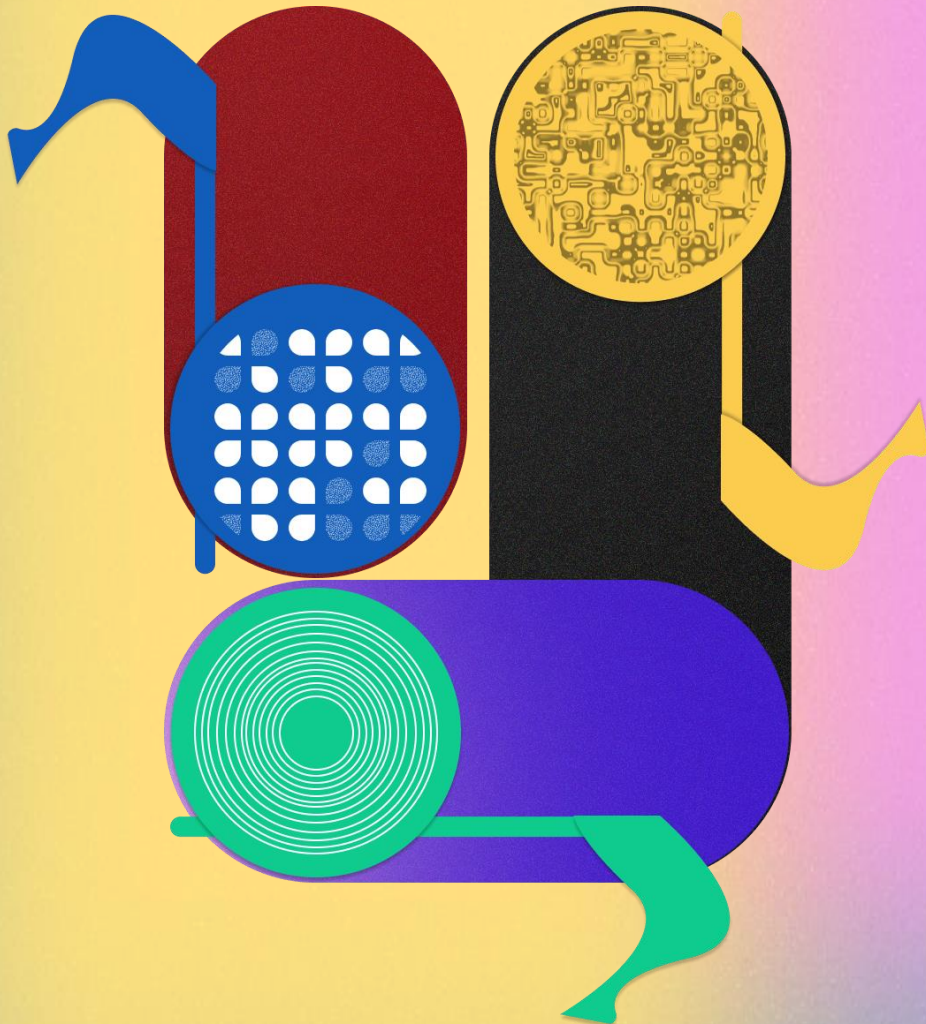




SCORESHARE

GRADO EN DESARROLLO DE APLICACIONES WEB



ALUMNO: GABRIEL ENGUÍDANOS

TUTOR: JOSÉ MIGUEL FAJARDO

Curso: 2023/2024

Centro: IES La Vereda

Información esencial:

scoreShare es una aplicación que permite a los músicos del mundo compartir, comentar y descargar audios, vídeos y partituras relacionadas con canciones específicas.

Informació essencial:

scoreShare es una aplicació que permet als músics d'arreu del món compartir, comentar i descarregar audis, vídeos i partitures vinculades a temes musicals específics.

Essential information:

scoreShare is a web service whose main purpose is to allow musicians all over the world to share, comment and download audio files, videos and music sheets related to specific music tracks.

TABLA DE CONTENIDO

1. Presentación del proyecto	4
1.1. Introducción al proyecto	4
1.1. Módulos que implica	5
1.3 Objetivos iniciales del proyecto.....	6
1.4 Dificultades que debía afrontar desde el inicio	7
2. Plan de trabajo - DIAGRAMA DE GANTT	8
3. Diseño de la aplicación.	10
3.1 Base de datos. Entidad-relación y modelo relacional	10
3.2 Base de datos. Triggers y funciones	13
3.3 Diagrama de casos de uso	15
4. Tecnologías y recursos empleados	16
4.1 Recursos hardware	16
4.2 Recursos software	16
4.2.1 Herramientas	16
4.2.2 Programas, frameworks y lenguajes	17
5. Desarrollo del proyecto	19
5.1 Estructura del proyecto	19
5.2 Servidor	23
5.2.1 Prisma JS.....	23
5.2.2 NextJS.....	25
5.2.3 SUPABASE.....	31
5.3 Cliente.....	32
5.3.1 AXIOS.....	32
5.3.2 SHADCN UI	33
5.3.3 TAILWINDCSS	34
5.3.4 REACT HOOK FORM Y VALIDACIONES.....	35
5.3.4 SPOTIFY	36
6. DESPLIEGUE DE LA APLICACIÓN.....	38
8. MANUAL DE USUARIO	40
9. ANÁLISIS EMPRESARIAL	41
9. CONCLUSIONES.....	43
9.1 Grado de consecución de objetivos	43
9.2 Posibles mejoras.....	43
9.3 Problemas encontrados.....	43

9.4 Sensaciones generales.....	44
ANEXO.....	45
Principales colores usados en la página:	45
ÍNDICE DE IMÁGENES	46
GITHUB.....	47
10. BIBLIOGRAFÍA.....	48

1. PRESENTACIÓN DEL PROYECTO

1.1. Introducción al proyecto

scoreShare es un servicio web que surge de la idea de centralizar la búsqueda de recursos para músicos. Es decir, conseguir que un/a músico amateur opte por visitar **scoreShare** como primera opción si quiere aprender una canción para su instrumento, o, por otra parte, si quiere compartir un arreglo con el mundo.

Para los músicos *amateurs*, en los cuáles me incluyo, es algo tedioso buscar un arreglo de una canción para nuestro instrumento. Es cierto que hay páginas famosas como UltimateGuitar.com, cifraClub.com o Songsterr.com, pero están muy enfocadas a los guitarristas, y la mayoría de las veces te proporcionan tablaturas insertadas en el propio navegador en formato texto.

Y no sólo guitarristas, sino cualquier otro instrumento tiene su primera opción en la web, pero ¿y si un músico toca más de un instrumento?, ¿y si quiere aprender no sólo las canciones grabadas en los discos, sino también alguna versión en vivo que considere mejor?, o ¿y si simplemente quiere descargar el archivo y olvidarse de acceder al navegador cada vez que quiera oírla?

En un intento por solucionar todos estos problemas, **scoreShare** permite la creación de canciones (tracks de aquí en adelante) con el objetivo de crear una marca temporal, un *snapshot* musical. Por poner un ejemplo, *Bohemian Rhapsody* de *Queen* es un instante específico en el tiempo, irreplicable. Y su versión en vivo en el *Live Aid* también lo es. Cada una con sus arreglos.

En la medida de lo posible, he intentado mantener la usabilidad y el flujo de la experiencia del usuario lo más simple posible. No es una aplicación con la idea de pasar mucho tiempo en ella con cada acceso, sino que tiene un objetivo específico y lo cumple de manera sencilla.

1.1. Módulos que implica

Módulo	Implementación
Sistemas informáticos	Instalación y uso de sistemas operativos de software libre (en mi caso Manjaro Linux) sobre el cuál he desarrollado mi proyecto. Además, uso de diferentes comandos en la terminal de Linux para ejecutar ciertos scripts y facilitar mi trabajo.
Bases de datos	La base de datos de scoreShare implementa el modelo entidad-relación visto en el primer año, útil para optimizar consultas y validar restricciones entre las diferentes tablas. Uso de transacciones y triggers.
Entornos de desarrollo	Uso de Git como software de control de versiones.
Programación	El código de scoreShare ha sido en la medida de lo posible, refactorizado para evitar código duplicado y <i>dummy code</i> , abstraer entidades y diseñar funciones pequeñas, específicas y modulares. Uso de bloques try-catch, bucles anidados, uso de Sets, Arrays, operadores ternarios, sorting y recursividad.
Lenguaje de marcas	Uso de multitud de etiquetas HTML como audio, video, img, form, footer, section, nav, input, label, button, select y uso de HTML semántico.
Entorno servidor	Los conocimientos en la creación de APIs y subida de formularios me han permitido separar la lógica de negocio de la vista frontal y poder subir archivos binarios al servidor.

Entorno cliente	Las prácticas realizadas con el framework React durante el curso nos han permitido desarrollar de manera dinámica lo que no se podría haber conseguido con Vanilla JS. La gestión de formularios complejos, el cambio de información al instante en la vista, la realización de peticiones https para obtener datos, desestructuración de objetos, programación asíncrona y componentes modulares e intercambiables han sido los puntos principales que han dado vida a esta aplicación.
Diseño de interfaces	Los conocimientos sobre animaciones, responsive design y mantener un estilo coherente a lo largo de la página me han permitido darle un aspecto moderno y estético al frontal.

1.3 Objetivos iniciales del proyecto

Los objetivos iniciales de la aplicación en su concepción fueron los siguientes:

- Registro de usuario mediante email y contraseña o incluir la opción de registrarse mediante proveedores de OAuth.
- El usuario registrado podrá:
 - Editar su información.
 - Crear y listar tracks.
 - Subir, listar, editar o eliminar archivos.
 - Descargar archivos subidos por otros usuarios.
 - Comentar y responder a otros comentarios.
 - Buscar canciones de manera sencilla.
 - Votar en comentarios y archivos.
 - Relacionarse con otros usuarios mediante solicitudes de amistad.
 - Recibir notificaciones.
 - Editar tracks mediante editRequests.

1.4 Dificultades que debía afrontar desde el inicio.

Desde el principio del desarrollo sabía que debía hacer frente a una serie de dificultades. He aquí expuestas:

- ¿Cómo crearía un usuario un **Track** para mantener la integridad del “snapshot musical”? ¿Qué restricciones debe tener?
- Algunos campos de formularios debían ser de dos tipos de datos totalmente diferentes y debían ser totalmente intercambiables (una imagen de portada puede ser un archivo o un enlace a internet), ¿cómo afrontarlo?
- ¿Cómo subir archivos binarios (formData) en un formulario gestionado por React, y, por tanto, en formato JSON?
- ¿Dónde guardaría los archivos binarios una vez han sido verificados?
- ¿Cómo obtendría de manera granular cada archivo una vez el usuario quiera acceder a él, pero sin descargarlo, sino proyectando una preview en el navegador?
- ¿Cómo obtengo información de una API tan grande y restrictiva como Apple-Music, AmazonMusic o Spotify?

2. PLAN DE TRABAJO - DIAGRAMA DE GANTT

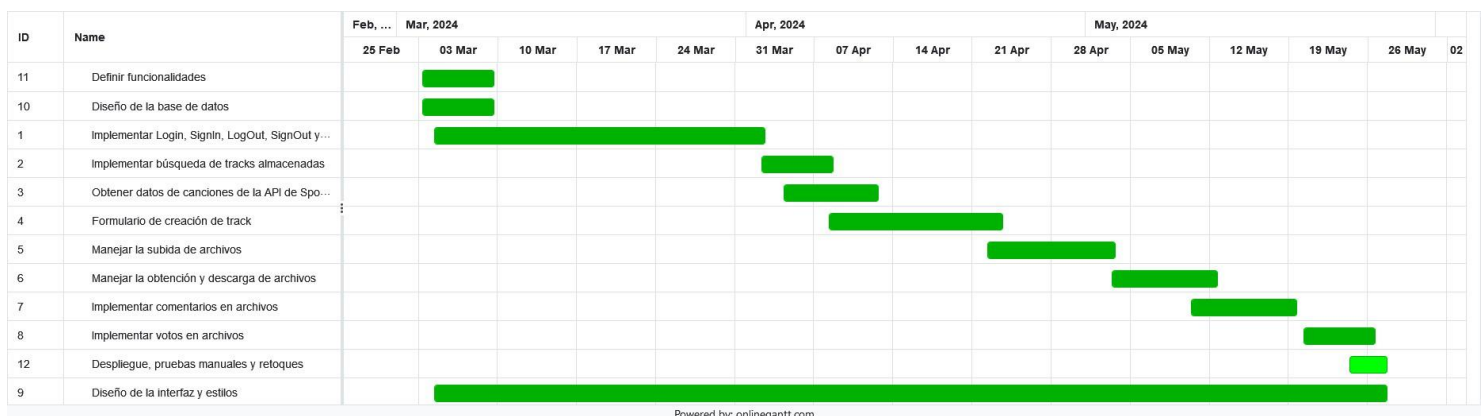
Al inicio del desarrollo se plantearon una serie de funcionalidades que se definieron como muy importantes para lograr el objetivo principal. Estas funcionalidades estaban ordenadas en el tiempo y teniendo en mente las metodologías ágiles, cada una debía aportar un **valor añadido real** a la aplicación.

