Express.js

- ¿Que es Express.js?
- Middleware
- Trabajo con peticiones y respuestas
- Routing
- Retornar paginas HTML (archivos)

Useful resources:

Express.js Official Docs: https://expressjs.com/en/starter/installing.html

1. ¿Qué es Express.js?

Escribir la lógica de un servidor es bastante complejo. Utilizamos Express para hacer el trabajo sucio y ocuparse de todos los detalles, normalmente además de Express vamos a usar otras dependencias para hacer diversas tareas.

La idea es enfocarse en la lógica de negocios y no en los pequeños detalles de la app.

Express es muy flexible y no añade muchas funcionalidades out-of-the-box pero nos da ciertas formas de trabajar con la aplicación que lo hace muy extensible, por lo que hace que tenga miles, cientos de miles de paquetes que se pueden agregar.

Un Framework son herramientas y funciones que siguen ciertas reglas, nos ayuda a construir una aplicación.

Alternativas a Express: Vanilla Node.js, Adonis.js, Koa, Sails.js, etc.

2. Instalando Express

"npm install --save express". Utilizamos "--save" ya que es no solamente lo vamos a utilizar como una herramienta durante el desarrollo, será una parte integra de la aplicación una vez lo enviemos a producción. Por lo tanto, tiene que estar en cualquier computadora que corra nuestra app.

const express = require("express"); Importo el modulo

const app = express(); Express exporta de por si una funcion. Inicializa un nuevo objeto donde Express va a guardar y manejar muchas cosas detrás de escena.

const app = express(); app es un valido manejador de peticiones.

3. Añadiendo un Middleware

Express utiliza muchísimo los Middleware. Los Middleware toman las peticiones y las pasan por una especie de embudo de funciones hasta devolver una respuesta. En vez de tener un manejador de peticiones vamos a tener muchísimas funciones enganchadas donde las peticiones van a pasar hasta que enviemos una respuesta. Esto permite dividir el código en piezas múltiples.

app.use(); Después de definir la función de express() pero antes de pasárselo al parámetro donde creamos el servidor.

Esta función nos permite añadir una función Middleware. Es muy flexible, por ejemplo, permite enviarle un array de manejadores de peticiones, una función, etc.

Todas estas funciones que le pasamos se ejecutarán por cada petición entrante, y la función recibe 3 argumentos: petición, respuesta y siguiente.

- Petición: Recibe la petición con algunas características extras.
- Respuesta: Envía una respuesta con algunas características extras.
- Siguiente: Es una función callback que pasa Express, permite que la petición viaje al siguiente middleware.

Si no llamamos a next/siguiente la petición muere en esa función, por lo tanto deberíamos enviar una respuesta.

Ejemplo:

```
app.use((req, res, next) => {
    console.log("In the middleware");
    next(); -> Pasa por el embudo al siguiente middleware
});
```

4. Como funcionan los Middleware

Express no envía repuestas por defecto ni nada parecido. En el middleware podemos enviar una respuesta con el objeto que recibe el middleware response.

En vez de setear un header o utilizar write (que podemos seguir utilizando) Express hace todo más simple con la función send(BODY). El Body puede ser HTML y los headers son seteados automáticamente por Express.

5. Express - Un vistazo detrás de escena.

En el repo podemos ver como hace Express para setear los headers de manera automática.

Con app.listen(PUERTO) podemos reemplazar la necesidad de importar el módulo http y crear un server.

6. Manejando diferentes rutas

Existen múltiples overloads de app.use(). El que veremos en esta sección es el siguiente: app.use("RUTA", "CALLBACK");

Hay que tener en cuenta que si pongo "/" como una ruta le estamos diciendo que tiene que empezar con esa ruta, pero no indicamos como debe terminar. Por ello si colocamos este PATH primero si pongo / o /ALGO siempre va a entrar en ese Middleware. Por esta razón el / va siempre ultimo como PATH. Del más específico al menos especifico.

Aclaración: Lo del / que siempre va a buscarlo es únicamente si utilizamos el .use(). Si especificamos el HTTP method a .get("/") va a buscar que matchee perfectamente, y si ponemos cualquier otra burrada el navegador nos va a dar un error en caso de no encontrar ese PATH, el use() encapsulaba todas las posibilidades de poner después del "/" y te redireccionaba al index. Aun así, el orden es importante. El use("/", CALLBACK). (O simplemente el callback ya que por defecto es /) nos puede servir para que esas páginas que no se manejan vayan a un error 404 en vez de mostrar un mensaje del navegador. Puedo tener el mismo PATH si los métodos http son distintos.

