GRADO EN INGENIERÍA INFORMÁTICA

INGENIERÍA DE SOFTWARE

ESCUELA DE INGENIERÍA INFORMÁTICA

FCHAIN: UNA CADENA DE BLOQUES PARA FACTORING

Desarrollado por: Tomás Pérez Márquez

45780569N

Las Palmas de Gran Canaria, 04 de enero de 2019

TABLA DE CONTENIDO

I. ESTADO ACTUAL Y OBJETIVOS INICIALES

MOTIVACIÓN DEL PROYECTO

Cobrar anticipadamente el importe de una deuda puede suponer una gran ayuda para una empresa en un momento determinado de falta de liquidez. El factoring es una de las soluciones financieras que hace esto posible.

Esta alternativa de financiación es un servicio en el que se contrata el cobro de facturas con vencimiento a medio plazo, solicitando el cobro casi inmediato con un determinado descuento. Con este contrato, una empresa, cede el derecho de cobro de una factura a una entidad financiera, a la que denominamos factor. Esta entidad financiera, tras evaluar los riesgos, presentará una oferta y eventualmente efectuará el pago inmediatamente, restando una comisión que se lleva el factor que además se encargará de gestionar el cobro final de la factura. De esta manera, la empresa puede transformar sus ventas a crédito en operaciones al contado, mejorando la capacidad de financiación de su negocio.

Coloquialmente, se habla de "descontar una factura" cuando se contrata este tipo de servicios. Cada operación implica una comisión que puede llegar a suponer el 3% del crédito en la que además puede repercutir el coste de otros servicios, como un seguro de tipo de cambio (si la moneda es extranjera), un informe comercial previo o un seguro que cubra la operación.

Hasta aquí todo es conocido, pero resulta que este tipo de servicios se presta en muchos países a fraudes como falsificación de facturas o descontar una misma factura con varias entidades financieras, lo que se conoce como el problema del doble cobro. Este y otro tipo de fraudes impiden el desarrollo de este modelo de financiación en mercados en los que las autoridades intentan que se potencie como uno de los modelos dinamizadores de la economía.

OBJETIVOS DEL PROYECTO

El objetivo de este proyecto es desarrollar una blockchain privada para evitar evitar los diferentes tipos de fraudes descritos anteriormente. Esta blockchain podrá ser utilizada por entidades financieras que presten servicios de factoring y deudores que necesiten saber a quién abonar el importe de la factura, de tal forma que se pueda consultar el estado de la factura en cualquier momento. Para ello la blockchain registraría todos los eventos desde que se genera una factura hasta que se paga, incluyendo además la cadena de contratación de servicios para factoring que pueda derivarse.

Utilizar blockchain, una base de datos distribuida formada por cadenas de bloques, tiene una serie de ventajas que la convierten en una potente arma contra el fraude.

Al haber muchos nodos conectados entre sí almacenando la misma cadena de bloques, es virtualmente imposible poder alterarla porque ello implica intervenir en todos los nodos a la vez. Si se diera el caso de que se alterase la cadena de bloques en uno de los nodos, automáticamente sería rechazara por el resto. Esto hace de la tecnología blockchain, una tecnología que garantiza la inmutabilidad de los datos almacenados en sus bloques.

Son varias la razones que nos hacen apostar por una red distribuida pero privada. En primer lugar, es posible que no se desee que el control de la cadena de bloques se distribuya aleatoriamente, para la aplicación que se le quiere dar, las organizaciones probablemente necesiten una autorización especial, además de restringir a personas ajenas a contenidos comerciales privados.

En segundo lugar, el registro de información en las cadenas de bloques públicas puede llegar a ser costosa, Un entorno privado supondría el ahorro de una considerable cantidad de dinero. A diferencia de una cadena de bloques pública, en la que todos los participantes son anónimos, en esta cadena de bloques privada, todos los participantes serían conocidos miembros de la red. Cualquier organización que desease levantar un nodo dentro de esta blockchain o que simplemente desee conectarse deberá solicitarlo y replicar la cadena, con lo que en todo momento se tendría una cadena de bloques en la que existirá confianza entre todos los participantes. Y los costes serían igualmente distribuidos.

Se Seguirían agrupando los datos en bloques y encadenándolos mediante un hash, de esta manera detectaríamos en cualquier momento que alguien ha cambiado el historial. Con lo que podemos asegurar así la inmutabilidad de la información almacenada en la cadena de bloques. El proyecto es completamente fiel al estándar blockchain.

Con blockchain podemos tener un sistema descentralizado donde todos los participantes puedan tener una copia y anexar nuevos datos validados por el resto de los miembros de la red, de esta manera se elimina un obstáculo muy grande evitando que sea una sola organización la que posea toda la información.

II. JUSTIFICACIÓN DE COMPETENCIAS BÁSICAS

IS01: Capacidad para desarrollar, mantener y evaluar servicios y sistemas software que satisfagan todos los requisitos del usuario y se comporten de forma fiable y eficiente, sean asequibles de desarrollar y mantener y cumplan normas de calidad, aplicando las teorías, principios, métodos y prácticas de la ingeniería del software.

El proceso de análisis, diseño, desarrollo de la API y la cadena de bloques que se ha desarrollado a lo largo de este proyecto, demuestra que además de tener dichas capacidades, este proyecto es asequible de desarrollar y mantener, para suplir los problemas presentados en los primeros puntos de esta memoria de manera efectiva, fiable y eficiente.

IS02: Capacidad para valorar las necesidades del cliente y especificar los requisitos software para satisfacer estas necesidades, reconciliando objetivos en conflicto mediante la búsqueda de compromisos aceptables dentro de las limitaciones derivadas del coste, del tiempo, de la existencia de sistemas ya desarrollados y de las propias organizaciones.

La capacidad se puede observar en la estipulación de las historias de usuario que deben ser satisfechas para suplir las necesidades del cliente.

IS03: Capacidad de dar solución a problemas de integración en función de las estrategias, estándares y tecnologías disponibles.

Se han levantado varios nodos para poder ejecutar la cadena de bloques, se ha creado un servidor Tomcat para alojar un servicio web y se ha desarrollado el proyecto por medio de metodologías de desarrollo iterativas, aplicando además técnicas de TDD, refactorización y clean code.

IS04: Capacidad de identificar y analizar problemas y diseñar, desarrollar, implementar, verificar y documentar soluciones software sobre la base de un conocimiento adecuado de las teorías, modelos y técnicas actuales.

La identificación del problema que sufren las entidades financieras y deudores en los procesos de factoring, el posterior análisis de este, con el estudio de las posibles soluciones y la decisión de desarrollar e implementar una cadena de bloques en este proyecto para solucionar este problema.

IS05: Capacidad de identificar, evaluar y gestionar los riesgos potenciales asociados que pudieran presentarse.

A lo largo del desarrollo del proyecto, se han tenido en cuenta posibles riesgos y se ha tratado de evitar y mitigar los posibles daños que podrían ocasionar si llegaran a efectuarse.

IS06: Capacidad para diseñar soluciones apropiadas en uno o más dominios de aplicación utilizando métodos de la ingeniería del software que integren aspectos éticos, sociales, legales y económicos.

La elección de la cadena de bloques como solución al problema que se está tratando, es una solución más que apropiada puesto que utiliza unas medidas de seguridad bastante elevadas que protegen la integridad de los datos de los usuarios que la utilicen.

III. APORTACION AL ENTORNO SOCIOECONÓMICO

Las empresas, para desarrollar su actividad comercial, tienen que tratar diariamente con algunas partes de esta vasta red de productores, minoristas, distribuidores, transportistas y proveedores en una compleja disposición de procesos, que generan una serie de pagos por los servicios realizados.

La complejidad y la escalabilidad de los sistemas involucrados llevan a altos costes de transacciones que podrían dañar las pequeñas y medianas empresas de nuestro entorno al no poder hacer frente a los posibles desajustes y errores de papeleo, así como pérdidas o el robo de las facturas.

Este problema afecta a toda la cadena comercial, repercutiendo también a las entidades financieras que quedan vulnerables ante la falsificación de documentos al gestionar el cobro de facturas que ya han sido cobradas.

La aplicación de una cadena de bloques en estos sistemas para tramitar las facturas tiene el potencial de mejorar la eficiencia en las transacciones puesto que los procesos de pago quedan garantizados y con un registro conforme a lo acordado por ambas partes. Siendo un sistema de seguridad que apoya la gestión y evolución de todas partes involucradas en el sistema comercial del que deriva la empresa. Tanto las compañías participantes como las entidades financieras se conoce que factura falta por ser abonada al beneficiario y cuales ya si que lo han sido, minimizando así el fraude

y verificando la autenticidad desde el origen del servicio con una primera factura hasta su cobro.

La automatización de este sistema colaborativo aumenta la seguridad en la trazabilidad del cobro de facturas reduciéndose así:

- Altos costes de transacción.
- Errores humanos al tratar las facturas.
- Burocracia que se genera al tener tantos participantes esperando dicho cobro.
- El tiempo perdido ya que podría demorarse durante meses.
- Gasto en papel que supone un gran impacto medio ambiental.

Impulsando a todas y cada una de las partes involucradas en su desarrollo productivo de bienes y servicios, aumentando la seguridad y la confianza en los sistemas de cobro, ya que las facturas están siendo gestionadas adecuadamente, y potenciando el desarrollo económico-financiero del país.

IV. DESARROLLO

TECNOLOGÍAS UTILIZADAS

Las herramientas que he utilizado para desarrollar este proyecto son:

- **Multichain 2.0**. Una plataforma de código abierto que permite diseñar, implementar y operar registros distribuidos de tipo blockchain.
- Como entornos de programación he utilizado **eclipse Jee v4.9** para el desarrollo de la API y **NetBeans IDE 8.2** para el desarrollo de la interfaz de usuario.
- Para testear la API he utilizado **Postman** v6.6.1.
- El control de versiones del proyecto se ha realizado creando dos repositorios en **GitHub** gestionados a través de git.
- He utilizado también varios servidores de **DigitalOcean**. Estos han sido alquilados gracias a un código proporcionado por github en el **student developer pack**.
- Los diagramas los he realizado utilizando **Star Uml**.
- Dos servidores **Ubuntu 16.08** para hostear la cadena de bloques y el servicio web tomcat.
- **Apache Tomcat v9.0.**
- Mi portátil, un **MSI GE73VR 7RF Rider** con sistema operativo **Windows 10.**

V. ESTUDIO DE LA BLOCKCHAIN

Lo primero que hice una vez establecidos los objetivos del proyecto y definidos los límites de este, fue realizar un estudio del arte de la tecnología blockchain bastante profundo para comprender a la perfección su funcionamiento.

Para saber que era la blockchain me vino muy bien este video[3], que recomiendo ver.

Estuve leyendo información acerca del funcionamiento de la cadena de bloques durante semanas, pero cuando realmente comprendí a la perfección su funcionamiento, fue cuando comencé a realizar los tutoriales de la página de desarrolladores de multichain[4].

En principio pensé en utilizar una interfaz java de las que presenta la propia página web, pero finalmente me decidí por crear mi propia interfaz para crear la API que utilizaría para cumplir los objetivos planteados en este proyecto.

