Título

PABLO Blanco Peris

Máster en INGENIERÍA Informática, Facultad de Informática,

Universidad Complutense de Madrid



Trabajo Fin de Máster en Ingeniería Informática

Fecha

Director:

Adrián Riesco Rodríguez

Autorización de Difusión

Pablo Blanco Peris

Fecha

El abajo firmante, matriculado/a en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: “TÍTULO”, realizado durante el curso académico 20XX-20XX bajo la dirección de XXXX [y con la colaboración externa de dirección de YYYY] en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen en castellano

[Enter abstract here, no longer than 350 words. Be sure to retain the Section Break below.]

Palabras clave

Resumen en inglés

[Enter abstract here, no longer than 350 words. Be sure to retain the Section Break below.]

Keywords

Índice de contenidos

[Autorización de Difusión ii](#_Toc10363295)

[Resumen en castellano iii](#_Toc10363296)

[Palabras clave iii](#_Toc10363297)

[Resumen en inglés iv](#_Toc10363298)

[Keywords iv](#_Toc10363299)

[Índice de contenidos 1](#_Toc10363300)

[Agradecimientos 2](#_Toc10363301)

[Capítulo 1 - Introducción y motivación 3](#_Toc10363302)

[Capítulo 2 - Preliminares 6](#_Toc10363303)

[2.1 Blockchain 6](#_Toc10363304)

[Historia 6](#_Toc10363305)

[¿Qué es blockchain? 7](#_Toc10363306)

[Tipos de blockchain 8](#_Toc10363307)

[Proof-of-work 10](#_Toc10363308)

[2.2 Ethereum 10](#_Toc10363309)

[Gas y pagos en Ethereum 11](#_Toc10363310)

[2.3 Smart Contracts 12](#_Toc10363311)

[Capítulo 3 - Diseño e implementación 14](#_Toc10363312)

[3.1 Front-end 16](#_Toc10363313)

[3.2 Herramientas de conexiones 22](#_Toc10363314)

[Infura 23](#_Toc10363315)

[Web3 25](#_Toc10363316)

[3.3 Back-end o DApp 29](#_Toc10363317)

[Capítulo 4 - Demo 38](#_Toc10363318)

[Capítulo 5 - Conclusión 39](#_Toc10363319)

[Capítulo 6 - Bibliografía 40](#_Toc10363320)

Agradecimientos

The Acknowledgement page is optional. If you include it, retain the Acknowledgements heading and enter your text here. If you do not include it, delete the entire page. Be sure to retain the Page Break that occurs after the List of Table page above.

# Introducción y motivación

A pesar de que el término blockchain nació entre los años 2008 y 2009 por parte de la persona u organización llamada Satoshi Nakamoto [1], empezó a escucharse con más fuerza con la revolución del *bitcoin* y las criptomonedas en el año 2017 en el que llegó a valer 20.000$ cada unidad de la moneda virtual *bitcoin* o también conocida BTC [2].

Además de los aspectos técnicos que representa este concepto es una palabra que está de moda en los últimos años. Aunque blockchain significa “cadena de bloques”, el propio nombre no deja muy claro qué es exactamente. Sin embargo, es un nuevo concepto en la informática que plantea una revolución en una gran cantidad de ámbitos de la vida cotidiana como por ejemplo la economía.

Uno de los principales objetivos de esta tecnología es eliminar a los intermediarios de las transacciones de dinero. Un ejemplo muy sencillo podría ser:

Una persona A quiere enviar dinero a una persona B; una de las soluciones más cómodas que existe a día de hoy sería mediante una transferencia bancaria. Sin embargo, para ello es necesario la intervención de terceros, en este caso los bancos, unas entidades centralizadas de las que se depende a la hora de realizar transacciones de capital de este tipo.

Esta gestión necesariamente cuenta con roles que podrían no aparecer en caso de que esta operación se hubiera llevado a cabo a través de una cadena de bloques, en donde se eliminan los intermediarios y se descentraliza la gestión. Aplicando estas operaciones en una *blockchain* provoca que el control del proceso sea de los usuarios y no de entidades externas.

Se podría entender de manera resumida y sencilla como una enorme base de datos o libro de cuentas en el que todos los registros están entrelazados entre sí y cifrados para mantener la seguridad y privacidad de las transacciones así como la identidad de los propios usuarios que participan en ellas. Para que todo este sistema sea estable, fiable y funcione correctamente necesita una red P2P de nodos comunicados entre ellos. Todos estos nodos contienen la misma *blockchain*, de manera que los bloques que se añaden a la cadena quedan registrados de forma permanente en ella y no pueden ser modificados. Cada vez que se crea un bloque válido nuevo se añade a la cadena y se sincroniza con el resto de nodos que comprueban la validez del mismo. Al depender todos los bloques del anterior se garantiza la seguridad ante un intento de fraude por parte de algún nodo, ya que el resto de nodos no aceptarían el nuevo bloque.

Sin embargo, las transacciones de dinero no son el único ámbito donde puede aplicarse el *blockchain.* La red Ethereum propone una solución en la que las transacciones pueden ser *Smart contracts* o contratos inteligentes, que permiten definir transacciones de distintos tipos sustituyendo productos y servicios que dependen de terceros.

Desde el 2017 la investigación de esta tecnología se ha ido incrementando progresivamente y ya está presente en una gran cantidad de proyectos de diferentes ámbitos. Estos *Smart contracts* ofrecen soluciones útiles a día de hoy y ya se están usando en múltiples ámbitos con diferentes objetivos:

* El gobierno japonés cuenta con un proyecto basado en registros de propiedad a través de *blockchain* [3].
* La conocida aplicación Spotify compró en 2017 una empresa especializada en esta tecnología para abordar una solución en la gestión de pagos y autorías de su plataforma [4].
* En sanidad, la mayoría de tecnólogos y profesionales de la salud ven en las cadenas de bloques la opción ideal para registrar historiales médicos de una manera segura y confidencial [5].
* En servicios públicos y gubernamentales, ya existen plataformas *blockchain* orientadas a contabilizar los votos en unas elecciones presidenciales de manera transparente y evitando fraudes [6].

El objetivo principal de este trabajo es combinar distintas tecnologías punteras actuales para desarrollar un proyecto útil y estable que propone una solución de negocio que puede ser utilizada a día de hoy en un ámbito concreto.

El resultado que se presenta en este proyecto implementa una plataforma completa llamada CarChain que proporciona un servicio de alquiler de coches. A través de un *Smart contract* dentro de la red *blockchain* Ethereum [7] como *back-end*, que se encarga de la gestión de la información dentro de la red de pruebas Rinkeby [8]. La comunicación del usuario con el *back-end* se realiza a través de una aplicación móvil para dispositivos móviles con sistema operativos iOS como *front-end,* que se encarga de gestionar las interfaces visuales, y la interacción y conexión con el *Smart contract*.

La solución de negocio descrita se asemeja a las ya conocidas empresas Car2Go [9] y eMov [10]. Sin embargo, como novedad a estos sistemas existentes se propone una investigación de la optimización del sistema y las ventajas que podría suponer mantener la gestión de la información y los datos del alquiler de los vehículos a través de un contrato inteligente.

Dentro de la plataforma existen dos roles diferenciados, el rol de manager y el rol de usuario. El rol de manager cuenta con la funcionalidad de registrar nuevos coches, pero sin embargo no puede alquilar coches, ya que su objetivo únicamente es gestionar los vehículos de la plataforma a través de la aplicación móvil. El rol de usuario no puede registrar nuevos coches, sus capacidades dentro de la aplicación se limitan a alquiler y devolución de coches y añadir crédito a su cuenta en caso de que quiera alquilar una mayor cantidad de coches. Cada alquiler cuesta un crédito y un usuario solo puede alquilar un coche al mismo tiempo. La aplicación tiene pantallas de *login* y registro de usuarios.

La pantalla principal de la aplicación de un usuario logado consiste en un mapa centrado en la ubicación del usuario siempre y cuando este haya consentido los permisos de ubicación del dispositivo. En el mapa aparecen los coches disponibles alrededor del usuario y se puede elegir cualquier coche disponible. En caso de que el usuario alquile un coche la pantalla del mapa desaparece y se ve una vista que muestra la matrícula del coche alquilado y la opción de devolverlo.

Existe también una pantalla de perfil del usuario en el que muestra su ID de usuario, su dirección de mail, el crédito disponible con el que cuenta y si tiene algún coche alquilado o no. Además hay una pantalla en el que el usuario puede añadir crédito a su cuenta. En el menú principal también está la opción de *logout* en caso de que el usuario quiera salir de su cuenta.

En las siguientes secciones de este documento se detalla toda la funcionalidad de la aplicación y de la gestión de la información tanto de alquiler de vehículos como de gestión de usuarios. También se pueden ver capturas de imágenes de las diferentes secciones y tecnologías con las que cuenta CarChain.

En el capítulo 2 se encuentran descritas las tecnologías utilizadas en este proyecto sin entrar en detalle en como se han usado para la implementación de la plataforma.

En el capítulo 3 se describe el diseño y la implementación del proyecto, es decir, la arquitectura global que sigue el proyecto, se entra en profundidad en las distintas partes del proyecto y los detalles de como se han conectado las tecnologías empleadas para el correcto funcionamiento de la plataforma. También aparecen ejemplos de las interacciones con el *Smart contract* desplegado y con el que se trata toda la gestión de la información acerca del alquiler de vehículos. Se muestran capturas de pantalla de los diferentes IDEs que se han utilizado y se muestran secciones de código de las distintas partes.

En el capítulo 4 se muestra una demo visual de la aplicación así como flujos que sigue y su comportamiento ante distintas situaciones.

En el capítulo 5 se detallan las conclusiones finales una vez implementada la plataforma al completo y probada.

# Preliminares

En esta sección se incluye una descripción detallada de las tecnologías que han sido utilizadas para llevar a cabo la realización del proyecto descrito en este documento.

La tecnología principal en la que se basa este proyecto es *blockchain* [11], también conocida como cadena de bloques. El término blockchain se aplicó por primera vez en 2009 como parte del protocolo Bitcoin [1], aunque no hace mucho que ha empezado a escucharse con fuerza en la sociedad. Sin embargo, la mayoría de la gente no sabe exactamente qué es el blockchain, para qué sirve o cómo funciona. En este trabajo se persigue el objetivo de explicar el blockchain con la mayor claridad posible para que cualquier persona con unos conocimientos básicos en informática sea capaz de entender sus beneficios y los usos que se le puede dar.

Esta tecnología se utiliza dentro de este proyecto para soportar la estructura de datos y la información, a través de un Smart contract [12] en el que se va a mantener la información dentro de la blockchain Ethereum [13], concretamente en Rinkeby [8], una cadena de bloques de *testing* proporcionada por Ethereum enfocada a desarrolladores. Los Smart contracts son scripts que residen en la cadena de bloques que permiten la automatización de procesos de varios pasos.

## Blockchain

### Historia

El concepto de moneda digital descentralizada, así como las aplicaciones alternativas o los registros de propiedad, han existido durante décadas. Los protocolos anónimos de efectivo electrónico de los años 80 y 90, que dependían principalmente de una primitiva criptográfica conocida como el cegamiento de Chaumian [14], proporcionaron una moneda con un alto grado de privacidad, pero los protocolos en gran parte no lograron ganar terreno debido a su dependencia de un intermediario centralizado. En 1998, Wei Dai se convirtió en la primera persona en introducir la idea de crear dinero mediante la resolución de acertijos computacionales así como en un consenso descentralizado [15], pero la propuesta fue escasa en detalles sobre cómo se podría implementar realmente el consenso descentralizado. Posteriormente, en 2005, Hal Finney introdujo un concepto de un sistema para crear un concepto de criptomoneda, pero no alcanzó el ideal al confiar en la computación confiable como *back-end*. En 2009, Satoshi Nakamoto implementó por primera vez una moneda descentralizada en la práctica, combinando las primitivas establecidas para administrar la propiedad a través de la criptografía de clave pública con un algoritmo de consenso para realizar un seguimiento de quién posee las monedas, conocida como prueba de trabajo *o proof of work* [1].

El mecanismo detrás de la prueba de trabajo fue un gran avance en el espacio porque resolvió simultáneamente dos problemas. Primero, proporcionó un algoritmo de consenso simple y moderadamente efectivo, que permite a los nodos de la red acordar colectivamente un conjunto de actualizaciones canónicas del estado del libro mayor de Bitcoin. En segundo lugar, proporcionó un mecanismo para permitir la libre entrada en el proceso de consenso, resolviendo el problema de decidir quién puede influir en el consenso y, al mismo tiempo, evitar los ataques de sibila. Lo hace sustituyendo una barrera formal a la participación, como el requisito de estar registrado como una entidad única en una lista particular, con una barrera económica: el peso de un solo nodo en el proceso de votación por consenso es directamente proporcional a la potencia de cálculo. que trae el nodo. Desde entonces, se ha propuesto un enfoque alternativo *llamado proof of stake*, que calcula el peso de un nodo como proporcional a sus tenencias de moneda y no a recursos computacionales [13].

### ¿Qué es blockchain?

Blockchain es una estructura de datos en la que la información contenida se agrupa en secuencias de bloques a los que se va agregando información relacionada con el bloque anterior, que, a su vez, contiene información relacionada con el bloque anterior a éste y así recursivamente, de manera que, usando técnicas criptográficas de clave pública con un algoritmo de consenso, si un bloque es modificado modificaría todos los bloques posteriores, provocando así su invalidez. Estas cadenas de bloques se reparten entre nodos distribuidos en la red, comúnmente llamados mineros. Esta estructura persigue el objetivo de mantener una descentralización de la información y evitar dependencias con ciertos nodos, de manera que ningún nodo está por encima de otro y ningún nodo es indispensable para el correcto funcionamiento de la cadena, manteniendo así una estructura jerárquica horizontal. El intercambio de información se realiza de manera que, cuando un nodo añade un nuevo bloque, se lo comunica al resto de mineros de la red para que actualicen su cadena de bloques y así contengan todos la misma información.

