

Práctica 3. Smart contracts en Ethereum.

Introducción

José Incera. Marzo 2018

1. Objetivo

Familiarizarse con los conceptos básicos de *smart contracts* en Ethereum en un ambiente de evaluación.

2. Introducción

Ethereum y su criptodivisa, *Ether*, es quizás el segundo blockchain más popular después de Bitcoin. Sin embargo, la gran contribución de Ethereum no está en una nueva criptodivisa sino en la capacidad de su blockchain de soportar **smart contracts**. Se trata de segmentos de código (como clases y funciones) que permiten definir las *reglas de negocio*, es decir, las condiciones en que se puede llevar a cabo una transacción.

Con los smart contracts se pueden desplegar **DApps**, aplicaciones descentralizadas, las cuales se espera que generarán un verdadero efecto transformador de blockchain en la sociedad.

En esta práctica se trabajará con un ambiente de desarrollo auto-contenido (un *sandbox*) implementado, entre otros, para validar smart contracts. Consiste de tres partes:

- En la primera simplemente se instala el ambiente y se intercambian tokens (criptodivisas)
- En la segunda aprenderemos cómo crear un smart contract que permita validar la existencia y la integridad de documentos
- En la tercera parte, basada en un tutorial de Ethereum, veremos cómo se despliega una interfaz de usuario (front-end) para ocultar los detalles del blockchain.

La práctica está inspirada en las siguientes referencias:

- [A gentle introduction to Ethereum programming](#)
- [The hitchhiker's guide to smart contracts in Ethereum](#)
- [Ethereum Pet Shop Tutorial](#)

3. Herramientas

Para desarrollar DApps, se necesita un cliente para conectarse al blockchain. Existen muchas alternativas compatibles con Ethereum. La más popular para hacer pruebas de concepto, es `testrpc`, recientemente renombrada `ganache-cli`. Testrpc también simula una cadena de bloques.

También se necesita un ambiente para compilar, desplegar y probar rápidamente smart contracts (es decir, un *IDE*). La herramienta más popular para ello es [Truffle](#).

En el apéndice se muestra cómo instalar testrpc y Truffle para ambients Windows 7 y Windows 10. La instalación en Ubuntu es trivial. Si no lo ha hecho ya, vaya al apéndice e instale las herramientas. **En el laboratorio estas herramientas ya se encuentran instaladas.**

4. Transferencias entre cuentas

4.1.- Ejecutar `testrpc`

Lo primero que necesitamos, es una cadena de bloques. Ésta es proporcionada por testrpc. Lance el comando **en una ventana PowerShell**; la ventana quedará ocupada por testrpc. Al ejecutarlo, testrpc genera diez cuentas con fondos ficticios para poder hacer pruebas. También queda esperando solicitudes **en el puerto 8545 por omisión** para agregar transacciones.

Nota: En Linux, el comando es `ganache-cli`

```

EthereumJS TestRPC v6.0.3 <ganache-core: 2.0.2>
Available Accounts
=====
<0> 0x3eed57ec42e4cac042feb27d4e2cdbf4490bae
<1> 0xa4a2b418cdfa62277f70f8f89893f6b497e0d878
<2> 0xc7142817dd983697fa976860cbaef177317b81d9
<3> 0xc9da1b81bfbf112909ac06b1a669acd1207e4dbf
<4> 0x370fe2087c4f2e8ccc5d97205faa2b0f49557c4c
<5> 0x251df6786d175465ac362aaac5edfffe9fb4e044f
<6> 0xd352849bd0d627e18a8d656d9dda0e5e50825e42
<7> 0xe78becdic0b99d0933b0d73917442fb7bdcd9aca
<8> 0x6985ee29b951a773dbe274ea7467f420dba9fb31
<9> 0xfb098ed1fbfffa0bf65312e95f8ddcd613d367aac

Private Keys
=====
<0> 5aba587f066ef4dc8f9c8beef00c8a7822d87b1a6ad3f93713f6e73f2c13c20e
<1> 1438d81e11c9845f2f8f0404491c15fce6fb099f865eed2bad0cec780f7c6ea3
<2> fa64226c1c99d73d2af022f56798c46a01943d98f72cc3684400565cf4a0be45
<3> d5f531583439ab56889c66eef1c586bb3381df301e1b1e884a12b1ecda9485ae
<4> 38547e8873ba5809079a5hdd62ba65f63d0999a9b6081d67a12ef384e82c65fe
<5> fd6fea3872351087561589f53be2a5453fcc6d3d131af744e8b77676649b393
<6> 170533a5dea3dad03a1912298192938320c2f694ecd0172df0aeb7de6b0e3d92
<7> 3f86de454b34f1af0ea281aaaf3411d615f306945ced9e32ef1f194020ca28f
<8> 2d593cf10dc797dc6a9157b1b8f4656c6fe0c303f5e070f6a648696d8faf9ba
<9> bcae5e5f4926eb656f2824457514e2148bc817a1f644cf46c0ee98f07205ec24

HD Wallet
=====
Mnemonic: wear hire imitate boat staff gloom gaze supreme name brief gorilla attend
Base HD Path: m/44'/60'/0'/0/{account_index}

Listening on localhost:8545
eth_getBlockByNumber

```

Fig. 2. Lanzamiento de testrpc

4.2.- Interacción entre cuentas

En este primer ejercicio, vamos a realizar transferencias entre algunas de las cuentas creadas por testrpc. Esta interacción se suele hacer con `web3`, una librería JavaScript que implementa JSON RPC para Ethereum. Para no entrar a ese nivel de detalle, utilizaremos algunas funcionalidades de `Truffle`, que usan `web3` internamente.

4.2.1.- Inicialización y configuración de Truffle

En una ventana de PowerShell **distinta a aquélla en la que se está ejecutando testrpc**, cree un directorio `pruebasETH`, colóquese ahí y lance la inicialización de Truffle:

```

PS> mkdir pruebasETH

Directorio: C:\CursoEthereum\

Mode          LastWriteTime    Length Name
----          -----        ----- 
d---  24/02/2018  07:12 p.m.      pruebasETH

PS> cd pruebasETH
PS> truffle init

```

Downloading...

Unpacking...

...

Se instalaron dos archivos y tres carpetas:

- contracts/ : Contiene los archivos fuente de los smart contracts en lenguaje Solidity .
- migrations/ : Truffle utiliza un sistema de migración para gestionar el despliegue de smart contracts. Migration es un smart contract especial que monitorea cambios.
- test/ : Aquí se almacenan las pruebas de nuestros contratos en JavaScript y Solidity.
- truffle.js : Es el archivo de configuración de Truffle.

El archivo de configuración se debe modificar para indicar en qué red se van a hacer las pruebas (se permiten distintas redes, como la local, una global llamada testnet y, desde luego, la de Ethereum).

Con un editor de textos abra el archivo truffle.js y sustituya su contenido por el siguiente (*suponemos que testrpc se ejecutó en el puerto por omisión, el 8545*):

```
module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",
      port: 8545,
      network_id: "*" // Match any network id
    }
  }
};
```

4.2.2.- Interacción desde la consola de Truffle

Lance truffle console en el directorio que tiene el archivo truffle.js .

```
PS> truffle console
truffle(development)>
```

Para irnos familiarizando con algunos aspectos del ambiente, consultemos el saldo de una cuenta:

```
truffle(development)> web3.eth.getBalance(web3.eth.accounts[0])
BigNumber { s: 1, e: 20, c: [ 1000000 ] }
```

El resultado es un poco extraño. En primer lugar, web3 utiliza objetos tipo BigNumber para las variables numéricas. Estos objetos se representan por un signo (1 = positivo), un exponente (e) y un coeficiente (c). En segundo, el valor no está representado en Ether sino en **wei**, el valor más pequeño en Ethereum, que equivale a 1/10E18 Ether.

