TESIS DE GRADO EN INGENIERÍA MECÁNICA

APLICACIÓN DE CONTRATOS INTELIGENTES EN ENSAYOS CLÍNICOS

Agustin Parrotta

Dr. Flavio Colavecchia
Director

Dr. René Cejas Bolecek
Co-director

Miembros del Jurado
Dr. Eugenio Urdapilleta
Ing. Eduardo Tapia

20 de Junio de 2019

Laboratorio de Física Médica Computacional – Fundación Intecnus
Centro Atómico Bariloche

Instituto Balseiro
Universidad Nacional de Cuyo
Comisión Nacional de Energía Atómica
Argentina

A mi familia
A mis amigos
A los de acá
A mi director

Índice de contenidos

Índice de contenidos **v**

Índice de figuras **vii**

Resumen **ix**

Abstract **xi**

1. Introducción **1**
1.1. Ensayos clínicos 1
1.2. Blockchain 2
1.2.1. Proceso de validación de transacciones 3
1.2.2. Contratos inteligentes 5
1.3. Blockchain como solución a la gestión de datos clínicos 5
1.4. Objetivos y desarrollo del proyecto 6

2. Marco teórico **7**
2.1. Ethereum 7
2.1.1. Transacciones 7
2.1.2. Cuentas 8
2.1.3. Tokens de Ethereum 8
2.2. Metodología 10
2.3. Herramientas utilizadas 10

3. Gestión de títulos académicos **15**
3.1. Planteo del problema 15
3.2. Solución propuesta 16
3.3. Objetivo 16
3.4. Requerimientos 17
3.5. Desarrollo 18
3.5.1. Diseño de la aplicación 18
3.5.2. Backend: creación del contrato inteligente 19

3.5.2.1. Codificación del contrato 19
3.5.2.2. Despliegue del contrato 27
3.5.2.3. Pruebas unitarias del contrato 29
3.5.3. Frontend: creación de la página web 30
3.5.3.1. Codificación de la página web 30
3.5.3.2. Configuración de las herramientas 32
3.6. Observaciones 32

4. Sistema de gestión de ensayos clínicos 35
4.1. Requerimientos 35
4.2. Desarrollo 37
4.2.1. Diseño de la aplicación 37
4.2.2. Backend: creación del contrato inteligente 39
4.2.2.1. Codificación del contrato 39
4.2.2.2. Despliegue del contrato 48
4.2.2.3. Pruebas unitarias del contrato 48
4.2.3. Frontend: creación de la página web 50
4.2.3.1. Codificación de la página web 50
4.2.3.2. Configuración de las herramientas 50
4.2.4. Pruebas 52

5. Conclusiones y perspectivas 57

A. Instalación de las herramientas y configuración del servicio 61

B. Contrato para gestión de ensayos clínicos 65
B.1. CT1.sol 65
B.2. CT2.sol 73

C. Práctica Profesional Supervisada y actividades de Proyecto y Diseño 77

Bibliografía 79

Agradecimientos 83

Índice de figuras

1.1. Esquema de una cadena de bloques. Cada bloque contiene un conjunto de transacciones, el hash del bloque anterior y el resultado de la prueba de trabajo. 4

2.1. Esquema conceptual de una aplicación descentralizada. 10

2.2. Esquema conceptual de la utilización de IPFS junto con *blokchain*. . . . 14

3.1. Interfaz de la herramienta Ganache. Se pueden observar las direcciones de las cuentas generadas y la cantidad de ETH disponible para cada una. 28

3.2. Interfaz de la sección *Blocks* de Ganache. Se destacan el gas total utilizado y el número de transacciones que contiene cada bloque. 28

3.3. Transacción contenida en un bloque. Se detalla el gas usado, el hash, el tipo y el emisor de la transacción. 29

3.4. Resultados del *test* realizado. Se han probado la totalidad de las funciones programadas. 30

3.5. Resultado al ejecutar el comando `npm run dev` en la terminal, observándose que Webpack se ha ejecutado correctamente. 32

3.6. Página web de la aplicación y complemento Metamask desplegado. . . . 33

4.1. Esquema que indica la relación entre los distintos actores del ensayo clínico. 38

4.2. Resultados del *test* realizado. Se han probado la totalidad de las funciones programadas. 49

4.3. Resultado al ejecutar el comando `ipfs daemon`, el cual inicializa un nodo IPFS. 51

4.4. Resultado al ejecutar el comando `npm run dev`, observándose que Webpack se ha ejecutado correctamente y corre en `http://localhost:8081`. 51

4.5. Página web de la aplicación y complemento Metamask desplegado. . . . 52

4.6. Estableciendo una entidad, en este caso, un laboratorio. Al agregar la dirección del nuevo laboratorio y apretar el botón Set, se despliega Metamask solicitando la autorización para realizar la transacción con el contrato. 53
4.7. Registro de transacciones de Ganache, donde se ha seleccionado aquella que corresponde a dar de alta una nueva entidad (en este caso, un laboratorio) en el contrato. La transacción no conlleva un envío de fondos, pero sí incurre en un gasto de gas. 54
4.8. Registro propio de eventos de la aplicación, donde puede observarse la autorización a agregar un laboratorio. 54
4.9. Proceso de almacenamiento del archivo `Documento.pdf` en el sistema IPFS y obtención del hash. 54
4.10. Captura de pantalla de la autorización para dar de alta a una CRO por parte del laboratorio. Nótese que la cuenta que emite la transacción en Metamask es diferente a la utilizada para dar de alta al laboratorio. . . 55
4.11. Registro de transacciones de Ganache, donde se puede observar la interacción con el contrato. 55
4.12. Registro de transacciones de Ganache, donde se ha seleccionado la transacción correspondiente al registro de la CRO. 56

Resumen

Los ensayos o pruebas clínicas consisten en estudios u observaciones sobre la respuesta en el cuerpo humano de nuevas tecnologías médicas, ya sean medicamentos, tratamientos o dispositivos médicos. En general, implican la recolección y el análisis de datos médicos de un gran número de individuos participantes. En el presente trabajo se desarrolló una aplicación descentralizada para facilitar el manejo de estos datos utilizando como base la tecnología *blockchain*. Este sistema garantiza privacidad, inmutabilidad y trazabilidad de los datos allí almacenados. La operación sobre el *blockchain* se realiza a través de contratos inteligentes, que consisten en programas que permiten la disposición de reglas lógicas. En este trabajo se identificaron los principales requerimientos del problema, se realizó el desarrollo de los contratos inteligentes sobre la red Ethereum y, finalmente, se programó la interfaz web que permite la interacción entre los usuarios y los contratos.

Palabras clave: CONTRATOS INTELIGENTES, ENSAYOS CLÍNICOS, BLOCKCHAIN, ETHEREUM

Abstract

Clinical trials consist of studies or observations on the response of new medical technologies in human body, whether medications, treatments or medical devices. In general, they involve the collection and analysis of medical data from a large number of individuals. In the present work, a descentralized application was developed to facilitate the management of this data using the blockchain technology. This system guarantees data privacy, inmutability and traceability. The blockchain operation is performed by smart contracts, which consist of programs that automate actions in the network according to logical rules. Three stages were developed: first, the main requirements or the application were identified. Second, smart contracts were coded over the Ethereum network and, finally, the interface that allows interaction between the user and the contract was programmed as a web application.

Keywords: SMART CONTRACTS, CLINICAL TRIALS, BLOCKCHAIN, ETHEREUM

Capítulo 1

Introducción

1.1. Ensayos clínicos

Los ensayos clínicos son investigaciones médicas en las que se evalúan medicamentos, tratamientos o dispositivos médicos, con el objeto de diagnosticar su eficacia y seguridad. Son llevadas a cabo por instituciones públicas o privadas y requieren la participación voluntaria de personas tanto sanas como enfermas [1].

Dentro del campo de los estudios clínicos, la farmacología clínica evalúa los efectos de los fármacos en los seres humanos. En estos estudios se distinguen tres fases previas a la comercialización de un nuevo fármaco y una fase posterior [2]:

- Fase 1: son pruebas de seguridad iniciales en un nuevo medicamento. Se intenta establecer el rango de dosis tolerado por los voluntarios. Estos ensayos a veces se realizan en pacientes gravemente enfermos (por ejemplo, en el campo del cáncer), y son monitoreados de cerca.

- Fase 2: son ensayos clínicos controlados para evaluar la eficacia y seguridad en pacientes con la enfermedad que se va a tratar. Se evalúan los efectos para distintos rangos de dosis. Estos ensayos generalmente representan la demostración más rigurosa del medicamento.

- Fase 3: son estudios realizados en grandes y variados grupos de participantes. Se exploran y discriminan las reacciones más frecuentes según las características de los mismos. Estos ensayos a menudo proporcionan gran parte de la información necesaria para el prospecto y el etiquetado del medicamento.

- Fase 4: son estudios llevados a cabo luego de aprobada la comercialización del medicamento para establecer la aparición de nuevas reacciones adversas, la confirmación de la frecuencia de las conocidas y las estrategias de tratamiento.

Los estudios clínicos generalmente involucran muchas partes interesadas a lo largo de todo el proceso, desde laboratorios y entes reguladores hasta médicos y pacientes, generando un gran flujo de información y datos sin procesar que luego serán evaluados [3]. Por lo tanto, es esencial que toda la información se registre cuidadosamente y no esté sujeta a manipulación, de modo tal que se asegure la integridad y la veracidad de la misma [4].

Sin embargo, ante las numerosas entidades involucradas, el gran flujo de datos, el dinero y el tiempo invertidos en un ensayo clínico, a menudo se presentan situaciones de manipulación y alteración de los estudios [3]. Los errores involuntarios y los fraudes, incluyendo el cambio de resultados y la publicación selectiva, son algunos de los problemas que atentan contra la integridad de los ensayos y perjudican la calidad de la investigación [5].

La auditoría de los datos también es una complicación. Actualmente, no están consolidados sistemas que brinden la supervisión de los datos en tiempo real, y son limitados los procesos de trazabilidad de los mismos [6].

A su vez, los crecientes costos de desarrollo y el tiempo son barreras importantes para la innovación en el desarrollo de los fármacos. Se estima que el costo promedio para comercializar un medicamento se encuentra entre USD 643 y USD 2000 millones de dólares, mientras que el tiempo ronda entre 6 y 14 años desde el inicio de las pruebas clínicas hasta la comercialización [7].

Ante esta problemática, la tecnología *blockchain* puede ser un método de bajo costo para auditar y confirmar la fiabilidad de los estudios científicos [8].

1.2. Blockchain

El *blockchain* o cadena de bloques es una tecnología que, si bien remonta sus orígenes a principios de los años 90, comenzó a ganar relevancia en 2008 con el surgimiento de la criptomoneda Bitcoin [9].

Blockchain se define como una red *peer–to–peer* que consiste esencialmente de una base de datos distribuida que almacena registros de transacciones en forma de bloques encriptados [10]. Funciona como un libro contable en el que se almacenan de forma permanente cada una de las transacciones de la red y no está bajo el control de ninguna organización. Las transacciones son intercambios de tokens, los cuales se definen como objetos que tienen valor solo dentro de cierto contexto. En el caso de Bitcoin, los tokens representan monedas digitales.

Una red *blockchain* está constituida por sus participantes, denominados nodos. Cada nodo en esta red *peer–to–peer* posee una copia del registro de transacciones. Si un usuario desea enviar una cantidad de tokens a otro, puede hacerlo anunciando públicamente esta transacción y le corresponde a la red verificar si es correcta. Sin embargo,

es factible que un usuario pueda tratar de manipular la red y emitir más de una transacción del mismo token a diferentes usuarios. Estas situaciones se evitan al exigir una prueba de trabajo de cada nodo que verifique la transacción.

1.2.1. Proceso de validación de transacciones

A continuación se describe en forma más detallada el proceso de validación de las transacciones.

Creación del bloque

Cuando se produce una transacción entre un usuario y otro de la red, esta se emite hacia todos los nodos. Las transacciones emitidas en un período de tiempo se agrupan en un bloque y espera a ser validado. El bloque contiene principalmente la información de las transacciones, el hash del último bloque validado y un campo denominado *nonce* [11].

El hash es una función criptográfica pseudoaleatoria que transforma cualquier bloque arbitrario de datos en una nueva serie de caracteres de longitud fija. Independientemente de la longitud de los datos de entrada, el valor hash de salida tiene siempre la misma longitud. Posee, además la siguiente propiedad: sea $h(x) = y$ la función hash, con x el valor de entrada e y el de salida; si se conoce el valor x, luego y es fácil de calcular; sin embargo, si se conoce el valor de y, la mejor manera de encontrar el valor x es a través de prueba y error.

Validación del bloque

Este proceso consiste en que los nodos utilicen su poder de cómputo para resolver el algoritmo de prueba de trabajo. Estos nodos se denominan mineros.

La prueba de trabajo o *proof of work* es un mecanismo para validar las transacciones de la red, y está basado en el poder de cómputo de los nodos. Aquellos que dispongan de mayor capacidad de procesamiento tienen mayor probabilidad de validar el siguiente bloque, obteniendo un incentivo económico.

La validación de un bloque consiste en encontrar la solución a un proceso inverso de hash. El protocolo establece y comunica una serie de bits como el valor de salida del hash del bloque a validar. Luego, se debe encontrar el valor *nonce* tal que el hash de todo el bloque produzca exactamente el valor establecido. Por ser la función hash una función pseudoaleatoria completamente impredecible, la mejor manera de encontrarlo es a través de prueba y error, variando repetidamente el *nonce*, calculando el hash y viendo si coincide. El resultado de esto es un número que, al combinarlo con el hash del nuevo bloque, permite enlazarlo a la cadena de bloques. Esto se observa en la Figura

1.1. En el *blockchain* de Bitcoin, la red debe realizar un promedio de 2^{69} intentos antes de encontrar un bloque válido.

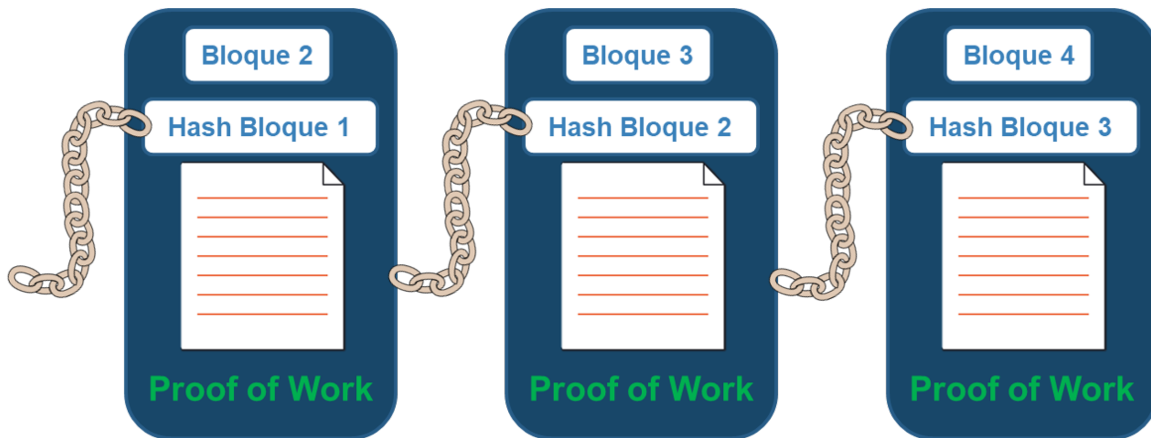

Figura 1.1: Esquema de una cadena de bloques. Cada bloque contiene un conjunto de transacciones, el hash del bloque anterior y el resultado de la prueba de trabajo.

Distribución y verificación de la solución

La solución es encontrada por algún nodo y es emitida hacia el resto de los mineros, los cuales verifican que la solución sea correcta de acuerdo a la información del bloque a validar. Este proceso no es computacionalmente costoso ya que requiere calcular el hash del bloque con la solución encontrada tan sólo una vez.

Encadenamiento del bloque

Si la solución es verificada por la mayoría de los nodos, el bloque nuevo es validado, al igual que las transacciones que contiene, añadiéndose a la cadena de bloques. En este proceso, con la finalidad de compensar a los mineros por este trabajo computacional, el nodo que encontró la solución recibe una recompensa en tokens que, en el caso de Bitcoin, es la criptomoneda que lleva su nombre. Además, las transacciones pueden realizarse con *fees* o tarifas de transacción que son otorgadas al minero como recompensa [9].

El algoritmo de prueba de trabajo es un mecanismo de consenso para validar las transacciones, siendo este uno de los pilares de esta tecnología. Además de la prueba de trabajo o *proof of work*, existen también la prueba de participación o *proof of stake* y la prueba de importancia o *proof of importance* [12].

Es importante resaltar que este protocolo brinda transparencia al tener un registro compartido y auditable. El hecho de que los bloques estén anidados indican que son dependientes entre sí y, por lo tanto, implica que los registros sean inmutables. Esto es así ya que si un nodo produce un cambio en la información de alguna transacción

anterior, modificaría el hash del bloque y el de los bloques sucesivos, por lo que no estaría avalado por el resto de los nodos.

1.2.2. Contratos inteligentes

Un contrato inteligente es un programa que es almacenado y ejecutado en una red *blockchain* y permite automatizar transacciones de acuerdo a reglas lógicas. Cuando se dispara una condición programada en un contrato, esta ejecuta la cláusula contractual automáticamente. Tiene como principal objetivo brindar la seguridad del cumplimiento de las cláusulas de un contrato tradicional. Se pueden utilizar en cualquier transacción que requiera un acuerdo registrado entre partes [13]. Sin embargo, las plataformas *blockchain* que permiten la ejecución de estos contratos, como Ethereum, permiten a su vez el almacenamiento de datos en la red, ampliando inconmensurablemente su funcionalidad.

1.3. Blockchain como solución a la gestión de datos clínicos

El principio básico de la tecnología *blockchain* es que cualquier servicio puede construirse de manera transparente, descentralizada y segura. Por lo tanto, los usuarios tienen un alto grado de control, autonomía y confianza en los datos y en la integridad de los mismos. *Blockchain* permite el seguimiento e inviolabilidad de datos para todo el flujo de documentos en un ensayo clínico. Por lo tanto, garantiza la trazabilidad, la transparencia y la gestión del ensayo a través del uso de contratos inteligentes. Al mismo tiempo, la tecnología garantiza un control preciso de los datos, su seguridad y su visualización [5].

Grandes farmacéuticas como Pfizer, Amgen y Sanofi están utilizando *blockchain* como un medio para agilizar el proceso de desarrollo y prueba de nuevos medicamentos, abordando los problemas actuales en el proceso de ensayos clínicos mencionados en la sección 1.1 [14]. Reducir el costo de los ensayos y mejorar su tasa de éxito depende de una mejor gestión y manipulación de los datos.

Específicamente, las características principales de la plataforma que podrían afrontar los desafíos de los ensayos clínicos son:

- Origen de los datos: las transacciones en la red tienen una marca de tiempo y son inmutables, por lo cual la información clínica almacenada posee trazabilidad.
- Digitalización: las transacciones están digitalizadas, por lo que se minimiza la necesidad de documentos escritos.

- Privacidad y uso compartido de los datos: dependiendo de cómo se establezcan las reglas en el contrato inteligente, los datos pueden ser privados, estableciendo derechos de permiso para su manipulación o visualización.

- Descentralización: las redes *blockchain* están formadas por nodos alrededor del mundo, existiendo una disponibilidad constante de los datos.

1.4. Objetivos y desarrollo del proyecto

El objetivo general de este Proyecto Integrador es diseñar y desarrollar un prototipo funcional de aplicación descentralizada basada en la tecnología *blockchain* para facilitar el manejo de los datos generados en un ensayo clínico mediante el uso de contratos inteligentes. En particular, se debe:

- Diseñar los contratos inteligentes que permitan el registro y la gestión de los datos clínicos, teniendo en cuenta las partes interesadas.