Todo el código secundario se ha dado por hecho en la funcionalidad principal. Por ejemplo, en el caso de la funcionalidad “Manejar obtención y bajada de archivos”, se da por hecho que se diseña un *HTML* dinámico capaz de albergar filas en una tabla, que se implementan los filtros para buscar entre los archivos obtenidos, que se diseña la ventana de previsualización y que se implementa el código necesario para descargarlo.

Por ello, los objetivos principales que han ido **aportando valor** de manera incremental han sido:

- Implementar Login, SignIn, LogOut, SignOut y protección de rutas con AuthJS.
- Implementar búsqueda de tracks almacenadas
- Obtener datos de las canciones de la API de Spotify
- Formulario de creación de track
- Manejar la subida de archivos
- Manejar la obtención y la descarga de archivos
- Implementar comentarios en archivos
- Implementar votos en archivos
- Diseño de la base de datos
- Diseño de la interfaz y estilos
- Escribir memoria

A continuación, se muestra el diagrama de Gantt de tales funcionalidades:



Como se aprecia en el diagrama, la primera semana se diseñó el primer diagrama ER y se concretaron las funcionalidades principales. Al mismo tiempo, se iba buscando información sobre la próxima funcionalidad.

Al principio el desarrollo fue lento y muy espaciado en el tiempo, por eso da la sensación de que la implantación del *login* fue lo más costoso. Sin embargo, cuando las sesiones de desarrollo fueron más cercanas en el tiempo, el formulario de creación de **Track** ha sido la funcionalidad que más ha costado de implementar para que funcione al 100%.

Hay funcionalidades que se pensaba que iban a llevar más tiempo, como lo es la obtención de datos de la API de Spotify. Sin embargo, la excelente documentación hizo posible que se implementara en poco tiempo.

Por último, comentaremos la funcionalidad “Diseño de la interfaz y estilos”.

El diseño de la interfaz ha comprendido desde la primera línea de código hasta la última. Esto es debido a que no se tenía una idea clara del diseño, sino que los estilos han ido evolucionando hasta encontrar un punto en el que se conciben como un todo. Cambios de colores, fuentes, maquetaciones, bordes, tamaños y fondos de pantalla se han ido sucediendo hasta dar con la interfaz moderna y juvenil que se ha presentado.

3. DISEÑO DE LA APLICACIÓN.

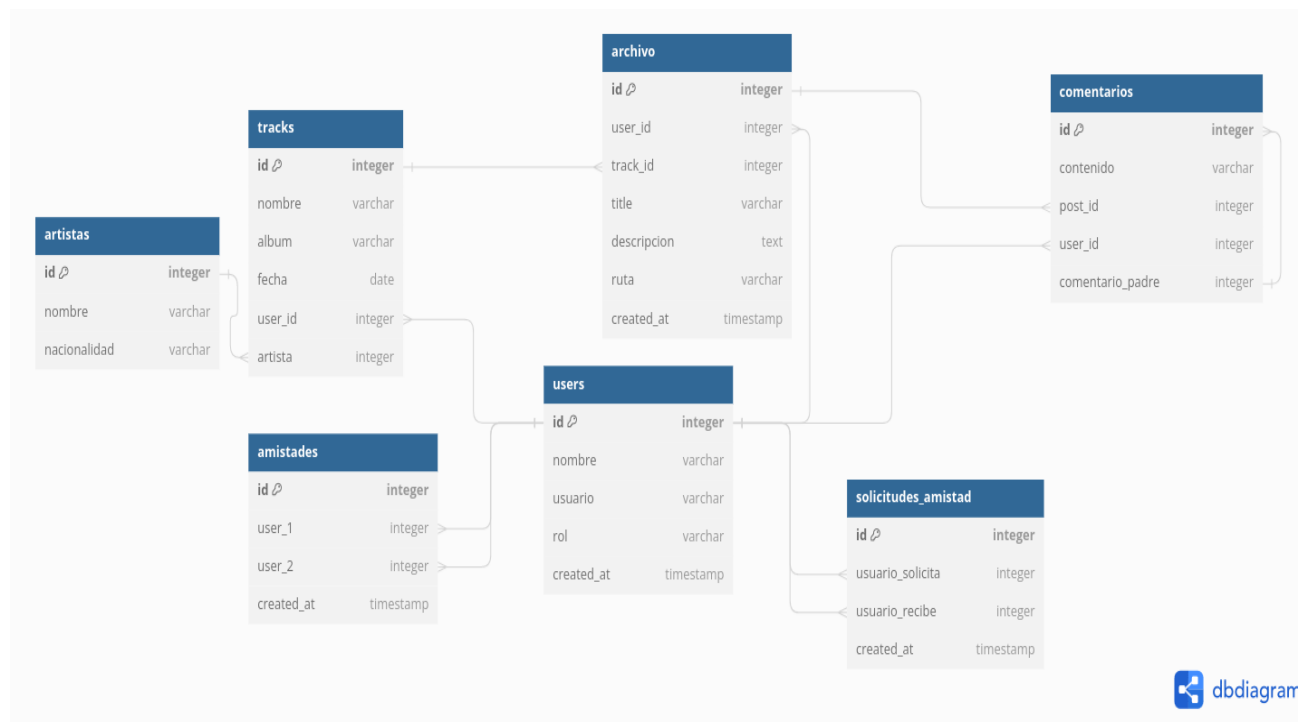
3.1 Base de datos. Entidad-relación y modelo relacional

Como motor de BBDD relacional se ha optado por **PostgreSQL**, que en la práctica es muy similar a **MySQL**. La elección de este programa se debe a dos factores:

1. Trabajar con software distinto al recibido durante el curso.
2. Facilidad para desplegar una base de datos **PostgreSQL** de manera gratuita y *serverless*. Me he apoyado en **NeonTech**, un servicio en la nube de BBDD.

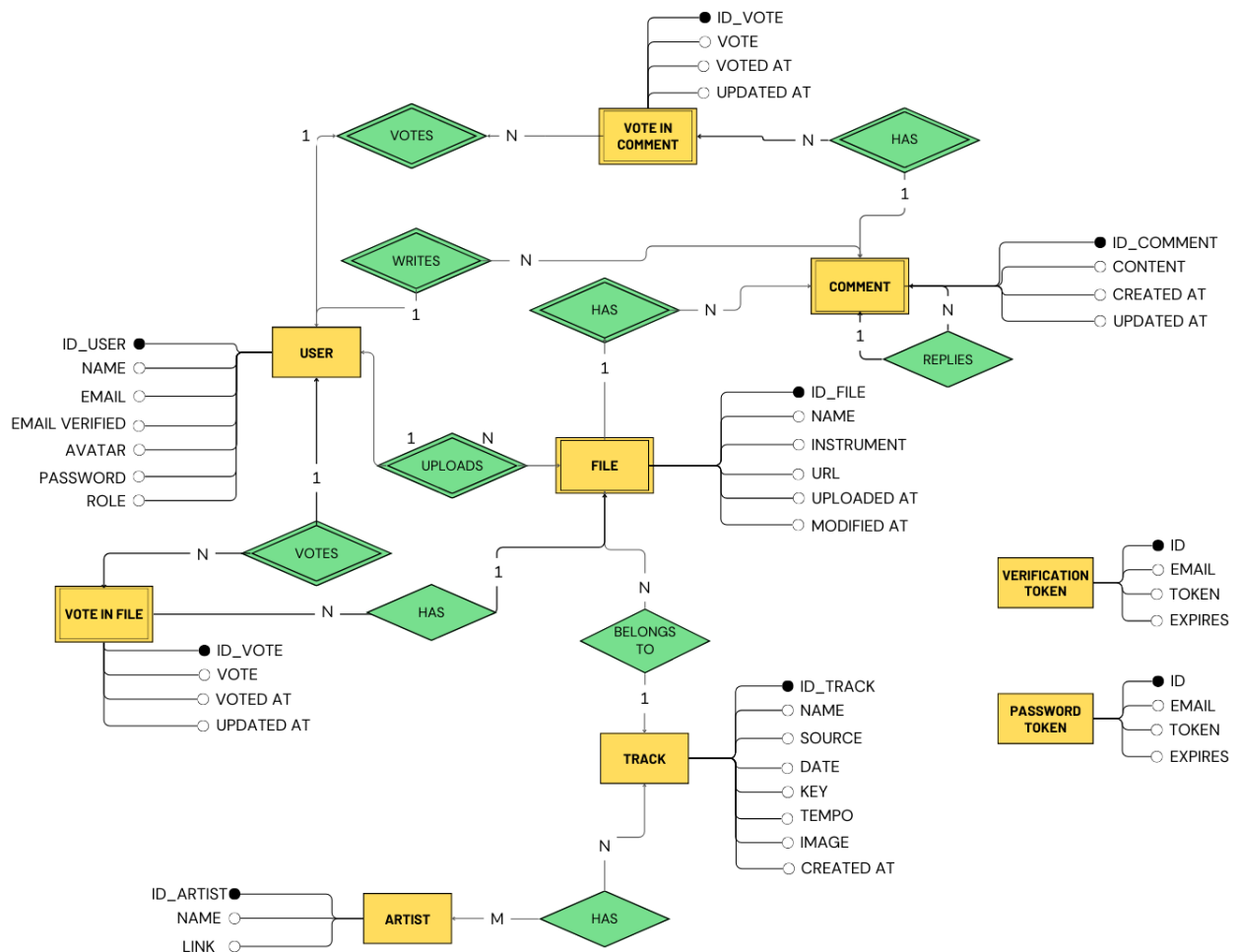
A continuación, se muestra el primer diagrama ER diseñado antes de iniciar el desarrollo.

Captura 1 – Diagrama ER, primera aproximación



Se puede apreciar cómo se intentaba, mediante las tablas *amistades* y *solicitud_amistad*, implementar una funcionalidad social en la aplicación que por falta de tiempo no se ha podido incluir. A continuación, se muestra el modelo ER actual:

Captura2 - Modelo ER actual



Lo cierto es que la base del proyecto no difiere mucho del borrador. Desde el principio tenía claro que las entidades importantes serían **User**, **File** y **Track**.

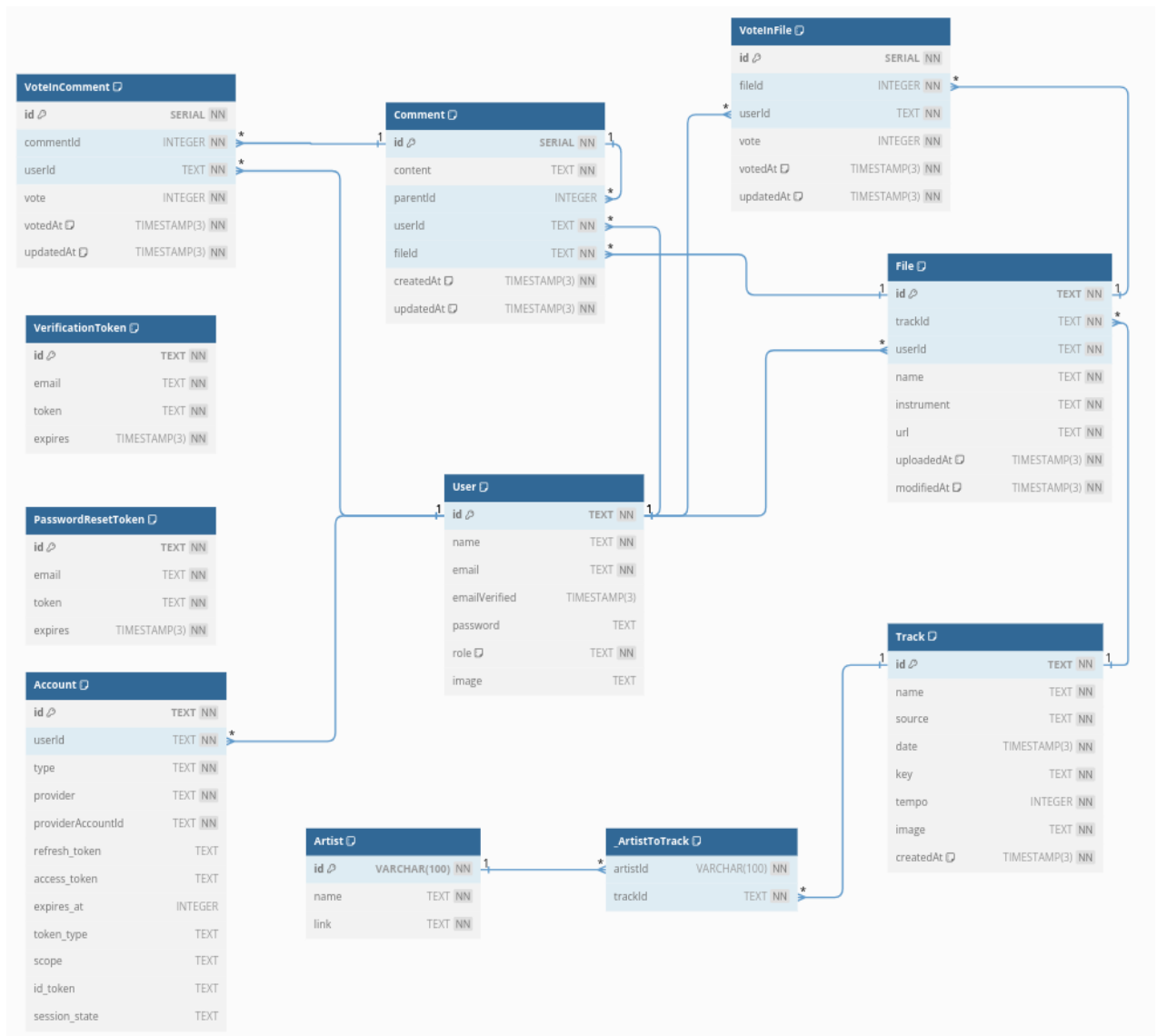
Una de las diferencias es que la primera aproximación con la relación entre **Track** y **Artist** era de 1-M. Sin embargo, fue más tarde modificada a M-N, ya que no se estuvo en cuenta la colaboración entre artistas. Un *track* debe ser accesible desde cualquiera de sus artistas colaboradores.

