Opcions avançades de Mongoose


En aquest document analitzarem algunes operacions una mica més complexes que podem fer amb bases de dades NoSQL, com la possibilitat d'establir connexions entre diferents col·leccions, o definir subdocuments dins d'un document.

1. Relacions entre col·leccions

Tornarem a la nostra base de dades de contactes que venim utilitzant en aquestes sessions. És una base de dades molt simple, amb una única col·lecció anomenada "contactes" els documents dels quals tenen tres camps: nom, telèfon i edat. Li afegirem més informació, i per a això continuarem treballant sobre el projecte "*ContactesMongo*" de la nostra carpeta "*ProjectesNode/Proves*". No obstant això, per a no mesclar els continguts bàsics que hem estat veient amb uns altres més avançats que tractarem a continuació, crea una còpia anomenada *ContactesMongo_v2* per al que farem a continuació.

1.1. Definir una relació simple

Suposem que volem afegir, per a cada contacte, quin és el seu restaurant favorit, de manera que diversos contactes puguen tindre el mateix. Del restaurant en qüestió ens interessa saber el seu nom, adreça i telèfon. Per a això, podem definir aquest esquema i model (en un fitxer anomenat `models/restaurant.js`):

```
let restaurantSchema = new mongoose.Schema({
    nom: {
        type: String,
        required: true,
        minlength: 1,
        trim: true
    },
    adreca: {
        type: String,
        required: true,
        minlength: 1,
        trim: true
    },
    telefon: {
        type: String,
        required: true,
        unique: true,
        trim: true,
        match: /^\d{9}$/
    }
});
let Restaurant = mongoose.model('restaurants', restaurantSchema);
module.export = Restaurant;
```

I ho associem a l'esquema de contactes amb un nou camp (ometem amb punts suspensius dades ja existents d'exemples previs):

```
let contacteSchema = new mongoose.Schema({
    nom: {
        ...
    },
    telefon: {
        ...
    },
    edat: {
        ...
    },
    restaurantFavorit: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'restaurants'
    }
});
let Contacte = mongoose.model('contactes', contacteSchema);
module.exports = Contacte;
```

Observem que el tipus de dada d'aquest nou camp és `ObjectId`, la qual cosa indica que fa referència a un *id* d'un document d'aquesta o una altra col·lecció. En concret, a través de la propietat `ref` indiquem a quin model o col·lecció fa referència dit *id* (al model *restaurants*, que es traduirà a la col·lecció *restaurants* en MongoDB).

1.2. Definir una relació múltiple

Farem un pas més, i a definir una relació que permeta associar a un element d'una col·lecció múltiples elements d'una altra (o d'aqueixa mateixa col·lecció). Per exemple, permetrem que cada contacte tinga un conjunt de mascotes. Definim un nou esquema per a les mascotes, que emmagatzeme el seu nom i tipus (gos, gat, etc.), en l'arxiu `models/mascota.js`.

```
let mascotaSchema = new mongoose.Schema({
    nom: {
        type: String,
        required: true,
        minlength: 1,
        trim: true
    },
    tipus: {
        type: String,
        required: true,
        enum: ['gos', 'gat', 'altres']
    }
});
let Mascota = mongoose.model('mascotes', mascotaSchema);
module.exports = Mascota;
```

> **NOTA**: com a nota al marge, observeu com es pot utilitzar el validador `enum` en un esquema per a forçar al fet que un determinat camp només admeta uns certs valors.

Per a permetre que un contacte puga tindre múltiples mascotes, afegim un nou camp en l'esquema de contactes que serà un array de *ids*, associats al model de mascotes definit prèviament:

```
let contacteSchema = new mongoose.Schema({
    nom: {
        ...
    },
    telefon: {
        ...
    },
    edat: {
        ...
    },
    restaurantFavorit: {
        ...
    },
    mascotes: [{
        type: mongoose.Schema.Types.ObjectId,
        ref: 'mascotes'
    }]
});
let Contacte = mongoose.model('contactes', contacteSchema);
module.exports = Contacte;
```

En aquest cas, observeu com la manera de definir la referència a la col·lecció de mascotes és la mateixa (s'estableix com a tipus de dada un `ObjectId`, amb referència al model de *mascotes*), però, a més, el tipus de dada d'aquest camp mascotes és un array (especificat pels claudàtors en definir-lo).

