Node.js : Express.js i npm

Express és un framework web minimalista per a Node.js que facilita la creació d'aplicacions i servidors web. Proporciona eines i funcionalitats per gestionar rutes, sol·licituds HTTP i respostes de manera eficient, permetent desenvolupar aplicacions web amb menys codi i major organització. Express és molt flexible, permet afegir extensions i middleware per afegir funcionalitats com autenticació, tractament d'errors, o manipulació de dades enviades pels usuaris.
És molt popular en el desenvolupament de backend gràcies a la seva simplicitat i la seva integració amb altres tecnologies.

npm (Node Package Manager) és el gestor de paquets oficial de Node.js. S'utilitza per instal·lar, gestionar i compartir llibreries o paquets de codi JavaScript, tant per aplicacions locals com per projectes a escala global.

Funcions principals de npm:

- Instal·lar paquets: Pots afegir llibreries al teu projecte, Express.js per exemple.
- Gestionar dependències: Automatitza la gestió de versions i dependències de llibreries.
- Publicar paquets: Permet que els desenvolupadors comparteixin el seu codi amb la comunitat.

Inicialitzar el projecte amb npm

Per començar, inicialitza el projecte amb npm, que crearà un fitxer **package.json** que conté informació sobre el projecte, és on es gestionaran les dependències.

```
npm init -y
```

El fitxer package.json és fonamental en qualsevol projecte de Node.js, ja que conté metadades importants sobre l'aplicació i les seves dependències.
Exemple del contingut inicial de fitxer package.json:

```
{
	"name": "my-project",
	"version": "1.0.0",
	"main": "server.js",
	"scripts": {
		"test": "echo \"Error: no test specified\" && exit 1",
		"start": "node server.js"
	},
	"keywords": [],
	"author": "",
	"license": "ISC",
	"description": ""
}
```

scripts: És un objecte que conté scripts personalitzats que es poden executar amb:
`npm run <script-name>`
Per executar l'script definit a start, que per convenció s'utilitza per iniciar l'aplicació, podem executar directament:
`npm start`

Instal·lar Express.js

Per instal·lar un paquet o mòdul específic de npm en el teu projecte pots utilitzar la comanda:
`npm install <module-name>`

Per tant, per instal·lar Express.js escriurem:
`npm install express`

npm descarregarà i afegirà el paquet i les seves dependències a la carpeta **node_modules**, crearà la carpeta si és el primer paquet que s'instal·la al projecte; també actualitzarà automàticament el fitxer package.json, afegint el paquet a la secció de dependències, i generarà el **package-lock.json** si no existia, aquest fitxer conté una representació exacta de les dependències instal·lades en un projecte, inclou informació detallada sobre les versions exactes de cada paquet, així com les dependències de les dependències

Així és com afegeix Express.js com a dependència del nostre projecte a package.json:

```
"dependencies": {
      "express": "^4.21.0"
}
```

El símbol «^» indica que es permetran actualitzacions automàtiques de les dependències dins de la mateixa versió principal, a dir que l'última versió a la que s'actualitzarà serà l'última abans de passar a la versió 5.

Les dependències que s'instal·lin especificant que només seran usades per desenvolupament, és a dir, que no s'instal·laran a producció, s'afegeixen a «devDependencies».
Per exemple, «nodemon», una eina que reinicia automàticament l'aplicació en detectar canvis, s'instal·laria amb:
`npm install nodemon –save-dev`

També podem desinstal·lar paquets del nostre projecte, eliminant-lo de node_modules i del package.json i package-lock.json. Així desinstal·lem Express.js:
`npm uninstall express`

No guardarem node_modules al nostre repositori, ja que podem generar la carpeta node_modules i el seu contingut a partir de package.json, on s'indica totes les dependències que el projecte requereix. Per això usem la comanda:
`npm install`
Utilitzarà package-lock.json per instal·lar la versió concreta de cada paquet.

ES Module

És un sistema d'importació i exportació de mòduls en JavaScript que permet organitzar el codi en fitxers separats. A diferència del sistema de mòduls CommonJS (que utilitza require i module.exports), els ES Modules utilitzen la sintaxi import i export per gestionar dependències i encapsular funcionalitats.

