Introducció a Mongoose


Existeixen diverses llibreries en el repositori oficial de NPM per a gestionar bases de dades MongoDB, però la més popular és Mongoose. Permet accedir de manera fàcil a les bases de dades i, a més, definir esquemes, una estructura de validació que determina el tipus de dada i rang de valors adequat per a cada camp o propietat dels documents d'una col·lecció. Així, podem establir si un camp és obligatori o no, si ha de tindre un valor mínim o màxim, etc. En la web oficial de Mongoose podem consultar alguns exemples de definició d'esquemes i documentació addicional.

1. Càrrega de la llibreria i connexió al servidor

Al llarg d'aquesta sessió farem algunes proves amb Mongoose en un projecte que anomenarem "*ContactesMongo*". Podem crear-ho ja en la nostra carpeta "*ProjectesNode/Proves*". Després, definirem l'arxiu `package.json` amb la comanda `npm init`, i posteriorment instal·larem mongoose en el projecte amb la comanda `npm install`.

```
npm install mongoose
```

Una vegada instal·lat, necessitem incorporar-ho al codi del projecte amb la corresponent instrucció `require`. Crea un arxiu font `index.js` en aquest projecte de proves, i incorpora la llibreria d'aquesta manera::

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

Per a connectar amb el servidor Mongo (suposant que ja el tenim iniciat) necessitem cridar a un mètode anomenat `connect`, dins de l'objecte `mongoose` que hem incorporat. Li passarem la URL de la base de dades com a primer paràmetre i, de manera opcional, un segon paràmetre amb un objecte amb propietats de connexió. Aquest segon objecte és necessari incloure'l en unes certes versions de Mongoose per a especificar algunes opcions addicionals, i de fet veurem un *warning* en executar l'aplicació advertint-nos, però en termes generals podem connectar directament amb la URL de connexió com a primer i únic paràmetre.

En el cas d'accedir a un servidor MongoDB local, podem usar una URL com aquesta:

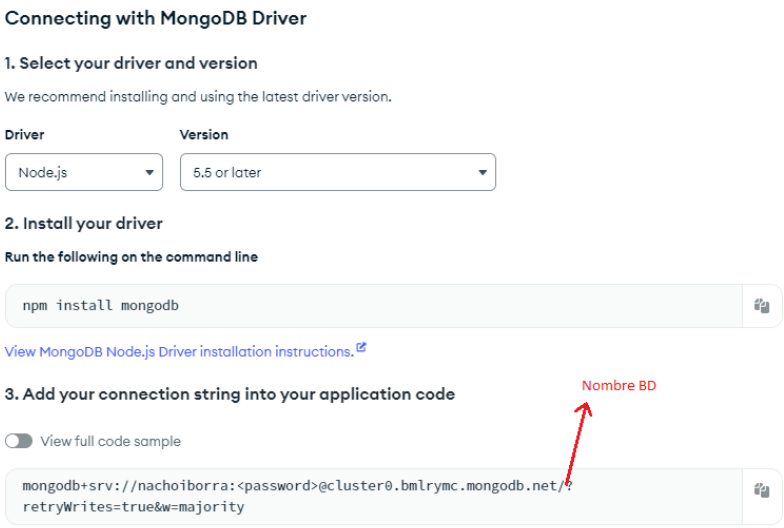
```
mongoose.connect('mongodb://127.0.0.1:27017/contactes');
```

> **NOTA**: en algunes versions Mongoose també accepta la connexió a *localhost* com a nom de servidor local, però en altres ho ha restringit a l'adreça IP local *127.0.0.1*.

Si connectem a un servidor en el núvol de *MongoDB Atlas*, podem emprar una URL amb aquest format (canviant-la per la qual ens indique el nostre compte en *Atlas*):

```
mongoose.connect('mongodb+srv://usuari:password@url_cluster/nomBD?parametres');
```

Això el podem consultar en la nostra secció de *Atlas*, prement en el botó *Connect* del nostre *clúster*, i veient com connectar amb el driver de Mongo (recorda afegir el nom de la base de dades al final de la URL, just abans de l'interrogant `?` per als paràmetres):



Com diem, és possible que en algunes versions ens "obligue" a especificar alguns paràmetres extra de connexió, com per exemple:

```
mongoose.connect('mongodb://127.0.0.1:27017/contactes',
 { useNewUrlParser: true, useUnifiedTopology: true });
```

Els paràmetres que calga incloure en aquest cas ja ens ho dirà el propi *warning* que s'emeta per consola en connectar, o la pròpia documentació oficial de Mongoose.

Quant a la connexió en si, no us preocupeu perquè la base de dades no existisca. Es crearà automàticament tan prompte com afegim dades en ella.

2. Models i esquemes

Com comentàvem abans, la llibreria Mongoose permet definir l'estructura que tindran els documents de les diferents col·leccions de la base de dades. Per a això, es defineixen esquemes (*schemas*) i s'associen a models

(les col·leccions corresponents en la base de dades).

2.1. Definir els esquemes

Per a definir un esquema, necessitem crear una instància de la classe `Schema` de Mongoose. Per tant, crearem aquest objecte, i en aqueixa creació definirem els atributs que tindrà la col·lecció corresponent, juntament amb el tipus de dada de cada atribut. És també recomanable separar aquestes definicions en arxius a part. Podem crear una subcarpeta `models` i emmagatzemar en ella els esquemes i models de la nostra base de dades.

En el cas de la base de dades de contactes proposada per a aquestes proves, podem definir un esquema per a emmagatzemar les dades de cada contacte: nom, número de telèfon i edat, per exemple. Això ho faríem d'aquesta manera, en un arxiu anomenat `contacte.js` dins de la subcarpeta `models` del nostre projecte:

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

let contacteSchema = new mongoose.Schema({
    nom: String,
    telefon: String,
    edat: Number
});
```

Els tipus de dades disponibles per a definir l'esquema són:

- Textos (`String`)
- Números (`Number`)
- Dates (`Date`)
- Booleans (`Boolean`)
- Arrays (`Array`)
- Uns altres (veurem alguns més endavant): `Buffer` , `Mixed` , `ObjectId`

2.2. Aplicar l'esquema a un model

Una vegada definit l'esquema, necessitem aplicar-lo a un model per a associar-lo així a una col·lecció en la base de dades. Per a això, disposem del mètode `model` en Mongoose. Com a primer paràmetre, indicarem el nom de la col·lecció a la qual associar l'esquema. Com a segon paràmetre, indicarem l'esquema a aplicar (objecte de tipus `Schema` creat anteriorment). Afegiríem aquestes línies al final del nostre arxiu `models/contacte.js` :

```
let Contacte = mongoose.model('contactes', contacteSchema);
module.exports = Contacte;
```

> **NOTA:** si indiquem un nom de model en singular, Mongoose automàticament crearà la col·lecció amb el nom en plural. Aquest plural no sempre serà correcte, ja que el que fa és simplement afegir una "s" al final del nom del model, si no li l'hem afegida nosaltres. Per aquest motiu, és recomanable que creem els models amb noms de col·leccions ja en plural.

2.3. Restriccions i validacions

Si definim un esquema senzill com l'exemple de contactes anterior, permetrem que s'afija qualsevol tipus de valor als camps dels documents. Així, per exemple, podríem tindre contactes sense nom, o amb edats negatives. Però amb Mongoose podem proporcionar mecanismes de validació que permeten descartar de manera automàtica els documents que no complisquen les especificacions.

En la documentació oficial de Mongoose podem trobar una descripció detallada dels diferents validadores que podem aplicar. Ací ens limitarem a descriure els més importants o habituals:

- El validador `required` permet definir que un determinat camp és obligatori.
- El validador `default` permet especificar un valor per defecte per al camp, en el cas que no s'especifique cap.
- Els validadores `min` i `max` s'utilitzen per a definir un rang de valors (mínim i/o màxim) permesos per a dades de tipus numèric (ambdós límits inclosos).
- Els validadores `minlength` i `maxlength` s'empren per a definir una grandària mínima o màxima de caràcters, en el cas de cadenes de text.
- El validador `unique` indica que el camp en qüestió no admet duplicats (seria una clau alternativa, en un sistema relacional). En la documentació de Mongoose s'especifica que això no és pròpiament un validador, sinó una ajuda per a indexar, i que depenent de quan s'indexe, és possible que no funcione adequadament.
- El validador `match` s'empra per a especificar una expressió regular que ha de complir el camp (ací teniu més informació sobre aquest tema).
- ...

Tornem al nostre esquema de contactes. Establirem que el nom i el telèfon siguen obligatoris, i només permetrem edats entre 18 i 120 anys (inclusivament). A més, el nom tindrà una longitud mínima d'1 caràcter, i el telèfon estarà compost per 9 dígits, emprant una expressió regular, i serà una clau única. Podem emprar algun validador més, com per exemple `trim`, per a netejar els espais en blanc a l'inici i final de les dades de text. Amb totes aquestes restriccions, l'esquema i model associat queden d'aquesta manera:

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

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

let Contacte = mongoose.model('contactes', contacteSchema);
module.exports = Contacte;
```

Ja tenim establida la connexió a la base de dades, i l'esquema de les dades que utilitzarem. Ara, podem afegir el model al nostre arxiu principal `index.js`, i ja podrem començar a realitzar algunes operacions bàsiques contra aquesta base de dades.

```
const mongoose = require('mongoose');
const Contacte = require(__dirname + "/models/contacte");

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

// Ací ja podrem realitzar operacions contra la BD
```

> **Exercici 1:**
>
> Crea una carpeta anomenada "**LlibresMongo**" en el teu espai de treball, en la carpeta "*Exercicis*". Ací anirem desenvolupant els exercicis que es proposen en aquesta sessió. Veuràs que es tracta d'un exercici incremental, on a poc a poc anirem afegint codi sobre un mateix projecte.
>
> Per a començar, instal·la Mongoose en aquest projecte seguint els passos indicats anteriorment, i crea un arxiu `index.js` que connecte amb una base de dades anomenada "llibres", en el servidor Mongo.

> Recorda que, encara que la base de dades encara no existisca, no és problema per a establir una connexió, fins que s'afigen col·leccions i documents a ella.
>
> A continuació, definirem un esquema per a emmagatzemar la informació que ens interesse dels llibres, a l'arxiu `models/llibre.js`. En concret, emmagatzemarem el seu títol, editorial i preu en euros. El títol i el preu són obligatoris, el títol ha de tindre una longitud mínima de 3 caràcters, i el preu ha de ser positiu (major o igual que 0). Defineix aquestes regles de validació en l'esquema, i associa'l a un model anomenat "llibre" (amb el que es crearà posteriorment la col·lecció "llibres" en la base de dades).

3. Afegir documents

Si volem inserir un document en una col·lecció hem de crear un objecte del corresponent model, i cridar al seu mètode `save`. Aquest mètode retorna una promesa, per la qual cosa emprarem:

- Un bloc de codi `then` per a quan l'operació haja anat correctament. En aquest bloc rebrem com a resultat l'objecte que s'ha inserit, podent examinar les dades del mateix si es vol.
- Un bloc de codi `catch` per a quan l'operació no haja pogut completar-se. Rebrem com a paràmetre un objecte amb l'error produït, que podrem examinar per a obtindre més informació sobre aquest.

Aquest mateix patró *then-catch* l'emprarem també amb la resta d'operacions més endavant (cerques, esborrats o modificacions), encara que el resultat retornat en cada cas variarà.

Així afegiríem un nou contacte a la nostra col·lecció de proves:

```
let contacte1 = new Contacte({
    nom: "Nacho",
    telefon: "966112233",
    edat: 45
});
contacte1.save().then(resultat => {
    console.log("Contacte afegit:", resultat);
}).catch(error => {
    console.log("ERROR afegint contacte:", error);
});
```

Afig aquest codi a l'arxiu `index.js` del nostre projecte "*ContactesMongo*", després de la connexió a la base de dades. Executa l'aplicació, i dona una ullada al resultat que es retorna quan tot funciona correctament. Serà alguna cosa semblança a això:

```
{
    nom: 'Nacho',
    telefono: '966112233',
    edat: 45,
    _id: new ObjectId("5a12a2e0e6219d68c00c6a00"),
    __v: 0
}
```

Observa que obtenim els mateixos camps que definim en l'esquema (nom, telèfon i edat), i dos camps addicionals que no hem especificat:

- `__v` fa referència a la versió del document. Inicialment, tots els documents parteixen de la versió 0 quan s'insereixen, i després aquesta versió pot modificar-se quan fem alguna actualització del document, com veurem després.
- `_id_` és un codi autogenerat per Mongo, per a qualsevol document de qualsevol col·lecció que es tinga. Analitzarem en què consisteix este *id* i la seua utilitat en breu.

Anem ara al plugin de Visual Studio Code i examinem les bases de dades en el panell esquerre. Si cliquem en la col·lecció de "contactes", veurem el nou contacte afegit en el panell dret:

Exercici 2:

Continuem amb l'exercici "**LlibresMongo**" creat anteriorment. Farem un parell d'insercions sobre la base de dades i model creats en l'exercici anterior. Sota el codi que ja hauràs de tindre implementat (connectar amb la base de dades i definir el model), fes el següent:

- Crea un llibre amb aquestes dades:
  - Títol: "El capità Alatriste"
  - Editorial: "Alfaguara"
  - Preu: 15 euros
- Crea un altre llibre amb aquestes altres dades:
  - Títol: "El joc d'Ender"
  - Editorial: "Edicions B"
  - Preu: 8.95 euros

Inserta els dos llibres en la base de dades. Hauran d'aparéixer en la col·lecció "llibres". En inserir, mostra per pantalla amb `console.log` el resultat de la inserció, i si alguna cosa falla, mostra l'error complet.

> **NOTA:** si executes l'aplicació més d'una vegada, s'afegiran els llibres novament a la col·lecció, ja que no hem posat cap regla de validació per a eliminar duplicats. No és problema. Sempre pots eliminar els duplicats a mà des del *plugin* de VS Code o l'eina de gestió que estigues utilitzant.

Si intentem inserir un contacte incorrecte, saltarem al bloc `catch`. Per exemple, aquest contacte és massa vell, segons la definició de l'esquema:

```
let contacte2 = new Contacte({
    nom: "Matuzalem",
    telefon: "965123456",
    edat: 200
});
contacte2.save().then(resultat => {
    console.log("Contacte afegit:", resultat);
}).catch(error => {
    console.log("ERROR afegint contacte:", error);
});
```

Si donem una ullada a l'error produït, veurem molta informació, però entre tota aqueixa informació hi ha un missatge *ValidatorError* amb la informació de l'error:

```
ValidatorError: Path `edat` (200) is more than maximum allowed value (120)
```

3.1. Sobre el *id* automàtic

Com has pogut veure en les proves d'inserció anteriors, cada vegada que s'afig un document a una col·lecció se li assigna automàticament una propietat anomenada `_id_` amb un codi autogenerat. A diferència d'altres sistemes de gestió de bases de dades (com MariaDB/MySQL, per exemple), aquest codi no és autonumèric, sinó que és una cadena. De fet, és un text de 12 bytes que emmagatzema informació important:

- El temps de creació de document (*timestamp*), amb el que podem obtindre el moment exacte (data i hora) d'aquesta creació
- L'ordinador que va crear el document. Això és particularment útil quan volem escalar l'aplicació i tenim diferents servidors Mongo accedint a la mateixa base de dades. Podem identificar quin de tots els servidors va ser el que va crear el document.
- El procés concret del sistema que va crear el document
- Un comptador aleatori, que s'empra per a evitar qualsevol tipus de duplicitat, en el cas que els tres valors anteriors coincidisquen en el temps.

Existeixen mètodes específics per a extraure part d'aquesta informació, en concret el moment de creació, però no els utilitzarem en aquest curs.

Malgrat disposar d'aquest enorme avantatge amb aquest *id* autogenerat, podem optar per crear els nostres propis ids i no utilitzar els de Mongo (encara que aquesta no és una bona idea):

```
let contacteX = new Contacte({_id:2, nom:"Juan", edat: 70,
 telefon:"611885599"});
```

4. Buscar documents

Si volem buscar qualsevol document, o conjunt de documents, en una col·lecció, podem emprar diversos mètodes.

4.1. Cerca genèrica amb *find*

La forma més general d'obtindre documents consisteix a emprar el mètode estàtic `find` associat al model en qüestió. Podem emprar-ho sense paràmetres (amb el que obtindrem tots els documents de la col·lecció com a resultat de la promesa):

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

4.2. Cerca parametritzada amb *find*

Podem també passar com a paràmetre a `find` un conjunt de criteris de cerca. Per exemple, per a buscar contactes el nom dels quals siga "Nacho" i l'edat siga de 29 anys, faríem això:

```
Contacte.find({nom: 'Nacho', edat: 29}).then(resultat => {
    console.log(resultat);
}).catch (error => {
    console.log("ERROR:", error);
});
```

> **NOTA:** qualsevol cridada a `find` retornarà un **array de resultats**, encara que només s'haja trobat un, o cap. És important tindre-ho en compte per a després saber com accedir a un element concret d'aquest resultat. El fet de no obtindre resultats no provocarà un error (no se saltarà al `catch` en aqueix cas).

També podem emprar alguns operadors de comparació en el cas de no buscar dades exactes. Per exemple, aquesta consulta obté tots els contactes el nom dels quals siga "Nacho" i les edats estiguen compreses entre

18 i 40 anys:

```
Contacte.find({nom:'Nacho', edat: {$gte: 18, $lte: 40}})
.then(resultat => {
    console.log('Resultat de la cerca:', resultat);
})
.catch(error => {
    console.log('ERROR:', error);
});
```

Ací podeu trobar un llistat detallat dels operadors que podeu utilitzar en les cerques.

A més, la cerca parametritzada amb `find` admet altres variants de sintaxis, com l'ús de mètodes enllaçats `where`, `limit`, `sort` ... fins a obtindre els resultats desitjats en l'ordre i quantitat desitjada. Per exemple, aquesta consulta mostra els 10 primers contactes majors d'edat, ordenats de major a menor edat:

```
Contacte.find()
.where('edat')
.gte(18)
.sort('-edat')
.limit(10)
.then(...
```

4.3. Altres opcions: *findOne* o *findById*

Existeixen altres alternatives que podem utilitzar per a buscar documents concrets (i no un conjunt o llista d'ells). Es tracta dels mètodes `findOne` i `findById`. El primer s'empra de manera similar a `find`, amb els mateixos paràmetres de filtrat, però només retorna un document (arbitrari) que concorde amb aqueixos criteris (no un array). Per exemple:

```
Contacte.findOne({nom:'Nacho', edat: 39})
.then(resultat => {
    console.log('Resultat de la cerca:', resultat);
})
.catch(error => {
    console.log('ERROR:', error);
});
```

El mètode `findById` s'empra, com el seu nom indica, per a buscar un document donat el seu *id* (recordem, aqueixa seqüència de 12 bytes autogenerada per Mongo). Per exemple:

```
Contacte.findById('5ab2dfb06cf5de1d626d5c09')
.then(resultat => {
    console.log('Resultat de la cerca per ID:', resultat);
})
.catch(error => {
    console.log('ERROR:', error);
});
```

En aquests mètodes, si la consulta no produeix cap resultat, obtindrem `null` com a resposta, però tampoc s'activarà la clàusula `catch` per això.

> **Exercici 3:**
>
> Continuem amb l'exercici "**LlibresMongo**" creat anteriorment. Sobre els llibres que hem inserit prèviament, mostrarem dues cerques:
>
> - En primer lloc, utilitza el mètode genèric `find` per a buscar els llibres el preu dels quals oscil·le entre els 10 i els 20 euros (inclusivament)
> - A continuació, utilitza `findById` per a mostrar la informació del llibre que vulgues (esbrina l'*id* d'algun dels llibres i saca la seua informació).
>
> **NOTA**: Pots deixar comentat el codi que fa les insercions de l'exercici anterior, perquè no estiga contínuament inserint nous llibres cada vegada que faces les consultes.

5. Esborrar documents

Per a eliminar documents d'una col·lecció, podem emprar diversos mètodes estàtics, depenent del que vulguem eliminar i les condicions per eliminar-lo.

5.1. Els mètodes *deleteOne* i *deleteMany*

Estos mètodes eliminen els documents que complisquen una determinada condició de filtrat. El primer d'ells elimina el primer document que es trobe, i el segon tots els que complisquen la condició.

```
// Eliminem tots els contactes que es criden Nacho
Contacto.deleteMany({nom: 'Nacho'}).then(resultat => {
    console.log(resultat.deletedCount, "documents esborrats");
}).catch (error => {
    console.log("ERROR:", error);
});
```

Com pot veure's en l'exemple, tots dos mètodes retornen un objecte amb una propietat anomenada `deletedCount` que mostra quants documents s'han esborrat.

> **NOTA**: si usem `deleteMany` sense especificar cap condició, s'eliminaran TOTS els documents de la col·lecció afectada.

5.2. Els mètodes *findOneAndDelete* i *findByIdAndDelete*

El mètode `findOneAndDelete` cerca el document que complisca el patró especificat (o el primer que trobe que el complisca) i l'elimina. A més, obté el document eliminat en el resultat, amb el que podríem desfer l'operació a posteriori, si volguérem, tornant-lo a afegir.

```
Contacte.findOneAndDelete({nom: 'Nacho'})
.then(resultat => {
    console.log("Contacte eliminat:", resultat);
}).catch (error => {
    console.log("ERROR:", error);
});
```

En aquest cas el paràmetre `resultat` és directament l'objecte eliminat. Per la seua part el mètode `findByIdAndDelete` cerca el document amb l'*id* indicat i l'elimina. També obté com a resultat l'objecte eliminat.

```
Contacte.findByIdAndDelete('5a16fed09ed79f03e490a648')
.then(resultat => {
    console.log("Contacte eliminat:", resultat);
}).catch (error => {
    console.log("ERROR:", error);
});
```

En el cas d'aquests dos últims mètodes, si no s'ha trobat cap element que complisca el criteri de filtrat, es retornarà `null` com a resultat, és a dir, no s'activarà la clàusula `catch` per aquest motiu. Sí que s'activaria aquesta clàusula, per exemple, si indiquem un *id* amb un format no vàlid (que no tinga 12 bytes).

> **NOTA**: en versions anteriors de Mongoose aquestos mètodes s'anomenaven `findOneAndRemove` i `findByIdAndRemove`, respectivament, y van ser reemplaçats per aquests altres en versions més recents.

6. Modificacions o actualitzacions de documents

Per a realitzar modificacions d'un document en una col·lecció, també podem emprar diferents mètodes estàtics.

6.1. El mètode *findByIdAndUpdate*

El mètode `findByIdAndUpdate` buscarà el document amb l'*id* indicat, i reemplaçarà els camps atesos els criteris que indiquem com a segon paràmetre.

En aquest enllaç podeu consultar els operadors d'actualització que podem emprar en el segon paràmetre d'anomenada a aquest mètode. El més habitual de tots és `$set`, que rep un objecte amb els parells clau-valor que volem modificar en el document original. Per exemple, així reemplacem el nom i l'edat d'un contacte amb un determinat *id*, deixant el telèfon sense modificar:

```
Contacte.findByIdAndUpdate('5a0e1991075e9407c4da8b0a',
 {$set: {nom:'Nacho Iborra', edat: 40}}, {new:true})
.then(resultat => {
    console.log("Modificat contacte:", resultat);
}).catch (error => {
    console.log("ERROR:", error);
});
```

Com a alternativa al codi anterior, podem ometre la paraula reservada `$set:` de manera que el codi se'ns quedaria:

```
Contacte.findByIdAndUpdate('5a0e1991075e9407c4da8b0a',
 { nom: 'Nacho Iborra', edat: 40 } , { new: true })
.then(resultat => {
    console.log("Modificat contacte:", resultat);
}).catch (error => {
    console.log("ERROR:", error);
});
```

El tercer paràmetre que rep `findByIdAndUpdate` és un conjunt d'opcions addicionals. Per exemple, l'opció `new` que s'ha usat en aquest exemple indica si volem obtindre com a resultat el nou objecte modificat (`true`) o l'antic abans de modificar-se (`false`, una cosa útil per a operacions de desfer). També podem passar-li com a tercer paràmetre l'opció `runValidators`, que ens permet validar els camps que modifiquem, per defecte aquesta opció està desactivada i per a poder validar els camps hem de donar-li el valor a `true`.

En intentar modificar un contacte amb una edat incorrecta, salta al bloc `catch`. Per exemple, aquest contacte és massa jove, segons la definició de l'esquema:

```
Contacte.findByIdAndUpdate('5a0e1991075e9407c4da8b0a',
 { nom: 'Juan Pérez', edat: 10 } , { new: true, runValidators: true })
.then(resultat => {
    console.log("Modificat contacte:", resultat);
}).catch (error => {
    console.log("ERROR:", error);
});
```

6.2. Els mètodes *updateOne* i *updateMany*

Aquests mètodes es poden utilitzar per a actualitzar les dades del primer document que encaixe amb la condició de cerca, o amb tots els que encaixen amb aquesta condició, respectivament. Per exemple, la següent instrucció posa a 20 els anys del primer contacte que trobe amb nom "Nacho":

```
Contacte.updateOne({nom: 'Nacho', {$set: { edat: 20 }})
.then(...
```

Aquesta altra alternativa actualitza les dades de tots els contactes amb aqueix nom:

```
Contacte.updateMany({nom: 'Nacho', {$set: { edat: 20 }})
.then(...
```

Tots dos mètodes són útils si volem buscar documents per camps que no siguen el seu identificador principal. En cas contrari, és preferible usar `findByIdAndUpdate`.

6.3. Actualitzar la versió del document

Hem vist que, entre els atributs d'un document, a més de l'*id* autogenerat per Mongo, es crea un número de versió en un atribut `__v`. Aquest número de versió al·ludeix a la versió del document en si, de manera que, si posteriorment es modifica (per exemple, amb una anomenada a `findByIdAndUpdate`), es puga també indicar amb un canvi de versió que aqueix document ha patit canvis des de la seua versió original. Si volguérem fer això amb l'exemple anterior, bastaria amb afegir l'operador `$inc` (al costat del `$set` utilitzat abans) per a indicar que incremente el número de versió, per exemple, en una unitat:

```
Contacte.findByIdAndUpdate('5a0e1991075e9407c4da8b0a',
 {$set: {nom:'Nacho Iborra', edat: 40},
 $inc: {__v: 1}}, {new:true})
.then(...
```

> **Exercici 4:**
>
> Realitzarem finalment algunes operacions d'esborrat i modificació sobre la nostra col·lecció de llibres, en el mateix projecte "**LlibresMongo**" d'exercicis anteriors.
>
> - Localitza un dels llibres que hauries de tindre inserits d'exercicis anteriors. Queda't amb el seu id, i esborra-ho de la col·lecció emprant el mètode `findByIdAndRemove`. Mostra per pantalla les dades del llibre esborrat, quan tot haja anat correctament.
> - Localitza un altre dels llibres que hages inserit, queda't amb el seu id i modifica el seu preu al valor que vulgues. Mostra per pantalla les dades del nou llibre modificat, una vegada s'haja completat l'operació. Opcionalment, prova també a incrementar la seua versió (camp `__v`) en una unitat.

7. Mongoose i les promeses

Hem indicat anteriorment que operacions com `find`, `save` i la resta de mètodes que hem emprat amb Mongoose retornen una promesa, però això no és del tot cert. El que retornen aquests mètodes és un *thenable*, és a dir, un objecte que es pot tractar amb el corresponent mètode `then`. No obstant això, existeixen altres formes alternatives de cridar a aquests mètodes, i podem emprar l'una o l'altra segons ens convinga.

7.1. Cridades com a simples funcions asíncrones

Els mètodes facilitats per Mongoose són simplement tasques asíncrones, és a dir, podem cridar-les i definir un *callback* de resposta que s'executarà quan la tasca finalitze. Per a això, afegim com a paràmetre addicional al mètode el *callback* en qüestió, amb dos paràmetres: l'error que es produirà si el mètode no s'executa satisfactòriament, i el resultat retornat si el mètode s'executa sense contratemps. Aquestos dos paràmetres han d'indicar-se en aquest mateix ordre (primer l'error i després el resultat correcte). Si, per exemple, volem buscar un contacte a partir del seu *id*, podem fer una cosa així:

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

Pensem ara en una cosa més complexa: busquem el contacte pel seu *id*, una vegada finalitzat, incrementem en un any la seua edat i guardem els canvis. En aquest cas, el codi pot quedar així:

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

Com podem veure, en enllaçar una crida asíncrona (`findById`) amb una altra (`save`), el que es produeix és un anidamiento de *callbacks*, amb les seues corresponents estructures `if..else` . Aquest fenomen es coneix com *callback hell* o *pyramid of doom*, perquè produeix en la part esquerra del codi una piràmide girada (el pic de la qual apunta cap a la dreta), que serà més gran quantes més dites enllacem entre si. Dit d'una altra manera, estarem tabulant cada vegada més el codi per a niar crides dins de crides, i aquesta gestió pot fer-se difícil de manejar.

7.2. Cridades com a promeses

Tornem ara al que sabem fer. Com enllaçaríem usant promeses les dues operacions anteriors? Recordem: buscar un contacte pel seu *id* i incrementar-li la seua edat en un any.

Podríem també cometre un *callback hell* niant clàusules `then` , amb una cosa així:

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

No obstant això, les promeses permeten concatenar clàusules `then` sense necessitat de niar-les, deixant un únic bloc `catch` al final per a recollir l'error que es produïsca en qualsevol d'elles. Per a això, n'hi ha prou

que dins d'un `then` es retorne (`return`) el resultat de la següent promesa. El codi anterior podríem reescriure'l així:

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

Aquesta forma és més neta i clara quan volem fer operacions complexes. No obstant això, pot simplificar-se molt més emprant *async/await*.

7.3. Anomenades amb *async/await*

L'especificació *async/await* permet cridar de manera síncrona a una sèrie de mètodes asíncrons, i esperar que finalitzen per a passar a la següent tasca. L'únic requisit per a poder fer això és que aquestes crides han de fer-se des de dins d'una funció que siga asíncrona, declarada amb la paraula reservada `async`.

Per a fer l'exemple anterior, hem de declarar una funció asíncrona amb el nom que vulguem (per exemple, `actualitzarEdat`), i dins cridar a cada funció asíncrona precedida de la paraula `await`. Si la crida retornarà un resultat (en aquest cas, el resultat de la promesa), es pot assignar a una constant o variable. Amb això, el codi ho podem reescriure així, i simplement cridar a la funció `actualitzarEdat` quan vulguem executar-ho:

```
async function actualitzarEdat() {
    let contacte = await Contacte.findById('35893...');
    contacte.edat++;
    let contactoGuardado = await contacte.save();
    console.log(contactoGuardado);
}

actualitzarEdat();
```

Ens faltaria tractar l'apartat dels errors: en els dos casos anteriors existia una clàusula `catch` o un paràmetre `error` que consultar i mostrar el missatge d'error corresponent. Com ho gestionem amb *async/await*?. En utilitzar `await`, estem convertint un codi asíncron en un altre síncron, i per tant, la gestió d'errors és una simple gestió d'excepcions amb `try..catch`:

```
async function actualitzarEdat() {
    try {
        let contacte = await Contacte.findById('35893...');
        contacte.edat++;
        let contacteGuardat = await contacte.save();
        console.log(contacteGuardat);
    } catch (error) {
        console.log("Error:", error);
    }
}

actualitzarEdat();
```

7.3.1 Execució seqüencial i paral·lela amb *async/*await

Fins ara, hem vist exemples de **execució seqüencial** utilitzant `async/await`, com en el cas de la busca d'un contacte pel seu id i l'actualització de la seua edat. En esta mena d'execució, cada operació espera que l'anterior acabe abans de continuar. Este enfocament és útil quan les operacions depenen entre si, i volem assegurar-nos que s'executen en un orde específic.

No obstant això, quan les tasques no depenen entre si i poden executar-se simultàniament, podem optar per una **execució paral·lela**. Això és útil en situacions en les quals les operacions no afecten les altres i necessitem optimitzar el temps d'execució.

Per a aconseguir la **execució paral·lela en async/await**, usem `Promise.all()`, que ens permet executar diverses promeses en paral·lel. `Promise.all()` rep un array de promeses i espera al fet que totes es resolguen abans de continuar.

Exemple: en este cas, realitzem la inserció de dos contactes al mateix temps, ja que les operacions no depenen entre si.

```
async function operacionsParaleles() {
    try {
        // Crear dos instàncies de contacte
        let contacto1 = new Contacto({
            nombre: "Ana",
            telefono: "987654321",
            edad: 25
        });

        let contacto2 = new Contacto({
            nombre: "Luis",
            telefono: "456123789",
            edad: 40
        });

        // Executar totes dues operacions de guardat en paral·lel
        const [resultado1, resultado2] = await Promise.all([
            contacto1.save(),
            contacto2.save()
        ]);

        console.log("Contactes afegits en paral·lel:", resultado1, resultado2);
    } catch (error) {
        console.log("ERROR en operacions paral·leles:", error);
    }
}

operacionsParaleles();
```

7.4. Quin triar?

Cadascun d'estos enfocaments té els seus avantatges:

- **Callbacks:** útils en situacions simples, però poden tornar-se difícils de gestionar amb múltiples operacions.
- **Promeses:** útils per a manejar múltiples operacions de forma encadenada sense niar *callbacks.
- **Async/await:** és l'enfocament més modern i senzill de llegir, especialment per a operacions complexes o seqüencials. A més, permet manejar tant operacions seqüencials com paral·leles.

En este curs utilitzarem promeses per a operacions simples i `async/await` per a operacions més complexes o quan les tasques depenguen les unes de les altres.