

Blockchain & Solidity Lab3 – Crowdfunding dApp Development

S2BC



Lab 3: Integrate Web App with Smart Contracts

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

Objective: The aim of this Lab3 is to integrate the smart contracts you developed in Lab1 and Lab2 with a Crowdfunding dApp for users to access the dApp using the web browser.

Deploy Compiled Smart Contract with Hardhat

To deploy the compiled contract to the Ethereum blockchain network, follow these steps:

Step 1: Configure a dotenv (.env) file

First, install the **dotenv** package using the following command:

```
npm install dotenv
```

Next, create a **.env** file in the root folder of your HardHat project.(hardhat/.env) This file will contain sensitive information that should be kept secure. Add the following variables to the **.env** file:

```
# This is the URL of the Ethereum RPC provider
RPC_URL="https://example.com/rpc" (optain from morpheus)

# This is a private key for signing transactions (private key of the
deployer account)
PRIVATE_KEY="your_private_key_here"

# This is the chain ID for the Ethereum network
CHAIN_ID=12345
```

Make sure to replace the placeholder values with your actual credentials.

Step 2: Configure hardhat.config.js

Modify your `hardhat.config.js` file as follows:

```
require("@nomicfoundation/hardhat-toolbox");
require("dotenv").config();

/** @type import('hardhat/config').HardhatUserConfig */
module.exports = {
  solidity: "0.8.22",
  networks: {
    // Add your network configuration here
    poa: {
      url: process.env.RPC_URL, // RPC URL of your network
      chainId: parseInt(process.env.CHAIN_ID), // Chain ID of your network
      accounts: [process.env.PRIVATE_KEY], // Array of private keys to use
      with this network
    },
  },
};
```

Step 3: Create a Combined Deployment Script

Create a new file named `deploy.js` inside the `hardhat/scripts` directory. Add the following content to the file:

```
const { ethers } = require("hardhat");
const fs = require("fs");

async function deployCampaignCreator() {
  // Get the deployer's address
  const [deployer] = await ethers.getSigners();
  console.log(
    "Deploying CampaignCreator contract with the account:",
    deployer.address
  );

  // Get the CampaignCreator contract factory
  const CampaignCreator = await
ethers.getContractFactory("CampaignCreator");

  // Deploy the CampaignCreator contract
  const campaignCreator = await CampaignCreator.deploy();
  // console.log(campaignCreator.target);

  // Save deployment information to a text file
  const deploymentInfo = `Deployer Address:
${deployer.address}\nCampaignCreator Contract Address:
${campaignCreator.target}`;
  console.log(
```

```

    `CampaignCreator Contract Address deployed: ${campaignCreator.target}`
  );
  fs.writeFileSync("deploymentInfoCampaignCreator.txt", deploymentInfo);

  // Return the deployed CampaignCreator contract instance
  return campaignCreator;
}

async function main() {
  try {
    // Deploy the CampaignCreator contract
    const campaignCreator = await deployCampaignCreator();

    console.log("Deployment completed successfully!");
  } catch (error) {
    console.error("Error deploying contracts:", error);
    process.exitCode = 1;
  }
}

main();

```

To deploy the contracts, use the following command in your terminal:

```
npx hardhat run scripts/deploy.js --network poa
```

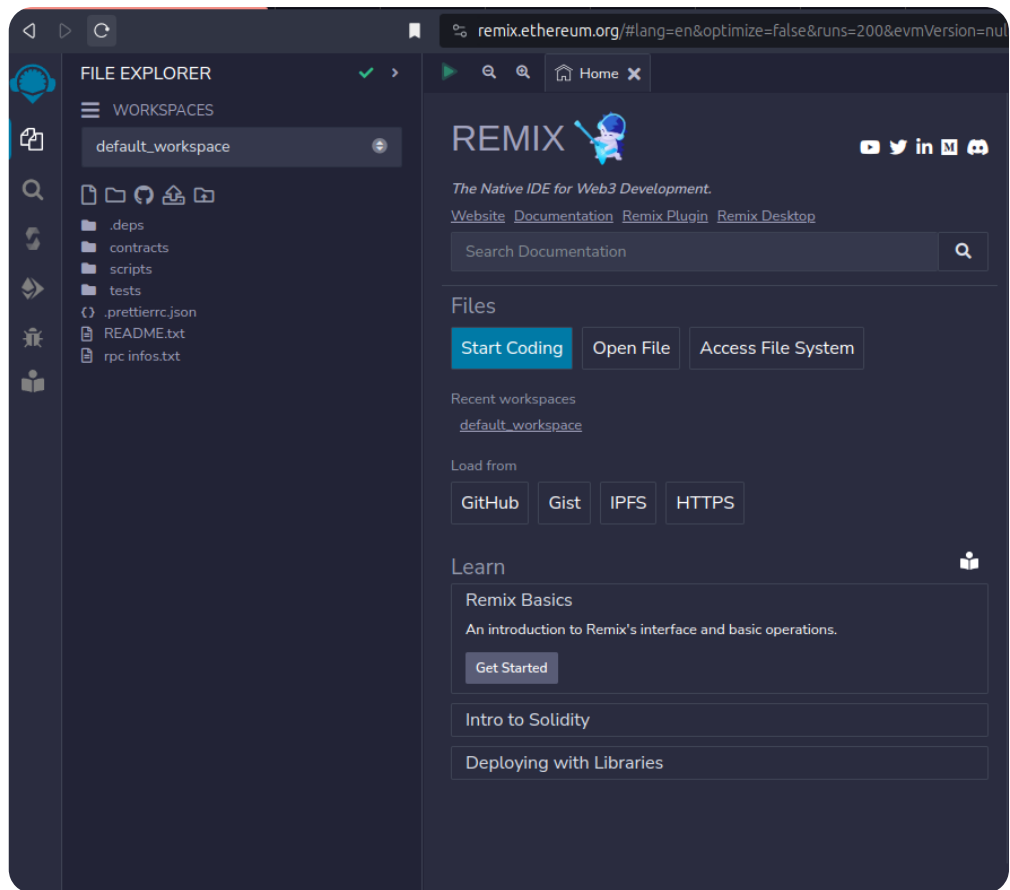
The result output from the terminal will provide the contract addresses.

A "deploymentInfoCampaignCreator.txt" file will be created with the CampaignCreator contract address.

