Gestión de formularios

Vamos ahora a añadir formularios a nuestra aplicación que nos permitirán insertar, borrar o modificar contenido de la base de datos. También modificaremos las correspondientes rutas para, por un lado, mostrar estos formularios, y por otro, recoger la petición y hacer la inserción/borrado/modificación propiamente dicha. Continuaremos con nuestro proyecto de ejemplo *ContactosWeb* iniciado en documentos anteriores, que ahora copiaremos y renombraremos a *ContactosWeb_v2*.

1. Formulario de inserción

En primer lugar, crearemos una vista llamada `contactos_nuevo.njk` en nuestra carpeta de `views`. Esta vista, como las anteriores, heredará de `base.njk` y definirá el formulario en su bloque de contenido:

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

{% block titulo %}Contactos | Nuevo{% endblock %}

{% block contenido %}

    <h1>Inserción de nuevo contacto</h1>

    <form action="/contactos" method="post">
        <div class="form-group">
            <label>Nombre:
                <input type="text" class="form-control" name="nombre"
                placeholder="Nombre del contacto...">
            </label>
        </div>
        <div class="form-group">
            <label>Edad:
                <input type="number" class="form-control" name="Edad"
                placeholder="Edad del contacto...">
            </label>
        </div>
        <div class="form-group">
            <label>Teléfono:
                <input type="text" class="form-control" name="telefono"
                placeholder="Teléfono del contacto...">
            </label>
        </div>
        <button type="submit" class="btn btn-primary">
            Enviar
        </button>
    </form>

{% endblock %}
```

Para mostrar esta vista, habrá un enlace "Nuevo contacto" en el menú de navegación de la aplicación (vista `menu.njk`), que enviará a la ruta `/contactos/nuevo`:

```
<div class="alert alert-secondary">
    <a href="/contactos">Listado de contactos</a>
      
    <a href="/contactos/nuevo">Nuevo contacto</a>
</div>
```

1.1. La ruta para mostrar el formulario

En segundo lugar, vamos a definir una ruta en el enrutador de contactos (`routes/contactos.js`) que, atendiendo una petición GET normal a la ruta `/contactos/nuevo` , renderizará la vista anterior:

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

> **NOTA**: esta nueva ruta deberemos ubicarla ANTES de la ruta de ficha del contacto, ya que, de lo contrario, el patrón de esta ruta coincide con `/contactos/loquesea` , que es lo que espera `/contactos/:id` , y en ese caso intentará mostrar la ficha del contacto. Como alternativa, podemos renombrarla a `/contactos/nuevo/contacto` para que no tenga el mismo patrón.

1.2. La ruta para realizar la inserción

Finalmente, el formulario se enviará por POST a la ruta `/contactos` . Nos falta definir (o redefinir, porque ya la teníamos de ejemplos previos) esta ruta para que recoja los datos de la petición, haga la inserción y, por ejemplo, renderice el listado de contactos como resultado final, para poder comprobar que el nuevo contacto se ha añadido satisfactoriamente. En caso de error al insertar, podemos renderizar una vista de error.

Hay que tener en cuenta, no obstante, que los datos del formulario no los vamos a enviar en formato JSON esta vez. Para ello tendríamos que utilizar algún mecanismo en el cliente que, mediante JavaScript, construyera la petición con los datos añadidos en formato JSON antes de enviar el formulario, pero no lo vamos a hacer. En su lugar, vamos a utilizar el *middleware* incorporado en Express para que procese la petición también cuando los datos lleguen desde un formulario normal. Para ello, además de habilitar el procesado JSON, habilitamos el procesado `urlencoded` , de esta forma (en el archivo `index.js` , justo después o antes de habilitar el procesado JSON):

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

> **NOTA:** el parámetro `extended` indica si se permite procesar datos que proporcionen información compleja, como objetos en sí mismos (*true*) o si sólo se procesará información simple (*false*).

Ahora ya podemos añadir/modificar nuestra ruta POST para insertar contactos, en el archivo `routes/contactos.js` :

