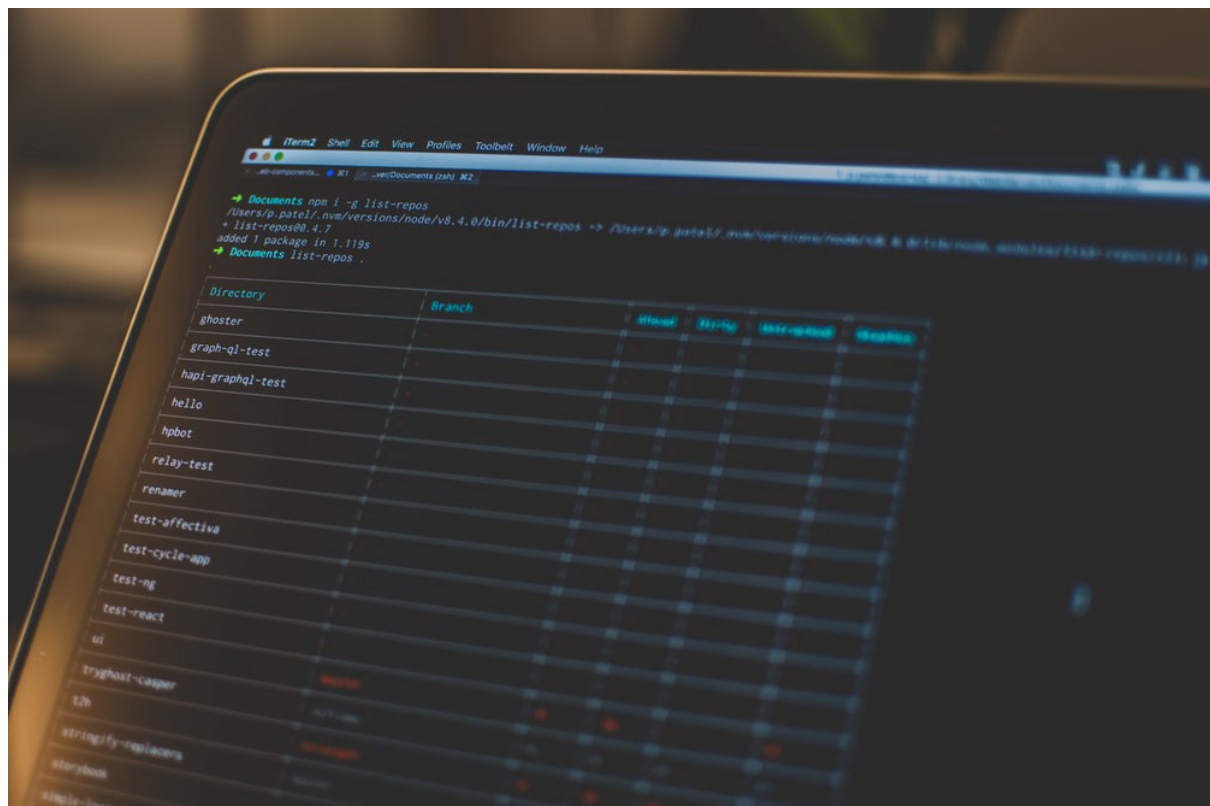


# Criando sua CLI com Node.js



Henrique Kuwai [Follow](#)

Feb 11 · 10 min read



Olá pessoal!

Qualquer desenvolvedor certamente precisa ou precisará ter contato com CLIs (Command Line Interface) durante o seu dia-a-dia. Ferramentas como Git, NPM, Maven, Heroku, Surge, entre outras, todas trazem CLIs para facilitar as tarefas do desenvolvedor; seja para manipular arquivos, como para

trazer/enviar informações, conectar com APIs, entre outros.

Por meio de shell scripts, é fácil realizar tarefas envolvendo arquivos do sistema, no entanto, programar em shell script nem sempre é uma tarefa tão simples assim. Para mim, que estou plenamente acostumado com JavaScript e já conhecendo as capacidades do Node.js, não demorou para vir a pergunta (para mim mesmo):

*Será que é possível criar CLIs com JavaScript?*

Com poucos minutos de pesquisa, a resposta foi um grande sim! Na verdade, o que não dá pra criar com JavaScript hoje em dia, não é mesmo?

Não só é possível criar CLIs com o Node.js como existe uma quantidade enorme de frameworks e bibliotecas para facilitar as tarefas mais comuns quando se fala de interfaces de linha de comando, como input de dados, manipulação dos comandos e flags, exibição das informações de uma maneira mais visual, execução de comandos mais “baixo-nível”, etc.

Para o exemplo que vou trazer aqui, vamos utilizar as seguintes bibliotecas:

- **Commander.js**, que facilita a criação de comandos e manipulação de flags e options (<https://github.com/tj/commander.js/>)

- **Inquirer.js**, um agrupador de inputs no CLI (para inputar dados, checkboxes, etc) (<https://github.com/SBoudrias/Inquirer.js/>)
- **Shelljs**, para executar comandos shell via JavaScript (<https://github.com/shelljs/shelljs>)
- **Chalk**, para facilitar o log de informações coloridas (<https://github.com/chalk/chalk>)
- **Figletjs**, para logar textos em letras garrafais — e pode ser utilizado em conjunto com o chalk! (<https://github.com/patorjk/figlet.js>)
- **CLI Table**, para exibir tabelas no terminal (<https://github.com/Automattic/cli-table>)

Vamos fazer um clássico to-do list, mas como CLI.

. . .

## Criando a base do CLI

Primeiro crie a pasta do seu projeto (ex: `/ todo-cli`) e vamos iniciar o projeto utilizando NodeJS 8+, já instalando os packages que vamos utilizar com o npm ou o yarn (no caso, utilizarei o yarn):

```
> yarn init
> yarn add chalk@2.4.2 commander@2.19.0 figlet@1.2.1
inquirer@6.2.2 shelljs@0.8.3 cli-table@0.3.1
```

Agora, com tudo instalado, vamos criar um `index.js` com o seguinte conteúdo:

```
1  #!/usr/bin/env node
2
3  const program = require('commander');
4  const package = require('./package.json');
5
6  program.version(package.version);
7
8  program
9    .command('add [todo]')
10   .description('Adiciona um to-do')
11   .action((todo) => {
12     console.log(todo);
13   });
14
15  program.parse(process.argv);
```

- Na primeiríssima linha, adicionamos um comentário iniciado com hashtag para que, em sistemas `*nix`, este script seja interpretado utilizando `node`. Em sistemas Windows isso será simplesmente ignorado.
- Depois, nas linhas 3 e 4, damos o `require` no `commander` e depois no `package.json`, apenas com o intuito de pegar a versão do projeto.
- Na linha 6, setamos a versão do CLI, utilizando a função `version` do `Commander.js`.
- A partir da linha 8, colocamos nosso primeiro comando do CLI: o comando de `add`. Primeiro você utiliza o método `command`, passando uma string, e os parâmetros esperados no comando

devem ser encapsulados com `<>` (se forem obrigatórios) e com `[]` se forem opcionais.

- Depois, com o método `description`, setei uma descrição para o comando (isso será útil para a criação automática — pelo `commander` — da flag `--help`). Perceba que utilizei a marcação `[todo]`, e depois esse parâmetro é passado na função de `callback` do método `action`.
- E na última linha, o `commander` requer que passemos o `process.argv`, algo que o Node.js já disponibiliza para nós, para ele poder interpretar os comandos.

Já podemos fazer nosso primeiro teste, executando `node index.js add teste`, e o output será mais ou menos assim:

```
rike_@DESKTOP-4GFJL47 MINGW64 /d/projetos/todo-cli (master)
$ node index.js add teste
teste
```

*Mas, peraí. A ideia era criar uma CLI, onde eu pudesse executar em qualquer lugar da minha máquina! Dessa forma tenho que passar o caminho do script!*

Calma, jovem. Isso você vai aprender exatamente agora:

## Registrando seu script como um comando global

Com o script criado, vamos alterar e adicionar um nó novo no

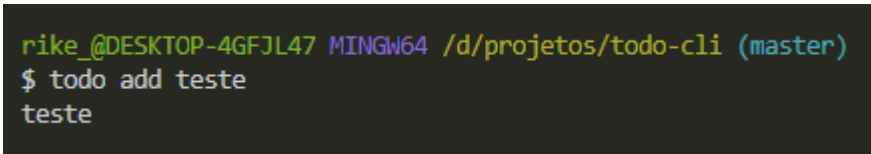
`package.json`:

```
"bin": {  
  "todo": "index.js"  
}
```

- O nó define qual será o comando principal do CLI;
- O arquivo define qual script será executado ao rodar o comando.

Salve o arquivo e, dentro do seu projeto, execute o comando `npm link`. Aguarde um momento e... simples assim. Em alguns sistemas, pode ser necessário o uso de `sudo`, pois ele registra isso de forma global.

Agora você pode utilizar o comando em qualquer local da sua máquina!



```
rike_@DESKTOP-4GFJL47 MINGW64 /d/projetos/todo-cli (master)  
$ todo add teste  
teste
```

E o mais legal disso é que ele mantém um link direto ao seu arquivo, sendo assim, você pode ir desenvolvendo e ir executando o comando para testar, sem precisar de mais nada. :)

## Adicionando, de fato, o to-do

Para minimizar o boilerplate aqui do tutorial, optei por realizar de uma forma que não fosse necessário a integração com uma API. Sendo assim, vamos simplesmente gravar e ler os nossos to-dos em um arquivo `todos.json`.

Vamos fazê-lo com duas possibilidades: caso você passe o argumento `[todo]`, ele já adicionará sem nenhuma pergunta:

```
1  #!/usr/bin/env node
2
3  const program = require('commander');
4  const { join } = require('path');
5  const fs = require('fs');
6
7  const package = require('./package.json');
8  const todosPath = join(__dirname, 'todos.json');
9
10 const getJson = (path) => {
11     const data = fs.existsSync(path) ? fs.readFileSync(path) : [];
12     try {
13         return JSON.parse(data);
14     } catch (e) {
15         return [];
16     }
17 };
18 const saveJson = (path, data) => fs.writeFileSync(path, JSON.stringify(data,
19
20 program.version(package.version);
21
22 program
23     .command('add <todo>')
24     .description('Adiciona um to-do')
25     .action((todo) => {
26         const data = getJson(todosPath);
27         data.push({
28             title: todo,
29             done: false
30         });
31         saveJson(todosPath, data);
32     });
```

- Nas linhas iniciais, agora eu incluo dois módulos do Node.js: path (mais especificamente a função join) e fs, para manipular arquivos do file system;
- Na linha 8, guardei numa variável o path do nosso “banco de dados”: todos.json;



- Nas linhas 10 e 18, criei funções utilitárias para pegar os dados do arquivo e salvar, prevenindo erros caso ele esteja vazio no `getJSON` e parseando como string com separadores no `saveJson` ;
- A partir da linha 26, simplesmente: pego os dados do `todos.json` com a função e guardo numa variável, acrescento o `to-do` passado como objeto, junto ao atributo `done` , e salvo o arquivo novamente.

Mas como o atributo `[todo]` está opcional, precisaremos adicionar um tratamento, e aí vem a segunda possibilidade:

```
1  #!/usr/bin/env node
2
3  const program = require('commander');
4  const { join } = require('path');
5  const fs = require('fs');
6  const inquirer = require('inquirer');
7
8  const package = require('./package.json');
9  const todosPath = join(__dirname, 'todos.json');
10
11 const getJson = (path) => {
12   const data = fs.existsSync(path) ? fs.readFileSync(path) : [];
13   try {
14     return JSON.parse(data);
15   } catch (e) {
16     return [];
17   }
18 };
19 const saveJson = (path, data) => fs.writeFileSync(path, JSON.stringify(data,
20
21 program.version(package.version);
22
23 program
24   .command('add [todo]')
25   .description('Adiciona um to-do')
26   .action(async (todo) => {
27     let answers;
28     if (!todo) {
29       answers = await inquirer.prompt([
30         {
31           type: 'input',
32           name: 'todo',
33           message: 'Qual é o seu to-do?'
34         }
35       ]);
36     }
37
38     const data = getJson(todosPath);
39     data.push({
40       title: todo || answers.todo,
41       done: false
42     });
43     saveJson(todosPath, data);
```

- Na linha 6, incluí o módulo `Inquirer`, para fazer nossas perguntas ao usuário;
- Na linha 26, perceba que adicionei a keyword `async` à função, pois vamos utilizar `await` dentro dela;
- Na linha 27, declaro a variável `answers` ;
- A partir da linha 28, caso o usuário não passe o argumento `todo` , vamos perguntar ao usuário qual o texto do to-do. O `inquirer` possui o método `prompt` , que espera um array de objetos contendo cada “campo” do seu formulário no CLI, tendo como parâmetro obrigatório o `type` do input — veja todos os tipos [aqui] (<https://github.com/SBoudrias/Inquirer.js/#examples>) e o `name` , pois depois será utilizado para pegar os dados. Isto retorna uma promise, então podemos utilizar o `await` por aqui!
- Finalmente, na linha 40, verificamos: o argumento `todo` foi passado? Então utilizamos ele; senão, utilize a resposta do `Inquirer`!

## Tratando os erros

O `Inquirer` permite que você valide os dados inputados, e não deixe o usuário prosseguir enquanto não estiver correto.

Sendo assim, vamos adicionar uma validação para que a pergunta do to-do não passe vazia:

```
1  #!/usr/bin/env node
2
3  const program = require('commander');
4  const { join } = require('path');
5  const fs = require('fs');
6  const inquirer = require('inquirer');
7
8  const package = require('./package.json');
9  const todosPath = join(__dirname, 'todos.json');
10
11 const getJson = (path) => {
12   const data = fs.existsSync(path) ? fs.readFileSync(path) : [];
13   try {
14     return JSON.parse(data);
15   } catch (e) {
16     return [];
17   }
18 };
19 const saveJson = (path, data) => fs.writeFileSync(path, JSON.stringify(data,
20
21 program.version(package.version);
22
23 program
24   .command('add [todo]')
25   .description('Adiciona um to-do')
26   .action(async (todo) => {
27     let answers;
28     if (!todo) {
29       answers = await inquirer.prompt([
30         {
31           type: 'input',
32           name: 'todo',
33           message: 'Qual é o seu to-do?',
34           validate: value => value ? true : 'Não é permitido um to-
35         }
36       ]);
37     }
38
39     const data = getJson(todosPath);
40     data.push({
41       title: todo || answers.todo,
42       done: false
43     });
```

- Na linha 34, adicionei o atributo `validate`, que recebe uma função passando o parâmetro `value`. No caso da minha validação, só quero verificar se o usuário passou qualquer `truthy value`; caso sim, retorna `true`; senão, exibe a mensagem de erro para o usuário e não deixa ele prosseguir.

O Inquirer possui ainda um atributo `filter`, que manipula os dados DEPOIS de inputados. Mas, como qualquer biblioteca na vida, dê uma lidinha na [documentação oficial] (<https://github.com/SBoudrias/Inquirer.js>).

Nosso programa já vai rodar dessa maneira:

```
rike_@DESKTOP-4GFJL47 MINGW64 /d/projetos/todo-cli (master)
$ todo add teste

rike_@DESKTOP-4GFJL47 MINGW64 /d/projetos/todo-cli (master)
$ todo add
? Qual é o seu to-do? Comprar pão
```

A primeira vez, passando o argumento. Na segunda, usando o Inquirer

E nossos dados ficarão assim:

```
[
  {
    "title": "teste",
    "done": false
  },
  {
    "title": "Comprar pão",
    "done": false
  }
]
```

# Adicionando mensagens bonitas de retorno

Utilizando o chalk, podemos adicionar cor aos nossos console.log de maneira muito, muito fácil:

```
1  #!/usr/bin/env node
2
3  const program = require('commander');
4  const { join } = require('path');
5  const fs = require('fs');
6  const inquirer = require('inquirer');
7  const chalk = require('chalk');
8
9  const package = require('./package.json');
10 const todosPath = join(__dirname, 'todos.json');
11
12 const getJson = (path) => {
13   const data = fs.existsSync(path) ? fs.readFileSync(path) : [];
14   try {
15     return JSON.parse(data);
16   } catch (e) {
17     return [];
18   }
19 };
20 const saveJson = (path, data) => fs.writeFileSync(path, JSON.stringify(data,
21
22 program.version(package.version);
23
24 program
25   .command('add [todo]')
26   .description('Adiciona um to-do')
27   .action(async (todo) => {
28     let answers;
29     if (!todo) {
30       answers = await inquirer.prompt([
31         {
32           type: 'input',
33           name: 'todo',
34           message: 'Qual é o seu to-do?',
35           validate: value => value ? true : 'Não é permitido um to-
36         }
37       ]);
38     }
39
40     const data = getJson(todosPath);
41     data.push({
42       title: todo || answers.todo,
43       done: false
```

- Na linha 7, agora incluo o módulo Chalk;
- E na linha 46, retorno a mensagem de sucesso utilizando o método `green` do chalk. Você pode escolher uma cor, um background e um decoration (como por exemplo, underline ou bold) — veja na [documentação do chalk](<https://github.com/chalk/chalk>).

E..... tcharam!

```
rike_@DESKTOP-4GFJL47 MINGW64 /d/projetos/todo-cli (master)
$ todo add to-do
To-do adicionado com sucesso!
```

Você pode adicionar quantas cores quiser; com template strings isso fica muito mais fácil e legível. :)

## E se eu quiser adicionar o to-do com opções adicionais?

Pra isso, podemos utilizar flags! Os famosos argumentos passados com `--` ou `-` depois do comando. O commander nos traz essa possibilidade de maneira EXTREMAMENTE fácil.

Vamos colocar uma opção `--status` no nosso comando de `add`. Dessa forma, se a flag for passada no comando, somada a algum parâmetro, o to-do já será salvo com o atributo `done: {status-passado-na-flag}`.



```
1  #!/usr/bin/env node
2
3  const program = require('commander');
4  const { join } = require('path');
5  const fs = require('fs');
6  const inquirer = require('inquirer');
7  const chalk = require('chalk');
8
9  const package = require('./package.json');
10 const todosPath = join(__dirname, 'todos.json');
11
12 const getJson = (path) => {
13   const data = fs.existsSync(path) ? fs.readFileSync(path) : [];
14   try {
15     return JSON.parse(data);
16   } catch (e) {
17     return [];
18   }
19 };
20 const saveJson = (path, data) => fs.writeFileSync(path, JSON.stringify(data,
21
22 program.version(package.version);
23
24 program
25   .command('add [todo]')
26   .description('Adiciona um to-do')
27   .option('-s, --status [status]', 'Status inicial do to-do')
28   .action(async (todo, options) => {
29     let answers;
30     if (!todo) {
31       answers = await inquirer.prompt([
32         {
33           type: 'input',
34           name: 'todo',
35           message: 'Qual é o seu to-do?',
36           validate: value => value ? true : 'Não é permitido um to-
37         }
38       ]);
39     }
40
41     const data = getJson(todosPath);
42     data.push({
43       title: todo || answers.todo,
```

- Na linha 27, usamos o método `option` pra criar a opção `--status`, junto com a descrição da flag (novamente, será útil para o comando `--help`). Você pode definir um shorthand (`-s`) junto com a flag completa;
- Na linha 28, o último parâmetro sempre será o `options`, para pegar as flags passadas pelo usuário (podem ser infinitas!);
- E na linha 44, adiciono um tratamento para pegar o dado inputado e finalizar como booleano, pois os dados aqui sempre virão como `String`.

```
rike_@DESKTOP-4GFJL47 MINGW64 /d/projetos/todo-cli (master)
$ todo add to-do --status true
To-do adicionado com sucesso!
```

Agora, passando a flag `--status`, o to-do já será salvo com o atributo `done: true`!

## Vamos completar as ações do nosso to-do?

Agora, precisamos de um método para listar nossos to-dos, outro para marcar como feito e outro para marcar como não feito. O básico do funcionamento da interação com a CLI já foi passado, então vou simplesmente exemplificar por aqui:

```
1  #!/usr/bin/env node
2
3  const program = require('commander');
4  const { join } = require('path');
5  const fs = require('fs');
6  const inquirer = require('inquirer');
7  const chalk = require('chalk');
8  const Table = require('cli-table');
9
10 const package = require('./package.json');
11 const todosPath = join(__dirname, 'todos.json');
12
13 const getJson = (path) => {
14   const data = fs.existsSync(path) ? fs.readFileSync(path) : [];
15   try {
16     return JSON.parse(data);
17   } catch (e) {
18     return [];
19   }
20 };
21 const saveJson = (path, data) => fs.writeFileSync(path, JSON.stringify(data,
22 const showTodoTable = (data) => {
23   const table = new Table({
24     head: ['id', 'to-do', 'status'],
25     colWidths: [10, 20, 10]
26   });
27   data.map((todo, index) =>
28     table.push(
29       [index, todo.title, todo.done ? chalk.green('feito') : 'pendente'
30     )
31   );
32   console.log(table.toString());
33 }
34
35 program.version(package.version);
36
37 program
38   .command('add [todo]')
39   .description('Adiciona um to-do')
40   .option('-s, --status [status]', 'Status inicial do to-do')
41   .action(async (todo, options) => {
42     let answers;
43     if (!todo) {
```

- Na linha 22, criei uma função utilitária para pegar os dados e exibir em forma de tabela. Utilizei o módulo `cli-table`, então apenas pego os dados do JSON, crio a instância definindo os headings e os tamanhos das colunas;
- A partir da linha 63, criei o comando `list`, que não precisa receber nenhum parâmetro;
- Depois passo os dados para a função — perceba que utilizei o `chalk` para deixar verdinho o status — e exibo para o usuário.

O resultado é este:

```
rike_@DESKTOP-4GFJL47 MINGW64 /d/projetos/todo-cli (master)
$ todo list
```

id	to-do	status
0	teste	pendente
1	Comprar pão	pendente
2	to-do	feito

Separei a exibição da tabela em uma função, pois vou reaproveitá-la nos métodos de `do` e `undo`, que recebem o id do to-do para serem atualizados:

```
1  #!/usr/bin/env node
2
3  const program = require('commander');
4  const { join } = require('path');
5  const fs = require('fs');
6  const inquirer = require('inquirer');
7  const chalk = require('chalk');
8  const Table = require('cli-table');
9
10 const package = require('./package.json');
11 const todosPath = join(__dirname, 'todos.json');
12
13 const getJson = (path) => {
14   const data = fs.existsSync(path) ? fs.readFileSync(path) : [];
15   try {
16     return JSON.parse(data);
17   } catch (e) {
18     return [];
19   }
20 };
21 const saveJson = (path, data) => fs.writeFileSync(path, JSON.stringify(data,
22 const showTodoTable = (data) => {
23   const table = new Table({
24     head: ['id', 'to-do', 'status'],
25     colWidths: [10, 20, 10]
26   });
27   data.map((todo, index) =>
28     table.push(
29       [index, todo.title, todo.done ? chalk.green('feito') : 'pendente
30     )
31   );
32   console.log(table.toString());
33 }
34
35 program.version(package.version);
36
37 program
38   .command('add [todo]')
39   .description('Adiciona um to-do')
40   .option('-s, --status [status]', 'Status inicial do to-do')
41   .action(async (todo, options) => {
42     let answers;
43     if (!todo) {
```

- Nas linhas 71 e 94, criei os comandos `do` e `undo`, respectivamente. Eles são como o comando `add`: recebem um parâmetro opcional, manipula o JSON, e exibe a tabela final para o usuário.

O resultado é este:

```
rike_@DESKTOP-4GFJL47 MINGW64 /d/projetos/todo-cli (master)
$ todo do 0
To-do salvo com sucesso!
```

id	to-do	status
0	teste	feito
1	Comprar pão	pendente
2	to-do	pendente

E já que reaproveitamos o método da tabela, por que não utilizar também no `add`?

```
rike_@DESKTOP-4GFJL47 MINGW64 /d/projetos/todo-cli (master)
$ todo add Comprar leite --status true
To-do adicionado com sucesso!
```

id	to-do	status
0	teste	feito
1	Comprar pão	pendente
2	to-do	pendente
3	Comprar	feito

## Agora, comandos shell no JavaScript!

Vamos criar um comando totalmente desnecessário — mas para fins didáticos, de backup, apenas com o intuito de aprender a usar o shelljs. Sendo assim, ele executará simplesmente um copy do arquivo `todos.json` para uma pasta `backup` utilizando comandos shell.

```
1  #!/usr/bin/env node
2
3  const program = require('commander');
4  const { join } = require('path');
5  const fs = require('fs');
6  const inquirer = require('inquirer');
7  const chalk = require('chalk');
8  const Table = require('cli-table');
9  const shell = require('shelljs');
10
11  const package = require('./package.json');
12  const todosPath = join(__dirname, 'todos.json');
13
14  const getJson = (path) => {
15    const data = fs.existsSync(path) ? fs.readFileSync(path) : [];
16    try {
17      return JSON.parse(data);
18    } catch (e) {
19      return [];
20    }
21  };
22  const saveJson = (path, data) => fs.writeFileSync(path, JSON.stringify(data,
23  const showTodoTable = (data) => {
24    const table = new Table({
25      head: ['id', 'to-do', 'status'],
26      colWidths: [10, 20, 10]
27    });
28    data.map((todo, index) =>
29      table.push(
30        [index, todo.title, todo.done ? chalk.green('feito') : 'pendente
31      )
32    );
33    console.log(table.toString());
34  }
35
36  program.version(package.version);
37
38  program
39    .command('add [todo]')
40    .description('Adiciona um to-do')
41    .option('-s, --status [status]', 'Status inicial do to-do')
42    .action(async (todo, options) => {
43      let answers;
```



- Na linha 9, incluo o módulo `shelljs` ;
- Na linha 119, comecei a criar o comando `backup` ;
- Primeiro, tento criar a pasta `backup` , caso ela não exista, utilizando o comando `mkdir` do `shelljs`;
- Depois, executo o comando de mover o arquivo passando um parâmetro para ele não logar nada no console, apenas se eu quiser, e guardo numa variável;
- Depois, faço o tratamento de erros: o `exec` pode retornar 3 atributos: `code` e `stderr` (caso dê erro), e `stdin` , caso dê tudo certo. Exibo as mensagens de acordo.

## Lembra do comando `--help`?

Agora o commander brilha:

```
rike_@DESKTOP-4GFJL47 MINGW64 /d/projetos/todo-cli (master)
$ todo --help
Usage: index [options] [command]

Options:
  -V, --version      output the version number
  -h, --help         output usage information

Commands:
  add [options] [todo]  Adiciona um to-do
  list                 Lista os to-dos
  do <todo>            Marca o to-do como feito
  undo <todo>          Marca o to-do como não feito
  backup              Faz um backup dos todos
```

Isso é gerado automaticamente por ele, conforme você utiliza os métodos corretos. Lindo, não?

E uma firula com o FigletJS para nosso terminal ficar mais bonito:

```
1  #!/usr/bin/env node
2
3  const program = require('commander');
4  const { join } = require('path');
5  const fs = require('fs');
6  const inquirer = require('inquirer');
7  const chalk = require('chalk');
8  const Table = require('cli-table');
9  const shell = require('shelljs');
10 const figlet = require('figlet');
11
12 const package = require('./package.json');
13 const todosPath = join(__dirname, 'todos.json');
14
15 const getJson = (path) => {
16   const data = fs.existsSync(path) ? fs.readFileSync(path) : [];
17   try {
18     return JSON.parse(data);
19   } catch (e) {
20     return [];
21   }
22 };
23
24 const saveJson = (path, data) => fs.writeFileSync(path, JSON.stringify(data,
25 const showTodoTable = (data) => {
26   const table = new Table({
27     head: ['id', 'to-do', 'status'],
28     colWidths: [10, 20, 10]
29   });
30   data.map((todo, index) =>
31     table.push(
32       [index, todo.title, todo.done ? chalk.green('feito') : 'pendente
33     )
34   );
35   console.log(table.toString());
36 }
37
38 program.version(package.version);
39
40 console.log(chalk.cyan(figlet.textSync('To-do CLI')));
41
42 program
43   .command('add [todo]')
44   .description('Adiciona um to-do')
```

- Na linha 10, inclui o FigletJS;
- Na linha 39, logo o figlet passando um texto qualquer, utilizando o chalk para deixá-lo com a cor cyan.

E agora sempre que executarmos comandos do nosso CLI, um texto em letras garrafais irá aparecer, dessa forma:

```
rike_@DESKTOP-4GFJL47 MINGW64 /d/projetos/todo-cli (master)
$ todo add "novo todo"
```



```
To-do CLI
```

To-do adicionado com sucesso!

id	to-do	status
0	novo todo	pendente

## Finalizando...

Passei aqui o básico de como começar seu CLI e definir seus comandos, algumas das bibliotecas mais utilizadas tanto para criar a UI quanto facilitar a criação de inputs de dados, bem como executar ações de shell com JavaScript, mas a partir disso, você pode utilizar CLIs para:

- Criar componentes no seu app — utilizando um template engine server-side, como o Pug ou Handlebars;
- Fazer scaffolding de uma estrutura definida de projeto;

- Enviar arquivos para um servidor (podendo-se utilizar libs de SSH ou SFTP);
- Executar tarefas monótonas e repetitivas, como talvez realizar um `git pull` em todos os seus projetos (sim, qualquer comando shell pode ser utilizado com o `shelljs`);
- E mais o que sua imaginação permitir :)

. . .

O mundo de criação de CLIs com JavaScript é ainda mais vasto do que isso; é possível utilizar até React para potencializá-las (utilizando o [Ink](<https://github.com/vadimdemedes/ink>)), libs para fazer loaders animados para requisições ou operações assíncronas ([Ora](<https://github.com/sindresorhus/ora>)), mas provavelmente rende outros artigos.

Agradecimentos especiais aos autores dos artigos abaixo, que me deram um primeiro contato com essa possibilidade:

- <https://codeburst.io/building-a-node-js-interactive-cli-3cb80ed76c86>
- <https://scotch.io/tutorials/build-an-interactive-command-line-application-with-nodejs>

O conteúdo do tutorial está todo neste repositório:

**Rikezenho/todo-cli**

Um to-do feito com CLI. Contribute to Rikezenho/todo-cli development by creating an account on GitHub.

[github.com](https://github.com)

Qualquer dúvida, não hesite em comentar.

Até a próxima, galera!

[Cli](#) [JavaScript](#) [Shell](#) [UI](#) [Commander](#)

**Discover Medium**

Welcome to a place where words matter. On Medium, smart voices and original ideas take center stage - with no ads in sight. [Watch](#)

**Make Medium yours**

Follow all the topics you care about, and we'll deliver the best stories for you to your homepage and inbox. [Explore](#)

**Become a member**

Get unlimited access to the best stories on Medium — and support writers while you're at it. Just \$5/month. [Upgrade](#)

[About](#)

[Help](#)

[Legal](#)