et les modules de fichiers.

1.Objectifs

Les modules se répartissent en trois catégories :

1️⃣Les modules de base,

2️⃣Les modules de fichiers et

3️⃣Les modules npm.

🪛Nous allons découvrir que dans ***node,*** les fichiers JavaScript sont des modules fichiers utilisant un mécanisme simple de partage de données.

Nous mettrons l'accent ici sur

- 2️⃣**les modules de fichiers**

Parlons rapidement des environnements de développement que sont le navigateur et node.

2. Le navigateur 

Dans le navigateur, nous avons un objet global qui est **window**. Ceci nous permet entre autres de simplifier les écritures.

🪛Comparez dans la console d'un navigateur les deux écritures :

- window.console.table([{a:1,b:2},{a:5,b:10}]);
- console.table([{a:1,b:2},{a:5,b:10}]);

```
> window.console.table([{a:1,b:2},{a:5,b:10}]);
  console.table([{a:1,b:2},{a:5,b:10}]);
```

(index)	a	b
0	1	2
1	5	10

▶ Array(2)

(index)	a	b
0	1	2
1	5	10

▶ Array(2)

```
undefined
```

Comme, tout appartient à window, window.console.log est équivalent à console.log, et nous pourrons simplifier l'écriture en omettant window.

3.node

Dans node, *window* n'existe pas, son équivalent est l'objet global ***global***.

Ainsi, global.console.log("hello") est équivalent à console.log("hello").

De la même façon, global.setTimeout est équivalent à setTimeout.

4.Scope (portée)

Dans node, les variables définies dans un fichier sont privées. Il n'y a aucune communication entre deux fichiers par l'intermédiaire d'un objet global.

On peut imaginer chaque fichier javascript comme une boîte fermée ne permettant pas la collision de variables portant le même nom et déclarées dans des fichiers séparés.

Dans node chaque fichier est considéré comme un module.

5.Découverte du mécanisme de base[1]

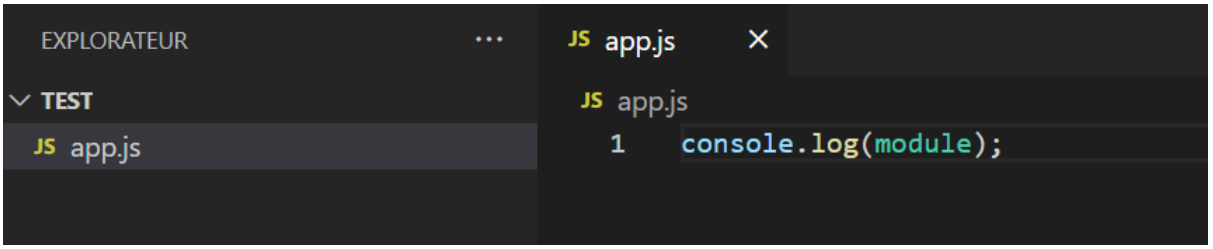
Une application node a au moins un module appelé module principal.

🔪Créez un fichier app.js dans un répertoire TEST. Ce fichier sera notre module principal.

Il a pour code :

app.js

```
1. console.log(module);
```



Pour exécuter app.js[2] lancer dans un terminal :

```
>node app.js
```

Nous affichons "module" qui est un objet JSON avec des pairs (key,value).

```
C:\Users\DD\Test>node app.js
```

Module

```
1. Module {
2.   id: '.',
3.   path: 'C:\\Users\\DD\\Test',
4.   exports: {},
```

[1] Ce mécanisme a un équivalent dans ES6
[2] On peut taper directement
C:\Users\DD\Test>**node**
> **module**

```
5.   parent: null,
6.   filename: 'C:\\Users\\DD\\Test\\app.js',
7.   loaded: false,
8.   children: [],
9.   paths: [
10.       'C:\\Users\\DD\\Test\\node_modules',
11.       'C:\\Users\\DD\\node_modules',
12.       'C:\\Users\\node_modules',
13.       'C:\\node_modules'
14.     ]
15.  }
```

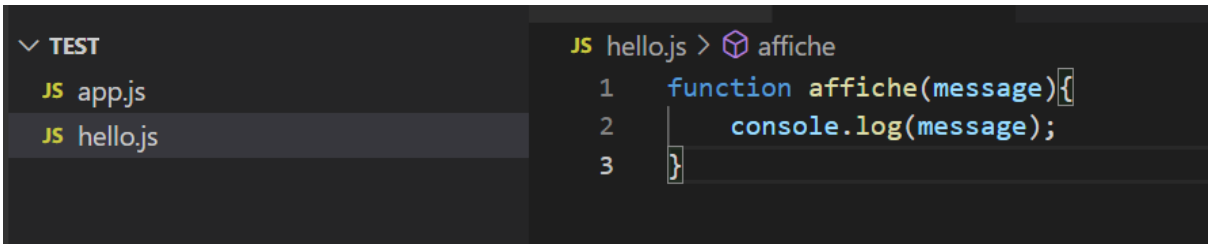
6.Création d'un module

🪛Ajoutez un fichier hello.js à notre application.

Ecrire le code suivant :

hello.js

```
1. function affiche(message){
2.     console.log(message);
3. }
```



la fonction *affiche* n'est pas visible par notre fichier app.js car son code est isolé par node.

💗 Nous allons voir comment rendre la fonction ***affiche*** disponible à l'extérieur. C'est à dire utilisable dans notre exemple dans le fichier app.js.

Mais, revenons à l'affichage de notre objet module. Nous voyons lig.4 une propriété exports qui est un objet.

```
1. Module {
2.   id: '.',
3.   path: 'C:\\Users\\DD\\Test',
4.   exports: {},
5.   parent: null,
6. …
7. }
```

🪛Rôle de l'environnement Node

Node met en place un mécanisme d'exportation/importation de ce module vers l'extérieur. **Ce mécanisme est transparent pour l'utilisateur.**

🆒 Il nous suffira d'exporter et d'importer les objets (fonction, valeur, objet …) que l'on souhaite partager en utilisant une syntaxe très simple.

Nous allons détailler les mécanismes d'exportation et d'importation.

7. 📤exports

🪛Modifions notre exemple hello.js pour illustrer la mécanique d'exportation.

Pour rendre notre fonction *affiche* visible à l'extérieur nous pouvons écrire :

📤**module.exports.affiche = affiche;** 

Le code du fichier  hello.js s'écrit.

 hello.js

```
1. function affiche(message){
2.     console.log(`${message}`);
3. }
4. module.exports.affiche = affiche
```

🥷Avec ce mécanisme, nous pouvons garder des variables et des fonctions privées et exporter les variables et fonctions de notre choix.

Exemple de variable ou constante privée

Dans l'exemple suivant, la constante ***urgence*** (lig. 1) n'est pas exportée. Elle est privée. C'est-à-dire que la constante urgence ne sera visible de l'extérieur que par l'intermédiaire de sa portée dans la fonction.

La constante **urgence** est visible à l'extérieur, mais elle ne sera pas modifiable.

🪛Modifiez le code du fichier  hello.js :

 hello.js

```
1. const urgence = "absolue";
2. function affiche(message){
3.     console.log(`${urgence} :${message}`);
4. }
```

5. module.exports.affiche = affiche

Après l'exportation, parlons de l'importation.

8. 📥require

Revenons au code du fichier app.js pour illustrer la mécanique d'importation.

Pour utiliser le code d'un module, nous appelons la fonction ***require***.

🪛Dans le fichier app.js ecrivez maintenant :

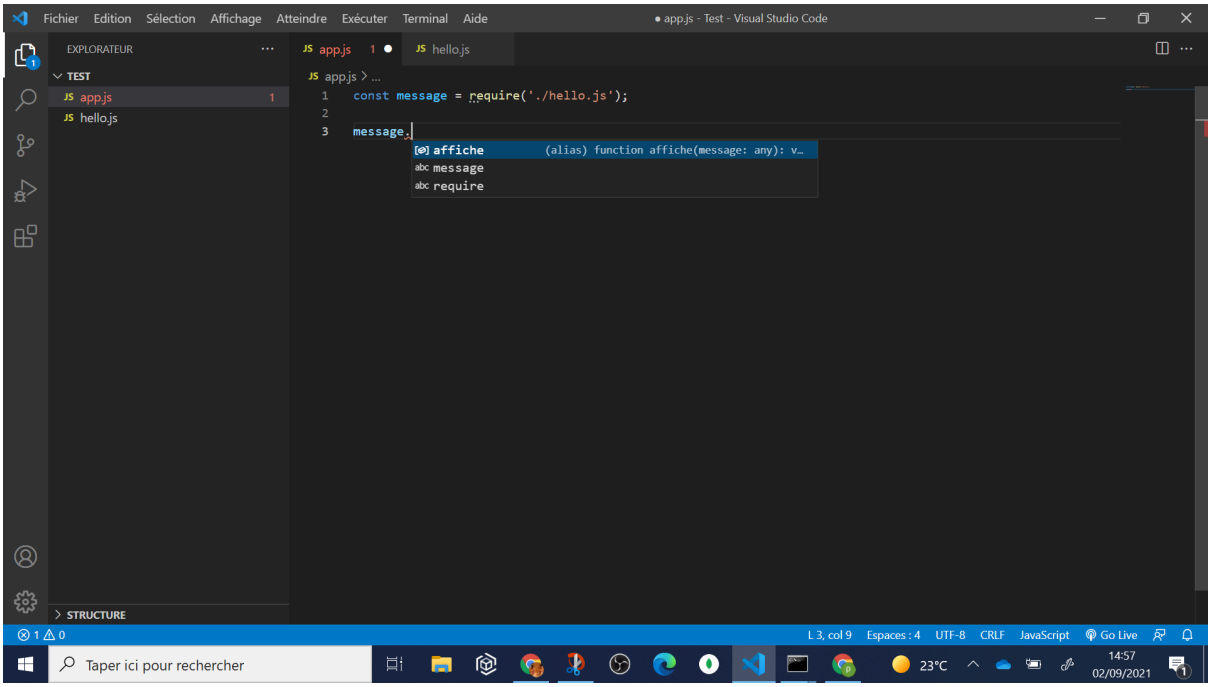
app.js

1. const message = require('./hello.js');
2. message.affiche('module');

Lig. 1 : On importe du code dispose dans hello.js.

lig. 2 : La fonction affiche du fichier hello.js par l'intermédiaire de l'objet message.

La figure suivante montre que VS permet la complétion du code. Ceci permettra une vérification de base en cas d'exportation.



🪛Exécutez l'application avec la commande `node app.js`.

```
C:\Users\DD\Test>node app.js
```

Nous vérifions que le code correspond à nos besoins.

On vérifiera également que la valeur de la variable urgence s'affiche, mais on ne peut pas ecrire *console.log(urgence)* dans le fichier app.js.

9.Autre écriture

🪛Dans le cas ou nous retournons uniquement une fonction ou une valeur, nous pouvons écrire directement le code suivant :

hello.js

```
1. const urgence = "absolue";
2. function affiche(message){
3.     console.log(`${urgence} : ${message}`);
4. }
5. module.exports = affiche;
```

Nous aurons pour le fichier app.js le code suivant pour la lig.1:

app.js

```
1. const log = require('./hello.js');
2.
3. console.log(typeof log)
4. log('module');
```

🥷Module : Le dessous des cartes+

Si vous connaissez IIFE https://developer.mozilla.org/fr/docs/Glossary/IIFE[3], il est facile de comprendre le mécanisme caché des modules.

Voici ce qui est appelé le "module wrapper function"

```
1. (function (exports, require, module, __filename, __dirname) {
2.   let exports = module.exports;
3.
4.   // notre code
5.
6.   return module.exports;
7. })(...);
```

Avant l'exécution de notre code, **node** enveloppe le code dans une fonction permettant ainsi de passer des variables.

Ainsi dans chaque module, nous disposons d'une variable __filename.

Remarques++

When you have *'type': 'module'* in the package.json file, your source code should use **import** syntax. When you do not have, you should use **require** syntax.

[3] Un bon exemple de ce à quoi "IIEF" est particulièrement utile se trouve dans la section Émulation de méthodes privées avec des fermetures de l'article Fermetures.

Node : Modules de fichiers p.11

Adding 'type': 'module' to the package.json enables ES6 modules. For more info, see here and here.

https://dupontnodejs.blogspot.com/2022/03/es6-module-in-nodejs.html