Imagen borrosa de una persona

Descripción generada automáticamente con confianza media

scoreShare

GRADO EN DESARROLLO DE APLICACIONES WEB

Imagen que contiene Patrón de fondo

Descripción generada automáticamente

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

[Información esencial: 1](#_Toc167711544)

[Informació essencial: 1](#_Toc167711545)

[Essential information: 1](#_Toc167711546)

[1.Presentación del proyecto 4](#_Toc167711547)

[1.1. Introducción al proyecto 4](#_Toc167711548)

[1.1. Módulos que implica 5](#_Toc167711549)

[1.3 Objetivos iniciales del proyecto 7](#_Toc167711550)

[1.4 Dificultades que debía afrontar desde el inicio. 8](#_Toc167711551)

[3. Diseño de la aplicación. 9](#_Toc167711552)

[3.1 Base de datos. Entidad-relación y modelo relacional 9](#_Toc167711553)

[4. Tecnologías y recursos empleados 13](#_Toc167711554)

[4.1 Recursos hardware 13](#_Toc167711555)

[4.2 Recursos software 13](#_Toc167711556)

[4.2.1 Herramientas 13](#_Toc167711557)

[4.2.2 Programas, frameworks y lenguajes 14](#_Toc167711558)

[5. Desarrollo del proyecto 17](#_Toc167711559)

[5.1 Estructura del proyecto 17](#_Toc167711560)

[5.2 Servidor 21](#_Toc167711561)

[5.2.1 Prisma JS 21](#_Toc167711562)

[5.2.2 NextJS 23](#_Toc167711563)

# 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 ojearlo?

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, irrepetible. 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 *dummy* code, 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, uso responsive, uso adecuado del color y mantener un estilo coherente a lo largo de la página me han permitido darle un aspecto moderno y estético al frontal. |
| Despliegue de aplicaciones | Uso de variables de entorno para que el despliegue sea más desacoplado. |

## 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 archivos.

Listar, editar o eliminar archivos.

Descargar archivos subidos por otros usuarios.

Comentar y responder a otros comentarios.

Buscar canciones de manera sencilla.

Comunicarse con otros usuarios mediante mensajería instantánea.

Votar en comentarios y archivos.

Relacionarse con otros usuarios mediante solicitudes de amistad.

Recibir notificaciones.

## 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 AppleMusic, AmazonMusic o Spotify?

# 3. Diseño de la aplicación.

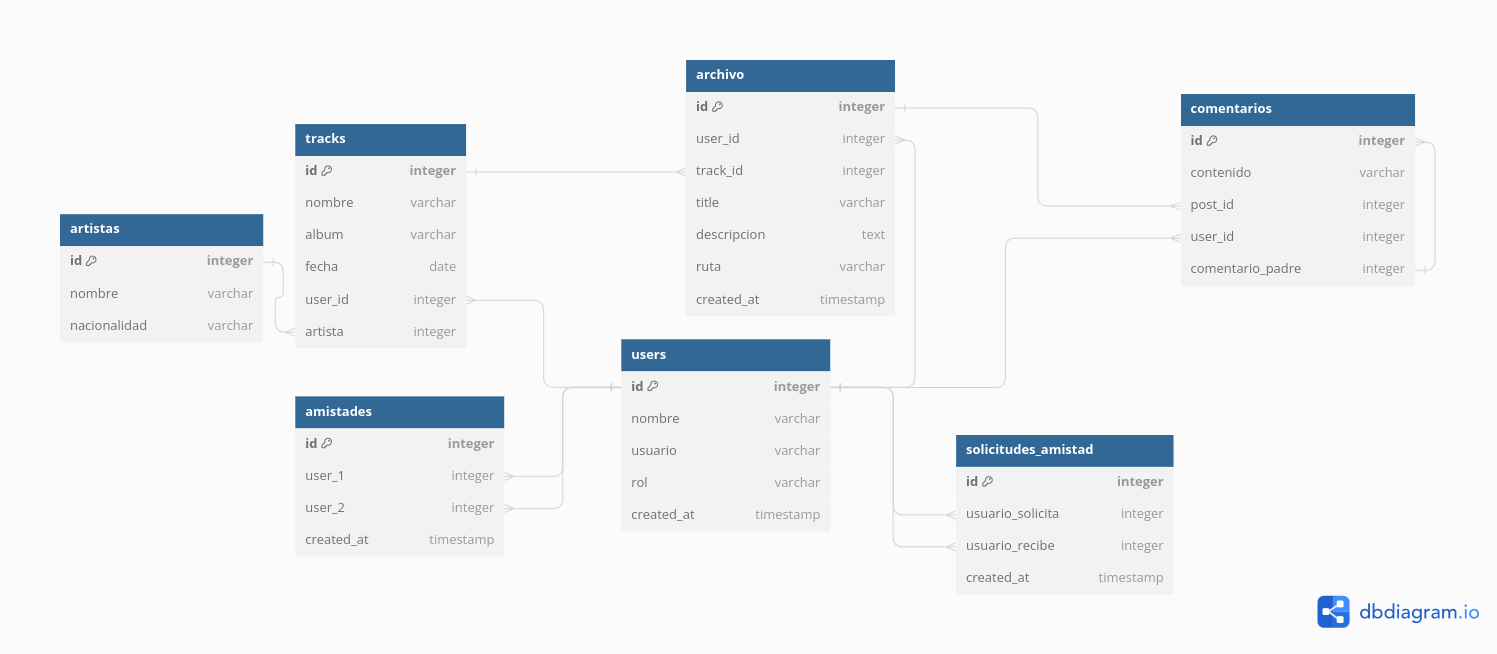
## 3.1 Base de datos. Entidad-relación y modelo relacional