1.3. Insercions d'elements relacionats

En MongoDB, quan volem inserir un nou contacte i especificar el seu restaurant favorit i/o les seues mascotes, hem de fer-ho en diverses etapes, similar al que ocorreria en un sistema relacional. Este procés es realitza en diversos passos, i és important controlar correctament el flux de les insercions asíncrones per a garantir que totes les referències siguen vàlides.

- Primer afegiríem el restaurant favorit a la col·lecció de restaurants, i/o les mascotes a la col·lecció de mascotes (llevat que existisca prèviament, i en aquest cas obtindríem la seua *id*):

```
let restaurant1 = new Restaurant({
    nom: "La Tagliatella",
    adreca: "C. c. Sant Vicent s/n",
    telefon: "965678912"
});
restaurant1.save().then(...

let mascota1 = new Mascota({
    nom: "Otto",
    tipus: "gos"
});
mascota1.save().then(...
```

- Després, afegiríem el nou contacte especificant l'*id* del seu restaurant favorit, afegit prèviament, i/o els *ids* de les seues mascotes en un array:

```
let contacte1 = new Contacte({
    nom: "Nacho",
    telefon: 677889900,
    edat: 40,
    restaurantFavorit: '5acd3c051d694d04fa26dd8b',
    mascotes: ['5acd3c051d694d04fa26dd90',
    '5acd3c051d694d04fa26dd91']
});
contacte1.save().then(...
```

Evidentment, en una operació "real" no haurem d'afegir a mà els *ids* dels documents relacionats. Bastaria amb triar-los d'alguna mena de desplegable per a quedar-nos amb el seu *id*. Una altra opció és utilitzar la propietat `_id` generada automàticament per MongoDB. Això evita el pas manual d'anar a la base de dades a copiar i pegar els id.

```
let contacte1 = new Contacte({
    nom: "Nacho",
    telefon: 677889900,
    edat: 40,
    restaurantFavorit: restaurant1._id,
    mascotes: [mascota1._id, mascota2._id]
});
contacte1.save().then(...
```

Problema: inserció de referències sense esperar les promeses

En la solució anterior no estem esperant que les promeses es resolguen. El contacte s'està creant sense assegurar-nos que el restaurant i les mascotes han sigut guardats correctament. Això pot resultar en un contacte amb referències a `_id` que no existixen en les col·leccions relacionades.

Solució 1: niar les promeses correctament

Per a assegurar-nos que les insercions ocorren en l'orde correcte, hem de niar les promeses utilitzant `then`. Això garantix que el contacte no es crearà fins que el restaurant i les mascotes s'hagen guardat correctament.

```
restaurant1.save().then((restaurantGuardat) => {
    mascota1.save().then((mascotaGuardada1) => {
        mascota2.save().then((mascotaGuardada2) => {
            let contacte1 = new Contacte({
                nom: "Nacho",
                telefon: 677889900,
                edat: 40,
                restaurantFavorit: restaurantGuardat._id,
                mascotes: [mascotaGuardada1._id, mascotaGuardada2._id]
            });
            contacte1.save().then(...);
        });
    });
});
```

Solució 2: ús de async/await per a un codi més llegible

Una alternativa més clara i manejable és utilitzar `async/await`, la qual cosa permet controlar el flux asíncron de manera seqüencial i evitar la implantació excessiva de promeses.

```
async function guardarContacteAmbRelacions() {
    try {
        // Guardar el restaurant
        let restaurantGuardat = await restaurant1.save();

        // Guardar les mascotes
        let mascotaGuardada1 = await mascota1.save();
        let mascotaGuardada2 = await mascota2.save();

        // Crear el contacte després que el restaurant i les mascotes estiguen guarda
        let contacte1 = new Contacte({
            nomb: "Nacho",
            telefon: 677889900,
            edat: 40,
            restaurantFavorit: restaurantGuardat._id,
            mascotes: [mascotaGuardada1._id, mascotaGuardada2._id]
        });

        // Guardar el contacte
        let contacteGuardat = await contacte1.save();
        console.log("Contacte guardat correctament:", contacteGuardat);

    } catch (error) {
        console.error("Error en guardar les dades relacionades:", error);
    }
}

// Cridar a la funció
guardarContacteAmbRelacions();
```

Solució 3: execució paral·lela d'insercions independents amb Promise.all()

Encara podem millorar la solució proposada, ja que la inserció del restaurant i de les mascotes són independents entre si. Això significa que, en lloc de niar estes operacions i executar-les de manera seqüencial, poden executar-se en paral·lel, optimitzant així el procés. Per a això, podem usar **Promise.all()** per a esperar que totes les promeses (insercions) es resolguen i després procedir amb la creació del contacte.

```
Promise.all([restaurant1.save(), mascota1.save(), mascota2.save()])
    .then(([restaurantGuardat, mascotaGuardada1, mascotaGuardada2]) => {
        // Una vegada que s'hagen guardat el restaurant i les mascotes, podem crear e
        let contacte1 = new Contacte({
            nomb: "Nacho",
            telefon: 677889900,
            edat: 40,
            restaurantFavorit: restaurantGuardat._id, // Referència al _id del restau
            mascotes: [mascotaGuardada1._id, mascotaGuardada2._id] // Referència als
        });

        // Guardar el contacte
        return contacte1.save();
    })
    .then((contacteGuardat) => {
        console.log("Contacte guardat correctament:", contacteGuardat);
    })
    .catch((error) => {
        console.error("Error en el procés d'inserció:", error);
    });
```

Solució 4: ús de async/await amb insercions paral·leles

També podem combinar `async/await` amb insercions en paral·lel per a obtindre un codi net i eficient.

```
async function guardarContacteAmbRelacions() {
    try {
        // Guardar restaurant i mascotes en paral·lel
        const [restaurantGuardat, mascotaGuardada1, mascotaGuardada2] = await Promise
            restaurant1.save(),
            mascota1.save(),
            mascota2.save()
        ]);

        // Crear i guardar el contacte una vegada que les insercions anteriors hagen
        let contacte1 = new Contacte({
            nom: "Nacho",
            telefon: 677889900,
            edat: 40,
            restaurantFavorit: restaurantGuardat._id, // Referència al _id del restau
            mascotes: [mascotaGuardada1._id, mascotaGuardada2._id] // Referència als
        });

        let contacteGuardat = await contacte1.save();
        console.log("Contacte guardat correctament:", contacteGuardat);

    } catch (error) {
        console.error("Error en el procés d'inserció:", error);
    }
}

// Cridar a la funció
guardarContacteAmbRelacions();
```

La combinació de **async/await amb Promise.all()** és la solució més recomanada, ja que combina claredat i eficiència en el maneig d'operacions asíncrones.

1.4. Sobre la integritat referencial

La integritat referencial és un concepte vinculat a bases de dades relacionals, mitjançant el qual es garanteix que els valors d'una clau aliena sempre existiran en la taula a la qual fa referència. Aplicat a una base de dades Mongo, podríem pensar que els *ids* d'un camp vinculat a una altra col·lecció haurien d'existir en aquesta col·lecció, però no té per què ser així.

Seguint amb l'exemple anterior, si intentem inserir un contacte amb un *id* de restaurant que no existisca en la col·lecció de restaurants, ens deixarà fer-ho, sempre que aqueix *id* siga vàlid (és a dir, tinga una extensió de 12 bytes). Per tant, corre per compte del programador assegurar-se que els *id* emprats en insercions que impliquen una referència a una altra col·lecció existisquen realment. Per a facilitar la tasca, existeixen algunes llibreries en el repositori NPM que podem emprar, com per exemple aquesta, encara que el seu ús va més enllà dels continguts d'aquest curs, i no el veurem ací.

