Práctica 4.2: Gestión con mongodb: sinopsis dinámicas de películas

Objetivo

Aprender a conectar el backend de nuestro proyecto CineVerse con una base de datos MongoDB, para que las películas ya no estén solo en un .json, sino en una base de datos real en la nube.

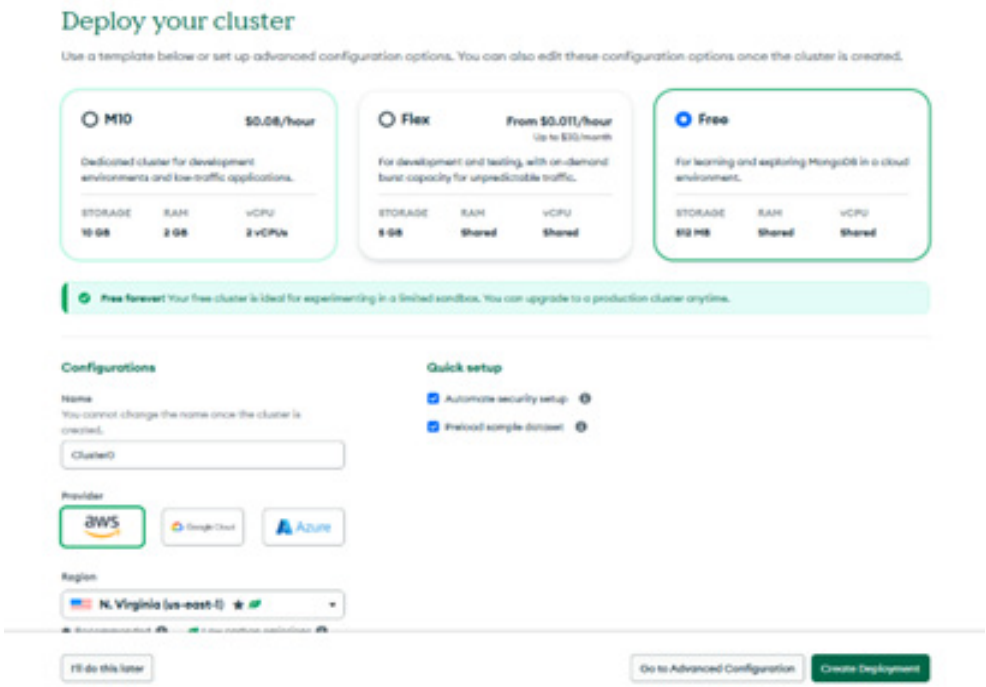
¿Qué haremos?

- Instalar MongoDB y conectar desde Node.js (backend)
- Mover las películas del archivo JSON a la base de datos
- Mostrar esas películas en React como antes
- Agregar posibilidad de agregar o editar sinopsis

PASOS

Paso 1: Registrarse en MongoDB Atlas.

- Ve a https://www.mongodb.com/cloud/atlas
- Da clic en "Start Free".
- Regístrate con tu cuenta de Google o correo electrónico.
- Cuando te pregunte el tipo de cluster, elige la opción gratuita (Shared Cluster).



- ## Crear un Cluster

  - Nombra tu cluster como: cineverseCluster
  - Elige la región más cercana (ej. North America - AWS - us-east-1)
  - Espera a que termine de crearse (tarda unos minutos)

- ## Crear una Base de Datos y Colección

- MongoDB se organiza así:
- Cluster > Base de datos > Colección > Documentos

  - Ve a la pestaña "Database"
  - Da clic en “Browse Collections”
  - Luego clic en “+ Add My Own Data”
  - Crea:
    - Database Name: cecyflix
    - Collection Name: peliculas
  - Luego te dejará añadir documentos, en este caso, agrega nuestro archivo peliculas.json.

Obtener URI de conexión

- Ve a tu cluster > botón "Connect"
- Elige "Connect your application"
- Copia la URI que se ve como:

mongodb+srv://<usuario>:<contraseña>@<cluster>.mongodb.net/cineverse?retryWrites=true&w=majority (URL DE EJEMPLO)

Paso 2: Conectar mongodb en node.js (backend).

Instalar dependencias

- Ingresamos a nuestra carpeta de backend con cd backend y escribimos el siguiente comando en la terminal de VS Code:

npm install mongoose

Archivo .env

- Escribe el siguiente código:

```
MONGO_URI=mongodb+srv://<usuario>:<contraseña>@<cluster>.mongodb.net/cineverse?retryWrites=true&w=majority
```

- Reemplaza <usuario>, <contraseña> y <cluster> con tus datos reales.

Nota: Este archivo nunca debe subirse a GitHub.

Paso 3: Crear el modelo de películas.

Esquema de carpetas en nuestro proyecto

- Creamos una nueva carpeta dentro de backend llamada "models" y dentro un archivo llamado "Pelicula.js".
- Creamos una nueva carpeta dentro de backend llamada "routers" y dentro un archivo llamado "peliculas.js"
- Creamos una nueva carpeta dentro de frontend/src llamada "components" y dentro un archivo llamado "FormularioAgregar.js".

```
recomendaciones-ia/
  backend/
    index.js            // Servidor con Express y conexión MongoDB
    models/
      Pelicula.js       // Esquema de película en MongoDB
    routes/
      peliculas.js      // Rutas para GET, POST, PUT, etc.
    .env                // Conexión MongoDB Atlas
  frontend/ (src)
    App.js              // Ahora obtiene películas desde backend/Mongo
    components/
      FormularioAgregar.js  // Componente para agregar/editar
    data/               // Ya no se usa directamente
App.css
```

- ## Código de Pelicula.js

- Este código define un modelo de datos para películas usando Mongoose, que es una librería de MongoDB para Node.js. Vamos a analizarlo línea por línea:

```
const mongoose = require('mongoose');
```

➡ Importa la librería Mongoose que nos permite interactuar con MongoDB de manera más sencilla.

➡ Mongoose proporciona una capa de abstracción sobre MongoDB con funcionalidades adicionales como validaciones, middlewares y métodos de instancia.

```
const peliculaSchema = new mongoose.Schema({
```

➡ Crea un nuevo esquema de Mongoose llamado peliculaSchema.

➡ Un esquema en Mongoose define la estructura de los documentos que se guardarán en la colección de MongoDB.

```
  titulo: { type: String, required: true },
```

➡ Define el campo titulo con las siguientes características:

- type: String: Indica que será de tipo texto (cadena de caracteres)
- required: true: Hace que este campo sea obligatorio (no se puede guardar un documento sin este campo)
- Esto generará un error de validación si se intenta guardar una película sin título

```
  genero: String,
```

➡ Define el campo genero como tipo String (texto), pero sin ser obligatorio.

➡ Al no tener required: true, este campo puede omitirse al crear una película.

```
descripcion: String,
```

➡ Similar al anterior, define el campo descripcion como String opcional.

```
poster: String
```

➡ Define el campo poster (que probablemente contendrá una URL a la imagen del póster) como String opcional.

```
});
```

➡ Cierra la definición del esquema.

```
module.exports = mongoose.model('Pelicula', peliculaSchema);
```

➡ Crea un modelo llamado 'Pelicula' basado en el esquema definido.

➡ El modelo es lo que realmente usaremos para interactuar con la base de datos.

➡ mongoose.model() toma dos parámetros:

- El nombre singular del modelo (Mongoose lo pluralizará para nombrar la colección en MongoDB)
- El esquema que define la estructura de los documentos

➡ module.exports exporta el modelo para que pueda ser importado y usado en otras partes de la aplicación (como en controladores o rutas).

¿Qué permite hacer este modelo?

1. Validación automática: Mongoose validará que el título esté presente y que todos los campos sean del tipo correcto antes de guardar en la base de datos.
2. Métodos útiles: El modelo tendrá métodos como .save(), .find(), .findById(), etc. para interactuar con la base de datos.
3. Consistencia: Garantiza que todos los documentos en la colección "peliculas" (Mongoose pluraliza el nombre) tengan la misma estructura básica.
4. Seguridad: Previene la inserción de documentos con estructuras no deseadas en la base de datos.

CÓDIGO COMPLETO DE PELICULA.JS

```
const mongoose = require('mongoose');

// Definimos el esquema de una película
const peliculaSchema = new mongoose.Schema({
  titulo: { type: String, required: true },
  genero: String,
  descripcion: String,
  poster: String
});

// Exportamos el modelo para usarlo en otras partes del backend
module.exports = mongoose.model('Pelicula', peliculaSchema);
```

Paso 4: Crear rutas de películas.

- **Este código define un router de Express para manejar las operaciones relacionadas con películas. Vamos a analizarlo línea por línea:**

```
const express = require('express');
```

➡ **Importa el framework Express que permite crear servidores web y rutas en Node.js.**

```
const router = express.Router();
```

➡ **Crea un nuevo objeto Router de Express.**

➡ **El router permite definir rutas específicas que luego pueden ser montadas en la aplicación principal.**

```
const Pelicula = require('../models/Pelicula');
```

➡ **Importa el modelo de Película que definimos anteriormente (en el archivo ../models/Pelicula.js).**

➡ **Este modelo nos permitirá interactuar con la colección de películas en la base de datos MongoDB.**

Ruta para obtener todas las películas

```
router.get('/', async (req, res) => {
```

