Universidad Nacional del Sur

Departamento de Ciencias e Ingeniería de la Computación

Blockchain Aplicado al Sistema de Salud: Sistema de Registros Médicos Compartidos

Proyecto Final de Carrera Ingeniería en Sistemas de Computación

presentado por

Juan Iglesias

Bahía Blanca, 2019

Índice

Índice **1**

Introducción **4**

Blockchain **5**

Tipos de Blockchain 5

Smart Contracts 7

Casos de Uso 8

Sistema de Registros Médicos **11**

Arquitectura del Sistema **12**

Smart Contract 13

Ledger 13

Nodo 13

Base de Datos para el Blockchain 14

Users DB 14

WEB API (Backend) 14

Frontend 15

Certificate Authority 15

Herramientas a Utilizar **16**

Framework para la Red Blockchain y Smart Contracts 16

Ethereum vs Hyperledger 16

Componentes 19

Flujo de una transacción 21

Servicios de Ordenamiento 24

Membership Service Provider 25

Docker 28

Introducción 28

Uso de Contenedores 30
Docker Compose 30
Hyperledger Fabric y Docker 31
Node.js y Express.js 31
ReactJS y Material-UI 32
MongoDB 34
Implementación 35
Red Blockchain 35
Arquitectura Hyperledger 35
Estructura de Carpetas 36
Generación, inicio y detención de la red 39
Smart Contract 40
Lenguaje 40
Implementación 41
Backend 42
Funcionalidades 42
Estructura 45
Asincronía en Javascript 45
Fabric Node SDK 48
Wallet 49
Registro de Nuevos Usuarios 51
Enroll de Usuario y Base de Datos 52
Autenticación y JSON Web Tokens 54
Autorización 56
Operaciones de Blockchain 56
Frontend 58
Funcionalidades 58
Estructura 59

Pantallas 60

Axios 67

En la Nube **68**

Amazon Web Services Blockchain Templates 69

IBM Blockchain 70

Conclusiones y Trabajo Futuro **73**

Referencias **74**

Bibliografía **77**

Introducción

La salud digital, o e-Health, definida por la OMS (Organización Mundial de la Salud) como el uso coste-efectivo y seguro de las tecnologías de la información y comunicación (TIC) en el campo de la salud, abarca un gran conjunto de productos y servicios, y presenta un avance de enorme magnitud dentro del área de la sanidad. Uno de los servicios más importantes es el de los Registros Médicos Electrónicos, también conocidos como Historias Clínicas Electrónicas, los cuales corresponden al conjunto de datos de salud propios de un individuo. Esta información es crítica y necesita ser compartida, pero también es privada y de alta sensibilidad, por lo cual el acceso a la misma debe ser regulado.

Blockchain es una lista de registros, llamados bloques, que están unidos entre sí mediante el uso de la criptografía. Cada bloque almacena el hash del anterior, una estampilla de tiempo y la información de la transacción [1]. Debido a esto, el Blockchain es resistente frente a la modificación de la información. Para ser utilizado como una "Distributed Ledger", un blockchain es manejado por una red peer-to-peer que colectivamente adhiere a un protocolo para la comunicación entre los nodos y la validación de nuevos bloques. Una vez almacenada, la información de cualquier de los bloques no puede ser alterada de forma retroactiva sin alterar todos los bloques siguientes, lo que requiere consenso de la mayoría de la red.

El Blockchain ofrece la posibilidad de desarrollar una plataforma en la cual las interacciones de los datos se hagan de forma descentralizada, pero garantizando el control de acceso e integridad de la información.

Este proyecto final consiste en la planificación e implementación de un Sistema de Registros Médicos Electrónicos Compartidos utilizando Blockchain. Se investigarán múltiples tecnologías para su desarrollo, así como también se evaluarán diversas arquitecturas para el diseño del sistema. También se considerará la posibilidad de integrar al sistema otros servicios como la gestión de turnos médicos, la prescripción de medicamentos y certificados, las investigaciones clínicas, y las obras sociales.

Blockchain

El concepto de blockchain ya ha planteado una revolución en la economía, mediante la aparición de criptomonedas, entre las cuales se destaca el Bitcoin. Estas, son monedas que utilizan técnicas criptográficas para salvaguardar la seguridad de las mismas, y evitar ser falsificadas [2]. Se gestionan a través de una base de datos en un "Blockchain", en la que se registran todas las operaciones que se realizan por los distintos usuarios. Además, para mejorar la seguridad, es necesario que cada transacción sea aprobada por toda la comunidad (red). Por ejemplo, si el usuario A transfiere cierta cantidad de bitcoins al usuario B, se modifican los registros que almacenan la cantidad de bitcoins de estos usuarios (Y toda la red debe hacer esta modificación, para mantener la consistencia dentro de la red).

Vale mencionar, que de hecho, el concepto de Blockchain fue introducido por primera vez en el mismo artículo que propone la criptomoneda Bitcoin [3]. En este artículo, el autor propone un sistema de transacciones electrónicas sin depender de una autoridad superior de confianza. Mediante una red peer-to-peer usando proof-of-work (PoW) - Un mecanismo de consenso que requiere que el usuario que solicita el servicio realice algún trabajo, como por ejemplo resolver un problema matemático complejo. Se basa en su asimetría, ya que el problema debe ser difícil de resolver pero sencillo de verificar - para almacenar una historia pública de transacciones que se vuelve rápidamente difícil de alterar para un individuo malicioso a menos que el mismo posea la mayoría del poder de CPU de la red. Los nodos pueden dejar y unirse a la red a voluntad, y el voto se realiza mediante su poder de CPU, validando bloques correctos y rechazando los incorrectos.

Tipos de Blockchain

Podemos mencionar 3 tipos diferentes [4]: **El blockchain público**. Redes transparentes que permiten el acceso de cualquier persona, simplemente descargando la aplicación y conectándose a la misma. Los usuarios de este tipo de redes suelen ser anónimos, no hay administradores y se validan las transacciones con diferentes protocolos de consenso. Un ejemplo de este caso es la red de Bitcoin, que antes mencionamos. Cualquier usuario puede

acceder a la misma instalando el programa adecuado. En la red de Bitcoin, los usuarios son anónimos, por lo que es necesario depositar la confianza en PoW u otros mecanismos de Consenso (Los nodos de la red realizando un acuerdo), mediante computaciones que requieren gran cantidad de tiempo, para confirmar identidades y validar transacciones [5].

Segundo, **blockchain privado**. Son redes con permiso, donde el control lo suele tener una sola entidad, perdiendo un poco la descentralización que se buscaba en primera instancia. No permite el acceso a nuevos usuarios y su uso es principalmente dentro de empresas.

Finalmente, **blockchain híbrido**. Normalmente no están abiertas al público general, pero los miembros de la misma pueden aceptar nuevos participantes para la red. Esta solución es muy útil para empresas que quieran trabajar con todas las partes interesadas.

En el siguiente cuadro es posible observar un breve resumen de las características de cada uno de los tipos:

	Acceso	Participantes	Seguridad	Velocidad de Transacción
Blockchain Público	Cualquiera	Sin permiso y anónimos.	Mecanismos de Consenso Descentralizados.	Lenta
Blockchain Privado	Una única organización	Con permisos e identidades conocidas.	Algoritmos de consenso propios dentro de la organización (Centralizado).	Rápida
Blockchain Híbrido	Organizaciones múltiples seleccionadas.	Con permisos e identidades conocidas.	Algoritmos de consenso propios con varios nodos seleccionados dentro de las organizaciones	Rápida

			(Parcialmente Descentralizado).	

Un cuarto grupo puede ser añadido, el cual ha surgido en los últimos años: **Blockchain como un servicio** (BaaS). Distintas empresas ofrecen servicios de blockchain en la nube, por ejemplo Amazon Web Services, IBM Cloud, Microsoft Azure, etc.

Smart Contracts

¿Pero cómo se puede añadir funcionalidad a una red de Blockchain? Mediante el uso de Smart Contracts. Estos son, como bien su nombre lo dice, Contratos que poseen la capacidad de ejecutarse de forma autónoma y automática. Es decir, al igual que los contratos jurídicos que conocemos en la actualidad, estos establecen un acuerdo entre dos o más partes. Pero con la diferencia de que los mismos producen la ejecución de un programa que se encarga de "procesar" este acuerdo. Es decir que conociendo las condiciones iniciales, puede preverse el resultado final, no hay grises ni vacíos legales.

Este contrato tiene validez y no depende de ninguna autoridad en particular, ya que posee un código visible del cual todos los miembros de la red tienen conocimiento, aceptan y utilizan para sus transacciones. Cuando uno de ellos ejecuta una transacción, la misma se ejecuta en todos los nodos de la red, de modo que es posible saber con facilidad si alguno de ellos modificó el smart contract para su propio provecho, y de esa forma exponer los nodos corruptos.

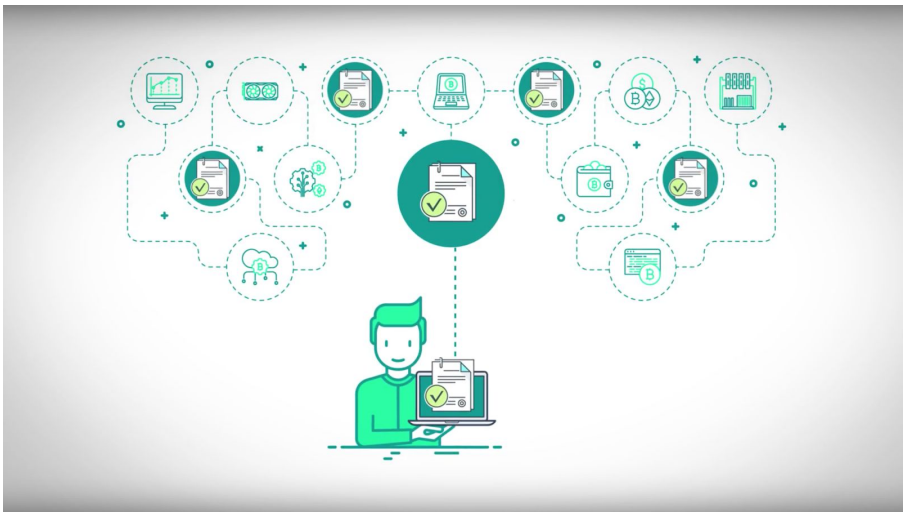


Imagen 1. Fuente: https://www.ethos.io/

Casos de Uso

Dentro de los diferentes casos de uso que podemos encontrar, los más conocidos son:

- **Criptomonedas:** Como se mencionó con anterioridad, justamente con el Bitcoin nació el concepto de blockchain. Ésta fue la primera pero hoy en día hay muchas más: Ethereum, Litecoin, Dash, Zcash, etc.
- **Finanzas y economía:** Dentro de este área, puede ser utilizado en transferencias financieras, pagos de alquiler, etc.
- **Bancos:** Soluciones de Blockchain se están aplicando para mejorar las actividades Financieras, permitiendo la automatización del proceso, eliminando a los intermediarios y comprobaciones manuales, reduciendo los tiempos. Un ejemplo de aplicación de este tipo es el servicio de transferencias internacionales One Pay de Banco santander, que permite que las transferencias lleguen a destino en muchos casos el mismo día o al día siguiente [6].
- **Escribanos:** Éstos se encargan de dar fe de documentos y contratos con su firma. Utilizando Blockchain para gestionar contratos, existe la posibilidad de eliminar la

figura intermediaria del notario o escribano. Realizando de esta manera los contratos, los mismos son totalmente inalterables y se resuelven de manera automática.

- **Identificación (Firma Digital):** Uno de los puntos que podrían ser de mayor uso, la verificación de la identidad de los usuarios para múltiples aplicaciones, como por ejemplo cualquier tipo de contrato que requiera firma.
- **IoT (Internet de las cosas):** Justamente un problema en la conexión de esta gran cantidad de dispositivos para facilitar nuestra vida diaria es la seguridad de los mismos, y sin dudas, el Blockchain ayudaría a mejorar este aspecto. Sin embargo, es importante saber dónde implementarlo, ya que no sería viable en dispositivos con poco poder de computabilidad, como sensores por ejemplo.
- **Trazabilidad de productos:** La posibilidad de conocer el proceso que un cierto producto, por ejemplo alimentos, ha recorrido permite brindar mayor transparencia, seguridad y confianza en el mismo y su proceso.
- **Seguros:** Mediante la ejecución automática de los smart contracts, es posible resolver el tema de los seguros de forma transparente.
- **Votaciones y elecciones:** El Blockchain nos proporciona una plataforma donde podríamos asegurar la Confidencialidad, Integridad y Disponibilidad de la información, lo cual promete mucho a la hora de implementar sistemas de votación electrónicos.

Además todos estos casos mencionados, se podrían seguir mencionando gran cantidad de mercados en los que esta tecnología promete irrumpir. Como podemos observar, el Blockchain es una plataforma que propone una gran cantidad de soluciones a problemas actuales y que puede ser de ayuda para implementar sistemas informáticos que no se creían posible, principalmente por cuestiones de seguridad.

En el caso del presente proyecto, me centraré en los beneficios que puede tener esta tecnología en un área acotada del sistema de salud: Los registros médicos compartidos.

El principal problema de la implementación de este tipo de sistema es que, al compartir cada hospital la información de sus pacientes a otros hospitales, de alguna forma está confiando

esa información a la seguridad de los otros. Es lógico que no exista confianza cuando se trata de un sistema de seguridad ajeno sobre el cual no tenemos ningún tipo de certezas. Aquí es donde el Blockchain promete irrumpir, brindando una capa de abstracción, un sistema entre los distintos hospitales e inyectando una capa de confianza al sistema, donde ahora sí habrá una certeza, el Blockchain llegó para quedarse.

Sistema de Registros Médicos

La idea de realizar un sistema de registros médicos surge por el hecho de querer implementar esta tecnología en un área que tuviese mucha repercusión dentro de nuestra sociedad. Algo muy común, como cambiar de médico, cualquiera sea el motivo, se vuelve muy tedioso por el hecho de que el nuevo especialista no tiene toda la información (Salvo que de alguna manera pertenezca al mismo hospital o clínica que el anterior), lo que produce que se deba explicar la situación al nuevo doctor e incluso repetir pruebas médicas.

Existe una solución a ésto: Un Sistema de Registros Médicos Compartidos (Electronic Health Records System). Mediante este sistema, es posible compartir la historia clínica de una persona entre diferentes hospitales y clínicas. El problema es, en este nuevo caso, la seguridad de los registros. ¿Quién es el encargado de la misma? ¿Dónde se almacenan los datos? Como todo sistema de tal magnitud, la seguridad depende de que todas las organizaciones que forman parte del mismo se encuentren protegidas. Es decir, que la seguridad de los datos depende, para un cierto hospital, de la seguridad en el resto de los hospitales. No suena a algo muy prometedor.

Debido a los problemas que surgen al intentar implementar un Sistema de Registros Médicos Compartidos, aparece una nueva solución, agregando todos los beneficios del Blockchain para mayor seguridad. Mediante el uso de la criptografía, ésta tecnología nos permitirá salvar la integridad de la información, evitar la modificación de los datos y documentar toda acción realizada en el sistema de forma inalterable.

Arquitectura del Sistema

Primero, es importante mencionar cómo se va a construir el sistema y las partes que lo componen. Imaginamos que cada hospital es una organización, con su propia red interna, y cada uno de estas organizaciones participa del "Blockchain", lo que significa que tienen acceso a los datos de todos los pacientes de la red.

Como se puede observar en el diagrama a continuación, la Arquitectura del sistema se simplificó para 2 Hospitales, tanto para su diagramación como para su implementación, debido a temas a tiempo y poder computacional necesario para simular estos sistemas en una Computadora Personal. Sin embargo, notaremos que el sistema es fácilmente escalable.

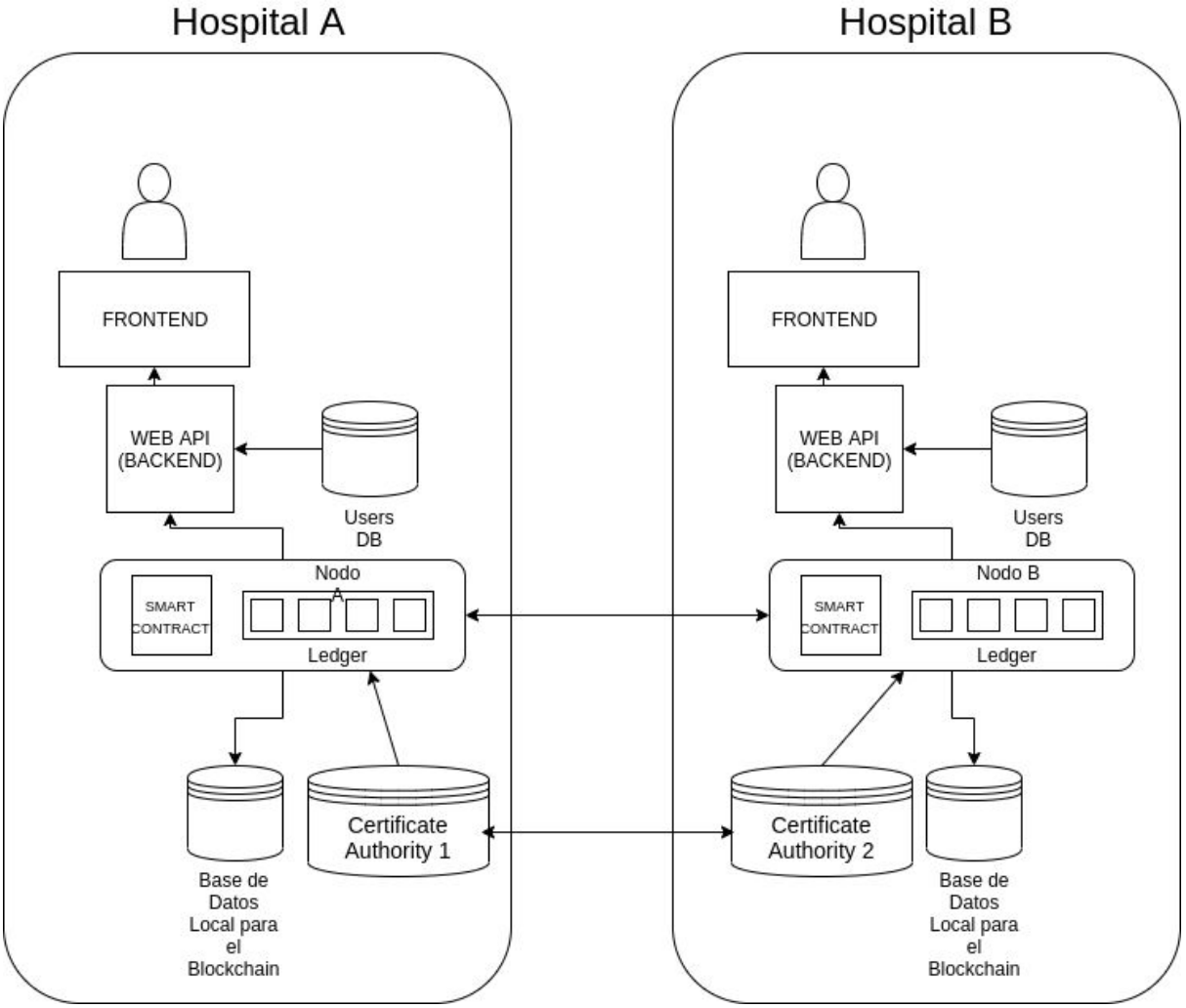


Imagen 2: Arquitectura del Sistema

A continuación, se mencionan las distintas partes que componen el sistema y sus funcionalidades.

Smart Contract

Este es el contrato que posee la información de las operaciones a ejecutar en el Blockchain. Para nuestro sistema, estas serían: Obtener datos de un paciente, agregar un nuevo paciente, añadir un registro a un paciente, etc.

Cabe aclarar que, como fue mencionado con anterioridad, los Smart Contracts de los distintos nodos deben ser idénticos, y es sencillo notar si algún nodo está ejecutando diferente código, pues el resultado no será el mismo en aquel nodo.