De esta manera se crea un entorno distribuido en donde la cadena de bloques ejerce como una base de datos pública no con capacidad de asegurar la veracidad del contenido. Constantemente este libro digital de registros va creciendo añadiendo bloques a la cadena, las operaciones se van registrando de manera cronológica en la propia cadena, y se permite realizar un seguimiento sin la necesidad de registros centrales. Es importante destacar que los bloques tienen un orden cronológico dentro de la cadena y esto no se puede modificar ya que los bloques están interrelacionados entre ellos. Por ello, una vez que un bloque es considerado válido dentro de la red de nodos, siempre permanecerá en la cadena de bloques.

La tecnología blockchain es muy adecuada para los entornos en los que se quiera almacenar información de manera cronológica y se quiera garantizar la integridad de la información, imposibilitando la modificación de la información con el objetivo de garantizar una confianza distribuida en lugar de que quede depositada en una entidad certificadora de confianza que se encarga de garantizar la seguridad de las comunicaciones y las transacciones digitales como podría ser una entidad financiera. Sirven para verificar las identidades de los participantes y evitar fraudes mediante claves públicas y privadas [16]. Un ejemplo de autoridad certificadora española es la Fábrica Nacional de Moneda y Timbre (FNMT).

Gracias a las funciones *hash*[[1]](#footnote-1) y la criptografía asimétrica que se emplean es posible implementar un registro contable distribuido, llamado *ledger*, que garantiza la seguridad y la integridad del dinero digital. Es por ello por lo que se está apostando a día de hoy en esta tecnología con las criptomonedas.

A día de hoy es habitual encontrar noticias relacionadas con la tecnología blockchain y las DApps (aplicaciones descentralizadas) y la fuerte inversión que se está movilizando en este sector. Por ejemplo, las DApps movilizaron 6.700 millones de dólares en 2018 [17]. El periódico digital Criptonoticias cubre los hechos más relevantes a las tecnologías Bitcoin ofreciendo noticias, avances y tendencias sobre estos temas [18].

Para evitar una entidad de confianza que centralice la información a la hora de garantizar la integridad de los datos por parte de todos los participantes de la red es necesario seguir un protocolo adecuado para todas las operaciones que se ejecuten sobre la *blockchain*. Gracias a esto se dice que la seguridad y la confianza de la tecnología dentro de todo el sistema se genera, se establece y se consolida por los propios miembros (los mineros). Incluso en un entorno en el que exista una minoría de nodos maliciosos por la red (nodos *sybil*) sería necesario que un atacante cubriera en mayoría la potencia de cómputo y presencia en la red que la que sumaría el resto de nodos combinados.

Gracias a estas características de confianza distribuida y mantenimiento de la integridad de los datos esta tecnología es útil en diferentes escenarios, como por ejemplo:

* Almacenamiento de la información, mediante la replicación de la información de la cadena.
* Confirmación de datos, mediante un protocolo de consenso entre los nodos participantes. El tipo de algoritmo más utilizado es el de prueba de trabajo (en inglés *proof of work*), en el que existe un proceso de validación de los nuevos bloques llamado minería.

Sin embargo, el blockchain está abarcando la mayoría de su peso en el ámbito financiero, en transacciones de dinero digital. Es por ello que no hace mucho comenzó la tendencia de las criptomonedas, que ha tenido y tiene mucha influencia a nivel mundial.

### Tipos de blockchain

Existen diferentes tipos de *blockchain*: *blockchains* públicas, *blockchains* privadas y *blockchains* híbridas. Cada una de ellas tiene características diferentes y, por tanto, tienen usos diferentes entre sí.

#### Blockchains públicas

Las *blockchains* públicas son accesibles para todo el mundo, lo único que se necesita para acceder a ellas es un ordenador y una conexión a internet. Bitcoin fue la primera *blockchain* pública, con la que nació en 2009 esta tecnología [1]. De hecho, a día de hoy, esta criptomoneda es la más fuerte y las más consolidada de todas las *blockchains* públicas que se encuentran en activo.

Las blockchains públicas se caracterizan por ser:

* **Descentralizadas**: evitando de esta manera la necesidad de una entidad central de confianza como podría ser un banco.
* **Distribuidas:** ya que, cada nodo de la red cuenta con una copia exacta de la cadena de bloques.
* **Consensuadas**: hay un consenso generalizado marcado por unas reglas para que las operaciones sean tomadas como válidas. Como por ejemplo un usuario no podría ejecutar una transacción en la que tratara de enviar más dinero del que realmente contiene.
* **Abiertas**: de manera que todo usuario que quiera es libre de participar de manera sencilla descargando el software necesario y realizando transacciones.
* **Seguras**: su seguridad es representada con la “verdad” que se encuentra en la integridad de los datos que se encuentran en ellas, así como la imposibilidad de modificación de datos anteriores.

Las cadenas de bloques públicas son mantenidas por todo aquel que quiera participar. En el caso de los Bitcoin, esto es posible gracias a los mineros, que deben contar con equipos con mucha capacidad de cómputo y por lo tanto gastar electricidad para poder mantenerlo. Sin embargo, esto no lo hacen gratuitamente, sino que existen ciertas recompensas cuando un nodo crea un bloque nuevo y lo añade a la cadena, por lo que se trabaja por incentivos. Esta parte se explicará con más detalle en la sección de Bitcoin.

Este proyecto utiliza la *blockchain* Ethereum, que es una de las *blockchains* públicas más conocidas, ya que ha sido el caso más exitoso de *blockchain* pública por detrás de Bitcoin. Cabe mencionar cuando se habla sobre *blockchains* públicas un grupo de criptomonedas que son mundialmente conocidas como Litecoin [19] o Monero [20], ya que tienen una gran repercusión en el mercado de las criptomonedas y están continuamente moviéndose entre los usuarios de la red a través de transacciones. A este tipo de cadenas también se las conoce como cadenas de bloques sin permisos.

#### Blockchains privadas

Estas cadenas de bloques se caracterizan porque el proceso de consenso que contienen, así como su participación, están limitados. De esta manera, solo ciertos usuarios tienen los derechos necesarios para acceder a ellas.

En estas cadenas es necesario contar con permisos para llevar a cabo transacciones. Por lo que, la lectura de la información de la *blockchain* esté limitada a estos usuarios. Existen diferentes tipos de permisos, por ejemplo pueden existir usuarios que solo sean capaces de leer la información y pueden existir por otro lado usuarios con la capacidad de acceder a la información y de realizar transacciones.

Sin embargo, estas cadenas de bloques dejan de lado inevitablemente la descentralización del poder por lo que ya no se representaría el sistema como totalmente descentralizado.

Dentro de este grupo de *blockchain* existen variaciones, de manera que una cadena de bloques privada puede contar con un nivel de descentralización mayor o menor según la cantidad de entidades o grupos que formen parte del consenso.

Cuanta mayor sea la cantidad de figuras que forman parte del consenso, mayor nivel de descentralización. Por lo que existen *blockchain* privadas con una sola figura o entidad en el consenso que tenga permisos de escritura provocando así una cadena completamente privada y parcialmente centralizada.

Estas cadenas cuentan con las siguientes características:

* **Privadas:** no son accesibles para todo el mundo por lo que para mantenerlas es necesario definir unas entidades preseleccionadas**.**
* **Intereses:** los usuarios que mantienen estas bases de datos lo hacen por intereses propios como puede ser reputación.
* **Jerarquía:** existe una jerarquía de poder y permisos dentro de la cadena por lo que no todos los usuarios tienen por defecto los permisos de escritura. Además, el contenido de los bloques no es accesible para todos los participantes.

Algunas de las *blockchains* privadas más famosas a día de hoy son:

* Hyperledger [21], para la fundación Linux.
* R3 [22], que se trata de un consorcio de bancos a nivel internacional para desarrollar soluciones bancarias a través de una *blockchain* privada.
* Ripple [23], una criptomoneda para realizar transferencias de dinero digital a nivel internacional.

#### Blockchains híbridas

Este tipo de cadenas de bloques son una combinación de las públicas y las privadas. En estas blockchains los nodos que participan han sido invitados previamente, sin embargo, todas las transacciones son públicas. De esta manera, los nodos se encargan de mantener y proporcionar seguridad a esta cadena, a pesar de que las transacciones sean públicas para el resto de usuarios.

Algunos ejemplos de blockchains híbridas son:

* BigchainDB [24], que permite a los desarrolladores y empresas implementar la prueba de conceptos, plataformas y aplicaciones de blockchain con una base de datos de blockchain, admitiendo una amplia gama de industrias y casos de uso.
* Evernym [25], que ofrece la posibilidad de presentar información de identidad de cualquier tipo a cualquier otra persona en el mundo.

### Proof-of-work

La prueba de trabajo implica la búsqueda de un valor que, cuando se aplica el hash, como por ejemplo con SHA-256, el resultado debe comenzar con un número de ceros definido en el convenio, el cual puede ir variando. El trabajo promedio requerido es exponencial con respecto al número de ceros requeridos y se puede verificar ejecutando un solo hash. Una vez que la CPU ha generado un esfuerzo para cumplir con la prueba de trabajo, el bloque no se puede cambiar sin rehacer el trabajo. Como los bloques posteriores se encadenan después de esto, el trabajo para cambiar el bloque incluiría rehacer todos los bloques posteriores.

La prueba de trabajo también resuelve el problema de determinar la representación en la toma de decisiones mayoritaria. Si la mayoría se basara en que una única dirección IP representa un voto, podría ser falsificada por cualquier persona que pueda asignar muchas IPs. La prueba de trabajo es esencialmente de un voto por CPU. La decisión mayoritaria está representada por la cadena más larga, que cuenta con el mayor esfuerzo de prueba de trabajo invertido en ella. Si la mayoría de la potencia de la CPU está controlada por nodos honestos, la cadena honesta crecerá más rápido y superará a cualquier cadena competidora.

Para modificar un bloque anterior, un atacante tendría que rehacer la prueba de trabajo del bloque y todos los bloques posteriores [1].

## Ethereum

Ethereum es una red pública y descentralizada de cadenas de bloques capaz de ejecutar código de programación de cualquier aplicación descentralizada. Esta plataforma permite compartir información con todo el mundo manteniendo su integridad ya que dicha información no puede ser manipulada ni modificada.

El objetivo de esta *blockchain* es crear un protocolo alternativo para la creación de aplicaciones descentralizadas, proporcionando un conjunto diferente de concesiones que pueden ser muy útiles para una gran clase de aplicaciones descentralizadas, con especial énfasis en situaciones en las que el rápido desarrollo, la seguridad para las aplicaciones y la capacidad de diferentes aplicaciones para interactuar de manera muy eficiente son importantes. Ethereum hace esto construyendo lo que es esencialmente la última capa fundamental abstracta: una cadena de bloques con un lenguaje de programación Turing-completo incorporado, que permite a cualquiera escribir contratos inteligentes y aplicaciones descentralizadas donde pueden crear sus propias reglas arbitrarias de propiedad, formatos de transacción y funciones de transición de estado. Los contratos inteligentes, o *Smart contracts*, también se pueden construir sobre la plataforma, con mucho más poder que el que ofrecen las secuencias de comandos de Bitcoin, que no son Turing-completas [26].

Esta *blockchain*, en su conjunto, puede verse como una máquina de estado basada en transacciones. En primer lugar está el estado de génesis, que es el estado inicial de la *blockchain* y con el tiempo se van ejecutando incrementalmente transacciones. Los estados puede incluir información como saldos de cuentas, acuerdos, datos pertenecientes al mundo físico, etc. En resumen, todo lo que actualmente puede ser representado por una computadora es admisible. También existen los cambios de estado no válidos como por ejemplo, reducir el saldo de una cuenta sin un aumento igual y opuesto en otras cuentas, o una transacción de un importe superior al importe que tiene el remitente.

Ethereum tiene una moneda intrínseca denominada *Ether*, también conocida por las siglas ETH. La subdenominación más pequeña del *Ether*, y por lo tanto aquella en la que se cuentan todos los valores enteros de la moneda, se hace llamar *Wei*. Un *Ether* se define como 1018 *Wei*. Existen otras subdenominaciones con sus correspondientes valores respectivos al Ether, sin embargo *Ether* y *Wei* son las más comunes y las más utilizadas [27].

### Gas y pagos en Ethereum

Para evitar problemas de uso indebido de la red todos los cálculos programables en Ethereum están sujetos a tarifas. La tabla de tarifas se especifica en unidades de gas. **Cualquier fragmento de cómputo programable como crear contratos, hacer llamadas a funciones, utilizar y acceder a la cuenta, ejecutar operaciones o realizar transacciones tiene un costo universalmente acordado en términos de gas.** Cada transacción tiene una cantidad específica de gas asociada denominada *gasLimit*. Esta es la cantidad de gas que se compra implícitamente del saldo de la cuenta del remitente. La transacción es considerada inválida si el saldo de la cuenta no puede soportar la compra del gas necesario. Se llama *gasLimit* ya que cualquier gas no utilizado al final de la transacción se reembolsa (a la misma tasa de compra) a la cuenta del remitente. El gas no existe fuera de la ejecución de una transacción.

