UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

MARCUS VINÍCIUS DE FARIAS BARBOSA

EXPRESS-IMAGE-SERVER: UMA BIBLIOTECA PARA CONSTRUÇÃO DE SERVIDOR DE IMAGENS COM NODE.JS

CAMPINA GRANDE - PB

2022

MARCUS VINÍCIUS DE FARIAS BARBOSA

EXPRESS-IMAGE-SERVER: UMA BIBLIOTECA PARA CONSTRUÇÃO DE SERVIDOR DE IMAGENS COM NODE.JS

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Professor Dr. João Arthur Brunet Monteiro.

CAMPINA GRANDE - PB

2022

MARCUS VINÍCIUS DE FARIAS BARBOSA

EXPRESS-IMAGE-SERVER: UMA BIBLIOTECA PARA CONSTRUÇÃO DE SERVIDOR DE IMAGENS COM NODE.JS

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. João Arthur Brunet Monteiro

Orientador – UASC/CEEI/UFCG

Professor Dr. Dalton Dario Serey Guerrero

Examinador – UASC/CEEI/UFCG

Francisco Vilar Brasileiro

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 02 de Setembro de 2022.

CAMPINA GRANDE - PB

RESUMO

Aplicações web que lidam com um grande volume de imagens precisam de estratégias para garantir uma boa performance. Uma delas é requisitar as imagens ao servidor especificando os atributos esperados. Existem serviços de terceiros que implementam essa estratégia e oferecem uma API para comunicação com um servidor com capacidades de manipulação de imagens e CDN. No entanto, tais serviços não são completamente gratuitos e não permitem que o desenvolvedor construa um servidor próprio sem um grande esforço para implementar todos os componentes necessários. Neste trabalho é apresentada a express-image-server, uma biblioteca Node.js open source cujo objetivo é auxiliar no processo de construção desse tipo de servidor. A express-image-server oferece recursos para processamento dinâmico de imagens e comunicação com bases de dados para busca e armazenamento, abstraindo toda a lógica necessária para verificação, transformação e geração de imagens variantes. Assim, a express-image-server se apresenta como uma solução alternativa para os desenvolvedores que desejam construir seus próprios sistemas para serviço e processamento de imagens com Node.js.

express-image-server: uma biblioteca para construção de servidor de imagens com Node.js

Marcus Vinícius de Farias Barbosa
marcus.barbosa@ccc.ufcg.edu.br
Departamento de Sistemas e Computação
Universidade Federal de Campina Grande
Campina Grande, Paraíba - Brasil

João Arthur Brunet Monteiro (Orientador)
joao.arthur@computacao.ufcg.edu.br
Departamento de Sistemas e Computação
Universidade Federal de Campina Grande
Campina Grande, Paraíba - Brasil

RESUMO

Aplicações *web* que lidam com um grande volume de imagens precisam de estratégias para garantir uma boa performance. Uma delas é requisitar as imagens ao servidor especificando os atributos esperados. Existem serviços de terceiros que implementam essa estratégia e oferecem uma API para comunicação com um servidor com capacidades de manipulação de imagens e CDN. No entanto, tais serviços não são completamente gratuitos e não permitem que o desenvolvedor construa um servidor próprio sem um grande esforço para implementar todos os componentes necessários. Neste trabalho é apresentada a *express-image-server*, uma biblioteca Node.js *open source* cujo objetivo é auxiliar no processo de construção desse tipo de servidor. A *express-image-server* oferece recursos para processamento dinâmico de imagens e comunicação com bases de dados para busca e armazenamento, abstraindo toda a lógica necessária para verificação, transformação e geração de imagens variantes. Assim, a *express-image-server* se apresenta como uma solução alternativa para os desenvolvedores que desejam construir seus próprios sistemas para serviço e processamento de imagens com Node.js.

Palavras-chave

Image processing server, Express server, Node.js library

Repositório

https://github.com/vinifarias/express-image-server

1 INTRODUÇÃO

Muitas aplicações *web* modernas têm que lidar com a exibição de imagens no lado do cliente. Essa tarefa pode se tornar um desafio quando a quantidade e tamanho dos arquivos de imagem são grandes, pois, dependendo da conexão à internet do usuário, poderá levar um longo tempo para que os arquivos sejam baixados e as imagens finalmente exibidas. É comum que o servidor forneça as imagens em seu tamanho e resolução originais, porém isso não é o ideal, já que muitas vezes elas não serão renderizadas no cliente com essas configurações. Nesse caso, o cliente pediria ao servidor uma imagem mais pesada que o necessário, o que aumentaria o tempo de *download* e o consumo de recursos.

Uma forma de solucionar esse problema é construir um servidor que armazena e fornece variantes das imagens, ou seja, imagens em tamanhos e resoluções diferentes das originais, e que permite que o cliente as requisite especificando os parâmetros desejados de acordo com a situação. Assim ele poderia realizar uma requisição passando as dimensões e resolução esperadas para a imagem, o que diminuiria o consumo de dados e o tempo de *download*, melhorando assim a performance da página e, consequentemente, a experiência do usuário final.

Existem soluções comerciais, como o ImageKit[1], Cloudinary[2] e Optimole[3], que fornecem serviços de armazenamento, distribuição e processamento de imagens através de API REST, e que já são integradas a serviços de CDN. No entanto, essas soluções se tornam pagas a partir de um certo ponto de uso e não disponibilizam recursos para a construção de um servidor próprio de forma gratuita, simplificada e sem a dependência e restrições de um serviço de terceiros.