Ledger

Es un registro de todas las transacciones ejecutadas en nuestra red. Toda esta información se encuentra almacenada de forma segura y protegida criptográficamente. Esta es compartida entre los nodos de la red y debe ser idéntica en cada uno de los mismos, lo que significa que es fácilmente detectable cualquier tipo de modificación indebida.

Nodo

Es un integrante de nuestra red de Blockchain, que tiene instalado el “Smart Contract” y por ende puede llevar a cabo las operaciones que el mismo permite.

No hay un número máximo de nodos para cada Organización (Hospital). Aunque en este caso se haya realizado un mapeo 1-en-1 de nodos por organización, se realizó para simplificar el diagrama. En la realidad cada hospital seguramente posea mucho más que un nodo.

Hay dos cuestiones a considerar respecto al número de nodos. Primero, un mayor número de nodos le dan complejidad a la red, la hacen más costosa ya que las transacciones se deben ejecutar en todos, pero a su vez la hacen más segura ya que la vuelven más difícil de penetrar. Recordemos que, como en toda red distribuida, para tener control de la misma o realizar algún tipo de ataque malicioso es necesario controlar o atacar un gran porcentaje de nodos de esta. Por ésto, llega la segunda cuestión: es necesario corroborar la cantidad de nodos de las diferentes organizaciones que componen la red. Ninguna de las mismas debería llegar a tener un porcentaje muy alto, pues entonces la red quedaría prácticamente a control de ésta organización.

Base de Datos para el Blockchain

Es una base de datos propia de cada Hospital donde se almacenarán el estado actual del Blockchain con toda la información del mismo.

Users DB

Es una base de datos propia de cada Hospital con los datos de sus Usuarios (Doctores). La utilidad de la misma es servir para realizar un LOGIN que autorice a los doctores a las diversas plataformas.

WEB API (Backend)

Es una aplicación que se encargará de proveer las herramientas necesarias para interactuar con el Blockchain.

La idea es respetar el esquema REST, el cual representa una interfaz entre sistemas que utiliza HTTP para obtener datos o indicar la ejecución de diferentes operaciones sobre los mismos. Se basa en un **protocolo cliente/servidor sin estado**, es decir que cada mensaje HTTP que se envía contiene la información necesaria para entender y procesar la transacción,

un conjunto de **operaciones bien definidas**, HTTP define un conjunto pequeño de operaciones como POST, GET, PUT, DELETE y se pueden utilizar para esto, una **sintaxis universal para identificar los recursos**, mediante el uso de URIs, y finalmente, la utilización de **hipermedios**, para vincular diversos recursos con el resto de los mismos.

La aplicación definirá rutas para el acceso a pacientes y sus registros, así como un sistema de login para evitar que personas desautorizadas tengan acceso a la información.

Frontend

Esta aplicación web se comunica con la Web API y ofrece al usuario una interfaz agradable mediante la cual interactuar con el sistema. Se buscará que ésta sea moderna, cuidando la UX, la *responsiveness* (adaptación de la página a los distintos tamaños de pantalla de los dispositivos que tenemos en la actualidad), sin descuidar por supuesto la seguridad del sistema.

Certificate Authority

Una CA es una entidad que emite certificados digitales. Éstos certifican que cierta clave pública pertenece a una determinada persona, la cual es especificada en el mismo certificado. Ésto permite a terceros confiar en firmas realizadas con la llave privada que corresponde a la llave pública certificada. Los certificados emitidos por la CA están especificados en el estándar **X.509**.

Este tipo de certificados se utiliza para crear conexiones seguras desde un punto A a un punto B, evitando que una tercera parte maliciosa pueda hacerse pasar por uno de los comunicantes, ataque conocido como *man-in-the-middle*.

Herramientas a Utilizar

En la presente sección se enumeran las herramientas que serán utilizadas para la implementación del sistema propuesto, se evaluarán las diferentes alternativas y se expresarán los motivos que dieron lugar a la elección por una u otra.

Framework para la Red Blockchain y Smart Contracts

Ethereum vs Hyperledger

Para esta elección se realizó bastante investigación al respecto y se encontraron dos alternativas muy interesantes: Ethereum [7] con Solidity [8], e Hyperledger [9] Fabric [10] de IBM.

Ethereum es una plataforma de código abierto, pública, distribuida y basada en blockchain que permite la implementación de Smart Contracts para correr programas en la misma. El token generado por esta plataforma es llamado Ether, puede transferirse entre diferentes cuentas y se utiliza como compensación para los nodos "mineros" de la red los cuales se encargan de computar operaciones. Provee una máquina virtual descentralizada llamada the Ethereum Virtual Machine (EVM), la cual puede ejecutar smart contracts en la red.

Imagen 3: Logo de ethereum

Los Smart contracts de esta plataforma pueden estar basados en diversos lenguajes de programación, éstos se compilan luego a EVM bytecode y se despliegan en el blockchain para una posterior ejecución. Entre los múltiples lenguajes de programación disponibles, se destaca Solidity como el más popular. Es orientado a contratos, de alto nivel, y la sintaxis es

similar a las de Javascript y C++. Otros de los que más se destaca es Vyper. También es orientado a contratos, pero en este caso basado en python y destaca por su facilidad a la hora de utilizarlo y la seguridad que provee.

La principal ventaja de esta plataforma es que nos provee la posibilidad de correr nuestras aplicaciones en una cadena de bloques pública. El problema que ésto conlleva es que no nos sirve para la aplicación que nosotros queremos desarrollar. Necesitamos una cadena de bloques privada, a la cual solo tengan acceso las distintas organizaciones que componen nuestra red privada (hospitales, etc).

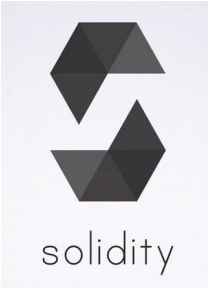


Imagen 4: Logo de solidity

Hyperledger Project es también una plataforma de código abierto para el blockchain. Fue iniciado en 2015 por la Fundación Linux y está apoyado por importantes compañías como el caso de IBM, Intel o SAP SE. La idea detrás de su creación es la de unir esfuerzos en la utilización del blockchain como herramienta para la solución de problemas que cruzan diversas industrias como las finanzas, bancos, IoT, cadena de suministros, etc, desarrollando estándares y protocolos abiertos.

A diferencia de lo que ocurre con el resto de las plataformas que hemos mencionado (Ethereum y Bitcoin), no existe ningún tipo de moneda o token habitando dentro de la red de Hyperledger. Esta decisión se debe a que los objetivos estratégicos de esta plataforma están orientados únicamente a la creación de aplicaciones tecnológicas.

Imagen 5: Logo de Hyperledger Project

Dentro de los diversos proyectos que componen Hyperledger, destaca **Hyperledger Fabric**, de IBM. Es un framework para desarrollar soluciones y aplicaciones dentro de un sistema distribuido con permisos, orientado al uso empresarial gracias a la posibilidad de realizar transacciones privadas.

Brinda varios lenguajes en los que programar los contratos inteligentes, los cuales en ésta plataforma son llamados **Chaincodes**. Inicialmente el único lenguaje soportado era Golang (De google), aunque ahora ya es posible programarlos en NodeJS, Java e incluso Javascript. Además se proveen varios SDK para poder crear aplicaciones que interactúen fácilmente con el Blockchain.

Algunas de las características particulares que propone Hyperledger Fabric y se deben tener en cuenta a la hora de pensar en esta plataforma como solución son los siguientes: Los participantes deben ser identificables, desaparece el anonimato de la mayoría de las redes de blockchain. Las redes deben ser **permisionadas**, lo que significa que los participantes se deben conocer entre ellos, aunque no es necesario que haya confianza, eso se encarga de otorgar la red. Se necesita un alto rendimiento de transacciones. Y, por último, se debe respetar la privacidad y confidencialidad de algunas transacciones y datos de los agentes interesados.

Imagen 6: Logo de Hyperledger Fabric

Debido a que Hyperledger Fabric parece proveernos la solución adecuada para nuestro problema, gracias al uso de transacciones privadas en redes permisionadas, ampliaremos la información respecto a esta plataforma, que será la seleccionada para llevar a cabo el presente proyecto. Mencionaremos los distintos componentes de una red implementada con esta plataforma y nos adentraremos en sus distintos servicios.

Componentes

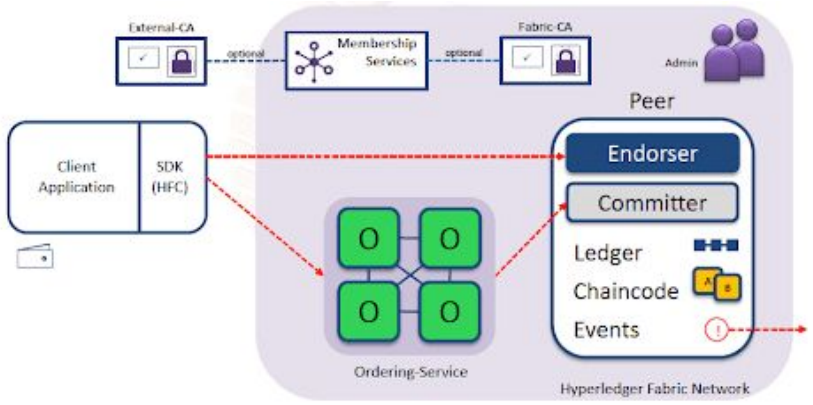


Imagen 7. Fuente: https://walkingtree.tech/actors-architecture-transaction-flow-in-hyperledger-fabric/

En la presente sección se verán los diferentes componentes que forman parte de una red de Hyperledger Fabric [11] [12] .

Gracias al uso de canales privados, Hyperledger Fabric permite que coexista tanto la información pública con la información privada. Estos canales representan rutas privadas y confidenciales mediante las cuales la información y transacciones son únicamente visibles en una porción de la red, para aquellos miembros que forman parte del mismo.

Un "Ledger" está compuesto por dos partes relacionadas entre sí. El Blockchain en sí, y la base de datos de estado. A diferencia de otros "ledgers", un blockchain es inmutable, es decir, que una vez que un bloque fue añadido a la cadena, no puede cambiarse. Por otro lado, la base de datos de estado contiene un set de pares clave-valor que fueron agregados, modificados o eliminados por las distintas transacciones que fueron validadas en el blockchain.

Podríamos pensar como si hubiese un "ledger" por cada canal privado que tenemos, aunque en la realidad cada peer que forma parte del canal contiene una copia del ledger, y a su vez se ocupa de mantener la consistencia de la misma respecto al resto de éstas mediante el consenso, que veremos a continuación. Al tipo de ledger que acabamos de mencionar también se le conoce comúnmente como **Distributed Ledger Technology** (DLT).

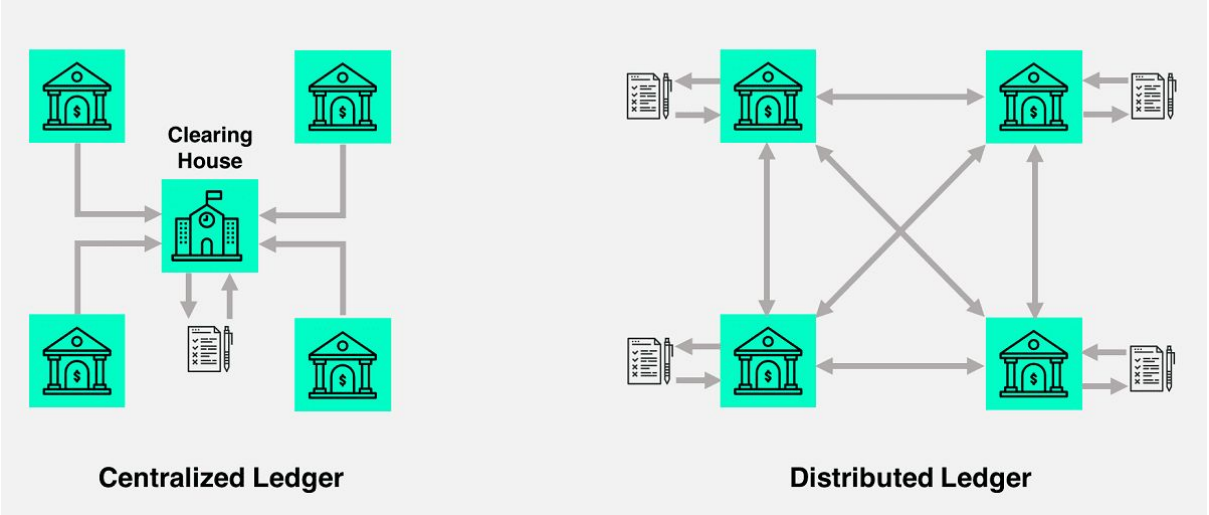


Imagen 8. Centralized ledger vs Distributed Ledger Fuente: https://tradeix.com/distributed-ledger-technology/

Los **smart contracts o chaincode**, como ya vimos anteriormente, es código que, invocado por una aplicación externa a la red de blockchain, permite acceder o modificar el conjunto de pares clave-valor que forman parte del "World State" de la misma. En Hyperledger Fabric, este código se instala en cada uno de los peers e instanciado dentro de canales privados.

El **Membership Service Provider** (MSP) es un componente que ofrece una abstracción en lo que respecta a todos los mecanismos criptográficos y protocolos que ocurren detrás de la emisión y validación de certificados, junto con la autenticación de usuarios. Dentro de una red de hyperledger puede haber uno o múltiples MSP, lo que provee modularidad y descentralización en este tipo de operaciones. Más adelante se darán más detalles de éste componente.

El **consenso** también forma parte de los conceptos destacados de cualquier red blockchain. Corresponde al proceso mediante el cual se mantienen las transacciones sincronizadas en toda la red. Para asegurar que las mismas se realicen en el mismo orden, se registren únicamente si son validadas y rechacen si no es así.

Pero el consenso es mucho más que ponerse de acuerdo en el orden. Es importante durante todo el flujo de una transacción: Desde la proposición y aprobación (**endorsement**), al ordenamiento, hasta la validación y commit de la misma.

Flujo de una transacción

A continuación se explicará detalladamente el flujo de la transacción mediante un ejemplo de una aplicación de compra/venta de rábanos [13], desde que es propuesta hasta su respectivo commit o, en su defecto, rechazo de la misma.

Asumiremos que el usuario de la aplicación se ha registrado e identificado con la CA (Certificate Authority) y ha recibido el **material criptográfico** - Un certificado X.509 conteniendo la clave pública del usuario, su clave privada, junto con metadata específica de fabric - que utilizará para autenticarse en la red. Además, el mismo chaincode fue instalado en cada uno de los peers. También fue definida una política de **endorsement**, estableciendo que todos los peers deben aprobar la transacción.

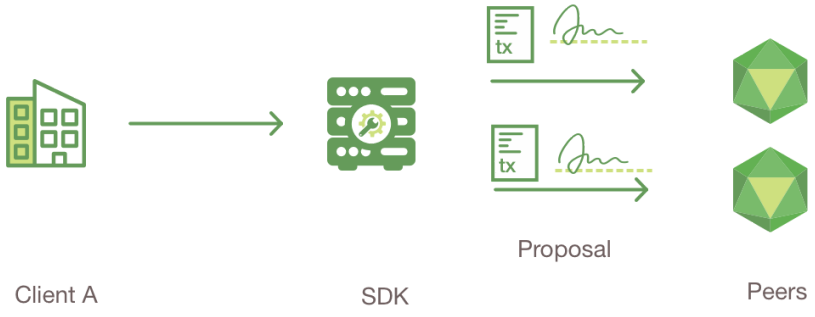


Imagen 9: Cliente envía petición, el sdk le da un formato y la redirige a los peers necesarios.

Inicialmente, un usuario envía una petición para comprar rábanos. Esta petición, de acuerdo con la política de endorsement, debe ser aprobada por todos los nodos, por lo que es reenviada a éstos - supongamos para el ejemplo que hay únicamente 2 peers, A y B.

Luego, la **propuesta de transacción** es construida, el cual es una petición de invocar una función del chaincode con ciertos parámetros.

El SDK funciona como intermediario, le da un formato específico a la transacción y utilizando las credenciales criptográficas del usuario produce una firma única para esta transacción.

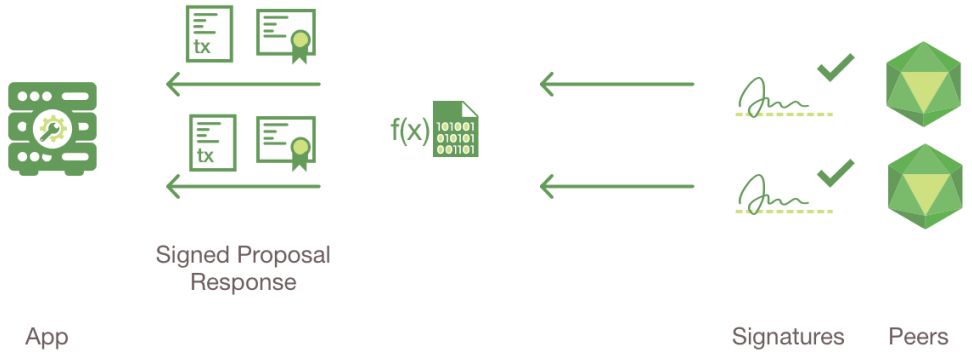


Imagen 10: Los endorsement peer verifican que todo sea correcto, ejecutan la función y devuelven la respuesta.

Los endorsement peer verifican que la propuesta es correcta, que no fue enviada antes, que la firma es válida, y que quien envía (Cliente A en este caso) está autorizado para realizar esta operación en el canal. Si todo ésto es válido, entonces cada uno de los peers ejecutan la función del chaincode respectiva y, sin modificar el estado del ledger, devuelven el valor de respuesta, junto con un read-set y un write-set. Estos valores son enviados, junto con la firma de cada uno de los peers, al SDK.

Imagen 11: El SDK verifica las firmas de los endorsement peers.

La aplicación entonces verifica las firmas y compara las respuestas para determinar si son iguales. Si se trata de una operación que modifica el ledger, entonces la aplicación debe ser enviada al servicio de ordenamiento, previamente chequeando que la política de aprobación se haya cumplido.

Imagen 12: La aplicación se comunica con el servicio de ordenamiento.

La aplicación entonces hace broadcast de la propuesta de petición junto con la respuesta de la misma al servicio de ordenamiento, el cual los ordena y crea bloques de transacciones.

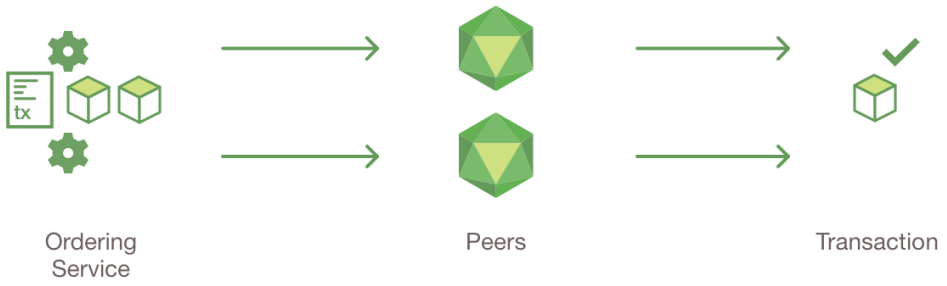


Imagen 13: Los peers verifican las transacciones.

Luego, los bloques son enviados a todos los peers del canal y las transacciones de los bloques son validadas para asegurarse que la política de aprobación se cumplió y asegurarse que no hubo modificaciones para las variables que componen el read set. Luego, las transacciones dentro de cada bloque se clasifican en válidas o inválidas.

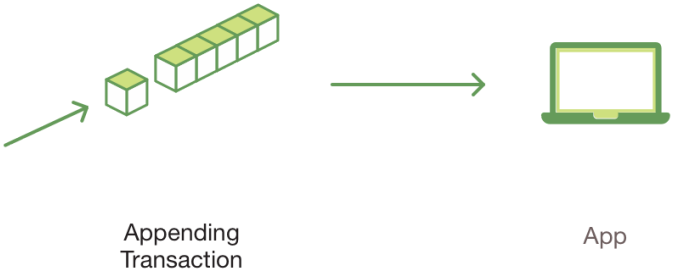


Imagen 14: Se agregan las nuevas transacciones.

