### **Technical Documentation: Voting Smart Contract**

### **Reference:** [**docs.soliditylang.org**](https://docs.soliditylang.org/en/v0.8.28/)

### **As per 17-11-2024**

### **1. Basics and Setup**

solidity

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract Voting {

address public superAdmin; // Primary administrator

uint256 public candidateCount;

uint256 public voterCount;

uint256 public votingStart;

uint256 public votingEnd;

bool public isPaused;

#### **Explanation:**

1. **SPDX License Identifier**:
   * // SPDX-License-Identifier: MIT: Specifies the license for the contract. This is important for legal purposes. MIT indicates this is open source contract.
2. **Pragma Directive**:
   * pragma solidity ^0.8.0: Tells the compiler to use Solidity version 0.8.0 or higher.
3. **Contract Declaration**:
   * contract Voting: Defines a smart contract named Voting.
4. **State Variables**:
   * superAdmin: The Ethereum address of the primary administrator (the person who deployed the contract).
   * candidateCount: Keeps track of how many candidates are added to the election.
   * voterCount: Keeps track of how many voters are registered.
   * votingStart & votingEnd: Store the start and end times of the voting period.
   * isPaused: A boolean variable to pause or resume contract activities.

### **2. Structs (Data Models)**

solidity

struct Candidate {

uint256 candidateId;

string name;

string slogan;

uint256 votes;

}

struct ElectionDetails {

string adminName;

string adminEmail;

string adminTitle;

string electionTitle;

string organizationTitle;

uint256 maxVotesPerCandidate;

}

struct Voter {

address voterAddress;

string name;

string phone;

bool isVerified;

bool hasVoted;

bool isRegistered;

}

#### **Explanation:**

**Structs** are custom data types used to group related variables together. Here, we have three structs:

1. **Candidate**:
   * candidateId: A unique ID for the candidate.
   * name: The candidate’s name.
   * slogan: A short slogan for the candidate.
   * votes: Tracks how many votes this candidate has received.
2. **ElectionDetails**:
   * Stores information about the election, such as:
     + Admin details (adminName, adminEmail, adminTitle).
     + Election details (electionTitle, organizationTitle).
     + maxVotesPerCandidate: Sets a limit on how many votes a candidate can receive.
3. **Voter**:
   * voterAddress: The Ethereum address of the voter.
   * name and phone: Personal details of the voter.
   * isVerified: Indicates whether the voter has been approved to vote.
   * hasVoted: Tracks whether the voter has already voted.
   * isRegistered: Indicates if the voter is registered.

### **3. Mappings (Data Storage)**

solidity

ElectionDetails public Election;

mapping(address => Voter) private Voters;

mapping(uint256 => Candidate) public Candidates;

mapping(address => bool) public admins;

#### **Explanation:**

**Mappings** are like key-value pairs used for storing data.

1. **Election**:
   * Stores all the details of the election.
2. **Voters**:
   * Maps a voter's Ethereum address (address) to their Voter struct (details like name, phone, etc.).
3. **Candidates**:
   * Maps a unique candidate ID (uint256) to their Candidate struct.
4. **Admins**:
   * Maps an address (address) to a boolean value indicating whether that address is an admin (true) or not (false).

### **4. Events**

solidity

event CandidateAdded(uint256 candidateId, string name, string slogan);

event VoterRegistered(address voterAddress, string name);

event VoterVerified(address voterAddress, bool status);

event VoteCast(address voter, uint256 candidateId);

event ElectionStarted(uint256 startTime, uint256 endTime);

event ElectionEnded(uint256 endTime);

event WinnerDeclared(uint256 candidateId, string name, uint256 votes);

event AdminAdded(address adminAddress);

event AdminRemoved(address adminAddress);

event ElectionDetailsUpdated(string field, string newValue);

event ContractPaused(bool isPaused);

#### **Explanation:**

* **Events**: Used to log activity on the blockchain (like a receipt of what happened). They can be accessed by external tools (like front-end apps or explorers).

For example:

* CandidateAdded: Logs when a new candidate is added.
* VoterRegistered: Logs when a voter registers.
* VoteCast: Logs when someone casts a vote.
* ElectionStarted and ElectionEnded: Track the start and end of voting.

### **5. Modifiers (Access Control)**

solidity

modifier isSuperAdmin() {

require(msg.sender == superAdmin, "Only super admin allowed");

\_;

}

modifier isAdmin() {

require(admins[msg.sender], "Only admin allowed");

\_;

}

modifier onlyWhenVotingActive() {

require(block.timestamp >= votingStart && block.timestamp <= votingEnd, "Voting is not active");

\_;

}

modifier onlyWhenVotingEnded() {

require(block.timestamp > votingEnd, "Voting has not ended");

\_;

}

modifier notPaused() {

require(!isPaused, "Contract is paused");

\_;

}

#### **Explanation:**

**Modifiers** are reusable pieces of code that act as checks before running a function.

1. **isSuperAdmin**:
   * Ensures the caller of the function is the super admin (person who deployed the contract).
2. **isAdmin**:
   * Ensures the caller is an admin.
3. **onlyWhenVotingActive**:
   * Ensures the voting period is active when calling a function.
4. **onlyWhenVotingEnded**:
   * Ensures voting is over before certain actions (like declaring a winner).
5. **notPaused**:
   * Ensures the contract is not paused when performing an action.

### **6. Constructor**

solidity

constructor() {

superAdmin = msg.sender;

admins[msg.sender] = true;

}

#### **Explanation:**

1. **Constructor**: Runs once when the contract is deployed.
2. msg.sender: The Ethereum address that deployed the contract.
3. superAdmin: Set to the deployer's address.
4. The deployer is also made an admin (admins[msg.sender] = true).

### **7. Admin Management**

#### **Add Admin**

solidity

function addAdmin(address adminAddress) external isSuperAdmin {

require(!admins[adminAddress], "Already an admin");

admins[adminAddress] = true;

emit AdminAdded(adminAddress);

}

* **What it does**: Allows the super admin to add other admins.
* **Checks**:
  + The caller must be the super admin.
  + The address must not already be an admin.
* **Logs**: Emits an AdminAdded event.

#### **Remove Admin**

solidity

function removeAdmin(address adminAddress) external isSuperAdmin {

require(admins[adminAddress], "Not an admin");

admins[adminAddress] = false;

emit AdminRemoved(adminAddress);

}

* **What it does**: Allows the super admin to remove admins.
* **Checks**:
  + The caller must be the super admin.
  + The address must currently be an admin.
* **Logs**: Emits an AdminRemoved event.

### **8. Election Management**

#### **Set Election Details**

solidity

function setElectionDetails(

string memory adminName,

string memory adminEmail,

string memory adminTitle,

string memory electionTitle,

string memory organizationTitle,

uint256 maxVotes

) external isAdmin {

Election = ElectionDetails({

adminName: adminName,

adminEmail: adminEmail,

adminTitle: adminTitle,

electionTitle: electionTitle,

organizationTitle: organizationTitle,

maxVotesPerCandidate: maxVotes

});

}

* **What it does**: Allows admins to set/update election details (admin details, titles, and vote limits).

#### **Start Election**

solidity

function startElection(uint256 durationInMinutes) external isAdmin notPaused {

require(votingStart == 0 || block.timestamp > votingEnd, "Previous election ongoing");

votingStart = block.timestamp;

votingEnd = votingStart + (durationInMinutes \* 1 minutes);

emit ElectionStarted(votingStart, votingEnd);

}

* **What it does**: Starts a new election for a set duration.
* **Checks**:
  + No other election is ongoing.
  + The contract is not paused.
* **Logs**: Emits an ElectionStarted event.

#### **End Election**

solidity

function endElection() external isAdmin notPaused {

require(block.timestamp >= votingEnd, "Voting time not over");

votingStart = 0;

votingEnd = 0;

emit ElectionEnded(block.timestamp);

}

* **What it does**: Ends an election manually.
* **Checks**:
  + Voting time must be over.
  + The contract must not be paused.
* **Logs**: Emits an ElectionEnded event.

### **9. Candidate Management**

#### **Add Candidate**

solidity

function addCandidate(string memory name, string memory slogan) external isAdmin notPaused {

require(bytes(name).length > 0, "Candidate name required");

for (uint256 i = 0; i < candidateCount; i++) {

require(keccak256(abi.encodePacked(Candidates[i].name)) != keccak256(abi.encodePacked(name)), "Duplicate candidate name");

}

Candidates[candidateCount] = Candidate(candidateCount, name, slogan, 0);

emit CandidateAdded(candidateCount, name, slogan);

candidateCount++;

}

* **What it does**: Adds a new candidate.
* **Checks**:
  + Candidate name is not empty.
  + Candidate name is unique.
  + The contract must not be paused.
* **Logs**: Emits a CandidateAdded event.

### **10. Voter Management**

#### **Register Voter**

solidity

function registerVoter(string memory name, string memory phone) external notPaused {

require(!Voters[msg.sender].isRegistered, "Already registered");

Voters[msg.sender] = Voter(msg.sender, name, phone, false, false, true);

voterCount++;

emit VoterRegistered(msg.sender, name);

}

* **What it does**: Allows a user to register as a voter.
* **Checks**:
  + The user must not already be registered.
  + The contract must not be paused.
* **Logs**: Emits a VoterRegistered event.

#### **Verify Voter**

solidity

function verifyVoter(address voterAddress) external isAdmin notPaused {

require(Voters[voterAddress].isRegistered, "Voter not registered");

Voters[voterAddress].isVerified = true;

emit VoterVerified(voterAddress, true);

}

* **What it does**: Verifies a voter to allow them to vote.
* **Checks**:
  + The voter must be registered.
  + The contract must not be paused.
* **Logs**: Emits a VoterVerified event.

### **11. Voting**

#### **Cast Vote**

solidity

function vote(uint256 candidateId) external onlyWhenVotingActive notPaused {

require(Voters[msg.sender].isVerified, "Not verified");

require(!Voters[msg.sender].hasVoted, "Already voted");

require(candidateId < candidateCount, "Invalid candidate");

require(Candidates[candidateId].votes < Election.maxVotesPerCandidate, "Vote limit reached");

Candidates[candidateId].votes++;

Voters[msg.sender].hasVoted = true;

emit VoteCast(msg.sender, candidateId);

}

* **What it does**: Lets a verified voter vote for a candidate.
* **Checks**:
  + Voting period must be active.
  + The voter is verified and hasn’t already voted.
  + The candidate ID is valid.
  + The candidate hasn’t exceeded the maximum votes.
* **Logs**: Emits a VoteCast event.

### **12. Utility Functions**

#### **Pause/Unpause Contract**

solidity

function pauseContract(bool pause) external isSuperAdmin {

isPaused = pause;

emit ContractPaused(pause);

}

* **What it does**: Allows the super admin to pause or resume the contract.
* **Logs**: Emits a ContractPaused event.

#### **Declare Winner**

solidity

function declareWinner() external view onlyWhenVotingEnded returns (uint256 winnerId, string memory winnerName, uint256 maxVotes) {

uint256 maxVoteCount = 0;

uint256 winnerIdx = 0;

for (uint256 i = 0; i < candidateCount; i++) {

if (Candidates[i].votes > maxVoteCount) {

maxVoteCount = Candidates[i].votes;

winnerIdx = i;

}

}

return (winnerIdx, Candidates[winnerIdx].name, maxVoteCount);

}

* **What it does**: Returns the winner details after the election ends.
* **Checks**:
  + Voting must have ended.

### **Key Takeaways:**

1. **Roles**:
   * Super Admin: Full control.
   * Admin: Manages election, candidates, and voter verification.
   * Voters: Register, verify, and vote.
2. **Core Functions**:
   * Admins manage elections and candidates.
   * Voters register, verify, and vote.
   * The contract can be paused by the super admin.

### **What is emit in Solidity?**

In Solidity, the emit keyword is used to trigger (or log) an **event**. Events are special constructs in Solidity that allow contracts to log data on the blockchain. This data can then be accessed by external users, applications (like frontends), or other systems to monitor what happened on the blockchain.

