Uso básico de la librería Mongoose

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.

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, de forma opcional, un segundo parámetro con un objeto con propiedades de conexión. Este segundo objeto es necesario incluirlo en ciertas versiones de Mongoose para especificar algunas opciones adicionales, y de hecho veremos un *warning* al ejecutar la aplicación advirtiéndonos, pero en términos generales podemos conectar directamente con la URL de conexión como primer y único parámetro:

```
mongoose.connect('mongodb://localhost:27017/contactos');
```

Como decimos, es posible que en algunas versiones nos "obligue" a especificar algunos parámetros extra de conexión, como por ejemplo:

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

Los parámetros que haya que incluir en este caso ya nos lo dirá el propio *warning* que se emita por consola al conectar, o la propia documentació oficial de Mongoose.

En cuanto a la conexión en sí, no os preocupéis por que la base de datos no exista. Se creará automáticamente tan pronto como añadamos datos en ella.

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).

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`

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.

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');

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

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);
});
```

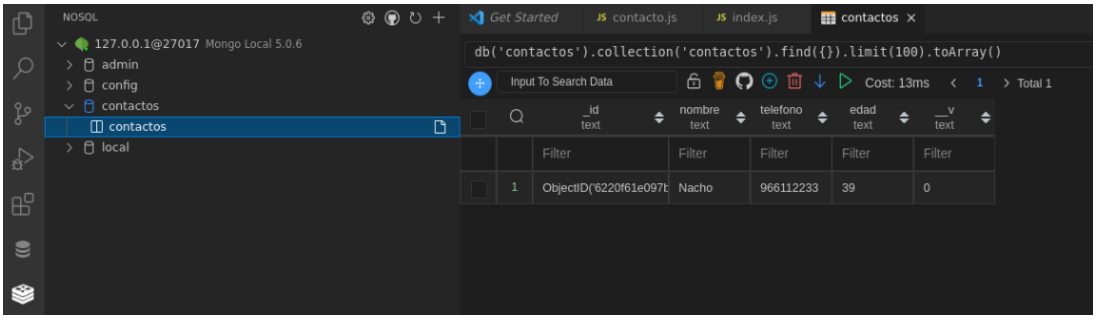
Añade este código al archivo `index.js` de nuestro proyecto "*PruebaContactosMongo*", tras la conexión a la base de datos. Ejecuta la aplicación, y echa un vistazo al resultado que se devuelve cuando todo funciona correctamente. Será algo parecido a esto:

```
{
  nombre: 'Nacho',
  telefono: '966112233',
  edad: 39,
  _id: new ObjectId("5a12a2e0e6219d68c00c6a00"),
  __v: 0
}
```

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 al plugin de Visual Studio Code 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 mensaje *ValidationError* con la información del error:

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

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", edad: 70,
    telefono:"611885599"});
```

4. Buscar documentos

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

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. 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(...
```

4.3. 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.

5. Borrar documentos

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

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:
  ...
```

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).

6. Modificaciones o actualizaciones de documentos

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

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*).

6.2. Los métodos *updateOne* y *updateMany*

Estos métodos se puede utilizar para actualizar los datos del primer documento que encaje con la condición de búsqueda, o con todos los que encajen con dicha condición, respectivamente. Por ejemplo, la siguiente instrucción pone a 20 los años del primer contacto que encuentre con nombre "Nacho":

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

Esta otra alternativa actualiza los datos de todos los contactos con ese nombre:

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

Ambos métodos son útiles si queremos buscar documentos por campos que no sean su identificador principal. En caso contrario, es preferible usar `findByIdAndUpdate`.

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(...
```

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.

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.

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*.

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();
```

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.