



20

Node S2 | Middlewares

Después de esta lección podrás:

1. Qué es un middleware en Express
2. Cómo funciona un middleware
3. Cómo crear una ruta por defecto y a utilizar el middleware de error
4. Utilizar middlewares de terceros, como Multer

¿Qué es un middleware en Express?

Hasta ahora hemos utilizado el método `use` del objeto Application y los distintos métodos (`get`, `post`, `put`, etc.) que nos ofrece el objeto Router para filtrar las peticiones por ruta, y tanto con uno como con otro teníamos que definir una función manejadora, un callback que siempre tenía dos argumentos (`request` y `response`): pues, bien, en Express, llamamos a estas funciones **middlewares**.



En ingeniería de software, un *middleware* es un tipo de software que interviene en la lógica de intercambio de información entre aplicaciones. Este concepto se suele utilizar, por ejemplo, para definir software que facilita la comunicación entre distintas capas del modelo TCP/IP.

En realidad, aunque muchas veces no vamos a declarar todos, estos callback reciben tres argumentos:

```
function (request, response, next) {  
    next();  
}
```

Ya sabemos para qué sirven `request` y `response` (aunque sea mínimamente), pero ¿qué representa el argumento `next`? Este argumento contiene una función (un callback), que nos permite, o bien dar el relevo a la siguiente función manejadora registrada (al siguiente middleware), o enviar un error al manejador de errores de Express.

Vamos a probar a ejecutar el siguiente código:

```
const express = require('express');  
  
const PORT = 3000;  
const server = express();  
  
server.listen(PORT, function () {  
    console.log(`Server running in http://localhost:${PORT}`);  
});  
  
server.use(function (req, res, next) {  
    console.log('Se ha ejecutado el middleware 1');  
    next();  
});
```

```

server.use(function (req, res, next) {
    console.log('Se ha ejecutado el middleware 2');
    next();
});

server.use(function (req, res, next) {
    console.log('Se ha ejecutado el middleware 3');
    res.send('Respuesta enviada!');
});

```

Si enviamos una petición desde un navegador (`http://localhost:3000`), veremos en la consola que hemos pasado por los tres middlewares y en el último hemos devuelto la respuesta. Prueba a quitar el `next();` en el segundo middleware y verás cómo, al repetir la misma petición, el navegador se queda esperando una respuesta hasta que pasa el tiempo máximo: y es que al quitar el `next`, ni estamos dando el relevo al siguiente middleware, ni estamos devolviendo la respuesta.

Tanto `use` como el resto de métodos que reciben como parámetro un middleware, aceptan varias funciones separadas por comas, así que el anterior ejemplo se podría haber escrito también de la siguiente forma:

```

const express = require('express');

const PORT = 3000;
const server = express();

server.listen(PORT, function () {
    console.log(`Server running in http://localhost:${PORT}`);
});

server.use(
    function (req, res, next) {
        console.log('Se ha ejecutado el middleware 1');
        next();
    },
    function (req, res, next) {
        console.log('Se ha ejecutado el middleware 2');
        next();
    },
    function (req, res, next) {
        console.log('Se ha ejecutado el middleware 3');
        res.send('Respuesta enviada!');
    }
);

```

Recordemos que habíamos dicho que las funciones manejadoras, a partir de ahora, middlewares, se iban registrando en un orden determinado y luego, cuando se recibía una petición, se ejecutaban en ese mismo orden (comprobando antes el filtro de ruta, si lo tuviese). Podemos ver la lista de middlewares registrados (que en la documentación de Express se llama la pila de middlewares) añadiendo el siguiente código al final del anterior ejemplo:

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

const PORT = 3000;
const server = express();

server.listen(PORT, function () {
    console.log(`Server running in http://localhost:${PORT}`);
});

server.use(
    function (req, res, next) {
        console.log('Se ha ejecutado el middleware 1');
        next();
    }, function (req, res, next) {
        console.log('Se ha ejecutado el middleware 2');
        next();
    }, function (req, res, next) {
        console.log('Se ha ejecutado el middleware 3');
        res.send('Respuesta enviada!');
    });

for (const layer of server._router.stack) {
    console.log(layer.handle.toString());
}
```

El código anterior también nos sirve para insistir en un concepto muy importante: cuando ejecutamos nuestra aplicación se registran todos los middleware, pero estos no se ejecutan hasta que se haga una petición al servidor. Cuando arrancamos el servidor, en este ejemplo, en la consola aparecen los middlewares que hemos registrado (siempre precedidos de dos que siempre registra el propio Express), pero para que alguno de estos se ejecute tendremos que hacer una petición a la URL apropiada.

404: página no encontrada

Al recibir una petición, Express repasa los middlewares que tiene en su pila y, de cada uno, va a comprobar que la ruta coincida. Pero ¿qué pasa si ninguna de las rutas de los middlewares registrados coincide con la ruta de la petición?

Después de todos los middleware de ruta deberíamos tener siempre uno último que diese una respuesta por defecto: este sería el que lanzaría el famoso error 404, página no encontrada, si la ruta que se ha pedido no existe.

Vemos un ejemplo:

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

const PORT = 3000;
const server = express();
const router = express.Router();

server.listen(PORT, function() {
  console.log(`Server running in http://localhost:${PORT}`);
});

server.use('/', router);

router.get('/hello', function(req, res) {
  res.send('Hi Upgrader!');
});

router.get('/movie', function(req, res) {
  const movies = ['Harry Potter', 'Titanic', 'Back to the Future'];
  res.send(movies);
});

server.use(function(req, res) {
  res.send('No existe la ruta solicitada en esta API');
});
```

De esta forma, después de comprobar todas las rutas registradas, si ninguna coincide, no se quedará colgada la petición, habrá una respuesta por defecto.

Manejando errores con Express

El middleware que intercepta los errores en Express tiene la peculiaridad de que tiene cuatro argumentos:

```
server.use( function(err, req, res, next) {  
    res.send(err.message || 'Unexpected error');  
});
```

Cuando en algún middleware (ya sea nuestro código o de terceros) se pase un parámetro en la función `next`, este dato se entenderá que es un error y Express salta al manejador de errores por defecto.

Veamos un ejemplo:

```
const express = require('express');  
  
const PORT = 3000;  
const server = express();  
  
server.listen(PORT, function () {  
    console.log(`Server running in http://localhost:${PORT}`);  
});  
  
server.use(function (req, res, next) {  
    console.log('Se ha ejecutado el middleware 1');  
    const error = new Error('Se ha producido un error!');  
    next(error);  
});  
  
server.use(function (req, res, next) {  
    console.log('Se ha ejecutado el middleware 2');  
    res.send('Todo ha salido genial!');  
});  
  
server.use(function (err, req, res, next) {  
    console.log('Se ha ejecutado el middleware manejador de errores');
```

```
    res.send(err.message);
});
```

Al hacer una petición a nuestro servidor después de escribir el código anterior, vemos que al ejecutar `next(error);` saltamos al manejador de errores

Usando un middleware de terceros

Como ejemplo de utilización de un middleware de terceros, vamos a explicar como utilizar Multer, un módulo que nos permitirá subir ficheros a nuestro servidor (por ejemplo, desde un formulario en nuestra web) y guardarlos en una carpeta.

Como primer paso vamos a instalar la librería multer en nuestro servidor (dependencia de pro):

```
$ npm install multer
```

La configuración inicial de Multer la vamos a hacer en un fichero aparte:

```
const multer = require('multer');
const path = require('path');

const upload = multer({
  storage,
  fileFilter,
});

module.exports = { upload };
```

Hemos creado una constante upload que es el resultado de invocar la función `multer()` con dos opciones, pero no estamos enviando nada como opción por ahora. Vamos paso a paso.

Creando un storage

Para crear un objeto de Storage de Multer del tipo que nos permitirá guardar ficheros en disco, tenemos que invocar a la función `multer.diskStorage`: su json de configuración tiene varias opciones pero nosotros nos vamos a centrar por ahora en `filename` y `destination`.

```
const storage = multer.diskStorage({
  filename: function (req, file, cb) {
    cb(null, Date.now() + file.originalname);
  },
  destination: function (req, file, cb) {
    cb(null, path.join(__dirname, '../public/uploads'));
  }
});
```

- Con la opción `filename` estamos invocando a una función que recibirá como parámetros la Request, el archivo que estamos subiendo y un callback. **El callback podrá invocarse con dos argumentos, un error y el nombre del archivo que queremos guardar.**

En nuestro caso, usaremos como nombre la fecha actual más el nombre original del archivo, creando así un nombre único y no replicable, de forma que no pise a los archivos previamente guardados en nuestro servidor.

- Con la opción `destination` crearemos un proceso parecido al anterior, pero en este caso enviaremos como segundo argumento del callback un string que

corresponda con la dirección en disco de la carpeta donde guardaremos los archivos. En este caso, será `/public/uploads`.

Creando un fileFilter

Usaremos un filtro de archivos en nuestro caso porque únicamente queremos permitir a los usuarios la subida de imágenes al servidor y no de archivos de texto o canciones.

Crearemos por tanto una lista de archivos permitidos e invocaremos al callback con un error si la extensión del archivo subido no está admitida en el servidor. En caso contrario simplemente invocaremos el callback sin error y con un Boolean que sea true.

```
const VALID_FILE_TYPES = ['image/png', 'image/jpg'];

const fileFilter = function (req, file, cb) {
  if (!VALID_FILE_TYPES.includes(file.mimetype)) {
    cb(new Error('Invalid file type'));
  } else {
    cb(null, true);
  }
}
```

Una vez configurado todo correctamente, el archivo tendrá el siguiente contenido:

```
const path = require('path');
const multer = require('multer');

const VALID_FILE_TYPES = ['image/png', 'image/jpg'];

const upload = multer({
  storage: multer.diskStorage({
    filename: function (req, file, cb) {
      cb(null, Date.now() + file.originalname);
    },
  },
})
```

```

destination: function (req, file, cb) {
  cb(null, path.join(__dirname, '..', 'public', 'uploads'));
},
}),
fileFilter: function (req, file, cb) {
  if (!VALID_FILE_TYPES.includes(file.mimetype)) {
    cb(new Error('Invalid file type'));
  } else {
    cb(null, true);
  }
},
);

module.exports = { upload };

```

Ahora que tenemos listo nuestro middleware, solo tenemos que aplicarlo a una ruta:

```

const fileMiddlewares = require('../middlewares/file.middleware');

router.post(
  '/create',
  fileMiddlewares.upload.single('picture'),
  function (req, res, next) {
    //response to client
  }
);

```

Antes de ver el contenido del callback de la ruta (o controlador) vamos a explicar que ocurre cuando aplicamos el middleware.

Nuestro middleware es `fileMiddlewares.upload` pero a través de multer se ha añadido la función `.single()` que nos permite indicar que únicamente vamos a subir un archivo. El argumento que enviamos será `'picture'` ya que es el nombre del campo del body que llega a nuestro servidor. Si por ejemplo estuviésemos creando un usuario con avatar, podríamos llamar a este campo `'avatar'`, este nombre lo elegimos nosotros.

Cada vez que multer consiga subir un archivo con éxito (si este existe), adjunta en el objeto request la nueva propiedad `req.file`, que contiene los detalles de nuestra imagen.

El campo `req.file` contendrá los siguientes campos:

```
{  
 fieldname: 'picture',  
originalname: 'image.png',  
encoding: '7bit',  
mimetype: 'image/png',  
destination: '',  
filename: '1588546283237.png',  
path: '',  
size: 460982  
}
```

Para guardar la información dentro de nuestros modelos, usaremos el campo `filename` ya que conocemos que las imágenes están en nuestro servidor dentro de la carpeta `public/uploads` y por tanto podremos utilizarlas con la url <http://localhost:3000/uploads/<FILENAME>>.



Los ficheros que hemos guardado en la carpeta del servidor no deberían ser accesibles sin más desde un navegador, esto sería un gran vulnerabilidad de seguridad. Lo normal es que si queremos dar acceso a ficheros en nuestro servidor usemos un servidor web, por ejemplo, Apache o Nginx, pero si no queremos complicarnos la vida, hay un sencillo módulo que podemos utilizar:

```
$ npm install serve-static
```

Para incluirlo en nuestra aplicación lo único que tenemos que hacer es requerirlo en el fichero en el que vayamos a usarlo:

```
const serveStatic = require('serve-static');
```

Y finalmente lo podremos utilizar como cualquier otro middleware:

```
server.use(serveStatic('public/uploads'));
```

Por supuesto, este módulo tiene infinidad de posibilidades, esto es solo lo más básico: dale un vistazo a la [documentación](#), si quieres profundizar.

EJERCICIO: API Express (II)

Añadamos lo que hemos aprendido en este bloque a nuestro ejemplo de API con Express.js. Además de añadir un middleware de ruta no encontrada y un manejador de errores, añade la posibilidad de que un recurso de tu API tenga una imagen asociada (que se guardará `public/uploads`).