Como motor de BBDD relacional he 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 del cuál hablaré más adelante.

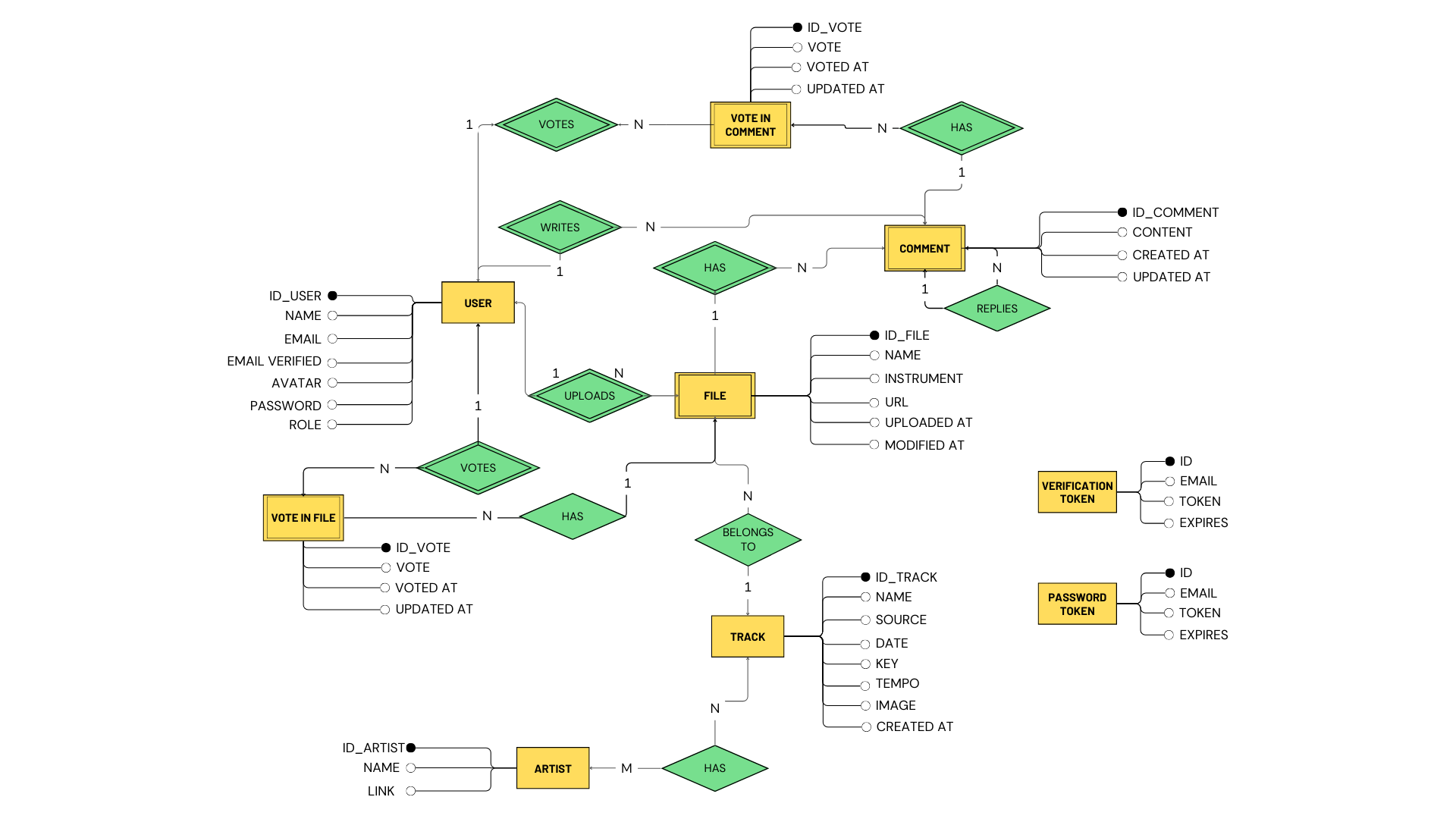
A continuación, se muestra el primer diagrama entidad-relación diseñado antes de iniciar el desarrollo con la herramienta online **dbDiagram**.

