Blockchain & Solidity Lab1 – Voting dApp Development

S2BC

Lab 1 - Developing Ethereum Smart Contracts

- **BUILD** / TEST / INTEGRATE / RUN

This Hands on Module will build up of 4 Labs:

1. Developing Ethereum Smart Contracts [**BUILD**]
2. Test Ethereum Smart Contracts [TEST]
3. Integrate Smart Contracts with Web3 and establish and run your 1st dApp [INTEGRATE]
4. Run a dApp and considering next steps to create a possible contribution [RUN]

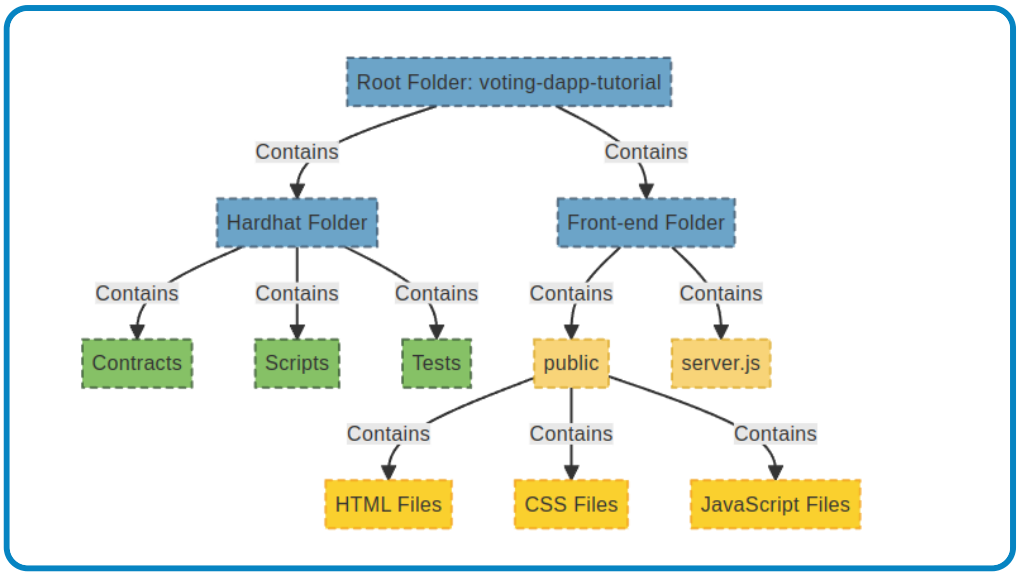
Prerequisites

To make the most out of this lab, a basic understanding of programming concepts and familiarity with JavaScript will be beneficial. However, even if you're new to blockchain development, we'll guide you through each step.

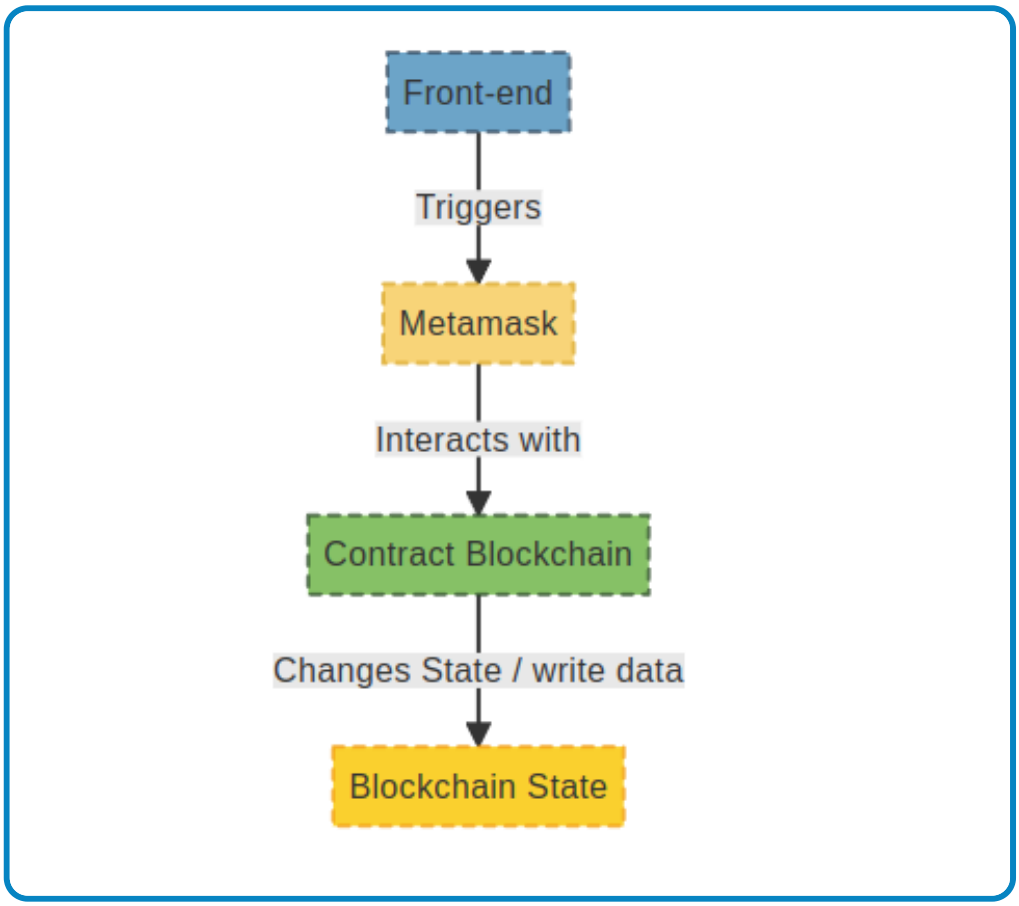
Let's dive in and get started with the first part of our journey: Developing Ethereum Smart Contracts!

