Desarrollo de servicios REST con Express

En este apartado veremos cómo emplear un enrutamiento simple para ofrecer diferentes servicios empleando Mongoose contra una base de datos MongoDB. Para ello, crearemos una carpeta llamada "*ContactosREST*" en nuestra carpeta de pruebas ("*ProyectosNode/Pruebas*"), continuación del proyecto *ContactosMongo_v2* de sesiones anteriores (copia y pega el código de ese proyecto en esta nueva carpeta). Instalaremos Express y Mongoose en ella, lo que puede hacerse con un simple comando (aunque previamente necesitaremos haber ejecutado `npm init` para crear el archivo *package.json*):

```
npm install mongoose express
```

Nuestra aplicación tendrá una carpeta `models` con los modelos de datos (contactos, restaurantes y mascotas, según habíamos creado en versiones anteriores) y un archivo `index.js`, que anteriormente lanzaba una serie de operaciones simples, y donde ahora vamos a definir nuestro servidor Express con las rutas para responder a las diferentes operaciones sobre los contactos.

1. Esqueleto básico del servidor principal

Vamos a definir la estructura básica que va a tener nuestro servidor `index.js`, antes de añadirle las rutas para dar respuesta a los servicios. Esta estructura consistirá en incorporar Express y Mongoose, incorporar los modelos de datos, conectar con la base de datos y poner en marcha el servidor Express:

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

const Contacto = require(__dirname + "/models/contacto");
const Restaurante = require(__dirname + "/models/restaurante");
const Mascota = require(__dirname + "/models/mascota");

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

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

2. Servicios de listado (GET)

Veremos ahora cómo ofrecer diferentes servicios de listado para recuperar información de la base de datos, o bien con listados generales de varios documentos, o con listados específicos de un solo documento.

2.1. Listado de todos los contactos

El servicio que lista todos los contactos es el más sencillo: atenderemos por GET a la URI `/contactos`, y en el código haremos un `find` de todos los contactos, usando el modelo Mongoose que ya hemos creado. Devolveremos el resultado directamente en la respuesta, lo que lo convertirá automáticamente a formato JSON. Añadimos el servicio en el archivo principal `index.js`. Normalmente se añaden antes de poner en marcha el servidor (después de haber creado la variable `app`).

```
app.get('/contactos', (req, res) => {
    Contacto.find().then(resultado => {
        res.status(200)
           .send( {ok: true, resultado: resultado});
    }).catch (error => {
        res.status(500)
           .send( {ok: false,
                   error: "Error obteniendo contactos");
    });
});
```

Observad que enviamos un código de estado (200 si todo ha ido bien, 500 o fallo del servidor si no hemos podido recuperar los contactos), y el objeto JSON con los campos que explicábamos antes: el dato booleano indicando si se ha podido servir o no la respuesta, y el mensaje de error o el resultado correspondiente, según sea el caso.

2.2. Ficha de un contacto a partir de su *id*

Veamos ahora cómo procesar con Express URIs dinámicas. En este caso, accederemos por GET a una URI con el formato `/contactos/:id`, siendo *:id* el *id* del contacto que queremos obtener. Si especificamos la URI con ese mismo formato en Express, automáticamente se le asocia al parámetro que venga a continuación de */contactos* el nombre `id`, con lo que podemos acceder a él directamente por el objeto `req.params` de la petición. De este modo, el servicio queda así de simple:

```
app.get('/contactos/:id', (req, res) => {
    Contacto.findById(req.params.id).then(resultado => {
        if(resultado)
            res.status(200)
               .send({ok: true, resultado: resultado});
        else
            res.status(400)
               .send({ok: false,
                      error: "No se han encontrado contactos"});
    }).catch (error => {
        res.status(400)
           .send({ok: false,
                  error: "Error buscando el contacto indicado"});
    });
});
```

En este caso, distinguimos si el objeto `resultado` obtenido con `findById` devuelve algo o no, para emitir una u otra respuesta. En caso de que no se pueda encontrar el resultado, asumimos que es a causa de que la petición del cliente no es correcta, y emitimos un código de estado 400 (por ejemplo).

2.3. Uso de la *query string* para pasar parámetros

En el caso de querer pasar los parámetros en la *query string* (es decir, por ejemplo, `/contactos?id=XXX`) no hay forma de establecer dichos parámetros en la URI del método `get`. En ese caso deberemos comprobar si existe el parámetro correspondiente dentro del objeto `req.query`:

```
app.get('/contactos', (req, res) => {
    if(req.query.id) {
        // Buscar por id
    }
    else {
        // Listado general de contactos
    });
});
```

En cualquier caso, en estos apuntes optaremos por la versión anterior, incluyendo el parámetro en la propia URI.

2.4. Prueba de los servicios desde el navegador

Estos dos servicios de listado (general y por id) se pueden probar fácilmente desde un navegador web. Basta con poner en marcha el servidor Node (y MongoDB), abrir un navegador y acceder a esta URL para el listado general:

http://localhost:8080/contactos

O a esta otra para la ficha de un contacto (sustituyendo el *id* de la URL por uno correcto que exista en la base de datos):

http://localhost:8080/contactos/5ab391d296b06243a7cc4c4e

En este último caso, observa que:

- Si pasamos un *id* que no exista, nos indicará con un mensaje de error que "No se han encontrado contactos"
- Si pasamos un *id* que no sea adecuado (por ejemplo, que no tenga 12 bytes), obtendremos una excepción, y por tanto el mensaje de "Error buscando el contacto indicado".

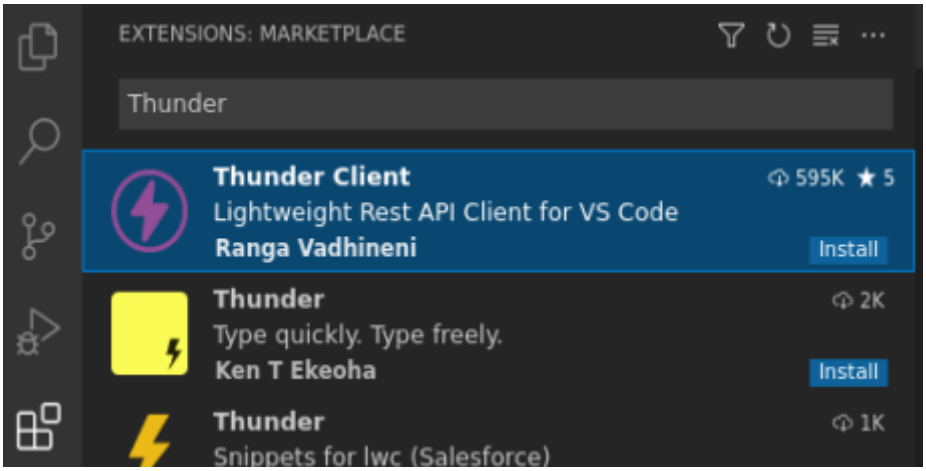
> **Ejercicio 1:**
>
> Crea un ejercicio llamado **LibrosREST** y copia dentro la estructura de modelos del proyecto **Libros_v2** de sesiones anteriores. Instala *Mongoose* y *Express* en el proyecto, y define un archivo `index.js` que incorpore el modelo de libros, cree una instancia de servidor Express, y dé respuesta a estos servicios:
>
> - `GET /libros`: devolverá un listado en formato JSON del array de libros completo de la colección.
> - `GET /libros/:id`: devolverá un objeto JSON con los datos del libro encontrado a partir de su *id*.

3. Probar servicios REST

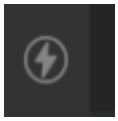
Ya hemos visto que probar unos servicios de listado (GET) es sencillo a través de un navegador. Sin embargo, pronto aprenderemos a hacer otros servicios (inserciones, modificaciones y borrados) que no son tan sencillos de probar con esta herramienta. Así que conviene ir entrando en contacto con otra más potente, que nos permita probar todos los servicios que vamos a desarrollar. Existen varias alternativas al respecto, como por ejemplo Postman, pero para evitar depender de más aplicaciones externas, vamos a utilizar la extensión **ThunderClient** de Visual Studio Code.

3.1. Instalación de la herramienta y primeros pasos

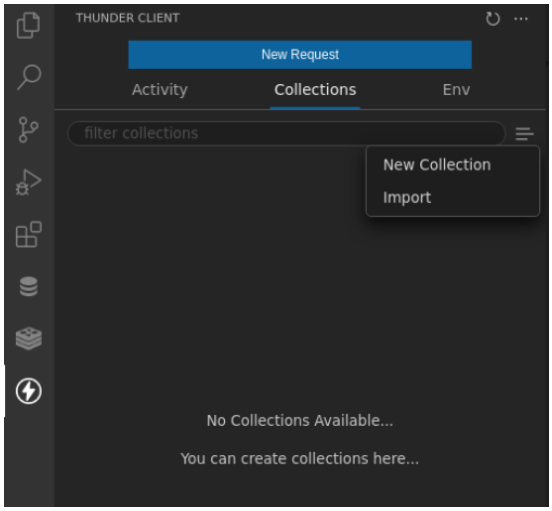
Esta herramienta nos servirá para simular peticiones a servidores web, y recoger y analizar la respuesta. La emplearemos para probar los servicios REST que desarrollaremos. La buscamos en el panel de extensiones y la instalamos:



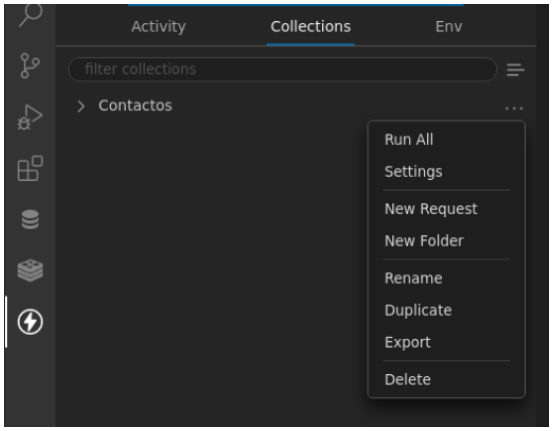
Nos aparecerá un icono en el panel izquierdo desde el que gestionaremos las conexiones y peticiones:



Es aconsejable que nuestras peticiones las agrupemos en colecciones (*collections*) de forma que cada colección se centre en un proyecto o aplicación concreta. Para ello vamos a la pestaña de colecciones y elegimos crear una, con el nombre que queramos (por ejemplo, *Contactos*).

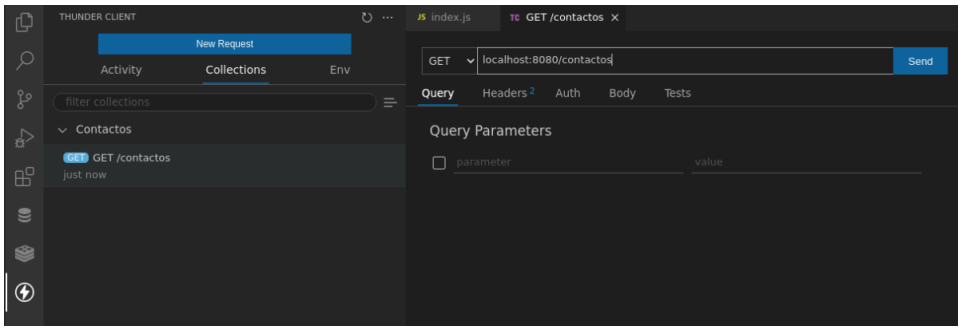


Desde el botón de puntos suspensivos junto al nombre de la colección podremos añadir nuevas peticiones asociadas a dicha colección (*New Request*), y después le pondremos un nombre.

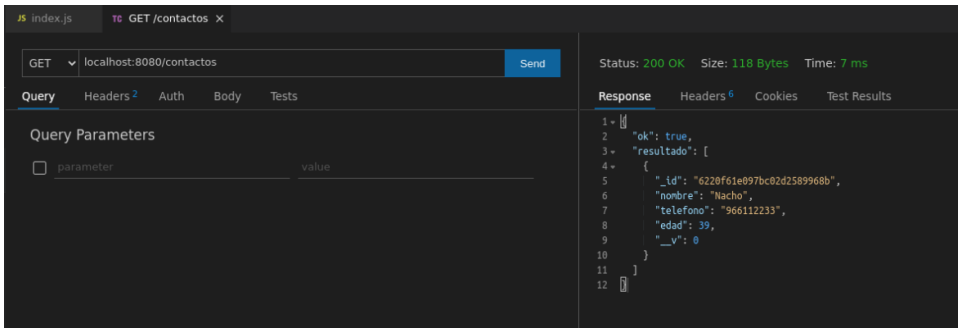


3.2. Añadir peticiones GET

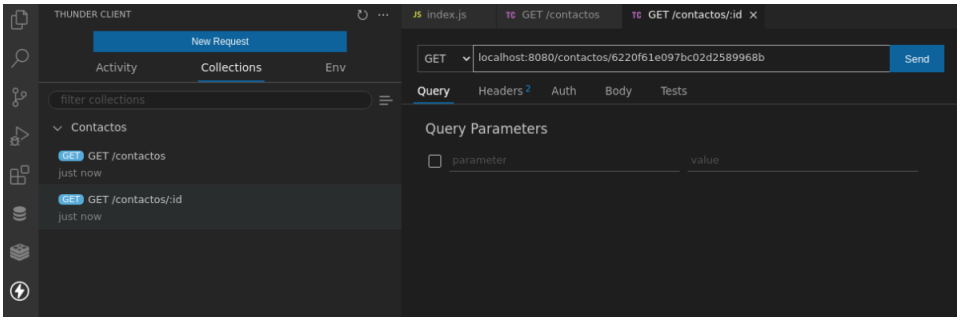
Para añadir una petición GET, podemos en primer lugar identificarla con su nombre (por ejemplo, *GET /contactos*), y en el panel que se abrirá definimos la URL de acceso (también podemos elegir el tipo de comando: GET, POST, etc). Por ejemplo:



Si pulsamos en el botón de *Send*, podemos ver la respuesta emitida por el servidor en el panel de respuesta:



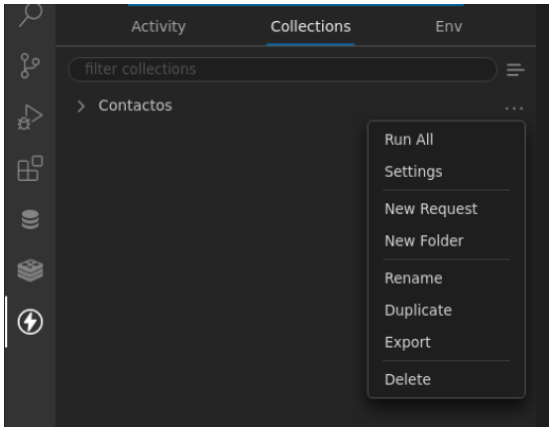
Siguiendo estos mismos pasos, podemos también crear una nueva petición para obtener un contacto a partir de su *id*, por GET:



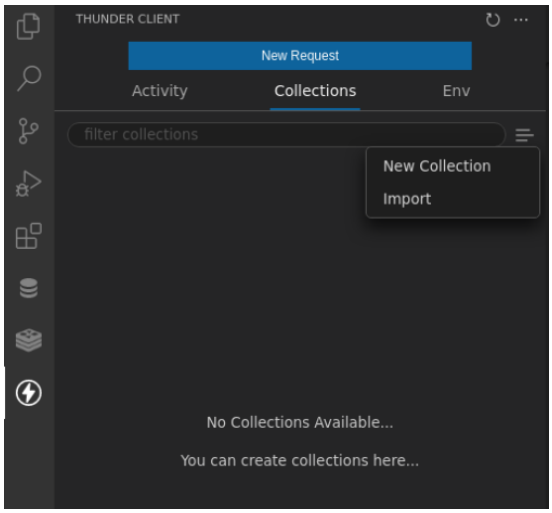
Bastaría con reemplazar el *id* de la URL por el que queramos consultar realmente.

3.3. Exportar/Importar colecciones

Podemos exportar e importar nuestras colecciones en *ThunderClient*, de forma que podemos llevarlas de un equipo a otro. Para **exportar** una colección, hacemos clic en el botón de puntos suspensivos (...) que hay junto a ella en el panel izquierdo, y elegimos *Export*.



Si queremos **importar** una colección previamente exportada, podemos hacer clic en el botón para crear colecciones, y elegimos la opción *Import*:



En cualquiera de los dos casos, las colecciones se exportan/importan a/desde un archivo en formato JSON que contiene la información de cada petición.

> **Ejercicio 2:**
>
> Crea una colección **Libros** en *ThunderClient* y define en ella una petición para cada uno de los dos servicios que has implementado antes (*GET /libros* y *GET /libros/:id*).

4. Operaciones de actualización (POST, PUT, DELETE)

Tras haber visto cómo añadir servicios GET a un servidor Express, quedan pendientes las otras tres operaciones básicas (POST, PUT y DELETE), así que veremos ahora cómo desarrollarlas en Express, y cómo probarlas con la herramienta *ThunderClient*.

4.1. Las inserciones (POST)

Vamos a insertar un nuevo contacto, pasando en el cuerpo de la petición los datos del mismo (nombre, teléfono y edad) en formato JSON. Para poder recoger esta información desde el cliente en nuestro servidor Node/Express, debemos utilizar un *middleware*, fragmento de programa que procesa la petición antes de emitir la respuesta para, en este caso, dejarnos accesibles y preparados estos datos.

Express incorpora, desde su versión 4.16, un middleware propio para este cometido. Basta con añadirlo a la aplicación, justo después de inicializar la `app` Express. Como lo que vamos a hacer es trabajar con objetos JSON, añadiremos el procesador JSON de este modo:

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

> **NOTA**: antes de la versión 4.16, para este cometido era necesario utilizar una librería de terceros llamada *body-parser*. Aquí tenéis la documentación sobre dicha librería.

Ahora vamos a nuestro servicio POST. Añadimos para ello un método `post` del objeto `app`, con los mismos parámetros que tenía el método `get` (recordemos: la ruta a la que responder, y el *callback* que se ejecutará como respuesta). El método en cuestión podría quedar así:

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

    let nuevoContacto = new Contacto({
        nombre: req.body.nombre,
        telefono: req.body.telefono,
        edad: req.body.edad
    });

    nuevoContacto.save().then(resultado => {
        res.status(200)
           .send({ok: true, resultado: resultado});
    }).catch(error => {
        res.status(400)
           .send({ok: false,
                  error: "Error añadiendo contacto"});
    });
});
```

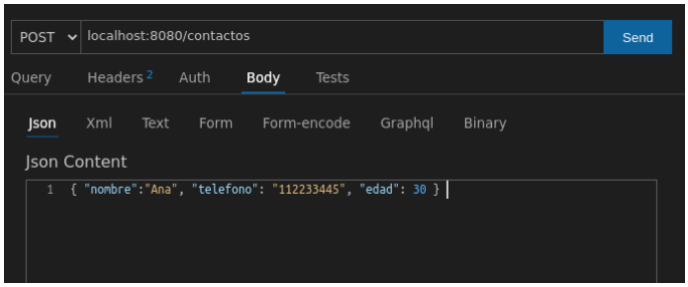
Al principio, construimos el contacto a partir de los datos JSON que llegan en el cuerpo, accediendo a cada campo por separado con `req.body.nombre_campo`. Esto es posible hacerlo gracias a que el *middleware* anterior ha pre-procesado la petición, y nos ha dejado los datos disponibles dentro del objeto `req.body`. De lo contrario, tendríamos que leer los bytes de la petición manualmente, almacenarlos en un array, y convertir desde JSON.

El resto del código es el que ya conoces de ejemplos previos con Mongoose (llamada a `save` y procesamiento del resultado), combinado con el envío del código de estado y la respuesta REST.

4.1.1. Prueba de operaciones POST

Las peticiones POST difieren de las peticiones GET en que se envía cierta información en el cuerpo de la petición. Esta información normalmente son los datos que se quieren añadir en el servidor. ¿Cómo podemos hacer esto con *ThunderClient*?

En primer lugar, creamos una nueva petición, elegimos el comando POST y definimos la URL (en este caso, *localhost:8080/contactos*). Entonces, hacemos clic en la pestaña *Body*, bajo la URL, y establecemos el tipo como *JSON* para que nos deje escribirlo en dicho formato. Después, en el cuadro de texto bajo estas opciones, especificamos el objeto JSON que queremos enviar para insertar:



Al enviar la petición, podremos ver en el cuadro la respuesta (en este caso, el documento que se acaba de insertar).

> **Ejercicio 3:**
>
> Añade el servicio `POST /libros` al proyecto *LibrosREST* anterior. Recogerá los datos del libro que le llegarán en el cuerpo de la petición e insertará el libro en cuestión en la base de datos, devolviendo un objeto JSON con el libro insertado. Añade también la correspondiente prueba en la colección de *ThunderClient*.

4.2. Las modificaciones (PUT)

La modificación de contactos es estructuralmente muy similar a la inserción: enviaremos en el cuerpo de la petición los datos nuevos del contacto a modificar (a partir de su *id*, normalmente), y utilizaremos el mismo *middleware* anterior para obtenerlos, y llamar a los métodos apropiados de Mongoose para realizar la modificación del contacto. La URI a la que asociaremos este servicio será similar a la del POST, pero añadiendo el *id* del contacto que queramos modificar. El código puede ser similar a éste:

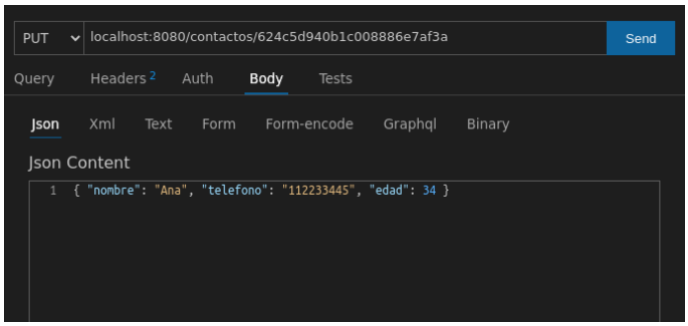
```
app.put('/contactos/:id', (req, res) => {

    Contacto.findByIdAndUpdate(req.params.id, {
        $set: {
            nombre: req.body.nombre,
            telefono: req.body.telefono,
            edad: req.body.edad
        }
    }, {new: true}).then(resultado => {
        res.status(200)
           .send({ok: true, resultado: resultado});
    }).catch(error => {
        res.status(400)
           .send({ok: false,
                  error:"Error actualizando contacto"});
    });
});
```

Como se puede ver, obtenemos el *id* desde los parámetros (`req.params`) como cuando consultábamos la ficha de un contacto, y utilizamos dicho *id* para buscar al contacto en cuestión y actualizar sus campos con `findByIdAndUpdate`. En este punto, volvemos a hacer uso del *middleware* de Express para procesar la petición y obtener los datos que llegan en formato JSON. Tras la llamada al método, devolvemos el estado y la respuesta JSON correspondiente.

4.2.1. Prueba de operaciones PUT

En el caso de peticiones PUT, procederemos de forma similar a las peticiones POST vistas antes: debemos elegir el comando (PUT en este caso), la URL, y completar el cuerpo de la petición con los datos que

queramos modificar del contacto. En este caso, además, el *id* del contacto lo enviaremos también en la propia URL:



> **Ejercicio 4:**
>
> Añade el servicio `PUT /libros/:id` al proyecto *LibrosREST* anterior. Recogerá los datos del libro que le llegarán en el cuerpo de la petición, junto con el *id* en la URL, y modificará los datos del libro indicado, devolviendo un objeto JSON con el libro modificado, o el mensaje de error correspondiente si no se ha podido modificar. Añade también la correspondiente prueba en la colección de *ThunderClient*.

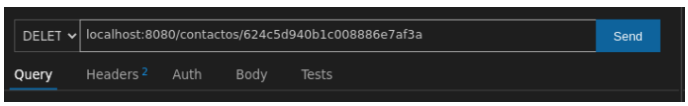
4.3. Los borrados (DELETE)

Para el borrado de contactos, emplearemos una URI similar a la ficha de un contacto o a la actualización, pero en este caso asociada al comando DELETE. Le pasaremos el *id* del contacto a borrar. Obtendremos dicho *id* también de `req.params`, y buscaremos y eliminaremos el contacto indicado.

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

    Contacto.findByIdAndRemove(req.params.id)
    .then(resultado => {
        res.status(200)
           .send({ok: true, resultado: resultado});
    }).catch(error => {
        res.status(400)
           .send({ok: false,
                  error:"Error eliminando contacto"});
    });
});
```

4.3.1. Prueba de operaciones DELETE

Para peticiones DELETE, la mecánica es similar a la ficha del contacto, cambiando el comando GET por DELETE, y sin necesidad de establecer nada en el cuerpo de la petición:



> **Ejercicio 5:**
>
> Añade el servicio `DELETE /libros/:id` al proyecto *LibrosREST* anterior. Recogerá el *id* del libro en la URL y lo borrará, devolviendo un objeto JSON con el libro eliminado, o el mensaje de error correspondiente si no se ha podido modificar. Añade también la correspondiente prueba en la colección de *ThunderClient*.

4.4. Sobre el resultado de la actualización o el borrado

En los ejemplos anteriores para PUT y DELETE, tras la llamada a `findByIdAndUpdate` o `findByIdAndRemove`, nos hemos limitado a devolver el resultado en la cláusula `then`. Sin embargo, conviene tener en cuenta que, si proporcionamos un *id* válido pero que no exista en la base de datos, el código de esta cláusula también se ejecutará, pero el objeto resultado será nulo (`null`). Podemos, por tanto, diferenciar con `if..else` si el resultado es correcto o no, y mostrar una u otra cosa:

```
if (resultado)
    res.status(200)
       .send({ok: true, resultado: resultado});
else
    res.status(400)
       .send({ok: false,
              error: "No se ha encontrado el contacto"});
```

> **Ejercicio 6:**
>
> Adapta el código de las peticiones PUT y DELETE del proyecto *LibrosREST* para que controlen si el resultado es nulo o no, mostrando una respuesta más controlada. Exporta la colección final de pruebas.

4.5. Más sobre el *middleware* de procesado de la petición

Existen otras posibilidades de uso del *middleware* que procesa la petición. En los ejemplos anteriores lo hemos empleado para procesar cuerpos con formato JSON. Pero es posible también que empleemos formularios tradicionales HTML, que envían los datos como si fueran parte de una *query-string*, pero por POST. Por ejemplo:

```
nombre=Nacho&telefono=911223344&edad=39
```

Para procesar contenidos de este otro tipo, basta con cargar el *middleware* de este otro modo:

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

También es posible añadir ambos modos juntos:

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

En este caso, el servidor Express aceptaría datos de la petición tanto en formato JSON como en formato *query-string*. En cualquier caso, deberemos asegurarnos desde el cliente (incluso si usamos *ThunderClient*) de que el tipo de contenido de la petición se ajusta al *middleware* correspondiente: para peticiones en formato JSON, el contenido deberá ser `application/json` (pestaña *JSON* en *ThunderClient*), mientras que para enviar los datos del formulario en formato *query-string*, el tipo deberá ser `application/x-www-form-urlencoded` (pestaña *Form-encode* en *ThunderClient*). Si añadimos los dos middlewares (tanto para JSON como para `urlencoded`), entonces se activará uno u otro automáticamente, dependiendo del tipo de petición que llegue desde el cliente.