Gestió de formularis


Anem ara a afegir formularis a la nostra aplicació que ens permetran inserir, esborrar o modificar contingut de la base de dades. També modificarem les corresponents rutes per a, d'una banda, mostrar aquests formularis, i per una altra, recollir la petició i fer la inserció/esborrat/modificació pròpiament dita. Continuarem amb el nostre projecte d'exemple *ContactesWeb* iniciat en documents anteriors, que ara copiarem i canviarem de nom a *ContactesWeb_v2*.

1. Formulari d'inserció

En primer lloc, crearem una vista anomenada `contactes_nou.njk` en la nostra carpeta de `views`. Aquesta vista, com les anteriors, heretarà de `base.njk` i definirà el formulari en el seu bloc de contingut:

```
{% extends "base.njk" %}

{% block titol %}Contactes | Nou{% endblock %}

{% block contingut %}

 <h1>Inserció de nou contacte</h1>

 <form action="/contactes" method="post">
    <div class="form-group">
        <label>Nom:
            <input type="text" class="form-control" name="nom"
                placeholder="Nom del contacte...">
        </label>
    </div>
    <div class="form-group">
        <label>Edat:
            <input type="number" class="form-control" name="edat"
                placeholder="Edat del contacte...">
        </label>
    </div>
    <div class="form-group">
        <label>Telèfon:
           <input type="text" class="form-control" name="telefon"
                placeholder="Telèfon del contacte...">
        </label>
    </div>
    <button type="submit" class="btn btn-primary">
        Enviar
    </button>
 </form>

{% endblock %}
```

Per a mostrar aquesta vista, hi haurà un enllaç "Nou contacte" en el menú de navegació de l'aplicació (vista `menu.njk`), que enviarà a la ruta `/contactes/nou`:

```
<div class="alert alert-secondary">
    <a href="/contactes">Llistat de contactes</a>
      
    <a href="/contactes/nou">Nou contacte</a>
</div>
```

1.1. La ruta per a mostrar el formulari

En segon lloc, definirem una ruta en l'encaminador de contactes (`routes/contactes.js`) que, atenent una petició GET normal a la ruta `/contactes/nou` , renderitzarà la vista anterior:

```
router.get('/nou', (req, res) => {
    res.render('contactes_nou');
})
```

> **NOTA**: aquesta nova ruta haurem de situar-la ABANS de la ruta de fitxa del contacte, ja que, en cas contrari, el patró d'aquesta ruta coincideix amb `/contactes/qualsevol_cosa` , que és el que espera `/contactes/:id` , i en aqueix cas intentarà mostrar la fitxa del contacte. Com a alternativa, podem canviar-la de nom a `/contactes/nou/contacte` perquè no tinga el mateix patró.

1.2. La ruta per a realitzar la inserció

Finalment, el formulari s'enviarà per POST a la ruta `/contactes` . Ens falta definir (o redefinir, perquè ja la teníem d'exemples previs) aquesta ruta perquè reculla les dades de la petició, faça la inserció i, per exemple, renderitze el llistat de contactes com a resultat final, per a poder comprovar que el nou contacte s'ha afegit satisfactòriament. En cas d'error en inserir, podem renderitzar una vista d'error.

Cal tindre en compte, no obstant això, que les dades del formulari no els enviarem en format JSON aquesta vegada. Per a això hauríem d'utilitzar algun mecanisme en el client que, mitjançant JavaScript, construïra la petició amb les dades afegides en format JSON abans d'enviar el formulari, però no el farem. En el seu lloc, utilitzarem el *middleware* incorporat en Express perquè processe la petició també quan les dades arriben des d'un formulari normal. Per a això, a més d'habilitar el processament JSON, habilitem el processament `urlencoded` , d'aquesta manera (en l'arxiu `index.js` , just després o abans d'habilitar el processament JSON):

```
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
...
```

> **NOTA:** el paràmetre `extended` indica si es permet processar dades que proporcionen informació complexa, com a objectes en si mateixos (*true*) o si només es processarà informació simple (*false*).

Ara ja podem afegir/modificar la nostra ruta POST per a inserir contactes, en l'arxiu `routes/contactes.js` :

```
router.post('/', (req, res) => {
    let nouContacte = new Contacte({
        nom: req.body.nom,
        telefon: req.body.telefon,
        edat: req.body.edat
    });
    nouContacte.save().then(resultat => {
        res.redirect(req.baseUrl);
    }).catch(error => {
        res.render('error',
            {error: "Error afegint contacte"});
    });
});
```

El que fem és similar al cas dels serveis REST: recollim les dades del contacte de la petició, creem un de nou, inserim en la base de dades i, si tot ha anat bé (i ací està la diferència amb el servei REST), renderitzem la vista del llistat de contactes (en realitat, redirigim a la ruta que la mostra, perquè carregue les dades del llistat). Si hi ha hagut algun error, renderitzem la vista d'error amb l'error indicat (suposant que tinguem definida alguna vista d'error).

2. Formulari d'esborrat

Anem ara amb l'esborrat. En aquest cas, afegirem un formulari amb un botó "Esborrar" en el llistat de contactes, associat a cada contacte. Aquest botó s'enviarà a la URL `/contactes`, però com els formularis no accepten un mètode DELETE, hem d'afegir algun mecanisme perquè el formulari arribe a la ruta correcta en el servidor.

2.1. Redefinir el mètode DELETE

Igual que en el cas anterior, podríem recórrer a utilitzar JavaScript en el client per a simular una petició AJAX que encapsule les dades necessàries, però per a evitar carregar llibreries addicionals en la part client, instal·larem un mòdul anomenat *method-override*, de NPM, que permet aparellar formularis del client amb mètodes del servidor de manera senzilla. Ho afegim al nostre projecte com qualsevol altre:

```
npm install method-override
```

I ho configurarem perquè, si li arriba en el formulari un camp (normalment ocult) anomenat `_method`, que utilitze aqueix mètode en lloc del propi del formulari. Així, podem emprar aqueix camp ocult per a indicar que en realitat volem fer un DELETE (o un PUT, si fora el cas), i que ometa l'atribut `method` del formulari. El primer que farem serà incloure el mòdul en el servidor principal `index.js`, juntament amb la resta de mòduls:

```
const methodOverride = require('method-override');
```

Després, afegim aquestes línies més a baix, just quan s'afig la resta de middleware. Podem afegir-ho després del middleware d*express*, per exemple, però és important definir-lo abans de carregar els encaminadors:

```
app.use(methodOverride(function (req, res) {
    if (req.body && typeof req.body === 'object' && '_method' in req.body) {
        let method = req.body._method;
        delete req.body._method;
        return method;
    }
}));
```

2.2. El formulari d'esborrat

Ara, en la vista de `contactes_llistat.njk`, definim un xicotet formulari al costat de cada contacte, amb un botó per a esborrar-lo a partir del seu *id*. En aquest formulari, incloem un camp *hidden* (ocult) amb el nom del qual siga `_method`, on indicarem que l'operació que volem realitzar en el servidor és DELETE:

```
<ul>
    {% for contacte in contactes %}
        <li>{{ contacte.nom }}
            <a href="/contactes/{{ contacto.id }}">Fitxa</a>
            <form action="/contactes/{{ contacto.id }}" method="post">
                <input type="hidden" name="_method" value="delete">
                <button type="submit" class="btn btn-danger">
                    Esborrar
                </button>
            </form>
        </li>
    {% endfor %}
</ul>
```

2.3. La ruta d'esborrat

Finalment, redefinim la ruta per a l'esborrat. El que fem és eliminar el contacte pel seu *id*, i redirigir al llistat de contactes si tot ha anat bé, o mostrar la vista d'error si no.

```
router.delete('/:id', (req, res) => {
    Contacte.findByIdAndRemove(req.params.id).then(resultat => {
        res.redirect(req.baseUrl);
    }).catch(error => {
        res.render('error', {error: "Error esborrant contacte"});
    });
});
```

3. Formulari d'actualització

Per a fer una actualització hem de combinar passos que hem seguit prèviament en la inserció i en l'esborrat:

1. Definirem el formulari d'actualització, de manera que li passarem com a paràmetre a la vista l'objecte que volem modificar, per a emplenar els camps del formulari amb aquest objecte.
2. També afegirem una nova ruta GET en l'encaminador per a renderitzar aquest formulari. Per exemple, `/contactes/editar`.
3. El formulari haurà d'enviar-se per PUT a la ruta corresponent del seu encaminador. Per a això, utilitzarem de nou el camp ocult `_method` per a indicar que volem fer *PUT*
4. En la ruta `put` de l'encaminador, recollim les dades del formulari, fem la corresponent actualització i redirigim on vulguem (llistat general o pàgina d'error, per exemple).

3.1. Formulari i ruta d'edició

Seguint aquests passos anteriors, el nostre formulari d'edició podríem definir-lo en un arxiu `contactes_editar.njk`, amb el següent aspecte:

```
{% extends "base.njk" %}

{% block titol %}Contactes | Edició{% endblock %}

{% block contingut %}

<h1>Edició de contacte</h1>

 <form action="/contactes/{{ contacto.id }}" method="post">
    <input type="hidden" name="_method" value="put">
    <div class="form-group">
        <label>Nom:
            <input type="text" class="form-control" name="nom"
                placeholder="Nom del contacte..."
                value="{{ contacte.nom }}">
        </label>
    </div>
    <div class="form-group">
        <label>Edat:
            <input type="number" class="form-control" name="Edat"
                placeholder="Edat del contacte..."
                value="{{ contacte.edat }}">
        </label>
    </div>
    <div class="form-group">
        <label>Telèfon:
            <input type="text" class="form-control" name="telefono"
                placeholder="Telèfon del contacte..."
                value="{{ contacte.telefono }}">
        </label>
    </div>
    <button type="submit" class="btn btn-primary">
        Enviar
    </button>
</form>

{% endblock %}
```

A més a més, afegiríem aquesta nova ruta en el controlador de contactes per a renderitzar el formulari:

```
router.get('/editar/:id', (req, res) => {
    Contacte.findById(req.params['id']).then(resultat => {
        if (resultat) {
            res.render('contactes_editar', {contacte: resultat});
        } else {
            res.render('error', {error: "Contacte no trobat"});
        }
    }).catch(error => {
        res.render('error', {error: "Contacte no trobat"});
    });
});
```

3.2. Actualització de dades del contacte

Finalment, la ruta *put* recollirà les dades de la petició i actualitzarà el contacte:

```
router.put('/:id', (req, res) => {
    Contacte.findByIdAndUpdate(req.params.id, {
        $set: {
            nom: req.body.nom,
            edat: req.body.edat,
            telefon: req.body.telefon
       }
    }, {new: true}).then(resultat => {
        res.redirect(req.baseUrl);
    }).catch(error => {
        res.render('error', {error: "Error modificant contacte"});
    });
});
```

Exercici 1:

Crea una còpia de l'exercici *LlibresWeb* de sessions anteriors en una altra anomenada **LlibresWeb_v2**. Aplicarem en aquesta nova versió els següents canvis.

Primer implementarem la inserció de nous llibres. Per a això:

- Defineix una vista anomenada `llibres_nou.njk` en la carpeta `views`, amb un formulari que permeta emplenar les dades d'un nou llibre. Pots basar-te en la vista feta per a inserir nous contactes, i modificar els camps del formulari perquè siguen els del llibre. Fes que el formulari s'envie per POST a `/llibres`.
- Defineix una ruta en `routes/llibres.js` que responga a `/llibres/nou` per GET, i renderitze la vista `llibres_nou` creada en el pas anterior.

- Defineix una altra ruta en `routes/llibres.js` que, amb POST, responga a la URL `/llibres`, recollint les dades del llibre de la petició (recorda configurar *express* com `urlencoded`), done d'alta el nou llibre i redirigisca al llistat de llibres, o a una pàgina d'error, segons siga el cas.

A continuació implementarem l'esborrat de llibres, seguint aquests passos:

- Començarem per instal·lar la llibreria `method-override` com hem fet en l'exemple dels contactes, i la incorporem a l'arxiu principal `index.js`
- Després afegim el mateix *middleware* que en el cas dels contactes perquè busque un camp `_method` en el cos de la petició, i l'use en lloc del *method* que puga tindre el formulari. Simplement, còpia i pega aqueixa funció de l'exemple dels contactes en l'arxiu principal d'aquesta aplicació de llibres, en el lloc indicat.
- A continuació, haurem d'editar la vista de `llibres_llistat.njk` i afegir un formulari d'esborrat al costat de cada llibre, perquè s'envie per DELETE a la ruta `/llibres`, com en l'exemple dels contactes.
- Finalment, definim la ruta en `routes/llibres.js` per a respondre a aquesta crida, eliminar el llibre i redirigir al llistat de llibres, o a la vista d'error, segons el resultat de l'operació.

Ací teniu una captura de pantalla de com podria quedar la vista del llistat de llibres, amb el nou formulari per a esborrar cada llibre:

NOTA: L'estil de la barra de menú superior pot ser el que vulgues, no ha de ser necessàriament com en la imatge. També l'estil dels items del llistat, o del formulari, poden variar segons els teus propis gustos.

Finalment implementarem l'edició de llibres. Haurà d'haver-hi:

- Un enllaç/botó en el llistat de llibres que mostre el formulari del llibre, amb els camps ja farcits. Pots crear el formulari en la vista `llibres_editar.njk`.
- Aquest formulari hauria d'enviar-se per PUT a la ruta de modificació de llibres
- En aquesta ruta, es modificaran les dades del llibre que es reba, i es redirigirà al llistat de llibres, o a la vista d'error.

4. Pujar fitxers en el formulari

Per a pujar fitxers en un formulari, necessitem que el tipus d'aquest formulari siga `multipart/form-data`. Dins, hi haurà un o diversos camps de tipus `file` amb els arxius que l'usuari triarà per a pujar:

```
<form action="..." method="post" enctype="multipart/form-data">
    ...
    <input type="file" class="form-control" name="imatge">
</form>
```

Per a poder processar aquest tipus de formularis, necessitarem alguna llibreria addicional. Utilitzarem una anomenada `multer`, que instal·larem en el nostre projecte al costat de la resta:

```
npm install multer
```

Ara, en els fitxers on anem a necessitar la pujada d'arxius necessitem incloure aquesta llibreria, i configurar els paràmetres de pujada i emmagatzematge:

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

...

let storage = multer.diskStorage({
    destination: function (req, file, cb) {
        cb(null, 'public/uploads')
    },
    filename: function (req, file, cb) {
        cb(null, Date.now() + "_" + file.originalname)
    }
});

let upload = multer({storage: storage});
```

L'element `storage` defineix, en primer lloc, quin serà la carpeta on es pugen els arxius (en el nostre exemple serà `public/uploads`), i després, quin nom assignarem als arxius quan els pugem. L'atribut `originalname` de l'objecte `file` que es rep conté el nom original de l'arxiu en el client, però per a evitar sobreescrituras, li concatenarem com a prefix la data o *timestamp* actual amb `Date.now()`. Aquest últim pas no és obligatori si no ens importa sobreescriure arxius existents.

Finalment, ens queda utilitzar el middleware `upload` que hem configurat abans en els mètodes o serveis que ho necessiten. Si, per exemple, en un servei POST esperem rebre un arxiu en un camp `file` anomenat `imatge`, podem fer que automàticament es puge a la carpeta especificada abans, amb el nom assignat en la configuració, simplement aplicant aquest *middleware* en el servei:

```
router.post('/', upload.single('imatge'), (req, res) => {
 // Ací ja estarà l'arxiu pujat
 // Amb req.file.filename obtenim el nom actual
 // Amb req.body.XXX obtenim la resta de camps
});
```

4.1. Pujada de fitxers i *method-override*

En exemples anteriors hem utilitzat el *middleware method-override* per a substituir comandes HTTP en una petició, i així poder usar les comandes PUT o DELETE encara que el formulari siga POST. No obstant això, cal tindre en compte que, quan el formulari puja fitxers i és *multipart/form-data*, el processament que s'aconsegueix amb `express.urlencoded` no és suficient, i no és capaç d'identificar les parts del cos del formulari. Dit d'una altra manera, si tenim un formulari com aquest, Express no serà capaç de reemplaçar el mètode POST per PUT amb *method-override*:

```
<form action="..." method="post" enctype="multipart/form-data">
 <input type="hidden" name="_method" value="put">
 ...
</form>
```

Per a solucionar aquest problema podem utilitzar algun *middleware* addicional, com per exemple `busboy-body-parser`, però té alguns problemes d'incompatibilitat amb uns altres *middlewares* que puguem utilitzar, com ara *multer*. Alternativament, una solució més senzilla pot ser reemplaçar el servei PUT del nostre encaminador per un altre POST al qual li passem l'*id* de l'element a editar

```
// Modificar contactes
// Ho definim com a "POST" per a integrar-ho millor en un formulari multipart
router.post('/:id', (req, res) => {
    // El codi intern del servei no canvia
});
```

> **Exercici 2:**
>
> Sobre l'exercici anterior crea una còpia anomenada **LlibresWeb_v3**. Afegirem la possibilitat de pujar portades de llibres. Podem seguir aquests passos:
>
> - Afegir en l'esquema de la col·lecció de llibres una nova propietat anomenada *portada*, de tipus text, que admetrà nuls per a respectar els llibres que no tinguen portada inserits fins ara.
> - En el formulari d'inserció i edició de llibres afegirem aqueix nou camp (tipus *file*) per a poder pujar imatges de llibres. Recorda afegir *enctype="multipart/form-data"* en la definició del formulari

- Instal·la i configura *multer* per a pujar fitxers a la subcarpeta *public/uploads*, amb el mateix nom que la imatge original
- Actualitza els serveis POST i PUT de llibres perquè pugen la imatge del llibre i actualitzen les dades corresponents
- Actualitza també la fitxa del llibre perquè es veja la portada

5. Validació de formularis

Una tasca important quan estem enviant formularis en una aplicació web és la validació d'aquests. Aquesta validació podem fer-la en la part del client utilitzant mecanismes de validació d'HTML5 i JavaScript, i també en la part del servidor, comprovant que les dades que arriben en la petició tenen els valors adequats. Aquesta última part és important fer-la, independentment que les dades es validen (també) en el client, com a pas previ a la seua possible inserció en una base de dades.

Tenim diferents alternatives per a realitzar aquesta validació de dades en el costat del servidor:

- Utilitzar alguna llibreria bàsica com validator.js, que permet comprovar i corregir cadenes de text.
- Emprar una llibreria una mica més avançada i específica per a aplicacions web, com express-validator.
- Utilitzar els propis mecanismes de validació que ens ofereixen els esquemes de Mongoose

Veurem a continuació algunes pinzellades de les dues primeres opcions, sense entrar en massa detalls en aquest curs, i ens centrarem més en els mecanismes de validació que tenim disponibles en Mongoose.

5.1. Validació de textos amb *validator.js*

La llibreria `validator.js` permet comprovar la validesa de cadenes de text, i sanejar-les perquè tinguen un contingut adequat. Per *sanejar* entenem operacions de neteja de textos, com eliminar espais a l'inici o al final (*trim*), escapar caràcters, etc.

Cal tindre en compte que les dades que rebem en una petició, fins que es processen i emmagatzemen en una base de dades, són textos. Per tant, podem emprar aquesta llibreria per a analitzar-los abans de fer l'operació indicada. Com a primer pas haurem d'instal·lar la llibreria en la nostra aplicació, amb la comanda `npm install validator`. Després la incorporem en el nostre projecte i podem, per exemple, utilitzar-la en els diferents serveis (POST, PUT, etc) on siga necessari validar dades d'entrada. Ací veiem un exemple bàsic on comprovem que el nom d'un contacte existeix i té almenys 3 caràcters, i el telèfon és una dada numèrica de 9 caràcters:

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

encaminador.post('/', (req, res) => {
    let nom = req.body.nom;
    let telefon = req.body.telefon;

    if(validator.isEmpty(nom) ||
    !validator.isLength(nom, {min: 3}))
    {
        // Redirigir a pàgina d'error pel nom
    }
    else if (!validator.isNumeric(telefon) ||
    !validator.isLength(telefon, {min: 9, max: 9}))
    {
        // Redirigir a pàgina d'error pel telèfon
    }
    else
    {
        // Correcte, fer la inserció
    }
});
```

- Ací podem consultar un llistat de validadors disponibles en *validator.js*
- Ací tenim alguns exemples de saneadores en aqueixa llibreria

5.2. Validació de peticions amb *express-validator*

A l'hora de validar les dades d'una petició en Node i Express podem emprar algunes llibreries addicionals que ens ajuden. Una de les més populars és `express-validator`, la web oficial de les quals podem consultar ací. Com la pròpia documentació explica, es tracta d'una llibreria que aglutina i amplia algunes opcions oferides per la llibreria *validator.js* vista abans.

Per a començar, haurem d'instal·lar la llibreria amb el corresponent comando `npm install express-validator` (no és necessari tindre instal·lada també la llibreria *validator.js* anterior). Una vegada instal·lada, tenim a la nostra disposició un ampli ventall de validadors i sanejadors (*sanitizers*), molts d'ells incorporats de *validator.js*.

Per a poder utilitzar els validadors, hem d'accedir als paràmetres de la petició, bé des del cos de la petició (propietat `body`) o bé a través de la *query string* (propietat `query`). Per exemple, així podem comprovar si el nom d'un contacte no està buit abans de fer una inserció:

```
const { body } = require('express-validator');

...

encaminador.post('/', body('nom').notEmpty(), (req, res) => {
    // Codi habitual d'inserció de contacte
});
```

Com podem comprovar, apliquem un *middleware* en el servei POST que accedeix amb `body('nom')` al valor enviat per al nom i comprova amb el validador `notEmpty` que no estiga buit. Si la validació es compleix es continua amb el procés, i si no ho fa s'interromp la petició. Si volguérem, per exemple, netejar espais en blanc en el nom una vegada sapiem que és vàlid, n'hi ha prou amb afegir el saneador `trim` després de la validació:

```
const { body } = require('express-validator');

...

encaminador.post('/', body('nom').notEmpty().trim(), (req, res) => {
    // Codi habitual d'inserció de contacte
});
```

Aquesta característica es denomina encadenament de validacions (*validation chain*) i permet enllaçar diverses comprovacions per a una mateixa dada en el procés. Podeu consultar més opcions d'ús d'aquesta llibreria en la seua web oficial.

5.3. Validació de peticions amb els esquemes de Mongoose

En sessions anteriors ja hem vist que podem configurar els esquemes de Mongoose per a definir algunes opcions de validació en els documents de les nostres col·leccions en MongoDB. Aquesta característica ja impedirà que s'afigen dades amb valors incorrectes, com per exemple contactes amb edats massa grans, o llibres amb preus negatius. Però, a més, podem indicar en aquests esquemes quins missatges d'error volem produir en el cas que alguna validació no siga correcta, per a després recollir aqueixos missatges en intentar fer la inserció/modificació.

Modificarem el nostre exemple de *ContactesWeb_v2*, en una altra carpeta anomenada *ContactesWeb_v3*. Editem el fitxer *models/contacte.js* i ho deixem d'aquesta manera:

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

// Definició de l'esquema de la nostra col·lecció
let contacteSchema = new mongoose.Schema({
    nom: {
        type: String,
        required: [true, 'El nom del contacte és obligatori'],
        minlength: [3, 'El nom del contacte és massa curt'],
        trim: true
    },
    telefon: {
        type: String,
        required: [true, 'El número de telèfon és obligatori'],
        unique: true,
        trim: true,
        match: [/^\d{9}$/, 'El telèfon ha de constar de 9 dígits']
    },
    edat: {
        type: Number,
        min: [18, 'L\'edat mínima ha de ser 18'],
        max: [120, 'L\'edat màxima ha de ser 120']
    }
});

// Associació amb el model (col·lecció contactes)
let Contacte = mongoose.model('contacte', contactoSchema);

module.exports = Contacte;
```

Notar com, en cada validador, afegim un array amb el valor que ha de tindre i el missatge d'error que mostrar en cas que no es complisca. Notar també que la propietat `unique` no és un validador com a tal, per la qual cosa no hauríem d'usar-la per a configurar un missatge d'error personalitzat.

Ara podem modificar els serveis d'inserció i/o esborrat perquè renderitzen una vista d'error amb el/els missatge(s) d'error produït(s). Així podria quedar la inserció:

```
// Ruta per a inserir contactes
router.post('/', (req, res) => {

    let nouContacte = new Contacte({
        nom: req.body.nom,
        telefon: req.body.telefon,
        edat: req.body.edat
    });
    nouContacte.save().then(resultat => {
        res.redirect(req.baseUrl);
    }).catch(error => {
        let errors = Object.keys(error.errors);
        let missatge = "";
        if(errors.length > 0)
        {
            errors.forEach(clau => {
                missatge += '<p>' + error.errors[clau].message + '</p>';
            });
        }
        else
        {
            missatge = 'Error afegint contacte';
        }
        res.render('error', {error: missatge});
    });
});
```

L'element `error.errors` és un objecte que conté detalls sobre els errors de validació que van ocórrer durant l'operació de guardat amb `save()`. En Mongoose, cada clau en *error.errors* correspon a un camp del model que no va passar la validació. D'altra banda, `Object.keys(error.errors)` retorna un array de les claus (noms de camp) presents en *error.errors*. En síntesi, en este cas, en la clàusula `catch` recorrem els errors produïts, si n'hi ha, dins del camp `errors` de l'error retornat. Anem construint amb ells una resposta que acumule els errors detectats, o vam mostrar un error genèric en el cas que no hàgem detectat cap error concret.

Cal tindre en compte també que, tal com tenim configurat Nunjucks, els paràgrafs d'errors que estem definint no es mostraran com a paràgrafs en la vista d'error, perquè hem dit que acte-escape els textos. Per tant, si posem un nom de menys de tres caràcters la vista d'error mostraria una cosa així:

```
<p>El nom del contacte és massa curt</p>
```

Per a llevar les marques HTML hem de dir-li a Nunjucks que no escape el contingut d'aqueixa dada en concret, a través del modificador `safe`. Editem la vista d'error per a deixar-la així:

```
<html>
    <head>
        <link rel="stylesheet" href="/css/bootstrap.min.css"/>
        <link rel="stylesheet" href="/public/css/estils.css"/>
    </head>
    <body>
        <div class="contenidor">
            <h1>Error</h1>
            <div class="alert alert-danger">
                {% if error %}
                    {{ error|safe }}
                {% else %}
                    Error en l'aplicació
                {% endif %}
            </div>
        </div>
    </body>
</html>
```

Exercici 3:

Sobre l'exercici anterior de llibres crea una còpia anomenada **LlibresWeb_v4**. Defineix missatges de validació personalitzats en l'esquema del llibre i fes que es mostre cadascun damunt del camp del formulari afectat, d'aquesta manera: