Contenido

[La Binance Smart Chain 2](#_Toc98454316)

[Configurar una Wallet - Cartera 2](#_Toc98454317)

[Exportar la clave privada de la wallet 6](#_Toc98454318)

[Contrato inteligente 9](#_Toc98454319)

[Configuración del proyecto 9](#_Toc98454320)

[Función 1: Modifica datos en la blockchain 11](#_Toc98454321)

[Función 2: Obtener las direcciones de las personas que adoptaron 12](#_Toc98454322)

[Compilar y migrar 13](#_Toc98454323)

[Fichero descripto del contrato inteligente 18](#_Toc98454324)

[Aplicación web que usar Smart Contract BNB 19](#_Toc98454325)

# La Binance Smart Chain

Es una de las blockchains más populares, la forma de programación y despliegue de contratos inteligentes es casi idéntica a la de **Ethereum** (EHT), se basa en el mismo lenguaje de programación **Solidity**.

**Binance Chain** y **Binance Smart Chain** son dos blockchains diferentes. Binance Smart Chain es compatible con Binance Chain, pero además soporta el uso de **contratos inteligentes**.

Unas de las diferencias más significativas entre la **Binance Smart Chain y Ethereum** son que los bloques de la blockhain tienen un tiempo más pequeño, además que las tasas por escribir en la blockchain son significativamente más bajas. En lugar del token ETH se utiliza el token **BNB** para pagar las comisiones. Este token sigue los estándares de **BEP-20**.

Binance Smart Chain **soporta la EVM** por lo que podemos utilizar la mayor parte de las tecnologías más comunes para el despliegue de contratos inteligentes como Solidity, Truffle, etc.

# Contrato inteligente

Vamos a crear una pequeña aplicación de **venta de entradas para un concierto** para ello vamos a usar **un contrato inteligente**

Tanto los **datos** como **la lógica de la aplicación** estarán alojados en **la blockchain de pruebas de BSC.**

De esta forma las transacciones realizadas relativas **a las entradas** serán totalmente auditables.

El contrato va a tener 15 entradas, se almacenarán **en una lista/array** en la blockchain, en cada posición del array tendremos **la dirección de wallet que adquirió la entrada**.

Ejemplo

|  |  |
| --- | --- |
| tikets[0] | 0x000… |
| tikets[1] | 0x50d3b84a2b1787b7704DD50c67733b6b5B293629 |
| tikets[2] | 0x000… |

## Configuración del proyecto Smart Contract

Crearemos estructura de un proyecto usando **Truffle**.

|  |
| --- |
| **Truffle** es una herramienta que sirve para simplificar la creación de Smart Contracts, te crea la estructura de los proyectos. |

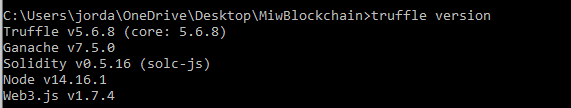
El primer paso es instalar truffle (necesitamos tener Instalado **Node.js** <https://nodejs.org/es/> )

npm install -g truffle

Puede tardar bastante en instalar, es posible que requiera permisos de administrador.

Para ver la versión de truffle que tenemos (y del resto de herramientas)

truffle version

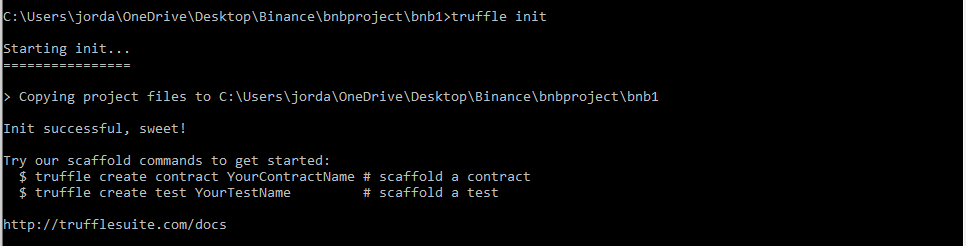


Creamos una nueva carpeta en Windows y la llamamos **bnb1**, esa va a ser la carpeta de nuestro proyecto, primero se crea la carpeta y luego se convierte en un proyecto.

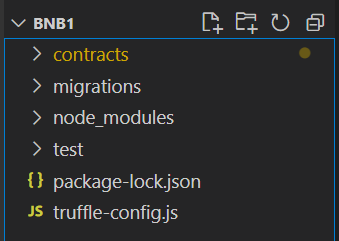


Para convertir la carpeta **/bnb1/** en un proyecto entramos en ella desde el cmd y ejecutamos el siguiente comando:

truffle init



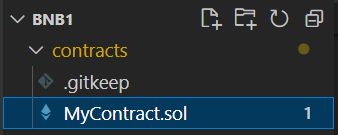
Abrimos el proyecto con un IDE que tenga ayuda de sintaxis para **Solidity** como por ejemplo **WebStorm** o **Visual Studio Code** (Con un plugin de Solidity) .



Carpetas que tiene dentro.

* **contracts/** contiene los Smart contracts implementados en lenguaje Solidity.
* **migrations/.** Esto lo usa truffle para probar los Smart contracts en desarrollo.
* **test:**/ directorio para meter pruebas de los contratos, en JavaScript y Solidty
* **truffle-config.js**: fichero de configuración de truffle, incluye información relevante para hacer el despliegue en la blockchain.

