06

Streaming de dados

Streaming de dados

Muitas das *apis* assíncronas utilizadas no Node.js trabalham utilizando o chamado *buffer mode*. Em uma operação de entrada de dados o *buffer mode* faz com que todos os dados oriundos da requisição fiquem armazenados em um *buffer*. Para que então seja passado para algum *callback* tão somente todo o recurso tenha sido lido.

Esse tipo de estratégia não é muito interessante para o Node.js, visto que ele é pensando para trabalhar com operações de I/O e faz isso muito bem, mas quando a necessidade é ficar manipulando muito recurso na memória, que é o que fazem os *buffers,* ele passa a perder bastante performance.

A solução ideal para resolver essa questão seria conseguir ir processando os dados conforme eles fossem chegando, ao invés de ter que esperar a leitura do dado por inteiro. É exatamente isso que a *api* de **Streams** do Node.js possibilita ao programador.

Existem diversas vantagens em se utilizar essa *api* sempre que possível. Veja as principais:

Eficiência espacial

Primeiro de tudo, os **Streams** nos permitem fazer coisas que não seriam possíveis somente bufferizando dados processando-os todos de uma vez.

Por exemplo: imagine que a aplicação precise ler arquivos muito grandes da ordem dos mega ou até gigabytes. Agora imagine que a aplicação precise ler alguns desses arquivos de forma concorrente. Claramente utilizar uma *api* que retorna um grande *buffer* quando o arquivo está completamente lido não é uma boa ideia. A aplicação iria fatalmente cair por falta de memória.

Além disso, os *buffers* na V8, a *runtime* sobre a qual o Node.js roda, suporta como tamanho máximo para *buffers* um número que sequer chega a 1 GB de memória. Ou seja a aplicação esbarraria em uma limitação do próprio ambiente antes mesmo de chegar ao limite físico de memória.

Utilizar **Streams** nessa situação reduz em muito o uso de memória da aplicação e faz com que ela rode de uma maneira mais suave.

Eficiência temporal

Vamos considerar agora o caso de uma aplicação que precisa comprimir um arquivo e após fazer o seu *upload* para um servidor HTTP remoto. Este servidor então recebe o arquivo, descompacta e salva-o em seu sistema de arquivos.

Se o cliente tiver sido implementando utilizando uma *api* com *buffers*, o *upload* só iria iniciar uma vez que o arquivo inteiro tenha sido lido e compactado. Do outro lado e descompressão também só iria ter início uma vez que o arquivo inteiro tenha sido recebido.

Uma melhor solução seria implementar a mesma funcionalidade com **streams**. Do lado do cliente, os **streams** permitem que aplicação vá lendo e enviando os arquivos conforme eles forem sendo lidos do sistema de arquivos. E do lado do servidor ela permite que seja feita a descompactação de cada pedaço de arquivo, conforme eles vão sendo recebidos do cliente remoto. Esses pedaços costumam ser chamados de **chunks**.

É bem fácil perceber que a abordagem com **streams** gera um ganho de tempo gigantesco e que escala bem melhor.

Lendo um arquivo com buffers

A maneira tradicional de ser ler arquivos em Node.js é utilizando a lib `fs` . Por ser uma lib do core do Node, ela nem precisa ser instalada via npm, bastando apenas ser importada através do `require` no arquivo em se deseja implementar a leitura:

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


fs.readFile(.arquivo.txt, function(err, buffer){
    console.log('lendo um arquivo');
});
```

Desde que o 'arquivo.txt' estea localizado na raíz do projeto, essa é a maneira mais simples de ler um arquivo. Note que a função `readFile` recebe como parâmetros o caminho do arquivo e uma função de *callback*. Você também deve ter percebido que nessa função de *callback* estamos passando uma variável para tratamento de erros e uma outra que identificamos como *buffer*. Portanto esta é obviamente a forma de leitura de arquivos *buffer mode* e que sabemos que não é a mais recomendada para essa tarefa.

Em breve veremos que implementação usando **streams** também é bem simples. Mas antes vamos enriquecer e dar um pouco mais de complexidade ao exemplo. Imagine que esse arquivo agora precisa ser lido e logo em seguida escrito com um nome diferente.

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

fs.readFile(arquivo, function(err, buffer){
  console.log('lendo um arquivo');

  fs.writeFile('arquivo2.txt' , buffer, function(err){
      console.log('escrevendo um arquivo');
  });

});
```

A implementação continua sendo muito simples. Basta que uma das ações executadas dentro do *callback* da ação de leitura seja exatamente a ação de escrita, que implementamos utilizando a função `writeFile` , que recebe também o nome do arquivo, o *buffer* si, ou seja aquilo que sera escrito e uma função de *callback*.

Essa forma de implementação funciona perfeitamente bem. O grande problema com ela é que as ações são sequenciais e uma depende totalmente que a anterior termine para que possa ser executada.

Lendo um arquivo com streams

A alternativa ao uso dos *buffers* é, como já foi falado, o uso de **streams** e a grande sacada do uso dessa *api* é que ela já possui integração nativa com diversos pacotes do Node. Para que passemos a fazer a leitura, compressão e escrita do arquivo com *stream* não precisamos importar mais nenhuma lib além das que já utilizamos. Basta mudar as funções invocadas e a estratégia será completamente diferente e bem mais eficiente.

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

fs.createReadStream('arquivo.txt')
  .on('finish', function(){
    console.log('arquivo lido.');
  });
```

Ao invés de invocar funções diretamente, a *api* de **streams** tem uma semântica diferente. A ideia é que cada invocação a essa *api* crie um fluxo de execução da tarefa solicitada. No nosso primeiro exemplo, queríamos somente ler e por isso criamos um fluxo de leitura através da função `createReadStream`, que pode ser invocada diretamente a partir do próprio objeto de leitura de arquivos, pois como já foi dito anteriormente, existe uma integração nativa entre essas duas libs.

Uma vez que o fluxo foi criado, precisávamos informar ao usuário o momento em que ele terminou para que ele saiba que o arquivo foi lido por completo corretamente. Como execução não é mais sequencial, não temos como utilizar uma função de callback. A alternativa para essa necessidade é o uso de **listeners** na *api* de **streams**.

Para saber o momento em que o processamento do fluxo terminou, utilizamos o *listener* `on`, recebendo como parâmetro a string 'finish' e uma função. Pronto! O Node já sabe exatamente que o que queremos é executar essa função, uma vez que este fluxo esteja finalizado.

Agora, e para implementar os outro requisito de escrever o arquivo, o que pode ser utilizado? Não existem mais *callbacks* e nem nos atenderia plenamente, pois queremos que essas execuções também seja assíncronas. O ideal seria que fosse criado um **pipeline** de execuções.

Para implementar tal necessidade, a *api* de **streams** tem a função `pipe()`. A função dela é exatamente a de criar um novo canal para que um novo fluxo possa ser criado e passe a executar ações concorrentemente à que já estava sendo executada e, possivelmente trabalhando sobre o mesmo conjunto de dados:

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

fs.createReadStream('arquivo.txt')

  .pipe(fs.createWriteStream('arquivo.txt'))

  .on('finish', function(){
    console.log('arquivo escrito.');
  });
```

A invocação do `pipe()` pode ser feita de maneira fluente a partir de uma chamada anterior, desde que essa também conheça a *api* de streams. Como parâmetro dessa função é passada uma invocação de uma nova função. No caso foi criar um fluxo para escrita de arquivo através da função `createWriteStream()`.

Implementando o upload de um arquivo

O sistema PayFast possui um novo requisito: é preciso agora que ele seja capaz de receber um arquivo e salvá-lo, pois em muitas situações as empresas de cartões de crédito ou de outros meios de pagamento podem exigir fotos de documentos para comprovar que a pessoa que está utilizando o cartão realmente é o seu proprietário.

Dessa forma, tudo que o PayFast precisa saber fazer é disponibilizar uma rota para receber as requisições para esse upload e salvar em uma pasta para que depois ele receba uma análise humana.

Pensando primeiro do lado do cliente, a forma que ele poderia utilizar para enviar o arquivo seria bem semelhante ao que já tem sido feito: uma requisição HTTP que envia um *body*, com a difierença que o *body* agora será um arquivo e não somente uma string representando um json.

Escrevendo essa requisição com o *curl* ficaria algo assim:

```
curl -X POST http://localhost:3000/upload/imagem -v
    -H "filename: <nome_do_arquivo>"
    -H "Content-Type: application/octet-stream"
    --data-binary @imagem.jpg
```

As novidades aqui ficam por conta dos 2 *headers*:

- **filename**: utilizado para passar a informação do nome do arquivo da maneira que o cliente deseja que chegue ao servidor.
- **Content-Type: application/octet-stream**: o header em si já é conhecido. A novidade é o valor que ele recebeu, que indica que o arquivo é um tipo binário.

Na última linha do exemplo, o parâmetro `--data-binary` indica que um arquivo binário será enviado no *body* da requisição e em seguida é passado seu valor acompanhado de uma '@' que informa que este é um arquivo e não uma String literal.

Criando a rota para receber um arquivo

A nova rota a ser criada atende um requisito bem específico, que é receber um *upload* de arquivo, então faz sentido que ele tenha um *controller* próprio. Podemos então criar o `upload.js` dentro da pasta `controllers` e implementar a rota, que deve ser acessada via POST:

```
module.exports = function(app) {
    app.post("/upload/imagem", function(req, res) {

      var arquivo = req.headers.filename;
      console.log('arquivo recebido: ' + arquivo);

    });
  }
```

O primeiro passo é criar a rota da mesma forma que já foi feito várias vezes e dentro da função de *callback* dessa rota, ler o cabeçalho que possui o nome do arquivo. Para ler um *header* da requisição, basta pedir ao objeto que contem os dados da mesma. Com essa informação em mãos, podemos imprimir no console para ter certeza de que o funcionamento está correto.

```
...
  var arquivo = req.headers.filename;
  console.log('arquivo recebido: ' + arquivo);
...
```

O próximo passo é implementar de fato a leitura e escrita do arquivo. Como desejamos fazer isso com uso de **streams** para evitar os contrapontos do uso de *buffer*, precisamos lançar mão dos `pipes()`.

Como o **express** também implementa a *api* de **stream** é possível invocar o `pipe` a partir do próprio `req` :

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

module.exports = function(app) {
    app.post("/upload/imagem",function(req, res) {

      var arquivo = req.headers.filename;
      console.log('arquivo recebido: ' + arquivo);

            req.pipe(fs.createWriteStream("files/" + arquivo))

        });
    });
  }
```

Por fim, a rota precisa informar ao cliente da requisição que o upload e escrita do arquivo foi concluido com sucesso ou não, da mesma maneira que todas as rotas anteriores fizeram. A diferença é que aqui não temos *callback*, então será preciso utilizar um *listener* que informe que o processamento chegou ao fim:

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

module.exports = function(app) {
    app.post("/upload/imagem",function(req, res) {

      var arquivo = req.headers.filename;
      console.log('arquivo recebido: ' + arquivo);

      req.pipe(fs.createWriteStream("files/" + arquivo))

        .on('finish', function(){
               console.log('arquivo escrito');
              res.status(201).send('ok');
        });
    });
  }
```

Agora temos o fluxo completo implementado, recebendo o arquivo aplicando todos os processamentos necessários via **stream**, o que garante uma boa performance.