***MASTER***

***“Blockchain y Bigdata”***

***UCM***

*Bloque Blockchain:*

*“Desarrollo de dApps con Ethereum”*

Autores: Julián Miguel Pérez Dones

Carlos Alberto Retamozo Llantoy

José David Privado Collados

Javier Espín García

Mateo Cerviño Aldao

Ignacio Jiménez Navarro

**Julio 2020**

**Índice**

[1.- Introducción 3](#_Toc45531561)

[1.1 Objetivo del trabajo 3](#_Toc45531562)

[1.2 Desarrollo del sistema. 3](#_Toc45531563)

[2.- Solución propuesta 5](#_Toc45531564)

[2.1 Smart Contracts 5](#_Toc45531565)

[2.2 Patrones de diseño 5](#_Toc45531566)

[2.2 Instrucciones para el despliegue de la aplicación 5](#_Toc45531567)

[2.4 Ventajas e inconvenientes de la aplicación. Mejoras futuras 5](#_Toc45531568)

*## ---------------------------------------------------------------------------------*

1.- Introducción

*## ---------------------------------------------------------------------------------*

1.1 Objetivo del trabajo

Se requiere la implementación de una aplicación que permita la realización de loterías.

Cada lotería del sistema consistirá en el sorteo de un premio entre los participantes de esta. Los requisitos que el sistema debe cumplir son las siguientes:

* Cualquiera puede crear una nueva lotería, indicando las características de esta:
  + número máximo de participantes,
  + precio de cada participación,
  + bote a recaudar y
  + premio del sorteo (siempre menor al importe del bote).
* Cada usuario solo puede participar una sola vez en la misma lotería. Los usuarios deberán pagar el precio de la participación para formar parte del sorteo.
* La lotería estará activa desde su creación hasta que se recaude el bote indicado. Si no se alcanza el importe del bote y se llega al número máximo de participantes, la lotería quedará anulada, devolviendo así el importe de la participación realizada por cada usuario.
* Si se recauda una cantidad equivalente al importe del bote, la lotería dejará de estar activa y no admitirá más participantes. En ese momento, se elegirá un ganador al azar entre los participantes, y se le transferirá el premio estipulado en el sorteo. La parte restante del bote se transferirá al creador del sorteo, como honorarios por su servicio.

Tal y como indica la Figura 1, la aplicación que se ha de construir consta de dos partes:

1. Servicio *API REST*, construido con NodeJS, Express y Swagger. Este servicio servirá como punto de entrada para el usuario a nuestro servicio de creación de loterías, conectando con el sistema de smart contracts.
2. Smart contracts, desarrollados en Solidity y desplegados en una red Ethereum. Contarán con toda la lógica necesaria para la creación y realización de loterías. Se aconseja utilizar el framework Truffle para el desarrollo.

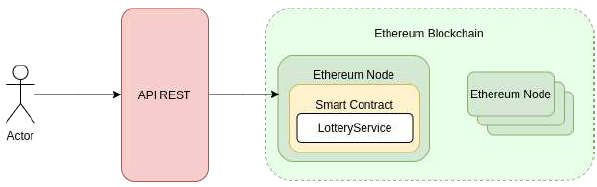


Figura 1. Diagrama de la arquitectura de la aplicación

1.2 Desarrollo del sistema.

A continuación, se recoge un breve resumen de las partes que componen el sistema con el ánimo de clarificar el desarrollo que debe realizarse para esta práctica.

***1. Servicio API REST.***

Se deberá construir un servicio *API REST* completo a partir del *boilerplate* aportado por el profesor. Se trata de un proyecto simple escrito en *JavaScript ES6*, que debe ejecutar en *NodeJS* y que utiliza principalmente las siguientes librerías:

- express: para la creación del servidor y la exposición de la API.

- swagger-express-mw: para la definición y exposición de la API.

El principal objetivo del servicio es ser el punto de acceso de los usuarios al sistema de loterías. Por tanto, el servicio sólo debe encargarse de “transformar” la petición del usuario y conectar con Ethereum para servir dicha petición. Es decir, se trata fundamentalmente de un conector puente a Ethereum, por lo que no debe contener

lógica de negocio.

Además de las librerías ya mencionadas, se deberá utilizar la librería web3js para conectar con nuestro nodo Ethereum e interactuar con la red y los contratos desplegados en la misma. Se deberán implementar todas las operaciones indicadas en los controladores, de forma que se puedan acceder a las funciones que contienen los

contratos. En caso de detectar la necesidad de crear un nuevo *endpoint* en la *API* para acceder a una funcionalidad de los contratos que no esté ya cubierta, podréis incluirla previo aviso al profesor.

