Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Notarización de documentos con Ethereum

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Loladze, George

Tutor: Muñoz Escoí, Francisco Daniel

Tutor externo: Beyer, Stefan

2019/2020

Resumen

En este TFG se va a exponer un sistema de notarización de documentos digitales aprovechando las cualidades de transparencia e inmutabilidad de Ethereum para almacenar los registros.

Se va a presentar 2 posibles arquitecturas para este sistema. Siendo la gestión de las cuentas Ethereum la principal diferencia.

En el primero, el usuario debe tener una cuenta de Ethereum con fondos para interactuar con el sistema. Los dos componentes fundamentales son: la interfaz web y el Smart Contract en Ethereum. El usuario se conecta al sistema en la página web, usando un Wallet específico de navegador (MetaMask). Desde la web podrá guardar, recuperar o validar cualquier registro en Ethereum.

El segundo diseño, no será necesario que el cliente tenga su propia cuenta de Ethereum. En su lugar, habrá un servidor que identificará a los usuarios e interactuará con Ethereum en su nombre.

Por último, se hablará de un sistema de gestión de identidades, basado en el estándar de W3C, el cual le dará una capa extra de seguridad a nuestro sistema de notarización.

Palabras clave: Notaría, firma digital, MetaMask, web3, Ethereum, DID

Abstract

In this final project a system of digital notarization is going to be presented. Transparency and immutability are a couple of qualities of Ethereum, and for that reason, it will be used as a storage to keep our records. Two possible architectures will be shown.

In the first model the final user must have an Ethereum account and funds to interact with Ethereum. The principal components will be a web page and the smart contract on Ethereum. Users will be given the choice to connect to our system on the web site through MetaMask, a browser wallet. MetaMask will oversee the user's account management, sign documents, and do the transactions to Ethereum. From the web site, the user will be able to read, store and validate the records kept on the smart contract.

In the second design, the user does not need to have his own Ethereum account. Instead, a server will act as a proxy between the users and the smart contract on Ethereum. A user ID will be assigned to each individual and the server will sign, store, read or validate the documents on behalf of those individuals, keeping in consideration their user ID.

In addition, we will introduce a system of identity management, based on a W3C standard. This identity management will add an extra layer of security to our notary project.

Keywords: Notary, digital signature, MetaMask, web3, Ethereum, DID.

Tabla de contenidos

Tabla de ilustraciones

1. Introducción

El blockchain es una tecnología emergente que ya está impactando a una gran variedad de sectores, entre ellos: la gestión de bienes, los mercados de valores, las finanzas descentralizadas (DeFi), el comercio internacional, los pagos, el dinero y el mercado inmobiliario. Gracias a su estructura de datos, se puede usar como una base de datos pública en un entorno distribuido, y sobre todo con garantías de integridad respaldadas mediante técnicas criptográficas.

El sistema desarrollado en este trabajo, aprovechando las características de inmutabilidad y transparencia del blockchain, se usará para probar la existencia de registros en determinado momento y garantizar que no hayan sido alterados.

1.1 Motivación

La mayoría de los trámites con la administración requieren de largas esperas. Primero para conseguir que te atiendan y luego para recibir el documento final, sobre todo si es necesaria la participación de varios órganos de la administración para la tramitación de dicho documento. El avance tecnológico de las últimas décadas ha permitido gestionar multitud de trámites desde la comodidad de nuestras casas.

Una de las muchas funciones del notario es la de la legitimación notarial de firma. Esto es usado para acreditar la pertenencia de dicha firma a una persona determinada y también para dar fecha fehaciente a un documento. Con nuestro sistema, simularemos esta función, sin costes excesivos ni largos tiempos de espera en las gestiones.

1.2 Objetivos

En los próximos párrafos describiremos los objetivos esperados para cumplir de forma satisfactoria con el desarrollo de este proyecto.

El primer objetivo que nos propondremos como meta será el de introducir la tecnología blockchain. Deberemos poder responder a preguntas como las siguientes: ¿cómo funciona?, ¿qué tipos de blockchain hay?, ¿cuáles son los casos de uso más comunes?

El siguiente objetivo que buscaremos alcanzar será extender la explicación del caso de uso elegido para este proyecto, la legitimación notarial de firma. Debemos exponer cómo se ha estado desarrollando esta tarea hasta el momento, qué desventajas tiene y cuáles son las ventajas que ofrece el desarrollar esta tarea haciendo uso de nuestra solución.

Otro objetivo que nos planteamos es reducir la complejidad que pueda aportar el uso de la tecnología blockchain para el usuario final. Para esto, propondremos realizar dos diseños, demostrando cómo mitigar esta complicación añadida.

Por último, como objetivo secundario, nos propondremos explicar el concepto de gestión de identidades y lo que podría aportar su implementación en nuestros sistemas.

1.3 Impacto Esperado

Con el desarrollo de este servicio de notarización descentralizada se espera dar a los usuarios una alternativa económica, segura y transparente, para probar la propiedad de documentos digitales.

Por otra parte, el desarrollo de este proyecto me aporta experiencia en una amplia gama de disciplinas. Me permite, además de profundizar en el funcionamiento interno de Ethereum[1] (un blockchain público referente), hacer uso de una gran variedad de herramientas de distintos roles (back-end, front-end, devOps). A lo largo de este proyecto, veré:

- Cómo crear un contrato inteligente, desplegarlo en la red de Ethereum y conectarlo con nuestro servicio. Para esto usaré frameworks como:
  - Truffle[2]
  - Web3.js[3]
  - MetaMask[4]
- Hacer uso de los servicios de AWS[5] para desplegar el sistema
  - EC2[6] (Elastic Compute Cloud) para hospedar el servidor.
  - Express[7] para diseñar las rutas (Rest API).
  - Node.js[8] para crear la lógica del servidor.
  - MongoDB[9] como base de datos NoSQL
  - Json Web Token (JWT[10]) para la autenticación de usuarios
  - Route 53[11] (DNS), traducirá la URL

1.4 Metodología

El proyecto estará repartido en las siguientes fases:

- Primero, se comienza con una investigación teórica sobre el blockchain en general y posteriormente en Ethereum en particular.
- A continuación, nos familiarizamos con las herramientas más comunes para el desarrollo de Smart Contracts (contratos inteligentes).
- Luego, analizaremos nuestro caso de uso y los componentes necesarios para su implementación.
- Proseguiremos con el diseño y la implementación del servicio.
- Y finalizaremos con el despliegue del sistema y las correspondientes pruebas de funcionamiento.

[1] https://ethereum.org/en/whitepaper/
[2] https://www.trufflesuite.com/
[3] https://web3js.readthedocs.io/en/v1.2.11/
[4] https://metamask.io/
[5] https://aws.amazon.com/es/
[6] https://aws.amazon.com/es/ec2/
[7] https://expressjs.com/es/
[8] https://nodejs.org/es/
[9] https://www.mongodb.com/es
[10] https://jwt.io/
[11] https://aws.amazon.com/es/route53/

A continuación, mostramos un diagrama Gantt con la planificación del proyecto (las fechas y tiempos son estimados). Para una mejor visualización de la gráfica, desglosaremos las tareas de la siguiente forma:

Estudio del blockchain

- Tarea 1 – Analizar los componentes principales de un blockchain.
- Tarea 2 – Estudiar los conceptos de clave pública/privada y sistema de consenso.

Análisis del funcionamiento de Ethereum

- Tarea 3 – Conocer las bases de Ethereum.
- Tarea 4 – Revisar el algoritmo de consenso.
- Tarea 5 – Investigar los contratos inteligentes.

Desarrollo y despliegue de contratos inteligentes

- Tarea 6 – Aprender a programar con el lenguaje Solidity[12].
- Tarea 7 – Familiarizarse con el framework de desarrollo de contratos inteligentes Truffle Suite.
- Tarea 8 – Usar Ganache[13] para simular un blockchain local y desplegar contratos inteligentes en él.
- Tarea 9 – Preparar una librería para hacer uso de los contratos inteligentes desplegados.

Análisis del caso de uso y propuesta de solución

- Tarea 10 – Documentarse sobre la legitimación de firma notarial.
- Tarea 11 – Proponer un diseño para la solución(componentes).

Desarrollo del sistema (Modelo 1 – Con MetaMask)

- Tarea 12 – Escribir y desplegar el contrato inteligente.
- Tarea 13 – Preparar un código base para la estructura de la página web.
- Tarea 14 – Darle funcionalidad

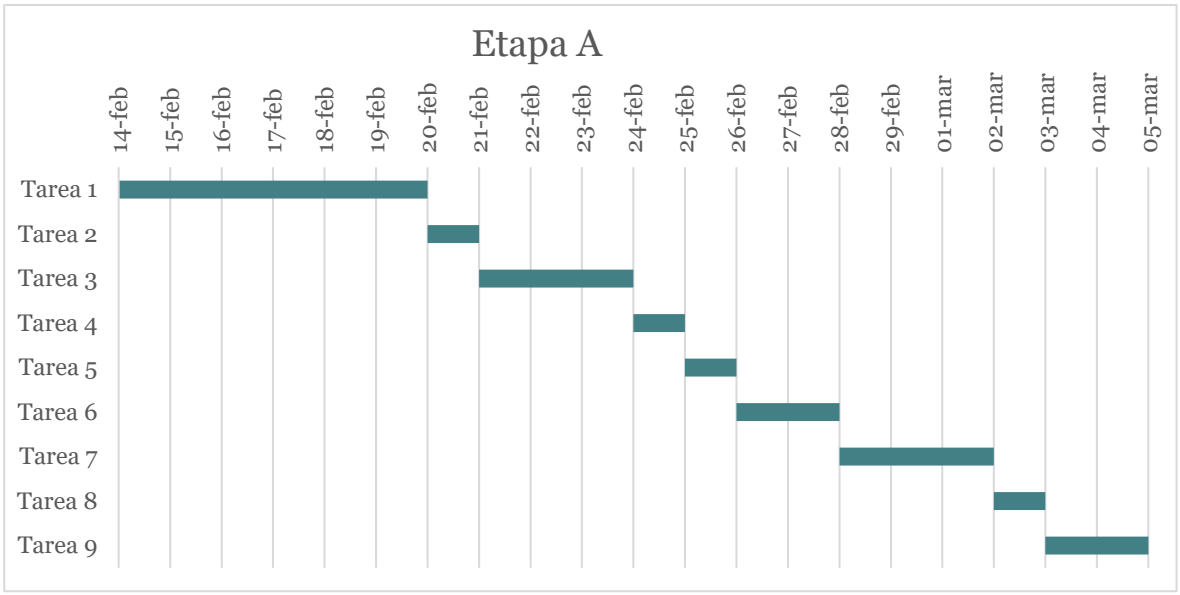
Desarrollo del sistema (Modelo 2 – Con servidor para gestionar los usuarios)

- Tarea 15 – Escribir y desplegar el contrato inteligente
- Tarea 16 – Preparar un sistema de login (JWT)
- Tarea 17 – Programar la lógica del servidor en nodejs
- Tarea 18 – Conectar con la base de datos MongoDB (usando mongoose[14])

Despliegue y testeo de los 2 modelos

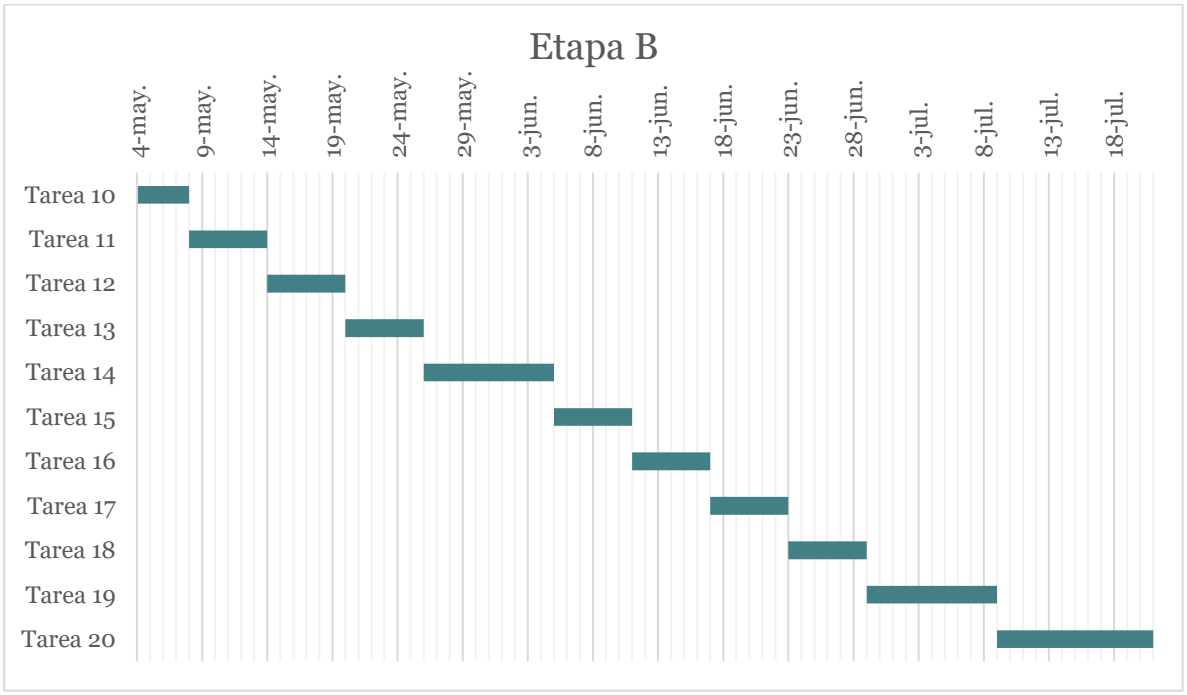
- Tarea 19 – Desplegar el modelo 1, testear y corregir los errores.
- Tarea 20 – Desplegar el modelo 2, testear y corregir los errores.

[12] https://solidity.readthedocs.io/en/latest/
[13] https://www.trufflesuite.com/docs/ganache/overview
[14] https://mongoosejs.com/



En esta primera etapa, las tareas se enfocan en la parte de aprendizaje y asimilación de las tecnologías a usar.

La segunda etapa, más relacionada con el desarrollo técnico del proyecto, debido al desarrollo de otros proyectos, se retoma más tarde. Además, la dedicación diaria ya no es completa, pasando a ser de alrededor de 2 horas diarias.



Adjuntamos la tabla para añadir mayor claridad en la planificación.

Nombre de la actividad	Fecha de inicio	Duración en horas	Duración en días	Fecha de fin	Duración en días (contando no laborables)
Columna1	**Columna2**	**Columna3**	**Columna4**	**Columna6**	**Columna7**
Tarea 1	14-feb	24	4	20-feb.	6
Tarea 2	20-feb	8	1	21-feb.	1
Tarea 3	21-feb	8	1	24-feb.	3
Tarea 4	24-feb	4	1	25-feb.	1
Tarea 5	25-feb	4	1	26-feb.	1
Tarea 6	26-feb	16	2	28-feb.	2
Tarea 7	28-feb	8	1	2-mar.	3
Tarea 8	02-mar	8	1	3-mar.	1
Tarea 9	03-mar	16	2	5-mar.	2
Tarea 10	4-may.	8	4	8-may.	4
Tarea 11	08-may	8	4	14-may.	6
Tarea 12	14-may	8	4	20-may.	6
Tarea 13	20-may	8	4	26-may.	6
Tarea 14	26-may	16	8	5-jun.	10
Tarea 15	05-jun	8	4	11-jun.	6
Tarea 16	11-jun	8	4	17-jun.	6
Tarea 17	17-jun	8	4	23-jun.	6
Tarea 18	23-jun	8	4	29-jun.	6
Tarea 19	29-jun	16	8	9-jul.	10
Tarea 20	09-jul	16	8	21-jul.	12

1.5 Estructura

El trabajo se distribuye de la siguiente forma:

1. **Introducción:** Un breve apartado donde se explica el proyecto a rasgos generales, su contenido, objetivos y estructura.
2. **Estado del arte:** En esta sección se comenta el contexto tecnológico. Se hablará de aplicaciones que realicen similar funcionalidad a la propuesta en este trabajo de fin de grado. También se explicará las ventajas de nuestra solución en contraste a las del mercado actual.
3. **Diseño de la solución:** El objetivo de este capítulo es mostrar los componentes que formarán la solución, explicar qué función realiza cada uno de ellos y por qué han sido elegidos.
4. **Desarrollo de la solución:** Aquí se ve una descripción en detalle de cómo se ha ido desarrollando el proyecto y de qué modo interactúan los componentes para dar la funcionalidad esperada.
5. **Implantación:** Una explicación de cómo se realiza el despliegue del sistema paso a paso.
6. **Pruebas:** Demostración de la solución final desde el punto de vista del usuario.

