Desenvolupament de serveis REST amb Express

En aquest apartat veurem com emprar un encaminament simple per a oferir diferents serveis emprant Mongoose contra una base de dades MongoDB. Per a això, crearem una carpeta anomenada "*ContactesREST*" en la nostra carpeta de proves ("*ProjectesNode/Proves*"), continuació del projecte *ContactesMongo_v2* de sessions anteriors (còpia i pega el codi d'aqueix projecte en aquesta nova carpeta). Instal·larem Express i Mongoose en ella, la qual cosa pot fer-se amb un simple comando (encara que prèviament necessitarem haver executat `npm init` per a crear l'arxiu *package.json*):

```
npm install mongoose express
```

La nostra aplicació tindrà una carpeta `models` amb els models de dades (contactes, restaurants i mascotes, segons havíem creat en versions anteriors) i un arxiu `index.js`, que anteriorment llançava una sèrie d'operacions simples, i on ara definirem el nostre servidor Express amb les rutes per a respondre a les diferents operacions sobre els contactes.

1. Esquelet bàsic del servidor principal

Definirem l'estructura bàsica que tindrà el nostre servidor `index.js`, abans d'afegir-li les rutes per a donar resposta als serveis. Aquesta estructura consistirà a incorporar Express i Mongoose, incorporar els models de dades, connectar amb la base de dades i posar en marxa el servidor Express:

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

const Contacte = require(__dirname + "/models/contacte");
const Restaurant = require(__dirname + "/models/restaurant");
const Mascota = require(__dirname + "/models/mascota");

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

let app = express();
app.listen(8080);
```

2. Serveis de llistat (GET)

Veurem ara com oferir diferents serveis de llistat per a recuperar informació de la base de dades, o bé amb llistats generals de diversos documents, o amb llistats específics d'un sol document.

2.1. Llistat de tots els contactes

El servei que llesta tots els contactes és el més senzill: atendrem per GET a la URI `/contactes`, i en el codi farem un `find` de tots els contactes, usant el model Mongoose que ja hem creat. Retornarem el resultat directament en la resposta, la qual cosa el convertirà automàticament a format JSON. Afegim el servei en l'arxiu principal `index.js`. Normalment s'afigen abans de posar en marxa el servidor (després d'haver creat la variable `app`).

```
app.get('/contactes', (req, res) => {
    Contacte.find().then(resultat => {
        res.status(200)
           .send( {ok: true, resultat: resultat});
    }).catch (error => {
        res.status(500)
           .send( {ok: false,
                   error: "Error obtenint contactes"});
    });
});
```

A continuació es mostra com quedaria reescrit el mateix codi usant `async/await`:

```
app.get('/contactes', async (req, res) => {
    try {
        const resultat = await Contacte.find(); // Espera la resposta de la base de d
        res.status(200).send({ ok: true, resultat: resultat });
    } catch (error) {
    res.status(500).send({ ok: false, error: "Error obtenint contactes" });
    }
});
```

Observeu que enviem un codi d'estat (200 si tot ha anat bé, 500 o fallada del servidor si no hem pogut recuperar els contactes), i l'objecte JSON amb els camps que explicàvem abans: la dada booleana indicant si s'ha pogut servir o no la resposta, i el missatge d'error o el resultat corresponent, segons siga el cas.

2.2. Fitxa d'un contacte a partir del seu *id*

Vegem ara com processar amb Express URIs dinàmiques per a manejar dades específiques que varien en les rutes, com un ID de contacte.

En este cas, accedirem per GET a una URI amb el format `/contactes/:id`, sent *:id* un paràmetre dinàmic que representa el *id* del contacte que volem obtindre.

Quan definim una ruta amb este format, la part *:id* automàticament s'associa amb qualsevol valor que seguisca a */contactes/* en la URL. Este valor es pot accedir a través de `req.params`, que és un objecte que

conté els paràmetres de la ruta.

D'esta manera, el servici queda així de simple:

```
app.get('/contactes/:id', (req, res) => {
    Contacte.findById(req.params.id).then(resultat => {
        if(resultat)
            res.status(200)
               .send({ok: true, resultat: resultat});
        else
            res.status(400)
               .send({ok: false,
                      error: "No s'han trobat contactes"});
    }).catch (error => {
        res.status(400)
           .send({ok: false,
                  error: "Error buscant el contacte indicat"});
    });
});
```

En aquest cas, distingim si l'objecte `resultat` obtingut amb `findById` retorna alguna cosa o no, per a emetre l'una o l'altra resposta. En cas que no es puga trobar el resultat, assumim que és a causa que la petició del client no és correcta, i emetem un codi d'estat 400 (per exemple).

2.3. Ús de la *query string* per a passar paràmetres

En el cas de voler passar els paràmetres en la *query string* (és a dir, per exemple, `/contactes?id=XXX`) no hi ha manera d'establir aquests paràmetres en la URI del mètode `get`. En aqueix cas haurem de comprovar si existeix el paràmetre corresponent dins de l'objecte `req.query`:

```
app.get('/contactes', (req, res) => {
    if(req.query.id) {
        // Buscar per id
    } else {
        // Llistat general de contactes
    });
});
```

En qualsevol cas, en aquestes anotacions optarem per la versió anterior, incloent-hi el paràmetre en la pròpia URI.

2.4. Prova dels serveis des del navegador

Aquests dos serveis de llistat (general i per id) es poden provar fàcilment des d'un navegador web. N'hi ha prou amb posar en marxa el servidor Node (i MongoDB), obrir un navegador i accedir a aquesta URL per al llistat general:

http://localhost:8080/contactes

O a aquesta altra per a la fitxa d'un contacte (substituint l'*id* de la URL per un correcte que existisca en la base de dades):

http://localhost:8080/contactes/5ab391d296b06243a7cc4c4e

En aquest últim cas, observa que:

- Si passem un *id* que no existisca, ens indicarà amb un missatge d'error que "No s'han trobat contactes"
- Si passem un *id* que no siga adequat (per exemple, que no tinga 12 bytes), obtindrem una excepció, i per tant el missatge de "Error buscant el contacte indicat".

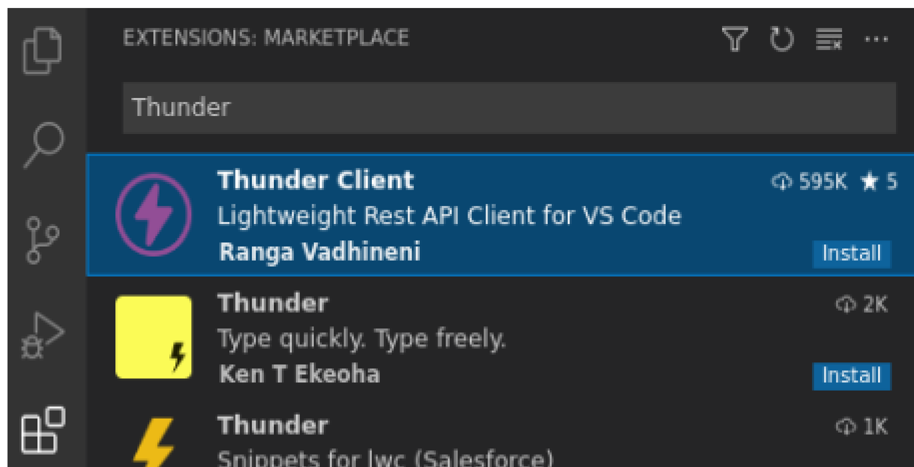
> **Exercici 1:**
>
