

Practica 2 - IOTA

Para esta práctica exploraremos sobre los smart contracts sobre IOTA. Para desarrollar la práctica nos basaremos en un ejemplo básico de Smart Contracs sobre Solidity, que permite registrar y actualizar nodos IoT, definiendo un ID de dispositivo, propietario y datos asociados al mismo.

Parte I

Explicación de las diferentes tecnologías utilizadas, estándares, herramientas, librerías, y código de terceros utilizado.

Para desarrollar la práctica utilizaremos las tecnologías y herramientas que se describen a continuación.

IOTA

Es una tecnología de registros distribuidos (DLT), que trabaja bajo una estructura del tipo Tangles, formando un gráfico acíclico, donde las nuevas transacciones validan a las anteriores y se puede ejecutar de manera descentralizada y paralela, a diferencia de blockchain que agrupan una serie de transacciones en bloques enlazadas entre sí.

A nivel de criptomonedas comparándolas con otras tecnologías como blockchain, aquí no se cobran comisiones por las transacciones, el dinero que envía el remitente es el mismo que recibe el destinatario, por lo que no hay intermediarios o mineros que se queden con una porción de la transacción.

Para entender un poco más el comportamiento y la arquitectura de IOTA nos basamos en información contenida en el siguiente blog:

Muñoz, Juan Manuel. (30/05/2018). IOTA – The Tangle, El blog de IoT en Español, IoT Futura. <https://iotfutura.com/arquitecturas-iot/iota-the-tangle>.

Smart Contracts.

Son programas informáticos que se despliegan sobre una cadena de bloques, que para ejecutarse deben cumplir ciertas condiciones, donde todos los participantes forman parte del proceso de validación, sin depender de una autoridad o mediador, los mismos se ejecutan por sí solos a través de scripts definidos en un código programable..

Solidity.

Es un lenguaje de programación de alto nivel orientado a objetos, diseñado exclusivamente para diseñar smart contracts, su sintaxis está basada en ECMAScript, donde la declaración de variables tiene bastante poder al momento de usarlas sobre todo para darle rigor al momento de programar un SC. Es un lenguaje que fue creado para ejecutarse en las EVM de ethereum, pero también se puede ejecutar sobre otras redes donde su comportamiento

sigue siendo el mismo, Su compilador analizará el código del contrato en tiempo de ejecución para verificar que intentamos realizar la operación adecuada con el tipo de valor adecuado.

Remix.

Es un entorno de desarrollo que permite crear, compilar y desplegar contratos inteligentes en la red blockchain de Ethereum, Ofrece opciones para desplegar contratos en redes locales, de prueba o la red principal, permite realizar pruebas directamente desde la interfaz llamando a funciones del contrato.

IOTA EVM Testnet.

Portal para obtener los token de testnet de IOTA que transferimos a nuestra Wallet para poder costear las transacciones de nuestro Smart Contracts en REMIX.

<https://evm-toolkit.evm.testnet.iotaledger.net/>

Wallet.

Usaremos **METAMASK** que no es más que un software de criptomonedas que brinda una extensión para nuestro navegador web, en nuestro caso usaremos Google Chrome, dicho monedero nos va a permitir a interactuar con las distintas redes de blockchain.

Explorer EVM Testnet Explorer.

Herramienta que nos va permitir escanear y verificar el despliegue de nuestro SC en la red de IOTA.

<https://explorer.evm.testnet.iotaledger.net/>

Parte II

Explicación de las distintas partes del código.

Para desarrollar nuestro código tomamos como referencia los siguientes ejemplos de distintos repositorios de GITHUB:

<https://github.com/kriss07in/IoT-with-Blockchain/blob/master/SensorRepo1.sol>

<https://github.com/nasserhaji/ParkingManagement/blob/main/ParkingManagement-A.sol>