```
router.post('/', (req, res) => {
    let nuevoContacto = new Contacto({
        nombre: req.body.nombre,
        telefono: req.body.telefono,
        edad: req.body.edad
    });
    nuevoContacto.save().then(resultado => {
        res.redirect(req.baseUrl);
    }).catch(error => {
        res.render('error',
            {error: "Error añadiendo contacto"});
    });
});
```

Lo que hacemos es similar al caso de los servicios REST: recogemos los datos del contacto de la petición, creamos uno nuevo, insertamos en la base de datos y, si todo ha ido bien (y aquí está la diferencia con el servicio REST), renderizamos la vista del listado de contactos (en realidad, redirigimos a la ruta que la muestra, para que cargue los datos del listado). Si ha habido algún error, renderizamos la vista de error con el error indicado (suponiendo que tengamos definida alguna vista de error).

2. Formulario de borrado

Vamos ahora con el borrado. En este caso, añadiremos un formulario con un botón de "Borrar" en el listado de contactos, asociado a cada contacto. Dicho botón se enviará a la URL `/contactos`, pero como los formularios no aceptan un método DELETE, tenemos que añadir algún mecanismo para que el formulario llegue a la ruta correcta en el servidor.

2.1. Redefinir el método DELETE

Igual que en el caso anterior, podríamos recurrir a utilizar JavaScript en el cliente para simular una petición AJAX que encapsule los datos necesarios, pero para evitar cargar librerías adicionales en la parte cliente, vamos a instalar un módulo llamado *method-override*, de NPM, que permite emparejar formularios del cliente con métodos del servidor de forma sencilla. Lo añadimos a nuestro proyecto como cualquier otro:

```
npm install method-override
```

Y vamos a configurarlo para que, si le llega en el formulario un campo (normalmente oculto) llamado `_method`, que utilice ese método en lugar del propio del formulario. Así, podemos emplear ese campo oculto para indicar que en realidad queremos hacer un DELETE (o un PUT, si fuera el caso), y que omita el atributo `method` del formulario. Lo primero que haremos será incluir el módulo en el servidor principal `index.js`, junto con el resto de módulos:

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

Después, añadimos estas líneas más abajo, justo cuando se añade el resto de middleware. Podemos añadirlo tras el middleware de *express*, por ejemplo, pero es importante definirlo antes de cargar los enrutadores:

```
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 formulario de borrado

Ahora, en la vista de `contactos_listado.njk`, definimos un pequeño formulario junto a cada contacto, con un botón para borrarlo a partir de su *id*. En dicho formulario, incluimos un campo *hidden* (oculto) cuyo nombre sea `_method`, donde indicaremos que la operación que queremos realizar en el servidor es DELETE:

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

2.3. La ruta de borrado

Finalmente, redefinimos la ruta para el borrado. Lo que hacemos es eliminar el contacto cuyo ID nos llega en la URL, y redirigir al listado de contactos si todo ha ido bien, o mostrar la vista de error si no.

```
router.delete('/:id', (req, res) => {
    Contacto.findByIdAndRemove(req.params.id).then(resultado => {
        res.redirect(req.baseUrl);
    }).catch(error => {
        res.render('error', {error: "Error borrando contacto"});
    });
});
```

3. Formulario de actualización

Para hacer una actualización debemos combinar pasos que hemos seguido previamente en la inserción y en el borrado:

1. Definiremos el formulario de actualización, de forma que le pasaremos como parámetro a la vista el objeto que queremos modificar, para rellenar los campos del formulario con dicho objeto.
2. También añadiremos una nueva ruta GET en el enrutador para renderizar este formulario. Por ejemplo, `/contactos/editar`.
3. El formulario deberá enviarse por PUT a la ruta correspondiente de su enrutador. Para ello, utilizaremos de nuevo el campo oculto `_method` para indicar que queremos hacer *PUT*
4. En la ruta `put` del enrutador, recogemos los datos del formulario, hacemos la correspondiente actualización y redirigimos donde queramos (listado general o página de error, por ejemplo).

3.1. Formulario y ruta de edición

Siguiendo estos pasos anteriores, nuestro formulario de edición podríamos definirlo en un archivo `contactos_editar.njk`, con el siguiente aspecto:

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

{% block titulo %}Contactos | Edición{% endblock %}

{% block contenido %}

    <h1>Edición de contacto</h1>

    <form action="/contactos/{{ contacto.id }}" method="post">
        <input type="hidden" name="_method" value="put">
        <div class="form-group">
            <label>Nombre:
                <input type="text" class="form-control" name="nombre"
                placeholder="Nombre del contacto..."
                value="{{ contacto.nombre }}">
            </label>
        </div>
        <div class="form-group">
            <label>Edad:
                <input type="number" class="form-control" name="Edad"
                placeholder="Edad del contacto..."
                value="{{ contacto.edad }}">
            </label>
        </div>
        <div class="form-group">
            <label>Teléfono:
                <input type="text" class="form-control" name="telefono"
                placeholder="Teléfono del contacto..."
                value="{{ contacto.telefono }}">
            </label>
        </div>
        <button type="submit" class="btn btn-primary">
            Enviar
        </button>
    </form>

{% endblock %}
```

Por su parte, añadiríamos esta nueva ruta en el controlador de contactos para renderizar el formulario:

```
router.get('/editar/:id', (req, res) => {
    Contacto.findById(req.params['id']).then(resultado => {
        if (resultado) {
            res.render('contactos_editar', {contacto: resultado});
        } else {
            res.render('error', {error: "Contacto no encontrado"});
        }
    }).catch(error => {
        res.render('error', {error: "Contacto no encontrado"});
    });
});
```

3.2. Actualización de datos del contacto

Finalmente, la ruta *put* recogerá los datos de la petición y actualizará el contacto:

```
router.put('/:id', (req, res) => {
    Contacto.findByIdAndUpdate(req.params.id, {
        $set: {
            nombre: req.body.nombre,
            edad: req.body.edad,
            telefono: req.body.telefono
        }
    }, {new: true}).then(resultado => {
        res.redirect(req.baseUrl);
    }).catch(error => {
        res.render('error', {error: "Error modificando contacto"});
    });
});
```

Ejercicio 1:

Crea una copia del ejercicio *LibrosWeb* de sesiones anteriores en otra llamada **LibrosWeb_v2**. Aplicaremos en esta nueva versión los siguientes cambios.

Primero implementaremos la inserción de nuevos libros. Para ello:

- Define una vista llamada `libros_nuevo.njk` en la carpeta `views`, con un formulario que permita rellenar los datos de un nuevo libro. Puedes basarte en la vista hecha para insertar nuevos contactos, y modificar los campos del formulario para que sean los del libro. Haz que el formulario se envíe por POST a `/libros`.
- Define una ruta en `routes/libros.js` que responda a `/libros/nuevo` por GET, y renderice la vista `libros_nuevo` creada en el paso anterior.

- Define otra ruta en `routes/libros.js` que, con POST, responda a la URL `/libros`, recogiendo los datos del libro de la petición (recuerda configurar *express* como `urlencoded`), dé de alta el nuevo libro y redirija al listado de libros, o a una página de error, según sea el caso.

A continuación implementaremos el borrado de libros, siguiendo estos pasos:

- Comenzaremos por instalar la librería `method-override` como hemos hecho en el ejemplo de los contactos, y la incorporamos al archivo principal `index.js`
- Después añadimos el mismo *middleware* que en el caso de los contactos para que busque un campo `_method` en el cuerpo de la petición, y lo use en lugar del *method* que pueda tener el formulario. Simplemente, copia y pega esa función del ejemplo de los contactos en el archivo principal de esta aplicación de libros, en el lugar indicado.
- A continuación, deberemos editar la vista de `libros_listado.njk` y añadir un formulario de borrado junto a cada libro, para que se envíe por DELETE a la ruta `/libros`, como en el ejemplo de los contactos.
- Finalmente, definimos la ruta en `routes/libros.js` para responder a esta llamada, eliminar el libro y redirigir al listado de libros, o a la vista de error, según el resultado de la operación.

Aquí tenéis una captura de pantalla de cómo podría quedar la vista del listado de libros, con el nuevo formulario para borrar cada libro:

NOTA: El estilo de la barra de menú superior puede ser el que quieras, no tiene que ser necesariamente como en la imagen. También el estilo de los items del listado, o del formulario, pueden variar según tus propios gustos.

Finalmente implementaremos la edición de libros. Deberá haber:

- Un enlace/botón en el listado de libros que muestre el formulario del libro, con los campos ya rellenos. Puedes crear el formulario en la vista `libros_editar.njk`.
- Dicho formulario debería enviarse por PUT a la ruta de modificación de libros
- En dicha ruta, se modificarán los datos del libro que se reciba, y se redirigirá al listado de libros, o a la vista de error.

4. Subir ficheros en el formulario

Para subir ficheros en un formulario, necesitamos que el tipo de dicho formulario sea `multipart/form-data`. Dentro, habrá uno o varios campos de tipo `file` con los archivos que el usuario elegirá para subir:

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

Para poder procesar este tipo de formularios, necesitaremos alguna librería adicional. Vamos a utilizar una llamada `multer`, que instalaremos en nuestro proyecto junto al resto:

```
npm install multer
```

Ahora, en los ficheros donde vayamos a necesitar la subida de archivos necesitamos incluir esta librería, y configurar los parámetros de subida y almacenamiento:

```
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});
```

El elemento `storage` define, en primer lugar, cuál va a ser la carpeta donde se suban los archivos (en nuestro ejemplo será `public/uploads`), y después, qué nombre asignaremos a los archivos cuando los subamos. El atributo `originalname` del objeto `file` que se recibe contiene el nombre original del archivo en el cliente, pero para evitar sobreescrituras, le vamos a concatenar como prefijo la fecha o *timestamp* actual con `Date.now()`. Este último paso no es obligatorio si no nos importa sobreescribir archivos existentes.

Finalmente, nos queda utilizar el middleware `upload` que hemos configurado antes en los métodos o servicios que lo necesiten. Si, por ejemplo, en un servicio POST esperamos recibir un archivo en un campo `file` llamado `imagen`, podemos hacer que automáticamente se suba a la carpeta especificada antes, con el nombre asignado en la configuración, simplemente aplicando este *middleware* en el servicio:

```
router.post('/', upload.single('imagen'), (req, res) => {
    // Aquí ya estará el archivo subido
    // Con req.file.filename obtenemos el nombre actual
    // Con req.body.XXX obtenemos el resto de campos
});
```

4.1. Subida de ficheros y *method-override*

En ejemplos anteriores hemos utilizado el *middleware method-override* para sustituir comandos HTTP en una petición, y así poder usar los comandos PUT o DELETE aunque el formulario sea POST. Sin embargo, hay que tener en cuenta que, cuando el formulario sube ficheros y es *multipart/form-data*, el procesamiento que se consigue con `express.urlencoded` no es suficiente, y no es capaz de identificar las partes del cuerpo del formulario. Dicho de otro modo, si tenemos un formulario como éste, Express no va a ser capaz de reemplazar el método POST por PUT con *method-override*:

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

Para solucionar este problema podemos utilizar algún *middleware* adicional, como por ejemplo `busboy-body-parser`, pero tiene algunos problemas de incompatibilidad con otros *middlewares* que podamos utilizar, como *multer*. Alternativamente, una solución más sencilla puede ser reemplazar el servicio PUT de nuestro enrutador por otro POST al que le pasemos la *id* del elemento a editar

```
// Modificar contactos
// Lo definimos como "POST" para integrarlo mejor en un formulario multipart
router.post('/:id', (req, res) => {
    // El código interno del servicio no cambia
});
```

> **Ejercicio 2:**
>
> Sobre el ejercicio anterior crea una copia llamada **LibrosWeb_v3**. Vamos a añadir la posibilidad de subir portadas de libros. Podemos seguir estos pasos:
>
> - Añadir en el esquema de la colección de libros una nueva propiedad llamada *portada*, de tipo texto, que admitirá nulos para respetar los libros que no tengan portada insertados hasta ahora.
> - En el formulario de inserción y edición de libros añadiremos ese nuevo campo (tipo *file*) para poder subir imágenes de libros. Recuerda añadir *enctype="multipart/form-data"* en la definición del formulario

> - Instala y configura *multer* para subir ficheros a la subcarpeta *public/uploads*, con el mismo nombre que la imagen original
> - Actualiza los servicios POST y PUT de libros para que suban la imagen del libro y actualicen los datos correspondientes
> - Actualiza también la ficha del libro para que se vea la portada

5. Validación de formularios

Una tarea importante cuando estamos enviando formularios en una aplicación web es la validación de los mismos. Esta validación podemos hacerla en la parte del cliente utilizando mecanismos de validación de HTML5 y JavaScript, y también en la parte del servidor, comprobando que los datos que llegan en la petición tienen los valores adecuados. Esta última parte es importante hacerla, independientemente de que los datos se validen (también) en el cliente, como paso previo a su posible inserción en una base de datos.

Tenemos diferentes alternativas para realizar esta validación de datos en el lado del servidor:

- Utilizar alguna librería básica como validator.js, que permite comprobar y corregir cadenas de texto.
- Emplear una librería algo más avanzada y específica para aplicaciones web, como express-validator.
- Utilizar los propios mecanismos de validación que nos ofrecen los esquemas de Mongoose

Veremos a continuación algunas pinceladas de las dos primeras opciones, sin entrar en demasiados detalles en este curso, y nos centraremos más en los mecanismos de validación que tenemos disponibles en Mongoose.

5.1. Validación de textos con *validator.js*

La librería `validator.js` permite comprobar la validez de cadenas de texto, y sanearlas para que tengan un contenido adecuado. Por *sanear* entendemos operaciones de limpieza de textos, como eliminar espacios al inicio o al final (*trim*), escapar caracteres, etc.

Hay que tener en cuenta que los datos que recibimos en una petición, hasta que se procesan y almacenan en una base de datos, son textos. Por tanto, podemos emplear esta librería para analizarlos antes de hacer la operación indicada. Como primer paso deberemos instalar la librería en nuestra aplicación, con el comando `npm install validator`. Después la incorporamos en nuestro proyecto y podemos, por ejemplo, utilizarla en los diferentes servicios (POST, PUT, etc) donde sea necesario validar datos de entrada. Aquí vemos un ejemplo básico donde comprobamos que el nombre de un contacto existe y tiene al menos 3 caracteres, y el teléfono es un dato numérico de 9 caracteres:

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

router.post('/', (req, res) => {
    let nombre = req.body.nombre;
    let telefono = req.body.telefono;

    if(validator.isEmpty(nombre) ||
    !validator.isLength(nombre, {min: 3}))
    {
        // Redirigir a página de error por el nombre
    }
    else if (!validator.isNumeric(telefono) ||
    !validator.isLength(telefono, {min: 9, max: 9}))
    {
        // Redirigir a página de error por el teléfono
    }
    else
    {
        // Correcto, hacer la inserción
    }
});
```

- Aquí podemos consultar un listado de validadores disponibles en *validator.js*
- Aquí tenemos algunos ejemplos de saneadores en esa librería

5.2. Validación de peticiones con *express-validator*

A la hora de validar los datos de una petición en Node y Express podemos emplear algunas librerías adicionales que nos ayuden. Una de las más populares es `express-validator`, cuya web oficial podemos consultar aquí. Como la propia documentación explica, se trata de una librería que aglutina y amplía algunas opciones ofrecidas por la librería *validator.js* vista antes.

Para empezar, deberemos instalar la librería con el correspondiente comando `npm install express-validator` (no es necesario tener instalada también la librería *validator.js* anterior). Una vez instalada, tenemos a nuestra disposición un amplio abanico de validadores y saneadores (*sanitizers*), muchos de ellos incorporados de *validator.js*.

Para poder utilizar los validadores, debemos acceder a los parámetros de la petición, bien desde el cuerpo de la petición (propiedad `body`) o bien a través de la *query string* (propiedad `query`). Por ejemplo, así podemos comprobar si el nombre de un contacto no está vacío antes de hacer una inserción:

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

...

router.post('/', body('nombre').notEmpty(), (req, res) => {
    // Código habitual de inserción de contacto
});
```

Como podemos comprobar, aplicamos un *middleware* en el servicio POST que accede con `body('nombre')` al valor enviado para el nombre y comprueba con el validador `notEmpty` que no esté vacío. Si la validación se cumple se continúa con el proceso, y si no lo hace se interrumpe la petición. Si quisiéramos, por ejemplo, limpiar espacios en blanco en el nombre una vez sepamos que es válido, basta con añadir el saneador `trim` tras la validación:

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

...

router.post('/', body('nombre').notEmpty().trim(), (req, res) => {
    // Código habitual de inserción de contacto
});
```

Esta característica se denomina encadenamiento de validaciones (*validation chain*) y permite enlazar varias comprobaciones para un mismo dato en el proceso. Podéis consultar más opciones de uso de esta librería en su web oficial.

5.3. Validación de peticiones con los esquemas de Mongoose

En sesiones anteriores ya hemos visto que podemos configurar los esquemas de Mongoose para definir algunas opciones de validación en los documentos de nuestras colecciones en MongoDB. Esta característica ya va a impedir que se añadan datos con valores incorrectos, como por ejemplo contactos con edades demasiado grandes, o libros con precios negativos. Pero, además, podemos indicar en estos esquemas qué mensajes de error queremos producir en el caso de que alguna validación no sea correcta, para luego recoger esos mensajes al intentar hacer la inserción/modificación.

Vamos a modificar nuestro ejemplo de *ContactosWeb_v2*, en otra carpeta llamada *ContactosWeb_v3*. Editamos el fichero *models/contacto.js* y lo dejamos de este modo:

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

// Definición del esquema de nuestra colección
let contactoSchema = new mongoose.Schema({
    nombre: {
        type: String,
        required: [true, 'El nombre del contacto es obligatorio'],
        minlength: [3, 'El nombre del contacto es demasiado corto'],
        trim: true
    },
    telefono: {
        type: String,
        required: [true, 'El número de teléfono es obligatorio'],
        unique: true,
        trim: true,
        match: [/^\d{9}$/, 'El teléfono debe constar de 9 dígitos']
    },
    edad: {
        type: Number,
        min: [18, 'La edad mínima debe ser 18'],
        max: [120, 'La edad máxima debe ser 120']
    }
});

// Asociación con el modelo (colección contactos)
let Contacto = mongoose.model('contacto', contactoSchema);

module.exports = Contacto;
```

Notar cómo, en cada validador, añadimos un array con el valor que debe tener y el mensaje de error que mostrar en caso de que no se cumpla. Notar también que la propiedad `unique` no es un validador como tal, por lo que no deberíamos usarla para configurar un mensaje de error personalizado.

Ahora podemos modificar los servicios de inserción y/o borrado para que rendericen una vista de error con el/los mensaje(s) de error producido(s). Así podría quedar la inserción:

```
// Ruta para insertar contactos
router.post('/', (req, res) => {

    let nuevoContacto = new Contacto({
        nombre: req.body.nombre,
        telefono: req.body.telefono,
        edad: req.body.edad
    });
    nuevoContacto.save().then(resultado => {
        res.redirect(req.baseUrl);
    }).catch(error => {
        let errores = Object.keys(error.errors);
        let mensaje = "";
        if(errores.length > 0)
        {
            errores.forEach(clave => {
                mensaje += '<p>' + error.errors[clave].message + '</p>';
            });
        }
        else
        {
            mensaje = 'Error añadiendo contacto';
        }
        res.render('error', {error: mensaje});
    });
});
```

El elemento `error.errors` es un objeto que contiene detalles sobre los errores de validación que ocurrieron durante la operación de guardado con `save()`. En Mongoose, cada clave en *error.errors* corresponde a un campo del modelo que no pasó la validación. Por otra parte, `Object.keys(error.errors)` devuelve un array de las claves (nombres de campo) presentes en *error.errors*. En síntesis, en este caso, en la cláusula `catch` recorremos los errores producidos, si los hay, dentro del campo `errors` del error devuelto. Vamos construyendo con ellos una respuesta que acumule los errores detectados, o mostramos un error genérico en el caso de que no hayamos detectado ningún error concreto.

Hay que tener en cuenta también que, tal y como tenemos configurado Nunjucks, los párrafos de errores que estamos definiendo no se van a mostrar como párrafos en la vista de error, porque hemos dicho que auto-escape los textos. Por lo tanto, si ponemos un nombre de menos de tres caracteres la vista de error mostraría algo así:

```
<p>El nombre del contacto es demasiado corto</p>
```

Para quitar las marcas HTML debemos decirle a Nunjucks que no escape el contenido de ese dato en concreto, a través del modificador `safe`. Editamos la vista de error para dejarla así:

```
<html>
    <head>
        <link rel="stylesheet" href="/css/bootstrap.min.css"/>
        <link rel="stylesheet" href="/public/css/estilos.css"/>
    </head>
    <body>
        <div class="container">
            <h1>Error</h1>
            <div class="alert alert-danger">
                {% if error %}
                    {{ error|safe }}
                {% else %}
                    Error en la aplicación
                {% endif %}
            </div>
        </div>
    </body>
</html>
```

Ejercicio 3:

Sobre el ejercicio anterior de libros crea una copia llamada **LibrosWeb_v4**. Define mensajes de validación personalizados en el esquema del libro y haz que se muestre cada uno encima del campo del formulario afectado, de este modo: