Manual técnico “Alquiler de Autos mediante contratos inteligentes”

1. Descripción general

1.1. Visión General

Este proyecto es una aplicación descentralizada (DApp) diseñada para gestionar el alquiler de automóviles utilizando la tecnología blockchain de Ethereum. La aplicación combina un sistema de gestión de identidad descentralizada (DID) con contratos inteligentes desplegados en Ganache, lo que garantiza seguridad, transparencia y trazabilidad en todas las transacciones realizadas.

El proyecto está desarrollado como una prueba de concepto (PoC) para mostrar cómo la tecnología blockchain puede utilizarse en la industria del alquiler de automóviles, proporcionando un modelo eficiente y seguro que elimina intermediarios tradicionales.

1.2. Objetivos del Proyecto

1. **Automatización Segura del Alquiler de Autos:**
    - Gestionar el alquiler de automóviles de forma descentralizada sin la intervención de terceros.

2. **Validación Descentralizada de Identidades (DID):**
    - Implementar un sistema de identidad descentralizada para validar usuarios mediante un DID único generado al iniciar sesión.

3. **Transparencia y Trazabilidad en Transacciones:**
    - Usar contratos inteligentes para garantizar que todas las transacciones sean transparentes, inmutables y verificables en la red Ethereum.

4. **Facilidad de Uso:**
    - Diseñar una interfaz gráfica minimalista y elegante para facilitar la experiencia del usuario al interactuar con la aplicación.

5. **Escalabilidad para Futuras Funcionalidades:**
    - Dejar una base sólida para futuras mejoras, como agregar más modelos de automóviles, sistemas de calificación para usuarios, seguros integrados y notificaciones automatizadas.

1.3. Características Principales:

1. **Inicio de Sesión Seguro:**
    - Los usuarios pueden iniciar sesión utilizando una dirección de Ethereum y una clave privada obtenida de Ganache.
    - Se genera un DID basado en la dirección Ethereum, que se utiliza posteriormente para validar la identidad en el sistema.

2. **Dashboard Dinámico:**
    - Muestra una lista de 40 autos disponibles para alquilar.
    - Cada auto incluye un modelo, un precio aleatorio entre 10 y 50 ETH, una imagen única y un estado de disponibilidad generado aleatoriamente.

3. **Formulario Emergente para Alquiler:**
    - Al pulsar "Rent" en un auto disponible, aparece un formulario emergente que solicita información personal (nombre, apellido, correo y DID).
    - El DID ingresado se valida antes de proceder con la transacción.

4. **Transacciones Seguras con Contratos Inteligentes:**
    - El contrato inteligente gestiona los pagos, verificando el saldo del usuario y actualizando el estado del auto a "No disponible" tras el alquiler.
    - Las transacciones se reflejan inmediatamente en Ganache, mostrando el cambio en el saldo de la cuenta.

5. **Gestión del Estado de los Autos:**
    - Los autos no disponibles se muestran con un estado "Unavailable" en color rojo y su botón "Rent" deshabilitado.
    - Los autos disponibles tienen un estado "Available" en verde con el botón habilitado.

6. **Actualización Automática del Saldo:**
    - Tras una transacción exitosa, el saldo del usuario se actualiza dinámicamente en el dashboard sin necesidad de recargar la página.

7. **Seguridad Mejorada:**
    - Se utilizan ventanas emergentes personalizadas en lugar de prompts para validar la información ingresada.
    - Se restringe el uso del botón "Rent" para evitar clics no autorizados.

1.4. Impacto del Proyecto

Este proyecto aborda varios desafíos comunes en la industria del alquiler de automóviles:

- **Confianza:**
  Al utilizar contratos inteligentes, se eliminan intermediarios, reduciendo fraudes y malentendidos sobre pagos o términos de alquiler.

- **Disponibilidad Global:**
  Los usuarios pueden acceder al sistema desde cualquier lugar, siempre que tengan una dirección Ethereum válida y saldo suficiente.

- **Privacidad:**
  El uso de identidades descentralizadas (DID) reduce la necesidad de almacenar información personal en bases de datos centralizadas, mejorando la seguridad de los datos.

- **Automatización Total:**
  Desde el pago hasta el estado del automóvil, todo el proceso está automatizado, eliminando la necesidad de intervención manual.