Por outro lado, construir um serviço próprio desse tipo não é uma tarefa trivial, pois envolve uma série de decisões e camadas de implementação para que venha a funcionar corretamente. É preciso definir onde as imagens serão armazenadas, se será localmente ou em algum serviço de nuvem; se será utilizada uma CDN; e, principalmente, como se dará a implementação da lógica de processamento de imagens e da API que será exposta para os usuários realizarem as operações. Isso significa que para o desenvolvedor construir um sistema englobando todos os pontos citados ele precisá implementar cada um deles do zero ou conhecer ferramentas específicas que os implementem.

Buscando oferecer uma ferramenta *open source* gratuita que auxilie no processo de construção desse tipo de sistema, foi desenvolvida a biblioteca *express-image-server*. A *express-image-server* é uma biblioteca pública Node.js[4] que disponibiliza *middlewares* que podem ser acoplados às rotas de um servidor Express[5]. Os *middlewares* disponibilizados fornecem uma API para processamento dinâmico de imagens, como redimensionamento e conversão de formato, e conseguem se comunicar com a base de dados para busca e armazenamento das imagens processadas, abstraindo toda a lógica necessária para verificação, transformação e geração de novas imagens. Dessa forma, a *express-image-server* se apresenta como uma solução alternativa para os desenvolvedores que desejam construir

[1] https://imagekit.io
[2] https://cloudinary.com
[3] https://optimole.com
[4] https://nodejs.org
[5] http://expressjs.com

seus próprios sistemas para serviço e processamento de imagens com Node.js.

O artigo está organizado como se segue: na Seção 2, apresenta-se a arquitetura e as funcionalidades da *express-image-server*. Na Seção 3, é descrito a utilização da ferramenta. Na Seção 4, são detalhados os testes de avaliação e os resultados obtidos. Na Seção 5, é apresentada a experiência adquirida ao longo do desenvolvimento do trabalho e, por fim, na Seção 6, os trabalhos futuros.

2 SOLUÇÃO PROPOSTA

A *express-image-server* é uma biblioteca Node.js que conta com as seguintes funcionalidades para construção de um serviço de processamento de imagens: busca e transformação dinâmica de imagens; lógica de comunicação com a base de dados para busca e envio de imagens; e *upload* de imagens direto para a base de dados.

Ela foi desenvolvida utilizando Node.js, que trata-se de um ambiente de execução Javascript no lado do servidor [1], o que possibilita a construção de aplicações em JavaScript fora do *browser*. Como linguagem de programação foi utilizada a TypeScript [6], uma linguagem fortemente tipada construída a partir de JavaScript. Além disso, foram utilizadas bibliotecas auxiliares que serão melhor descritas ao longo desta seção.

As funcionalidades disponíveis são fornecidas através de *middlewares* que podem ser acoplados às rotas de um servidor Express, que consiste em um *framework* Node.js para construção de aplicações *web*. Os *middlewares* oferecidos são:

- `processMiddleware`: utilizado para busca e processamento dinâmico de imagens;
- `uploadMiddleware`: utilizado para envio de imagens diretamente para a base de dados..

Conforme detalhado na Figura 1, a biblioteca é organizada em:(1) Middlewares, que já foram descritos brevemente; (2) ImageTransformer, que realiza as operações de transformação de imagem; (3) Helpers, funções de uso geral da ferramenta; e (4) Storages, que são responsáveis pela comunicação com a base de dados.

Ao longo desta seção, os componentes arquiteturais e as decisões tomadas serão descritos em detalhes.

2.1 *Middlewares*

No contexto de Node.js e Express, *middlewares* são funções que atuam entre uma requisição HTTP e a resposta final que o servidor envia para o cliente. Essas funções possuem acesso ao objeto de requisição e ao objeto de resposta, e podem realizar as seguintes operações: (1) executar qualquer código; (2) aplicar alterações nos objetos de requisição e resposta; (3) invocar o próximo *middleware* na pilha; e (4) encerrar o ciclo requisição-resposta [2].

Nesse contexto, a *express-image-server* disponibiliza dois *middlewares* para uso, os quais serão detalhados em seguida.

2.1.1 *processMiddleware.*

Este *middleware* provê a uma rota da aplicação a funcionalidade de pesquisa e processamento dinâmico de imagens, através de uma API disponibilizada pela biblioteca. Ele é capaz de receber uma requisição, buscar a imagem na base de dados, processá-la aplicando

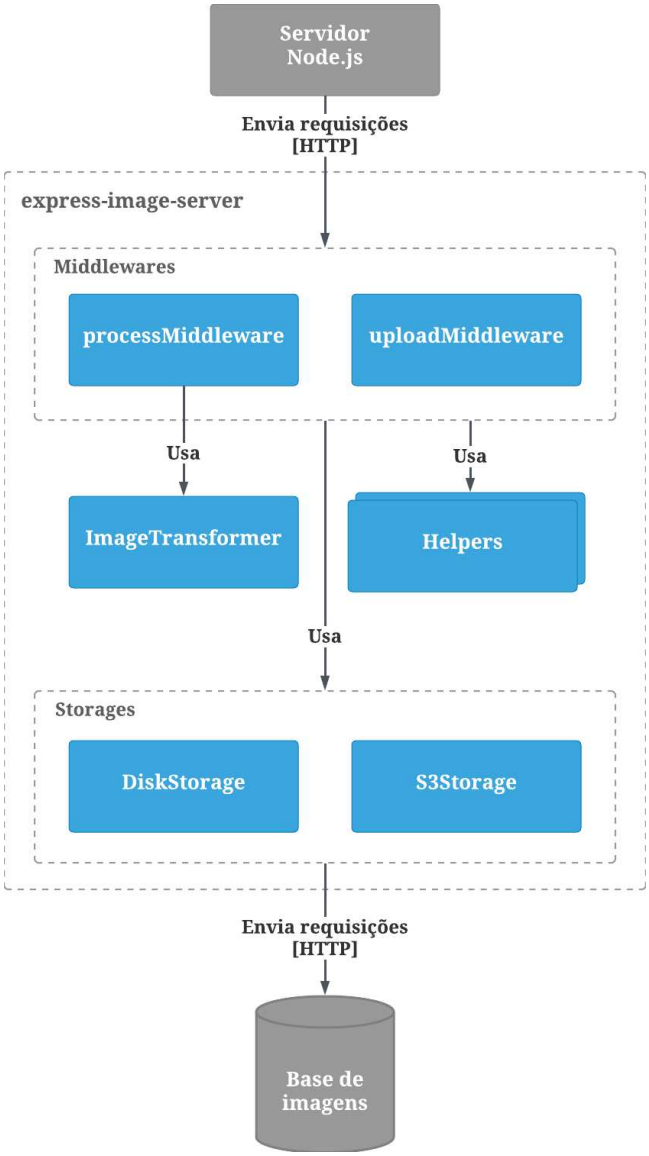

Figura 1: Arquitetura base da *express-image-server*.

as operações requisitadas e devolvê-la como resposta, além de salvá-la na base dados para requisições idênticas futuras.

Quando esse *middleware* é acoplado a uma rota, esta poderá receber requisições GET no formato mostrado pela Figura 2. Nessa requisição, é esperado que o identificador da imagem seja enviado no parâmetro *id* da URL, isto é, no *req.params.id*, e as operações desejadas como *query params* da URL, ou seja, no *req.query*.

A Figura 3 descreve o fluxo de ações realizadas por esse *middleware*. A primeira ação, ao receber a requisição, é normalizar os parâmetros recebidos. Isso é feito para garantir que apenas as operações esperadas estejam presentes e validar os valores passados para cada uma delas. Após isso, é verificado se a imagem processada com as operações pedidas já existe na base de dados. Se ela existe, ela é capturada da base e retornada como resposta da requisição.

[6] https://www.typescriptlang.org/

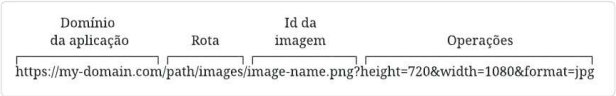

Figura 2: Estrutura da rota do *processMiddleware*.

Se não, o *middleware* captura a imagem original da base de dados e a processa aplicando as operações pedidas, gerando uma variante. Feito isso, a imagem variante recém-processada é retornada como resposta da requisição e salva na base de dados para consultas futuras.

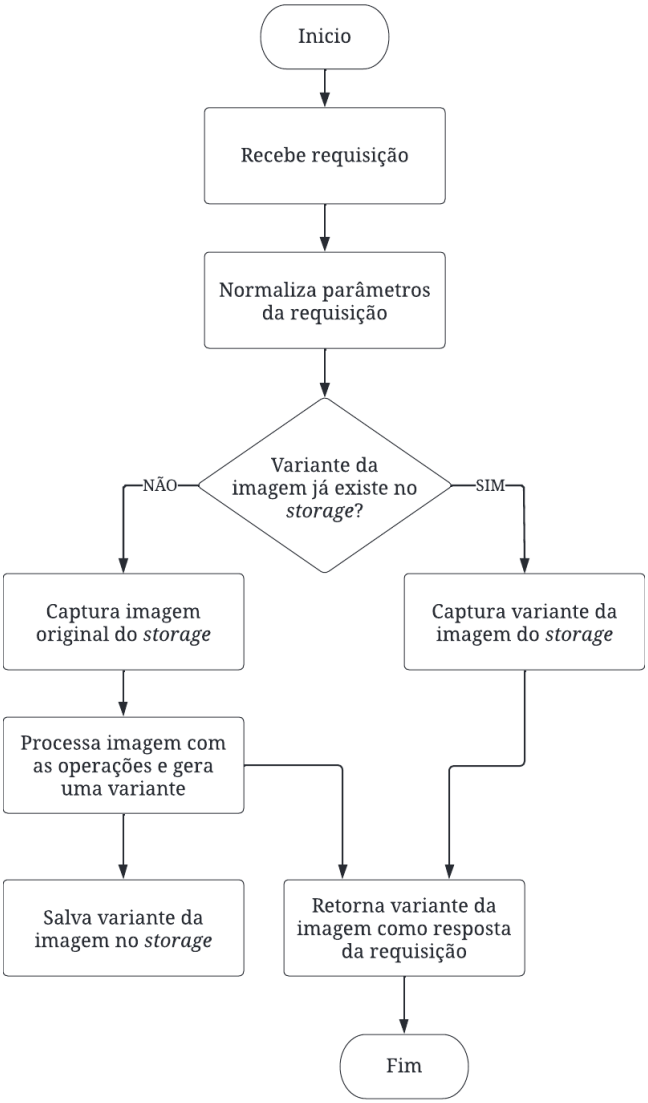

Figura 3: Fluxograma do *processMiddleware*.

Ao processar uma imagem, é gerado um novo nome de arquivo a partir do nome original da imagem (identificador), seu formato e as operações aplicadas. De modo que, quando ela é retornada como resposta da requisição ou salva na base de dados, ela é enviada com esse nome que foi gerado. O padrão de nomenclatura é o seguinte:

<identificador-da-imagem> , <operação-valor> , ... , <operação-valor> . <formato>

Por exemplo, para a requisição:
GET https://.../image1.jpeg?height=100&width=200&format=png
O seguinte nome será gerado: *image1,height-100,width-200.png*

A Tabela 1 descreve as operações de transformação de imagem disponíveis pelo `processMiddleware`.

