INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE DAMM
CURSO DE GRADUAÇÂO EM ENGENHARIA MECATRÔNICA

Matheus Zweibrucker Gonçalves

Integração de tecnologias para Aquisição, Persistência, Tratamento e Apresentação de Dados no Contexto de Big Data

FLORIANÓPOLIS, 2024

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE DAMM
CURSO DE GRADUAÇÂO EM ENGENHARIA MECATRÔNICA

Matheus Zweibrucker Gonçalves

Integração de tecnologias para Aquisição, Persistência, Tratamento e Apresentação de Dados no Contexto de Big Data

Trabalho de Conclusão de Curso submetido ao Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina como parte dos requisitos para obtenção do título de Engenheiro Mecatrônico.

Orientador:
Prof. Delcino Picinin Júnior, Dr

FLORIANÓPOLIS, 2024

Ficha de identificação da obra elaborada pelo autor.

Gonçalves, Matheus
Integração de tecnologias para Aquisição, Persistência, **Tratamento e Apresentação de Dados no Contexto de** Big **Data / Matheus Gonçalves; orientação de** Delcino Picinin **Júnior. - Florianópolis, SC,** 2024.
47 p.

Trabalho de Conclusão de Curso (TCC) - Instituto Federal **de Santa Catarina, Câmpus** Florianópolis. Bacharelado **em Engenharia Mecatrônica.** Departamento **Acadêmico de Metal** Mecânica.
Inclui Referências.

1. Big data. 2. Programação. 3. Banco de dados. **4. Software. 5. Tecnologia. I. Picinin Júnior,** Delcino. **II. Instituto Federal de Santa Catarina. III.** Integração **de tecnologias para Aquisição, Persistência,** Tratamento **e Apresentação de Dados no Contexto de Big** Data.

INTEGRAÇÃO DE TECNOLOGIAS PARA AQUISIÇÃO, PERSISTÊNCIA, TRATAMENTO E APRESENTAÇÃO DE DADOS NO CONTEXTO DE BIG DATA

MATHEUS ZWEIBRUCKER GONÇALVES

Este trabalho foi julgado adequado para obtenção do título de Engenheiro em mecatrônica e aprovado na sua forma final pela banca examinadora do Curso Engenharia Mecatrônica do Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina.

Florianópolis, 17 de outubro de 2024.

Banca Examinadora:

gov.br Documento assinado digitalmente
DELCINO PICININ JUNIOR
Data: 23/09/2024 12:52:47-0300
Verifique em https://validar.iti.gov.br

Delcino Picinin Júnior, Dr

gov.br Documento assinado digitalmente
VALDIR NOLL
Data: 23/09/2024 13:23:35-0300
Verifique em https://validar.iti.gov.br

Valdir Noll, Dr

gov.br Documento assinado digitalmente
MAURICIO EDGAR STIVANELLO
Data: 20/09/2024 15:12:11-0300
Verifique em https://validar.iti.gov.br

Maurício Edgar Stivanello, Dr

AGRADECIMENTOS

Em primeiro lugar, quero dar toda a glória a Deus, que me concedeu forças, sabedoria e perseverança ao longo desta jornada. Agradeço ao meu orientador, Delcino Picinin Junior, pela orientação paciente e pela ajuda essencial para a realização deste trabalho. Sou profundamente grato aos meus pais, Enezir e Rosani, por seu amor incondicional, apoio constante e por acreditarem em mim em todos os momentos.Um agradecimento especial à minha irmã, Thaís, que sempre esteve ao meu lado desde a infância, me ajudando e me inspirando a ir cada vez mais longe, também minha namorada, Amanda, que fez parte nos momentos mais precisos para que eu pudesse finalizar com êxito. Agradeço também aos meus amigos e colegas, que compartilharam comigo as alegrias e desafios do curso, tornando essa caminhada mais leve e significativa. Finalmente, agradeço ao IFSC e aos professores que, com dedicação e comprometimento, contribuíram de forma significativa para a minha formação acadêmica e pessoal.

"Esta é a vida eterna: que te conheçam, o único Deus verdadeiro, e a Jesus Cristo, a quem enviaste"
João 17:3

RESUMO

Big Data se refere ao conjunto de dados extremamente grandes e complexos e por isso tem se desenvolvido muitos softwares que apresentam diferentes tecnologias para gerar, tratar e apresentar da melhor maneira possível esse conjunto de dados. Para isso, a programação é a ferramenta essencial para gerenciar todas essas tecnologias devido a sua facilidade de integração de variedade de dados junto com a presença de softwares. O objetivo central do trabalho é estudar e analisar essas tecnologias que mais se ajustam ao Big Data, buscando as principais ferramentas de backend, frontend e banco de dados. Propõe-se, assim, um estudo de caso onde será desenvolvido um protótipo com as três principais ferramentas de backend (flask e node), frontend (react) e banco de dados (MongoDB e PostgreSQL) associados ao gerenciamento de imagens. Nessa perspectiva o estudo dessas ferramentas são de grande importância devido ao cenário tecnológico atual com bilhões de dados gerados e dúvidas ainda emergentes sobre a melhor maneira de gerenciar esses dados.

Palavras-chave: Big Data. Programação. Banco de dados. Software. Tecnologia.

ABSTRACT

Big Data refers to the set of extremely large and complex data, leading to the development of various softwares solutions employing different technologies to effectively manage, process, and present this data. Programming serves as a crucial tool in handling these technologies due to its ability to integrate diverse data types alongside software functionalities. This study aims to explore and analyze technologies best suited for Big Data, focusing on key backend (flask e node), frontend (react), and database (MongoDB e PostgreSQL) tools. A case study approach is proposed, involving the development of a prototype integrating the three main backend, frontend, and database tools for image management. Given the current technological landscape with the generation of billions of data points and ongoing questions about optimal data management practices, understanding these tools is of paramount importance.

Keywords: Big Data. Programming. Database. Software. Technology.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Nos últimos anos, o mundo tem sido impulsionado por uma revolução tecnológica que tem transformado a forma como vivemos e interagimos com as informações. Nesse contexto, um termo tem se destacado cada vez mais: o *Big Data*. Trata-se de um conceito que se refere ao imenso volume de dados estruturados e não estruturados que são gerados diariamente em diferentes fontes, como redes sociais, transações financeiras, dispositivos móveis, sensores, entre outros.

A importância do *Big Data* reside no fato de que esses dados contêm insights valiosos e informações úteis que podem ser explorados para obter vantagens competitivas, tomar decisões estratégicas, compreender comportamentos de consumidores e identificar tendências. Por meio de técnicas e ferramentas analíticas avançadas, é possível extrair significado dessas enormes quantidades de dados, revelando padrões ocultos e obtendo um conhecimento mais profundo sobre diversos aspectos da realidade.

Além disso, o *Big Data* está fortemente ligado à era da transformação digital. Com o advento de tecnologias como a Internet das Coisas (IoT) e a Inteligência Artificial (IA), o volume e a variedade dos dados continuam a crescer exponencialmente. Organizações de diferentes setores, como saúde, varejo, finanças e governo, estão buscando aproveitar esse vasto recurso para otimizar processos, melhorar a tomada de decisões, personalizar produtos e serviços, identificar fraudes, prever demandas e até mesmo solucionar problemas complexos.

No entanto, o desafio do *Big Data* não reside apenas no seu volume, mas também na sua velocidade e variedade. A capacidade de lidar com dados em tempo real, provenientes de diversas fontes e formatos diferentes, exige infraestrutura tecnológica robusta, algoritmos sofisticados e profissionais capacitados. Portanto, a gestão do *Big Data* é um elemento-chave para o sucesso das organizações no mundo moderno.

Nesta era de informação abundante, o *Big Data* se torna um recurso indispensável para a descoberta de insights relevantes e a criação de valor. A sua utilização inteligente e ética promete impulsionar a inovação, melhorar a qualidade de vida das pessoas e impulsionar o progresso em diversas áreas. Diante desse cenário, compreender e aproveitar o potencial do *Big Data* é fundamental para se manter competitivo e relevante no mundo atual.

1.1 Justificativa

O estudo do *Big Data* tornou-se crucial em um mundo cada vez mais conectado e movido por dados. A compreensão dos princípios e técnicas por trás dessa

imensa quantidade de informações permite que profissionais de diversas áreas identifiquem oportunidades, resolvam problemas complexos e tomem decisões mais embasadas. O *Big Data* oferece um potencial imenso para impulsionar a inovação, melhorar a eficiência operacional e transformar modelos de negócios. Além disso, a análise desses dados pode trazer benefícios significativos para a sociedade, como avanços na medicina, previsão de desastres naturais, segurança pública aprimorada e políticas governamentais mais eficazes. Portanto, estudar e compreender o *Big Data* é essencial para acompanhar o ritmo acelerado da transformação digital e se posicionar de maneira estratégica em um mundo cada vez mais baseado em dados.

Com base no que foi dito no parágrafo anterior, o estudo das tecnologias que permeiam o *Big Data* se faz necessário devido ao seu alto grau de relação ao cotidiano de uma pessoa comum. Assim o conhecimento dessas ferramentas da tecnologia é fundamental para compreender melhor o futuro e utilizar estratégias importantes com todos esses dados gerados.

1.2 Definição do Problema

O controle de qualidade é uma etapa essencial em qualquer indústria, garantindo a conformidade dos produtos fabricados com os padrões estabelecidos. Nos últimos anos, a tecnologia de sistemas de visão tem sido amplamente adotada nesse processo, permitindo a análise automatizada de componentes ou peças por meio de imagens capturadas por sistemas de inspeção. No entanto, surge um desafio crucial: o gerenciamento dessas imagens.

O problema reside na ausência de uma solução adequada para organizar, acessar e processar as imagens geradas pelo sistema de visão. A falta de uma plataforma centralizada e intuitiva dificulta a identificação e classificação das peças, bem como a análise posterior dos dados obtidos. Sem um sistema de gerenciamento de imagens, torna-se difícil para os operadores e engenheiros realizar uma triagem precisa entre peças boas e defeituosas, impactando diretamente na qualidade e na eficiência do processo produtivo.

Portanto, a definição clara desse problema é o primeiro passo para a busca de uma solução viável. É imperativo desenvolver uma aplicação que atenda às necessidades específicas da indústria, proporcionando uma interface amigável para o acesso e a organização das imagens, além de ferramentas robustas para análise e classificação automática das peças. Somente assim será possível otimizar o controle de qualidade e impulsionar a eficiência operacional da indústria, garantindo a produção de produtos

de alta qualidade e a satisfação dos clientes.

1.3 Objetivo Geral

O objetivo geral deste trabalho é desenvolver uma aplicação web capaz de oferecer gerenciamento de imagens em processos de controle de qualidade industrial. Através da implementação de um sistema completo, que engloba frontend, backend e banco de dados integrado, almeja-se proporcionar uma plataforma intuitiva e acessível que facilite o acesso, organização e análise das imagens capturadas durante o processo de fabricação. Dessa forma, busca-se promover uma gestão das peças produzidas, contribuindo para a otimização do controle de qualidade e a melhoria contínua do processo produtivo.

1.4 Objetivos Específicos

a) Levantamento de requisitos: Realizar um levantamento detalhado dos requisitos funcionais e não funcionais da aplicação, identificando as necessidades específicas dos usuários e os padrões de qualidade a serem atendidos;

b) Desenvolvimento do frontend: Implementar uma interface de usuário intuitiva e responsiva, utilizando tecnologias modernas de desenvolvimento web, que permita aos usuários acessar e visualizar as imagens de forma rápida e eficiente;

c) Implementação do backend: Desenvolver uma estrutura robusta de backend que seja capaz de processar as requisições do frontend de maneira eficiente e segura, garantindo a integridade e segurança dos dados;

d) Integração com banco de dados: Integrar um banco de dados adequado para armazenar as imagens e os dados relacionados, garantindo a escalabilidade e a eficiência na recuperação das informações;

1.5 Estrutura do Trabalho

O trabalho foi estruturado em fundamentação teórica, metodologia, apresentação de resultados e considerações finais. Com isso teremos uma boa perspectiva do que foi desenvolvido neste trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção, serão abordados os conceitos fundamentais relacionados ao Big Data, tecnologias para aquisição, persistência, tratamento e apresentação de dados também como controle de qualidade industrial e a importancia do auxilio da tecnologia.

2.1 Big Data

2.1.1 Conceito

O termo *Big Data* refere-se a conjuntos de dados extremamente grandes e complexos que não podem ser processados utilizando métodos tradicionais de processamento de dados como planilhas ou até mesmo processos manuais. Esses conjuntos de dados são caracterizados por três principais "Vs": Volume, Velocidade e Variedade.

Volume refere-se à enorme quantidade de dados gerados diariamente por diversas fontes, como transações comerciais, redes sociais, dispositivos móveis, sensores, entre outros. Essa quantidade massiva de dados excede a capacidade das ferramentas tradicionais de processamento e armazenamento.

Velocidade refere-se à taxa de geração e processamento de dados em tempo real. Com a explosão da Internet das Coisas (IoT) e outras fontes de dados em tempo real, os sistemas de Big Data precisam lidar com dados que chegam em alta velocidade e devem ser processados instantaneamente para insights rápidos.

Variedade refere-se à diversidade de tipos de dados que são gerados, incluindo dados estruturados, semi-estruturados e não estruturados. Isso inclui texto, imagens, vídeos, áudio, redes sociais, transações financeiras e muito mais.

2.1.2 Caracteristicas

Além dos três "Vs"principais, o *Big Data* também é caracterizado por outras características importantes

Veracidade refere-se à confiabilidade e precisão dos dados. Como os dados podem ser originados de diversas fontes e em diferentes formatos, é crucial garantir a qualidade e a integridade dos dados para análises precisas.

Valor refere-se à capacidade de extrair insights valiosos e significativos a partir dos dados. O verdadeiro valor do *Big Data* reside na capacidade de transformar

esses dados em informações acionáveis que impulsionam decisões estratégicas e inovação.

Variabilidade refere-se à estrutura e na qualidade dos dados. Os dados podem variar em termos de formatos, qualidade e confiabilidade, exigindo abordagens flexíveis e adaptáveis para lidar com essa variabilidade.

2.1.3 Desafios

Embora o *Big Data* ofereça oportunidades significativas para insights e inovação, também apresenta uma série de desafios.

Armazenamento e processamento: Lidar com o volume massivo de dados requer sistemas de armazenamento e processamento escaláveis e distribuídos, capazes de lidar com grandes cargas de trabalho de forma eficiente.

Segurança e privacidade: Com o aumento da quantidade de dados, surge a preocupação com a segurança e privacidade dos dados. É fundamental garantir a proteção dos dados contra acessos não autorizados e violações de segurança.

Qualidade dos dados: Garantir a qualidade e a integridade dos dados é um desafio, especialmente quando os dados são originados de múltiplas fontes e em diferentes formatos. A limpeza e a integração dos dados são etapas críticas para garantir a confiabilidade das análises.

2.2 Tecnologias

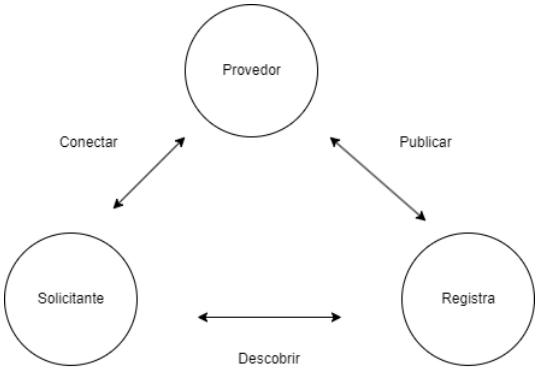
2.2.1 Aquisição

2.2.1.1 Web service

Web services são serviços disponíveis na internet, que não possuem vínculo a nenhuma linguagem de programação, arquitetura ou sistema operacional (PAIK, 2017). Esse serviço pode ser acessado por diversos dispositivos e utiliza XML como padrão de escrita de mensagem. Uma das formas mais utilizadas na implementação é o padrão REST (Transferência de Estado Representacional).

Os serviços são composto por três elementos: o provedor, o registro e o solicitante. Eles se comunicam por meio das funções de publicar, procurar e conectar (Figura 1). Estes serviços são conectados pelo solicitante, implementado pelo provedor e as informações são mantidas no regsitro.

Figura 1 – Adaptado



Fonte: Autor.

O padrão REST é uma arquitetura utilizada para sistemas distribuídos, amplamente utilizado em Web services. Segundo Massé (2012) um Web service RESTful segue os princípios REST. Esses princípios são: cliente-servidor, interface uniforme, sistema por camada, cache, stateless e código sob demanda. Junto a eles se utilizam alguns recursos que podem ser identificados como URIs (Identificador uniforme de recurso), métodos HTTP (Protocolo de Transferência de Hipertexto) utlizados na implementação CRUD (Criar, Ler, Atualizar, Deletar) em formatos JSON ou XML.

O cliente-servidor nos diz a respeito de um repertório de serviços que irá ser solicitado e uma instância cliente que depende do serviço. O sistema de camadas define a hierarquia do sistema e ela é determinante no aspecto de como o sistema irá funcionar, ajudando na latência e segurança. O cache faz referência a autorização pelo cliente e esta tem a responsabilidade de enviar informações para o servidor. O stateless fala a respeito que o servidor não tem controle do estado do cliente e vice-versa, fazendo com que sejam totalmente desacopladas e idependentes.

Entre suas vantagens em relação a outras abordagens é a sua simplicidade, flexibilidade e escalabilidade. Devido a sua criação ter sido direcionada aos padrões web através do HTTP e URIs, eles suportam diversas linguagens e tornaram-se populares entre os desenvolvedores que buscam comunicação eficiente. Para integrar todos esses itens foi definido um padrão semantico que nos direcionam para qual ação está sendo direcionado. Entre eles estão get, post, delete, put, options, head, trace, connect e patch. A Tabela 1 mostra o resumo.

2.2.1.2 Web socket

O WebSocket é uma tecnologia de comunicação bidirecional que resolve o desafio de estabelecer interações em tempo real entre navegadores web e servidores

Tabela 1 – Funções métodos HTTP

Método	Função
get	Solicita a representação de um dado
post	Submete uma entidade a um recurso específico
delete	Remove um recurso.
put	Descreve as opções de comunicações com o recurso de estino do recurso de destino pelos dados da requisição
options	Descreve as opções de comunicações com o recurso de destino.
head	Solicita uma representação dos dados, porém sem conter o body.
trace	Envia uma mensagem por todo o caminho até o destino.
connect	Estabelece uma comunicação de duas vias com o recurso solicitado.
patch	Aplica modificações parciais em um recurso.

Fonte: Mozilla (2021).

de forma eficiente. Enquanto o protocolo HTTP tradicional é adequado para comunicações unidirecionais, como solicitações de páginas da web, ele não é otimizado para atualizações em tempo real, como atualizações de chat, feeds de notícias ao vivo ou jogos multiplayer. O WebSocket surge como uma solução ao permitir uma conexão persistente e bidirecional entre cliente e servidor, possibilitando uma comunicação contínua e de baixa latência.

Com características como conexão persistente e bidirecional, o WebSocket oferece uma comunicação eficiente e em tempo real entre o cliente (navegador) e o servidor. Ao estabelecer uma conexão persistente, o WebSocket elimina a necessidade de solicitações HTTP adicionais para cada interação, reduzindo assim a sobrecarga de rede e os recursos do servidor. Essa eficiência torna o WebSocket uma escolha ideal para aplicações que exigem comunicação em tempo real e alto desempenho.

Além de sua eficiência e simplicidade de implementação, o WebSocket é amplamente suportado em navegadores modernos e oferece compatibilidade com várias linguagens de programação no lado do servidor. Com bibliotecas e frameworks disponíveis para facilitar sua implementação, o WebSocket se torna uma opção acessível e prática para desenvolvedores web que buscam criar aplicações em tempo real interativas e responsivas.

2.2.2 Persistência

2.2.2.1 Banco de dados relacional

Os bancos de dados relacionais são sistemas de gerenciamento de dados baseados em tabelas relacionais, projetados para armazenar, manipular e recuperar informações de forma eficiente e segura. Esse modelo de dados relacional oferece uma estrutura organizada e consistente, facilitando a integração de dados e a manutenção da integridade referencial entre as tabelas.

O modelo relacional se fundamenta em conceitos matemáticos - teoria dos conjuntos e lógica de predicado. De acordo com DATE (2004), o modelo relacional tem três características principais : estrutura de dados, integridade dos dados e manipulação dos dados. Esses mecanismos garantem que as operações realizadas no banco de dados sejam confiáveis e que os dados permaneçam consistentes mesmo em casos de falha do sistema ou interrupções inesperadas.

Com o passar dos anos e com o crescente volume de dados sendo gerados, notou-se uma dificuldade em bancos de dados relacionais que é a sua escalabilidade. Quando utilizado para administrar um grande volume de dados e realizar operações com essas informações, o banco de dados perde perfomance. Com isso é necessário de mais recursos para realizar essas operações caso queira que a consistência dos dados continue.

Conhecido por sua confiabilidade e desenpenho temos o PostgreSQL que é referência por ser robusto e avançado. Ele segue rigorosamente os padrões SQL e oferece divesas funcionalidades que contribuem com sua fama de lidar com problemas complexos.

Sua principal característica é a manipulação de dados JSON, facilitando uma integração eficiente entre o modelo relacional e não relacional de um banco de dados que contém dados semi estruturados. Além disso suas transações são implementadas através da ACID (Atomicidade, Consistência, Isolamento e Durabilidade), garantindo todas as operações que sejam realizadas mesmo em casos de falha ou interrupção.

2.2.2.2 Banco de dados não relacional

Os bancos de dados não relacionais foram criados com uma estrutura de sistemas de gerenciamento de dados com o intuito de lidar com um grande volume

de dados não estruturados ou semiestruturados (LÓSCIO et al., 2011), em contraste com os modelos relacionais. Essas estruturas foram desenvolvidas para atender às demandas de escalabilidade horizontal, flexibilidade de esquema e alta disponibilidade, facilmente encontradas em ambientes de *Big Data*, aplicações web e IoT (Internet das Coisas). Ao contrário dos bancos de dados relacionais, os bancos de dados não relacionais funcionam a partir de modelos de dados flexíveis, como documentos, grafos, colunas ou chave-valor, oferecendo uma alternativa viável para cenários onde a estrutura dos dados pode variar dinamicamente.

Uma das características principais dos bancos de dados não relacionais é sua capacidade de escalar horizontalmente de forma eficiente, distribuindo os dados entre vários servidores e permitindo que o sistema cresça conforme a demanda, sem comprometer o desempenho ou a disponibilidade. Essa abordagem é adequada para aplicações que lidam com grandes volumes de dados distribuídos geograficamente, como redes sociais, sistemas de análise em tempo real e armazenamento de logs.

Os bancos de dados não relacionaios oferecem uma variedade de modelos de dados e estruturas de armazenamento, cada um adaptado a diferentes tipos de dados e casos de uso específicos. Por exemplo, bancos de dados de documentos, como o MongoDB, são ideais para armazenar dados semi-estruturados em formato JSON ou BSON, enquanto bancos de dados de grafos, como o Neo4j, são otimizados para representar relações complexas entre os dados. Essa diversidade de modelos permite aos desenvolvedores escolher a melhor solução para seus requisitos de aplicação e facilita o desenvolvimento de sistemas altamente eficientes.

Entre os bancos não relacionais temos MongoDB, que trabalha com dados semi estruturados e conhecido por sua flexibilidade. Banco de dados orientado a documentos e que arzena seus dados em JSON ou BSON, fator que torna flexível podendo lidar com os dois tipo de dados.

MongoDB também possibilita o escalamento horizontal, que é alcançada através do sharding, uma técnica que divide o banco em fragmentos e que tem a possibilidade de distribuir eles em outros servidores. Com isso o MongoDB consegue gerenciar um grande volkume de dados em várias máquinas, sem precisar comprometer seu desenpenho ou disponibilidade.

2.2.3 Tratamento

2.2.3.1 Node.js

Node.js é um ambiente de execução JavaScript no servidor, que permite aos desenvolvedores utilizar JavaScript tanto no lado do cliente quanto no lado do servidor. Sua arquitetura baseada em eventos e operações de I/O não bloqueantes o que o torna ideal para o desenvolvimento de aplicações web altamente escaláveis e de alta performance. Uma das principais vantagens do Node.js é sua capacidade de lidar com operações intensivas de I/O de forma eficiente, o que o torna adequado para aplicações que requerem comunicação em tempo real (TILKOV; VINOSKI, 2010).

O node.js foi construído a partir do interpretador V8 de Javascript da Google, isto possibilitou uma enorme comunidade envolvida para o desenvolvimento do Node, já que este possui código aberto e gratuito. Uma das características principais é o uso da linguagem Javascript, uma das linguagens mais populares do mundo. Além disso segundo Young (YOUNG *et al.*, 2017) o fato de utilizar o Javscript permite algumas particularidades como :

a) Mesma linguagem para servidor e cliente;

b) Comunicação via Javascript Object Notation (JSON);

c) Possibilidade de utlizar banco de dados NoSQL, já que armazenam JSON;

d) O motor V8 é atualizado junto com o padrão ECMAScript, o que garante o desenvolvimento e evolução mútua;

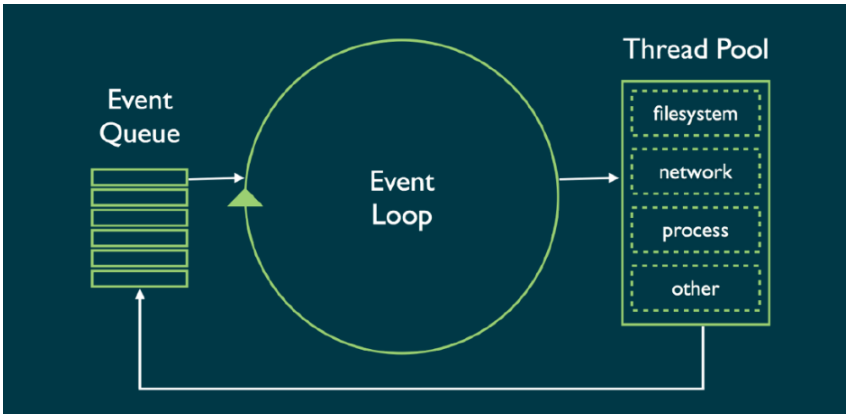
e) Intepretador tem desenpenho definido através dos principios de acesso rápido a propriedade, geração de código dinâmico e coleta de lixo eficiente.

O Node.js é construído a partir de duas divisões de bibliotecas. A primeira, responsável pelo laço de eventos de E/S, de forma rápida para redes ou sistemas de arquivos, como exemplo a libuv. A segunda divisão corresponde a biblioteca Protocolo de Transferência de Hipertexto Seguro (HTTP), na qual estão definidos os métodos de manipulação do seu protocolo (WILSON, 2018).

De acordo com Teixeira (TEIXEIRA, 2012), a disponibilidade de funções de primeira classe e recursos como closures, são alguns dos fatores que tornam o Node uma das ferramentas mais adequadas a orientação a eventos. A forma de processamento dos eventos, conforme explicação de Cantelon et al 2017, é chamado de Event-Loop (EL). O EL, apresentado na Figura 9, é um mecanismo interno, que utiliza de bibliotecas para manipular e gerenciar os eventos, com funcionamento ininterrupto,

onde cada interação verifica os estados dos eventos.

Figura 2 – Adaptado



Fonte: Felipemonobe: evolucao assincrono nodejs.

2.2.3.2 Flask

Flask é um micro-framework de código aberto para desenvolvimento de aplicações web. Ela foi construída em python e particularmente sua arquitetura não segue os padrões de arquitetura como o MVC (Model, View, Control). Sua arquitetura inicia com a escolha de um ORM (Object-Relational Mapping) utilizado para mapear o banco de dados o que difere de outros frameworks que possuem ORM proprios como Django.

Este framework permite ao desenvolvedor escolher exatamente o que utilizar das bibliotecas e outras ferramentas envolvidas. Com isso a aplicação pode ter maior desempenho e a curva de aprendizagem também. Flask surgiu depois de uma brincadeira de "1º de Abril"quando seu criador, Armin Ronacher, publicou um artigo sobre um novo "micro-web-framework"e acabou que a comunidade gostou da ideia, ganhando assim notoriedade.

2.2.4 Apresentação

React é uma biblioteca JavaScript desenvolvida pelo Facebook, amplamente reconhecida e adotada na indústria de desenvolvimento de software. Sua abordagem é inovadora para construção de interfaces de usuário (UI) interativas e responsivas e teve sua primeira versão publicada em 2013. Uma das principais características do React é sua arquitetura baseada em componentes reutilizáveis, que permite aos desenvolvedores dividir a interface do usuário em pequenos componentes. Esses componentes podem facilmente criar interfaces complexas, facilitando a manutenção e a

escalabilidade do código.

Outro aspecto fundamental do React é sua eficiência no gerenciamento do DOM (Document Object Model). O React utiliza uma técnica chamada "Virtual DOM", onde uma representação virtual da árvore de elementos do DOM é mantida em memória. Isso faz com que o React realize atualizações no DOM de forma eficiente, minimizando o número de manipulações diretas no DOM, o que resulta em melhor desempenho e uma experiência de usuário mais fluida.

Além disso, o React é amplamente suportado pela comunidade de desenvolvedores, o que significa que existem uma vasta quantidade de recursos, bibliotecas e ferramentas disponíveis para facilitar o desenvolvimento de aplicações web com React.

3 METODOLOGIA

Este capítulo apresenta o desenvolvimento de uma aplicação para aquisição, persistência, tratamento e apresentação de dados e, para isso será utilizado a metodologia de estudo de caso para comparar diferentes tecnologias e diferentes linguagens de programação.

A fundamentação teórica se baseou na pesquisa bibliográfica, que é um tipo de pesquisa que se origina de outros materiais já criados, composto de livros e artigos científicos.

3.1 Métodos Aplicados

Na construção deste estudo, foi utilizado duas linguagens de programação: Javascript e Python, ambos produzidos no editor VS Code. Para a aquisição dos dados foi utilizado web socket que possui baixa latência e envio em tempo real para o servidor e método HTTP via request realizados. Para a persistência dos dados foi utilizado banco de dados não relacional e que tem maior perfomance em volume de dados e banco de dados relacional robusto de alta confiabilidade. Tratamento de dados se deu por dois backends, dois frameworks escolhidos que possuem grande envolvimento em diversas aplicações hoje me dia e que irão funcionar com protocolos diferentes. Já a apresentação de dados foi desenvolvido uma plataforma no frontend focado na dinâmica de estados.

3.2 Levantamento de requisitos do sistema

A primeira etapa da metodologia utilizada foi levantar os requisitos funcionais do sistema que deveria atender. Com isso poderiamos planejar o desenvolvimento atingindo cada quesito estabelecido. Cada requisito foi estabelecido baseado nos objetivos específicos apresentados no começo deste trabalho.

- Requisitos funcionais

    1. O sistema desenvolvido deve se capaz de realizar uploads de imagens.
    2. O sistema deve apresentar as imagens armazenadas no banco de dados para o usuário.
    3. O sistema deve tratar com lógicas o processamento dessas imagens.
    4. O sistema deve armazenar as imagens em um banco de dados.

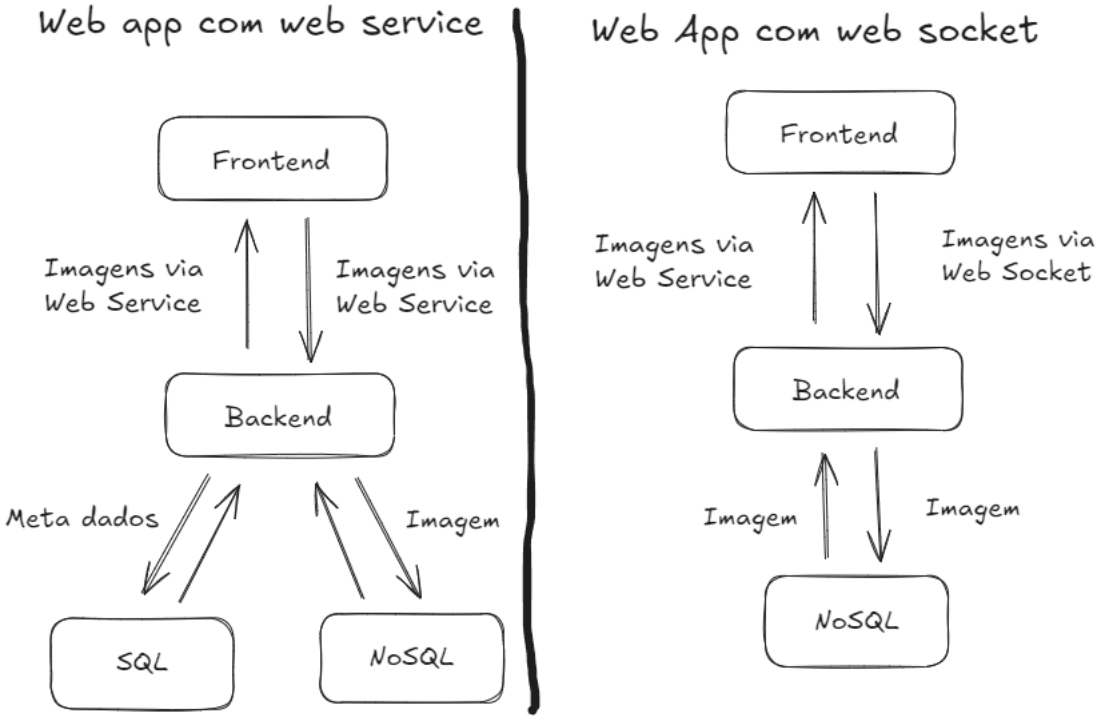
- Requisitos não funcionais

1. O sistema deve armazenar os meta dados das imagens em um banco de dados SQL.
2. O sistema deve correlacionar imagens do banco de dados SQL e os meta dados através de um ID único.
3. O sistema deve conter dois servidores backend em linguagens de programação diferentes.
4. O sistema deve conter dois servidores backends que funcionam com protocoles web diferentes para fazer upload das imagens.
5. O sistema deve converter a imagem em objeto *BLOB*(Binary Large Object)

3.3 Descrição da solução

Com o objetivo de conseguir armazenar e visualizar as imagens, foi desenvolvido uma plataforma web. Essa plataforma utilizou dois servidores web como ponto de partida, começando com o primeiro servidor que faz conexão com frontend através de protocolo web socket e armazena as imagens no banco de dados não relacional (NoSQL), depois temos o servidor que também se conecta com o frontend mas utiliza protocolo de web service, além de armazenar as imagens no banco de dados não relacional (NoSQL) e os metadados no banco de dados relacional (SQL). A baixo podemos ter uma visualização do diagrama do projeto para melhor entendimento:

Figura 3 – Diagrama



Com os requisitos do projeto definido e os estudos relacionados, foi planejado o desenvolvimento em três etapas:

- Desenvolver Frontend
- Desenvolver os Backends
- Orquestrar os Bancos de dados com os servidores

3.3.1 Frontend

Nesta seção, será detalhado o processo de desenvolvimento da interface de usuário do web app, com foco nas bibliotecas utilizadas, na arquitetura de componentização e nos passos seguidos para implementar as funcionalidades relacionadas ao gerenciamento de imagens.

O primeiro passo no desenvolvimento do frontend foi a seleção de um conjunto de bibliotecas que oferecessem as ferramentas necessárias para criar uma interface moderna, responsiva e funcional. As bibliotecas escolhidas foram:

- React: Biblioteca principal para construção da interface de usuário, permitindo a criação de componentes reutilizáveis e uma gestão eficiente do estado da aplicação.
- React-DOM: Utilizada para a integração entre React e o Document Object Model (DOM), facilitando a renderização dos componentes na página web.
- React-Router-DOM: Implementada para gerenciar a navegação entre as diferentes páginas do web app, garantindo uma experiência de usuário fluida.
- Axios: Biblioteca usada para realizar requisições HTTP ao backend, simplificando a comunicação e o gerenciamento de dados.
- React-Toastify: Adicionada para exibir notificações e mensagens ao usuário, proporcionando feedback sobre as ações realizadas, como uploads de arquivos e atualizações de página.
- TailwindCSS: Framework de CSS utilizado para estilizar os componentes, permitindo a criação de um layout responsivo e visualmente atraente com maior rapidez.

O frontend foi desenvolvido com uma forte ênfase na componentização, uma abordagem que permite dividir a interface do usuário em módulos independentes e reutilizáveis. Essa estratégia foi aplicada na construção do dashboard principal do web

app, que exibe as imagens gerenciadas pelo usuário. O dashboard foi projetado como o núcleo do sistema, centralizando as principais operações de visualização e gestão de imagens. Isso incluiu a exibição das imagens carregadas, a atualização da lista de imagens e a realização de novas inserções. A figura abaixo mostra o código do componente dashboard:

Figura 4 – Componente Dashboard

```
return (
  <div className="shadow-lg sm:rounded-lg flex flex-col bg-white p-3 max-h-full pb-10 my-4 mx-4
  overflow-y-auto h-full">
    {headerComponent}
    <div className="overflow-y-auto h-full">
      {contentComponent}
    </div>
    {buttonComponent}
    <nav className="flex items-center justify-between pt-4 px-6" aria-label="Table navigation">
      <span className="text-sm font-normal text-gray-500">Showing {showPages()} of <span
      className="font-semibold text-gray-900">{totalItems}</span></span>
      <ul className="inline-flex -space-x-px text-sm h-8">
        <li>
          <a href="#" className="flex items-center justify-center px-3 h-8 ml-0 leading-tight
          text-gray-500 bg-white border border-gray-300 rounded-l-lg hover:bg-gray-100
          hover:text-gray-700 " onClick={previousPage}>Previous</a>
        </li>
        {pages()}
        <li>
          <a href="#" className="flex items-center justify-center px-3 h-8 leading-tight
          text-gray-500 bg-white border border-gray-300 rounded-r-lg hover:bg-gray-100
          hover:text-gray-700" onClick={nextPage}>Next</a>
        </li>
      </ul>
    </nav>
  </div>
)
```

No dashboard, foram implementados dois botões principais para facilitar as operações do usuário: o botão Atualizar e o botão Upload. O botão de Atualizar foi projetado para permitir que o usuário recarregue a página e atualize a lista de imagens exibidas, garantindo que qualquer alteração realizada no backend seja imediatamente refletida na interface. Já o botão de Upload, ao ser acionado, abre um modal que permite ao usuário selecionar novos arquivos para upload. Com isso nossa tela principal fica resposiva e dinamica, cumprindo seus requisitos de apresentação e gerenciamento.

Figura 5 – Componente botão

```
export default function Buttons({text, modal, isOpen}) {
  return (
    <div className="flex items-center p-6 space-x-2 border-t border-gray-200 rounded-b justify-between z-0">
      <button onClick={() => window.location.reload()} className="bg-blue-300 hover:bg-blue-400 focus:ring-4 focus:outline-non
      focus:ring-gray-200 rounded-lg border border-gray-200 text-sm font-medium px-5 py-2.5 h-12">{text[0]}</button>
      <div className="flex justify-between">
      {
        Complexity is 3 Everything is cool!
        modal.map((button) => {
          const {isOpen, setModalOpen} = button.props
          return (
            <button key={button.key} onClick={setModalOpen} className="bg-green-400 hover:bg-green-500 focus:ring-4
            focus:outline-none focus:ring-gray-200 rounded-lg border border-gray-200 text-sm font-medium px-5 py-2.5
            hover:text-gray-900 focus:z-10 h-12">{button.key}</button>
          )
        })
      }
      { isOpen &&  modal.filter((button) => button.props.isOpen == true) }
      </div>
    </div>
  )
}
```

Após a realização do componente principal, foi a vez de construir um novo componente que pudesse receber os arquivos de imagem e que possuisse um botão para enviar para backend nossos arquivos.

O componente criado foi o modal Upload que foi desenvolvido para suportar a inserção de um ou vários arquivos de imagem simultaneamente. Além disso, o modal oferece um campo onde o usuário pode inserir um nome para cada imagem, personalizando a inserção de acordo com suas necessidades como mostra o trecho do código abaixo:

Figura 6 – Componente modal

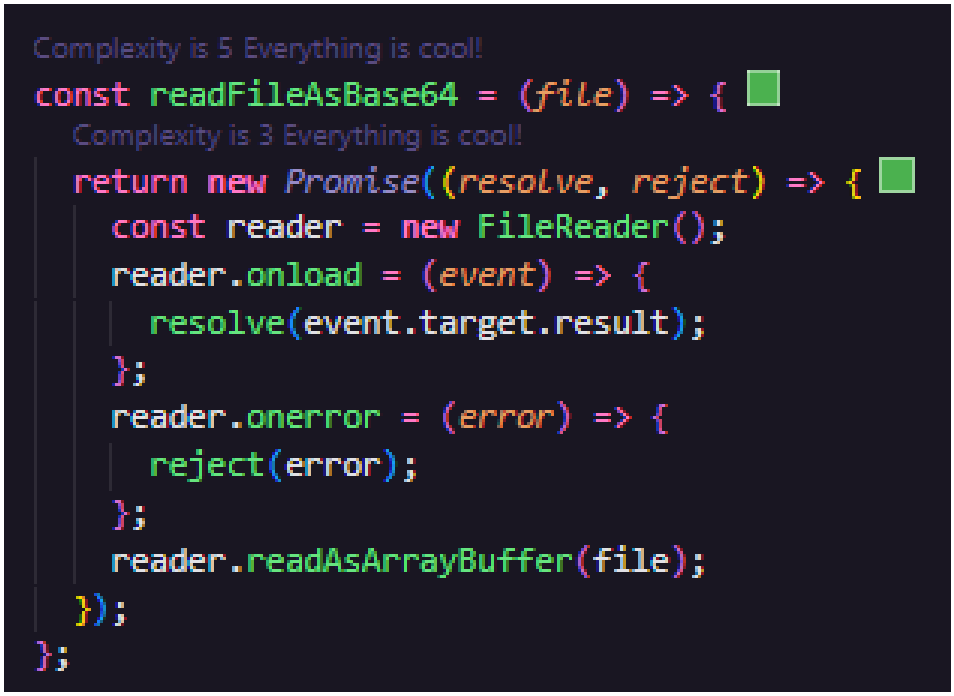
```
return (
  <>
    <div className='flex w-full flex-col'>
      <div className='flex items-center gap-4 w-56 pb-6 pl-6'>
        <label htmlFor="small-input" className="block text-base font-medium text-gray-900">Nome</label>
        <input type="text" id="small-input" className="bg-gray-50 border border-gray-300 text-gray-900 text-sm rounded-lg
        w-full p-2.5" value={name} onChange={handleValue} />
      </div>

      <div className="relative overflow-x-auto">
        <div className="flex items-center justify-center w-full p-4">
          <label htmlFor="dropzone-file" className="flex flex-col items-center justify-center w-full h-64 border-2
          border-gray-300 border-dashed rounded-lg cursor-pointer bg-gray-50 hover:bg-gray-100">
            <div className="flex flex-col items-center justify-center pt-5 pb-6">
              <CloudArrowUp size={50} />
              <p className="mb-2 text-sm text-gray-500"><span className="font-semibold">Clique para fazer upload ou</span>
              Arraste o arquivo aqui</p>
              <p className="text-xs text-gray-500">PNG ou JPG</p>
            </div>
            <input id='dropzone-file' type='file' onChange={handleFileChange} required className="hidden" multiple />
          </label>
        </div>
      </div>

      <div className="flex items-center pt-6 space-x-2 border-t border-gray-200 rounded-b justify-between">
        <button type="button" className="text-black bg-green-400 hover:bg-green-500 focus:ring-4 focus:outline-none
        focus:ring-gray-200 rounded-lg border border-gray-200 text-sm font-medium px-5 py-2.5 hover:text-gray-900 focus:z-
        onClick={handleConsultReq}>Upload</button>
      </div>
    </div>
  </>
);
}
```

Para enviar as imagens e os dados associados ao backend, foi utilizado o método HTTP POST por meio da biblioteca Axios e web socket. Esse método foi escolhido por ser ideal para o envio de dados ao servidor, garantindo que as imagens fossem transferidas de maneira segura e eficiente. Um dos passos cruciais nesse processo é a leitura das imagens no formato Base64. Essa função é essencial porque o formato Base64 permite que os arquivos sejam convertidos em uma string de texto, facilitando a manipulação e o transporte dos dados, especialmente quando se lida com a transmissão via WebSocket. Este método é prático porque encapsula os dados da imagem em um formato que pode ser facilmente manipulado e transportado sem a necessidade de transformações complexas. A função abaixo realiza essa leitura do arquivo:

Figura 7 – Função ler arquivo



Além disso, para o envio das imagens, duas abordagens foram utilizadas, dependendo da tecnologia empregada no backend. Quando a comunicação é feita via WebSocket, o arquivo é simplesmente enviado como está, beneficiando-se da conexão contínua estabelecida entre o cliente e o servidor. Esse método é direto e eficiente, já que permite a transmissão de imagens uma por uma, sem a necessidade de carregar o sistema com grandes quantidades de dados de uma só vez.

Por outro lado, na abordagem via HTTP, que é utilizada com a biblioteca Axios, as imagens são enviadas como BLOBs (Binary Large Objects) encapsulados dentro de um FormData. Utilizar BLOBs é fundamental aqui porque eles permitem o envio de arquivos binários (como imagens) de forma eficiente e segura. Ao encapsular as imagens em BLOBs, garantimos que os dados sejam enviados na sua forma binária original, sem perdas de qualidade, o que é essencial para preservar a integridade dos arquivos durante a transmissão. O uso do FormData, por sua vez, permite organizar esses BLOBs junto com outros dados (como nomes e datas) em uma única requisição, simplificando o processo de upload e garantindo que todas as informações necessárias cheguem ao backend de maneira estruturada. A figura 8 abaixo mostra essa função:

Figura 8 – Função método HTTP

```
Complexity is 6 It's time to do something...
const handleConsultReq = async () => {
  if (name === "") {
    toast.error("Por favor coloque um nome no filtro");
    return;
  }
  try {
    const formData = new FormData();
    files.forEach((fileObj, index) => {
      const blob = new Blob([fileObj.data], { type: fileObj.type });
      formData.append('files', blob, fileObj.name);
    });

    await imagesApi.import(formData);

    console.log("Enviando imagens");
    toast.success("Imagens enviadas com sucesso");
    setModalOpen(false);

  } catch (error) {
    console.error('Error sending images:', error);
    toast.error("Erro ao enviar imagens");
  }
};
```

Figura 9 – Função Web Socket

```
Complexity is 7 It's time to do something...
const handleConsultWS = async () => {
  if (name === "") {
    toast.error("Por favor coloque um nome no filtro");
    return;
  }

  if (!socketInitialized) {
    console.error('Socket is not initialized');
    return;
  }

  try {
    socketRef.current.emit('image', files);
    console.log("Enviando imagens");
    toast.success("Imagens enviadas com sucesso");
    setModalOpen(false);

  } catch (error) {
    console.error('Error sending images:', error);
    toast.error("Erro ao enviar imagens");
  }
};
```

Após a conclusão do upload, o sistema foi programado para atualizar automaticamente o dashboard, proporcionando ao usuário uma visão imediata das novas imagens carregadas. Esse fluxo contínuo e eficiente entre a leitura, o envio e a atualização garante uma experiência de usuário fluida e responsiva, independentemente da tecnologia de backend utilizada.

Essa abordagem de arquitetura garantiu que o frontend fosse desenvolvido de maneira modular, eficiente e responsiva, atendendo plenamente aos requisitos do projeto. A utilização de tecnologias modernas e a adoção de práticas de desenvolvimento bem estabelecidas permitiram a criação de um web app intuitivo, funcional e capaz de oferecer uma experiência de usuário de alta qualidade.

3.3.2 Backend

O desenvolvimento do backend do web app envolveu a criação de dois servidores distintos, cada um implementado com diferentes tecnologias, com o objetivo de realizar uma comparação entre eles. O primeiro servidor foi desenvolvido utilizando Flask, um microframework em Python, enquanto o segundo servidor foi construído com Node.js, um ambiente de execução em JavaScript. Ambos os servidores foram projetados para lidar com o armazenamento e gerenciamento de imagens, mas diferem na forma como se comunicam com o frontend e nos bancos de dados que utilizam.

3.3.2.1 Servidor Flask

O primeiro servidor foi desenvolvido utilizando Flask, uma ferramenta leve e flexível, ideal para a construção de APIs rápidas e eficientes. Este servidor foi configurado para se comunicar com o frontend através de WebSocket, uma tecnologia que permite a troca de dados em tempo real entre o servidor e o cliente. Essa escolha foi feita para explorar as vantagens do WebSocket em aplicações onde a comunicação bidirecional contínua é essencial, como no caso do gerenciamento de uploads de imagens, onde é importante fornecer feedback imediato ao usuário.

Para implementar o servidor Flask, foram utilizadas as seguintes bibliotecas:

- Flask: O núcleo do servidor, responsável por definir as rotas, processar requisições e responder ao frontend.
- flask-cors: Utilizada para permitir o compartilhamento de recursos entre diferentes origens (CORS), o que é fundamental para a comunicação segura entre o frontend (hospedado em uma origem) e o backend Flask (hospedado em outra).
- flask-socketio: Esta biblioteca adiciona suporte a WebSocket no Flask, possibilitando a comunicação em tempo real entre o servidor e o cliente.
- pymongo: Biblioteca utilizada para a interação com o MongoDB, que é o banco de dados responsável por armazenar as imagens enviadas pelo frontend.
- Werkzeug: Ferramenta que fornece várias utilidades para WSGI, como segurança, roteamento e depuração, e que é utilizada pelo Flask como o seu servidor web..

O fluxo de trabalho do servidor Flask é o seguinte: quando uma imagem é enviada a partir do frontend, ela é recebida pelo servidor via WebSocket. O servidor, então, processa essa imagem e a armazena no MongoDB. O MongoDB foi escolhido por sua capacidade de armazenar grandes volumes de dados não estruturados, como

imagens, de maneira eficiente. Essa arquitetura permite que as imagens sejam facilmente recuperadas e gerenciadas pelo sistema. A seguir iremos ver o fluxo das funções que são executadas para manipular e salvar esses dados enviados.

O fluxo de trabalho no Flask começa com a conexão via WebSocket, que é estabelecida para permitir a comunicação em tempo real entre o cliente e o servidor. Esta conexão contínua é essencial para transmitir as imagens de forma eficiente, especialmente em cenários onde o upload de múltiplas imagens pode ocorrer simultaneamente.

Figura 10 – Conexão Web Socket - Flask

```python
@socketio.on('image')
def handle_image(binary_data):
    try:
        save_images_to_db(binary_data)
        emit('imageSaved', {"message": "Image saved successfully"})
    except Exception as e:
        emit('error', {"error": "Internal server error"})
    finally:
        disconnect()
```

Fonte: Autor.

Após a conexão ser estabelecida, a próxima etapa é a chamada da função responsável por salvar as imagens recebidas. Esta função processa cada imagem enviada através do WebSocket, garantindo que sejam armazenadas corretamente no banco de dados MongoDB. O uso de WebSocket permite que as imagens sejam enviadas uma por uma, assegurando que cada arquivo seja processado e armazenado sem sobrecarregar o servidor ou o cliente com um grande volume de dados de uma só vez.

Figura 11 – Função salvar imagens - Flask

```python
def save_images_to_db(binary_images):
    try:
        insert_results = collection.insert_many([{"data": binary} for binary in binary_images])
        return insert_results.inserted_ids
    except Exception as e:
        print('Error saving images to MongoDB:', e)
        raise e
```

Fonte: Autor.

Além da comunicação via WebSocket, o servidor Flask também mapeia rotas HTTP para facilitar a interação com o frontend em diferentes cenários. Duas rotas

principais foram implementadas: uma utilizando o método POST e outra utilizando o método GET.

Figura 12 – Rotas - Flask

```python
@app.route('/images', methods=['GET'])
def get_images():
    try:
        images = get_images_from_db()
        return jsonify(images)
    except Exception as e:
        return jsonify({"error": "Failed to fetch images from database"}), 500

@app.route('/images', methods=['POST'])
def post_images():
    try:
        files = request.files.getlist('files')
        binary_data = [file.read() for file in files]
        save_images_to_db(binary_data)
        return jsonify({"message": "Images saved successfully"})
    except Exception as e:
        return jsonify({"error": "Internal server error"}), 500
```

Fonte: Autor.

- A rota mapeada com o método POST é utilizada para receber imagens do frontend via HTTP. Quando esta rota é chamada, as imagens são enviadas ao servidor, processadas, e então armazenadas no MongoDB. Este método é uma alternativa à comunicação via WebSocket, oferecendo uma forma mais tradicional de upload de arquivos, especialmente útil em casos onde o WebSocket não é necessário ou não está disponível.
- A rota mapeada com o método GET serve para recuperar as imagens armazenadas no servidor. Quando o frontend faz uma solicitação GET a esta rota, o servidor Flask busca as imagens no MongoDB e as envia de volta ao cliente. Isso permite que as imagens sejam exibidas no dashboard do web app, proporcionando ao usuário acesso fácil e rápido aos arquivos armazenados.

Figura 13 – Função pegar imagens - Flask

```python
def get_images_from_db():
    try:
        images = list(collection.find())
        return [{"_id": str(image["_id"]), "data": base64.b64encode(image["data"]).decode('utf-8')} for image
        in images]
    except Exception as e:
        print('Error fetching images from MongoDB:', e)
        raise e
```

Fonte: Autor.

Essa combinação de WebSocket para comunicação em tempo real e rotas HTTP para interações mais tradicionais oferece flexibilidade e robustez ao servidor

Flask, garantindo que ele possa lidar com diferentes tipos de requisições e cenários de uso. Seja enviando imagens uma a uma através do WebSocket ou utilizando o método POST para enviar dados por HTTP, o servidor está preparado para armazenar e recuperar as imagens de maneira eficiente, mantendo a integridade dos dados e a responsividade do sistema.

3.3.2.2 Servidor Node.js

Paralelamente ao desenvolvimento do servidor Flask, foi implementado um segundo servidor utilizando Node.js. Este servidor, em contraste, utiliza HTTP como protocolo de comunicação com o frontend, explorando um método mais tradicional e amplamente utilizado para a transferência de dados na web. A escolha de Node.js foi motivada por sua natureza não bloqueante e escalável, o que o torna uma excelente opção para aplicações que precisam lidar com um grande número de requisições simultâneas.

O servidor Node.js foi construído utilizando as seguintes bibliotecas:

- Axios: Biblioteca utilizada para realizar requisições HTTP, facilitando a comunicação entre o servidor e o frontend.
- body-parser: Middleware utilizado para processar dados enviados no corpo das requisições HTTP, como os arquivos de imagem e informações associadas.
- cors: Utilizado para habilitar o compartilhamento de recursos entre diferentes origens, semelhante à biblioteca flask-cors no servidor Flask.
- express: Framework web minimalista e flexível que simplifica a criação de rotas e o gerenciamento de requisições e respostas no servidor Node.js.
- mongodb: Biblioteca usada para interagir com o MongoDB, onde as imagens recebidas são armazenadas.
- multer: Middleware usado para lidar com uploads de arquivos em aplicações Node.js, facilitando o processamento e armazenamento de imagens enviadas pelo frontend.
- pg: Biblioteca para interagir com o banco de dados PostgreSQL, onde são armazenados os metadados das imagens, como o nome e a data de inserção.

No fluxo de trabalho do servidor Node.js, quando o usuário faz o upload de uma imagem, esta é enviada ao servidor via HTTP. O servidor utiliza o multer para

processar e armazenar a imagem no MongoDB. Em seguida, os metadados associados à imagem, como o nome e a data de inserção, são salvos em um banco de dados PostgreSQL. A escolha de utilizar dois bancos de dados distintos—MongoDB para as imagens e PostgreSQL para os metadados—foi feita para aproveitar as forças de cada um: o MongoDB para o armazenamento eficiente de arquivos grandes e não estruturados, e o PostgreSQL para o gerenciamento de dados relacionais e estruturados, com o benefício de suas capacidades de consulta avançadas.

A primeira função tem como responsabilidade o salvamento das imagens no banco de dados. Quando uma imagem é enviada pelo usuário através do frontend, essa função é chamada para processar o arquivo e armazená-lo corretamente no banco de dados. O processo começa com a recepção do arquivo de imagem já convertido para um formato adequado para o armazenamento, como um um documento BSON no caso do MongoDB.

Figura 14 – Função salvar imagens - Node

```
Complexity is 5 Everything is cool!
async function saveImagesToDB(binaryImages) {
  try {
    const insertPromises = binaryImages.map((binaryData) => {
      collection.insertOne({ imageId: binaryData.id, data: binaryData.data })
  });
    await Promise.all(insertPromises);
    console.log('Images saved to MongoDB');
  } catch (error) {
    console.error('Error saving images to MongoDB:', error);
    throw error; // Rethrow error to handle it in the route handler
  }
}
```

Fonte: Autor.

A segunda função é responsável por recuperar as imagens armazenadas no banco de dados e enviá-las ao frontend para exibição. Quando o frontend faz uma solicitação para exibir as imagens, essa função é chamada para buscar os arquivos no banco de dados. A função consulta o banco de dados e retorna todas as imagens associadas em um JSON. Uma vez enviadas, o frontend recebe e converte para o formato correto, possibilitando a exibição das imagens.

Figura 15 – Função pegar imagens - Node

```
Complexity is 6 It's time to do something...
async function getImagesFromDB() {
  try {
    const images = await collection.find().toArray();
    return images.map(image => ({
      _id: image._id.toString(), // Convert ObjectId to string
      data: image.data.toString('base64') // Convert binary data to base64 string
    }));
  } catch (error) {
    console.error('Error fetching images from MongoDB:', error);
    throw error; // Rethrow error to handle it in the route handler
  }
}
```

Fonte: Autor.

Para integrar essas funções ao sistema, foram mapeadas duas rotas HTTP principais: uma para o método POST e outra para o método GET.

- A rota POST é utilizada para receber as imagens enviadas pelo frontend. Quando o usuário faz o upload de uma imagem, essa rota é acionada, e a imagem é passada para a função de salvamento descrita anteriormente. A função processa e armazena a imagem no banco de dados, e a rota responde com uma confirmação de que o upload foi bem-sucedido.
- A rota GET é usada para recuperar e enviar as imagens ao frontend. Quando o frontend solicita a exibição de imagens, essa rota é acionada, e a função de recuperação é chamada para buscar as imagens no banco de dados. As imagens são então enviadas de volta ao frontend através dessa rota, permitindo que o usuário visualize os arquivos armazenados.

Figura 16 – Rota GET - Node

```
Complexity is 3 Everything is cool!
app.get('/images', async (req, res) => {
  try {
    const images = await getImagesFromDB();
    res.json(images);
  } catch (error) {
    console.error('Failed to fetch images from database:', error);
    res.status(500).json({ error: 'Failed to fetch images from database' });
  }
});
```

Fonte: Autor.

Figura 17 – Rota POST - Node

```
Complexity is 8 It's time to do something...
app.post('/images', upload.array('files'), async (req, res) => {
  const pgClient = await pgPool.connect(); // Conectar ao PostgreSQL
  try {
    const files = req.files;
    if (!files || files.length === 0) {
      return res.status(400).send('No files uploaded.');
    }

    // Log para cada arquivo recebido
    files.forEach(file => {
      console.log(`Received file: ${file.originalname}`);
    });

    // Array para armazenar as imagens processadas
    let binaryImages = [];

    // Loop assíncrono para processar cada arquivo
    for (const file of files) {
      const uuid = uuidv4();

      const result = await pgClient.query( // Usar await para garantir que a operação seja concluída
        'INSERT INTO images (id, name, date) VALUES ($1, $2, $3) RETURNING id',
        [uuid, file.originalname, new Date()]
      );

      const imageId = result.rows[0].id; // Obter o ID retornado
      // Adicionar a imagem e seu ID ao array
      binaryImages.push({
        id: imageId,
        data: file.buffer
      });
    }
```

Fonte: Autor.

Essas duas rotas trabalham em conjunto para garantir que o fluxo de upload e recuperação de imagens seja contínuo e eficiente, proporcionando uma experiência de usuário fluida e confiável. A implementação dessas funções e rotas é fundamental para o funcionamento do sistema, permitindo que as imagens sejam manipuladas de maneira segura e eficiente, desde o upload até a exibição no frontend.

3.3.3 Banco de dados

A escolha e o uso de bancos de dados desempenham um papel crucial na eficiência e funcionalidade de qualquer aplicação web. No desenvolvimento do backend do web app, optou-se por utilizar dois sistemas de gerenciamento de banco de dados distintos: o MongoDB e o PostgreSQL. Esses bancos foram selecionados com base em suas características específicas e na capacidade de atender aos requisitos do projeto, tanto em termos de armazenamento quanto de manipulação de dados.

3.3.3.1 MongoDB

O MongoDB foi utilizado tanto na aplicação desenvolvida em Flask quanto na aplicação desenvolvida em Node.js. O MongoDB é um banco de dados NoSQL que armazena dados em documentos do tipo JSON (JavaScript Object Notation), permitindo uma estrutura de dados flexível e escalável. Esta característica o torna especialmente adequado para o armazenamento de imagens e outros tipos de dados não estruturados ou semi-estruturados.

A escolha do MongoDB no projeto, especialmente na aplicação Flask, foi motivada pela necessidade de uma solução rápida e eficiente para o armazenamento de arquivos de imagem. Como o MongoDB não impõe esquemas rígidos, ele oferece uma abordagem mais ágil, permitindo que as imagens sejam armazenadas de maneira direta e sem a necessidade de transformações complexas. Essa flexibilidade é particularmente vantajosa em projetos onde os requisitos de dados podem evoluir ao longo do tempo, pois o MongoDB facilita a adaptação a novas necessidades sem grandes mudanças na estrutura do banco.

Além disso, o MongoDB foi integrado à aplicação Flask, que utiliza WebSocket para comunicação em tempo real com o frontend. Essa combinação foi escolhida para explorar a rapidez e a simplicidade do MongoDB na inserção de dados, proporcionando uma experiência de upload de imagens fluida e responsiva para o usuário.

Na aplicação Node.js, o MongoDB também desempenha um papel central, mas em um contexto diferente. Aqui, ele é usado para armazenar as imagens enviadas pelo frontend, aproveitando sua capacidade de lidar com grandes volumes de dados binários. Essa escolha se alinha à necessidade de um armazenamento eficiente de arquivos, enquanto o PostgreSQL foi introduzido para gerenciar os metadados relacionados às imagens.

Figura 18 – Configuração MongoDB

```
app = Flask(__name__)
CORS(app)
socketio = SocketIO(app, cors_allowed_origins="http://localhost:5173")

# MongoDB setup
mongoURI = "mongodb://localhost:27017"
dbName = 'imagesDB'
collectionName = 'images'

client = MongoClient(mongoURI)
db = client[dbName]
collection = db[collectionName]
```

Fonte: Autor.

3.3.3.2 PostgreSQL

O PostgreSQL foi utilizado exclusivamente na aplicação Node.js, em conjunto com o MongoDB. Este banco de dados relacional é conhecido por sua robustez, confiabilidade e suporte a operações complexas, como consultas avançadas, transações, e relacionamentos entre tabelas. A escolha do PostgreSQL foi motivada pela necessidade de armazenar e gerenciar metadados das imagens de forma estruturada e relacional.

No contexto do projeto, o PostgreSQL foi utilizado para armazenar informações como o nome das imagens e a data de inserção, associadas aos arquivos armazenados no MongoDB. Essa combinação de MongoDB e PostgreSQL permite que o sistema aproveite o melhor dos dois mundos: a flexibilidade e eficiência no armazenamento de grandes volumes de dados não estruturados no MongoDB, juntamente com a robustez e capacidade de gerenciamento de dados estruturados no PostgreSQL.

Essa arquitetura de banco de dados híbrida foi escolhida para permitir a correlação entre dados não estruturados e estruturados de forma eficaz. Por exemplo, enquanto o MongoDB armazena os arquivos de imagem em si, o PostgreSQL armazena informações adicionais que podem ser facilmente consultadas e relacionadas, como histórico de uploads, registros de usuário, ou outras informações contextuais que possam ser necessárias para expandir as funcionalidades do sistema no futuro.

Ao optar por essa combinação de bancos de dados, o projeto ganhou a capacidade de escalar e adaptar-se a diferentes cenários de uso. O PostgreSQL não só facilita a organização e recuperação de dados estruturados, mas também possibilita a futura inclusão de novos tipos de dados ou relações entre eles, sem comprometer o desempenho ou a integridade do sistema.

Figura 19 – Configuração PostgreSQL

```
// MongoDB setup
const mongoURI = "mongodb://localhost:27017";
const dbName = 'imagesDB';
const collectionName = 'images';

let client;
let collection;

Complexity is 3 Everything is cool!
async function connectToMongoDB() {
  try {
    client = new MongoClient(mongoURI, { useNewUrlParser: true, useUnifiedTopology: true });
    await client.connect();
    const db = client.db(dbName);
    collection = db.collection(collectionName);
    console.log('Connected to MongoDB');
  } catch (error) {
    console.error('Error connecting to MongoDB:', error);
    process.exit(1); // Exit process on connection error
  }
}

const pgPool = new Pool({
  user: 'postgres',
  host: 'localhost',
  database: 'postgres',
  password: 'matheus2024',
  port: 5432,
});
```

Fonte: Autor.

4 APRESENTAÇÃO DOS RESULTADOS

Nesta seção, são apresentados os resultados obtidos após a implementação completa e o teste das duas abordagens de backend desenvolvidas para o web app. Serão exibidas imagens do web app em funcionamento, demonstrando como as funcionalidades de gerenciamento de imagens foram realizadas utilizando tanto o Flask quanto o Node.js.

Após a conclusão dos testes, foi observado que ambas as tecnologias – Flask e Node.js – se comportaram de maneira muito semelhante em termos de desempenho e funcionalidade geral. As duas abordagens foram capazes de atender aos requisitos do projeto, gerenciando o armazenamento e a manipulação de imagens com eficiência. No entanto, cada tecnologia apresentou suas particularidades, que influenciaram na experiência de desenvolvimento e na forma como os dados foram processados.

4.1 Análise e discussão dos resultados

Uma das vantagens do Flask foi sua curva de aprendizagem, que se mostrou tão acessível quanto a do JavaScript, utilizado no desenvolvimento com Node.js. Isso facilitou o desenvolvimento rápido e eficaz do backend com Flask, especialmente para aqueles já familiarizados com a sintaxe e conceitos de JavaScript. A semelhança nas abordagens de programação permitiu uma transição suave entre as duas tecnologias, sem grandes desafios ou dificuldades para o desenvolvedor envolvido no projeto.

Apesar da semelhança no desempenho, uma limitação significativa foi identificada no backend desenvolvido com Node.js, relacionada ao método de transferência de dados via HTTP. Durante os testes, foi constatado que o navegador Google Chrome, assim como outros browsers, impõe um limite máximo de 4GB para o envio de dados em uma única requisição HTTP. Esse limite apresentou uma barreira ao tentar enviar grandes volumes de imagens de uma só vez, o que poderia ser um problema em cenários onde é necessário fazer o upload de múltiplas imagens simultaneamente.

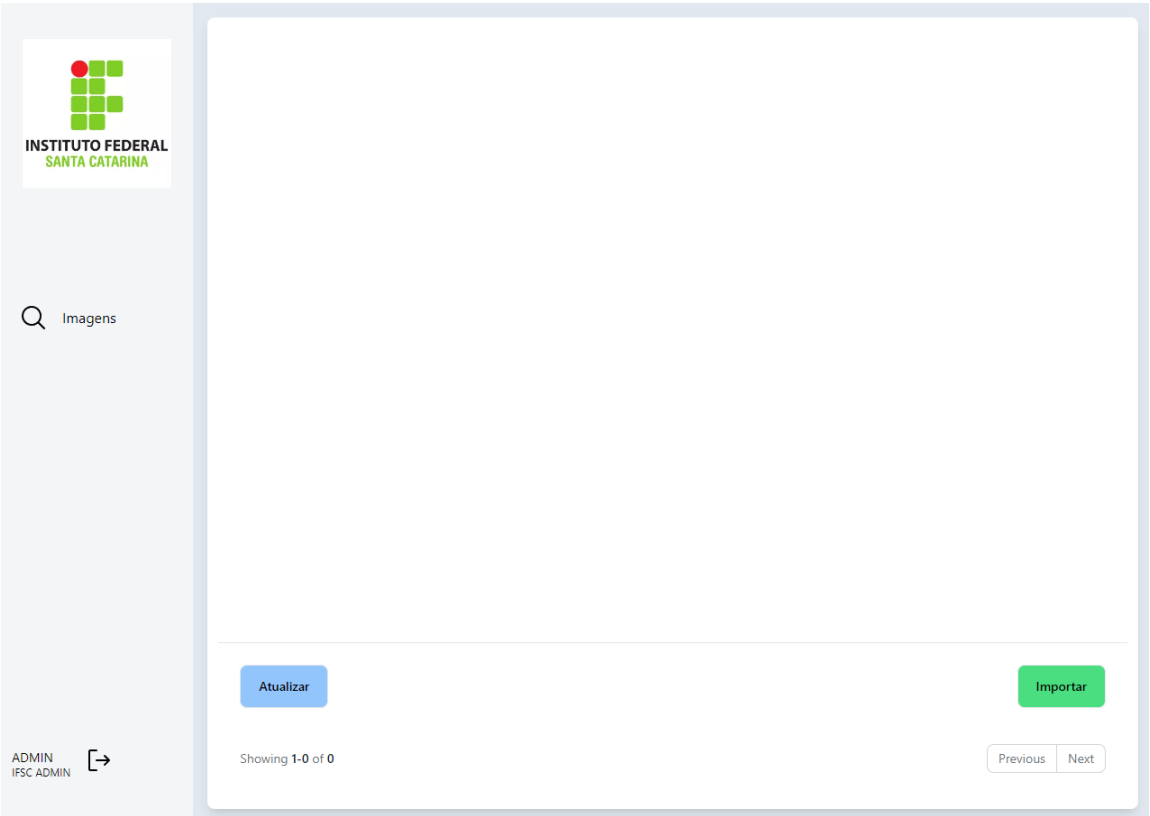
Em contraste, o Flask, utilizando WebSocket para a comunicação com o frontend, demonstrou uma vantagem significativa neste aspecto. Diferente do método HTTP, que tenta enviar o payload completo de uma só vez, o WebSocket estabelece uma conexão mútua contínua entre o cliente e o servidor. Isso permite que as imagens sejam enviadas uma a uma, mesmo quando o usuário seleciona todas as imagens ao mesmo tempo. Esse método de envio contínuo e fracionado evita os problemas

de limite de tamanho enfrentados pelo HTTP, garantindo que todas as imagens sejam transferidas sem interrupções, independentemente do tamanho total dos arquivos.

Portanto, embora ambas as tecnologias tenham se mostrado adequadas e eficientes para o desenvolvimento do web app, a escolha entre Flask e Node.js pode depender das necessidades específicas do projeto. Se o cenário envolve o upload de grandes volumes de dados, a abordagem com Flask utilizando WebSocket pode oferecer uma solução mais robusta e flexível. No entanto, para aplicações onde o limite de tamanho de dados não é um fator crítico, o Node.js continua sendo uma opção sólida e confiável.

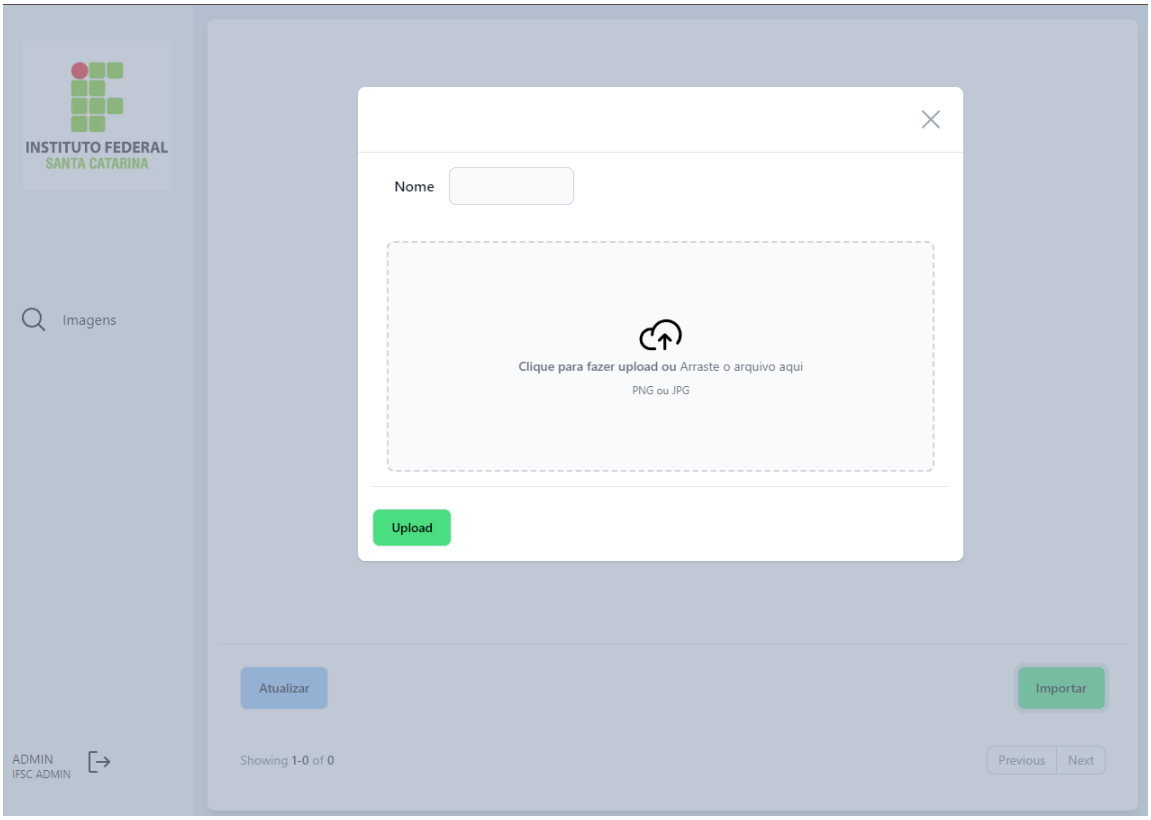
As imagens a seguir ilustram o web app em operação, mostrando as interfaces de upload e gerenciamento de imagens.

Figura 20 – Tela inicial



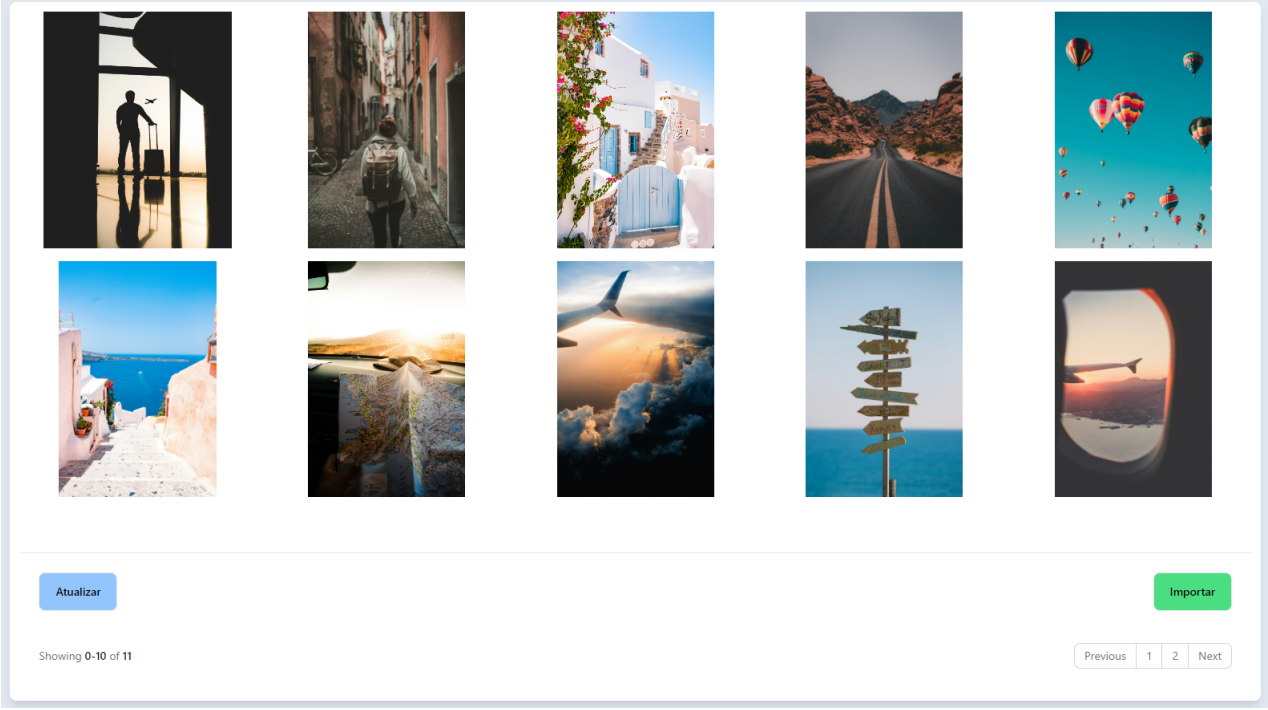
Fonte: Autor.

Figura 21 – modal



Fonte: Autor.

Figura 22 – Galeria



Fonte: Autor.

5 CONSIDERAÇÕES FINAIS

Em suma, este trabalho apresentou uma abordagem prática para o desenvolvimento de uma aplicação web utilizando tecnologias modernas no contexto de *Big Data*. Através do estudo de caso realizado, foi possível constatar a eficácia das tecnologias escolhidas na implementação de um gerenciador de imagens para controle de qualidade em uma indústria.

Embora tenha sido alcançado sucesso na implementação da aplicação, é importante ressaltar que ainda existem desafios a serem superados, como a otimização do desempenho e a garantia da segurança dos dados. Sugere-se que trabalhos futuros explorem esses aspectos e investiguem novas tecnologias e abordagens para o desenvolvimento de aplicações semelhantes.

REFERÊNCIAS

COUTINHO, P. C. Rest tutorial. *DevMedia*, 2013. Acesso em: 11 jul. 2024, Disponível em: https://www.devmedia.com.br/rest-tutorial/28912.

DATE, C. *Introdução a sistemas de bancos de dados*. ELSEVIER EDITORA, 2004. ISBN 9788535212730. Disponível em: https://books.google.com.br/books?id=xBeO9LSlK7UC.

GOTTSCHALK, K. D. *et al.* Introduction to web services architecture. *IBM Systems Journal*, v. 41, n. 2, p. 170–177, 2002.

GURUGE, A. *Web Services: Theory and Practice*. 1. ed. [S.l.]: Digital Press, 2004.

HALILI, F.; RAMADANI, E. Web services: a comparison of soap and rest services. *Modern Applied Science*, v. 12, n. 3, p. 175, 2018.

MASSE, M. *REST API design rulebook: designing consistent RESTful web service interfaces*. [S.l.]: O'Reilly Media, Inc., 2011.

MUMBAIKAR, S.; PADIYA, P. Web services based on soap and rest principles. In: *International Journal of Scientific and Research Publications*. [S.l.: s.n.], 2013. v. 3.

PAIK, H.-Y. *Web service implementation and composition techniques*. [S.l.]: Springer International Publishing, 2017. 15

RICHARDSON, L.; RUBY, S. *RESTful web services*. [S.l.]: O'Reilly Media, Inc., 2008.

TEIXEIRA, P. *Professional Node. js: Building Javascript based scalable software*. [S.l.]: John Wiley & Sons, 2012. 20

TILKOV, S.; VINOSKI, S. Using javascript to build high-performance network programs. *IEEE Internet Computing*, v. 14, n. 6, p. 80–83, 2010. Acesso em: 23 jul. 2024, Disponível em: https://ieeexplore.ieee.org/abstract/document/5617064. 20

VLADUTU, A. *Mastering Web Application Development with Express*. [S.l.]: Packt Publishing Ltd, 2014.

WILSON, J. *Node.js 8 the Right Way: Practical, Server-side Javascript that Scales*. [S.l.]: Pragmatic Bookshelf, 2018. 20

YOUNG, A. *et al. Node.js in Action*. Manning, 2017. ISBN 9781617292576. Disponível em: https://books.google.com.br/books?id=YzfuvQAACAAJ. 20