1.5. Casos de Uso

1. Usuario Alquila un Auto Disponible:

- El usuario inicia sesión.
- Selecciona un auto disponible, ingresa sus datos y valida el DID.
- El contrato inteligente verifica el saldo y completa la transacción.
- El auto queda marcado como "No disponible".

2. Intento de Alquilar un Auto No Disponible:

- El botón "Rent" aparece deshabilitado.
- El usuario no puede proceder con la transacción.

3. Validación Fallida del DID:

- Si el DID ingresado no coincide con el generado al inicio de sesión, la transacción no se procesa y se muestra un mensaje de error.

4. Saldo Insuficiente:

- Si el saldo del usuario no cubre el costo del auto, la transacción falla y muestra un mensaje detallado.

1.6. Tecnologías Utilizadas

1. **Frontend:**
    - **HTML5, CSS3, Bootstrap** para el diseño minimalista y responsivo.
    - **JavaScript (dashboard.js, login.js)** para la lógica del frontend y manejo de formularios.

2. **Backend:**
    - **Node.js con Express.js** para el servidor.
    - Manejo de rutas, sesiones y conexiones con Ganache.

3. **Blockchain y Contratos Inteligentes:**
    - **Solidity** para el desarrollo del contrato inteligente.
    - **Truffle** para la compilación, migración y pruebas.
    - **Ganache** para simular la red Ethereum.
    - **ethers.js** para interactuar con el contrato desde el frontend.

2. Código del proyecto

2.1. CarRental.sol

El archivo CarRental.sol define el contrato inteligente principal para el sistema de alquiler de automóviles descentralizado. Su objetivo es proporcionar un mecanismo seguro y transparente para gestionar el proceso de alquiler utilizando la tecnología blockchain de Ethereum.

Este contrato gestiona lo siguiente:

1. **Almacenamiento de autos:** Mantiene un registro de todos los autos disponibles, sus características (modelo, precio) y su estado (disponible o no disponible).
2. **Pagos seguros:** Garantiza que las transacciones sean ejecutadas únicamente si el usuario cumple con las condiciones establecidas (como saldo suficiente y auto disponible).

3. **Eventos registrados:** Utiliza eventos para registrar información clave (por ejemplo, autos añadidos y alquilados), lo que facilita el monitoreo desde aplicaciones externas.
4. **Seguridad y validaciones:** Incluye validaciones estrictas para prevenir errores, como intentar alquilar autos no disponibles o con pagos insuficientes.

-Declaraciones iniciales

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;
```

- **Licencia MIT:**
  Define que el código tiene licencia MIT, lo que permite su uso y modificación sin restricciones.
- **Versión del compilador:**
  Se especifica que el contrato requiere Solidity 0.8.13. Esta versión incluye características como verificación automática de desbordamiento/underflow, mejorando la seguridad.

-Declaración del contrato y estructura para representar un auto

```
contract CarRental {
    struct Car {
        uint id;
        string model;
        uint price;
        bool isAvailable;
    }
```

- En Solidity, un contrato actúa como una clase en programación orientada a objetos, conteniendo datos y funciones para gestionar esos datos.
- *struct Car*:
  Es una estructura que almacena los atributos principales de un auto:
  - **id**: Identificador único.
  - **model**: Modelo o nombre del auto.
  - **price**: Precio en wei (1 ETH = 10^18 wei).
  - **isAvailable**: Un booleano que indica si el auto está disponible para alquilar.

Este diseño permite representar fácilmente cada auto como un objeto estructurado.

-Variables de estado

```
mapping(uint => Car) public cars;
mapping(uint => address) public carRenter;
uint public carCount = 0;
```

- **cars**:
  - Un mapeo (mapping) que asocia un id único con un objeto de tipo Car.
  - Sirve como base de datos para almacenar todos los autos.

- **carRenter**:
  - Almacena la dirección (address) de Ethereum del usuario que alquiló un auto específico.

- **carCount**:
  - Lleva un conteo del número total de autos registrados en el contrato.

Estas variables de estado son almacenadas en la blockchain y son inmutables después de ser escritas, lo que garantiza transparencia y trazabilidad.

-Eventos

