TP 1: Gestion des entrées/sorties

Objectifs

- Gestion des E/S standard
- Gestion des fichiers avec le module **fs**
- Inspection des métadonnées du fichier
- Surveiller les mises à jour des fichiers
- Création de la communication TCP serveur et client
- Créer des flux dans Node.js
- Transformer des données avec “transform streams”
- Construire des pipelines de flux

Note importante:

Le TP est long tant que vous essayez d'écrire chaque exemple de code. Je recommande de copier et coller le code et de ne pas perdre de temps à l'écrire. Ce qui compte, c'est de comprendre la logique et le fonctionnement et non de mémoriser une syntaxe qui changera probablement au bout d'un ou deux ans.

Avant Node.js, JavaScript était principalement utilisé dans le navigateur. Node.js a apporté JavaScript au serveur et nous a permis d'interagir avec le système d'exploitation avec JavaScript. Aujourd'hui, Node.js est l'une des technologies les plus populaires pour créer des applications côté serveur.

Node.js interagit avec le système d'exploitation à un niveau fondamental : entrée et sortie. Ce TP explorera les API de base fournies par Node.js qui nous permettent d'interagir avec les E/S standard, le système de fichiers et la pile réseau.

Gestion des E/S standard

STDIN (standard in) fait référence à un flux d'entrée qu'un programme peut utiliser pour lire l'entrée d'un shell de commande ou d'un terminal. De même, STDOUT (standard output) fait référence au flux utilisé pour écrire la sortie. STDERR (standard error) est un flux distinct qui est généralement réservé à la sortie des erreurs et des données de diagnostic.

Créons un répertoire dans lequel travailler (tp1 par exemple). Dans ce répertoire, créez un fichier javascript "bonjour.js".

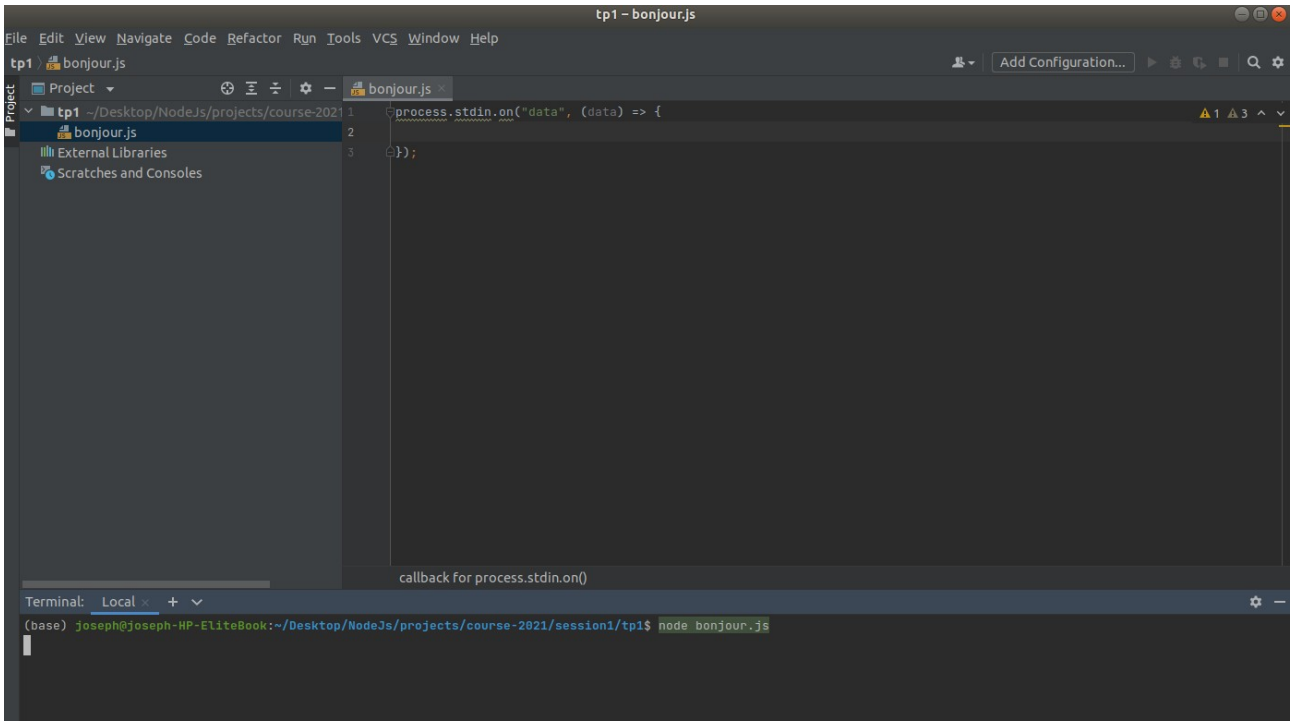
```
$ mkdir tp1
$ cd tp1
$ touch bonjour.js
```

Tout d'abord, nous devons dire au programme d'écouter les entrées de l'utilisateur. Cela peut être fait en ajoutant les lignes suivantes à bonjour.js

```
process.stdin.on("data", (data) => {
        //traitement sur chaque événement de données
});
```

Nous pouvons exécuter le fichier en utilisant la commande suivante. Notez que l'application ne se ferme pas car elle continue d'écouter les événements de données process.stdin:

```
node bonjour.js
```

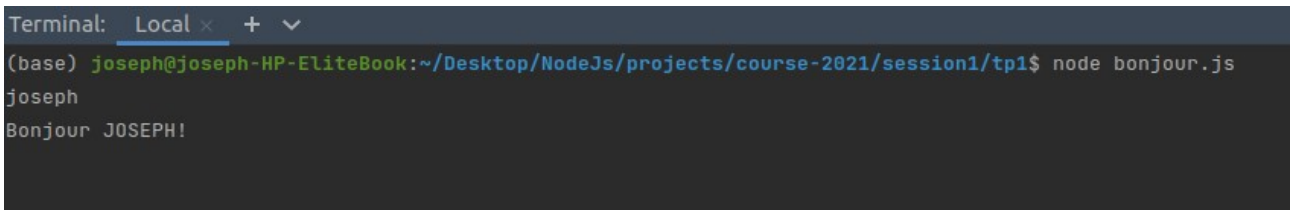


Vous pouvez utiliser webstorm ou tout autre IDE ou éditeur de texte.

Quittez le programme en utilisant CTRL + C. Nous pouvons maintenant dire au programme ce qu'il doit faire chaque fois qu'il détecte un événement de données. Ajoutez les lignes suivantes:

```
const nom = data.toString().trim().toUpperCase();
process.stdout.write(`Bonjour ${nom}!`);
```

Vous pouvez maintenant taper votre nom comme entrée dans votre programme, et il renverra le message d'accueil et votre nom en majuscules:



Nous pouvons maintenant ajouter une vérification pour savoir si la chaîne d'entrée est vide et ajouter un log STDERR si c'est le cas. Remplacez votre fichier par le suivant:

```
process.stdin.on("data", (data) => {
	const nom = data.toString().trim().toUpperCase();
	if ( nom !== "") {
		process.stdout.write(`Bonjour ${nom}!`);
	} else {
		process.stderr.write("L'entrée est vide.");
	}
});
```

```
(base) joseph@joseph-HP-EliteBook:~/Desktop/NodeJs/projects/course-2021/session1/tp1$ node bonjour.js
joseph
Bonjour JOSEPH!
L'entrée est vide.
L'entrée est vide.
```

process.stdin, process.stdout et process.stderr sont toutes des propriétés sur l'objet “process”. Un objet de processus global fournit les informations et le contrôle du processus Node.js. Pour chacun des canaux d'E/S, ils émettent des événements de données pour chaque bloc de données reçu. Dans cet exemple, nous exécutions le programme en mode interactif où chaque bloc de données était déterminé par le caractère de nouvelle ligne lorsque vous appuyez sur “Enter” dans votre shell. **process.stdin.on("data", (data) => {...});** est ce qui écoute ces événements de données. Chaque événement de données renvoie un objet Buffer. L'objet Buffer (généralement nommé data) renvoie une représentation binaire de l'entrée.

const nom = data.toString() est ce qui transforme l'objet Buffer en une chaîne. La fonction **trim()** supprime le caractère de nouvelle ligne qui dénotait la fin de chaque entrée.