<https://github.com/antonio-polastris/iot-solidity-sc/blob/main/IOT.sol>

En color verde a modo de comentario se explica cada una de las partes del código.

```
// SPDX-License-Identifier: MIT //Licencia de software
pragma solidity ^0.8.28; //Version de solidity

contract IoTDeviceManager { // Nombre del contrato

//Estructura que representa todos los datos de los dispositivos.

    struct Device {
//Información de los datos del dispositivo, que se cargarán al momento
del registro.
        string deviceId;
        address owner;
        string data;
        uint256 timestamp;
        string sensorType;
        string location;
        string client;
        bool isActive;
        uint256 installationDate;
        string protocolType;
        string brand;
        string model;
    }

//Devices, función que almacena los datos de los dispositivos según su
ID,

    mapping(string => Device) private devices;

//deviceId, es un arreglo que que aloja a los dispositivos segun su ID.

    string[] private deviceIds;
//Marca de tiempo que se genera cada vez que se registra o modifica un
dispositivo.

    event DeviceUpdated(string deviceId, string data, uint256
timestamp);

//Función para registrar un nuevo dispositivo y que dispara un
almacenamiento en memoria, para luego alojar los datos en el mapping
devices y los ID en el array deviceId.

    function registerDevice(
```

```
    string memory _deviceId,  
    string memory _sensorType,  
    string memory _location,  
    string memory _client,  
    uint256 _installationDate,  
    string memory _protocolType,  
    string memory _brand,  
    string memory _model  
    ) public {  
        require(bytes(devices[_deviceId].deviceId).length == 0, "Device  
already registered");  
  
        devices[_deviceId] = Device({  
            deviceId: _deviceId,  
            owner: msg.sender,  
            data: "",  
            timestamp: block.timestamp,  
            sensorType: _sensorType,  
            location: _location,  
            client: _client,  
            isActive: true,  
            installationDate: _installationDate,  
            protocolType: _protocolType,  
            brand: _brand,  
            model: _model  
        });  
  
        deviceIds.push(_deviceId);  
    }  
  
//Función para actualizar los datos, requiere que el dispositivo se  
encuentre registrado y que la ejecute el propietario del mismo, en este  
caso la misma address del usuario que lo registró.  
  
    function updateDeviceData(string memory _deviceId, string memory  
_data) public {  
        require(bytes(devices[_deviceId].deviceId).length != 0, "Device  
not found.");  
        require(devices[_deviceId].owner == msg.sender, "No eres el  
propietario del dispositivo.");  
  
        devices[_deviceId].data = _data;  
    }
```

```
        devices[_deviceId].timestamp = block.timestamp;

//Emisión de evento de actualización del dispositivo y marca de tiempo
de ejecución.

        emit DeviceUpdated(_deviceId, _data, block.timestamp);
    }

//Función auxiliar para convertir uint256 a string para mostrar los
datos de manera legible cuando se vaya a ejecutar un getdevice.

    function uintToString(uint256 _value) private pure returns (string
memory) {
        if (_value == 0) {
            return "0";
        }
        uint256 temp = _value;
        uint256 digits;
        while (temp != 0) {
            digits++;
            temp /= 10;
        }
        bytes memory buffer = new bytes(digits);
        while (_value != 0) {
            buffer[--digits] = bytes1(uint8(48 + _value % 10));
            _value /= 10;
        }
        return string(buffer);
    }

//Función para obtener información detallada del dispositivo con
descripción legible que usa funciones auxiliares uinttostring y
addresstostring, también requiere que el dispositivo se haya registrado
previamente.

    function getDevice(string memory _deviceId)
        public
        view
        returns (
            string memory ownerDescription,
            string memory dataDescription,
            string memory timestampDescription,
```

```
        string memory sensorTypeDescription,
        string memory locationDescription,
        string memory clientDescription,
        string memory isActiveDescription,
        string memory installationDateDescription,
        string memory protocolTypeDescription,
        string memory brandDescription,
        string memory modelDescription
    )
}

    require(bytes(devices[_deviceId].deviceId).length != 0, "Device
not found.");

    Device memory device = devices[_deviceId];

//Definir descripciones elegibles para la posterior visualización.

    ownerDescription = string(abi.encodePacked("Propietario del
dispositivo: ", addressToString(device.owner)));
    dataDescription = string(abi.encodePacked("Datos del sensor: ",
device.data));

//Convertir timestamp a string y concatenar sin caracteres especiales.

    string memory timestampStr = uintToString(device.timestamp);
    timestampDescription = string(abi.encodePacked("Ultima
actualizacion (timestamp): ", timestampStr));

    sensorTypeDescription = string(abi.encodePacked("Tipo de
sensor: ", device.sensorType));
    locationDescription = string(abi.encodePacked("Ubicacion: ",
device.location));
    clientDescription = string(abi.encodePacked("Cliente asociado:
", device.client));
    isActiveDescription = device.isActive ? "Estado del
dispositivo: Activo" : "Estado del dispositivo: Inactivo";
    installationDateDescription = string(abi.encodePacked("Fecha de
instalacion: ", uintToString(device.installationDate)));
    protocolTypeDescription = string(abi.encodePacked("Tipo de
protocolo: ", device.protocolType));
    brandDescription = string(abi.encodePacked("Marca del
dispositivo: ", device.brand));
```

```
        modelDescription = string(abi.encodePacked("Modelo del
dispositivo: ", device.model));

    return (
        ownerDescription,
        dataDescription,
        timestampDescription,
        sensorTypeDescription,
        locationDescription,
        clientDescription,
        isActiveDescription,
        installationDateDescription,
        protocolTypeDescription,
        brandDescription,
        modelDescription
    );
}

//Función auxiliar para convertir address a string.

function addressToString(address _address) private pure returns
(string memory) {
    bytes32 value = bytes32(uint256(uint160(_address)));
    bytes memory alphabet = "0123456789abcdef";
    bytes memory str = new bytes(42);
    str[0] = '0';
    str[1] = 'x';
    for (uint i = 0; i < 20; i++) {
        str[2+i*2] = alphabet[uint8(value[i + 12] >> 4)];
        str[3+i*2] = alphabet[uint8(value[i + 12] & 0x0f)];
    }
    return string(str);
}

//Función para cambiar el estado del dispositivo (activo/inactivo.

function toggleDeviceStatus(string memory _deviceId, bool
_isActive) public {
    require(bytes(devices[_deviceId].deviceId).length != 0, "Device
not found.");
    require(devices[_deviceId].owner == msg.sender, "No eres el
propietario del dispositivo.");
}
```

```
        devices[_deviceId].isActive = _isActive;
    }

//Función pública para obtener el estado de un dispositivo (solo
algunos datos.

function getDeviceStatus(string memory _deviceId)
    public
    view
    returns (
        string memory,
        bool
    )
{
    require(bytes(devices[_deviceId].deviceId).length != 0, "Device
not found.");

    Device memory device = devices[_deviceId];
    return (
        device.deviceId, // Devuelve el ID del dispositivo
        device.isActive // Devuelve si el dispositivo está activo
    );
}
}
```

Las funciones auxiliares para la conversión de datos a string, tiene como objetivo dar una mejor visibilidad al momento de hacer una búsqueda de manera que los datos sean legibles.

Se adjunta archivo. **Practica2_IOTA.sol**

Parte III

Explicación de las pruebas realizadas.

Primeramente, obtenemos las faucet necesarias para poder desplegar nuestro SC sobre la red de pruebas de IOTA.

<https://evm-toolkit.evm.testnet.iotaedger.net/>

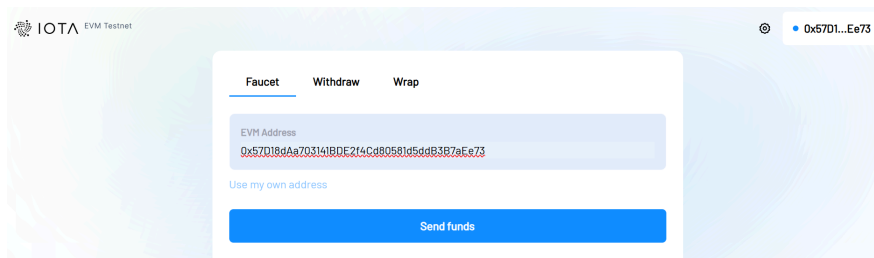


Imagen 1. Adquisición de tokens para testnet.

Haciendo uso de la Wallet Metamask, enviamos las testnet de IOTA, la misma se configuró para poder conectarse a la red de IOTA.