7. **Trabajos futuros:** En este apartado se introduce el concepto de identificadores descentralizados (DID) y cómo se podría integrar con nuestro sistema de notarización.
8. **Conclusiones:** Breves reflexiones sobre el desarrollo del proyecto, dificultades encontradas y soluciones buscadas.
9. **Referencias:** En este capítulo listamos todas las fuentes usadas a lo largo del proyecto.
10. **Glosario:** Catálogo de términos y expresiones que requieran de aclaración.
11. **Anexos:** En este apartado se coloca el código del contrato inteligente y un enlace al repositorio público para satisfacer la curiosidad del lector.

2. Estado del arte

2.1 El rol del notario

"El notario como funcionario público es una garantía para el ejercicio de las libertades individuales y patrimoniales, en cualquier situación, incluso frente a poderes públicos" [2].

Hay multitud de definiciones de la palabra notario, pero, una definición formal de esta podría ser la siguiente: "El notario es un profesional del derecho que ejerce una función pública para robustecer, con una presunción de verdad, los actos en que interviene, para colaborar en la formación correcta del negocio jurídico y para solemnizar y dar forma legal a los negocios jurídicos, y de cuya competencia, solo por razones históricas están sustraídos los actos de la llamada jurisdicción voluntaria" [3].

En resumidas cuentas, el notario garantiza la legitimidad de los documentos en los que participa, dándoles autorización pública. Podríamos sintetizar su función en la realización de dos clases de documentos genéricos, la Escritura y el Acta, siendo la escritura una agrupación de actas [4].

Los documentos comúnmente redactados por los notarios son [5]:

- Testamento
- Herencia
- Capitulaciones matrimoniales
- Bodas, separaciones y divorcios
- Poder
- Actas
- Compraventa
- Préstamo hipotecario
- Préstamo personal
- Donación
- Constitución de sociedades mercantiles
- Póliza
- Protesto
- Reclamación de deudas
- Conciliación
- Legitimación de firma

2.2 Blockchain

El blockchain se da a conocer en el año 2008 con la publicación de un artículo titulado "Bitcoin: A Peer-to-Peer Electronic Cash System" bajo el alias de Satoshi Nakamoto [6]. Tomando como bases sistemas como b-money[15] y HashCash[16], Nakamoto crea un sistema de dinero electrónico completamente descentralizado, sorteando así intermediarios como los bancos.

[15] http://www.weidai.com/bmoney.txt
[16] http://www.hashcash.org/

El componente clave es un algoritmo llamado "Proof-Of-Work". Esto es un sistema de computación distribuida que se encarga de mantener en común (consenso) el estado global de las transacciones mediante "elecciones" periódicas. De este modo, se resuelve el problema del doble gasto, el cual suponía una de las debilidades que impedían el desarrollo del dinero electrónico, ya que la misma unidad de dinero se podría gastar varias veces si no se controlaba adecuadamente.

El blockchain se podría entender como un registro distribuido implementado usando estructuras de datos de listas enlazadas criptográficamente, siendo un registro distribuido una base de datos de transacciones mantenidas consistentemente sobre una red peer-to-peer [7]. Los componentes principales son:

- Red P2P, una red descentralizada donde no se sigue un modelo típico de cliente-servidor, si no que todos los participantes hacen tanto de clientes como de servidores.
- Mensajes a modo de transacciones que representan cambios de estado.
- Una serie de reglas de consenso que rigen qué es una transacción y qué cambio de estado es válido.
- Una máquina de estados que procese transacciones bajo las reglas de consenso.
- Una cadena de bloques criptográficamente asegurados que actúan como registro de todos los cambios de estado aceptados y validados.
- Un algoritmo de consenso que descentralice el control del blockchain, forzando a los participantes cooperar para mantener el cumplimiento de las reglas de consenso.
- En blockchains públicas, un sistema de incentivación para mantener el estado de la máquina en un entorno abierto.

Podemos clasificar el blockchain en tres tipos representativos, dependiendo del nivel de exposición al uso público.

- **Blockchain público:** en este modelo, cualquiera puede acceder a la red y participar en el consenso. Las transacciones son anónimas, pero completamente transparentes. Es un registro completamente descentralizado, seguro e inmutable. El ejemplo claro de blockchain público es Bitcoin, pero también son bien conocidos Ethereum o Litecoin[17] entre otros.
- **Blockchain federado/consorcio:** este tipo de red suele ser cerrada. Los participantes deben recibir permiso para formar parte de la red. No necesita un sistema de incentivación y tiene la capacidad de realizar un mayor número de transacciones por segundo en comparación al blockchain público. Este modelo se suele llevar a cabo cuando múltiples empresas colaboran en un proyecto y requieren un registro común. Un ejemplo sería el IBM Food Trust[18].
- **Blockchain privado:** este modelo es similar al anterior, tiene las mismas ventajas de velocidad y escalabilidad, pero con la diferencia de que en lugar de ser varias empresas rigiendo la red, tan sólo hay una entidad al mando. Ejemplos de este modelo serían Multichain[19], Corda[20], Hyperledger fabric[21], etc.

2.3 Ethereum

Ethereum es un blockchain público de uso genérico que fue concebido por Vitalik Buterin en el artículo publicado en el año 2013 [8]. El proyecto fue lanzado en el año 2015 y el objetivo era

[17] https://litecoin.org/es/
[18] https://www.ibm.com/uk-en/blockchain/solutions/food-trust
[19] https://www.multichain.com/
[20] https://www.corda.net/
[21] https://www.hyperledger.org/use/fabric

dar la posibilidad a los programadores de desarrollar aplicaciones sobre esta plataforma sin tener que preocuparse de desarrollar su propio blockchain, algoritmo de consenso, etc. Los componentes del blockchain que hemos mencionado en el apartado anterior son implementados por los siguientes protocolos en Ethereum [9]:

- Red P2P: Lo que se conoce como la red principal de Ethereum (Ethereum's main net) es accesible mediante TCP por el puerto 30303 y el protocolo es llamado DEVp2p[22].
- Transacciones: Las transacciones de Ethereum son mensajes de la red que incluyen, entre otros, información sobre el remitente, destinatario, valor y datos en el payload.
- Máquina virtual de Ethereum: Los cambios de estado son procesados por la máquina virtual de Ethereum (EVM), una máquina virtual basada en pila que ejecuta bytecodes (instrucciones en lenguaje máquina). Los programas de EVM son llamados "Smart Contracts" y son escritos en un lenguaje de alto nivel (Solidity, Vyper[23] …), compilados a lenguaje máquina y posteriormente ejecutados por la máquina virtual de Ethereum.
- Estructura de datos: los datos de Ethereum se almacenan en una base de datos localmente en cada uno de los nodos. Esos datos contienen transacciones y estados del sistema en una estructura de datos conocida como árbol hash de Merkle Patricia (Merkle Patricia Tree[24]).
- Algoritmo de consenso: Ethereum usa el modelo de consenso de Bitcoin, Consenso de Nakamoto, el cual usa bloques secuenciales de firma única, cuyo valor es sopesado por el protocolo "Proof of Work" y quedándose con la cadena más larga como estado actual del sistema.
- Seguridad económica: para incentivar la participación en la red, Ethereum usa el protocolo "Proof of Work" llamado Ethash. Se espera cambiar este algoritmo a un "Proof of Stake" eventualmente.
- Clientes: Ethereum dispone de multitud de implementaciones interoperables de clientes, siendo los más reconocidos Go-Ethereum (Geth) y Parity.

2.4 Caso de uso: legitimación notarial de firma

La legitimación de firma sirve para acreditar el hecho de que una firma ha sido puesta en presencia del notario, o el juicio de este sobre su pertenencia a persona determinada. También para dar fecha fehaciente a un documento [10 y 11].

Uno de los muchos casos de uso del Blockchain es lo que se conoce como "Proof of Existence". Esto consiste en almacenar el hash de un documento en la red, aprovechando las cualidades de inmutabilidad de esta tecnología, quedando demostrada la existencia en la red de un documento en particular. Si a este concepto le añadimos la capacidad de probar la propiedad de la cuenta (identificador único de usuario) que firma dicho documento, el resultado es un sistema capaz de simular la legitimación notarial de firma [12].

Lo que nos permite realizar la prueba de pertenencia de una cuenta es el sistema de llaves pública/privada que usa Ethereum. Los usuarios disponen de un par de llaves cuya generación está basada en la curva elíptica ecp256k1. La que se conoce como llave privada es usada para generar la llave pública, la dirección Ethereum y la función que hace de firma digital. Gracias al funcionamiento de la criptografía de curva elíptica, cualquiera puede comprobar la validez de una transacción.

[22] https://github.com/ethereum/devp2p
[23] https://vyper.readthedocs.io/en/stable/
[24] https://merkle-patricia-trie.readthedocs.io/en/latest/

2.5 Estudio del mercado

En los siguientes párrafos se va a listar una serie de servicios que se ofrecen en internet con relación al caso de uso mencionado en el apartado anterior.

DocStamp[25]: es un servicio que almacena el hash del documento, junto al mail del usuario que servirá para probar la propiedad, en Ethereum. El coste de la transacción y la firma lo realiza el proveedor del servicio. Por este motivo, en el momento de realizar el registro se requiere información de la tarjeta bancaria para realizar el pago.

ETH Notary[26]: este servicio guarda el hash del documento que pretendemos registrar en Ethereum. A diferencia del anterior, no requiere de correo ni de información bancaria. Esto es debido a que el coste que genera el uso del Blockchain recae en el usuario. Por eso, para hacer uso de este servicio debes tener MetaMask y estar conectado a tu cuenta con fondos disponibles.

2.6 Propuesta de proyecto

El proyecto que vamos a desarrollar tendrá la misma funcionalidad. Guardará el hash de un documento en la red de pruebas de Ethereum (Rinkeby). Tendrá dos variaciones, la primera variación será similar al servicio ETH Notary ya mencionado. En esta variación, el usuario debe contar con una cuenta Ethereum con fondos y estar conectado a MetaMask para hacer uso de nuestro servicio. La segunda, contará con un servidor que gestionará los usuarios registrados en el sistema y se encargará de la firma de las transacciones en nombre de los usuarios.

El objetivo de tener estos dos modelos es reducir la complejidad que conlleva la asimilación de una nueva tecnología (no todos los usuarios dispondrán de una cuenta Ethereum).

2.7 Proyectos relacionados

En esta sección vamos a ver una serie de trabajos de fin de grado/máster que puedan tener relevancia con la temática de este proyecto. Los trabajos que se han considerado son los siguientes:

Desarrollo de contratos inteligentes basados en el sistema Ethereum – Diego Sales Bellés (2019). En este documento se realiza una introducción en la tecnología Blockchain y el desarrollo de contratos inteligentes, por ese motivo se incluye en esta lista.

Análisis de la tecnología Blockchain y desarrollo de una implementación en Java José Blasco Núñez de Cela (2018). La razón de mencionar este trabajo es debido a que realiza un desarrollo de una plataforma de prueba de existencia (Proof of Existence).

Implantación de la tecnología Blockchain en infraestructura serverless – Daniel Ortiz Sánchez. Este trabajo de fin de máster tiene conexión con el modelo con back-end de nuestra solución, ya que pretendemos realizar el despliegue en una infraestructura serverless.

[25] https://docstamp.io
[26] https://lab.miguelmota.com/ethnotary/public/

3. Diseño de la solución

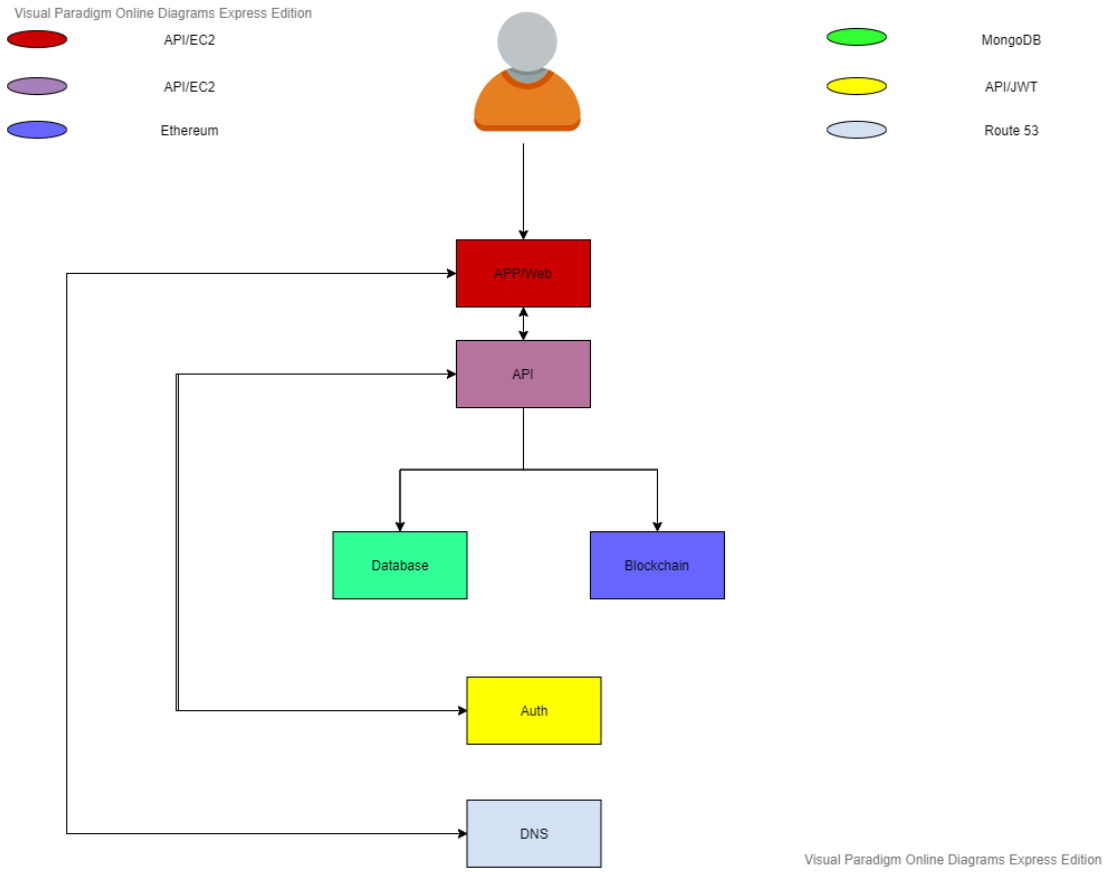
Nuestra solución mantiene un registro de los documentos, con sus respectivas firmas y marcas temporales, en el blockchain público Ethereum. Tenemos 2 diseños:

Diseño con back-end: En esta solución, hemos decidido mantener un esquema similar al de los sistemas actuales, donde los usuarios se registran y acceden al sistema con las típicas credenciales de usuario y contraseña. Además, el servidor será el encargado de gestionar toda la complejidad que pueda aportar en la solución el blockchain. Esto se ha realizado de este modo para evitar abrumar a usuarios sin conocimientos básicos de Ethereum.

Diseño sin back-end: El segundo diseño está orientado a usuarios que hayan tenido al menos un contacto mínimo con Ethereum. El motivo de esto es debido a que la página web interactúa con el blockchain usando MetaMask para gestionar sus firmas.

3.1 Arquitectura del Sistema

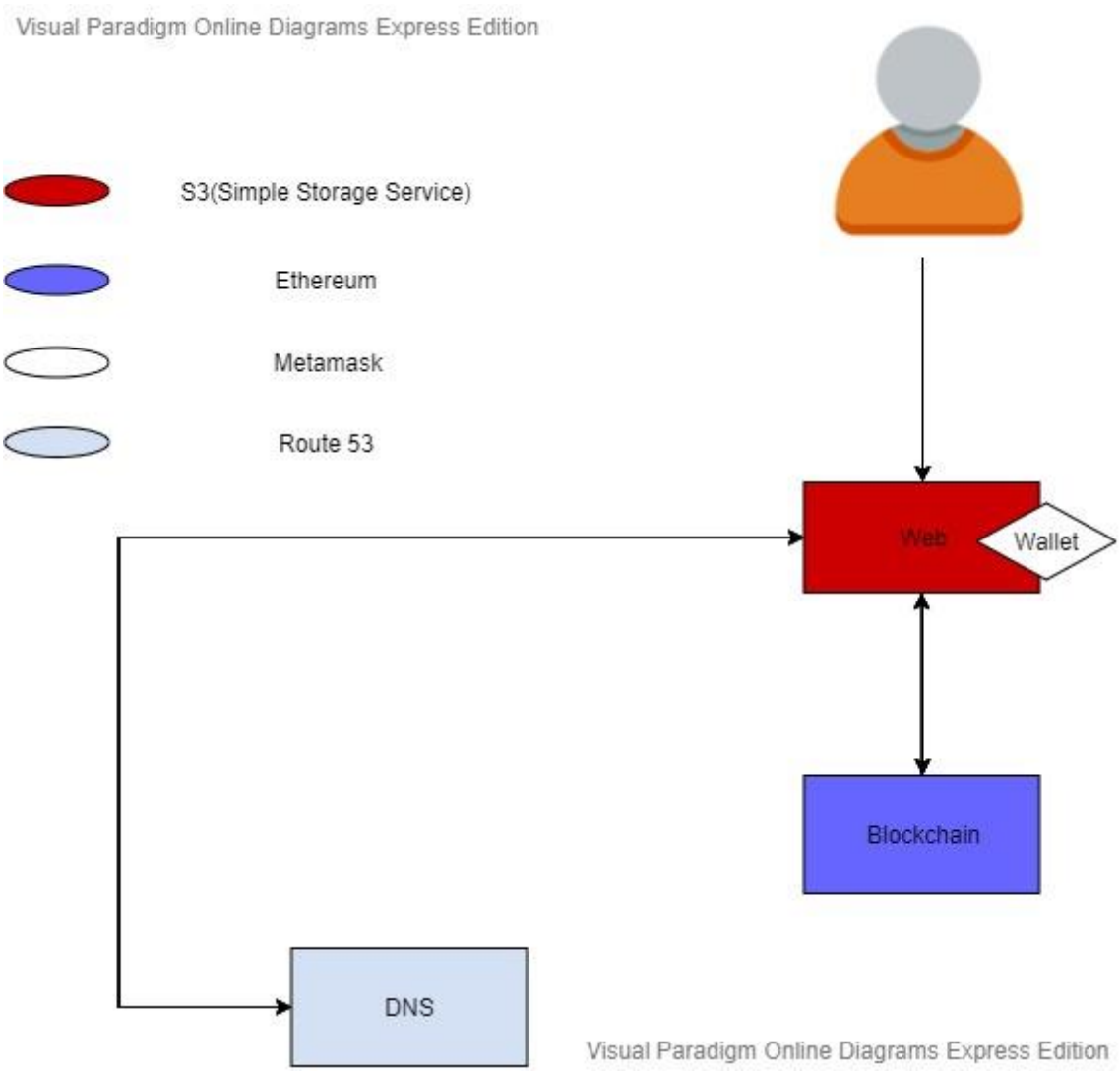
3.1.1 Arquitectura del diseño con back-end: En este diseño el servidor se encargará de gestionar las firmas de las transacciones y de almacenar la información de acceso del usuario.