Nous écrivons dans STDOUT et STDERR en utilisant les propriétés respectives sur l'objet “process” (**process.stdout.write, process.stderr.write**).

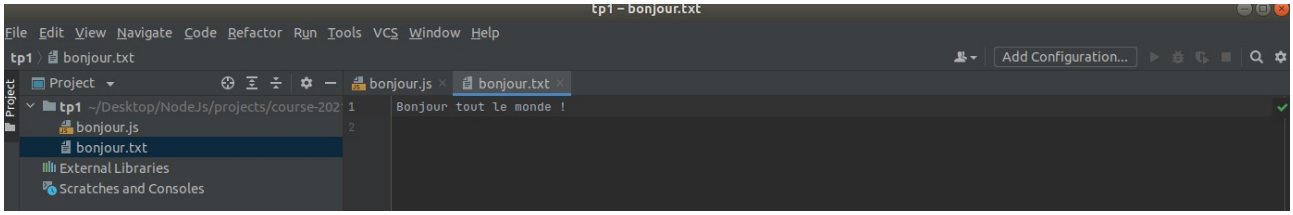
Dans cet exemple, nous avons également utilisé CTRL + C pour quitter le programme dans le shell. CTRL + C envoie SIGINT, ou interruption de signal, au processus Node.js. Pour plus d'informations sur les événements de signal, reportez-vous à la documentation de l'API Node.js Process à l'adresse https://nodejs.org/api/process.html#process_signal_events.

Gestion des fichiers avec le module fs

Node.js fournit plusieurs modules de base, y compris le module **fs**. **fs** signifie File System, et ce module fournit les API pour interagir avec le système de fichiers.

créons un fichier à lire. Exécutez ce qui suit dans votre shell pour créer un fichier contenant du texte simple:

```
$ echo Bonjour tout le monde ! > bonjour.txt
```



Nous aurons également besoin d'un fichier pour notre programme—créez un fichier nommé *readWriteSync.js*:

```
$ touch readWriteSync.js
```

Dans cet exemple, nous allons lire de manière synchrone le fichier nommé *bonjour.txt*, manipuler le contenu du fichier, puis mettre à jour le fichier à l'aide des fonctions synchrones fournies par le module **fs**:

Nous commencerons par importer les modules intégrés fs et path. Ajoutez les lignes suivantes à *readWriteSync.js*:

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

```
const path = require("path");
```

Créons maintenant une variable pour stocker le chemin du fichier *bonjour.txt* que nous avons créé précédemment:

```
const filepath = path.join(process.cwd(), "bonjour.txt");
```

Nous pouvons maintenant lire de manière synchrone le contenu du fichier en utilisant la fonction **readFileSync()** fournie par le module **fs**. Nous imprimerons également le contenu du fichier sur STDOUT en utilisant **console.log()** :

```
const contenu = fs.readFileSync(filepath, "utf8");
```

```
console.log("Contenu du fichier:", contenu);
```

Maintenant, nous pouvons éditer le contenu du fichier—nous allons convertir le texte en minuscules en majuscules:

```
const majContenu = contenu.toUpperCase();
```

Pour mettre à jour le fichier, nous pouvons utiliser la fonction **writeFileSync()**. Nous ajouterons également une instruction de log par la suite indiquant que le fichier a été mis à jour :

```
fs.writeFileSync(filepath, majContenu);

console.log("Fichier mis à jour.");
```

Exécutez votre programme avec la commande suivante :

```
$ node readWriteSync.js
```

```
(base) joseph@joseph-HP-EliteBook:~/Desktop/NodeJs/projects/course-2021/session1/tp1$ node readWriteSync.js
Contenu du fichier: Bonjour tout le monde !

Fichier mis à jour.
```

Vous avez maintenant un programme qui, une fois exécuté, lira le contenu de *bonjour.txt*, convertira le contenu du texte en majuscules et mettra à jour le fichier.

Les deux premières lignes nécessitent les modules de base nécessaires au programme.

const fs = require("fs"); importera le module principal du système de fichiers Node.js. La documentation de l'API pour le module Node.js File System est disponible à l'adresse https://nodejs.org/api/fs.html. Le module fs fournit des APIs pour interagir avec le système de fichiers à l'aide de Node.js. De même, le module core path fournit des APIs pour travailler avec les chemins de fichiers et de répertoires. La documentation de l'API du module de chemin est disponible à l'adresse https://nodejs.org/api/path.html.

Ensuite, nous avons défini une variable pour stocker le chemin du fichier *bonjour.txt* en utilisant la fonction **path.join()** et **process.cwd()**. La fonction **path.join()** joint les sections de chemin fournies en paramètres avec le séparateur pour la plate-forme spécifique (par exemple, / sur les environnements Unix et \ sur les environnements Windows).

process.cwd() est une fonction sur l'objet de processus global qui renvoie le répertoire actuel du processus Node.js. Dans ce programme, il s'attend à ce que le fichier *bonjour.txt* se trouve dans le même répertoire que le programme.

Ensuite, nous lisons le fichier à l'aide de la fonction **fs.readFileSync()**. On passe à cette fonction le chemin du fichier à lire et l'encodage, "utf8". Le paramètre d'encodage est facultatif : lorsque le paramètre est omis, la fonction renvoie par défaut un objet Buffer.

Pour effectuer la manipulation du contenu du fichier, nous avons utilisé la fonction **toUpperCase()** disponible sur les objets chaîne.

Enfin, nous avons mis à jour le fichier à l'aide de la fonction **fs.writeFileSync()**. Nous avons passé deux paramètres à la fonction **fs.writeFileSync()**. Le premier était le chemin d'accès au fichier que nous souhaitions mettre à jour, et le second paramètre était le contenu du fichier mis à jour.

Note importante:

Les APIs **readFileSync()** et **writeFileSync()** sont synchrones, ce qui signifie qu'elles bloqueront/retarderont les opérations simultanées jusqu'à ce que la lecture ou l'écriture du fichier soit terminée. Pour éviter le blocage, vous voudrez utiliser les versions asynchrones de ces fonctions.

Node.js a été développé en mettant l'accent sur l'activation du modèle d'E/S non bloquant. Par conséquent, dans de nombreux cas (sinon la plupart), vous souhaiterez que vos opérations soient asynchrones.

Aujourd'hui, il existe trois manières notables de gérer le code asynchrone dans Node.js : les **Callbacks, les Promises et la syntaxe async/await**. Les premières versions de Node.js ne prenaient en charge que le modèle de **Callbacks**. Les **Promises** ont été ajoutées à la spécification JavaScript avec ECMAScript 2015, connue sous le nom d'ES6, et par la suite, la prise en charge des **Promises** a été ajoutée à Node.js. Suite à l'ajout de la prise en charge de Promise, la prise en charge de la syntaxe **async/await** a également été ajoutée à Node.js.

Toutes les versions actuellement prises en charge de Node.js prennent désormais en charge les charge, les **Promises** et la syntaxe **async/await**. Explorons comment nous pouvons travailler avec des fichiers de manière asynchrone en utilisant ces techniques. La programmation asynchrone peut permettre à certaines tâches ou à certains traitements de se poursuivre pendant que d'autres opérations se produisent.

