Ophir Workshop Voting Dapp

What are We Doing Here?

Ophir is building a voting dapp for @dappsoverapps community that allows members to vote on whether the event resources, such as grants, donations or investment funds should be spent on any of the two proposals:

Research projects or Operational expenses.

What is a Voting Dapp?

Let's break things down 🚀

➢ **Voting 🗳️**

Voting in terms of this smart contract means the act of participants selecting their proposals among available options. Each participants can cast a single vote for a specific proposal.

➢ **Smart Contracts**

They are set of rules that self-execute only when pre-determined conditions are met.

Examples: **BNB/USDT**

DApps

These are applications built with smart contracts. 😀

➢ **Intrinsic Function of the Voting Dapp.**

1. It will accept proposals Name, and number for tracking.
2. Allow for members to vote and exercise voting ability. (keep track of voting, check if voters are authenticated to vote.
3. There will be an “authority” that will serve as an authenticator.

Let's get familiar with some terms:

The Solidity Syntax-

- **Comments**:

  Example:

  Single line: //

  Multi line: /* *\

- **Variables:**

  Variables store different datatypes

  Solidity supports several data types for variables such as:

  - Value data types and,
  - Reference data types

- **Value data types are:**

  Data types that holds their data directly, the variable value is stored in the location the variable has access to.

  E.g bool, uint, int, address

- **Reference data types are:**

  Data types that serve as reference to data rather than data itself.

  They are often used to create more complex data structures.

  E.g Arrays, structs, mappings and strings

- **Other syntax are:**

  Contracts

  Functions

  Events

  Modifiers

➢ FULL CODE

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

pragma solidity >= 0.7.0<0.9.0;

//making a voting contract

//1. We want the ability to accept proposals and store them

//Proposal: their name, number

//2. Voters & Voting ability
//Keep track of voting
//Check Voters are authenticated to vote

//3. Chairman (Entity rep)
//authenticate and deploy contract
```

```
contract VotersBox {
// all the code goes here

//struct is a refrence data type that can hold multiple pieces of data together (data structures)

//voters: anyvotes? = bool, access to vote = uint (who gets access to vote), vote index = uint(votes per user)

struct Voter {
uint vote;
bool anyvotes;
uint value;
}

struct Proposal {
//bytes are a basic unit measurement of information in computer processing
bytes32 name; //name of each proposal
uint voteCount; //Number of accumulated votes
}
```

```
//Array stores data in series
Proposal [] public proposals;

//mappings allow to create a store value with keys and indexes

//Keeping record of voters by address

mapping(address => Voter ) public voters; //Voters gets the address as a key and voter for value.

//chairman

address public chairman;

//should be in an array

constructor(bytes32[] memory proposalNames) {
//memory: Defines a temporary data location during runtime
//we guarantee space for it
```

```
//??? //We want the chairman to be the one signing the contract so we sign him to msg.sender

//msg.sender = is a global variable that states the person
//who is currently connecting to the contract

chairman = msg.sender;

// add 1 to chairman value
voters[chairman].value = 1;

//Types of loops

//will add proposal names to the smartcontract upon deployment
for(uint i=0; i < proposalNames.length; i++ ) {
//accessing the proposal array nd bytes
proposals.push(Proposal({
name: proposalNames[i],
voteCount: 0
}));
}
```

```
//(first prameter i=0 initiallizes the loop, second tell me how long you want loop to run)


//for loop allows us to loop through proposals in arrays
}

//function authenticate votes

function giveRightToVote(address voter) public {
require(msg.sender == chairman,
'Only the chairman give access to vote');
//require that the voter hasn't voted yet
require(!voters[voter].anyvotes,
'The voter has already voted');
require(voters[voter].value == 0);

voters[voter].value = 1;
}
```

```
//function for voting

function vote(uint proposal) public{

Voter storage sender = voters[msg.sender];
require(sender.value != 0, 'Has no right to vote');
require(!sender.anyvotes, 'Already voted');
sender.anyvotes = true;
sender.vote = proposal;

proposals[proposal].voteCount = proposals[proposal].voteCount + sender.value;

}
```

```
// functions for showing results

//1. function that shows the winning proposal by integer

function winningProposal() public view returns (uint winningProposal_) {

uint winningVoteCount = 0;
for(uint i = 0; i < proposals.length; i++ ) {
if(proposals[i].voteCount > winningVoteCount) {
winningVoteCount = proposals[i].voteCount;
winningProposal_ = i;
}

}
}
```

```
//2. function that shows the winner by name

function winningName() public view returns (bytes32 winningName_) {

winningName_ = proposals[winningProposal()].name;

}

}
```

The voting dapp Code structure demystified

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

This line specifies the license under which the code is released.