Para poder crear la interfaz que se comunicara con la cadena de bloques, tuve que aprender a comunicarme con la cadena de bloques utilizando peticiones basadas en JSON. Todos los métodos que se pueden ejecutar en la cadena de bloques de multichain se encuentran en esta página[5].

Una vez comprendí por completo el funcionamiento de la cadena de bloques multichain, tuve que aprender a utilizar gson para transformar JSON en objetos java con los que pudiera operar fácilmente. Para ello me serví de este tutorial[6].

También tuve que aprender a realizar peticiones get y post con autenticación básica http, desde java, pues saber esto iba a ser esencial para el desarrollo del proyecto. De paso aproveché y refresqué conocimientos acerca del desarrollo de servlets en java con este tutorial[7].

Una vez ya supe hacer todo lo mencionado anteriormente, tuve que comprender el proceso de vida de una factura y como podría utilizar la blockchain para llevar un registro de este ciclo de vida. Para ello me puse en contacto con los trabajadores del proyecto Wupplier[8], una plataforma para gestionar la cadena de valor en las relaciones entre proveedores y suministradores, de la empresa singular factory. Quienes me explicaron todos los procesos y estados por los que pasa una factura desde que se genera hasta que se paga. Tuve bastantes conversaciones con ellos, cada vez que pensaba en una manera de registrar todos los estados del ciclo de vida de una factura los procesos de factoring, lo consultaba con ellos para comprobar si realmente había ideado un buen sistema de registro.

Las facturas que utilizaría en mi proyecto provienen de México. Esto es así debido a que el Sistema Tributario Mexicano está muy avanzado tecnológicamente y todas las facturas son digitales y están en formato xml.

Con lo que me puse en contacto con la oficina del sistema tributario Mexicano por medio de un chats que tienen en su página web[9] para solicitar información acerca de las facturas digitales y tratar de conseguir algún ejemplo real para mi proyecto. Hablé con 4 o 5 departamentos y finalmente no conseguí absolutamente nada, así que navegué por internet hasta un portal de desarrolladores en donde estaban discutiendo acerca de las versiones de los esquemas xml de las facturas digitales mexicanas. Tras informarme bastante sobre el tema, decidí utilizar el tipo de factura cfdv32 que no es el más actual, pero consideré que me sería más sencillo encontrar ejemplos reales. Tras emplear un par de horas buscando, conseguí una factura digital real en formato xml, que actualmente utilizo como modelo para crear mis propias facturas y realizar pruebas sobre la cadena de bloques.

VI. APLICACIÓN BLOCKCHAIN EN LOS PROCESOS DE FACTORAJE.

Tras estudiar en profundidad el funcionamiento de la cadena de bloques y su aplicación en algunos proyectos reales. Pude pensar en la manera de cumplir los objetivos mediante el uso de esta cadena de bloques. El funcionamiento de esta cadena de bloques será el siguiente:

El deudor genera un token (un identificador único), que representa el derecho de cobro de una factura de una compañía. Dicho token, contendrá toda la información relevante de la factura. Una vez generado, será entonces transferido al beneficiario, quién ahora tendrá en su poder el token, ahora es este posee el derecho de cobro de la factura.

Si el beneficiario, usuario que actualmente posee el token, decidiera solicitar un adelanto a una entidad financiera, la entidad financiera podrá consultar en la cadena de bloques si este presunto beneficiario posee en su poder el token que realmente confirma que es el beneficiario actual de la factura, pudiendo así adelantar la factura con la seguridad de que el beneficiario no está cometiendo fraude. Es en este momento cuando el beneficiario inicial de la factura deberá transferir su token a esta entidad financiera, quien ahora tiene el derecho de cobro de la factura.

Cuando llegue el momento en el que el deudor tenga que saldar su deuda, consultará la cadena de bloques para comprobar quien es el usuario que actualmente posee en su poder el derecho de cobro de una factura, evitando así que el beneficiario inicial cometa fraude cobrando de nuevo una factura que ya ha cobrado.

Una vez la deuda esté saldada, el token que representa el derecho de cobro de dicha factura será quemado (enviado a una dirección inaccesible) por el poseedor de este, dejando patente en la cadena de bloques que dicha factura ya ha sido pagada.

VII. FASE DE ANÁLISIS Y DISEÑO

Tras acabar con la fase de estudio de las tecnologías que iba a utilizar en el proyecto e ideado el proceso de registro de todos los ciclos de vida de una factura incluyendo los servicios derivados de factoring, comencé con la fase de análisis y diseño.

ESTIPULACIÓN DE HISTORIAS DE USUARIO

Lo primero que hice en esta fase fue redactar las historias de usuario para definir los requisitos que debía cumplir el proyecto. Cada una de estas historias fue descompuesta entonces en una o varias tareas bien diferenciadas entre sí. Este proyecto consta de una API que se comunica con la cadena de bloques y una Interfaz de usuario que se comunica con dicha API, así que hice una diferenciación entre las historias de usuario de la interfaz de usuario con la API y del usuario con la interfaz de usuario.

HISTORIAS DE USUARIO INTERFAZ GRÁFICA -> API

Historia de usuario	**Como sistema quiero interactuar con una cadena de bloques para poder registrar todos los estados del ciclo de vida de una factura**
Tareas	▪ Levantar un par de servidores para alojar la blockchain. ▪ Instalar multichain. ▪ Configurar la cadena de bloques. ▪ Ejecutar la cadena de bloques. ▪ Intercambiar información entre la API y la cadena de bloques
Validación	La cadena de bloques se ejecuta correctamente en varios nodos a la vez y se comunica con la API.

Historia de usuario	**Como sistema quiero crear un par de claves en la blockchain para asignárselas a un usuario cuando se registre.**
Tarea	Generar par de claves en la cadena de bloques
Validación	La cadena de bloques se ejecuta correctamente en varios nodos a la vez y se comunica con la API.

Historia de usuario	**Como sistema quiero validar una clave privada en la cadena de bloques para comprobar que es válida.**
Tarea	Validar una clave privada en la cadena de bloques.
Validación	La cadena de bloques devuelve el par de claves asociado a una clave privada.

Historia de usuario	**Como sistema quiero generar un token para registrar en la blockchain la factura que desea el usuario.**
Tarea	Validar una clave privada en la cadena de bloques.
Validación	La cadena de bloques devuelve el par de claves asociado a una clave privada.

Historia de usuario	**Como sistema quiero generar un token para registrar en la blockchain la factura que desea el usuario.**
Tarea	Generar un token en la blockchain que almacene la información deseada.
Validación	Se genera un token en la cadena de bloques.

Historia de usuario	**Como sistema quiero obtener un token específico de la blockchain para poder devolver los datos que este almacena.**
Tarea	Obtener un token de la cadena de bloques a partir del nombre
Validación	Se puede obtener un token a partir del nombre

Historia de usuario	**Como sistema quiero obtener el dueño de un token para saber quien es el beneficiario de una factura**
Tarea	Obtener al creador del token a partir del nombre de este.
Validación	Se puede obtener al creador de un token

Historia de usuario	**Como sistema quiero obtener el último dueño de un token quemado para saber quién fue el beneficiario de una factura liquidada.**
Tarea	Obtener la dirección del usuario que ha quemado un token.
Validación	La dirección del usuario que quemó un token es obtenida

Historia de usuario	**Como sistema quiero transferir tokens entre carteras para registrar la cesión del derecho de cobro de una factura.**
Tarea	Llevar a cabo transacciones firmadas entre direcciones de la cadena de bloques.
Validación	Un token es transferido de una cartera a otra.

Historia de usuario	**Como sistema quiero quemar tokens para registrar que una factura ha sido liquidada**
Tarea	Llevar a cabo transacciones entre un usuario y la dirección de quemado.
Validación	Un token es enviado a una dirección inaccesible.

Historia de usuario	**Como sistema quiero obtener el saldo de tokens de una cartera para obtener las facturas que el usuario aún debe cobrar.**
Tarea	Obtener los tokens que posee una cartera determinada.
Validación	Los tokens que posee un usuario son obtenidos.

Historia de usuario	**Como sistema quiero obtener al creador de un token para mostrar quien es el deudor de una factura.**
Tarea	Obtener la dirección del creador de un token.
Validación	La dirección del creador de un token es obtenida

Historia de usuario	**Como sistema quiero obtener al primer receptor de un token para mostrar quien es el beneficiario original de una factura.**
Tarea	Obtener la dirección de la primera dirección a la que fue transferido un token.
Validación	La primera dirección a la que fue transferido un token ha sido obtenida.

HISTORIAS DE USUARIO INTERFAZ GRÁFICA -> API

Historia de usuario	**Como usuario quiero poder registrarme en el sistema para disfrutar de las funcionalidades que este ofrece.**
Tarea	Generación de un par de claves para que el usuario interactúe con la cadena de bloques.
Validación	La API devuelve un par de claves cuando se ejecuta una llamada específica.

Historia de usuario	**Como usuario quiero iniciar sesión en el sistema para disfrutar de las funcionalidades que este ofrece.**
Tarea	Validación de una clave privada.
Validación	Al consultar una clave privada la API devuelve el par de claves asociado.

Historia de usuario	**Como usuario quiero registrar una factura para poder consultar en cualquier momento la información que contiene.**
Tarea	▪ Extracción de información de una factura en formato xml ▪ Generación de un código hexadecimal de 16 dígitos mediante el algoritmo de encriptación MD5 a través de los datos recogidos de la factura que servirá como nombre al token. ▪ Generar en las carteras de la blockchain tokens generados a partir de la información de la factura que representen.
Validación	El usuario puede generar un token con un nombre único generado por MD5 a partir de los datos de una factura y que almacene la información de la factura.

Historia de usuario	**Como usuario quiero saber quién es el beneficiario actual de una factura para evitar posibles actos fraudulentos.**
Tarea	Obtener la dirección de la cartera del dueño del token que representa la factura.
Validación	El usuario puede consultar quien posee un determinado token.

Historia de usuario	**Como usuario quiero saber quién fue el beneficiario original de una factura.**
Tarea	Obtener la dirección del destinatario de la primera transacción del token.
Validación	El usuario puede consultar quien fue el beneficiario original de una factura.

Historia de usuario	**Como usuario quiero saber quién es el deudor de una factura para poder solicitarle el pago de esta**
Tarea	Obtener la dirección del creador del token que representa la factura
Validación	El usuario puede consultar quien creó determinado token.

Historia de usuario	**Como usuario quiero transferir el derecho de cobro de una factura para poder obtener un adelanto de esta.**
Tarea	Crear transacciones firmadas de tokens entre carteras de la cadena de bloques.
Validación	Los tokens pueden ser transferidos de una cartera a otra

Historia de usuario	**Como usuario quiero cambiar el estado de una factura a pagada para dejar registrado que esta ha sido liquidada.**
Tarea	Crear transacciones firmadas de tokens entre una cartera y una dirección de quemado (dirección inaccesible).
Validación	Los tokens pueden ser transferidos de una cartera a una dirección de quemado

Historia de usuario	**Como usuario quiero comprobar el estado de una factura para saber si esta ya ha sido liquidada o no.**
Tarea	Comprobar si el dueño actual de un token es la dirección de quemado en cuyo caso obtener la dirección de la wallet que quemó dicho token.
Validación	Obtener el penúltimo dueño de un token.

DIAGRAMA DE CASOS DE USO

Una vez estipuladas todas las historias de usuario de mi proyecto, decidí desarrollar el diagrama de usos pertinente para describir todas las actividades que deberá realizar tanto la API como la interfaz de usuario para llevar a cabo un registro de todos los estados del ciclo de vida de una factura incluyendo los procesos de factoring que puedan derivarse.

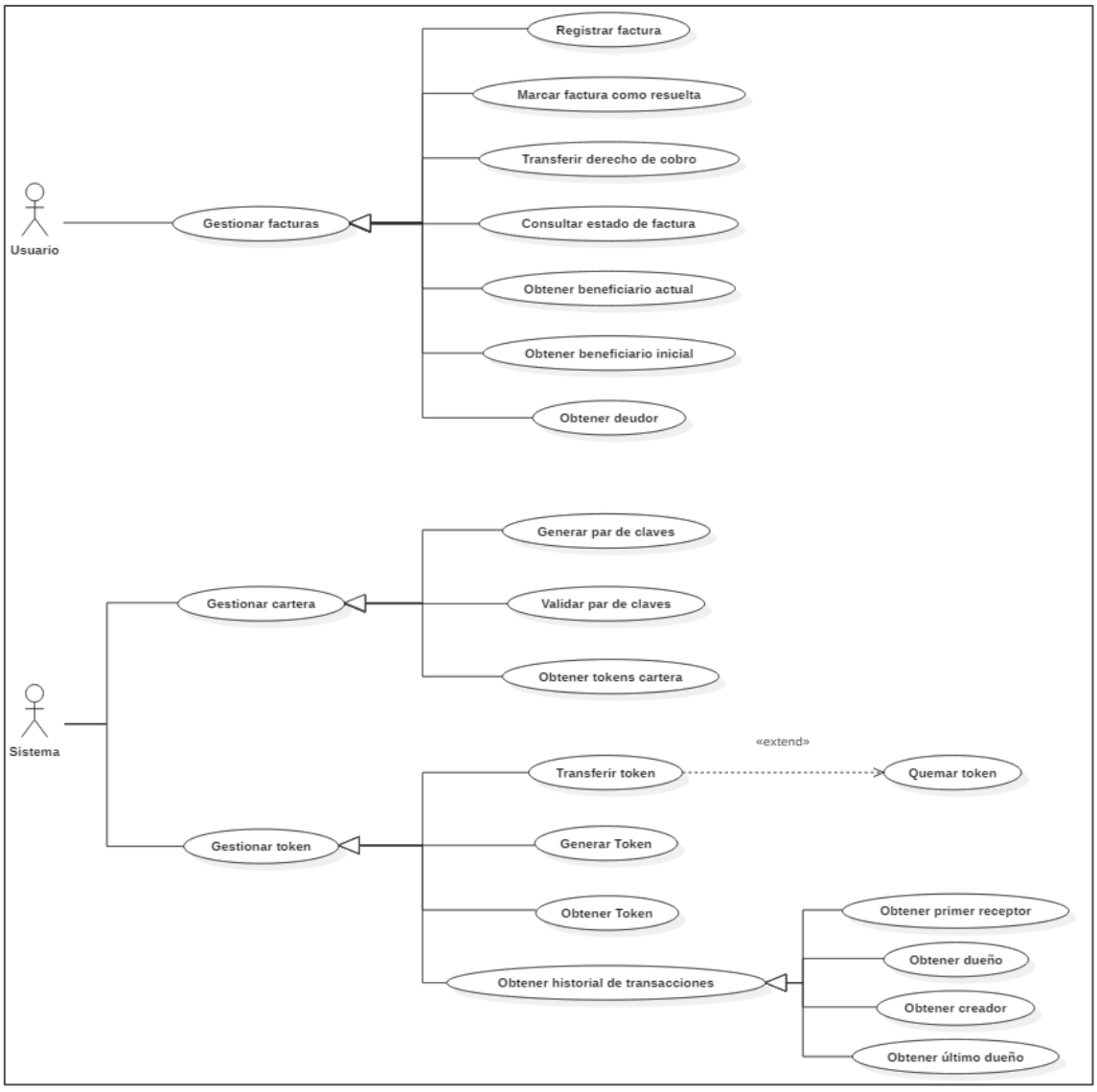

Ilustración 1: Diagrama de casos de uso

VIII. ITERACIONES DE DESARROLLO

Aquí está el enlace a los repositorios del proyecto[1][2]

Para el desarrollo de este proyecto he seguido una metodología iterativa dividida por fases.

Las iteraciones que he seguido a la hora de desarrollar el proyecto han sido las siguientes:

1. Instalación y configuración de un servidor tomcat que posteriormente alojará el servicio web restFul que se comunicará con la cadena de bloques.
2. Diseño, configuración e implementación de la cadena de bloques.
3. Creación de la API que se comunicará con la cadena de bloques implementada con multichain.
4. API: Desarrollo de funciones para la ejecución de comandos en la cadena de bloques.
5. Desarrollo de las consultas al servicio web desde la interfaz de usuario
6. Lectura, Cifrado y registro de facturas en la cadena de bloques
7. Diseño y desarrollo de una interfaz básica de usuario

1. INSTALACIÓN Y CONFIGURACIÓN DEL SERVIDOR TOMCAT

La primera iteración en el desarrollo del proyecto, fue simplemente levantar un servidor Ubuntu en Digital Ocean e instalar y configurar un servidor apache Tomcat que alojará el servicio web Restful con el que se comunicará la interfaz de usuario. Para cumplir con esta iteración, seguí los pasos de esta guía[10].

2. DISEÑO, CONFIGURACIÓN E IMPLEMENTACIÓN DE LA CADENA DE BLOQUES.

Lo primero que hice en esta iteración fue instalar multichain en los dos servidores de Digital Ocean que tengo alquilados. También instalé multichain en mi ordenador, que efectuará el papel de nodo durante las fases de desarrollo del servicio web.

Para crear la cadena de bloques ejecuté por console el comando *multihcain-util créate FChain* en uno de los nodos, que me crea todos los archivos de configuración de la cadena de bloques cuyos parámetros debo modificar a mi gusto.

Antes de inicializar esta cadena de bloques estudié los archivos de configuración[11] y comprobé que se ajustaban correctamente a los objetivos de mi proyecto. Solo modifiqué un parámetro relacionado con el minado de bloques para permitir que cualquier miembro de la red pudiera minar bloques.

una vez creada y configurada la cadena de bloques procedí a inicializarla creando también el bloque génesis y minando una cantidad determinada de bloques para que la cadena fuera lo bastante larga como para que la información que se comience a registrar a partir de ahora sea incorrompible. Una vez inicializada, debo establecer conexión desde los otros nodos para que estos puedan participar en las actividades de minado de la cadena de bloques

Cuando un nuevo nodo intente conectarse a la cadena de bloques, primero descarga un conjunto mínimo de parámetros de la cadena de bloques del nodo existente y los escriben en el archivo params.dat en el directorio correspondiente. Una vez que se le conceda permiso, desde un nodo con permisos de administrador, para conectarse, pueden descargar el conjunto completo de parámetros de blockchain. Así evitamos que haya participantes desconocidos en la red que pueda tratar de robar o corromper los datos almacenados.

Una vez definidos los parámetros de configuración, procedí a conectar el otro nodo con el primero, para ello le di permiso desde el nodo principal. Una vez conectado, este segundo nodo descargó la cadena de bloques y comenzó a minar.

La cadena de bloques ya estaba creada y ejecutándose remota y distribuidamente en los servidores de Digital Ocean, también me conecté desde mi portátil para poder desarrollar cómodamente el servicio web, pues si quisiera realizar peticiones directamente de mi ordenador a la cadena de bloques, debería modificar el archivo de configuración de la cadena de bloques y permitir que puedan entrar peticiones de miembros externos a la cadena de bloques, lo cual no me interesa por temas de seguridad. con lo que procedí a crear la API que se comunicaría con esta cadena de bloques.

ANTES DE SEGUIR CON LA SIGUIENTE ITERACIÓN DEL PROYECTO, CABE DESTACAR…

GIT

Para el control de versiones de este proyecto he usado Git, el repositorio está alojado en github y es accesible desde los enlaces que aparecen en la bibliografía.

Como cliente git he utilizado gitKraken, que me ha facilitado bastante el trabajo a la hora de gestionar el repositorio. Tras inicializar el repositorio git, he creado una rama develop a partir de la rama master para trabajar en el proyecto. Cada vez que empezaba una nueva funcionalidad creaba una nueva rama para desarrollar la funcionalidad en cuestión. Una vez acababa el desarrollo de dicha funcionalidad hacía un merge con develop.

Para acceder a dichos repositorios, usar los hipervínculos que hay en la bibliografía.

TDD Y CLEAN CODE

Para el desarrollo de este proyecto, he utilizado el desarrollo guiado por pruebas (TDD). Considero que es la manera más efectiva, rápida y limpia de programar. Además aplicar esta metodología de desarrollo a mi proyecto me ha permitido profundizar bastante en los conocimientos adquiridos en las asignaturas de desarrollo software.

A lo largo de todo el desarrollo de mi proyecto he seguido la metodología de desarrollo guiado por pruebas. Creando la correspondiente prueba unitaria, resolviendo dicha prueba y refactorizando el código. Considero que el código de mi proyecto está bastante limpio siendo muy autodescriptivo, fácil de leer y comprender.

REPOSITORIOS MAVEN DEL SERVICIO RESTFUL

Una vez inicializado el repositorio git, procedí a instalar una serie de repositorios Maven que me resultarían muy útiles para desarrollar esta API.

La API que he creado se basa en realizar consultas JSON con autenticación básica a la cadena de bloques que ya había creado.

Las dependencias de mi archivo pom.xml son las siguientes:

```
            <dependency>
                  <groupId>junit</groupId>
                  <artifactId>junit</artifactId>
```

```
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>com.sun.jersey</groupId>
            <artifactId>jersey-core</artifactId>
            <version>1.19</version>
        </dependency>
        <dependency>
            <groupId>com.sun.jersey</groupId>
            <artifactId>jersey-json</artifactId>
            <version>1.19</version>
        </dependency>
        <dependency>
            <groupId>com.sun.jersey</groupId>
            <artifactId>jersey-servlet</artifactId>
            <version>1.19</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.5</version>
        </dependency>
        <dependency>
            <groupId>com.google.code.gson</groupId>
            <artifactId>gson</artifactId>
            <version>2.8.5</version>
        </dependency>
        <dependency>
            <groupId>org.apache.httpcomponents</groupId>
            <artifactId>httpclient</artifactId>
            <version>4.3.4</version>
        </dependency>
```

- Los repositorios de jersey los utilizaré para crear el servlet que recibirá las peticiones desde la interfaz gráfica de usuario y que enviará las respuestas convenientes.
- El repositorio de Google lo utilicé para traducir las respuestas JSON que recibo desde la cadena de bloques a objetos con los que pueda operar fácilmente.
- El repositorio de apache lo utilizo para realizar las peticiones con autenticación http básica a la cadena de bloques.

3. DESARROLLO DE LA API QUE SE COMUNICARÁ CON LA CADENA DE BLOQUES IMPLEMENTADA CON MULTICHAIN

La primera clase que cree fue *FChainInterface.java* que está alojada en el paquete *com.tfg2018.ws.rest.fchain*. Esta clase la utilizo para conectarme y comunicarme con la cadena de bloques.

```
public FchainInterface(String ip, String port, String login, String password)
{
	connect(ip, port, login, password);
}
```

En el constructor de esta clase lo único que hago es establecer conexión con la cadena de bloques mediante el método *connect()*.

```
protected void connect(String ip, String port, String login, String password)
{
	this.httppost = new HttpPost("http://" + ip + ":" + port);

	CredentialsProvider provider = new BasicCredentialsProvider();
	UsernamePasswordCredentials credentials = new
UsernamePasswordCredentials(login, password);
		provider.setCredentials(AuthScope.ANY, credentials);

	this.httpclient =
HttpClientBuilder.create().setDefaultCredentialsProvider(provider).build();
}
```

Como se puede observar, para poder establecer conexión con la cadena de bloques además del puerto y la dirección ip de un nodo, necesito pasar las credenciales y realizar una autenticación vía http básica. Todos estos parámetros los almaceno en una clase llamada *FchainConst.Java* Para no tener que copiarlos a mano cada vez que quiera crear una instancia de esta clase.

Esta clase tiene además otros dos métodos que son utilizados para ejecutar las consultas a la cadena de bloques (*executeRequest()*)y para traducir las respuestas de esta a objetos con los que pueda operar fácilmente (*translateResponse()*.

Al método *executeRequest()* le paso un parámetro de tipo StringEntity. Lo primero que hago es comprobar que el objeto *FchainInterface* haya sido inicializado y sea capaz de establecer conexión con la cadena de bloques. Una vez realizada esta comprobación, establezco este Stringentity como cuerpo de la petición post que voy a enviar a la cadena de bloques y la ejecuto.

```
public Object executeRequest(StringEntity rpcEntity) throws Exception {
	if (this.httpclient != null && this.httppost != null) {
		this.httppost.setEntity(rpcEntity);
	} else {
		throw new Exception("Connection with blockchain failed");
	}
	CloseableHttpResponse response = httpclient.execute(httppost);
	HttpEntity entity = response.getEntity();

	String answer = EntityUtils.toString(entity);
	response.close();

	return translateResponse(answer);
}
```

Una vez ejecutada la petición, cojo la respuesta y antes de retornarla, utilizo el método *translateResponse*() para traducirla a un objeto con el que pueda operar.

```
private Object translateResponse(String answer) throws Exception {
	final Gson gson = new GsonBuilder().create();
	final FchainResponse fChainresponse = gson.fromJson(answer,
FchainResponse.class);

	if (fChainresponse != null && fChainresponse.getError() == null) {
		return fChainresponse.getResult();
	} else {
		throw new Exception("The response is null");
	}
}
```

Como se puede observar utilizo Gson para traducir la respuesta a un objeto *FChainResponse* con el que pueda operar. De este objeto *FChainResponse* me interesan dos parámetros.

- *Object response*. Es la respuesta con la que nos interesa operar.
- *String id*. Se trata el id de la transacción que ha ejecutado nuestra petición.

Aquí ya comprobaría que la respuesta no es nula y la retorno.

El siguiente paso que hice fue crear una clase para poder traducir los comandos que deseo ejecutar en la cadena de bloques y sus respectivos parámetros a un formato JSON que la cadena de bloques sea capaz de interpretar. Ha esta clase la llame *CommandTranslator.java* y está alojada en el paquete *com.tfg2018.ws.rest.utils*. Esta clase esta compuesta por 3 métodos.

El método *getJsonMap()* se encarga de generar un *Map<String, Object>* a partir del comando y los parámetros del comando que le paso como atributos. El formato JSON

que requiere la cadena de bloques tiene 3 componentes principales. El primero es un identificador único universal que genero aleatoriamente, el segundo es el nombre del comando que se desea ejecutar y el último son los parámetros que requiere dicho comando.

```
private static Map<String, Object> getJsonMap(String command, Object...
parameters) {
	Map<String, Object> mappedCommand = new HashMap<String, Object>();
	mappedCommand.put("id", UUID.randomUUID().toString());
	mappedCommand.put("method", command.toString().toLowerCase());
	List<Object> paramList = new
ArrayList<Object>(Arrays.asList(parameters));
	mappedCommand.put("params", paramList);
	return mappedCommand;
	}
```

El método *formatJson()* se encarga de transformar el Mapa que generamos en el método anteriormente explicado a JSON.

```
public static String formatJson(Object value) {
	final GsonBuilder builder = new GsonBuilder();
	final Gson gson = builder.create();
	return gson.toJson(value);
}
```

El método principal de la clase es *commandToJson()* se sirve de los otros dos métodos para transformar el JSON generado a partir del Map que contiene los parámetros que requiere la cadena de bloques para ejecutar un comando a una StringEntity que utilizaré posteriormente en todas las operaciones que quiera realizar con la cadena de bloques.

```
public static StringEntity commandToJson(String command, Object...
parameters) throws Exception {
	Map<String, Object> mappedCommand = getJsonMap(command, parameters);
	StringEntity jsonResult;
	try {
		jsonResult = new StringEntity(formatJson(mappedCommand));
		return jsonResult;
	} catch (UnsupportedEncodingException e) {
		throw new Exception("Json traduction error");
	}
}
```

Una vez creadas estas dos clases, ya podía operar con la cadena de bloques y comenzar el desarrollo de todas las funciones que utilizará mi servicio web.

4. API: DESARROLLO DE FUNCIONES PARA LA EJECUCIÓN DE COMANDOS EN LA CADENA DE BLOQUES

GESTIÓN DE CARTERAS

Las primeras funcionalidades que desarrollé fueron las relacionadas con la gestión de carteras de la cadena de bloques. Antes de comenzar a programar estas funciones, debía ser capaz de traducir las respuestas de la cadena de bloques que recibiría en formato JSON, para ello cree las clases *KeyPairs.java* y *AddressValidator.java*. Estas clases la utilizo para traducir las respuestas de la cadena de bloques a que llegan en formato JSON y necesitan ser transformadas a un objeto con el que poder operar.

La clase *KeyPairs.java* tiene 3 atributos, estos son la dirección, clave pública y la clave privada, estando estos atributos convenientemente encapsulados. Esta clase me resulta muy útil no solo para transformar las respuestas provenientes de la cadena de bloques si no también para crear los cuerpos de las consultas que posteriormente traduciré a JSON para operar con dicha cadena de bloques.

Las respuestas de la cadena de bloques cuando solicite la validación de una dirección o clave privada, serán traducidos a una instancia de la clase *AddressValidator.java,* que me permitirá comprobar fácilmente la validez de la clave o dirección consultada.

Además de estas dos clases creé otra clase llamada GsonToObjectTranslator que está alojada en el paquete *com.tfg2018.ws.rest.utils*. Que se encargará de traducir las respuestas de tipo *Object* que provienen de la cadena de bloques en objetos con los que poder operar.

En esta clase creé dos métodos que utilizaré en la clase que gestiona las carteras en la cadena de bloques. El primero método me transforma el *object* que le paso como parámetro a un objeto *KeyPairs*.

```
public static KeyPairs getKeys(Object keyPair) {
	String a = keyPair.toString().substring(1, keyPair.toString().length()
- 1);
	Gson gson = new Gson();
	return gson.fromJson(a, KeyPairs.class);
}
```

El segundo método método transforma un *Object* que le paso como parámetro a un objeto *AddressValidator*.

```
public static AddressValidator isKeyValid(Object validator) {
	String a = validator.toString();
	Gson gson = new Gson();
	return gson.fromJson(a, AddressValidator.class);
}
```

Una vez creada las clases a las que transformar las respuestas de la cadena de bloques y las funciones para llevarlo a cabo, pude comenzar a desarrollar los métodos que utilizo

para gestionar las carteras. Para ello cree la clase *WalletManager.java* que está alojada en el en el paquete *com.tfg2018.ws.rest.fchain*. Esta clase me permite:

- generar pares de claves.
- Importar pares de claves a los nodos de la cadena de bloques.
- Validar direcciones y claves privadas en la cadena de bloques
- Garantizar a las carteras permisos para generar, recibir y enviar tokens.

Procederé a explicar estos métodos.

El método *importAddress()* ejecuta un comando en la cadena de bloques, que se encarga de importar un par de claves que haya sido generado al nodo al que le estemos enviando la consulta. Es necesario importar los pares de claves a los nodos para poder consultar su saldo, de otra manera no podríamos acceder a dicha dirección.

```
private void importAddress(String address) throws Exception {
	try {
		StringEntity request = 
CommandTranslator.commandToJson("importaddress", address);
		this.fChainQuerier.executeRequest(request);
	} catch (Exception e) {
		throw new Exception("Address importation error");
	}
}
```

El método *getNewKeyPair()* se encarga de solicitar a la cadena de bloques la generación de un nuevo par de claves y de importar la dichas claves al nodo de la cadena de bloques con el que estemos operando.

```
public KeyPairs getNewKeyPair() throws Exception {
	Object keyPair;
	try {
		keyPair = 
CommandTranslator.formatJson(this.fChainQuerier.executeRequest(CommandTransla
tor.commandToJson("createkeypairs")));
		KeyPairs response = GsontoObjectTranslator.getKeys(keyPair);
		importAddress(response.getAddress());
		return response;
	} catch (Exception e) {
		throw new Exception("Key Pair generation error");
	}
}
```

El método *validateAddress()* se encarga de validar en la cadena de bloques una dirección para asegurar que esta es correcta y que pertenece a la cadena de bloques

```
public String validateAddress(String key) throws Exception {
	StringEntity request = CommandTranslator.commandToJson("validateaddress", key);
	Object validator = CommandTranslator.formatJson(this.fChainQuerier.executeRequest(request));
	AddressValidator addressValidator = GsontoObjectTranslator.isKeyValid(validator);
	if (!addressValidator.getIsvalid()) {
		throw new Exception("this address is not valid");
	}
	return addressValidator.getAddress();
}
```

El método *grantPermission()* solicita al nodo de la cadena de bloques que otorgue permisos para generar, enviar y recibir tokens a la dirección que le pase como parámetro. Para poder otorgar los permisos necesito una dirección de una cartera administrador que es la que tiene permiso para conceder permisos.

```
public void grantPermission(String address) throws Exception {
	try {
	this.fChainQuerier.executeRequest(CommandTranslator.commandToJson("grantfrom", FchainConst.ADMIN_ADDRESS , address, "issue,send,receive"));
	} catch (Exception e) {
		throw new Exception("Fallo al otorgar permisos");
	}
}
```

GESTIÓN DE TOKENS

Al haber podido generar claves en la cadena de bloques, había llegado el momento de crear las funciones relacionadas con la gestión de los tokens. Comencé creando la clase *Token.js* para poder traducir las respuestas de la cadena de bloques a objetos con los que poder operar.

La clase *Token.js* me servirá tanto para traducir las respuestas de la cadena de bloques que contengan información sobre los tokens como para generar

```
public Token(String assetName, Map<String, String> details) {
	this.name = assetName;
	this.issueqty = 1.0;
	this.units = 0.1;
	this.issueraw = 0;
	this.details = details;
}
```

Los atributos *issueqty, units* e *issueraw*, los inicializo a esos valores para especificarle a la cadena de bloques que de cada token creado solo quiero que haya una única unidad

no divisible y que no se puedan volver a generar más unidades de este token, garantizando así que no puedan existir dos facturas iguales registradas en la cadena de bloques.

Además de esta clase también tuve que crear el método que ejecutará la traducción de la respuesta en la clase *GsonToObjectTranslator.java*

```
public static Token getToken(Object token) {
	String a = token.toString().substring(1, token.toString().length() -
1);
	Gson gson = new Gson();
	return gson.fromJson(a, Token.class);
}
```

Una vez hecho esto, procedí a desarrollar la clase *TokenManager.java* esta clase me permite:

- Generar tokens.
- Obtener información de los tokens generados en la cadena de bloques.
- Suscribirme a los tokens que genere.

El método *subscribeToken()* genera la petición para que el nodo de la cadena de bloques se suscriba a un token específico y de esta manera, poder hacer un seguimiento de este. Este método solo requiere el nombre del token al que deseo suscribirme.

```
private void subscribeToken(String tokenName) throws Exception{
	StringEntity request = CommandTranslator.commandToJson("subscribe",
tokenName);
	try {
	CommandTranslator.formatJson(this.fChainQuerier.executeRequest(request
));
	} catch (Exception e) {
		throw new Exception("Error in token subscription");
	}
}
```

El método *generateToken()* me permite generar un token en la cadena de bloques a partir de una instancia del objeto *Token* en una determinada cartera. He de dejar claro que es la propia cartera la que genera este token en su propia cuenta, Este hecho me servirá en el futuro para saber quien es el pagador de la factura, pues este es quien debe subir el token a la cadena de bloques.

```
public void generateToken(Token token, String address) throws Exception {
	StringEntity request = CommandTranslator.commandToJson("issue",
address, token.getName(), token.getIssueqty(),
			token.getUnits(), token.getIssueraw(),
token.getDetails());
	try {
```

```
		CommandTranslator.formatJson(this.fChainQuerier.executeRequest(request
));
			subscribeToken(token.getName());
		} catch (Exception e) {
			throw new Exception("Token generation fail");
		}
}
```

Por último el método *getToken()* me permite obtener un token de la cadena de bloques a partir del nombre del mismo. Es entonces cuando utilizo la clase GsonToObjectTranslator para traducir la respuesta a un objeto de tipo *Token*.

```
public Token getToken(String tokenName) throws Exception {
		StringEntity request = CommandTranslator.commandToJson("listassets",
tokenName);
		try {
			String resultToken =
CommandTranslator.formatJson(this.fChainQuerier.executeRequest(request));
			Token token = GsontoObjectTranslator.getToken(resultToken);
			return token;
		} catch (Exception e) {
			throw new Exception("this token does not exist");
		}
}
```

GESTIÓN DE TRANSACCIONES

Para la gestión de las transacciones no me ha hecho falta crear ningún objeto para traducir las respuestas de la cadena de bloques, pues la propia cadena de bloques en cuanto a transacciones se refiere, solo me devuelve el número identificativo de la transacción.

Aprovecharé para profundizar más en el funcionamiento de las transacciones que está utilizando mi cadena de bloques. Lo primero que hay que hacer es crear una transacción no firmada, una vez hecho esto, la cadena de bloques me devolverá un número identificativo de esta transacción. Posteriormente, gracias a este número identificativo, la transacción en cuestión podrá ser firmada por la cartera que la creó. Una vez firmada, la cadena de bloques me devolverá un número identificativo de esta. Para acabar, cogeré ese número identificativo de la transacción firmada para ejecutarla en la cadena de bloques y que así se lleve cabo todo el proceso de minado que la confirmará.

Para llevar a cabo estos procesos, creé la clase *TransactionManager.java* que está alojada en *com.tfg2018.ws.rest.fchain*. Esta clase me permite:

- Crear y firmar transacciones
- Enviar transacciones firmadas

Esta clase utiliza varios métodos para llevar a cabo sus objetivos. El método *prepareMap()* se encarga de preparar los parámetros de la transacción que pretendemos crear en la cadena de bloques. Para ello es necesario que le pasemos como parámetros, el nombre del token que deseamos enviar y la dirección de destino a la que deseamos enviar el token. Dentro del método, yo especifiqué que la cantidad enviada es un solo token completo, sin ser dividido en ninguna parte.

```
private Map<String, Object> prepareMap(String destination, String tokenName)
{
	Map<String, Object> mapParams = new HashMap<String, Object>();
	Map<String, Double> filledAsset = new HashMap<String, Double>();
	filledAsset.put(tokenName, 1.0);
	mapParams.put(destination, filledAsset);
	return mapParams;
}
```

Con el método *prepareMap()* creado, desarrollé el método *createAndSignRawTransaction()*.

```
public String createAndSignRawTransaction(String senderAddress, String
senderPrivKey, String destination, String tokenName) throws Exception {
	String hexBlob;
	try {
		Map<String, Object> mapParams = prepareMap(destination,
tokenName);
		StringEntity request =
CommandTranslator.commandToJson("createrawsendfrom",senderAddress,
mapParams);
		hexBlob = this.fChainQuerier.executeRequest(request).toString();
	} catch (Exception e) {
		throw new Exception("error creating the transaction, the asset
doesn't belong to the sender address");
	}
	return signRequest(senderPrivKey, hexBlob);
}
```

Como se puede observar, cuando creo la transacción no firmada, la cadena de bloques me devuelve un hexblob, lo que viene siendo un código hexadecimal bastante largo. Este número hexadecimal es el identificador de la transacción que hemos creado, lo que tengo que hacer a continuación es firmar dicha transacción. Para ello desarrollé el método *signRequest()*. Que firma la transacción con la clave privada de la cartera del usuario que la ha creado. Este método me devuelve un número hexadecimal que identifica a la transacción que acabamos de firmar.

```
private String signRequest(String sender, String hexBlob) throws Exception {
	List<String> label = new ArrayList<String>();
	List<String> privKey = new ArrayList<String>();
	privKey.add(sender);

	StringEntity request =
CommandTranslator.commandToJson("signrawtransaction", hexBlob, label,
privKey);
	try {
		return
GsontoObjectTranslator.getSignedTransactionInfo(this.fChainQuerier.executeReq
uest(request)).getHex();
	} catch (Exception e) {
		throw new Exception("error ocurred while singing raw
transaction");
	}
}
```

Una vez creados los métodos para crear y firmar la transacción, quedaría crear el método para enviar la transacción, dicho método es *sendConfirmedRawTransaction()*. Este método requiere el número identificativo de la transacción firmada que se desea ejecutar y solicita a la cadena de bloques que ejecuté dicha transacción.

```
public String sendConfirmedTransaction(String hexBlob) throws Exception {
	StringEntity request =
CommandTranslator.commandToJson("sendrawtransaction", hexBlob);
	try {
		return this.fChainQuerier.executeRequest(request).toString();
	} catch (Exception e) {
		throw new Exception("an error ocurred while sending the
assets");
	}
}
```

SEGUIMIENTO DEL TOKEN

Antes de comenzar a crear las funciones que me permiten consultar en la cadena de bloques el registro de transacciones de un token. Tuve que crear dos clases, *AssetTransaction.java* y *AddressBalance.java* para poder traducir las respuestas provenientes de la cadena de bloques.

AssetTransaction es una clase que me permitirá traducir las respuestas de la cadena de bloques a consultar referidas con el registro de transacciones. Gracias a esta clase, podré saber quién fue el creador del token, el primer receptor de una transacción del token, el poseedor actual de un token o el último poseedor de un token antes de que este fuera quemado. Todos estos métodos los implementé en la clase GsonToObjectTranslator y son bastante similares, se basan en ir recorriendo la lista de objetos que me devuelve la cadena de bloques cuando le hago una consulta referente

al registro de transacciones de un token, traduciendo cada elemento y retornando el valor que me convenga.

```
public static String getLastOwner(List<Object> assetTransactions) {
	String address = "";
	if (assetTransactions != null) {
		for (Object assetTransaction : assetTransactions) {
			address = 
formatTransactions(assetTransaction).getAddresses().keySet().toArray()[0].toS
tring();
			if(address.equals(FchainConst.BURN_ADDRESS)) {
				return 
formatTransactions(assetTransaction).getAddresses().keySet().toArray()[1].toS
tring();
			}
		}
	}
	return address;
}
```

Por ejemplo este método, desea obtener el último poseedor de un token que ha sido quemado, se puede observar como voy recorriendo el historial de transacciones hasta que encuentro la dirección de quemado y entonces obtengo la dirección de la cartera que quemó dicho token. El método *formatTransactions()* traduce el objeto a una instancia de la clase *AssetTransaction*.

En la clase *GsonToObjectTranslator* también desarrollé dos métodos que me sirven para traducir las consultas que devuelven el saldo de tokens de una cartera.

```
public static List<AddressBalance> getAddressBalances(List<Object>
addressBalances) {
	List<AddressBalance> balances = new ArrayList<AddressBalance>();
	if(addressBalances != null) {
		for(Object addressBalance :addressBalances) {
			balances.add(getBalance(addressBalance));
		}
	}
	return balances;
}
```

Este método recorre la lista de objetos que me ha devuelto la cadena de bloques y los va transformando a instancias de *addressBalance* para añadirlos a una lista que representa todos los tokens que una cartera tiene en su posesión. El método *getBalance()* traduce cada elemento de la lista utiliza y utiliza gson para transformar las respuestas a objetos de tipo *addressBalance*.

Una vez desarrollado los métodos y las clases que me permitirán desarrollar las funciones que me permitirán realizar un seguimiento sobre los tokens. Creé la clase *FchainTracer.java* que me permitirá:

- Obtener el saldo de una cartera. *getAddressBalances()*
- Obtener el creador de un token. *getTokenCreator()*
- Obtener al primer poseedor de un token tras su creador. *getTokenInitialOwner()*
- Obtener al actual poseedor del token. *getTokenOwner()*
- Obtener al último poseedor de un token quemado. getLastTokenOwner().

Todos los métodos de esta clase son bastante similares a los explicados en las otras clases y considero que no requieren explicación.

CREACIÓN DEL SERVLET

Cuando ya desarrollé todas las funcionalidades que la API necesitaba para comunicarse eficazmente con la cadena de bloques, procedí a crear el servlet al que debía realizar las consultas. Las funcionalidades de este servlet están explicadas en el manual de usuario.

Para desarrollar los métodos que atenderán a las consultas que reciba mi web service, cree la clase *ServiceFchain.java* en el paquete *com.tfg2018.ws.rest.service* y me serví de los repositorios de jersey para desarrollar fácilmente dichos métodos.

De esta clase, quiero destacar que las anotaciones que se ven encima de cada uno de los métodos definen el tipo de consultas que atenderá cada método, su ruta y el objeto que consume y produce.

- *@Post* / *@Get*. Indica si la consulta es de tipo post o tipo get.
- *@Path*. Indica la ruta a la que se debe enviar la consulta, todas los métodos comparten la ruta raíz *http://ip:port/FChainWS/services/Fchain/{nombreMétodo}*.
- *@Consumes/@Produces.* Indica el tipo de objeto que consume la consulta y el tipo de objeto que produce. Todos los métodos de esta API consumen y producen objetos de tipo *APPLICATION_JSON*.

Una vez terminada esta clase, procedí a modificar la información de la clase Fchain.java para introducir la contraseña, el puerto, la dirección de quemado y la dirección administrador del nodo de la cadena de bloques con la que interactuaré con la cadena de bloques que a su vez, es el servidor Tomcat que alojará este servicio web.

5. Desarrollo de las consultas al servicio web desde la interfaz de usuario

Tras dar por terminada la iteración de desarrollo anterior, comencé con la siguiente iteración que consistía en desarrollar las llamadas que ejecutaría desde la interfaz de usuario para comunicarme con la API que acababa de desarrollar.

Lo primero que hice fue desarrollar una clase para ejecutar las solicitudes a la API que había desarrollado en las iteraciones anteriores. Esta clase se llama *RequestExecutioner.java* y pertenece al paquete *com.tfg2018.gui.ApiManager*. Esta clase utiliza dependencias de apache para ejecutar las consultas y tiene dos métodos

principales, uno para la ejecución de consultas post y otro para la ejecución de consultas get. Ambos métodos se llaman igual y solo se diferencian en el número de parámetros.

Para ejecutar las consultas post, lo primero que hago es coger el cuerpo de la consulta y la operación que deseo ejecutar para generar un objeto de tipo HttpPost que luego ejecutaré utilizando un objeto de tipo *CloseableHttpResponse*, acto seguido tan solo debo retornar la respuesta de la consulta.

```
public String executeRequest(StringEntity request, String operation) throws
Exception {
    try {
        this.httppost = new HttpPost(url.concat(operation));
        this.httppost.setEntity(request);
        this.httppost.setHeader("Content-type", "application/json");
        CloseableHttpResponse response = this.httpclient.execute(httppost);
        HttpEntity entity = response.getEntity();
        String answer = EntityUtils.toString(entity);
        response.close();
        return answer;
    } catch (IOException ex) {
        throw new Exception("Post request execution error");
    }
}
```

El método para ejecutar consultas get, funciona de la misma manera pero directamente paso como parámetro el objeto HttpGet, con lo que solo necesito ejecutar la consulta a mi API.

Acto seguido cree la clase *GsonTranslator* esta clase me sirve tanto para traducir objetos a formato JSON que utilizaré en las consultas, como JSON a objetos con los que pueda operar. Esta clase me permite:

El funcionamiento de los métodos de esta clase es bastante similar al funcionamiento de los métodos de la clase que utilizo en la API para el mismo fin. La diferencia que tienen en cuanto a funcionalidad radica en el método que transforma de objetos a JSON.

```
public static String formatJson(Object value) {
    final GsonBuilder builder = new GsonBuilder();
    final Gson gson = builder.create();
    return gson.toJson(value);
}
```

Simplemente paso como parámetro el objeto y utilizando gson, transformo el objeto a JSON, para utilizarlo como cuerpo en mis consultas.

En la interfaz de usuario también utilizo las clases *Token.java* y *KeyPair.java* que además con iguales a las clases con el mismo nombre de la API, para traducir los JSON a un objeto de este tipo. La clase ResponseMessage es un objeto que tiene como parámetro un mensaje de tipo string, y lo utilizo para los métodos en los que el cuerpo es un único parámetro. Los métodos *getKeys(), getMessage()* y *getToken()*. Ejecutan la misma operación cambiando simplemente el tipo de objeto al que traduzco el JSON.

```
gson.fromJson(keyPair, TipodeObjeto.class);
```

el único método diferente es el *getAddressBalance()* este es un método bastante, lo que hago en este método es obtener directamente los nombres de los tokens que posee una cartera.

Tras desarrollar los métodos necesarios para ejecutar consultas a mi API, comencé a desarrollar las consultas que mi interfaz de usuario necesita para cumplir con los objetivos del proyecto.

La clase *GetOperation.java* que está en el mismo paquete que *RequestExecutioner.java*, en esta clase he implementado todos los métodos en los que solicito información sobre la cadena de bloques a mi API. Esta clase me permite:

- Generar un par de claves. *getNewKeyPair()*.
- Obtener al dueño de un token. *getTokenOwner()*.
- Obtener información de un token. *getTokenInfo()*.
- Obtener al último dueño de un token quemado. *getTokenLastOwner()*.
- Obtener al creador de un token. *getTokenCreator()*.
- Obtener al primer dueño de un token. *getTokenInitialOwner()*.
- Obtener el saldo de tokens de una cartera. *getAddressBalances()*.

El único método que ejecuta una consulta de tipo get, es el método *getNewKeyPair()*. Para ejecutar dicha consulta, simplemente concateno la url del servidor tomcat en el que está alojado servicio web con el método que quiero ejecutar y genero una instancia deHttpGet que luego ejecuto con el método *executeRequest()* anteriormente explicado.

```
public KeyPair getNewKeyPair() throws IOException, Exception {
    HttpGet request = new HttpGet(url.concat("createKeyPair"));
    String answer = executeRequest(request);
    return GsonTranslator.getKeys(answer);
}
```

El resto de métodos son parecidos entre sí, simplemente transformo instancias de objetos que quiero pasar como parámetros a JSON, y llamo al método *executeRequest()* pasándole como parámetros el nombre del método que deseo ejecutar en la API y la *StringEntity* que deseo que pase como cuerpo de la consulta *POST* que va a ser ejecutada.

```
StringEntity request = new
StringEntity(GsonTranslator.formatJson(ObjetoATraducir));
String answer = executeRequest(request, "Método");
```

También cree la clase *PostOperation.java* que está en el mismo paquete que *GetOperation.java* en esta clase implementé todos los métodos en los que solicito a la API que efectúe operaciones que necesiten de potencia de minado para ser ejecutadas. Esta clase me permite:

- Validar una dirección en la cadena de bloques. *validateAddress()*.
- Generar un nuevo token. *generateToken()*.
- Efectuar una transacción de un token entre dos carteras. *createInstantTransaction()*.
- Quemar un token. *burnToken()*.

Todos estos métodos se basan en transformar instancias de objetos que quiero pasar como parámetros a JSON, y en llamar al método *executeRequest()* pasándole el JSON que acabo de generar y el nombre del método que quiero que ejecute la API. Básicamente la misma metodología utilizada en los métodos de la clase *getOperation.java*.

Con estas dos clases y sus respectivos métodos desarrollados, ya tenía todo lo que necesitaba para ejecutar todas las llamadas necesarias para efectuar todas las operaciones que ofrece la API que desarrollé y terminé esta iteración del desarrollo.

6. LECTURA, CIFRADO Y REGISTRO DE FACTURAS EN LA CADENA DE BLOQUES

En esta iteración me centré en desarrollar las clases y los métodos necesarios para leer las facturas electrónicas de México en formato xml en su versión cfdv32 cifrarlas a través del algoritmo de encriptación MD5 y generar el respectivo token representativo en la cadena de bloques con toda la información relevante de la factura. Todas las clases que a continuación explico, se encuentran en el paquete *com.tfg2018.gui.factura*.

Antes de proceder a leer la factura, me creé dos clases para poder operar más fácilmente con la información que obtengo de las facturas que leo.

1. *Participants.java*.

Los parámetros de esta clase son los relacionados con los participantes de la factura (Emisor y beneficiario). Los parámetros son el rfc (número de identificación del registro federal de contribuyentes), el nombre del participante y la localización, que tiene como parámetros el país, el estado, el municipio, el código postal y la calle.

2. *ComprobanteInfo.java*

Los parámetros de esta clase con los relacionados con la propia información de la factura. La fecha de generación de la factura, la forma de pago, el tipo de comprobante, el total, el subtotal, el lugar de expedición dela factura y el método de pago.

Una vez creados estas dos clases, cree la clase *InvoiceReader.java* que se ocupará de leer la factura, para posteriormente generar una instancia de la clase *Factura.java* utilizando como atributos la siguiente información generada a partir de la información extraída de la factura:

- 2 instancias de *participants*, uno el emisor y otro el beneficiario.
- 1 instancia de *comprobanteInfo*.
- 2 arrays de *String* para los conceptos de la factura y los impuestos aplicados.

```
public InvoiceReader() throws ParserConfigurationException {
    this.dbf = DocumentBuilderFactory.newInstance();
    this.documentBuilder = this.dbf.newDocumentBuilder();
}
```

Para leer la factura utilizo las dependencias java *DocumentBuilder y DocumentBuilderFactory*. Con estas dependencias parseo los archivos.xml para generar un objeto de tipo Document que luego normalizo para poder operar con él.

```
public Factura readInvoice(File archivo) throws ParserConfigurationException,
SAXException, IOException {
    Document document = this.documentBuilder.parse(archivo);
    document.getDocumentElement().normalize();
…
}
```

Posteriormente en este método me genero las instancias de los objetos que mencioné arriba.

En los métodos *getComprobanteInfo()* y *getParticipantInfo()*, genero un objeto NodeList a partir del nombre de la etiqueta que me convenga para luego ir obteniendo los parámetros necesarios para generar cada instancia a partir de los valores que voy obteniendo a través de los atributos que desee leer.

```
private ComprobanteInfo getComprobanteInfo(Document document) {
    NodeList Comprobante = document.getElementsByTagName("cfdi:Comprobante");
    Element elementComprobante = (Element) Comprobante.item(0);

    String fecha = elementComprobante.getAttribute("fecha");
      …
      …
    return new ComprobanteInfo(fecha, formaDePago, tipoDeComprobante, total,
subTotal, LugarExpedicion, metodoDePago);
}

private Participant getParticipantInfo(Document document, boolean isEmisor) {
    String participant = "";
    String domicilio = "";
```

```
    if (isEmisor) {
        participant = "cfdi:Emisor";
        domicilio = "cfdi:DomicilioFiscal";
    } else {
        participant = "cfdi:Receptor";
        domicilio = "cfdi:Domicilio";
    }
    NodeList Emisor = document.getElementsByTagName(participant);
    Element elementEmisor = (Element) Emisor.item(0);

    String rfc = elementEmisor.getAttribute("rfc");
    String nombre = elementEmisor.getAttribute("nombre");

    NodeList EmisorDetalles = elementEmisor.getElementsByTagName(domicilio);
    Element elementEmisordetalles = (Element) EmisorDetalles.item(0);

    String pais = elementEmisordetalles.getAttribute("pais");
    …
    …
    Localizacion localizacion = new Localizacion(pais, calle, estado,
municipio, CP);

    return new Participant(rfc, nombre, localizacion);
}
```