### **What is event in Solidity?**

An event is a way for a smart contract to log data when something happens. Think of it like a print statement in other programming languages, but instead of printing to a console, it writes to the blockchain for others to see. Events are lightweight and cost-efficient compared to storing data directly on the blockchain.

event CandidateAdded(uint256 candidateId, string name, string slogan);

* This declares an **event** named CandidateAdded.
* It has three parameters:
  + candidateId: The unique ID of the candidate.
  + name: The name of the candidate.
  + slogan: The slogan of the candidate.

The event doesn't do anything by itself; it's like defining a data structure for logging specific actions.

solidity

emit CandidateAdded(candidateCount, name, slogan);

* This **emits** the CandidateAdded event.
* It logs the following information:
  + candidateCount: The ID of the newly added candidate.
  + name: The name of the candidate being added.
  + slogan: The candidate's slogan.

### **How Events and emit Work Together**

1. **Event Declaration**:
   * You declare an event in the contract to specify what kind of information can be logged.
2. **Event Emission**:
   * When a specific action happens (e.g., adding a new candidate), the contract uses emit to log the event.
3. **Accessing the Event**:
   * External applications or users can listen for this event using tools like **Web3.js** or **Ethers.js** to react to it. For example:
     + A front-end application might display a "Candidate Added" message when this event is detected.

### **Example Workflow**

#### **Solidity Contract:**

solidity

event CandidateAdded(uint256 candidateId, string name, string slogan);

function addCandidate(string memory name, string memory slogan) external {

Candidates[candidateCount] = Candidate(candidateCount, name, slogan, 0);

emit CandidateAdded(candidateCount, name, slogan);

candidateCount++;

}

#### **Logs Triggered by emit:**

When you call addCandidate("Alice", "A brighter future"), the blockchain logs:

yaml

Event: CandidateAdded

- candidateId: 0

- name: Alice

- slogan: A brighter future

#### **Front-End Application:**

Using a library like **Web3.js**, you can listen for this event:

javascript

contractInstance.events.CandidateAdded()

.on("data", (event) => {

console.log("New Candidate Added:", event.returnValues);

});

### **Why Use Events?**

* **Transparency**: Logs important actions publicly on the blockchain.
* **Efficiency**: Events are cheaper than storing the same data in the blockchain's state.
* **Interactivity**: External applications can react to events in real time, enabling dynamic updates (e.g., showing the list of candidates).

In Solidity, the keyword external is one of the four visibility specifiers for functions. Here's what it means:

### **What Does external Do?**

1. **Function Visibility**:
   * external means the function can only be called from **outside** the contract.
   * It cannot be called internally (i.e., from other functions inside the same contract) unless you use this.functionName() explicitly.
2. **Use Case**:
   * external is typically used for functions that interact with external users or other contracts.
   * It optimizes gas costs slightly when passing large amounts of data because arguments are read directly from calldata (more efficient than memory).

### **How external Works in endElection**

solidity

function endElection() external isAdmin notPaused {

require(block.timestamp >= votingEnd, "Voting time not over");

votingStart = 0;

votingEnd = 0;

emit ElectionEnded(block.timestamp);

}

#### **Key Details:**

1. **Who Can Call It?**
   * Only external entities (e.g., users or other contracts) can call endElection().
2. **What Happens?**
   * The function checks that the current time (block.timestamp) is greater than or equal to votingEnd.
   * If the condition is true, it resets votingStart and votingEnd to 0, effectively ending the election.
   * The ElectionEnded event is emitted to signal that the election has been terminated.
3. **Modifiers Applied**:
   * isAdmin: Ensures only an admin can call this function.
   * notPaused: Ensures the contract is not paused.

### **Difference Between Visibility Specifiers**

| **Visibility** | **Who Can Call?** | **Usage** |
| --- | --- | --- |
| public | Everyone (both internally and externally). | Functions callable internally or by external users/contracts. |
| external | Only external users or contracts. | Optimized for external calls; can't be called internally without this. keyword. |
| internal | Only within the contract and derived contracts. | Used for logic that should only be accessible within inheritance chains. |
| private | Only within the contract where it’s defined. | Completely restricted to the defining contract; not even derived contracts can access it. |

### **Why Use external for endElection?**

* It makes sense because endElection is designed to be triggered by an **admin** interacting with the contract, not by internal functions.