

Codelab: Création d'une Application de Prise de Notes en Ligne de Commande avec Node.js, Yargs et Chalk

Joseph Azar

Introduction

Dans ce codelab, vous allez créer une application de ligne de commande pour gérer des notes en utilisant Node.js. L'application permettra aux utilisateurs d'ajouter, de supprimer, de lister et de lire des notes directement depuis le terminal. Nous utiliserons deux bibliothèques essentielles : Yargs pour l'analyse des arguments de ligne de commande et Chalk pour styliser la sortie du terminal. Ce projet est conçu pour vous initier à Node.js, au travail avec le système de fichiers, et à la création d'une interface en ligne de commande simple.

Prérequis

Avant de commencer, assurez-vous que Node.js est installé sur votre système. Vous pouvez vérifier l'installation en exécutant les commandes suivantes dans votre terminal :

1. Vérifier l'installation de Node.js:

```
...
```

```
node -v
```

```
...
```

Cela devrait retourner la version de Node.js installée.

2. Vérifier l'installation de NPM :

```
...
```

```
npm -v
```

```
...
```

Cela devrait retourner la version de NPM (Node Package Manager) installée.

3. Vérifier l'installation de NVM (Optionnel mais recommandé) :

NVM (Node Version Manager) est un outil qui vous permet de gérer plusieurs versions de Node.js sur votre système. Vérifiez s'il est installé en exécutant :

```
...
```

```
nvm --version
```

...

Si ce n'est pas installé, suivez les instructions d'installation sur le site officiel.

Configuration du Projet

1. Créez un nouveau répertoire pour votre projet et accédez-y :

...

```
mkdir notes-app
```

```
cd notes-app
```

...

2. Initialisez un nouveau projet Node.js :

...

```
npm init
```

...

Il vous sera demandé de remplir certaines informations.

3. Installez les dépendances nécessaires :

...

```
npm install chalk@4 yargs
```

...

Cela installera Chalk pour la colorisation de la sortie et Yargs pour la gestion des arguments en ligne de commande. Vous devriez voir ces bibliothèques sous l'objet *dependencies* dans **package.json**

Explication du Module notes.js

Dans cette section, nous allons créer un module appelé `notes.js` (*touch notes.js*) qui contiendra toutes les fonctions nécessaires pour gérer les notes. Nous utiliserons le module `fs` (système de fichiers) de Node.js pour lire et écrire des fichiers, et `chalk` pour styliser les messages dans le terminal.

4. 1. Importation des modules nécessaires :

...

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

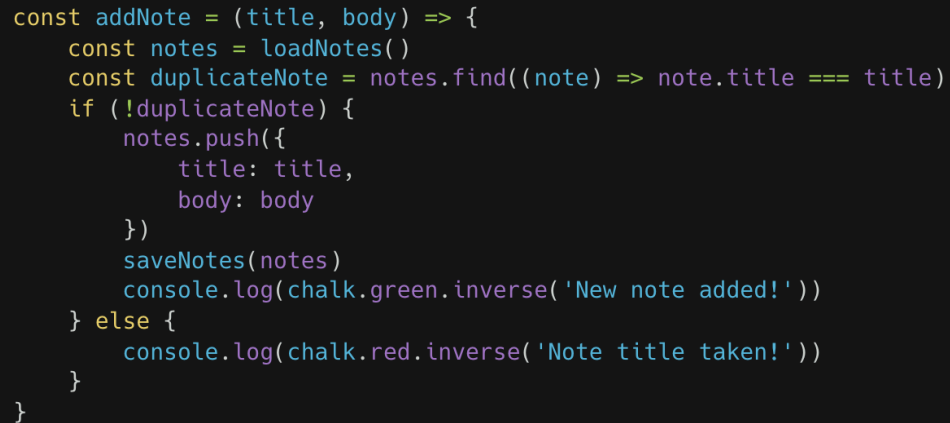
```
const chalk = require('chalk');
```

...

Nous importons ici les modules `fs` et `chalk`. Le module `fs` nous permet de manipuler le système de fichiers, tandis que `chalk` est utilisé pour styliser les messages dans le terminal.

Fonction addNote

La fonction `addNote` permet d'ajouter une nouvelle note. Elle vérifie d'abord si une note avec le même titre existe déjà. Si ce n'est pas le cas, elle ajoute la nouvelle note et la sauvegarde dans un fichier JSON.



```
const addNote = (title, body) => {
  const notes = loadNotes()
  const duplicateNote = notes.find((note) => note.title === title)
  if (!duplicateNote) {
    notes.push({
      title: title,
      body: body
    })
    saveNotes(notes)
    console.log(chalk.green.inverse('New note added!'))
  } else {
    console.log(chalk.red.inverse('Note title taken!'))
  }
}
```

La fonction `addNote` utilise `loadNotes` pour charger les notes existantes, puis `saveNotes` pour les sauvegarder après avoir ajouté une nouvelle note.

Fonction removeNote

La fonction `removeNote` permet de supprimer une note existante par son titre. Elle compare la longueur de la liste avant et après la suppression pour déterminer si la note a été trouvée et supprimée.

```
const removeNote = (title) => {
  const notes = loadNotes()
  const notesToKeep = notes.filter((note) => note.title !== title)

  if (notes.length > notesToKeep.length) {
    console.log(chalk.green.inverse('Note removed!'))
    saveNotes(notesToKeep)
  } else {
    console.log(chalk.red.inverse('No note found!'))
  }
}
```

Fonctions loadNotes, readNote, listNotes, et saveNotes

```
const saveNotes = (notes) => {
  const dataJSON = JSON.stringify(notes)
  fs.writeFileSync('notes.json', dataJSON)
}
```

Objectif : La fonction saveNotes est responsable de sauvegarder un tableau de notes dans un fichier JSON (notes.json).

Étapes :

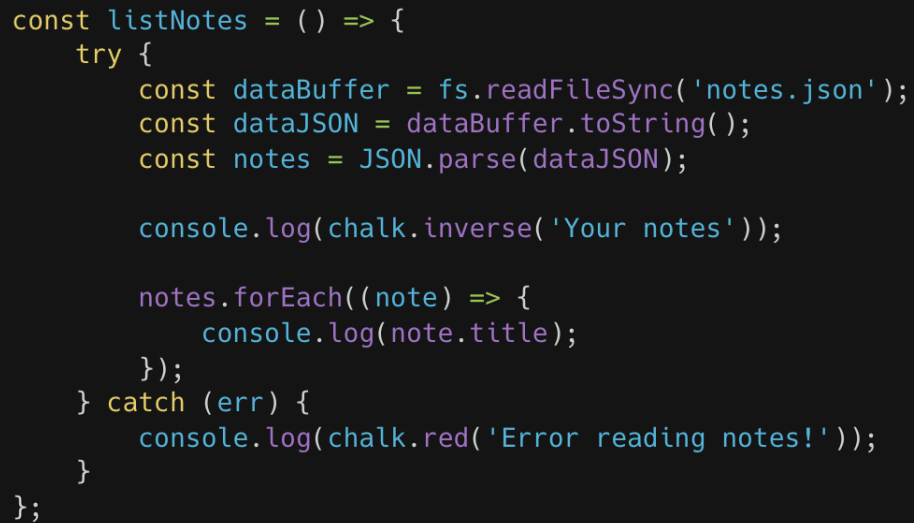
- Conversion en JSON : Le tableau de notes est d'abord converti en une chaîne JSON avec `JSON.stringify(notes)`.
- Écriture dans le fichier : Ensuite, cette chaîne JSON est écrite dans le fichier `notes.json` en utilisant `fs.writeFileSync`. Cette méthode est synchrone et garantit que les données sont correctement écrites avant que le programme ne continue.



Objectif : La fonction loadNotes est utilisée pour charger les notes à partir du fichier notes.json.

Étapes :

- Lecture du fichier : La fonction essaie de lire le fichier notes.json en utilisant fs.readFileSync, qui renvoie un buffer (ensemble de données brut).
- Conversion en chaîne de caractères : Ce buffer est ensuite converti en une chaîne de caractères avec toString().
- Parsing du JSON : La chaîne JSON est convertie en un objet JavaScript avec JSON.parse et est renvoyée par la fonction.
- Gestion des erreurs : Si une erreur survient (par exemple, si le fichier n'existe pas), la fonction attrape cette erreur avec catch et renvoie un tableau vide ([]), ce qui évite que le programme ne plante.



```
const listNotes = () => {
  try {
    const dataBuffer = fs.readFileSync('notes.json');
    const dataJSON = dataBuffer.toString();
    const notes = JSON.parse(dataJSON);


    console.log(chalk.inverse('Your notes'));

    notes.forEach((note) => {
      console.log(note.title);
    });
  } catch (err) {
    console.log(chalk.red('Error reading notes!'));
  }
};
```

Objectif : La fonction listNotes permet de lister toutes les notes stockées dans le fichier notes.json.

Étapes :

- Lecture du fichier : Comme dans loadNotes, la fonction commence par lire le fichier notes.json en utilisant fs.readFileSync et le convertit en chaîne de caractères.
- Parsing du JSON : La chaîne est ensuite convertie en un tableau de notes en utilisant JSON.parse.
- Affichage des titres : Un message "Your notes" est affiché en couleur inversée, puis chaque titre de note est affiché grâce à forEach.
- Gestion des erreurs : Si une erreur survient pendant la lecture du fichier, un message d'erreur "Error reading notes!" est affiché en rouge.



```
const readNote = (title) => {  
  const notes = loadNotes()  
  const note = notes.find((note) => note.title === title)  
  
  if (note) {  
    console.log(chalk.inverse(note.title))  
    console.log(note.body)  
  } else {  
    console.log(chalk.red.inverse('Note not found!'))  
  }  
}
```

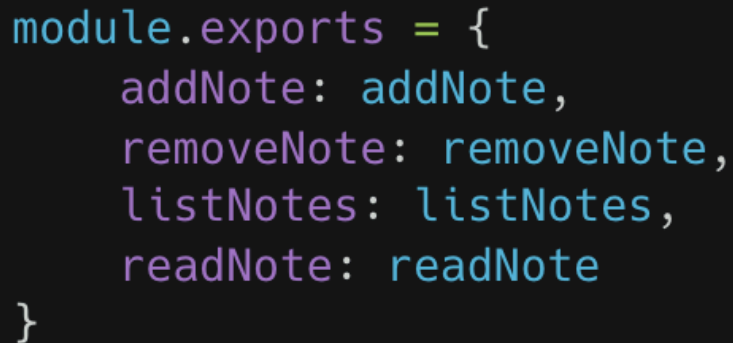
Objectif : La fonction `readNote` est utilisée pour lire le contenu d'une note spécifique en fonction de son titre.

Étapes :

- Charger les notes : La fonction appelle `loadNotes()` pour charger toutes les notes stockées dans le fichier `notes.json`.
- Trouver la note : Elle utilise la méthode `find` sur le tableau de notes pour rechercher celle dont le titre correspond à celui fourni en paramètre (`title`).
- Vérification : Si une note est trouvée (`if (note)`), son titre est affiché en couleur inversée avec `chalk.inverse`, suivi de son contenu. Si aucune note ne correspond, un message d'erreur "Note not found!" est affiché en rouge avec `chalk.red.inverse`.

Partager les fonctionnalités

À la fin du fichier `notes.js`, vous trouverez une section très importante qui ressemble à ceci :



```
module.exports = {  
  addNote: addNote,  
  removeNote: removeNote,  
  listNotes: listNotes,  
  readNote: readNote  
}
```

Objectif :

Cette partie du code est cruciale car elle rend les fonctions définies dans **notes.js** accessibles depuis d'autres fichiers JavaScript de votre projet. En d'autres termes, elle permet à ces fonctions d'être "exportées" et utilisées ailleurs, par exemple dans votre fichier principal **app.js**.

Comment ça fonctionne :

module.exports est un objet spécial en Node.js qui est utilisé pour définir ce qu'un fichier doit exposer lorsqu'il est importé via **require()** dans un autre fichier.

Structure de l'objet :

Le code ci-dessus crée un objet contenant des paires clé-valeur. Chaque clé est le nom sous lequel vous souhaitez que la fonction soit accessible dans les autres fichiers. Chaque valeur est la référence à la fonction définie dans le fichier `notes.js`.

Par exemple, `addNote: addNote` signifie que la fonction `addNote` définie dans `notes.js` sera exportée et pourra être utilisée sous le même nom (`addNote`) lorsqu'elle est importée dans un autre fichier.

Utilisation dans d'autres fichiers :

Lorsque vous souhaitez utiliser ces fonctions dans un autre fichier, comme `app.js`, vous utilisez la syntaxe suivante :

```
const notes = require('./notes.js');
```

Cette ligne importe tout ce qui a été exporté via `module.exports` dans `notes.js`. Vous pouvez ensuite accéder aux fonctions en utilisant `notes.addNote`, `notes.removeNote`, etc.

Importance :

- **Modularité** : Ce mécanisme permet de structurer votre code de manière modulaire. Chaque fichier peut se concentrer sur une partie spécifique de la logique de votre application, ce qui améliore la maintenabilité et la réutilisabilité du code.
- **Réutilisation** : En exposant ces fonctions, vous pouvez les réutiliser dans plusieurs parties de votre application sans avoir à dupliquer le code, ce qui réduit le risque d'erreurs et facilite les mises à jour.

Le fichier principal (`app.js`)

Dans cette section, nous allons expliquer comment utiliser le fichier principal `app.js` pour intégrer nos fonctions et créer une application en ligne de commande.

Commencez par importer les modules `'yargs'` et `'chalk'`, ainsi que les fonctions définies dans `'notes.js'` :

```
...
```

```
const chalk = require("chalk");  
const yargs = require("yargs");  
const notes = require("./notes.js");  
...
```

Ensuite, vous pouvez définir des commandes en utilisant `yargs.command`. Voici un exemple pour créer une commande `add` qui ajoute une nouvelle note :



```
// app.js
const chalk = require("chalk");
const yargs = require("yargs");
const notes = require('./notes.js');

console.log(process.argv);

// Customize yargs version
yargs.version('1.1.0')

// utilisation de YARGS
// Create add command
yargs.command({
  command: 'add',
  describe: 'Add a new note',
  builder: {
    title: {
      describe: 'Note title',
      demandOption: true,
      type: 'string'
    },
    body: {
      describe: 'Note body',
      demandOption: true,
      type: 'string'
    }
  },
  handler(argv) {
    notes.addNote(argv.title, argv.body)
  }
});
```

yargs est une bibliothèque pour Node.js qui simplifie la gestion des arguments en ligne de commande. La structure ci-dessus définit une commande yargs pour ajouter une nouvelle note. Voici une explication détaillée de chaque composant :

1. command: 'add'

Définition : Ce champ spécifie le nom de la commande que l'utilisateur doit taper dans la ligne de commande pour exécuter cette fonction. Ici, 'add' est la commande qui déclenche l'ajout d'une nouvelle note.

Utilisation : Lorsque l'utilisateur tape `node app.js add`, cette commande est activée.

2. describe: 'Add a new note'

Définition : Ce champ fournit une brève description de la commande. Cette description apparaît lorsque l'utilisateur demande de l'aide (par exemple, `node app.js --help`).

Utilisation : Elle aide l'utilisateur à comprendre ce que fait la commande 'add'.

3. builder: { ... }

Définition : La section builder est utilisée pour définir les options que la commande accepte. Chaque option peut avoir ses propres propriétés comme `describe`, `demandOption`, et `type`.

Contenu :

- `title` et `body` : Ce sont les options ou les arguments que l'utilisateur doit fournir avec la commande `add`.
- `describe` : Décrit ce que fait chaque option. Par exemple, `title` représente le titre de la note, et `body` représente le corps ou le contenu de la note.
- `demandOption: true` : Indique que cette option est obligatoire. Si l'utilisateur omet de fournir cette option, yargs affichera une erreur.
- `type: 'string'` : Spécifie le type de données attendu pour cette option. Ici, les deux options `title` et `body` doivent être des chaînes de caractères.

4. handler(argv) { ... }

Définition : La fonction handler est le cœur de la commande. Elle définit ce qui se passe lorsque la commande est exécutée.

Paramètre `argv` : `argv` est un objet qui contient les valeurs des arguments passés dans la ligne de commande. Par exemple, si l'utilisateur tape `node app.js add --title="Titre" --body="Contenu"`, alors `argv.title` sera "Titre" et `argv.body` sera "Contenu".

Utilisation : Dans cet exemple, handler appelle la fonction `addNote` de `notes.js` en passant les arguments `title` et `body` fournis par l'utilisateur.

4. `yargs.parse()`;

À la fin du fichier **app.js**, pour que toutes les configurations prennent effet, vous devez ajouter **`yargs.parse()`**.

```
yargs.parse();
```

À FAIRE : De la même manière, vous pouvez créer des commandes pour ``remove``, ``list``, et ``read``.

Exécution de l'application

Une fois toutes les commandes définies, vous pouvez exécuter l'application en ligne de commande. Voici comment utiliser les commandes :

```
'''
```

```
node app.js add --title="test1" --body="Ceci est une note de test."
```

```
node app.js list
```

```
node app.js read --title="test1"
```

```
node app.js remove --title="test1"
```

```
'''
```

Cela vous permettra d'ajouter, de lister, de supprimer et de lire des notes directement depuis le terminal.

