Node.js

en el desarrollo de aplicaciones web y servicios

6. El framework Express (II)
Enrutadores y uso de middleware

Ignacio Iborra Baeza

Índice de contenidos

1. El *middleware* en Express

El término *middleware* ha sido mencionado algunas veces con anterioridad, pero aún no habíamos explicado de qué se trata. En el ámbito en que lo estamos tratando, un *middleware* es una función Javascript que gestiona peticiones y respuestas HTTP, de forma que puede manipularlas y pasarlas al siguiente *middleware* para que las siga procesando, o bien terminar el proceso y enviar por sí misma una respuesta al cliente.

Existen diferentes *middlewares* disponibles en los módulos de Node, tales como *body-parse* para procesar el cuerpo de peticiones POST y poder acceder a su contenido de forma más cómoda. Utilizaremos este y otros ejemplos en sucesivas pruebas. También podemos definir nuestra propia función de *middleware*, y mediante el método `use` de la aplicación, incluirla en la cadena de procesamiento de la petición y la respuesta. Por ejemplo, el siguiente *middleware* saca por consola la dirección IP del cliente que hace la petición:

```
app.use((req, res, next) => {
    console.log("Petición desde", req.ip);
    next();
});
```

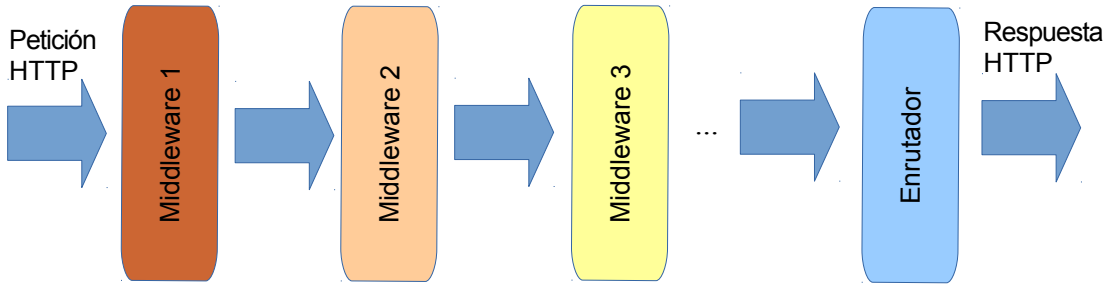
Como vemos, el *middleware* no es más que una función que acepta tres parámetros: la petición, la respuesta, y una referencia al siguiente *middleware* que debe ser llamado (`next`). Cualquier *middleware* puede finalizar la cadena enviando algo en la respuesta al cliente, pero si no lo hace, debe obligatoriamente llamar al siguiente eslabón (`next`).

Podemos emplear diversas llamadas a `use` para cargar distintos *middlewares*, y el orden en que hagamos estas llamadas es importante, porque marcará el orden en que se ejecutarán dichos *middlewares*.

```
app.use(bodyParser);
app.use(function(...) { ... });
app.use(...);
```

1.1. Flujo básico de una petición y respuesta

Teniendo en cuenta todo lo visto anteriormente, el flujo elemental de una petición cliente que llega a un servidor Express es el siguiente:



Existen algunos *middlewares* realmente útiles, como el ya citado *body-parser,* o el enrutador (*router*), que típicamente es el último eslabón de la cadena, y se emplea para configurar diferentes rutas de petición disponibles, y la respuesta para cada una de ellas. Veremos ejemplos de su uso a continuación.

2. Enrutar con Express

Ya sabemos qué es Express y sus mecanismos básicos de funcionamiento, y hemos visto una forma básica de definir rutas para responder a peticiones GET, POST, PUT o DELETE en sesiones anteriores. Este es uno de los usos principales que se le da a este framework: proporcionar servicios REST a los clientes que lo soliciten.

En esta sesión veremos cómo podemos estructurar nuestro código si el número de rutas de la aplicación, o su heterogeneidad, hacen que sea difícil mantener el código en un solo archivo, cosa bastante habitual, por otra parte.

2.1. Reestructurando el proyecto Express

Hasta ahora, las pruebas que hemos hecho para proyectos web con Express se han basado en un único archivo "index.js" o similar donde incluíamos todo el código necesario para que la mini-aplicación funcionara:

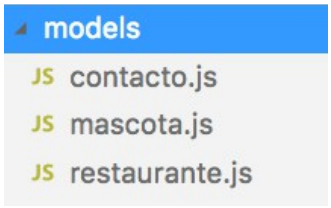
- Conexión a la base de datos
- Definición de esquemas y modelos
- Inicialización de aplicación Express
- Definición de rutas GET, POST, etc.
- Puesta en marcha del servidor.

Como decíamos, y especialmente cuando la aplicación es más compleja y requiere de más servicios, o queremos tratar de forma independiente los servicios relativos a conceptos distintos, podemos definir un enrutador para cada grupo de servicios, y crearlos en archivos aparte, que luego se enlazan como *middleware* con el servidor principal. Además, también conviene separar los modelos y esquemas de las distintas colecciones en archivos aparte, para poderlos tratar de forma independiente desde los archivos que los requieran.

Volvamos a nuestra aplicación de contactos que nos viene sirviendo de ejemplo en sesiones anteriores. Ya teníamos una versión con servicios básicos GET, POST, PUT y DELETE en un proyecto llamado "PruebaContactosExpress", en nuestra carpeta "ProyectosNode/Pruebas", que realizaban la gestión básica de la colección de contactos. Vamos a partir de esa versión (puedes hacer una copia de seguridad de ella para no perderla, si quieres), y vamos a añadirle funcionalidades para gestionar también los restaurantes y mascotas que hemos añadido después a la base de datos.

2.1.1. La carpeta de modelos

Es habitual encontrarnos con una carpeta "models" en las aplicaciones Express donde se definen los modelos de las diferentes colecciones de datos. 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: {
        ...
```

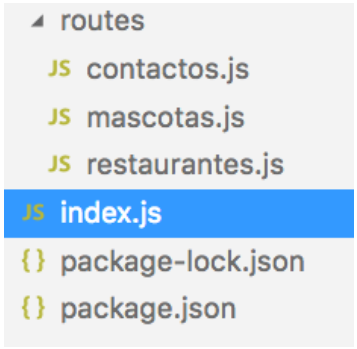
```
    }
});

// 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;
```

2.1.2. La carpeta de 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.



2.2. Definiendo enrutadores independientes

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.

2.2.1. El enrutador para mascotas

Comencemos por la colección más sencilla de gestionar: la de mascotas. El código del enrutador "routes/mascotas.js" quedaría así:

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

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

let router = express.Router();

// Servicio de listado
router.get('/', (req, res) => {
    Mascota.find().then(resultado => {
        res.send(resultado);
    }).catch(error => {
        res.send([]);
    });
});

// Servicio de inserción
router.post('/', (req, res) => {
    let nuevaMascota = new Mascota({
        nombre: req.body.nombre,
        tipo: req.body.tipo
    });
    nuevaMascota.save().then(resultado => {
        res.send({error: false, resultado: resultado});
    }).catch(error => {
        res.send({error: true, mensajeError:"Error añadiendo mascota"});
    });
});

// Servicio de borrado
router.delete('/:id', (req, res) => {
    Mascota.findByIdAndRemove(req.params.id).then(resultado => {
        res.send({resultado: resultado});
    }).catch(error => {
        res.send({error: true, mensajeError: "Error borrando mascota"});
    });
});

module.exports = router;
```

En este caso, hemos definido tres servicios básicos (GET, POST y DELETE), pero podríamos definir los que quisiéramos. 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.

2.2.2. El enrutador para restaurantes

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) => {
    Restaurante.find().then(resultado => {
        res.send(resultado);
    }).catch(error => {
        res.send([]);
    });
});

// Servicio de inserción
router.post('/', (req, res) => {
    let nuevoRestaurante = new Restaurante({
        nombre: req.body.nombre,
        direccion: req.body.direccion,
        telefono: req.body.telefono
    });
    nuevoRestaurante.save().then(resultado => {
        res.send({error: false, resultado: resultado});
    }).catch(error => {
        res.send({error: true, mensajeError:"Error añadiendo restaurante"});
    });
});

// Servicio de borrado
router.delete('/:id', (req, res) => {
    Restaurante.findByIdAndRemove(req.params.id).then(resultado => {
        res.send({resultado: resultado});
    }).catch(error => {
        res.send({error: true, mensajeError: "Error borrando restaurante"});
    });
});

