Solidity Example

Christian Gimenez

12 de junio de 2025

Índice

1. Herramientas

Herramientas que vamos a utilizar:

Editor y Emulador Web `https://remix.ethereum.org/`

Solidity Language, documentación `https://docs.soliditylang.org/en/v0.8.3/`

2. Tener en cuenta

- **Siempre prestar atención al flujo del valor.**
  - Consumo y transferencia de ethereums.
  - **¿Cuánto cuesta hacer/almacenar . . . ?**
- Usar `require()` donde haya que chequear algo importante.
  - Revierte una transacción cuando la condición es false.
  - ¿Y si preguntamos al final y usamos `revert()`?
  - ¿Consume ethereum? ¿Qué consume más?
- Cuáles IDs de usuarios se guardan y por qué.
  - ¿Es necesario reutilizar estos IDs luego?
- Cantidad de información.
  - ¿Por qué hay varios tipos de datos?
  - ¿Qué se guarda en el blockchain?
  - ¿Cómo se guarda? ¿cuándo en un bloque nuevo?
  - ¿Consume ethereums guardar mucha información?

2.1. ¿Públicos o ocultos?

Los smart contract de **Ethereum son públicos**:

- Se pueden leer porque se guardan en el blockchain.
  - En realidad pueden tener un precompilado, pero se puede decompilar.

    - Ver un smart contract de cryptokitties

- Almacenar datos que no deben ser públicos: ¿se debe cifrar previamente?
    - ¿Qué quedaría en el blockchain?
- ¿Puede haber blockchains cuyos smart contracts cargados sean ocultos? → Sí, existen.
- Pregunta 1: ¿qué sucede con la privacidad de los datos?
- Pregunta 2: **¿sería más seguro si el smart contract fuese oculto?**
    - Pista: principio de Kerckhoffs y esta imágen

3. Un ejemplo

Vamos a hacer un Smart Contract para registrar el alquiler de películas.

Por el momento, llamemos çliente.[a] la persona que quiere ver la película (no el cliente del desarrollador)

- Las películas se alquilan por 24 h.
- Queremos registrar qué ID del cliente y qué ID de la película alquila.
- Un booleano indicando si se alquiló o si ya se devolvió.
- Queremos implementar las siguientes acciones:
    - `rent`: alquilar una película a un cliente
        - Se transfiere un monto desde el cliente.
        - No nos preocupemos por si la película está alquilada a otra persona.
    - `get_film`: ver si una película está alquilada y a qué ID de cliente.
    - Devolver no se implementa en este ejemplo.

4. Encabezado

4.1. Comentario de licencia

```
/*

   Copyright 2018 Christian Gimenez

   Author: Christian Gimenez

   example.sol

   This program is free software: you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation, either version 3 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */
```

4.2. Pragma

- Define qué versión de Solidity vamos a usar.
- El lenguaje sufrió muchos cambios.

```
pragma solidity ^0.4.22;
```

4.3. Comienza la definición del contrato

- Crearemos un contrato llamado "Films".
- La idea es guardar alquileres de películas.

```
contract Films {
```

5. Estructuras dentro del contrato

Una estructura para guardar los datos de las pelis.

Requiere el identificador de la película, si fue alquilada o no y el ID del cliente.

```
struct Film {
  uint256 film_id;
  bool hd_rented;
  uint256 client_id;
}
```

Una lista de películas alquiladas. Aquí se guardará las películas que se vayan alquilando.

```
Film[] public lst_films;
```

5.1. Owner data

También necesitaremos lo siguiente, que es muy habitual agregarlo para poder manipular el contrato.

Guardaremos el precio base del alquiler y la cuenta del owner. Esta cuenta será necesaria para chequear qué funciones podrán ser ejecutadas por el owner.

Estos datos serán pasados cuando se construya y se suba el contrato (*deploy*).

```
uint256 rent_price; // gwei
address owner;
```

6. Macros/Modifiers

El guión bajo (_) indica que el resto de la función debe ejecutarse aquí.

```
modifier onlyOwner(){
  require(msg.sender == owner,
          "You are not the owner!");
  _;
}
```

7. Constructor

- ¿Qué se debe hacer al realizar un *deploy*?
- ¿Qué datos se solicitan?

```
constructor(address owner_adr, uint256 price) public {
  rent_price = price;
  owner = owner_adr;
}
```

8. ¡Comienza la API!

8.1. Función rent

- Cuando se llama a `rent` se requieren del ID del cliente y del film.
- Es `payable` y `public`.

```
function rent (uint256 film_id, uint256 client_id) payable public {
```

Extremado cuidado con el precio: Utiliza `require` para forzar un chequeo.

```
require(msg.value >= rent_price,
        "The rent price is higher.");
```

- Transferir el valor al owner.
- Usa `msg`.
- Crea una estructura nueva de Film y la agrega al listado.

```
owner.transfer(msg.value);
lst_films.push(
               Film({
                   film_id: film_id,
                   hd_rented: true,
                   client_id: client_id
                   })
               );
} // end rent()
```

8.2. Función get_film

- ¿Quién alquiló esta película?
- Es `view` y `public`.
- Observar el modificador `onlyOwner`.
- ¿Quién puede ejecutar este comando? ¿por qué?
- Define qué retorna: Una tupla.

```
function get_film (uint256 num) view public onlyOwner returns (uint256 film_id, uint256
{
    film_id = lst_films[num].film_id;
    client_id = lst_films[num].client_id;
}
```

8.3. Función get_amount

- Obtener la cantidad de dinero que se obtuvo.
- Solo el dueño del Smart Contract puede utilizar esta función.

```
function get_amount () view public onlyOwner
  returns (uint256 ret) {
  ret = owner.balance;
}
```

9. Fin del contrato

```
}
```

10. Licencia