Finalmente, cada peer agrega el bloque a la cadena del canal, y por cada transacción válida los write-sets son commiteados a la base de datos del estado actual. Luego se emite un

evento, avisado a la aplicación que la transacción fue correctamente procesada y agregada a la cadena, o invalidada si así lo fuese.

Servicios de Ordenamiento

Para lograr el ordenamiento adecuado de los bloques, tenemos diversos servicios disponibles. Por ejemplo, PBFT (Practical Byzantine Fault Tolerance) provee un mecanismo para que las copias se comuniquen entre sí y se mantengan consistentes. Por otro lado, en la red de Bitcoin el orden es logrado mediante un proceso llamado "minería", donde se realiza una competencia computacional para resolver un acertijo matemático. Fabric fue diseñado para permitir la elección del mecanismo de consenso que mejor se adapte a las necesidades de los miembros de la red, entre los cuales se destacan SOLO, Raft y Kafka [14].

El **servicio de ordenamiento SOLO** es una excelente elección a la hora de realizar desarrollo o testing, sin embargo no es viable en producción. Ésto se debe a que, aunque procesa las transacciones de la misma forma que métodos más avanzados como lo son Kafka y Raft, no brinda tolerancia a las fallas.

El segundo, **Raft**, es un servicio de ordenamiento orientado a redes en producción. Utiliza un esquema Leader-Follower, en el que un líder es seleccionado dinámicamente de entre los nodos de ordenamiento disponibles del canal. El líder es el encargado de replicar los mensajes hacia estos nodos. Este algoritmo es tolerante a fallas ya que, es posible perder nodos (Siempre y cuando sean menos de la mitad) e incluso el líder - llegado el caso se selecciona uno nuevo.

El tercero y último, **Kafka**, es el otro servicio tolerante a fallas que nos brinda Hyperledger Fabric. Usa el mismo concepto de Leader-Follower que el anterior en los que las transacciones, ahora llamadas "mensajes", se replican desde el nodo líder al resto de los nodos. El manejo del cluster Kafka, junto con la coordinación de tareas, control de acceso, etc, es manejado junto con **ZooKeeper**.

Si bien los dos últimos son similares, tienen algunas pequeñas diferencias que pueden resultar importantes a la hora de decantarse por uno o el otro.

Raft es más fácil de configurar que Kafka, debido a que éste último es mucho más complejo y posee más componente que deben ser administrados. También Kafka posee sus propias versiones, las cuales se deben coordinar con los nodos ordenadores. Por el contrario, en Raft, todo está embebido dentro de cada nodo ordenador, simplificando las cosas.

Kafka y ZooKeeper no están preparados para correr en grandes redes. De hecho, es necesario que una organización sea la encargada de ejecutar el cluster Kafka. Debido a ésto, tener nodos ordenadores de diferentes organizaciones no tiene mucho sentido, ya que todos se ejecutarán en el mismo cluster, el cual estará en control de una única organización. Por otro lado, Raft permite que cada organización tenga sus propios nodos ordenadores, logrando así un sistema mucho más descentralizado.

Además, Raft tiene soporte nativo en Hyperledger Fabric, mientras que para utilizar Kafka, si bien es compatible, es necesario descargar las imágenes necesarias y tener conocimientos en el manejo de Kafka y ZooKeeper, es decir, que no hay gran soporte y documentación respecto a ésto.

Membership Service Provider

Un MSP [15] [16] identifica qué CAs son confiables, para definir los miembros de un cierto dominio, por ejemplo una organización. Puede identificar el rol de un miembro, junto con sus privilegios de acceso en el contexto tanto de la red como del canal en sí.

La configuración de un cierto MSP es anunciada a todos los miembros del canal mediante un **Channel MSP**. A ésto se le suma también que cada peer o nodo ordenador mantiene su propio **Local MSP** para autenticar miembros y definir permisos.

Una organización es manejada por un grupo de miembros mediante un único MSP, exceptuando casos donde puede haber múltiples MSP, por ejemplo si diferentes canales son utilizados para diferentes funciones entre organizaciones.

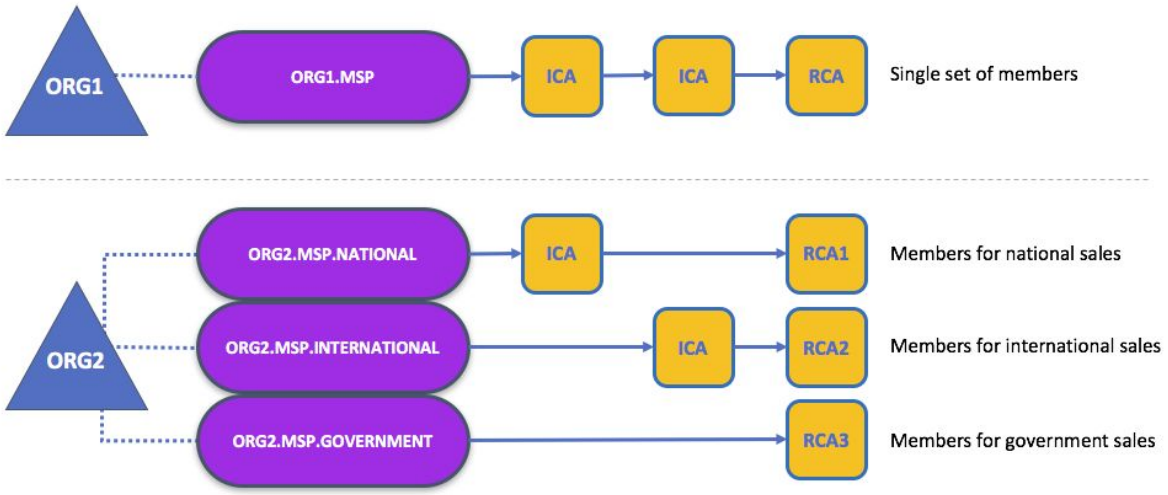


Imagen 15: Ejemplo de una organización con múltiples MSP.

Como se mencionó antes, los MSP aparecen en dos lugares del blockchain: En los canales y localmente en cada miembro, ya sea peer, orderer o cliente (user). Permiten a los nodos definir los permisos de los mismos, y a los usuarios a autenticarse en transacciones como miembros del canal. Por otro lado, en cuanto a los canales, todos los participantes de éste tendrán acceso al Channel MSP y podrán entonces autenticar a los participantes del mismo. Entonces, por ejemplo si una organización quiere sumarse a un canal, será necesario que el MSP incorpore a la cadena de confianza del mismo a todos los miembros de esta organización. De otra manera, todas las transacciones iniciadas por ésta serían rechazadas.

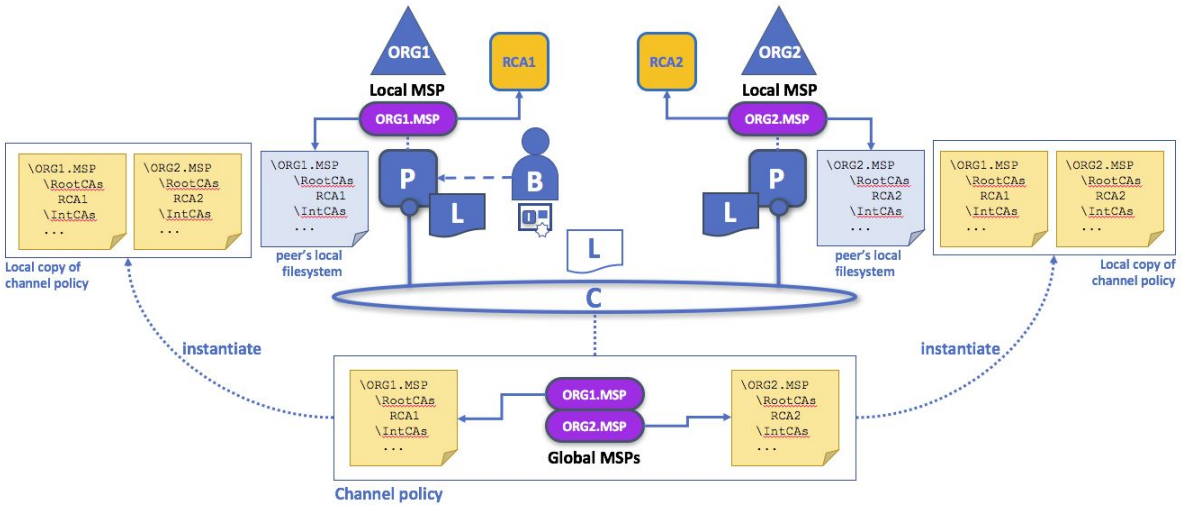


Imagen 16: Ejemplo de una red con múltiples organizaciones y MSPs locales y globales.

Supongamos la situación en que un administrador (B) se conecta al peer mediante una identidad emitida por RCA1 y almacenada en su local MSP. Si B intenta instalar un Smart

Contract en el peer, éste último corrobora primero, en ORG1.MSP (local), si realmente B es miembro de ORG1. En caso de que así lo sea, la operación se realizará correctamente. Pero luego supongamos que B desea instanciar ese Smart Contract en todo el canal C. Debido a que esta operación actúa sobre el canal, es necesario que todas las organizaciones que pertenecen al mismo estén de acuerdo. Entonces, el peer debe chequear primero con los MSPs del canal (globales), cuya copia -sincronizada- se almacena dentro del sistema de archivos de cada uno de los nodos miembros del canal.

Por último, es muy importante conocer la estructura de los MSP, junto con todos los elementos que hacen posible el funcionamiento.

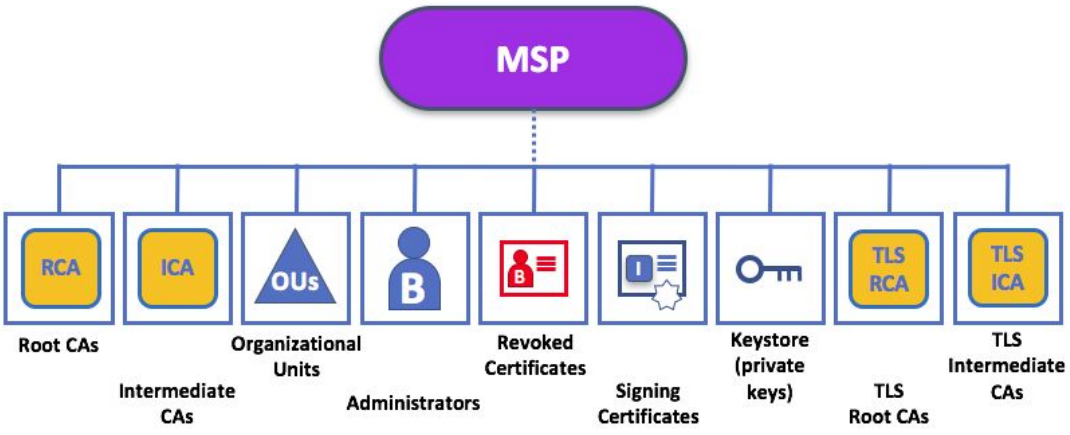


Imagen 17: Archivos de un MSP

La carpeta **Root CAs** contiene una lista de certificados auto-firmados por las autoridades de certificación raíz de confianza de las organizaciones representadas por este MSP. Debe haber al menos un certificado X.509 en esta carpeta.

La carpeta **Intermediate CAs** es opcional y contiene una lista de certificados de las autoridades de certificación intermedias. Estos certificados deben estar firmados por un Root CA o por un Intermediate CA cuya cadena de firmas finalice en un Root CA.

Las **Organizational Units** o Unidades Organizacionales representan partes diferentes de la organización con diferentes responsabilidades. No se entrará más en detalle respecto a este tema debido a que no es de importancia para el correspondiente proyecto.

Administrators contiene una lista de identidades que definen a los diferentes usuarios con éste rol dentro de la organización.

Revoked Certificates permite administrar una lista de identidades que fueron revocadas. Para identidades basadas en X.509 se utilizan pares de Subject Key Identifiers (SKIs) y Authority Access Identifiers (AKIs), que son chequeados cada vez que un certificado es utilizado.

La carpeta **Signing Certificates** contiene un certificado X.509 que identifica al nodo. Ésto, combinado con la clave privada que se encuentra en la carpeta **KeyStore**, permiten al nodo autenticarse en los mensajes que envía al resto de los participantes de la red, ya que el certificado indica cuál es su clave pública que desencripta los mensajes codificados con la privada.

Finalmente, **TLS Root CA** y **TLS Intermediate CA** son carpetas opcionales que únicamente están presentes en el caso de utilizar la comunicación TLS y contiene una lista de certificados X.509 de CAs de confianza de la organización para la comunicación TLS.

Docker

Introducción

Docker [17] es un proyecto de código abierto que está compuesto por un conjunto de productos que, mediante virtualización a nivel del Sistema Operativo, entregan software en paquetes que son llamados **containers**.

Los containers se encuentran aislados entre sí y cada uno de éstos posee su propio software, librerías y archivos de configuración, aunque pueden comunicarse mediante canales bien definidos. Todos los containers corren bajo el mismo kernel del SO y por éste motivo son mucho más livianos que las máquinas virtuales.

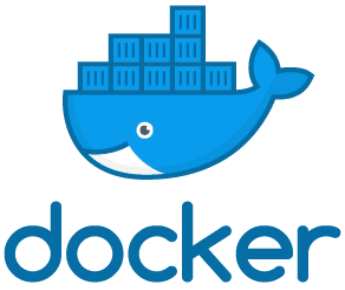


Imagen 18: Logo de Docker

El software sobre el que corren los containers se llama **Container Engine**, y en particular el de Docker recibe el nombre de **Docker Engine**. El Container Engine se encarga de proveer al usuario de una interfaz y le permite obtener las imágenes de los diferentes contenedores - Conjunto de instrucciones para construir los contenidos junto con la metadata del repositorio del contenedor- y extraerlas para su creación.

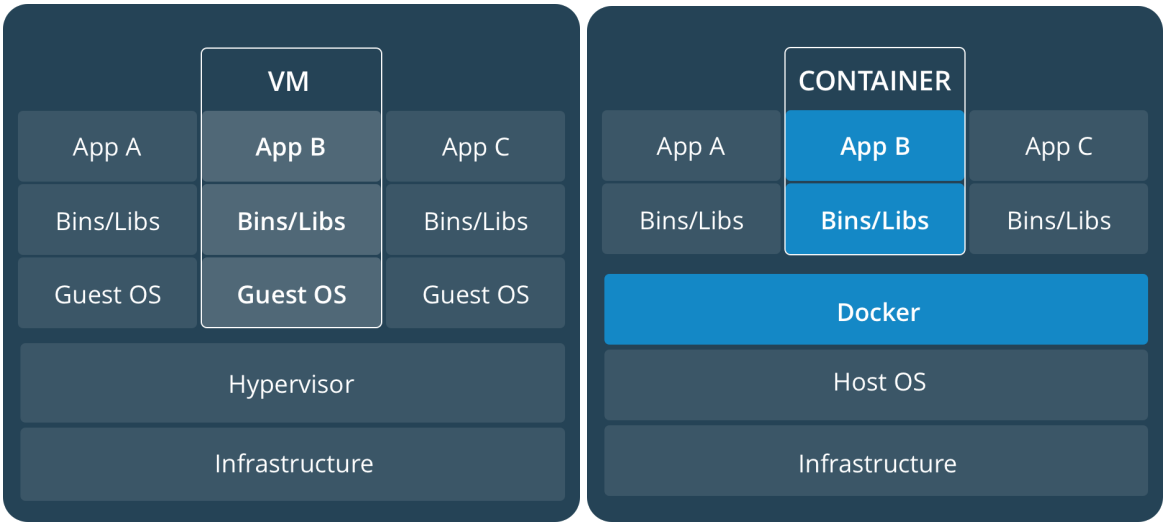


Imagen 19: VM vs Containers. Fuente https://docs.docker.com/get-started/

Hoy en día esta plataforma se ha vuelto muy popular debido a que los contenedores son:

- **Livianos**: Utilizan y comparten el kernel del sistema operativo sobre el que se ejecutan.
- **Flexibles**: Incluso las aplicaciones más complejas pueden colocarse en éstos.
- **Intercambiables**: Pueden realizarse updates y upgrades on-the-fly.
- **Portables**: Pueden construirse localmente y luego deployarse en la nube, por ejemplo.

- **Escalables**: Pueden incrementarse fácilmente las réplicas y distribuirse automáticamente.
- **Apilable**: Pueden apilarse servicios on-the-fly.

Uso de Contenedores

En el pasado, para escribir por ejemplo una aplicación con Python, era primero necesario instalar un conjunto de programas de forma tal que el ambiente de la máquina sea el indicado para poder correr la aplicación de la forma esperada. Además ese ambiente luego deberá ser el mismo en producción para que allí también funcione correctamente.

Con Docker, es tan simple como descargar una imagen del ambiente de Python y construir el código de la aplicación junto con la misma.

Para realizar esto es necesario configurar un archivo llamado **Dockerfile**. En el mismo debemos especificar la imagen, el directorio del contenedor donde serán copiados los archivos, comandos para la instalación de dependencias, comandos para la ejecución del programa, etc. Luego creamos la nueva imagen con el comando "**docker build**" desde el directorio raíz y finalmente podemos correr la aplicación mediante el comando "**docker run**".

Docker Compose

El archivo **docker-compose.yml** es un archivo YAML que define cómo se comportará un contenedor en producción.

En este archivo se especifica la composición de la red, junto con los nombres de los distintos servicios que ejecutaremos y las imágenes asociadas a los mismos. Podemos especificar para cada servicio el tiempo de CPU dedicado, la memoria RAM y el mapeo de puertos del host a la aplicación.

Hyperledger Fabric y Docker

Es necesaria la utilización de Docker junto con Hyperledger: Nos permitirá correr los distintos miembros de nuestra red blockchain mediante imágenes que serán descargadas automáticamente desde **Docker Hub**, un repositorio en el que puede distribuirse las imágenes de los usuarios o socios de Docker.

Node.js y Express.js

Imagen 20: Logo de Node.js

Node.js [18] es un entorno de ejecución de Javascript orientado a eventos asincrónicos, diseñado para construir aplicaciones en red de forma escalable. De esta forma, se posibilita el manejo de múltiples conexiones concurrentes, sin embargo, si no hay trabajo que hacer Node.js se encontrará durmiendo.

Este modelo, contrasta con el modelo de concurrencia más común que tenemos en la actualidad, que es la utilización de hilos del sistema operativo. Node.js presenta un bucle de eventos como entorno en vez de una librería, lo que lo diferencia de otros sistemas de este tipo, en los que una llamada bloquea el bucle, es que Node ingresa al bucle automáticamente después de ejecutar el script de entrada y sale del mismo cuando no hay más eventos que ejecutar.

Debido a ésto, es posible programar un servidor muy liviano y eficiente que se encargue correctamente de despachar todos los pedidos de múltiples clientes.

Imagen 21: Logo de express.js

Por otro lado, express [19] es una infraestructura web rápida, minimalista y flexible para Node.js. Permite programar con facilidad una API mediante el uso de middlewares y rutas.

Se decidió por la utilización de estas dos herramientas a la hora de programar el Backend ya que se tenían conocimientos previos de las mismas y buenas experiencias en su utilización. Considero que son tecnologías que permiten realizar una gran modularización de las aplicaciones y lograr una aplicación muy ordenada.

ReactJS y Material-UI

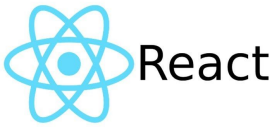


Imagen 22: Logo de ReactJS

React [20] es una biblioteca de Javascript que ayuda en la creación de UIs interactivas de forma sencilla. Permite al usuario diseñar múltiples vistas simples y se encarga de actualizar y renderizar la que corresponda de acuerdo a la situación.

Está basado en componentes encapsulados que poseen su propio estado, lo cual permite al usuario complejas interfaces combinando los mismos.

La lógica de los componentes está escrita en Javascript, lo cual permite pasar los datos a través de la aplicación, manteniendo fuera del DOM el estado de la misma, haciéndola más eficiente.

