

Memoria Final - PTI

Javier Armaza Bravo /@estudiantat.upc.edu
Sergi Colomer Segalés /@estudiantat.upc.edu
Sofía López Olivares /@estudiantat.upc.edu
Pablo Rosas Roda /@estudiantat.upc.edu

Junio del 2024

ÍNDICE

1. Resumen propuesta.....	3
2. Objetivos tecnológicos del proyecto.....	4
3. Propuesta de proyecto.....	5
4. Paquetes de trabajo.....	6
4.1. PT1 - Uso de NginX como web server y reverse proxy.....	6
4.2. PT2 - Creación del FrontEnd.....	6
4.3. PT3 - Creación del servidor de autenticación.....	6
4.4. PT4 - Creación de los nodos e interacción de los nodos de IPFS.....	7
4.5. PT5 - Kubernetes.....	7
4.6. PT6 - Creación de un sistema de popularidad de archivos.....	7
5. Grupo del proyecto.....	8
5.1. Rol de los participantes.....	8
5.2. Organización y gestión del proyecto.....	8
6. Arquitectura.....	9
7. Implementación.....	11
7.1. Nginx.....	11
7.1.1. Instalación.....	11
7.1.2. Nginx como servidor web.....	12
7.1.3. Https, ssl y Nginx.....	14
7.1.4. Nginx como reverse proxy.....	15
7.1.5. Nginx.conf.....	16
7.2. FrontEnd.....	16
7.2.1. Instalación.....	16
7.2.2. Comandos básicos.....	17
7.2.3. App.js.....	18
7.2.4. LandingPage.tsx.....	19
7.2.5. AboutUs.tsx.....	22
7.2.6. SignUp.tsx.....	23
7.2.7. SignIn.tsx.....	26
7.2.8. StoragePage.tsx.....	27
7.2.8.1. Header.tsx.....	28
7.2.8.2. Sidebar.tsx.....	29
7.2.8.3. Data.tsx.....	32
7.2.8.4. DataShared.tsx.....	48
7.2.8.5. DataTrash.tsx.....	51
7.2.8.6. DataStorage.tsx.....	53
7.2.8.7. StoragePage.tsx.....	59
7.2.9. Tema claro y tema oscuro.....	62
7.3. BackEnd.....	65



7.3.1. Estructura de la Base de Datos.....	65
7.3.2. Enrutamiento y Controladores.....	67
7.3.3. Autenticación y Gestión de Tokens.....	84
7.3.4. Integración con IPFS.....	85
7.3.5. Interacciones del Usuario.....	85
7.3.6. Seguridad y Manejo de Errores.....	86
7.4. IPFS.....	87
7.5. Kubernetes.....	92
7.6. Blockchain.....	97
8. Mejoras a futuro.....	98
8.1. Autoscaler.....	98
8.2. Blockchain.....	99
9. Conclusión.....	100

1. Resumen propuesta

La propuesta de trabajo se enfoca en la creación de un sistema de almacenamiento de datos revolucionario y altamente eficiente. En contraposición a las plataformas convencionales como Google Drive, este sistema se fundamenta en una arquitectura descentralizada, utilizando un cluster privado de IPFS para administrar nodos de almacenamiento distribuido. Esta elección tecnológica no solo asegurará la disponibilidad de los datos en todo momento, sino que también proporcionará redundancia y escalabilidad, características esenciales para un servicio de almacenamiento confiable en el mundo digital actual.

La experiencia del usuario será prioritaria en este proyecto. Los usuarios accederán al sistema mediante una plataforma web con un sólido sistema de autenticación de usuarios. Este portal permitirá a los usuarios subir, descargar y compartir archivos de manera segura y sencilla. La utilización de React para el desarrollo del frontend asegura una interfaz de usuario intuitiva y atractiva, mientras que Node.js y MongoDB se encargan de gestionar las credenciales y los archivos asociados a cada cuenta de usuario. Para dirigir el tráfico y comunicar las diferentes partes del sistema, se implementará Nginx como servidor web central, garantizando una experiencia fluida y sin interrupciones para los usuarios.

Una característica destacada que se propone para futura implementación es la utilización de tecnología blockchain para registrar la popularidad de los archivos compartidos entre usuarios. Esta funcionalidad proporcionará transparencia y seguridad en el intercambio de archivos, permitiendo a los usuarios conocer el alcance y la relevancia de sus contenidos compartidos. En resumen, se espera que este proyecto resulte en un sistema de almacenamiento de datos sólido, escalable y altamente funcional, que satisfaga las demandas de los usuarios y ofrezca una experiencia de usuario excepcional en el entorno digital en constante evolución.

2. Objetivos tecnológicos del proyecto

- Crear un servidor de disco descentralizado utilizando un cluster privado de IPFS para gestionar nodos de almacenamiento, usando IPFS-Cluster para facilitar la implementación
- Dockerizar tanto los nodos de IPFS y IPFS-Cluster para permitir que sean desplegados en diversas máquinas con diversas plataformas
- Usar los nodos previamente dockerizados para permitir la orquestación con Kubernetes y la escalabilidad automática
- Desarrollar una página web con autenticación de usuarios para que estos puedan acceder al servicio y gestionar sus archivos utilizando React y NginX para mostrarla a Internet.
- Implementar el backend para el control de credenciales con Node.js y MongoDB.
- Configurar un servidor web con Nginx para servir la página web y comunicar las diferentes partes del proyecto, como el back-end y la base de datos con el servidor del frontend.
- Configurar NginX para permitir usar HTTPS, con un certificado autofirmado, asegurando la conexión.
- Investigar y potencialmente implementar la tecnología blockchain para crear un registro del uso de los archivos, creando un NFT que rastrea las veces que se descarga o que se comparte un archivo.

3. Propuesta de proyecto

Este proyecto tiene como objetivo crear un sistema de disco, con acceso simple, protegiendo la integridad y la confidencialidad de la información, tanto en el almacenamiento como en el acceso, con un sistema distribuido de almacenamiento de disco, escalable para permitir añadir espacio de almacenamiento de forma simple y sin tener que interrumpir el sistema, a prueba de fallos de caídas de máquinas, usando replicaciones de datos.

Al plantear el problema se vió que IPFS era un candidato a realizar esas características. Además, IPFS es un sistema que tiene una implementación que viene dockerizada, y que sobre el papel es fácil de administrar a través de contenedores. Con ello, falta un sistema de orquestación de contenedores para que IPFS sea realmente escalable de forma automática, así que usamos Kubernetes para encargarse de la orquestación de las instancias de IPFS, creándose y eliminando bajo demanda, e distribuyendo las instancias entre diversas máquinas. En este caso, la replicación de datos se hace a nivel de bloques entre nodos.

Para poder acceder a estos datos, tenemos que crear una página web, con una interfaz moderna y fácil de usar, escrita en React. Para distribuir esta página web usaremos un servidor Nginx en la máquina.

El frontend, para obtener los atributos de los usuarios y los archivos se comunica con un backend robusto (escrito en JavaScript usando la librería express) que se encarga de la autenticación de los usuarios, del almacenamiento de los atributos de los archivos y de implementar características extras de las que se consideraban al inicio del proyecto (el backend se encarga de dar las características de compartición de archivos y también hace de gateway con el frontend para dar los datos de blockchain). Este backend se apoya en una base de datos noSQL (mongoDB) para almacenar quien comparte los archivos a quién y ciertos atributos simples de los archivos.

Para almacenar los índices de popularidad de un archivo (en nuestro caso cuantas veces se ha compartido y descargado) tenemos planteado externalizar esto a un sistema blockchain, que nos permite hacer que estos datos sean provenientes de la blockchain. Para ello, tendremos que implementar un Smart Contract que nos permita hacer deploy y que en blockchain se almacenen esos atributos de todos los archivos.

4. Paquetes de trabajo

Para la correcta realización del proyecto y poder cumplir con las fechas previstas de entrega, decidimos repartir la carga de trabajo en los siguientes 6 paquetes de trabajo.

4.1. PT1 - Uso de NginX como web server y reverse proxy

Tal y como dice el título de este subapartado, para poder tener una correcta comunicación con las diferentes secciones de trabajo y entre las diferentes máquinas que utilizamos, decidimos hacer uso de Nginx. Esta decisión fue tomada en base a que Nginx cumple completamente con las necesidades que teníamos y es capaz de aportar con un gran rendimiento con un consumo de carga relativamente pequeño.

A parte de esto, Nginx es actualmente el paradigma para los servidores web, por lo que, puestos a utilizar alguna tecnología concreta, mejor si es la que se utiliza actualmente en el sector de TI a nivel profesional.

4.2. PT2 - Creación del FrontEnd

Este paquete de trabajo consiste en la creación de un frontend funcional (y visualmente agradable) que nos permita hacer que el usuario interaccione con nuestro sistema. Esto quiere decir que, puesto que nuestro sistema dispone de varios usuarios, necesitaremos interfaces gráficas para el inicio de sesión y el registro a parte de la visualización y gestión de los archivos de cada usuario individual.

Siguiendo el consejo de Félix, decidimos realizar el frontend con React.js que, de nuevo, es actualmente una tecnología muy versátil y polivalente que está muy demandada en el mundo profesional, por lo que consideramos adecuada su utilización.

4.3. PT3 - Creación del servidor de autenticación

Este apartado se basa en la implementación de un sistema de inicio de sesión que permite a los usuarios acceder de forma segura y conveniente a los recursos ofrecidos por la plataforma. Para lograr esta implementación, se ha utilizado una base de datos creada y gestionada mediante Node.js y MongoDB.

El proceso de inicio de sesión se realiza mediante tokens de autenticación, los cuales se generan al verificar las credenciales del usuario. Estos tokens tienen una duración de 24 horas, asegurando la seguridad y validez de la autenticación durante un período adecuado. Una vez que el usuario inicia sesión correctamente, se le proporciona un token que puede utilizar para acceder a los recursos de la plataforma durante el período especificado.

La generación y gestión de estos tokens se lleva a cabo de manera eficiente y segura gracias a Node.js y MongoDB. Node.js se encarga de gestionar la

lógica del sistema de inicio de sesión, mientras que MongoDB se utiliza para almacenar y administrar la información de autenticación de los usuarios. Esta combinación de tecnologías ha permitido crear un sistema de inicio de sesión robusto y confiable que garantiza la seguridad y la comodidad de los usuarios al acceder a los recursos de la plataforma.

4.4. PT4 - Creación de los nodos e interacción de los nodos de IPFS

Con el uso de IPFS queremos plantear la posibilidad de usar un método alternativo para el almacenamiento de datos diferente a lo convencional, ya que en vez de usar un sistema de discos alojado directamente en un disco duro. Para ello, tenemos que configurar los nodos de IPFS de forma que nos permitan comunicarse con ellos de forma simple, que podamos subir y bajar archivos, que los archivos se distribuyan de forma útil entre los nodos, y que la creación de los nodos sea simple. Para ello, se ha creado una configuración inicial, que nos sea fácil de adaptar entre nodos para facilitar la implementación.

4.5. PT5 - Kubernetes.

El uso de Kubernetes para este proyecto surge con el objetivo de dar soporte a IPFS, dándonos de esta forma una alta disponibilidad de los datos, tolerancia a fallos y escalabilidad de forma inherente. En este paquete esperamos poder mejorar el sistema de IPFS creado inicialmente. Kubernetes nos permitirá gestionar los diferentes nodos de almacenamiento que se encuentren en nuestro clúster, de la misma forma que crear nuevos y destruir los ya existentes en caso de ser necesario.

Por lo que el objetivo principal de este paquete de trabajo será conseguir que nuestro sistema sea fácilmente escalable y ampliable a nuestro gusto.

4.6. PT6 - Creación de un sistema de popularidad de archivos.

En este segmento del proyecto, nos enfocamos en la creación de un sistema de popularidad de archivos basado en tecnología blockchain. Para el desarrollo y las pruebas de esta aplicación descentralizada, utilizamos herramientas especializadas como Truffle y Ganache. El propósito principal de este sistema es implementar un contador de popularidad que registre y muestre cuántas veces se han compartido los archivos. Esta funcionalidad busca ofrecer métricas claras y confiables sobre la relevancia y el alcance de los archivos dentro de la plataforma, aprovechando la seguridad y la transparencia que proporciona la blockchain.

5. Grupo del proyecto

5.1. Rol de los participantes

Para la realización de este proyecto decidimos dividirnos los paquetes de trabajo mencionados anteriormente de la siguiente manera:

Paquete de trabajo	Javier	Sergi	Sofía	Pablo
PT1 (NginX)	Responsable			
PT2 (FrontEnd)	Responsable			
PT3 (BBDD)			Responsable	
PT4 (IPFS)		Ayudante		Responsable
PT5 (Kubernetes)	Ayudante	Responsable		
PT6 (Blockchain)			Responsable	Ayudante

El responsable del paquete de trabajo es el encargado de supervisar que todo va correctamente en el desarrollo del paquete y el que realiza la mayor parte del trabajo de ese paquete de trabajo.

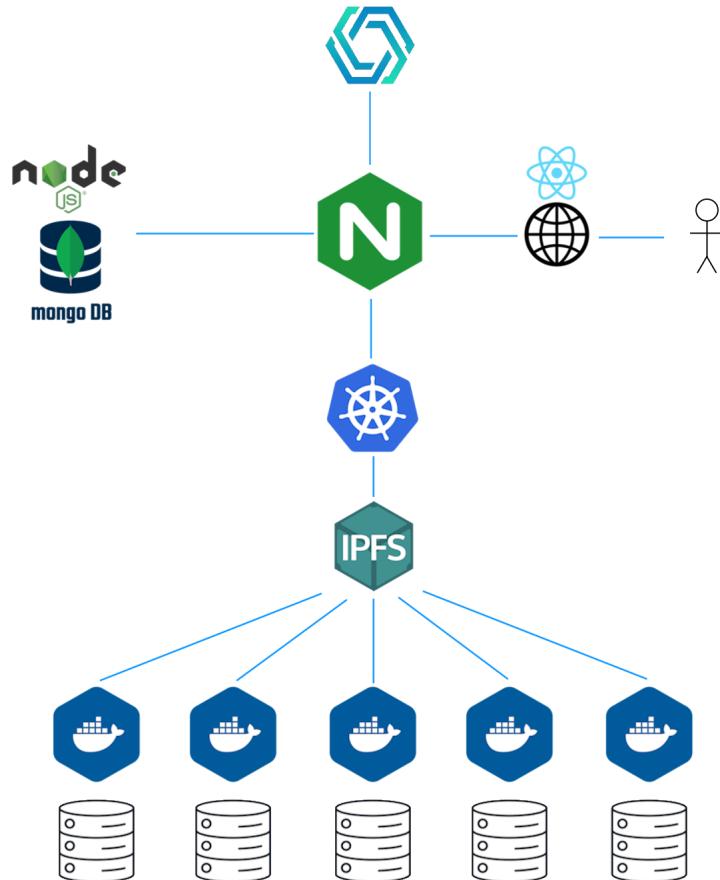
5.2. Organización y gestión del proyecto

Durante el desarrollo de este proyecto el grupo se ha reunido para poder compartir los avances y discutir los problemas de forma semanal. Los miembros del grupo nos hemos reservado una franja horaria cada domingo para este propósito.

Para la correcta organización y planificación del proyecto contamos con un repositorio en GitLab y con un servidor de Discord, por el cual se han realizado las reuniones y las que hemos comentado los diferentes puntos del proyecto, que han ido surgiendo, en diferentes hilos de conversación para tenerlo todo más organizado.

6. Arquitectura

Ahora que ya sabemos cual es el objetivo del proyecto, como queremos hacer su implementación y en qué secciones está dividido, vamos a ver como es el conexiónado interno.



Esquema inicial del proyecto. Marzo del 2024

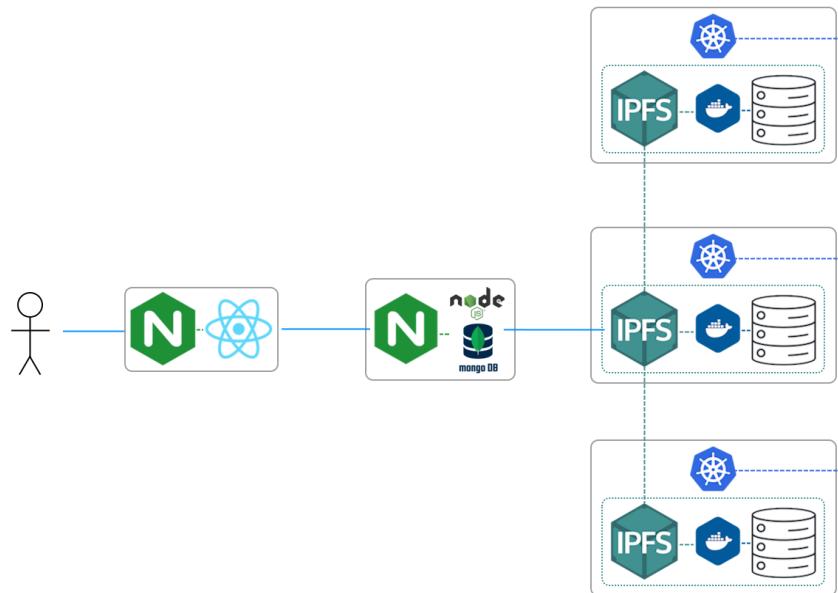
En esta imagen se puede observar la estructura inicial que se pensó y se propuso para el proyecto. Esto incluye el frontend web desde el que interactúa el usuario, un servidor Nginx que sirve de conexión común, un back-end, la comunicación directa con kubernetes y la comunicación de este con el nodo de IPFS y el sistema blockchain colocado arriba del todo.

Este esquema presenta varios problemas dados principalmente por la falta de conocimiento respecto al desarrollo de un sistema que integra diferentes tecnologías como este.

Estos errores pueden ser por ejemplo, Nginx como nodo común de todo que, aunque esto puede llegar a ser posible dentro de una misma máquina, no tendría sentido que no exista una comunicación directa entre frontend y backend.

Otras muestras de estos errores pueden ser, por ejemplo, el sistema blockchain como un nodo a parte en vez de estar colocado al lado del backend representando la integración sobre este. Una última muestra de fallo es la suposición de que IPFS trabaja con un solo nodo independiente y que es este el que gestiona todas las unidades de almacenamiento.

Una vez finalizado el proyecto y habiendo entendido como es el funcionamiento real del sistema que hemos desarrollado se nos presenta el siguiente esquema:



Esquema final del proyecto. Mayo del 2024

En esta imagen nos encontramos con 5 recuadros grises. Estos representan las 5 máquinas virtuales utilizadas.

Si analizamos el esquema, y por tanto la estructura final del proyecto, tenemos el usuario que hace peticiones a una web, en nuestro caso <https://10.4.41.67>. Este contenido es mostrado y gestionado por Nginx, que, de cara al frontend, está actuando como servidor web.

Si el usuario, o el propio sistema, necesitan consultar información sobre el usuario, desde el mismo frontend se realizarán peticiones al backend que, gracias a que utilizamos un Nginx a modo de reverse proxy en la máquina virtual que contiene el backend, podemos ofrecer al usuario una conexión totalmente segura y encriptada con ssl.

Si el sistema necesita hacer algún cambio sobre los ficheros del usuario, ya sea subir un fichero nuevo, descargar uno existente, eliminar o cambiarle el nombre, el backend se comunicará con uno de los nodos de IPFS responsable de la gestión del almacenamiento.

Este nodo de IPFS, en caso de poder realizar la operación requerida, operará como sea necesario. En caso de no poder, por ejemplo porque sea una petición de descarga de un fichero que no está localizado en ese nodo, será el propio nodo el que se encargue de comunicarse con el resto de nodos para poder obtener los datos y operar como sea necesario.

Adicionalmente disponemos de un kubernetes sobre los nodos de IPFS que son capaces de realizar un balanceo de carga y escalar el sistema.

Como se puede observar, el sistema blockchain no se muestra en ninguna máquina, esto es debido a que se consiguieron hacer pruebas en local pero debido a los problemas que encontramos a la hora de hacer la migración a las máquinas virtuales, decidimos descartar la tecnología.

7. Implementación

En este punto veremos un poco más en detalle cómo hemos realizado la implementación del sistema y como se ha trabajado con las diferentes tecnologías mencionadas anteriormente.

7.1. Nginx

Tal y como se ha comentado previamente, hacemos uso de Nginx tanto de servidor web como de reverse proxy. A continuación veremos más en detalle ambas implementaciones.

7.1.1. Instalación

Decidimos utilizar Nginx como servidor web gracias a su gran rendimiento y su bajo coste que, dado que el proyecto se despliega en unas máquinas virtuales limitadas en recursos, que consuman pocos de estos era algo indispensable.

Para hacer uso de este servidor, lo primero fue instalar el programa. Todo y que se pueden instalar versiones más recientes desde el repositorio propio de Nginx, por la facilidad y por la poca diferencia de características, decidimos instalar y usar la versión que se instala por defecto a través del instalador apt.

Para esto es necesario ejecutar:

```
$ sudo apt-get install nginx
```

Una vez instalado el programa, es necesario saber que su ejecución funciona a través del propio daemon de Nginx, por lo que no hay ningún binario a ejecutar. En su defecto usaremos el control que nos proporciona systemctl. Para esto existen los diferentes comandos:

```
$ sudo systemctl start nginx
```

```
$ sudo systemctl stop nginx
```

```
$ sudo systemctl restart nginx
```

Como se puede observar, los comandos son bastante descriptivos por sí mismos.

A parte de estos comandos, si quisieramos comprobar el estado del daemon podemos ejecutar:

```
$ sudo systemctl status nginx
```

Con esto obtendríamos un output de este estilo:

```
alumne@dugtrio:~$ sudo systemctl status nginx
[sudo] password for alumne:
● nginx.service - A high performance web server and a reverse proxy server
  Loaded: loaded (/lib/systemd/system/nginx.service; enabled; vendor preset: enabled)
  Active: active (running) since Wed 2024-05-22 21:03:48 CEST; 1 weeks 0 days ago
    Docs: man:nginx(8)
   Process: 118347 ExecStartPre=/usr/sbin/nginx -t -q -g daemon on; master_process on; (code=exited, status=0)
   Process: 118359 ExecStart=/usr/sbin/nginx -g daemon on; master_process on; (code=exited, status=0)
 Main PID: 118360 (nginx)
   Tasks: 3 (limit: 4683)
  Memory: 4.2M
  CGroup: /system.slice/nginx.service
          └─118360 nginx: master process /usr/sbin/nginx -g daemon on; master_process on;
              ├─118361 nginx: worker process
              ├─118362 nginx: worker process
              └─118362 nginx: worker process

May 22 21:03:48 dugtrio systemd[1]: Starting A high performance web server and a reverse proxy serv...
May 22 21:03:48 dugtrio systemd[1]: Started A high performance web server and a reverse proxy serv...
lines 1-16/16 (END)
```

En este output podemos comprobar información como si el daemon está activo o no, el pid del proceso y recursos consumidos entre otras cosas.

En este momento, si accedemos a través del navegador a localhost (o hacemos un curl) podremos ver algo parecido a esto:

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

7.1.2. Nginx como servidor web

Ahora que ya tenemos nginx instalado y funcionando, podemos configurarlo para que muestre el contenido que queremos.

Para poder conseguir esto, primero tenemos que entender la estructura de directorios que dispone Nginx, eso es:

- **/etc/nginx**: directorio raíz de Nginx.
- **/etc/nginx/conf.d**: directorio de configuración por defecto.
- **/etc/nginx/sites-enabled**: directorio que contiene softlinks que apuntan al directorio /etc/nginx/sites-available.
- **/etc/nginx/sites-available**: directorio que contiene los ficheros de configuración de cada servicio que ofrecemos.

El funcionamiento de los directorios sites-enabled y sites-available, es decir, el porque en uno hay softlinks y en otro los ficheros de configuración, es porque Nginx cargará toda la configuración que encuentre en en /etc/nginx/sites-enabled, esto quiere decir que podemos tener más de un servicio creado y configurado en sites-available pero hasta que no creamos el softlink en sites-enabled será completamente inaccesible desde el exterior, en gran parte porque ni siquiera nginx “conoce” de su existencia.

Ahora que ya tenemos nociones básicas del funcionamiento de Nginx, veamos el archivo de configuración del frontend:

```
server {  
    listen 443 ssl default_server;  
    include snippets/self-signed.conf;  
    include snippets/ssl-params.conf;  
  
    root /var/www/upccloud/html;  
  
    index index.html index.htm index.nginx-debian.html;  
  
    server_name upccloud.com www.upccloud.com;  
  
    location / {  
        try_files $uri /index.html;  
    }  
}  
  
server {  
    listen 80;  
    server_name upccloud.com www.upccloud.com;  
    return 301 https://$host$request_uri;  
}
```

En esta imagen podemos ver, aparte de la poca configuración necesaria para mostrar el frontend, la estructura de los bloques de servidor que utiliza Nginx. Esta siempre tiene esta estructura:

```
server {  
    listen ...;  
    server_name _;  
}
```

Esto claramente indica un servidor que está escuchando al puerto que le digamos.

Aunque según la imagen parezca que el campo server_name es obligatorio, este se puede dejar con el carácter ‘_’ sin ningún problema. Este campo se utiliza para decidir qué bloque de servidor se escogerá para atender la petición.

Vamos a analizar más en detalle la implementación de nuestro caso.

En el primer bloque de servidor podemos observar cómo utilizamos el puerto 443 para atender peticiones web entrantes, esto es porque disponemos de una conexión cifrada por ssl, más adelante veremos más en detalle esto.

Las siguientes dos líneas hacen referencia a la configuración ssl utilizada.

A continuación tenemos el campo “root path/to/files;”, esto nos indica el directorio raíz en el que Nginx buscará que ficheros mostrar al usuario.

Estos ficheros vienen definidos en la siguiente línea, “index * .html”, que, dado que lo que mostramos es una página web, indicaremos a nginx que utilice nuestro index.html o variantes similares.

El punto más importante posiblemente sea el campo location / {}.

Esto nos quiere decir que cualquier petición que llegue a / (o a cualquier sub ruta dentro de /, es decir, todo) se mostrará el contenido indicado por el fichero index.html.

Por último tenemos un segundo bloque de servidor que escucha al puerto 80. Esto es únicamente para poder crear una redirección automática hacia el puerto seguro, el 443, en caso de intentar conectarse al puerto 80.

7.1.3. **Https, ssl y Nginx**

Para poder activar la conexión por https y dado que al tener el servicio dentro de la VPN de la FIB no podemos utilizar ninguna autoridad de certificación, decidimos optar por crear un certificado autofirmado.

En uno de los ficheros incluidos en la configuración mostrada previamente, en el /etc/nginx/snippets/self-signed.conf, tenemos el siguiente código:

```
ssl_certificate /etc/ssl/certs/nginx-selfsigned.crt;
ssl_certificate_key /etc/ssl/private/nginx-selfsigned.key;
```

Esto simplemente indica el path donde se encuentran tanto el certificado como la key generados a través de openssl.

En el segundo fichero, /etc/nginx/snippets/ssl-params.conf, tenemos los siguientes parámetros:

```
ssl_protocols TLSv1.2;
ssl_prefer_server_ciphers on;
ssl_dhparam /etc/ssl/certs/dhparam.pem;
ssl_ciphers
ECDHE-RSA-AES256-GCM-SHA512:DHE-RSA-AES256-GCM-SHA512:E
CDHE-RSA-AES256-GCM-SHA384:DHE-RSA-AES256-GCM-SHA384:EC
DHE-RSA-AES256-SHA384;
ssl_ecdh_curve secp384r1; # Requires nginx >= 1.1.0
ssl_session_timeout 10m;
ssl_session_cache shared:SSL:10m;
```

```
ssl_session_tickets off; # Requires nginx >= 1.5.9
# ssl_stapling on; # Requires nginx >= 1.3.7
# ssl_stapling_verify on; # Requires nginx >= 1.3.7
resolver 8.8.8.8 8.8.4.4 valid=300s;
resolver_timeout 5s;
add_header X-Frame-Options DENY;
add_header X-Content-Type-Options nosniff;
add_header X-XSS-Protection "1; mode=block";
```

Estos parámetros indican medidas de seguridad típicas de https y son completamente modificables.

Para que esto funcione correctamente, también tenemos que crear, de nuevo con openssl, el fichero dhparam.pem, el cual contiene información sobre cómo nginx debe tratar el protocolo diffie hellman.

Un problema que nos encontramos al configurar todo esto fue que como la comunicación con el backend se encuentra en otra máquina distinta, y esta en principio funcionaba sin https, el propio navegador nos devolvía un error de seguridad de CORS e impedía la conexión por completo. Debido a esto, nos vimos obligados a configurar otro Nginx en la máquina virtual del backend, esta vez como reverse proxy, y realizar la misma configuración de ssl que se acaba de explicar.

7.1.4. Nginx como reverse proxy

Como se ha explicado previamente, fue necesario habilitar un Nginx con ssl en la máquina del backend para poder disponer de conexión https de forma completa.

La configuración de Nginx utilizada en este caso es la siguiente:

```
upstream backend {
    server 10.4.41.68:3001;
}

server {
    listen 443 ssl;
    include snippets/self-signed.conf;
    include snippets/ssl-params.conf;
    location / {
        proxy_pass http://backend/;
        proxy_set_header Host $host;
    }
}

server {
    listen 80;
    return 302 https://10.4.41.68$request_uri;
}
```

Esta configuración, al igual que la anterior, tiene un bloque de servidor principal escuchando al puerto 443, habilitado con toda la configuración de ssl y un segundo bloque de servidor que redirecciona al puerto seguro.

La parte diferente en este caso es que, en vez de mostrar un contenido index.html indicado por la directriz root, tenemos un “proxy_pass <http://backend>” el cual redirige al server definido dentro del bloque “upstream backend” y establece un header con el “proxy_set_header Host \$host”. Esto último es para redireccionar las diferentes urls a las que se generan peticiones hacia el propio backend, es decir, si desde el frontend hacemos una petición a \$backend_url/archivo/getall, esta directriz envia la petición http con /archivo/getall al final de la url en vez de omitirlo.

7.1.5. Nginx.conf

Aunque parezca que toda la configuración ya está acabada, lo cierto es que, dado el funcionamiento de nuestro sistema, es decir, poder subir archivos, fue necesario modificar el fichero /etc/nginx/nginx.conf añadiendo la siguiente línea dentro del bloque http {}:

```
client_max_body_size 10G;
```

Esto indica el tamaño máximo de fichero que puede subir un usuario y fue necesario añadirlo ya que de lo contrario, Nginx no es capaz de tratar con ficheros de más de 1,5MB.

7.2. FrontEnd

Respecto al frontend, hacemos uso de React.js en todas las vistas de nuestro sistema. Al ser una tecnología muy versátil y completa, nos permite realizar cualquier tipo de interfaz con cualquier componente visual que queramos.

7.2.1. Instalación

Para poder utilizar el framework de React.js, primero tenemos que instalar las librerías y dependencias necesarias. Cómo React depende de Node.js, instalaremos la última versión de este.

Para esto, necesitaremos instalar el NodeVersionManager, ya que a través de apt no conseguiremos la versión actualizada.

```
curl -o-
https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.3/install.sh | bash
```

```
source ~/.bashrc
```

Con estos comandos conseguiremos instalar el nvm para, a través de aquí, instalar la versión correspondiente de Node.js.

Para instalar la versión de Node.js que queramos, ejecutaremos el siguiente comando:

```
nvm install v20.11.1
```

Ahora que ya tenemos instalado Node.js, para crear nuestra aplicación en React.js es tan simple como ejecutar:

```
npx create-react-app upccloud-deploy
```

En nuestro caso el nombre de la aplicación es upccloud-deploy.

En caso de querer abrir un proyecto de React ya existente, como puede ser el nuestro, será necesario entrar a la carpeta upccloud-deploy y ejecutar:

```
npm install
```

Esto instalará todos los paquetes y librerías utilizadas dentro del proyecto de forma automática.

7.2.2. Comandos básicos

Para poder probar el correcto funcionamiento de la aplicación existe el siguiente comando:

```
npm start
```

Este comando lo que hará será ejecutar la aplicación desde el código fuente y abrir una conexión al servicio a través de la ip local y el puerto por defecto, es decir, localhost:3000. Por lo que si nos conectamos a través del navegador a esta dirección, podremos ver la aplicación en ejecución y, lo más importante, actualiza los cambios en caliente, es decir, no es necesario parar el servicio y volver a ejecutarlo en caso de haber cambiado algo de código.

De cara al final del proyecto y, por tanto, querer hacer un build para eliminar dependencias no utilizadas y optimizar el código de forma automática gracias al compilador de npm, ejecutaremos el siguiente comando:

```
npm run build
```

Esto nos creará una carpeta “build” dentro de la carpeta root de nuestro proyecto. Aquí dentro tendremos todos los archivos necesarios para poder ejecutar la aplicación en un servidor web y, por tanto, son los ficheros que se necesitan poner en la carpeta indicada por el root path del bloque de servidor de Nginx.

7.2.3. App.js

Por defecto, nuestro index.js lo único que hace es obtener y mostrar el contenido del fichero App.js. Por este motivo y por simplicidad, en vez de modificar el contenido de index.js, modificaremos el contenido de App.js.

```
upccloud-deploy > src > JS App.js > ...
1  import { BrowserRouter as Router, Route, Routes} from 'react-router-dom';
2  import { default as LandingPage } from './landing-page/LandingPage.tsx';
3  import { default as SignUp } from './sign-up/SignUp.tsx';
4  import { default as SignIn } from './sign-in/SignIn.tsx';
5  import { default as StoragePage } from './storage-page/StoragePage.tsx';
6  import { default as AboutUs } from './about-us/AboutUs.tsx';
7
8  export default function App() {
9    return (
10      <Router>
11        <Routes>
12          <Route path="/" element={<LandingPage />}></Route>
13          <Route path="/sign-up" element={<SignUp />}></Route>
14          <Route path="/sign-in" element={<SignIn />}></Route>
15          <Route path="/storage" element={<StoragePage />}></Route>
16          <Route path="/about-us" element={<AboutUs />}></Route>
17        </Routes>
18      </Router>
19    );
20 }
```

En esta imagen se puede observar el contenido completo del fichero App.js. Este fichero es el encargado de enrutar los diferentes contenidos que tenemos dentro de la jerarquía de nuestra web. Para poder realizar esto, utilizamos unos componentes de React. Estos son BrowserRouter, Route y Routes.

Como se puede ver en la implementación, estos componentes se utilizan de forma similar a los componentes de html.

En el componente Route, se establece el path a partir del cual se accede al contenido, es decir la url del navegador y el contenido a mostrar. Este contenido no deja de ser otra clase diferente de React, la cual tiene un *export default* para poder realizar un import desde este fichero.

7.2.4. LandingPage.tsx

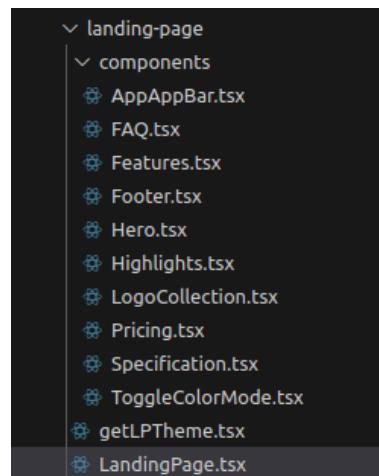
Decidimos crear una landing page para poder explicar un poco la estructura de nuestro proyecto y las ventajas que presenta. Esto fue para poder tener un producto presentable como corporativo o empresarial.

Nuestra página de landing tiene la siguiente estructura:

```
19  export default function LandingPage() {
20    const start_mode = (localStorage.getItem("mode") === null ? 'light' : localStorage.getItem("mode"));
21    //@ts-ignore
22    const [mode, setMode] = React.useState<PaletteMode>(start_mode);
23    const LPtheme = createTheme(getLPTheme(mode));
24
25    const toggleColorMode = () => {
26      setMode((prev) => (prev === 'dark' ? 'light' : 'dark'));
27      localStorage.setItem("mode", mode === 'light' ? 'dark' : 'light');
28    };
29
30    return [
31      <ThemeProvider theme={LPtheme}>
32        <CssBaseline />
33        <AppBar mode={mode} toggleColorMode={toggleColorMode} />
34        <Hero />
35        <Box sx={{ bgcolor: 'background.default' }}>
36          <LogoCollection />
37          <Features />
38          <Divider />
39          <Specification/>
40          <Divider />
41          <Highlights />
42          <Divider />
43          <Pricing />
44          <Divider />
45          <FAQ />
46          <Divider />
47          <Footer />
48        </Box>
49      </ThemeProvider>
50    ];
51 }
```

Como se puede ver, hay muy poco código para todo lo que mostramos. Esto es principalmente por la encapsulación en diferentes componentes para hacer el código más sencillo y portable.

En la función return se pueden ver todos los componentes utilizados y su estructura para la realización de la página inicial.





Aquí se muestra la jerarquía de componentes utilizada para la landing page.

El resultado final es el siguiente:

The screenshot shows the main landing page of the UPC Cloud website. At the top, there is a navigation bar with links for Features, Specification, Highlights, Pricing, and FAQ, along with Sign in and Sign up buttons. The main title "Welcome to UPC Cloud" is prominently displayed in large, bold, black and blue text. Below the title, a subtitle reads: "Explore our cutting-edge online web-based storage system, delivering high-quality solutions tailored to your needs. Elevate your experience with top-tier features and services." A "Start now" button is visible, followed by a note: "By clicking "Start now" you agree to our Terms & Conditions." On the left, a sidebar menu includes options for New, My Files, Shared, and Trash, with a total storage usage of 17.69MB / 1.00GB. The central area features a "Storage space used" section with a progress bar and a "Storage progression on time" chart showing used space in MB and total space in MB over time.

Vista inicial de la landing page.

The screenshot shows the "Product features" page of the UPC Cloud website. At the top, there is a navigation bar with links for Features, Specification, Highlights, Pricing, and FAQ, along with Sign in and Sign up buttons. The main heading is "Product features". Below it, a sub-headline states: "UPC Cloud is a highly scalable and redundant system to store all your files safely. Explore and learn the features and the incredible capabilities of our product." The page lists four features with corresponding icons and descriptions: "Dashboard" (Discover and use our new user friendly interface. Designed to maximize to the fullest the user experience of our customers. Completely different from Google Drive.), "Dark theme" (Easy dark theme switch integrated natively into our product for a better user comfort.), "Share files" (Share your files easily with other users using the platform and track the popularity with our blockchain system-based sharing record.), and "Highly scalable" (Thanks to the use and implementation of our system with IPFS, Docker and Kubernetes, we obtained a highly scalable and redundant system capable of creating a new storage node or duplicating an existing one completely automatic.). To the right, there is a preview window showing a screenshot of the UPC Cloud file management interface.

Vista de las funcionalidades principales del producto.



The diagram illustrates the initial project structure. At the top, a central Node.js icon (green hexagon with a white 'N') is connected to a MongoDB icon (green cylinder with a white database symbol) on the left and a Docker icon (blue hexagon with a white ship symbol) on the right. Below this, a React.js icon (blue hexagon with a white atom symbol) is connected to the Docker icon. Further down, an IPFS icon (teal hexagon with a white chain symbol) is connected to five smaller storage icons (each showing a cylinder with a white database symbol). At the bottom, a Kubernetes icon (blue hexagon with a white triangle symbol) is connected to the IPFS icon.

nginx
web server. Nginx lets us serve the corresponding content to our customers and thanks to its low resource usage, we are able to attend thousands of simultaneous users working on their account.

React.js
Using React as our main programming language to implement the front-end, we are able to provide a comfortable and cozy fully customizable visual interface to maximize the productivity of our clients.

MongoDB & Node.js
We use MongoDB to implement our non-SQL database, in which we store all the necessary data to maintain the user accounts and its files. The implementation responsible of managing this database is done with the powerful Node.js, which makes the database highly scalable and minimizes the wait time thanks to the asynchronous calls that implements.

Docker
The storage in our disk system is managed by Docker, which lets us fully automate the configuration needed to function with IPFS.

Inter Planetary File System
With IPFS we are capable of having a fully secure and distributed storage system, making our storage system more resilient and redundant.

Kubernetes
Thanks to Kubernetes, our storage system is fully automated, and

Vista de la estructura inicial del proyecto.

Pricing
Explore our different tiers and choose the one it suits you best.

Free
\$0 per month

- Personal account included
- 2 GB of storage
- Email support

[Sign up for free](#)

Professional Recommended
\$15 per month

- Personal account included
- 30 GB of storage
- Help center access
- Priority email support
- Dedicated team
- Best deals

[Start now](#)

Private hosting
\$10 per month

- Private storage in your local system
- Our team ensures the correct installation in your machines
- Help center access
- Phone & email support
- In-site support

[Contact us](#)

Vista del plan de precios disponibles.

7.2.5. AboutUs.tsx

Esta página está enfocada a mostrar algo más de información sobre los integrantes del grupo.

El código desarrollado es el siguiente:

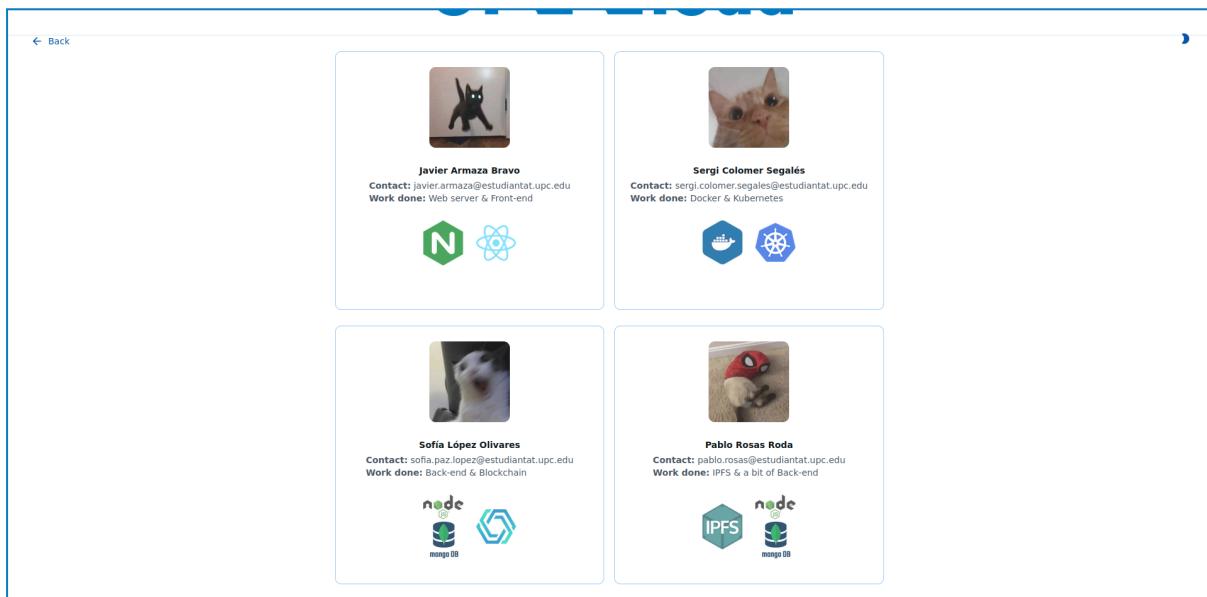
```
139          <Card
140            variant="outlined"
141            sx={{
142              alignItems: 'center',
143              alignSelf: 'center',
144              p: 3,
145              height: '410px',
146              width: '23%',
147              background: 'none',
148              borderColor: (theme) => {
149                if (theme.palette.mode === 'light') {
150                  return 'primary.light';
151                }
152                return 'primary.dark';
153              },
154            }}
155          >
156          <Box
157            sx={{
158              width: '100%',
159              display: 'flex',
160              flexDirection: { xs: 'column', md: 'column' },
161              alignItems: { md: 'center' },
162              gap: 2.5,
163            }}
164          >
165          <Box
166            sx={{
167              color: 'primary.main'
168            }}
169          >
170          <img src={javier} style={profileStyle} alt='javier'/>
171        <Box sx={{ textTransform: 'none' }}>
172          <Typography
173            color="text.primary"
174            variant="body2"
175            fontWeight="bold"
176            align='center'
177            >
178              Javier Armaza Bravo
179            </Typography>
180            <Typography
181              color="text.secondary"
182              variant="body2"
183              align='left'
184              sx={{ my: 0.5 }}
185            >
186              <b>Contact:</b> javier.armaza@estudiantat.upc.edu <br/>
187              <b>Work done:</b> Web server & Front-end
188            </Typography>
189          </Box>
190          <Box sx={{ display:'flex', alignSelf:'center', alignItems:'center'}}>
191            <img src={nginxLogo} style={logoStyle} alt='nginx'/>
192            <img src={frontLogo} style={logoStyle} alt='react'/>
193          </Box>
194        </Box>
195      </Card>
```

Esta implementación utiliza principalmente el componente *Card*, este componente, a parte de tener su configuración visual correspondiente, tiene dentro diversos componentes *Box*. El motivo de esto es para poder utilizar todo el espacio disponible y ser capaces de alinear las imágenes y textos independientemente del tamaño de estas.

Dentro de uno de estos componentes *Box*, están localizados los componentes responsables de la imagen de perfil seleccionada, el texto descriptivo, y las imágenes de la tecnología desarrollada.

Esta estructura se repite un total de 4 veces, una por cada integrante del grupo.

El resultado final es el siguiente:



7.2.6. SignUp.tsx

Este código es el encargado de obtener los datos introducidos por el nuevo usuario, comprobar su validez, y mandar una petición de registro al backend.

La obtención de datos se realiza mediante la siguiente función:

```

82 | const handleSubmit = (event: React.FormEvent<HTMLFormElement>) => {
83 |   event.preventDefault();
84 |   if (!validateInputs()) return;
85 |   const data = new FormData(event.currentTarget);
86 |   registerSup(data.get('name') as string, data.get('email') as string, data.get('password') as string);
87 |

```

Esto lo que hace es validar los valores introducidos por el usuario que, en caso de ser válidos, se obtiene el formulario y se llama a la función auxiliar de llamada al backend.

La función encargada de validar los inputs es la siguiente:

```
39  const validateInputs = () => {
40    const email = document.getElementById('email') as HTMLInputElement;
41    const password = document.getElementById('password') as HTMLInputElement;
42    const name = document.getElementById('name') as HTMLInputElement;
43
44    let isValid = true;
45
46    if (!email.value || !/\S+@\S+\.\S+/.test(email.value)) {
47      setEmailError(true);
48      setEmailErrorMessage('Please enter a valid email address.');
49      isValid = false;
50    } else {
51      setEmailError(false);
52      setEmailErrorMessage('');
53    }
54
55    if (!password.value || password.value.length < 4) {
56      setPasswordError(true);
57      setPasswordErrorMessage('Password must be at least 4 characters long.');
58      isValid = false;
59    } else {
60      setPasswordError(false);
61      setPasswordErrorMessage('');
62    }
63
64    if (!name.value || name.value.length < 1) {
65      setNameError(true);
66      setNameErrorMessage('Name is required.');
67      isValid = false;
68    } else {
69      setNameError(false);
70      setNameErrorMessage('');
71    }
72
73    return isValid;
74  };
```

Esta función obtiene los datos directamente de los elementos de input declarados en el documento, es decir, en el elemento general donde estos están declarados.

Las comprobaciones realizadas son básicas, es decir, un correo electrónico válido (aunque no comprobamos si el correo existe realmente o no), una contraseña de al menos 4 caracteres y cualquier nombre introducido.

La función de soporte de contacto al backend es la siguiente:

```

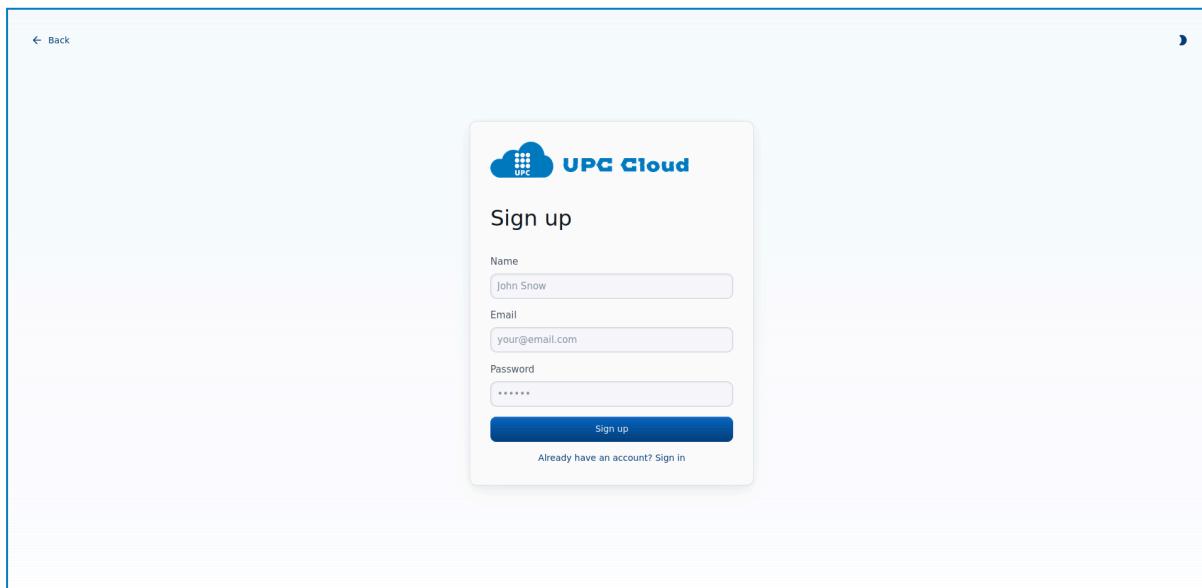
89  const registerSup = async (name, email, password) => {
90    const userData = {
91      nombre: name,
92      clave: password,
93      correo: email
94    };
95    try {
96      const response = await fetch(`${backendUrl}/usuario/crearusuario`, {
97        method: "POST",
98        headers: {
99          "Content-Type": "application/json",
100        },
101        body: JSON.stringify(userData),
102      });
103      if (response.ok) {
104        console.log("register complete");
105
106        // @ts-ignore
107        window.location = '/sign-in';
108      } else {
109        alert("An unexpected error occurred while trying to register the user. Please try again later.");
110      }
111    } catch (error) {
112      console.error("Error while registering the new user:", error);
113      alert("An unexpected error occurred while trying to register the user. Please try again later.");
114    }
115  };
116

```

Es una función que simplemente recibe los datos por parámetro, los encapsula dentro de un JSON y gracias a la funcionalidad de fetch, hacemos una petición http a una url configurada. Esta petición tiene el método POST, el header contiene un “application/json” y el mensaje es el JSON formateado de los datos mencionados previamente.

Esta función dependiendo de la respuesta del servidor del backend, realiza una cosa u otra. En caso de recibir una respuesta correcta, es decir, se ha registrado el usuario satisfactoriamente, redirige al usuario de forma automática a la página de inicio de sesión. En caso contrario, devuelve un error.

El resultado final es el siguiente:



7.2.7. SignIn.tsx

De forma muy similar a la página de registro, en la página de inicio de sesión también disponemos de una función de obtención de datos, una de validación de datos y la función auxiliar de comunicación con el backend.

Debido a la gran similitud entre las dos primeras funciones, solo explicaremos la de comunicación al backend.

```
49 const LoginSup = async (email, password) => {
50   const userData = {
51     correo: email,
52     clave: password
53   };
54   try {
55     const response = await fetch(`${backendUrl}/usuario/login`, {
56       method: "POST",
57       headers: {
58         "Content-Type": "application/json",
59       },
60       body: JSON.stringify(userData),
61     });
62     if (response.ok) {
63       const data = await response.json();
64       //console.log(data.message);
65       localStorage.setItem('token', data.message);
66       localStorage.setItem('email', userData.correo);
67       // @ts-ignore
68       window.location = '/storage';
69     } else {
70       setPasswordError(true);
71       setPasswordErrorMessage('Please enter the correct password.');
72       //window.location = '/storage';
73     }
74   } catch (error) {
75     console.error("Error at login:", error);
76     alert("An unexpected error occurred while trying to login to your account. Please try again later.");
77   }
78 }
```

Esta función, al igual que la mencionada en la página anterior, recibe los datos por parámetro, los formatea dentro de un JSON y hace una petición a la url del backend correspondiente.

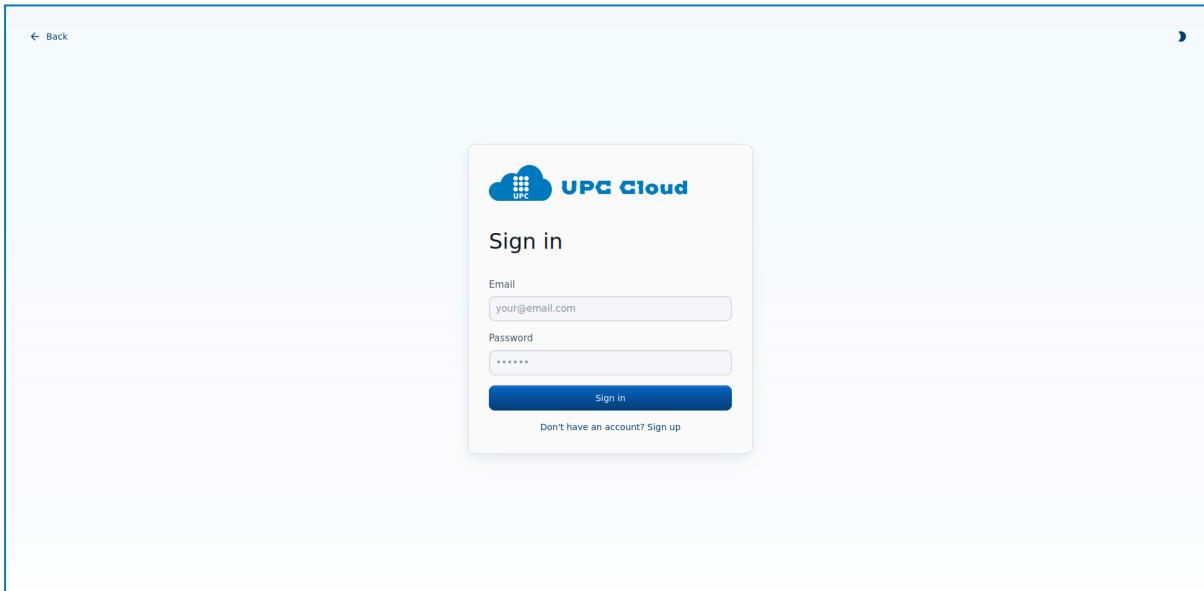
En caso de recibir una respuesta correcta, es decir, el usuario ha introducido las credenciales correctamente, se guarda en el local storage del navegador el correo del usuario y el token enviado por el backend a la hora de iniciar sesión. Necesitamos guardar estos datos en el local storage del navegador ya que, como esto lo recibimos en la url /sign-up, al redireccionar a /storage perderíamos estos datos de no ser por el almacenamiento en el navegador del usuario.

Esto nos será de utilidad a la hora de cargar los archivos del usuario correspondiente y añadir una capa de seguridad extra a nuestro sistema ya que, para poder visualizar los archivos concretos de un usuario, hay que saber (tener guardado en el local storage) tanto el correo del usuario como el token asociado a ese correo.

Para más seguridad, el token está configurado para que caduque al cabo de un cierto tiempo después de haber sido generado.

Una vez guardado estos atributos, se redirige al usuario de forma automática a la página gestora de archivos.

El resultado final es el siguiente:



7.2.8. StoragePage.tsx

La página de storage es la responsable de mostrar al usuario todos los ficheros que tiene disponibles en el sistema de almacenamiento. Esto quiere decir que esta página es la que reúne todas la funcionalidades de nuestro sistema como, por ejemplo, subir, descargar y compartir ficheros o mostrar estadísticas. Por todo esto, esta es la implementación más compleja y con más interacción con el usuario en la parte del frontend.

Para la implementación de la vista de esta página se han utilizado 2 componentes fijos y uno variable. Estos componentes fijos son el header y el side bar.

7.2.8.1. Header.tsx

Este componente simplemente está de forma “estática” en la parte superior de la interfaz y su código es el siguiente:

```

90  function Header ({ mode, toggleColorMode, setInputTextState}: HeaderProps) {
91
92      function handleChange (e) {
93          setInputTextState(e.target.value)
94      }
95
96      return (
97          <HeaderContainer>
98              <HeaderLogo>
99                  <img src={logo} alt="UPC Cloud" />
100                 <span>UPC Cloud</span>
101             </HeaderLogo>
102             <Search>
103                 <SearchIconWrapper>
104                     <SearchIcon />
105                 </SearchIconWrapper>
106                 <StyledInputBase
107                     placeholder="Search..."
108                     inputProps={{ 'aria-label': 'search' }}
109                     onChange={handleChange}
110                 />
111             </Search>
112             <div style = {{position:'relative', left:'169px', top:'0px'}}>
113                 <ToggleColorMode mode={mode} toggleColorMode={toggleColorMode} />
114             </div>
115         </HeaderContainer>
116     )
117 }
118
119 export default Header;

```

Tal y como se puede observar, tenemos la imagen del logo dentro de un componente tipo *HeaderLogo*. Este componente es un componente propio y tiene los siguientes parámetros básicos:

```

20  const HeaderLogo = styled.div` 
21      display: flex;
22      align-items: center;
23      img {
24          width: 40px;
25      }
26      span{
27          font-size: 22px;
28          margin-left: 10px;
29          color: gray;
30      }

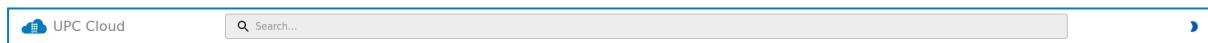
```

Estas pocas líneas de código simplemente establecen el tamaño de la imagen y el formato del texto utilizado.

Posterior a este componente, tenemos la implementación de la barra de búsqueda, esta está formada por el icono de búsqueda y por un campo de input, el cual al detectar un cambio obtiene el valor escrito y lo guarda en una variable para poder usarla en otros componentes dentro de esta página.

Adicionalmente, este componente también dispone del botón para cambiar de tema claro a oscuro y viceversa.

El resultado de este componente es el siguiente:



7.2.8.2. Sidebar.tsx

El código implementado en este componente es el responsable de, principalmente, permitir al usuario de cambiar entre las diferentes vistas de los componentes variables y la posibilidad de subir diferentes tipos de archivos a los usuarios.

Si revisamos el código, lo primero que nos encontramos es el componente *Modal*.