En esta arquitectura, el punto de entrada para el usuario será una página web hospedada en el servicio EC2 de Amazon Web Services y cuyo registro de nombre será gestionado por el servicio

DNS Route 53. En la página web habrá un sistema de autenticación de usuarios que será cotejado por el servidor usando una base de datos MongoDB. Una vez autenticado, el usuario tendrá acceso a los servicios de la aplicación, que será atendidos por el API REST también alojado en EC2. El API se hará cargo de guardar y modificar los cambios necesarios en la base de datos y en Ethereum.

3.1.2 Arquitectura con MetaMask: Aquí, el usuario usa un wallet de navegador para gestionar las claves y firmar las transacciones desde el navegador.



La arquitectura elegida aquí es mucho más simple que la anterior, esto es debido a que ya no es necesario un sistema de autenticación ni un servidor que firme por los usuarios, gracias al uso del wallet MetaMask. En este diseño, el punto de entrada sigue siendo una página web hospedada por S3 y enrutada por Route 53.

3.2 Tecnología Utilizada

Para hacer uso de los servicios de AWS se requiere de una cuenta. Estas cuentas dispondrán de servicio gratuito con ciertos límites, en caso de sobrepasarlos tendrá un coste añadido.

S3

Amazon Simple Storage Service (conocido como Amazon S3), como el nombre bien indica, es un servicio que ofrece Amazon para hospedar objetos en "buckets" o cubos, que serán accesibles mediante una URI. También se puede usar como proveedor de páginas webs estáticas, con funcionalidad cargada en el lado del cliente mediante HTML, CSS y JS. Usar S3 como servicio de hosting en este caso, aporta costes muy bajos (gratuitos en ciertos casos), baja latencia y mantenimiento reducido [1].

El proceso es muy sencillo. Primero creamos un "bucket", que representará un contenedor donde almacenaremos los objetos de interés. Luego, subimos dichos objetos al contenedor creado. Por último, configuramos los permisos de acceso al contenedor.

EC2

Es el servicio de cómputo en la nube que ofrece Amazon web services. Tiene una gran variedad de imágenes de sistemas preconfigurados para agilizar la instalación. Aquí será donde desplegaremos nuestro servidor para la versión 1(con back-end) de la aplicación.

Route 53

Route 53 es el servicio de Amazon encargado de la traducción de IP's a los nombres de hosts asociados (Domain Name System).

Truffle Suite

Truffle Suite es un framework de desarrollo para Ethereum. Dispone de las herramientas necesarias para poder desarrollar contratos inteligentes, compilarlos y desplegarlos en cualquier red compatible con Ethereum.

Infura

Infura[27] ofrece acceso a su nodo de Ethereum, de forma pública mediante su API, permitiendo así el desarrollo de aplicaciones sin tener un nodo propio.

NodeJS

NodeJS es un entorno en tiempo de ejecución basado en el lenguaje de programación Javascript, orientado a la capa del servidor, con programación asíncrona preparado para gestionar eventos.

Express

Express es una librería diseñada para crear aplicaciones web y API. Es la herramienta que usaremos para encaminar las peticiones del cliente a un evento en concreto.

JSON Web Token

Json web token es una librería para el intercambio de información segura con formato JSON, generalmente usado para la etapa de autenticación / autorización en las aplicaciones web.

MongoDB

[27] https://infura.io/

Es una base de datos NoSQL orientada a documentos. Lo usaremos para registrar a los usuarios y enlazarlos con los documentos registrados.

Mongoose

Mongoose es la librería que usaremos para facilitar las gestiones con la base de datos MongoDB.

Web3.js

Web3.js es una colección de librerías que permite interactuar con nodos Ethereum locales o remotos mediante HTTP, IPC o WebSocket.

Ethers[28]

Es una librería con una funcionalidad similar a web3. Lo incluimos porque incluye una gestión local de "nonce" que no tiene web3 y es necesario para un correcto funcionamiento de las transacciones múltiples simultáneas, cosa que se da en una aplicación web que recibe peticiones de multitud usuarios.

MetaMask

MetaMask es una wallet de tipo navegador. Almacena y gestiona la información del usuario (claves privadas/publicas) para interactuar con el entorno blockchain.

[28] https://docs.ethers.io/v5/

4. Desarrollo de la solución propuesta

Comenzamos por escribir el contrato inteligente usando el lenguaje Solidity. Lo que debemos tener en cuenta a la hora de escribir el código es que necesitamos un registro de strings(hash) para identificar cada documento y la marca temporal quedando guardada la fecha en que se realiza dicho registro. También, deberemos disponer de un método que añada los registros y otro que sea capaz de leerlos. Se puede consultar el código del contrato en el anexo.

Una vez preparado nuestro contrato inteligente lo desplegamos en la red de Ethereum, en la red de pruebas "rinkeby" para ser más exactos, así podemos usarlos sin incurrir en gastos. En el momento de realizar el despliegue, tenemos que prestar especial atención a la dirección donde se almacena el contrato y el ABI que se genera, para así poder interactuar con el contrato desde el exterior, sea nuestro servidor o el navegador del cliente. Para ver cómo desplegar un contrato inteligente, se puede consultar la documentación de Truffle.

Ahora que tenemos el contrato inteligente escrito y desplegado en la red Ethereum, nos centramos en cómo interactuar con éste.

Modelo 1

Como idea general, decir que esta versión es una simple página web que contiene la lógica necesaria en él para interactuar con el contrato inteligente con la ayuda de MetaMask. Estará desplegado en un bucket de S3 con acceso público y tendrá el código necesario para detectar la configuración de MetaMask (la red y la cuenta en uso), firmar con las credenciales del usuario, guardar y leer registros en el blockchain. El cálculo del hash del documento se hará en el lado del cliente con una librería llamada js-sha256 [29]. Las acciones de firma y confirmación de transacciones se realizarán a través de MetaMask y serán enviadas a la cadena de bloques donde se almacena el contrato inteligente usando web3. En el siguiente párrafo realizamos una explicación detallada de lo comentado hasta ahora.

El modelo 1 cuenta con que el cliente tiene instalada la extensión MetaMask. En el momento en que carga la página web, detecta si existe esta extensión y en caso de no ser así avisa al usuario y lo redirige a la web de descarga. Una vez cargada la página y habiendo comprobado que se está usando MetaMask, se le da al usuario la opción de conectarse a Ethereum con su cuenta usando un botón explícitamente visible. El concepto de conectarse, en este caso, significa habilitar la instancia de Ethereum que previamente inyecta MetaMask. El hecho de tener que realizarlo de forma explícita es una sugerencia de buena práctica realizada por los desarrolladores de esta extensión. Esto es debido a que, de este modo, el usuario tiene el control y la certeza de no estar interactuando indebidamente con ningún blockchain en segundo plano. Hay que tener en cuenta que el cambio de, tanto de la red como de la cuenta, son detectados por EventListeners y mostrados por pantalla. En esta etapa, además, se realiza la creación de la instancia del contrato inteligente a partir de la función *web3.eth.Contract(abi,contractAddress)*,habilitando la posibilidad de interacción con los métodos definidos en este.

Después de conectar el proveedor (la instancia Ethereum que hemos mencionado antes), se activará el botón "Sign" de la sección de "Load a new file". Lo que se desarrolla en esta sección es el cálculo del hash del archivo seleccionado y la firma de éste. La función hash es ejecutada por la librería js-sha256, la cual recibe el contenido del archivo mediante un FileReader. De este

[29] https://github.com/emn178/js-sha256

modo, cualquier alteración indebida del documento se verá reflejada en el hash resultante. El paso que prosigue al cálculo del hash es la realización de la firma. Esto es activado mediante el botón de "Sign" y ejecuta la función *web3.eth.personal.sign(hash, address)*, que tiene en consideración el hash y la dirección Ethereum con la que realizaremos la firma, obteniendo como resultado una secuencia de caracteres (firma compatible con Ethereum). La llamada a esta función activa un evento de MetaMask que resulta en una ventana emergente que solicita la acción del usuario, siendo las dos opciones posibles, realizar la firma o denegarla. Aceptar la realización de la firma activa las otras dos secciones restantes.

El botón "Save" acciona el proceso que se realiza en la sección de "Store record from the blockchain". Lo que esto desencadena es una llamada al método encargado de guardar registros del contrato inteligente que hemos desplegado anteriormente. La llamada se realiza a través la función *notary.methods.writeRecord("0x"+msg,a,"0x"+b,"0x"+c).send({from:selectedAddress, gas:1000000})*, donde "notary" es la variable donde se almacena la instancia creada en la etapa de conexión y la variable "selectedAddress" es la dirección Ethereum conectada y detectada por MetaMask. La llamada a esta función activa un evento de MetaMask que muestra una ventana emergente pidiendo confirmación de la realización de esta transacción, una vez confirmada, sigue este curso: MetaMask usa la clave privada asociada a la cuenta que se está usando para crear una transacción cruda (raw transaction) y firmarla, convirtiéndolo en una transacción Ethereum válida y lista para ser propagada a la red, en este caso, a través de los nodos de Infura. El sistema de propagación funciona de la siguiente forma, la transacción es seleccionada por un nodo minero o varios e incluido en el bloque pendiente. En ese momento, comienza el algoritmo conocido como Proof of Work(PoW), donde el primer nodo minero que lo finalice correctamente consigue validar y añadir el bloque pendiente a la red Ethereum como un bloque confirmado. Al obtener la confirmación en nuestro sistema, mostramos en la pantalla el hash de la transacción, el estado de la transacción, el número del bloque, la fecha y la cuenta que se ha usado. Con el hash de la transacción podremos explorar, en servicios como etherescan.io, los detalles de esta. El estado de la transacción es un mecanismo que hemos añadido para darle retroalimentación al usuario sobre la situación en curso, al fin y al cabo, las transacciones demoran un rato en ser validadas y adheridas en la cadena de bloques. Los estados de la transacción son: Pending, Success o Failed.

Por último, en la sección de "Read record from the blockchain", se pretende consultar el blockchain para recuperar la información sobre un registro. Para tal propósito, el botón "Search" acciona la llamada a la función notary.methods.getRecord("0x"+hash).call(). La respuesta, en caso de existir un registro con ese hash, sería la firma que se guardó en la sección anterior y la fecha en la que se realizó el registro. A partir de la firma, obtenemos la cuenta origen de la transacción haciendo uso de la *función web3.eth.personal.ecRecover(msg,signature)*.

Modelo 2

Tal y como hemos hecho en el modelo anterior, comenzaremos por dar una visión general del diseño. Esta versión con back-end, se desplegará en EC2 y será diseñada en NodeJS. Con la ayuda de librerías como Express, jasonwebtoken, mongoose, web3 y ethers, gestionará la base de datos de los usuarios, la lógica para interactuar con el blockchain y también proveerá la página web. En esta versión, las transacciones serán firmadas por el servidor, no por el usuario como en el otro modelo. Los hashes de los archivos que guarden los usuarios quedarán almacenados en la base de datos, estando asociados a sus respectivos propietarios.

Ahora, después de ofrecer una perspectiva global del proyecto, daremos una explicación más detalla del desarrollo, comenzando por mostrar la estructura.

- node_modules
- src

    - blockchain
        - library.js
    - configurations
    - config.js
    - db.js
    - env.variables.json
- controllers
    - auth.controller.js
    - user.controller.js
- front
    - css
        - home.css
        - index.css
        - register.css
    - html
        - home.html
        - notary.html
        - register.html
    - img
        - avatar1.png
        - Cryptonics-Logo-4105x1000-1-500x122.png
        - diagram1.png
        - L1-Logo.png
    - js
        - app.js
        - index.js
        - register.js
    - index.html
- middlewares
    - app.auth.js
    - blockchain.js
    - user.validate.params.js
    - web.auth.js
- models
    - user.model.js
- routes
    - auth.routes.js
    - routes.js
    - user.routes.js
- utils
    - jwt.utils.js
- server.js
- .gitignore
- package.json

A continuación, nos disponemos a explicar cada uno de los componentes y cómo interactuar para crear el servidor final que buscamos.

node_modules: aquí se almacena el código de todos los módulos que se usen en el proyecto, tanto los que vienen de base como los que se especifican en el archivo package.json.

src: es un directorio general donde se almacena el grueso del proyecto.

blockchain: el propósito de este directorio es almacenar el código relacionado con la cadena de bloques. En este caso tan solo está compuesto de library.js, donde se reúne todo el código

necesario para conectar e interactuar con el contrato inteligente. Está compuesto por las siguientes funciones: *init*, *writeRecord* y *getRecord*.

La función *init* recibe 2 argumentos, la dirección del contrato inteligente y un objecto que contiene los atributos account y privKey, siendo el primero la dirección Ethereum y el segundo su correspondiente clave privada. El propósito de esta función es inicializar las instancias, tanto de ethereum, con web3 y etherjs, como del contrato inteligente, con la información de la cuenta que usaremos para realizar las firmas de las transacciones.

Pasando a la función *writeRecord*, recibe 1 argumento y es el hash del documento que queremos registrar. Del mismo modo que en el modelo 1, para almacenar el registro en Ethereum, pasamos el hash por la función *web3.eth.personal.sign(hash, address)* para obtenerlo firmado, y de nuevo, realizamos la transacción llamando a la función del contrato inteligente *contract.writeRecord(hash, r , "0x" + s,"0x" + v,{gas:1000000})*, pero esta vez a través de la instancia del contrato generado por etherjs. El motivo de esto es debido a que ethersjs dispone de un módulo que permite gestionar el nonce localmente, evitando errores.

La última función de este archivo, *getRecord*, recibe el hash como único argumento y lo que hace es recuperar el hash firmado y la marca temporal registrada y asociada al hash que se pasa como argumento. Para hacer esto, usamos la llamada a la función *contract.getRecord(hash)* de ethersjs. Con el hash firmado y el hash que recibe la función inicial, podemos ejecutar la función *web3.eth.accounts.recover(hash,signature)* y recuperar la dirección origen de la firma, de este modo, en caso de ser una dirección distinta a la del servidor, podemos avisar al usuario que este registro en concreto ha sido realizado por el sistema del modelo 1.

configurations: dentro de este directorio encontramos los elementos y variables de entorno que forman parte de la configuración inicial. Esto es, configuración del secreto y tiempo de expiración de los tokens de JWT, puertos en los que se va a conectar el servidor y la base de datos, y principalmente el objeto KEYS que debe contener los atributos account,privKey y contractAddress, donde los valores indican respectivamente, la cuenta ethereum a usar, su contraseña privada y la dirección donde está desplegado el contrato inteligente.

controllers: es el directorio donde almacenamos los controladores de autenticación y usuarios (auth.controller.js y user.controller.js). El controlador de autenticación, auth.controller.js, está formado por dos funciones, login y logout. La función login recibe, mediante el cuerpo de la petición, el usuario y la contraseña. Comprueba si existe tal usuario en la base de datos, haciendo uso de funciones de consulta de mongoose, y en el caso de existir, corrobora la contraseña facilitada con la contraseña encriptada que hay almacenada en la base de datos. Para dicha comprobación, se usa una función de la librería Bcrypt llamada compareSync, a la que le pasas la contraseña facilitada por el usuario y la contraseña almacenada en la base de datos. Si la contraseña es correcta, se genera un token con JWT y configura la cabecera Authorization del cliente para almacenar dicho token.

