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 formulario, y por otro, recoger la petición y hacer la inserción/borrado/modificación propiamente dicha.

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 contactos</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 `/nuevo_contacto`:

```
<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('nuevo_contacto');
})
```

> **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 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 al principio de todo, o 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 | Nuevo{% endblock %}

{% block contenido %}

    <h1>Inserción de nuevo contactos</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"});
    });
});
```

4. Subir ficheros en el formulario

Para subir archivos 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 otra librería adicional 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
});
```