Imagen 2. Tokens en metamask.

En REMIX IDE no es posible desplegar nuestro SC en una VM de prueba como lo hicimos en la primera práctica, ya que la idea de esta práctica es desplegar nuestro SC en la red de pruebas de IOTA y no de Ethereum.

Procedimos a crear nuestro proyecto y a configurar nuestro entorno de trabajo, definiendo la versión de solidity a utilizar e inyectar la wallet que utilizaremos para poder desplegar nuestro contrato.

Contrato compilado.

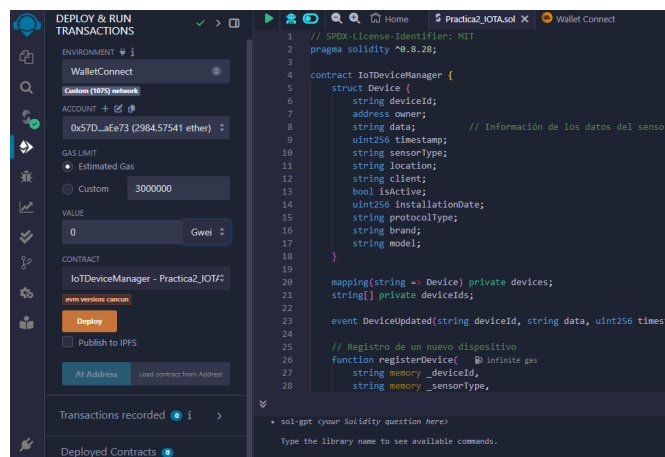


Imagen 3. Contrato compilado en Remix

Deploy.

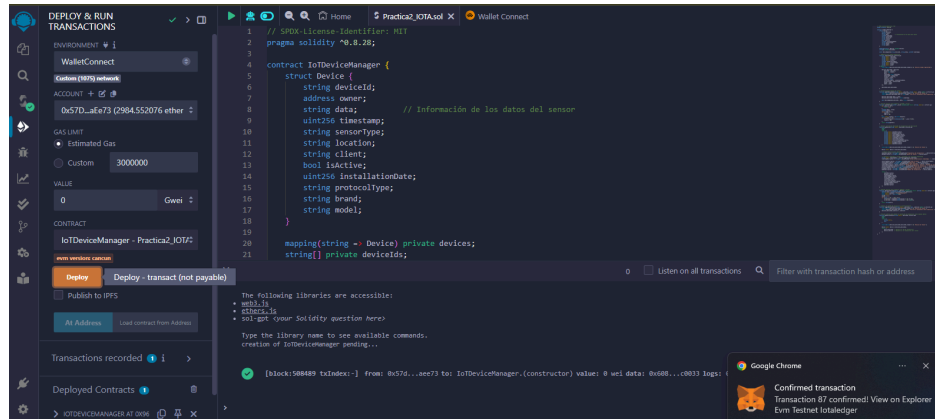


Imagen 4. Deploy del contrato

Fee por Deploy.

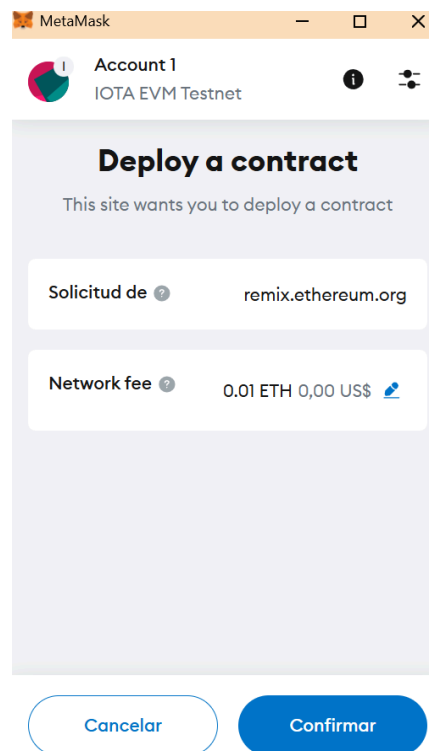


Imagen 5. Fee por deploy del contrato.

Registrar un Dispositivo.

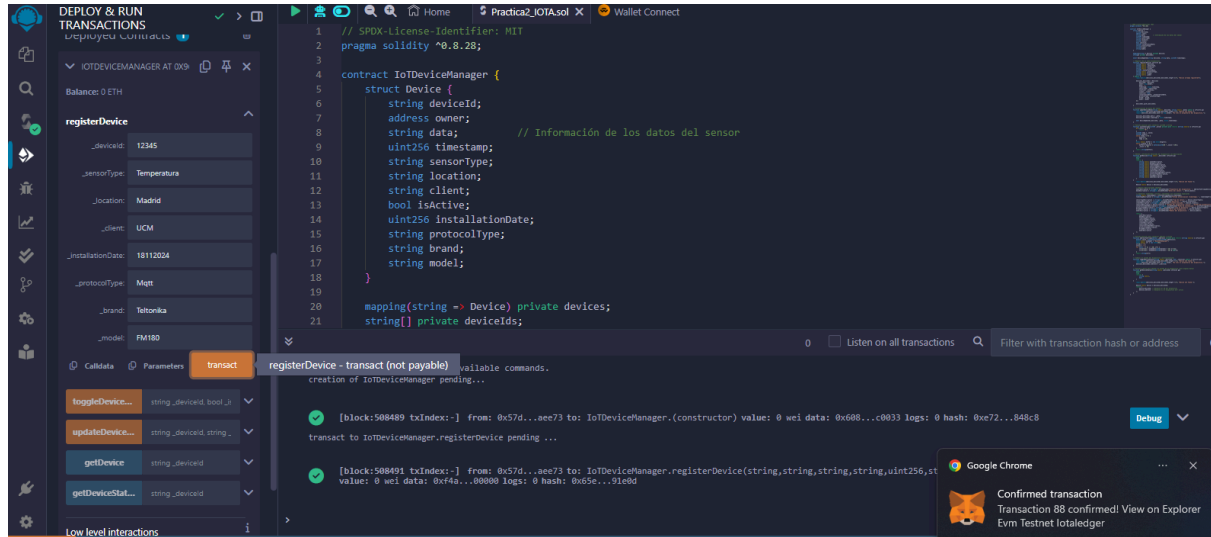


Imagen 6. Registro de dispositivo.