Per indicar que l'usarem en lloc del sistema de mòduls CommonJS, hem d'afegir la següent línia a package.json:

```
"type": "module",
```

Exemple d'importació de 4 funcions del fitxer mates.js:

```
import { suma, resta, multiplicacio, divisio } from './mates.js';
```

Les funcions importades han d'estar exportades al fitxer d'origen mates.js, davant de les funcions hi escrivim export:

```
export function suma(a, b) {
    return a + b;
}
```

D'aquesta manera les funcions definides a un fitxer es poden utilitzar al fitxer on s'importen. Les funcions es criden usant el nom de la importació, exemple d'ús:

```
suma(5, 3);
```

Arquitectura Client-Servidor Web

Model d'arquitectura on les aplicacions es divideixen en dues parts:

- Client: Realitza sol·licituds **HTTP** mitjançant un navegador.
- Servidor: Processa les sol·licituds arribades a un port específic i els retorna resposta.

HTTP (Hypertext Transfer Protocol) és el protocol de comunicació que s'utilitza per la transferència de dades a la web. HTTPS és la seva versió segura, que utilitza TLS/SSL per xifrar la comunicació.

Mètodes HTTP:

- **GET**: Recupera dades del servidor.
- **POST**: Envia dades al servidor per crear o actualitzar recursos.
- **PUT**: Actualitza un recurs existent al servidor.
- **DELETE**: Elimina un recurs del servidor.
- **PATCH**: Aplica canvis parcials a un recurs.

A més de la sol·licitud i la resposta, el client i el servidor proporcionen informació addicional mitjançant les capçaleres (**Headers**), les quals ajuden a l'altra part a entendre millor la sol·licitud i personalitzar la resposta.
Alguns camps de les Capçaleres: Tipus de contingut (Content-Type), l'autenticació (Authorization), i la informació del navegador (User-Agent).

Un cop el servidor processa la sol·licitud, envia una resposta amb un codi d'estat HTTP que indica si la sol·licitud ha estat satisfactòria o no. Els més comuns són:

- **200 OK**: Sol·licitud exitosa.
- **400 Bad Request**: Error del client. El servidor no pot interpretar la sol·licitud.
- **404 Not Found**: Error del client. Recursos no trobats.
- **500 Internal Server Error**: Error del servidor.

Codi d'estat de respostes HTTP:
https://developer.mozilla.org/es/docs/Web/HTTP/Status

Configuració d’un servidor

Creem un servidor web que escolta peticions HTTP al port 3000 i respon amb un missatge quan es realitza una petició de tipus GET a la ruta ‘/’.

```
import express from 'express';
const app = express(); // Crea una instància de l'aplicació Express
const PORT = 3000; // Defineix el port del servidor

// Ruta principal
app.get('/', (req, res) => {
  res.send('<h1>Hello, World!</h1>'); // Resposta a la petició
});

// Inicia el servidor
app.listen(PORT, () => {
  console.log(`Servidor en funcionament a http://localhost:${PORT}`);
});
```

Executa el servidor. Obre el navegador i accedeix a http://localhost:3000/. Veuràs el missatge "Hello, World!".

app.get(route, callback)
Defineix una ruta per gestionar peticions HTTP, en aquest cas, de tipus GET.
route: És la URL on s’espera la petició.
callback: funció que es crida al rebre una petició:

- req (request): conté la informació sobre la petició HTTP. Podem obtenir les dades passades a través de la URL consultant l’objecte: «req.query».
- res (response): és l'objecte que es fa servir per enviar la resposta al client.

res.send()
Envia una resposta HTTP al client. Aquest mètode finalitza el cicle de petició-resposta, de manera que no es poden enviar més dades després d'usar-lo.

app.listen(port, [host], [backlog], [callback])
Permet escoltar peticions HTTP a un port específic.
port: És el número de port on el servidor escoltarà les peticions.
host: És l'adreça IP o el domini on vols que l'aplicació escolti. Si no es proporciona, per defecte serà localhost (‘127.0.0.1’). Si volem que escolti totes les interfícies de xarxa indicarem ‘0.0.0.0’.
callback: Una funció que s'executa quan el servidor comença a escoltar correctament. Normalment es fa servir per mostrar un missatge a la consola confirmant que el servidor s'ha iniciat.

Middleware

Són funcions que tenen accés a l'objecte de sol·licitud (req), a l'objecte de resposta (res) i a la següent funció de middleware en el cicle de sol·licitud-resposta de l'aplicació (next). Permeten executar codi per processar les sol·licituds abans d'arribar a les rutes definides. S'utilitzen per modificar els objectes req i res per a tasques com ara analitzar els cossos de les sol·licituds, afegir encapçalaments de resposta, etc.
Els middleware s'han d'ubicar abans de definir les rutes a les que volem que s'apliquin.

Alguns exemples de middleware personalitzats:

Middleware per mostrar per consola informació de les peticions.

```
app.use('/routes/*', (req, res, next) => {
  const log = {
    time: new Date().toISOString(),
    method: req.method,
    url: req.url,
    headers: req.headers,
    ip: req.ip,
  };

  console.log("Nova petició rebuda: ", log);
  next(); // Passa al següent middleware o ruta
});
```

En aquest cas, només s'aplica a les rutes que comencin per «/routes».

Middleware per mostrar per consola les respostes a les peticions. Utilitza un event listener que s'executa quan la resposta és enviada.

```
app.use((req, res, next) => {
  res.on('finish', () => {
    console.log(`Resposta enviada:
      Estat: ${res.statusCode},
      Missatge: ${res.statusMessage},
      Capçaleres: ${JSON.stringify(res.getHeaders())}`);
  });
  next();
});
```

També hi ha middleware de tercers, per exemple, els següents venen inclosos amb Express:

Permet interpretar les dades del cos de la sol·licitud enviades en format JSON. Les dades JSON es converteixen en l'objecte «req.body»:

```
app.use(express.json());
```

Permet analitzar les dades d'un formulari no-GET (URL-encoded). Permet que Express entengui les dades rebudes des d'un formulari. Les dades es guarden a l'objecte «req.body»:

```
app.use(express.urlencoded({ extended: true }));
```

Ambdues funcions formen part de body-parser, un middleware per a Node.js que permet analitzar el cos de les sol·licituds HTTP i convertir-lo en un objecte JavaScript que es pot utilitzar a l'aplicació. Des d'Express 4.16.0, la funcionalitat de body-parser s'ha integrat directament dins d'Express.

Middleware que permet servir fitxers estàtics (imatges, CSS, JavaScript, HTML, etc.) des d'un directori especificat.

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

En aquest cas, servirà tots els fitxers del directori «public». Intentarà obrir automàticament el fitxer de nom «index.html» quan accedim a l'arrel del servidor, si vols entregar un altre fitxer o en una altra ruta pots usar **res.sendFile()**.

Altres middleware, integrats o no, dins d'Express:
https://expressjs.com/en/resources/middleware.html

app.use([path], callback)
Mètode que s'utilitza per registrar un middleware a l'aplicació
path: Una cadena que representa la ruta en què s'aplicarà el middleware. Si no es proporciona, el middleware s'aplicarà a totes les rutes.
callback: Funció o array de funcions middleware que processaran les sol·licituds HTTP.

Gestió d'errors a través de middleware

És essencial per garantir que els usuaris rebin respostes adequades davant de situacions inesperades. Amb Express.js podem especificar el codi d'estat de la resposta HTTP en el nostre servidor.

Gestió de rutes no definides (404). Si una sol·licitud arriba aquí, és perquè cap de les rutes definides anteriorment ha coincidit amb la sol·licitud, i aquest middleware actua com a capturador de "tot" al final, és a dir, per gestionar les rutes inexistents. Retorna un estat HTTP 404 i un missatge:

```
app.use((req, res, next) => {
  res.status(404).send('Ruta no trobada!');
});
```

Per exemple si accedim a la ruta no definida: http://localhost:3000/holabondia

Gestió d'errors generals (500). S'encarrega de gestionar errors que poden ocórrer durant el processament de les sol·licituds, errors generats pel codi. Registra l'error a la consola i retorna un estat HTTP 500 i un missatge:

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Hi ha hagut un error!');
});
```

app.use((err, req, res, next) => { ... })
Afegeix a la funció de callback el paràmetre «err» que conté l'error generat pel codi.

res.status().send()
Estableix el codi d'status HTTP, si no s'estableix s'envia 200 OK. També envia una resposta HTTP al client.

Ordre en que es defineixen les instruccions

És important pel correcte funcionament de l'aplicació. Express processa les sol·licituds de manera seqüencial, d'acord amb l'ordre en què es registren les rutes i els middleware. Si les instruccions no estan en l'ordre adequat, això pot causar problemes o comportaments no desitjats.

Ordre general en què s'ha d'estructurar el codi:

1. **Imports**: Importa totes les dependències necessàries, com Express i qualsevol middleware que puguis necessitar.
2. **Configuració de l'aplicació**: Crea una instància d'Express i defineix constants com el port.
3. **Middlewares**: S'han d'afegir abans de definir i processar les rutes que esperen les dades.
    - Servei de fitxers estàtics.
    - Parseig de dades.
    - Middlewares personalitzats.
4. **Definició de rutes**: Defineix les rutes de la teva aplicació utilitzant app.get(), app.post(), etc.
5. **Gestió d'errors i rutes no trobades**: Al final del teu codi, afegeix un middleware per gestionar errors i una ruta per a gestionar sol·licituds a rutes no definides (404). Això és important perquè aquestes funcionalitats s'han d'executar si cap de les rutes definides prèviament no coincideix.

ACTIVITATS

Crearem aplicacions **server-side**: Aquest tipus d'aplicacions s'executen en un servidor i gestionen les peticions dels clients, generalment des d'un navegador web. Aquestes aplicacions són responsables de processar la lògica de negoci, accedir a les dades i generar contingut que es retorna al client en forma d'HTML, JSON o altres formats.

Valida les dades i gestiona els possibles errors adequadament.

1. **Formulari per filtrar videojocs**

   L'aplicació ha d'entregar un formulari estàtic al client en rebre peticions a l'arrel, a través del port 3000.

   L'usuari podrà omplir el formulari per filtrar pel tipus de videojocs que vol que se li mostrin. Obtenim la petició al servidor i retornem una resposta en format JSON. Genera un array amb els objectes videojoc que compleixin els requisits de l'usuari. Per retornar un JSON, en lloc d'usar res.send() és més recomanable usar **res.json()**.

   Utilitza, sense modificar, el formulari HTML i el fitxer JSON, que actua com a base de dades dels videojocs disponibles, que trobaràs al següent enllaç: https://github.com/xbaubes/DesenvolupamentWeb/tree/main/Backend/Node.js/Express.js/videogames

   El formulari conté la obligatorietat d'indicar la dècada. Valida també, des del servidor, que la dècada s'ha enviat, si no es rep correctament retorna un error adequat.

   Procura homogeneïtzar les dades perquè encaixi la representació del formulari i del JSON.

2. Aplicació amb formulari estàtic per inserció i consulta dinàmica de dades

Crea una aplicació que segueixi el següent esquema de navegació, rutes i interaccions:

Cal guardar les dades que s'introdueixen des del formulari a un fitxer en format JSON.
Valora si és necessari encriptar les dades guardades en fitxers.

Per aquest exemple guardaríem un array d'objectes. Sintaxi de cada objecte:

```
{
  "nom": "Alzina",
  "fullaPerenne": true,
  "fullaAgulla": false,
  "alcadaHabitual": 20,
  "clima": "Mediterrani",
  "imatge": "https://upload.wikimedia.org/wikipedia/commons/5/55/Mendaza%2C_Navarra_Spanien-Steineiche.jpg"
}
```

Pots realitzar la pràctica sobre qualsevol tema **excepte d'arbres**.
Has de guardar almenys 5 camps clau/valor, incloent una imatge, un booleà i un nombre.

El formulari per introduïr dades és un fitxer estàtic. Els fitxers estàtics es guarden a la carpeta «public».

És possible que necessitis redirigir el client a una altra URL, usant **res.redirect()** redirigeixes a través d'una sol·licitud GET.

Les pàgines HTML que continguin informació obtinguda al servidor han de ser generades completament des del servidor i enviades al client fent servir: **res.send()**.
Per exemple:

```
res.send(`
      <!DOCTYPE html>
      <html lang="ca">
      <head>
          <meta charset="UTF-8">
          <meta name="viewport" content="width=device-width, initial-scale=1.0">
      </head>
      <body>
            ${userData}
      </body>
      </html>
`);
```

OPCIONAL: En lloc de llegir imatges d'internet, guardar-les al servidor i mostrar-les.

3. Modularització en fitxers

Separa el codi de l'activitat anterior en diferents fitxers per tal de mantenir l'aplicació més organitzada, procura que segueixi el següent esquema:

```
project
│
├── /routes // Rutes
│   └── routes.js
│
├── /controllers // Lògica
│   └── routesController.js
│
├── /public // Fitxers estàtics entregats directament pel servidor
│   ├── index.html
│   ├── stylesCreate.css
│   ├── stylesList.css
│   └── stylesDetail.css
│
├── server.js // Fitxer principal de l'aplicació
├── package.json
└── node_modules/
```

Utilitza IA per generar 3 fitxers CSS per donar estil a les 3 pàgines creades.

Modifica les URL per adaptar-les a la nova configuració, per exemple, hauràs de modificar l'atribut «action» del formulari.

Exemple d'aplicació modularitzada separant el codi per agrupar-lo en diferents fitxers. Aquesta aplicació respon únicament a les següents peticions GET:
http://localhost:3000/rt/
http://localhost:3000/rt/about

server.js: Importem les rutes del fitxer «routes.js» i definim que s'han de cridar afegint «/rt» a la URL. S'aplica un middleware que registra totes les peticions a aquestes rutes.

```
import express from 'express';
import rutes from './routes/routes.js'; // Importa "routes.js"
const app = express();
const PORT = 3000;

//Aquest middleware s'aplicarà a totes les rutes incloses dins de "/rt/"
app.use('/rt/', (req, res, next) => {
  console.log("Nova petició rebuda");
  next();
});

// A partir de la URL "/rt" usem les rutes definides a "routes.js"
app.use('/rt', rutes);

app.listen(PORT, () => {
  console.log(`Servidor en funcionament a http://localhost:${PORT}`);
});
```

routes.js: Importem l'objecte Router d'Express per definir rutes específiques; a més, importem els controladors «home» i «about» per gestionar les peticions a les rutes, i exportem el router per a la seva utilització al fitxer main: «server.js».

```
// Només importem una part específica del mòdul Express: l'objecte Router
import { Router } from 'express';
// Importa controlador
import { home, about } from '../controllers/routesController.js';

const router = Router();

// Ruta bàsica
router.get('/', home); // Utilitza el controlador home

// Ruta addicional
router.get('/about', about); // Utilitza el controlador about

export default router;
```

routesController.js: Els controladors són funcions que encapsulen la lògica per gestionar una petició específica a una ruta.

```
// Controlador per a la ruta bàsica
export const home = (req, res) => {
  res.send('<h1>Hello, World!</h1>');
};

// Controlador per a la ruta addicional
export const about = (req, res) => {
  res.send('<h2>Aquesta és la pàgina d\'informació!</h2>');
};
```

Aquesta és l'estructura de fitxers d'aquesta aplicació modularitzada d'exemple:

```
project
│
├── /routes
│   └── routes.js
│
├── /controllers
│   └── routesController.js
│
└── server.js
```

BIBLIOGRAFIA I WEBGRAFIA

«AlvaroPerdiz». https://alvaroperdiz.com/javascript/node-js
«MDN Web Docs». https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs
«Tutorials Point». https://www.tutorialspoint.com/expressjs/index.htm
«w3Schools». https://www.w3schools.in/express-js/introduction