Primer Smart Contract

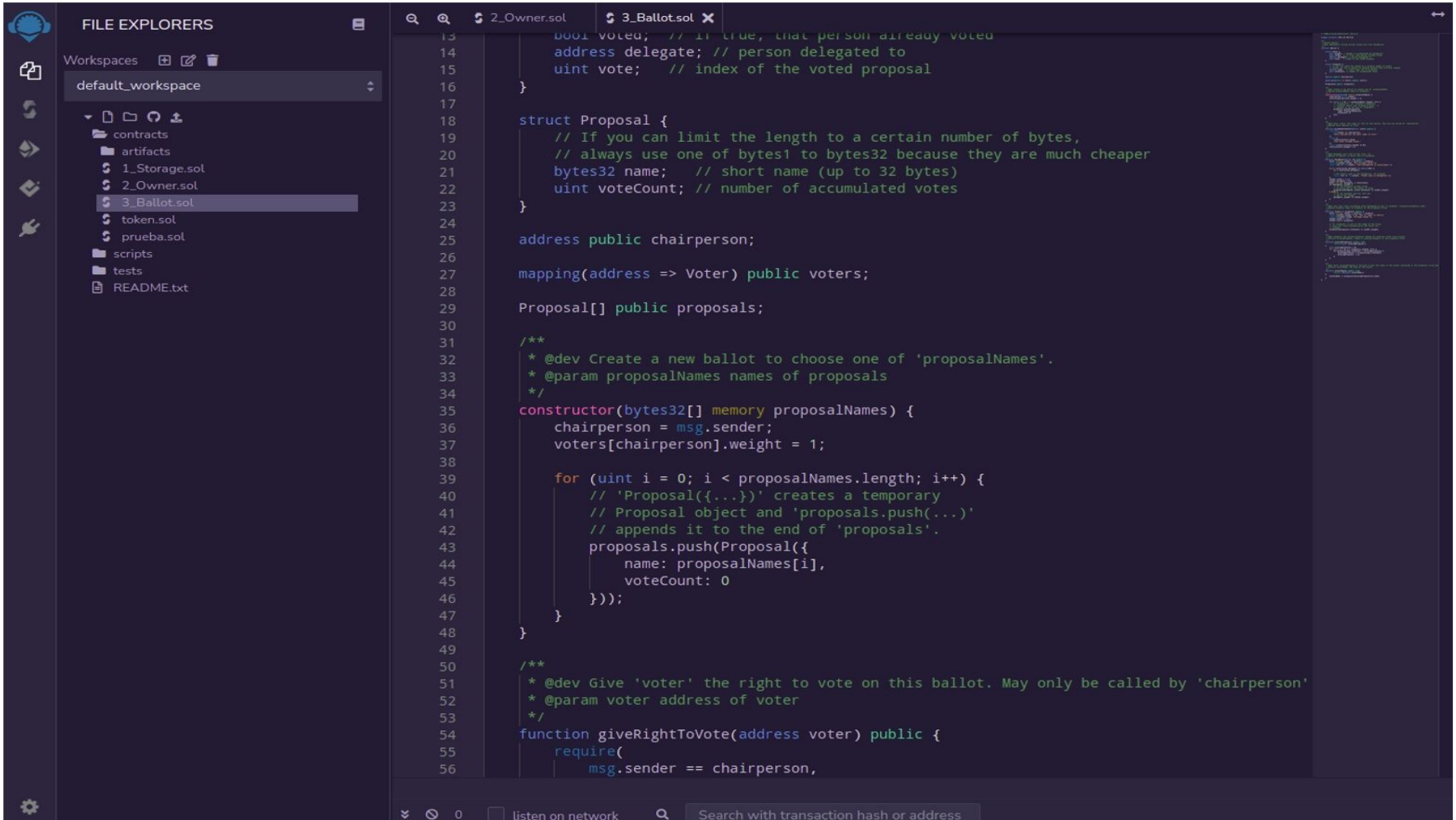
Con Solidity!

https://www.elladodelmal.com/2021/12/blockchain-smartcontrats-primer-smart.html

Primer SmartContract con Solidity

Como herramienta de desarrollo vamos a utilizar Remix, es un IDE diseñado para Solidity y que además corre sobre la web, con lo que no hace falta instalarlo, también ofrece numerosas utilidades para testear los contratos y “subirlos” a la Blockchain.