module.exports = router;
```

2.2.3. El enrutador para contactos

Quedan, finalmente, los servicios para contactos. Adaptaremos los que ya hicimos en sesiones anteriores, copiándolos en el enrutador "routes/contactos.js" y añadiéndoles los restaurantes y mascotas en las operaciones de POST y PUT. 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) => {
    Contacto.find().then(resultado => {
        res.send(resultado);
    }).catch (error => {
        res.send([]);
    });
});
```

```
// Servicio de listado por id
router.get('/:id', (req, res) => {
    Contacto.findById(req.params.id).then(resultado => {
        if(resultado)
            res.send({error: false, resultado: resultado});
        else
            res.send({error: true,
                      mensajeError: "No se han encontrado contactos"});
    }).catch (error => {
        res.send({error: true,
                  mensajeError: "Error buscando el contacto indicado"});
    });
});

// Servicio para insertar contactos
router.post('/', (req, res) => {
    let nuevoContacto = new Contacto({
        nombre: req.body.nombre,
        telefono: req.body.telefono,
        edad: req.body.edad,
        restauranteFavorito: req.body.restauranteFavorito,
        mascotas: req.body.mascotas
    });
    nuevoContacto.save().then(resultado => {
        res.send({error: false, resultado: resultado});
    }).catch(error => {
        res.send({error: true,
                  mensajeError: "Error añadiendo contacto"});
    });
});

// Servicio para modificar contactos
router.put('/:id', (req, res) => {
    Contacto.findByIdAndUpdate(req.params.id, {
        $set: {
            nombre: req.body.nombre,
            telefono: req.body.telefono,
            edad: req.body.edad,
            restauranteFavorito: req.body.restauranteFavorito,
            mascotas: req.body.mascotas
        }
    }, {new: true}).then(resultado => {
        res.send({error: false, resultado: resultado});
    }).catch(error => {
        res.send({error: true,
                  mensajeError:"Error actualizando contacto"});
    });
});

// Servicio para borrar contactos
router.delete('/:id', (req, res) => {
    Contacto.findByIdAndRemove(req.params.id).then(resultado => {
        res.send({error: false, resultado: resultado});
    }).catch(error => {
        res.send({error: true,
                  mensajeError:"Error eliminando contacto"});
    });
});
```

```
module.exports = router;
```

Adicionalmente, también podríamos poblar con (`populate`) los datos de los restaurantes y mascotas en los listados GET, aunque no lo veremos en este ejemplo.

2.2.4. El servidor 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.Promise = global.Promise;
mongoose.connect('mongodb://localhost:27017/contactos');

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 1 de los propuestos al final de la sesión.

3. Más sobre los enrutadores y el middleware

Ya hemos visto qué es el middleware en Express, y cómo dividir el código de nuestra aplicación en módulos con enrutadores independientes para cada concepto o colección. Veamos ahora algunos aspectos adicionales que conviene tener en cuenta acerca de los enrutadores y el middleware

3.1. Añadir middleware a enrutadores concretos

Cuando definimos enrutadores independientes en archivos separados, podemos añadir *middleware* por separado también a cada enrutador, empleando el método `use` del propio enrutador. Por ejemplo, podemos añadir un middleware en el archivo "routes/contactos.js" que muestre por consola la fecha actual:

```
let router = express.Router();

router.use((req, res, next) => {
    console.log(new Date().toString());
    next();
});
...
```

Prueba a realizar aquí el Ejercicio 2 del final de la sesión, de carácter optativo, para practicar con la inclusión de middleware en aplicaciones Express.

3.2. Dividir los métodos del servidor principal en módulos

Existe alguna que otra alternativa para poder dividir nuestro código en módulos que enruten los servicios de forma independiente entre ellos. Consiste en no usar enrutadores (*routers*), pero separar los métodos de enrutamiento de la aplicación principal en diferentes módulos. Por ejemplo, si tenemos una aplicación principal como ésta:

```
let app = express();

app.get('/noticias', (req, res) => {
    ...
});

app.get('/usuarios', (req, res) => {
    ...
});

app.post('/usuarios', (req, res) => {
    ...
});
```

podemos crear una carpeta con los diferentes módulos (podemos llamarla "routes", o "controllers", por ejemplo), y dentro crear un archivo para cada grupo de servicios. Por ejemplo, siguiendo el mismo ejemplo que antes, podríamos crear "controllers/noticias.js" y "controllers/usuarios.js". En ambos casos, lo que vamos a hacer es exportar una función que defina los distintos servicios. Dicha función recibirá como parámetro la aplicación

sobre la que trabajar (el objeto app), para así poder definir los servicios sobre ella. En el caso de "noticias.js", quedaría así:

```
module.exports = (app) => {
    app.get('/noticias', (req, res) => {
        ...
    });
}
```

Y el archivo "usuarios.js" sería así:

```
module.exports = (app) => {
    app.get('/usuarios', (req, res) => {
        ...
    });
    app.post('/usuarios', (req, res) => {
        ...
    });
}
```

Notar que ya no tenemos una ruta relativa a la base de cada enrutador, sino que debemos especificar la URI completa en cada servicio. La aplicación principal quedaría de este modo:

```
const express = require('express');
const usuarios = require('./controllers/usuarios');
const principal = require('./controllers/noticias');

let app = express();
usuarios(app);
noticias(app);

app.listen(8080);
```

Elegir esta estrategia o la anterior con enrutadores para separar nuestro código es algo más o menos arbitrario, y la propia documentación de Express no se inclina hacia ninguna tendencia en concreto. Sí es cierto que la estrategia de enrutadores independientes favorece, por ejemplo, el poder definir *middleware* específico para un determinado enrutador, pero si no vamos a emplear estas características, ambas opciones son válidas.

4. Ejercicios

4.1. Ejercicio 1

En este ejercicio vamos a combinar lo hecho durante las sesiones anteriores: utilizaremos la aplicación web básica que gestionaba los libros, y le añadiremos la colección y modelo de autores, más los comentarios, generando los enrutadores y servicios para cada cosa. Para empezar, crea una carpeta "Sesion6" en tu carpeta de "ProyectosNode/Ejercicios".

Aquí detallamos los pasos para completar el ejercicio. En cuanto finalice el plazo de entrega de la sesión anterior, daremos una plantilla que comprenderá los 3 primeros pasos, ya realizados.

1. Copia la carpeta "Ejercicio_4_2" de la sesión 4 del curso en la carpeta de la "Sesion6", y renombra el ejercicio a "Ejercicio_6".
2. Crea la subcarpeta "models" dentro del proyecto, crea dentro un archivo llamado "libro.js" y copia en él la definición del esquema de libros y comentarios que hiciste en la sesión 5, y exporta el modelo de libros con `module.exports`, como hemos hecho en el ejemplo de esta sesión con los modelos realizados. En este caso, no será necesario exportar los comentarios, ya que no tienen colección propia (son subdocumentos). La estructura será más o menos así:

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

// Definir esquemas de comentario y libro

let comentarioSchema = new mongoose.Schema({
    ...
});

let libroSchema = new mongoose.Schema({
   ...
});

// Asociar a modelo de libro con la colección correspondiente
let Libro = mongoose.model('libro', libroSchema);

module.exports = Libro;
```

3. Crea la carpeta "routes", y añade dentro el archivo "libros.js". Copia todas las rutas de libros que hicimos en el Ejercicio 4.2, adaptándolas al enrutador (recuerda que las rutas son relativas al enrutador). La estructura deberá quedarte así:

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

let Libro = require(__dirname + '/../models/libro.js');

let router = express.Router();

router.get('/', (req, res) => {
    Libro.find().then(...
    ...
}

// Resto de servicios aquí...
```

```
module.exports = router;
```

4. Crea el archivo "autor.js" dentro de "models" y copia dentro el esquema y modelo para los autores hecho en la sesión 5, exportándolo con `module.exports`.
5. Crea el archivo "autores.js" dentro de "routes" y añade dentro tres servicios básicos para los autores: GET (listar todos), POST (añadir nuevo) y PUT (borrar por *id*)
6. Modifica ahora el archivo "libros.js" de "routes" para que las operaciones POST y PUT permitan añadir y modificar autores y comentarios del libro. En el caso de los comentarios, al modificar (PUT) inicialmente se sobreescribirán los viejos con los que se indiquen. Se deja como opcional que se puedan añadir los comentarios que se indiquen a los que ya hay, sin perder los comentarios antiguos.
7. Modifica el archivo principal "index.js". Para empezar, vamos a renombrarlo a "app.js" (la mayoría de aplicaciones tienen este nombre de archivo principal), y dentro haremos algo similar a lo que hemos hecho para la aplicación de contactos:
    - Cargar (`require`) las librerías de *mongoose*, *express* y *body-parser*.
    - Cargar (`require`) los enrutadores para libros y autores
    - Conectar a la base de datos de libros
    - Inicializar la *app* Express
    - Añadir el *middleware* de *body-parser* para procesar datos en formato JSON
    - Mapear los enrutadores de libros y autores a las rutas */libros* y */autores*, respectivamente
    - Poner en marcha el servidor

    Para este paso, puedes basarte en el archivo "app.js" del ejemplo de contactos que se te proporciona.

Opcionalmente (aunque es recomendable hacerlo), define nuevas pruebas en la colección de Postman para probar los nuevos servicios. En concreto, la colección debería tener pruebas para probar estos servicios:

- GET /libros
- GET /libros/:id
- POST /libros (incluyendo autor y comentarios)
- PUT /libros/:id (incluyendo autor y comentarios)
- DELETE /libros/:id
- GET /autores
- POST /autores
- DELETE /autores/:id

4.2. Ejercicio 2 (opcional)

Vamos a definir dos *middlewares* propios en nuestra aplicación de libros del ejercicio anterior:

- Uno lo definiremos únicamente para el enrutador de libros ("routes/libros.js"), de forma que cada vez que accedamos a un servicio de ese enrutador, se mostrará por consola la fecha, método (GET, POST, etc) y URI solicitada. Para obtener estos dos últimos datos, deberás utilizar las propiedades `req.method` y `req.url` del objeto de la petición (`req`).
- El otro será global (en "app.js"), y deberá enviar por JSON un mensaje de "En mantenimiento", sin permitir acceder a ningún servicio (es decir, que no llame al método `next`). Siempre que este middleware esté activo, la aplicación no funcionará (sólo enviará "En mantenimiento" para cada solicitud que le llegue). Cuando se desactive/comente, la aplicación funcionará con normalidad.