4. Acceso a bases de datos NoSQL: MongoDB

4.1. Introducción a MongoDB

En esta sesión daremos unas nociones básicas de cómo conectar y gestionar una base de datos MongoDB desde Node. Para los no iniciados en el tema, MongoDB es el principal representante, actualmente, de los sistemas de bases de datos NoSQL. Estos sistemas se han vuelto muy populares en los últimos años, y permiten dotar de persistencia a los datos de nuestra aplicación de una forma diferente a los tradicionales sistemas SQL.

En lugar de almacenar la información en tablas con sus correspondientes campos y registros, lo que haremos será almacenar estructuras de datos en formato BSON (similar a JSON), lo que facilita la integración con ciertas aplicaciones, como las aplicaciones Node.

4.1.1. Algunos conceptos de bases de datos NoSQL

Las bases de datos NoSQL tienen algunas similitudes y diferencias con las tradicionales bases de datos SQL. Entre las similitudes, las dos trabajan con bases de datos, es decir, lo que creamos en uno u otro gestor es siempre una base de datos, pero la principal diferencia radica en cómo se almacenan los datos. En una base de datos SQL, la información se almacena en forma de tablas, mientras que en una NoSQL lo que se almacena se denominan **colecciones** (arrays de objetos en formato BSON, en el caso de Mongo). Las tablas están compuestas de registros (cada fila de la tabla), mientras que las colecciones se componen de **documentos** (cada objeto de la colección). Finalmente, cada registro de una tabla SQL tiene una serie de campos fijos (todos los registros de la tabla tienen los mismos campos), mientras que en una colección NoSQL, cada documento puede tener un conjunto diferente de **propiedades** (que también se suelen llamar campos). En cualquier caso, lo habitual es que los documentos de una misma colección compartan las mismas propiedades.

4.1.2. Acceso a sistemas MongoDB

Como decíamos, el ejemplo más representativo de los sistemas NoSQL actualmente vigentes es MongoDB, un sistema de bases de datos de código abierto y multiplataforma, que podemos instalar en sistemas Windows, Mac OSX o Linux. En el documento de instalación ya se han visto los pasos para instalarlo, y en la máquina virtual completa del curso ya se proporciona instalado junto con la herramienta que utilizaremos para acceder a él, llamada Robo3T, por lo que a estas alturas ya deberías tenerlo listo para funcionar.

Si utilizas la máquina virtual con todo el software instalado, recuerda que dispones de un acceso directo en el escritorio para poner en marcha el servidor MongoDB, y otro para ejecutar Robo3T y conectar con el servidor

para examinar las bases de datos. En el documento de instalación se explicaba cómo poner en marcha ambas cosas y acceder a MongoDB desde Robo3T.

4.2. La librería *mongoose*. Primeros pasos

Existen varias librerías en el repositorio oficial de NPM para gestionar bases de datos MongoDB, pero la más popular es Mongoose. Permite acceder de forma fácil a las bases de datos y, además, definir esquemas, una estructura de validación que determina el tipo de dato y rango de valores adecuado para cada campo de los documentos de una colección. Así, podemos establecer si un campo es obligatorio o no, si debe tener un valor mínimo o máximo, etc. En la web oficial de Mongoose podemos consultar algunos ejemplos de definición de esquemas y documentación adicional.

4.2.1. Carga de la librería y conexión al servidor

A lo largo de esta sesión vamos a hacer algunas pruebas con Mongoose en un proyecto que llamaremos "*PruebaContactosMongo*". Podemos crearlo ya en nuestra carpeta "*ProyectosNode/Pruebas*". Después, definiremos el archivo `package.json` con el comando `npm init`, y posteriormente instalaremos mongoose en el proyecto con el comando `npm install`.

```
npm install mongoose
```

Una vez instalado, necesitamos incorporarlo al código del proyecto con la correspondiente instrucción `require`. Crea un archivo fuente `index.js` en este proyecto de pruebas, e incorpora la librería de este modo::

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

Para conectar con el servidor Mongo (y suponiendo que lo tenemos iniciado, siguiendo los pasos indicados en el documento de instalación), necesitamos llamar a un método llamado `connect`, dentro del objeto `mongoose` que hemos incorporado. Le pasaremos la URL de la base de datos como primer parámetro, y un objeto con propiedades de conexión como segundo parámetro. A lo largo de las distintas versiones de Mongoose ha ido variando lo que había que incluir dentro de este objeto. Actualmente, debemos indicar una propiedad `useNewUrlParser` a `true` para que el driver Node.js que conecta con MongoDB utilice la nueva forma que incorpora de parsear la URL de conexión a la base de datos, y también debemos añadir una segunda propiedad `useUnifiedTopology` a `true`. Si no ponemos estas propiedades, obtendremos un *warning* indicando que esa configuración no es la recomendada, pero aún así podremos establecer la conexión.

