NODEJS

Es un lenguaje de back end. Interpreta js al nivel de la computadora (es un lenguaje de entrada y salida de datos). Para ello utiliza el motor V8 (compila el código en js a lenguaje a máquina con este motor basado en la tecnología de Chrome).

Qué puedo hacer con Node

Puedo manipular archivos del servidor (leer, editar, borrar, crear). Puedo acceder a una base de datos y trabajar con sus funciones básicas (C.R.U.D. : create, read, update, delete) o lo que es lo mismo en español, sistema A.B.M. (alta, baja, modificación). También puedo generar contenido dinámico.

Crear archivos

Append : crea un archivo con contenido. Si ya existe el archivo, le agrega información.

Open: crea un archivo vacío y si el archivo ya existe lo reabre.

writeFile: si el archivo existe, reemplaza su contenido (lo sobreescribe). Si no existe, crea uno nuevo con el contenido que le pasemos.

Obtener módulos

Son el equivalente a las librerías de js. Se bajan como paquetes (o packages) a través del npm (node package manager). Algunos ya vienen preinstalados dentro de Node. Los módulos de terceros que no vienen preinstalados pueden instalarse de manera global (funcionan en todos los proyectos) o local (sólo funcionan en un proyecto puntual, donde instalamos el paquete). Se ejecutan a través de la terminal.

Los llamo dentro de mis proyectos con la función require. La función require tiene los parámetros request (req) y response (res).

Crear un servidor

Llamo al módulo:

var nombremódulo = require(“nombremódulo”)

Ej: var http = require(“http”)

Creo el servidor:

nombrevariable.nombremódulo(function(requerimientodelcliente, respuestadelservidor))

Ej: http.createServer(function(req, res))

Manipular archivos con el módulo FS

Es un módulo con funciones síncronas y asíncronas (que usan callbacks). Permite crear, leer, modificar y borrar archivos y carpetas. El archivo donde guardamos la info se guarda dentro de la carpeta donde están las rutas. El módulo se llama de la siguiente manera:

const fs = require(“FS”)

Este módulo se usa cuando trabajamos con pocos datos guardados en archivos pequeños (como txt o json). Para archivos de datos más grandes se trabaja con bases de datos (SQL, MongoDB, etc).

Leer archivos:

FS.readFile(“nombrearchivo.html”, function(err, data)

O bien FS.readFileSync(“nombrearchivo.txt”, “utf8”) → esto último se agrega sólo si es un archivo de texto.
Esta función me devuelve el contenido del archivo, que puedo guardar en una variable para luego usarlo en mi proyecto.
Abrir archivos:
FS.open(“nombrearchivo.txt”, “w”, function(err, data)
El “w” es un flag o argumento de modo de apertura (para que abra el archivo).
Agregar contenido a los archivos:
Si el archivo ya existe, reemplaza su contenido.
FS.writeFile(“nombrearchivo.txt”, “contenidoaagregar”, function(err, data))
O bien, para guardar objetos en el archivo:
FS.writeFileSync(“archivo.json”,JSON.stringify(nombreobjeto))
Como recibimos un archivo .json, debemos parsearlo con la función stringify y guardarlo en una variable para poder usar esa info que contiene en nuestro proyecto y que el cliente la pueda ver.

Entorno del cliente y del servidor

Cliente:

- Navegador
- Local storage
- Jquery
- Html
- Js de la carpeta public

Servidor:

- Express
- NodeJS

Ajax ayuda a conectar ambos entornos.
La parte lógica de mi web está del lado del cliente (dentro de la carpeta public en diferentes archivos de js). El servidor envía y guarda información a través de express y en los archivos js escritos con node.

EXPRESS

Es un framework para NodeJs que estructura el código de una manera determinada. Se instala desde el npm de manera local dentro la carpeta de nuestro proyecto. Cuando quiero subir mi proyecto a github debo crear un archivo gitignore que dentro tenga escrito el nombre de la carpeta node_modules. Dentro de esta carpeta están guardadas todas las dependencias de mi proyecto (son los módulos/paquetes necesarios para que el proyecto funcione). Si alguien se baja mi proyecto de github, debe ejecutar npm install para bajarse todas las dependencias necesarias en su computadora.

Estructura de carpetas

Los archivos estáticos están dentro de la carpeta public. Son los que ve el cliente. Accedo con la función static:
app.use(express.static(path.join(__dirname, “/carpeta”)))

Carpeta routes: dentro de la misma está el archivo indexRoutes (que devuelve archivos html) y el archivo apiRoutes (que devuelve objetos en formato json).

Módulos importantes

Routes: guarda la información de respuesta para cada ruta a la que le pegue el cliente.

Router: define las rutas de nuestro proyecto.

Query values: el cliente le manda parámetros al servidor directamente en la url. La estructura es la siguiente:
nombrederuta/?nombredekey=nombredelvalue&otrokey=otrovalue
En este ejemplo le mando un nombre:
users/?name=matilde&name=eva
res.send(req.query)
Puedo también mandar parámetros dentro de una ruta. Ejemplo:
user/:name
res.send(req.params)

Body: Es donde se guarda la información que manda el cliente al servidor. Puedo ver la info escribiendo req.body. Ejemplo, recibo un objeto:
let persona = {nombre: req.body.nombre, edad: req.body.edad}

Body parser: parsea y guarda los parámetros enviados por el cliente en formato string a un formato json para que el servidor los pueda utilizar (leer, modificar, borrar).

Path: nos ayuda a armar las rutas para acceder a un archivo y mandarlo al cliente. Se usa de la siguiente manera:
res.sendFile(path.join(__dirname, "..", "", "archivo.html")
La variable __dirname se usa para mostrar la ruta de un archivo. Luego se usa ".." para subir una carpeta. Luego se indica el nombre de la carpeta donde está el archivo (en este caso se lo deja en blanco porque se encuentra dentro del directorio raíz del proyecto). Por último se indica el nombre del archivo.

Reutilizar módulos

module.exports = nombredelmódulo;
Puedo reutilizar el mismo código en otros archivos de mi proyecto.

Tipos de archivos dentro de mi proyecto

Package.json: guarda todas las dependencias de nuestro proyecto y muestra información general sobre el mismo.

Archivos .env:

Trabajan con variables de entorno. Guardan información sensible, como el puerto donde correrá mi proyecto, contraseñas, información de la conexión a la base de datos, etc.

Archivo start:

En este lugar seteo el puerto en el que correrá el servidor y lo inicializo: app.set("port"), app.listen. Uso también dotenv para enlazar al archivo .env.

Archivo app:

Configuro lo que hará express. Si necesito una ruta, la seteo. Creo la app. Uso el body parser para convertir a json la data que manda el cliente en el request.

Archivo index:

Estará dentro de la carpeta de rutas (routes). Importo express y su router, que es lo que me ayudará a definir las rutas. Defino las rutas de la siguiente manera: router.get("/nombrederuta", (req, res)), res. send("lo que le quiero mandar al usuario"). Con module.exports(router) exporto mis rutas declaradas en el archivo index hacia otros archivos de mi proyecto. Con los require llamo a los módulos y a las rutas.

Redireccionar

location.href="/ruta/?id=${id}" → parámetro, en este caso un id.

Recuperar parámetros de un html

URLSearchParams(window.location.search).get("nombreparametrobuscado"). Todo lo que está después del signo ? En la url es un parámetro.

Función .done

Se ejecuta sólo si la respuesta del servidor llegó bien al cliente.

Función .fail

Se ejecuta cuando algo salió mal y no llega al cliente la respuesta del servidor.

Pasos para iniciar un proyecto (versión simple)

npm init: lo escribimos en consola, abierta dentro de la carpeta de nuestro proyecto. Aquí se configura la información guardada dentro del archivo package.json (nombre del archivo y del autor, descripción de nuestro proyecto, lista de scripts, lista de dependencias usadas -entre ellas, express-, etc).

Instalar express: escribimos en consola (también abierta dentro de la carpeta de nuestro proyecto) npm i -S express.

Crear el archivo app.js: dentro de este archivo llamamos al módulo express (require: express). Creamos la app express. Creamos las rutas con el router (express.Router), luego router.get("/nombredelaruta", (req, res)) y por último app.use(router).

Seguimiento del puerto: app.listen(númerodepuerto). El número de puerto debería estar guardado dentro de un archivo .env ya que es información sensible. Este es el puerto desde donde se correrá nuestra app.

Correr la app: La forma de iniciar la app se setea en el archivo package.json dentro de la parte de scripts.

Ej: "start" : "npm run start.js"

Pongo en consola npm start y arranca mi app. Se usa cuando la app ya funciona sin bugs.

Otra forma es correrla con nodemon, de manera de no tener que reiniciar el servidor cada vez que hacemos un cambio en el proyecto (nodemon lo hace automáticamente y la reinicia después de cada cambio), escribiendo en consola: nodemon app.js. Se usa cuando estoy probando mi app. También se puede setear dentro del script "watch". En este caso lo ejecutamos escribiendo en consola: npm run watch.

Express generator
Generador de aplicaciones express, para crear rápidamente un esqueleto de la app (scafolding). Se instala de manera global: npm install express-generator -g
Luego crea la estructura de carpetas y archivos:
express —no-view nombre-carpeta-proyecto
Crea una carpeta llamada bin (de binario), que es una carpeta que contiene archivos para ejecutar un programa. Esta carpeta contiene un archivo llamado www (sin ninguna extensión) que hace lo mismo que el start.js: setea los puertos para correr el servidor y muestra mensajes del servidor en la consola (listening…).
Para iniciar nuestra app, modificamos los scripts del package.json: en el "start" ponemos "nodemon ./bin/www". O sino podemos crear un nuevo script que se llame "watch" y ejecute lo anterior. Y lo llamamos con npm run watch.

APIs
La interfaz de programación de aplicaciones, conocida también por la sigla **API** del inglés *application programming interface*, es un conjunto de subrutinas, funciones y procedimientos (o métodos) que ofrece cierta biblioteca para ser utilizado por otro software. Uno de los principales propósitos de una API consiste en proporcionar un conjunto de funciones de uso general, por ejemplo, la API de Google Books para ver información de libros. De esta forma, los programadores se benefician de las ventajas de la API haciendo uso de su funcionalidad, evitándose el trabajo de programar todo desde el principio.

AXIOS

Es una librería (como Jquery) que nos permite conectarnos a una API y pedirle datos. Se instala de manera local en cada proyecto. Se usa tanto del lado del cliente como del servidor, a diferencia de Ajax que sólo trabaja del lado del cliente.
Instalación
npm install axios
Lo incluyo en mi proyecto con: const axios = require("axios")
Crear rutas
router.get("/ruta", function(req, res, next)
axios.get("url").then(function(result)
Métodos
Get: axios.get("/api/users").then(function(result)
Luego de axios va el método que queremos usar, en este caso el get. Luego la ruta. La función then es un callback (equivalente a la función .done). El parámetro que guarda todo el conjunto de información recibida es el result. Axios manda esa info dentro de un objeto con diferentes parámetros. Los más importantes son data (que guarda la info requerida) y status (que indica el estado del servidor). Accedo escribiendo result.data o result.status.
Post: axios.post("ruta", dataquequieroenviar).then(function())

La data se manda dentro de un objeto o array de objetos en formato json. El cliente envía esa data al servidor directamente en el parámetro luego de la url.
Errores: Luego de la función con el método utilizado se llama a la función catch en el caso de que haya errores. Es el equivalente a la función .fail.

```
.catch(function(err){
console.log("algo salió mal", err)})
```