

Sistema de Votación para una Comunidad

Descripción

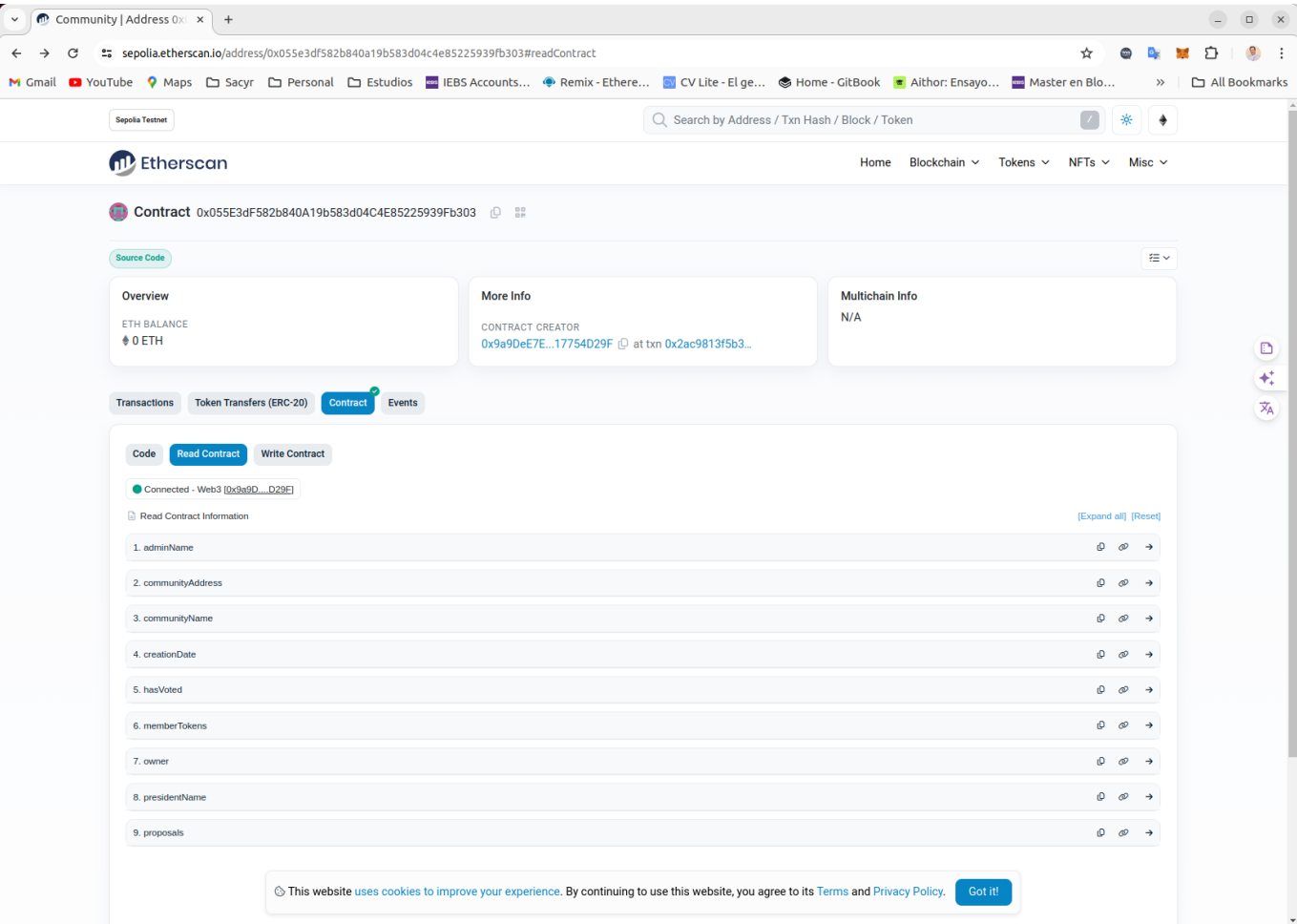
Este proyecto es un sistema de votación descentralizado para una comunidad, implementado utilizando un smart contract en Ethereum y una interfaz de usuario desarrollada en React. Permite a los miembros registrados votar en propuestas y ver los resultados de las votaciones de manera transparente y segura.

Tecnologías Utilizadas

- **Solidity:** Lenguaje de programación para escribir smart contracts.
- **Ethereum Testnet (Sepolia):** Red de prueba donde se despliega el contrato.
- **MetaMask:** Wallet de Ethereum para interactuar con el blockchain.
- **Remix:** IDE de Ethereum usado para desarrollar y desplegar el smart contract.
- **Visual Studio Code:** Editor de código utilizado para el desarrollo del frontend.
- **React:** Biblioteca de JavaScript para construir la interfaz de usuario.
- **Web3.js:** Biblioteca para interactuar con nodos Ethereum desde el navegador.
- **Material-UI:** Biblioteca de componentes React para un diseño de interfaz moderno y responsive.

Smart Contract

El contrato está desplegado en la dirección: [0x055e3df582b840a19b583d04c4e85225939fb303](#) en la red de prueba Sepolia.



Funcionalidades del Contrato

- **Registro de miembros:** Los miembros pueden registrarse almacenando una cantidad de tokens que les permitirá votar.
- **Creación de propuestas:** Los administradores pueden crear propuestas sobre las cuales los miembros podrán votar.
- **Votación:** Los miembros utilizan sus tokens para votar en propuestas activas.
- **Ejecución de propuestas:** Las propuestas que alcanzan los votos necesarios pueden ser ejecutadas por los administradores.

Métodos Principales

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

// Contrato de la Comunidad
contract Community is Ownable(msg.sender) {
    string public communityName;
    string public communityAddress;
    string public presidentName;
    string public adminName;
    uint256 public creationDate;

    mapping(address => uint256) public memberTokens; // Tokens de votación
    asignados a cada miembro
    Proposal[] public proposals; // Lista de propuestas
    mapping(uint256 => mapping(address => bool)) public hasVoted; //
    Mapping to track if a member has voted on a proposal

    struct Proposal {
        string description;
        uint256 voteCount;
        bool executed;
    }

    event ProposalCreated(uint256 proposalId, string description);
    event VoteReceived(uint256 proposalId, address voter, uint256 votes);
    event ProposalExecuted(uint256 proposalId);

    constructor(
        string memory _name,
        string memory _address,
        string memory _president,
        string memory _admin,
        uint256 _creationDate
    ) {
        communityName = _name;
        communityAddress = _address;
    }
}
```

```
        presidentName = _president;
        adminName = _admin;
        creationDate = _creationDate;
    }

    function registerMember(address member, uint256 tokens) public
onlyOwner {
        memberTokens[member] = tokens;
    }

    function createProposal(string memory description) public onlyOwner {
        proposals.push(Proposal({
            description: description,
            voteCount: 0,
            executed: false
        }));

        uint256 newProposalId = proposals.length - 1;
        emit ProposalCreated(newProposalId, description);
    }

    function voteOnProposal(uint256 proposalId, uint256 votes) public {
        require(memberTokens[msg.sender] >= votes, "Not enough tokens to
vote");
        require(!hasVoted[proposalId][msg.sender], "Already voted");

        proposals[proposalId].voteCount += votes;
        hasVoted[proposalId][msg.sender] = true;
        memberTokens[msg.sender] -= votes;

        emit VoteReceived(proposalId, msg.sender, votes);
    }

    function executeProposal(uint256 proposalId) public onlyOwner {
        require(!proposals[proposalId].executed, "Proposal already
executed");
        proposals[proposalId].executed = true;
        // Logic to execute the proposal goes here

        emit ProposalExecuted(proposalId);
    }
}
```

Contrato de la Comunidad

Descripción General

Este contrato inteligente, desarrollado con Solidity, permite la gestión de una comunidad mediante la asignación de tokens de votación a sus miembros y la creación y votación de propuestas. Está diseñado para ser administrado por el propietario del contrato, quien tiene derechos exclusivos para ciertas operaciones.

Características del Contrato

- **Gestión de Miembros y Propuestas:** Solo el propietario puede añadir miembros y crear propuestas.
- **Votación:** Los miembros pueden votar en propuestas activas utilizando tokens asignados.

Detalles del Contrato

Importaciones

- **OpenZeppelin Contracts:** Utiliza `Ownable` para funciones de propiedad y `ERC20` para manejar tokens.

```
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
```

Variables de Estado

communityName, communityAddress, presidentName, adminName, creationDate: Almacenan información básica sobre la comunidad. **memberTokens:** Un mapping que relaciona cada dirección de miembro con sus tokens de votación. **proposals:** Un array que almacena todas las propuestas creadas. **hasVoted:** Un mapping doble para rastrear si un miembro ha votado en una propuesta.

Structs

Proposal: Define una propuesta con descripción, conteo de votos y estado de ejecución.

```
struct Proposal {
    string description;
    uint256 voteCount;
    bool executed;
}
```

Eventos

ProposalCreated: Se emite cuando se crea una nueva propuesta. **VoteReceived:** Se emite cuando un voto es registrado. **ProposalExecuted:** Se emite cuando una propuesta es ejecutada. ****Constructor Inicializa el contrato con datos básicos de la comunidad y establece el propietario.

```
constructor(string memory _name, string memory _address, string memory
_president, string memory _admin, uint256 _creationDate) Ownable() {
    communityName = _name;
    communityAddress = _address;
    presidentName = _president;
    adminName = _admin;
    creationDate = _creationDate;
}
```

Funciones del Contrato

registerMember: Asigna tokens a un nuevo miembro. Solo puede ser llamada por el propietario.

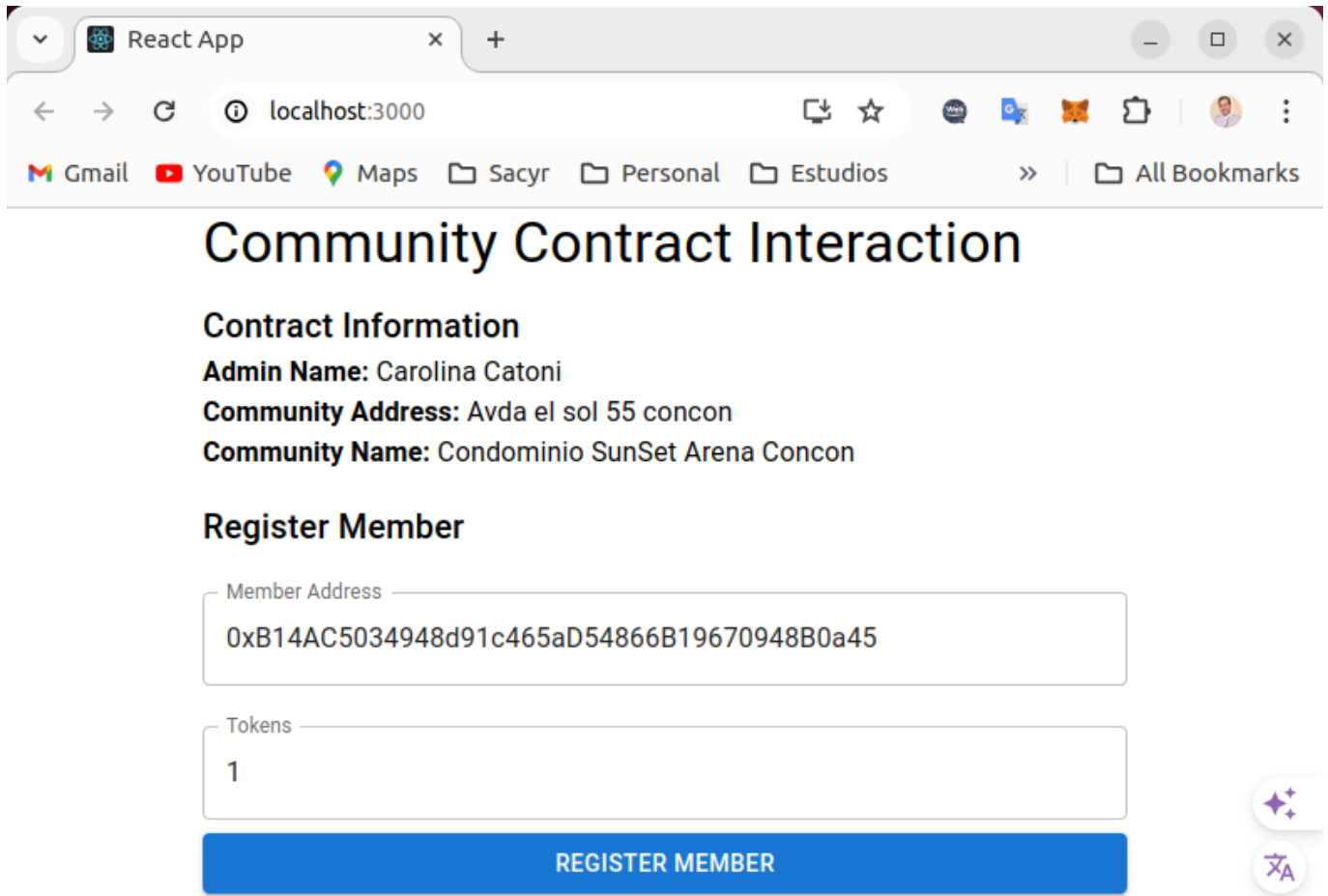
createProposal: Crea una nueva propuesta. Solo puede ser iniciada por el propietario. **voteOnProposal:**

Permite a los miembros votar en propuestas activas. **executeProposal:** Ejecuta una propuesta aprobada.

Solo el propietario puede ejecutarla.

Frontend

El frontend está desarrollado en React y utiliza Web3.js para interactuar con el contrato inteligente a través de MetaMask.



The screenshot shows a web browser window with the title 'React App' and the address 'localhost:3000'. The browser's address bar and bookmarks are visible. The main content of the page is titled 'Community Contract Interaction'. Below this title, there is a section for 'Contract Information' which lists: 'Admin Name: Carolina Catoni', 'Community Address: Avda el sol 55 concon', and 'Community Name: Condominio SunSet Arena Concon'. Below the contract information is a section titled 'Register Member'. This section contains two input fields: 'Member Address' with the value '0xB14AC5034948d91c465aD54866B19670948B0a45' and 'Tokens' with the value '1'. A blue button labeled 'REGISTER MEMBER' is positioned below the input fields. On the right side of the page, there are two circular icons: one with a purple star and another with a purple 'A'.

Componentes Principales

Formulario de Registro de Miembro: Permite a los usuarios registrarse como miembros. Creador de

Propuestas: Interfaz para que los administradores creen nuevas propuestas. Votaciones: Permite a los

miembros votar en propuestas activas.

Fuentes de los Componentes

```
import React, { useState, useEffect } from 'react';
import Web3 from 'web3';
import { TextField, Button, CircularProgress, Typography, Container, Box }
from '@mui/material';
```

```
const contractAddress = '0x055e3df582b840a19b583d04c4e85225939fb303';
const abi = []; // Acá va el ABI del contrato

function App() {
  const [web3, setWeb3] = useState(null);
  const [contract, setContract] = useState(null);
  const [accounts, setAccounts] = useState([]);
  const [adminName, setAdminName] = useState('');
  const [communityAddress, setCommunityAddress] = useState('');
  const [communityName, setCommunityName] = useState('');
  const [member, setMember] = useState('');
  const [tokens, setTokens] = useState('');
  const [loading, setLoading] = useState(false);

  useEffect(() => {
    async function loadWeb3() {
      if (window.ethereum) {
        const web3 = new Web3(window.ethereum);
        await window.ethereum.enable();
        const accounts = await web3.eth.getAccounts();
        const contract = new web3.eth.Contract(abi, contractAddress);
        setWeb3(web3);
        setAccounts(accounts);
        setContract(contract);

        const adminName = await contract.methods.adminName().call();
        const communityAddr = await
contract.methods.communityAddress().call();
        const communityNm = await contract.methods.communityName().call();

        setAdminName(adminName);
        setCommunityAddress(communityAddr);
        setCommunityName(communityNm);
      } else {
        alert('Please install MetaMask!');
      }
    }

    loadWeb3();
  }, []);

  const registerMember = async () => {
    if (!contract || !member || !tokens) {
      alert('All fields are required');
      return;
    }
    setLoading(true);
    try {
      await contract.methods.registerMember(member, tokens).send({ from:
accounts[0] });
      alert('Member registered successfully');
    } catch (error) {
      console.error('Error registering member:', error);
      alert('Failed to register member');
    }
  }
}
```

```

    } finally {
      setLoading(false);
    }
  };

  return (
    <Container maxWidth="sm">
      <Typography variant="h4" component="h1" gutterBottom>
        Community Contract Interaction
      </Typography>
      <Box sx={{ mb: 2 }}>
        <Typography variant="h6">Contract Information</Typography>
        <Typography><strong>Admin Name:</strong> {adminName}</Typography>
        <Typography><strong>Community Address:</strong> {communityAddress}
      </Typography>
        <Typography><strong>Community Name:</strong> {communityName}
      </Typography>
      </Box>
      <Box>
        <Typography variant="h6" gutterBottom>Register Member</Typography>
        <TextField
          fullWidth
          label="Member Address"
          value={member}
          onChange={e => setMember(e.target.value)}
          margin="normal"
        />
        <TextField
          fullWidth
          label="Tokens"
          type="number"
          value={tokens}
          onChange={e => setTokens(e.target.value)}
          margin="normal"
        />
        <Button
          variant="contained"
          color="primary"
          onClick={registerMember}
          disabled={loading}
          fullWidth
        >
          {loading ? <CircularProgress size={24} /> : 'Register Member'}
        </Button>
      </Box>
    </Container>
  );
}

export default App;

```

Interacción con el Contrato de la Comunidad

Este proyecto React permite interactuar con un contrato inteligente de Ethereum para gestionar una comunidad. Los usuarios pueden registrar miembros y consultar información del contrato directamente desde la interfaz.

Funcionalidades

- **Conexión con MetaMask:** Conexión automática a MetaMask para interactuar con la blockchain.
- **Registro de miembros:** Permite a los administradores registrar nuevos miembros y asignarles tokens.
- **Visualización de información del contrato:** Muestra datos como el nombre del administrador, la dirección y el nombre de la comunidad.

Tecnologías Utilizadas

- React.js
- Web3.js
- Material-UI: para los componentes de la interfaz de usuario.
- Ethereum Blockchain

Instalación y Configuración

Asegúrate de tener instalado **Node.js** y **npm**. Sigue los pasos para ejecutar la aplicación:

1. Clona el repositorio.
2. Instala las dependencias necesarias:

Detalles del Código

Componente **App**

Es el componente principal que maneja la conexión con Ethereum y muestra la interfaz de usuario.

Variables de Estado

- **web3:** Instancia de Web3 usada en toda la aplicación.
- **contract:** Instancia del contrato inteligente.
- **accounts:** Lista de cuentas disponibles desde el wallet.
- **adminName, communityAddress, communityName:** Variables para almacenar y mostrar la información del contrato.
- **member:** Dirección del nuevo miembro a registrar.
- **tokens:** Número de tokens a asignar al nuevo miembro.
- **loading:** Estado para controlar la visualización del indicador de carga durante las transacciones.

Hook **useEffect**

Inicializa la conexión con MetaMask y carga los datos iniciales desde el contrato:

```
useEffect(() => {  
  async function loadWeb3() {
```



```
if (window.ethereum) {
  const web3 = new Web3(window.ethereum);
  await window.ethereum.enable();
  const accounts = await web3.eth.getAccounts();
  const contract = new web3.eth.Contract(abi, contractAddress);
  setWeb3(web3);
  setAccounts(accounts);
  setContract(contract);

  const adminName = await contract.methods.adminName().call();
  const communityAddr = await contract.methods.communityAddress().call();
  const communityNm = await contract.methods.communityName().call();

  setAdminName(adminName);
  setCommunityAddress(communityAddr);
  setCommunityName(communityNm);
} else {
  alert('¡Instala MetaMask!');
}

loadWeb3();
}, []);
```

Función registerMember

Maneja el registro de un nuevo miembro enviando una transacción al contrato inteligente:

```
const registerMember = async () => {
  if (!contract || !member || !tokens) {
    alert('Todos los campos son obligatorios');
    return;
  }
  setLoading(true);
  try {
    await contract.methods.registerMember(member, tokens).send({ from:
accounts[0] });
    alert('Miembro registrado con éxito');
  } catch (error) {
    console.error('Error al registrar miembro:', error);
    alert('Falló el registro del miembro');
  } finally {
    setLoading(false);
  }
};
```

Mejoras Futuras

Manejo avanzado de errores: Proporcionar retroalimentación más detallada sobre los errores de blockchain. **Actualizaciones en tiempo real:** Implementar WebSocket u otra tecnología para obtener actualizaciones en tiempo real del estado del contrato. **Accesibilidad mejorada:** Asegurar que la interfaz cumpla con los estándares de accesibilidad.

Instalación y Configuración

Para ejecutar este proyecto localmente, necesitarás instalar las dependencias y configurar MetaMask en tu navegador.

Prerrequisitos

Node.js npm o yarn MetaMask instalado en tu navegador

Pasos para la Instalación

Clona el repositorio. Instala las dependencias:

```
npm install  
npm start
```

Licencia

Este proyecto está bajo la Licencia MIT.