



Estructura de documentos

Desarrollo de Sistemas Web - Back End





Base de datos orientada a documentos

Las bases de datos orientadas a documentos son bases de datos que poseen una naturaleza flexible, semiestructurada y jerárquica de su contenido. Estas pertenecen al grupo NoSQL y guardan principalmente información basada en una estructura de documentos anidados. Los documentos que contienen estas bases de datos, pueden referirse a documentos de texto, XML o JSON.

Este tipo de base de datos destaca por su gran rendimiento en operaciones de lectura de la información con grandes volúmenes de datos.

MongoDB es una base de datos de documentos open-source que nos permite indexar registros de texto completo que se encuentra estructurado en un Documento. En MongoDB un documento es un registro compuesto por pares “campo : valor”. Estos documentos son muy parecidos a los objetos JSON.

```
{
  name : "Marcela Sena",
  age : 29,
  alias : "MarceStarlet",
  memberOf : ["GimnasioFit", "NodeDevs"],
  address : { street : "Av Corrientes", number : 234 },
  zipCodes : [
    { zipCode : 1234, zone: "Palermo" },
    { zipCode : 4321, zone: "Centro" }
  ]
}
```

Bases de datos

En una base de datos, la información puede organizarse en tablas o en documentos. Cuando organizamos información en un Excel, lo hacemos en formato tabla y, cuando los médicos hacen fichas a sus pacientes, están guardando la información en documentos. Lo habitual es que las bases de datos basadas en tablas sean bases de datos relacionales y las basadas en documentos sean no relacionales, pero esto no tiene que ser siempre así. En realidad, una tabla puede transformarse en documentos, cada uno formado por cada fila de la tabla. Solo es una cuestión de visualización.

student_id	age	score
1	12	77
2	12	68
3	11	75



```
[
  {
    "student_id":1,
    "age":12,
    "score":77
  },
  {
    "student_id":2,
    "age":12,
    "score":68
  },
  {
    "student_id":3,
    "age":11,
    "score":75
  }
]
```



Bases de datos relacionales

Como su nombre lo indica, utilizan el modelo relacional y siempre es mejor usarlas cuando los datos que vamos a utilizar son consistentes y ya tienen una estructura planificada.

- Las bases de datos relacionales funcionan bien con datos estructurados.
- Las organizaciones que tienen muchos datos no estructurados o semiestructurados no deberían considerar una base de datos relacional.



Ventajas

- Al ser una tecnología bastante madura, cuenta con una documentación muy extensa y una comunidad bastante activa. Cualquier duda puede ser resuelta con un poco de investigación.
- Los estándares SQL se encuentran bien definidos y son ampliamente aceptados.
- Una gran cantidad de desarrolladores cuentan con amplia experiencia en esta tecnología.
- Toda base de datos relacional debe cumplir con los principios ACID, por lo cual los datos son confiables.



Desventajas

Hasta hace un tiempo, las bases de datos relacionales eran la primera opción al momento de desarrollar casi cualquier aplicación, eran robustas, confiables y ampliamente conocidas por los desarrolladores, pero pronto surgió un problema, las bases de datos relacionales **tienen muy poca escalabilidad, esto quiere decir que si queremos agregar una nueva funcionalidad a nuestra aplicación probablemente será necesario un rediseño del modelo de nuestra base de datos**, lo cual requiere tiempo y recursos por parte del equipo de desarrollo.

Además de esto, no funcionan muy bien cuando no estructuramos correctamente los datos, y tratar de migrar de un sistema gestor a otro implica un proceso muy amplio de análisis para asegurarnos que el esquema establecido para el sistema receptor es idéntico al del origen.

Fundamentos



Las apps de Express pueden usar cualquier mecanismo de bases de datos soportadas por Node (Express en sí mismo no define ninguna conducta/requerimiento específico adicional para administración de bases de datos). Las opciones más populares son PostgreSQL, MongoDB, SQLite, MySQL, SQL Server, Redis.

En este curso implementaremos una solución NoSQL porque precisamente nos vamos a mover en un entorno en el que van a existir constantes operaciones de E/S con la base de datos.

Existe una base de datos documental de código abierto, cuya unión a node es muy fácil mediante driver: MongoDB. El paquete mongoose nos permite MongoDB en node.js mediante el mencionado driver. Además, MongoDB tiene un conjunto de gestores visuales que facilitan mucho el trabajo con el motor de la Base de Datos.



Bases de datos no relacionales (NoSQL)

A diferencia de las bases de datos relacionales, los datos de una base de datos NO-SQL (Not Only SQL) son más flexibles en cuanto a consistencia de datos y se han convertido en una opción que intenta solucionar algunas limitaciones que tiene el modelo relacional.

La información se organiza normalmente mediante documentos y es muy útil cuando no tenemos un esquema exacto de lo que se va a almacenar.

Las bases de datos no relacionales son buenas para guardar modelos con una alta transaccionalidad, pues su tiempo de respuesta es más bajo, comparadas con las relacionales.

Este tipo de bases de datos no es un reemplazo para las bases de datos relacionales, más bien es una alternativa que surgió para satisfacer las necesidades de aplicaciones cada vez más complejas y que se encuentran en constante actualización.



Ventajas de NoSQL

Si se comparan con las bases de datos relacionales, las bases de datos NoSQL son más escalables y ofrecen un mayor rendimiento; además, su modelo de datos aborda varias cuestiones que el modelo relacional pasa por alto:

- Sumamente escalables, se pueden editar al mismo tiempo que nuestro código
- Grandes volúmenes de datos estructurados, semiestructurados y no estructurados en constante cambio
- Los datos pueden adaptarse a otro tipo de sistema de manera sumamente sencilla ya que no presentan un esquema definido.
- Fáciles de respaldar, basta con hacer una copia del archivo.



Desventajas

- Son una tecnología relativamente reciente, como consecuencia, el conocimiento y la información son más limitados.
- Cada base de datos tiene su propia manera de formatear sus datos, a diferencia de un lenguaje de consulta estructurado como SQL.



¿Cómo saber qué base de datos necesitamos?

Es muy común entre desarrolladores de aplicaciones encontrarse en una situación de tener que elegir si se va a usar una base de datos relacional o no relacional. Tener un buen diseño de base de datos desde el principio nos puede ayudar a ahorrar tiempo a la hora de programar. Las bases de datos relacionales (con el estándar SQL para actualizar/recuperar datos) y no relacionales se organizan de forma diferente, también admiten tipos de datos específicos, por eso debes entender cómo se diseña cada una.

Para decidir entre una base de datos SQL o NoSQL, debemos reflexionar en la relación entre los diferentes tipos de datos que vas a almacenar: si pueden vivir separados y no tienen relaciones, podemos utilizar una base de datos NoSQL. Pero si necesitamos relacionar esos datos entre sí, es mejor una base de datos SQL.

Decantarse por el uso de NoSQL frente a SQL es una cuestión de análisis del problema: en general, se recomienda usar una base de datos no relacional cuando se tienen grandes cantidades de datos y/o se encuentran dificultades para modelarlos según un esquema. De otra manera, SQL es una tecnología muy madura con muchísimos años de investigación y desarrollo sobre ella



Integridad de datos

La integridad de datos es la garantía de que los datos almacenados mantendrán su exactitud y consistencia a través del tiempo.

SQL: Las tablas tienen estructuras rígidas, donde cada dato tiene un tipo definido, no podemos almacenar datos de otro tipo diferente, y no se vale más de un dato en un mismo campo. Puesto que todos los registros cumplen las mismas reglas, si nuestro código funciona con un solo registro, servirá con todos los demás.

NoSQL: Hay varios tipos de base de datos NoSQL, pero en general, ninguna te exige que definas el tipo de datos que vas a almacenar. Un día un campo puede ser un número y al otro día un String o Array o hasta un JSON. Más que saber qué es la data, NoSQL pone mayor prioridad en cómo acceder dicha data.