La función *logout* comprueba si existe la cabecera de autorización de la petición recibida en cuyo caso, coloca el token de autorización en la lista de tokens no válidos.

En el controlador user.controller.js residen las funciones que gestionan toda interacción relacionadas con el usuario: creación de una cuenta, lectura, actualización y eliminación (siguiendo las 4 funciones básicas de persistencia, CRUD) y también *writeRecord* y *getRecord*. El primer grupo de funciones mencionado es predecible, pero sólo explicaremos la función de creación(*create*) porque los otros 3, a pesar de estar implementados en el API, no se les da uso en el frontend. Lo que realiza, como era de esperar, es crear un usuario en la base de datos, con su correspondiente contraseña encriptada por Bcrypt, y configurar el nuevo token como valor de la cabecera de autorización de la respuesta. La función *writeRecord* añade en la base de datos el hash, recibido en el cuerpo de la petición, en la lista de documentos guardados por el usuario que ha desencadenado la llamada a esta función. La función *getRecord* comprueba que exista el hash,

recibido en el cuerpo de la petición, en la lista de documentos del usuario en concreto. En caso de existir, pasa el control al siguiente middleware. En caso contrario, devuelve un mensaje de error.

front: es el directorio donde almacenamos todos los archivos que se van a exponer públicamente en la web. Está formado por la típica estructura css-html-img-js y un index.html actuando como ventana de entrada para el usuario. En el punto de entrada, se ofrece el habitual sistema de login con usuario y contraseña, además de un enlace que abre la ventana de registro(register.html). Una vez logeado o registrado, se redirige al usuario a la ventana de inicio(home.html). En esta ventana se da una breve presentación sobre el proyecto e indicaciones de cómo usarlo. En el menú de esta ventana hay un botón para cambiar a la ventana de la aplicación principal(notary.html). En esta ventana es donde el usuario carga el archivo, calcula el hash, guarda el registro en el blockchain y lo lee. Los botones de cada una de las ventanas mencionadas activan las llamadas a las funciones que se encuentran en los archivos js con nombres homólogos a los nombres de las ventanas. Estas funciones a su vez realizan las peticiones a nuestro API. Por supuesto, el css de cada ventana y las imágenes son proporcionados por los archivos dentro de los directorios css e img.

middlewares: los middlewares son funciones intermedias que se ejecutan para cosas como: validar la entrada del usuario, proteger rutas verificando la existencia de tokens de autorización o como ocurre en nuestro caso, realizar llamadas intermedias a la librería que conecta con el blockchain.

models: compuesto por user.model.js define la estructura que va a tener el esquema usuario que almacenaremos en la base de datos mediante mongoose. Los atributos que hemos decidido otorgarle son: username, password, hashedDocuments, creationDate, lastUpdatedDate. La información que almacenan dichos atributos es evidente, pero si alguno es digno de mención ese es hashedDocuments, el cual es una lista de strings, que representan el hash de cada documento almacenado por este usuario.

routes: en este directorio, el archivo principal es routes.js, que importa los otros dos, controlando las rutas disponibles en el API del servidor y las funciones que éstas desencadenan. Las rutas definidas son:

POST - “auth/login”, pasando la petición a la función *login* de auth.controller

POST - “auth/logout”, pasando la petición a la función *logout* de auth.controller

POST - “user/create”, comprueba que la entrada del usuario es adecuada mediante el middleware user.validate.params. Sólo en caso de ser correcto, crea el usuario en la base de datos con los parámetros facilitados en el cuerpo de la petición.

POST - “user/writeRecord”, después de comprobar que el usuario dispone de autorización para realizar esta petición (middleware web.auth), se realiza una llamada a la función de *writeRecord* del middleware blockchain para añadir un registro en la cadena de bloques. Si esto no da error, se almacena el registro en la base de datos haciendo uso de la función *writeRecord* del controlador user.controller.

GET - “user/getRecord/:hash”, del mismo modo que en el caso anterior, primero se comprueba que el usuario tiene permiso para usar esta ruta. Luego se consulta en la base de datos si el registro a recuperar ha sido registrado por este usuario, en cuyo caso, se pasaría a realizar la consulta en la cadena de bloques usando la función consultora del middleware de blockchain.

También existen 3 rutas más para leer, actualizar y eliminar usuarios de la base de datos. Pero como no se usa en el frontend, además de ser fácil de intuir de lo que se trata, vamos a obviar su explicación.

utils: en este directorio encontramos el archivo jwt.utils que se encarga de gestionar los tokens, la creación, validación y eliminación (lista de tokens no válidos) de estas.

server.js: es el archivo principal que junta todos los componentes mencionados hasta ahora para dar forma al servidor que buscábamos.

.gitignore: es un archivo que configuramos para que en el momento de subir el proyecto a gitHub, se ignoren los elementos que indiquemos.

package.json: aquí tenemos información como nombre, versión, scripts y otras cosas, pero, la parte importante es la lista de dependencias. Esta lista automatiza la instalación de módulos de nodeJS en el proyecto.

Habiendo visto esto, finalizamos el desarrollo del modelo 2 y continuamos con un análisis de ventajas y desventajas de cada modelo.

Ventajas y desventajas de los dos modelos

Modelo 1

Una de las principales ventajas de este modelo es la simplicidad en su desarrollo y despliegue. Tanto MetaMask como el código JavaScript requerido es ejecutado en el propio navegador del cliente, por este motivo, lo único que se necesita es un servidor que exponga la página web en internet. Esto resulta en un coste económico mínimo, al fin y al cabo, la tarifa de la transacción es descontada de la cuenta que la realiza de forma automática al aceptar la realización de la transacción. Además de esto, otra ventaja es que MetaMask gestiona el valor nonce de cada una de las cuentas que gestiona, lo que evita problemas como la pérdida de transacciones por tener un nonce erróneo. La gestión del valor del nonce en arquitecturas similares al modelo 2 puede acarrear problemas que desencadenen la pérdida de las transacciones. Una desventaja de este modelo es la monetización del sistema, siendo sus opciones la publicidad de pago por click, venta de espacios publicitarios, marketing de afiliados y similares. Esto es algo negativo porque estos modelos de monetización dependen mucho del tráfico generado en la web, y teniendo en cuenta el propósito del sistema, un alto tráfico en esta web es algo poco probable que ocurra.

Modelo 2

Como se ha mencionado en el párrafo anterior, la desventaja de este modelo es la necesidad de gestionar los valores del nonce apropiadamente. En el momento en el que se realicen varias transacciones concurrentemente o de forma inmediatamente consecutiva, todas ellas tendrán configurado el mismo valor nonce. Esto resultará en un error que descartará todas las transacciones menos la primera que se acepte. Esto se debe a que el valor del nonce para cada transacción es recibido en base a las transacciones aceptadas, sin tener en cuenta las pendientes o canceladas en ese momento. Una ventaja que tiene este modelo frente al otro es que el usuario final no necesita tener conocimientos sobre blockchain ni disponer de extensiones como MetaMask. Esto es gracias a que la gestión de toda interacción con el blockchain es realizada por el servidor. También es interesante mencionar, que a pesar de que el coste de las transacciones recaiga en el servidor (considerando que el propietario del servidor es el mismo que el dueño de la cuenta Ethereum que usa el servidor para realizar las transacciones) este gasto se puede recuperar mediante suscripción de pago al sistema (o un sistema fijo de pago por registro realizado).

Como detalle interesante, las dos versiones interactuarán con el mismo contrato inteligente, por lo que los registros guardados por un modelo serán accesibles mediante el otro. Hay que tener en cuenta que, si se intenta recuperar un registro creado por el modelo 2 desde el modelo 1, sabremos que existe dicho registro, pero no conoceremos el propietario. En el caso opuesto, podremos ver el propietario sin problema, más bien, la dirección Ethereum que ha realizado el registro.

5. Implantación

En este apartado vamos a explicar cómo se ha desplegado el sistema en los dos modelos. A pesar de plantear el registro del dominio en el apartado de la arquitectura, hemos optado por no hacerlo debido al coste económico que acarrea.

Modelo 1

Empezamos creando un bucket. Para esto, vamos a la consola de Amazon S3 y seleccionamos la opción para crear un bucket nuevo. Elegimos nombre, seleccionamos la región y lo configuramos para el alojamiento web, yendo a las propiedades del bucket y activando la opción de "usar este bucket para alojar un sitio web". Una vez realizado esto, subimos el archivo html y js. Para terminar, configuramos los permisos del bucket para ser accesible públicamente. En la ventana de propiedades del bucket, concretamente en la opción de static web hosting, podremos encontrar el endpoint(url) que nos dará acceso a la web desde el navegador.

Modelo 2

El primer paso es lanzar una instancia en Amazon Elastic Compute Cloud (EC2). En la consola de Amazon EC2, en el panel principal, seleccionamos "Launch Instance" y elegimos una imagen de máquina de Amazon (AMI) del tier gratuito, t2.micro. En las etapas de configuración, modificamos el grupo de seguridad para aceptar la salida y entrada en los puertos que vamos a usar. En el último paso de la configuración, seleccionamos el par de claves que requeriremos posteriormente para conectarnos y con esto terminamos, e iniciamos la instancia.

Una vez iniciada la instancia, nos conectamos mediante SSH[30] haciendo uso de la clave privada que habremos guardado del paso anterior. Para continuar, nos aseguraremos de que tenemos todas las herramientas necesarias en la instancia y subimos a la nube nuestro servidor mediante SCP[31]. Para finalizar, estando conectado en nuestra instancia, iniciamos el servidor con la herramienta PM2[32] para mantenerlo activo. El endpoint de acceso lo podemos ver en las propiedades de la instancia, sea la dirección IP pública o el DNS generado por Amazon.

[30] https://man.openbsd.org/ssh.1
[31] https://man.openbsd.org/scp.1
[32] https://pm2.keymetrics.io/

6. Pruebas

En este apartado vamos a realizar pruebas del funcionamiento de los dos modelos, además de comprobar posibles fallos y mostrar sus correcciones. Comenzamos con el modelo 1.

Modelo 1

Punto de acceso: http://notary2.s3-website.eu-west-3.amazonaws.com

Comenzamos revisando la conexión con MetaMask (inicialización de la instancia Ethereum)

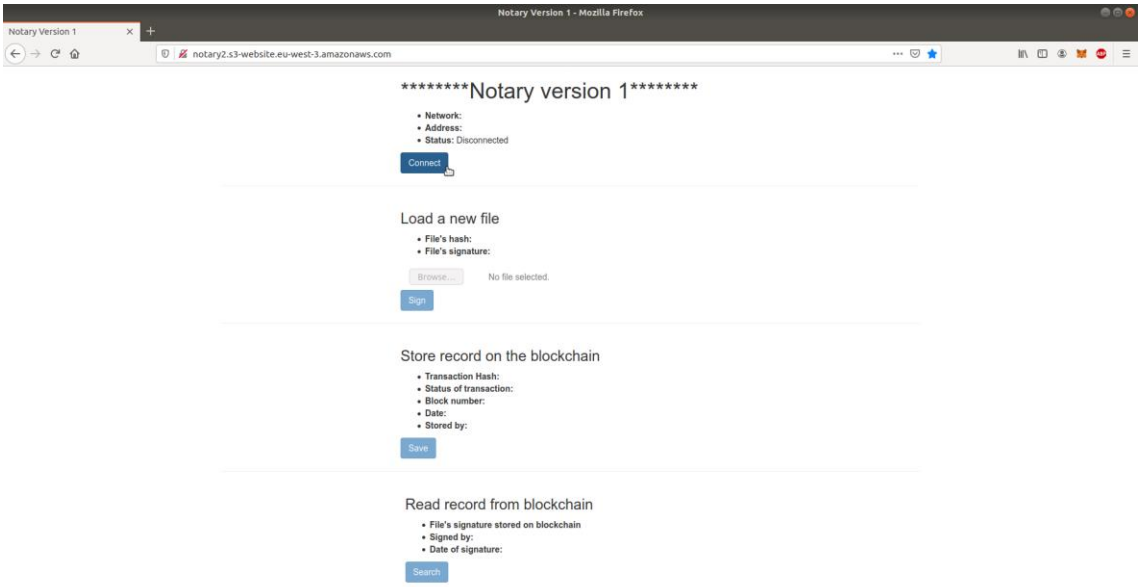


Ilustración 1- Iniciar la conexión con MetaMask

Como se puede ver en la ilustración 1, para poder iniciar la conexión con MetaMask debemos pulsar el botón Connect. Se recuerda que el concepto de iniciar conexión significa detectar la extensión de forma apropiada e inicializar la instancia de Ethereum.

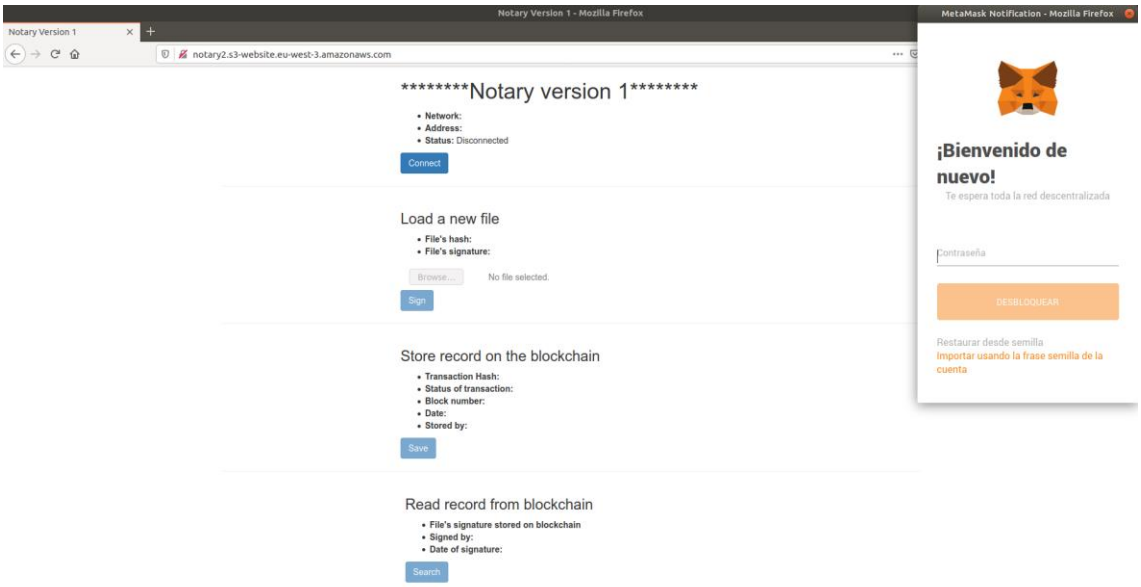


Ilustración 2 - Ventana emergente

Como se puede ver en la ilustración 2, al pulsar el botón de Connect, MetaMask nos muestra una ventana emergente donde introducir la contraseña para acceder a nuestras cuentas. Una vez que introducimos las credenciales se muestra la información de la conexión (red, cuenta, estado). Tal y como podemos ver en la siguiente ilustración.

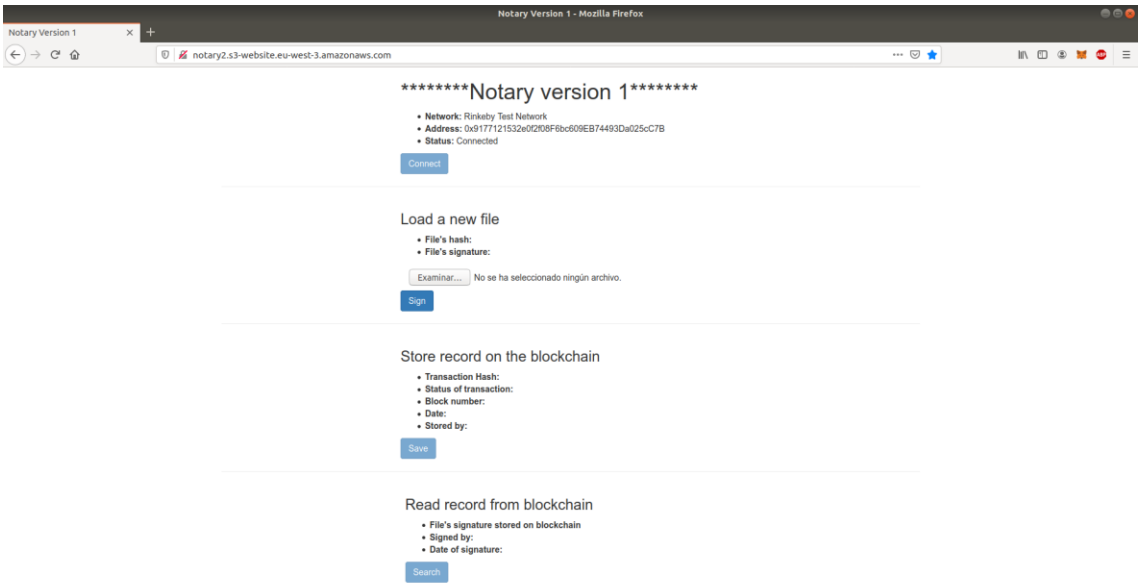


Ilustración 3 - Conexión confirmada en MetaMask

Visto esto, podemos decir que la acción de iniciar conexión con MetaMask funciona de forma satisfactoria.

Continuamos con la carga de un documento nuevo y la acción de firmar el hash de dicho documento.

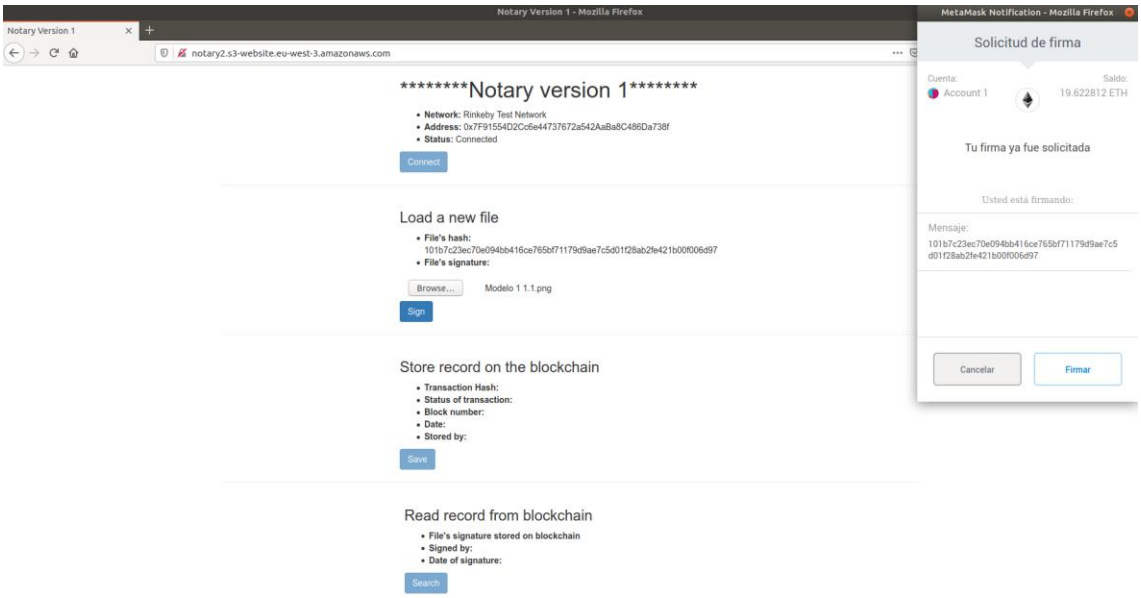


Ilustración 4 - Carga del archivo y firma

En la ilustración 4 se ve la ventana emergente de confirmación para firma que aparece tras haber seleccionado un archivo y pulsado el botón de Sign. Como resultado tras aceptar la confirmación, obtenemos el hash firmado por nuestra cuenta bajo la línea File's signature (ilustración 5)

Ilustración 5- Resultado tras la confirmación de firma

Como se muestra en las ilustraciones 4 y 5, la carga de un archivo (cálculo de su hash) y firma del hash del archivo, funciona correctamente.

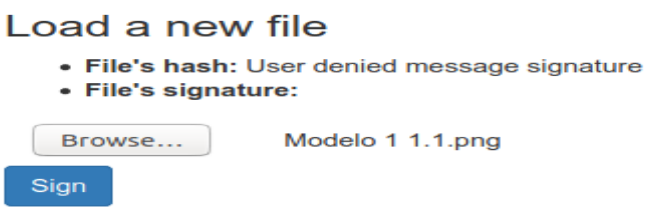


Ilustración 6- El usuario rechaza la confirmación de firma

Avisar al usuario en caso de rechazar la firma funciona adecuadamente.

Ahora veremos cómo interactuar con la aplicación para registrar el hash del documento en Ethereum.

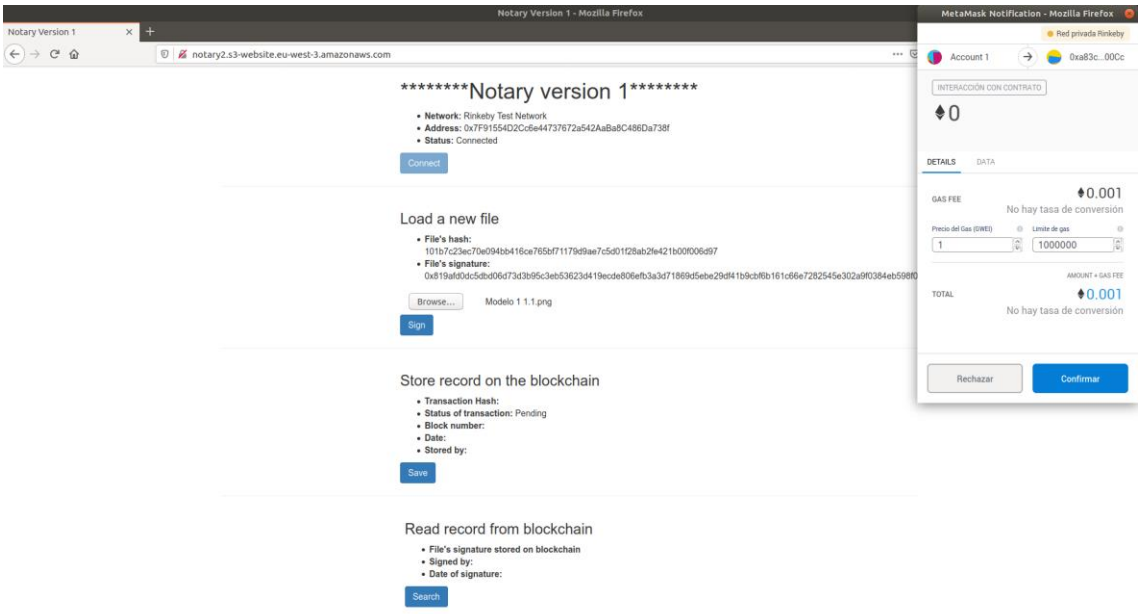


Ilustración 7 - Realizar un registro de hash

Como se aprecia en la ilustración 7, al presionar el botón Save nos aparece una ventana para confirmar la transacción. Se puede ver que MetaMask también nos muestra el coste de realizar dicha transacción, el cual corre a cargo del usuario de la cuenta. Tras confirmar la realización de la transacción, el estado se coloca en pending hasta que recibe una confirmación del blockchain o un error.

Ilustración 8- Transacción confirmada

En la ilustración 8 vemos la información que recuperamos tras confirmarse la transacción en la red Ethereum. Por esto, podemos decir que almacenar el registro en el blockchain funciona adecuadamente.

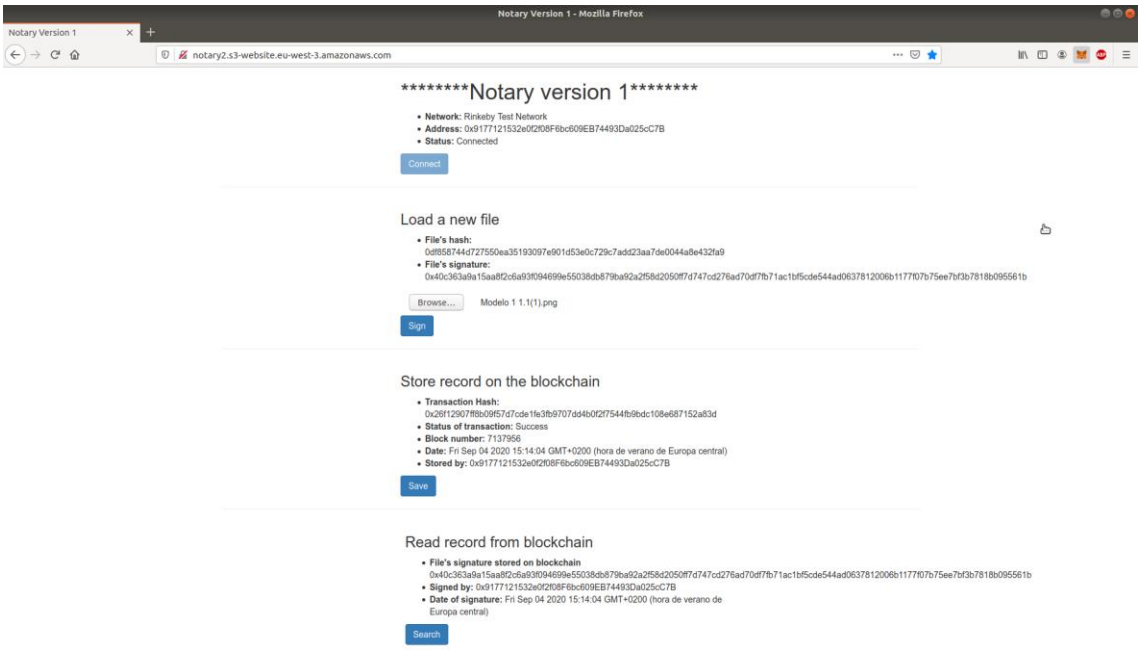


Ilustración 9 - Lectura de un registro del blockchain

En la ilustración 9 se manifiesta el resultado tras leer un registro del blockchain que recientemente habíamos almacenado. Visto esto, confirmamos que la lectura de un registro existente en el blockchain se lee apropiadamente.

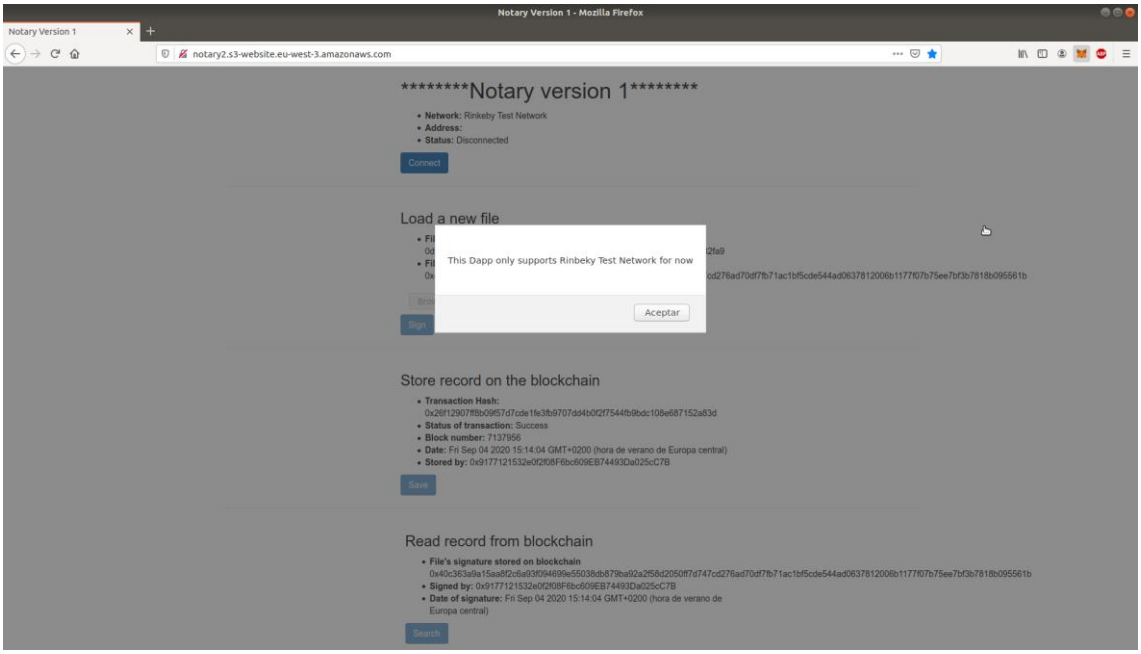


Ilustración 10- Cambio de la red Ethereum

El sistema está preparado para detectar un cambio de red en MetaMask. Del mismo modo lo está para reaccionar al cambio de cuentas, ilustración 11.

Ilustración 11- Reacción al cambio de cuenta

En la ilustración 11 se ve cómo el valor de Address ha sido modificado, pasando de la cuenta que antes acababa en 38f a la actual.

Con las próximas ilustraciones veremos cómo reacciona el sistema cuando intentamos crear un registro ya existente o cuando leemos un registro que no existe previamente.

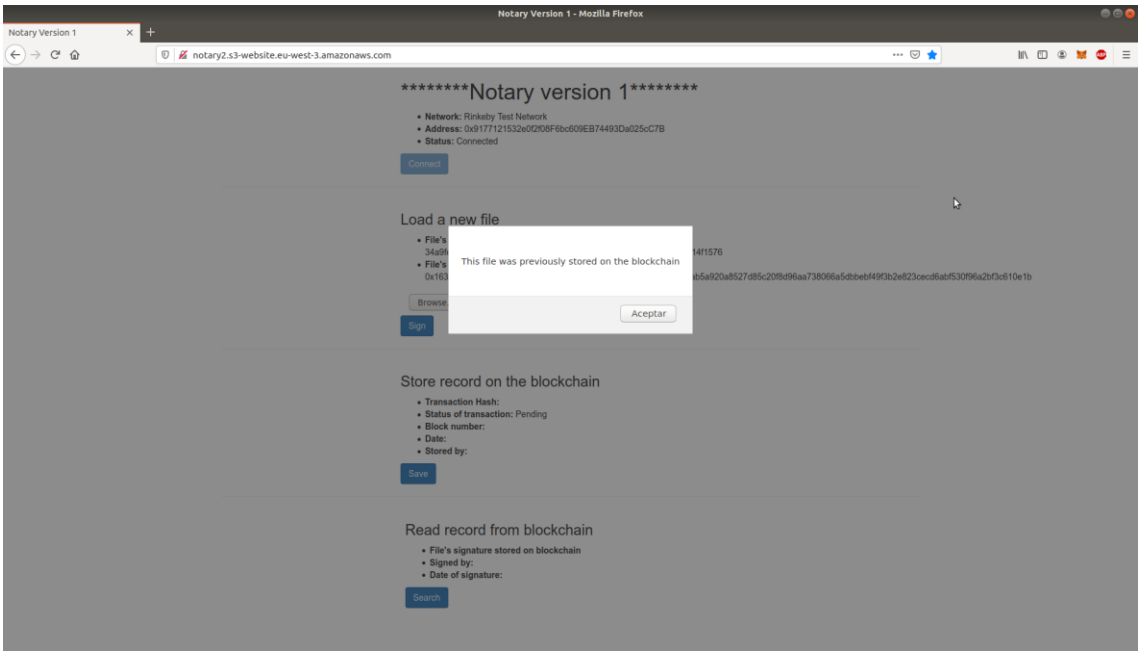


Ilustración 12- Mensaje de error tras realizar un registro que ya existía con anterioridad

Tal y como podemos ver en la ilustración 12, el sistema está listo para evitar registrar el mismo archivo varias veces.

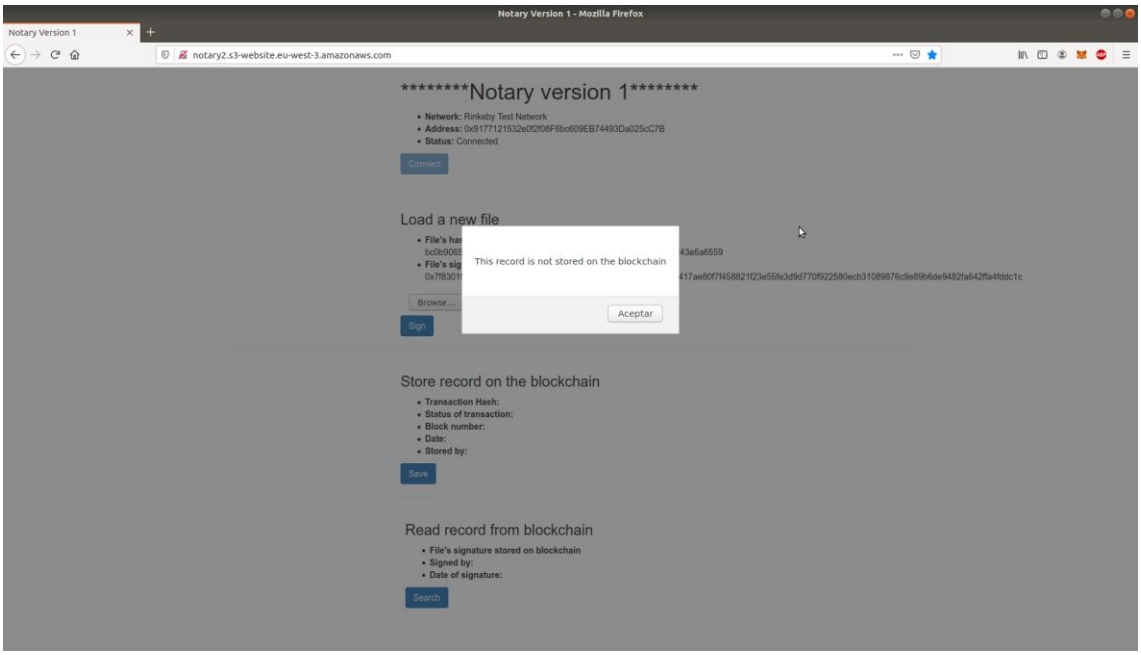


Ilustración 13- Error tras intento de lectura de un registro inexistente

En la ilustración 13 se muestra que el diseño está preparado para avisar cuando se intenta leer un registro que no existe en la cadena de bloques. Con esto acabamos las pruebas del modelo 1.

Modelo 2

Punto de acceso: http://15.236.209.190

Ventana de login

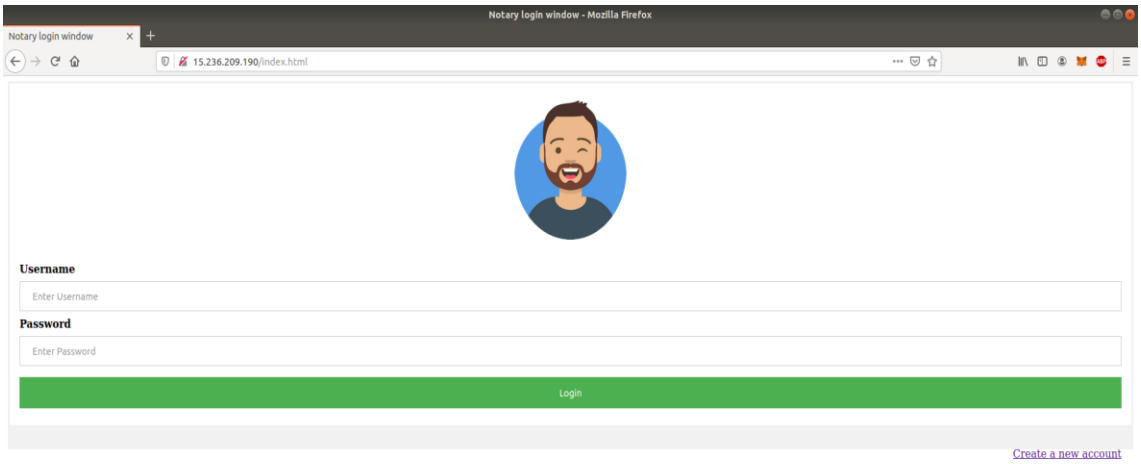


Ilustración 14- Ventana de acceso

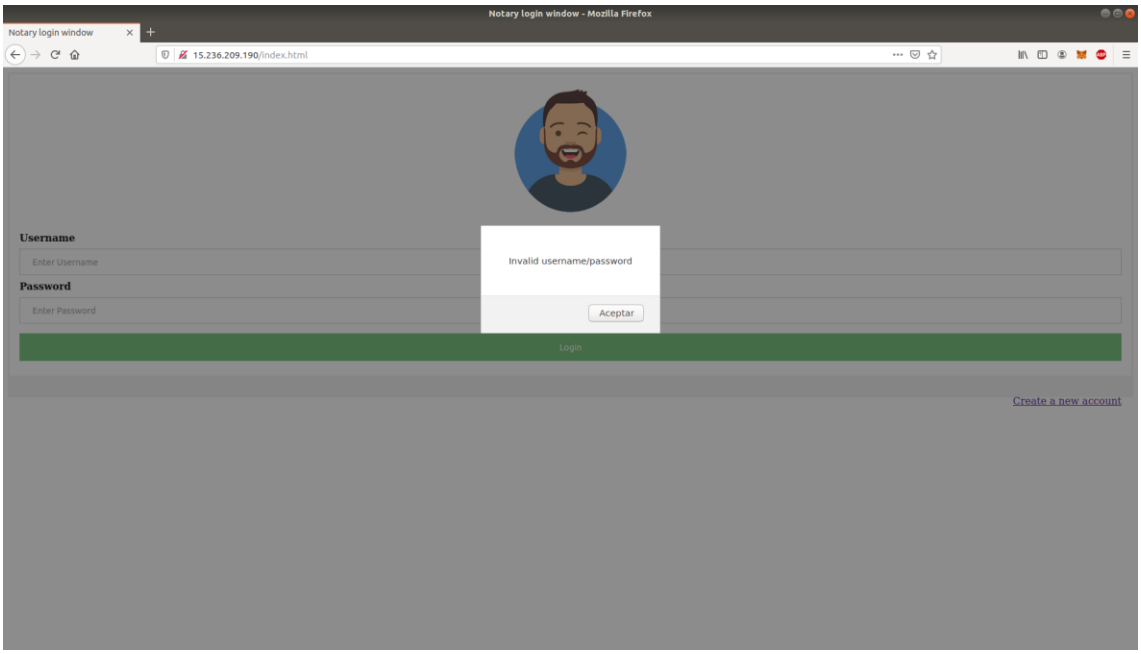


Ilustración 15: En la ventana de login, el sistema está preparado para avisar en caso de credenciales erróneos, sea porque estén vacíos o porque el usuario/contraseña no son correctos.

En caso de introducir credenciales existentes, el cliente es redirigido a la página de inicio de la aplicación (ilustraciones 15 y 16).

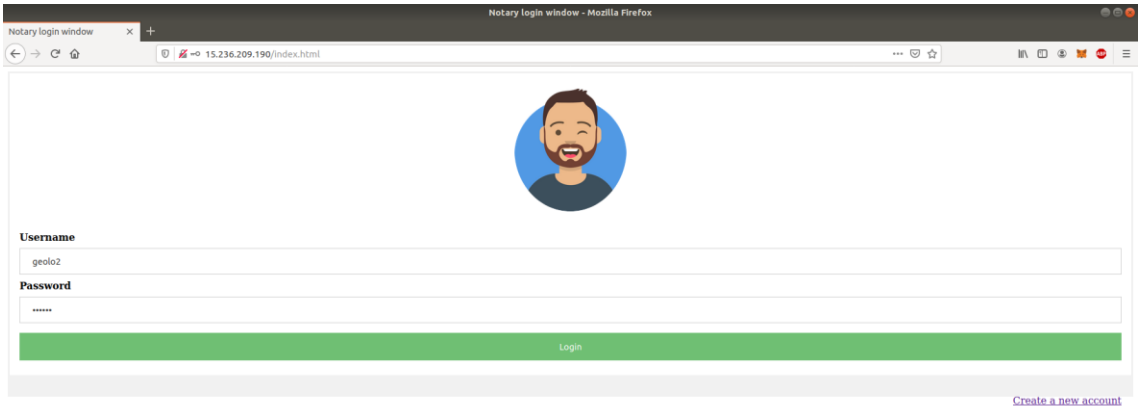


Ilustración 15 - Introduciendo credenciales existentes

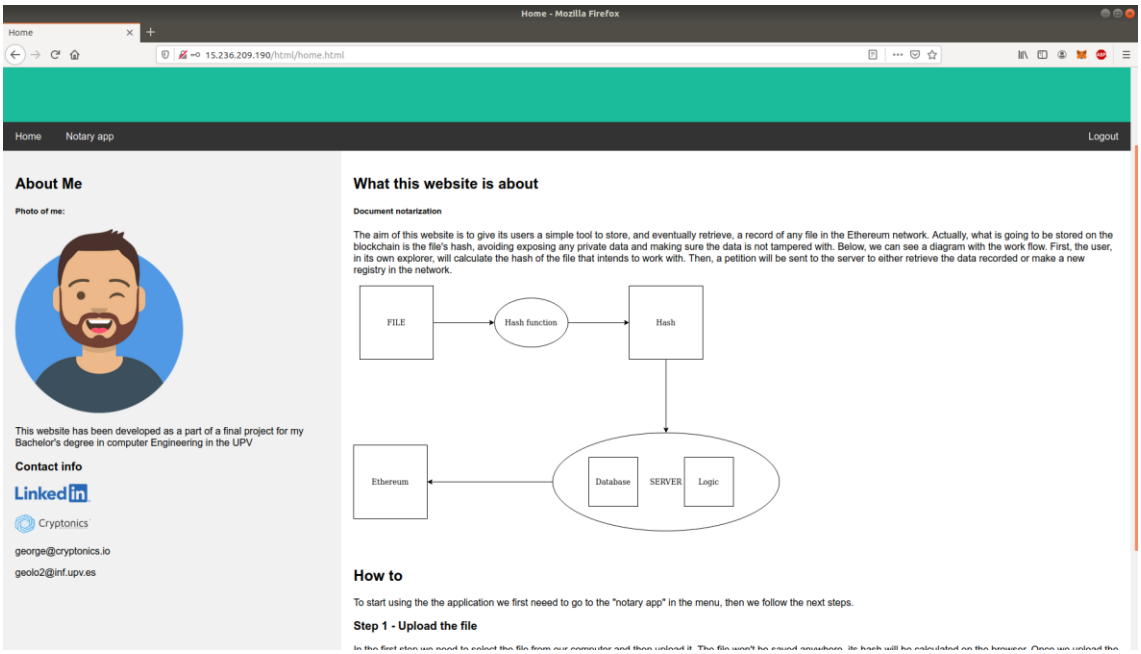


Ilustración 16 - Resultado tras introducir las credenciales correctas

Introduciendo las credenciales de acceso de un usuario registrado nos topamos con la ventana introductoria de la aplicación. Aquí se puede ver una breve explicación del proyecto y las indicaciones de cómo usarlo.

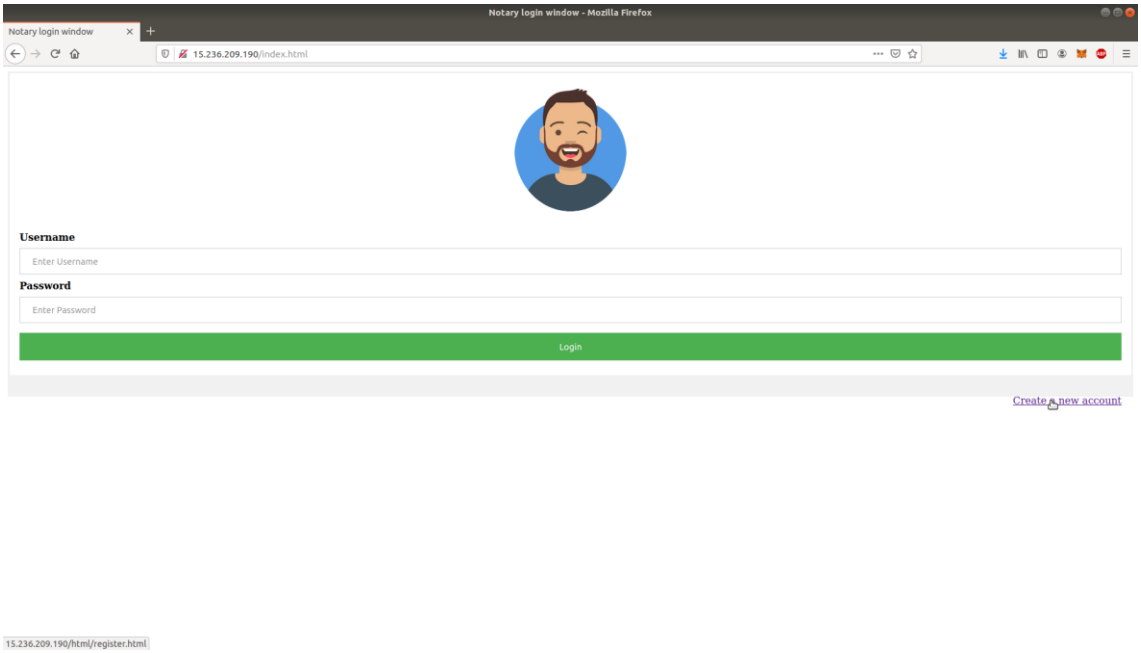


Ilustración 17- Pulsando el enlace para crear una cuenta nueva

Todavía en la ventana de login, vemos en la ilustración 18 qué ocurre cuando pulsamos el enlace para crear una cuenta nueva.

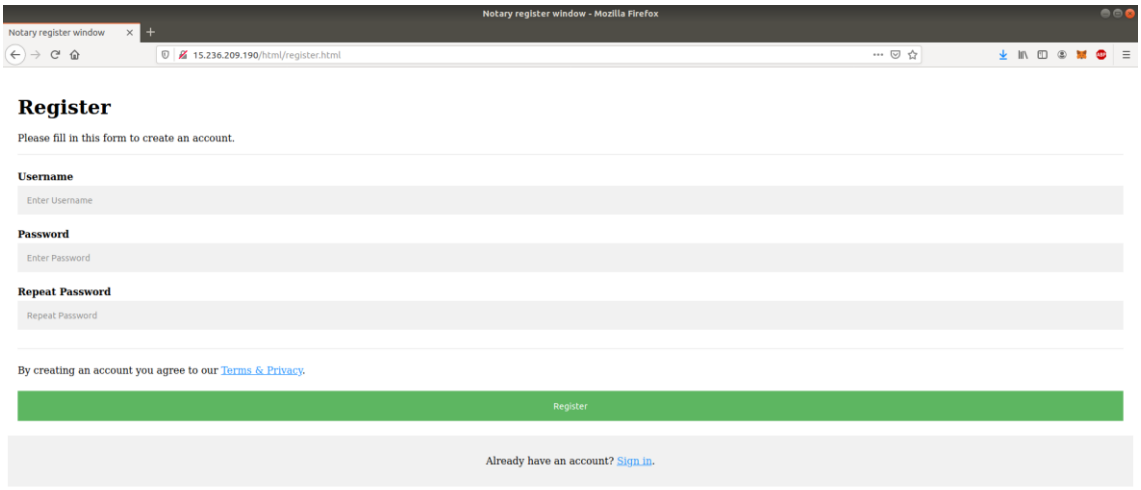


Ilustración 18- Ventana de registro

Como era de esperar, el enlace para crear una nueva cuenta envía a la ventana de registro

Ventana de registro

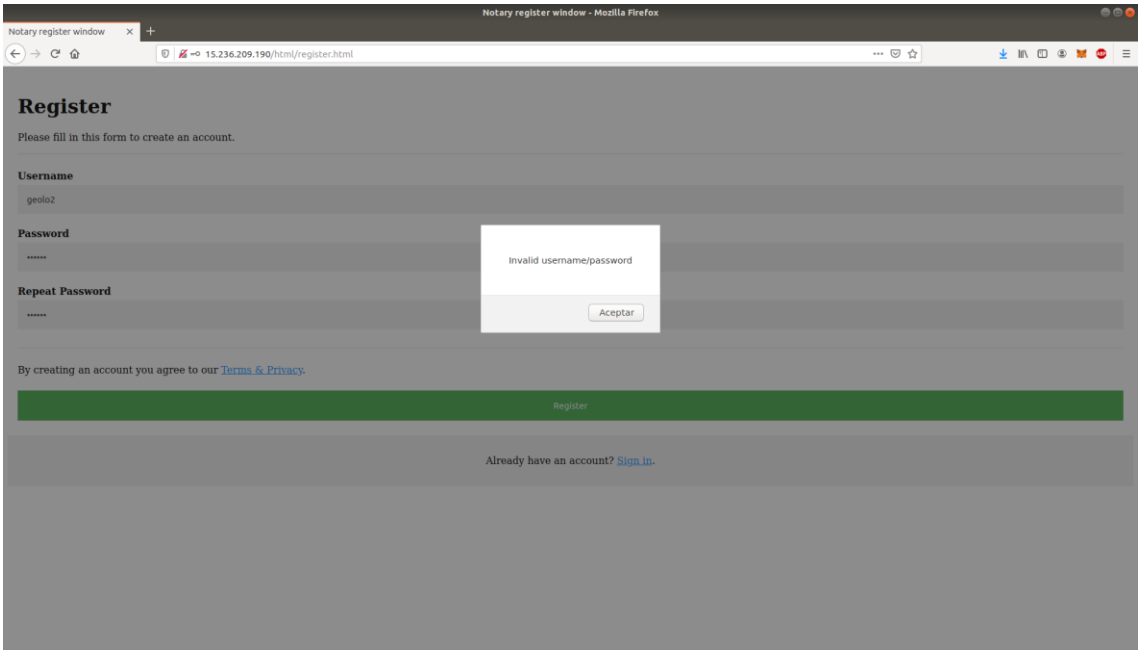


Ilustración 19- Mensaje de error, usuario existente o campos vacíos

El mensaje de error mostrado en la ilustración 19 es provocado cuando el usuario que se intenta registrar ya existe, o hay algún campo vacío.

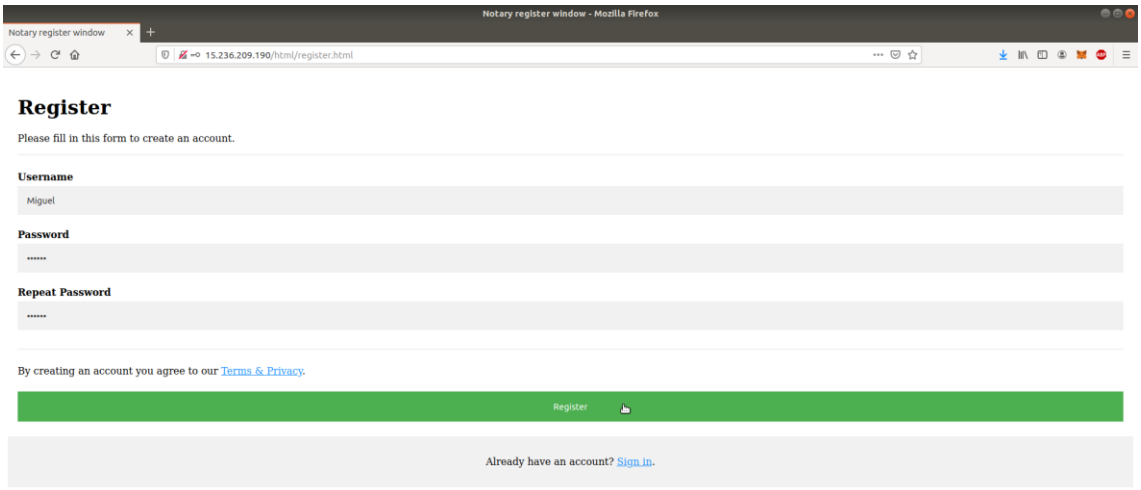


Ilustración 20- Registrando un usuario nuevo válido

Al registrar un nuevo usuario con credenciales válidas se le manda a la ventana principal de la aplicación, tal y como se ve en la siguiente imagen.

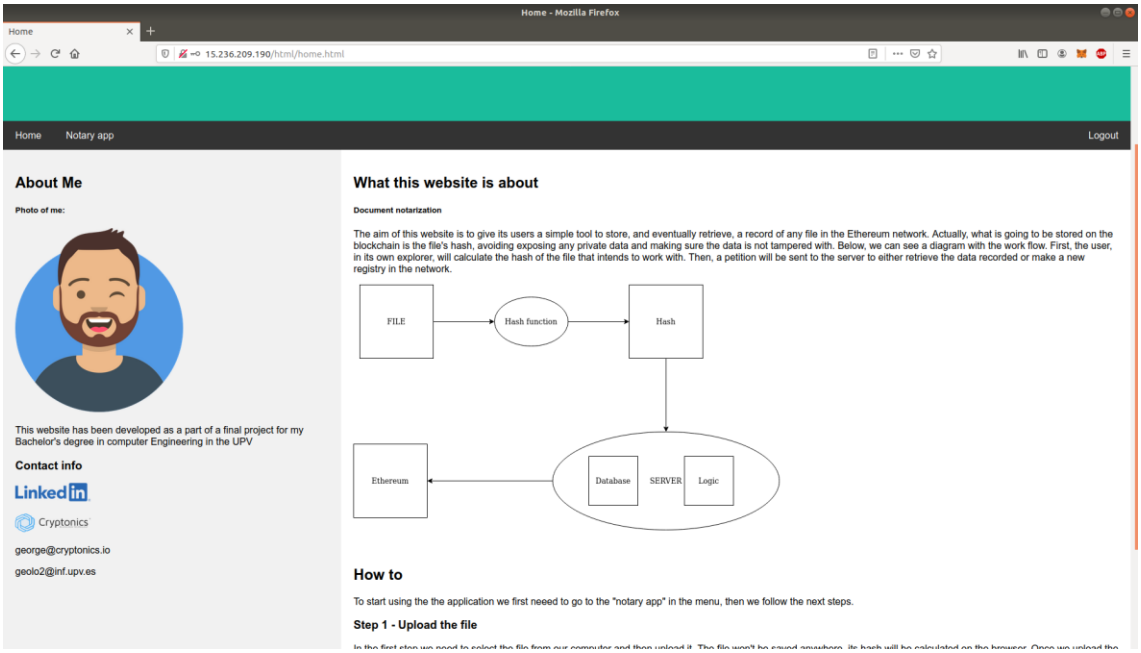


Ilustración 21- Ventana principal tras un nuevo registro

Para acabar con las pruebas de la ventana de registro, al dar al enlace de sign in, nos envía a la ventana de login (ilustración 22).

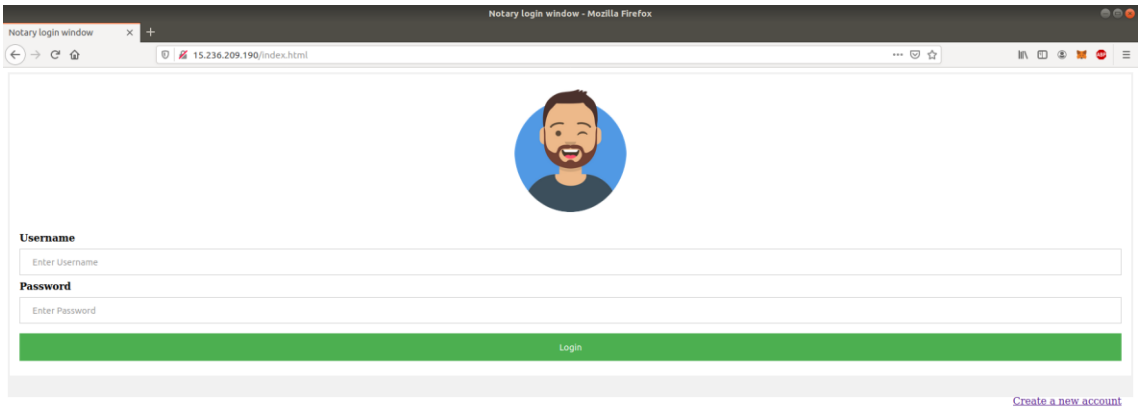


Ilustración 22- Ventana destino tras pulsar sign in (ventana de registro)

Con esto finalizamos las pruebas de la ventana de registro.

Ventana principal

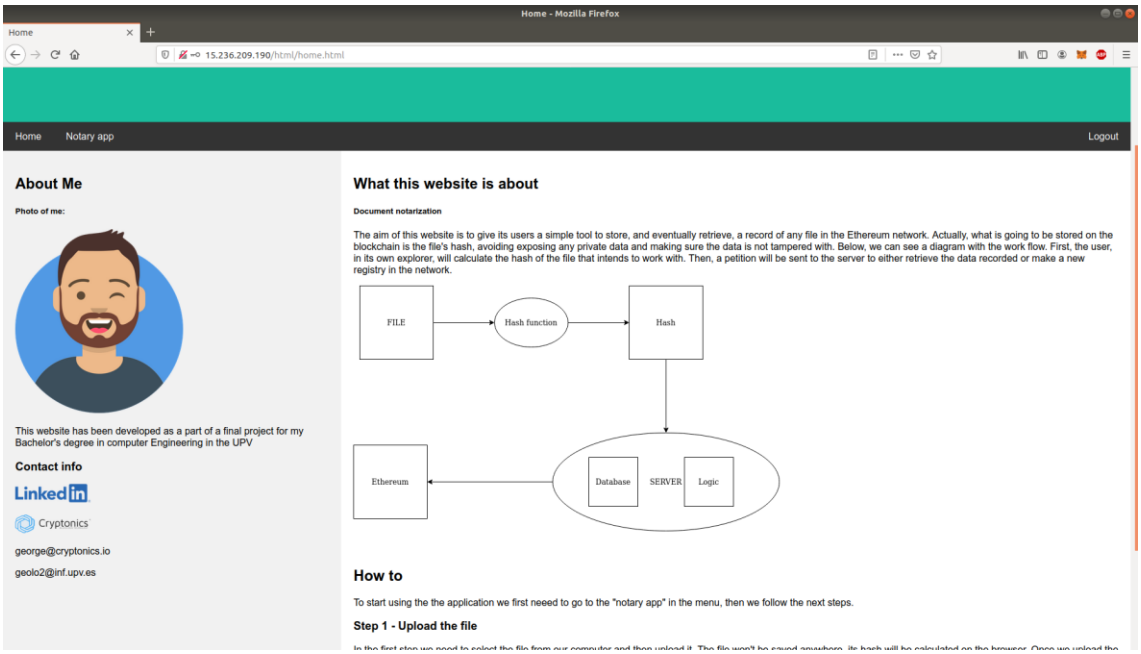


Ilustración 23 Ventana principal

La ventana principal no tiene mucha funcionalidad que probar (ilustración 23). La mayor parte es texto explicando el proyecto. Tan solo tiene un menú con algunos botones para navegar a distintas ventanas: Home (ventana principal), Notary app (dirige a la página de la aplicación Notary) y logout (este botón cierra la sesión y manda a la ventana de login).

Ventana Notary APP

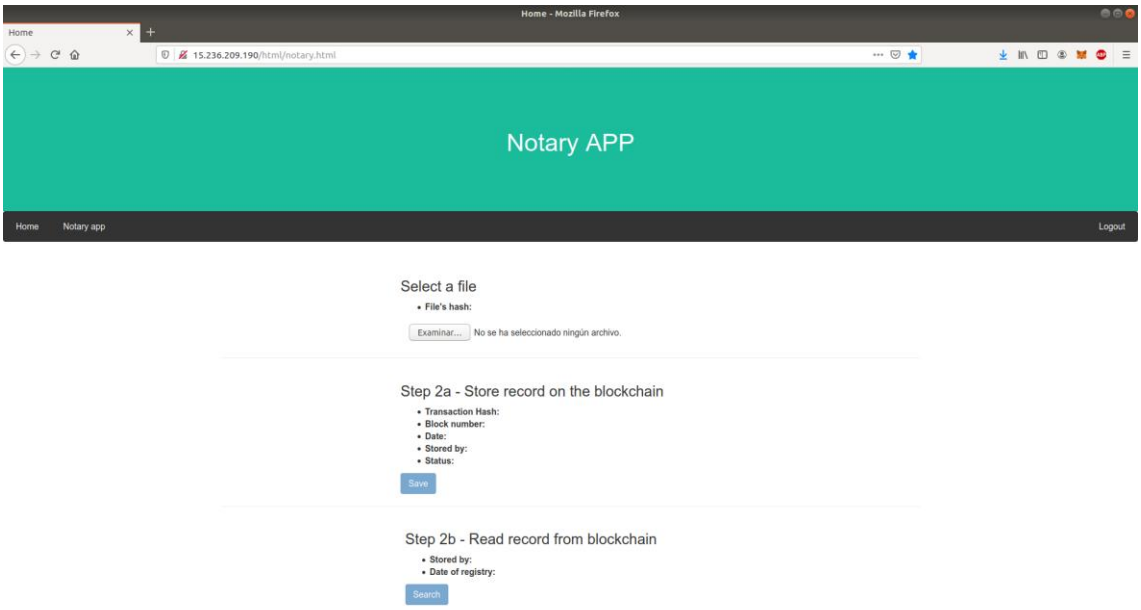


Ilustración 24- Ventana de la aplicación Notary

En esta ventana, los botones del menú tienen la misma funcionalidad que los de la ventana principal (ilustración 24). Cada vez que se selecciona un archivo, el hash se calcula, se muestra automáticamente y se activan los botones de save y search. El botón de save, cuando usamos un documento que no está registrado ya en el blockchain, crea el registro en Ethereum y muestra la información resultante en la ventana (esto suele tardar entre 10 y 20 segundos). La información resultante de un registro se puede ver en la ilustración 25.

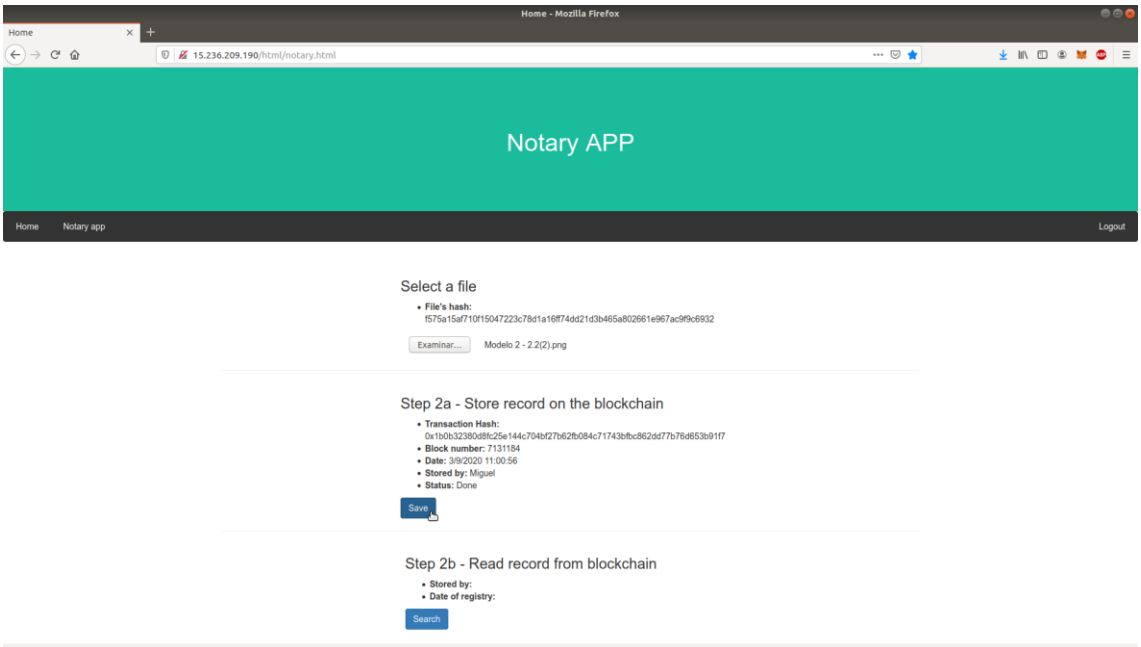


Ilustración 25 Resultado de un registro almacenado satisfactoriamente

En caso de que intentemos registrar un documento que ya existía previamente en la cadena de bloques, el sistema retorna un mensaje de error avisando de este hecho (ilustración 26).

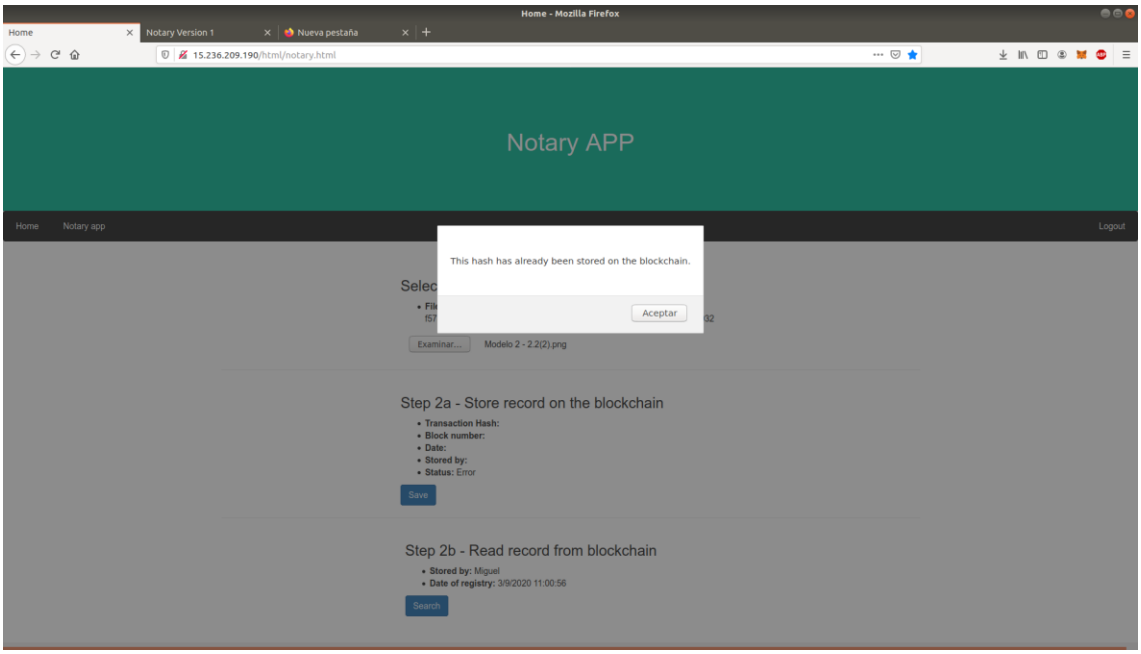
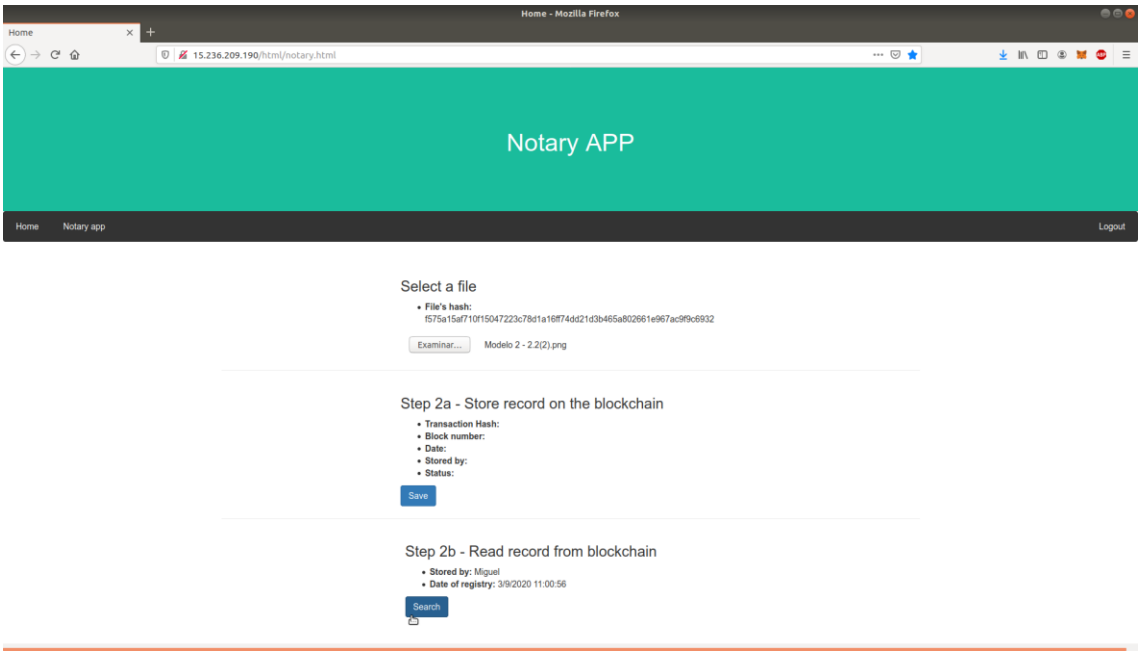


Ilustración 26- Mensaje de error. El registro ya existía previamente.