Captura – 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:

Captura - 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.

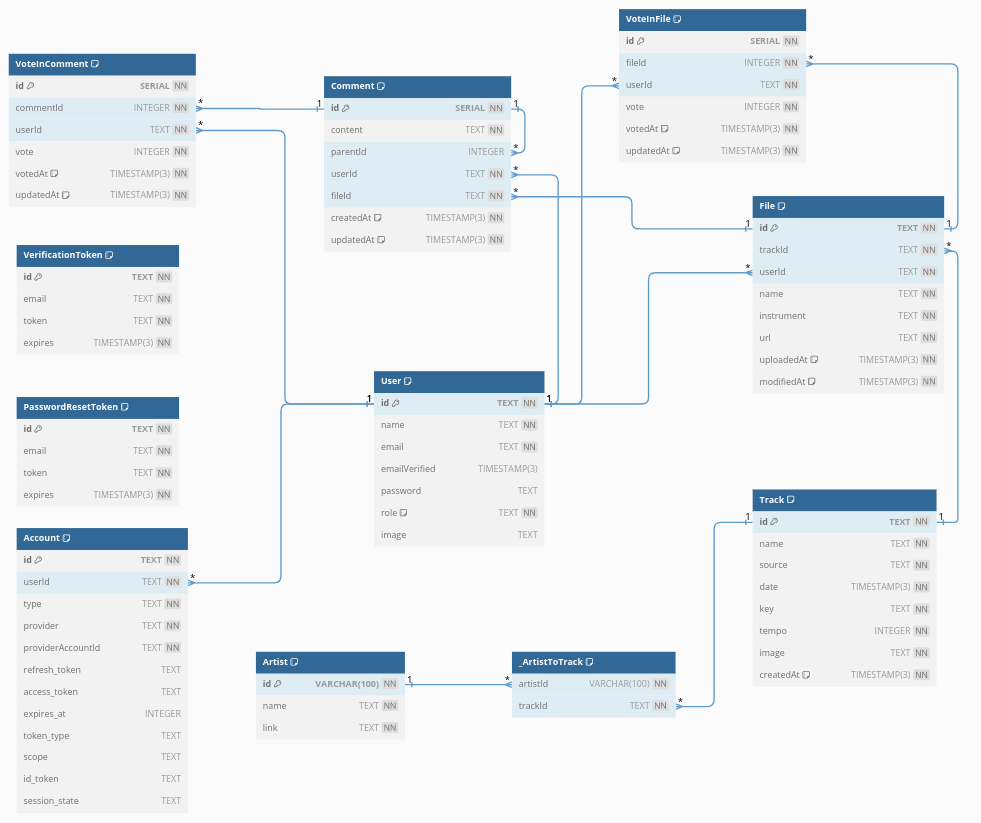
Otra diferencia radica en la inclusión de las tablas *vote\_in\_comment* y *vote\_in\_file* para llevar una cuenta de los votos de cada usuario en cada comentario y archivo*, y verification\_token* y *password\_token* para tareas de recuperación de contraseña y verificación de email*.*

Además, en la imagen anterior ([Figura 2](#figura2)) 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.](https://authjs.dev/getting-started/adapters/prisma)

Se ha considerado no incluirla debido a que no es fruto de una conclusión propia, si no que el propio *framework* obliga a implementar dicha tabla para el correcto funcionamiento de *Authjs*.

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

Captura – Diagrama ER actual



**Algunos apuntes sobre el diagrama anterior (**[**Captura 3**](#captura3)**):**

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



* Los usuarios podrán comentar un **File**, y cada comentario podrá ser respondido, y también la propia respuesta. Por 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 lo 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 ésta 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 -– Restricción única en la tabla “VoteInComment” para evitar que un usuario vote más de una vez.



# 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 resolición de dudas.
* dbDiagram para realizar los diagramas ER.
* Canvas para realizar el modelo ER.

### 4.2.2 Programas, frameworks y lenguajes

|  |  |
| --- | --- |
| **Tecnología** | **Descripción** |
| **Nextjs** | Nextjs es un framework **Node.js** fullstack de React, que permite el renderizado de componentes en el servidor y en el cliente.  Al ser fullstack, permite la definición de APIs, middleware, rutas, servicio de autenticación y conexiones con base de datos. |
| **NeonTech** | Se definen como un Postgres Serverless.  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. |
| **Prisma ORM** | 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.  Prisma está totalmente integrado con JS y Postgres. |
| **AXIOS** | **Axios** es un cliente de JavaScript utilizado principalmente para realizar y manejar peticiones asíncronas. |
| **Shadcn UI** | **Shadcn** es una librería de componentes basados en TailwindCSS. Sin embargo, a diferencia de **MUI**, que instala los componentes en *node\_modules* y está cerrada a grandes modificaciones, **shadcn** instala el componente ya estilizado en una carpeta autogenerada a la que puedes acceder como si se tratase de un componente propio y personalizarlo infinitamente. |
| **TailwindCSS** | 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. |
| **React** | **React** es un framework de JS, y a su vez es la base del framework **NextJS**.  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. |
| **Supabase** | Supabase es un *backend as a service.* 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. |

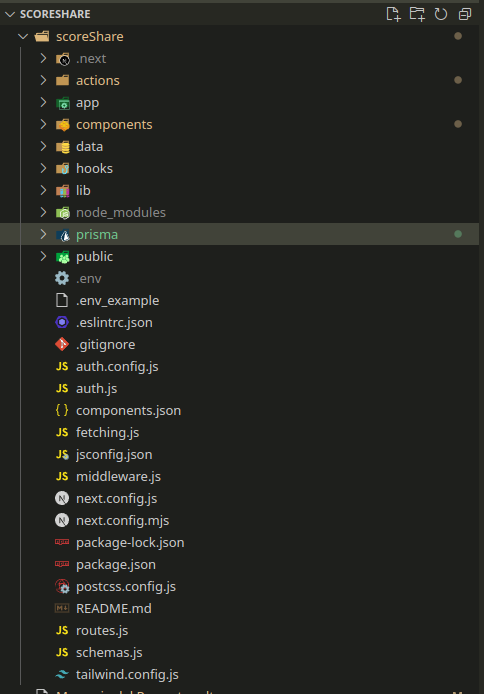
Como se puede apreciar, ninguna de las tecnologías, excepto **Axios y React**, se han visto durante el temario del curso. Si bien su aprendizaje ha lastrado el ritmo del desarrollo, la facilidad con la que resuelven los problemas compensa todo ese tiempo invertido, excepto algún caso concreto (hablaremos 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. Tiene ciertas peculiaridades que comentaremos a continuación, pero antes se mostrará una captura de la estructura de directorios de la aplicación:

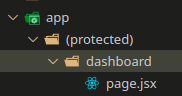
Captura – Estructura de directorios del proyecto



Obviando las carpetas autogeneradas *.next, node\_modules* y los archivos de configuración autogenerados por **Nextjs** como *.eslintsrc, next.config.\** y *jsconfig.json*, se dará una pequeña explicación de cada carpeta y archivo que componen este 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 HTTPS al uso. Es una funcionalidad nativa del framework.
* **./*app***: en esta carpeta se definen las rutas y las páginas. Aquí es importante seguir la convención de nombres del propio framework. 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 renderize, debemos incluir un archivo***page.jsx*** (obligatorio este nombre).

Captura – 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*. Lo normal es que en ruta del navegador viéramos **www.scoreshare.org/protected/dashboard**, pero si vamos al navegador:

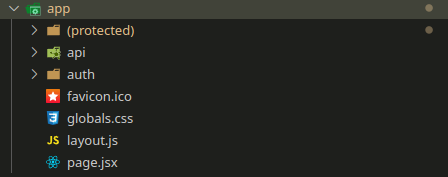
Captura – Url que el usuario visualiza en el navegador tras el ejemplo [**anterior**](#figura7)



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:**

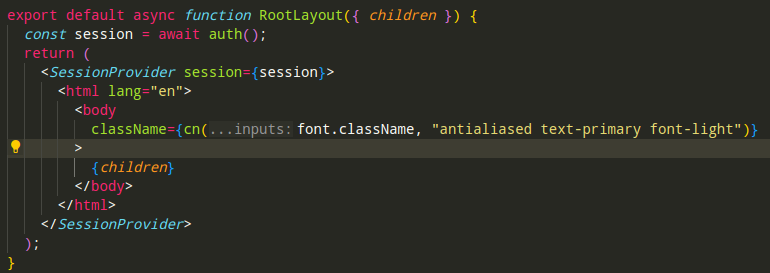
Captura – Archivo layout en la raíz de la aplicación



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:

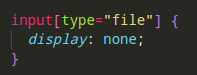
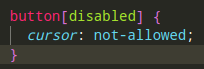
Captura - Contenido del archivo layout.js en la raíz de la aplicación



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.

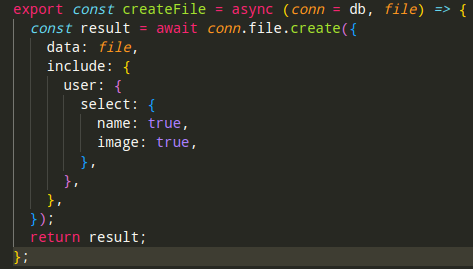
* También podemos ver el archivo **globals.css**. A pesar de que TailwindCSS es el encargado de estilizar los componentes y páginas mediante clases directamente en las etiquetas HTML, hay ciertos atributos que abarcan tantos componentes que es óptimo incluirlos en un archivo común, como por ejemplo:

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

* ***./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 - Función CREATE para insertar un archivo en la BBDD.



* ***./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 - Ejemplo de definición de hook personalizado.

Una captura de pantalla de un celular

Descripción generada automáticamente con confianza media

Y en cada formulario debo incluir lo siguiente:

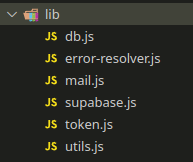
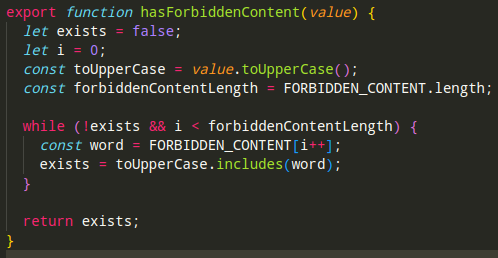
Captura – Uso del hook personalizado definido en la [**Captura 13**](#captura13)



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

* ***./lib:*** esta carpeta contiene todas las funciones específicas para realizar ciertas operaciones que no se engloban dentro de un componente o un repositorio, 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 - Archivos en la carpeta lib y ejemplo de función dentro de utils.

