Treasury Smart Contract

The Treasury smart contract is designed to manage funds by depositing USDC tokens into Uniswap and Aave protocols. The deposited funds are then swapped for BUSD and DAI tokens, respectively. The contract allows the owner to set the distribution ratios for Uniswap and Aave, deposit funds, withdraw funds, and calculate the yield.

Constructor

The constructor initializes the contract with the following parameters:

- **_usdcToken**: Address of the USDC token
- **_busdToken**: Address of the BUSD token
- **_daiToken**: Address of the DAI token
- **_uniswapRouter**: Address of the Uniswap V2 Router
- **_aaveLendingPool**: Address of the Aave Lending Pool

These addresses are stored in the contract as public variables, allowing external parties to interact with them.

Functions

1. **deposit:** the deposit function in the Treasury contract performs several actions upon receiving a deposit of USDC tokens:

- Token Transfer:Transfers the specified amount of USDC tokens from the sender's account (msg.sender) to the Treasury contract.
- Quantity Calculation:Calculates the equivalent amounts of BUSD and DAI tokens based on the allocations provided by the allocation ratios.
- Approve Spending:Approves the spending of the calculated amounts of BUSD and DAI tokens on the Uniswap router contract.
- Uniswap Exchange:Performs exchanges in Uniswap to convert part of the USDC tokens to BUSD and DAI tokens according to the calculated amounts.
- Event Emitted: Emitted a Deposit event to record the action, indicating the sender, the amount deposited and the token (USDC) address.

The function automates the process of receiving USDC tokens, performing trades in Uniswap to adjust the portfolio according to the set allocation ratios, and records the action via events. This allows maintaining the desired balance of tokens in the Treasury contract.

```
// Deposit funds into the treasury contract
function deposit(uint256 amount) external {
        IERC20 token = IERC20(usdcTokenAddress);

        require(
            token.transferFrom(msg.sender, address(this), amount),
            "Failed to transfer tokens"
        );
        uint256 busdAmount = (amount * busdAllocationRatio) / 100;
        uint256 daiAmount = (amount * daiAllocationRatio) / 100;
        address[] memory pathToBUSD = new address[](2);
        pathToBUSD[0] = usdcTokenAddress;
        pathToBUSD[1] = busdTokenAddress;
        address[] memory pathToDAI = new address[](2);
        pathToDAI[0] = usdcTokenAddress;
        pathToDAI[1] = address(daiTokenAddress);
        if (busdAmount > 0) {
            token.approve(address(uniswapRouterAddress), busdAmount);
            swap(pathToBUSD, busdAmount);
        }
        if (daiAmount > 0) {
            token.approve(address(uniswapRouterAddress), daiAmount);
            swap(pathToDAI, daiAmount);
        }
        emit Deposit(msg.sender, amount, usdcTokenAddress);
  }
```

2. **withdraw:** allows the contract owner to withdraw funds from the treasury, ensuring that sufficient funds are available and recording the action through events for proper transparency.

- Token Transfer to Owner: Transfers the specified amount of tokens from the Treasury contract to the owner's (contract owner) account.
- Sufficient Balance Check: Verifies that the Treasury contract has a token balance equal to or greater than the amount to be withdrawn.
- Event Emit: Emitted a Withdrawal event to record the withdrawal action, indicating the recipient (owner), amount withdrawn, and token address.

```
// Withdraw funds from the treasury contract
 function withdraw(uint256 amount, address tokenAddress) external onlyOwner {
 IERC20 token = IERC20(tokenAddress);
        require(
            token.balanceOf(address(this)) >= amount,
             "Insufficient token balance"
        );
        require(
            token.transfer(msg.sender, amount),
            "Failed to transfer tokens"
        );
        emit Withdrawal(msg.sender, amount, tokenAddress);
    }
```

3. **setAllocationRatios**: allows the owner of the Treasury contract to adjust the allocation proportions for three different tokens (USDC, BUSD, DAI). These proportions represent the percentage distribution of the funds deposited in the contract among the specified tokens.

- Input parameters: The function takes three parameters of type uint256 representing the new allocation ratios for USDC, BUSD and DAI.
- Sum validation: The function verifies that the sum of the new allocation ratios equals 100. If not, an error message is issued indicating that the ratios must sum to 100.
- Proportions update: If the validation passes, the usdcAllocationRatio, busdAllocationRatio and daiAllocationRatio status variables are updated with the new values provided.
- Event emit: A Ratios event is emitted with the new allocation ratios as parameters.