Se optará por una gestión simplificada de las cuentas Ethereum de los usuarios (*wallets*). Cada usuario será responsable de guardar sus propias claves (*private key* y *address*) de forma segura, y tendrá que aportar su clave privada (en una de las cabeceras de la petición, llamada *privateKey*) para ejecutar funciones en el sistema

de smart contracts. Esta aproximación supone un gran problema de seguridad y privacidad para el usuario, siendo totalmente desaconsejable su uso en una aplicación en producción.

***2. Smart Contracts.***

El sistema de contratos debe permitir toda la funcionalidad asociada al sistema de loterías que se quiere construir. Se deben tener en cuenta aspectos como: buen diseño de los contratos, aplicación de patrones, medidas de seguridad y ahorro de gas. A continuación, se presentan unas notas que pueden ayudar al desarrollo del sistema:

* El sistema está dividido funcionalmente en dos partes: proceso de creación y registro de loterías, gestión de ciclo de vida de la lotería creada. La separación de lógica en contratos facilita el diseño de estos (modularidad y mantenibilidad) y favorece la seguridad.
* Se utilizará la moneda nativa de Ethereum, Ether (ETH), para realizar el pago de las participaciones en cada lotería. El precio de la participación se deberá indicar en la unidad *wei*.
* El creador de una lotería será el propietario de esta. Sólo el propietario de la lotería podrá realizar el sorteo, dando por terminada la misma y transfiriendo el premio al ganador y el monto restante a sí mismo.
* Si no se recauda el bote establecido de una lotería con las participaciones de los usuarios, la lotería queda en estado fallida, y los usuarios podrán solicitar el reembolso del importe de la participación.
* Si un usuario intenta participar en una lotería que ha finalizado (ha recaudado el bote acordado) o ha llegado a su máximo de participantes, la transacción debe ser revertida. Tampoco se debe permitir que un usuario participe en una lotería fallida o terminada, ya que no tendría sentido.
* Se debe poder saber el estado que tiene la lotería en un momento determinado: si está activa y espera nuevos participantes, si ha finalizado y está pendiente de realizarse el sorteo, si ha terminado y el premio se ha transferido o si ha fallado y los usuarios pueden solicitar el importe de su participación. La Figura 2 describe los estados por los que puede pasar una lotería.
* La elección del ganador del sorteo la realizará el creador y propietario de la lotería de forma aleatoria. La aleatoriedad en un sistema distribuido como es Ethereum es un problema bastante complejo. Se va a utilizar un enfoque simplificado, mezclando parámetros intrínsecos de la red (*block.difficulty y block.timestamp*) con parámetros de nuestro propio sistema (como el número de loterías creadas en el sistema) para generar esa aleatoriedad.

Además, se debe tener en cuenta estos dos aspectos:

* Se tiene pensado crear un *front-end* en un futuro para hacer más accesible la aplicación. Los contratos deben estar preparados para que la interfaz pueda actualizar la información en tiempo real, como la creación de nuevas loterías en el sistema.
* Teniendo en cuenta el carácter novel y cambiante de la tecnología y la posible aparición de errores de programación y bugs en los contratos, se quiere separar la lógica principal del sistema (creación y registro de loterías) del almacenamiento. De esta forma, se podrán desplegar nuevas versiones del sistema, manteniendo toda la información de las loterías.

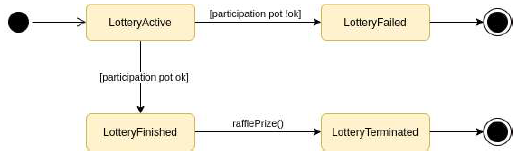


Figura 2. Diagrama de estados de una lotería

***3. Infraestructura.***

Por simplicidad, se utilizará el nodo Ethereum de pruebas ganache-cli. Por lo tanto, el sistema de contratos se debe poder desplegar y ejecutar en la red de pruebas que crea dicho nodo. Se debe configurar para que utilice siempre las mismas cuentas (*wallets*) (modo determinista). Al iniciar el nodo en dicho modo nos indica las cuentas que se

han precargado con 100 ETH de saldo.

*## ---------------------------------------------------------------------------------*

2.- Solución propuesta

*## ---------------------------------------------------------------------------------*

2.1 Smart Contracts

Explicar qué funcionalidades, responsabilidades y características tiene cada uno de los contratos desarrollados. Indica las interacciones que se producen entre ellos.