Tabela 1: Operações do *processMiddleware*

Query param	Descrição
height	Altura da imagem
width	Largura da imagem
format	Formato da imagem

2.1.2 uploadMiddleware.

Este *middleware* provê a uma rota do servidor a funcionalidade de realizar *upload* de imagens diretamente para a base de dados. Permitindo assim que imagens sejam armazenadas através de uma simples requisição.

Para a implementação dessa funcionalidade foi utilizada a biblioteca Multer[7], que já fornece um *middleware* para *upload* de arquivos. Desse modo, para realizar o envio de uma imagem, é preciso enviar uma requisição POST com um *body* configurado como *form-data* e o arquivo no parâmetro *file*, o qual é esperado internamente pela função.

2.2 *Storages*

Storages são classes responsáveis por realizar a comunicação com uma base de dados de imagens. Por padrão, a *express-image-server* fornece duas classes *Storage*:

- **DiskStorage**: utilizada quando o armazenamento de imagens é o próprio disco rígido da máquina em que o servidor está sendo executado;
- **S3Storage**: utilizada quando o armazenamento é feito no serviço de nuvem AWS S3[8].

Essas classes são implementações de uma interface chamada *Storage*, que é definida pela *express-image-server* e é esperada na configuração dos *middlewares*. Essa interface é disponibilizada para o desenvolvedor, de modo que ele tenha a liberdade de configurar a *express-image-server* para que se comunique com a base de dados que desejar e implemente sua própria forma de realizar isso.

A interface *Storage* define os seguintes métodos:

- `save`: salva uma imagem na base de dados;
- `fetch`: captura uma imagem da base de dados;
- `exists`: verifica se uma determinada imagem existe na base de dados;

[7] https://www.npmjs.com/package/multer
[8] https://aws.amazon.com/pt/s3/

- `getMulterStorage`: retorna um *Store Engine*[3] do *Multer*.

Todos os métodos definidos na interface são obrigatórios, com exceção do `getMulterStorage`. No entanto, este método se torna obrigatório quando a classe é utilizada para configurar o `uploadMiddleware`. Isso acontece pois o Multer, utilizado pelo *middleware*, precisa receber em sua configuração um *Store Engine*, que é uma classe própria do Multer responsável por armazenar arquivos e retornar informações sobre como acessá-los[3].

O método `getMulterStorage` da classe DiskStorage retorna um *Store Engine* do Multer específico para armazenamento de imagens em disco [9]. A classe S3Storage retorna um *Store Engine* customizado para AWS S3, para isso foi utilizada a biblioteca MulterS3[10], que já disponibiliza esse recurso.

2.3 *ImageTransformer*

O *ImageTransformer* é o componente responsável pela aplicação das operações pedidas numa imagem, gerando assim uma nova variante.

A implementação foi feita utilizando a Sharp.js[11], uma biblioteca Node.js de alta performance para processamento de imagens. Ela foi utilizada pois é de fácil uso e fornece um extensa gama de operações, que vão desde o redimensionamento de imagens e conversão de formato até manipulação de cor e composição de imagens.

Além disso, através de um teste de *benchmarking*[12] realizado por sua equipe de desenvolvimento, ela se demonstrou mais eficiente que outras ferramentas similares, o que corroborou a decisão de utilizá-la.

2.4 *Helpers*

Os *Helpers* são funções de uso geral que podem ser utilizadas ao longo de toda a ferramenta. Destacam-se aqui duas funções importantes utilizada durante o fluxo de ações do `processMiddleware`:

- `generateFileName`: gera um nome de arquivo considerando as operações requisitadas e o identificador da imagem, esse nome é utilizado para nomear a variante gerada pelo processamento de uma imagem;
- `normalizeQuery`: normaliza os parâmetros recebidos advindos do `processMiddleware`, removendo as operações não esperadas pela ferramenta, checando os tipos dos valores recebidos e corrigindo-os para os tipos esperados quando possível.

3 UTILIZAÇÃO DA FERRAMENTA

3.1 Instalação

Como a *express-image-server* foi disponibilizada como uma biblioteca npm[13], é preciso instalá-la como dependência em um projeto Node.js. Isso pode ser feito através dos comandos mostrados na Figura 4.

Tendo concluído a instalação, os recursos fornecidos pela biblioteca já estarão disponíveis para uso.

[9] https://github.com/expressjs/multer#diskstorage
[10] https://www.npmjs.com/package/multer-s3
[11] https://sharp.pixelplumbing.com
[12] https://sharp.pixelplumbing.com/performance
[13] https://www.npmjs.com/

```
// Instalação usando o npm
$ npm install <lib-name>

// Instalação usando o yarn
$ yarn add <lib-name>
```

Figura 4: Instalação da *express-image-server*

3.2 Configuração

Para realizar a configuração da ferramenta no projeto em que ela foi instalada, é preciso importar os *middlewares* e a classe Storage que serão utilizados.

A Figura 5 mostra a utilização da *express-image-server* com o DiskStorage, ou seja, o disco rígido será utilizado para armazenar as imagens. Na linha 12 o objeto `diskStorage` é criado, ele espera o parâmetro *dest* que recebe como valor o caminho para um diretório local no qual as imagens serão salvas, nesse caso, é passado o diretório *images*.

Na linha 18, é definida a rota *GET /images/:id* e a ela é acoplado o `processMiddleware`, que recebe no parâmetro *storage* o objeto `diskStorage` que foi criado anteriormente.

Na linha 24, é definida a rota *POST /images* a qual é acoplado o `uploadMiddleware`, que também recebe o `diskStorage` no parâmetro *storage*. Desse modo, ambos os *middlewares* se comunicarão com a mesma base de imagens.

Feito isso, as funcionalidades disponibilizadas pela *express-image-server* estarão disponíveis através das rotas que foram configuradas e ambas se comunicarão com o armazenamento de imagens no disco rígido.

A configuração utilizando a classe S3Storage é similar à DiskStorage, a única diferença será na criação, pois ela precisa receber outros parâmetros específicos e necessários para a comunicação com serviço AWS S3. A Figura 6 exemplifica esse processo.

Nessa figura é possível ver que essa classe Storage requer os seguintes atributos, todos referentes às configurações do S3: o nome do *bucket*; o *accessKeyId*; a *secretAccessKey*; e a *region*. Portanto, é preciso realizar a criação e configuração de um *bucket* na AWS S3 para ter esses dados, o que pode ser feito através da documentação disponibilizada pelo próprio serviço [14].

3.3 Uso do *uploadMiddleware*

Após realizar as configurações da seção anterior, o *uploadMiddleware* estará disponível através da seguinte requisição:

POST http://localhost:3000/images.

A Figura 7 mostra como uma requisição desse tipo pode ser feita através do Postman[15]. No exemplo é enviada a imagem `universe.jpg`, que é vista na Figura 8. Essa imagem é enviada no campo *file* do *body* da requisição, e possui as seguintes propriedades:

[14] https://docs.aws.amazon.com/pt_br/AmazonS3/latest/userguide/creating-buckets-s3.html
[15] https://www.postman.com/

```
import path from 'path'
import express from 'express'

import {
  DiskStorage,
  processMiddleware,
  uploadMiddleware
} from 'express-image-server'

const app = express()

const diskStorage = new DiskStorage({
  dest: path.resolve(__dirname, '..', 'images'),
})

app.get(
  '/images/:id',
  processMiddleware({ storage: diskStorage }),
)

app.post(
  '/images',
  uploadMiddleware({ storage: diskStorage }),
)

app.listen(3000)
```

Figura 5: Configuração com o DiskStorage

```
import { S3Storage, ... } from 'express-image-server'

...

const s3Storage = new S3Storage({
  bucketName: 'bucket-name',
  accessKeyId: 'access-key-id',
  secretAccessKey: 'secret-access-key',
  region: 'region'
})

...

app.listen(3000)
```

Figura 6: Configuração com o S3Storage

- largura: 1080 pixels;
- altura: 1349 pixels;
- formato: JPG;

Ao receber essa requisição, a *express-image-server* enviará a imagem com as propriedades originais acima para a base de dados configurada e ela será salva com o nome original, nesse caso, `universe.jpg`.

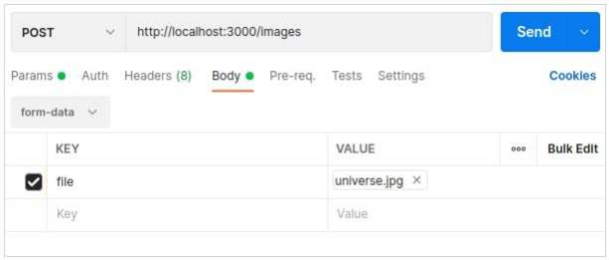

Figura 7: Requisição para o *uploadMiddleware* no Postman

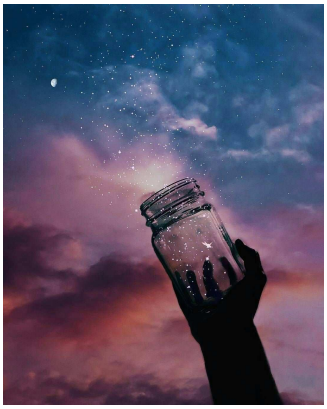

Figura 8: Imagem enviada ao *uploadMiddleware*

3.4 Uso do *processMiddleware*

O *processMiddleware* estará disponível através da seguinte requisição:

GET http://localhost:3000/images/:id.

A Figura 9 exemplifica a utilização desse *middleware* através do Postman. Nessa requisição é buscada a imagem `universe.jpg` com os parâmetros:

- `width`: 200 pixels;
- `height`: 200 pixels;
- `format`: PNG;

Isso significa que a imagem `universe.jpg` será processada dinamicamente e a *express-image-server* gerará uma variante da imagem com as dimensões 200px de largura por 200px de altura e com o formato PNG. Essa imagem variante (ver Figura 10) será retornada como resposta da requisição realizada e será salva na base de dados com o seguinte nome de arquivo:

universe,height-200,width-200.png.

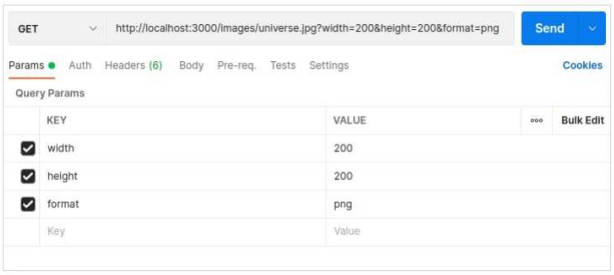

Figura 9: Requisição para o *processMiddleware* no Postman

Figura 10: Imagem gerada pelo *processMiddleware*

3.5 Storage customizado

Como apresentado na Seção 2.2, a *express-image-server* possibilita que o desenvolvedor implemente classes Storage customizadas. Isso é possível através da criação de uma classe que implemente a interface Storage disponibilizada pela *express-image-server*. A Figura 11 mostra o *template* básico para criação de uma classe customizada.

```
import { Storage } from 'express-image-server'

class MyCustomStorage implements Storage {
  async save(id: string, image: Buffer): Promise<boolean> {
    ...
  }

  async fetch(id: string): Promise<Buffer | undefined> {
    ...
  }

  async exists(id: string): Promise<boolean> {
    ...
  }

  getMulterStorage(): StorageEngine {
    ...
  }
}
```

Figura 11: Definição de uma classe Storage customizada

4 AVALIAÇÃO DA FERRAMENTA

Com o objetivo de avaliar o desempenho da *express-image-server*, foram realizados alguns testes que serão descritos ao longo desta seção.