- Diseñar una interfaz que permita a los usuarios interactuar con los datos.

- Realizar las pruebas correspondientes para asegurar la ausencia de errores en el *software*.

El proyecto se dividió en dos etapas. La primera consistió en la adquisición de experiencia con el entorno de programación de contratos inteligentes. Para eso, se realizó una aplicación para la emisión, validación y visualización de certificados académicos digitales del Instituto Balseiro. La segunda etapa consistió íntegramente en el desarrollo del *software* de gestión de datos de los ensayos clínicos.

Esta presentación escrita del trabajo realizado se organiza de la siguiente manera. En el capítulo 2 se describe el marco teórico, detallando la tecnología con la cual se trabaja y las herramientas utilizadas, mientras que en el capítulo 3 se detalla el proceso de desarrollo de la aplicación de gestión de certificados académicos. Luego, en el capítulo 4 se expone el proceso del diseño, programación y evaluación del *software* de gestión de datos de los ensayos clínicos. Finalmente, en el capítulo 5 se presentan algunas conclusiones, y se discuten algunos aspectos claves y perspectivas del trabajo a futuro. Además se incluyen en este documento tres apéndices. En el apéndice A se describen las instrucciones para la instalación y puesta en marcha de un proyecto de contratos inteligentes sobre la red Ethereum. En el apéndice B se encuentra el código completo de los contratos inteligentes para el manejo de ensayos clínicos, mientras que en el apéndice C se detallan las actividades realizadas en el presente Proyecto Integrador, de acuerdo a los requerimientos académicos específicos de la carrera de Ingeniería Mecánica.

Capítulo 2

Marco teórico

En este capítulo se describe el marco conceptual establecido para llevar adelante este trabajo, al igual que las herramientas computacionales de programación, verificación y ejecución de los contratos inteligentes presentados en los próximos capítulos. Es muy importante destacar en este punto que todas estas herramientas son de código abierto.

2.1. Ethereum

Ethereum es una red *blockchain* especializada en ejecutar contratos inteligentes [11, 15]. Está basada en la *Ethereum Virtual Machine* (EVM), *software* que posee la propiedad de ser Turing completo. En este sentido, la EVM puede ejecutar programas con todas las características de los códigos que se utilizan habitualmente en una computadora, como asignar variables, utilizar bucles, etc.; lo cual permite crear aplicaciones más complejas y de una manera más simple y eficiente utilizando contratos inteligentes. Sin embargo, dado que esta máquina virtual está conectada a un *blockchain*, es necesario impedir que se puedan correr códigos de ejecución en tiempo infinito. Para ello, la arquitectura Ethereum dispone de un mecanismo denominado *gas* para que el sistema no colapse [16].

2.1.1. Transacciones

El gas es la forma que implementa Ethereum para calcular el coste de las transacciones y está estrictamente relacionado al gasto computacional de procesar una transacción en la red. Cada tipo de operación básica en la EVM está cuantificada a través de su costo en gas.

Por otro lado, la red Ethereum implementa su propia criptomoneda, el *ether* (ETH). Este token surge esencialmente como la forma de pago entre los usuarios que realizan transacciones para registrar datos en la red y los nodos mineros por la prestación de los recursos computacionales.

En Ethereum, el costo total de una transacción se calcula de la siguiente manera:

$$\texttt{CostoTotal} = \texttt{GasUsado} \text{ x } \texttt{PrecioDelGas}$$

Donde el gas usado es el total de gas consumido por la transacción y el precio del gas es el precio que se está dispuesto a pagar por la transacción. Este último es establecido por el usuario que quiere realizar la transacción (aunque existe un valor de mercado), y mientras mayor sea su valor, mayor prioridad tendrá para ser validada por los mineros, por lo que menor será el tiempo de validación. El precio del gas estándar es 20 gwei (1 ETH = 10^9 gwei), siendo el tiempo de validación de la transacción de aproximadamente 13 segundos [17].

El motivo por el que el precio de las transacciones se calcula en base al gas y no al *ether* se debe al valor volátil del último. Si se pagase en *ether* y su valor sube, el costo de la transacción aumentaría, pudiendo alcanzar valores insostenibles.

2.1.2. Cuentas

Uno de los componentes principales de Ethereum son las cuentas. Cada cuenta está representada por un arreglo de 20 bytes. Existen dos tipos:

- Cuentas de propiedad externa: cuentas de usuario, controladas por claves privadas.
- Cuentas de contrato: relacionadas a los contratos y controladas por su propio código.

La diferencia fundamental entre éstas es que las primeras pueden enviar mensajes a otras cuentas externas o a cuentas de contrato mediante la creación y firma de una transacción utilizando su clave privada. Un mensaje entre dos cuentas externas es simplemente una transferencia de valor a través de (*ether*). Pero un mensaje de una cuenta de propiedad externa a una de contrato activa el código de la cuenta del contrato, permitiéndole realizar las acciones para las que este esté programado (por ejemplo, transferir tokens, almacenar datos en la red, realizar algún cálculo o crear nuevos contratos) [18].

Por otra parte, las cuentas de contrato no pueden iniciar nuevas transacciones por su cuenta, solo pueden realizar transacciones en respuesta a otras transacciones que hayan recibido.

2.1.3. Tokens de Ethereum

Los tokens de Ethereum son estructuras producidas por un contrato inteligente con el objeto de representar unidades de valor, desde objetos físicos como el oro o la

propiedad de un terreno hasta monedas y servicios. Las propiedades y funciones de cada token están completamente sujetas al uso que se establezca para ellos.

Token ERC20

ERC20 es una interfaz estandarizada que garantiza la interoperabilidad entre tokens [19]. Los tokens ERC20 son un subconjunto de tokens de Ethereum que implementan un conjunto de funciones en común. De esta manera, los contratos inteligentes que desarrollen tokens que cumplan con este estándar podrán interactuar fácilmente entre sí y los tokens podrán ser intercambiados [20].

La particularidad de este tipo de token es que son fungibles, es decir, son divisibles o gastables. La fungibilidad es una de las propiedades más importantes del dinero.

Token ERC721

ERC721 es similar a ERC20 ya que ambos son estándares que garantizan la interoperabilidad entre tokens [21]. La principal diferencia es que los tokens ERC721 son no fungibles. Son utilizados para representar unidades de valor que no pueden ser divisibles: el voto en un sistema de elección, la propiedad de un inmueble o el registro clínico de un paciente son algunos ejemplos. Esto implica que también son únicos, es decir, cada token posee atributos específicos que pueden ser valorados de manera diferente según el intercambio y no siempre se pueden usar de manera intercambiable. Por esta razón, estos tokens son llamados coleccionables [22].

Al igual que los tokens ERC20, pueden ser intercambiados, y exige la implementación de las siguientes funcionalidades básicas:

- `name`: muestra el nombre del token.
- `symbol`: muestra el símbolo del token.
- `totalSupply`: retorna la cantidad de tokens de esta clase existentes en el *blockchain*.
- `balanceOf`: cantidad de tokens que posee una cuenta.
- `ownerOf`: retorna la cuenta que es dueña de un determinado token.
- `approve`: otorga permiso a otra cuenta para transferir el token.
- `takeOwnership`: es usada por una cuenta aprobada para obtener la posesión del token.
- `transfer`: transfiere un token

- `tokenOfOwnerByIndex`: muestra un token específico de un usuario.
- `tokenMetadata`: retorna un dato almacenado o un link externo correspondiente a un token.
- `Transfer`: no es una función, sino un evento, un mensaje que se emite a la red. En este caso, el mensaje se emite cuando se transfiere un token.
- `Approval`: evento emitido cuando se otorga permiso a una cuenta para transferir un token.

2.2. Metodología

El desarrollo de la aplicación consta fundamentalmente de dos partes: el *backend* y el *frontend*. El primero tiene relación con la codificación de los contratos inteligentes y su despliegue en el *blockchain*. El *frontend*, en cambio, es la parte de interfaz de usuario, es decir, el desarrollo de la página web. En la Figura 2.1 se observa un esquema de la aplicación, en donde se identifican el *backend* y el *frontend* y dónde son utilizadas las distintas herramientas descriptas en la sección 2.3.

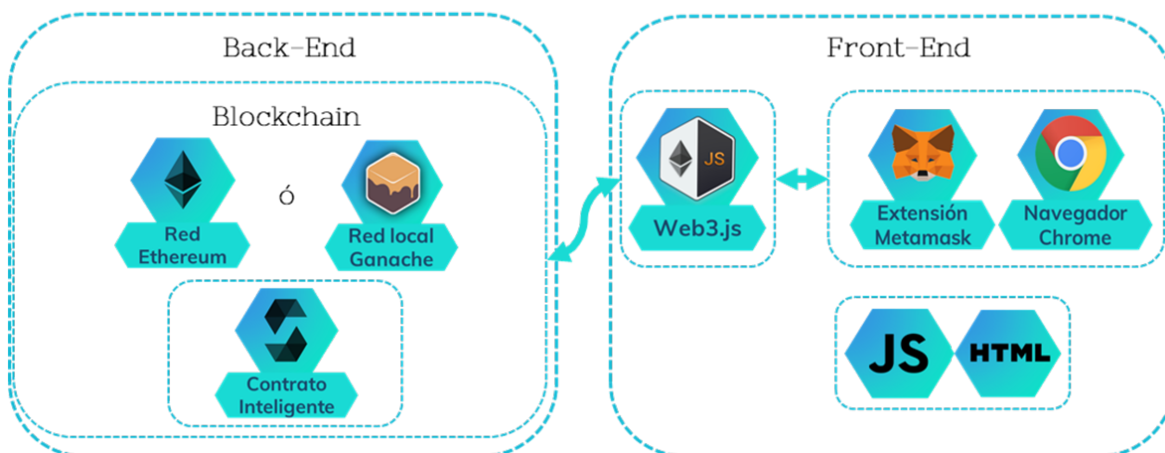

Figura 2.1: Esquema conceptual de una aplicación descentralizada.

2.3. Herramientas utilizadas

Esta sección tiene como objetivo realizar un breve análisis de los componentes que se requieren para llevar a cabo el desarrollo de la aplicación, de modo que permita tener una idea clara de cómo ensamblar un entorno de trabajo adecuado que cumpla con las características y requerimientos necesarios. En el apéndice A se detalla la instalación y configuración de estas herramientas.

Realizar el contrato inteligente y desplegarlo en el *blockchain* es poco conveniente, debido a que hay que pagar por cada transacción allí realizada. Por esta razón, para el desarrollo de este trabajo se utiliza una red virtual que simula todas las características del real. Esto incluye una cadena de bloques propia en la que se pueden tener distintas cuentas de usuario que se utilizarán para realizar diversos tipos de transacciones. A su vez, incluye un entorno de desarrollo que permite facilitar y agilizar algunas tareas como, por ejemplo, la compilación y el despliegue del contrato.

Se requiere también una serie de herramientas que permitan la ejecución de la aplicación web.

Lenguajes utilizados

- Solidity: es un lenguaje de alto nivel utilizado para la implementación de contratos inteligentes. Su sintaxis es similar a la de JavaScript y está enfocado específicamente a la EVM. Solidity está tipado de manera estática y acepta, entre otras cosas, herencias, bibliotecas y tipos complejos definidos por el usuario [23].

- HTML: es un lenguaje utilizado, junto con JavaScript y CSS, para el desarrollo de una página web. Describe la estructura básica de una página y organiza la forma en que se muestra su contenido, por ejemplo definiendo párrafos, tablas e imágenes [24].

- CSS: es un lenguaje de reglas en cascada que se utiliza para aplicar un estilo al contenido en HTML, por ejemplo colocando colores de fondo y fuentes [24].

- JavaScript: es un lenguaje de programación comúnmente utilizado en el desarrollo web. Es un lenguaje de *script* del lado del cliente, lo que significa que el código es procesado por el navegador web del cliente en lugar de hacerlo en el servidor. Permite mejorar la interfaz del usuario logrando que sean interactivas y creando contenido dinámico (por ejemplo animaciones complejas, botones pulsables y menus emergentes) [25]. En este proyecto, este lenguaje es utilizado en la programación de los tests de los contratos inteligentes y en el desarrollo del *frontend*, es decir, de la página web.

NodeJS

NodeJS es un entorno que permite ejecutar código JavaScript del lado del servidor. Se necesita esta característica porque ciertos archivos de configuración de los contratos inteligentes se escriben en JavaScript y NodeJS permite ejecutarlos de manera local [26].

Node Package Manager (NPM)

NPM es un gestor de paquetes que permite descargar e instalar módulos de JavaScript para poder utilizarlos en el desarrollo de programas. Este se ejecuta a través de la línea de comandos y permite manejar las dependencias para las aplicaciones [27].

Ganache

No es conveniente desplegar los contratos inteligentes directamente a Ethereum en la etapa de desarrollo, ya que implica un costo económico, por lo que una buena idea es realizar el despliegue en un *blockchain* de prueba. Si bien existen algunas redes de prueba de Ethereum (Rinkeby, Ropsten, etc.), es conveniente poder simular al *blockchain* en forma local, dado que permite incluso trabajar desconectado de Internet. Esto se puede conseguir con la herramienta Ganache, la cual posee dos características fundamentales: un *blockchain* privado y una serie de cuentas de usuario que permiten la interacción con el contrato inteligente.

Ganache permite iniciar rápidamente un *blockchain* personal de Ethereum de manera local, que puede usarse para desplegar contratos inteligentes sin tener que gastar *ether* real y ejecutar pruebas del mismo, inspeccionar el estado del *blockchain* (bloques, transacciones, cuentas, saldos, costos de gas), todo esto de manera sencilla [28].

Para este trabajo se lo utilizó en la ejecución de un *blockchain* privado de Ethereum de manera local en la cual se despliegan los contratos inteligentes, además de monitorear las transacciones y la creación de bloques.

Truffle

Truffle es un *framework* o entorno de trabajo que permite realizar el proceso de compilación, despliegue y testeo de los contratos inteligentes de una manera sencilla. Permite además la facilidad de estructuración y fácil manejo de datos, ya que cuenta con diferentes proyectos prearmados que pueden utilizarse como base para el proyecto propio, permitiendo trabajar tanto el *frontend* como el *backend*. La ejecución de Truffle es por medio de línea de comando.

Web3

La interfaz de usuario se desarrolla en los lenguajes JavaScript y HTML. Para que la aplicación interactúe con el contrato almacenado en el *blockchain*, se utiliza la biblioteca Web3.js. Esta, además de permitir llamadas a las funciones del contrato, posibilita obtener información más detallada de la red, como los eventos, las transacciones, el número de bloques, entre otros.

Webpack

Webpack es una herramienta que permite ciertas facilidades en el desarrollo del *frontend* de la aplicación. Esencialmente, es una herramienta de compilación que gestiona los códigos escritos en JavaScript, HTML, CSS, entre otros, y ofrece un servidor web para poder lanzar la aplicación.

Google Chrome

Naturalmente, se necesita un navegador web el cual permita mostrar la página web. Se utilizó Google Chrome debido a que es compatible con el complemento Metamask, el cual se describe a continuación.

Metamask

Metamask es un complemento compatible, entre otros, con el navegador Chrome, y hace de puente entre el *blockchain* Ethereum y el navegador, permitiendo interactuar con contratos inteligentes de una forma fácil. También funciona como una billetera, es decir, un gestor de cuentas con las cuales se puede interactuar con el *blockchain*. Metamask puede conectarse a cualquiera de las redes Ethereum, ya sea la red principal (MainNet), las redes de prueba (Rinkeby, Ropsten, etc.) o una red local. En este caso, Metamask se conecta con Ganache y gestiona las cuentas de usuario simuladas por dicho *blockchain*.

IPFS

IPFS (del inglés *Interplanetary File System*) es un sistema de archivos distribuido que recopila ideas exitosas de sistemas anteriores *peer–to–peer* como BitTorrent. Ofrece ventajas como la posibilidad de tener sitios web completamente distribuidos, incluyendo archivos o gran cantidad de datos.

Como este sistema es *peer–to–peer*, no existen nodos que lo conformen que sean privilegiados. Los nodos almacenan objetos IPFS de forma local, se conectan entre sí y transfieren estos objetos, los cuales representan archivos (como imágenes o vídeos) y otras estructuras de datos.

La particularidad que tienen los archivos almacenados bajo este sistema es que cada uno está codificado criptográficamente (es decir, hasheado). A diferencia del protocolo HTTP, la dirección del archivo está determinada por su contenido, es decir, por el hash, el cual es único.

IPFS tiene la particularidad de que cada nodo de la red almacena sólo el contenido en el que está interesado, y cierta información de indexación que ayuda a determinar

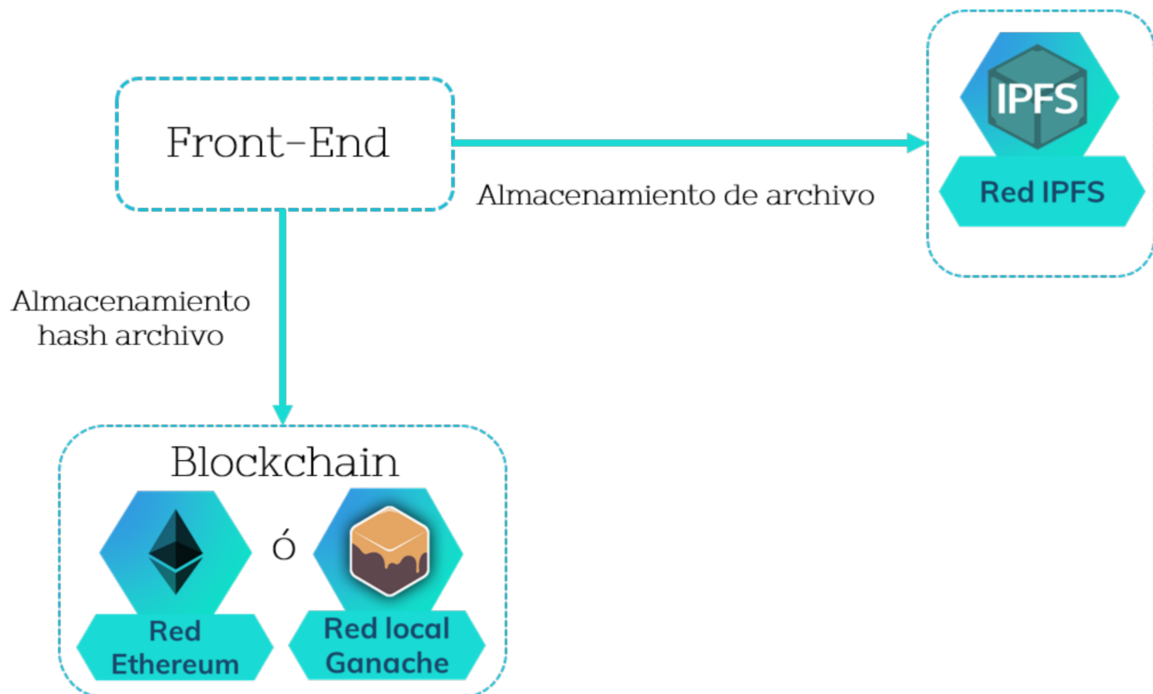

Figura 2.2: Esquema conceptual de la utilización de IPFS junto con *blokchain*.

quién está almacenando qué. Al buscar archivos, se está pidiendo a la red que encuentre los nodos que almacenen el contenido detrás del hash del archivo. Por ende, en lugar de un servidor, la comunicación se produce con un objeto específico y se está buscando una ruta de acceso a ese objeto. Este funcionamiento se diferencia al del protocolo HTTP, el cual el identificador es la ubicación del archivo o servidor [29].