React utiliza JSX para mostrar sus componentes. JSX es una sintaxis de etiquetas que no es ni un string ni HTML, aunque se le parece. De esta manera, React permite combinar el poder de Javascript junto con HTML para lograr agregar lógica en la renderización de componentes

Se decidió por esta tecnología para el Frontend debido a que actualmente recibe un gran interés por parte de la industria, ha crecido mucho en los últimos años y es actualmente una de las tecnologías más utilizadas para el diseño y desarrollo de interfaces de usuario. Todo esto, sumado al interés por aprender la misma motivaron su elección.

Imagen 23: Logo de Material-UI

Sumado a ReactJS, se buscó también algún framework o librería que ofrezca componentes más estilizados que los que react trae por defecto y que sea de ayuda en el diseño de la interfaz.

En el proceso de investigación se encontraron varias herramientas como React Bootstrap, Material Kit React y Blueprint entre otros. Sin embargo el que resultó más interesante, debido a los elementos gráficos que ofrecía, fue Material-UI [21].

Material-UI es un framework para react que permite diseñar interfaces gráficas siguiendo las normativas de Material Design, de Google. Ésto resultó muy interesante ya que Material Design está inspirado por el mundo real y sus texturas, incluyendo el reflejo de la luz, la proyección de las sombras y las diferentes superficies, que simulan el papel y la tinta. Esto, sumado al hecho de que hoy en día las mayoría de las aplicaciones que utilizamos en nuestros teléfonos celulares están diseñadas siguiendo estos estándares, permiten obtener interfaces que resultan muy amigables para el usuario, logrando una navegación mucho más cómoda.

MongoDB

Imagen 24: Logo de mongoDB

Finalmente, para el diseño de la base de datos Local de cada hospital, se decidió por la utilización de mongoDB [22], una base de datos NoSQL que apunta a la agilidad y escalabilidad.

Es una base de datos ágil ya que permite a los esquemas cambiar rápidamente a medida que las aplicaciones evolucionan, proporcionando índices secundarios, un lenguaje completo de búsquedas y consistencia eventual.

Se decidió por utilizar esta tecnología ya que es muy sencilla de utilizar y provee facilidades para utilizarse junto a Node.js gracias a la librería **mongoose**.

Implementación

Red Blockchain

Arquitectura Hyperledger

Basándose en la arquitectura diagramada con anterioridad, se comenzó con el diseño del sistema EHRecords partiendo de las características provistas por Hyperledger Fabric para la misma.

La red (Ver Imagen 25), estará compuesta por solamente dos hospitales para simplificar la complejidad de la misma, donde cada uno de los nodos posee dos peers (Peer0 y peer1). En ambos casos el Peer0 será el Anchor peer.

El anchor peer es el encargado de, mediante Gossip (Chisme o susurros en español), asegurarse que los nodos de las diferentes organizaciones se conozcan entre sí. Éste es el encargado de conocer todos los nodos del canal y permitir la comunicación entre los mismos.

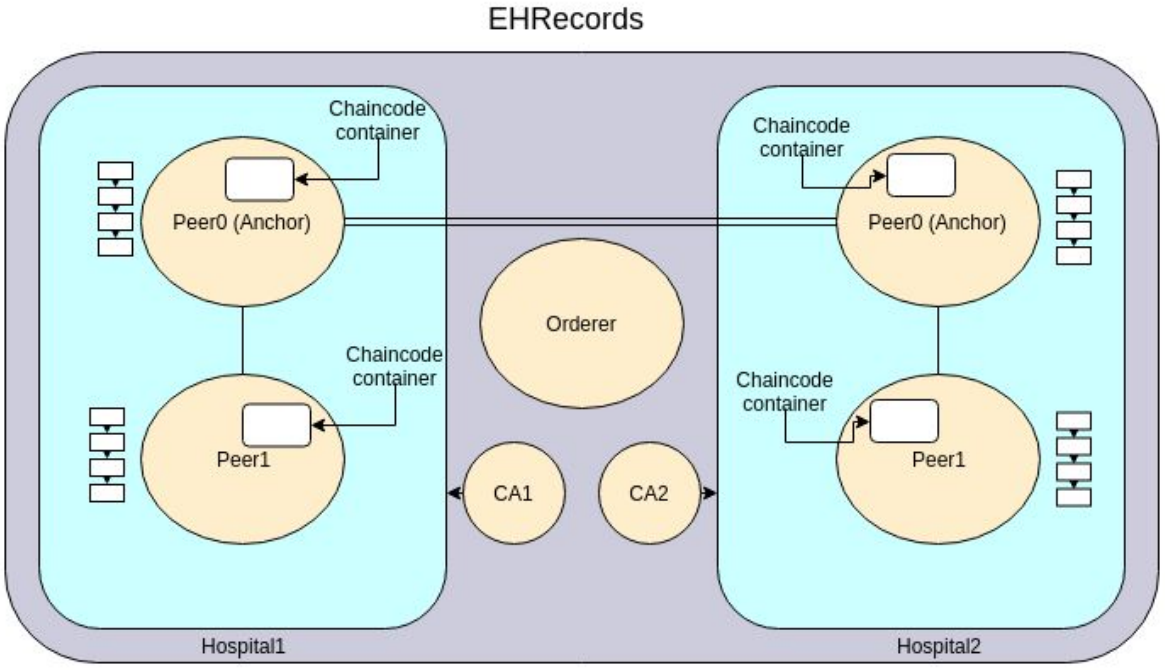


Imagen 25: Arquitectura de la red

Además de los peers pertenecientes a cada uno de los hospitales, es necesario tener Orderer Peers dentro de la red de Blockchain que se encarguen del servicio de ordenamiento. Como mencionamos en la sección previa, hay diferentes tipos de servicios y dependerá del servicio elegido la cantidad de nodos necesarios para llevarlo a cabo.

Para el desarrollo, lo conveniente es utilizar el servicio de ordenamiento SOLO, que puede llevarse a cabo con un único Orderer Peer. Sin embargo, más adelante se brindarán también detalles sobre la configuración necesaria para la utilización de Kafka y Raft, servicios óptimos para la etapa de producción.

Además de los peers para el ordenamiento, tenemos dos CA (Certificate Authorities), uno por cada Hospital. Estos se encargan de emitir los certificados para los distintos peers dentro de cada hospital (El que le corresponda a cada uno, por supuesto).

En cada uno de los peer se instanciará el mismo chaincode. De más está decir que todos los peers formarán parte del mismo canal de comunicación. Cuando se produce la instanciación del chaincode, Hyperledger dispone que se cree un container mediante docker con todo el ambiente necesario para que el mismo funcione correctamente. Luego con el blockchain inicializado en cada uno de los peer el sistema puede funcionar correctamente.

Estructura de Carpetas

Se adoptó una estructura de carpeta con el objetivo de poder modularizar correctamente las distintas partes de la aplicación, principalmente debido al gran tamaño de la misma.

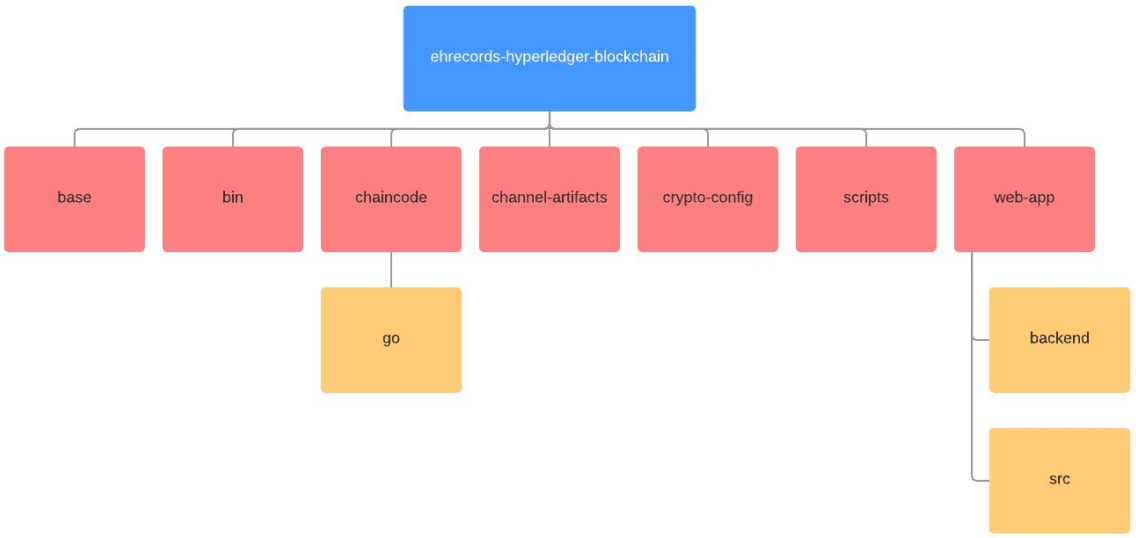


Imagen 26: Estructura de Carpetas

En el directorio raíz **ehrecords-hyperledger-blockchain** es posible encontrar, además de las carpetas que podemos observar en el diagrama, diversos archivos para la generación y puesta en marcha de la red.

El archivo **configtx.yaml** contiene cinco bloques y permite describir distintos aspectos de la red. En el bloque **Organizations** se describen las distintas organizaciones que forman parte, cada una con el ID del MSP y la definición de su anchor peer. En este caso tenemos tres organizaciones: Hospital1, Hospital2 y OrdererOrg. En el bloque **Capabilities** es posible seleccionar las configuraciones del Canal, Orderer y la Aplicación entre V1_1, V1_2 y V1_3 pero no entraremos en detalles en ésto. En el bloque **Application** se definen políticas de la aplicación. En este caso seguiremos las políticas por defecto y no entraremos en detalles. En el bloque **Orderer** se definen parámetros para el ordenamiento y por último en el bloque **Profiles** se definen distintos perfiles con parámetros que luego pueden llegar a ser utilizados para la generación de certificados.

Otro archivo de gran importancia y que es posible encontrar en el directorio raíz es **crypto-config.yaml**. Este archivo genera los cripto materiales - certificados y claves - para los peers y usuarios de la red. Estos se generan y se almacenan en la carpeta de nombre **crypto-config** que podemos observar en el diagrama.

Otro archivo es necesario para la generación de la red y este es **docker-compose-e2e.yaml**. Este archivo es el encargado de generar los distintos contenedores donde se ejecutarán los

distintos servicios de nuestro sistema, las CA, los peers de la red, etc. En este especificaremos las imágenes para cada uno, la ubicación de los certificados y claves públicas y privadas, además que se mapean archivos desde el sistema al contenedor, entre ellos los cripto materiales correspondientes.

En este caso, debido a que muchos de los parámetros, como las claves públicas y privadas, se generarán dinámicamente, utilizaremos un archivo **docker-compose-e2e-template.yaml** con el que luego finalmente se generará el primer archivo mencionado.

Además, se manejarán diversas alternativas para la creación de la red de acuerdo al tipo de ordenamiento que deseemos, por lo que se proveerán también los archivos **docker-compose-kafka.yaml** y **docker-compose-etcdraft2.yaml**.

Finalmente hay un último archivo de suma importancia en este carpeta: **ehrNetwork.sh**. Este archivo es, ni más ni menos, un script de consola al cual ejecuta múltiples de tareas de la red: Generar el cripto material junto con el docker-compose, poner la red en marcha, o pararla. Más adelante se mencionan con mayor detalle las acciones que lleva a cabo este script.

A continuación se describen uno a uno el contenido de las carpetas que forman parte del directorio raíz del proyecto.

En el directorio **base** se encuentran los archivos **docker-compose-base.yaml** y **peer-base.yaml** que son utilizados para la correcta generación de los **docker-compose**.

En el directorio **bin** se hallan los binarios de varios comandos que se utilizan para la generación de la red: **configtxgen**, **configtxlator**, **cryptogen**, **discover**, **fabric-ca-client**, **idemixgen**, **orderer** y **peer**.

En el directorio **chaincode** se encuentra el smart contract de la aplicación, dentro de la carpeta **go**, que indica el lenguaje con el cual fue programado el mismo.

Dentro de **channel-artifacts** se encontrarán, una vez generada la red, el orderer genesis block (Primer bloque de la cadena) -**genesis.block**-, la transacción de configuración de canal -**channel.tx**-, y se actualizarán los anchor peers para cada una de las organizaciones -**Hospital1MSPanchors.tx** y **Hospital2MSPanchors.tx**-.

Como mencionamos anteriormente, en **crypto-config**, se encontrarán los cripto materiales generados.

En el directorio **scripts** se encuentran los archivos **script.sh** y **utils.sh**, que son utilizados junto con el ya mencionado **ehrNetwork.sh** para la realización de sus tareas.

Finalmente, en **web-app**, se hallan las aplicaciones: Tanto el backend como el frontend. En la carpeta **backend** se encuentra justamente el primero, mientras que en **src** tenemos los archivos del segundo.

Generación, inicio y detención de la red

Para estas tres tareas utilizaremos el script de consola **ehrNetwork.sh**. A continuación se menciona en detalle cómo se realiza cada una de éstas.

Para la generación de todos los componentes, se ejecuta el script junto con el parámetro **start**. Primero se generan los certificados, mediante la herramienta **cryptogen** y el archivo **crypto-config.yaml**. Luego se reemplazan las claves privadas generadas durante el paso anterior en **docker-compose-e2e-template.yaml** generando entonces el archivo **docker-compose-e2e.yaml**. Además, dentro del backend, siguiendo la misma estrategia, se utiliza un archivo **connection-template.yaml** para la generación de **connection.yaml**, que será luego utilizado para conectarse con la red blockchain. Finalmente, como tercer y último paso, se utiliza **configtxgen** para la generación del orderer genesis block, el channel configuration transaction y la actualización de los anchor peers. Para este último paso se utilizan los perfiles de configuración definidos en **configtx.yaml**, y de acuerdo al tipo de consenso elegido (Por defecto SOLO, pero puede pasarse como parámetro junto al flag **-o** la opción de utilizar kafka o raft.

Para comenzar la red, se utiliza el parámetro **up**. Si no se realizó la generación con anterioridad, se realiza automáticamente. Luego se generan los contenedores mediante el comando **docker-compose**, tomando los archivos correspondientes de acuerdo al tipo de consenso utilizado. Y finalmente ejecuta el **script.sh**, de la carpeta **scripts**, en un contenedor llamado **cli** en docker. Este contenedor se define en **docker-compose-e2e-template.yaml** y

tiene seteadas las variables de entorno del peer0 del hospital1. De esta manera, se simula la ejecución del script en este peer.

Lo que realiza **script.sh** es, importando el script **utils.sh** y utilizándolo como librería, crea un canal y une a todos los peers al mismo. Luego, actualiza los anchor peers con los cripto materiales generados, e instala el chaincode en todos los peers. Instancia el chaincode en peer0.hospital1, lo que automáticamente lo instancia -inicializa- en todo el canal. Por último, testea que funcione correctamente consultando el blockchain. Finalmente, la red se encuentra funcionando y lista para comenzar a trabajar.

Para detener la red se ejecuta nuevamente el script con el parámetro **down**. De este modo, se eliminan los contenedores y archivos generados para nuestra red.

A continuación se describen las distintas funciones del chaincode instalado.

Smart Contract

Lenguaje

Para la implementación del chaincode, Hyperledger fabric ofrece tres alternativas a la hora de programarlo: Java, node.js y Golang.

Golang [23] es la más madura de las tres, debido a que fue la primera disponible. Es posible encontrar gran cantidad de documentación al respecto y gran cantidad de recursos para aprender, además de soporte de parte de la comunidad en caso de encontrar cualquier tipo de inconveniente. La única desventaja es que esta opción es un poco más compleja que el resto.

Por otro lado Node.js permite realizar un desarrollo simple, con bastante documentación disponible para su aprendizaje. Además permite tener una stack completa en Javascript.

La tercera opción, Java, parece ser la menos tentadora de las tres. Si bien es un lenguaje muy popular, su implementación como lenguaje para la programación de smart contracts es relativamente nuevo, por lo que no posee gran cantidad de documentación, recursos de aprendizaje, o una comunidad de soporte amplia.

Se decidió por la utilización de Golang, no solo por la amplia cantidad de documentación, sino también ya que es un lenguaje que ha crecido mucho este último tiempo y despertó mucho interés en mí para aprenderlo.

Imagen 27: Logo de Golang

Golang es un lenguaje de programación de código abierto que se caracteriza por permitir la creación de software simple, confiable y eficiente. Está desarrollado por Google y su primer versión vio la luz en marzo del 2012, por lo que podríamos considerarlo como un lenguaje relativamente nuevo aunque maduro.

Es expresivo, conciso, claro y eficiente. Utiliza mecanismos de concurrencia para permitir al programador escribir código que aproveche de la mejor forma los múltiples núcleos de una computadora y los sistemas distribuidos, mediante una construcción de programas flexible y modular. Se compila rápidamente a código máquina e incluso goza de los beneficios del recolector de basura.

Es un lenguaje compilado rápido y estáticamente tipado, que se siente como un lenguaje interpretado, dinámicamente tipado.

Implementación

A la hora de diseñar el smart contract se pensaron en las distintas estructuras y funciones a implementar.

Como estructuras tenemos Registros médicos (**Record**) con Información, fecha y DoctorId - del doctor que agregó este registro -, y Pacientes (**Patients**), que poseen nombre, apellido, edad, dirección y una lista de registros médicos (Arreglo de records).

Por otro lado tenemos múltiples métodos para operar dentro del blockchain:

- **CreatePatient** crea un nuevo paciente y lo añade al blockchain, previamente corroborando que los parámetros sean correctos.
- **AddRecordToPatient** añade un nuevo registro a paciente. Para realizar esto, primero debe obtener al correspondiente paciente del blockchain, añadir el nuevo registro en el arreglo y finalmente actualizar al paciente en el blockchain.
- **GetPatient** permite obtener la información de un paciente a través de su id.
- **GetAllPatients** permite obtener la información de todos los pacientes que se encuentran almacenados en el blockchain.

Para mayor detalle, consultar el código del chaincode que se encuentra en **chaincode/go/chaincode.go**.

Backend

Funcionalidades

La intención detrás del desarrollo de un backend es lograr una interacción con el blockchain ágil y eficiente, autenticar a los usuarios para que los datos sean consultados por usuarios válidos y responder con objetos json que permitan al frontend trabajar con los mismos sin necesidad de modificarlos.

Las funcionalidades que ofrece al backend son las siguientes:

- Registro y enroll del administrador (Se utiliza únicamente al inicializar el sistema). En hyperledger la creación de usuarios se debe hacer en dos pasos, primero registrarlos y luego alistarlos al sistema.
- Implementación de Enroll de usuarios, permitiendo a doctores registrados en el blockchain loguearse en el sistema con su propio usuario y contraseña.
- Implementación de Login mediante el uso de **JSON Web Tokens**.
- Validación de la identidad del usuario mediante el uso de **Wallets**.

- Interacción con el blockchain para poder realizar todas las operaciones que ofrece el chaincode: Obtener los datos de un paciente mediante su id, obtener datos de todos los pacientes, añadir un nuevo paciente, o añadir un nuevo registro a un paciente en particular indicando su id. Para realizar ésto se utilizará el SDK para Node.js de Hyperledger Fabric, que permite interactuar con el blockchain mediante funciones provistas por diversas librerías. A continuación se explica con mayor detalle.

Es importante aclarar que la aplicación en sí (Backend y Frontend) corre en un servidor dedicado de un cierto hospital, que tiene comunicación con algún peer de la red blockchain.

En este caso, se decidió que la aplicación a realizar sería para el Hospital 1 - aunque la misma puede replicarse para el hospital 2 con facilidad.

En el siguiente diagrama de flujo (Imagen 28) se representa el funcionamiento básico de la API para que el mismo quede más claro.

