDT       == xxxxxxxxxxxx;
require address_of_our_smart_contract == address(this); // address of our smart contract;

//check if the customer is already index/ in the customer array.
require (getIndex(customer_send_Eth_Value) != 9999999999999999999,
"You are not a customer yet.");

//after checking the requirements, call the swapfunction with the parameters given.
// basically do the swap usdc into eth
swap(address USDC, address Native_ETH, uint usdc_amount_Of_Customer ,
uint _amountOutMin, address address_of_our_smart_contract,
address customer_send_Eth_Value);
```

In the end another security check ensures that the customer has enough funds to swap his tokens back to ETH and finish the process of swapping USDC/USDT back into ETH.



```
// get the index of the customer
uint i = getIndex(customer_send_Eth_Value);

//check if usdc_amount_of_customer == safe balance of the customer,
// cant spend more than he has on his name/address, and only exaclty the whole amount
require usdc_amount_Of_Customer == customers_array[i].amount_usdc ;

//set amount of USDC of customer to zero customer
customers_array[i].amount_usdc = 0;

//assign the amount(amount of received ETH) because of the swap to the customer
customers_array[i].amount = amountOutMin;
```

4. Faced Problems and Challenges

1. The code is not completely tested (only the part without swap is tested and it works): We could not find correct addresses for USDC and USDT tokens for Goerli testnet, which are used in the smart contract of the decentralized exchange uniswap. Also we could not find the correct address of uniswap itself . Therefore, the smart contract is impossible to test on Goerli testnet.
2. Possible problem with for loops: if our token basket contract becomes popular and more customers are stored in the array, checking for the correspondent walletAddress through for loops in the array can be very costly in terms of the gas. We would need to find a different method.

5. Future-Added Features and Improvements of the Smart Contract

To make our smart contract attractive and beneficial for our customers we plan to upgrade it by making one group swap automatically for all the individual interested customers every 24 hours (adding a time.stamp as a deadline). All the swap requests from the customers during the day will be added to the queue, grouped and swapped as one whole group. The received funds from the swaps then will be added to the customers amounts/balances. This way we reduce gas fees needed for each individual swap by paying only once for the whole group swap operation. At the moment, we have not yet solved how and who will pay the gas fees for each swap. But we plan to implement that the gas fees for a group swap should be paid by the smart contracts owners balance/amount (that collects 1% fee from each customer).