* ***./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 **Artists**:

Captura - Definición de tablas mediante la sintaxis de Prisma



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 - Comando de Prisma para actualizar la base de datos.

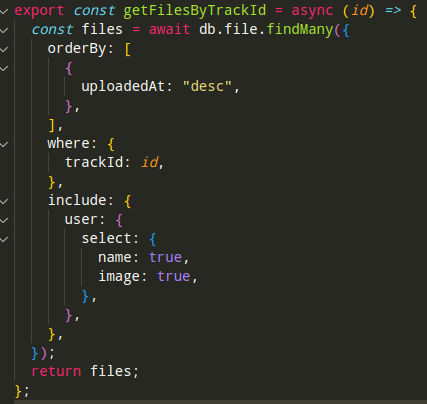


Para realizar operaciones CRUD, Prisma define una serie de convenciones.

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:

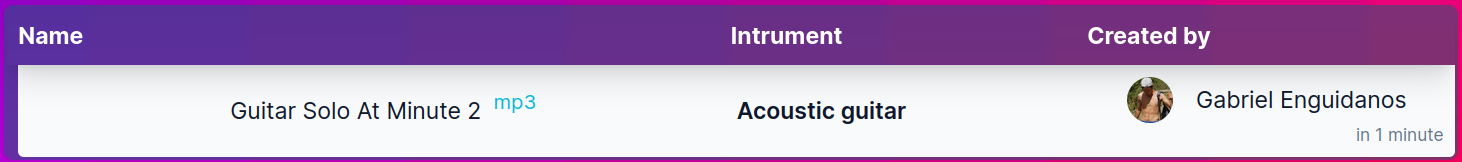
Captura - Ejemplo de consulta a base de datos mediante Prisma.



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



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](#figura7)) y la opción de definir *server actions* y APIs.

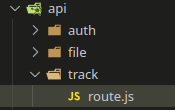
Con tal de poner en práctica lo aprendido durante el curso, se han definido algunos **endpoints** que reciben peticiones y devuelven respuestas *HTTP*.

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:**

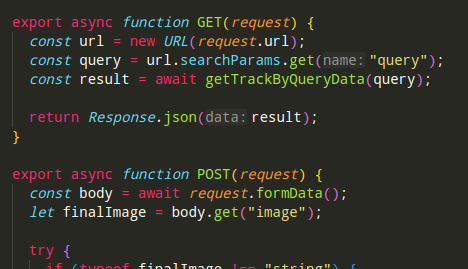
Para definir un endpoint que reciba peticiones *HTTP,* en NextJS debemos especificar una carpeta llamada ***api*** (este nombre debe ser exacto ya que forma parte de la convención de nombres). Dentro podemos concretar cuántas carpetas queramos, cada una referenciando una ruta anidada ([ver ejemplo](#figura7)), y finalmente debemos definir el archivo encargado de manejar las peticiones, ***route.js*** (nombre obligatorio).

Captura - Ejemplo de definición de un API endpoint.



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

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



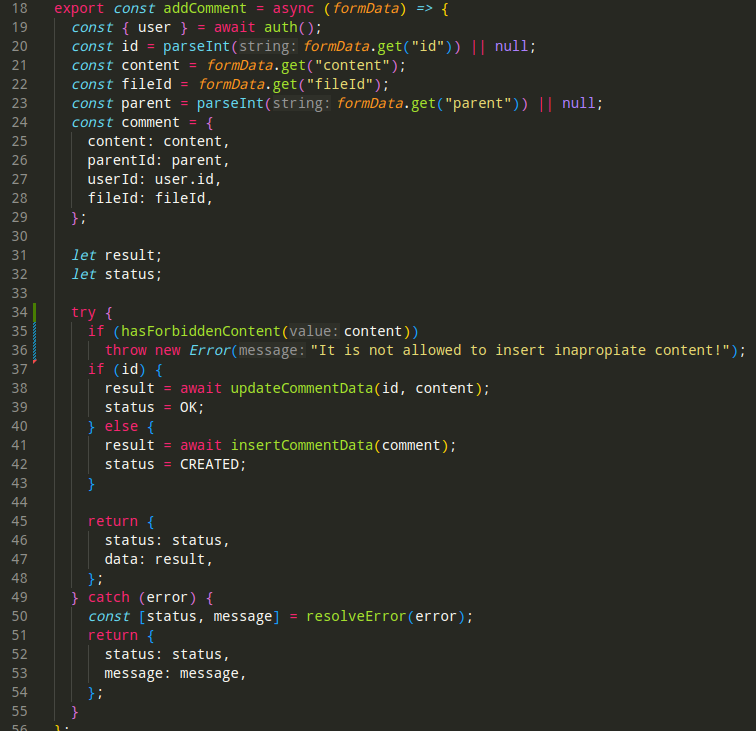
El **endpoint** quedaría de este modo: **www.scoreshare.org/api/track/.**

