Express.js : API REST

Què és una API REST? Clicka!

Una API REST ha de seguir els principis del model **CRUD**.

CRUD és un acrònim que fa referència a les quatre operacions bàsiques que es poden realitzar sobre les dades en una aplicació: **Create** (C), **Read** (R), **Update** (U) i **Delete** (D).

Create (**HTTP POST**): Aquesta operació s'utilitza per afegir nous registres a la base de dades.
Exemple: Afegir un nou usuari a la base de dades.

Read (**HTTP GET**): Aquesta operació permet recuperar dades existents, un o més registres.
Exemple: Obtenir la llista de tots els usuaris.

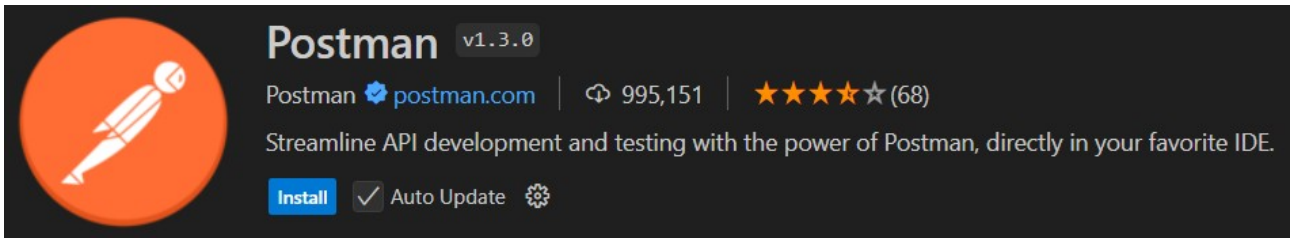
Update (**HTTP PUT** o **HTTP PATCH**): Aquesta operació modifica registres existents.
Exemple: Actualitzar la informació d'un usuari específic.

Delete (**HTTP DELETE**): Aquesta operació s'encarrega d'eliminar registres de la base de dades.
Exemple: Eliminar un usuari de la base de dades.

Per provar i verificar que una API REST segueix correctament els principis CRUD es pot utilitzar **Postman**, una eina molt popular que facilita l'enviament de sol·licituds HTTP i la visualització de les respostes.

Postman permet simular les operacions CRUD enviant sol·licituds HTTP com POST, GET, PUT, PATCH i DELETE directament des de la seva interfície. A més, tens la possibilitat de configurar paràmetres i capçaleres, afegint capçaleres HTTP, paràmetres de sol·licitud i variables per emular situacions reals, cosa que facilita l'enviament d'informació al servidor de manera precisa. L'eina permet també visualitzar les respostes de l'API, de manera que pots analitzar les dades que retorna el servidor per comprovar que els resultats són correctes per a cada operació.

El podem instal·lar com una extensió de **Visual Studio Code**:



Per realitzar les peticions clicka **«New HTTP Request»** a l'extensió de Postman.
Revisa els resultats usant la **Postman Console**.

Regular l’accés a l’API amb CORS

CORS (Cross-Origin Resource Sharing) és un mecanisme de seguretat que permet que les pàgines web demanin informació d'altres pàgines que són de dominis diferents. Per defecte, això no és possible.
Aquest sistema controla i permet l'accés a recursos compartits entre diferents orígens (dominis), ajudant a prevenir atacs com el cross-site scripting (XSS).
CORS funciona mitjançant l'ús d'encapçalaments HTTP que indiquen si un recurs pot ser compartit amb un origen diferent.

Amb el següent codi, aconsegueixes habilitar CORS a la teva API Express amb la configuració per defecte:

```
import cors from 'cors';

// Habilita CORS amb la configuració per defecte (permet tots els orígens)
app.use(cors());
```

Permets l'accés a la teva API des de qualsevol origen, facilitant les proves i el desenvolupament. És a dir, podràs provar el servidor usant una pàgina web ubicada en un ordinador de la mateixa xarxa.

Però, al no fer cap limitació d’accés, exposes la teva API a possibles riscos de seguretat. Per tant, per a entorns de producció, és recomanable restringir l'accés especificant quins dominis poden fer sol·licituds a la teva API.

Autenticació d’usuaris amb JWT

JWT (JSON Web Token) s’utilitza en aplicacions web per gestionar l’autenticació d'usuaris mitjançant tokens que emmagatzemen informació de manera segura i es poden utilitzar per identificar i autenticar usuaris.
JWT és un tipus de **Bearer Token**.

Exemple de l’us de JWT:
https://github.com/xbaubes/DesenvolupamentWeb/tree/main/Backend/Node.js/Express.js/JWT
Per major seguretat, farem servir variables d’entorn per guardar la key utilitzant el paquet **dotenv**.
El fitxer és en format **.mjs** perquè s’hi usa la importació ES6 sense configurar el package.json.

Funcionament del codi d’exemple:

1. Inici de Sessió
   Per autenticar com a usuari de l’API necessites obtenir un token, has de fer una petició POST a l'endpoint **/login** amb la següent informació en format JSON al body de la petició:

   ```
   {
     "username": "usuari",
     "password": "contrasenya"
   }
   ```

   Si el login és correcte, rebràs un token en el body de la resposta.
   Si les credencials són incorrectes, rebràs una resposta d'error amb un codi d'estat 401.

2. Accés a Rutes Protegides
   Guarda el token, amb ell pots fer peticions a rutes protegides.
   Fes una petició GET a **/perfil**, incloent el token a la capçalera **Authorization**.

   Si el token és vàlid, rebràs un missatge de benvinguda amb el nom d'usuari al response body.
   Si el token no és vàlid o no s'ha proporcionat, rebràs un codi d'estat 403 o 401 respectivament.

Validació i depuració de dades amb express-validator

Exemple senzill per la validació de les dades d'usuari:

```
// npm install express express-validator
import express from 'express';
import { body, validationResult } from 'express-validator';
const app = express();
const port = 3000;

app.use(express.json());

// Ruta per crear un usuari
app.post('/users', [
  // Validació dels camps
  body('nom').notEmpty().withMessage('El nom és obligatori.'),
  body('email').isEmail().withMessage('El format de l\'email és incorrecte.'),
  body('password').isLength({ min: 6 }).withMessage('Contrasenya mínim 6 caràcters.')
], (req, res) => {
  // Comprovar si hi ha errors de validació
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }

  // Si tot és correcte, processar la sol·licitud
  const { nom, email, password } = req.body;
  res.status(201).json({ message: 'Usuari creat', nom, email });
});

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

body i **validationResult** són funcions de express-validator per definir regles de validació i gestionar errors.

Si intentem crear un usuari amb les següents dades:

```
{
	"nom": "Joan",
	"email": "joan@example.com",
	"password": "secreta123"
}
```

Passarem la validació. Response Body:

```
{
	"message":"Usuari creat","nom":"Joan","email":"joan@example.com"
}
```

Si intentem crear un usuari amb les següents dades:

```
{
	"email": "joan@example.com",
	"password": "secreta123"
}
```

No passarem la validació. Response Body:

```
{
	"errors":[{"type":"field","msg":"El nom és obligatori.","path":"nom","location":"body"}]
}
```

Rutes dinàmiques

Les rutes dinàmiques en Express permeten crear URLs que inclouen paràmetres variables. Això facilita la gestió de recursos amb identitats variables, com per exemple, obtenir un element per ID. Els camps inclosos en rutes dinàmiques són obligatoris al realitzar la crida i no es poden desordenar.

```
app.get('/dinamic/:id', function(req, res){
  res.send('The id you specified is ' + req.params.id);
});
```

Una crida al següent endpoint: http://localhost:3000/dinamic/12345
Resultaria en que **req.params**.id tindria el valor: 12345.
«:id» és una variable que pot prendre diferents valors segons la crida, cosa que permet que Express capturi aquest valor i l'utilitzi dins de la ruta.
Aquesta flexibilitat permet gestionar una gran varietat de sol·licituds amb una sola ruta, ja que el paràmetre dinàmic «:id» pot representar diferents identificadors de recursos segons la sol·licitud.

Si posem 2 o 3 rutes dinàmiques seguides usarem una estructura que inclogui camps informatius:

Declaració	Ús
/productes/:productes/casa/:casa	/productes/televisors/casa/panasonic

No posarem més de 3 rutes dinàmiques seguides, si necessitem més camps llavors ho complementarem usant query parameters, a diferència de les rutes dinàmiques no són obligatòries i es poden desordenar.

Les rutes estàtiques s'han de definir abans de les dinàmiques.
Si definíssim erròniament la següent seqüència de rutes:

```
'/:id'
'/home'
```

una crida «/home» a l'API sempre seria processat per la primera ruta, ja que interpretaria id=«home».

Aquestes són bones pràctiques en una API RESTful.

ACTIVITATS

1. **API per la gestió de llibres**

   L'API ha de permetre gestionar botigues de llibres, llibres i les seves vendes.
   El disseny inclou 3 models de dades relacionats entre ells, guardats en els següents fitxers JSON:
   https://github.com/xbaubes/DesenvolupamentWeb/tree/main/Backend/Node.js/Express.js/books

   Tasques:
   - L'accés a les dades requereix autenticació. S'ha d'oferir un endpoint per la creació d'usuaris que retorni un **JWT** per poder accedir a les dades. Els usuaris registrats i les seves contrasenyes es guardaran un fitxer encriptat: users.json.
   - Valora quines operacions **CRUD** tenen sentit per cada model i dissenya-les, assegura't d'entendre les relacions entre els models abans de començar. L'inventari de la llibreria s'ha de mantenir actualitzat. sales.json és un registre de les vendes realitzades, no és necessari ni actualitzar ni eliminar dades, valora com gestionar les devolucions.
   - Només retornarem respostes en format **JSON**, usant res.json().
   - Retornar les dades **ordenades**. Pots usar el mètode sort() de JS.
   - Cal **validar** les dades rebudes, cal assegurar-se que les dades compleixen el format, tipus i rang esperat abans de ser guardades.
   - Cal **depurar** les dades rebudes, per exemple, eliminant espais sobrants.
   - **Gestió dels errors** que es puguin produir.
   - Retornar **codis d'estat HTTP** adequats per les peticions rebudes.
   - Implementació de **CORS** per permetre accés des d'altres dominis a l'API.
   - Genera **logs** de les interaccions amb l'API: Registre totes les peticions a l'API i els resultats.
   - Inclou **start** a l'objecte scripts del package.json. Si està correctament definit, per exemple, si el fitxer principal és server.js seria: "start": "node server.js".
     Permet arrancar el servidor sense saber res del projecte usant la comanda: npm start.
   - Els identificadors de les dades es generen amb **uuid**, un identificador de 128 bits que per probabilística gairebé assegura que no es repetiran.
   - Actualitza **updateAt** inclòs en cada fitxer en cada modificació de les dades que guarden. Pots utilitzar l'objecte Date de JS.
   - L'api començarà amb la següent ruta: **«api/v1»**. Usar un indicador de versió com v1 permet gestionar canvis i incompatibilitats a mesura que l'API evoluciona. Podríem mantenir la versió antiga perquè no doni problemes a les aplicacions que la usen, mentre desenvolupem noves funcionalitats a v2.
   - Ha de permetre **filtrar** les dades segons camps determinats. Implementa filtratge usant els paràmetres enviats per URL amb GET. Per exemple:
     - Obtenir els llibres escrits per un determinat autor.
   - Desenvolupa **operacions més complexes o que relacionin varis fitxers**.
     Per exemple:
     - Obtenir el número d'exemplars venuts d'un llibre concret.
     - Obtenir les llibreries que tinguin més de 20 exemplars d'un llibre concret.
     - Obtenir la facturació d'una llibreria concreta.

Organització del projecte:

```
api
│
├── routes/
├── controllers/
├── models/
│
├── middleware/
│
├── server.js // Fitxer principal de l'aplicació
├── config.js // Configuració de l'API; per exemple, el port del server
│
├── .env // Variables d'entorn
├── .gitignore // Fitxers i carpetes a ignorar per Git; per exemple, .env
│
├── server.log // Logs de l'API
│
├── package.json
├── package-lock.json
└── node_modules/
```

Crea un fitxer de rutes i un controlador per cada fitxer de dades JSON, això farà l'aplicació més fàcil de mantenir i escalable.

Pots modularitzar més l'aplicació, per exemple, separant l'accés a dades.

BIBLIOGRAFIA I WEBGRAFIA

«Brave Developer». https://bravedeveloper.com/
«DiegoSF». https://diegosf.es/big-data-y-analisis-de-datos/que-es-crud/
«ExpressJS». https://expressjs.com/es/api.html
«Microsoft Learn». https://learn.microsoft.com/es-es/training/
«Postman». https://blog.postman.com/how-to-create-a-rest-api-with-node-js-and-express/