

Apunte 15 - Express - Middleware

Middleware

El concepto de middlewares es popular en muchos frameworks de Desarrollo Web. En el caso particular de Express, este depende fuertemente de esta construcción.

Un Middleware tiene como propósito tomar dos piezas de la aplicación y conectarlas, como un puente en el que fluye la información. Normalmente decimos que una rutina de código tiene como propósito recibir información y retornarla transformada, la única característica especial de un Middleware es que la información la obtiene de otra función de código para luego enviársela a una función distinta más.

Los middlewares en Express se montan por múltiples razones, una de ellas por ejemplo es validar la información antes de que llegue a la rutina que enviará respuesta hacia el cliente, también pueden usarse para hacer una consulta y guardar información antes de que pase a las funciones que responderán.

Un middleware en Express es una función que recibe 3 argumentos:

```
function(req,res,next){ ... }
```

Los primeros dos argumentos, como cualquier función que responde peticiones del cliente contiene la información de la solicitud en el primer argumento Request (req), y el objeto Response (res) como segundo argumento nos sirve para modificar la respuesta que se enviará al cliente.

El tercer argumento (next) es muy utilizado en un middleware de una función de respuesta. Este tercer argumento es una función que contiene el próximo middleware o función que va a ejecutar luego de que la actual termine su ejecución.

Esta función *next* termina la ejecución de nuestro middleware y puede hacerlo de dos formas:

- **Con éxito**, la función next en este caso no recibe argumentos al ejecutarse, indicándole al siguiente punto de la ejecución que todo salió bien.

```
function miMiddleware(req,res,next){  
  next();  
}
```

- **Con un error**, el mismo se envía como argumento de la función, indicando al siguiente punto de la ejecución que algo salió mal y no puede continuar con la respuesta de la petición del cliente.

```
function miMiddleware(req,res,next){
  if(user.permisos !== "admin"){
    next(new Error('No tienes permisos para estar aquí'));
  }
}
```

Estas funciones se montan:

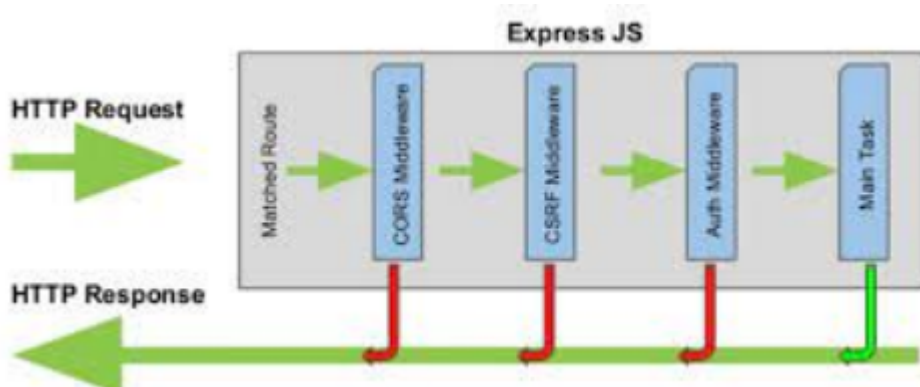
- en el proceso de respuesta a una petición usando el método `use` del objeto `app`. El middleware se ejecuta en el orden en que se define en el código, y se aplica a todas las solicitudes que llegan a la aplicación.

```
const express = require('express');
const app = express();
function miMiddleware(req,res,next){
  console.log('Este es un middleware');
  next();
}
app.use(miMiddleware); //Esto indica que antes de cualquier función de respuesta
se debe ejecutar este middleware
```

- O bien como parte de la respuesta de una ruta:

```
const express = require('express');
const app = express();
function miMiddleware(req,res,next){
  next();
}
app.get('/', miMiddleware, function(req,res){
  //Se ejecutará esta función luego del middleware
});
```

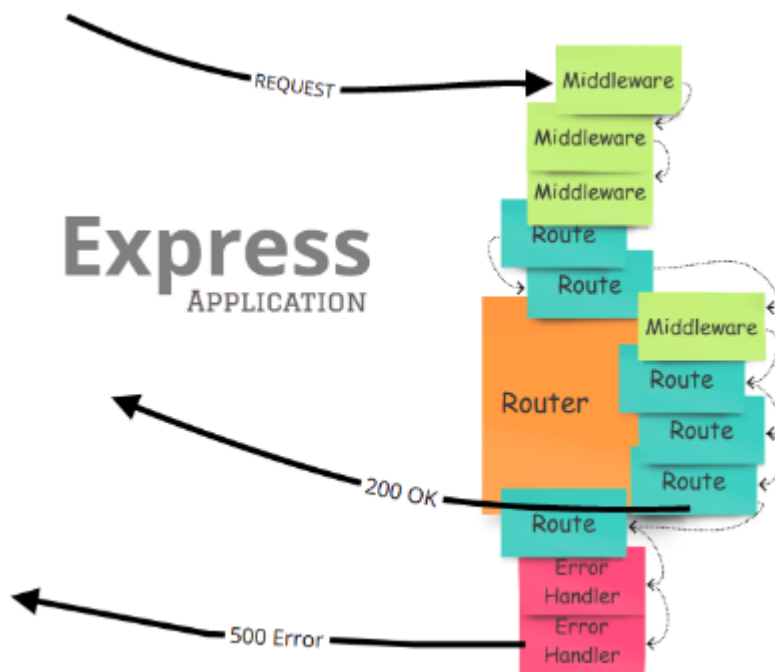
En ambos casos, es posible que podamos colocar cuantos middlewares necesitemos definir, lo importante es que cada uno llame la función **`next()`**, sin argumentos, para que el siguiente middleware se ejecute hasta llegar a la función de respuesta.



A continuación vemos un ejemplo de middleware que servirá para generar un registro (log) de las peticiones del cliente

```
app.use((req, res, next) => {
  let current_datetime = new Date();
  let formatted_date =
    current_datetime.getFullYear() +
    "-" +
    (current_datetime.getMonth() + 1) +
    "-" +
    current_datetime.getDate() +
    " " +
    current_datetime.getHours() +
    ":" +
    current_datetime.getMinutes() +
    ":" +
    current_datetime.getSeconds();
  let method = req.method;
  let URL = req.URL;
  let status = res.statusCode;
  let log = `[${formatted_date}] ${method}:${URL} ${status}`;
  console.log(log);
  next();
})
```

Manejo de Errores



El manejo de errores es uno de los temas más importantes en la programación. Ayuda a los usuarios finales a comprender lo que está mal y hace que el código sea más fácil de corregir y mantener. Como premisa general necesitamos que todos los posibles errores de nuestra aplicación no hagan caer nuestro servidor (daemon, o

servicio) y queden registrados para poder analizarlos y tomar las medidas necesarias para corregirlos o mitigarlos.

El manejo de errores en express se hace con un middleware, que debe definirse al final de otras llamadas a rutas.

```
app.use(function(err, req, res, next) {  
  console.error(err.stack);  
  res.status(500).send('Actualmente tenemos inconvenientes con procesar su solicitud, intente nuevamente mas tarde!');  
});
```

Los errores que ocurren en el código síncrono dentro de los controladores de ruta y el middleware no requieren trabajo adicional. Si el código síncrono arroja un error, Express lo detectará y lo procesará.

```
// función con error síncrono  
app.get('/testerror', (req, res) => {  
  throw new Error('probando desencadenar un error');  
});
```

Pero para el caso de los errores devueltos por funciones asíncronas invocadas por controladores de ruta y middleware, debemos pasarlos a la función **next()**, para que Express pueda detectarlos y procesarlos, caso contrario harán caer nuestro servidor.

```
// función con error asíncrono, con uso de la función next()  
router.get("/api/articulosfamilias-testerror", async function (req, res, next) {  
  try {  
    let items = await db.articulosfamilias.findAll({  
      attributes: ["CampoInexistenteParaGenerarUnError"],  
    });  
    res.json(items);  
  } catch (error) {  
    next(error);  
  }  
});
```

Para evitar tener que capturar todos los errores asíncronos en forma manual, usaremos un middleware específico para tal tarea, seguiremos los siguientes pasos.

1. Instalar paquete: ***npm install express-async-errors***
 2. Modificar todos los métodos asíncronos para que usen `async/await` en lugar de promesas, para que funcione la librería y evitemos que se caiga el servidor web.
- Ejercicio: Modificar el controlador de errores para que registre los errores en un archivo, al que luego solo como desarrolladores tendremos acceso.