> Crea un exercici anomenat **LlibresREST** i còpia dins l'estructura de models del projecte **LlibresMongo_v2** de sessions anteriors. Instal·la *Mongoose* i *Express* en el projecte, i defineix un arxiu `index.js` que incorpore el model de llibres, cree una instància de servidor Express, i done resposta a aquests serveis:
>
> - `GET /llibres` : retornarà un llistat en format JSON de l'array de llibres complet de la col·lecció.
> - `GET /llibres/:id` : retornarà un objecte JSON amb les dades del llibre trobat a partir del seu *id*.

3. Provar serveis REST

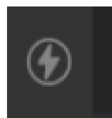
Ja hem vist que provar uns serveis de llistat (GET) és senzill a través d'un navegador. No obstant això, prompte aprendrem a fer altres serveis (insercions, modificacions i esborrats) que no són tan senzills de provar amb aquesta eina. Així que convé anar entrant en contacte amb una altra més potent, que ens permeta provar tots els serveis que desenvoluparem. Existeixen diverses alternatives sobre aquest tema, com per exemple Postman, però per a evitar dependre de més aplicacions externes, utilitzarem l'extensió **ThunderClient** de Visual Studio Code.

3.1. Instal·lació de l'eina i primers passos

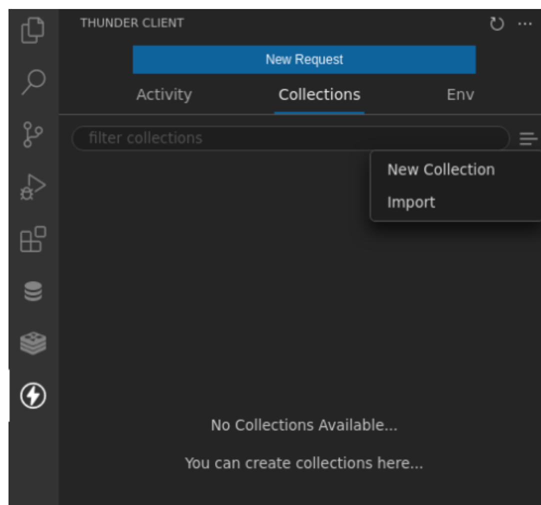
Aquesta eina ens servirà per a simular peticions a servidors web, i recollir i analitzar la resposta. L'emprarem per a provar els serveis REST que desenvoluparem. La busquem en el panell d'extensions i la instal·lem:



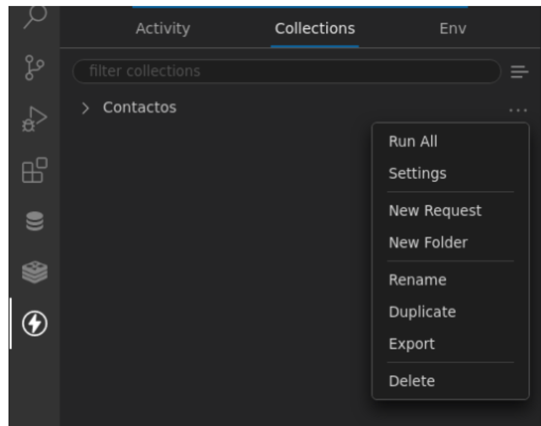
Ens apareixerà una icona en el panell esquerre des del qual gestionarem les connexions i peticions:



És aconsellable que les nostres peticions les agrupem en col·leccions (*collections*) de manera que cada col·lecció se centre en un projecte o aplicació concreta. Per a això anem a la pestanya de col·leccions i triem crear una, amb el nom que vulguem (per exemple, *Contactes*).

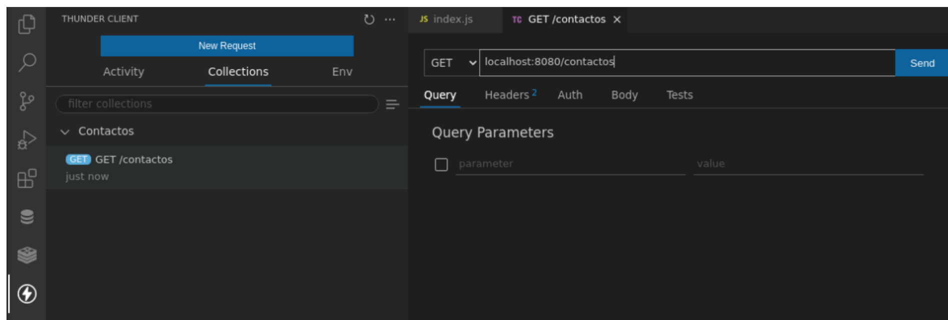


Des del botó de punts suspensius al costat del nom de la col·lecció podrem afegir noves peticions associades a aquesta col·lecció (*New Request*), i després li posarem un nom.

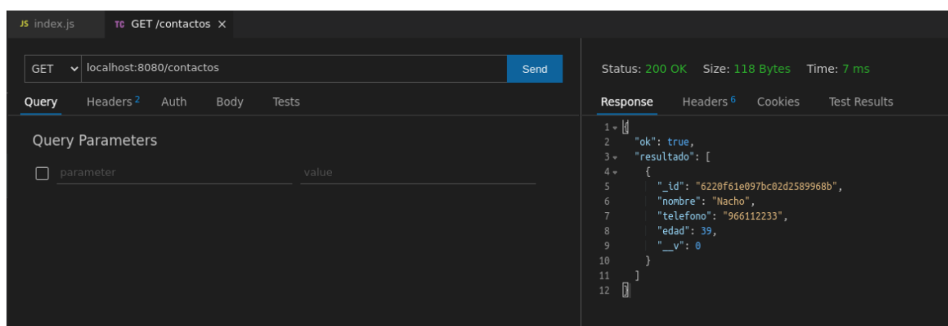


3.2. Afegir peticions GET

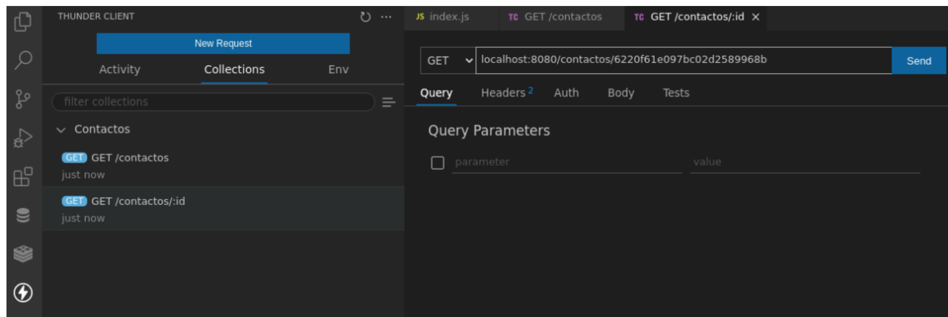
Per a afegir una petició GET, podem en primer lloc identificar-la amb el seu nom (per exemple, *GET /contactes*), i en el panell que s'obrirà definim la URL d'accés (també podem triar el tipus de comando: GET, POST, etc). Per exemple:



Si premem en el botó de *Send*, podem veure la resposta emesa pel servidor en el panell de resposta:



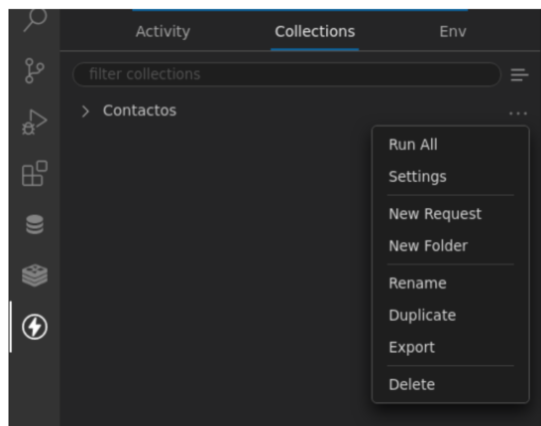
Seguint aquests mateixos passos, podem també crear una nova petició per a obtindre un contacte a partir del seu *id*, per GET:



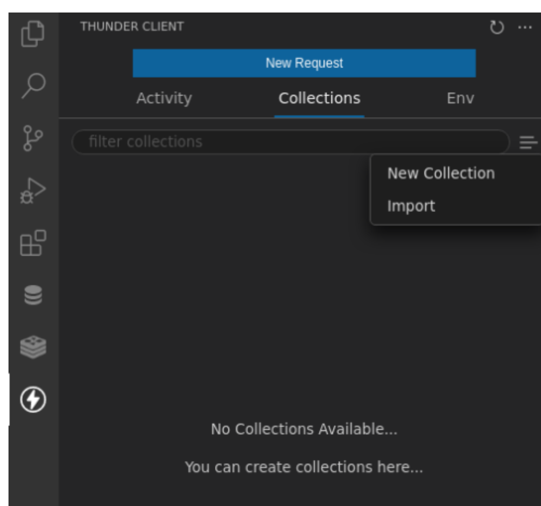
Bastaria amb reemplaçar l'*id* de la URL pel qual vulguem consultar realment.

3.3. Exportar/Importar col·leccions

Podem exportar i importar les nostres col·leccions en *ThunderClient*, de manera que podem portar-les d'un equip a un altre. Per a **exportar** una col·lecció, fem clic en el botó de punts suspensius (...) que hi ha al costat d'ella en el panell esquerre, i triem *Export*.



Si volem **importar** una col·lecció prèviament exportada, podem fer clic en el botó per a crear col·leccions, i triem l'opció *Import*:



En qualsevol dels dos casos, les col·leccions s'exporten/importen a/des d'un arxiu en format JSON que conté la informació de cada petició.

> **Exercici 2:**
>
> Crea una col·lecció **Llibres** en *ThunderClient* i defineix en ella una petició per a cadascun dels dos serveis que has implementat abans (*GET /llibres* i *GET /llibres/:id*).

4. Operacions d'actualització (POST, PUT, DELETE)

Després d'haver vist com afegir serveis GET a un servidor Express, queden pendents les altres tres operacions bàsiques (POST, PUT i DELETE), així que veurem ara com desenvolupar-les en Express, i com provar-les amb l'eina *ThunderClient*.

4.1. Les insercions (POST)

Inserirem un nou contacte, passant en el cos de la petició les dades del mateix (nom, telèfon i edat) en format JSON. Per a poder recollir aquesta informació des del client en el nostre servidor Node/Express, hem d'utilitzar un *middleware*, fragment de programa que processa la petició abans d'emetre la resposta per a, en aquest cas, deixar-nos accessibles i preparades aquestes dades.

Express incorpora, des de la seua versió 4.16, un middleware propi per a aquesta comesa. N'hi ha prou amb afegir-ho a l'aplicació, just després d'inicialitzar la `app` Express. Com el que farem és treballar amb objectes JSON, afegirem el processador JSON d'aquesta manera:

```
let app = express();
app.use(express.json());
...
```

> **NOTA**: abans de la versió 4.16, per a aquesta comesa era necessari utilitzar una llibreria de tercers anomenada *bodi-parser*. Ací teniu la documentació sobre aquesta llibreria.

Ara anem al nostre servei POST. Afegim per a això un mètode `post` de l'objecte `app`, amb els mateixos paràmetres que tenia el mètode `get` (recordem: la ruta a la qual respondre, i el *callback* que s'executarà com a resposta). El mètode en qüestió podria quedar així:

```
app.post('/contactes', (req, res) => {

    let nouContacte = new Contacte({
        nom: req.body.nom,
        telefon: req.body.telefon,
        edat: req.body.edat
    });

    nouContacte.save().then(resultat => {
        res.status(200)
           .send({ok: true, resultat: resultat});
    }).catch(error => {
        res.status(400)
           .send({ok: false,
                  error: "Error afegint contacte"});
    });
});
```

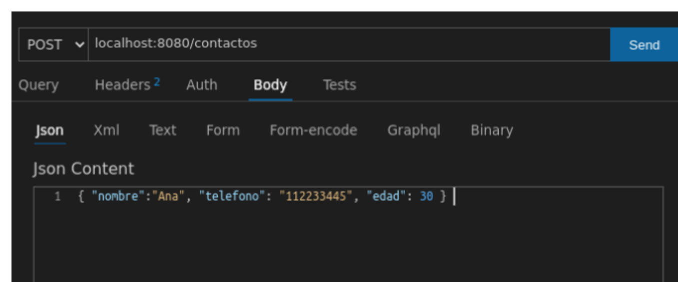
Al principi, construïm el contacte a partir de les dades JSON que arriben en el cos, accedint a cada camp per separat amb `req.body.nom_camp`. Això és possible fer-ho gràcies a que el *middleware* anterior ha pre-processat la petició, i ens ha deixat les dades disponibles dins de l'objecte `req.body`. En cas contrari, hauríem de llegir els bytes de la petició manualment, emmagatzemar-los en un array, i convertir des de JSON.

La resta del codi és el que ja coneixes d'exemples previs amb Mongoose (cridar a `save` i processar el resultat), combinat amb l'enviament del codi d'estat i la resposta REST.

4.1.1. Prova d'operacions POST

Les peticions POST difereixen de les peticions GET en què s'envia una certa informació en el cos de la petició. Aquesta informació normalment són les dades que es volen afegir en el servidor. Com podem fer això amb *ThunderClient*?

En primer lloc, creem una nova petició, triem el comando POST i definim la URL (en aquest cas, *localhost:8080/contactes*). Llavors, fem clic en la pestanya *Body*, sota la URL, i establim el tipus com *JSON* perquè ens deixe escriure'l en aquest format. Després, en el quadre de text sota aquestes opcions, especifiquem l'objecte JSON que volem enviar per a inserir:



En enviar la petició, podrem veure en el quadre la resposta (en aquest cas, el document que s'acaba d'inserir).

> **Exercici 3:**
>
> Afig el servei `POST /llibres` al projecte *LlibresREST* anterior. Recollirà les dades del llibre que li arribaran en el cos de la petició i inserirà el llibre en qüestió en la base de dades, retornant un objecte JSON amb el llibre inserit. Afig també la corresponent prova en la col·lecció de *ThunderClient*.

