Escola Politècnica Superior
d'Enginyeria de Vilanova i la Geltrú

UNIVERSITAT POLITÈCNICA DE CATALUNYA

TRABAJO FINAL DE GRADO

TÍTULO:	Plataforma de compra y venta de entradas para eventos integrada en la tecnología blockchain
AUTORES:	Granados Romero, Albert; Marches Parra, Josep
FECHA:	Octubre, 2022

APELLIDOS:	GRANADOS ROMERO	**NOMBRE**:	ALBERT
TITULACIÓN:	GRADO EN INGENIERÍA INFORMÁTICA		
PLAN:	2018		
DIRECTOR:	LLORENS GARCÍA, ARIADNA MARIA		
DEPARTAMENTO:	DEPARTAMENTO DE ORGANIZACIÓN DE EMPRESAS		

APELLIDOS:	MARCHES PARRA	**NOMBRE**:	JOSEP
TITULACIÓN:	GRADO EN INGENIERÍA INFORMÁTICA		
PLAN:	2018		
DIRECTOR:	LLORENS GARCÍA, ARIADNA MARIA		
DEPARTAMENTO:	DEPARTAMENTO DE ORGANIZACIÓN DE EMPRESAS		

CALIFICACIÓN DEL TFE

TRIBUNAL

PRESIDENTE	**SECRETARIA**	**VOCAL**
COLOMER MUR, JOSE MARIA	BERBEGAL MIRABENT, JASMINA	LOPEZ GARCIA, MARIANO

FECHA DE LECTURA:

Este proyecto tiene en cuenta aspectos medioambientales: ☐ **Sí** ☑ **No**

RESUMEN

Desde la llegada de internet tal y como la conocemos, el sector de las plataformas de ventas de tickets para eventos cómo conciertos, festivales, teatro o deportes, tienen algunos desafíos a nivel de seguridad y fiabilidad con los que a día de hoy siguen teniendo que paliar.

La falsificación, la especulación y el fraude han sido tareas sencillas en el sector y en la actualidad seguimos viendo casos con demasiada frecuencia.

Tecnologías de almacenamiento de datos y de seguridad en internet novedosas como la blockchain pueden ayudar a crear un mercado de venta de entradas mucho más fiable y eficaz.

Hemos aprovechado las características de la blockchain con el objetivo de conseguir que las entradas para eventos no puedan ser falsificadas, duplicadas o revendidas por terceros a precios abusivos, evitando así el fraude que es frecuente en este tipo de servicios.

Para demostrarlo, hemos desarrollado un prototipo de plataforma de compra y venta de entradas para eventos integrada en la tecnología blockchain, a la que llamamos Tickbit.

El prototipo consta de una plataforma web para clientes y un apartado de gestión para las empresas promotoras de eventos, ambas completamente integradas en la blockchain. También consta de un sistema de validación para que las empresas puedan validar las entradas de los clientes.

Hemos podido crear una base sobre la cual empresas de venta de entradas para eventos desarrollarán el futuro de las plataformas de “ticketing”. Los resultados han sido muy positivos, y logramos que el dueño de la entrada tenga la fiabilidad de que su entrada es única, e intransferible.

La tecnología blockchain, eso sí, es aún poco accesible para perfiles de clientes no-técnicos y necesita maduración, estabilización y estandarización para poder funcionar óptimamente en un entorno realista.

Palabras clave (máximo 10):

Cadena de bloques	Contrato inteligente	NFT	Plataforma
Entrada	Aplicación web	Polygon	DApp

RESUM

Des de l'arribada d'internet tal com la coneixem, el sector de les plataformes de vendes de tiquets per a esdeveniments com concerts, festivals, teatre o esports, tenen alguns desafiaments a nivell de seguretat i fiabilitat amb els quals avui dia continuen havent de pal·liar.

La falsificació, l'especulació i el frau han estat tasques senzilles en el sector i en l'actualitat continuem veient casos amb massa freqüència.

Tecnologies d'emmagatzematge de dades i de seguretat en internet noves com la blockchain poden ajudar a crear un mercat de venda d'entrades molt més fiable i eficaç.

Hem aprofitat les característiques de la blockchain amb l'objectiu d'aconseguir que les entrades per a esdeveniments no puguin ser falsificades, duplicades o revenudes per tercers a preus abusius, evitant així el frau que és freqüent en aquesta mena de serveis.

Per a demostrar-ho, hem desenvolupat un prototip de plataforma de compra i venda d'entrades per a esdeveniments integrada en la tecnologia blockchain, a la qual anomenem Tickbit.

El prototip consta d'una plataforma web per a clients i un apartat de gestió per a les empreses promotores d'esdeveniments, ambdues completament integrades en la blockchain. També consta d'un sistema de validació perquè les empreses puguin validar les entrades dels clients.

Hem pogut crear una base sobre la qual empreses de venda d'entrades per a esdeveniments desenvoluparan el futur de les plataformes de “ticketing”. Els resultats han estat molt positius, i aconseguim que l'amo de l'entrada tingui la fiabilitat que la seva entrada és única, i intransferible.

La tecnologia blockchain, això sí, és encara poc accessible per a perfils de clients no-tècnics i necessita maduració, estabilització i estandardització per a poder funcionar òptimament en un entorn realista.

Paraules clau (màxim 10):

Cadena de blocs	Contracte intel·ligent	NFT	Token no fungible
Entrada	Aplicació web	Polygon	DApp

ABSTRACT

Since the arrival of the internet as we know it, the sector of ticket sales platforms for events such as concerts, festivals, theater or sports, have had some challenges in terms of security and reliability with which they still have to deal today.

Falsification, speculation and fraud have been easy tasks in the industry and today we still see cases very frequently.

New internet security and data storage technologies such as blockchain can help create a much more reliable and efficient ticketing marketplace.

We have taken advantage of the characteristics of the blockchain with the aim of ensuring that tickets for events cannot be counterfeited, duplicated or resold by third parties at abusive prices, preventing the fraud that is frequent in this type of service.

To demonstrate this, we have developed a prototype platform for buying and selling tickets for events integrated in blockchain technology, called Tickbit.

The prototype consists of a web platform for clients and a management section for events companies, both fully integrated into the blockchain. It also consists of a validation system so that companies can validate customer entries.

We have been able to create the basis upon which event ticketing companies will build the future of ticketing platforms. The results have been very positive, and we have achieved that the owner of the ticket has the confidence that his ticket is unique and non-transferable.

However, blockchain technology is still not very accessible to non-technical customer profiles and needs to mature, stabilize and standardize in order to function optimally in a realistic environment.

Keywords (10 maximum):

Blockchain	Smart contract	NFT	Platform
Ticket	Web application	Polygon	DApp

APORTACIÓN INDIVIDUAL AL GRUPO

Albert Granados Romero y Josep Marches Parra, han sido los desarrolladores, diseñadores e ideólogos de Tickbit, el prototipo de plataforma de compra y venta de entradas para eventos integrada en la tecnología blockchain.

Durante el proyecto hemos repartido las tareas con equidad y sin área de especialización específica, haciéndonos partícipes a ambos de todo el desarrollo del producto. Trabajar en conjunto durante el desarrollo, sin abordar individualmente aspectos específicos del proyecto, ha sido la metodología de trabajo que mejor resultado nos ha dado tanto en efectividad como en productividad y por lo tanto es la que decidimos emplear.

El redactado de la memoria en su mayoría también ha sido conjunto, aunque Josep se ha focalizado más en explicar la parte teórica y Albert en explicar la parte de proyecto. Hemos trabajado en conjunto para las partes más claves de la memoria, como la conclusión y la introducción.

De igual forma, a la hora de testear el producto, ambos nos reunimos para ejecutar las pruebas a la vez y comprobar que el producto funcionara correctamente.

SUMARIO

SUMARIO DE FIGURAS

GLOSARIO DE SIGNOS, SÍMBOLOS, ABREVIATURAS, ACRÓNIMOS Y TÉRMINOS

Dapp: siglas que corresponden a Aplicación Descentralizada, es una aplicación que funciona sobre una red peer-to-peer.

API: siglas que corresponden a Interfaz de Programación de Aplicaciones, permite la comunicación entre dos aplicaciones a través de un conjunto de reglas.

Hash: algoritmo matemático que transforma un bloque de datos en una serie de caracteres con una longitud fija.

Sidechain: cadena de bloques alternativa usada para mejorar las prestaciones de una cadena de bloques existente.

Web3: nuevo tipo de servicio construido sobre una cadena de bloques descentralizada.

Billetera: monedero virtual con el que se puede gestionar activos criptográficos.

View: diferentes vistas que corresponden a páginas de una web.

Tab: pestaña que permite alternar entre varios paneles de información dentro de una ventana.

Mintear: creación de un objeto digital único dentro de una blockchain.

Hosting: servicio de alojamiento online que permite publicar una página web en Internet.

Dominio: nombre que identifica una página web.

Servicio web cloud: servicio que permite el almacenamiento de archivos en la nube a través de Internet.

JSON-RPC: siglas que corresponden a JavaScript Object Notation Remote Procedure Call, es un protocolo que permite enviar notificaciones y llamadas a un servidor y se puedan responder de forma asincrónica.

Framework: marco de trabajo utilizado como punto de partida para elaborar un proyecto.

Front-end: parte de un sitio web que interactúa con los usuarios.

Backoffice: engloba todas las actividades relacionadas con la gestión interna de una página web. Popularmente conocido como “trabajo de oficina”.

NFT: (Token No Fungible) propiedad digital que está identificada y registrada en la blockchain.

1. INTRODUCCIÓN

1.1. LA PROBLEMÁTICA CON LA VENTA DE TICKETS

El sector de las plataformas de ventas de tickets para eventos cómo conciertos, festivales, teatro o deportes, tienen algunos desafíos a nivel de seguridad y fiabilidad con los que a día de hoy siguen teniendo que paliar.

El fraude que a menudo se produce en el mercado secundario de tickets, ha tenido un efecto negativo en el sector, provocando desconfianza tanto a clientes como organizadores de eventos.

Los principales problemas que encontramos son:

Reventa de entradas: Los eventos mediáticos y con alta demanda son con frecuencia el objetivo de los especuladores. A menudo, hay personas que se dedican a comprar una gran cantidad de estos tickets para bajar la oferta, ya que conocen su alta demanda y pretenden sacar partido de ello vendiéndolos en el mercado secundario a un precio mucho más elevado cuando se haya agotado en las plataformas oficiales de distribución.

Tickets falsos o duplicados: También es frecuente el uso de software informático de emisión de tickets falsos y/o duplicados que provocan el caos en este tipo de eventos cuando el cliente se da cuenta de que el ticket que ha comprado es falso o pertenece a varias personas, es decir, que es un duplicado.

Falta de protocolo de rastreo: Es imposible rastrear al dueño del ticket más allá de la venta de un proveedor. Cuando una entrada se vende de nuevo en el mercado secundario, ya no representa ninguna propiedad del comprador original cuyos datos tiene el organizador del evento. Por lo tanto, los organizadores de eventos no saben quién está en su evento.

Falta de un sistema de reventa de tickets regulado: Las entradas de los eventos se emiten de una manera que no permiten cambios en el mercado secundario, no existe la posibilidad de cambiar el nombre del titular del ticket ni controlar cuántas veces éste cambiará de manos. Una regulación que facilite la reventa del ticket a un precio justo, favorece a un mejor funcionamiento del sistema.

Baja confianza de los clientes: Hay falta de confianza por parte de los clientes que no saben si lo que han pagado es una entrada real, falsa o que ha sido duplicada por un tercero, ya que hay muchos portales en internet que se dedican a vender entradas de segunda mano haciéndose pasar por agentes de ventas de entradas autorizados.

Descontento de los artistas y organizadores de eventos: Los artistas y los promotores u organizadores de eventos están descontentos viendo que se produce una reventa de sus tickets que no pueden controlar, que perjudica a sus clientes y seguidores y de la que no sacan ningún tipo de beneficio.

1.2. LA BLOCKCHAIN COMO SOLUCIÓN

Popularmente, cuando se piensa en la blockchain, se piensa en un sistema de pago en criptodivisas y una forma de inversión y de especulación. Sin embargo, la blockchain es una tecnología de seguridad y almacenamiento de datos que va mucho más allá de únicamente este fin.

La tecnología blockchain tiene otras utilidades tan diversas como: garantizar el cumplimiento de la cadena de frío de unas gambas, registrar el número de accidentes de un coche o el número de reparaciones, crear contratos hipotecarios, crear patentes o incluso tener un registro público y seguro de todos los movimientos de fondos públicos de un estado.

La tecnología blockchain es una solución para muchos problemas de fiabilidad del presente, ya que es un sistema que garantiza una gran seguridad de los datos almacenados, así como neutralidad y transparencia.

Sin embargo, está tecnología tiene aún grandes retos que abordar como su impacto medioambiental y su estandarización para hacerlo accesible a usuarios de perfil no-técnico.

1.3. MOTIVACIONES Y SOLUCIONES DEL PROYECTO

Lo que la blockchain nos aporta a nivel de seguridad y fiabilidad, es el motivo por el cual pensamos que es una tecnología perfecta para llevar a cabo un nuevo proyecto, con una parte fuerte en investigación y con soluciones novedosas todavía no explotadas en el mercado.

Creemos que el interés por las nuevas tecnologías, la curiosidad y las ganas de aprender y adaptarse son las que nos convertirán en mejores ingenieros el día de mañana.

Siempre tuvimos claro desde el principio que el trabajo de final de grado era el momento perfecto para encontrar la motivación y la excusa para superarnos y aprender algo nuevo que nos aportará nuevos conocimientos y también un nuevo posible mercado laboral.

La idea de aplicar la blockchain como solución al sistema de ticketing, viene (como muchas de las ideas que surgen en la vida) de la experiencia propia: A una persona de nuestro entorno, le estafaron con una entrada del festival de Eurovisión y se dió cuenta en la puerta del evento, ya en Portugal.

De aquí surgió la idea de mejorar el sistema y la decisión de hacer de ello el trabajo de final de grado. La idea nos motivó mucho a los dos desde el principio y eso ayudó al buen resultado final.

2. LA BLOCKCHAIN

2.1. INTRODUCCIÓN

Cómo hemos expuesto en la introducción, la plataforma que hemos desarrollado en el proyecto corre sobre la blockchain por los motivos indicados. Por lo tanto, vamos a exponer cómo funciona esta tecnología por dentro y los diferentes tipos de blockchain que existen.

2.1.1. ¿QUÉ ES Y CÓMO FUNCIONA?

La blockchain [1] o la cadena de bloques es una tecnología de seguridad y almacenamiento de datos en red que se compone por un conjunto de nodos interconectados entre sí. Este tipo de estructura lo conocemos como redes P2P (Peer to Peer), lo que significa que todos los nodos operan de forma igual e equivalentes entre sí.

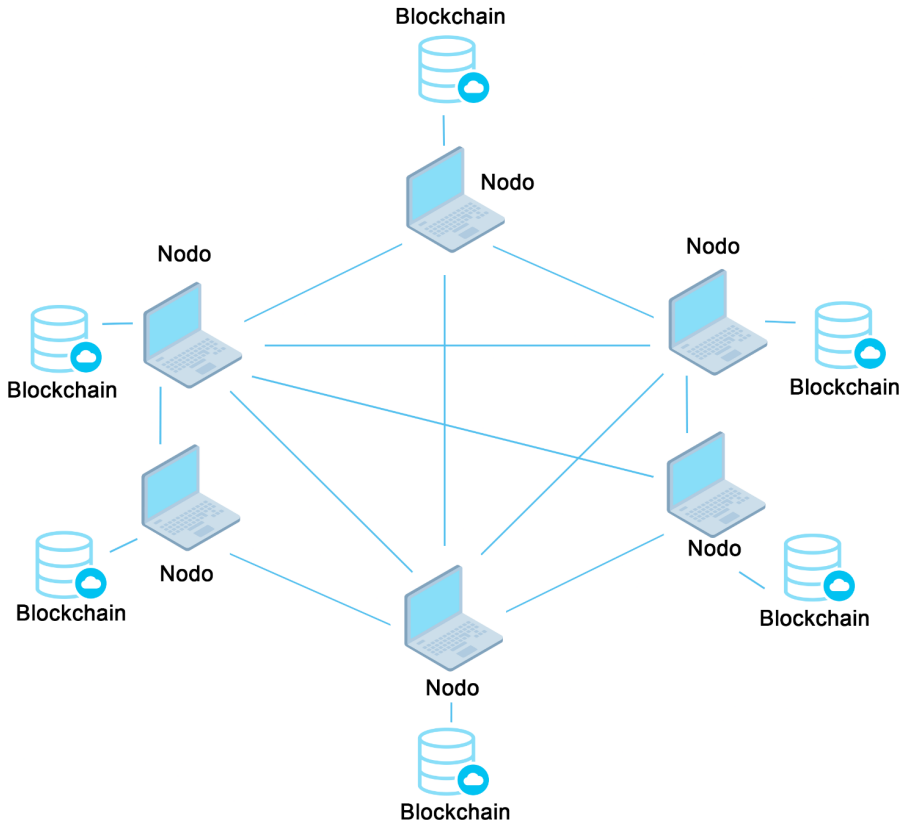


Figura 1: Ejemplo de red P2P con particularidad de la blockchain.
Fuente: Propia

Cada uno de estos nodos, son ordenadores que pueden crear, enviar y recibir información. Cada ordenador es aportado por un usuario que participa en el sistema y se convierte así en un nuevo nodo de la red.

El objetivo de esta red, será validar transacciones de información y guardar el historial de transacciones. Cada conjunto de transacciones que se realizan en la blockchain se representa como un bloque de información. Este bloque de información se difunde a todos los nodos de la red y estos se encargan de validar el bloque que contiene las transacciones.

Una vez que los nodos han validado el bloque, éste se añade a la cadena de bloques de forma permanente e inmutable, enlazandose con el bloque anterior y así progresivamente.

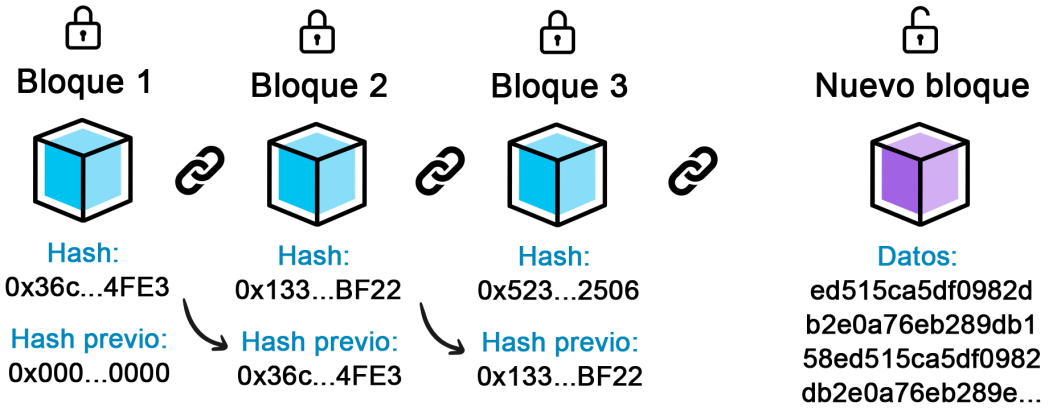


Figura 2: Diagrama del funcionamiento del sistema de cadena de bloques (blockchain)
Fuente: Propia

Todas las transacciones realizadas en la blockchain son públicas y han sido previamente validadas y añadidas a la cadena de bloques. Del mismo modo, no se pueden ni modificar ni eliminar transacciones previamente validadas.

Sintetizadamente, la blockchain es lo equivalente a un libro de registros digitales, dónde cada registro (transacción) solo puede ser validado por el consenso del sistema (nodos) y una vez grabado queda registrado públicamente para siempre.

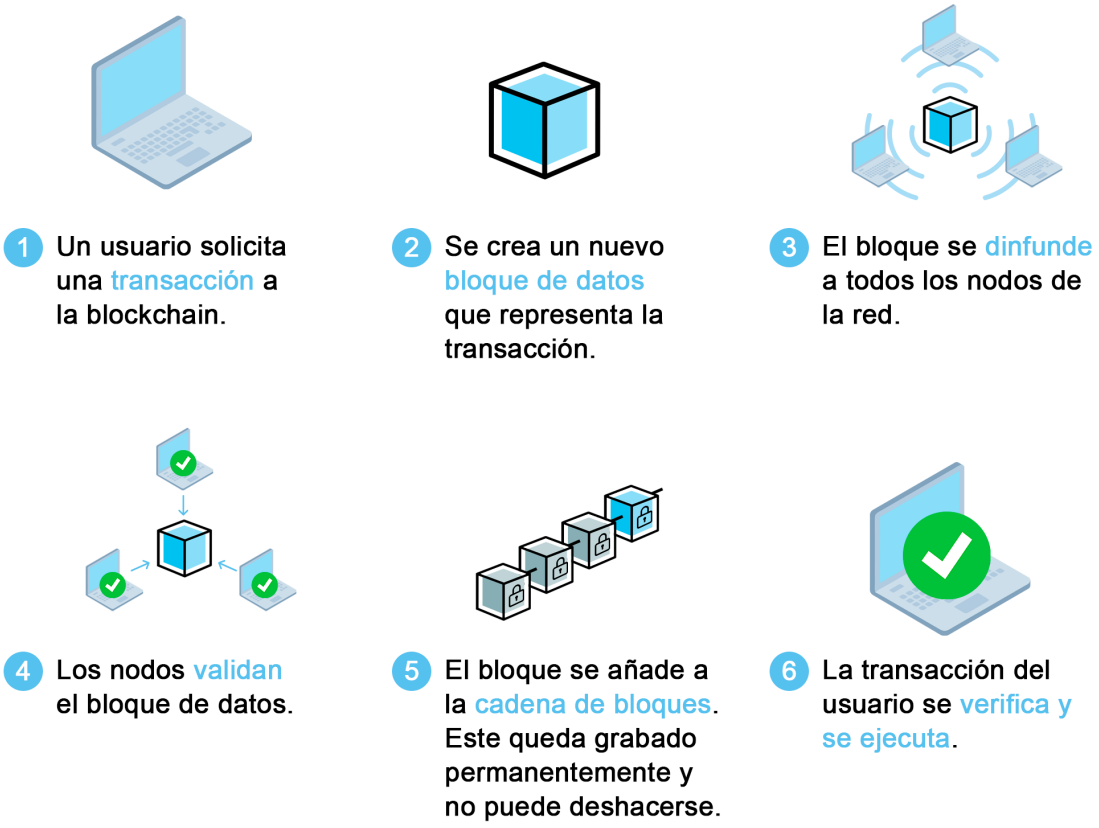


Figura 3: Diagrama de flujo de una transacción en la blockchain
Fuente: Propia

2.1.2. VENTAJAS RESPECTO A LOS SISTEMAS TRADICIONALES

- **Seguridad**: Su seguridad se basa en la dificultad de que los atacantes modifiquen o eliminen los datos contenidos en los bloques. Debido a que los datos se almacenan en múltiples lugares, es muy difícil para un atacante modificar o eliminar los datos de forma simultánea.

- **Transparencia**: Es una de sus principales ventajas debido a que todos los participantes pueden ver los datos contenidos en los bloques y todas las transacciones en el libro de registros público.

- **Eliminan intermediarios**: Se eliminan terceras partes a la hora de hacer transacciones en la blockchain, por ejemplo, al pagar con divisas digitales.

- **Descentralización**: Con respecto a muchos de los sistemas de almacenamiento de datos que se utilizan en la actualidad la blockchain no es propiedad íntegra de ninguna empresa u organización.

- **Anonimato**: En todo momento los usuarios que hacen uso de ella se identifican únicamente mediante una clave pública conformada por una larga cadena de caracteres aleatorios.

2.2. TIPOS DE BLOCKCHAIN

En la actualidad existen diferentes tipos de blockchain:

- **Privadas:** Es una red de bloques la cual se caracteriza porque una sola empresa o organización administra y limita la red y a su vez controla el acceso de las personas que participan, además también se encargan del mantenimiento. Un ejemplo de red privada en la actualidad es Hyperledger Fabric [2] de Linux Foundation. Se puede dar el caso de que una blockchain privada sea controlada por más de una empresa o usuario, estas tienen el nombre de blockchain de consorcio.

- **Pública:** Són las más habituales, se caracterizan porque cualquier persona en el mundo puede unirse y participar en ella sin tener ningún tipo de permiso. Todos los participantes de la red se tratan por igual y son anónimos. Además tienen asociada una criptomoneda con la cual se pagan las comisiones por hacer uso de ellas.

  La desventaja de estas es que a mayor número de usuarios se necesita un mayor nivel de seguridad, que se consigue gracias a los protocolos de consenso. Un ejemplo de red pública en la actualidad son Bitcoin [3] y Ethereum [4]

  Este tipo de blockchain han ido evolucionando con el paso de los años, actualmente tenemos tres tipos de generaciones las cuales cada una de ellas introdujeron nuevos cambios

  - **Primera generación**: Nació con un sistema de Pago P2P descentralizado llamado Bitcoin en el año 2009. Esta primera generación se basa en un protocolo de consenso llamado Proof-of-Work (PoW). Un protocolo de consenso es un mecanismo que permite regular la manera en la que los nodos que participan en la red validan los bloques que contienen transacciones de forma simultánea para que pase a formar parte de la cadena de bloques. En este protocolo (PoW) todos los nodos de la red compiten con el resto para validar un nuevo bloque, para validar el bloque se necesita resolver una operación matemática compleja cuya solución es un número de hash con un número determinado de ceros. Por lo tanto el nodo que tenga mayor capacidad computacional será el que consiga validar el bloque antes. Una vez validado se le otorga una recompensa mediante la criptomoneda asociada a la blockchain por validar el bloque. La desventaja de este tipo de protocolo de consenso es el alto consumo energético que conlleva tener una cantidad computacional en funcionamiento constante. A pesar de ello es un protocolo muy seguro, sencillo, fácil de implementar y altamente adaptable a las necesidades del hardware.

- **Segunda generación**: En esta nueva generación se introdujeron nuevas funcionalidades en la cadena de bloques una de ellas fueron los Smart Contracts. Los Smart Contracts son un código escrito en un lenguaje de programación que ejecuta funciones de manera autónoma y automática. Es decir, estas cadenas de bloques pueden implementar en los contratos estructuras de datos tradicionales y permite crear plataformas descentralizadas que funcionan automáticamente. Todo esto es gracias a que esta generación es Turing-completo, lo que significa que puede realizar cualquier tipo de cálculo ya que tiene un poder computacional equivalente a lo que se denomina la Máquina de Turing Universal. La primera en implementar estas nuevas actualizaciones fue la blockchain de Ethereum fundada en el 2015.

- **Tercera generación:** Uno de los principales problemas que están teniendo las primeras blockchain en la actualidad como Bitcoin o Ethereum es la escalabilidad. Cada vez hay más usuarios que utilizan la tecnología blockchain lo que produce que el número de transacciones concurrentes vaya aumentando significativamente con el paso del tiempo. Esto es un problema ya que se produce el denominado cuello de botella al haber más transacciones de las que la blockchain puede soportar, por eso en esta generación lo que se ha priorizado sobre todo es en mejorar la escalabilidad.

  Una de las principales novedades que se han introducido es cambiar el protocolo Proof-of-Work (PoW) por el Proof-of-Stake (PoS). Este nuevo protocolo cambia la forma de validar los bloques de la cadena, para que los nodos puedan validar un bloque ya no hace falta grandes capacidades computacionales como en el PoW, si no que el nodo validador necesita poseer parte de las criptomonedas nativas de la red en la que están tratando de validar los bloques. En este caso se elegirá de forma totalmente aleatoria el nodo que se hará cargo de validar el bloque siempre y cuando posea un número determinado de criptomonedas nativas de la red, dando mayor probabilidad al nodo que posea más. Con este protocolo se consigue que se consuma una menor cantidad de energía, que no haga falta una inversión muy grande en hardware o espacio para almacenarlo y por último mejora la rapidez de las transacciones.

  Otra de las mejoras que se introdujeron es el Sharding o en castellano Fragmentación. Sharding es el proceso de dividir el estado de toda la cadena de bloques en pequeños conjuntos denominados como “shards”, una vez dividida la red procesa todos estos “shards” en paralelo, lo que permite que se produzca un procesamiento secuencial de múltiples transacciones consiguiendo un mejor manejo y haciendo que sean más fáciles de validar.

  Actualmente se está empezando a utilizar lo que conocemos como soluciones de segunda capa, lo que se hace en estas es cambiar la carga de transacciones de una blockchain a una arquitectura off-chain, es decir, se

derivan las transacciones a otra blockchain paralela, esta procesa la información y notifica los resultados finales de la transacción transferida otra vez de vuelta. Con esto conseguimos que se reduzca notablemente la congestión de la red.

Estas soluciones principalmente se aplican en la blockchain de Ethereum que es la que más problemas de escalabilidad está teniendo a día de hoy ya que es de las más utilizadas. Las solución de segunda capa más conocida a día de hoy es Polygon [5]

- **Híbridas**: Intentan utilizar las mejores soluciones de las blockchain privadas y públicas. En este caso solo pueden participar personas que hayan sido previamente autorizadas pero la información contenida en la blockchain es pública entre las dos partes que hacen la transacción. Esto implica que este tipo de redes sean más transparentes y seguras. Un ejemplo de este tipo de cadena de bloques es Corda [6].

2.4. ETHEREUM

Ethereum es una cadena de bloques que permite desarrollar DApps mediante Smart Contracts. Desde que se creó Ethereum en 2015 ha utilizado el protocolo de consenso Proof-of-Work hasta que el 15 de septiembre de 2022 se sacó la versión de ETH 2.0 con el cual se pasaba a utilizar Proof-of-Stake. Se caracteriza por ser una de las blockchain más descentralizadas y con más seguridad, a ello se debe que sea la red más usada por los desarrolladores.

2.4.1. CUENTAS

Las cuentas [7] en Ethereum sirven principalmente para recibir, almacenar y enviar Ethereum y otros tokens de la red e interactuar con los contratos inteligentes implementados. Principalmente constan de cuatro campos:

- **nonce**: contador que indica el número de transacciones enviadas desde la cuenta y asegura que estas transacciones solo se procesan una única vez. En una cuenta de contrato indica el número de contratos creados por esa cuenta.

- **saldo**: número de wei que pertenecen a la dirección de la cuenta. Wei es una denominación de la moneda ETH, cada ETH equivale a 1e+18 wei.

- **codeHash**: hace referencia al código de una cuenta en la máquina virtual de Ethereum (EVM). Este código se ejecuta si la cuenta recibe una llamada de mensaje, es inmutable y por lo tanto a diferencia de todos los demás campos, no se

puede modificar. Está contenido en la base de datos de estado con sus correspondientes hashes para su recuperación.

- **storageRoot**: se le conoce también como hash de almacenamiento, es un hash de 256 bits que codifica el contenido de almacenamiento de la cuenta.