Otra diferencia radica en la inclusión de las tablas *vote_in_comment* y *vote_in_file* para llevar una cuenta de los votos de cada usuario en cada comentario y archivo, y *verification_token* y *password_token* para tareas de recuperación de contraseña y verificación de email.

Además, en la imagen anterior (Figura 2) no aparece la tabla *Account*, que se relaciona con usuarios con una cardinalidad N-1. Realmente esta tabla es requerida por *Authjs* para adoptar su servicio de autenticación, como se describe en su documentación.

De este modo el diagrama ER actual quedaría de este modo:

Captura 3 – Diagrama ER actual



Algunos apuntes sobre el diagrama anterior ([Captura 3](#)):

- Los usuarios se podrán registrar mediante correo y contraseña o servicios de autenticación como **Google** y **Spotify**. Es por ello por lo que el campo password es opcional. Si un usuario se registra mediante Google, el servicio de autenticación pedirá al usuario que se identifique en su plataforma, pero no le solicitará una contraseña de acceso.
- El campo *name* en **User** NO es único. Si existe la opción de registro mediante Google o Spotify, es muy probable que dos nombres se repitan. El campo *email* sí que es único.

- El campo *role* en **User** fue creado con vistas a que en un futuro se implementase rutas protegidas por roles. Actualmente todas las rutas están protegidas por **autenticación**.
- **File** es la tabla dónde se almacena la información de cada archivo, que podrá ser **mp3**, **mp4** o **pdf**.

Name es el nombre que verán los usuarios en su búsqueda, y la *url* es la ubicación del recurso. En este caso, se ha optado por no guardar los archivos en la propia base de datos, sino relegar dicha tarea a un proveedor externo.

Name, *userId* y *instrument* en la tabla **File** forman una **clave única compuesta**. Dos archivos con el mismo nombre, instrumento y subidos por el mismo usuario puede llevar a una confusión innecesaria para otros usuarios y derivaría en ambigüedades dentro del modelo de datos.

Captura 4 – Restricción única de los campos *userId*, *name* y *instrument*.

```
CREATE UNIQUE INDEX "File_userId_name_instrument_key" ON "File"("userId", "name", "instrument");
```

- Se ha querido implementar un sistema de comentarios recursivo. Para ello es necesario mantener un registro de qué respuestas **pertenecen** a qué comentario. En la entidad **Comment** se ha incluido finalmente una relación **reflexiva**, dónde el campo *parentId* hace referencia al comentario padre. Debe poder ser **null** ya que el primer nivel de comentarios no tendrá un padre.
- A su vez los comentarios podrán ser puntuados por los usuarios mediante **like** o **dislike** (+1 ó -1). Para validar que un usuario sólo inserte un voto en un comentario, cada voto debía ir asociado a un comentario y a un usuario, siendo esta restricción **compuesta** y **única**. De esta manera nos aseguramos de que un usuario sólo podrá votar como máximo una vez en cada comentario, si bien es cierto que podrá cambiar su puntuación si lo desea. Lo mismo ocurre con los votos de los archivos. La puntuación total de un comentario o archivo se obtiene haciendo un sumatorio de todos los votos vinculados.

Captura 5 - Restricción única en la tabla "VoteInComment" para evitar que un usuario vote más de una vez.

```
CREATE UNIQUE INDEX "VoteInComment_userId_commentId_key" ON "VoteInComment"("userId", "commentId");
```

3.2 Base de datos. Triggers y funciones

A parte del diseño de la base de datos y las tablas, también se han definido dos *triggers* con sus correspondientes funciones. Se ejecutarán antes de cada inserción en las tablas *voteInFile* y *voteInComment* y verificarán que el recurso en cuestión no sume un total de **20** votos negativos o más.

Esta regla de negocio se ha establecido así porque se ha considerado lo siguiente: si un recurso sobrepasa el límite de los 20 votos negativos teniendo en cuenta que un usuario sólo puede votar una vez por cada recurso, se da por hecho que dicho recurso no aporta valor a la aplicación y, por lo tanto, es eliminado.

A continuación, se muestra la definición de dichas funciones para la tabla *Comment*:

Función PLSQL:

```
CREATE OR REPLACE FUNCTION deleteCommentWhenReviewsUnderMinus20 ()
RETURNS TRIGGER
LANGUAGE PLPGSQL
AS
$$
DECLARE
    total integer;
BEGIN
    select sum("vote")
    into total
    from "VoteInComment"
    where "commentId" = NEW."commentId";
    RAISE NOTICE 'Value: %', total;

    if (total <= -20) then
        delete from "Comment" where "id" = NEW."commentId";
        RETURN NULL;
    end if;

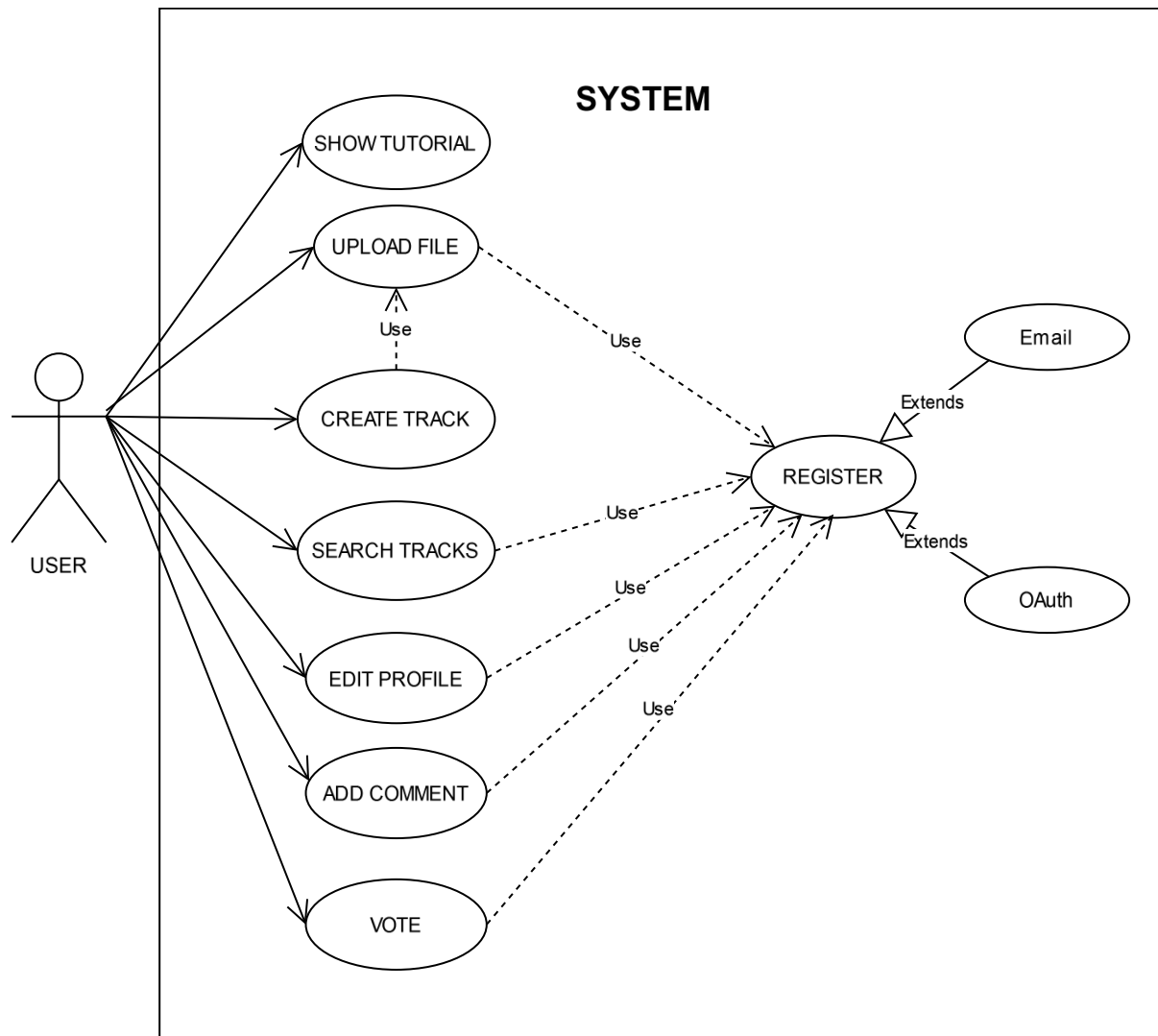
    RETURN NEW;
END;
$$
```

Trigger:

```
CREATE OR REPLACE TRIGGER trigger_check_comment_votes
AFTER INSERT OR UPDATE ON "VoteInComment"
FOR EACH ROW
EXECUTE FUNCTION deleteCommentWhenReviewsUnderMinus20 ();
```


3.3 Diagrama de casos de uso

Captura 6 – Diagrama de casos de uso.



Como se puede apreciar, la aplicación restringe cualquier acción al usuario a menos que este esté registrado. La única excepción es el tutorial de uso.

Por otra parte, la creación de un **track** implica subir un archivo (no se permitirá la creación de *tracks* sin archivos). Por ello se incluye el caso de uso *upload file* en el de *create track*.

4. TECNOLOGÍAS Y RECURSOS EMPLEADOS

4.1 Recursos hardware

- Ordenador de sobremesa con SO Manjaro Linux.
- Doble pantalla HDMI 2.0 para un mejor flujo de trabajo.
- Conexión Ethernet.
- Ratón ergonómico y teclado de membrana.


4.2 Recursos software

4.2.1 HERRAMIENTAS

- Visual Studio Code como IDE.
- Mozilla Firefox como navegador.
- Figma para el diseño del logo.
- Youtube para búsqueda de información.
- StackOverflow para resolución de dudas.
- dbDiagram para realizar los diagramas ER.
- Drawio.com para realizar el diagrama de casos de uso.
- Canvas para realizar el modelo ER.
- CSSGRADIENT.IO para generar gradientes de manera interactiva.
- Microsoft Word para la realización.