Dentro de la carpeta **/contracts** vamos a crear un fichero **MyContract.sol**



Comenzamos incluyendo la versión de Solidity y la declaración del contrato. Vamos a usar **solidity 0.5.\*** porque es la que venía establecida en truffle

|  |
| --- |
| pragma solidity ^0.8.0;  contract MyContract {  } |

Vamos a declarar un array de datos de tipo **address**, (cada addres permite incluir una cadena de 20 caracteres). Las address se usan para guardar direcciones de wallets o contratos inteligentes.

|  |
| --- |
| pragma solidity ^0.8.0;  contract MyContract {  address[16] public tikets; } |

|  |
| --- |
| **Visibilidad.**  Cuando las variables son **public** Solidiy genera automáticamente funciones para obtener esos valores “getters” (no genera setters). Si queremos evitar esto debemos utilizar **internal** o **private**.   * **internal**, pueden ser usadas por contratos derivados. * **private,** no pueden ser usadas por contratos derivados. |

Cada tipo de dato en solidity tiene un valor por defecto:

* boolean: false
* string: ""
* int: 0
* uint: 0
* fixed: 0.0 (presumably; this type is not fully supported)
* **address: 0x0000000000000000000000000000000000000000 (or address(0))**

## Función 1: Modifica datos en la blockchain

Se pueden declarar funciones en el contrato. Estas funciones podrán ser llamadas a través de la blockchain. **buyTiket(uint tiketIndex)** va a ser publica y retornar un **bool**.

Lo primero va a ser comprobar el valor del parámetro de **tiketIndex** con una condición **require(<lógica>),** si no está entre 0 y 15 se produce un error.

Una vez nos dan un valor valido de **tiketIndex**, vamos a comprobar que ese tiket esta sin comprar, en el array **tikets** la posición **tiketIndex** no tiene ninguna dirección almacenada. La dirección vacía es igual a **address(0),** si está libre guardamos en el array la dirección/wallet del usuario que llamo al contrato, se obtiene con **msg.sender.**

|  |
| --- |
| contract MyContract {   address[16] public tikets;   function buyTiket(uint tiketIndex) public returns (bool) {  // Comprobación  require(tiketIndex >= 0 && tiketIndex <= 15);   bool sucess = true;    if ( tikets[tiketIndex] == address(0) ){  // msg.sender address del usuario que invoco al contrato  tikets[tiketIndex] = msg.sender;  } else {  sucess = false ;  }   return sucess;  }   } |

|  |
| --- |
| **Require –** si alguna de las condiciones require no se cumple la función deja de ejecutarse y se produce una especie de “rollback”, el coste del gas si se puede perder, pero si el usuario hubiese por ejemplo enviado cryptomoneda esta no llegaría a asignarse al contrato, el usuario la recupera. |

Llamar a esta función **tiene un coste para los usuarios (gas)** ya que modifica la información de la blockchain. Este coste se paga en la moneda nativa de la blockchain.

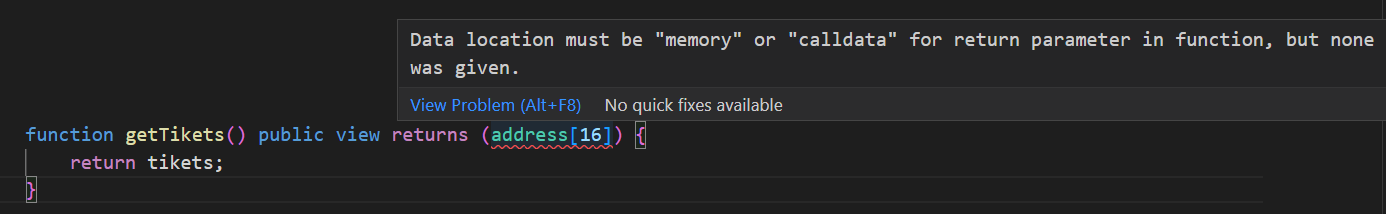
## Función 2: Obtener las direcciones de las personas que compraron tikets

Implementamos una función **getTikets()** que simplemente retorna el array con las 16 posiciones donde están las direcciones de los usuarios.

La palabra **view** significa que la función no modifica el estado del contrato (no cambia el valor de nada), estas funciones se pueden llamar sin gasto de **Gas** (tarifa por ejecutarlas).

|  |
| --- |
| function getTikets() public view returns (address[16]) {  return tikets;  }  } |

En el **retorno** vamos a tener que utilizar la palabra clave **memory**,



Cuanto retornamos, creamos localmente o recibimos como parámetro **un elemento de tipo no básico,** (por ejemplo, un array, struct o string) **solidity** nos obliga a especificar la localización del dato

Debemos ser muy cuidadosos con las copias de datos en Solidity es por eso por lo que se deben utilizar ciertas palabras clave, como:

* **memory:** el dato a generar se guarda solo en memoria (no en él contrato).   
  Cuando usamos “memory” en un retorno estamos copiando en memoria el objeto almacenado en el **storage** (tikets).