Por ejemplo, con esta instrucción conectamos con una base de datos llamada *contactos* en el servidor local:

```
mongoose.connect('mongodb://localhost:27017/contactos',
    { useNewUrlParser: true, useUnifiedTopology: true });
```

No os preocupéis por que la base de datos no exista. Se creará automáticamente tan pronto como añadamos datos en ella.

4.2.2. Modelos y esquemas

Como comentábamos antes, la librería Mongoose permite definir la estructura que van a tener los documentos de las distintas colecciones de la base de datos. Para ello, se definen esquemas (*schemas*) y se asocian a modelos (las colecciones correspondientes en la base de datos).

4.2.2.1. Definir los esquemas

Para definir un esquema, necesitamos crear una instancia de la clase `Schema` de Mongoose. Por lo tanto, crearemos este objeto, y en esa creación definiremos los atributos que va a tener la colección correspondiente, junto con el tipo de dato de cada atributo. Es también recomendable separar estas definiciones en archivos aparte. Podemos crear una subcarpeta `models` y almacenar en ella los esquemas y modelos de nuestra base de datos.

En el caso de la base de datos de contactos propuesta para estas pruebas, podemos definir un esquema para almacenar los datos de cada contacto: nombre, número de teléfono y edad, por ejemplo. Esto lo haríamos de esta forma, en un archivo llamado `contacto.js` dentro de la subcarpeta `models` de nuestro proyecto:

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

let contactoSchema = new mongoose.Schema({
    nombre: String,
    telefono: String,
    edad: Number
});
```

Los tipos de datos disponibles para definir el esquema son:

- Textos (`String`)
- Números (`Number`)
- Fechas (`Date`)
- Booleanos (`Boolean`)
- Arrays (`Array`)
- Otros (veremos algunos más adelante): `Buffer` , `Mixed` , `ObjectId`

4.2.2.2. Aplicar el esquema a un modelo

Una vez definido el esquema, necesitamos aplicarlo a un modelo para asociarlo así a una colección en la base de datos. Para ello, disponemos del método `model` en Mongoose. Como primer parámetro, indicaremos el nombre de la colección a la que asociar el esquema. Como segundo parámetro, indicaremos el esquema a aplicar (objeto de tipo `Schema` creado anteriormente). Añadiríamos estas líneas al final de nuestro archivo `models/contacto.js`:

```
let Contacto = mongoose.model('contactos', contactoSchema);
module.exports = Contacto;
```

> **NOTA:** si indicamos un nombre de modelo en singular, Mongoose automáticamente creará la colección con el nombre en plural. Este plural no siempre será correcto, ya que lo que hace es simplemente añadir una "s" al final del nombre del modelo, si no se la hemos añadido nosotros. Por este motivo, es recomendable que creemos los modelos con nombres de colecciones ya en plural.

4.2.2.3. Restricciones y validaciones

Si definimos un esquema sencillo como el ejemplo de contactos anterior, permitiremos que se añada cualquier tipo de valor a los campos de los documentos. Así, por ejemplo, podríamos tener contactos sin nombre, o con edades negativas. Pero con Mongoose podemos proporcionar mecanismos de validación que permitan descartar de forma automática los documentos que no cumplan las especificaciones.

En la documentación oficial de Mongoose podemos encontrar una descripción detallada de los diferentes validadores que podemos aplicar. Aquí nos limitaremos a describir los más importantes o habituales:

- El validador `required` permite definir que un determinado campo es obligatorio.
- El validador `default` permite especificar un valor por defecto para el campo, en el caso de que no se especifique ninguno.
- Los validadores `min` y `max` se utilizan para definir un rango de valores (mínimo y/o máximo) permitidos para datos de tipo numérico.
- Los validadores `minlength` y `maxlength` se emplean para definir un tamaño mínimo o máximo de caracteres, en el caso de cadenas de texto.
- El validador `unique` indica que el campo en cuestión no admite duplicados (sería una clave alternativa, en un sistema relacional). En la documentación de Mongoose se especifica que esto no es propiamente un validador, sino una ayuda para indexar, y que dependiendo de cuándo se indexe, es posible que no funcione adecuadamente.
- El validador `match` se emplea para especificar una expresión regular que debe cumplir el campo (aquí tenéis más información al respecto).
- ...

Volvamos a nuestro esquema de contactos. Vamos a establecer que el nombre y el teléfono sean obligatorios, y sólo permitiremos edades entre 18 y 120 años (inclusive). Además, el nombre tendrá una longitud mínima de 1 carácter, y el teléfono estará compuesto por 9 dígitos, empleando una expresión regular, y será una clave única. Podemos emplear algún validador más, como por ejemplo `trim`, para limpiar los espacios en blanco

al inicio y final de los datos de texto. Con todas estas restricciones, el esquema y modelo asociado quedan de esta forma:

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

let contactoSchema = new mongoose.Schema({
    nombre: {
        type: String,
        required: true,
        minlength: 1,
        trim: true
    },
    telefono: {
        type: String,
        required: true,
        unique: true,
        trim: true,
        match: /^\d{9}$/
    },
    edad: {
        type: Number,
        min: 18,
        max: 120
    }
});

let Contacto = mongoose.model('contactos', contactoSchema);
module.exports = Contacto;
```

