Persistència amb fitxers amb nodejs

IES Jaume II El Just
Tavernes de la Valldigna

Aplicacions multiplataforma amb tecnologies web

Curs 20-21

Continguts

1. El mòdul `fs` . . . 3
1.1. Què són els mòduls? Mòduls integrats . . . 3
1.2. El mòdul integrat `fs` . . . 3
1.3. Llistat de directoris: `fs.readdir()` fs.`readdirSync()`: . . . 3
1.4. Esborrat, renomenat i còpia de fitxers . . . 6
2. Lectura de fitxers . . . 8
2.1. Lectura línia a línia i paraula a paraula . . . 9
3. Escriptura de fitxers . . . 10
3.1. fs.appendFile . . . 10
3.2. fs.appendFileSync . . . 11
3.4. writeFile i writeFileSync . . . 12
4. Fitxers JSON . . . 12
5. Fitxers XML . . . 14

1. El mòdul `fs`

1.1. Què són els mòduls? Mòduls integrats

Com ja sabem, **nodejs** és un runtime que ens permet fer ús de javascript per desenvolupar qualsevol tipus d'aplicació multiplataforma, i que ha estat cresquent molt als últims temps.

A la introducció ja hem vist algunes generalitats sobre l'entorn, la instal·lació, i una xicoteta introducció a Javascript, pel que en aquest document ens centrarem en la persistència mitjançant fitxers.

Nodejs es un entorn molt modularitzat, i es compon de diverses llibreries i paquets (mòduls) que podem utilitzar als nostres projectes.

Nodejs incorpora alguns mòduls integrats al nucli (built-in modules), que proporcionen funcionalitat genèrica a moltes aplicacions. Algunes d'elles són les llibreries `http` o `https`, per crear servidors web, `utils`, amb funcions d'utilitat, `os`, per obtenir informació del sistema, o la que ens ocuparà gran part d'aquest document: la llibreria `fs` per a la gestió de fitxers.

1.2. El mòdul integrat `fs`

El mòdul `fs` de nodejs https://nodejs.org/api/fs.html ens ofereix una API per interactuar amb el sistema de fitxers, segons les funcions estàndards de POSIX.

Per tal d'utilitzar el mòdul, farem:

```
1 const fs = require('fs');
```

Cal dir també, que totes les funcions que se'ns proporcionen des de `fs` pe a l'accés al sistema de fitxers tenen la seua versió síncrona o asíncrona.

Quan s'utilitzen les formes asíncrones cal indicar com a últim argument una *funció de callback de finalització (completion callback)*, el primer argument de la qual estarà sempre reservat per a una excepció, i serà **`null`** o `undefined`quan l'execució ha estat exitosa.

Podem trobar tota una llista sobre els mètodes oferits per aquesta llibreria a l'enllaç especificat més amunt. Anem a veure ara alguna dels de major utilitat en quant a propòsit general.

1.3. Llistat de directoris: `fs.readdir()` `fs.readdirSync()`:

Anem a veure com a primer exemple introductori com llistariem els fitxers d'un directori, i de pas, fer un repàs als bucles en js:

```
// Importem la llibrería amb "require"
const fs = require('fs');
// Definim en una constant (podría ser una variable) el path
const path = '/etc';

// Invoquem al método readdirSync() passant com a paràmetre el path
// El resultat (serà un vector) l'assignem amb "let" a una variable
let fitxers=fs.readdirSync(path);

// I finalment, recorrem el vector, mostrant els components
// per la consola, amb console.log
for (let index=0;index<fitxers.length;index++){
    console.log(fitxers[index]);
}
```

En l'exemple podem veure algunes peculiaritats, com:

- La importació de llibreríes al nostre codi l'hem de fer mitjançant la funció `require('mòdul')`, i l'hem assignada a una constant `fs` a través de la qual accedirem a la llibrería. A més, cal dir que aquest és un mòdul *predefinit* al sistema, de manera que només cal importar-lo, però no hem de fer cap instal·lació externa amb *npm install*.
- No hem hagut de definir tipus de variables, ja que javascript suporta tipat dinàmic, i els tipus de les variables canvien en funció del contingut.
- Les variables suporten diferents àmbits: si les definim sense cap paraula reservada, són d'àmbit global (visibles a tot el codi), mentre que si les definim amb les paraules reservades `var` o `let`, són d'àmbit local: Amb `var` les definim locals a la funció, i amb `let` locals al bloc de codi on es defineixen (com és el cas).

Com veiem, la funció readFileSync ens torna els fitxers del directori en un vector, que podem recórrer de diferents formes mitjançant Javascript. A l'exemple anterior hem vist la forma més clàssica de l'estructura **`for`**, i coneixem també les estructure **`while`** i **`do..while`**. A continuació veiem un parell d'alternatives del **`for`** amb les que també podem recórrer els elements d'un vector: **`for`**`..in` i **`for`**`..of`:

```
// Forma 2: for (item in vector)
for (index in fitxers){
    console.log(fitxers[index]);
}
```

En la forma **`for`**`..in`, recorrem els índex del vector, pel que haurem d'accedir a la posicó indicada per l'índex per obtindre el contingut d'aquest.

```
// Forma 3: for (item of vector)
for (fitxer of fitxers){
    console.log(fitxer);
```

```
4 }
```

En aquest cas, la construcció **`for`**`..of` ens oferix la possibilitat de recórrer directament els elements del vector, sense necessitat d'accedir directament a ell.

Hi ha una tercera forma amb què podríem recórrer aquest vector, i és fent ús del mètode forEach del vector:

```
1 // Forma 4: forEach
2 fs.readdirSync(path).forEach(function(fitxer) {
3     console.log(fitxer);
4 });
```

Com veiem, el vector que ens retorna readdirSync es pot recórrer amb el mètode forEach. Aquest rep com a paràmetre una funció anònima, que serà invocada per a cadascun dels diferents elements del vector, al que hem anomenat `fitxer`.

Aquesta última implementació, encara es pot *endolçar sintàcticament* amb *funcions fletxa*. Les expressions de *funcions fletxa* tenen una sintaxi més curta, són sempre anònimes, no estan relacionades amb mètodes i no es poden utilitzar com a constructors. El bucle anterior amb aquest tipus de funcions, quedaria:

```
1 // Forma 4b: forEach amb funcions fletxa
2 fs.readdirSync(path).forEach(fitxer => {console.log(fitxer);});
```

Com veiem, hem suprimit la paraula `function`, i hem indicat directament el paràmetre que rep, seguit d'una fletxa i el cos de la funció.

Podem trobar més opcions sobre aquest mètode a la documentació oficial de nodejs

Readdir() A l'exemple anterior hem vist el mètode síncron `fs.ReaddirSync()`, però aquesta funcionalitat també l'oferix un mètode en versió asíncrona: `fs.ReadDir()`. La seua sintaxi és la següent:

```
1 fs.readdir(path[, options], callback)
```

Sent paràmetres obligatoris el primer, que és el path del que volem llegir el directori, i l'últim, que és la funció de callback.

Veiem un exemple de funcionament d'aquest mètode:

```
1 const fs = require('fs');
2
3 const path = '/etc';
4
5 console.log("Inici del programa principal");
```

```
6  fs.readdir(path, function(err, files) {
7      for (let file of files) {
8          console.log(file);
9      }
10 });
11 console.log("Final del programa principal");
```

Si llancem el programa, veurem que les dues primeres línies que apareixen són `Inici del programa principal` i `Final del programa principal`, ja que la invocació a la funció de callback s'ha afegit a la cúa d'events, i no serà executada fins que el programa principal no acabe.

En alguns llocs (en la documentació oficial de nodejs, entre ells), també podrem trobar-nos les funcions anònimes de callback expressades mitjançant funcions fletxa:

```
1  console.log("Inici del programa principal");
2  fs.readdir(path, (err, files) => {
3      for (let file of files) {
4          console.log(file);
5      }
6  })
7  console.log("Final del programa principal");
```

Que tot i ser sintàcticament diferent a l'anterior, la semàntica és exactament la mateixa.

1.4. Esborrat, renomenat i còpia de fitxers

El mòdul `fs` ens permet manipular també el sistema de fitxers, tant de forma síncrona com asíncrona. Veiem els principals mètodes per tal d'aconseguir-ho:

Mètode	Significat	Síncron/Asíncron
fs.rename(PathAnterior, PathNou, callback)	Renomena el fitxer al PathAnterior pel Path Nou	Asíncron
fs.renameSync(PathAnterior, PathNou)	Renomena el fitxer al PathAnterior pel Path Nou	Síncron
fs.rmdir(path, callback)	Esborra el directori especificat al path	Asíncron
fs.rmdirSync(path)	Esborra el directori especificat al path	Síncron
fs.unlink(path, callback)	Esborra el fitxer indicat al path	Asíncron

Mètode	Significat	Síncron/Asíncron
fs.unlinkSync(path)	Esborra el fitxer indicat al path	Síncron
fs.copyFile(orige, desti[, perm], callback)	Copia un fitxer de l'orige al destí	Asíncron
fs.copyFileSync(orige, desti[, perm])	Copia un fitxer de l'orige al destí	Síncron
fs.stat(path[, opcions], callback)	Comprova l'existència d'un fitxer, i convé utilitzar-los abans de fer qualsevol operació sobre fitxers o directoris	Asíncron
fs.statSync(path[, opcions])	Comprova l'existència d'un fitxer, i convé utilitzar-los abans de fer qualsevol operació sobre fitxers o directoris	Síncron

Per tal d'exemplificar com s'utilitzaríen aquests mètodes, veiem un xicotet exemple per als mètodes per renomenar fitxers. La resta de mètodes s'utilitzaríen de la mateixa forma:

- **Renomenar fitxers de forma asíncrona:**

```
fs.rename('/tmp/fitxer1', '/tmp/fitxer2', (err) => {
  if (err) throw err;
  console.log("S'ha renomenat el fitxer");
});
```

Com veiem, rename ens pot tornar un error, que en aquest cas, llançarem com una excepció amb **throw**.

- **Renomenar fitxers de forma síncrona:**

```
try {
    fs.renameSync('/tmp/fitxer1', '/tmp/fitxer2');
    console.log("S'ha renomenat el fitxer");
} catch  (err){
    console.log("Error"+err);
}
```

Com veiem, en aquest cas, hem inclòs l'ordre de renomenar dins un **try-catch** per tal de gestionar els errors.

- **De forma asíncrona:**

```
fs.rename(PathAnterior, PathNou, callback)
```

Exemple:

```
fs.rename('/tmp/fitxer1', '/tmp/fitxer2', (err) => {
  if (err) throw err;
  console.log("S'ha renomenat el fitxer");
});
```

Com veiem, rename ens pot tornar un error, que en aquest cas, llançarem com una excepció amb **throw**.

- **De forma síncrona:**

```
fs.renameSync(PathAnterior, PathNou)
```

Exemple:

```
try {
    fs.renameSync('/tmp/fitxer1', '/tmp/fitxer2');
    console.log("S'ha renomenat el fitxer");
} catch  (err){
    console.log("Error"+err);
}
```

Com veiem, en aquest cas, hem inclòs l'ordre de renomenar dins un **try-catch** per tal de gestionar els errors.

2. Lectura de fitxers

La forma més senzilla de llegir un fitxer és fer ús de `fs.readFile()`. Per exemple, per llegir un fitxer i mostrar-lo per pantalla faríem:

```
const fs = require('fs');
fs.readFile('/etc/passwd', function(err, data) {
    console.log(data);
  });
```

En aquest cas, si executem l'script anterior, obtindrem:

```
$ nodejs fs.js
```

```
<Buffer 72 6f 6f 74 3a 78 3a 30 3a 30 3a 72 6f 6f 74 3a 2f 72 6f 6f 74
   3a 2f 62 69 6e 2f 62 61 73 68 0a 64 61 65 6d 6f 6e 3a 78 3a 31 3a 31
    3a 64 61 65 6d 6f ... >
```

És a dir, un objecte de tipus buffer, amb els bytes que formen el fitxer. Si volem espeficicar la codificació d'aquest, haurem d'indicar-ho com a opcions, de la següent forma:

```
const fs = require('fs');
fs.readFile('/etc/passwd', 'utf-8',function(err, data) {
    console.log(data);
  });
```

Amb el que indiquem que el format del fitxer a llegir s'ha d'interpretar amb una codificació UFT-8.

Fixem-nos també en que la funció de callback, rep dos paràmetres per part de readFile: `err` que representa els errors que s'hagen produït, i `data`, amb la resposta, en aquest cas, el contingut del fitxer. Si no hi ha error o resposta, aquesta valors prendran el valor **`null`**.

A l'igual que la resta de mètodes sobre el sistema de fitxers, aquest mètode té la seua variant síncrona:

```
fs.readFileSync(path[, optcons])
```

2.1. Lectura línia a línia i paraula a paraula

Una manera senzilla de llegir línia a línia un fitxer fent ús només de les llibreríes que coneixem sería la següent:

```
const fs=require("fs");
const os=require("os");
fs.readFile('/etc/passwd', 'utf-8',function(err, data) {
    line="";
    for (car of data){
        if (car==os.EOL){
            console.log(line);
            line="";
        } else line=line+car;
    }
  });
```

Amb açò, en fer el **`for`** `car of data`, llegiríem el fitxer caràcter a caràcter, aniríem afegint els caràcters a la línia actual, fins que detectàrem el final de la línia (amb `os.EOL`). En trobar aquest caràcter, mostraríem la línia i la tornaríem a deixar en blanc per emmagatzemar la següent.

Per altra banda, si volguérem llegir paraula a paraula, faríem ús del mateix codi, però reemplaçant el caràcter `os.EOL` per un espai " ": **`if`** `(car=="")`{.

Per altra banda, existeix un mòdul anomenat `readline` https://nodejs.org/api/readline.html que realitza aquesta fncionalitat. Veiem un exemple d'ús d'aquest:

```
1
2  // Importem els mòduls fs i readline
3  const fs=require("fs");
4  var lr = require('readline');
5
6  // Creem un readStream a partir del fitxer a llegir
7  var rs=fs.createReadStream('/etc/passwd');
8  // Creem una interfície per llegir el readStream
9  reader=lr.createInterface(rs);
10
11 // Quan hi haja una línia preparada al reader, la imprimim
12 reader.on('line', function (line) {
13     console.log('Line from file:', line);
14 });
```

3. Escriptura de fitxers

Per tal de crear fitxers, nodejs ens ofereix tres mètodes diferents: `appendFile`, `open` i `writeFile`.

3.1. fs.appendFile

El mètode appendFile serveix per afegir contingut a un fitxer. La seua sintaxi és:

```
1  fs.appendFile(path, data[, options], callback)
```

La forma més senzilla d'utilitzar-lo és com es veu al següent exemple:

```
1  const fs=require("fs");
2  fs.appendFile('fitxer', 'Informació a escriure al fiter\nSegona línia
       del fitxer', function(err) {
3    if (err) throw err;
4    console.log("S'han afegit dades al fitxer");
5  });
```

Si el fitxer al que volem afegir informació no existeix, aquest es crearà.

Per altra banda, també podem fer ús d'un descriptor de fitxer obtingut amb «open», que caldrà tancar després de fer l'escriptura:

```
1  const fs=require("fs");
2
3  fs.open('fitxer', 'a', function(err, fd) {
4    // En fd tindrme el descriptor de fitxer que ens passa open
5    if (err) throw err;
```

```
6    // I haurem d'indicar en appendFile el descriptor del fitxer
7    fs.appendFile(fd, '\nDades noves per al fitxer', 'utf8', function(err
        ) {
8      if (err) throw err;
9      fs.close(fd, function(err) {
10       if (err) throw err;
11     });
12
13   });
14 });
```

Com veiem, en aquest cas, hem hagut de declarar tres funcions de callback anidades. Aquesta situació, lleva legibilitat al codi, i quan el nombre de callbacks anidats creix, es produeix el conegut *Callback hell*.

3.2. fs.appendFileSync

El mètode appendFile també ofereix la seua versió síncrona amb appendFileSync:

```
1 fs.appendFileSync(path, dades[, opcions])
```

Que s'usaria:

```
1 fs=require("fs");
2 try {
3     fs.appendFileSync('fitxer', 'Dades a afegir');
4     console.log("S'han afegit dades al fitxer");
5   } catch (err) {
6       console.log ("Excepció "+err.code+": "+err.message);
7   }
```

En aquest cas, hem inclòs la instrucció dins un bloc **`try..catch`**. L'excepció que es recull (err), és un objecte de tipus excepció, que podem imprimir directament, o accedir als seus components de codi d'error i missatge, com hem fet a l'exemple.

A l'igual que amb `appendFile`, també podem obrir el fitxer i fer ús d'un descriptor:

```
1 let fd;
2
3 try {
4   fd = fs.openSync('fitxer', 'a');
5   fs.appendFileSync(fd, 'Dades a afegir', 'utf8');
6 } catch (err) {
7   console.log ("Excepció "+err.code+": "+err.message);
8 } finally {
9   if (fd !== undefined)
10    fs.closeSync(fd);
11 }
```

3.4. writeFile i writeFileSync

Els mètodes `writeFile` i `writeFileSync` creen o sobreescriuren un fitxer. El seu ús és pràcticament el mateix que `appendFile` i `appendFileSync`:

Exemple amb `writeFile`

```
var fs = require('fs');

fs.writeFile('fitxer', 'Dades a afegir', function (err) {
  if (err) throw err;
  console.log("S'han guardat les dades.");
});
```

Exemple amb `writeFileSync`

```
fs=require("fs");
try {
    fs.writeFileSync('fitxer', 'Dades a afegir');
    console.log("S'ha escrit el fitxer");
  } catch (err) {
      console.log ("Excepció "+err.code+": "+err.message);
  }
```

4. Fitxers JSON

Si recordem, JSON és un format de fitxer per emmagatzemar dades basat en la sintaxi d'objectes de Javascript, de manera que el seu tractament resulta bastant natural.

Anem a veure-ho amb un exemple. Disposem del següent fitxer JSON:

```
{
    "curs": [
            {
                "nom": "Accés a Dades",
                "hores": 6,
                "qualificacio":  8.45
            },
            {
                "nom": "Programació de serveis i processos",
                "hores": 3,
                "qualificacio": 9.0
            },
            {
                "nom": "Desenvolupament d'interfícies",
                "hores": 6,
                "qualificacio": 8.0
            },
```

```
            {
                "nom": "Programació Multimédia i dispositiud mòbils",
                "hores": 5,
                "qualificacio": 7.34
            },
            {
                "nom": "Sistemes de Gestió Empresarial",
                "hores": 5,
                "qualificacio": 8.2
            },
            {
                "nom": "Empresa i iniciativa emprenedora",
                "hores": 3,
                "qualificacio": 7.4
            }
        ]
}
```

La seua lectura i tractament es realitzaría de la següent forma:

```
const fs=require("fs");
fs.readFile("notes.json", function(err, data){
    let arrel = JSON.parse(data);
    for (i in arrel.curs){
        let modul=arrel.curs[i].nom;
        let hores=arrel.curs[i].hores;
        let nota=arrel.curs[i].qualificacio;
        console.log(modul+" ("+hores+" hores): "+nota);
    }
});
```

Com veiem, només hem hagut de fer ús del mètode `JSON.parse`, per tal de *parsejar* el fitxer en un objecte JSON.

Si volguèrem afegir elements o modificar el fitxer, podríem fer:

```
fs.readFile("notes.json", function(err, data){
    let arrel = JSON.parse(data);

    // Anem a modificar la nota d'accés a dades:
    for (i in arrel.curs){
        if (arrel.curs[i].nom=="Accés a Dades")
         arrel.curs[i].qualificacio=9;
    }

    // I afegim ara un nou registre:

    let EDD={ "nom": "Entorns de Desenvolupament",
              "hores": 3,
              "qualificacio": 10 }

```

```
16      arrel.curs.push(EDD);
17
18      // Ara convertim el JSON a un string
19      newJSON = JSON.stringify(arrel);
20      fs.writeFile('notes2.json', newJSON, 'utf8', function(err){
21          if (err) throw err;
22          console.log("S'ha modificat el fitxer correctament");
23      });
24  });
```

Com veiem, hem fet ús del mètode `push` per tal d'afegir un nou objecte JSON dins el JSON de notes, i per altra banda, hem utilitzat el mètode `JSON.stringify` per tal de convertir l'objecte JSON a un string i poder guardar-lo en disc amb `writeFile`.

5. Fitxers XML

Com hem vist, el format per a fitxers JSON és extrmadament còmode i fàcil d'utilitzar, ja que es tracta, per una banda d'un mòdul integrat, i d'un tractament d'elements natiu per al llenguatge (JSON=Javascript). Així i tot, i tot i que no ho vorem en aquest curs, nodejs disposa de mòduls externs que ens permeten treballar també amb fitxers XML.

El mòdul més conegut és el mòdul «xml», que podem trobar en https://www.npmjs.com/package/xml.

Si volem utilitzar-lo als nostres projectes, només haurem de descarregar-lo amb:

```
1  npm install xml
```

Açò ens crearà, si no la tenim, una carpeta anomenada node-modules, amb les llibreríes necessàries per al tractament de fitxers XML.

Si desitgeu aprofundir amb ell, podeu consultar la documentació del mòdul.