En general, el *Ether* utilizado para comprar gas que no se reembolsa se entrega a la dirección del beneficiario, la dirección de una cuenta que generalmente está bajo el control del minero. Los operadores pueden especificar el precio del gas que deseen, sin embargo, los mineros pueden ignorar las transacciones que elijan. Por lo tanto, un precio de gas más alto en una transacción le costará más al remitente en términos de *Ether* y le dará un mayor valor al minero y, por ello, será más probable que sea seleccionado por más mineros para ser finalmente el bloque correcto. Por regla general, los mineros elegirán anunciar el precio mínimo del gas para el cual ejecutarán las transacciones y los operadores serán libres de cubrir estos precios para determinar qué precio del gas ofrecerá. Dado que habrá una distribución (ponderada) de los precios mínimos aceptables del gas, los mineros tendrán necesariamente una compensación entre bajar el precio del gas y maximizar la posibilidad de que su transacción se realice de manera oportuna [27].

## Smart Contracts

Los *Smart contracts* son *scripts* que residen en la blockchain de manera que son capaces ejecutar código dentro de la propia cadena bloques una vez desplegados. Estos contratos inteligentes aprovechan las propiedades de las blockchain como la confiabilidad dentro de la cadena y las técnicas criptográficas empleadas para las interacciones con el fin de ofrecer flujos de trabajo distribuidos y automatizados. Esto provoca que los *Smart contracts* sean un punto fuerte en la investigación dentro del dominio del Internet de las Cosas para nuevos casos de uso.

Al pertenecer a la cadena y estar en un bloque concreto estos *Smart* *contracts* tienen una dirección única, la cual se necesita para la conexión y la interacción con ellos. Estos contratos se ejecutan de manera independiente y automática en cada nodo de la red, por lo que cada nodo de la red habilitado para ello ejecuta una máquina virtual con la capacidad de ejecutar el código que se encuentra en ellos. De esta manera la *blockchain* actúa como una máquina virtual distribuida [12].

Los Smart contracts se programan en el lenguaje Solidity, que usan un concepto similar a las clases en los lenguajes orientados a objetos pero en lugar de utilizar la palabra class se usa contract, indicando que se van a crear contratos.

Existen funciones *get* y *set* para leer y escribir en variables, estructuras de datos y tipos enumerados. También puede utilizarse la herencia entre contratos y existe lo que se llama modificadores (de funciones), que sirven para que las funciones comprueben que se cumple cierta condición como requisito para ejecutarse. También cuenta con eventos que se pueden interpretar como disparadores (*triggers*) que reaccionan a algún comportamiento con algún objetivo.

Todos los contratos necesitan indicar en la primera línea de su archivo la versión del compilador para la que está escrito, ya que, al estar esta tecnología en continua evolución es muy cambiante por el momento y salen nuevas versión de compilación cada pocos intervalos de tiempo. Ethereum proporciona un compilador online llamado Remix [28] en el que permite desarrollar *Smart contracts*, probarlos, depurarlos y desplegarlos [29].

Cuando se despliega un contrato o se interactúa con uno ya desplegado en la red se gasta gas, ya que los nodos deben emplear energía de computo para ejecutar las operaciones necesarias y este coste debe ser pagado por el usuario que está interactuando. Existe una tabla de costo de gas dependiendo de la instrucción que se tenga que ejecutar. Este gas se podría ver como una comisión de los nodos por realizar las operaciones necesarias a la hora de desplegar un contrato o de interactuar con él. De esta manera se “paga” a los mineros de la red como recompensa por los recursos utilizados (*hardware*, electricidad y tiempo) [30].

Estos fragmentos de código tienen ciertas ventajas, como por ejemplo: seguridad, confianza y autonomía. Gracias a ellos no es necesario recurrir a terceros para ciertas transacciones. Esto implica evitar la gestión de papeles que ocasionaría un tercero, por lo que optimizan la velocidad de las transacciones. Además es transparente para todas las partes ya que se encuentran en una red pública.

Sin embargo, cuenta con la desventaja o ventaja, según el punto de mira desde el que se enfoque, de ser inmodificable, una vez el contrato esta desplegado no puede cambiarse, es por ello que antes de desplegar el *Smart contract* definitivo tiene que ser analizado muy detalladamente y exhaustivamente probado para evitar cualquier tipo de fallo que pueda ocurrir en un futuro.

Algunas de las soluciones más conocidas, por el momento, para automatizar operaciones y gestiones gracias a este tipo de software son:

* Automatización de pagos.
* Registros y cambios de propiedad.
* Propiedades intelectuales.
* Apuestas.
* Compras automáticas.
* Votaciones.

A pesar del protagonismo que están ganando estos contratos inteligentes hoy en día, queda mucho por evolucionar en esta tecnología para explotar aún más sus cualidades y beneficios, por lo que, con el tiempo se abordarán nuevos casos de uso gracias a las investigaciones e inversiones que existen a día de hoy sobre los *Smart* *contracts*.

En el capítulo 3, en la sección de back-end y DApps se encuentran ejemplos de interacciones con Smart contracts.

# Diseño e implementación

En esta sección se describe técnicamente el diseño completo de la plataforma ofreciendo una visión de la arquitectura del proyecto al completo y las tecnologías que han sido utilizadas para su implementación. Tal y como se ha mencionado anteriormente, se han desarrollado dos secciones bien diferenciadas, que podrían dividirse en dos subproyectos: por una parte el *front-end* y por otra parte el *back-end*.

El *front-end* del proyecto consiste en la parte visual del proyecto, en este caso la aplicación móvil para dispositivos iOS. La totalidad del proyecto ha sido implementado con el lenguaje Swift en el entorno de desarrollo Xcode.

Swift, creado por Apple y presentado en 2014, es un lenguaje de programación orientado a objetos que cuenta con ciertas características que le hacen ser un lenguaje muy potente aunque restrictivo a la hora de programar con el fin de evitar código inseguro. Comprueba automáticamente desbordamiento de enteros y administra automáticamente la memoria a través de ARC (*Automatic Reference Counting*) eliminando la necesidad de utilizar punteros. Este lenguaje está fuertemente tipado y es por ello que permite al propio compilador inferir el tipo de variables en tiempo de compilación sin asignarle un tipo por defecto. Permite enviar funciones o bloques de código como parámetros de otras funciones. Cuenta con un tipo de variable llamado Opcional que obliga al desarrollador a implementar un código seguro a partir de valores por defecto para poder compilar el código [31].

La parte de *back-end* está desarrollada a través de un *Smart contract* en la red *blockchain* de pruebas de Ethereum llamada Rinkeby. El contrato inteligente con el que se realiza toda la gestión de alquiler de coches está desarrollado en el lenguaje de programación Solidity. Solidity es un lenguaje de programación de alto nivel Turing-completo.

Cuenta con una sintaxis similar a *Javascript* y con un tipado estático, que admite herencia y polimorfismo. Los *Smart contracts* se estructuran dentro de este lenguaje de manera similar a la programación orientada a objetos. Dentro de Solidity se utilizan variables y funciones como en la programación imperativa tradicional [32].

La arquitectura global de este proyecto se puede ver representada detalladamente en la siguiente imagen:

Figura 3.1 Arquitectura global del proyecto

En la figura 3.1 se representa la arquitectura global del proyecto se pueden apreciar las diferentes partes que forman el proyecto al completo. Como se ha introducido anteriormente por una parte está la parte de *front-end* y por otro lado está la parte de *back-end*. Sin embargo, este gráfico ignora el tratamiento de usuarios dentro de la app, que se puede considerar una funcionalidad paralela que permanece en segundo plano dentro del proyecto global CarChain.

En primer lugar, en la parte izquierda de la figura está representada la parte de *front-end*, en este caso la app móvil para dispositivos con sistema operativo iOS, donde se encuentra la interfaz de usuario del proyecto. En la siguiente sección se describe detalladamente la totalidad del proyecto de la app, como la arquitectura que sigue el proyecto, las dependencias con las que cuenta para hacer todas las conexiones posibles y las complejidades que existen a la hora de integrar el *framework* de Web3.

En segundo lugar, en el centro de la figura se encuentran las herramientas Infura y Web3. Web3 es el *framework* que se está utilizando en la app que hace posible la conexión con la red Ethereum, en este caso con la red de pruebas Rinkeby. Realmente el framework de Web3 a través de Infura es capaz de conectarse a la blockchain Ethereum. Infura es la herramienta que hace posible la conexión y la comunicación con los nodos de la red, para poder utilizarse es necesario registrarse en la plataforma y cumplir una serie de requisitos como por ejemplo tener un proyecto creado. A través de una serie de *tokens* y *endpoints* Infura hace posible la comunicación con el Smart contract desplegado en la red *blockchain* de Rinkeby.

Por último, en la parte de la derecha de la figura se encuentra la parte de *back-end* del proyecto, que representa lo que se denomina un *DApp*. *DApp* es el acrónimo de *Decentralized Application*, es decir, una aplicación descentralizada. Estas aplicaciones consisten en un código *back-end*, en este caso el *Smart contract* que se ejecuta en una red P2P (peer-to-peer). Este tipo de programas de software no necesitan de terceros de confianza o una autoridad central para funcionar, ya que, permite la interacción directa entre usuarios y proveedores [33].

La gestión de usuarios dentro de la aplicación se trata a través de Firebase de Google [34]. El gráfico de conexión entre la aplicación y esta plataforma, representado en la figura 3.2 es muy parecido al de la figura 3.1 de la arquitectura global del proyecto.

Figura 3.2 Arquitectura del proyecto sobre gestión de usuarios

A través del framework propio de Firebase que se incrusta dentro del proyecto de Xcode se gestionan los usuarios de la aplicación. Esta plataforma realiza el tratamiento de usuarios facilitando el registro de nuevos usuarios y el *login* de los mismos.

Al configurar Firebase en el proyecto a través de la consola de Google se puede ver también estadísticas de la aplicación como por ejemplo usuarios activos, interacción de los usuarios dentro de la app, y la función principal por la que se ha incluido este *framework* en el proyecto: la gestión de usuarios.

## Front-end

La parte de front-end de la plataforma se ha implementado en una aplicación móvil para dispositivos iOS. El proyecto se ha desarrollado a través del IDE Xcode proporcionado por Apple y el lenguaje que se ha empleado ha sido Swift.

Para el desarrollo de la app se han utilizado distintos paradigmas de programación con el objetivo de cumplir con los principios SOLID de la programación orientada a objetos.

Se ha empleado una estructura modular dividiendo los diferentes módulos para hacerlos independientes los unos de los otros.

Los módulos principales enfocados a las distintas funcionalidades que existen dentro de la aplicación son:

* Login.
* Registro de usuarios.
* Mapa.
* Mi perfil.
* Añadir crédito.
* Registro de vehículos.
* Menú lateral.

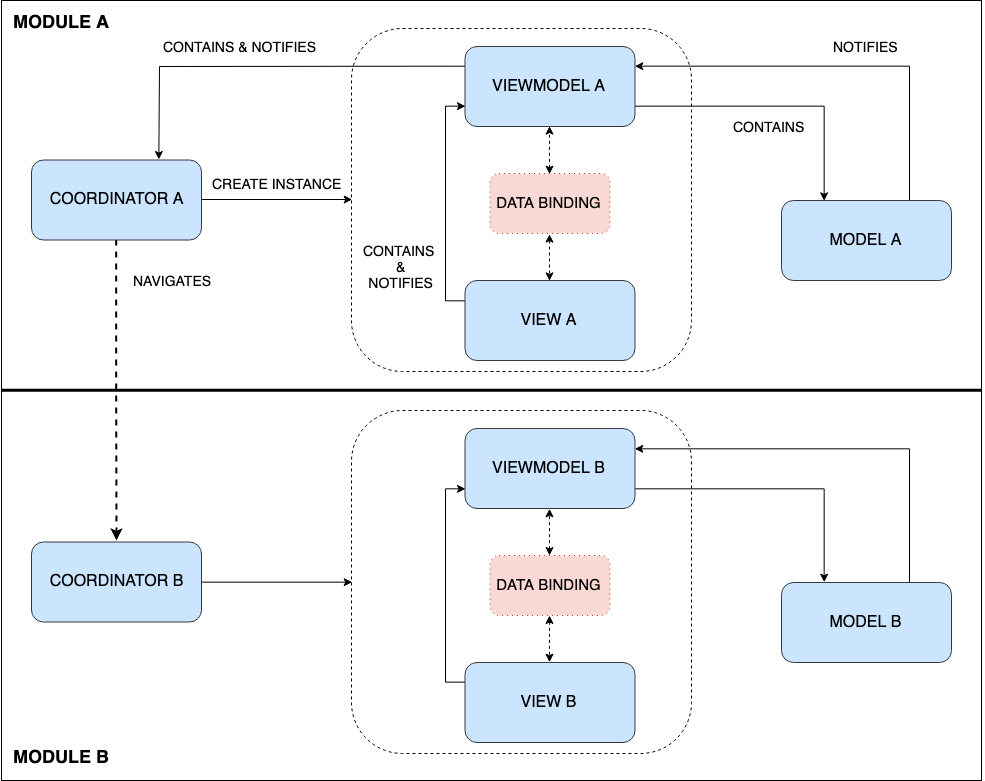
También se han implementado clases base de las que heredan las distintas partes de los módulos para evitar repetición de código y aprovechar la reutilización del mismo.

La arquitectura que se sigue dentro del proyecto sigue las siglas MVVM-C que representan la estructura: *Model*, *View*, *ViewModel*, *Coordinator*. Cada módulo implementado dentro de la app sigue este patrón de diseño. En esta arquitectura destaca la presencia del *ViewModel*, que es la figura que se encarga de acoplar el modelo de datos a la vista. Sobre el *ViewModel* recae la responsabilidad de ejecutar la lógica necesaria para mandar los datos a la vista. Sin embargo el *ViewModel* no sabe nada de la vista, es la vista la que contiene este elemento.