Ya tenemos establecida la conexión a la base de datos, y el esquema de los datos que vamos a utilizar. Ahora, podemos añadir el modelo a nuestro archivo principal `index.js`, y ya podremos empezar a realizar algunas operaciones básicas contra dicha base de datos.

```
const mongoose = require('mongoose');
const Contacto = require(__dirname + "/models/contacto");

mongoose.connect('mongodb://localhost:27017/contactos',
    { useNewUrlParser: true, useUnifiedTopology: true });

// Aquí ya podremos realizar operaciones contra la BD
```

4.2.3. Añadir documentos

Si queremos insertar o añadir un documento en una colección, debemos crear un objeto del correspondiente modelo, y llamar a su método `save`. Este método devuelve una promesa, por lo que emplearemos:

- Un bloque de código `then` para cuando la operación haya ido correctamente. En este bloque, recibiremos como resultado el objeto que se ha insertado, pudiendo examinar los datos del mismo si se quiere.
- Un bloque de código `catch` para cuando la operación no haya podido completarse. Recibiremos como parámetro un objeto con el error producido, que podremos examinar para obtener más información sobre el mismo.

Este mismo patrón *then-catch* lo emplearemos también con el resto de operaciones más adelante (búsquedas, borrados o modificaciones), aunque el resultado devuelto en cada caso variará.

Así añadiríamos un nuevo contacto a nuestra colección de pruebas:

```
let contacto1 = new Contacto({
    nombre: "Nacho",
    telefono: "966112233",
    edad: 39
});
contacto1.save().then(resultado => {
    console.log("Contacto añadido:", resultado);
}).catch(error => {
    console.log("ERROR añadiendo contacto:", error);
});
```

Añade este código al archivo `index.js` de nuestro proyecto "*PruebaContactosMongo*", tras la conexión a la base de datos. Ejecuta la aplicación, y echa un vistazo al resultado que se devuelve cuando todo funciona correctamente. Será algo parecido a esto:

```
{ __v: 0,
  nombre: 'Nacho',
  telefono: '966112233',
  edad: 39,
  _id: 5a12a2e0e6219d68c00c6a00 }
```

Observa que obtenemos los mismos campos que definimos en el esquema (nombre, teléfono y edad), y dos campos adicionales que no hemos especificado:

- `__v` hace referencia a la versión del documento. Inicialmente, todos los documentos parten de la versión 0 cuando se insertan, y luego esta versión puede modificarse cuando hagamos alguna actualización del documento, como veremos después.
- `_id` es un código autogenerado por Mongo, para cualquier documento de cualquier colección que se tenga. Analizaremos en qué consiste este id y su utilidad en breve.

Vayamos ahora a Robo 3T y examinemos las bases de datos en el panel izquierdo. Si clicamos en la colección de "contactos", veremos el nuevo contacto añadido en el panel derecho:

Si intentamos insertar un contacto incorrecto, saltaremos al bloque `catch`. Por ejemplo, este contacto es demasiado viejo, según la definición del esquema:

```
let contacto2 = new Contacto({
    nombre: "Matuzalem",
    telefono: "965123456",
    edad: 200
});
contacto2.save().then(resultado => {
    console.log("Contacto añadido:", resultado);
}).catch(error => {
    console.log("ERROR añadiendo contacto:", error);
});
```

Si echamos un vistazo al error producido, veremos mucha información, pero entre toda esa información hay un atributo llamado `ValidationError` con la información del error:

```
ValidationError: contacto validation failed: edad:
Path `edad` (200) is more than maximum allowed value (120)
```

4.2.3.1. Sobre el *id* automático

Como has podido ver en las pruebas de inserción anteriores, cada vez que se añade un documento a una colección se le asigna automáticamente una propiedad llamada `_id` con un código autogenerado. A diferencia de otros sistemas de gestión de bases de datos (como MariaDB/MySQL, por ejemplo), este código no es autonumérico, sino que es una cadena. De hecho, es un texto de 12 bytes que almacena información importante:

- El tiempo de creación de documento (*timestamp*), con lo que podemos obtener el momento exacto (fecha y hora) de dicha creación
- El ordenador que creó el documento. Esto es particularmente útil cuando queremos escalar la aplicación y tenemos distintos servidores Mongo accediendo a la misma base de datos. Podemos identificar cuál de todos los servidores fue el que creó el documento.
- El proceso concreto del sistema que creó el documento

- Un contador aleatorio, que se emplea para evitar cualquier tipo de duplicidad, en el caso de que los tres valores anteriores coincidan en el tiempo.

Existen métodos específicos para extraer parte de esta información, en concreto el momento de creación, pero no los utilizaremos en este curso.

A pesar de disponer de esta enorme ventaja con este id autogenerado, podemos optar por crear nuestros propios ids y no utilizar los de Mongo (aunque ésta no es una buena idea):

```
let contactoX = new Contacto({_id:2, nombre:"Juan",
    telefono:"611885599"});
```

4.2.4. Buscar documentos

Si queremos buscar cualquier documento, o conjunto de documentos, en una colección, podemos emplear diversos métodos.

4.2.4.1. Búsqueda genérica con *find*

La forma más general de obtener documentos consiste en emplear el método estático `find` asociado al modelo en cuestión. Podemos emplearlo sin parámetros (con lo que obtendremos todos los documentos de la colección como resultado de la promesa):

```
Contacto.find().then(resultado => {
    console.log(resultado);
}).catch (error => {
    console.log("ERROR:", error);
});
```

4.2.4.2. Búsqueda parametrizada con *find*

Podemos también pasar como parámetro a `find` un conjunto de criterios de búsqueda. Por ejemplo, para buscar contactos cuyo nombre sea "Nacho" y la edad sea de 29 años, haríamos esto:

```
Contacto.find({nombre: 'Nacho', edad: 29}).then(resultado => {
    console.log(resultado);
}).catch (error => {
    console.log("ERROR:", error);
});
```

> **NOTA:** cualquier llamada a `find` devolverá un **array de resultados**, aunque sólo se haya encontrado uno, o ninguno. Es importante tenerlo en cuenta para luego saber cómo acceder a un elemento

> concreto de dicho resultado. El hecho de no obtener resultados no va a provocar un error (no se saltará al `catch` en ese caso).

También podemos emplear algunos operadores de comparación en el caso de no buscar datos exactos. Por ejemplo, esta consulta obtiene todos los contactos cuyo nombre sea "Nacho" y las edades estén comprendidas entre 18 y 40 años:

```
Contacto.find({nombre:'Nacho', edad: {$gte: 18, $lte: 40}})
.then(resultado => {
    console.log('Resultado de la búsqueda:', resultado);
})
.catch(error => {
    console.log('ERROR:', error);
});
```

Aquí podéis encontrar un listado detallado de los operadores que podéis utilizar en las búsquedas.

Además, la búsqueda parametrizada con `find` admite otras variantes de sintaxis, como el uso de métodos enlazados `where`, `limit`, `sort`... hasta obtener los resultados deseados en el orden y cantidad deseada. Por ejemplo, esta consulta muestra los 10 primeros contactos mayores de edad, ordenados de mayor a menor edad:

```
Contacto.find()
.where('edad')
.gte(18)
.sort('-edad')
.limit(10)
.then(...
```

4.2.4.3. Otras opciones: *findOne* o *findById*

Existen otras alternativas que podemos utilizar para buscar documentos concretos (y no un conjunto o lista de ellos). Se trata de los métodos `findOne` y `findById`. El primero se emplea de forma similar a `find`, con los mismos parámetros de filtrado, pero sólo devuelve un documento (arbitrario) que concuerde con esos criterios (no un array). Por ejemplo:

```
Contacto.findOne({nombre:'Nacho', edad: 39})
.then(resultado => {
    console.log('Resultado de la búsqueda:', resultado);
})
.catch(error => {
    console.log('ERROR:', error);
});
```

El método `findById` se emplea, como su nombre indica, para buscar un documento dado su *id* (recordemos, esa secuencia de 12 bytes autogenerada por Mongo). Por ejemplo:

```
Contacto.findById('5ab2dfb06cf5de1d626d5c09')
.then(resultado => {
    console.log('Resultado de la búsqueda por ID:', resultado);
})
.catch(error => {
    console.log('ERROR:', error);
});
```

En estos métodos, si la consulta no produce ningún resultado, obtendremos `null` como respuesta, pero tampoco se activará la cláusula `catch` por ello.

4.2.5. Borrar documentos

Para eliminar documentos de una colección, podemos emplear los métodos estáticos `remove`, `findOneAndRemove` y `findByIdAndRemove`.

4.2.5.1. El método *remove*

Este método elimina los documentos que cumplan los criterios indicados como parámetro. Estos criterios se especifican de la misma forma que hemos visto para el método `find`. Si no se especifican parámetros, se eliminan TODOS los documentos de la colección.

```
// Eliminamos todos los contactos que se llamen Nacho
Contacto.remove({nombre: 'Nacho'}).then(resultado => {
    console.log(resultado);
}).catch (error => {
    console.log("ERROR:", error);
});
```

El resultado que se obtiene en este caso contiene múltiples propiedades. En la propiedad `result` podemos consultar el número de filas afectadas (`n`), y el resultado de la operación (`ok`).

```
CommandResult {
  result: { n: 1, ok: 1 },
  connection:
  ...
```

4.2.5.2. Los métodos *findOneAndRemove* y *findByIdAndRemove*

El método `findOneAndRemove` busca el documento que cumpla el patrón especificado (o el primero que encuentre que lo cumpla) y lo elimina. Además, obtiene el documento eliminado en el resultado, con lo que podríamos deshacer la operación a posteriori, si quisiéramos, volviéndolo a añadir.

```
Contacto.findOneAndRemove({nombre: 'Nacho'})
.then(resultado => {
    console.log("Contacto eliminado:", resultado);
}).catch (error => {
    console.log("ERROR:", error);
});
```

Observad que, en este caso, el parámetro `resultado` es directamente el objeto eliminado.

El método `findByIdAndRemove` busca el documento con el *id* indicado y lo elimina. También obtiene como resultado el objeto eliminado.

```
Contacto.findByIdAndRemove('5a16fed09ed79f03e490a648')
.then(resultado => {
    console.log("Contacto eliminado:", resultado);
}).catch (error => {
    console.log("ERROR:", error);
});
```

En el caso de estos dos últimos métodos, si no se ha encontrado ningún elemento que cumpla el criterio de filtrado, se devolverá `null` como resultado, es decir, no se activará la cláusula `catch` por este motivo. Sí se activaría dicha cláusula, por ejemplo, si indicamos un *id* con un formato no válido (que no tenga 12 bytes).

4.2.6. Modificaciones o actualizaciones de documentos

Para realizar modificaciones de un documento en una colección, también podemos emplear distintos métodos estáticos.

4.2.6.1. El método *findByIdAndUpdate*

El método `findByIdAndUpdate` buscará el documento con el *id* indicado, y reemplazará los campos atendiendo a los criterios que indiquemos como segundo parámetro.

En este enlace podéis consultar los operadores de actualización que podemos emplear en el segundo parámetro de llamada a este método. El más habitual de todos es `$set`, que recibe un objeto con los pares clave-valor que queremos modificar en el documento original. Por ejemplo, así reemplazamos el nombre y la edad de un contacto con un determinado id, dejando el teléfono sin modificar:

```
Contacto.findByIdAndUpdate('5a0e1991075e9407c4da8b0a',
    {$set: {nombre:'Nacho Iborra', edad: 40}}, {new:true})
.then(resultado => {
    console.log("Modificado contacto:", resultado);
}).catch (error => {
    console.log("ERROR:", error);
});
```

El tercer parámetro que recibe `findByIdAndUpdate` es un conjunto de opciones adicionales. Por ejemplo, la opción `new` que se ha usado en este ejemplo indica si queremos obtener como resultado el nuevo objeto modificado (`true`) o el antiguo antes de modificarse (`false`, algo útil para operaciones de deshacer). Si no se especifica, por defecto se obtiene el viejo (*false*).

4.2.6.2. El método *update*

Este método se puede utilizar de distintas formas, dependiendo de los parámetros que se le pasen. Por defecto, actualiza los datos del primer documento que encaje con los criterios de búsqueda. Por ejemplo, la siguiente instrucción pone a 20 los años del primer contacto que encuentre con nombre "Nacho":

```
Contacto.update({nombre: 'Nacho', {$set: { edad: 20 }})
.then(...
```

Sin embargo, podemos pasarle más opciones como parámetros para, por ejemplo, indicar que haga actualizaciones en bloque (opción `multi`). Podéis consultar su uso en la documentación oficial.

4.2.6.3. Actualizar la versión del documento

Hemos visto que, entre los atributos de un documento, además del *id* autogenerado por Mongo, se crea un número de versión en un atributo `__v`. Este número de versión alude a la versión del documento en sí, de forma que, si posteriormente se modifica (por ejemplo, con una llamada a `findByIdAndUpdate`), se pueda también indicar con un cambio de versión que ese documento ha sufrido cambios desde su versión original. Si quisiéramos hacer eso con el ejemplo anterior, bastaría con añadir el operador `$inc` (junto al `$set` utilizado antes) para indicar que incremente el número de versión, por ejemplo, en una unidad:

```
Contacto.findByIdAndUpdate('5a0e1991075e9407c4da8b0a',
    {$set: {nombre:'Nacho Iborra', edad: 40},
    $inc: {__v: 1}}, {new:true})
.then(...
```

4.2.7. Mongoose y las promesas

Hemos indicado anteriormente que operaciones como `find`, `save` y el resto de métodos que hemos empleado con Mongoose devuelven una promesa, pero eso no es del todo cierto. Lo que devuelven estos métodos es un *thenable*, es decir, un objeto que se puede tratar con el correspondiente método `then`. Sin embargo, existen otras formas alternativas de llamar a estos métodos, y podemos emplear una u otra según nos convenga.