```
pragma solidity >= 0.7.0<0.9.0;
```

This line specifies the version of the Solidity programming language that the code is compatible with.

/**

1. It will accept proposals Name, and number for tracking
2. Allow for members to vote and exercise voting ability. (keep track of voting, check if voters are authenticated to vote.
3. There will be an “authority” that will serve as an authenticator

**/

This comment indicates the use case of the contract

```
contract DOAvotingDapp {
```

This is the start of our smart contract, which we're calling "DOAvotingDapp".

```
struct Voter {

uint vote;
bool anyvotes;
uint value;
}
```

This code defines a structure called Voter that represents a voter.

It contains three properties:

vote (the proposal number the voter voted for),

anyvotes (a flag indicating if the voter has voted),

and value (used for authentication purposes).

```
struct Proposal {
bytes32 name;
uint voteCount;
}
```

This code defines a structure called Proposal that represents a proposal.

It contains two properties: name (the name of the proposal) and

voteCount (the number of accumulated votes for the proposal).

Proposal [ ] public proposals;

Here we're declaring a public array of Proposal structs. This means that anyone can see the list of proposals.

mapping(address => Voter) public voters;

We're creating a "mapping" (similar to a dictionary) from addresses to Voters. This means we can look up a Voter by their Ethereum address.

address public authority;

We're creating a variable to store the Ethereum address of the authority.

constructor(bytes32[] memory proposalNames) {

This is the function that gets called when the contract is first deployed. It takes an array of proposal names as input.

```
authority = msg.sender;
```

The person who deploys the contract (represented by msg.sender) is assigned as the “authority”.

```
voters[authority ].value = 1;
```

The authority is given a voting value of 1.

```
for(uint i=0; i < proposalNames.length; i++ ) {
```

This is a loop that goes through each proposal name given when the contract was deployed.

```
proposals.push(Proposal({ name: proposalNames[i], voteCount: 0 }));
```

Each proposal name is used to create a new Proposal struct, which is then added to the array of proposals.

```
//Function for authenticating vote

function giveRightToVote(address voter) public {
```

This function, which can be called by anyone, gives a specific Ethereum address the right to vote.

```
require(msg.sender == authority, 'Only the authority give access to vote');
```

This line checks that the person calling the function is the authority. If they're not, the function will stop and an error message will be displayed.

```
voters[voter].value = 1;
```

The specified Ethereum address is given a voting value of 1 note that authority value is 1

//function for voting

“function vote(uint proposal) public{

This function allows an Ethereum address to cast their vote for a specific proposal.

Voter storage sender = voters[msg.sender];

This line gets the Voter information for the person calling the function.

proposals[proposal].voteCount = proposals[proposal].voteCount + sender.value;

}

The vote count of the specified proposal is increased by the voting value of the person calling the function.

```
// functions for showing results

//1. function that shows the winning proposal by integer

function winningProposal() public view returns (uint winningProposal_) {

uint winningVoteCount = 0;
for(uint i = 0; i < proposals.length; i++ ) {
if(proposals[i].voteCount > winningVoteCount) {
winningVoteCount = proposals[i].voteCount;
winningProposal_ = i;
}
}
}
```

```
//2. function that shows the winner by name

function winningName() public view returns (bytes32 winningName_) {

winningName_ = proposals[winningProposal()].name;

}

}
```

1. winningProposal() function:

   • Line 1: This line is defining a function called winningProposal that anyone can view, and when called, it will return an unsigned integer (uint), which represents the index of the winning proposal in the proposals array.
   • Line 3: Here, we're creating a variable called winningVoteCount and setting it initially to 0. This variable will be used to keep track of the highest number of votes we've seen so far as we go through all the proposals.
   • Line 4: This line starts a for loop that will go through each proposal in the proposals array. The i variable represents the current index of the proposal we're looking at.
   • Line 5-8: These lines are inside the for loop. They check if the current proposal (proposals[i]) has more votes (voteCount) than our current winningVoteCount. If it does, then this proposal is our new front-runner for the winner, so we update winningVoteCount to this higher vote count, and we also update winningProposal_ to be the current index I.

   • Line 10: This ends the for loop. After this loop finishes, winningProposal_ will hold the index of the proposal with the most votes, and that's what gets returned by the function.

2. winningName() function:

- Line 13: This line is defining a function called winningName that anyone can view, and when called, it will return a bytes32 which represents the name of the winning proposal.
- Line 15: This line calls the winningProposal() function to get the index of the winning proposal, and then it uses this index to look up the name of the winning proposal in the proposals array (proposals[winningProposal()].name). This name is then stored in the winningName_ variable.
- Line 17: This ends the winningName function. The function returns the value of winningName_, which is the name of the proposal that got the most votes.

THANK YOU!