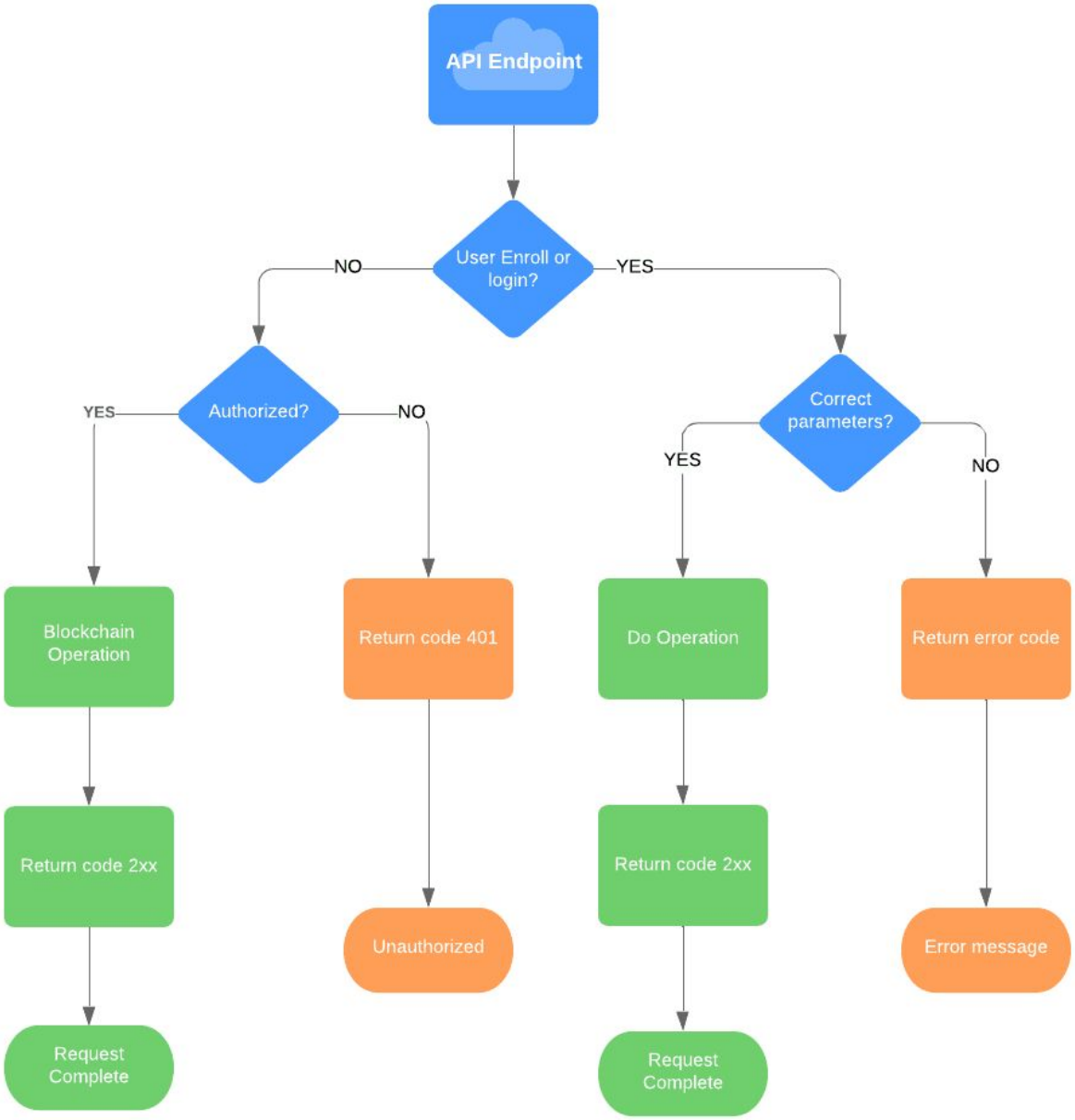


Imagen 28: Diagrama de flujo de un request API

Se observa que si no se trata de una operación de enroll o login, se trata de una operación de blockchain, por lo tanto la autenticación es necesaria. En caso de que el usuario no esté autenticado se rechaza la operación y se devuelve un código de error.

Para el manejo de las diferentes operaciones se usaron diferentes rutas, de forma tal que es posible llamar diferentes operaciones únicamente consultando en determinados endpoints.

Estructura

Para la estructura del backend en node.js y express.js utilizamos rutas, controladores, servicios y - en menor medida - modelos (Ver imagen 29).

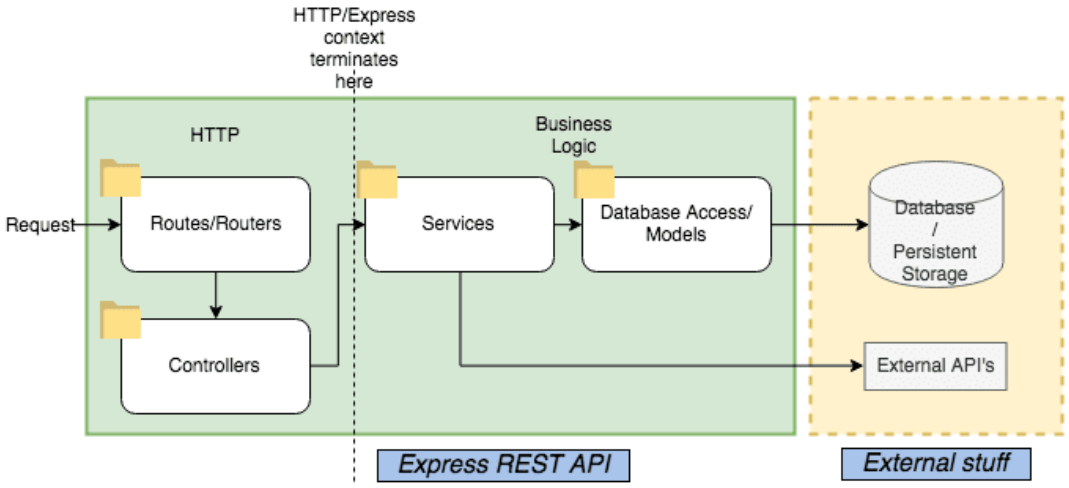


Imagen 29: Estructura del Backend. Fuente: https://www.coreycleary.me/project-structure-for-an-express-rest-api-when-there-is-no-standard-way/

Como se observa en la imagen, las rutas permiten mapear las distintas URL a los controladores, los cuales a su vez manejan la lógica detrás de los parámetros y llaman a los distintos servicios y deciden qué hacer con la respuesta obtenida. Los servicios son los encargados de realizar la mayor parte del trabajo, consultar APIs externas o utilizar los modelos para acceder a la base de datos, los cuales simplifican el acceso a la misma.

Asincronía en Javascript

Cabe aclarar que el control de la asincronía es una parte más que importante dentro de Javascript [24], ya que hay gran cantidad de funciones asincrónicas que notifican el resultado más tarde, mientras la función principal continúa su flujo de operación. Eso surge debido a que Javascript fue originalmente diseñado para ejecutar pequeñas interacciones con el usuario o peticiones en red, manteniendo una interfaz fluida.

Cuando se produce una llamada asincrónica no-bloqueantes, que son las que habitan en el mundo de Javascript, la ejecución del programa continúa normalmente sin esperar una respuesta. Más tarde, cuando la operación de la llamada se ha completado, se envía una notificación y la función que procesa la respuesta (callback) se encola para ser ejecutada más tarde.

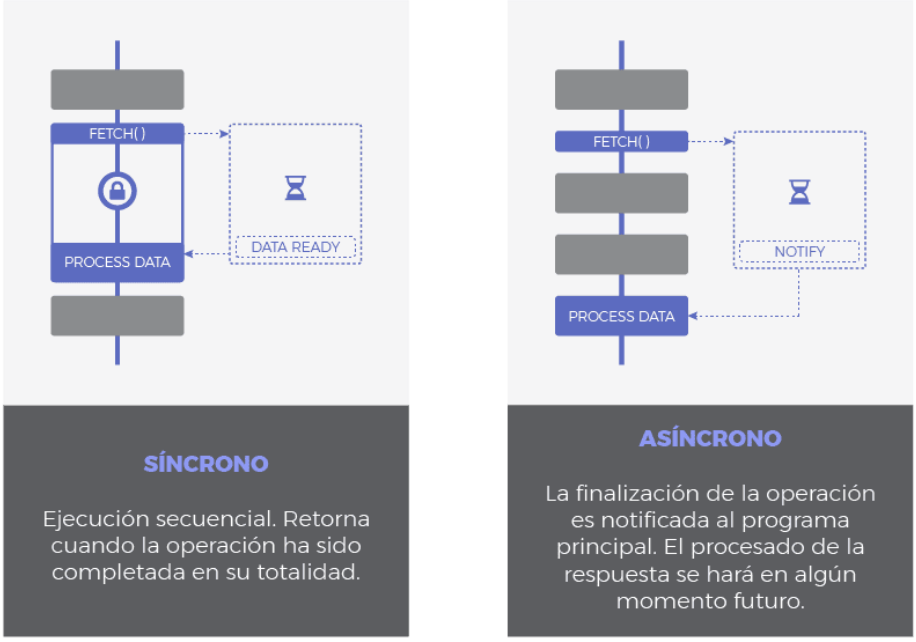


Imagen 30: Síncrono vs. Asíncrono

Como se mencionó, Javascript emplea un modelo asincrónico y no bloqueante, con un loop de eventos implementado por un único thread. Este loop de eventos, encola los callbacks a medida que las distintas funciones terminan su ejecución asincrónica. Cuando no haya más funciones que ejecutar en el call stack, entonces comenzarán a procesarse las callbacks del loop de eventos.

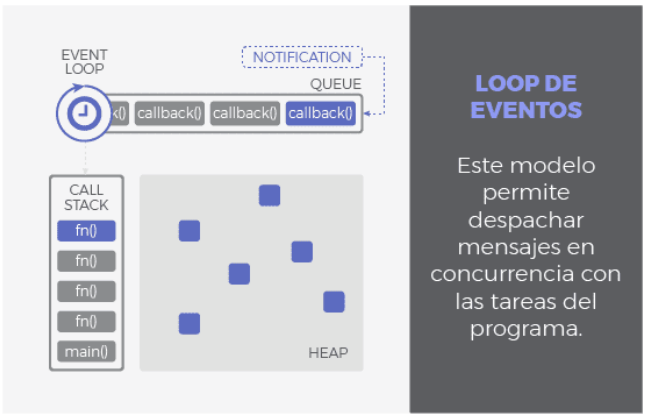


Imagen 31: Loop de Eventos de Javascript

Javascript entonces, propone diferentes opciones para lidiar con esta asincronicidad y las mencionadas callbacks: **Promesas** y **Async/Await**.

Las promesas son objetos que representan el resultado de una operación asincrónica, tanto la correcta terminación como un eventual fracaso, y pueden estar disponibles ahora o en el futuro. Por ejemplo la función `then` se ejecuta sobre una promesa y recibe dos argumentos, una función callback a ejecutar en caso de éxito, y otra en caso de fracaso, de esta manera podemos manejar la asincronicidad de forma mucho más sencilla.

Ejemplo:

```
promise.then(function(result) {
    console.log(result); //Funcionó!
}, function(err) {
    console.log(err); //Error :(
});
```

Por otro lado, la alternativa es utilizar las palabras clave Async/Await. Surgió para simplificar el uso de promesas: Mientras que la etiqueta async declara una función como asincrónica e indica que una promesa será devuelta, await puede ser utilizado dentro de una función declarada como async para esperar automáticamente que la promesa se resuelva - de forma no bloqueante.

Ejemplo:

```
function delay() {
      return new Promise(resolve => {
            setTimeout(() => {
                  resolve('Hubo un poco de delay...');
            }, 1000);
      });
}
async function main() {
      const result = await delay();
      console.log(result);
}
```

Como se puede observar, se espera a la resolución de la promesa y luego continúa la ejecución del programa. Ésta es una técnica muy efectiva para el manejo de la asincronicidad en Javascript.

Fabric Node SDK

El SDK para Node.js de Hyperledger Fabric [25] provee una poderosa API para interactuar con el blockchain de Fabric de diferentes maneras, entre ellas:

1. Crear canales
2. Consultar a los nodos para unirse a estos canales
3. Instalar chaincodes
4. Instanciar chaincode en canales
5. Invocar transacciones
6. Consultar el inventario (ledger)
7. Registrar usuarios
8. Alistar usuarios
9. Revocar permisos

En este caso utilizaremos el SDK para los puntos 5, 6, 7, 8 y 9.

El SDK contiene tres módulos principales:

- **fabric-network:** Este módulo nos permite invocar transacciones y consultar el inventario.
- **fabric-client:** Provee APIs para interactuar con los componentes de una red blockchain de Hyperledger Fabric, por ejemplo los peer, orderers, etc.
- **fabric-ca-client:** Provee APIs para interactuar con el componente fabric-ca, que se encarga del servicio de manejo de membresías (membership management). De esta manera es posible registrar y alistar usuarios - enrollment -, o incluso revocar sus permisos.

Wallet

La utilización de una wallet por parte del servidor es una de las partes vitales de la aplicación. Ésta contiene un conjunto de identidades de usuario, que son luego utilizadas para acceder a los distintos recursos y operar en la red del blockchain [26].

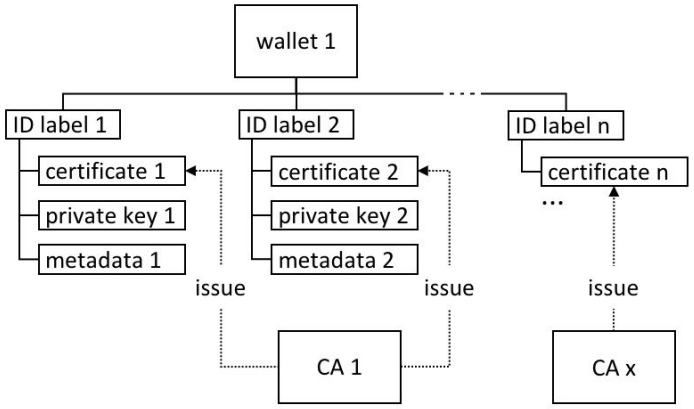


Imagen 32: Ejemplo de la estructura de una Wallet

Una wallet puede contener múltiples identidades, cada una emitida por una CA particular. A su vez, cada una de esas identidades está compuesta por los siguientes elementos: Un certificado X.509 conteniendo una clave pública, una clave privada, y metadata específica de Hyperledger Fabric.

A su vez, diferentes tipos de wallets pueden utilizarse, de acuerdo al lugar que se utilice para almacenar las identidades.

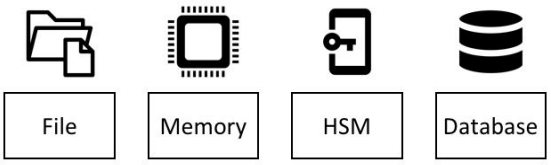


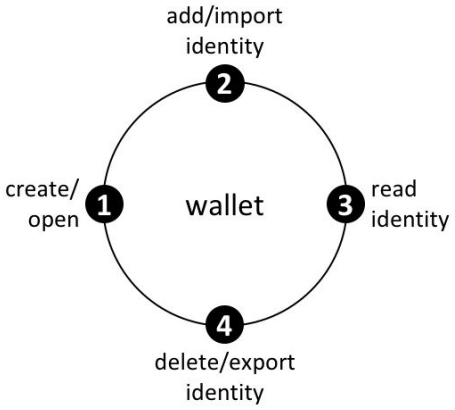
Imagen 33: Tipos de Wallets

Los cuatro tipos diferentes son:

- **Sistema de Archivos**: Es el más común y es el utilizado en este proyecto. Son sencillos de implementar y son una buena elección por defecto. Es importante aclarar que es necesario que estos sean protegidos para evitar que queden comprometidos. Se utiliza la clase **FileSystemWallet** de **fabric-network**.
- **En memoria:** Es útil para casos donde la aplicación corre en ambientes sin acceso a un sistema de archivos, normalmente un navegador. Este tipo de wallet es volátil, por lo que las identidades se perderán luego que la aplicación finalice o en caso de un cierre inesperado. Se utiliza la clase **InMemoryWallet**.
- **Hardware Security Module (HSM):** Es un mecanismo de alta seguridad que puede formar parte de la computadora o estar accesible en red para su utilización. Proveen la habilidad de realizar encriptaciones de las claves privadas de forma que las mismas nunca salgan realmente del HSM. Para su implementación se deben utilizar las clases **FileSystemWallet** junto con **HSMWalletMixin**.
- **CouchDB:** Es la forma más extraña pero es útil para aquellos usuarios que prefieren gozar de los mecanismos de backup y restauración de una base de datos. Se utiliza la clase **CouchDBWallet**.

Es importante aclarar que todas los diferentes tipos son derivados de una clase común **Wallet** que provee un conjunto de APIs para el manejo de identidades. Ésto significa que es posible implementar la aplicación más allá del mecanismo utilizado.

El ciclo de vida de las diferentes Wallet es el mismo para todas: Pueden ser creadas o abiertas, luego dentro es posible agregar o importar distintas identidades, las cuales luego pueden ser leídas, eliminadas o exportadas.



Registro de Nuevos Usuarios

Primero que nada cabe mencionar que es necesario realizar un conjunto de operaciones antes de que sea posible el registro de usuarios.

Por defecto un administrador fue creado para cada CA con usuario “admin” y contraseña “adminpw”. Se requiere que este usuario administrador se comunique con el CA respecto a la organización en la que se encuentra - como se aclaró con anterioridad en este caso es el hospital 1 - y registre al usuario. Si bien es posible realizar esto ejecutando una consola en el contenedor correspondiente al CA del hospital 1 y utilizando el comando `fabric-ca-client register`, la idea fue proveer esta funcionalidad - aunque sea de forma básica - en el backend, para simplificar la operación.

Para que el administrador puede realizar esto, es necesario primero realizar un alistamiento de su identidad - a partir de ahora lo llamaremos enrollment - de administrador en la Wallet.

Para estas operaciones se proveen los endpoints **/admin** para el enroll de su identidad y **/admin/register/:id** para el registro de un nuevo usuario, donde **:id** es la identificación del mismo. Esta última operación devuelve como resultado un secreto, que el usuario podrá utilizar para registrarse al sistema y setear su propia contraseña. La forma en cómo se le

comunica el mismo al usuario va más allá de los objetivos de este proyecto, aunque se recomienda un medio de alta seguridad.

Para el enrollment del administrador se utilizó FabricCAServices de **fabric-ca-client** para comunicarse con la CA, X509WalletMixit de **fabric-network** para la creación de la identidad y FileSystemWallet para importar la identidad y asociarla al administrador.

Por el otro lado, para el registro del usuario es necesaria también la utilización de un Gateway también de **fabric-network** para la obtención del certificate authority a través de la identidad del administrador, y posteriormente el registro del nuevo usuario ante la CA. La diferencia con el anterior es que esta vez se realiza un registro de usuario en vez de un enrollment, por lo que la operación del CA que se llama es diferente.

Ambos métodos **enrollAdmin** y **registerUser** pueden encontrarse en el controlador **blockchain.user.controller** para mayor detalles.

Enroll de Usuario y Base de Datos

El usuario mismo es el encargado de alistarse en el sistema proveyendo el secreto y una contraseña para su sesión.

El enrollment se realiza en el método **enrollUser** en el controlador **blockchain.user.controller** y no varía mucho del método **enrollAdmin**. Una vez, en caso de que la ejecución del enrollment haya sido exitosa, es necesario añadir al usuario con su respectiva contraseña en la base de datos de MongoDB.

Para conectar la base de datos con el backend se utilizó la librería mongoose [27], la cual ofrece una interfaz muy sencilla para conectarse con la base de datos en dos sencillos pasos. Primero hay que importarla `const mongoose = require('mongoose');` y luego realizar la conexión mediante `mongoose.connect(DB_URI, OPTIONS)`.

La conexión a la base de datos se puede encontrar en **/databases/mongodb.js**.

Para añadir y obtener usuarios se implementó un modelo User utilizando los esquemas que brinda mongoose (**mongoose.Schema**).

```
const UserSchema = new mongoose.Schema({
    username: {
        type: String,
        required: true,
        unique: true
    },
    password: {
        type: String,
        required: true
    },
    type: {
        type: String,
        enum: ['doctor'],
        default: 'doctor',
        required: true
    }
});
```

Imagen 34: User Schema

El esquema posee, como puede observarse, 3 propiedades. El username o nombre de usuario, la contraseña (Que obviamente se almacena encriptada) y un tipo, que por el momento solo podrá ser “doctor” pero la idea es que en el futuro se pueda expandir para que el sistema pueda ser utilizado también por pacientes, los cuales puedan ver su propia información, otorgar permisos, etc.