4.1 Testes de tamanho de imagens variantes

O objetivo desta avaliação foi verificar se a *express-image-server* obteria um bom resultado na redução do tamanho de imagens ao gerar uma variante com dimensões menores e formato diferentes da original.

A avaliação considerou quatro imagens com formato PNG de tamanhos 1MB, 5MB, 10MB e 20MB, aproximadamente. E, para cada uma delas, foi gerada uma variante no formato JPG com dimensões 152x152 pixels, que são as dimensões recomendadas para imagem de perfil na rede social Instagram[16]. Esse contexto foi utilizado pois se assemelha a um caso de uso real em que se poderia utilizar a *express-image-server* para reduzir o tamanho de imagens grandes de perfil para exibí-las numa página em tamanho reduzido.

Os resultados obtidos são apresentados na Tabela 2. Através dela é possível concluir que a *express-image-server* obteve um ótimo desempenho, pois para as quatro imagens consideradas houve uma redução superior a 95% no tamanho do arquivo.

Tabela 2: Resultado da geração de imagens variantes

Tamanho original	Tamanho da variante	% de redução
1,1 MB	22,1 kB	98%
5,2 MB	28,7 kB	99.5%
10,5 MB	15,7 kB	99.8%
21,1 MB	15,7 kB	99.9%

4.2 Testes de carga

Esta avaliação consistiu em realizar testes de carga em um servidor utilizando a *express-image-server*, a fim de avaliar quanta carga de requisições a ferramenta suportaria em um contexto real de uso. Para isso foi utilizada a Artillery[17], uma ferramenta para construção e execução de testes de carga para sistemas *web*.

Com a Artillery foram definidos os *scripts* de teste utilizando `yml`. Nesses arquivos são especificados a URL alvo do teste, a duração em segundos, a quantidade de requisições feitas por segundo e o fluxo de ações. Para este trabalho, foram construídos quatro *scripts* de testes, cada um deles considerando um cenário diferente:

- Cenário 1: envio de requisições por uma imagem de 1MB;
- Cenário 2: envio de requisições por uma imagem de 5MB;
- Cenário 3: envio de requisições por uma imagem de 10MB;
- Cenário 4: envio de requisições por uma imagem de 20MB.

Cada cenário descrito teve duração de 60 segundos e, ao longo desse tempo, 10 requisições eram feitas para a URL especificada, totalizando 600 requisições. Os resultados obtidos relacionados ao tempo de resposta são apresentados na Figura 12. Nessa figura, são exibidos a mediana, o percentil 95 e o percentil 99 do tempo de resposta para cada cenário.

[16] https://www.instagram.com/
[17] https://www.artillery.io/

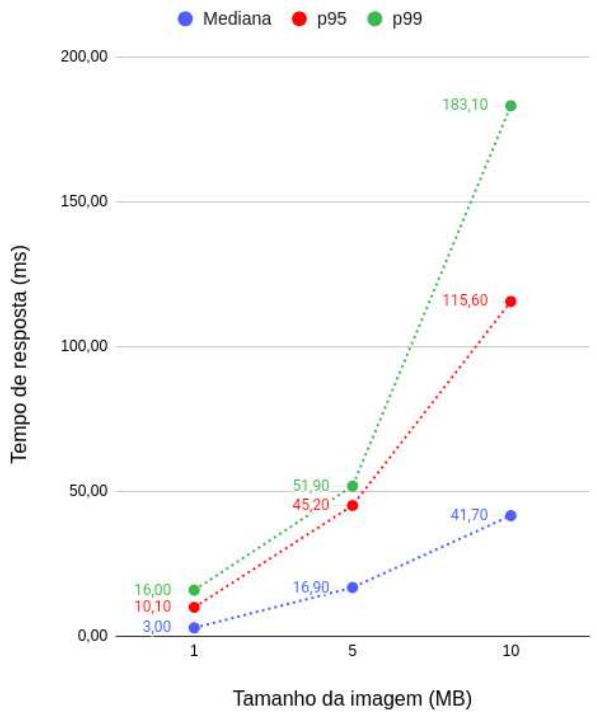

Figura 12: Resultados dos testes de carga

O cenário 4, em que foram realizadas requisições para uma imagem de 20MB, não foi inserido na Figura 12, pois os tempos de resposta obtidos foram muito elevados, o que tornava o gráfico ilegível. Como esperado, esse cenário demonstrou que a utilização do servidor para requisições de imagens grandes não é eficiente, chegando em alguns casos a demorar cerca de 10 segundos para enviar uma resposta. Além disso, para 108 das 600 requisições realizadas o sistema resultou em falha. Os dados de tempo de resposta obtidos nesse cenário foram:

- Mediana: 3828,5 ms;
- p95: 6976,1 ms;
- p99: 8692,8 ms.

4.3 Testes de uso de imagens em páginas HTML

Para avaliar o uso prático da biblioteca em *websites*, foram construídas páginas HTML contendo elementos `img` tendo como fonte uma URL que aponta para uma imagem disponível através de um servidor utilizando a *express-image-server*. Para que a avaliação se aproximasse o máximo possível da realidade, o *deploy* do servidor foi feito através do Heroku[18], uma plataforma de computação em nuvem que possibilita a implantação de aplicações *web*.