This function is crucial for dynamically adjusting how funds are distributed among the different tokens, allowing the owner to adapt the investment strategy of the contract as needed.

```
// Set the allocation ratios for tokens
function setAllocationRatios(
        uint256 _usdcAllocationRatio,
        uint256 _busdAllocationRatio,
        uint256 _daiAllocationRatio
    ) external onlyOwner {
        require(
            _usdcAllocationRatio + _busdAllocationRatio + _daiAllocationRatio ==
                100,
            "Allocation ratios must add up to 100"
        );
        usdcAllocationRatio = _usdcAllocationRatio;
        busdAllocationRatio = _busdAllocationRatio;
        daiAllocationRatio = _daiAllocationRatio;
        emit Ratios(
            usdcAllocationRatio,
            busdAllocationRatio,
            daiAllocationRatio
        );
    }
```

4. **swap:** this function facilitates the exchange of allocated funds between different tokens using the Uniswap protocol, ensuring that there are sufficient funds and that only the owner can perform such operations.

- Token Approval: Approves the expenditure of a specific amount of tokens to the Uniswap router contract via the IERC20 interface.
- Sufficient Balance Verification: Verifies that the Treasury contract has a token balance equal to or greater than the amount to be exchanged.
- Token Exchange: Uses the Uniswap router to swap a specified amount of tokens. The swapExactTokensForTokens function ensures that at least the specified amount of output tokens is received, and the excess can be returned.
- Use of the onlyOwner modifier: The function is marked as private and uses the onlyOwner modifier, which means that only the owner of the contract can execute this function.

```
// Swap allocated funds between different tokens using Uniswap

  function swap(address[] memory path, uint256 amount) private onlyOwner {
    IERC20 token = IERC20(path[0]);
    require(token.balanceOf(address(this)) >= amount, "low balance");
    token.approve(address(uniswapRouterAddress), amount);
    IUniswapRouter(uniswapRouterAddress).swapExactTokensForTokens(
        amount,
        0,
            path,
            address(this)
        );
    }
```

5. **swapToken:** this feature facilitates the exchange of funds from a specific token to another desired token using the Uniswap protocol, ensuring that there are sufficient funds from the source token and that only the owner can perform such operations.

- Token Approval: Approves the expenditure of a specific amount of tokens to the Uniswap router contract via the IERC20 interface.
- Sufficient Balance Verification: Verifies that the Treasury contract has a balance of tokens of the source type equal to or greater than the amount to be exchanged.
- Exchange Path Construction: Creates an address array (path) with two elements, where the first element is the address of the source token and the second is the address of the destination token.
- Internal Swap Function Call: Calls the internal swap function with the constructed swap path and the specified quantity.
- Use of Modifier onlyOwner: The function is marked as external and uses the onlyOwner modifier, which means that only the owner of the contract can execute this function.

```
//swap token to desired token
    function swapToken(
        address from,
        address to,
        uint256 amount
    ) external onlyOwner {
        IERC20 token = IERC20(from);
        require(token.balanceOf(address(this)) >= amount, "low balance");
        address[] memory path = new address[](2);
        path[0] = from;
        path[1] = to;
        swap(path, amount);
    }
```

6. **depositToAave**: facilitates the deposit of tokens into the Aave protocol.

- Input parameters: the function takes two parameters: amount (amount of tokens to deposit) and tokenAddress (contract address of the token to deposit).

- Contract instantiation: An instance of the IERC20 interface is created for the token and of the IAaveLendingPool interface for the Aave Lending Pool.
- Spend Approval: Token spending is approved by calling the approve method of the token. This allows the contract to spend the specified amount of tokens on behalf of the user.
- Deposit to Aave: The deposit is made to Aave by calling the deposit method of the Aave Lending Pool. The deposit includes the amount, the address of the sender (the contract) and the parameter 0.

This function is essential to allow the Treasury contract owner to deposit funds into Aave in a controlled and secure manner, using a simplified interface to interact with token contracts and Aave.

```
function depositToAave(
        uint256 amount,
        address tokenAddress
    ) external onlyOwner {
        IERC20 token = IERC20(tokenAddress);
        IAaveLendingPool aaveLendingPool = IAaveLendingPool(
            aaveLendingPoolAddress
        );

        token.approve(aaveLendingPoolAddress, amount);
        aaveLendingPool.deposit(tokenAddress, amount, address(this), 0);
    }
```

7. **withdrawFromAave**:is based on the asset management of the Treasury smart contract, allowing the owner to withdraw funds previously deposited in the Aave protocol.

- Input Parameters:

  amount: Number of tokens to be withdrawn.

  tokenAddress: Address of the token to be withdrawn.
- Aave Withdrawal:

  Uses the Aave Lending Pool to withdraw the specified amount of tokens.
- TokenTransfer:

  Transfers the withdrawn tokens to the owner of the contract.
- Verification:

Verifies whether the transfer was successful.