Además de definir el esquema, se añadió una función que se ejecuta previamente a la creación de un nuevo Usuario y hashea la contraseña enviada. Utiliza la librería **bcrypt** [28], la cual implementa este algoritmo de hashing, ideal para contraseñas ya que itera una cantidad de veces produciendo un hash difícil de descifrar mediante ingeniería inversa (obtener la contraseña utilizando el hash e imitando un proceso inverso al del hashing).

```
UserSchema.pre('save', function(next) {
    // Check if document is new or a new password has been set
    if (this.isNew || this.isModified('password')) {
        // Saving reference to this because of changing scopes
        const document = this;
        bcrypt.hash(document.password, saltRounds, function(err, hashedPassword) {
            if (err) {
                next(err);
            }
            else {
                document.password = hashedPassword;
                next();
            }
        });
    } else {
      next();
    }
});
```

Imagen 35: Hashing de la contraseña antes de almacenarla

Además también se añadió una función para corroborar que el password enviado por el usuario es el correcto, comparando el hash del mismo con el hash almacenado en la base de datos.

```
UserSchema.methods.isCorrectPassword = async function(password){
    return await bcrypt.compare(password, this.password);
}
```

Imagen 36: Corroboración de la contraseña

Finalmente el Schema se exporta como un modelo (`module.exports = mongoose.model('User', UserSchema);`) para poder ser utilizado en otras partes de la aplicación. Es importante aclarar que gracias a esta funcionalidad de mongoose, la base de datos creará automáticamente una colección asociada a este Schema al añadir el primer usuario.

Autenticación y JSON Web Tokens

De cara a la autenticación de usuarios se decidió utilizar una librería de node.js llamada **jsonwebtoken** [29]. La misma corresponde a la implementación de un standard abierto (RFC 7519 [30]) que define una forma compacta y auto-contenida para transmitir información de

forma segura entre dos partes, firmada digitalmente, ya sea mediante la utilización de un secreto (HMAC algorithm) o un par de claves pública/privada (RSA o ECDSA).

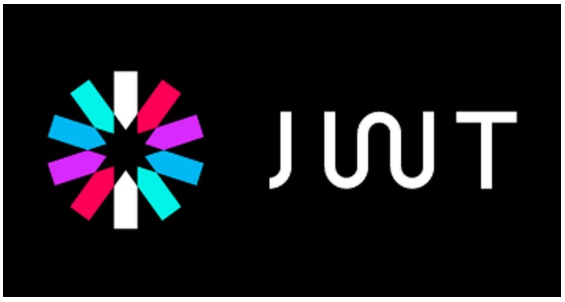


Imagen 37: Logo de jsonwebtoken

Un JWT se ve de la siguiente manera:

```
xxxxx.yyyyy.zzzzz
```

La primera parte es el header o cabecera, la cual es la codificación en Base64Url de un objeto json que indica el tipo de token junto con el algoritmo utilizado para firmar. Por ejemplo:

```
{ “alg”: “HS256”, “typ”: “JWT” }
```

La segunda parte es el payload como objeto JSON, también codificado en Base64Url.

Finalmente la tercera parte es la firma. Para esto debemos tomar las primeras dos partes codificadas, agregar un secreto y crear un hash de todo ésto.

De esta manera, conociendo el secreto es posible verificar que la información no fue alterada y el token es auténtico.

Uno de los escenarios más comunes donde se utiliza JWT es para Autorización. Es decir, una vez que el usuario se autentica y está logueado, cada request subsecuente incluirá el token, permitiendo el acceso a distintas partes de la aplicación. Esta es una excelente técnica ya que representa una sobrecarga muy pequeña y una gran facilidad para implementarlo entre diferentes dominios.

La autenticación se realiza mediante el endpoint **/api/user/login/**. Mediante un POST request con el nombre y contraseña del usuario como parámetros. Luego, se llama a la función login del controlador **users.controller**.

Dentro de esta función se corrobora con la base de datos que exista un usuario con ese nombre y contraseñas, en caso afirmativo entonces lo que se hace es generar un token, cuyo payload es el nombre del usuario y se firma con un secreto que es una variable de entorno. También se le puede agregar un tiempo de expiración. Finalmente se envía este token al usuario y éste lo puede utilizar en los próximos requests para que sean autorizados.

Autorización

La autorización se lleva a cabo mediante el uso de un middleware llamado **authenticated.middleware**.

Un middleware en express tiene acceso al objeto de solicitud (req) y puede ejecutar cualquier código previo a que la solicitud siga su camino, o incluso puede encargarse de finalizar el ciclo, por ejemplo rechazando la solicitud si determina que la misma no es válida.

El middleware de autenticación se ejecutará antes de cualquier llamada que requiere que usuario se encuentre logueado. El trabajo del mismo es corroborar que el token se encuentra embebido en la solicitud y que es válido.

En caso de que haya un error se envía una respuesta informando del mismo, y en caso de que el token sea válido se continúa el proceso de la solicitud llamando a la próxima función mediante `next()`.

Operaciones de Blockchain

Una vez autorizado, el usuario puede interactuar con el blockchain y llevar a cabo cualquiera de sus múltiples operaciones. Los endpoints para las mismas son:

- **GET /api/patients/** para obtener todos los pacientes
- **POST /api/patients/** para crear un nuevo paciente
- **GET /api/patients/:id** para obtener un paciente cuya identificación es id
- **POST /api/records/** para agregar un registro a un paciente

Si bien cada ruta redirige la consulta a un controlador y éste llama al respectivo servicio el cual se comunicará con el blockchain, es necesario que primero la consulta pase por otro middleware.

contract.middleware será el encargado de realizar la conexión con la red blockchain para obtener el contrato (Objeto **Contract** que puede ser utilizado para llamar las distintas funciones del mismo) y añadirlo como variable a **req** para poder ser accedido más tarde en el proceso.

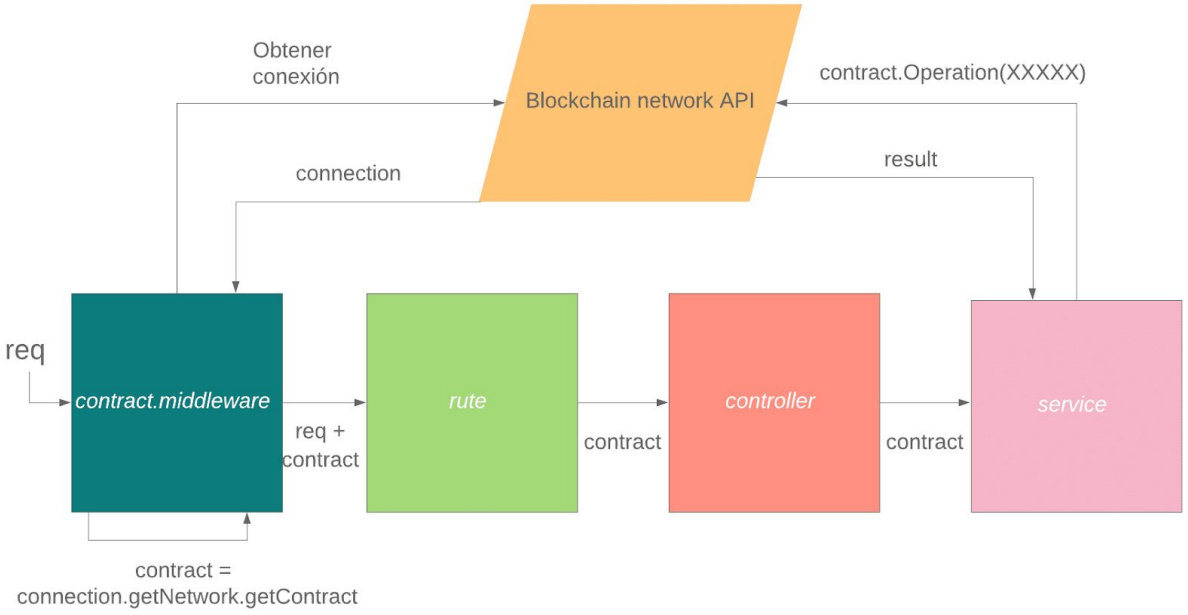


Imagen 38: Flujo de un request por una operación blockchain

Como se puede observar en el diagrama, el contrato pasa hasta el servicio, donde se llama la operación correspondiente del blockchain de acuerdo a la ruta a la que se envió la consulta.

Para las operaciones del blockchain el objeto **Contract** provee dos funciones:

- **evaluateTransaction(nombreOperacion, ...params)** para las consultas que no modifican el ledger, por ejemplo consultar por pacientes, donde **nombreOperacion** corresponde al nombre de la operación en el chaincode y **params** a todos los parámetros que necesita esta función.
- **submitTransaction(nombreOperación, ...params)** para las operaciones que modifican el ledger, por ejemplo agregar un nuevo paciente o un nuevo registro.

Una vez realizada las operaciones, se devuelven los objetos JSON correspondientes o, en caso de que no se haya podido ejecutar correctamente, el error pertinente.

Frontend

Funcionalidades

La principal función del Frontend es proveer al usuario de una interfaz simple y amigable mediante la cual sea posible interactuar con el Backend fácilmente.

La distintas pantallas que se pensaron son:

- **Inicial:** Breve descripción y portada de la aplicación.
- **Login:** En esta pantalla el usuario se loguea al sistema.
- **Registro:** El usuario se debe registrar mediante el uso del código secreto generado por el administrador.
- **About us:** Descripción un poco más amplia.
- **Home:** Aquí la aplicación funciona como una SPA (Single Page Application) ya que todas las interacciones se pueden hacer desde allí. La acciones disponibles son ver los pacientes, agregar un nuevo paciente, ver un paciente en detalle y añadirle un registro. A la hora de mostrar todos los pacientes es posible filtrar por nombre y apellido de los mismos.

También es importante mencionar que a la hora de idear la aplicación se consideró que la misma fuese responsive, para permitir su correcta navegación desde diferentes dispositivos, como tablets, celulares, notebooks, etc. Además, se pensó en evitar la utilización de pop-ups, ya que los mismos hacen que la navegación por la página no sea tan fluida para el usuario.

En el siguiente diagrama se puede observar cómo ocurre la navegación entre las diferentes pantallas.

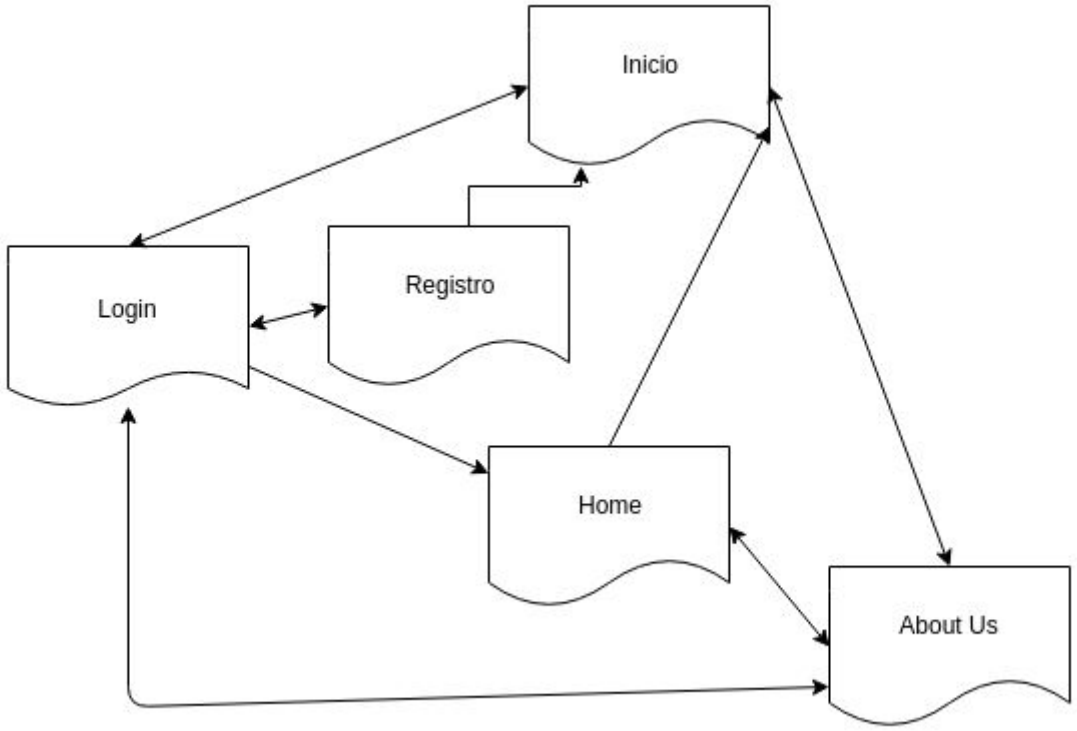


Imagen 39: Conexiones entre las distintas pantallas.

Estructura

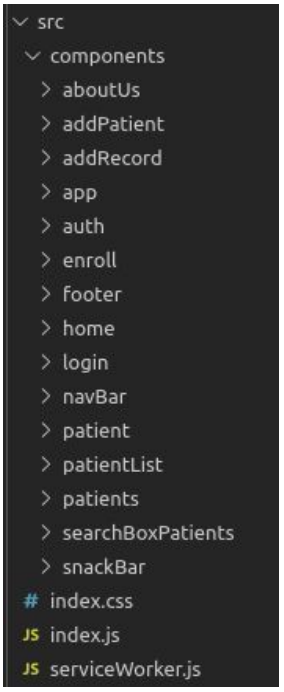


Imagen 40: Estructura de la Aplicación ReactJS

La estructura de la aplicación es más bien sencilla, gracias a **ReactJS**. Si bien hay un **index.js**, el resto de los elementos son componentes y se encuentran en directorios dentro de **/src/components**.

Los componentes permiten modularizar la interfaz en piezas independientes y reutilizables, permitiendo pensar en las mismas de forma aislada, logrando obtener una aplicación con alta cohesión (Cada módulo guarda una alta relación entre sus funcionalidades) y bajo acoplamiento (Mínima dependencia entre los módulos).

Pantallas

La pantalla de Inicio muestra únicamente una descripción breve de la aplicación y mediante el componente navBar (La barra de navegación, la cual se visualiza desde cualquier página del sitio) es posible acceder a las distintas secciones.

Imagen 41: Pantalla Inicio

En la sección **About Us** la descripción es un poco más amplia. Se puede leer una breve introducción sobre el Blockchain, la salud digital y este proyecto.

Imagen 41: Pantalla About Us

En la sección **login** el usuario (doctor) puede loguearse en el sistema mediante el usuario y contraseña. Si no se asignó una contraseña aún, debe primero realizar el **enrollment**, utilizando la contraseña de único uso que le fue entregada por el administrador del sistema.

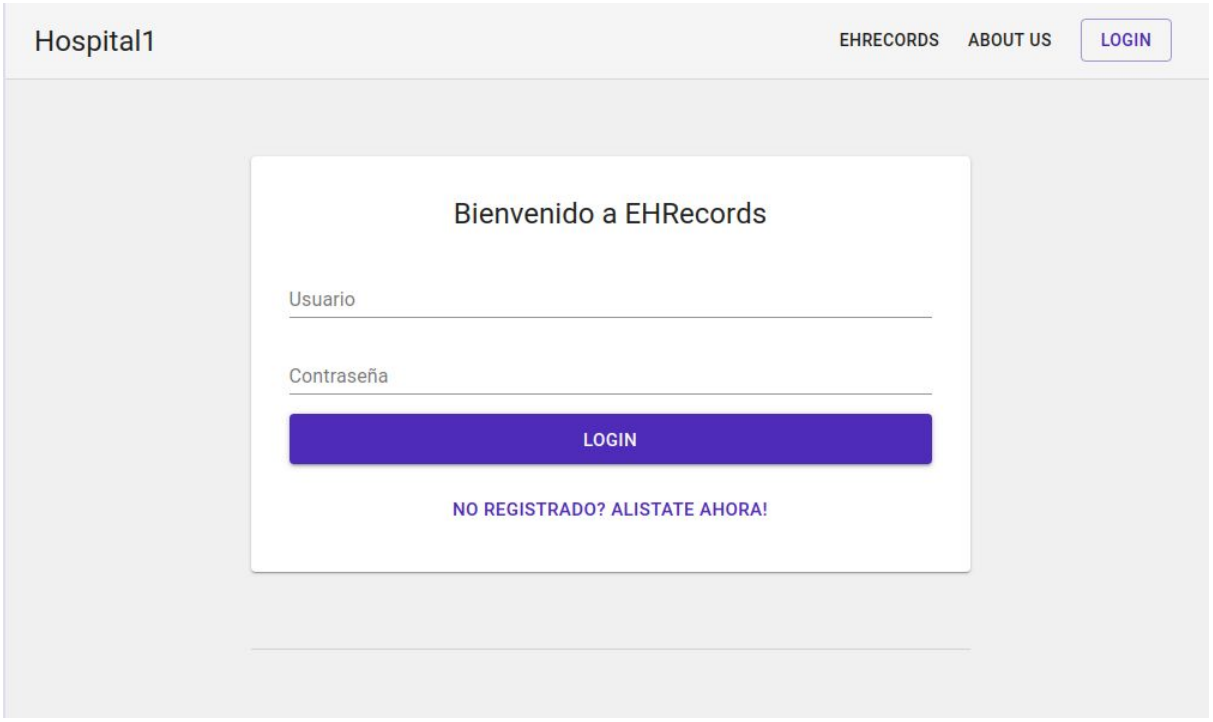


Imagen 42: Pantalla Login

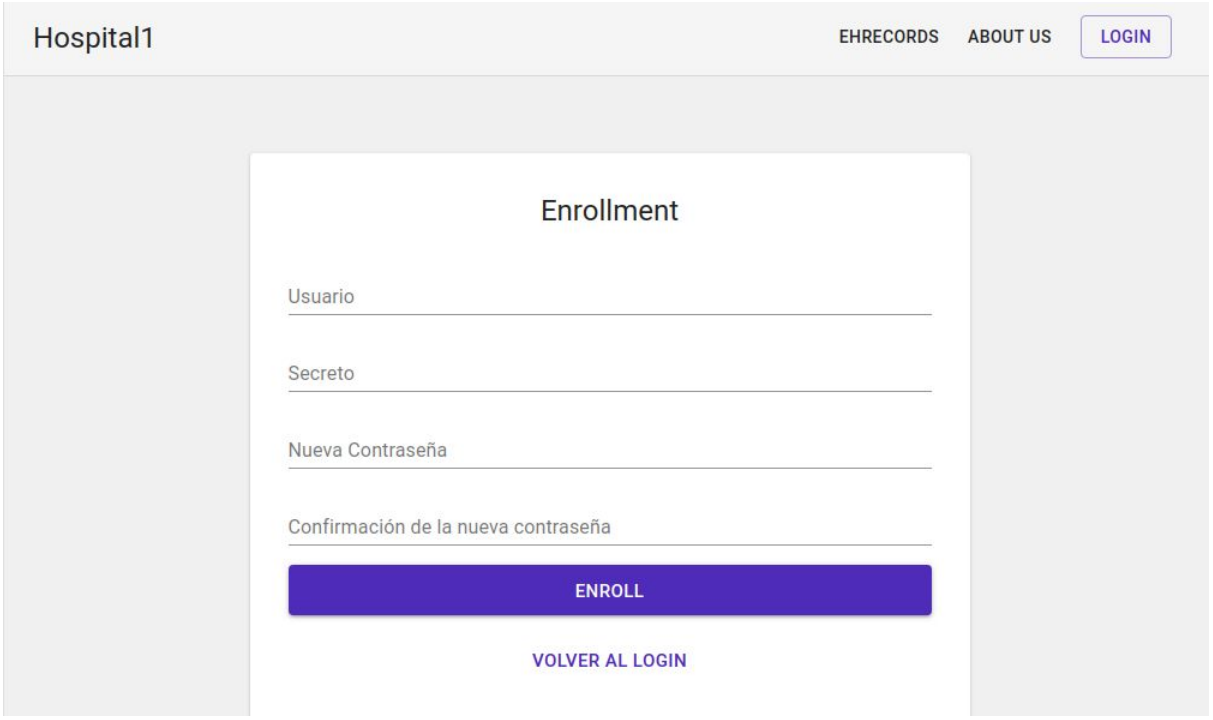


Imagen 43: Pantalla Enrollment

Para los errores se utilizaron Snack Bars de color rojo.