Con IPFS es posible resolver el problema del almacenamiento de archivos de gran tamaño en las redes *blockchain*. De esta manera, los archivos se almacenan en el sistema IPFS, mientras que en la red *blockchain* sólo se guardan los hashes de los archivos, como se observa en la Figura 2.2

Capítulo 3

Gestión de títulos académicos

La emisión, validación y visualización de los datos clínicos de un paciente se puede asimilar al manejo de otros tipos de datos. En una primera instancia del proyecto, y con el objeto de adquirir experiencia en el manejo de la tecnología *blockchain* y las herramientas a utilizar, se desarrolló una aplicación descentralizada cuya función es la emisión, validación y visualización de títulos académicos de egresados del Instituto Balseiro.

En este capítulo se presenta una aplicación basada en la tecnología *blockchain* en la cual, a través del uso de contratos inteligentes, se puede asignar títulos académicos a estudiantes. Se identifican los requerimientos básicos, así como las soluciones para solventarlos. Se realiza el desarrollo del *backend* y el *frontend*, describiendo cada una de sus partes y la funcionalidad que cumplen de manera detallada.

3.1. Planteo del problema

El proceso de emisión de certificados universitarios es costoso y generalmente demora mucho tiempo, ya que implican el pago de personal y una serie de trámites. Estos trámites incluyen no solo la verificación de las materias cursadas y aprobadas del candidato al título en evaluación, sino también la firma de distintos niveles jerárquicos dentro de la universidad. Además, se requiere un nivel extra-universitario de gestión, a través de la verificación por parte de distintos estamentos a nivel ministerial nacional. Si bien las universidades nacionales han adoptado distintos sistemas informáticos, como los sistemas SIU Guaraní [30], todavía se requiere del soporte en papel del título.

Más aún, existen numerosas situaciones donde otras instituciones necesitan verificar el título de una persona, por lo que la demora en la emisión del título y su indisponibilidad en la red presentan cuellos de botella.

Por otro lado, se han registrado numerosos casos de falsificación de títulos académicos. La adulteración del título de abogado en la UBA [31] y del título de grado en

ingeniería mecánica en la UTN [32] son algunos ejemplos. Aunque es penado por la Ley, es relativamente sencillo adulterar un documento público, y por la recompensa algunas personas deciden asumir los riesgos.

La causa común de estos problemas es que el sistema de validación de certificados es gestionado por personas y sistemas de información centralizados, siendo ineficiente a lo largo de todo el proceso y generando una confianza desmesurada en estos, volviendo el sistema susceptible a actos de corrupción. Todo esto produce la pérdida de confianza en los títulos en papel, así como otras dificultades asociadas. Por tal motivo, es conveniente buscar otra metodología que permita moderar las dificultades que estas situaciones implican.

3.2. Solución propuesta

La aplicación desarrollada busca mitigar la problemática planteada. La solución propuesta es gestionar los títulos por medio de la tecnología *blockchain*, específicamente utilizando un contrato inteligente, de manera tal de aprovechar su ventaja de inmutabilidad y trazabilidad de los datos almacenados en esa red. De esta manera, se desarrolla una aplicación que permita al Instituto Balseiro el registro, validación y visualización de los títulos académicos de sus egresados. En una primera instancia, funcionaría como respaldo a los títulos en papel.

Una de las ventajas que proporciona el registrar los títulos usando los contratos inteligentes es la inmutabilidad, ya que al almacenar los datos en el *blockchain* estos no pueden ser alterados, lo que a su vez permite tener acceso público a dichos datos. Esto es, cualquier persona o sistema será capaz de acceder a la red para visualizarlos. Otra ventaja sería la rapidez con la que se validaría el título, ya que si un estudiante se gradúa y obtiene su título profesional, el registro de este será prácticamente inmediato, lo cual resulta útil tanto para trámites relacionados con la vida laboral o afines a estudios de un grado más alto.

3.3. Objetivo

El objetivo es diseñar y desarrollar un prototipo funcional de aplicación descentralizada basada en la tecnología *blockchain* para la emisión, validación y visualización de certificados académicos digitales, mediante el uso de contratos inteligentes. En particular, se debe:

- Diseñar los contratos inteligentes que permitan el manejo de los datos, que en este caso son los certificados digitales.
- Diseñar una interfaz que permita a los usuarios interactuar con los certificados.

- Realizar las pruebas correspondientes para asegurar la ausencia de errores en el *software*.

3.4. Requerimientos

Se enumeran a continuación los requerimientos básicos que debe cumplir la aplicación a desarrollar:

- Inmutabilidad de los datos: la información correspondiente a los títulos académicos debe estar registrada en una red descentralizada, principalmente para asegurar la inmutabilidad de los datos.
- Disponibilidad de los datos: la información de los certificados emitidos, de los estudiantes y de las autoridades debe ser visible para todo el público, de modo tal que cualquiera pueda consultar la información.
- Roles: se debe asegurar que solo las autoridades puedan validar los certificados digitales. Por esto, deben existir distintos tipos de usuarios.
- Flexibilidad: la funcionalidad de la aplicación debe ser primitiva. Sin embargo, se debe diseñar una estructura y una lógica de funcionamiento que permita en un futuro añadir características nuevas.

De esta manera, se identifican los requerimientos específicos que la aplicación debe cumplir:

1. Registrar datos básicos de las carreras universitarias del Instituto Balseiro. Esto solo puede realizarlo alguna autoridad de la institución.
2. Modificar los datos de una carrera. Esto solo puede realizarlo alguna autoridad de la institución. No se pueden modificar carreras que hayan sido validadas.
3. Validar los datos de una carrera anteriormente añadida. Esta función debe ser accesible exclusivamente por la autoridad máxima de la institución.
4. Visualizar información de una carrera universitaria que esté registrada.
5. Registrar datos básicos de los títulos académicos. Esto solo puede realizarlo alguna autoridad de la institución.
6. Modificar los datos de un título. Esto solo puede realizarlo alguna autoridad de la institución. No se pueden modificar títulos que hayan sido validados.

7. Validar los datos de un título anteriormente añadido. Esta función debe ser accesible exclusivamente por la autoridad máxima de la institución.
8. Además de estar referenciados en la información del títulos registrado en la red, los egresados deben ser dueños de su título.
9. Debe existir un único registro por cada título.
10. Visualizar información de un título académico que haya sido emitido.
11. Visualizar los usuarios correspondientes a las autoridades de la institución responsables del registro y validación de los títulos.

3.5. Desarrollo

3.5.1. Diseño de la aplicación

En base a los requerimientos especificados, se desarrolló un prototipo funcional de un sistema distribuido para la emisión, visualización y verificación de certificados educativos digitales utilizando la tecnología *blockchain*, específicamente la red Ethereum, donde se tiene en cuenta las necesidades de los distintos actores (estudiantes, universidades y entidades que deseen visualizar los títulos). Mediante la implementación de contratos inteligentes, se añaden en Ethereum los títulos académicos de los egresados, de modo que estos queden registrados en la red y, al ser validados, no puedan ser modificados. A su vez, estos datos pueden ser consultados con relativa facilidad.

Se diseñó la aplicación para cuatro tipos de usuarios:

- Director: es la máxima autoridad del instituto y responsable de la validación del certificado.
- Secretario: entidad responsable del registro del título en la red. Es una autoridad del instituto y es designada por el director.
- Egresado: es la entidad dueña del título.
- Público: pueden visualizar los títulos.

En primer lugar, la aplicación debe asegurar que sólo las cuentas de secretario y director puedan emitir y validar el título, respectivamente. Es en este punto en que se plantea de qué manera se pueden representar y gestionar los títulos en un contrato inteligente. La solución a este problema la tiene el estándar ERC721, el cual se detalló en la sección 2.1.

3.5.2. Backend: creación del contrato inteligente

El desarrollo del *backend* consiste en la codificación, compilación, despliegue y testeo del contrato.

El primer paso es codificar el contrato inteligente. Aquí se realizan las funciones y se declaran las variables y estructuras de datos necesarias que almacenan los datos en el *blockchain*.

Una vez escrito el código, el siguiente paso es compilarlo con la ayuda de Truffle, el cual descarga y utiliza el compilador de Solidity. Mediante el comando `truffle compile`, se compilan los contratos inteligentes escritos en lenguaje Solidity.

Una vez que se logra la compilación sin errores se procede al despliegue, que consiste en enviar el código del contrato al *blockchain* de prueba Ganache, nuevamente por medio de Truffle, utilizando el comando `truffle migrate`. Al momento de desplegar el contrato, se crearán transacciones en bloques que serán minados y adheridos a la cadena de bloques.

De esta manera, en este punto se puede interactuar con el contrato por medio de Truffle y la consola de comandos. Sin embargo, en este proyecto se utilizó un método más efectivo para probar su funcionalidad y comprobar que no tiene errores. Este método consiste en desarrollar un código a partir de *pruebas unitarias* (*unit test* en inglés). Para ello, se escribe un código en lenguaje JavaScript que interactúa con el contrato llamando a sus funciones y comprobando sus salidas. El objetivo es *atacar* al contrato de distintas formas y asegurarse de que cada función del contrato funciona perfectamente. Este paso es muy importante ya que, una vez que la aplicación esté lista y se despliegue el contrato en el *blockchain* real de Ethereum, este no puede ser modificado nunca más. La ejecución de las pruebas codificadas en archivos JavaScript se realiza mediante el comando `truffle test`.

3.5.2.1. Codificación del contrato

Importación de archivos

En primer lugar, se importan los archivos externos necesarios. El lenguaje Solidity soporta declaraciones de importación de archivos, las cuales permiten ampliar la funcionalidad del contrato con las características de los contratos importados. En este caso, se importan dos contratos: `Ownable.sol` y `ERC721.sol`, ambos de la biblioteca OpenZeppelin [33].

Esta biblioteca cuenta con una serie de contratos inteligentes estandarizados que aliviana la labor de los desarrolladores. Cuenta con una API estable, lo que significa que los contratos que utilizan esta biblioteca seguirán estando vigentes al actualizar a una nueva versión. Los contratos de esta biblioteca implementan funcionalidades

comunes en los proyectos de Ethereum, lo que significa que están ampliamente probados y revisados.

El contrato `Ownable.sol` posee las funciones relacionadas al dueño del contrato. El mismo añade una nueva entidad, la cual es el dueño del contrato, e implementa funciones para la gestión de esta entidad. Por otro lado, el contrato `ERC721.sol` posee la implementación del token ERC721 detallada en la sección 2.1.

Variables y estructuras de datos

La elección de las variables de estado y estructuras a utilizar es una parte importante del desarrollo. Si bien los datos almacenados en el *blockchain* tienen la ventaja de la inmutabilidad, el precio a pagar por ello es relativamente elevado. Por esta razón, es necesario que el uso de este recurso sea eficiente.

Para el almacenamiento de datos, Solidity brinda varios tipos elementales que pueden ser combinados para crear estructuras más complejas. Los tipos que se utilizan en este contrato son [34]:

- Booleano o `bool`: los posibles valores son `true` y `false`.
- Entero
  - `int`: entero con signo. Se pueden asignar tamaños variables, desde 8 hasta 256 bits en pasos de 8, aclarando el valor a continuación de `int`. Por ejemplo, `int32` es un entero con signo de 32 bits. `int` es un alias para `int256`.
  - `uint`: entero sin signo. La aclaración para `int` es equivalente en este caso.
- Dirección o `address`: contiene un valor de 20 bytes y se utilizan para almacenar direcciones de Ethereum.
- `string`: cadena de caracteres UTF-8-codificado de tamaño dinámico.

Luego, las estructuras de datos son:

- Estructura o `struct`: son tipos de datos construidos utilizando otros tipos.
- Mapeo o `mapping`: son tipos de datos declarados como `mapping(Key => Value)`. `Key` es la clave del `mapping` y puede ser casi cualquier tipo de dato. `Value` es el valor del `mapping` y puede ser casi cualquier tipo de dato, incluso otro `mapping`. Esta estructura puede ser vista como tabla hash, aunque en este caso la clave no es realmente almacenada en el `mapping`, sólo su hash. Por esto, los `mappings` no tienen un *length* o un concepto de fijar clave o valor, implicando que no pueda encontrarse la clave a partir del valor, pero sí en sentido inverso.

A continuación se muestran y describen las variables y estructuras utilizadas para almacenar los datos ingresados:

- `address director, secretary`

  Descripción: variables tipo `address` que almacenan las direcciones del director y el secretario, respectivamente.

- `uint careerId`

  Descripción: es utilizada para asignar un *id* a las carreras que se vayan registrando. Sirve también para llevar un registro de la cantidad de carreras registradas.

- `uint titleId`

  Descripción: es utilizada para asignar un *id* a los títulos que se vayan registrando. Sirve también para llevar un registro de la cantidad de títulos registrados.

- ```
  struct Career {
      string nameCareer;
      string dataCareer;
      bool isValidCareer;
  }
  ```

  Descripción: estructura de datos que almacena la información de una carrera.
  `nameCareer`: nombre de la carrera,
  `dataCareer`: título de la carrera,
  `isValidCareer`: `bool` que determina si la carrera está validada o no.

- ```
  struct Title {
      string nameStudent;
      string dniStudent;
      uint toCareerId;
      bool isValidTitle;
  }
  ```

  Descripción: estructura de datos que almacena la información de un título.
  `nameStudent`: nombre del estudiante,
  `dniStudent`: dni del estudiante,
  `toCareerId`: *id* de la carrera correspondiente,
  `isValidTitle`: `bool` que determina si el título está validado o no.

- `mapping (uint => Career) _idToCareer`

  Descripción: relaciona un *id* de carrera con la estructura que almacena su información.

- `mapping (uint => Title) _idToTitle`

  Descripción: relaciona un *id* de título con la estructura que almacena su información.

Modificadores de funciones

Un modificador de función es una propiedad que permite cambiar el comportamiento que tienen las funciones del contrato de una manera ágil. Permiten comprobar el cumplimiento de una condición antes de ejecutar la función [35]. En este caso se han creado dos modificadores, los cuales cada uno comprueba si la cuenta que ejecuta la función es la del director o del secretario. Los mismos se denominan `onlyDirector` y `onlySecretary`, respectivamente. Por ejemplo, sea la función:

```
function setDirector(address _director) external onlyDirector
```

Puede observarse que en la declaración de esta función está expresado el modificador `onlyDirector`, por lo que se comprobará que el usuario que la llame sea efectivamente el director.

Constructor

Un constructor es una función que se ejecuta exclusivamente cuando el contrato es desplegado en el *blockchain*. En este caso, es utilizada para establecer las cuentas de director y secretario, asignándolas al dueño del contrato, es decir, al que lo desplegó.

Funciones

Con el objeto de cumplir con los requerimientos expuestos en 3.4, se implementaron en el contrato inteligente una serie de funciones. En la siguiente lista se enumeran las funciones desarrolladas, describiendo en cada una los argumentos de entrada y de salida, el tipo de usuario que tiene acceso y una breve descripción.

- `function setDirector(address _director) external onlyDirector`

  Descripción: asigna la cuenta que juega el rol de director en el contrato inteligente. El director al inicio del contrato es el que lo desplegó.

Tipo de usuario con acceso: Director.

- `function getDirector() external view returns (address)`

  Descripción: retorna la dirección del director.

  Tipo de usuario con acceso: Público.

- `function setSecretary(address _secretary) external onlyDirector`

  Descripción: asigna la cuenta que juega el rol de secretario en el contrato inteligente. El secretario al inicio del contrato es el que lo desplegó.

  Tipo de usuario con acceso: Director.

- `function getSecretary() external view returns (address)`

  Descripción: retorna la dirección del secretario.

  Tipo de usuario con acceso: Público.

- `function addCareer(string _nameCareer, string _dataCareer) external onlySecretary`

  Descripción: registra en el *blockchain* los datos básicos de una carrera universitaria, específicamente su nombre y su titulación. Esto lo hace utilizando la estructura `Career`. Le otorga a la carrera un número identificador utilizando la variable `careerId`.

  Tipo de usuario con acceso: Secretario.

- `function updateCareer(uint _careerId, string _nameCareer, string _dataCareer) external onlySecretary`

  Descripción: permite modificar una carrera ingresando su *id*, siempre y cuando la carrera no haya sido validada.

  Tipo de usuario con acceso: Secretario.

- `function validateCareer(uint _careerId, string _nameCareer, string _dataCareer) external onlyDirector`

  Descripción: valida los datos de una carrera anteriormente añadida, utilizando la variable `isValidCareer`.

Tipo de usuario con acceso: Director.

- `function numberOfCareers() external view returns (uint)`

  Descripción: retorna la cantidad de carreras existentes en el contrato inteligente.

  Tipo de usuario con acceso: Público.

- `function addTitle(uint _toCareerId, string _nameStudent, string _dniStudent, string _uri, address _student) external onlySecretary`

  Descripción: registra en el *blockchain* los datos de un título universitario, específicamente el nombre y dni del egresado, la carrera e información extra del certificado. Esto lo hace utilizando la estructura `Career`. Le otorga al título un número identificador utilizando la variable `titleId` y crea el token ERC721 representativo, transfiriéndolo a la cuenta del estudiante.

  Tipo de usuario con acceso: Secretario.

- `function updateTitle(uint _titleId, uint _toCareerId, string _nameStudent,string _dniStudent,string _uri)external onlySecretary`

  Descripción: permite modificar un título ingresando su *id*, siempre y cuando este no haya sido validado.

  Tipo de usuario con acceso: Secretario.

- `function validateTitle(uint _titleId, uint _toCareerId, string _nameStudent, string _dniStudent,string _uri)external onlyDirector`

  Descripción: valida los datos de un título anteriormente añadido, utilizando la variable `isValidToken`.

  Tipo de usuario con acceso: Director.

- `function numberOfTitles() external view returns (uint)`

  Descripción: retorna la cantidad de títulos existentes en el contrato.

  Tipo de usuario con acceso: Público.

- `function getTitlesByOwner(address _student) external view returns(uint[])`

Descripción: retorna los títulos (tokens) que posee una cuenta.

Tipo de usuario con acceso: Público.

- `function dataCareer(uint _careerId) external view returns(string, string, bool)`

  Descripción: retorna los datos de una carrera.

  Tipo de usuario con acceso: Público.

- `function dataTitle(uint _titleId) external view returns (string, string, uint, string, bool, address)`

  Descripción: retorna los datos de un título.

  Tipo de usuario con acceso: Público.

Eventos

La emisión de eventos es una herramienta de Ethereum que facilita la comunicación entre el contrato inteligente y la interfaz de usuario. Cuando una transacción es realizada, el contrato puede emitir un evento que el *frontend* puede recibir y procesar. En este contrato, los eventos implementados están relacionados con el registro y la validación de carreras y títulos, emitiendo información, a través parámetros, como su *id* y sus datos básicos. Estos son:

- `event NewDirector(address previousDirector, address newDirector)`

  Descripción: se ha establecido la dirección del director.

  Parámetros: `address previousDirector`: dirección anterior,
  `address newDirector`: nueva dirección.

- `event NewSecretary(address previousSecretary, addressnewSecretary)`

  Descripción: se ha establecido la dirección del secretario.

  Parámetros: `address previousSecretary`: dirección anterior,
  `address newSecretary`: nueva dirección.

- `event CareerAdded(uint careerId, string nameCareer, string dataCareer)`

Descripción: una carrera ha sido registrada.

Parámetros: `uint careerId`: *id* de la carrera,
`string nameCareer`: nombre de la carrera,
`string dataCareer`: titulación de la carrera.

- `event CareerUpdated(uint careerId, string nameCareer, string dataCareer)`

  Descripción: una carrera ha sido modificada.

  Parámetros: `uint careerId`: *id* de la carrera,
  `string nameCareer`: nombre de la carrera,
  `string dataCareer`: titulación de la carrera.

- `event CareerValidated(uint careerId, string nameCareer, string dataCareer)`

  Descripción: una carrera ha sido validada.

  Parámetros: `uint careerId`: *id* de la carrera,
  `string nameCareer`: nombre de la carrera,
  `string dataCareer`: titulación de la carrera.

- `event TitleAdded(uint titleId, string nameStudent, string dniStudent, string nameCareer, string uri, address student)`

  Descripción: un título ha sido registrado.

  Parámetros: `uint titleId`: *id* del título,
  `string nameStudent`: nombre del estudiante,
  `string dniStudent`: dni del estudiante,
  `string nameCareer`: nombre de la carrera correspondiente,
  `string uri`: uri del título,
  `address student`: dirección del estudiante.

- `event TitleUpdated(uint titleId, string nameStudent, string dniStudent, string nameCareer, string uri, address student)`

  Descripción: un título ha sido modificado.

Parámetros: `uint titleId`: *id* del título,
`string nameStudent`: nombre del estudiante,
`string dniStudent`: dni del estudiante,
`string nameCareer`: nombre de la carrera correspondiente,
`string uri`: uri del título,
`address student`: dirección del estudiante.

- `event TitleValidated(uint titleId, string nameStudent, string dniStudent, string nameCareer, string uri, address student)`

  Descripción: un título ha sido validado.

  Parámetros: `uint titleId`: *id* del título,
  `string nameStudent`: nombre del estudiante,
  `string dniStudent`: dni del estudiante,
  `string nameCareer`: nombre de la carrera correspondiente,
  `string uri`: uri del título,
  `address student`: dirección del estudiante.

3.5.2.2. Despliegue del contrato

Configuración de la red Ganache

Para poder desplegar el contrato, el *blockchain* Ganache debe estar funcionando. Además de la red, esta herramienta genera 10 cuentas con sus respectivas claves públicas y privadas y 100 ETH iniciales. Esto se puede observar en la Figura 3.1.

Migración del contrato

A continuación, se despliega el contrato mediante el comando `truffle migrate`. En este caso, Truffle se ocupa de registrar el contrato y devolver su dirección en el *blockchain*. Por otra parte, Truffle expone el método `artifact.require()` al *frontend*. Este método funciona en forma similar al `require()` usual de JavaScript, pero además expone la abstracción del contrato para que pueda ser utilizada por el *frontend*.

Si bien las últimas versiones de Truffle brindan mucha información a la hora de comprobar si el despliegue del contrato ha sido realizado con éxito, se puede inspeccionar el *blockchain* en Ganache. En la Figura 3.2 se expone la sección *Blocks*, donde se observan los bloques que han sido minados con el gas total utilizado y el número de transacciones que contienen.

Figura 3.1: Interfaz de la herramienta Ganache. Se pueden observar las direcciones de las cuentas generadas y la cantidad de ETH disponible para cada una.

Figura 3.2: Interfaz de la sección *Blocks* de Ganache. Se destacan el gas total utilizado y el número de transacciones que contiene cada bloque.

Al ingresar a cada uno de los bloques, se puede ver información más detallada del bloque minado y de las transacciones que contiene, como el gas usado, el hash, el tipo y el emisor de la transacción. Esto se muestra en la Figura 3.3.

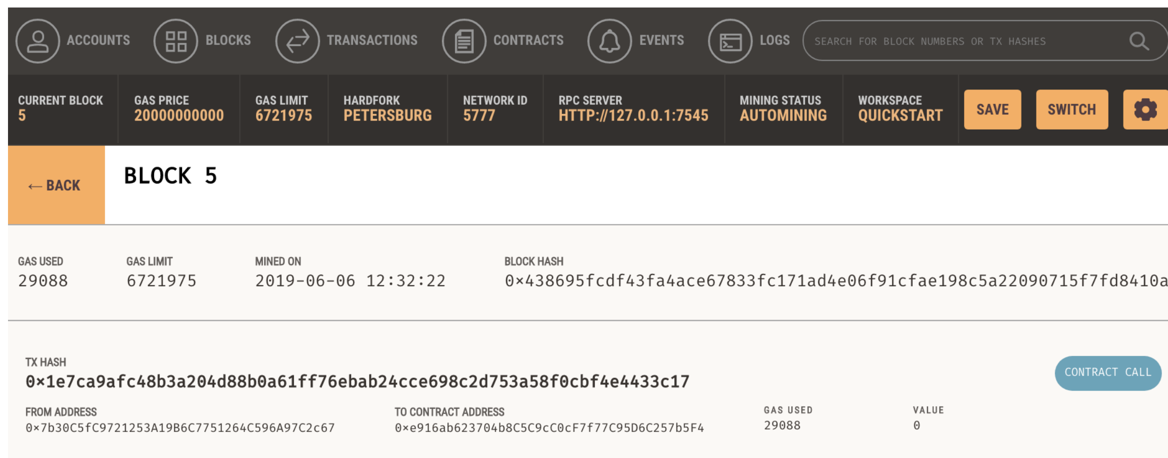

Figura 3.3: Transacción contenida en un bloque. Se detalla el gas usado, el hash, el tipo y el emisor de la transacción.

3.5.2.3. Pruebas unitarias del contrato

Las pruebas unitarias consistieron en verificar la funcionalidad de cada línea del código. Para eso, se crea un programa en lenguaje JavaScript conteniendo una serie de funciones que interactúan con el contrato inteligente. Específicamente, el programa realizado cumple con los siguientes propósitos, comprobados a partir de pruebas unitarias:

- Verifica que el dueño del contrato es el que lo desplegó.
- Interactúa con las funciones `setDirector` y `setSecretary` estableciendo el director y el secretario del contrato, respectivamente.
- Registra carreras con `addCareer`.
- Valida carreras con `validateCareer`.
- Interactúa con `updateCareer` modificando carreras. Verifica que sólo pueden modificarse carreras que no han sido validadas.
- Registra títulos con `addTitle`.
- Valida títulos con `validateTitle`.
- Interactúa con `updateTitle` modificando títulos. Verifica que sólo pueden modificarse títulos que no han sido validados.
- Para cada función, verifica que sólo las direcciones permitidas pueden acceder. Esto es:
    - Sólo el secretario puede registrar carreras con `addCareer` y modificarlas con `updateCareer`, y únicamente el director puede validarlas con `validateCareer`.

   - Sólo el secretario puede registrar títulos con `addTitle` y modificarlos con `updateTitle`, y únicamente el director puede validarlos con `validateTitle`.

- Verifica los valores de retorno de las funciones: `getDirector`, `getSecretary`, `numberOfCareers`, `numberOfTitles`, `getTitlesByOwner`, `dataCareer` y `dataTitle`.

- Observa todos los eventos emitidos por el contrato, comprobando la correcta información de los parámetros.

Para ejecutar las pruebas unitarias se utiliza el comando `truffle test`, que permite lanzar todas las pruebas descriptas anteriormente. En la figura 3.4 se muestran los resultados del *test*.

```
Compiling your contracts...
===========================
> Everything is up to date, there is nothing to compile.


  Contract: Titulos IB test
    Titulos IB test
      Titulos info
        ✓ Owner is accounts[0] (57ms)
        ✓ Owner is director
        ✓ Owner is secretary
        ✓ Director can set director (71ms)
        ✓ Director can set secretary (44ms)
        ✓ Career is added (91ms)
        ✓ Another career is added (61ms)
        ✓ Get Number of careers
        ✓ Career is updated (61ms)
        ✓ Career is validated (77ms)
        ✓ Another career is validated (85ms)
        ✓ Career 3 is added (48ms)
        ✓ Career 3 is updated (54ms)
        ✓ Career 3 is validated (126ms)
        ✓ Get Number of careers
        ✓ Title is added (113ms)
        ✓ Another title is added (135ms)
        ✓ Get Number of titles with numberOfTitles
        ✓ Get Number of titles with totalSupply
        ✓ Title is updated (72ms)
        ✓ Title is validated (87ms)
        ✓ Another title is validated (69ms)
        ✓ Title 3 is added (160ms)
        ✓ Title 3 is updated (72ms)
        ✓ Title 3 is validated (64ms)
        ✓ Get Number of titles with numberOfTitles
        ✓ Get Number of titles with totalSupply (40ms)
        ✓ I should have a title (191ms)
        ✓ Career 1 is ok
        ✓ My title is ok


  30 passing (2s)
```

Figura 3.4: Resultados del *test* realizado. Se han probado la totalidad de las funciones programadas.

3.5.3. Frontend: creación de la página web

3.5.3.1. Codificación de la página web

Para que el usuario pueda interactuar con las funciones del contrato, en el código JavaScript de la página se desarrollaron funciones análogas que interactúan con las

del contrato. Además, se realizan funciones relacionadas a obtener información de la cuenta y de la red. Estas son:

- `renderCareer`

  Descripción: muestra en una tabla todas las carreras registradas en el `blockchain`. Para eso, utiliza la función `dataCareer` del contrato.

- `renderTitle`

  Descripción: muestra en una tabla todos los títulos registrados en el `blockchain`. Para eso, utiliza la función `dataTitle` del contrato.

- `setProvider`

  Descripción: configura web3 con Metamask y el nodo local Ganache.

- `showNetwork`

  Descripción: muestra la información de la red `blockchain` con la cual está conectada. Para eso, se utiliza `web3.eth.net.getNetworkType()`.

- `showCurrentBlockNumber`

  Descripción: muestra el número de bloques existentes en la red `blockchain`, utilizando `web3.eth.getBlockNumber()`.

- `showAccounts`

  Descripción: muestra la dirección del usuario. Para eso, se utiliza la función de Web3 `web3.eth.getAccounts()`.

- `showGetBalance`

  Descripción: muestra el número de *ethers* que posee el usuario. Para ellos, se usa la función de Web3 `web3.eth.getBalance()`.

A su vez, se escriben los códigos en HTML y CSS que proporcionan la estructura visual de la página.

3.5.3.2. Configuración de las herramientas

Una vez realizado el programa, se lanza Webpack mediante el comando `npm run dev` en la terminal. En la Figura 3.5 se observa que Webpack se ha ejecutado correctamente y corre en `http://localhost:8081`.

```
(base) agustin@agustin-N552VW:~/blockchain/ibtrials/titulosIB$ npm run dev

> titulosib@1.0.0 dev /home/agustin/blockchain/ibtrials/titulosIB
> webpack-dev-server

ℹ ｢wds｣: Project is running at http://localhost:8081/
ℹ ｢wds｣: webpack output is served from /
ℹ ｢wds｣: Content not from webpack is served from /home/agustin/blockchain/
ibtrials/titulosIB
⚠ ｢wdm｣: Hash: 6fda4e731d9800d519ce
Version: webpack 4.33.0
Time: 7091ms
Built at: 06/19/2019 9:58:18 PM
           Asset      Size  Chunks                    Chunk Names
      index.html  19.3 KiB          [emitted]
    titulosIB.js  2.05 MiB       0  [emitted]  [big]  main
titulosIB.js.map   5.5 MiB       0  [emitted]         main
Entrypoint main [big] = titulosIB.js titulosIB.js.map
[114] (webpack)-dev-server/client/utils/log.js 964 bytes {0} [built]
[214] multi (webpack)-dev-server/client?http://localhost ./app/javascript/
titulosIB.js 40 bytes {0} [built]
[215] (webpack)-dev-server/client?http://localhost 4.29 KiB {0} [built]
[216] ./node_modules/strip-ansi/index.js 161 bytes {0} [built]
[218] (webpack)-dev-server/client/socket.js 1.04 KiB {0} [built]
[221] (webpack)-dev-server/client/overlay.js 3.59 KiB {0} [built]
[227] (webpack)-dev-server/client/utils/sendMessage.js 402 bytes {0} [buil
t]
[228] (webpack)-dev-server/client/utils/reloadApp.js 1.63 KiB {0} [built]
[230] (webpack)-dev-server/client/utils/createSocketUrl.js 2.77 KiB {0} [b
uilt]
[235] (webpack)/hot sync nonrecursive ^\.\/log$ 170 bytes {0} [built]
[237] ./app/javascript/titulosIB.js 43.5 KiB {0} [built]
[238] ./node_modules/babel-runtime/regenerator/index.js 49 bytes {0} [buil
t]
[241] ./node_modules/babel-runtime/helpers/asyncToGenerator.js 906 bytes {
0} [built]
[278] ./node_modules/web3/src/index.js 2.01 KiB {0} [built]
[490] ./node_modules/truffle-contract/index.js 437 bytes {0} [built]
    + 584 hidden modules
ℹ ｢wdm｣: Compiled successfully.
```

Figura 3.5: Resultado al ejecutar el comando `npm run dev` en la terminal, observándose que Webpack se ha ejecutado correctamente.

Asimismo, se abre el navegador Chrome mostrando la página web que se ha desarrollado, que permite probar la funcionalidad completa de la aplicación. Es indispensable para ello que el navegador tenga el complemento Metamask instalado, y que el usuario esté familiarizado con el mecanismo de firma de trasacciones en Ethereum. En la Figura 3.6 se observa la página web de la aplicación junto con el complemento Metamask desplegado.

3.6. Observaciones

Se desarrolló una aplicación para el manejo de los títulos académicos del Instituto Balseiro utilizando la plataforma Ethereum. Se planteó la funcionalidad básica, regis-

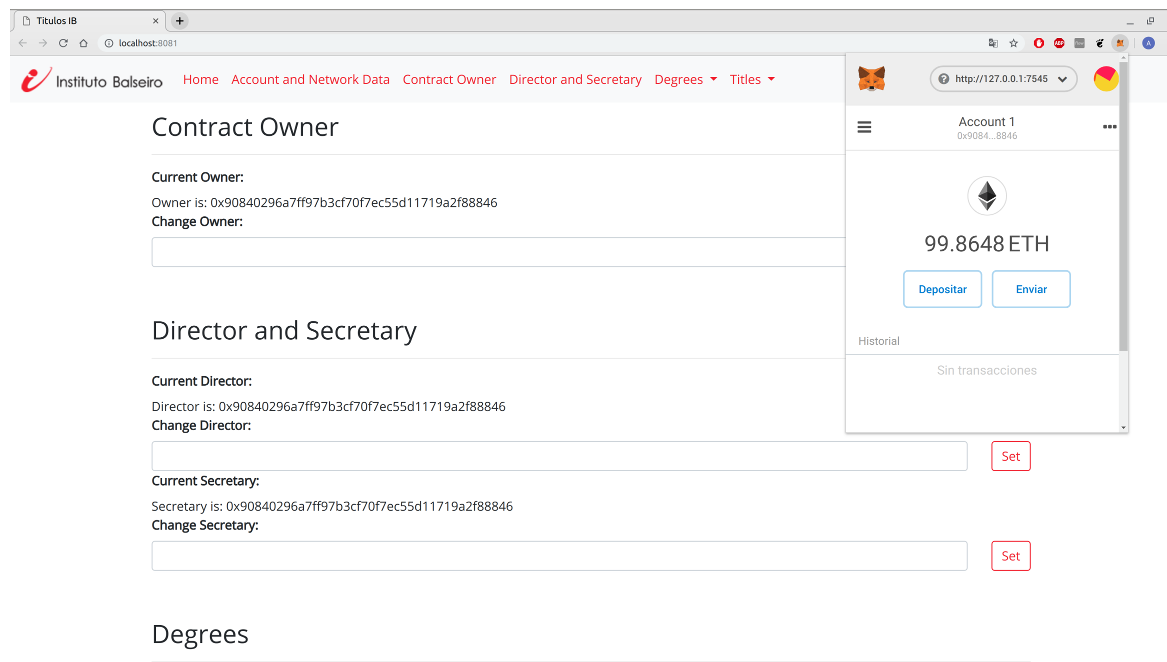

Figura 3.6: Página web de la aplicación y complemento Metamask desplegado.

trando sólo los datos básicos de las carreras y los títulos e implementando el token ERC721 para la gestión de los últimos. Sin embargo, esta estructura es flexible y sirve como base para futuros desarrollos de la aplicación. Algunas de las características que podrían añadirse son:

- Gestionar las cuentas de los egresados, registrando y validando las mismas en el contrato inteligente.
- Añadir nuevos datos de las carreras y los títulos. Incluso podrían registrarse datos de las materias y utilizar la plataforma al igual que los sistemas SIU Guarani.
- Implementar el sistema IPFS para asociar documentos como imágenes y PDF a los títulos. Este sistema es explicado y utilizado en el capítulo 4 bajo un contexto similar.

Capítulo 4

Sistema de gestión de ensayos clínicos

En este capítulo se presenta el trabajo llevado a cabo para desarrollar la aplicación descentralizada de gestión de ensayos clínicos en el *blockchain* de Ethereum. De forma similar al capítulo anterior, se detallan los requerimientos del sistema, el diseño y la implementación del *frontend* y del *backend*, las pruebas unitarias realizadas y una breve demostración de las funcionalidades del código.

4.1. Requerimientos

A continuación se establecen los requerimientos básicos que la aplicación a desarrollar debe cumplir:

- Inmutabilidad de los datos: los datos clínicos de los pacientes deben almacenarse en una red descentralizada, de tal modo que se asegure la inmutabilidad de los mismos.
- Disponibilidad de los datos: la información del ensayo clínico, del laboratorio que lo promueve y de las autoridades reguladoras debe ser de dominio público. Asimismo, los datos médicos de los pacientes deben ser visibles solo para usuarios que estén autorizados.
- Roles: debe asegurarse la existencia de distintos tipos de usuarios. Por un lado, deben estar representados el laboratorio que promueve el ensayo y las autoridades reguladoras que revisan la información. Por otro lado, deben existir los roles correspondientes a los pacientes y médicos, dueños de sus datos clínicos.
- Flexibilidad: la funcionalidad de la aplicación debe ser primitiva. Sin embargo, la estructura y su lógica de funcionamiento deben estar diseñadas de tal modo que puedan añadirse características nuevas en un futuro.

De esta manera, se identifican los requerimientos específicos que la aplicación debe cumplir:

1. Añadir usuarios a la red. Registrar los datos de los mismos y asociarlos al rol correspondiente. Esto sólo lo pueden realizar otros usuarios que estén registrados y validados en la red.

2. Modificar los datos de un usuario. Esto solo puede realizarlo la cuenta que registró originalmente el usuario. No se pueden modificar usuarios que hayan sido validados.

3. Validar los datos de un usuario anteriormente añadido. Esta función debe ser accesible exclusivamente por el laboratorio.

4. Visualizar información de un usuario que esté registrado.

5. Registrar la información del ensayo clínico. Esto solo lo puede realizar el laboratorio.

6. Visualizar la información del ensayo clínico.

7. Cada paciente debe estar asociado a un médico, el cual es el responsable del control de sus datos clínicos.

8. Registrar la ficha técnica de un paciente. Esto solo puede realizarlo el médico responsable del paciente. La ficha técnica puede ser modificada por el mismo médico.

9. Visualizar la ficha técnica de un paciente.

10. Añadir un dato clínico de un paciente. Esto solo puede realizarlo el médico responsable del paciente.

11. Para cada paciente debe existir un historial médico, formado por los datos clínicos que fueron registrados al mismo.

12. Visualizar el historial médico de un paciente. Esto solo pueden realizarlo las autoridades reguladoras, el laboratorio y los usuarios que tengan autorización.

13. El médico responsable de un paciente debe poder dar autorización a usuarios para visualizar los datos clínicos del paciente. También debe asegurarse que esas autorizaciones puedan revocarse cuando el médico lo requiera.

4.2. Desarrollo

4.2.1. Diseño de la aplicación

En base a los requerimientos especificados, se desarrolló un sistema distribuido que permita la gestión de los datos clínicos de pacientes sometidos a ensayos clínicos utilizando la tecnología *blockchain*. La aplicación se diseñó teniendo en cuenta los siguientes actores dentro de un ensayo clínico:

- Laboratorio: es la entidad que promueve el ensayo clínico que se realiza, siendo el principal interesado en la obtención y el análisis de los datos clínicos.
- Reguladores: representan a las entidades reguladoras (privadas y públicas) que verifican las condiciones en las que se desarrolla el ensayo clínico, de acuerdo a las normas legales vigentes.
- CROs (del inglés *Contract Research Organization*): son organizaciones privadas que prestan sus servicios en la promoción del ensayo clínico y en la búsqueda de médicos y pacientes que participen del mismo.
- Médicos: entidades que participan del ensayo de diversas formas. Principalmente, están encargadas de registrar los datos clínicos de sus pacientes que participan en el ensayo.
- Pacientes: aquellas entidades que participan activamente del ensayo.

En la Figura 4.1 se puede observar el esquema que muestra la relación entre estos actores.

El laboratorio es la entidad que impulsa la realización del ensayo y es la responsable de su organización, desarrollo y financiación. En la lógica del contrato desarrollado, esta entidad es la encargada de la publicación de la información del ensayo, el registro de CROs contratadas y de las autoridades que regulan el ensayo. Asimismo, tienen el poder de la validación de todos los usuarios involucrados y la visualización de todos los datos clínicos que se registren en la red.

Por otro lado, las autoridades reguladoras son miembros pertenecientes o autorizados por el organismo encargado de la autorización y el control del ensayo clínico, el cual realiza inspecciones regularmente para verificar que los estudios se estén llevando a cabo de acuerdo a las reglas vigentes. En Argentina ese rol lo posee ANMAT [36].

Uno de los roles de las CROs, por su parte, es buscar pacientes que participen del ensayo clínico. Generalmente, esto lo logran con el contacto con médicos que posean pacientes que reúnan las características del ensayo. En la lógica del contrato, están

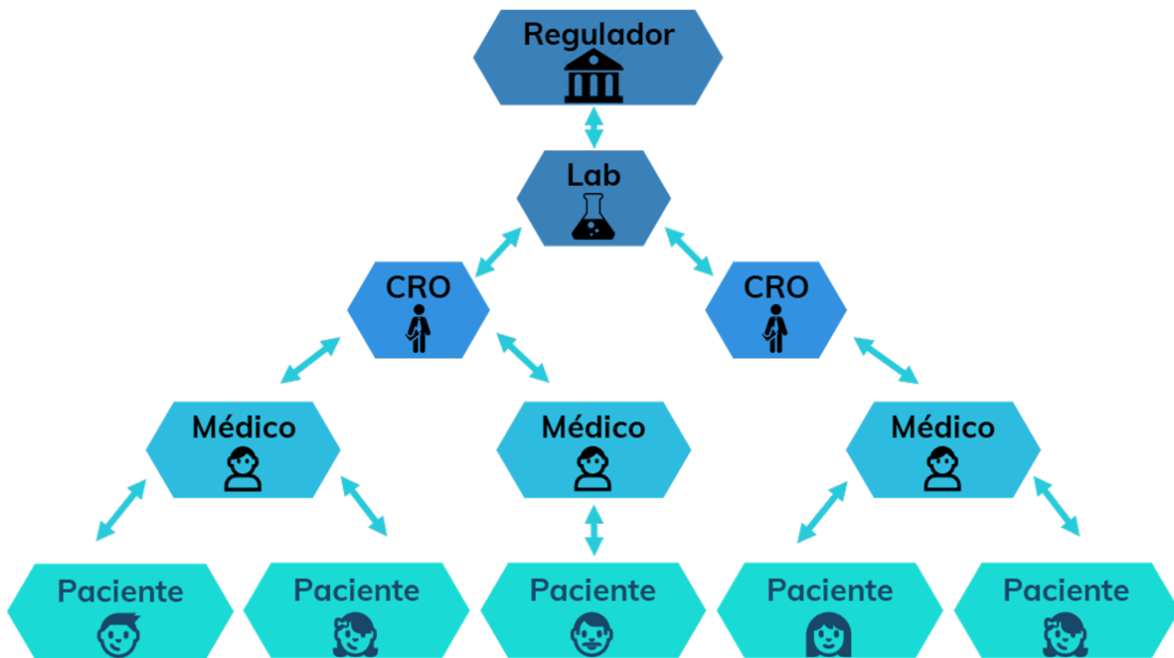

Figura 4.1: Esquema que indica la relación entre los distintos actores del ensayo clínico.

encargados de registrar a los médicos responsables de la atención de los pacientes contratados.

Los médicos poseen un abanico de funciones dentro de un ensayo. Algunos están encargados del análisis y el tratamiento de los datos que este arroja, mientras que otros se encargan de auditar los datos. También, como se menciona en el párrafo anterior, forman parte del ensayo los médicos responsables del seguimiento de sus pacientes. En la aplicación, se considera este último papel como el que tiene el rol de médico (al fin y al cabo, los demás papeles forman parte de los roles del laboratorio y las autoridades). En este caso, el médico registra a sus pacientes en el *blockchain*, crea sus fichas médicas y añade los datos clínicos de cada uno.

Por último, los pacientes son registrados por sus respectivos médicos. Si bien no tienen relevancia en el estado actual de la aplicación, son importantes para avances futuros de la misma, como se detalla en el capítulo 5.

La aplicación debe cumplir una serie de funcionalidades básicas, asociadas principalmente al registro y a la validación de los distintos usuarios, al registro de los datos clínicos de los pacientes y a la asignación de permisos para su visualización.

Los documentos asociados a los datos clínicos no se almacenarán en el *blockchain* debido a que es económicamente inviable el almacenamiento de archivos de gran tamaño en esta red. Esto se soluciona con la implementación de IPFS. En primer lugar, cada documento clínico se almacenará en la red IPFS, obteniéndose un hash que indica su paradero. Luego, este hash es almacenado en la red *blockchain*. De esta manera, pueden almacenarse los datos clínicos de una manera mucho más económica.

Por otra parte, la gestión de los datos clínicos se realiza nuevamente utilizando el estándar ERC721. En el capítulo 3 se utilizó este estándar en la tokenización de los títulos. En este caso, cada token representa los datos clínicos de un paciente.

En las secciones siguientes se describe el desarrollo del *backend* y el *frontend* de la aplicación.

4.2.2. Backend: creación del contrato inteligente

4.2.2.1. Codificación del contrato

Importación de archivos

Al igual que en el desarrollo de la aplicación de gestión de los títulos académicos en la sección 3.5, se importan los contratos Ownable.sol y ERC721.sol, ambos de la biblioteca OpenZeppelin.

La funcionalidad básica del token ERC721 se describió en la sección 2.1. Por otra parte, se describen las funciones básicas del contrato Ownable.sol.

- `address owner`

  Descripción: variable que almacena la dirección del dueño del contrato. Inicialmente, el dueño del contrato es el que lo desplegó.

- `function owner() public returns (address)`

  Descripción: retorna la dirección del dueño del contrato.

  Tipo de usuario con acceso: Público.

- `function transferOwnership(address newOwner) public onlyOwner`

  Descripción: establece un nuevo dueño del contrato.

  Tipo de usuario con acceso: Dueño del contrato.

Variables y estructuras de datos

A continuación se muestran las variables y estructuras utilizadas para describir los datos en la aplicación de ensayos clínicos:

- `uint doctorId, croId, regulatorId, patientId`

Descripción: variables tipo *integer* utilizadas para asignar un *id* a las distintas entidades que se registran. Sirven también para llevar un registro del número de usuarios registrados por cada rol.

- ```
  struct DataCT {
      uint startDate;
      uint endDate;
      string data;
  }
  DataCT document;
  ```

  Descripción: estructura de datos que almacena la información general del ensayo clínico.
  `startDate`: fecha de inicio del ensayo,
  `endDate`: fecha de finalización del ensayo,
  `string data`: hash de la documentación del ensayo.

- ```
  struct Identity {
      address addr;
      string name;
      string data;
  }
  ```

  Descripción: Estructura de datos que almacena la información de una entidad.
  `address addr`: dirección de la entidad,
  `string name`: nombre de la entidad,
  `string data`: hash de la documentación de la entidad.

- ```
  mapping (uint => Identity) _doctors
  mapping (uint => Identity) _cros
  mapping (uint => Identity) _regulators
  mapping (uint => Identity) _patients
  ```

  Descripción: relacionan un *id* de un usuario con su estructura correspondiente.

- ```
  mapping (address => bool) _isDoctor
  mapping (address => bool) _isCro
  mapping (address => bool) _isRegulator
  mapping (address => bool) _isPatient
  ```

  Descripción: relacionan una dirección con un determinado rol.

- `mapping (address => address) _creator`

  Descripción: establece la relación de superioridad entre direcciones en el ensayo clínico.

- `uint tokenId`

  Descripción: es utilizada para asignar un *id* a la ficha médica de un paciente. Análogamente, es el número de token de los datos clínicos del paciente.

- `mapping (uint => address) _tokenPatient`
  `mapping (uint => string) _tokenUri`

  Descripción: relacionan el `tokenId` de un paciente con su dirección y el hash de su documentación.

- ```
  struct DataPoint {
     uint timestamp;
     string clinicalData;
  }
  ```

  Descripción: estructura de datos que almacena un dato clínico de un paciente.
  `uint timestamp`: momento en que se registra el dato,
  `string clinicalData`: nombre de la entidad.

- `mapping (uint => DataPoint[]) _patientRecord`

  Descripción: relaciona el `tokenId` de un paciente con un array que contiene todos sus datos clínicos.

- `mapping (uint => mapping (address => bool)) _hasPermission`

  Descripción: secuencia de mapeo que establece un valor booleano en la relación entre un `tokenId` y una dirección. Si el valor `bool` establecido es verdadero, entonces la dirección tiene permiso para visualizar los datos clínicos correspondientes al `tokenId`. De lo contrario, no los tiene, por lo que no podrá visualizarlos.

Modificadores de funciones

Se creó el modificador `onlyLab` que comprueba si la cuenta que ejecuta la función es la del laboratorio. También se implementa el modificador `onlyEntity`, el cual es utilizado en la función `getDataPoint` y comprueba si la cuenta que llama a la función está autorizada.
  ```

Constructor

Se programó el constructor para establecer la cuenta de laboratorio, asignándola al usuario que despliega el contrato.

Funciones

Para cumplir con los requerimientos expuestos en 4.1, se implementaron en el contrato inteligente una serie de funciones. En la siguiente lista se enumeran y describen las funciones desarrolladas.

- `function setLab(address _lab) external onlyLab`

  Descripción: asigna la cuenta que juega el rol de laboratorio en el contrato inteligente. El laboratorio al inicio del contrato es el que lo desplegó.

  Tipo de usuario con acceso: Laboratorio.

- `function getLab() external view returns (address)`

  Descripción: retorna la dirección del laboratorio.

  Tipo de usuario con acceso: Público.

- `function addDataCT(uint _startDate, uint _endDate, string _data) external onlyLab`

  Descripción: registra la información del ensayo clínico, específicamente la fecha de inicio, la fecha de finalización y el hash de la documentación.

  Tipo de usuario con acceso: Laboratorio.

- `function getDataCT() external view returns (uint, uint, string)`

  Descripción: retorna la información almacenada del ensayo clínico.

  Tipo de usuario con acceso: Público.

- `function addEntity(string _entity, address _addr, string _name, string _data) external`

  Descripción: registra en el `blockchain` los datos de un usuario: dirección, nombre y hash de la documentación. `entity` indica el rol del usuario a registrar. Por ejemplo, si se quiere registrar una CRO, la función le asigna un número identificador utilizando la variable `croId` y almacena los datos en la estructura `_doctors`.

Tipo de usuario con acceso: Usuarios validados.

- `function updateEntity(string _entity, uint _id, address _addr, string _name, string _data) external`

  Descripción: modifica la entidad, identificada con `_entity` y `_id`. No pueden modificarse entidades que hayan sido validadas.

  Tipo de usuario con acceso: Usuario que registró la entidad a modificar.

- `function approveEntity(string _entity, uint _id) external onlyLab`

  Descripción: valida el usuario, otorgándole el rol especificado.

  Tipo de usuario con acceso: Laboratorio.

- `function numberOfEntity(string _entity) external view returns(uint)`

  Descripción: retorna la cantidad de usuarios existentes de acuerdo al rol especificado con `_entity`.

  Tipo de usuario con acceso: Público.

- `function getEntity(string _entity, uint _id) external view returns (address, address, string, string, bool)`

  Descripción: retorna los datos de un usuario especificado.

  Tipo de usuario con acceso: Público.

- `function addPatientRecord(string _data, address _patientAddress) external`

  Descripción: registra la ficha médica (que es el hash del documento) del paciente con la cuenta `_patientAddress`, almacenando los datos en `_tokenUri` y `_tokenPatient`. Le otorga a la ficha un número identificador utilizando la variable `tokenId` y crea el token ERC721 representativo.

  Tipo de usuario con acceso: Doctor responsable del paciente a añadir la ficha médica.

- `function changePatientRecord(uint _tokenId, string _data, address _patientAddress) external`

Descripción: modifica la ficha médica de un paciente.

Tipo de usuario con acceso: Doctor responsable del paciente a modificar la ficha médica.

```
function getPatientRecord(uint _tokenId) external
view returns (string, address)
```

Descripción: retorna la ficha médica de un paciente.

Tipo de usuario con acceso: Público.

- `function numberOfPatientRecord() external view returns (uint)`

  Descripción: retorna la cantidad de fichas médicas registradas en el *blockchain*.

  Tipo de usuario con acceso: Público.

- `function addDataPoint(uint _tokenId, string _clinicalData) external`

  Descripción: registra el dato clínico de un paciente. `_clinicalData` es el hash de la documentación del dato clínico, mientras que `_tokenId` es el número de token que refiere a la ficha médica del paciente.

  Tipo de usuario con acceso: Doctor responsable del paciente a añadir el dato clínico.

- ```
  function getDataPoint(uint _tokenId, uint _dataPointId)
  external view returns (string)
  ```

  Descripción: retorna un dato clínico especificado por `_tokenId` y `_dataPointId`.

  Tipo de usuario con acceso: Laboratorio, Reguladores y usuarios validados.

- ```
  function numberOfDataPoint(uint _tokenId) external view returns(uint
  )
  ```

  Descripción: retorna la cantidad de datos clínicos que posee el historial médico especificado por `_tokenId`.

  Tipo de usuario con acceso: Laboratorio, Reguladores y usuarios validados.

- `function setPermission(uint _tokenId, address _newAddress) external`

Descripción: establece autorización a la dirección `_newAddress` para visualizar los datos clínicos de un paciente.

Tipo de usuario con acceso: Doctor responsable del paciente.

- `function unsetPermission(uint _tokenId, address _address) external`

  Descripción: revoca a la dirección `_newAddress` la autorización de visualización de los datos clínicos de un paciente.

  Tipo de usuario con acceso: Doctor responsable del paciente.

En cuanto a la función `addEntity`, no cualquier usuario validado puede añadir cuentas de cualquier rol. De acuerdo a la Figura 4.1, las reglas establecidas son:

- El laboratorio puede añadir a entidades reguladoras.
- El laboratorio puede añadir a CROs.
- Las CROs que estén validadas pueden añadir a doctores.
- Los doctores pueden añadir a sus pacientes.

Eventos

Todos los eventos implementados están relacionados a la escritura de los datos en el *blockchain*. Los mismos se enumeran a continuación:

- `event NewLab(address previousLab, address newLab)`

  Descripción: se ha establecido la dirección de laboratorio.

  Parámetros: `address previousLab`: dirección anterior,
  `address newLab`: nueva dirección.

- `event ClinicalTrialDataAdded(uint startDate, uint endDate, string data, uint timeStamp)`

  Descripción: se ha añadido la información del ensayo clínico.

  Parámetros: `uint startDate`: fecha de inicio del ensayo clínico,
  `uint endDate`: fecha de finalización del ensayo clínico,
  `string data`: hash de la documentación del ensayo clínico,
  `uint timeStamp`: momento en que se registra el evento.

- `event EntityAdded(string entity, uint id, address addr, string name, string data, address creator, uint timeStamp)`

  Descripción: una entidad ha sido registrada.

  Parámetros: `string entity`: tipo de entidad registrada,
  `uint id`: *id* de la entidad,
  `address addr`: dirección de la entidad,
  `string name`: nombre de la entidad,
  `string data`: hash del documento de la identidad,
  `address creator`: dirección del creador de la identidad,
  `uint timeStamp`: momento en que se registra el evento.

- `event EntityUpdated(string entity, uint id, address addr, string name, string data, address creator, uint timeStamp)`

  Descripción: una entidad ha sido modificada.

  Parámetros: `string entity`: tipo de la entidad modificada,
  `uint id`: *id* de la entidad,
  `address addr`: dirección de la entidad,
  `string name`: nombre de la entidad,
  `string data`: hash del documento de la identidad,
  `address creator`: dirección del creador de la identidad,
  `uint timeStamp`: momento en que se registra el evento.

- `event EntityApproved(string entity, uint id, address addr, string name, string data, address creator, uint timeStamp)`

  Descripción: una entidad ha sido validada.

  Parámetros: `string entity`: tipo de la entidad validada,
  `uint id`: *id* de la entidad,
  `address addr`: dirección de la entidad,
  `string name`: nombre de la entidad,
  `string data`: hash del documento de la identidad,
  `address creator`: dirección del creador de la identidad,
  `uint timeStamp`: momento en que se registra el evento.

- `event DataPatientAdded(uint tokenId, address tokenOwner, uint timeStamp)`

Descripción: se ha añadido la ficha médica de un paciente.

Parámetros: `uint tokenId`: *id* del registro clínico del paciente,
`address tokenOwner`: dirección dueña del registro del paciente,
`uint timeStamp`: momento en que se registra el evento.

- `event DataPatientChanged(uint tokenId,address tokenOwner, uint timeStamp)`

  Descripción: se ha modificado la ficha médica de un paciente.

  Parámetros: `uint tokenId`: *id* del registro clínico del paciente,
  `address tokenOwner`: dirección dueña del registro del paciente,
  `uint timeStamp`: momento en que se registra el evento.

- `event DataPointAdded(uint tokenId, address tokenOwner, uint timeStamp)`

  Descripción: se ha añadido un nuevo dato clínico.

  Parámetros: `uint tokenId`: *id* del registro clínico del paciente,
  `address tokenOwner`: dirección dueña del registro del paciente,
  `uint timeStamp`: momento en que se registra el evento.

- `event SetPermission(uint tokenId, address tokenOwner, address to, uint timeStamp)`

  Descripción: se ha dado permiso a una cuenta para visualizar los datos clínicos de un paciente.

  Parámetros: `uint tokenId`: *id* del registro clínico del paciente,
  `uint tokenOwner`: dirección dueña del registro del paciente,
  `uint timeStamp`: momento en el que se registra el evento.

- `event UnsetPermission(uint tokenId, address tokenOwner, address to, uint timeStamp)`

  Descripción: se ha quitado permiso a una cuenta para visualizar los datos clínicos de un paciente.

  Parámetros: `uint tokenId`: *id* del registro del paciente,
  `uint tokenOwner`: dirección dueña del registro del paciente,
  `uint timeStamp`: momento en el que se registra el evento.

4.2.2.2. Despliegue del contrato

A continuación, se despliega el contrato a la red Ganache mediante el comando `truffle migrate`. En la tabla 4.1 se muestra información relacionada al coste y al gas utilizado en este proceso. Los costos son de carácter estimativo. El precio de gas es establecido en 20 gwei de acuerdo a lo mencionado en la sección 2.1.

Gas usado	6470000
Precio del gas	20 gwei
Costo total (en ETH)	0.129 ETH
Valor del ETH	USD 270
Costo total (en USD)	35

Tabla 4.1: Tabla de costos estimados de desplegar el contrato en el *blockchain*.

4.2.2.3. Pruebas unitarias del contrato

Las funciones programadas en la etapa de testeo buscan interactuar con todas las funciones y estructuras del contrato inteligente de forma meticulosa con el objeto de comprobar la ausencia de errores en cada línea de código. Específicamente, el testeo realizado cumple con los siguientes propósitos:

- Verifica que el dueño del contrato es quien que lo desplegó.
- Interactúa con la función `setLab` estableciendo una nueva dirección del laboratorio.
- Registra la información del ensayo clínico con `addDataCT`.
- Interactúa con `addEntity` registrando numerosas veces todos los tipos de entidades.
- Interactúa con `updateEntity` modificando entidades.
- Valida entidades con `approveEntity`.
- Registra fichas médicas de pacientes con `addPatientRecord`.
- Modifica fichas médicas de pacientes con `changePatientRecord`.
- Verifica los valores de retorno de las funciones: `getLab`, `getDataCT`, `getEntity`, `getPatientRecord`, `getDataPoint`.
- Otorga permisos a distintas direcciones para la visualización de algunos registros médicos de pacientes mediante la función `setPermission`.

- Interactúa con `unsetPermission` revocando permisos anteriormente otorgados.
- Observa todos los eventos emitidos por el contrato, comprobando la correcta información de los parámetros.
- Para cada función, verifica que sólo las direcciones permitidas pueden acceder. Esto es:
  - Solo el laboratorio puede acceder a `setLab` y `addDataCT`.
  - Solo entidades que estén validadas pueden registrar otras entidades con `addEntity`.
  - Al querer modificar una entidad con `updateEntity`, únicamente la entidad que la registró inicialmente puede hacerlo.
  - El laboratorio es el único que puede validar entidades con `approveEntity`.
  - Solo pueden añadir fichas médicas de pacientes los doctores que estén validados. Asimismo, sólo el doctor responsable del paciente puede añadir y modificar su ficha médica con `addPatientRecord` y `changePatientRecord`.
  - Solo el doctor responsable del paciente puede añadir sus datos clínicos con `addDataPoint`, establecer permisos de visualización de estos datos a determinadas direcciones con `setPermission` y revocarlos con `unsetPermission`.

```
Compiling your contracts...
===========================
> Everything is up to date, there is nothing to compile.


  Contract: Clinical Trial test
    ✓ Deploys successfully
    ✓ Owner is accounts[0] (59ms)
    ✓ Owner is lab
    ✓ Lab can set lab (79ms)
    ✓ Data CT is added (58ms)
    ✓ Cro is added (86ms)
    ✓ Cro is updated (85ms)
    ✓ Cro is approved (68ms)
    ✓ Regulator is added (165ms)
    ✓ Regulator is updated (161ms)
    ✓ Regulator is approved (83ms)
    ✓ Doctor is added (79ms)
    ✓ Doctor is updated (62ms)
    ✓ Doctor is approved (64ms)
    ✓ Patient is added (81ms)
    ✓ Patient is updated (114ms)
    ✓ Patient is approved (69ms)
    ✓ numberOfEntity is 1 for each Entity (109ms)
    ✓ PatientRecord is added (73ms)
    ✓ PatientRecord is changed (48ms)
    ✓ PatientRecord from numberOfPatientRecord
    ✓ DataPoint 0 is added (81ms)
    ✓ DataPoint 1 is added (44ms)
    ✓ numberOfDataPoint is 2 for tokenId = 1
    ✓ User has permission
    ✓ User has not permission (44ms)


  26 passing (2s)
```

Figura 4.2: Resultados del *test* realizado. Se han probado la totalidad de las funciones programadas.

En la figura 4.2 se muestran los resultados del test, observando que todas las pruebas realizadas han resultado exitosas.

4.2.3. Frontend: creación de la página web

4.2.3.1. Codificación de la página web

Las funciones desarrolladas en el código JavaScript son análogas a las del contrato. Además, se realizan funciones relacionadas a obtener información de la cuenta y de la red, armar tablas que muestren la información de los usuarios registrados, y almacenar la documentación ingresada por el usuario en el sistema IPFS. Las mismas son:

- `loadWeb3`

  Descripción: configura Web3 con Metamask y el nodo local Ganache.

- `loadAccount`

  Descripción: Obtiene la dirección del usuario. Para eso, utiliza `web3.eth.accounts[0]`.

- `loadContract`

  Descripción: Crea una versión de JavaScript del contrato utilizando la biblioteca `Truffe Contract`.

- `pastEvents`

  Descripción: Muestra todos los eventos emitidos al interactuar con el contrato inteligente.

- `blockInfo`

  Descripción: Recopila y muestra la información de los bloques de Ganache.

- `upload`

  Descripción: Recibe un documento del usuario, lo almacena en IPFS y emite el hash correspondiente. Este hash luego es almacenado en el *blockchain*.

A su vez, se escriben los códigos en HTML y CSS que proporcionan la estructura visual de la página.

4.2.3.2. Configuración de las herramientas

Por un lado, es necesario inicializar un nodo de IPFS, por lo que en una terminal se ejecuta el comando `ipfs daemon`. Esto se observa en la Figura 4.3

Luego, se lanza el servidor Webpack con el comando `npm run dev`. En la Figura 4.4 se observa la correcta ejecución del servidor.

```
(base) agustin@agustin-N552VW:~$ sudo ipfs daemon
[sudo] password for agustin:
Initializing daemon...
go-ipfs version: 0.4.19-
Repo version: 7
System version: amd64/linux
Golang version: go1.11.5
Swarm listening on /ip4/10.73.98.159/tcp/4001
Swarm listening on /ip4/127.0.0.1/tcp/4001
Swarm listening on /ip6/::1/tcp/4001
Swarm listening on /p2p-circuit
Swarm announcing /ip4/10.73.98.159/tcp/4001
Swarm announcing /ip4/127.0.0.1/tcp/4001
Swarm announcing /ip6/::1/tcp/4001
API server listening on /ip4/127.0.0.1/tcp/5001
WebUI: http://127.0.0.1:5001/webui
Gateway (readonly) server listening on /ip4/127.0.0.1/tcp/8080
Daemon is ready
```

Figura 4.3: Resultado al ejecutar el comando `ipfs daemon`, el cual inicializa un nodo IPFS.

```
(base) agustin@agustin-N552VW:~/blockchain/ibtrials/clinical-trials$ npm run dev

> clinical-trials@1.0.0 dev /home/agustin/blockchain/ibtrials/clinical-trials
> webpack-dev-server

i ｢wds｣: Project is running at http://localhost:8081/
i ｢wds｣: webpack output is served from /
i ｢wds｣: Content not from webpack is served from /home/agustin/blockchain/ibtria
ls/clinical-trials/dist
i ｢wdm｣: Hash: 16b8d68a2e56593ac4eb
Version: webpack 4.31.0
Time: 2423ms
Built at: 06/20/2019 2:57:42 AM
     Asset       Size  Chunks             Chunk Names
index.html   59.9 KiB          [emitted]
  index.js   4.96 MiB    main  [emitted]  main
Entrypoint main = index.js
[0] multi (webpack)-dev-server/client?http://localhost ./src/index.js 40 bytes {
main} [built]
[./build/contracts/CT2.json] 674 KiB {main} [built]
[./node_modules/ansi-html/index.js] 4.16 KiB {main} [built]
[./node_modules/loglevel/lib/loglevel.js] 7.68 KiB {main} [built]
[./node_modules/querystring-es3/index.js] 127 bytes {main} [built]
[./node_modules/truffle-contract/index.js] 555 bytes {main} [built]
[./node_modules/url/url.js] 22.8 KiB {main} [built]
[./node_modules/web3/src/index.js] 2.01 KiB {main} [built]
[./node_modules/webpack-dev-server/client/index.js?http://localhost] (webpack)-d
ev-server/client?http://localhost 8.26 KiB {main} [built]
[./node_modules/webpack-dev-server/client/overlay.js] (webpack)-dev-server/clien
t/overlay.js 3.59 KiB {main} [built]
[./node_modules/webpack-dev-server/client/socket.js] (webpack)-dev-server/client
/socket.js 1.05 KiB {main} [built]
[./node_modules/webpack-dev-server/node_modules/strip-ansi/index.js] (webpack)-d
ev-server/node_modules/strip-ansi/index.js 161 bytes {main} [built]
[./node_modules/webpack/hot sync ^\.\/log$] (webpack)/hot sync nonrecursive ^\.\
/log$ 170 bytes {main} [built]
[./node_modules/webpack/hot/emitter.js] (webpack)/hot/emitter.js 75 bytes {main}
 [built]
[./src/index.js] 26.8 KiB {main} [built]
    + 547 hidden modules
i ｢wdm｣: Compiled successfully.
```

Figura 4.4: Resultado al ejecutar el comando `npm run dev`, observándose que Webpack se ha ejecutado correctamente y corre en `http://localhost:8081`.

En la Figura 4.5 se puede observar la página web que lanza el servidor, junto con el complemento Metamask desplegado.

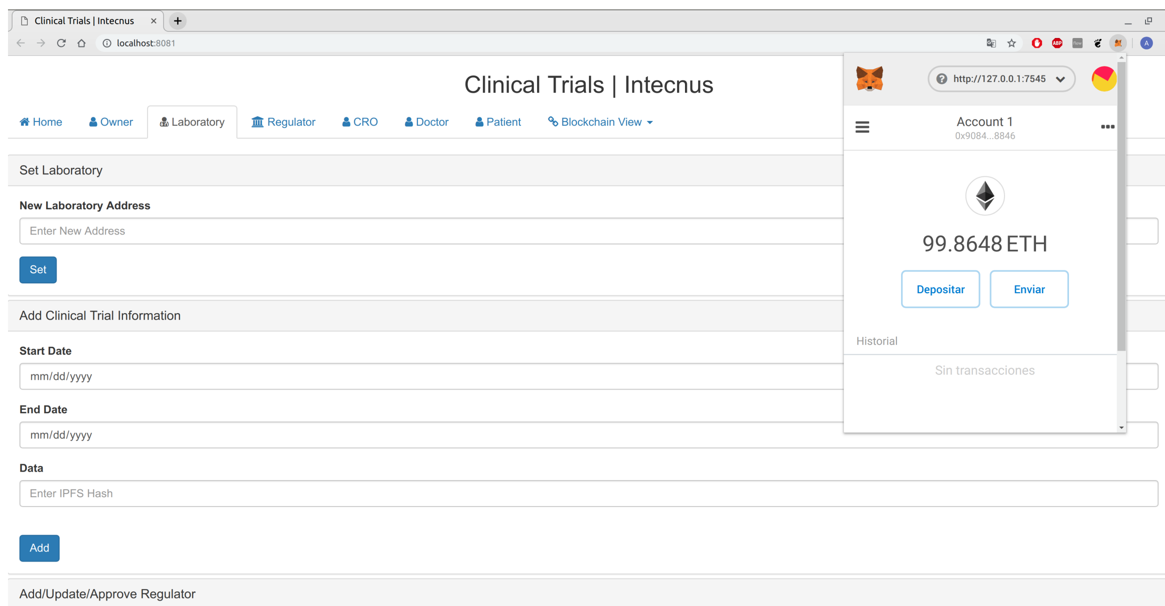

Figura 4.5: Página web de la aplicación y complemento Metamask desplegado.

En la página se pueden observar distintas secciones:

- *Home*: es la sección orientada para el usuario en general. En esta se puede obtener la información del ensayo clínico, de todas las entidades que forman parte del mismo y de las fichas médicas registradas.
- *Owner*: aquí se encuentra la función de establecer un nuevo dueño del contrato.
- *Laboratory*: esta sección posee todas las funciones que puede realizar exclusivamente el laboratorio, como añadir la información del ensayo clínico, añadir los reguladores del ensayo y validar todas las entidades.
- *CRO*: esta sección está orientada a las CROs del ensayo clínico. Posee las funciones de añadir y modificar médicos.
- *Doctor*: aquí los doctores pueden acceder a sus funciones exclusivas como las de añadir y modificar pacientes, añadir fichas médicas y registrar datos clínicos.
- *Audit Trail*: en esta sección se muestran todos los eventos que emitió el contrato desde que fue desplegado. Es útil para observar la cronología de las transacciones.
- *Block Explorer*: esta sección muestra información técnica relacionada al contenido de los bloques generados por la red.

4.2.4. Pruebas

Se procede a interactuar con algunas funciones de la página. En primer lugar, se establece como entidad laboratorio a la dirección `0xa083a7...` correspondiente a una

de las cuentas proporcionada por Ganache. Esto se hace desde la cuenta que desplegó el contrato, la cual es `0x908402...`. El agregar una entidad en el contrato inteligente implica salvar un nuevo dato dentro del estado, lo cual requiere una cierta cantidad de gas. En la Figura 4.6 se puede observar un aviso de Metamask informando el costo y el pedido de confirmación de la transacción.

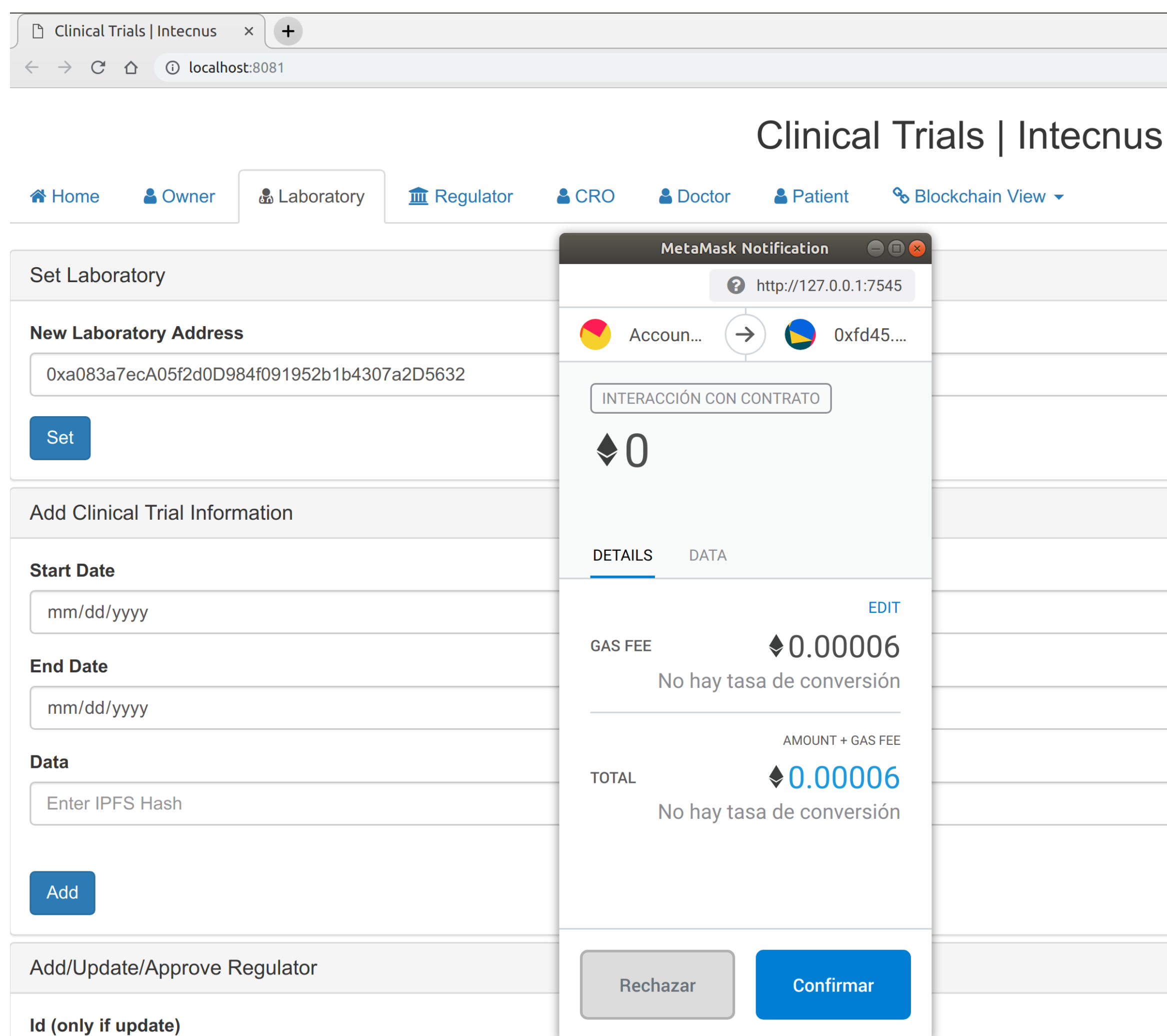

Figura 4.6: Estableciendo una entidad, en este caso, un laboratorio. Al agregar la dirección del nuevo laboratorio y apretar el botón Set, se despliega Metamask solicitando la autorización para realizar la transacción con el contrato.

Una vez confirmada la transacción, en la interfaz de Ganache puede verificarse los detalles de la misma. En la Figura 4.7 se observa que el emisor de la transacción es la cuenta que desplegó el contrato y que en los datos de la transacción aparece la cuenta establecida como laboratorio. El receptor de esta transacción es el contrato inteligente, cuya dirección se puede ver en la figura.

También en la sección *Blockchain View* en la Figura 4.8 pueden observarse los eventos emitidos por el contrato. El último evento confirma lo expuesto en la Figura 4.7.

ACCOUNTS BLOCKS TRANSACTIONS LOGS

CURRENT BLOCK	GAS PRICE	GAS LIMIT	NETWORK ID	RPC SERVER	MINING STATUS
5	2000000000	6721975	5777	HTTP://127.0.0.1:7545	AUTOMINING

← BACK TX 0x e4a73ccfeedb1d387770f8b1cf070ae7d8c3e8c4593519d1aae2ce8687b9a82e

SENDER ADDRESS 0x90840296a7Ff97B3cf70f7Ec55D11719A2f88846
TO CONTRACT ADDRESS 0xfd45Ed00D98A8e287d08bFaE733AA7fd3d68c075
CONTRACT CALL

VALUE	GAS USED	GAS PRICE	GAS LIMIT	MINED IN BLOCK
0.00 ETH	30054	2000000000	30054	5

TX DATA
0x2b43f213000000000000000000000000a083a7eca05f2d0d984f091952b1b4307a2d5632

Figura 4.7: Registro de transacciones de Ganache, donde se ha seleccionado aquella que corresponde a dar de alta una nueva entidad (en este caso, un laboratorio) en el contrato. La transacción no conlleva un envío de fondos, pero sí incurre en un gasto de gas.

Clinical Trials | Intecnus

Home Owner Laboratory Regulator CRO Doctor Patient Blockchain View

Ownership Events

Event	From	To
NewLab	0x90840296a7Ff97B3cf70f7Ec55D11719A2f88846	0xa083a7ecA05f2d0D984f091952b1b4307a2D5632
NewLab	0x0000000000000000000000000000000000000000	0x90840296a7Ff97B3cf70f7Ec55D11719A2f88846
OwnershipTransferred	0x0000000000000000000000000000000000000000	0x90840296a7Ff97B3cf70f7Ec55D11719A2f88846

Figura 4.8: Registro propio de eventos de la aplicación, donde puede observarse la autorización a agregar un laboratorio.

En forma análoga, se puede agregar una CRO. Sin embargo, la entidad autorizante en este caso es el laboratorio. Por lo tanto, es necesario seleccionar dicha dirección en el menú de cuentas de Metamask para que la operación pueda concretarse. Una vez seleccionada la cuenta del laboratorio, se debe añadir el documento al sistema IPFS y obtener el hash. Esta función está disponible en la pestaña *Home*, y se observa en la Figura 4.9.

Save Document on IPFS

Choose File Documento.pdf

Upload

Succesful.
The hash is: QmX5qNAwXDmr7Kbw4KxPAt1yqM9FM5Mn7VdKnjHxMkWEUb
The url is: https://ipfs.io/ipfs/QmX5qNAwXDmr7Kbw4KxPAt1yqM9FM5Mn7VdKnjHxMkWEUb

Figura 4.9: Proceso de almacenamiento del archivo `Documento.pdf` en el sistema IPFS y obtención del hash.

Una vez obtenido el hash, se procede a interactuar con la función `addEntity` del contrato para registrar la nueva CRO, como se observa en la Figura 4.10.

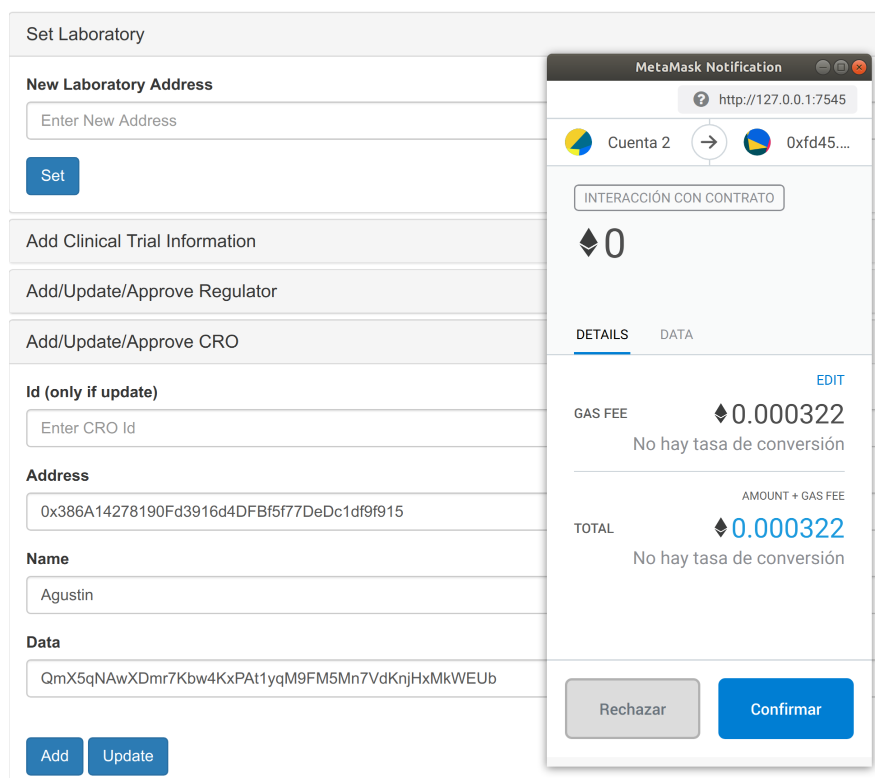

Figura 4.10: Captura de pantalla de la autorización para dar de alta a una CRO por parte del laboratorio. Nótese que la cuenta que emite la transacción en Metamask es diferente a la utilizada para dar de alta al laboratorio.

Nuevamente, como muestra la Figura 4.11, se observa el evento emitido por el contrato al registrar los datos de la CRO, visualizándose su nombre, tipo de entidad, *id*, dirección, link de acceso al archivo almacenado en IPFS y el registro de tiempo.

Entity Events

Event	Entity	Id	Address	Name	Data	Timestamp
EntityAdded	Cro	1	0x386A14278190Fd3916d4DFBf5f77DeDc1df9f915	Agustin	QmX5qNAwXDmr7Kbw4KxPAt1yqM9FM5Mn7VdKnjHxMkWEUb	Thu Jun 20 2019 04:20:22 GMT-0300 (Argentina Standard Time)

Figura 4.11: Registro de transacciones de Ganache, donde se puede observar la interacción con el contrato.

También se puede observar en la Figura 4.12 que en Ganache se generó la transacción. En la sección *TX DATA*, los valores entre ceros corresponden al código hexadecimal de los datos establecidos.

Se puede encontrar una demostración más completa de la aplicación en el repositorio del proyecto en gitlab [37].

ACCOUNTS BLOCKS TRANSACTIONS LOGS SEARCH FOR BLOCK NUMBERS OR TX HASHES

CURRENT BLOCK 6 | GAS PRICE 2000000000 | GAS LIMIT 6721975 | NETWORK ID 5777 | RPC SERVER HTTP://127.0.0.1:7545 | MINING STATUS AUTOMINING

← BACK **TX 0×cd98b289ba4d82cfbb761cf01723a544e9c64b6ac7b0987ebd609d1a62a25017**

SENDER ADDRESS
0×a083a7ecA05f2d0D984f091952b1b4307a2D5632

TO CONTRACT ADDRESS
0×fd45Ed00D98A8e287d08bFaE733AA7fd3d68c075

CONTRACT CALL

VALUE
0.00 ETH

GAS USED
160831

GAS PRICE
2000000000

GAS LIMIT
160831

MINED IN BLOCK
6

TX DATA
0×6f3ffaad0000000000000000000000000000000000000000000000000000000000000000800000000000000000000000000386a14278190fd3916d4dfbf5f77dedc1df9f9150
00000000000000000000000000000000000000000000000000000000000000c000000000000000000000000000000000000000000000000000000000000100000000000000
000000000000000000000000000000000000000000000000000000000343726f00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000074167757374696e000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000002e516d5835714e417758446d72374b6277344b785041743179714d39464d354d6e3756644b6e6a48784d6b574555620000000000000000
0000000000000000000

Figura 4.12: Registro de transacciones de Ganache, donde se ha seleccionado la transacción correspondiente al registro de la CRO.

Capítulo 5

Conclusiones y perspectivas

En el presente proyecto se desarrolló una aplicación para gestionar la emisión, validación y control de los datos producidos en un ensayo clínico utilizando como base una red *blockchain*, específicamente la plataforma Ethereum, por medio de contratos inteligentes.

En una primera instancia del proyecto, con el objeto de adquirir experiencia y conocimiento en el manejo de las herramientas de desarrollo y los lenguajes de programación Solidity y Javascript, se realizó una aplicación para gestionar la emisión, validación y visualización de los certificados académicos digitales del Instituto Balseiro. Si bien la implementación de esta aplicación se concibió como una introducción al ámbito de contratos inteligentes, puede pensarse como un instrumento complementario al sistema de ensayos clínicos: en efecto, puede ser utilizada para validar los títulos y las especialidades de los médicos usuarios del mismo.

A partir de los conocimientos adquiridos en esta primera etapa, se continuó con el desarrollo del *backend* y el *frontend* de la aplicación para ensayos clínicos. El primero consistió en la programación del contrato inteligente, estableciendo la funcionalidad a implementar en el *blockchain*. Debido a que no es económicamente rentable almacenar documentos en esta red, se utilizó la plataforma IPFS para almacenarlos. De esta manera, en el *blockchain* se realizó la gestión de los hashes de estos documentos, que involucra la generación adecuada de los mismos y su almacenamiento en la red. Además, se estableció la estructura lógica dentro de los contratos inteligentes destinada al otorgamiento de permisos y autorizaciones de visualización a entidades que lo requieran. Así, se pudo comprobar la compatibilidad y la sinergia entre la red Ethereum y el sistema IPFS.

Por otra parte, el *frontend* consistió en el desarrollo de la interfaz de usuario, específicamente, la creación de una página web con toda la funcionalidad necesaria para conectar con el sistema de contratos.

Se pudo verificar con ambas aplicaciones la viabilidad que la tecnología *blockchain*

presenta para ser utilizada como base para desarrollar aplicaciones, otorgándole a estas la posibilidad de garantizar la transparencia, inmutabilidad, control, disponibilidad y trazabilidad de los datos.

Blockchain es una tecnología que ha sido bien tomada por el público y ha crecido enormemente en los últimos años, logrando adentrarse en diversas áreas. Sin embargo, al ser una tecnología tan vanguardista, en algunas ocasiones la falta de información y la constante actualización de las herramientas resultan en un proceso de programación laborioso.

De esta manera, se sentaron las bases en el desarrollo de ambas aplicaciones. Las mismas pueden seguir creciendo desde puntos de vista de eficiencia, estética y, principalmente, funcional. Por esta razón, se propuso la implementación de una serie de funcionalidades para aumentar la utilidad de la aplicación.

La aplicación desarrollada cumple una funcionalidad básica en el manejo de los datos de un ensayo clínico y la gestión de las entidades involucradas. Permite almacenar los datos en el sistema de archivos IPFS y realizar la gestión de los hashes en el *blockchain*. Esta gestión consiste en el almacenamiento de los hashes en la red y en el otorgamiento de permisos de visualización a entidades que lo requieran.

Como recomendaciones a implementar, la aplicación puede extenderse de varias formas. En primer lugar, se podría crear una función para que un usuario pueda visualizar sus datos clínicos, lo cual requiere tanto modificaciones en el contrato inteligente como en la página web. También podría permitirse que ciertos usuarios puedan modificar o agregar datos clínicos de otro paciente, siempre y cuando esto quede registrado. Por ejemplo, los médicos que analizan los datos podrían registrar los análisis de los mismos y referenciarlos.

En segundo lugar, una característica atractiva que podría implementarse consiste en asociar una economía al contrato inteligente. En efecto, a través de la emisión de tokens se puede incentivar a los distintos actores del ensayo clínico. Por ejemplo, se pueden incorporar al contrato ciertas reglas lógicas como la adición o validación de datos clínicos, el cumplimiento de ciertos objetivos o la misma finalización del ensayo. Cuando un usuario cumpla con dichas condiciones, el contrato puede realizar automáticamente el depósito económico a las respectivas entidades.

En tercer lugar, una limitación que presenta la implementación actual es que no se contempla la actualización de los contratos. Por ende, es recomendable diseñar un proceso que permita que el contrato inteligente sea útil para numerosos ensayos clínicos. Actualmente, si el contrato se desea utilizar para un nuevo ensayo, es necesario desplegar y utilizar un contrato nuevo, quedando inutilizable toda la información almacenada del ensayo anterior. A modo de solución, una opción es que exista un contrato madre que registre toda la información de los usuarios que participen de los ensayos clínicos. Cuando se propone un nuevo ensayo, el contrato madre crea un contrato en el

que se establecen las relaciones entre los usuarios registrados, y no es necesario volver a registrar y validar las entidades participantes del estudio anterior. Estos contratos se denominan *proxies* o contratos actualizables. La idea de fondo es separar el contrato inteligente de acuerdo a su funcionalidad, de tal modo de tener un contrato de almacenamiento, un contrato *proxy* y un contrato de lógica. De esta manera, cuando sea necesario cambiar o modificar el contrato lógico, simplemente se crea uno nuevo y el contrato *proxy* lo delega como nuevo contrato lógico, heredando el contrato de almacenamiento. Así, los datos almacenados pueden volver a ser utilizados [38].

Finalmente, una opción viable a desarrollar es la implementación del contrato inteligente en el Blockchain Federal Argentina (BFA), en particular, la referida a la emisión de títulos académicos. BFA surge como una iniciativa para desarrollar una plataforma multiservicio de alcance federal basada en la tecnología *blockchain*. Es una plataforma nacional de uso público que permite ejecutar aplicaciones y sistemas que mejoren los procesos de organizaciones del sector tanto público como privado de todo el país. La plataforma está pensada para funcionar sin una criptomoneda asociada, y funciona como un espacio abierto para soportar desarrollos y aportes de todos los sectores de la sociedad [39].

Apéndice A

Instalación de las herramientas y configuración del servicio

Requerimientos

Se utilizaron las siguientes herramientas:

- Sistema operativo Ubuntu 18.04 LTS de 64 bits.
- Node v10.6.0.
- npm 6.9.0.
- Truffle v5.0.13.
- Solidity 0.5.2.
- Web3.js v1.0.0-beta.37.
- Open-Zeppelin 2.2.0.
- Webpack 4.31.0.
- Ganache 1.2.2.
- IPFS 0.4.19.
- Chrome 73.0.
- Metamask 6.6.2.

Instalación

- Instalación de NodeJS: acceder a la página oficial [40] y descargar el instalador. El paquete NPM viene incluido en esta instalación.
- Creación y acceso al directorio: `mkdir proyecto && cd proyecto`.
- Creación del archivo `package.json`: `npm init`. En el mismo se nombran todos los paquetes a instalar, como Webpack [41] y las bibliotecas Web3 [42] y OpenZeppelin [33].
- Instalación de los paquetes: `npm install`.
- Instalación de Truffle: `npm install -g truffle`. Visitar la documentación [43] para obtener más detalles.
- Creación de un proyecto de Truffle: `truffle init`. Se crean los directorios `contracts`, `migrations`, `test` y el archivo `truffle-config.js`. En este último se configura el servicio para la comunicación mediante el puerto 7545.
- Instalación de Ganache: descargar el instalador de su página oficial [28] y seguir las instrucciones. Una vez dentro de su interfaz, configurar el servicio para la comunicación mediante el puerto 7545.
- Instalación de IPFS: acceder a la página oficial [44] y descargar el instalador.
- Instalación de Google Chrome: acceder a la página oficial [45] y descargar el instalador.
- Instalación de Metamask: acceder a la página oficial [45] y descargar la extensión. Una vez dentro de su interfaz, configurar el servicio con la red `http://127.0.0.1:7545`.

Uso

- Inicio del nodo Ganache: `npm start`. Asegurarse de que este nodo esté en funcionamiento antes del despliegue del contrato, en el puerto 7545.
- Inicio del nodo IPFS: `ipfs daemon`.
- Compilación del contrato inteligente: `truffle compile`.
- Despliegue del contrato inteligente en la red Ganache: `truffle migrate`.
- Testeo del contrato: `truffle test`.

- Inicio de Metamask: abrir el navegador Chrome y asegurarse de que Metamask esté conectado con la red Ganache.

- Ejecución de Webpack para el funcionamiento de la página web: `npm run dev`. El mismo inicia un servidor http en el puerto 8081. Seguir las instrucciones en consola para conocer el estado.

Apéndice B

Contrato para gestión de ensayos clínicos

En este anexo se presentan los contratos inteligentes realizados para la gestión de los datos de un ensayo clínico. El código completo se encuentra disponible en el repositorio del proyecto en gitlab [37].

B.1. CT1.sol

```
pragma solidity ^0.5.0;

import "openzeppelin-solidity/contracts/ownership/Ownable.sol";

//el contrato CT1 hereda Ownable
contract CT1 is Ownable {

//eventos
  event NewLab (address _previousLab, address _newLab);
  event ClinicalTrialDataAdded (uint256 _startDate, uint256 _endDate, string _data,
      uint256 _timeStamp);
  event EntityAdded (string _entity, uint256 _id, address _addr, string _name, string
      _data, address _creator, uint256 _timeStamp);
  event EntityUpdated (string _entity, uint256 _id, address _addr, string _name,
      string _data, address _creator, uint256 _timeStamp);
  event EntityApproved (string _entity, uint256 _id, address _addr, string _name,
      string _data, address _creator, uint256 _timeStamp);

//variables
  address internal lab;

  uint256 internal doctorId;
  uint256 internal croId;
  uint256 internal regulatorId;
  uint256 internal patientId;

  DataCT internal document;
```

```
//estructuras
  struct Identity {
    address addr;
    string name;
    string data;
  }

  struct DataCT {
    uint256 startDate;
    uint256 endDate;
    string data;
  }

//mappings
  mapping (uint256 => Identity) _doctors;
  mapping (uint256 => Identity) _cros;
  mapping (uint256 => Identity) _regulators;
  mapping (uint256 => Identity) _patients;

  mapping (address => bool) _isDoctor;
  mapping (address => bool) _isCro;
  mapping (address => bool) _isRegulator;
  mapping (address => bool) _isPatient;

  mapping (address => address) _creator;

//modificadores
  modifier onlyLab {
    require(lab == msg.sender);
    _;
  }

//constructor
  constructor () internal {
      lab = msg.sender;
      emit NewLab(address(0), lab);
  }

//funcion interna que evalua si el string ingresado es una entidad
  function _onlyEntity (string memory _entity) internal pure returns (bool) {
    return(keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('Doctor'
        )) ||
    keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('Cro')) ||
    keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('Regulator'))
        ||
    keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('Patient')));
  }

//funcion interna que asegura que no se ingresen dos identidades del mismo tipo y con
    la misma direcci n asociada
  function _notRepeatAddress (string memory _entity, address _addr, uint256 _id)
      internal view returns (bool) {

    if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('Doctor')))
        {

      if(doctorId != 0) {
        for (uint256 i = 1; i <= doctorId; i++) {
          if (_addr == _doctors[i].addr && i != _id) {
```

```
          return false;
        }
      }
    }

    return true;

  } else if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('Cro
      '))) {

    if(croId != 0) {
      for (uint256 i = 1; i <= croId; i++) {
        if (_addr == _cros[i].addr && i != _id) {
          return false;
        }
      }
    }

    return true;

  } else if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('
      Regulator'))) {

    if(regulatorId != 0) {
      for (uint256 i = 1; i <= regulatorId; i++) {
        if (_addr == _regulators[i].addr && i != _id) {
          return false;
        }
      }
    }

    return true;

  } else if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('
      Patient'))) {

    if(patientId != 0) {
      for (uint256 i = 1; i <= patientId; i++) {
        if (_addr == _patients[i].addr && i != _id) {
          return false;
        }
      }
    }

    return true;

  }
 }

//establece el lab
 function setLab (address _lab) external onlyLab {

   lab = _lab;
   emit NewLab(msg.sender, _lab);

 }

//retorna el lab
 function getLab () external view returns (address) {
```

```
    return lab;

  }

//registra la informacion del ensayo cl nico
  function addDataCT (uint256 _startDate, uint256 _endDate, string calldata _data)
      external onlyLab {

    document.startDate = _startDate;
    document.endDate = _endDate;
    document.data = _data;

    emit ClinicalTrialDataAdded(_startDate, _endDate, _data, block.timestamp);
  }

//retorna la informacion del ensayo clinico
  function getDataCT () external view returns (uint256, uint256, string memory) {

    return (document.startDate, document.endDate, document.data);

  }

//registra una entidad
  function addEntity (string calldata _entity, address _addr, string calldata _name,
      string calldata _data) external {

    require(_onlyEntity(_entity),'Invalid Entity. Must be Doctor, Cro, Regulator or
        Patient');

    if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('Doctor')))
        {

      require(_isCro[msg.sender], 'Invalid caller. Only Cro can add a Doctor');
      _notRepeatAddress('Doctor', _addr, 0);

      doctorId += 1;
      Identity memory doctor;
      doctor.addr = _addr;
      doctor.name = _name;
      doctor.data = _data;

      _doctors[doctorId] = doctor;

      _creator[_addr] = msg.sender;
      emit EntityAdded ('Doctor', doctorId, _addr, _name, _data, msg.sender, block.
          timestamp);

    } else if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('Cro
        '))) {

      require(msg.sender == lab, 'Invalid caller. Only Lab can add a Cro');
      _notRepeatAddress('Cro', _addr, 0);

      croId += 1;
      Identity memory cro;
      cro.addr = _addr;
      cro.name = _name;
      cro.data = _data;
```

```
      _cros[croId] = cro;

      emit EntityAdded ('Cro', croId, _addr, _name, _data, msg.sender, block.timestamp
          );

    } else if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('
        Regulator'))) {

      require(msg.sender == lab, 'Invalid caller. Only Lab can add a Regulator');
      _notRepeatAddress('Regulator', _addr, 0);

      regulatorId += 1;
      Identity memory regulator;
      regulator.addr = _addr;
      regulator.name = _name;
      regulator.data = _data;

      _regulators[regulatorId] = regulator;

      emit EntityAdded ('Regulator', regulatorId, _addr, _name, _data, msg.sender,
          block.timestamp);

    } else if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('
        Patient'))) {

      require(_isDoctor[msg.sender], 'Invalid caller. Only Doctor can add a Patient');
      _notRepeatAddress('Patient', _addr, 0);

      patientId += 1;
      Identity memory patient;
      patient.addr = _addr;
      patient.name = _name;
      patient.data = _data;

      _patients[patientId] = patient;

      _creator[_addr] = msg.sender;
      emit EntityAdded ('Patient', patientId, _addr, _name, _data, msg.sender, block.
          timestamp);

    }

  }

//modifica una entidad
  function updateEntity (string calldata _entity, uint256 _id, address _addr, string
      calldata _name, string calldata _data) external {

    require(_onlyEntity(_entity),'Invalid Entity. Must be Doctor, Cro, Regulator or
        Patient');

    if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('Doctor')))
        {

      require(msg.sender == _creator[_addr], 'Invalid caller. Only the Super can
          update a Doctor');
      require (_id <= doctorId, 'Invalid Id');
      _notRepeatAddress('Doctor', _addr, _id);
```

```
    require (_isDoctor[_doctors[_id].addr] == false, 'Doctor is already accepted');

    _doctors[_id].addr = _addr;
    _doctors[_id].name = _name;
    _doctors[_id].data = _data;

    emit EntityUpdated ('Doctor', _id, _addr, _name, _data, msg.sender, block.
        timestamp);

  } else if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('Cro
      '))) {

    require(msg.sender == lab, 'Invalid caller. Only the Lab can update a Cro');
    require (_id <= croId, 'Invalid Id');
    _notRepeatAddress('Cro', _addr, _id);
    require (_isCro[_cros[_id].addr] == false, 'Cro is already accepted');

    _cros[_id].addr = _addr;
    _cros[_id].name = _name;
    _cros[_id].data = _data;

    emit EntityUpdated ('Cro', _id, _addr, _name, _data, msg.sender, block.timestamp
        );

  } else if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('
      Regulator'))) {

    require(msg.sender == lab, 'Invalid caller. Only the Lab can update a Regulator'
        );
    require (_id <= regulatorId, 'Invalid Id');
    _notRepeatAddress('Regulator', _addr, _id);
    require (_isRegulator[_regulators[_id].addr] == false, 'Regulator is already
        accepted');

    _regulators[_id].addr = _addr;
    _regulators[_id].name = _name;
    _regulators[_id].data = _data;

    emit EntityUpdated ('Regulator', _id, _addr, _name, _data, msg.sender, block.
        timestamp);

  } else if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('
      Patient'))) {

    require(msg.sender == _creator[_addr], 'Invalid caller. Only the Super can
        update a Patient');
    require (_id <= patientId, 'Invalid Id');
    _notRepeatAddress('Patient', _addr, _id);
    require (_isPatient[_patients[_id].addr] == false, 'Patient is already accepted'
        );

    _patients[_id].addr = _addr;
    _patients[_id].name = _name;
    _patients[_id].data = _data;

    emit EntityUpdated ('Patient', _id, _addr, _name, _data, msg.sender, block.
        timestamp);

  }
```

```
  }

//valida una entidad
  function approveEntity (string calldata _entity, uint256 _id) external onlyLab {

    require(_onlyEntity(_entity),'Invalid Entity. Must be Doctor, Cro, Regulator or
        Patient');

    if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('Doctor')))
         {

      require (_id <= doctorId, 'Invalid Id');

      require (_isDoctor[_doctors[_id].addr] == false, 'Doctor is already accepted');

      _isDoctor[_doctors[_id].addr] = true;

      emit EntityApproved ('Doctor', _id, _doctors[_id].addr, _doctors[_id].name,
          _doctors[_id].data, _creator[_doctors[_id].addr], block.timestamp);

    } else if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('Cro
        '))) {

      require (_id <= croId, 'Invalid Id');

      require (_isCro[_cros[_id].addr] == false, 'Cro is already accepted');

      _isCro[_cros[_id].addr] = true;

      emit EntityApproved ('Cro', _id, _cros[_id].addr, _cros[_id].name, _cros[_id].
          data, msg.sender, block.timestamp);

    } else if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('
        Regulator'))) {

      require (_id <= regulatorId, 'Invalid Id');

      require (_isRegulator[_regulators[_id].addr] == false, 'Regulator is already
          accepted');

      _isRegulator[_regulators[_id].addr] = true;

      emit EntityApproved ('Regulator', _id, _regulators[_id].addr, _regulators[_id].
          name, _regulators[_id].data, msg.sender, block.timestamp);

    } else if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('
        Patient'))) {

      require (_id <= patientId, 'Invalid Id');

      require (_isPatient[_patients[_id].addr] == false, 'Patient is already accepted'
          );

      _isPatient[_patients[_id].addr] = true;

      emit EntityApproved ('Patient', _id, _patients[_id].addr, _patients[_id].name,
          _patients[_id].data, _creator[_patients[_id].addr], block.timestamp);
```

```
    }

  }

//retorna el numero de entidades de un mismo tipo
  function numberOfEntity (string calldata _entity) external view returns (uint256) {

    require(_onlyEntity(_entity),'Invalid Entity. Must be Doctor, Cro, Regulator or
        Patient');

    if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('Doctor')))
         {

      return doctorId;

    } else if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('Cro
        '))) {

      return croId;

    } else if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('
        Regulator'))) {

      return regulatorId;

    } else if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('
        Patient'))) {

      return patientId;

    }

  }

//retorna la informacion de una entidad
  function getEntity (string calldata _entity, uint256 _id) external view returns (
      address, address, string memory, string memory, bool) {

    require(_onlyEntity(_entity),'Invalid Entity. Must be Doctor, Cro, Regulator or
        Patient');

    if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('Doctor')))
         {

      require (_id <= doctorId, 'Invalid Id');

      return (_doctors[_id].addr, _creator[_doctors[_id].addr], _doctors[_id].name,
          _doctors[_id].data, _isDoctor[_doctors[_id].addr]);


    } else if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('Cro
        '))) {

      require (_id <= croId, 'Invalid Id');

      return (_cros[_id].addr, lab, _cros[_id].name, _cros[_id].data, _isCro[_cros[_id
          ].addr]);

```

```
    } else if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('
        Regulator'))) {

      require (_id <= regulatorId, 'Invalid Id');

      return (_regulators[_id].addr, lab, _regulators[_id].name, _regulators[_id].data
          , _isRegulator[_regulators[_id].addr]);

    } else if (keccak256(abi.encodePacked(_entity)) == keccak256(abi.encodePacked('
        Patient'))) {

      require (_id <= patientId, 'Invalid Id');

      return (_patients[_id].addr, _creator[_patients[_id].addr], _patients[_id].name,
           _patients[_id].data, _isPatient[_patients[_id].addr]);

    }

  }

}
```

B.2. CT2.sol

```
pragma solidity ^0.5.0;

import "./CT1.sol";
import "openzeppelin-solidity/contracts/token/ERC721/ERC721Full.sol";

//el contrato CT2 hereda CT1 y ERC721Full
contract CT2 is CT1, ERC721Full("ClinicalTrialsToken", "CTT") {

//eventos
  event DataPatientAdded (uint256 _tokenId, address _tokenOwner, uint256 _timeStamp);
  event DataPatientChanged (uint256 _tokenId, address _tokenOwner, uint256 _timeStamp)
      ;
  event DataPointAdded (uint256 _tokenId, address _tokenOwner,  uint256 _timeStamp);
  event SetPermission (uint256 _tokenId, address _tokenOwner, address _to, uint256
      _timeStamp);
  event UnsetPermission (uint256 _tokenId, address _tokenOwner, address _to, uint256
      _timeStamp);

//variables
  uint256 tokenId;

//estructuras
  struct DataPoint {
    uint256 timestamp;
    string clinicalData;
  }

//mappings
  mapping (uint256 => address) _tokenPatient;
  mapping (uint256 => string) _tokenUri; //tokenURI is external

  mapping (uint256 => DataPoint[]) _patientRecord;
```

```
  mapping (uint256 => mapping (address => bool)) _hasPermission;

//funciona como un modificador. Es una funcion interna que verifica que el usuario que
    la
//llama es un miembro validado para visualizar un determinado registro clinico
  function _isApproved (address spender, uint256 _tokenId) internal view returns (bool
    ) {
    address owner = ownerOf(_tokenId);
    return(spender == owner ||
    spender == _creator[owner] ||
    spender == _tokenPatient[_tokenId] ||
    spender == lab ||
    _isRegulator[spender] ||
    _hasPermission[_tokenId][spender]);
  }

//a ade un registro clinico de un paciente
  function addPatientRecord (string calldata _data, address _patientAddress) external
    {

    require(_isDoctor[msg.sender], 'Invalid address');
    require(msg.sender == _creator[_patientAddress], 'Error. Address must be a patient
        of the doctor');

    tokenId = tokenId.add(1);
    _mint(msg.sender, tokenId);
    /* _setTokenURI(tokenId, _data); */
    _tokenUri[tokenId] = _data;
    _tokenPatient[tokenId] = _patientAddress;

    emit DataPatientAdded (tokenId, msg.sender, block.timestamp);

  }

//modifica un registro clinico
  function changePatientRecord (uint256 _tokenId, string calldata _data, address
    _patientAddress) external {

    require(ownerOf(_tokenId) == msg.sender, 'Invalid address');
    require(msg.sender == _creator[_patientAddress], 'Error. Address must be a patient
        of the doctor');

    /* _setTokenURI(_tokenId, _data); */
    _tokenUri[_tokenId] = _data;
    _tokenPatient[_tokenId] = _patientAddress;

    emit DataPatientChanged (_tokenId, msg.sender, block.timestamp);

  }

//retorna un registro clinico
  function getPatientRecord (uint256 _tokenId) external view returns (string memory,
    address) {

    //Descomentar esto si se quiere pedir acceso a la ficha del paciente
    /* require(_isApproved(msg.sender, _tokenId), 'Invalid address. You are not
       approved'); */
```

```
    /* return tokenURI(_tokenId); */
    return (_tokenUri[_tokenId], _tokenPatient[_tokenId]);

  }

//retorna la cantidad de registros medicos
  function numberOfPatientRecord () external view returns (uint256) {
    return tokenId;
  }

//a ade un dato clinico al registro de un paciente
  function addDataPoint (uint256 _tokenId, string calldata _clinicalData) external {

    require(ownerOf(_tokenId) == msg.sender, 'Invalid address');

    DataPoint memory dp;
    dp.timestamp = now;
    dp.clinicalData = _clinicalData;
    _patientRecord[_tokenId].push(dp);

    emit DataPointAdded (_tokenId, msg.sender, block.timestamp);

  }

//retorna un dato clinico, siempre y cuando el usuario que llame a la funcion tenga
    los
//permisos correspondientes
  function getDataPoint (uint256 _tokenId, uint256 _dataPointId) external view returns
      (string memory) {

    require(_isApproved(msg.sender, _tokenId), 'Invalid address. You are not approved'
        );

    return _patientRecord[_tokenId][_dataPointId].clinicalData;

  }

//retorna el numero de datos a adidos a un registro clinico
  function numberOfDataPoint (uint256 _tokenId) external view returns (uint256) {

    require(_isApproved(msg.sender, _tokenId), 'Invalid address. You are not approved'
        );

    return _patientRecord[_tokenId].length;

  }

//establece permiso de visualizacion de un registro a un usuario
  function setPermission (uint256 _tokenId, address _newAddress) external {

    require(ownerOf(_tokenId) == msg.sender, 'Invalid address');

    _hasPermission[_tokenId][_newAddress] = true;

    emit SetPermission (_tokenId, msg.sender, _newAddress, block.timestamp);

  }

//quita el permiso de visualizacion de un registro a un usuario
```

```
  function unsetPermission (uint256 _tokenId, address _address) external {

    require(ownerOf(_tokenId) == msg.sender, 'Invalid address');

    _hasPermission[_tokenId][_address] = false;

    emit UnsetPermission (_tokenId, msg.sender, _address, block.timestamp);

  }

}
```

Apéndice C

Práctica Profesional Supervisada y actividades de Proyecto y Diseño

La Práctica Profesional Supervisada fue llevada a cabo en el Instituto de Tecnologías Nucleares para la Salud (INTECNUS) en el Centro Atómico Bariloche, durante el último año de la carrera de Ingeniería Mecánica. Fue realizada bajo la supervisión del Dr. Flavio Colavecchia.

Las actividades de Proyecto y Diseño realizadas por el alumno para desarrollar el Proyecto Integrador fueron:

- Adquisición de conocimiento para el manejo de las herramientas necesarias para el desarrollo de las aplicaciones. Pruebas de modelos sencillos de contratos inteligentes.
- Desarrollo de una aplicación para digitalizar y gestionar la emisión, visualización y validación de los títulos académicos del Instituto Balseiro.
- Establecimiento de requerimientos a cumplir a partir del conocimiento de las necesidades planteadas.
- Codificación del contrato inteligente para el manejo de los datos clínicos.
- Realización de la interfaz de usuario. Evaluación del diseño implementado.

Bibliografía

[1] ANMAT. Investigaciones clínicas farmacológicas, 2019. URL https://www.argentina.gob.ar/anmat/regulados/investigaciones-clinicas-farmacologicas. 1

[2] Spilker, B. Guide to Clinical Trials. *Raven Press*, págs. 22–23, 1984. 1

[3] Wong, D. R., Bhattacharya, S., Butte, A. J. Prototype of running clinical trials in an untrustworthy environment using blockchain. *Nature Communications*, **10** (917), 2, 2019. 2

[4] Brooks, R. R., Wang, K. C., Yu, L., Oakley, J., Skjellum, A., SimCenter, *et al.* Scrybe : A blockchain ledger for clinical trials. *IEEE*, págs. 1–2, 2018. 2

[5] Benchoufi, M., Ravaud, P. Blockchain technology for improving clinical research quality. *Trials*, **18** (335), 1–3, 2017. 2, 5

[6] Hume, S., Sarnikar, S., Becnel, L., Bennedett, D. Visualizing and Validating Metadata Traceabilidty within the CDISC Standars. *AMIA Jt Summits Transl Sci Proc*, págs. 1–3, 2017. 2

[7] Palombini, M. How Blockchain Could Revolutionize Clinical Trials/Research while Enhancing Patient Safety and Engagement. *IEEE*, pág. 1, 2017. 2

[8] Irving, G., Holden, J. How blockchain-timestamped protocols could improve the trustworthiness of medical science. *F1000Res*, **5** (222), 2, 2017. 2

[9] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system, Dec 2008. URL https://bitcoin.org/bitcoin.pdf. 2, 4

[10] Hans, R., Zuber, H., Rizk, A., Steinmetz, R. Blockchain and Smart Contracts: Disruptive Technologies for the Insurance Market. *AMCIS*, págs. 1–2, 2017. 2

[11] Buterin, V. Ethereum: A next-generation smart contract and decentralized application platform, 2014. URL https://github.com/ethereum/wiki/wiki/White-Paper. 3, 7

[12] Perez, E. Mecanismos de consenso en blockchain: PoW, PoS, PoI, 2019. URL https://www.tucriptomoneda.com/mecanismos-de-consenso-en-blockchain-pow-pos-poi/. 4

[13] Wikipedia. Contrato inteligente, 2019. URL https://es.wikipedia.org/wiki/Contrato_inteligente. 5

[14] Stanley, A. Big pharma seeks dlt solution for drug costs, 2018. URL https://www.coindesk.com/blockchain-day-big-pharma-seeks-dlt-solution-drug-costs. 5

[15] Wood, G. Ethereum: A secure decentralised generalised transaction ledger byzantium version 3e36772 - 2019-05-12), 2014. URL https://ethereum.github.io/yellowpaper/paper.pdf. 7

[16] Ethereum, M. Solidity: Todos los recursos sobre el lenguaje de programación de los smart contracts, 2018. URL https://www.miethereum.com/smart-contracts/solidity/. 7

[17] Etherscan. Ethereum gas tracker, 2019. URL https://etherscan.io/gastracker. 8

[18] Kasireddy, P. How does Ethereum work, anyway?, 2017. URL https://medium.com/@preethikasireddy/how-does-ethereum-work-anyway-22d1df506369. 8

[19] Entriken, W. ERC-20 Token Standard, 2015. URL https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md. 9

[20] Surga, J. ¿Qué son los Tokens ERC20 de Ethereum y cómo funcionan?, 2017. URL https://www.criptonoticias.com/colecciones/tokens-erc20-ethereum-como-funcionan/. 9

[21] Entriken, W. ERC-721 Token Standard, 2018. URL https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md. 9

[22] Nash, G. The Anatomy of ERC721, 2017. URL https://medium.com/crypto-currently/the-anatomy-of-erc721-e9db77abfc24. 9

[23] Solidity. Solidity documentation, 2019. URL https://solidity-es.readthedocs.io/es/latest/. 11

[24] web docs, M. Primeros pasos con javascript, 2019. URL https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps. 11

[25] Terms, T. Javascript, 2014. URL https://techterms.com/definition/javascript. 11

[26] freeCodeCamp. What exactly is Node.js?, 2018. URL https://www.freecodecamp.org/news/what-exactly-is-node-js-ae36e97449f5/. 11

[27] Larrañaga Cahis, F. Tú, yo y package.json, 2018. URL https://medium.com/noders/t%C3%BA-yo-y-package-json-9553929fb2e3. 12

[28] Suite, T. Ganache quickstart, 2019. URL https://www.trufflesuite.com/docs/ganache/quickstart. 12, 62

[29] Benet, J. IPFS - Content Addresses, Versioned, P2P File System (draft 3), year = 2017. *IPFS*, págs. 1–2. 14

[30] de Información Universitaria, S. Siu guaraní, 2018. URL https://www.siu.edu.ar/siu-guarani/. 15

[31] Nación, D. L. La UBA investigará los títulos fraguados, 1997. URL https://www.lanacion.com.ar/sociedad/la-uba-investigara-los-titulos-fraguados-nid65882. 15

[32] Negro, D. R. Condenan al ingeniero trucho descubierto por “Río Negro”, 2018. URL https://www.rionegro.com.ar/condenan-al-ingeniero-trucho-descubierto-por-rio-negro-ED5849978/. 16

[33] Zeppelin, O. Github: Open zeppelin, 2019. URL https://github.com/OpenZeppelin/openzeppelin-solidity. 19, 62

[34] Solidity. Solidity documentation: types, 2019. URL https://solidity.readthedocs.io/en/v0.5.9/types.html. 20

[35] Community, E. Solidity documentation, 2018. URL https://solidity.readthedocs.io/en/latest/contracts.html#modifiers. 22

[36] ANMAT. Investigaciones clínicas farmacológicas: Normativa, 2019. URL https://www.argentina.gob.ar/anmat/regulados/investigaciones-clinicas-farmacologicas/normativa. 37

[37] Repositorio del proyecto, 2019. URL https://gitlab.com/ib-trials/. 55, 65

[38] Peh, B. The Basics of Upgradable Proxy Contracts in Ethereum, 2018. URL https://medium.com/@blockchain101/the-basics-of-upgradable-proxy-contracts-in-ethereum-479b5d3363d6. 59

[39] BFA. Blockchain Federal Argentina: una plataforma multiservicios sólida, transparente y confiable, 2019. URL https://bfa.ar/. 59

[40] NojeJS. Descarga de NodeJS, 2019. URL https://nodejs.org/. 62

[41] Webpack. Webpack installation, 2019. URL https://webpack.js.org/guides/installation/. 62

[42] Furter, S. web3.js - Ethereum JavaScript API, 2019. URL https://github.com/ethereum/web3.js/. 62

[43] Truffle. Truffle Overview, 2019. URL https://www.trufflesuite.com/docs/truffle/overview. 62

[44] IPFS. Ipfs documentation, 2019. URL https://docs.ipfs.io/introduction/usage/. 62

[45] Google. Descargar chrome, 2019. URL https://www.google.com/intl/es/chrome/. 62

Agradecimientos

A mi familia, en especial a mis viejos, por su inmenso amor y sacrificio, por inculcarme los valores que me hicieron ser lo que soy y que me han hecho llegar hasta donde he llegado.

A mis amigos, hermanos de toda la vida, que estuvieron presentes incluso en la distancia, y que forman parte de mi ser.

A los de acá, maravillosas personas que conocí y que me hicieron disfrutar de esta etapa en Bariloche.

A mi director, Flavio, por su infinita paciencia y predisposición, y por ayudarme a terminar esta carrera de la mejor manera.