1.  Les modules Node

Les modules se répartissent en trois catégories :

1️⃣Les modules de base,

2️⃣Les modules de fichiers et

3️⃣Les modules npm.

Nous introduisons ici 1️⃣les modules de base. Ils sont fournis par Node lui-même.

Nous reviendrons en détail sur ces modules dans d'autres articles.

2. Path module

L'idée générale est d'écrire des modules pour qu'ils soient indépendants de toutes les plateformes.

👹L'utilisation de la concaténation de chaînes pour joindre des noms de fichiers n'est pas très compatible avec la plate-forme.

🆒Pour rappel, voici les deux prompts des interpréteurs bash et CMD.

Observez les séparateurs / et \ dans les deux lignes suivantes :

```
DD@DESKTOP-1UTD9HP MINGW64 ~/Desktop/nodejs
$
```

```
C:\Users\DD\Desktop\nodejs>
```

Core modules p.2

Voici la lecture du répertoire courant pour les lignes CMD et bash.

bash : pwd	CMD : cd
/c/Users/DD/Desktop/nodejs	C:\Users\DD\Desktop\nodejs

 Node fournit des utilitaires de nom de chemins indépendants de la plateforme.

🥷Le but est de pouvoir écrire des modules pour qu'ils soient indépendants de toutes les plateformes.

🔪Écrivez un fichier app.js. Vous testerez dans ce fichier différentes options.

app.js

```
1. const path = require('path');
2. console.log(typeof path) // objet
```

Lig.1 📥on importe le module path. Le module path est un objet disposant d'un certain nombre de propriétés et méthodes.

Core modules p.3

Vous allez étudier la méthode **parse** de l'objet path.

🪛Modifier le fichier app.js.

app.js

1. const path = require('path');
2. const pathObject = path.parse(__filename);
3. console.log(pathObject)

lig.2 : __filename donne l'adresse du fichier app.js

🪛Lancez la commande **>node app.js**

`l'affichage donne`[1] :

```
1. {
2.   root: 'C:\\',
3.   dir: 'C:\\Users\\xx\\Test',
4.   base: 'app.js',
5.   ext: '.js',
6.   name: 'app'
7. }
```

🆘__filename[2] n'existe pas pour les fichiers ES module.

Si vous utilisez __filename vous aurez un message du type :

*ReferenceError: __filename **is not defined** in ES module scope*

🥷Il vous faudra trouver un équivalent.

---

[1] Le résultat sera différent pour vous.
[2] et __dirname

🪛Commencez par créer un fichier app.mjs et ajouter le code :

app.mjs

```
1. import path from "path";
2. import { fileURLToPath } from "url";
3.
4. const __filename[3] = fileURLToPath(import.meta.url);
5.
6. const pathObject = path.parse(__filename);
7. console.log(pathObject);
```

>node app.mjs

Destructuration

Pour n'utiliser qu'une propriété d'un module, on peut utiliser la destructuration.

🪛Modifier le fichier app.js comme suit :

app.js

```
1. const {parse} = require('path');
2. const pathObject = parse(__filename);
```

On peut même renommer la destructuration.

🪛Modifiez le fichier app.js.

app.js

```
1. const { parse:decomposer } = require('path');
2. const pathObject = decomposer(__filename);
```

[3] const __dirname = path.dirname(__filename);

Passons à un autre module fourni par node  file module.

3. File module[4]

De nombreux livres d'introduction à la programmation traitent de l'accès aux systèmes de fichiers, car ils sont considérés comme une partie essentielle de la programmation.

Jusqu'à Node, JS n'était pas dans le club des systèmes de fichiers.

Le module fs permet d'interagir avec le système de fichiers d'une manière calquée sur les fonctions POSIX standard.

`Le module permet d'utiliser des méthodes synchronisées ou non. Ainsi, nous avons les méthodes accessSync ou accessAsync[5].`

`Voici un exemple de l'affichage des fichiers du répertoire courant` en fonction du type de synchronisation.

type : sync	type : async[6]

`const files = fs.readdirSync("./");`
`console.log(files);`
	`fs.readdir('./',function(err,files){`
	`  if (err) console.log(err);`
	`  else console.log(files);`
	`})`

[4] https://nodejs.org/api/fs.html
[5] Nous utiliserons de préférences les méthodes asynchrones.

[6] avec callback

4. Module events

La classe[7] EventEmitter est définie par le module events.

🪛Modifier le fichier app.js.

app.js

```
1. const EventEmitter = require('events');
2. const emitter = new EventEmitter();
3.
4. //ajout d'un écouteur
5. emitter.on('messageLogged', function(){
6.     console.log('listener called')
7. })
8.
9. //emission
10. emitter.emit('messageLogged')
```

Nous pouvons définir un grand nombre d'écouteurs, voici un exemple de code.

🪛Modifier le fichier app.js.

app.js

```
1. const EventEmitter = require('events');
2. const emitter = new EventEmitter();
3.
4. //ajout d'un écouteur
5. emitter.on('messageLogged', function(){
6.     console.log('listener called')
```

[7] Et non une fonction ou un objet

```
7. })
8.
9. //ajout d'un second écouteur
10. emitter.on('messageLogged', function test(){
11.     console.log('listener called')
12. })
13.
14. console.log(emitter.listeners('messageLogged'))
```

Lig. 14 : on récupère un tableau avec la liste des écouteurs sur l'événement *messageLogged* [[Function (anonymous)],[Function: test]]

Les arguments

On peut passer des arguments lors de l'émission d'événements en second paramètre : `emitter.emit('messageLogged', arg)`

l'arg peut être une liste d'arguments mais on préfère passer l'argument sous forme d'objet.

Voici un exemple où l'argument est un objet :

```
emitter.emit('messageLogged', {id:1, url:"http"});
```

Il est facile ensuite de récupérer l'argument dans la fonction de l'écouteur.

🪛Modifier le fichier app.js.

app.js

```
1. const EventEmitter = require('events');
2. const emitter = new EventEmitter();
3.
4. //ajout d'un écouteur
5. emitter.on('messageLogged', (arg) => {
```

```
    console.log(`listener called ${JSON.stringify(arg)}`)
})

//emission
emitter.emit('messageLogged', {id:1, url:"http"});
```

Passons au module souvent mis en avant : http.

5. Module http

`Voici un exemple de code montrant la méthode` *`on()`* héritée `de la classe listener` par http.

server.js

```
const http = require('http');
const server = http.createServer();

server.on("connection", (socket) => {
    console.log("new connection");
})
server.listen(3000);
```

Pour tester ce code il faut se connecter à http://localhost:3000/.

P.S. `Nous n'utilisons pas les sockets dans la suite du cours.`

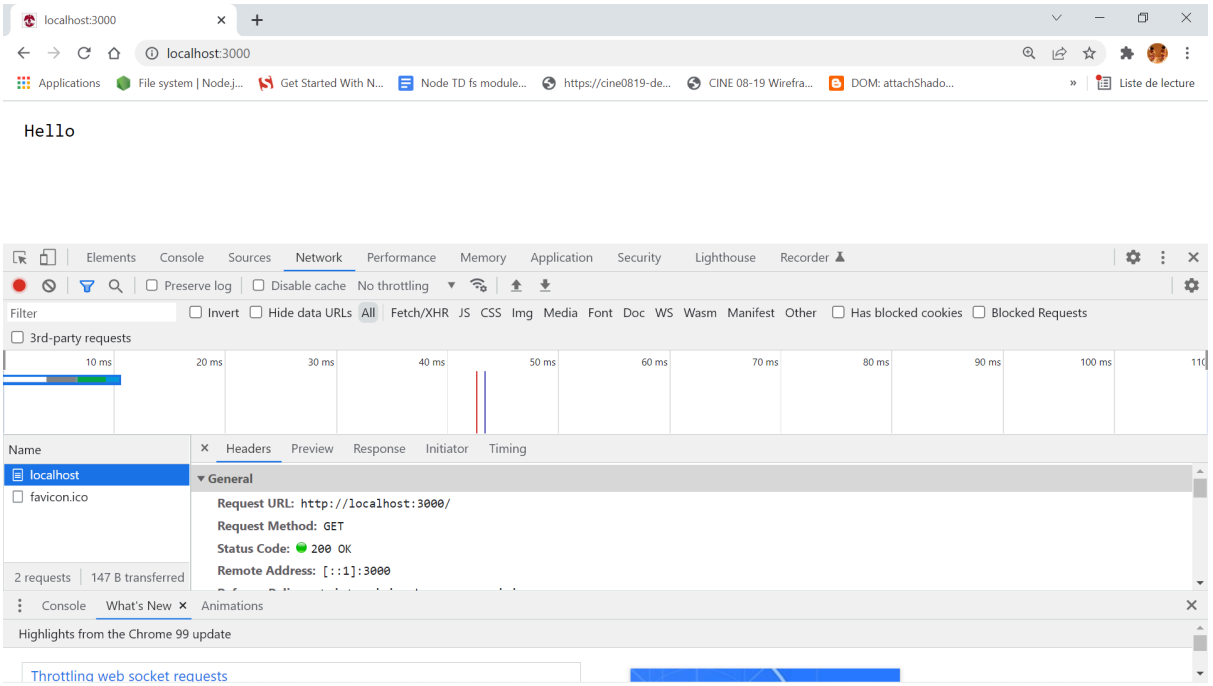
Création d'un serveur

Voici un code qui crée un serveur et affiche un message. Nous étudierons plus en détail ce code dans un prochain cours.

app.js

```
1. const http = require('http');
2.
3. const server = http.createServer( (req,res) => {
4.     if (req.url === '/') {
5.         res.write(" Hello")
6.         res.end();
7.     }
8. });
9. server.listen(3000);
```

Rendez vous à l'adresse : http://localhost:3000/



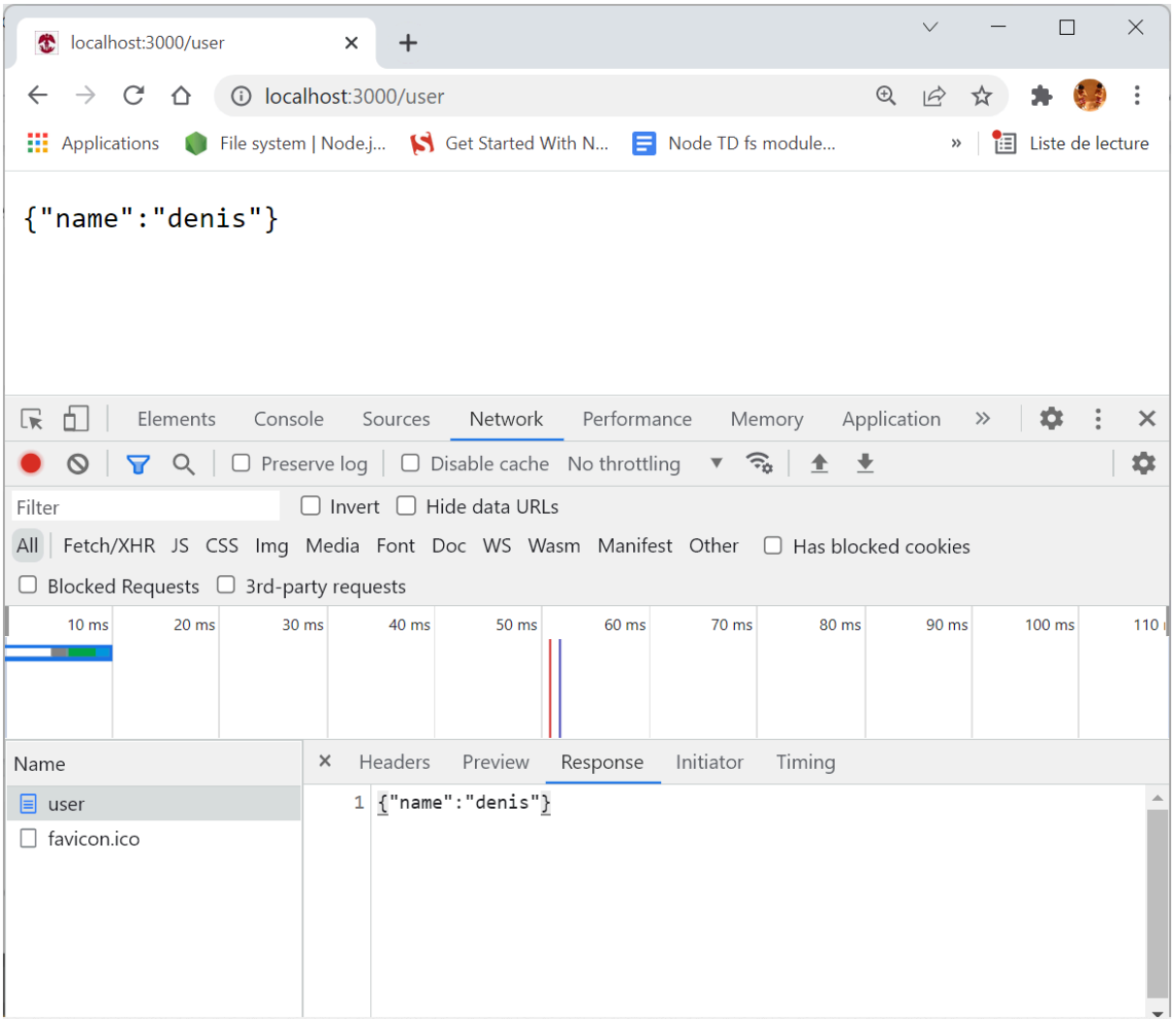
Voici un autre exemple avec une route en http://localhost:3000/user

Une route est un chemin vers un service.

app.js

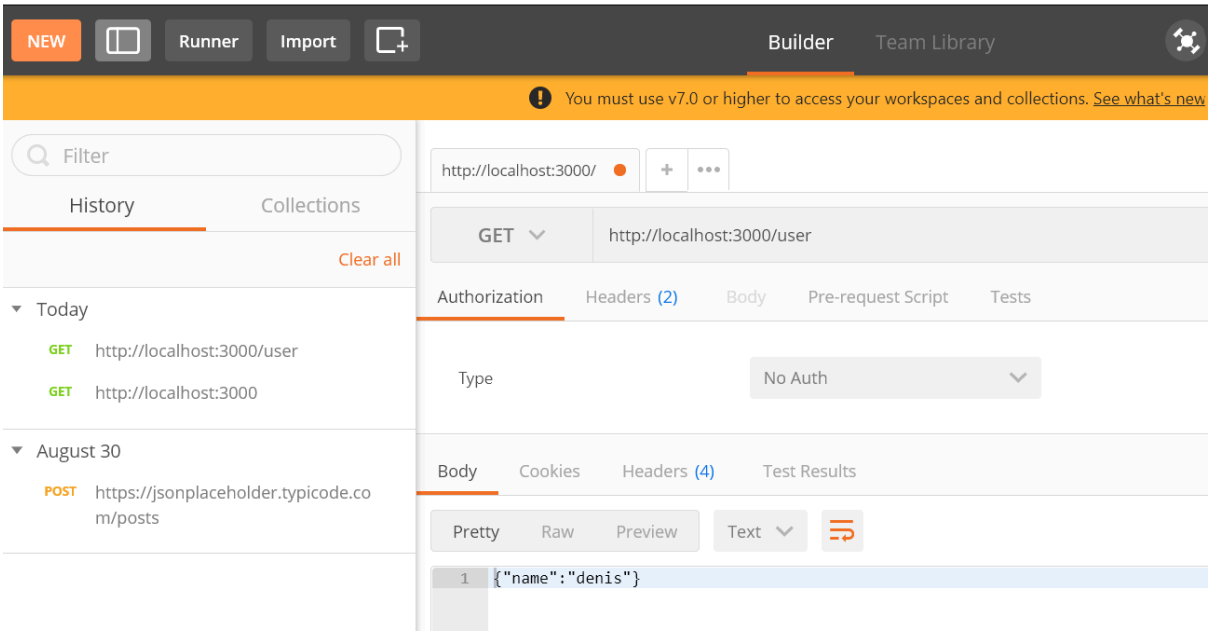
```
1. const http = require('http');
2.
3. const server = http.createServer( (req,res)=>{
4.     if (req.url === '/user') {
5.         res.write(JSON.stringify({"name":"denis"}))
6.         res.end();
7.     }
8. });
9.
10. server.listen(3000);
```

Rendez vous sur http://localhost:3000/user



Voici une capture d'écran sur postman.

Core modules p.12



Nous allons revenir en détail sur ces modules de base !

Annexe

Différence entre URL et path !

Exemples :

```
// URL
const fileURL = 'file:///C:/Users/username/Documents/2024-03-14-feed.json';


// Equivalent file path
const filePath = 'C:\\Users\\username\\Documents\\2024-03-14-feed.json';


// Equivalent file path in Git Bash
const filePath = '/c/Users/username/Documents/2024-03-14-feed.json';
```

Notez que dans les chemins de fichiers Windows, les barres obliques inverses (\) sont utilisées comme séparateurs, mais dans les chaînes JavaScript, les barres obliques inverses sont des caractères d'échappement, et doivent donc être échappées elles-mêmes avec une autre barre oblique inversée.

En action :

Un chemin d'accès est une chaîne de caractères qui spécifie l'emplacement d'un fichier dans un système de fichiers. Il peut être absolu ou relatif. Voici un exemple de lecture d'un fichier à l'aide d'un chemin d'accès dans Node.js :

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

const path = require('path');

// Absolute file path

const absoluteFilePath = path.join(__dirname, '2024-03-14-feed.json');

fs.readFile(absoluteFilePath, 'utf8', (err, data) => {

  if (err) {

    console.error(err);

    return;
```

```
  }

  console.log(data);

});
```

Une URL (Uniform Resource Locator) est une référence à une ressource web qui spécifie son emplacement sur un réseau informatique et un mécanisme pour la récupérer.

Dans Node.js, vous pouvez également utiliser les URL pour les chemins d'accès aux fichiers, ce qui est particulièrement utile pour les projets de développement.

```
import { readFile } from 'fs/promises';
import { fileURLToPath } from 'url';
import { dirname } from 'path';

// URL for the current module
const currentModuleURL = new URL(import.meta.url);

// Convert the URL to a file path
const currentModulePath = dirname(fileURLToPath(currentModuleURL));

// Use the file path to read a file
const data = await readFile(`${currentModulePath}/2024-03-14-feed.json`,
'utf8');
console.log(data);
```

Dans ce deuxième exemple, import.meta.url fournit une URL pour le module en cours. Cette URL est ensuite convertie en chemin de fichier à l'aide de fileURLToPath, et le chemin de fichier est utilisé pour lire un fichier.

La principale différence entre les deux est que les chemins d'accès aux fichiers sont spécifiques au système de fichiers et sont utilisés pour lire/écrire des fichiers, tandis que les URL sont plus généraux et peuvent être utilisés pour localiser des ressources sur Internet ou sur un réseau local, en plus du système de fichiers local.