Los métodos *getConcepts()* y *getImpuesto()* funcionan de una manera un poco diferente. Utilizo dos NodeList en estos métodos, esto lo hago porque la información a la que deseo acceder está un nivel por debajo. Además, el parámetro que deseo obtener en estos métodos esta formado por varios atributos, con lo que debo recorrer dicho parámetro para ir recogiendo la información que me interesa para posteriormente crear un array de *Strings* con esta información y retornarla.

```
private String[] getConcepts(Document document) {
    NodeList Conceptos = document.getElementsByTagName("cfdi:Conceptos");
    Element elementsConcepto = (Element) Conceptos.item(0);

    NodeList Concepto =
elementsConcepto.getElementsByTagName("cfdi:Concepto");
    String[] concepts = new String[Concepto.getLength()];

    for (int temp = 0; temp < Concepto.getLength(); temp++) {
        Node nodo = Concepto.item(temp);
        if (nodo.getNodeType() == Node.ELEMENT_NODE) {
            Element elementConcepto = (Element) nodo;
```

```
            concepts[temp] = " unidad-> " +
elementConcepto.getAttribute("unidad")
                    + ", importe-> " +
elementConcepto.getAttribute("importe")
                    + ", cantidad-> " +
elementConcepto.getAttribute("cantidad")
                    + ", descripcion-> " +
elementConcepto.getAttribute("descripcion")
                    + ", valorUnitario-> " +
elementConcepto.getAttribute("valorUnitario");
        }
    }
    return concepts;
}

private String[] getImpuesto(Document document){
    NodeList Impuestos = document.getElementsByTagName("cfdi:Traslados");
    Element elementsImpuesto = (Element) Impuestos.item(0);

    NodeList Impuesto =
elementsImpuesto.getElementsByTagName("cfdi:Traslado");
    String[] impuestos = new String[Impuesto.getLength()];

    for (int temp = 0; temp < Impuesto.getLength(); temp++) {
        Node nodo = Impuesto.item(temp);
        if (nodo.getNodeType() == Node.ELEMENT_NODE) {
            Element elementConcepto = (Element) nodo;
            impuestos[temp] = "tasa-> " +
elementConcepto.getAttribute("tasa")
                    + ", importe-> " +
elementConcepto.getAttribute("importe")
                    + ", impuesto-> " +
elementConcepto.getAttribute("impuesto");

        }
    }
    return impuestos;
}
```

El método encargado de leerme la factura, llamará a todos estos métodos para ir obteniendo las instancias de los objetos que posteriormente utilizaré como atributos en la instancia que cree de la clase *Factura.java*. en la clase *Factura.java* cree un método que se encarga de retornarme todos los parámetros de la factura en forma de un Map<String, String>, que utilizo para generar la consulta a la API.

Una vez era capaz de leer, extraer y generar una instancia de Factura, comencé a modificar la clase *createTokenStructure.java* que posteriormente traduzco a JSON para utilizarla en la función que ejecuta la consulta post que hace que la API genere un token en la cadena de bloques.

La modificación que le hice a esta clase fue en el constructor, ya que antes le pasaba como atributo el nombre del token, ahora solamente necesito pasarle como atributo los parámetros del token que genero a raíz de la instancia de factura. Lo que hago entonces es llamar a un método que me aplica el algoritmo md5 a este Map<String,String> y generar entonces un identificador único que utilizo como nombre del token que genero en la cadena de bloques.

```
public String generateTokenName(Map<String, String> details) throws
NoSuchAlgorithmException {
    MessageDigest digest = MessageDigest.getInstance("MD5");
    byte[] hash = digest.digest(
            details.toString().getBytes(StandardCharsets.UTF_8));
    return new String(Hex.encode(hash));
}
```

Para utilizar el algoritmo MD5 utilicé el objeto java MessageDigest, al que directamente le paso los detalles de la factura parseados a string y retorna el código hexadecimal de 16 dígitos que necesito para el nombre del token.

Una vez desarrolladas estas funcionalidades, decidí dar por concluida esta iteración de desarrollo.

7. DISEÑO E IMPLEMENTACIÓN DE UNA INTERFAZ BÁSICA DE USUARIO

Para diseñar esta interfaz de usuario, utilicé swing, la herramienta de diseño de interfaces de usuario que ofrece netbeans. Esta herramienta me permite desarrollar una interfaz de usuario simple de manera rápida y efectiva, que cumpla con el objetivo del proyecto de diseñar una interfaz de usuario que permita interactuar con la API que hemos desarrollado, para poder comprobar la efectividad de la cadena de bloques resolviendo un grave problema que atañe a las pequeñas y medianas empresas.

He tratado de crear un diseño simple e intuitivo que sea muy fácil de usar. Considero que con el diseño actual de la aplicación, cualquier usuario sería capaz de entender para que sirve y como se utiliza en poco tiempo. Es un diseño limpio y sencillo con colores claros y sin demasiado contraste entre sí, evitando que el usuario tenga que forzar la vista y se sienta cómodo navegando por el sistema. He decidido utilizar una paleta de colores azules porque el color azul transmite seguridad, que es justamente el punto fuerte de esta aplicación.

Considero que habiendo explicado todo el proceso de desarrollo de la API y de las funciones que ejecutan las consultas a dicha API, no es necesario que profundice demasiado en como funciona este código.

MANUAL DE USUARIO

1. MANUAL DE LA API

A. CREAR UN PAR DE CLAVES

Ruta	**http://209.97.180.81:8080/FChainWS/services/Fchain/createKeyPair**
Body	--
Descripción	Genera una nueva cartera y retorna el par de claves y la dirección de dicha cartera. Esta es la única llamada de tipo Get.

B. VALIDAR UNA DIRECCIÓN

Ruta	**http://209.97.180.81:8080/FChainWS/services/Fchain/validateAddress**

Body	{
	"privkey":{Clave o dirección a validar}
	}

Descripción	Valida una clave o dirección, devuelve los parámetros de la cartera a la que pertenece.

C. GENERAR UN TOKEN

Ruta	**http://209.97.180.81:8080/FChainWS/services/Fchain/generateNewToken**

Body	{
	"address": {Dirección en la que se genera el token},
	"tokenName": {nombre del token},
	"details": {
	{nombre parámetro}:{valor}
	…
	{nombre parámetro}:{valor}
	}
	}

Descripción	Genera un token con los parámetros deseados en la dirección desead.

D. OBTENER INFORMACIÓN DE UN TOKEN

Ruta	**http://209.97.180.81:8080/FChainWS/services/Fchain/getToken**

Body	{
	"message": {nombre del token}
	}

Descripción	Retorna información acerca del token solicitado

E. OBTENER EL DUEÑO DE UN TOKEN

Ruta	**http://209.97.180.81:8080/FChainWS/services/Fchain/getTokenOwner**

Body	{
	"message": {nombre del token}
	}

Descripción	Devuelve la dirección de la cartera que posee el token solicitado

F. OBTENER EL ÚLTIMO DUEÑO DE UN TOKEN

Ruta	**http://209.97.180.81:8080/FChainWS/services/Fchain/getLastTokenOwner**

Body	{
	"message": {nombre del token}
	}

Descripción	Devuelve la dirección de la cartera que tuvo en su posesión el token antes de ser quemado

G. OBTENER AL CREADOR DE UN TOKEN

Ruta	**http://209.97.180.81:8080/FChainWS/services/Fchain/getTokenCreator**

Body	{
	"message": {nombre del token}
	}

Descripción	Devuelve la dirección de la cartera que generó el token

H. OBTENER AL PRIMER DUEÑO DE UN TOKEN

Ruta	**http://209.97.180.81:8080/FChainWS/services/Fchain/getInitialTokenOwner**

Body	{
	"message": {nombre del token}
	}

Descripción	Devuelve la dirección de la cartera que recibió la primera transacción del token

I. EFECTUAR UNA TRANSACCIÓN DE UN TOKEN

Ruta	**http://209.97.180.81:8080/FChainWS/services/Fchain/createInstantTransaction**
Body	{ "addressSender": {dirección del emisor}, "privKey": {clave privada}, "addressReceiver": {dirección del receptor},

	"tokenName": {nombre del token} }
Descripción	Crea, firma y ejecuta una transacción firmada en la cadena de bloques.

J. QUEMAR UN TOKEN

Ruta	**http://209.97.180.81:8080/FChainWS/services/Fchain/burnToken**

Body	{
	"addressSender": {dirección del emisor},
	"privKey": {clave privada},
	"addressReceiver": “”,
	"tokenName": {nombre del token}
	}

Descripción	Crea, firma y ejecuta una transacción firmada en la cadena de bloques con destino dirección de quemado

K. OBTENER EL REGISTRO DE TRANSACCIONES DE UN TOKEN

Ruta	**http://209.97.180.81:8080/FChainWS/services/Fchain/getTokenStackTrace**

Body	{
	"message": {nombre del token}
	}

Descripción	Devuelve el registro de transacciones del token

L. OBTENER EL SALDO DE TOKENS DE UNA CARTERA

Ruta	**http://209.97.180.81:8080/FChainWS/services/Fchain/getAddressBalance**

Body	{
	"message": {dirección de la cartera}
	}

Descripción	Devuelve el saldo de la cartera a la que pertenece dicha dirección

2. MANUAL DE INTERFAZ DE USUARIO

Para comenzar a operar con la interfaz de usuario, lo primero que debemos hacer es iniciar el ejecutable *FchainUserInterface*.

Los usuarios que hay disponibles para probar la aplicación son los siguientes: Sabadell, PcComponentes, Juan.Perez, PharmaPLus, Alberto.Escuela, IngBank y todos utilizan la misma contraseña “password”.

A. FORMULARIO DE INICIO DE SESIÓN

La primera vista que se nos presenta es el formulario de inicio de sesión. En este proyecto no he implementado un sistema de inicio de sesión real, sino que he creado una serie de usuarios preestablecidos con los que podemos interactuar con la cadena de bloques. Los usuarios con los que podemos iniciar sesión son los siguientes:

Ilustración 2: Vista Inicio Sesión

Para iniciar sesión simplemente debemos introducir un nombre de usuario, una contraseña y pulsar en el botón Iniciar sesión.

B. BARRA DE USUARIO

Una vez iniciamos sesión podemos observar que arriba a la derecha aparece nuestro nombre de usuario, si pulsamos en nuestro nombre de usuario o en el icono que se encuentra a la derecha del mismo, el sistema nos mostrará información acerca de nuestra cartera.

Ilustración 3: Botones de gestión de sesión

Si pulsamos en el icono que hay más a la izquierda cerraremos nuestra sesión.

VISTA PRINCIPAL

Cuando iniciamos sesión se nos presenta la vista principal de la aplicación, desde aquí podemos realizar 3 acciones.

Ilustración 4: Vista principal de la aplicación

C. REGISTRAR UNA FACTURA EN LA CADENA DE BLOQUES

Si desde la vista principal pulsamos el botón de registrar factura, se nos abrirá el explorador de archivos, en este momento debemos seleccionar la factura (en formato xml) que queremos registrar en la cadena de bloques. Esta acción debe ser realizada por los deudores de la cadena de bloques. Una vez seleccionada la factura, aparecerá una vista mostrando la información de la factura que estamos a punto de registrar en la cadena de bloques. En este momento, debemos comprobar que efectivamente los datos mostrados por pantalla son correctos.

Una vez realizadas las comprobaciones de que los datos mostrados por pantalla son correctos, debemos pulsar el botón, registrar factura. Una vez hecho esto, se nos mostrará un cuadro de texto en el que deberemos seleccionar de la agenda de contactos, al usuario que es el beneficiario de la factura que deseamos registrar. En cuanto hayamos seleccionado un usuario y aceptado, el sistema generará un token desde nuestra cartera con los datos de la factura que hemos seleccionado desde el explorador de archivos y creará, firmará y ejecutará una transacción de este token a la cartera del usuario que hemos seleccionado desde este cuadro de texto. Esta transacción tarda aproximadamente 15 segundos en confirmarse y ejecutarse, que es lo que tarda en minarse un bloque.

Ilustración 5: Vista de registro de factura

D. CONSULTAR EL ESTADO DE UNA FACTURA FACTURA

Si pulsamos el botón de consultar factura desde la vista principal, se nos abrirá el explorador de archivos, en este momento debemos seleccionar la factura (en formato xml) de la que queremos obtener información de la cadena de bloques para que el sistema nos muestre una nueva vista con información acerca de dicha factura.

En esta vista, se nos muestra el estado de la factura y los actores que han participado en el ciclo de vida de esta.

El estado de la factura puede ser pagada o no pagada. En el caso de que esté pagada, el sistema mostrará quién ha sido el beneficiario final de la factura, en caso de que no esté pagada, el sistema mostrará quién es el que actualmente tiene el derecho de cobro sobre esta factura. Esto resulta muy útil a las empresas, que ahora sabrán en todo momento a quien deben abonar el importe total de la factura.

Esta vista muestra también quien es el beneficiario actual de la factura, lo que resulta muy útil a las entidades financieras que quieran comprobar si la persona que solicita el adelanto de la factura posee el derecho de cobro de dicha factura o no.

Ilustración 6: Vista de Estado de factura

E. MIS FACTURAS

Si seleccionamos la opción mis facturas desde la vista principal, el sistema nos mostrará todas los tokens, facturas que debemos cobrar, que posee la cartera del usuario con el que estamos logueados en el sistema actualmente en una lista. Podemos seleccionar cualquier token de la lista para que el sistema consulte la cadena de bloques y muestre por pantalla toda la información relevante de la factura devuelta por la cadena de bloques. Esta información incluye tanto información acerca de la propia factura, como quien es el usuario deudor de esta. Gracias a esto sabremos en todo momento quien debe abonarnos el importe de la factura.

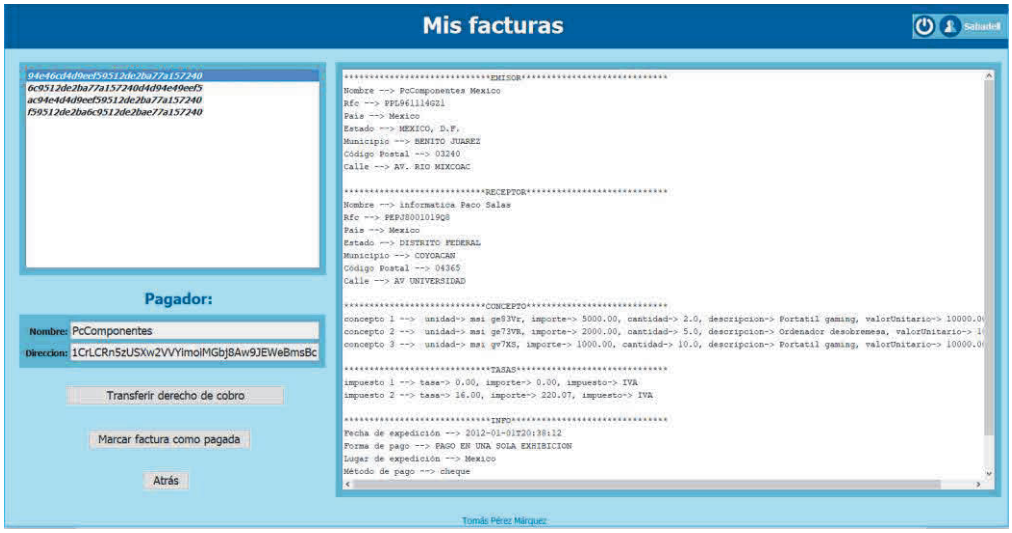

Ilustración 7: Vista mis facturas

NORMATIVA Y LEGISLACIÓN

El hecho de que la información almacenada en la cadena de bloques sea inmutable e inalterable garantiza la seguridad de los datos y la reducción de las vulnerabilidades, esto garantiza el cumplimiento de muchas de las exigencias que impone el Reglamento General de Protección de Datos.

Pese a esto, la cadena de bloques almacena datos personales relacionados con las actividades financieras de los usuarios. Lo que irremediablemente hace que caiga en el ámbito del GDPR.

Existen una serie de artículos en el reglamento general de protección de datos que hacen que la cadena de bloques esté rodeada de incertidumbre.

ARTÍCULO 17

El artículo 17 del GPDR garantiza a los ciudadanos de la Unión Europea el derecho "de borrado", lo que exige a las compañías, en caso de que el ciudadano europeo así lo solicité, paré de operar con sus datos personales y los borré por completo de sus servidores. Esto nos presenta un grave problema que es imposible de solucionar pues, toda la información que se almacene en la cadena de bloques es bien sabido que es inmutable e incorrompible.

ARTÍCULOS 25,32

Pero no todo es malo, la GPDR también exige que medidas extremas de protección de los datos de los ciudadanos europeos, y la cadena de bloques posee uno de los más altos niveles de seguridad y de encriptación de datos del mundo.

ARTÍCULOS 33,34

Exige a las compañías reportar filtraciones de información privada de sus usuarios en un plazo de 72 horas. En el caso de mi blockchain al ser privada, tengo medidas de seguridad contra esto. En mi caso, la cadena de bloques no acepta peticiones de miembros que no participen o que no sean conocidos de la cadena de bloques.

ARTÍCULO 35

Sería necesario desarrollar un sistema de evaluación de impactos para identificar los riesgos para los ciudadanos europeos y esbozar medidas para asegurar que esos riesgos son controlados.

CONCLUSIÓN

En el ámbito legislativo se nos presenta un problema y es que muchos de los principios enunciados en el GDPR, se incide en el "derecho de borrado", y es sabido que la información que se almacena en la cadena de bloques ya es imposible de corromper o eliminar, con lo que no se me ocurre la manera de solucionar este problema.

Me he apoyado en este artículo[12] para escribir esta sección.

ASPECTOS ECONÓMICOS Y TEMPORALES

Este proyecto es un software libre que servirá como una buena herramienta para luchar contra el fraude. Al ser un software libre, no va a generar ingresos.

TRABAJOS FUTUROS

SISTEMA DE REGISTRO E INICIO DE SESIÓN

La siguiente iteración que yo desarrollaría sería la configuración de un servidor que hosteara una base de datos para almacenar información de los usuarios que utilizan la aplicación. Esta base de datos solo tendría una tabla, para almacenar datos de los usuarios.

Una vez que hayamos implementado al base de datos, se podría desarrollar un sistema de registro de usuarios, en el que un cliente se crearía un usuario indicando su nombre de usuario, su correo para implementar un sistema de recuperación de contraseñas y una serie de datos personales.

Cuando el usuario se registra, se le genera una cartera en la cadena de bloques, entonces como atributos del usuario se guardaría la clave pública y la dirección de la cartera en la base de datos, la clave privada sería impresa por pantalla para que el usuario se la apunte en alguna parte. No solo sería mostrada por pantalla, La clave privada se guardaría en el propio ordenador del usuario utilizando un algoritmo de encriptación que requiriera de una clave que solo este conociese.

Cuando el usuario desee iniciar sesión, introducirá su contraseña, lo que desencriptará la clave privada, que podrá ser validada en la cadena de bloques para la cartera asociada a la cuenta de ese usuario. En caso de que el usuario olvidase la contraseña, debería introducir su clave privada manualmente en el sistema, para volver a generar un archivo con la clave privada encriptada.

AGENDA DE CONTACTOS

Como ahora tenemos una base de datos, podríamos añadir una agenda de contactos para que los usuarios no tengan que ir copiando a mano la dirección de la cartera a la que desean enviar un token. Además cada usuario podría tener una página de perfil en la que se muestren los datos personales de la persona que posee la cartera en cuestión.

CREACIÓN DE TOKENS PARA LOS DOCUMENTOS DE CESIÓN DE CRÉDITOS

Para registrar aún mejor el proceso de factoraje en su completitud, podríamos crear un nuevo tipo de token que tuviera como parámetros información relacionada con el documento de cesión de créditos. Indicando si es una cesión total o parcial de la factura, cuanta comisión se está cobrando por el adelanto, etc.

MEJORA DE LA INTERFAZ DE USUARIO

Se debería utilizar otra herramienta para generar una nueva interfaz de usuario que fuera más moderna. Además se podría mejorar el diseño, que ahora mismo es bastante simple.

CREAR VERSIÓN MÓVIL

Utilizaría react-native para crear rápidamente una aplicación nativa que me permita interactuar con mi api desde móviles Android e iOs.

CONCLUSIONES

Las aplicaciones de la cadena de bloques en los procesos de registro y seguimiento de facturas de la empresa pueden proporcionar nuevas herramientas que impulsen el desarrollo de la actividad empresarial. El comercio mundial se basa en el sector de la cadena de suministro estimado en 16 billones de euros, la aplicación de la cadena de bloques en los sistemas de suministros supondría la reducción del fraude y la falsificación que conlleva este tipo de sistemas, y ayudaría a la trazabilidad de una serie de procesos como la gestión de contratos, pagos, etiquetado, sellado y logística entre los diferentes intermediarios hasta llegar al consumidor final.

Por otro lado, el sistema de registros y licencias de negocios podrían otorgar a las administraciones una disminución considerable de costes relacionados con la burocracia del sistema anterior, así como la disminución del tiempo de espera.

Otra aplicación en las organizaciones que actualmente trae de cabeza a la industria financiera es el de préstamos basados en esta tecnología para poder realizar préstamos sin intermediarios, reduciendo numerosos costes y reduciendo riesgos. Gracias a la seguridad que ofrece la cadena de bloques se puede crear un historial crediticio más fiable frente a las entidades financieras que asegura la confianza entre organización y banco.

Ofreciendo facilidades a las organizaciones supondría un mayor desarrollo económico que impulsaría el país y lo colocaría en una de las primeras en esta carrera por dominar los mercados.

REFERENCIAS

- [1] Repositorio de la API

https://github.com/Topema/proyectoFChain

- [2] Repositorio de la GUI

https://github.com/Topema/FChainGui

- [3] Charla Ted acerca de la cadena de bloques.

https://www.ted.com/talks/don_tapscott_how_the_blockchain_is_changing_money_and_business

- [4] Tutoriales Multichain

https://www.multichain.com/developers/

- [5] Comandos Json que se pueden ejecutar en Multichain

https://www.multichain.com/developers/json-rpc-api-2-0-temp/

- [6] Tutorial Gson

https://www.mkyong.com/java/how-do-convert-java-object-to-from-json-format-gson-api/

- [7] Tutorial servlet

https://www.youtube.com/watch?v=ARsfnwfc_Pw&t=0s

- [8] Página web de wupplier

https://www.wupplier.com/es/

- [9] Servicio de administración tributaria Mejicano

https://www.sat.gob.mx/home

- [10] Guía instalación Tomcat

https://www.digitalocean.com/community/tutorials/how-to-install-apache-tomcat-8-on-ubuntu-16-04

- [11] Parámetros archivos de configuración

https://www.multichain.com/developers/blockchain-parameters/

- [12] Artículo GPDR y blockchain

https://es.cointelegraph.com/news/gdpr-and-blockchain-is-the-new-eu-data-protection-regulation-a-threat-or-an-incentive