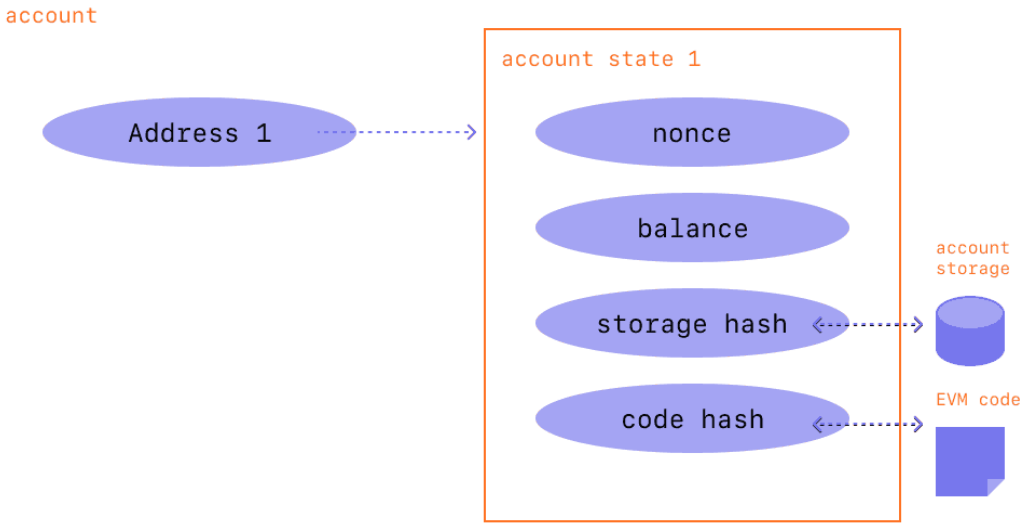


Figura 4: Diagrama de los componentes de una cuenta en la blockchain de Ethereum. Fuente: ethereum.org

En la blockchain de Ethereum existen dos tipos de cuentas en esta blockchain que son:

- **De propiedad externa**: las puede tener cualquier persona que quiera hacer uso de la red. Está formada por dos claves criptográficas, una pública y otra privada, estas claves ayudan a probar que una transacción fue realmente firmada por el remitente y evita posibles falsificaciones.

  La clave privada cuenta con 64 carácteres hexadecimales y sirve para firmar las transacciones que realizas, garantizando la custodia de los fondos de la cuenta. Este tipo de clave se puede llegar a cifrar con una contraseña y no se debe compartir con nadie. Un ejemplo de clave privada sería el siguiente:

  ```
  ffffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd036415f
  ```

  La clave pública es una dirección que se genera a partir de la clave privada mediante el algoritmo de forma digital de curva elíptica. Se toman los últimos 20 bytes del hash Keccak-256 de la clave pública, añadiendo 0x al principio. Esta clave

como su propio nombre indica es pública y se puede compartir con otras personas para la realización de transacciones. Un ejemplo de clave pública sería el siguiente:

```
0x5e97870f263700f46aa00d967821199b9bc5a120
```

- **De contrato**: se asignan automáticamente cuando se implementa un contrato inteligente en la cadena de bloques, en este caso la de Ethereum. Cuentan con una dirección similar a las claves públicas de las cuentas de propiedad externa y se genera mediante la dirección del creador del contrato y el número de transacciones enviadas desde esa dirección. Ejemplo:

```
0x06012c8cf97bead5deae237070f9587f8e7a266d
```

2.4.1. TRANSACCIONES

Una transacción [8] se puede definir como una acción que está firmada criptográficamente iniciada por una cuenta del tipo propiedad externa. Para realizar una transacción es necesario pagar una pequeña comisión en ETH la moneda nativa de la red a la que llamamos gas con tal de que los nodos a los que la transmitimos la validen. Para facilitar las comisiones mediante el uso de ETH, este se divide en unidades más pequeñas llamadas Wei [9], un Wei equivale a un ETH dividido por 10^{18}.

Una transacción que ha sido realizada por una cuenta contiene diferentes campos, los cuales són:

- **destinatario**: dirección del destinatario que puede ser otra cuenta del tipo propiedad externa en la cual se transfiere un valor o del tipo contrato en la cual se ejecutará código del contrato.

- **firma**: identificador del que realiza la transacción, esta es generada mediante la clave privada y confirma que el remitente ha autorizado la transacción.

- **valor**: cantidad de ETH que se transfiere a la cuenta del destinatario.

- **datos**: campo opcional en el cual se pueden incluir otros tipos de datos.

- **límite de gas**: cantidad máxima de gas que puede consumir al realizarse la transacción.

- **prioridad máxima de comisión por gas**: cantidad máxima de gas que se incluirá como recompensa al nodo que valide antes la transacción.

- **máxima comisión por gas**: cantidad mínima de gas que se quiere pagar para realizar la transacción.

Ya sabiendo los campos que incluyen todas las transacciones que se realizan sobre la blockchain de Ethereum, podemos diferenciar tres tipos diferentes de transacciones. Primero tenemos las transacciones regulares que són las que se realizan desde una cuenta a otra, por otro lado tenemos las de despliegue de contrato que són las que se realizan para poder publicar un contrato en la blockchain de Ethereum y que este pase a ser funcional. Por último tenemos las de ejecución de contrato que són las que interactúan con un contrato inteligente ya desplegado en la blockchain.

2.4.2. BLOQUES

Como hemos explicado anteriormente los bloques [10] son conjuntos de transacciones que contienen un hash del bloque anterior de la cadena para estar interconectados entre sí y prevenir fraudulencias, ya que cualquier cambio de algún bloque invalidará todos los siguientes bloques. Una vez que se validan todas las transacciones del bloque y por lo tanto el bloque, se propaga al resto de la red. Los bloques aseguran que todos los participantes de la blockchain de Ethereum mantengan el mismo estado sincronizado en todo momento.

Los bloques están formados por diferentes campos:

- **timestamp**: hora en la que el bloque ha sido validado.
- **número de bloque**: longitud de bloques de la blockchain.
- **comisión base por gas**: comisión mínima por gas necesaria para que una transacción sea incluida.
- **dificultad**: define la complejidad para poder validar el bloque.
- **hash**: identificador único del bloque.
- **parent hash**: identificador único del bloque anterior, permite que los bloques estén enlazados.
- **lista de transacciones**: número de transacciones que han sido incluidas.
- **state root**: estado completo del sistema, incluyendo entre ellos el saldo de todas las cuentas, el almacenamiento de los contratos y los nonces de las cuentas.
- **nonce**: número entero que deben encontrar los validadores y que combinado con los demás datos que incluye el bloque permiten encontrar un hash válido.

2.4.3. TOKENS Y TIPOS DE ESTÁNDARES

Los tokens [11] son activos digitales dentro de una blockchain que representan valores, coleccionables virtuales o activos físicos del mundo real y se pueden diferenciar en dos grandes tipos de tokens los fungibles y los no fungibles.

Los tokens fungibles son aquellos no pueden distinguirse uno de otro y que tienen el mismo valor, es decir que són iguales. Por otro lado, los tokens no fungibles presentan características únicas, se pueden diferenciar unos de otros y tienen una trazabilidad personalizada, estos tokens normalmente representan una propiedad de un activo digital o un activo del mundo real, como por ejemplo una entrada para un evento.

La blockchain de Ethereum tiene diferentes tipos de estándares ERC (Ethereum Request for Comments), ERC es un mecanismo dentro de la blockchain de Ethereum para definir estos estándares de forma que los tokens tengan propiedades en común y les permitan ser interoperables entre ellos.

Los dos estándares más conocidos y que vamos a estar utilizando en el proyecto son:

- **ERC-20**: estándar para tokens fungibles, permite que los tokens creados en este estándar y que pertenezcan al mismo proyecto tengan todos el mismo valor y sean iguales, un ejemplo de este estándar es la moneda nativa ETH de la blockchain de Ethereum.

- **ERC-721**: estándar para tokens no fungibles o más conocidos como (Non-Fungible-Token o NFT), permite que los tokens presenten características únicas y hacen posible su distinción y trazabilidad dentro del mismo proyecto, además el valor puede variar uno respecto al otro.

2.4.4. MÁQUINA VIRTUAL DE ETHEREUM (EVM)

La blockchain de Ethereum al ser una blockchain de segunda generación permite la ejecución de Smart Contracts dentro de su red y por lo tanto necesita algún mecanismo para hacer que funcionen. Para esto existe la Ethereum Virtual Machine [12] (EVM), como dice su propio nombre es una máquina virtual de Turing que permite la ejecución de código dentro de la blockchain a cualquier usuario garantizando un alto nivel en cuanto a seguridad se refiere, ya que al tratarse de una máquina virtual tiene ciertas limitaciones y le permite poder ejecutar códigos sin que pueda comprometer al sistema. Además es descentralizada ya que existen muchos nodos que ejecutan una copia exacta de la misma máquina y actúan de forma simultánea.

La forma en la que se ejecuta la EVM hace que se asemeje a una máquina de pila, una máquina de pila es un modelo computacional en el cual su memoria está formada por estructuras de datos de pila o en inglés stack. Estos stacks contienen las cuentas y saldos

conjuntamente con el estado global de la máquina y tienen una profundidad de 1024 ítems, donde cada ítem es una palabra de 256 bits.

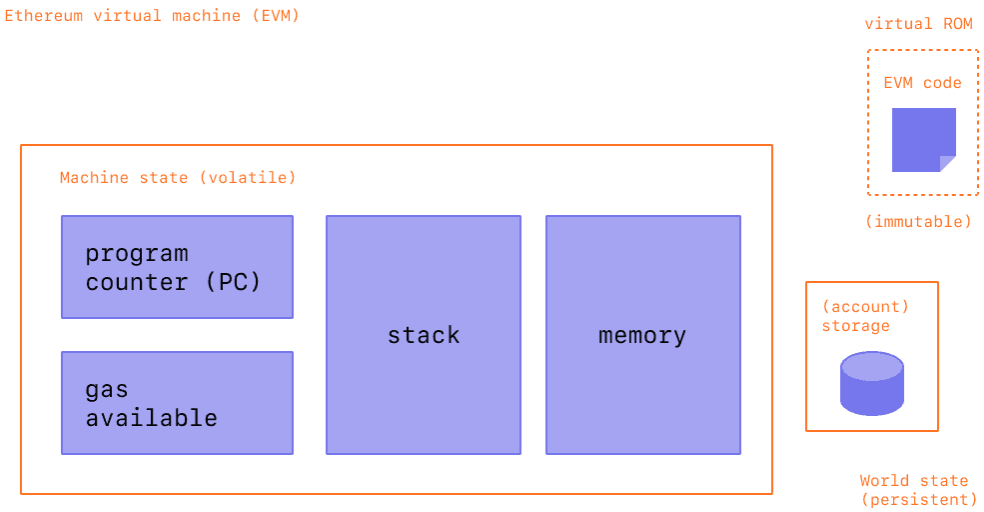


Figura 5: Diagrama de la arquitectura de la EVM.
Fuente: ethereum.org

Se creó un nuevo lenguaje de programación de alto nivel para el desarrollo de la EVM y los Smarts Contracts el cual se llama Solidity [13]. Cuando un usuario hace una transacción con un Smart Contract, el código de Solidity se transforma en códigos de operación (OPCODES) como por ejemplo XOR, AND, ADD, SUB, PUSH, POP entre otros y una vez transformado se ejecuta un ByteCode que realizará las operaciones en el Smart Contract.

```
PUSH1 0x60 PUSH1 0x40 MSTORE PUSH1 0x18 PUSH1 0x0 SSTORE CALLVALUE
ISZERO PUSH1 0x13 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST JUMPDEST PUSH1
0x36 DUP1 PUSH1 0x21 PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN STOP PUSH1
0x60 PUSH1 0x40 MSTORE JUMPDEST PUSH1 0x0 DUP1 REVERT STOP LOG1 PUSH6
0x627A7A723058 KECCAK256 SLT 0xc9 0xbd STOP ISZERO 0x2f LOG1 0xc4 DUP1
0xf6 DUP3 PUSH32
0x81515BB19C3E63BF7ED9FFBB5FDA0265983AC79800290000000000000000000
```

Figura 6: Ejemplo de OPCODES.
Fuente: medium.com

```
60606040525b600080fd00a165627a7a7230582012c9bd00152fa1c480f6827f81515b
b19c3e63bf7ed9ffbb5fda0265983ac7980029
```

Figura 7: Ejemplo de ByteCode.
Fuente: medium.com

2.5. POLYGON

A pesar de todas las ventajas que nos ofrece Ethereum a día de hoy, su Blockchain presenta varios problemas de escalabilidad y eso produce efectos negativos como la lentitud a la hora de hacer transacciones o las grandes comisiones que se pagan ya que la red normalmente está siempre saturada y no puede procesar tanta carga. Por eso decidimos no implementar nuestra plataforma de compra venta en esta red y decidimos optar por una opción que fuera más escalable, rápida y cuyas comisiones fueran muy bajas para que a la hora de comprar tickets o publicar eventos en la plataforma no tardara tanto en hacerlo y que no tuviera un coste muy elevado.

Por lo tanto decidimos usar Polygon, una solución de segunda capa de la Blockchain de Ethereum la cual nos ofrece unas transacciones rápidas, muy bajas comisiones y podemos seguir trabajando sobre su red ya que están interconectadas.

Polygon es una sidechain, es decir, una red de nodos descentralizados conectada a la red de Ethereum que permite que se puedan comunicar entre ellas. Su protocolo de consenso para validar transacciones es Proof-of-Stake (PoS) y las comisiones por las transacciones realizadas son pagadas con su token nativo MATIC.

Cada bloque que se genera en Polygon genera un árbol hash de Merkle [14], este tipo de estructura de datos es un árbol en el cual cada nodo que no es una hoja, es decir que tiene nodos hijos, está etiquetado con el hash de la concatenación de los valores de sus hijos. Una vez el árbol está creado, asocia todas las transacciones dentro de ese bloque de la red de Polygon y se envía a la blockchain de Ethereum como información usando la tecnología Plasma que permite la transferencia de información entre las dos blockchain.

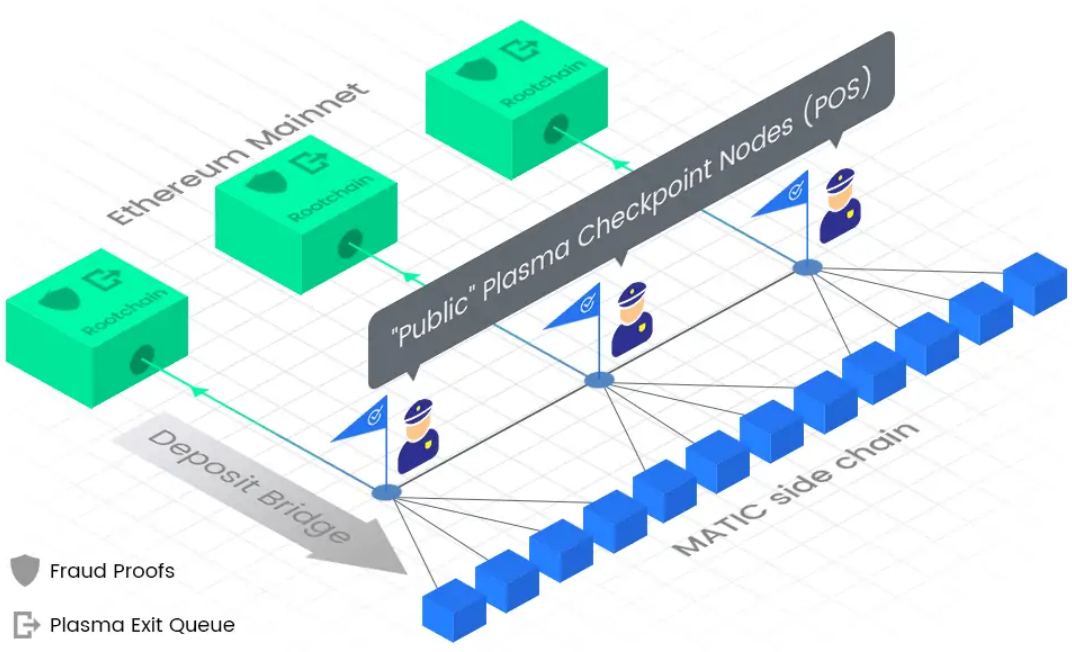


Figura 8: Funcionamiento de la sidechain Polygon
Fuente: academy.bit2me.com

2.5.1. MUMBAI POLYGON TESTNET

A la hora de desarrollar proyectos se necesitan hacer pruebas para ver que todo funciona correctamente en el Smart Contract y que no haya errores o fallos de seguridad que puedan comprometer a la empresa o el dinero de la gente. Para esto existen las llamadas redes de pruebas o Testnet, que són copias idénticas de la blockchain original en la cual se pueden hacer pruebas sin que afecte a la blockchain principal y donde las comisiones por transacciones o adquisición de activos se realizan con dinero ficticio
.
En nuestro caso cómo hemos usado la blockchain Polygon para desarrollar la plataforma de compra venta, hemos tenido que utilizar una Testnet, en este caso la red de pruebas oficial de Polygon que es la Mumbai Polygon Testnet donde hemos podido hacer pruebas para comprobar que el Smart Contract funcionaba de una forma correcta y desplegado el contrato para que esté de forma pública sin que tenga un coste de dinero real para nosotros.

3. PLANIFICACIÓN DEL PROTOTIPO

La planificación de cómo iba a ser el prototipo, y cómo sería la forma de proceder con el desarrollo a partir de la idea principal, se divide en 3 partes: la especificación del proyecto y sus objetivos, la organización de los plazos y el equipo, y por último, la metodología de trabajo.

Pero lo primero, obviamente, es definir un nombre para el proyecto: Tickbit.

Este nombre es una combinación de las palabras “ticket” y “bit”, aludiendo al producto físico que simboliza el ticket (o entrada), y a la digitalización con el término “bit”. Hay que comprobar también que este nombre tenga algún dominio disponible y nos decidimos por comprar el dominio https://tickb.it.

3.1. ESPECIFICACIÓN Y OBJETIVOS

Una vez adquirida una base de conocimiento sobre el desarrollo en la blockchain y con el nombre decidido, pasamos a decidir de qué partes va a constar el desarrollo.

Estas partes a especificar serían los objetivos que queríamos alcanzar, por lo que posteriormente el desarrollo y los plazos tenían que girar en torno a cumplir estos objetivos.

Las partes y los objetivos en los que constaría el desarrollo del proyecto son:

- **Contratos en la blockchain** que van a definir las normas que rigen nuestro proyecto.

- **Una web para la gestión** dónde una empresa promotora de eventos o artistas en particular puedan registrar sus propios eventos para ser listados en la web principal y empezar a vender tickets.

- **Una web principal de venta de tickets** dónde cualquier persona puede comprar tickets para eventos. La compra se haría con divisas digitales y se controlaría el número de entradas por compra máximo que se pueden adquirir.

- **Un sistema de validación de entradas** que pueda validar la propiedad de un ticket, validando tanto la autenticidad como la propiedad de la persona que lo valida.

- **Un sistema de reventa de entradas regulado** dónde alguien que quiera vender su entrada pueda hacerlo, únicamente al precio oficial y aceptando una comisión de reventa.

- **Un sistema de promoción de eventos** mediante campañas de publicidad para la web principal con el que los promotores o artistas puedan pagar para dar más

relevancia a sus eventos.

- **Un sistema de control de ventas e ingresos** por parte de los promotores de eventos o artistas.

Y marcamos algunas normas y características que tenía que cumplir el proyecto:

- **Estamos diseñando un prototipo**, es decir, que no íbamos a intentar abarcar todas las posibilidades que puede tener una web de venta de entradas, ya que es un desarrollo no realista de realizar entre dos personas en el plazo marcado para un trabajo de final de grado. Fue importante definir límites y nuestro foco tenía que ser la implementación de la tecnología blockchain, y el desarrollo de la funcionalidad de venta-compra-validación de tickets de forma segura. Tener claro nuestro foco de importancia fué uno de los puntos claves en el posterior desarrollo.
  Esto también implica que el prototipo tiene que estar más orientado al testeo y la demostración que a la venta real en sí (funcionalidades abiertas para que todo el mundo pueda probarlas y precios de servicios asequibles para testear).

- **Creación de webs de desarrollo propio (sin plantillas)**, atractiva y sencilla.

- **Web online de libre acceso** la web tendría que funcionar online para que cualquiera pueda verla y probarla, funcionando en la red de pruebas de la blockchain.

- **Crear una imagen corporativa básica**: tener un nombre, un logo corporativo, y un dominio propio.

- **Las webs tienen que tener explicaciones** y ser relativamente sencillas de utilizar por alguien que no tenga conocimientos expresos sobre la blockchain.

- **Una fuente de ingresos** como funcionalidad del sistema, ya sea con comisiones y/o publicidad. Sin profundizar más allá en el análisis económico del modelo de negocio, pero mostrando algún posible método de ingresos que pueda sostener el proyecto.

3.2. ORGANIZACIÓN Y PLAZOS

Para encarar el proyecto del prototipo fue crucial una buena asignación de plazos límite de los objetivos, y una buena organización en equipo.

El proyecto se registró el 8 de febrero de 2022. Desde ese mismo día, después de especificar el proyecto y los objetivos, el siguiente paso inmediato fué definir dos planes de acción: el optimista y el realista. Cada uno iría según las dos opciones de fecha de entrega: en julio o en octubre de 2022.

En estos planes se apuntan las tareas a grandes rasgos que íbamos a tener que realizar teniendo en cuenta la especificación inicial.

Los planes tenían como finalidad marcar fechas límites a cada tarea, para no excedernos en el desarrollo de cada una de ellas. A cada tarea le asignamos un peso proporcional en función de la cantidad de tiempo de dedicación que sería necesario invertir en esta tarea. Este valor va representado por un porcentaje y el total de las tareas tiene que sumar el 100% del trabajo.

Finalmente según el peso, y las características de la tarea (si esa tarea es propensa a contratiempos o no) se asigna a cada tarea una fecha límite.

Plan optimista

El plan optimista tiene como objetivo hacer la entrega del TFE en la primera ronda de entrega, el día 5 de julio de 2022.

Esta era la definición del plan optimista:

Tarea	Peso	Fecha inicial	Fecha límite
Registro del TFE	0%	08/02/2022	08/02/2022
Especificación y objetivos del proyecto	1,5%	08/02/2022	10/02/2022
Definición, planificación y preparación de la metodología de trabajo en grupo	2%	10/02/2022	15/02/2022
Proceso de investigación y aprendizaje	10%	15/02/2022	15/03/2022
Desarrollo de los contratos	12,5%	15/03/2022	01/04/2022
Desarrollo del backoffice	30%	01/04/2022	01/05/2022
Desarrollo de la web principal	20%	01/05/2022	01/06/2022
Redacción de la memoria	24%	01/06/2022	05/07/2022
Entrega del TFE	100%	05/07/2022	

Figura 9: Tabla de planificación de tareas del plan optimista.
Fuente: propia

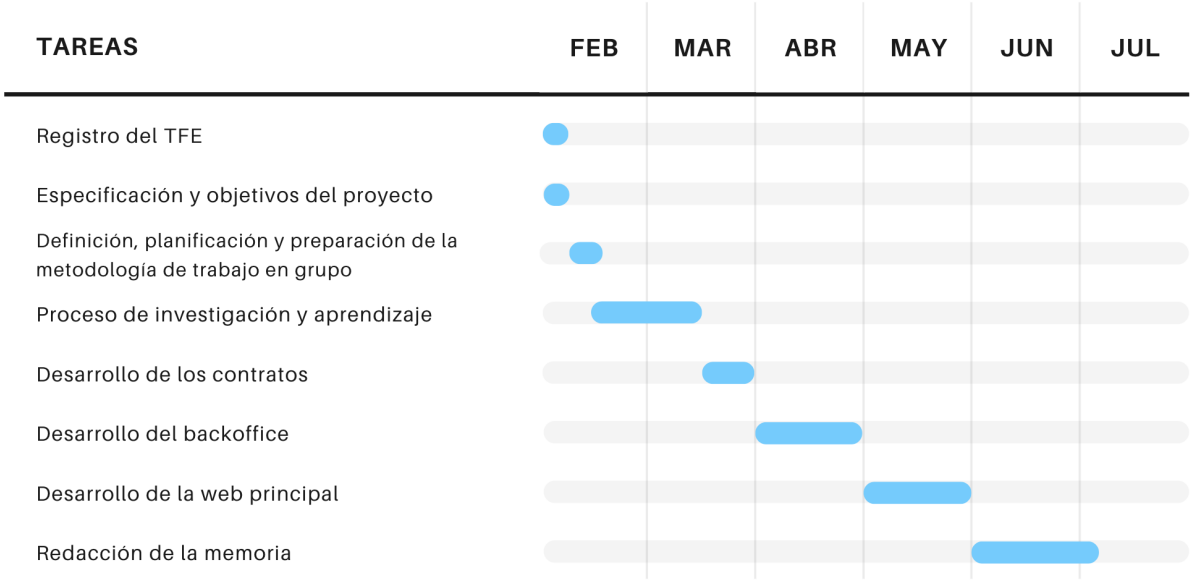


Figura 10: Diagrama de Gantt del plan optimista.
Fuente: propia

Desde el principio pensamos que está opción era complicada de asumir, pero era necesario ponerla sobre la mesa.

Plan realista

El plan realista tiene como objetivo hacer la entrega del TFE en la segunda ronda de entrega, el día 18 de octubre de 2022, es decir con un cuatrimestre extra para presentarlo. Por el volumen de trabajo, era el plan más probable.

Esta era la definición del plan realista:

Tarea	Peso	Fecha inicial	Fecha límite
Registro del TFE	0%	08/02/2022	08/02/2022
Especificación y objetivos del proyecto	1,5%	08/02/2022	20/02/2022
Definición, planificación y preparación de la metodología de trabajo en grupo	2%	20/02/2022	05/03/2022
Proceso de investigación y aprendizaje	10%	05/03/2022	05/04/2022
Desarrollo de los contratos	12,5%	05/04/2022	15/05/2022
Desarrollo del backoffice	30%	15/05/2022	15/07/2022
Desarrollo de la web principal	20%	15/07/2022	01/09/2022
Redacción de la memoria	24%	01/09/2022	18/10/2022
Entrega del TFE	100%	18/10/2022	

Figura 11: Tabla de planificación de tareas del plan realista.
Fuente: propia

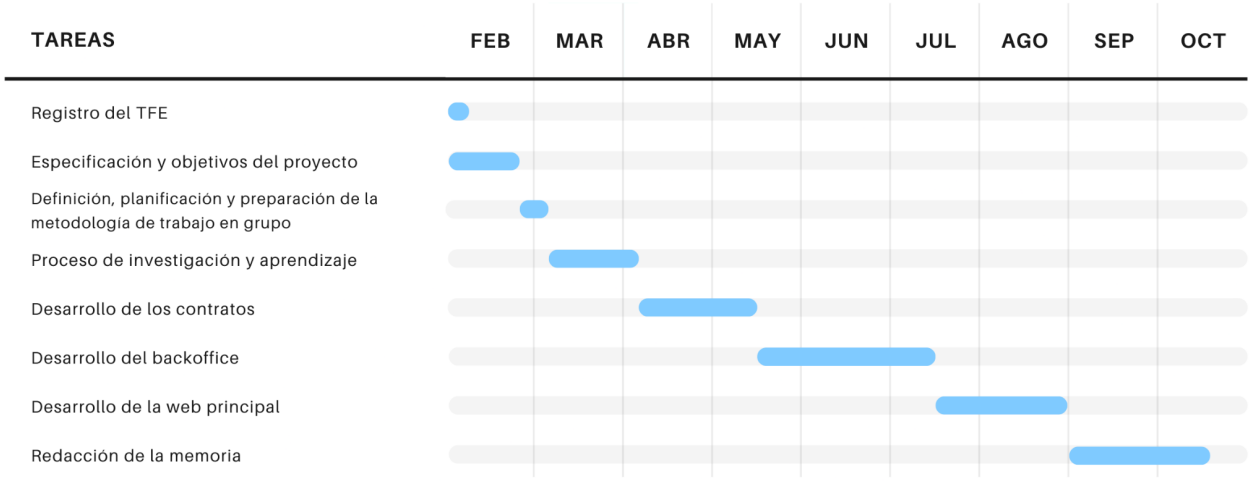


Figura 12: Diagrama de Gantt del plan realista.
Fuente: propia

El proyecto que teníamos pensado era extenso para dos personas, así que desde el principio teníamos la sospecha de que el plan realista iba a acabar siendo el elegido. Al poco tiempo de empezar con el proceso de investigación, confirmamos que el plan realista era la hoja de ruta correcta.

3.3. METODOLOGÍA DE TRABAJO EN GRUPO

Para organizar la metodología de trabajo en equipo, decidimos que lo mejor sería utilizar la metodología de desarrollo ágil SCRUM.

SCRUM es una metodología de buenas prácticas muy típica para proyectos en equipo (sobretodo en proyectos de software) que ya habíamos trabajado previamente en la carrera en la asignatura PTIN (Proyecto de Tecnologías de la Información) que basa su temario en la realización de un proyecto en equipo en estructura de empresa del sector tecnológico.

El objetivo del SCRUM es entregar un producto de alta calidad en un plazo de tiempo reducido. Se llevan a cabo lo que se llaman Sprints que son lapsos de un tiempo determinado donde se trabaja en unas tareas previamente definidas al inicio de cada sprint.

SCRUM se divide en tres fases:

- **Planificación**: en esta fase, el equipo se reúne para planificar el trabajo a realizar en el sprint. Se establecen los objetivos y se asignan tareas a cada miembro del equipo.

- **Ejecución**: en esta fase, el equipo trabaja en conjunto para completar las tareas asignadas.

- **Revisión**: en esta fase, el equipo se reúne para revisar el trabajo realizado y evaluar los resultados.

SCRUM es un proceso iterativo, lo que significa que cada fase se repite en cada sprint. El ciclo de vida del SCRUM se divide en sprints, que son períodos de tiempo (generalmente 2-4 semanas) en los que el equipo trabaja en un conjunto de tareas.

Al final de cada sprint, el equipo realiza una revisión del trabajo realizado y evalúa los resultados. A partir de esta revisión, el equipo planifica el trabajo para el siguiente sprint.

Para llevarlo a cabo la herramienta que utilizamos es Trello [15] que es un gestor de tareas en forma de tarjetas muy útil para esta metodología.

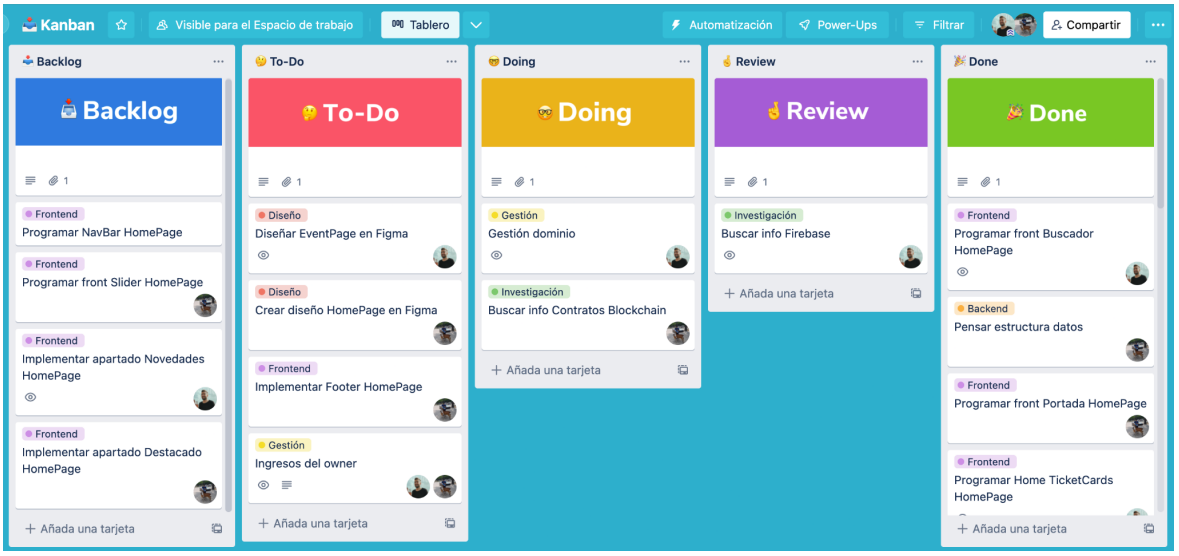


Figura 13: Panel de tareas de Trello
Fuente: https://trello.com

4. MODELO DE NEGOCIO

Para llevar a cabo el desarrollo, es imprescindible definir algunos aspectos básicos a cumplir del modelo de negocio en el que se basaría nuestro producto. Sin entrar a materia en profundidad, debemos definir y tener en cuenta que el modelo de negocio tiene que cumplir los requisitos y objetivos que marcamos al inicio del proyecto.

4.1. SERVICIOS QUE OFRECEMOS

4.1.1. SERVICIO DE COMPRA Y REVENTA DE TICKETS

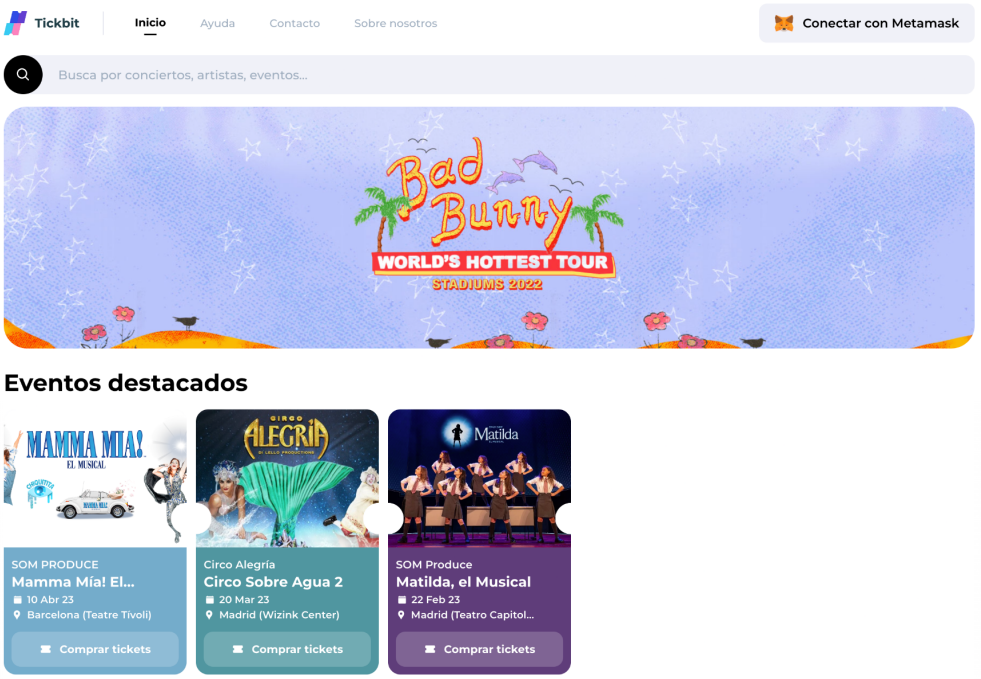


Figura 14: Página principal de la web clientes Tickbit
Fuente: tickb.it

La compra de tickets es el servicio que proporciona la web principal de nuestra plataforma Tickbit. Un usuario puede acceder a nuestra web y comprar entradas para diferentes tipos de eventos ya sean de música como conciertos y festivales, como de deportes o teatro entre otros.

Además si un usuario que ha adquirido un ticket en nuestra plataforma no puede asistir a un evento o ha cambiado de opinión y no quiere ir, tiene la posibilidad de, una vez acabada la disponibilidad de entradas del evento del cual la ha comprado, revender la entrada y recuperar casi íntegramente el importe pagado con el servicio de reventa dentro de la plataforma.

4.1.2. SERVICIO DE GESTIÓN DE EVENTOS Y PUBLICIDAD

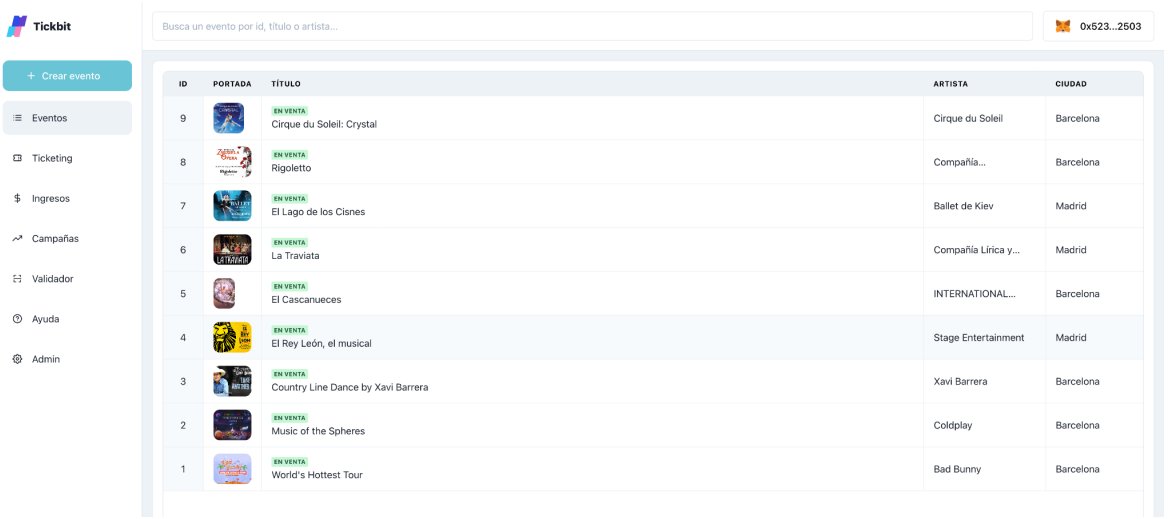


Figura 15: Página principal de la web backoffice Tickbit
Fuente: business.tickb.it

La gestión de eventos es el servicio principal que proporciona la web backoffice del proyecto. En esta web cualquier artista o organizador de eventos puede publicar un evento o hacer modificaciones sobre los eventos que ha publicado. En ella además se puede tener un control sobre de los ingresos en el cual se mostraran los tickets vendidos y el dinero recaudado por mes.

También está la posibilidad de la compra de publicidad para promocionar un evento que hayas publicado. La publicidad comprada en la plataforma tiene como duración una semana, es decir, podrás seleccionar un rango de días que va desde el lunes al domingo, una vez acabada esa semana se reiniciará la publicidad y se pasará a mostrar en la plataforma de clientes la de la semana siguiente.

Actualmente tenemos dos tipos de publicidad que son la sección de **portada** (solo una disponible por semana) en la cual el evento promocionado como portada se va a ver en una imagen grande nada más entrar a la web de clientes haciendo que cuando un cliente entre a la web lo primero que vea sea ese evento. Por otro lado tenemos la sección de **destacados** en la que cinco eventos por semana compartirán un espacio debajo de la portada y estarán diferenciados del resto de eventos de la web.

La sección de portada tendrá un coste más elevado que la sección de destacados ya que es una sección para un solo evento y su visibilidad es más grande que la de destacados. El precio de ambos, podría variar dependiendo de la demanda.

Figura 16: Sección de compra de campañas
Fuente: business.tickb.it

4.1.2. SERVICIO DE VALIDACIÓN DE TICKETS

Se proporciona una herramienta que está desarrollada en dos partes para la validación de los tickets en la asistencia de los eventos: Una es para los clientes en la cual desde su perfil en la web de clientes de Tickbit pueden acceder a los tickets disponibles que han comprado recientemente y en cada ticket hay un botón para validarlo con el escaneo de un código QR.

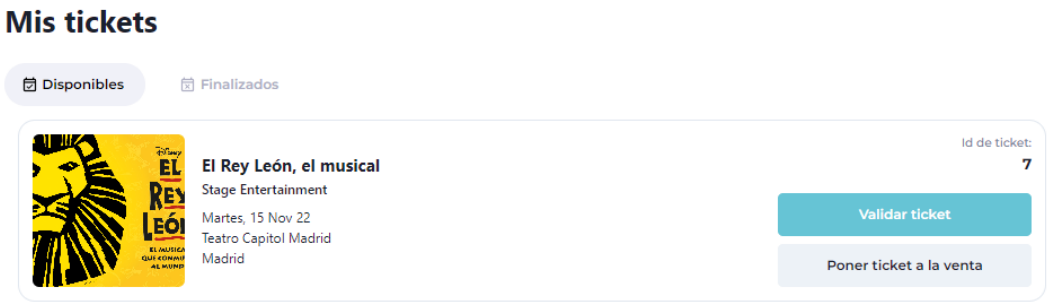


Figura 17: Sistema validación tickets cliente
Fuente: business.tickb.it

La otra es para los organizadores de eventos y artistas en la cual desde la web backoffice podrán seleccionar el evento a validar y se irá generando un QR automático que los clientes irán escaneando y se validan las entradas automáticamente.

Figura 18: Sistema validación de tickets artistas
Fuente: business.tickb.it

4.2. SOLUCIONES

Problemas en el mercado	Soluciones que aportamos
Reventa a precios excesivos	Sistema de reventa regulado con un precio máximo fijado
Falsificación de entradas	Entradas únicas mediante NFTs y con posibilidad de comprobar su veracidad
Descontento de los artistas por la reventa	Comisión para el artista por cada reventa producida.
Baja confianza de los clientes	Plataforma segura y confiable
Falta protocolo de rastreo de entradas	Entradas rastreables mediante la tecnología blockchain

Figura 19: Tabla con los problemas y soluciones del modelo de negocio
Fuente: Propia

4.3.TIPOS DE CLIENTES

Debemos distinguir los diferentes tipos de clientes que puede tener la plataforma Tickbit y serán explicados en los siguientes apartados.

4.3.1. CLIENTE ESTÁNDAR

El primer cliente es un usuario que quiere comprar tickets para algún evento que quiera asistir y va hacer uso de la web de clientes de Tickbit. Consideramos que el prototipo de cliente estándar es una persona no muy mayor más bien joven, ya que el proceso de interactuar con la web para hacer compras necesitas tener un poco de conocimiento de tecnologías y estar acostumbrado y llevar con soltura una interfaz digital.

4.3.2. ARTISTA U ORGANIZADOR DE EVENTOS

El otro tipo de clientes que puede tener nuestra plataforma son los artistas y los organizadores de eventos los cuales podrán publicar eventos en nuestra plataforma sin necesidad de tantos intermediarios y que se pueden ver atraídos por las comisiones por reventa ya que se llevan un 9% de cada reventa y eso puede suponer más ingresos para ellos.

4.4. FUENTES DE INGRESOS

La plataforma tiene diferentes formas de obtener ingresos las cuales son:

- **Venta de tickets:** Por cada venta de un ticket, Tickbit se lleva una comisión del 5%.
- **Reventa de tickets:** Por cada venta de un ticket, Tickbit se lleva una comisión del 1%.
- **Compra de campañas de publicidad:** El dinero íntegro de lo que cueste la campaña en ese momento ya sea de portada o destacados.

4.5. COMPROMISO ÉTICO

Iniciamos el proyecto con una serie de compromisos éticos que teníamos claro que debían ser parte de la identidad de Tickbit:

- Los tickets no deberían servir para especular. Queremos introducir una reventa controlada, dónde gane el cliente que puede recuperar la mayoría del dinero y gane el dueño del evento que se lleva una comisión por ello. En ningún caso la reventa queremos que lleve un beneficio económico para el dueño del ticket, el objetivo es acabar con la especulación.

- Uno de los principales objetivos es que Tickbit sirva para que artistas y promotores puedan prescindir de muchos intermediarios y abrir nuevas vías al monopolio de los tickets que encontramos en las webs tradicionales, dando igualdad de oportunidades a artistas independientes.

- Consideramos que es importante tener un compromiso con el medio ambiente, y éste fué uno de los motivos por los que utilizamos la red de Polygon. Polygon al ser una red Proof-of-Stake (PoS) es mucho menos contaminante que una red de Proof-of-work (PoW). Aún así, parte de nuestro compromiso es cambiar la red de Tickbit conforme aparezcan nuevas tecnologías y redes menos contaminantes y más eficientes en la blockchain.

5. TECNOLOGÍAS Y SERVICIOS

5.1. REACT.JS

React.js [16] es considerado un framework de código abierto de Javascript que sirve para desarrollar interfaces web de usuario. Este framework se centra principalmente en los componentes, un componente es una pieza de la interfaz del usuario. Con React.js lo que se hace es crear componentes independientes que juntándolos hacen interfaces de usuarios más complejas.

Este framework permite desarrollar aplicaciones más ordenadas y utilizando menos código que si se programase solo usando Javascript, también permite que las diferentes páginas dentro de la web se asocien con datos, si se modifica un dato de una página se actualiza este dato en la web al momento.

5.2. CHAKRA UI

Chakra UI [17] es una librería de componentes la cual proporciona plantillas de componentes predefinidos que facilitan la implementación de interfaces web. Una de las principales características de esta librería es que los componentes pueden recibir estilos y por lo tanto la hace muy personalizable y te otorga más libertad para hacer los diseños que quieras. Además todos estos componentes son adaptables a los diferentes tamaños de pantalla.

5.3. ETHERS.JS

Ethers.js [18] es una librería que sirve para interactuar desde el front-end del proyecto en Javascript a la blockchain de Ethereum y su ecosistema. En nuestro caso hemos podido conectar la interfaz de la web a las funciones que tenemos programadas en los Smart Contract.

La información que se envía a los nodos que són los encargados de validar las transacciones que realizamos desde la plataforma se hace mediante el protocolo JSON-RPC [19].

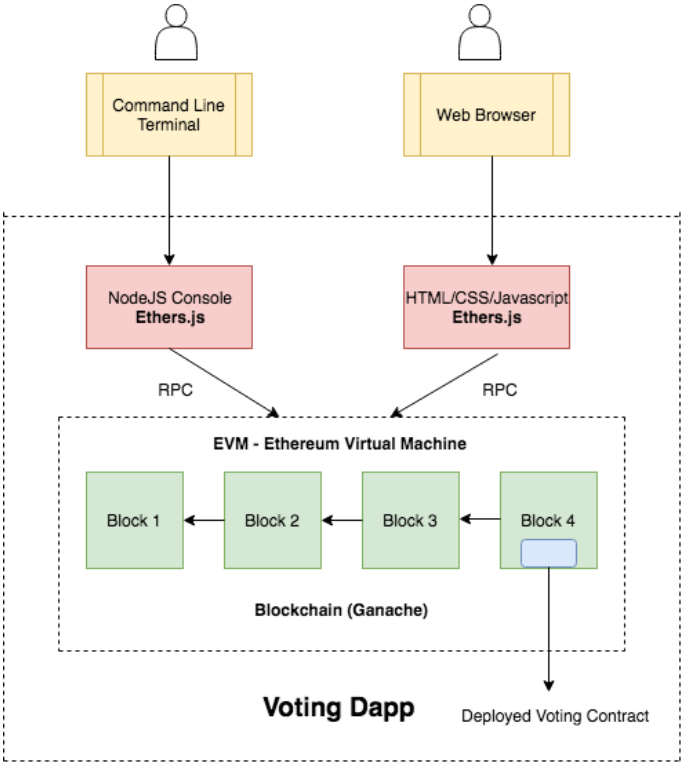


Figura 20: Comunicación entre la plataforma y la blockchain mediante la librería Ethers.js
Fuente: medium.com

5.4. GITHUB

Github [20] es una web que sirve a los desarrolladores para alojar el código de las aplicaciones y mantener un control de las diferentes versiones según se van desarrollando. Al poder mantener diferentes versiones del proyecto se puede comparar el código entre ellas, fusionar cambios o incluso restaurar antiguas versiones por si pasara algo. Además permite trabajar con distintas ramas en paralelo en un proyecto, por ejemplo puedes tener una rama de desarrollo en la cual puedes probar nuevos cambios y una de desarrollo donde se lanzarán los nuevos cambios si estos funcionan correctamente.

5.5. NPM/YARN

NPM [21] y Yarn [22] son los dos gestores de paquetes que hemos utilizado. Los gestores de paquetes mantienen un registro de las librerías instaladas en el proyecto, y permite instalar software nuevo, actualizarlo a versiones más recientes, o eliminar librerías de una manera sencilla.

5.6. BINANCE API

Binance [23] es una de las mayores plataformas de compra venta de criptomonedas en la actualidad. Esta plataforma ofrece una API que nos proporciona diferente información sobre todas las monedas que tienen listadas en su plataforma.

Esta API nos ha sido útil porque necesitamos saber el precio de la moneda MATIC con la que realizamos los pagos ya que a la hora de hacerlos se proporciona una conversión del dinero a MATIC para saber de cuantos se tiene que disponer en la billetera.

Haciendo una petición al siguiente enlace:

https://api.binance.com/api/v3/ticker/price?symbol=MATICUSDT

Obtenemos esta respuesta:

```
{"symbol":"MATICUSDT","price":"0.84250000"}
```

En la cual obtenemos el símbolo de la conversión y el precio de un MATIC en dólares.

5.7. DONDOMINIO

Dondominio [24] es una plataforma web que ofrece dominios para webs. Nosotros la utilizamos para comprar el dominio “tickb.it”.

5.8. FIREBASE

Firebase [25] es una plataforma web de Google que ofrece servicios web cloud para empresas. En este caso nosotros lo utilizamos cómo hosting para las webs en su plan gratuito.

5.9. NETLIFY

Netlify [26] es una plataforma web que ofrece servicios web cloud, muy similar a Firebase, pero gratis para las funciones en la nube.

La usamos como servidor para alojar la función que envía el mail del formulario que se encuentra en el apartado ayuda de la web. La función que alojamos es la siguiente:

```
async function sendMail(descriptor) {
  // create reusable transporter object using the default SMTP
     transport
  let transporter = nodemailer.createTransport({
    host: "smtp.dondominio.com",
    port: 465,
    secure: true, // true for 465, false for other ports
    auth: {
      user: process.env.USER_SMTP, // generated ethereal user
      pass: process.env.PASS_SMTP, // generated ethereal password
    },
  });

  // send mail with defined transport object
  let info = await transporter.sendMail(descriptor);

  console.log("Message sent: %s", info.messageId);
}


const handler = async (event) => {
  var body = JSON.parse(event.body)

  const descriptor = {
    from: '"' + body.name + '" <' + process.env.USER_SMTP + '>',
    to: process.env.FORM_MAIL, // list of receivers
    subject: "✉ [FORM] " + body.subject, // Subject line
    replyTo: '"' + body.name + '" <' + body.email + '>',
    html: "<b>De:</b> " + body.name + " &#60;" + body.email +
    "&#62; &#40;" + body.phone + "&#41; " + "<br><hr><br>" +
    body.message.replace(/\n/g, "<br />") + "<br><br>", // html body
  }

  try {
    await sendMail(descriptor)
    return { statusCode: 200, body: 'Success!', headers:
    {"Access-Control-Allow-Origin": "*"} }
  } catch (error) {
    console.log(error.message)
    return { statusCode: 500, body: error.message, headers:
    {"Access-Control-Allow-Origin": "*"} }
  }
```

```
}

module.exports = { handler }
```

Figura 21: Funciones Netlify del formulario de contacto
Fuente: send-email.js

El código genera un correo electrónico con el cuerpo que el usuario rellena en el formulario de la web y lo envía contact@tickb.it.

5.10. TECNOLOGÍAS BLOCKCHAIN

5.10.1. SMART CONTRACTS

Como hemos explicado anteriormente los Smart Contracts en Ethereum son un código escrito en un lenguaje de programación llamado Solidity que ejecuta funciones de manera autónoma y automática en la EVM. Cada Smart Contract se puede representar como una clase en un lenguaje de programación orientado a objetos, donde esta contiene diferentes estructuras de datos para almacenar información, objetos con distintas variables, funciones que se ejecutan de manera manual o automática y eventos que publican cierta información sobre las transacciones en la blockchain.

Además en estos contratos podemos establecer qué persona o grupos de personas pueden ejecutar las funciones que hemos programado. Por ejemplo podemos establecer que una función solo la ejecute el creador del contrato para que pueda obtener más información sobre lo que tenemos almacenado en las variables del contrato o que la pueda ejecutar todo el mundo como por ejemplo una función para comprar un ticket de un evento o consultar tus tickets.

En nuestro proyecto hemos seleccionado un intervalo de versiones de Solidity para que no surja ningún problema de versiones si se actualiza en un futuro y así el contrato pueda seguir operativo sin necesidad de cambios. La versión que hemos utilizado ha sido >=0.7.0 <0.9.0, es decir cualquier versión que vaya desde la 0.7.0 para arriba y desde la anterior a la 0.9.0 hacia abajo.

Para nuestro proyecto hemos necesitado utilizar dos Smarts Contracts diferentes los cuales estarán alojados y funcionando en la red de pruebas de la blockchain Polygon, que es la Polygon Mumbai Testnet. Los contratos son los siguientes:

- **Tickbit.sol**: en este contrato están programadas las funciones para crear, modificar, eliminar y visualizar eventos y comprar campañas para promocionarlos.

- **TickbitTicket.sol**: se encarga de toda la parte del ticketing, ya sea crear, validar o vender tickets comprados.

Para mantener el formato del documento, la implementación y la explicación al detalle de los contratos se muestra en el apartado 1 del anexo, el cual se recomienda revisar para poder entender exactamente qué funcionalidades realizan.

5.10.1.1. ORÁCULOS

Los contratos inteligentes están implementados dentro de la blockchain y por lo tanto solo se conoce la información que se procesa dentro de esta. A veces se necesita información del mundo exterior para poder realizar ciertas funciones, para este propósito se crearon los llamados oráculos, los oráculos no són nada más que piezas de código que sirven de intermediarios entre el mundo real y la blockchain.

En nuestro proyecto hemos necesitado los oráculos para poder realizar tanto las compras o re-ventas de tickets y la compra de campañas, ya que realizamos los pagos con Matic que es el token nativo de la red de Polygon y normalmente este tipo de tokens tienen una alta volatilidad porque el precio fluctúa constantemente haciendo que varíe mucho en poco tiempo. Por lo tanto es necesario tener una actualización constante del precio del token Matic en dólares para poder realizar el pago al precio en el que se encuentra en ese momento el token.

Para obtener esta información hemos hecho uso de los oráculos de Chainlink [27], una red de nodos descentralizada que actúa como middleware entre los Smarts Contracts y el mundo real, que da soporte a varias blockchain entre ellas Polygon Mumbai que es la que estamos utilizando y que nos proporciona la conversión de precio a dólares de diferentes divisas a tiempo real.

5.10.2. POLYGON MUMBAI FAUCET

Como hemos mencionado anteriormente en las redes de pruebas de las blockchain se trabaja con dinero ficticio que emula el token oficial de la red principal. Para poder hacer pruebas sobre el desarrollo realizado se necesita de este dinero ficticio para poder pagar las comisiones de las transacciones de las pruebas que se hagan.

Para esto existe lo que se denomina como faucet, que normalmente es una página web en la cual puedes conseguir una pequeña cantidad de tokens ficticios poniendo la dirección de tu billetera. En nuestro caso hemos hecho uso de la Mumbai Faucet de Alchemy [28].

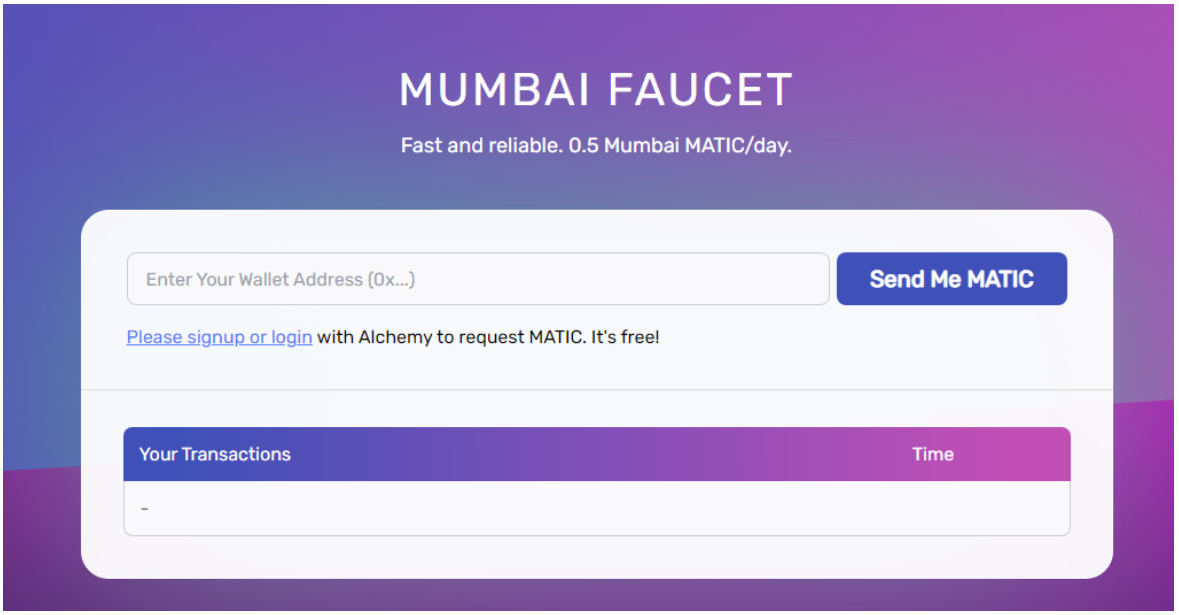


Figura 22: Página principal de la Mumbai Faucet de Alchemy
Fuente: mumbaifaucet.com

5.10.3. POLYGON SCAN

Las transacciones en las blockchain son públicas y por lo tanto todas las transacciones quedan grabadas como en una especie de libro de cuentas, cada blockchain tiene el suyo propio.

En este caso la Testnet de Polygon Mumbai tiene PolygonScan [29], una web que permite explorar y buscar transacciones, direcciones, tokens, precios y otras actividades en la cadena de bloques de Polygon.

Con esta web podremos verificar que las transacciones que se realizan en la plataforma de Tickbit se producen de forma correcta y podremos ver como se mueven los saldos y tokens de billetera a billetera.

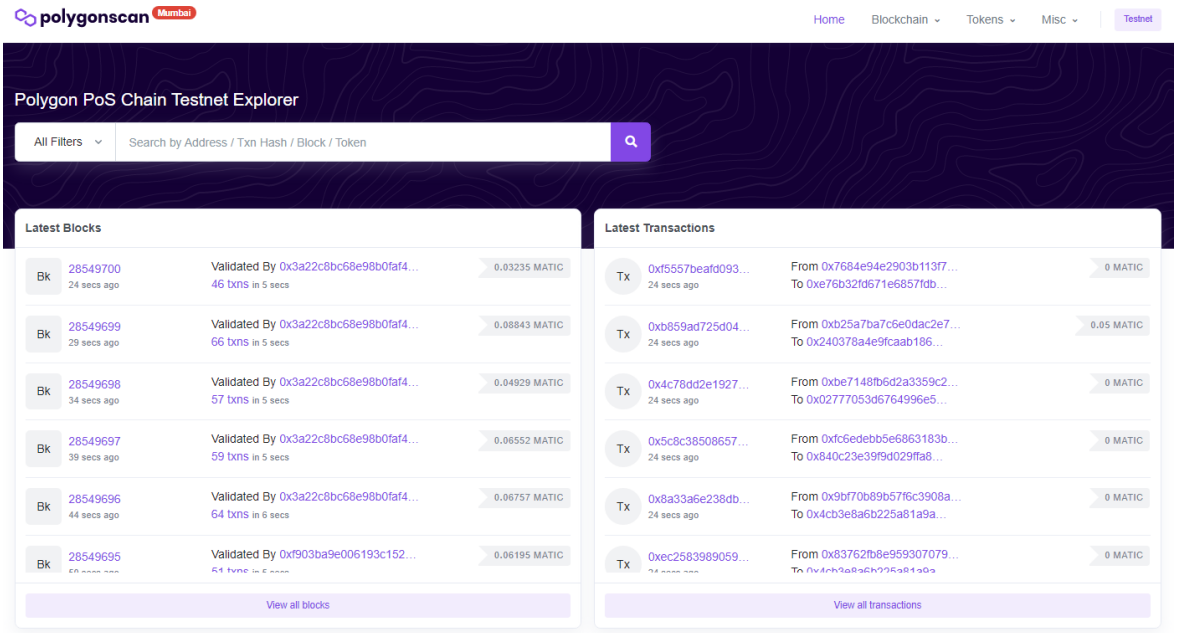


Figura 23: Página principal de la web Mumbai PolygonScan
Fuente: mumbai.polygonscan.com

5.10.4. METAMASK

Metamask [30] es una extensión de navegador y aplicación de teléfono móvil que permite tener múltiples billeteras en diferentes blockchain. Esta extensión permite la interacción de usuarios con plataformas que funcionan sobre la blockchain y realizar transacciones.

Permite tanto crear diferentes billeteras en una misma cuenta cómo importar billeteras previamente ya creadas mediante las claves privadas. Por defecto estas billeteras se crean sobre la blockchain de Ethereum, pero Metamask permite cambiar la red desde la configuración. Como nuestro proyecto funciona sobre la blockchain de Mumbai Polygon en el punto 2 del anexo adjuntamos un tutorial sobre cómo configurarla.

5.10.5. ALCHEMY

Alchemy [31] es una plataforma que ofrece servicios de desarrollo para plataformas que implementan Web3. Uno de los servicios que ofrece es el de crear un nodo que aloja e implementa tus contratos en la blockchain.

Por lo tanto, la función de Alchemy es facilitar la implementación de aplicaciones en la blockchain. Es lo equivalente a un servidor cloud que aloja una API en los sistemas tradicionales.

Esta es la plataforma que hemos escogido como nodo para alojar los contratos de la blockchain, ya que ofrece un servicio base gratuito.

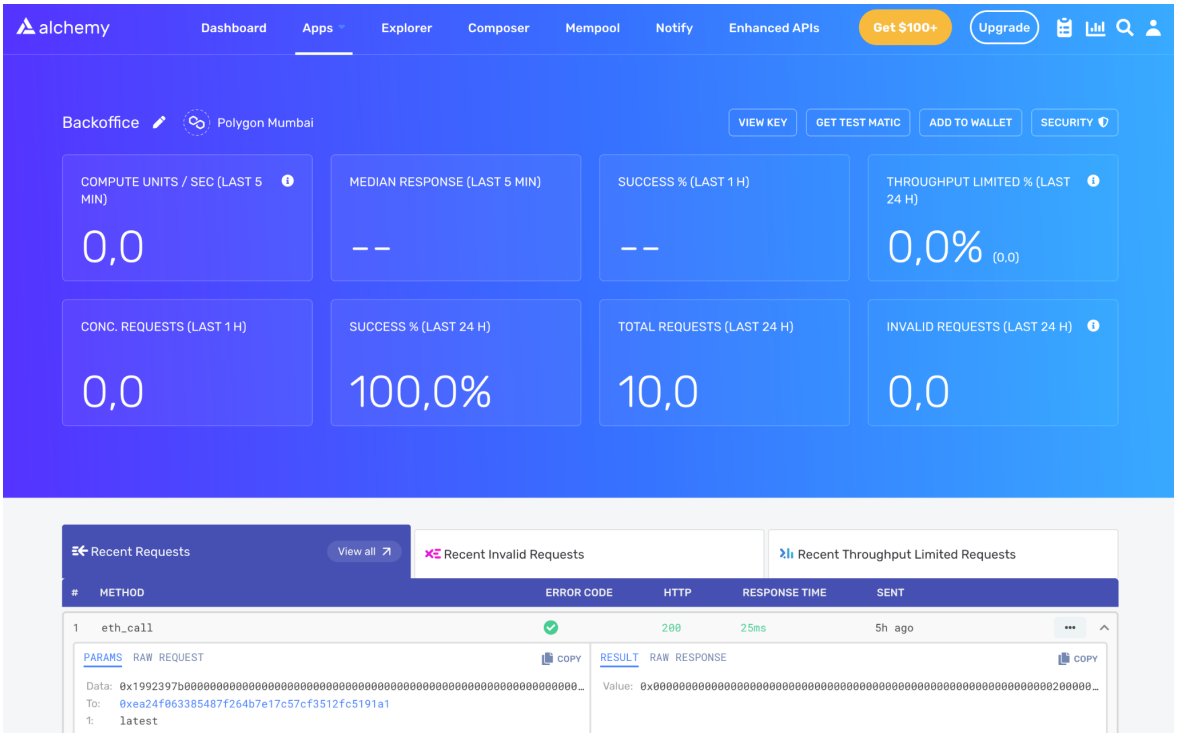


Figura 24: Página de usuario de la web de Alchemy.
Fuente: alchemy.com

5.10.6. HARDHAT

Ethereum Hardhat [32] es un cliente de Ethereum que proporciona una manera fácil de implementar, administrar y monitorear aplicaciones de Ethereum. Es muy útil para implementar proyectos en local, ya que te permite virtualizar una blockchain en un entorno local, esto a veces imprescindible ya que si se despliega el proyecto en la blockchain directamente los cambios son irreversibles.

Lo hemos usado durante todo el desarrollo para implementar los contratos y testearlos en una red local.

El proyecto que contiene el código que gestiona el cliente de hardhat se puede encontrar aquí:

https://github.com/tickbit-dev/tickbit-solidity

También nos facilita la opción de subir nuestro contrato a los nodos para que opere en la blockchain a partir de la clave privada de Alchemy, este proceso está configurado en el fichero hardhat.config.js.

Para poder hacer pruebas en local de un smart contract en una web es necesario hacer una serie de pasos. Lo primero de todo es abrir el proyecto adjuntado en el link de github, una vez abierto abrir dos terminales, si es la primera vez que lo abres es necesario hacer en un terminal:

```
npm install
```

Una vez hecho este comando se habrán descargado todos los paquetes necesarios para su correcta ejecución. Seguidamente se ejecuta el siguiente comando en el mismo terminal para poder compilar los contratos y ver que no contienen ningún error de código.

```
npx hardhat node
```

Si no nos ha dado ningún error de compilación vamos ahora a emular un nodo de una blockchain en local con el siguiente comando en el otro terminal:

```
npx hardhat run scripts/deploy.js --network localhost
```

Con esto habremos ejecutado el script deploy.js en el cual se especifica cual es el contrato que se va a desplegar en nuestro nodo local y que se va a hacer en la red localhost. Una vez desplegado el contrato se nos habrán generado unas billeteras de pruebas con fondos para hacer pruebas que podremos utilizar en la extensión Metamask.

Una vez finalizado se nos proporcionará una dirección que es la dirección del contrato la cual podremos utilizar para hacer llamadas desde el código de la página web.

6. ARQUITECTURA E IMPLEMENTACIÓN DEL PROTOTIPO

6.1. ESQUEMA GENERAL

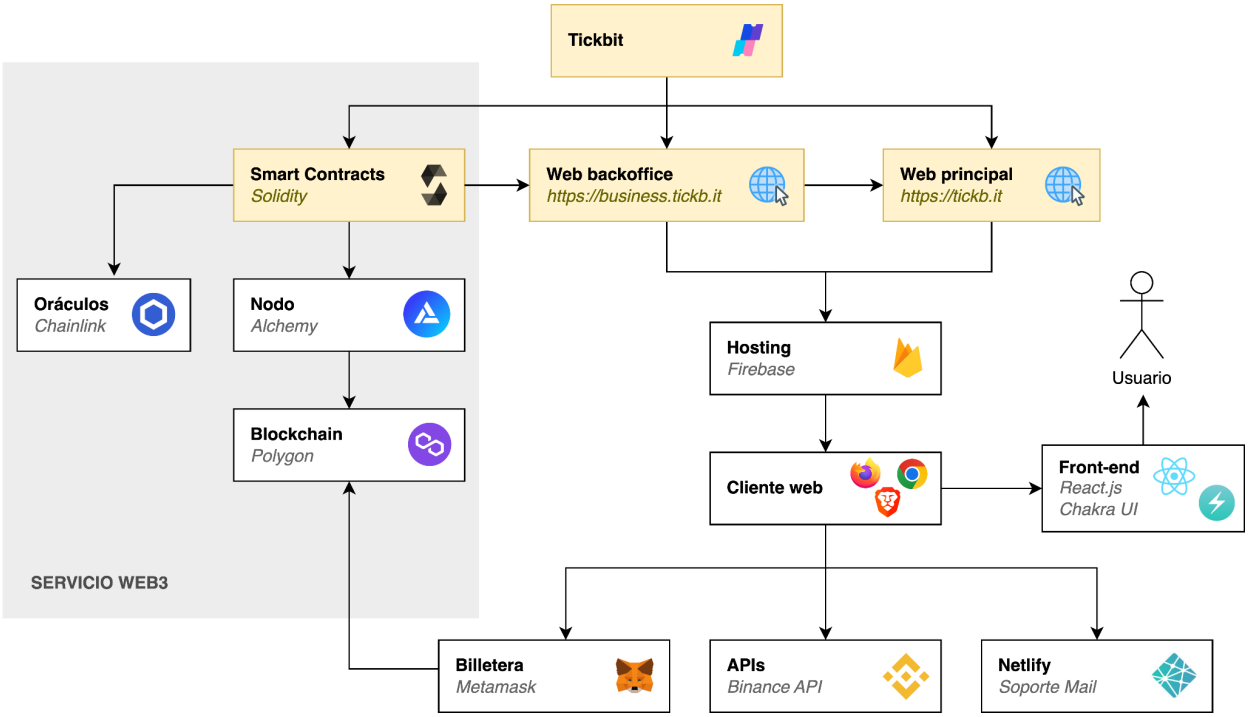


Figura 25: Arquitectura de la plataforma Tickbit
Fuente: medium.com

En la figura '25' podemos observar que la plataforma Tickbit se divide principalmente en dos grandes partes.

La primera que forma el servicio Web3 són los **smart contracts**, estos están programados en Solidity y están alojados en un nodo de la blockchain Mumbai Polygon que nos proporciona la plataforma Alchemy, además estos contratos reciben información del exterior grácias a los oráculos de Chainlink para operaciones de cambio de divisas.

La segunda está formada por la **web backoffice** que es la plataforma de gestión de eventos para organizadores y la **web principal** para clientes. Estas dos webs están alojadas en un hosting de Firebase y se pueden acceder desde cualquier navegador a través del Front-end desarrollado en React.js y Chakra UI. Además la web principal recibe información de la API de Binance para obtener la equivalencia de dólares a Matic y hace uso del servicio de Netlify para la parte de Soporte Mail de la web. Tanto la web principal como la web backoffice utilizan y reciben información de la extensión Metamask para realizar y firmar las transacciones que producen los usuarios y poder ejecutarlas en la blockchain.

6.2. CASOS DE USO

Después de ver cómo está estructurada la plataforma, en este apartado se presenta la especificación de los casos de uso existentes. Se dividen en dos grupos, los casos de uso de acciones y los de consultas.

6.2.1. ACCIONES

<table>
<tr><td colspan="2" rowspan="2">Caso de uso</td><td>Identificador</td><td>Nombre</td></tr>
<tr><td>CU-001</td><td>Crear nuevo evento</td></tr>
<tr><td colspan="2">Descripción</td><td colspan="2">El usuario crea un nuevo evento.</td></tr>
<tr><td colspan="2">Precondición</td><td colspan="2">1. Existe un usuario identificado con una billetera.
2. La billetera conectada dispone de los fondos suficientes tal que el saldo sea mayor o igual al importe de las comisiones para realizar una transacción.</td></tr>
<tr><td rowspan="6">Secuencia</td><td>Paso</td><td colspan="2">Acción</td></tr>
<tr><td>1</td><td colspan="2">El usuario pulsa el botón de “Crear evento” del lateral izquierdo de la página.</td></tr>
<tr><td>2</td><td colspan="2">El usuario rellena el formulario con la información del evento</td></tr>
<tr><td>3</td><td colspan="2">El usuario pulsa el botón de “Crear evento” de la parte inferior de la página.</td></tr>
<tr><td>4</td><td colspan="2">Se almacena toda la información del evento y se asigna como propietario del evento la billetera del usuario.</td></tr>
<tr><td>5</td><td colspan="2">Se muestra un mensaje por pantalla indicando que el evento se ha creado correctamente.</td></tr>
<tr><td colspan="2">Postcondición</td><td colspan="2">1. Se ha creado el evento correctamente.</td></tr>
<tr><td colspan="2">Comentario</td><td colspan="2">Se explica el proceso y se muestra el código en el punto 1 del Anexo.</td></tr>
</table>

<table>
<tr><td colspan="2" rowspan="3">Caso de uso</td><td>Identificador</td><td>Nombre</td></tr>
<tr><td></td><td></td></tr>
<tr><td>CU-002</td><td>Modificar evento</td></tr>
<tr><td colspan="2">Descripción</td><td colspan="2">El usuario modifica la información de un evento ya existente.</td></tr>
<tr><td colspan="2">Precondición</td><td colspan="2">1. Existe un usuario identificado con una billetera.
2. La billetera conectada dispone de los fondos suficientes tal que el saldo sea mayor o igual al importe de las comisiones para realizar una transacción.
3. El evento a modificar existe previamente.
4. El usuario debe ser el creador del contrato o el creador del evento.</td></tr>
<tr><td rowspan="6">Secuencia</td><td>Paso</td><td colspan="2">Acción</td></tr>
<tr><td>1</td><td colspan="2">El usuario selecciona el evento a modificar en la pestaña de “Eventos”</td></tr>
<tr><td>2</td><td colspan="2">El usuario modifica el formulario con la información del evento</td></tr>
<tr><td>3</td><td colspan="2">El usuario pulsa el botón de “Modificar evento” de la parte inferior de la página.</td></tr>
<tr><td>4</td><td colspan="2">Se almacena toda la información del evento con los nuevos cambios.</td></tr>
<tr><td>5</td><td colspan="2">Se muestra un mensaje por pantalla indicando que el evento se ha modificado correctamente.</td></tr>
<tr><td colspan="2">Postcondición</td><td colspan="2">1. Se ha modificado el evento correctamente.</td></tr>
<tr><td colspan="2">Comentario</td><td colspan="2">Se explica el proceso y se muestra el código en el punto 1 del Anexo.</td></tr>
</table>

<table>
<tr><td colspan="2" rowspan="2">Caso de uso</td><td>Identificador</td><td>Nombre</td></tr>
<tr><td>CU-003</td><td>Eliminar evento</td></tr>
<tr><td colspan="2">Descripción</td><td colspan="2">El usuario elimina un evento existente.</td></tr>
<tr><td colspan="2">Precondición</td><td colspan="2">1. Existe un usuario identificado con una billetera.
2. La billetera conectada dispone de los fondos suficientes tal que el saldo sea mayor o igual al importe de las comisiones para realizar una transacción.
3. El evento a eliminar existe previamente.
4. El usuario debe ser el creador del contrato o el creador del evento.</td></tr>
<tr><td rowspan="5">Secuencia</td><td>Paso</td><td colspan="2">Acción</td></tr>
<tr><td>1</td><td colspan="2">El usuario selecciona el evento a eliminar en la pestaña de “Eventos”</td></tr>
<tr><td>2</td><td colspan="2">El usuario pulsa el botón de “Eliminar evento” de la parte inferior de la página.</td></tr>
<tr><td>3</td><td colspan="2">Se modifica la información de elemento y se marca como eliminado.</td></tr>
<tr><td>4</td><td colspan="2">Se muestra un mensaje por pantalla indicando que el evento se ha eliminado correctamente.</td></tr>
<tr><td colspan="2">Postcondición</td><td colspan="2">1. Se ha eliminado el evento correctamente.</td></tr>
<tr><td colspan="2">Comentario</td><td colspan="2">Se explica el proceso y se muestra el código en el punto 1 del Anexo.</td></tr>
</table>

<table>
<tr><td colspan="2" rowspan="2">Caso de uso</td><td>Identificador</td><td>Nombre</td></tr>
<tr><td>CU-004</td><td>Restaurar evento</td></tr>
<tr><td colspan="2">Descripción</td><td colspan="2">El usuario elimina un evento existente.</td></tr>
<tr><td colspan="2">Precondición</td><td colspan="2">1. Existe un usuario identificado con una billetera.
2. La billetera conectada dispone de los fondos suficientes tal que el saldo sea mayor o igual al importe de las comisiones para realizar una transacción.
3. El evento a restaurar existe y ha sido eliminado previamente.
4. El usuario debe ser el creador del contrato.</td></tr>
<tr><td rowspan="5">Secuencia</td><td>Paso</td><td colspan="2">Acción</td></tr>
<tr><td>1</td><td colspan="2">El usuario selecciona el evento a restaurar en la pestaña de “Eventos”</td></tr>
<tr><td>2</td><td colspan="2">El usuario pulsa el botón de “Restaurar evento” de la parte inferior de la página.</td></tr>
<tr><td>3</td><td colspan="2">Se modifica la información de elemento y se marca como no eliminado.</td></tr>
<tr><td>4</td><td colspan="2">Se muestra un mensaje por pantalla indicando que el evento se ha restaurado correctamente.</td></tr>
<tr><td colspan="2">Postcondición</td><td colspan="2">1. Se ha restaurado el evento correctamente.</td></tr>
<tr><td colspan="2">Comentario</td><td colspan="2">Se explica el proceso y se muestra el código en el punto 1 del Anexo.</td></tr>
</table>

<table>
<tr><td colspan="2" rowspan="2">Caso de uso</td><td>Identificador</td><td>Nombre</td></tr>
<tr><td>CU-005</td><td>Comprar campaña</td></tr>
<tr><td colspan="2">Descripción</td><td colspan="2">El usuario compra una campaña para un evento.</td></tr>
<tr><td colspan="2">Precondición</td><td colspan="2">1. Existe un usuario identificado con una billetera.
2. La billetera conectada dispone de los fondos suficientes tal que el saldo sea mayor o igual al del precio de la campaña.
3. El evento sobre el cual vas a comprar la campaña debe haber sido creado previamente.
4. El usuario debe ser el creador del contrato o el creador del evento.
5. Han de existir campañas disponibles para el rango de fechas que se quiera seleccionar.
6. El importe de la compra en el momento de efectuar la operación debe ser igual al establecido en los tickets con una variación máxima de ±1 USD en el cambio de divisas.</td></tr>
<tr><td rowspan="6">Secuencia</td><td>Paso</td><td colspan="2">Acción</td></tr>
<tr><td>1</td><td colspan="2">El usuario selecciona el evento y el rango de fechas sobre el cual quiere comprar la campaña en el apartado “Campañas”</td></tr>
<tr><td>2</td><td colspan="2">El usuario pulsa el botón de “Comprar”.</td></tr>
<tr><td>3</td><td colspan="2">Se almacena la información de la campaña junto con el evento y la billetera del comprador.</td></tr>
<tr><td>4</td><td colspan="2">Se transfiere el dinero del pago de la campaña desde la billetera del comprador a la billetera del creador del contrato.</td></tr>
<tr><td>5</td><td colspan="2">Se muestra un mensaje por pantalla indicando que la campaña se ha comprado correctamente.</td></tr>
<tr><td colspan="2">Postcondición</td><td colspan="2">1. Se ha comprado la campaña correctamente.</td></tr>
<tr><td colspan="2">Comentario</td><td colspan="2">Se explica el proceso y se muestra el código en el punto 1 del Anexo.</td></tr>
</table>

<table>
<tr><td colspan="2" rowspan="2">Caso de uso</td><td>Identificador</td><td>Nombre</td></tr>
<tr><td>CU-006</td><td>Comprar ticket/s</td></tr>
<tr><td colspan="2">Descripción</td><td colspan="2">El usuario compra uno o más tickets disponibles para un evento.</td></tr>
<tr><td colspan="2">Precondición</td><td colspan="2">1. Existe un usuario identificado con una billetera.
2. La billetera conectada dispone de los fondos suficientes tal que el saldo sea mayor o igual al importe de la compra y los gastos de comisión de la transacción.
3. El número de tickets de la compra no supera 5.
4. El importe de la compra en el momento de efectuar la operación debe ser igual al establecido en los tickets con una variación máxima de ±1 USD en el cambio de divisas.
5. El número de tickets de la compra, no supera la disponibilidad total de tickets para ese evento (según aforo).</td></tr>
<tr><td rowspan="8">Secuencia</td><td>Paso</td><td colspan="2">Acción</td></tr>
<tr><td>1</td><td colspan="2">El usuario selecciona el evento a comprar.</td></tr>
<tr><td>2</td><td colspan="2">El usuario pulsa el botón de “Comprar tickets”.</td></tr>
<tr><td>3</td><td colspan="2">El usuario selecciona el número de tickets que quiere comprar y presiona el botón “Pagar”.</td></tr>
<tr><td>4</td><td colspan="2">Se transfiere una parte del valor de los tickets desde la billetera del comprador a la billetera del creador del evento y otra parte a la billetera del creador del contrato.</td></tr>
<tr><td>5</td><td colspan="2">Se genera un NFT para cada ticket comprado del evento y se les asigna como propietario la billetera que realiza la compra.</td></tr>
<tr><td>6</td><td colspan="2">Los tickets se transfieren a la billetera del comprador.</td></tr>
<tr><td>7</td><td colspan="2">Se muestra un mensaje por pantalla indicando que la compra se ha realizado correctamente.</td></tr>
<tr><td colspan="2">Postcondición</td><td colspan="2">1. El comprador ha recibido uno o más tickets únicos para el evento.
2. El creador del evento ha recibido su parte de la venta de los tickets.</td></tr>
</table>

	3. Tickbit ha recibido la comisión por venta.
Comentario	Se explica el proceso y se muestra el código en el punto 1 del Anexo.

Caso de uso		Identificador	Nombre
		CU-007	Revender ticket
Descripción		El usuario pone un ticket de su propiedad como disponible para reventa.	
Precondición		1. Existe un usuario identificado con una billetera. 2. La billetera conectada dispone de los fondos suficientes tal que el saldo sea mayor o igual al importe de las comisiones para realizar una transacción. 3. El usuario es el propietario del ticket. 4. El ticket no ha sido previamente validado. 5. El ticket no está actualmente en reventa.	
Secuencia	Paso	Acción	
	1	El usuario identifica el ticket a poner en reventa.	
	2	El usuario pulsa el botón de “Poner ticket a la venta”.	
	3	El usuario confirma la acción haciendo click en “Poner ticket a la venta”.	
	4	El ticket se marca como “En venta” y aparece en la disponibilidad del evento.	
Postcondición		1. El ticket del usuario se ha puesto a la venta y puede ser comprado en la página del evento.	
Comentario		Se explica el proceso y se muestra el código en el punto 1 del Anexo.	

<table>
<tr><td colspan="2" rowspan="2">Caso de uso</td><td>Identificador</td><td>Nombre</td></tr>
<tr><td>CU-008</td><td>Cancelar reventa</td></tr>
<tr><td colspan="2">Descripción</td><td colspan="2">El usuario quita un ticket de su propiedad de la reventa.</td></tr>
<tr><td colspan="2">Precondición</td><td colspan="2">1. Existe un usuario identificado con una billetera.
2. La billetera conectada dispone de los fondos suficientes tal que el saldo sea mayor o igual al importe de las comisiones para realizar una transacción.
3. El usuario es el propietario del ticket.
4. El ticket se encuentra actualmente en reventa.</td></tr>
<tr><td rowspan="5">Secuencia</td><td>Paso</td><td colspan="2">Acción</td></tr>
<tr><td>1</td><td colspan="2">El usuario identifica el ticket a quitar de la reventa.</td></tr>
<tr><td>2</td><td colspan="2">El usuario pulsa el botón de “Cancelar venta”.</td></tr>
<tr><td>3</td><td colspan="2">El usuario confirma la acción haciendo click en “Cancelar venta”.</td></tr>
<tr><td>4</td><td colspan="2">El ticket se quita de la venta y desaparece de la disponibilidad del evento.</td></tr>
<tr><td colspan="2">Postcondición</td><td colspan="2">1. El ticket del usuario se ha quitado de la venta y ya no puede ser comprado en la página del evento.</td></tr>
<tr><td colspan="2">Comentario</td><td colspan="2">Se explica el proceso y se muestra el código en el punto 1 del Anexo.</td></tr>
</table>

Caso de uso		Identificador	Nombre
		CU-009	Comprar ticket/s reventa
Descripción		El usuario compra uno o más tickets en reventa.	
Precondición		1. Existe un usuario identificado con una billetera. 2. La billetera conectada dispone de los fondos suficientes tal que el saldo sea mayor o igual al importe de la compra y los gastos de comisión de la transacción. 3. El número de tickets disponibles (de no reventa) para el evento es 0. 4. El número de tickets en reventa para el evento es superior a 1. 5. El número de tickets de la compra no supera el número de tickets disponibles en reventa. 6. El importe de la compra en el momento de efectuar la operación debe ser igual al establecido en los tickets con una variación máxima de ±1 USD en el cambio de divisas.	
Secuencia	Paso	Acción	
	1	El usuario selecciona el evento a comprar.	
	2	El usuario pulsa el botón de “Comprar tickets”.	
	3	El usuario selecciona el número de tickets que quiere comprar y presiona el botón “Pagar”.	
	4	Se transfiere una parte del valor de los tickets desde la billetera del comprador a la billetera del creador del evento, otra parte a la billetera del creador del contrato y por último el dinero restante a la billetera del usuario que ha publicado la reventa.	
	5	Se asigna como nuevo propietario de los tickets al comprador.	
	6	Los tickets se transfieren a la billetera del comprador.	
	7	Se muestra un mensaje por pantalla indicando que la compra se ha realizado correctamente.	

Postcondición	1. El comprador ha recibido uno o más tickets únicos para el evento. 2. El creador del evento ha recibido su parte de la venta de los tickets. 3. Tickbit ha recibido la comisión por venta. 4. El usuario que ha publicado la reventa ha recibido el dinero de la reventa.
Comentario	Se explica el proceso y se muestra el código en el punto 1 del Anexo.

Caso de uso		Identificador	Nombre
		CU-010	Validar ticket
Descripción		El usuario confirma la validez de un ticket y añade una instancia al registro de validaciones.	
Precondición		1. Existe un usuario identificado con una billetera. 2. La billetera conectada dispone de los fondos suficientes tal que el saldo sea mayor o igual al importe de las comisiones para realizar una transacción. 3. El usuario es el dueño del ticket. 4. El ticket corresponde al evento. 5. El ticket no ha sido marcado como validado previamente. 6. El ticket no está a la venta en el momento de validar.	
Secuencia	Paso	Acción	
	1	El usuario identifica el ticket a validar.	
	2	El usuario pulsa el botón de “Validar ticket”.	
	3	El usuario lee el código QR que contiene un hash único de validación.	
	4	Se añade una instancia al registro de validaciones con el hash de validación leído en el código QR.	
	5	Se muestra un mensaje por pantalla indicando que la validación del ticket ha sido registrada correctamente.	

Postcondición	1. Se ha creado una instancia de validación en el registro de validaciones con un hash único.
Comentario	Se explica el proceso y se muestra el código en el punto 1 del Anexo.

Caso de uso		Identificador	Nombre
		CU-011	Comprobar validación de ticket
Descripción		El usuario confirma si existe en el registro, una instancia de validación de un ticket que corresponde al hash de validación actual.	
Precondición		1. Existe un usuario identificado con una billetera. 2. La billetera conectada dispone de los fondos suficientes tal que el saldo sea mayor o igual al importe de las comisiones para realizar una transacción. 3. El usuario es el dueño del evento.	
Secuencia	Paso	Acción	
	1	El usuario selecciona selecciona uno de sus eventos y opcionalmente configura la Secret Key de la cartera.	
	2	El usuario pulsa el botón “Empezar validación”.	
	3	Si el usuario no ha configurado el Secret Key, firma la confirmación de la validación.	
	4	El ticket se marca como validado.	
	5	Se muestra un mensaje por pantalla indicando que el ticket se ha marcado como validado correctamente.	
Postcondición		1. El ticket se ha marcado como validado.	
Comentario		Se explica el proceso y se muestra el código en el punto 1 del Anexo.	

Caso de uso		Identificador	Nombre
		CU-012	Enviar mail a soporte
Descripción		El usuario envía un mail a contact@tickb.it	
Precondición		-	
Secuencia	Paso	Acción	
	1	El usuario rellena el formulario en la sección de contacto de la web.	
	2	El usuario pulsa el botón “Enviar mensaje”.	
Postcondición		1. Se ha enviado un mensaje con el contenido del formulario a contact@tickb.it	
Comentario		Se explica el proceso y se muestra el código en el punto 5.9 del documento.	

6.2.2. CONSULTAS

Caso de uso	Identificador	Nombre
	CU-013	Consultar evento
Descripción	El usuario recibe la información de un evento.	
Comentario	Se explica el proceso y se muestra el código en el punto 1 del Anexo.	

Caso de uso	Identificador	Nombre
	CU-014	Consultar eventos

Descripción	El usuario recibe la información de todos los eventos.
Comentario	Se explica el proceso y se muestra el código en el punto 1 del Anexo.

Caso de uso	Identificador	Nombre
	CU-015	Consultar campañas
Descripción	El usuario recibe la información de todas las campañas.	
Comentario	Se explica el proceso y se muestra el código en el punto 1 del Anexo.	

Caso de uso	Identificador	Nombre
	CU-016	Consultar disponibilidad
Descripción	El usuario recibe la disponibilidad de tickets para un evento concreto.	
Comentario	Se explica el proceso y se muestra el código en el punto 1 del Anexo.	

Caso de uso	Identificador	Nombre
	CU-017	Consultar ventas de tickets
Descripción	El usuario recibe la información de los tickets vendidos.	
Comentario	Se explica el proceso y se muestra el código en el punto 1 del Anexo.	

Caso de uso	Identificador	Nombre
	CU-018	Consultar tickets
Descripción	El usuario recibe la información de todos sus tickets.	
Comentario	Se explica el proceso y se muestra el código en el punto 1 del Anexo.	

Caso de uso	Identificador	Nombre
	CU-019	Consultar ingresos por reventas
Descripción	El usuario recibe la información de los ingresos recibidos por	

	reventas.
Comentario	Se explica el proceso y se muestra el código en el punto 1 del Anexo.

Caso de uso	Identificador	Nombre
	CU-020	Consultar disponibilidad de reventa
Descripción	El usuario recibe la disponibilidad de tickets en reventa para un evento concreto.	
Comentario	Se explica el proceso y se muestra el código en el punto 1 del Anexo.	

Caso de uso	Identificador	Nombre
	CU-021	Consultar validación
Descripción	El usuario recibe la información de la validación de un ticket.	
Comentario	Se explica el proceso y se muestra el código en el punto 1 del Anexo.	

6.3. PANTALLAS, NAVEGACIÓN Y RESPONSABILIDADES

Una vez expuesto cómo se estructura la plataforma y los casos de uso que tiene esta, en el presente apartado se va a mostrar en forma de esquema las diferentes vistas y las responsabilidades que tiene cada una de ellas dentro de la plataforma con la referencia al caso de uso que le corresponde.

- **Web principal:**

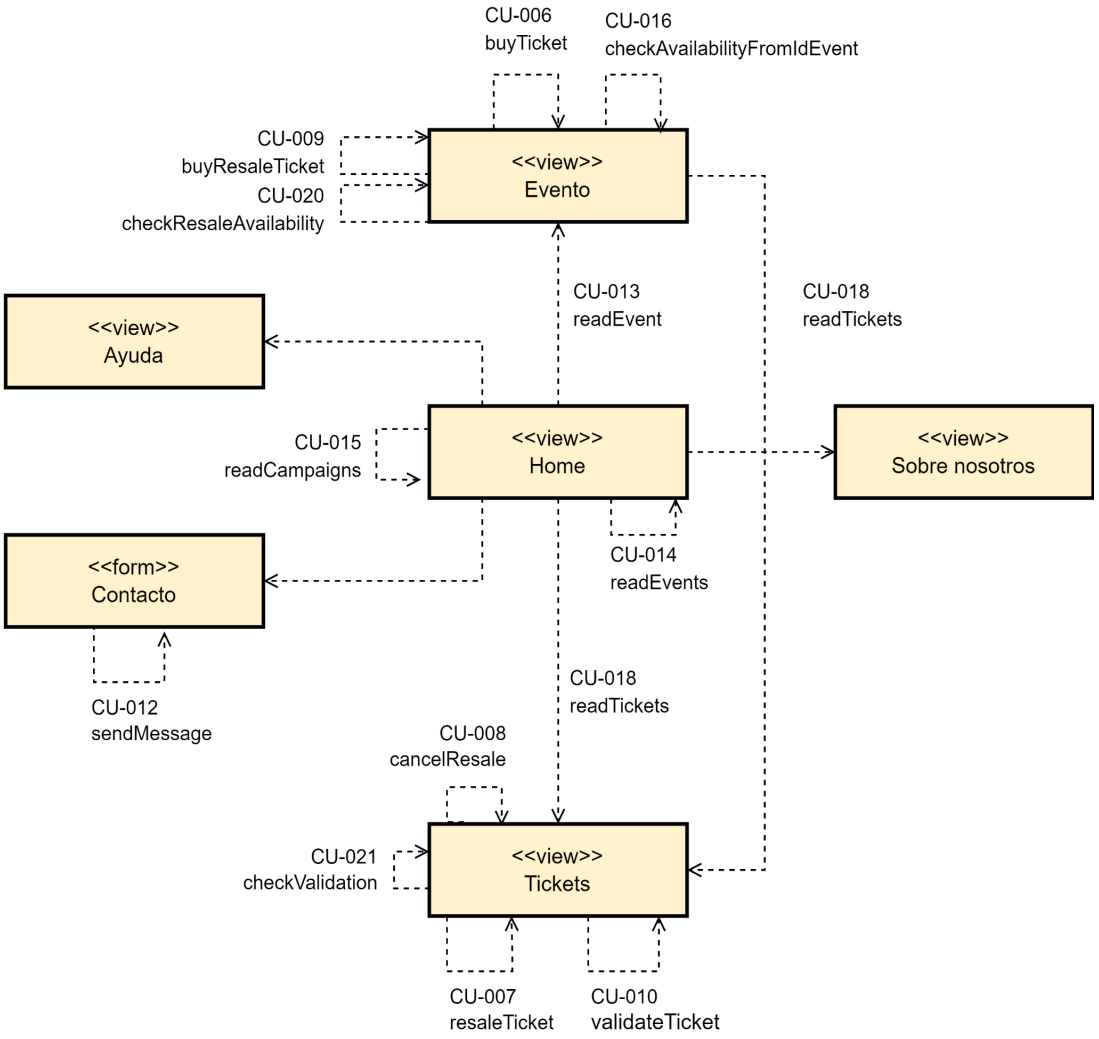


Figura 26: Vistas y responsabilidades de la web principal
Fuente: propia

- **Web backoffice:**

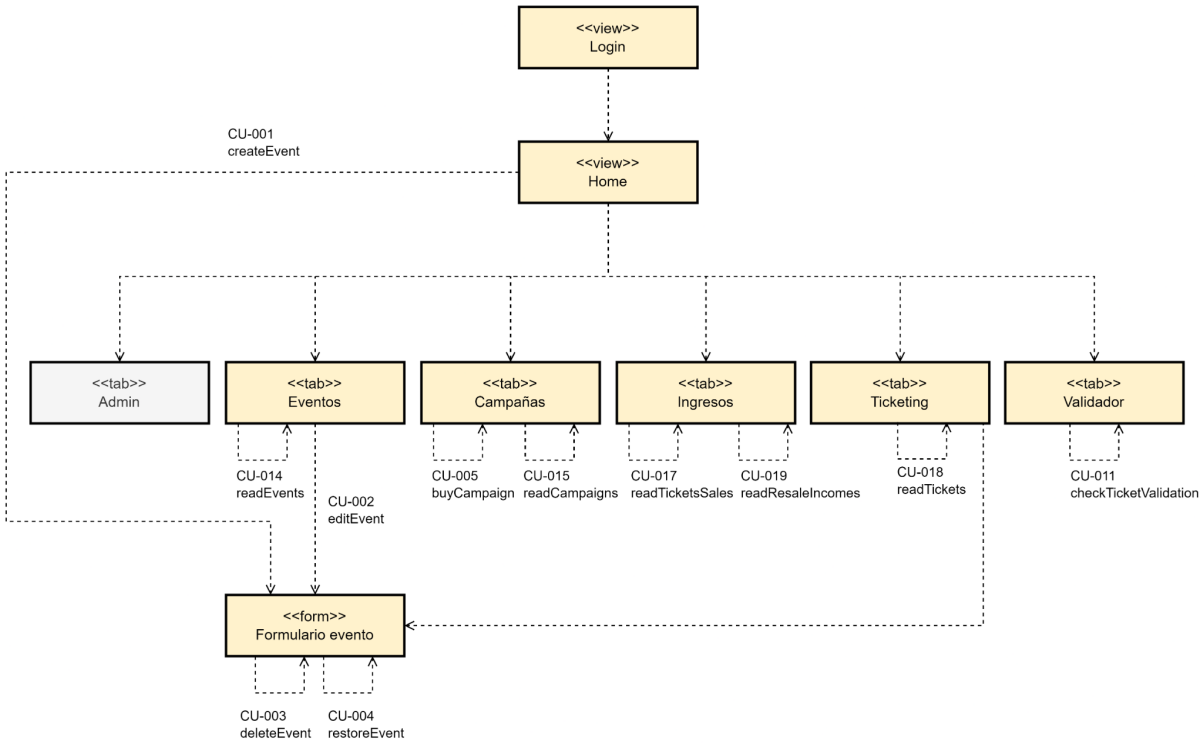


Figura 27: Vistas y responsabilidades de la web backoffice
Fuente: propia

Las dos webs se encuentran online y son accesibles para hacer pruebas (en el apartado 3 del anexo adjuntamos un listado de las pantallas explicando todas las acciones que se pueden hacer):

- **Página principal de venta y reventa de tickets:**
  - **Enlace**: https://tickb.it
  - **Código**: https://github.com/tickbit-dev/tickbit-web

- **Página de gestión backoffice:**
  - **Enlace**: https://business.tickb.it
  - **Código**: https://github.com/tickbit-dev/tickbit-backoffice

CONCLUSIONES

1. CUMPLIMIENTO DE LOS OBJETIVOS

Hemos implementado todas las funcionalidades y cumplido los objetivos que nos marcamos en el inicio de forma satisfactoria. Estas son las funcionalidades implementadas:

- **Contratos en la blockchain**: Los contratos están implementados y cargados en la red de pruebas Mumbai de Polygon.
- **Una web para la gestión:** La web para la gestión está online en https://business.tickb.it.
- **Una web principal de venta de tickets:** La web principal está online en https://tickb.it.
- **Un sistema de validación de entradas:** Hemos implementado un sistema de validación de los tickets registrados en la blockchain que está operativo desde la web de gestión.
- **Un sistema de reventa de entradas regulado**: La web principal permite a los usuarios poner a la venta tickets comprados y recuperar el 90% del precio original.
- **Un sistema de promoción de eventos:** en la web de gestión tenemos opción a promocionar eventos con campañas que dan visibilidad al evento en la página principal.
- **Un sistema de control de ventas e ingresos**: desde la web de gestión los promotores de los eventos tienen acceso a una tabla y gráficos de ventas y de ingresos.

Y también hemos cumplido las normas marcadas así como características que tenía que cumplir el proyecto:

- **Estamos diseñando un prototipo:** Cumplimos en definir límites en la especificación del trabajo y nuestro foco fué la implementación de la tecnología blockchain, y el desarrollo de la funcionalidad de venta-compra-validación de tickets de forma segura.
- **Creación de webs de desarrollo propio (sin plantillas)**: Pensamos un diseño propio y lo implementamos con React.js y ChakraUI.
- **Web online de libre acceso:** Las webs están online en https://tickb.it y https://business.tickb.it, así como el contrato está cargado en la red pública de pruebas de Polygon. Sólo hace falta un navegador para probarlas.

- **Crear una imagen corporativa básica**: Creamos un nombre y un logo.

- **Las webs tienen que tener explicaciones**: Hemos puesto especial esfuerzo en hacer una web intuitiva, que en todo momento sepas qué estás haciendo y hemos creado una sección de ayuda que responde alguna de las preguntas más comunes.

- **Una fuente de ingresos:** Hemos creado más de una fuente de ingresos que pueden sostener el proyecto.

2. CONCLUSIONES GENERALES

Una vez testeado el prototipo podemos llegar a las siguientes conclusiones:

- **La blockchain SÍ aporta valor al mercado de los tickets**

  El prototipo es funcional y mejora en muchos aspectos muy importantes de las plataformas tradicionales de venta de tickets. Además, lo hace de una forma mucho más ética tanto con el creador del evento como con los clientes.

  Los puntos más destacados en los que el prototipo Tickbit, mejora el mercado actual de los tickets son:

  - **Imposibilidad de falsificación de entradas**: La posibilidad de asegurar la propiedad del ticket gracias a los NFTs y a una estructura de datos y transacciones públicas que solo gestiona nuestra plataforma hace que falsificar un ticket sea imposible sin tener la propiedad de la cartera. Hemos podido comprobar que Tickbit realiza este proceso con total éxito.

  - **Imposibilidad de duplicación/robo de entradas**: A diferencia de los sistemas actuales, nadie puede duplicar una entrada con solo una fotografía. Podemos garantizar que cada entrada es única y está registrada públicamente.

  - **Posibilidad de una venta controlada de la entrada**: Si el usuario no pudiera o finalmente no quisiera asistir, tiene la posibilidad de revender el ticket y poder recuperar el 90% del precio original de la entrada.

  - **Fin a la especulación**: Al no poder usar entradas que hayan sido revendidas fuera de la plataforma, la única opción es la venta controlada desde la propia web, de la que nadie puede sacar ningún beneficio especulando.

Sin embargo, Tickbit hereda algunos problemas al usar la tecnología blockchain. Estos problemas son:

- **Tiempo de las transacciones**: El tiempo de las transacciones en condiciones normales, en la red de pruebas de Polygon es de entre 5 y 10 segundos por

transacción y en la red principal es entre 1 y 3 segundos aproximadamente. Sin embargo hay momentos en los que la red está sobrecargada y los tiempos de espera pueden exceder los 20 y 30 segundos en la red principal. Esto podría suponer un problema en algunos procesos como la validación del ticket a la entrada del evento ya que validar 10.000 personas en un evento grande a 30 segundos por persona no es viable.

- **Impacto medioambiental**: La blockchain es una tecnología con un alto impacto medioambiental dado el volumen de energía que consume para validar todas las transacciones. Es una tarea pendiente de esta tecnología, que poco a poco está mejorando y que en un futuro próximo tiene solución, pero todavía no está implementada.

- **Falta de una estandarización**: La tecnología blockchain es una tecnología todavía muy enfocada a un perfil de usuario más técnico, ya que no hay todavía una estandarización que facilite los procesos para personas sin conocimientos sobre la tecnología. Esto dificulta la adopción de usuarios, ya que en un proyecto como una plataforma de venta de tickets el público objetivo no es exclusivamente de perfil técnico.

Sin embargo, estos problemas, como hemos dicho anteriormente, son adoptados de la tecnología blockchain. La tecnología es muy novedosa, está en plena expansión y va mejorando muchos aspectos de forma muy rápida. Es probable que pronto, algunos de estos problemas o no existan, o existan en mucha menor medida.

3. CONCLUSIONES PERSONALES

Iniciamos el proyecto de final de grado con el objetivo personal de aprender algo nuevo que nos motivara y que no hayamos trabajado durante la carrera. Escogimos realizar un proyecto con tecnología blockchain porque es una tecnología con un mercado muy interesante que todavía está por explotar y es un sector en auge y con oportunidades dónde emprender así como una buena opción laboral. Entonces surgió la idea de Tickbit y nos pareció un proyecto perfecto, así que decidimos presentarlo.

Realizar el proyecto fué complicado al inicio, por que no teníamos ningún tipo de base sobre desarrollo en blockchain y no fué una tarea sencilla entender las bases. Sin embargo, la experiencia de trabajar juntos durante toda la carrera hizo que el trabajo en equipo saque lo mejor de nosotros.

Estamos muy satisfechos con el resultado, no esperábamos llegar a implementar funcionalidades hasta el punto que lo hemos logrado, y aunque ha sido duro por momentos, el proceso ha sido gratificante conforme íbamos cumpliento los objetivos.

4. FUTURAS MEJORAS

Al tratarse de un prototipo, el proyecto tiene muchas opciones de mejora. Aquí detallamos algunas posibles mejoras:

- **Más tipos de entrada:** Añadir tipos de entradas numeradas para los eventos así como un selector de asiento, en definitiva, una arquitectura de los eventos más completa con más opciones y flexibilidad para tipos de eventos.

- **Más opciones de búsqueda y filtrado de eventos según gustos:** Nuevos métodos para que el cliente pueda encontrar fácilmente eventos afines a él. Analizar los gustos del cliente sería clave para una mayor probabilidad de compra.

- **Crear coleccionables con los tickets:** Los tickets NFT son propiedad del usuario que va al evento y se le puede asignar un diseño digitalizado para convertirlos en objetos coleccionables que le añadan algún valor.

- **Crear un token propio**: Tener un token propio como divisa, podría dar acceso a descuentos y exclusividades en los eventos de la plataforma para los clientes y una forma de capitalizar la empresa.

- **Campañas de promoción mediante subasta**: Para mejorar la monetización de la plataforma, podría ser interesante implementar un sistema de subastas para la compra de campañas de promoción que ofrecemos para dar visibilidad a los eventos en la web.

AGRADECIMIENTOS

Queremos agradecer en primer lugar a Ariadna Llorens por aceptar nuestra propuesta de trabajo y dejarnos libertad creativa para realizar el proyecto que teníamos en mente y tanto nos motivaba y también por aconsejarnos en decisiones importantes para realizar un trabajo más completo.

En segundo lugar, nos gustaría agradecer a Alberto Bozal Chaves, ingeniero de telecomunicaciones titulado en la UPC de Barcelona, y un buen amigo, por aconsejarnos en los primeros pasos del proyecto y hacernos entender mejor el desarrollo en la blockchain y sus beneficios reales.

Por último, agradecer a amigos, familiares y parejas que tanto nos han apoyado en este largo proceso.

BIBLIOGRAFÍA

[1] https://www.ibm.com/es-es/topics/what-is-blockchain
[2] https://www.hyperledger.org/
[3] https://bitcoin.org/es/
[4] https://ethereum.org/es/
[5] https://polygon.technology/
[6] https://www.corda.net/
[7] https://ethereum.org/es/developers/docs/accounts/
[8] https://ethereum.org/es/developers/docs/transactions/
[9] https://gwei.io/es
[10] https://ethereum.org/es/developers/docs/blocks/
[11] https://ethereum.org/es/developers/docs/standards/tokens/#top
[12] https://ethereum.org/es/developers/docs/evm/
[13] https://solidity-es.readthedocs.io/es/latest/
[14] https://es.wikipedia.org/wiki/%C3%81rbol_de_Merkle
[15] https://trello.com/
[16] https://es.reactjs.org/
[17] https://chakra-ui.com/
[18] https://docs.ethers.io/v5/
[19] https://www.jsonrpc.org/
[20] https://github.com/
[21] https://www.npmjs.com/
[22] https://yarnpkg.com/
[23] https://www.binance.com/es
[24] https://www.dondominio.com/
[25] https://firebase.google.com/es
[26] https://www.netlify.com/
[27] https://chain.link/
[28] https://mumbaifaucet.com/
[29] https://mumbai.polygonscan.com/
[30] https://metamask.io/
[31] https://www.alchemy.com/
[32] https://hardhat.org/

ANEXO

1. CONTRATOS

1.1. TICKBIT.SOL

El contrato Tickbit.sol es el contrato encargado de la creación de campañas y eventos, además de las modificaciones, eliminaciones o restauraciones de estos últimos.

1.1.1. VERSIÓN Y IMPORTS

Lo primero de todo que hay que definir en el Smart Contract es la versión que se va a utilizar, en este caso y como ya hemos mencionado anteriormente utilizamos el rango de versiones entre el 0.7.0 y 0.8.x.

Además de todos los import necesarios para poder utilizar ciertas funciones y propiedades dentro del contrato.

```
pragma solidity >=0.7.0 <0.9.0;

import "@openzeppelin/contracts/utils/Counters.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
```

Figura 28: Selección de versión y imports y del Smart Contract Tickbit.sol
Fuente: propia

Para desarrollar el contrato utilizamos Open Zeppelin, una librería que nos proporciona un conjunto de herramientas para desarrollar aplicaciones descentralizadas y smart contracts. Entre ellas nosotros utilizamos las siguientes herramientas:

- **Counters.sol:** sirve para utilizar contadores que solo pueden incrementar o decrementar. Es muy útil para la generación de IDs, por ejemplo para identificar los eventos.

- **Ownable.sol:** proporciona funciones básicas de control de autorización, esto simplifica la implementación de "permisos de usuario" en las funciones del contrato.

1.1.2. CONTADORES

```
using Counters for Counters.Counter;
Counters.Counter private _eventsIds;
Counters.Counter private _campaignsIds;
```

Figura 29: Counters del Smart Contract Tickbit.sol
Fuente: propia

En este contrato utilizamos dos contadores distintos, el primero `_eventsIds` va servir para identificar los eventos que vayamos creando con un identificador único. El `_campaignsIds` va a servir para identificar las campañas que creemos.

1.1.3. MAPPING

Los mapping en Solidity son como una tabla hash o diccionario. Estos se utilizan para almacenar los datos en forma de pares clave-valor, una clave puede ser cualquiera de los tipos de datos que acepte Solidity, pero los tipos que hacen referencia a otro objeto no están permitidos, mientras que el valor puede ser de cualquier tipo.

```
//Events
mapping(uint256 => EventItem) private idToEvent;

//Campaigns
mapping(uint256 => CampaignItem) private idToCampaign;
mapping(address => uint256[]) private addressToIdCampaignArray;
mapping(uint256 => uint256[]) private initialDateToIdCampaignArray;
```

Figura 30: Mappings del Smart Contract Tickbit.sol
Fuente: propia

En el contrato utilizamos cuatro mappings, uno para los eventos y tres para las campañas:

- **idToEvent:** contiene todas las propiedades de los eventos creados identificados con un id.
- **idToCampaign:** contiene todas las propiedades de las campañas creadas identificados con un id.
- **addressToIdCampaignArray:** contiene todas las propiedades de las campañas creadas identificadas por la dirección de la billetera que las ha creado.

- **initialDateToIdCampaignArray:** contiene todas las propiedades de las campañas creadas identificadas por el valor de la fecha inicial insertada.

1.1.4. STRUCTS

Las estructuras de datos (Structs) son tipos de objetos definidos por el propio usuario y pueden agrupar múltiples variables. En este contrato tenemos definidos varios structs principalmente para los eventos y las campañas, cada uno de estos structs contienen todos los datos que los definen.

```
//Event data
struct EventItem {
    address _owner;
    uint256 _id;
    uint256 _insertionDate;
    string title;
    uint256 idCity;
    uint256 idVenue;
    uint256 idCategory;
    string description;
    string artist;
    uint256 capacity;
    uint256 price;
    string coverImageUrl;
    uint256 initialSaleDate;
    uint256 initialDate;
    uint256 finalDate;
    bool aproved;
    bool deleted;
}

//Data to create an event
struct CreateEventInfo {
    address _owner;
    string title;
    uint256 idCity;
    uint256 idVenue;
    uint256 idCategory;
    string description;
```

```
        string artist;
        uint256 capacity;
        uint256 price;
        string coverImageUrl;
        uint256 initialSaleDate;
        uint256 initialDate;
        uint256 finalDate;
    }

    //Modified event data
    struct ModifyEventInfo {
        uint256 _id;
        string title;
        uint256 idCity;
        uint256 idVenue;
        uint256 idCategory;
        string description;
        string artist;
        uint256 capacity;
        uint256 price;
        string coverImageUrl;
        uint256 initialSaleDate;
        uint256 initialDate;
        uint256 finalDate;
    }

    //Campaign data
    struct CampaignItem {
        address _owner;
        uint256 _id;
        uint256 idType;
        uint256 eventId;
        uint256 initialDate;
        uint256 finalDate;
        uint256 price;
        uint256 purchaseDate;
    }
```

Figura 31: Structs del Smart Contract Tickbit.sol
Fuente: propia

El struct `EventItem` que define las propiedades de los eventos contiene los siguientes datos:

- **_owner:** Direccion billetera del usuario creador del evento.
- **_id:** Identificador numérico.
- **_insertionDate:** Fecha de creación.
- **title:** Título.
- **idCity:** Identificador numérico de la ciudad donde se realiza.
- **idVenue:** Identificador numérico del recinto donde se realiza.
- **idCategory:** Identificador de la categoría (Música, teatro, deportes…).
- **description:** Descripción.
- **artist:** Nombre del artista.
- **capacity:** Número de entradas a vender.
- **price:** Precio.
- **coverImageUrl:** Url con la imagen del evento.
- **initialSaleDate:** Fecha inicial de puesta a la venta de entradas**.**
- **initialDate:** Fecha de inicio del evento.
- **finalDate:** Fecha final del evento.
- **aproved:** Indica si el evento se ha aprobado o no.
- **deleted:** Indica si el elemento ha sido eliminado o no.

El struct `CampaignItem` que define las propiedades de las campañas y contiene los siguientes datos:

- **_owner:** Direccion billetera del usuario que ha comprado la campaña.
- **_id:** Identificador numérico.
- **idType:** Identificador numérico del tipo (portada o destacados).
- **eventId:** Identificador numérico del evento sobre el cual se realiza la campaña.
- **initialDate:** Fecha de inicio de la campaña.
- **finalDate:** Fecha de final de la campaña.
- **price:** Precio.
- **purchaseDate:** Fecha de compra de la campaña.

Ademas de estos dos structs hay dos structs más de eventos que són auxiliares, el `ModifyEventInfo` para la modificación de los parámetros de los eventos y el `CreateEventInfo` para la creación de eventos.

Este segundo struct auxiliar lo hemos tenido que crear porque como hemos explicado anteriormente la máquina virtual de Ethereum (EVM), trabaja con una pila virtual, o stack, por lo que Solidity limita el número de elementos que se pueden pasar por parámetro, para

evitar que esta pila pueda llenarse. Por lo tanto este struct tiene menos elementos, el resto de elementos que faltan los introducimos directamente en la función de crear eventos.

1.1.5. EVENTOS

Los eventos en Solidity ayudan a informar cuando sucede algo en el contrato inteligente, es decir gracias a los eventos podemos recopilar información en la plataforma web y notificar si una transacción se ha realizado correctamente o ha habido algún error. En este contrato tenemos declarados diferentes eventos que se emitirán cuando se cree, edite, elimine o restaure un evento o una campaña y contendrán parte de su información.

```
event EventItemCreated(
      address indexed _owner,
      uint256 indexed _id,
      uint256 _insertionDate
  );

  event EventItemModified(
      address indexed _owner,
      uint256 indexed _id,
      uint256 _insertionDate
  );

  event EventItemDeleted(
      address indexed _owner,
      uint256 indexed _id,
      uint256 _insertionDate
  );

  event EventItemRestored(
      address indexed _owner,
      uint256 indexed _id,
      uint256 _insertionDate
  );

  event CampaignItemCreated(
      address indexed _owner,
      uint256 indexed _id,
      uint256 idType,
      uint256 eventId,
      uint256 initialDate,
```

```
        uint256 finalDate,
        uint256 _purchaseDate
    );
```

Figura 32: Eventos del Smart Contract Tickbit.sol
Fuente: propia

1.1.6. FUNCIONES

- **[CU-001] createEvent:**

```
function createEvent(CreateEventInfo memory eventInfo) public {
    //increments _eventsIds global counter
    _eventsIds.increment();
    uint256 eventId = _eventsIds.current();

    //Creates a new event and saves it in idToEvent mapping at
    the eventId position
    idToEvent[eventId] = EventItem(
        msg.sender,
        eventId,
        block.timestamp,
        eventInfo.title,
        eventInfo.idCity,
        eventInfo.idVenue,
        eventInfo.idCategory,
        eventInfo.description,
        eventInfo.artist,
        eventInfo.capacity,
        eventInfo.price,
        eventInfo.coverImageUrl,
        eventInfo.initialSaleDate,
        eventInfo.initialDate,
        eventInfo.finalDate,
        false,
        false
    );

    emit EventItemCreated(msg.sender, eventId, block.timestamp);
```

```
}
```

Figura 33: Función createEvent del Smart Contract Tickbit.sol
Fuente: propia

Crea y pública un nuevo evento, recibe por parámetros un struct auxiliar del tipo `CreateEventInfo` , a partir de este struct, crea un nuevo struct `EventItem` y le añade las propiedades que le faltan a la vez que lo guarda en el mapping `idToEvent` con el ID nuevo generado del evento como clave. Una vez hecho esto, se emite el evento `EventItemCreated`.

- **[CU-002] editEvent:**

```
function editEvent(ModifyEventInfo memory eventInfo) public {
     uint256 id = eventInfo._id;

     //Needs to exist an event created with that id
     require(idToEvent[id]._id != 0, "This event does not exist");

     //The sender needs to be the creator of the event or be the
     contract owen
     require(idToEvent[id]._owner == msg.sender || msg.sender ==
     owner(), "Invalid owner");

     //Modifies the actual data with the new data
     idToEvent[id].title = eventInfo.title;
     idToEvent[id].idCity = eventInfo.idCity;
     idToEvent[id].idVenue = eventInfo.idVenue;
     idToEvent[id].idCategory = eventInfo.idCategory;
     idToEvent[id].description = eventInfo.description;
     idToEvent[id].artist = eventInfo.artist;
     idToEvent[id].capacity = eventInfo.capacity;
     idToEvent[id].price = eventInfo.price;
     idToEvent[id].coverImageUrl = eventInfo.coverImageUrl;
     idToEvent[id].initialSaleDate = eventInfo.initialSaleDate;
     idToEvent[id].initialDate = eventInfo.initialDate;
```

```
        idToEvent[id].finalDate = eventInfo.finalDate;

        emit EventItemModified(msg.sender, id, block.timestamp);
    }
```

Figura 34: Función editEvent del Smart Contract Tickbit.sol
Fuente: propia

Modifica un evento ya existente, para modificarlo es necesario ser el propietario del contrato o el propietario del evento, de otra forma no te dejará hacer modificaciones sobre ese evento. Por parámetros se recibe un struct auxiliar `ModifyEventInfo`, una vez recibido se sustituyen todas las propiedades del evento ya existente en el mapping `idToEvent` identificado con el ID que coincide con el del struct auxiliar `ModifyEventInfo`. Una vez modificado se emite el evento `EventItemModified`.

- **[CU-003] deleteEvent:**

```
function deleteEvent(uint256 id) public {
        //Needs to exist an event created with that id
        require(idToEvent[id]._id != 0, "This event does not exist");

        //The sender needs to be the creator of the event or be the
        contract owner
        require(idToEvent[id]._owner == msg.sender || msg.sender ==
        owner(), "Invalid owner");

        idToEvent[id].deleted = true;
        emit EventItemDeleted(msg.sender, id, block.timestamp);
    }
```

Figura 35: Función deleteEvent del Smart Contract Tickbit.sol
Fuente: propia

Elimina un evento ya existente, para poder eliminarlo es necesario ser el propietario del contrato o el propietario del evento, de otra forma no te dejará eliminarlo. Se recibe por parámetros un ID numérico y se modifica la variable deleted a true del evento dentro del mapping `idToEvent` cuyo ID coincida con el pasado por parámetros. Una vez eliminado se emite el evento `EventItemRestored`.

- **[CU-004] restoreEvent:**

```
function restoreEvent(uint256 id) public {
    //Needs to exist an event created with that id
    require(idToEvent[id]._id != 0, "This event does not exist");

    //The sender needs to be the creator of the event or be the
    contract owner
    require(idToEvent[id]._owner == msg.sender || msg.sender ==
    owner(), "Invalid owner");

    idToEvent[id].deleted = false;
    emit EventItemRestored(msg.sender, id, block.timestamp);
}

}
```

Figura 36: Función restoreEvent del Smart Contract Tickbit.sol
Fuente: propia

Restaura un elemento que había sido eliminado previamente, para restaurarlo es necesario ser el propietario del contrato o el propietario del evento, de otra forma no se podrá restaurar. Similar a la función `deleteEvent`, se recibe por parámetros un ID numérico y se modifica la variable deleted a false del evento dentro del mapping `idToEvent` cuyo ID coincida con el pasado por parámetros. Una vez restaurado se emite el evento `EventItemRestored`.

- **[CU-013] readEvent:**

```
function readEvent(uint256 id, bool isPublicRead) public view
returns (EventItem memory) {
    //Needs to exist an event created with that id
    require(idToEvent[id]._id != 0, "This event does not exist");

    if(isPublicRead == false){
    //The sender needs to be the creator of the event or be the
    contract owner
        require(idToEvent[id]._owner == msg.sender || msg.sender
        == owner(), "Invalid owner");
    }

    return idToEvent[id];
}
```

Figura 37: Función readEvent del Smart Contract Tickbit.sol
Fuente: propia

Nos permite obtener la información de un evento en concreto ya existente devolviendo un objeto `EventItem`. La función recibe por parámetros el ID del evento a leer y un booleano. El booleano determinará si el que puede leer el evento es el propietario del contrato o del evento o cualquier persona. Una vez comprobado quien la puede leer devolverá un struct `EventItem` cuyo ID coincida con el ID pasado por parámetros.

- **[CU-014] readEvents**:

```
 function readEvents(bool isPublicRead) public view returns
(EventItem[] memory) {
        uint256 eventCount = _eventsIds.current();
        uint256 myEventCount = 0;
        uint256 currentIndex = 0;

        //Contract owner gets a counter of all the existing events
        //User gets a counters of all the events created by him
        for (uint256 i = 0; i < eventCount; i++) {
            //Checks the owner of the event
            if(idToEvent[i + 1]._owner == msg.sender || msg.sender ==
            owner() || isPublicRead == true) {
                //Checks if the event is deleted or if the owner is
                the contract owner
                if(idToEvent[i + 1].deleted == false || msg.sender ==
                owner()) {
                    myEventCount += 1;
                }
            }
        }

        EventItem[] memory events = new EventItem[](myEventCount);

        //Contract owner gets an array with all the events
        //User gets an array with the events created by him
        for (uint256 i = 0; i < eventCount; i++) {
            //Checks the owner of the event
            if(idToEvent[i + 1]._owner == msg.sender || msg.sender ==
            owner() || isPublicRead == true) {
                //Checks if the event is deleted or if the owner is
                the contract owner
                if(idToEvent[i + 1].deleted == false || msg.sender ==
                owner()) {
                    //Add event to the array
                    uint256 currentId = i + 1;
                    EventItem storage currentEvent =
                    idToEvent[currentId];
                    events[currentIndex] = currentEvent;
                    currentIndex += 1;
```

```
                }
            }
        }

        return events;
    }
```

Figura 38: Función readEvents del Smart Contract Tickbit.sol
Fuente: propia

Permite leer un conjunto de eventos previamente creados. Recibe por parámetros un booleano que nos indicará si se puede hacer una lectura de todos los eventos o no. Esta función está diseñada para que se puedan leer todos los eventos que existen o solo los eventos que ha creado un usuario. Primero de todo se comprueba el número de eventos que se van a leer para poder crear un array con un tamaño fijo, una vez hecho esto se procede a leer todos los eventos o solo los eventos creados por el usuario que hace la petición y se van guardando en un array de structs de `EventItem` que es lo que devolverá la función.

- **[CU-005] createCampaign:**

```
function createCampaign(
    uint256 idType,
    uint256 eventId,
    uint256 initialDate,
    uint256 finalDate,
    uint256 price
) public payable {
    //The price needs to be equal to the amount the sender address
    sends.
    require(msg.value == price, "wrong amount sent");
    //Transfers the money to the contract owner address
    payable(owner()).transfer(msg.value);
    //Increments _campaignsIds global counter
    _campaignsIds.increment();
    uint256 campaignId = _campaignsIds.current();
```

```
        //Creates a new campaign and saves it in idToCampaign mapping
        at the campaignId position
        idToCampaign[campaignId] = CampaignItem(
            msg.sender,
            campaignId,
            idType,
            eventId,
            initialDate,
            finalDate,
            price,
            block.timestamp
        );

        //Adds the campaignId to the addressToIdCampaignArray mapping
        at the msg.sender position.
        addressToIdCampaignArray[msg.sender].push(campaignId);
        initialDateToIdCampaignArray[initialDate].push(campaignId);

        emit CampaignItemCreated(
            msg.sender,
            campaignId,
            idType,
            eventId,
            initialDate,
            finalDate,
            block.timestamp
        );
    }
```

Figura 39: Función createCampaign del Smart Contract Tickbit.sol
Fuente: propia

Permite la compra y creación de una campaña sobre un evento existente. Recibe por parámetros todos los valores necesarios para poder crear un struct `CampaignItem`. Primero se comprueba si el dinero pagado corresponde con el precio, si el dinero enviado es correcto, este se transfiere a la billetera de Tickbit que es la creadora del contrato. Una vez enviado se añade al mapping `idToCampaign` un struct `CampaignItem` identificado con un ID único, además también se añade en dos mapping más, en el `addressToIdCampaignArray` donde cada campaña estará identificada por la dirección de billetera del creador y en el `initialDateToIdCampaignArray` donde cada campaña estará identificada por la fecha inicial de la campaña. Finalmente se hará un emit del event `CampaignItemCreated`.