Si no se completan todos los campos no se habilita la opción para ejecutar la transacción, por lo que es necesario completarlos en su totalidad.

This is a close-up of the 'registerDevice' form. It contains the following fields and values: _deviceId: 12345, _sensorType: Temperatura, _location: Madrid, _client: UCM, _installationDate: 18112024, _protocolType: Mqtt, _brand: Teltonika, and _model: string. At the bottom, there are three buttons: 'Calldata', 'Parameters', and 'transact'.

Imagen 7. Carga de datos para el registro.

Si el campo **installationData** se completa con caracteres especiales, el registro no se ejecuta y arroja error.

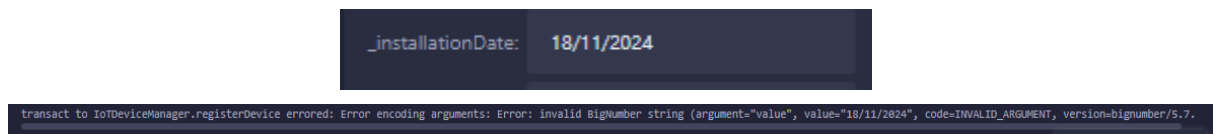


Imagen 8. Error por carga errónea de fecha de instalación.

Registrar un segundo dispositivo:

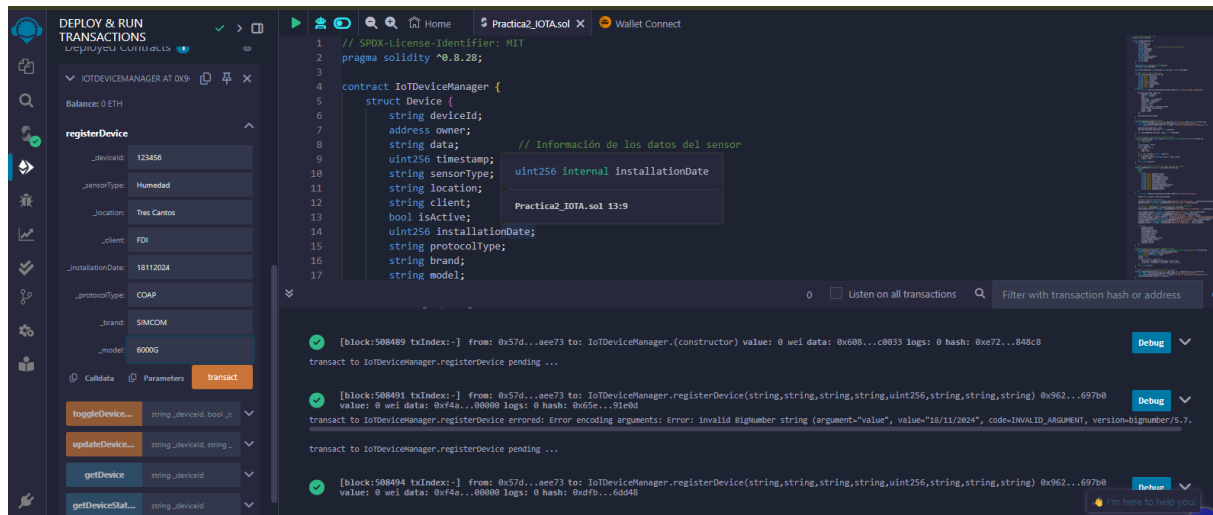


Imagen 9. Registro de dispositivo 2.

Registro de un tercer dispositivo.

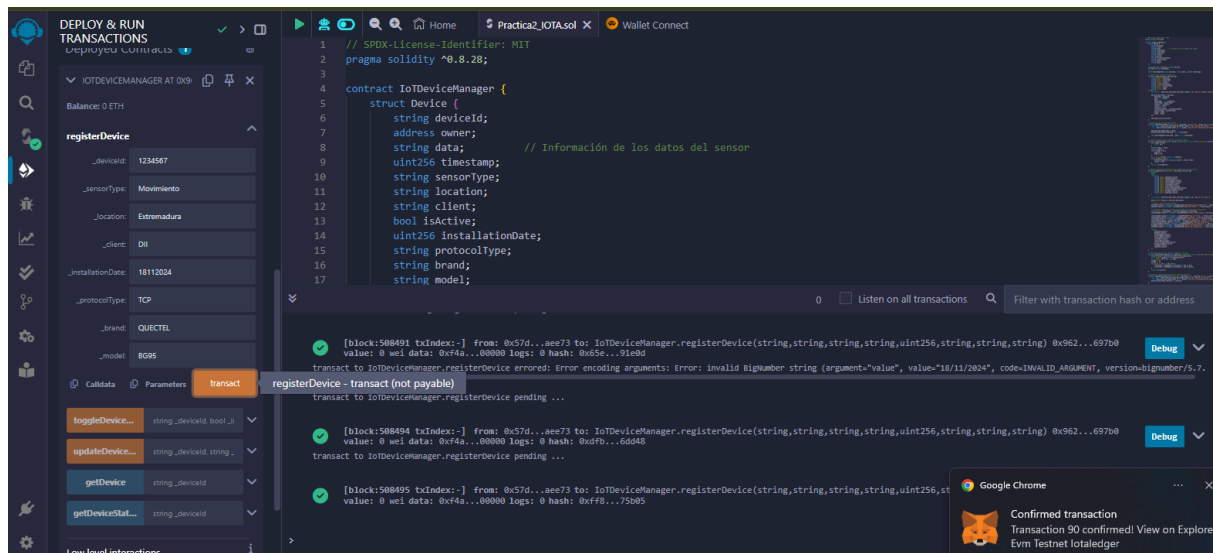


Imagen 10. Registro de dispositivo 3.

Utilización de la función **Toggledevice**, esta función permite cambiar el estado del dispositivo.

- true = activo

- false = inactivo

El contrato se definió para que cuando se registra el dispositivo ingresa en un estado inicial activo. false o true se deben ingresar en letra minúscula, si se ingresan datos distintos o en mayúscula dará error.

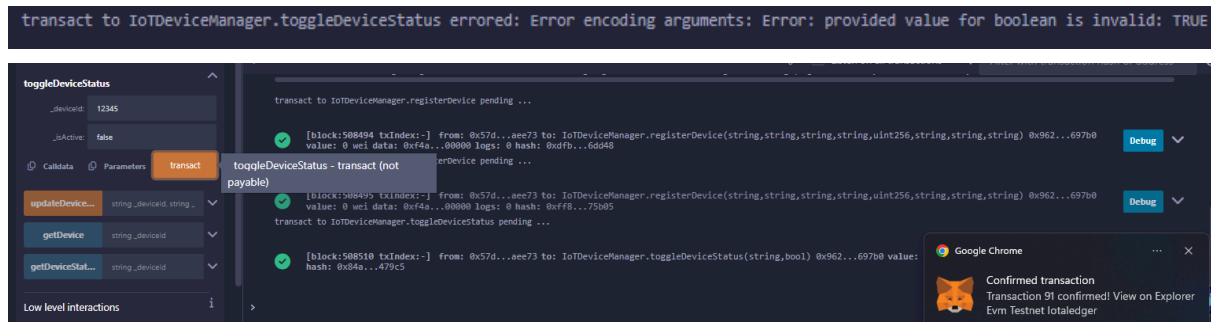


Imagen 11. Cambio de estado de un dispositivo.

Si el id del dispositivo es erróneo o no se encuentra registrado la transacción dará error.

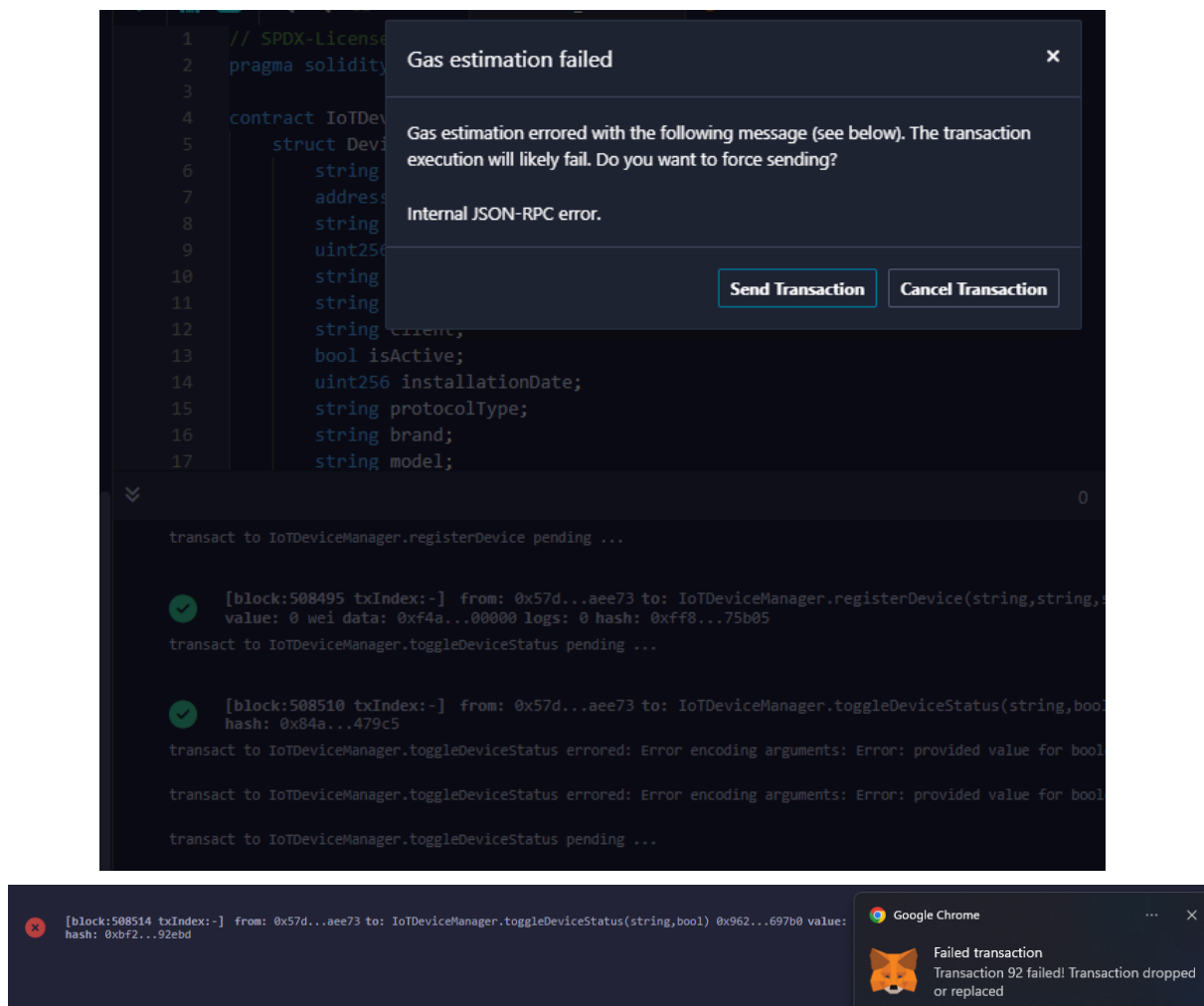


Imagen 12. Error cambio de estado si el dispositivo no se encuentra registrado.

Funcion **updatedevicedata**, acà podremos cargar datos adicionales asociados a cada dispositivo.

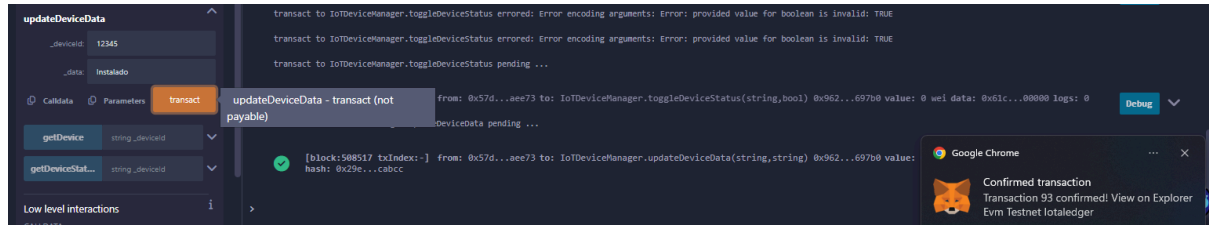


Imagen 13. Carga de datos adicionales asociados a cada dispositivo.

El dispositivo a actualizar tiene que estar previamente registrado.

La función **getdevice** permite consultar todos los datos de un dispositivo registrado solo con el número de ID.

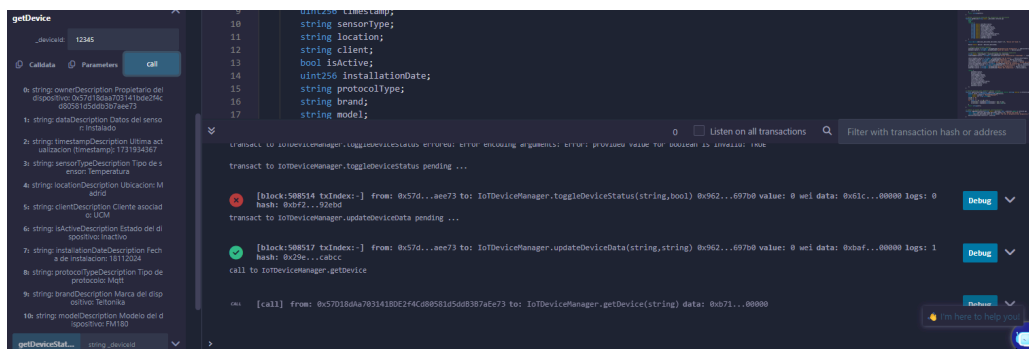


Imagen 14. Función para consultar los datos de un dispositivo registrado.

Ahora vamos a probar la función, **getdevicestatus**, que nos permite consultar si un dispositivo está activo o no, ingresando su número de ID.

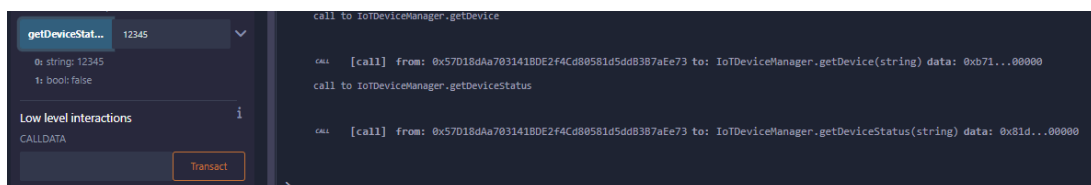


Imagen 15. Consultar el estado de un dispositivo.

Recordando que true = activo y false = inactivo.

Luego accediendo a la red de pruebas de IOTA EVM TESTNET.

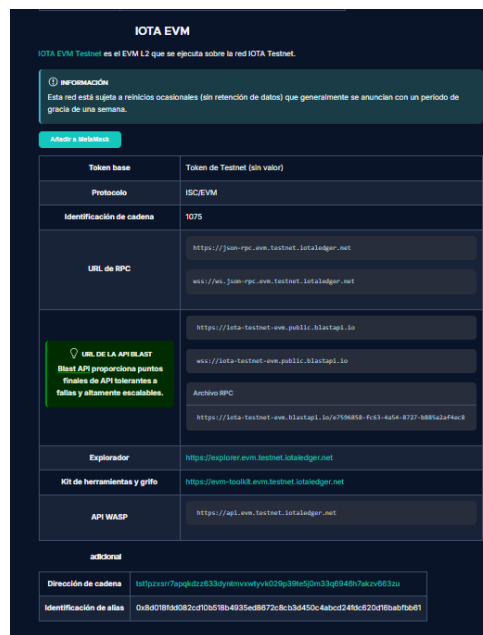
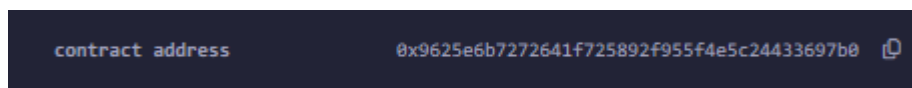


Imagen 16. Fuente: <https://wiki.iota.org/build/networks-endpoints/#shimmerevm-testnet>

<https://explorer.evm.testnet.iotaledger.net>

Buscando por la dirección del contrato.



Contract Address: 0x9625e6b7272641f725892f955f4e5c24433697b0

Se pueden observar los detalles del contrato desplegado con el historial de todas la transacciones realizadas como se observa en la siguiente figura:

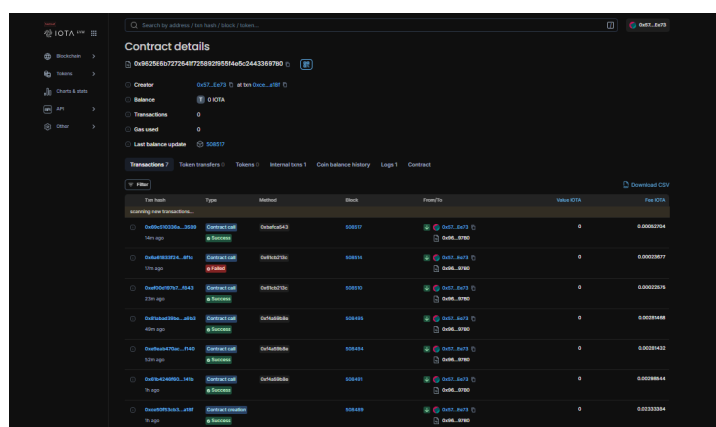


Imagen 17. Historial de verificación de cada una de las transacciones.

Transacción asociada a la creación de contrato, cuando ejecutamos el deploy.

Transaction details

0xce50f53cb3e2155d5051f97679ccd035f9ed0e3a4727261e54ad924069cca18f

Details Token transfers Internal txns Logs State Raw trace

This is a testnet transaction only

| | |
|--------------------------|----------------------------------------------------------------------------------|
| Transaction hash | 0xce50f53cb3e2155d5051f97679ccd035f9ed0e3a4727261e54ad924069cca18f |
| Status and method | Success |
| Block | 508489 32 Block confirmations |
| Timestamp | 1h ago Nov 18 2024 13:00:16 PM (+01:00 UTC) Confirmed within <= 326.971 secs |
| From | 0x57D18dAa703141BDE2f4Cd80581d5ddB3B7aEe73 |
| To | [Contract 0x9625E6b7272641f725892f955f4e5c24433697B0 created] |
| Value | 0 IOTA |
| Transaction fee | 0.02333384 IOTA |
| Gas price | 0.00000001 IOTA (10 Gwei) |
| Gas usage & limit by txn | 2,333,384 2,333,384 100% |

Imagen 18. Transacción por la creación del contrato.

Transacción asociada al registro del primer dispositivo con ID 12345

Transaction details

0x57...Ee73 called 0xf4a59b8e on 0x96...97B0

Details Token transfers Internal txns Logs State Raw trace

This is a testnet transaction only

| | |
|--------------------------|----------------------------------------------------------------------------------|
| Transaction hash | 0x61b4246f60d1b45b61ec294a210a28fe4b93489d4978f83d2255da8d9a91141b |
| Status and method | Success 0xf4a59b8e |
| Block | 508491 30 Block confirmations |
| Timestamp | 1h ago Nov 18 2024 13:06:10 PM (+01:00 UTC) Confirmed within <= 328.568 secs |
| From | 0x57D18dAa703141BDE2f4Cd80581d5ddB3B7aEe73 |
| Interacted with contract | 0x9625E6b7272641f725892f955f4e5c24433697B0 |
| Value | 0 IOTA |
| Transaction fee | 0.00298544 IOTA |
| Gas price | 0.00000001 IOTA (10 Gwei) |
| Gas usage & limit by txn | 298,544 298,544 100% |

Imagen 19. Registro del primer dispositivo.

Transacción en REMIX.

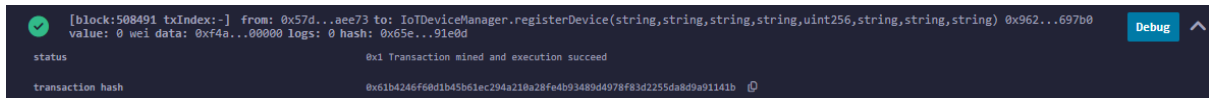


Imagen 20. Registro del primer dispositivo en REMIX.

Mismo flujo y detalle para los dispositivos 2 y 3 registrados.

Detalle de la transacción cuando se cambió el estado del primer dispositivo registrado.

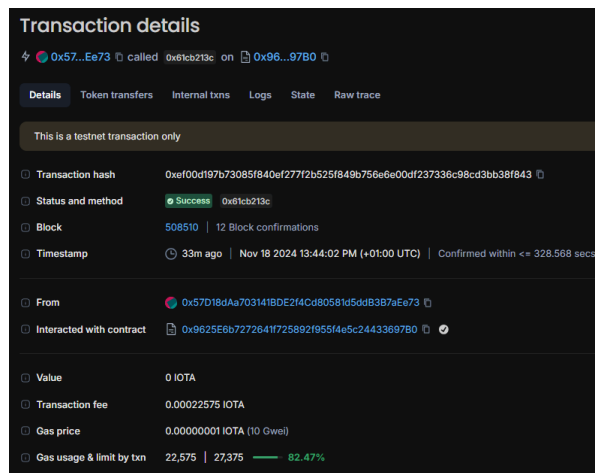


Imagen 21. Transacción por cambio de estado.

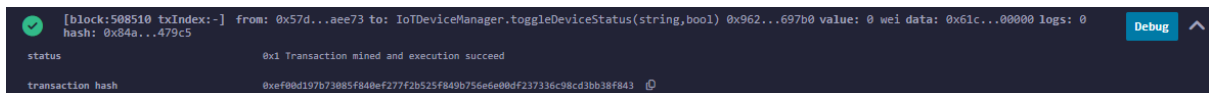


Imagen 22. Transacción por cambio de estado en REMIX.

Detalle de transacción fallida cuando intentamos cambiar el estado de un dispositivo con ID erróneo o dispositivo no registrado.

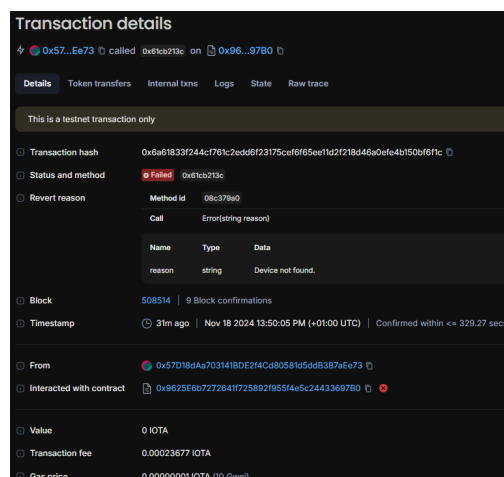


Imagen 23. Transacción errónea de cambio de estado..



Imagen 24. Transacción errónea por cambio de estado en REMIX.

Parte IV

Guía de usuario.

A continuación se explica el paso a paso para hacer uso de la aplicación definida en nuestro contrato.

Antes que nada una breve introducción, la aplicación consiste en un sistema que permite registrar, gestionar y controlar dispositivos IoT en nuestro caso sensores de cualquier tipo.

Función 1 registrar los dispositivos en la red cargando los siguientes datos:

Hacer click en la función RegisterDevice.

A screenshot of a web application form titled 'registerDevice'. The form contains several input fields with labels and values: '_deviceId:' with value '12345', '_sensorType:' with value 'Temperatura', '_location:' with value 'Madrid', '_client:' with value 'UCM', '_installationDate:' with value '18112024', '_protocolType:' with value 'Mqtt', '_brand:' with value 'Teltonika', and '_model:' with value 'string'. At the bottom, there are three buttons: 'CallData', 'Parameters', and 'transact'.

Imagen 25. Campos a completar para registrar un dispositivo.

Se deben respetar los criterios de carga de cada uno de los campos, por ejemplo para el campo **InstallationData** no se deben incluir caracteres especiales (., /, -) o cualquier otro tipo de caracter, solo acepta numeros consecutivos sin espacios, ejemplo 19112024 (19 de noviembre del 2024).

Función 2, posterior al registro del dispositivo tenemos dos funcionalidades para poder gestionarlo las cuales se explican a continuación.

Ingresando a la función **toggleDeviceStatus** podemos cambiar el estado de un dispositivo previamente registrado. simplemente basta con completar los siguientes campos.

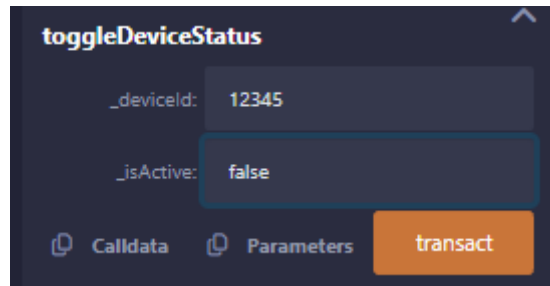


Imagen 26. Cambiar de estado un dispositivo.

Deviceld: ID con el cual se registró el dispositivo.

isActive:

 true = activo

 false = inactivo

El valor booleano se debe ingresar en letras minúsculas por el contrario dará error. Es importante tener en cuenta que cuando se registra el dispositivo toma automáticamente el estado true o activo.

Función 3, actualización de los datos asociados a un dispositivo mediante la función **updateDevicedata**, simplemente basta con ejecutar lo siguiente:

Ingresando a la función se deben completar los siguientes campos:

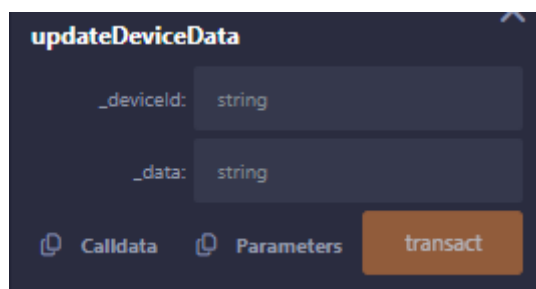


Imagen 27. Cargar datos asociados a un dispositivo.

Cargar el número de ID del dispositivo que queremos actualizar e ingresar los datos a asociar a nuestro dispositivo, ejemplo dispositivo con ID 12345 queremos documentar que se encuentra instalado.

Función 4 getdevice, esta función nos permite consultar todos los datos asociados a un dispositivo.

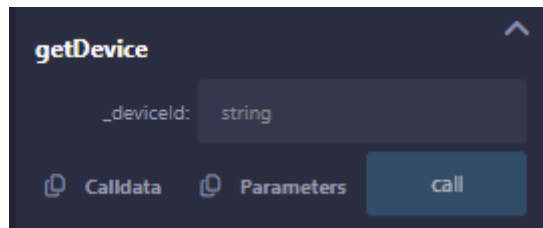


Imagen 28. Obtener los datos asociados al dispositivo.

Solo ingresando el número de ID obtenemos todos los datos de un dispositivo como se muestra a continuación

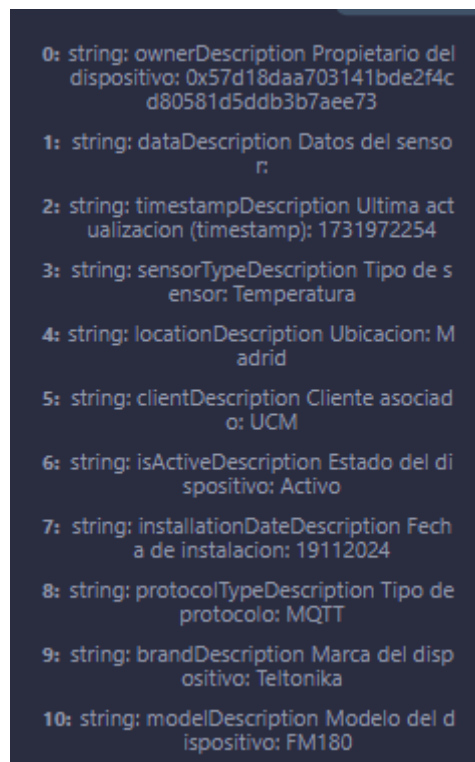


Imagen 29. Resultado de consultar los datos de un dispositivo.

Función 5 getdevicestatus, seleccionando la función e ingresando el número de ID del dispositivo que queremos consultar podemos ver si el dispositivo se encuentra activo o inactivo como se muestra a continuación.

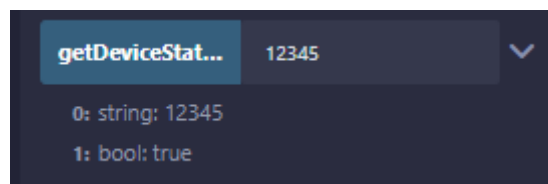


Imagen 30. Consultar el estado de un dispositivo.

Parte V

Referencias.

Wiki Iota. <https://wiki.iota.org/build/networks-endpoints/#shimmerevm-testnet>

IOTA Smart Contracts, 2021. https://files.iota.org/papers/ISC_WP_Nov_10_2021.pdf

Solidity. <https://docs.soliditylang.org/en/v0.8.28/>

RemixIDE. <https://remix.ethereum.org/#lang=en&optimize=false&runs=200&evmVersion=null&version=soljson-v0.8.28+commit.7893614a.js>

Documentación Remix. <https://remix-ide.readthedocs.io/es/latest/run.html>

Muñoz, Juan Manuel. (30/05/2018). IOTA – The Tangle, El blog de IoT en Español, IoT Futura. <https://iotfutura.com/arquitecturas-iot/iota-the-tangle>.

Parte VI

1. Repositorio público de Github.

La implementación se encuentra en el siguiente [repositorio público en GitHub](#).

2. Licencia utilizada

La licencia que hemos utilizado para la creación de este repositorio es la licencia MIT, pues es una de las más utilizadas en el ámbito del software libre y es muy poco restrictiva, lo que permite a los desarrolladores la libertad de reutilizar el código para sus propios proyectos.