En el cas de l'esborrat, podem trobar-nos amb una situació similar: si, seguint amb el cas dels contactes, volem esborrar un restaurant, haurem d'anar amb compte amb els contactes que el tenen assignat com a restaurant favorit, ja que l'*id* deixarà d'existir en la col·lecció de restaurants. Així, seria convenient triar entre una d'aquestes dues opcions, encara que les dues requereixen un tractament manual per part del programador:

- Denegar l'operació si existeixen contactes amb el restaurant seleccionat
- Reassignar (o posar a nul) el restaurant favorit d'aqueixos contactes, abans d'eliminar el restaurant seleccionat.

> **Exercici 1:**
>
> Modificarem l'exercici *LlibresMongo* iniciat en la sessió anterior. Fes una còpia i canvia-la de nom a **LlibresMongo_v2** per a treballar ara amb aquesta nova versió.
>
> - Definirem ara un segon esquema per a emmagatzemar informació sobre el **autor** de cada llibre. Aquest autor tindrà un nom (obligatori) i un any de naixement (opcional, però amb valors entre 0 i 2000). Defineix també el model per a la col·lecció, associat a aquest esquema.
> - Després, relaciona la col·lecció de llibres amb la d'autors, afegint a la primera un nou camp anomenat `autor`, que enllaçarà amb l'*id* de l'autor corresponent en la col·lecció d'autors. Aquest camp *autor* no serà obligatori, per a respectar així els llibres sense autor que tinguem afegits amb anterioritat.
> - Després de la connexió a la base de dades i la definició d'esquemes que hem fet, elimina el codi de l'exercici anterior relatiu a insercions, esborrats, modificacions i consultes, i afig el codi per a inserir un o dos nous autors, i algun llibre vinculat a cadascun d'ells.

2. Subdocuments

Mongoose ofereix també la possibilitat de definir **subdocuments**. Vegem un exemple concret d'això, i per a això, farem una versió alternativa del nostre exemple de contactes. Còpia la carpeta *ContactesMongo_v2* que hem vingut completant fins ara, i anomena a la nova còpia *ContactesMongo_v3*.

Sobre aquest nou projecte, en el nostre arxiu `index.js`, connectarem amb una nova base de dades, que anomenarem `contactes_subdocuments`, per a no interferir amb la base de dades anterior:

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

I reagruparem els tres esquemes que hem fet fins ara (restaurants, mascotes i contactes), per a unir-los en el de contactes. Deixarem, per tant, un únic arxiu en la carpeta `models`, que serà `contacte.js`, amb aquest contingut (ometem amb punts suspensius part del codi que és el mateix de l'exemple anterior):

```
// Restaurants
let restaurantSchema = new mongoose.Schema({
    ... // Codi de l'esquema de restaurant
});

// Mascotes
let mascotaSchema = new mongoose.Schema({
    ... // Codi de l'esquema de mascota
});

// Contactes
let contacteSchema = new mongoose.Schema({
    nom: {
        ...
    },
    telefon: {
        ...
    },
    edat: {
        ...
    },
    restaurantFavorit: restaurantSchema,
    mascotes: [mascotaSchema]
});
let Contacte = mongoose.model('contactes', contacteSchema);
module.exports = Contacte;
```

Observeu les línies que es refereixen a les propietats `restaurantFavorit` i `mascotes`. És la manera d'associar un esquema sencer com a tipus de dada d'un camp d'un altre esquema. D'aquesta manera, convertim l'esquema en una part de l'altre, creant així **subdocuments** dins del document principal. Observeu també que no s'han definit models ni per als restaurants ni per a les mascotes, ja que ara no tindran una col·lecció pròpia.

Un subdocument, a priori, pot semblar una cosa equivalent a definir una relació entre col·leccions. No obstant això, la principal diferència entre un subdocument i una relació entre documents de col·leccions diferents és que el subdocument queda embegut dins del document principal, i és diferent de qualsevol altre objecte que puga haver-hi en un altre document, encara que els seus camps siguen iguals. Per contra, en la relació simple vista abans entre restaurants i contactes, un restaurant favorit podia ser compartit per diversos contactes, simplement enllaçant amb el mateix *id* de restaurant. Però, d'aquesta altra manera, creem el restaurant per a cada contacte, diferenciant-lo dels altres restaurants, encara que siguen iguals. El mateix ocorreria amb l'array de mascotes: les mascotes serien diferents per a cada contacte, encara que volguérem que foren la mateixa o pogueren compartir-se.