Este patrón de diseño mejora si se acopla la programación reactiva, a través de los frameworks RxSwift y RxCocoa, proporcionados por Apple, a través de los cuales se realiza un *data binding* de los datos entre la *View* y el *ViewModel*. La mayoría de la programación orientada a objetos es imperativa en donde cada vez que se cambia un dato se tiene que avisar a la vista para que se actualice. Sin embargo, con la programación reactiva se automatiza el proceso de eventos y de cambios de datos ya que la vista está continuamente escuchando y reaccionando a los distintos comportamientos y actualizándose de manera asíncrona. Gracias al *data binding* se genera una sincronización automática los elementos de la vista como puede ser un *Label* a la información que debe mostrarse a través de la vista, como puede ser un *String*. De manera que si se sincroniza un *Label* con un *String*, el *Label* siempre está escuchando al *String*, por lo que, si cambia de valor el *String* cambia automáticamente también el *Label* sin necesidad de estar actualizando la vista cada vez que se cambia un valor.

En la figura 3.3 se muestra detalladamente el patrón de diseño seguido dentro de la app.

Figura 3.3 Arquitectura del proyecto en Xcode



El *coordinator* es el que se encarga de crear el módulo de manera que instancia en primer lugar el *ViewModel* y la *View* y le asigna el *ViewModel* a la *View*. Posteriormente dentro de la *View* se realiza el data binding con el *ViewModel*. La vista es la que notifica al *ViewModel* de las interacciones del usuario con la misma. El *ViewModel* contiene al modelo donde se encuentran los datos y al coordinator el cual lo usa para las navegaciones a diferentes módulos. El *coordinator* cuenta también con la responsabilidad de crear nuevos módulos a la hora de navegar instanciando el *coordinator* del módulo destino.

En las siguientes figuras se muestran ejemplos de distintas partes de la arquitectura del módulo.

En la figura 3.4 se muestra un ejemplo de un *coordinator* de un módulo de la app, concretamente del módulo de *login*. Esta capa se encarga de la creación del módulo al completo y de las navegaciones del mismo. Los *coordinator* cuentan con la función viewController() que se encarga de inicializar la *View* y el *ViewModel*, y dos funciones (start() y show()) que se encargan de navegar a la vista dependiendo el estilo de presentación de la misma. Según el módulo el *Coordinator* suele tener funciones de navegación, como se muestra en la figura XX las funciones navigateToMap() y navigateToRegister() que se encargan de instanciar diferentes módulos a través de otros *coordinator*.

Esta capa elimina la necesidad de tratar las navegaciones en la vista mejorando la separación por capas y desacoplando las diferentes partes de la arquitectura. Esto facilita también la reutilización de la vista en diferentes pantallas evitando dependencias de navegaciones a partir de pantallas posteriores o anteriores. Gracias a este patrón de diseño todas las navegaciones de la aplicación se gestionan a través de los coordinator de cada módulo manejando así el flujo de vistas.

A través del *ViewModel* que es la capa donde se maneja la lógica y que contiene al *Coordinator* se acceden a las funciones de esta capa navegando a las distintas pantallas.

Figura 3.4 Coordinator

import Foundation

import UIKit

class CCLoginCoordinator : Coordinator {

static let storyboardId = "Login"

func start(){

navigator?.setViewControllers([viewController()] as [UIViewController], animated: false)

}

func viewController() -> CCLoginViewController {

let vc : CCLoginViewController = UIStoryboard(name: CCLoginCoordinator.storyboardId, bundle: nil).instantiateInitialViewController() as! CCLoginViewController

vc.viewModel = CCLoginViewModel(self)

return vc

}

func show(\_ vc : [UIViewController]){

navigator?.setViewControllers(vc , animated: false)

}

func navigateToRegister(){

CCRegisterCoordinator(navigator).start()

}

func navigateToMap(){

if navigator?.children.count ?? 0 > 0 {navigator?.viewControllers.removeAll()}

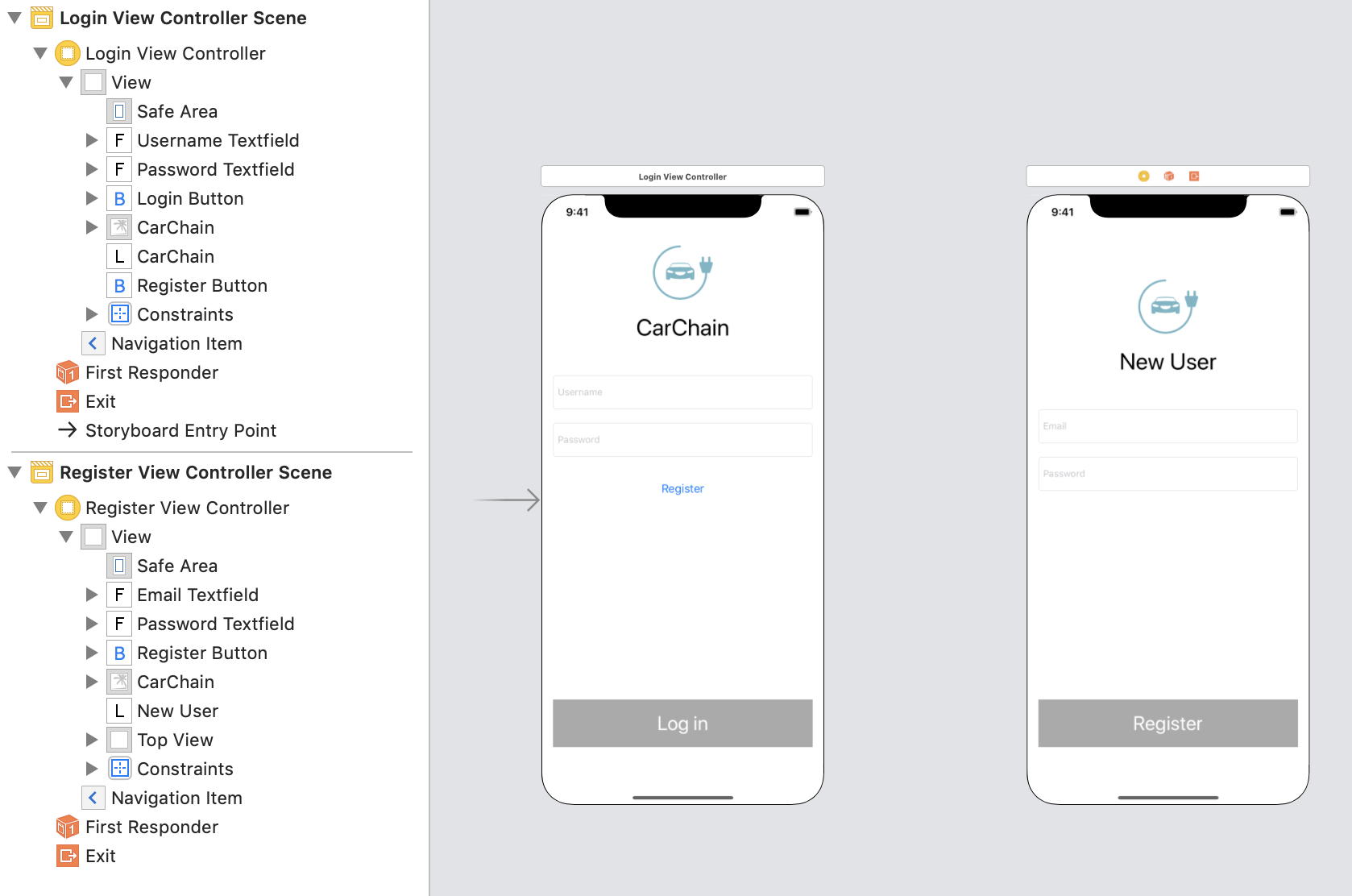
navigator?.pushViewController(UIStoryboard(name: "Main", bundle: nil).instantiateInitialViewController()!, animated: false)

}

}

La *View* se compone de distintos elementos, en primer lugar está la escena que es la parte visual, es decir, las interfaces de usuario. Hay diferentes tipos de ficheros y se instancian de diferentes maneras. En este proyecto se han empleado los ficheros llamados *Storyboards*. En estas vistas se pueden arrastrar distintos elementos que aparecen en la pantalla y configurar cosas simples como puede ser el color de una vista o de un botón. En la figura 3.5 se muestra un ejemplo de un *Storyboard* del proyecto, concretamente el de *login*, en el que aparecen la pantalla de *login* y la de registro de nuevo usuario.

Figura 3.5 View – Storyboard



Desde la escena se asocian los elementos a un *ViewController* que es donde se configuran los elementos de la vista más detalladamente y donde se implementan los comportamientos de las interacciones del usuario como puede ser el pulsado de un botón. El *ViewController* es el controlador de la vista, encargado de tratar los diferentes comportamientos de la vista como pueden ser vistas que se muestras y se esconden. Con la arquitectura que sigue el proyecto de MVVM-C y la programación reactiva en esta capa de la arquitectura se realiza el *data* *binding* entre el *ViewModel* y la propia vista.

En la figura 3.6 se muestra un ejemplo del código de un *ViewController* que implementa la capa de la vista en la arquitectura. El método viewDidLoad() indica que la pantalla ya se ha cargado y ya se pueden configurar los elementos. Se invoca a la función bindViewModel() que se encarga de hacer el *data binding* entre los datos del *ViewModel* y los elementos de la vista. En esta misma función se observan los eventos a los que se quiere escuchar y se invocan funciones del *ViewModel* para tratarlos como por ejemplo el botón de login a través del código: self.viewModel?.loginButtonAction(self) .

Figura 3.6 View - Código

import Foundation

import UIKit

class CCLoginViewController : CCBaseViewController {

var viewModel : CCLoginViewModel?

@IBOutlet weak var passwordTextfield: UITextField!

@IBOutlet weak var usernameTextfield: UITextField!

@IBOutlet weak var loginButton: CCButton!

@IBOutlet weak var registerButton: UIButton!

override func viewDidLoad() {

super.viewDidLoad()

bindViewModel()

}

func bindViewModel(){

viewModel?.username.asObservable().bind(to: usernameTextfield.rx.text).disposed(by: disposeBag)

usernameTextfield.rx.text.orEmpty.bind(to:viewModel!.username).disposed(by: disposeBag)

viewModel?.password.asObservable().bind(to: passwordTextfield.rx.text).disposed(by: disposeBag)

passwordTextfield.rx.text.orEmpty.bind(to: viewModel!.password).disposed(by: disposeBag)

\_ = loginButton.rx.tap.subscribe(){

value in self.viewModel?.loginButtonAction(self)

}

\_ = registerButton.rx.tap.subscribe(){value in self.viewModel?.registerButtonPressed()}

let tap = UITapGestureRecognizer()

view.addGestureRecognizer(tap)

\_ = tap.rx.event.bind(onNext: { recognizer in

self.view.endEditing(true)

})

}

}

La capa del *ViewModel* de esta arquitectura podría verse como un controlador, ya que se encarga de la lógica de negocio. De esta manera se desacopla la lógica del controlador de la vista manteniendo el principio de responsabilidad única. Esta capa debe verse completamente independiente del resto de partes de la arquitectura de manera que si el código de esta parte del módulo se copiase y se pegase en un proyecto de otra plataforma debería realizar el mismo comportamiento siempre y cuando se implementara el *data binding* y los métodos de navegación.

Por regla general el *ViewModel* suele contener al modelo pero existen algunos casos en que no es necesario como el módulo de *login*. También contiene al *coordinator* para poder realizar las navegaciones correspondientes.

En la Figura 3.7 se muestra un ejemplo del *ViewModel* del módulo de *login*. Se pueden las dos variables de tipo String que son el usuario y la contraseña, con los que realiza el *data* *binding* en la *View* con unos campos de texto. Además cuenta con funciones como el pulsado del botón de login que se invoca desde la vista al reaccionar al evento del botón. También puede verse un ejemplo de navegación en la función loginSuccess(\_ user: User) en donde se llama a un método del coordinator para que navegue : coordinator?.navigateToMap().

Figura 3.7 ViewModel

import Foundation

import RxCocoa

import RxSwift

import Firebase

class CCLoginViewModel {

let coordinator : CCLoginCoordinator?

var username = Variable<String>("")

var password = Variable<String>("")

init(\_ coordinator:CCLoginCoordinator?) { self.coordinator = coordinator ; checkUserSaved() }

func checkUserSaved(){

if UserDefaults.standard.bool(forKey: k.UserDefaults.userRegistered){

username.value = UserDefaults.standard.value(forKey: k.UserDefaults.username) as! String

password.value = UserDefaults.standard.value(forKey: k.UserDefaults.password) as! String

}

}

func loginButtonAction(\_ vc : CCLoginViewController){

if !username.value.isEmpty && !password.value.isEmpty{

vc.showLoader()

Auth.auth().signIn(withEmail: username.value, password: password.value) { (user, error) in

vc.hideLoader()

guard let user = user?.user else {self.showError(); return}

self.loginSuccess(user)

}

}

else{ UIAlertController(title: "Error", message: "Data missing", preferredStyle: .alert).show() }

}

func loginSuccess(\_ user: User){

UserDefaults.standard.set(true, forKey: k.UserDefaults.userRegistered)

UserSession.sharedInstance.saveUser(user, username.value, password.value)

coordinator?.navigateToMap()

}

func showError(){

UIAlertController(title: "Error", message: "User/password incorrect", preferredStyle: .alert).show()

}

func registerButtonPressed(){

coordinator?.navigateToRegister()

}

}

El modelo representa la capa de datos, contiene la información pero no realiza ningún tipo de lógica. Normalmente en esta capa se suele llamar a los servicios si son necesarios para obtener la información necesaria. En la Figura 3.8 se muestra un ejemplo de modelo del módulo de mi perfil en el que se realizan las consultas a la información del usuario como por ejemplo el crédito del usuario o la matrícula del vehículo alquilado si es que tiene alguno a través de los métodos performUserCredit() y performUserCar().