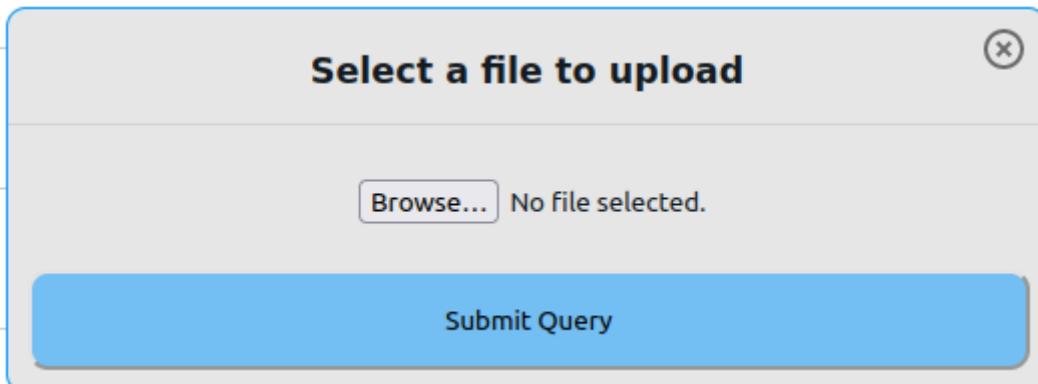
```

191 |   <>
192 |   <Modal open={open} onClose={() => setOpen(false)}
193 |     style={{
194 |       position: 'absolute',
195 |       top: '40vh',
196 |       left: '40vw',
197 |       height: '193px',
198 |       width: '520px',
199 |       backgroundColor: mode === 'light'
200 |         ? 'rgba(230, 230, 230, 1)'
201 |         : 'rgba(30, 30, 30, 1)',
202 |       border: '1px solid',
203 |       borderRadius: '10px',
204 |       borderColor: 'rgba(0,150,255,1)'}}>
205 |   <div>
206 |
207 |     <form onSubmit={handleUpload} style={{position: 'relative'}}>
208 |
209 |       <ModalHeading>
210 |         <h3>Select a file to upload</h3>
211 |       </ModalHeading>
212 |       <ModalBody>
213 |         {uploading ?
214 |           <>
215 |             <h4 style={{textAlign: 'center'}}>Uploading...</h4>
216 |             <Box sx={{ width: '92%' }} style={{position: 'relative',
217 |               alignItems: 'center',
218 |               margin: '20px'}}>
219 |               <LinearProgress style={{height:'20px',borderRadius: '10px'}}/>
220 |             </Box>
221 |           : (
222 |             <div style={{
223 |               position: 'relative',
224 |               alignItems: 'center',
225 |               top: '7px',
226 |               left: '155px',
227 |               margin: '20px'
228 |             }}>
229 |               <input type="file" onChange={handleFile}/>
230 |             </div>
231 |             <input type="submit" style={{
232 |               margin:'10px 10px 10px 10px',
233 |               height:'50px',
234 |               width:'500px',
235 |               borderRadius: '10px',
236 |               backgroundColor:'rgba(0,150,255,0.5)',

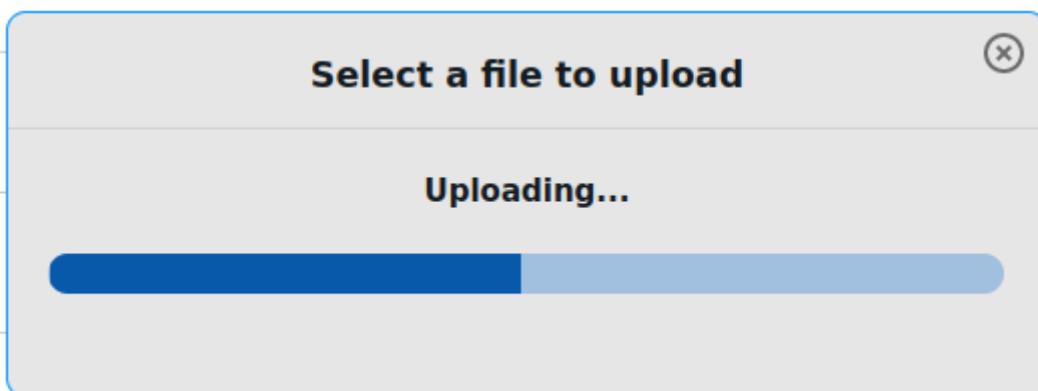
237 |             }} />
238 |           )
239 |         }
240 |       </ModalBody>
241 |       <IconButton onClick={() => setOpen(false)} style={{position:'relative',display:'flex',left:'477px',top:
242 |         uploading ? '-160px' : '-190px'}}>
243 |         <HighlightOffIcon/>
244 |       </IconButton>
245 |     </form>
246 |   </div>
247 | </Modal>
```

Este componente es el encargado de, cuando el usuario pulsa el botón para subir un nuevo fichero, mostrar el popup que permite al usuario seleccionar el fichero correspondiente. Aparte de esto, una vez se ha iniciado la subida, se encarga de mostrar una barra de carga indeterminada al usuario para informarle de que el sistema está trabajando en subir su fichero.

Este componente se ve de la siguiente forma:



Visualización por defecto del componente al abrirse.



Visualización del componente en proceso de subir un fichero.

Todo y que este componente parezca que debería estar más abajo en el código, está justo al inicio para que, una vez que se tenga que mostrar, se pinte por encima del resto de elementos de la pantalla.

Si continuamos analizando el código nos encontramos con el botón responsable de subir ficheros.

```

255
256
257
258
259
260
261
262
263
264
265
      <Button onClick={() => setOpen(true)} variant="outlined" style={{
        borderRadius: '999px',
        margin: '0px 0px 0px 15px'
      }>
        <img src={plus} width={30} height={30} alt='New' />
        <text style={{color:
          mode === "light"
            ? "black"
            : 'rgba(190, 190, 190, 1)',
          fontSize: "15px"}>&nbsp;&nbsp;&nbsp;New</text>
      </Button>
    
```

Este botón aparte de tener la imagen del “+” y el texto “New” es el encargado de establecer el estado del componente Modal explicado previamente. Este estado se define con un booleano, True = mostrar y False = no mostrar.

A continuación de este botón nos encontramos con los botones de visualización. Estos son los botones “My files”, “Shared”, “Trash” y “Storage” y están implementados de la siguiente forma:

```

267   <Button onClick={() => setSideSelection("MyFiles")}>
268     <InsertDriveFileIcon style={{color:
269       mode === "light"
270         ? 'rgba(90, 90, 90, 1)'
271         : 'rgba(190, 190, 190, 1)'}}/>
272     <div style={{display: 'flex', transform: 'translate(-95px, 0px)'}}>
273       <InsertDriveFileIcon style={{color:
274         mode === "light"
275           ? 'rgba(90, 90, 90, 1)'
276           : 'rgba(190, 190, 190, 1)'}}/>
277       <text style={{color:
278         mode === "light"
279           ? 'rgba(90, 90, 90, 1)'
280           : 'rgba(190, 190, 190, 1)'},
281         fontSize: "13px"}>&nbsp;&nbsp;&nbsp;My Files</text>
282     </div>
283   </Button>

```

Disponemos de un componente *Button*, el cual al pulsarlo establece el valor de una variable compartida entre componentes, al correspondiente del botón recién pulsado. Este es el mecanismo que se utiliza para decidir cual de todos los componentes variables mostrar en cada momento.

Dentro de este botón, tenemos la correspondiente imagen y el texto necesario para que el usuario entienda su funcionalidad.

Al tener un total de 4 botones, tenemos este mismo código repetido 4 veces, una por cada botón.

Adicionalmente y, aunque entraremos más en detalle en este punto más adelante, también disponemos de una barra de progreso que indica al usuario el espacio que le queda disponible.

Esto está implementado con el siguiente código:

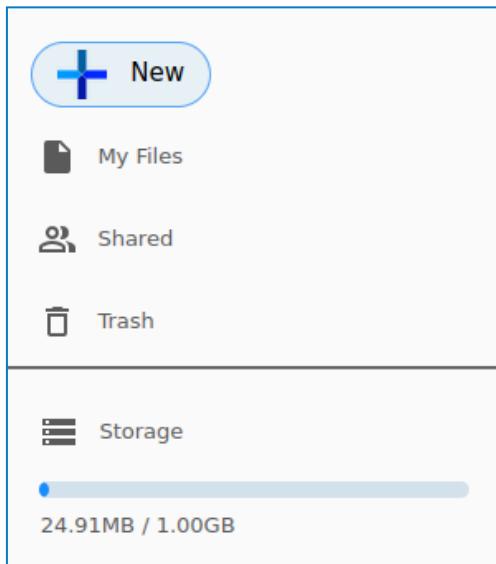
```

340   <SidebarOptions>
341     <BorderLinearProgress variant="determinate" value={(diskusage.current/diskmax)*100} style={{margin: '05px 20px'}}/>
342     <text style={{color:
343       mode === "light"
344         ? 'rgba(90, 90, 90, 1)'
345         : 'rgba(190, 190, 190, 1)'},
346       fontSize: "13px"}>&nbsp;&nbsp;&nbsp;${(reformatSize(diskusage.current)) / (reformatSize(diskmax))}</text>
347   </SidebarOptions>

```

Esta implementación es simplemente un componente *BorderLiner-Progress* ya implementado en una librería de componentes y establecer sus valores actual y máximo a los correspondientes de espacio usado por el usuario.

El resultado final de todo este componente es el siguiente:



7.2.8.3. Data.tsx

Ahora que ya hemos visto ambos componentes fijos, veamos en detalle los componentes variables.

Para poder mostrar al usuario los ficheros que dispone, hemos de conseguir esta información de alguna forma (más adelante veremos la forma en la que hemos implementado esto). Por el momento pensemos en que de alguna forma tenemos en una variable un array con todos los ficheros del usuario.

Antes de ver la implementación del código, hemos de saber que el estado por defecto de esta vista al iniciarse, es mostrar los ficheros en casilla en la parte superior y en una fila en la parte inferior.

Los ficheros representados de estas dos formas pueden ser ordenados por nombre, propietario, fecha y tamaño.

Adicionalmente, la vista de los ficheros en casilla puede ocultarse para solamente mostrar las filas de ficheros.

Sabiendo esto, para implementar la primera visualización de los ficheros, es decir, los ficheros en casilla, tenemos la siguiente funcionalidad:

```

250  useEffect(() => {
251    var filesLoopGrid = () => {
252      if (grid_state) {
253        var res: React.JSX.Element [] = [];
254        var i = 0;
255        if (input_text === "") {
256          for (i = 0; i < filesArray.length; i++) {
257            //@ts-ignore
258            if (filesArray[i].estado === "activo") {
259              const file = filesArray[i];
260              res.push(
261                //@ts-ignore
262                <DataFile key={file.id}>
263                  {"/@ts-ignore "}
264                  {selectIcon(file.nombreArchivo)}
265                  <Divider sx={{"borderColor: mode === 'light' ? 'rgba(210, 210, 210, 0.8)' : 'rgba(240,240,240,0.6)'}}/>
266                  {"/@ts-ignore "}
267                  <Button sx={{width:'100%', borderRadius:'0px'}} onMouseDown={handleMouseDown} onClick={() => handleMoreOptions(file)}>
268                    {"/@ts-ignore"}
269                    <p>{file.nombreArchivo.length <= 28 ? file.nombreArchivo : file.nombreArchivo.slice(0,22)+".." +file.nombreArchivo.slice(file.nombreArchivo.length-4,file.nombreArchivo.length)}</p>
270                  </Button>
271                </DataFile>
272              )
273            }
274          }
275        }
276        return res
277      } else {
278        for (i = 0; i < filesArray.length; i++) {
279          //@ts-ignore
280          if (filesArray[i].estado === "activo") {
281            //@ts-ignore
282            if (filesArray[i].nombreArchivo.includes(input_text)) {
283              const file = filesArray[i];
284              res.push(
285                //@ts-ignore
286                <DataFile key={file.id}>
287                  {"/@ts-ignore "}
288                  {selectIcon(file.nombreArchivo)}
289                  <Divider sx={{"borderColor: mode === 'light' ? 'rgba(210, 210, 210, 0.8)' : 'rgba(240,240,240,0.6)'}}/>
290                  {"/@ts-ignore "}
291                  <Button sx={{width:'100%', borderRadius:'0px'}} onMouseDown={handleMouseDown} onClick={() => handleMoreOptions(file)}>
292                    {"/@ts-ignore"}
293                    <p>{file.nombreArchivo.length <= 28 ? file.nombreArchivo : file.nombreArchivo.slice(0,22)+".." +file.nombreArchivo.slice(file.nombreArchivo.length-4,file.nombreArchivo.length)}</p>
294                  </Button>
295                </DataFile>
296              )
297            }
298          }
299        }
300      }
301    }
302    return res
303  }
304}
305 else return <div/>
306
307 //@ts-ignore
308 setFilesGrid(filesLoopGrid());
309 ,[filesArray.mode,grid_state,mOptions_state,handleMoreOptions,input_text,setFilesArray])

```

Lo primero que podemos ver es que la función principal, *filesLoopGrid*, está dentro de otra función llamada *useEffect(() => {}, [])*.

Esta función *useEffect* es propia de React.js y su principal funcionalidad es ejecutar código al cumplirse ciertas condiciones.

El código ejecutado se encuentra entre los {}, en nuestro caso la función *filesLoopGrid*. La forma en la que se decide cuando ejecutarse viene dada por las dependencias, estas son las que están dentro de los [].

Por lo tanto, la función de esta imagen ejecutará la función *filesLoopGrid* cada vez que detecte un cambio en alguna de las variables “filesArray”, “mode”, “mOptions_state” o “input_text”.

Ahora que sabemos en esencia el funcionamiento del código, vamos a analizarlo en profundidad.

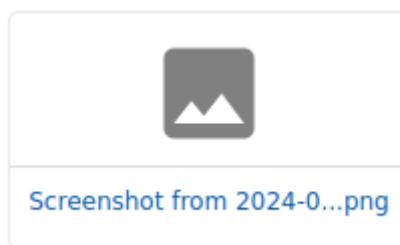
La tercera línea muestra un *if(grid_state)*, esta comprobación es la encargada de decidir si mostrar los elementos en casilla o no, es decir, decide si la totalidad de esta función se ejecutará, mostrando archivos, o si por el contrario no hará nada y no se mostrará nada.

Dentro de esta comprobación tenemos un `if(input_text== "")`. Esta variable es la que se actualiza con el input del componente [Header.tsx](#) mencionado anteriormente, por lo que si no tiene nada escrito, es decir, `input_text== ""`, muestra todos los ficheros. En caso contrario, es decir, el usuario ha introducido algo y por tanto está buscando unos archivos en concreto, solo se mostrarán los ficheros cuyos nombres contengan de forma parcial o total el texto introducido por el usuario. Esta comprobación de nombres está en la línea 283 de la imagen. Adicionalmente, hacemos una comprobación para ver si los archivos tienen un estado “activo” o “archivado”. Esta parte se explicará más en detalle en la parte del componente [DataTrash.tsx](#)

Independientemente del valor de la variable de input, el tratamiento de los archivos es el mismo, consiste en crear un componente de tipo *DataFile*, el cual vuelve a ser un componente propio y tiene la siguiente implementación:

```
55 const DataFile = styled.div`  
56   text-align: center;  
57   border: 1px solid rgb(204 204 204 / 46%);  
58   margin: 10px;  
59   min-width: 200px;  
60   padding: 10px 0px 0px 0px;  
61   border-radius: 5px;  
62   svg {  
63     font-size: 60px;  
64     color:gray  
65   }  
66   p {  
67     margin-top: 5px;  
68     font-size: 12px;  
69     padding: 10px 0px;  
70   }  
71`
```

Dentro de este componente se añade un icono, que variará dependiendo de la extensión del fichero, un botón que será el encargado de abrir el menú de opciones de cada fichero y el nombre del fichero como texto. Este texto está tratado de forma que, en caso de ser muy largo, y que por tanto se saldría del recuadro delimitador del fichero, se corta a los 22 primeros caracteres y se le añade los correspondientes a la extensión de fichero. Esto es lo que hace que el nombre de algunos ficheros se vea de la siguiente forma:



Ahora que tenemos una casilla creada, para poder devolver múltiples de forma correcta lo que hacemos es un push de este elemento dentro de otro general que contiene todas las casillas.

Este elemento general es el que se devuelve al finalizar la ejecución de la función y, por tanto, lo que se acaba mostrando en la pantalla.

Para implementar la segunda visualización, es decir, las filas de ficheros tenemos la siguiente funcionalidad:

```

310 useEffect(() => {
311   var filesLoopRow = () => {
312     var res: React.SA.Element[] = []
313     var i = 0
314     if (input_text === "") {
315       for (i = 0; i < filesArray.length; i++) {
316         //@ts-ignore
317         if (filesArray[i].estado === "activo") {
318           const file = filesArray[i]
319           res.push(
320             <DataListRow style={{height: '70px', width: '100%'}}>
321               <Box width='25%'>
322                 /*@ts-ignore*/
323                 <Button onMouseDown={handleMouseDown} onClick={() => handleMoreOptions(file)}>
324                   /*@ts-ignore*/
325                   <p>(selectIcon(file.nombreArchivo){file.nombreArchivo.length <= 65 ? file.nombreArchivo : file.nombreArchivo.slice(0,61)+"..."}+file.nombreArchivo.slice(file.nombreArchivo.length-4,file.nombreArchivo.length)}</p>
326                 </Button>
327               </Box>
328               <Box width='25%' style={{transform: 'translate(200px, 0px)'}}>
329                 /*@ts-ignore*/
330                 <p style={{position: 'relative',alignItems:'center'}}>(file.correo)</p>
331               </Box>
332               <Box width='25%' style={{transform: 'translate(250px, 0px)'}}>
333                 /*@ts-ignore*/
334                 <p style={{position: 'relative',alignItems:'center'}}>(file.fecha)</p>
335               </Box>
336               <Box width='25%' style={{transform: 'translate(250px, 0px)'}}>
337                 /*@ts-ignore*/
338                 <p style={{position: 'relative',alignContent:'center'}}>(reformatSize(file.tamano))</p>
339               </Box>
340             </DataListRow>
341           )
342         }
343       }
344     }
345   }
346   return res
347 }
348 } else {
349   for (i = 0; i < filesArray.length; i++) {
350     //@ts-ignore
351     if (filesArray[i].estado === "activo") {
352       //@ts-ignore
353       if (filesArray[i].nombreArchivo.includes(input_text)) {
354         const file = filesArray[i]
355         res.push(
356           <DataListRow style={{height: '70px', width: '100%'}}>
357             <Box width='25%'>
358               /*@ts-ignore*/
359               <Button onMouseDown={handleMouseDown} onClick={() => handleMoreOptions(file)}>
360                 /*@ts-ignore*/
361                 <p>(selectIcon(file.nombreArchivo){file.nombreArchivo.length <= 65 ? file.nombreArchivo : file.nombreArchivo.slice(0,61)+"..."}+file.nombreArchivo.slice(file.nombreArchivo.length-4,file.nombreArchivo.length)}</p>
362               </Button>
363             </Box>
364             <Box width='25%' style={{transform: 'translate(200px, 0px)'}}>
365               /*@ts-ignore*/
366               <p style={{position: 'relative',alignItems:'center'}}>(file.correo)</p>
367             </Box>
368             <Box width='25%' style={{transform: 'translate(250px, 0px)'}}>
369               /*@ts-ignore*/
370               <p style={{position: 'relative',alignItems:'center'}}>(file.fecha)</p>
371             </Box>
372             <Box width='25%' style={{transform: 'translate(250px, 0px)'}}>
373               /*@ts-ignore*/
374               <p style={{position: 'relative',alignContent:'center'}}>(reformatSize(file.tamano))</p>
375             </Box>
376           </DataListRow>
377         )
378       }
379     }
380   }
381   return res
382 }
383 //@ts-ignore
384 setFilesRow(filesLoopRow(""));
385 ,[filesArray.mode,handleMoreOptions,input_text])

```

Al igual que en la implementación previa, la función (en este caso *filesLoopRow*), está dentro de un *useEffect* con dependencias “filesArray”, “mode” e “input_text”.

Dado que en esta visualización no disponemos de la opción de ocultarla, la primera comprobación que nos encontramos es directamente la del `input_text` y a continuación la comprobación de estado del fichero.

En esencia esta función y la anterior se comportan de la misma forma, la diferencia es la forma de pintar el archivo. En este caso tenemos de nuevo una variable general a la que hacer push de los elementos individuales. Estos elementos individuales se forman con un elemento `DataListRow` que, de nuevo, es un componente propio que está implementado de la siguiente manera:

```
72 const DataListRow = styled.div`  
73   display: flex;  
74   align-items: center;  
75   justify-content: space-between;  
76   border-bottom: 1px solid #ccc;  
77   padding: 10px;  
78   p {  
79     display: flex;  
80     align-items: center;  
81     font-size: 13px;  
82     b {  
83       display: flex;  
84       align-items: center;  
85     }  
86     svg {  
87       font-size: 22px;  
88       margin: 10px;  
89     }  
90   }  
91`
```

Dentro de este componente tenemos diversos componentes *Box*. Esto es para ocupar completamente el espacio disponible y que todo lo que pongamos dentro de este componente *Box* tenga unos límites. De esta forma es como conseguimos que todos los nombres de ficheros, fechas y más información que damos salga completamente alineada verticalmente.

Dentro del primer componente *Box* tenemos un botón con el icono correspondiente en función de la extensión del fichero y el nombre del fichero. Este nombre tiene el mismo tratamiento que en las casillas para tener controlado el caso de un nombre excesivamente largo.

En el segundo componente *Box* tenemos el correo del propietario del fichero.

En el tercero tenemos la fecha de la última modificación. Esto es la fecha en la que se le cambió el nombre al fichero a través de nuestro sistema o la fecha en la que se subió el fichero.

En el último componente tenemos el tamaño del fichero. Este tamaño como viene dado en bytes por parte del backend, lo pasamos por una función para que lo formatee y devuelva un valor más legible para el usuario. Esta función es la siguiente:

```

421     function reformatSize(size: number) {
422       if ((size / 1000000000) >= 1) return (size / 1000000000).toFixed(2) + "GB";
423       else if ((size / 1000000) >= 1) return (size / 1000000).toFixed(2) + "MB";
424       else if ((size / 1000) >= 1) return (size / 1000).toFixed(2) + "KB";
425       else return size + "B";
426     }

```

Como se puede ver, la función devuelve el tamaño del fichero en GB, MB, KB o B y con 2 decimales de precisión.

Si nos adentramos en la parte del código que no utiliza la actualización dinámica del use effect, tenemos las siguientes implementaciones:

```

1211   <DataHeader>
1212     <div className="headerLeft">
1213       <text style={{color:
1214         mode === "light"
1215           ? "black"
1216           : 'rgba(190, 190, 190, 1)',
1217             fontSize: "15px"}}>My Storage</text>
1218     <Button onClick={handleStorageGrid} >
1219       {swapIconOrder("grid")}
1220     </Button>
1221   </div>
1222 </DataHeader>

```

Esto corresponde a la parte superior de este componente. Aunque el componente utilizado se llama *DataHeader*, no tiene nada que ver con el componente [Header.tsx](#) explicado anteriormente.

Este es un componente propio implementado de la siguiente forma:

```

33   const DataHeader = styled.div` 
34     display: flex;
35     align-items: center;
36     justify-content: space-between;
37     border-bottom: 1px solid lightgray;
38     height: 40px;
39     .headerLeft {
40       display: flex;
41       align-items: center;
42     }
43     .headerRight svg {
44       margin: 0px 10px;
45     }
46

```

Este componente simplemente muestra un texto de “My Storage” y tiene un botón que cambia el estado de la variable “grid_state” para mostrar la visualización en casillas o no.

Este componente en ejecución se ve de la siguiente manera:



Otra sección del código con una funcionalidad muy similar es el header de los ficheros pintados en filas.

Esto tiene la siguiente implementación:

```
1347 | <DataListRow>
1348 |   <Box width={'25%'>
1349 |     <p><b>Name
1350 |     <Button onClick={() => handleRowOrder("name")}>
1351 |       {swapIconOrder("name")}
1352 |     </Button> </b></p>
1353 |   </Box>
1354 |   <Box width={'25%'> style={{transform: 'translate(200px, 0px)'>
1355 |     <p><b>Owner
1356 |     <Button onClick={() => handleRowOrder("owner")}>
1357 |       {swapIconOrder("owner")}
1358 |     </Button></b></p>
1359 |   </Box>
1360 |   <Box width={'25%'> style={{transform: 'translate(250px, 0px)'>
1361 |     <p><b>Last Modified
1362 |     <Button onClick={() => handleRowOrder("mod")}>
1363 |       {swapIconOrder("mod")}
1364 |     </Button></b></p>
1365 |   </Box>
1366 |   <Box width={'25%'> style={{transform: 'translate(250px, 0px)'>
1367 |     <p><b>File Size
1368 |     <Button onClick={() => handleRowOrder("size")}>
1369 |       {swapIconOrder("size")}
1370 |     </Button></b></p>
1371 |   </Box>
1372 | </DataListRow>
```

Como se puede observar, es el mismo elemento de fila que utilizábamos para los ficheros, solo que en este caso simplemente son textos y los botones correspondientes para ordenar los ficheros.

La función a la que llaman estos botones es la siguiente:

```
702     function handleRowOrder (section: string) {
703         switch (section) {
704             case "name":
705                 setNameState(!name_state);
706                 name_state ? filesSorter("name") : filesSorter(["rname"]);
707                 setFilesRow(filesLoopRow());
708                 setFilesGrid(filesLoopGrid());
709                 break;
710             case "owner":
711                 setOwnerState(!owner_state);
712                 owner_state ? filesSorter("owner") : filesSorter("rowner");
713                 setFilesRow(filesLoopRow());
714                 setFilesGrid(filesLoopGrid());
715                 break;
716             case "mod":
717                 setModState(!mod_state);
718                 mod_state ? filesSorter("mod") : filesSorter("rmod");
719                 setFilesRow(filesLoopRow());
720                 setFilesGrid(filesLoopGrid());
721                 break;
722             case "size":
723                 setSizeState(!size_state);
724                 size_state ? filesSorter("size") : filesSorter("rsize");
725                 setFilesRow(filesLoopRow());
726                 setFilesGrid(filesLoopGrid());
727                 break;
728         }
729     }
```

Esta función cambia el estado de la variable correspondiente, llama a la función *filesSorter* para ordenar los ficheros y fuerza una actualización.

Esta función para ordenar está implementada de la siguiente forma:

```

574     function compareFn_size (a: { tamano: string; }, b: { tamano: string; }) {
575         if (parseInt(a.tamano) < parseInt(b.tamano)) return -1;
576         else if (parseInt(a.tamano) > parseInt(b.tamano)) return 1;
577         else return 0;
578     }
579     function compareFn_rsize (a: { tamano: string; }, b: { tamano: string; }) {
580         if (parseInt(a.tamano) > parseInt(b.tamano)) return -1;
581         else if (parseInt(a.tamano) < parseInt(b.tamano)) return 1;
582         else return 0;
583     }
584
585     function filesSorter (order: string) {
586         switch (order) {
587             case "name":
588                 setFilesArray(filesArray.sort(compareFn_name));
589                 lastOrder.current = "name"
590                 break;
591             case "rname":
592                 setFilesArray(filesArray.sort(compareFn_rname));
593                 lastOrder.current = "rname"
594                 break;
595             case "owner":
596                 setFilesArray(filesArray.sort(compareFn_owner));
597                 lastOrder.current = "owner"
598                 break;
599             case "rowner":
600                 setFilesArray(filesArray.sort(compareFn_rowner));
601                 lastOrder.current = "rowner"
602                 break;

```

Como se puede ver, la función *filesSorter* está implementada con un switch case para seleccionar el método de ordenación adecuado. Una vez seleccionado el caso, se llama a la función auxiliar correspondiente para ordenar el array de ficheros.

En la parte superior de la imagen podemos observar la función auxiliar para ordenar por tamaño, tanto de forma ascendente como descendente.

Este header de ficheros, una vez se está ejecutando, se ve de la siguiente forma:

Name	Owner	Last Modified	File Size
------	-------	---------------	-----------

A continuación tenemos la gestión de la visualización del menú de opciones.

Dado que la creación de un menú por cada archivo era demasiado costosa en cuanto a espacio y procesamiento en cargar todos los ficheros, optamos por tener un solo menú de opciones que mostramos y colocamos en unas ciertas coordenadas de la pantalla al seleccionar un fichero concreto.

Si recordamos las funciones de *filesLoopGrid* y *filesLoopRow*, el botón que creabamos llamaba a la función *handleMoreOptions*.

```

207     var handleMoreOptions = useCallback((file) => {
208       setMOptionsState(!mOptions_state);
209       optionId.current = file._id
210       optionName.current = file.nombreArchivo
211       optionFile.current = file
212     }, [mOptions_state])

```

Como se puede ver, guardamos el identificador y el nombre para un acceso rápido y el fichero entero por si necesitamos información adicional no tan crítica para el tiempo de respuesta.

Esta función tiene una sintaxis similar al *useEffect*, dispone de un *useCallback* y también tiene un listado de dependencias. Esto es porque como es una función que se llama dentro de un *useEffect*, es necesario usarla de esta forma o de lo contrario no se ejecutará correctamente.

Continuando con el menú de opciones. Este está implementado de la siguiente forma:

```

1223   <div>
1224     {mOptions_state === true &&
1225      <Box ref={wrapperRef} sx={{<
1226        position: 'absolute',
1227        left: optionX.current,
1228        top: optionY.current,
1229        border: '1px solid',
1230        borderRadius: '10px',
1231        borderColor:
1232          mode === 'light' ? 'rgba(155,155,155,1)' : 'rgba(200,200,200,1),
1233          width: '200px',
1234          background:
1235            mode === 'light' ? 'rgba(255,255,255,1)' : 'rgba(20,20,20,1),
1236            zIndex: 1,
1237        }}>
1238       <Box sx={{<
1239         maxWidth: '100%',
1240       }}>
1241         <b><p style={{color:
1242           mode === "light"
1243             ? 'rgba(90, 90, 90, 1)'
1244             : 'rgba(190, 190, 190, 1)',
1245           textAlign: 'center',
1246           wordWrap: 'break-word',
1247           fontSize: "13px"}>{optionName.current}</p></b>
1248       </Box>
1249       <Divider sx={{borderColor: mode === 'light' ? 'rgba(210, 210, 210, 0.8)' : 'rgba(240,240,240,0.6)'}}/>
1250       <Button sx={{<
1251         width: '100%',
1252         borderRadius: '0px'
1253       }} onClick={handleDownload}>
1254         <DownloadIcon style={{margin:'0px 10px 0px 0px',
1255           color:
1256             mode === "light"
1257               ? 'rgba(90, 90, 90, 1)'
1258               : 'rgba(190, 190, 190, 1)
1259             }}/><p style={{color:
1260               mode === "light"
1261                 ? 'rgba(90, 90, 90, 1)'
1262                 : 'rgba(190, 190, 190, 1),
1263                   fontSize: "13px"}>Download</p>
1264       </Button>
1265       <Divider sx={{borderColor: mode === 'light' ? 'rgba(210, 210, 210, 0.8)' : 'rgba(240,240,240,0.6)'}}/>

```

En caso de que la variable del menú sea cierta, es decir, hemos de pintar el menú, establecemos la posición del componente como absoluta, es decir, podemos colocarlo en cualquier posición de la pantalla que queramos sin que nos molesten otros elementos. A continuación obtenemos la posición del cursor tanto en x como en y a través de la función *handleMouseDown*. A esta función también se llama al pulsar el botón de los ficheros.

```

200  const handleMouseDown = (e: React.MouseEvent<HTMLElement>) => {
201    if ((e.clientX + 200 + 10) > window.innerWidth) optionX.current = e.clientX - (e.clientX + 200 + 10 - window.innerWidth);
202    else optionX.current = e.clientX + 10;
203    if ((e.clientY + 300 + 10) > window.innerHeight) optionY.current = e.clientY - (e.clientY + 300 + 10 - window.innerHeight);
204    else optionY.current = e.clientY;
205  }

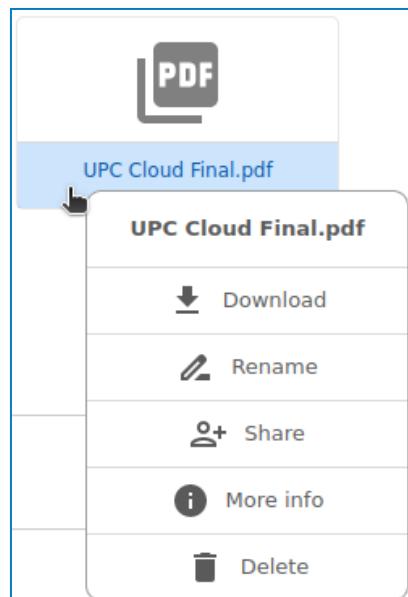
```

Esta función simplemente obtiene las coordenadas del mouse del usuario y le suma un cierto offset para que el menú de opciones no se genere justo encima y resulte más cómodo. Aparte de esto también se hace una comprobación para ver que el menú se generará siempre dentro de los límites de la pantalla.

Otro aspecto importante sobre los valores de los parámetros de este menú es el atributo “zIndex”, el cual nos permite pintar por encima de todo el resto de componentes como si tuviéramos diferentes capas de pintado disponibles.

Una vez dentro de este componente, se muestra el nombre del fichero y a continuación están los diferentes botones correspondientes a las diferentes acciones que puede realizar el usuario. En la imagen solo se muestra el botón de “Download” ya que la implementación del resto es exactamente la misma solo que llamando a las funciones que le corresponde a cada botón.

Este menú se ve de la siguiente forma:



Dado que el menú es el mismo para todos los archivos solo que se mueve de posición, cada una de las funciones de descargar, cambiar nombre, compartir, más información y eliminar, acceden a esta variable global mencionada para obtener la id o nombre del fichero para saber sobre cual operar.

El botón de descarga llama a la siguiente función:

```
870 function handleDownload() {
871     const recieveData = async () => {
872         const userData = {
873             _id: optionFile.current._id,
874             correo: localStorage.getItem("email"),
875             token: localStorage.getItem("token")
876         }
877         try {
878             const response = await fetch(` ${backendUrl}/archivo/obtenerarchivo`, {
879                 method: "POST",
880                 headers: {
881                     "Content-Type": "application/json",
882                 },
883                 body: JSON.stringify(userData),
884             });
885             if (response.ok) {
886                 const data = await response.json()
887                 //console.log(data.message)
888                 window.open(data.message)
889                 setMOptionsState(false)
890             } else {
891                 alert("An error occurred while trying to download your file. Please try again later.");
892             }
893         } catch (error) {
894             console.error("Error at data rename:", error);
895             alert("An unexpected error occurred while trying to download your file. Please try again later.");
896         }
897     };
898     recieveData();
899 }
```

Esta función, al igual que algunas que hemos visto anteriormente, hace una petición http al backend para descargar el fichero. En el mensaje en formato JSON se añaden el id del fichero, y el correo y token del usuario.

Esta petición, en caso de ser correcta, devuelve un string que corresponde a la url para hacer una petición http al nodo de IPFS para descargar el archivo correspondiente. Por lo tanto, una vez recibimos la respuesta solamente hemos de abrir una nueva ventana del navegador con la url devuelta por el backend para comenzar la descarga del fichero.

Esta url tiene el siguiente aspecto:

```
"http://10.4.41.54:8080/ipfs/" + fichero.ipfs_hash?"  
download=true&filename=" + fichero.nombreArchivo
```

El botón de eliminar funciona de forma similar, crea una petición http a una url del backend para que cambie el estado del fichero de “activo” a “archivado”.

En cuanto al botón de cambiar nombre y compartir, tienen un funcionamiento similar, solo que antes de realizar esta llamada al backend primero muestra un popup con un input para poder introducir el nuevo nombre del fichero y el correo del usuario a compartir correspondientemente.

La implementación del popup para cambiar el nombre es la siguiente:

```

1089 <Modal open={rename} onClose={() => setRename(false)}
1090   style={{
1091     position: 'absolute',
1092     top: '25vh',
1093     left: '40vw',
1094     width: '520px',
1095     backgroundColor: mode === 'light'
1096       ? 'rgba(230, 230, 230, 1)'
1097       : 'rgba(30, 30, 30, 1)',
1098     border: '1px solid',
1099     borderRadius: '10px',
1100     borderColor: 'rgba(0,150,255,1)',
1101     zIndex: 2,
1102   }}>
1103   <div>
1104     <IconButton onClick={() => setRename(false)} style={{position:'relative',display:'flex',left:'477px'}}>
1105       <HighlightOffIcon/>
1106     </IconButton>
1107     <div style={{display:'flex'}}>
1108       <Box sx={{
1109         margin: '0px 0px 0px 0px',
1110         width:'200px'
1111       }}>
1112         <DataFile style={{
1113           minWidth: '0px',
1114           height: '80%',
1115           width: '100%'
1116         }}>{selectIcon(optionFile.current.nombreArchivo)}</DataFile>
1117       </Box>
1118       <p style={{
1119         maxWidth:'270px',
1120         wordWrap: 'break-word',
1121         margin: '0px 0px 0px 25px',
1122         alignSelf:'center'
1123       }}><b>{optionFile.current.nombreArchivo}</b></p>
1124     </div>
1125     <Divider sx={{borderColor: mode === 'light' ? 'rgba(210, 210, 210, 0.8)' : 'rgba(240,240,240,0.6')}}/>
1126     <div>
1127       <Box sx={{display:'flex'}}>
1128         <InputRename sx={{
1129           marginTop:'10px'
1130         }}>
1131           <StyledInputBase
1132             placeholder="Input_New_Name.txt"
1133             onChange={handleInputRename}
1134           />
1135         </InputRename>
1136       </Box>
1137       <Box sx={{display:'flex'}}>
1138         <Button sx={{
1139           width:'100%',
1140           margin:'10px'
1141         }} variant='contained' onClick={() => handleRename(optionNewName.current)}>
1142           <p>Rename</p>
1143         </Button>
1144       </Box>
1145     </div>
1146   </div>
1147 </Modal>

```

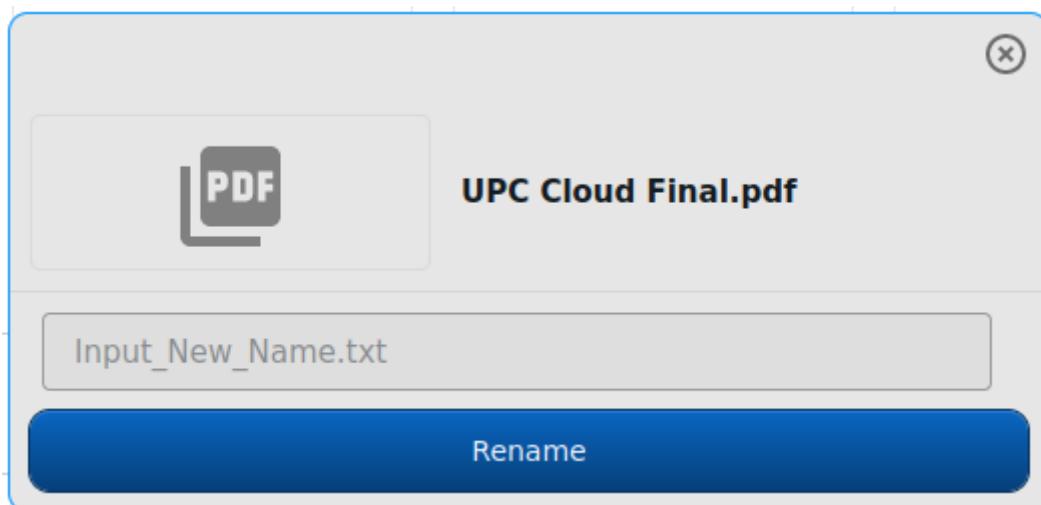
Podemos observar atributos importantes como una posición absoluta o un zIndex de 2. Dentro del propio componente Modal creado tenemos el botón para desactivar el componente, el icono del fichero que, de

nuevo, se selecciona en función de la extensión del fichero, y el nombre del fichero.

Más abajo en el código podemos ver la parte responsable de gestionar el input del usuario. Esto funciona de forma similar a la barra de búsqueda del componente [Header.tsx](#), almacena el nombre introducido en una variable.

Una vez que se pulsa el botón para confirmar el cambio de nombre, se llama a la función que genera la petición http al backend, pasándole el nuevo nombre introducido como parámetro.

Este popup tiene el siguiente aspecto:



En referencia al botón de más información, funciona de forma muy similar a los botones de cambiar nombre y compartir. El botón cambia el estado del componente Modal correspondiente y este pasa a mostrarse en pantalla como un popup.

Este componente se implementa de la siguiente forma:

```

925      <Modal open={open} onClose={() => setOpen(false)}>
926        style={
927          position: 'absolute',
928          top: '25vh',
929          left: '40vw',
930          width: '520px',
931          backgroundColor: mode === 'light'
932            ? 'rgba(230, 230, 230, 1)'
933            : 'rgba(30, 30, 30, 1)',
934          border: '1px solid',
935          borderRadius: '10px',
936          borderColor: 'rgba(0,150,255,1)',
937          zIndex: 2,
938        }>
939        <div>
940          <IconButton onClick={() => setOpen(false)} style={{position:'relative',display:'flex',left:'477px'}}>
941            <HighlightOffIcon/>
942          </IconButton>
943          <div style={{display:'flex'}}>
944            <Box sx={{
945              margin: '0px 0px 0px 0px',
946              width: '200px'
947            }}>
948              <DataFile style={{
949                minWidth: '0px',
950                height: '80%',
951                width: '100%'
952              }}>{selectIcon(optionFile.current.nombreArchivo)}</DataFile>
953            </Box>
954            <p style={{
955              maxWidth:'270px',
956              wordWrap: 'break-word',
957              margin: '0px 0px 0px 25px',
958              alignSelf:'center'
959            }}><b>{optionFile.current.nombreArchivo}</b></p>
960          </div>
961          <Divider sx={{borderColor: mode === 'light' ? 'rgba(210, 210, 210, 0.8)' : 'rgba(240,240,240,0.6')}}/>
962          <div>
963            <Box sx={{display:'flex'}}>
964              <Box sx={{
965                width:'30%',
966                margin: '0px 0px 0px 20px',
967              }}>
968                <p>id:</p>
969              </Box>
970              <Box sx={{
971                width: '62%',
972                margin: '0px 0px 0px 0px',
973              }}>
974                <p style={{
975                  maxWidth: '100%',
976                  wordWrap: 'break-word',
977                }}>{optionFile.current._id}</p>
978              </Box>
979            </Box>

```

Tenemos el mismo componente mencionado previamente con las mismas características definidas y también volvemos a tener el icono del fichero y su nombre.

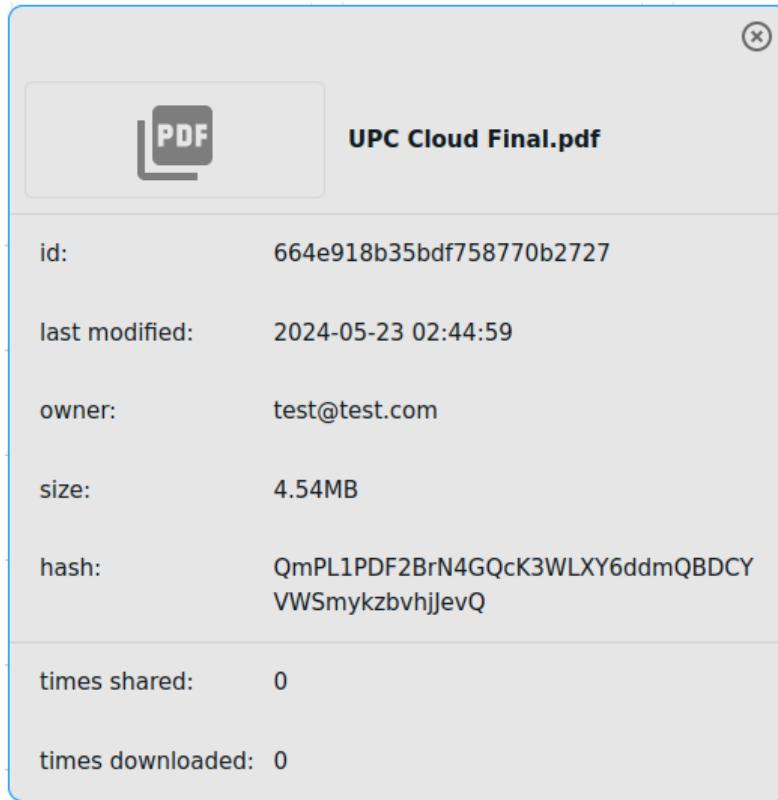
A partir de este punto es donde la implementación es diferente (línea 961 en la imagen).

Ahora tenemos dos componentes *Box* para ocupar el máximo del espacio disponible y simular dos columnas. En la primera se coloca un texto indicando el atributo que se va a mostrar y en la segunda se coloca el valor del atributo en cuestión.

De esta forma podemos tener las columnas de este popup completamente alineadas independientemente de la cantidad de texto que pongamos.



Este popup se ve de la siguiente manera:



El componente [Data.tsx](#) en conjunto a todo lo explicado se ve de la siguiente forma:

The storage interface displays the following files:

Name	Owner	Last Modified	File Size
archivo.controller.js	test@test.com	2024-05-15 19:56:43	7.85KB
Enunciado.pdf	test@test.com	2024-05-23 02:51:36	59.12KB
front.tar.gz	test@test.com	2024-05-15 20:04:24	16.80MB
IMG_20211102_185507.jpg	test@test.com	2024-05-28 18:15:16	552.66KB
Screenshot from 2024-06-01 22:04:09.png	test@test.com	2024-06-01 22:04:58	39.54KB
SupoTemptReal.pdf	test@test.com	2024-06-01 21:15:05	2.95MB
UPC Cloud Final.pdf	test@test.com	2024-05-23 02:44:59	4.54MB

7.2.8.4. DataShared.tsx

El segundo de los componentes variables es el responsable de mostrar los elementos compartidos.

Este funciona de forma muy similar al componente [Data.tsx](#), es decir, tiene ambos headers mencionados, la función de filesLoopRow para mostrar los ficheros y la interacción del usuario con los diferentes botones de menús, opciones y descargas funciona de la misma forma.

Este componente lo único en lo que se diferencia a rasgos generales es el array de datos con el que trabaja desde un inicio. En este caso, para poder obtener los ficheros compartidos hemos de crear una petición al backend.

```
203 const fetchData = async () => {
204     if (localStorage.getItem("email") === null) {
205         alert("An unexpected error occurred while trying to load your files. Please try again later.");
206         // @ts-ignore
207         window.location = '/sign-in';
208     }
209     else {
210         const userData = {
211             correo: localStorage.getItem("email"),
212             token: localStorage.getItem("token")
213         }
214         try {
215             const response = await fetch(`${backendUrl}/compartido/obtenerArchivosCompartidos`, {
216                 method: "POST",
217                 headers: {
218                     "Content-Type": "application/json",
219                 },
220                 body: JSON.stringify(userData),
221             });
222             if (response.ok) {
223                 const data = await response.json();
224                 setSharedFilesArray(data.archivosCompartidos);
225
226             } else if (response.status === 404) {
227                 console.log("No shared files yet")
228                 setSharedFilesArray([])
229             } else {
230                 alert("An unexpected error occurred while trying to load your files. Please try again later.");
231             }
232             } catch (error) {
233                 console.error("Error at storage:", error);
234                 alert("An unexpected error occurred while trying to load your files. Please try again later.");
235             }
236         }
237     };
238     useEffect(() => {
239         fetchData();
240     },[])
241 }
```

Esta petición http se realiza a la url de backend preparada para devolver los ficheros compartidos a cierto usuario.

El contenido de la petición que se envía en este caso es solamente el correo del usuario y el token.

Si la respuesta por parte del backend resulta exitosa, guarda el array de ficheros devuelto en una variable global a este código para poder operar de igual manera que explicamos en el componente [Data.tsx](#).

Como podemos ver, esta función se llama dentro de un *useEffect*, por lo que se ejecutará de forma dinámica para obtener los ficheros de forma actualizada.

Una funcionalidad que no se ha explicado en el componente anterior es el contador de particiones y descargas. Todo y que el sistema blockchain no acabó de implementarse, el frontend realiza peticiones al backend de forma dinámica para obtener el valor de estos contadores para que, cuando el blockchain sea viable y se acabe de implementar obtener estos valores de la propia blockchain.

Dentro del popup de más información, tenemos las siguientes líneas de código:

```
829 <Divider sx={{"borderColor: mode === 'light' ? 'rgba(210, 210, 210, 0.8)' : 'rgba(240,240,240,0.6)'}}/>
830 <div>
831   <Box sx={{"display:'flex'}}>
832     <Box sx={{
833       width:'30%',
834       margin: '0px 0px 0px 20px',
835     }}>
836       <p>times shared:</p>
837     </Box>
838     <Box sx={{
839       width:'62%',
840       margin: '0px 0px 0px 0px',
841     }}>
842       <p style={{{
843         maxWidth: '100%',
844         wordWrap: 'break-word',
845       }}}>{timesShared.current}</p>
846     </Box>
847   </Box>
848   <Box sx={{"display:'flex'}}>
849     <Box sx={{
850       width:'30%',
851       margin: '0px 0px 0px 20px',
852     }}>
853       <p>times downloaded:</p>
854     </Box>
855     <Box sx={{
856       width:'62%',
857       margin: '0px 0px 0px 0px',
858     }}>
859       <p style={{{
860         maxWidth: '100%',
861         wordWrap: 'break-word',
862       }}}>{timesDownloaded.current}</p>
863     </Box>
864   </Box>
865 </div>
```

Podemos ver las variables “timesShared” y “timesDownloaded”. Estas están declaradas como globales y se actualizan dentro de la siguiente función:

```
315  useEffect(() => {
316    var blockchainData = async () => {
317      const userData = {
318        _id: optionFile.current._id,
319        correo: localStorage.getItem("email"),
320        token: localStorage.getItem("token")
321      }
322      try {
323        const response = await fetch(`${backendUrl}/archivo/blockchain`, {
324          method: "POST",
325          headers: {
326            "Content-Type": "application/json",
327          },
328          body: JSON.stringify(userData),
329        });
330        if (response.ok) {
331          const data = await response.json()
332          timesShared.current = data.shared;
333          timesDownloaded.current = data.viewed;
334        } else {
335          alert("An error occurred while trying to get info of your file. Please try again later.");
336        }
337      } catch (error) {
338        console.error("Error at data rename:", error);
339        alert("An unexpected error occurred while trying to get info of your file. Please try again later.");
340      }
341    };
342    blockchainData();
343  }, [optionFile.current._id])
```

Esta función al igual que las ya vistas anteriormente, hace una petición http al backend a la url correspondiente para obtener los contadores mencionados y actualiza los valores de las variables en caso de una respuesta correcta.

Esta implementación se ve de la siguiente forma:

times shared: 0

times downloaded: 0

El componente en general una vez ejecutado se ve de la siguiente manera:



My Shared Files			
Name	Owner	Last Modified	File Size
front.tar.gz	test@test.com	2024-05-22 20:55:34	16.80MB
2.6_research_innovation.pdf	a@a.com	2024-05-23 11:43:36	2.42MB

7.2.8.5. DataTrash.tsx

Este es el tercer de los componentes variables. Este componente funciona de forma muy similar al componente [Data.tsx](#).

Tal y como se puede intuir, la única diferencia a la hora de tratar los archivos es solamente mostrar los que tienen el estado “archivado”.

Por lo que la comprobación del código de *filesLoopRow* en este caso es la contraria.

En esta vista el menú de opciones pierde los botones de descargar, compartir y cambiar el nombre. A cambio obtiene un botón de eliminar permanentemente y uno para restaurar el fichero, es decir, para sacarlo de la papelera y devolverlo a su vista original.

Este botón de restaurar llama a una función que hace una petición al backend para cambiar el estado del fichero de “archivado” a “activo”.

El botón de eliminar permanentemente cambia el estado del Modal de eliminar permanentemente para que esté activo.

Este Modal tiene el siguiente código:

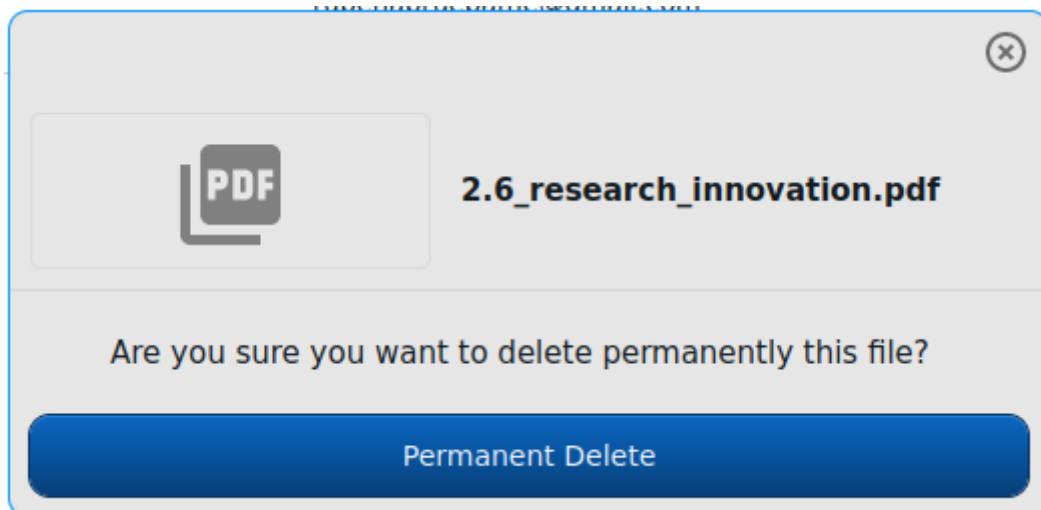
```

814      <Modal open={permDelete} onClose={() => setPermDelete(false)}>
815        style={{
816          position: 'absolute',
817          top: '25vh',
818          left: '40vw',
819          width: '520px',
820          backgroundColor: mode === 'light'
821            ? 'rgba(230, 230, 230, 1)'
822            : 'rgba(30, 30, 30, 1)',
823          border: '1px solid',
824          borderRadius: '10px',
825          borderColor: 'rgba(0,150,255,1)',
826          zIndex: 2,
827        }}>
828        <div>
829          <IconButton onClick={() => setPermDelete(false)} style={{position:'relative',display:'flex',left:'477px'}}>
830            <HighlightOffIcon/>
831          </IconButton>
832          <div style={{display:'flex'}}>
833            <Box sx={{
834              margin: '0px 0px 0px 0px',
835              width:'200px'
836            }}>
837              <DataFile style={{
838                minWidth: '0px',
839                height: '80%',
840                width: '100%'
841              }}>{selectIcon(optionFile.current.nombreArchivo)}</DataFile>
842            </Box>
843            <p style={{
844              maxWidth: '270px',
845              wordWrap: 'break-word',
846              margin: '0px 0px 0px 25px',
847              alignSelf:'center'
848            }}><b>{optionFile.current.nombreArchivo}</b></p>
849          </div>
850          <Divider sx={{borderColor: mode === 'light' ? 'rgba(210, 210, 210, 0.8)' : 'rgba(240,240,240,0.6)}}>
851          <div>
852            <Box sx={{display:'flex'}}>
853              <p style={{margin:'20px 0px 10px 50px'}}>Are you sure you want to delete permanently this file?</p>
854            </Box>
855            <Box sx={{display:'flex'}}>
856              <Button sx={{
857                width:'100%',
858                margin:'10px'
859              }} variant='contained' onClick={() => handlePermanentDelete()}>
860                <p>Permanent Delete</p>
861              </Button>
862            </Box>
863          </div>
864        </div>

```

Podemos ver una configuración muy similar a la del resto de popups, la única diferencia es que en este no disponemos de un input si no de un mensaje avisando al usuario sobre si realmente quiere eliminar el fichero.

Esto se ve de la siguiente manera:



La visión general de este componente se ve de la siguiente forma:

In Thrash			
Name	Owner	Last Modified	File Size
2.6_research_innovation.pdf	rubenapruebame@gmail.com	2024-05-23 11:44:05	2.42MB

7.2.8.6. DataStorage.tsx

Este es el último de los componentes variables y se encarga de la visualización de gráficos en función de los diferentes ficheros subidos por el usuario.

Este componente está formado por cuatro gráficos distintos que, cada uno, representa información distinta.

El primer gráfico es de tipo *Gauge*. Este muestra la cantidad de cuota usada por el usuario respecto a la cantidad máxima que le ofrece su plan de suscripción.

El código de este gráfico es el siguiente:

```

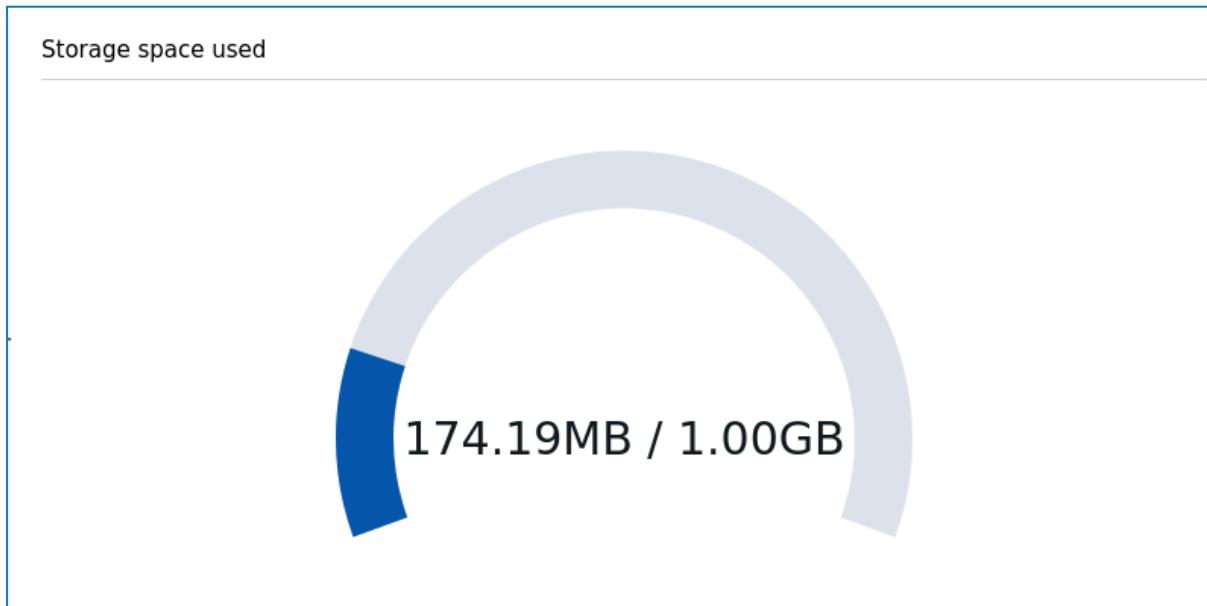
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
      <DataHeader>
        <div className="headerLeft">
          <text style={{color:
            mode === "light"
              ? "black"
              : 'rgba(190, 190, 190, 1)',
            fontSize: "15px"}>Storage space used</text>
        </div>
      </DataHeader>

      <Stack direction={{ xs: 'column', md: 'row' }} spacing={{ xs: 1, md: 3 }}>
        <Gauge width={400} height={300} value={(diskusage.current/diskmax)*100} startAngle={-110} endAngle={110}
          sx={{fontSize:30}}
          text={`${reformatSize(diskusage.current)} / ${reformatSize(diskmax)}}` />
      </Stack>
    
```

La primera parte del código es para alinear el título del gráfico. La segunda es el gráfico propiamente dicho.

Gracias al uso de una librería de componentes, solamente hemos de pasarle al gráfico los datos formateados con los que queremos que trabaje. Para esto le pasamos las variables “diskusage” y “diskmax”, más adelante veremos de dónde vienen estas variables.

El resultado de este gráfico es el siguiente:



El segundo gráfico utilizado es uno tipo PieChart y muestra la cantidad de ficheros que el usuario ha subido al sistema de cada tipo. Este gráfico está implementado de la siguiente forma:

```

182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
    <DataHeader>
        <div className="headerLeft">
            <text style={{color:
                mode === "light"
                    ? "black"
                    : 'rgba(190, 190, 190, 1)',
                fontSize: "15px"}}>File type uploaded</text>
        </div>
    </DataHeader>

    <Stack direction={{ xs: 'column', md: 'row' }} spacing={{ xs: 1, md: 3 }}>
        <PieChart
            series={[
                {
                    //@ts-ignore
                    data: mapTypes()
                },
            ]}
            width={600}
            height={350}
        />
    </Stack>

```

Para el correcto funcionamiento de este gráfico simplemente seleccionamos los valores a utilizar en el atributo “data”. Estos valores tienen que ordenarse para que el gráfico sea entendible para el usuario. Esto se consigue con la función *mapTypes()*, la cual está implementada de la siguiente manera:

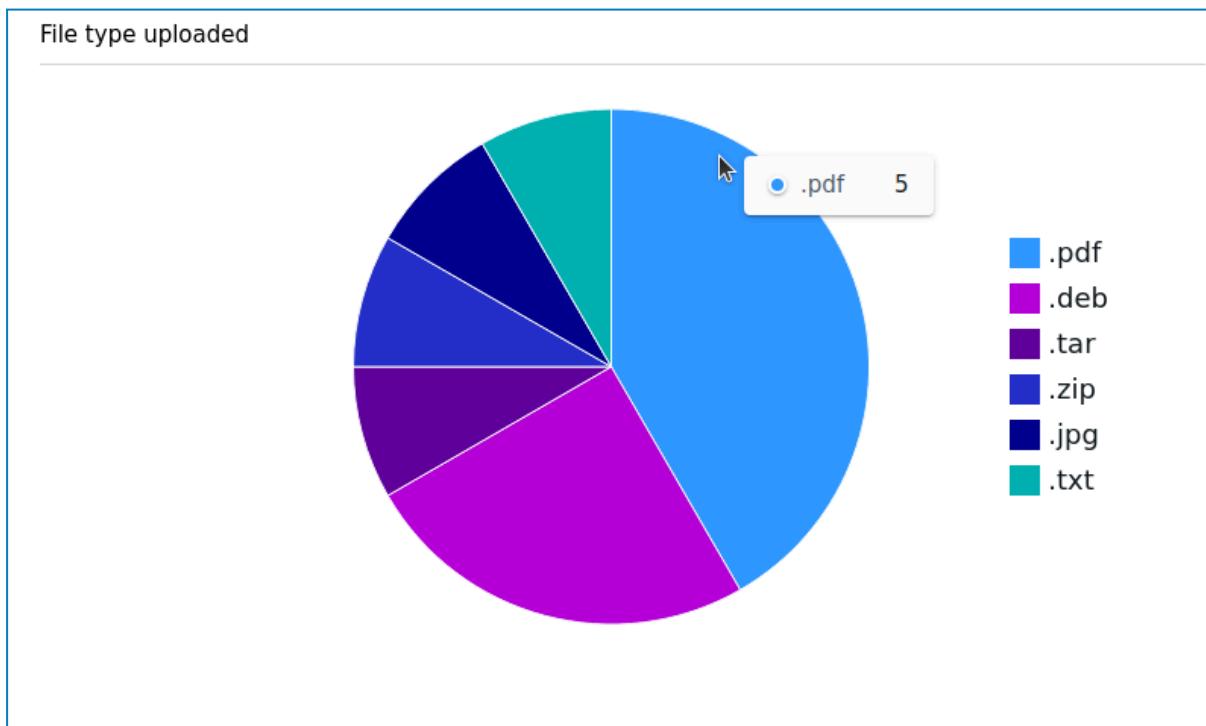
```

54     function mapTypes () {
55       if (filesArray.length > 0) {
56         var filetype = new Map()
57         for (var i = 0; i < filesArray.length; i++) {
58           // @ts-ignore
59           var offset = filesArray[i].nombreArchivo.length
60           // @ts-ignore
61           while (filesArray[i].nombreArchivo[offset] !== '.') {
62             --offset;
63             if (offset <= 0) break;
64           }
65           var format: any;
66           if (offset > 0) {
67             // @ts-ignore
68             format = filesArray[i].nombreArchivo.slice(offset,filesArray[i].nombreArchivo.length);
69           }
70           else {
71             format = "undefined"
72           }
73           if (filetype.has(format)) filetype.get(format).count++
74           else filetype.set(format,{count:1})
75         }
76         var res = []
77         var filetype2 = new Map([...filetype.entries()].sort((a,b) => b[1].count - a[1].count))
78         var pos = 0;
79         for (const [key, count] of filetype2) {
80           res.push({id: pos, value: count.count, label: key})
81           pos++
82         }
83       }
84     }
85   }
86 }
87 }
```

Esta función crea una variable de tipo mapa para guardar una tupla {tipo de fichero, cantidad de veces aparecido} y recorre el array de ficheros comprobando justamente esto.

Una vez ha recorrido el array de forma completa, ordena el mapa por la cantidad de ocurrencias.

El resultado del gráfico es el siguiente:



El tercer gráfico es de tipo LineChart y representa la cantidad de cuota utilizada a través del tiempo respecto a la cuota máxima. Este gráfico está implementado de la siguiente forma:

```

208          <DataHeader>
209            <div className="headerLeft">
210              <text style={{color:
211                mode === "light"
212                  ? "black"
213                  : 'rgba(190, 190, 190, 1)',
214                  fontSize: "15px"}}>Storage progression over time</text>
215        </div>
216      </DataHeader>
217      <LineChart
218        xAxis={[
219          {
220            id: 'Date',
221            data: datesLine,
222            scaleType: 'time',
223          },
224        ]}
225        series={[
226          {
227            id: 'Current',
228            label: 'Used space in MB',
229            data: sizesLine,
230            area: true,
231            showMark: false,
232            color: 'DodgerBlue'
233          },
234          {
235            id: 'Total',
236            label: 'Total space in MB',
237            data: new Array(sizesLine.length).fill(diskmax/1000000),
238            area: true,
239            showMark: false,
240            color: 'lightblue'
241          }
242        ]}
243        width={800}
244        height={300}
245        margin={{ left: 70 }}
246      />

```

Podemos ver que este es un gráfico bastante más complejo que los dos anteriores.

Para este gráfico es necesario establecer un conjunto de datos como eje X y otro conjunto de datos como el eje Y.

Los datos utilizados para el eje X son las fechas de los ficheros subidos por el usuario.

Estas fechas se obtienen llamando a la siguiente función:

```

130      function formatDatesLine () {
131        if (filesArray.length > 0) {
132          var newfiles= filesArray.sort(compareFn_rmod);
133          var res = []
134          for (var i = 0; i < newfiles.length; i++) {
135            //@ts-ignore
136            res.push(new Date(newfiles[i].fecha))
137          }
138        }
139        return res;
140      }
141    }

```

Esta función recorre el array de ficheros, ordenándolos por fecha de adición y devuelve un array de fechas formateadas extraídas a partir de la fecha original de cada fichero.

Para el eje Y es necesario calcular la suma progresiva del tamaño de los ficheros. Para esto se llama a la siguiente función:

```

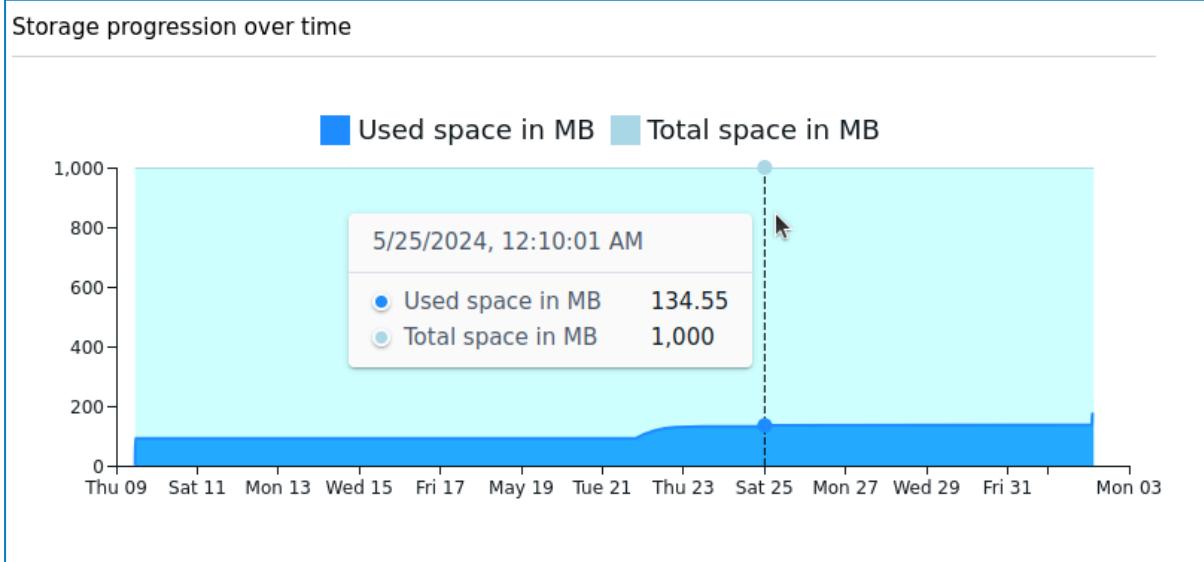
143     function formatSizesLine () {
144         if (filesArray.length > 0) {
145             var newfiles= filesArray.sort(compareFn_rmod);
146             //@ts-ignore
147             var res = [parseFloat(newfiles[0].tamano)/1000000];
148             for (var i = 1; i < newfiles.length; i++) {
149                 //@ts-ignore
150                 res.push(parseFloat(newfiles[i].tamano)/1000000+res[i-1]);
151             }
152             return res;
153         }
154     else return [];
155 }
```

Esta función recorre el array de ficheros ordenado cronológicamente. A medida que recorre este array, calcula que el valor de la posición actual será el valor de la posición anterior más el tamaño del fichero actual.

Al finalizar esta función tendremos un array del mismo tamaño que el mencionado anteriormente, solo que en este caso en vez de las fechas de cada fichero, tendremos el espacio acumulado progresivamente.

Como se ha mencionado previamente, este gráfico muestra el espacio usado acumulado respecto al máximo disponible. Para el cálculo de este máximo se ha creado una segunda serie de datos con la capacidad máxima en todas las posiciones.

El resultado de este gráfico es el siguiente:



El cuarto gráfico es de tipo BarChart y representa el espacio utilizado cada mes del año. Este gráfico está implementado de la siguiente forma:

```

249      <DataHeader>
250          <div className="headerLeft">
251              <text style={{color:
252                  mode === "light"
253                      ? "black"
254                      : 'rgba(190, 190, 190, 1)',
255                      fontSize: "15px"}}>Storage used per month</text>
256      </div>
257  </DataHeader>
258  <BarChart
259      dataset={dataset}
260      yAxis={[{ scaleType: 'band', dataKey: 'month' }]}
261      series={[{ dataKey: 'size', valueFormatter }]}
262      layout="horizontal"
263      {...chartSetting}
264      sx={{transform: 'translate(0px, -40px)', marginBottom: '-50px'}}}

```

Para poder pasárselos a este gráfico hubo que formatearlos de manera en que quedasen tuplas de {espacio usado, mes del año}.

Para esto se llama a la siguiente función:

```

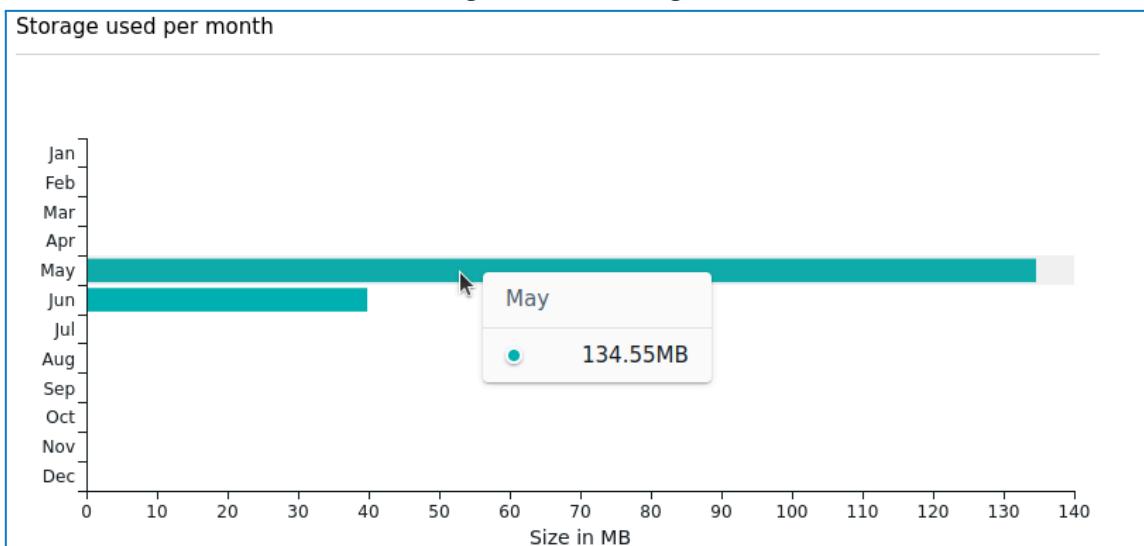
89  function formatDataset () {
90      var res = [{size:0,month:"Jan"},{size:0,month:"Feb"},{size:0,month:"Mar"},{size:0,month:"Apr"},
91      {size:0,month:"May"},{size:0,month:"Jun"},{size:0,month:"Jul"},{size:0,month:"Aug"},
92      {size:0,month:"Sep"},{size:0,month:"Oct"},{size:0,month:"Nov"},{size:0,month:"Dec"}]
93      if (filesArray.length > 0) {
94          for (var i = 0; i < filesArray.length; i++) {
95              // @ts-ignore
96              res[new Date(filesArray[i].fecha).getMonth()].size += parseFloat(filesArray[i].tamano)/1000000
97          }
98          for (var j = 0; j < res.length; j++) {
99              res[j].size = parseFloat(res[j].size).toFixed(2)
100         }
101     }
102     return res;

```

Esta función inicialmente crea un array de tuplas {espacio usado, mes del año} con el espacio usado a 0.

A medida que recorre el array de ficheros obtiene el mes del año en el que se han subido con la función interna `getMonth()` de un elemento tipo `Date` y va sumando el valor del tamaño de cada fichero.

El resultado de este gráfico es el siguiente:



7.2.8.7. StoragePage.tsx

Ahora que ya hemos visto la implementación de todos los componentes que forman parte del gran componente [StoragePage.tsx](#), podemos ver en detalle su implementación.

Inicialmente podemos ver las siguientes variables:

```
19 export default function StoragePage() {
20   const start_mode = (localStorage.getItem("mode") === null ? 'light' : localStorage.getItem("mode"));
21   //@ts-ignore
22   const [mode, setMode] = React.useState<PaletteMode>(start_mode);
23   const LPTheme = createTheme(getLPTheme(mode));
24   const [input_text, setInputTextState] = React.useState("");
25   const [side_selection, setSideSelection] = React.useState("MyFiles");
26
27   const [filesArray, setFilesArray] = React.useState([]);
28
29   var diskusage = useRef(5000000000);
30   var diskmax = useRef(1000000000);
```

Como podemos observar, en este componente padre de los explicados anteriormente, tenemos declaradas las variables “input_text” utilizada en el componente [Header.tsx](#), la variable “side_selection” utilizada en el componente [Sidebar.tsx](#), las variables “diskusage” y “diskmax” utilizadas en el componente [DataStorage.tsx](#) y sobretodo, la variable “filesArray” utilizada en prácticamente todos los componentes.

Estas variables están declaradas en esta clase común para que puedan ser compartidas y utilizadas por todos los componentes que cuelgan de este jerárquicamente.

Para la variable “filesArray” tenemos el siguiente uso:

```

38  const fetchData = async () => {
39    if (localStorage.getItem("email") === null) {
40      alert("An unexpected error occurred while trying to load your files. Please try again later.");
41      // @ts-ignore
42      window.location = '/sign-in';
43    }
44    else {
45      const userData = {
46        correo: localStorage.getItem("email"),
47        token: localStorage.getItem("token")
48      }
49      try {
50        const response = await fetch(`${backendUrl}/archivo/getall`, {
51          method: "POST",
52          headers: {
53            "Content-Type": "application/json",
54          },
55          body: JSON.stringify(userData),
56        });
57        if (response.ok) {
58          const data = await response.json();
59          setFilesArray(data.archivos);
60        }
61        else if (response.status === 404) {
62          console.log("No files yet")
63        }
64        else {
65          alert("An unexpected error occurred while trying to load your files. Please try again later.");
66          console.log(response.status)
67        }
68      } catch (error) {
69        console.error("Error at storage:", error);
70        alert("An unexpected error occurred while trying to load your files. Please try again later.");
71      }
72    }
73  };
74
75  useEffect(() => {
76
77    fetchData();
78
79  }, []);

```

Podemos ver que la variable “filesArray” se actualiza con el contenido de la respuesta por parte del backend.

La petición generada al backend está dentro de un *useEffect*, por lo que se ejecutará de forma dinámica. Esta petición es a la url especificada para obtener todos los archivos pertenecientes a un usuario, y los atributos necesarios a enviar en el body de esta petición son el correo y el token.

Analizando el código de la clase en general tenemos lo siguiente:

```

99  return []
100 <ThemeProvider theme={LPtheme}>
101   <CssBaseline />
102   <div style={{maxHeight:'100vh', overflowY:'hidden', overflowX:'hidden'}}>
103     <Header mode={mode} toggleColorMode={toggleColorMode} setInputTextState={setInputTextState}/>
104     <div style={{display: 'flex'}}>
105       {/*@ts-ignore*/}
106       <Sidebar mode={mode} diskmax={diskmax.current} setSideSelection={setSideSelection} filesArray={filesArray} setFilesArray={setFilesArray}>
107         <renderSelection()
108       </div>
109     </div>
110   </ThemeProvider>
111 
```

Como podemos ver, el código de este componente, gracias a haber separado todo el código explicado hasta ahora en diversos componentes, resulta muy sencillo, simplemente pinta los componentes fijos, es decir, el [Header.tsx](#) y el [Sidebar.tsx](#), y para decidir qué componente variable ha de pintar utiliza la siguiente función:

```

81   function renderSelection () {
82     switch (side_selection) {
83       case "MyFiles":
84         return <Data mode={mode} input_text={input_text} filesArray={filesArray} setFilesArray={setFilesArray} />
85       case "Shared":
86         return <DataShared mode={mode} input_text={input_text} />
87       case "Trash":
88         return <DataTrash mode={mode} input_text={input_text} filesArray={filesArray} setFilesArray={setFilesArray} />
89       case "Storage":
90         return <DataStorage mode={mode} input_text={input_text} filesArray={filesArray} setFilesArray={setFilesArray} />
91     }
92   }

```

Esta función simplemente, dependiendo del botón pulsado desde el [Sidebar.tsx](#), decide devolver un componente u otro.

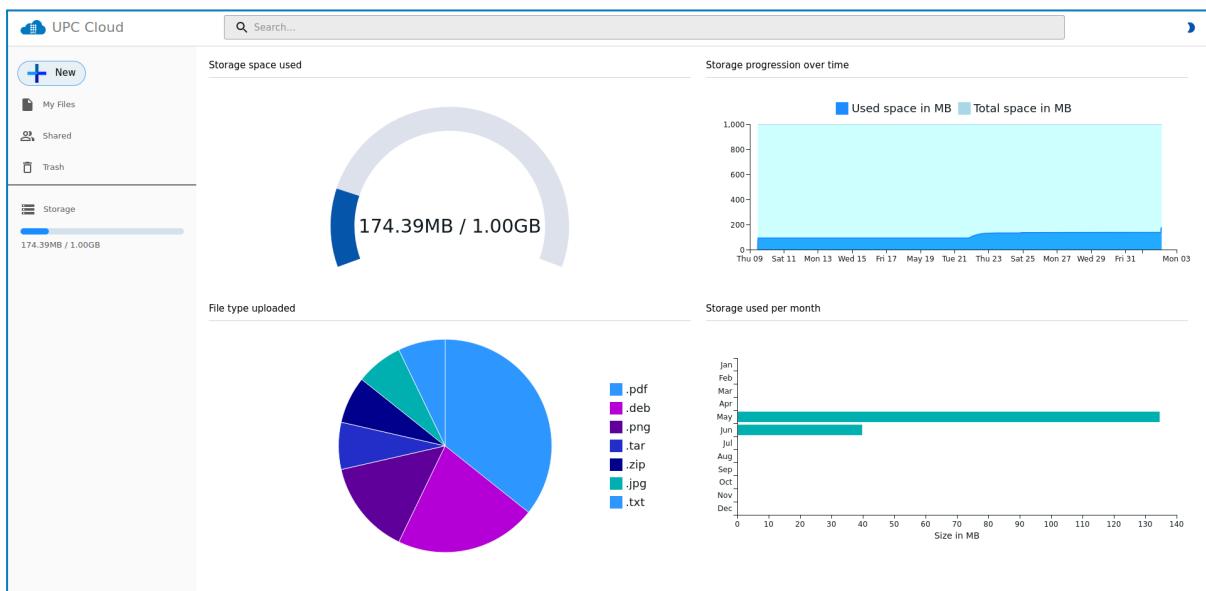
Por defecto devuelve el componente [Data.tsx](#).

La visualización final del componente [StoragePage.tsx](#) es la siguiente:

The screenshot shows a file management interface titled 'My Storage'. On the left, there's a sidebar with 'New', 'My Files', 'Shared', and 'Trash' buttons. Below them is a 'Storage' section showing '174.39MB / 1.00GB'. The main area displays a grid of file thumbnails and a detailed table below it. The table columns are 'Name', 'Owner', 'Last Modified', and 'File Size'. The files listed are:

Name	Owner	Last Modified	File Size
2.6research_innovation.pdf	rubenapruabame@gmail.com	2024-05-24 20:22:46	2.42MB
Enunciado-1.pdf	rubenapruabame@gmail.com	2024-05-25 00:09:44	669.36KB
Erlang.pdf	rubenapruabame@gmail.com	2024-05-21 20:03:17	464.26KB
google-chrome-stable_current_amd64.deb	rubenapruabame@gmail.com	2024-05-09 11:15:51	90.27MB
mongodb-org-server_7.0.8_amd64.deb	rubenapruabame@gmail.com	2024-05-22 19:20:40	36.69MB
mongodb-org-server_7.0.8_amd64.deb	rubenapruabame@gmail.com	2024-06-02 02:06:41	36.69MB
Screenshot from 2024-06-02 02:49:54.png	rubenapruabame@gmail.com	2024-06-02 02:51:53	96.51KB

Visualización del componente `StoragePage.tsx` con el componente variable `Data.tsx`.



Visualización del componente `StoragePage.tsx` con el componente variable `DataStorage.tsx`.

7.2.9. Tema claro y tema oscuro

Tal y como se ha podido ir viendo en los extractos de código anteriores, existe la posibilidad de cambiar de tema claro a oscuro y viceversa.

Para el correcto funcionamiento de esta funcionalidad, es necesario añadir las siguientes líneas de código en los componentes principales de las páginas mostradas, es decir, en [LandingPage.tsx](#), [SignUp.tsx](#), [SignIn.tsx](#) y [StoragePage.tsx](#):

```
const start_mode = (localStorage.getItem("mode") === null ? 'light' : localStorage.getItem("mode"));
const [mode, setMode] = React.useState<PaletteMode>(start_mode);
```

Este código revisa el local storage en busca de la variable "mode" guardada con el último valor de tema elegido por el usuario.

Esta variable será la que se utilizará en todos los componentes del código para decidir si pintar con colores claros u oscuros.

Un ejemplo de esto es el selector de color del popup Modal para subir ficheros del componente [Sidebar.tsx](#).

```
192     <Modal open={open} onClose={() => setOpen(false)} style={{ 
193       position: 'absolute',
194       top: '40vh',
195       left: '40vw',
196       height: '193px',
197       width: '520px',
198       backgroundColor: mode === 'light'
199       ? 'rgba(230, 230, 230, 1)'
200       : 'rgba(30, 30, 30, 1)',
201       border: '1px solid',
202       borderRadius: '10px',
203       borderColor: 'rgba(0,150,255,1)' }}>
```

Este componente tiene claramente un selector de colores en función de la variable del tema. En este caso, el componente [StoragePage.tsx](#) es el que obtiene el valor del local storage y se lo manda a este componente a través de la llamada a la función principal.

```
76 interface SideBarProps {
77   mode: PaletteMode;
78   diskmax: number;
79   setSideSelection: (arg0: string) => void;
80   filesArray: [];
81   setFilesArray: (arg0: []) => void
82 }
83 function Sidebar ({ mode, diskmax, setSideSelection, filesArray, setFilesArray }: SideBarProps) {
```

Podemos observar en este extracto de código correspondiente a la función principal del [Sidebar.tsx](#) cómo le llega la variable “mode: PaletteMode” desde el componente padre.

Para que esta funcionalidad pueda funcionar en su totalidad, es necesario llamar a la siguiente función en cada uno de los botones existentes para cambiar el tema de color:

```
const toggleColorMode = () => {
  setMode((prev) => (prev === 'dark' ? 'light' : 'dark'));
  localStorage.setItem("mode", mode === 'light' ? 'dark' : 'light');
};
```

Esta función, al igual que la variable “mode”, la reciben los componentes de sus padres a través de su función principal.

```
84 interface HeaderProps {
85   mode: PaletteMode;
86   toggleColorMode: () => void;
87   setInputTextState: (input_text: any) => void;
88 }
89
90 function Header ({ mode, toggleColorMode, setInputTextState}: HeaderProps) {
91
92   function handleChange (e) {
93     setInputTextState(e.target.value)
94   }
95
96   return (
97     <HeaderContainer>
98       <HeaderLogo>
99         <img src={logo} alt="UPC Cloud" />
100        <span>UPC Cloud</span>
101      </HeaderLogo>
102      <Search>
103        <SearchIconWrapper>
104          <SearchIcon />
105        </SearchIconWrapper>
106        <StyledInputBase
107          placeholder="Search..."
108          inputProps={{ 'aria-label': 'search' }}
109          onChange={handleChange}
110        />
111      </Search>
112      <div style = {{position:'relative', left:'169px', top:'0px'}}>
113        <ToggleColorMode mode={mode} toggleColorMode={toggleColorMode} />
114      </div>
115    </HeaderContainer>
116  )
117 }
```



Un ejemplo es la utilización de esta función en el componente [Header.tsx](#). La función `ToggleColorMode` a la que se está llamando en la línea 113 es realmente la declarada en el componente [StoragePage.tsx](#), el padre de este componente.

La visualización del tema oscuro es, por ejemplo, la siguiente:

The screenshot shows the UPC Cloud landing page. At the top, there's a navigation bar with links for Features, Specification, Highlights, Pricing, FAQ, Sign In, and Sign up. Below the navigation is a large "Welcome to UPC Cloud" heading with the UPC logo. A sub-headline reads: "Explore our cutting-edge online web-based storage system, delivering high-quality solutions tailored to your needs. Elevate your experience with top-tier features and services." There's a "start now" button and a note about agreeing to Terms & Conditions. The main area features a storage usage summary: "17.69MB / 1.00GB". To the right is a chart titled "Storage progression on time" showing used space in MB over time. On the left, there's a sidebar with New, My Files, Shared, and Trash options, and a Storage section showing 174.39MB / 1.00GB.

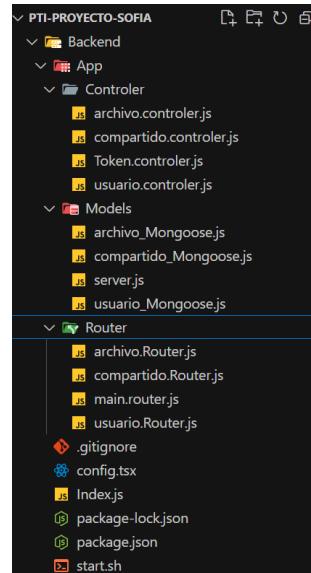
Visualización del componente `LandingPage.tsx`.

The screenshot shows the UPC Cloud Storage Page. The left sidebar includes New, My Files, Shared, and Trash sections, and a Storage section showing 174.39MB / 1.00GB. The main area has a "My Storage" heading and a search bar. It displays a grid of files: 2.research_innovation.pdf (PDF), Enunciado-1.pdf (PDF), Erlang.pdf (PDF), google-chrome-stable_current_amd64.deb (DEB), mongodb-org-server_7.0_8_amd64.deb (DEB), mongodb-org-server_7.0_8_amd64.deb (DEB), and Screenshot from 2024-0...png (Image). Below this is a detailed table of files:

Name	Owner	Last Modified	File Size
2.research_innovation.pdf	rubenapruebame@gmail.com	2024-05-24 20:22:46	2.42MB
Enunciado-1.pdf	rubenapruebame@gmail.com	2024-05-25 00:09:44	669.36KB
Erlang.pdf	rubenapruebame@gmail.com	2024-05-21 20:03:17	464.26KB
google-chrome-stable_current_amd64.deb	rubenapruebame@gmail.com	2024-05-09 11:15:51	90.27MB
mongodb-org-server_7.0_8_amd64.deb	rubenapruebame@gmail.com	2024-05-22 19:20:40	36.69MB
mongodb-org-server_7.0_8_amd64.deb	rubenapruebame@gmail.com	2024-06-02 02:06:41	36.69MB
Screenshot from 2024-06-02 02:49:54.png	rubenapruebame@gmail.com	2024-06-02 02:51:53	96.51KB

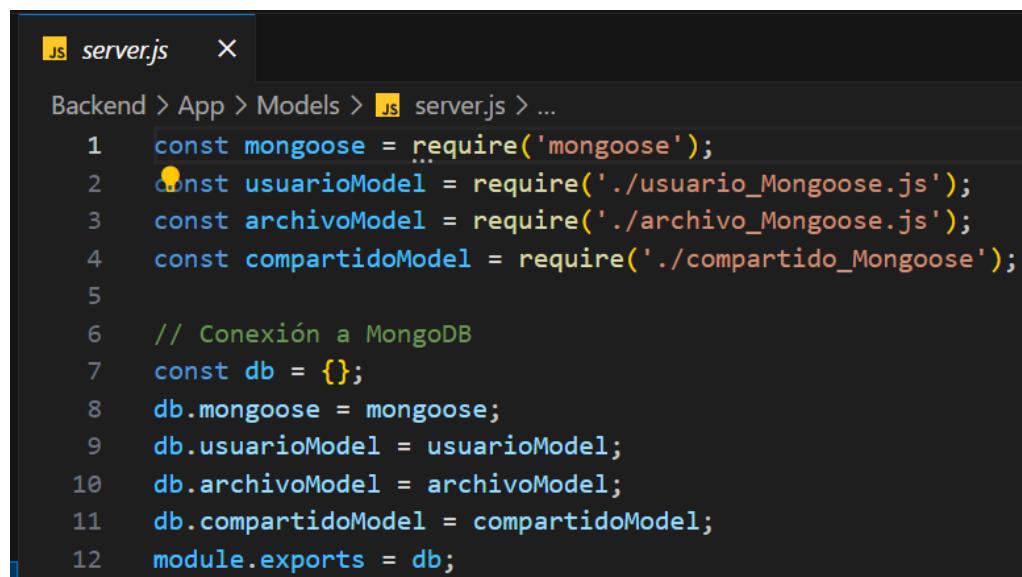
Visualización del componente `StoragePage.tsx` con el componente variable `Data.tsx` en tema oscuro.

