



Scout Arg (TA044)
Integración de Bases de Datos Relacionales y NoSQL
Segundo Cuatrimestre de 2024
Base de Datos (TA044)

Alumno	Padrón	Mail
Theo Lijs	109472	tlijs@fi.uba.ar
Juan Martin de la Cruz	109588	jdelacruz@fi.uba.ar
Franco Ariel Alani	111147	falani@fi.uba.ar
Franco Martin Fusco	102692	ffusco@fi.uba.ar
Benicio Braunstein	110126	bbraunstein@fi.uba.ar
Joaquin Batemarco	110222	jbatemarco@fi.uba.ar

Índice

1. Elección de las Tecnologías	3
1.1. Backend: Node.js + NestJS	3
1.2. Frontend: React.js	3
1.3. Bases de Datos: PostgreSQL y Firebase	3
1.3.1. PostgreSQL	3
1.3.2. Firebase	3
2. Diagrama de Arquitectura	4
3. Configuración y Conexión a las Bases de Datos	4
3.1. Configuración y conexión con SQL	4
3.2. Conexión a Firestore con Firebase	5
3.3. Uso de Variables de Entorno	6
4. Operaciones CRUD y datos	6
4.1. Datos en SQL	6
4.2. Tipos de Usuarios	7
4.2.1. Teams	7
4.2.2. Players	11
4.2.3. Users	14
4.2.4. Implementación de las operaciones en PostgreSQL	14
4.3. Datos en Firestore	16
4.3.1. Opinions	18
4.3.2. Comentarios	20
4.3.3. Implementación de las operaciones en Firestore	22
5. Comparación entre Bases de Datos Relacionales y NoSQL	23
5.1. Ventajas y Desventajas	25
5.2. Conclusión sobre los diferentes casos de uso	25
6. Dificultades Principales y Soluciones Implementadas	26
6.1. Integración entre PostgreSQL y Firestore	26
6.2. Manejo de Datos Relacionales en Firestore	27
6.3. Gestión de Usuarios y Seguridad	27
6.4. Escalabilidad y Eficiencia	27

7. Conclusiones	28
8. Aprendizajes	28
9. Reflexión Final	29

1. Elección de las Tecnologías

1.1. Backend: Node.js + NestJS

La elección de Node.js junto con NestJS para el backend está fundamentada en las siguientes razones:

- **Node.js:** Ofrece un entorno eficiente para manejar aplicaciones que requieren alta concurrencia, gracias a su arquitectura basada en un modelo de eventos y su capacidad de manejo sincrónico de operaciones. Esto lo hace ideal para un sistema que gestione múltiples usuarios simultáneamente.
- **NestJS:** Es un framework progresivo que utiliza TypeScript, lo cual aporta robustez al desarrollo gracias a su tipado estático. Además, proporciona una arquitectura modular y orientada a controladores que facilita la escalabilidad y el mantenimiento de la aplicación.
- La integración de NestJS con bases de datos como PostgreSQL y Firebase es directa, gracias a herramientas como Prisma y Firebase SDK. Esto simplifica la gestión de múltiples orígenes de datos en el sistema.

1.2. Frontend: React.js

React.js es una de las librerías más populares para el desarrollo frontend. Su elección se debe a:

- **Componentización:** Permite dividir la interfaz en componentes reutilizables, facilitando el mantenimiento y escalabilidad del sistema.
- **Virtual DOM:** Mejora significativamente el rendimiento de la aplicación, especialmente en aplicaciones interactivas con datos en tiempo real como comentarios y opiniones.

1.3. Bases de Datos: PostgreSQL y Firebase

La integración de PostgreSQL y Firebase combina las fortalezas de una base de datos relacional y una solución NoSQL en tiempo real, ofreciendo una arquitectura flexible y eficiente para satisfacer diferentes necesidades del proyecto. A continuación, se detallan las características y el rol específico de cada tecnología en el sistema.

1.3.1. PostgreSQL

PostgreSQL se utilizó para el manejo de usuarios, equipos y jugadores. En el caso de jugadores, equipos y usuarios, estos datos tienen relaciones bien definidas (por ejemplo, jugadores pertenecientes a equipos), lo que hace que PostgreSQL sea la opción más adecuada para gestionar estas relaciones mediante claves primarias y foráneas. Además, se eligió por las siguientes razones:

- Es una base de datos relacional robusta y altamente escalable que se adapta perfectamente al modelado de datos estructurados, como los jugadores, equipos y usuarios.
- Soporta consultas complejas y transacciones, asegurando la consistencia de los datos y las relaciones entre entidades.
- Ofrece extensiones avanzadas, como índices personalizados y JSON, útiles para consultas complejas.

1.3.2. Firebase

Firebase se utilizó para los comentarios y opiniones debido a su naturaleza más flexible y menos estructurada.

- Ideal para datos no estructurados o semi-estructurados, como opiniones y comentarios, permitiendo acceso en tiempo real y sincronización fluida entre usuarios.
- Su arquitectura en la nube facilita la escalabilidad dinámica y el manejo de eventos en tiempo real.
- Proporciona características adicionales, como autenticación y almacenamiento, que pueden integrarse para mejorar la experiencia del usuario. Como por ejemplo el firebase storage.

2. Diagrama de Arquitectura

En el siguiente diagrama se pueden observar cómo se comunican las diferentes partes de la aplicación.

El cliente/host se comunica a través del backend para acceder/modificar/crear y borrar los recursos que se encuentran en las bases de datos.

Notar que el cliente se comunica directo con el sistema de imágenes para poder cargarlas directamente (no están las imágenes en la base de datos, sino que se encuentran en el sistema de storage).

La base de datos PostgreSQL se encuentra hospedada en el servicio Supabase, mientras que Firestore se encuentra hospedada en Firebase.

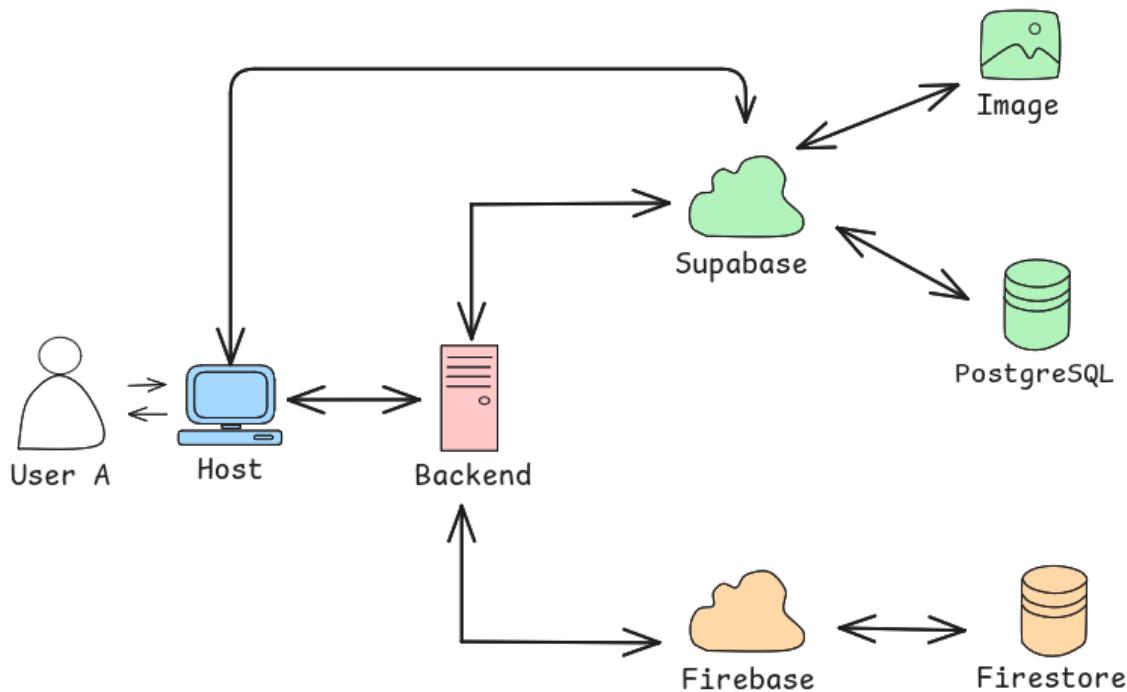


Figura 1: Diagrama de Arquitectura

3. Configuración y Conexión a las Bases de Datos

3.1. Configuración y conexión con SQL

Se utilizó **Prisma**, una herramienta ORM (Object-Relational Mapping) para manejar la conexión con PostgreSQL, que en este caso está alojado en Supabase. El archivo de configuración `prisma/schema.prisma` define el esquema de las tablas y sus relaciones:

```

1 datasource db {
2   provider  = "postgresql"
3   url       = env("DATABASE_URL") // URL configurada en variables de entorno
4   directUrl = env("DIRECT_URL")
5 }
6
7 generator client {
8   provider = "prisma-client-js"
9 }

```

Prisma genera automáticamente un cliente en base al esquema definido, que se utiliza para realizar consultas y operaciones en la base de datos.

Librerías utilizadas

- `@prisma/client`: Para interactuar con la base de datos desde el código.
- `prisma`: Para migraciones y administración del esquema.

Conexión

La conexión a PostgreSQL se realiza utilizando una variable de entorno `DATABASE_URL`, que contiene la URL proporcionada por Supabase.

```

1 import { PrismaClient } from '@prisma/client';
2
3 const prisma = new PrismaClient();
4
5 async function main() {
6   const data = await prisma.user.findMany(); // Ejemplo de consulta
7   console.log(data);
8 }
9
10 main()
11   .catch((e) => console.error(e))
12   .finally(async () => await prisma.$disconnect());

```

3.2. Conexión a Firestore con Firebase

Configuración

Se utilizó la biblioteca oficial de Firebase, configurando el cliente con las credenciales obtenidas de Firebase. Las credenciales están almacenadas en un archivo JSON que no se sube al repositorio (gestión segura a través de `.gitignore` y variables de entorno).

Librerías utilizadas

- `firebase-admin`: Para la configuración del servidor y conexión segura.
- `firebase/firestore`: Para el manejo de operaciones en la base de datos.

Conexión

El archivo `firebase.js` configura y exporta la instancia de Firestore para que pueda ser utilizada en toda la aplicación.

```

1 import admin from "firebase-admin";
2 import { getFirestore } from "firebase-admin/firestore";
3

```

```
4 // Usa la Variable de Entorno: GOOGLE_APPLICATION_CREDENTIALS. Esta apunta al ./
  serviceAccountKey.json
5 admin.initializeApp({
6   credential: applicationDefault();
7 });
8
9 const db = getFirestore();
10
11 export default db;
```

En este ejemplo, la variable `GOOGLE_APPLICATION_CREDENTIALS` contiene las credenciales JSON codificadas como cadena. Esto evita almacenar las credenciales directamente en el código fuente.

3.3. Uso de Variables de Entorno

Todas las configuraciones sensibles, como las URLs y credenciales de conexión, se almacenan en un archivo `.env`. Ejemplo:

```
1 // Las diferentes URLs son por los tipos de conexion a la
2 DATABASE_URL=postgresql://username:password@host:port/database
3 DIRECT_URL=postgresql://username:password@host:port/database
4 GOOGLE\_APPLICATION\_CREDENTIALS="path/to/serviceAccountKey.json"
```

- **PostgreSQL (Supabase):** Conexión a través de Prisma con `DATABASE_URL` en variables de entorno.
- **Firestore (Firebase):** Conexión mediante `firebase-admin` con credenciales en variables de entorno.
- **Variables de Entorno:** Gestionan URLs, credenciales, y claves de acceso para asegurar que la información sensible no esté expuesta.

4. Operaciones CRUD y datos

4.1. Datos en SQL

Los datos en PostgreSQL son 3 tablas pertenecientes a Teams, Players y Users. En la figura 2 se puede apreciar la estructura de las mismas.

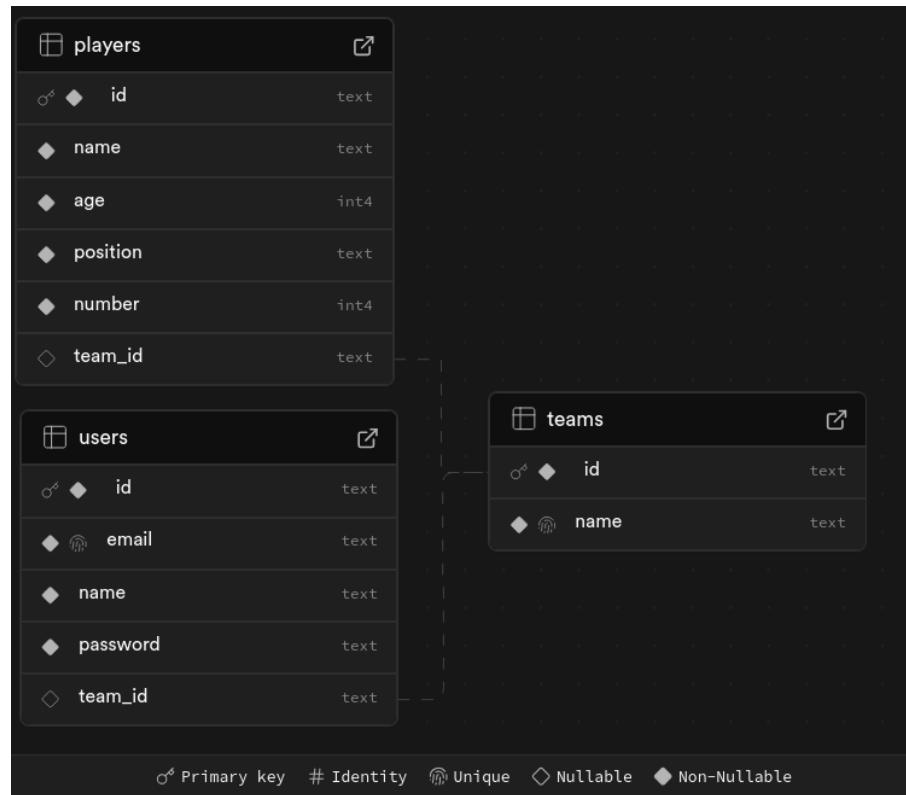


Figura 2: Tablas visualizadas en Supabase

4.2. Tipos de Usuarios

La aplicación permite la interacción de dos tipos de usuarios, cada uno con diferentes niveles de acceso y funcionalidades:

- **ADMIN:** Los administradores tienen acceso completo a todas las funciones de la aplicación, incluyendo las operaciones CRUD (Crear, Leer, Actualizar y Eliminar). La creación y modificación de jugadores y equipos está restringida exclusivamente a este tipo de usuario. Las credenciales para acceder como administrador son:
 - **Correo:** admin@gmail.com
 - **Contraseña:** 12345678
- **USER:** Los usuarios regulares logueados pueden explorar la base de datos, buscar jugadores y equipos, y emitir opiniones sobre los mismos. Sin embargo, no tienen permisos para realizar modificaciones de ciertos datos almacenados, a excepción de sus opiniones o comentarios. Este tipo de usuario está diseñado para fomentar la interacción sin comprometer la integridad de la información.

4.2.1. Teams

POST/Creación de equipos

Para crear un nuevo equipo, el usuario debe estar logueado como *ADMIN* en la aplicación web. En la página principal (homepage), el administrador verá un botón exclusivo que lo llevará a la sección de creación de equipos, como se muestra en la figura siguiente:

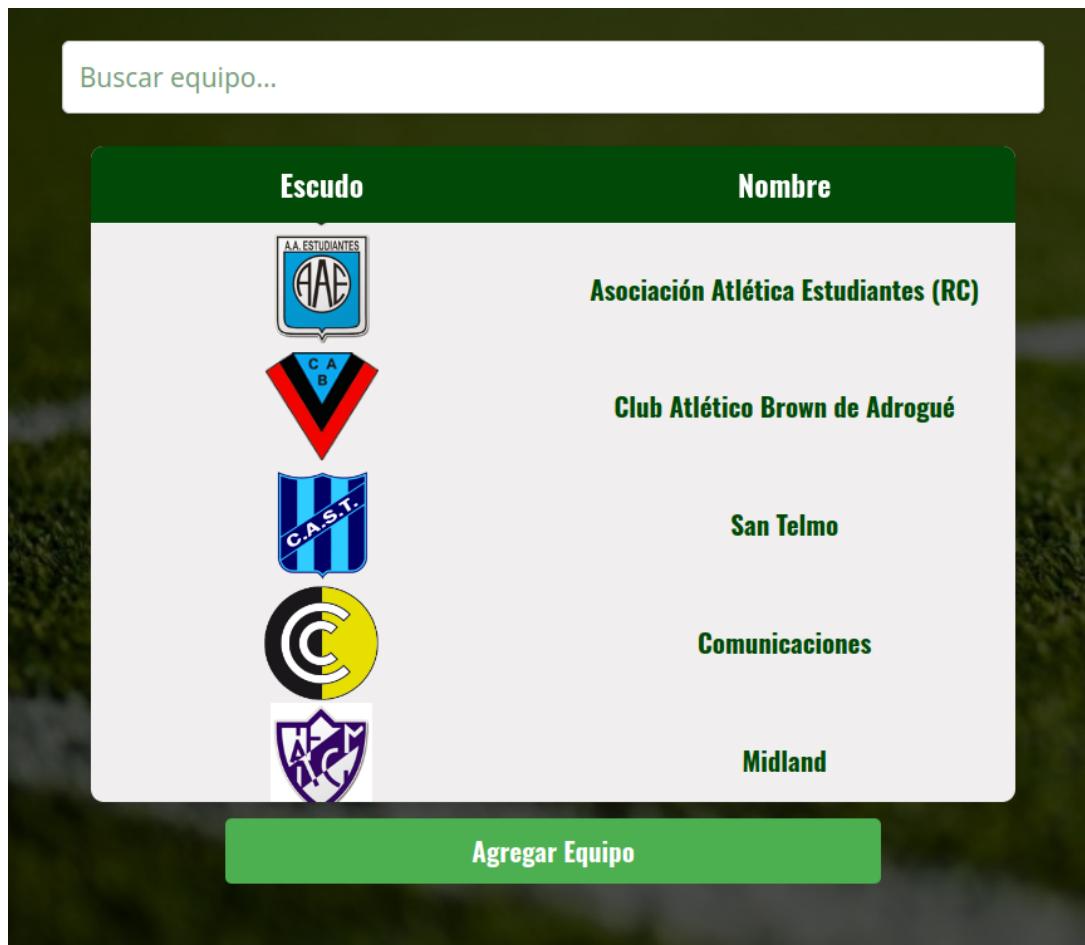


Figura 3: Homepage con acceso exclusivo para administradores

Al hacer clic en el botón **Agregar Equipo**, el usuario será redirigido a una interfaz donde podrá completar un formulario para añadir un nuevo equipo. En esta sección, se solicita información como el nombre del equipo, descripción y otros detalles. La carga de una foto del equipo es opcional, como se muestra en la figura siguiente:

The form has a title "Agregar nuevo equipo". It contains a single input field labeled "Nombre del equipo" (Team name) with the value "El Mejor Equipo Del Mundo". Below the input field is a green button labeled "Seleccionar escudo" (Select shield). A message "No se ha seleccionado un archivo" (No file selected) is displayed below the button. At the bottom are two buttons: "Confirmar" (Confirm) in green and "Cancelar" (Cancel) in red.

Figura 4: Formulario para la creación de un equipo

Una vez completados los datos, el administrador tiene la opción de cargar una imagen representativa del equipo. Este paso es opcional y se ilustra en la siguiente figura:

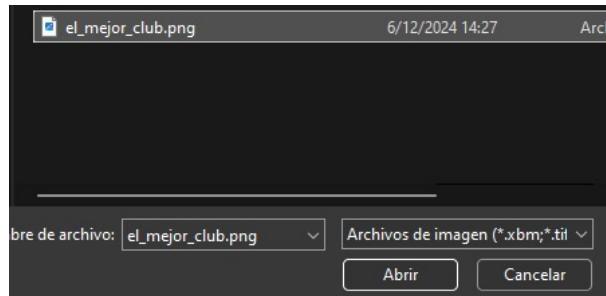


Figura 5: Carga opcional de una foto del equipo

Después de confirmar los datos y hacer clic en el botón de creación, el nuevo equipo aparecerá en la página web. Esto demuestra que el sistema está funcionando correctamente, como se muestra en la siguiente figura:

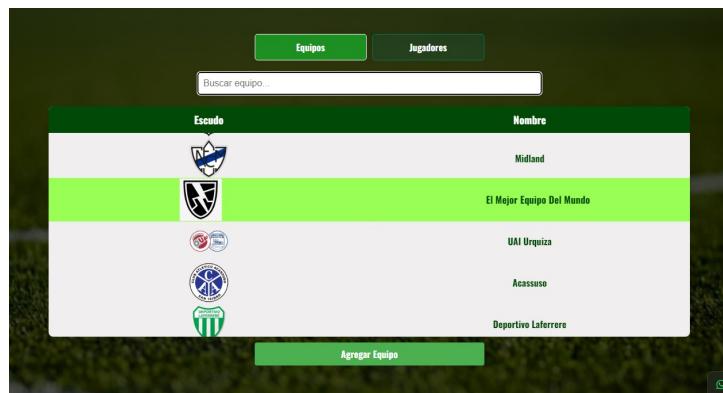


Figura 6: Visualización del nuevo equipo en la página web

PUT/Edición de equipos

Para editar un equipo, el administrador debe hacer clic sobre el equipo que desea modificar. Esto lo llevará a la sección específica del equipo, como se muestra en la figura siguiente:

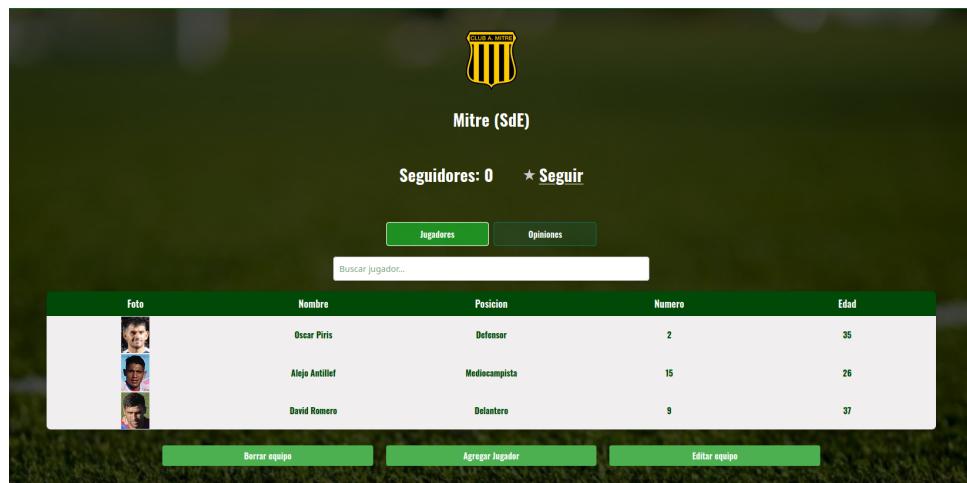


Figura 7: Sección del equipo seleccionado para editar

Una vez en la sección del equipo, el administrador puede hacer clic en el botón **Editar Equipo**. Esto abrirá un *popup* similar al formulario de creación de equipo (figura 4), donde podrá realizar las modificaciones deseadas.



Figura 8: Formulario de edición de equipo

DELETE/Borrado de equipos

Para borrar un equipo, el administrador debe seleccionar el botón **Borrar Equipo**, visible en la sección del equipo (figura 7). Al presionar este botón, aparecerá un *popup* de confirmación, donde el administrador deberá confirmar la acción.

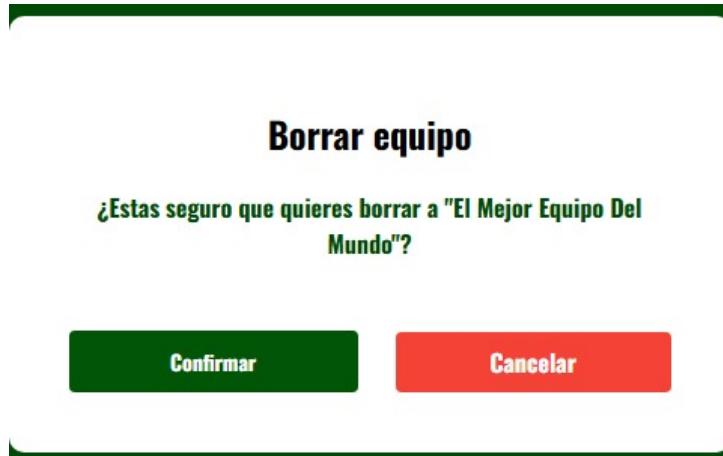


Figura 9: Ventana emergente para confirmar el borrado de un equipo

GET/Visualización de equipos

La información de los equipos se puede consultar a través de un *fetch* desde la homepage. Los administradores y usuarios pueden buscar equipos fácilmente utilizando la barra de búsqueda para encontrar equipos específicos de manera rápida.

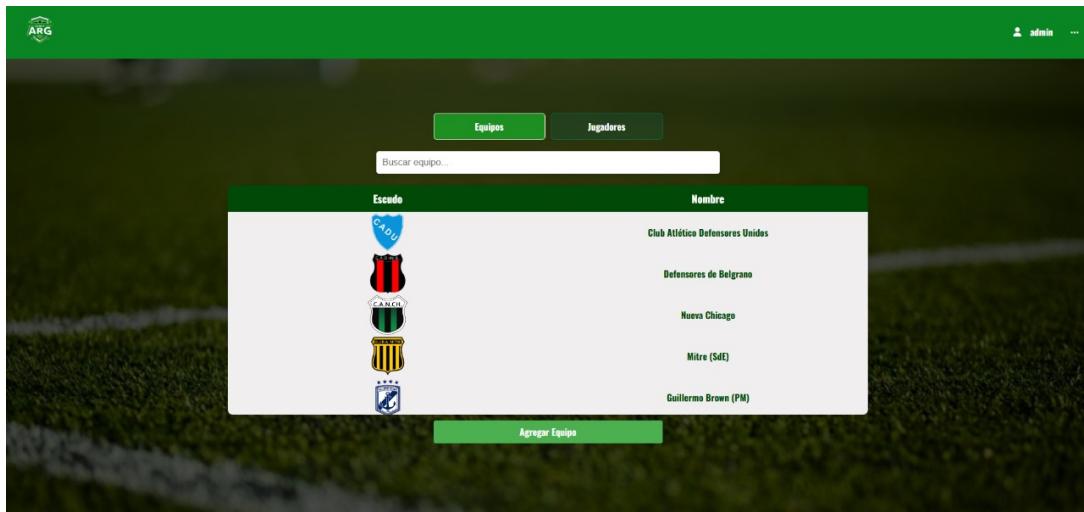


Figura 10: Visualización de equipos en la página principal con barra de búsqueda

4.2.2. Players

POST/Creación de jugadores

Los jugadores pueden estar afiliados a un equipo específico. Para agregar un nuevo jugador a un equipo, el administrador debe dirigirse a la sección de equipos, como se explicó anteriormente (figura 7), y hacer clic en el botón **Agregar Jugador**. Esto abrirá un formulario para ingresar los datos del jugador, como se muestra en la figura 11. Al igual que con los equipos, la carga de una foto del jugador es opcional.

Agregar nuevo jugador

Nombre

Edad

Posicion

Numero

Seleccionar foto

No se ha seleccionado un archivo

Confirmar Cancelar

Figura 11: Formulario para la creación de un jugador

GET/Vista de jugadores

Para ver más detalles sobre un jugador, el administrador o usuario puede hacer clic en el jugador desde la sección de equipos, lo que lo llevará a su sección específica, como se muestra en la figura 12.

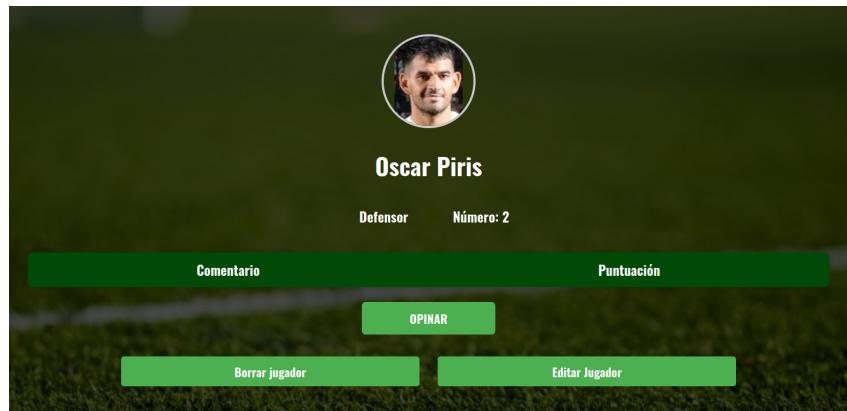


Figura 12: Sección específica del jugador

También existe una sección donde se pueden visualizar todos los jugadores registrados en el sistema. Esta lista es accesible desde la página principal y se puede filtrar para facilitar la búsqueda de jugadores, como se muestra en la figura 13.

Foto	Nombre	Posición	Número	Edad	Equipo
	Matías Laba	Mediocampista	5	32	Defensores de Belgrano
	Facundo Pereyra	Mediocampista	10	37	Defensores de Belgrano
	Brian Gómez	Delantero	7	30	Defensores de Belgrano
	Lisandro Mitre	Arquero	1	28	Club Atlético All Boys
	Jonathan Ferreira	Defensor	2	32	Club Atlético All Boys

Figura 13: Sección de jugadores

PUT/Edición de jugadores

De manera similar a la edición de equipos, en la sección individual de cada jugador (figura 12), el administrador tiene la opción de editar los datos del jugador. Para ello, debe hacer clic en el botón **Editar Jugador**, lo cual abrirá un *popup* similar al siguiente, como se muestra en la figura 14, donde podrá modificar la información del jugador.

Editar Jugador

Nombre
Matias Laba

Edad
32

Posicion
Mediocampista

Numero
5

Equipo
Buscar...
Defensores de Belgrano

Seleccionar foto

No se ha seleccionado un archivo

Confirmar **Cancelar**

Figura 14: Formulario para editar un jugador existente

DELETE/Borrado de jugadores

Al igual que en el caso de los equipos, para borrar un jugador, el administrador debe hacer clic en el botón **Borrar Jugador** dentro de la sección individual del jugador. Este botón abrirá un *popup* de confirmación, como se ilustra en la figura 15, donde el administrador deberá confirmar la acción para eliminar al jugador.

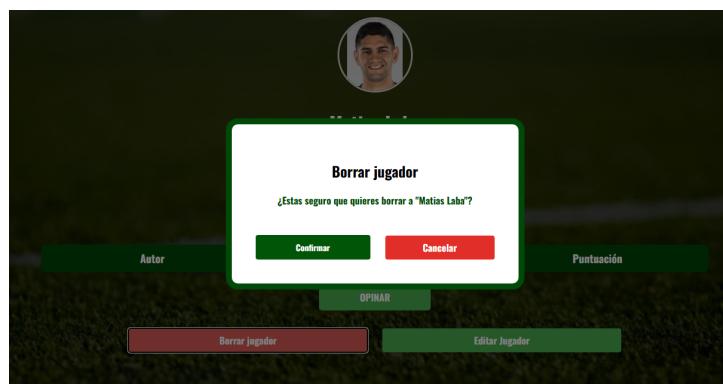


Figura 15: Ventana emergente para borrar un jugador

4.2.3. Users

Registro/Inicio de sesión

Al acceder a la página por primera vez, los usuarios tendrán la opción de iniciar sesión o registrarse, según corresponda. Si el usuario ya tiene una cuenta, puede ingresar sus credenciales en la sección de inicio de sesión. Por otro lado, si el usuario no tiene cuenta, puede registrarse proporcionando la información necesaria. Un usuario ordinario, una vez logueado, podrá interactuar con la plataforma pero no tendrá privilegios para modificar, crear o eliminar equipos o jugadores, funciones reservadas exclusivamente para los administradores.

Como recordatorio, las credenciales predeterminadas para acceder como administrador son:

- **Correo:** admin@gmail.com
- **Contraseña:** 12345678

Seguir Equipos

Cada usuario tiene la opción de seguir un equipo de su interés. Para hacerlo, debe ir a la sección del equipo específico y hacer clic en el botón **Seguir**. Este botón permite al usuario recibir actualizaciones y mostrar su apoyo al equipo. Cabe destacar que un usuario solo puede seguir un equipo a la vez, lo que le permite concentrarse en su equipo favorito. A continuación, se muestra una imagen de la sección donde los usuarios pueden seguir un equipo:



Figura 16: Sección para seguir un equipo. El usuario puede ver los detalles del equipo y optar por seguirlo.

4.2.4. Implementación de las operaciones en PostgreSQL

Todas las operaciones para interactuar y ejecutar consultas con la base de datos PostgreSQL se realizan mediante la abstracción de Prisma.

Los esquemas, tal como se menciona en la configuración de la conexión a la base de datos, se definen en el archivo `schema.prisma`:

```

1   model users {
2     id      String    @id @default(cuid())
3     email   String    @unique
4     name    String
5     password String
6     team_id String?  // Clave foranea hacia 'teams' (opcional)
7     team    teams?   @relation(fields: [team_id], references: [id], onDelete:
8       SetNull)
9   }
10
11  model teams {
12    id      String    @id @default(cuid())
13    name   String    @unique
14    users  users[]
15    players players[]
16  }
17
18  model players {
19    id      String    @id @default(cuid())
20    name   String
21    age    Int
22    position String
23    number  Int
24    team_id String?  // Clave foranea hacia 'teams' (opcional)
25    team    teams?   @relation(fields: [team_id], references: [id], onDelete:
     SetNull)
}

```

Esto permite definir rápidamente las tablas y los esquemas con los cuales se trabajará en el código.

Inserciones

Las inserciones, como por ejemplo la de un equipo en la base de datos, se realizan a través del esquema `teams` en el servicio de Prisma utilizando el método `create`. Esta función recibe una clase con todos los atributos que están presentes también en la tabla de equipos:

```

1  async create(createTeamDto: CreateTeamDto): Promise<TeamEntity> {
2    const team = await this.prisma.teams.create({
3      data: createTeamDto,
4    });
5    return new TeamEntity({
6      ...team,
7    });
8  }

```

Prisma envuelve el `insert` que se realiza internamente con los valores proporcionados por la `data`. Además, devuelve los datos recién insertados de forma correcta.

Ediciones

De manera similar a las inserciones, las actualizaciones o ediciones se realizan con el mismo servicio de Prisma utilizando el método `update`:

```

1  async update(id: string, updateTeamDto: UpdateTeamDto): Promise<TeamEntity> {
2    const team = await this.prisma.teams.update({
3      where: {
4        id,
5      },
6      data: updateTeamDto,
7    });
8    if (!team) {
9      throw new NotFoundException(`Team with id ${id} not found`);
10   }
11   return new TeamEntity({
12     ...team,
13   });
14 }

```

Es importante notar que en este caso, el `update` requiere el `id` del equipo que se quiere actualizar.

Borrados

Siguiendo la lógica de las operaciones anteriores, Prisma nuevamente facilita el `delete`. En este caso, es necesario gestionar ambas bases de datos. No solo se realiza una consulta a la base de datos a través de Prisma, sino que también se hace una llamada a Firestore para eliminar las opiniones o comentarios relacionados con ese jugador.

```

1  async remove(id: string) {
2      await this.opinions.deleteOpinionsByPlayerId(id);
3
4      const playerDeleted = await this.prisma.players.delete({
5          where: {
6              id,
7          },
8      });
9      if (!playerDeleted) {
10          throw new NotFoundException(`Player with id ${id} not found`);
11      }
12
13      return playerDeleted;
14  }

```

Es fundamental destacar que es necesario especificar qué jugador se quiere eliminar.

Búsquedas

Finalmente, Prisma también incluye envoltorios para consultas SELECT mediante los métodos `findMany` y `findOne`, que se utilizan para buscar registros.

En el caso de `findMany`, se trata de un envoltorio de un SELECT con una condición que puede o no devolver más de un registro:

```

1  async findAll(): Promise<PlayerEntity[]> {
2      return await this.prisma.players.findMany({
3          include: {
4              team: true,
5          },
6      });
7  }

```

Este sería un envoltorio de `SELECT * FROM TEAMS;`.

Por otro lado, `findOne` sería un envoltorio de `SELECT * FROM TEAMS WHERE TEAMS.id = 'id';`:

```

1  async findAll(): Promise<PlayerEntity[]> {
2      return await this.prisma.players.findMany({
3          include: {
4              team: true,
5          },
6      });
7  }

```

4.3. Datos en Firestore

En contraste con las bases de datos relacionales como PostgreSQL, donde los datos se organizan en tablas, en Firestore se trabaja con colecciones y documentos. Las colecciones son grupos de documentos, y cada documento puede contener una variedad de datos estructurados. En Firestore no se necesita definir un esquema rígido, lo que permite una mayor flexibilidad en la organización de los datos. Sin embargo, esta flexibilidad también implica que la definición y gestión de las entidades debe ser realizada manualmente, ya que Firestore no ofrece migraciones automáticas como Prisma.

Para el proyecto, se han definido cuatro colecciones principales en Firestore, que están relacionadas con los dos dominios centrales del sistema: equipos y jugadores. A continuación, se describe brevemente la estructura de estas colecciones y su interrelación:

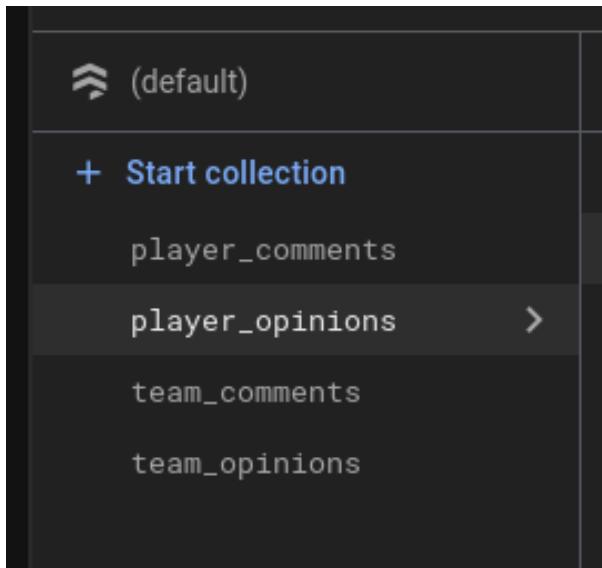


Figura 17: Collections en Firestore

- **Opiniones de Jugadores:** Esta colección contiene los documentos de las opiniones que los usuarios dejan sobre un jugador. Cada documento en esta colección está vinculado al ID del jugador correspondiente.
- **Opiniones de Equipos:** Similar a las opiniones de jugadores, esta colección contiene las opiniones de los usuarios sobre los equipos. Cada documento está asociado a un equipo específico.
- **Comentarios sobre Opiniones de Jugadores:** En esta colección se almacenan los comentarios que los usuarios dejan sobre las opiniones relacionadas con los jugadores. Cada comentario está vinculado a una opinión de jugador específica.
- **Comentarios sobre Opiniones de Equipos:** De la misma manera que los comentarios sobre jugadores, esta colección incluye los comentarios que los usuarios dejan sobre las opiniones relacionadas con los equipos.

Con esta estructura, un usuario puede dejar opiniones tanto sobre un jugador como sobre un equipo, y además puede agregar comentarios a las opiniones de otros usuarios, ya sea sobre jugadores o equipos.

Ventajas de tener colecciones separadas para los comentarios y las opiniones

Es importante destacar la decisión de separar los comentarios de las opiniones en colecciones independientes. Esta estructura de datos tiene varias ventajas tanto desde el punto de vista del rendimiento como de la organización de los datos:

1. **Escalabilidad y rendimiento:** Al mantener los comentarios en una colección separada, se reduce la complejidad de las consultas al limitar la cantidad de datos que se necesitan recuperar de cada documento. Si los comentarios estuvieran incluidos directamente en las opiniones, cada vez que se accediera a una opinión, también se cargarían los comentarios, lo que podría ser ineficiente, especialmente si una opinión tiene muchos comentarios. Al mantenerlos separados, solo se accede a los comentarios cuando es necesario, mejorando el rendimiento de las lecturas y evitando la sobrecarga de datos.
2. **Flexibilidad en la gestión de datos:** Separar las colecciones permite una mayor flexibilidad para gestionar y manipular los datos de manera independiente. Por ejemplo, si se desea agregar o eliminar comentarios, esto no afectará directamente las opiniones. De igual forma, la estructura modular facilita la implementación de funciones adicionales en el futuro, como la posibilidad de ordenar comentarios, realizar búsquedas específicas, o incluso limitar el número de comentarios que se muestran en una interfaz de usuario sin afectar las opiniones mismas.

3. **Optimización de consultas:** Firestore es una base de datos NoSQL, lo que significa que la forma en que se organizan los datos afecta directamente la eficiencia de las consultas. Al mantener las colecciones de comentarios y opiniones separadas, se pueden optimizar las consultas para que accedan solo a los documentos necesarios. Por ejemplo, si un usuario desea ver las opiniones de un equipo, no será necesario recuperar también los comentarios asociados, lo que mejora la eficiencia de las consultas. Además, se puede usar un sistema de paginación o de carga diferida (lazy loading) para los comentarios, lo que mejora aún más la experiencia del usuario.
4. **Estructura más limpia y mantenible:** Al tener colecciones separadas para los comentarios y las opiniones, la estructura de la base de datos se vuelve más clara y fácil de mantener. Las colecciones de comentarios pueden tener su propia lógica de acceso y reglas de seguridad, sin interferir con las reglas y la estructura de las colecciones de opiniones. Esto facilita la gestión de permisos y la implementación de nuevas funcionalidades sin comprometer la integridad de las opiniones o comentarios.

De esta forma, tener las opiniones y los comentarios en colecciones separadas no solo mejora la organización y flexibilidad de los datos, sino que también optimiza el rendimiento y escalabilidad de las consultas, lo que es crucial cuando se trabaja con grandes volúmenes de datos en un sistema distribuido como Firestore. Esta estructura modular permite realizar ajustes de manera más eficiente, ofreciendo una mejor experiencia tanto para los usuarios como para los desarrolladores a largo plazo.

4.3.1. Opinions

Post/Crear opiniones

En la siguiente figura se muestra la sección de opiniones de un equipo:



Figura 18: Sección de opiniones de un equipo

Para crear una nueva opinión, basta con hacer clic en el botón **Opinar** (Figura 19):



Figura 19: Pop-up para crear una opinión de equipo

Este botón abre un pop-up donde el usuario puede ingresar su opinión sobre el equipo, permitiendo la creación de una nueva entrada en la base de datos.

Put/Editar opiniones

Para editar una opinión existente, el usuario debe hacer clic sobre la opinión correspondiente del equipo o jugador. Esto abrirá un pop-up de edición, que solo será visible para el autor de la opinión o un administrador:



Figura 20: Pop-up de edición de opinión

En este pop-up, se pueden modificar detalles como el rating (valoración) y la puntuación de la opinión. Una vez realizados los cambios, el usuario debe hacer clic en el botón Guardar cambios para aplicar las modificaciones.

Delete/Borrar opiniones

Para eliminar una opinión, el proceso es similar al de la edición: el usuario debe hacer clic en el botón Borrar Opinión, que se encuentra en la misma interfaz de edición mostrada en la Figura 20. Al presionar este botón, la opinión será eliminada de la base de datos de forma permanente.

Get/Obtener opiniones

Para ver las opiniones de un equipo o jugador, el usuario solo tiene que navegar hasta la sección correspondiente del equipo y luego acceder a su apartado de opiniones. A continuación, se muestra la opinión previamente creada en la figura:



Figura 21: Opinión en un equipo

En esta sección, el usuario podrá leer todas las opiniones que otros usuarios han dejado sobre el equipo o jugador en cuestión.

4.3.2. Comentarios

Post/Crear comentario

Para crear un comentario, el primer paso es dirigirse a la opinión sobre la cual se desea comentar. Para ello, se debe hacer clic en la opinión, lo que abrirá un menú similar al de edición (si el usuario no es el autor de la opinión). Luego, al hacer clic en el botón **Comentar**, se abrirá una pequeña sección donde el usuario podrá escribir su comentario:



Figura 22: Sección para crear un comentario

Una vez escrito el comentario, el usuario podrá publicarlo y este se asociará a la opinión correspondiente.

Put/Editar comentario

Para editar un comentario previamente creado, el usuario debe hacer clic en el botón de **Ver comentarios** ubicado en la sección de la figura 22:

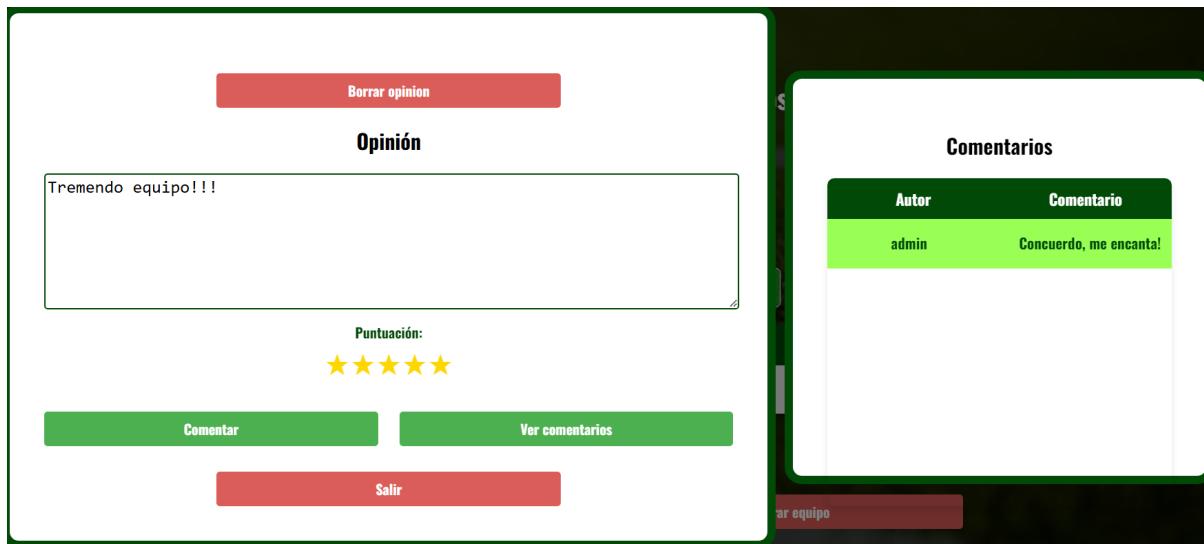


Figura 23: Vista de los comentarios de una opinión

Al hacer clic, se mostrará una lista de los comentarios, y al seleccionar uno, se abrirá un pop-up de edición que permitirá modificar el contenido del comentario:



Figura 24: Pop-up de edición de comentario

Una vez realizados los cambios necesarios, el usuario podrá guardar las modificaciones para actualizar el comentario.

Delete/Borrar comentario

Para eliminar un comentario, el proceso es similar al de la edición. El usuario debe hacer clic en el botón **Borrar comentario**, que se encuentra en la misma interfaz de edición mostrada en la Figura 24. Al presionar este botón, el comentario será eliminado permanentemente.

Get/Obtener comentarios

Para ver los comentarios de una opinión, el usuario debe hacer clic en el botón **Ver comentarios** como se muestra en la figura 23. Esto abrirá una sección dedicada exclusivamente a la visualización de los comentarios asociados a la opinión seleccionada.

En esta sección, se podrán leer todos los comentarios que los usuarios hayan dejado sobre esa opinión, permitiendo una interacción más rica y detallada.

4.3.3. Implementación de las operaciones en Firestore

Inserciones

Las inserciones, como por ejemplo la creación de un comentario en la base de datos, se realizan a través de la conexión con Firebase. Para ello, se utiliza el método `add` sobre la referencia de la colección correspondiente, después de convertir los datos a formato JSON. A continuación se muestra el código para insertar una nueva opinión de jugador:

```

1  async createOpinion(opinion: CreatePlayerOpinionDto): Promise<any> {
2      const plainOpinion = JSON.parse(JSON.stringify(opinion));
3      const docRef = await this.playerOpinionsCollection.add(plainOpinion);
4      return { id: docRef.id, ...opinion };
5  }

```

En este ejemplo, se convierte la opinión a `.JSON` antes de insertarla en la colección `playerOpinionsCollection`, lo que garantiza que los datos sean correctamente estructurados en Firestore.

Ediciones

Para las actualizaciones o ediciones de documentos, se sigue un proceso similar al de las inserciones. En este caso, primero se obtiene el documento específico a través de su `id` utilizando el método `doc`. Posteriormente, se aplica el método `update` sobre el documento para modificar los campos deseados. El código para actualizar una opinión de jugador es el siguiente:

```

1  async updateOpinion(id: string, opinion: UpdatePlayerOpinionDto): Promise<any> {
2      const plainOpinion = JSON.parse(JSON.stringify(opinion));
3      await this.playerOpinionsCollection.doc(id).update(plainOpinion);
4      return { id, ...opinion };
5  }

```

Este código actualiza los datos de una opinión de jugador, identificada por su `id`, utilizando el método `update` para modificar los campos específicos.

Borrados

Para eliminar un documento en Firestore, primero se obtiene el documento correspondiente mediante su `id`, y luego se realiza una consulta a la colección de comentarios para eliminar las opiniones asociadas. A continuación, se elimina el documento de la colección principal. El código para borrar una opinión junto con sus comentarios asociados es el siguiente:

```

1  async deleteOpinion(id: string): Promise<any> {
2      const docRef = this.playerOpinionsCollection.doc(id);
3      const docSnapshot = await docRef.get();
4      const deletedID = docSnapshot.id;
5      const deletedData = docSnapshot.data();
6      const commentsSnapshot = await this.playerCommentsCollection.where(
7          'player_opinion_id', '==', id).get();
8      commentsSnapshot.docs.forEach(async (comment) => {
9          await comment.ref.delete();
10     });
11     await docRef.delete();
12     return { id: deletedID, ...deletedData };
13 }

```

Este método no solo elimina la opinión principal, sino que también busca y elimina todos los comentarios asociados a esa opinión mediante una consulta `where` en la colección `playerCommentsCollection`.

Búsquedas

Finalmente, Firestore permite realizar búsquedas de documentos dentro de una colección de manera intuitiva. A continuación se muestra un ejemplo de cómo buscar las opiniones de un jugador utilizando su `playerId`. El código para realizar esta búsqueda es el siguiente:

```

1  async getOpinionsByPlayerId(playerId: string): Promise<any[]> {
2      const snapshot = await this.playerOpinionsCollection.where('player_id', '==',
3          playerId).get();
4      return snapshot.docs.map((doc) => ({ id: doc.id, ...doc.data() }));

```

En este caso, se utiliza el método `where` para filtrar los documentos que coinciden con el `playerId` proporcionado, y luego se mapean los resultados para devolver los datos completos de las opiniones del jugador en un formato adecuado.

5. Comparación entre Bases de Datos Relacionales y NoSQL

En el desarrollo de esta aplicación, como fuimos mencionando, hemos decidido emplear dos tipos de bases de datos para distintos propósitos, aprovechando las características únicas de cada una. Específicamente, hemos utilizado:

- **PostgreSQL**, una base de datos relacional, para manejar las entidades `users`, `teams`, y `players`.

- **Firestore**, una base de datos NoSQL, para almacenar opiniones y comentarios relacionados con equipos y jugadores.

Inicialmente, teníamos pensado utilizar **MongoDB** debido a su flexibilidad y facilidad para manejar datos no estructurados. Sin embargo, después de evaluar diversas opciones y los requisitos específicos del proyecto, decidimos incursionar en otras bases de datos como Firestore, ya que ofrece una integración directa con el ecosistema de Google y una gestión automática de escalabilidad. A pesar de la popularidad de MongoDB en aplicaciones NoSQL, Firestore resultó ser una opción más adecuada debido a su rendimiento optimizado para lecturas frecuentes y su simplicidad al integrar funciones como la autenticación y el almacenamiento de archivos.

A continuación, se presentarán una serie de ventajas y desventajas observadas al trabajar con estas dos tecnologías, seguidas de una reflexión sobre cuándo sería más conveniente utilizar una u otra, según los requerimientos específicos del proyecto.

5.1. Ventajas y Desventajas

La siguiente tabla resume las principales ventajas y desventajas de PostgreSQL y Firestore, comparando los aspectos clave que influyen en la elección de cada una:

Aspecto	PostgreSQL (Relacional)	Firestore (NoSQL)
Ventajas	<ul style="list-style-type: none"> ■ Estructura de datos bien definida y basada en esquemas (<code>users</code>, <code>teams</code>, <code>players</code>). ■ Integridad referencial garantizada mediante el uso de claves foráneas, lo que asegura la consistencia entre tablas relacionadas. ■ Potentes herramientas para realizar consultas complejas, como <code>JOINs</code>, subconsultas y agrupaciones. ■ Soporte completo de transacciones, lo que asegura la atomicidad y consistencia en las operaciones de múltiples tablas. 	<ul style="list-style-type: none"> ■ Flexible, permite agregar o modificar campos sin necesidad de una estructura fija. ■ Integración nativa con el ecosistema de Google y Firebase, lo que facilita su uso en aplicaciones móviles y web. ■ Excelente rendimiento para consultas simples y lecturas frecuentes, especialmente en grandes volúmenes de datos. ■ Escalabilidad automática: Firestore maneja la distribución de datos y la carga de trabajo de manera eficiente sin intervención del usuario.
Desventajas	<ul style="list-style-type: none"> ■ Menor flexibilidad frente a cambios en el esquema, lo que requiere migraciones complicadas si se desea modificar el modelo de datos. ■ Escalabilidad más compleja que Firestore sin configuraciones adicionales, como el uso de particiones de tablas o sharding. ■ Dependencia de relaciones complejas: mantener la integridad referencial entre tablas puede resultar costoso en términos de rendimiento. ■ Requiere más esfuerzo en la gestión de índices para asegurar un buen rendimiento en grandes bases de datos. 	<ul style="list-style-type: none"> ■ Las consultas complejas pueden requerir múltiples accesos a la base de datos, lo que puede afectar el rendimiento en operaciones más sofisticadas. ■ Consistencia eventual: Firestore no garantiza consistencia inmediata en todos los nodos, lo que puede ser problemático en operaciones críticas de escritura. ■ No permite modelar relaciones de manera tan eficiente, lo que puede generar duplicación de datos para evitar consultas complejas.

Cuadro 1: Ventajas y desventajas de PostgreSQL y Firestore.

5.2. Conclusión sobre los diferentes casos de uso

La combinación de PostgreSQL y Firestore en esta aplicación ha sido una decisión estratégica que ha permitido aprovechar lo mejor de ambos mundos. PostgreSQL, al ser una base de datos relacional, se ha

utilizado para gestionar las entidades fundamentales de la aplicación, como `users`, `teams`, y `players`. Esta elección asegura la consistencia, integridad y eficiencia en las relaciones complejas entre usuarios, equipos y jugadores, permitiendo realizar operaciones transaccionales con alta fiabilidad y optimización en consultas complejas, como `JOINS` y filtros avanzados.

Por otro lado, Firestore, al ser una base de datos NoSQL, ha sido la opción ideal para almacenar opiniones y comentarios relacionados con equipos y jugadores. Firestore ofrece la flexibilidad, escalabilidad y rendimiento necesarios para manejar grandes volúmenes de datos no estructurados o semi-estructurados. Gracias a su modelo de documentos, que permite almacenar campos dinámicos y trabajar con colecciones, Firestore facilita la expansión y la adaptación a cambios en el esquema sin complicaciones adicionales.

La decisión de combinar una base de datos relacional y una NoSQL se basó en una evaluación de los requisitos específicos del proyecto, y ha permitido que la arquitectura de la aplicación se beneficie de las fortalezas de ambas tecnologías. En resumen, las bases de datos relacionales, como PostgreSQL, son ideales para gestionar datos estructurados y relaciones complejas, mientras que las bases de datos NoSQL, como Firestore, son más adecuadas para manejar grandes volúmenes de datos flexibles, especialmente cuando las consultas son más simples o la escalabilidad es un factor crítico.

Al elegir entre una base de datos relacional o NoSQL, es fundamental considerar el tipo de datos que se manejarán y los requisitos específicos de la aplicación. Mientras que las bases de datos relacionales son óptimas para datos con relaciones estrictas y necesidades de consistencia transaccional, las bases de datos NoSQL son más eficaces cuando se requieren altas tasas de lectura y flexibilidad en el esquema.

Esta arquitectura híbrida ha permitido maximizar el rendimiento de la aplicación mientras se mantiene la consistencia y la integridad de los datos. Además, ha brindado una solución eficaz y escalable para las necesidades del proyecto, demostrando que, al combinar tecnologías complementarias, se pueden superar las limitaciones de cada tipo de base de datos por separado.

6. Dificultades Principales y Soluciones Implementadas

Durante el desarrollo de la aplicación, enfrentamos diversos desafíos técnicos que pusieron a prueba nuestra capacidad para integrar tecnologías con paradigmas diferentes y optimizar el rendimiento de la solución. A continuación, se detallan los principales desafíos, las soluciones implementadas y las lecciones aprendidas.

6.1. Integración entre PostgreSQL y Firestore

Desafío: Uno de los mayores retos fue integrar dos bases de datos con paradigmas muy distintos: PostgreSQL, una base de datos relacional, y Firestore, una base de datos NoSQL. Cada una tiene su propio modelo de datos y enfoques para manejar las relaciones entre entidades, lo que hizo que la sincronización de datos fuera más compleja. En particular, la necesidad de relacionar entidades en PostgreSQL con documentos en Firestore fue una tarea desafiante, debido a la ausencia de relaciones nativas en Firestore y las diferencias en la forma de manejar los esquemas de datos.

Solución: Para resolver este problema, implementamos una capa de abstracción en el backend utilizando NestJS, lo que permitió desacoplar las operaciones específicas de cada base de datos y simplificar la lógica de integración. Además:

- Se emplearon variables de entorno para manejar las URLs y configuraciones de cada base de datos, garantizando flexibilidad y seguridad en el acceso a las bases de datos desde el servidor.
- Desarrollamos adaptadores específicos para interactuar con Firestore y PostgreSQL, de modo que cada base de datos tuviera su propia lógica de interacción, lo que mejoró la mantenibilidad y escalabilidad del código.
- Para las relaciones entre opiniones/comentarios (Firestore) y usuarios/jugadores (PostgreSQL), adoptamos una estrategia de referencia cruzada", donde se almacenaron identificadores únicos (`id`)

de los documentos de Firestore en los registros de PostgreSQL, lo que permitió mantener una referencia consistente entre las bases de datos sin requerir una relación directa.

6.2. Manejo de Datos Relacionales en Firestore

Desafío: Una de las limitaciones más notorias de Firestore es que no permite definir relaciones nativas entre documentos, lo que llevó a la duplicación de datos en algunas colecciones. Por ejemplo, las colecciones `team-opinions` y `team-opinions-comments`, así como `players-opinions` y `players-opinions-comments`, contenían campos repetidos como `player_id` o `team_id` dentro de los documentos para facilitar las consultas.

Esto causó un desafío de consistencia, ya que cualquier cambio en los datos duplicados podría requerir actualizaciones en múltiples lugares, lo que incrementaba la posibilidad de inconsistencias.

Solución: Decidimos optimizar las consultas a costa de duplicar ciertos datos esenciales en los documentos para reducir el número de lecturas necesarias. Esto simplificó las operaciones y mejoró la velocidad de acceso. Sin embargo, para minimizar la posibilidad de inconsistencias:

- Implementamos funciones en el backend que aseguraban la coherencia de los datos durante las operaciones de creación, actualización y eliminación, garantizando que los datos duplicados se mantuvieran sincronizados.
- Adoptamos un enfoque de consistencia eventual en Firestore para opiniones y comentarios, lo que permitió priorizar el rendimiento en las consultas y la escalabilidad, sin requerir una consistencia estricta en todo momento.
- Para las actualizaciones, se implementaron procesos de validación en el backend para verificar que los cambios en los documentos no violaran la integridad de los datos relacionados.

6.3. Gestión de Usuarios y Seguridad

Desafío: La gestión de usuarios, especialmente la autenticación y autorización mediante JWT (*JSON Web Tokens*), fue un desafío importante. Asegurar que solo los usuarios con los permisos adecuados pudieran acceder a funcionalidades críticas, como la creación de equipos y la edición de opiniones, requería una implementación sólida de seguridad tanto en el backend como en el frontend.

Solución:

- Configuramos un middleware de autenticación en NestJS para validar los tokens en cada petición, asegurando que el acceso a las rutas protegidas estuviera restringido a usuarios autenticados.
- Establecimos una política de roles en la aplicación, diferenciando entre usuarios regulares y administradores, y asignando permisos adecuados a cada rol.
- Para mejorar la seguridad, implementamos mecanismos de validación de expiración de los tokens y la renovación automática de los mismos cuando era necesario, evitando que los usuarios se quedaran sin acceso inesperadamente.

6.4. Escalabilidad y Eficiencia

Desafío: El crecimiento potencial del volumen de datos, especialmente opiniones y comentarios en Firestore, podía generar consultas costosas y tiempos de respuesta elevados, lo que afectaría la experiencia del usuario. A medida que la base de datos se expandía, las operaciones de lectura frecuentes se volvían cada vez más lentas, lo que generaba un desafío para la escalabilidad.

Solución: Para enfrentar este reto, implementamos diversas estrategias:

- Diseñamos índices compuestos en Firestore para optimizar las consultas más frecuentes, como aquellas que implican la búsqueda de opiniones por `player_id` o `team_id`.
- Implementamos paginación en el frontend para limitar la cantidad de datos recuperados en cada consulta, evitando cargar grandes volúmenes de información innecesaria y mejorando la velocidad de las respuestas.
- En PostgreSQL, definimos índices sobre las columnas clave como `email` y `name` para acelerar las búsquedas y reducir el tiempo de consulta, asegurando que las operaciones de acceso a datos fueran rápidas y eficientes.

7. Conclusiones

- **Combinación de Bases de Datos Relacionales y NoSQL:** La elección de utilizar dos tipos de bases de datos, una relacional como PostgreSQL y una NoSQL como Firestore, ha resultado ser una solución altamente efectiva para abordar los distintos requerimientos del proyecto. Si bien PostgreSQL proporcionó robustez y consistencia en las relaciones entre las entidades fundamentales (usuarios, equipos, jugadores), Firestore demostró ser una opción más adecuada para gestionar datos semi-estructurados, como opiniones y comentarios, con una mayor flexibilidad. Sin embargo, esta estrategia requirió un diseño arquitectónico cuidadoso para evitar inconsistencias en los datos y para minimizar la duplicación innecesaria de información, particularmente cuando las relaciones entre las dos bases de datos no son nativas.
- **Uso de NestJS para una Arquitectura Modular:** NestJS fue elegido como el framework principal para el desarrollo del backend debido a su enfoque modular, que permitió separar la lógica de negocio en componentes reutilizables y fáciles de mantener. Esta modularidad resultó en una arquitectura escalable, que facilitó la incorporación de nuevas funcionalidades sin comprometer la estructura existente de la aplicación. Además, NestJS proporcionó un entorno robusto y bien organizado, lo que facilitó la integración de las dos bases de datos y la implementación de la lógica de negocio necesaria para interactuar con ellas.
- **Importancia de la Documentación y Variables de Entorno:** La correcta documentación del código y el uso adecuado de las variables de entorno fueron clave para el éxito del proyecto. En un entorno de desarrollo colaborativo, la documentación permitió que todos los miembros del equipo estuvieran alineados con respecto a las decisiones técnicas y las mejores prácticas adoptadas. Además, el uso de variables de entorno para gestionar configuraciones sensibles, como las credenciales de la base de datos y las claves API, garantizó que la aplicación fuera segura y fácilmente portable entre diferentes entornos de desarrollo, pruebas y producción.
- **Desafíos y Soluciones en la Integración de Sistemas:** La integración de bases de datos con arquitecturas diferentes supuso varios desafíos. Uno de los principales fue la sincronización de datos entre PostgreSQL y Firestore, especialmente cuando se trataba de mantener la consistencia de las relaciones entre las entidades almacenadas en ambas bases de datos. Sin embargo, a través de un diseño de referencia cruzada, donde se almacenaban identificadores únicos en los documentos de Firestore, pudimos establecer un vínculo eficiente entre las dos bases de datos. Esta estrategia permitió que las consultas y las operaciones de actualización o eliminación se realizaran de manera coherente y sin duplicación de datos.

8. Aprendizajes

- **Aprendizajes Técnicos:** Este proyecto ha sido una oportunidad invaluable para profundizar nuestros conocimientos en el uso de bases de datos tanto relacionales como NoSQL. La capacidad de manejar dos paradigmas diferentes de bases de datos en un mismo proyecto nos permitió evaluar y aplicar sus ventajas y limitaciones en función de las necesidades específicas de cada tipo de dato. Trabajar con PostgreSQL nos permitió mejorar nuestras habilidades en el diseño de esquemas y la

normalización de datos, mientras que el uso de Firestore nos dio una comprensión más profunda de cómo trabajar con datos semi-estructurados y cómo manejar la escalabilidad de manera eficiente. Además, aprendimos a utilizar herramientas como Prisma para interactuar con PostgreSQL y el SDK de Firestore, lo que facilitó enormemente el proceso de integración y consulta de datos.

- **Aprendizajes Organizacionales:** El desarrollo de este proyecto destacó la importancia de la comunicación y la coordinación dentro del equipo. La integración de tecnologías dispares, como bases de datos relacionales y NoSQL, requería que todos los miembros estuvieran alineados sobre cómo se debían manejar los datos y cómo interactuarían los diferentes componentes del sistema. Además, la planificación y distribución de tareas de forma eficiente fue esencial para evitar cuellos de botella y garantizar que todas las partes del proyecto avanzaran de manera sincronizada. En proyectos de esta magnitud, la colaboración y la capacidad de adaptarse a los cambios son cruciales, y este proyecto no fue la excepción.
- **Resolución de Problemas y Manejo de Errores:** A lo largo del desarrollo, nos enfrentamos a varios problemas técnicos que requirieron soluciones innovadoras y una constante iteración. Uno de los mayores desafíos fue la inconsistencia de los datos en Firestore debido a la falta de relaciones nativas, lo que llevó a la duplicación de información y potenciales incoherencias. Para resolver esto, implementamos mecanismos de sincronización en el *backend* para garantizar que los datos entre las diferentes bases de datos estuvieran alineados. También enfrentamos problemas con la latencia de las consultas en PostgreSQL, lo que nos llevó a optimizar las consultas mediante la creación de índices y la implementación de paginación en las consultas de Firestore. Estos problemas nos enseñaron la importancia de la planificación y las pruebas constantes para garantizar un rendimiento eficiente y una integridad de los datos a lo largo de todo el ciclo de vida de la aplicación.
- **Lecciones de Escalabilidad:** La escalabilidad fue uno de los principales factores a tener en cuenta durante el desarrollo de la aplicación. A medida que los datos crecían, era crucial garantizar que las consultas siguieran siendo eficientes. La implementación de índices compuestos en Firestore y la paginación en el frontend ayudaron a mitigar el impacto de las consultas lentas. En PostgreSQL, se optimizaron las consultas clave mediante la creación de índices en columnas específicas, lo que aceleró las búsquedas y mejoró el rendimiento general. Este proceso nos enseñó a evaluar los cuellos de botella y a diseñar soluciones de escalabilidad de manera proactiva, lo que resultará esencial en proyectos más grandes en el futuro.

9. Reflexión Final

Este proyecto ha sido una experiencia invaluable que ha combinado tanto desafíos técnicos como oportunidades de aprendizaje. La integración de tecnologías tan diversas como PostgreSQL y Firestore nos permitió abordar una amplia gama de problemas y encontrar soluciones efectivas para cada uno. Además, nos ha permitido comprender de manera más profunda las características, ventajas y limitaciones de ambas bases de datos, lo que nos proporcionó una perspectiva más amplia sobre cómo elegir la tecnología adecuada para cada caso de uso.

La combinación de habilidades técnicas adquiridas y las lecciones organizacionales aprendidas fortalecerán nuestro enfoque hacia proyectos futuros. Este proyecto no solo ha mejorado nuestra capacidad para manejar arquitecturas complejas, sino que también ha demostrado la importancia de la comunicación, la planificación y la toma de decisiones informadas en cada etapa del desarrollo. Estamos mejor preparados para enfrentar proyectos aún más complejos y desafiantes, con una comprensión más sólida de cómo gestionar y escalar aplicaciones modernas.

Finalmente, esta experiencia nos ha dejado con un fuerte sentido de colaboración y un enfoque más robusto hacia la resolución de problemas. Cada desafío nos permitió crecer como equipo y como profesionales, y estamos listos para aplicar estos aprendizajes en nuestros futuros desarrollos.