Opciones avanzadas de Mongoose

En este documento vamos a analizar algunas operaciones algo más complejas que podemos hacer con bases de datos NoSQL, como la posibilidad de establecer conexiones entre diferentes colecciones, o definir subdocumentos dentro de un documento.

1. Relaciones entre colecciones

Vamos a volver a nuestra base de datos de contactos que venimos utilizando en estas sesiones. Es una base de datos muy simple, con una única colección llamada "contactos" cuyos documentos tienen tres campos: nombre, teléfono y edad. Vamos a añadirle más información, y para ello seguiremos trabajando sobre el proyecto "*ContactosMongo*" de nuestra carpeta "*ProyectosNode/Pruebas*". Sin embargo, para no mezclar los contenidos básicos que hemos estado viendo con otros más avanzados que trataremos a continuación, crea una copia llamada *ContactosMongo_v2* para lo que haremos a continuación.

1.1. Definir una relación simple

Supongamos que queremos añadir, para cada contacto, cuál es su restaurante favorito, de forma que varios contactos puedan tener el mismo. Del restaurante en cuestión nos interesa saber su nombre, dirección y teléfono. Para ello, podemos definir este esquema y modelo (en un fichero llamado `models/restaurante.js`):

```
let restauranteSchema = new mongoose.Schema({
    nombre: {
        type: String,
        required: true,
        minlength: 1,
        trim: true
    },
    direccion: {
        type: String,
        required: true,
        minlength: 1,
        trim: true
    },
    telefono: {
        type: String,
        required: true,
        unique: true,
        trim: true,
        match: /^\d{9}$/
    }
});
let Restaurante = mongoose.model('restaurantes', restauranteSchema);
module.export = Restaurante;
```

Y lo asociamos al esquema de contactos con un nuevo campo (omitimos con puntos suspensivos datos ya existentes de ejemplos previos):

```
let contactoSchema = new mongoose.Schema({
    nombre: {
        ...
    },
    telefono: {
        ...
    },
    edad: {
        ...
    },
    restauranteFavorito: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'restaurantes'
    }
});
let Contacto = mongoose.model('contactos', contactoSchema);
module.exports = Contacto;
```

Observemos que el tipo de dato de este nuevo campo es `ObjectId`, lo que indica que hace referencia a un *id* de un documento de ésta u otra colección. En concreto, a través de la propiedad `ref` indicamos a qué modelo o colección hace referencia dicho *id* (al modelo *restaurantes*, que se traducirá a la colección *restaurantes* en MongoDB).

1.2. Definir una relación múltiple

Vamos a dar un paso más, y a definir una relación que permita asociar a un elemento de una colección múltiples elementos de otra (o de esa misma colección). Por ejemplo, vamos a permitir que cada contacto tenga un conjunto de mascotas. Definimos un nuevo esquema para las mascotas, que almacene su nombre y tipo (perro, gato, etc.), en el archivo `models/mascota.js`.

```
let mascotaSchema = new mongoose.Schema({
    nombre: {
        type: String,
        required: true,
        minlength: 1,
        trim: true
    },
    tipo: {
        type: String,
        required: true,
        enum: ['perro', 'gato', 'otros']
    }
});
let Mascota = mongoose.model('mascotas', mascotaSchema);
module.exports = Mascota;
```

> **NOTA**: como nota al margen, observad cómo se puede utilizar el validador `enum` en un esquema para forzar a que un determinado campo sólo admita ciertos valores.

Para permitir que un contacto pueda tener múltiples mascotas, añadimos un nuevo campo en el esquema de contactos que será un array de *ids*, asociados al modelo de mascotas definido previamente:

```
let contactoSchema = new mongoose.Schema({
    nombre: {
        ...
    },
    telefono: {
        ...
    },
    edad: {
        ...
    },
    restauranteFavorito: {
        ...
    },
    mascotas: [{
        type: mongoose.Schema.Types.ObjectId,
        ref: 'mascotas'
    }]
});
let Contacto = mongoose.model('contactos', contactoSchema);
module.exports = Contacto;
```

En este caso, observad cómo la forma de definir la referencia a la colección de mascotas es la misma (se establece como tipo de dato un `ObjectId`, con referencia al modelo de *mascotas*), pero, además, el tipo de dato de este campo mascotas es un array (especificado por los corchetes al definirlo).

1.3. Inserciones de elementos relacionados

En MongoDB, cuando queremos insertar un nuevo contacto y especificar su restaurante favorito y/o sus mascotas, debemos hacerlo en varias etapas, similar a lo que ocurriría en un sistema relacional. Este proceso se realiza en varios pasos, y es importante controlar correctamente el flujo de las inserciones asíncronas para garantizar que todas las referencias sean válidas.

- Primero añadiríamos el restaurante favorito a la colección de restaurantes, y/o las mascotas a la colección de mascotas (salvo que exista previamente, en cuyo caso obtendríamos su *id*):

```
let restaurante1 = new Restaurante({
    nombre: "La Tagliatella",
    direccion: "C.C. San Vicente s/n",
    telefono: "965678912"
});
restaurante1.save().then(...

let mascota1 = new Mascota({
    nombre: "Otto",
    tipo: "perro"
});
mascota1.save().then(...

let mascota2 = new Mascota({
    nombre: "Luna",
    tipo: "gato"
});
mascota2.save().then(...
```

- Después, añadiríamos el nuevo contacto especificando el *id* de su restaurante favorito, añadido previamente, y/o los *ids* de sus mascotas en un array:

```
let contacto1 = new Contacto({
    nombre: "Nacho",
    telefono: 677889900,
    edad: 40,
    restauranteFavorito: '5acd3c051d694d04fa26dd8b',
    mascotas: ['5acd3c051d694d04fa26dd90',
               '5acd3c051d694d04fa26dd91']
});
contacto1.save().then(...
```

Evidentemente, en una operación "real" no tendremos que añadir a mano los *ids* de los documentos relacionados. Bastaría con elegirlos de algún tipo de desplegable para quedarnos con su *id*. Otra opción es utilizar la propiedad `_id` generada automáticamente por MongoDB. Esto evita el paso manual de ir a la base de datos a copiar y pegar los id.

```
let contacto1 = new Contacto({
    nombre: "Nacho",
    telefono: 677889900,
    edad: 40,
    restauranteFavorito: restaurante1._id,
    mascotas: [mascota1._id, mascota2._id]
});
contacto1.save().then(...
```

Problema: inserción de referencias sin esperar las promesas

En la solución anterior no estamos esperando a que las promesas se resuelvan. El contacto se está creando sin asegurarnos de que el restaurante y las mascotas han sido guardados correctamente. Esto puede resultar en un contacto con referencias a `_id` que no existen en las colecciones relacionadas.

Solución 1: anidar las promesas correctamente

Para asegurarnos de que las inserciones ocurren en el orden correcto, debemos anidar las promesas utilizando `then`. Esto garantiza que el contacto no se creará hasta que el restaurante y las mascotas se hayan guardado correctamente.

```
restaurante1.save().then((restauranteGuardado) => {
    mascota1.save().then((mascotaGuardada1) => {
        mascota2.save().then((mascotaGuardada2) => {
            let contacto1 = new Contacto({
                nombre: "Nacho",
                telefono: 677889900,
                edad: 40,
                restauranteFavorito: restauranteGuardado._id,
                mascotas: [mascotaGuardada1._id, mascotaGuardada2._id]
            });
            contacto1.save().then(...);
        });
    });
});
```

Solución 2: uso de async/await para un código más legible

Una alternativa más clara y manejable es utilizar `async/await`, lo que permite controlar el flujo asíncrono de manera secuencial y evitar la anidación excesiva de promesas.

```
async function guardarContactoConRelaciones() {
    try {
        // Guardar el restaurante
        let restauranteGuardado = await restaurante1.save();

        // Guardar las mascotas
        let mascotaGuardada1 = await mascota1.save();
        let mascotaGuardada2 = await mascota2.save();

        // Crear el contacto después de que el restaurante y las mascotas estén guarc
        let contacto1 = new Contacto({
            nombre: "Nacho",
            telefono: 677889900,
            edad: 40,
            restauranteFavorito: restauranteGuardado._id,
            mascotas: [mascotaGuardada1._id, mascotaGuardada2._id]
        });

        // Guardar el contacto
        let contactoGuardado = await contacto1.save();
        console.log("Contacto guardado correctamente:", contactoGuardado);

    } catch (error) {
        console.error("Error al guardar los datos relacionados:", error);
    }
}

// Llamar a la función
guardarContactoConRelaciones();
```

Solución 3: ejecución paralela de inserciones independientes con Promise.all()

Aún podemos mejorar la solución propuesta, ya que la inserción del restaurante y de las mascotas son independientes entre sí. Esto significa que, en lugar de anidar estas operaciones y ejecutarlas de forma secuencial, pueden ejecutarse en paralelo, optimizando así el proceso. Para ello, podemos usar **Promise.all()** para esperar a que todas las promesas (inserciones) se resuelvan y luego proceder con la creación del contacto.

```
Promise.all([restaurante1.save(), mascota1.save(), mascota2.save()])
    .then(([restauranteGuardado, mascotaGuardada1, mascotaGuardada2]) => {
        // Una vez que se hayan guardado el restaurante y las mascotas, podemos crear
        let contacto1 = new Contacto({
            nombre: "Nacho",
            telefono: 677889900,
            edad: 40,
            restauranteFavorito: restauranteGuardado._id, // Referencia al _id del re
            mascotas: [mascotaGuardada1._id, mascotaGuardada2._id] // Referencia a lo

        });

        // Guardar el contacto
        return contacto1.save();
    })
    .then((contactoGuardado) => {
        console.log("Contacto guardado correctamente:", contactoGuardado);
    })
    .catch((error) => {
        console.error("Error en el proceso de inserción:", error);
    });
```

Solución 4: uso de async/await con inserciones paralelas

También podemos combinar `async/await` con inserciones en paralelo para obtener un código limpio y eficiente.

```
async function guardarContactoConRelaciones() {
    try {
        // Guardar restaurante y mascotas en paralelo
        const [restauranteGuardado, mascotaGuardada1, mascotaGuardada2] = await Prom
            restaurante1.save(),
            mascota1.save(),
            mascota2.save()
        ]);

        // Crear y guardar el contacto una vez que las inserciones anteriores hayan t
        let contacto1 = new Contacto({
            nombre: "Nacho",
            telefono: 677889900,
            edad: 40,
            restauranteFavorito: restauranteGuardado._id, // Referencia al _id del re
            mascotas: [mascotaGuardada1._id, mascotaGuardada2._id] // Referencia a lo
        });

        let contactoGuardado = await contacto1.save();
        console.log("Contacto guardado correctamente:", contactoGuardado);

    } catch (error) {
        console.error("Error en el proceso de inserción:", error);
    }
}

// Llamar a la función
guardarContactoConRelaciones();
```

La combinación de **async/await con Promise.all()** es la solución más recomendada, ya que combina claridad y eficiencia en el manejo de operaciones asíncronas.

1.4. Sobre la integridad referencial

La integridad referencial es un concepto vinculado a bases de datos relacionales, mediante el cual se garantiza que los valores de una clave ajena siempre van a existir en la tabla a la que hace referencia. Aplicado a una base de datos Mongo, podríamos pensar que los *ids* de un campo vinculado a otra colección deberían existir en dicha colección, pero no tiene por qué ser así.

Siguiendo con el ejemplo anterior, si intentamos insertar un contacto con un *id* de restaurante que no exista en la colección de restaurantes, nos dejará hacerlo, siempre que ese *id* sea válido (es decir, tenga una extensión de 12 bytes). Por lo tanto, corre por cuenta del programador asegurarse de que los *id* empleados en inserciones que impliquen una referencia a otra colección existan realmente. Para facilitar la tarea, existen algunas librerías en el repositorio NPM que podemos emplear, como por ejemplo ésta, aunque su uso va más allá de los contenidos de este curso, y no lo veremos aquí.

En el caso del borrado, podemos encontrarnos con una situación similar: si, siguiendo con el caso de los contactos, queremos borrar un restaurante, deberemos tener cuidado con los contactos que lo tienen asignado como restaurante favorito, ya que el *id* dejará de existir en la colección de restaurantes. Así, sería conveniente elegir entre una de estas dos opciones, aunque las dos requieren un tratamiento manual por parte del programador:

- Denegar la operación si existen contactos con el restaurante seleccionado
- Reasignar (o poner a nulo) el restaurante favorito de esos contactos, antes de eliminar el restaurante seleccionado.

> **Ejercicio 1:**
>
> Vamos a modificar el ejercicio *LibrosMongo* iniciado en la sesión anterior. Haz una copia y renómbrala a **LibrosMongo_v2** para trabajar ahora con esta nueva versión.
>
> - Vamos a definir ahora un segundo esquema para almacenar información sobre el **autor** de cada libro. Dicho autor tendrá un nombre (obligatorio) y un año de nacimiento (opcional, pero con valores entre 0 y 2000). Define también el modelo para la colección, asociado a este esquema.
> - Después, relaciona la colección de libros con la de autores, añadiendo a la primera un nuevo campo llamado `autor`, que enlazará con el *id* del autor correspondiente en la colección de autores. Dicho campo *autor* no será obligatorio, para respetar así los libros sin autor que tengamos añadidos con anterioridad.
> - Tras la conexión a la base de datos y la definición de esquemas que hemos hecho, elimina el código del ejercicio anterior relativo a inserciones, borrados, modificaciones y consultas, y añade el código para insertar uno o dos nuevos autores, y algún libro vinculado a cada uno de ellos.

2. Subdocumentos

Mongoose ofrece también la posibilidad de definir **subdocumentos**. Veamos un ejemplo concreto de ello, y para eso, vamos a hacer una versión alternativa de nuestro ejemplo de contactos. Copia la carpeta *ContactosMongo_v2* que hemos venido completando hasta ahora, y llama a la nueva copia *ContactosMongo_v3*.

Sobre este nuevo proyecto, en nuestro archivo `index.js`, vamos a conectar con una nueva base de datos, que llamaremos `contactos_subdocumentos`, para no interferir con la base de datos anterior:

```
mongoose.connect('mongodb://127.0.0.1:27017/contactos_subdocumentos');
```

Y vamos a reagrupar los tres esquemas que hemos hecho hasta ahora (restaurantes, mascotas y contactos), para unirlos en el de contactos. Dejaremos, por tanto, un único archivo en la carpeta `models`, que será `contacto.js`, con este contenido (omitimos con puntos suspensivos parte del código que es el mismo del ejemplo anterior):

```
// Restaurantes
let restauranteSchema = new mongoose.Schema({
    ... // Código del esquema de restaurante
});

// Mascotas
let mascotaSchema = new mongoose.Schema({
    ... // Código del esquema de mascota
});

// Contactos
let contactoSchema = new mongoose.Schema({
    nombre: {
        ...
    },
    telefono: {
        ...
    },
    edad: {
        ...
    },
    restauranteFavorito: restauranteSchema,
    mascotas: [mascotaSchema]
});
let Contacto = mongoose.model('contactos', contactoSchema);
module.exports = Contacto;
```

Observad las líneas que se refieren a las propiedades `restauranteFavorito` y `mascotas`. Es la forma de asociar un esquema entero como tipo de dato de un campo de otro esquema. De este modo, convertimos el esquema en una parte del otro, creando así **subdocumentos** dentro del documento principal. Observad también que no se han definido modelos ni para los restaurantes ni para las mascotas, ya que ahora no van a tener una colección propia.

Un subdocumento, a priori, puede parecer algo equivalente a definir una relación entre colecciones. Sin embargo, la principal diferencia entre un subdocumento y una relación entre documentos de colecciones diferentes es que el subdocumento queda embebido dentro del documento principal, y es diferente a cualquier otro objeto que pueda haber en otro documento, aunque sus campos sean iguales. Por el contrario, en la relación simple vista antes entre restaurantes y contactos, un restaurante favorito podía ser compartido por varios contactos, simplemente enlazando con el mismo *id* de restaurante. Pero, de este otro modo, creamos el restaurante para cada contacto, diferenciándolo de los otros restaurantes, aunque sean iguales. Lo mismo ocurriría con el array de mascotas: las mascotas serían diferentes para cada contacto, aunque quisiéramos que fueran la misma o pudieran compartirse.

2.1. Inserción de documentos con subdocumentos

Si queremos crear y guardar un contacto que contiene como subdocumentos el restaurante favorito y sus mascotas, podemos crear todo el objeto completo, y hacer un único guardado (`save`).

```
let contacto1 = new Contacto({
    nombre: 'Nacho',
    telefono: 966112233,
    edad: 39,
    restauranteFavorito: {
        nombre: 'La Tagliatella',
        direccion: 'C.C. San Vicente s/n',
        telefono: 961234567
    }
});
contacto1.mascotas.push({nombre:'Otto', tipo:'perro'});
contacto1.mascotas.push({nombre:'Piolín', tipo:'otros'});
contacto1.save().then(...
```

En este ejemplo se muestran dos formas posibles de rellenar los subdocumentos del documento principal: sobre la marcha cuando creamos dicho documento (caso del restaurante), o a posteriori, accediendo a los campos y dándoles valor (caso de las mascotas).

En la base de datos que se crea, veremos que sólo existe una colección, *contactos*, y al examinar los elementos que insertemos veremos que contienen embebidos los subdocumentos que hemos definido:

2.2. ¿Cuándo definir relaciones y cuándo subdocumentos?

La respuesta a esta pregunta puede resultar compleja o evidente, dependiendo de cómo hayamos entendido los conceptos vistos hasta ahora, pero vamos a intentar dar unas normas básicas para distinguir cuándo usar cada concepto:

- Emplearemos **relaciones entre colecciones** cuando queramos poder compartir un mismo documento de una colección por varios documentos de otra. Así, en el caso de los restaurantes favoritos, parece lógico utilizar una relación entre colecciones a partir del *id* del restaurante, y así permitir que varios contactos puedan compartir una misma instancia de un restaurante favorito.
- Emplearemos **subdocumentos** cuando no importe dicha compartición de información, o cuando prime la simplicidad de definición de un objeto frente a la asociatividad entre colecciones. Dicho de otro modo, y aplicado al ejemplo de las mascotas, si queremos acceder de forma sencilla a las mascotas de un contacto, sin importar si otro contacto tiene las mismas mascotas, utilizaremos subdocumentos.

En el caso de los subdocumentos queda, por tanto, una asignatura pendiente: la posible duplicidad de información. Si hay dos personas que tienen la misma mascota, deberemos crear dos objetos iguales para ambas personas, duplicando así los datos de la mascota. Sin embargo, esta duplicidad de datos nos va a facilitar el acceder a las mascotas de una persona, sin tener que recurrir a otras herramientas que veremos a continuación.

> **Ejercicio 2:**
>
> Sobre el ejercicio anterior, define un nuevo esquema en el fichero de `models/libro.js` para almacenar comentarios relativos a un libro. Cada comentario tendrá una fecha (tipo `Date`), el nick de quien hace el comentario (`String`) y el comentario en sí (`String`), siendo todos estos campos obligatorios. Además, en el caso de la fecha, estableceremos como valor por defecto (`default`) la fecha actual (`Date.now`).
>
> Esta vez no definas un modelo para este esquema. Vamos a crear un subdocumento dentro del esquema de libros que almacene un array de comentarios para dicho libro, utilizando el esquema de comentarios que acabas de crear.
>
> Una vez hecho esto, crea un nuevo libro con sus datos, y añade a mano un par de comentarios al array, antes de guardar todos los datos.

3. Consultas avanzadas

Ahora que ya sabemos definir distintos tipos de colecciones vinculadas entre sí, veamos cómo definir consultas que se aprovechen de estas vinculaciones para extraer la información que necesitamos. Volveremos a trabajar, en este caso, con el proyecto *ContactosMongo_v2*.

3.1. Las poblaciones (*populate*)

El hecho de relacionar documentos de una colección con documentos de otra a través de los *id* correspondientes, permite obtener en un solo listado la información de ambas colecciones, aunque para ello necesitamos de algún paso intermedio. Por ejemplo, si queremos obtener toda la información de nuestros

contactos, relacionados con las colecciones de restaurantes y mascotas (archivo `index.js` de nuestro proyecto de "*ContactosMongo_v2*"), podemos hacer algo como esto:

```
Contacto.find().then(resultado => {
    console.log(resultado);
});
```

Sin embargo, esta instrucción se limita, obviamente, a mostrar el id de los restaurantes favoritos y de las mascotas, pero no los datos completos de las mismas. Para hacer esto, tenemos que echar mano de un método muy útil ofrecido por Mongoose, llamado `populate`. Este método permite incorporar la información asociada al modelo que se le indique. Por ejemplo, si queremos incorporar al listado anterior toda la información del restaurante favorito de cada contacto, haremos algo así:

```
Contacto.find().populate('restauranteFavorito').then(resultado => {
    console.log(resultado);
});
```

Si tuviéramos más campos relacionados, podríamos enlazar varias sentencias `populate`, una tras otra, para poblarlos. Por ejemplo, así poblaríamos tanto el restaurante como las mascotas:

```
Contacto.find()
.populate('restauranteFavorito')
.populate('mascotas')
.then(resultado => {
    console.log(resultado);
});
```

Existen otras opciones para poblar los campos. Por ejemplo, podemos querer poblar sólo parte de la información, como el nombre del restaurante nada más. En ese caso, utilizamos una serie de parámetros adicionales en el método `populate`:

```
Contacto.find()
.populate('restauranteFavorito', 'nombre')
...
```

3.2. Consultas que relacionan varias colecciones

Establecer una consulta general sobre una colección es sencillo, como hemos visto en sesiones anteriores. Podemos utilizar el método `find` para obtener documentos que cumplan determinados criterios, o alternativas como `findOne` o `findById` para obtener el documento que cumpla el filtrado.

Las bases de datos No-SQL, como es el caso de MongoDB, no están preparadas para consultar información proveniente de varias colecciones, lo que en parte "invita" a utilizar colecciones independientes basadas en subdocumentos para agregarles información adicional.

Supongamos que queremos, por ejemplo, obtener los datos de los restaurantes favoritos de aquellos contactos que sean mayores de 30 años. Si tuviéramos una base de datos SQL, podríamos resolver esto con una query como la siguiente:

```
SELECT * FROM restaurantes
WHERE id IN
(SELECT restauranteFavorito FROM contactos
WHERE edad > 30)
```

Sin embargo, esto no es posible en MongoDB o, al menos, no de forma tan inmediata. Haría falta dividir esta consulta en dos partes: primero obtener los *id* de los restaurantes de las personas mayores de 30 años, y a partir de ahí obtener con otra consulta los datos de esos restaurantes. Podría quedar más o menos así:

```
Contacto.find({edad: {$gt: 30}}).then(resultadoContactos => {
    let idsRestaurantes =
       resultadoContactos.map(contacto => contacto.restauranteFavorito);
    Restaurante.find({_id: {$in: idsRestaurantes}})
    .then(resultadoFinal => {
        console.log(resultadoFinal);
    });
});
```

Observad que la primera consulta obtiene todos los contactos mayores de 30 años. Una vez conseguidos, hacemos un mapeo (`map`) para quedarnos sólo con los *id* de los restaurantes favoritos, y ese listado de *ids* lo utilizamos en la segunda consulta, para quedarnos con los restaurantes cuyo *id* esté en ese listado.

> **Ejercicio 3**:
>
> Antes de seguir con este ejercicio, procura que haya al menos dos o tres autores en la colección de autores, y al menos tres o cuatro libros con autores diferentes. Una vez hecho eso, añade al programa anterior una consulta que muestre los nombres de los autores que tengan algún libro a la venta por menos de 10 euros (únicamente deberán mostrarse los nombres de los autores en el listado).

3.3. Otras opciones en las consultas

Cuando utilizamos el método `find` o similares, existen opciones adicionales que permiten, por ejemplo, especificar qué campos queremos obtener, o criterios de ordenación, o de límite máximo de resultados a obtener, etc. En la anterior sesión vimos algún ejemplo al respecto, pero veamos ahora con algo más de detalle algunas de estas opciones:

- Si queremos especificar concretamente qué campos queremos obtener en el listado, se especifica como una cadena de texto en los parámetros de `find`. Como alternativa, se puede enlazar la llamada normal a `find` con el método `select`, donde se especifican los campos a obtener. Estos dos ejemplos hacen lo mismo: mostrar el nombre y la edad de los contactos mayores de 30 años:

```
Contacto.find({edad: {$gt: 30}}, 'nombre edad').then(...
Contacto.find({edad: {$gt: 30}}).select('nombre edad').then(...
```

- Para **ordenar** el listado por algún criterio, se enlaza la llamada a `find` con otra al método `sort`, en el que se especifica el campo por el que ordenar, y el orden (1 para orden ascendente, -1 para orden descendente). El siguiente listado muestra ordenados de mayor a menor edad los contactos. Se indica también una alternativa equivalente, que consiste en anteponer el signo menos `-` al campo por el que ordenar, indicando que se quiere un orden descendente:

```
Contacto.find().sort({edad: -1}).then(...
Contacto.find().sort('-edad').then(...
```

- Para **limitar el número de resultados a obtener**, se emplea el método `limit`, indicando cuántos documentos obtener. El siguiente ejemplo muestra, de mayor a menor edad, los 5 primeros contactos:

```
Contacto.find().sort('-edad').limit(5).then(...
```

> **Ejercicio 4:**
>
> Añade al ejercicio anterior una consulta que muestre el título y precio (junto con el *id*) de los tres libros más baratos, ordenados de menor a mayor precio. En el caso de que haya menos de tres libros en el listado, se mostrarán sólo los libros disponibles, obviamente.

> **Ejercicio 5:** Consultas extra sobre la colección de libros:
>
> - **Buscar libros por editorial:** realiza una consulta que muestre el título y la editorial de todos los libros que pertenezcan a una editorial específica.
> - **Filtrar libros ordenados por rango de precios:** realiza una consulta que devuelva los libros cuyo precio esté entre dos valores dados, ordenados de mayor a menor precio.
> - **Libros sin comentarios:** realiza una consulta que devuelva todos los libros que no tengan comentarios asociados. Filtra aquellos libros cuyo array de comentarios esté vacío o no exista. Puedes utilizar los operadores `$exists` (para verificar si un campo esta presente) y `$size` (para saber la cantidad de elementos en el array).
> - **Actualización masiva de precios:** crea una consulta que actualice el precio de todos los libros de una editorial específica, incrementándolo en un 10%. Muestra los resultados actualizados después de realizar la operación. Puedes utilizar el operador `$mul` (multiplica el valor de un campo por un

número determinado en una operación de actualización). No es necesario usar `$set` en este caso, porque `$mul` directamente modifica el valor del campo. `$set` se usa cuando se asigna un valor nuevo, pero aquí solo se está modificando el valor existente mediante una multiplicación.

- **Buscar libros por autor nacido en un rango de años**: realiza una consulta que muestre los libros cuyo autor haya nacido entre dos años específicos. Utiliza la relación entre las colecciones de Libro y Autor para realizar esta consulta.
- **Listar los comentarios de un libro específico:** realiza una consulta que recupere el título de un libro y todos los comentarios asociados, utilizando su ID.