Con el siguiente comando se pueden obtener los balances de las cuentas creadas por testrpc:

```
truffle(development)> web3.eth.accounts.forEach(cuenta => {balance = web3.eth.getBalance(cuenta); console.log(balance); })
BigNumber { s: 1, e: 20, c: [ 1000000 ] }
BigNumber { s: 1, e: 20, c: [ 1000000 ] }
BigNumber { s: 1, e: 20, c: [ 1000000 ] }
BigNumber { s: 1, e: 20, c: [ 1000000 ] }
BigNumber { s: 1, e: 20, c: [ 1000000 ] }
BigNumber { s: 1, e: 20, c: [ 1000000 ] }
BigNumber { s: 1, e: 20, c: [ 1000000 ] }
BigNumber { s: 1, e: 20, c: [ 1000000 ] }
BigNumber { s: 1, e: 20, c: [ 1000000 ] }

truffle(development)>
```

Para enviar transferencias de una cuenta a otra, usaremos la función `sendTransaction`, que es parte de la API de web3.js:

```
truffle(development)> De = web3.eth.accounts[0]
'0xd08d700d5009bb27b5e945cfa3ca034eb470fab1'
truffle(development)> Para = web3.eth.accounts[1]
'0x099642d1c7d34cf185b697d0afa6bb1c36b78b32'

truffle(development)> operacion = { from: De, to: Para, value: 5000000000000000000 }

{
  from: '0xd08d700d5009bb27b5e945cfa3ca034eb470fab1',
  to: '0x099642d1c7d34cf185b697d0afa6bb1c36b78b32',
  value: 5000000000000000000
}
truffle(development)> operHash = web3.eth.sendTransaction(operacion)
```

```
'0xd78f3e0026a223809aeffbc264e3866c3a00221abed354da5676dc97b7c5050e'
```

Se envían 5E14 wei de la cuenta 0 a la 1. El último valor devuelto es el hash de la transacción. Lo podemos usar para obtener información sobre ésta:

```
truffle(development)> web3.eth.getTransaction(operHash)
{ hash: '0xd78f3e0026a223809aeffbc264e3866c3a00221abed354da5676dc97b7c5050e',
  nonce: 0,
  blockHash: '0x955c04da97f5de3cf4288cdad8253a099e1c0ba83c32e8ff528c4a49f9a48fcf',
  blockNumber: 6,
  transactionIndex: 0,
  from: '0xd08d700d5009bb27b5e945cfa3ca034eb470fab1',
  to: '0x099642d1c7d34cf185b697d0afa6bb1c36b78b32',
  value: BigNumber { s: 1, e: 14, c: [ 5 ] },
  gas: 90000,
  gasPrice: BigNumber { s: 1, e: 0, c: [ 1 ] },
  input: '0x0'
truffle(development)>
```

Se pueden gastar hasta 90,000 unidades de *gas* para esta transacción y el precio de la unidad es de 1 wei.

Verifiquemos que la transacción efectivamente tuvo lugar:

```
truffle(development)> web3.eth.accounts.forEach(cuenta => { balance = web3.eth.getBalance(cuenta) ; console.log(balance); })
BigNumber { s: 1, e: 19, c: [ 999994, 99999999979000 ] }
BigNumber { s: 1, e: 20, c: [ 1000005 ] }
BigNumber { s: 1, e: 20, c: [ 1000000 ] }
BigNumber { s: 1, e: 20, c: [ 1000000 ] }
...
...
```

Vemos que la cuenta 1 efectivamente recibió 5 Ether, pero la cuenta 0 se decrementó un poco más que eso. La diferencia es precisamente el `gas` gastado en la transacción: 21,000 unidades, como puede comprobarlo en la ventana en la que se está ejecutando `testrpc`.

Para terminar esta sección, siéntase en libertad de practicar haciendo transferencias entre otras cuentas.

¿Qué pasa si el monto a transferir es mayor al saldo de una cuenta? ¿Si el origen y el

destinatario son la misma cuenta?

=====

AL TERMINAR, ABANDONE LA CONSOLA DE TRUFFLE CON EL COMANDO .exit

```
truffle(development)> .exit  
PS>
```

5. Smart contracts

Vamos a escribir una aplicación **notario digital** que almacena el hash de un documento como prueba de su existencia y de su integridad.

5.1.- Smart contract. Primera versión.

Nuestros ejemplos están lejos de tener la calidad de un smart contract en producción. De hecho, este primer ejemplo ni siquiera será útil para almacenar más de un documento.

Debemos crear un nuevo proyecto que se llamará ValidaDocs1:

```
# Si aún está dentro de la consola de Truffle, sálgase  
truffle(development)> .exit ; observe el punto al inicio  
  
PS>truffle create contract ValidaDocs1  
  
PS> dir contracts  
  
Directorio: D:\Cursos\TallerBlockChain\Ethereum\PruebasSolidity\contracts  
  
Mode LastWriteTime Length Name  
---- ----- ----  
-a--- 20/02/2018 11:58 515 Migrations.sol  
-a--- 20/02/2018 20:48 100 ValidaDocs1.sol
```

Ahora con su editor de textos favorito, por ejemplo sublime , abra el archivo ValidaDocs1.sol e inserte este código:

```
pragma solidity ^0.4.4;
```

```

// Contrato ValidaDocs, versión 1
contract ValidaDocs1 {
    // estado
    bytes32 public hashDoc; // El hash del documento
    uint256 accComision; // Eventual acumulador de comisiones
    address public creador; // Quién invocó el contrato

    // Constructor
    function ValidaDocs1() {
        creador = msg.sender;
        accComision = 0;
    }

    // Calcula y almacena el hash de un doc
    // "función transaccional"
    function registra(string documento) {
        hashDoc = hashPara(documento);
    }

    // función auxiliar. Calcula sha256
    // solo lectura
    function hashPara(string documento) constant returns (bytes32) {
        return sha256(documento);
    }
}

```

Algunos puntos sobre el código:

- El `pragma` le indica al compilador que se requiere de una versión mayor o igual (^) a la señalada (0.4.4).
- `creador` almacenará la dirección de quien invocó el contrato. Como cada llamada a función es una transacción, el invocante es quien envió la transacción, que es `msg.sender`, como se observa en el constructor.
- `hashDoc` almacenará el hash del documento. Como se calcula con sha256, esta variable es tipo `byte32`, es decir 32 bytes. **Es una variable public y puede ser leída por cualquiera para verificar la existencia e integridad de un documento.**
- `accComisión` es un entero sin signo de 256 bits. Sería un acumulador donde se almacenan las comisiones cargadas por generar el hash. En este tutorial se utiliza únicamente como un contador de las veces que se han invocado estas operaciones.
- `Constructor` como en los lenguajes orientados a objetos, en Solidity el constructor tiene el mismo tipo que el smart contract (una clase) y se invoca cada vez que una nueva instancia del contrato se despliega en el blockchain. Además de registrar el dueño del contrato, en el

constructor se inicia el acumulador de comisiones a cero.

- `registra` es la función donde se guardará el hash del documento en `hashDoc` . Esta función es “transaccional” o no-constante pues cambia el estado de la red (genera una transacción) y su ejecución gasta `gas` .
- `hashPara` es la función que invoca el cálculo del hash y lo devuelve. Esta función no cambia el estado de la cadena por lo que se considera de sólo lectura. Como *cualquier nodo* puede calcular el hash, esta función no gasta `gas` .

5.2.- Despliegue del contrato

Para desplegar el smart contract `ValidaDocs1`, en esta ocasión será necesario editar (o crear) el archivo de migración `migrations/2_deploy_contracts.js` para que Truffle despliegue el nuevo contrato con el siguiente contenido:

```
var ValidaDocs1 = artifacts.require("./ValidaDocs1.sol");
module.exports = function(deployer) {
  deployer.deploy(ValidaDocs1);
};
```

Vamos a enviar los smart contracts a la red. Esto se llama una migración; como vamos a hacer una nueva migración del ambiente que hemos estado utilizando (el creado con `truffle init`) debe utilizarse la bandera `reset` :

```
PS> truffle migrate --reset
Using network 'development'.
```

```
Running migration: 1_initial_migration.js
Deploying Migrations...
```

```
...
```

```
Running migration: 2_deploy_contracts.js
Deploying ValidaDocs1...
...
```

5.3.- Interactuando con el smart contract

Con el contrato desplegado, podemos interactuar con él a través de la consola de Truffle como lo hicimos anteriormente.

Empecemos por conocer la dirección del smart contract:

```

PS> truffle console
PS> truffle(development)> var vd = ValidaDocs1.at(ValidaDocs1.address)
undefined
PS> truffle(development)> vd.address
'0x57406d4a49469956b0492ce236bddc5afe7c703a'

```

Ahora registremos un “documento” y verifiquemos que efectivamente su prueba se almacena y es correcta:

```

truffle(development)> vd.registra('Un tutorial muy interesante')
{ tx: '0x2a9c98bb747be867efeee9632638540e29b634e48f9f1a7aee8bdbe1ac36d34b',
  receipt:
    { transactionHash: '0x2a9c98bb747be867efeee9632638540e29b634e48f9f1a7aee8bdbe1ac36d34b',
      transactionIndex: 0,
      blockHash: '0xea89c5fb51a10b8cef826fe2bdab2a5f555367c877987159fa00ef0fe4902fa3',
      ,
      blockNumber: 9,
      gasUsed: 45015,
      cumulativeGasUsed: 45015,
      contractAddress: null,
      logs: [],
      status: 1 },
    logs: [] }

// La prueba del documento es:
truffle(development)> vd.hashPara('Un tutorial muy interesante')
'0x2dccd5bd331f2e122f33960b4d47c69b602ec0c2448a906c49592488733bfaee'

// ... y lo que se almacenó en la memoria del contrato:
truffle(development)> vd.hashDoc()
'0x2dccd5bd331f2e122f33960b4d47c69b602ec0c2448a906c49592488733bfaee'

```

Como se esperaba, el hash almacenado y el calculado son iguales. Ese “documento” existe y no ha sido modificado. *Observe que para acceder al valor de una variable Public, basta invocar una función con el mismo nombre. Estas funciones “getter” se crean automáticamente.*

5.4.- Mejorando el contrato

Como pudo observar el código anterior solo permite almacenar el hash de un documento a la vez. Podemos usar una estructura *map* para asociar hashes que se van almacenando, a cierto o falso.

La nueva variable tipo `map` se llamará `hashesDocs` y será privada. Para checar si un hash está o no registrado (es decir, si existe un documento y no ha sido alterado), tendremos que agregar una nueva función `chechaDocumento()`. Aprovecharemos para agregar una línea que incremente en 1 el acumulador de comisiones cada vez que se registra un hash.

El código final es el siguiente. Edite `ValidaDocs1.sol`, haga las modificaciones que corresponda y guárdelo como `ValidaDocs2.sol`.

```
pragma solidity ^0.4.4;

// Contrato ValidaDocs, version 2
contract ValidaDocs2 {
    // estado
    mapping (bytes32 =>bool) private hashesDocs; // hashes -> true/false

    uint256 public accComision; // Eventual acumulador de comisiones
    address public creador; // Quien invocó el contrato

    // Constructor
    function ValidaDocs2() {
        creador = msg.sender;
        accComision = 0;
    }

    // Calcula y almacena el hash de un doc
    // "función transaccional"
    function registra(string documento) {
        var hash = hashPara(documento);
        guardaHash(hash);

        accComision += 1;
    }

    // Guarda el hash de un documento en el mapa y asigna true
    function guardaHash(bytes32 h) {
        hashesDocs[h] = true;
    }

    // función auxiliar. Calcula sha256
    // "solo lectura"
    function hashPara(string documento) constant returns (bytes32) {
```

```

        return sha256(documento);
    }

// Checa si un documento ha sido registrado
function checaDocumento(string doc) constant returns (bool) {
    var h = hashPara(doc);
    return estaHash(h);
}

//Devuelve true si está el hash
function estaHash(bytes32 h) constant returns (bool) {
    return hashesDocs[h];
}
}

```

Recuerde que para desplegar el nuevo contrato deberá:

- Salir de la consola de Truffle (`.exit`)
- Modificar el archivo `migrations/2_deploy_contracts.js` (`ValidaDocs2`)
- Compilar el código (`truffle compile`)
- Migrar con bandera de reset (`truffle migrate --reset`)

Despliegue el nuevo contrato y juegue con él. Verifique que se puede guardar más de un hash y que el accComisión funciona como se espera.

```

PS> truffle console
truffle(development)> var vd = ValidaDocs2.at(ValidaDocs2.address)

truffle(development)> vd.address
'0x2f72dd8f793c27f8c1f66eb7bfebb86ea909da5c'
truffle(development)> vd.checaDocumento('hola')
false
truffle(development)> vd.registra('hola')
{ tx: '0x06f55cd0b79fecb12d6899cdb81571c3e483ea477e0bc296430155a069397ec4',
...
logs: [] }

truffle(development)> vd.checaDocumento('hola')
true

truffle(development)> vd.registra('otro')

```

```

{ tx:
...
  logs: [],
  status: 1 },
logs: [] }

truffle(development)> vd.checaDocumento('otro')
true

truffle(development)>.exit
PS>

```

6. Tutorial del Pet shop

En esta sección, adaptada de <http://truffleframework.com/tutorials/pet-shop>, aprenderemos a:

1. Crear un proyecto de los disponibles como *Truffle Box*
2. Crear una interfaz de usuario (el *front end*) para interactuar con el smart contract.

6.1 Contexto

Pete Scandlon, propietario de *Pete's Pet Shop* está interesado en utilizar Ethereum para administrar la adopción de mascotas. La tienda tiene capacidad de albergar hasta 16 mascotas y ya cuenta con una base de datos de mascotas. Como prueba de concepto, Pete quisiera tener una *DApp* que asocie una dirección Ethereum con la mascota a adoptar.

El proyecto ya integra la estructura del sitio web para interactuar con la DApp. Nuestra tarea es escribir el *smart contract* y la lógica del front end.

6.2 Creación de un proyecto utilizando un *Truffle Box*

Primero creamos un directorio en la carpeta de nuestra elección y nos posicionamos en ella.

```

PS> mkdir pet-shop-tutorial
PS> cd pet-shop-tutorial

```

Se ha creado una *Truffle Box* especial para este tutorial llamada `pet-shop` que incluye la estructura del proyecto y el código para la interfaz de usuario. Desempaque la caja:

```
$ truffle unbox pet-shop  
Downloading...  
Unpacking...  
Setting up...
```

6.3 Escribir el smart contract

Empezamos a codificar nuestra DApp escribiendo el smart contract que implementará la lógica del back-end y servirá como almacenamiento.

6.3.1.- Cree un nuevo archivo llamado `Adoption.sol` en el directorio `contracts\ .` y agregue el siguiente contenido al archivo:

```
pragma solidity ^0.4.4;  
  
// Contrato Adoption, version 1  
contract Adoption {  
    address[16] public adopters; // Direcciones de los que van a adoptar  
  
    uint256 public accComision; // Eventual acumulador de comisiones  
    address public creador; // Quien invocó el contrato  
  
    // Para adoptar una mascota. Recibe qué mascota quiere  
    // devuelve lo mismo sólo para saber que ok  
    function adopt(uint petId) public returns (uint) {  
        require(petId >= 0 && petId <=15);  
        adopters[petId] = msg.sender;  
        return petId;  
    }  
  
    // Obten todo el arreglo de adopters  
    // getters proporcionados solo devuelven un valor  
    function getAdopters() public view returns(address[16]) {  
        return adopters;  
    }  
}
```

- La variable `adopters` es un arreglo de 16 direcciones Ethereum. Es una variable **Public**, lo que significa que **se generan métodos get para obtener el valor de la variable, automáticamente**. Sin embargo, para los arreglos se requiere de una llave (un índice) y se

devuelve un solo valor. Más adelante escribiremos una función para devolver todo el arreglo.

- `adopt` permite que los usuarios soliciten una adopción. La función verifica que el argumento esté dentro del rango [0:15]. Los arreglos se indexan a partir del cero. Si el argumento del comando `require()` es falso, se suspende el contrato y se hace un roll back, pero sí se cobra el valor en gas por la ejecución del contrato.

Si el argumento está dentro del rango, se agrega la dirección de quien hizo la llamada a la función. Recuerde que la dirección de quien invoca a la función se encuentra en `msg.sender`. Como confirmación de que la función se ejecutó correctamente, se devuelve el argumento.

6.3.2. Compilación y migración del smart contract

El contrato está listo. Ahora hay que compilarlo, verificar que no tiene errores y migrarlo.

```
$ truffle compile

Compiling ./contracts/Migrations.sol...
Compiling ./contracts/Adoption.sol...
Writing artifacts to ./build/contracts
```

Vamos a crear nuestro script de migración en el archivo `2_deploy_contracts.js` en el directorio `migrations/`. Cree el archivo y agregue el siguiente código:

```
var Adoption = artifacts.require("Adoption");

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

El box de pet shop supone que utilizaremos Ganache (otro blockchain para pruebas con interfaz gráfica) en vez de testrpc. Por omisión Ganache escucha en el puerto 7545. Cambie el puerto en el archivo `truffle.js` a 8545

Ahora lanzamos el comando para migrar los contratos:

```
$ truffle migrate

Using network 'development'.

Running migration: 1_initial_migration.js
```

```
Deploying Migrations...
...
Migrations: 0xd23f5ca2521af27b3ad579febd92b2310f71baac
Saving successful migration to network
...
```

Ahora que hemos escrito nuestro primer smart contract y lo desplegamos en un blockchain, es momento de interactuar a través de un front-end.

6.4 Interfaz de usuario

En vez de interactuar con nuestro smart contract via la terminal, lo haremos a través de una interfaz de usuario más amigable que ya ha sido programada y se encuentra dentro del proyecto `pet-shop`.

En la carpeta `src/` se encuentra el código del front-end. Es un código muy sencillo que ya tiene la estructura necesaria para la App; simplemente agregaremos las funciones específicas a Ethereum.

6.4.1 Instanciar web3

Ya hemos comentado que `web3` es *una API de Ethereum compatible con JavaScript que implementa la especificación genérica JSON RPC*.

Abra el archivo `/src/js/app.js` en un editor de texto y examine el archivo. Encontrará un objeto global `App` para administrar la aplicación. En `init()` carga los datos de las mascotas en formato json desde el archivo `pets.json` alojado en el directorio anterior.

Después llama la función `intiWeb3()`. La librería `web3` puede recuperar cuentas de usuarios, enviar transacciones e interactuar con smart contracts, entre otros.

Sustituya el comentario multi-línea en `initweb3` por el siguiente código:

```
// Is there an injected web3 instance?
if (typeof web3 !== 'undefined') {
  App.web3Provider = web3.currentProvider;
} else {
  // If no injected web3 instance is detected, fall back to Ganache
  App.web3Provider = new Web3.providers.HttpProvider('http://localhost:8545');
}
web3 = new Web3(App.web3Provider);
```

Este código primero revisa si no hay ya una instancia de web3 ejecutándose (algunos navegadores de Ethereum como Mist o MetaMask/Chrome crean sus propias instancias de web3). De ser así, se toma; de lo contrario, se crea una en localhost en el puerto 8545. En ambos casos, se crea el objeto `web3`.

6.4.2 Instanciar el contrato

Ya se puede interactuar con Ethereum a través de web3; ahora es necesario instanciar el smart contract para que web3 sepa dónde encontrarlo y cómo trabaja. Para ello, se utiliza la librería `contract` de Truffle. Esta librería mantiene información sobre el contrato en sincronía con las migraciones para no tener que cambiar manualmente las direcciones de los contratos desplegados.

Sustituya el comentario multi-línea en `initContract` por el siguiente código:

```
$getJSON('Adoption.json', function(data) {  
    // Get the necessary contract artifact file and instantiate it with truffle-contra  
    ct  
    var AdoptionArtifact = data;  
    App.contracts.Adoption = TruffleContract(AdoptionArtifact);  
  
    // Set the provider for our contract  
    App.contracts.Adoption.setProvider(App.web3Provider);  
  
    // Use our contract to retrieve and mark the adopted pets  
    return App.markAdopted();  
});
```

Esta función toma el archivo de Artefactos de nuestro smart contract. *Artefactos* se refiere a *información del contrato como su dirección al ser desplegado, y la interfaz de aplicación binaria ABI*. Esta interfaz en un objeto JavaScript que define cómo interactuar con el contrato, incluyendo sus variables, funciones y parámetros.

Una vez recuperados los Artefactos, éstos se pasan a `TruffleContract()`, con lo que **se crea una instancia del contrato con la que se puede interactuar**.

Se asigna el proveedor web3 al contrato instanciado con el valor almacenado al configurar web3. Finalmente se llama a la función `markAdopted()` de la App en caso de que algunas mascotas ya hayan sido adoptadas de una visita previa. Esto se encapsula en una función por separado ya que necesitaremos actualizar la UI cada vez que se haga un cambio a los datos del smart contract.

6.4.3 Actualización de la UI con las mascotas adoptadas

Todavía dentro de `/src/js/app.js`, sustituya el comentario multi-línea en `markAdopted` por el siguiente código:

```
var adoptionInstance;

App.contracts.Adoption.deployed().then(function(instance) {
    adoptionInstance = instance;

    return adoptionInstance.getAdopters.call();
}).then(function(adopters) {
    for (i = 0; i < adopters.length; i++) {
        if (adopters[i] !== '0x0000000000000000000000000000000000000000') {
            $('.panel-pet').eq(i).find('button').text('Success').attr('disabled', true);
        }
    }
}).catch(function(err) {
    console.log(err.message);
});
```

Se accede al contrato desplegado `Adoption` y se llama a la función `getAdopters()`. La variable `adoptionInstance` fuera de las llamadas al smart contract, permite acceder la instancia después de haberla recuperado.

El usar `call()` nos permite leer datos de blockchain sin tener que mandar una transacción completa. Por supuesto, tampoco debemos gastar `gas`.

Tras llamar a `getAdopters`, se hace un ciclo con todos ellos verificando si hay una dirección almacenada para cada una de las mascotas. Dado que el arreglo contiene tipos `address`, Ethereum inicializa el arreglo con 16 entradas vacías. Así es como se checa una dirección vacía, en vez de comparar con `NULL` o algún otro valor confuso.

Si se ha hallado un identificador de mascota con una dirección no vacía, se deshabilita su botón de `Adopt` y se cambia el texto a `Success` para que el usuario sepa que esa mascota ya fue adoptada.

Finalmente, si hay errores, éstos se envían a la consola.

6.4.4 Manejando la función `adopt()`

Sustituya el comentario multi-línea en `handleAdopt` por el siguiente código:

```

var adoptionInstance;

web3.eth.getAccounts(function(error, accounts) {
  if (error) {
    console.log(error);
  }

  var account = accounts[0];

  App.contracts.Adoption.deployed().then(function(instance) {
    adoptionInstance = instance;

    // Execute adopt as a transaction by sending account
    return adoptionInstance.adopt(petId, {from: account});
  }).then(function(result) {
    return App.markAdopted();
  }).catch(function(err) {
    console.log(err.message);
  });
});

```

Con `web3` se accede a las cuentas de usuario y se selecciona la primera cuenta. Posteriormente se toma el contrato desplegado y su instancia se almacena en `adoptionInstance`.

En esta ocasión se enviará una **transacción** en vez de un **call**, por lo que se gastará `gas`. La transacción requiere de una dirección *from*. La transacción se envía al ejecutar la función `adopt()` con el ID de la mascota y la dirección de la cuenta.

El resultado debe enviar la transacción es el “transaction object”. Si no hay errores, continuamos con la llamada a `markAdopted()` para sincronizar la interfaz de usuario con los nuevos datos almacenados.

6.5 Interactuando con la DApp a través de un navegador

¡Por fin estamos listos para usar la Dapp! Pero necesitamos un navegador. La manera más sencilla de obtenerlo, es a través de la extensión `MetaMask` para Chrome y Firefox.

6.5.1. En un navegador Chrome (o Firefox), instale la extensión `MetaMask`. Una vez instalado, verá el ícono de un zorro arriba a la derecha, junto a la barra de direcciones URL. De clic y aparecerá una pantalla con indicaciones de privacidad:

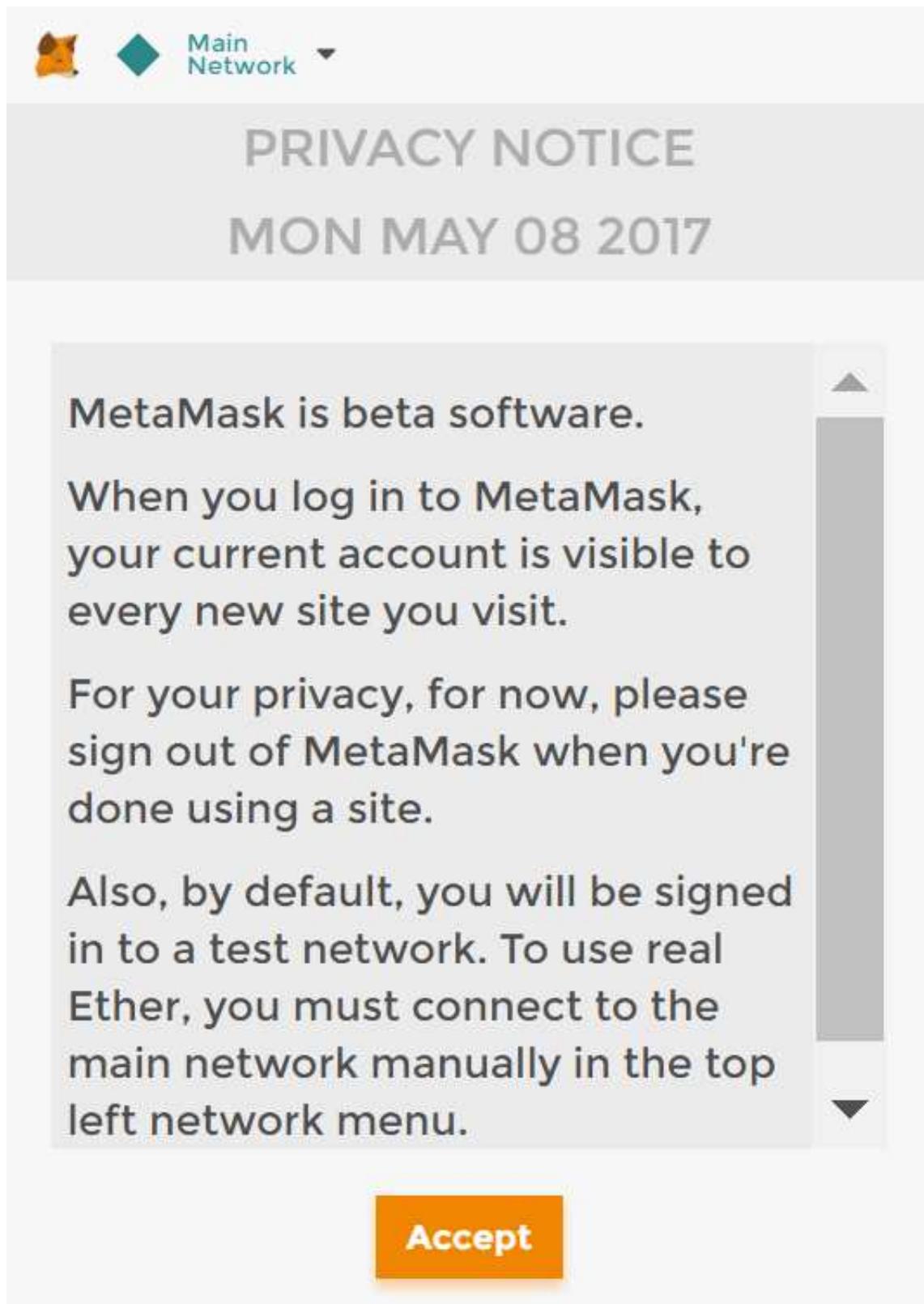


Fig. 3. Pantalla de privacidad MetaMask

6.5.2. De clic para aceptar el mensaje de privacidad. Ahora aparece la pantalla de términos de uso. Si puede léalos; de scroll hasta el final y de clic para aceptarlos.

6.5.3. Ahora encontrará la pantalla inicial de MetaMask. De clic en *Import Existing DEN*

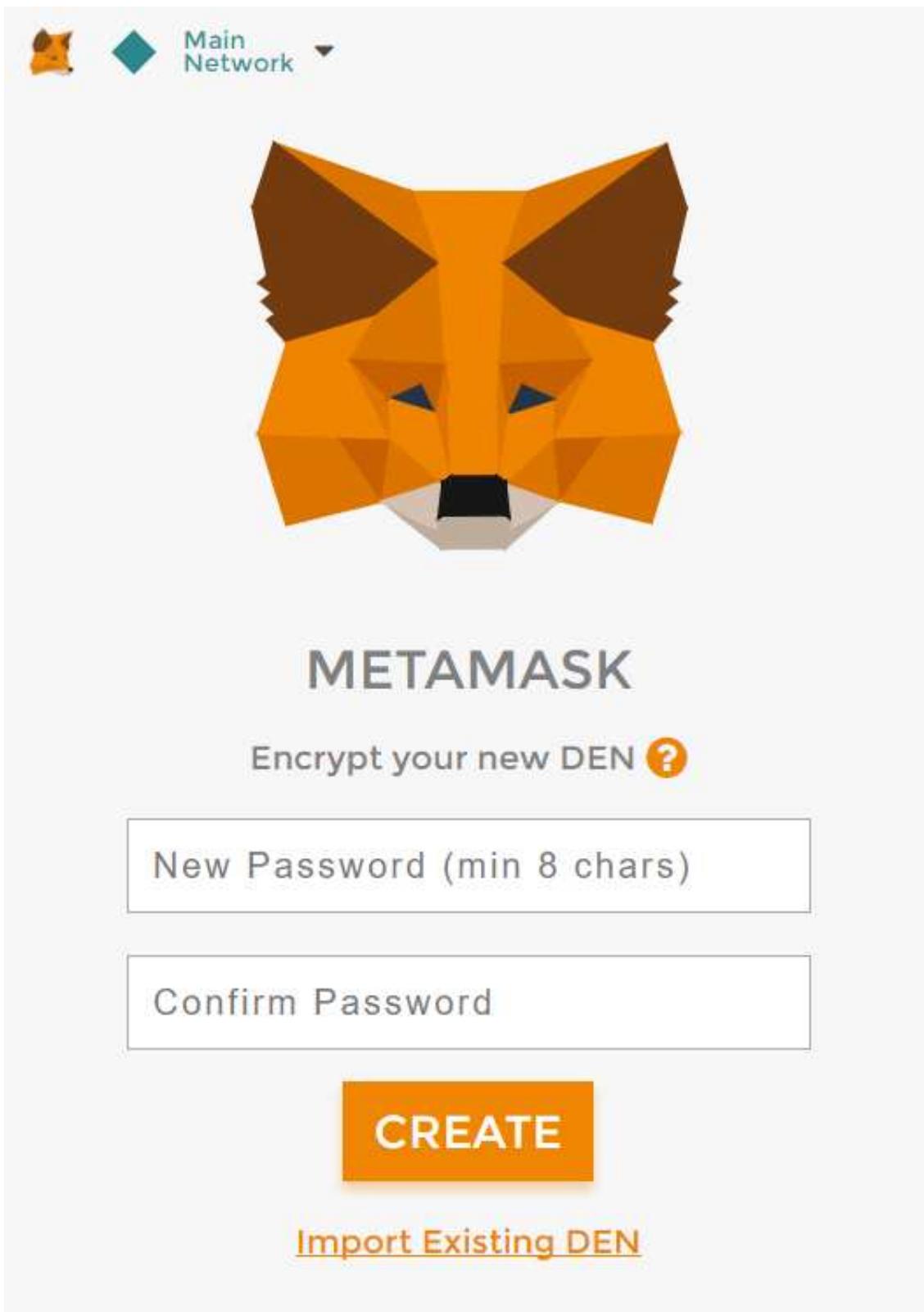


Fig. 4. Pantalla de inicio MetaMask

6.5.4. En la caja llamada **Wallet Seed** introduzca el mnemónico desplegado en **testrpc**, Fig.2, elija una contraseña, reescríbala y de clic en **OK**. Si no puede ver el mnemónico, simplemente detenga y lance de nuevo **testrpc**.

```
wear hire imitate boat staff gloom gaze supreme name brief gorilla attend
```

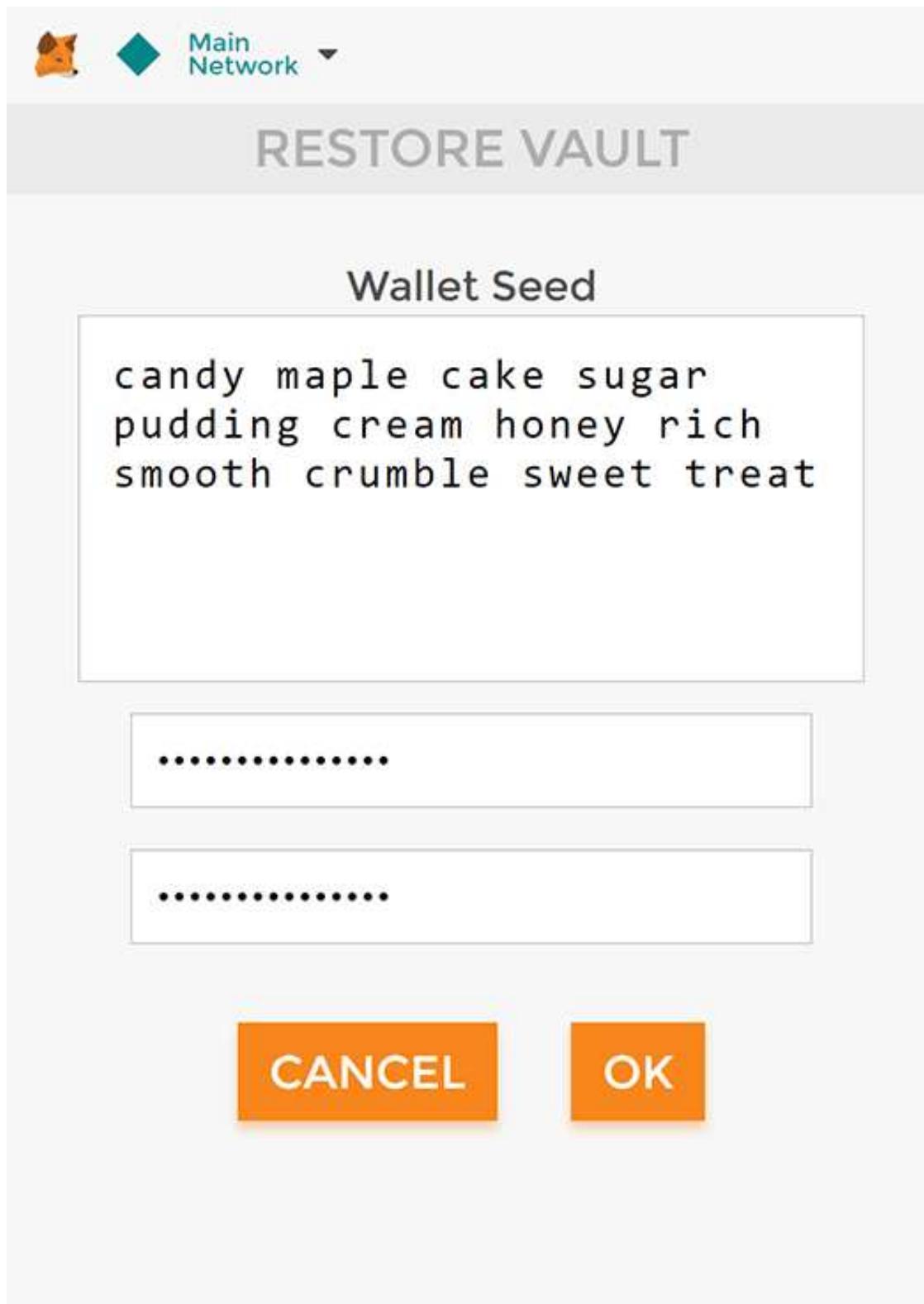


Fig. 5. Pantalla para entrar con la frase semilla a MetaMask

6.5.5. Se debe conectar MetaMask al blockchain creado por Ganache. De clic en el menú que muestra *MainNetwork* y seleccione **Custom RPC**.

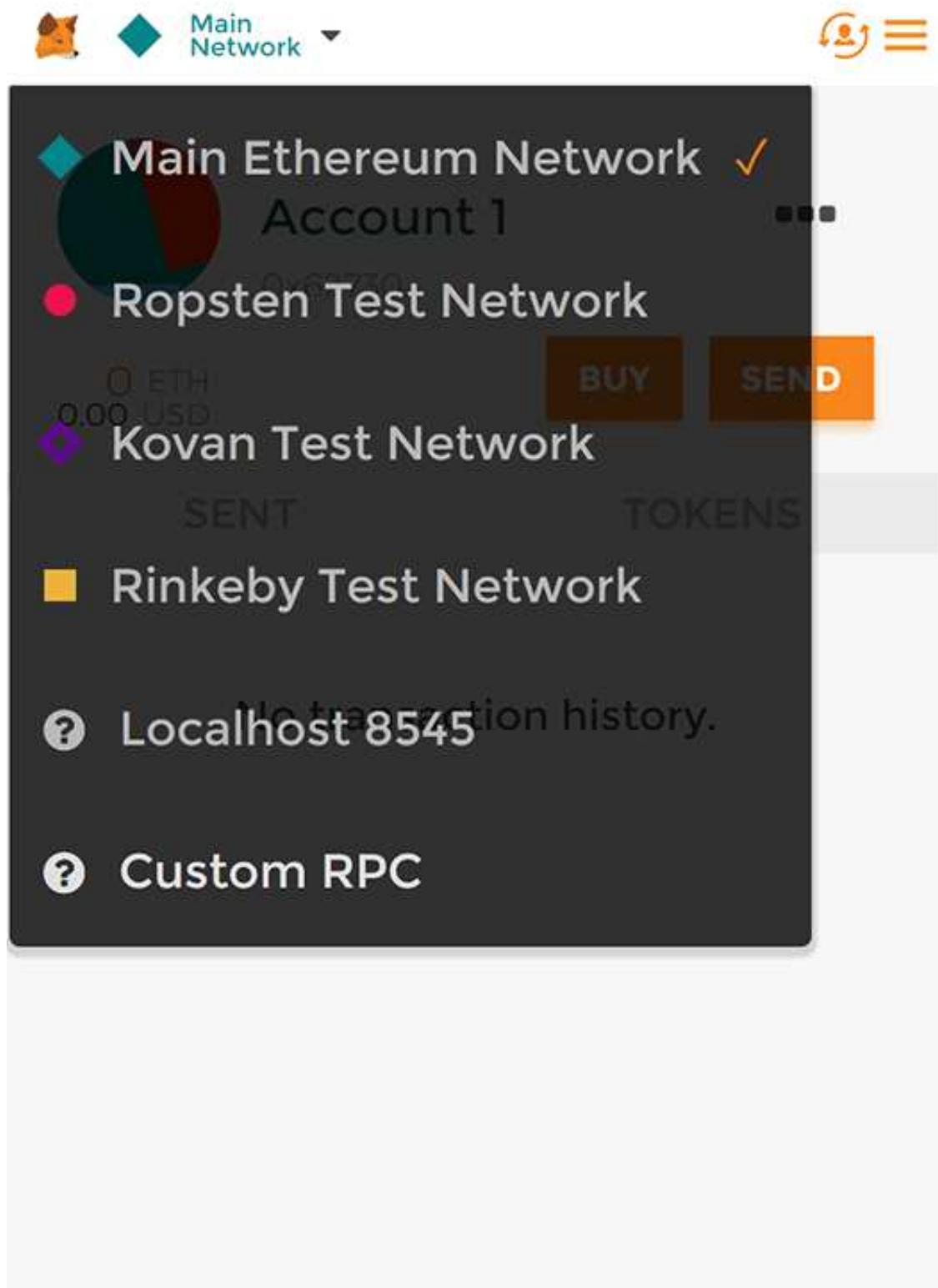


Fig. 6. Pantalla de menú de red MetaMask

6.5.6. En la pantalla que aparece con la leyenda *New RPC URL* inserte `http://127.0.0.1:8545` y de clic en *Save*

El nombre de la red aparecerá arriba como *Private Network*

6.5.7: De clic en la flecha que apunta hacia la izquierda para regresar a la página de las cuentas.

Cada cuenta creada por testrpc recibe 100 ether. Observará que en la cuenta desplegada hay una cantidad un poco menor. La diferencia es el gas utilizado para desplegar el contrato y en las pruebas que se hicieron.

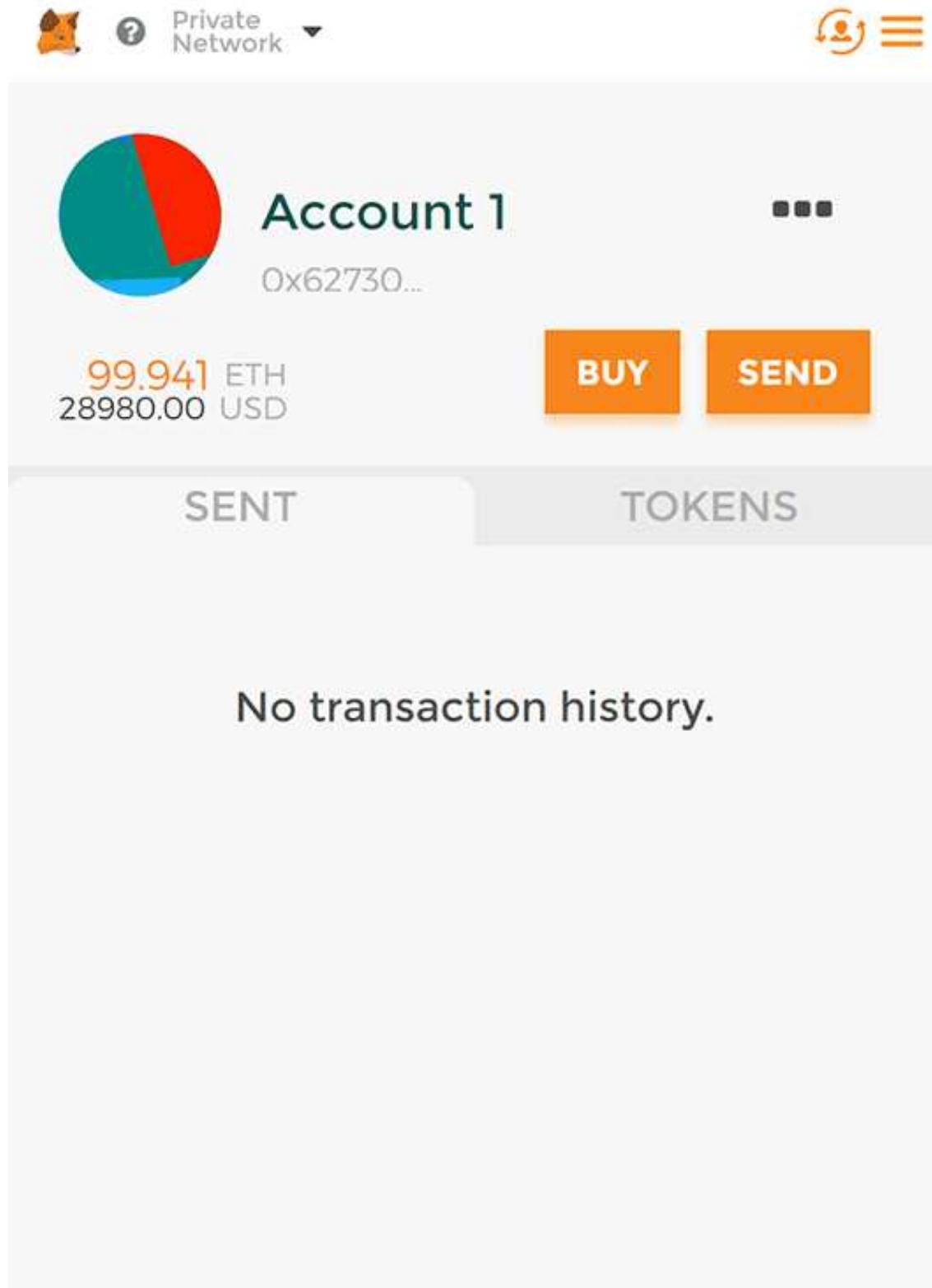


Fig. 7. Pantalla para entrar con la frase semilla a MetaMask

6.6. Instalar y configurar *lite-server*

Ahora podemos arrancar un servidor web local para usar la DApp. La librería `lite-server` viene con el paquete del tutorial `pet-shop` y es suficiente para mostrar los archivos estáticos que se utilizarán. Veamos cómo funciona.

Abra el archivo `bs-config.json` que se encuentra en el directorio raíz del proyecto, con un editor de textos y examine su contenido:

```
{  
  "server": {  
    "baseDir": ["./src", "./build/contracts"]  
  }  
}
```

Se está indicando a `lite-server` qué archivos incluir en nuestro directorio base: `/src` para los archivos del sitio web y `./build/contracts/` para los artefactos del contrato.

En el archivo `package.json` también se ha agregado un comando `dev` al objeto `scripts`. Este objeto permite *apodar* comandos de la consola en un solo comando `npm`. En este caso se está incluyendo un solo comando pero es posible tener configuraciones más complejas.

Al ejecutar `npm run dev` se correrá la instalación local de `lite-server`:

```
"scripts": {  
  "dev": "lite-server",  
  "test": "echo \\\"Error: no test specified\\\" && exit 1"  
},
```

6.7. Uso de la dapp

Asegúrese de que el navegador en donde tiene las extensiones **MetaMask** es el navegador por omisión

Ejecute el comando `npm run dev` y se lanzará una nueva pestaña en el navegador conteniendo la vista de la aplicación:

Pete's Pet Shop

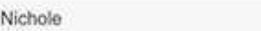
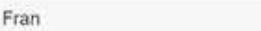
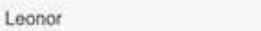
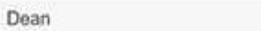
Frieda	Gina	Collins	Melissa
			
Breed: Scottish Terrier Age: 3 Location: Lisco, Alabama	Breed: Scottish Terrier Age: 3 Location: Tooleville, West Virginia	Breed: French Bulldog Age: 2 Location: Freeburn, Idaho	Breed: Boxer Age: 2 Location: Camas, Pennsylvania
Adopt	Adopt	Adopt	Adopt
Jeanine	Elvia	Latisha	Coleman
			
Breed: French Bulldog Age: 2 Location: Gerber, South Dakota	Breed: French Bulldog Age: 3 Location: Innsbrook, Illinois	Breed: Golden Retriever Age: 3 Location: Soudan, Louisiana	Breed: Golden Retriever Age: 3 Location: Jacksonwald, Palau
Adopt	Adopt	Adopt	Adopt
Nichole	Fran	Leonor	Dean
			

Fig. 8. Pantalla de la tienda de mascotas

De clic en el botón **Adopt** de alguna mascota. Automáticamente MetaMask le solicitará aprobar la transacción. De clic en **Submit** para aprobarla.

CONFIRM TRANSACTION

?

Private Network

Account 1

627306...Ef57 > 345cA3...3e10

99.941 ETH
28867.07 USD

Amount 0 ETH
0.00 USD

Gas Limit 63162 UNITS

Gas Price 20 GWEI

Max Transaction Fee 0.001263 ETH
0.36 USD

Max Total 0.001263 ETH
0.36 USD

Data included: 36 bytes

RESET **SUBMIT** **REJECT**

Fig. 9. Pantalla para aprobar la transacción

Tal como se programó, verá cómo cambia el botón a Success y queda deshabilitado.

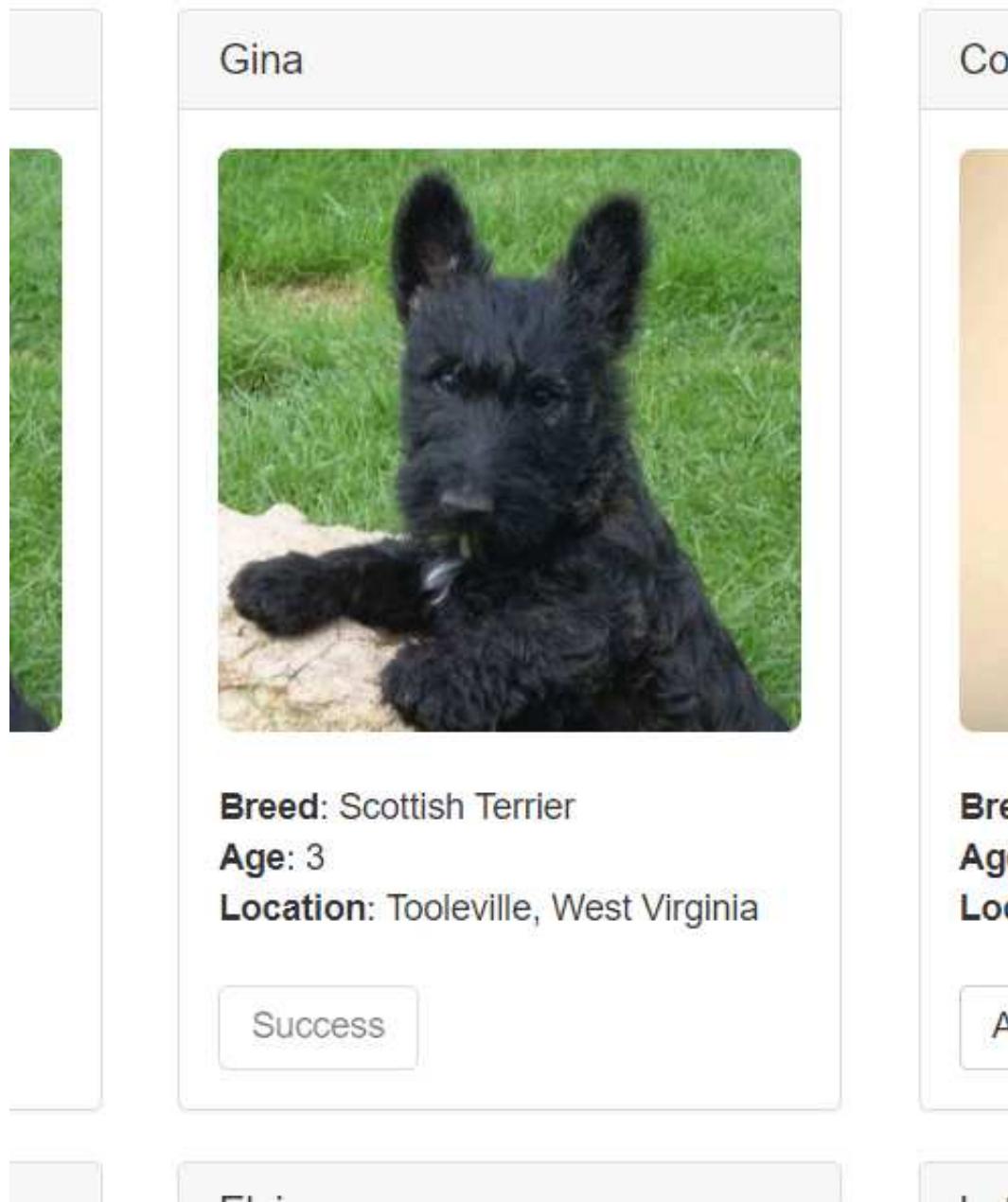


Fig. 10. Pantalla para aprobar la transacción

Tanto en MetaMask como en Ganache (menu *Transactions*) podrá ver listada la transacción.

Apéndice. Instalación de las herramientas

Instalación en Windows 7

La guía de instalación está tomada de [aquí](#).

1.- Instalar [Chocolatey](#)

- Lanzar una ventana de PowerShell con permisos de administrador (clic botón derecho, Ejecutar como Administrador).
- Ejecute el comando `Get-ExecutionPolicy`. Si devuelve `Restricted`, ejecute `Set-ExecutionPolicy AllSigned` o `Set-ExecutionPolicy Bypass -Scope Process`.

```
PS C:\WINDOWS\system32> Get-ExecutionPolicy
Restricted
PS C:\WINDOWS\system32> Set-ExecutionPolicy Bypass -Scope Process

Cambio de directiva de ejecución
La directiva de ejecución te ayuda a protegerte de scripts en los que no confías. Si cambias dicha directiva, podrías exponerte a los riesgos de seguridad descritos en el tema de la Ayuda about_Execution_Policies en https://go.microsoft.com/fwlink/?LinkId=135170. ¿Quieres cambiar la directiva de ejecución?
[S] Sí [O] Sí a todo [N] No [T] No a todo [U] Suspender [?] Ayuda (el valor predeterminado es "N"): S
PS C:\WINDOWS\system32>
```

Fig. A1. Cambio de política de ejecución en PowerShell

- Ejecute el siguiente comando para instalar chocolatey :

```
PS> Set-ExecutionPolicy Bypass -Scope Process -Force; iex ((New-Object System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))
```

2.- Instalar las herramientas `npm` y `git` a través de chocolatey .

Desde la misma consola de PowerShell con permisos de administrador, ejecutar los siguientes comandos:

```
PS> choco install nodejs.install -y ; Va a tomar algo de tiempo
PS> choco install git -y
PS> choco install VisualStudioCode -y ; Esto es opcional
```

```
PS C:\WINDOWS\system32> choco install nodejs.install -y
Chocolatey v0.10.8
Installing the following packages:
nodejs.install
By installing you accept licenses for the packages.
Progress: Downloading nodejs.install 9.5.0... 100%

nodejs.install v9.5.0 [Approved]
nodejs.install package files install completed. Performing other installation steps.
Installing 64 bit version
Installing nodejs.install...
nodejs.install has been installed.
  nodejs.install may be able to be automatically uninstalled.
Environment Vars (like PATH) have changed. Close/reopen your shell to
see the changes (or in powershell/cmd.exe just type `refreshenv`).
The install of nodejs.install was successful.
  Software installed as 'msi', install location is likely default.

Chocolatey installed 1/1 packages.
See the log for details (C:\ProgramData\chocolatey\logs\chocolatey.log).
PS C:\WINDOWS\system32> choco install git -y
Chocolatey v0.10.8
Installing the following packages:
git
By installing you accept licenses for the packages.
Progress: Downloading git.install 2.16.1.4... 26%
```

Fig. A2. Instalación de npm y de git

3.- Instalar las herramientas con npm :

Abrir una ventana **NUEVA** de PowerShell con permisos de administrador para asegurar que la instalación anterior se recargó, y ejecutar los siguientes comandos:

```
PS> npm install -g npm
PS> npm install -g -production windows-build-tools
PS> npm install -g ethereumjs-testrpc
PS> npm install -g truffle
```

```

PS D:\Aplicaciones> npm install -g ethereumjs/testrpc
¿Desea terminar el trabajo por lotes (S/N)? S
PS D:\Aplicaciones> npm install -g ethereumjs-testrpc
npm WARN          ethereumjs-testrpc@6.0.3: ethereumjs-testrpc has been renamed to gan
e from now on.
C:\Users\jincera\AppData\Roaming\npm\testrpc -> C:\Users\jincera\AppData\Roaming\npm\nod
e\cli.node.js

> uglifyjs-webpack-plugin@0.4.6 postinstall C:\Users\jincera\AppData\Roaming\npm\node_m
odules\uglifyjs-webpack-plugin
> node lib/post_install.js

npm WARN          SKIPPING OPTIONAL DEPENDENCY: fsevents@1.1.3 (node_modules\ethereumjs-
npm WARN          SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.1.3: w
"} (current: {"os":"win32","arch":"x64"})

+ ethereumjs-testrpc@6.0.3
added 251 packages in 49.316s
PS D:\Aplicaciones> npm install -g truffle
C:\Users\jincera\AppData\Roaming\npm\truffle -> C:\Users\jincera\AppData\Roaming\npm\nod
led.js
+ truffle@4.0.6
updated 1 package in 9.548s
PS D:\Aplicaciones>

```

Fig. A3. Instalación de testrpc y truffle

Las instalaciones anteriores pueden marcar unos errores; la mayoría son mensajes informativos o componentes no críticos. Podemos probar si la instalación se efectuó correctamente ejecutando los comandos `truffle` y `testrpc`.

Instalación en Windows 10 con GitBash

En Windows 10 se puede trabajar con `ganache-cli`, la versión actualizada de `testrpc`.

Opción 1 (La mejor): Instalación desde cero, sin un node

```

# (opcional) Asegúrese de que Ubuntu está actualizado
$ sudo apt-get update -y && sudo apt-get upgrade -y

# instalar nvm
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.6/install.sh | bas
h

# reiniciar bash para habilitar nvm
$ exec bash

```

```
# instalar node y los paquetes npm
$ nvm install node
$ npm install -g truffle ganache-cli
```

Opción 2: Instalación en un equipo que ya tiene nvm

Se debe tener la versión 8 o superior de node y la 5.3.0 o superior de npm. Esto se checa con `node -v` , `npm -v`

```
```bash
Actualice node a la última versión e instale las herramientas
$ nvm install node
$ npm install -g truffle ganache-cli
```