- Define una ruta GET en la raíz ('/') del router.
- Usa una función asíncrona (async) porque hará operaciones con la base de datos que devuelven Promesas.
- req (request) contiene la petición HTTP recibida.
- res (response) permite enviar la respuesta al cliente.

```
try {
```

- Inicia un bloque try-catch para manejar posibles errores al interactuar con la base de datos.

```
const peliculas = await Pelicula.find();
```

- Usa el método find() del modelo Pelicula para obtener todos los documentos de la colección.
- await espera a que la operación asíncrona se complete antes de continuar.
- Si no se pasan parámetros a find(), devuelve todos los documentos.

```
res.json(peliculas);
```

- Envía la respuesta al cliente en formato JSON con el array de películas obtenido.
- Express convierte automáticamente el array de JavaScript a JSON.

```
} catch (error) {
```

- Captura cualquier error que ocurra en el bloque try.

```
res.status(500).json({ mensaje: 'Error al obtener películas' });
```

- Si hay un error:
  - Establece el código de estado HTTP 500 (Error interno del servidor)
  - Envía un mensaje de error en formato JSON

```
module.exports = router;
```

- Exporta el router para que pueda ser importado y usado en otros archivos (normalmente en el archivo principal de la aplicación).

¿Cómo se usaría este router?

- En la aplicación principal (index.js), lo importamos y montarías así:

```
const peliculasRouter = require('./routes/peliculas');
app.use('/peliculas', peliculasRouter);
```

- Esto haría que:
  - Una petición GET a /peliculas/ ejecutaría este código
  - Una petición POST a /peliculas/ (si estuviera definida) también usaría este router
- Funcionalidad completa
- Este endpoint permite:
  - Obtener todas las películas almacenadas en la base de datos
  - Manejar errores adecuadamente si algo falla
  - Devolver los datos en formato JSON listos para ser consumidos por el frontend
- Es un ejemplo básico que podría extenderse con:
  - Paginación para no devolver todas las películas a la vez
  - Filtrado por género, año, etc.
  - Ordenación de resultados
  - Búsqueda por título
- Creamos una ruta tipo REST API que responde con todas las películas almacenadas. Usamos Pelicula.find() para obtenerlas desde MongoDB.
- **CÓDIGO COMPLETO PELICULAS.JS**

```
const express = require('express');
const router = express.Router();
const Pelicula = require('../models/Pelicula');

// Ruta para obtener todas las películas
router.get('/', async (req, res) => {
  try {
    const peliculas = await Pelicula.find();
    res.json(peliculas);
  } catch (error) {
    res.status(500).json({ mensaje: 'Error al obtener películas' });
  }
});

module.exports = router;
```

Paso 5: Conectar todo en index.js

- Archivo: backend/index.js
- Si ya tienes algo de OpenRouter, agrégalo al final, esto no afecta.

```
const express = require('express');
const cors = require('cors');
const mongoose = require('mongoose');
const peliculasRouter = require('./routes/peliculas');
require('dotenv').config();

const app = express();

app.use(cors());
app.use(express.json());

// Conectar a MongoDB
mongoose.connect(process.env.MONGO_URI)
  .then(() => console.log('✅ Conectado a MongoDB Atlas'))
  .catch((err) => console.error('❌ Error al conectar a MongoDB:', err));

// Usar la ruta de películas
app.use('/api/peliculas', peliculasRouter);

// (Tu ruta de /api/recomendaciones puede ir aquí también)

app.listen(4000, () => console.log('🚀 Backend corriendo en http://localhost:4000'));
```

Paso 6: conectar el frontend a mongodb (vía backend).

- Eliminaremos la dependencia del archivo peliculas.json
- Usaremos useEffect para cargar las películas desde la API http://localhost:4000/api/peliculas
- Manejaremos los estados de React (useState) para mostrar las películas filtradas

- Antes de comenzar
- Asegúrate de tener:
  - Tu backend corriendo: node index.js en la carpeta backend
  - Tu frontend abierto: npm start en la carpeta cineverse o frontend

Archivo a modificar: App.js adaptado a MongoDB

- Este es el componente principal de la aplicación React que maneja la interfaz de usuario y la lógica de búsqueda. Vamos a analizarlo en detalle:
- Reemplaza el contenido actual de App.js con este:

Estados del componente

```
const [peliculas, setPeliculas] = useState([]);
```

➡ Almacena la lista completa de películas obtenidas de la API.

```
const [peliculasFiltradas, setPeliculasFiltradas] = useState([]);
```

➡ Almacena las películas filtradas según la búsqueda.

```
const [busqueda, setBusqueda] = useState('');
```

➡ Guarda el texto ingresado en el campo de búsqueda.

```
const [modoDescripcion, setModoDescripcion] = useState(false);
```

➡ Controla si el modo de búsqueda es por descripción (IA) o por título/género.

```
const [recomendacion, setRecomendacion] = useState('');
```

➡ Almacena la recomendación generada por la IA.

Efectos secundarios

```
useEffect(() => {
  fetch('/api/peliculas')
    .then(res => res.json())
    .then(data => {
      setPeliculas(data);
      setPeliculasFiltradas(data);
    })
    .catch(err => console.error('Error al obtener películas:', err));
}, []);
```

➡ Se ejecuta al montar el componente (array de dependencias vacío).

➡ Obtiene todas las películas del backend y actualiza ambos estados.

Funciones de búsqueda

```
const handleBuscar = (e) => {
  e.preventDefault();
  const texto = busqueda.toLowerCase();

  const resultado = peliculas.filter(p =>
    p.titulo.toLowerCase().includes(texto) ||
    p.genero.toLowerCase().includes(texto) ||
    p.titulo.toLowerCase().startsWith(texto)
  );

  setPeliculasFiltradas(resultado);
  setRecomendacion('');
};
```

- Filtra películas por título o género.
- Compara con includes y startsWith para mayor flexibilidad.

Búsqueda por IA

```
const handleBuscarPorDescripcion = async () => {
  try {
    const res = await fetch('/api/recomendaciones', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ prompt: `Dame una recomendación basada en esta descripción:
      ${busqueda}. Usa solo películas de este catálogo:
      ${peliculas.map(p => p.titulo).join(', ')}.` })
    });

    const data = await res.json();
    setRecomendacion(data.recomendacion);

    const seleccionadas = peliculas.filter(p =>
      data.recomendacion.toLowerCase().includes(p.titulo.toLowerCase())
    );

    if (seleccionadas.length > 0) {
      setPeliculasFiltradas(seleccionadas);
    }
  } catch (err) {
    console.error('Error con IA:', err);
  }
};
```

- Envía la descripción al backend para obtener recomendaciones de IA.
- Filtra películas cuyo título aparezca en la recomendación.

Renderizado (JSX): Título y formulario de búsqueda

```jsx
<form className="buscador" onSubmit={handleBuscar}>
  <input
    type="text"
    placeholder={modoDescripcion ? 'Describe la peli que buscas...' :
      'Busca por título o género'}
    value={busqueda}
    onChange={e => setBusqueda(e.target.value)}
  />
  {/* Botones condicionales según el modo */}
</form>
```

- Muestra placeholder diferente según el modo de búsqueda.
- Alterna entre botones de búsqueda tradicional y por IA.

Bloque de recomendación IA

```jsx
{recomendacion && (
  <div className="bloque-recomendaciones">
    <h2>IA sugiere:</h2>
    <p>{recomendacion}</p>
  </div>
)}
```

- Muestra la recomendación solo cuando existe.

Galería de películas

```jsx
<div className="grid">
  {peliculasFiltradas.map((p, i) => (
    <div className="tarjeta" key={i}>
      <img src={p.poster} alt={p.titulo} />
      <div className="info">
        <h3>{p.titulo}</h3>
        <p>{p.genero}</p>
        <span>{p.descripcion?.slice(0, 60)}...</span>
      </div>
    </div>
  ))}
</div>
```

- Muestra tarjetas para cada película filtrada.
- Usa slice para limitar la descripción a 60 caracteres.

Características clave

- Búsqueda dual: Implementa dos modos de búsqueda (tradicional y por IA)
- Interfaz responsive: Usa CSS Grid para el diseño de tarjetas
- Manejo de errores: Captura errores en las peticiones a la API
- Feedback visual: Muestra claramente los resultados de búsqueda y recomendaciones

Flujo completo

- El usuario ingresa un término de búsqueda
- Según el modo seleccionado:
  - Busca coincidencias directas en títulos/géneros
  - O consulta a la IA para recomendaciones basadas en descripción
- Muestra los resultados filtrados y cualquier recomendación de IA
- Permite alternar entre modos de búsqueda fácilmente

Paso 7: Ejecución.

Ejecución de backend

En la terminal que usaremos para backend, navegamos a nuestra carpeta cd backend y una vez que nos encontremos en la carpeta de backend escribe:

```
node index.js
```

Ejecución de frontend

Por defecto nos situará en la carpeta raíz, ahí es donde debemos estar, en caso de no encontrarnos en la carpeta recomendaciones-ia, debemos navegar o regresar hasta ella. Una vez dentro de la carpeta escribimos el siguiente comando:

```
npm start
```

Se abrirá el navegador en: http://localhost:3000

Ahora las peliculas serán cargadas directamente de MongoDB Cloud.