##### **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** y se pueden llamar de 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 - Server action addComment: Insertar/actualizar un comentario en la base de datos.

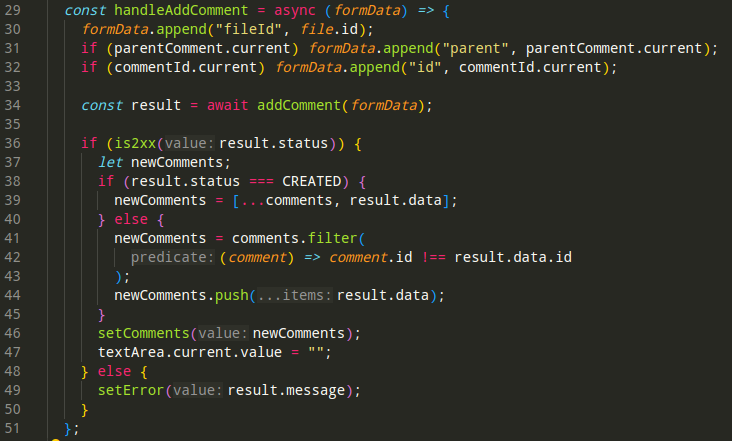


La función anterior se encarga de validar un comentario e insertarlo en la base de datos. Sería el análogo a un **POST** en *api/comment/.*

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***.

Como se ha comentado antes, se han definido unas contantes para simular la respuesta *HTTP* de una API. 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*



En la captura anterior se muestra como el componente cliente reacciona al click del usuario. En las líneas **30** a **32** se recoge la información del formulario y se envía al servidor mediante el *server action* ***addComment.***

Los códigos de estado predefinidos aquí hacen un gran papel, ya que 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 se reemplazará.

##### **SERVER COMPONENTS:**

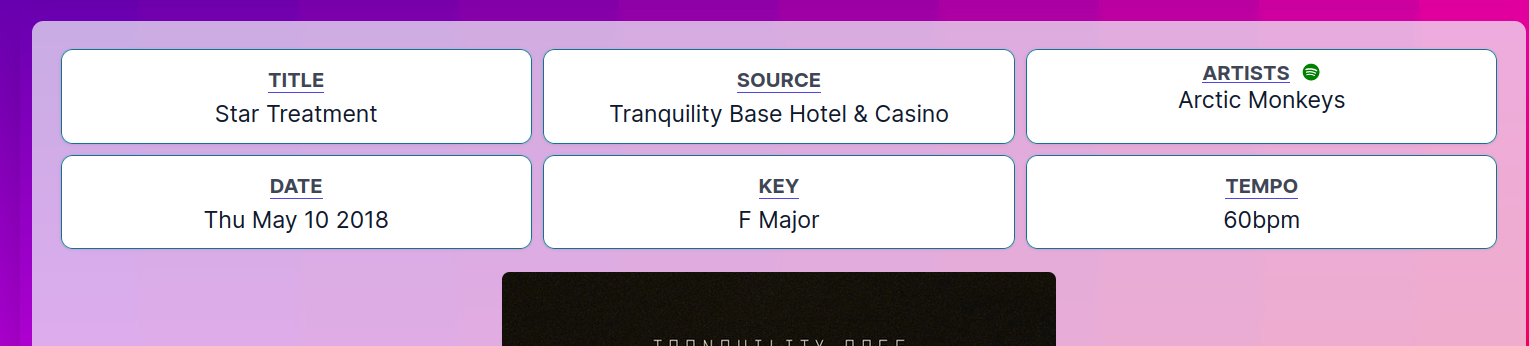
Cabe que recalcar que **NextJS** es capaz de renderizar **dos tipos** de componentes:

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.