Le programme précédent a été écrit en utilisant les fonctions synchrones disponibles sur le module fs :

```
const fs = require("fs");
const path = require("path");
const filepath = path.join(process.cwd(), "bonjour.txt");
const contenu = fs.readFileSync(filepath, "utf8");
console.log("Contenu du fichier:", contenu);
const majContenu = contenu.toUpperCase();
fs.writeFileSync(filepath, majContenu);
console.log("Fichier mis à jour.");
```

Cela signifie que le programme a été bloqué en attendant la fin des opérations **readFileSync()** et **writeFileSync()**. Ce programme peut être réécrit pour utiliser les API asynchrones.

La version asynchrone de **readFileSync()** est **readFile()**. La convention générale est que les API synchrones auront le terme "sync" ajouté à leur nom. La fonction asynchrone nécessite qu'une fonction de rappel (**Callback**) lui soit transmise. La fonction de rappel contient le code que nous souhaitons exécuter lorsque la fonction asynchrone se termine.

La fonction **readFileSync()** dans cet exemple peut être modifiée pour utiliser la fonction asynchrone:

```
const fs = require("fs");
const path = require("path");
const filepath = path.join(process.cwd(), "bonjour.txt");
fs.readFile(filepath, "utf8", (err, contenu) => {
        if (err) { return console.log(err); }
      console.log("Contenu du fichier:", contenu);
      const majContenu = contenu.toUpperCase();
       fs.writeFileSync(filepath, majContenu);
       console.log("Fichier mis à jour.");
 }
);
```

Notez que tout le traitement qui dépend de la lecture du fichier doit avoir lieu à l'intérieur de la fonction de **Callback**.

La fonction **writeFileSync()** peut également être remplacée par la fonction asynchrone, **writeFile()** :

```
const fs = require("fs");
const path = require("path");
const filepath = path.join(process.cwd(), "bonjour.txt");
fs.readFile(filepath, "utf8", (err, contenu) => {
        if (err) { return console.log(err); }
      console.log("Contenu du fichier:", contenu);
      const majContenu = contenu.toUpperCase();
       fs.writeFile(filepath, majContenu, function (err) {
              if (err) throw err;
              console.log("Fichier mis à jour.");
       });
   }
);
```

Notez que nous avons maintenant une fonction asynchrone qui appelle une autre fonction asynchrone. Il n'est pas recommandé d'avoir trop de **Callbacks** imbriqués car cela peut avoir un impact négatif sur la lisibilité du code. Considérer ce qui suit:

```
first(args, () => {
    second(args, () => {
        third(args, () => {});
    });
});
```

Il existe des approches qui peuvent être adoptées pour éviter l'enfer du **Callbacks** imbriqués. Une approche serait de diviser les **Callbacks** en fonctions nommées. Par exemple, notre fichier pourrait être réécrit de sorte que l'appel **writeFile()** soit contenu dans sa propre fonction nommée, **updateFile()** :

```
const fs = require("fs");
const path = require("path");
const filepath = path.join(process.cwd(), "bonjour.txt");
fs.readFile(filepath, "utf8", (err, contenu) => {
  if (err) {
    return console.log(err);
  }
  console.log("Contenu du fichier:", contenu);
  const majContenu = contenu.toUpperCase();
  updateFile(filepath, majContenu);
});
function updateFile(filepath, contenu) {
  fs.writeFile(filepath, contenu, (err) => {
    if (err) throw err;
    console.log("Fichier mis à jour.");
  });
}
```

Pour démontrer que ce code est asynchrone, nous pouvons utiliser la fonction **setInterval()** pour imprimer une chaîne à l'écran pendant l'exécution du programme. La fonction **setInterval()** vous permet de planifier une fonction pour qu'elle se produise à un délai spécifié en millisecondes. Ajoutez la ligne suivante à la fin de votre programme :

```
setInterval(() => process.stdout.write("**** \n"), 1).unref();
```

Observez que la chaîne continue d'être imprimée toutes les millisecondes, même entre la lecture et la réécriture du fichier. Cela montre que la lecture et l'écriture du fichier ont été implémentées de manière non bloquante car les opérations se terminent encore pendant que le fichier est traité.

```
(base) joseph@joseph-HP-EliteBook:~/Desktop/NodeJs/projects/course-2021/session1/tp1$ node readWriteSync.js
****
****
Contenu du fichier: bonjour tout le monde !

****
Fichier mis à jour.
```

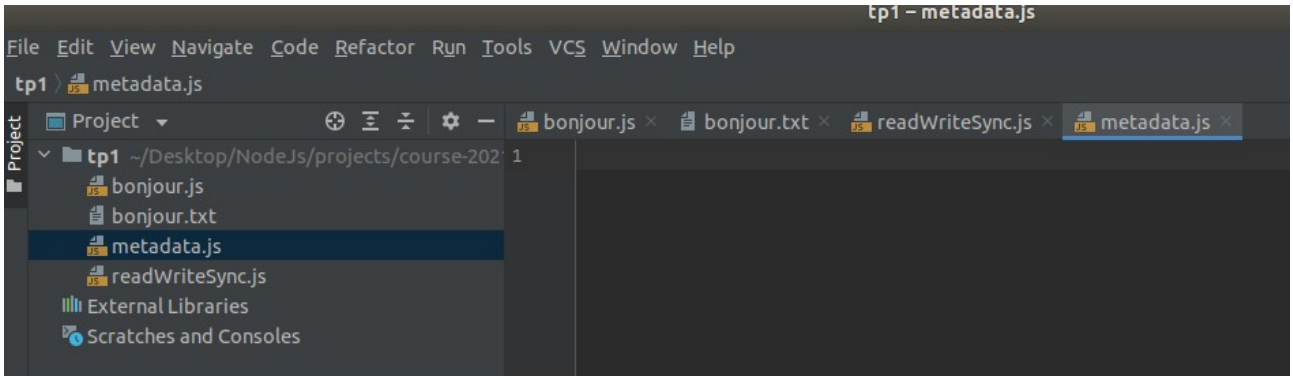
Inspection des métadonnées du fichier

Le module **fs** fournit généralement des API modélisées autour des fonctions POSIX (Portable Operating System Interface). Le module **fs** comprend des API qui facilitent la lecture des répertoires et des métadonnées de fichiers.

Dans cet exemple, nous allons créer un petit programme qui renvoie des informations sur un fichier, en utilisant les fonctions fournies par le module **fs**.

Créez un nouveau fichier JS:

$ touch metadata.js



Comme dans les exemples de gestion d'E/S précédents, nous devons d'abord importer les modules de base nécessaires. Pour cet exemple, il suffit d'importer le module **fs**:

const fs = require("fs");

Ensuite, nous avons besoin que le programme puisse lire le nom de fichier en tant qu'argument de ligne de commande. Pour lire l'argument du fichier, nous pouvons utiliser process.argv[2]. Ajoutez la ligne suivante à votre programme :

const fichier = process.argv[2];

Maintenant, nous allons créer notre fonction **afficherMetadata()** :

```
function afficherMetadata(fichier) {
const fileStats = fs.statSync(fichier);
console.log(fileStats);
}
```

Ajoutez un appel à la fonction **afficherMetadata()** :

```
afficherMetadata(fichier);
```

Vous pouvez maintenant exécuter le programme en lui passant le paramètre**./bonjour.txt**. Exécutez votre programme:

```
$ node metadata.js ./bonjour.txt
```

```
(base) joseph@joseph-HP-EliteBook:~/Desktop/NodeJs/projects/course-2021/session1/tp1$ node metadata.js ./bonjour.txt
Stats {
  dev: 66306,
  mode: 33204,
  nlink: 1,
  uid: 1000,
  gid: 1000,
  rdev: 0,
  blksize: 4096,
  ino: 8136397,
  size: 24,
  blocks: 8,
  atimeMs: 1630891006837.1204,
  mtimeMs: 1630891004621.1282,
  ctimeMs: 1630891004621.1282,
  birthtimeMs: 1630885687202.0452,
  atime: 2021-09-06T01:16:46.837Z,
  mtime: 2021-09-06T01:16:44.621Z,
  ctime: 2021-09-06T01:16:44.621Z,
  birthtime: 2021-09-05T23:48:07.202Z
}
```

Vous pouvez essayer d'ajouter du texte aléatoire à *bonjour.txt* et réexécuter votre programme; observez que les valeurs **size** et **mtime** ont été mises à jour.

Voyons maintenant ce qui se passe lorsque nous passons un fichier inexistant au programme:

```
(base) joseph@joseph-HP-EliteBook:~/Desktop/NodeJs/projects/course-2021/session1/tp1$ node metadata.js ./test.txt
internal/fs/utils.js:314
    throw err;
    ^

Error: ENOENT: no such file or directory, stat './test.txt'
    at Object.statSync (fs.js:1127:3)
    at afficherMetadata (/home/joseph/Desktop/NodeJs/projects/course-2021/session1/tp1/metadata.js:4:26)
    at Object.<anonymous> (/home/joseph/Desktop/NodeJs/projects/course-2021/session1/tp1/metadata.js:8:1)
    at Module._compile (internal/modules/cjs/loader.js:1072:14)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1101:10)
    at Module.load (internal/modules/cjs/loader.js:937:32)
    at Function.Module._load (internal/modules/cjs/loader.js:778:12)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:76:12)
    at internal/main/run_main_module.js:17:47 {
  errno: -2,
  syscall: 'stat',
  code: 'ENOENT',
  path: './test.txt'
}
```

Le programme affiche une exception. Nous devrions intercepter cette exception et envoyer un message à l'utilisateur indiquant que le chemin d'accès au fichier fourni n'existe pas. Pour ce faire, remplacez la fonction **afficherMetadata()** par ceci:

```
function afficherMetadata(fichier) {
  try {
    const fileStats = fs.statSync(fichier);
    console.log(fileStats);
  } catch (err) {
    console.error("Erreur de lecture du chemin du fichier:", fichier);
  }
}
```

Exécutez à nouveau le programme avec un fichier inexistant; cette fois, vous devriez voir que le programme a géré l'erreur plutôt que de lever une exception.

process.argv est une propriété sur l'objet de processus global qui renvoie un tableau contenant les arguments qui ont été transmis au processus Node.js. Le premier élément du tableau process.argv, process.argv[0] est le chemin du nœud binaire en cours d'exécution. Le deuxième élément est le chemin du fichier que nous exécutons, dans ce cas, *metadata.js*. Dans cet exemple, nous avons passé le nom de fichier comme troisième argument de la ligne de commande et l'avons donc référencé avec **process.argv[2]**.

Ensuite, nous avons créé une fonction **afficherMetadata()** qui a appelé **statSync(fichier)**. **statSync()** est une fonction synchrone qui renvoie des informations sur le chemin du fichier qui lui est transmis. Le chemin d'accès au fichier transmis peut être soit un fichier, soit un répertoire. Les informations renvoyées se présentent sous la forme d'un objet de statistiques (**stats**).

Le module **fs** fournit des API qui peuvent être utilisées pour modifier les autorisations sur un fichier donné. Comme pour la plupart des autres fonctions **fs**, il existe à la fois une API asynchrone, **chmod()**, et une API synchrone équivalente, **chmodSync()**. Les deux fonctions prennent un chemin de fichier et un mode comme premier et deuxième arguments, respectivement. La fonction asynchrone accepte un troisième paramètre, qui est la fonction de **Callback** à exécuter à la fin.

Nous allons changer les permissions en utilisant l'équivalent de chmod 666 depuis la ligne de commande, mais via Node.js:

```
fs.chmodSync(fichier, 0o666);
```

metadata.js

```js
const fs = require("fs");
const fichier = process.argv[2];
function afficherMetadata(fichier) {
    try {
        const fileStats = fs.statSync(fichier);
        console.log(fileStats);
    } catch (err) {
        console.error("Erreur de lecture du chemin du fichier:", fichier);
    }
}

afficherMetadata(fichier);
fs.chmodSync(fichier, 0o666);
```

```
(base) joseph@joseph-HP-EliteBook:~/Desktop/NodeJs/projects/course-2021/session1/tp1$ ls -l bonjour.txt
-rw-rw-r-- 1 joseph joseph 24 sept.  6 03:16 bonjour.txt
(base) joseph@joseph-HP-EliteBook:~/Desktop/NodeJs/projects/course-2021/session1/tp1$ node metadata.js ./bonjour.txt
Stats {
  dev: 66306,
  mode: 33204,
  nlink: 1,
  uid: 1000,
  gid: 1000,
  rdev: 0,
  blksize: 4096,
  ino: 8136397,
  size: 24,
  blocks: 8,
  atimeMs: 1630891006837.1204,
  mtimeMs: 1630891004621.1282,
  ctimeMs: 1630892773572.616,
  birthtimeMs: 1630885687202.0452,
  atime: 2021-09-06T01:16:46.837Z,
  mtime: 2021-09-06T01:16:44.621Z,
  ctime: 2021-09-06T01:46:13.573Z,
  birthtime: 2021-09-05T23:48:07.202Z
}
(base) joseph@joseph-HP-EliteBook:~/Desktop/NodeJs/projects/course-2021/session1/tp1$ ls -l bonjour.txt
-rw-rw-rw- 1 joseph joseph 24 sept.  6 03:16 bonjour.txt
```

Surveiller les mises à jour des fichiers

Le module **fs** de Node.js fournit une fonctionnalité qui vous permet de regarder des fichiers et de suivre quand des fichiers ou des répertoires sont créés, mis à jour ou supprimés.

Dans cet exemple, nous allons créer un petit programme nommé ***watch.js*** qui surveille les modifications dans un fichier à l'aide de l'API **watchFile()**, puis imprime un message lorsqu'une modification s'est produite.

Créez le fichier **watch.js** :

$ touch watch.js

Importez les modules Node.js de base requis et appelez la fonction **fs.watchFile()**:

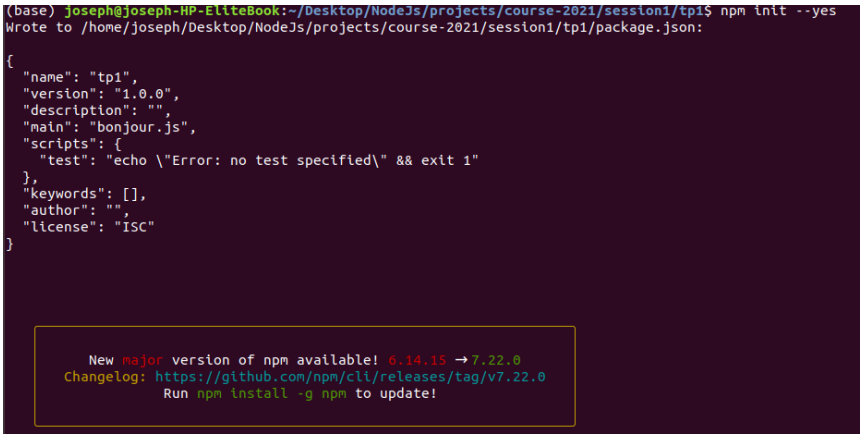
```
const fs = require("fs");
const fichier = "./bonjour.txt";
fs.watchFile(fichier, (current, previous) => {
        return console.log(`${fichier} mis à jour ${(current.mtime)}`);
});
```

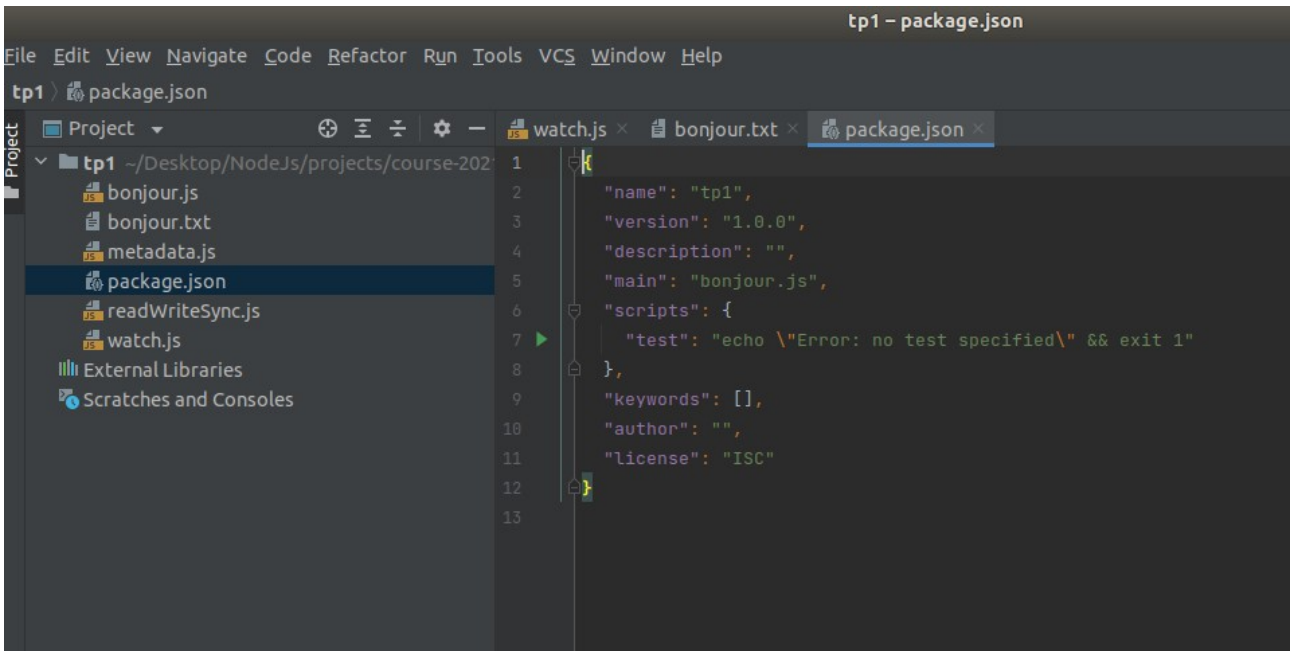
Dans votre éditeur, ouvrez *bonjour.txt* et apportez quelques modifications, en sauvegardant entre chacune. Vous remarquerez que chaque fois que vous enregistrez, une entrée de log apparaît dans le terminal où vous exécutez ***watch.js***:

```
(base) joseph@joseph-HP-EliteBook:~/Desktop/NodeJs/projects/course-2021/session1/tp1$ node watch.js
./bonjour.txt mis à jour Mon Sep 06 2021 03:57:39 GMT+0200 (Central European Summer Time)
./bonjour.txt mis à jour Mon Sep 06 2021 03:57:47 GMT+0200 (Central European Summer Time)
./bonjour.txt mis à jour Mon Sep 06 2021 03:57:54 GMT+0200 (Central European Summer Time)
^C
```

Pendant que nous sommes ici, nous pouvons rendre l'horodatage plus lisible. Pour ce faire, nous allons utiliser le module **moment.js.** C'est un module externe qui vous permet de manipuler les dates et les heures en JavaScript.

Tout d'abord, nous devons initialiser un nouveau projet. Pour ce faire, tapez **$ npm init --yes**. Pour l'instant, nous allons passer l'option **--yes** pour accepter les valeurs par défaut. Vous devriez maintenant avoir un fichier **package.json** dans votre répertoire de projet.





Nous pouvons maintenant installer le module **moment.js**. Notez que cette étape nécessitera une connexion Internet, car le package sera téléchargé à partir du registre public **npm**(npm est un gestionnaire de packages pour le langage de programmation JavaScript):

$ npm install moment

```
(base) joseph@joseph-HP-EliteBook:~/Desktop/NodeJs/projects/course-2021/session1/tp1$ npm install moment
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN tp1@1.0.0 No description
npm WARN tp1@1.0.0 No repository field.

+ moment@2.29.1
added 1 package from 6 contributors and audited 1 package in 0.683s
found 0 vulnerabilities
```

Si vous ouvrez **package.json**, vous remarquerez que **moment** a été ajouté sous le champ des dépendances.

Nous devons maintenant importer **moment** dans notre fichier **watch.js**. Ajoutez ce qui suit, juste en dessous de votre déclaration de constante de fichier:

const moment = require("moment");

Ajoutez et modifiez les lignes suivantes pour formater la date à l'aide de moment.js:

const temps = moment().format("MMMM Do YYYY, h:mm:ss a");

return console.log(`${fichier} mis à jour ${temps}`);

Réexécutez le programme et apportez d'autres modifications à *bonjour.txt*—observez que l'heure est maintenant dans un format plus lisible:

```
(base) joseph@joseph-HP-EliteBook:~/Desktop/NodeJs/projects/course-2021/session1/tp1$ node watch.js
./bonjour.txt mis à jour September 6th 2021, 4:12:01 am
./bonjour.txt mis à jour September 6th 2021, 4:12:06 am
./bonjour.txt mis à jour September 6th 2021, 4:12:11 am
```

Création de la communication TCP serveur et client

Les sockets permettent aux machines et aux appareils de communiquer. Les sockets sont également utilisés pour coordonner les E/S sur les réseaux. Le terme socket est utilisé pour désigner un point d'extrémité d'une liaison de communication réseau bidirectionnelle. Les sockets nous permettent de créer des applications Web en temps réel, telles que des applications de messagerie instantanée.

Dans cet exemple, nous allons créer un serveur TCP et un client TCP et leur permettre de communiquer. TCP signifie Transmission Control Protocol. TCP fournit une norme qui permet aux appareils de communiquer sur un réseau.

Vous devez créer deux fichiers JS distincts, un pour le serveur et un pour le client:

```
$ touch server.js
```

```
$ touch client.js
```

Tout d'abord, nous allons créer un serveur TCP à l'aide du module de base **net**. Nous devons importer le module **net** dans **server.js**:

```
const net = require("net");
```

Maintenant, configurons quelques variables pour stocker le nom d'hôte et le port sur lesquels nous voulons que notre serveur s'exécute:

```
const HOSTNAME = "localhost";
```

```
const PORT = 3000;
```

Maintenant, nous pouvons créer le serveur, en passant les variables **HOSTNAME** et **PORT** dans la fonction **listen()**:

```
net.createServer((socket) => {
   console.log("Client connecté.");
 }).listen(PORT, HOSTNAME);
```

Maintenant, nous devons ajouter des écouteurs d'événements de socket (event listeners). Ajoutez les deux écouteurs d'événements suivants sous *console.log("Client connecté.");* ligne:

```
socket.on("data", (nom) => {
      socket.write(`Bonjour ${nom}!`);
});
```

Créons maintenant le client. Encore une fois, nous devons commencer par importer le module **net** dans **client.js**:

```
const net = require("net");
```

Ensuite, nous pouvons essayer de nous connecter au serveur que nous avons configuré dans **server.js**. Nous allons redéfinir les variables **HOSTNAME** et **PORT** dans ce fichier:

```
const HOSTNAME = "localhost";
const PORT = 3000;
const socket = net.connect(PORT, HOSTNAME);
```

Maintenant que nous sommes connectés à ce socket, nous pouvons y écrire:

```
socket.write("CouCou!!!");
```

Nous devons également ajouter une fonction qui écoutera les données renvoyées par la socket:

```
socket.on("data", (data) => {
        console.log(data.toString());
});
```

Exécutez votre serveur avec la commande suivante:

```
$ node server.js
```

Dans un second shell, exécutez client.js:

```
$ node client.js
```

```
(base) joseph@joseph-HP-EliteBook:~/Desktop/NodeJs/projects/course-2021/session1/tp1$ node server.js
Client connecté.
```

```
(base) joseph@joseph-HP-EliteBook:~/Desktop/NodeJs/projects/course-2021/session1/tp1$ node client.js
Bonjour CouCou!!!!
```

L'exemple a utilisé la fonction **createServer()** et la fonction **net** pour créer le serveur. La fonction passée à **createServer()** accepte une fonction, et cette fonction est exécutée à chaque fois qu'une nouvelle connexion est établie avec le serveur. La documentation de l'API Node.js décrit cette fonction comme une fonction **connectionListener**.

socket est passé en argument à cette fonction d'écoute de connexion. Il est possible d'écouter les événements sur l'objet socket. Dans l'exemple, nous avons écouté l'événement de données et enregistré une fonction à exécuter à chaque réception de données.

Nous avons ensuite appelé la fonction **listen()** sur **createServer()**—cette fonction démarre le serveur à l'écoute des connexions. Dans l'exemple, nous avons passé à la fonction **listen()** le nom

d'hôte et le port auxquels nous voulions que le serveur soit accessible. A noter qu'il est également possible d'écouter sur un socket Unix tel que:

```
const socket = net.connect("/tmp/my.socket");
```

De même, nous avons également enregistré un écouteur de données dans **client.js**, qui écoutait les données réécrites depuis le serveur.

Flux, Flux, Flux

Les flux sont l'une des fonctionnalités clés de Node.js. La plupart des applications Node.js reposent sur l'implémentation des flux Node.js sous-jacents, que ce soit pour la lecture/écriture de fichiers, la gestion des requêtes HTTP ou d'autres communications réseau. Les flux fournissent un mécanisme pour lire séquentiellement l'entrée et écrire la sortie.

En lisant des morceaux de données de manière séquentielle, nous pouvons travailler avec de très gros fichiers (ou d'autres données d'entrée) qui seraient généralement trop volumineux pour être lus en mémoire et traités dans leur ensemble. Les flux sont fondamentaux pour les applications Big Data ou les services de streaming multimédia, où les données sont trop volumineuses pour être consommées en même temps.

Il existe quatre principaux types de flux dans Node.js:

- **Flux lisibles (Readable streams)**: utilisés pour lire les données
- **Flux d'écritures (Writable streams):** utilisés pour écrire des données
- **Flux duplex (Duplex streams)**: utilisés à la fois pour la lecture et l'écriture de données
- **Flux de transformation (Transform streams)**: un type de flux duplex qui transforme l'entrée de données, puis génère les données transformées

Créer des flux dans Node.js

L'API de flux est fournie par le module de base **stream**. Cet exemple fournira une introduction à l'utilisation de flux dans Node.js. Il expliquera comment créer à la fois un flux lisible et un flux accessible en écriture pour interagir avec les fichiers, à l'aide du module **fs**.

Créez les deux fichiers suivants:

```
$ touch write-stream.js
```

```
$ touch read-stream.js
```

Nous allons d'abord créer un flux accessible en écriture pour écrire un gros fichier. Nous lirons ensuite ce gros fichier à l'aide d'un flux lisible. Commencez par importer le module principal du système de fichiers Node.js dans ***write-stream.js:***

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

Ensuite, nous allons créer le flux d'écriture à l'aide de la méthode **createWriteStream()** disponible sur le module **fs**:

```
const fichier = fs.createWriteStream("./fichier.txt");
```

Maintenant, nous allons commencer à écrire du contenu dans notre fichier. Écrivons plusieurs fois une chaîne aléatoire dans le fichier:

```
for (let i = 0; i <= 1000000; i++) {
  fichier.write(
    "Node.js est un environnement d’exécution JavaScript construit sur le moteur JavaScript V8 de Chrome.\n"
  );
}
```

Now, we can run the script with the following command:

```
$ node write-stream.js
```

Cela aura créé un fichier nommé ***fichier.txt*** dans votre répertoire actuel. Le fichier aura une taille d'environ 75Mo. Pour vérifier que le fichier existe, saisissez la commande suivante dans votre Terminal :

```
$ ls -lh fichier.txt
```

```
(base) joseph@joseph-HP-EliteBook:~/Desktop/NodeJs/projects/course-2021/session1/tp1$ node write-stream.js
(base) joseph@joseph-HP-EliteBook:~/Desktop/NodeJs/projects/course-2021/session1/tp1$ ls -lh fichier.txt
-rw-rw-r-- 1 joseph joseph 100M sept.  6 11:12 fichier.txt
```

Créons maintenant un script qui créera un flux lisible pour lire le contenu du fichier. Démarrez le fichier **read-stream.js** en important le module principal **fs**:

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

Maintenant, nous pouvons créer notre flux lisible en utilisant la méthode **createReadStream()**:

```
const rs = fs.createReadStream("./fichier.txt");
```

Ensuite, nous pouvons enregistrer un gestionnaire d'événements de données, qui s'exécutera à chaque fois qu'un bloc de données sera lu:

```
rs.on("data", (data) => {
        console.log("Lire le morceau:", data);
});
```

Nous ajouterons également un gestionnaire d'événement de fin, qui sera déclenché lorsqu'il ne restera plus de données à consommer dans le flux:

```
rs.on("end", () => {
        console.log("Plus de données.");
});
```

Exécutez le programme avec la commande suivante et attendez-vous à ce que les blocs de données soient enregistrés au fur et à mesure de leur lecture:

```
$ node read-stream.js
```

Si nous appelons **toString()** sur les morceaux de données individuels dans la fonction de gestionnaire d'événements de données, nous verrons la sortie du contenu de la chaîne au fur et à mesure de son traitement. Modifiez la fonction du gestionnaire d'événements de données comme suit:

```
rs.on("data", (data) => {
        console.log("Lire le morceau:", data.toString());
});
```

Réexécutez le script.

Interagir avec des données infinies

Les flux permettent d'interagir avec des quantités infinies de données. Écrivons un script qui traitera les données de manière séquentielle, indéfiniment. créez un fichier nommé **infinite-read.js**:

```
$ touch infinite-read.js
```

Nous avons besoin d'une source de données infinie. Nous utiliserons le fichier **/dev/urandom**, qui est disponible sur les systèmes d'exploitation de type Unix. Ce fichier est un générateur de nombres pseudo-aléatoires. Ajoutez ce qui suit à **infinite-read.js** pour calculer la taille continue de **/dev/urandom**:

```
const fs = require("fs");
const rs = fs.createReadStream("/dev/urandom");
let taille = 0;
rs.on("data", (data) => {
        taille += data.length;
        console.log("Taille du fichier:", taille);
});
```

Exécutez le script avec la commande suivante:

```
$ node infinite-read.js
```

Attendez-vous à voir une sortie similaire à la suivante, montrant la taille toujours croissante du fichier **/dev/urandom:**

Cet exemple montre comment nous pouvons utiliser des flux pour traiter des quantités infinies de données.

Flux lisibles avec des itérateurs asynchrones

Les flux lisibles sont des itérables asynchrones. Cela signifie que nous pouvons utiliser la syntaxe “for await...of” pour parcourir les données du flux. Créez un fichier nommé ***for-await-read-stream.js:***

$ touch for-await-read-stream.js

Pour implémenter la logique **read-stream.js** de l'exemple précédent à l'aide d'itérables asynchrones, utilisez le code suivant:

```
const fs = require("fs");
const rs = fs.createReadStream("./fichier.txt");
async function run() {
  for await (const chunk of rs) {
    console.log("Lire le morceau:", chunk);
  }
  console.log("Plus de données.");
}
run();
```

Pour plus d'informations sur la syntaxe **for await...of,** reportez-vous à la documentation Web MDN: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for-await...of

Transformer des données avec “transform streams”

Les flux de transformation nous permettent de consommer des données d'entrée, puis de traiter ces données, puis de sortir les données sous forme traitée. Nous pouvons utiliser des flux de transformation pour gérer la manipulation des données de manière fonctionnelle et asynchrone. Il est possible de regrouper de nombreux flux de transformation, ce qui nous permet de décomposer le traitement complexe en tâches séquentielles.

Créez un fichier nommé **transform-stream.js:**

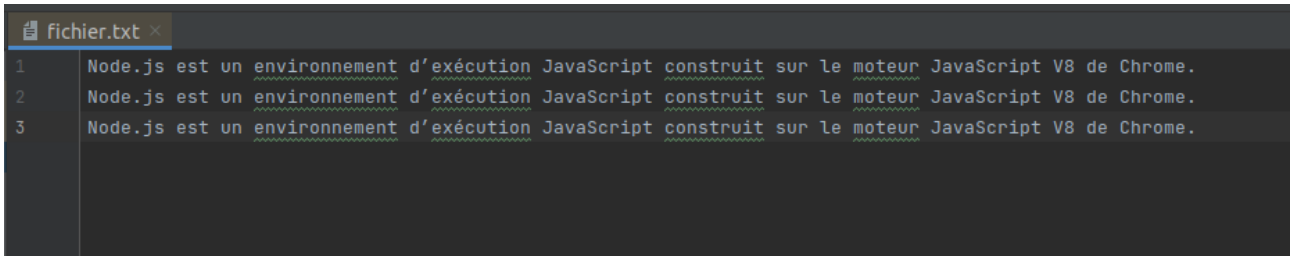
```
$ touch transform-stream.js
```

Nous aurons également besoin de quelques exemples de données à transformer. Créez un fichier nommé **fichier.txt:**

```
$ touch fichier.txt
```

Ajoutez des données de texte au fichier **fichier.txt**, telles que les suivantes:

```
Node.js est un environnement d’exécution JavaScript construit sur le moteur JavaScript V8 de Chrome.
Node.js est un environnement d’exécution JavaScript construit sur le moteur JavaScript V8 de Chrome.
Node.js est un environnement d’exécution JavaScript construit sur le moteur JavaScript V8 de Chrome.
```



Nous devons importer la classe **Transform** à partir du module “**Stream**”et créer un flux lisible pour lire le fichier **fichier.txt:**

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

const { Transform } = require("stream");

const rs = fs.createReadStream("./fichier.txt");
```

Une fois que le contenu de notre fichier a été traité par notre “Stream Transform” nous l'écrirons dans un nouveau fichier nommé **newFichier.txt.** Créez un flux accessible en écriture pour écrire ce fichier à l'aide de la méthode **createWriteStream():**

```
const newFichier = fs.createWriteStream("./newFichier.txt");
```

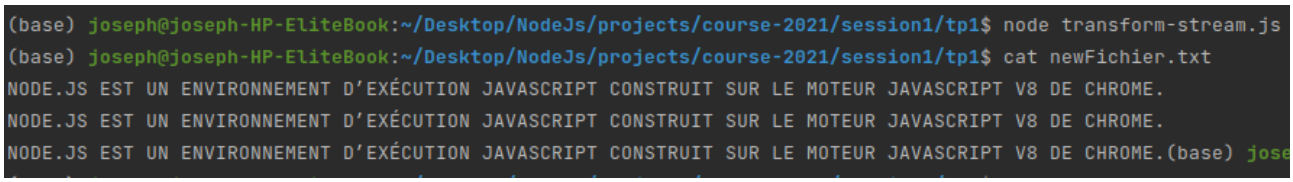
Ensuite, nous devons commencer à définir notre flux de transformation. Nous nommerons notre flux de transformation **enMajuscule()**:

```
const enMajuscule = new Transform({
  transform(chunk, encoding, callback) {
    callback(null, chunk.toString().toUpperCase());
  },
});
```

Cela appelle la fonction de rappel (callback) du flux de transformation avec le morceau transformé. Cette logique transforme le morceau en une chaîne en majuscule.

Nous devons maintenant enchaîner tous nos flux ensemble. Nous allons le faire en utilisant la méthode **pipe()**. Ajoutez la ligne suivante au bas du fichier **transform-stream.js:**

```
rs.pipe(enMajuscule).pipe(newFichier);
```



Les flux de transformation sont des flux duplex, ce qui signifie qu'ils implémentent à la fois des interfaces de flux lisibles et inscriptibles. Les flux de transformation sont utilisés pour traiter (ou transformer) l'entrée, puis la transmettre en sortie.

Pour créer un flux de transformation, nous importons la classe Transform à partir du module "**stream**" . Le constructeur "transform stream" accepte les deux arguments suivants:

- **transform** : La fonction qui implémente la logique de traitement/transformation des données
- **flush** : si le processus de transformation émet des données supplémentaires, la méthode flush est utilisée pour vider les données. Cet argument est facultatif

C'est la fonction **transform()** qui traite l'entrée du flux et produit la sortie. Notez qu'il n'est pas nécessaire que le nombre de morceaux (**chuncks**) fournis via le flux d'entrée soit égal au nombre sorti par le flux de transformation - certains morceaux pourraient être omis pendant la transformation/le traitement.

Création de flux de transformation en mode objet

Par défaut, les flux Node.js fonctionnent sur des objets **String**, **Buffer** ou **Uint8Array**. Cependant, il est également possible de travailler avec des flux Node.js en mode **objet**. Cela nous permet de travailler avec d'autres valeurs JavaScript (sauf la valeur nulle). En mode objet, les valeurs renvoyées par le flux sont des objets JavaScript génériques. Pour définir un flux en mode objet, nous passons { **objectMode: true** } via l'objet **options**. Montrons comment créer un flux de transformation en mode objet. Créer un fichier nommé **object-stream.js:**

```
$ touch object-stream.js
```

Installez le module **ndjson**:

$ npm install ndjson

```
(base) joseph@joseph-HP-EliteBook:~/Desktop/NodeJs/projects/course-2021/session1/tp1$  npm install ndjson
npm WARN tp1@1.0.0 No description
npm WARN tp1@1.0.0 No repository field.

+ ndjson@2.0.0
added 10 packages from 8 contributors and audited 11 packages in 1.287s

1 package is looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Dans **object-stream.js**, importez la classe **Transform** du module "**stream**" et la méthode **stringify()** du module **ndjson**:

```
const { Transform } = require("stream");

const { stringify } = require("ndjson");
```

Créez le flux de transformation en spécifiant { **objectMode: true** }:

```
const nom = Transform({
  objectMode: true,
  transform: ({ prenom, nom }, encoding, callback) => {
    callback(null, { nom: prenom + " " + nom });
  },
});
```

Maintenant, nous pouvons créer notre chaîne de flux. Nous allons rediriger le flux de transformation "**nom**" vers la méthode **stringify()** (depuis **ndjson**), puis rediriger le résultat vers **process.stdout**:

```
nom.pipe(stringify()).pipe(process.stdout);
```

Enfin, toujours dans **object-stream.js**, nous écrirons des données dans le flux de transformation "**nom"** en utilisant la méthode **write()**:

```
nom.write({ prenom: "Clara", nom: "Clara" });

nom.write({ prenom: "Ronald", nom: "Ronald" });
```

Dans cet exemple, nous avons créé un flux de transformation appelé "**nom**" qui agrège la valeur de deux propriétés JSON (prénom et nom) et renvoie une nouvelle propriété (nom) avec la valeur agrégée. Le flux de transformation "**nom**" est en mode objet et lit et écrit des objets.

Nous dirigeons notre flux de transformation "**nom**" vers la fonction **stringify()** fournie par le module **ndjson**. La fonction **stringify()** convertit les objets JSON diffusés en JSON délimité par une nouvelle ligne. Le flux **stringify()** est un flux de transformation où le côté inscriptible est en mode objet, mais le côté lisible ne l'est pas.

object-stream.js

```
const { Transform } = require("stream");
const { stringify } = require("ndjson");

const nom = Transform({
    objectMode: true,
    transform: ({ prenom, nom }, encoding, callback) => {
        callback(null, { nom: prenom + " " + nom });
    },
});

nom.pipe(stringify()).pipe(process.stdout);

nom.write({ prenom: "Clara", nom: "Clara" });
nom.write({ prenom: "Ronald", nom: "Ronald" });
```

```
(base) joseph@joseph-HP-EliteBook:~/Desktop/NodeJs/projects/course-2021/session1/tp1$ node object-stream.js
{"nom":"Clara Clara"}
{"nom":"Ronald Ronald"}
```

Construire des pipelines de flux

Le module de flux "**stream**" fournit une méthode **pipeline()**. De la même manière que nous pouvons utiliser la méthode **pipe()** pour diriger un flux vers un autre, nous pouvons également utiliser la méthode **pipeline()** pour chaîner plusieurs flux ensemble.

Contrairement à la méthode **pipe()**, la méthode **pipeline()** transmet également les erreurs, ce qui facilite la gestion des erreurs dans le flux de flux.

Nous allons créer un pipeline de flux en utilisant la méthode **pipeline()**. Créez un fichier nommé **pipeline.js**:

$ touch pipeline.js

Utilisez le même fichier ***fichier.txt*** que nous avons créé auparavant:

fichier.txt

```
Node.js est un environnement d'exécution JavaScript construit sur le moteur JavaScript V8 de Chrome.
Node.js est un environnement d'exécution JavaScript construit sur le moteur JavaScript V8 de Chrome.
Node.js est un environnement d'exécution JavaScript construit sur le moteur JavaScript V8 de Chrome.
```

Notre pipeline lira le fichier **fichier.txt,** convertira le contenu du fichier en majuscules à l'aide d'un flux de transformation, puis écrira le nouveau contenu du fichier dans un nouveau fichier:

```
const fs = require("fs");
const { pipeline, Transform } = require("stream");
const enMajuscule = new Transform({
  transform(chunk, encoding, callback) {
    callback(null, chunk.toString().toUpperCase());
  },
});
```

La méthode pipeline s'attend à ce que le premier argument soit un flux lisible. Notre premier argument sera un flux lisible qui lira le fichier ***fichier.txt***, en utilisant la méthode **createReadStream().** Ensuite, nous devons ajouter notre flux de transformation comme deuxième argument à la méthode **pipeline().** Ensuite, nous pouvons ajouter notre flux d'écriture pour écrire le fichier ***newFichier.txt*** dans le pipeline. Enfin, le dernier argument de notre pipeline est une fonction de callback qui s'exécutera une fois le pipeline terminé. Cette fonction de callback gérera toutes les erreurs dans notre pipeline:

```
pipeline(
  fs.createReadStream("./fichier.txt"),
  enMajuscule,
  fs.createWriteStream("./newFichier.txt"),
  (err) => {
    if (err) {
      console.error("Échec du pipeline.", err);
    } else {
      console.log("Pipeline réussi.");
    }
  }
);
```

Dans votre Terminal, lancez le programme avec la commande suivante:

```
$ node pipeline.js
```

```
(base) joseph@joseph-HP-EliteBook:~/Desktop/NodeJs/projects/course-2021/session1/tp1$ node pipeline.js
Pipeline réussi.
(base) joseph@joseph-HP-EliteBook:~/Desktop/NodeJs/projects/course-2021/session1/tp1$ cat newFichier.txt
NODE.JS EST UN ENVIRONNEMENT D'EXÉCUTION JAVASCRIPT CONSTRUIT SUR LE MOTEUR JAVASCRIPT V8 DE CHROME.
NODE.JS EST UN ENVIRONNEMENT D'EXÉCUTION JAVASCRIPT CONSTRUIT SUR LE MOTEUR JAVASCRIPT V8 DE CHROME.
NODE.JS EST UN ENVIRONNEMENT D'EXÉCUTION JAVASCRIPT CONSTRUIT SUR LE MOTEUR JAVASCRIPT V8 DE CHROME.(base)
```

À faire

Ajoutez un fichier **data.json** à votre projet et à l'intérieur de ce fichier ajoutez ce qui suit:


```
[
  {
    "studentID": 1,
    "nom": "Zuckerberg",
    "prenom": "Mark",
    "groupe": "A1",
    "note": 16
  },
  {
    "studentID": 2,
    "nom": "Gates",
    "prenom": "Bill",
    "groupe": "A2",
    "note": 9
  },
  {
    "studentID": 3,
    "nom": "Musk",
    "prenom": "Elon",
    "groupe": "B1",
    "note": 19
  },
  {
    "studentID": 3,
    "nom": "Dorsey",
    "prenom": "Jack",
    "groupe": "B2",
    "note": 12
  }
]
```


Créez un script Node.js qui génère une page html qui affiche le contenu du fichier data.json dans un tableau HTML comme suit:

ID	Nom	Prenom	Groupe	Note
1	Zuckerberg	Mark	A1	16
2	Gates	Bill	A2	9
3	Musk	Elon	B1	19
3	Dorsey	Jack	B2	12

Si vous souhaitez aller plus loin, créez un nouveau script Node.js qui permet à un utilisateur d'ajouter plus d'étudiants au tableau JSON actuel.

À faire

créez un fichier html "error.html" avec le contenu suivant:

```
<!DOCTYPE html>
<html>
<style>
 table, th, td {
   border:1px solid black;
 }
</style>
<body>
<table style="width:100%">
 <tr>
   <th>Company</th>
   <th>Contact</th>
   <th>Country</th>
 </tr>
 <tr>
   <tk>Alfreds Futterkiste</tk>
   <tk>Maria Anders</tk>
   <tk>Germany</tk>
 </tr>
 <tr>
   <tk>Centro comercial Moctezuma</tk>
   <tk>Francisco Chang</tk>
   <tk>Mexico</tk>
 </tr>
</table>
</body>
</html>
```

Vous pouvez remarquer qu'il y a des erreurs dans le tableau HTML (<tk>). Créez un script NodeJS qui crée une copie corrigée de ce fichier. Lors de la création d'une copie dans un autre fichier, disons "corrected.html", assurez-vous que le script NodeJS remplace <tk> et </tk> par <td> et </td>.

À faire

Créez un fichier "files.txt" (**$ touch files.txt**). Dans ce fichier, ajoutez les lignes suivantes:

mydir/doc1.txt

mydir/doc2.csv

mydir/doc3.html

mydir/doc4.xml

Créez deux scripts NodeJS. Le premier passe par toutes les lignes du fichier **files.txt** et les crée de manière **asynchrone**. Le second passe par toutes les lignes et les supprime de manière **asynchrone**.

Reportez-vous à l'API NodeJS pour obtenir de l'aide (https://nodejs.org/api/).