Medusa: una aplicación web progresiva para la simulación de inversiones en criptodivisas

Medusa: a progressive web app for simulating cryptocurrency investments

Trabajo de Fin de Grado
Curso 2020–2021

Autor
Jorge Roselló Martín

Director
Enrique Martín Martín

Grado en Ingeniería de Software
Facultad de Informática
Universidad Complutense de Madrid

Medusa: una aplicación web progresiva para la simulación de inversiones en criptodivisas

Medusa: a progressive web app for simulating cryptocurrency investments

Trabajo de Fin de Grado en Ingeniería de Software
Departamento de Sistemas Informáticos y Computación

Autor
Jorge Roselló Martín

Director
Enrique Martín Martín

Convocatoria: *Junio 2021*

Grado en Ingeniería de Software
Facultad de Informática
Universidad Complutense de Madrid

Agradecimientos

A mi tutor Enrique, que me ha ayudado con el proyecto y siempre ha estado ahí contestando mis dudas y dándome muy buenos consejos. A mis padres por siempre creer en mí y apoyarme en todo lo que me he propuesto. A Álvaro por ayudarme a corregir fallos de escritura a última hora. Y por último a ti, Valeria, gracias por escucharme y por motivarme a seguir adelante, sin ti Medusa no hubiera sido posible.

Resumen

Medusa es una aplicación web progresiva que te permite simular la compra y la venta de criptodivisas. El proyecto tiene dos componentes: el servidor y el cliente. El servidor integra una API para poder acceder a los precios de las divisas en tiempo real. El lado del cliente de Medusa se ha diseñado para que sea fácil e intuitivo de usar, permitiendo el acceso a usuarios que sienten curiosidad por invertir en criptodivisas pero desconocen cómo hacerlo. Para acceder a Medusa no es necesario registrarse con un correo y una contraseña, solo se necesita una cuenta de Google. La aplicación web es accesible a través del enlace:

`https://medusapp.live`

El código tanto del cliente como del servidor se encuentran alojados en un repositorio de Github y pueden ser accedidos a través de los siguientes enlaces:

`https://github.com/beybo/medusa`

`https://github.com/beybo/medusa-server`

Palabras clave

Vue.js, criptodivisa, PWA, aplicación web progresiva, SPA, aplicación página única, Node.js, JavaScript, simulador, WebSocket

Abstract

Medusa is a progressive web application that allows you to simulate the purchase and sale of cryptocurrencies. The project has two components: the server and the client. The server integrates and API to access cryptocurrency prices in real time. The client has been designed to be easy and intuitive to use, allowing access to users interested in investing but do not know how to do it. To use Medusa, not necessary to register using a user and a password, you only need a Google account. The web application is accessible through the link:

`https://medusapp.live`

The code of the client and the server is hosted on a Github repo and can be accessed through the following links:

`https://github.com/beybo/medusa`

`https://github.com/beybo/medusa-server`

Keywords

Vue.js, criptodivisa, PWA, progressive web application, SPA, single page application, Node.js, JavaScript, simulator, WebSocket

Índice

Índice de figuras

Índice de tablas

Índice de código

Capítulo 1

Introducción

En este capítulo se va a explicar la motivación para desarrollar Medusa, los objetivos y el plan de trabajo que se ha seguido.

1.1. Motivación

Las criptodivisas son un tema de conversación cada vez más recurrente, a pesar de que muchas personas no las entienden o no saben cómo funcionan, están interesadas en invertir algo de dinero. Siempre que ocurre una subida o una bajada del precio todo el mundo habla de ello. El problema de las criptodivisas es que son un mercado muy volátil, lo que puede provocar que las inversiones realizadas caigan en picado o se tripliquen. Por ello, es importante educar y permitir que las personas puedan simular la inversión en el mercado de las criptodivisas sin ningún tipo riesgo, de esta idea nace Medusa.

1.2. Objetivos

El objetivo principal de este proyecto es desarrollar un simulador web de inversión en criptodivisas usando tecnologías y bibliotecas modernas. Las metas concretas de Medusa son:

- Construir una aplicación web progresiva con un diseño moderno y atractivo que sea fácil de usar para que cualquier usuario independientemente de su nivel técnico pueda utilizarla.

- Diseñar la web para que sea una aplicación de página única, así la experiencia de usuario de la web va a ser comparable a la de una aplicación nativa, ya que no hay cargas entre páginas.

- Realizar una comunicación en tiempo real con el servidor, que permita actualizar elementos como los precios de las criptodivisas sin interacción por parte del usuario.

- Registrar y autenticar usuarios mediante su cuenta de Google para no tener que implementar un sistema de registro propio, y con esto alentar a los usuarios a usar la aplicación ya que no es necesario registrarse usando un usuario y una contraseña.

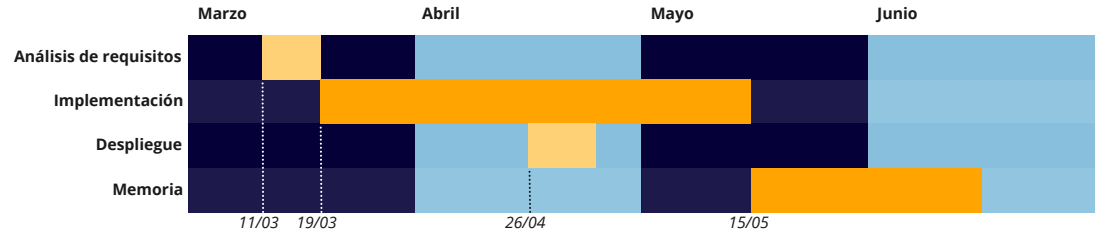

Figura 1.1: Planificación del trabajo de fin de grado

1.3. Plan de trabajo

Medusa empezó su desarrollo en el mes de marzo, esto se debe a que estaba en otro trabajo de fin de grado al comienzo del curso, pero en el mes de marzo decidí salirme del proyecto ya que estaba más orientado a la investigación, y me costaba compaginarlo con las clases y mi actual trabajo. Así que decidí empezar un trabajo de fin de grado por mi cuenta orientado al desarrollo de una aplicación. Es por esto que las tareas se han tenido que condensar para poder cumplir con las fechas establecidas. En la figura 1.1 se puede apreciar la planificación que se ha seguido en el proyecto de Medusa, las tareas del diagrama son:

- **Análisis de requisitos** *(11 de marzo de 2021 a 18 de marzo de 2021)*: elección de las tecnologías clave para el desarrollo de Medusa. En esta fase del proyecto se decidió el uso de tecnologías como Vue.js [1] para el lado del cliente. Se decidió usar Vue.js en vez de *frameworks* similares como Angular [2] o React [3] por que Vue.js tiene una curva de aprendizaje más sencilla y por el sistema de componentes que utiliza.
- **Implementación** *(19 de marzo de 2021 a 15 de mayo de 2021)*: desarrollo de la aplicación web progresiva y el servidor. En esta tarea se ha invertido el mayor tiempo del proyecto. La figura 1.2 contiene una linea temporal con los hitos más importantes de la implementación de Medusa.
- **Despliegue** *(16 abril de 2021 a 24 abril de 2021)*: despliegue de Medusa para que pueda ser accedida de manera pública. Fue necesario la configuración de un dominio, además de la puesta en marcha del servidor y del cliente.
- **Memoria** *(16 de mayo de 2021 a 15 de junio de 2021*: elaboración de la memoria del proyecto donde se detalla el desarrollo.

1.4. Organización de la memoria

En el capítulo 3 se presentan y se explican todas las tecnologías y conceptos que forman parte de Medusa. En el siguiente capítulo, el 4, se aclara la organización de los sistemas que componen Medusa y cómo interactúan entre ellos. En el capítulo 5 se explica con detalle las principales tareas realizadas para la implementación del cliente en Medusa. El capítulo 6 es muy similar al anterior solo que se detalla todo lo relacionado con el servidor de Medusa. En el capítulo 7 se detalla todo lo necesario para que una web pueda ser aplicación web progresiva y también se explica la implementación de Medusa como PWA.

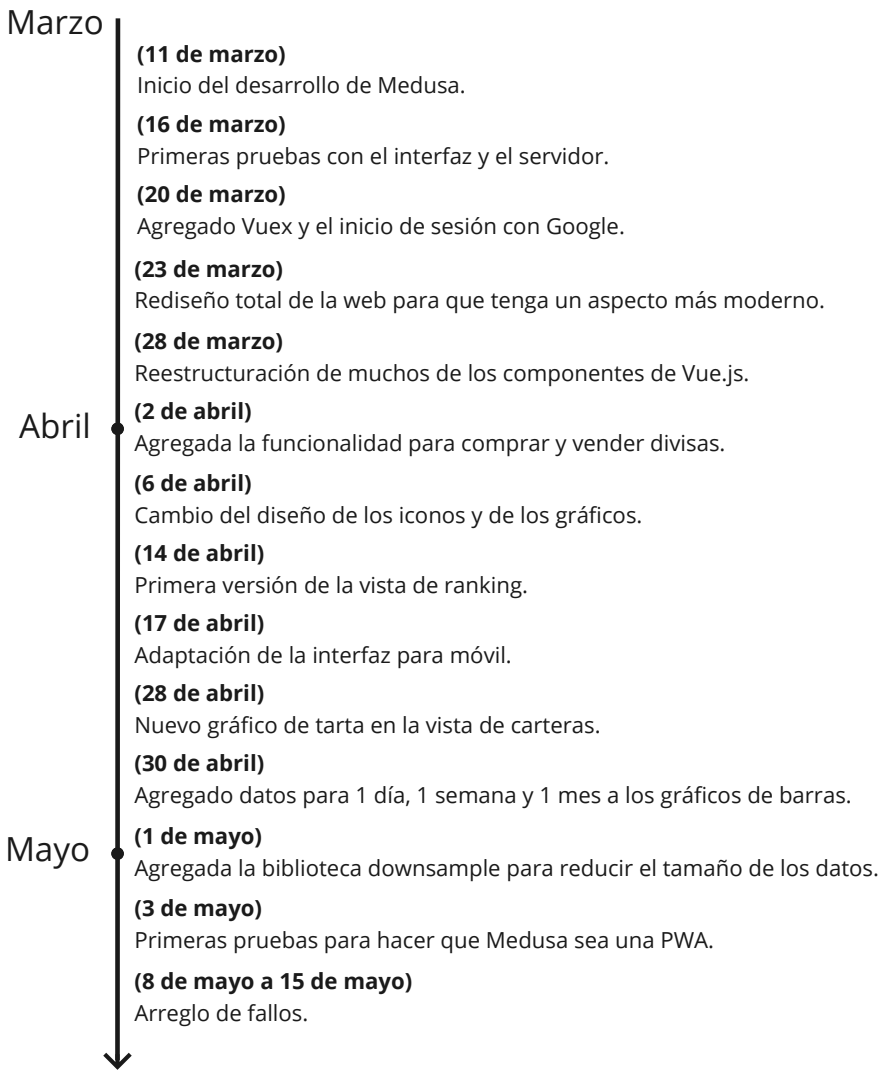

Figura 1.2: Hitos de la implementación de Medusa

En el capítulo 8 se especifica cómo ha sido el despliegue de Medusa y qué servicios se han utilizado. El capítulo 9 contiene un ejemplo paso a paso donde se muestra el uso de Medusa. Por último, el capítulo 10 contiene las conclusiones del desarrollo de Medusa y el trabajo futuro que se puede llegar a realizar.

Chapter 2

Introduction

This chapter will explain the motivation behind Medusa, the objectives and the work-plan that has been followed.

2.1. Motivation

Cryptocurrencies are an increasingly recurring topic of conversation nowadays, despite the fact that many people don't understand them or don't know how they work, they are interested in investing some money. Whenever the price rises or falls everyone is talking about it. The problem with cryptocurrencies is that the market is very volatile, which can cause investments made to plummet or triple in value. That's why it's very important to educate and allow people to simulate cryptocurrencies investments without any risk, Medusa was born from this idea.

2.2. Objectives

The main objectives of this project is to develop a web simulator for criptocurrency investments using modern technologies an libraries. The main goals are:

- Build a progressive web application with a modern and attractive design while being easy to use, so any user regardless of their technical level can use it.
- The web application must be a single page app, so the user experience will be comparable to that of a native application, since there are no refresh between pages.
- The communication between the server and the client must be in real time, which allows updating elements such as cryptocurrencies prices without user interaction.
- The registration and login should be done using a Google account, so that there is no need to implement a custom authentication system. It also encourages users to use the application because they don't need to register using a custom user and password.

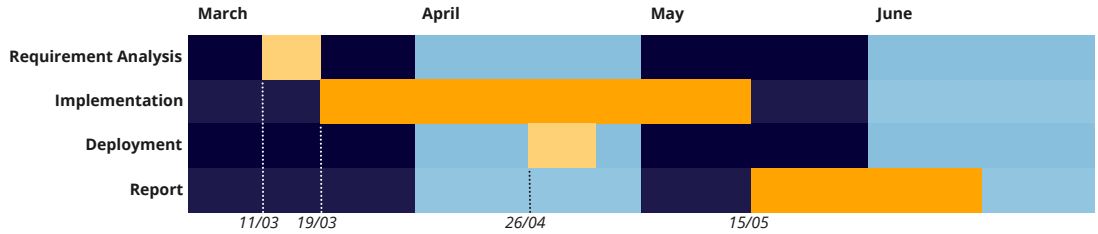

Figure 2.1: Medusa project planning

2.3. Workplan

Medusa development started in the month of March, this is because I was in another project at the beginning of the course, but in March I decided to abandon the project as it was more research oriented and it was extremely challenging for me to combine it with work and classes. The figure 2.1 shows the planning that has been followed in the project, the tasks in the diagram are:

- **Requirements analysis** *(March 11, 2021 to March 18, 2021)*: choice of the key technologies for the project. This is when technologies like Vue.js [1] where selected for the client-side. It was decided to use Vue.js instead of similar frameworks like Angular [2] or React [3], because Vue.js has an easier learning curve and because of the components system that it uses.
- **Implementation** *(March 19, 2021 to May 15, 2021)*: development of the progressive web application and the server. This task has been the most expensive in terms of time invested. The figure 2.2 contains a timeline of the key milestones of the implementation of Medusa.
- **Deployment** *(April 16, 2021 to April 24, 2021)*: deployment of Medusa so that it can be publicly accessed. A domain was required for this task, in addition to the configuration of the server and the client.
- **Report** *(May 16, 2021 to June 15, 2021)*: creation of the project report where the development of the project is detailed.

2.4. Content

In chapter 3 all the technologies and concepts that are part of Medusa are presented and explained. In the next chapter, chapter 4, clarifies the organization of the systems that make up Medusa and how they interact with each other. In chapter 5 the implementation of the client is explained in detail. Chapter 6 is very similar to the previous one but instead of the client, the server is explained. In chapter 7 all the requirements for a website to be a progressive web application are detailed and the implementation of Medusa as a progressive web app is also explained. Chapter 8 specifies how Medusa has been deployed and what services have been used for the deployment. Chapter 9 contains a step-by-step example showing the use of Medusa. Finally, chapter 11 contains the conclusion of the development of Medusa an the future work that may be done.

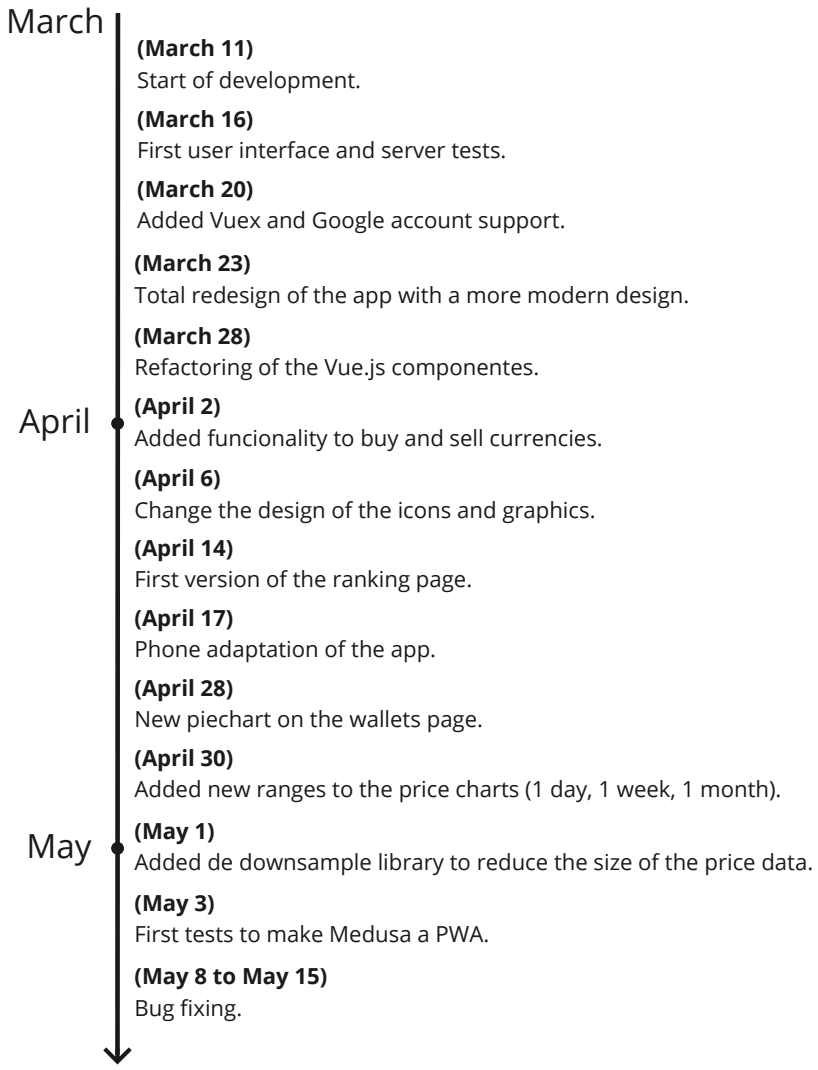

Figure 2.2: Milestones of the implementation of Medusa

Capítulo 3

Preliminares

En este capítulo vamos a profundizar en algunos conceptos y tecnologías necesarias para el funcionamiento de Medusa.

3.1. Criptodivisas

Las criptodivisas son un tipo de monedas digitales aseguradas usando técnicas criptográficas, lo que hace casi imposible poder hacer ataques de doble gasto [4]. Permiten el intercambio, la compra y la venta de las mismas, aunque hay muchas personas que las utilizan para la especulación, cada vez hay más sitios donde puedes utilizarlas para comprar productos y servicios. En total hay más de 10.000 criptodivisas distintas, cuyo valor total en abril de 2021 alcanzó los 2.2 billones de dólares.

La primera criptodivisa y la más popular es Bitcoin [5], fue creada por una persona (o grupo de personas) llamada Satoshi Nakamoto. Bitcoin está basada en lo que se conoce como cadenas de bloques. La gran mayoría de las criptodivisas utilizan cadenas de bloques para su funcionamiento, aunque hay algunas como IOTA [6] que no lo utilizan. Una cadena de bloques es un registro descentralizado, distribuido, y generalmente público que contiene registros que se conocen como bloques, estos bloques contienen información sobre cierto número de transacciones.

3.2. WebSocket

Tradicionalmente las páginas web han sido creadas usando HTTP y su paradigma solicitud/respuesta unidireccional, donde el cliente se encarga de realizar las peticiones, y el servidor se encarga de responder. Sin la interacción del usuario no se podía actualizar los datos de la web, para intentar solucionar este problema se introdujeron tecnologías como AJAX [7] que permitieron explorar la posibilidad de hacer que la conexión del cliente con el servidor fuera bidireccional.

Una de las técnicas más utilizadas con AJAX para hacer que la comunicación sea bidireccional, es usar lo conocido como sondeo largo [8] (*long polling en inglés*), donde la petición HTTP se mantiene abierta hasta que el servidor decide cerrarla respondiendo con datos nuevos, inmediatamente después el cliente lanza una nueva petición HTTP que se queda abierta y el ciclo se repite. El problema de técnicas como el sondeo largo es que arrastran la sobrecarga propia del protocolo HTTP. Cada vez que se realiza una petición HTTP se mandan las cabeceras y las *cookies*, esto puede sumar una cantidad grande de

datos que deben transferirse, lo que a su vez aumenta la latencia. Para solucionar todas estas limitaciones y problemas se creó el protocolo WebSocket.

WebSocket [9] es un protocolo de comunicación que permite crear conexiones persistentes y de baja latencia entre un cliente y un servidor. Proporciona un canal de comunicación bidireccional y *ifull-duplex* sobre un único socket TCP, permitiendo que tanto el cliente como el servidor puedan enviar datos en cualquier momento.

3.3. Tecnologías del cliente

El cliente de Medusa está desarrollado usando tecnologías modernas y vanguardistas, en esta sección voy a explicar las tecnologías más importantes del cliente.

3.3.1. Aplicación de página única (SPA)

Una aplicación de página única o SPA, es una aplicación web donde no es necesario recargar la página para navegar a través de las diferentes páginas. Una SPA bien configurada mejora la experiencia de usuario (UX) al no tener que mostrar páginas en blanco mientras el contenido se carga, ya que los ficheros necesarios para el funcionamiento de la web (HTML + CSS + JavaScript) se descargan la primera vez que se accede a la web. Como una SPA carga todo el contenido que va a usar en la primera carga, las conexiones posteriores al servidor se realizan usando tecnologías asíncronas como AJAX o WebSocket, y con JSON como formato para transmitir los datos. La diferencia del ciclo de vida de una web tradicional con una SPA se puede observar en la figura 3.1.

En las páginas web tradicionales lo más común es que el servidor tenga un controlador que combina las vistas con los datos del modelo para así generar el documento HTML con toda la información. En aplicaciones de página única, se puede desacoplar más fácilmente la vista del controlador, permitiendo que la vista esté en un servidor completamente distinto al modelo con los datos.

3.3.2. Service Worker

Un Service Worker [10] es un tipo específico de *script* de JavaScript, el cuál se ejecuta en segundo plano del navegador. Funciona como un proxy integrado en el navegador del cliente y que se localiza entre la página web y el servidor. Entre otras muchas cosas, una de las funcionalidades más importantes de los Service Worker es la de cachear ficheros y permitir el acceso a ellos aunque el cliente no tenga conexión a internet.

Los Service Workers son independientes y tienen un ciclo de vida propio. El *script* es registrado e instalado por el navegador, una vez que se ha realizado la instalación del Service Worker de una web puede empezar a cachear los ficheros. Los Service Workers no tienen acceso directo al DOM, por lo que para enviar y recibir información de los scripts propios de la web, usa un sistema de mensajes y eventos.

3.3.3. Aplicación web progresiva (PWA)

PWA [11] son las siglas de aplicación web progresiva, y son páginas web con algunas funcionalidades extra que no tienen las páginas webs tradicionales, esto permite proporcionar una experiencia de usuario muy similar a la de una aplicación nativa. Algunas de estas funcionalidades son:

- La posibilidad de instalar la página web como si fuera una aplicación nativa.

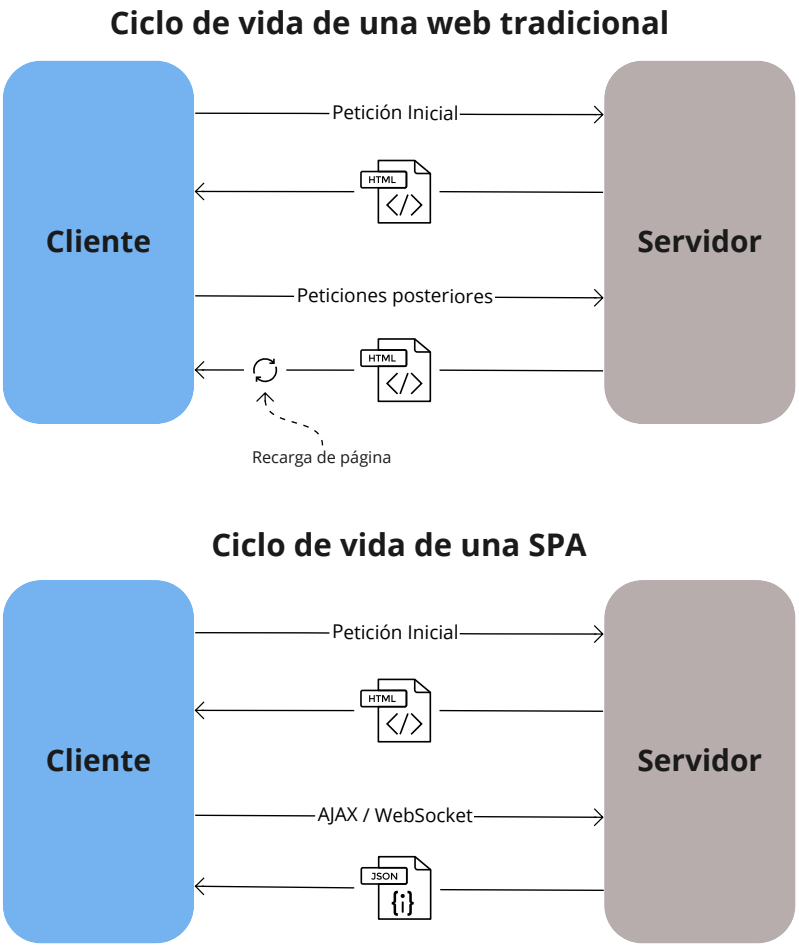

Figura 3.1: Comparación del ciclo de vida de una web tradicional con una SPA

- Proporcionar una página de error cuando no hay conexión a internet.
- Cachear ficheros para no tener que cargarlos y permitir el acceso a los mismos sin necesidad de internet.

Gracias a las aplicaciones web progresivas, la barrera de entrada para programar una aplicación que se asemeje a la experiencia que brindan las aplicaciones nativas se está reduciendo. En muchos aspectos las PWA están consideradas como el futuro del desarrollo web para dispositivos móviles.

3.3.4. Vue.js

Vue.js es un *framework* progresivo que te permite crear interfaces de usuario y aplicaciones de página única. Vue.js se conoce como *framework* progresivo porque está diseñado para ser fácil de integrar con otros *frameworks* y bibliotecas. La renderización de la interfaz de Vue.js es declarativa, es decir, las vistas dependen del estado y si el estado se modifica la vista también. Además, tiene un alto desacoplamiento permitiendo e incentivando reusar componentes de la interfaz. Gracias a Webpack [12] y a Vue Loader [13] los proyectos de Vue.js se compilan a ficheros de HTML, CSS y JavaScript, para que puedan ser accedidos

desde cualquier navegador. Los componentes en Vue.js conforman los elementos visuales de una página web junto con la lógica y el estado de ese elemento visual. Un componente se puede definir en un fichero con extensión `.vue`, el fichero tienen tres partes: JavaScript, plantilla y estilos.

3.3.4.1. JavaScript

Dentro de las etiquetas de *script* de un componente de Vue.js, se definen elementos como el nombre, el estado, las funciones que va a utilizar, etc. Las variables que se definan como estado del componente son lo que se denominan como reactivas, es decir, en el momento en el que el estado del componente se modifique, el interfaz se va a regenerar con los valores nuevos.

3.3.4.2. Plantilla

Dentro de las etiquetas `<template></template>` de un componente se define todo el HTML que va a utilizar para su interfaz. En el caso de que se quiera usar un componente dentro de otro, solo hay que incluir el subcomponente como una etiqueta HTML con el nombre del componente. Para facilitar la generación del HTML, Vue.js integra directivas que te permiten interactuar con el estado del componente. Todas las directivas se agregan como atributos a las etiquetas HTML. Por ejemplo, la directiva `v-if` [14] evalúa una condición y en caso de ser verdadera se renderiza la etiqueta a la que está asociada.

3.3.4.3. Estilos

Los estilos específicos de un componente se definen dentro de la etiqueta `<style scoped>`. El atributo `scoped` permite que todos los estilos CSS que se definan afecten únicamente al HTML del componente. Si un componente tiene subcomponentes de Vue.js, los estilos definidos con `scoped` en el componente padre no afectan a los subcomponentes.

3.4. Tecnologías del servidor

El servidor en Medusa gestiona el inicio de sesión/registro y todas las interacciones que tiene el usuario con la aplicación. En esta sección voy a explicar las tecnologías clave del servidor.

3.4.1. Node.js

Node.js [15] es un entorno de ejecución de JavaScript construido a partir del motor de JavaScript V8 [16] de Google. Node.js tiene una amplia variedad de bibliotecas para ayudar en el desarrollo de aplicaciones. Algunas de sus características principales son:

- Asíncrono y basado en eventos: todas las APIs de Node.js son asíncronas, es decir, sin bloqueo en la ejecución.
- Monohilo: Node.js usa un modelo de un solo subproceso con un bucle de eventos.
- Rápido: al utlizar el motor V8 de JavaScript, Node.js es muy rápido en ejecución de código.

Para poder gestionar las bibliotecas en Node.js se utiliza el gestor de paquetes npm [17]. Con npm, solo necesitas definir un fichero `package.json` con las bibliotecas que deseas utilizar en el proyecto y al ejecutar `npm install` se descargan todos los elementos en la carpeta `node_modules`.

3.4.2. MongoDB

MonoDB [18] es una base de datos NoSQL orientada a documentos. En vez de utilizar tablas y filas como en las bases de datos relaciones, MongoDB utiliza colecciones y documentos. Los documentos consisten en pares clave-valor, que es la unidad de datos básica de MongoDB, por otro lado, las colecciones contienen conjuntos de documentos. Uno de los puntos fuertes de MongoDB es que te permite almacenar datos sin definir un esquema ya que los datos se almacenan en formato JSON.

3.5. Seguridad y autenticación

En esta sección se va a comentar algunos conceptos relacionados con la seguridad y la autenticación de Medusa, como JWT que es el estándar que implementa Medusa para poder iniciar sesión.

3.5.1. JSON Web Token (JWT)

JSON Web Token es un estándar abierto (RFC 71519 [19]) que define una forma segura para transmitir información en formato JSON entre varias partes. Toda la información enviada puede ser verificada y fiable porque está firmada digitalmente. Los JWTs pueden ser firmados usando un secreto (utilizando el algoritmo HMAC) o con un par de claves pública/privada usando RSA o ECDSA. En la figura 3.2 se puede observar la estructura de un JWT, tiene tres partes codificadas en Base64 separadas por puntos, estas partes son:

1. **Cabecera**: la cabecera tiene dos partes, el tipo de token (que es JWT) y el algoritmo que se ha utilizado para firmar el token.
2. **Carga útil**: contiene toda la información que se quiera transmitir en formato JSON.
3. **Firma**: para crear la firma hay que usar la cabecera codificada (en Base64) y la carga útil separadas por un punto. Se codifica usando el algoritmo especificado en la cabecera usando un secreto.

Cómo la firma se calcula usando la cabecera y la carga útil, es fácil detectar si alguien ha alterado algún elemento porque la firma no va a ser la misma que la del JWT.

3.5.2. Google OAuth 2.0

La API de Google para realizar el inicio de sesión en webs externas utiliza el protocolo OAuth 2.0 [20]. Para poder utilizar la API de Google OAuth 2.0 es necesario registrarse como desarrollador en Google Developer Console [21]. Al registrarse es necesario definir a qué información del usuario de la cuenta de Google necesita acceder tu aplicación. Dependiendo de los datos que quieras acceder, se necesita pedir permiso al usuario. Medusa solo necesita verificar que la cuenta de Google es correcta, por lo que no requiere permiso del usuario para poder acceder a los datos de la cuenta.

Figura 3.2: JWT y decodificación de los tres elementos que lo componen

Todas las cuentas de Google tienen un identificador único. Cuando se inicia sesión usando la API de Google el identificador es accesible. En el caso de Medusa el ID de Google es utilizado para poder contrastar con la base de datos el usuario, en el caso de que no exista se inicia el proceso de registro de un usuario nuevo.

3.5.3. Intercambio de recursos de origen cruzado

El intercambio de recursos de origen cruzado (CORS) [22] es un mecanismo del navegador que habilita a una página web el acceso controlado a recursos que están ubicados fuera del dominio actual. La configuración de CORS se manda desde el servidor al cliente en las cabeceras HTTP. Las peticiones que utilizan CORS son:

- Peticiones AJAX.
- Fuentes web.
- Texturas WebGL.
- Imágenes dibujadas en una etiqueta `canvas` usando el método `drawImage`.
- Hojas de estilo (para acceso CSSOM).

- Scripts (para excepciones inmutadas).

En todas las peticiones del listado anterior, el navegador por defecto bloquea el acceso a los recursos en el caso de que estén en un origen distinto, por origen distinto se entiende: un esquema diferente (HTTP o HTTPS), un dominio diferente (incluye los subdominios) y un puerto diferente. Para ponerlo en contexto con un ejemplo, si un *script* de la web `ejemplo.com` quiere acceder a una fuente alojada en `prueba.ejemplo.com`, es necesario que `prueba.ejemplo.com` tenga configurada las cabeceras CORS correctamente para permitir el acceso a `ejemplo.com`.

Capítulo 4

Organización del sistema

Los componentes principales de Medusa son el cliente y el servidor, que se comunican entre ellos usando dos protocolos: HTTP y *WebSockets*. La autenticación se realiza usando la API *OAuth 2.0* de Google. La API que el proyecto usa para obtener los precios de las divisas en tiempo real, es la de Coingecko. Por último, la aplicación utiliza MongoDB para almecenar los datos relativos a los usuarios.

En la figura 4.1 se puede observar el mapa de sistemas de Medusa. El cliente interactúa tanto con el propio servidor de Medusa, como con la API de Google para realizar el inicio de sesión. Por otro lado, el servidor de Medusa (todos los elementos dentro del rectángulo negro), interactúa con la base de datos, con la API de Coingecko y con la API de Google para verificar el inicio de sesión. A continuación, detallaremos la interacción entre componentes que se produce en las dos tareas que utilizan APIs externas: el proceso de autenticación y la obtención de precios de las distintas divisas.

4.1. Autenticación

Como se ha mencionado anteriormente, toda la autenticación se hace utilizando la API de Google *OAuth 2.0*. Esta decisión se tomó al principio del desarrollo por dos razones principalmente:

1. Agilizar el desarrollo al no tener que hacer un sistema de inicio de sesión propio, donde tener que almacenar las contraseñas, enviar correos de confirmación, reseteos de contraseña, etc.

2. Incentivar su uso al no tener que registrarte con tu correo, lo que consigue que más usuarios se acaben registrando [23].

El proceso de autenticación de Medusa interactúa con dos sistemas, primero con Google para verificar que la cuenta es válida, y segundo con la propia base de datos de Medusa para verificar si el usuario existe. El proceso de inicio de sesión/registro se puede observar en la figura 4.2 y tiene los siguientes pasos:

1. El cliente inicia sesión usando su cuenta de Google. Google devuelve al cliente un token de acceso.

2. El token de acceso es mandado al servidor por el cliente.

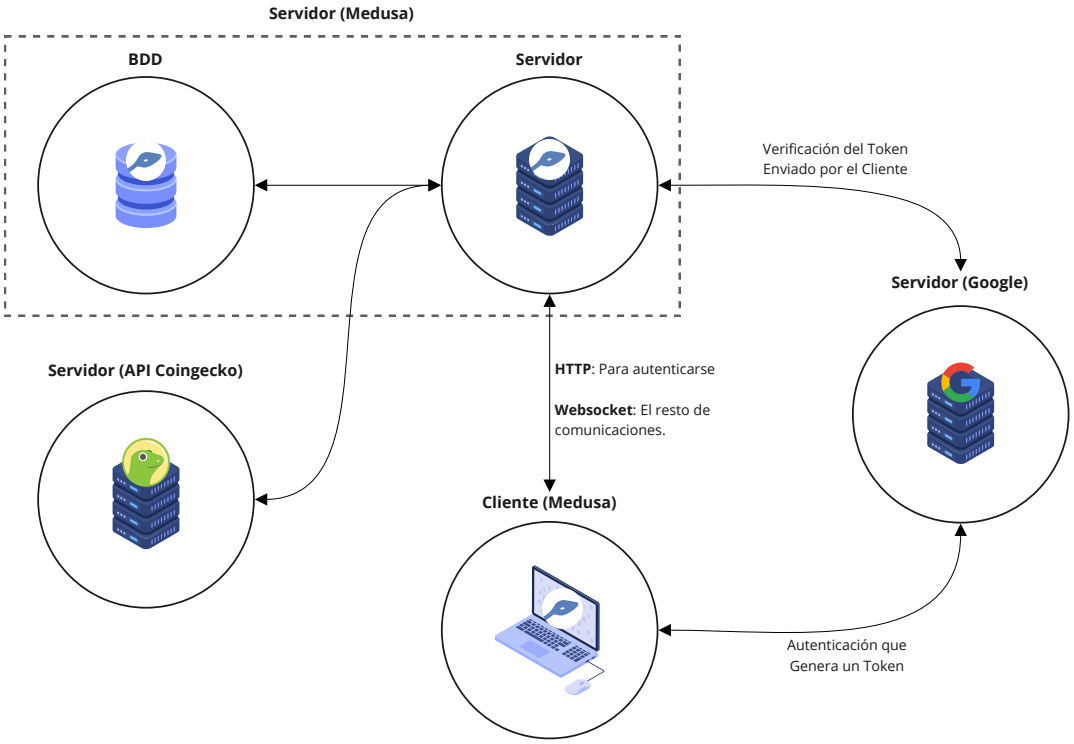

Figura 4.1: Mapa de sistemas de Medusa

3. El servidor verifica que el token de acceso es válido y comprueba si el ID de Google ya está registrado en Medusa.

4-a. En el caso de que el ID de Google exista, el servidor genera un JWT con el campo `registrado` como `true`, que manda al cliente.

4-b. Si no existe el ID de Google, el servidor manda al cliente un JWT con el campo `registrado` como `false` para que se proceda con el registro.

4-b1. El cliente manda el JWT generado anteriormente junto con el nombre de usuario elegido.

4-b2 Si el nombre de cliente es válido, el servidor manda al cliente un JWT con el campo `registrado` a `true` para que realice el inicio de sesión.

5. El cliente se conecta al servidor mediante un *WebSocket* usando el JWT generado por el servidor.

4.2. Precios de las divisas

Un elemento crucial para el funcionamiento de Medusa es disponer del precio de las divisas en tiempo real. Como puedes comprar y vender divisas y tiene que reflejarse en tu cuenta, es importante que el servidor y el cliente tengan el precio de las divisas sincronizado en todo momento. El servidor lo necesita para verificar que se puede realizar una compra de la divisa, y el cliente para que el usuario pueda consultar los precios y decidir saber qué divisas comprar y vender.

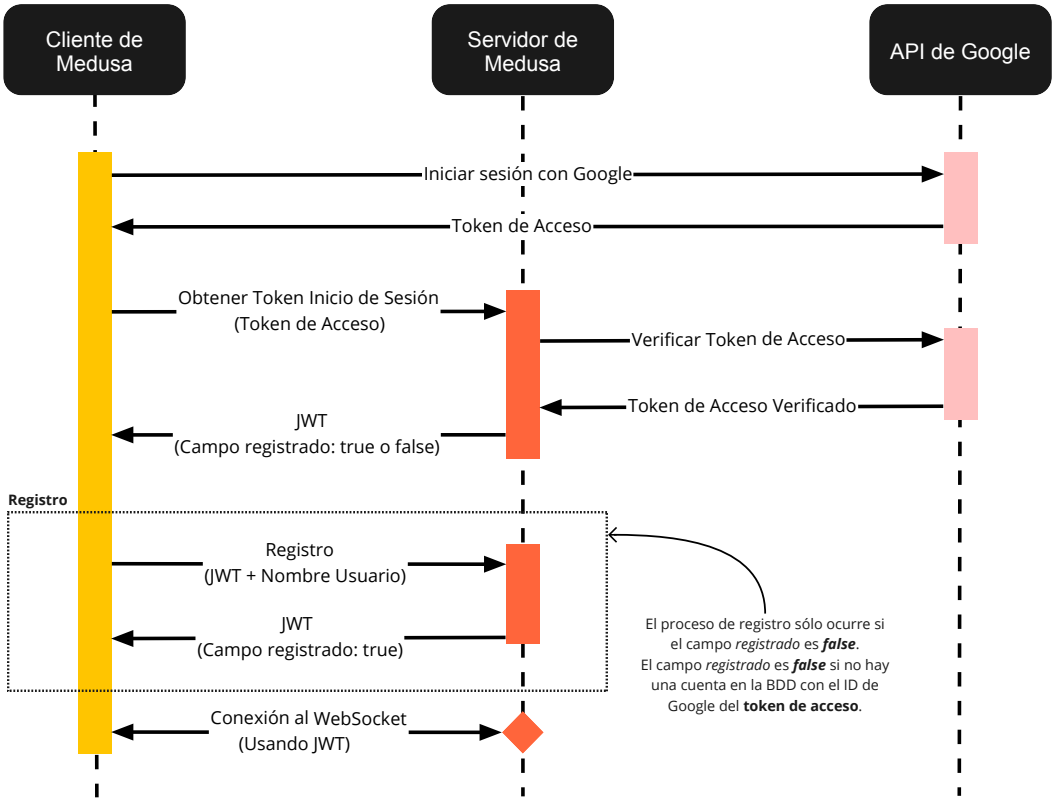

Figura 4.2: Diagrama del proceso para iniciar sesión y registro de Medusa

Para que funcione todo este sistema era importante encontrar una API robusta y estable y que se adaptara a mis necesidades. Finalmente me decidí por la API de Coingecko [24]. Como tiene una biblioteca de JavaScript [25], la implementación en el servidor fue muy sencilla. Uno de los puntos positivos de usar la API de Coingecko es que no necesitas ningún tipo de clave para poder usar su servicio. La única limitación es que no puedes hacer más de 10 llamadas al segundo por dirección IP [26]. Coingecko además opera el 99,78 % del tiempo sin caídas [27].

La API es llamada cada 30 segundos para poder refrescar los precios, si detecta que hay precios nuevos, primero los almacena para los nuevos clientes que se conecten, y luego los manda a los clientes que están conectados en ese momento al servidor mediante los *WebSockets*.

Capítulo 5

Detalles del cliente

En este capítulo se detalla todo lo relacionado con el cliente de Medusa. El objetivo al desarrollar la interfaz de Medusa era que fuera lo más intuitiva posible. La figura 5.1 es el mapa web de Medusa, donde las dos únicas páginas accesibles sin haber iniciado sesión son: *Login* y *Registro*. En el caso de que un usuario inicie sesión por primera vez, debe completar el registro.

5.1. Creación de la interfaz usando Vue.js

Uno de los principales objetivos al realizar Medusa fue el de crear una aplicación de página única (*SPA*). Para facilitar el desarrollo de una SPA se ha utilizado el *framework* de Vue.js. Adicionalmente, existen algunas bibliotecas oficiales que brindan funcionalidad extra a un proyecto de Vue.js. Las dos bibliotecas que se han integrado en Medusa son: Vue Router [28] y Vuex [29].

5.1.1. Vue Router

Como Medusa es una SPA, es importante implementar un sistema que te permita navegar entre los distintos componentes. Gracias a Vue Router puedes definir todas las rutas de la web y enlazarlas a los distintos componentes de Vue.js. En la tabla 5.1 se pueden apreciar las rutas principales de Medusa con el componente de Vue.js al que se han enlazado. Una de las ventajas de usar Vue Router es que en el momento en el que se navega a otro componente, la URL de la barra del navegador cambiará, permitiendo por ejemplo poder pulsar atrás en el navegador para volver al componente anterior. Esto permite que la experiencia de usuario sea la misma que al usar una página web tradicional.

Vue Router te permite definir una ruta comodín para cuando la URL introducida no exista. En el caso de Medusa, esa ruta se enlaza con el componente de `Error404.vue`, el cual contiene una página de error 404 con el diseño de Medusa. Para que Vue Router funcione correctamente, es importante configurar el servidor del cliente para que todas las peticiones se redirijan al fichero `index.html`.

5.1.2. Vuex

Como se ha explicado en la sección 3.3.4 de los preliminares, Vue.js es un *framework* progresivo que implementa un renderizado progresivo de los componentes. Para ello, cada componente de Vue.js tiene asociado un estado. El estado de un componente como

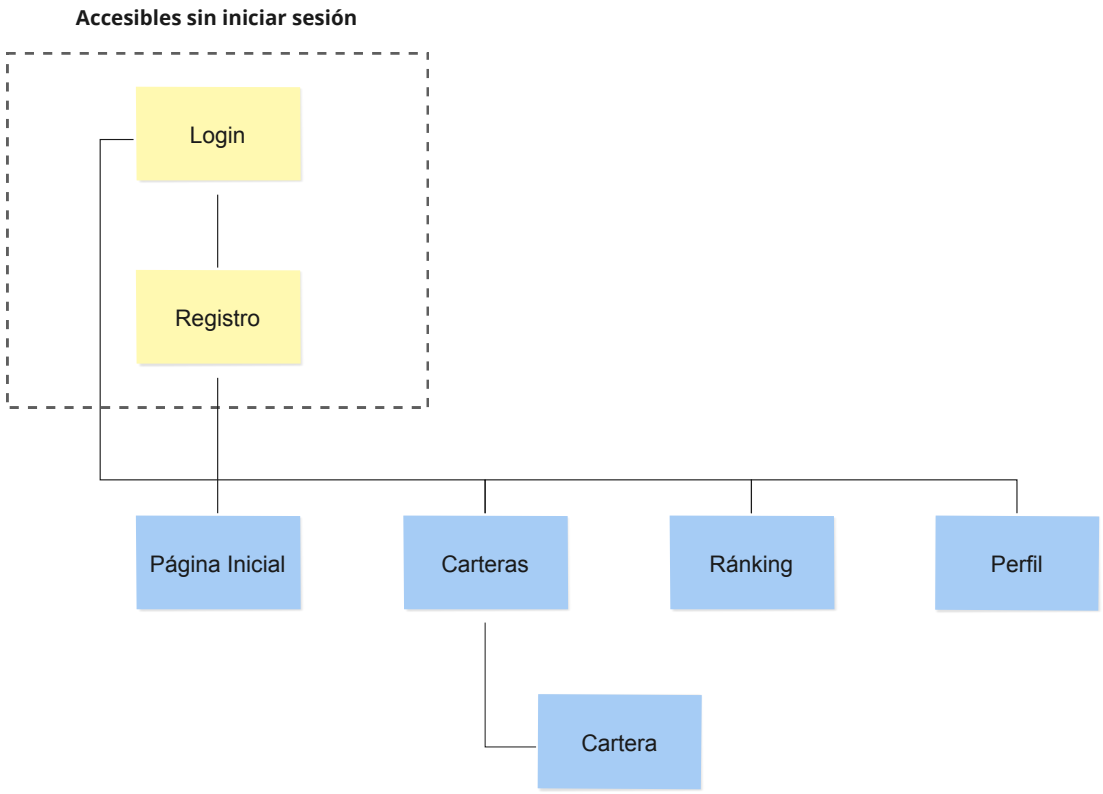

Figura 5.1: Mapa de la web de Medusa

`Cartera.vue` es una variable que contiene los datos de la cartera que se esté viendo. La complejidad aumenta cuando existen muchos componentes que comparten estado, en este caso, hay que desarrollar un sistema en cascada donde el componente padre tiene el estado y lo va compartiendo con los componentes hijos. En el caso de que uno de los subcomponentes realice una modificación del estado, se lo tiene que comunicar al componente padre a través de un evento para que pueda actualizar el estado y a su vez a los otros subcomponentes. Hacerlo de esta forma implica aumentar la complejidad del código, y para evitarlo los desarrolladores de Vue.js crearon Vuex.

Vuex está diseñado usando el patrón de diseño de Flux [30]. Vuex es una biblioteca para administrar el estado de la aplicación permitiendo concentrar el estado que comparten los componentes en lo que se denomina un almacén. El almacén es accesible desde cualquier componente, además, es reactivo, lo que quiere decir que en el momento en el que un componente modifique el estado del almacén, el resto de componentes que usan ese estado van a ser notificados de manera automática recibiendo el valor actualizado. El almacén de Vuex está dividido en:

- `state`: contiene el estado del almacén. En el extracto de código 5.1 se puede observar la definición del estado para el almacén de Vuex en Medusa. Hay valores como el de `conectado` que permite a los componentes saber si el cliente está conectado al *socket*.
- `mutations`: las mutaciones son las funciones que modifican el estado del almacén.

Ruta	Componente
/	Login.vue
/registro	Registro.vue
/inicio	Inicio.vue
/carteras	Carteras.vue
/cartera/:id	Cartera.vue
/cartera-fiat	CarteraFiat.vue
/ranking	Ranking.vue
/perfil	Perfil.vue
*	Error404.vue

Tabla 5.1: Rutas y a que componente de Medusa pertenecen.

```
{

    usuario: {
        id_google: '',
        nombre: '',
        resets: 0,
        fecha_registro: '',
        cartera: {...}
    },

    ranking: [],

    divisas: {...},

    conectado: false,
    tema: "claro",
    cargando: false
}
```

Código 5.1: Estado del almacén de Medusa.

Solo son llamadas desde las acciones del almacén.

- `actions`: las acciones son funciones que se llaman desde el componente. Se ejecutan siempre de forma asíncrona, por lo tanto, es aquí donde se llama por ejemplo a una API si queremos usar su resultado en el almacén. Para que una acción modifique el estado del almacén tiene que usar una función definida como mutación.

- `getters`: son funciones que devuelven el estado del almacén. Los `getters` son las funciones que se usan en los componentes para poder acceder al estado del almacén.

En la figura 5.2 se puede observar un esquema con todos los componentes del almacén de Vuex. Un componente de Vue.js interactúa solo con las acciones y los *getters*.

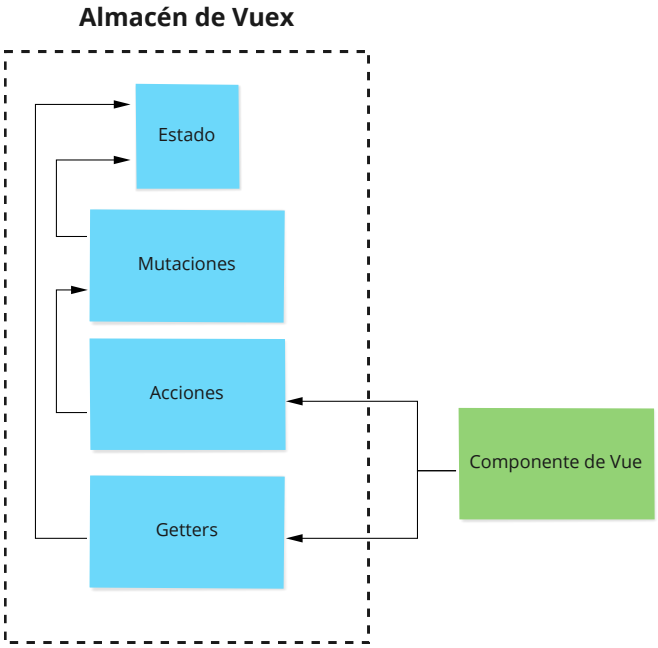

Figura 5.2: Elementos del almacén de Vuex.

5.2. Socket.io

Socket.io es la biblioteca que se ha utilizado para la implementación de la comunicación a través de WebSockets. Uno de los aspectos más importantes de la implementación de Socket.io en el cliente es el uso de una biblioteca auxiliar a Socket.io que se llama Vue-Socket.io [31].

Vue-Socket.io facilita mucho el uso de Socket.io con un proyecto de Vue.js que use Vuex [32]. Cuando se recibe un mensaje a través del *socket*, Vue-Socket.io comprueba si hay una mutación o una acción definida en el almacén de Vuex que tenga el nombre del mensaje, en caso de que exista la función, la ejecuta. Las acciones y las mutaciones que se quieran ejecutar a través de Vue-Socket.io deben de tener en el nombre un prefijo idetificativo, en el caso de Medusa el prefijo `SOCKET_`.

Como se ha visto anteriormente, cuando por ejemplo se reciba un mensaje con precios nuevos, la mutación va a modificar el estado del almacén de Vuex, haciendo que todos los componentes de Vue.js que usan el estado con los precios se actualicen automáticamente. En el almacén de Medusa solo se han definido mutaciones para Vue-Socket.io. Las mutaciones que se han definido son:

- `SOCKET_TRANSACCION`: recibe transacciones y actualiza las carteras del usuario con los nuevos balances. Esta mutación es llamada únicamente cuando un usuario tiene varias conexiones simultáneas y una de ellas ha realizado una transacción nueva. Se trata de una mutación para mantener todas las conexiones de un mismo cliente sincronizadas.
- `SOCKET_INICIO`: esta mutación ocurre cuando el cliente se conecta al *socket* por primera vez. Actualiza el estado del almacén con los datos del usuario y de las divisas.

- `SOCKET_DIVISAS`: actualiza los precios de todas las divisas. Solo se ejecuta en el caso de que el cliente se conecte al servidor cuando se acaba de reiniciar, por lo que todavía no tendrá los precios de las divisas (ya que se está esperando a la respuesta de la API).
- `SOCKET_PRECIO`: actualiza los precios de las divisas. La mutación suele ser llamada cada 30 segundos si el servidor tiene precios nuevos para las divisas.

Además de permitirte enlazar mutaciones, Vue-Socket.io permite asociar eventos de mensajes a componentes de Vue [33]. El fichero `App.vue` (el fichero raíz del proyecto de Vue) escucha dos eventos de Socket.io:

- `version`: recibe del servidor la versión actual y la compara con la última que ha recibido. En caso de que sea una versión nueva muestra una ventana emergente para refrescar la página.
- `desconectar`: cierra la sesión y carga la página de inicio de sesión.

5.3. Gráficos de información

Los gráficos de líneas y de tarta están realizados usando la biblioteca de Chart.js [34]. El gráfico de líneas se usa para mostrar el histórico de los precios, mientras que el de tarta se usa en la vista de carteras para poder saber qué porcentaje se tiene de cada divisa. En ambos casos, para que Chart.js represente los datos de forma gráfica tienes que utilizar un vector que contenga los valores a representar.

La biblioteca vue-chartjs [35] se ha utilizado para facilitar la implementación de los gráficos de Chart.js en Vue.js. Una de las funcionalidades más importantes de vue-chartjs es la posibilidad de hacer que los gráficos sean reactivos, por ejemplo, en el momento en el que exista un nuevo precio, el gráfico se va a actualizar automáticamente. Por defecto, todos los gráficos tienen animaciones cuando los datos se eliminan o modifican.

Los gráficos de líneas con los precios soportan 3 franjas de tiempo: 24 horas, 7 días y 30 días. Con tres botones se puede cambiar la franja que está usando el gráfico en ese momento. El problema que nos encontramos es que como cada franja horaria tiene asociado un vector con distinto número de precios, la animación no funciona correctamente y tarda mucho en responder. Para hacer funcionar correctamente la animación, hay que reducir los tres vectores de precios de cada divisa para que tengan el mismo número de elementos. La biblioteca Downsample [36] permite reducir un vector antes de visualizarlo sin perder las características visuales de los datos. Downsample ofrece varios algoritmos de reducción, en Medusa para todos los precios se aplica el algoritmo LTTB [37] con una resolución objetivo de 190 elementos. En la figura 5.3 se puede ver la diferencia en los gráficos de un dataset al aplicar LTTB, en la que se aprecia cómo con 190 elementos se mantiene una representación fidedigna de la gráfica original.

5.4. Imágenes de perfil

Las imágenes de perfil de los usuarios de Medusa son generadas a partir de sus nombres, el diseño se inspira en los Identicons de Github [38]. No se permite subir al usuario una fotografía por dos razones:

1. Evitar tener que almacenar las imágenes en el servidor.

2. No tener que moderar/supervisar las imágenes de los usuarios.

Las imágenes de usuario consisten en una matriz de 6x6 donde cada cuadrado puede estar o no visible. El color de los cuadrados su visibilidad se calcula usando un generador de números pseudo-aleatorios. El generador de números pesudo-aleatorios usa como semilla el nombre de usuario, gracias a esto, cada nombre de usuario va a generar siempre la misma imagen. Como JavaScript no permite alimentar con una semilla la función `Math.random()` [39], he utilizado la biblioteca Seedrandom.js [40]. Los cuadrados se pintan en un elemento *canvas* de HTML, del cual se extrae una URI [41] que contiene toda la información sobre la imagen, permitiendo su visualización.

Como el color de fondo de las imágenes de perfil es blanco, hay algunos colores que no se distinguen correctamente. Por lo tanto, al generarse los colores aleatoriamente, es importante verificar que el color tiene suficiente contraste con el fondo. Para saber si un color RGB tiene suficiente contraste, se convierte al espacio de colores YIQ [42], que es el espacio usado por el estándar de televisión de NTSC. Lo interesante de este espacio de colores es que la "Y" es el componente que representa la información de luminancia. La luminancia es un número de 0 a 255 que representa la luminosidad de la imagen. Utilizando el valor "Y" únicamente es necesario verificar que no es mayor de 200, en el caso de que sea mayor se vuelve a generar otro color que no sea tan claro.

Gráfico sin reducción (~700 elementos)

Gráfico con reducción LTTB a 190 elementos

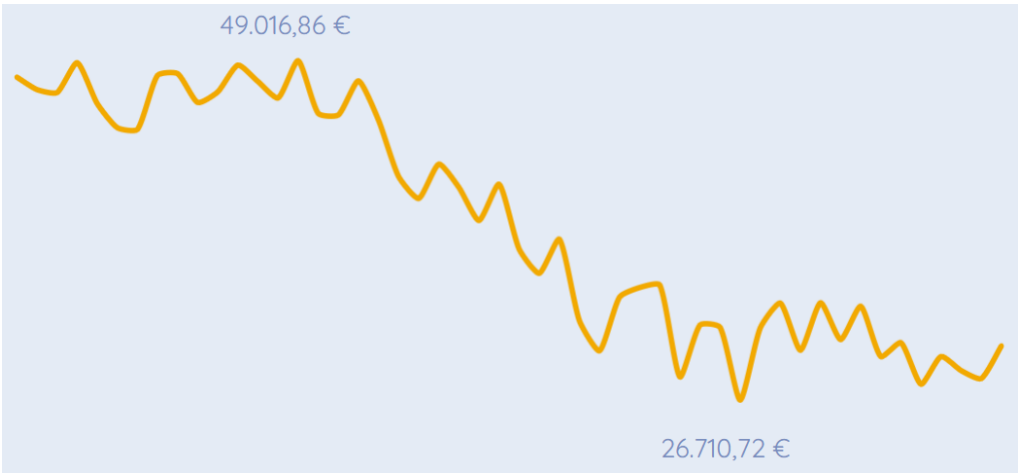

Gráfico con reducción LTTB a 50 elementos

Figura 5.3: Comparación del LTTB.

Capítulo 6

Detalles del servidor

El servidor de Medusa está implementado usando Node.js. Las peticiones HTTP se procesan gracias a la biblioteca de Express [43], por otro lado la comunicación con el cliente a través de los WebSockets se ha realizado usando la biblioteca de Socket.io [44]. Una de las ventajas de que el servidor este implementado en Node.js, es que como el cliente también está programado en JavaScript, se puede usar JSON para enviar los datos sin necesidad de librerías externas para procesar el JSON. El servidor tiene la estructura de ficheros de la figura 6.1. A continuación, comentaremos los elementos más importantes del servidor.

6.1. Manejo de la base de datos

Como se ha mencionado en los capítulos anteriores, la base de datos que se ha usado es MongoDB. Para acceder a la base de datos desde Node.js, se ha utilizado la biblioteca de Mongoose [45]. Mongoose es lo que se denomina como ODM (*Object-Document Mapper*), permite definir esquemas para modelar los datos que se vayan a usar con MongoDB. Mongoose se asemeja mucho a lo que es un ORM (*Object-Relational Mapper*) [46] usado en bases de datos relacionales. Uno de los beneficios de usar un ODM es que integra muchas funciones básicas para trabajar con MongoDB, lo que te permite ahorrar tiempo al no tener que escribir directamente sentencias para la base de datos.

Para definir un esquema con Mongoose, se debe crear un objeto de JavaScript que defina los atributos que va a tener. Hay dos formas de definir un atributo:

1. Escribir directamente el tipo de atributo que va a ser (*ej:* `nombre: String`).
2. Usar `type` (*ej:* `nombre: { type: String }`).

La segunda forma es la recomendada si quieres que el atributo tenga un valor por defecto, o si necesitas validarlo. En Medusa se han definido dos esquemas para la base de datos: *Usuario* y *Eliminado*.

6.1.1. Esquema de Usuario

Este es el esquema principal de la aplicación. Contiene toda la información sobre el usuario y sus carteras. En el extracto de código 6.1 se puede observar la definición del esquema Usuario. El atributo `resets` es el número de veces que el usuario ha reiniciado

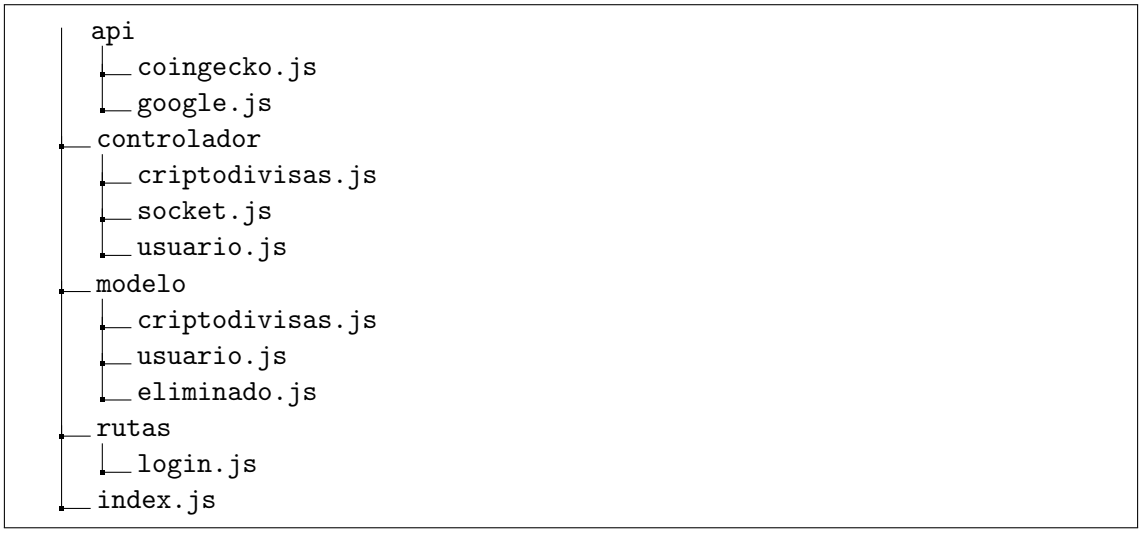

Figura 6.1: Estructura del servidor de Medusa

o borrado su cuenta. El atributo `cartera` se trata de un objeto donde cada atributo es una cartera distinta. Además de las criptodivisas existe la cartera de `fiat`, que contiene los euros del usuario. Cuando se crea un nuevo usuario o se *resetea* uno actual, el valor de la cartera de `fiat` es de 10.000,00 €, y el resto de 0,00 €.

El extracto de código 6.2 contiene la definición del atributo `cartera` del esquema de Usuario. Como se ha mencionado anteriormente, el atributo cartera contiene todas las carteras del usuario (8 carteras de criptodivisas y 1 de fiat). Cada cartera tiene como atributos: la cantidad y un vector con las transacciones.

Las transacciones se definen como en la referencia de código 6.3. El atributo `tipo` es un enumerado con dos valores (compra y venta), de esta forma, en el momento de la inserción de una nueva transacción Mongoose puede realizar una validación. Un atributo que es utilizado solo en las transacciones de la cartera de `fiat` es el de `detalles`, cuyo valor es el nombre de la criptodivisa implicada en la transacción. Por ejemplo, en el caso de una comprar 100,00 € de Bitcoin, se añade una nueva transacción de tipo venta a la cartera de `fiat` con el valor de "bitcoin" en el atributo `detalles`.

6.1.2. Esquema de Eliminado

Cuando un usuario borra su cuenta, se agrega a la colección de eliminados. El extracto de código 6.4 es la definición del esquema de Eliminado. Los atributos que tiene son el ID de Google y el número de *resets* que tiene el usuario. De esta manera si un usuario vuelve a crear su cuenta con el mismo ID de Google, se restaura su número de *resets*.

6.2. Express

Express es la biblioteca que se ha usado para procesar las peticiones HTTP que recibe el servidor. Se han integrado dos bibliotecas auxiliares a Express:

- **helmet**: permite mejorar la seguridad de una aplicación que usa Express, gracias a la configuración de algunas cabeceras HTTP. En total configura 11 cabeceras HTTP a través de *middlewares* de Express. Algunas de estas cabeceras son:

```
{
    id_google: {
        type: String,
        required: true
    },
    nombre: {
        type: String,
        required: true
    },
    email: {
        type: String,
        required: true
    },
    fecha_registro: {
        type: Date,
        default: Date.now
    },
    resets: {
        type: Number,
        default: 0
    },
    cartera: {...}
}
```

Código 6.1: Definición del esquema de Usuario

```
   cartera: {
      bitcoin: {
         cantidad: Number,
         transacciones: [...]
      },

      ...

      fiat: {
         cantidad: Number,
         transacciones: [...]
      }

   }
```

Código 6.2: Definición del atributo “cartera” perteneciente al esquema de Usuario

```
{
    _id: false,
    cantidad: Number,
    fecha: {
        type: Date,
        default: Date.now
    },
    precio: Number,
    tipo: {
        type: String,
        enum: ['compra', 'venta'],
        default: 'compra'
    },
    comision: {
        type: Number,
        default: 0
    },
    detalles: String
}
```

Código 6.3: Definición de un transacción perteneciente al atributo "transacciones" de una cartera.

```
{
    id_google: {
        type: String,
        required: true
    },
    resets: {
        type: Number,
        default: 0
    },
}
```

Código 6.4: Definición del esquema de Eliminado.

  - `Content-Security-Policy`: se configura para mitigar ataques de cross-site scripting.
  - `Strict-Transport-Security`: esta cabecera permite a un navegador priorizar el uso de HTTPS en el caso de que la conexión se haya realizado a través de HTTP.

- **body-parser**: permite parsear los cuerpos de las peticiones HTTP entrantes. Se usa para poder acceder a los datos que se mandan desde el cliente en una petición POST.

En el manejador de rutas para la URI `/login` se han definido dos rutas POST: `/google` y `/registro`.

6.2.1. Ruta /login/google

Esta es la ruta donde se inicia sesión usando Google. Recibe un objeto llamado `google` que tiene todos los datos necesarios para verificar que el inicio de sesión se ha realizado correctamente en el cliente usando una cuenta de Google. Una vez comprobado que el inicio de sesión y los datos son correctos, comprueba si en la base de datos existe el ID de Google de la cuenta. El servidor genera un JWT con los siguientes datos:

- `email`: el email de la cuenta de Google que se ha utilizado para iniciar sesión.
- `id`: el identificador de la cuenta de Google.
- `registrado`: si el usuario ya está registrado en Medusa o no (*puede ser true o false*).

6.2.2. Ruta /login/registro

En esta ruta se hace el registro de un usuario de Medusa. Recibe dos valores por POST: `nombre` y `token`. El valor del campo `nombre` contiene el nombre de usuario, y el campo `token` contiene el JWT que se generó al intentar hacer inicio de sesión. Para que el registro se complete es necesario que:

1. El JWT enviado sea válido.
2. El campo `registrado` del *token* tenga el valor `false`.
3. El nombre de usuario solo tenga letras y números, y una longitud de 5 a 14 caracteres.
4. No haya un usuario de Medusa con el mismo nombre.
5. No exista otro usuario con el mismo ID de la cuenta de Google.

Antes de crear el usuario, también se comprueba si el ID de Google está en la colección de eliminados, esto es necesario en el caso de que el usuario ya tuviera una cuenta anteriormente y así poder cargar el número de *resets* que tenía. Finalmente, el servidor devuelve un JWT al cliente para que pueda realizar la conexión al servidor mediante los WebSockets.

6.3. Comunicación con el cliente usando Socket.io

Para realizar la comunicación con el cliente a través de WebSockets se ha utilizado la biblioteca de Socket.io [44]. Como es necesario verificar que la conexión al *socket* se está realizando con una persona que ha iniciado sesión, se ha integrado la biblioteca socketio-jwt [47]. De esta forma, si en la conexión inicial al *socket* el cliente no adjunta un JWT válido se rechaza la conexión.

Socket.io utiliza un sistema de mensajes donde puedes escuchar o enviar mensajes. El servidor puede emitir un mensaje a:

- Un *socket* específico.
- Todos los *sockets*.
- Los *sockets* de una sala (*en inglés: room*). Para enviar un mensaje a una sala, es necesario saber su nombre.

Las salas te permiten dividir a los *sockets* en grupos. Cuando un nuevo cliente se conecta a Medusa, su *socket* se une a una sala que tiene como nombre el `_id` del documento con sus datos de MongoDB. Esto se hace para que en el caso de que haya varias conexiones del mismo usuario, las acciones se puedan propagar. Es decir, si un usuario está conectado desde dos pestañas del navegador y en una hace una compra de una divisa, se va a reflejar en tiempo real en la otra pestaña.

Como el *token* tiene el ID del usuario, en el momento de la conexión se accede a la base de datos para cargar todos los datos del usuario y enviarlos al cliente. El ID del usuario se agrega como atributo al *socket* para que en el momento en el que se quiera hacer una operación contra la base de datos, se pueda saber a qué usuario pertenece el *socket*.

El servidor escucha los siguientes mensajes que emite el cliente:

- `TRANSACCION`: este mensaje lo emite el cliente cuando quiere crear una nueva transacción. Antes de crear la transacción comprueba si es válida y para ello verifica aspectos como: si el ID de la divisa existe o si el precio es el mismo que tiene registrado el servidor. En el caso de la compra, se comprueba si el usuario tiene suficientes fondos en su cartera de `fiat` para poder realizarla.
- `RANKING`: devuelve un listado con el *ranking* de los usuarios que tienen más dinero. Para calcular el orden del *ranking*, se suma el valor en euros de todas las carteras.
- `resetear`: vacía todas las carteras de criptodivisas del usuario. La cartera de `fiat` se modifica para que tenga un balance de 10.000,00 €. Además, suma uno al número de *resets* del usuario y lo guarda en la base de datos.
- `borrar-cuenta`: se borra la cuenta del usuario de la base de datos. Asimismo, suma uno al número de *resets* y lo guarda en la colección de eliminados de la base de datos junto al ID de la cuenta de Google. Además, desconecta a todos los clientes que hayan hecho inicio de sesión con ese usuario.

Capítulo 7

Generación de la aplicación web progresiva

Tal y como se verá a continuación, los requisitos principales para que una página web sea PWA se implementan en el lado del cliente, salvo el de que sea accesible por HTTPS. Al usar Vue.js, he podido utilizar el *plugin* oficial `cli-plugin-pwa` [48], facilitándome mucho el proceso de hacer de Medusa una aplicación web progresiva.

7.1. Requisitos para la instalación

Para que una página web pueda ser instalable como PWA, tiene que cumplir los siguientes requisitos:

- La aplicación web no tiene que haberse instalado anteriormente como PWA.
- La página web tiene que ser accesible por HTTPS.
- La web tiene que tener en la raíz el fichero `manifest.json` [49] que contenga los campos:
  - `short_name`: el nombre de la aplicación.
  - `icons`: los iconos que va a usar, debe incluir obligatoriamente uno de *192px* y otro de *512px*.
  - `start_url`: la ruta inicial.
  - `display`: la forma en la que se va a mostrar la aplicación cuando se instale. Puede tener 3 valores: `fullscreen`, `standalone` o `minimal-ui`.
- Registrar un *service worker* que contenga un manejador para el evento `fetch`

7.2. Medusa como PWA

Como Medusa es una PWA, en el momento en el que detecta que el usuario tiene un dispositivo donde se puede realizar la instalación, se muestra una pequeña ventana emergente, informando al usuario de la posibilidad de instalar Medusa. En iPhone y en Safari para Mac no se muestra la ventana emergente, ya que Apple bloquea la API de JavaScript.

```
{
  "name": "Medusa",
  "short_name": "Medusa",
  "theme_color": "#fff",
  "icons": [
    {
      "src": "icon/android-icon-192x192.png",
      "sizes": "192x192",
      "type": "image/png"
    },
    {
      "src": "icon/android-icon-512x512.png",
      "sizes": "512x512",
      "type": "image/png"
    },
    ...
  ],
  "start_url": "/",
  "display": "standalone",
  "background_color": "#677eb6"
}
```

Código 7.1: Manifest.json generado por Workbox

7.3. Service worker

Por defecto, el *service worker* que genera Workbox de forma automática implementa únicamente la funcionalidad mínima para que la web sea instalable. Cómo yo quería implementar funcionalidad extra, tuve que hacer un *script* para que luego Workbox lo inyecte al *service worker* que genera.

La primera funcionalidad extra que quise implementar en el *service worker* fue la de poder cachear un listado de ficheros. De esta forma en el caso de que el usuario intentase entrar a Medusa sin conexión a internet, se cargaría una página de error con un diseño propio. Los ficheros que cachea Medusa para lograr esto son:

- `offline.html`: es el fichero HTML con la página de error.
- `Quicksand-VariableFont_wght.ttf`: el fichero con la fuente de Medusa.
- `favicon.ico`: el *favicon* de la web.

La segunda funcionalidad extra implementada en el *service worker*, es que en el caso de que se lance una versión nueva de Medusa, la web lo detectaría e informaría al usuario mostrando una ventana emergente que refresca la página. Refrescando la página se reinstala el *service worker* y se cargan los ficheros actualizados de la web.

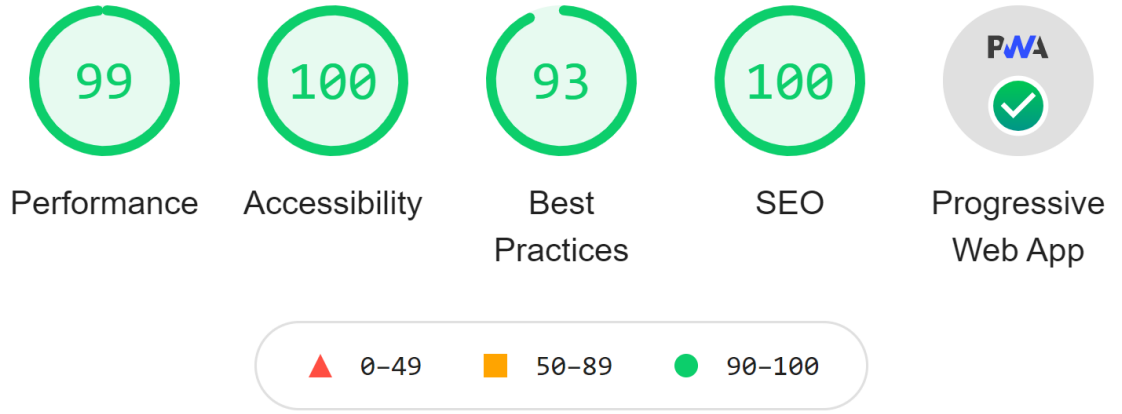

Figura 7.1: Resultado de Medusa en un test de Lighthouse

7.4. Prueba de la implementación de la aplicación web progresiva

Para comprobar que la implementación de PWA es correcta, usé Lighthouse [50]. Lighthouse es una herramienta creada por Google que, entre otras cosas, te informa de todos los requisitos que faltan para que tu web sea PWA. Otra de las utilidades de Lighthouse es la de poder realizar tests a tu web. La primera vez que lancé el test de Lighthouse la puntuación de accesibilidad fue de un 89, esto se debía a que había añadido una etiqueta HTML que impedía hacer zoom [51]. En la figura 7.1 se muestra un resultado de un test de Lighthouse de Medusa habiendo modificado la etiqueta HTML mencionada anteriormente. El 93 de *Best Practices* se debe a que Lighthouse recomienda que no se muestre por consola ningún tipo de aviso (*warning*), pero la API de Google *OAuth 2.0* muestra avisos por consola que no se pueden ocultar.

Capítulo 8

Despliegue de la aplicación

Para realizar el despliegue de Medusa, me aproveché de muchas de las ofertas disponibles al registrarme en el lote para estudiantes Github [52]. En la tabla 8.1 se pueden encontrar un resumen de todos los servicios utilizados para el despliegue de Medusa.

8.1. Dominio para la web

Al realizar el despliegue, el nombre del dominio fue lo primero que se registró. Para el registro se uso Name.com [53], ya que te regalaban un registro gratis. El dominio elegido fue: "medusapp.live". El dominio principal se ha configurado para que tenga dos subdominios:

- **servidor.medusapp.live**: apunta al servidor de Medusa.
- **dev.medusapp.live**: apunta a la versión de desarrollo del cliente de Medusa. El cliente se conecta a una ejecución local del servidor para poder hacer pruebas, sin que afecten a los usuarios finales.

Tanto los subdominios como el dominio principal están configurados para que solo sean accesibles utilizando HTTPS.

8.2. Cliente

La parte del cliente se hospeda en la plataforma de aplicaciones de Digital Ocean [54]. Elegí esta plataforma porque ya había trabajado antes con ella. Una de las ventajas de usar una plataforma de aplicaciones como la de Digital Ocean, es que no tienes que configurar nada del servidor. Lo único necesario es subir el código y ellos se encargan de construirlo y

Componente	**Servicio usado**
Dominio web	Name.com [53]
Hosting para el cliente	Digital Ocean [54]
Hosting para el servidor	Heroku [55]
Base de datos	MongoDB Atlas [56]
Control de errores	Sentry.io [57]

Tabla 8.1: Servicios externos utilizados para el despliegue de Medusa

de lanzar el servidor. Además, se puede conectar con una de las ramas de tu repositorio de Github/Gitlab. Al estar conectado con tu repositorio, en el momento en el que haces un *commit*, se construye tu aplicación y la vuelve a lanzar. Haciendo que el proceso de "Subir código al repositorio → Construir el proyecto → Subir al servidor el código compilado" se agilice mucho, ya que se hace de manera automática.

La configuración de elementos como por ejemplo, la dirección del servidor a la que el cliente se va a conectar, se realiza mediante variables de entorno. Las variables de entorno que hay que definir para el correcto funcionamiento del cliente son:

- `VUE_APP_G_AUTH_ID`: el ID que te proporciona Google para usar la API de *OAuth 2.0*.
- `VUE_APP_MEDUSA_SERVER_URL`: la URL del servidor de Medusa.
- `VUE_APP_SENTRY_DSN`: el identificador único que te proporciona Sentry.io al crear un proyecto. Esta variable de entorno es opcional.

8.3. Servidor

El servidor en un primer momento se desplegó en la plataforma de aplicaciones de Digital Ocean para que el cliente y el servidor estuvieran en el mismo servicio. Como es una plataforma que se ha lanzado recientemente, no está tan refinada como otras. Hubo numerosos problemas con la implementación, como por ejemplo, el bloqueo de las peticiones del cliente al servidor por una mala configuración de las cabeceras CORS.

Finalmente, me decidí por Heroku [55]. Heroku es una plataforma de aplicaciones similar a la de Digital Ocean, pero mucho más madura, ya que lleva desde 2011 soportando aplicaciones en Node.js [58]. El proceso de puesta en marcha del servidor fue muy similar al del cliente. Conecté directamente mi repositorio de Github del servidor de Medusa, permitiendo así que con cada *commit* se actualice y se relance el servidor automáticamente. Todas las configuraciones del servidor, como la URL del servidor de MongoDB al que se va a conectar, o la URL del cliente de Medusa, se hacen usando variables de entorno. Las variables de entorno necesarias para el funcionamiento del servidor de Medusa son:

- `G_AUTH_ID`: el ID que te proporciona Google para usar la API de *OAuth 2.0*. Para que el login se pueda realizar, tiene que ser el mismo ID que el cliente.
- `MONGO_DB_URL`: la URL de conexión de la base de datos MongoDB.
- `APP_SECRET`: el secreto que se usa para firmar los JWT.
- `MEDUSA_APP_URL`: la URL del cliente de Medusa.
- `MEDUSA_SERVER_PORT`: el puerto del servidor que va a usar Medusa.

8.4. Base de datos

Al usar un servicio donde no tengo acceso directo al servidor de Medusa, es necesario que la base de datos de MongoDB esté alojada en un servicio diferente. Tenía dos opciones para lanzar la base de datos:

1. Montar y configurar yo un servidor donde poder instalar MongoDB.

2. Usar un servicio donde me gestionan toda la configuración de la base de datos.

La primera opción la quería evitar a toda costa, ya que me iba a llevar mucho tiempo montar un servidor y securizarlo correctamente. Así que decidí que la mejor opción era la segunda.

MongoDB Atlas [56] es un servicio de base de datos, que te permite lanzar *clusters* de MongoDB a través de una interfaz web muy fácil de usar. Usando este servicio pude lanzar la base de datos en cuestión de minutos. Lo único que tuve que hacer fue configurar en la variable de entorno del servidor la URL de la base de datos que proporciona Atlas. Otra de las funcionalidades que te da Atlas es la de replicarte la base de datos de forma automática para que en el caso de que se caiga uno de los nodos, se levante otro, haciendo que siempre sea accesible y que los datos no se pierdan.

8.5. Control de errores

Uno de los problemas que me encontré al lanzar la web es que si saltaba una excepción en el lado del cliente se mostraba por la consola del navegador, pero no tenía forma de saber qué había pasado y por qué había ocurrido la excepción.

Por lo tanto, necesitaba un sistema que me permitiera monitorizar la página web de Medusa, y que me avisara cuando ocurren incidencias como una excepción de código. Me decidí por implementar Sentry.io [57] en el lado del cliente, ya que era justo lo que estaba buscando. Su integración fue muy sencilla ya que tienen una biblioteca específica para Vue.js. Gracias al registro de errores de Sentry.io pude solucionar algunos fallos en el lado del cliente, de los que no me había percatado en el momento de lanzarla. Algunos de los errores que arregle gracias a Sentry.io son:

- Un fallo al registrar el *service worker* en una de las subpáginas (*ej: /cartera/bitcoin*). El error ocurría porque no registraba el fichero `serviceworker.js` con una ruta absoluta, entonces se registraba como si el fichero estuviera a la misma altura que la subpágina.
- Un fallo que ocurrió al cambiar el modelo de los datos que intercambia el servidor con el cliente. Como en el cliente no había implementado un sistema que detectara una nueva versión del servidor, el cliente esperaba recibir los datos con la estructura de la versión anterior. A raíz de esto implementé que el cliente refrescara la web de manera automática cuando la versión del servidor se actualiza.

Sentry.io te permite ver un listado con todos los errores que ha detectado. Puedes filtrarlos y marcarlos como resueltos. También te permite ver información detallada de cada error como: el navegador que estaban usando en el momento del error, la pila de ejecución, la IP de la persona, etc.

Otro de los beneficios de haber implementado Sentry.io, es que te proporciona métricas del tiempo de carga de cada página, permitiéndote ver cuáles son las páginas que tardan más en cargar y que están menos optimizadas. En la figura 8.1 se pueden ver alguna de las estadísticas que Sentry.io proporciona sobre el cliente de Medusa. Las estadísticas de la figura son:

- *First contentful paint*: tiempo que se tarda en pintar en pantalla el primer elemento del documento HTML. El 88 % de las peticiones tardan menos de 1 segundo. El 10 % entre 1 y 3 segundos. Por último, el 2 % tarda 3 o más segundos.

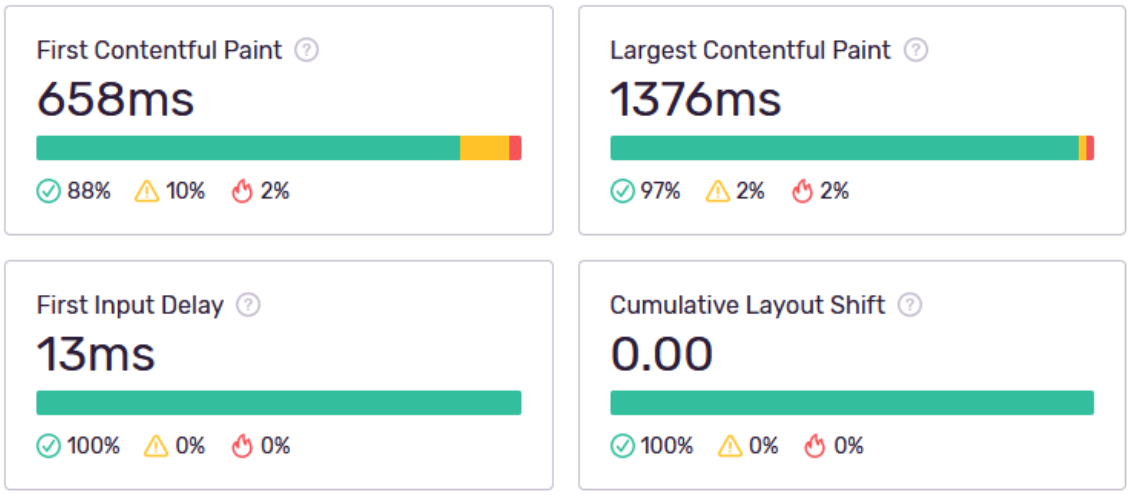

Figura 8.1: Métricas que Sentry proporciona de Medusa

- *Largest contentful paint*: tiempo de renderizado del elemento más grande del documento HTML. El 97 % de las peticiones tardan menos de 2.5 segundos. El 2 % entre 2.5 segundos y 4 segundos. Finalmente, el 2 % de las peticiones tardan más de 4 segundos.
- *First input delay*: tiempo de respuesta del navegador a la interacción del usuario. El 100 % de las peticiones tardan menos de 100 milisegundos.
- *Cumulative layout shift*: mide el movimiento inesperado de contenido. Esta métrica se centra en medir la estabilidad visual de la web, intentando cuantificar la frecuencia con la que ocurren cambios inesperados en el diseño de la web.

Capítulo 9

Ejemplo de uso

El diseño de Medusa es minimalista y simple para no confundir al usuario. A continuación, vamos a analizar las distintas páginas que componen Medusa. Cualquier usuario puede acceder a Medusa a través de la URL: `https://medusapp.live`.

En la figura 9.1 se puede observar la pantalla de inicio de sesión, el logo de Medusa ha sido diseñado por Valeria Pérez Marrero. El botón para iniciar sesión abre el selector de cuentas de Google.

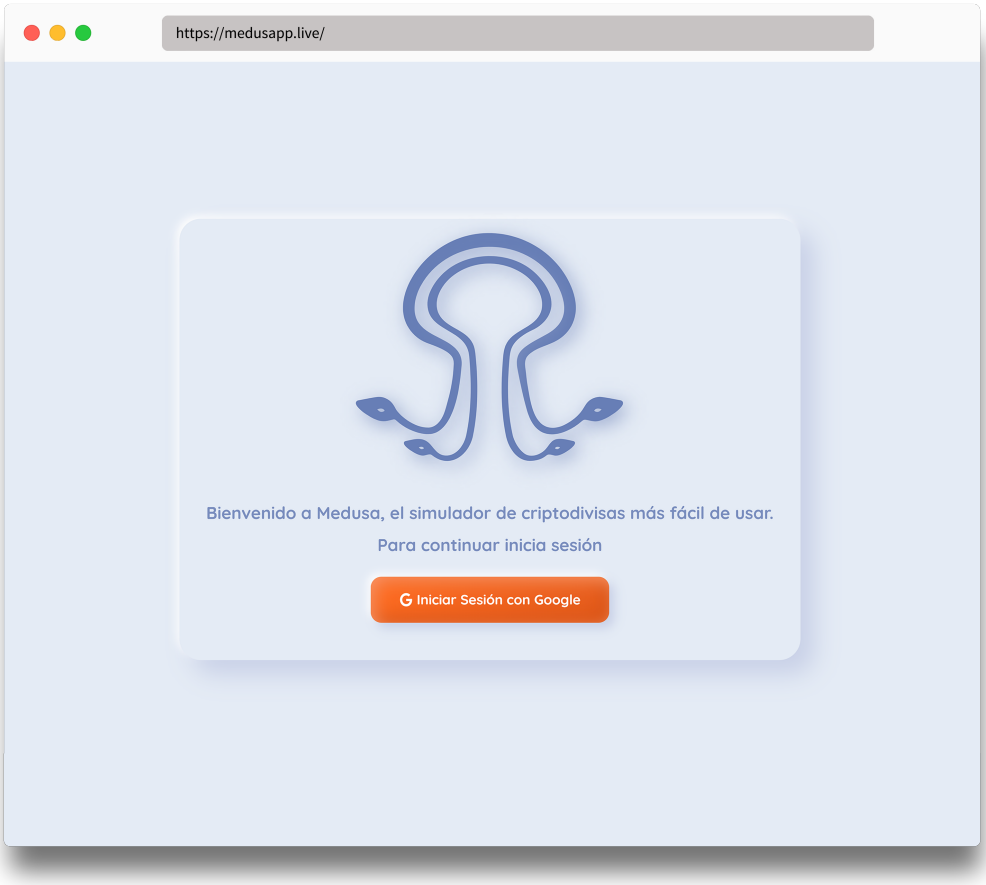

Figura 9.1: Página de inicio de sesión

La figura 9.2 contiene la página de registro de Medusa, la imagen de perfil cambia dinámicamente dependiendo del nombre de usuario. Al pulsar *Continuar* se completa el registro, en caso de que el nombre de usuario ya exista, se muestra una alerta al usuario y se pide que escribe un nombre de usuario distinto.

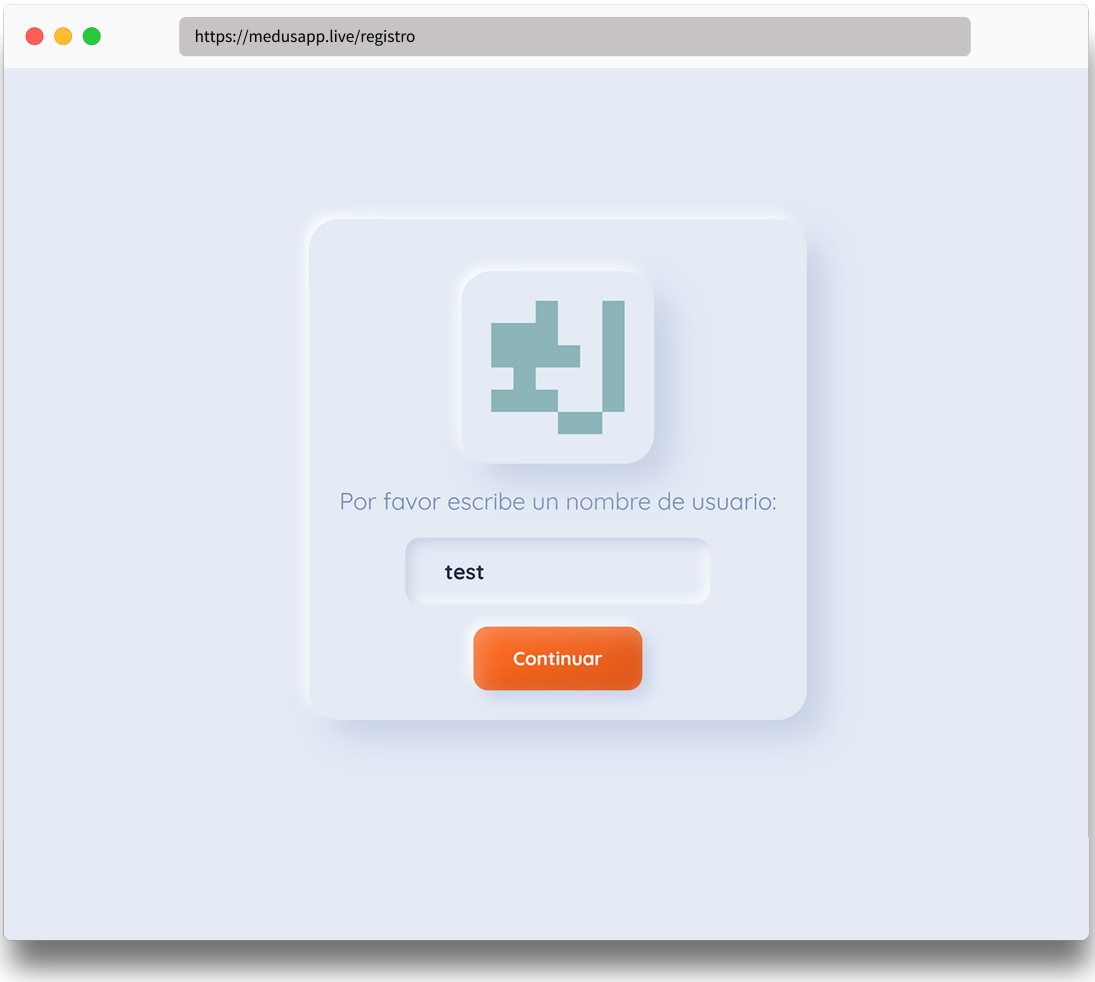

Figura 9.2: Página de registro

En la figura 9.3 se puede observar la página de inicio. Esta página es la primera que se muestra al iniciar sesión. Para navegar por Medusa en la versión de escrituro se usa una barra de navegación superior. Con el selector desplegable de criptodivisa puedes cambiar la divisa actual, para poder ver sus gráficos y su precio actual. En el caso de que se pulse el botón *Crear Transacción*, se va a abrir la página de la divisa seleccionada en ese momento.

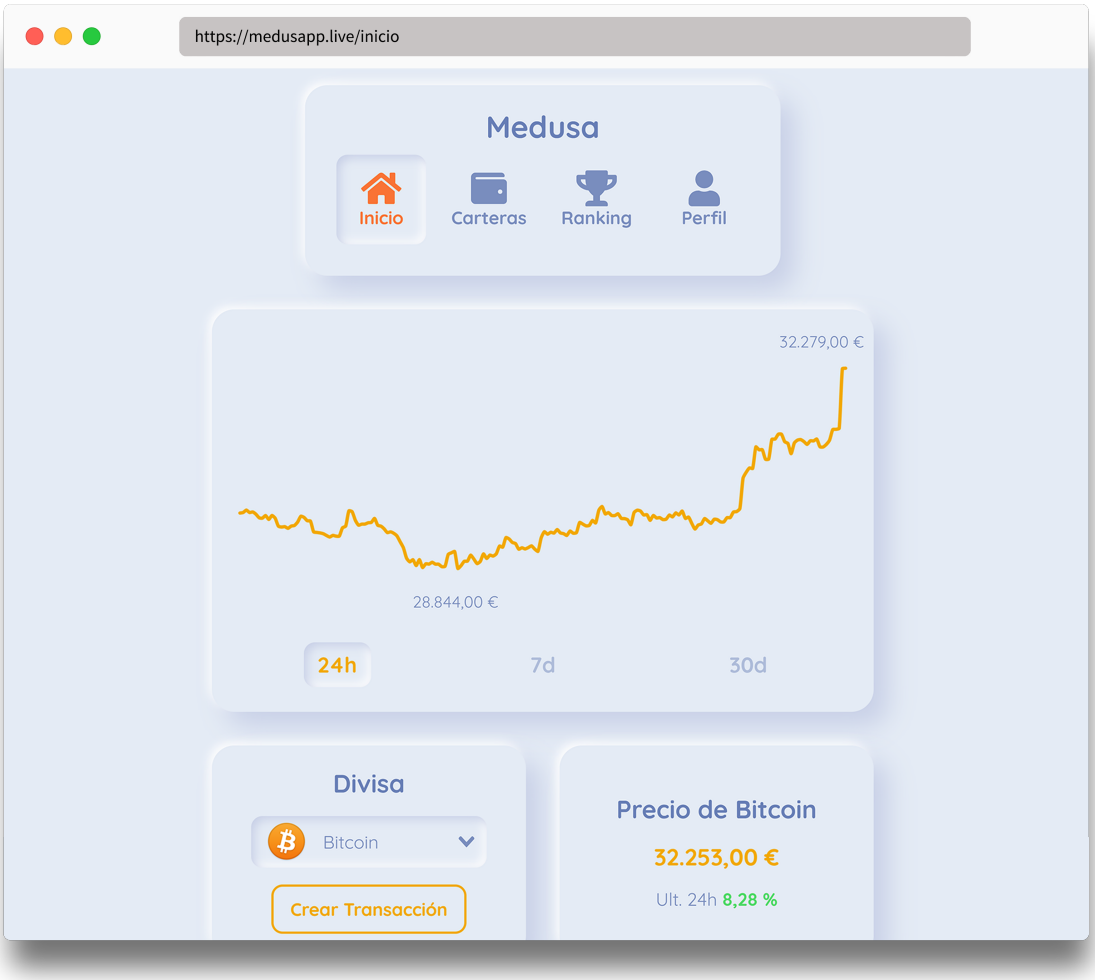

Figura 9.3: Página de inicio (parte 1)

La figura 9.4 muestra la parte inferior de la página de inicio. La parte inferior contiene información sobre la criptodivisa seleccionada, se describe el proyecto y los objetivos de la divisa, además existe un botón para abrir el artículo de Wikipedia de la criptodivisa para poder seguir informándose. Por último, hay un apartado de enlaces relevantes que contiene los siguientes enlaces:

1. Página principal de la criptodivisa.
2. Twitter de la criptodivisa.
3. Enlace al subreddit oficial.
4. Repositorio de Github del proyecto.

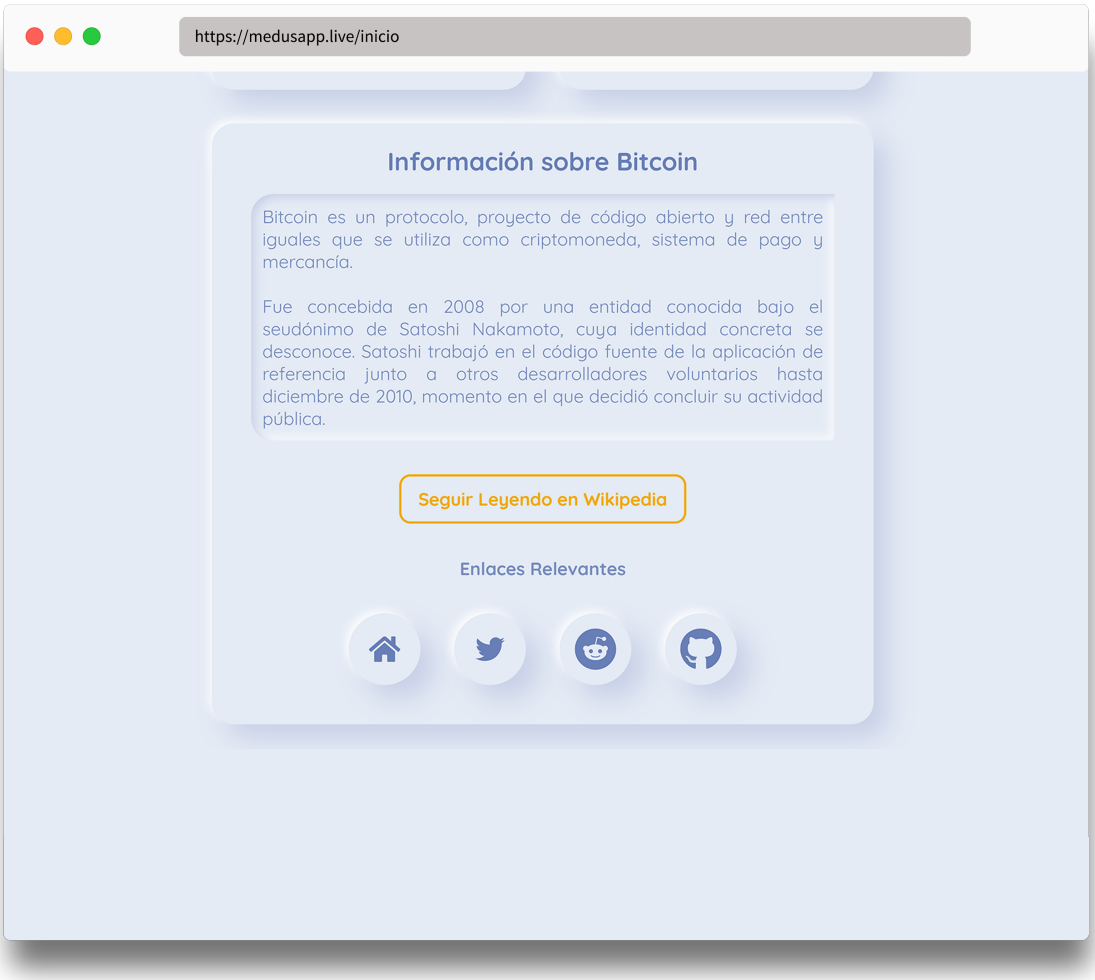

Figura 9.4: Página de inicio (parte 2)

En la figura 9.5 se puede observar la página de carteras. El *Total* contiene la suma total en euros del precio de todas las divisas que tiene el usuario. A la izquierda está el botón para poder ver un gráfico de las carteras, y a la derecha está el botón que te permite reordenar las carteras por valor o por nombre. El porcentaje que está debajo del valor en euros de cada cartera representa las ganancias o perdidas actuales para esa divisa.

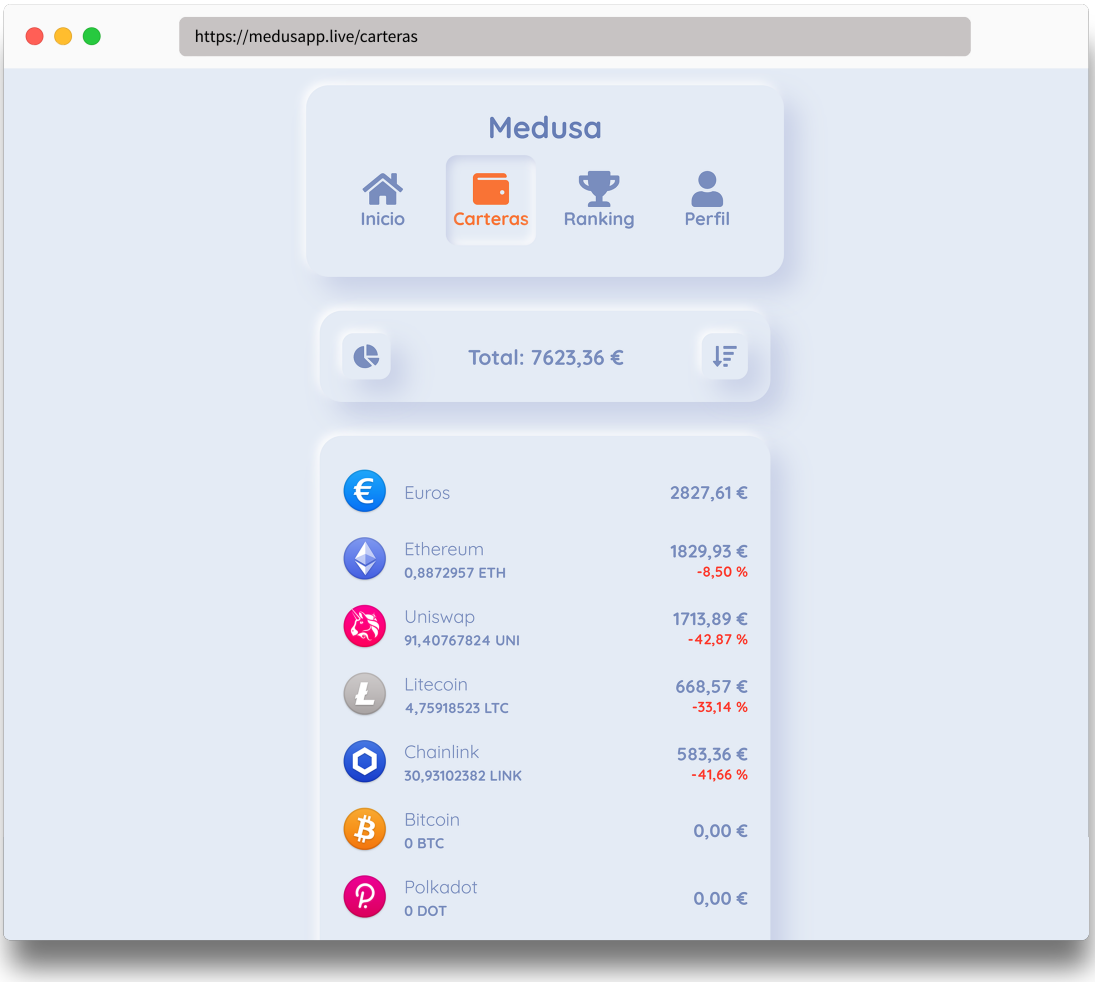

Figura 9.5: Página de las carteras

La figura 9.6 contiene el gráfico en forma de tarta con la información de todas las carteras de criptodivisas del usuario. El gráfico es interactivo, en el momento en el que se haga clic o se pase el ratón por encima de uno de los componentes, se muestra en el centro el logo de la divisa y en la parte inferior el valor de la cartera de esa divisa.

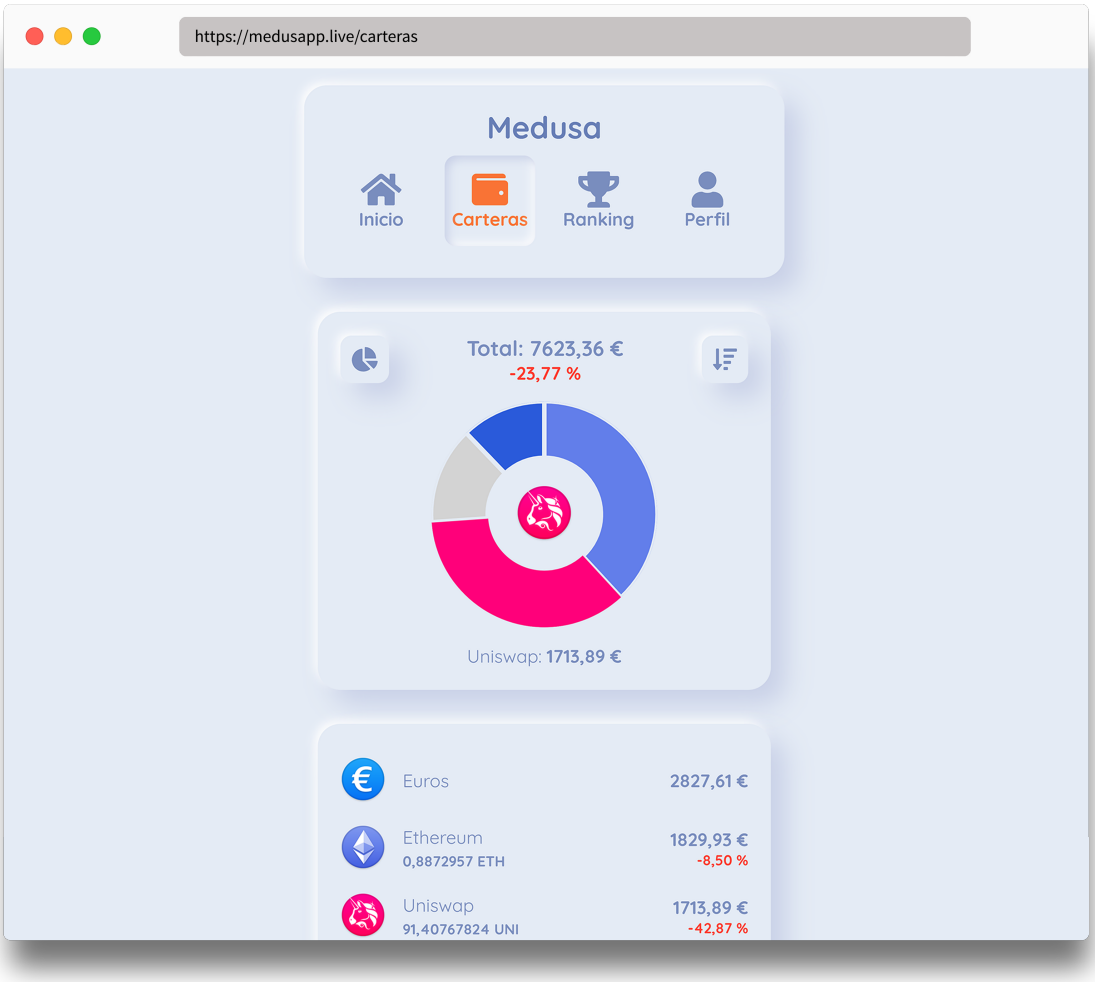

Figura 9.6: Página de las carteras con el gráfico

En la figura 9.7 se muestra la página con la cartera de Ethereum. Todos los gráficos de precios de Medusa tienen el color de la divisa seleccionada. Además, con los botones que están en la parte inferior del gráfico se puede cambiar la franja de tiempo para visualizar los precios de las divisas.

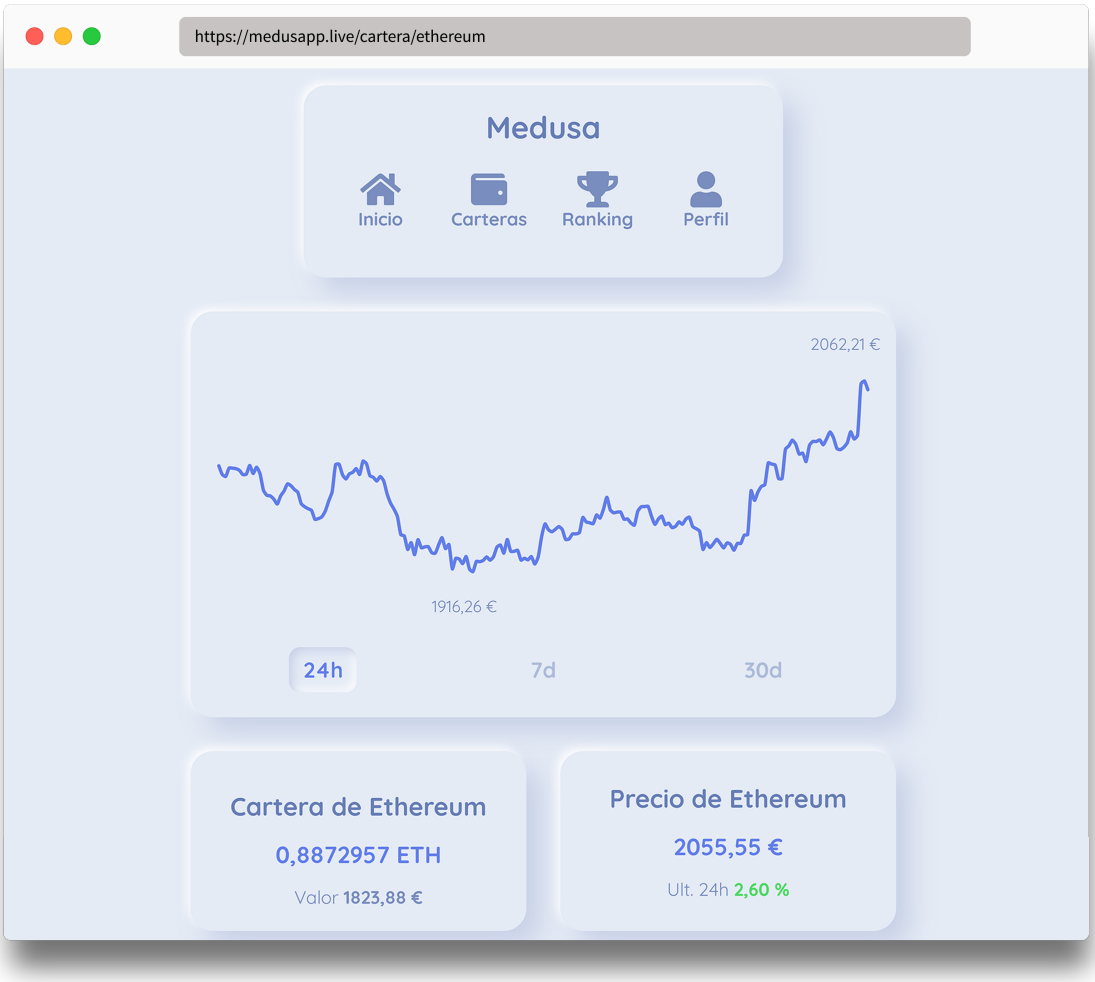

Figura 9.7: Página de la cartera de Ethereum (parte 1)

La figura 9.8 muestra la parte inferior de la página de la cartera de Ethereum. Inicialmente se compra o vende las divisas escribiendo el número de euros deseados, con el botón de las flechas se puede cambiar para en vez de tener que escribir en euros, puedas usar las unidades de la moneda (ej: comprar 1 bitcoin). El módulo de transacciones realizadas te permite visualizar todas las transacciones de la cartera actual.

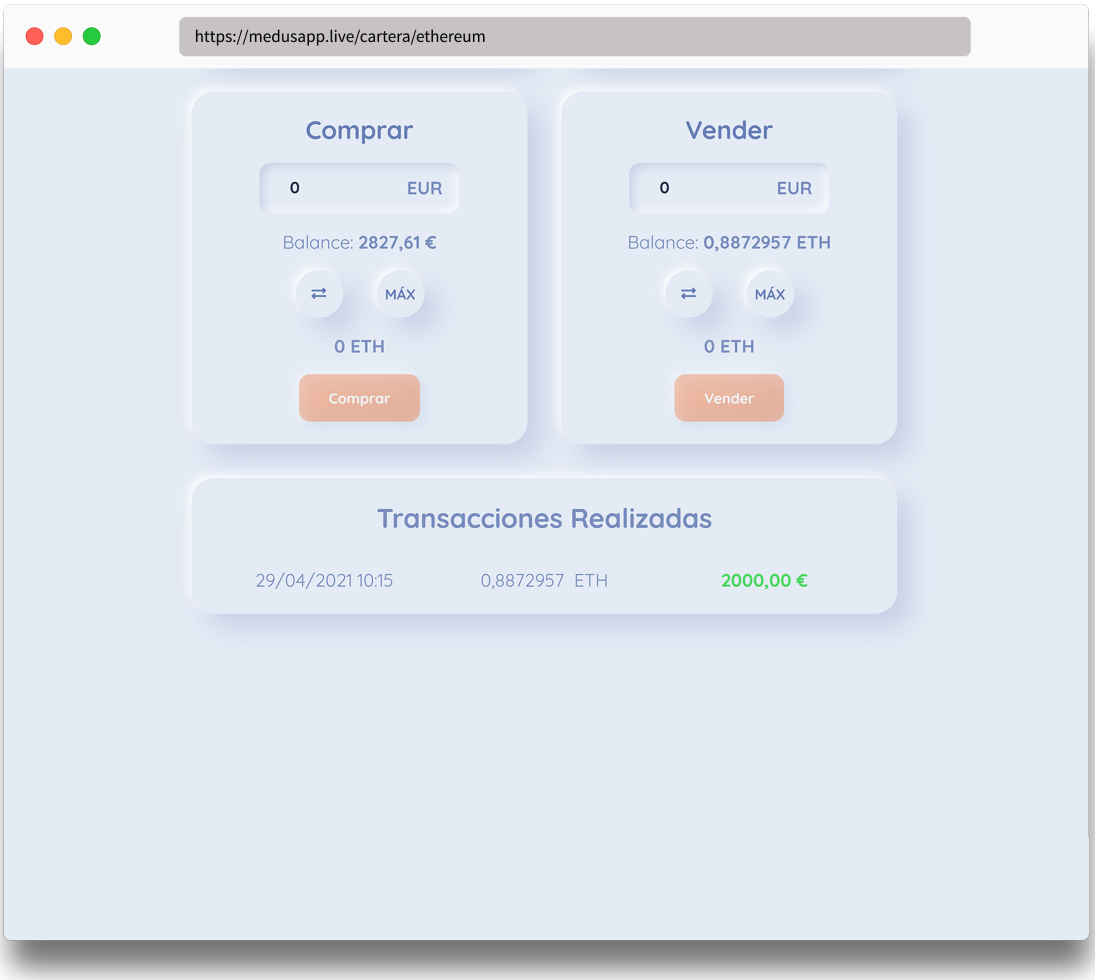

Figura 9.8: Página de la cartera de Ethereum (parte 2)

En la figura 9.9 se puede observar la página del ranking. En ella se ordena de mayor a menor todos los usuarios de la web por el valor en euros de todas sus carteras. También se muestra el número de *resets* que ha realizado el usuario.

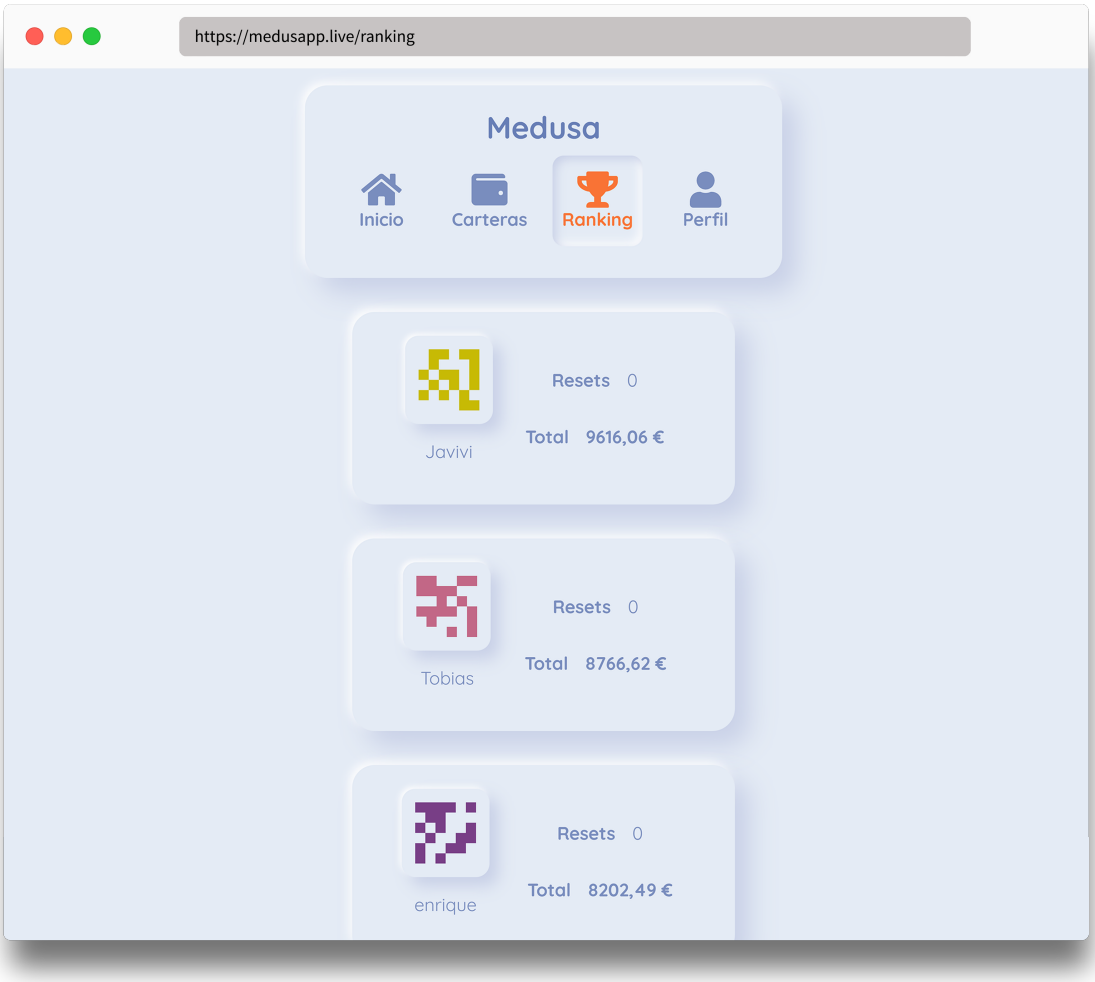

Figura 9.9: Página del ranking

La figura 9.10 contiene la página del perfil de un usuario. En esta página se muestra la siguiente información: el nombre de usuario, imagen de perfil, fecha de registro y número de *resets*. Además, tienes los siguientes cuatro botones:

- **Cerrar Sesión**: cierra la sesión actual y redirige a la página de inicio de sesión.
- **Tema Oscuro**: habilita o deshabilita el tema oscuro para Medusa. Por defecto Medusa carga con el mismo tema (claro o oscuro) que el dispositivo gracias a la propiedad de CSS *prefers-color-scheme* [59].
- **Resetear Cuenta**: vacía todas las carteras, menos la de euros que toma un valor de 10.000 € y suma un *reset* a la cuenta. Se muestra una ventana emergente de confirmación antes de realizar el *reset* para verificar que realmente se quiere realizar esa acción.
- **Borrar Cuenta**: se elimina la cuenta y sus datos. El ID de Google de la cuenta se almacena en una colección para llevar un registro de los *resets* en caso de que se vuelva a registrar el mismo usuario. Igual que con el *reset*, se muestra una ventana emergente de confirmación.

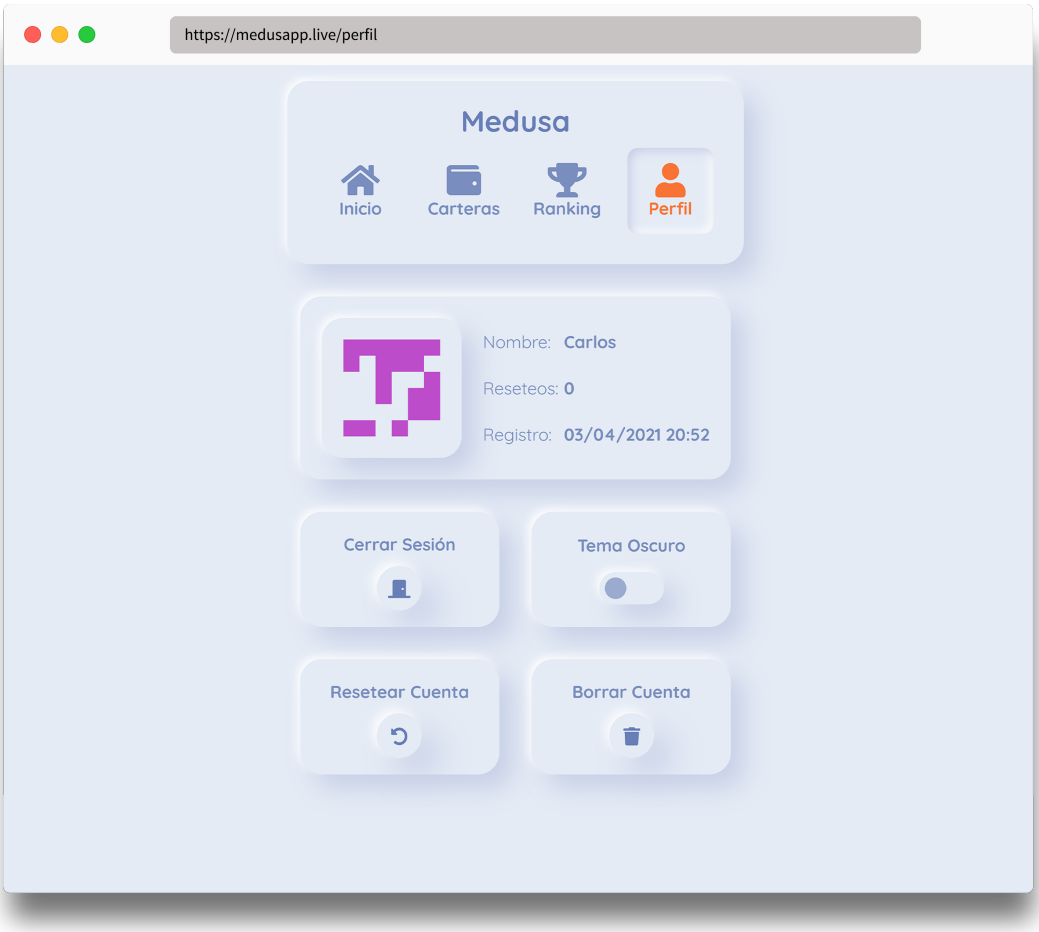

Figura 9.10: Página de perfil

En la figura 9.11 se puede apreciar la página de carteras con el tema oscuro de Medusa. Todos los componentes de la interfaz se adaptan al tema oscuro para que no haya ningún tipo de error visual.

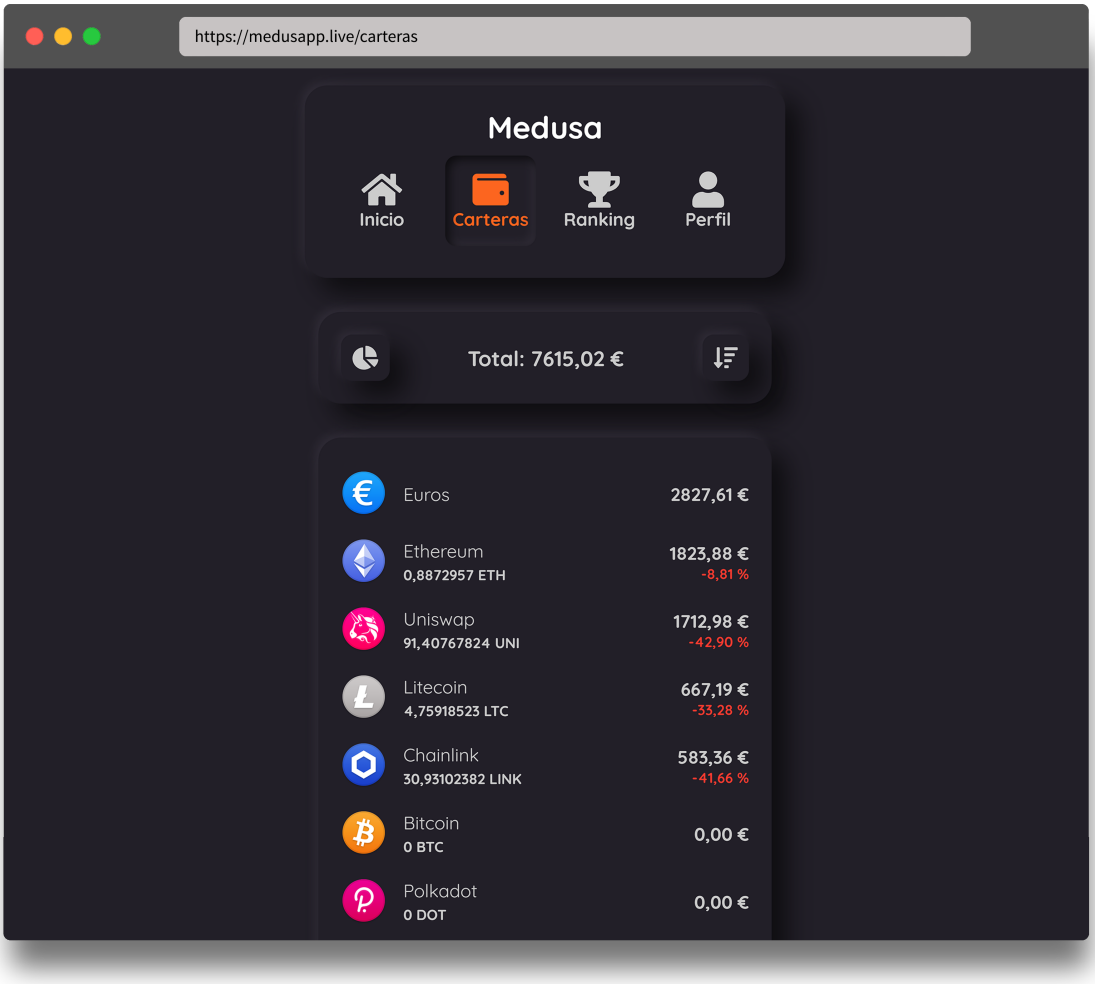

Figura 9.11: Página de las carteras con el tema oscuro

La figura 9.12 contiene la página de la cartera de Bitcoin con el tema oscuro. Como se puede observar, los gráficos también se adaptan sin ningún tipo de problema al tema oscuro.

Figura 9.12: Página de la cartera de Bitcoin con el tema oscuro

Por último, en la figura 9.13 se puede visualizar cómo se muestra Medusa en móvil. Toda la web se adapta sin ningún tipo de problema a dispositivos móviles, ya que es totalmente *responsive*. El menú superior se traslada a la parte inferior para actuar como una barra de navegación típica de las aplicaciones móviles.

Figura 9.13: Medusa en dispositivos móviles

Capítulo 10

Conclusiones y trabajo futuro

El principal objetivo de este proyecto ha sido desarrollar una aplicación web progresiva para simular la inversión en criptodivisas. Tras el desarrollo de Medusa, los objetivos propuestos en la sección 1.2 se han cumplido en su totalidad:

- Medusa es una aplicación de página única. Este objetivo se ha conseguido gracias al uso de Vue.js.
- La comunicación en tiempo real entre el cliente y el servidor, se ha implementado utilizando el protocolo de WebSocket.
- Gracias a la API de Google OAuth 2.0, no se ha necesitado implementar un sistema de inicio de sesión y de registro propio.

A pesar de haber cumplido todos los objetivos, hay alguna funcionalidad que no se ha llegado a implementar y que se propone como trabajo futuro.

Cuando empecé a desarrollar Medusa decidí desde el principio hacer la parte del cliente usando Vue.js. Como no había utilizado el *framework* nunca, tuve que dedicar algún tiempo a aprender cómo utilizarlo. A pesar de haber "perdido" tiempo aprendiendo a usarlo, la implementación del lado del cliente no hubiera sido posible sin haber utilizado Vue.js, o por lo menos en las fechas dictadas en el plan de trabajo.

Haber realizado el despliegue de forma pública me permitió recoger *feedback* de los usuarios sobre Medusa. Uno de los aspectos que más ha gustado a los usuarios que han probado Medusa es el ranking, ya que había cierta rivalidad por conseguir estar el primero. Como con el despliegue se implementó Sentry.io en el lado del cliente, se pudo resolver algunos errores críticos que impedían el correcto funcionamiento del cliente.

Por último, desarrollando Medusa me he dado cuenta del futuro que tienen las aplicaciones web progresivas, aunque Apple siga poniendo impedimentos a las PWA (por ejemplo: las notificaciones web en iOS no están disponibles), a mi parecer creo que son el futuro del desarrollo de aplicaciones multiplataforma para iOS y Android. Las aplicaciones web resuelven algunos problemas muy importantes relacionados con la publicación de una aplicación nativa en plataformas móviles, algunos de estos problemas son:

- Largos tiempos de aprobación para las actualizaciones de las aplicaciones. Esto puede suponer, que en el caso de que se detecte un fallo crítico en la aplicación, pueda tardar en algunos casos varios días hasta que la actualización se publique. En el caso anterior las consecuencias pueden ser catastróficas, ya que muchos usuarios pueden optar por desinstalar la aplicación incluso poner malas *reviews* en la tienda de aplicaciones.

- Las ventas de contenido conllevan una comisión del 30 %. Esto puede suponer una cantidad de dinero muy importante en el caso de desarrolladores pequeños.
- Tiempo de desarrollo. En el caso de querer tener una aplicación nativa en iOS y Android, es necesario tener dos fuentes de código, por lo que el tiempo que se invierte en el desarrollo es más largo. Además, hay veces que la paridad de las características de la misma aplicación para iOS y Android no es la misma ya que hay funcionalidades que en un sistema operativo se pueden implementar y en el otro no.

10.1. Trabajo futuro

Durante el desarrollo de Medusa han surgido algunas ideas que mejorarían la aplicación y que no se han integrado por falta de tiempo. Entre ellas podemos destacar las siguientes:

- **Más criptodivisas**: en total hay más de 10.000 criptodivisas. En Medusa actualmente hay ocho criptodivisas distintas, es por ello que una de las mejoras es aumentar el número de criptodivisas disponibles a por ejemplo el top 100 por capitalización de mercado [60]. Esto puede permitir a los usuarios aprender sobre nuevas criptodivisas. Uno de los retos de esta mejora sería adaptar la interfaz correctamente a un listado tan grande de criptodivisas. Por ejemplo, en la vista de carteras se debería de implementar un botón buscador para poder encontrar más facilmente la divisa deseada.
- **Aspectos sociales**: aunque Medusa es un simulador, me hubiera gustado agregar elementos sociales como: poder agregar usuarios a un listado de amigos o crear un ranking personalizado para tu grupo de amigos. Esto sería una extensión interesante porque puede permitir la difusión de Medusa a un mayor número de usuarios gracias al boca a boca. Además seria interesante integrar un chat dentro de Medusa para poder comunicarte con los usuarios para comentar estrategias, nuevas divisas, etc. Otro aspecto social que puede integrarse en Medusa sería que cada criptodivisa tuviera un foro donde los usuarios pueden comentar las últimas noticias o cualquier tema relacionado con la divisa.
- **Más servicios para iniciar sesión**: además de Google, se pueden implementar servicios para iniciar sesión cómo: Apple, Facebook, Twitter, Github... Esto puede suponer acercar Medusa a un mayor número de usuarios ya que no estás obligando que se tenga una cuenta de Google para poder iniciar sesión. Lo problemático de hacer esta ampliación es que el hecho de integrar tantas bibliotecas distintas para poder iniciar sesión, puede hacer que la web se ralentice.
- **Testing**: el testing en el desarrollo de aplicaciones software es crucial porque te permite detectar fallos de manera automática antes de lanzar la aplicación al público. Me hubiera gustado implementar test unitarios en el cliente y el servidor usando bibliotecas como Jest [61] o Mocha [62]. En el caso de Medusa me hubiera ayudado haber implementado test unitarios para poder saber si algunas de las funcionalidades nuevas, introducían errores en el sistema. Con Sentry.io he podido detectar estos errores, pero siempre ha sido porque les ha ocurrido a un usuario, esta forma de resolver errores no es la más óptima, ya que el objetivo siempre debe ser que el usuario final no tenga ningún error al usar la aplicación.

Chapter 11

Conclusions and Future Work

The primary goal of this project has been to develop a progressive web application for simulating cryptocurrency investments. All the objectives proposed in the section 2.2 have been accomplished:

- Medusa is a single page app. This objective has been achieved thanks to the use of Vue.js.
- Real time communication between the client and the server has been implemented using WebSocket protocol.
- Thanks to the Google OAuth 2.0 API, there was no need to implement a custom login and registration system.

Despite having fulfilled all the goals, there are some features that hasn't being implemented, they are proposed as future work.

When I started to develop Medusa, I decided from the start to implement the client using Vue.js. Since I had never used the framework, I needed to spend some time learning how to utilize it. Despite "wasting" my time learning how to use it, the implementation of the client-side would not have been possible without using Vue.js, or at least on the dates dictated on the workplan.

Thanks to making the deployment publicly I was able to gather feedback from the users. One of the features that users liked the most has been the ranking since there was some rivalry for being the richest user. With the deployment, Sentry.io was implemented on the client-side, which was key to detect and fix some critical bugs.

Finally, while developing Medusa I realized the future that progressive web apps have, despite Apple not implementing some features (e.g. web push notifications don't work on iOS). In my opinion, progressive web apps are the future of cross-platform development for iOS and Android. Web applications resolve some of the key problems related to the publishing of a native application on mobile platforms, some of these problems are:

- Long review times for the updates of applications. This may mean that in the event of a critical bug, it may take several days for the update to be published. In the previous case the consequences can be fatal, because many users could opt to uninstall the application and in the worst cases write negative reviews for the app.
- All sales have a 30% commission. This can be a very significant amount of money for small developers.

- Development time. In the case of wanting to have a native application for iOS and Android, it is necessary to have two different codebases, which translates to longer development times. Furthermore, there are some cases where the feature parity of the same application on iOS and Android is not the equals, since as the OS is different, not all APIs are the same.

11.1. Future Work

During the development of Medusa, some ideas arisen that could improve the application, these ideas have not been implemented because of the time constraints. These ideas are:

- **More cryptocurrencies**: in total there are more than 10.000 cryptocurrencies. In Medusa at the moment, there are eight cryptocurrencies, that's why it could be interesting to add the top 100 coins by market cap [60]. This could enable users to learn about new cryptocurrencies. One of the challenges of adding 100 coins would be the adaptation of the user interface. For example, on the wallets view, a search button should be implemented to be able to find the desired currency more easily.
- **Social features**: even though Medusa is first and foremost a simulator, I would have likes to add social features like: being able to add friends to a friend list or create a custom ranking for a group of friends. These features could cause Medusa to grow as users would recommend it to their friends because they want to compete against them. It could also be interesting to add a chat inside Medusa so the users could talk between them. Another social feature that could be implemented in Medusa would be the creation of a forum for each cryptocurrency, in the forums the users would be able to talk about anything related to the coin.
- **More login services**: besides Google, there could be implemented more login services such as: Apple, Facebook, Twitter, Github... This could mean more users as you are not limiting them to using a Google account. The problem with this feature is that adding all these libraries could make the application slower.
- **Testing**: unit testing during the development of an application is essential because it allows the finding of bugs automatically before release. I would have liked to implement unit testing on the client and the server using libraries like: Jest [61] and Mocha [62]. In this project adding unit tests would have helped me identify errors when implementing new features. With Sentry.io I have been able to detect these errors, but it has always been because it has happened to a user, this way of solving errors is not the best practice, since the goal should always be that the end user does not encounter any error while using the application.

Bibliografía

[1] Vue.js. Vue.js - the progressive javascript framework. https://vuejs.org/, 2021. Accedido: 2021-05-23.

[2] Google. Angular - the modern web developer's platform. https://angular.io/, 2021. Accedido: 2021-05-23.

[3] Facebook. React - una biblioteca de javascript para construir interfaces de usuario. https://es.reactjs.org/, 2021. Accedido: 2021-05-23.

[4] Binance. Guía para principiantes sobre el doble gasto. https://academy.binance.com/es/articles/double-spending-explained, 2021. Accedido: 2021-05-04.

[5] Satoshi Nakamoto. Bitcoin whitepaper. *URL: https://bitcoin.org/bitcoin.pdf*, 2008.

[6] Fundación IOTA. Iota. https://www.iota.org/, 2021. Accedido: 2021-05-04.

[7] Mozilla. Ajax. https://developer.mozilla.org/es/docs/Web/Guide/AJAX, 2021. Accedido: 2021-05-04.

[8] Javascript.indo. Long polling. https://javascript.info/long-polling, 2021. Accedido: 2021-05-04.

[9] Mozilla. Websockets. https://developer.mozilla.org/es/docs/Web/API/WebSockets_API, 2021. Accedido: 2021-05-04.

[10] Matt Gaunt. Introducción a los service workers. https://developers.google.com/web/fundamentals/primers/service-workers?hl=es, 2021. Accedido: 2021-05-04.

[11] Google. Progressive web apps. https://web.dev/progressive-web-apps/, 2021. Accedido: 2021-05-21.

[12] Webpack. Webpack. https://webpack.js.org/, 2021. Accedido: 2021-05-23.

[13] Vue. Vue loader. https://vue-loader.vuejs.org/, 2021. Accedido: 2021-05-23.

[14] Vue. v-if - renderización condicional. https://es.vuejs.org/v2/guide/conditional.html#v-if, 2021. Accedido: 2021-05-23.

[15] OpenJS Fundation. Node.js. https://nodejs.org/es/, 2021. Accedido: 2021-05-04.

[16] Google. V8. https://v8.dev/, 2021. Accedido: 2021-05-04.

[17] NPM. npm - build amazing things. https://www.npmjs.com/, 2021. Accedido: 2021-05-04.

[18] Google. Mongodb. https://www.mongodb.com/es, 2021. Accedido: 2021-05-04.

[19] . Rfc 7519 - jwt. https://datatracker.ietf.org/doc/html/rfc7519, 2021. Accedido: 2021-05-23.

[20] Hardt, D., Ed. The oauth 2.0 authorization framework. https://datatracker.ietf.org/doc/html/rfc6749, 2021. Accedido: 2021-05-23.

[21] Google. Google developer console. https://console.developers.google.com/, 2021. Accedido: 2021-05-23.

[22] Mozilla. Control de acceso http (cors). https://developer.mozilla.org/es/docs/Web/HTTP/CORS, 2021. Accedido: 2021-05-23.

[23] Google. Case studies featured apps. https://developers.google.com/identity/sign-in/case-studies?hl=en, 2021. Accedido: 2021-05-20.

[24] Coingecko. Coingecko api. https://www.coingecko.com/es/api, 2021. Accedido: 2021-05-21.

[25] miscavage. Coingecko node api. https://github.com/miscavage/CoinGecko-API, 2021. Accedido: 2021-05-24.

[26] Coingecko. Coingecko api terms. https://www.coingecko.com/es/api_terms, 2021. Accedido: 2021-05-24.

[27] Coingecko. Coingecko status. https://status.coingecko.com/, 2021. Accedido: 2021-05-21.

[28] Vue.js. Vue router - the official router for vue.js. https://router.vuejs.org/, 2021. Accedido: 2021-05-30.

[29] Vue.js. Vuex. https://vuex.vuejs.org/, 2021. Accedido: 2021-05-30.

[30] Facebook. Flux - application architecture for building user interfaces. https://facebook.github.io/flux/, 2021. Accedido: 2021-05-30.

[31] MetinSeylan. Vue-socket.io. https://github.com/MetinSeylan/Vue-Socket.io, 2021. Accedido: 2021-05-30.

[32] MetinSeylan. Vue-socket.io - vuex integration. https://github.com/MetinSeylan/Vue-Socket.io#-vuex-integration, 2021. Accedido: 2021-05-30.

[33] MetinSeylan. Vue-socket.io - component level usage. https://github.com/MetinSeylan/Vue-Socket.io#-component-level-usage, 2021. Accedido: 2021-05-30.

[34] Chart.js. Chart.js - open source html5 charts for your website. https://www.chartjs.org, 2021. Accedido: 2021-05-22.

[35] apertureless. vue-chartjs - easy and beautiful charts with chart.js and vue.js. https://vue-chartjs.org/, 2021. Accedido: 2021-05-31.

[36] janjakubnanista. downsample - downsampling methods for time series visualisation. https://github.com/janjakubnanista/downsample, 2021. Accedido: 2021-05-31.

[37] Sveinn Steinarsson. *Downsampling time series for visual representation*. PhD thesis, University of Iceland, 2013. LTTB en la página 21 a la 25.

[38] Jason Long. Identicons! https://github.blog/2013-08-14-identicons/, 2021. Accedido: 2021-05-31.

[39] Mozilla. Math.random(). https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Math/random, 2021. Accedido: 2021-05-31.

[40] davidbau. seedrandom.js - seeded random number generator for javascript. https://github.com/davidbau/seedrandom, 2021. Accedido: 2021-05-31.

[41] Mozilla. Htmlcanvaselement.todataurl(). https://developer.mozilla.org/es/docs/Web/API/HTMLCanvasElement/toDataURL, 2021. Accedido: 2021-05-31.

[42] hisour. Espacio de color yiq. https://www.hisour.com/es/yiq-color-space-26084/, 2021. Accedido: 2021-05-31.

[43] Express. Express - infraestructura web rápida, minimalista y flexible para node.js. https://expressjs.com/es/, 2021. Accedido: 2021-05-27.

[44] Socket.io. Socket.io. https://socket.io/, 2021. Accedido: 2021-05-27.

[45] Mongoose. Mongoose - elegant mongodb object modeling for node.js. https://mongoosejs.com/, 2021. Accedido: 2021-05-29.

[46] Martin Lorenz, Günter Hesse, and Jan-Peer Rudolph. Object-relational mapping revised-a guideline review and consolidation. In *ICSOFT-EA*, pages 157–168, 2016.

[47] auth0. socketio-jwt. https://www.npmjs.com/package/socketio-jwt, 2021. Accedido: 2021-05-29.

[48] Vue.js. @vue/cli-plugin-pwa. https://cli.vuejs.org/core-plugins/pwa.html, 2021. Accedido: 2021-05-24.

[49] Pete LePage, François Beaufort, Thomas Steiner. Add a web app manifest. https://web.dev/add-manifest/, 2021. Accedido: 2021-05-24.

[50] Google. Lighthouse. https://developers.google.com/web/tools/lighthouse?hl=es, 2021. Accedido: 2021-05-24.

[51] Mozilla. Viewport width and screen width. https://developer.mozilla.org/en-US/docs/Web/HTML/Viewport_meta_tag#viewport_width_and_screen_width, 2021. Accedido: 2021-05-30.

[52] Github. Github student developer pack. https://education.github.com/pack, 2021. Accedido: 2021-05-23.

[53] Name.com. Name.com. https://www.name.com/es-la, 2021. Accedido: 2021-05-23.

[54] Digital Ocean. Digital ocean app platform. https://www.digitalocean.com/products/app-platform/, 2021. Accedido: 2021-05-23.

[55] Heroku. Heroku - cloud application platform. https://www.heroku.com, 2021. Accedido: 2021-05-23.

[56] MongoDB. Mongodb atlas. https://www.mongodb.com/es/cloud/atlas, 2021. Accedido: 2021-05-23.

[57] Sentry. Sentry.io. https://sentry.io, 2021. Accedido: 2021-05-23.

[58] Heroku. Heroku - about. https://www.heroku.com/about, 2021. Accedido: 2021-05-23.

[59] W3C. prefers-color-scheme media feature. https://drafts.csswg.org/mediaqueries-5/#prefers-color-scheme, 2021. Accedido: 2021-05-23.

[60] Coinmarketcap. Principales 100 criptomonedas por capitalización de mercado. https://coinmarketcap.com/, 2021. Accedido: 2021-05-23.

[61] Facebook. Jest - delightful javascript testing. https://github.com/facebook/jest, 2021. Accedido: 2021-05-23.

[62] Mocha.js. Mocha - simple, flexible, fun. https://mochajs.org/, 2021. Accedido: 2021-05-23.