Estructura de una API REST en Express

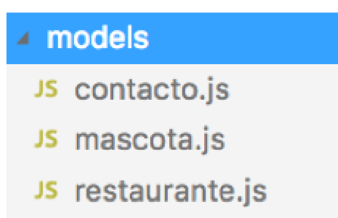
Los ejemplos hechos hasta ahora de aplicaciones Express como proveedor de servicios REST son bastante monolíticos: en un solo archivo fuente hemos ubicado la aplicación Express en sí y las rutas a las que responderá.

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.

Para empezar, crearemos una copia de nuestro proyecto *ContactosREST* en otro llamado *ContactosREST_v2*, donde iremos incorporando los cambios que veremos a continuación.

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" ya hemos definido los archivos para nuestros tres modelos de datos: `contacto.js`, `restaurante.js` y `mascota.js`, y los hemos incorporado con `require` desde el programa principal:



2. Las rutas y enrutadores

Imaginemos que la gestión de contactos en sí (alta / baja / modificación / consulta de contactos) se realiza 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.

- routes
  - contactos.js
  - mascotas.js
  - restaurantes.js

> **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 pasos anteriores, copiándolos en el enrutador `routes/contactos.js`. El código quedaría así:

```javascript
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;
```

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');

// 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://127.0.0.1:27017/contactos');

let app = express();

// Carga de middleware y enrutadores
app.use(express.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 `/`.

> **Ejercicio 1:**
>
> Crea una copia del ejercicio *LibrosREST* de sesiones anteriores en otra carpeta llamada "**LibrosREST_v2**", y estructura aquí la aplicación tal y como se ha explicado en este documento, separando el modelo de datos, los enrutadores o controladores y la aplicación principal.
>
> Define un enrutador para los autores, con el prefijo `/autores`. En este enrutador sólo vamos a definir los servicios de listado general (GET), inserción (POST) y borrado (DELETE). Añade también las correspondientes pruebas en la colección de *ThunderClient*.

4. Servicios REST y llamadas asíncronas

En documentos anteriores ya comentamos que el uso de métodos de Mongoose era asíncrono, y que podíamos invocar a estos métodos tanto usando promesas simples como mediante la especificación `async/await`. Esta última opción es más cómoda cuando tenemos que hacer varias operaciones enlazadas ya que, de lo contrario, nos vemos "obligados" a anidar cláusulas *then* y que el código sea más difícil de seguir.

Relacionado con lo que vimos en aquel apartado, hay que tener en cuenta que los servicios que desarrollamos en Express pueden ser asíncronos (*async*), por lo que podemos emplear dentro de ellos llamadas de tipo `await` para enlazar de forma síncrona una serie de operaciones. Por ejemplo, un servicio que añada una mascota al array de subdocumentos de un contacto, dado su *id*, podría quedar así:

```
app.put('/:id/mascotas', async (req, res) => {
    try
    {
        let contacto = await Contacto.findById(req.params.id);
        // Recogemos los datos de la mascota del cuerpo de la petición
        let datosMascota = {
            nombre: req.body.nombre,
            tipo: req.body.tipo
        };
        contacto.mascotas.push(datosMascota);
        let resultado = await contacto.save();
        res.status(200).send({ok: true, resultado: resultado});
    }
    catch(error)
    {
        res.status(400).send({ok: false, error:"Error añadiendo mascota"});
    }
});
```