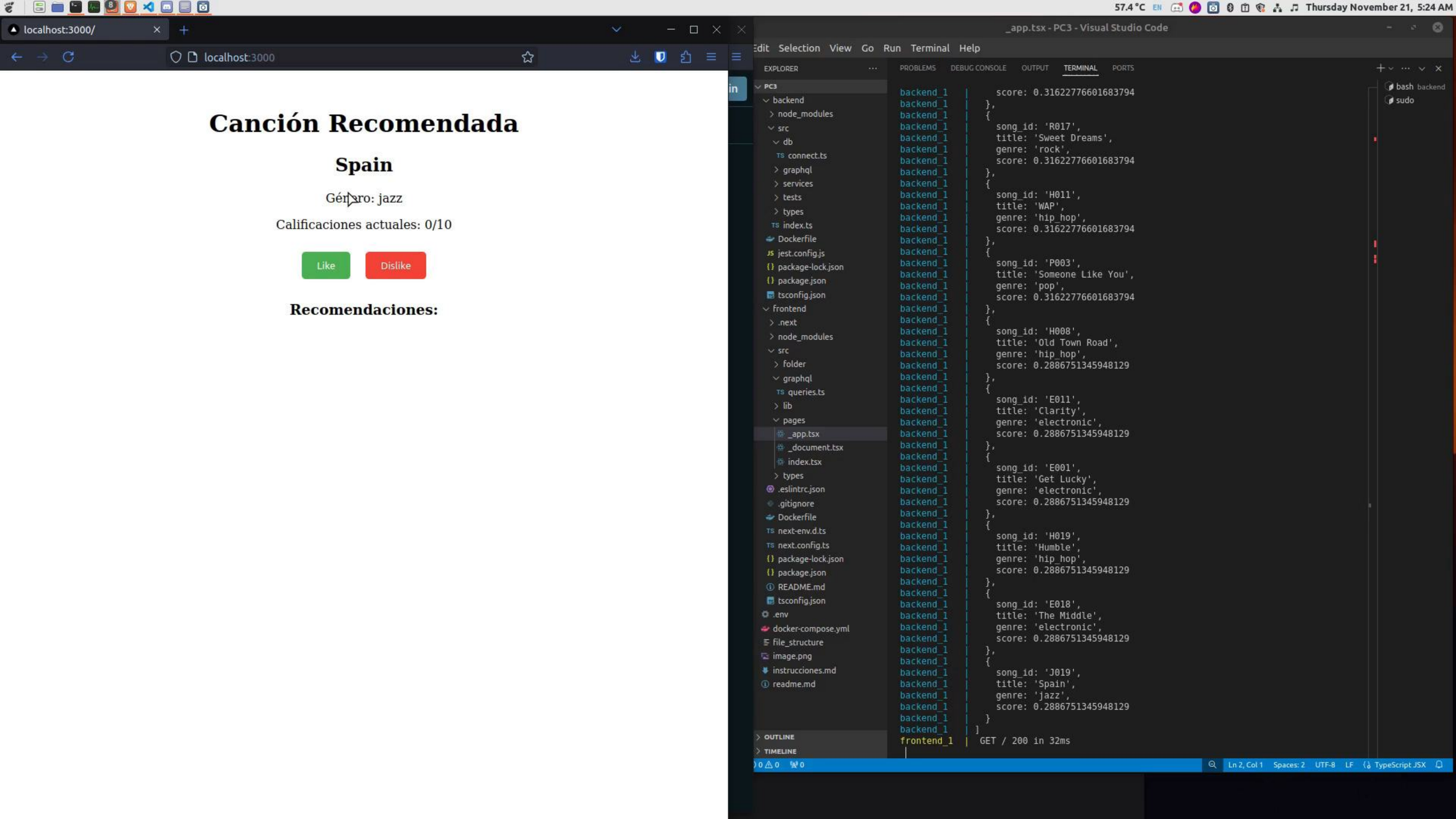
21 NOV 2024

Aplicación de recomendaciones personalizadas usando aprendizaje automático simplificado

Descripción:
Aplicación que ofrezca recomendaciones básicas basadas en un conjunto de datos estático utilizando un algoritmo sencillo de filtrado colaborativo con una biblioteca como simple-recommender.

Requerimientos:
Utiliza Next.js y React en el frontend, y Node.js con TypeScript en el backend.
Emplea MongoDB para almacenar datos de usuarios y preferencias predefinidas.
Implementa una API GraphQL básica que devuelve recomendaciones estáticas.
Escribe pruebas unitarias para el algoritmo de recomendación y pruebas de integración para la API con Jest.
Conteneriza todo con Docker, separando backend y frontend.