```
event CarAdded(uint id, string model, uint price);
event CarRented(uint id, address renter, uint amount);
```

- **CarAdded**:
  - Se dispara cuando se agrega un nuevo auto al sistema.
  - Incluye información como el ID, modelo y precio.

- **CarRented**:
  - Se emite cuando un auto es alquilado.
  - Incluye el ID del auto, la dirección del arrendatario y el monto pagado.

Eventos facilitan el monitoreo en aplicaciones externas como dApps o scripts en ethers.js.

-Constructor

```
constructor() {
    string[40] memory carModels = [
        "Toyota Supra", "Nissan GT-R", "Chevrolet Camaro", "Ford Mustang", "Dodge Charger",
        "BMW M3", "Audi R8", "Mercedes AMG GT", "Porsche 911", "Lamborghini Huracan",
        "Ferrari 488", "McLaren 720S", "Aston Martin DB11", "Jaguar F-Type", "Mazda RX-7",
        "Subaru WRX", "Honda NSX", "Tesla Model S", "Koenigsegg Agera", "Bugatti Chiron",
        "Pagani Huayra", "Lexus LC500", "Alfa Romeo Giulia", "Volkswagen Golf GTI",
        "Mitsubishi Lancer Evo", "Hyundai Veloster N", "Kia Stinger", "Ford Focus RS",
        "Chevrolet Corvette", "Dodge Viper", "Ferrari F40", "Lamborghini Aventador",
        "Bentley Continental GT", "Rolls-Royce Wraith", "Mini Cooper S", "Jeep Grand Cherokee",
        "Range Rover Sport", "Volvo XC90", "Tesla Model X", "Porsche Cayenne"
    ];

    for (uint i = 0; i < 40; i++) {
        uint price = (uint(keccak256(abi.encodePacked(block.timestamp, i))) % 41 + 10) * 1 ether;
        addCar(carModels[i], price);
    }
}
```

constructor:

- Es una función especial que se ejecuta solo una vez al desplegar el contrato.
- Inicializa el contrato agregando autos de ejemplo al sistema con precios fijos en ETH (convertido internamente a wei).
- Usa la función addCar() para registrar los autos iniciales.

-Añadir autos

```
function addCar(string memory _model, uint _price) public {
    carCount++;
    cars[carCount] = Car(carCount, _model, _price, true);
    emit CarAdded(carCount, _model, _price);
}
```

Función pública: Puede ser llamada desde fuera del contrato.

Parámetros:

1. *_model*: Modelo del auto.
2. *_price*: Precio en wei.

Proceso:

- Incrementa el contador carCount.
- Almacena la información del auto en el mapeo cars.
- Emite un evento CarAdded.

-Alquilar un auto

```
function rentCar(uint _id) public payable {
    require(_id > 0 && _id <= carCount, "Car ID is invalid");
    Car storage car = cars[_id];

    require(car.isAvailable, "Car is not available");
    require(msg.value >= car.price, "Insufficient payment");

    car.isAvailable = false; // Marcar como alquilado
    carRenter[_id] = msg.sender;

    emit CarRented(_id, msg.sender, msg.value);
}
```

Pago (payable): La función acepta pagos en ETH.

Validaciones (require):

1. El ID es válido.
2. El auto está disponible.
3. El monto pagado es suficiente.

Acciones:

- Cambia el estado del auto a No disponible.
- Asigna al arrendatario en carRenter.
- Emite un evento CarRented.

2.2. 2_deploy_contracts.js

Este archivo define cómo debe desplegarse el contrato inteligente CarRental.sol en la red Ethereum simulada por Ganache. Utiliza el framework Truffle, que simplifica el proceso de migración y administración de contratos inteligentes.

-Detallamiento del código

```
const CarRental = artifacts.require("CarRental");

module.exports = function (deployer) {
  deployer.deploy(CarRental);
};
```

- **artifacts.require**:
    - Importa el contrato CarRental compilado.
    - Busca el archivo CarRental.json generado en la carpeta build/contracts/ después de ejecutar el comando: *truffle compile*
    - Este archivo contiene información como el ABI y la dirección del contrato después de ser desplegado.

- **module.exports:**
    - Exporta una función que será ejecutada por Truffle durante el proceso de migración.

- **deployer:**
    - Objeto proporcionado por Truffle para gestionar el despliegue del contrato.
    - Proporciona métodos para desplegar contratos y configurar dependencias.

- **deployer.deploy(CarRental):**
    - Despliega el contrato CarRental en la red Ethereum configurada (en este caso, Ganache).
    - Guarda automáticamente la dirección del contrato en la red para ser reutilizada en el frontend.

2.3. Login.js

Este archivo gestiona el inicio de sesión en la aplicación mediante:

1. Validación de una dirección Ethereum y una clave privada.
2. Generación de un DID (Decentralized Identifier) para identificar al usuario.
3. Autenticación segura usando una firma digital.
4. Prevención de ataques por fuerza bruta mediante intentos fallidos limitados y tiempos de bloqueo.
5. Almacenamiento seguro de las credenciales en localStorage para mantener la sesión activa.

-Eventos iniciales y constantes