Figura 3.8 Model

import UIKit

class CCMyProfileModel: NSObject {

override init() { super.init() }

func performUserCredit(completion :@escaping ( \_ result : [String:Any]) -> () ){

CCSmartContractManager().getUserCredit(UserSession.sharedInstance.user?.uid ?? ""){value in

completion(value)

}

}

func perfomUserCar(completion :@escaping ( \_ result : [String:Any]?) -> () ){

CCSmartContractManager().getUserCar(UserSession.sharedInstance.user?.uid ?? ""){value in

completion(value)

}

}

}

## Herramientas de conexiones

Tal y como se explica previamente, las herramientas de conexiones que se han empleado para conectar el *front-end* con el *back-end* son Infura y Web3. Estas dos herramientas son necesarias para la comunicación con el Smart contract desplegado, pero cada una tiene su función dentro del proyecto. A pesar de que aparezcan en la misma sección en la figura de la arquitectura global Web3 podría incluirse en la parte de *front-end* ya que se implementa dentro de la app y apunta a Infura para poder conectarse a la *blockchain*. Estas dos herramientas cumplen una función importantísima dentro del proyecto CarChain, ya que son las que hacen posible comunicación de la parte de *front-end* con la parte de *back-end*.

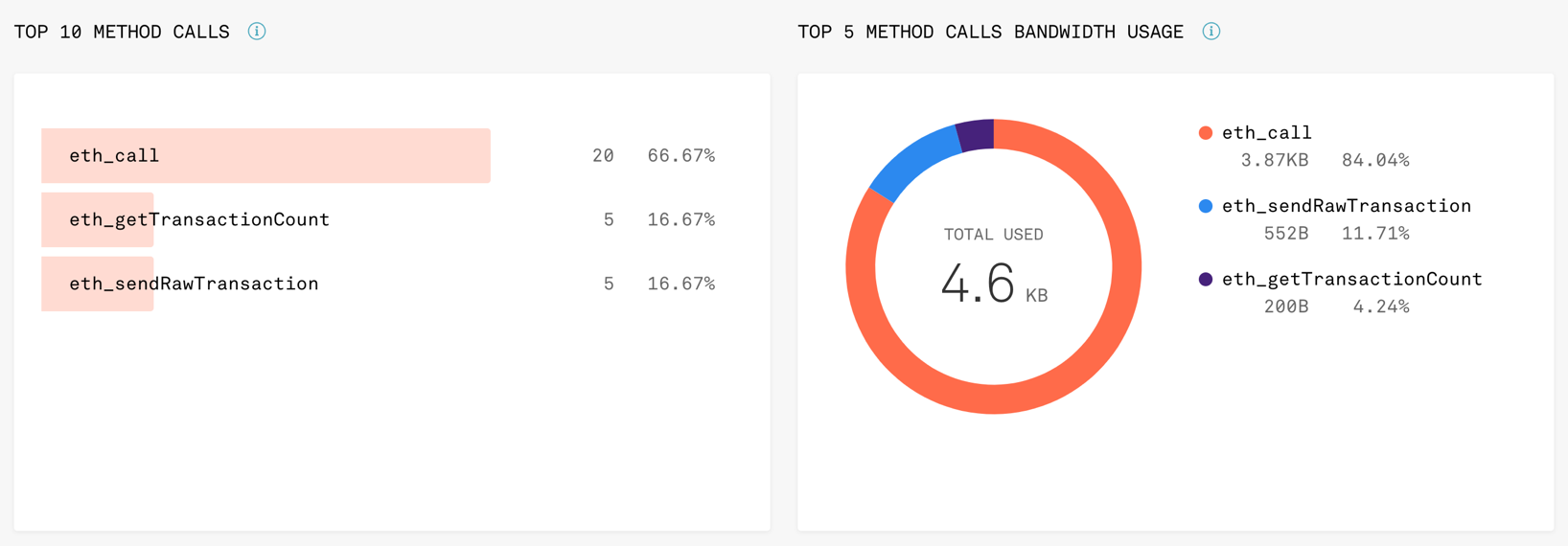
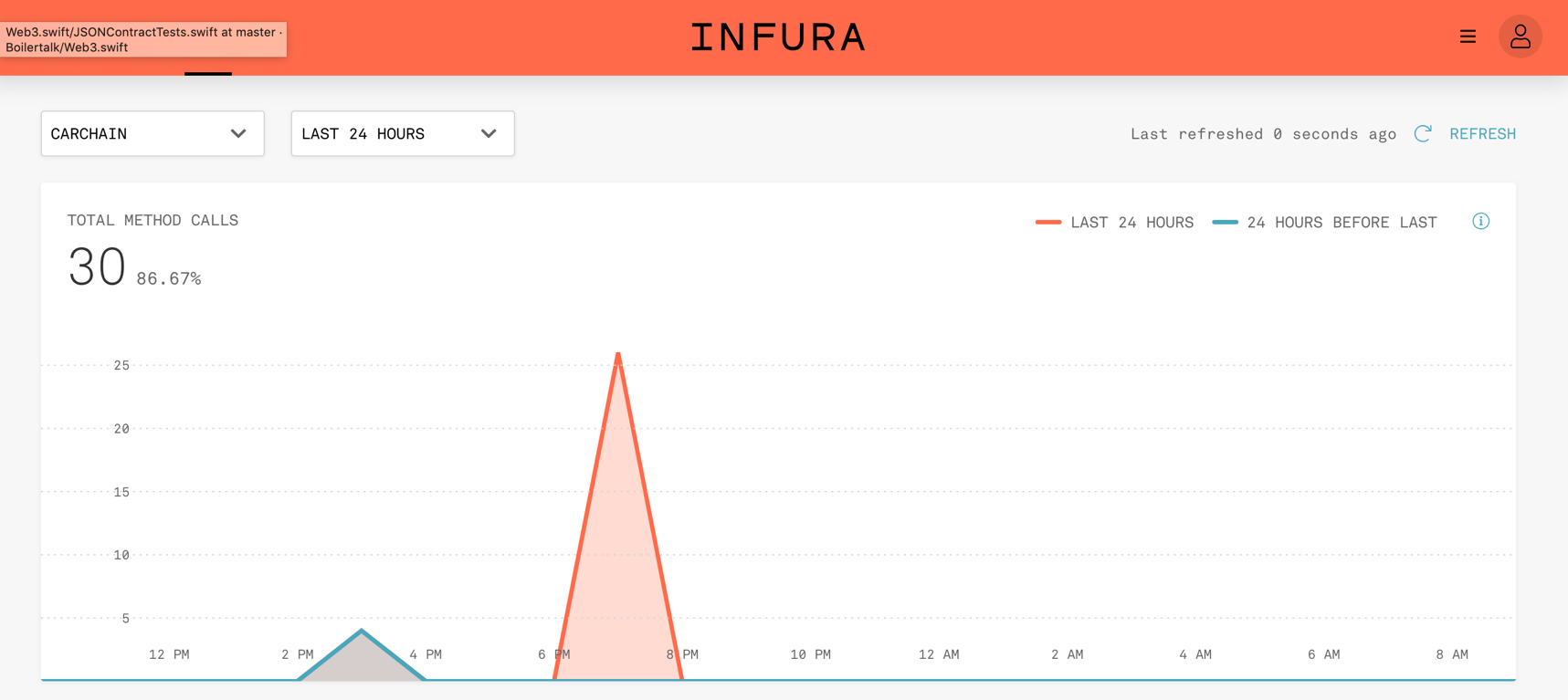
### Infura

Por una parte Infura proporciona acceso seguro y de confianza a Ethereum para que las aplicaciones descentralizadas sin tener que ejecutar localmente un nodo de la *blockchain*. Esta plataforma puede verse como una infraestructura de back-end escalable para la construcción de DApps en Ethereum.

Infura está formada por una colección de nodos completos de Ethereum que permite a los desarrolladores conectarse a través de su interfaz o su API. Para poder utilizarlo es necesario crear una cuenta y crear un proyecto. Al crear un proyecto ofrece diferentes *endpoints* y *tokens* para conectarse con las distintas *blockchains* de Ethereum.

Cuenta con un panel de estadísticas en las que se puede ver la interacción con el proyecto, asociado comúnmente a un *Smart contract*. En la Figura 3.9 se muestra una captura de las estadísticas de Infura del proyecto CarChain.

Figura 3.9 Interfaz de Infura, estadísticas de interacciones con el Smart Contract



Por una parte esta plataforma ofrece gráficas sobre la interacción que ha tenido el *Smart contract* durante periodos de tiempo seleccionables. Por otro lado, Infura proporciona estadísticas del tipo de los métodos que se han utilizado para comunicarse con el contrato, es decir si han sido métodos de consulta, llamados *eth\_call*, métodos de envío de transacción, categorizados como *eth\_sendRawTransaction*, o funciones de consulta del *nonce*, *eth\_getTransactionCount*.

La cantidad de llamadas al método que devuelve el *nonce* es la misma que la cantidad de llamadas a funciones de envío de transacciones, esto se debe a que para ejecutar una transacción es necesario primeramente conocer el *nonce* necesario. El *nonce* es un número aleatorio que se emplea en criptografía dentro de los protocolos de autenticación. En una red *blockchain* pública basada en *proof-of-work* el *nonce* funciona en combinación con el *hash* como un elemento de control para evitar la manipulación de la información de los bloques. Este número aleatorio garantiza que los hash antiguos no se puedan volver a utilizar en lo que se denominan ataques de repetición. Normalmente el *nonce* también puede representar un *timestamp* o una marca de tiempo. [35]

### Web3

Web3 es un *framework* en Swift que permite firmar las transacciones e interactuar con los Smart contracts de la red de Ethereum. Ofrece la posibilidad de conectarse a un nodo de Ethereum por ejemplo a través de Infura para enviar transacciones y leer valores de Smart contracts sin la necesidad de implementaciones propias de protocolos [33].

Para poder utilizar este framework es necesario incorporar al proyecto de Xcode Cocoapods [37], Carthage [38] y Swift Package Manager [39].

Cocoapods, Carthage y Swift Package Manager son herramientas que cuentan con el objetivo de facilitar la incorporación de *framework* y sus dependencias a la app modificando automáticamente archivos de proyecto y configuraciones de compilación. Estos gestores de dependencias hacen posible el uso de la librería Web3.swift dentro del proyecto ya que a pesar de que dentro del proyecto se utilice el módulo Web3, éste necesita de otros frameworks para poder compilarse y ejecutarse correctamente. Para poder utilizar estos gestores de dependencias se tienen que implementar los archivos:

* Podfile: fichero que gestiona Cocoapods, como se muestra en la figura 3.10.
* Cartfile: necesario para el uso Carthage, como se muestra en la figura 3.11.
* Package.swift: archivo que maneja Swift Package Manager, como se muestra en la figura 3.12.

Figura 3.10 Podfile

target 'CarChain' do

platform :ios, '11.0'

use\_frameworks!

# Pods for CarChain

pod 'Web3'

pod 'Web3/PromiseKit'

pod 'Web3/ContractABI'

pod 'RxSwift'

pod 'RxCocoa'

pod 'Firebase/Core'

pod 'Firebase/Auth'

end

Figura 3.11 Package.swift

import PackageDescription

let package = Package(

name: "CarChain",

products: [.library(name: "CarChain",targets: ["CarChain"]),],

dependencies: [.package(url: "https://github.com/Boilertalk/Web3.swift.git", from: "0.3.0")],

targets: [ .target(name: "CarChain", dependencies: ["Web3", "Web3PromiseKit", "Web3ContractABI"]),

.testTarget( name: "CarChainTests", dependencies: ["CarChain"])]

)

Figura 3.12 Cartfile

github "Boilertalk/Web3.swift"

github "attaswift/BigInt" ~> 3.1

github "krzyzanowskim/CryptoSwift" ~> 0.8

github "Boilertalk/secp256k1.swift" ~> 0.1

github "mxcl/PromiseKit" ~> 6.0

La combinación de estos tres archivos permiten utilizar el *framework* Web3.swift necesario para la comunicación de la app con el *Smart contract*. Estas herramientas de integración continua manejan automáticamente las librerías externas y las dependencias de estos módulos manteniendo actualizado el proyecto e integrando los distintos *frameworks* con el objetivo de facilitar el proceso de compilación. De esta manera se automatiza el proceso de descarga y mantenimiento, la compilación y la vinculación de librerías y paquetes. Una vez incluidos en el proyecto los distintos módulos cuentan con una gran facilidad a la hora de usarlos ya que basta con importarlos en la parte del proyecto que quiera usarse. Estos módulos externos suelen tener información incorporada describiendo las variables y las funciones definidas con su objetivo y su uso.

Una vez implementados los requisitos necesarios de dependencias para el uso del *framework*, se ha creado dentro del proyecto de Xcode un fichero llamado SmartContractManager a través del cual se gestiona toda la interacción con el *Smart contract*. Este fichero declara las variables que se van a utilizar en los métodos. Estos métodos hacen llamadas a las funciones del contrato. Las respuestas de estos métodos se devuelven por *callbacks* para ejecutar un bloque u otro en caso de éxito o fallo. Una parte de esta clase se encuentra en la Figura 3.13.