localhost:3000/
localhost:3000
Canción Recomendada
Spain
Género: jazz
Calificaciones actuales: 0/10
Like
Dislike
Recomendaciones:
_app.tsx - PC3 - Visual Studio Code
Edit Selection View Go Run Terminal Help
EXPLORER
PROBLEMS DEBUG CONSOLE OUTPUT TERMINAL PORTS
PC3
backend
node_modules
src
db
connect.ts
graphql
services
tests
types
index.ts
Dockerfile
jest.config.js
package-lock.json
package.json
tsconfig.json
frontend
.next
node_modules
src
folder
graphql
queries.ts
lib
pages
_app.tsx
_document.tsx
index.tsx
types
.eslintrc.json
.gitignore
Dockerfile
next-env.d.ts
next.config.ts
package-lock.json
package.json
README.md
tsconfig.json
.env
docker-compose.yml
file_structure
image.png
instrucciones.md
readme.md
OUTLINE
TIMELINE
bash backend
sudo
backend_1 | score: 0.31622776601683794
backend_1 | },
backend_1 | {
backend_1 | song_id: 'R017',
backend_1 | title: 'Sweet Dreams',
backend_1 | genre: 'rock',
backend_1 | score: 0.31622776601683794
backend_1 | },
backend_1 | {
backend_1 | song_id: 'H011',
backend_1 | title: 'WAP',
backend_1 | genre: 'hip_hop',
backend_1 | score: 0.31622776601683794
backend_1 | },
backend_1 | {
backend_1 | song_id: 'P003',
backend_1 | title: 'Someone Like You',
backend_1 | genre: 'pop',
backend_1 | score: 0.31622776601683794
backend_1 | },
backend_1 | {
backend_1 | song_id: 'H008',
backend_1 | title: 'Old Town Road',
backend_1 | genre: 'hip_hop',
backend_1 | score: 0.2886751345948129
backend_1 | },
backend_1 | {
backend_1 | song_id: 'E011',
backend_1 | title: 'Clarity',
backend_1 | genre: 'electronic',
backend_1 | score: 0.2886751345948129
backend_1 | },
backend_1 | {
backend_1 | song_id: 'E001',
backend_1 | title: 'Get Lucky',
backend_1 | genre: 'electronic',
backend_1 | score: 0.2886751345948129
backend_1 | },
backend_1 | {
backend_1 | song_id: 'H019',
backend_1 | title: 'Humble',
backend_1 | genre: 'hip_hop',
backend_1 | score: 0.2886751345948129
backend_1 | },
backend_1 | {
backend_1 | song_id: 'E018',
backend_1 | title: 'The Middle',
backend_1 | genre: 'electronic',
backend_1 | score: 0.2886751345948129
backend_1 | },
backend_1 | {
backend_1 | song_id: 'J019',
backend_1 | title: 'Spain',
backend_1 | genre: 'jazz',
backend_1 | score: 0.2886751345948129
backend_1 | }
backend_1 |]
frontend_1 | GET / 200 in 32ms
Ln 2, Col 1 Spaces: 2 UTF-8 LF TypeScript JSX
57.4 °C
Thursday November 21, 5:24 AM

Creación de un JSON file que contiene rating de 100 usuarios -> cargado a MongoDB Atlas