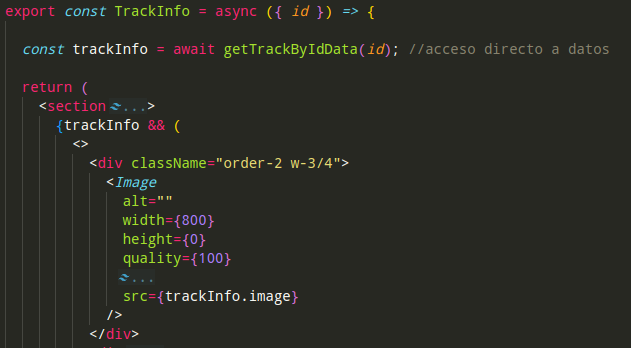
Una implementación de un componente servidor es la página dónde el usuario puede ver la información de un *track:*

Captura - Información mostrada al usuario obtenida mediante componente servidor.



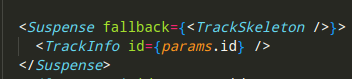
Debido a que esta información no va a cambiar de estado ni el usuario va a interactuar con ella, podemos obtenerla directamente desde el servidor, que nos devolverá (**return**) la tabla ya renderizada una vez se haya cargado.

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



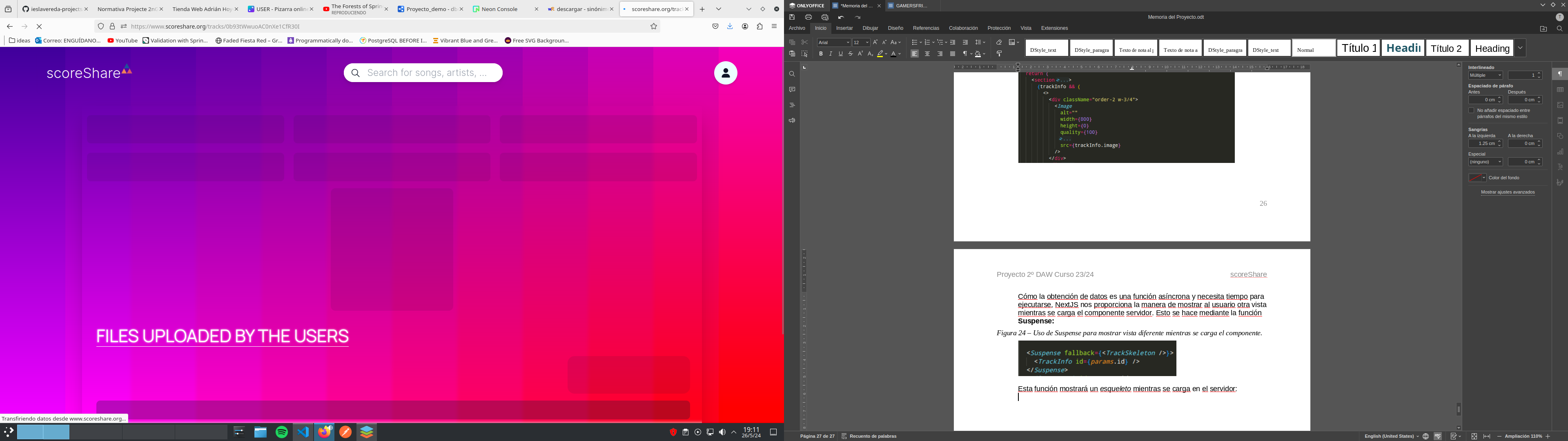
Cómo la obtención de datos es una función asíncrona y necesita tiempo para ejecutarse, NextJS nos proporciona una manera de mostrar al usuario otra vista mientras se carga el componente. Esto se hace mediante la función **Suspense:**

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



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

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



##### **AUTHJS:**