Estructura d'una API REST amb Express

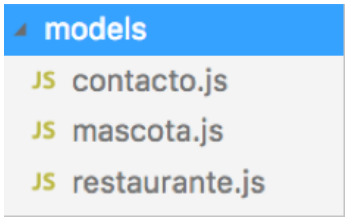

Els exemples fets fins ara d'aplicacions Express com a proveïdor de serveis REST són bastant monolítics: en un sol arxive font hem situat l'aplicació Express en si i les rutes a les quals respondrà.

A pesar que el propi framework Express es defineix en el seu web oficial com *unopinionated*, és a dir, sense opinió sobre com ha de ser una arquitectura d'aplicació Express, sí que convé seguir unes certes normes mínimes de modularidad en el nostre codi. Consultant exemples en Internet podem trobar diferents maneres d'estructurar aplicacions Express, i podríem considerar correctes moltes d'elles, des del punt de vista de modularidad del codi. Ací proposarem una estructura que seguir en les nostres aplicacions, basant-nos en altres exemples vistos en Internet, però que no té per què ser la millor ni la més universal.

Per a començar, crearem una còpia del nostre projecte *ContactesREST* en un altre anomenat *ContactesREST_v2*, on anirem incorporant els canvis que veurem a continuació.

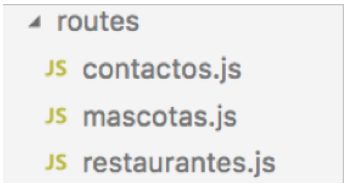
1. Els models de dades

És habitual trobar-nos amb una carpeta `models` en les aplicacions Express on es defineixen els models de les diferents col·leccions de dades. En el nostre exemple de contactes, dins d'aqueixa carpeta "models" ja hem definit els arxius per als nostres tres models de dades: `contacte.js`, `restaurant.js` i `mascota.js`, i els hem incorporats amb `require` des del programa principal:



2. Les rutes i encaminadors

Imaginem que la gestió de contactes en si (alta / baixa / modificació / consulta de contactes) es realitza mitjançant serveis englobats en una URI que comença per `/contactes`. Per al cas de restaurants i mascotes, utilitzarem les URIs `/restaurants` i `/mascotes`, respectivament. Definirem tres encaminadors diferents, un per a cada cosa. El normal en aquests casos és crear una subcarpeta `routes` en el nostre projecte, i definir dins un arxiu font per a cada grup de rutes. En el nostre cas, definiríem un arxiu `contactes.js` per a les rutes relatives a la gestió de contactes, un altre `restaurants.js` per als restaurants, i un altre `mascotes.js` per a les mascotes.



> **NOTA**: és també habitual que la carpeta `routes` es cride `controllers` en alguns exemples que podem trobar per Internet, ja que el que estem definint en aquests arxius són bàsicament controladors, que s'encarreguen de comunicar-se amb el model de dades i oferir al client una resposta determinada.

Definirem el codi d'aquests tres encaminadors que hem creat. En cadascun d'ells, utilitzarem el model corresponent de la carpeta "*models*" per a poder manipular la col·lecció associada.

Comencem per la col·lecció més senzilla de gestionar: la de **mascotes**. Definirem únicament serveis per a llistar (GET), inserir (POST) i esborrar (DELETE). El codi de l'encaminador `routes/mascotes.js` quedaria així (s'omet el codi intern de cada servei, que sí que pot consultar-se en els exemples de codi de la sessió):

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

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

let router = express.Encaminador();

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

// Servei d'inserció
router.post('/', (req, res) => {
 ...
});

// Servei d'esborrat
router.delete('/:id', (req, res) => {
 ...
});

module.exports = router;
```

Notar que utilitzem un objecte `Encaminador` d'Express per a gestionar els serveis, a diferència del que féiem en sessions anteriors, on ens basàvem en la pròpia aplicació (objecte `app`) per a gestionar-los. D'aquesta manera, definim un encaminador per a cada grup de serveis, que s'encarregarà del seu processament. El mateix ocorrerà per als dos encaminadors següents (restaurants i contactes).

Notar també que les rutes no fan referència a la URI `/mascotes`, sinó que apunten a una arrel `/`. El motiu d'això ho veurem en breu.

De manera anàloga, podríem definir els serveis GET, POST i DELETE per als **restaurants** en l'encaminador `routes/restaurants.js`:

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

let Restaurant = require(__dirname + '/../models/restaurant.js');

let router = express.Encaminador();

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

// Servei d'inserció
router.post('/', (req, res) => {
 ...
});

// Servei d'esborrat
router.delete('/:id', (req, res) => {
 ...
});

module.exports = router;
```

Queden, finalment, els serveis per a **contactes**. Adaptarem els que ja vam fer en passos anteriors, copiant-los en l'encaminador `routes/contactes.js`. El codi quedaria així:

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

let Contacte = require(__dirname + '/../models/contacte.js');

let router = express.Encaminador();

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

// Servei de llistat per id
router.get('/:id', (req, res) => {
 ...
});

// Servei per a inserir contactes
router.post('/', (req, res) => {
 ...
});

// Servei per a modificar contactes
router.put('/:id', (req, res) => {
 ...
});

// Servei per a esborrar contactes
router.delete('/:id', (req, res) => {
 ...
});

module.exports = router;
```

3. L'aplicació principal

El servidor principal veu molt alleugerit el seu codi. Bàsicament s'encarregarà de carregar les llibreries i encaminadors, connectar amb la base de dades i posar en marxa el servidor:

```
// Llibreries externes
const express = require('express');
const mongoose = require('mongoose');

// Encaminadors
const mascotes = require(__dirname + '/routes/mascotes');
const restaurants = require(__dirname + '/routes/restaurants');
const contactes = require(__dirname + '/routes/contactes');

// Connexió amb la BD
mongoose.connect('mongodb://127.0.0.1:27017/contactes');

let app = express();

// Càrrega de middleware i encaminadors
app.use(express.json());
app.use('/mascotes', mascotes);
app.use('/restaurants', restaurants);
app.use('/contactes', contactes);

// Posada en marxa del servidor
app.listen(8080);
```

Els encaminadors es carreguen com *middleware*, emprant `app.use`. En aqueixa instrucció, s'especifica la ruta amb la qual es mapea cada encaminador, i per aquest motiu, dins de cada encaminador les rutes ja fan referència a aqueixa ruta base que se'ls assigna des del servidor principal; per això totes comencen per `/`.

> **Exercici 1:**
>
> Crea una còpia de l'exercici *LlibresREST* de sessions anteriors en una altra carpeta anomenada "**LlibresREST_v2**", i estructura ací l'aplicació tal com s'ha explicat en aquest document, separant el model de dades, els encaminadors o controladors i l'aplicació principal.
>
> Defineix un encaminador per als autors, amb el prefix `/autors`. En aquest encaminador només definirem els serveis de llistat general (GET), inserció (POST) i esborrat (DELETE). Afig també les corresponents proves en la col·lecció de *ThunderClient*.

4. Serveis REST i crides asíncrones

En documents anteriors ja vam comentar que l'ús de mètodes de Mongoose era asíncron, i que podíem invocar a estos mètodes tant usant promeses simples com mitjançant l'especificació `async/await`. Esta última opció és més còmoda quan hem de fer diverses operacions enllaçades ja que, en cas contrari, ens veiem "obligats" a niar clàusules *then* i que el codi siga més difícil de seguir.

Relacionat amb el que vam veure en aquell apartat, cal tindre en compte que els serveis que desenvolupem en Express poden ser asíncrons (*async*), per la qual cosa podem emprar dins d'ells anomenades de tipus

`await` per a enllaçar de manera síncrona una sèrie d'operacions. Per exemple, un servei que afija una mascota a l'array de subdocumentos d'un contacte, donat el seu *id*, podria quedar així:

```
app.put('/:id/mascotes', async (req, res) => {
    try
    {
        let contacte = await Contacte.findById(req.params.id);
        // Recollim les dades de la mascota del cos de la petició
        let dadesMascota = {
            nom: req.body.nom,
            tipus: req.body.tipus
        };
        contacte.mascotes.push(dadesMascota);
        let resultat = await contacte.save();
        res.status(200).send({ok: true, resultat: resultat});
    }
    catch(error)
    {
        res.status(400).send({ok: false, error:"Error afegint mascota"});
    }
});
```