4.2. Les modificacions (PUT)

La modificació de contactes és estructuralment molt similar a la inserció: enviarem en el cos de la petició les dades noves del contacte a modificar (a partir del seu *id*, normalment), i utilitzarem el mateix *middleware* anterior per a obtindre'ls, i cridar als mètodes apropiats de Mongoose per a realitzar la modificació del contacte. La URI a la qual associarem aquest servei serà similar a la del POST, però afegint l'*id* del contacte que vulguem modificar. El codi pot ser similar a aquest:

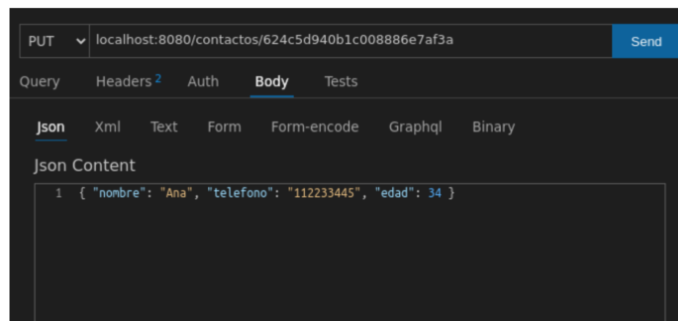
```
app.put('/contactes/:id', (req, res) => {

    Contacte.findByIdAndUpdate(req.params.id, {
        $set: {
            nom: req.body.nom,
            telefon: req.body.telefon,
            edat: req.body.edat
        }
    }, {new: true}).then(resultat => {
        res.status(200)
           .send({ok: true, resultat: resultat});
    }).catch(error => {
        res.status(400)
           .send({ok: false,
                  error:"Error actualitzant contacte"});
    });
});
```

Com es pot veure, obtenim l'*id* des dels paràmetres (`req.params`) com quan consultàvem la fitxa d'un contacte, i utilitzem dit *id* per a buscar al contacte en qüestió i actualitzar els seus camps amb `findByIdAndUpdate` . En aquest punt, tornem a fer ús del *middleware* d'Express per a processar la petició i obtindre les dades que arriben en format JSON. Després de la cridada al mètode, retornem l'estat i la resposta JSON corresponent.

4.2.1. Prova d'operacions PUT

En el cas de peticions PUT, procedirem de manera similar a les peticions POST vistes abans: hem de triar el comando (PUT en aquest cas), la URL, i completar el cos de la petició amb les dades que vulguem modificar del contacte. En aquest cas, a més, l'*id* del contacte l'enviarem també en la pròpia URL:



> **Exercici 4:**
>
> Afig el servei `PUT /llibres/:id` al projecte *LlibresREST* anterior. Recollirà les dades del llibre que li arribaran en el cos de la petició, juntament amb l'*id* en la URL, i modificarà les dades del llibre indicat, retornant un objecte JSON amb el llibre modificat, o el missatge d'error corresponent si no s'ha pogut modificar. Afig també la corresponent prova en la col·lecció de *ThunderClient*.

4.3. Els esborrats (DELETE)

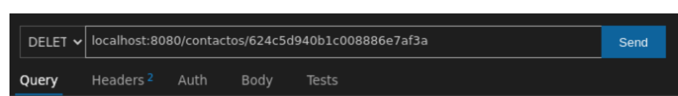
Per a l'esborrat de contactes, emprarem una URI similar a la fitxa d'un contacte o a l'actualització, però en aquest cas associada al comando DELETE. Li passarem l'*id* del contacte a esborrar. Obtindrem dit *id* també de `req.params`, i buscarem i eliminarem el contacte indicat.