7.3. BackEnd



7.3.1. Estructura de la Base de Datos

Se implementaron modelos en MongoDB utilizando Mongoose para definir los esquemas de datos de usuarios, archivos y compartidos. Estos esquemas están diseñados para almacenar información detallada sobre los usuarios, los archivos que suben y la manera en que estos archivos se comparten entre usuarios, incluyendo detalles como permisos de acceso y metadatos relacionados con IPFS. Estos modelos facilitan las interacciones con la base de datos y garantizan la integridad de los datos.



```
js server.js X
Backend > App > Models > js server.js > ...
1 const mongoose = require('mongoose');
2 const usuarioModel = require('./usuario_Mongoose.js');
3 const archivoModel = require('./archivo_Mongoose.js');
4 const compartidoModel = require('./compartido_Mongoose');
5
6 // Conexión a MongoDB
7 const db = {};
8 db.mongoose = mongoose;
9 db.usuarioModel = usuarioModel;
10 db.archivoModel = archivoModel;
11 db.compartidoModel = compartidoModel;
12 module.exports = db;
```

```
js archivo_Mongoose.js ×
Backend > App > Models > js archivo_Mongoose.js > ...
1 const mongoose = require("mongoose");
2 const Schema = mongoose.Schema;
3 const model = mongoose.model;
4
5 const archivo_Mongoose = new Schema({
6   "_id": mongoose.ObjectId,
7   "nombreArchivo": String,
8   "fecha": String,
9   "tamano": String,
10  "ipfs_hash": String,
11  "estado": { type: String, enum: ['activo', 'eliminado', 'archivado'], default: 'activo' },
12  propietario: {
13    type: mongoose.Schema.Types.ObjectId,
14    ref: 'Usuario',
15  },
16 }, { collection: "archivo" })
17
18 const archivo_MongooseModel = model("archivo_Mongoose", archivo_Mongoose);
19 module.exports = { archivo_Mongoose, archivo_MongooseModel };

js usuario_Mongoose.js ×
Backend > App > Models > js usuario_Mongoose.js > ...
1 const mongoose = require("mongoose");
2
3 const Schema = mongoose.Schema;
4 const model = mongoose.model;
5
6 const usuario_Mongoose = new Schema({
7   "_id": mongoose.ObjectId,
8   "nombre": String,
9   "clave": String,
10  "correo": String,
11 }, { collection: "usuario" });
12
13 const usuario_MongooseModel = model("usuario_Mongoose", usuario_Mongoose);
14
15 module.exports = { usuario_Mongoose, usuario_MongooseModel };

js compartido_Mongoose.js ×
Backend > App > Models > js compartido_Mongoose.js > ...
1 const mongoose = require("mongoose");
2 const Schema = mongoose.Schema;
3 const model = mongoose.model;
4
5 const compartido_Mongoose = new Schema({
6   "_id": mongoose.ObjectId,
7   "ArchivoCompartido": String,
8   "tamañoArchivo": String,
9   "correoCompartido": String, //Quien me comparte, a quien le comarto
10  "correoOwner": String,
11  "fechaCompartido": String,
12  "ipfs_hash": String,
13  "permisos": { type: String, enum: ['lectura', 'escritura'], default: 'lectura' },
14  archivoOriginal: {
15    type: mongoose.Schema.Types.ObjectId,
16    ref: 'archivo_Mongoose',
17  },
18 }, { collection: "Compartido" });
19
20 const compartido_MongooseModel = model("compartido_Mongoose", compartido_Mongoose);
21 module.exports = { compartido_Mongoose, compartido_MongooseModel };

```

Collection	Storage size:	Documents:	Avg. document size:	Indexes:	Total index size:
archivo	20.48 kB	2	159.00 B	1	36.86 kB
Compartido	20.48 kB	1	221.00 B	1	24.58 kB
usuario	20.48 kB	2	96.00 B	1	36.86 kB

7.3.2. Enrutamiento y Controladores

El servidor está configurado utilizando Express, proporcionando un marco de trabajo flexible para manejar las solicitudes HTTP. Se han definido rutas específicas para usuarios (usuario.Router.js), archivos (archivo.Router.js), y archivos compartidos (compartido.Router.js), cada una manejando operaciones CRUD para su respectiva entidad. Los controladores asociados (usuario.controler.js, archivo.controler.js, compartido.controler.js) implementan la lógica necesaria para interactuar con la base de datos y realizar las operaciones solicitadas, tales como crear, obtener, modificar y eliminar recursos.

- Main Router

```

main.router.js X
Backend > App > Router > main.router.js ...
1 const router = require('express').Router();
2 const usuarioRouter = require('../usuario.Router.js');
3 const archivoRouter = require('../archivo.Router.js');
4 const compartidoRouter = require('../compartido.Router.js');

5
6 module.exports = app => {
7   app.use('/usuario', usuarioRouter);
8   app.use('/archivo', archivoRouter);
9   app.use('/compartido', compartidoRouter);
10 }
11

```

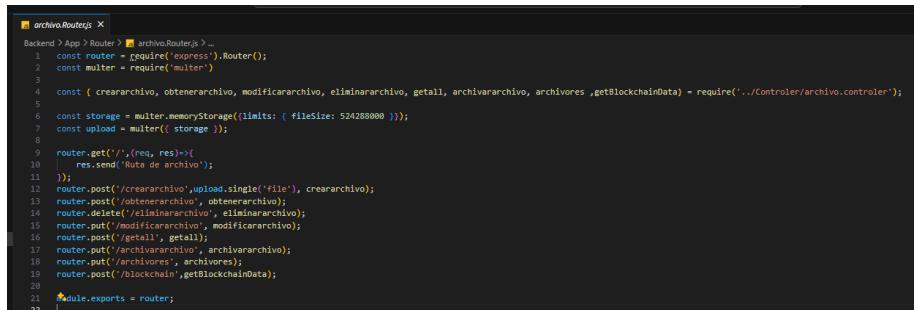
- Usuario Router

```

usuario.Router.js X
Backend > App > Router > usuario.Router.js ...
1 const router = require('express').Router();
2 const { crearusuario, obtenerusuario, eliminarusuario, modificarusuario, login } = require('../Controller/usuario.controller');
3
4 router.get('/', (req, res) => {
5   res.send('Ruta de usuario');
6 });
7
8 router.post('/crearusuario', crearusuario);
9 router.get('/obtenerusuario', obtenerusuario);
10 router.delete('/eliminarusuario', eliminarusuario);
11 router.put('/modificarusuario', modificarusuario);
12 router.post('/login', login);
13
14 module.exports = router;
15

```

- Archivo Router

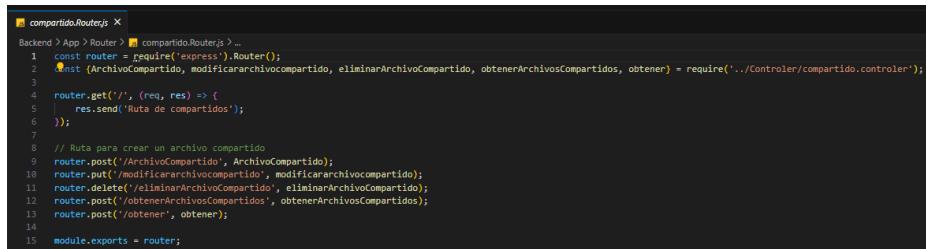


```

Backend > App > Router > archivo.Router.js > ...
1 const router = require('express').Router();
2 const multer = require('multer');
3
4 const { creararchivo, obtenerarchivo, modificararchivo, eliminararchivo, getall, archivararchivo, archivores ,getBlockchainData} = require('../Controler/archivo.controler');
5
6 const storage = multer.memoryStorage({limits: { fileSize: 52428800 }});
7 const upload = multer({storage});
8
9 router.get('/<req, res>',(req, res) => {
10   res.send('Ruta de archivo');
11 });
12 router.post('/creararchivo',upload.single('file'), creararchivo);
13 router.post('/obtenerarchivo', obtenerarchivo);
14 router.delete('/eliminararchivo', eliminararchivo);
15 router.put('/modificararchivo', modificararchivo);
16 router.post('/getall', getall);
17 router.put('/archivararchivo', archivararchivo);
18 router.put('/archivores', archivores);
19 router.post('/blockchain',getBlockchainData);
20
21 module.exports = router;
22

```

- Compartido Router



```

Backend > App > Router > compartido.Router.js > ...
1 const router = require('express').Router();
2 const { ArchivoCompartido, modificarArchivoCompartido, eliminarArchivoCompartido, obtenerArchivosCompartidos, obtener } = require('../Controler/compartido.controler');
3
4 router.get('/', (req, res) => {
5   res.send('Ruta de compartidos');
6 });
7
8 // Ruta para crear un archivo compartido
9 router.put('/modificarArchivoCompartido', modificarArchivoCompartido);
10 router.put('/eliminarArchivoCompartido', eliminarArchivoCompartido);
11 router.delete('/eliminarArchivoCompartido', eliminarArchivoCompartido);
12 router.post('/obtenerArchivosCompartidos', obtenerArchivosCompartidos);
13 router.post('/obtener', obtener);
14
15 module.exports = router;

```

- Consultas Usuario

- crearusuario

La función `crearusuario` es un controlador en Node.js diseñado para manejar la creación de nuevos usuarios en una aplicación. Cuando se invoca, esta función toma datos del cuerpo de la solicitud HTTPS, específicamente el nombre, clave (contraseña) y correo del usuario. Primero verifica si ya existe un usuario con el mismo correo usando `usuario.Usuario_MongooseModel.findOne({ correo })`. Si el usuario ya existe, devuelve un error con un código de estado 400 indicando que el usuario ya existe. Si no, crea un nuevo usuario en la base de datos de MongoDB utilizando el modelo `usuario_MongooseModel`, asignándole un ID único generado por `mongoose.Types.ObjectId()`, y guarda este nuevo usuario en la base de datos. Si el registro es exitoso, responde con un código de estado 201 y un mensaje de confirmación. En caso de errores durante el proceso, captura y registra estos errores, y responde con un código de estado 500 indicando un error interno del servidor.

```

7  const crearusuario = async (req, res) => {
8    const { nombre, clave, correo } = req.body;
9    try {
10      const usuarioExistente = await usuario.usuario_MongooseModel.findOne({ correo });
11      if (usuarioExistente) {
12        return res.status(400).send('El usuario ya existe');
13      }
14      const nuevoUsuario = new usuario.usuario_MongooseModel({
15        _id: new mongoose.Types.ObjectId(),
16        nombre,
17        clave,
18        correo
19      });
20      await nuevoUsuario.save();
21      res.status(201).send('Usuario registrado correctamente');
22    } catch (error) {
23      console.error('Error al registrar usuario:', error);
24      res.status(500).send('Error interno del servidor: ' + error.message);
25    }
26  };

```

- **obtenerusuario**

La función `obtenerusuario` es un controlador asincrónico en Node.js que facilita la recuperación de un usuario específico de una base de datos MongoDB. Funciona extrayendo el correo electrónico del cuerpo de una solicitud HTTP. Si el correo está presente en la solicitud, el sistema intenta localizar un usuario en la base de datos con ese correo utilizando `usuario.usuario_MongooseModel.findOne({ correo })`. Si encuentra un usuario, el sistema lo registra en la consola y devuelve los datos del usuario como una respuesta JSON al cliente. Si no se encuentra ningún usuario con el correo proporcionado, devuelve un error con un código de estado 404, indicando que el usuario no fue encontrado. Si ocurre un error durante el proceso de búsqueda en la base de datos, captura este error, lo registra y devuelve un mensaje de error general con un código de estado 500, señalando un problema interno del servidor.

```

const obtenerusuario = async (req, res) => {
  try {
    const { correo } = req.body;
    if (correo) {
      const usuarioExistente = await usuario.usuario_MongooseModel.findOne({ correo });
      if (usuarioExistente) {
        console.log(usuarioExistente);
        return res.json(usuarioExistente);
      } else {
        return res.status(404).send('Usuario no encontrado');
      }
    }
  } catch (error) {
    console.error('Error al obtener usuarios:', error);
    res.status(500).send('Error interno del servidor: ' + error.message);
  }
}

```

- **eliminarusuario**

La función `eliminarusuario` es un controlador asincrónico en Node.js diseñado para manejar la eliminación de usuarios de una base de datos MongoDB basada en la autenticación de token y el correo electrónico del usuario. La función primero verifica la validez del token enviado con la solicitud HTTP usando la función `validartoken(token, correo)`. Si el token es

válido y corresponde al correo del usuario que se intenta eliminar, procede a verificar si se ha proporcionado un correo. Si el correo está presente, intenta eliminar el usuario correspondiente utilizando el método `usuario.usuario_MongooseModel.findOneAndDelete({ correo })`.

Si el usuario es encontrado y eliminado exitosamente, devuelve una confirmación de que el usuario ha sido eliminado correctamente. Si no se encuentra ningún usuario con ese correo, responde con un error 404 indicando que el usuario no fue encontrado. Si el correo no se proporciona en la solicitud, devuelve un error 400 indicando que el correo no fue proporcionado. Si el token no es válido, devuelve un error 401 señalando que el token no es válido.

En caso de que surja un error durante cualquier parte del proceso, como un fallo en la consulta de la base de datos o en la validación del token, la función captura este error, lo registra y envía una respuesta de error 500 al cliente, indicando que ha ocurrido un error interno del servidor.

```
const eliminarusuario = async (req, res) => {
  try {
    const { correo, token } = req.body;
    const tokenValido = await validartoken(token, correo);
    if (tokenValido) {
      if (correo) {
        const usuarioEliminado = await usuario.usuario_MongooseModel.findOneAndDelete({ correo });
        if (usuarioEliminado) {
          return res.send('Usuario eliminado correctamente');
        } else {
          return res.status(404).send('Usuario no encontrado');
        }
      } else {
        return res.status(400).send('Correo no proporcionado');
      }
    } else {
      return res.status(401).send('Token no válido');
    }
  } catch (error) {
    console.error('Error al eliminar usuario:', error);
    res.status(500).send('Error interno del servidor: ' + error.message);
  }
};
```

- **modificarusuario**

La función `modificarusuario` en Node.js gestiona la actualización de los datos de un usuario en una base de datos MongoDB, verificando primero la autenticidad mediante un token. El proceso inicia validando el token proporcionado con la función `validartoken(token, correo)`. Si el token es válido, verifica que se haya suministrado un correo y, de ser así, busca al usuario correspondiente en la base de datos. Si encuentra al usuario, actualiza su nombre y clave con los nuevos valores proporcionados en la solicitud, si estos se han incluido; de lo contrario, mantiene los valores actuales. Tras actualizar los datos, guarda los cambios en la base de datos. Si el usuario es modificado con éxito, envía una respuesta

indicando que la modificación se realizó correctamente. Si no encuentra al usuario o si el correo no se proporciona, devuelve un error adecuado. Si ocurre algún error durante el proceso, lo registra y devuelve un mensaje de error interno del servidor.

```
const modificarusuario = async (req, res) => {
  try {
    const { correo, nuevoNombre, nuevaClave, token } = req.body;
    const tokenValido = await validartoken(token, correo);
    if (tokenValido) {
      if (correo) {
        const usuarioExistente = await usuario.usuario_MongooseModel.findOne({ correo });
        if (usuarioExistente) {
          usuarioExistente.nombre = nuevoNombre || usuarioExistente.nombre;
          usuarioExistente.clave = nuevaClave || usuarioExistente.clave;
          await usuarioExistente.save();
          return res.send('Usuario modificado correctamente');
        } else {
          return res.status(404).send('Usuario no encontrado');
        }
      } else {
        return res.status(400).send('Correo no proporcionado');
      }
    }
  } catch (error) {
    console.error('Error al modificar usuario:', error);
    res.status(500).send('Error interno del servidor: ' + error.message);
  }
};
```

- login

La función `login` es un controlador asíncrono en Node.js diseñado para manejar el proceso de autenticación de usuarios en una aplicación. Este método verifica las credenciales de los usuarios, específicamente correo y clave, proporcionadas a través de una solicitud HTTP. Al recibir los datos, verifica que tanto el correo como la clave hayan sido proporcionados. Si ambos están presentes, busca en la base de datos MongoDB un usuario que coincida exactamente con estas credenciales usando `usuario.usuario_MongooseModel.findOne({ correo, clave })`.

Si encuentra un usuario que coincide con las credenciales proporcionadas, genera un token de autenticación para ese usuario utilizando la función `token.crearToken(usuarioExistente)`, lo que facilita la gestión de sesiones y la autenticación en solicitudes futuras. Este token se envía de vuelta al usuario en formato JSON, indicando un inicio de sesión exitoso. Si no se encuentra ningún usuario con las credenciales dadas, se devuelve un mensaje de error 404 indicando que el usuario no fue encontrado. En caso de que las credenciales no se proporcionen completamente, se devuelve un error 400 indicando que faltan el correo o la clave. Si ocurre un error durante el proceso de consulta a la base de datos o durante la generación del token, se captura este error y se devuelve un mensaje de error 500, indicando un fallo interno del servidor.

```

const login = async (req, res) => {
  try {
    const { correo, clave } = req.body;
    if (correo && clave) {
      const usuarioExistente = await usuario.Usuario_MongooseModel.findOne({ correo, clave });
      if (usuarioExistente) {
        const tokenres = await token.crearToken(usuarioExistente);
        console.log("Conectado");
        return res.send(JSON.stringify({ message: tokenres }));
      } else {
        return res.status(404).send('Usuario no encontrado');
      }
    } else {
      return res.status(400).send('Correo o clave no proporcionados');
    }
  } catch (error) {
    console.error('Error al iniciar sesión:', error);
    res.status(500).send('Error interno del servidor: ' + error.message);
  }
};

```

- Consultas Archivo

- creararchivo

```

9
10 const creararchivo = async (req, res) => {
11   const file = req.file
12   const jsonreq = JSON.parse(req.body.JSON)
13   const correo = jsonreq.correo
14   const token = jsonreq.token
15
16   try{
17     if (!file){
18       return res.status(400).send('No se ha enviado archivo');
19     }
20     const nombreArchivo= file.originalname
21     const tamano = file.size
22
23     const tokenValido = await Token.validartoken(token, correo);
24     if (!tokenValido) {
25       return res.status(500).send('El token no es valido');
26     }
27
28     const usuarioExistente = await usuario.Usuario_MongooseModel.findOne({ correo });
29     if (!usuarioExistente) {
30       return res.status(500).send('El propietario especificado no existe');
31     }
32
33     const blob = new Blob([file.buffer],{type:file.mimetype})
34     formData = new FormData()
35     formData.append('file',blob,nombreArchivo)
36     const response = await fetch( 'http://10.4.41.54:9095/api/v0/add' , {
37       method: "POST",
38       body: formData
39     });
40     if (response.ok) {
41       const data = await response.json();
42       var ipfs_hash = data["Hash"]
43       console.log("Upload successful :)");
44
45       const nuevoArchivo = new archivo.archivo_MongooseModel({
46         _id: new mongoose.Types.ObjectId(),
47         nombreArchivo,
48         fecha: dayjs(Date.now()).format('YYYY-MM-DD HH:mm:ss'),
49         tamano,
50         ipfs_hash,
51         estado: 'activo',
52         propietario: usuarioExistente._id,
53         permisos: 'escritura'
54       });
55       await nuevoArchivo.save();
56       res.status(201).send('Archivo creado correctamente');
57     }
58     else {
59       console.error("Error at upload :(");
60       throw "IPFS ERROR response"
61     }
62   }
63   catch(error){
64     console.error('Error: ',error)
65     res.status(500).send('Error interno del servidor: ' + error);
66   }
67 };
68

```

La función `creararchivo` en Node.js es un controlador asincrónico diseñado para manejar la carga de archivos de los usuarios en el sistema y su almacenamiento en el sistema IPFS (InterPlanetary File System). Inicia verificando la presencia de un archivo en la solicitud entrante y desempaquetá la información del usuario y el token de autenticidad desde el cuerpo de la solicitud. Tras validar el token y confirmar la existencia del usuario en la base de datos, procede a preparar el archivo para su carga, encapsulando el buffer del archivo en un objeto `Blob` y utilizando `FormData` para prepararlo para la transmisión. El archivo se envía a un servidor IPFS mediante una solicitud POST. Si la carga es exitosa y se recibe una respuesta afirmativa del servidor IPFS, se guarda un nuevo registro en la base de datos MongoDB con detalles como el nombre del archivo, el tamaño, la fecha de carga, y el hash IPFS del archivo, marcando el estado del archivo como 'activo' y asignándole permisos de escritura. En caso de errores en cualquier etapa del proceso, captura y registra estos errores, devolviendo un mensaje de error interno del servidor.

- obtenerarchivo

La función `obtenerarchivo` es un controlador asincrónico en Node.js diseñado para recuperar un archivo específico de la base de datos MongoDB y proporcionar un enlace de descarga a través de IPFS. Primero verifica la validez del token de autenticación enviado junto con la solicitud, asegurándose de que el usuario tenga permisos para acceder al archivo solicitado. Si el token es válido, procede a buscar en la base de datos el archivo específico usando el identificador `_id` proporcionado. Si el archivo es encontrado, genera y envía al usuario un enlace directo para descargar el archivo desde el servidor IPFS, utilizando el hash IPFS almacenado y el nombre del archivo como parte del enlace. Si no se encuentra el archivo, devuelve un error 404 indicando que el archivo no fue encontrado. En caso de errores durante la búsqueda en la base de datos o en la generación del enlace, captura y registra estos errores, devolviendo un mensaje de error 500, señalando un fallo interno del servidor.

```
const obtenerarchivo = async (req, res) => {
  const {_id, correo, token} = req.body;
  const tokenValido = await Token.validartoken(token, correo);
  if (tokenValido) {
    try {
      const archivoEncontrado = await archivo.archivo.MongooseModel.findById({_id});
      if (archivoEncontrado) {
        return res.status(201).send(`350W.Stringify({message:'http://10.4.41.54:8080/ipfs/' + archivoEncontrado.ipfs_hash'?'download=true&filename=' + archivoEncontrado.nombreArchivo})`);
      } else {
        return res.status(404).send('Archivo no encontrado');
      }
    } catch (error) {
      console.error(`Error al obtener archivo: ${error}`);
      res.status(500).send(`Error interno del servidor: ${error.message}`);
    }
  }
};
```

- **getall**

La función `getall` en Node.js está diseñada para recuperar y devolver todos los archivos asociados a un usuario específico en la base de datos MongoDB. La función primero valida el token de autenticación suministrado con la solicitud para verificar que el usuario tenga permisos para acceder a esta información. Si el token es válido, la función procede a buscar el usuario por su correo electrónico y, si lo encuentra, busca todos los archivos que están asociados con el ID de ese usuario como propietario en la colección de archivos.

Durante la consulta de archivos, se omite la selección de ciertos campos como `__v` y `propietario` para optimizar la respuesta y enfocarse en la información relevante. Si se encuentran archivos, modifica cada objeto de archivo para incluir el correo del usuario y luego estructura la respuesta para incluir el ID del propietario y la lista modificada de archivos. Si la búsqueda es exitosa y hay archivos disponibles, envía esta información al usuario en formato JSON. En caso contrario, si no se encuentran archivos, devuelve un error 404 indicando que no se encontraron archivos para ese usuario. Si ocurre algún error durante el proceso de búsqueda o durante la manipulación de datos, la función captura y registra este error, devolviendo un mensaje de error 500, señalando un problema interno del servidor.

```
const getall = async (req, res) => {
  const { correo, token } = req.body;
  const tokenValido = await Token.validartoken(token, correo);
  if (tokenValido) {
    try {
      const usuarioExistente = await usuario.Usuario_MongooseModel.findOne({ correo }).select('-__v');
      const archivosEncontrados = await archivo.Archivo_MongooseModel
        .find({ propietario: usuarioExistente._id })
        .select('__v -propietario');
      if (archivosEncontrados.length > 0) {
        const archivosModificados = archivosEncontrados.map(archivo => [
          const archivoObj = archivo.toObject ? archivo.toObject() : archivo;
          return { ...archivoObj, correo: usuarioExistente.correo };
        ]);
        const respuesta = {
          propietario: usuarioExistente._id,
          archivos: archivosModificados
        };
        return res.json(respuesta);
      } else {
        return res.status(404).send('Archivos no encontrados para el usuario');
      }
    } catch (error) {
      console.error('Error al obtener los archivos:', error);
      res.status(500).send('Error interno del servidor: ' + error.message);
    }
  }
};
```

- **modificararchivo**

La función `modificararchivo` en Node.js está diseñada para actualizar el nombre de un archivo específico en la base de datos MongoDB. Primero, verifica la validez del token de autenticación proporcionado para asegurarse de que el usuario que realiza la solicitud tenga permiso para modificar el archivo. Si el token es válido, la función busca en la base de

datos el archivo específico utilizando el identificador `_id` proporcionado en la solicitud.

Si encuentra el archivo, procede a actualizar la fecha de modificación y el nombre del archivo con los nuevos valores proporcionados, siendo `nuevonombreArchivo` el nuevo nombre asignado al archivo. Después de realizar estas actualizaciones, guarda los cambios en la base de datos. Si el archivo se actualiza con éxito, envía una respuesta al cliente confirmando que el archivo ha sido modificado correctamente.

Si no encuentra el archivo en la base de datos, devuelve un error 404 indicando que el archivo no fue encontrado. Cualquier error que ocurra durante el proceso de búsqueda o actualización en la base de datos es capturado y registrado, y se envía al cliente un mensaje de error 500 indicando un error interno del servidor.

```
const modificararchivo = async (req, res) => {
  const { _id, nuevonombreArchivo, correo, token } = req.body;
  const tokenValido = await Token.validartoken(token, correo);
  if (tokenValido) {
    try {
      const archivoExistente = await archivo.archivo_MongooseModel.findOne({ _id });
      if (!archivoExistente) {
        return res.status(404).send('Archivo no encontrado');
      }
      archivoExistente.fecha = dayjs(Date.now()).format('YYYY-MM-DD HH:mm:ss'),
      archivoExistente.nombreArchivo = nuevonombreArchivo;
      await archivoExistente.save();
      return res.send('Archivo modificado correctamente');
    } catch (error) {
      console.error('Error al modificar archivo:', error);
      res.status(500).send('Error interno del servidor: ' + error.message);
    }
  }
};
```

- **eliminararchivo**

La función `eliminararchivo` en Node.js está diseñada para eliminar un archivo específico de la base de datos MongoDB y también intentar desanclarlo del nodo IPFS asociado. Primero verifica la validez del token de autenticación proporcionado usando la función `validartoken`, para asegurarse de que el usuario tiene permisos adecuados para realizar esta operación. Si el token es válido, procede a buscar y eliminar el archivo en la base de datos utilizando el método `findOneAndDelete`, basado en el identificador `_id` proporcionado en la solicitud.

Si encuentra y elimina el archivo con éxito, intenta desanclar el archivo del nodo IPFS utilizando una solicitud DELETE al endpoint IPFS, pasando el hash del archivo eliminado como parte de la URL. Esta acción es importante para asegurar que el archivo no solo sea eliminado de la base de datos, sino

también del almacenamiento distribuido en IPFS, liberando así recursos y manteniendo la coherencia de datos.

Si el archivo no se encuentra en la base de datos, devuelve un error 404 indicando que el archivo no fue encontrado. Si la eliminación en la base de datos es exitosa pero surge algún problema durante el proceso de desanclaje en IPFS, o si hay algún otro error durante el proceso, captura y registra este error, enviando una respuesta de error 500 al cliente para indicar un error interno del servidor.

```
const eliminararchivo = async (req, res) => {
  const {correo, token, _id} = req.body;
  const tokenValido = await Token.validartoken(token, correo);
  if (tokenValido) {
    try {
      const archivoEliminado = await archivo.archivo_MongooseModel.findOneAndDelete({ _id });
      if (!archivoEliminado) {
        return res.status(404).send('Archivo no encontrado');
      }
      const hash=archivoEliminado.hash;
      const response = await fetch("http://10.4.41.54:8080/ipfs/"+"pins/"+hash,{ 
        method: "delete",
      });
      //await compartido.deleteMany({ ArchivoCompartido: _id });
      res.send('Archivo eliminado correctamente');
    } catch (error) {
      console.error('Error al eliminar archivo:', error);
      res.status(500).send('Error interno del servidor: ' + error.message);
    }
  }
};
```

- archivararchivo

La función `archivararchivo` en Node.js se encarga de cambiar el estado de un archivo específico en la base de datos MongoDB a "archivado", lo que simula el envío del archivo a una papelera de reciclaje virtual dentro de la aplicación. Este proceso comienza verificando la validez del token de autenticación proporcionado, utilizando la función `Token.validartoken`, para confirmar que el usuario tiene permiso para realizar esta acción sobre el archivo.

Si el token es válido, la función procede a buscar y actualizar el estado del archivo especificado mediante el identificador `_id` proporcionado en la solicitud. Utiliza el método `findOneAndUpdate` para actualizar el campo de estado del archivo a 'archivado', asegurando que cualquier cambio devuelva la versión actualizada del documento con la opción `{ new: true }`.

Si el archivo se encuentra y se actualiza correctamente, envía una respuesta confirmando que el archivo ha sido archivado correctamente. Si no encuentra el archivo en la base de datos, devuelve un error 404 indicando que el archivo no fue encontrado. Además, esta función podría incluir la capacidad de eliminar cualquier referencia compartida del archivo en la

colección `compartido`, aunque esto está comentado en el código actual y podría ser implementado si es necesario.

Si ocurre algún error durante el proceso de actualización en la base de datos, captura y registra este error, enviando un mensaje de error 500 al cliente, lo que indica un fallo interno del servidor. Esta función es útil para gestionar la conservación de datos sin eliminar completamente los archivos, permitiendo una recuperación fácil si se decide restaurarlos posteriormente.

```
const archivararchivo = async (req, res) => {
    const {correo, token, _id} = req.body;
    const tokenValido = await Token.validartoken(token, correo);
    if (tokenValido) {
        try {
            const archivoArchado = await archivo.archivo_MongooseModel.findOneAndUpdate(
                {_id},
                {estado: 'archivado'},
                {new: true}
            );
            if (!archivoArchado) {
                return res.status(404).send('Archivo no encontrado');
            }
            //await compartido.deleteMany({ ArchivoCompartido: _id });
            res.send('Archivo archivado correctamente');
        } catch (error) {
            console.error('Error al archivar archivo:', error);
            res.status(500).send('Error interno del servidor: ' + error.message);
        }
    }
};
```

- archivores

La función `archivores` en Node.js está diseñada para restaurar un archivo previamente archivado, cambiando su estado de "archivado" a "activo" dentro de la base de datos MongoDB. La función comienza validando el token de autenticación proporcionado usando la función `Token.validartoken`, para asegurarse de que el usuario que realiza la solicitud tenga permisos adecuados para acceder y modificar el archivo.

Una vez confirmada la validez del token, la función intenta actualizar el estado del archivo especificado por el identificador `_id` en la solicitud. Utiliza el método `findOneAndUpdate` para cambiar el estado del archivo a 'activo', lo que indica que el archivo está nuevamente disponible y accesible dentro del sistema. La opción `{ new: true }` asegura que la respuesta incluya la versión actualizada del documento.

Si el archivo se encuentra y se actualiza con éxito, envía una confirmación al usuario de que el archivo ha sido restaurado correctamente. Si el archivo no se encuentra en la base de

datos, devuelve un error 404, indicando que el archivo no fue encontrado.

En caso de que surja algún error durante el proceso de búsqueda o actualización en la base de datos, como un fallo en la conexión o problemas con el servidor de datos, la función captura y registra este error, enviando una respuesta de error 500 al cliente, indicando un problema interno del servidor. Esta función es crucial para permitir a los usuarios revertir la acción de archivar y mantener la integridad y accesibilidad de sus archivos en el sistema.

```
const archivores = async (req, res) => {
  const {correo, token, _id} = req.body;
  const tokenValido = await Token.validartoken(token, correo);
  if (tokenValido) {
    try {
      const archivoresta = await archivo.archivo_MongooseModel.findOneAndUpdate(
        { _id },
        { estado: 'activo' },
        { new: true }
      );
      if (!archivoresta) {
        return res.status(404).send('Archivo no encontrado');
      }
      res.send('Archivo restaurado correctamente');
    } catch (error) {
      console.error('Error al restaurar archivo:', error);
      res.status(500).send('Error interno del servidor: ' + error.message);
    }
  }
};
```

- getBlockchainData

```
const getBlockchainData = async(req,res)=>{
  const { _id, correo, token } = req.body;
  const tokenValido = await Token.validartoken(token, correo);
  if (tokenValido) {
    try{
      var TimesShared = 0;
      var TimesViewed = 0;
      const respuesta = {
        shared: TimesShared,
        viewed: TimesViewed
      };
      return res.json(respuesta);
    }
    catch(error){
      console.error("blockchain error: ",error)
      const respuesta = {
        shared: 0,
        viewed: 0
      };
      return res.json(respuesta);
    }
  }
};
```

La función `getBlockchainData` en Node.js está diseñada para proporcionar datos estadísticos de un archivo específico, particularmente la cantidad de veces que el archivo ha sido compartido (`TimesShared`) y visto (`TimesViewed`). Inicia verificando la validez del token de autenticación proporcionado mediante la función `Token.validartoken` para asegurar que el usuario tenga permisos adecuados para acceder a esta información.

Una vez que se confirma que el token es válido, la función intenta recuperar y devolver los datos relacionados con cuántas veces se ha compartido y visto el archivo. Sin embargo, como se muestra en el cuerpo de la función, actualmente los valores de `TimesShared` y `TimesViewed` están establecidos en cero y no hay una integración funcional con una base de datos o sistema blockchain que proporcione estos datos dinámicamente. En su estado actual, simplemente devuelve estos valores predeterminados.

En caso de que ocurra un error durante el proceso, como una falla en la conexión a la base de datos o al sistema blockchain, la función captura y registra el error, devolviendo una respuesta con los valores predeterminados de `TimesShared` y `TimesViewed` establecidos en cero. Esto asegura que el usuario reciba una respuesta consistente independientemente de los problemas técnicos que puedan surgir.

Esta función está para monitorear la popularidad y el uso de un archivo dentro de la plataforma, aunque requiere una implementación completa que conecte realmente blockchain para extraer información en tiempo real sobre la interacción de los usuarios con los archivos.

- Consultas Compartido

- ArchivoCompartido

La función `ArchivoCompartido` en Node.js está diseñada para facilitar el proceso de compartir archivos entre usuarios dentro de la aplicación. Esta función comienza verificando la validez del token de autenticación proporcionado usando la función `Token.validartoken`, para asegurar que el usuario tenga permisos adecuados para compartir archivos.

Si el token es validado exitosamente, la función procede a buscar el archivo específico por su identificador (`_id` del archivo a compartir), usando el modelo `archivo.archivo_MongooseModel`. Si el archivo es encontrado, recopila datos relevantes como el nombre del

archivo, el hash de IPFS, y el tamaño del archivo para ser utilizados en el registro del archivo compartido.

Un nuevo registro es entonces creado en el modelo `compartido.compartido_MongooseModel`, donde se almacena información como el nombre del archivo, tamaño, hash de IPFS, correo del usuario que comparte el archivo, correo del propietario original del archivo, fecha en la que se comparte el archivo, y los permisos asignados (por ejemplo, permisos de 'lectura' o 'escritura'). Este nuevo registro también incluye una referencia al archivo original a través de su `'_id`.

Una vez que la información está completa y el nuevo archivo compartido es guardado en la base de datos, la función devuelve una respuesta positiva indicando que el archivo ha sido compartido correctamente. Si el archivo original no se encuentra, se devuelve un error 404 indicando que el archivo no fue encontrado. Cualquier otro error durante el proceso de búsqueda o guardado es capturado y resulta en un error 500, indicando un problema interno del servidor.

Esta funcionalidad es crucial para permitir una colaboración efectiva entre usuarios, asegurando que los archivos puedan ser compartidos de manera segura y controlada, respetando los permisos y la propiedad definidos por los usuarios.

```
const ArchivoCompartido = async (req, res) => {
  const {ArchivoCompartido, correoCompartido, permisos, token, correo} = req.body;
  const tokenValido = await Token.validartoken(token, correo);
  if (tokenValido) {
    try {
      const archivoExistente = await Archivo.archive_MongooseModel.findOne({ '_id: ArchivoCompartido'});
      const nombreArchivo = archivoExistente.nombreArchivo;
      const ipfs_hash = archivoExistente.ipfs_hash;
      const tamanoArchivo = archivoExistente.tamano;
      if (!archivoExistente) {
        return res.status(404).send('Archivo no encontrado');
      }
      const nuevoArchivo = new compartido.compartido_MongooseModel({
        _id: new mongoose.Types.ObjectId(),
        ArchivoCompartido: nombreArchivo,
        tamanoArchivo: tamanoArchivo,
        ipfs_hash: ipfs_hash,
        correoCompartido,
        correoOwner: correo,
        fechacompartido: dayjs(Date.now()).format('YYYY-MM-DD HH:mm:ss'),
        permisos: permisos,
        archivoOriginal: archivoExistente._id,
      });
      await nuevoArchivo.save();
      res.status(201).send('Archivo compartido correctamente');
    } catch (error) {
      console.error('Error al compartir archivo:', error);
      res.status(500).send('Error interno del servidor: ' + error.message);
    }
  }
};
```

- modificararchivocompartido

La función `modificararchivocompartido` en Node.js está diseñada para permitir la modificación de los metadatos de un archivo compartido específico dentro del sistema, siempre y cuando el usuario tenga los permisos adecuados. Esta operación comienza con la validación del token de

autenticación para verificar la identidad y los permisos del usuario que realiza la solicitud. Utiliza `Token.validartoken` para confirmar que el token y el correo asociado son válidos.

Una vez verificado el token, la función procede a buscar en la base de datos MongoDB la existencia del usuario que intenta realizar la modificación, y si no lo encuentra, devuelve un error indicando que el usuario no fue encontrado. Si el usuario existe, el siguiente paso es localizar el archivo compartido específico utilizando el identificador `'_id'` proporcionado en la solicitud.

Tras encontrar el archivo compartido, la función verifica si el usuario tiene permisos de escritura para modificar el archivo. Si se confirma que tiene permisos (`permisos: 'escritura'`), procede a actualizar el nombre del archivo compartido y la fecha de la última modificación. Esta actualización se realiza utilizando la librería `dayjs` para formatear la fecha actual.

Si el usuario no tiene permisos para editar el archivo, se devuelve un mensaje de error 403, indicando la falta de permisos. Si todo es correcto y se logra modificar el archivo compartido, se confirma la operación enviando una respuesta positiva al cliente. En caso de cualquier error durante este proceso, como un fallo en la búsqueda o en la actualización de datos, se captura y registra el error, devolviendo un mensaje de error 500 indicando un problema interno del servidor. Si el token no es válido, se devuelve un error 401, señalando la invalidez del token proporcionado.

```
const modificararchivocompanido = async (req, res) => {
  const { _id, nuevonombreArchivo, token, correo } = req.body;
  const tokenValido = await token.validartoken(token, correo);
  if (tokenValido) {
    const usuarioExistente = await usuario.usuario_MongooseModel.findOne({ correo });
    if (!usuarioExistente) {
      return res.status(404).send('Usuario no encontrado');
    }
    try {
      const archivoExistente = await companido.compartido_MongooseModel.findOne({ _id });
      if (!archivoExistente) {
        return res.status(404).send('Archivo no encontrado');
      }
      // Verificar si el usuario tiene permisos para editar el archivo compartido
      const permisosUsuario = await companido.compartido_MongooseModel.findOne({
        _id: archivoExistente._id, // Asegurarse de que se consulta por el ID del archivo
        permisos: 'escritura'
      });
      console.log(permisosUsuario);
      if (!permisosUsuario) {
        return res.status(403).send('El usuario no tiene permisos para editar este archivo compartido');
      }
      permisosUsuario.fechacompanido = dayjs(Date.now()).format('YYYY-MM-DD HH:mm:ss'),
      permisosUsuario.ArchivoCompartido = nuevonombreArchivo
      await permisosUsuario.save();
      return res.send('Archivo modificado correctamente');
    } catch (error) {
      console.error('Error al modificar archivo:', error);
      res.status(500).send('Error interno del servidor: ' + error.message);
    }
  } else {
    return res.status(401).send('Token no válido');
  }
};
```

- eliminarArchivoCompartido

```
const eliminarArchivoCompartido = async (req, res) => {
  const { _id, token, correo } = req.body;
  const tokenValido = await Token.validartoken(token, correo);
  if (tokenValido) {
    const usuarioExiste = await usuario.usuario_MongooseModel.findOne({ correo });
    if (!usuarioExiste) {
      return res.status(404).send('Usuario no encontrado');
    }
  }
  try {
    const archivoCompartido = await compartido.compartido_MongooseModel.findOne({ _id });
    if (!archivoCompartido) {
      return res.status(404).send('Archivo compartido no encontrado');
    }
    // Verificar si el usuario tiene permisos para eliminar el archivo compartido
    if (archivoCompartido.correoCompartido !== correo) {
      return res.status(403).send('El usuario no tiene permisos para eliminar este archivo compartido');
    }
    // Eliminar el archivo compartido
    await compartido.compartido_MongooseModel.findByIdAndDelete(_id);
    return res.send('Archivo compartido eliminado correctamente');
  } catch (error) {
    console.error('Error al eliminar archivo compartido:', error);
    res.status(500).send('Error interno del servidor: ' + error.message);
  }
};
```

La función `eliminarArchivoCompartido` en Node.js gestiona la eliminación de un archivo compartido específico dentro del sistema. Inicia validando el token de autenticación con la función `Token.validartoken` para asegurarse de que el usuario realizando la solicitud tiene permisos para actuar. Esta validación es crucial para mantener la seguridad y la integridad del sistema.

Una vez confirmada la validez del token, la función busca al usuario asociado a dicho correo en la base de datos MongoDB. Si el usuario no existe, se devuelve un error 404 indicando que el usuario no fue encontrado, deteniendo el proceso.

Si el usuario existe, procede a buscar el archivo compartido específico utilizando el identificador `_id` proporcionado en la solicitud. Si no encuentra el archivo compartido, devuelve un error 404 señalando que el archivo compartido no fue encontrado.

Antes de proceder con la eliminación, verifica que el correo del usuario que intenta eliminar el archivo coincida con el `correoCompartido` del registro del archivo. Esto asegura que sólo el usuario que compartió el archivo inicialmente, o un usuario con permisos adecuados, pueda eliminarlo. Si el usuario no tiene los permisos necesarios, se devuelve un error 403 indicando falta de permisos.

Si todas las verificaciones son correctas, elimina el archivo compartido utilizando el método `findByIdAndDelete` y confirma la operación enviando una respuesta que indica que el archivo compartido ha sido eliminado correctamente.

Cualquier error capturado durante el proceso, ya sea en la búsqueda o en la eliminación, se registra y se comunica al usuario mediante un error 500, señalando un problema interno del servidor. Esto asegura que el usuario reciba feedback adecuado en caso de fallos en la operación.

- **obtenerArchivosCompartidos**
La función `obtenerArchivosCompartidos` en Node.js está diseñada para recuperar todos los archivos que han sido compartidos con un usuario específico, verificando primero la autenticidad y los permisos del usuario mediante un token de autenticación. Al confirmar la validez del token, la función busca en la base de datos MongoDB todos los registros de archivos que han sido compartidos con el correo electrónico proporcionado. Si no encuentra archivos compartidos o la lista está vacía, devuelve un error 404 indicando que no se encontraron archivos compartidos. En caso contrario, procesa cada registro para eliminar campos innecesarios como `__v` y `archivoOriginal` para simplificar la información retornada, luego envía los datos filtrados al usuario en formato JSON. Si ocurre algún error durante la consulta a la base de datos, capture este error y devuelve un error 500, indicando un problema interno del servidor. Si el token no es válido, notifica al usuario con un error 401. Esta funcionalidad es esencial para permitir a los usuarios acceder y gestionar archivos que otros usuarios han compartido específicamente con ellos, manteniendo la organización y la seguridad de la información compartida.

```
const obtenerArchivosCompartidos = async (req, res) => {
  const { correo, token } = req.body;
  const tokenValido = await Token.validartoken(token, correo);
  if (tokenValido) {
    try {
      const archivosCompartidos = await compartido.compartido_MongooseModel.find({ correoCompartido: correo });
      if (!archivosCompartidos || archivosCompartidos.length === 0) {
        return res.status(404).send('No se encontraron archivos compartidos contigo');
      }
      const archivossinCampos = archivosCompartidos.map(archivo => {
        const { __v, archivoOriginal, ...archivossinCampos } = archivo.toObject();
        return archivossinCampos;
      });
      return res.json({ archivosCompartidos: archivossinCampos });
    } catch (error) {
      console.error('Error al obtener archivos compartidos:', error);
      res.status(500).send('Error interno del servidor: ' + error.message);
    }
  } else {
    return res.status(401).send('Token no válido');
  }
};
```

- **obtener**
La función `obtener` en Node.js está diseñada para facilitar el acceso a un archivo específico compartido mediante IPFS, verificando primero la autenticidad del usuario a través de un token. Al validar el token, procede a buscar en la base de datos MongoDB el archivo compartido utilizando el identificador `__id` proporcionado. Si encuentra el archivo,

genera y devuelve un enlace para descargar el archivo directamente desde el servidor IPFS, utilizando el hash IPFS almacenado y el nombre del archivo para crear una URL personalizada que incluye opciones para la descarga directa.

Esta función es especialmente útil para proporcionar un acceso seguro y controlado a archivos compartidos, asegurando que solo los usuarios autorizados puedan descargar los archivos. Si el archivo específico no se encuentra, no se devuelve ningún enlace y se informa al usuario con un error 404, indicando que el archivo compartido no fue encontrado. En caso de que el token proporcionado no sea válido, se rechaza la solicitud con un error 401. Si ocurre un error durante la búsqueda en la base de datos o al generar la URL, se registra este error y se devuelve un error 500, señalando un fallo interno del servidor.

```
const obtener = async (req, res) => {
    const { id, correo, token } = req.body;
    const tokenValido = await Token.validarToken(token, correo);
    if (tokenValido) {
        try {
            const archivoCompartido = await Compartido.Compartido_MongooseModel.findById(id);
            if (!archivoCompartido) {
                return res.status(404).send(JSON.stringify({ message: 'http://10.4.41.54:8080/ipfs/' + archivoCompartido.ipfs_hash + '?download=true&filename=' + archivoCompartido.ArchivoCompartido.ArchivoCompartido }));
            }
        } catch (error) {
            console.error('Error al obtener archivo compartido por ID:', error);
            res.status(500).send("Error interno del servidor: " + error.message);
        }
    } else {
        return res.status(401).send("Token no válido");
    }
};
```

7.3.3. Autenticación y Gestión de Tokens

```
async function crearToken(usuario) {
    const clave = usuario.clave;
    const correo = usuario.correo;
    try {
        const usuarioExistente = await Usuario.Usuario_MongooseModel.findOne({ correo });
        if (!usuarioExistente) {
            throw new Error('El usuario no existe');
        }
        const token = jwt.sign({ correo, clave }, key, { expiresIn: '24h' });
        return token;
    } catch (error) {
        console.error('Error al crear token:', error);
        throw new Error('Error interno del servidor: ' + error.message);
    }
}
```

Se implementó un sistema de autenticación basado en tokens utilizando JSON Web Tokens (JWT). Este sistema genera tokens al iniciar sesión que tienen una duración de 1 hora, proporcionando así un método seguro y temporal de autenticación. Los tokens se generan mediante una clave secreta y almacenan información del usuario como el correo y la clave, lo que permite validar las sesiones de usuario de manera eficiente sin necesidad de enviar constantemente las credenciales. Para asegurar que sólo los usuarios autorizados puedan acceder y manipular los datos, el controlador

```
async function validartoken(token, correo) {
    try {
        const usuarioExistente = await Usuario.usuario_MongooseModel.findOne({ correo });
        if (!usuarioExistente) {
            throw new Error('El usuario no existe');
        }
        const tokenValido = jwt.verify(token, key);
        if (tokenValido.correo == correo){
            return tokenValido;
        }
        else {
            throw new Error('El token no corresponde al usuario');
        }
    } catch (error) {
        console.error('Error al validar token:', error);
        throw new Error('Error interno del servidor: ' + error.message);
    }
}
module.exports = { crearToken, validartoken };
```

Token.controler.js gestiona la creación y validación de estos tokens, que son necesarios para realizar operaciones sensibles como modificar o eliminar archivos. Los tokens aseguran que las operaciones sobre los archivos compartidos sólo puedan ser realizadas por usuarios con los permisos adecuados, y son validados en cada solicitud pertinente a través de middleware antes de permitir el acceso a la funcionalidad correspondiente.

7.3.4. Integración con IPFS

Una de las características clave de nuestro sistema es la integración con IPFS para el almacenamiento distribuido de archivos. Esto permite que los archivos no residan en un solo punto, sino que estén distribuidos a través de una red, mejorando así la redundancia y la disponibilidad. Cada archivo almacenado en nuestro sistema tiene un hash IPFS correspondiente almacenado en la base de datos, lo que facilita su recuperación y compartición segura a través de la red IPFS.

7.3.5. Interacciones del Usuario

Desde el punto de vista del usuario, el proceso en nuestro sistema de almacenamiento y compartición de archivos es intuitivo y seguro, comenzando con la creación de una cuenta. Durante este proceso inicial, se solicitan datos esenciales como el nombre, correo electrónico y contraseña. Este registro se gestiona a través de nuestro backend, donde los detalles son almacenados de forma segura en nuestra base de datos MongoDB.

Una vez registrados, los usuarios pueden iniciar sesión mediante nuestras interfaces diseñadas para la autenticación. Al iniciar sesión correctamente, el sistema genera un token de seguridad utilizando JSON Web Tokens (JWT). Este token no solo autentica al usuario en cada sesión, sino que también garantiza la seguridad de las transacciones al requerirse para cada operación significativa dentro del sistema.

Con el token, los usuarios están habilitados para realizar diversas operaciones relacionadas con sus archivos:

- Subir nuevos archivos: Los usuarios pueden cargar archivos al sistema, los cuales son luego almacenados en IPFS, asegurando su disponibilidad y redundancia en la red.
- Modificar datos de archivos existentes: Los usuarios pueden cambiar detalles como el nombre del archivo o actualizar los archivos mismos, siempre que posean los permisos adecuados, verificados por el token de seguridad.
- Eliminar archivos: Esta operación también requiere verificación de permisos a través del token, asegurando que solo los usuarios autorizados puedan eliminar archivos.
- Compartir archivos: Los usuarios pueden compartir archivos con otros usuarios dentro del sistema, estableciendo permisos de acceso como 'lectura' o 'escritura'. Cabe destacar que el permiso de escritura solo permite cambiar el nombre del archivo, mas no el contenido. Esta funcionalidad es manejada a través del modelo `compartido_Mongoose`, que registra cada acción de compartición junto con los permisos y el usuario receptor.

El token tiene una duración limitada, promoviendo así sesiones seguras y minimizando el riesgo de accesos no autorizados en caso de que el token sea comprometido. Si el token expira durante una sesión, el usuario debe volver a iniciar sesión para obtener un nuevo token y continuar operando dentro del sistema.

Este diseño no solo facilita la experiencia del usuario al mantener la interfaz simple y directa, sino que también subraya nuestro compromiso con la seguridad y la privacidad de los datos del usuario, garantizando que todas las interacciones dentro del sistema sean seguras y verificadas.

7.3.6. Seguridad y Manejo de Errores

Durante toda la implementación, se prestó especial atención a la seguridad y al manejo de errores. Las interacciones con la base de datos están protegidas contra inyecciones de SQL y otros tipos de ataques. Además, se implementaron manejadores de errores robustos que capturan y registran fallos en las operaciones de la base de datos o en la validación de datos, mejorando así la estabilidad y la seguridad del sistema.

7.4. IPFS

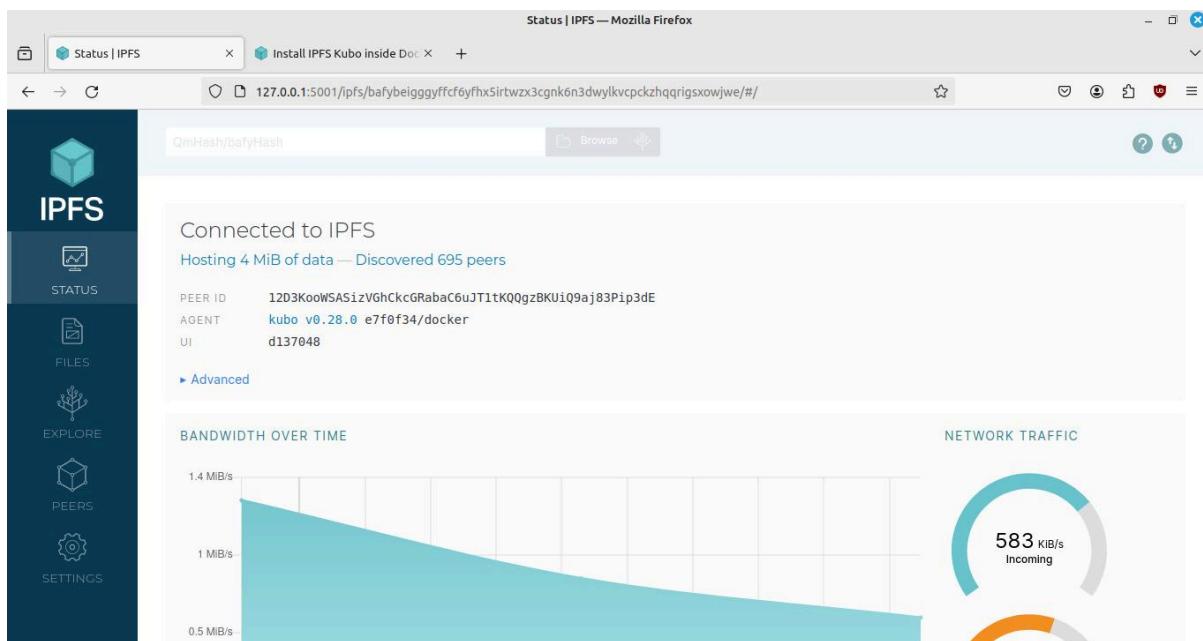
Con IPFS realmente la implementación se debía separar en dos ámbitos: El ámbito de configuración de la imagen a desplegar, y luego la interconexión entre las imágenes (primeramente en la misma máquina y luego entre máquinas).

Como objetivos del desarrollo, pusimos 4 targets:

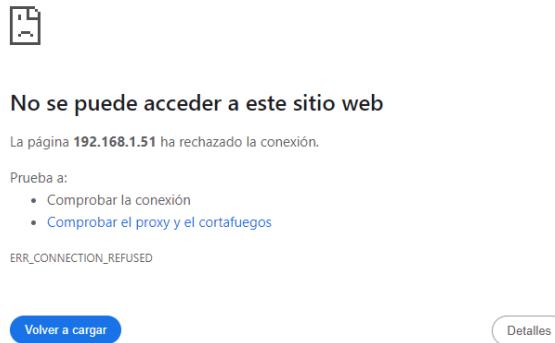
1. IPFS funciona en local .
2. IPFS funciona en la máquina virtual.
3. IPFS funciona de forma privada.
4. IPFS funciona de forma privada y con diversos nodos.

Para llegar a la primera meta y para empezar a trastear con IPFS, empezamos descargando e instalando la versión de IPFS Kubo alojada en docker, y para descargarla tendremos que previamente descargar docker.

Si ejecutamos este contenedor sin previamente configurar, veremos esto desde el mismo equipo:



Por lo que podemos ver, IPFS se ha conectado a peers de la red global, pero vemos que se ha conectado a peers que no nos interesan. Además, si intentamos acceder desde otro equipo al nodo de IPFS, veremos:



Esto nos puede indicar que para acceder al nodo tendremos que configurar el firewall del equipo para que se pueda acceder desde fuera del equipo.

Para ello, hemos modificado el firewall por defecto de Ubuntu para que deje pasar todos los puertos de IPFS. Con ello, desde otros equipos podemos acceder al nodo de IPFS. Tendremos que aceptar conexiones desde el puerto 4001(IPFS swarm), 5001(IPFS API), 8080(IPFS Gateway), 9094(IPFS Cluster API), 9095(IPFS proxy), 9096 (cluster swarm).

Luego de probar en local, se desplegó IPFS en la máquina virtual. Para ello, se desplegó una prueba, creando un par de directorios para el almacenamiento de los datos. Cuando se encendió el nodo, previamente configurado el firewall, funcionaba sin problemas.

Después de haber implementado el primer nodo en la máquina virtual de la FIB, y ver que la configuración inicial funciona, el tercer objetivo se ponía a la vista: hacer que dos nodos en la misma máquina se comuniquen exclusivamente entre ellos, sin que llegue a la red pública.

Uno de los principales problemas que ha habido en todo el apartado de IPFS ha sido la mala documentación de muchos apartados de IPFS, ya que hay guías, pero a la mínima que haya un cambio en cualquier apartado (ya sea por alguna depreciación de alguna característica o por incompatibilidad con tu sistema) la posibilidad de encontrar algo útil fuera del prueba y error es baja.

Por ejemplo, si queremos hacer que los nodos de IPFS se comuniquen entre ellos con su clave privada, no podemos dársela de forma de las swarm keys, ni cómo environment variable ni como archivo, ya que cuando se hace eso se aplica un protocolo de compartición de claves desactualizado, no dejando que los nodos se hagan privados.

Para setear la privacidad de los nodos, lo que se hace esta vez es a través de IPFS-Cluster, que será el que se encargue de hacer que los nodos sean privados. Para ello, se ha creado un script (secret.sh) que te crea una clave, pero se tiene que insertar en las configuraciones manualmente, pero todavía no tenemos la capacidad de acceder al sistema de archivos del contenedor, ya que es temporal.

Además, queremos hacer que los contenedores puedan tener dependencias con otros contenedores.

Para solucionar ambos problemas, usamos Docker Compose. Docker Compose nos permite crear diversos nodos de forma muy simple, sin que los contenedores sean dependientes de otros contenedores (queremos que los contenedores de IPFS-Cluster sean dependientes de los de IPFS). Para ello, instalamos docker compose, y descargamos una configuración inicial por defecto. Modificando unos pocos atributos hacemos que los contenedores accedan al sistema de archivos local, permitiendo que se puedan mantener ciertas configuraciones entre inicios de los contenedores (aunque no debería caer los nodos IPFS en funcionamiento) y que los archivos se mantengan cuando esté el sistema apagado.

También modificamos la configuración para que solo se invoquen 2 nodos (2 de IPFS y 2 de IPFS-cluster) en vez de 3 y para que de los dos pares en uno tanto IPFS y IPFS-cluster expongan los puertos a la red.

```
ipfs0:  
  container_name: ipfs0  
  image: ipfs/kubo:latest  
  ports:  
    - "4001:4001" # ipfs swarm  
    - "5001:5001" # ipfs api  
    - "8080:8080" # ipfs gateway
```

Con esto en mente, podemos modificar la clave privada, yendo al archivo /compose/clusterx/service.json, modificando la clave peername de cluster.

```
"cluster": {  
  "peername": "clusteri1",  
  "secret": "3a3bb244a1eefdef63c9e19d68d97bd7a45fab6bf13b154254142b3e12a15b9f",  
  "leave_on_shutdown": false,  
  "listen_multiaddress": [  
    "/ip4/0.0.0.0/tcp/9096",  
    "/ip4/0.0.0.0/udp/9096/quic"  
  ],
```

Al modificar esta clave lo que estamos consiguiendo es que solo se comunican los nodos entre sí si están en el mismo cluster.

Ahora hay un problema, y es que los nodos de IPFS no están conectados de forma correcta, ya que no podemos hacer peticiones de forma correcta. Para ello, tenemos que activar las Cross-Origin Resource Sharing (CORS), ya que no nos deja hacer peticiones que sean respondidas de otro origen.

Para activarlo, lo que tenemos que hacer es ejecutar en el contenedor de IPFS un par de comandos, que modifican el archivo de configuración. A diferencia de la configuración de IPFS-Cluster, IPFS nos permite configurar a través de la línea de comandos. Para ello, lo que hacemos es crear un script, initIPFS.sh, que será un script que se ejecutará dentro del contenedor en la inicialización, que se encargará de aceptar las CORS de los demás nodos (con la ip de los equipos donde se ejecutan los nodos), y que sean del tipo POST y PUT.

```
ipfs config --json API.HTTPHeaders.Access-Control-Allow-Origin '["http://0.0.0.0:5001","http://10.4.41.54:5001","http://10.4.41.39:5001", "http://10.4.41.38:5001"]'  
ipfs config --json API.HTTPHeaders.Access-Control-Allow-Methods ['[PUT", "POST"]'
```

Para que el script se ejecute cada vez que se corre el contenedor, hacemos que se monte el script en una carpeta del contenedor, /container-init.d, que hace que se ejecuten los scripts en la inicialización.

Para montar el volumen, lo hacemos desde el docker-compose.yml modificado, en el apartado volumes:

```
volumes:  
  - ./compose/ipfs0:/data/ipfs  
  - ./initipfs.sh:/container-init.d/initipfs.sh  
  - ./swarm.key:/swarm.key
```

Con esto, se monta el archivo en la ubicación en el contenedor, y también lo hacemos para la parte de persistencia de los archivos.

Con esto, creamos dos scripts

El primer script (initClean.sh) es para borrar todos los archivos de ipfs, y matar todos los contenedores de IPFS, permitiendo reiniciar todo de un comando.

El otro script es para ejecutar ipfs, init.sh, que realmente lo que hace es pasar el hostname a los contenedores, que es algo que pensaba que se iba a usar pero al final no hizo falta.

Con todo esto ya hemos creado el cluster privado, ahora falta que el cluster pueda tener nodos en diversos equipos.

Para ello, he tenido que configurar y desactivar el peering inicial de los nodos, ya que al ser la red privada generaba conflictos el swarming de los nodos con lo privado, y se hace el peering de forma semiautomática.

Para ello, se ha modificado el script initipfs.sh para que, cada vez que se inicialice el nodo, se borren los peers guardados (por tema de borrar solo los peers antiguos si no se configuraba bien), y luego se añaden los peers a mano. Se estuvo mirando de hacer de forma que el archivo se obtenga de

una fuente centralizada, pero se vio que no hacia falta, ya que si se añade una parte del camino (del nodo 1 al 2 y del 2 al 3) el sistema hace peering de forma automática, llegando al consenso deseado.

```
1 set -ex
2
3 #Borramos los peers y los iniciamos manualmente
4 ipfs bootstrap rm all
5 ipfs bootstrap add "/ip4/10.4.41.54/tcp/4001/ipfs/12D3KooWSFEj6rx1cjhg2m4idJcgxS2DbByQJzQThCmhu73EM6Q"
6 ipfs bootstrap add "/ip4/10.4.41.54/tcp/4001/ipfs/12D3KooWKbJSLJSbMqBDuyTjSTnDhvZ8C5Zh4ETw2B18ac5HNqD5"
7 ipfs bootstrap add "/ip4/10.4.41.39/tcp/4001/ipfs/12D3KooWBDQwHudbJNMrh7wtsFqGqEx6yhAb9MEJeRp86KyVy7fP"
8 ipfs bootstrap add "/ip4/10.4.41.39/tcp/4001/ipfs/12D3KooWCCqwkz894skYVfkmF3tzHH55fuXXMfvb5GbxaLQLY6vt"
9
10
11 #Acceptamos la comunicación de los nodos
12 ipfs config --json API.HTTPHeaders.Access-Control-Allow-Origin '[["http://0.0.0.0:5001", "http://10.4.41.54:4001"]]' 
13 ipfs config --json API.HTTPHeaders.Access-Control-Allow-Methods ['["PUT", "POST"]']
14
15
```

Cuando ya hemos llegado a los 4 objetivos, falta ver qué operaciones tenemos que hacer para interactuar con UPCloud.

Pero antes de ver las operaciones tenemos que entender cómo está configurado el sistema y cómo almacena los archivos.

Por defecto, cuando subimos un archivo y se rastrea por IPFS-Cluster, IPFS decide en el momento donde almacenarlos. Si hay almacenamiento sobrante, lo almacena en todos los nodos, los marca como almacenados (pin), y cuando vaya faltando espacio en el nodo, va eliminando archivos (siempre evitando borrarlo del todo). Además, IPFS-Cluster no borra los archivos a excepción de que tenga que reescribirlos, lo que hace es desmarcarlos como almacenados (unpin), y cuando el recolector de basura (gc) recorra los archivos, si vé que está unpin, lo borra. El gc solo corre si se activa de forma manual o si falta memoria.

Además, si haces la petición a un nodo y no tiene el archivo:

1. El nodo le pregunta al resto de los nodos que nodo tiene el archivo, y devuelve el nodo el resultado.
2. El nodo guarda en caché el resultado, haciendo que la próxima petición la pueda responder directamente.

Las operaciones que se vio que se tendrían que hacer son:

1. Carga de archivos: La carga se hace a través del backend, así que se tendrá que recibir el archivo desde el backend (al tener que almacenar las propiedades del archivo). Esta carga se hace a través de IPFS proxy, en el IPFS-Cluster, ya que los clusters tienen que tener constancia de los archivos.
2. Descarga del archivo: La descarga se hace a través del frontEnd, previa consulta al backend para obtener los parámetros del archivo (IPFS puede ignorarlos). Esta descarga se hace a través de IPFS gateway, ya que no tiene que hacer ninguna actualización de los pins del cluster, y el gateway da buena implementación con http.

3. Eliminación del archivo: Cuando se elimina definitivamente un archivo, realmente lo que se le dice a IPFS-Cluster no es que lo elimine, sino que quite todos los pins de todos los nodos. La petición se hace desde el Backend. Por ello, hacemos la petición a IPFS proxy.

Con esto en mente, tenemos planteado hacer las peticiones solo a un nodo, aunque es algo que podría ser cambiado insertando un load balancer.

Con todo configurado, tenemos ya la red IPFS privada implementada, y con capacidad de comunicación entre equipos.

Posteriormente se ha creado una guía rápida para la implementación en otro equipo.

7.5. Kubernetes

Kubernetes es una plataforma portable y extensible de código abierto para administrar cargas de trabajo y servicios. Kubernetes facilita la automatización y la configuración declarativa, por lo tanto facilita la gestión de los contenedores que podemos llegar a tener.

Kubernetes fue elegido en este proyecto para poder dar soporte a IPFS, elegimos precisamente Kubernetes porque ya habíamos encontrado ejemplos de como hacer la integración de estos de este con IPFS. A continuación veremos en más detalle como hicimos uso de Kubernetes para este proyecto.

Instalación de Kubernetes

El primer paso era hacer la instalación de Kubernetes, o K8s, y todas sus dependencias para que podamos crear el cluster que nosotros queramos.

Pero antes de instalar Kubernetes instalamos docker ya que es la herramienta que usaremos para crear los contenedores. Hubo un pequeño problema, al instalar docker se instalan también sus dependencias y entre ellas está containerd, el cual tenía desactivado por defecto que pudiera actuar como un container runtime que es lo que necesitamos para Kubernetes. Por lo que tuvimos que modificar el fichero de configuración “/etc/container/config.toml”.

Para poder usar Kubernetes durante nuestro proyecto hicimos uso de kubeadm, esta es una herramienta muy buena para probar Kubernetes, si eres nuevo en esto, y automatizar el hecho de levantar un cluster en el que tu puedas probar ahí la aplicación. Para su instalación seguimos la documentación proporcionada por Kubernetes mismo.

Para poder usar kubeadm necesitaremos también instalar otras herramientas como docker, junto a las dependencias de este, kubelet que correrá en todos los nodos y es el encargado de inicializar los pods y contenedores, y kubectl que será la herramienta a través de la cual “hablaremos” con nuestro cluster.

El siguiente paso fue instalar un add-on de red para que los nodos de nuestro cluster puedan conectarse al nodo que actúa como Master, hay muchos add-ons de red distintos (se pueden encontrar en:

<https://kubernetes.io/docs/concepts/cluster-administration/addons/>) pero nosotros elegimos Flannel, el cual nos dio ciertos problemas durante la implementación, para hacerlo si vas a su página de github, a través del link anterior, te dan indicaciones de como hacerlo que es ejecutando la comanda: `kubectl apply -f https://github.com/flannel-io/flannel/releases/latest/download/kube-flannel.yml`, una vez ya hemos inicializado el master, que lo veremos a continuación.

Una vez ya está todo lo necesario instalado procedemos a crear el cluster. El primer paso es inicializar el Master con la comanda `kubeadm init`, si queremos especificar alguna opción extra, como la configuración, también se puede, en nuestro caso usamos una configuración muy simple para especificar que el cgroupDriver que usaríamos era el systemd, para hacer eso creamos un fichero .yaml llamado kube-config y ejecutamos la comanda:

`kubeadm init --config kube-config.yaml`

Si esto ha funcionado sin problemas saldrá el mensaje: "Your Kubernetes control-plane has initialized successfully!". También se imprimía en la terminal la comanda a ejecutar para unir los nodos workers al cluster:

`kubeadm join <control-plane-host>:<control-plane-port> --token <token> --discovery-token-ca-cert-hash sha256:<hash>`

En el caso que pasará el tiempo de expiración del token necesario para unirse, que por defecto eran veinticuatro horas, también se podía ejecutar la siguiente comanda en el nodo maestro:

`kubeadm token create --print-join-command`

Esta crearía un token e imprimiría la comanda a ejecutar en el nodo que quieras unir al cluster.

Una vez realizado todo lo mencionado anteriormente ya tenemos nuestro cluster de Kubernetes creado y podemos proceder a crear lo necesario para la integración con IPFS. En nuestro cluster tendremos un nodo maestro, que se encargará de la gestión, y tres workers, que se encargarán de la ejecución de los pods.

Integración con IPFS

Una vez ya tenemos nuestra infraestructura creada que en nuestro caso consiste en un nodo Master, el cual no ejecutará ningún pod, y tres nodos workers que se encargarán de la ejecución de los pods y de tener los persistent volumes, de los cuales hablaremos más adelante.

Investigando por Internet y buscando información encontramos una guía hecha por los mismos desarrolladores de IPFS que explicaba cómo juntar estas dos tecnologías. El problema con esta guía era que si la seguías paso a paso no funcionaba correctamente, por lo que tuvimos que encontrar qué era lo que fallaba exactamente y modificarlo, otra página web estaba mejor hecha y nos ayudó a resolver la mayoría de estos problemas (la página web es: <https://medium.com/rahasak/deploy-ipfs-cluster-with-kubernetes-c4cd8d64b7c8>). Aún así no nos ayudó para todos por lo que tuvimos que seguir investigando hasta dar con la tecla y que funcionará sin problemas

Para poder integrar IPFS con Kubernetes necesitamos crear cuatro ficheros .yaml: un statefulset, un service, un secret y un configmap. Ahora explicaremos en más detalle qué hace cada uno de estos ficheros.

- Secret: es el encargado de contener los tokens el identificador del peer, la clave privada de este y el cluster secret. El peer id es para cada nodo es decir el resultado de lo que veremos a continuación es el identificador para el primer nodo, los demás generan su propio peer id. Para crear estos tokens ejecutaremos los siguientes comandos:

```
Para obtener el cluster secret: $ od -vN 32 -An -tx1 /dev/urandom | tr -d '\n' | base64 -w 0 -
```

Para el peer id: \$ ipfs-key | base64 -w 0 al ejecutar esta comanda nos sale por terminal algo como:

```
Generating a 2048 bit RSA key...
```

```
Success!
```

```
ID for generated key: QmSq5fw3Wpff3teYngdKbLT6xx89KMAp2Da8FLvQZkPmvZ  
CAASpwkwggSjAgEAAoIBAQC6F+iIdatW8GhL7JSbdVrJOpsNlaj8CXurzxSRJozOGIMcUu  
WdxQypLBhwzS/Fd46p4zQel/KAMg3+lWlTIa9EQp0+XsNB7rbLBO9upzg2pYHNwiRj7wJ9  
iIY6tfoF0g+BsqQIHld6Ur+W9MKv7VmZCFwE4sD/4MSBr+3ptisN82wTO7mebsi7L78CU  
6IJQ+IpcwRhEW4Et91bdrnPQoL2+D2GIyNvqdkxL36ga+BtokOs/EL6NzJtHSClsYNUH1  
1HYPX7XDKVm6TZMMC31u9K6SgFN9AYqHQLI0H6ZEfxtYh20/9qrro4DukaInZEUGANpnw  
k11kgaXFj0FOC5AgMBAAECggEAdi764bcBMryJMDa2hig4mPWcRTtXz4DiRtHDuQ8nezf8  
BD/tTY40BG/ZscHN5fWw59nAXvGW861WT+1ps38ABTTdEiAylvvYcQTYXojaXXItBpSwcd  
T32uuae6zYdVvbzt3RiMpkOe5VeLRG9F2jhWAEEF6I+egKwOgGrabLkkkjBBBy9wzxuyjN  
rgwhm/ZjBNcQrGLgN9BKQRSi40Szq6kQkibJq//VykJ7CKISQwEdnAISkLmKbSL34ZX7h8  
9ZOYcdOJC8wMotj9/pM9zIz+gvOfX03OdUAkPzZ4ouvyD7vPhyT39GxxUvOw2VM673pu  
SC0kzAfWgMs5dFoPBQKBgQD20p9ZJCYMPeyvqmGUwrGt4tAV9A9Kq/9f9Wpjvoo98IZ/r  
VWpxRskzUvRYQnt+fbtw2V6WzugccGFJBr6wbnx1om8kITZ+X8s31R3NfRfTt/9aI/9woJ  
+zuyAS3c0PcVE5p+xqoAotQcpe8EOuV7NCh3zJzgrouLgJmfAZaWrwKBgQDBAzwVziMJ0H
```

```
ke2CeTz7ErG3paTSn1NcpZme5Beaci+KjyFGdmPMvID1KVI48cC7dGuuDI sDp0DSxR+HNQ
vBNpbEGDWpxyvxvJj/T9B4eWlai6YDKrk57KYHXR2BBWA2bdyD0jiRb+XcvWmunvGKTTbI
bw8WsoZGBd16cSM1/alwKBgE/Wu4KrJb+JdcjQ1TWyFUTiLbwHa62NdFYM1sFbgUnlMp0T
/fZknz+rsmVzGA8T8OTqZnOahu6x/f3igwDVimti2cmFTYhXcFZ4e4LWa90wpCF1BCGquE
2YPawLo5ks4u+nYq81dfi0uWX26ss0hGyA56IwxoILE8pxM7/yVJKzAoGBAJt+vaNbhMLd
XxbIUWo1gqqgV5QZm8AyKn1B8QIQj09Fivsj4Qy0MgafVIFTiZuLTVX34BCvqPc1M543KOS
yruH/QJR9rmmqfKAXEojxPGp7oRxyPb4Lm1KB6AbFi/hYte2vla8v7H7khW5yzBRAIRJFF
936ZIBlUdIYyLPjuxqRVAoGAdU9z4LoB6xRzGWPjT6xrZ1HUH1GG7L7omP5iY8SafcT4v4
DsbNHHiFOgXwNrhU+n8XYFHpWVcy4PwhBXHtihDOHn4F5QR3cr1CauidBgEm5XWTIxAree
7P2CQZ6Q2jzJHeULz8velVTPKZKZ0CdL1+hmAAfZ+N/yMV6XJODqS8Y=
```

De esta salida nos interesa lo que está marcado en negrita, todo lo demás lo usaremos para crear la private key del bootstrap peer.

Y finalmente para el bootstrap private key usando lo generado con la comanda anterior lo copiaremos, a partir del identificador, y lo codificamos en base64:

```
$ echo "<INSERT_PRIV_KEY_VALUE_HERE>" | base64 -w 0 -
```

El resultado será nuestra clave privada y lo copiaremos en el fichero.

- StatefulSet: en este statefulset se declaran dos pods y él se encargará de correr estos pods. Cada pod estará compuesto de dos contenedores el ipfs-cluster y el de ipfs, como indican sus nombres uno representa el nodo IPFS y el otro el IPFS Cluster.

En este fichero añadimos una toleration para que el nodo maestro no pudiera ejecutar los pods, es decir que no se tuviera en cuenta en el scheduler, ya que al tener las máquinas tan cargadas queríamos que el control-plane se encargaría solo de la gestión del cluster. Para hacerlo añadimos un campo de tolerations dentro de spec:

```
spec:
  tolerations:
    - key: node-role.kubernetes.io/control-plane
      effect: NoSchedule
```

- ConfigMap: en este fichero tendremos dos scripts el entrypoint.sh, el cual se encarga de habilitar el bootstrap del cluster de ipfs-cluster, y el configure-ipfs.sh, que configura el daemon de ipfs con los valores de producción. Estos scripts se usarán en el StatefulSet.
- Service: este es el paso final y es el encargado de exponer los endpoints del IPFS Cluster a fuera del Pod, exponiéndolo como un LoadBalancer es decir que hará un balanceo de cargas entre estos pods.

Una vez tengamos estos ficheros configurados podemos ya aplicarlo, pero antes si nos fijamos en el statefulset demanda volúmenes para cada uno de los contenedores de los pods. Al no tener ningún proveedor de nube, tendremos que crear los persistent volumes necesarios manualmente y un storageclass para indicar que usamos el almacenamiento local. Para indicar que no tenemos proveedor añadimos la siguiente línea en el storageclass.yaml:

```
provisioner: kubernetes.io/no-provisioner
```

Ahora si que ya tenemos todo lo necesario para que pueda funcionar, es decir ya podemos aplicar los ficheros al cluster y ver el resultado. Nos movemos al directorio donde se encuentren todos los ficheros .yaml que necesitamos y ejecutamos: *kubectl apply -f*.

Si todo ha salido de forma correcta deberías ver todo corriendo y sin estados de error. Haríamos los gets correspondientes y obtendremos los resultados:

Miramos si el statefulset funciona correctamente, vemos que las dos réplicas de los pods creados por este están en ready, lo indica el 2/2.

```
alumne@ninetales:~/kube-ipfs$ kubectl get sts
NAME          READY   AGE
ipfs-cluster  2/2    23d
```

Podemos comprobar que efectivamente los pods están funcionando correctamente de la siguiente manera:

```
alumne@ninetales:~/kube-ipfs$ kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
ipfs-cluster-0 2/2    Running   39 (4h4m ago) 23d
ipfs-cluster-1 2/2    Running   89 (9h ago)   23d
```

Una vez tuvimos esto funcionando, nos dispusimos a intentar hacer que el clúster pueda escalar de forma automática. Al no tener proveedor esto sería complicado ya que no podíamos usar todos los nodos que quisiéramos y para escalar en pods tendríamos que crear los persistent volumes correspondientes de forma manual y con limitaciones por el espacio de nuestras máquinas. El cómo mejorar en este aspecto se explica con más detalle en el apartado de mejoras a futuro.

Aunque sin proveedores de nube es complicado, intentamos hacer el escalado, de manera muy limitada y reducida, para los pods para aliviar la carga de trabajo de estos si recibían muchas peticiones a la vez y llegaban a consumir mucha cpu o memoria, para hacerlo necesitamos usar el Horizontal Pod Autoscaler (HPA). Así que creamos este HPA pero para que este funcione necesitamos obtener las métricas del clúster, es en este punto donde no tuvimos éxito. Al intentar obtener las métricas surgían fallos, no se

podían obtener por culpa de que no se comunicaban correctamente, el error era causado por Flannel que estaba dando problemas y a la hora de intentar comunicar estas métricas no funcionaba como debía.

7.6. Blockchain

Actualmente, nuestra implementación se basa en un contrato inteligente denominado RegistroArchivos, desarrollado en Solidity y diseñado para ser desplegado en una red compatible con Ethereum. Este contrato ofrece la funcionalidad de registrar archivos mediante un hash de IPFS, así como de registrar y monitorear los accesos a estos archivos, permitiendo la consulta del contador de accesos correspondiente. La estructura fundamental del contrato incluye funciones como agregarArchivo, que facilita el registro de nuevos archivos, y registrarAcceso, que actualiza el contador cada vez que se accede a un archivo.

Para llevar a cabo el despliegue y la gestión del contrato, utilizamos Truffle, una herramienta de desarrollo ampliamente reconocida en el ecosistema de Ethereum. Truffle simplifica tanto el proceso de despliegue como la interacción con el contrato, ofreciendo un entorno robusto para el desarrollo y la prueba de contratos inteligentes. En particular, el archivo truffle-config.js contiene configuraciones necesarias para establecer conexiones con redes Ethereum locales y, potencialmente, con redes públicas mediante proveedores como Infura.

Uno de los problemas que se tienen hasta el momento es en la complejidad para migrar con éxito el contrato a un entorno de uso práctico. Esto puede deberse a muchos factores, tales como configuraciones de red incorrectas, problemas relacionados con el Gas fee o incluso errores inherentes al contrato mismo.

```

truffle-config.js          RegistroArchivos.sol
blockchain > contracts > RegistroArchivos.sol
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
3
4 contract RegistroArchivos {
5     struct Archivo {
6         uint256 id;
7         string hashIPFS;
8         uint contador;
9     }
10
11    Archivo[] public archivos;
12
13    // Mapping que asocia el hash de IPFS con el índice del archivo en el array 'archivos'.
14    mapping(string => uint256) hashToId;
15
16    // Función que toma el hash de un archivo en IPFS.
17    function agregarArchivo(string memory _hashIPFS) public {
18        // Añade un nuevo 'Archivo' al array 'archivos'.
19        // El id del archivo es la longitud actual del array (que también será su índice),
20        // el hash es el pasado a la función y el contador se inicializa a 0.
21        archivos.push(Archivo(archivos.length, _hashIPFS, 0));
22
23        // Actualiza el mapping 'hashToId' con el nuevo índice del archivo que se añade.
24        hashToId[_hashIPFS] = archivos.length - 1;
25    }
26
27    // Función que actualiza el contador de accesos de un archivo basado en su hash de IPFS.
28    function registrarAcceso(string memory _hashIPFS) public {
29        // Obtiene el índice del archivo del mapping 'hashToId' usando el hash.
30        uint256 id = hashToId[_hashIPFS];
31        archivos[id].contador++;
32    }
33
34    // Función que devuelve el contador de accesos de un archivo.
35    function obtenerContador(string memory _hashIPFS) public view returns (uint256) {
36        // Devuelve el valor del contador accediendo directamente al array 'archivos' con el índice obtenido del mapping 'hashToId'.
37        return archivos[hashToId[_hashIPFS]].contador;
38    }
39 }
```

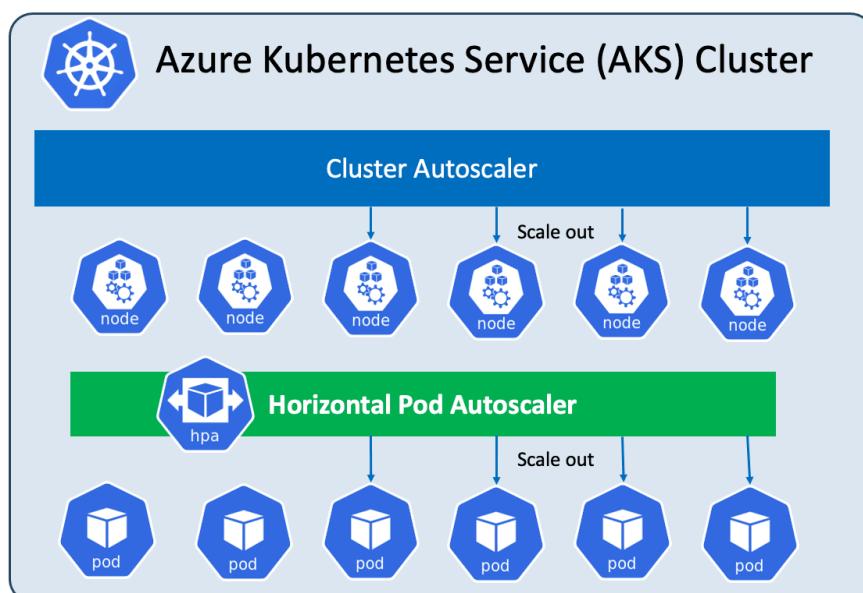
8. Mejoras a futuro

Si este proyecto se acabará llevando a cabo de una forma más profesional lo adecuado sería llevar a cabo diferentes mejoras que no hemos podido implementar durante el desarrollo.

8.1. Autoscaler

Una mejora importante, en cuanto a escalabilidad se refiere, podría ser el escalado automático que nos ofrece Kubernetes. De la forma en la que está desarrollado nuestro proyecto no disponemos de esta funcionalidad y puesto a que tener una facilidad para escalar es muy importante en un proyecto que se basa en crear un sistema de almacenamiento descentralizado, creemos que esta mejora en concreto aportaría muchos beneficios al proyecto.

La forma mediante la que implementaríamos esta mejora sería usando el Horizontal Pod Autoscaler (HPA) para poder escalar orientado a los pods y el Autoscaler que ese sí que sería para poder escalar los nodos de nuestro clúster.



El HPA sería el encargado de escalar de manera automática los pods que se han desplegado de los Deployments o StatefulSets que han sido aplicados al clúster. A este le indicaremos el objetivo a escalar, junto a un número máximo y mínimo de Pods que puede haber corriendo al mismo tiempo, y se establecerán unas condiciones como la memoria ocupada o la cpu usada, ya sea en forma de porcentaje o en números concretos, y si estos son superados el HPA creará un nuevo Pod con las mismas características que el Pod que tiene establecido como target ("objetivo").

Para saber cuántos recursos está utilizando el Pod se puede hacer de distintas maneras: a través de las mismas métricas del clúster que son muy simples y solo pueden ser métricas como cpu y memoria o si instalas un agente externo expresamente para controlar las métricas y lo que está pasando, los eventos, en el clúster como podría ser Prometheus, este además te permite crear métricas más complejas y personalizadas.

Con el HPA tendríamos solución a la escalabilidad dentro de un mismo nodo, ya que sirve para aliviar la carga a los pods muy solicitados. Entonces para poder ampliar el almacenamiento, que es realmente lo que nos interesa, necesitamos poder escalar los nodos y para hacerlo usaremos el Cluster Autoscaler. Este Cluster Autoscaler se puede integrar con el proveedor de nube que tengas, ese sería el mejor caso, o con la API de tu cluster de Kubernetes.

Lo que hará este autoscaler será comprobar si hay algún Pod en estado “Unscheduled” y si encuentra alguno añadirá más nodos al cluster, si esos nodos añadidos acaban con su tarea y están vacíos el autoscaler los “borrará” del cluster y volverá a estar nuevamente como al inicio.

8.2. Blockchain

A pesar de los problemas tenidos para la implementación de blockchain, lo realizado ayuda a poder identificar tanto las falencias que se tienen y áreas a mejorar. Ante esto, se requiere un desarrollo y la adaptación nuevos que nos permita superar los obstáculos iniciales. A continuación se señalan algunas posibles mejoras y enfoques destinados a reestructurar nuestra implementación de blockchain.

- **Optimización del Contrato:** Revisar y optimizar el contrato inteligente para asegurar que las transacciones sean eficientes en términos de Gas fee.
- **Integración con Interfaces de Usuario:** Realizar la conexión e integración con Frontend
- **Integración con base de datos** Realizar la conexión e integración con Blockchain
- **Mejorar la Seguridad:** Implementar medidas de seguridad adicionales para verificar la autenticidad de los archivos y los accesos registrados, posiblemente utilizando firmas digitales.
- **Escalabilidad y Redes de Prueba:** Realizar pruebas en redes de prueba de Ethereum como Rinkeby o Ropsten para asegurarse de que el contrato funciona como se espera antes de un despliegue en la red principal.
- **Automatización y Eventos:** Implementar eventos en el contrato que permitan a las aplicaciones clientes recibir actualizaciones en tiempo real cuando se registren o accedan a los archivos, facilitando la integración con otros sistemas y aplicaciones.
- **Despliegue y Gestión de Infraestructura:** Utilizar herramientas avanzadas para el despliegue y la gestión del contrato, como el Truffle Dashboard, que ofrece una mayor seguridad y control sobre las transacciones.

9. Conclusión

Este proyecto, que consta del desarrollo de un sistema de almacenamiento y gestión de archivos descentralizado, incorpora tecnologías avanzadas como NginX, React.js, Node.js, MongoDB, IPFS, Kubernetes y Blockchain. Representa un esfuerzo significativo hacia la innovación en el manejo de datos digitales. A través de la implementación de estas tecnologías, hemos logrado establecer una plataforma no solo escalable y expandible, sino también capaz de adaptarse a las crecientes necesidades del mercado y de la infraestructura tecnológica moderna.

Descentralización con IPFS

La integración de IPFS fue fundamental para descentralizar el almacenamiento de archivos, superando así las limitaciones de los sistemas centralizados tradicionales. IPFS nos permite que los archivos no residan en un solo punto, sino que están distribuidos a través de una red, mejorando así la redundancia y la disponibilidad.

Automatización y Manejo de Cargas de Trabajo con Kubernetes y Docker

La automatización y la gestión de cargas de trabajo utilizando Kubernetes y Docker ha sido crucial para optimizar la gestión de nuestros microservicios, facilitando la alta disponibilidad de datos, la tolerancia a fallos y la escalabilidad. A medida que se buscan mejoras futuras, la implementación de funcionalidades de escalabilidad automática a través de Kubernetes se destaca como una mejora significativa en el sistema. Actualmente, el sistema no dispone de autoescalado, pero se reconoce la importancia crítica de esta funcionalidad para adaptarse dinámicamente a las variaciones en la carga y la demanda. Se planea integrar el Horizontal Pod Autoscaler (HPA) y el Cluster Autoscaler en nuestro sistema para mejorar la escalabilidad. El HPA ajustará automáticamente la cantidad de pods basándose en el uso de CPU y memoria, mientras que el Cluster Autoscaler modificará la cantidad de nodos en función de las necesidades, integrándose con la infraestructura de nube. Estas mejoras facilitarán la adaptación dinámica a las demandas del sistema, mejorando la gestión de la carga, la eficiencia y la resiliencia, y garantizando un rendimiento óptimo y la satisfacción del cliente.

Integración y desarrollo web

- **NginX:** La utilización de Nginx fue una parte fundamental de nuestro proyecto ya que, como hemos podido comprobar hasta ahora, toda nuestra infraestructura se basa en un solo punto de acceso a través de un navegador web desde donde el usuario puede interactuar con nuestro sistema. Desde un punto de vista más técnico, gracias a la versatilidad de esta tecnología pudimos realizar un servicio web y un reverse proxy para poder separar la carga de cada máquina y disponer de una comunicación entre ellas totalmente segura y encriptada y, debido al poco consumo de recursos que presenta esta tecnología, era la opción más acertada.
- **React.js:** La implementación con React.js es, debido al mecanismo de funcionamiento de nuestro sistema, de las partes más importantes

implementadas.

Gracias a la gran versatilidad y posibilidades que ofrece este framework, hemos podido crear una interfaz para el usuario atractiva e intuitiva, que expresa al máximo las posibilidades que ofrecen las capas subyacentes a lo que es el frontend.

- **Node.js:** Utilizado en el backend, Node.js permite ejecutar nuestro sistema de manera eficiente, optimizando el manejo de las solicitudes de API gracias a su modelo de eventos no bloqueante. Esta característica es fundamental para procesar un alto volumen de transacciones simultáneamente, lo que genera respuestas rápidas del servidor y una mejor experiencia de usuario. Su integración fluida con tecnologías como MongoDB y su vasto ecosistema de módulos npm amplían aún más su funcionalidad, haciendo que Node.js sea perfecto para la escalabilidad y el rendimiento.
- **MongoDB:** Como base de datos NoSQL, MongoDB juega un papel crucial en el manejo de grandes volúmenes de datos con alta disponibilidad y elasticidad. Su capacidad para escalar horizontalmente y manejar eficientemente datos estructurados y no estructurados es vital para sistemas descentralizados, facilitando la gestión y el acceso rápido a los datos, y apoyando las demandas dinámicas del sistema al garantizar que los datos estén siempre disponibles y sean fácilmente accesibles.

Integración y Desafíos con Blockchain

La implementación de la blockchain prometía mejoras en la trazabilidad y la seguridad de las transacciones para el contador de popularidad de archivos. Sin embargo, durante la implantación se presentaron obstáculos significativos que impidieron su efectiva integración en el sistema. Estos desafíos han permitido identificar áreas críticas para mejora, y destacan la necesidad de un desarrollo continuo y adaptación de estrategias para superar las dificultades iniciales. El aspecto más crucial para abordar es la mejora del contrato inteligente para asegurar una funcionalidad completa y eficiente.