Todas as imagens variantes requisitadas pelos elementos `img` foram em relação a uma única imagem chamada `image-sample` com, originalmente, os seguintes atributos:

- tamanho: 10,5 MB;
- formato: PNG;
- altura: 1538 pixels;
- largura: 3984 pixels.

Com essa avaliação buscou-se entender se a *express-image-server*, ao gerar e fornecer variantes das imagens para uma página *web*, estava realmente permitindo que as imagens carregassem mais rapidamente do que as imagens em seu estado original.

4.3.1 *Cenário 1.*

O primeiro cenário consistiu em utilizar a imagem original especificando as dimensões desejadas diretamente nos elementos `img`. A Figura 13 mostra como o HTML da página foi construído.

O resultado obtido foi um tempo de **20,4 segundos** para que as imagens fossem exibidas por completo, o que se configura como um resultado insatisfatório, pois o usuário levaria muito tempo para visualizar todas as imagens por completo na página.

```
<html>
<body>
  <div>
    <img src="https:// ... /images/image-sample.png" width="2400" height="1600">
    <img src="https:// ... /images/image-sample.png" width="1500" height="1000">
    <img src="https:// ... /images/image-sample.png" width="500" height="500">
    <img src="https:// ... /images/image-sample.png" width="200" height="200">
    <img src="https:// ... /images/image-sample.png" width="100" height="100">
    <img src="https:// ... /images/image-sample.png" width="50" height="50">
    <img src="https:// ... /images/image-sample.png" width="10" height="10">
  </div>
</body>
</html>
```

Figura 13: Página com atributos de imagens no elemento *img*

4.3.2 *Cenário 2.*

No segundo cenário, as imagens foram requisitadas utilizando os parâmetros de altura, largura e formato disponibilizados pela *express-image-server* através do parâmetro `src` do elemento `img`. A Figura 14 mostra a construção do HTML deste cenário.

```
<html>
<body>
  <div>
    <img src="https:// ... /images/image-sample.png?width=2400&height=1600&format=jpg">
    <img src="https:// ... /images/image-sample.png?width=1500&height=1000&format=jpg">
    <img src="https:// ... /images/image-sample.png?width=500&height=500&format=jpg">
    <img src="https:// ... /images/image-sample.png?width=200&height=200&format=jpg">
    <img src="https:// ... /images/image-sample.png?width=100&height=100&format=jpg">
    <img src="https:// ... /images/image-sample.png?width=50&height=50&format=jpg">
    <img src="https:// ... /images/image-sample.png?width=10&height=10&format=jpg">
  </div>
</body>
</html>
```

Figura 14: Página com atributos de imagens na URL

Aqui foram construídas duas páginas HTML: uma requisitando as imagens no formato JPG, mostrada na Figura 14; e a outra utilizando a mesma estrutura da figura, mas requisitando as imagens no formato WebP[19]. Para cada imagem das duas páginas foi coletado o

[18] https://dashboard.heroku.com/

[19] https://developers.google.com/speed/webp

tempo de exibição completa em tela. Esses dados são apresentados na Figura 15.

Interpretando a figura e comparando os resultados com os do cenário 1, chega-se à conclusão que requisitar as imagens já com os parâmetros desejados e com um formato mais adequado para a *web* possibilita que as imagens sejam exibidas muito mais rapidamente, trazendo uma performance melhor para o site e, consequentemente, uma experiência melhor para o usuário. Ainda é possível notar que as imagens em formato WebP são exibidas com uma performance melhor que as imagens em formato JPG.

Figura 15: Tempo para exibição completa em tela das imagens

5 EXPERIÊNCIA

5.1 Processo de desenvolvimento

A primeira etapa do processo de desenvolvimento tratou do estudo das soluções existentes de serviço de imagens. Isso foi feito a fim de entender as funcionalidades fornecidas, os componentes arquiteturais utilizados e os requisitos não-funcionais esperados para esse tipo de serviço.

Essa fase possibilitou o estudo de forma mais aprofundada sobre as estratégias utilizadas pelos grandes provedores de serviço de imagens, o que permitiu ter uma visão mais específica sobre em que camada a *express-image-server* poderia atuar.

Tendo por base os materiais gerados na etapa um, a segunda etapa consistiu na elaboração de uma arquitetura base e de estratégias de implementação que poderiam ser utilizadas. Além disso, foi realizado um levantamento de ferramentas e bibliotecas Node.js que forneciam funcionalidades que poderiam ser usadas para simplificar o processo de construção da ferramenta, como a Sharp.js.

Por fim, a terceira etapa consistiu na implementação da ferramenta com base nos requisitos levantados e a arquitetura e estratégias descritas anteriormente. Nessa fase também foram implementados testes unitários para os componentes desenvolvidos, com o intuito de garantir o funcionamento correto de cada um deles.

Como a implementação foi realizada num caráter de descoberta e de experimentação de ferramentas, o código produzido foi armazenado em um repositório privado do GitHub ao longo de todo o processo de desenvolvimento. Esse repositório foi liberado para público e a biblioteca disponibiliza no npm apenas quando se chegou numa versão sólida da ferramenta.

5.2 Desafios

O primeiro desafio encontrado, ainda na fase de pesquisa, foi chegar a uma arquitetura que permitisse que a biblioteca fosse utilizada de forma simples e que pudesse ser acoplada a um servidor já existente, oferecendo a flexibilidade do desenvolvedor customizar a comunicação com a base de dados de seu interesse. Isso foi sendo resolvido na medida em que se foi estudando as ferramentas existentes no mercado, mas também códigos públicos disponíveis na internet que apresentavam uma estrutura que poderia ser aproveitada e implementada na ferramenta.

Outro desafio foi em relação à implementação do S3Storage, pois este necessitaria de uma comunicação com o serviço S3 da AWS, que até o momento do desenvolvimento não tinha sido utilizado pelos autores, o que exigiu a realização de pesquisa para entender como essa comunicação poderia ser feita programaticamente, além de uma organização de estudos e experimentação em código das bibliotecas que viabilizariam essa implementação.

Por último, ao fim do processo de desenvolvimento, a produção de testes de performance também demonstrou ser um desafio, pois não haviam sido realizados anteriormente pelo primeiro autor, o que exigiu uma série de estudos sobre o tema, mas que trouxe um importante aprendizado que será utilizado nas próximas experiências enquanto cientista da computação.

6 TRABALHOS FUTUROS

O objetivo da *express-image-server* é oferecer uma ferramenta gratuita que simplifique a construção de servidor de processamento dinâmico de imagens em Node.js. No entanto, existem melhorias e funcionalidades que precisam ser implementadas no futuro para que a ferramenta traga mais valor e realmente realize o objetivo para o qual foi pensada.

A implementação de uma camada de cache se demonstra essencial, pois isso melhorará a performance da ferramenta e diminuirá o envio de requisições de acesso à base de dados. Tal funcionalidade pode ser implementada utilizando o Redis[20] e permitir que o desenvolvedor especifique se deseja ou não utilizar o recurso.

Atualmente a ferramenta só conta com duas operações de transformação de imagens: redimensionamento e a conversão de formato. No entanto, o ideal é que a biblioteca possa realizar diversas outras operações que são fornecidas pela Sharp.js, como especificação de qualidade da imagem e configuração de posicionamento ao redimensionar. O que pode ser alcançado com a evolução do ImageTransformer e dos componentes que tratam as requisições do *processMiddleware*.

É importante também adicionar opções para otimização de imagem ao realizar o *upload*, que será útil para realizar algum processamento nas imagens antes de serem enviadas para a base de dados.

[20] https://redis.io/

Essas otimizações podem consistir em conversão para um formato específico e compressão sem perda de qualidade.

Ademais, será preciso realizar um trabalho de melhoria no tratamento e comunicação de erros na ferramenta, fornecendo ao cliente mensagens claras e respostas HTTP com estrutura e código de *status* semânticos.

7 AGRADECIMENTOS

Agradeço primeiramente a Deus e à Nossa Senhora por todas as graças dispensadas diariamente ao longo de toda a minha jornada. Agradeço à minha família, por todo os ensinamentos e apoio incondicional. Agradeço aos amigos que estiveram junto a mim ao longo da graduação, com os quais pude crescer e partilhar conhecimentos e conquistas. Agradeço a Nicácio Oliveira, por ter me apresentado o assunto que culminou neste trabalho e por todas as conversas, disponibilidade e mentoria. Por fim, agradeço ao professor João Arthur, por ter aceitado me orientar ao longo do desenvolvimento deste trabalho.

REFERÊNCIAS

[1] [n. d.]. Sobre Node.js®. Retrieved August 3, 2022 from https://nodejs.org/pt-br/about/

[2] [n. d.]. Writing middleware for use in Express apps. Retrieved August 3, 2022 from http://expressjs.com/guide/writing-middleware.html

[3] 2016. Multer Storage Engine. Retrieved August 4, 2022 from https://github.com/expressjs/multer/blob/master/StorageEngine.md

[4] Rafaela Azevedo. 2020. Load Tests: Jmeter vs Artillery. Retrieved August 2, 2022 from https://azevedorafaela.com/2020/06/09/load-tests-jmeter-vs-artillery/l

[5] Lou Bichard. 2019. Node.js Performance Testing and Tuning. Retrieved August 2, 2022 from https://stackify.com/node-js-performance-tuning/

[6] Ayooluwa Isaiah. 2021. A Guide to Load Testing Node.js APIs with Artillery. Retrieved August 2, 2022 from https://blog.appsignal.com/2021/11/10/a-guide-to-load-testing-nodejs-apis-with-artillery.html

[7] Brian Jackson. 2022. How To Optimize Images for Web and Performance. Retrieved August 2, 2022 from https://kinsta.com/blog/optimize-images-for-web/

[8] Keenan Jaenicke. 2019. How To Create a Custom Middleware in Express.js. Retrieved August 2, 2022 from https://www.digitalocean.com/community/tutorials/nodejs-creating-your-own-express-middleware

[9] Krunal Lathiya. 2022. Node Express Image Upload and Resize Guide. Retrieved August 2, 2022 from https://appdividend.com/2022/03/03/node-express-image-upload-and-resize/

[10] Thiago Marinho. 2020. Upload de imagens no S3 da AWS com Node.js. Retrieved August 2, 2022 from https://blog.rocketseat.com.br/upload-de-imagens-no-s3-da-aws-com-node-js/

[11] Rahul Nanwani. 2020. What is an Image CDN - The Complete Guide. Retrieved August 2, 2022 from https://imagekit.io/blog/what-is-image-cdn-guide/

[12] Colin Newcomer. 2022. Cloudinary vs imgix vs ImageKit vs Optimole: Which Is Best for Image Optimization? Retrieved August 2, 2022 from https://themeisle.com/blog/cloudinary-vs-imgix-vs-imagekit-vs-optimole/

[13] Stanley Ulili. 2021. How To Process Images in Node.js With Sharp. Retrieved August 2, 2022 from https://www.digitalocean.com/community/tutorials/how-to-process-images-in-node-js-with-sharp