Try Your Contracts on Remix IDE

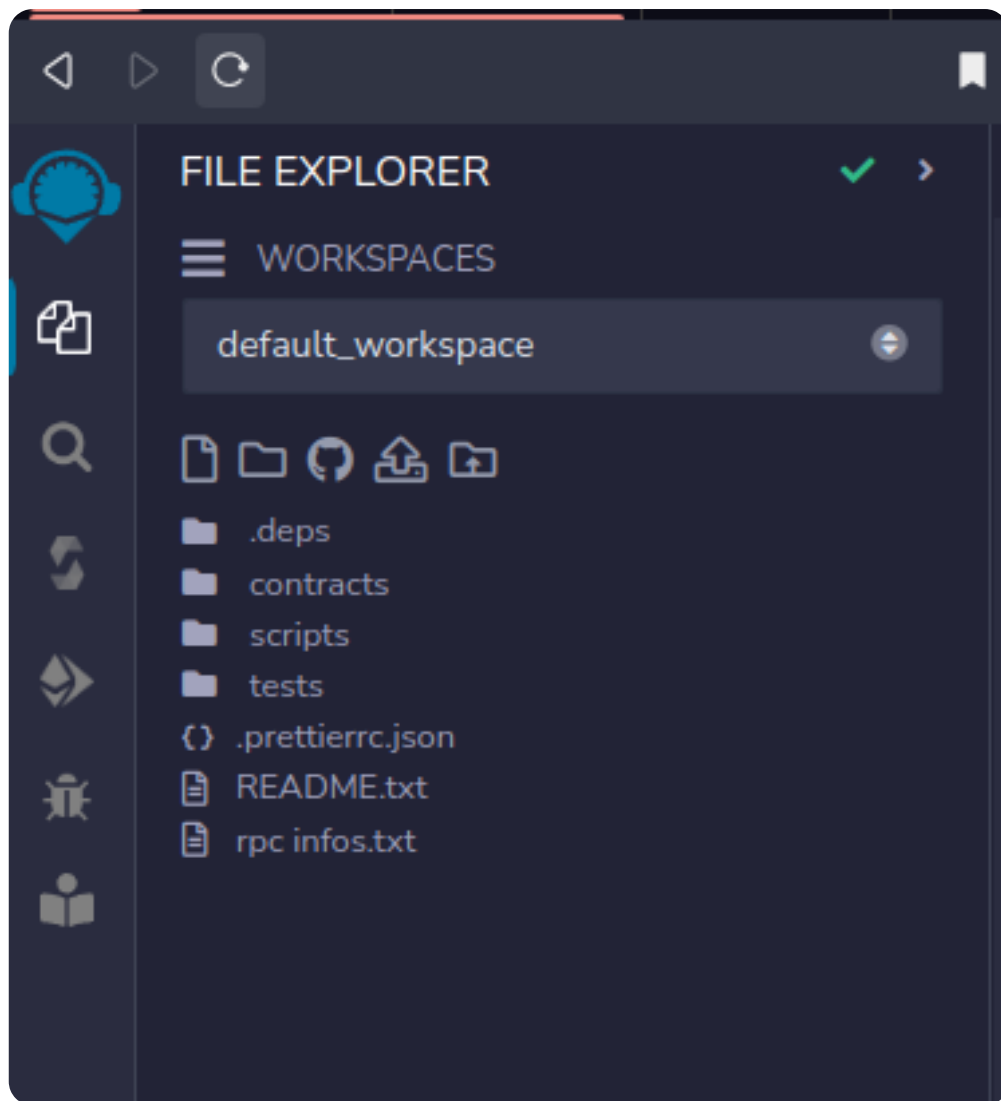
Remix IDE provides a visual way to interact with your contracts before implementing your frontend. Follow these steps to test your contracts:

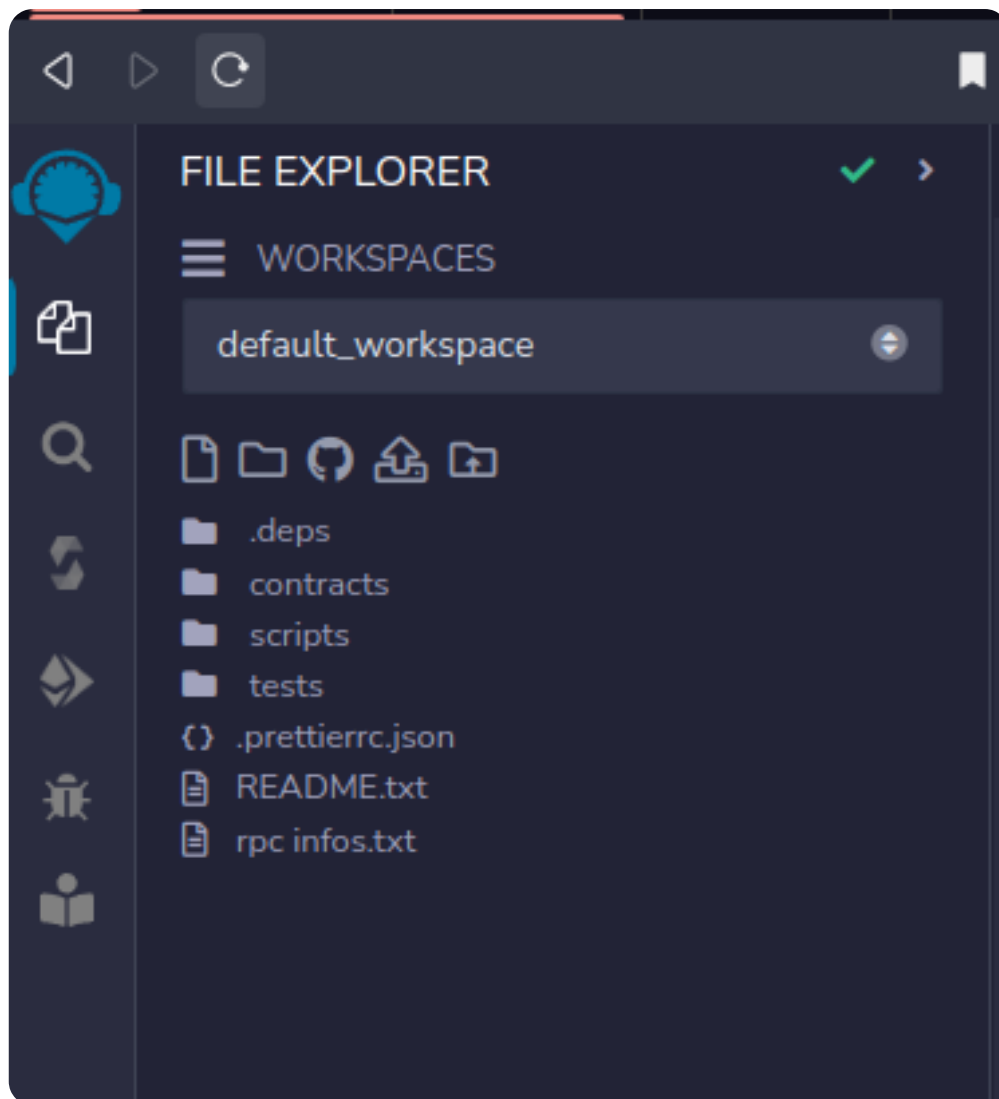
1. Visit the Remix website: [Remix IDE](#).

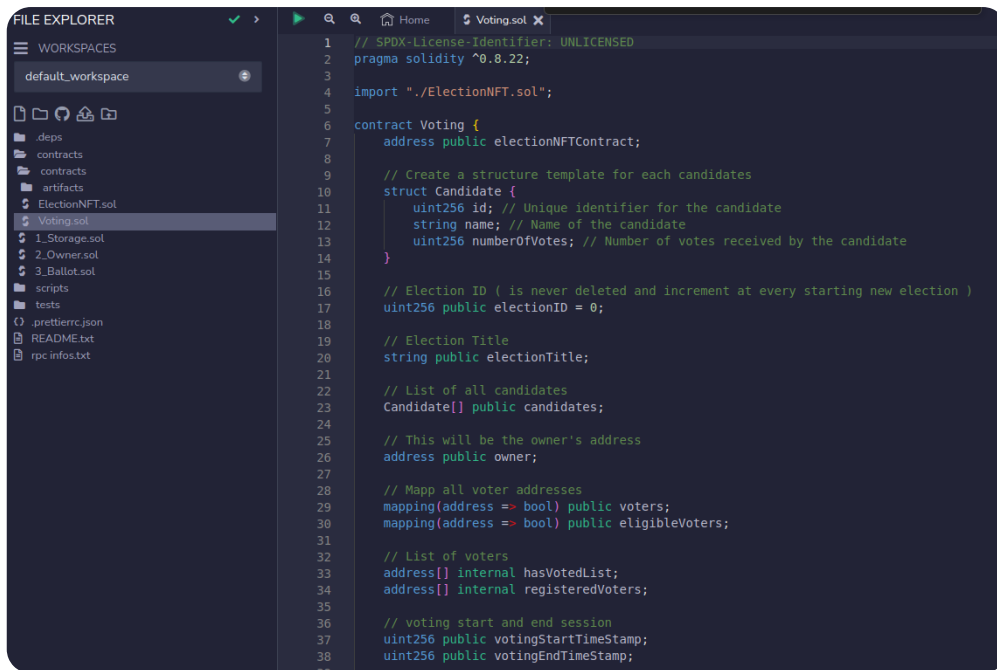


2. Upload your contracts CampaignCreator.sol and CrowdCollab.sol:

- Navigate to the contract folder.
- Click on one contract and press the compile green arrow.





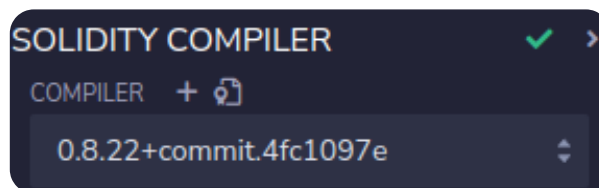


```
1 // SPDX-License-Identifier: UNLICENSED
2 pragma solidity ^0.8.22;
3
4 import "./ElectionNFT.sol";
5
6 contract Voting {
7     address public electionNFTContract;
8
9     // Create a structure template for each candidates
10    struct Candidate {
11        uint256 id; // Unique identifier for the candidate
12        string name; // Name of the candidate
13        uint256 numberOfVotes; // Number of votes received by the candidate
14    }
15
16    // Election ID ( is never deleted and increment at every starting new election )
17    uint256 public electionID = 0;
18
19    // Election Title
20    string public electionTitle;
21
22    // List of all candidates
23    Candidate[] public candidates;
24
25    // This will be the owner's address
26    address public owner;
27
28    // Mapp all voter addresses
29    mapping(address => bool) public voters;
30    mapping(address => bool) public eligibleVoters;
31
32    // List of voters
33    address[] internal hasVotedList;
34    address[] internal registeredVoters;
35
36    // voting start and end session
37    uint256 public votingStartTimeStamp;
38    uint256 public votingEndTimeStamp;
```



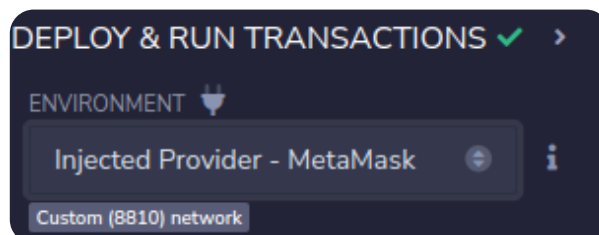
3. Ensure that the compiler version is set to 0.8.22:

- Select the "Compiler" tab.
- Confirm that version 0.8.22 is checked.

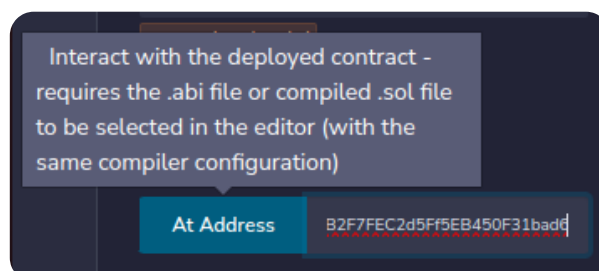


4. Go to the "Deploy" tab:

- In the deploy tab, select "Wallet Injected Provider."
- Connect your MetaMask account to Remix IDE.

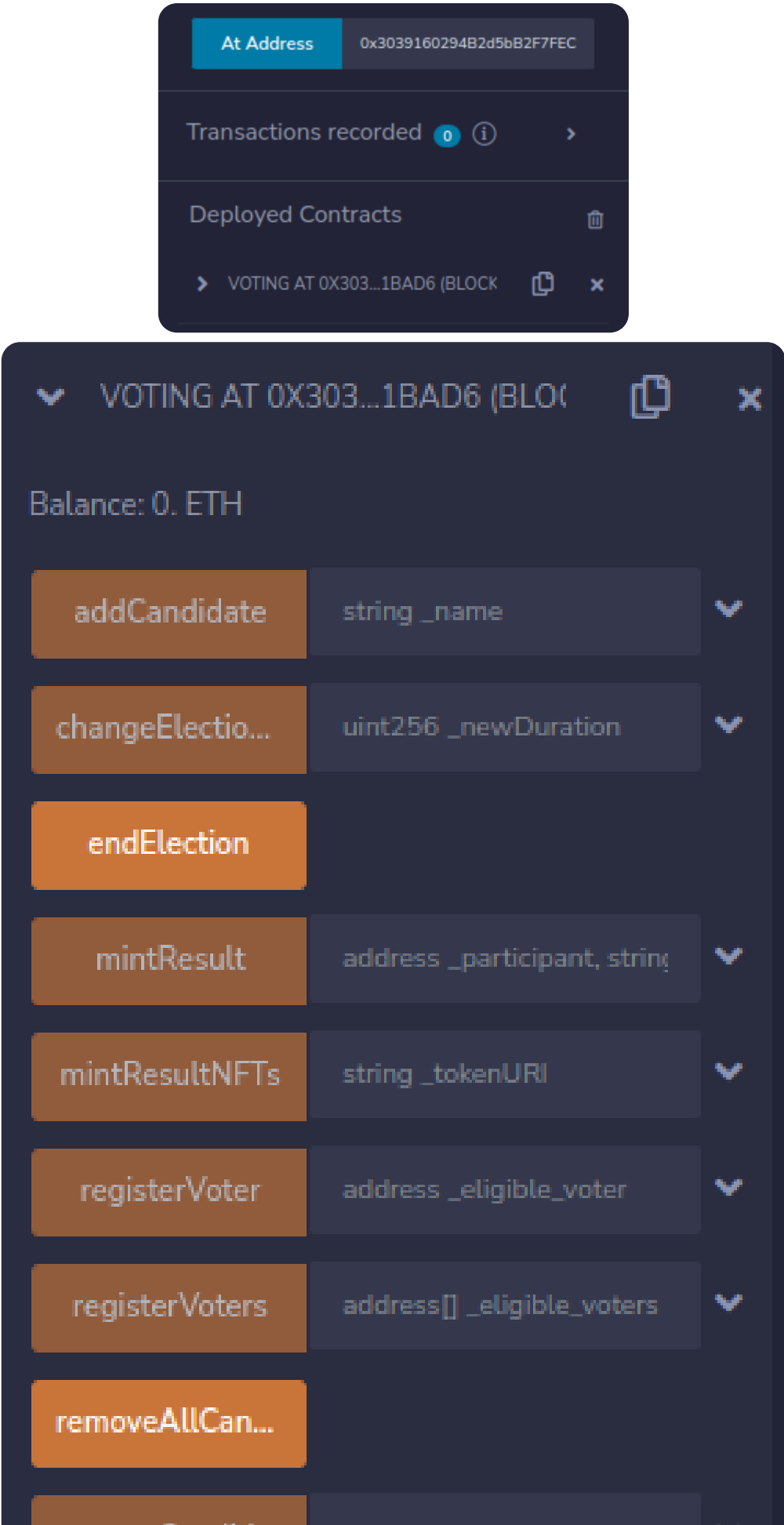


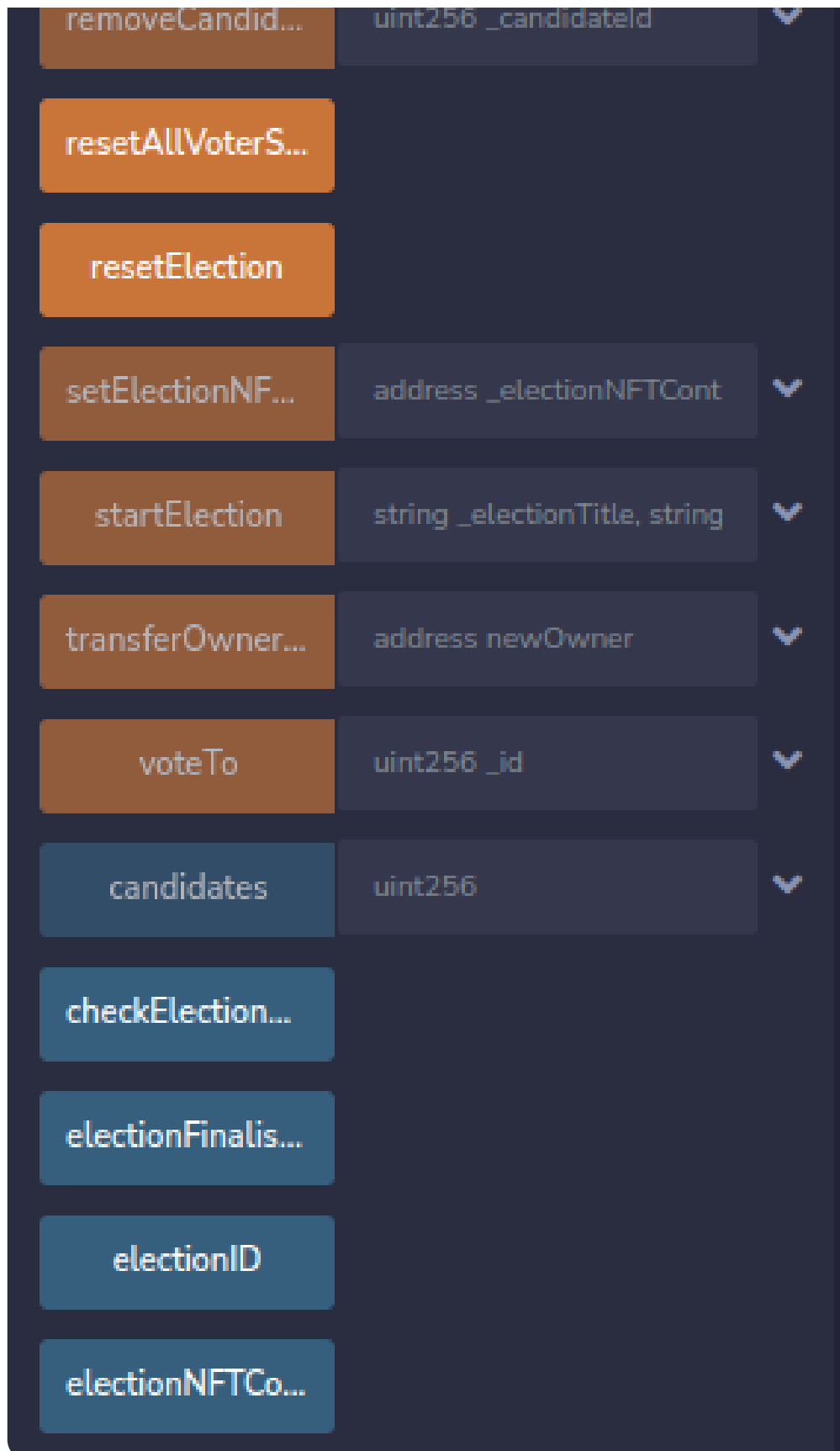
5. Paste the address of your deployed CampaignCreator.sol contract at the bottom of the deploy tab.



and click on "address" button

6. Load your already deployed contract:
- This action allows you to interact with your contract in the newly appeared menu.



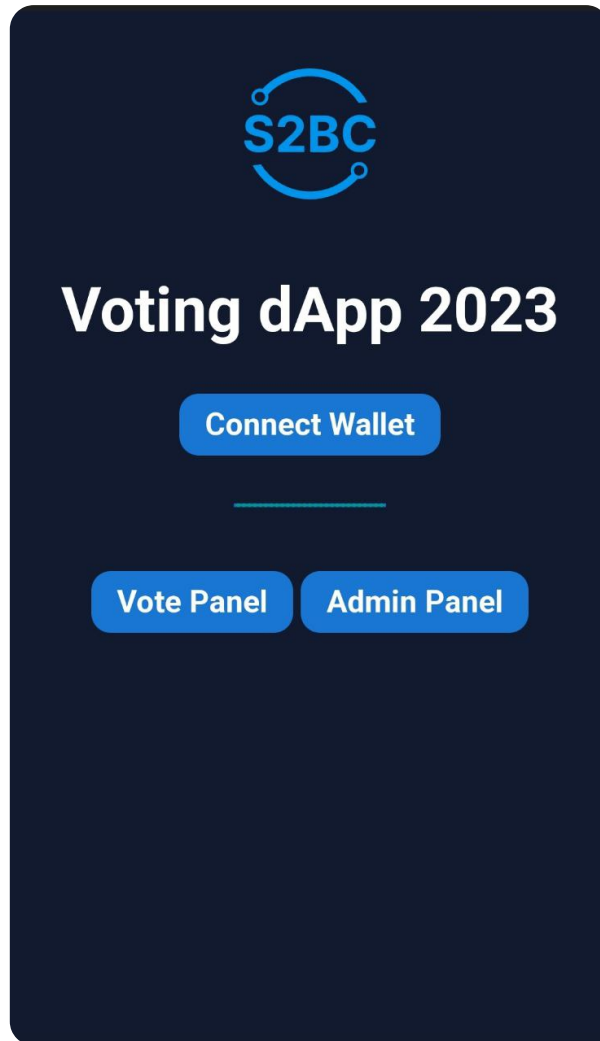


By adhering to these guidelines, you can efficiently verify and engage with your contracts through Remix IDE before advancing to frontend development.

Once you've established your initial campaign, you may access the CrowdCollab instance address by repeating the earlier procedure, this time selecting the CrowdCollab contract and ensuring it's compiled before invocation.

Frontend integration

UI-Screenshoots :



Voting panel :

Vote Panel

Admin Panel

Election Title: No title yet

Election ID: Not started yet

Timer: 0 s left

ID	Candidates	Votes count	Vote Button
No candidates yet			

Refresh voting board

Send Vote

Election is ongoing :

Vote Panel

Admin Panel

Election Title: Who is the strongest?

Election ID: 10

Timer: 2 days 1 h 59 min 48 s left

ID	Candidates	Votes count	Vote Button
0	Hulk	0	Vote
1	Batman	0	Vote
2	Conan	0	Vote

Refresh voting board

Someone voted:

Vote Panel

Admin Panel

Election Title: Who is the strongest?

Election ID: 10

Timer: 2 days 1 h 58 min 24 s left

ID	Candidates	Votes count	Vote Button
0	Hulk	1	<div>Vote</div>
1	Batman	0	<div>Vote</div>
2	Conan	0	<div>Vote</div>

Administrator panel :

Vote PanelAdmin Panel

Start an Election:

Election Title

Candidate 1

Candidate 2

Candidate 3

+ candidates separated by (,)

How long the election will last?:

Duration in minutes

Start Election

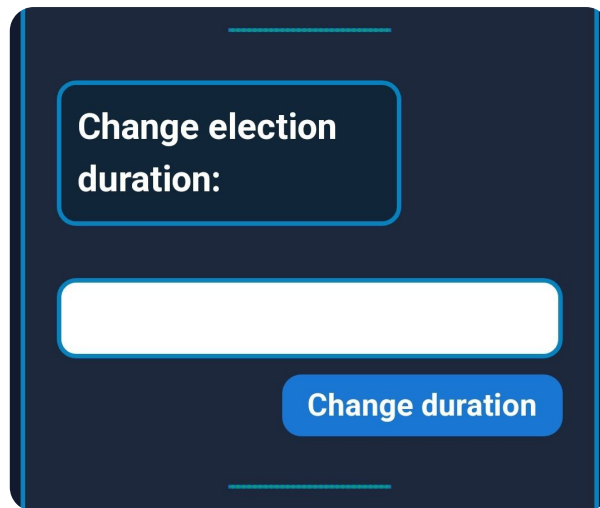
Register voter :

Register voters
addresses:

+ addresses separated by(,)

Register Voters

Change election duration :

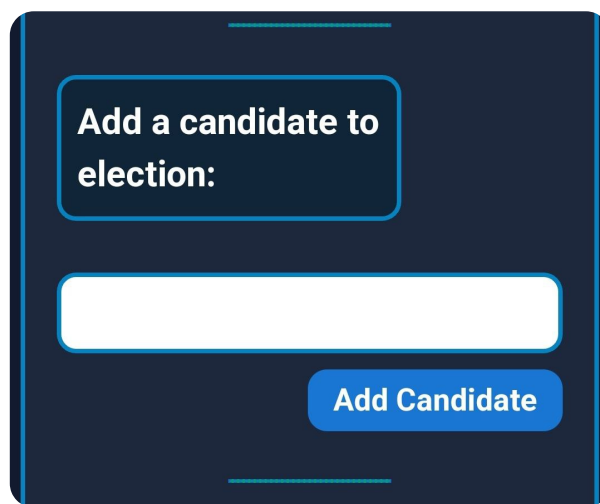


Change election duration:

Change duration

This screen has a dark blue background with a light blue border. It features a rounded rectangle containing the text 'Change election duration:'. Below this is a white text input field. At the bottom right is a blue button with the text 'Change duration'.

Add a candidate to election :

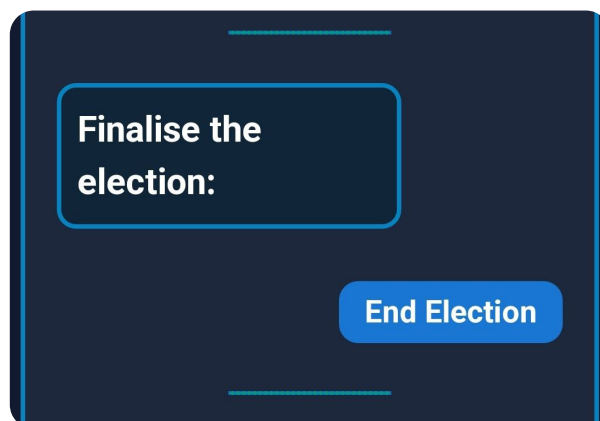


Add a candidate to election:

Add Candidate

This screen has a dark blue background with a light blue border. It features a rounded rectangle containing the text 'Add a candidate to election:'. Below this is a white text input field. At the bottom right is a blue button with the text 'Add Candidate'.

Finalise the election :

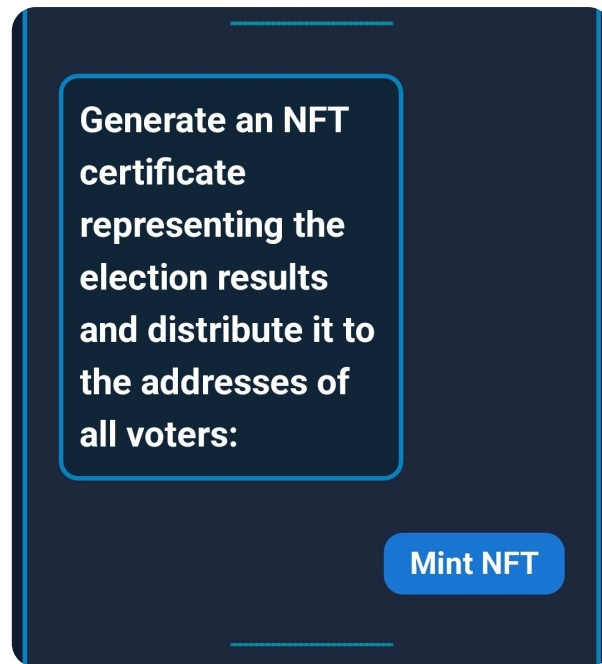


Finalise the election:

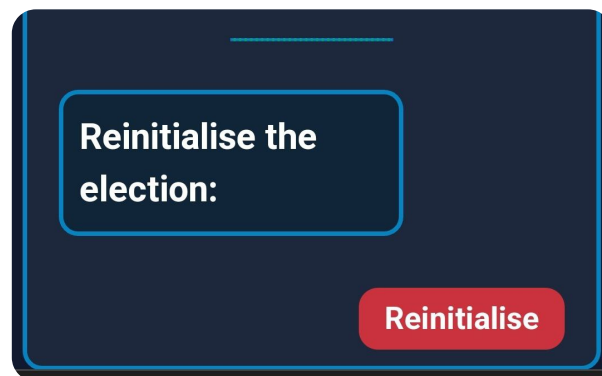
End Election

This screen has a dark blue background with a light blue border. It features a rounded rectangle containing the text 'Finalise the election:'. At the bottom right is a blue button with the text 'End Election'.

Generate an NFT certificate :



Reinitialise the election :



Setting Up the Frontend

In this section, we will guide you through setting up the frontend of your Voting dApp. Follow these steps to create the necessary folders and files:

1. Create a Frontend Folder

Begin by creating a folder named **frontend** within your project directory. This folder will house all the files related to the frontend of your dApp.

Your tree folder should be like:

```
- voting-dapp-2023
  - hardhat
  - frontend
```

So if you were in hardhat folder, come back to your root folder:

```
cd ..
```


then create the frontend folder

```
mkdir frontend  
cd frontend
```

2. Let's initiate Node.js

```
npm init -y
```

That will create a package.json file into your frontend folder.

2. Then install Express.js

```
npm install express
```

Now your package.json file should look like this:

```
{  
  "name": "frontend",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "express": "^4.18.2"  
  }  
}
```

That is where you can add more dependencies if needed in the future.

You have two ways to add dependencies:

- use of "npm install express" or
- add a line "express": "^4.18.2" to the package.json file

2. Set Up a Server by creating a server.js file

If your dApp requires server-side functionality, create a file named `server.js` in the `frontend` folder. This file will handle any backend logic your application may need.

Create a `server.js` file and add those lines:

```
// server.js
const express = require('express');
const path = require('path');

const app = express();
const port = process.env.PORT || 3000;

app.use(express.static(path.join(__dirname, 'public')));

app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'index.html'));
});

app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});
```

To test out if the server works you can run while you are located into frontend folder:

```
npm start
```

You should get this terminal output:

```
npm start

> frontend@1.0.0 start
> node server.js

Server is running on http://localhost:3000
```

and then be able to open your explorer and check `http://localhost:3000` if the server is running.

2. Create a Public Folder

Inside the `frontend` folder, create a subfolder named `public`. This folder will hold any publicly accessible files, such as HTML and images.

```
mkdir public
cd public
```

The **public** folder is where you store files that can be accessed by users, such as the main HTML file (**index.html**) and client-side JavaScript (**script.js**). As well that where you store the images of the app like the **logo.svg** and **favicon.ico**.

3. Create HTML and JavaScript Files

Within the **public** folder, create the following files:

- **index.html**: This is the main HTML file that will serve as the entry point of your web application. You can use this file to structure the layout of your dApp's user interface.

```
<html>
  Server is Up
</html>
```

- **script.js**: This JavaScript file will contain the client-side code that interacts with the smart contracts and updates the UI based on user actions.

```
// JavaScript code will be here
console.log("Server is Up and Running");
```

3. Create styles.css Files

Within the **public** folder, create a styles.css file, it will contain the custom styles for your UI.

```
:root {
  --prussian-blue: #102537ff; /* Main background color */
  --prussian-blue-2: #24344cff; /* Table background color */
  --oxford-blue: #111a2eff; /* Alternate background color */
  --oxford-blue-2: #1d283cff; /* Panel background color */
  --oxford-blue-3: #172237ff; /* Header background color */
  --persian-green: #019b83ff; /* Button background color (e.g., Connect
Wallet Button) */
  --blue-ncs: #0684c2ff; /* Border color and accent color */
  --paynes-gray: #5d7084ff; /* Text color and secondary elements */
  --rusty-red: #ca323eff; /* Alert or error color (e.g., Reinitialise
Election Button) */
}

body {
  font-family: "Roboto Regular", Helvetica, Arial, Lucida, sans-serif;
  background-color: var(--oxford-blue);
  line-height: 1.7em;
  font-weight: 500;
  display: flex;
  flex-direction: column;
  justify-content: center;
```

```
align-items: center;
height: 100vh;
margin: 0 20px;
font-size: 1.2em;
color: #ffffff;
}
```

Frontend Folder Structure

At this point your folder structure should look like that:

```
- voting-dapp-2023
  - hardhat
  - frontend
    - server.js
    - public
      - index.html
      - script.js
      - styles.css
```

Test out if the server works again:

```
npm start
```

Then check <http://localhost:3000> in your browser. Open console with F12 and check the console logs after refreshing the page.

```
Server is Up and Running
```

Building the index.html Page

Let's concentrate on the HTML page for now. We will reference the script.js file, include the ethers CDN link to load the ethers functions into our app, and make them accessible. Afterward, we will design the layout, integrate some buttons, and set up IDs and classes for some elements to make them interactable with the scripts.

Including Ethers.js CDN link, reference to script.js and style.css

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```

<title>Voting dApp</title>
<!-- Include Ethers.js CDN link with crossorigin attribute -->
  <script src="https://cdn.ethers.io/lib/ethers-5.2.umd.min.js"
type="application/javascript"></script>
  <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
  <div id="votingStation">
    <!-- all the following code into this div -->
  </div>
  <script src="script.js"></script>
</body>

</html>

```

Create votingStation DIV into `<body></body>`

```

<body>
<div id="votingStation">
// all the folowing code into this div
</div>
<script src="script.js"></script>
</body>

```

The votingStation DIV encapsulates all components related to the voting process, making it modular and easy to manage. Each subcomponent, such as wallet connection, panel buttons, and voting forms, resides within this container for a clean and organized structure.

In the upcoming sections, we will continue building upon the votingStation DIV, adding interactive elements and functionalities to create a comprehensive and user-friendly decentralized voting experience. Let's proceed with the next steps to enrich the user interface and interaction capabilities of your voting dApp.

Structure expected:

```

<html>
  <head>
    <!-- Head content goes here -->
  </head>
  <body>
    <!-- Voting Station DIV -->
    <div id="votingStation">

      <!-- Logo -->
      

      <!-- Title -->
      <h1>Voting dApp 2023</h1>
    </div>
  </body>
</html>

```

```

        <!-- Connect Wallet Section -->
        <div id="connectWallet">
            <button id="connectWalletbutton">Connect Wallet</button>
            <p id="connectWalletMessage"><span
id="connectWalletMessageSpan"></span></p>
        </div>

        <!-- Panel Buttons Section -->
        <div id="buttonPanel">
            <hr class="custom-line">
            <button id="votePanelButton">Vote Panel</button>
            <button id="adminPanelButton">Admin Panel</button>
        </div>

        <!-- Vote Panel and Admin Panel Sections -->
        <div id="votePanel" class="panel">
            <!-- Vote Panel content -->
        </div>

        <div id="admin" class="panel">
            <!-- Admin Panel content -->
        </div>

        <!-- Other sections within the votingStation DIV -->
    </div>
</body>
</html>

```

Add a Logo and the app Title

```



<h1>Voting dApp 2023</h1>

```

Add a Connect Button

```

<div id="connectWallet">
    <button id="connectWalletbutton">Connect Wallet</button>
    <p id="connectWalletMessage"><span id="connectWalletMessageSpan">
</span></p>
</div>

```

We provide an ID to this button and span message to be able to interact with it from the javascript code in script.js

Add Panel Buttons

```

<div id="buttonPanel">
  <hr class="custom-line" />
  <button id="votePanelButton">Vote Panel</button>
  <button id="adminPanelButton">Admin Panel</button>
</div>

```

These buttons, with the IDs `votePanelButton` and `adminPanelButton`, are designed to serve as navigation elements for users interacting with your voting dApp. These IDs will be utilized in your `script.js` file to enable specific functionalities associated with the vote panel and admin panel.

Add Voting panel

```

<div id="votePanel" class="panel">
  <!-- Main board for displaying election information -->
  <div id="mainBoard">
    <!-- Section for displaying election ID and timer -->
    <div id="topBoard">
      <p style="margin-right: 20px;">
        Election ID: <span id="electionID"></span>
      </p>
      <p id="timerMessage" style="margin-right: 20px;">
        Timer: <span id="time" onclick="showTimer()">[refresh]</span>
      </p>
    </div>

    <!-- Table for displaying candidates -->
    <table id="candidateBoard">
      <tr>
        <th>ID</th>
        <th>Candidates</th>
        <th>Votes count</th>
        <th>Vote Button</th>
      </tr>
      <!-- Rows for each candidate will be dynamically added here using
      JavaScript -->
    </table>
  </div>

  <!-- Button to refresh the voting board -->
  <div id="showCandidateListContainer">
    <button id="showCandidateList" onclick="showTimer()">
      Refresh voting board
    </button>
  </div>

  <hr class="custom-line" />

  <!-- Section for casting a vote -->
  <div id="voteForm">
    <h3>Please enter the candidate's ID for your vote:</h3>

```

```

    <label for="">
      <input type="number" id="vote" />
      <button id="sendVote">Send Vote</button>
    </label>
  </div>
</div>

```

Here's a breakdown of what each section does:

1. **Main Board:** This section contains the key information about the ongoing election, including the Election ID and a timer.
2. **Candidate Board Table:** A table where candidate information will be dynamically displayed. The table headers indicate the information presented for each candidate.
3. **Show Candidate List Button:** A button to refresh and display the current voting board.
4. **Vote Form:** This section allows users to cast their votes by entering the candidate's ID and clicking the "Send Vote" button.

Add Administration panel

And then, to finish with the HTML file, we will add the admin panel div.

```

<div id="admin" class="panel">
  <!-- Section for starting an election -->
  <div id="startElection">
    <h3 id="startElectionMessage">Start an Election:</h3>
    <input type="text" id="addCandidateInput" placeholder="Candidate 1" />
    <input type="text" id="addCandidateInput2" placeholder="Candidate 2" />
    <input type="text" id="addCandidateInput3" placeholder="Candidate 3" />
    <input
      type="text"
      id="addCandidateInput4"
      placeholder="+ candidates separated by (,) "
    />
    <p>How long the election will last?:</p>
    <input
      type="number"
      id="specifyDuration"
      placeholder="Duration in minutes"
    />
    <button id="startElectionButton">Start Election</button>
  </div>

  <hr class="custom-line" />

  <!-- Section for registering voters' addresses -->
  <div id="addVoterArray">
    <h3>Register voters' addresses:</h3>
    <input

```



```

        type="text"
        id="addVoterInputArray"
        placeholder="+ addresses separated by(,)"
    />
    <button id="addVoterButtonArray">Register Voters</button>
</div>

<hr class="custom-line" />

<!-- Section for changing the election duration -->
<div id="changeElectionDuration">
    <h3>Change election duration:</h3>
    <input type="number" id="changeElectionDurationInput" />
    <button id="changeElectionDurationButton">Change duration</button>
</div>

<hr class="custom-line" />

<!-- Section for adding a candidate to the election -->
<div id="addCandidate">
    <h3>Add a candidate to the election:</h3>
    <input type="text" id="addCandidateInputBonus" />
    <button id="addCandidateButton">Add Candidate</button>
</div>

<hr class="custom-line" />

<!-- Section for finalizing the election -->
<div id="endElection">
    <h3>Finalize the election:</h3>
    <button id="endElectionButton">End Election</button>
</div>

<hr class="custom-line" />

<!-- Section for minting an NFT certificate -->
<div id="saveResultsNFT">
    <h3>
        Generate an NFT certificate representing the election results and
        distribute it to the addresses of all voters:
    </h3>
    <button id="generateAndUploadMetadataButton">Mint NFT</button>
</div>

<hr class="custom-line" />

<!-- Section for reinitializing the election -->
<div id="resetElection">
    <h3>Reinitialize the election:</h3>
    <button id="resetElectionButton">Reinitialize</button>
</div>
</div>

```

Here's a breakdown of what each section does:

1. **Start Election Section:** Allows administrators to start a new election by providing candidate names, election duration, and clicking the "Start Election" button.
2. **Register Voters Section:** Enables administrators to register voters' addresses by providing a list of addresses separated by commas and clicking the "Register Voters" button.
3. **Change Election Duration Section:** Allows administrators to change the duration of the ongoing election by entering a new duration and clicking the "Change duration" button.
4. **Add Candidate Section:** Provides administrators with the ability to add additional candidates to the ongoing election by entering the candidate's name and clicking the "Add Candidate" button.
5. **Finalize Election Section:** Allows administrators to finalize the election by clicking the "End Election" button.
6. **Save Results NFT Section:** Provides a button for administrators to generate an NFT certificate representing the election results and distribute it to all voters.
7. **Reset Election Section:** Allows administrators to reinitialize the election by clicking the "Reinitialize" button.

Build script.js: Interacting with the Smart Contract by adding javascript functions

Now, we will focus on writing the javascript functions we need to interact with index.html

first, let's make the connect button working!

1. Connecting the Wallet

- Explain the purpose of this function and its significance in blockchain development.
- Emphasize the security aspect of connecting to a wallet and handling accounts.

```
// Function to connect Metamask
async function connectToWallet() {
  try {
    await window.ethereum.request({ method: "eth_requestAccounts" });

    const provider = new ethers.providers.Web3Provider(window.ethereum,
1303);

    provider.send("eth_requestAccounts", []).then(() => {
      console.log("Accounts requested");

      provider.listAccounts().then((accounts) => {
        console.log("List of accounts:", accounts);

        signer = provider.getSigner(accounts[0]);
        contract = new ethers.Contract(contractAddress, contractABI,
```

```

signer);

    console.log("Signer and Contract set up");

    // Update UI elements and display messages
    connectWalletBtn.textContent = "Connected";
    connectWalletBtn.style.backgroundColor = "#019B83ff"; // Change the
background color to light green

    // Display address connected
    connectWalletMessageSpan.innerHTML = `${accounts[0]}`;
    getElectionID();
    fetchElectionTitle();
  });
});

votingStation.style.display = "block";
} catch (error) {
  console.error(error);
  console.log(
    "Error connecting to Metamask. Please make sure it's installed and
unlocked."
  );
}
}

```

2. Starting an Election

- The process of starting an election includes providing an election title, specifying candidate names, and specifying the duration.

```

// Function to start the election
startElectionButton.addEventListener("click", async () => {
  try {
    const electionTitle =
document.querySelector("#electionTitleInput").value;

    const candidates = [
      addCandidateInput.value,
      addCandidateInput2.value,
      addCandidateInput3.value,
      ...addCandidateInput4.value.split(","),
    ].filter(Boolean);
    console.log(candidates);
    const votingDuration = specifyDuration.value;

    const provider = new ethers.providers.Web3Provider(window.ethereum,
1303);

    provider.send("eth_requestAccounts", []).then(() => {
      provider.listAccounts().then((accounts) => {

```

```

        console.log("List of accounts:", accounts);

        signer = provider.getSigner(accounts[0]);
        contract = new ethers.Contract(contractAddress, contractABI,
signer);
    });
});

    await contract.startElection(electionTitle, candidates,
votingDuration);
    console.log("Election is starting, wait for blockchain confirmation");
} catch (error) {
    console.error(error);
    console.log("Error starting the election: " + error.message);
    const errorMessage =
        "Error starting the election: " + extractErrorMessage(error);
    console.log(errorMessage);
    alert(errorMessage);
}
});

```

3. Register Voters

- Walk through the process of registering voters to an election, and discuss how this might be relevant in a real-world scenario.

```

// Event listening to registering of voter
addVoterButtonArray.addEventListener("click", async () => {
    try {
        const voterAddresses = addVoterInputArray.value
            .split(",")
            .map((address) => address.trim());
        await contract.registerVoters(voterAddresses);
        console.log(`Voters registered successfully!`);
    } catch (error) {
        console.error(error);
        console.log(`Error registering voters: ${error.message}`);
        const errorMessage =
            "Error registering voters: " + extractErrorMessage(error);
        console.log(errorMessage);
        alert(errorMessage);
    }
});

```

3. Adding Candidates

- Walk through the process of adding candidates to an election, and discuss how this might be relevant in a real-world scenario.

```
// Function to add candidate after election start but before anyone voted
addCandidateButton.addEventListener("click", async () => {
  try {
    const candidateName = addCandidateInputBonus.value;
    await contract.addCandidate(candidateName);
    console.log("Candidate added successfully!");
  } catch (error) {
    console.error(error);
    console.log("Error adding candidate: " + error.message);
    const errorMessage =
      "Error adding candidate: " + extractErrorMessage(error);
    console.log(errorMessage);
    alert(errorMessage);
  }
});
```

4. Changing Election Duration

- Explain the purpose of this functionality and its potential use cases.
- Discuss any considerations regarding the duration of an election.

```
// Function to change the election duration
changeDurationButton.addEventListener("click", async () => {
  try {
    const newDuration = specifyNewDuration.value;
    await contract.changeElectionDuration(newDuration);
    console.log("Election duration changed successfully!");
  } catch (error) {
    console.error(error);
    console.log("Error changing election duration: " + error.message);
    const errorMessage =
      "Error changing election duration: " + extractErrorMessage(error);
    console.log(errorMessage);
    alert(errorMessage);
  }
});
```

5. Resetting the Election

- Describe the purpose of resetting an election and any safeguards in place to prevent accidental resets.

```
// Function to reset the election
resetElectionButton.addEventListener("click", async () => {
  try {
    await contract.resetElection();
    console.log("Election reset successfully!");
  }
});
```

```

    } catch (error) {
      console.error(error);
      console.log("Error resetting the election: " + error.message);
      const errorMessage =
        "Error resetting the election: " + extractErrorMessage(error);
      console.log(errorMessage);
      alert(errorMessage);
    }
  });

```

6. Voting

- Walk through the process of casting a vote, emphasizing the importance of secure and transparent voting mechanisms.
- Explain any error handling or feedback mechanisms for incorrect inputs.

```

// Function to cast a vote
castVoteButton.addEventListener("click", async () => {
  try {
    const selectedCandidate = getSelectedCandidate(); // Implement a
function to get the selected candidate
    await contract.vote(selectedCandidate);
    console.log("Vote cast successfully!");
  } catch (error) {
    console.error(error);
    console.log("Error casting vote: " + error.message);
    const errorMessage = "Error casting vote: " +
extractErrorMessage(error);
    console.log(errorMessage);
    alert(errorMessage);
  }
});

```

7. Displaying Candidates and Timer

- Explain how the UI displays the list of candidates and the associated information.

```

// Function to display main voting board with candidate and timer
async function displayCandidates() {
  if (contract) {
    const candidates = await contract.retrieveVotes();
    const candidateBoard = document.querySelector("#candidateBoard");
    const rows = candidateBoard.querySelectorAll("tr");

    for (let i = 1; i < rows.length; i++) {
      candidateBoard.removeChild(rows[i]);
    }
  }
}

```

```

if (candidates.length === 0) {
  const noCandidatesRow = document.createElement("tr");
  noCandidatesRow.innerHTML = `
    <td colspan="3">No candidates yet</td>
  `;
  candidateBoard.appendChild(noCandidatesRow);
} else {
  candidates.forEach((candidate) => {
    const row = document.createElement("tr");
    row.innerHTML = `
      <th style="word-break: break-all;">${
        candidate.id || "No ID yet"
      }</th>
      <th style="word-break: break-all;">${
        candidate.name || "No name yet"
      }</th>
      <th style="word-break: break-all;">${
        candidate.numberOfVotes || "No vote yet"
      }</th>
      <th><button class="voteBtnRow">Vote</button></th>
    `;
    candidateBoard.appendChild(row);

    const voteRowButton = row.querySelector(".voteBtnRow");
    voteRowButton.addEventListener("click", async () => {
      try {
        const candidateId = candidate.id;
        await contract.voteTo(candidateId);
        console.log("Vote cast successfully!");
      } catch (error) {
        console.error(error);
        console.log("Error casting vote: " + error.message);
        const errorMessage =
          "Error casting vote: " + extractErrorMessage(error);
        console.log(errorMessage);
        alert(errorMessage);
      }
    });
    getElectionID();
  });
}
}
}

```

8. Showing the Timer

- Describe how the timer functionality works, and its significance in the context of an election.

```

// Function to call the timer value
async function showTimer() {
  try {

```

```

    let secondsLeft = await contract.electionTimer();
    console.log("Seconds left:", secondsLeft);
    let formattedDuration = formatDuration(secondsLeft); // <-- Corrected
this line
    console.log("Formatted Duration:", formattedDuration);
    updateTimerMessage(formattedDuration);

    // ...
} catch (error) {
    console.error("Error:", error); // Added console.error
    // Handle errors
}
}

```

9. Handling Events

- Discuss how events are used to track various actions within the smart contract and how they can be beneficial for auditing.

```
// Add your code snippet for handling events here
```

10. Minting NFTs

Minting NFTs in the context of an election serves as a unique and immutable representation of the election results. Each NFT corresponds to a specific voter and their participation in the electoral process. Here's the breakdown of the concept and the potential benefits:

1. Concept of Minting NFTs for Election Results:

- **Unique Representation:** Each NFT represents a specific voter and their role in the election.
- **Immutable Record:** The blockchain ensures that once an NFT is minted, its details cannot be altered or tampered with, providing a secure and transparent record.

2. Code Snippet for Minting NFTs:

```

// Function to mint the results as an NFT by interacting with
ElectionNFT contract
async function mintResultNFTs(tokenURI) {
    try {
        // Assuming you've already set up the connection to the Election
contract and the ElectionNFT contract

        const ListOfVoters = await contract.ListOfVoters(); // Assuming
ListOfVoters is a public state variable
        for (let i = 0; i < ListOfVoters.length; i++) {
            const voterAddress = ListOfVoters[i];
            const isEligible = await contract.eligibleVoters(voterAddress);

```



```
    if (isEligible) {
        await electionNFTContract.mintNFT(voterAddress, tokenURI);
        console.log(`NFT minted for voter at address
${voterAddress}`);
    }
}
} catch (error) {
    console.error(error);
    console.log("Error minting NFT: " + error.message);
    const errorMessage = "Error minting NFT: " +
extractErrorMessage(error);
    console.log(errorMessage);
    alert(errorMessage);
}
}
```

3. Potential Use Cases and Benefits:

- **Voter Engagement:** Issuing NFTs to voters can enhance their engagement by providing a digital token as a symbol of their participation in the democratic process.
- **Transparency:** NFTs on the blockchain provide a transparent and publicly accessible record of each voter's involvement, fostering trust in the election results.
- **Historical Records:** NFTs can serve as historical records, allowing stakeholders to review and analyze past elections.

4. Considerations:

- Ensure that the NFT minting process adheres to any legal or regulatory requirements.
- Clearly communicate the significance of the NFTs to voters to promote understanding and acceptance.

Integrating NFT minting into your election system adds a layer of transparency and uniqueness to the results, contributing to the overall integrity of the electoral process.