4.2.2 PROGRAMAS, FRAMEWORKS Y LENGUAJES

Tecnología	Descripción
 <p data-bbox="373 656 472 689">Nextjs</p>	<p data-bbox="668 459 1388 577">Nextjs es un framework Node.js fullstack de React, que permite el renderizado de componentes en el servidor y en el cliente.</p> <p data-bbox="668 622 1388 741">Al ser fullstack, permite la definición de APIs, middleware, rutas, servicio de autenticación y conexiones con base de datos.</p>
 <p data-bbox="344 1037 501 1070">NeonTech</p>	<p data-bbox="668 846 1260 880">Se definen como un Postgres Serverless.</p> <p data-bbox="668 925 1388 1133">Lo cierto es que es una base de datos Postgres en la nube, fácil de acceder y con plan gratuito. Me ha facilitado mucho el flujo de trabajo al no tener que preocuparme de levantar en local o de exportar el esquema si quisiera desplegarlo en otro equipo.</p>
 <p data-bbox="327 1435 518 1469">Prisma ORM</p>	<p data-bbox="668 1238 1388 1525">Para el proyecto he optado por usar un ORM como Prisma. Un ORM es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y la utilización de una base de datos relacional como motor de persistencia. Ejemplos similares podrían ser Hibernate o Eloquent.</p> <p data-bbox="668 1570 1388 1648">Prisma está totalmente integrado con JS y Postgres.</p>
 <p data-bbox="429 1798 518 1832">AXIOS</p>	<p data-bbox="668 1753 1388 1872">Axios es un cliente de JavaScript utilizado principalmente para realizar y manejar peticiones asíncronas.</p>

 Shadcn UI	<p>Shadcn es una librería de componentes basados en TailwindCSS que se instalan de manera abierta para permitir modificaciones integrales.</p>
 TailwindCSS	<p>TailwindCSS es un framework de CSS basado en clases predefinidas cuya personalización excede de manera notable frente a otros frameworks de la competencia. Aunque shadcn haga uso de Tailwind, lo incluyo por separado porque lo he usado en el 99% de los estilos. Permite tener un archivo CSS prácticamente vacío y personalizar sus valores por defecto con sintaxis JS.</p>
 React	<p>React es un framework de JS, y a su vez es la base del framework NextJS.</p> <p>Permite crear componentes reutilizables y modulares en el lado cliente mediante funciones y estados y actualizar la vista mediante eventos, entre otras cosas. Se integra perfectamente con miles de librerías de todo tipo.</p>
 Supabase	<p>Supabase es un <i>backend as a service</i>. Esto quiere decir que permite a los desarrolladores disponer de una base de datos en la nube, así como un servicio de autenticación y distintas herramientas para gestionar archivos.</p>

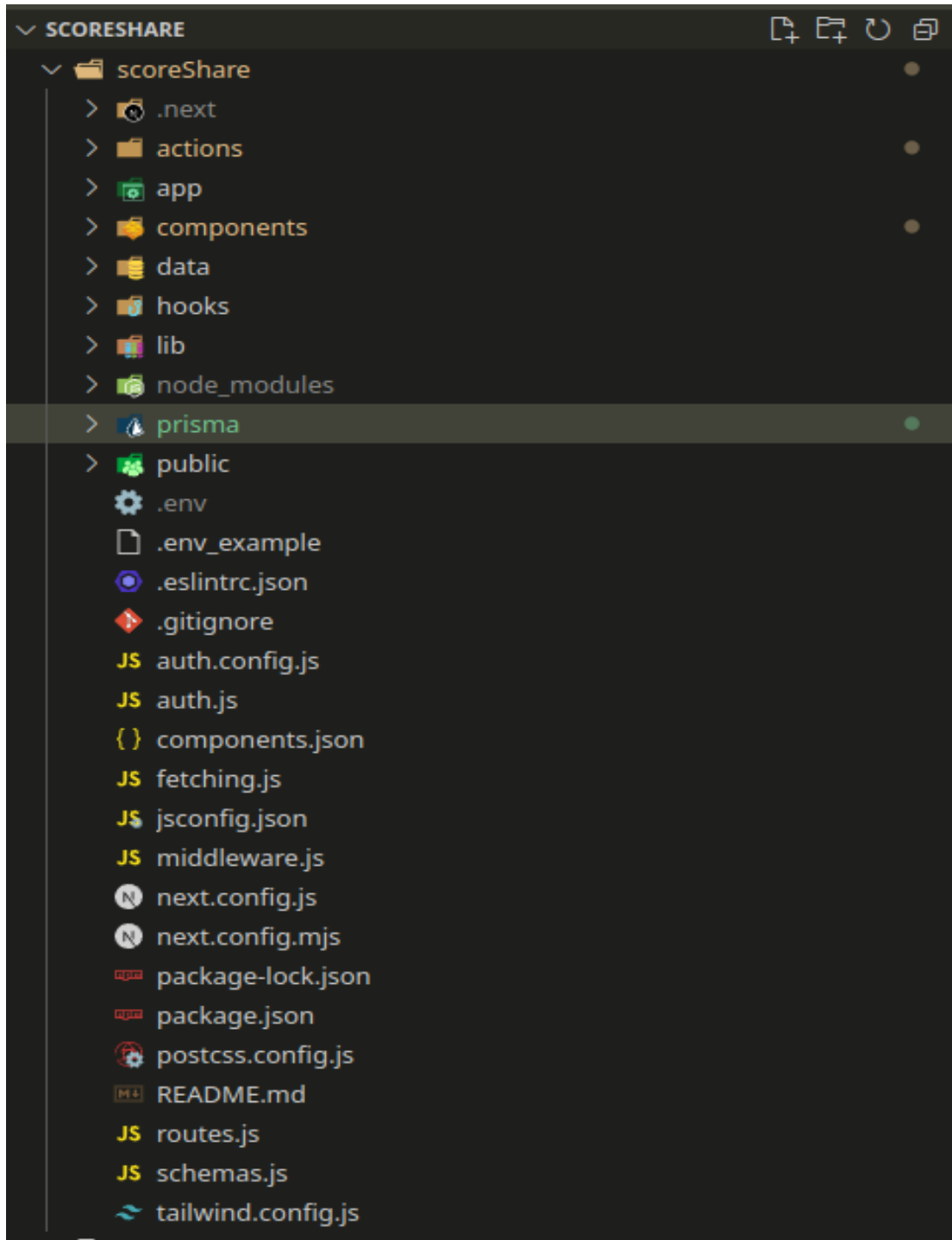
Como se puede apreciar, ninguna de las tecnologías, excepto **Axios y React**, se han visto durante el temario del curso. Si bien su aprendizaje ha lastrado el ritmo del desarrollo, la facilidad con la que resuelven los problemas compensa todo ese tiempo invertido, excepto algún caso concreto (hablaremos de **Prisma** en el apartado *Problemas encontrados*).

5. DESARROLLO DEL PROYECTO

5.1 Estructura del proyecto

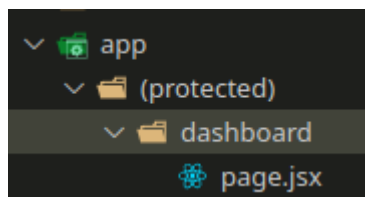
Nextjs es el framework utilizado para desarrollar la aplicación. Se mostrará una captura de la estructura de directorios de la aplicación acompañada de una explicación (se obviarán las carpetas autogeneradas *.next*, *node_modules* y los archivos de configuración autogenerados por **NextJS** como *.eslintsrc*, *next.config.** y *jsconfig.json*):

Captura 7 – Estructura de directorios del proyecto



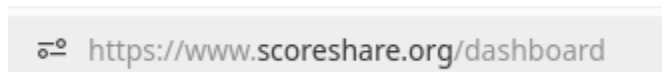
- **./actions:** las acciones o *server actions*, son funciones que se ejecutarán en el lado del servidor (acceso a datos mayoritariamente) y que pueden ser llamados desde un componente **cliente** sin necesidad de realizar una petición HTTP al uso. Es una funcionalidad nativa del framework.
- **./app:** en esta carpeta se definen las rutas y las páginas. Las rutas se crean conforme se anidan las carpetas. Por ejemplo, si quiero una ruta **/protected/dashboard**, deberemos crear una carpeta **protected** que contenga la carpeta **dashboard**. A su vez si queremos que una página se renderice, debemos incluir un archivo **page.jsx** (obligatorio este nombre).

Captura 8 – Ejemplo de anidación de carpetas para crear rutas anidadas



De este modo tenemos lo que queríamos, dentro de la ruta *protected* podremos navegar a las rutas hijas, entre ellas *dashboard*. Si vamos al navegador:

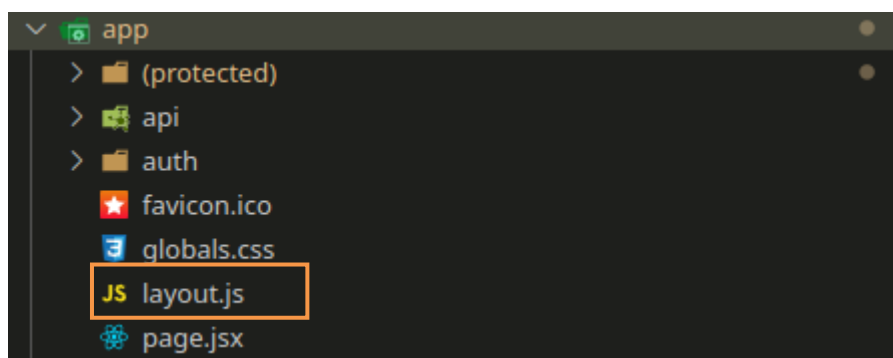
Captura 9 – Url que el usuario visualiza en el navegador tras el ejemplo [anterior](#)



Aquí entra la convención de nombres de **NextJS**. Entre una de ellas tenemos la opción de envolver el nombre de una carpeta entre paréntesis para que no aparezca en la *url* que se muestra al usuario.

- En la carpeta **./app** también se puede ver otra de las bondades de **NextJS**, el uso de **layouts**:

Captura 10 – Archivo layout en la raíz de la aplicación



Un *layout* no es una página renderizable por sí misma, sino que es el conjunto de elementos que comparten un grupo de páginas hijas.

Si nos vamos a la definición de *layout* de la raíz de la aplicación veremos el siguiente código:

Captura 11 - Contenido del archivo layout.js en la raíz de la aplicación

```
export default async function RootLayout({ children }) {
  const session = await auth();
  return (
    <SessionProvider session={session}>
      <html lang="en">
        <body
          className={cn(...inputs: font.className, "antialiased text-primary font-l...
        >
          {children}
        </body>
      </html>
    </SessionProvider>
  );
}
```

Una etiqueta **html** y una etiqueta **body** envuelven a la variable **children**. El objetivo de esto es que todas las páginas hijas (*children*) se incluyan dentro de sendas etiquetas, ahorrando código repetido. La inclusión de un layout es opcional por cada ruta que existe.

- También podemos ver el archivo **globals.css**. Incluye ciertos atributos que abarcan muchos componentes y es más óptimo incluirlos en un archivo común, como por ejemplo:

Captura 12 - Atributos compartidos por todas las etiquetas input y button.

```
input[type="file"] {
  display: none;
}

button[disabled] {
  cursor: not-allowed;
}
```

- **./components:** esta carpeta contiene todos los componentes reutilizables de la aplicación.
- **./data:** es la encargada del acceso directo a la base de datos. Es una suerte de *pseudorepositorio* con el objetivo de separar la lógica de negocio del acceso a datos. Hay un archivo por cada clase. Un ejemplo sería la creación de un **File** en el archivo **./data/file.js**. Como se puede apreciar, la función recibe e inserta datos, pero no ejecuta ningún tipo de lógica de negocio:

Captura 13 - Función CREATE para insertar un archivo en la BBDD.

```
65 export const createFileData = async (conn = db, file) => {
66   const result = await conn.file.create({
67     data: file,
68     include: { user: { select: { name: true, image: true } } }
69   });
70   return result;
71 };
```


- **./hooks:** esta carpeta contiene las funciones mutables de React, que permiten manejar y cambiar el estado de un componente. Los formularios en la aplicación se comportan de manera similar: todos reciben una respuesta, y depende del estado de ésta, se informará al usuario de un error o de un éxito en la operación. El ejemplo anterior se ejemplifica del siguiente modo:

Captura 14 - Ejemplo de definición de hook personalizado.

```
export const useResponseMessages = () => {
  const [success, setSuccess] = useState(initialState: "");
  const [failed, setFailed] = useState(initialState: "");

  const resetMessages = () => {
    setSuccess(value: "");
    setFailed(value: "");
  };

  return { success, failed, setFailed, setSuccess, resetMessages };
};
```

Y en cada formulario debo incluir lo siguiente:

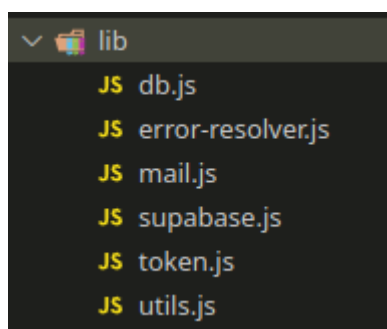
Captura 15 – Uso del hook personalizado definido en la [Captura 13](#)

```
const { success, failed, setFailed, setSuccess, resetMessages } = useResponseMessages();
```

De este modo la lógica queda encapsulada y puede ser reutilizada de manera independiente por cada formulario.

- **./lib:** esta carpeta contiene todas las funciones auxiliares, como por ejemplo son la conexión a la BBDD, el envío de emails, la función generadora de tokens..., o las funciones **utils**. El filtro de palabras prohibidas es una de ellas:

Captura 16 - Archivos en la carpeta lib y ejemplo de función dentro de utils.



```
export function hasForbiddenContent(value) {
  let exists = false;
  let i = 0;
  const toUpperCase = value.toUpperCase();
  const forbiddenContentLength = FORBIDDEN_CONTENT.length;

  while (!exists && i < forbiddenContentLength) {
    const word = FORBIDDEN_CONTENT[i++];
    exists = toUpperCase.includes(word);
  }

  return exists;
}
```

- **./prisma:** esta carpeta contiene el archivo **schema.prisma** donde se definen las tablas mediante sintaxis específica del ORM. Será el propio programa quien se encargue de crear el **DDL** de manera implícita.
- **./public:** en esta carpeta se guardan las imágenes estáticas que usa la aplicación (fondos de pantalla, avatares por defecto, etc...)

- **.env**: aquí se definen las variables de entorno, como la **url** de la base de datos, el usuario y la contraseña, los **client_id** y **client_secret** de los distintos servicios de **Oauth** y el dominio de la aplicación.
- **middleware.js**: en este archivo se definen qué rutas estarán disponibles para usuarios autenticados y cómo se debe redirigir al usuario en caso de que ya estén autenticados y quieran acceder a las rutas de *login* o *register*, por ejemplo.

5.2 Servidor

5.2.1 PRISMA JS

Como ya se ha comentado, el encargado de comunicarse con la base de datos es el ORM **Prisma**, que goza de completa integración con JS y PostgreSQL.

Prisma permite definir un esquema relacional de manera muy sencilla. Veamos la definición de la tabla **Track** y **Artist**:

Captura 17 - Definición de tablas mediante la sintaxis de Prisma

```
model Track {
  id      String  @id @default(cuid())
  name    String
  source  String
  date    DateTime
  key     String
  tempo   Int
  image   String
  createdAt DateTime @default(now())
  artists Artist[]
  files   File[]
}

model Artist {
  id      String  @id @default(uuid()) @db.VarChar(100)
  name    String
  link    String
  tracks  Track[]
}
```

Como se puede apreciar, el lado N de la relación se representa con un **Array**, y las etiquetas “@” permiten definir restricciones (@@unique) o valores por defecto.

La magia de Prisma es que, pese a que **Artist** y **Track** mantienen una relación N-M, nosotros como desarrolladores no tenemos que definirla. Es el propio ORM el encargado de instanciar la relación de manera implícita.

Además, si se necesita añadir una nueva tabla, se define el modelo del mismo modo que en el ejemplo anterior, y con el comando “**prisma db push**”, la base de datos se actualiza para incluir el nuevo modelo y sus relaciones:

Captura 18 - Comando de Prisma para actualizar la base de datos.

```
npx prisma db push
```

Para realizar operaciones CRUD, Prisma define una serie de convenciones.

Lo primero que hay que hacer es **obtener una instancia** del ORM. Después le indicamos en qué **tabla** (*camelCase*) queremos realizar la operación, y por último indicamos el **tipo de operación**.

Obtener todos los archivos vinculados a un *track*: en el archivo `./data/file.js`, que es el encargado del acceso a datos, se realizará la operación ***findMany***, y filtraremos todos aquellos archivos cuyo ***trackId*** sea igual al ***id*** recibido por parámetro:


Captura 19 - Ejemplo de consulta a base de datos mediante Prisma.

```
✓ export const getFilesByTrackId = async (id) => {
✓   const files = await db.file.findMany({
✓     orderBy: [
✓       {
✓         uploadedAt: "desc",
✓       },
✓     ],
✓     where: {
✓       trackId: id,
✓     },
✓     include: {
✓       user: {
✓         select: {
✓           name: true,
✓           image: true,
✓         },
✓       },
✓     },
✓   });
  return files;
};
```

Además, a la hora de hacer **JOINS**, con *Prisma* sólo debemos decirle que incluya alguna de las tablas relacionadas, y además de esta tabla, podemos indicarle qué campos queremos obtener. En este ejemplo, hacemos un *JOIN* con la tabla **User**, y sólo obtenemos *name* y *image*.

Este tipo de consultas son muy cómodas para mostrar información de dos tablas al mismo tiempo, como, por ejemplo:

Captura 20 - Uso de JOINS para obtener datos de archivo y usuario al mismo tiempo.

Name	Intrument	Created by
Guitar Solo At Minute 2 <small>mp3</small>	Acoustic guitar	 Gabriel Enguidanos in 1 minute

En el ejemplo anterior mostramos tanto información del **archivo** (nombre e instrumento), como información del **usuario** (nombre y avatar).

5.2.2 NEXTJS

NextJS nos provee de rutas anidadas (como se ha visto [anteriormente](#)) y la opción de definir *server actions* y APIs.

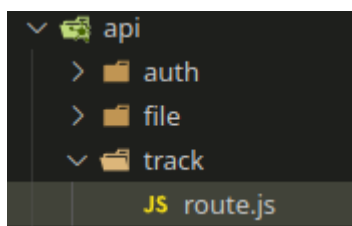
Con tal de poner en práctica lo aprendido durante el curso, se han definido algunos **endpoints** que reciben peticiones y devuelven respuestas *HTTP*.

Sin embargo, para integrarse con el propio framework y utilizar las herramientas que **NextJS** nos proporciona de manera nativa, también se ha hecho uso de los *server actions* de manera análoga a los API *endpoints*.

API:

Para definir un *endpoint* que reciba peticiones *HTTP*, en NextJS debemos especificar una carpeta llamada **api** (este nombre debe ser exacto ya que forma parte de la convención de nombres). Dentro podemos concretar cuántas carpetas queramos, cada una referenciando una ruta anidada ([ver ejemplo](#)), y finalmente debemos definir el archivo encargado de manejar las peticiones, **route.js** (nombre obligatorio).

Captura 21 - Ejemplo de definición de un API endpoint.



Y en el archivo **route.js** concretamos el manejo de peticiones:

Captura 22 - Ejemplo para manejar peticiones en un archivo route.js.

```
export async function GET(request) {
  const url = new URL(request.url);
  const query = url.searchParams.get(name: "query");
  const result = await getTrackByQueryData(query);

  return Response.json(data: result);
}

export async function POST(request) {
  const body = await request.formData();
  let finalImage = body.get("image");

  try {
    if (typeof finalImage != "string") {
```

El **endpoint** quedaría de este modo: **www.scoreshare.org/api/track/**.

SERVER ACTIONS:

Las acciones en el servidor cumplen la misma función que un API *endpoint* en este proyecto. Son archivos JS que definen operaciones **CRUD** más específicas y se pueden llamar desde un componente cliente, es decir, no es necesario realizar una llamada *HTTP*.

Para poder controlar las respuestas de estas acciones y mostrar un mensaje acorde en la vista, se han **definido en la carpeta *utils* algunas constantes** indicando el código de estado de la respuesta, que se maneja manualmente.

Captura 23 - Server action addComment: Insertar/actualizar un comentario en la base de datos.

```
27 export const addComment = async (formData) => {
28   const { user } = await auth();
29   const id = parseInt(formData.get("id")) || null;
30   const content = formData.get("content");
31   const fileId = formData.get("fileId");
32   const parent = parseInt(formData.get("parent")) || null;
33   const comment = {
34     content: content,
35     parentId: parent,
36     userId: user.id,
37     fileId: fileId,
38   };
39
40   let result;
41   let status;
42
43   try {
44     if (hasForbiddenContent(content))
45       throw new Error("It is not allowed to insert inappropriate content!");
46     if (id) {
47       result = await updateCommentData(id, content);
48       status = OK;
49     } else {
50       result = await insertCommentData(comment);
51       status = CREATED;
52     }
53
54     return { status: status, data: result };
55   } catch (error) {
56     const [status, message] = resolveError(error);
57     return { status: status, message: message };
58   }
59 };
```

La función anterior se encarga de validar un comentario e insertarlo en la base de datos. Sería el análogo a un **POST** en *api/comment/*.

Para insertar un comentario, primero se verifica su **id**. Si éste existe, se trata de un comentario que se quiere actualizar. Si es nulo o indefinido, se tratará como un comentario nuevo, previa validación de contenido inapropiado. A continuación, se hace la llamada al método correspondiente en el archivo *./data/comment.js*.

En la **línea 36** se verifica que es un código de éxito con la función personalizada **is2xx()** del archivo *utils*:

Captura 23 – Llamada a server action en línea 34 y manejo de respuesta desde el cliente

```

29  const handleAddComment = async (formData) => {
30    formData.append("fileId", file.id);
31    if (parentComment.current) formData.append("parent", parentComment.current);
32    if (commentId.current) formData.append("id", commentId.current);
33
34    const result = await addComment(formData);
35
36    if (is2xx(value: result.status)) {
37      let newComments;
38      if (result.status === CREATED) {
39        newComments = [...comments, result.data];
40      } else {
41        newComments = comments.filter(
42          predicate: (comment) => comment.id !== result.data.id
43        );
44        newComments.push(...items: result.data);
45      }
46      setComments(value: newComments);
47      textArea.current.value = "";
48    } else {
49      setError(value: result.message);
50    }
51  };

```

En las líneas **30** a **32** se recoge la información del formulario y se envía al servidor mediante el *server action* **addComment**.

Si el código devuelto por el *server action* es un **CREATE (201)**, se añadirá el nuevo comentario a la lista que se está mostrando. Si por el contrario es un **UPDATE (200)**, se buscará dicho *id* en la lista y el objeto (comentario) se reemplazará.

SERVER COMPONENTS:

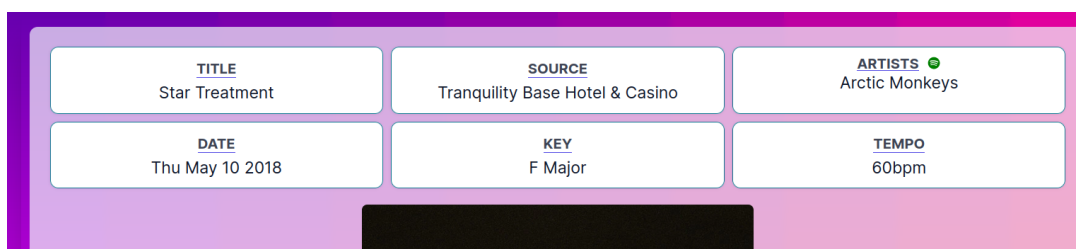
NextJS es capaz de renderizar **dos tipos** de componentes:

Los **componentes cliente** son los que interactúan directamente con el navegador y el usuario, y generan cambios de estado y cambios en la vista, típicos de React.

Los **componentes de servidor** permiten obtener datos y entregar componentes ya renderizados al navegador para aligerarlo de carga de trabajo.

Una implementación de un componente servidor es la página dónde el usuario puede ver la información de un *track*:

Captura 24 - Información mostrada al usuario obtenida mediante componente servidor.



Debido a que esta información no va a cambiar de estado ni el usuario va a interactuar con ella, podemos obtenerla directamente desde el servidor, que nos devolverá (**return**) la tabla ya renderizada.

Captura 25 - TrackInfo: Ejemplo de un componente servidor accediendo a datos directamente.

```
export const TrackInfo = async ({ id }) => {  
  const trackInfo = await getTrackByIdData(id); //acceso directo a datos  
  return (  
    <section <...>  
      {trackInfo && (  
        <>  
          <div className="order-2 w-3/4">  
            <Image  
              alt=""  
              width={800}  
              height={0}  
              quality={100}  
              <...>  
              src={trackInfo.image}  
            />  
          </div>  
        </>  
      )}  
    </section>  
  )  
}
```

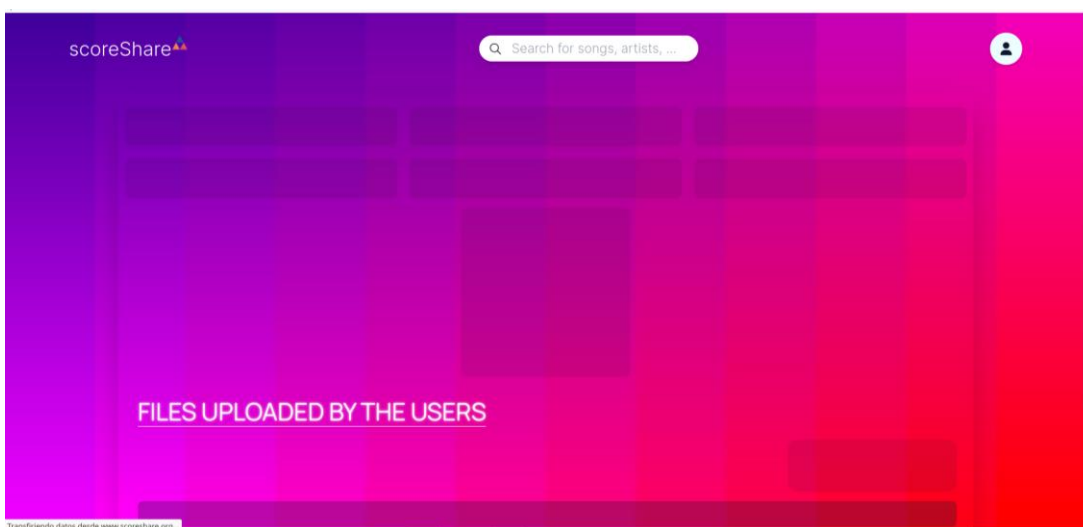
Cómo la llamada a base de datos es asíncrona, NextJS nos proporciona una manera de mostrar al usuario otra vista mientras se obtienen los datos. Esto se hace mediante la función **Suspense**:

Captura 26 - Uso de Suspense para mostrar una vista diferente mientras se carga el componente.

```
<Suspense fallback={<TrackSkeleton />}>  
  <TrackInfo id={params.id} />  
</Suspense>
```

Esta función mostrará un *esqueleto* mientras se obtienen los datos:

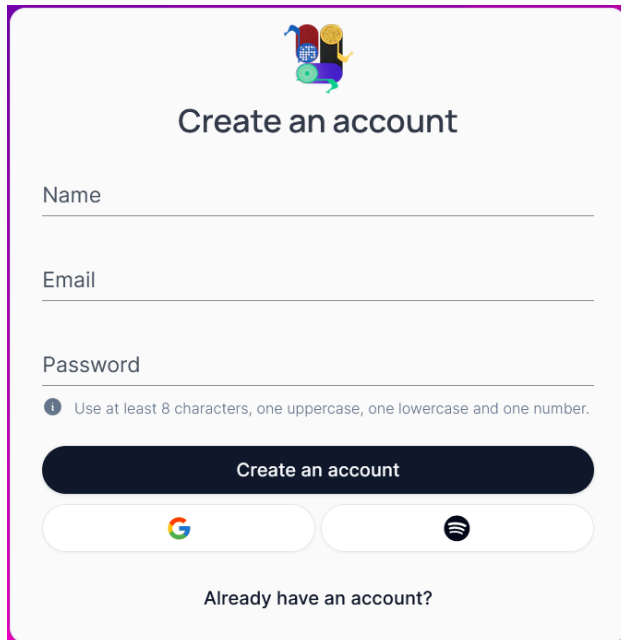
Captura 27 - Vista de un esqueleto mientras se carga el componente.



AUTHJS:

Authjs es el servicio que incorpora Nextjs por defecto y se usa en la aplicación. Nos permite a los desarrolladores despreocuparnos de gestionar las sesiones y nos proporciona herramientas para el inicio de sesión mediante proveedores externos.

Captura 28 – Opciones de registro para el usuario.



La aplicación permitirá a los usuarios registrarse manualmente o mediante Google o Spotify. Para ello definimos los distintos proveedores en el archivo *auth.config.js*:

Captura 29 – Definición de los proveedores de autenticación en el archivo auth.config.js.

```
12 export default {
13   providers: [
14     Spotify({
15       clientId: process.env.NEXT_PUBLIC_SPOTIFY_CLIENT_ID,
16       clientSecret: process.env.NEXT_PUBLIC_SPOTIFY_CLIENT_SECRET,
17     }),
18     Google({
19       clientId: process.env.GOOGLE_CLIENT_ID,
20       clientSecret: process.env.GOOGLE_CLIENT_SECRET,
21     }),
22     Credentials({
23       async authorize(credentials) {
24         try {
25           const { email, password } = await LoginSchema.validate(credentials);
26
27           const user = await getUserByEmail(email);
28           if (!user || !user.password) return null;
29
30           const passwordMatch = await bcrypt.compare(password, user.password);
31           if (passwordMatch) return user;
32
33         } catch (err) {
34           resolveError(err)
35         }
36         return null;
37       },
38     }),
39   ],
40 };
```

Siendo **Credentials** el típico *login* por correo y contraseña:

1. Verificamos que los campos **email** y **password** cumplen ciertos requisitos gracias a la validación de formularios con **Yup**.
2. Se busca al usuario con el email recibido. Si no existe el usuario o en cambio sí que existe, pero la contraseña es **null**, (se ha dado de alta mediante proveedor externo) devolvemos **null**.
3. Después comparamos la contraseña recibida con el **hash** en la base de datos. Si coinciden, devolvemos el usuario para que Authjs se encargue de iniciar la sesión. Si no coinciden, se devuelve **null**.

Para iniciar sesión con uno de los proveedores externos, invocamos el método **signIn** de Authjs con el nombre del proveedor y será la librería quién se encargue de crear el usuario en nuestra base de datos o recuperarlo si ya existe:

Captura 30 – Invocamos el método signIn de la librería de authjs para hacer un login con un proveedor externo recibido por parámetro.

```
const onClick = async (provider) => {  
  await signIn(provider, {  
    callbackUrl: DEFAULT_LOGIN_REDIRECT,  
  })  
}
```

OTRAS HERRAMIENTAS DE NEXTJS:

Además de las ya comentadas, se ha usado una variedad de herramientas que NextJS pone a disposición del desarrollador:

- **<Image/>**: componente nativo para sustituir la etiqueta **** de **HTML**. Se caracteriza por cargarse sólo cuando entra en la vista del usuario.
- **Función *revalidatePath***: cuando se ha ejecutado correctamente un *server action*, nos permite actualizar la vista del usuario con los nuevos datos insertados sin necesidad de recargar la página o realizar otra petición.

Captura 31 – Usando revalidatePath cuando la subida ha tenido éxito. El nuevo archivo aparecerá como una nueva fila en vista de archivos sin necesidad de recargar la página.

```
81 export const uploadFileWithTransaction = async (trackId, formData) => {  
05   revalidatePath("/(protected)/track/[id]", "page");  
06 } catch (error) {  
07   const [status, message] = resolveError(error);  
08   return {success: false, status: status, message: message};  
09 }
```

5.2.3 SUPABASE

Los archivos guardados en la base de datos realmente son referencias a un archivo real guardado en otra parte. Desde el principio se descartó guardar los archivos como **BLOBS** en la base de datos, así como la idea de guardarlos en local. No sería escalable y realmente, tratándose de videos y audios, nos quedaríamos sin espacio rápidamente. La otra opción era subir los archivos a un servidor externo y guardar la referencia (*url*) y metadatos (instrumento, autor) en la base de datos.

Supabase resuelve este ofreciéndonos un cliente compatible con NextJS. Para ello debemos definir los **buckets** (carpetas) y definir las operaciones disponibles para esos buckets en la web. Tras este proceso, el desarrollador obtiene unas *keys* que deberá usar para instanciar el cliente y usar el servicio.

Captura 32 – Instanciando el cliente de Supabase y definiendo la función de subida de archivos.

```
4
5  const supabase = createClient(
6    process.env.SUPABASE_URL,
7    process.env.SUPABASE_KEY
8  );
9
10 export const supUploadFile = async (file, name) => {
11   const { data, error } = await supabase.storage
12     .from("files")
13     .upload(name, file);
14   return { data, error };
15 };
16
```

En la anterior captura del módulo `./lib/supabase` se puede ver el uso del cliente Supabase para subir un archivo binario (*file*) y el nombre asignado (*name*).

Por último, podemos invocar esta función desde cualquier controlador o *server action* y despreocuparnos del archivo. El cliente nos devuelve un objeto con las constantes **data** y **error**. Si *data* no está definida y *error* no es **null**, lanzamos excepción. Si *data* está definida, el archivo se subió correctamente:

Captura 33 – Acción encargada de recoger la información del archivo, guardarla en base de datos y subir el archivo al servidor de Supabase.

```
94  await db.$transaction(async (tx) => {
95    result = await createFileData(tx, fileToInsert);
96    const uploadInfo = await supUploadFile(file, name);
97    if (uploadInfo.error) {
98      let err = new Error(uploadInfo.error.message);
99      err.code = uploadInfo.error.statusCode;
100     throw err;
101   }
102 });
103 revalidatePath("/(protected)/track/[id]", "page");
104
```

En el fragmento de código anterior, se persiste la información del archivo (**línea 95**). Acto seguido, se sube el archivo mediante el cliente de *Supabase* (**línea 95**). Si la variable **error** está presente, lanzamos una excepción. Como las dos funciones se ejecutan dentro de una **transacción** (**línea 94**), la excepción impide que se confirme la operación en base de datos y evitamos así la existencia de archivos con referencias nulas.

Para obtener un archivo específico desde el servidor y descargarlo en el navegador, usamos la función nativa del cliente *Supabase* **download()**:

Captura 34 – Función download definida en el módulo supabase.js para obtener archivos.

```
51 export const supDownloadFile = async (name) => {  
52   const { data, error } = await supabase.storage.from("files").download(name);  
53   return { data, error };  
54 };
```

5.3 Cliente

5.3.1 AXIOS

Para la realización de llamadas asíncronas en el lado cliente se ha optado por Axios. Su elección se debe a la facilidad para encapsular lógica y reutilizar código.

La clase *fetching.js* en la raíz del proyecto gestiona todo lo relacionado con llamadas *HTTP*. Veamos cómo se ha implementado:

Captura 35 – Implementación de Axios. Inicialización del recurso.

```
1 import axios from "axios";  
2 const API = axios.create({  
3   baseURL: `${process.env.NEXT_PUBLIC_DOMAIN_URL}`,  
4   headers: {  
5     "Content-Type": "application/json",  
6   },  
7 });
```

En el código anterior, creamos una instancia de la clase Axios con parámetros comunes a todas las llamadas: el **dominio** al cuál hacer la petición, cargado desde una variable de entorno, y el **tipo de dato** esperado, en nuestro caso texto en formato JSON.

Después, para definir cualquier llamada a un *endpoint* de nuestra aplicación:

Captura 36- Función para obtener las tracks almacenadas en el sistema.

```
export const getAvailableTracks = (query) => API.get(`/api/track?query=${query}`).then((res) => res.data);
```

Como se puede apreciar, invocar a la instancia **API** nos permite definir llamadas HTTP en una sola línea sin repetir código.

Al devolver una promesa y no el resultado, nos permite gestionar el error directamente en el componente que invocó la función:

Captura 37 – Uso de la función `getAvailableTracks()`.

```

    try {
      const data = await getAvailableTracks(value);
      setExistingTracks(data);
    } catch (error) {
      setExistingTracks([]);
      setError(error.message)
    } finally {
      setLoading(false)
    }
  }, 250);
};

```

5.3.2 SHADCN UI

Shadcn es una librería de componentes ya estilizados y con cierta lógica interna pre-definida que se ha usado como complemento de los componentes propios.

La ventaja de *shdcn* es que **instala el componente en una carpeta especificada** (en nuestro caso, la carpeta ***ui*** dentro de ***./componentes***) y lo abre completamente a modificaciones, a diferencia de MUI. De este modo, nos permite a los desarrolladores tener un prototipo que ir modificando a nuestro gusto.

Captura 38 – Componente instalado por shadcn, accesible desde nuestra carpeta ui.

```

1  "use client"
2  import * as React from "react"
3  import * as AvatarPrimitive from "@radix-ui/react-avatar"
4  import { cn } from "@lib/Utils"
5
6  const Avatar = React.forwardRef(({ className, ...props }, ref) => (
7    <AvatarPrimitive.Root
8      ref={ref}
9      className={cn("relative flex h-10 w-10 shrink-0 overflow-hidden rounded-full", className)}
10     {...props} />
11  ))
12  Avatar.displayName = AvatarPrimitive.Root.displayName
13
14  const AvatarImage = React.forwardRef(({ className, ...props }, ref) => (
15    <AvatarPrimitive.Image
16      ref={ref}
17      className={cn("aspect-square h-full w-full", className)}
18     {...props} />
19  ))
20  AvatarImage.displayName = AvatarPrimitive.Image.displayName
21
22  const AvatarFallback = React.forwardRef(({ className, ...props }, ref) => (
23    <AvatarPrimitive.Fallback
24      ref={ref}
25      className={cn("flex h-full w-full items-center justify-center rounded-full bg-muted", className)}
26     {...props} />
27  ))
28  AvatarFallback.displayName = AvatarPrimitive.Fallback.displayName
29
30  export { Avatar, AvatarImage, AvatarFallback }

```

El código anterior muestra el componente **Avatar** instalado por *shadcn* en nuestra carpeta *ui*. Podemos invocarlo desde cualquier otro componente, pero también podemos entrar y modificarlo sin restricciones. Vemos que está estilizado con **TailwindCSS** (líneas 17 y 25), lo que facilita la edición manual.