```
document.addEventListener('DOMContentLoaded', function () {
    const loginForm = document.getElementById('loginForm');
    const MAX_ATTEMPTS = 3;
    const LOCKOUT_TIME = 30000; // 30 segundos de bloqueo
```

- *DOMContentLoaded*: Asegura que el código solo se ejecute cuando el DOM haya sido completamente cargado.
- *loginForm*: Referencia al formulario de inicio de sesión en el HTML.
- *MAX_ATTEMPTS*: Número máximo de intentos de inicio de sesión permitidos antes de bloquear al usuario.
- *LOCKOUT_TIME*: Duración del bloqueo cuando se exceden los intentos fallidos, definido en milisegundos.

-Creación de ventanas emergentes

```
function createPopup(message) {
    let popup = document.createElement('div');
    popup.style.position = 'fixed';
    popup.style.top = '50%';
    popup.style.left = '50%';
    popup.style.transform = 'translate(-50%, -50%)';
    popup.style.padding = '20px';
    popup.style.backgroundColor = 'white';
    popup.style.border = '1px solid #ccc';
    popup.style.boxShadow = '0 0 10px rgba(0, 0, 0, 0.1)';
    popup.style.zIndex = '1000';
    popup.style.textAlign = 'center';
    popup.style.fontFamily = 'Arial, sans-serif';

    popup.innerHTML = `<p>${message}</p>`;

    let button = document.createElement('button');
    button.innerText = 'OK';
    button.onclick = function () {
        document.body.removeChild(popup);  // Cerrar la ventana emergente
    };

    popup.appendChild(button);
    document.body.appendChild(popup);
}
```

- Crea dinámicamente un cuadro de diálogo personalizado para mostrar mensajes al usuario.
- Este cuadro incluye un botón "OK" para cerrarlo.
- El cuadro de diálogo advierte al usuario el número de intentos fallidos en el inicio de sesión.

-Gestión de intentos fallidos

```
function getFailedAttempts() {
    return parseInt(localStorage.getItem('failedAttempts') || '0');
}

function incrementFailedAttempts() {
    let attempts = getFailedAttempts();
    attempts += 1;
    localStorage.setItem('failedAttempts', attempts);

    if (attempts >= MAX_ATTEMPTS) {
        localStorage.setItem('lockout', Date.now());
    }
}

function isLockedOut() {
    const lockoutTime = parseInt(localStorage.getItem('lockout') || '0');
    if (lockoutTime && Date.now() - lockoutTime < LOCKOUT_TIME) {
        return true;
    }
    return false;
}

function resetFailedAttempts() {
    localStorage.removeItem('failedAttempts');
    localStorage.removeItem('lockout');
}
```

- *getFailedAttempts()*: Obtiene el número de intentos fallidos almacenados en el navegador.
- *incrementFailedAttempts()*: Incrementa el contador de intentos fallidos. Si se alcanza el límite, activa un bloqueo temporal.
- *isLockedOut()*: Verifica si el usuario está bloqueado y aún dentro del período de bloqueo.
- *resetFailedAttempts()*: Restablece el contador de intentos fallidos y elimina el estado de bloqueo.

-Autenticación y redirección

```
const userDID = localStorage.getItem('userDID');
if (userDID) {
    // Si ya está autenticado, redirigir directamente al dashboard
    window.location.href = 'dashboard.html';
    return;
}
```

- Si un DID (Decentralized Identifier) ya está almacenado en el navegador, el usuario se redirige automáticamente al dashboard.

-Manejo del evento envío de formulario