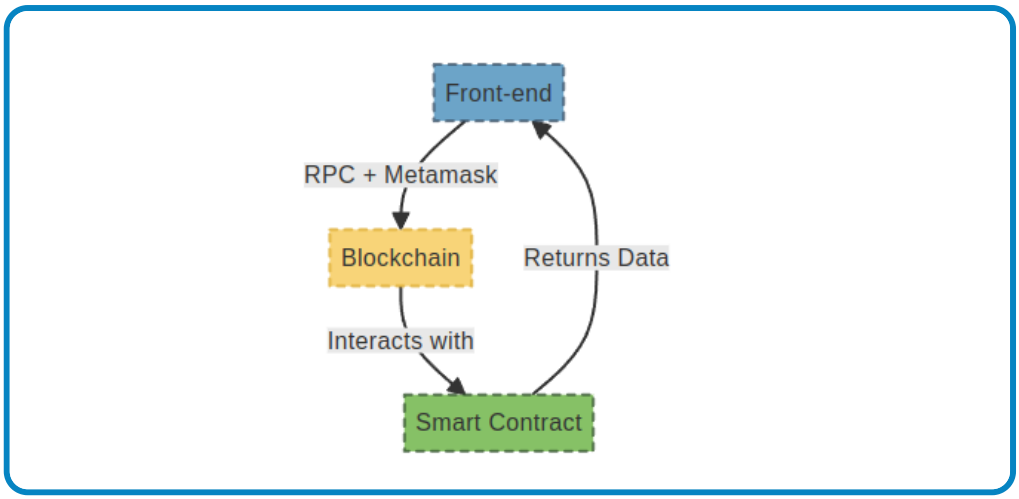
---

dApp structure Overview



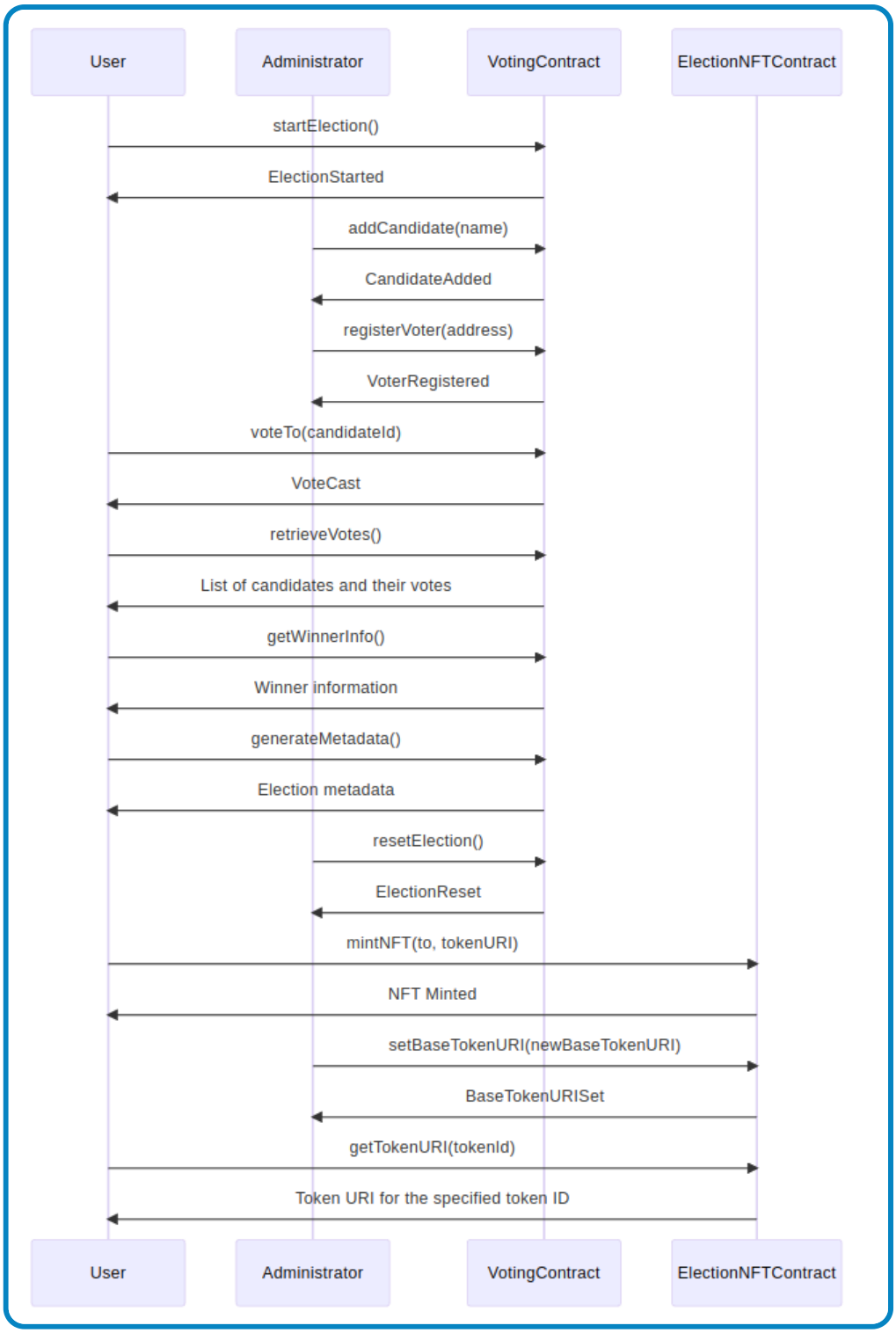
Flow overview





Overview contract Voting.sol

Diagram of interactions



DEVELOPING ETHEREUM SMART CONTRACTS

We will start this hands-on development course with a smart contract that aims to solve a problem of traditional centralized voting platforms.

What is the problem?

Traditional centralized voting systems pose significant security risks and lack transparency. It's challenging to provide concrete evidence of their security measures. Moreover, a central authority can potentially manipulate results, eroding trust in the electoral process.

What is the solution?

Integrating cryptocurrencies and smart contracts offers a robust solution to address these challenges. This approach empowers supporters by involving them directly in the development and decision-making processes of the projects they choose to back, ensuring transparency and security in the voting system.

How will it work?

- **Administrator Role:** An administrator has the authority to initiate an election by specifying candidates, setting a defined time duration, and registering eligible voters.
- **Voter Registration:** Registered voters are allowed to cast a single vote within the election period.
- **Dynamic Updates:** The administrator retains the flexibility to add candidates and adjust the voting duration as necessary, even after the election has started.
- **Timer Expiry:** Once the timer reaches zero, voting is automatically closed, preventing further participation.
- **End Election:** The administrator can formally conclude the election process, signaling that no more votes can be cast.
- **Result Minting:** After concluding the election, the administrator can initiate the "minting" process, creating unique NFTs for each voter. These NFTs will serve as proof of the winner of the election.
- **Election Reset:** The administrator can choose to reinitialize the election, starting a fresh cycle for a new round of voting.
- **Result Verification:** The NFT collection acts as an immutable record, providing a time-stamped, verifiable history of all transactions within the voting system.

This comprehensive system ensures transparency, security, and integrity throughout the entire voting process, bolstered by the use of NFTs to memorialize the results and actions taken.

ELEMENTS OF THE APP

Variables:

- `electionNFTContract`: Address of the ElectionNFT contract.
- `electionID`: Unique identifier for each election.
- `candidates`: Array holding candidate details, including their ID, name, and the number of votes received.
- `owner`: Address of the contract owner.
- `voters`: Mapping of voter addresses to their voting status.
- `eligibleVoters`: Mapping of addresses eligible to vote.
- `ListOfVoters`: Internal list of addresses who have participated in the election.
- `ListOfVotersEligible`: Internal list of addresses eligible to vote.
- `votingStartTimeStamp`: Timestamp when the voting period starts.
- `votingEndTimeStamp`: Timestamp when the voting period ends.
- `electionStarted`: Status indicating if an election is in progress.

Functions:

- `onlyOwner`: Modifier restricting certain functions to be executed only by the contract owner.
- `electionOnGoing`: Modifier ensuring that an election is currently in progress.
- `startElection(_candidates, _votingDuration)`: Function to initiate an election, specifying candidates and duration.
- `voterStatus(_voter)`: Function to check if a voter has already cast their vote.
- `voteTo(_id)`: Function for voters to cast their votes.
- `retrieveVotes()`: Function to get the number of votes received by each candidate.
- `electionTimer()`: Function to monitor the remaining time for the ongoing election.
- `checkElectionPeriod()`: Function to verify if the election period is still ongoing.
- `resetAllVoterStatus()`: Function to reset the status of all voters.
- `resetElection()`: Function to completely reset the entire election process.
- `endElection()`: Function to conclude an ongoing election.
- `removeCandidate(_candidateId)`: Function to remove a candidate from the list.
- `removeAllCandidates()`: Function to remove all candidates.
- `transferOwnership(newOwner)`: Function to transfer ownership of the contract.
- `changeElectionDuration(_newDuration)`: Function to change the duration of the ongoing election.
- `addCandidate(_name)`: Function to add a new candidate.
- `registerVoter(_eligible_voter)`: Function to register a voter.
- `registerVoters(_eligible_voters)`: Function to register multiple voters.
- `mintResultNFTs(_tokenURI)`: Function to mint NFTs for election results.
- `mintResult(_participant, _tokenURI)`: Function to mint an NFT for a participant.
- `setElectionNFTContract(_electionNFTContract)`: Function to set the address of the ElectionNFT contract.
- `getWinnerInfo()`: Function to get information about the winner of the election.
- `generateMetadata()`: Function to generate metadata for the election results.

A BRIEF INTRODUCTION TO SOME PROGRAMMING BASICS (SKIP, IF YOU HAVE SOME CODING EXPERIENCE)

Introduction to Command Line

Chances are, you're working on a computer powered by a Linux, Mac, or Windows Operating System. Mastering the command line is pivotal for the tasks ahead. While you may write your code in editors like Microsoft Visual Studio Code or use Remix for Solidity smart contracts, the command line is indispensable for executing and testing your programs, even on the Morpheus Labs Platform. It's a versatile tool, with one of its primary roles being to leverage the npm package manager for installing necessary packages.

If you're using Windows, you might want to consider using Ubuntu Linux on Windows Subsystem for Linux (WSL) for a smoother experience. WSL provides a Linux environment within your Windows system, greatly benefiting blockchain development.

- **For Linux**: Open the terminal. (Shortcut: CTRL+ALT+T)
- **For Windows (with Ubuntu on WSL)**: Launch the WSL terminal.
- **For Mac**: Utilize the terminal.
- **On Morpheus Labs SEED BPaaS**: Access the terminal. (Ubuntu docker)

Navigating your file system

When navigating your file system, keep these commands in mind:

- `cd ..`: Move up one directory level.
- `cd <folder name>`: Enter a specific folder.
- `ls`: List files and folders in the current directory.
- `ls -a`: List all files and folders, including hidden ones.
- `cat <file name>`: Display the contents of a file.
- `touch <file name>`: Create a new file.
- `mkdir <folder name>`: Create a new folder.
- `rm <file name>`: Remove a file (be cautious, this action is irreversible).
- `rm -r <folder name>`: Remove a folder and its contents.
- `mv <source> <destination>`: Move or rename files and folders.
- `cp <source> <destination>`: Copy files or folders.
- `pwd`: Display the current working directory.
- `clear`: Clear the terminal screen.
- `history`: Display a list of recently used commands.
- `grep <pattern> <file>`: Search for a specific pattern in a file.
- `chmod <permissions> <file>`: Change the permissions of a file.
- `nano <file name>`: Open the Nano text editor to edit a file.
- `wget <URL>`: Download a file from the internet.
- `curl <URL>`: Transfer data from or to a server.

Once you're in the desired folder, you can run your program by typing `<program name>` in the command line. NPM, a potent package manager, will be your go-to tool for installing, updating, and removing packages like Hardhat, ethers, Express...

Introduction to Basic Programming Concepts

Variables and Constants

In programming, variables and constants are essential components. They are used to store and manipulate data.

- **Variables**: These are containers that can hold various types of data, such as strings, integers, or booleans (true/false). The value of a variable can be changed during the execution of a program.

```
let name = "John"; // Here, 'name' is a variable storing a string value "John".
let age = 30;      // 'age' is a variable storing an integer value 30.
let isStudent = true; // 'isStudent' is a variable storing a boolean value true.
```

- **Constants**: Unlike variables, constants hold fixed values that do not change during the execution of a program.

```
const PI = 3.14; // Here, 'PI' is a constant with a fixed value of 3.14.
const MAX_SIZE = 100; // 'MAX_SIZE' is a constant with a fixed value of 100.
```

Data Structures

Understanding data structures is crucial for efficient data management in programming. Here are some common data structures:

- **Array**: An array is defined within square brackets `[]`. It can hold multiple values, each separated by commas.

```
let numbers = [1, 2, 3, 4, 5]; // 'numbers' is an array containing five integers.
let names = ["Alice", "Bob", "Charlie"]; // 'names' is an array containing three strings.
```

- **Object (Dictionary)**: An object is defined within curly braces `{}`. It consists of key-value pairs, where each piece of data is mapped to a specific value.

```
let person = {
    name: "John",
    age: 30,
    isStudent: false
}; // 'person' is an object with name, age, and isStudent as keys and their respective values.
```

Functions

Functions play a vital role in programming. They take input values, process them, and return a result. Functions are defined with a name, input parameters within parentheses `()`, and the code to be executed within curly braces `{}`.

They can be called by using the function name and providing the necessary input values.

```
function addNumbers(num1, num2) {
    return num1 + num2;
}

let result = addNumbers(5, 3); // 'result' will be 8.
```

Classes

Classes are fundamental to object-oriented programming (OOP). They encapsulate data and functions into a single unit. A class can contain multiple functions that define the behavior of objects created from that class.

To Set Up the Development Environment MORPHEUSLABS BPAAS SEED

Configure repositery :

- https://docs.morpheuslabs.io/docs/configuration

Create workspace morpheus doc page :

- https://docs.morpheuslabs.io/docs/configure-a-workspace#section-create-a-workspace

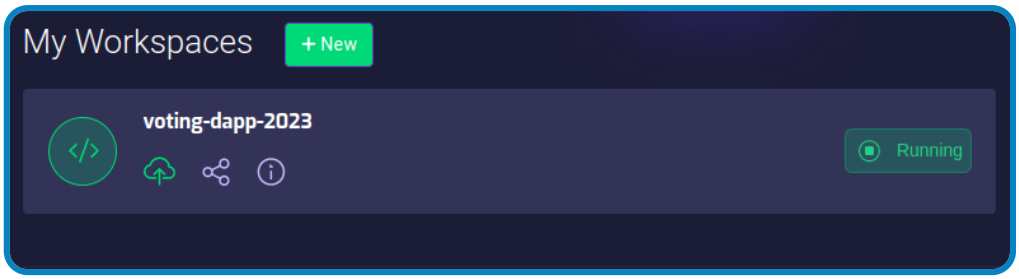
Create a blockchain network :

- https://docs.morpheuslabs.io/docs/blockchain-networks
- We will use **Sepolia** ChainID 11155111(0xaa36a7) Currency ETH

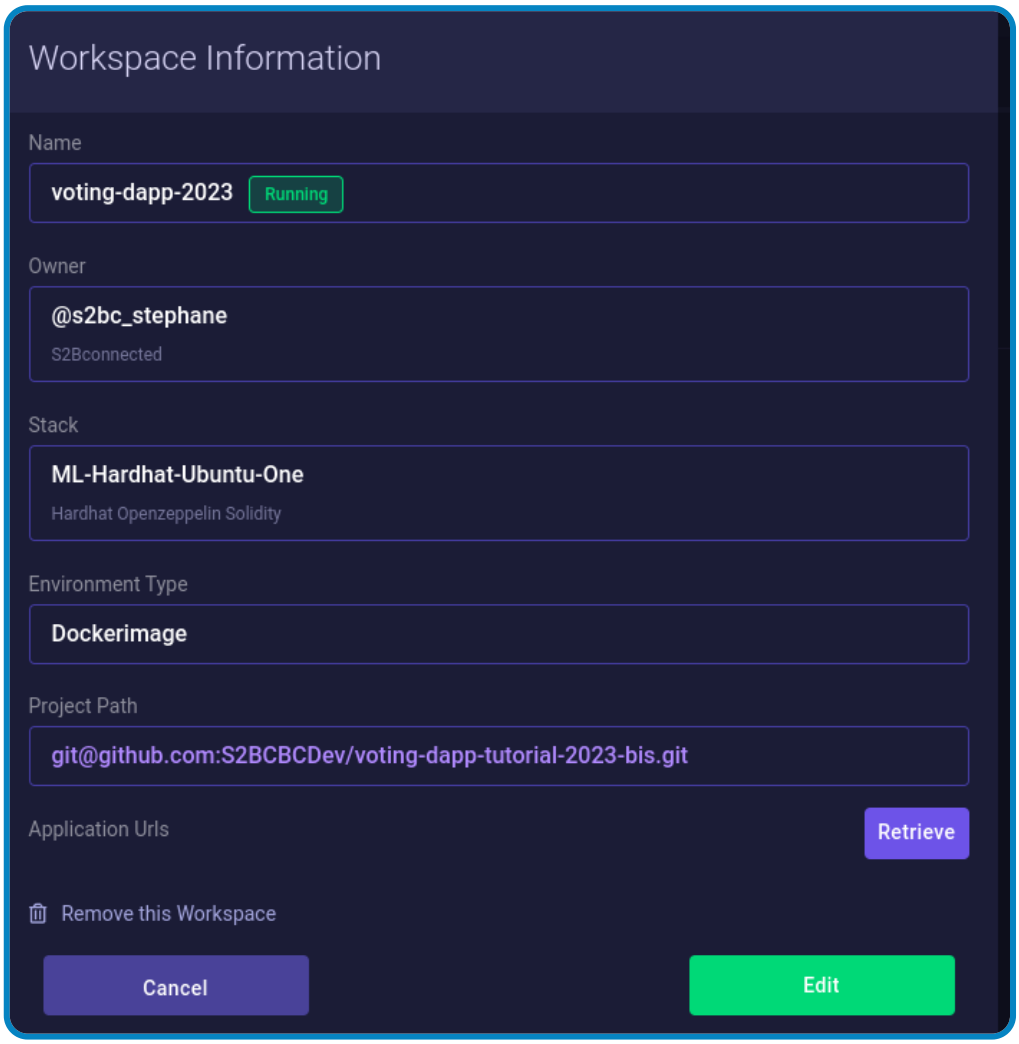
Launch

- click on this info icon to get to the VSCode interface of morpheus by retrieving the workspace url.

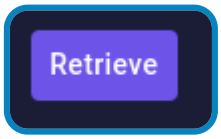
into this



You will get to the Workspace information board:



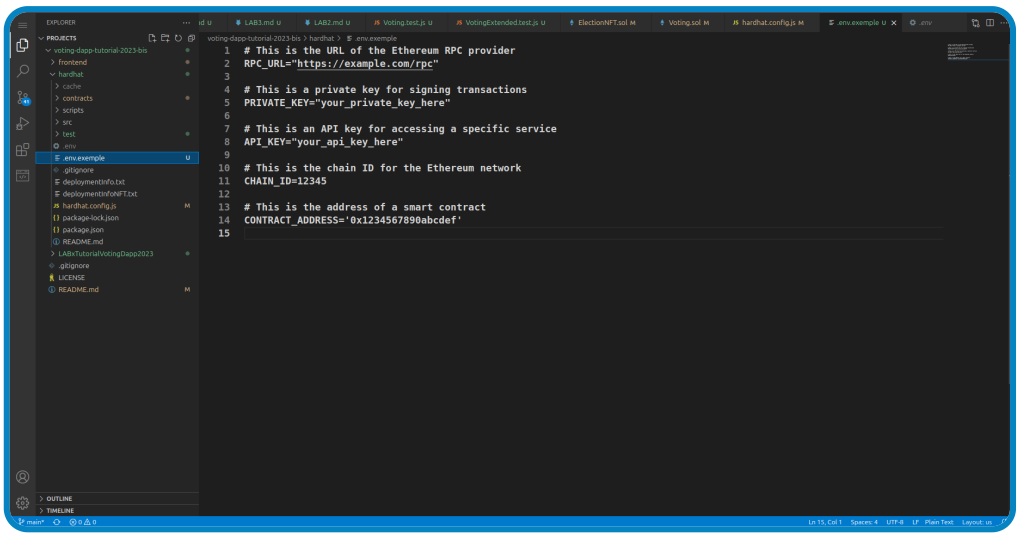
Click on "Retrive" button:



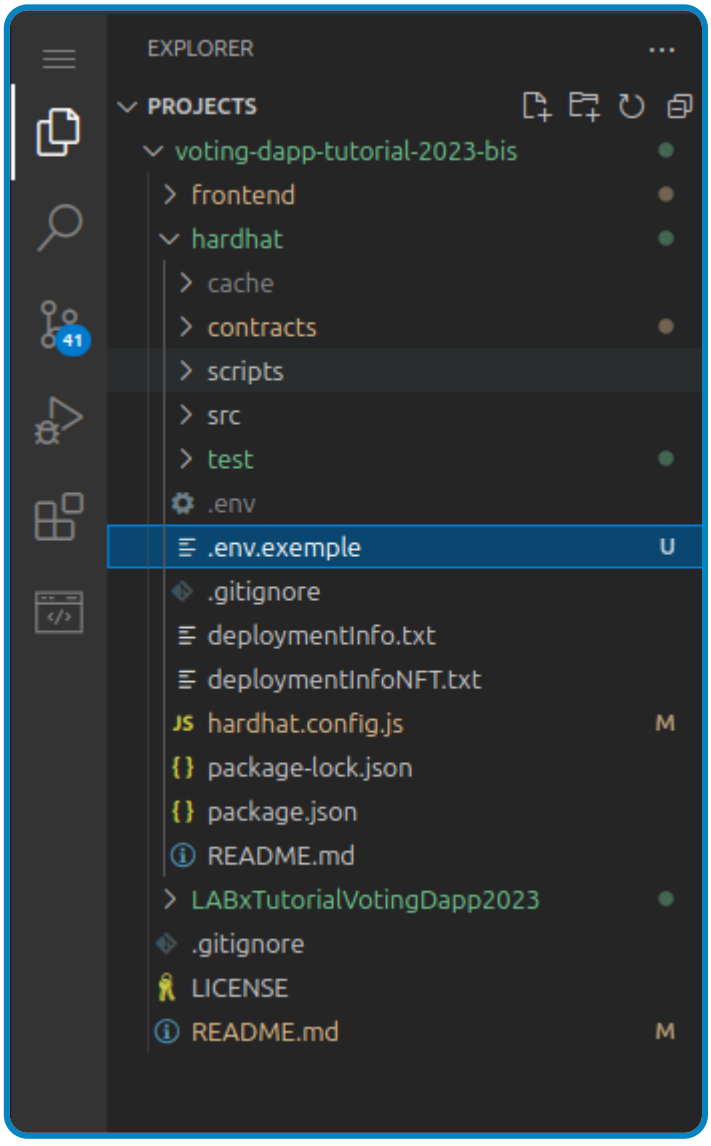
It should transform into:

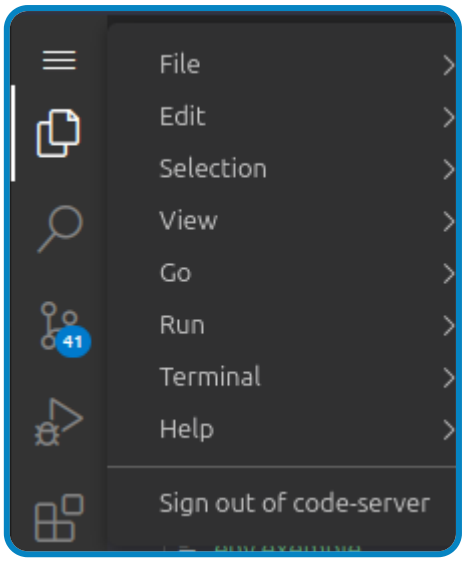
Click on Workspace Url to get to your IDE:

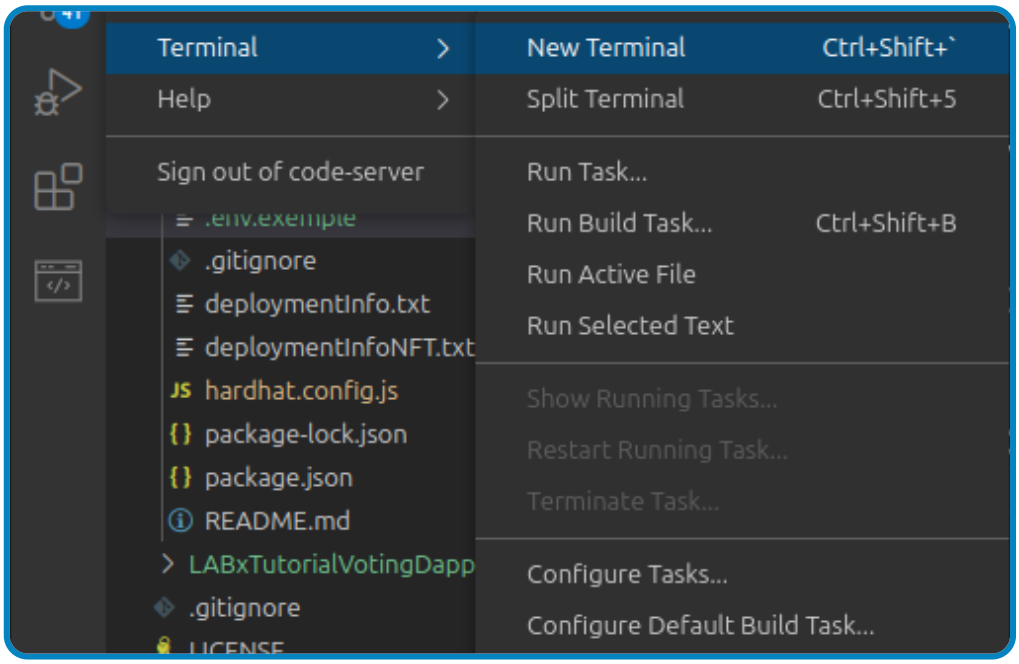
You should have your IDE open now:



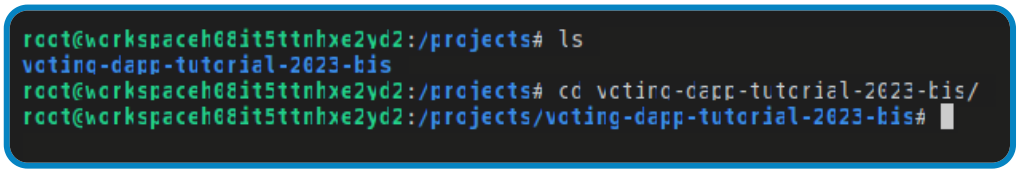
Go to your left menu and find for new Terminal:







Check what is in your project and then change directory to your app or go next step by setting up your main directory:



Set up the main dapp repository

To get started with our decentralized voting application tutorial, we'll first set up the main repository. Follow these steps:

Step 1: Create a Folder

Open a terminal and execute the following commands to create a new folder for our project:

```
mkdir voting-dapp-tutorial
cd voting-dapp-tutorial
touch README.md
```

```
git init
git add .
git commit -m "Initial commit"
```

This will create a new directory named `voting-dapp-tutorial` and a `README.md` file, which will serve as the main documentation for our project. Additionaly this will initialise Git for our project.

Step 2: Install HardHat

Next, we'll install HardHat, a popular development environment for Ethereum. HardHat provides a set of tools that make it easy to compile, deploy, and test smart contracts. Execute the following command to install HardHat:

```
mkdir hardhat    // this will create a hardhat folder
cd hardhat       // this will make you move into hardhat folder
npx hardhat init // this will initialise hardhat and create a folder
structure
```

This command will fetch and set up the HardHat environment in our project directory.

In the dialogue box of hardhat:

- Create a JavaScript project - YES
- Confirm root folder location - ENTER
- Add gitignore - YES
- install dependencies with npm (hardhat @nomicfoundation/hardhat-toolbox)- YES

Verify HardHat installation

```
npx hardhat --version
```

With these initial steps completed, we're now ready to proceed with the creation of our smart contract for the decentralized voting system. Let's move on to the next section!

Creating the `Voting.sol` smart contract with Solidity

In this section, we'll guide you through the process of creating the `Voting.sol` file, which will house the smart contract for our decentralized voting application. This Solidity file will define the behavior and rules of our voting system on the Ethereum blockchain.

Step 0: Create a file named Voting.sol in hardhat/contract folder

With vscode web interface:

- navigate to voting-dapp-tutorial-2023/hardhat/contracts
- create a new file named Voting.sol

or with the terminal:

```
cd hardhat/contracts
touch Voting.sol
```

Step 1: Set the Compiler Version and Import Dependencies

- At the top of the file, specify the License type (SPDX-License-Identifier: UNLICENSED)
- Then specify the compiler version pragma solidity (pragma solidity ^0.8.19;)
- Then import any necessary dependencies. In our case, we're importing the `ElectionNFT.sol` (not yet created) contract, which will handle the creation of unique NFTs for each voter.
- Finaly write the name of the contract and open curly braces {}

Your file should looks like this:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.19; // Specifies the compiler version

import "./ElectionNFT.sol"; // Import the future NFT contract

contract Voting {
    // ... (content of the Voting.sol contract will be added later)
}
```

Step 2: Define the Contract Structure

Within the `Voting.sol` file, we'll outline the fundamental structure of our smart contract. This involves declaring variables, creating functions, and implementing modifiers. The entire content of the contract will be placed within the curly braces `{}`.

```
contract Voting {
    // Declare variables
    address public electionNFTContract;

    // Define data structures, modifiers, and events

    struct Candidate {
        uint256 id;
        string name;
        uint256 numberOfVotes;
    }

    uint256 public electionID = 0;
    Candidate[] public candidates;
    address public owner;
```

```
    // Define access modifiers
    modifier onlyOwner() {
        // Modifier code to restrict access
        require(msg.sender == owner, "Only the owner can call this
function");
        _;
    }

    modifier electionOnGoing() {
        // Modifier code to check if election is ongoing
        require(block.timestamp >= votingStartTimeStamp && block.timestamp
<= votingEndTimeStamp, "Election is not ongoing");
        _;
    }

    // Define events
    event ElectionStarted(
        address indexed owner,
        uint256 startTimestamp,
        uint256 endTimestamp
    );

    event CandidateAdded(
        uint256 indexed candidateId,
        string name
    );

    // ... (more events)

    // Define functions
    function startElection(string[] memory _candidates, uint256
_votingDuration)
        public
        onlyOwner
    {
        // Function code to start an election
        require(_candidates.length > 0, "At least one candidate is
required");
        require(electionNFTContract != address(0), "ElectionNFT contract
address not set");

        // Initialize election variables
        electionID++;
        votingStartTimeStamp = block.timestamp;
        votingEndTimeStamp = block.timestamp + _votingDuration;

        // Create candidates
        for (uint256 i = 0; i < _candidates.length; i++) {
            candidates.push(Candidate({
                id: i,
                name: _candidates[i],
                numberOfVotes: 0
            }));
```

```
            emit CandidateAdded(i, _candidates[i]);
        }

        emit ElectionStarted(owner, votingStartTimeStamp,
votingEndTimeStamp);
    }

    // ... (more functions)

}
```

In this step, we've defined the basic structure of the `Voting.sol` contract. We've declared variables for essential components, defined a `Candidate` struct, implemented access modifiers for security, and created events for key occurrences. Additionally, we've started the implementation of the `startElection` function, which initializes the election and creates candidate entries.

Step 3: Implementing the Voting System Logic part 1

In this step, we will dive into the actual implementation of the decentralized voting system. This encompasses defining essential data structures, setting up administrator roles, managing candidates, registering voters, and orchestrating the entire voting process.

Defining Data Structures

We begin by defining the necessary data structures that will facilitate the functioning of our voting system.

```
contract Voting {
    address public electionNFTContract;

    // Define data structures, modifiers, and events

    struct Candidate {
        uint256 id;
        string name;
        uint256 numberOfVotes;
    }

    uint256 public electionID = 0;
    Candidate[] public candidates;
    address public owner;
    // ... (more variables)
}
```

Here, we've declared a `Candidate` struct to represent each individual in the election. It includes an ID, a name, and a count of their received votes. Additionally, we have variables like `electionID`, `candidates`, and `owner` that will be crucial throughout the process.

Setting Access Modifiers

Access modifiers are essential for controlling who can execute certain functions. We'll use modifiers to restrict access to specific actions.

```
    modifier onlyOwner() {
        // Modifier code to restrict access
    }

    modifier electionOnGoing() {
        // Modifier code to check if election is ongoing
    }
```

The `onlyOwner` modifier ensures that certain functions can only be executed by the contract owner. The `electionOnGoing` modifier verifies if an election is currently in progress.

Declaring Events

Events in Solidity allow smart contracts to communicate information to external consumers. We'll define events to signal important occurrences in our voting system.

```
    event ElectionStarted(
        address indexed owner,
        uint256 startTimestamp,
        uint256 endTimestamp
    );
    // ... (more events)
```

For instance, the `ElectionStarted` event will be emitted when a new election commences, providing details like the owner's address and the start/end timestamps.

Implementing Functions

We'll create functions to perform crucial tasks, such as initiating an election, registering voters, casting votes, and more.

```
    function startElection(string[] memory _candidates, uint256
_votingDuration)
        public
        onlyOwner
    {
        // Function code to start an election
    }

    // ... (more functions)
```

The `startElection` function, for instance, allows the owner to start a new election by specifying candidate names and the voting duration.

Step 4: Implementing the Voting System Logic in `Voting.sol` part 2

Now that we've set up the basic structure of our decentralized voting system in `Voting.sol`, it's time to delve into the actual logic that governs the election process. In this step, we'll be adding functions and features that handle candidate management, voter registration, voting, and more.

Managing Candidates

In our smart contract, each candidate is represented by a `Candidate` struct. This struct includes essential details such as an ID, name, and the number of votes received.

```
struct Candidate {
    uint256 id;
    string name;
    uint256 numberOfVotes;
}
```

We have implemented functions to facilitate the addition and removal of candidates, allowing for flexibility even during an ongoing election. Additionally, these functions provide the capability to reset the election session, facilitating the creation of a new election process when needed.

```
function addCandidate(string memory _name) public onlyOwner electionOnGoing
{
    // ... (Function code to add a new candidate)
    emit CandidateAdded(candidates.length - 1, _name);
}

function removeCandidate(uint256 _candidateId) public onlyOwner {
    // ... (Function code to remove a candidate)
}
```

Voter Registration and Voting

The voting process involves ensuring that each voter can only cast one vote. We track voters through the `voters` mapping and verify eligibility using the `eligibleVoters` mapping.

```
mapping(address => bool) public voters;
mapping(address => bool) public eligibleVoters;

// ... (other mappings and variables)

function registerVoter(address _eligible_voter) public onlyOwner {
    // ... (Function code to register an eligible voter)
```

```
}

function voteTo(uint256 _id) public electionOnGoing {
    // ... (Function code to cast a vote)
    emit VoteCast(msg.sender, _id);
}
```

Election Management

The contract provides functionality to start, end, and reset elections. The owner can also extend the duration of an ongoing election.

```
function startElection(string[] memory _candidates, uint256
_votingDuration) public onlyOwner {
    // ... (Function code to start a new election)
    emit ElectionStarted(owner, votingStartTimeStamp, votingEndTimeStamp);
}

function endElection() public onlyOwner electionOnGoing {
    // ... (Function code to end the current election)
    emit ElectionFinished(owner);
}

function resetElection() public onlyOwner {
    // ... (Function code to reset the election)
    emit ElectionReset(owner);
}

function changeElectionDuration(uint256 _newDuration) public onlyOwner
electionOnGoing {
    // ... (Function code to change the duration of the ongoing election)
    emit ElectionDurationChanged(_newDuration);
}
```

Winner Determination and NFT Minting

The contract determines the winner based on the candidate with the highest number of votes. Additionally, it supports minting NFTs for voters and participants.

```
function getWinnerInfo() public view returns (Winner memory) {
    // ... (Function code to determine the winner)
}

function mintResultNFTs(string memory _tokenURI) public onlyOwner {
    // ... (Function code to mint NFTs for all voters)
}

function mintResult(address _participant, string memory _tokenURI) public
onlyOwner {
```

```
    // ... (Function code to mint an NFT for a specific participant)
}
```

Election Metadata

The `generateMetadata` function provides essential metadata about the election.

```
function generateMetadata() public view returns (ElectionMetadata memory) {
    // ... (Function code to generate metadata)
}
```

This marks the completion of the implementation of the voting system logic in our `Voting.sol` contract. In the next step, we'll proceed to explore additional features like handling candidate removal and examining the final results.

Step 5: Managing Candidates and Reviewing Results

Handling Candidate Removal

In a dynamic voting system, the ability to remove candidates is crucial. We'll add functionality to remove a candidate, updating the state and ensuring a fair and accurate representation of the election.

```
function removeCandidate(uint256 _candidateId) public onlyOwner
electionOnGoing {
    // ... (Function code to remove a candidate)
    emit CandidateRemoved(_candidateId);
}
```

This function allows the owner to remove a candidate during an ongoing election, triggering the `CandidateRemoved` event.

Reviewing Final Results

Once the election concludes, it's essential to review the final results. The contract provides functions to retrieve winner information and overall election metadata.

```
function getWinnerInfo() public view returns (Winner memory) {
    // ... (Function code to determine the winner)
}

function generateMetadata() public view returns (ElectionMetadata memory) {
    // ... (Function code to generate metadata)
}
```

These functions provide a comprehensive overview of the election results, including the winner and additional metadata.

Emitting Events for Transparency

To ensure transparency and accountability, we emit events throughout the process. Events such as `CandidateRemoved` and `ElectionFinished` provide a clear record of actions taken.

```
event CandidateRemoved(uint256 indexed candidateId);
event ElectionFinished(address indexed owner);
// ... (other events)
```

These events serve as a transparent log of significant occurrences within the contract.

Full Voting.sol contract code

Your smart contract Voting.sol should looks like this:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.22;

import "./ElectionNFT.sol";

contract Voting {
    address public electionNFTContract;

    // Create a structure template for each candidates
    struct Candidate {
        uint256 id; // Unique identifier for the candidate
        string name; // Name of the candidate
        uint256 numberOfVotes; // Number of votes received by the candidate
    }

    // Election ID ( is never deleted and increment at every starting new
election )
    uint256 public electionID = 0;

    // Election Title
    string public electionTitle;

    // List of all candidates
    Candidate[] public candidates;

    // This will be the owner's address
    address public owner;

    // Mapp all voter addresses
    mapping(address => bool) public voters;
```

```
    mapping(address => bool) public eligibleVoters;

    // List of voters
    address[] internal hasVotedList;
    address[] internal registeredVoters;

    // voting start and end session
    uint256 public votingStartTimeStamp;
    uint256 public votingEndTimeStamp;

    // Create an election status
    bool public electionStarted;
    bool public electionFinalised;

    // Restrict creating vote to the owner only
    modifier onlyOwner() {
        require(msg.sender == owner, "not authorized to start election");
        _;
    }

    // Check if an election is ongoing
    modifier electionOnGoing() {
        require(electionStarted, "no election yet");
        _;
    }

    // Event emitted when the election starts
    event ElectionStarted(
        address indexed owner,
        uint256 startTimestamp,
        uint256 endTimestamp,
        string title
    );

    // Event emitted when a vote is cast
    event VoteCast(address indexed voter, uint256 candidateId);

    // Event emitted when a new Candidate is added
    event CandidateAdded(uint256 indexed id, string name);

    // Event emitted when time has been added to the election period
    event ElectionDurationChanged(uint256 newDuration);

    // Event emitted when the election finishes
    event ElectionFinished(address indexed owner);

    // Event emitted when the election is reset
    event ElectionReset(address indexed owner);

    constructor() {
        // Initialize owner
        owner = msg.sender;
    }
```

```
    function startElection(
        string memory _electionTitle,
        string[] memory _candidates,
        uint256 _votingDuration
    ) public onlyOwner {
        require(electionStarted == false, "Election is currently ongoing");
        require(
            electionFinalised == false,
            "Election is not yet Reinitialized"
        );

        // Increment electionID
        electionID += 1;

        // Clear existing candidates
        while (candidates.length > 0) {
            removeCandidate(0);
        }

        // Add new candidates
        for (uint256 i = 0; i < _candidates.length; i++) {
            candidates.push(
                Candidate({id: i, name: _candidates[i], numberOfVotes: 0})
            );
        }

        // Set the election title
        electionTitle = _electionTitle;

        electionStarted = true;
        votingStartTimeStamp = block.timestamp;
        votingEndTimeStamp = block.timestamp + (_votingDuration * 1
minutes);

        emit ElectionStarted(
            owner,
            votingStartTimeStamp,
            votingEndTimeStamp,
            electionTitle
        );
    }

    // Check voter's status
    function voterStatus(address _voter)
        public
        view
        electionOnGoing
        returns (bool)
    {
        if (voters[_voter] == true) {
            return true;
        }
        return false;
    }
```

```
    // To vote function
    function voteTo(uint256 _id) public electionOnGoing {
        require(checkElectionPeriod(), "Election period has ended");
        require(
            !voterStatus(msg.sender),
            "You already voted. You can only vote once."
        );
        require(_id < candidates.length, "Invalid candidate ID");
        require(eligibleVoters[msg.sender], "You are not an eligible
voter.");

        candidates[_id].numberOfVotes++;
        voters[msg.sender] = true;

        // Add to the has voted voters list
        hasVotedList.push(msg.sender);

        emit VoteCast(msg.sender, _id);
    }

    // Get the number of votes
    function retrieveVotes() public view returns (Candidate[] memory) {
        return candidates;
    }

    // Monitor the election time
    function electionTimer() public view returns (uint256) {
        if (block.timestamp >= votingEndTimeStamp) {
            return 0;
        }
        return (votingEndTimeStamp - block.timestamp);
    }

    // Check if election period is still ongoing
    function checkElectionPeriod() public view returns (bool) {
        return electionTimer() > 0;
    }

    // Reset all voters status
    function resetAllVoterStatus() public onlyOwner {
        for (uint256 i = 0; i < hasVotedList.length; i++) {
            voters[hasVotedList[i]] = false;
        }
        delete hasVotedList;

        emit ElectionReset(owner);
    }

    // Completely resetting the entire election process
    function resetElection() public onlyOwner {
        require(!electionStarted, "Election is currently ongoing");

        // Reset registeredVoters mappings
```

```
        for (uint256 i = 0; i < registeredVoters.length; i++) {
            eligibleVoters[registeredVoters[i]] = false;
        }

        // Clear registeredVoters
        delete registeredVoters;

        // Reset voter status
        resetAllVoterStatus();

        // Reset election status and timers
        electionStarted = false;
        votingStartTimeStamp = 0;
        votingEndTimeStamp = 0;
        electionFinalised = false;

        // Remove all candidates
        removeAllCandidates();

        // Reset the election title to the default value
        electionTitle = "No title yet";

        // emit Election Reset
        emit ElectionReset(owner);
    }

    function endElection() public onlyOwner electionOnGoing {
        electionStarted = false;
        votingEndTimeStamp = block.timestamp;
        electionFinalised = true;

        emit ElectionFinished(owner);
    }

    function removeCandidate(uint256 _candidateId) public onlyOwner {
        require(_candidateId < candidates.length, "Invalid candidate ID");
        require(!electionStarted, "Election is ongoing");

        for (uint256 i = _candidateId; i < candidates.length - 1; i++) {
            candidates[i] = candidates[i + 1];
        }

        delete candidates[candidates.length - 1];
        candidates.pop();
    }

    function removeAllCandidates() public onlyOwner {
        require(!electionStarted, "Election is currently ongoing");
        while (candidates.length > 0) {
            removeCandidate(0);
        }
    }

    function transferOwnership(address newOwner) public onlyOwner {
```

```
        require(newOwner != address(0), "Invalid new owner address");
        owner = newOwner;
    }

    function changeElectionDuration(uint256 _newDuration)
        public
        onlyOwner
        electionOnGoing
    {
        require(_newDuration > 0, "Invalid duration");

        votingEndTimeStamp = votingStartTimeStamp + (_newDuration * 1
minutes);

        emit ElectionDurationChanged(_newDuration);
    }

    function addCandidate(string memory _name)
        public
        onlyOwner
        electionOnGoing
    {
        require(
            hasVotedList.length == 0,
            "Cannot add new candidates once votes have been cast."
        );
        candidates.push(
            Candidate({id: candidates.length, name: _name, numberOfVotes:
0})
        );

        emit CandidateAdded(candidates.length - 1, _name);
    }

    function registerVoter(address _eligible_voter) public onlyOwner {
        eligibleVoters[_eligible_voter] = true;
        registeredVoters.push(_eligible_voter); // Add the voter to the
registeredVoters
    }

    function registerVoters(address[] memory _eligible_voters)
        public
        onlyOwner
    {
        for (uint256 i = 0; i < _eligible_voters.length; i++) {
            eligibleVoters[_eligible_voters[i]] = true;
            registeredVoters.push(_eligible_voters[i]);
        }
    }

    function mintResultNFTs(string memory _tokenURI) public onlyOwner {
        require(!electionStarted, "Election is ongoing, cannot mint NFTs
yet");
        require(
```

```
            votingStartTimeStamp != 0,
            "Timestamp is 0, election not even started"
        );

        for (uint256 i = 0; i < registeredVoters.length; i++) {
            // Mint NFT to each eligible voter
            ElectionNFT(electionNFTContract).mintNFT(
                registeredVoters[i],
                _tokenURI
            );
        }
    }

    function mintResult(address _participant, string memory _tokenURI)
        public
        onlyOwner
    {
        require(!electionStarted, "Election is ongoing, cannot mint NFTs
yet");
        require(
            votingStartTimeStamp != 0,
            "Timestamp is 0, election not even started"
        );
        // Mint an NFT for the participant
        ElectionNFT(electionNFTContract).mintNFT(_participant, _tokenURI);

        // Mark the participant as having received an NFT
        // voters[_participant] = true;
    }

    // Function to set ElectionNFT contract address
    function setElectionNFTContract(address _electionNFTContract)
        public
        onlyOwner
    {
        electionNFTContract = _electionNFTContract;
    }

    // Structure template for election metadata
    struct ElectionMetadata {
        uint256 electionID;
        uint256 winnerID;
        string winnerName;
        uint256 numberOfVotes;
        uint256 startTime;
        uint256 endTime;
        string title;
    }

    struct Winner {
        uint256 candidateID;
        string name;
        uint256 numberOfVotes;
        uint256 electionID;
```

```
    }

    function getWinnerInfo() public view returns (Winner memory) {
        require(!electionStarted, "Election is still ongoing");
        require(candidates.length > 0, "No candidates available");

        uint256 maxVotes = 0;
        uint256 winningCandidateIndex;

        for (uint256 i = 0; i < candidates.length; i++) {
            if (candidates[i].numberOfVotes > maxVotes) {
                maxVotes = candidates[i].numberOfVotes;
                winningCandidateIndex = i;
            }
        }

        Winner memory winner = Winner({
            candidateID: candidates[winningCandidateIndex].id,
            name: candidates[winningCandidateIndex].name,
            numberOfVotes: maxVotes,
            electionID: electionID
        });

        return winner;
    }

    function generateMetadata() public view returns (ElectionMetadata
memory) {
        require(!electionStarted, "Election is still ongoing");
        require(candidates.length > 0, "No candidates available");

        uint256 maxVotes = 0;
        uint256 winningCandidateIndex;

        for (uint256 i = 0; i < candidates.length; i++) {
            if (candidates[i].numberOfVotes > maxVotes) {
                maxVotes = candidates[i].numberOfVotes;
                winningCandidateIndex = i;
            }
        }

        Winner memory winner = Winner({
            candidateID: candidates[winningCandidateIndex].id,
            name: candidates[winningCandidateIndex].name,
            numberOfVotes: maxVotes,
            electionID: electionID
        });

        // Create ElectionMetadata struct
        ElectionMetadata memory metadata = ElectionMetadata({
            electionID: electionID,
            winnerID: winner.candidateID,
            winnerName: winner.name,
            numberOfVotes: winner.numberOfVotes,
```

```
            startTime: votingStartTimeStamp,
            endTime: votingEndTimeStamp,
            title: electionTitle
        });

        return metadata;
    }
}
```

CREATING THE ElectionNFT.SOL FILE

In this section, we'll guide you through the process of creating the `ElectionNFT.sol` file, which will handle the creation of unique NFTs for each voter in our decentralized voting application.

Create a new file named `ElectionNFT.sol` and paste the following code:

It's a good start for the explanation of the `ElectionNFT.sol` file. However, you can provide more details and context for each part of the code to ensure that your audience, especially those new to blockchain and smart contracts, can understand the purpose and functionality of each component. Here's a more detailed breakdown of the code and additional explanations you can include:

CREATING THE ElectionNFT.SOL FILE

In this section, we'll guide you through the process of creating the `ElectionNFT.sol` file, which will handle the creation of unique NFTs for each voter in our decentralized voting application.

Code Explanation:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.22;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";

contract ElectionNFT is ERC721 {
    address public electionContractAddress;
    uint256 public electionId;
    uint256 private tokenIdCounter;
    string private baseTokenURI;
    mapping(uint256 => string) private tokenURIs;
```

- **SPDX-License-Identifier:** Specifies the license type under which the code is released. In this case, it's set to "UNLICENSED," indicating that the code doesn't have a specific license.
- **Solidity Version:** Indicates the compiler version the code is compatible with (Solidity version 0.8.19 in this case).

- **Importing ERC721:** This contract inherits from the ERC721 standard, which is the Ethereum standard for non-fungible tokens (NFTs). It provides the basic functionality needed for NFTs.
- **Variables:**
    - `electionContractAddress`: Stores the address of the election contract that interacts with this NFT contract.
    - `electionId`: Represents the unique identifier for the election.
    - `tokenIdCounter`: Keeps track of the unique token IDs minted.
    - `baseTokenURI`: Stores the base URI for token metadata.
    - `tokenURIs`: Maps token IDs to their respective token URIs.

Constructor:

```
    constructor(address _electionContractAddress) ERC721("Election NFT",
"ENFT") {
        electionContractAddress = _electionContractAddress;
    }
```

- **Constructor:** Initializes the NFT contract with the name "Election NFT" and the symbol "ENFT." It also sets the `electionContractAddress` to the provided address.

Base URI Function:

```
    function _baseURI() internal view virtual override returns (string
memory) {
        return baseTokenURI;
    }
```

- **Base URI Function:** Overrides the internal `_baseURI` function from the ERC721 standard. It returns the base URI for all token metadata.

Set Base Token URI Function:

```
    function setBaseTokenURI(string memory _newBaseTokenURI) external {
        require(msg.sender == electionContractAddress, "Only the election
contract can set base URI");
        baseTokenURI = _newBaseTokenURI;
    }
```

- **Set Base Token URI Function:** Allows the election contract to set the base URI for token metadata. This is essential for generating unique token URIs.

Mint NFT Function:

```
    function mintNFT(address _to, string memory _tokenURI) external {
        require(msg.sender == electionContractAddress, "Only the election
contract can mint NFTs");
        _safeMint(_to, tokenIdCounter);
        tokenURIs[tokenIdCounter] = _tokenURI;
        tokenIdCounter++;
    }
```

- **Mint NFT Function:** Enables the election contract to mint a new NFT for a specified address (`_to`). It uses `_safeMint` from the ERC721 standard and associates the new token ID with the provided token URI.

Get Token URI Function:

```
    function getTokenURI(uint256 _tokenId) external view returns (string
memory) {
        return tokenURIs[_tokenId];
    }
}
```

- **Get Token URI Function:** Allows external parties to retrieve the token URI associated with a specific token ID. (This is generaly the URL of already uploaded metadata on IPFS.)

Conclusion:

In conclusion, the `ElectionNFT.sol` file defines a contract that complements the main voting contract by handling the creation and management of unique NFTs for each voter. These NFTs serve as a transparent and verifiable record of each voter's participation in the election.

Full ElectionNFT.sol contract code

Your smart contract `ElectionNFT.sol` should look like this:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.22;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";

contract ElectionNFT is ERC721 {
    address public electionContractAddress;
    uint256 public electionId;
    uint256 public tokenIdCounter;
    string private baseTokenURI;
    mapping(uint256 => string) private tokenURIs;

    constructor(address _electionContractAddress)
        ERC721("Election NFT", "ENFT")
```

```
    {
        electionContractAddress = _electionContractAddress;
    }

    event NFTMinted(address indexed to, uint256 tokenId, string tokenURI);

    function _baseURI() internal view virtual override returns (string
memory) {
        return baseTokenURI;
    }

    function setBaseTokenURI(string memory _newBaseTokenURI) external {
        require(
            msg.sender == electionContractAddress,
            "Only the election contract can set base URI"
        );
        baseTokenURI = _newBaseTokenURI;
    }

    function mintNFT(address _to, string memory _tokenURI) external {
        require(
            msg.sender == electionContractAddress,
            "Only the election contract can mint NFTs"
        );
        tokenIdCounter++;
        _safeMint(_to, tokenIdCounter);
        tokenURIs[tokenIdCounter] = _tokenURI;
        emit NFTMinted(_to, tokenIdCounter, _tokenURI);
    }

    function getTokenURI(uint256 _tokenId)
        external
        view
        returns (string memory)
    {
        return tokenURIs[_tokenId];
    }
}
```