- **[CU-015] readCampaigns:**

```
function readCampaigns(bool isPublicRead) public view returns
(CampaignItem[] memory) {
    uint256 campaignsCount = _campaignsIds.current();
    uint256 myCampaignsCount = 0;
    uint256 currentIndex = 0;

    for (uint256 i = 0; i < campaignsCount; i++) {
        uint256 id = idToCampaign[i + 1].eventId;
        //Checks the owner of the event
        if(idToEvent[id]._owner == msg.sender || msg.sender ==
        owner() || isPublicRead == true) {
            myCampaignsCount += 1;
        }
    }

    CampaignItem[] memory campaigns = new
    CampaignItem[](myCampaignsCount);

    //Contract owner gets an array with all the events
    //User gets an array with the events created by him
    for (uint256 i = 0; i < campaignsCount; i++) {
        uint256 id = idToCampaign[i + 1].eventId;
        //Checks the owner of the event
        if(idToEvent[id]._owner == msg.sender || msg.sender ==
         owner() || isPublicRead == true) {
            //Add campaign to the array
            uint256 currentId = i + 1;
            CampaignItem storage currentCampaign =
            idToCampaign[currentId];
            campaigns[currentIndex] = currentCampaign;
            currentIndex += 1;
        }
    }

    return campaigns;
}
```

Figura 40: Función readCampaigns del Smart Contract Tickbit.sol
Fuente: propia

Permite leer un conjunto de campañas previamente creadas, es muy parecida a la función para leer los eventos. Recibe por parámetros un booleano que nos indicará si se puede hacer una lectura de todas las campañas o no. Esta función está diseñada para que se puedan leer todas las campañas que existen o solo las campañas que ha comprado un usuario. Primero de todo se comprueba el número de campañas que se van a leer para poder crear un array con un tamaño fijo, una vez hecho esto se procede a leer todos las campañas o solo las campañas compradas por el usuario que hace la petición y se van guardando en un array de structs de `CampaignItem` que es lo que devolverá la función.

1.2. TICKBITTICKET.SOL

El contrato TickbitTicket.sol es el contrato que administra la emisión, gestión, la compra y la venta de tickets. Los tickets se generan con el estándar ERC721 y se generan con un id propio siendo cada ticket un token no fungible (NFT).

El contrato está deployado en la testnet Mumbai de Polygon.

1.2.1. VERSIÓN Y IMPORTS

La versión que se va a utilizar en este caso y como ya hemos mencionado anteriormente es el rango de versiones entre el 0.7.0 y 0.8.x.

Además de todos los import necesarios para poder utilizar ciertas funciones y propiedades dentro del contrato.

```
pragma solidity >=0.7.0 <0.9.0;

import "@openzeppelin/contracts/utils/Counters.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import
"@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol";
import "./Tickbit.sol";
```

Figura 41: Selección de versión y imports y del Smart Contract TickbitTicket.sol
Fuente: propia

Para desarrollar el contrato utilizamos Open Zeppelin, una librería que nos proporciona un conjunto de herramientas para desarrollar aplicaciones descentralizadas y smart contracts. Entre ellas nosotros utilizamos las siguientes herramientas:

- **Counters.sol:** sirve para utilizar contadores que solo pueden incrementar o decrementar. Es muy útil para la generación de IDs, por ejemplo para identificar los eventos.

- **Ownable.sol:** proporciona funciones básicas de control de autorización, esto simplifica la implementación de "permisos de usuario" en las funciones del contrato.

- **ERC721.sol:** proporciona los métodos necesarios para la generación del Estándar de token no fungible (NFT).

- **AggregatorV3Interface.sol:** proporciona los métodos necesarios para referenciar y utilizar funciones de otro contrato.

- **Tickbit.sol:** instancia al contrato Tickbit.sol.

1.2.2. CONTADORES

```
using Counters for Counters.Counter;
Counters.Counter private _ticketsIds;
Counters.Counter private _resalesIds;
```

Figura 42: Counters del Smart Contract TickbitTicket.sol
Fuente: propia

En este contrato utilizamos dos contadores distintos, el primero `_ticketsIds` va servir para identificar los tickets que vayamos creando con un identificador único. El `_resalesIds` va a servir para identificar las reventas que se hagan en la plataforma.

1.2.3. MAPPING

Los mapping en Solidity son como una tabla hash o diccionario. Estos se utilizan para almacenar los datos en forma de pares clave-valor, una clave puede ser cualquiera de los tipos de datos que acepte Solidity, pero los tipos que hacen referencia a otro objeto no están permitidos, mientras que el valor puede ser de cualquier tipo.

```
  //Ticketing
    mapping(uint256 => TicketItem) private idToTicket;
    mapping(uint256 => uint256[]) private idEventToIdTicketArray;
    mapping(uint256 => TicketValidation) private
idTicketToTicketValidation;
    mapping(uint256 => TicketResale) private idToTicketResale;
```

Figura 43: Mappings del Smart Contract TickbitTicket.sol
Fuente: propia

En el contrato utilizamos cuatro mappings, uno para los eventos y tres para las campañas:

- **idToTicket:** contiene todas las propiedades de los tickets creados identificados con un id.

- **idEventToIdTicketArray:** contiene todos los id de tickets que pertenecen a cada id de evento.

- **idTicketToTicketValidation:** contiene todas las propiedades de las validaciones de tickets que pertenecen a un id de ticket concreto.

- **idToTicketResale:** contiene todas las propiedades de los tickets revendidos identificados con un id.

1.2.4. STRUCTS

Las estructuras de datos (Structs) son tipos de objetos definidos por el propio usuario y pueden agrupar múltiples variables. En este contrato tenemos definidos varios structs principalmente para los tickets y las validaciones, cada uno de estos structs contienen todos los datos que los definen.

```
//Ticket data
struct TicketItem {
    address _owner;
    address _eventOwner;
    uint256 _id;
    uint256 _purchaseDate;
    uint256 idEvent;
    uint256 price;
    bool validated;
    bool isOnSale;
}

//Ticket data
struct TicketValidation {
    uint256 _validationDate;
    uint256 idTicket;
    uint256 idEvent;
    uint256 validationHash;
}

//Ticket data
struct TicketResale {
    uint256 _id;
    uint256 _resaleDate;
    uint256 idTicket;
    uint256 idEvent;
    bool isSold;
    bool isCancelled;
}
```

Figura 44: Structs del Smart Contract TickbitTicket.sol
Fuente: propia

El struct `TicketItem` que define las propiedades de los tickets contiene los siguientes datos:

- **_owner:** Dirección de billetera del usuario dueño del ticket.
- **_eventOwner**: Dirección de billetera del usuario dueño del evento.
- **_id:** Identificador numérico.
- **_purchaseDate:** Fecha de creación.
- **idEvent:** Identificador del evento al cual pertenece el ticket.
- **price:** precio del evento en dólares.
- **validated:** booleano que indica si el ticket ha sido validado.
- **isOnSale:** booleano que indica si el ticket está a la venta en reventa.

El struct `TicketValidation` define las propiedades de los tickets que se vayan a validar y contiene los siguientes datos:

- **_validationDate:** Fecha de la validación.
- **idTicket**: Identificador del ticket al que pertenece la validación.
- **idEvent:** Identificador del evento al cual pertenece el ticket.
- **validationHash:** Hash único de verificación de la validación.

El struct `TicketResale` define las propiedades de los tickets que se vayan a revender y contiene los siguientes datos:

- **_id:** Identificador numérico.
- **_resaleDate:** Fecha de puesta en venta.
- **idTicket**: Identificador del ticket al que pertenece la reventa.
- **idEvent:** Identificador del evento al cual pertenece el ticket.
- **isSold:** booleano que indica si la reventa ha sido realizada.
- **isCancelled:** booleano que indica si la reventa ha sido cancelada.

1.2.5. EVENTOS

Los eventos en Solidity ayudan a informar cuando sucede algo en el contrato inteligente, es decir gracias a los eventos podemos recopilar información en la plataforma web y notificar si una transacción se ha realizado correctamente o ha habido algún error. En este contrato tenemos declarados diferentes eventos que se emitirán cuando se cree o valide un ticket.

```
event TicketItemCreated(
    address indexed _owner,
    address indexed _eventOwner,
    uint256 indexed _id,
    uint256 _purchaseDate,
    uint256 idEvent
);

event TicketItemValidation(
    uint256 _validationDate,
    uint256 idTicket,
    uint256 idEvent,
    uint256 validationHash
);

event TicketItemValidated(
    bool validated
);
```

Figura 45: Eventos del Smart Contract TickbitTicket.sol
Fuente: propia

1.2.6. CONSTRUCTOR

El constructor define acciones que se ejecutarán en la primera instancia del contrato, en su creación.

```
    constructor() ERC721("Tickbit Tickets", "TCKB") {
        chainlinkMaticUSD =

AggregatorV3Interface(0xd0D5e3DB44DE05E9F294BB0a3bEEaF030DE24Ada);
        tickbitContract =
          Tickbit(0xEa24F063385487F264b7E17c57cf3512fc5191a1);
    }
```

Figura 46: Constructor del Smart Contract TickbitTicket.sol
Fuente: propia

En este constructor podemos ver que lo primero que definimos es la identificación del token ERC721 con el nombre “Tickbit Ticket” y la abreviatura identificativa “TCKB”.

Después creamos una instancia con el `AggregatorV3Interface` a la dirección del contrato del oráculo de Chainlink que corresponde a la fuente de datos MATIC / USD en la red Polygon Mumbai, que es la que vamos a necesitar para hacer la conversión de dólares a MATIC.

También creamos una instancia de nuestro otro contrato Tickbit.sol para poder ejecutar funciones de este contrato dentro de TickbitTicket.sol.

1.2.7. FUNCIONES

- **[CU-006] buyTicket:**

```
function buyTicket(uint256 idEvent, uint256 noOfTickets) public
payable {
      //Checks if the noOfTickets is a valid number
      require(noOfTickets <= 5, "You can't buy more than 5 tickets
      in one order");

      Tickbit.EventItem memory eventItem =
      tickbitContract.readEvent(idEvent, true);

      //The price needs to be equal to the amount the sender
      address sends.
```

```
require(msg.value <= ((getMaticWeiFromUSD(eventItem.price) *
noOfTickets) + getMaticWeiFromUSD(1)), "Wrong amount sent");
require(msg.value >= ((getMaticWeiFromUSD(eventItem.price) *
noOfTickets) - getMaticWeiFromUSD(1)), "Wrong amount sent");

//Checks if the event has availability
require(eventItem.capacity >=
(idEventToIdTicketArray[eventItem._id].length + noOfTickets),
 "No availability for this event");

//Transfers the money to the event owner address
uint256 sellerPercent = (msg.value * 99) / 100;
payable(eventItem._owner).transfer(sellerPercent);

//Transfers the fee to the contract owner address
payable(owner()).transfer(msg.value - sellerPercent);

for(uint256 i = 0; i < noOfTickets; i++) {
    //increments _ticketsIds global counter
    _ticketsIds.increment();
    uint256 ticketId = _ticketsIds.current();

    idToTicket[ticketId] = TicketItem(
        msg.sender,
        eventItem._owner,
        ticketId,
        block.timestamp,
        idEvent,
        eventItem.price,
        false,
        false
    );

    _mint(msg.sender, ticketId);

    //Adds the ticketId in the idEventToIdTicketArray mapping
    at idEventt positon
    idEventToIdTicketArray[idEvent].push(ticketId);

    emit TicketItemCreated(msg.sender, eventItem._owner,
    ticketId, block.timestamp, idEvent);
```

```
        }
    }
```

Figura 47: Función buyTicket del Smart Contract TickbitTicket.sol
Fuente: propia

Genera y el usuario paga y recibe un nuevo ticket para un evento concreto.

Recibe por parámetros un entero que es el identificador del evento y el número de tickets que se van a comprar. Además, esta función está marcada cómo `payable` por lo que hay que enviar también el precio a pagar en la transacción.

Con el primer require, comprobamos que el número de tickets no sea mayor a 5. Seguidamente comprobamos que el valor pasado de la compra a la función no sobrepase el ±1 dólar. Por último también hacemos un require para comprobar que el evento tiene disponibilidad.

También tenemos que tener en cuenta los porcentajes del dinero que se lleva cada parte. Hacemos el cálculo del 95% del precio que se transferirá al creador del evento con la función `payable` y el 5% restante lo transferiremos a la cartera administradora, es decir a nosotros como dueños del contrato.

Para finalizar mintearemos el token ERC721 con el comando `_mint` y el id del ticket generado y registramos el ticket con su id en el mapping `idToTicket` además de llamar al `emit TicketItemCreated`.

- **[CU-009] buyResaleTicket:**

```
function buyResaleTicket(uint256 idEvent, uint256 noOfTickets)
public payable {
    //Checks if the noOfTickets is a valid number
    require(noOfTickets <= 5, "You can't buy more than 5 tickets
    in one order");

    Tickbit.EventItem memory eventItem =
    tickbitContract.readEvent(idEvent, true);
    uint256[] memory resaleIdItems = getResalesForEvent(idEvent);

    //The price needs to be equal to the amount the sender
    address sends.
```

```
require(msg.value <= ((getMaticWeiFromUSD(eventItem.price) *
noOfTickets) + getMaticWeiFromUSD(1)), "Wrong amount sent");
require(msg.value >= ((getMaticWeiFromUSD(eventItem.price) *
noOfTickets) - getMaticWeiFromUSD(1)), "Wrong amount sent");

//Checks if the event has no availability
require(eventItem.capacity ==
idEventToIdTicketArray[eventItem._id].length, "Event already
available");

//Checks if the event has resell availability
require(resaleIdItems.length >= noOfTickets, "No availability
for this event");

for(uint256 i = 0; i < noOfTickets; i++) {
    //Último ticket
    uint256 resaleIdItem = resaleIdItems[i];
    TicketItem memory ticketItem =
    idToTicket[idToTicketResale[resaleIdItem].idTicket];

    //Transfers the money to the event owner address
    uint256 sellerPercent = ((msg.value / noOfTickets) * 90)
    / 100;
    payable(ticketItem._owner).transfer(sellerPercent);

    //Transfers 9% fee to the event owner address
    uint256 eventOwnerPercent = ((msg.value / noOfTickets) *
    9) / 100;
    payable(eventItem._owner).transfer(eventOwnerPercent);

    //Transfers the 1% fee to the tickbit owner address
    payable(owner()).transfer((msg.value / noOfTickets) -
    sellerPercent - eventOwnerPercent);

    //Transfers the ticket token to the new owner
    _transfer(ticketItem._owner, msg.sender, ticketItem._id);

    //Change the parameters
   idToTicket[idToTicketResale[resaleIdItem].idTicket]._owner
   = msg.sender;
```

```
idToTicket[idToTicketResale[resaleIdItem].idTicket].isOnSale = false;
            idToTicketResale[resaleIdItem].isSold = true;
        }
    }
```

Figura 48: Función buyResaleTicket del Smart Contract TickbitTicket.sol
Fuente: propia

El usuario paga y recibe un nuevo ticket para un evento concreto y el vendedor recibe la parte que le corresponde de la venta.

Recibe por parámetros un entero que es el identificador del evento y el número de tickets que se van a comprar. Además, esta función está marcada cómo `payable` por lo que hay que enviar también el precio a pagar en la transacción.

Con el primer require, comprobamos que el número de tickets no sea mayor a 5. Seguidamente comprobamos que el valor pasado de la compra a la función no sobrepase el ±1 dólar. También hacemos un require para comprobar que no hay disponibilidad para este evento y por lo tanto la reventa es la única opción, y para acabar comprobamos que hay unidades suficientes en reventa para cubrir el número de tickets de la petición.

También tenemos que tener en cuenta los porcentajes del dinero que se lleva cada parte. Hacemos el cálculo del 90% del precio que se transferirá al vendedor del ticket con la función `payable`, el 9% lo transferiremos al dueño del evento, y el 1% restante al dueño del contrato, es decir a nosotros como dueños del contrato.

Para finalizar traspasamos el token ERC721 del ticket del vendedor al comprador con el comando `_transfer` y el id del ticket y actualizamos el ticket con su id en el mapping `idToTicket` además de añadirlo también al `idToTicketResale`.

- **getResalesForEvent:**

```
function getResalesForEvent(uint256 idEvent) public view returns
   (uint256[] memory) {
        uint256 myResalesCount = 0;

        for(uint256 i = 0; i < _resalesIds.current(); i++) {
            if(idToTicketResale[i + 1].isSold == false &&
            idToTicketResale[i + 1].idEvent == idEvent &&
            idToTicketResale[i + 1].isCancelled == false){
                myResalesCount += 1;
            }
        }

        uint256[] memory idResalesItems = new
        uint256[](myResalesCount);
        uint256 currentCount = 0;

        for(uint256 i = 0; i < _resalesIds.current(); i++) {
            if(idToTicketResale[i + 1].isSold == false &&
            idToTicketResale[i + 1].idEvent == idEvent &&
            idToTicketResale[i + 1].isCancelled == false){
                idResalesItems[currentCount] =
                idToTicketResale[i + 1]._id;
                currentCount += 1;
            }
        }

        return idResalesItems;
    }
```

Figura 49: Función getResalesForEvent del Smart Contract TickbitTicket.sol
Fuente: propia

Retorna los identificadores de los tickets que están en reventa.

Es necesario pasar el identificador del evento como parámetro y el proceso se basa en recorrer el `idToTicket` y el `idResalesItems` para obtener los datos que queremos.

- **[CU-019] getResalesIncomes:**

```
function getResalesIncomes() public view returns (TicketResale[]
memory) {
        uint256 myResalesCount = 0;

        for(uint256 i = 0; i < _resalesIds.current(); i++) {
            Tickbit.EventItem memory eventItem =
            tickbitContract.readEvent(idToTicketResale[i + 1]
            .idEvent, true);

            if(idToTicketResale[i + 1].isSold == true && (msg.sender
               == owner() || eventItem._owner == msg.sender)){
                myResalesCount += 1;
            }
        }

        TicketResale[] memory resalesItems =
        new TicketResale[](myResalesCount);
        uint256 currentCount = 0;

        for(uint256 i = 0; i < _resalesIds.current(); i++) {
            Tickbit.EventItem memory eventItem =
            tickbitContract.readEvent(idToTicketResale[i + 1]
            .idEvent, true);

            if(idToTicketResale[i + 1].isSold == true && (msg.sender
               == owner() || eventItem._owner == msg.sender)){
                resalesItems[currentCount] = idToTicketResale[i + 1];
                currentCount += 1;
            }
        }

        return resalesItems;
}
```

Figura 50: Función getResalesIncomes del Smart Contract TickbitTicket.sol
Fuente: propia

Retorna los ingresos de los tickets que han sido revendidos reventa. El proceso se basa en recorrer el `idToTicketResale` y obtener un array del struct `TicketResale` que queremos.

- **[CU-020] checkResaleAvailability:**

```
function checkResaleAvailability(uint256 idEvent) public view returns
(uint) {
        return getResalesForEvent(idEvent).length;
}
```

Figura 51: Función checkResaleAvailability del Smart Contract TickbitTicket.sol
Fuente: propia

Función auxiliar de `getResalesForEvent` para obtener la disponibilidad de reventa.

- **[CU-007] resaleTicket:**

```
function resaleTicket(uint256 idTicket) public {
        //Checks the ownership of the ticket
        require(idToTicket[idTicket]._owner == msg.sender, "You are
        not the ticket owner");

        //Checks if the ticket is already validated
        require(idToTicket[idTicket].validated == false, "Ticket
        already validated");

        //Checks if the ticket is already validated
        require(idToTicket[idTicket].isOnSale == false, "Ticket
        already on sale");

        //Increments _resalesIds global counter
        _resalesIds.increment();
        uint256 resaleId = _resalesIds.current();

        //Creates a new resale and saves it in idToResale mapping
```

```
        idToTicketResale[resaleId] = TicketResale(
            resaleId,
            block.timestamp,
            idTicket,
            idToTicket[idTicket].idEvent,
            false,
            false
        );

        //Change parameters
        idToTicket[idTicket].isOnSale = true;
}
```

Figura 52: Función resaleTicket del Smart Contract TickbitTicket.sol
Fuente: propia

Pone un ticket disponible para reventa.

Recibe por parámetro el identificador del ticket sobre el que vamos a cambiar el estado de reventa.

Con el primer require, comprobamos si el usuario que hace la petición es el dueño del ticket. Con el segundo require comprobamos si el ticket ha sido previamente validado y con el tercero si ya está puesto a la venta.

Si cumple los requires pondremos en el mapping `idToTicketResale` un nuevo struct del tipo `TicketResale` para que el ticket salga a la venta.

Por último, marcamos el ticket como puesto en venta usando el mapping `idToTicket`.

- **[CU-008] cancelResale:**

```
function cancelResale(uint256 idTicket) public {
        //Checks the ownership of the ticket
        require(idToTicket[idTicket]._owner == msg.sender, "You are
        not the ticket owner");

        //Checks if the ticket is already validated
        require(idToTicket[idTicket].validated == false, "Ticket
        already validated");

        //Checks if the ticket is already validated
        require(idToTicket[idTicket].isOnSale == true, "Ticket is not
        on sale");

        uint256 idResale = 0;

        for(uint256 i = 0; i < _resalesIds.current(); i++) {
            if(idToTicketResale[i + 1].isSold == false &&
               idToTicketResale[i + 1].idTicket ==
               idTicket && idToTicketResale[i + 1].isCancelled ==
               false){
                idResale = i + 1;
            }
        }

        require(idResale != 0, "Invalid ticket");

        //Change parameters
        idToTicket[idTicket].isOnSale = false;
        idToTicketResale[idResale].isCancelled = true;
}
```

Figura 53: Función cancelResale del Smart Contract TickbitTicket.sol
Fuente: propia

Quita un ticket disponible de la reventa. Recibe por parámetro el identificador del ticket sobre el que vamos a cambiar el estado de reventa.

Con el primer require, comprobamos si el usuario que hace la petición es el dueño del ticket. Con el segundo require comprobamos si el ticket ha sido previamente validado y con el tercero si no está puesto a la venta.

Si cumple los requires modificaremos el objeto que pertenecía a esta reventa (`TicketResale`) y le modificaremos el estado de en venta a false para que se quite de la reventa.

Por último, marcamos el ticket como puesto en venta usando el mapping `idToTicket`.

- **[CU-016] checkAvailavilityFromIdEvent:**

```
function checkAvailavilityFromIdEvent(uint256 idEvent) public view
returns (uint) {
        Tickbit.EventItem memory eventItem =
        tickbitContract.readEvent(idEvent, true);

        return eventItem.capacity -
         idEventToIdTicketArray[eventItem._id].length;
}
```

Figura 54: Función checkAvailavilityFromIdEvent del Smart Contract TickbitTicket.sol
Fuente: propia

Retorna la disponibilidad de un evento.

Recibe como parámetro el identificador del evento y retorna la resta entre la capacidad del evento y los tickets vendidos.

- **[CU-017] readTicketingSales:**

```
function readTicketingSales() public view returns (TicketItem[]
memory) {
        uint256 ticketCount = _ticketsIds.current();
        uint256 myTicketCount = 0;
        uint256 currentIndex = 0;

        //Contract owner gets a counter of all the existing tickets
        //User gets a counter of all the tickets created by him
        for (uint256 i = 0; i < ticketCount; i++) {
            if(idToTicket[i + 1]._eventOwner == msg.sender ||
               msg.sender == owner()) {
                myTicketCount += 1;
            }
        }

        TicketItem[] memory tickets =
        new TicketItem[](myTicketCount);

        //Contract owner gets an array with all the tickets
        //User gets an array with the tickets purchased by him
        for (uint256 i = 0; i < ticketCount; i++) {
            //Checks the owner of the ticket
            if (idToTicket[i + 1]._eventOwner == msg.sender ||
                msg.sender == owner()) {
                //Add ticket to the array
                uint256 currentId = i + 1;
                TicketItem storage currentTicket =
                idToTicket[currentId];
                tickets[currentIndex] = currentTicket;
                currentIndex += 1;
            }
        }

        return tickets;
    }
```

Figura 55: Función readTicketingSales del Smart Contract TickbitTicket.sol
Fuente: propia

Retorna los tickets vendidos cuyos eventos pertenecen a la cartera que hace la petición o es la cartera dueña del smart contract.

Recorre `idToTicket` y devuelve la respuesta.

- **[CU-018] readTickets:**

```
function readTickets() public view returns (TicketItem[] memory) {
        uint256 ticketCount = _ticketsIds.current();
        uint256 myTicketCount = 0;
        uint256 currentIndex = 0;

        //Contract owner gets a counter of all the existing tickets
        //User gets a counter of all the tickets created by him
        for (uint256 i = 0; i < ticketCount; i++) {
            if (idToTicket[i + 1]._owner == msg.sender ) {
                myTicketCount += 1;
            }
        }

        TicketItem[] memory tickets =
        new TicketItem[](myTicketCount);

        //Contract owner gets an array with all the tickets
        //User gets an array with the tickets purchased by him
        for (uint256 i = 0; i < ticketCount; i++) {
            //Checks the owner of the ticket
            if (idToTicket[i + 1]._owner == msg.sender) {
                //Add ticket to the array
                uint256 currentId = i + 1;
                TicketItem storage currentTicket =
                idToTicket[currentId];
                tickets[currentIndex] = currentTicket;
                currentIndex += 1;
            }
        }

        return tickets;
 }
```

Figura 56: Función readTickets del Smart Contract TickbitTicket.sol
Fuente: propia

Retorna los tickets vendidos cuyos eventos pertenecen a la cartera que hace la petición.

Recorre `idToTicket` y devuelve la respuesta.

- **ticketsSoldByIdEvent:**

```
function ticketsSoldByIdEvent(uint256 idEvent) public view returns
(uint256) {
        uint256[] memory idTickets = idEventToIdTicketArray[idEvent];

        return idTickets.length;
}
```

Figura 57: Función ticketsSoldByIdEvent del Smart Contract TickbitTicket.sol
Fuente: propia

Retorna los tickets vendidos para un evento pasado por parámetro.

Consulta el mapping `idEventToIdTicketArray`.

- **getMaticUsd:**

```
function getMaticUsd() public view returns (int) {(,int price,,,) =
chainlinkMaticUSD.latestRoundData();
        return price * 1e10;
}
```

Figura 58: Función getMaticUsd del Smart Contract TickbitTicket.sol
Fuente: propia

Lo primero de todo los pagos se realizan en Wei, como hemos explicado anteriormente los Wei son una unidad más pequeña en la que se puede dividir el token ETH o MATIC en este caso, por lo tanto un MATIC equivale a 10^{18} Wei, por lo tanto las unidades finales que debemos obtener son Wei.

Esta función es una función auxiliar para calcular los valores de precio en Wei, se realiza una petición a Chainlink del precio en USD de un MATIC, lo que nos devuelve el oráculo es el precio del MATIC en Wei pero por 10^8 por lo tanto para pasarlo a 10^{18} lo multiplicamos por 10^{10}.

- **getMaticWeiFromUSD:**

```
function getMaticWeiFromUSD(uint _amuontinUsd) public view returns
(uint){
        uint newInput = _amuontinUsd * 10 ** 18;
        uint MaticUsd = uint(getMaticUsd());

        return (newInput * 10 ** 18) / MaticUsd;
}
```

Figura 59: Función getMaticWeiFromUSD del Smart Contract TickbitTicket.sol
Fuente: propia

Función que llama al oráculo de Chainlink para obtener el cambio de divisa de MATIC a USD. Recibimos el valor en dólares lo multiplicamos por 10^{18}, luego conseguimos el precio del Matic llamando a la función explicada anteriormente y hacemos una división del precio en dólares y el precio del matic para obtener el número exacto de Wei a pagar.

- **[CU-010] validateTicket:**

```
function validateTicket(uint256 idTicket, uint256 validationHash,
uint256 idEvent) public {
        TicketItem memory ticketItem = idToTicket[idTicket];

        require(ticketItem.idEvent == idEvent, "This ticket does not
        correspond to this event");
        require(ticketItem._owner == msg.sender, "Invalid owner");
        require(ticketItem.validated == false, "Ticket already
        validated");
        require(ticketItem.isOnSale == false, "Ticket on sale");

        idTicketToTicketValidation[ticketItem._id] =
        TicketValidation(
            block.timestamp,
            idTicket,
            ticketItem.idEvent,
            validationHash
        );

        emit TicketItemValidation(block.timestamp, idTicket,
        ticketItem.idEvent, validationHash);
}
```

Figura 60: Función validateTicket del Smart Contract TickbitTicket.sol
Fuente: propia

Genera una instancia de validación de ticket.

Recibe por parámetro el identificador del ticket, el hash de validación y el identificador del evento.

Con el primer require, comprobamos si el ticket pertenece al evento que se va a validar. Con el segundo comprobamos si quien realiza la petición es el dueño del ticket. Luego comprobamos si el ticket ha sido previamente validado y con el tercero si está puesto a la venta.

Si cumple los requires añadiremos el objeto `TicketValidation` en el mapping `idTicketToTicketValidation` que guarda las instancias de validación y llamaremos a emitir el evento `TicketItemValidation`.

- **[CU-011] checkTicketValidation:**

```
function checkTicketValidation(uint256 idEvent, uint256
validationHash) public {
        Tickbit.EventItem memory eventItem =
        tickbitContract.readEvent(idEvent, true);

        require(eventItem._owner == msg.sender, "Invalid event
        owner");

        uint256[] memory tickets = idEventToIdTicketArray[idEvent];

        TicketItem memory currentTicket;
        uint256 currentHash = 0;
        bool valid = false;

        for(uint256 i = 0; i < tickets.length; i++) {
            currentTicket = idToTicket[tickets[i]];
            currentHash =
            idTicketToTicketValidation[currentTicket._id]
            .validationHash;

            if(currentTicket.validated != true){
                if(currentHash == validationHash &&
                   currentTicket.isOnSale == false){
                    idToTicket[currentTicket._id].validated = true;
                    valid = true;
                }
            }
        }

        require(valid == true, "Validation not valid");
}
```

Figura 61: Función checkTicketValidation del Smart Contract TickbitTicket.sol
Fuente: propia

Consulta y confirma la validez de la instancia de validación de un ticket.

Se pasa por parámetro el identificador del evento y el hash de validación a consultar.

Primero, consultamos con un require si el validador de la petición es el diseño del evento que se está validando.

Después recorremos el mapping `idToTicket` para buscar el ticket que coincide con el hashing en `idTicketToTicketValidation`. En caso de que el Hashing coincida, comprobamos que el ticket en cuestión no haya sido ya validado. Si no lo está lo marcamos como validado para finalizar la validación.

Por último hacemos un require para asegurarnos de que lanzamos una excepción en caso de que no hayamos encontrado ninguna estancia de validación en el recorrido del mapping.

Si la función no devuelve ninguna excepción quiere decir que la validación ha sido correcta.

- **[CU-021] checkTicketValidationTest:**

```
function checkTicketValidationTest(uint256 idEvent, uint256
validationHash) public view {
        Tickbit.EventItem memory eventItem =
        tickbitContract.readEvent(idEvent, true);

        require(eventItem._owner == msg.sender, "Invalid event
        owner");

        uint256[] memory tickets = idEventToIdTicketArray[idEvent];

        TicketItem memory currentTicket;
        uint256 currentHash = 0;
        bool valid = false;

        for(uint256 i = 0; i < tickets.length; i++) {
            currentTicket = idToTicket[tickets[i]];
            currentHash =
            idTicketToTicketValidation[currentTicket._id]
            .validationHash;
```

```
            if(currentTicket.validated != true){
                if(currentHash == validationHash &&
                   currentTicket.isOnSale == false){
                    valid = true;
                }
            }
        }

        require(valid == true, "Validation not valid");
    }
}
```

Figura 62: Función checkTicketValidationTest del Smart Contract TickbitTicket.sol
Fuente: propia

Misma función que el `checkTicketValidation` pero en esta ocasión no modificamos el mapping `idToTicket` por que solo se utiliza a modo de consulta.

2. CONFIGURACIÓN METAMASK

Ya instalada la extensión y creada la cuenta en Metamask, lo primero que haremos es hacer click en la parte superior derecha del navegador en el símbolo de la extensión de Metamask, una vez abierta presionaremos el botón de la parte superior donde indica la red en la que estamos y se nos desplegará un menú donde iremos a agregar red.

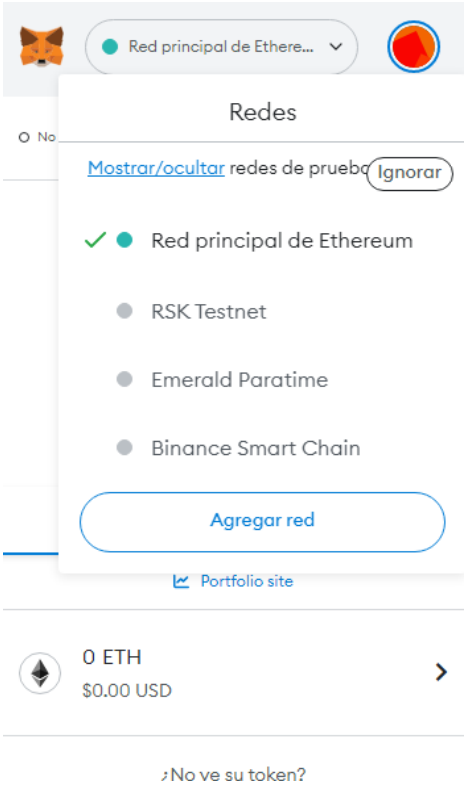


Figura 63: Proceso de configuración de la red Mumbai Polygon en Metamask. Fuente: propia

Ahora se habrá abierto una página en la cual hay un formulario a rellenar con los datos de la red que queremos añadir y lo rellenaremos con los siguientes datos:

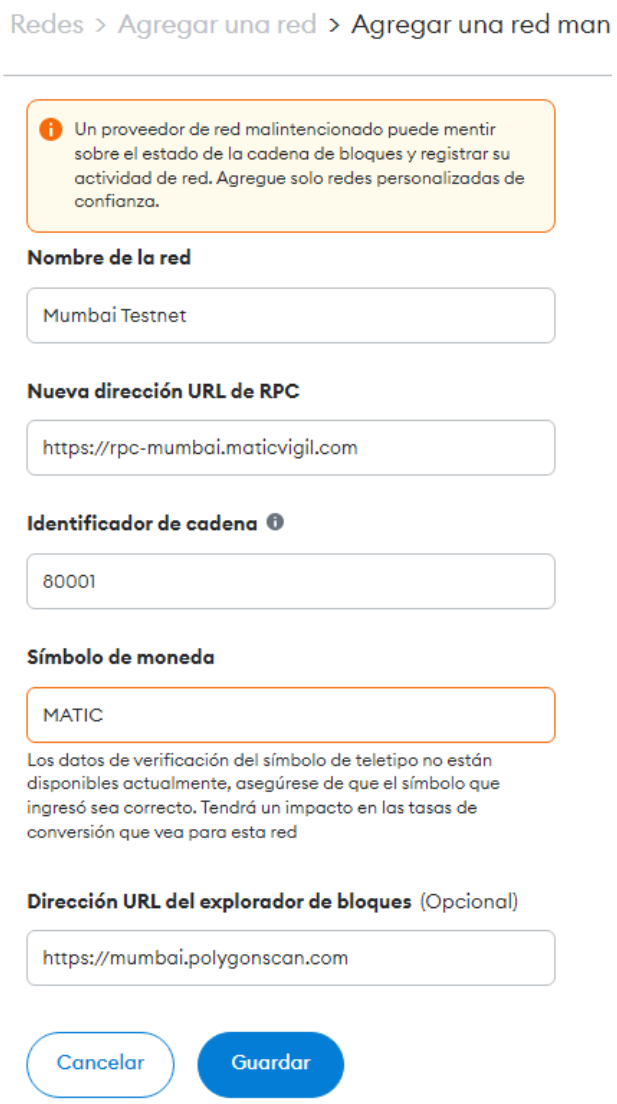


Figura 64: Proceso de configuración de la red Mumbai Polygon en Metamask. Fuente: propia

Nombre de la red: Mumbai Testnet

Nueva dirección URL de RPC: https://rpc-mumbai.maticvigil.com

Identificador de cadena: 80001

Símbolo de moneda: MATIC

Dirección URL del explorador de bloques: https://mumbai.polygonscan.com

Una vez hecho esto ya tendremos la red configurada y disponible para utilizar en la plataforma Tickbit en el menú de redes de la parte superior de la extensión.

3. ÍNDICE DE NAVEGACIÓN Y USO

3.1. WEB CLIENTES

El código íntegro de esta web puede consultarse en:

https://github.com/tickbit-dev/tickbit-web

- **Inicio:**

  La primera pantalla que nos vamos a encontrar entrando en la web de clientes https://tickb.it es la pantalla de Inicio:

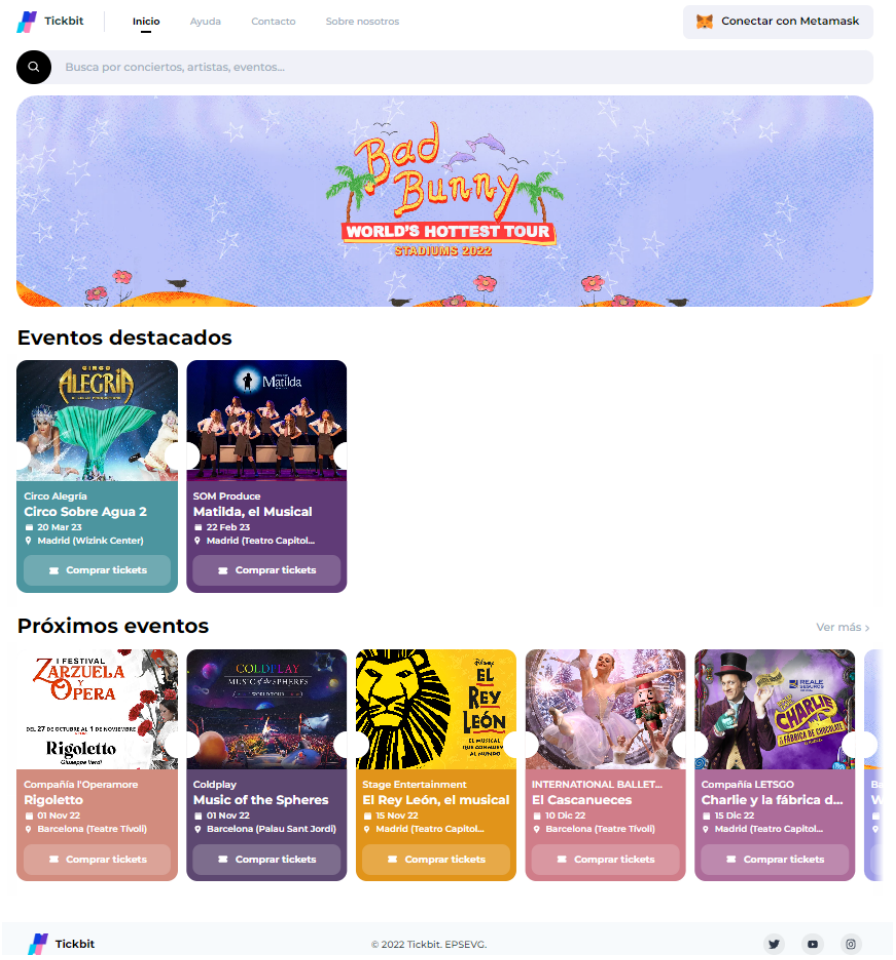


Figura 65: Pantalla de Inicio

Esta página corresponde al componente HomePage.js, en la parte superior encontramos una barra de navegación que va a estar presente en todas las demás pantallas que corresponde al componente NavigationBar.js la cual contiene un botón para conectar tu billetera Metamask y los enlaces a las páginas Inicio, Ayuda, Contacto y Sobre nosotros.

Más abajo hay un buscador el cual permite buscar el evento deseado. Además la página Inicio consta de una imagen principal que es el evento activo actualmente como Portada y más abajo de los eventos destacados, ambos comprados a través del sistema de campañas de la web backoffice.

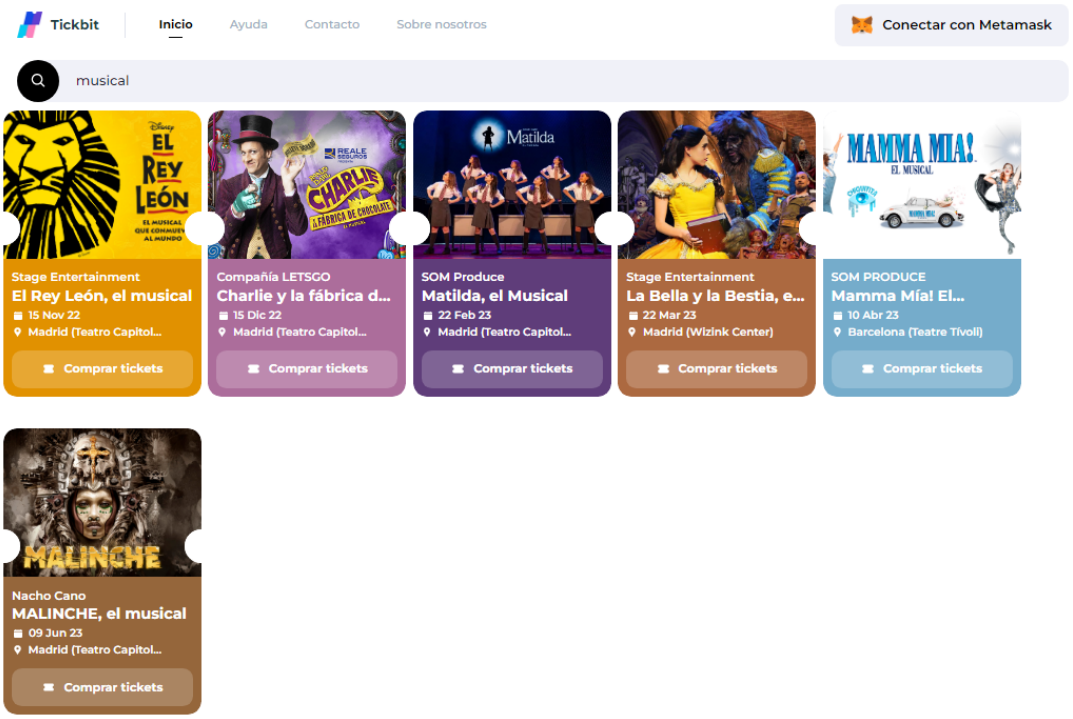


Figura 66: Filtrado de eventos

En la parte inferior están los eventos ordenados de fecha más cercana a la actual a la más lejana y un pie de página correspondiente al componente Footer.js que también estará presente en las demás páginas.

- **Ayuda:**

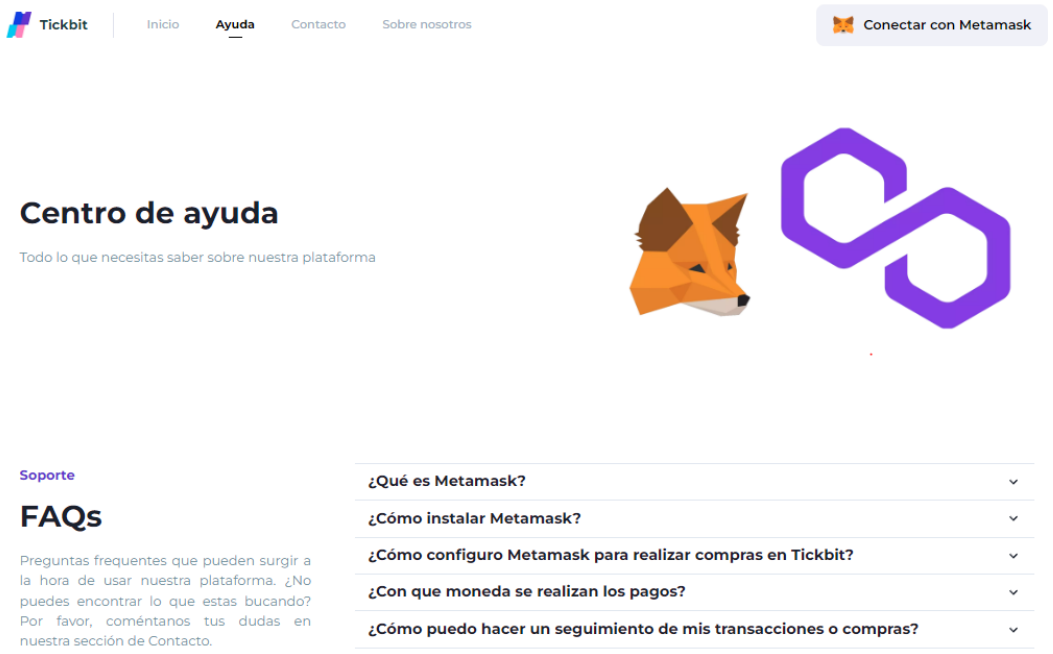


Figura 67: Pantalla de Ayuda

Corresponde al componente HelpPage.js y sólo tiene un propósito informativo. En ella hay diferentes preguntas típicas que pueden surgir al utilizar la plataforma y sus respuestas

- **Contacto:**

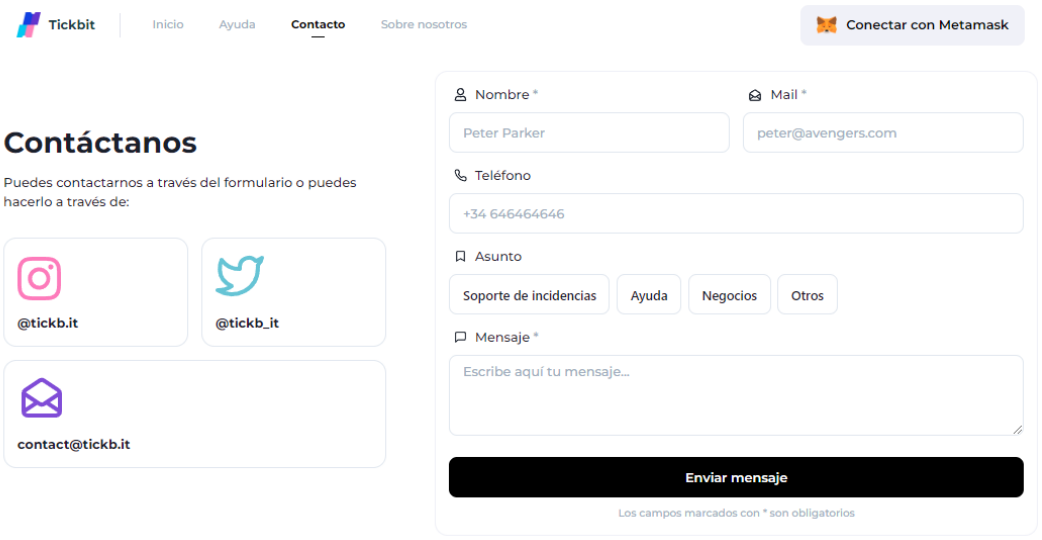


Figura 68: Pantalla de Contacto

Corresponde al componente ContactPage.js y contiene un formulario donde poder escribirnos al correo de contact@tickb.it si ocurre cualquier incidencia o duda además de nuestras redes sociales.

Si hacemos una prueba y rellenamos el formulario con la siguiente información y lo enviamos:

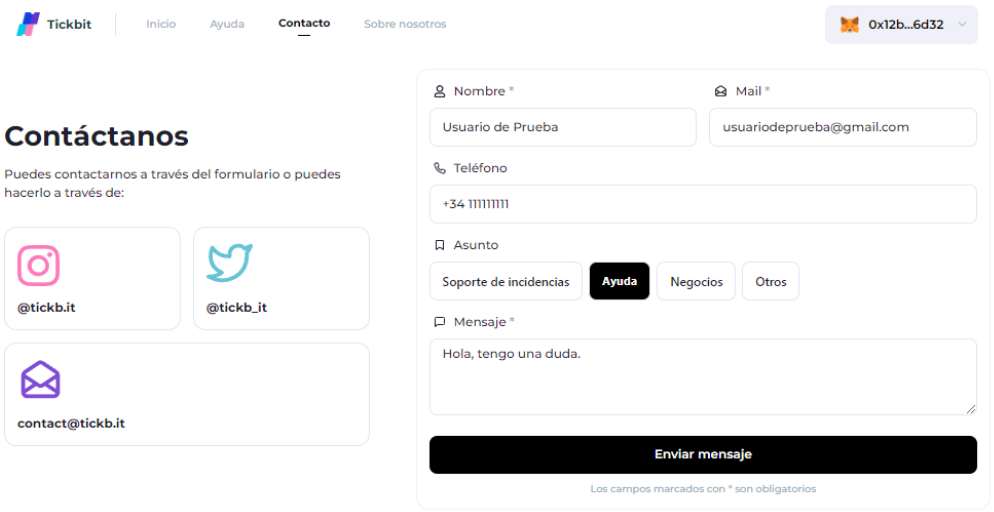


Figura 69: Formulario Contacto rellenado

Podemos ver como nos llega una respuesta a nuestro correo con toda la información puesta en el formulario.

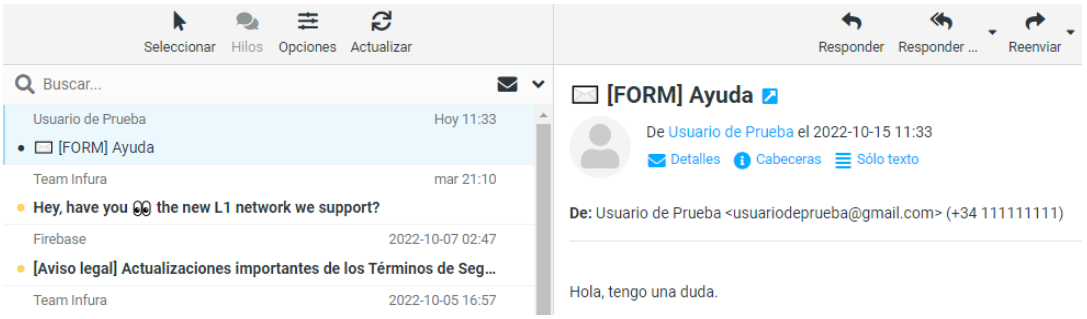


Figura 70: Correo recibido

- **Sobre nosotros:**

Figura 71: Pantalla Sobre nosotros

Corresponde al componente AboutUsPage.js y es una página de información sobre nosotros los creadores de la Plataforma Tickbit.

- **Evento:**

La pantalla de eventos corresponde al componente EventsDetailsPage.js y está compuesta por tres pasos que forman el proceso de compra que se va a realizar a continuación.

En el primer paso tenemos la información sobre el evento donde indica el título el artista, el título del evento, la fecha, el lugar donde se hace, la ciudad el precio y una breve descripción.

Esta pantalla se puede mostrar de tres formas distintas, la primera es cuando aún existen entradas disponibles para la venta normal, la segunda es cuando los tickets están agotados y la última cuando los tickets están agotados pero hay algún ticket en reventa.

Figura 72: Pantalla Selección de evento con tickets aún disponibles

Figura 73: Pantalla Selección de evento con tickets agotados

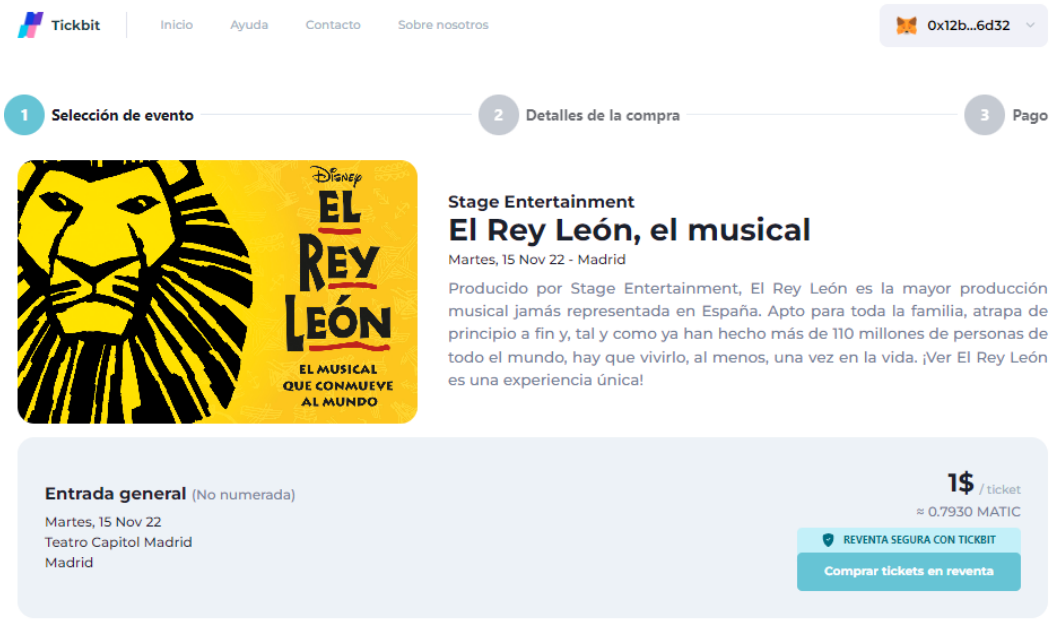


Figura 74: Pantalla Selección de evento con tickets agotados pero con algún ticket en reventa

En el segundo paso tenemos los detalles de la compra donde encontramos la selección del número de tickets y un apartado donde se ve el precio total a pagar por los tickets y el botón para realizar el pago.

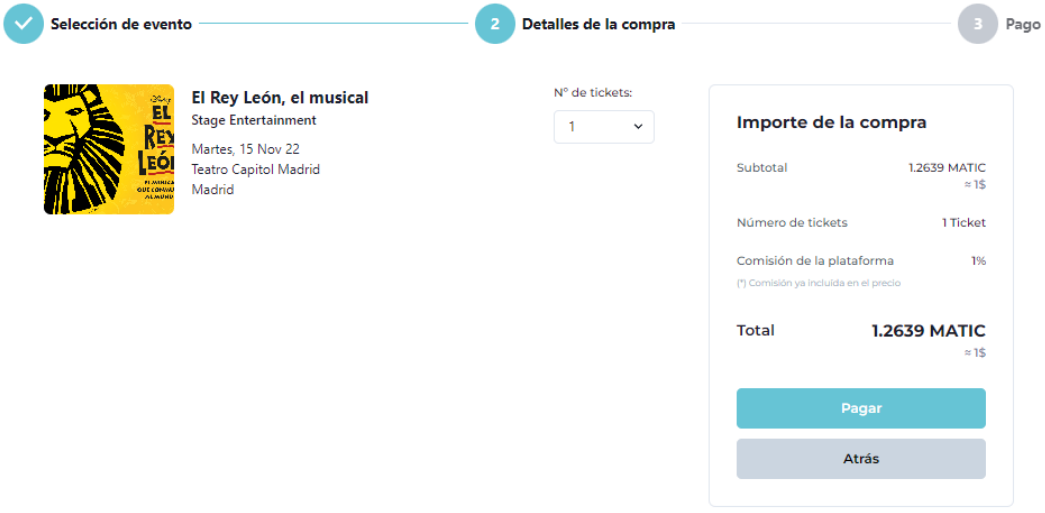


Figura 75: Pantalla Detalles de la compra

Por último en el tercer paso tenemos donde se nos informa que el pago está aún pendiente y se nos abrirá Metamask para realizar el pago. Una vez se ha realizado y procesado correctamente nos pondrá un mensaje informativo y todos los detalles de los tickets comprados. Además hay un botón que nos redirigirá a nuestro perfil para observar los tickets adquiridos.

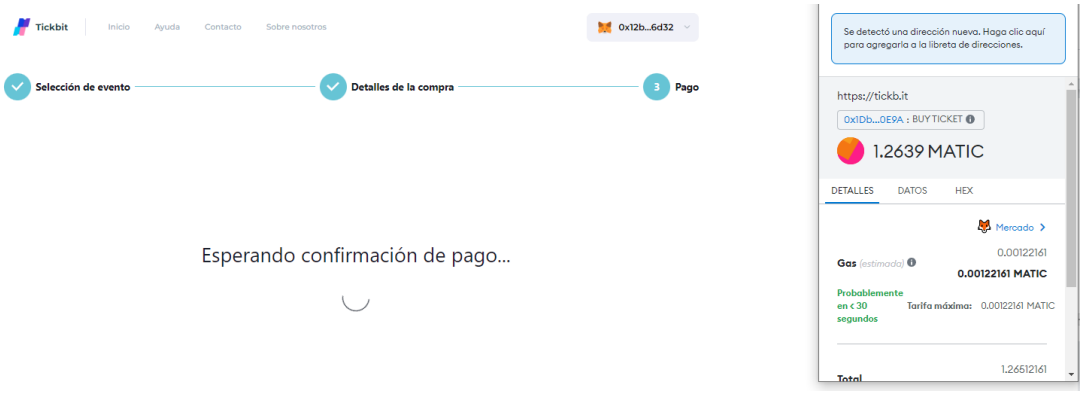


Figura 76: Pantalla Pago pendiente

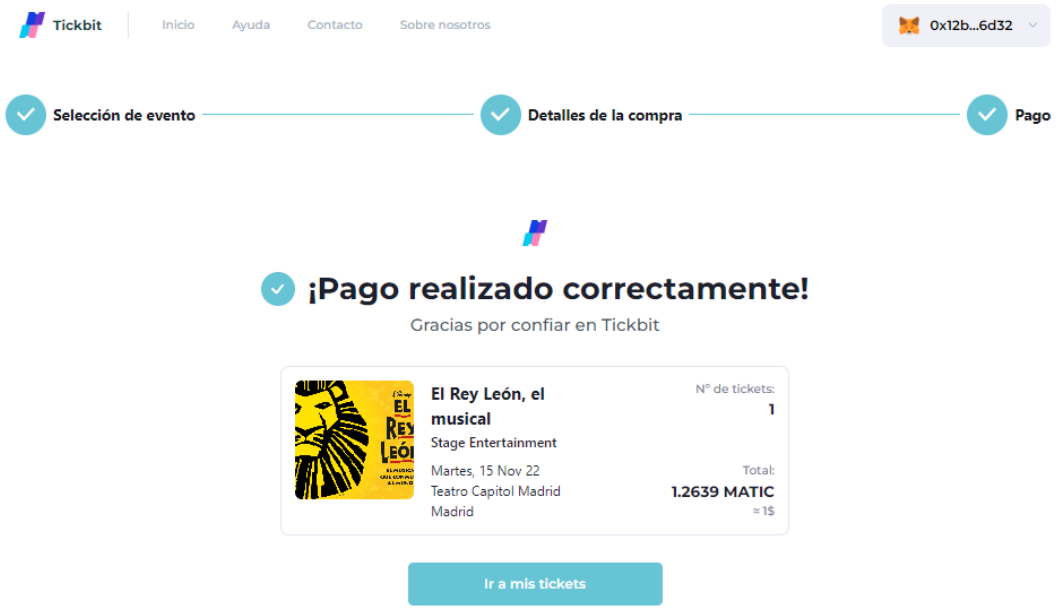


Figura 77: Pantalla Pago finalizado

Además para asegurarnos que se ha hecho todo bien, podemos consultar la información de la billetera `0x12B98A91392075077b2a84d7889140734d736d32` que es con la que se realiza esta transacción en la web https://mumbai.polygonscan.com y veremos como se ha realizado el pago correctamente a la dirección del contrato y que la billetera del cliente posee el ticket.

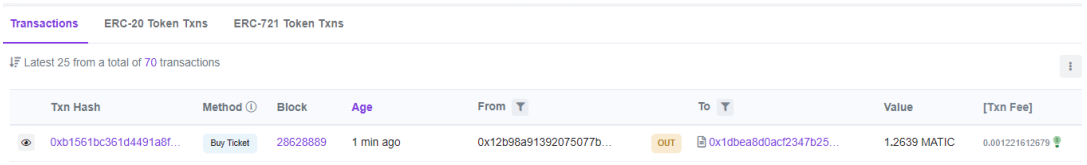


Figura 78: Transacción de compra de ticket

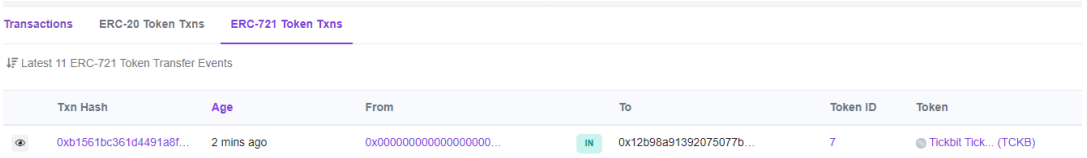


Figura 79: Posesión del ticket

SI ahora vamos a la dirección del smart contract `0x1Dbea8d0acf2347b25BaFfe4977F11C276600E9A`, cómo tenemos fijado que por cada compra de un ticket nos llevamos una pequeña comisión, podemos ver que los 1.2639 Matic del pago se han dividido en dos partes una pequeña parte que es la comisión se ha transferido a la billetera de Ticbkit y el restante a la billetera del artista.

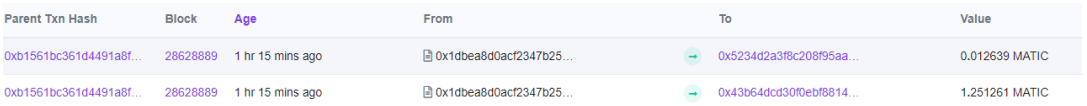


Figura 80: División del pago

- **Tickets:**

  Esta pantalla corresponde con el componente MyTicketsPage.js y es el apartado más personal ya que es donde se pueden ver los tickets comprados, tanto los aún vigentes como los finalizados. Además cada ticket que aun siga disponible, se puede poner en reventa o validarlo para acceder al evento. Al ponerlo en reventa saldrá un mensaje de confirmación conforme si estamos de acuerdo en aceptar las condiciones de las comisiones por reventa de la plataforma Tickbit. Si en cambio se quiere validar el ticket, se abrirá una ventana con la cámara activada por la cual deberemos escanear un QR para poder validar el ticket.

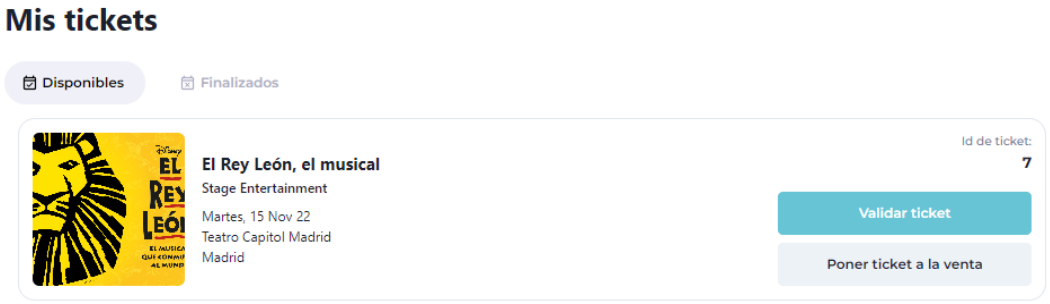


Figura 81: Pantalla Tickets

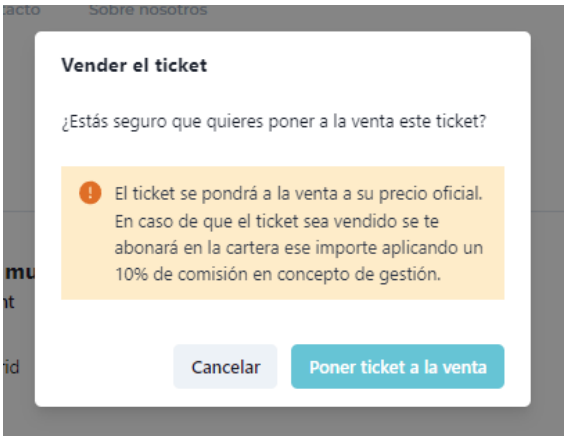


Figura 82: Poner ticket a la venta

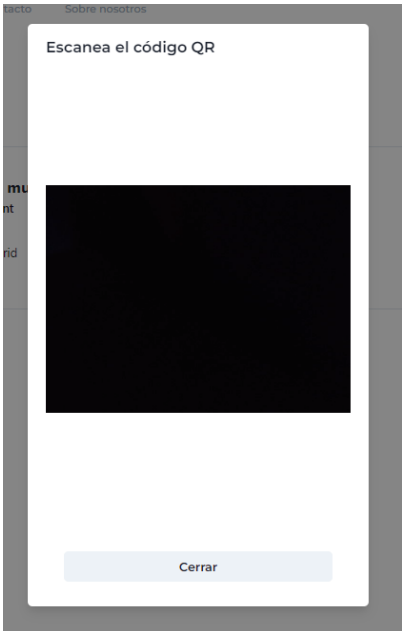


Figura 83: Validar ticket

Si el ticket está puesto en reventa su apariencia cambia y pasa a tener solo un botón de cancelar venta, el cual si presionamos podemos anular la reventa si aún no ha sido vendido a otra persona y nos volverá a dejarlo a validar en caso de que queramos.

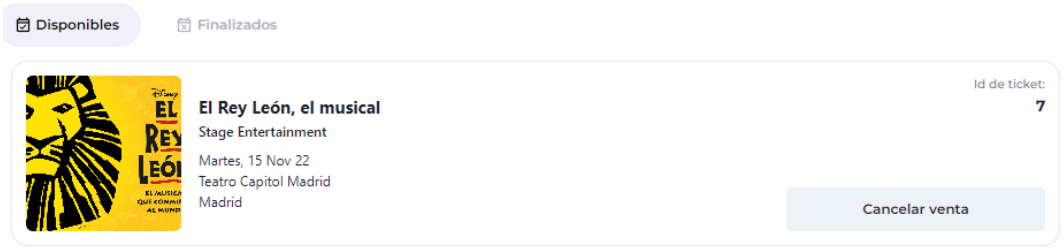


Figura 84: Ticket puesto en reventa

Si en cambio el ticket ya ha sido validado aparecerá en la sección de finalizados donde ya no se podrá realizar ninguna acción sobre él.

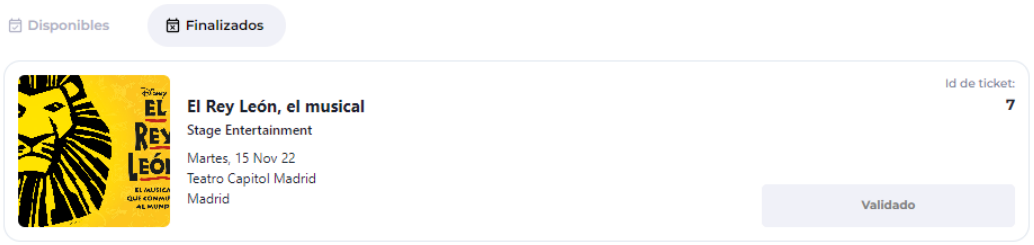


Figura 85: Ticket validado

3.2. WEB BACKOFFICE

El código íntegro de esta web puede consultarse en:

https://github.com/tickbit-dev/tickbit-backoffice

- **Login:**

  La primera pantalla que nos vamos a encontrar entrando en la web de clientes https://business.tickb.it es la pantalla de login que corresponde al componente LoginScreen.js donde se nos requerirá entrar con una cuenta de Metamask.

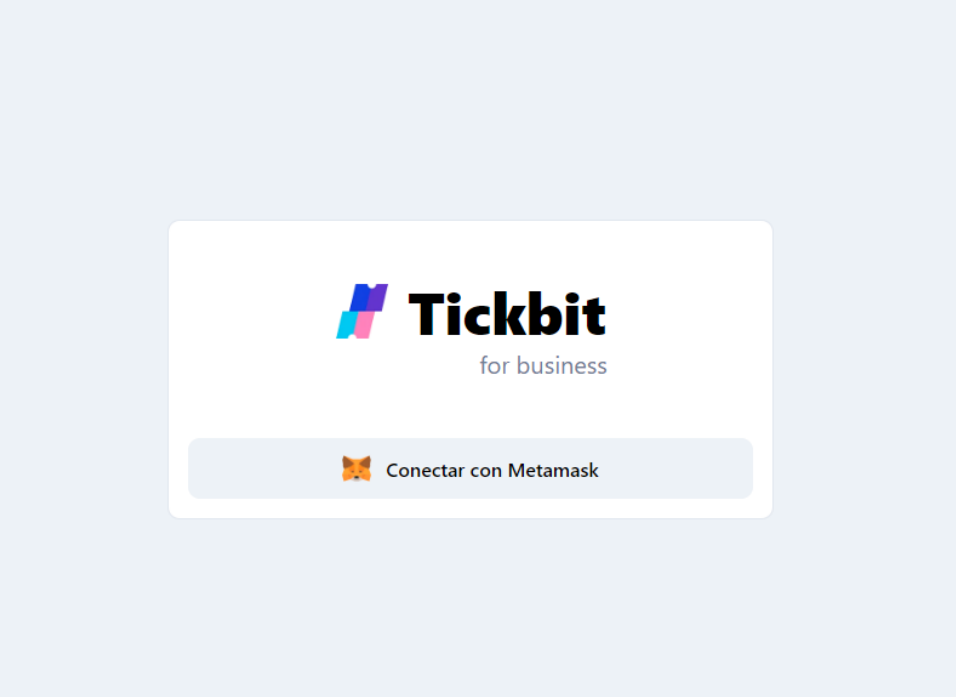


Figura 86: Página de Login

- **Home/Eventos:**

  La siguiente pantalla al Login es el Home que te redirige automáticamente a la pantalla de Eventos que corresponde al componente EventsTab.js.

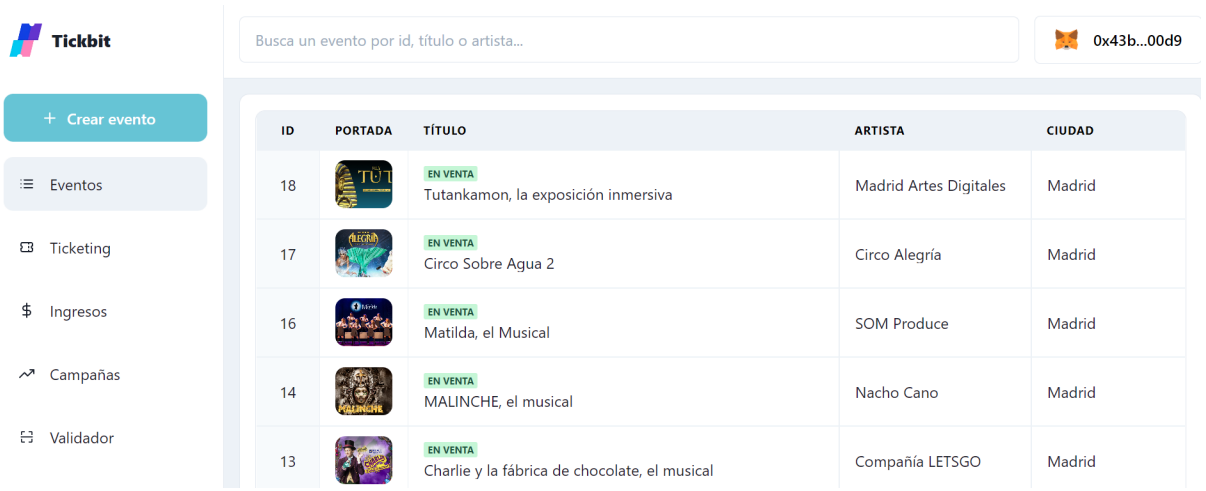


Figura 87: Página Home/Eventos

Esta página consta de un menú lateral que está presente en todas las pantallas de la web backoffice y que nos permite navegar entre todas las vistas que són, Crear evento, Eventos, Ticketing, Ingresos, Campañas, Validador, Ayuda y Admin.

En el centro encontramos una tabla donde están listados todos los eventos con algunos datos como su ID, título, artista y ciudad.

En la parte superior hay una barra de navegación que nos permite filtrar los eventos por id, por título o artista y un apartado que nos indica la billetera con la que estamos conectados.

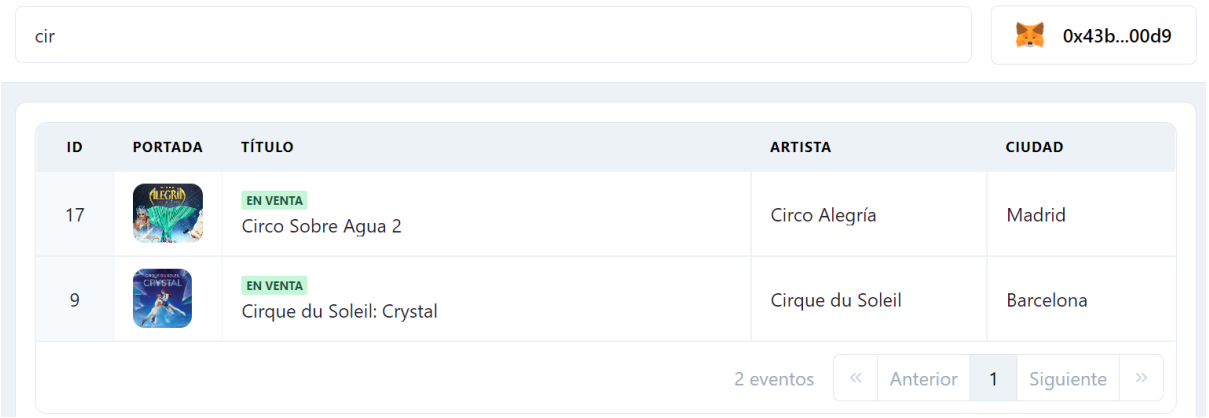


Figura 88: Buscador para filtrar eventos

Si se pulsa sobre cualquier evento ya creado te envía a la página de un evento determinado que corresponde al componente CreateOrUpdateEventTab.js donde se nos muestra información más detallada y donde se nos permite hacer modificaciones o eliminaciones. Para modificar solo hace falta cambiar algún valor del formulario y presionar el botón Modificar evento, si por el contrario quieres eliminar un evento se ha de presionar el botón Eliminar evento.

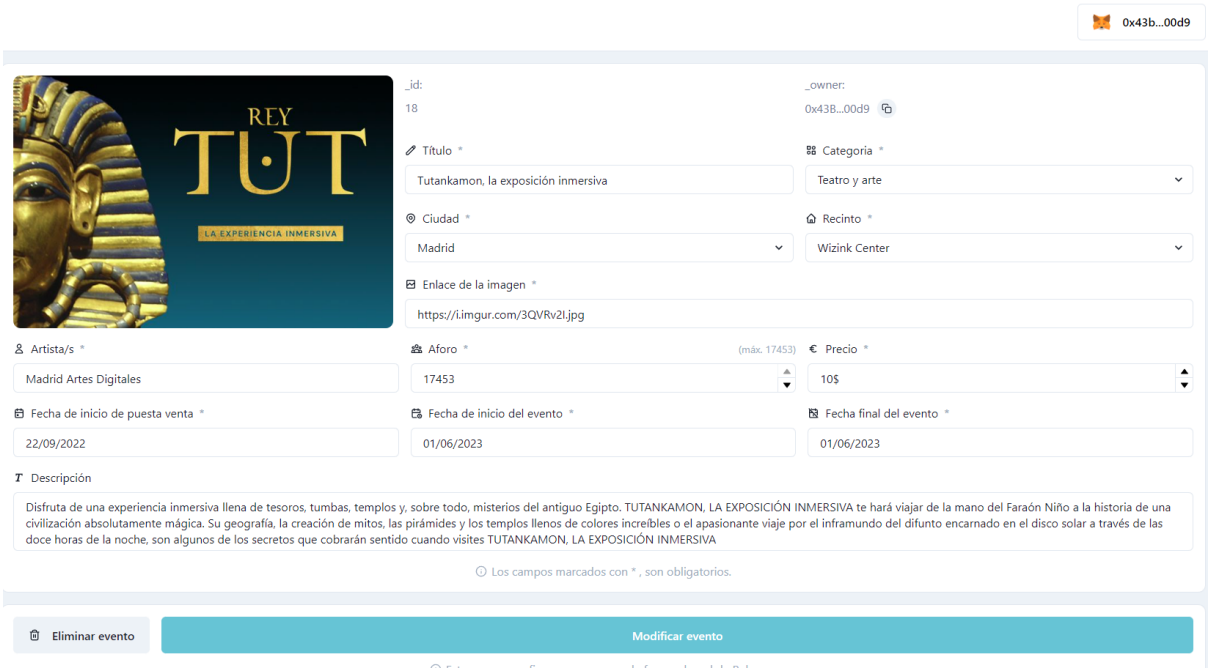


Figura 89: Información detallada sobre un evento

Los eventos eliminados solo están disponibles en la tabla de eventos si la billetera que has conectado es la creadora del smart contract, es decir la billetera de Tickbit. Además si estás conectado con la billetera de Tickbit tienes la posibilidad de poder restaurar un evento eliminado.

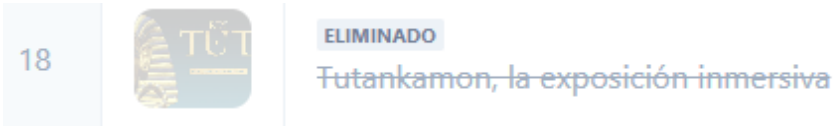


Figura 90: Evento eliminado

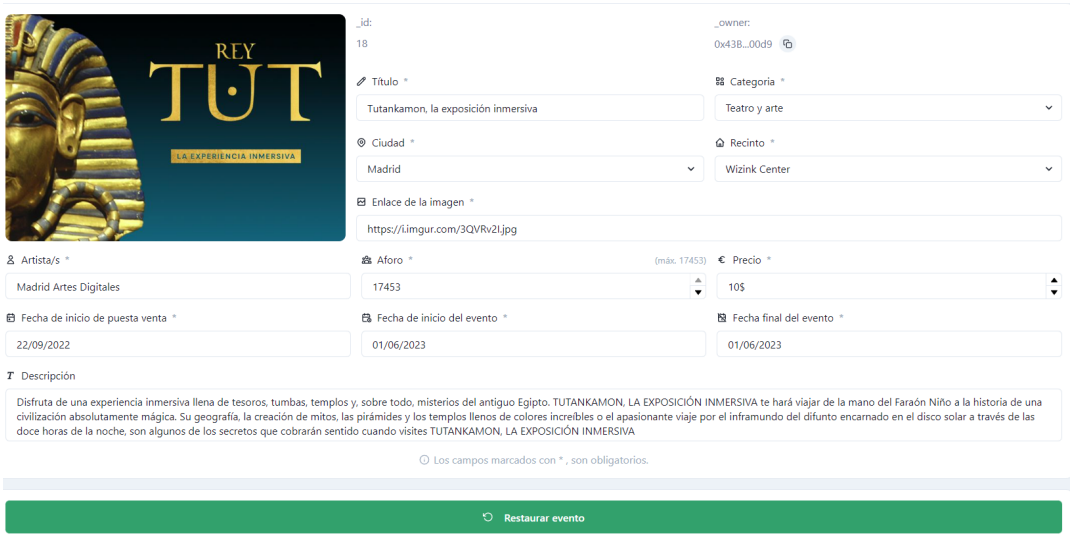


Figura 91: Opción de restaurar un evento eliminado.

- **Formulario evento:**

  A esta página se accede mediante el botón de Crear evento del menú lateral y corresponde al mismo componente que se ha explicado anteriormente CreateOrUpdateEventTab.js. En esta página encontramos un formulario vacío que se ha de rellenar con la información del evento que queramos crear.

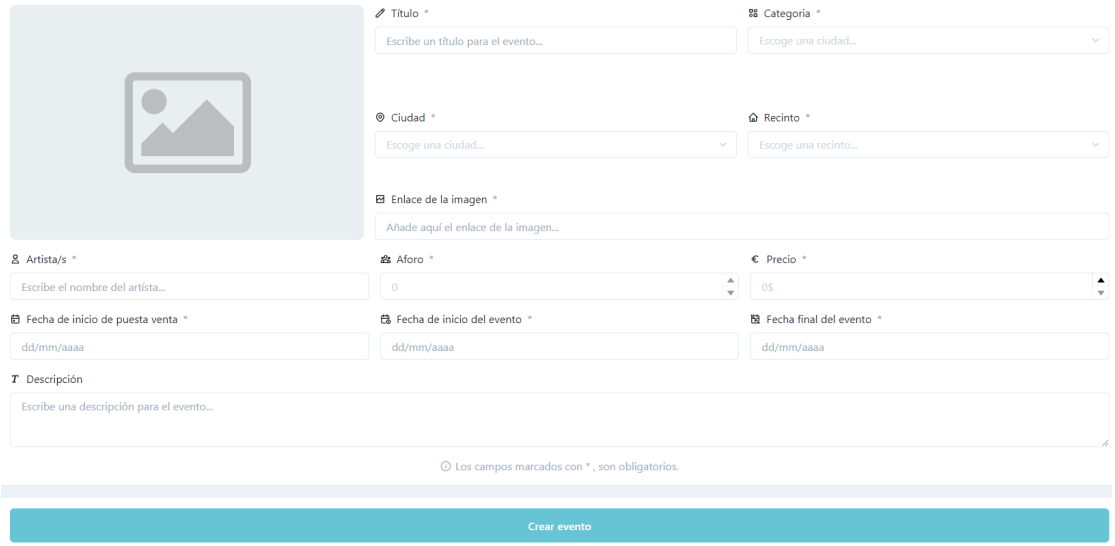


Figura 92: Formulario para crear un evento

- **Ticketing:**

  Esta pantalla corresponde al componente TicketingTab.js, se muestra un gráfico de barras desglosado por meses que nos indica el número de tickets vendidos por mes. Más abajo hay una tabla que muestra todos los tickets vendidos con su ID, el evento al que pertenece, el propietario, la fecha de compra, la ciudad y el lugar y el precio. En la parte superior hay una barra de búsqueda para filtrar tickets por ID o dirección de billetera del comprador.

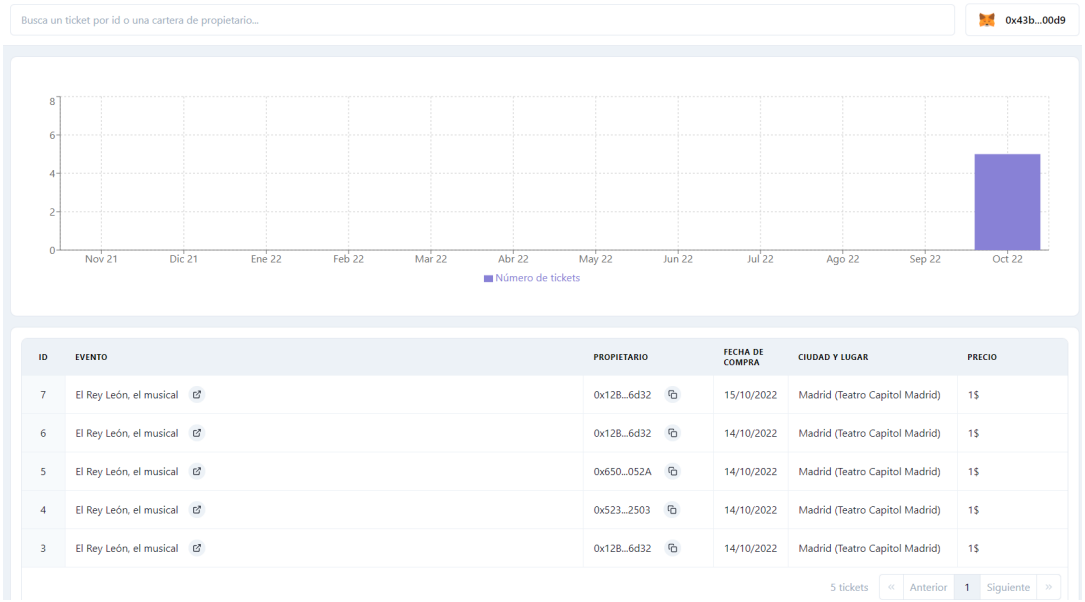
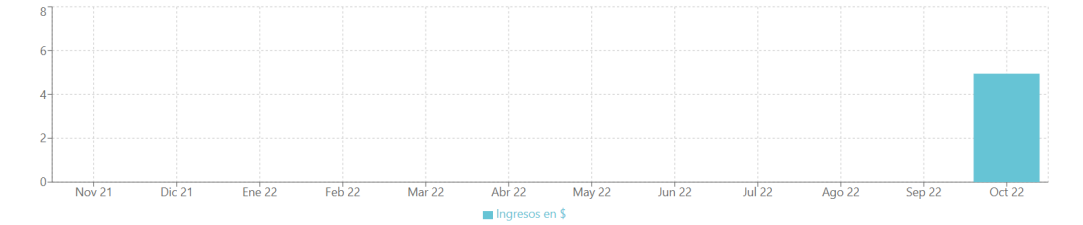


Figura 93: Pantalla Ticketing

- **Ingresos:**

  Esta página corresponde al componente IncomesTab.js y está enfocada a la parte de ingresos tanto para nosotros como plataforma como para los artistas y organizadores de eventos. Al estar conectado con una billetera de Tickbit o con una normal de artista, la visión de esta página tiene unos pequeños cambios.

  Ambas versiones tienen un gráfico desglosado por meses con los ingresos generados en dólares. Y en la parte inferior una tabla donde nos indica una información más detallada sobre los ingresos: su fecha, el número de tickets vendidos con las reventas y el total en dólares. Si estás conectado con la billetera de Ticbkit además en la tabla se añade un campo que son los ingresos por campañas publicitarias.



FECHA	TICKETS VENDIDOS	REVENTAS	INGRESOS
Octubre de 2022	5	0	4,95$
Septiembre de 2022	0	0	0,00$
Agosto de 2022	0	0	0,00$
Julio de 2022	0	0	0,00$
Junio de 2022	0	0	0,00$

Figura 94: Pantalla Ingresos billetera artista

FECHA	TICKETS VENDIDOS	REVENTAS	CAMPAÑAS VENDIDAS	INGRESOS
Octubre de 2022	7	0	4	24,07$
Septiembre de 2022	0	0	0	0,00$
Agosto de 2022	0	0	0	0,00$
Julio de 2022	0	0	0	0,00$
Junio de 2022	0	0	0	0,00$
Mayo de 2022	0	0	0	0,00$

Figura 95: Tabla Ingresos billetera Tickbit

- **Campañas:**

  En esta sección es donde se pueden comprar las campañas publicitarias, corresponde al componente CampaignsTab.js. Hay dos tipos de campañas disponibles, las de portada que solo puede haber una a la semana y las de destacado que hay 5 disponibles por semana, cada una tiene un importe diferente.

  Para comprar una campaña es necesario escoger la semana para la cual vas a realizar la campaña y el evento que quieres publicitar, una vez seleccionado solo has de darle al botón de comprar.

  En la parte inferior hay una tabla en la cual se mostraran las campañas que has comprado, con su ID, el evento, el tipo de campaña, la semana, y el precio en Matic.

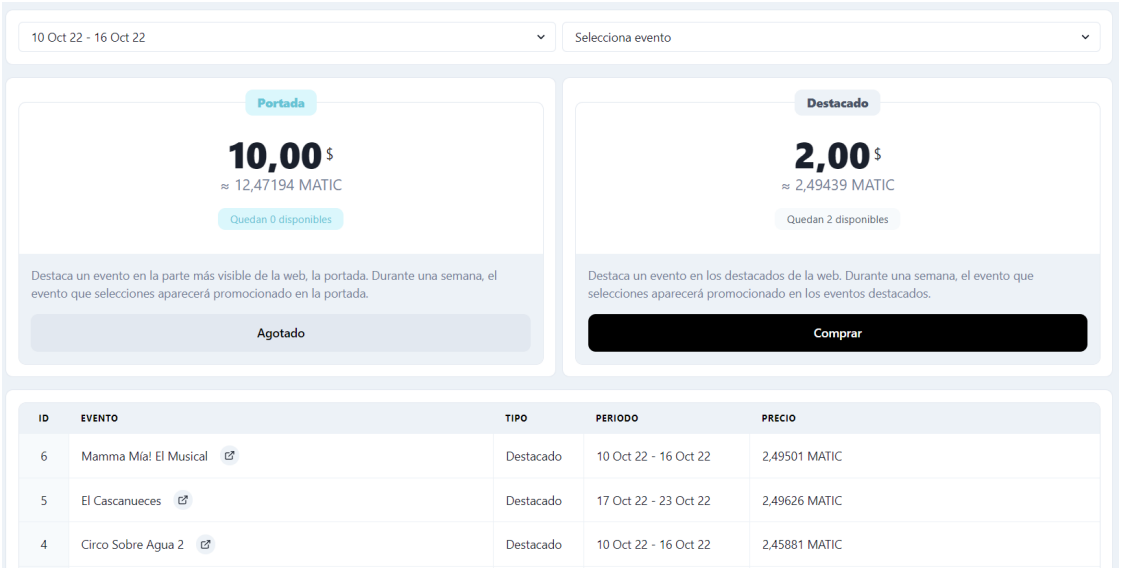


Figura 96: Pantalla de Campañas

Como se puede observar en la figura de arriba en la tabla se ve una nueva campaña de destacados del evento Mamma Mía! El Musical que acabamos de comprar. Para ver que todo se ha realizado correctamente nos volvemos a ir a la web de Mumbai Polygon Scan con la dirección de la billetera que ha realizado la compra `0x43B64DcD30F0EBf8814D752B86eFAb3926EA00d9`.

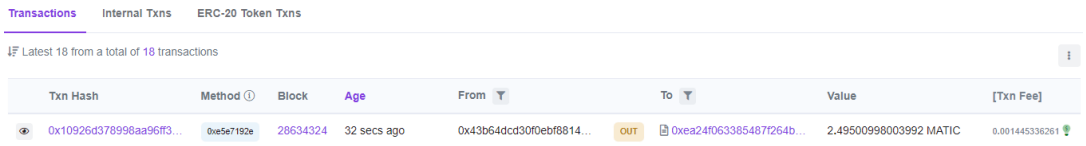


Figura 97: Pago realizado al contrato

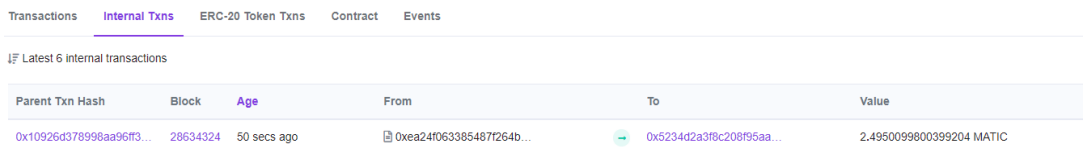


Figura 98: Transferencia de dinero a la billetera de Tickbit

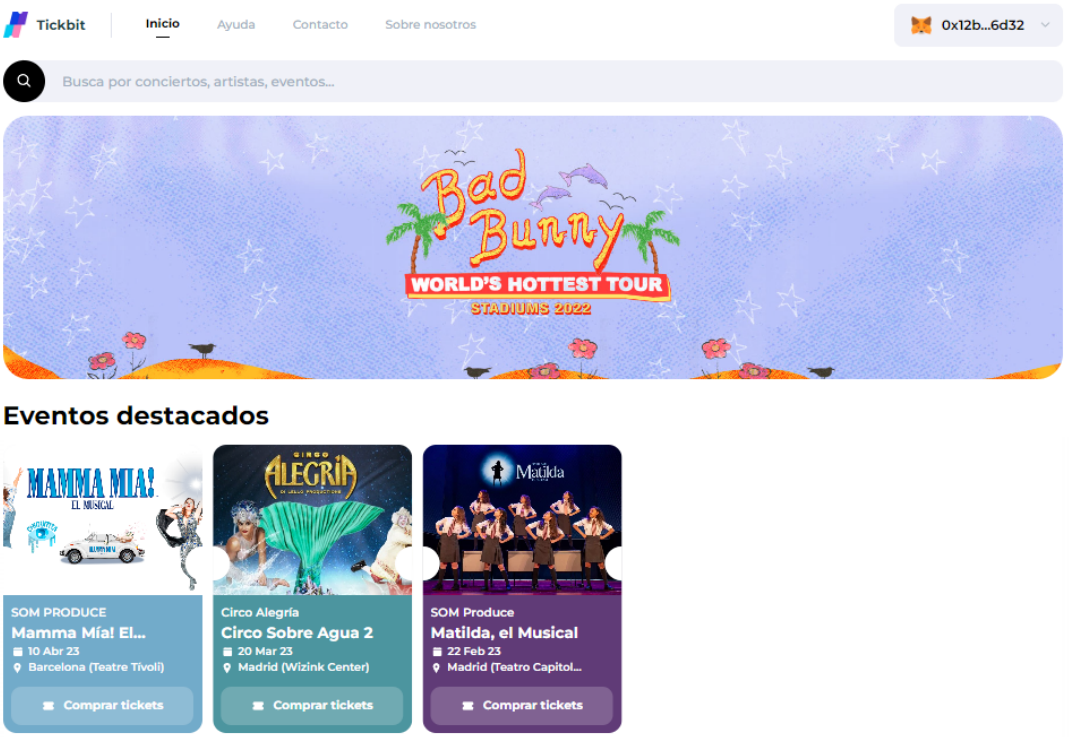


Figura 99: nueva campaña añadida al Home de la web de clientes

Vemos que los pagos y la transferencia de dinero se ha hecho bien y que la campaña se ha añadido correctamente al Home de la web de clientes.

- **Validador:**

Esta sección les sirve a los organizadores de eventos y artistas para validar los tickets en la entrada de un evento y corresponde al componente ValidatorTab.js. En ella podemos encontrar dos campos vacíos que nos pide seleccionar el evento a validar y la clave privada de la cartera.

Para exportar la clave privada de una cuenta de Metamask hay que ir a la extensión y hacer click en los tres puntos suspensivos, detalles de la cuenta y exportar clave privada.

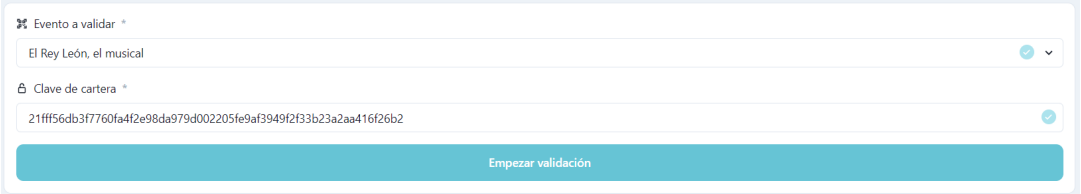


Figura 100: Página de Validación

Una vez seleccionado el evento y introducido la clave pulsamos el botón de Empezar validación y se nos mostrará un código QR el cual los clientes tendrán que escanear cuando le den a validar el ticket. Si el ticket se valida correctamente sale un mensaje conforme se ha validado y se genera un nuevo QR, al contrario si no se valida correctamente sale un mensaje de error y el cliente tendrá que volver a validarlo.

Figura 101: Código QR para validar