2.1. Inserció de documents amb subdocuments

Si volem crear i guardar un contacte que conté com subdocuments el restaurant favorit i les seues mascotes, podem crear tot l'objecte complet, i fer un únic guardat (`save`).

```
let contacte1 = new Contacte({
    nom: 'Nacho',
    telefon: 966112233,
    edat: 39,
    restaurantFavorit: {
        nom: 'La Tagliatella',
        adreca: 'C. c. Sant Vicent s/n',
        telefon: 961234567
    }
});
contacte1.mascotes.push({nom:'Otto', tipus:'gos'});
contacte1.mascotes.push({nom:'Piolín', tipus:'altres'});
contacte1.save().then(...
```

En aquest exemple es mostren dues formes possibles d'emplenar els subdocuments del document principal: sobre la marxa quan creem aquest document (cas del restaurant), o a posteriori, accedint als camps i donant-los valor (cas de les mascotes).

En la base de dades que es crea, veurem que només existeix una col·lecció, *contactes*, i en examinar els elements que inserim veurem que contenen embeguts els subdocuments que hem definit:


```
{
  "_id": {
    "$oid": "64ad57647496c43432b0472f"
  },
  "nombre": "Nacho",
  "telefono": "966112233",
  "edad": 39,
  "restauranteFavorito": {
    "nombre": "La Tagliatella",
    "direccion": "C.C. San Vicente s/n",
    "telefono": "961234567",
    "_id": {
      "$oid": "64ad57647496c43432b04730"
    }
  },
  "mascotas": [
    {
      "nombre": "Otto",
      "tipo": "perro",
      "_id": {
        "$oid": "64ad57647496c43432b04731"
      }
    },
    {
      "nombre": "Piolín",
      "tipo": "otros",
      "_id": {
        "$oid": "64ad57647496c43432b04732"
      }
    }
  ],
  "__v": 0
```


2.2. Quan definir relacions i quan subdocuments?

La resposta a aquesta pregunta pot resultar complexa o evident, depenent de com hàgem entés els conceptes vistos fins ara, però intentarem donar unes normes bàsiques per a distingir quan usar cada concepte:

- Emprarem **relacions entre col·leccions** quan vulguem poder compartir un mateix document d'una col·lecció per diversos documents d'una altra. Així, en el cas dels restaurants favorits, sembla lògic utilitzar una relació entre col·leccions a partir de l'*id* del restaurant, i així permetre que diversos contactes puguen compartir una mateixa instància d'un restaurant favorit.
- Emprarem **subdocuments** quan no importe aquesta compartició d'informació, o quan prevalga la simplicitat de definició d'un objecte enfront de l'asociatividad entre col·leccions. Dit d'una altra manera, i aplicat a l'exemple de les mascotes, si volem accedir de manera senzilla a les mascotes d'un contacte, sense importar si un altre contacte té les mateixes mascotes, utilitzarem subdocuments.

En el cas dels subdocuments queda, per tant, una assignatura pendent: la possible duplicitat d'informació. Si hi ha dues persones que tenen la mateixa mascota, haurem de crear dos objectes iguals per a totes dues persones, duplicant així les dades de la mascota. No obstant això, aquesta duplicitat de dades ens facilitarà l'accedir a les mascotes d'una persona, sense haver de recórrer a altres eines que veurem a continuació.

> **Exercici 2:**
>
> Sobre l'exercici anterior, defineix un nou esquema en el fitxer de `models/llibre.js` per a emmagatzemar comentaris relatius a un llibre. Cada comentari tindrà una data (tipus `Date`), el nick de qui fa el comentari (`String`) i el comentari en si (`String`), sent tots aquests camps obligatoris. A més, en el cas de la data, establirem com a valor per defecte (`default`) la data actual (`Date.now`).
>
> Aquesta vegada no definisques un model per a aquest esquema. Crearem un subdocument dins de l'esquema de llibres que emmagatzeme un array de comentaris per a aquest llibre, utilitzant l'esquema de comentaris que acabes de crear.
>
> Una vegada fet això, crea un nou llibre amb les seues dades, i afig a mà un parell de comentaris a l'array, abans de guardar totes les dades.

3. Consultes avançades

Ara que ja sabem definir diferents tipus de col·leccions vinculades entre si, vegem com definir consultes que s'aprofiten d'aquestes vinculacions per a extraure la informació que necessitem. Tornarem a treballar, en aquest cas, amb el projecte *ContactesMongo_v2*.

3.1. Les poblacions (*populate*)

El fet de relacionar documents d'una col·lecció amb documents d'una altra a través dels *id* corresponents permet obtindre en un sol llistat la informació de totes dues col·leccions, encara que per a això necessitem d'algun pas intermedi. Per exemple, si volem obtindre tota la informació dels nostres contactes, relacionats amb les col·leccions de restaurants i mascotes (arxiu `index.js` del nostre projecte de "*ContactesMongo_v2*"), podem fer alguna cosa com això:

```
Contacte.find().then(resultat => {
    console.log(resultat);
});
```

No obstant això, aquesta instrucció es limita, òbviament, a mostrar l'*id* dels restaurants favorits i de les mascotes, però no les dades completes d'aquests. Per a fer això, hem de tirar mà d'un mètode molt útil oferit per Mongoose, anomenat `populate`. Aquest mètode permet incorporar la informació associada al model que se li indique. Per exemple, si volem incorporar al llistat anterior tota la informació del restaurant favorit de cada contacte, farem una cosa així:

```
Contacte.find().populate('restaurantFavorit').then(resultat => {
    console.log(resultat);
});
```

Si tinguérem més camps relacionats, podríem enllaçar diverses sentències `populate`, l'una després de l'altra, per a poblar-los. Per exemple, així poblaríem tant el restaurant com les mascotes:

```
Contacte.find()
.populate('restaurantFavorit')
.populate('mascotes')
.then(resultat => {
    console.log(resultat);
});
```

Existeixen altres opcions per a poblar els camps. Per exemple, podem voler poblar només part de la informació, com el nom del restaurant només. En aqueix cas, utilitzem una sèrie de paràmetres addicionals en el mètode `populate`:

```
Contacte.find()
.populate('restaurantFavorit', 'nom')
...
```

3.2. Consultes que relacionen diverses col·leccions

Establir una consulta general sobre una col·lecció és senzill, com hem vist en sessions anteriors. Podem utilitzar el mètode `find` per a obtindre documents que complisquen determinats criteris, o alternatives com `findOne` o `findById` per a obtindre el document que complisca el filtrat.

Les bases de dades No-SQL, com és el cas de MongoDB, no estan preparades per a consultar informació provinent de diverses col·leccions, la qual cosa en part "convida" a utilitzar col·leccions independents basades

en subdocuments per a agregar informació addicional.

Suposem que volem, per exemple, obtindre les dades dels restaurants favorits d'aquells contactes que siguen majors de 30 anys. Si tinguérem una base de dades SQL, podríem resoldre això amb una query com la següent:

```
SELECT * FROM restaurants
WHERE id IN
(SELECT restaurantFavorit FROM contactes
WHERE edat > 30)
```

Tanmateix, això no és possible en MongoDB o, almenys, no de forma tan immediata. Caldria dividir aquesta consulta en dues parts: primer obtindre els *id* dels restaurants de les persones majors de 30 anys, i a partir d'ací obtindre amb una altra consulta les dades d'aqueixos restaurants. Podria quedar més o menys així:

```
Contacte.find({edat: {$gt: 30}}).then(resultatContactes => {
    let idsRestaurants =
        resultatContactes.map(contacte => contacte.restaurantFavorit);
    Restaurant.find({_id: {$in: idsRestaurants}})
    .then(resultatFinal => {
        console.log(resultatFinal);
    });
});
```

Observeu que la primera consulta obté tots els contactes majors de 30 anys. Una vegada aconseguits, fem un mapatge (`map`) per a quedar-nos només amb els *id* dels restaurants favorits, i aqueix llistat de *ids* l'utilitzem en la segona consulta, per a quedar-nos amb els restaurants que el seu *id* estiga en aqueix llistat.

> **Exercici 3**:
>
> Abans de seguir amb aquest exercici, procura que hi haja almenys dos o tres autors en la col·lecció d'autors, i almenys tres o quatre llibres amb autors diferents. Una vegada fet això, afig al programa anterior una consulta que mostre els noms dels autors que tinguen algun llibre a la venda per menys de 10 euros (únicament hauran de mostrar-se els noms dels autors en el llistat).

3.3. Altres opcions en les consultes

Quan utilitzem el mètode `find` o similars, existeixen opcions addicionals que permeten, per exemple, especificar quins camps volem obtindre, o criteris d'ordenació, o de límit màxim de resultats a obtindre, etc. En l'anterior sessió vam veure algun exemple sobre aquest tema, però vegem ara amb una mica més de detall algunes d'aquestes opcions:

- Si volem especificar concretament quins camps volem obtindre en el llistat, s'especifica com una cadena de text en els paràmetres de `find` . Com a alternativa, es pot enllaçar la crida normal a `find` amb el

mètode `select`, on s'especifiquen els camps a obtindre. Aquests dos exemples fan el mateix: mostrar el nom i l'edat dels contactes majors de 30 anys:

```
Contacte.find({edat: {$gt: 30}}, 'nom edat').then(...
Contacte.find({edat: {$gt: 30}}).select('nom edat').then(...
```

- Per a **ordenar** el llistat per algun criteri, s'enllaça la cridada a `find` amb una altra al mètode `sort`, en el qual s'especifica el camp pel qual ordenar, i l'ordre (1 per a ordre ascendent, -1 per a ordre descendent). El següent llistat mostra ordenats de major a menor edat els contactes. S'indica també una alternativa equivalent, que consisteix a anteposar el signe menys `-` al camp pel qual ordenar, indicant que es vol un ordre descendent:

```
Contacte.find().sort({edat: -1}).then(...
Contacte.find().sort('-edat').then(...
```

- Per a **limitar el nombre de resultats a obtindre**, s'empra el mètode `limit`, indicant quants documents obtindre. El següent exemple mostra, de major a menor edat, els 5 primers contactes:

```
Contacte.find().sort('-edat').limit(5).then(...
```

> **Exercici 4:**
>
> Afig a l'exercici anterior una consulta que mostre el títol i preu (juntament amb l'*id*) dels tres llibres més barats, ordenats de menor a major preu. En el cas que hi haja menys de tres llibres en el llistat, es mostraran només els llibres disponibles, òbviament.

Exercici 5:

> Consultes extra sobre la col·lecció de llibres:
>
> - **Buscar llibres per editorial:** realitza una consulta que mostre el títol i l'editorial de tots els llibres que pertanguen a una editorial específica.
> - **Filtrar llibres ordenats per rang de preus:** realitza una consulta que retorne els llibres el preu dels quals estiga entre dos valors donats, ordenats de major a menor preu.
> - **Llibres sense comentaris:** realitza una consulta que retorne tots els llibres que no tinguen comentaris associats. Filtra aquells llibres que el seu *array de comentaris estiga buit o no existisca. Pots utilitzar els operadors `$exists` (per a verificar si un camp esta present) i `$size` (per a saber la quantitat d'elements en el *array).
> - **Actualització massiva de preus:** crea una consulta que actualitze el preu de tots els llibres d'una editorial específica, incrementant-lo en un 10%. Mostra els resultats actualitzats després de realitzar l'operació. Pots utilitzar l'operador `$mul` (multiplica el valor d'un camp per un número determinat

en una operació d'actualització). No és necessari usar `$set` en este cas, perquè `$mul` directament modifica el valor del camp. `$set` s'usa quan s'assigna un valor nou, però ací només s'està modificant el valor existent mitjançant una multiplicació.

- **Buscar llibres per autor nascut en un rang d'anys**: realitza una consulta que mostre els llibres l'autor dels quals haja nascut entre dos anys específics. Utilitza la relació entre les col·leccions de Llibre i Autor per a realitzar esta consulta.
- **Llistar els comentaris d'un llibre específic:** realitza una consulta que recupere el títol d'un llibre i tots els comentaris associats, utilitzant el seu ID.