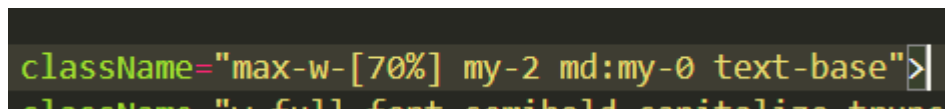
Los componentes de *shdcn* se han usado frecuentemente para completar fragmentos de interfaz. Ejemplo de ello son los **botones**, **menús desplegables**, **acordeones**, **diálogos**, **alertas y esqueletos**.

5.3.3 TAILWINDCSS

Para estilizar todos aquellos componentes propios que no forman parte de *shadcn*, se han usado los estilos en línea de **Tailwind**. Estos estilos son clases predefinidas que permiten asignar atributos CSS en la propia etiqueta **class** del elemento *HTML*.

Veamos un ejemplo de cómo hemos asignado clases usando Tailwind:

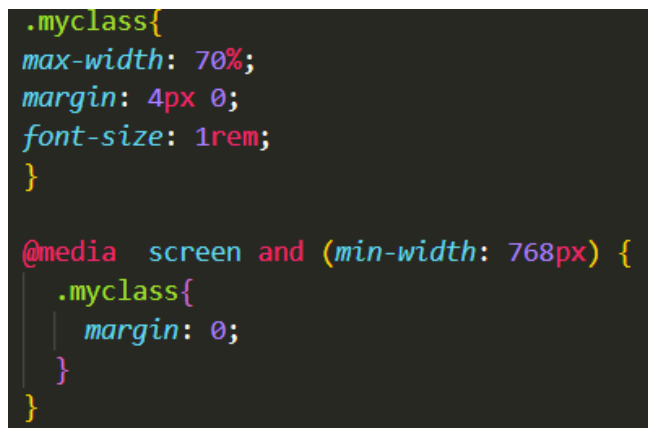
Captura 39 – Usando clases de Tailwind para estilizar un elemento.



```
className="max-w-[70%] my-2 md:my-0 text-base">
```

El anterior código en una línea corresponde al siguiente código en CSS tradicional:

Captura 40 – Equivalente en CCSS tradicional del código en la [captura 36](#).



```
.myclass{
  max-width: 70%;
  margin: 4px 0;
  font-size: 1rem;
}

@media screen and (min-width: 768px) {
  .myclass{
    margin: 0;
  }
}
```

El ahorro de código es notable. Además, *Tailwind* permite la inserción de valores arbitrarios si no se está conforme con alguno de sus valores predefinidos. Esto se hace usando las llaves ([**valor arbitrario**]) a continuación del atributo a modificar.

A nivel responsive, se ha usado los atributos de *Tailwind* para gestionar los distintos breakpoints: **sm**, **md**, **lg**, **xl** y **2xl**. De esta manera podemos dar estilos dinámicos añadiendo un prefijo a la asignación de la clase:

Captura 41 – Definiendo el ancho (w-*) del contenedor según el tamaño de la pantalla usando los prefijos adecuados.

```
sm:w-11/12 md:w-10/12 xl:w-4/6">
```

5.3.4 REACT HOOK FORM Y VALIDACIONES

Para el tratamiento de los formularios se ha implementado la librería nativa **react-hook-form**. Esta clase lleva un seguimiento de todos los campos del formulario para actualizarlos automáticamente sin necesidad de renderizar el formulario de nuevo. Basta con usar el **hook useForm** incluido en la librería y asociar el formulario **HMTL** con la instancia **form**.

Captura 42 – Uso del hook useForm.

```
76 const form = useForm({
```

Captura 43 – Asociando la instancia de useForm al formulario HTML

```
180 </div>
187 <Form {...form}>
188 <form
```

Con esto conseguimos que los **inputs** definidos dentro del formulario sean actualizados por la librería.

Además, **react-hook-form** permite la inclusión de librerías de validación. En nuestro caso se ha hecho uso de **Yup**. Yup permite definir esquemas de datos y aplicarlos a un formulario de React.

De este modo, los campos de formulario que coincidan con los nombres definidos en el esquema (email, password, name), se validarán con las restricciones especificadas dentro del método **shape()** (línea 18) antes de enviar los datos al servidor:

Captura 44 – Definiendo el esquema de datos para el registro del usuario en el archivo **schemas.js**.

```
16 export const RegisterSchema = yup
17   .object()
18   .shape({
19     email: yup.string().email().min(6).required(),
20     password: yup
21       .string()
22       .matches(/^(?=.*[A-Z])(?=.*[a-z])(?=.*\d){8,}$/)
23       .required(),
24     name: yup.string().min(3).required(),
25   })
26   .required();
```


Finalmente, en el propio formulario debemos importar el esquema del archivo correspondiente (en nuestro caso `schemas.js`) y establecer el resolver del hook `useForm` a dicho esquema (*línea 36*).

Captura 45 - Incluyendo un esquema de validación en el formulario de registro.

```
35   const form = useForm({
36     resolver: yupResolver(RegisterSchema),
37     defaultValues: {
38       email: "",
39       password: "",
40       name: "",
41     },
42   });
```

Cabe destacar que *Yup* permite validaciones personalizadas con los métodos `test()`, que se han usado para determinar si un archivo debería ser aceptado o no:

Captura 46 – Validaciones personalizadas para determinar si un archivo cumple los requisitos de subida.

```
98   file: yup
99     .mixed()
100    .required()
101    .test("is-valid-size", "Max allowed size is 4MB", (value) => {
102      return value && value.size <= MAX_FILE_SIZE;
103    })
104    .test(
105      "is-valid-extension",
106      "Allowed extension are mp3, mp4 and pdf",
107      (value) => {
108        return value && ALLOWED_TYPES.includes(value.type);
109      }
110    ),
```

5.3.4 SPOTIFY

Dado que el objetivo de la aplicación es crear *snapshots* musicales, era difícil limitar la libertad del usuario a la hora de crear un track. Debido a esto se decidió tomar como *fuentes de la verdad* las canciones disponibles en **Spotify**. Para ello, se tuvo que dar de alta la aplicación en el servicio de Spotify, lo que nos proporcionaría unas *keys* que se usarían en cada a la API. Después de leer la documentación oficial, se definieron un par de funciones que buscarían canciones y artistas en la plataforma.

Estas funciones siguen el mismo patrón que las llamadas *HTTP* a los *endpoints* de nuestra aplicación. Creamos dos instancias *Axios* (una para obtener el token de acceso y otra para obtener información con ese mismo token) y las usamos como base para definir funciones más específicas:

Captura 47 – Instancias de Axios para comunicarse con la API de Spotify.

```
9  ✓ const SPOTIFY_TOKEN = axios.create({
10    |   baseUrl: "https://accounts.spotify.com",
11    |   headers: {
12    |     "Content-Type": "application/x-www-form-urlencoded",
13    |   },
14    | });
15
16  ✓ const SPOTIFY_SEARCH = axios.create({
17    |   baseUrl: "https://api.spotify.com/v1/search?limit=10&market=ES&q=",
18    | });
19
```

Captura 48 – Llamadas a la API de Spotify para obtener token de acceso, buscar canciones y buscar artistas.

```
51  ✓ export const getAccessTokenSpotify = () =>
52  ✓   SPOTIFY_TOKEN.post("/api/token", {
53    |     grant_type: "client_credentials",
54    |     client_id: process.env.NEXT_PUBLIC_SPOTIFY_CLIENT_ID,
55    |     client_secret: process.env.NEXT_PUBLIC_SPOTIFY_CLIENT_SECRET,
56    |   }).then((response) => response.data);
57
58  ✓ export const searchTracksInSpotify = (query, token) =>
59  ✓   SPOTIFY_SEARCH.get(encodeURIComponent(query) + "&type=track", {
60  ✓   |   headers: {
61    |     |   Authorization: `Bearer ${token}`,
62    |   },
63    |   }).then((res) => res.data);
64
65  ✓ export const searchArtistsInSpotify = (query, token) =>
66  ✓   SPOTIFY_SEARCH.get(encodeURIComponent(query) + "&type=artist", {
67  ✓   |   headers: {
68    |     |   Authorization: `Bearer ${token}`,
69    |   },
70    |   }).then((res) => res.data);
```

A partir de aquí ya somos capaces de obtener información desde el cliente. El token se guardará en la sesión del navegador. Si éste no existe o está caducado, se obtendrá el token y se volverá a ejecutar la llamada inicial.

Por último, queda explicar el diseño del formulario y la **implicación** de la API de **Spotify** en él.

Pese a los millones de canciones de las cuáles la plataforma lleva un seguimiento, hay *performances* o conciertos en directo que no están disponibles. Por eso los campos del formulario están técnicamente abiertos al usuario, pero se le empuja a usar el buscador de Spotify:

Captura 49 – Captura del formulario de creación de tracks.

Como se aprecia en la anterior imagen, lo primero que llama la atención del usuario es el buscador. Además, se ha añadido un **tooltip** que informa al usuario de esta posibilidad. El objetivo de este diseño es que el usuario opte siempre por usar el buscador primero.

Si la plataforma devuelve una lista de canciones entre las cuáles se encuentra la que el usuario quería, *clickando* sobre la opción deseada se rellenará automáticamente los campos del formulario con esa información, y **estos quedarán desactivados** para evitar que se corrompa la información oficial proveniente de la API.

6. DESPLIEGUE DE LA APLICACIÓN

NextJS es, en última instancia, un framework de JS. Por ello, su despliegue ha sido bastante sencillo. **Vercel**, la creadora de NextJS, pone a disposición de usuarios registrados un servicio de despliegue automatizado y actualizado con cada commit nuevo en el repositorio asociado.

Los comandos para instanciar la aplicación fueron los siguientes:

1. **npm install**: este comando instala todas las librerías definidas en el archivo *package.json* y por tanto las que se usan a lo largo del proyecto. Una vez finalizada la *task*, se creará en la raíz del proyecto una carpeta *node_modules* con el código fuente necesario para usar las librerías.
2. **npx prisma generate**: este comando de la librería *Prisma* permite generar el cliente de Prisma que se instanciará para hacer la conexión a la base de datos.
3. **npm run build**: por último, se compilan todos los módulos JS, funciones y se generan las páginas estáticas, así como el CSS.
4. **npm run start**: este comando pone en marcha la aplicación.

De este modo ya tenemos la aplicación desplegada en cualquier parte.

Esta facilidad se debe a dos factores:

- **Nodejs** a diferencia de PHP, por ejemplo, no necesita de un servidor Apache para levantar un servidor. Con tener instalado nodejs basta.
- La base de datos **Neontech**: en el apartado de tecnologías utilizadas, se hablaba de *Neontech* como instancia de base de datos. Es un servicio remoto, cuya conexión se realiza mediante una *url*, usuario y contraseña. Como éstas son variables de entorno que se cargan en tiempo de compilación, se produce un desacoplamiento que permite desplegar una aplicación fullstack con 4 comandos.

Captura 50 – Ajustes de despliegue en el dashboard de Vercel.

The screenshot shows the 'Framework Preset' dropdown set to 'Next.js'. Below it, there are four configuration rows, each with a label, a text input field, and an 'Override' toggle switch.

Configuration	Value	Override
Build Command	npm run build	Enabled
Output Directory	Next.js default	Disabled
Install Command	npm install	Enabled
Development Command	next	Disabled

Captura 51 – Definiendo variables de entorno para cargarlas durante el despliegue.

The screenshot shows a list of environment variables defined in the Vercel dashboard. Each row includes a variable name, its usage (Development, Preview, Production), a visibility icon, a masked value, the date added, and a menu icon.

Variable Name	Usage	Visibility	Value	Added	Actions
NEXT_PUBLIC_DOMAIN_URL	Development, Preview, Production	Public	*****	Added 10d ago	...
DATABASE_URL	Development, Preview, Production	Private	*****	Added 10d ago	...
AUTH_SECRET	Development, Preview, Production	Private	*****	Added 10d ago	...
AUTH_TRUST_HOST	Development, Preview, Production	Private	*****	Added 10d ago	...
GOOGLE_CLIENT_ID	Development, Preview, Production	Private	*****	Added 10d ago	...

En cuanto al dominio, se ha registrado el nombre **scoreshare.org** en [Cloudfare](https://www.cloudflare.com/).

A continuación, se sobrescribe el dominio por defecto que proporciona Vercel con **scoreShare.org**:

Captura 52 – Asociando un nombre de dominio a la aplicación.

