6. Desarrollo de servicios REST con Express (Parte II)

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

En la sesión anterior vimos una introducción a lo que son los servicios REST, y cómo definir servicios de consulta (GET) con el framework Express, para acceder a una base de datos (MongoDB, en este caso), obtener una serie de datos y enviarlos al cliente en formato JSON.

Quedan, por tanto, pendientes las otras tres operaciones básicas de los servicios REST (POST, PUT y DELETE), así que veremos ahora cómo desarrollarlas en Express, y cómo probarlas con la herramienta Postman. Como hicimos en la sesión anterior, continuaremos con nuestra aplicación de ejemplo "*PruebaContactosExpress*" de nuestra carpeta "*ProyectosNode/Pruebas*".

6.1.1. 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. En esta ocasión, vamos a valernos de un *middleware* para Express llamado `body-parser`, que facilita el procesamiento del cuerpo de las peticiones para acceder directamente a los datos que se envían en ella. En primer lugar, deberemos instalar el módulo en nuestro proyecto:

```
npm install body-parser
```

Después, lo incluimos con require junto al resto:

```
const express = require('express');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');
...
```

Y finalmente, lo añadimos como *middleware* con la instrucción `app.use`, justo después de inicializar la `app` Express. Como lo que vamos a hacer es trabajar con objetos JSON, añadiremos el procesador JSON de *body-parser*:

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

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 body-parser* 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.

6.1.1.1. Prueba de operaciones POST con Postman

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 Postman?

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 *raw* para que nos deje escribirlo sin restricciones. También conviene cambiar la propiedad *Text* para que sea *application/json*, y que así el servidor recoja el tipo de dato adecuado y se active el *middleware body-parser*. Se añadirá automáticamente una cabecera de petición (Header) que especificará que el tipo de contenido que se va a enviar son datos JSON. Después, en el cuadro de texto bajo estas opciones, especificamos el objeto JSON que queremos enviar para insertar:

6.1.2. 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 *body-parser* 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 de la librería *body-parser* para procesar los datos que llegan desde el cuerpo de la petición. Tras la llamada al método, devolvemos el estado y la respuesta JSON correspondiente.

6.1.2.1. Prueba de operaciones PUT con Postman

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:

6.1.3. 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"});
    });
});
```

6.1.3.1. Prueba de operaciones DELETE con Postman

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:

6.1.4. 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"});
```

En este punto puedes realizar el Ejercicio 1 de los propuestos al final de la sesión.

6.1.5. Más sobre *body-parser*

Existen otras posibilidades de uso del *middleware body-parser*. 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 añadir de nuevo la librería como *middleware*, indicando en este caso otro método:

```
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended:false}));
```

En este caso, el servidor Express aceptaría datos de la petición tanto en formato JSON como en formato *query-string*. El parámetro `extended` indica qué tipo de *parser* queremos utilizar para procesar los datos de la petición: si lo dejamos a `false`, emplearemos la librería *querystring*, y si está a `true`, se empleará la librería *qs*, con algunas opciones algo más avanzadas para incluir objetos más complejos.

En cualquier caso, deberemos asegurarnos desde el cliente (incluso si usamos Postman) 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`, mientras que para enviar los datos del formulario en formato *query-string*, el tipo deberá ser `application/x-www-form-urlencoded`. 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.

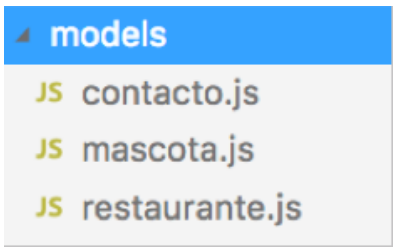
6.2. Estructurando una API REST en Express

Los ejemplos hechos hasta ahora de aplicaciones Express como proveedor de servicios REST son bastante monolíticos: en un solo archivo fuente hemos ubicado los modelos de datos, la aplicación Express en sí y las rutas a las que responderá.

A pesar de que el propio framework Express se define en su web oficial como *unopinionated*, es decir, sin opinión acerca de cómo debe ser una arquitectura de aplicación Express, sí conviene seguir ciertas normas mínimas de modularidad en nuestro código. Consultando ejemplos en Internet podemos encontrar distintas formas de estructurar aplicaciones Express, y podríamos considerar correctas muchas de ellas, desde el punto de vista de modularidad del código. Aquí vamos a proponer una estructura que seguir en nuestras aplicaciones, basándonos en otros ejemplos vistos en Internet, pero que no tiene por qué ser la mejor ni la más universal.

6.2.1. Los modelos de datos

Es habitual encontrarnos con una carpeta `models` en las aplicaciones Express donde se definen los modelos de las diferentes colecciones de datos. En nuestro ejemplo de contactos, dentro de esa carpeta "models" vamos a definir los archivos para nuestros tres modelos de datos: `contacto.js`, `restaurante.js` y `mascota.js`:



En el archivo `contacto.js` definimos el esquema y modelo de nuestra colección de contactos. Necesitaremos incluir con `require` la librería "*mongoose*" para hacer uso de los esquemas y modelos. En el siguiente código, omitimos con puntos suspensivos partes que ya se tienen de sesiones previas, para no dejar el código demasiado largo en estos apuntes:

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

// Definición del esquema
let contactoSchema = new mongoose.Schema({
    nombre: {
        ...
    },
    telefono: {
        ...
    },
    edad: {
        ...
    },
    restauranteFavorito: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'restaurante'
    },
    mascotas: [{
        type: mongoose.Schema.Types.ObjectId,
        ref: 'mascota'
    }]
});

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

module.exports = Contacto;
```

De forma similar, definimos el código del modelo `restaurante.js`:

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

// Definición del esquema
let restauranteSchema = new mongoose.Schema({
    nombre: {
        ...
    },
    direccion: {
        ...
    },
    telefono: {
        ...
    }
});

// Asociación con el modelo
let Restaurante = mongoose.model('restaurante', restauranteSchema);

module.exports = Restaurante;
```

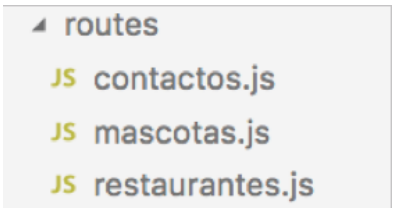
Y también para `mascota.js`:

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

// Definición del esquema
let mascotaSchema = new mongoose.Schema({
    nombre: {
        ...
    },
    tipo: {
        ...
    }
});

// Asociación con el modelo
let Mascota = mongoose.model('mascota', mascotaSchema);

module.exports = Mascota;
```

6.2.2. Las rutas y enrutadores

Imaginemos que la gestión de contactos en sí (alta / baja / modificación / consulta de contactos) se realizará mediante servicios englobados en una URI que empieza por

`/contactos`. Para el caso de restaurantes y mascotas, utilizaremos las URIs `/restaurantes` y `/mascotas`, respectivamente. Vamos a definir tres enrutadores diferentes, uno para cada cosa. Lo normal en estos casos es crear una subcarpeta `routes` en nuestro proyecto, y definir dentro un archivo fuente para cada grupo de rutas. En nuestro caso, definiríamos un archivo `contactos.js` para las rutas relativas a la gestión de contactos, otro `restaurantes.js` para los restaurantes, y otro `mascotas.js` para las mascotas.



> **NOTA**: es también habitual que la carpeta `routes` se llame `controllers` en algunos ejemplos que podemos encontrar por Internet, ya que lo que estamos definiendo en estos archivos son básicamente controladores, que se encargan de comunicarse con el modelo de datos y ofrecer al cliente una respuesta determinada.

Vamos a definir el código de estos tres enrutadores que hemos creado. En cada uno de ellos, utilizaremos el modelo correspondiente de la carpeta "*models*" para poder manipular la colección asociada.

Comencemos por la colección más sencilla de gestionar: la de **mascotas**. Definiremos únicamente servicios para listar (GET), insertar (POST) y borrar (DELETE). El código del enrutador `routes/mascotas.js` quedaría así (se omite el código interno de cada servicio, que sí puede consultarse en los ejemplos de código de la sesión):

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

let Mascota = require(__dirname + '/../models/mascota.js');

let router = express.Router();

// Servicio de listado
router.get('/', (req, res) => {
    ...
});

// Servicio de inserción
router.post('/', (req, res) => {
    ...
});

// Servicio de borrado
router.delete('/:id', (req, res) => {
    ...
});

module.exports = router;
```

Notar que utilizamos un objeto `Router` de Express para gestionar los servicios, a diferencia de lo que veníamos haciendo en sesiones anteriores, donde nos basábamos en la propia aplicación (objeto `app`) para gestionarlos. De esta forma, definimos un router para cada grupo de servicios, que se encargará de su procesamiento. Lo mismo ocurrirá para los dos enrutadores siguientes (restaurantes y contactos).

Notar también que las rutas no hacen referencia a la URI `/mascotas`, sino que apuntan a una raíz `/`. El motivo de esto lo veremos en breve.

De forma análoga, podríamos definir los servicios GET, POST y DELETE para los **restaurantes** en el enrutador `routes/restaurantes.js`:

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

let Restaurante = require(__dirname + '/../models/restaurante.js');

let router = express.Router();

// Servicio de listado
router.get('/', (req, res) => {
    ...
});

// Servicio de inserción
router.post('/', (req, res) => {
    ...
});

// Servicio de borrado
router.delete('/:id', (req, res) => {
    ...
});

module.exports = router;
```

Quedan, finalmente, los servicios para **contactos**. Adaptaremos los que ya hicimos en sesiones anteriores, copiándolos en el enrutador `routes/contactos.js`. El código quedaría así:

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

let Contacto = require(__dirname + '/../models/contacto.js');

let router = express.Router();

// Servicio de listado general
router.get('/', (req, res) => {
    ...
});

// Servicio de listado por id
router.get('/:id', (req, res) => {
    ...
});

// Servicio para insertar contactos
router.post('/', (req, res) => {
    ...
});

// Servicio para modificar contactos
router.put('/:id', (req, res) => {
    ...
});

// Servicio para borrar contactos
router.delete('/:id', (req, res) => {
    ...
});

module.exports = router;
```

6.2.3. La aplicación principal

El servidor principal ve muy aligerado su código. Básicamente se encargará de cargar las librerías y enrutadores, conectar con la base de datos y poner en marcha el servidor:

```
// Librerías externas
const express = require('express');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');

// Enrutadores
const mascotas = require(__dirname + '/routes/mascotas');
const restaurantes = require(__dirname + '/routes/restaurantes');
const contactos = require(__dirname + '/routes/contactos');

// Conexión con la BD
mongoose.connect('mongodb://localhost:27017/contactos',
    {useNewUrlParser: true});

let app = express();

// Carga de middleware y enrutadores
app.use(bodyParser.json());
app.use('/mascotas', mascotas);
app.use('/restaurantes', restaurantes);
app.use('/contactos', contactos);

// Puesta en marcha del servidor
app.listen(8080);
```

Los enrutadores se cargan como *middleware*, empleando `app.use`. En esa instrucción, se especifica la ruta con la que se mapea cada enrutador, y por este motivo, dentro de cada enrutador las rutas ya hacen referencia a esa ruta base que se les asigna desde el servidor principal; por ello todas comienzan por `/`.

En este punto, puedes realizar el Ejercicio 2 de los propuestos al final de la sesión. También se deja propuesto como opcional el Ejercicio 3.

6.3. Ejercicios propuestos

Para los ejercicios de esta sesión, crea una subcarpeta llamada "**Sesion6**" en tu carpeta "**ProyectosNode/Ejercicios**", para dentro ir creando un proyecto para cada ejercicio.

Ejercicio 1

Crea una carpeta llamada "**Ejercicio_6_1**" en la carpeta de ejercicios "*ProyectosNode/Ejercicios/Sesion6*", que sea una copia del *Ejercicio_5_2* de la sesión anterior, donde definimos una serie de servicios GET para la la base de datos de libros. En este ejercicio vamos a añadir las operaciones de POST, PUT y DELETE sobre esta aplicación

- **Insertar** un nuevo libro. Accederá por POST a la URI `/libros`

- **Modificar** un libro a partir de su *id*. Accederá por PUT a la URI `/libros/:id`
- **Borrar** un libro a partir de su *id*. Accederá por DELETE a la URI / `libros/:id`

Para finalizar, añade las pruebas en Postman para estos servicios, sobre la misma colección que ya iniciaste en la sesión anterior. Exporta la colección con los nuevos cambios añadidos.

Ejercicio 2

Crea una copia del ejercicio anterior en otra carpeta llamada "**Ejercicio_6_2**", y estructura aquí la aplicación tal y como se ha explicado en el apartado 6.2 de esta sesión, separando el modelo de datos, los enrutadores o controladores y la aplicación principal. Comprueba con la colección de pruebas Postman del ejercicio anterior que los servicios siguen funcionando de la misma manera.

Ejercicio 3

Opcional

Sobre el ejercicio anterior, añade en las carpetas correspondientes los siguientes archivos:

- El modelo de autores, y su correspondiente relación con el modelo de libros. El propio modelo de autores puedes recuperarlo de los ejercicios de la sesión 4.
- El enrutador para los autores. En este enrutador sólo vamos a definir los servicios de listado general (GET), inserción (POST) y borrado (DELETE).
- Modifica el programa principal para que los libros respondan a URIs con el prefijo `/libros` y los autores estén asociados al prefijo `/autores`.
- Añade las tres pruebas de GET, POST y DELETE para autores a la colección de Libros de Postman.