https://remix.ethereum.org/


FILE EXPLORERS
Workspaces
default_workspace
contracts
artifacts
1_Storage.sol
2_Owner.sol
3_Ballot.sol
token.sol
prueba.sol
scripts
tests
README.txt
2_Owner.sol
3_Ballot.sol
bool voted; // if true, that person already voted
address delegate; // person delegated to
uint vote;   // index of the voted proposal
}
struct Proposal {
// If you can limit the length to a certain number of bytes,
// always use one of bytes1 to bytes32 because they are much cheaper
bytes32 name;   // short name (up to 32 bytes)
uint voteCount; // number of accumulated votes
}
address public chairperson;
mapping(address => Voter) public voters;
Proposal[] public proposals;
/**
* @dev Create a new ballot to choose one of 'proposalNames'.
* @param proposalNames names of proposals
*/
constructor(bytes32[] memory proposalNames) {
chairperson = msg.sender;
voters[chairperson].weight = 1;
for (uint i = 0; i < proposalNames.length; i++) {
// 'Proposal({...})' creates a temporary
// Proposal object and 'proposals.push(...)'
// appends it to the end of 'proposals'.
proposals.push(Proposal({
name: proposalNames[i],
voteCount: 0
}));
}
}
/**
* @dev Give 'voter' the right to vote on this ballot. May only be called by 'chairperson'
* @param voter address of voter
*/
function giveRightToVote(address voter) public {
require(
msg.sender == chairperson,
0
listen on network
Search with transaction hash or address

¿Cómo se compila?

Se Utiliza la EVM(Ethereum virtual machine) de Ethereum que viene instalado por defecto en la app de Remix.

Es parecido a Java y orientado a objetos, en este caso cada objeto es un contrato diferente mezclado como si fueran clases estáticas de java.

1: nvim +

```
" Press ? for help

.. (up a dir)
</nft-marketplace/
▸ .git/
▸ .next/
▸ artifacts/
▸ cache/
▾ contracts/
    Greeter.sol
    NFT.sol
    NFTMarket.sol
▸ node_modules/
▸ public/
▸ scripts/
▸ src/
▸ test/
  .config
  .eslintrc.json
  .gitignore
  .key
  .prettierignore
  .prettierrc.js
  .solhint.json
  config.js
  hardhat.config.js
  next.config.js
  package-lock.json
  package.json
  postcss.config.js
```

```
1  // SPDX-License-Identifier: MIT OR Apache-2.0
2  pragma solidity ^0.8.3;
1
2  import "@openzeppelin/contracts/utils/Counters.sol";
3  import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
4  import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
5
6  import "hardhat/console.sol";
7
8  contract NFT is ERC721URIStorage {
9      using Counters for Counters.Counter;
10     Counters.Counter private _tokenIds;
11     address contractAddress;
12
13     constructor(address marketplaceAddress) ERC721("Metaverse", "METT") {
14         contractAddress = marketplaceAddress;
15     }
16
17     function createToken(string memory tokenURI) public returns (uint256) {
18         _tokenIds.increment();
19         uint256 newItemId = _tokenIds.current();
20
21         _mint(msg.sender, newItemId);
22         _setTokenURI(newItemId, tokenURI);
23         setApprovalForAll(contractAddress, true);
24         return newItemId;
25     }
26 }
```

Los archivos en Solidity deben tener la extensión .sol, la EVM no requiere que tus proyectos tengan una estructura exacta, en este caso se parece a lenguajes como JavaScript. Pero por comodidad vamos a seguir las convenciones establecidas por la comunidad.

Un proyecto cualquiera de Web3 que requiera Smart Contracts se suele estructurar como se ve en el slide anterior.

Como se ve los contratos que se usan en esta webapp se guardan bajo el directorio de contratos. No se suelen usar más porque es muy raro que para una aplicación se utilicen más de 3 o 4 contratos.