2.2 Patrones de diseño

Indica los principales patrones de diseño que se han utilizado en el desarrollo de los contratos, y a qué parte (o funcionalidad) de los contratos afecta.

2.2 Instrucciones para el despliegue de la aplicación

Detalla los pasos a seguir para realizar el despliegue completo de la aplicación (infraestructura, contratos, servicio *API REST*…) y la realización completa de una lotería (creación, participación, y sorteo del ganador). Para ello, añade las instrucciones que consideres oportuno (comandos, peticiones *curl*…).

Despliegue y configuración (proyecto truffle): 3 partes

|  |  |
| --- | --- |
|  | * Infraestuctura:   Habrá que levantar el nodo ganache en modo determinista (mismas cuentas, prefondeadas). Para ello utilizaremos uno de los scripts preparados en el fichero *package.json,* que nos levantará el cliente ganache en el puerto 8545 de nuestra misma máquina (es decir, localhost:8545)   * Contratos: *lottery-contracts (proyecto truffle)* * Toda la lógica de negocio de nuestra lotería se recoge en el fichero “lottery.sol” (desarrollado en Solidity), * Habrá que reflejar la configuración adecuada en los ficheros .js y .json (puntos 2, 3 y 4) * Una vez compilado y desplegado nuestro contrato obtendremos su dirección y sus parámetros ABI, información necesaria para poder ejecutar el contrato * API/backend: *lottery-back (proyecto node, con swagger y con express)* * La información anterior referida al contrato (dirección y ABI) la meteremos en el fichero de configuración del backend (fichero *config.js* para el servicio API Rest, punto 5) para poder a dicho contrato * En el fichero *lotteries.services.js* es donde se indica toda la implementación, indicándose cómo realizar las llamadas a las diferentes funciones/métodos del Smart contract desde el backen   Veamos todo esto en detalle, punto por punto: |

**CONFIGURACION DE LOS CONTRATOS (sobre la carpeta *lottery-contracts)***

1. Dejamos el contrato *“Lottery.sol”* (desarrollado en Solidity y conteniendo toda la lógica de negocio) en la carpeta lottery-contracts > contracts (ver punto 1). Apuntaremos la versión de compilador a utilizar, indicado en la primera línea de código, pues ese valor deberá indicarse también en el fichero de configuración *“truffle-config.js”* que veremos en el punto 3:

pragma solidity >=0.4.22 <0.7.0;

1. Creamos el fichero *“2\_deploy\_lottery.js”* para la migración y despliegue del contrato anterior. Este fichero debe dejarse en la carpeta lottery-contracts > migrations, y su nombre debe empezar por un número (pues se irán ejecutando los ficheros que contenga esa carpeta en orden, empezando por el 1 => el hecho que nuestro fichero empiece por 2 se debe a que a modo de ejemplo ya hay un fichero de ejemplo, llamado *“1\_initial\_migration.js”* que hemos conservado y cuyo nombre empieza por 1).

En este fichero se indican las referencias al artifact “Lottery” de nuestro contrato, que será el que queramos migrar:

// 2\_deploy\_lottery.js

const Lottery = artifacts.require(‘Lottery’);

module.exports = async (deployer) => {

  await deployer.deploy(Lottery);

};

1. Indicaremos los datos de configuración del proyecto truffle en el fichero *“truffle-config.js”,* fichero en el que indicaremos:
   * La versión del compilador solc que estamos utilizando (ver punto 1)
   * Detalles de la red blockchain que vayamos a utilizar, indicando su nombre (‘development’), IP (localhost) y puerto (para esta práctica levantaremos el nodo ganache en el puerto 20000)

  networks: {

     development: {

      host: '127.0.0.1',     // Localhost (default: none)

      port: 20000,            // Standard Ethereum port (default: none)

      network\_id: '\*',       // Any network (default: none)

    },

1. En el fichero *“package.json”* se indicarán las dependencias de versiones para truffle y cliente ganache, así como 3 scripts de ayuda para levantar ganache y utilizar truffle (invocaciones por línea de comandos)

{

  "name": "lottery-contracts",

  "version": "1.0.0",

  "description": "Lottery Ethereum Contracts",

  "keywords": [

    "lottery",

    "blockchain"

  ],

  "author": "@Práctica ETH - Máster UCM - Grupo 1",

  "main": "truffle-config.js",

  "dependencies": {

    "truffle": "^5.1.18",

    "ganache-cli": "^6.9.1"

  },

  "scripts": {

    "truffle": "./node\_modules/.bin/truffle",

    "ganache": "./node\_modules/.bin/ganache-cli -d -p 20000”,

    "migrate": "./node\_modules/.bin/truffle migrate"

  }

}

NOTA: si quisiéramos que el nodo ganache corriera en un puerto diferente al indicado (2000 en esta práctica) bastaría con modificar su invocación en consecuencia, tanto aquí como en el fichero *“truffle-config.js”* visto antes.

Por tanto, y estando en la carpeta lottery- contracts y en línea de comandos, habría que levantar la infraestutura del proyecto y migrar los contratos (compilación y despliegue):

* + **$npm install** -> para construir las librerías (según las dependencias indicadas *para truffle y ganache-cli* en el fichero *“package.json”* propio de la parte contracts). Tras ello se deberá crear en esa misma carpeta el fichero “*package-lock.json”* con todo el detalle de dicha creación de librerías
  + **f$ npm run ganache**  -> para levantar ganache (llamada al script anterior, con la opción -d para indicar vel determinismo)

Dejamos este proceso ganache corriendo en background, y abrimos nueva terminal (y volvemos a ir a la carpeta lottery-contracts:

* + **$ npm run migrate** 🡪 se conecta a nuestro nodo (indicado en el fichero de configuración), compila el contrato y hace su migración (va ejecutando los diferentes scripts que haya en la carpeta de migraciones: para nuestra práctica ejecutará el fichero *“2.deploy\_lottery.js”,* ya comentado en el punto 2., se obtiene por pantalla el detalle de esta migración => de esta información deberemos anotar la dirección del contrato desplegado, pues esa información deberá proporcionarse después al backend para poder interaccionar con él:

2\_deploy\_lottery.js

===================

Deploying ‘Lottery’

-------------------

> transaction hash: 0x39fde9e44618a26d17369cb34a5d055fe1cbd30d5c41bc6052b8f03aea220a72

> Blocks: 0 Seconds: 0

> contract address: **0xCfEB869F69431e42cdB54A4F4f105C19C080A601**

> block number: 3

> block timestamp: 1594894035

> account: 0x90F8bf6A479f320ead074411a4B0e7944Ea8c9C1

> balance: 99.9939459

> gas used: 96189 (0x177bd)

> gas price: 20 gwei

> value sent: 0 ETH

> total cost: 0.00192378 ETH

> Saving migration to chain.

> Saving artifacts

-------------------------------------

> Total cost: 0.00192378 ETH

Nótese que al compilarse el contrato se crea la carpeta /build/contracts el fichero *“Lottery.json”* con la información de la compilación, de la cual nos interesa anotar el apartado relativo a los parámetros abi, pues esa información habrá que pasársela también al backend para poder hacer la comunicación con el contrato

**CONFIGURACION DEL BACKEND (sobre la carpeta *lottery-back*)**

1. Dentro de la carpeta lottery-back > config, y dentro del fichero *“config.js” (fichero que unifica todos los parámetros de configuración para la conexión backend <-> contratos),* habrá que indicarle al backend los parámetros de conexión al contrato y nodo ya desplegado, esto es:

* Datos del contrato con el que comunicarse: en nuestro caso “*Lottery*”
  + 1. El nombre del contrato: en nuestro caso “*Lottery*”
    2. La dirección del contrato desplegado (ver punto anterior)
    3. Parámetros del contract.abi (ver punto anterior)
  + Datos del nodo: en nuestro caso a este módulo le llamaremos “eth”, dato importante pues sobre ese nombre podremos realizar después las futuras invocaciones para conocer detalles de la especificación del nodo (por ejemplo conocer su url mediante config.eth*.nodeUrl*, conocer su puerto de escucha mediante config.*eth.nodePort*, etc… de la misma manera que podríamos conocer a especificación de los contratos mediante *config.contracts.Lottery.xxxx:*
    1. Url y puerto del nodo (parámetros de conexión a nuestro nodo ganache)
    2. Opciones de gas en las transacciones (si es que se quiere indicar alguno: de indicarse la única limitación será no superar al gas definido por la propia red)

module.exports = {

  eth: {

    nodeUrl: 'localhost',

    nodePort: '20000',

    transactionOptions: {

      gas: 6721975,

      gasPrice: 0,

    },

  },

  contracts: {

    Lottery: {

      contractName: 'Lottery',

      contractAddress: **'0xCfEB869F69431e42cdB54A4F4f105C19C080A601'**,

      contractAbi: '[{"constant":true,"inputs":[],"name":"retreive","outputs":[{"internalType":"uint256","name":"","type":"uint256"}],"payable":false,"stateMutability":"view","type":"function"},{"constant":false,"inputs":[{"internalType":"uint256","name":"num","type":"uint256"}],"name":"store","outputs":[],"payable":false,"stateMutability":"nonpayable","type":"function"}]',

      maxNumberOfParticipants: 5,

      participationPrice: 1,

      participationPot: 2,

      prize: 100

    },

    Migrations: {

    }

  },

};

1. Implementar en el fichero *“lotteries.service.js”* las llamadas a los diferentes servicios de la lotería implementada, importando para ello en su código toda la configuración indicada en el fichero anterior de manera que se permita así las comunicaciones con el contrato desplegado, recordando que:
   * *config.eth.xxxxx* para obtener detalles de la especificación del nodo
   * *config.contracts.Lottery.xxxx* para obtener detalles de la especificación del contrato

// lotteries.service.js

const config = require('../../config/config');

const Web3 = require('web3')

const ethereumUrl = `http://${config.eth.nodeUrl}:${config.eth.nodePort}`;

const web3 = new Web3(ethereumUrl)

/\*\*

 \* Return a list of registered lotteries.

 \*/

async function getLotteries() {

  // Ethereum config

  const ethereumUrl = `http://${config.eth.nodeUrl}:${config.eth.nodePort}`;

  const gasLimit = config.eth.transactionOptions.gas;

  const gasPrice = config.eth.transactionOptions.gasPrice;

  // Lottery contract specs

  const LotteryContractName = config.contracts.Lottery.contractName;

  const LotteryContractAddress = config.contracts.Lottery.contractAddress;

  const LotteryContractAbi = config.contracts.Lottery.contractAbi;

  // TODO

  return { message: 'Ok' };

}

Sobre este fichero “*lotteries.service.js” se ha implementado* la lógica que permita la conexión y llamadas a la función correspondiente del contrato que ofrece la lógica requerida, mediante la librería Web.js3:. Tras esas invocaciones obtenemos mediante la API la respuesta a las llamadas al contrato en un objeto json

Una vez tengamos el fichero “*lotteries.service.js”* correctamente construido, y estando en la carpeta lottery-back y en línea de comandos, habría que levantar la infraestutura del servicio NodeJ. Para ello, y tal y como hicimos en la parte de contratos:

* + **$npm install** -> para construir las librerías que se necesitan (según las dependencias indicadas *para express, swagger y web3* en el fichero *“package.json”* propio de la parte back*)*. Tras ello se deberá crear en esa misma carpeta el fichero “*package-lock.json”* con todo el detalle de dicha creación de librerías
  + **f$ PORT=30000 npm start**  -> para levantar la aplicación (y en el puerto 30000 concretamente)

Si todo ha ido ok en consola se indicará los siguiente:

Server listening on port 30000

Application started successfully.

… y lo dejamos también corriendo en background (tal y como hicimos con el proceso de ganache-cli)

NOTA: el hecho que hayamos decidido modificar los parámetros por defecto para el nodo ganache (por defecto 8454, cambiado en esta práctica al 2000) y el servidor backend (por defecto 10010, cambiado a 30000) se debe a que las máquinas clouds que la UCM nos proporciona para la realización de la práctica tiene capados la mayoría de los puertos, por seguridad, siendo el 20000 y el 30000 de os pocos que están abiertos para acceder a servicos que corran en eloos desde fuera de la máquina).

Una vez hayamos completado todos estos pasos podríamos ya probar la correcta ejecución del contrato, sin más que invocar desde línea de comandos a las llamadas de prueba que tenemos en el fichero *“request-guide.txt”* (ubicado en la raíz del proyecto), recordando “apuntar”2 en las llamadas al puerto de escucha de nuestro backend (en este caso al puerto 10010):

* Abrir una nueva terminal
* Por ejemplo probamos la llamada a la función getLotteries():

**$ curl --location --request GET 'http://localhost:30000/lotteries'**

… que desencadena una llamada al método getLotteries() en la API (dentro de “*lotteries.services.json”*), que a su vez contiene la llamada web3 a la función equivalente en el fichero “*Lotteries.sol”* desplegado en el nodo ganache de nuestra red blockchain.

2.4 Ventajas e inconvenientes de la aplicación. Mejoras futuras

¿Qué puntos positivos y negativos tiene el sistema construido? ¿Qué cambios se podrían incluir para mejorar el sistema?

1. Puntos positivos del sistema:

Fxhdfdhfhfdsh

1. Puntos negativos del sistema:

* Problema de Seguridad/Privacidad: tal y como está construido el sistema, en nuestra práctica y tan sólo como prueba de concepto, en aquellas transacciones que requieran modificaciones sobre la blockchain (no en consultas de lectura) para poder interactuar con la aplicación cada participante debe aportar su clave privada (viajando en el parámetro --*header* de la petición, llamada *privateKey*) lo que es totalmente desaconsejable en entornos productivos, al no haberse definido limitaciones de acceso a la información sobre la blockchain pública empleada.

1. Cambios propuestos para mejorar el sistema
2. MEJORA al problema de la aleatoriedad en sistemas distribuidos

El problema de la elección del ganador utilizado en los contratos inteligentes es un problema inherente a las computadoras, pues éstas son deterministas y generan siempre las mismas salidas ante las mismas entradas. Hoy días las computadoras generan números aleatorios mediante el algoritmo PRNG (Pseudo Random Number Generator), basados en valores de semilla altos en entropía (alta aleatoriedad), que siempre producirán el mismo número aleatorio si se repiten las circunstancias que lo generan, si bien la solución ideal sería las basadas en algoritmos TRNG (True Random Number Generator), en la práctica muy costoso de implementar. En esta práctica la aleatroriedad se consigue mezclando parámetros intrínsecos de la red (*block.difficulty y block.timestamp*) con parámetros de nuestro propio sistema (como el número de loterías creadas en el sistema), que no deja de ser un algoritmo PRNG (no TRNG) pues como decimos siempre produciría el mismo número aleatorio bajo las mismas condiciones iniciales.

NOTA: podría pensarse que el “tiempo” podría ser un parámetro ideal para utilizarlo como semilla, pues sería imposible volver al mismo estado ya pasado, pero realmente no lo es, puesto que no se puede decir realmente la hora en una blockchain (y menos aún si habláramos de una blockchain sobre infraestutura distribuida, aunque no es este caso).

Por tanto como mejora futura se propondría, caso de llevarse el sistema a entornos productivos, la utilización de verdaderos algoritmos TRNG (por ejemplo, generación de números aleatorios basados en movimientos del mouse, ruido ambiente, etc), si bien para entornos de pruebas y académicos es suficiente el actual método basado en números de bloque, marcas de tiempo o cualquier número derivado de un nuevo bloque.

1. MEJORA al problema de Seguridad

Se propone modificar la lógica interna del sistema de manera que no sea necesario que los participantes tengan que aportar su private key en cada una de las interacciones con el sistema (problema comentado como punto negativo del sistema en el apartado anterior).

1. MEJORA sobre la lógica interna:

* Automatización de las devoluciones: las condiciones iniciales de esta lotería indican que “*Si no se recauda el bote establecido de una lotería con las participaciones de los usuarios, la lotería queda en estado fallida, y los usuarios podrán solicitar el reembolso del importe de la participación”.* -> Se propone como mejora automatizar esto, y modificar los Smart Contracts de manera que se realice el reembolso a todos y cada uno de los participantes de manera automática, sin necesidad de que estos lo desencadenen.
* Eliminar la dependencia de la figura del “creador/propietario de la lotería”: bajo las condiciones actuales sólo el propietario de la lotería puede realizar el sorteo, y sólo él puede transferir el premio al ganador -> se propone como mejora que sea el sistema quien realice/gestione todo esto, eliminando por tanto la dependencia de la figura del “creador de la lotería”
* Imposibilidad de tener loterías “fallidas/anuladas” por haberse alcanzado el máximo número de participantes pero no haberse alcanzado el importe del bote, sin más que añadiendo la lógica apropiada (por ejemplo, añadiendo en el Smart Contract la restricción

Bote a recaudar = (Num. Max participantes) \* (Precio por participación).

1. MEJORA sobre la infraestuctura:

* Modificar la red blockchain utilizada, de manera que el sistema pueda funcionar sobra otras redes diferentes a las creadas por defecto por ganache-cli (permitir por tanto que el sistema funcione con otras direcciones a las 10 prefondeadas por ganache-cli)
* *Creación de un front--end* para hacer más accesible la aplicación, mostrando y actualizando la información en tiempo real (por ej. permitiendo la creación de nuevas loterías en el sistema, poder consultar el estado de las mismas, etc).

15/07: prueba de modificación y subida con github

*####################################################################################*

*FIN PRACTICA*

*####################################################################################*