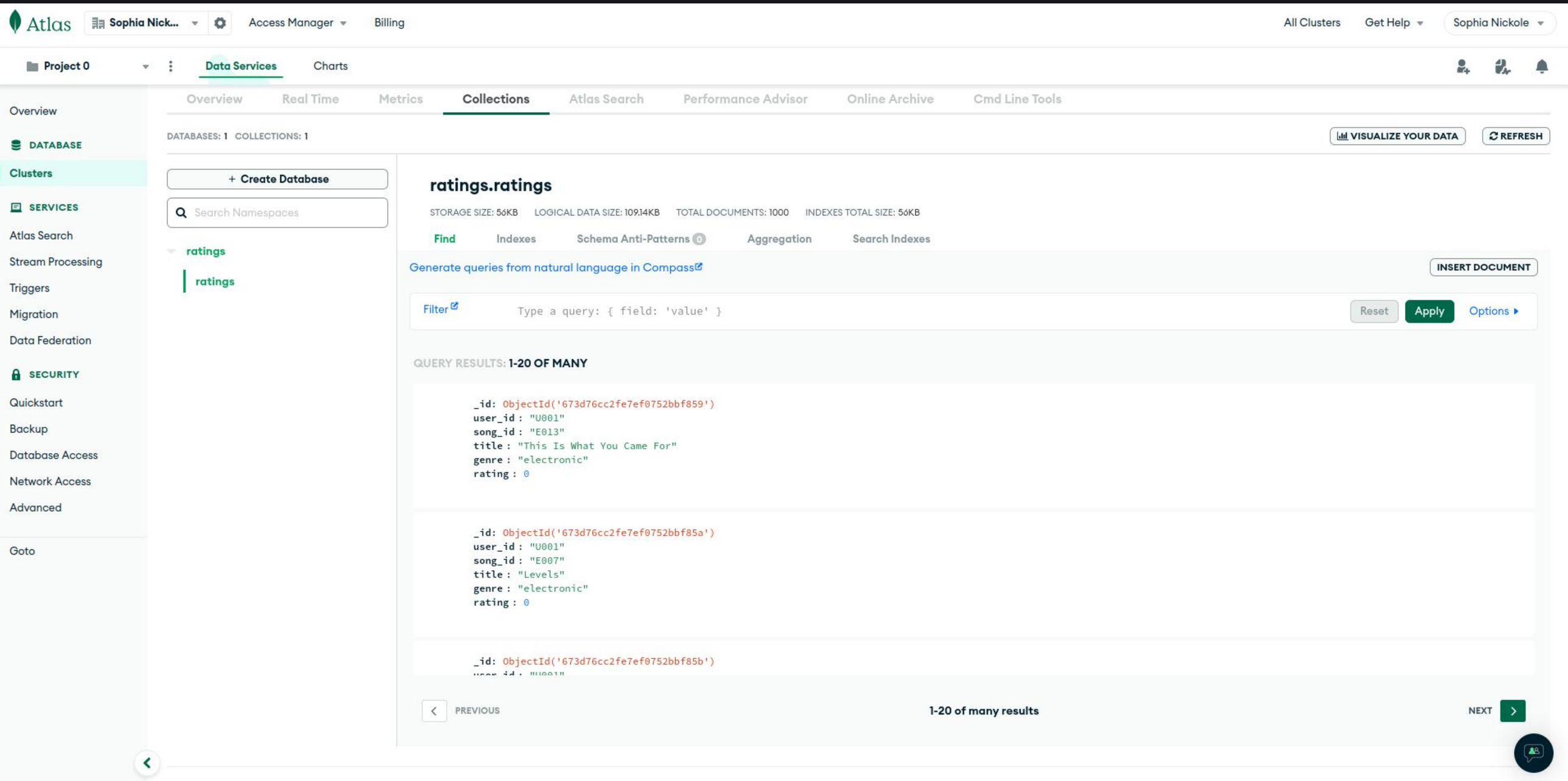
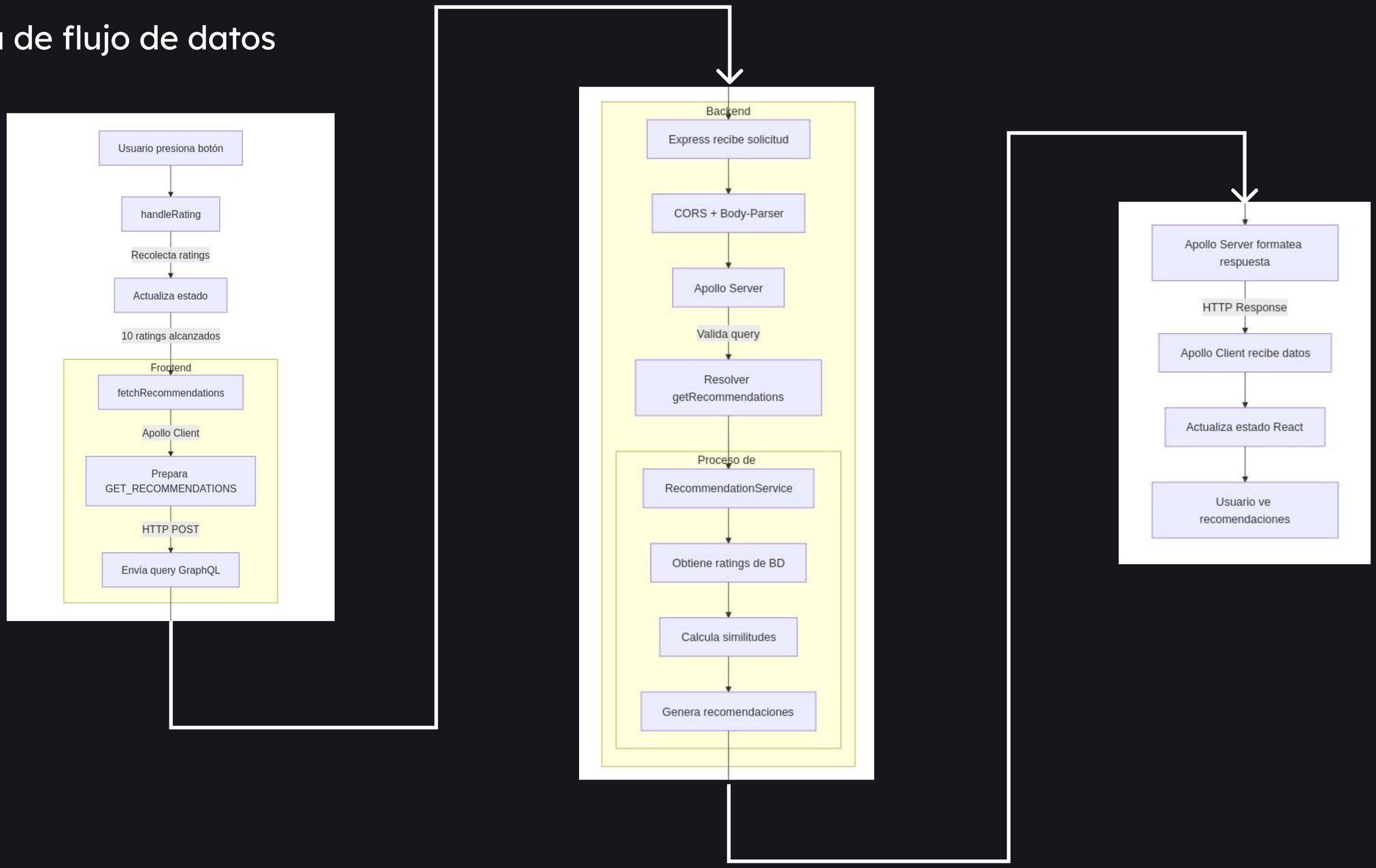
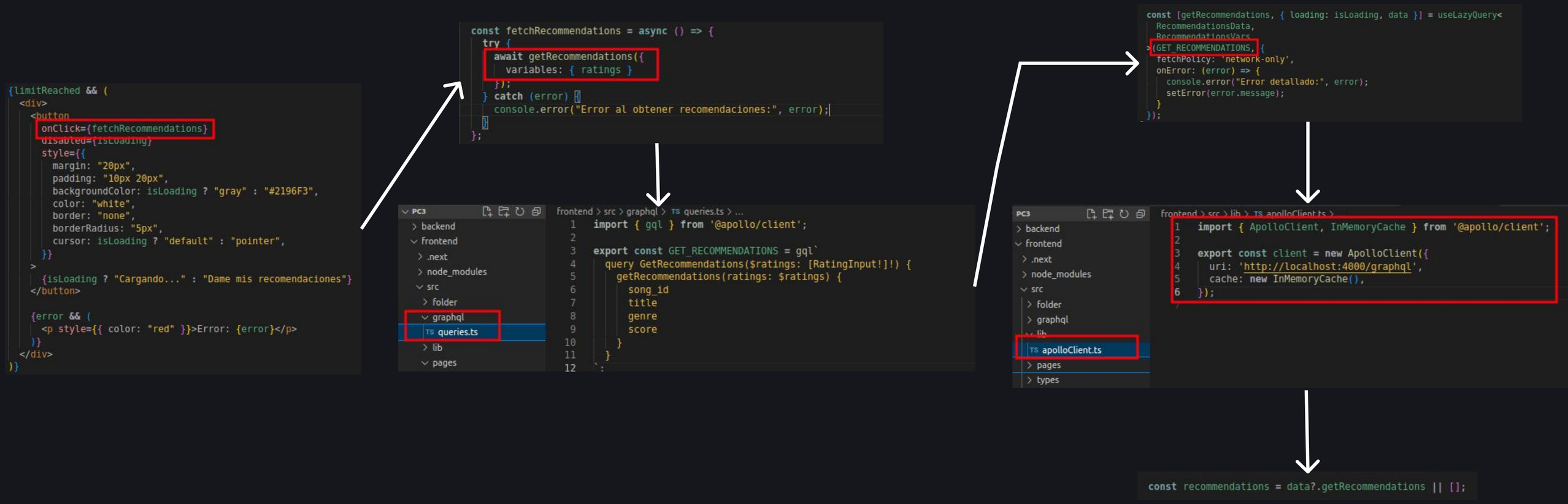