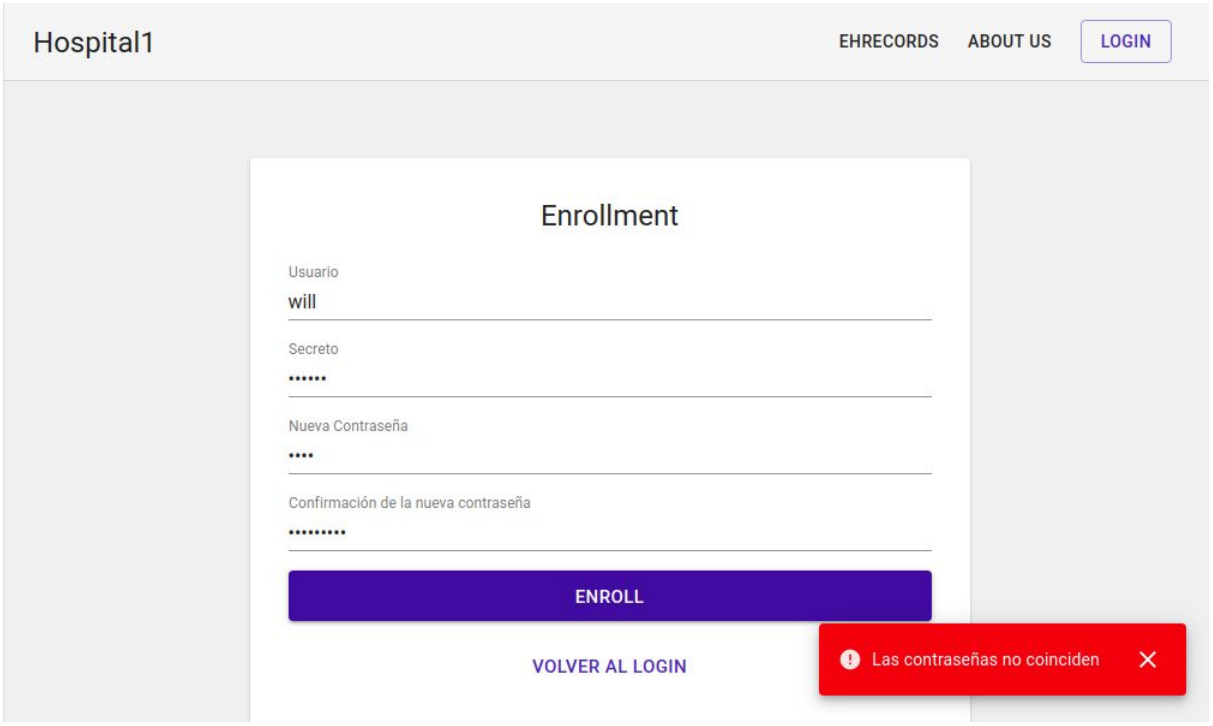


Imagen 44: Snackbar error

Una vez dentro del sistema, en el **home** es posible observar los pacientes actuales del sistema e incluso agregar nuevos pacientes.

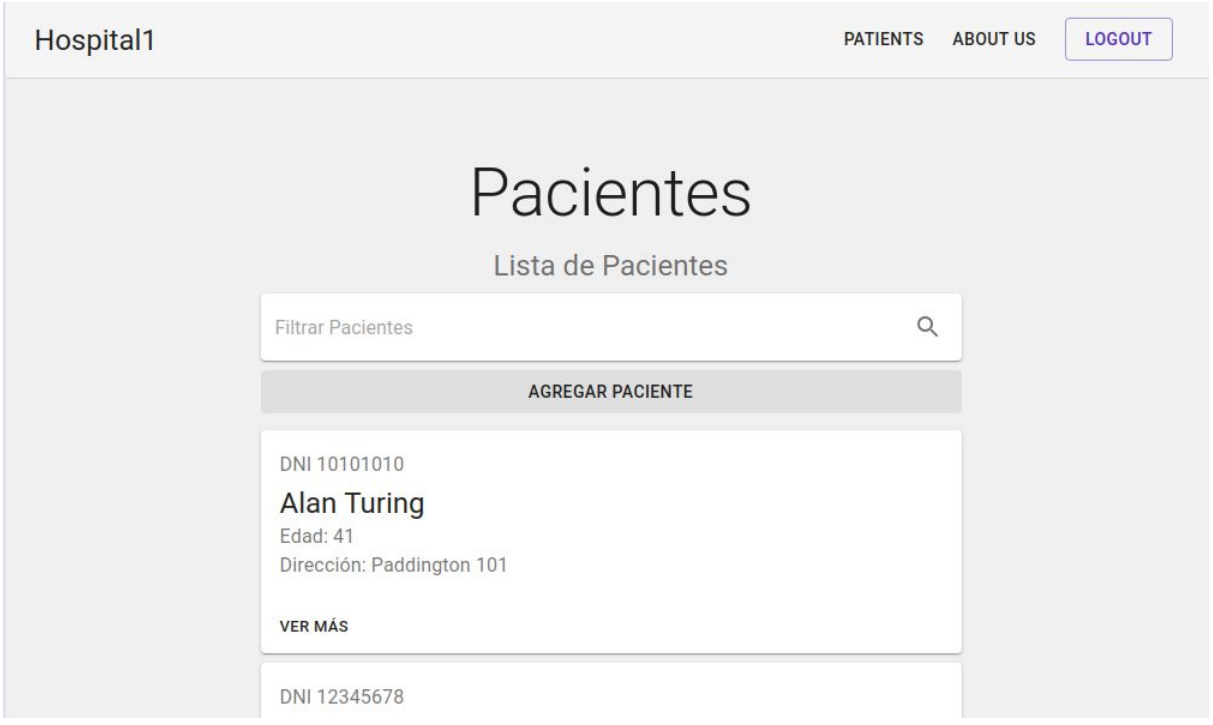


Imagen 45: Pantalla Home

Imagen 46: Pantalla Home, agregar nuevo paciente

Imagen 47: Pantalla Home, agregar nuevo paciente y sus datos

Al expandir la vista sobre un paciente también es posible ver sus registros y añadir nuevos.

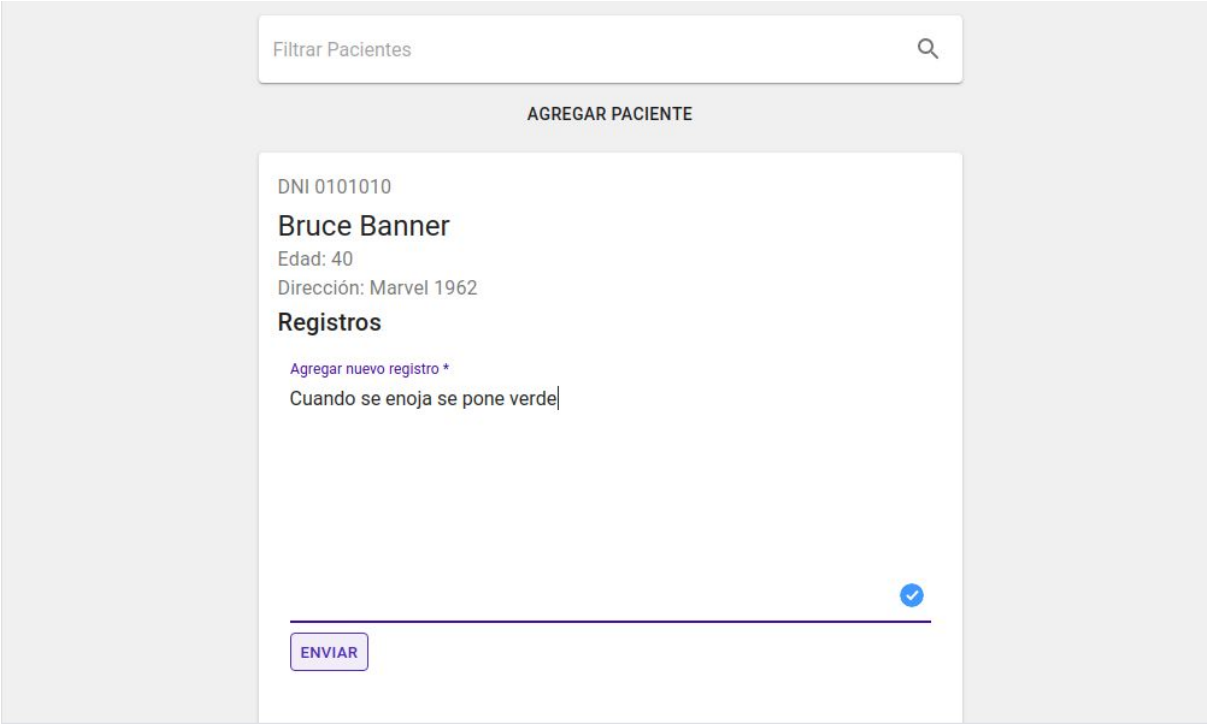


Imagen 48: Pantalla Home, añadiendo registro a un paciente

Se puede observar que se utilizaron Snack Bars verdes para informar que una operación fue realizada exitosamente.

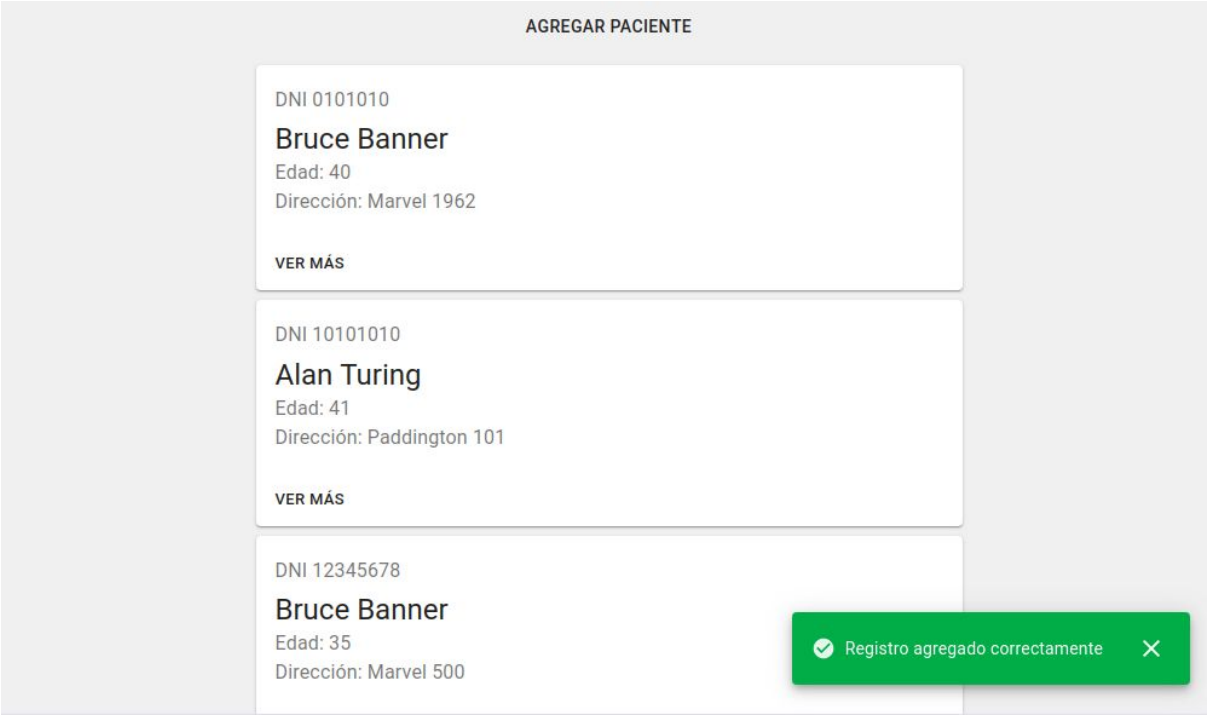


Imagen 49: Pantalla Home, registro agregado correctamente

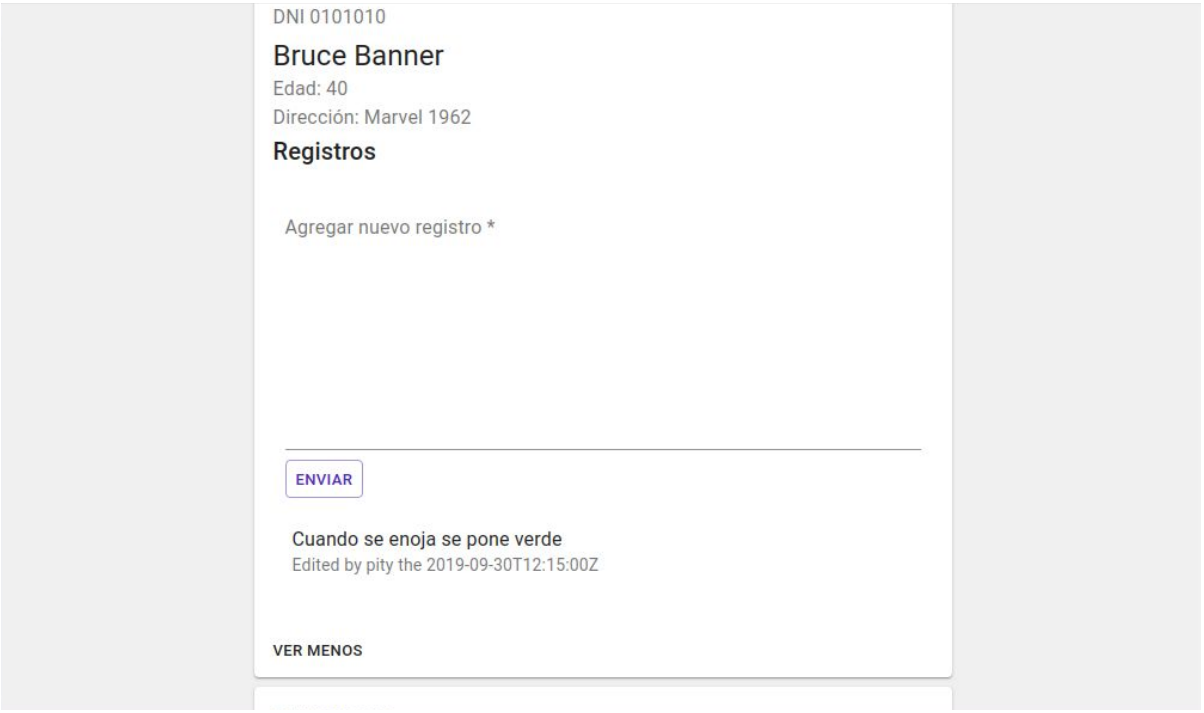


Imagen 50: Pantalla Home, se muestran los registros

El uso del filtrado se debió principalmente a que, imaginando la red a mayor escala, se tendrá una gran cantidad de pacientes, por lo que es necesario contar con una herramienta como ésta para lograr que la búsqueda de un paciente en particular se pueda realizar de forma mucho más rápida.

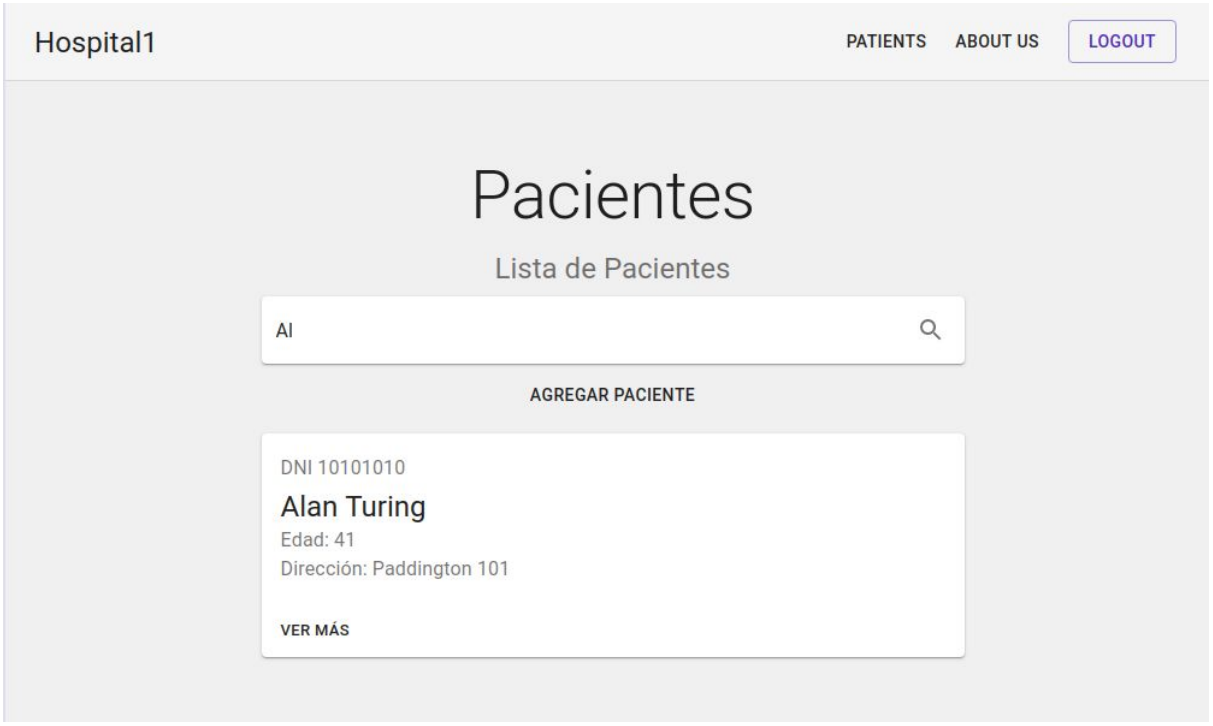


Imagen 51: Pantalla Home, filtrando pacientes

Axios

Es preciso mencionar todas las herramientas utilizadas para poder llevar a cabo la realización de este Frontend.

Además del framework ya mencionado para la interfaz gráfica y los componentes estilizados Material-UI, también se utilizó la librería **axios** [31] para realizar la comunicación entre el Frontend y el Backend.

Esta librería permite realizar con facilidad solicitudes HTTP (GET, POST, etc) a distintos endpoint y con gran facilidad. De esta forma fue posible integrar ambas aplicaciones sin ningun problema.

En la Nube

La nube se ha ganado un lugar muy importante dentro de la computación en los últimos años. También conocida como **cloud computing**, es un paradigma que nos ofrece servicios de computación a través de una red, normalmente el internet.

Es un nuevo modelo de prestación de servicios que ha ido avanzando a gran medida y permiten al usuario acceder a un conjunto de servicios, respondiendo con los mismos a las necesidades de su negocio de una manera flexible y adaptativa. Es decir, pagando únicamente por el servicio utilizado, lo cual genera beneficios tanto para los usuarios, los cuales ahorran costos salariales o costes de inversión económica en infraestructura, especialización, etc.

Las bases de la computación en la nube son la infraestructura dinámica, un alto grado de automatización, la capacidad de adaptación a las demandas variables y la virtualización avanzada, además del precio flexible ya mencionado.

Dentro de esta metodología es posible encontrar tres modalidades: El software como servicio (SaaS), donde las aplicaciones se encuentran alojadas en una red propia de la empresa y se ofrecen al usuario a través de una red, como el internet. Plataforma como servicio (PaaS) donde se le ofrece al usuario la posibilidad de contar con Sistemas Operativos y servicios asociados a través de internet. Y finalmente Infraestructura como servicio (IaaS), que permite al usuario contar con equipos para realizar sus operaciones, hardware, almacenamiento, servidores, etc.

A continuación, se mencionan dos tecnologías que se investigaron en pos de mover la red Blockchain de Hyperledger Fabric a una plataforma en la nube: Amazon Web Services Blockchain Templates y IBM Blockchain. Ambas plataformas dan soporte para este tipo de redes e incluso facilitan su desarrollo e implementación. Por este motivo se consideró importante mínimamente investigar estas tecnologías incluso sin llegar a implementarlas, debido a que se corresponden a tecnologías pagas.

Amazon Web Services Blockchain Templates

Imagen 52: Amazon Web Services y Blockchain Templates

Amazon Web Services (AWS) [32] es una plataforma de servicios en la nube. La cantidad de servicios que ofrece es enorme y continúa en aumento, lo que la hace una de las plataformas más tentadoras.

Entre los múltiples servicios que ofrece se encuentran AWS Lambda, que permite la ejecución de código que responde a eventos sin administrar servidores, pagando únicamente por el tiempo de utilización durante la ejecución. También AWS EC2, la cual permite alquilar máquinas virtuales en los que es posible alojar aplicaciones propias, transfiriendo la misma a hardware de mayores o menores capacidades de acuerdo a la demanda de estas aplicaciones. Otro servicio importante es Amazon Elastic Container Service, mediante el cual es posible gestionar contenedores docker simplemente corriendo aplicaciones compatibles. Además ofrece también almacenamiento, bases de datos, herramientas para IoT, etc.

El servicio que es de interés para este proyecto es el de Blockchain Templates o Plantillas de Blockchain [33]. Éste provee una forma rápida y sencilla de crear y deployar redes de blockchain seguras utilizando populares frameworks de código abierto como Ethereum y Hyperledger Fabric. Estas plantillas permiten concentrarse en construir la aplicación blockchain sin prestar tanta atención a la construcción de la red, el cual se realiza de forma mucho más automatizada en la plataforma de AWS.

Para lograr esto AWS Blockchain Templates realiza un deploy del framework de blockchain elegido como contenedores en un cluster ECS (Elastic Container Service) o directamente en una instancia EC2 corriendo docker. La red de blockchain es creada en el propio Amazon VPC (Red virtual de amazon), por lo que es posible subdividir la red (VPC Subnets), controlar permisos, etc.

Es importante mencionar que AWS no cobra por el uso de las plantillas de blockchain, sino por los recursos utilizados (ECS y EC2).

IBM Blockchain

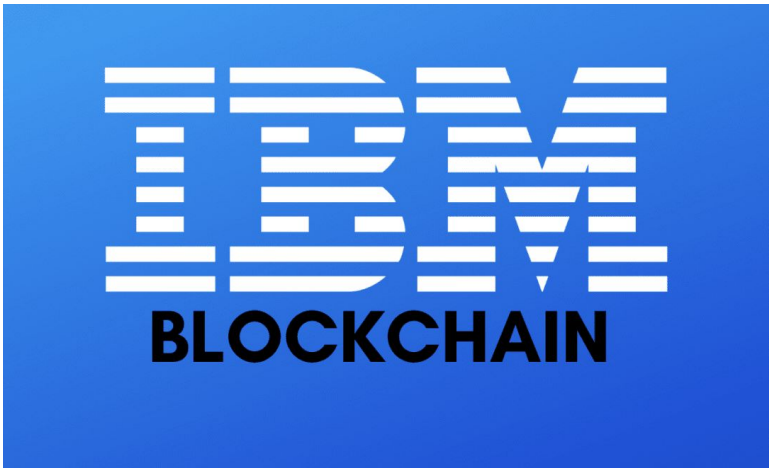


Imagen 53: Logo de IBM Blockchain

Por otro lado tenemos a la plataforma de justamente uno de los creadores de Hyperledger Fabric. IBM Blockchain [34] permite crear las redes de forma mucho más rápida y fácil. Fue creada específicamente para Hyperledger Fabric por lo que en ese sentido es mucho más específica que las plantilla de AWS.

La plataforma ofrece la posibilidad de implementar los componentes del blockchain necesarios (Peers, Orderers, CAs, etc), gestionarlos a través de una única consola y controlar de forma sencilla las identidades, el ledger y los smart contracts.

Para los desarrolladores ofrece una integración continua entre el desarrollo del chaincode y la gestión de la red gracias a una extensión de Visual Studio llamada justamente IBM

Blockchain. De esta manera el proceso es muy integrado y el salto desde desarrollo a producción se puede dar muy rápidamente.

Algo muy importante que ofrece este servicio es que es posible poner en producción los componentes de la red en los ambientes que se decidan, tanto IBM Cloud como nubes de terceros, por ejemplo AWS. De esta forma es posible segmentar la red en múltiples plataformas diferentes.

Entre las facilidades que ofrece la plataforma, esta permite administrar los peers junto a las CAs de forma muy simple desde un menú, observar información sobre los bloques e incluso registrar usuarios para las distintas organizaciones.

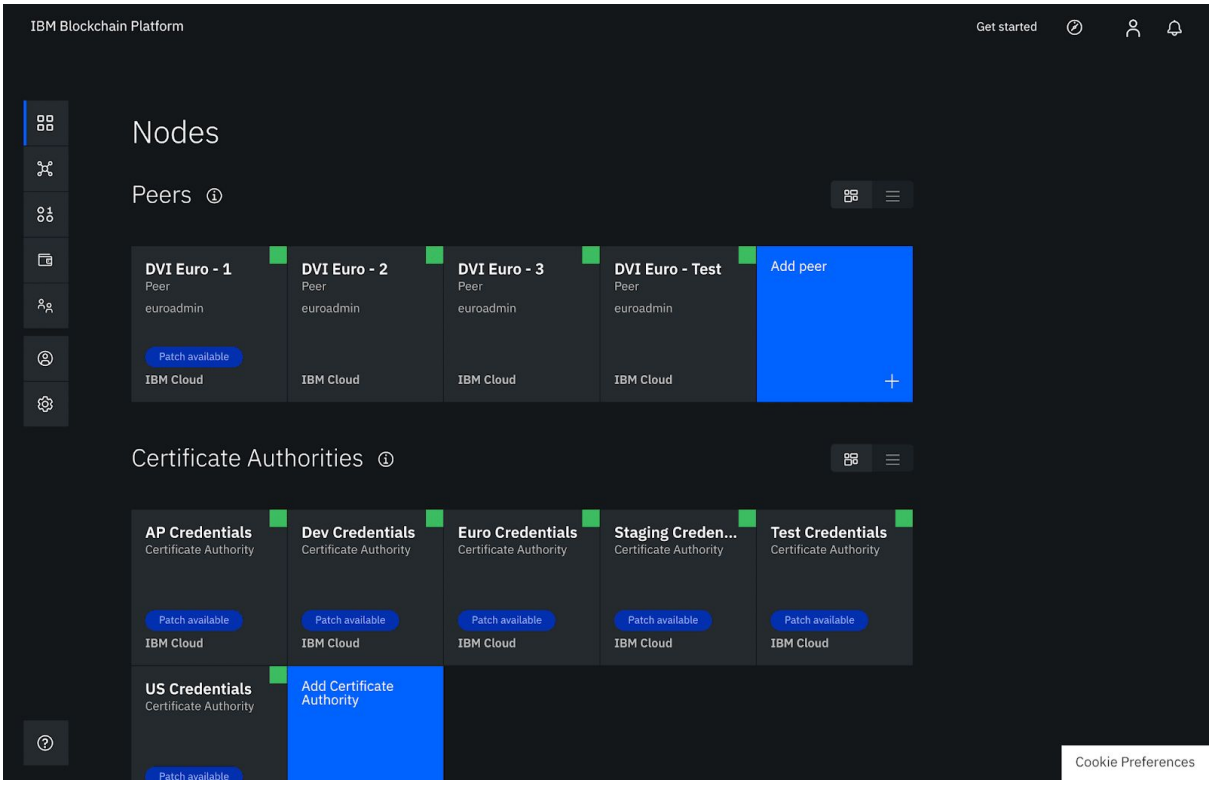


Imagen 54: IBM Blockchain Platform - Panel de Nodos de la Red

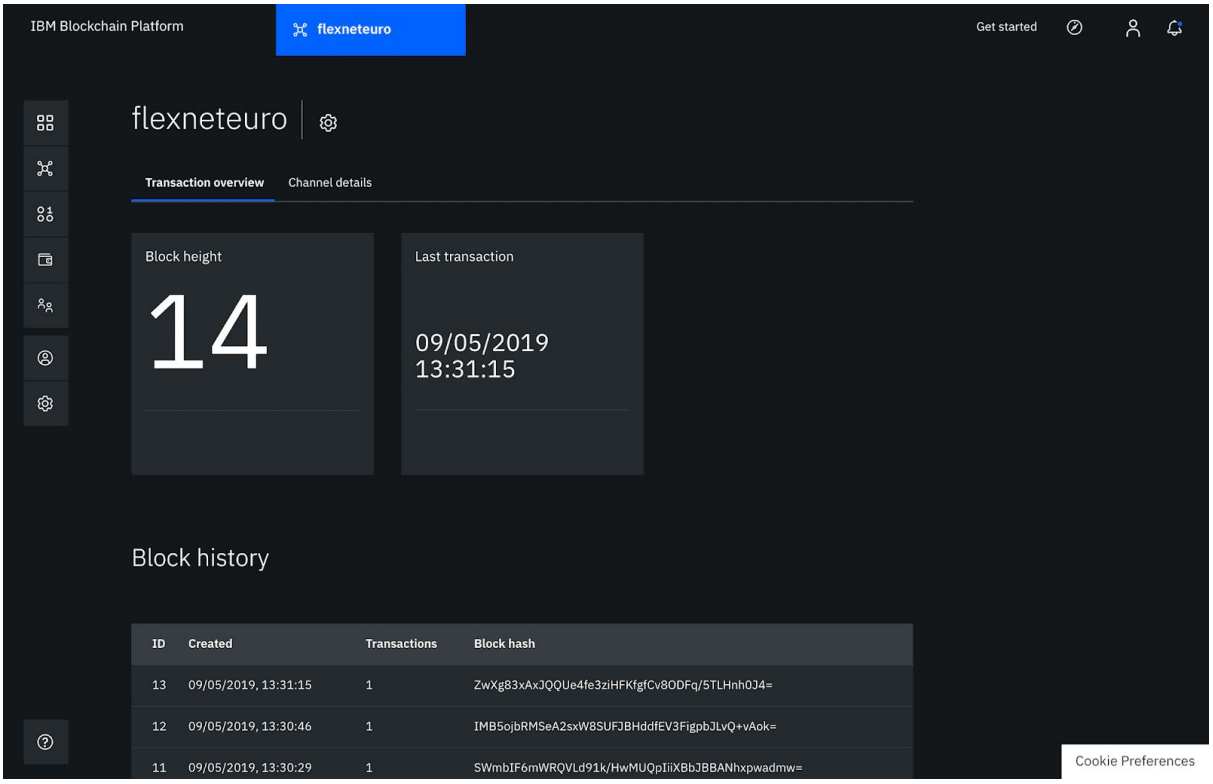


Imagen 55: IBM Blockchain Platform - Panel bloques de la red

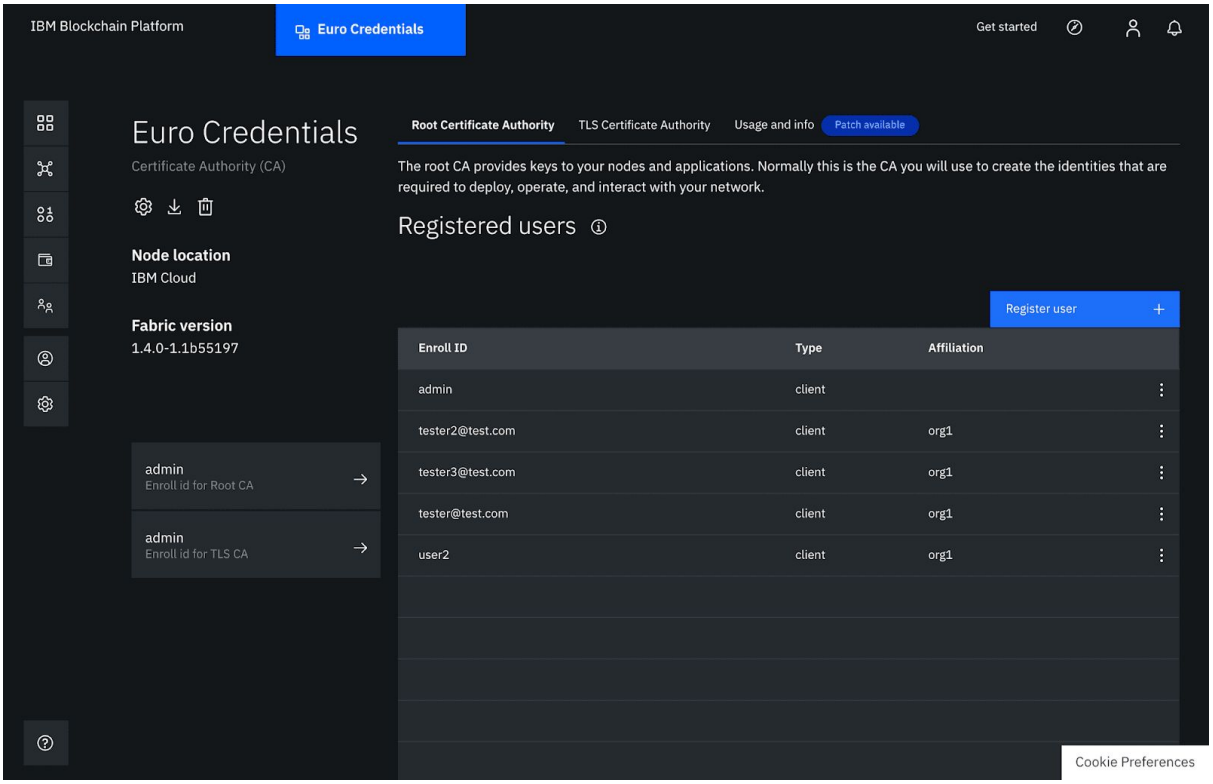


Imagen 56: IBM Blockchain Platform - Panel de Administrador de Usuarios Registrados

Conclusiones y Trabajo Futuro

Como conclusión puedo decir que la realización de este proyecto me dejó una grata satisfacción al haber logrado cumplir con todo lo propuesto. Sin conocimientos en el tema del Blockchain previo a comenzar el mismo, el diseño y desarrollo de un sistema completo, desde la red hasta las plataformas de acceso, parecía algo muy complejo y lejano. Sin embargo, gracias al apoyo docente y un constante deseo de seguir aprendiendo, puedo decir que he logrado finalizar el proyecto más grande de mi carrera y el cual me llena de orgullo.

Durante este proceso no solo he aplicado una gran cantidad de conceptos adquiridos durante mi formación como Ingeniero en Sistemas de Computación en la Universidad Nacional del Sur, sino también de conocimientos aprendidos en particular para la realización del mismo y que sin dudas me serán de enorme utilidad en el futuro que me depare esta profesión. Blockchain, contenedores Docker, Hyperledger Fabric, Golang, API Rest, ReactJS, por mencionar algunos de los temas que han sido motivo de investigación estos últimos meses.

El principal motivo por el que decidí encarar este sistema, además de un notable interés por la tecnología del Blockchain, moderna y uno de los temas del momento, fue porque realmente estoy convencido de que soluciona un problema real que nos afecta como individuos. Los sistemas digitales de salud, al menos en nuestro país, están bastante atrasados en modernidad y sin dudas hay mucho trabajo que puede ser realizado, desde el lado de la tecnología, para mejorarlos y simplificar la vida de muchas personas.

Si bien la realización del presente proyecto representa una simplificación de lo que sería un sistema aplicado en la realidad, con múltiples hospitales y muchos más nodos por hospital, corriendo en las facilidades de cada uno o en distintas nubes privadas, creo que es un sistema muy posible de implementar en la realidad si se tienen los recursos y se logran acuerdos entre los distintos hospitales.

Referencias

1. Wikipedia. *Blockchain*. https://en.wikipedia.org/wiki/Blockchain
2. Economía Simple. *¿Qué son las criptomonedas?* https://www.economiasimple.net/que-son-las-criptomonedas.html
3. Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.*
4. Blockchain Services. *Conoce los distintos tipos de Blockchain.* http://www.blockchainservices.es/novedades/conoce-los-diferentes-tipos-de-blockchain/
5. IBM Developer. *IBM Blockchain 101: Quick-start guide for developers.* https://developer.ibm.com/tutorials/cl-ibm-blockchain-101-quick-start-guide-for-developers-bluemix-trs/
6. Banco Santander. *Santander lanza en cuatro países el primer servicio de transferencias internacionales con blockchain.* https://www.santander.com/csgs/Satellite/CFWCSancomQP01/es_ES/Corporativo/Sala-de-comunicacion/Santander-Noticias/2018/04/12/Santander-lanza-en-cuatro-paises-el-primer-servicio-de-transferencias-internacionales-con-blockchain.html
7. Ethereum. *Sitio Oficial.* https://www.ethereum.org/
8. Solidity. *Sitio Oficial.* https://solidity.readthedocs.io/en/v0.5.11/
9. Hyperledger. *Sitio Oficial.* https://www.hyperledger.org
10. Hyperledger Fabric. *Sitio Oficial.* https://www.hyperledger.org/projects/fabric
11. Hyperledger Fabric Documentation. *Componentes de una red de Hyperledger Fabric.* https://hyperledger-fabric.readthedocs.io/en/release-1.2/network/network.html#components-of-a-network
12. Hyperledger Fabric Documentation. *Fabric Model.* https://hyperledger-fabric.readthedocs.io/en/release-1.4/fabric_model.html

13. Hyperledger Fabric Documentation. *Flujo de las transacciones.* https://hyperledger-fabric.readthedocs.io/en/release-1.4/txflow.html

14. Hyperledger Fabric Documentation. *Servicios de Ordenamiento.* https://hyperledger-fabric.readthedocs.io/en/release-1.4/orderer/ordering_service.html

15. Hyperledger Fabric Documentation. *Proveedor de membresía y crypto materiales.* https://hyperledger-fabric.readthedocs.io/en/latest/msp.html

16. Hyperledger Fabric Documentation. *Membresía.* https://hyperledger-fabric.readthedocs.io/en/release-1.4/membership/membership.html

17. Docker. *Sitio Oficial.* https://docs.docker.com/get-started/

18. Node.js. *Sitio Oficial.* https://nodejs.org/es/about/

19. Express. *Sitio Oficial.* https://expressjs.com

20. ReactJS. *Sitio Oficial.* https://es.reactjs.org/

21. Material UI. *Sitio Oficial.* https://material-ui.com/

22. MongoDB. *Sitio Oficial.* https://www.mongodb.com/

23. Golang. *Sitio Oficial.* https://golang.org

24. Lemoncode. *Javascript Asíncrono.* https://lemoncode.net/lemoncode-blog/2018/1/29/javascript-asincrono

25. Fabric Node SDK. *Sitio Oficial.* https://fabric-sdk-node.github.io

26. Hyperledger Fabric Documentation. *Wallets.* https://hyperledger-fabric.readthedocs.io/en/release-1.4/developapps/wallet.html

27. Mongoose. *Sitio Oficial.* https://mongoosejs.com/

28. Auth0. *Hashing in Action: Understanding Bcrypt* https://auth0.com/blog/hashing-in-action-understanding-bcrypt/

29. JSON Web Tokens. *Introduction.* https://jwt.io/introduction/

30. IETF. *RFC 7519.* https://tools.ietf.org/html/rfc7519

31. NPM. *Package: Axios.* https://www.npmjs.com/package/axios

32. Amazon Web Services. *Sitio Oficial.* https://aws.amazon.com

33. Amazon Web Services. *AWS Blockchain templates.* https://aws.amazon.com/blockchain/templates/

34. IBM Blockchain. *Sitio Oficial.* https://www.ibm.com/blockchain

Bibliografía

- Ariel Ekblaw, Asaph Azaria, John D. Halamka, Andrew Lippman, Beth Isreal. *A case Study for Blockchain in Healthcare: "MedRec" prototype for electronic health records and medical research data.* 2016.
- Alevtina Dubovitskaya, Zhigang Xamuel Ryu et al. *Secure and Trustable Electronic Medical Records Sharin using Blockchain*.
- Guy Zyskind, Oz Nathan, Alex Pentland. *Decentralizing Privacy: Using Blockchain to Protect Personal Data.* 2015.
- Shuai Wang, Jing Wang, Xiao Want, Tianyu Qiu, Yong Yuan, Liwei Ouyang, Yuanyuan Guo, Fei-Yue Wen. *Blockchain-Powered Parallel Healthcare Systems Base on the ACP Approach.* 2018.