```
loginForm.addEventListener('submit', async function (e) {
    e.preventDefault();

    if (isLockedOut()) {
        createPopup('Too many failed attempts. Please try again later.');
        return;
    }

    const address = document.getElementById('address').value;
    const privateKey = document.getElementById('privateKey').value;
```

Prevención del comportamiento por defecto:

- El evento de envío se intercepta para ejecutar el código personalizado de validación y autenticación.

Verificación de bloqueo:

- Si el usuario está bloqueado, muestra una ventana emergente de advertencia y detiene el proceso.

-Validación de credenciales

```
try {
    // Crear una wallet con ethers.js
    const wallet = new ethers.Wallet(privateKey);

    // Verificar que la dirección proporcionada sea correcta
    if (wallet.address.toLowerCase() !== address.toLowerCase()) {
        incrementFailedAttempts();
        createPopup('Warning. Check your address or your private key and try again.');
        return;
    }

    // Generar un mensaje para firmar
    const message = `Authenticate as ${address}`;
    const signature = await wallet.signMessage(message);

    // Generar el DID a partir de la dirección Ethereum
    const did = `did:ethr:${wallet.address}`;

    // Enviar la firma y el DID al servidor
    const response = await fetch('/auth', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
```

```
        },
        body: JSON.stringify({
            address: wallet.address,
            did: did,
            signature: signature
        })
    });

    const result = await response.json();
    if (result.success) {
        resetFailedAttempts();
        localStorage.setItem('userDID', did);
        window.location.href = 'dashboard.html';  // Redirigir al dashboard
    } else {
        incrementFailedAttempts();
        createPopup('Authentication failed: ' + result.message);
    }
} catch (error) {
    incrementFailedAttempts();
    createPopup('Error during authentication. Please try again.');
}
```

Validación de la dirección:

- ☐ Compara la dirección Ethereum generada con la proporcionada por el usuario.

Firma de Mensaje:

- ☐ Genera un mensaje que el usuario debe firmar con su clave privada.

Generación del DID:

- ☐ Crea un Decentralized Identifier basado en la dirección Ethereum.

Autenticación con el Servidor:

- ☐ Envía la dirección, el DID y la firma al servidor para su validación.

Manejo de Errores:

- ☐ Incrementa los intentos fallidos si hay errores y muestra una ventana emergente adecuada.

2.4. Dashboard.js

-Configuración inicial del contrato

```
const contractABI = [ ···
];

const contractAddress = '0xc96a108cd7A30154353091752Fcf50807a10ADf3';

const provider = new ethers.JsonRpcProvider("http://127.0.0.1:7545");

// Declarar la clave privada
const privateKey = localStorage.getItem('privateKey');

// Validar si la clave privada es válida
if (!privateKey || !privateKey.startsWith('0x') || privateKey.length !== 66) {
    alert('Invalid private key. Redirecting to login...');
    window.location.href = 'login.html';
}

// Crear wallet con la clave privada
const wallet = new ethers.Wallet(privateKey, provider);
const contract = new ethers.Contract(contractAddress, contractABI, wallet);
```

- **ABI**: Define la interfaz del contrato inteligente. Contiene información sobre las funciones y eventos disponibles.
- **Contract Address**: Dirección del contrato desplegado en Ganache.
- **Provider**: Conecta la aplicación con Ganache como proveedor RPC.
- **Private Key**: Recupera la clave privada almacenada en el navegador al iniciar sesión. Esta clave se usa para firmar transacciones.

-Mostrar información del usuario

```
const did = localStorage.getItem('userDID'); // Obtener el DID desde el almacenamiento local
const logoutButton = document.getElementById('logout');

if (!did) {
    alert('No session found. Redirecting to login...');
    window.location.href = 'login.html';
    return;
}

try {
    // Mostrar el DID en el dashboard
    document.getElementById('did').textContent = did;

    // Obtener información del balance desde el servidor
    const response = await fetch(`/balance/${did}`);
    const data = await response.json();
```

- Recupera el DID generado durante el inicio de sesión.
- Se asegura de que el usuario esté autenticado; en caso contrario, redirige al login.
- Muestra el DID del usuario en el dashboard.

-Generar autos dinámicamente

```
const carModels = [
    'Toyota Supra', 'Nissan GT-R', 'Chevrolet Camaro', 'Ford Mustang', 'Dodge Charger',
    'BMW M3', 'Audi R8', 'Mercedes AMG GT', 'Porsche 911', 'Lamborghini Huracan',
    'Ferrari 488', 'McLaren 720S', 'Aston Martin DB11', 'Jaguar F-Type', 'Mazda RX-7',
    'Subaru WRX', 'Honda NSX', 'Tesla Model S', 'Koenigsegg Agera', 'Bugatti Chiron',
    'Pagani Huayra', 'Lexus LC500', 'Alfa Romeo Giulia', 'Volkswagen Golf GTI',
    'Mitsubishi Lancer Evo', 'Hyundai Veloster N', 'Kia Stinger', 'Ford Focus RS',
    'Chevrolet Corvette', 'Dodge Viper', 'Ferrari F40', 'Lamborghini Aventador',
    'Bentley Continental GT', 'Rolls-Royce Wraith', 'Mini Cooper S', 'Jeep Grand Cherokee',
    'Range Rover Sport', 'Volvo XC90', 'Tesla Model X', 'Porsche Cayenne'
];
```

```
const carList = document.getElementById('car-list');
const cars = [];

for (let i = 1; i <= 40; i++) {
    const available = Math.random() > 0.5;
    const car = {
        id: i,
        model: carModels[i - 1],
        price: (Math.random() * (50 - 10) + 10).toFixed(3),
        available: available,
        image: `images/cars/car${i}.png`
    };
    cars.push(car);
```

- Lista de 40 modelos de autos predefinidos.
- **Disponibilidad aleatoria:** Los autos se marcan como disponibles o no disponibles al azar.
- **Precios aleatorios:** Se generan entre 10 y 50 ETH.
- **Imágenes dinámicas:** Cada auto tiene una imagen específica asociada.

-Mostrar los autos en el dashboard

```
const carCard = document.createElement('div');
carCard.className = 'col-md-3 mb-4';
carCard.innerHTML = `
    <div class="card">
        <img src="${car.image}" class="card-img-top" alt="Car Image">
        <div class="card-body">
            <h5 class="card-title">${car.model}</h5>
            <p class="card-text">Price: ${car.price} ETH</p>
            <p class="card-text">Available: ${car.available ? '<span style="color:green;">✓ Yes</span>' : '<span style="color:red;">✕ No</span>'}
            <button class="btn btn-primary rent-btn" data-id="${car.id}" ${!car.available ? 'disabled' : ''}>Rent</button>
        </div>
    </div>
`;
carList.appendChild(carCard);
```

- **Tarjeta del auto:** Muestra el modelo, el precio y la disponibilidad.
- **Botón "Rent":** Aparece habilitado si el auto está disponible y deshabilitado si no lo está.

-Proceso de alquiler

```
document.querySelectorAll('.rent-btn').forEach(button => {
    button.addEventListener('click', async (e) => {
```

- Asocia un evento al botón "Rent" para cada auto.

```
const userDID = prompt('Enter your DID:');
const inputDID = window.confirm(`Is this your DID? ${userDID}`);
```

- Solicita el **DID** para validar la identidad del usuario antes de proceder.
- Si el **DID** es incorrecto, muestra un mensaje de error.

```
const tx = await contract.rentCar(car.id, {
    value: ethers.parseEther(car.price),
    gasLimit: 300000
});
await tx.wait();
```

- Llama a la función rentCar en el contrato inteligente.
- Envía el pago en ETH y especifica un límite de gas.
- Espera la confirmación de la transacción antes de continuar.

```
car.available = false;
e.target.disabled = true;
location.reload();
```

- Marca el auto como no disponible.
- Recarga la página para actualizar la información.

-Cerrar sesión

```
logoutButton.addEventListener('click', async () => {
    await fetch('/logout', { method: 'POST' });
    localStorage.clear();
    window.location.href = 'login.html';
});
```

- Borra los datos almacenados en localStorage.
- Redirige al usuario a la página de inicio de sesión.

2.5. Server.js

Este archivo actúa como el servidor backend de la aplicación. Su principal objetivo es:

1. Servir los archivos estáticos del frontend.
2. Proveer rutas API para la autenticación, obtención de saldo y cierre de sesión.
3. Conectarse a Ganache para interactuar con la blockchain simulada.
4. Validar firmas digitales y DID (Decentralized Identifiers).

-Configuración inicial