The withdrawFromAave function provides a secure interface for the Treasury contract owner to effectively manage funds deposited in the Aave protocol. It allows assets to be withdrawn as needed, thus maintaining strategic flexibility and facilitating adaptation to market conditions and financial needs.

```
function withdrawFromAave(
        uint256 amount,
        address tokenAddress
    ) external onlyOwner {
        IAaveLendingPool aaveLendingPool = IAaveLendingPool(
            aaveLendingPoolAddress
        );

        aaveLendingPool.withdraw(tokenAddress, amount, address(this));

        IERC20 token = IERC20(tokenAddress);
        require(
            token.transfer(owner, amount),
            "Failed to transfer tokens from Aave"
        );
    }
```

Block Diagram

In this diagram, you can see the following interactions:

1. The Treasury smart contract deposits USDC tokens into Uniswap V2 Router to swap for BUSD tokens.
2. The Treasury smart contract deposits USDC tokens into Aave Lending Pool, which returns DAI tokens.
3. The Treasury smart contract can withdraw BUSD and DAI tokens.

This diagram provides a comprehensive view of the interactions between the Treasury smart contract, Uniswap V2 Router, Aave Lending Pool, and the respective tokens involved.

Here's a description of the main components and their relationships:

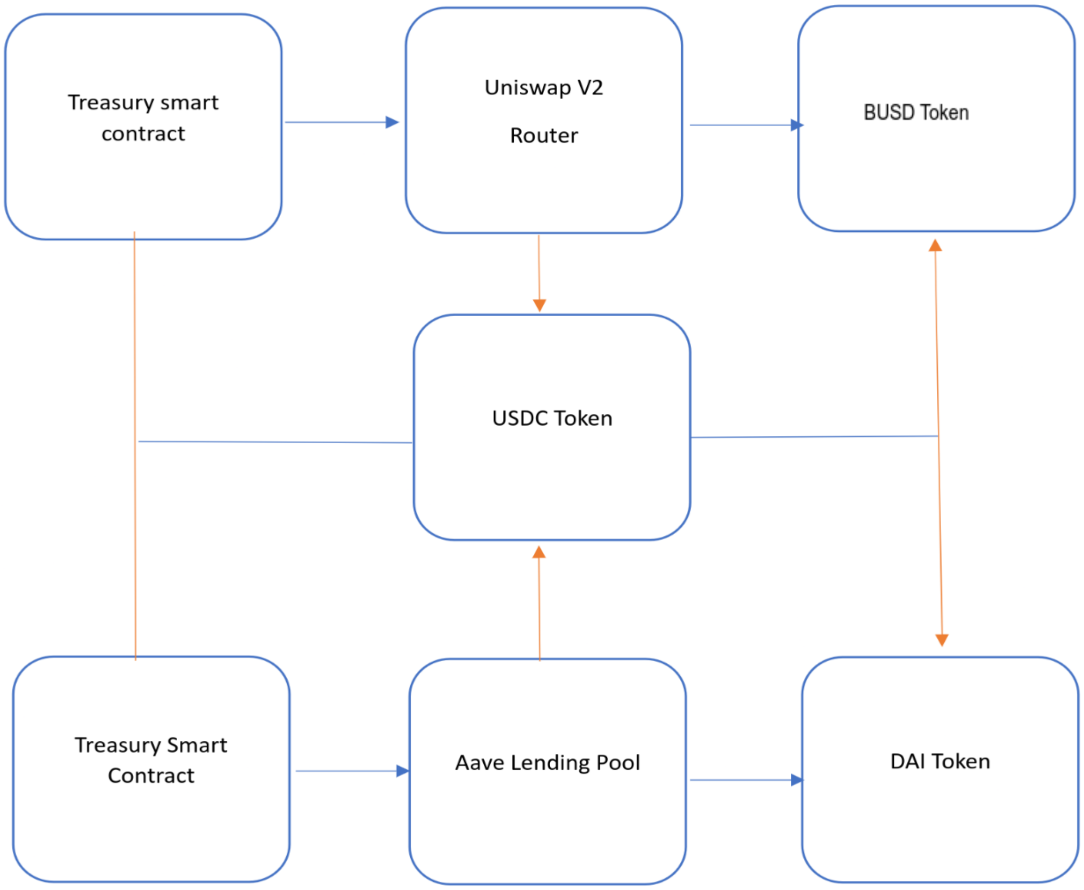
1. User: Interacts with the Treasury smart contract by depositing USDC tokens.

2. Treasury Smart Contract: Holds the deposited USDC tokens and distributes them between Uniswap and Aave based on the set ratios.

3. Uniswap V2 Router: The Treasury smart contract interacts with the Uniswap V2 Router to swap USDC tokens for BUSD tokens.

4. Aave Lending Pool: The Treasury smart contract interacts with the Aave Lending Pool to deposit USDC tokens and receive DAI tokens in return.

5. USDC, BUSD, and DAI Tokens: These are the ERC20 tokens involved in the transactions within the Treasury smart contract.


Treasury smart contract
Uniswap V2 Router
BUSD Token
USDC Token
Treasury Smart Contract
Aave Lending Pool
DAI Token