Domains

These domains are assigned to your Production Deployments. Optionally, a different [Git branch](#) or a [redirection](#) to another domain can be configured for each one.

Add

www.scoreshare.org Production Refresh Edit

✓ Valid Configuration ✓ Assigned to main

scoreshare.org Refresh Edit

✓ Valid Configuration ✓ Redirects to `www.scoreshare.org`

8. MANUAL DE USUARIO

Se ha incluido un tutorial opcional, paso a paso, acompañado de imágenes y estilizado en la propia página web (está en inglés, como toda la página). De este modo, los usuarios casuales que entren por primera vez podrán aprender a usar la aplicación.

Sin embargo, se ha considerado que el **propio tutorial** sustituye al manual de usuario *per se*. Por ello se comparte el siguiente enlace que redirige directamente al tutorial incluido en la página:

<https://www.scoreshare.org/dashboard/tutorial>

9. ANÁLISIS EMPRESARIAL

El desarrollo de la app ha tenido en mente la competencia y se ha diseñado un esquema **DAFO** para analizar su estado:

scoreShare



En cuanto a la viabilidad económica, se ha hecho un estudio del gasto que supondría mantener la aplicación durante un año para soportar un tráfico bajo-medio:

Objetivo	Gasto mensual
Servicio de despliegue (Vercel)	20€
Servicio de almacenamiento (Supabase)	25€
Técnico de mantenimiento (mantenimiento semanal)	752€
Alquiler del dominio en Clouflare	9,85€ (anual)
Publicidad en redes sociales (Instagram, webs de terceros)	
TOTAL ANUAL:	9573,85€

Para que la aplicación sea viable se implementaría un sistema de dos subscripciones, una gratuita y otra de pago:



La suscripción estándar tiene un precio bastante razonable y es mensual. Para que los gastos fueran compensados, se necesitaría un **mínimo de 2400 subscriptores** anuales. Teniendo en cuenta que la app está internacionalizada (idioma inglés) y no hay restricción por países, es una cifra bastante razonable.

9. CONCLUSIONES

A continuación, se expondrán las conclusiones a las que se ha llegado tras el desarrollo de la aplicación. Se planteará el grado de consecución de los objetivos, los problemas encontrados, posibles mejoras y sensaciones generales:

9.1 Grado de consecución de objetivos

Lo cierto es que, pese a haber desplegado una aplicación que cumple con el objetivo principal inicial, que era el de centralizar recursos mediante *snapshots* musicales y poder filtrarlos por tipo e instrumento, no se ha implementado por falta de tiempo la funcionalidad social que hubiera completado el cuadro: **solicitudes de amistades** y **un sistema colaborativo** entre usuarios para modificar información de las *tracks* mediante ***editRequests*** (similar a un merge request de GitHub). El usuario creador podría comparar la información introducida por él con la *editRequest* entrante de otro usuario y seleccionar qué información persistir. Esto sería útil cuando el creador del *track* no acierta inicialmente con los campos de ***key***, ***tempo*** o ***source***.

9.2 Posibles mejoras

Aparte de las ya comentadas en el punto anterior, una posible mejora sería admitir otro tipo de archivos específicos del mundo musical como **MIDI** o **.mscz** y poder reproducirlos en directo con una barra de tiempo.

9.3 Problemas encontrados

El problema de mayor envergadura ha sido el uso del ORM **Prisma**. Si bien es cierto que su sintaxis reduce las sentencias SQL y las consultas son más intuitivas, carece, a 29 de mayo de 2024, de ciertas funcionalidades que se presuponen básicas para el acceso a datos:

- Ausencia de método ***exists*** y ***notExists***. Esto nos ha forzado a obtener un recurso entero cuando sólo necesitábamos saber de su existencia.
- Nula compatibilidad con **triggers**: a pesar de haber definido dos *triggers* y dos funciones, y de haberlas testado manualmente contra la base de datos, Prisma **no soporta** los *triggers*, por lo que levanta un error.
- Restringe el uso de la función **SUM()** a consultas agrupadas: para hacer una consulta agrupada en Prisma, se usa el método ***.groupBy()***. El problema es que nos obliga a realizar dos consultas separadas al no poder incorporar la función *SUM()* dentro de un *.findMany()*.

El uso de Prisma en el futuro queda en el aire en vistas a que implementen las funcionalidades descritas.

La otra gran dificultad tiene que ver con el formulario de creación de **Track**. Es un formulario bastante complejo y dinámico. El campo *image*, por ejemplo, admite tanto un **File** (si el usuario crea el *track* por su cuenta) y su posterior validación (tamaño, tipo) como una *string* (la *url* de la imagen de Spotify). Dependiendo del tipo de dato, el controlador guardará la imagen y recuperará la *url* o no.

Además, al obligar¹ al usuario a subir un archivo con la creación del *track* realmente se validan dos formularios en uno, y se debe realizar todo en la misma transacción.

9.4 Sensaciones generales

Las sensaciones finales son positivas. NextJS es un framework muy polivalente que nos ha permitido incluir muchas funcionalidades interesantes, mientras que React es un framework en el cuál siempre nos hemos sentido muy cómodos desarrollando.

En general el diseño de la página ha ido evolucionando hasta ofrecer un mensaje moderno y dinámico. Gracias al descubrimiento de **shadcn** y **Tailwind** hemos podido diseñar una interfaz de usuario limpia coherente. La conclusión es que los módulos impartidos durante el grado se han implementado satisfactoriamente.

¹ esta regla de negocio se implementa para no tener **Tracks** sin archivos vinculados y que, por tanto, no aportan valor a la aplicación

ANEXO

Principales colores usados en la página:



F8FAFC



10B981



155E75



E0E7FF



4F46E5



C026D3



0F172A

Diseño de los logos (diseño propio):



ÍNDICE DE IMÁGENES

Captura 1 – Diagrama ER, primera aproximación	10
Captura2 - Modelo ER actual	11
Captura 3 – Diagrama ER actual	12
Captura 4 – Restricción única de los campos <code>userId</code> , <code>name</code> y <code>instrument</code> .	13
Captura 5 - Restricción única en la tabla "VoteInComment".	13
Captura 6 – Estructura de directorios del proyecto	19
Captura 7 – Ejemplo de anidación de carpetas para crear rutas anidadas	20
Captura 8 – Url que el usuario visualiza en el navegador.	20
Captura 9 – Archivo layout en la raíz de la aplicación	20
Captura 10 - Contenido del archivo layout.js en la raíz de la aplicación	21
Captura 11 - Atributos CSS globales.	21
Captura 12 - Función CREATE para insertar un archivo en la BBDD.	21
Captura 13 - Ejemplo de definición de hook personalizado.	22
Captura 14 – Uso del hook personalizado definido en la Captura 13	22
Captura 15 - Archivos en la carpeta lib y ejemplo de función dentro de utils.	22
Captura 16 - Definición de tablas mediante la sintaxis de Prisma	23
Captura 17 - Comando de Prisma para actualizar la base de datos.	23
Captura 18 - Ejemplo de consulta a base de datos mediante Prisma.	24
Captura 19 - Uso de JOINS para obtener datos de archivo y usuario al mismo tiempo.	24
Captura 20 - Ejemplo de definición de un API endpoint.	25
Captura 21 - Ejemplo para manejar peticiones en un archivo route.js.	25
Captura 22 - Server action addComment: Insertar/actualizar un comentario en la base de datos.	26
Captura 23 - Información mostrada al usuario obtenida mediante componente servidor.	27
Captura 24 - TrackInfo: Ejemplo de un componente servidor accediendo a datos directamente.	28
Captura 25 - Uso de Suspense para mostrar una vista diferente mientras se carga el componente.	28
Captura 26 - Vista de un esqueleto mientras se carga el componente.	28
Captura 27 – Opciones de registro para el usuario.	29
Captura 28 – Definición de los proveedores de autenticación en el archivo auth.config.js.	29
Captura 29 – Invocamos el método signIn de la librería de authjs.	30
Captura 30 – Usando revalidatePath.	30
Captura 31 – Instanciando el cliente de Supabase y definiendo la función de subida de archivos.	31
Captura 33 – Función download definida en el módulo supabase.js para obtener archivos.	32
Captura 34 – Implementación de Axios. Inicialización del recurso.	32
Captura 35- Función para obtener las tracks almacenadas en el sistema.	32
Captura 36 – Uso de la función mostrada anteriormente en un component de búsqueda.	33
Captura 39 – Usando clases de Tailwind para estilizar un elemento.	34
Captura 40 – Equivalente en CCSS tradicional del código en la captura 36 .	34
Captura 41 – Definiendo atributos responsive con Tailwind.	35
Captura 42 – Uso del hook useForm.	35
Captura 43 – Asociando la instancia de form al formulario HTML	35
Captura 44 – Definiendo el esquema de datos para el registro del usuario en el archivo schemas.js .	35
Estableciendo el resolver del hook useForm a un esquema Yup(línea 36).	36

<i>Captura 45 - Incluyendo un esquema de validación en el formulario de registro.</i>	<i>36</i>
<i>Captura 47 – Instancias de Axios para comunicarse con la API de Spotify.</i>	<i>37</i>
<i>Captura 48 – Llamadas a la API de Spotify para obtener token de acceso, buscar canciones y buscar artistas.</i>	<i>37</i>
<i>Captura 49 – Captura del formulario de creación de tracks.</i>	<i>38</i>
<i>Captura 50 – Ajustes de despliegue en el dashboard e Vercel.</i>	<i>39</i>
<i>Captura 51 - Captura 54 – Definiendo variables de entorno para cargarlas durante el despliegue.</i>	<i>39</i>

**Las capturas 49, 50 y 51 pertenecen a la página web de Vercel. Copyright: Vercel*

GITHUB

A continuación, se indica el enlace al repositorio de GitHub del proyecto, en el grupo de proyectos de IES La Vereda:

https://github.com/ieslavereda-projects/23_24_Gabriel_Enguidanos_Web_Musica

Al acceder al repositorio, aparecerá una carpeta con el DDL tanto en sintaxis PostgreSQL como MySQL, además de una captura del modelo ER.

La carpeta scoreShare contiene el código de la aplicación.

Se incluye un archivo README con los pasos mínimos para instalar y levantar la aplicación.

También se incluye una copia de la presente memoria en formato PDF.

BIBLIOGRAFÍA

1. *Learn Next.js*: <https://nextjs.org/learn>
2. *Programmatically downloading files in the browser*: <https://blog.logrocket.com/programmatically-downloading-files-browser/>
3. *Setup & configuration*: <https://www.prisma.io/docs/orm/prisma-client/setup-and-configuration>
4. *One to many relationships*: <https://www.prisma.io/docs/orm/prisma-schema/data-model/relations/one-to-many-relations>.
5. *CRUD (reference)*: <https://www.prisma.io/docs/orm/prisma-client/queries/crud>
6. *JavaScript Client Library*: <https://supabase.com/docs/reference/javascript/>
7. *Getting started with Tailwind CSS*: <https://v2.tailwindcss.com/docs>
8. *Web API (Spotify)*: <https://developer.spotify.com/documentation/web-api>
9. *Install and configure Next.js*: <https://ui.shadcn.com/docs/installation/next>
10. *Routing - Route Handlers*: <https://nextjs.org/docs/app/building-your-application/routing/route-handlers>.
11. *How can I get query string from URL in NextJS?*: <https://stackoverflow.com/questions/43862600/how-can-i-get-query-string-parameters-from-the-url-in-next-js>
12. *React Hook Form*: <https://claritydev.net/blog/react-hook-form-multipart-form-data-file-uploads>.
13. *Handling formData in NexJS*: <https://dev.to/divinenaman/intro-to-nextjs-apis-and-handling-form-data-1ola>.
14. *Controller in React forms*: <https://8design.medium.com/controller-in-react-hook-form-how-to-use-it-b0594a580fda>.
15. *Getting started with Yup*: <https://www.npmjs.com/package/yup>.
16. *Yup: Validating type and size*: <https://medium.com/webisora/yup-validating-file-type-and-size-81cd26ca7dfb>
17. *AuthJS implementacion example*: <https://www.youtube.com/watch?v=1MTyCvS05V4>.
18. *NextJS tutorial*: https://www.youtube.com/watch?v=ZjAgaIC_3c&list=PLC3y8-rFHvwjOKd6gdf4QtV1uYNiQnrul
19. *Prisma Adapter for AuthJS*: <https://next-auth.js.org/v3/adapters/prisma>.
20. *How to create a full screen video background*: https://www.w3schools.com/howto/howto_css_fullscreen_video.asp
21. *Imagen de fondo obtenida de la siguiente web*: <https://www.svgbackgrounds.com/set/free-svg-backgrounds-and-patterns/>.