Diplomatura en

FullStack Developer

Backend con Node.js
Esquemas en bases de datos NO-SQL

Propiedades de los Esquemas

Los esquemas en MongoDB pueden tener una variedad de propiedades que permiten definir la estructura y validar los datos de una colección. Algunas de las propiedades más comunes incluyen:

1.Tipo de Datos:

1. Define el tipo de datos que se espera para un campo específico, como String, Number, Date, Array, Object, etc.

2.Validaciones:

1. Permite especificar reglas de validación para los campos, como si son requeridos, su longitud mínima/máxima, si deben ser únicos, etc.

3.Opciones Adicionales:

1. Proporciona opciones adicionales para personalizar el comportamiento de los campos, como valores predeterminados (default), si deben ser seleccionados (select), si deben recortarse (trim), etc.

Tipos de Datos:

1.String:
- Almacena datos de texto, como nombres, descripciones, etc.
- Ejemplo: "John Doe"

2.Number:
- Almacena datos numéricos, ya sea enteros o decimales.
- Ejemplo: 42, 3.14

3.Date:
- Almacena fechas y horas.
- Ejemplo: 2024-05-24T12:00:00Z

4.Boolean:
- Almacena valores verdaderos o falsos.
- Ejemplo: true, false

5.Array:
- Almacena una lista ordenada de valores.
- Ejemplo: ["apple", "banana", "orange"]

6.Object:
- Almacena datos estructurados en forma de objetos JSON.
- Ejemplo: { "name": "John", "age": 30 }

Validaciones:

1.Requerido (required):

- Indica que un campo debe tener un valor.
- Ejemplo: { name: { type: String, required: true } }

2.Longitud Mínima/Máxima (minlength, maxlength):

- Establece la longitud mínima o máxima permitida para un campo de tipo String.
- Ejemplo: { password: { type: String, minlength: 6, maxlength: 20 } }

3.Valores Únicos (unique):

- Garantiza que no haya valores duplicados en un campo.
- Ejemplo: { email: { type: String, unique: true } }

4.Expresiones Regulares (match):

- Valida un campo según una expresión regular.
- Ejemplo: { email: { type: String, match: /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$/ } }

5.Valores por Defecto (default):

- Asigna un valor predeterminado al campo si no se proporciona ningún valor durante la creación del documento.
- Ejemplo: { status: { type: String, default: "activo" } }

Opciones Adicionales en los Esquemas:

1.Select (select):

- Controla si el campo es seleccionable en las consultas.
- Ejemplo: { status: { type: String, select: false } }

2.Trim (trim):

- Elimina los espacios en blanco al principio y al final de un campo de tipo String.
- Ejemplo: { username: { type: String, trim: true } }

3.Lowercase (lowercase):

- Convierte automáticamente los valores de tipo String en minúsculas.
- Ejemplo: { email: { type: String, lowercase: true } }

4.Uppercase (uppercase):

- Convierte automáticamente los valores de tipo String en mayúsculas.
- Ejemplo: { role: { type: String, uppercase: true } }

5.Index (index):

- Crea un índice en el campo para mejorar el rendimiento de las consultas.
- Ejemplo: { email: { type: String, index: true } }

Ejemplo:

```
import mongoose from 'mongoose';
const productSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true,
        trim: true
    },
    description: {
        type: String,
        trim: true
    },
    price: {
        type: Number,
        required: true,
        min: 0
    },
    category: {
        type: String,
        enum: ['Electronics', 'Clothing', 'Books', 'Home & Kitchen']
    },
    quantity: {
        type: Number,
        default: 0
    },
    createdAt: {
        type: Date,
        default: Date.now
    }
});
const Product = mongoose.model('Product', productSchema);
export default Product;
```

Métodos para Obtener Datos con Mongoose

1.find(query, projection, options):
Este método busca documentos en base a un conjunto de criterios dados por la consulta (query). Puedes especificar qué campos se desean incluir o excluir utilizando la proyección (projection) y ajustar el comportamiento de la búsqueda con las opciones (options) como límite, orden, etc.

2.findOne(query, projection, options):
3.Similar a find(), pero solo devuelve el primer documento que cumpla con los criterios de búsqueda.

3.findById(id):
Encuentra un documento por su ID único.

4.create(data):
Crea un nuevo documento en la colección basado en los datos proporcionados.

5.updateOne(filter, update, options):
Actualiza un solo documento que cumpla con los criterios de filtrado (filter) con los nuevos datos proporcionados (update). Puedes ajustar el comportamiento de la actualización con las opciones (options).

6.updateMany(filter, update, options):
Actualiza varios documentos que cumplan con los criterios de filtrado (filter) con los nuevos datos proporcionados (update). También puedes ajustar el comportamiento con las opciones (options).

7.deleteOne(filter):
Elimina un solo documento que cumpla con los criterios de filtrado (filter).

8.deleteMany(filter):
Elimina varios documentos que cumplan con los criterios de filtrado (filter).
Estos métodos son esenciales para un controlador en una aplicación web o API, ya que permiten realizar operaciones CRUD (crear, leer, actualizar y eliminar) en la base de datos de manera eficiente y segura.

Operadores de Filtrado en MongoDB

Utilizar operadores en consultas de bases de datos proporciona una flexibilidad significativa al desarrollar aplicaciones.

$eq (equal): Selecciona documentos donde el valor del campo especificado sea igual al valor especificado.

```
async getProductsWithPriceEqualTo(price) {
    try {
        const products = await Product.find({ price: { $eq: price } });
        return products;
    } catch (error) {
        throw new Error('Error al obtener los productos con precio igual a ' + price);
    }
}
```

$ne (not equal): Selecciona documentos donde el valor del campo especificado no sea igual al valor especificado.

```
async getProductsWithPriceNotEqualTo(price) {
    try {
        const products = await Product.find({ price: { $ne: price } });
        return products;
    } catch (error) {
        throw new Error('Error al obtener los productos con precio distinto de ' + price);
    }
}
```

$gt (greater than): Selecciona documentos donde el valor del campo especificado sea mayor que el valor especificado. (Como se mencionó anteriormente)

$lt (less than): Selecciona documentos donde el valor del campo especificado sea menor que el valor especificado.

```
async getProductsWithPriceLessThan(price) {
    try {
        const products = await Product.find({ price: { $lt: price } });
        return products;
    } catch (error) {
        throw new Error('Error al obtener los productos con precio menor que ' + price);
    }
}
```

Lista de operadores

Lista de operadores:

1. $eq (Equal): Coincide con valores que son iguales a un valor especificado.
   { age: { $eq: 25 } }

2. $ne (Not Equal): Coincide con todos los valores que no son iguales a un valor especificado.
   { age: { $ne: 25 } }

3. $gt (Greater Than): Coincide con valores que son mayores que un valor especificado.
   { age: { $gt: 25 } }

4. $gte (Greater Than or Equal): Coincide con valores que son mayores o iguales a un valor especificado.
   { age: { $gte: 25 } }

5. $lt (Less Than): Coincide con valores que son menores que un valor especificado.
   { age: { $lt: 25 } }

6. $lte (Less Than or Equal): Coincide con valores que son menores o iguales a un valor especificado.
   { age: { $lte: 25 } }

7. $in (In): Coincide con cualquier valor en una matriz especificada.
   { name: { $in: ['Alice', 'Bob'] } }

8. $nin (Not In): Coincide con ninguno de los valores en una matriz especificada.
   { name: { $nin: ['Alice', 'Bob'] } }

9. $and: Une cláusulas de consulta con un lógico AND.
   { $and: [{ age: { $gte: 25 } }, { age: { $lt: 30 } }] }

10. $or: Une cláusulas de consulta con un lógico OR.
    { $or: [{ age: { $lt: 25 } }, { age: { $gt: 30 } }] }

11. $not: Invierte el efecto de un operador de consulta.
    { age: { $not: { $gt: 25 } } }

12. $nor: Une cláusulas de consulta con un lógico NOR.
    { $nor: [{ age: { $lt: 25 } }, { age: { $gt: 30 } }] }

13. $exists: Coincide con documentos que tienen el campo especificado.
    { email: { $exists: true } }

14. $regex: Coincide con valores que coinciden con una expresión regular.
    { name: { $regex: /pattern/, $options: 'i' } }

15. $size: Coincide con cualquier array con el número de elementos especificado.
    { tags: { $size: 3 } }

16. $all: Coincide con arrays que contienen todos los elementos especificados.
    { tags: { $all: ['tech', 'code'] } }

17. $elemMatch: Coincide con documentos que contienen un elemento que coincide con todos los criterios de la consulta.
    { comments: { $elemMatch: { author: 'Alice', votes: { $gte: 5 } } } }

18. $type: Coincide con valores de un tipo BSON especificado.
    { age: { $type: 'number' } }

19. $text: Realiza una búsqueda de texto.
    { $text: { $search: "cadena de búsqueda" } }

20. $mod: Coincide con valores donde el residuo de la división del campo por un divisor especificado es igual a un valor especificado.

```
{ age: { $mod: [4, 0] } } // Coincide con edades que son múltiplos de 4
```

El uso de operadores en Mongoose permite realizar consultas avanzadas en MongoDB de manera eficiente y flexible. Estos operadores facilitan filtrar datos con precisión, combinar múltiples condiciones, realizar búsquedas de texto, y manejar arrays y tipos específicos. Con operadores como `$eq`, `$gt`, `$in`, `$regex`, y `$and`, entre otros, se pueden construir consultas complejas de manera eficiente, mejorando el rendimiento y la capacidad de obtener resultados relevantes y específicos según las necesidades de la aplicación. Esto facilita el manejo y análisis de grandes volúmenes de datos, optimizando las operaciones de lectura y escritura en la base de datos.

También podemos configurar los esquemas como las querys que hacemos a la DB para evitar sobrecargas a la hora de obtener algunos datos.

1.Índices:

- **Definición**: Los índices mejoran el rendimiento de las consultas al permitir búsquedas más rápidas. Puedes definir índices en tus esquemas de Mongoose.
- **Ejemplo**:

```
const userSchema = new mongoose.Schema({
       name: { type: String, index: true },
       age: Number,
       email: { type: String, unique: true }
     });
```

2.Paginación:

- **Uso de limit y skip**: Estas funciones ayudan a dividir los resultados en páginas para evitar la sobrecarga de datos en una sola consulta.
- **Ejemplo**:

```
const users = await User.find().skip(10).limit(10);
```

3.Validaciones:

- **Incorporación de reglas**: Mongoose permite agregar validaciones en los esquemas para asegurar la integridad de los datos.
- **Ejemplo**:

```
const userSchema = new mongoose.Schema({
    name: { type: String, required: true },
    age: { type: Number, min: 0 },
    email: { type: String, required: true, match: /.+@.+\..+/ }
});
```

4.Hooks (Middlewares):

- **Pre y post hooks**: Mongoose permite definir middleware para realizar acciones antes o después de ciertas operaciones (como guardar o eliminar).
- **Ejemplo**:

```
userSchema.pre('save', function(next) {
    this.updatedAt = Date.now();
    next();
  });
```

5.Esquemas y Subdocumentos:

- **Estructuración de datos**: Podés usar subdocumentos para organizar datos relacionados dentro de un solo documento.
- **Ejemplo**:

```
const commentSchema = new mongoose.Schema({
    content: String,
    author: String
});
const postSchema = new mongoose.Schema({
    title: String,
    content: String,
    comments: [commentSchema]
});
```

6.Agregación:

- **Pipelines de agregación**: Utiliza pipelines de agregación para realizar operaciones más complejas de análisis de datos.
- **Ejemplo**:

```
const result = await User.aggregate([
    { $match: { age: { $gte: 18 } } },
    { $group: { _id: '$age', total: { $sum: 1 } } }
]);
```

7. **Population**:
   - **Referencias entre documentos**: Utiliza el método populate para llenar automáticamente los campos con datos de documentos relacionados.
   - **Ejemplo**:

```
const authorSchema = new mongoose.Schema({
    name: String,
    age: Number,
    bio: String
});

const Author = mongoose.model('Author', authorSchema);

const bookSchema = new mongoose.Schema({
    title: String,
    author: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Author'
    }
});
const books = await Book.find().populate('author');
```

8. **Gestión de Errores**:
    - **Manejo de excepciones**: Implementa un manejo adecuado de errores en tus operaciones con la base de datos para mejorar la robustez de tu aplicación.
    - **Ejemplo**:

```
try {
    const user = await User.findById(id);
} catch (error) {
    console.error('Error fetching user:', error);
}
```

9. **Optimización de Consultas**:
    - **Proyección**: Selecciona solo los campos necesarios para mejorar el rendimiento de las consultas.
    - **Ejemplo**:

```
const users = await User.find({}, 'name email');
```

Recomendamos usar esta `const users = await User.find({}, ['name', 'email']);`

10. **Relaciones Unidireccionales**:
    - Una relación unidireccional ocurre cuando solo uno de los modelos tiene una referencia al otro. Por ejemplo, si cada libro tiene un autor, pero no necesitas acceder directamente desde el autor a todos sus libros, puedes establecer una relación unidireccional desde Book hacia Author.

    - **Ejemplo**:

```
// author model
import mongoose from "mongoose";

const authorSchema = new  mongoose.Schema({
  name: {
     type: String,
     required: true
  },
  // No necesitamos referenciar los libros aquí
});

const Author = mongoose.model('Author', authorSchema);
```

```
// book model
import mongoose from "mongoose";

const bookSchema = new mongoose.Schema({
    title: {
        type: String,
        required: true
    },
    pages: {
        type:
            Number,
        required: true
    },
    author: {
        type: Schema.Types.ObjectId,
        ref: 'Author'
    } // Referencia al modelo de Author
});

const Book = mongoose.model('Book', bookSchema);
```

11. **Relaciones Bidireccionales**:
    - Una relación bidireccional es útil cuando necesitas acceder fácilmente a la información relacionada desde ambos modelos. Por ejemplo, si además de saber el autor de un libro, necesitas listar todos los libros de un autor específico, podrías establecer una referencia en ambos modelos.

    - **Ejemplo**:

```
import mongoose from "mongoose";

const authorSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true
    },
    books: [
        {
            type: Schema.Types.ObjectId,
            ref: 'Book'
        }
    ] // Array de IDs de Book
});

const Author = mongoose.model('Author', authorSchema);

// Modelo de Book (sin cambios):
```

Crear un Autor y un Libro:

Ejemplo:

```
import express from 'express';
const app = express();
app.use(express.json());

app.post('/authors', async (req, res) => {
  const author = new Author({ name: req.body.name });
  await author.save();
  res.status(201).json(author);
});

app.post('/books', async (req, res) => {
  const book = new Book({
    title: req.body.title,
    pages: req.body.pages,
    author: req.body.authorId  // ID del autor
  });
  await book.save();

  // Opcional: Actualizar el autor si la relación es bidireccional
  await Author.findByIdAndUpdate(req.body.authorId, { $push: { books: book._id } });

  res.status(201).json(book);
});

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Seeding

¿Qué es el Seeding?

Inicialización de la base de datos:
- Proceso de cargar datos de ejemplo o de prueba en la base de datos.

Utilidad:
- Facilita el desarrollo y las pruebas proporcionando un conjunto de datos consistentes.
- Esencial para el despliegue inicial de la aplicación.

Beneficios:
- **Consistencia**: Asegura que todos los entornos de desarrollo y prueba tengan los mismos datos.
- **Automatización**: Configura y restablece el entorno de desarrollo o pruebas automáticamente.
- **Ahorro de tiempo**: Evita la necesidad de ingresar datos manualmente cada vez.

Ejemplo de Proceso de Seeding:

1.Conectar a la base de datos: Configurar la conexión a la base de datos.

2.Eliminar datos existentes: (Opcional) Borrar datos existentes para evitar duplicados.

3.Insertar datos iniciales: Cargar los datos necesarios.

4.Cerrar la conexión: Finalizar la conexión a la base de datos.

¿Qué es Faker?

Faker es una biblioteca muy útil en el desarrollo de software para generar datos ficticios de manera rápida y fácil. Se utiliza principalmente para crear datos de prueba durante el desarrollo, lo que permite simular bases de datos, llenar formularios con datos aleatorios, o incluso realizar pruebas de carga. Faker puede generar nombres, direcciones, números de teléfono, correos electrónicos, descripciones, y más, lo que es ideal cuando necesitás realizar pruebas sin datos reales.

Instalación de Faker:

```
npm install @faker-js/faker
```

Ejemplo

Vamos a crear un proyecto desde cero

Creamos una carpeta y el archivo package.json

- mkdir proyecto
- cd proyecto
- npm init -y

package.json:

- Agregamos la propiedad “type” con el valor “module
- En la propiedad scripts agregaremos 2 props “dev” y “seed”

```
"scripts": {
    "dev": "nodemon ./src/app.js",
    "seed": "node ./src/models/seed.js"
  },
```

Instalamos las dependencias:

- npm install express dotenv mongoose @faker-js/faker

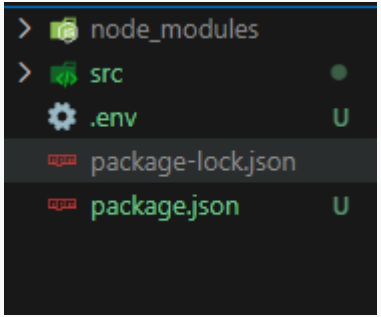
.env:

- Creamos las variables para la base de datos y el puerto en la carpeta raiz

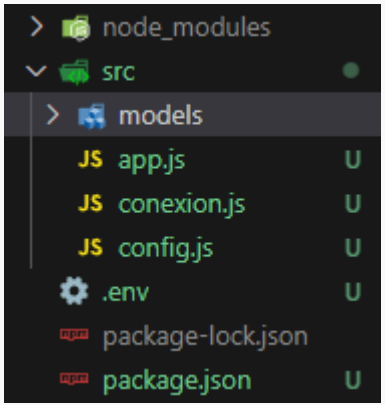
Archivos JavaScript:

- Creamos una carpeta src, dentro de esta app.js config.js conexion.js y una carpeta models

```
// config.js
import dotenv from 'dotenv'

dotenv.config()

export default {
    port: process.env.PORT || '3000',
    MONGO_DB_URI: process.env.MONGO_DB_URI
}
```




```
// conexion.js
import mongoose from "mongoose";
import config from "./config.js";

export default async function connectToDB() {
    try {
        await mongoose.connect(config.MONGO_DB_URI);
        console.log('Conectado a MongoDB Atlas');
    } catch (err) {
        console.error('Error al conectar a MongoDB Atlas:', err);
    }
}
```




```
// app.js
import express from 'express';
import config from './config.js';
import connectToDB from './conexion.js';

const app = express();
app.use(express.json());

connectToDB()

app.listen(config.port, () => {
    console.log(`Listening on ${config.port}`)
});
```

Dentro de la caperta models:

- Creamos 2 modelos (author, book) y el archivo para crear datos “seed.js”

```
import mongoose from "mongoose";

const authorSchema = new mongoose.Schema({
    name: {
      type: String,
      required: true
    },
    age: {
      type: Number,
      min: 0,
      required: true
    },
    bio: {
      type: String,
      required: true
    }
  });

  const Author = mongoose.model('Author', authorSchema);

  export default Author;
```

Tenemos un campo author que tiene una prop “ref” este tendrá una referencia al modelo author para luego poder usar la población al momento de mostrar los datos

```
import mongoose from 'mongoose';

const bookSchema = new mongoose.Schema({
  title: {
    type: String,
    required: true
  },
  pages: {
    type: Number,
    min: 1,
    required: true
  },
  author: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Author',
    required: true
  }
});

const Book = mongoose.model('Book', bookSchema);

export default Book;
```

El archivo seed solo se ejecuta cada vez que queremos resetear los datos, o bien, luego de crear los modelos para poder poblar de datos los esquemas

```
import mongoose from 'mongoose';
import { faker } from '@faker-js/faker';
import Author from './Author.js';
import Book from './Book.js';
import connectToDB from '../conexion.js';

connectToDB()
// Crear autores ficticios
const generateAuthors = async (numAuthors = 10) => {
    const authors = [];

    for (let i = 0; i < numAuthors; i++) {
      const author = await Author.create({
        name: faker.person.fullName(), // Usar faker.person para nombres
        age: faker.number.int({ min: 20, max: 80 }), // Usar faker.number.int en lugar de faker.datatype.number
        bio: faker.lorem.paragraph(),
      });

      authors.push(author);
      console.log(`Autor creado: ${author.name}`);
    }

    return authors;
  };
```

```
  // Crear libros ficticios
const generateBooks = async (authors, numBooks = 20) => {
  for (let i = 0; i < numBooks; i++) {
    const randomAuthor = authors[Math.floor(Math.random() * authors.length)];

    const book = await Book.create({
      title: faker.lorem.words(3),
      pages: faker.number.int({ min: 100, max: 500 }), // Usar faker.number.int para generar páginas
      author: randomAuthor._id,
    });

    console.log(`Libro creado: ${book.title} por ${randomAuthor.name}`);
  }
};

// Función principal para generar los datos
const seedDatabase = async () => {
  try {
    // Limpiar la base de datos
    await Author.deleteMany({});
    await Book.deleteMany({});

    // Generar autores y libros
    const authors = await generateAuthors();
    await generateBooks(authors);

    console.log('Seeding completado exitosamente');
  } catch (error) {
    console.error('Error generando datos:', error);
  } finally {
    mongoose.connection.close(); // Cerrar la conexión cuando termina
  }
};
seedDatabase();
```

A tener en cuenta que el código en la diapositivas se tuvo que cortar en 2 para que se pueda apreciar

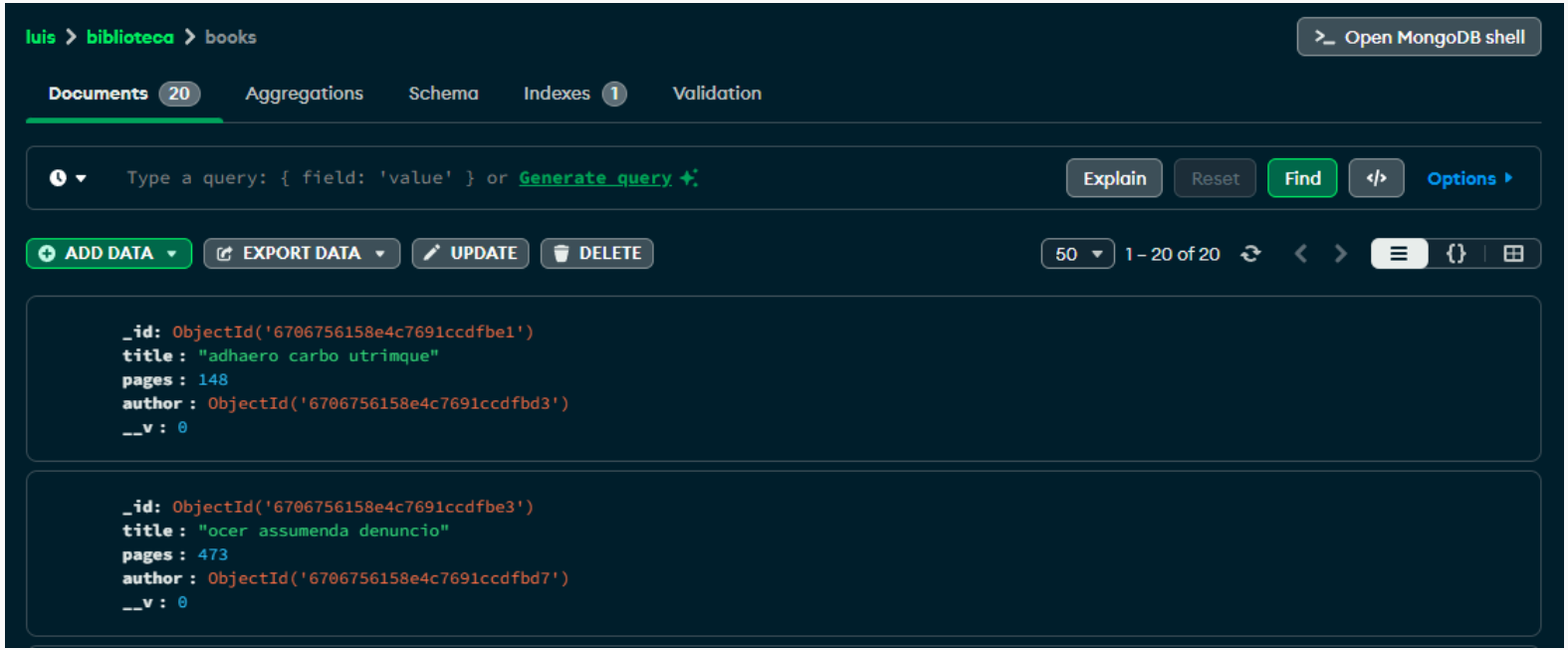
Carguemos los datos a la DB

```
npm run seed
```

Si todo salió OK veremos este mensaje

```
> code@1.0.0 seed
> node ./models/seed.js

Conectado a MongoDB
Seeding completado exitosamente
```

Para corroborar que se cargaron los datos basta con abrir MongoDB Compass y verificar



Hagamos un par de endpoints para traer los datos y verificar, para eso vamos a crear un carpeta para los controladores y otra para las rutas

controllers/ y routes/

```
import Author from '../models/Author.js';

// Obtener todos los autores
export const getAllAuthors = async (req, res) => {
  try {
    const authors = await Author.find(); // Obtiene todos los autores de la
base de datos
    res.status(200).json(authors);
  } catch (error) {
    res.status(500).json({ message: 'Error obteniendo autores', error });
  }
};

// Obtener un autor por su ID
export const getAuthorById = async (req, res) => {
  try {
    const author = await Author.findById(req.params.id); // Busca el autor por
ID

    if (!author) {
      return res.status(404).json({ message: 'Autor no encontrado' });
    }

    res.status(200).json(author);
  } catch (error) {
    res.status(500).json({ message: 'Error obteniendo el autor', error });
  }
};
```

```
import Book from '../models/Book.js';

// Obtener todos los libros
export const getAllBooks = async (req, res) => {
  try {
    const books = await Book.find().populate('author'); // Obtiene todos los
libros y los autores asociados
    res.status(200).json(books);
  } catch (error) {
    res.status(500).json({ message: 'Error obteniendo libros', error });
  }
};

// Obtener un libro por su ID
export const getBookById = async (req, res) => {
  try {
    const book = await Book.findById(req.params.id).populate('author'); //
Busca el libro por ID y trae el autor relacionado

    if (!book) {
      return res.status(404).json({ message: 'Libro no encontrado' });
    }

    res.status(200).json(book);
  } catch (error) {
    res.status(500).json({ message: 'Error obteniendo el libro', error });
  }
};
```

```js
import Router from 'express';
import { getAllAuthors, getAuthorById } from '../controllers/authorController.js';

const router = Router();

// Ruta para obtener todos los autores
router.get('/', getAllAuthors);

// Ruta para obtener un autor por su ID
router.get('/:id', getAuthorById);

export default router;
```

```js
import express from 'express';
import { getAllBooks, getBookById } from '../controllers/bookController.js';

const router = express.Router();

// Ruta para obtener todos los libros
router.get('/', getAllBooks);

// Ruta para obtener un libro por su ID
router.get('/:id', getBookById);

export default router;
```

Para poder manejar los campos a mostrar como segundo parámetro pasaremos un array con los campos que queremos traer solamente.

NOTA IMPORTANTE, a pesar de no querer traer el _id, si no tenemos un elemnto “-_id” en el array lo va a traer igual

```
    try {
        const books = await Book.find().populate('author', ['name', 'age', '-_id']);
        res.status(200).json(books);
    } catch (error) {
        res.status(500).json({ error: error });
    }
})
```

Si queremos filtrar los libros por algún parámetro, podemos usar los operadores ya vistos

Ejemplo: Vamos a traer a todos los libros que tengan más de 220 páginas

```
try {
    const books = await Book.find({pages: {$gt: 220}}).populate('author', ['name', 'age', '-_id']);
    res.status(200).json(books);
} catch (error) {
    res.status(500).json({ error: error });
}
```


```
[
  {
    "_id": "6660809a7fd5601302ba47e8",
    "title": "Book Two",
    "pages": 300,
    "author": {
      "name": "Jane Doe",
      "age": 45
    },
    "__v": 0
  },
  {
    "_id": "6660809a7fd5601302ba47e9",
    "title": "Book Three",
    "pages": 250,
    "author": {
      "name": "John Smith",
      "age": 38
    },
    "__v": 0
  },
```

De esta manera podemos crear métodos que con los parámetros recibidos desde querys tengamos la posibilidad de traer datos que se requieran, por ejemplo:

http://localhost:3000/books/?pag=200

```
try {
    const { pag = 0 } = req.query

    const books = await Book.find({ pages: { $gt: pag } }).populate('author', ['name', 'age', '-_id']);
    res.status(200).json(books);
} catch (error) {
    res.status(500).json({ error: error });
}
```


```
[
  {
    "_id": "6660809a7fd5601302ba47e8",
    "title": "Book Two",
    "pages": 300,
    "author": {
      "name": "Jane Doe",
      "age": 45
    },
    "__v": 0
  },
  {
    "_id": "6660809a7fd5601302ba47e9",
    "title": "Book Three",
    "pages": 250,
    "author": {
      "name": "John Smith",
      "age": 38
    },
```

Ejercicio

Crear 2 enpoints:

- En tu proyecto o en uno nuevo, deberás crear 2 enpoints que use parámetros de mongoose y algún get que tenga alguna de las funciones que nos ayudan a no sobrecargar las consultas a la DB