La recuperación de un registro almacenado en el blockchain se realiza de forma correcta como se puede ver en la ilustración de abajo.



Como bien se puede ver, devuelve la fecha en la que se realizó el registro en el sistema y qué usuario lo hizo.

Por último, en la próxima ilustración mostramos un mensaje de error del sistema devuelto al intentar leer un registro que no existe en Ethereum.

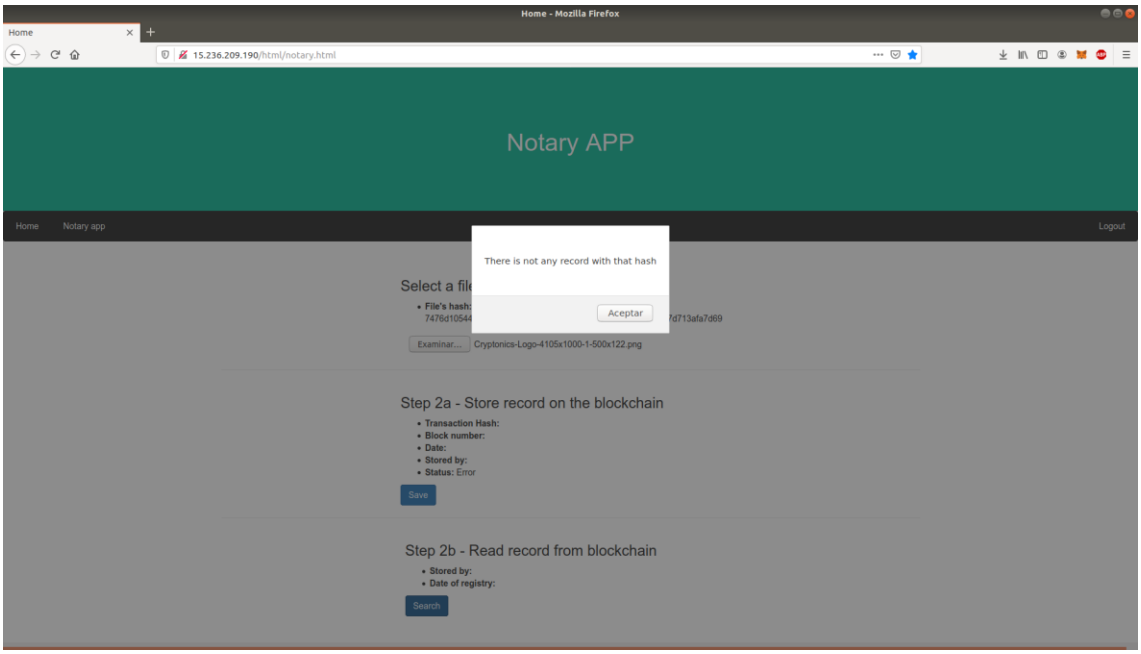


Ilustración 27- Mensaje de error generado por leer un registro que no existe

Con esto finalizamos las pruebas de funcionamiento de los dos modelos. Como hemos podido ver, tanto en el modelo 1 como en el modelo 2, se cumplen con los requisitos para dar el servicio que buscábamos poder ofrecer.

7. Trabajos futuros

Por motivos de tiempo disponible se han quedado por terminar ciertos aspectos del sistema:

- Añadir certificados auto firmados para ofrecer un servicio web encriptado y seguro.
- Usar DID para identificar al usuario en lugar del sistema de login tradicional

En cuanto a los certificados, una herramienta extendida para casos de prueba, como el nuestro, es Let's Encrypt, que permite generar certificados x.509 para ofrecer cifrado de seguridad a nivel de transporte (https). Con el uso de esta herramienta y el módulo https de Express, podríamos desplegar una web https en lugar de http.

Antes de explicar cómo usar un identificador descentralizado (DID) para el sistema de login, es necesario dar una breve introducción sobre estos. La W3C diseña una especificación para solventar el problema que genera la gestión centralizada de usuarios en internet.

Actualmente, las compañías almacenan la información en sus propias bases de datos, convirtiéndose en silos difícilmente accesibles. Si tenemos en cuenta, que cada empresa que dé servicio por internet tiene su base de datos propia, esto deriva en una cantidad absurda de credenciales que debe guardar o recordar el usuario. A no ser que se usen las mismas credenciales para todos los servicios usados, cosa poco aconsejable por motivos de seguridad.

Para solucionar este problema se proponen las identidades descentralizadas, haciendo uso de tecnologías descentralizadas (blockchain, libro mayor distribuido, bases de datos distribuidas, redes p2p), se consigue una forma de generar identificadores bajo el control del propio usuario (persona, entidad, máquina).

Mediante técnicas criptográficas se puede demostrar el propietario de cada identificador, lo que nos da la capacidad de usar los DID para iniciar sesión en nuestro sistema sin usar credenciales del tipo usuario/contraseña. Pero, esta tarea queda propuesta para una posible implementación futura.

8. Conclusiones

A lo largo de este proyecto hemos visto cómo funciona la tecnología blockchain y cómo se puede sacarle provecho. Para esto, hemos desarrollado una aplicación para que el registro de documentos sea cuestión de unos pocos minutos. Además, hemos implementado dos variantes para reducir el impacto que supone a los usuarios la asimilación de nuevas tecnologías. También se ha explicado cómo funciona la legitimación de firma de forma ordinaria y cómo el uso de nuestro sistema beneficia a los usuarios finales. Asimismo, se ha introducido el concepto de identidades descentralizadas y cómo podría añadirse a nuestro sistema, añadiendo un modo de acceso más robusto. Dicho esto, concluimos que los objetivos de este trabajo final de grado se han cumplido satisfactoriamente.

El hecho de realizar un desarrollo provechoso es debido a unos conocimientos que hemos ido adquiriendo, en una gran variedad de asignaturas, a lo largo del grado. La programación del front-end y el back-end se ha hecho con JavaScript, por lo que el dominar este lenguaje ha sido una ventaja, especialmente la programación asincrónica. Las asignaturas en las que se ha enseñado su uso han sido Tecnologías de Sistemas de Información en Red (TSR) y Desarrollo Web (DEW). Otro conocimiento fundamental que ha ayudado en este trabajo ha sido disponer de bases en sistemas distribuidos y programación teniendo en mente la concurrencia. La asignatura que ha tenido impacto directo en esto ha sido Concurrencia y Sistemas Distribuidos (CSD), además, de forma indirecta, la Computación Paralela (CPA). Por último, la habilidad de preparar una página web y un servidor con base de datos se lo debemos a una combinación de las siguientes asignaturas: Base de Datos (BDA), tecnologías de sistemas de información en red y desarrollo web (de nuevo), Desarrollo Centrado en el Usuario (DCU) y Sistemas y Servicios en Red (SSR).

Para calcular el coste del desarrollo de este proyecto vamos a tener en cuenta una estimación del salario promedio de un recién graduado, que según la universidad europea es entre 18.000€ y 20.000€ en bruto (entre 1.000€ y 1.200€ mensuales en neto / aproximadamente 7€ la hora). Teniendo en cuenta esos números y teniendo en cuenta sólo las horas de desarrollo (96 horas), podemos estimar el coste económico de la mano de obra en 672€, sin contar los costes que pueda conllevar el despliegue del sistema. En el caso de este TFG, ha sido nulo, debido a que hemos usado los servicios del tier gratuito de Amazon Web Services y la red de pruebas de Rinkeby. También debemos considerar que no estamos incluyendo en estos costes las más de 100 horas de formación en tecnologías relacionadas con blockchain que ha requerido este TFG, ni el tiempo invertido en la redacción de este documento.

Para finalizar con la conclusión, me gustaría hablar sobre los objetivos de desarrollo sostenible. La Organización de las Naciones Unidas hace llamamiento universal para poner fin a la pobreza, la destrucción medioambiental y la paz mundial. Para esto, propone una serie de objetivos (Objetivos de Desarrollo Sostenible, ODS[33]) que tienen como meta reducir a la mínima expresión cosas como la pobreza, el hambre, el SIDA o la discriminación de género, entre otros muchos. Teniendo en cuenta la finalidad de este TFG y la tecnología que usa, podría incluirse dentro del criterio de 2 de esos objetivos. Estos son:

1- Industria, innovación e infraestructura
2- Acción por el clima

Menciono el objetivo número 1 debido a que el pilar fundamental de este proyecto es el blockchain, una tecnología innovadora y con oportunidad para influir en el avance en el sector y para la sociedad. En cuanto al objetivo 2, podríamos decir que gracias al hecho de ofrecer una alternativa para gestionar la notarización de documentos remotamente, en unos minutos y con un

[33] https://www.un.org/sustainabledevelopment/es/sustainable-development-goals/

coste reducido, colaboramos con el impacto medioambiental que supone el consumo de combustible y energía a consecuencia del desplazamiento (además de tiempo).

9. Referencias

[1] N. Hashimoto. *Amazon S3 Cookbook*. Packt Publishing, 2015.

[2] https://www.notariado.org/portal/historia-del-notariado

[3] Enrique Giménez-Arnau Gran: "*Derecho Notarial*", Ediciones de la Universidad de Navarra, Pamplona, 1976

[4] https://es.wikipedia.org/wiki/Notario

[5] https://www.notariado.org/portal/tipos-de-documentos-notariales

[6] Satoshi Nakamoto. Bitcoin: *A Peer-to-Peer Electronic Cash System*, 2008. URL https://bitcoin.org/bitcoin.pdf

[7] Alan T. Sherman , Farid Javani, Haibin Zhang, Enis Golaszewski: *On the Origins and Variations of Blockchain Technologies*. IEEE Secur. Priv. 17(1): 72-77 (2019)

[8] Vitalik Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform, 2013 URL https://github.com/ethereum/wiki/wiki/White-Paper

[9] Andreas M.Antonopoulos, Gavin Wood, *Mastering Ethereum*, O'Reilly Media, Inc., 2018

[10] https://www.notariofranciscorosales.com/legitimacion-de-firma-ante-notario/

[11] https://www.boe.es/buscar/act.php?id=BOE-A-1889-4763

[12] https://www.newsbtc.com/proof-of-existence/

10. Glosario

- **Address**: Refiriéndose a las direcciones Ethereum, es un identificador único que es derivado a partir de la clave pública. Partiendo de esta, se calcula su hash. Los últimos 20 bytes del resultado conforman la dirección Ethereum.
- **Algoritmo de consenso**: se refiere al mecanismo por el cual la red blockchain alcanza un consenso sobre su estado entre múltiples nodos.
- **Back-end**: esto hace referencia a toda la lógica que consigue hacer funcionar un servidor (servidor web, restAPI…).
- **Base de datos NoSQL**: es una base de datos no relacional que permite almacenar los datos en documentos.
- **Bucket**: bucket es el término con el que hacen referencia a los contenedores de almacenaje en el servicio S3 de Amazon Web Services.
- **Cliente Ethereum**: esto es una implementación de la especificación técnica de Ethereum. Un ejemplo popular sería Geth.
- **Contrato inteligente/Smart contract**: Un contrato inteligente es un programa escrito para poder ser ejecutado en la máquina virtual de Ethereum.
- **Crypto wallet / wallet / criptomonedero**: la función de un wallet es la de almacenar de forma segura claves públicas y privadas. Un wallet puede ser un programa, como la extensión que se usa en este proyecto (MetaMask), también puede ser un servicio o un dispositivo físico (por ejemplo, Ledger nano S).
- **Cuenta Ethereum**: A la combinación de dirección Ethereum y su clave privada se le llama cuenta Ethereum.
- **DevOps**: es una metodología de desarrollo software que se encarga de la puesta en producción del sistema.
- **DID**: hace referencia a identificadores descentralizados (decentralized identifiers).
- **DNS:** el Domain Name System se encarga de traducir las direcciones ip a nombres más legibles para el ser humano.
- **Endpoint:** en el contexto de los servidores API el endpoint es el punto de acceso a un dispositivo o servicio (URL).
- **Ether:** es la criptomoneda generada por Ethereum. Es usado para pagar por gas, unidades de cómputo usadas en transacciones o cualquier otro cambio de estado en la red.
- **Firma:** en el ecosistema blockchain, una firma consiste en una función criptográfica que permite conocer el origen de la firma sin necesidad de disponer de la llave privada que se ha usado para el encriptado.
- **Framework:** es un entorno de trabajo, con una gran cantidad de herramientas diseñadas para facilitar el desarrollo de software.
- **Front-end:** esto hace referencia al desarrollo enfocado en la capa de presentación. Generalmente mediante el uso de librerías o frameworks orientados a ellos. Aunque se puede realizar de forma rudimentaria con puro HTML, CSS y JavaScript.
- **Gas limit:** esto indica el límite de gas que puede usar una transacción. Generalmente, el wallet será el encargado de configurarlo en base a la complejidad de la interacción.
- **Gas Price:** el precio de gas que ofreces pagar al minero por añadir tu transacción en el blockchain. Cuanto más alto sea este valor, menos tardarán en validar y añadir la transacción. En caso opuesto, cabe la posibilidad de que incluso acabe siendo descartada.
- **Hash:** es una función criptográfica que transforma la información de entrada en una serie de caracteres finita y de tamaño fijo. Es una función de un solo sentido, lo que quiere decir que la dificultad de calcular el resultado a partir de una entrada es baja, pero en el caso opuesto es virtualmente imposible.

- **Marca temporal/timestamp:** tiene la información completa del tiempo: fecha, hora, minutos y segundos.
- **Network:** en este contexto, network se refiere a la red Ethereum en la que el nodo está conectado.
- **Nodos:** los nodos de una red Ethereum son los dispositivos que forman parte de la red. Estos pueden ser clientes
- **Nonce:** contador que acumula el número de transacciones enviadas por el remitente de la transacción.
- **PM2:** es una librería que administra procesos para el entorno de producción de NodeJS.
- **Proof of Work:** es un tipo de algoritmo de consenso que permite mantener un estado común en un entorno distribuido y hostil.
- **Recipient:** esto hace referencia al destinatario de la transacción.
- **scp:** es un comando para transferir archivos de forma segura a través de ssh.
- **ssh:** es un protocolo que permite la conexión a máquinas remotas.
- **Transaction:** una transacción es un mensaje que contiene la información necesaria para alterar el estado de la cadena de bloques.
- **Transaction hash:** el hash de la transacción es un identificador único que se usa como comprobante de la realización de una transacción.
- **v, r, s:** corresponde a los valores para la firma de la transacción.
- **WebSocket:** el websocket proporciona una comunicación bidireccional y full-duplex sobre el protoco TCP.

11. Anexos

Código del contrato inteligente

```
// SPDX-License-Identifier: MIT
pragma solidity >= 0.6;
contract Notary {
  struct Record{
    uint256 timestamp;
    bytes32 r;
    bytes32 s;
    byte v;
  }
  mapping (bytes32 => Record) records;
  function getRecord(bytes32 _hash) public view returns(uint256 timestamp, bytes32 r, bytes32 s, byte v) {
    timestamp = records[_hash].timestamp;
    r = records[_hash].r;
    s = records[_hash].s;
    v = records[_hash].v;
  }
  function writeRecord(bytes32 _hash, bytes32 r, bytes32 s, byte v) public {
    require(records[_hash].timestamp == 0,"This document has already been recorded");
    records[_hash].r = r;
    records[_hash].s = s;
    records[_hash].v = v;
    records[_hash].timestamp = now;
  }
}
```