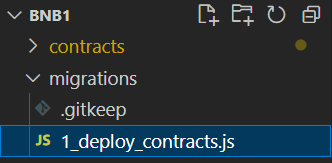
Las variables **globales** como **tikets**, se almacenan en el **store**, son siempre persistentes, se pueden usar en todas las funciones y son computacionalmente caras.

|  |
| --- |
| function getTikets() public view returns (address[16] memory) {  return tikets;  }  } |

## Compilar y migrar

Para desplegar el contrato debemos ir al directorio **/migrations .**

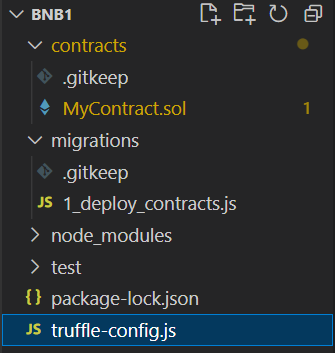
Creamos el fichero **1\_deploy\_contracts.js** . El número es importante ya que se usa para identificar el orden de ejecución.



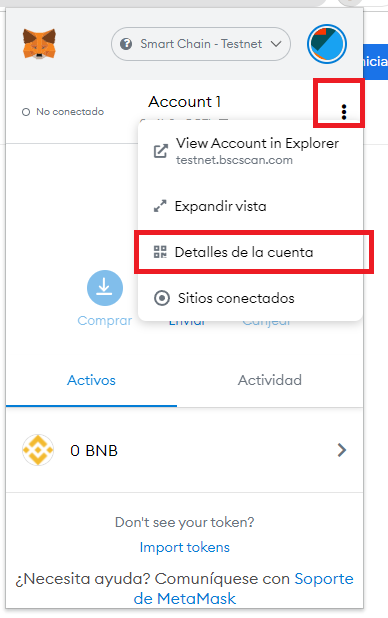
Incluimos el siguiente código en el fichero **1\_deploy\_contracts.js**. Es muy sencillo es solo importar con un requiere el contrato inteligente, en nuestro caso el contrato se llamada **MyContract**Es el nombre del contrato no del fichero, aunque suele coincidir no es obligatorio.

|  |
| --- |
| var MyContract = artifacts.require("MyContract");  module.exports = function(deployer) {  deployer.deploy(MyContract); }; |

Accedemos al fichero **truffle-config.js** este fichero tiene definida la blockchain sobre la que se va a desplegar el contrato inteligente.



Vamos a tener que mirar la **clave privada de nuestra wallet**.





**Clave privada:** 6cd9daea5ffff4b51875eb50dabb6b16585d787b261097015ecac0ce8d58f5cf

Con esta clave el despliegue tendrá acceso a nuestra wallet y obtendrá los BNB necesarios para pagar el gas que cuesta desplegar el contrato.

Configuramos el **truffle-config.js**. Para hacer el despliegue en la **BSC** hay que utilizar el módulo **@truffle/hdwallet-provider ,** necesita la clave privada de la wallet que utilizaremos para pagar las tasas y la **provideOrUrl** (url de la red TESTNET[**https://data-seed-prebsc-1-s1.binance.org:8545**](https://data-seed-prebsc-1-s1.binance.org:8545) **)**

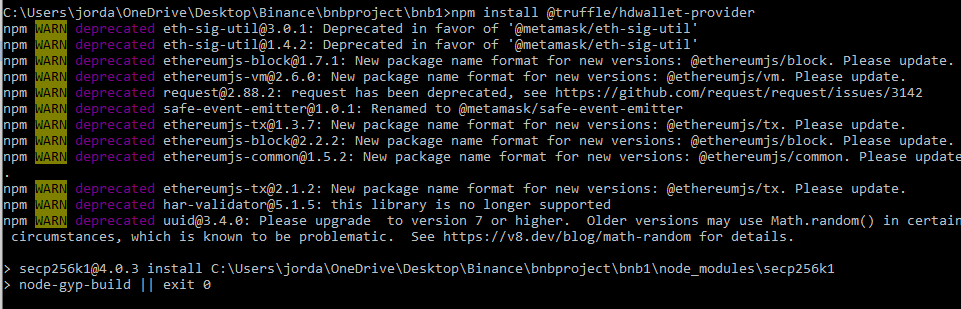
|  |
| --- |
| const HDWalletProvider = require('@truffle/hdwallet-provider');  const provider = new HDWalletProvider({     privateKeys: ['7b898623aabc049b8bb9587c3d55c119cc704ea76419471170bc9efc04f50ced'],     providerOrUrl: 'https://data-seed-prebsc-1-s1.binance.org:8545',  })  module.exports = {    networks: {      binanceTestnet: {        provider: () => provider,        network\_id: "97",        gas: 5000000      },      develop: {        port: 8545      }    },    compilers: {      solc: {        version: "0.8.17" // Fetch exact version from solc-bin (default: truffle's version)      }    }  }; |

\*La cantidad de **gas** que permitimos es muy importante, si ponemos un gas inferior al que se necesita para desplegar el contrato se producirá un fallo.

Tenemos que descargar el módulo **@truffle/hdwallet-provider** abrimos el **cmd** en el directorio del proyecto y ejecutamos

npm install @truffle/hdwallet-provider

Este paquete requiere tener instalado GIT y añadido al path del equipo.

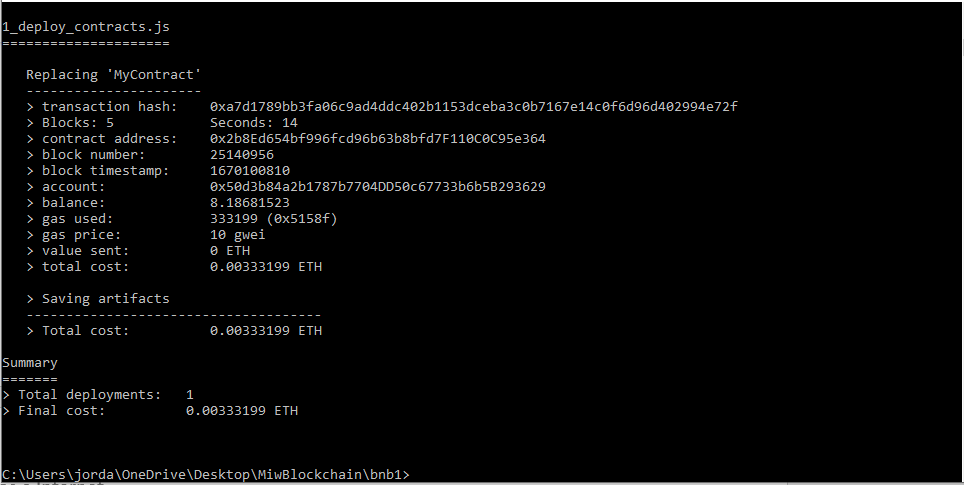


Puede tardar bastante tiempo en descargar la dependencia.

En el mismo **cmd** (directorio del proyecto) migramos y ejecutamos el proyecto

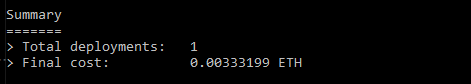
truffle migrate --reset --network binanceTestnet

El nombre “binanceTestnet” tiene que coincidir con el nombre del fichero **truffle-config.js**

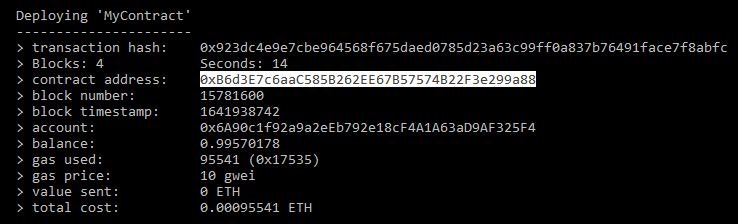


|  |
| --- |
| **Los costes en ETH (Ethereum)** realmente se refiere a **BNB**, trae **ETH** porque todas las herramientas son una migración directa de las Ethereum, adaptado a la blockchain de Binance Smart Chain, aquí la cryptomoneda “nativa” es el **BNB** |

La parte final de la consola indicará que los contratos se han **desplegado correctamente**.



Podemos ver la **address** del contrato dentro de la blockchain de Testnet

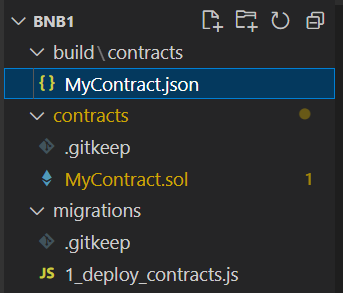


**0x6B9444df948903850c7eE605242f9f25E8747559**

Es posible que el despliegue tarde un tiempo.

# Fichero descripto del contrato inteligente

Una vez desplegado un contrato aparece un fichero **.json** en la carpeta **/contracts**



Este fichero contiene la información de conexión al contrato inteligente y las interfaces funciones.

El fichero **MyContract.json** será de gran utilidad cuando queramos usar el contrato desde una aplicación web. Las **liberias js** pueden hacer uso de estos ficheros, simplificando el uso del contrato inteligente, desde el punto de vista de la implementación se maneja de forma muy similar a una clase local.

Sí volvemos a desplegar los contratos asegurarse de borrar el antiguo contenido de **/build/contracts**, porque no siempre lo borra automáticamente…

# Aplicación web que usar Smart Contract BNB

**React** es una popular librería JavaScript para construir interfaces de usuario (es decir las vistas, no se trata de JavaScript ejecutado en el Servidor como Node.js) <https://es.reactjs.org/>

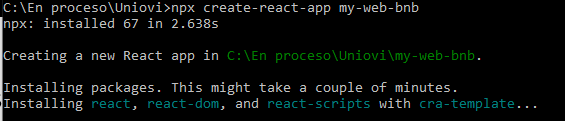
En este ejemplo vamos a crear una pequeña aplicación web con React, todo en el lado del cliente, no va a tener parte del servidor.

## Creación de la aplicación

Creamos un proyecto de **React** al que llamaremos **my-web-bnb**, el nombre del proyecto no admite mayúsculas. Ejecutamos en el cmd:

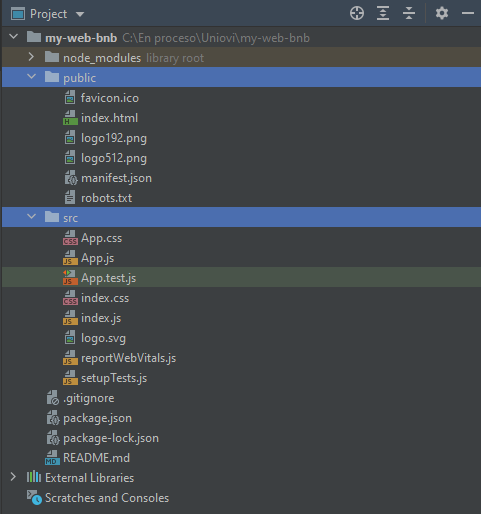
Lo hacemos fuera del proyecto Solidity, se trata de otro proyecto.

*npx create-react-app my-web-bnb*



Tardará en descargar las dependencias y generar en crear el proyecto, genera una estructura completa en la carpeta **my-web-bnb** , podemos abrir la carpeta con un IDE.

|  |
| --- |
| **Crear aplicación de React usando un asistente**  Por ejemplo, **Vite** <https://vitejs.dev/guide/> es otra alternativa al comando oficial, suele ser mucho más rápido  Abrimos un **cmd** y ejecutamos los comandos:  npm create vite@latest react-data --template react  Esto crea el Proyecto, a continuación, accedemos a la carpeta **react-data** y ejecutamos  npm install  npm run dev  La aplicación se arranca con el comando **npm run dev** (si se crea a través de Vite) |



Podemos “limpiar” un poco el directorio **/src/** y dejar solo **index.js**

****

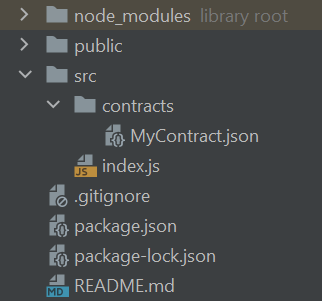
Vamos a simplificar el contenido de **index.js** incluyendo únicamente el siguiente código.

|  |
| --- |
| import ***React*** from 'react'; import ReactDOM from 'react-dom'; import 'bootstrap/dist/css/bootstrap.min.css';  const root = ReactDOM.createRoot(***document***.getElementById('root')); root.render(  <React.StrictMode>  <App />  </React.StrictMode> ); |

Estamos usando un componente **App** que aun no ha sido creado.

## Interfaces del Smart Contract

Creamos la carpeta /**src/contracts** y arrastramos dentro el fichero **MyContract.json** que teníamos en el build del proyecto Solidty, contiene la Interfaz para conectarse al contrato inteligente.



## Implementación de la aplicación

Vamos a instalar los paquetes necesarios para implementar la web (incluidos algunos que utilizaremos más adelante), abrimos un cmd dentro del directorio de la web.

npm install @metamask/detect-provider@1.2.0

Es la librería para poder acceder a la wallet desde nuestra web.



npm install[ethers@5.0.32](mailto:ethers@5.0.32)

Ether es la librería que nos permite conectarnos al contrato inteligente (hay otras alternativas)



Documentación dispone en: <https://docs.ethers.io/v5/getting-started/>

npm install bootstrap@4.6.0

Bootstrap para la interfaz web



Lo primero que vamos a hacer es crear el componente **App**, utilizando una función

|  |
| --- |
| import ***React*** from 'react'; import ReactDOM from 'react-dom'; import 'bootstrap/dist/css/bootstrap.min.css';  function App(){  return (  <div>  <h1>Hello World</h1>  </div>  ) }  const root = ReactDOM.createRoot(***document***.getElementById('root')); root.render(  <React.StrictMode>  <App />  </React.StrictMode> ); |

Dentro del componente creamos el método **getBlockchain** que tiene dos misiones:

1. Acceder a la wallet Metamask instalada en el navegador (el provider) y sacar su firma.
2. Obtener el objeto que da acceso al contrato inteligente, para ello necesita los datos de conexión (especificados en el fichero /contracts/MyContract.json) y la firma de la cartera del usuario.

**Obtener la firma de la wallet**

El primer paso es detectar que el usuario tiene **wallet** instalada en el navegador, para ello utiliza **detectEthereumProvider()**. Se trata de un método asíncrono, estamos usando promesas de JavaScript.

Después va a necesitar solicitar que el usuario apruebe utilizar su cuenta en la aplicación **provider.request({ method: 'eth\_requestAccounts' });** La wallet MetaMask puede trabajar con diferentes redes, no solo BSC, por eso se le solicita al usuario la ID de la red en la que tiene configurada la cartera **provider.request({ method: 'net\_version' }).** Finalmente se obtiene el **provider** utilizando la librería **ether**.

El siguiente paso es obtener la firma del **provider**.

|  |
| --- |
| import ***React*** from 'react'; import ReactDOM from 'react-dom'; import 'bootstrap/dist/css/bootstrap.min.css'; import detectEthereumProvider from "@metamask/detect-provider"; import {Contract, ethers} from "ethers";  function App(){   let getBlockchain = async () => {  let provider = await detectEthereumProvider();  if(provider) {  await provider.request({ method: 'eth\_requestAccounts' });  const networkId = await provider.request({ method: 'net\_version' })   provider = new ethers.providers.Web3Provider(provider);  const signer = provider.getSigner();   }  return null;  } |

**Obtener el objeto que da acceso al contrato inteligente**

Con los datos almacenados en el manifiesto (./contracts/MyContract.json): dirección del contrato y abi , podemos obtener el objeto que da acceso al contrato. También necesitamos la **firma de la cartera**, es el usuario que realmente está ejecutando el contrato.

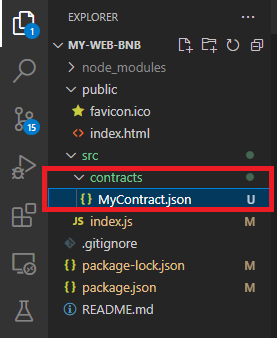
**\*abi** es un objeto json con las interfaces de los métodos públicos del contrato.

El fichero **MyContract.json** debe ser importado.

De forma temporal vamos a cargar el contrato en la variable **tempContract**.

|  |
| --- |
| import detectEthereumProvider from "@metamask/detect-provider"; import {Contract, ethers} from "ethers"; import myContractManifest from "./contracts/MyContract.json";  function App(){   let getBlockchain = async () => {  let provider = await detectEthereumProvider();  if(provider) {  await provider.request({ method: 'eth\_requestAccounts' });  const networkId = await provider.request({ method: 'net\_version' })   provider = new ethers.providers.Web3Provider(provider);  const signer = provider.getSigner();   const tempContract = new Contract(  myContractManifest.networks[networkId].address,  myContractManifest.abi,  signer  );   }  return null;  } |

Importante, debemos asegurarnos de que realmente tenemos en el proyecto, el descriptor JSON que define la conexión e interfaz del contrato inteligente



**Hooks, estado y useEffects (React)**

Necesitamos que nuestro componente tenga lo siguiente:

1. **Interfaz del contrato**. Una variable accesible en todo el componente que me permita usar las funciones del contrato inteligente
2. **Lista de tikets**. Una variable accesible en todo el componente con la lista de tikets que además se va a renderizar en la parte gráfica.

Los **Hooks de react** son funciones que nos permiten añadir comportamientos a nuestro componente.

Vamos a tener que utilizar **3 Hooks** (son los más comunes)

El primer paso es importarlos.

|  |
| --- |
| import {Contract, ethers} from "ethers"; import myContractManifest from "./contracts/MyContract.json"; import { useState, useEffect, useRef } from 'react'; |

**useRef – referencia,** se usa para crear variables globales. Se crean utilizando **useRef(defaultValue).** La variable es accesible/editable en todo el componente mediante **<nombreVariable>.current** . Creamos una variable **myContract** para guardar el objeto de acceso al contrato. También asignamos el contrato creado a esa variable.

|  |
| --- |
| function App(){  const myContract = useRef(null);   let getBlockchain = async () => {  let provider = await detectEthereumProvider();  if(provider) {  await provider.request({ method: 'eth\_requestAccounts' });  const networkId = await provider.request({ method: 'net\_version' })   provider = new ethers.providers.Web3Provider(provider);  const signer = provider.getSigner();   myContract.current = new Contract(  myContractManifest.networks[networkId].address,  myContractManifest.abi,  signer  );   }  return null;  } |

**useState – estado,** sí estamos renderizando una **variable de estado** y el valor de esta cambia la vista se actualiza de forma automática. Las variables de estado se declaran con **useState(defaultValue)** , suelen definir el nombre de la variable y método set.

El cambio de valor del estado se hace siempre desde un método, por ejemplo **setTikets(),** nunca modificamos la variable directamente.

|  |
| --- |
| function App(){  const myContract = useRef(null);  const [tikets, setTikets] = useState([]); |

**useEffect,** se trata de una función muy popular en React se ejecuta cuando se carga el componente y cuando se realiza alguna actualización en el estado o propiedades, si añadimos un segundo parámetro **[ ]** al **useEffect()** pasa a ejecutarse únicamente al inicio del componente.

En este caso va a ser una especie de “inicializador”.

|  |
| --- |
| function App(){  const myContract = useRef(null);  const [tikets, setTikets] = useState([]);   useEffect( () => {    }, [] ) |

**Cargar la información inicial**

Vamos a incluir un método **initContracts.** encargado de :

1. Inicializar el contrato llamando a **getBlockchain()** (previamente implementado)
2. Llamar al método **getTikets()** del contrato usando myContract.current (variable creada con useRef)
3. Finalmente, los **tikets** en la variable de estado **(usando el método setTikets)**

|  |
| --- |
| function App(){  const myContract = useRef(null);  const [tikets, setTikets] = useState([]);   useEffect( () => {  initContracts();  }, [])   let initContracts = async () => {  await getBlockchain();  let tiketsFromBlockchain = await myContract.current.getTikets();  setTikets(tiketsFromBlockchain)  } |

**Renderizado**

Los componentes función deben retornar el código de vista, vamos a usar JSX, una especie de lenguaje HTML que permite incluir código JS utilizando { } hay que tener en cuenta unas consideraciones

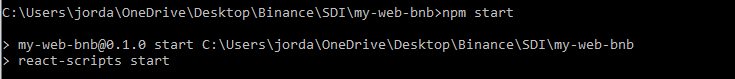
* Si el JSX ocupa más de una línea hay que utilizar paréntesis
* Es obligatorio que tenga un elemento padre por ejemplo un <div>
* Algunos elementos de HTML/CSS estándar han sido modificados como **class**, se usa **className ,** nunca se admiten etiquetas XML que no cierren, usar <img/> en lugar de <img>

El return de la función App va a recorrer, la lista **tikets** utilizando un map e imprimirá la dirección de la wallep que ha comprado cada entrada.

|  |
| --- |
| function App(){  const myContract = useRef(null);  const [tikets, setTikets] = useState([]);   useEffect( () => {  initContracts();  }, [])   let initContracts = async () => {  await getBlockchain();  let tiketsFromBlockchain = await myContract.current.getTikets();  setTikets(tiketsFromBlockchain)  }   let getBlockchain = async () => {  let provider = await detectEthereumProvider();  if(provider) {  await provider.request({ method: 'eth\_requestAccounts' });  const networkId = await provider.request({ method: 'net\_version' })   provider = new ethers.providers.Web3Provider(provider);  const signer = provider.getSigner();   myContract.current = new Contract(  myContractManifest.networks[networkId].address,  myContractManifest.abi,  signer  );   }  return null;  }  return (  <div>  <h1>Tikets store</h1>  <ul>  { tikets.map( (address, i) =>  <li>Tiket { i } comprado por { address } </li>  )}  </ul>  </div>  ) } |

Llegados a este punto podríamos ejecutar la aplicación, aunque todavía no está completa abriendo un **cmd** en la ruta del proyecto y ejecutando el comando

npm start

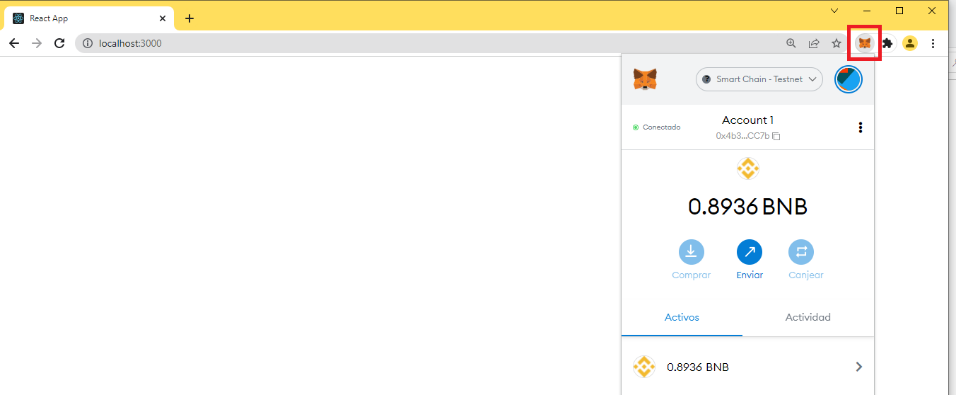


Abrimos el Chrome y desbloqueamos la wallet de Metamask

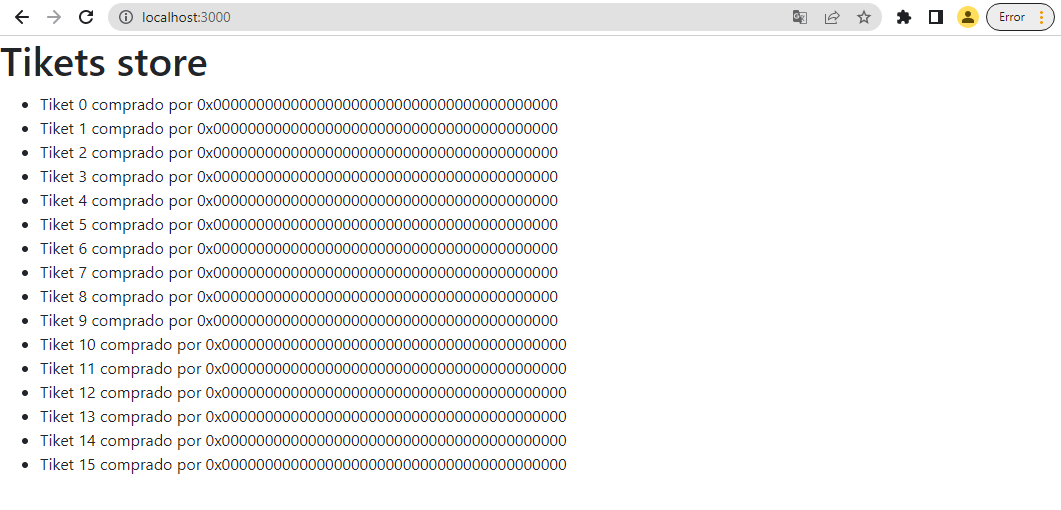
Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Accedemos a la web, por defecto es el puerto del start de react es <http://localhost:3000/> sí todo funciona bien no aparece el mensaje de **Loading…** en la web, solo aparece si no carga MyContract o Data.



Una vez abrimos la cartera aparecerá lo siguiente, una lista con todos los tikets.



Comprar  
  
Vamos a permitir que se puedan comprar tikets desde la web. Accedemos el **return** del componente **App.**

Comprobamos sí el valor de:

**address == “0x0000000000000000000000000000000000000000"** esto significa que el tiket no ha sido comprado, incluimos un enlace que ejecuta el método **clickBuyTiket()** (aun sin implementar).

No hace falta escribir “manualmente” la address vacía, hay una variable en la librería **ethers** ( **ethers.constas.AddressZero** ) que nos permite especificar la address vacía.

El operador && funciona como una condición simple

*{ <condición> && código a insertar si se cumple la condición }*

|  |
| --- |
| return (  <div>  <h1>Tikets store</h1>  <ul>  { tikets.map( (address, i) =>  <li>Tiket { i } comprado por { address }  { address == ethers.constants.AddressZero &&   <a href="#" onClick={()=>clickBuyTiket(i)}> buy</a> }  </li>  )}  </ul>  </div>   ) } |

|  |
| --- |
| **Librería ethers**  Además de las llamadas a los contratos inteligentes (Contract Interaction) esta librería permite manipular de forma completa las carteras (Providers) y contiene una gran cantidad de utilidades (Utilities).  <https://docs.ethers.io/v5/api/utils/> |

Finalmente, implementados **buyTiket** **()**. Está función se ejecuta al pulsar el botón debe ejecutar **myContract.current.buyTiket(tiketIndex),** como la función es asíncrona se utiliza **await.**

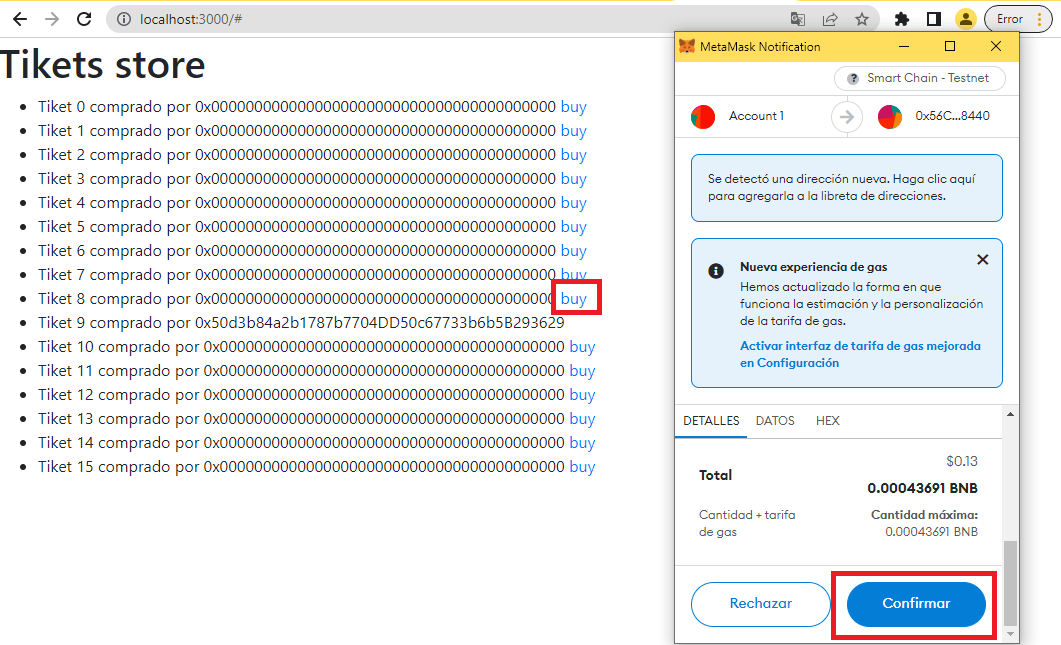
IMPORTANTE: La función **buyTiket(tiketIndex)** modifica la blockchain, por lo que tiene coste (gas), cuando se llama a las funciones con coste la wallet del usuario se abre y debe aprobar expresamente el gasto. Para esperar esta aprobación incluimos la llamada **wait();**

Una vez se produce la compra deberíamos actualizar la **tikets** para ello volvemos a llamar **myContract.current.getTikets(),** si cambiamos el valor a la variable Tiket (estado), el componente se vuelve a renderizar.

|  |
| --- |
| let clickBuyTiket = async (i) => {  const tx = await myContract.current.buyTiket(i);  await tx.wait();   const tiketsUpdated = await myContract.current.getTikets();  setTikets(tiketsUpdated);  }   return (  <div>  <h1>Tikets store</h1>  <ul>  { tikets.map( (address, i) =>  <li>Tiket { i } comprado por { address }  { address == ethers.constants.AddressZero &&  <a href="#" onClick={()=>clickBuyTiket(i)}>buy</a> }  </li>  )}  </ul>  </div>  ) } |

Accedemos a la web <http://localhost:3000/> y probamos a comprar.

Para hacer una compra habrá que **confirmar** la transacción en la wallet.



La información tarda en actualizarse en la blockchain ya que requiere varias confirmaciones, aparece una notificación en la parte baja de la pantalla.