Diagrama de flujo de datos



Al presionar el botón:

- Sr llama a una función (fetchRecommendations)
- Esta función activa la consulta GraphQL definida (GET_RECOMMENDATIONS)
- Se envían las calificaciones al backend, y las recomendaciones resultantes se muestran al usuario.



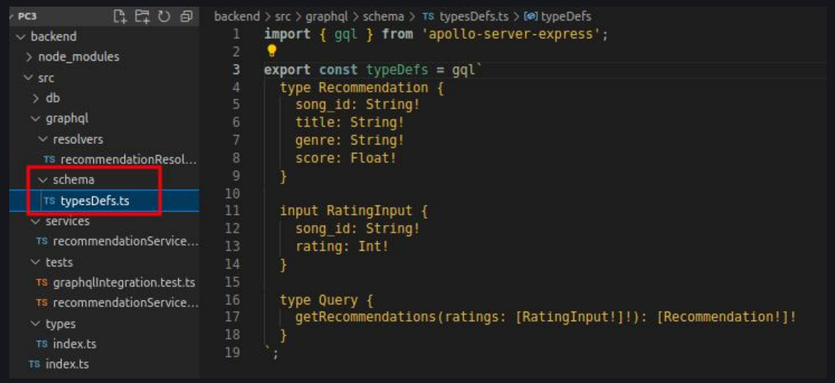
Express recibe la solicitud

```
// Middleware CORS: permite solicitudes desde diferentes dominios (Cross-Origin Res
app.use(cors());
// Middleware Body-Parser: convierte el cuerpo de las solicitudes HTTP a JSON.
app.use(json());

// Ruta GraphQL
app.use(
    '/graphql', // Endpoint donde escuchará GraphQL.
    expressMiddleware(server, {
        context: async () => ({
            recommendationService, // Pasamos el servicio de recomendaciones al cor
        }),
    })
);
```

```
/// Creamos el servidor Apollo para manejar consultas GraphQL.
const server = new ApolloServer({
    typeDefs, // Esquema GraphQL: define las consultas y tipos.
    resolvers, // Resolvers: implementan la lógica para responder consultas.
});

// Iniciar el servidor Apollo
await server.start();
```

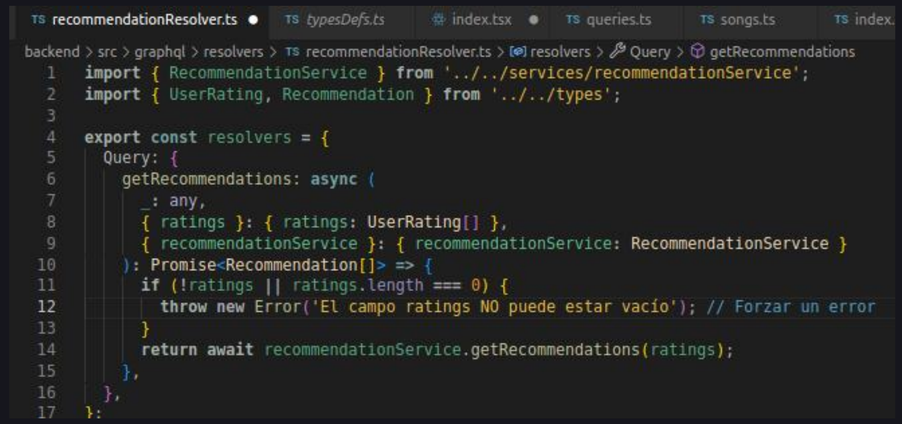
context:

Proporciona datos adicionales (como recommendationService) que estarán disponibles para los resolvers.



backend > src > graphql > schema > typesDefs.ts > typeDefs

```
import { gql } from 'apollo-server-express';

export const typeDefs = gql`
  type Recommendation {
    song_id: String!
    title: String!
    genre: String!
    score: Float!
  }

  input RatingInput {
    song_id: String!
    rating: Int!
  }

  type Query {
    getRecommendations(ratings: [RatingInput!]!): [Recommendation!]!
  }
`;
```




```
import { RecommendationService } from '../../services/recommendationService';
import { UserRating, Recommendation } from '../../types';

export const resolvers = {
  Query: {
    getRecommendations: async (
      _: any,
      { ratings }: { ratings: UserRating[] },
      { recommendationService }: { recommendationService: RecommendationService }
    ): Promise<Recommendation[]> => {
      if (!ratings || ratings.length === 0) {
        throw new Error('El campo ratings NO puede estar vacío'); // Forzar un error
      }
      return await recommendationService.getRecommendations(ratings);
    },
  },
};
```

Resolving...

```
try {
    console.log("1. Ratings recibidos:", ratings);

    const currentUserVector = this.createUserVector(ratings);
    console.log("2. Vector del usuario actual:", currentUserVector);

    const allRatings = await this.getAllRatings();
    console.log("3. Número total de ratings en BD:", allRatings.length);

    const userVectors = this.createUserVectors(allRatings);
    console.log("4. Número de usuarios encontrados:", Object.keys(userVectors).length);

    const similarities = this.calculateSimilarities(currentUserVector, userVectors);
    console.log("5. Top 3 usuarios similares:", similarities.slice(0, 3));

    const ratedSongIds = new Set(ratings.map(r => r.song_id));
    const recommendations = new Map<string, RecommendationScore>();

    // Usar los 5 usuarios más similares
    for (const { userId, similarity } of similarities.slice(0, 5)) {
        if (similarity > 0) {
            const userVector = userVectors[userId];
            Object.entries(userVector).forEach(([songId, rating]) => {
                if (!ratedSongIds.has(songId) && rating > 0) {
                    const current = recommendations.get(songId) || { score: 0, count: 0 };
                    recommendations.set(songId, {
                        score: current.score + (rating * similarity),
                        count: current.count + 1
                    });
                }
            });
        }
```

```
nst recommendedSongs = await Promise.all(
  Array.from(recommendations.entries())
    .map(async ([songId, { score, count }]) => {
        const song = await this.ratingsCollection.findOne({ song_id: songId }) as unknown as RatingDocument | null;
        if (!song) {
            console.log(`No se encontró la canción con song_id: ${songId}`);
            return null;
        }
        return {
            song_id: songId,
            title: song.title,
            genre: song.genre,
            score: score / count
        };
    })
```

```
    const topRecommendations = recommendedSongs
        .filter((song): song is NonNullable<typeof song> => song !== null)
        .sort((a, b) => b.score - a.score)
        .slice(0, 10);

    console.log("7. Recomendaciones finales:", topRecommendations);
    return topRecommendations;
```

```
private createUserVectors(allRatings: RatingDocument[]): Record<string, UserVector> {
    const userVectors: Record<string, UserVector> = {};
    allRatings.forEach(rating => {
        if (rating.user_id && rating.song_id && typeof rating.rating === 'number') {
            if (!userVectors[rating.user_id]) {
                userVectors[rating.user_id] = {};
            }
            userVectors[rating.user_id][rating.song_id] = rating.rating;
        }
    });
    return userVectors;
}
```

```
private async getAllRatings(): Promise<RatingDocument[]> {
    const allRatingsRaw = await this.ratingsCollection.find({}).toArray();
    return allRatingsRaw as unknown as RatingDocument[];
}
```

```
private createUserVector(ratings: UserRating[]): UserVector {
    return ratings.reduce((acc: UserVector, rating: UserRating) => {
        acc[rating.song_id] = rating.rating;
        return acc;
    }, {});
}
```

```
calculateSimilarities(currentUserVector: UserVector, userVectors: Record<string, UserVector>): UserSimilarit
rn Object.entries(userVectors)
.map(([userId, vector]) => ({
    userId,
    similarity: this.calculateCosineSimilarity(currentUserVector, vector)
}))
.sort((a, b) => b.similarity - a.similarity);
```

```
private calculateCosineSimilarity(vector1: UserVector, vector2: UserVector): number {
    const songs = new Set([...Object.keys(vector1), ...Object.keys(vector2)]);
    let dotProduct = 0;
    let norm1 = 0;
    let norm2 = 0;

    songs.forEach(song => {
        const rating1 = vector1[song] || 0;
        const rating2 = vector2[song] || 0;
        dotProduct += rating1 * rating2;
        norm1 += rating1 * rating1;
        norm2 += rating2 * rating2;
    });

    if (norm1 === 0 || norm2 === 0) return 0;
    return dotProduct / (Math.sqrt(norm1) * Math.sqrt(norm2));
}
```

ya se tiene topRecommendations...cómo viaja hasta el frontend?

```ts
import { RecommendationService } from '../../services/recommendationService';
import { UserRating, Recommendation } from '../../types';

export const resolvers = {
  Query: {
    getRecommendations: async (
      _: any,
      { ratings }: { ratings: UserRating[] },
      { recommendationService }: { recommendationService: RecommendationService }
    ): Promise<Recommendation[]> => {
      if (!ratings || ratings.length === 0) {
        throw new Error('El campo ratings NO puede estar vacío'); // Forzar un error
      }
      return await recommendationService.getRecommendations(ratings);
    },
  },
};
```

await indica que Apollo Server espera a que el servicio termine de procesar las recomendaciones.

```ts
/// Creamos el servidor Apollo para manejar consultas GraphQL.
const server = new ApolloServer({
    typeDefs, // Esquema GraphQL: define las consultas y tipos.
    resolvers, // Resolvers: implementan la lógica para responder consultas.
});
```

```ts
// Ruta GraphQL
app.use(
    '/graphql', // Endpoint donde escuchará GraphQL.
    expressMiddleware(server, {
        context: async () => ({
            recommendationService, // Pasamos el servicio de recomendaciones al contexto de Apollo Server.
        }),
    })
);
```

Algunos logs que usé para validar que todo esté yendo bien...

```
frontend_1  |  GET / 200 in 32ms
backend_1   | 1. Ratings recibidos: [
backend_1   |   { song_id: 'J019', rating: 1 },
backend_1   |   { song_id: 'H019', rating: 1 },
backend_1   |   { song_id: 'J012', rating: 0 },
backend_1   |   { song_id: 'J007', rating: 0 },
backend_1   |   { song_id: 'H008', rating: 1 },
backend_1   |   { song_id: 'P001', rating: 1 },
backend_1   |   { song_id: 'P020', rating: 1 },
backend_1   |   { song_id: 'E015', rating: 0 },
backend_1   |   { song_id: 'H013', rating: 1 },
backend_1   |   { song_id: 'J019', rating: 1 }
backend_1   | ]
backend_1   | 2. Vector del usuario actual: {
backend_1   |   J019: 1,
backend_1   |   H019: 1,
backend_1   |   J012: 0,
backend_1   |   J007: 0,
backend_1   |   H008: 1,
backend_1   |   P001: 1,
backend_1   |   P020: 1,
backend_1   |   E015: 0,
backend_1   |   H013: 1
backend_1   | }
backend_1   | 3. Número total de ratings en BD: 1000
backend_1   | 4. Número de usuarios encontrados: 100
backend_1   | 5. Top 3 usuarios similares: [
backend_1   |   { userId: 'U083', similarity: 0.3651483716701107 },
backend_1   |   { userId: 'U086', similarity: 0.3651483716701107 },
backend_1   |   { userId: 'U015', similarity: 0.333333333333333337 }
backend_1   | ]
backend_1   | 6. Número de canciones recomendadas: 15
```

```
(base) terry@terry-pc:~/Desktop/PC3$ sudo docker-compose up
[sudo] password for terry:
Starting pc3_frontend_1 ... done
Starting pc3_backend_1  ... done
Attaching to pc3_frontend_1, pc3_backend_1
backend_1   |
backend_1   | > backend@1.0.0 dev
backend_1   | > ts-node-dev src/index.ts
backend_1   |
frontend_1  |
frontend_1  | > frontend@0.1.0 dev
frontend_1  | > next dev
frontend_1  |
backend_1   | [INFO] 10:31:54 ts-node-dev ver. 2.0.0 (using ts-node ver. 10.9.2, typesc
ript ver. 5.6.3)
frontend_1  |    ▲ Next.js 15.0.3
frontend_1  |    - Local:        http://localhost:3000
frontend_1  |
frontend_1  |  ✓ Starting...
backend_1   | (node:25) [MONGODB DRIVER] Warning: useNewUrlParser is a deprecated optio
n: useNewUrlParser has no effect since Node.js Driver version 4.0.0 and will be removed
 in the next major version
backend_1   | (Use `node --trace-warnings ...` to show where the warning was created)
backend_1   | (node:25) [MONGODB DRIVER] Warning: useUnifiedTopology is a deprecated op
tion: useUnifiedTopology has no effect since Node.js Driver version 4.0.0 and will be r
emoved in the next major version
frontend_1  |  ✓ Ready in 1601ms
backend_1   | MongoDB connected successfully
backend_1   | Colecciones disponibles: [ 'ratings', 'ratings.ratings' ]
backend_1   | 🚀 Server listo ennn http://localhost:4000/graphql
frontend_1  |  ○ Compiling /_error ...
frontend_1  |  ✓ Compiled /_error in 2s (557 modules)
frontend_1  |  GET /_next/static/webpack/f3795284bd77f521.webpack.hot-update.json 404 i
n 2188ms
frontend_1  |  ⚠ Fast Refresh had to perform a full reload. Read more: https://nextjs.o
rg/docs/messages/fast-refresh-reload
frontend_1  |  ✓ Compiled / in 134ms (566 modules)
frontend_1  |  GET / 200 in 172ms
frontend_1  |  POST /api/graphql 404 in 28ms
frontend_1  |  POST /api/graphql 404 in 9ms
frontend_1  |  POST /api/graphql 404 in 7ms
frontend_1  |  POST /api/graphql 404 in 7ms
frontend_1  |  POST /api/graphql 404 in 7ms
frontend_1  |  POST /api/graphql 404 in 4ms
frontend_1  |  POST /api/graphql 404 in 8ms
frontend_1  |  POST /api/graphql 404 in 6ms
frontend_1  |  POST /api/graphql 404 in 7ms
frontend_1  |  POST /api/graphql 404 in 5ms
frontend_1  |  POST /api/graphql 404 in 6ms
frontend_1  |  GET /api/graphql 404 in 6ms
frontend_1  |  GET /api/graphql 404 in 4ms
frontend_1  |  GET / 200 in 18ms
```

Pruebas unitarias para el algoritmo de recomendación

```
Test Suites: 2 passed, 2 total
Tests:       8 passed, 8 total
Snapshots:   0 total
Time:        3.159 s
Ran all test suites.
```

TESTING!!

Queremos probar el servicio de recomendaciones para:
La conversion de calificaciones en vectores
El calculo de similitudes
La generacion de recomendaciones
Ls interacción con MongoDB : consulta de datos y transformacion de datos en recomendaciones
Qué pasa si no hay calificaciones?
Qué pasa si la base de datos no devuelve nada?

Estructura General del archivo de Test

```
> db                                                    Aa ab .* ? of 5
import { RecommendationService } from '../services/recommendationService';
import { Collection } from 'mongodb';

describe('RecommendationService', () => {
  // Mock de la colección de MongoDB
  const mockCollection = {
    find: jest.fn(),
    findOne: jest.fn(),
  } as unknown as Collection;

  const recommendationService = new RecommendationService(mockCollection);
```

Test de Similitud Coseno para Vectores Idénticos

```
describe('calculateCosineSimilarity', () => {
  // Acceder al método privado para testing
  const calculateCosineSimilarity = (recommendationService as any).calculateCosineSimilarity.bind(re

  test('debería retornar 1 para vectores idénticos', () => {
    const vector1 = { 'song1': 1, 'song2': 0, 'song3': 1 };
    const vector2 = { 'song1': 1, 'song2': 0, 'song3': 1 };

    const similarity = calculateCosineSimilarity(vector1, vector2);

    expect(similarity).toBeCloseTo(1, 10); // Adjusted for floating-point precision
  });
```

Esperado: La similitud entre vectores idénticos debe ser 1.

Test de Similitud Coseno para Vectores Completamente Diferentes

```
test('debería retornar 0 para vectores completamente diferentes', () => {
  const vector1 = { 'song1': 1, 'song2': 1 };
  const vector2 = { 'song3': 1, 'song4': 1 };

  const similarity = calculateCosineSimilarity(vector1, vector2);

  expect(similarity).toBe(0);
});
```

Esperado: La similitud entre estos vectores debe ser 0.

Pruebas de integración para la API con Jest.

```
Test Suites: 2 passed, 2 total
Tests:       8 passed, 8 total
Snapshots:   0 total
Time:        3.159 s
Ran all test suites.
```

Apollo Server procesa correctamente las consultas:

- ¿El servidor entiende y valida las consultas GraphQL?
- ¿Se ejecutan los resolvers correctos?

Los datos viajan correctamente a través del backend:

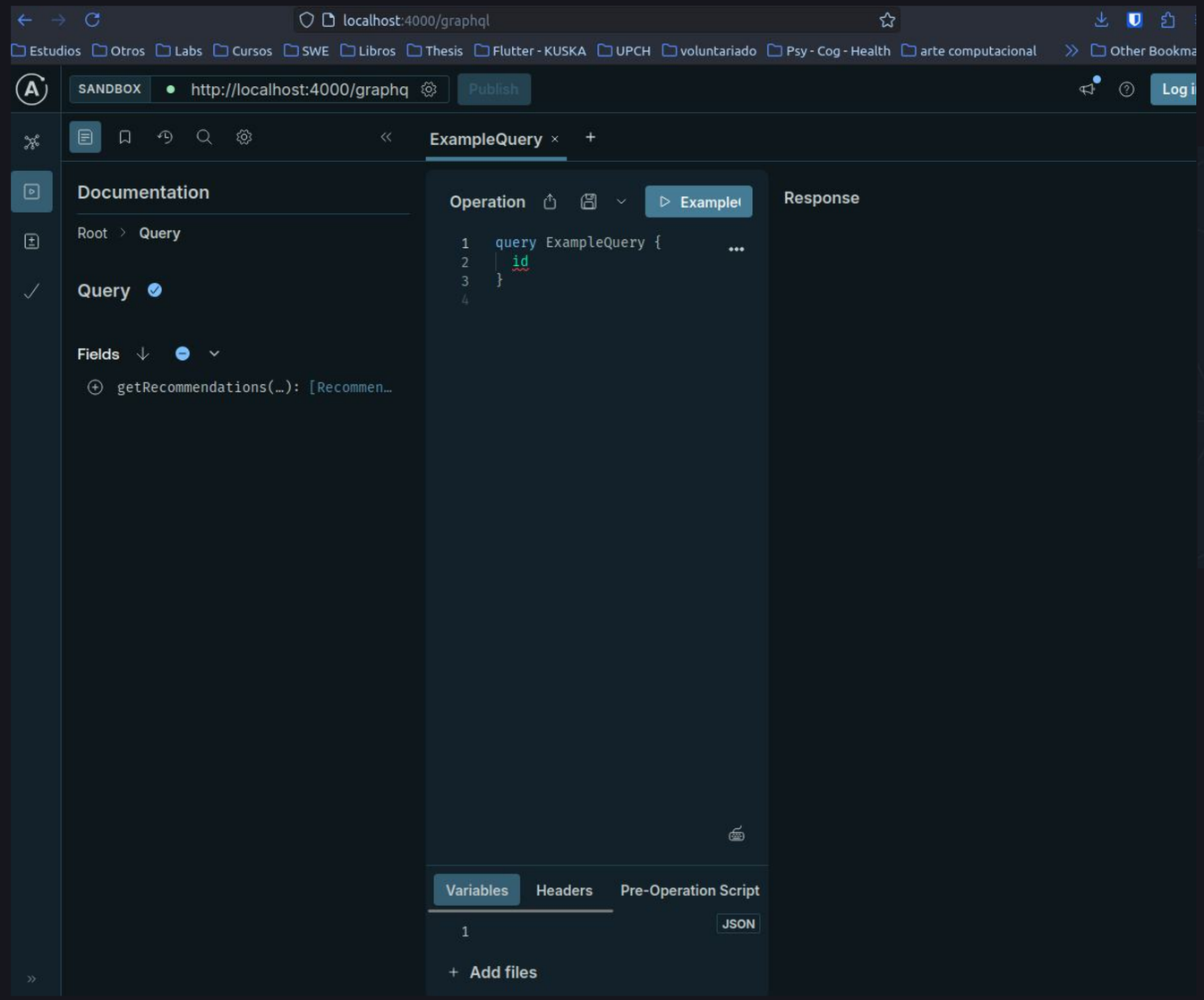
- Desde el esquema (typeDefs).
- A través de los resolvers.
- Hasta los servicios (mockeados para estas pruebas).

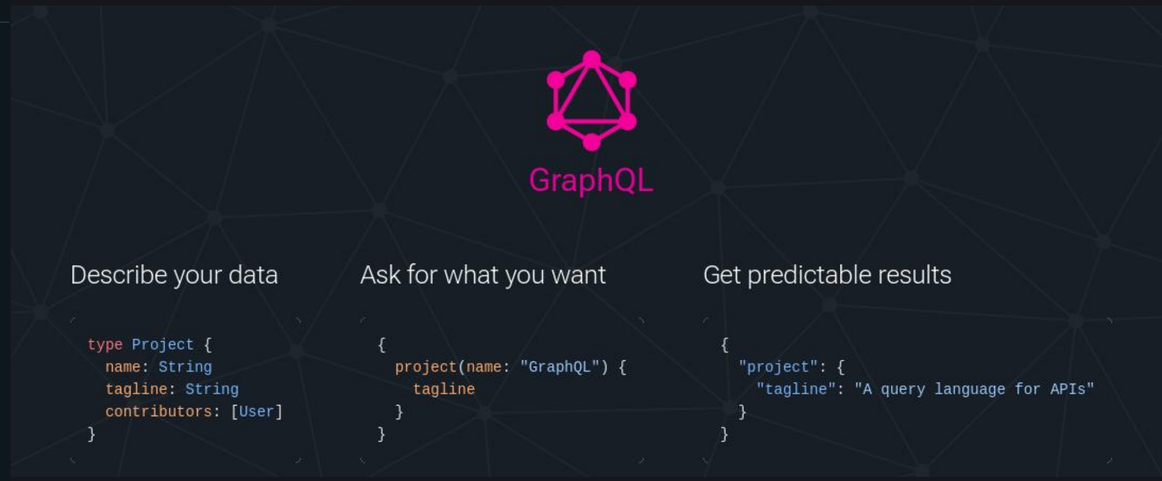
La respuesta generada es correcta:

- ¿El formato de los datos devueltos coincide con lo esperado?

```
describe('GraphQL API Integration Tests', () => {
  it('should return recommendations for valid ratings input', async () => {
    const query = `
      query GetRecommendations($ratings: [RatingInput!]!) {
        getRecommendations(ratings: $ratings) {
          song_id
          title
          genre
          score
        }
      }
    `;

    const variables = {
      ratings: [
        { song_id: 'S1', rating: 1 },
        { song_id: 'S2', rating: 0 },
      ],
    };

    const response = await request(app)
      .post('/graphql')
      .send({ query, variables });

    expect(response.status).toBe(200); // Verificar que el estado HTTP sea 200
    expect(response.body.data).toBeDefined();
    expect(response.body.data.getRecommendations).toHaveLength(1);
    expect(response.body.data.getRecommendations[0]).toEqual({
      song_id: 'S3',
      title: 'Song 3',
      genre: 'pop',
      score: expect.any(Number),
    });
  });
});
```

Imagenes que podrian ayudar





A query language for your API

GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools.