

Bienvenidos

Clase 08.
Aplicaciones Móviles y Cloud Computing

API RestFull

Temario

07

Desarrollo FullStack

- ✓ Arquitectura REST
- ✓ CRUD con NodeJS y Express
- ✓ Middlewares / Routers

08

Desarrollo de aplicaciones Restfull

- ✓ [Arquitectura REST](#)
- ✓ [CRUD con NodeJS y Express](#)
- ✓ [Middlewares / Routers](#)

09

Desarrollo de aplicaciones Restfull

- ✓ WebSockets
- ✓ Motores de Plantillas
- ✓ Dependencias socket.io y express-handlebars



Objetivos de la clase

- Comprender los principios básicos de funcionamiento de una API-RESTFull
- Introducción a Node. Introducción a ExpressJS. Desarrollo de un CRUD con persistencia en memoria con ExpressJS
- Middlewares / Routers



Comprendiendo una API REST



API (Application Programming Interface)

Es un conjunto de definiciones y reglas que permiten que dos equipos puedan integrarse para trabajar juntos. La mejor analogía que hay para comprender ésto es que una API trabaja como un "contrato" entre el front y el back.

La API permite entonces que se respondan preguntas como:

- ✓ ¿A qué endpoint debo apuntar para la tarea que necesito?
- ✓ ¿Qué método debo utilizar para ese recurso?
- ✓ ¿Qué información debo enviar para realizar correctamente mi petición?



El cliente necesita algo del servidor, por lo que tiene que realizar una petición (request)

Para que la petición llegue correctamente al servidor, deberá apuntar al endpoint correcto, con el método correcto, con la información correcta

El servidor recibe la petición. Si se cumplieron todas las especificaciones de la API, el procesamiento se podrá llevar a cabo con éxito

Petición

Procesamiento



Respuesta

Resultado

El cliente, al haber cumplido con lo que especificaba la API, podrá obtener su resultado satisfactoriamente y utilizarlo.

Cumplir con el contrato de la API asegura (en la mayoría de los casos) que habrá un resultado satisfactorio



REST

Ya tenemos las reglas para comunicarse, ¿Pero qué tal la estructura del mensaje? Cuando hacemos una petición o cuando recibimos una respuesta, ésta debe tener un **formato**. REST (**RE**presentational **State** **T**ransfer) permite definir **la estructura** que deben tener los datos para poder transferirse.

La API respondía a preguntas sobre cómo comunicarse correctamente. En cambio, REST define cómo debe ser el cuerpo del mensaje a transmitir. Puedes llegar a hablar con el presidente si cumples con el protocolo (HTTP) y las reglas (API), pero ¿de qué nos servirá si la forma en que estructuramos nuestro mensaje (REST) no es correcta?



Los dos formatos más importantes son **JSON** y **XML**. La utilización de la estructura dependerá de las necesidades del proyecto. Nosotros utilizaremos **JSON**. Como notarás, ¡un JSON parece un objeto! así que es mucho más amigable la sintaxis.

✓ XML

```
<factura>
  <cliente>Gomez</cliente>
  <emisor>Perez S.A.</emisor>
  <tipo>A</tipo>
  <items>
    <item>Producto 1</item>
    <item>Producto 2</item>
    <item>Producto 3</item>
  </items>
</factura>
```

✓ JSON

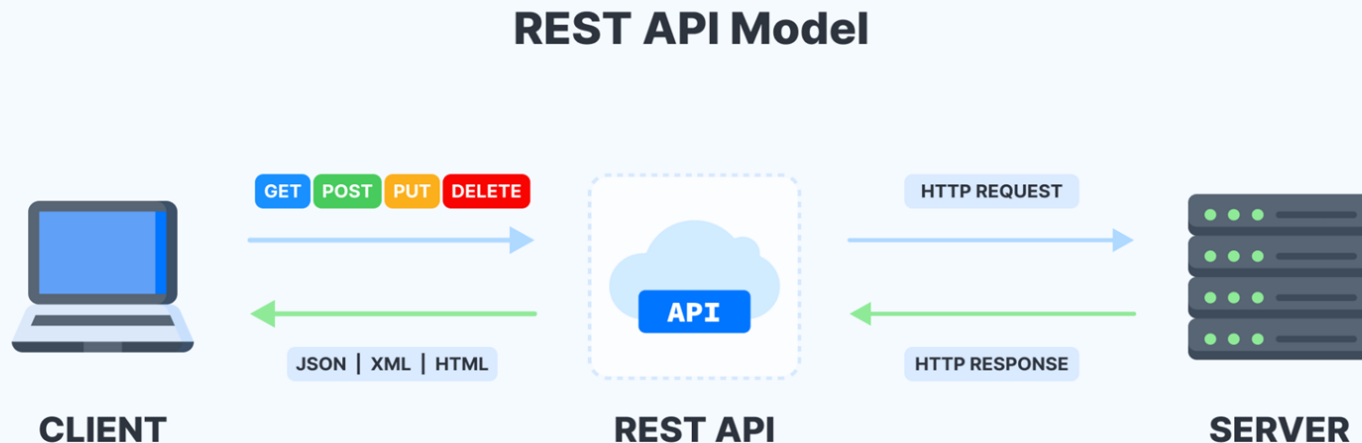
```
{
  "cliente": "Gomez",
  "emisor": "Perez S.A.",
  "tipo": "A",
  "items": [
    "Producto 1",
    "Producto 2",
    "Producto 3"
  ]
}
```

Entonces una API REST es...

Un modelo completo para tener perfectamente estipulados los protocolos, las reglas, e incluso la estructura de la información, con el fin de poder hacer un sistema de comunicación completo entre las computadoras.



Modelo de una API REST



**¿Qué características
debe tener una API REST?**



Arquitectura Cliente-Servidor sin estado

- ✓ Cada mensaje HTTP contiene toda la información necesaria para comprender la petición.
- ✓ En consecuencia, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre mensajes.
- ✓ Esta restricción mantiene al **cliente** y al **servidor débilmente acoplados**: el cliente no necesita conocer los detalles de implementación del servidor y el servidor se “despreocupa” de cómo son usados los datos que envía al cliente.



Cacheable

- ✓ Debe admitir un sistema de almacenamiento en caché.
- ✓ La infraestructura de red debe soportar una caché de varios niveles.
- ✓ Este almacenamiento evita repetir varias conexiones entre el servidor y el cliente, en casos en que peticiones idénticas fueran a generar la misma respuesta.



Operaciones comunes

- ✓ Todos los recursos detrás de nuestra API deben poder ser consumidos mediante peticiones HTTP, preferentemente sus principales (POST, GET, PUT y DELETE).
- ✓ Con frecuencia estas operaciones se equiparan a las operaciones CRUD en bases de datos (en inglés: Create, Read, Update, Delete, en español: Alta, Lectura, Modificación, y Baja).
- ✓ Al tratarse de peticiones HTTP, éstas deberán devolver con sus respuestas los correspondientes códigos de estado, informando el resultado de las mismas.



Interfaz uniforme

- ✓ En un sistema REST, cada acción (más correctamente, cada recurso) debe contar con una URI (Uniform Resource Identifier), o identificador único.
- ✓ Ésta nos facilita el acceso a la información, tanto para consultarla, como para modificarla o eliminarla, pero también para compartir su ubicación exacta a terceros.



Node js

Surgió de la necesidad de ejecutar javascript fuera del navegador, y ha crecido hasta convertirse en uno de los elementos principales para el desarrollo web.

Cuenta con el mismo motor V8 de Google Chrome, el cual permite convertir el código javascript a código máquina para poder ser procesado correctamente.

Además, cuenta con muchas funcionalidades internas del mismo lenguaje javascript gracias a sus ajustes con ECMAScript.



Módulos nativos en Nodejs

fs

Módulo utilizado para manejo de archivos

Sirve para manejar otro modelo de persistencia.

crypto

Permite hacer operaciones de encriptación y cifrado para información sensible

Sirve para mejorar la seguridad de los datos

http

Permite crear un servidor básico bajo el protocolo http

Sirve para crear nuestro primer servidor de solicitud/respuesta

path

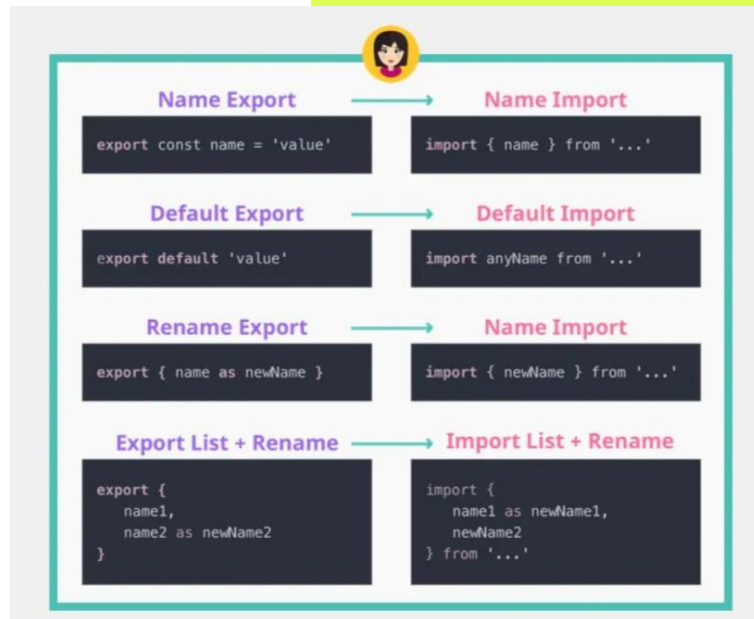
Permite el correcto manejo de rutas

Sirve para evitar ambigüedad al trabajar con rutas



Manejo de módulos en Node:

- ✓ **JavaScript nativo:** previo al lanzamiento de ES6, mediante scripts separados, invocados en orden desde nuestro index.html
- ✓ **CommonJS:** en 2009 nace NodeJS, y propone este esquema para manejo de módulos. Es el que utiliza la sintaxis require y module.exports
- ✓ **ES Modules:** En 2015, con el lanzamiento de ES6, surge este nuevo esquema de manejo de módulos, que utiliza la sintaxis import / export



Códigos de estado

Cuando realizamos alguna petición al servidor mediante el protocolo HTTP, el servidor debe respondernos **no sólo con información**, sino con un estado del proceso. Este es un código que nos permitirá saber cómo se encuentra el proceso, o cómo finalizó.

HTTP Status Codes



Códigos de estado HTTP

¿Cómo funciona?

Cuando el servidor responde con un código de estado, esto permite saber qué ocurrió con la consulta que estábamos haciendo, y da información al cliente sobre qué ha ocurrido.

- ✓ **1xx : Status "informativo"**
- ✓ **2xx : Status "ok"**
- ✓ **3xx: Status de redirección.**
- ✓ **4xx: Status de error de cliente.**
- ✓ **5xx: Status de error en servidor.**

HTTP STATUS CODES	
2xx Success	
200	Success / OK
3xx Redirection	
301	Permanent Redirect
302	Temporary Redirect
304	Not Modified
4xx Client Error	
401	Unauthorized Error
403	Forbidden
404	Not Found
405	Method Not Allowed
5xx Server Error	
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout

Algunos de los más importantes

- ✓ **200:** Indica que la petición se procesó correctamente. No hubo ningún tipo de inconveniente desde la consulta hasta la respuesta.
- ✓ **3xx:** Hace referencia a redirecciones. Cuando un recurso se ha movido o necesitamos apuntar a otro servicio.
- ✓ **400:** Se utiliza cuando el cliente realiza alguna petición que no cumpla con las reglas de comunicación (una mala consulta, tal vez le faltó enviar un dato, o venía en un formato incorrecto).
- ✓ **401:** Se utiliza cuando el cliente no se ha identificado con el servidor bajo alguna credencial, y por lo tanto no puede acceder al recurso



Algunos de los más importantes

- ✓ **403:** Se utiliza cuando el cliente ya se identificó, pero sus credenciales no tienen el nivel de privilegios suficiente para acceder al recurso. Es el equivalente a decir "Ya qué quién eres, pero no tiene acceso a este recurso"
- ✓ **404:** Se utiliza cuando el recurso no se ha encontrado, ya sea algún dato solicitado o incluso el endpoint mismo.
- ✓ **500:** Se utiliza cuando algo ocurrió en el servidor. No necesariamente un error del cliente, sino un error o "detalle" que no haya considerado el servidor al tratar con algún caso.



Métodos de petición

Un método es una definición que forma parte del protocolo HTTP, el cual nos sirve para canalizar el tipo de petición que estoy realizando sobre un cierto endpoint. De esta manera, el cliente puede llamar al mismo endpoint, **pero con diferentes métodos**, indicando qué operación quiere realizar con dicho recurso. Los principales métodos son:

- ✓ GET: Obtener un recurso
- ✓ POST: Crear o añadir un recurso
- ✓ PUT: Modificar un recurso
- ✓ DELETE: Eliminar un recurso



¿Qué es Express js?

Express js es un framework minimalista que permitirá desarrollar servidores más complejos.

Express nos facilitará:

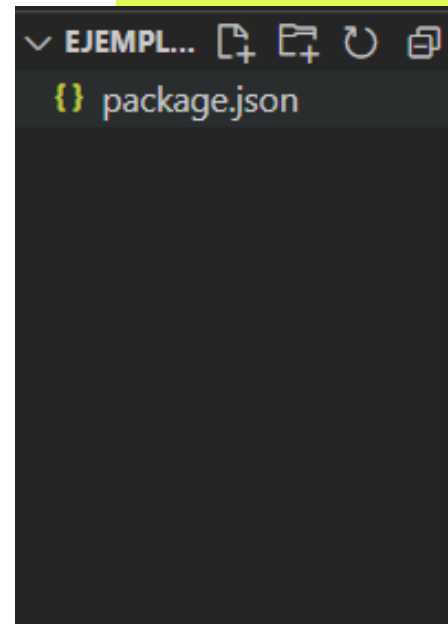
- ✓ Utilizar diferentes rutas para las peticiones.
- ✓ Mejorar la estructura de nuestro proyecto.
- ✓ Manejar funcionalidades más complejas y utilización de middlewares.



Paso 1: npm init -y

Express no es nativo de nodejs, por lo tanto, necesitaremos primero contar con un **package.json** para gestionar las dependencias a instalar.

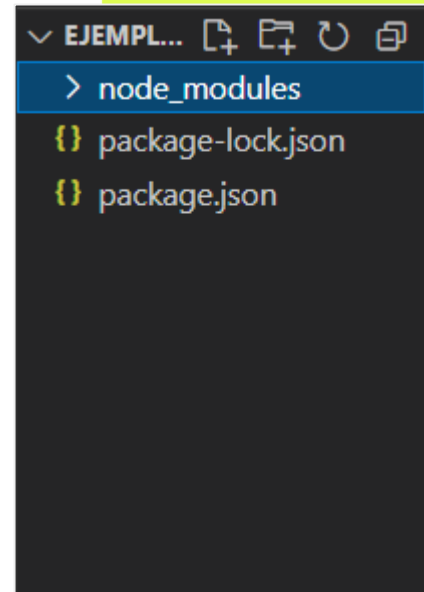
Una vez que tenemos package.json en nuestra carpeta, podemos continuar instalando dependencias.



Paso 2: npm install express

Procedemos a instalar de manera local express js. Al ejecutar este comando, notaremos cómo se genera una carpeta `node_modules`, que es donde se encuentra almacenado express

A partir de este punto, ya contamos con la estructura elemental instalada. El resto es más “flexible”.



Paso 3: Estructurar el proyecto

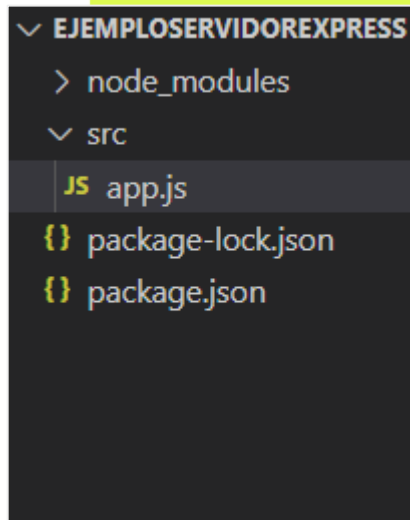
Se recomienda tener una carpeta src, donde vivirá todo nuestro código, dentro de la cual crearemos un archivo con el nombre "app.js"

Finalmente, el archivo app.js ya puede importar la dependencia instalada de **express js**, ya sea por commonjs:

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

o bien por module (recordar colocar el type:"module" en package.json):

```
import express from 'express';
```



Objeto request

Al hacer los endpoints, hasta el momento sólo hemos utilizado el elemento “res”, el cual utilizamos para responder a una petición.

Sin embargo, estos últimos ejercicios hemos dejado de lado el “req” que viene a su izquierda. Ahora abordaremos qué utilidad tiene y cómo podemos utilizarlo.

El objeto req cuenta con tres propiedades principales: **req.query**, **req.params**, **req.body**.



Router en Express

Al diseñar un servidor nos encontramos con un problema: comienzan a aparecer rutas “iguales” que sólo diferían en métodos. Esto puede resultar confuso y engorroso.

Uno podría estar trabajando con una clase o entidad “usuarios”, ¿pero qué pasaría si tuviéramos...?

- ✓ Usuarios
- ✓ Productos
- ✓ Tickets
- ✓ Eventos
- ✓ Membresías
- ✓ Transportes
- ✓ Sucursales

¿Cuántos métodos atiborrados tendríamos en un solo archivo?

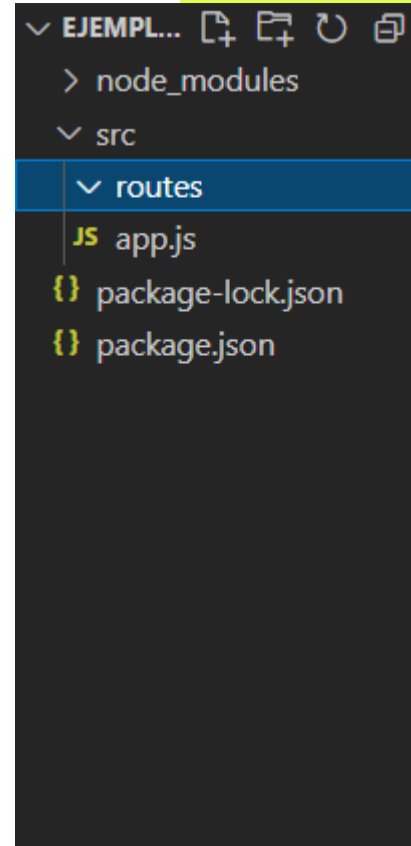
Un router en express nos permitirá separar los endpoints “comunes” en entidades separadas que actuarán como “mini aplicaciones”, las cuales tomarán peticiones que concuerden con dicho endpoint y las redireccionaran a esta mini aplicación.

De esta forma, nuestro código resultará más organizado, y las diferentes entidades tendrán **aislado** el comportamiento interno, como configuraciones, middlewares, etc.



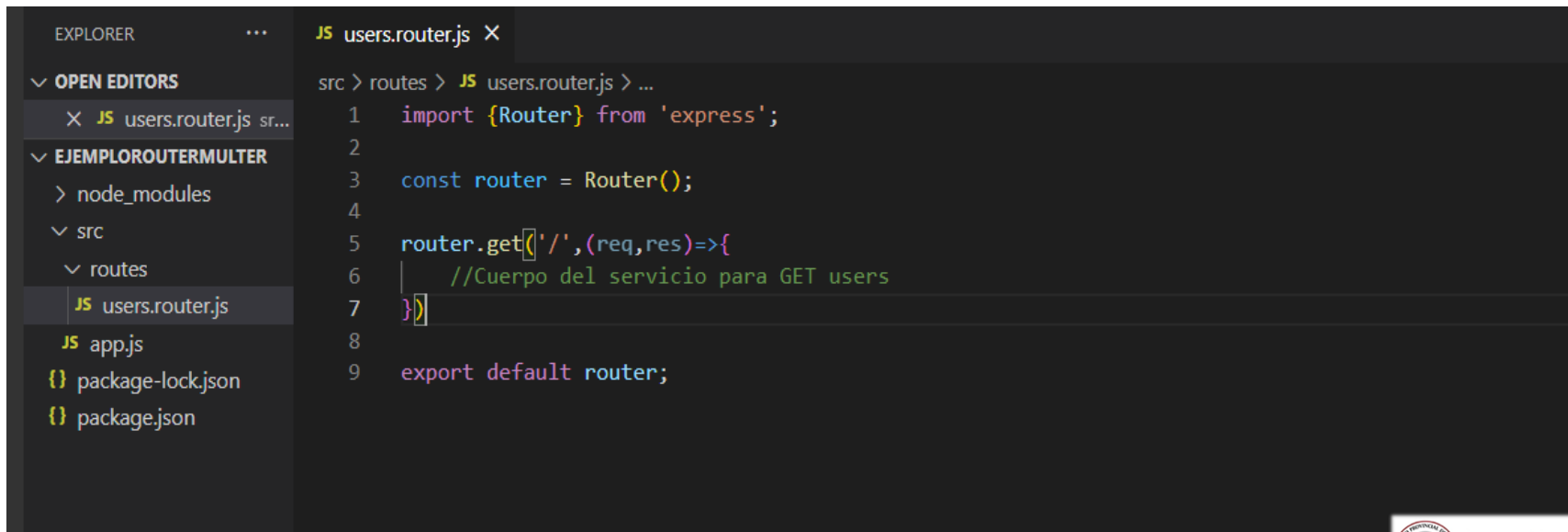
¿Cómo aplicar un router?

Ahora, agregaremos una carpeta “routes” donde vivirán nuestros diferentes routers (Nota que app.js se queda fuera de routes, pero sigue dentro de src).



Cuerpo de un router

Para agregar un router a nuestra nueva carpeta, éste debe contener la siguiente sintaxis:



The screenshot shows a code editor with a sidebar on the left and a main editor area on the right. The sidebar, titled 'EXPLORER', shows a file tree with the following structure:

- EXPLORED
- OPEN EDITORS
 - JS users.router.js sr...
- EJEMPLO ROUTER MULTER
 - node_modules
 - src
 - routes
 - JS users.router.js (selected)
 - JS app.js
 - package-lock.json
 - package.json

The main editor area shows the content of the selected file, `users.router.js`. The code is as follows:

```
src > routes > JS users.router.js > ...
1  import {Router} from 'express';
2
3  const router = Router();
4
5  router.get('/', (req, res) => {
6    //Cuerpo del servicio para GET users
7  })
8
9  export default router;
```



Archivos estáticos

¿Cómo funciona?

- ✓ Nuestro servidor tiene la posibilidad de alojar recursos que pueden ser visibles para el cliente de manera directa.
- ✓ Podemos configurar una carpeta para que el usuario pueda acceder y ver dichos recursos de manera directa sólo con acceder a la ruta donde se encuentra ubicada.
- ✓ En este curso y en proyectos profesionales podrás encontrar estos archivos en la carpeta "public", haciendo referencia como dice el nombre, a recursos públicos de fácil acceso para el cliente.

```
> node_modules
✓ public
  index.html
  logo-node.png
  index.js
  package-lock.json
  package.json
```

Archivos estáticos

¿Cuándo utilizarlos?

Los dos casos principales para los cuales encontrarás un uso para esta carpeta “public” que guarda archivos estáticos son:

- ✓ Cuando necesitemos alojar imágenes y servirlos directamente al cliente.
- ✓ Cuando necesitemos alojar una página web en todos sus sentidos: html, css, js. En esta clase haremos una página sencilla para mostrar el alcance de public.

```
> node_modules
✓ public
  index.html
  logo-node.png
  index.js
  package-lock.json
  package.json
```

¿Cómo convertir una carpeta en un recurso estático?

Para poder utilizar los recursos de una carpeta de manera estática, basta con que en el servidor especifiquemos como “express.static” dicha carpeta con la siguiente sintaxis:

```
app.use(express.static('public'))
```

Indicamos que, todo lo que viva en la carpeta public, podrá ser accedido directamente desde la carpeta *public*.



¿Qué es un middleware?

Seguramente te has dado cuenta de que que hemos utilizado mucho la sintaxis **app.use**. ¿Qué pasa de manera interna en este punto?

Cada vez que utilizamos un **app.use** estamos utilizando un **middleware**. Éstas son operaciones que se ejecutan de manera intermedia entre la petición del cliente, y el servicio de nuestro servidor.

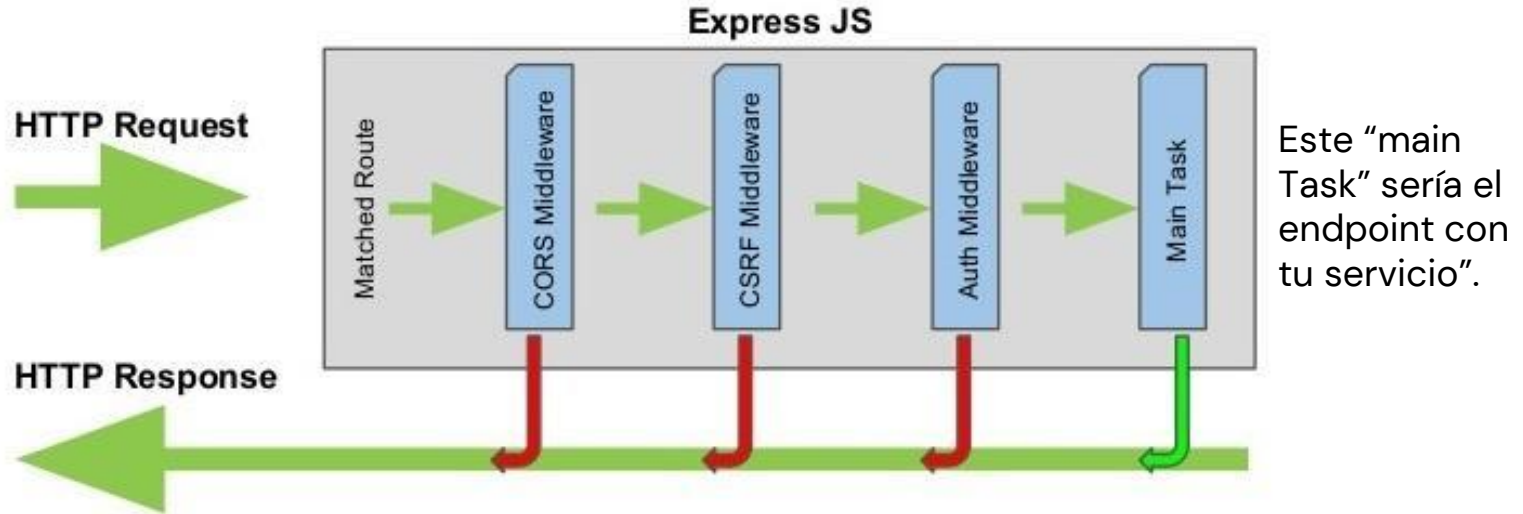
Como lo indica el nombre: “middleware” hace referencia a un intermediario. **Siempre se ejecuta antes de llegar al endpoint que corresponde.**

Podemos utilizar un middleware para:

- ✓ Dar información sobre las consultas que se están haciendo (logs)
- ✓ Autorizar o rechazar usuarios antes de que lleguen al endpoint (seguridad)
- ✓ Agregar o alterar información al objeto **req** antes de que llegue al endpoint (formato)
- ✓ Redireccionar según sea necesario (router)
- ✓ En ciertos casos, finalizar la petición sin que llegue al endpoint (seguridad)



Flujo de múltiples middlewares a través de una petición



¡Importante!

Como lo ves en el diagrama anterior, los middlewares se ejecutan EN ORDEN, eso quiere decir que, si algún middleware depende de que se haya realizado otra operación ejecutada por un middleware previo, debemos colocarlos en cascada según prioridad.

Middleware de nivel de aplicación

```
const app = express();

app.use(function (req, res, next) {
  console.log('Time:', Date.now());
  next();
});
```

Este ejemplo muestra una función de middleware sin ninguna vía de acceso de montaje. La función se ejecuta cada vez que la aplicación recibe una solicitud.



¿Qué es multer?

Multer es un **middleware de terceros**, pensado para poder realizar carga de archivos al servidor.

En ocasiones el cliente necesitará subir una imagen, un vídeo o un archivo, según sea nuestra aplicación. Ello nos lleva a configurar nuestro servidor para soportar estos archivos y poder almacenarlos en donde nosotros le indiquemos.

Al ser de terceros, necesitaremos instalarlo para poder utilizarlo.



¿Preguntas?



Universidad
Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región

Muchas gracias.



Universidad
Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región