```
const express = require('express');
const bodyParser = require('body-parser');
const { ethers } = require('ethers');
const app = express();

app.use(bodyParser.json());
app.use(express.static('public')); // Servir archivos estáticos
```

- *express*: Framework utilizado para gestionar rutas y solicitudes HTTP.
- *bodyParser*: Middleware que analiza el cuerpo de las solicitudes entrantes como JSON.
- *ethers*: Biblioteca utilizada para interactuar con la blockchain Ethereum.
- *app.use(express.static('public'))*:
    - Sirve archivos estáticos desde la carpeta *public*, como HTML, CSS y JS.

-Conexión con Ganache

```
let provider;
try {
    provider = new ethers.JsonRpcProvider('http://127.0.0.1:7545');
    console.log('Connected to Ganache on http://127.0.0.1:7545');
} catch (error) {
    console.error('Error connecting to Ganache:', error.message);
    process.exit(1);
}
```

- Establece la conexión con Ganache, una blockchain local para pruebas.
- *JsonRpcProvider*: Se utiliza para comunicarse con Ganache a través de JSON-RPC.
- Si la conexión falla, el servidor se detiene con un mensaje de error.

-Ruta: Autenticación DID y Firma

```
app.post('/auth', async (req, res) => {
    const { address, did, signature } = req.body;

    try {
        // Mensaje original que se firmó
        const message = `Authenticate as ${address}`;

        // Verificar la firma del mensaje
        const recoveredAddress = ethers.verifyMessage(message, signature);

        if (recoveredAddress.toLowerCase() === address.toLowerCase()) {
            console.log('Firma verificada, autenticación exitosa.');

            // (Opcional) Emitir una Verifiable Credential (VC)
            const verifiableCredential = {
                "@context": ["https://www.w3.org/2018/credentials/v1"],
                "type": ["VerifiableCredential"],
                "issuer": did,
                "credentialSubject": {
                    "id": did,
                    "walletAddress": address
                }
            };
```

```
            // Aquí se puede firmar la VC o simplemente devolverla
            res.json({ success: true, credential: verifiableCredential });
        } else {
            res.json({ success: false, message: 'Signature verification failed' });
        }
    } catch (error) {
        console.error('Error during verification:', error);
        res.status(500).json({ success: false, message: 'Error during verification' });
    }
});
```

1. Extrae la dirección, el DID y la firma del cuerpo de la solicitud.
2. Verifica que la firma sea válida utilizando *ethers.verifyMessage*.
3. Si la firma es válida:
    - Genera un **Verifiable Credential (VC)** basado en el DID y la dirección de la wallet.

- Devuelve el VC como parte de la respuesta JSON.

4. Si falla la verificación, responde con un mensaje de error.

-Obtener balance

```
app.get('/balance/:did', async (req, res) => {
    const { did } = req.params;

    try {
        // Extraer la dirección Ethereum del DID (asumiendo que el formato es 'did:ethr:{address}')
        const address = did.split(':')[2];  // La dirección está en la tercera parte del DID

        // Obtener el balance de la cuenta de Ethereum
        const balance = await provider.getBalance(address);

        // Convertir el balance a ETH legible
        const balanceInEth = ethers.formatEther(balance);

        res.json({ success: true, address: address, balance: balanceInEth });
    } catch (error) {
        console.error('Error fetching balance:', error);
        res.status(500).json({ success: false, message: 'Error fetching balance' });
    }
});
```

1. Extrae el DID del usuario de la URL.
2. Obtiene la dirección Ethereum del DID, que sigue el formato *did:ethr:{address}*.
3. Usa *provider.getBalance* para recuperar el balance asociado a la dirección.
4. Convierte el balance a ETH legible mediante *ethers.formatEther*.
5. Devuelve el balance y la dirección en un JSON.

-Cierre de sesión

```
app.post('/logout', (req, res) => {
    // Aquí se maneja cualquier lógica de cierre de sesión necesaria (limpiar sesiones, cookies, etc.)

    console.log('El usuario ha cerrado sesión exitosamente.');

    // Devolver una respuesta exitosa en formato JSON
    res.json({ success: true, message: 'Logout successful' });
});
```

- Proporciona una ruta para manejar el cierre de sesión del usuario.
- Registra el evento de cierre de sesión en la consola.
- Responde con un mensaje de éxito.

-Arranque del servidor

```
app.listen(3000, () => {
    console.log('Server running on http://localhost:3000');
});
```

- El servidor se inicia en el puerto **3000**.
- Pone en marcha el servidor para aceptar solicitudes de la aplicación cliente.

-