📚Module file System

Commençons par la méthode d'écriture.

1. write

🔪Le code  write.js crée un fichier 📄hello.txt dans le répertoire dans lequel vous vous trouvez lorsque vous exécutez le fichier.

 write.js

```
1. const fs = require('fs');
2. fs.writeFile('hello.txt', 'CJS', function (err) {
3.     if (err) return console.log('Error writing to file.');
4.     console.log('end');
5. })
```

Exécutez la commande >`node write.js`

Dans le cas des modules ESM, la syntaxe des imp/exp change ainsi que l'extension[1] du fichier.

[1] Sinon, modifier le fichier `package.json` `"type": "module",`

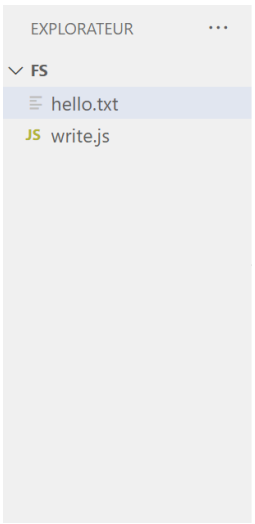
Créez un fichier write.**mjs** (notez bien le nom de l'extension mjs)

write.mjs

```
1. import { writeFile } from "fs";
2.
3. writeFile("hello.txt", "ESM", function (err) {
4.   if (err) return console.log("Error writing to file.");
5.   console.log("end");
6. });
```

Exécutez la commande >node write.mjs

🆒Voici l'arborescence des fichiers après exécution du programme à partir de son📂répertoire de *travail actuel*.



👹Remarques

CJS ESM

Chaque fois que vous invoquez une application Node, elle hérite de son📂répertoire de *travail actuel* à partir duquel vous exécutez votre code.

🥷Nous allons comprendre les conséquences de cette remarque.

Pour cela, nous exécutons le code précédent dans un emplacement différent de celui où se trouve le fichier write.js.

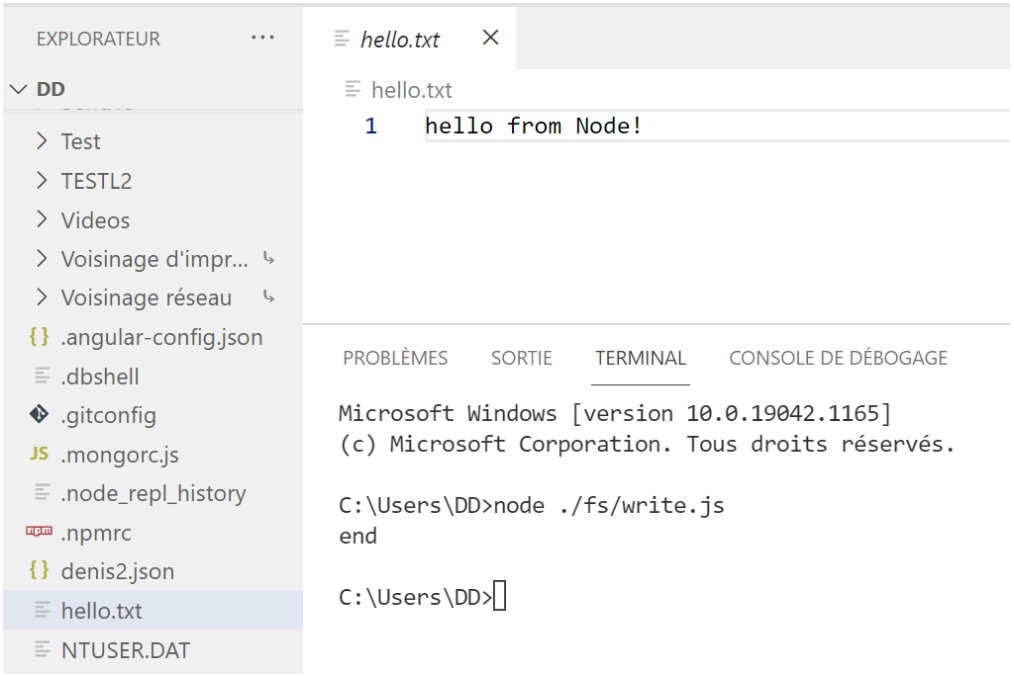
🪛Allez maintenant dans le répertoire 📂name[2] (ici DD) et exécutez le programme write.js

```
>pwd
C:/users/DD/fs
>cd ..
>node fs/write.js
```

🆒Vérifiez qu'il y a création du fichier *hello.txt*.

Voici l'arborescence après exécution, **le fichier `hello.txt` est à l'endroit où nous avons lancé le programme write.js**

[2] Vous pouvez également créer et vous déplacer dans un nouveau répertoire.



Nous venons de vérifier que le code crée un fichier à l'endroit où l'on exécute le code.

🥷Comment rendre la création du fichier indépendante de l'endroit où nous lançons le programme ?

__dirname

Common JS fournit, dans sa gestion des modules, une variable spéciale, **__dirname**, qui est toujours définie comme le répertoire dans lequel réside le fichier source.

L'utilisation de __dirname permet de rendre notre code indépendant du répertoire 📂où nous lançons l'exécution d'un programme.

🥷Remarque

__dirname[3] n'est pas disponible dans les modules EJS. Nous donnerons deux écritures équivalentes.

CJS ESM

write.js

```
1. const fs = require('fs');
2.
3. fs.writeFile(__dirname + '/hello.txt',
4.     'hello from Node!', function (err) {
5.         console.log(__dirname + '/hello.txt');
6.         if (err) return console.error('Error writing to file.');
7.     });
```

Lig. 3 __dirname vaut /home/name/fs (*le chemin où write.js est défini)*, on concatène __dirname avec /hello.txt

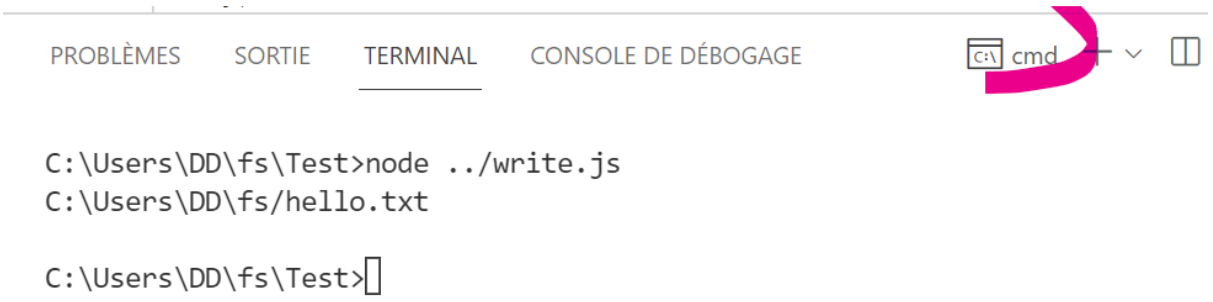
🆒Maintenant le fichier write.js crée le fichier *hello.txt* dans /home/name/fs (où le fichier write.js est défini) et ceci **indépendamment** de l'endroit où l'on lance le fichier write.js.

[3] __filename n'est pas disponible aussi.

Remarque : Les chemin d'accès sont différents suivant le terminal (cmd, bash …), vérifiez la valeur de l'affichage :

1. console.log(__dirname + '/hello.txt');

Pour le terminal cmd, l'affichage est C:\Users\DD\fs/hello.txt



🪛Nous pourrons utiliser le module path pour rendre le code indépendant du terminal avec :

write.js

```
1. const fs = require('fs');
2. const path = require('path');
3.
4. console.log(path.join(__dirname, 'hello.txt'));
5.
6. fs.writeFile(path.join(__dirname, 'hello.txt'),
7.     'hello from Node!', function (err) {
8.         if (err) return console.error('Error writing to file.');
9.     });
```

CJS ESM

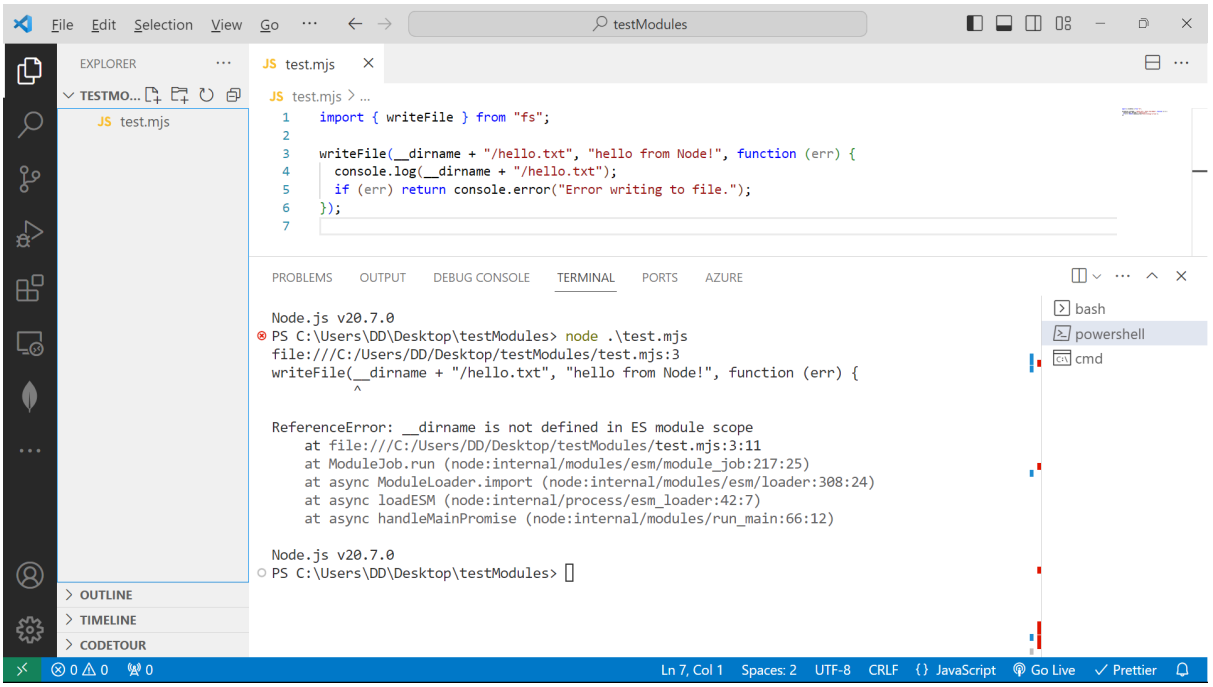
Passons à l'étude de notre code avec les modules ESM.

Je vous rappelle que vous devez changer l'extension du fichier (js->mjs)

🪛Ecrivons le fichier test.mjs

test.mjs

1. **import { writeFile } from "fs";**
2.
3. writeFile(__dirname + "/hello.txt", "hello from Node!", function (err) {
4. console.log(__dirname + "/hello.txt");
5. if (err) return console.error("Error writing to file.");
6. });



🪛Exécutez la commande> node test.mjs

Nous avons un message d'erreur[4] !

ReferenceError: __dirname is not defined in ES module scope

[4] https://flaviocopes.com/fix-dirname-not-defined-es-module-scope/

Une solution consiste à définir un équivalent de __dirname.

Nous pourrons écrire test.mjs de la façon suivante :

test.mjs

```
1. import { writeFile } from "fs";
2. import path from "path";
3. import { fileURLToPath } from "url";
4.
5. const __filename = fileURLToPath(import.meta.url);
6. const __dirname = path.dirname(__filename);
7.
8. writeFile(__dirname + "/hello.txt", "hello from Node!", function (err) {
9.   console.log(__dirname + "/hello.txt");
10. if (err) return console.error("Error writing to file.");
11.});
```

Lig. 6 : on définit la valeur de __dirname

🥷Une autre solution est de se passer de la valeur de __dirname[5]. Cette solution s'avère possible dans la plupart des cas.

Rappel : Dans un navigateur pour accéder à un fichier

C:\Users\DD\Desktop\testFetch\hello.txt on se sert de son URL

file:///C:/Users/DD/Desktop/testFetch/hello.txt

[5] le paramètre file peut en effet être soit un <string> | <Buffer> | <URL> | <FileHandle> filename or FileHandle https://nodejs.org/api/fs.html#fspromiseswritefilefile-data-options

https://url.spec.whatwg.org/#example-url-parsing

Finalement, nous pouvons écrire test.mjs de la façon suivante :

test.mjs

```
1. import { writeFile } from "fs";
2. import { URL } from "node:url";
3.
4. writeFile(
5.   new URL("./hello.txt", import.meta.url),
6.   "hello from Node!",
7.   function (err) {
8.     console.log(new URL("./hello.txt", import.meta.url));
9.     if (err) return console.error("Error writing to file.");
10.  }
11.);
```

Passons à la méthode read.

2. read

Le code read.js lit le fichier 📄hello.txt dans le 📂répertoire dans lequel ce trouve le fichier read.js.

La lecture d'un fichier donnera un résultat en hexadécimal (Buffer).

read.js

```
1. const fs = require('fs');
2. const path = require('path');
3. fs.readFile(path.join(__dirname, 'hello.txt'), function (err, data) {
4.     if (err) return console.error('Error reading file.');
5.     console.log('Read file contents:');
6.     console.log(data);
7. });
```

Lig. 6 : On pourrait utiliser String() pour retrouver le texte. testez en lig.6 : console.log(String(data));

Nous pourrons préciser l'encodage dans les arguments.

 read.js

```
1. const fs = require('fs');
2. const path = require('path');
3. fs.readFile(path.join(__dirname, 'hello.txt'),
4.     { encoding: 'utf8' },
5.     function (err, data) {
6.         if (err) return console.error('Error reading file.');
7.         console.log('Read file contents:');
8.         console.log(data);
9.     });
```

Passons à la méthode readdir.

readdir

Avec readdir on pourra lire les fichiers et les répertoires d'un répertoire.

readdir.js

```
1. const fs = require('fs');
2.
3. fs.readdir(__dirname, function (err, files) {
4.     if (err) return console.error('Unable to read directory contents');
5.     console.log(`Contents of ${__dirname}:`);
6.     console.log(files)
7. });
```

Lig. 6 : Un tableau est retourné contenant les noms des fichiers et répertoires.

Nous pourrons améliorer l'affichage en utilisant une méthode de transformation sur le tableau retourné avec la méthode **map**.

readdir.js

```
1. const fs = require('fs');
2.
3. fs.readdir(__dirname, function (err, files) {
4.     if (err) return console.error('Unable to read directory contents');
5.     console.log(`Contents of ${__dirname}:`);
6.     console.log(files.map( (f,i) => `\t ${i} : ${f}`).join('\n'));
7. });
```

Comment ne garder que des fichiers d'un certain type ?

Nous pouvons simplement utiliser un filtre. Avec comme fonction de filtre un test écrit avec une expression régulière.

Voici le test pour les fichiers .txt et .js

1. files.filter(f => /\.(txt|js)$/i.test(f));

La règle dit que le fichier fini ($) par .txt ou (|) .js

Voici le test pour tous les fichiers d'extension sauf (!) .txt

1. const txtFiles = files.filter(f => !/\.txt$/i.test(f));

Ecriture async/await

fetch.js

1. //const fetch = require('node-fetch') et npm install **node-fetch@2**
2. const _ = require('lodash');
3. const path = require('path');
4. const fs = require('fs');
5.
6. async function run() {
7. const response = await fetch("https://dev.to/api/articles?state=rising");
8. const json = await response.json();
9. const sorted = _.sortBy(json, ["public_reactions_count"], ['desc']);
10. const top5 = _.take(sorted, 5);
11.
12. const filePrefix = new Date().toISOString().split('T')[0];
13. fs.**writeFileSync**(path.join(__dirname, `${filePrefix}-feed.json`), JSON.stringify(top5, null, 2));
14. }
15.

```
16.run();
```

Autre exemple de code avec un fichier .json en local. Si le fichier data.json est dans le répertoire courant, on lit les données avec fs.readFile. Il est important de parser la réponse.

Repérez dans ce code l'utilisation de async/await.

 test.js

```
1. const _ = require('lodash');
2. const path = require('path');
3. const fs = require('fs').promises;
4.
5. async function run() {
6.   const response = await fs.readFile("data.json");
7.   const json = await JSON.parse(response);
8.   const sorted = _.orderBy(json, ["public_reactions_count"], ['desc']);
9.   const top1 = _.take(sorted, 1);
10.
11. const filePrefix = new Date().toISOString().split('T')[0];
12. await fs.writeFile(path.join(__dirname, `${filePrefix}-feed.json`),
    JSON.stringify(top1, null, 1));
13.}
14.
15.run();
```

Lien du code : https://github.com/dupontdenis/testNPM.git

Il vous faudra cloner le code

- npm install

- npm install lodash@latest // mise à jour de la bibliothèque
- node .\test.js

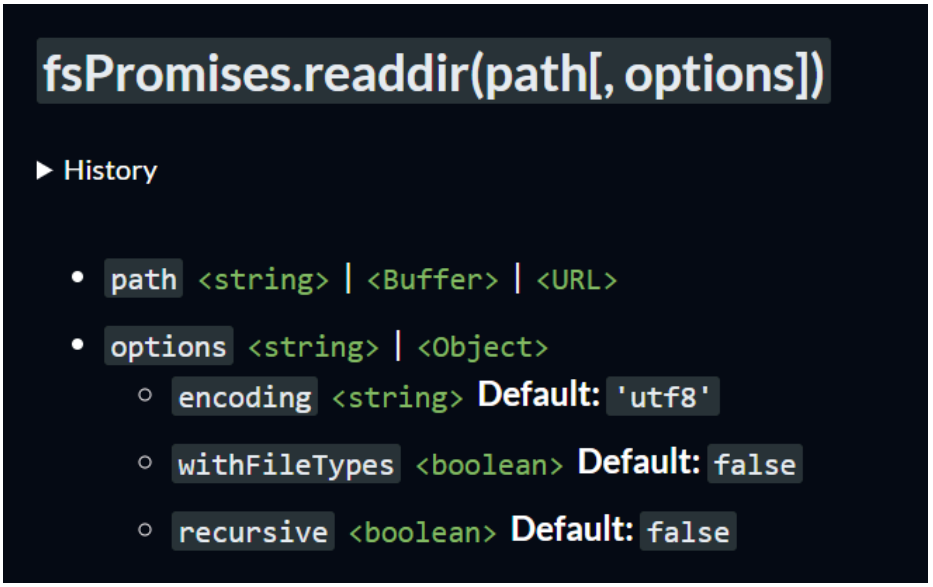
Suite au passage en EJS, notez l'extension de app.mjs (m = module)

app.mjs

```mjs
import fetch from "node-fetch";
import _ from "lodash";
import path from "path";
import fs from "fs";

import * as url from "url";
const __filename = url.fileURLToPath(import.meta.url);
const __dirname = url.fileURLToPath(new URL(".", import.meta.url));

async function run() {
  const response = await fetch("https://dev.to/api/articles?state=rising");
  const json = await response.json();
  const sorted = _.orderBy(json, ["public_reactions_count"], ["desc"]);
  const top5 = _.take(sorted, 3);

  const filePrefix = new Date().toISOString().split("T")[0];
  fs.writeFileSync(
    path.join(__dirname, `${filePrefix}-feed.json`),
    JSON.stringify(top5, null, 2)
  );
}

run();
```

Notez qu'avec le high level await, vous pourrez supprimer la fonction run !

🥷Les options des méthodes !



👹Il est important de lire la documentation pour dénicher les bonnes options.

Voici un exemple de code, où nous écrivons un programme récursif pour rechercher les fichiers .json dans toutes les sous-répertoires.

On utilise l'option **{ withFileTypes: true }**

🥈Version recursive

```
1. import fs from "node:fs/promises";
2. import { extname, join } from "node:path";
3.
4. export default async function findSalesFiles(folderName) {
5.   // this array will hold sales files as they are found
6.   let salesFiles = [];
7.
8.   async function findFiles(folderName) {
9.     // read all the items in the current folder
```

```
10.    const items = await fs.readdir(folderName, { withFileTypes: true });
11.// iterate over each found item
12.    for (const item of items) {
13.      // if the item is a directory, it will need to be searched
14.      if (item.isDirectory()) {
15.        // call this method recursively, appending the folder name to
     make a new path
16.
17.        await findFiles(join(folderName, item.name));
18.      } else {
19.        // Make sure the discovered file is a .json file
20.        if (extname(item.name) === ".json") {
21.          // store the file path in the salesFiles array
22.          salesFiles.push(join(folderName, item.name));
23.        }
24.      }
25.    }
26.  }
27.  await findFiles(folderName);
28.  return salesFiles;
29.}
```

Voici une nouvelle version qui utilise simplement l'option **{ recursive: true }**

🥇version option:recursif

app.mjs

```
1.  import fs from "node:fs/promises";
2.  import { extname, join } from "node:path";
3.
```

```
4.  export default async function findSalesFiles(folderName) {
5.    // this array will hold sales files as they are found
6.    let salesFiles = [];
7.
8.    const items = await fs.readdir(folderName, { recursive: true });
9.
10.  for (const item of items) {
11.    // Make sure the discovered file is a .json file
12.    if (extname(item) === ".json") {
13.      // store the file path in the salesFiles array
14.      salesFiles.push(join(folderName, item));
15.    }
16.  }
17.  return salesFiles;
18.}
```

Le fichier importe les deux types de recherche et affiche les fichiers de type json.  compare.mjs

```
1.  import find from "./findFiles.mjs";
2.  import recursif from "./findRecursifFiles.mjs";
3.
4.  const tab = await recursif("./stores");
5.  console.table(tab);
6.
7.  const tab2 = await find("./stores");
8.  console.table(tab2);
```

Les deux appels asynchrones affichent le même résultat !