4.2.7.1. Llamadas como simples funciones asíncronas

Los métodos facilitados por Mongoose son simplemente tareas asíncronas, es decir, podemos llamarlas y definir un *callback* de respuesta que se ejecutará cuando la tarea finalice. Para ello, añadimos como parámetro adicional al método el *callback* en cuestión, con dos parámetros: el error que se producirá si el método no se ejecuta satisfactoriamente, y el resultado devuelto si el método se ejecuta sin contratiempos. Si, por ejemplo, queremos buscar un contacto a partir de su *id*, podemos hacer algo así:

```
Contacto.findById('35893ad987af7e87aa5b113c',
(error, contacto) => {
    if (error)
        console.log("Error:", error);
    else
        console.log(contacto);
});
```

Pensemos ahora en algo más complejo: buscamos el contacto por su *id*, una vez finalizado, incrementamos en un año su edad y guardamos los cambios. En este caso, el código puede quedar así:

```
Contacto.findById('35893ad987af7e87aa5b113c',
(error, contacto) => {
    if (error)
        console.log("Error:", error);
    else {
        contacto.edad++;
        contacto.save((error2, contacto2) => {
            if (error2)
                console.log("Error:", error2);
            else
                console.log(contacto2);
        });
    }
});
```

Como podemos ver, al enlazar una llamada asíncrona (`findById`) con otra (`save`), lo que se produce es un anidamiento de *callbacks*, con sus correspondientes estructuras `if..else` . Este fenómeno se conoce como *callback hell* o *pyramid of doom*, porque produce en la parte izquierda del código una pirámide girada (cuyo pico apunta hacia la derecha), que será más grande cuantas más llamadas enlacemos entre sí. Dicho de otro modo, estaremos tabulando cada vez más el código para anidar llamadas dentro de llamadas, y esta gestión puede hacerse difícil de manejar.

4.2.7.2. Llamadas como promesas

Volvamos ahora a lo que sabemos hacer. ¿Cómo enlazaríamos usando promesas las dos operaciones anteriores? Recordemos: buscar un contacto por su *id* e incrementarle su edad en un año.

Podríamos también cometer un *callback hell* anidando cláusulas `then` , con algo así:

```
Contacto.findById('35893ad987af7e87aa5b113c')
.then(contacto => {
    contacto.edad++;
    contacto.save()
    .then(contacto2 => {
        console.log(contacto2);
    }).catch(error2 => {
        console.log("Error:", error2);
    });
}).catch (error => {
    console.log("Error:", error);
});
```

Sin embargo, las promesas permiten concatenar cláusulas `then` sin necesidad de anidarlas, dejando un único bloque `catch` al final para recoger el error que se produzca en cualquiera de ellas. Para ello, basta

con que dentro de un `then` se devuelva (`return`) el resultado de la siguiente promesa. El código anterior podríamos reescribirlo así:

```
Contacto.findById('35893ad987af7e87aa5b113c')
.then(contacto => {
    contacto.edad++;
    return contacto.save();
}).then(contacto => {
    console.log(contacto);
}).catch (error => {
    console.log("Error:", error);
});
```

Esta forma es más limpia y clara cuando queremos hacer operaciones complejas. Sin embargo, puede simplificarse mucho más empleando *async/await*.

4.2.7.3. Llamadas con *async/await*

La especificación *async/await* permite llamar de forma síncrona a una serie de métodos asíncronos, y esperar a que finalicen para pasar a la siguiente tarea. El único requisito para poder hacer esto es que estas llamadas deben hacerse desde dentro de una función que sea asíncrona, declarada con la palabra reservada `async` .

Para hacer el ejemplo anterior, debemos declarar una función asíncrona con el nombre que queramos (por ejemplo, `actualizarEdad`), y dentro llamar a cada función asíncrona precedida de la palabra `await` . Si la llamada va a devolver un resultado (en este caso, el resultado de la promesa), se puede asignar a una constante o variable. Con esto, el código lo podemos reescribir así, y simplemente llamar a la función `actualizarEdad` cuando queramos ejecutarlo:

```
async function actualizarEdad() {
    let contacto = await Contacto.findById('35893...');
    contacto.edad++;
    let contactoGuardado = await contacto.save();
    console.log(contactoGuardado);
}

actualizarEdad();
```

Nos faltaría tratar el apartado de los errores: en los dos casos anteriores existía una cláusula `catch` o un parámetro `error` que consultar y mostrar el mensaje de error correspondiente. ¿Cómo lo gestionamos con *async/await*?. Al utilizar `await` , estamos convirtiendo un código asíncrono en otro síncrono, y por tanto, la gestión de errores es una simple gestión de excepciones con `try..catch` :

```
async function actualizarEdad() {
    try {
        let contacto = await Contacto.findById('35893...');
        contacto.edad++;
        let contactoGuardado = await contacto.save();
        console.log(contactoGuardado);
    } catch (error) {
        console.log("Error:", error);
    }
}

actualizarEdad();
```

4.2.7.4. ¿Cuál elegir?

Se puede emplear en cualquier situación cualquiera de estas tres opciones, según convenga. En este curso utilizaremos el tratamiento mediante promesas para tareas simples, ya que permiten separar de forma clara el código de ejecución correcto del incorrecto, y emplearemos la especificación *async/await* para tareas complejas o enlazadas, donde un método dependa del resultado del anterior para iniciarse.

En este punto, puedes realizar el Ejercicio 1 de los propuestos al final de la sesión.

4.3. Relaciones entre colecciones

Vamos a volver a nuestra base de datos de contactos que venimos utilizando en esta sesión. 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 "*PruebaContactosMongo*" 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, deja renombrada la carpeta anterior como *PruebaContactosMongo_v1*, y crea una copia llamada *PruebaContactosMongo_v2* para lo que haremos a continuación.

4.3.1. 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).

4.3.2. 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).

4.3.3. Inserciones de elementos relacionados

Si quisiéramos insertar un nuevo contacto y, a la vez, especificar su restaurante favorito y/o sus mascotas, deberíamos hacerlo por partes, como ocurriría en un sistema relacional:

- 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(...
```

- 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*.

4.3.4. 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.

En este punto, puedes realizar el Ejercicio 2 de los propuestos al final de la sesión.

4.4. 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 *PruebaContactosMongo_v2* que hemos venido completando hasta ahora, y llama a la nueva copia *PruebaContactosMongo_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://localhost:27017/contactos_subdocumentos',
    {useNewUrlParser: true, useUnifiedTopology: true});
```

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 modelo de restaurante
});

// Mascotas
let mascotaSchema = new mongoose.Schema({
    ... // Código del modelo 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.

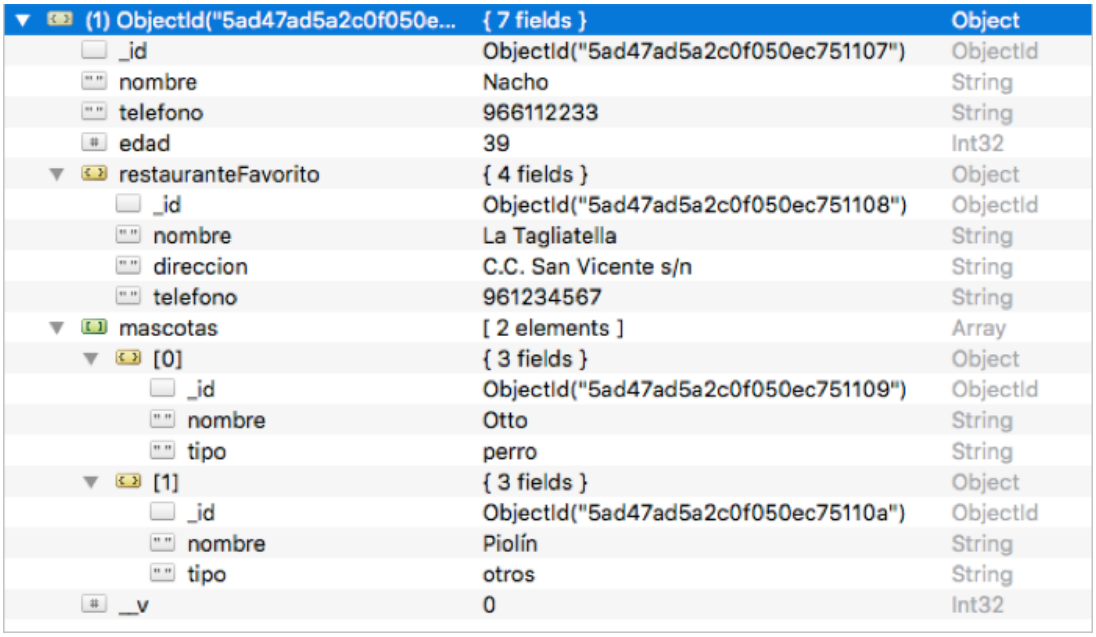
4.4.1. Inserción de documentos con subdocumentos

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:



(1) ObjectId("5ad47ad5a2c0f050e...	{ 7 fields }	Object
_id	ObjectId("5ad47ad5a2c0f050ec751107")	ObjectId
nombre	Nacho	String
telefono	966112233	String
edad	39	Int32
restauranteFavorito	{ 4 fields }	Object
_id	ObjectId("5ad47ad5a2c0f050ec751108")	ObjectId
nombre	La Tagliatella	String
direccion	C.C. San Vicente s/n	String
telefono	961234567	String
mascotas	[2 elements]	Array
[0]	{ 3 fields }	Object
_id	ObjectId("5ad47ad5a2c0f050ec751109")	ObjectId
nombre	Otto	String
tipo	perro	String
[1]	{ 3 fields }	Object
_id	ObjectId("5ad47ad5a2c0f050ec75110a")	ObjectId
nombre	Piolín	String
tipo	otros	String
__v	0	Int32

4.4.2. ¿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.

En este punto, puedes realizar el Ejercicio 3 de los propuestos al final de la sesión.

4.5. 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 *PruebaContactosMongo_v2*.

4.5.1. 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 "*PruebaContactosMongo*"), 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')
...
```

4.5.2. 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.

En este punto, puedes realizar el Ejercicio 4 de los propuestos al final de la sesión. Es de carácter opcional, no es necesario realizarlo para superar el curso.

4.6. Ejercicios propuestos

Para los ejercicios de esta sesión, crea una subcarpeta llamada "**Sesion4**" en tu carpeta "**ProyectosNode/Ejercicios**", para dentro ir creando un proyecto para cada ejercicio.

Ejercicio 1

Crea una carpeta llamada "**Ejercicio_4**" dentro de "*ProyectosNode/Ejercicios/Sesion4*". Aquí iremos desarrollando los ejercicios que se proponen en esta sesión. Verás que se trata de un ejercicio incremental, donde poco a poco iremos añadiendo código sobre un mismo proyecto.

Para empezar, instala Mongoose en dicho proyecto, y crea un archivo `index.js` que conecte con una base de datos llamada "libros", en el servidor Mongo local. Recuerda que, aunque la base de datos aún no exista, no es problema para establecer una conexión, hasta que se añadan colecciones y documentos a ella.

A continuación, vamos a definir un esquema para almacenar la información que nos interese de los libros. En concreto, almacenaremos su título, editorial y precio en euros. El título y el precio son obligatorios, el título debe tener una longitud mínima de 3 caracteres, y el precio debe ser positivo (mayor o igual que 0). Define estas reglas de validación en el esquema, y asócialo a un modelo llamado "libro" (con lo que se creará posteriormente la colección "libros" en la base de datos). Define todo este código en el archivo `models/libro.js`, que deberás crear.

Desde el archivo principal `index.js`, incorpora con `require` el modelo de libros creado, y, después de conectar con la base de datos, realiza estas operaciones:

- Crea un par de libros con estos datos. Muestra por pantalla con `console.log` el resultado de las inserciones y, si algo falla, muestra el error completo:
    - Libro 1:
        - Título: "El capitán Alatriste"
        - Editorial: "Alfaguara"
        - Precio: 15 euros
    - Libro 2:
        - Título: "El juego de Ender"
        - Editorial: "Ediciones B"
        - Precio: 8.95 euros
- Utiliza el método genérico `find` para buscar los libros cuyo precio oscile entre los 10 y los 20 euros (inclusive)
- Utiliza `findById` para mostrar la información del libro que quieras (averigua el id de alguno de los libros y saca su información).
- Borra uno de los libros de la colección empleando el método `findByIdAndRemove`. Muestra por pantalla los datos del libro borrado, cuando todo haya ido correctamente.
- Modifica el precio de otro de los libros de la colección. Muestra por pantalla los datos del nuevo libro modificado, una vez se haya completado la operación. Opcionalmente, prueba también a incrementar su versión (campo `__v`) en una unidad.

> **NOTA**: Puedes dejar comentado el código que no vayas a probar de los pasos anteriores, para centrarte sólo en el del paso en cuestión. También ten en cuenta que, si haces las inserciones más de una vez, se

> añadirán los libros nuevamente a la colección, ya que no hemos puesto ninguna regla de validación para eliminar duplicados. No es problema. Siempre puedes eliminar los duplicados a mano desde Robo 3T.

Ejercicio 2

Vamos ahora a definir 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. Todo este código deberá ir en el archivo `models/autor.js`.

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.

En el archivo principal `index.js`, incorpora con `require` el modelo de autores, además del de libros que ya teníamos previamente, y añade el código para insertar uno o dos nuevos autores, y algún libro vinculado a cada uno de ellos.

Ejercicio 3

Sobre el ejercicio anterior, edita el archivo `models/libro.js` y define un nuevo esquema 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`). 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, desde el programa principal `index.js` crea un nuevo libro con sus datos, y añade a mano un par de comentarios al array, antes de guardar todos los datos.

Ejercicio 4

Opcional

Añade al ejercicio anterior estas dos consultas:

- 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.
- 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). Antes de realizar esta consulta, procura que haya al menos dos o tres autores en la colección de autores, y al menos tres o cuatro libros con autores diferentes.