## IDO:

1- **Module:** The smart contract is defined in a module called "IDO."

2- **Sale Struct:** The smart contract defines a Sale struct that represents the IDO sale. The Sale struct has the following fields:

start: A u64 integer that represents the start time of the IDO sale in Unix timestamp format.

end: A u64 integer that represents the end time of the IDO sale in Unix timestamp format.

cap: A u64 integer that represents the token sale limit.

whitelist: A vector of Address structs that represents a list of wallet addresses that are allowed to participate in the IDO.

owner: An address that represents the address of the contract owner.

3- **Address Struct:** The smart contract also defines an Address struct that represents an address. The Address struct has one field:

value: A vector of u8 integers that represents the byte array representation of the address.

4- **Create Sale Function:** The smart contract defines a function called "create\_sale" that creates a new IDO sale. The create\_sale function takes the following parameters:

start: A u64 integer that represents the start time of the IDO sale.

end: A u64 integer that represents the end time of the IDO sale.

cap: A u64 integer that represents the token sale limit.

whitelist: A vector of Address structs that represents a list of wallet addresses that are allowed to participate in the IDO.

owner: An address that represents the address of the contract owner.

The create\_sale function creates a new Sale struct with the provided information, converts the Sale struct to bytes, stores the Sale information on chain, and returns the Sale reference.

5- **Is Whitelisted Function:** The smart contract defines a function called "is\_whitelisted" that checks if an address is whitelisted for the IDO sale. The is\_whitelisted function takes the following parameters:

sale\_ref: A reference to the Sale struct.

address: An Address struct that represents the address being checked.

The is\_whitelisted function checks if the provided address is in the whitelist by using the vector\_index\_of function to find the index of the provided address in the whitelist vector. If the index is not -1, the function returns true.

6- **Buy Tokens Function**: The smart contract defines a function called "buy\_tokens" that allows participants to buy tokens during the IDO sale. The buy\_tokens function takes the following parameters:

sale\_ref: A reference to the Sale struct.

amount: A u64 integer that represents the amount of tokens being purchased.

The buy\_tokens function checks if the sale is still ongoing, if the amount being purchased is within the cap, and if the sender is whitelisted. If all the conditions are met, the function transfers tokens to the sender.

## **Fund IDO:**

1- The contract defines two structs, "Whitelist" and "IDO", which hold data about whitelisted addresses and the IDO itself.

2- The "init" function initializes the IDO and whitelist data in storage, and performs several checks to ensure that the data is valid.

3- The "whitelist\_address" function allows the IDO owner to add an address to the whitelist for the IDO.

4- The function checks that the sender of the transaction is the IDO owner before adding the address to the whitelist.

5- The function checks that the address is not already whitelisted before adding it to the whitelist.

6- The "fund" function allows a whitelisted address to fund the IDO with Move tokens.

7- The function checks that the sender's address is whitelisted for the IDO before allowing them to fund it.

8- The function checks that the IDO has not ended before allowing funding.

9- The function checks that the IDO has not reached its funding cap before allowing funding.

10- If all checks pass, the function transfers the Move tokens from the sender to the IDO owner.

## Transfer IDO fund:

1- The IDO struct is defined with several fields, including the start and end date/time of the IDO, the funding cap, a vector of whitelisted addresses, the owner of the IDO, and the current balance of the IDO.

2- The TransferIDOFund module is defined, which contains a public transfer\_fund function that takes in a mutable reference to an IDO, a recipient address, and an amount to transfer.

3- Within the transfer\_fund function, an assertion is made that only the owner of the IDO can transfer funds.

4- An assertion is made to check that the recipient address is whitelisted for the IDO.

5- An assertion is made to check that the amount to transfer is within the available balance of the IDO.

6- The IDO balance is updated to subtract the transferred amount.

7- The funds are transferred to the recipient using a send\_from\_faucet function (not shown in this code snippet).

8- An assertion is made to check that the recipient's balance has increased by the transferred amount.

## **Deployment Process:**

1- Choose a testnet: Select a testnet that is compatible with the Move language, such as the Libra testnet or the Diem testnet.

2- Install the Move CLI: Install the Move CLI tool on your local machine. The Move CLI allows you to interact with the testnet and deploy your code.

3- Set up your environment: Set up your development environment, including your account credentials and any necessary API keys. You may need to create an account on the testnet and obtain a testnet wallet address.

4- Write and compile your code: Write your Move language code in a text editor or integrated development environment (IDE). Use the Move CLI to compile your code into a bytecode file that can be executed on the testnet.

5- Test your code: Test your bytecode file locally using the Move CLI. Use the Move CLI to run your bytecode file and verify that it behaves as expected.

6- Deploy your code: Use the Move CLI to deploy your bytecode file to the testnet. You may need to specify parameters such as the location of the bytecode file and the address of the testnet.

7- Interact with your code: Use the Move CLI or other development tools to interact with your deployed code on the testnet. You can test its functionality, debug any issues, and make updates as necessary.

## **Frontend application:**

* The code imports the React and axios libraries.
* The component function App is defined using the useState hook to define state variables for IDO1Data, fundIDOData, and transferIDOFundsData, which will be used to store data retrieved from the web API.
* Three event handler functions (handleIDO1Click, handleFundIDOClick, and handleTransferIDOFundsClick) are defined using the axios.get function to make HTTP GET requests to the specified API endpoint and update the corresponding state variable with the retrieved data.
* A return statement defines the layout of the component. It displays a title, three buttons, and any data that has been retrieved from the API in response to button clicks.
* The component is exported using the export default statement to make it available for use in other parts of the application.

## **Web3.js For Smart Contract interaction:**

1- Import the Web3.js library:

const Web3 = require('web3');

This line imports the Web3.js library into your project.

2- Set up a connection to the network:

const web3 = new Web3('https://ropsten.infura.io/v3/your-project-id');

This line connects your application to the Ethereum network via Infura's API, using your project ID.

3- Set up a contract object:

These lines define the contract's address and ABI (Application Binary Interface), which are necessary to interact with the contract. The new web3.eth.Contract() method creates a contract object based on the provided ABI and address.

4- Call a contract function:

These lines call a function on the smart contract using the contract.methods object, passing in the function name and any necessary arguments. The call() method sends the transaction to the network and returns the function's result.

5- Send a transaction to a contract function:

hese lines send a transaction to a contract function, using a private key to sign the transaction. The getTransactionCount() method gets the current nonce for the sender address, and the getGasPrice() method gets the current gas price. The encodeABI() method encodes the function call and arguments to be included in the transaction data. The signTransaction() method signs the transaction with the private key, and the sendSignedTransaction() method sends the signed transaction to the network. Finally, the transaction hash is logged to the console.