

# Introduccion al Desarrollo de Smart Contracts con Ethereum

Elvi Mihai Sabau Sabau<sup>[emss5]</sup>

<sup>1</sup> Universidad de Alicante, Alicante, España.  
*frenzoid@pm.me*



**Abstract:** En este documento explicaremos cómo programar en Solidity, como crear una interfaz para nuestro contrato, como usar Hardhat para compilar y probar / desplegar localmente nuestros contratos, y como desplegar un contrato en su totalidad ( crear una Dapp ).

**Keywords:** Ethereum, Solidity, Dapps, DLT, Blockchain, Hardhat, Testnets.

## 1 Introducción.

### 1.1 Sobre este documento.

Este documento es la continuación del documento “Estudio y Investigación de Ethereum”, es recomendable echarle un vistazo antes de empezar este documento, ya que en ese se explican ciertos conceptos fundamentales para el desarrollo de dapps, igualmente, no es obligatorio, ya que intentaremos explicar cada paso.

No vamos a indagar en totalidad sobre la programación en Solidity, ya que este lenguaje de programación tiene muchas características ( conversiones implícitas, truncamientos, abi encoders, etc ), solo nos basaremos en dar una introducción a este lenguaje de programación, y a como desarrollar una aplicación funcional.

## 2 Smart Contracts.

Los smart contracts son en general programas en bytecode ( código a bajo nivel ) que se ejecutan en los nodos de la blockchain de una forma u otra ( dependiendo del consenso o del tipo de blockchain ), pero normalmente, se suele desarrollar en otros lenguajes de alto nivel que compilan a bytecode.

En el caso de Ethereum, hay varias librerías para lenguajes como JS, Python o incluso C++ que compilan a bytecode que ejecuta la EVM ( Ethereum Virtual Machine ), aún así, se ha desarrollado un lenguaje específico para el desarrollo de smart contracts que compila a bytecode de EVM llamado Solidity.

## 3 Solidity.

Este lenguaje nos permite desarrollar Smart Contracts de forma cómoda, además Solidity posee sintaxis y definiciones que son similares a C++ y JS, aún así es **IMPORTANTE NO CONFUNDIR** Solidity como un lenguaje orientado a objetos, ya que este no entra en este paradigma ni de lejos.

### 3.1 Comportamiento.

Un smart contract, al menos en Ethereum, se comporta muy similar a un microservicio REST. Cada contrato tiene un conjunto de funciones, que se ejecutan sólo cuando se las llaman, estas pueden recibir, procesar, y devolver datos, además de alterar el estado de la blockchain, o incluso llamar a funciones de otros contratos desplegados en la misma blockchain.

Una vez desplegado un contrato, se genera una instancia de este en la blockchain, **en todos los nodos de la red**, y su código fuente no puede ser modificado y cada instancia es independiente.

Debido a que la EVM es una máquina de estados, al cambiar el valor de una *variable de estado* en nuestro contrato, lo que estamos haciendo en realidad es cambiar el estado de la blockchain.

Esto no significa que se esté sobrescribiendo dicho valor, sino, que se actualiza el valor más reciente para dicho espacio de almacenamiento en la blockchain.

En cualquier momento podemos ver los valores previos de estas variables a través del explorador de bloques, ya que cada cambio de estado es provocado por una transacción que debe ser minada, validada y finalmente archivada en un bloque.

Además, para cada contrato desplegado, se generan una cuenta de contrato, esto significa que el contrato en la blockchain tiene una dirección y una cartera para mantener Ether.

Una vez desplegado un contrato, para llamar a sus funciones deberemos saber la dirección del contrato, y la firma de la función ( identificador de función ).

## 4 Estructura de un smart contract en Solidity.

La estructura de un Smart Contract es bastante sencilla, los Smart Contracts en Solidity suelen tener principalmente un conjunto de variables globales ( variables de estado ), un constructor y funciones. A esto se le pueden añadir eventos, modificadores y librerías que queramos importar, pero al fin y al cabo, cuando interactuemos con un contrato, lo que haremos será llamar a una función de este, o obtener el valor de una variable de estado.

### 4.1 Variables.

#### 4.1.1 Almacenamiento.

Las variables de un contrato se guardan en 3 ubicaciones diferentes en la blockchain.

- **Storage:** Persistente, se guarda en la blockchain.
- **Memory:** Variables locales / temporales, arrays.
- **Calldata:** Valores recibidos por parámetros de funciones, acceso externo.

Al igual que la RAM, Memory en Solidity es un lugar temporal para almacenar datos, mientras que Storage guarda datos entre llamadas a funciones ( Variables de Estado ).

Un smart contract puede usar cualquier cantidad de Memory durante la ejecución, pero una vez que la ejecución se detiene, la memoria se borra por completo para la próxima ejecución.

Mientras que Storage, por otro lado, es persistente, cada ejecución del contrato tiene acceso a los datos previamente almacenados en el área de almacenamiento.

Una lógica que cambia el valor de una variable de estado altera el valor del Storage, y por ello requiere una transacción, Storage es costoso ( en gas ) de leer y mucho más de declarar y actualizar, así que se recomienda minimizar el número de accesos a las variables de estado, y solo acceder a las variables de estado cuando sea necesario.

Las variables de estado solo se pueden declarar a nivel de contrato, y las variables en Memory solo se pueden declarar a nivel de función.

El tercer espacio de almacenamiento se llama Calldata, y es similar a Memory, pero esta ubicación es especial, ya que contiene los argumentos de la función, y solo es disponible para parámetros de llamadas a funciones externas.

Solo hace falta especificar la zona de memoria cuando se hacen traspasos de datos o a la hora de declarar variables dinámicas o secuenciales como arrays.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Storage {
    /// @dev numero en este caso es una variable de estado almacenada en Storage.
    /// Por defecto, las variables declaradas a nivel de contrato se guardan en Storage.
    uint256 numero;

    /// @dev _numero en este caso es una variable recibida por parametro almacenada en calldata.
    function store(uint[] calldata _numero) public {
        // Declaramos un array de enteros sin signo de tamaño 5.
        // Esta variable es local, y guardada en Memory.
        uint[] memory misNumeros = new uint[](5);

        // Guardamos en Storage el valor de la posición 0 del array _numero.
        numero = _numero[0];

        // Variable local, guardada en Memory.
        uint8 n = 4;
    }
}
```

#### 4.1.2 Tipos de variables.

En solidity tenemos varios tipos de variables, y de tamaños. Tenemos los conocidos, string, int, uint, arrays ( int[] etc ...), pero además podemos especificar el tamaño de la variable, como por ejemplo: uint8, uint32, uint64..., esto nos sirve para reducir el coste de gas asociado a la reserva de memoria a la hora de definir las y interactuar con las variables.

Si no especificamos un tamaño, eje: uint, se definirá con su tamaño máximo:  
uint = uint256

Además de estos clásicos tipos de variables, tenemos los siguientes:

- bytes: Define un byte.
- address: Define una dirección de cuenta.
- mapping(tipo (clave) => tipo (valor)): Define un diccionario.
- Struct: Clasicos structs de C.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Variables {
    uint numeroSigno;
    int numeroConSigno;
    string nombre;
    bytes2 dosChars;
    address direccion;
    mapping(address => string) nombres;

    struct persona {
        string nombre;
        uint edad;
    }

    function setNombre(address _direccion, string memory _nombre) public {
        nombres[_direccion] = _nombre;
    }
}
```

#### 4.1.3 Inmutables.

Las variables inmutables son como constantes. Los valores de las variables inmutables se pueden establecer dentro del constructor, pero no se pueden modificar después.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Immutable {

    address immutable MY_ADDRESS;
    uint immutable MY_UINT;

    constructor(uint _myUint) {
        MY_ADDRESS = msg.sender;
        MY_UINT = _myUint;
    }
}
```

## 4.2 Pragma y Licencia.

Las primeras 2 líneas de cada archivo .sol:

- La licencia del código del archivo.
- La versión del compilador de solidity a usar.

En caso de no especificarlo, nos saltará un warning, y en caso de especificarlo y compilar con una versión diferente o incompatible, se abortará el proceso.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
```

En este ejemplo se especifica que la licencia es GPL-3, y que la versión del compilador debe ser superior o igual a 0.7.0 y inferior a la 0.9.0.

## 4.3 Constructor.

El constructor de un smart contract solo se ejecuta UNA VEZ, cuando se despliega, puede recibir datos por parámetro, o incluso llamar al constructor padre si este contrato hereda de otro.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Acontract {
    string name;

    /// @dev Así es un constructor.
    constructor(string memory _name) {
        name = _name;
    }
}

/// @dev Así se hereda.
contract Constructor is Acontract {
    uint number;

    /// @dev Así se llama al constructor padre.
    constructor(string memory _name, uint _number) Acontract(_name){
        number = _number;
    }
}
```

## 4.4 Herencia.

La herencia de un contrato NO IMPLICA heredar el ESTADO del padre ( valores actuales de las variables del padre ), solo sus variables, funciones, valores iniciales etc..

No se puede heredar de un contrato desplegado, solo del código fuente, también se pueden crear interfaces.

## 4.5 Visibilidades.

La visibilidad de una función o variable restringe quién puede acceder a dicho elemento:

- **Public:** Cualquiera puede llamar a la función, (usuario, otro contrato, función interna, contrato heredado ).
- **External:** Solo se puede llamar desde fuera del contrato actual, ( usuario, otro contrato ).
- **Private:** Solo contratos heredados o funciones internas pueden llamar a la función ( función interna, contrato heredado ).
- **Internal:** Solo funciones internas del contrato actual pueden llamar a la función ( función interna ).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Funciones {
    function f1() public {
        // Cualquier puede llamarla.
    }

    function f2() external {
        // Solo cuentas externas puede llamarla.
    }

    function f3() private {
        // Solo el contrato actual y heredados pueden llamarla.
    }

    function f4() internal {
        // Solo el contrato actual puede llamarla.
    }
}
```

Cuando una variable es pública, el compilador crea un getter implícito, por ello no es necesario crear getters para los atributos.

En el caso de las variables, **no es posible ocultar el valor** denegando el acceso a la variable usando private o internal, lo único que provocará será que sea más difícil de acceder al dato, de todas formas, el dato se sigue pudiendo leer en la blockchain usando un block explorer, hay que recordar que ethereum es una blockchain publica, así que todo lo que guardamos en los contratos es público para **TODO** el mundo.

```
contract Variables {

    address public immutable MI_DIRECCION;
    uint private immutable MI_UNIDAD;
    string external immutable MI_STRING;
    bytes8 internal immutable B8;

}
```

## 4.6 Funciones.

Como hemos mencionado anteriormente, el punto principal de conexión son las funciones, a esta son a las que se llaman para realizar operaciones sobre la blockchain.

### 4.6.1 Mutabilidad.

La mutabilidad define si la función actual va a cambiar / acceder a alguna variable de estado, esto influye muchísimo, ya que si una función no cambia el estado de la blockchain, tampoco habrá propagación, y por lo tanto no se necesita realizar una transacción, cose que hace que el llamar a esta función sea gratis. Las operaciones lógicas se ejecutarán sólo en un nodo.

- **View:** Especifica que la función no va a modificar las variables de estado ( no se altera la blockchain ).
- **Pure:** Igual que la anterior pero que además especifica que no se va a leer de las variables de estado ( directamente no se accede a la blockchain ).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Funciones {
    uint256 numero;

    function setNumero(uint256 _numero) public {
        numero = _numero;
    }

    function getNumero() public view returns (uint256) {
        return numero;
    }

    function suma(uint256 a, uint256 b) public pure returns (uint256) {
        return a + b;
    }
}
```

### 4.6.2 Valor a devolver.

Como hemos visto en el ejemplo anterior de mutabilidad, si una función devuelve un valor, se debe especificar el tipo, si no devuelve nada, no se especifica nada.

También se puede declarar directamente la variable a devolver como un “contenedor”, que la función al acabar su ejecución, devuelve automáticamente su valor.

- Metodo A:

```
contract Funciones {
    function Ma() public pure returns(uint){
        uint numero = 1;
        return numero;
    }
}
```

- Metodo B:

```
contract Funciones {
    function Mb() public pure returns(uint numero){
        numero = 1;
    }
}
```

De todas maneras, ambos métodos son equivalentes.

También se puede devolver múltiples valores.

```
function devuelve2() internal pure returns(uint a, string b) {
    a = 1;
    b = "hola";
}

function captura2() public pure {
    (uint numero, string palabra) = devuelve2();
}
```

#### 4.6.3 Payable.

Un contrato al fin y al cabo es una cuenta de ethereum que al igual que una cuenta normal puede mantener un balance, llamar a otros contratos, y mandar transferencias ordinarias de Ether tanto a usuarios como a otros contratos.

Dicho esto, cada contrato tiene un balance, puedes ver el balance actual del contrato del siguiente modo: `address(this).balance`.

Dicho esto otro, se puede habilitar que una función pueda recibir Ether y se guarde en la cuenta del contrato, similar a como sería una transacción entre 2 personas, pero con código ejecutado por la función.

Para esto solo tenemos que usar la palabra “payable”, esta palabra también se puede aplicar a direcciones para transferir ether del contrato a una cuenta.

También podemos acceder a la dirección que la cuenta que ha llamado a la función (`msg.sender`) y a la cantidad de Ether enviada (`msg.value`).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Deposito {
    mapping(address => uint) balances;

    function depositar() public payable {
        balances[msg.sender] += msg.value;
    }

    function retirar(uint256 cantidad) public {
        balances[msg.sender] -= cantidad;
        (bool success, ) = msg.sender.call{value: cantidad}("");
        require(success, "Fallo en la transferencia, revirtiendo");
    }

    function totalDepositadoPorTodos() public view returns(uint256) {
        return address(this).balance;
    }
}
```



#### 4.7 Requires.

Los requires son comprobaciones que en caso de fallar, revierte la transacción, esto implica que CUALQUIER CAMBIO realizado HASTA EL MOMENTO se deshace.

El gas para la transacción se consume solo hasta que se ejecuta la reversión, debido a que, aunque se revientan todos los cambios, se necesitaba poder computacional para realizar dichas operaciones.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Deposito {

    address propietario;

    mapping(address => uint) balances;

    constructor() {
        /// Cuando se despliega, se setea "propietario" con la
        /// dirección de la cuenta ( usuario ) que ha desplegado el contrato.
        propietario = msg.sender;
    }

    function depositar() public payable {
        /// ...
    }

    function retirar(uint cantidad) public {
        /// ...
    }

    function vaciar() public {
        /// (CONDICION, MENSAJE DE REVERSIÓN)
        /// Si la condicion falla, se revierte con el mensaje de error.
        require(msg.sender == propietario, "El usuario no es el propietario!");
        (bool success, ) = msg.sender.call{value: address(this).balance}("");
        require(success, "Error en la transferencia de Ether");
    }
}
```

## 4.8 Modificadores.

Los modificadores son funciones auxiliares que se declaran con la palabra “modifier”, y se adjuntan a otras funciones, de tal manera que la lógica de la función adjunta se ejecuta antes, durante o después de la auxiliar. Se entenderá mejor con un ejemplo:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Deposito {

    address propietario;

    // ...

    /// Modificador que comprueba si el usuario que ha llamado a la funcion
    /// es el propietario.
    modifier soloPropietario() {
        require(
            msg.sender == propietario,
            "El usuario no es el propietario!"
        );

        /// El _ se reemplaza al compilar con la lógica a ejecutar
        /// de la función a la que se adjunta.
        _;
    }

    // ...

    function vaciar() public soloPropietario {
        (bool success, ) = msg.sender.call{value: address(this).balance}("");
        require(success, "Error en la transferencia de Ether");
    }
}
```

También soporta atributos.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Suma {

    modifier esPar(uint numero1, uint numero2) {
        require((numero1 % 2 == 0) && (numero2 % 2 == 0), "Uno de los dos numeros no es par");
        _;
    }

    function sumaPares(uint a, uint b) public pure esPar(a, b) returns(uint) {
        return a + b;
    }
}
```

#### 4.9 Funciones por defecto.

En Solidity hay 2 funciones que se ejecutan por defecto ( si están definidas ):

- **receive() external payable:** Si el contrato solo recibe ether, sin que se haya llamado a ninguna función ( recordemos que un contrato al fin y al cabo es una cuenta, y puede recibir transferencias ordinarias ).
- **fallback() external payable:** Si la función que se ha llamado no existe.

Por defecto, estas funciones no están definidas, pero si se requiere realizar alguna lógica específica en caso de que alguna de las situaciones anteriores ocurra, solo hay que definir la función con el nombre y parámetros especificados anteriormente.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Contrato {

    receive() external payable {
        // Se ejecuta cuandor recibe ether a través de una transferencia ordinaria.
    }

    fallback() external payable {
        // Se ejecuta cuando el identificador de funcion no existe. ( como un default en un switch ).
    }

}
```

#### 4.10 Transacciones.

En Solidity tenemos 3 maneras de mandar ether desde nuestro contrato a un usuario u otro contrato.

- **Send:** devuelve false si falla, gas NO ajustable.

```
function enviarEther() public {
    bool success = payable(msg.sender).send(1 ether);
    require(success);
}
```

- **Transfer:** revierte si falla, gas NO ajustable.

```
function enviarEther() public {
    payable(msg.sender).transfer(1 ether);
}
```

- **Call{value}(""):** devuelve false si falla, gas ajustable.

```
function enviarEther() public {
    (bool success1, ) = msg.sender.call{value: 1 ether, gas: 500000}("");
    // o
    (bool success2, ) = msg.sender.call{value: 1 ether }("");
    require(success1 && success2);
}
```

Call también sirve para llamar a funciones de otros contratos y obtener su resultado, especificando la interfaz de la función a llamar, en este caso al llamar a la función "" estaremos ejecutando la función por defecto "receive" ( si la tiene, si no la tiene, el contrato destino simplemente recibirá el ether enviado [1 ether] ).

Es recomendable usar call si se va a transferir ether a otro contrato, debido a que se puede ajustar el gas de la transacción, el problema con transfer y send es que no se puede ajustar el gas, y si el contrato destino tiene una función receive definida, es muy probable que la transferencia se quede sin gas al ejecutar instrucciones extra.

#### 4.11 Llamadas externas.

Como ya hemos mencionado, un contrato puede llamar a otro contrato ( de una función a otra ).

Para hacer esto tenemos 2 maneras:

- Usando el código del contrato, o una Interfaz ( La manera sencilla ).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract ContratoExterno {

    function saludo(string memory nombre) external pure returns(string memory) {
        return string(bytes.concat("Hola ", bytes(nombre)));
    }
}

contract Contrato {
    address direccionContratoExterno;

    constructor(address _dirContExterno) {
        direccionContratoExterno = _dirContExterno;
    }

    function saludar(string memory nombre) public payable returns(string memory) {
        return ContratoExterno(direccionContratoExterno).saludo(nombre);
    }
}
```

- Especificando el método en la llamada ( La manera complicada ), al especificar el método, “abi.encodeWithSignature” genera el identificador de la función para el contrato a llamar.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract ContratoExterno {

    function saludo(string memory nombre) external pure returns(string memory) {
        return string(bytes.concat("Hola ", bytes(nombre)));
    }
}

contract Contrato {
    address direccionContratoExterno;

    constructor(address _dirContExterno) {
        direccionContratoExterno = _dirContExterno;
    }

    function saludar(string memory nombre) public payable returns(string memory) {
        /// .call devuelve (bool, bytes).
        /// Si la llamada ha sido una transacción ordinaria de ether, bool se debe comprobar si o si, bytes son datos
        /// sobre la transferencia.
        /// Si la llamada ha sido una llamada a otra función de otro contrato, bytes son los datos devueltos.
        (bool success, bytes memory data) = direccionContratoExterno.call(abi.encodeWithSignature("saludo(string)", nombre));
        require(success, "Fallo al llamar a ContratoExterno.saludo.");
        return string(data);
    }
}
```

#### 4.12 Eventos.

Los eventos, explicado sencillamente son registros de eventos que suceden en el contrato y se guardan en la blockchain, mayormente sirve para mostrar el historial de un evento en concreto, en la interfaz que interactúan con el contrato, o para actualizar los datos en la interfaz en vivo.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Contrato {
    address direccionContratoExterno;

    constructor(address _dirContExterno) {
        direccionContratoExterno = _dirContExterno;
    }

    // Definimos la estructura del evento ( los datos a loggear ).
    event Transferencia(address indexed destino, uint etherEnviado);

    function enviarEther() public {
        // Enviamos 1 ether del contrato al usuario que llama la función.
        (bool success, ) = msg.sender.call{value: 1 ether }("");
        require(success);

        // Emitimos el evento.
        emit Transferencia(msg.sender, 1 ether);
    }
}
```

La palabra “indexed” permite después ( en la interfaz ) filtrar por el atributo indexado.

### 5 Redes.

Debido a que el protocolo Ethereum Open Source, cualquier persona puede desplegar su propia implementación del protocolo, por lo tanto no solo existe la blockchain de Ethereum principal ( Mainnet ), sino que también hay muchísimas más con diferentes arquitecturas. Pero, como al fin y al cabo son implementaciones de Ethereum, estas siguen ejecutando una EVM, así que nuestro código en Solidity funcionará en cualquier red derivada de la EVM. Si quieres ver una lista de las redes más populares, hechale un vistazo a <https://chainlist.org/>

#### 5.1 Testnets.

También, cada red, suele tener una o varias “Testnets”, que son blockchains de prueba, donde los desarrolladores pueden desplegar sus contratos para probarlos.

Estas redes de prueba se suelen resetear cada cierto tiempo, y poseen **Faucets**, sistemas que provee a los desarrolladores del Ether necesarios para desplegar y interactuar con los contratos, en resumen es como tener dinero del Monopoly, no sirve de nada en el mundo real, pero sí para realizar pruebas con contratos.

## 6 Cartera / Cuenta.

Las carteras son aplicaciones que nos sirven para interactuar con nuestra cuenta de Ethereum.

Hay 2 tipos de cuentas:

- Cuentas de usuario: Manejada por un usuario.
- Cuentas de contrato: Manejada mediante código.

Gracias al sistema de gestión de cuentas de Ethereum, una cuenta de usuario, una vez creada es **ÚNICA PARA TODAS LAS BLOCKCHAINS**, además de que nos sirve para interactuar con todas las blockchains que funcionan sobre el protocolo Ethereum.

Así que en resumen, una vez que creamos nuestra cuenta, dicha clave privada nos servirá para todas las redes que ejecuten la EVM.

### 6.1 Metamask.

Metamask es una cartera que nos permite gestionar nuestra cuenta de Ethereum, pero que además existe en formato de extensión de navegador, esto hace más seguro y fácil desarrollar e interactuar con interfaces web que conectan con un contrato.

#### 6.1.1 Instalando Metamask.

Primero, antes de empezar a desarrollar contratos, vamos a necesitar una cartera y una cuenta, así que vamos a instalar meta mask desde la tienda de extensiones de nuestro navegador.

#### 6.1.2 Creando cuenta de metamask.

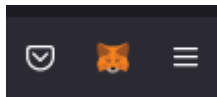
Vamos a la tienda de extensiones, y instalamos metamask, seguimos las instrucciones de creación de la cuenta, elegimos nueva cuenta, configuramos una contraseña para la cartera,

##### 6.1.2.1 Clave privada y Mnemonico.

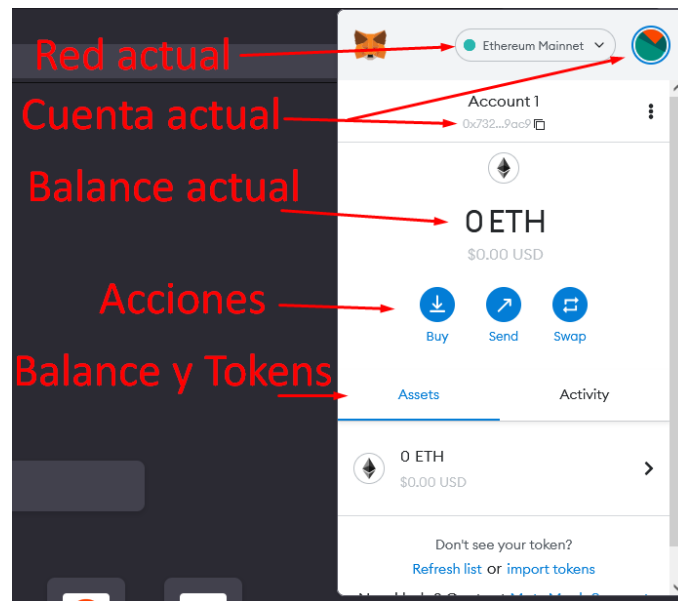
El mnemónico, o “frase de recuperación” es muy importante, ya que te permitirá recuperar tu cuenta en caso de perder la clave privada, desinstalar la extensión u otro evento similar.

#### 6.1.3 Interactuando con Metamask.

Una vez que hayamos acabado, reiniciamos el navegador, y veremos que tenemos un nuevo icono en la bandeja de extensiones.



Pinchamos encima del icono, y nos pedirá la clave, una vez introducida veremos el menú de metamask.

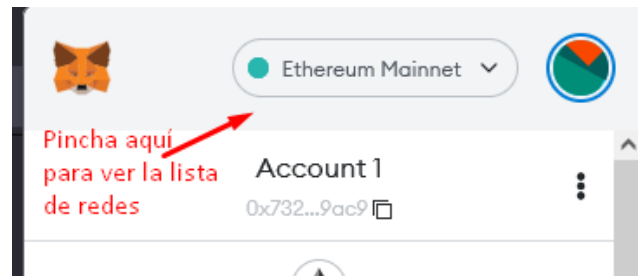


Estas son las opciones principales de Metamask, tomate tu tiempo familiarizado con las opciones y botones.

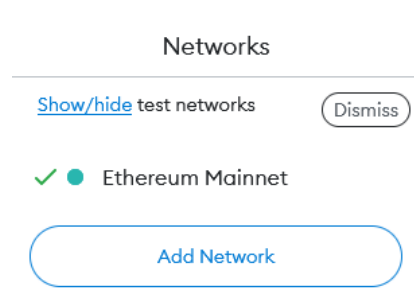
Como hemos dicho antes, una cartera es solo una aplicación que nos sirve para conectar y gestionar nuestras cuentas de Ethereum. Metamask soporta múltiples cuentas, y múltiples conexiones a diferentes redes.

#### 6.1.4 Habilitando Redes de prueba

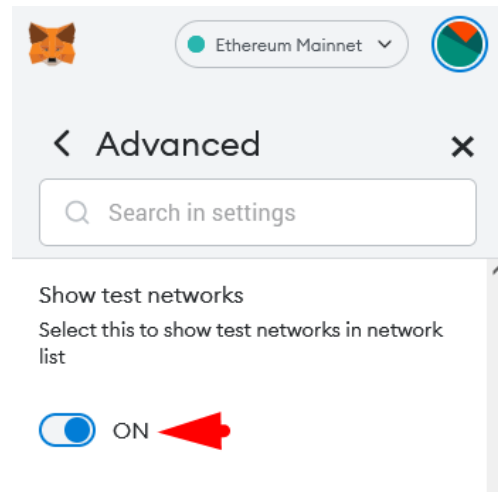
Por defecto las redes de prueba no se muestran en la lista de redes.



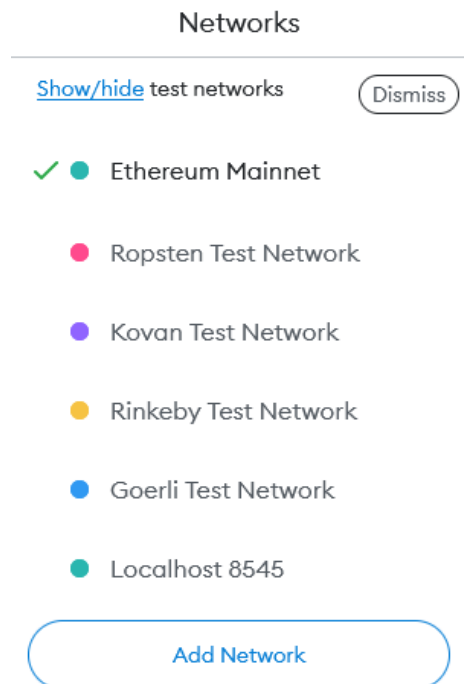
Por ello, vamos a pinchar encima de “Show/Hide test networks”.



Esto nos redirigirá a la configuración de metamask, deberemos habilitar esta opción.



Una vez hecho esto, ahora podremos ver las redes de pruebas de la Mainnet ( red principal de Ethereum ).



Esto nos servirá más adelante para desplegar en estas redes de prueba.

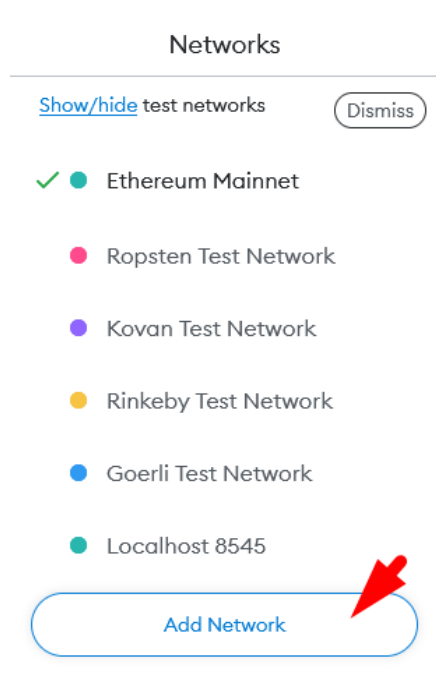


### 6.1.5 Añadimos una nueva red.

Como hemos dicho, hay más redes que la principal de Ethereum, nosotros podemos añadir una nueva red a Metamask desde la lista de redes.


#### 6.1.5.1 A mano.

En este ejemplo añadiremos la red principal de Polygon.



Y rellenamos con estos datos:

#### Networks > Add a network

 A malicious network provider can lie about the state of the blockchain and record your network activity. Only add custom networks you trust.

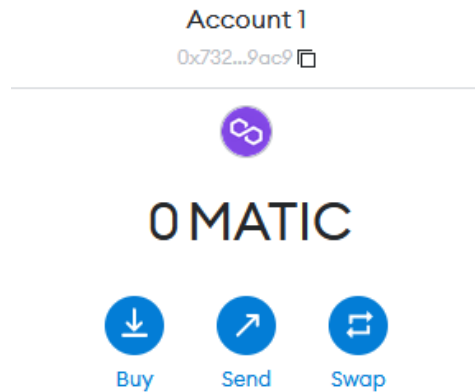
<b>Network Name</b>	<b>New RPC URL</b>
<input type="text" value="Polygon"/>	<input type="text" value="https://polygon-rpc.com/"/>
<b>Chain ID ⓘ</b>	<b>Currency Symbol</b>
<input type="text" value="137"/>	<input type="text" value="MATIC"/>
<small>Ticker symbol verification data is currently unavailable, make sure that the symbol you have entered is correct. It will impact the conversion rates that you see for this network</small>	
<b>Block Explorer URL (Optional)</b>	
<input type="text" value="https://polygonscan.com/"/>	
<input type="button" value="Cancel"/>	<input type="button" value="Save"/>

- Network Name: Nombre de la blockchain.
- RPC URL: Dirección URL RPC de uno de los nodos de la blockchain.
- Chain ID: Identificador de la blockchain ( se usa para firmar transacciones ).
- Current Symbol: Simbolo de la criptomoneda ( Ether propio a usar ).
- Block Explorer URL: URL de la página del explorador de bloques.

Estos datos se pueden conseguir desde aquí:

<https://docs.polygon.technology/docs/develop/metamask/config-polygon-on-metamask/>

Y ya está, ya podemos interactuar con la blockchain de Polygon usando nuestra cuenta.

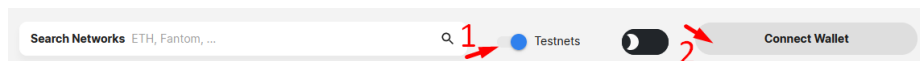


6.1.5.2 Desde un enlazador ( Automaticamente ).

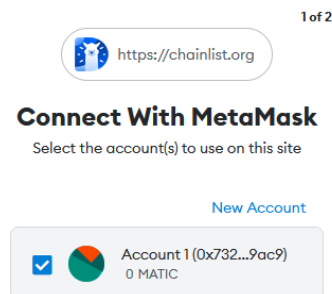
También podemos añadir una nueva red usando un enlazador, esta forma es menos segura porque la propia página web mandará los datos de configuración a Metamask.

En este ejemplo añadiremos la red de pruebas de Polygon.

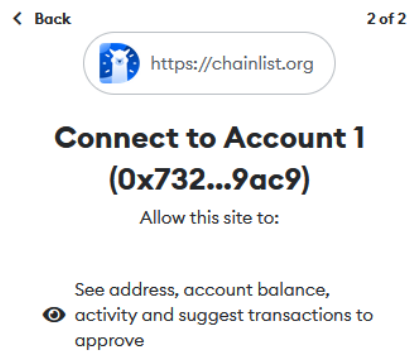
Vamos a <https://chainlist.org/> habilitamos que se muestren las redes de prueba y conectamos nuestra cartera.



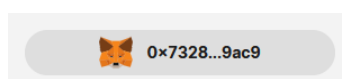
Seleccionamos nuestra cuenta, y le damos a siguiente.



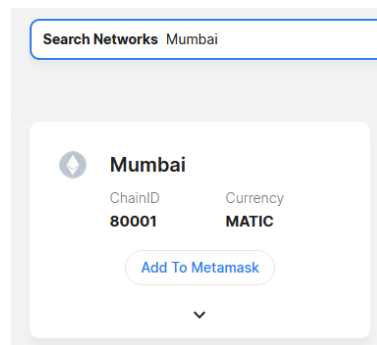
Damos permiso a la página para leer la dirección, balance, y actividad de la cuenta.



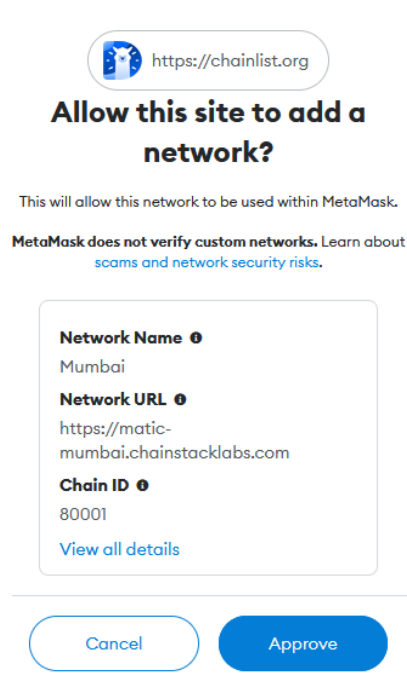
Una vez conectados, podremos ver nuestra dirección sobre el botón de conexión de la página.



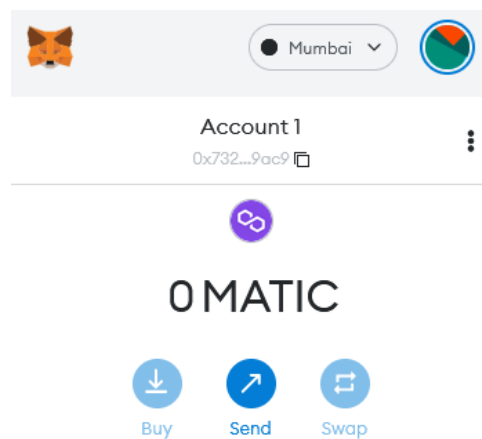
Una vez conectada la cuenta, buscamos en la lista la red de pruebas de Polygon, esta se llama MUMBAI.



Le damos a “Add to metamask”, y seguimos los pasos.



Y listo, ya tenemos esta red añadida también.



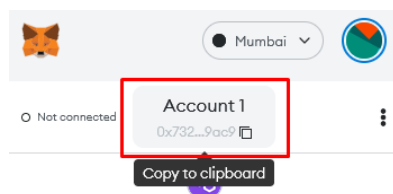
## 7 Faucets.

Como hemos mencionado, los Faucets son sistemas que proveen de Ether a las cuentas que lo soliciten. En general suele ser un minero o validador de la red, que distribuye el Ether conseguido a las personas que lo solicitan desde una interfaz web o similar.

No suele haber una “principal” sino varias, ya que cada minero de la red de pruebas puede hacer lo que quiera con su Ether.

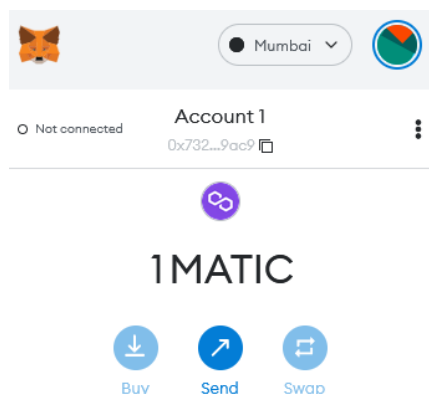
Vamos a buscar una faucet en google para MUMBAI, y vamos a conseguir un poco de dinero del monopoly, y fascinarnos al ver como ese 0 en nuestra cuenta sube un poco, aunque no sirva de nada hahaha!.

Si habeis buscando en google habeis encontrado <https://mumbaifaucet.com/>  
Así que vamos a copiar nuestra dirección de la cuenta, ( pinchar en el cuadro en rojo )



Y pegarlo, y pinchar encima de “Send Me MATIC”, la transacción ( dependiendo de la red ) puede tardar entre 5s hasta 1 min dependiendo de la carga de la red.

Y bueno, aquí está, la primera criptomoneda que recibimos.



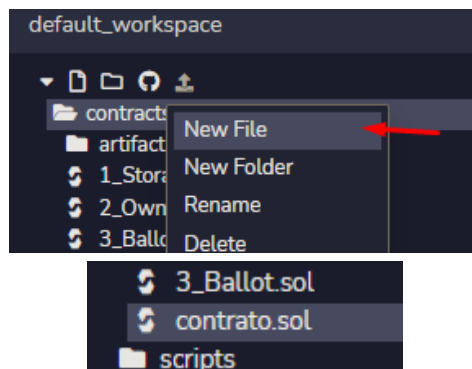
Ahora es tu turno, busca un poco de Ether para tu cuenta en la red Goerli.

## 8 RemixIDE.

RemixIDE ([remix.ethereum.org](https://remix.ethereum.org)) es un IDE para desarrollar smart contracts en Solidity en el navegador. Es muy potente, ya que permite compilar con diferentes compiladores, y por defecto los despliegues se hacen en una blockchain virtual, que no requiere ninguna cuenta ni ether para el gas, aunque también se puede conectar una cartera, y usar la cuenta y red conectada a la cartera.

### 8.1 Desarrollo de un smart contract.

Vamos a abrir RemixIDE, y vamos a la carpeta “contracts”, aquí encontraremos unos contratos de ejemplo, en esta carpeta vamos a crear un nuevo archivo llamado “contrato.sol”.



Dentro de este archivo, especificaremos la licencia, el pragma, y crearemos un nuevo contrato llamado “Calculadora”.

```
1 // SPDX-License-Identifier: GPL-3.0
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract Calculadora {
5
6 }
```

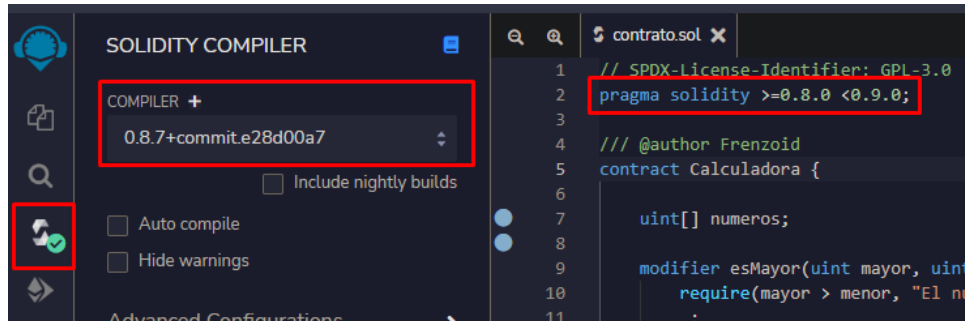
Este contrato tendrá 4 funciones, y un atributo de estado:

- Una realizará la suma de 2 números.
- Otra realizará la resta de 2 numeros solo si el primero es mayor que el segundo, realiza la comprobación usando un modificador
- Otra devuelve el valor de un número elevado a  $10^8$ .
- Tendremos un array de uints como variable de estado, y esta última función añadirá un número nuevo ( pasado por atributo ) al array, además de devolver el array actualizado.

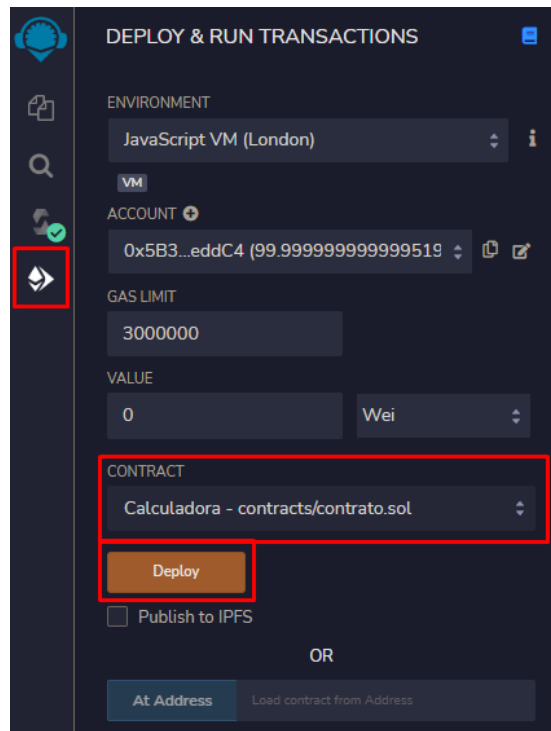
Prueba a hacerlo por tu cuenta, al guardar (ctrl + s) se compilará automáticamente, y te mostrará cualquier error que tengas. Si ves que no le das cabo, aquí tienes la solución: <https://gist.github.com/Frenzoid/4054c61da422704e3b22f4f01f341917>

## 8.2 Compilado y Despliegue.

Si tenemos problemas al compilar debido a la versión del pragma, podemos ir a la pestaña del compilador, y revisar que el compilador seleccionado es el mismo que el especificado en el pragma.

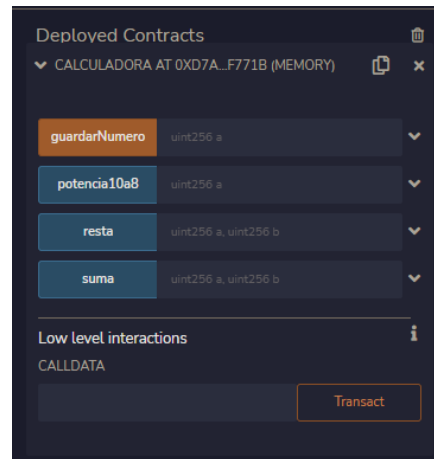


Y ahora para desplegar, vamos a la pestaña de despliegue, comprobamos que el archivo que queremos desplegar es el correcto, y le damos a “Deploy”.

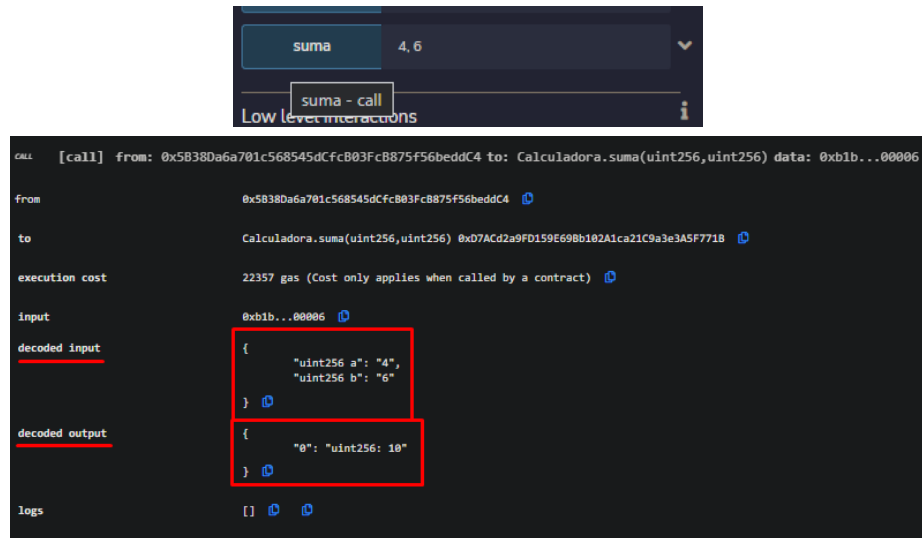


Nuestro contrato se desplegará a la blockchain virtual por defecto, por lo tanto no nos pedirá usar ninguna cartera / cuenta ni nos solicitará gastar ether en gas.

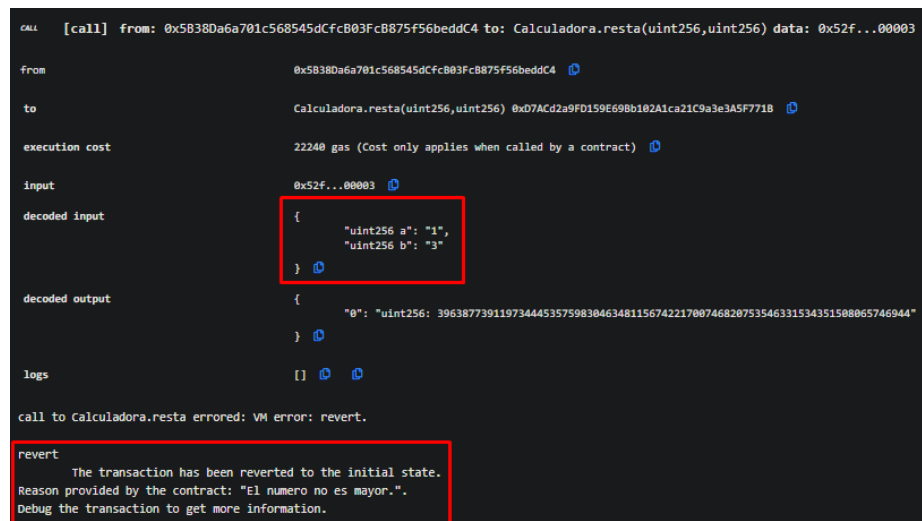
Justo debajo, en esta mismo apartado, una vez desplegado veremos una pequeña interfaz que nos servirá para probar los métodos creados.



En la consola podrás ver los detalles de cada transacción ( llamada a función ), los datos recibidos y los devueltos.

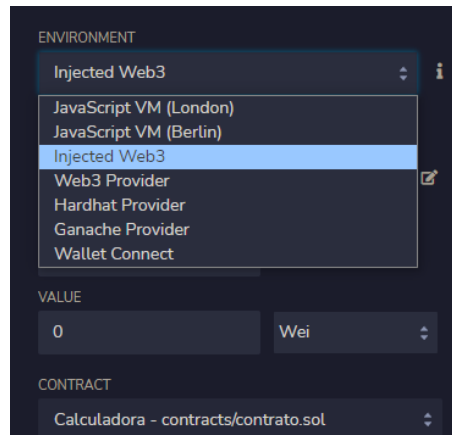


Vamos a ver qué pasa cuando en la función de resta, el primer número es menor que el segundo.



El texto “El número no es mayor”, es el motivo de fallo que hemos especificado en el require del modificador.

Vamos ahora a desplegar nuestro contrato en una blockchain real, para esto especificaremos que el entorno del despliegue sea “Injected web3”.



De normal, para que una interfaz se pueda conectar con un contrato, este requiere conocer la dirección del Nodo al cual conectarse (URL RPC), pero en este caso, debido a que la interfaz es una página web en nuestro navegador, podemos usar Metamask como intermediario al ser esta una extensión de nuestro navegador.

Metamask lo que hace como extensión es inyectar una configuración en nuestro navegador en forma de variable global en Javascript.

De hecho, si abrimos la consola de desarrollador del navegador, podrás ver una variable llamada “ethereum” definida.

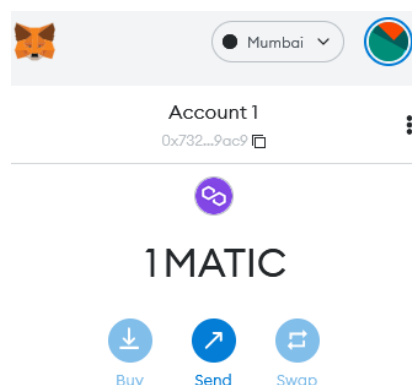
```
> ethereum
< ▶ Proxy {_events: {...}, _eventsCount: 1, _maxListeners: 100, _log: u, _state: {...}, ...}
> ethereum.isMetaMask
< true
```

Gracias a esto, cualquier página web que quiera interactuar con nuestra cuenta, o conectarse a un contrato desplegado en nuestra red lo puede hacer mediante esta configuración.

Esta manera es mucho más sencilla y segura en comparación con otras carteras, ya que no requiere del traspaso de la clave privada de la cuenta a la interfaz, ya que metamask lo gestiona todo, sin enviar nada sensible a la interfaz.

Volviendo al tema, al usar la opción de despliegue de remix “Injected web3”, la página de remix se sincronizará con nuestra cartera Metamask, deberás seguir unos pasos para permitir a remix acceder a tu cuenta, similar a cuando accedimos a chainlist.org en los apartados anteriores.

Antes de todo, verificamos que estamos conectados a la red de pruebas de Polygon ( mumbai ), donde tendremos nuestro MATIC de prueba.





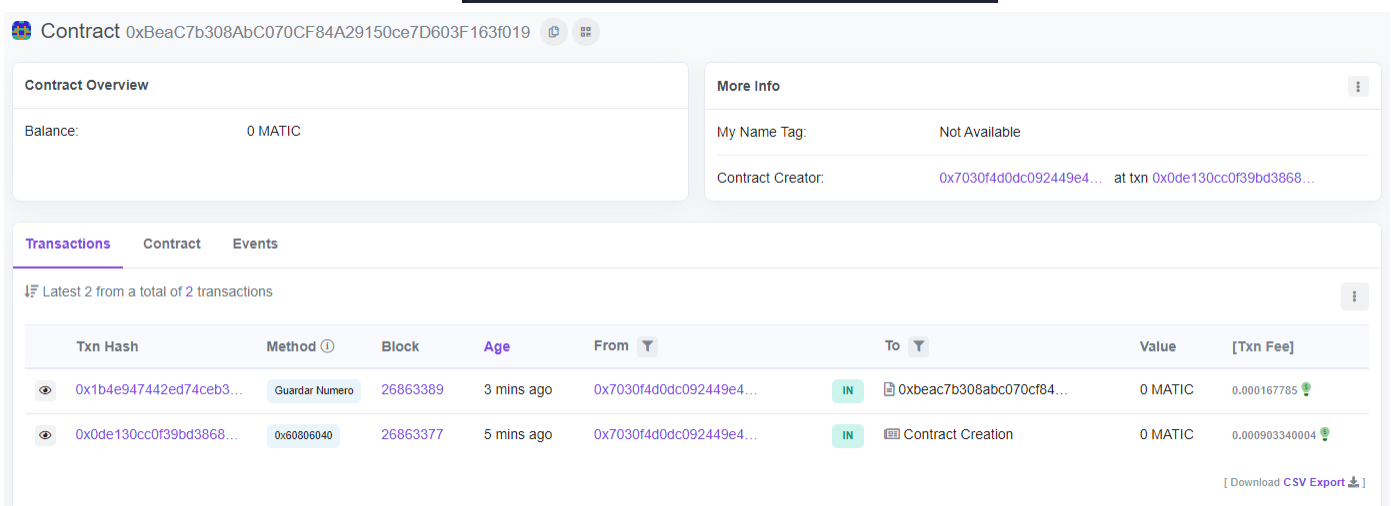
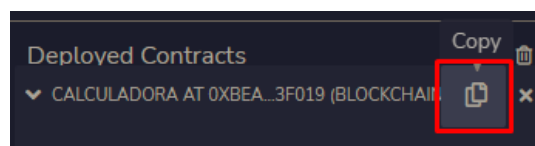
Después, pincharemos de nuevo encima del botón “Deploy”, esto generará una nueva transacción de creación de contrato que deberemos aceptar. Podremos ver gas estimado, y el coste total de la operación.



Le damos a confirmar, y esperaremos ya que el minado de la transacción puede tardar unos segundos. Una vez finalizado, tendremos nuevamente nuestra pequeña interfaz para interactuar con el contrato, en este caso, el método “guardarNumero” provocará una transacción ya que se modifica el estado de la blockchain, en cambio el resto de métodos al ser “pure”, solo se ejecuta los cálculos en un único Nodo (en este caso, el nodo que especificamos al añadir la red Mumbai) y devuelve los resultados.

Si ejecutamos la 4ta función, veremos que se requiere firmar y pagar el gas de la transacción, debido a que se cambia el valor del array de uints, éste al ser una variable de estado, su cambio actualiza el estado de la blockchain.

Si copiamos la dirección del contrato, podremos buscarlo en [mumbai.polygonscan.com](https://mumbai.polygonscan.com), el blockchain explorer de mumbai.



Txn Hash	Method	Block	Age	From	To	Value	[Txn Fee]
0x1b4e947442ed74ceb3...	Guardar Numero	26863389	3 mins ago	0x7030f4d0dc092449e4...	0xbeac7b308abc070cf84...	0 MATIC	0.000167785
0x0de130cc0f39bd3868...	0x60806040	26863377	5 mins ago	0x7030f4d0dc092449e4...	Contract Creation	0 MATIC	0.000903340004

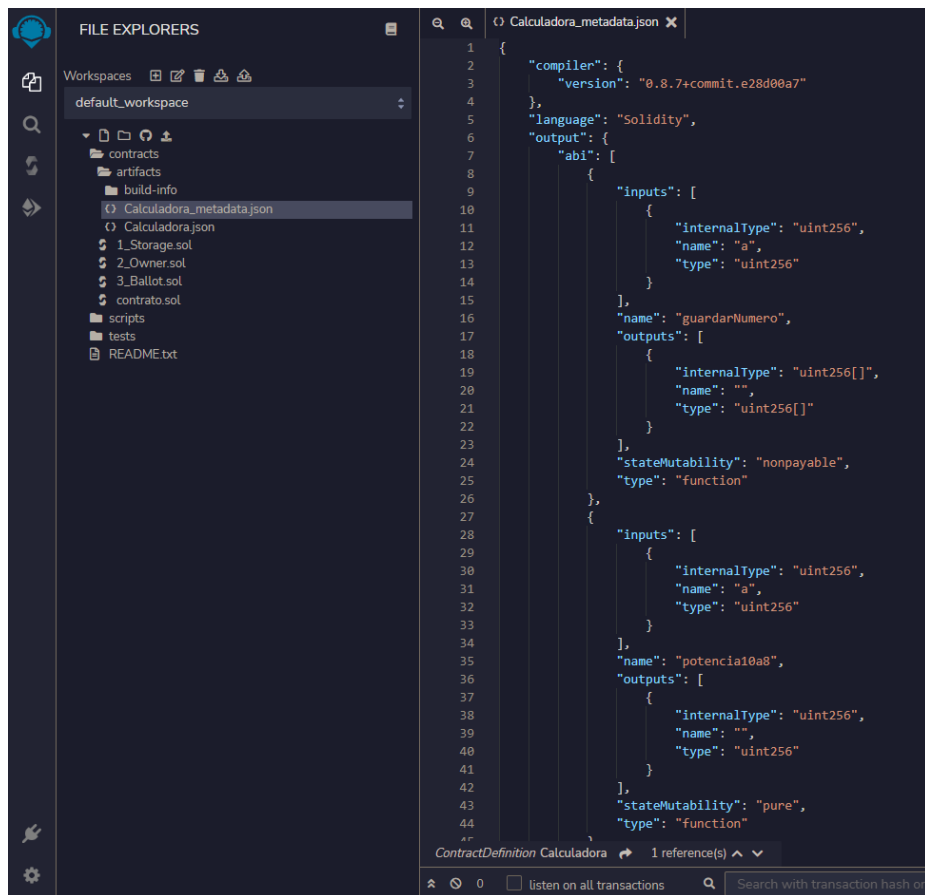
Los blockchain explorers son páginas web que permiten ver el estado de la blockchain, desde contratos hasta transacciones ordinarias.

### 8.3 ABI.

El ABI o Application Binary Interface, es un archivo ( normalmente en formato JSON ) que representa la interfaz de un smart contract, además de metadatos sobre el contrato.

Como ya sabemos, el identificador de una función se genera usando su nombre, parámetros, tipos recibidos, visibilidad, mutabilidad, valor a devolver, etc. El ABI sirve como plantilla del contrato para permitir a la interfaz saber que métodos y atributos tiene el contrato a que se está llamado, para poder generar de manera satisfactoria su identificador.

Este archivo se genera automáticamente cuando compilamos un contrato, normalmente ubicado dentro de una carpeta llamada “artifacts”. En remix lo puedes encontrar en “contracts/artifacts”,

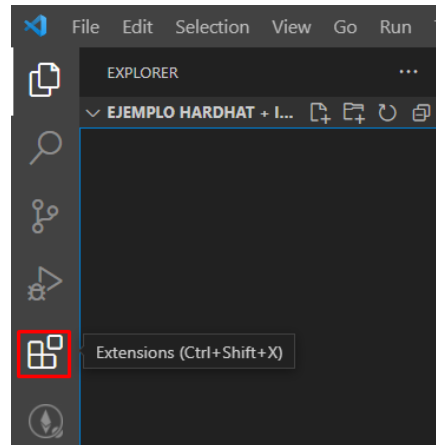


## 9 VSCode.

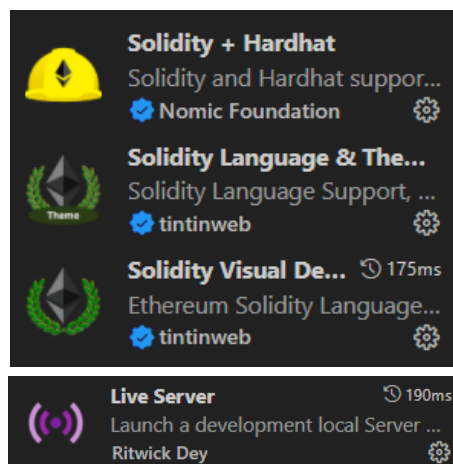
Visual Studio Code es un IDE de desarrollo en múltiples lenguajes, y de código abierto. Soporta extensiones creadas por la comunidad, y una alta personalización.

Usaremos VS Code como nuestro editor de código en Solidity de ahora en adelante (vete instalándolo), ya que además de programar en solidity, vamos a escribir código en HTML y JS.

Para esto vamos a necesitar instalar unas extensiones. Una vez instalado VSCode, vamos a la parte izquierda superior, y presionamos sobre este botón.



Este es el menú de extensiones, aquí buscaremos las siguientes extensiones a instalar:



Una vez instaladas estas extensiones, podremos desarrollar código Solidity en VSCode.

## 10 NodeJS.

Node es un entorno de ejecución de código javascript, este nos permitirá instalar paquetes y otros programas como Hardhat en nuestro sistema, además de ejecutar código en javascript. Node viene con un gestor de paquetes llamado NPM, y para poder gestionar cualquier módulo / paquete / librería necesitaremos crear un proyecto NPM primero.

Por ahora, procede a instalarlo para el siguiente paso, **procura que sea la versión LTS**.

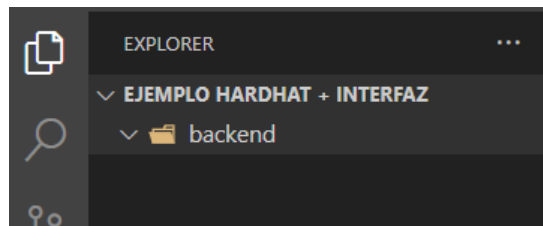
## 11 Hardhat.

Hardhat es un entorno de desarrollo de smart contracts desarrollado en JavaScript, este nos permite compilar código en Solidity, desplegar nodos con cuentas de prueba, probar nuestros contratos y desplegarlos. Es altamente configurable además de soportar tests unitarios para nuestros contratos.

### 11.1 Creando un Proyecto.

Para crear un proyecto en Node para desarrollar con Hardhat, lo primero que vamos a hacer es crear una carpeta vacía, y la abriremos con VSCode.

Dentro de VSCode, y dentro de nuestra carpeta vacía vamos a crear una nueva carpeta llamada “backend” que representará el entorno de desarrollo de hardhat.



Ahora, abriremos una nueva terminal “Ver > Terminal” o “Ctrl + Ñ”. En esta terminal, nos moveremos a la carpeta “backend” ejecutando el comando “cd backend”.

Aquí dentro, tendremos que inicializar un proyecto NPM que será la base de nuestro proyecto hardhat. Para eso ejecutaremos los siguientes comandos:

`npm init --yes` : Crea un nuevo proyecto npm en vacío.

`npx hardhat` : inicia el lanzador de hardhat, una vez ejecutado este comando seleccionamos la primera opción.

```
888      888      888      888      888
888      888      888      888      888
888      888      888      888      888
888888888888  8888b.  888d888 .d88888 88888b.  8888b.  888888
888      888      "88b 888p"  d88"  888 888  "88b      "88b 888
888      888 .d888888 888      888 888 888 888 .d888888 888
888      888 888 888 888      Y88b 888 888 888 888 888 Y88b.
888      888 "Y888888 888      "Y88888 888 888 "Y888888 "Y888

Welcome to Hardhat v2.9.9

? What do you want to do? ...
> Create a basic sample project
  Create an advanced sample project
  Create an advanced sample project that uses TypeScript
  Create an empty hardhat.config.js
  Quit
```

En la siguiente pantalla nos preguntará si está “la carpeta donde estamos” carpeta es donde queremos crear el proyecto.

```
✓ Hardhat project root: - G:\node_projects\labs\SOLIDITY\CDSCE
\ejemplo hardhat + interfaz\backend
```

La última pantalla nos pregunta si queremos crear un archivo gitignore, especificamos que Sí.

```
✓ Do you want to add a .gitignore? (Y/n) - y
```

Esto generará un proyecto hardhat vacío, podemos leer que hardhat nos recomienda instalar ciertas librerías para poder arrancar el proyecto localmente.

```
You need to install these dependencies to run the sample project:
npm install --save-dev "hardhat@^2.9.9" "@nomiclabs/hardhat-waffle@^2.0.0" "ethereu
m-waffle@^3.0.0" "chai@^4.2.0" "@nomiclabs/hardhat-ethers@^2.0.0" "ethers@^5.0.0"
```

Copiamos desde “npm ...” hasta el final, y lo pegamos en la consola, y lo ejecutamos.

Estos módulos son necesarios para poder ejecutar un proyecto hardhat.

Además instalaremos un paquete más llamado “dotenv” que nos permitirá ocultar nuestras claves privadas, en caso de subir nuestro proyecto a algún repositorio remoto.

Ejecutamos “npm install dotenv”.

## 11.2 Estructura del proyecto.

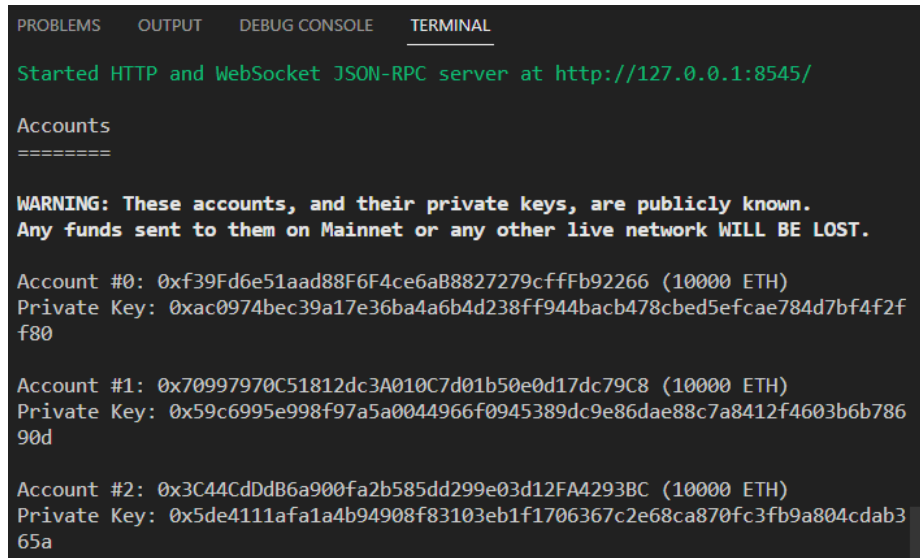
En nuestro proyecto tenemos varias carpetas y ficheros:

- contracts: Carpeta donde se ubicarán los contratos a desarrollar y compilar.
- node\_modules: Carpeta que alberga los módulos instalados.
- scripts: Scripts para configurar el despliegue de nuestro contrato.
- tests: Tests para nuestro contrato.
- .gitignore: Archivo de texto que alberga que ficheros habrá de ignorarse en caso de subir el código a algún repositorio Git.
- hardhat.config.js: Configuración de Hardhat.
- package-lock: Lista de versiones y dependencias de los módulos.
- package.json: Metadatos sobre el proyecto, y scripts.
- README.md: Archivo “Leeme!” del proyecto.

### 11.3 Nodo local.

Con hardhat podemos desplegar nuestro propio nodo privado local, que nos servirá para poder ver cómo se altera la blockchain ante un cambio, y desplegar nuestros contratos.

Vamos a abrir otra terminal, y dentro de la carpeta “backend” ejecutaremos el comando “npx hardhat node”, esto arrancará un nodo ethereum privado en nuestro ordenador, que solo podrá ser accedido localmente.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Started HTTP and WebSocket JSON-RPC server at http://127.0.0.1:8545/

Accounts
=====

WARNING: These accounts, and their private keys, are publicly known.
Any funds sent to them on Mainnet or any other live network WILL BE LOST.

Account #0: 0xf39Fd6e51aad88F6F4ce6aB8827279cFfB92266 (10000 ETH)
Private Key: 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2f
f80

Account #1: 0x70997970C51812dc3A010C7d01b50e0d17dc79C8 (10000 ETH)
Private Key: 0x59c6995e998f97a5a0044966f0945389dc9e86dae88c7a8412f4603b6b786
90d

Account #2: 0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC (10000 ETH)
Private Key: 0x5de4111afa1a4b94908f83103eb1f1706367c2e68ca870fc3fb9a804cdab3
65a
```

Lo primero que veremos será una lista de cuentas, con sus direcciones y su clave privada. Estas cuentas son cuentas de desarrollo, cualquier persona tiene acceso a estas claves privadas así que no las uses para hacer nada en una red real.

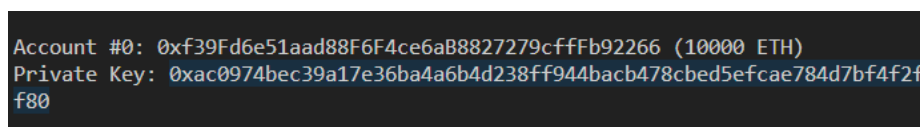
### 11.4 Cuentas de desarrollo.

Las cuentas de desarrollo de Hardhat son cuentas que tienen 10k Ether cada una, y sirven para probar contratos y transacciones sin tener que estar limitado a una testnet.

#### 11.4.1 Añadimos la cuenta de desarrollo de Hardhat a Metamask.

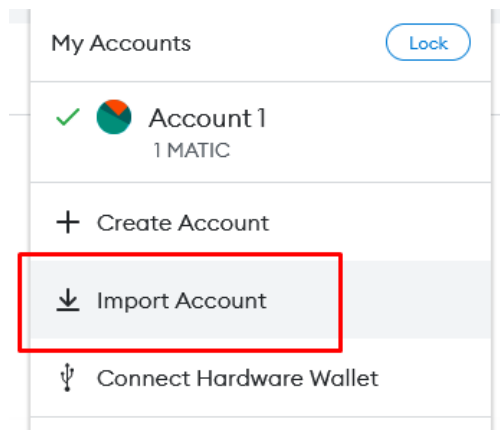
Recuerda que cualquier persona tiene acceso a las claves privadas de estas cuentas, es como si todo el mundo supiera sus contraseñas, así que procura no hacer nada serio con estas, y solo usarlas con el nodo local, dicho esto, vamos a importar la primera cuenta que vemos a nuestra cartera metamask.

Para esto, primero copiamos la clave privada de la primera cuenta.



```
Account #0: 0xf39Fd6e51aad88F6F4ce6aB8827279cFfB92266 (10000 ETH)
Private Key: 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2f
f80
```

Vamos a metamask > cuentas > importar cuentas.



En el desplegable de “tipo” seleccionamos “clave privada”.

## Import Account

Imported accounts will not be associated with your originally created MetaMask account Secret Recovery Phrase. Learn more about imported accounts [here](#)

Select Type

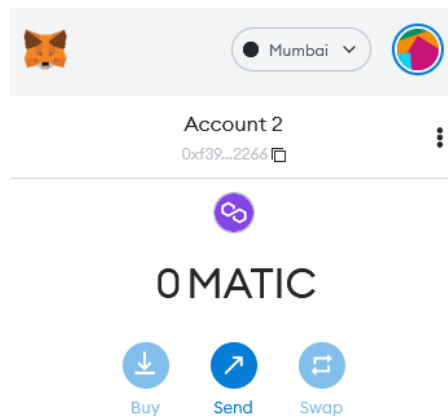
Private Key ▾

Enter your private key string here:

Cancel

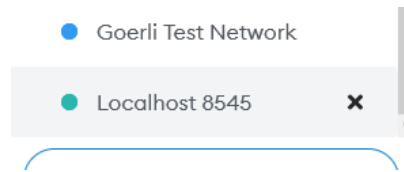
Import

Y lo pegamos aquí, y lo importamos.

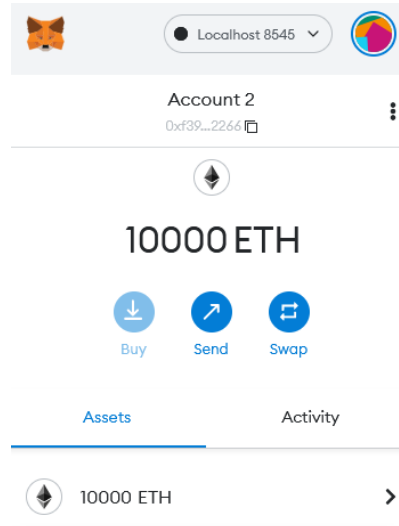


Y ya está, la cuenta está importada! Pero espera un segundo, no tenemos ether, esto se debe a que estamos en la red equivocada. Una de las redes de prueba que vienen por defecto en metamask es Localhost, que es donde se está ejecutando nuestro nodo.

Cambiamos de red...



Y revisamos nuevamente el balance



Ahora sí que podemos ver los 10k de la cuenta.

### 11.5 Configuración de hardhat.

El archivo de configuración “hardhat.config.js” representa la configuración y tareas del entorno de desarrollo de hardhat que usamos actualmente. Por defecto, la única configuración que este archivo tiene es la versión del compilador en uso, y la tarea por defecto es “account” que nos lista las cuentas actuales..

```
hardhat.config.js U X
backend > hardhat.config.js > ...
1  require("@nomiclabs/hardhat-waffle");
2
3  // This is a sample Hardhat task. To learn how to create your own go to
4  // https://hardhat.org/guides/create-task.html
5  task("accounts", "Prints the list of accounts", async (taskArgs, hre) => {
6    const accounts = await hre.ethers.getSigners();
7
8    for (const account of accounts) {
9      console.log(account.address);
10    }
11  });
12
13  // You need to export an object to set up your config
14  // Go to https://hardhat.org/config/ to learn more
15
16  /**
17   * @type import('hardhat/config').HardhatUserConfig
18   */
19  module.exports = {
20    solidity: "0.8.4",
21  };
22
```

Las tareas son scripts predefinidos por el usuario, para facilitar el desarrollo. Nosotros no haremos mucho uso de las tareas, solo de la configuración.



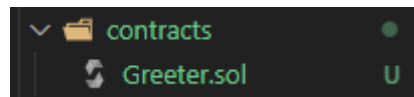
En la configuración, vamos a añadir 2 redes, para poder desplegar nuestro contrato. Uno local, haciendo referencia al propio nodo de Hardhat, y otro a Mumbai.

Primero vamos a configurar la red local, y más adelante veremos como desplegar en una red de prueba. Para esto solo tenemos que modificar el JSON de la configuración, especificando la red “localhost” con la URL del nodo “http://localhost:8545”.

```
module.exports = {
  solidity: "0.8.4",
  networks: {
    localhost: {
      url: "http://localhost:8545",
    },
  },
};
```

### 11.6 Smart contract de ejemplo.

Si ahora vamos a la carpeta “contracts” tendremos un contrato por defecto, generado por el proyecto.



Este contrato almacena un string, el constructor recibe el valor por defecto, además de tener 2 funciones, una para obtener el string ( variable greeting ), y otra para setearlo.

```
//SPDX-License-Identifier: Unlicense
pragma solidity ^0.8.0;

import "hardhat/console.sol";

UnitTest stub | dependencies | uml | draw.io
contract Greeter {
  string private greeting;

  ftrace
  constructor(string memory _greeting!) {
    console.log("Deploying a Greeter with greeting:", _greeting!);
    greeting = _greeting!;
  }

  ftrace | funcSig
  function greet() public view returns (string memory) {
    return greeting;
  }

  ftrace | funcSig
  function setGreeting(string memory _greeting!) public {
    console.log("Changing greeting from '%s' to '%s'", greeting, _greeting!);
    greeting = _greeting!;
  }
}
```

Además vemos que el contrato usa de una librería de hardhat llamada console, esta librería nos permitirá realizar logs a la consola de nuestro nodo de hardhat.

### 11.7 Compilado y Despliegue.

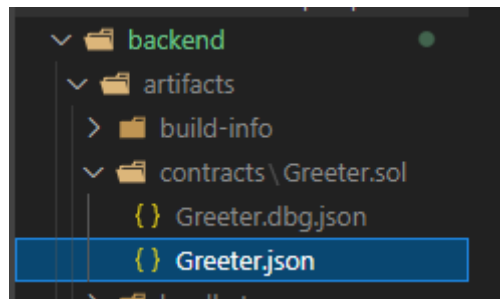
Para compilar un contrato, solo tenemos que ejecutar el comando “npx hardhat compile”, en caso de no tener descargada la version del compilador especificado en la configuración, hardhat la descargará automaticamente. Mientras que la version especificada en la configuración sea la misma que la del pragma de nuestro contrato no deberia haber problemas en el compilado.

El ABI del contrato por defecto se guardará en la carpeta “artifacts”, pero esta ubicacion se puede cambiar desde la configuración.

Procedemos a compilar el contrato.

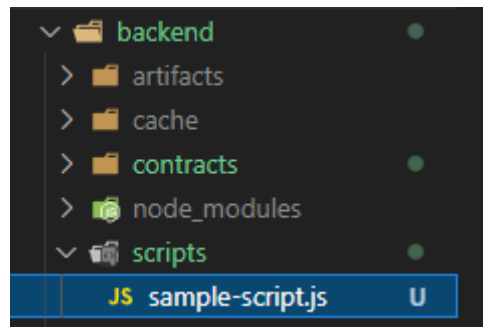
```
PS G:\node_projects\labs\SOLIDITY\CDSCE\greeter\backend> npx hardhat compile
Compiled 2 Solidity files successfully
PS G:\node_projects\labs\SOLIDITY\CDSCE\greeter\backend> 
```

Una vez compilado, podemos revisar el ABI generado.



### 11.7.1 Despliegue en el nodo local.

Los scripts sirven para configurar el despliegue de un contrato. Vamos a la carpeta “scripts” y abrimos el archivo “sample-script.js”.



Y vemos el siguiente código en JS.

```
1 // Importa hardhat, y sus funcionalidades
2 const hre = require("hardhat");
3
4 // Funcion principal de despliegue.
5 async function main() {
6
7     // Obtiene el contrato usando la factoria de contratos de Hardhat.
8     // Hardhat automaticamente detecta que contratos hay creatos en la
9     // carpeta "contracts" y los compila en una factoria de contratos.
10    const Greeter = await hre.ethers.getContractFactory("Greeter");
11
12    // Desplegamos el contrato, pasamos al constructor un valor.
13    // devuelve una instancia de la transaccion de creacion del contrato.
14    const greeterInstance = await Greeter.deploy("Hola mundo!");
15
16    // Esperamos a que la transaccion se complete.
17    await greeterInstance.deployed();
18
19    // Una vez desplegado, mostramos por consola la dirección del contrato.
20    console.log("Greeter deployed to:", greeterInstance.address);
21 }
22
23
24 // Ejecutamos la funcion principal.
25 main()
26   .then(() => process.exit(0))
27   .catch((error) => {
28     console.error(error);
29     process.exit(1);
30   });
31
```

Este código en JS se ejecuta por hardhat, el código en la foto difiere un poco del original, solo se ha cambiado los comentarios que venian por defecto, y el valor por defecto del parametro pasado por constructor.

Para desplegar el contrato en nuestro nodo local, ejecutaremos el siguiente comando:

```
“npx hardhat run .\scripts\sample-script.js --network localhost”
```

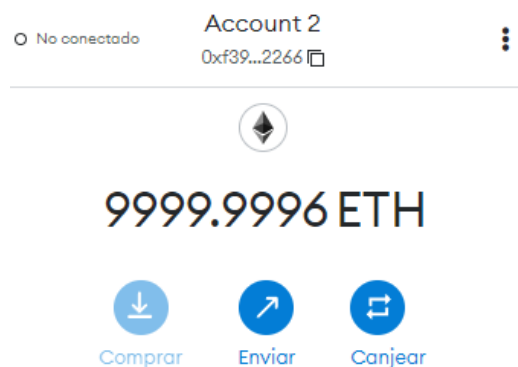
```
PS G:\node_projects\labs\SOLIDITY\CDSCE\greeter\backend> npx hardhat run .\scripts\sample-script.js --network localhost
Greeter deployed to: 0x5FbDB2315678afecb367f032d93F642f64180aa3
```

Y así de sencillo, ya tendríamos nuestro contrato desplegado en nuestro nodo local de hardhat.

```
Contract deployment: Greeter
Contract address: 0x5fbdb2315678afecb367f032d93f642f64180aa3
Transaction: 0x3a45b220c8e428a5187a8f91ed6b602645048a0c88e
90796043f873bec7582f2
From: 0xf39fd6e51aad88f6f4ce6ab8827279cfff92266
Value: 0 ETH
Gas used: 496978 of 496978
Block #1: 0xbe46f8b050c06087b2c88180b35fced4bf5e42c60b9ee11ccd1897ec885f00

console.log:
Deploying a Greeter with greeting: Hola mundo!
```

La cuenta usada para desplegar es la primera cuenta de las cuentas de prueba de hardhat, de hecho, si revisamos el saldo de esta cuenta, veremos que se ha reducido.



En caso de no haber previamente compilado el contrato, hardhat automáticamente compilará el contrato antes del ejecutar el script para desplegar el contrato.

#### 11.7.2 Despliegue en una red de pruebas.

Para desplegar en una red de pruebas, como Mumbai, vamos a tener que añadir dicha red a la configuración, además de la clave privada de alguna cuenta que posea ether en dicha red.

Para esto, lo primero que vamos a hacer será obtener el enlace RPC del nodo de Mumbai. Es el mismo paso que realizamos a la hora de añadir la red a Metamask.

Vamos a [chainlist.org](https://chainlist.org), buscamos mumbai, desplegamos la lista de nodos, y copiamos el que menor latencia tenga.

A screenshot of the Chainlist.org website. The search bar at the top contains 'mumbai'. Below the search bar, there's a card for 'Mumbai' with 'ChainID 80001' and 'Currency MATIC'. A 'Connect Wallet' button is visible. Below this card is a red rectangular box containing an upward arrow. At the bottom, there's a table titled 'Mumbai RPC URL List' with columns: 'RPC Server Address', 'Height', 'Latency', 'Score', and an action column. The first row is highlighted with a red box around the URL 'https://matic-mumbai.chainstacklabs.com'.

Mumbai RPC URL List				
RPC Server Address	Height	Latency	Score	
https://matic-mumbai.chainstacklabs.com	26944414	0.070s	●	Connect Wallet
https://rpc-mumbai.maticvigil.com	26944414	0.123s	●	Connect Wallet

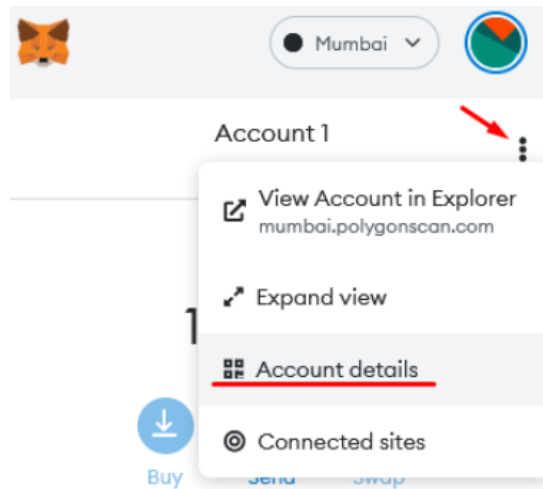
Una vez copiado, vamos a la configuración de hardhat, y creamos una nueva red, la llamamos “mumbai”, y especificamos la URL del nodo.

```
module.exports = {
  solidity: "0.8.4",
  networks: {
    localhost: {
      url: "http://localhost:8545",
    },
    mumbai: {
      url: "https://matic-mumbai.chainstacklabs.com",
    },
  },
};
```

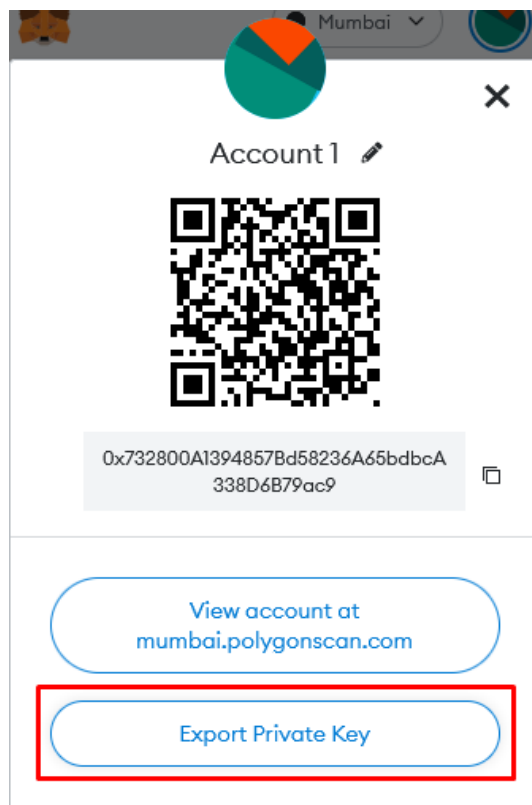
Además, tenemos que especificar la / las cuenta/s que se usarán para esta red, para ello creamos un nuevo parametro llamado “accounts”, que contendrá un array con las claves privadas.

```
module.exports = {
  solidity: "0.8.4",
  networks: {
    localhost: {
      url: "http://localhost:8545",
    },
    mumbai: {
      url: "https://matic-mumbai.chainstacklabs.com",
      accounts: ["clave privada de nuestra cuenta"],
    },
  },
};
```

Para obtener la clave privada de nuestra cuenta, vamos a metamask y en los detalles de nuestra cuenta la podemos obtener.



Y exportamos nuestra clave privada, esto nos mostrará nuestra clave privada, y la podremos copiar y pegar en la configuración.



Obviamente, no es seguro tener una contraseña o clave privada escrita en un documento de configuración que al subirse a un repositorio sería accesible por cualquiera, así que usaremos la librería dotenv para ocultarla.

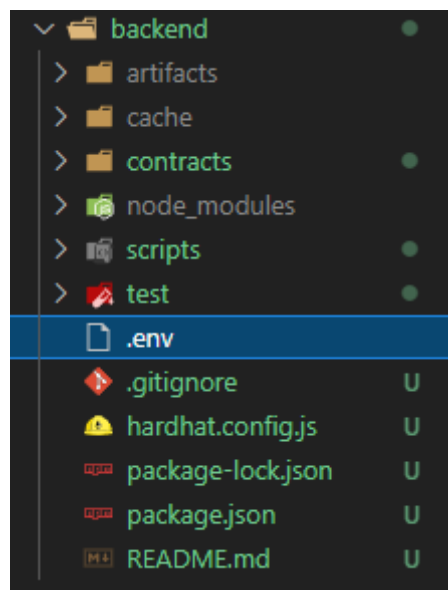
Para ello importamos dotenv al principio de la configuración de hardhat.

```
1  require("@nomiclabs/hardhat-waffle");  
2  require("dotenv").config({ path: ".env" });  
3  
4  
5  // This is a sample Hardhat task. To learn how to create your own go to  
6  // https://hardhat.org/guides/create-task.html  
7  task("accounts", "Prints the list of accounts", async (taskArgs, hre) => {  
8    const accounts = await hre.ethers.getSigners();  
9  
10   for (const account of accounts) {  
11     console.log(account.address);  
12   }  
13 });
```

Dotenv lo que hace es generar y simular variables de entorno ( variables de sistema ) desde una lista de variables llamada “.env”.

De esta manera, en nuestra configuración solo tendremos que llamar a la variable de entorno que almacena nuestra clave privada exportada desde un archivo “.env”. Nuestro archivo “.env” no se deberá subir a ningún lado ( verifica que “.env” está añadido a la lista de exclusión de git ( revisa .gitignore )).

Entonces, para esto, primero crearemos un archivo llamado “.env” al mismo nivel que el proyecto.



La estructura de este archivo es de “clave=valor” por cada línea, donde la “clave” será el nombre de la variable de entorno donde se exportará el “valor”.

En nuestro caso llamaremos a la variable “CLAVE\_PRIVADA”.

```
.env  
1  CLAVE_PRIVADA=aquíescribestuclaveprivada|
```

Y en nuestra configuración de hardhat, solo tendremos que escribir “process.env.CLAVE\_PRIVADA”.

```
21 module.exports = {
22   solidity: "0.8.4",
23   networks: {
24     localhost: {
25       url: "http://localhost:8545",
26     },
27     mumbai: {
28       url: "https://matic-mumbai.chainstacklabs.com",
29       accounts: [process.env.CLAVE_PRIVADA],
30     },
31   },
32 };
33
```

La configuración debería quedar así:

<https://gist.github.com/Frenzoid/98a848ec413e3d667ecb1759301e019>

Ahora si, vamos a probar a desplegar nuestro contrato en Mumbai. Ejecutamos el comando: “npx hardhat run scripts/sample-script.js --network mumbai”.

```
PS G:\node_projects\labs\SOLIDITY\CDSCE\greeter\backend> npx hardhat run .\scripts\sample-script.js --network mumbai
Greeter deployed to: 0xb56e487c1f4a1ef3d6f1C5686f33924003160Ff0
```

Y ya está. Ahora si buscamos en polyscan por nuestro contrato, lo veremos desplegado.

Contract 0xb56e487c1f4a1ef3d6f1C5686f33924003160Ff0

Contract Overview

Balance: 0 MATIC

More Info

My Name Tag: Not Available

Contract Creator: 0x732800a1394857bd58... at txn 0xf8c858571b914fc2b63...

Transactions

ERC-20 Token Txns

Contract

Events

Latest 1 from a total of 1 transactions

Txn Hash	Method	Block	Age	From	To	Value	[Txn Fee]
<a href="#">0xf8c858571b914fc2b63...</a>	<a href="#">0x60806040</a>	<a href="#">26944666</a>	1 min ago	<a href="#">0x732800a1394857bd58...</a>	<a href="#">IN</a> Contract Creation	0 MATIC	0.001591903081

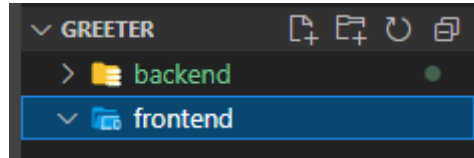
[ Download CSV Export ]



## 12 Creando una página web para nuestro smart contract.

Para poder interactuar con el contrato vamos a necesitar una interfaz, para ello vamos a crear una página web, y usaremos la librería Ethers para poder interactuar de forma cómoda con metamask y nuestro contrato.

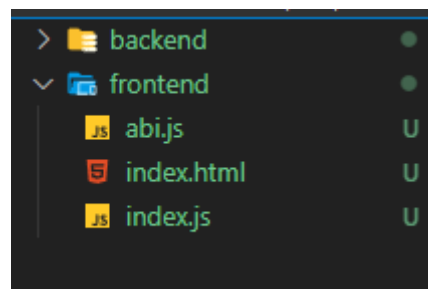
Empezaremos por crear una carpeta llamada “frontend” al mismo nivel que la carpeta “backend”.



Dentro de esta carpeta vamos a crear 3 archivos, uno llamado “index.html” que contendrá nuestro código en html, que es lo que renderiza el navegador.

Y otro archivo llamado “index.js” que contendrá nuestro código en Javascript, que dará interactividad a nuestro código html.

Además, vamos a crear otro archivo llamado abi.js, donde copiaremos nuestro ABI de la carpeta “backend/artifacts/contracts/Greeter.sol/Greeter.json”.



### 12.1 Importando el ABI.

Dentro del archivo “abi.js” vamos a crear una nueva variable constante llamada “abiJSON” a la cual le asignaremos el json abi de “backend/artifacts/contracts/Greeter.sol/Greeter.json”.

```
frontend > .js abi.js > [e] abiJSON > "abi"
1  const abiJSON = {
2    "_format": "hh-sol-artifact-1",
3    "contractName": "Greeter",
4    "sourceName": "contracts/Greeter.sol",
5    "abi": [
6      {
7        "inputs": [
8          {
9            "internalType": "string",
10           "name": "_greeting",
11           "type": "string"
12         }
13       ],
14       "stateMutability": "nonpayable",
15       "type": "constructor"
16     },
17     {
18       "inputs": [],
19       "stateMutability": "payable",
20       "type": "function"
21     }
22   ]
23 }
```

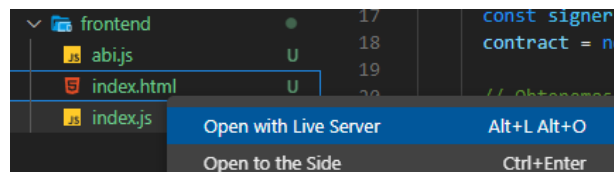
## 12.2 Pagina web.

Para la página web, como no vamos a complicarnos mucho porque este informe es solo smart contracts y no sobre JS o HTML, vamos a crear una página web sencilla.

Dentro de index.html escribiremos el siguiente código.

```
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4
5    </head>
6    <body>
7      <h3>SALUDO</h3>
8      <input type="text"></input>
9      <button>Set Saludo</button>
10   </body>
```

Abrimos el archivo con “click derecho > open with Live Server”.



Esto renderizará lo siguiente:

**SALUDO**

Además, importaremos el ABI y nuestro archivo JS que de momento está vacío ( el orden de importación importa ).

```
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4
5    </head>
6    <body>
7
8      <h3>SALUDO</h3>
9      <input type="text"></input>
10     <button>Set Saludo</button>
11
12     <!-- ABI -->
13     <script src="./abi.js" type="application/javascript"></script>
14
15     <!-- Código JS -->
16     <script src="./index.js" type="application/javascript"></script>
17   </body>
```

### 12.2.1 Bootstrap CSS.

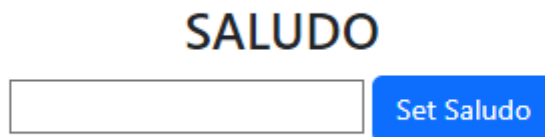
Vamos a darle un poquito de estilo usando bootstrap, para esto importaremos las librerías de bootstrap dentro de nuestro html.

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <!-- Bootstrap CSS -->
5     <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0-beta1/dist/c
6   </head>
7   <body>
8
9     <h3>SALUDO</h3>
10    <input type="text"></input>
11    <button>Set Saludo</button>
12
13    <!-- Bootstrap JS -->
14    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0-beta1/dist/
15
16    <!-- ABI -->
17    <script src="./abi.js" type="application/javascript"></script>
18
19    <!-- Código JS -->
20    <script src="./index.js" type="application/javascript"></script>
21  </body>
```

Y ahora, vamos a darle unas clases a nuestra interfaz.

```
<body class="container text-center mt-5">
  <h3 class="label">SALUDO</h3>
  <input type="text"></input>
  <button class="btn btn-primary">Set Saludo</button>
  <!-- Bootstrap JS -->
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0-beta1/d
  <!-- ABI -->
  <script src="./abi.js" type="application/javascript"></script>
  <!-- Código JS -->
  <script src="./index.js" type="application/javascript"></script>
</body>
```

Estas clases le darán el siguiente estilo a nuestra página web.



El “SALUDO” se susituirá mediante JS por el valor de “greeting” del contrato.

### 12.3 Código en JS.

En el archivo “index.js” escribiremos nuestro código, tendremos que hacer 3 cosas.

- Funcion principal.
  - Acceder a Metamask.
  - Obtener el firmante “acceso a la cuenta”.
  - Crear una instancia del contrato.
- Crear funcion para obtener “greeting”.
- Crear funcion para setear “greeting”.

### 12.4 EthersJS.

Ethers es una libreria que nos permitirá interactuar de manera comoda con metamask, primero lo importaremos en el HTML, y lo usaremos desde el archivo JS.

```
6 </head>
7 <body class="container text-center mt-5">
8
9   <h3 class="label">SALUDO</h3>
10  <input type="text"></input>
11  <button class="btn btn-primary">Set Saludo</button>
12
13  <!-- Bootstrap JS -->
14  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0-beta1/dist/js/bootstrap.bundle.min.js" integrity="
15
16  <!-- ABI -->
17  <script src="./abi.js" type="application/javascript"></script>
18
19  <!-- EthersJS -->
20  <script src="https://cdn.ethers.io/lib/ethers-5.2.umd.min.js" type="application/javascript"></script>
21
22  <!-- Código JS -->
23  <script src="./index.js" type="application/javascript"></script>
24 </body>
```

Y en el código JS, crearemos 2 variables globales, una para el proveedor ( acceso al nodo ) y otro para almacenar la instancia del contrato cuando la creemos.

```
1 // Variables globales.
2 let provider;
3 let contract;
```

Cuando la página haya acabado de cargar, vamos a realizar 2 cosas.

#### 12.4.1 Conexión con la cartera.

Este código nos permitirá obtener un proveedor, una manera de conectarnos a nuestra cartera.

```
// Conectamos con la cartera.
provider = new ethers.providers.Web3Provider(window.ethereum);
```

#### 12.4.2 Conexión con el contrato.

Este código nos permite conectar con una o varias de las cuentas de nuestra cartera. además de obtener una instancia de nuestro contrato.

```
// Accedemos a la cuenta.
await provider.send("eth_requestAccounts");

// Obtenemos el firmante, y obtenemos una instancia del contrato
// usando la direccion, el abi, y el firmante.
const signer = provider.getSigner();
contract = new ethers.Contract("0xb56e487c1f4a1ef3d6f1c5686f33924003160ff0", abiJSON.abi, signer);
```

La instancia de nuestro contrato nos permitirá interactuar con el contrato.

## 12.5 Funcion principal.

El código en JS en nuestra funcion principal quedará de la siguiente manera.

```
1 // Variables globales.
2 let provider;
3 let contract;
4
5 // Cuando cargue la página, conectamos con la cartera.
6 window.addEventListener("load", connectWallet);
7
8 async function connectWallet() {
9     // Conectamos con la cartera.
10    provider = new ethers.providers.Web3Provider(window.ethereum);
11
12    // Accedemos a la cuenta.
13    await provider.send("eth_requestAccounts");
14
15    // Obtenemos el firmante, y obtenemos una instancia del contrato
16    // usando la direccion, el abi, y el firmante.
17    const signer = provider.getSigner();
18    contract = new ethers.Contract("0xb56e487c1f4a1ef3d6f1c5686f33924003160ff0", abiJSON.abi, signer);
19 }
20
```

## 12.6 Creamos las funciones.

### 12.6.1 Obtener greeting.

Esta función obtendrá el valor de “greeting” de nuestro contrato, y lo setteará como valor a nuestro `<h3>`, para poder obtener este `<h3>` en nuestro código, tendremos que darle una id a nuestro `<h3>`.

```
<h3 id="greeter" class="label" >SALUDO</h3>
<input type="text"></input>
```

Y en el código JS haremos lo siguiente, obtendremos el valor de “greeting” del contrato, y lo settearemos como valor para nuestro `<h3 id="greeter">`.

```
// Funciones principales.
async function getGreeter() {
    const greeter = await contract.greet();
    document.getElementById("greeter").innerText = greeter;
}
```

La llamada a “contract.greet()” es la funcion de nuestro contrato.

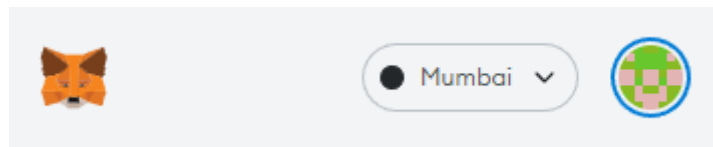
```
fttrace | funcSig
function greet() public view returns (string memory) {
    return greeting;
}
```

Así que lo devuelto, será el valor de greeting.

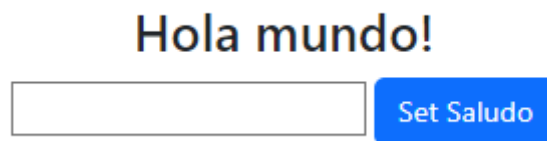
Llamamos a esta funcion en la funcion que se ejecuta al cargar la página:

```
5 // Cuando cargue la página, conectamos con la cartera.
6 window.addEventListener("load", connectWallet);
7
8 async function connectWallet() {
9     // Conectamos con la cartera.
10    provider = new ethers.providers.Web3Provider(window.ethereum);
11
12    // Accedemos a la cuenta.
13    await provider.send("eth_requestAccounts");
14
15    // Obtenemos el firmante, y obtenemos una instancia del contrato
16    // usando la dirección, el abi, y el firmante.
17    const signer = provider.getSigner();
18    contract = new ethers.Contract("0xb56e487c1f4a1ef3d6f1c5686f33924003160ff0", abiJSON.abi, signer);
19
20    // Obtenemos el greeter.
21    getGreeter();
22 }
```

Nos aseguramos de que estamos en la red de Mumbai.



Y recargamos la página, y voala, ya podemos observar el valor del smart contract en nuestra página web.



#### 12.6.2 Settear greeter.

Haremos lo mismo para el input, le damos una id.

```
<input id="input" type="text"></input>
```

Y en nuestro código JS, obtenemos el valor del input, y se lo mandamos por parametro a la funcion “setGreeter” de nuestro contrato.

```
async function setGreeter() {
    const greeter = document.getElementById("input").value;
    const tx = await contract.setGreeting(greeter);
    await tx.wait();
    getGreeter();
}
```

Lo que devolverá la función será la transacción, nos esperamos a que acabe, y llamamos a la función que hemos creado antes para actualizar el valor de la página.

Solo nos falta una cosa, hacer que cuando apretamos el boton de “Set Saludo” se llame a esta función. Para ello, desde nuestro HTML podemos definir un evento para el boton, que cuando se haga click se llame a esta función.

```
<button onclick="setGreeter()" class="btn btn-primary">Set Saludo</button>
```

Así, cuando apretamos el boto, se obtendrá el valor del input, y se mandará al contrato.

Vamos a probarlo.

# Hola mundo!

Set Saludo

Le damos a set saludo, nos aparecerá la transacción por confirmar en metamask. Y esperamos un minuto o menos a que finalice la transacción.

# Hola perola!!!

Set Saludo

## 13 Código final.

El código de todo este documento lo podeis encontrar en el siguiente repositorio:

<https://github.com/Frenzoid/labs/tree/master/SOLIDITY/CDSCE>

## 14 Consejos.

### 14.1 Estandares de Contratos.

Openzeppelin provee de varios estandares para contratos, que sirven para crear facilmente tokens, o otro tipo de valores o sistemas.

### 14.2 Verificación de contratos.

Una vez subido el contrato a una blockchain, es posible verificar el código compilado con el explorador de la blockchain. Esto sirve para dar una capa extra de fiabilidad, al publicar el código fuente del contrato en el explorador.

### 14.3 Optimización de gas.

Debido a que la EVM, las variables de Storage se guardan en ranura de 32 bytes, al declarar las variables de estado se pueden empaquetar en ranuras de 32 bytes para optimizar el acceso a dicho pack, ya que cuando se lee o guarda una variable de estado, se lee la ranura entera de 32 bytes, y se accede a cada posición individualmente.

Empaquetar tus variables significa que empaquetas o juntas variables de menor tamaño para que en conjunto formen 32 bytes.

Por ejemplo, puede empaquetar 32 uint8 en una ranura de almacenamiento, pero para que eso suceda, es importante que se declaren consecutivamente porque el orden de declaración de las variables importa en Solidity.

Un ejemplo:

<pre>uint8 num1; uint256 num2; uint8 num3; uint8 num4; uint8 num5;</pre>	<pre>uint8 num1; uint8 num3; uint8 num4; uint8 num5; uint256 num2;</pre>
--	--

El segundo es mejor porque en el segundo ejemplo, el compilador de solidity pondrá todos los uint8 en una ranura de almacenamiento, pero en el primer caso pondrá uint8 num1 en una ranura, pero la siguiente variable es un uint256 que en sí mismo requiere 32 bytes (  $256/8 \text{ bits} = 32 \text{ bytes}$  ), por lo que no se puede colocar en la misma ranura de almacenamiento que uint8 num1, por lo que ahora necesitará otra ranura de almacenamiento.

Después de eso, uint8 num3, num4, num5 se colocarán en otra ranura de almacenamiento.

Así que el segundo ejemplo está más optimizado, ya que requiere 2 ranuras de almacenamiento en comparación con el primer ejemplo que requiere 3 ranuras de almacenamiento. Esto hace que el despliegue sea más barato, ya que no se gastará tanto gas, ya que no se reservará más memoria en la blockchain para estas variables.

También es importante tener en cuenta que los elementos en memory y los calldata no se pueden empaquetar y no están optimizados por el compilador de solidity.

#### 14.4 Contratos Actualizables.

Debido a que un contrato puede llamar a otros contratos, existe la posibilidad de implementar el patron de desarrollo Proxy.

## 15 Fin.

Este documento, como he mencionado al principio, se ha enfocado más en explicar Solidity que en otra cosa, se que muchas cosas del documento están explicadas rápidamente, esto es debido a que el documento es una introducción al desarrollo de smart contracts. Si quieres aprender más, te recomiendo echarle un vistazo a [learnweb3.io](https://learnweb3.io). Esta página tiene cursos gratuitos para aprender de una forma escalada a desarrollar DAPPs.

Si encuentras alguna errata, o tienes alguna sugerencia para editar el documento, te invito a que me mandes un email a: [frenzoid@pm.me](mailto:frenzoid@pm.me).

Espero que te haya servido, un saludo!