Figura 3.13 Web3.swift dentro de la app

import Foundation

import Web3

class CCSmartContractManager{

let managerAddress = try! EthereumAddress(hex: k.SmartContractData.managerAddress, eip55: false)

let contractJsonABI = k.SmartContractData.contractJsonABI.data(using: .utf8)!

let web3 = Web3(rpcURL: k.SmartContractData.rinkebyEndpoint)

let contractAddress = try! EthereumAddress(hex: k.SmartContractData.contractAddress, eip55: false)

let gasPrice = EthereumQuantity(quantity: 1.gwei)

let myPrivateKey = try! EthereumPrivateKey(hexPrivateKey: k.SmartContractData.privateKey)

lazy var contract = try! web3.eth.Contract(json: contractJsonABI, abiKey: nil, address: contractAddress)

func getUserCredit(\_ userId : String, completion :@escaping ( \_ result : [String:Any]) -> () ){

firstly { contract[k.SmartContractFunctions.getUserCredit]!(userId).call() }

.done { hash in print(hash); completion(hash) }

.catch { error in print(error) }

}

func getUserCar(\_ userId : String, completion :@escaping ( \_ result : [String:Any]?) -> () ){

firstly { contract[k.SmartContractFunctions.getUserCar]!(userId).call() }

.done {

hash in print(hash) ;

completion(hash) }

.catch {

error in print(error) ;

completion(nil) }

}

Al inicio de la clase se encuentran definidas las variables con las que se trabaja en las funciones.

Para poder realizar la comunicación con el Smart contract se necesita tener:

* La dirección de la cuenta desde la que se va acceder.
* ABI del contrato, es decir el binario de la interfaz, necesario para realizar las llamadas a las funciones.
* Una instancia del módulo Web3 con el *endpoint* proporcionado por Infura.
* La dirección del *Smart contract* dentro de la *blockchain.*
* El precio de gas máximo por el que se pagaría.
* La clave privada de la cuenta de MetaMask para poder firmar las transacciones.

Con esto es posible crear una especie de instancia del contrato para poder trabajar con ella en el código.

Posteriormente se encuentran dos funciones de consulta, es por ello que empiezan por la palabra get. Las funciones getUserCredit() y getUserCar() se encargan de obtener la información del usuario. Se invoca el método call() del módulo Web3 y a través de *callbacks* devuelve un *hash* con el valor que devuelve la función del *Smart contract* en caso de que se haya ejecutado correctamente o un error en caso de que algo haya fallado. En la figura 3.14 se encuentran dos métodos que invocan funciones de transacción. Las funciones que muestran el siguiente código son las que llaman a los métodos del Smart contract que se encargan de alquilar un coche y devolver un coche.

Figura 3.14 Web3.swift dentro de la app

func rentCar(\_ plate : String, \_ userId : String, completion :@escaping ( \_ result : EthereumData?) -> ()){

web3.eth.getTransactionCount(address: myPrivateKey.address, block: .latest) {response in

do {

let c = self.contract[k.SmartContractFunctions.rentCar]?(plate, userId)

let transaction: EthereumTransaction = c!.createTransaction(nonce: response.result, from: self.myPrivateKey.address,

value: 0, gas: 210000, gasPrice: self.gasPrice)!

let signedTx: EthereumSignedTransaction = try transaction.sign(with: self.myPrivateKey, chainId: 4)

firstly { self.web3.eth.sendRawTransaction(transaction: signedTx) }

.done {

txHash in print(txHash) ;

completion(txHash) }

.catch {

error in print(error) ;

completion(nil) }

} catch { print(error) }

}

}

func returnCar(\_ plate : String, \_ userId : String, completion :@escaping ( \_ result : EthereumData?) -> ()){

web3.eth.getTransactionCount(address: myPrivateKey.address, block: .latest) {response in

do {

let c = self.contract[k.SmartContractFunctions.returnCar]?(plate, userId)

let transaction: EthereumTransaction = c!.createTransaction(nonce: response.result, from: self.myPrivateKey.address,

value: 0, gas: 210000, gasPrice: self.gasPrice)!

let signedTx: EthereumSignedTransaction = try transaction.sign(with: self.myPrivateKey, chainId: 4)

firstly { self.web3.eth.sendRawTransaction(transaction: signedTx) }

.done {

txHash in print(txHash) ;

completion(txHash) }

.catch {

error in print(error) ;

completion(nil) }

} catch { print(error) }

}

}

Ambas funciones reciben como parámetros la matricula del vehículo y el id del usuario. Dentro de estas funciones primeramente se invoca una función de Web3 que devuelve el valor del *nonce* necesario para invocar posteriormente en el envío de la transacción. Se crea una instancia de la transacción y a partir de ella una instancia de la transacción firmada con la clave privada. Por último con el módulo Web3 se envía la transacción y se espera la respuesta también a través de *callbacks*, que devuelve el *hash* de la transacción en caso que haya sido exitosa o un error en caso de fallo.

## Back-end o DApp

En esta sección se detalla al completo la implementación del *Smart contract* utilizado para la plataforma. También se explica el proceso de compilación y despliegue del mismo sobre la *blockchain* y se muestran ejemplos gráficos de las interacciones con las funciones y del gasto de gas al conectarse con el contrato.

En primer lugar, Ethereum proporciona la *blockchain* Rinkeby como red de pruebas con el objetivo de que desarrolladores puedan probar sus contratos inteligentes sobre ella sin gastar Ethers o Weis reales. A través de la plataforma Remix se pueden crear *Smart contracts*, así como interactuar con ellos, depurarlos y desplegarlos. Por un lado permite utilizar una maquina virtual para probar y depurar sin gastar Ethers y por otro lado también es posible comunicarse con contratos ya desplegados en la *blockchain*. Como esta tecnología es tan versátil a día de hoy debido a que tiene que adaptarse a cambios de manera rápida Remix ofrece distintas versiones de compilación de Solidity, ya que según la versión que se seleccione es posible que el lenguaje haya cambiado y el mismo contrato que compila con una versión anterior no compile con una versión más nueva.

El contrato implementado en este proyecto se ha compilado con la versión 0.4.18. Sin embargo, a día de hoy ya se encuentra disponible en Remix la versión de compilación 0.6.0.

Una parte del *Smart contract* que se usa en este proyecto se puede ver en la Figura 3.15. Este contrato se encarga de toda la gestión del alquiler de vehículos de la aplicación. Implementa las principales funciones necesarias para alquilar o devolver vehículos, registrar nuevos coches o nuevos usuarios y todos los métodos que estas funcionalidades requiere. También maneja la información respecto al crédito del usuario y a los coches que están alquilados por distintos usuarios. En la primera línea aparece la versión de compilación. Al principio del contrato se encuentran las estructuras de datos que se han usado para los coches y los usuarios. Posteriormente, se declaran las variables que se han utilizado, en donde la estructura *mapping* puede entenderse como una mezcla de *array* y diccionario. El constructor del *Smart contract* inicializa las variables al desplegar el contrato sobre la red. A partir del constructor se encuentran las funciones a través de las cuales se realiza todo el tratamiento del alquiler de vehículos, es por ello que existen funciones con el objetivo de registrar nuevos usuarios, nuevos coches, alquilar coches, devolverlos, etc.

Existen dos tipos diferenciados de funciones en los *Smart contracts*. Por un lado están las que devuelven un valor como podría ser consultar un int. Este tipo de funciones gastan gas pero mucho menos que las que necesitan de transacciones. Tienen un return al final que devuelve un valor. Las variables declaradas como public también cuentan con una función get automática que devuelve su valor sin tener que implementarla. Por otro lado están las funciones que asignan valores a las variables, es decir que modifican el estado del contrato realizando cambios sobre sus datos. Este tipo de funciones también llamadas funciones de transacción son más costosas en términos de gas ya que son mas costosas de ejecutar para los nodos.

Figura 3.15 Smart contract del proyecto

pragma solidity ^0.4.18;

contract CarChain {

struct Car {

string plate;

string userId;

bool registered;

}

struct User{

string id;

uint credit;

string carPlate;

bool registered;

}

mapping(string => User) users;

mapping(string => Car) cars;

address manager;

uint public minimumRentCredit;

uint public registeredUsers;

uint public registeredCars;

uint public rentedCars;

uint public availableCars;

function CarChain() public {

manager = msg.sender;

minimumRentCredit = 1;

registeredUsers = 0;

registeredCars = 0;

rentedCars = 0;

availableCars = 0;

}

function registerNewUser(string \_userId) public{

require(!users[\_userId].registered);

User storage newUser = users[\_userId];

newUser.id = \_userId;

newUser.credit = 0;

newUser.registered = true;

newUser.carPlate = "-";

registeredUsers++;

}

function registerNewCar(string \_plate) public restricted{

//Check there is no Car registered with this id

require(!cars[\_plate].registered);

Car storage newCar = cars[\_plate];

newCar.plate = \_plate;

newCar.registered = true;

newCar.userId = "-";

registeredCars++;

updateAvailableCars();

}

function rentCar(string \_plate, string \_userId) public {

require(cars[\_plate].registered);

require(users[\_userId].registered);

require(users[\_userId].credit >= minimumRentCredit);

require(keccak256(cars[\_plate].userId) == keccak256("-"));

require(keccak256(users[\_userId].carPlate) == keccak256("-"));

User storage user = users[\_userId];

user.credit -= minimumRentCredit;

user.carPlate = \_plate;

Car storage car = cars[\_plate];

car.userId = \_userId;

rentedCars++;

updateAvailableCars();

}

function returnCar(string \_plate, string \_userId) public {

require(cars[\_plate].registered);

require(users[\_userId].registered);

require(keccak256(cars[\_plate].userId) == keccak256(\_userId));

require(keccak256(users[\_userId].carPlate) == keccak256(\_plate));

User storage user = users[\_userId];

user.carPlate = "-";

Car storage car = cars[\_plate];

car.userId = "-";

rentedCars--;

updateAvailableCars();

}

function getUserCredit(string \_userId) public view returns (uint){

return users[\_userId].credit;

}

function getUserCar(string \_userId) public view returns (string){

return users[\_userId].carPlate;

}

function updateAvailableCars() private {

availableCars = registeredCars - rentedCars;

}

function addCredit(string \_userId, uint \_credit) public {

require(users[\_userId].registered);

User storage user = users[\_userId];

user.credit += \_credit;

}

modifier restricted(){

require(msg.sender == manager);

\_;

}

}

Una vez probado el Smart contract en la plataforma Remix se ha compilado localmente para desplegarlo posteriormente a la red de Rinkeby. Para ello es necesario trabajar con el lenguaje node a través del cual se compila el contrato y se extrae la interfaz que necesita la blockchain para desplegarlo correctamente.

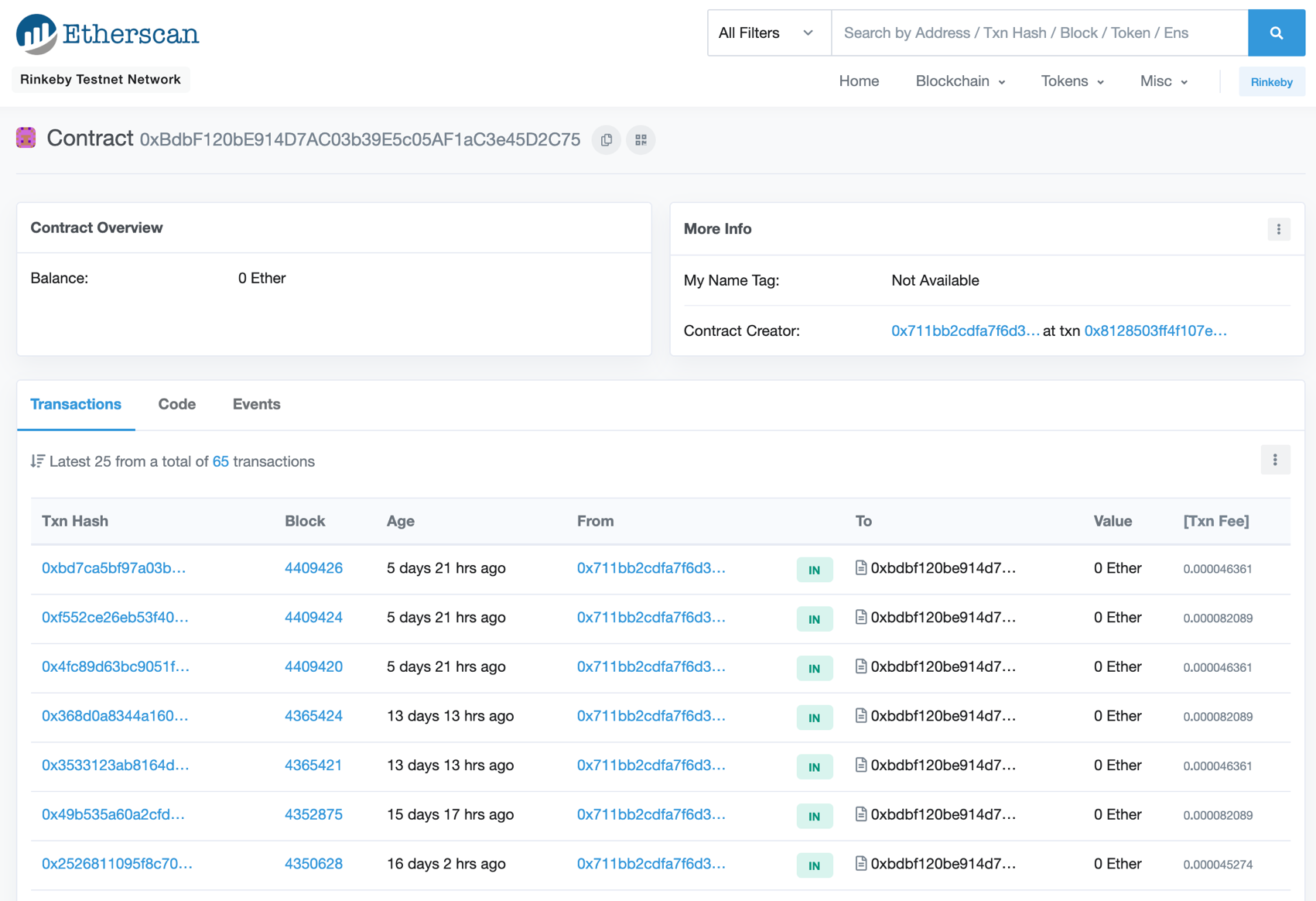
Node necesita ciertas dependencias que deben ser instaladas para la compilación y despliegue de los Smart contracts como por ejemplo:

* Solc: necesario para compilar *Smart* *contracts* en Solidity.
* Web3: que permite interactuar con un nodo Ethereum en remoto para el despliegue del contrato.
* Truffle-hdwallet-provider: necesario para firmar transacciones, es decir, para realizar la transacción del despliegue del contrato. Para ello se debe tener una cuenta en la plataforma MetaMask[40] que puede entenderse como un *wallet* de Ethereum y permite la comunicación con la blockchain a través de navegadores de internet como Chrome o Firefox.

El *Smart contract* se despliega en una dirección de la *blockchain* a través de la cual se puede interactuar con él. La dirección del contrato implementado y desplegado en este proyecto es: 0xbdbf120be914d7ac03b39e5c05af1ac3e45d2c75.

Rinkeby cuenta con un explorador de bloques llamado Etherscan [41] a través del cual se puede buscar una dirección de un *Smart* *contract*, la dirección de una cuenta para ver sus interacciones en la red, *tokens* etc. Si se busca la dirección del *Smart contract* de este proyecto se pueden ver las interacciones que ha tenido desde su despliegue. En la siguiente figura se puede ver un ejemplo de la interfaz que ofrece esta plataforma y la información que ofrece sobre las comunicaciones con los contratos.

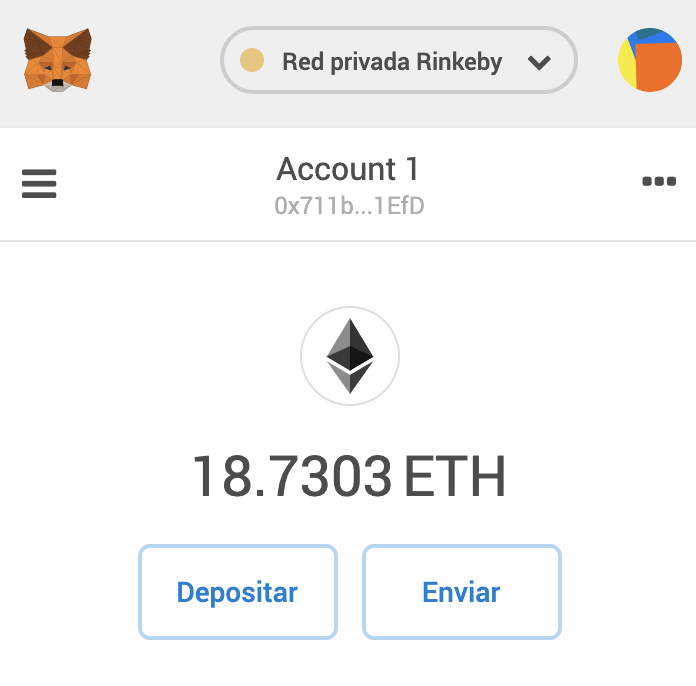
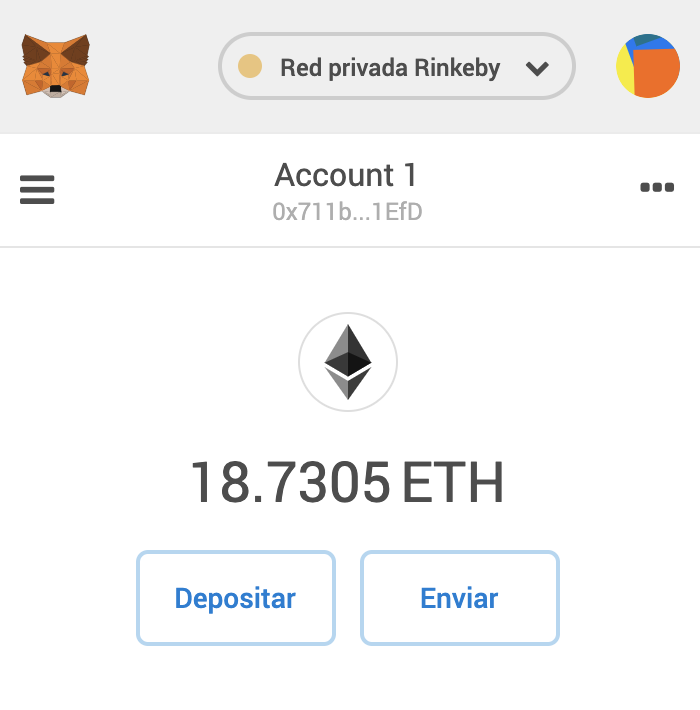
Figura 3.16 Interfaz de Etherscan, registros de la interacción del *Smart contract*



En la figura 3.16 se puede ver información acerca de las transacciones como por ejemplo en que bloque se han incluido, la dirección de la cuenta desde la que se ha interactuado, el *hash* de la transacción o la “multa” de gas que ha costado. Cada vez que se interactúa con el Smart contract, ya sea una función de consulta de datos o una transacción, queda registrado y se puede consultar a través de Etherscan. También puede verse la cuenta desde la que se desplegó el contrato y el *hash* de la transacción mediante la que se llevo a cabo el despliegue.

Todas las transacciones se han ejecutado desde la misma cuenta de MetaMask: 0x711bb2cDfA7f6d3C2D4d2dd167E45D80A4Af1EfD.Esta cuenta fue creada al inicio del proyecto y gracias a la extensión del explorador Chrome puede verse el saldo en Ethers de la cuenta. En las siguientes imágenes se muestra un ejemplo de la interfaz que ofrece el *plugin* y como el saldo disminuye con las transacciones aunque sea sólo con funciones de consulta pagando así la “multa” de gas que corresponde con cada función.

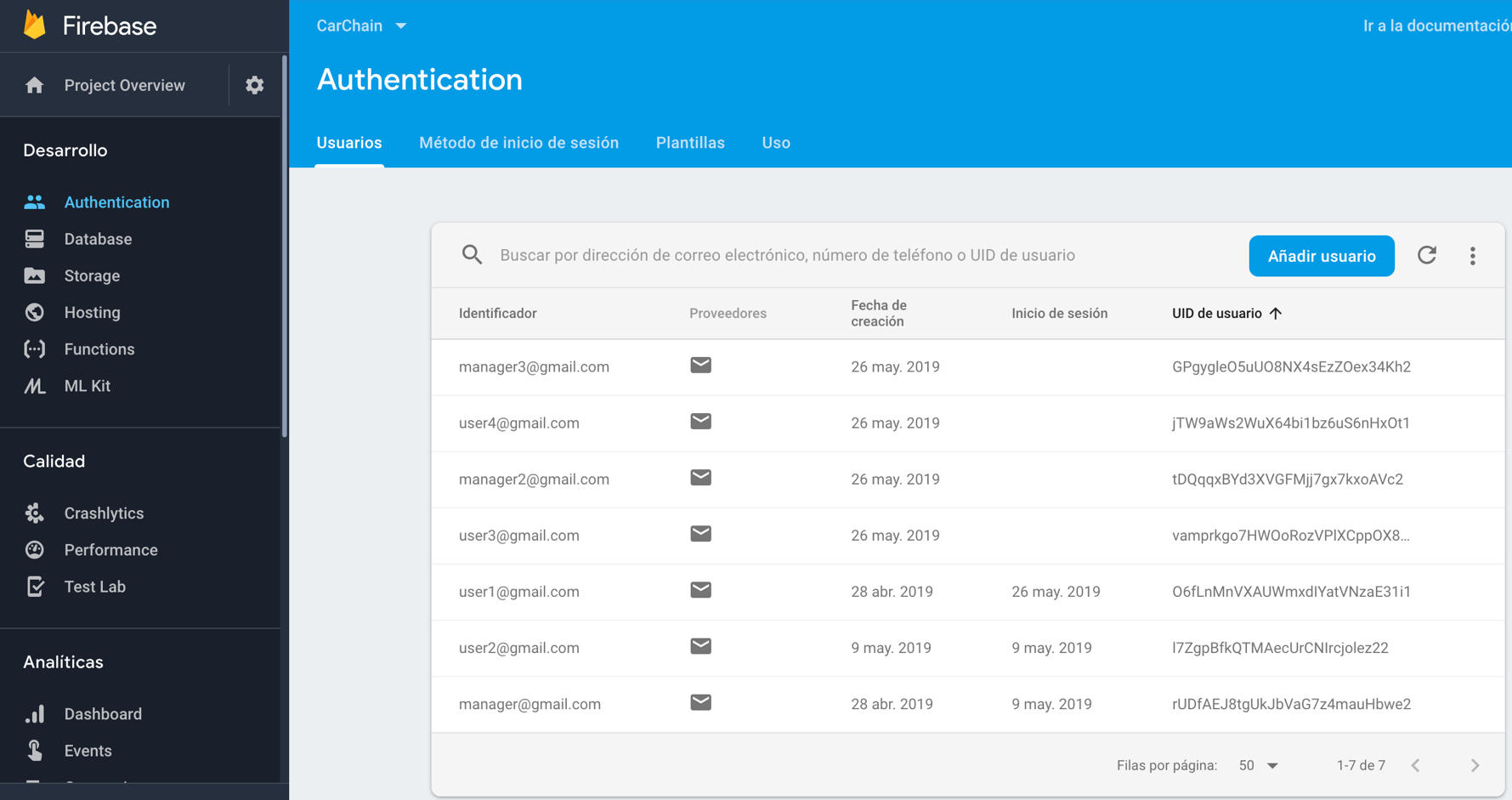
Figura 3.17 Interfaz de MetaMask tras gastar Ethers



En la figura 3.17 aparecen unas imágenes que fueron tomadas durante unas pruebas de la aplicación móvil. La cuenta que aparece en las capturas es la que se ha utilizado para el despliegue de los contratos y para toda su interacción. La imagen de la izquierda se realizó antes de comenzar las pruebas y la segunda imagen al terminar. Es por ello que el saldo de la segunda imagen es inferior a la de la izquierda. Estas pruebas se realizaron a través de la app pero si se quisiera a través de Remix también es posible realizar una prueba del mismo estilo. Para ello se necesita el código del contrato, la dirección en la que se encuentra dentro de la blockchain y una cuenta válida que cuente con Ethers disponibles para ejecutar transacciones.

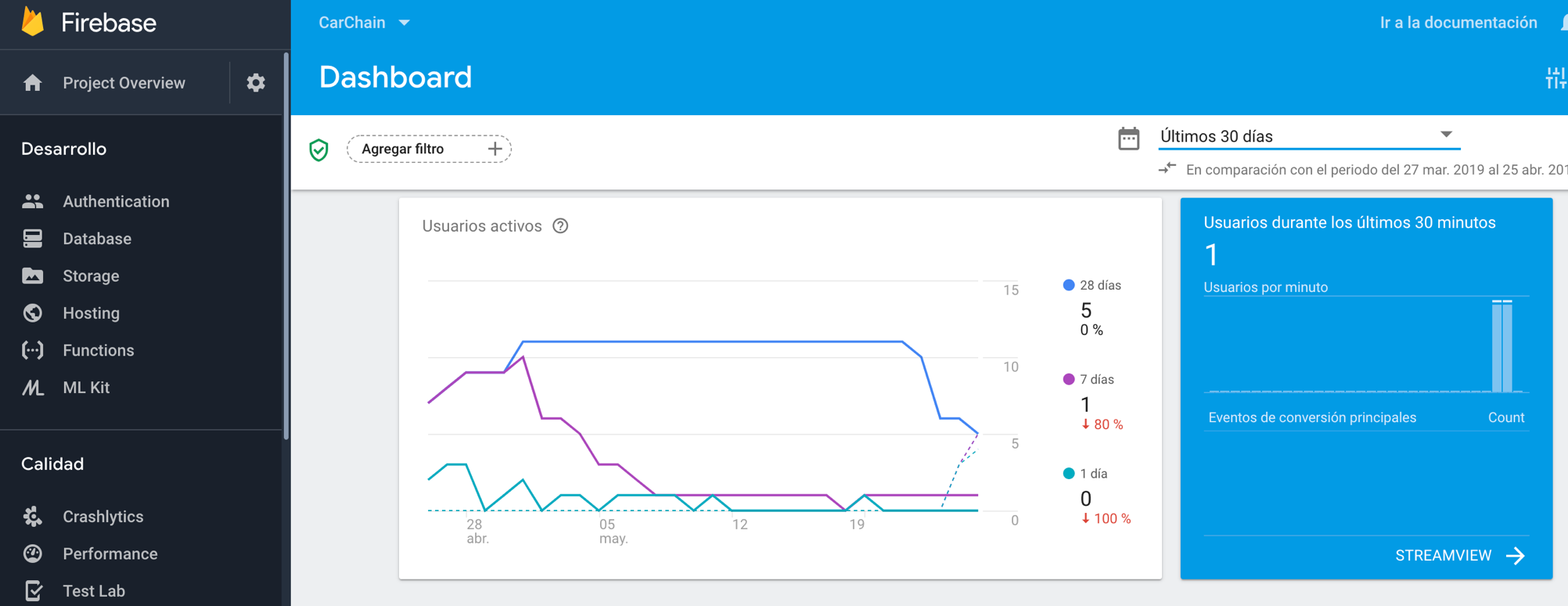
Dentro de la sección de back-end, debe incluirse la gestión de usuarios de la aplicación, la cual se trata a través del *framework* de Firebase [34]. Al igual el módulo de Web3, Firebase también se añade al proyecto a través de Cocoapods. Esta plataforma de Google permite registrar aplicaciones para las que ofrece gratuitamente un soporte de estadísticas, gestión de usuarios, analíticas, informes de fallos, etc.

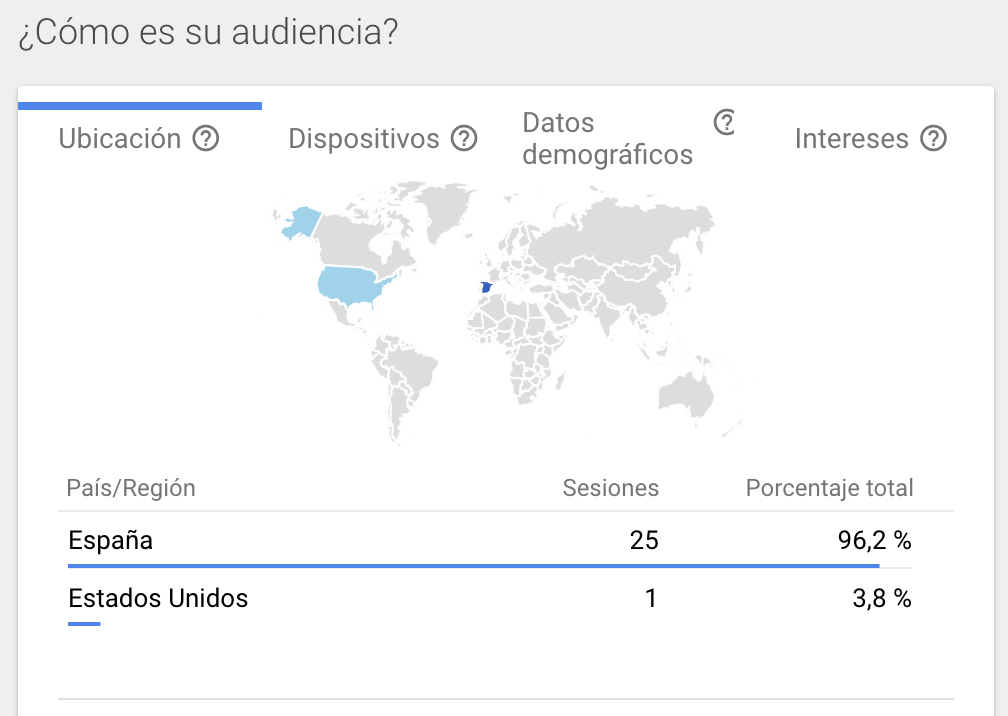
Firebase proporciona la siguiente interfaz para el tratamiento de usuarios:

Figura 3.18 Consola de Firebase gestión de usuarios

En la Figura 3.18 se pueden ver los distintos usuarios registrados, así como su ID de usuario, las direcciones de mail, la fecha de creación o el último inicio de sesión. Como la gestión de usuarios queda en segundo plano dentro de este proyecto se ha utilizado el registro de la manera más sencilla posible a través de la dirección de correo electrónico. Sin embargo Firebase ofrece la posibilidad de registrar usuarios a partir del teléfono móvil o de plataformas como Facebook, Twitter y Google por ejemplo.

Esta plataforma además gestiona automáticamente una gran cantidad de estadísticas útiles relacionada con el uso de la app. Ofrece por ejemplo el número de usuarios activos, la interacción de los usuarios dentro de las pantallas de la aplicación para ver donde pasan más tiempo los usuarios dentro de la app o la audiencia en términos de ubicación a través de un mapa.

Figura 3.19 Analíticas y estadísticas en la consola de Firebase



Las capturas de la figura 3.19 son un ejemplo de las estadísticas que ofrece la incorporación de Firebase dentro de la aplicación. Se envían por defecto a la plataforma a través de la app y se generan automáticamente ahorrando al usuario le gestión manual de las analíticas. Estas analíticas podrían ser útiles de cara a trabajos futuros para mejorar la aplicación utilizando el foco de los usuarios en términos de uso o de ubicaciones.

# Demo

# Conclusión

# Bibliografía

[1] S. Nakamoto, «Bitcoin: A Peer-to-Peer Electronic Cash System», p. 9.

[2] «46 días tomó Bitcoin en saltar de $6900 a $20000 en 2017, ¿Se repite la historia? – CRIPTO TENDENCIA». [En línea]. Disponible en: https://criptotendencia.com/2019/05/14/46-dias-tomo-bitcoin-en-saltar-de-6900-a-20000-en-2017-se-repite-la-historia/. [Accedido: 18-may-2019].

[3] «Gobierno japonés reinventará su registro de propiedades con blockchain», *CriptoNoticias - Bitcoin, blockchains y criptomonedas*, 23-jun-2017. [En línea]. Disponible en: https://www.criptonoticias.com/aplicaciones/gobierno-japones-reinventara-registro-propiedades-blockchain/. [Accedido: 18-may-2019].

[4] J. Pastor, «Spotify sí que cree en blockchain: así funciona Mediachain, la empresa que acaba de comprar», *Xataka*, 08-may-2017. [En línea]. Disponible en: https://www.xataka.com/empresas-y-economia/spotify-si-que-cree-en-blockchain-asi-funciona-mediachain-la-empresa-que-acaba-de-comprar. [Accedido: 18-may-2019].

[5] M. Orcutt, «Blockchain technology will revolutionize medical records—just not anytime soon», *MIT Technology Review*. [En línea]. Disponible en: https://www.technologyreview.com/s/608821/who-will-build-the-health-care-blockchain/. [Accedido: 18-may-2019].

[6] «Transparencia electoral: 5 plataformas blockchain para votaciones», *CriptoNoticias - Bitcoin, blockchains y criptomonedas*, 06-may-2018. [En línea]. Disponible en: https://www.criptonoticias.com/colecciones/transparencia-electoral-5-plataformas-blockchain-para-votaciones/. [Accedido: 18-may-2019].

[7] «Ethereum», *ethereum.org*. [En línea]. Disponible en: https://ethereum.org. [Accedido: 19-may-2019].

[8] «Rinkeby: Ethereum Testnet». [En línea]. Disponible en: https://www.rinkeby.io/#stats. [Accedido: 15-may-2019].

[9] «car2go carsharing España», *car2go*. [En línea]. Disponible en: https://www.car2go.com/ES/es/. [Accedido: 19-may-2019].

[10] «Muévete de forma sostenible e inteligente por Madrid con emov». [En línea]. Disponible en: https://www.emov.eco/. [Accedido: 19-may-2019].

[11] D. Tapscott, A. Tapscott, y J. M. Salmerón, *La revolución blockchain: descubre cómo esta nueva tecnología transformará la economía global*. Barcelona: Deusto, 2018.

[12] K. Christidis y M. Devetsikiotis, «Blockchains and Smart Contracts for the Internet of Things», *IEEE Access*, vol. 4, pp. 2292-2303, 2016.

[13] BUTERIN, Vitalik, «Ethereum white paper». ethereum, 2013.

[14] W. Egbertsen y G. Hardeman, «Replacing Paper Contracts With Ethereum Smart Contracts», p. 35.

[15] C. Wei, J. Luo, H. Dai, Z. Yin, y J. Yuan, «Low-complexity differentiator-based decentralized fault-tolerant control of uncertain large-scale nonlinear systems with unknown dead zone», *Nonlinear Dyn.*, vol. 89, n.o 4, pp. 2573-2592, sep. 2017.

[16] I. Mendívil, «El ABC de los Documentos Electrónicos Seguros», p. 28.

[17] «Dapps movilizaron US$ 6.700 millones en 2018», *CriptoNoticias - Bitcoin, blockchains y criptomonedas*, 18-ene-2019. [En línea]. Disponible en: https://www.criptonoticias.com/comunidad/arte-entretenimiento/dapps-movilizaron-usd-6700-millones-2018/. [Accedido: 15-may-2019].

[18] «Criptonoticias», *CriptoNoticias - Bitcoin, blockchains y criptomonedas*, 27-abr-2019. [En línea]. Disponible en: https://www.criptonoticias.com. [Accedido: 15-may-2019].

[19] «Litecoin - La moneda electrónica». [En línea]. Disponible en: https://litecoin.org/es/. [Accedido: 15-may-2019].

[20] «Monero», *getmonero.org, The Monero Project*. [En línea]. Disponible en: https://getmonero.org/index.html. [Accedido: 15-may-2019].

[21] «Hyperledger – Open Source Blockchain Technologies», *Hyperledger*. [En línea]. Disponible en: https://www.hyperledger.org/. [Accedido: 15-may-2019].

[22] «r3.com», *r3.com*. [En línea]. Disponible en: https://www.r3.com/. [Accedido: 15-may-2019].

[23] «Ripple - One Frictionless Experience To Send Money Globally», *Ripple*. [En línea]. Disponible en: https://ripple.com/. [Accedido: 15-may-2019].

[24] «BigchainDB • • The blockchain database.», *BigchainDB*. [En línea]. Disponible en: https://www.bigchaindb.com/. [Accedido: 15-may-2019].

[25] «The Solution», *Evernym*. [En línea]. Disponible en: https://www.evernym.com/solution/. [Accedido: 15-may-2019].

[26] «Cuestiones básicas de Ethereum | Billetera de Blockchain». [En línea]. Disponible en: https://www.blockchain.com/es/learning-portal/ether-basics. [Accedido: 09-may-2019].

[27] D. G. Wood, «ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER», p. 32.

[28] «Remix - Solidity IDE». [En línea]. Disponible en: https://remix.ethereum.org/#optimize=false&version=soljson-v0.5.1+commit.c8a2cb62.js. [Accedido: 19-may-2019].

[29] «Estructura y elementos de un contrato», *APRENDE BLOCKCHAIN*, 28-feb-2018. .

[30] «¿Qué es y para qué sirve el “Gas” en Ethereum?», *Ethereum*, 25-ene-2018. .

[31] C. G. García, J. P. Espada, B. C. P. G. Bustelo, y J. M. C. Lovelle, «Swift vs. Objective-C: A New Programming Language», *IJIMAI*, vol. 3, n.o 3, pp. 74-81, 2015.

[32] M. Wohrer y U. Zdun, «Smart contracts: security patterns in the ethereum ecosystem and solidity», en *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, Campobasso, 2018, pp. 2-8.

[33] W. Warren y A. Bandeali, «0x: An open protocol for decentralized exchange on the Ethereum blockchain», p. 16.

[34] «Firebase», *Firebase*. [En línea]. Disponible en: https://firebase.google.com/. [Accedido: 26-may-2019].

[35] N. Bozic, G. Pujolle, y S. Secci, «A tutorial on blockchain and applications to secure network control-planes», en *2016 3rd Smart Cloud Networks & Systems (SCNS)*, Dubai, United Arab Emirates, 2016, pp. 1-8.

[36] *A pure swift Ethereum Web3 library. Contribute to Boilertalk/Web3.swift development by creating an account on GitHub*. Boilertalk, 2019.

[37] «CocoaPods.org». [En línea]. Disponible en: https://cocoapods.org/. [Accedido: 25-may-2019].

[38] *A simple, decentralized dependency manager for Cocoa: Carthage/Carthage*. Carthage, 2019.

[39] A. Inc, «Swift.org», *Swift.org*. [En línea]. Disponible en: https://swift.org. [Accedido: 25-may-2019].

[40] «MetaMask». [En línea]. Disponible en: https://metamask.io/. [Accedido: 25-may-2019].

[41] «TESTNET Rinkeby (ETH) Blockchain Explorer». [En línea]. Disponible en: https://rinkeby.etherscan.io/. [Accedido: 25-may-2019].

asdasd

###### Enter Your Appendix Title Here

Appendices must be identified by letters (A, B, etc.) rather than by numbers. For this reason, different style headings are used with appendices. (The style at the top of this page is “Appendix A - Heading 6.”)

First-level Subhead (Heading 7 style)

Within an appendix, Heading 7 is the style to use for all first-level subheads. If you need to add another subhead level within Heading 7, use Heading 8 as shown below.

Second-level Subhead (Heading 8 style)

Use Heading 8 for all second-level subheads within an appendix. If you need to add another subhead level within Heading 8, use Heading 9 as shown below.

Third-level Subhead (Heading 9 style)

If you need a third-level subhead in an appendix, use Heading 4.

Figures and Tables Within Appendices

When you first add a figure or table to an appendix, it will be numbered as though it were in a regular chapter. For example, when the figure below was first inserted, it became “Figure 4.1”. As a figure within Appendix A, it should be “Figure A.1”

To make this change, the codes in the caption labels must be modified, and it’s best to wait until all figures and tables have been added to appendices.

Figure 0.1 First Figure in Appendix A

MP00640_

###### Enter Your Appendix Title Here

If you need additional appendices, use style “Appendix A – Heading 6” for the appendix heading. This will label appendices in alphabetical order (A, B, C, etc.).

1. Una función *hash* es una función computable mediante un algoritmo. Es una operación criptográfica que genera una especie de firma digital de un contenido. [↑](#footnote-ref-1)