Para registrar los errores en un archivo, puedes usar el módulo fs (File System) de Node.js. Ejemplo:

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

app.get('/', function(req, res, next){
  // Simula un error
  const err = new Error('Algo salió mal');
  next(err);
});

// Middleware de manejo de errores
app.use(function(err, req, res, next) {
  // Escribe el error en el archivo de registro
  fs.appendFile('error.log', `${new Date().toISOString()}: ${err.stack}\n`,
    function(err) {
      if (err) {
        console.error('No se pudo escribir en el archivo de registro');
      }
    });

  res.status(500).send('Algo salió mal');
});

app.listen(3000, function(){
  console.log('Aplicación escuchando en el puerto 3000');
});
```

En este ejemplo, cuando ocurre un error, se registra en un archivo llamado error.log con una marca de tiempo. Si el archivo no existe, fs.appendFile lo creará. Si ya existe, añadirá el error al final del archivo.

Tené en cuenta que debes tener los permisos adecuados para leer y escribir en el archivo error.log. Además, este es un ejemplo básico y en un entorno de producción, es posible que desees utilizar una biblioteca de registro más sofisticada como Winston o Bunyan que proporcionan características como el registro en varios medios, el control de los niveles de registro, etc.

- Ejercicio: Modificar las validaciones que hacemos con Sequelize, tanto en el alta como en la modificaciones, encapsulándola en una única función común a ambas (alta y modificación).

Excepciones no controladas

¿Qué sucede cuando ocurre un error regular y no es detectado por try..catch?: El script finaliza con un mensaje en la consola. El mismo puede ser capturado por el evento de Node **"uncaughtException"** que se emite precisamente cuando una excepción JavaScript no capturada vuelve al ciclo de eventos de node.js. Mediante este evento no podremos evitar que la aplicación se caiga pero nos permitirá registrar la causa de la misma, para analizarla posteriormente.

```
import process from 'node:process';
process.on('uncaughtExceptionMonitor', (err, origin) => {
  console.log('Olvidamos manejar este error en el código: ', error);
  process.exit(1); // salir de la aplicación
});
```

Análogamente existe el evento **"unhandledRejection"**, que se emite cuando una Promesa es rechazada y no hay ningún catch de errores adjuntado a la misma dentro del bucle de eventos de node.js. El cual también deberíamos usar.

```
process.on('unhandledRejection', (error, promise) => {
  console.log(' Olvidamos de manejar este error en la promesa', promise);
  console.log(' El error fue: ', error );
});
```

Como vemos siempre existe que la posibilidad que nuestra aplicación se interrumpa por algún tipo de error o malfuncionamiento, para salvar esta situación cuando publiquemos en un entorno de producción nos valdremos de algún gestor de procesos que nos dará otra capa más de protección para asegurar el funcionamiento de nuestro servidor.

Gestores de procesos para las aplicaciones Express en producción

Cuando ejecuta aplicaciones Express en producción, es muy útil utilizar un **gestor de procesos** para realizar las siguientes tareas:

- Reiniciar la aplicación automáticamente si se bloquea o se interrumpe.
- Obtener información útil sobre el rendimiento en tiempo de ejecución y el consumo de recursos.
- Modificar dinámicamente los valores para mejorar el rendimiento.
- Controlar la agrupación en clúster.

Un gestor de procesos es una especie de servidor de aplicaciones: un "contenedor" de aplicaciones que facilita el despliegue, proporciona una alta disponibilidad y permite gestionar la aplicación en el tiempo de ejecución.

Los gestores de procesos más conocidos para Express y otras aplicaciones Node.js son los siguientes:

- StrongLoop Process Manager
- PM2
- Forever

Nota: En nuestros ejemplos cuando publiquemos nuestro paso a paso en Microsoft Azure

mediante la funcionalidad "App Service Linux, stack Node", esta nos proveerá automáticamente del gestor de proyectos PM2. Azure App Service le permite crear y hospedar aplicaciones web, back-ends móviles y API RESTful en varios lenguaje disponibles de programación sin tener que administrar la infraestructura.