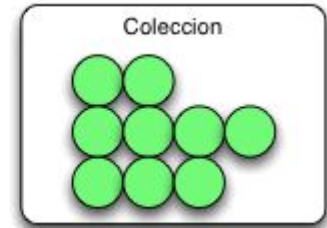
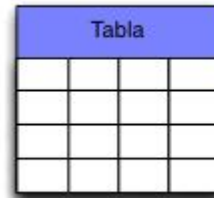
Ventajas



- **flexibilidad**, por usar como modelo de datos documentos en notación JavaScript (JSON). Estos documentos no tienen un esquema fijo, como sucede con los documentos XML, más rígidos en el sentido que los elementos deben ajustarse al esquema establecido para ellos, y como sucede en las bases de datos relacionales, donde un elemento a insertar en una tabla debe cumplir las condiciones fijadas para las columnas de dicha tabla..
- **potencia**, por incorporar mecanismos propios de bases de datos relacionales que se suman al resto de ventajas. Entre ellos, los autores destacan los índices secundarios, consultas dinámicas, ordenación o operaciones “upsert” (update si el documento existe, insert si no)
- **velocidad**, porque agrupa toda la información relacionada en documentos en lugar de en tablas como ocurre en las bases de datos relacionales.
- **escalabilidad horizontal**, es decir, posee mecanismos para que sea muy fácil y seguro, sin interrupciones de servicio, añadir más máquinas corriendo MongoDB al cluster.
- **facilidad de uso**, por la sencillez con la que se instala y configura MongoDB, teniendo una instancia corriendo en muy pocos pasos.

Estructura de gestión de la información

- **Colecciones:** Las colecciones son una homologación directa a las tablas en una base de datos relacional. Las tablas almacenan registros (filas), mientras que las colecciones almacenan documentos.
- Son grupos de documentos que comparten un propósito o tipo de datos común. Se asemejan a las tablas en bases de datos relacionales, pero no requieren un esquema rígido.
- **Documentos:** Los documentos son almacenados en formato BSON que es una representación binaria de mapas basados en JSON. Estos documentos están compuestos por pares de clave valor y cada documento puede tener variaciones importantes con el documento anterior, puede tener más de una clave valor o nombres distintos.
- Son objetos individuales que contienen datos. Tienen una estructura flexible y pueden variar de un documento a otro dentro de la misma colección.





Comparación con Bases de Datos Relacionales

En bases de datos relacionales:

Tablas (equivalente a colecciones) tienen un esquema fijo (cada fila debe tener los mismos campos).

Filas (equivalente a documentos) son registros que cumplen con ese esquema.

En MongoDB / JSON:

Las colecciones pueden contener documentos con diferentes estructuras y campos.



Documentos

Definición: Un documento es la unidad básica de almacenamiento en MongoDB. Se puede considerar como un objeto que contiene datos en un formato similar a JSON (JavaScript Object Notation), conocido como BSON (Binary JSON) en MongoDB.

Estructura: Un documento tiene una estructura de pares clave-valor. Por ejemplo:

```
{
  "_id": "60d5f484f49f1e001f1d3e42",
  "name": "Juan",
  "number": "123456789",
  "email": "juan@example.com"
}
```

En este ejemplo:

- `"_id"`: Es un identificador único generado automáticamente por MongoDB.
- `"name"`, `"number"`, `"email"`: Son campos que contienen datos específicos.

Flexibilidad: Los documentos en MongoDB pueden tener diferentes esquemas. Esto significa que no todos los documentos dentro de una colección necesitan tener los mismos campos. Por ejemplo, podrías tener documentos que representen usuarios, algunos con el campo `email` y otros sin él.



Colecciones

Definición: Una colección es un conjunto de documentos. En términos de bases de datos relacionales, se asemeja a una tabla. Las colecciones agrupan documentos que comparten un propósito o tipo de datos común.

Ejemplo de Colección: Si tienes una colección llamada `users`, podría contener múltiples documentos que representan diferentes usuarios:

```
[
  {
    "_id": "60d5f484f49f1e001f1d3e42",
    "name": "Juan",
    "number": "123456789",
    "email": "juan@example.com"
  },
  {
    "_id": "60d5f484f49f1e001f1d3e43",
    "name": "Maria",
    "number": "987654321"
  }
]
```



Tipos de datos BSON

Según lo comentado anteriormente, una [estructura BSON](#) está conformada por clave valor, en donde la clave debe ser del tipo String, mientras que el valor puede ser uno de los siguientes tipos:

- Strings.
- Enteros de 32 o 64 bits.
- Tipo de dato real de 64 bits IEEE 754.
- Fecha (número entero de milisegundos en Tiempo Unix).
- Array de bytes (datos binarios).
- booleanos.
- Nulo (null).
- Objetos anidados BSON.
- Array BSON.
- Expresiones Regulares

```
{
  _id: ObjectId("5f339953491024badf1138ec"),
  title: "MongoDB Tutorial",
  isbn: "978-4-7766-7944-8",
  published_date: new Date('June 01, 2020'),
  author: {
    first_name: "John",
    last_name: "Doe"
  }
}
```



Relaciones en MongoDB

Uno de los criterios que se tiende a confundir es el término de bases de datos no relacionales o NoSQL con la ausencia de relaciones, pero la verdad es que MongoDB desde hace un par de años permite implementar relaciones a través de un concepto llamado “referencias”, incluso en la documentación oficial se habla de cardinalidades y otras características que se conocen en una base de datos relacional, con la distinción de que MongoDB no se apega fielmente al modelo ER (Entidad Relación) y otras cualidades de las bases de datos relacionales.

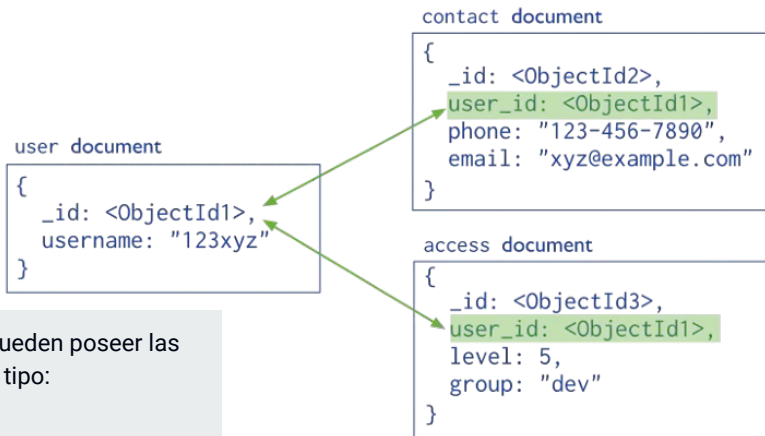
Las relaciones en MongoDB se pueden modelar en 2 enfoques diferentes: Relación incrustada o relación referenciada, la elección de estos enfoques dependerá del tipo de casuística a abordar y decisiones de modelamiento de datos. A continuación una breve descripción de cada enfoque:

Enfoque de relaciones

Relación incrustada: Relación que implica almacenar documentos secundarios incrustados dentro de un documento principal.



Relación referenciada: Relación que implica conectar colecciones diferentes por medio de referencias. La referencia se realiza a través de una propiedad o clave en común. Estas referencias pueden ser de dos tipos; referencias manuales o por DBRefs



Estas relaciones pueden poseer las cardinalidades del tipo:

- 1:1 — Uno a uno.
- 1:N — Uno a muchos.
- N:M — Muchos a muchos.



MongoDB Query Language (MQL)

MQL es el lenguaje de consulta y manipulación de información que mongoDB nos provee por defecto. Destacando que las consultas de mongoDB se basan en el lenguaje de programación JavaScript con algunas leves diferencias.

MQL tiene una extraordinaria variedad de opciones, operadores, expresiones y filtros ya disponibles en su core.

La documentación oficial de MongoDB compara MQL con SQL para operaciones comunes de bases de datos. Aquí un breve ejemplo de una operación INSERT en MongoDB:

```
db.users.insertOne(  ← collection
{
  name: "sue",        ← field: value
  age: 26,             ← field: value
  status: "pending"   ← field: value
}                    } document
)
```



Relación 1:1

Las relaciones uno a uno se crean al vincular dos documentos de diferentes colecciones mediante una propiedad clave compartida del mismo valor. A menudo, la mejor forma de crear esta relación es hacer que la colección secundaria consulte el valor clave de la primera colección. Comúnmente se utilizan los `ObjectId` para realizar la conexión, aunque puede ser cualquier otro tipo.

Por ejemplo Supongamos que tienes dos colecciones: Persona y Pasaporte. Cada persona solo tiene un pasaporte. Cada pasaporte solo pertenece a una persona.

Relación 1:N

Las relaciones uno a muchos se crean al vincular un documento de una colección con múltiples documentos de otra colección. Mencionando que esta relación se puede hacer de múltiples maneras, pero una opción muy utilizada es usar matrices crecientes y mutables, la cual se mostrará a continuación.

Por ejemplo Supongamos que una editorial puede revisar muchos libros

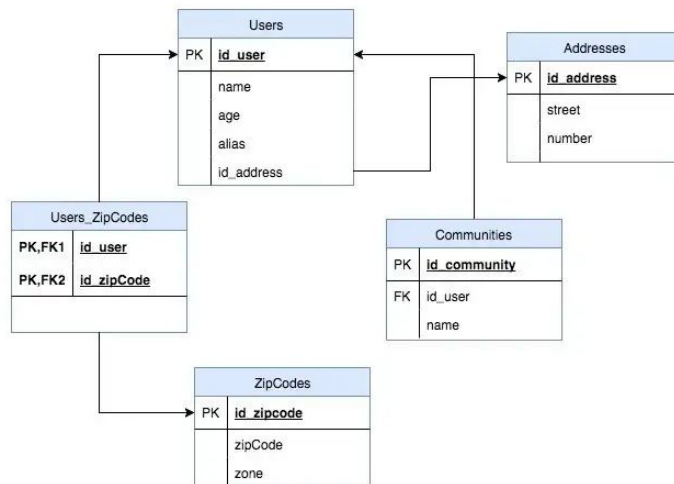
Relación N:M

Las relaciones de muchos a muchos se crean al vincular múltiples documentos en una colección asociados con múltiples documentos de otra colección. Nuevamente, para realizar esta relación se utilizan varias estrategias, entre las cuales para el ejemplo seguiremos utilizando matrices crecientes y mutables.

Un ejemplo de una relación **N** (muchos a muchos) sería entre **estudiantes** y **cursos**. Un estudiante puede inscribirse en muchos cursos, y un curso puede tener muchos estudiantes.

Migración de SQL a NoSql

Para ver cómo migrar un modelo relacional a un modelo de documentos, necesitamos un ejemplo. Imaginemos que tenemos el siguiente modelo relacional:



Representamos usuarios definiendo su nombre, edad, alias, las comunidades de las cuales es miembro, domicilio y las zonas donde se ubica.

Ahora veamos la representación en el modelo de documentos:

```
{
  "name" : "Marcela Sena",
  "age" : 29,
  "alias" : "MarceStarlet",
  "memberOf" : ["GimnasioFit", "NodeDevs"],
  "address" : { "street" : "Av Corrientes", "number" : 234 },
  "zipCodes" : [
    { "zipCode" : 1234, "zone": "Palermo" },
    { "zipCode" : 4321, "zone": "Centro" }
  ]
}
```



Manejo de relaciones

Teniendo en cuenta que esta no es una base de datos relacional, podemos manejar todas las sub-entidades dentro de un mismo documento, evitando usar joins para conocer la información que está estrechamente integrada.

Una buena práctica antes de construir nuestras colecciones y documentos es pensar en cuáles serán las búsquedas más comunes y así tener esas relaciones dentro de nuestro documento principal.

Otra opción si necesitamos (en el caso de arriba) saber qué usuario creó el comentario y algunos datos extras, podemos crear un Array de objetos y así tener más detalles de los datos relacionados:



Concepto de tuberías y transformaciones

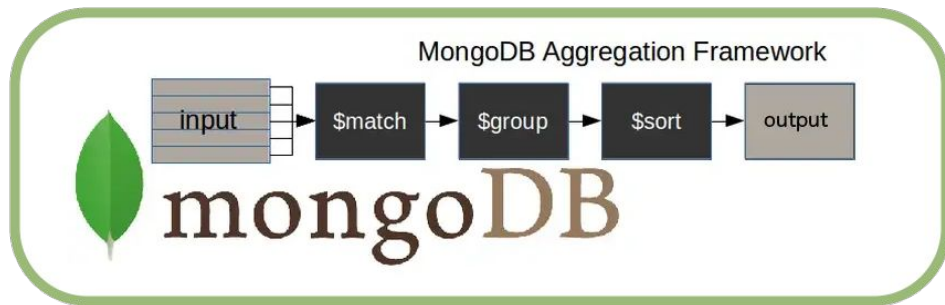
El concepto de **tuberías y transformaciones** en MongoDB puede ser un poco abstracto al principio. Para realizar el procesamiento de documentos, MongoDB se basa en el “Patrón de filtro de tubería”, utilizado comúnmente en arquitecturas de software. Este patrón consta de una o más etapas, en donde cada etapa realiza una operación con los datos de entrada y la salida o resultado se la entrega a la siguiente etapa para su procesamiento. **Ejemplo:**

Supongamos que tienes una colección de usuarios y quieres obtener el número de usuarios mayores de 18 años agrupados por país.

- La **primera etapa** filtra a los usuarios mayores de 18 años.
- La **segunda etapa** agrupa los usuarios por país y cuenta cuántos hay en cada grupo.
- La **tercera etapa** ordena los resultados por el número de usuarios en cada país.

MongoDB utiliza este concepto de **tuberías y transformaciones** para procesar datos en etapas. Cada etapa realiza una operación sobre los datos y pasa el resultado a la siguiente etapa para seguir transformándolos hasta llegar al resultado final

Tubería (Pipeline):



Una tubería en MongoDB es similar a una línea de ensamblaje. Imagina que tienes una serie de **etapas** (fases) y cada una de estas etapas realiza una **transformación** o un **filtro** sobre los datos que recibe.

- Los datos pasan por la **primera etapa** de la tubería, donde se aplica una transformación (por ejemplo, filtrar por una condición específica).
- Luego, los datos filtrados pasan a la **segunda etapa**, donde se realiza otro proceso (por ejemplo, agrupar resultados).
- Esto se repite hasta que los datos han pasado por todas las etapas de la tubería.

Etapas de una Tubería:

Cada etapa es como un paso en el procesamiento de los datos. MongoDB ofrece varias operaciones que pueden usarse en las etapas de una tubería, como:

- **\$match**: Filtrar los documentos según una condición.
- **\$group**: Agrupar documentos por un campo y aplicar funciones agregadas (como contar, sumar, etc.).
- **\$project**: Seleccionar qué campos incluir o excluir en el resultado.
- **\$sort**: Ordenar los documentos según un campo.
- **\$limit**: Limitar el número de documentos devueltos.



Transformaciones

Las transformaciones son las operaciones que aplicas sobre los datos en cada etapa de la tubería. Estas operaciones modifican los datos de entrada y generan una salida que pasa a la siguiente etapa.

Por ejemplo:

- En la primera etapa puedes filtrar todos los documentos que tienen una edad mayor a 18 años.
- En la segunda etapa puedes agrupar esos documentos por país de residencia.
- En la tercera etapa puedes ordenar los resultados por el número de personas en cada grupo.

Conexión con Mongoose



1. Instalación y Configuración de Mongoose

Mongoose es una biblioteca de modelado de objetos para MongoDB y Node.js, que proporciona una forma sencilla de interactuar con la base de datos.

Ejecuta el siguiente comando en la terminal: `npm install mongoose`

Para la conexión debemos tener un módulo de conexión (db.js, por ejemplo)

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

const conectarDB = async () => {
  try {
    await mongoose.connect(process.env.MONGODB_URI);
    console.log('Conectado a la base de datos MongoDB');
  } catch (error) {
    console.error('Error de conexión a MongoDB:', error.message);
    process.exit(1); // Termina el proceso si hay un error
  }
};

module.exports = conectarDB;
```



Operaciones básicas de manipulación

MongoDB Query Language (MQL)

Como se mencionó, MQL en su core ya contiene múltiples métodos definidos que nos ayudarán a manipular nuestra base de datos. Aquí los métodos más utilizados para el manejo de colecciones y documentos:

- Creación de colecciones: `createCollection`.
- Inserción de documentos: `insertOne` y `insertMany`.
- Búsqueda de documentos: `find`.
- Actualización de documentos: `updateOne` y `updateMany`.
- Borrado de documentos: `deleteOne` y `deleteMany`.

A continuación veremos algunos ejemplos básicos con estas funcionalidades.



Creación de colecciones

Para crear una nueva colección en nuestra base de datos, se usa el [método createCollection](#) y como parámetro se le indica el nombre de la colección a crear.

```
db.createCollection("usuarios");
```



Agregar ciertas reglas de negocio

Opcionalmente, el método `createCollection` recibe un objeto de configuración en donde se pueden agregar ciertas reglas de negocio y validaciones a nuestra colección, como por ejemplo validaciones y restricciones de nuestro modelo.

Inserción de documentos



Para insertar un documento a nuestra colección, se usa el [método insertOne](#) y como parámetro se le indica la estructura a guardar.

```
db.usuarios.insertOne({  
  
    nombre: "Diego",  
    email: "dcortes@example.com",  
    edad: 25  
  
});
```

Para insertar múltiples documentos a nuestra colección, se usa el [método insertMany](#) y como parámetro se le indica un array de estructuras a guardar.

```
db.usuarios.insertMany([  
    {  
        nombre: "Franco",  
        email: "franco@example.com",  
        edad: 20  
    },  
    {  
        nombre: "Juan",  
        email: "juan@example.com",  
        edad: 40  
    }  
]);
```


Búsqueda de documentos



Para la búsqueda de documentos se usa el método `find`. Para listar todos los documentos dentro de nuestra colección utilizamos `find` en conjunto de un objeto vacío.

```
db.usuarios.find({})
```

Para filtrar documentos por uno o más campos en específico, debemos indicar en un objeto los criterios de campos de búsqueda. En este caso filtraremos todos los usuarios que tienen el nombre de Diego con edad igual a 25.

```
db.usuarios.find({nombre: "Diego", edad: 25});
```

Actualización de documentos



Para actualizar un documento de nuestra colección, se usa el [método updateOne](#) y como parámetro se le indica el filtro de búsqueda más los elementos a actualizar con el operador \$set para su reemplazo.

En conjunto del método updateOne es usual utilizar el _id como criterio de búsqueda, ya que este filtro solo encuentra la primera coincidencia.

```
db.usuarios.updateOne(  
  { _id: ObjectId("62c0946602d55365f6a8b737") },  
  { $set: { email: "dcortes@update.com" } }  
  
);
```

[método updateMany](#) A diferencia de updateOne, este método actualiza todas las coincidencias.

```
db.usuarios.updateMany(  
  {},  
  { $set: { email: "all@update.com" } }  
);
```



Borrado de documentos

método `deleteOne` Este método solo eliminará la primera coincidencia, por lo que es común utilizar el campo `_id`.

```
db.usuarios.deleteOne({_id: ObjectId("62c0957402d55365f6a8b739")})
```

método `deleteMany` Para eliminar múltiples documentos de nuestra colección,

```
db.usuarios.deleteMany({email: "all@update.com"});
```

Algunos de los operadores más comunes



1. Operadores de Comparación

- **\$eq**: Igual a.
- **\$ne**: No igual a.
- **\$gt**: Mayor que.
- **\$gte**: Mayor o igual que.
- **\$lt**: Menor que.
- **\$lte**: Menor o igual que.

2. Operadores Lógicos

- **\$and**: Realiza una operación lógica "y" entre las condiciones.
- **\$or**: Realiza una operación lógica "o" entre las condiciones.
- **\$not**: Inversa de la expresión.
- **\$nor**: Realiza una operación lógica "ni" (no en ninguno de los criterios).

3. Operadores de Elementos

- **\$exists**: Verifica si un campo existe.
- **\$type**: Verifica el tipo de un campo.

4. Operadores de Matriz

- **\$all**: Coincide con todos los elementos especificados en un arreglo.
- **\$elemMatch**: Coincide con un documento en un arreglo que cumple con todos los criterios especificados.
- **\$size**: Coincide con el número de elementos en un arreglo.

Operadores de Agregación



Las operaciones de agregación son herramientas de MQL que nos ayudan a procesar documentos y retornar resultados calculados. Las operaciones de agregación se utilizan mayoritariamente para:

Agrupar valores de varios documentos.

Procesamiento y operaciones para el retorno de resultados.

Analizar cambios de datos a lo largo del tiempo.

Las operaciones de agregaciones se pueden realizar de dos formas:

El Tuberias y transformaciones.

El Métodos de agregación de propósito único.

- **\$group**: Agrupa documentos por un campo específico y permite realizar operaciones de agregación.
- **\$match**: Filtra documentos que coinciden con un criterio.
- **\$project**: Modifica la forma en que se devuelve el documento, permitiendo incluir o excluir campos.
- **\$sort**: Ordena los documentos en la salida.
- **\$limit**: Limita el número de documentos devueltos.
- **\$skip**: Omite un número especificado de documentos en la salida.




8. Operadores de Texto

- **\$text**: Permite realizar búsquedas de texto en campos indexados para texto completo.
- **\$regex**: Permite realizar búsquedas con expresiones regulares.

7. Operadores de Expresión

- **\$expr**: Permite usar expresiones en la consulta.
- **\$where**: Permite especificar una expresión JavaScript para filtrar documentos (menos eficiente, se recomienda evitar).

Resumen de los métodos para compararlos



```
// 5.1. Creación de colecciones
// Crea una nueva colección llamada
"usuarios"
db.createCollection("usuarios");
```

```
// 5.2. Inserción de documentos
// Inserta un solo documento en la
colección "usuarios"
db.usuarios.insertOne({ nombre:
"Juan", edad: 30 });
```

```
// Inserta múltiples documentos en la
colección "usuarios"
db.usuarios.insertMany([
  { nombre: "Maria", edad: 25 },
  { nombre: "Carlos", edad: 28 }
]);
```

```
// 5.3. Búsqueda de documentos
// Encuentra todos los usuarios con edad mayor o
igual a 25
db.usuarios.find({
  $and: [
    { edad: { $gte: 25 } }, // Mayor o igual que 25
    { nombre: { $regex: /^J/ } } // Empieza con 'J'
  ]
});
```

```
// Encuentra un solo usuario llamado "Juan"
db.usuarios.findOne({ nombre: "Juan" });
```

```
// 5.4. Actualización de documentos
// Actualiza la edad de "Juan" a 31
db.usuarios.updateOne(
  { nombre: "Juan" },
  { $set: { edad: 31 } }
);
```

```
// Marca a los usuarios menores de 30 como
"joven"
db.usuarios.updateMany(
  { edad: { $lt: 30 } },
  { $set: { estado: "joven" } }
);
```

```
// 5.5. Borrado de documentos
// Elimina un usuario llamado "Juan"
db.usuarios.deleteOne({ nombre: "Juan" });
```

```
// Elimina a todos los usuarios menores de 25
años
db.usuarios.deleteMany({ edad: { $lt: 25 } });
```



Atención con los módulos

En el aula virtual, en varios lugares la versión de Mongoose que estás usando no acepta callbacks en el método `connect()`. A partir de Mongoose 6, el método `connect()` se basa completamente en promesas.

Cómo Solucionar el Error

Para solucionar esto, debes asegurarte de que estás utilizando `async/await` o `then/catch` para manejar la conexión.