1: nvim +

```
" Press ? for help

.. (up a dir)
</nft-marketplace/
▸ .git/
▸ .next/
▸ artifacts/
▸ cache/
▾ contracts/
    Greeter.sol
    NFT.sol
    NFTMarket.sol
▸ node_modules/
▸ public/
▸ scripts/
▸ src/
▸ test/
  .config
  .eslintrc.json
  .gitignore
  .key
  .prettierignore
  .prettierrc.js
  .solhint.json
  config.js
  hardhat.config.js
  next.config.js
  package-lock.json
  package.json
  postcss.config.js
```

```
1  // SPDX-License-Identifier: MIT OR Apache-2.0
2  pragma solidity ^0.8.3;
1
2  import "@openzeppelin/contracts/utils/Counters.sol";
3  import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
4  import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
5
6  import "hardhat/console.sol";
7
8  contract NFT is ERC721URIStorage {
9      using Counters for Counters.Counter;
10     Counters.Counter private _tokenIds;
11     address contractAddress;
12
13     constructor(address marketplaceAddress) ERC721("Metaverse", "METT") {
14         contractAddress = marketplaceAddress;
15     }
16
17     function createToken(string memory tokenURI) public returns (uint256) {
18         _tokenIds.increment();
19         uint256 newItemId = _tokenIds.current();
20
21         _mint(msg.sender, newItemId);
22         _setTokenURI(newItemId, tokenURI);
23         setApprovalForAll(contractAddress, true);
24         return newItemId;
25     }
26 }
```

Primer paso: versión de Solidity y definir el contrato

La unidad mínima de ejecución de código en Solidity es un contrato, al igual que en Java si no tienes una clase Main no puedes ejecutar código. Para crear este contrato crearemos un archivo llamado FirstContract.sol.

Lo primero que tenemos que hacer es definir la versión de Solidity que utilizaremos, en nuestro caso usaremos la última versión la 0.8.x que ha traído numerosas funcionalidades al lenguaje. La versión se define así:

```
// SPDX-License-Identifier: MIT OR Apache-2.0

pragma solidity ^0.8.3;
```

Primer paso: versión de Solidity y definir el contrato

Después definiremos contrato y le daremos nombre:

// SPDX-License-Identifier: MIT OR Apache-2.0

pragma solidity ^0.8.3;

contract FirstContract {}

En nuestro ejemplo queremos que el contrato tenga algunos datos del dueño del blog como su edad, su nombre, su dirección de eth (muy importante) y los datos del blog propiamente, que será una lista con todas las URLs de sus posts.

Para ello tenemos que conocer las variables y tipos de datos básicos que podemos utilizar.

Variables y tipos de datos básicos

Una cosa muy importante que debemos tener en cuenta es que todo lo que se guarde como estado se guardará para siempre en la Blockchain y eso tiene un coste. En Solidity existen los siguientes tipos de datos básicos:

- ★ Bool: Son expresiones booleanas.
- ★ Strings: Listas de caracteres, palabras, frases. Es decir, cadenas de texto.
- ★ Integers: números enteros que se pueden definir de diferentes maneras en función de lo que necesitemos.
- ★ Address: que es un tipo de dato muy especial en el que se guardan direcciones de ethereum ya sean cuentas de wallets o direcciones de otros contratos.
- ★ Arrays: En Solidity los arrays pueden ser de cualquier tipo pero solo pueden contener elementos de un mismo tipo. En versiones anteriores de Solidity solo se permitía usar arrays con una longitud fija, lo que cambiaba el paradigma un poco, pero a partir de la versión 0.8.x se pueden utilizar arrays dinámicos.

Variables y tipos de datos básicos

En el código se definirían escribiendo primero la palabra clave del tipo de dato después el nombre y la asignación.

```
// SPDX-License-Identifier: MIT OR Apache-2.0

pragma solidity ^0.8.3;

contract FirstContract {

// a boolean is defined like bool isAdult = ownerAge <= 18

address public owner; // In the form 0x923c89…

string public ownerName = "Chiquito de la Calzada";

uint256 public ownerAge = 23;

// An array that will contain all the posts urls

string[] public posts;

}
```

Propiedades y métodos

También queremos unas cuantas funciones para actualizar el dueño del contrato, añadir o eliminar Posts, obtenerlos.

Los contratos en Solidity funcionan de manera muy parecida a las clases en programación orientada a objetos. Tienen propiedades y métodos pero una vez subidos a la Blockchain solo se pueden acceder a los métodos.

Al igual que en las clases de OOP creamos un constructor que servirá para inicializar el objeto con unas ciertas propiedades.

Los métodos (también llamados funciones) y el constructor se definen de la siguiente manera:

```
// SPDX-License-Identifier: MIT OR Apache-2.0
pragma solidity ^0.8.3;

contract FirstContract {
    // code that goes before …
    //Contract contstructor
constructor(address _firstOwner){
    owner = _firstOwner;
     }
  //Definition of contract methods
    function setOwnerState(
        address _newOwner,
        string memory  _ownerName,
        uint256  _age
    ) public {
        require(msg.sender == owner, "Error, you are not the owner");
        owner = _newOwner;
        ownerName = _ownerName;
        ownerAge = _age;
    }
}
```

Como se ve en el código he introducido muchas cosas nuevas, vamos a explicarlas una a una:

Para declarar la función se utiliza la palabra reservada **function** y como en la mayoría de los lenguajes se definen los parámetros como si fueran variables pero sin asignación. Se dieron cuenta de que usó la palabra **memory** al declarar el string _ownerName, eso lo veremos más adelante.


FirsContract.sol
function setOwnerState(
address _newOwner,
string memory _ownerName,
uint256 _age
) public {
...
}

Modificador de acceso

Acto seguido podemos ver que está escrita la palabra reservada “**public**”. Esta palabra es un modificador de acceso, estos se pueden establecer en las variables globales si queremos pero en las funciones son obligatorias. Por defecto en Solidity existen los siguientes:

Public: Establece la función como pública y a está podrá acceder cualquier persona que posea el contrato y también podrá ser usada en los contratos que hereden de este.

Private: La función solo puede ser llamada por el mismo contrato pero no será heredable.

Internal: Es como public solo que deja de ser accesible por las personas.

External: La función será accesible por otros contratos, pero ni se heredará ni será accesible por el contrato actual.

Es una función en la que evaluamos condiciones y si no se cumplen rompen el flujo de ejecución del Smart Contract y devuelve al cliente un error con el string que pasemos. En este caso accedemos a quién llama la función y si esta persona no es el dueño la función no se ejecutará.

Toda función que pueda ser llamada desde fuera tiene una variable accesible llamada “msg”. Esta tendría las siguientes propiedades:

msg.sender: La dirección de eth de quién llamo la función.

msg.value: La cantidad de eth que se ha enviado con esa función.

Otras más que por ahora no voy a nombrar que no son importantes en este momento.

Las Naming Conventions son simplemente las convenciones establecidas a la hora de nombrar variables, funciones…
En Solidity son las siguientes:

- Uso de barra baja para variables locales, sirve para distinguir fácilmente el ámbito en el que puede actuar una variable, si la variable es global pues no hay que hacer nada, pero si por el contrario la variable tiene un ámbito local en una función por ejemplo se utiliza la barra baja para nombrarla “_variable”.

- CamelCase: La convención para nombrar variables, funciones, objetos… Es la misma que en JavaScript, el uso de la nomenclatura CamelCase. La cual simplemente es “esMayorDeEdad”.

Condicionales y bucles

Ahora podemos cambiar el dueño del contrato y sus propiedades siempre que queramos. Vamos a implementar unas funciones que nos permitan añadir posts, eliminarlos y leerlos utilizando condicionales y bucles.

- Condicionales: En Solidity funcionan igual que en otros lenguajes pero las expresiones de evaluación no, además como todo tiene que correr dentro de un nodo de la Blockchain que solo maneja números y direcciones no existen como tal strings puros, entonces manejar strings se vuelve peliagudo a veces como cuando hay que compararlos. Lo que debemos hacer es hashearlos mediante la función keccka256 y entonces compararlos.

- Bucles: Funcionan igual que en lenguajes como Java solo que en Solidity solo tenemos la versión normal del “for” en la que hay que hacer i++ y demás.

Vamos a crear ahora unas cuantas funciones para añadir, eliminar posts.

_existsUrl: Esta función se va a encargar de decirnos si ya se guardó antes una URL.

```
// SPDX-License-Identifier: MIT OR Apach.0
pragma solidity ^0.8.3;

contract FirstContract {
    // code that goes before …
    //Contract contstructor
    constructor(address _firstOwner){
    owner = _firstOwner;
    }
    function addPostUrl(string memory _url) public {
        require(msg.sender == owner, "Error, you are not the owner");
        require(!_existsUrl(_url), "The url already exists");
        posts.push(_url);
    }
}
```

- Primero el parámetro que se le pasa es un string memory porque no queremos que eth guarde esos datos en la Blockchain.

- Después le asignamos el modificador private dado que no queremos que la función sea accedida desde fuera de nuestro contrato. Si se fijan tenemos unos cuantos modificadores más, eso lo explicaremos en otra clase.

- Posteriormente le decimos que retorne un valor booleano. Iniciamos un bucle tal y como vemos en el código para iterar los posts que están dentro del array. Dentro del bucle evaluamos que si existe retorna falso. Puede parecer un poco raro como se hace la comparación entre “_url” y “posts[i]” pero Solidity requiere que los strings se comparen de esa forma.
- Finalmente si ningún valor ha coincidido le retornamos false.

addPost: Esta función se va a encargar de añadir las URLs de los post a la Blockchain.

```
// SPDX-License-Identifier: MIT OR Apache-2.0
pragma solidity ^0.8.3;

contract FirstContract {
    // code that goes before …
    //Contract contstructor
    constructor(address _firstOwner){
    owner = _firstOwner;
     }
  //Definition of contract methods
    function addPostUrl(string memory _url) public {
        require(msg.sender == owner, "Error, you are not the owner");
        require(!_existsUrl(_url), "The url already exists");
        posts.push(_url);
    }
}
```

Definimos la función con el parámetro de la URL que será de tipo string y además con el modificador memory para asegurarnos que eth no guarda este valor en la Blockchain. Como modificador de acceso le ponemos publicidad puesto que queremos poder llamarla desde fuera del contrato. Evaluamos si quien envía la petición es el dueño del contrato, si no lo es rompemos el flujo de ejecución. De lo contrario evaluamos que no exista ninguna url como la que nos pasa. Por último hacemos un push al array de los posts. Cabe destacar que el método push solo está disponible en los arrays de tipo memory.

getPosts: Esta función retorna todos los posts almacenados en la Blockchain.

```
// SPDX-License-Identifier: MIT OR Apache-2.0
pragma solidity ^0.8.3;

contract FirstContract {
    // code that goes before …
    //Contract contstructor
    constructor(address _firstOwner){
    owner = _firstOwner;
     }
    function getPosts() public view returns(string[] memory){
       return posts;
    }
}
```

Primero definimos la función y le ponemos un modificador de acceso “público”, después le añadimos “view” un modificador que simplemente le dice a Solidity que la función no realiza escritura en la base de datos con lo que no nos cobrara fees.

Finalmente le decimos que retorne un array de strings.

deletePost: Esta función se va a encargar de eliminar la url proveída de la Blockchain.

```
// SPDX-License-Identifier: MIT OR Apache-2.0
pragma solidity ^0.8.3;

contract FirstContract {
// code that goes before …
//Contract contstructor
constructor(address _firstOwner){
owner = _firstOwner;
 }
function deletePostUrl(string memory _url) public {
require(_existsUrl(_url),
"The provided url does not exist"
);
for(uint i=0; i<posts.length; i++){
if( keccak256(abi.encodePacked(_url))==
    keccak256(abi.encodePacked(posts[i]))
){
// as we do not care about the orde
delete posts[i];
    }
   }
  }
}
```

Primero definimos la función y el parámetro que va a recibir, que va a ser igual que en la anterior función.

Le añadimos el modificador de acceso público y como no retorna nada pues no hace falta poner el “returns”.

Después nos aseguramos de que existe la URL que hay que borrar. Por último creamos el array y cuando encontramos el elemento simplemente lo eliminamos con la palabra reservada delete.

Cabe destacar que solo habrá que eliminar una vez la URL dado que por el diseño que hemos implementado no puede existir una misma URL dos veces.

Ahora tenemos un Smart Contract funcional.

Para una clase ya quedo muy larga continuaremos desarrollando el contrato en la siguiente clase El código está en este repositorio de GitHub.

https://github.com/chgara/introasmartcontracts

Solidity by Example

v 0.8.13

an introduction to Solidity with simple examples

https://solidity-by-example.org/