```
app.delete('/contactes/:id', (req, res) => {

    Contacte.findByIdAndRemove(req.params.id)
    .then(resultat => {
        res.status(200)
           .send({ok: true, resultat: resultat});
    }).catch(error => {
        res.status(400)
           .send({ok: false,
                  error:"Error eliminant contacte"});
    });
});
```

4.3.1. Prova d'operacions DELETE

Per a peticions DELETE, la mecànica és similar a la fitxa del contacte, canviant el comando GET per DELETE, i sense necessitat d'establir res en el cos de la petició:



> **Exercici 5:**

> Afig el servei `DELETE /llibres/:id` al projecte *LlibresREST* anterior. Recollirà l'*id* del llibre en la URL i l'esborrarà, retornant un objecte JSON amb el llibre eliminat, o el missatge d'error corresponent si no s'ha pogut modificar. Afig també la corresponent prova en la col·lecció de *ThunderClient*.

4.4. Sobre el resultat de l'actualització o l'esborrat

En els exemples anteriors per a PUT i DELETE, després de cridar a `findByIdAndUpdate` o `findByIdAndRemove`, ens hem limitat a retornar el resultat en la clàusula `then`. No obstant això, convé tindre en compte que, si proporcionem un *id* vàlid però que no existisca en la base de dades, el codi d'aquesta clàusula també s'executarà, però l'objecte resultat serà nul (`null`). Podem, per tant, diferenciar amb `if..else` si el resultat és correcte o no, i mostrar l'una o l'altra cosa:

```
if (resultat)
    res.status(200)
       .send({ok: true, resultat: resultat});
else
    res.status(400)
       .send({ok: false,
              error: "No s'ha trobat el contacte"});
```

> **Exercici 6:**
>
> Adapta el codi de les peticions PUT i DELETE del projecte *LlibresREST* perquè controlen si el resultat és nul o no, mostrant una resposta més controlada. Exporta la col·lecció final de proves.

4.5. Més sobre el *middleware* de processament de la petició

Existeixen altres possibilitats d'ús del *middleware* que processa la petició. En els exemples anteriors ho hem emprat per a processar cossos amb format JSON. Però és possible també que emprem formularis tradicionals HTML, que envien les dades com si foren part d'una *query-string*, però per POST. Per exemple:

```
nom=Nacho&telefono=911223344&edat=39
```

Per a processar continguts d'aquest altre tipus, n'hi ha prou amb carregar el *middleware* d'aquesta altra manera:

```
app.use(express.urlencoded());
```

També és possible afegir totes dues maneres juntes:

```
app.use(express.json());
app.use(express.urlencoded());
```

En aquest cas, el servidor Express acceptaria dades de la petició tant en format JSON com en format *query-string*. En qualsevol cas, haurem d'assegurar-nos des del client (fins i tot si usem *ThunderClient*) que el tipus de contingut de la petició s'ajusta al *middleware* corresponent: per a peticions en format JSON, el contingut haurà de ser `application/json` (pestanya *JSON* en *ThunderClient*), mentre que per a enviar les dades del formulari en format *query-string*, el tipus haurà de ser `application/x-www-form-urlencoded` (pestanya *Form-encode* en *ThunderClient*). Si afegim els dos middlewares (tant per a JSON com per a `urlencoded`), llavors s'activarà l'un o l'altre automàticament, depenent del tipus de petició que arribe des del client.