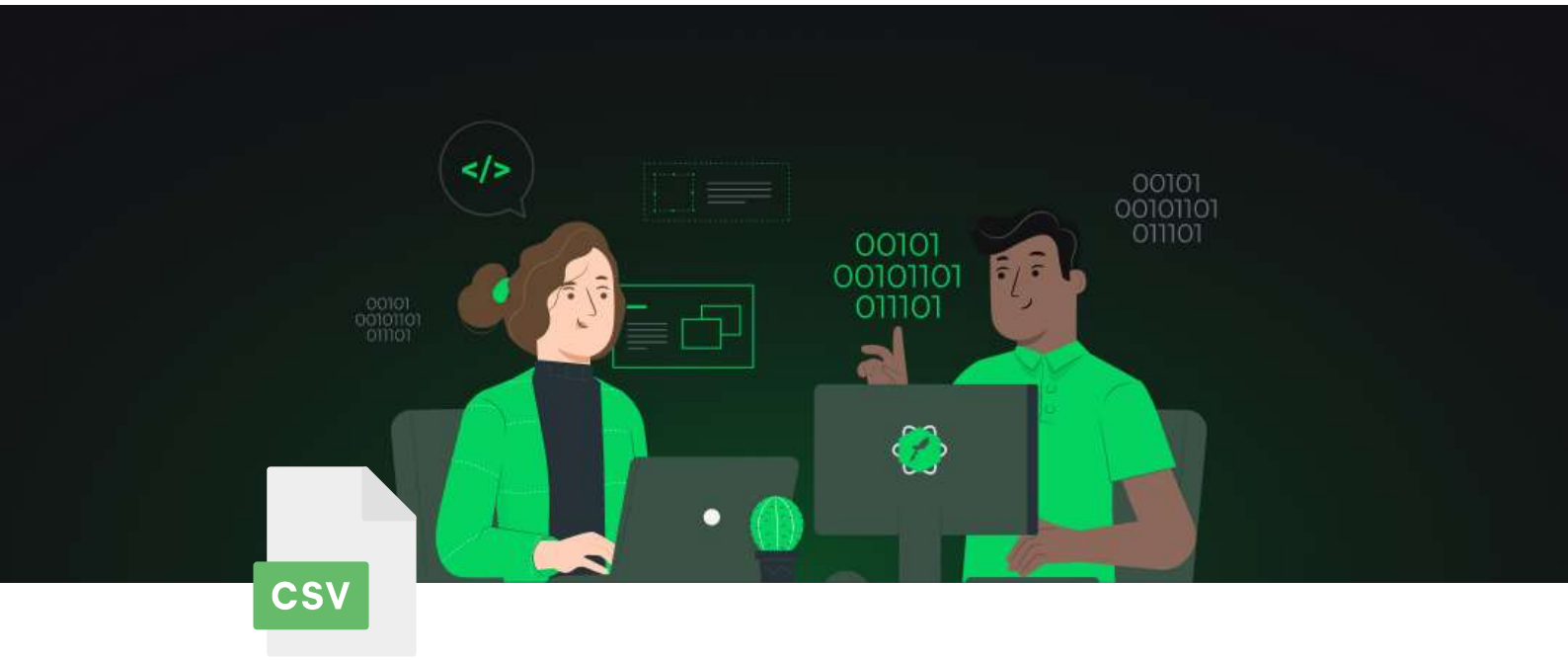

Importando arquivos CSV com Node.js

Nesse documento vamos aprender a importar arquivos CSV (Comma Separated Values) utilizando Node.js e uma biblioteca chamada `csv-parse`.

O que são arquivos .csv?

Lendo arquivos CSV

- Arquivo de exemplo
- Configurando csv-parse
- Identificar final da leitura

Utilizando Promises (async/await)

O que são arquivos `.csv`?

Arquivos `.csv` são exportados através de softwares de planilhas e utilizam um formato semelhante ao seguir:

```
name, country, age Diego, Brazil, 25 John, Russia, 31 Carla, Mexico, 41
```

Shell

Perceba que a primeira linha (opcional) descreve o título dos parâmetros e as demais linhas identificam os valores de cada informação.

Baseado na informação acima entendemos que existem **3 usuários** que poderiam ser convertidos para objetos no JavaScript para facilitar o entendimento:

```
[ { name: 'Diego', country: 'Brazil', age: 25 }, { name: 'John', country:
'Russia', age: 31 }, { name: 'Carla', country: 'Mexico', age: 41 } ]
```

JavaScript

Agora que temos o conhecimento necessário para interpretar os dados contidos em um arquivo CSV precisamos saber como ler as informações contidas dentro de um no Node.js.

Lendo arquivos CSV

Existem milhares (literalmente) de formas para lermos um arquivo CSV no Node.js e converter seus dados em objetos do JavaScript. Podemos utilizar o módulo nativo de arquivos do Node.js, o `fs` ou utilizar alguma biblioteca específica pra isso.

Uma das ferramentas mais fantásticas pra realizar esse processo, na minha opinião, é o `csv-parse` (https://www.npmjs.com/package/csv-parse), mas por que?

Essa biblioteca consegue ler o arquivo CSV por partes, sem precisar lê-lo todo de uma só vez, otimizando o uso de memória do nosso servidor e processamento. Esse processo "por partes" consegue ser realizado graças a uma funcionalidade que o Node.js possui que são as streams.

> 💡 Streams são formas de recebermos ou enviarmos dados por partes sem precisarmos saber exatamente quando essa mensagem vai terminar, nos preocupamos apenas em ler/escrever os dados assim que disponíveis.
>
> Assim, mesmo que a quantidade de dados seja imensa, pro nosso servidor importa apenas uma pequena parte desses dados, tornando tudo mais leve.
>
> Você pode comparar isso com o processo do Youtube onde você não precisa baixar o vídeo completo pra assistir e sim apenas um pequeno pedaço.

Arquivo de exemplo

Para exemplificarmos a leitura de um arquivo `.csv` baixe o arquivo abaixo e mantenha-o dentro de uma pasta qualquer da sua aplicação Node.js:

import_template.csv 0.1KB

Configurando `csv-parse`

Instale o `csv-parse` utilizando o Yarn ou NPM e depois disso crie um arquivo TypeScript que vai importar seu CSV:

```
import csvParse from 'csv-parse'; import fs from 'fs'; import path from
'path'; const csvFilePath = path.resolve(__dirname, 'import_template.csv');
const readCSVStream = fs.createReadStream(csvFilePath);
```

TypeScript

No exemplo importamos, além do csv-parse, o módulo nativo do Node.js para lidar com arquivos físicos, o `fs` e o módulo nativo `path` que utilizamos para referenciar o caminho até o arquivo CSV, que no exemplo estaria na mesma pasta que esse arquivo TypeScript;

Na última linha do exemplo criamos nossa `stream` de leitura com o método `createReadStream` do módulo `fs`. Essa função faz o papel de ler o arquivo por partes conforme for necessário sem precisar armazenar o arquivo inteiro na memória da aplicação.

```
const parseStream = csvParse({ from_line: 2, ltrim: true, rtrim: true, });
const parseCSV = readCSVStream.pipe(parseStream);
```

TypeScript

Continuando, utilizamos o módulo `csv-parse` para criar uma variável `parseStream` que também é uma stream de leitura, assim como o `readCSVStream`.

Passamos o parâmetro `from_line: 2` pra que ele descarte a primeira linha do arquivo que, nesse caso, não são dados e sim o título para cada coluna. Os outros dois parâmetros servem pra remover espaços em branco desnecessários que ficam entre cada um dos valores.

Na linha abaixo, criamos uma variável que armazena o resultado da instrução: `readCSVStream.pipe(parseStream)`.

Mas o que é esse bendito `.pipe`? O `pipe` é um método presente em toda `stream` que simplesmente transmite a informação de uma stream pra outra, ou seja, estamos falando pra `stream` de leitura do arquivo CSV representada pela variável `readCSVStream` que sempre que ela tiver uma informação disponível, deve envia-la para a nossa outra stream, a `parseStream`.

E salvamos essa comunicação entre as duas streams porque assim conseguimos monitorar quando esses dados são transmitidos de um lado para o outro:

```
parseCSV.on('data', line => { console.log(line); });
```

TypeScript

Nessa linha agora estamos "ouvindo" as novas informações obtidas do arquivo CSV, linha por linha e imprimindo-as em tela. A biblioteca `csv-parse` nos garante que sempre teremos a linha completa nesse caso, ela não retornará parte do arquivo se não for uma linha completa.

Executando o exemplo até aqui teríamos um resultado semelhante ao seguir:

```
['Loan', 'income', '1500', 'Others'] ['Website Hosting', 'outcome', '50 ',
'Others'] ['Ice cream', 'outcome', '3 ', 'Food']
```

TypeScript

O que podemos perceber nesse resultado é que cada linha é retornada como um array (vetor) e cada posição no vetor representa uma coluna do CSV.

Identificar final da leitura

Agora que entendemos como ler linha por linha do arquivo CSV, como saber que a leitura foi totalmente finalizada? Toda `stream` no Node dispara um evento chamado `end` assim que a comunicação é finalizada e, com isso, podemos ouvi-lo:

```
parseCSV.on('end', () => { console.log('Leitura do CSV finalizada'); });
```

TypeScript

Utilizando Promises (async/await)

Por padrão a leitura das linhas do arquivo CSV acontece de forma gradativa e por isso utilizamos a sintaxe de Event Listeners (`.on('event')`) pra saber quando um novo dado está disponível e também pra saber quando o processo finaliza.

Mas quando precisamos lidar com `async/await` nem sempre essa sintaxe trará uma flexibilidade bacana, então podemos utilizar de alguns hacks para usar o `await`, por exemplo, na hora de saber que a leitura finalizou:

```
import csvParse from 'csv-parse'; import fs from 'fs'; import path from
'path'; async function loadCSV(filePath: string): any[] { const
readCSVStream = fs.createReadStream(csvFilePath); const parseStream =
csvParse({ from_line: 2, ltrim: true, rtrim: true, }); const parseCSV =
readCSVStream.pipe(parseStream); const lines = []; parseCSV.on('data', line
=> { lines.push(line); }); await new Promise(resolve => {
parseCSV.on('end', resolve); }); return lines; } const csvFilePath =
path.resolve(__dirname, 'import_template.csv'); const data =
loadCSV(csvFilePath);
```

TypeScript

No exemplo acima precisamos prestar atenção em três pontos importantes:

1. Criamos uma função chamada `loadCSV` pois assim podemos transforma-la em uma função assíncrona `async` para conseguir usar o `await`;
2. Criamos um array que armazenará os dados das linhas ao invés de utiliza-los assim que disponíveis dentro do método `parseCSV.on('data', () => { ... })`;
3. Criamos uma Promise e a marcamos como resolvida (finalizada com sucesso) assim que recebermos o evento `end` do `parseCSV` que indica que a leitura foi finalizada. A criação dessa Promise nos permitiu utilizar o método `await` para aguardar a leitura completa antes de retornar os dados lidos no final da função.