Ejemplo

```
app.use("/add-product",(req, res, next) => {
    res.send("<h1>Add Product</h1>");
});
app.use("/",(req, res, next) => {
```

```
    res.send("<h1>Hello from Express</h1>");
});
```

7. Parsing de Peticiones Entrantes

https://stackoverflow.com/questions/25471856/express-throws-error-as-body-parser-deprecated-undefined-extended

ACLRACION: En el curso se usa el paquete body-parser pero Express ya trae uno, por lo tanto no voy a instalar el paquete.

Express añade varias nuevas capas para hacer nuestra vida más fácil. Una de ellas es req.body. Esta función nos da el body de toda la data que entra por la petición SIN PARSEAR.

Para parsear la data tenemos que agregar un middleware (que ira antes de todos los middelware que se encarguen de ruteo) y en ese middleware parsear la data.

Para parsear Express nos trae una función llamada urlencoded("CONFIGURACIONES"). Añadimos la línea app.use(express.urlencoded("CONFIGURACIONES")); -> Este código se va a ejecutar SIEMPRE que el servidor registre un listen ya que el middleware no tiene un PATH predefinido y siempre llama a next() implícitamente, por lo tanto en todo momento va a estar parseando un body de un form en POST.

No va a funcionar con JSON, archivos, body de peticiones GET, etc. Se verá más adelante pero para esta parte del curso por el momento lo vamos a dejar así.

La función redirect("PATH") para redireccionar.

8. Limitando Middleware a peticiones POST

Si queremos que un PATH de un Middleware se ejecute solo cuando cierto método se ingresa (un POST x ejemplo) deberíamos usar en vez de app.use, app.get, app.post, app.put, etc. De esta manera filtramos de mejor manera que espero recibir cuando se ingresa a ese path, por lo tanto en nuestro ejemplo el /product en vez de usar el app.use() va a pasar a utilizar el app.post().

9. Usando el Express Router

Normalmente vamos a querer dividir nuestro routing en múltiples archivos, dividir la lógica en diferentes archivos y hacer las importaciones necesarias allí.

El código que se debería ejecutar para diferentes PATHS y http methods se guarda en una carpeta que por convención se llamará routes.

Express nos trae un Router que podemos asignar a una variable: express.router(); Este router es como una mini express app que puede ser "pegada" en otras express app.

En vez de usar app.use/get/put etc vamos a utilizar el router para manejar este tipo de ruteos. Exportándolo vamos a poder llamarlo en el servidor. La referencia a la función de router que exportamos es válido como un Middleware, por lo tanto en el archivo principal puedo hacer el .use() con el router exportado. El orden sigue importando.

Ejemplo:

En la carpeta routes el archivo que es para el admin:

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

router.get("/add-product",(req, res, next) => {

    res.send("<form action='/product' method='POST'><input type='text' name='title'><input type='submit' value='Add Product'></form>")

});

router.post("/product", (req, res)=> {

    console.log(req.body)

    res.redirect("/");

})
```

En el app.js o archivo donde levantamos el servidor:

```
const adminRoutes = require("./routes/admin");

app.use(adminRoutes);
```

10. Añadiendo una pagina de error 404

Como aclare anteriormente, podemos usar el .use() que por default se setea en "/" para que si un usuario ingresa un PATH invalido vaya a un error genérico.

Ejemplo:

```
app.use((req, res, next) => {
   res.status(404).send("<h1>Page not fund</h1>");
})
```

11. Filtrando PATHS

Algunas veces estos routers organizados en archivos (routes para admin, routes para user, routes general, etc) tiene un punto de partida en común (/admin/X, /user/X, etc).

Para no tener que repetir código por cada /admin /users o lo que sea, podemos enganchar en el archivo donde importamos nuestros ruteos en el .use() y como primer parámetro pasarle el PATH en común que van a tener.

Ejemplo: app.use("/admin", adminRoutes);

12. Creando páginas HTML

Por convención toda la vista que le devolvamos al cliente irá en la carpeta "views". Durante el curso veremos la estructura Modelo Vista Controlado (MVC), de esta manera nos vamos acercando poco a poco al tema.

13. Servir paginas HTML

En vez de generar código HTML en el ruteo con res.send() vamos a proporcionar el HTML ya hecho en la página Views.

Utilizaremos el res.sendFile("PATH") para enviar el archivo HTML. Por defecto genera cabeceras dependiendo del tipo de archivo como lo hacía Express al escribir manualmente el HTML, aun así lo vamos a poder modificar si queremos.

El PATH tiene que ser absoluto, pero a la hora de generarlo nos puede dar problemas. Si utilizamos el "/" no va a tomar como root la carpeta del proyecto, sino la del Sistema Operativo. Y si utilizamos "./" para que sea la del root nos pide que sea un PATH absoluto. Para solucionar este inconveniente vamos a importar un core module de Node llamado "path".

Path nos da un función llamada path.join(), join nos da un path al final pero construye el PATH concatenando los diferentes segmentos. El primer argumento que debe recibir es el __dirname (es el PATH absoluto del S.O hasta la carpeta donde lo llamamos), el segundo será la carpeta donde queremos ingresar (en el caso del ejemplo "views"), tenemos que tener en cuenta que __dirname va a ir al directorio donde lo llamamos, por lo tanto, si queremos regresar tenemos que ir X niveles hacia atrás y el tercero es el archivo.

Ejemplo: res.sendFile(path.join(__dirname, "../views", "shop.html")); O res.sendFile(path.join(__dirname, "../", "views", "shop.html"));

No agregamos / porque al concatenar el join() va a tener en cuenta el S.O para agregar los / y lo hace de manera automática la función. (Unix tiene paths /algo/algo2 y Windows tiene \algo\algo2)

Podríamos construir el PATH concatenando manualmente estos tres paths, pero join() ya se encarga de eso y además toma los distintos sistemas operativos, por lo que hacerlo manualmente puede llevar muchísimo trabajo.

14. Usando Funciones de Ayuda para la Navegación

En el path.join() Deberíamos usar ".." en vez de "../" si le pasamos cuatro argumentos como vimos en el siguiente ejemplo: res.sendFile(path.join(__dirname, "../", "views", "shop.html")); Esto sería preferible a pesar de que la ../ debería funcionar en Unix y Windows ya que no asumimos el separador que se utiliza para construir un PATH.

Hay una mejor manera de implementar esto. Podríamos crear una carpeta llamada helpers/util que nos ayude creando una función que nos traiga el directorio padre en vez de utilizar el __dirname. La función sería exportada directamente en la línea de exports. Ejemplo: module.export = path.dirname(require.main.filename);

dirname retorna el nombre del directorio de un PATH, el main se refiere al módulo principal de inicio de aplicación, y filename el archivo encargado de correr ese modulo principal.

Básicamente module.export = path.dirname(require.main.filename); Nos da el directorio raíz donde estamos trabajando.

require.main.filename returns the entry point of your application.

15. Sirviendo archivos estáticos

Los archivos CSS por convención van a la carpeta "public" ya que queremos indicar que esta carpeta contiene contenido que siempre está expuesto al público.

Para poder linkear un archivo CSS a un HTML necesitamos una de las características que nos trae Express. Necesitamos poder servir archivos de forma estática. Que sean estáticos significa que no van a ser manejados por un enrutador de Express/Middleware o nada de este tipo, sino que se envían directamente al sistema de archivos.

Para linkear el CSS (puede ser cualquier archivo estático, para el ejemplo vamos a suponer que queremos linkear CSS y HTML) de la carpeta public public a todos los <link> que queramos dentro de HTML vamos a utilizar el método .use() y dentro vamos a poner el método express.static("PATH"), static es un metodo para servir archivos estaticos. El PATH va a ser como vimos anteriormente, path.join("__dirname", "public"). Solo nos hace falta acceder a la carpeta public, después cuando hagamos el linkeo en el HTML por defecto va a tomar como punto de partida este PATH que le indicamos.

Ejemplo:

app.use(express.static(path.join(__dirname, "public"))); -> En donde levantamos el servidor y manejamos el ruteo.

<link rel="stylesheet" href="/css/main.css"> -> Al definir arriba el PATH de los archivos estáticos, en este caso de la carpeta public, toma como punto de partida esa carpeta.

Podemos definir más de una carpeta para contenidos estáticos, para el ejemplo utilizamos public ya que contiene los archivos css.

Si tenemos más de una carpeta va a recorrer el contenido de todas hasta encontrar alguna coincidencia.

Los archivos estáticos no están limitados a CSS y JavaScript, también tenemos en cuenta imágenes, videos, gifs, etc.

16. Resumén del módulo

¿Que es Express.js?

- Express.js es un Framework de Node.JS - Un paquete que añade muchisimas funciones y herramientas, y un set claro de reglas de como la app debería ser construida (middlewares).
- Es muy extenso y otros paquetes puede ser incluidos en él.

Middleware, next() y res()

- Express se basa mucho en las funciones Middelware, podemos añadirlas facilmente con .use().
- Las funcinoes Middleware manejan las respuestas y deberían llamar a next() para enviarla respuesta al siguiente Middelware en linea o enviar la respuesta.

Routing

- Podemos filtrar peticiones por el PATH y el metodo.
- Si filtramos por metodo el path tiene que matchear al 100%, sino, el primer segmento de la URL se matchea.
- Podemos utilizar express.Router() para dividir las rutas en archivos dentro de la carpeta routes.

Serve Files

- No estamos limitados a enviar texto de relleno como respuesta.
- Podemos utilizar sendFile para enviar archivos al cliente.
- Si una peticion esta hecha directamente para un archivo donde se requiera CSS o algo estatico, podemos habilitar la funcion de express.static y generar un path.join a la carpeta donde se sirven esos archivos estaticos.