

**CENTRO UNIVERSITÁRIO PARA O DESENVOLVIMENTO DO ALTO VALE DO
ITAJAÍ - UNIDAVI**

WELSON DOUGLAS OLIVEIRA GOMES

API DE INTEGRAÇÃO PARA WEBSERVICE DE ACADEMIA

RIO DO SUL

2024

**CENTRO UNIVERSITÁRIO PARA O DESENVOLVIMENTO DO ALTO VALE DO
ITAJAÍ - UNIDAVI**

WELSON DOUGLAS OLIVEIRA GOMES

API DE INTEGRAÇÃO PARA WEBSERVICE DE ACADEMIA

Trabalho de Conclusão de Curso a ser apresentado ao curso de Sistemas da Informação, da Área das Ciências Naturais, da Computação e das Engenharias, do Centro Universitário para o Desenvolvimento do Alto Vale do Itajaí, como condição parcial para a obtenção do grau de Bacharel em Sistemas de Informação.

Prof. Orientador: Cleber Nardelli

RIO DO SUL

2024

**CENTRO UNIVERSITÁRIO PARA O DESENVOLVIMENTO DO ALTO VALE DO
ITAJAÍ - UNIDAVI**

WELSON DOUGLAS OLIVEIRA GOMES

API DE INTEGRAÇÃO PARA WEBSERVICE DE ACADEMIA

Trabalho de Conclusão de Curso a ser apresentado ao curso de Sistemas da Informação, da Área das Ciências Naturais, da Computação e das Engenharias, do Centro Universitário para o Desenvolvimento do Alto Vale do Itajaí- UNIDAVI, a ser apreciado pela Banca Examinadora, formada por:

Professor Orientador Cleber Nardelli

Banca Examinadora

Professor M.e Fernando Andrade Bastos

Professor M.e Jullian Hermann Creutzberg

Rio do Sul, 21 de novembro de 2024.

“A tecnologia é apenas uma ferramenta. No que diz respeito a motivar as pessoas e a torná-las produtivas, o professor é o mais importante.”

Bill Gates.

Dedico este trabalho de conclusão de curso primeiramente a Deus, por me guiar e fortalecer em todos os momentos dessa jornada. À minha família, pelo amor, apoio e por sempre acreditarem no meu potencial. Aos meus amigos, pelo incentivo e compreensão em momentos de dificuldade. Aos colegas de aula, pela parceria e trocas de conhecimento que enriqueceram minha trajetória acadêmica. Aos professores, pela dedicação e pelo compartilhamento de seu valioso conhecimento. E, a UNIDAVI, por proporcionar um ambiente de aprendizado e crescimento. A todos, minha sincera gratidão.

AGRADECIMENTOS

Gostaria de expressar minha profunda gratidão primeiro a Deus, por ter me guiado e fortalecido ao longo de toda esta jornada acadêmica nestes quatro anos. Agradeço à minha família, cujo amor e apoio incondicional foram fundamentais para a realização desta graduação. Aos meus amigos, por seu incentivo constante e compreensão em momentos de dificuldade e principalmente aos meus padrinhos Moises e Thais, os quais me incentivaram muito a voltar aos estudos e terminar à graduação. Aos colegas de aula, pela parceria e enriquecedoras trocas de conhecimento que tornaram essa caminhada mais leve e produtiva. Aos professores, pela dedicação e por compartilharem seu valioso conhecimento, que foi essencial para o meu crescimento profissional. À UNIDAVI, por proporcionar um ambiente propício ao aprendizado e ao desenvolvimento. A todos, meu mais sincero agradecimento.

RESUMO

Milhares de pessoas pensando no seu bem-estar ou até mesmo sendo um *hobby*, praticam exercícios físicos diariamente em uma ou mais academias. Diante deste contexto, o presente trabalho de conclusão de curso, apresenta uma proposta com base em pesquisa e desenvolvimento de um projeto *webservice* voltado para a manutenção e gerenciamento de dados e informações de academias, tendo este como finalidade principal, melhorar os processos e facilitar a vida diária de quem trabalha ou frequenta uma academia. A metodologia utilizada para realização da pesquisa foi de pesquisa aplicada e descritiva. Visando alcançar os objetivos da pesquisa, foi feita uma revisão da literatura, onde são abordados temas sobre as principais tecnologias como, linguagens de programação, engenharia de software, banco de dados, além de ferramentas que facilitam o processo e outras metodologias que englobam o desenvolvimento do software. Na metodologia da pesquisa, é apresentado um fluxo de como a pesquisa e o projeto foi elaborado bem como também uma prévia da aplicação que consome os *endpoints*. No capítulo de desenvolvimento, é detalhado toda a análise do projeto, contendo, entendimento do processo de negócio, regras de negócio, levantamento de requisitos funcionais, não funcionais, fluxogramas, diagramas, requisições e testes. Por fim, foram feitos testes com os *endpoints* da API, utilizando como ferramenta o postman, onde foram desenvolvidos *scripts* de testes para as validar as requisições REST que o *webservice* trabalha.

Palavras-Chave: *webservice*, academia, sistemas de informação.

ABSTRACT

Thousands of people, whether focused on their well-being or even as a hobby, practice physical exercises daily in one or more gyms. With this context in mind, this final course project presents a proposal based on research and development of a web service project aimed at maintaining and managing data and information for gyms, with the goal of improving processes and making daily life easier for those who work at or attend a gym. The methodology used for conducting the research was applied and descriptive research. To achieve the research objectives, a literature review was conducted, covering topics on the main technologies such as programming languages, software engineering, databases, as well as tools that facilitate the process and other methodologies that encompass software development. In the research methodology, a flowchart is presented, showing how the research and project were developed, as well as a preview of the application that consumes the endpoints. In the development chapter, the entire project analysis is detailed, including business process understanding, business rules, gathering of functional and non-functional requirements, flowcharts, diagrams, requests, and tests. Finally, tests were conducted on the API endpoints using Postman as a tool, where test scripts were developed to validate the REST requests that the web service operates on.

Keywords: web service, gym, information systems.

LISTA DE FIGURAS

Figura 1 - Relacionamento de tabelas.....	27
Figura 2 - Modelo de apresentação do <i>Swagger</i>	32
Figura 3 - Fluxograma Processo	34
Figura 4 - Fluxograma completo da API.....	38
Figura 5 - Requisição de <i>login</i> da API.....	38
Figura 6 - Requisição de <i>login</i> da API com falha.....	39
Figura 7 - Requisição de cadastro de aluno da API	40
Figura 8 - Diagrama de sequência de <i>login</i> na API	43
Figura 9 - Diagrama de sequência para rota de cadastro de pessoa na API	43
Figura 10 - Diagrama de sequência para cadastro de um novo exercício	44
Figura 11 - Diagrama de sequência para cadastro de um novo treino.....	44
Figura 12 - Modelagem de dados da API.....	45
Figura 13 - Estrutura principal do projeto	46
Figura 14 - Classe <i>middleware</i>	46
Figura 15 - Código da classe de <i>login</i>	47
Figura 16 - Classe de rotas da API	48
Figura 17 - Classe controladora de cadastro de professor.....	49
Figura 18 - Classe de processamento para cadastro de pessoa (admin, professor e aluno)	50
Figura 19 - Classe controladora de cadastro de exercício	51
Figura 20 - Classe de processamento para cadastro de exercício.....	51
Figura 21 - Classe controladora de cadastro de treino.....	51
Figura 22 - Classe de processamento para cadastro de treino	52
Figura 23 - Estrutura das classes de processamento de dados de pessoa	52
Figura 24 - Teste de <i>login</i> - Validação de JSON de retorno.....	53
Figura 25 - Teste de <i>login</i> - Validação do <i>token</i> JWT.....	54
Figura 26 - Teste de <i>login</i> - Validação de retorno não autorizado.....	54
Figura 27 - Teste de sucesso para cadastro de professor.	55
Figura 28 - Teste de <i>status</i> 400 no cadastro de professor	55
Figura 29 - Teste de não autorizado no cadastro de professor.....	56
Figura 30 - Teste de sucesso para cadastro de aluno	56
Figura 31 - Teste de não autorizado para cadastro de aluno	57

Figura 32 - Teste de cadastro de exercício	57
Figura 33 - Teste de não autorizado para cadastro de exercício	58
Figura 34 - Teste de cadastro de treino	58
Figura 35 - Teste de não autorizado para cadastro de treino.....	59

LISTA DE QUADROS

Quadro 1 - Linguagens mais usadas em abril de 2024	20
Quadro 2 - Exemplo de código Javascript.....	21
Quadro 3 - Resultado da execução do código da Quadro 2	22
Quadro 4 - Recursos do PostgreSQL.....	28
Quadro 5 - Funções do gerenciador do banco de dados	28
Quadro 6 - Comandos sql e suas funções	29
Quadro 7 - Explicações das funções dos métodos	31
Quadro 8 - Regras de negócio	40
Quadro 9 - Requisitos funcionais	41
Quadro 10 - Requisitos não funcionais	42

LISTA DE SIGLAS E ABREVIATURAS

API	<i>Application Programming Interface</i>
CMMI	<i>Capability Maturity Model Integration</i>
HTTP	<i>Hypertext Transfer Protocol</i>
IBM	<i>International Business Machines Corporation</i>
IEEE	<i>Institute of Electrical and Electronic Engineers</i>
JSON	<i>Javascript object notation</i>
JWT	<i>JSON Web Token</i>
MDN	<i>Mozilla Developer Network</i>
NPM	<i>Node Package Manager</i>
REST	<i>Representational State Transfer</i>
SEQUEL	<i>Structured query language</i>
SGBD	<i>Data Base Management System</i>
SQL	<i>Structured Query Language</i>
UNIDAVI	Centro Universitário para o Desenvolvimento do Alto Vale do Itajaí
URL	<i>Uniform Resource Locator</i>
XML	<i>Extensible Markup Language</i>
YAML	<i>Yin ´t Markup Language</i>

SUMÁRIO

1 INTRODUÇÃO	15
1.1 PROBLEMA DE PESQUISA.....	16
1.2 Objetivos.....	16
1.2.1 Geral	16
1.2.2 Específicos	16
1.3 JUSTIFICATIVA.....	17
2 REVISÃO DA LITERATURA	18
2.1 ENGENHARIA DE SOFTWARE	18
2.1.1 Desenvolvimento de software	18
2.1.2 Levantamento de requisitos	19
2.2 LINGUAGEM DE PROGRAMAÇÃO.....	19
2.2.1 Javascript	21
2.2.2 NodeJs	22
2.2.3 TypeScript	23
2.2.4 Express	23
2.2.5 Mapeamento Objeto-Relacional – PRISMA ORM	24
2.2.6 Npm	24
2.3 BANCO DE DADOS	25
2.3.1 Banco de dados relacional	26
2.3.2 PostgreSQL	27
2.3.3 Gerenciador de banco de dados SGBD	28
2.3.4 Linguagem SQL	29
2.4 APPLICATION PROGRAMMING INTERFACE (API)	30
2.4.1 Rest	30
2.4.2 Métodos	31
2.4.3 OpenAPI	31
2.4.4 Swagger	32
2.4.5 Insomnia	33
2.4.6 Postman	33
3 METODOLOGIA DA PESQUISA	34
3.1 CLIENTE DE CONSUMO DOS <i>ENDPOINTS</i>	35
4 API DE INTEGRAÇÃO PARA <i>WEBSERVICE</i> DE ACADEMIA	37
4.1 ANÁLISE.....	37
4.2 VISÃO GERAL.....	37
4.3 REGRAS DE NEGÓCIOS	40
4.4 REQUISITOS.....	41

4.4.1 Funcionais	41
4.4.2 Não funcionais	42
4.5 FUNCIONAMENTO DA API	42
4.6 MODELAGEM DE DADOS.....	45
4.7 DESENVOLVIMENTO	45
4.8 TESTES COM POSTMAN.....	53
4.8.1 Teste de <i>LOGIN</i>	53
4.8.2 Teste de professor	55
4.8.3 Teste de aluno	56
4.8.4 Teste de exercício	57
4.8.5 Teste de treino	58
5 CONSIDERAÇÕES FINAIS	60
5.1 RECOMENDAÇÕES DE ATUALIZAÇÕES FUTURAS	61
REFERÊNCIAS	62

1 INTRODUÇÃO

A transformação digital tem impactado positivamente diversos setores, proporcionando inovações que otimizam processos e facilitam a vida das pessoas e da sociedade. Pensando no ambiente de academias, é fundamental uma administração eficaz e unificada para garantir o êxito e a eficiência dos serviços oferecidos pelo estabelecimento.

Algumas academias enfrentam vários obstáculos particulares ou específicos no que diz respeito ao auxílio tecnológico aos professores ou instrutores, assim como no suporte, controle e supervisão dos treinamentos dos alunos, o que mostra a importância da criação de soluções tecnológicas personalizadas para suprir estas demandas.

Por outro lado, os últimos anos acarretaram grandes avanços da tecnologia, a exemplos temos os sistemas integrados, sistemas distribuídos, transmissão de dados pela internet e dos equipamentos de hardware, possibilitando a criação de arquiteturas modernas e robustas. Essas inovações permitem uma comunicação rápida e segura entre os sistemas, tornando viável a implementação de tecnologias sofisticadas.

Nesse contexto, a construção da camada de *backend* para atender requisições vindas da aplicação *frontend*, mostram-se extremamente benéfica, pois neste método de processo, a aplicação terá uma escalabilidade e flexibilidade maior, tornando a aplicação de fácil integração com outros sistemas como por exemplo sistemas de pagamentos ou de ponto, o que permitirá ao usuário operacionalizar o negócio, oferecendo uma solução completa e eficiente para a gestão da academia.

Sendo assim, a escolha de tal tema surge para suprir a demanda de diversas academias, remediando deficiências referentes a gestão de tarefas, como controlar acessos, gravar, editar, apagar e selecionar dados em banco de dados e manter a aplicação com a maior disponibilidade possível, gerenciando todas as exceções geradas por ela, escalando o serviço conforme a demanda de solicitações que chegam ao serviço.

Portanto o presente trabalho tem como objetivo principal implementar a camada de *backend* para atender as requisições vinda do sistema web (*frontend*) e aplicativo móvel, com arquitetura de API, para gerenciamento completo, desenvolvido com as tecnologias modernas, visando atender todas as necessidades e demandas geradas por uma academia.

Além do capítulo introdutório, esta pesquisa apresenta no segundo capítulo a revisão de literatura, onde são abordadas as tecnologias utilizadas no desenvolvimento do projeto, trazendo os conceitos sobre cada um dos pontos utilizados. Já o terceiro capítulo, apresenta a metodologia da pesquisa, detalhando o processo metodológico adotado. No quarto capítulo, é abordado o desenvolvimento da aplicação, trazendo uma análise detalhada de todo o projeto, desde os requisitos até os testes. E no capítulo final, são apresentadas as considerações finais.

1.1 PROBLEMA DE PESQUISA

Como melhorar o processamento, persistência e disponibilidade segura de dados, gerados e utilizados por academias?

1.2 OBJETIVOS

1.2.1 Geral

- Criar uma *Application Programming Interface* – API escalável para gerenciar o processamento, persistência e a segurança dos dados, atendendo as requisições cliente;

1.2.2 Específicos

- Descrever os principais requisitos funcionais e não funcionais;
- Modelar o banco de dados da aplicação para PostgreSQL;
- Implementar API, utilizando as tecnologias NodeJs, TypeScript, Express e PRISMA;
- Avaliar a eficácia da aplicação através de testes dos *endpoints*;

1.3 JUSTIFICATIVA

As academias desempenham um papel crucial no bem-estar e na saúde de muitas pessoas que as frequentam diariamente, proporcionando não apenas um ambiente adequado para a prática de exercícios físicos, mas também um espaço de socialização e motivação. Com uma variedade de equipamentos e aulas, como musculação, pilates, yoga e atividades aeróbicas, as academias atendem a diferentes objetivos de saúde, desde o condicionamento físico, a perda de peso, o fortalecimento muscular até a reabilitação de lesões.

Atualmente, muitas academias ainda não tem um sistema que auxilie nas atividades diárias. Diante disso, propõe-se o desenvolvimento de uma API de gerenciamento com NodeJs, projetada para receber requisições de uma aplicação *frontend*, permitindo que os usuários acessem através de autenticação. A iniciativa da API, visa manter uma estrutura flexível e escalável que possa atender sem interrupções de forma ágil e segura, os dados gerados pela aplicação que estiver consumindo os *endpoints* disponibilizados na API. Esta API, processará os dados das atividades geradas por instrutores, alunos, funcionários e quaisquer aplicações externa que possa ter comunicação com a API.

A API buscará atender todas as requisições que chegarem aos seus *endpoints* disponibilizados, sejam elas para gerar um cadastro, alterar, deletar ou mesmo apenas consultar um dado persistido na base, processando esses dados de forma assíncrona, garantindo para o *frontend* a segurança dos dados.

2 REVISÃO DA LITERATURA

Este capítulo apresenta assuntos pertinentes ao desenvolvimento de software, engenharia de software, linguagens de programação, *frameworks* e bibliotecas que foram empregados na elaboração do projeto. Tais elementos serão abordados detalhadamente no contexto do trabalho objetivando o bom entendimento do leitor.

2.1 ENGENHARIA DE SOFTWARE

Sommerville (2011) afirma que a engenharia de software é uma disciplina que se preocupa com todos os aspectos da produção de software, desde os estágios iniciais de especificação do sistema até a manutenção desse sistema, após a sua entrega. Ela envolve a aplicação de princípios de engenharia para o desenvolvimento de software de forma sistemática, eficiente e confiável, utilizando abordagens e técnicas baseadas em evidências.

A engenharia de software ainda de acordo com Sommerville (2011), também aborda questões relacionadas à gestão de projetos, qualidade do software, processos de desenvolvimento, entre outros aspectos cruciais para o sucesso na criação e manutenção de sistemas de software. “Engenharia de software é uma disciplina de engenharia cujo foco está em todos os aspectos da produção de software, desde os estágios iniciais da especificação do sistema até sua manutenção, quando o sistema já está sendo usado.” (Sommerville, 2011, p. 5).

2.1.1 Desenvolvimento de software

De acordo com Sommerville (2011), a concepção de software ultrapassa a mera definição de programas de computador, englobando toda uma gama de elementos associados à engenharia de software. Ao considerar a engenharia de software, torna-se necessário compreender que ela não se restringe apenas ao desenvolvimento do programa em si, mas também à documentação relacionada e aos dados de configuração necessários para garantir o funcionamento adequado desse programa.

Sommerville (2011) ainda ressalta que um sistema de software desenvolvido profissionalmente, vai além de apenas um produto, mas muitas vezes é consistindo em uma coleção de programas individuais e arquivos de configuração que são essenciais para sua operação. Tal sistema pode abranger diversos aspectos, incluindo a documentação do sistema descrevendo sua estrutura, a documentação do usuário para orientação sobre seu uso e *websites* que oferecem informações atualizadas sobre o produto para os usuários. “A engenharia de software tem por objetivo apoiar o desenvolvimento profissional mais do que a programação individual” (Sommerville, 2011, p. 3).

2.1.2 Levantamento de requisitos

Conforme destacado por Paula Filho (2019), a disciplina de requisitos abrange uma série de atividades destinadas a obter uma declaração completa, clara e precisa dos requisitos de um produto de software. O mesmo autor afirma que as seguintes definições de requisitos são aplicáveis e compatíveis com a terminologia do CMMI e dos padrões IEEE.

Paula Filho (2019) ainda afirma que, os requisitos devem ser identificados pela equipe do projeto em colaboração com representantes do cliente, usuários e, possivelmente, especialistas da área de aplicação. Projetos de desenvolvimento de produtos mais complexos, geralmente demandam um investimento maior em engenharia de requisitos em comparação com projetos que lidam com produtos simples.

A complexidade da engenharia de requisitos é amplificada especialmente em projetos envolvendo produtos novos, enquanto projetos que visam desenvolver novas versões de produtos existentes se beneficiam da experiência dos usuários adquirida com versões anteriores, facilitando a identificação das necessidades prioritárias, desta forma o cenário é diferente para produtos totalmente novos. (Paula Filho, 2019).

2.2 LINGUAGEM DE PROGRAMAÇÃO

De acordo com Gotardo (2015) a linguagem de programação é uma sequência de passos e regras que utilizamos para comunicar instruções para um computador

processar. Essas instruções são escritas de maneira padronizada, seguindo orientações e normas sintáticas e semânticas específicas. As regras sintáticas se referem à forma como as palavras e símbolos devem ser organizados no código, enquanto as regras semânticas determinam o significado e a lógica na qual essa sequência de instruções está representada.

Gotardo (2015) diz que, ao construir um programa em uma determinada linguagem de programação podemos especificar diversos aspectos, a exemplo temos os tipos de dados que o computador irá manipular, qual forma esses dados serão armazenados e processados e quais ações devem ser realizadas em diferentes situações.

Gotardo (2015) ainda afirma que, as instruções escritas em qualquer linguagem de programação de sua escolha, são conhecidas como código fonte, que é uma sequência de instruções escritas de acordo com as regras da linguagem escolhida. O código fonte é o ponto de partida para a criação de um programa funcional, site, sistemas, aplicativos e até aparelhos domésticos inteligentes como televisores, ar-condicionado e cafeteiras. Esse código posteriormente será transformado em binário, que será interpretado por uma máquina, na qual executará o código fonte escrito pelo desenvolvedor. A seguir no Quadro 1, é apresentada as linguagens de programação mais utilizadas em abril de 2024.

Quadro 1 - Linguagens mais usadas em abril de 2024

Abril de 2024	Abril de 2023	Linguagem	Avaliação	Mudou
1	1	Python	16.41	1.90
2	2	C	10.21	-4.20
3	4	C++	9.76	-3.20
4	3	Java	8.94	-4.29
5	5	C#	6.77	-1.44
6	7	Javascript	2.89	0.79
7	10	Go	1.85	0.57
8	6	Visual Basic	1.70	-2.70
9	8	SQL	1.61	-0.06
10	20	Fortran	1.47	0.88
11	11	Delphi	1.47	0.24
12	12	Assemble	1.30	0.26
13	18	Ruby	1.24	0.58
14	17	Swift	1.23	0.51
15	15	Scratch	1.14	0.35
16	14	Matlab	1.11	0.25
17	9	Php	1.09	-0.26
18	38	Kotlin	1.05	0.80
19	19	Rust	1.03	0.41
20	16	R	0.84	0.09

Fonte: Elaborado a partir de Tiobe (2024)

2.2.1 Javascript

Segundo Oliveira e Zanetti (2021), Javascript é uma linguagem de programação orientada a objetos que pode ser interpretada e executada tanto por navegadores quanto por aplicativos como NodeJs. Com uma sintaxe que lembra a linguagem Java, Javascript é projetado principalmente para aumentar a interatividade nas páginas da internet.

O Javascript é uma linguagem de programação orientada a objetos que é interpretada e executada pelo navegador web (*client-side* script) ou através de uma aplicação (NodeJs). Apresenta uma sintaxe similar à linguagem Java e tem como objetivo principal dar uma maior interatividade às páginas (Oliveira; Zanetti, 2021, p. 8).

Oliveira e Zanetti (2021) afirmam que, uma característica fundamental do Javascript é a sua flexibilidade em relação aos tipos de dados. Diferentemente de outras linguagens de programação, o Javascript não faz tipagem de elementos, e sim, adequa conforme a necessidade, por exemplo em uma linha do código a variável é do tipo texto e em outra linha do código é do tipo inteiro, conforme observamos o código no Quadro 2.

Quadro 2 - Exemplo de código Javascript

Código Javascript
<pre>var x = 'Olá'; console.log('Variáveis em Javascript'); console.log('Texto: ' + x + ', tipo de dados: ' + typeof x); x = 12 + 13; console.log('Número inteiro: ' + x + 'tipo de dados: ' + typeof x); x = 4.50 + 5.25; console.log('Número real: ' + x + 'tipo de dados: ' + typeof x); x = false; console.log('Lógico: ' + x + 'tipo de dados: ' + typeof x);</pre>

Fonte: Oliveira e Zanetti (2022, p. 8)

De acordo com Oliveira e Zanetti (2021), isso significa que uma variável em Javascript pode conter diferentes tipos de dados ao longo da execução do programa, e o tipo dos dados é definido de acordo com a informação que está armazenada naquela variável em um dado momento conforme demonstrado no Quadro 3. Além disso, o tipo de dados de uma variável pode ser modificado dinamicamente ao longo da execução da aplicação, conforme o seu conteúdo é alterado.

Quadro 3 - Resultado da execução do código da Quadro 2

Retorno da operação do código da Quadro 2
Variáveis em Javascript Texto: Olá, tipo de dado: string Número inteiro: 25, tipo de dado: number Número real: 9.75, tipo de dado: number Lógico: false, tipo de dado: boolean

Fonte: Oliveira e Zanetti (2022, p. 8)

2.2.2 NodeJs

De acordo com Oliveira e Zanetti (2021) o NodeJs é uma plataforma de servidor de código aberto que permite a execução de aplicações escritas em Javascript e pode funcionar como uma linguagem de programação *server-side* em aplicações para a internet. Tecnicamente, o NodeJs consiste em um ambiente de execução Javascript construído para executar em aplicações em nuvem.

Oliveira e Zanetti (2021) afirmam que, a edição do código fonte pode ser realizada em editores simples como o bloco de notas ou em editores mais avançados, como o notepad++ ou o sublime. No entanto, é altamente recomendado utilizar ambientes integrados de desenvolvimento (IDEs) como o NetBeans, que oferece suporte específico para o desenvolvimento em Java ou o Microsoft Visual Studio Code, entre outras opções disponíveis. Estas IDEs proporcionam uma gama de recursos adicionais que facilitam o desenvolvimento e a depuração de aplicações NodeJs, além de oferecerem uma interface amigável e eficiente para os desenvolvedores.

De acordo com Pereira (2016), para criar ou utilizar uma aplicação em NodeJs é preciso primeiro instalar o node na sua máquina. Para fazer isto, é preciso fazer o download do NodeJs no site oficial <https://nodejs.org/en/download/package-manager> escolhendo a versão do sistema operacional que deseja utilizar e instalar após o download.

Pereira (2016), complementa que para verificar se o NodeJs foi instalado corretamente basta abrir o terminal e executar o seguinte comando “node –version” ou “node -v”, esse comando irá mostrar a versão do NodeJs que está instalado na sua máquina.

2.2.3 TypeScript

De acordo com a documentação de software do NodeJs (2024) que complementa o TypeScript, encontra-se afirmado que TypeScript é uma linguagem *open source* mantida pela Microsoft na qual basicamente adiciona-se um superconjunto de Javascript na linguagem.

TypeScript ainda de acordo com a documentação do NodeJs (2024), oferece uma variedade de mecanismos poderosos, incluindo interfaces, classes e tipos utilitários que enriquecem o desenvolvimento de aplicações. Em projetos de maior escala é possível configurar o comportamento do compilador TypeScript de forma detalhada através de um arquivo de configuração separado. Essa configuração permite ajustar a rigidez das verificações de tipo, definir diretórios de saída para os arquivos compilados e controlar diversos outros aspectos do processo de compilação, proporcionando flexibilidade e controle granular sobre o desenvolvimento do projeto. (NodeJs, 2024).

2.2.4 Express

Conforme Cardoso (2021) explica, *express* é um *framework* muito utilizado no desenvolvimento de aplicações em NodeJs, funcionando como uma biblioteca base para muitos outros *frameworks* que são utilizados pelo NodeJs e oferecendo várias ferramentas para trabalhar com requisições HTTP em diferentes URL. O *express* facilita a criação de aplicações web controlando configurações padrões de forma transparente, como a porta de conexão e a localização dos modelos usados para renderizar as respostas.

Cardoso (2021) diz que uma variedade de bibliotecas se encontra disponível através do *express*, nestas o desenvolvedor pode encontrar ferramentas para utilizar em cenários diferentes, que oferecem uma maior diversidade de opções, para o desenvolvimento de aplicações para internet. Em resumo, o *express* simplificou o desenvolvimento, proporcionando ao projeto uma estrutura mais sólida para lidar com as requisições que chegam, simplificando a criação de projetos com muito mais eficiência e segurança.

Cardoso (2021) afirma que, o *express* oferece a praticidade de inserir novos procedimentos de requisições através de *middleware* em diferentes etapas da fila de requisições. A ferramenta é conhecida por sua abordagem minimalista, porém os desenvolvedores têm a liberdade de elaborar conjuntos de *middleware* personalizados. Esses conjuntos podem solucionar obstáculos que surgem no decorrer da criação de um aplicativo, além de contar com bibliotecas para serem aplicadas em variadas circunstâncias.

2.2.5 Mapeamento Objeto-Relacional – PRISMA ORM

A definição exposta por PRISMA (2024), caracteriza-o como um conjunto de ferramentas de desenvolvimento de banco de dados que simplifica e agiliza o processo de construção e manutenção de aplicações NodeJs, dentre outras. A biblioteca PRISMA funciona como uma camada de interação entre a aplicação e o banco de dados, oferecendo um ORM (*Object-Relational Mapping*) completo.

Ainda podemos aprender com PRISMA (2024) que, a principal função do PRISMA é facilitar a comunicação entre a aplicação e o banco de dados, fornecendo uma interface intuitiva para realizar consultas, modificações e migrações de esquemas de banco de dados. Ele suporta vários tipos de bancos de dados, como por exemplo o PostgreSQL, MySQL.

PRISMA (2024), ainda oferece recursos avançados como migrações de banco de dados automáticas, que facilitam atualizações e mudanças no banco de dados ao longo do tempo, garantindo que as alterações sejam aplicadas de forma consistente e segura.

2.2.6 Npm

Ainda conforme explica Cardoso (2021), o NPM é o gerenciador de pacotes que o NodeJs utiliza. Ele é uma ferramenta essencial para quem desenvolve projetos com o NodeJs e outras tecnologias como o TypeScript. O NPM proporciona acesso a uma diversidade de pacotes que podem ser facilmente instalados e reutilizados em projetos. Uma das principais vantagens do NPM é a capacidade de automatizar tarefas de compilação, simplificando o processo de desenvolvimento.

Segundo Cardoso (2021), entre as vantagens do NPM, encontra-se sua configuração multiplataforma, que permite seu funcionamento em diversos sistemas operacionais e ainda oferece suporte a vários provedores de hospedagem na nuvem, simplificando a construção de aplicações com NodeJs em servidores.

Cardoso (2021) diz que um exemplo do uso de NodeJs encontra-se na criação de um servidor web, com uma API implementado com NodeJs, que responde requisições HTTP. Com poucas linhas de código é possível implementar uma API que recebe requisições e retorna uma mensagem em JSON ou XML. Por exemplo, podemos criar um *endpoints* que responde com a *string* "Conectado com sucesso" para uma requisição GET enviada para a URL de exemplo: `http://localhost:3000`.

2.3 BANCO DE DADOS

De acordo com Alves (2014) primeiro é preciso explicar o que é a diferença entre informação e dado. Informação consiste em qualquer fato ou conhecimento sobre o mundo real que pode ou não ser registrado ou gravado, como por exemplo: "Está muito quente hoje.", por sua vez o dado é a representação dessa informação, que pode ser escrita em papel, quadro de avisos ou no disco de um computador, a exemplo temos: "A temperatura hoje é de 38 graus celsius". Em resumo, informação é o que entendemos sobre algo, enquanto dado é a forma como essa informação é gravada.

Como Alves (2014) expõe, um banco de dados é um conjunto de dados que contém um significado. Com essa explicação, podemos entender equivocadamente que, um agrupamento de palavras em um texto é equivalente a um banco de dados, porém, o termo banco de dados se destaca por suas características como por exemplo: um sistema de banco de dados é criado com o propósito de retratar e armazenar dados referentes a um segmento específico da realidade. Da mesma forma, é necessário que ele mantenha registros exatos e atualizados sobre essa parcela da realidade. Sendo assim, qualquer modificação que ocorra, como a contratação de um novo funcionário, a venda de um produto ou o tratamento de um paciente, é preciso que este dado seja inserido ou atualizado no banco de dados, a fim de garantir a sua integridade e refletir com fidelidade a realidade que ele representa.

“Um banco de dados é um conjunto lógico e ordenado de dados que possuem algum significado, e não uma coleção aleatória sem um fim ou objetivo específico.” (Alves, 2014, p. 17).

Ainda de acordo com Alves (2014), um banco de dados é criado e alimentado a partir de dados que são utilizados com um propósito definido, sendo que há usuários e aplicações desenvolvidos para gerenciá-los. Para que um banco de dados exista são necessários três elementos essenciais, primeiramente uma fonte de informação da qual os dados são originados, em segundo lugar uma interação com o mundo real, e por fim um público interessado nas informações contidas na base de dados.

Alves (2014) ainda explica que existem sistemas de bancos de dados que são considerados genéricos, ou seja, podem armazenar praticamente qualquer tipo de informação, uma vez que o próprio usuário define a estrutura do arquivo. Isso é observado em softwares como Access, Interbase, MySQL, PostgreSQL, SQL Server, Oracle, entre outros. Essas soluções são conhecidas por sua flexibilidade e capacidade de lidar com tarefas complexas de gerenciamento. Alguns desses sistemas possuem uma linguagem de programação própria, porém a maioria utiliza a SQL, uma linguagem declarativa popular para manipular bancos de dados relacionais.

Segundo Alves (2014), tal conjunto de recursos possibilita aos usuários a criação de rotinas específicas e até mesmo de aplicativos completos. No entanto, é importante ressaltar que essa versatilidade tem seu custo, os bancos de dados genéricos costumam ter um desempenho inferior quando comparados a sistemas especializados, além de demandarem mais recursos de memória RAM e espaço em disco.

2.3.1 Banco de dados relacional

Ainda de acordo com Alves (2014), na maior parte dos sistemas de bancos de dados é utilizada a forma de banco relacional. Esses bancos de dados se destacam por sua organização dos dados em tabelas compostas por linhas e colunas. Dessa forma, as tabelas assemelham a conjuntos de elementos ou objetos, relacionando as informações de maneira estruturada. Na Figura 1 temos um exemplo de tabelas com suas ligações relacionais. Como podemos observar, a tabela de produtos faz referência através da coluna “CodigoCategoria” para a tabela de categoria de

produtos, e a coluna “CodigoFornecedor”, faz referência para a tabela de fornecedores.

Figura 1 - Relacionamento de tabelas

Tabela de categoria de produtos

Categorias	
CodigoCategoria	DescricaoCategoria
1	Eletrônicos
2	Eletrrodomésticos
3	Brinquedos
4	Móveis
*	(Novo)

Tabela de produtos

Produtos				
CodigoProduto	CodigoCategoria	CodigoFornecedor	NomeProduto	Preco
0101231	3	2	Teclado musical ExpertMusic	650
123456	1	3	Aparelho de som Quasar	230
5123511	4	1	Jogo de dormitório Colibri	1230
*				0

Tabela de fornecedores

Fornecedores					
CodigoFornecedor	NomeFornecedor	Endereco	Bairro	Cidade	Estado
1	ABC Móveis Domésticos Ltda	R. Doze, 120	Centro	São Paulo	SP
2	Brinquedos & Jogos Educar	Av. das Nações, 280	Jd. América	Atibaia	SP
4	SomMaster	Av. do Lago	Jd. do Lago	Osasco	SP
*	(Novo)				

Fonte: Alves (2014, p. 17).

2.3.2 PostgreSQL

Segundo Carvalho (2017), PostgreSQL é um sistema de gerenciamento de banco de dados relacional muito poderoso e conciso totalmente *open source*. O PostgreSQL tem mais de 15 anos de desenvolvimento ativo e uma arquitetura reconhecida e comprovada por sua confiabilidade, integridade, segurança e conformidade com os padrões de SQL.

De acordo com Carvalho (2017), o PostgreSQL possui várias características que o tornam tão robusto, como a alta segurança dos dados, por ser confiável, rápido e fácil de usar, ou seja, para o que precisar de um banco de dados, o PostgreSQL

oferece com excelência e totalmente gratuito seja para qualquer finalidade pessoal ou corporativo. No Quadro 4, é apresentado alguns recursos do PostgreSQL.

Quadro 4 - Recursos do PostgreSQL

Recursos do sistema	Descrição
Controle de concorrência multiversionado (MVCC)	Garante a consistência dos dados permitindo transações simultâneas sem bloqueios conflitantes.
Recuperação em um ponto no tempo (PITR)	Permite a restauração do banco de dados a um estado anterior em caso de falhas.
<i>Tablespaces</i>	Oferece a capacidade de definir locais específicos no sistema de arquivos para armazenar dados.
Replicação assíncrona	Permite a cópia dos dados para outros servidores sem impactar a performance do servidor principal.
Transações agrupadas (<i>savepoints</i>)	Habilita pontos intermediários dentro de uma transação, possibilitando <i>rollback</i> parcial.
Cópias de segurança quente (online/hot backup)	Possibilita a realização de backups sem interromper o funcionamento do banco de dados.
Planejador de consultas sofisticado (otimizador) e registrador de transações sequencial (WAL)	Melhora a performance das consultas e oferece tolerância a falhas através de registros de transações.
Suporte a conjuntos de caracteres internacionais	Oferece suporte a diversos conjuntos de caracteres, permitindo a utilização de várias línguas no banco de dados.
Codificação de caracteres <i>multibyte</i> , Unicode e ordenação por localização	Suporta caracteres de múltiplos bytes e ordenação de acordo com regras específicas de cada localidade.
Sensibilidade a caixa (maiúsculas e minúsculas) e formatação	Diferencia maiúsculas de minúsculas e mantém a formatação dos dados.
Alta escalabilidade	Gerencia grandes volumes de dados e acomoda um elevado número de usuários concorrentes. Existem sistemas em produção com PostgreSQL gerenciando mais de 4TB de dados.

Fonte: Elaborado conforme Carvalho (2017, p. 6)

2.3.3 Gerenciador de banco de dados SGBD

A partir do momento em que se aborda o conceito de sistema de gerenciamento de banco de dados (SGBD) observa-se a comparação levantada por Alves (2014). Em sua definição o autor correlaciona o SGBD com um banco de dados que em seu funcionamento representa um conjunto de dados interligados atuando em uma escala superior na qual o gerenciador fornece recursos e aplicativos que possibilitam a manipulação de seus respectivos bancos de dados. Logo, o SGBD pode ser visto como um software avançado voltado para a elaboração, criação e controle de dados. As definições das funções do SGBD, são listadas no Quadro 5.

Quadro 5 - Funções do gerenciador do banco de dados

Definição	Detalhamento dos formatos de informação, das configurações das Quadros e das condições que precisam ser aplicadas aos dados a serem guardados.
-----------	------------------------------------------------------------------------------------------------------------------------------------------------

Construção	Procedimento de reunir as informações em um ambiente de armazenamento sob total controle do Sistema de Gerenciamento de Banco de Dados (SGBD).
Manipulação	Realizar tarefas como manipular o banco de dados adicionando, removendo e alterando registros, obter informações, como consultas e elaboração de relatórios.

Fonte: Elaborado de acordo com Alves (2014, p. 14).

2.3.4 Linguagem SQL

Em seu trabalho Alves (2014) explana que a linguagem SQL teve sua origem na década de 1970 como parte de um projeto de pesquisa realizado pela IBM. Esta linguagem foi concebida nos laboratórios da empresa em San Jose - EUA, estando seu desenvolvimento impulsionado por um artigo escrito por Edgar F. Codd, pesquisador da IBM, que propôs o modelo relacional para bancos de dados.

Segundo Alves (2014), a IBM começou a trabalhar em uma linguagem que poderia manipular e gerenciar dados dentro do modelo relacional proposto. Esta linguagem inicial foi chamada SEQUEL. A primeira implementação de SEQUEL foi desenvolvida para um projeto de banco de dados chamado *System R*, um sistema de protótipo que demonstrava a viabilidade do modelo relacional. Posteriormente o nome da linguagem foi abreviado para SQL.

Ainda conforme explica Alves (2014), o SQL é a linguagem padrão de utilização nos principais bancos de dados relacionais, ou seja, podemos manipular os dados e a estrutura do banco, utilizando apenas a linguagem SQL. No Quadro 6 são listados alguns comandos básicos de SQL e a sua finalidade.

Quadro 6 - Comandos SQL e suas funções

Comando	Função
Create table	Comando SQL utilizado para a criação de Quadros na base de dados Create table Quadro { id integer not null, nome varchar, data timestamp }
Drop table	Comando SQL utilizado para deletar Quadro na base de dados Drop table Quadro;
Insert	Comando SQL utilizado para inserir dados em uma Quadro na base de dados insert into Quadro (id, nome, data) values (1, 'Nome do cadastro', '2024-05-14')
Delete	Comando SQL utilizado para deletar dados em uma Quadro na base de dados delete from Quadro No comando delete deve ser utilizado o comando Where junto para garantir integridade dos dados

	delete from Quadro Where id = 1
Update	Comando SQL utilizado para atualizar dados em uma Quadro na base de dados update Quadro set coluna = 'Valor desejável' Where id = 1
Select	Comando SQL utilizado para selecionar dados em uma Quadro na base de dados Para selecionar todos as colunas da Quadro usamos Select * from Quadro Para selecionar apenas as colunas desejado utilizamos Select nome from Quadro
Where	Comando SQL utilizado para criar filtros na execução da query O Comando Where é utilizado em quase todos os parâmetros do SQL, pode ser usado no comando Select, pode ou deve ser usado no comando delete e, é de extrema importância sempre usar no comando update

Fonte: Elaborado de acordo com Alves (2014, p. 122).

2.4 APPLICATION PROGRAMMING INTERFACE (API)

Conforme Ferreira (2021) API significa *Application Programming Interface*, o qual consiste em um conjunto de regras e métodos de programação com o objetivo de estabelecer uma comunicação entre aplicações em nuvem, resumidamente, são ferramentas de desenvolvimento e protocolos que possibilitam integrar sistemas através da internet.

Ferreira (2021) diz que, API permite que outras aplicações possam utilizar suas funcionalidades sem o conhecimento de como a sua implementação foi desenvolvida, sendo assim, sem a necessidade de ficar preso a apenas uma linguagem de programação, possibilitando a comunicação entre sistemas que foram desenvolvidos com tecnologias distintas.

2.4.1 Rest

Conforme Ferreira (2021) o REST foi desenvolvido por Roy Fielding e alude a um conjunto organizado de arquiteturas que busca reduzir a dependência e a complexidade das implementações dos componentes. O REST foi desenvolvido com a finalidade de preservar o protocolo HTTP para a realização de transações na internet sem a necessidade de outro protocolo, trabalhando apenas com seus próprios recursos internos.

Ferreira (2021) ainda diz que a arquitetura REST é responsável por efetuar as operações de inserir, atualizar, excluir e recuperar dados na internet. Sistemas que

adotam esta arquitetura, são chamados de RESTFULL e seguem fielmente os princípios estabelecidos por Roy Fielding.

2.4.2 Métodos

De acordo com MDN (2024) o HTTP protocolo estabelece um grupo de métodos de solicitação que determinam a ação a ser realizada em um recurso específico. Apesar de serem muitas vezes identificados como verbos HTTP esses métodos são, na verdade, considerados substantivos. Cada um desses métodos desempenha uma função distinta, no entanto, alguns conceitos são compartilhados por alguns deles, ilustrado no Quadro 7.

Quadro 7 - Explicações das funções dos métodos

Métodos	Função
Get	Método GET é para obter a representação de um recurso específico. As solicitações usando o método GET devem retornar apenas dados.
Head	Método HEAD solicita uma resposta de forma idêntica ao GET, porém sem o conteúdo da resposta.
Post	Método POST é empregado para submeter uma entidade a um recurso específico, muitas vezes resultando em uma mudança no estado do recurso ou efeitos secundários no servidor.
Put	Método PUT substitui todas as representações atuais do recurso de destino pela carga de dados da requisição.
Delete	Método DELETE exclui um recurso específico.
Connect	Método CONNECT cria um túnel para o servidor identificado pelo recurso de destino.
Options	Método OPTIONS é usado para descrever as opções de comunicação com o recurso de destino.
Trace	Método TRACE realiza um teste de chamada de <i>loop-back</i> juntamente com o caminho para o recurso de destino.
Patch	Método PATCH é utilizado para aplicar modificações parciais a um recurso.

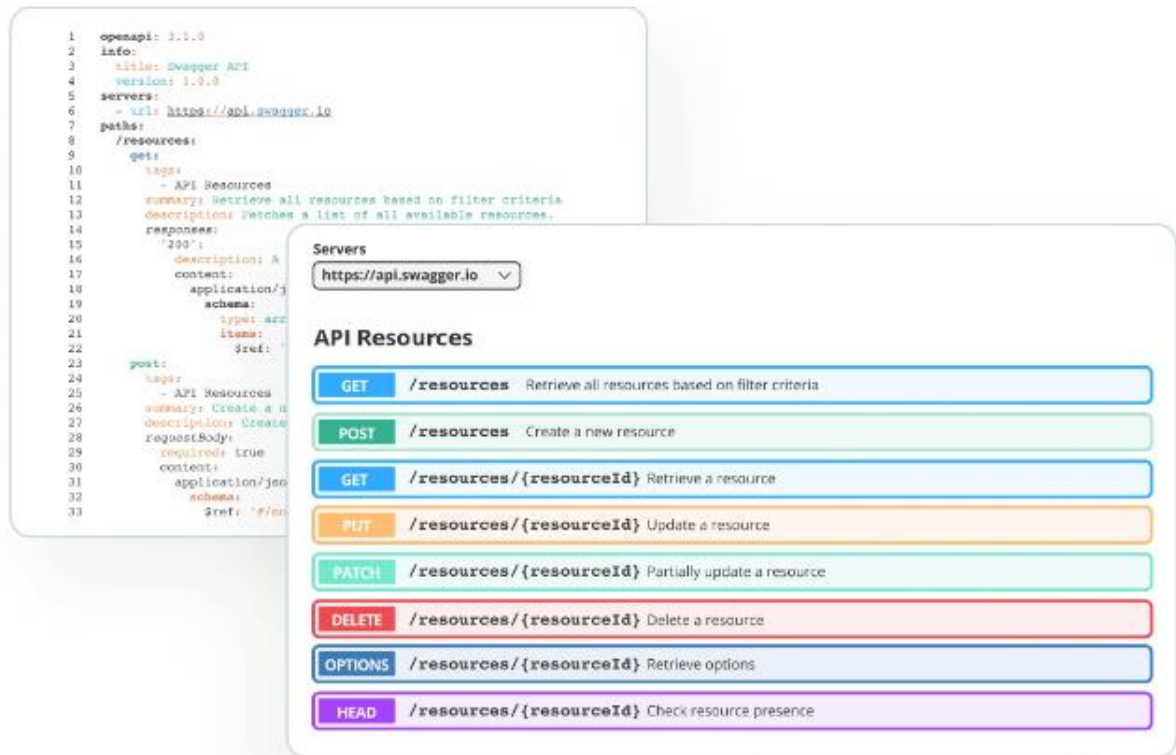
Fonte: MDN (2024)

2.4.3 OpenAPI

De acordo com a documentação do *Swagger* (2024), a especificação *OpenAPI*, também chamada de *Swagger Specification* no passado, é um formato usado para caracterizar APIs REST. Um documento *OpenAPI* possibilita o registro completo da API, abrangendo os pontos de acesso disponíveis, como o `/users`, e as ações executadas em cada um deles, como o método GET em `/users` para a listagem de usuários ou o método POST para a criação de novos. Ela define os parâmetros de entrada e saída de cada procedimento, os métodos de autenticação imprescindíveis,

além de fornecer informações extras, tais como contatos, licenças, termos de uso e mais. As especificações podem ser redigidas em YAML ou JSON, que são formatos de fácil aprendizado e compreensão para indivíduos e sistemas. Na Figura 2 mostra o modelo de apresentação do *Swagger*.

Figura 2 - Modelo de apresentação do *Swagger*



Fonte: Documentação *Swagger* (2024)

2.4.4 *Swagger*

De acordo com a documentação do *Swagger* (2024), ele é um conjunto de ferramentas *open source* criado com base no *OpenAPI*, no qual ajudam na projeção, construção, documentação e consumo de APIs REST. Conforme especificado na documentação, o *Swagger* possui várias ferramentas que ajudam no processo sendo *Swagger Editor*, *Swagger UI*, *Swagger Codegen*, *Swagger Editor Next*, *Swagger Core*, *Swagger Parser*, *Swagger APIDom*, estas ferramentas cada uma em sua especificação ajudam em todo o processo desde o planejamento a conclusão do projeto.

2.4.5 Insomnia

Segundo a documentação do Insomnia (2024), é uma aplicação *open source* na qual permite ao usuário uma interface fácil de utilizar na qual está alinhada a recursos sofisticados como criação de códigos e a utilização de variáveis de ambientes. Com o Insomnia é possível a depuração de APIs com a utilização de protocolos e outras formas comuns, ainda é possível desenvolver APIs com o editor *OpenAPI* que está embutido na estrutura do Insomnia.

Ainda conforme a documentação do Insomnia (2024) explica, ele possibilita a simulação de APIs com a utilização dos servidores em nuvem, além disso, é possível criar *pipelines* de CI/CD através do próprio Insomnia CLI, que dispõe de funcionalidades de testes automáticos.

2.4.6 Postman

De acordo com Postman (2024), ele é uma plataforma para o desenvolvimento, testes e ciclos de vida de uma API. Com o postman é possível testar requisições REST da API, analisando as informações de *request* e *response* de acordo com a necessidade do desenvolvimento. Também é possível criar testes de requisições para validar os dados e processos da API.

Segundo a documentação do Postman (2024), também é possível criar a documentação da API através do postman, onde a documentação pode ser gerada automaticamente através da definição do *OpenAPI*.

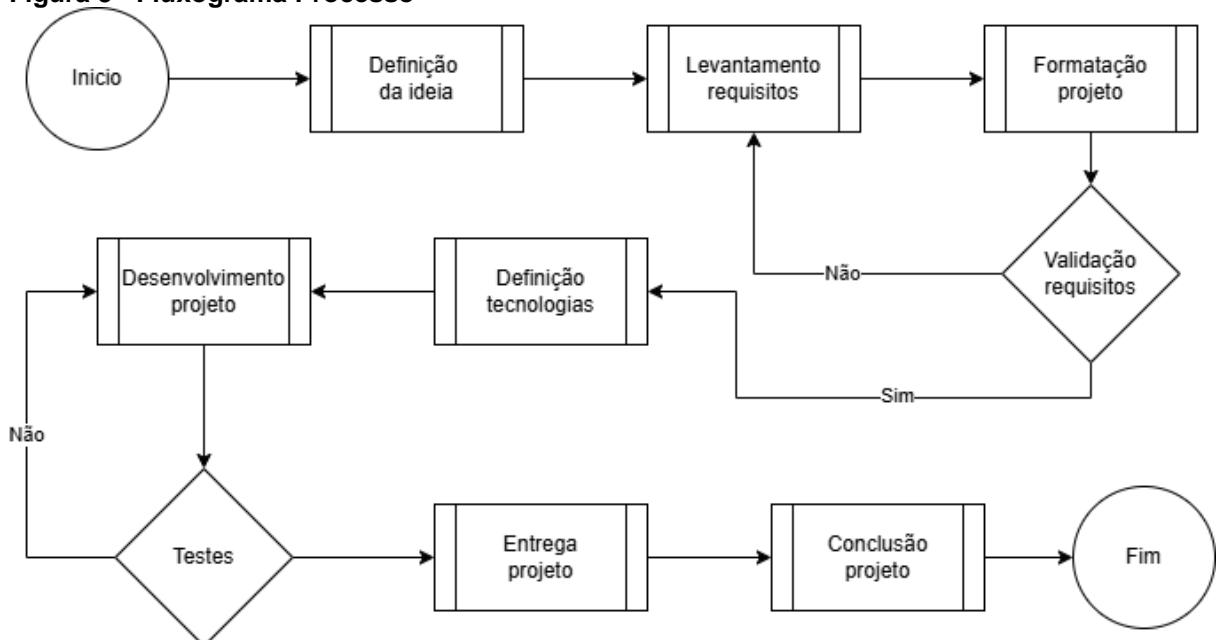
Ainda conforme especificado na documentação do Postman (2024), ele tem um poderoso tempo de execução baseado em NodeJs, no qual é possível adicionar comportamentos dinâmicos nas requisições, ou seja, é possível criar *scripts* de testes que podem ser executados antes ou depois das requisições, adicionar parâmetros dinâmicos nas requisições, transmitir dados entre as requisições e muitas outras formas de trabalhar.

3 METODOLOGIA DA PESQUISA

O presente trabalho de conclusão de curso caracteriza-se como uma pesquisa aplicada e descritiva. Seu objetivo, é o desenvolvimento de uma camada de *endpoints* (API *backend*), para o consumo destes processos por uma aplicação *web* (*frontend*). Através deste *backend*, são processados dados de pessoas, onde separamos os cadastros por tipo de pessoas, administrador, professor e aluno, também processando os dados de treinos e exercícios. Nestas classes incluímos os métodos de cadastrar, alterar, deletar e selecionar os dados, por meio de requisições REST. Estes dados são armazenados em banco de dados relacional onde utilizamos o PostgreSQL, por ser um sistema confiável, gratuito e moderno. A pesquisa limitou o escopo para subsidiar a melhor estrutura e processo de dados, que demandam as academias.

Na realização da pesquisa, foram utilizados os procedimentos de levantamento documental e de campo, com foco na análise qualitativa. Para esta pesquisa, utilizamos como base uma academia que não possui um sistema de gerenciamento para verificar a necessidade, conceitos, práticas e métodos que são utilizados, onde adquirimos o entendimento para o desenvolvimento do projeto. Na Figura 3, é apresentado um fluxograma que mostra os pontos e processos da pesquisa.

Figura 3 - Fluxograma Processo



Fonte: Acervo do Autor.

A operacionalização da pesquisa seguiu os seguintes passos: inicialmente, foram realizadas conversas com a pessoa responsável pela academia em questão, objetivando compreender a forma de trabalho atual da academia. Em seguida foi feito um levantamento detalhado das operações e processos utilizados. Posteriormente analisaram-se os requisitos necessários para que o sistema possa atender a demanda da academia, incluindo a identificação dos métodos e processos que precisam ser implementados. Ela também buscou entender a dinâmica de trabalho dos profissionais que atuam na academia, para projetar um software que atenda e auxilie da melhor forma a necessidade que eles precisam. Por fim, foram coletadas evidências de um antigo sistema da academia, ainda contido no computador da academia. Tal sistema não está mais em operação, entretanto serve como subsídio para ser analisado e agregar ainda mais à estrutura de operação que o novo sistema contém.

3.1 CLIENTE DE CONSUMO DOS *ENDPOINTS*

Uma aplicação web (*frontend*) e um aplicativo móvel foram desenvolvidos em parceria com outro acadêmico de sistemas de informação, com o objetivo de consumir os *endpoints* da API. Com a responsabilidade do desenvolvimento das interfaces e telas nas quais os usuários utilizarão para interagir com o sistema, gerenciar dados e acessar as informações processadas pela API.

Esta aplicação web (*frontend*) e o aplicativo móvel contém um conjunto de funcionalidades, com foco em proporcionar aos usuários uma experiência completa e eficiente. O sistema contém telas específicas para *login*, página inicial (*home*), e módulos de manutenção que abrangem diferentes áreas do sistema. Entre essas áreas estão a gestão de alunos, professores, treinos e exercícios, que são fundamentais para o funcionamento e organização do sistema.

A tela de *login* é projetada para autenticar os usuários, garantindo que apenas pessoas autorizadas tenham acesso ao sistema. Após a autenticação, o usuário é direcionado à tela inicial (*home*), onde pode visualizar informações gerais e acessar os módulos nos quais tiver disponível para seu acesso.

Os módulos de manutenção foram pensados para simplificar a gestão dos dados. O módulo de alunos, por exemplo, permite ao usuário cadastrar, alterar, deletar

e pesquisar os dados do aluno, enquanto o de professores possibilita o gerenciamento dos dados dos instrutores da academia. Já os módulos de treino e exercício foram desenvolvidos para estruturar e organizar rotinas de treinamento, facilitando a configuração de atividades físicas de maneira prática e personalizada.

Além disso, o aplicativo móvel foi projetado para funcionar como uma extensão do sistema, permitindo que os alunos tenham acesso a informações de treinamentos que foram geradas pelo instrutor responsável, facilitando os treinamentos do aluno. A integração entre o aplicativo, a aplicação web e a API mantém a consistência dos dados e o funcionamento entre os sistemas como um todo.

Com essa divisão de responsabilidades e a colaboração entre as partes, o sistema entrega um ambiente completo para gerenciar todas as informações necessárias, oferecendo uma solução robusta e adaptada às necessidades dos usuários.

4 API DE INTEGRAÇÃO PARA *WEBSERVICE* DE ACADEMIA

Neste capítulo, é abordado temas referentes a implementação da API, demonstrando uma análise do projeto e o seu desenvolvimento.

4.1 ANÁLISE

A engenharia de software nos ensina práticas, métodos e padrões de desenvolvimento de sistemas, que nos auxiliam em projetos de software. Com a utilização dos métodos da engenharia de software, temos a possibilidade de desenvolver sistemas nos quais ofereceriam um maior desempenho, uma manutenção facilitada e maior qualidade no software.

A criação de uma API *backend* utilizando NodeJs, PRISMA e PostgreSQL, é uma boa prática para processar os dados fornecidos pelo *frontend*. A escolha do NodeJs foi motivada pela simplicidade de desenvolvimento e fácil manutenção, além da capacidade de processamento assíncrono. Isso é especialmente importante em aplicações *backend* onde a API precisa suportar muitas operações em tempo real.

O PRISMA, por sua vez, permite a interação entre o *backend* e o banco de dados, que neste caso utilizamos o PostgreSQL. O PRISMA elimina a complexidade das consultas SQL e permite que os desenvolvedores trabalhem com dados usando uma sintaxe mais intuitiva vinculada à lógica do código, ajudando a garantir operações no banco de dados de forma segura e eficientes.

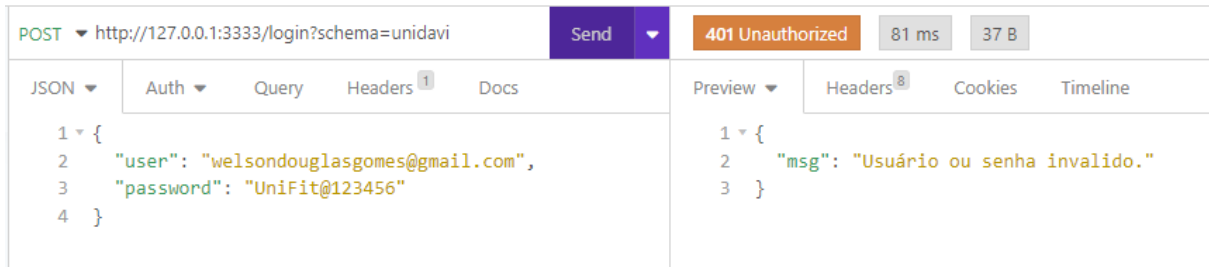
4.2 VISÃO GERAL

O desenvolvimento da API teve como objetivo o processamento de dados de forma eficiente, garantindo a integridade e a disponibilidade das informações para os usuários em tempo real. Como a aplicação interage com dados sensíveis em diversos pontos, a proteção dos dados não pode ficar de fora, e o usuário não poderá ter acesso a todas as informações, mas sim somente as que são de sua responsabilidade e interesse. Na Figura 4, é apresentado um fluxograma dos processos que a API contém. É destacado em verde a entrada e em vermelho as possíveis saídas previstas.

Fonte: Acervo do Autor.

Caso a requisição não seja autorizada, será retornado uma mensagem de usuário ou senha invalido com *status* 401 de não autorizado, conforme apresentado na Figura 6.

Figura 6 - Requisição de *login* da API com falha



The screenshot displays a REST client interface for a failed login attempt. The request is a POST to `http://127.0.0.1:3333/login?schema=unidavi`. The request body is a JSON object with the following structure:

```
1 {
2   "user": "welsondouglassgomes@gmail.com",
3   "password": "UniFit@123456"
4 }
```

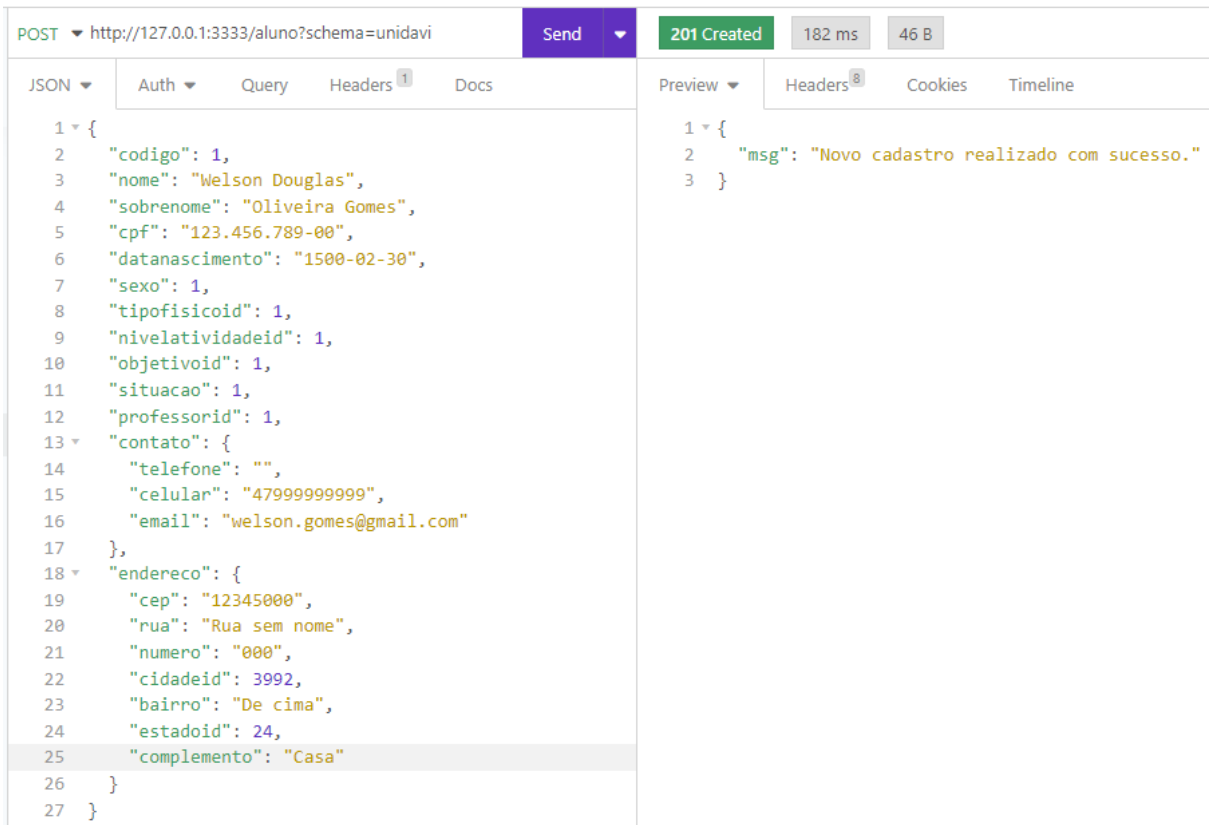
The response is a 401 Unauthorized status, with a response time of 81 ms and a body size of 37 B. The response body is a JSON object with the following structure:

```
1 {
2   "msg": "Usuário ou senha invalido."
3 }
```

Fonte: Acervo do Autor.

Outro exemplo é para cadastro de aluno, neste caso, a API envia uma requisição JSON para o *endpoints* de pessoa, contendo no cabeçalho da requisição o *token* de autorização e no corpo, o JSON com os dados do aluno que está sendo cadastrado. Na Figura 7, apresenta um exemplo de requisição para cadastro de aluno e o retorno.

Figura 7 - Requisição de cadastro de aluno da API



```

POST http://127.0.0.1:3333/aluno?schema=unidavi
201 Created 182 ms 46 B

JSON Auth Query Headers 1 Docs Preview Headers 8 Cookies Timeline

1 {
2   "codigo": 1,
3   "nome": "Welson Douglas",
4   "sobrenome": "Oliveira Gomes",
5   "cpf": "123.456.789-00",
6   "datanascimento": "1500-02-30",
7   "sexo": 1,
8   "tipofisicoid": 1,
9   "nivelatividadeid": 1,
10  "objetivoid": 1,
11  "situacao": 1,
12  "professorid": 1,
13  "contato": {
14    "telefone": "",
15    "celular": "47999999999",
16    "email": "welson.gomes@gmail.com"
17  },
18  "endereco": {
19    "cep": "12345000",
20    "rua": "Rua sem nome",
21    "numero": "000",
22    "cidadeid": 3992,
23    "bairro": "De cima",
24    "estadoid": 24,
25    "complemento": "Casa"
26  }
27 }

1 {
2   "msg": "Novo cadastro realizado com sucesso."
3 }

```

Fonte: Acervo do Autor.

4.3 REGRAS DE NEGÓCIOS

As regras de negócio são responsáveis por definir as normas e diretrizes que devem ser implementadas para assegurar que a aplicação funcione de maneira correta e alinhada aos objetivos do negócio. Elas orientam o desenvolvimento, garantindo que a API atenda às necessidades específicas e operacionais conforme contexto do escopo. No Quadro 8, são listadas as principais regras de negócio que devem ser consideradas para o desenvolvimento da API.

Quadro 8 - Regras de negócio

Código	Descrição
Professores/instrutores	
RN01	Para cadastrar, alterar ou deletar um cadastro de professor/instrutor, o usuário deve ser um administrador do sistema.
RN02	Os professores/instrutores podem cadastrar novos alunos.
Alunos	
RN03	Todo aluno cadastrado deve ter obrigatoriamente um professor vinculado a ele.
RN04	Os alunos devem conseguir marcar o peso que ele está utilizando nos exercícios.
RN05	Os alunos devem ter acesso a informações detalhadas sobre os exercícios, incluindo imagens ou vídeos, para assegurar a execução correta das atividades.
Exercícios	

RN06	Para cadastrar, alterar ou deletar exercício só pode ser feito por um professor/instrutor.
RN07	Um ou vários exercícios podem serem removidos ao mesmo tempo e somente por um professor/instrutor.
RN08	Um exercício não pode ser removido se estiver vinculado a um treino ativo.
Treinos	
RN09	Para cadastrar, alterar ou deletar um treino deve ser somente por um professor/instrutor e conter uma data inicial e uma data final.
RN10	Um treino deve estar vinculado a um aluno.
RN11	Vários exercícios de treino podem ser agrupados para o aluno.

Fonte: Acervo do Autor.

4.4 REQUISITOS

Para desenvolver uma aplicação com qualidade e segundo a engenharia de software, precisamos descrever primeiro os requisitos funcionais e não funcionais.

4.4.1 Funcionais

Os requisitos funcionais, descrevem o comportamento da funcionalidade da API, definindo como cada função deverá responder, quais os dados que serão recebidos e quais os dados que deverão ser retornados. No Quadro 9, é apresentado os requisitos funcionais da API, bem como quais regras de negócios eles complementam.

Quadro 9 - Requisitos funcionais

Código	Nome	Descrição	Código RN
RF01	Manutenção de Professor	Para cadastrar, alterar ou deletar um professor/instrutor a API receberá um JSON com os dados necessários e o usuário logado deve conter privilégios de administrador.	RN01
RF02	Manutenção de Alunos	API terá uma rota chamada /Aluno, esta rota encaminha a chamada para a classe de PESSOA, esta classe contém os métodos de get, post, put e delete no qual de acordo com a requisição solicitada será chamada.	RN02 RN03
RF03	Execução de exercício	API deve receber do aluno a quantidade de peso que ele está utilizando no exercício e enviar ao aluno informações importantes do treino, como um gif de como executar o movimento do exercício	RN04 RN05
RF04	Manutenção de exercício	Para cadastrar, alterar ou deletar um exercício a API receberá um JSON com os dados necessários e o usuário logado deve conter privilégios de administrador ou professor.	RN06 RN07 RN08
RF05	Manutenção de treinos	Para cadastrar, alterar ou deletar um treino a API receberá um JSON com os dados necessários e o usuário logado deve conter privilégios de administrador ou professor.	RN09
RF06	Treino com aluno	Para cadastrar um treino, o professor deverá obrigatoriamente vincular o mesmo a um aluno que esteja	RN10 RN11

		vinculado com o professor, podendo ainda ser agrupado vários exercícios.	
RF07	Login	Todo usuário que acessar o sistema deverá ser autenticado antes, para que possa acessar somente o que esteja liberado de acordo com o seu nível de acesso.	RN01 RN02 RN06

Fonte: Acervo do Autor.

4.4.2 Não funcionais

Os requisitos não funcionais são critérios que descrevem características e atributos importantes para a qualidade da aplicação, mas que não estão diretamente relacionados às funcionalidades específicas. No Quadro 10, é apresentado alguns dos requisitos não funcionais da API.

Quadro 10 - Requisitos não funcionais

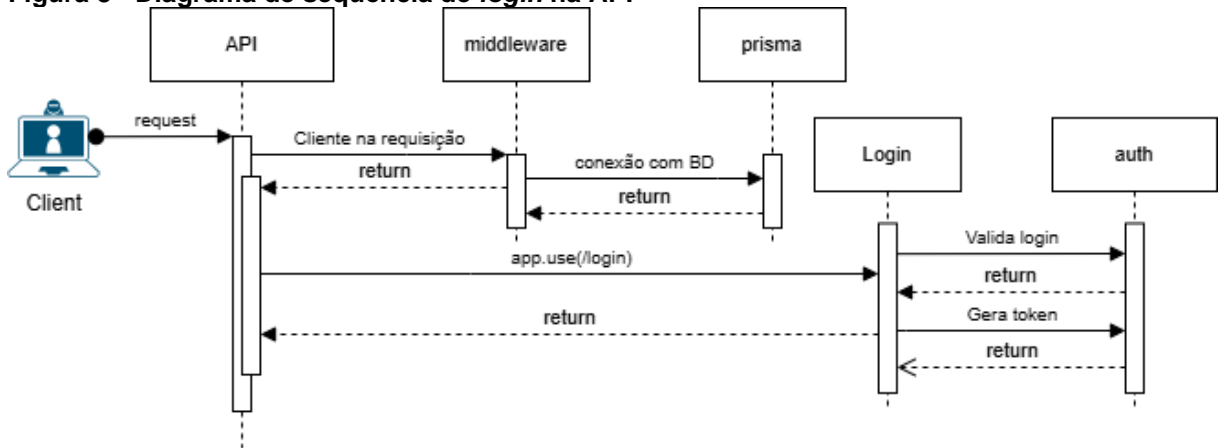
Código	Nome	Descrição
RNF01	Autenticação autorização	API é protegida e os usuários devem estar cadastrado e com acesso liberado contendo um nível de permissão de acesso atribuído a seu tipo de usuário
RNF02	Linguagem de programação	API será construída utilizando a linguagem de programação NodeJs.
RNF03	Banco de dados	Banco de dados utilizado para a persistência de dados será o PostgreSQL
RNF04	ORM	Para a comunicação da API com banco de dados utiliza-se a biblioteca PRISMA
RNF05	Documentação	Deverá dispor da documentação dos <i>endpoints</i> no formato do <i>Swagger</i> .
RNF06	Testabilidade	Para cada <i>endpoints</i> deverá haver um ou mais <i>request</i> de testes de integração no postman
RNF07	Validação de dados	API deverá validar os dados de entrada e utilizado o mecanismo express.
RNF08	Retorno de API	API terá os retornos de dados em forma de JSON com os seguintes <i>status</i> de resposta, 201 para cadastro, 400 para erro de processamento, 401 para não autorizados e 500 para erro de servidor

fonte: Acervo do Autor.

4.5 FUNCIONAMENTO DA API

Para representar de forma visual o funcionamento da API foram desenvolvidos alguns diagramas de sequência representando as classes da API. Através destes diagramas, é apresentado a sequência de passos que a requisição percorre quando chega na API até finalizar seu processo. Na Figura 8, é apresentado um diagrama demonstrando a sequência de passos da requisição de *login* dentro da API, bem como as classes nas quais ela aciona.

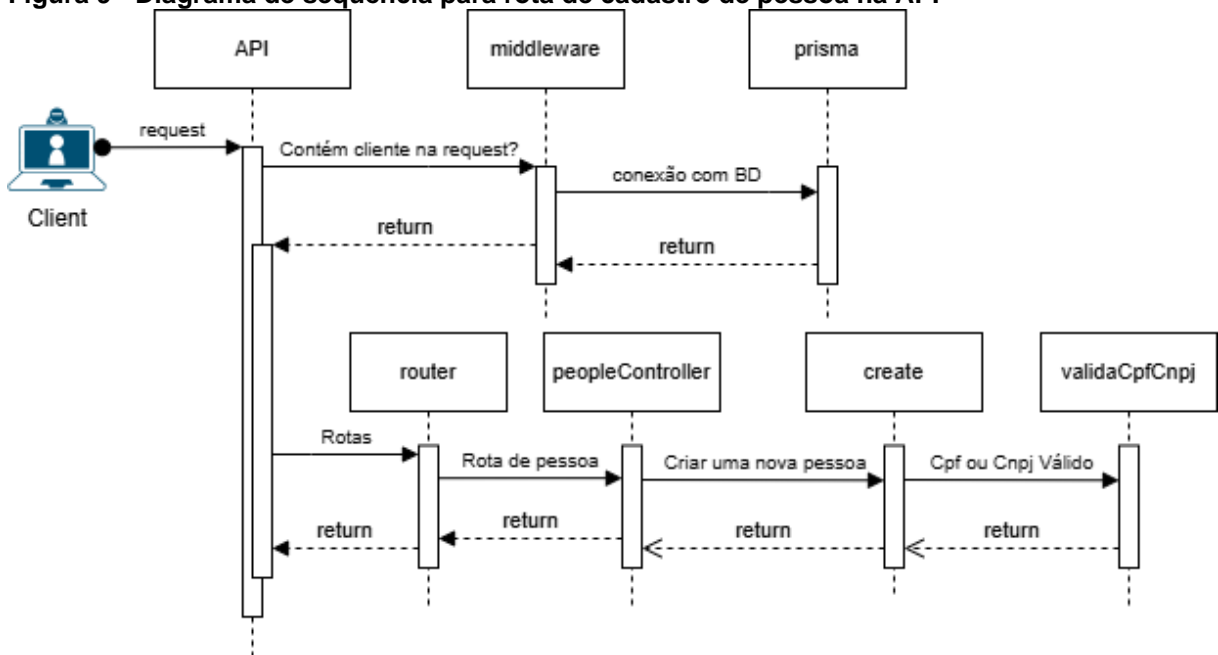
Figura 8 - Diagrama de seqüência de login na API



Fonte: Acervo do Autor.

É demonstrado na Figura 9, o diagrama que ilustra a seqüência de passos da requisição para cadastrar uma nova pessoa. As rotas de acesso à esta classe “Pessoa”, são as rotas de admin, professor e aluno, ambas as rotas acionam esta classe para fazer o devido tratamento solicitado.

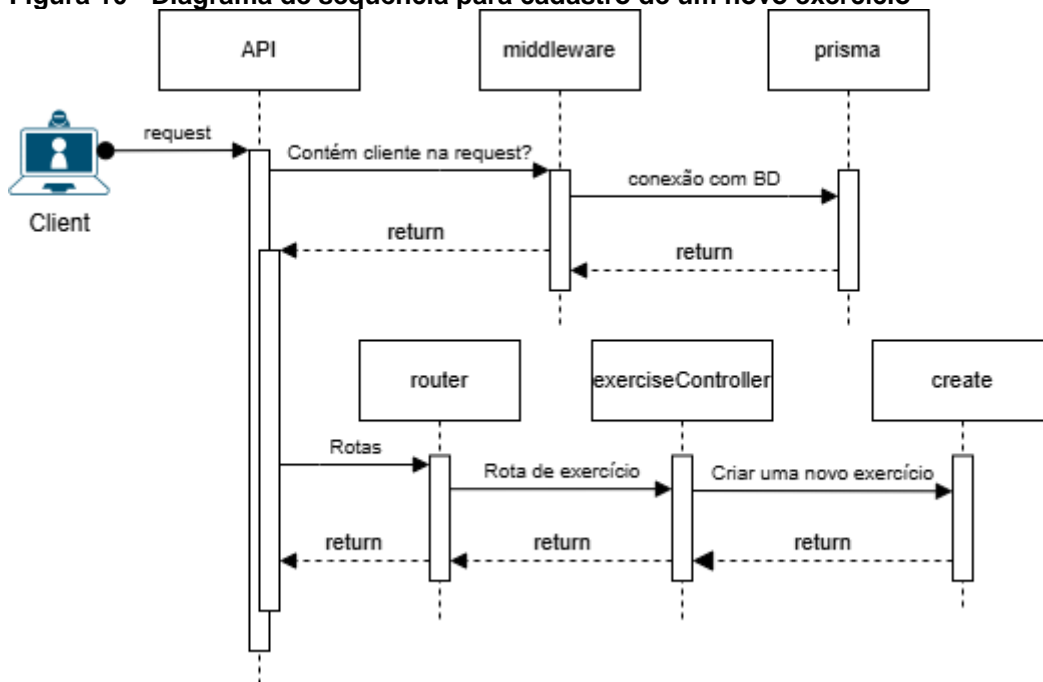
Figura 9 - Diagrama de seqüência para rota de cadastro de pessoa na API



Fonte: Acervo do Autor.

O diagrama que ilustra a seqüência de passos da requisição para cadastrar um novo exercício, é apresentado na Figura 10, nele observamos as classes que são acionadas para processar os dados enviados nesta requisição.

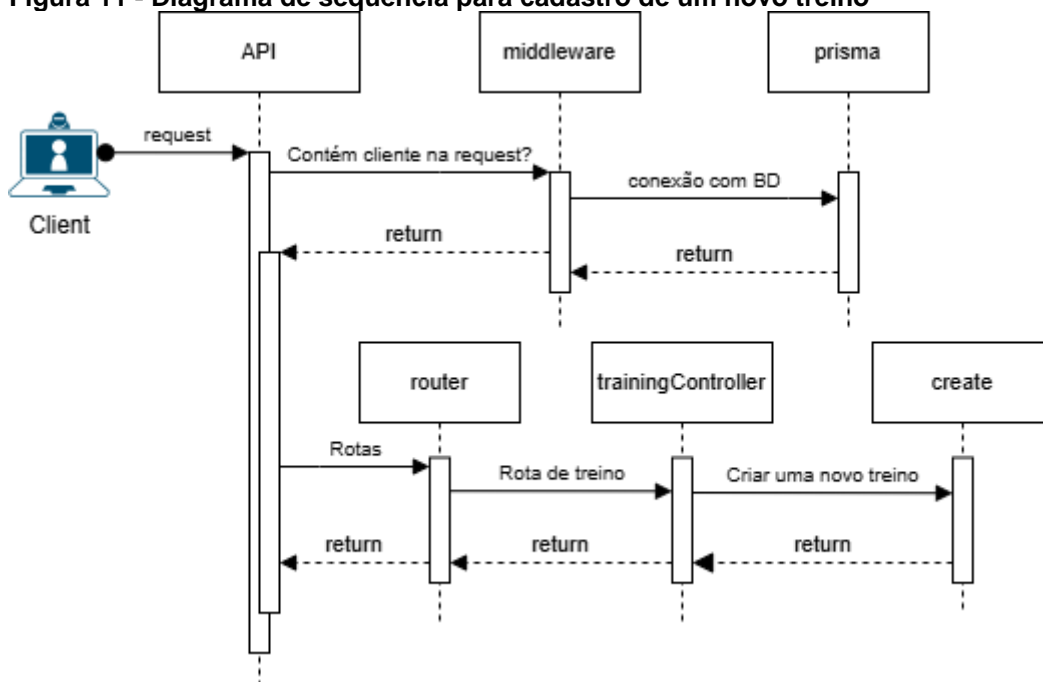
Figura 10 - Diagrama de sequência para cadastro de um novo exercício



Fonte: Acervo do autor.

No diagrama que ilustra a sequência de passos da requisição para cadastrar um novo treino, é apresentado na Figura 11. Nele é observado a sequência de classes que a requisição acessa quando é acionada.

Figura 11 - Diagrama de sequência para cadastro de um novo treino

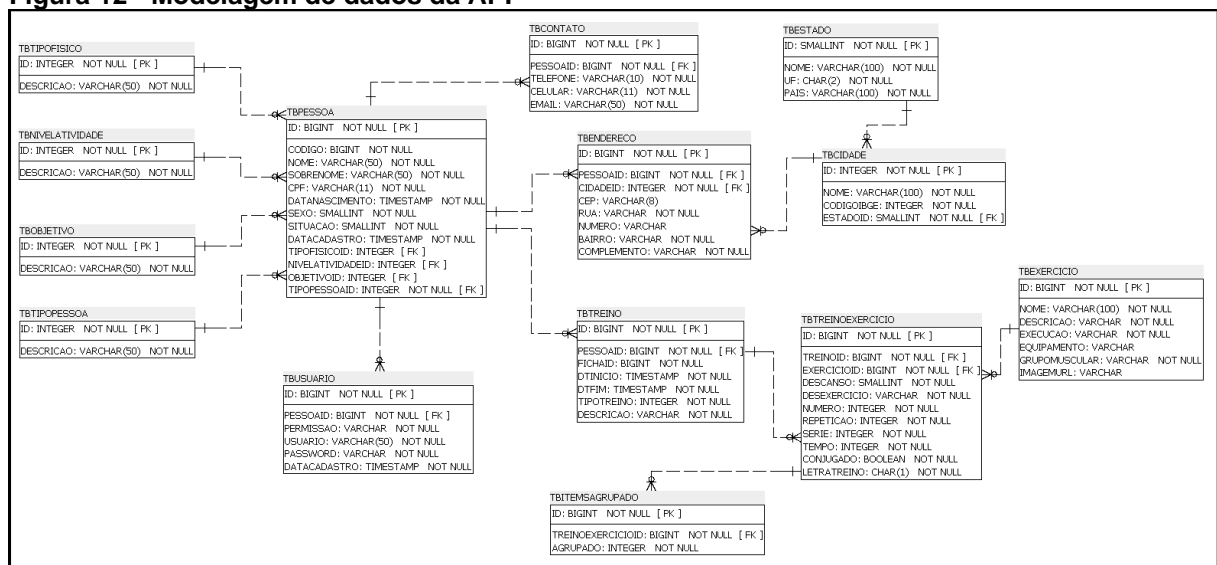


Fonte: Acervo do autor.

4.6 MODELAGEM DE DADOS

A modelagem de dados consiste em elaborar uma representação que estabelece a forma de como os dados de um determinado sistema serão estruturados e armazenados. Na Figura 12, é apresentada uma modelagem de dados onde representa a estrutura de como os dados da API serão organizados e armazenados. Ela define as entidades de dados que são as tabelas, detalhando os atributos que cada uma contém, ou seja, as colunas das tabelas e estabelece os relacionamentos entre as entidades, fazendo estes por meios das chaves primarias e estrangeiras. Este modelo foi projetado para ter escalabilidade e eficiência. Ainda possui uma estrutura bem definida, facilitando a manutenção e expansão futuras, mantendo a organização e a qualidade dos dados armazenados.

Figura 12 - Modelagem de dados da API

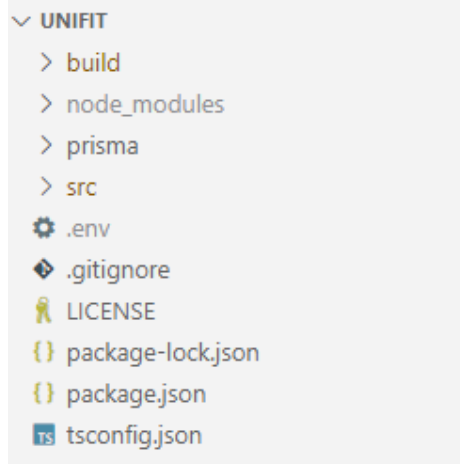


Fonte: Acervo do Autor.

4.7 DESENVOLVIMENTO

Com o objetivo de criar uma solução que integrasse com eficiência, segurança, flexibilidade e separada da aplicação *frontend*, foi desenvolvido uma API *backend* que recebe requisições para processamento de dados. Para este desenvolvimento foram utilizados o NodeJs e suas bibliotecas. O projeto foi implementado com modelo monolito, mas desenvolvido cada modulo separadamente dentro do projeto. Na Figura 13, é apresentada a estrutura de pastas do projeto da API.

Figura 13 - Estrutura principal do projeto



Fonte: Acervo do autor.

O processo da API está desenvolvido dentro da pasta SRC, nela desenvolvemos a estrutura dos processos de dados como o *server* que é o ponto de partida de acesso, *login*, rotas, controladores, *middlewares* entre outras classes. O arquivo principal da API é o *server.ts*, nele configuramos todas as informações necessárias para que funcione corretamente. O primeiro processo que ela faz quando chega uma chamada é acionar uma *middleware* que faz a verificação se existe um cliente sendo informado na requisição, caso exista ele chama uma terceira classe que faz a comunicação do PRISMA com o banco de dados para o cliente informado, caso não exista o cliente, é devolvido um erro 400 com a mensagem de “Cliente não informado”. Na Figura 14 é apresentado o código do *middleware*.

Figura 14 - Classe *middleware*

```
function middleware(req: Request, res: Response, next: NextFunction) {
  console.log('verificando cliente na requisição');
  const schema = req.query.schema as string;
  console.log('Cliente request: ' + schema);
  if (!schema) {
    return res.status(400).json({ error: 'Cliente não definido...' });
  };
  req.prisma = getPrismaClient(schema);
  next();
}
```

Fonte: Acervo do autor.

Após o processo do *middleware* ter obtido sucesso, volta para a classe principal e se a rota for *login* acessa a classe chamada “Login”. Nela existe o método

“validacao”, neste processo a API recebe o usuário e senha informados na requisição, na sequência busca no banco de dados as informações deste usuário, e caso este usuário exista, aciona uma função chamada “compararSenha” que está dentro de outra classe chamada “auth” passando como parâmetros a senha informada e a senha salva no banco de dados.

Caso a comparação da senha obtenha sucesso, é convertido os dados do usuário para uma propriedade de usuário logado, gerado *token* dentro da classe “auth”, atribuído os dados do usuário e o *token* na propriedade de retorno e devolvido a requisição com os dados do *login*. Na Figura 15 é apresentado o código de como foi desenvolvido a classe de *login*.

Figura 15 - Código da classe de *login*

```
export default class Login{
  static async validacao(req: Request, res: Response){
    try{
      console.log("Fazendo login");
      const { user, password } = req.body as LoginDTO;
      const usuario = await req.prisma.usuario.findFirst({ ...
    });
    if(usuario) {
      console.log("Autenticação para o usuário "+ usuario?.pessoa.nome + " "+usuario?.pessoa.sobrenome);
      if(await auth.compararSenha(password, usuario.password)) {
        const result = converteUsuario(usuario);
        const token = auth.gerarToken(result);
        const logado: UsuarioLogadoDTO = {
          id: usuario.id,
          nome: result.nome,
          tipo: result.permissao,
          token: token
        }
        return res.status(200).json(logado);
      } else {
        return res.status(401).json({msg: "Usuário ou senha invalido."});
      }
    } else {
      return res.status(401).json({msg: "Usuário ou senha invalido."});
    }
  } catch (error){
    console.log(error)
    return res.status(500).json({msg: 'Houve uma falha crítica no servidor, tente novamente em alguns instantes'});
  } finally {
    await req.prisma.$disconnect();
  }
};
```

Fonte: Acervo do autor.

Seguindo o fluxo do desenvolvimento, se a rota acessada for uma rota diferente de “login”, a API aciona o método “validaToken” que pertence a classe “auth” que é responsável por verificar se o *token* foi informado e validar sua autenticidade. Após a validação do *token* a requisição é encaminhada para uma classe chamada “router” que organiza as rotas e direciona a requisição para a rota correspondente.

O arquivo de rotas gerencia todas as rotas da API e é responsável por distribuir as requisições de acordo com o solicitado. Entre as rotas que estão disponíveis na API estão as rotas de pessoas, treinos e exercícios. A rota de pessoas está distribuída de acordo com o tipo de pessoa como por exemplo rota de professor e rota de aluno, ambas que apontam para a mesma classe de processamento de pessoa.

As rotas foram implementadas por métodos de processamento padrão: GET, POST, PUT e DELETE, cobrindo todas as operações necessárias que a API necessita. Além disso, as rotas foram estruturadas de forma modular, permitindo que cada rota tenha seu próprio conjunto de controladores, simplificando a manutenção e a adição de novas funcionalidades. Na Figura 16 são apresentadas as rotas da API.

Figura 16 - Classe de rotas da API

```

/*****Rota de Admin*****/
router.post('/admin', createPeople);
router.get('/admin', selectPeople);
router.put('/admin', updatePeople);
router.delete('/admin', deletePeople);

/*****Rota de Professor*****/
router.post('/professor', createPeople);
router.get('/professor', selectPeople);
router.put('/professor', updatePeople);
router.delete('/professor', deletePeople);

/*****Rota de Aluno*****/
router.post('/aluno', createPeople);
router.get('/aluno', selectPeople);
router.put('/aluno', updatePeople);
router.delete('/aluno', deletePeople);

/*****Rota de Exercício*****/
router.post('/exercico', createExercise);
router.get('/exercico', selectExercise);
router.put('/exercico', updateExercise);
router.delete('/exercico', deleteExercise);

/*****Rota de Treino*****/
router.post('/treino', createTreino);
router.get('/treino', selectTreino);
router.put('/treino', updateTreino);
router.delete('/treino', deleteTreino);

/*****Rota de Usuário*****/
router.get('/usuario', selectUser);
router.put('/usuario', updateUser);
router.delete('/usuario', deleteUser);

```

Fonte: Acervo do autor.

Usando a rota de professor como exemplo, o fluxo da API é organizado de acordo com o tipo de requisição. Utilizando a requisição do tipo POST para a criação de um novo cadastro de professor, é acionado um controlador chamado “peopleController”, que foi desenvolvido de forma genérica para que possa processar os dados de qualquer tipo de pessoa na API, sendo elas de administradores, professores e alunos.

O controlador recebe os dados enviado no corpo da requisição que contém as informações do professor a ser cadastrado e aciona a classe que fará o processamento destes dados e inserção no banco. Na Figura 17 é apresentado o código desenvolvido na classe controladora.

Figura 17 - Classe controladora de cadastro de professor

```

async function createPeople(req: Request, res: Response) {
  try {
    const pessoaDTO = req.body;
    const create = await CreatePeople(req.prisma, pessoaDTO, req, res);
    return create;
  } catch (error) {
    console.log(error);
    return res.status(500).json({msg: 'Houve uma falha crítica no servidor, tente novamente em alguns instantes'});
  } finally {
    console.log('Desconectando conexão do prisma');
    req.prisma.$disconnect();
    console.log('Prisma desconectado com sucesso');
  }
}

```

Fonte: Acervo do autor.

Após acionar o método “CreatePeople” que fica na classe “create”, que é responsável por realizar toda a lógica de validação e tratamento dos dados recebidos para o cadastro de uma pessoa. Nela, são verificadas as informações obrigatórias, como a validade do cpf, a presença de um e-mail válido e o preenchimento correto de todos os campos para o cadastro.

Com base na rota que foi acessada a classe de cadastro de pessoa é preenchido o tipo de pessoa, que neste caso de exemplo é o professor, garantindo assim que o tipo da pessoa seja cadastrado de acordo com o que se espera. Após validar e processar as informações, é utilizado o PRISMA para interagir com o banco de dados e persistir os dados nas tabelas.

Além disso, o desenvolvimento da API utiliza práticas de reuso de código, o que permite que o mesmo processo seja usado para diferentes tipos de pessoas, como administradores, professores e alunos. Essa prática garante consistência e eficiência, evitando redundâncias.

Na Figura 18 é apresentado o código da classe “createPeople”, que contém como o processo está estruturado, desde as validações inicial até a inserção dos dados no banco.

Figura 18 - Classe de processamento para cadastro de pessoa (admin, professor e aluno)

```

async function CreatePeople(prisma: PrismaClient, pessoaDTO: PessoaDTO, req: Request, res: Response) {
  try {
    console.log('Validando cpf');
    if(!await getValidaCpfCnpj(pessoaDTO.cpf)){ ...
    };
    console.log('CPF valido');
    console.log('Valida se tem email');
    if(!pessoaDTO.contato?.email){ ...
    };
    console.log('Pegando o tipo da pessoa');
    let tipoPessoa = 0;
    let permissao = '';
    const rota = req.route.path as string;
    switch (rota){
      case '/admin':
        tipoPessoa = 1;
        permissao = 'admin';
      case '/professor':
        tipoPessoa = 2;
        permissao = 'professor';
        break;
      case '/aluno':
        tipoPessoa = 3;
        permissao = 'aluno';
        break;
      default:
        return res.status(400).json({msg: 'Não foi possível identificar o tipo da pessoa.'});
    };
    console.log('Abrindo transação com o banco de dados para persistencia dos dados');
    const response = await prisma.$transaction(async (prismaTransaction) => { ...
    });
    if(response){
      console.log('Novo cadastro realizado com sucesso.');
```

```

      return res.status(201).json({msg: 'Novo cadastro realizado com sucesso.'})
    } else {
      console.log('Houve um erro ao realizar o cadastro.');
```

```

      return res.status(400).json({msg: 'Houve um erro ao realizar o cadastro.'})
    };
  } catch (error) {
    console.log(error);
    return res.status(400).json({msg: `Houve um erro ao realizar o cadastro. ${error}`})
  }
}

```

Fonte: Acervo do autor.

O mesmo fluxo segue para a rota de exercícios, na Figura 19 é apresentado o código da classe controladora de exercício para cadastro de um novo exercício.

Figura 19 - Classe controladora de cadastro de exercício

```

async function createExercise(req: Request, res: Response){
  try {
    const pessoaDTO = req.body;
    const create = await CreateExercise(req.prisma, pessoaDTO, req, res);
    return create;
  } catch (error) {
    console.log(error);
    return res.status(500).json({msg: 'Houve uma falha crítica no servidor, tente novamente em alguns instantes'});
  } finally {
    console.log('Desconectando conexão do prisma');
    req.prisma.$disconnect();
    console.log('Prisma desconectado com sucesso');
  }
}

```

Fonte: Acervo do autor.

Na Figura 20, é apresentado o código da classe que faz o processamento dos dados de criação de um novo exercício e a persistência no banco de dados.

Figura 20 - Classe de processamento para cadastro de exercício

```

async function CreateExercise(prisma: PrismaClient, exercicioDTO: ExercicioDTO, req: Request, res: Response){
  try {
    await prisma.$transaction(async (prismaTransaction) => {
      return prismaTransaction.exercicio.create({ data: exercicioDTO });
    });
    return res.status(201).json({msg: "Novo exercício cadastrado com sucesso."});
  } catch (error) {
    console.log(error);
    return res.status(400).json({msg: `Houve um erro ao realizar o cadastro do exercício. ${error}`});
  }
}

```

Fonte: Acervo do autor.

Seguindo para a rota de treino, o fluxo permanece igual, na Figura 21 é apresentado a classe controladora para o tratamento de cadastro de um treino.

Figura 21 - Classe controladora de cadastro de treino

```

async function createTraining(req: Request, res: Response){
  try {
    const pessoaDTO = req.body;
    const create = await createTraining(req.prisma, pessoaDTO, req, res);
    return create;
  } catch (error) {
    console.log(error);
    return res.status(500).json({msg: 'Houve uma falha crítica no servidor, tente novamente em alguns instantes'});
  } finally {
    console.log('Desconectando conexão do prisma');
    req.prisma.$disconnect();
    console.log('Prisma desconectado com sucesso');
  }
}

```

Fonte: Acervo do autor.

Na Figura 22 é apresentado o código da classe de cadastro do treino no banco de dados.

Figura 22 - Classe de processamento para cadastro de treino

```

async function CreateTraining(prisma: PrismaClient, treinoDTO: TreinoDTO, req: Request, res: Response){
  try {
    await prisma.$transaction(async (prismaTransaction) => {
      return prismaTransaction.treino.create({ data: treinoDTO });
    });

    return res.status(201).json({msg: "Novo treino cadastrado com sucesso."});
  } catch (error) {
    console.log(error);
    return res.status(400).json({msg: `Houve um erro ao realizar o cadastro do treino. ${error}`});
  }
}

```

Fonte: Acervo do autor.

Todas as outras rotas da API seguem o mesmo padrão desta apresentada, existe a classe controladora, e a função que processa a informação, todas separadas de acordo com a finalidade desejada, como classe para criar apresentada, classe para atualizar os dados, classe para selecionar os dados e classe para deletar conforme apresentado a estrutura na Figura 23.

Figura 23 - Estrutura das classes de processamento de dados de pessoa

```

src
├── > controller
├── > functions
├── > exercise
├── > people
│   ├── TS create.ts
│   ├── TS delete.ts
│   ├── TS select.ts
│   └── TS update.ts

```

Fonte: Acervo do autor.

4.8 TESTES COM POSTMAN

Com a utilização da ferramenta postman, para atingir os objetivos de verificar a integridade e a conformidade dos retornos recebidos da API, criamos em alguns dos *endpoints*, *scripts* de teste de retorno, onde foi possível a partir dos testes, validar as expectativas dos retornos das requisições.

4.8.1 Teste de *LOGIN*

O primeiro teste foi realizado na requisição de *login*, onde foi validado a estrutura de retorno para quando a requisição de *login* é válida. Na Figura 24 é apresentado o *script* de teste realizado para validar este retorno. A função deste teste é validar a estrutura de retorno do JSON, para que o teste seja valido, o retorno precisa conter a seguinte estrutura:

```
{
  "id": ID do Usuário,
  "nome": "Nome do Usuário",
  "tipo": "professor - Função",
  "token": "Token de acesso a API"
}
```

Figura 24 - Teste de *login* - Validação de JSON de retorno.

The screenshot shows the Postman interface with the URL `http://127.0.0.1:3333/login?schema=unidavi`. The 'Scripts' tab is active, displaying the following JavaScript code:

```
1 //valida a estrutura de retorno do json
2 pm.test("Verifica se o login retorna a estrutura esperada", () => {
3   const statusCode = pm.response.code;
4   if (statusCode === 200) {
5     const responseData = pm.response.json();
6     pm.expect(responseData).to.have.all.keys("id", "nome", "tipo", "token");
7     const jwtPattern = /^[A-Za-z0-9-_.]+.[A-Za-z0-9-_.]+.[A-Za-z0-9-_.]+$/;
8     pm.expect(responseData.token).to.be.a("string").and.match(jwtPattern);
9   }
10  });
```

Fonte: Acervo do autor.

O segundo *script* também realizado com o *login*, tendo a finalidade de validar se *token* de autenticação retornado na requisição, contém a estrutura JWT. Na Figura 25, podemos verificar o *script* desenvolvido para validar o retorno do *token*.

Figura 25 - Teste de *login* - Validação do *token* JWT

The screenshot shows a REST client interface with the URL `http://127.0.0.1:3333/login?schema=unidavi`. The 'Scripts' tab is active, displaying the following JavaScript code:

```

12 //valida se o token tem a estrutura JWT
13 pm.test("Verifica a estrutura do token", () => {
14     const statusCode = pm.response.code;
15     if (statusCode === 200) {
16         const responseData = pm.response.json();
17         const jwtPattern = /^[A-Za-z0-9-_\]+\. [A-Za-z0-9-_\]+\. [A-Za-z0-9-_\]+$/;
18         pm.expect(responseData.token).to.be.a("string").and.match(jwtPattern);
19     }
20 });

```

Fonte: Acervo do autor.

O terceiro *script* de teste, verifica também no *endpoints* de *login* o retorno de quando a requisição retorna não autorizado, este retorno precisa conter apenas uma mensagem informando que o usuário ou a senha está inválido. Verifique na Figura 26, o *script* desenvolvido para validar este retorno.

Figura 26 - Teste de *login* - Validação de retorno não autorizado

The screenshot shows a REST client interface with the URL `http://127.0.0.1:3333/login?schema=unidavi`. The 'Scripts' tab is active, displaying the following JavaScript code:

```

22 //valida se não autorizado
23 pm.test('Verifica se não foi autorizado', () => {
24     const statusCode = pm.response.code;
25     if (statusCode === 401) {
26         const responseData = pm.response.json();
27         pm.expect(responseData).to.equal("Usuário ou senha inválido.");
28     }
29 });

```

Fonte: Acervo do autor.

4.8.2 Teste de professor

O primeiro teste realizado no *endpoints* de cadastro para um novo professor, válida se o retorno da operação é o código 201 e a mensagem padrão de cadastro é o que retorna na requisição. Na Figura 27, é apresentado o *script* desenvolvido para validar esta requisição.

Figura 27 - Teste de sucesso para cadastro de professor.

The screenshot shows a REST client interface with the URL `http://127.0.0.1:3333/Professor?schema=unidavi`. The 'Scripts' tab is active, displaying the following JavaScript code:

```

1 //teste de status de sucesso
2 pm.test('Teste de mensagem de sucesso - status 201', () => {
3     const statusCode = pm.response.code;
4     if (statusCode === 201) {
5         const { msg } = pm.response.json();
6         pm.expect(msg).to.equal("Novo cadastro realizado com sucesso.");
7     }
8 });
  
```

Fonte: Acervo do autor.

O segundo teste, verifica os erros com *status* 400 – *Bad Request*, onde a mensagem de retorno contém algumas frases padrão de erro. É apresentado na Figura 28, o *script* desenvolvido para validar a requisição de cadastro para um novo professor com retorno de erro 400.

Figura 28 - Teste de *status* 400 no cadastro de professor

The screenshot shows the same REST client interface with the URL `http://127.0.0.1:3333/Professor?schema=unidavi`. The 'Scripts' tab is active, displaying the following JavaScript code:

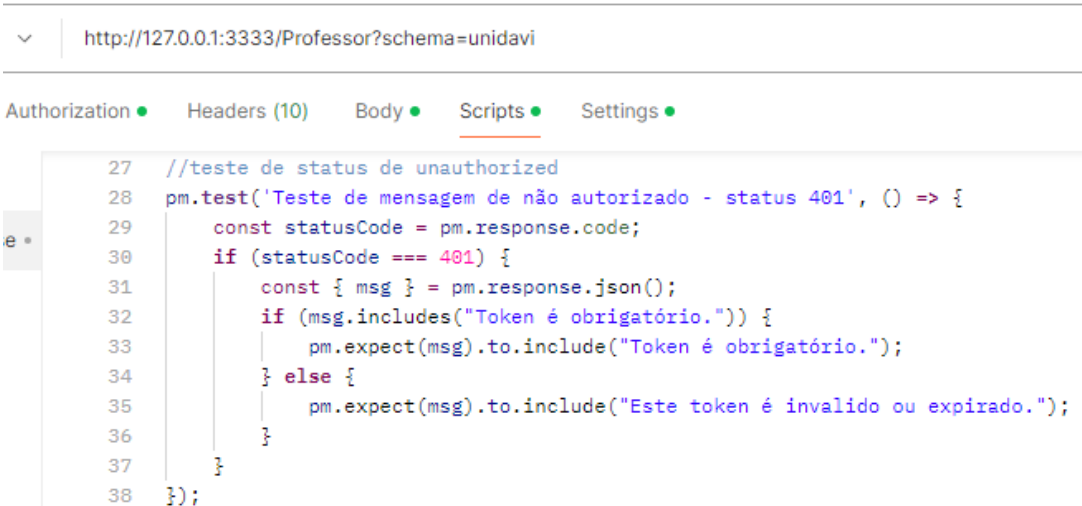
```

10 //teste de status de bad request
11 pm.test('Teste de mensagem de erro - status 400', () => {
12     const statusCode = pm.response.code;
13
14     if (statusCode === 400) {
15         const { msg } = pm.response.json();
16
17         if (msg.includes("Unique constraint failed on the fields: (`cpf`)")) {
18             pm.expect(msg).to.include("Unique constraint failed on the fields: (`cpf`)");
19         } else if (msg.includes("Unique constraint failed on the fields: (`email`)")) {
20             pm.expect(msg).to.include("Unique constraint failed on the fields: (`email`)");
21         } else {
22             console.log("Outro erro:", msg);
23         }
24     }
25 });
  
```

Fonte: Acervo do autor.

Para o teste de não autorizado, foi criado um *script* que valida quando o *token* de autenticação não está correto ou quando não está informado. Na Figura 29, é apresentado o *script* de validação da requisição de cadastro de professor, quando não autorizado.

Figura 29 - Teste de não autorizado no cadastro de professor.



```

27 //teste de status de unauthorized
28 pm.test('Teste de mensagem de não autorizado - status 401', () => {
29     const statusCode = pm.response.code;
30     if (statusCode === 401) {
31         const { msg } = pm.response.json();
32         if (msg.includes("Token é obrigatório.")) {
33             pm.expect(msg).to.include("Token é obrigatório.");
34         } else {
35             pm.expect(msg).to.include("Este token é inválido ou expirado.");
36         }
37     }
38 });

```

Fonte: Acervo do autor.

4.8.3 Teste de aluno

O primeiro teste realizado no *endpoints* de cadastro para um novo aluno, válida se o retorno da operação é o código 201 e a mensagem padrão de cadastro é o que retorna na requisição. Na Figura 30, é apresentado o *script* desenvolvido para validar esta requisição.

Figura 30 - Teste de sucesso para cadastro de aluno

```

pm.test('Teste de mensagem de sucesso - status 201', () => {
    const statusCode = pm.response.code;
    if (statusCode === 201) {
        const { msg } = pm.response.json();
        pm.expect(msg).to.equal("Novo cadastro realizado com sucesso.");
    }
});

```

Fonte: Acervo do autor.

Para o teste de não autorizado, foi criado um *script* que valida quando o *token* de autenticação não está correto ou quando não está informado. Na Figura 31, é

apresentado o *script* de validação da requisição de cadastro de aluno, quando não for autorizado.

Figura 31 - Teste de não autorizado para cadastro de aluno

```
pm.test('Teste de mensagem de não autorizado - status 401', () => {
  const statusCode = pm.response.code;
  if (statusCode === 401) {
    const { msg } = pm.response.json();
    if (msg.includes("Token é obrigatório.")) {
      pm.expect(msg).to.include("Token é obrigatório.");
    } else {
      pm.expect(msg).to.include("Este token é inválido ou expirado.");
    }
  }
});
```

Fonte: Acervo do autor.

4.8.4 Teste de exercício

O primeiro teste realizado no *endpoints* de cadastro de exercício, válida se o retorno da operação é o código 201 e a mensagem padrão de cadastro é o que retorna na requisição. Na Figura 32, é apresentado o *script* desenvolvido para validar esta requisição.

Figura 32 - Teste de cadastro de exercício

```
pm.test('Teste de mensagem de sucesso - status 201', () => {
  const statusCode = pm.response.code;
  if (statusCode === 201) {
    const { msg } = pm.response.json();
    pm.expect(msg).to.equal("Novo cadastro realizado com sucesso.");
  }
});
```

Fonte: Acervo do autor.

Para o teste de não autorizado, foi criado um *script* que valida quando o *token* de autenticação não está correto ou quando não está informado. Na Figura 33, é apresentado o *script* de validação da requisição de cadastro de exercício, quando não for autorizado.

Figura 33 - Teste de não autorizado para cadastro de exercício

```

pm.test('Teste de mensagem de não autorizado - status 401', () => {
  const statusCode = pm.response.code;
  if (statusCode === 401) {
    const { msg } = pm.response.json();
    if (msg.includes("Token é obrigatório.")) {
      pm.expect(msg).to.include("Token é obrigatório.");
    } else {
      pm.expect(msg).to.include("Este token é inválido ou expirado.");
    }
  }
});

```

Fonte: Acervo do autor.

4.8.5 Teste de treino

O primeiro teste realizado no *endpoints* de cadastro de treino, válida se o retorno da operação é o código 201 e a mensagem padrão de cadastro é o que retorna na requisição. Na Figura 34, é apresentado o *script* desenvolvido para validar esta requisição.

Figura 34 - Teste de cadastro de treino

```

pm.test('Teste de mensagem de sucesso - status 201', () => {
  const statusCode = pm.response.code;
  if (statusCode === 201) {
    const { msg } = pm.response.json();
    pm.expect(msg).to.equal("Novo cadastro realizado com sucesso.");
  }
});

```

Fonte: Acervo do autor.

Para o teste de não autorizado, foi criado um *script* que valida quando o *token* de autenticação não está correto ou quando não está informado. Na Figura 35, é apresentado o *script* de validação da requisição de cadastro de treino, quando não for autorizado.

Figura 35 - Teste de não autorizado para cadastro de treino

```
pm.test('Teste de mensagem de não autorizado - status 401', () => {  
  const statusCode = pm.response.code;  
  if (statusCode === 401) {  
    const { msg } = pm.response.json();  
    if (msg.includes("Token é obrigatório.")) {  
      pm.expect(msg).to.include("Token é obrigatório.");  
    } else {  
      pm.expect(msg).to.include("Este token é inválido ou expirado.");  
    }  
  }  
});
```

Fonte: Acervo do autor.

Com o desenvolvimento dos *scripts* de teste, qualquer alteração realizada na aplicação poderá ser facilmente testada em completude, ou seja, todos os *endpoints* da aplicação previamente configurados serão novamente validados onde, em caso de falha, poderão ser corrigidos.

5 CONSIDERAÇÕES FINAIS

O presente trabalho de conclusão de curso, apresentou uma pesquisa e desenvolvimento de uma camada de software *webservice* (API), para fazer o processamento de dados gerados pela academia, como dados de pessoas, exercícios e treinos entre outros a fim de gerar uma solução escalável, segura e com disponibilidade. Neste projeto foram utilizadas as tecnologias como NodeJs, TypeScript, PRISMA, PostgreSQL, Insomnia, Postman, que permitiram o desenvolvimento de uma aplicação mais leve e com processos desacoplados, possibilitando uma fácil distribuição da API, em caso de necessidade, deixando o sistema fácil de compreender, prezando pela manutenibilidade.

Para persistência dos dados da academia, foi desenvolvido uma modelagem de dados, baseada nos requisitos e informações adquiridas em uma academia na qual tem a necessidade de obter um sistema para gerenciar suas atividades, estes dados foram modelados com o aplicativo SQL Power Architect, e posterior criados através de *migrations* do PRISMA para criar o banco de dados dentro do PostgreSQL.

Foram criados *scripts* de teste utilizando o programa postman, para testar os retornos de dados e informações da API, garantindo sempre uma qualidade no processo da API.

Os requisitos funcionais e não funcionais estão descritos na seção de requisitos no tópico de desenvolvimento. Quanto à modelagem do banco de dados, está detalhada na seção modelagem de dados, onde é apresentado o diagrama entidade relacionamento com todas as tabelas e suas relações. O desenvolvimento da API, também está detalhado no tópico de desenvolvimento, mostrando que foram utilizadas as tecnologias de NodeJs, TypeScript, PRISMA e Express. Os testes dos *endpoints* da API, estão detalhados na seção de requisição com postman, onde é apresentado alguns testes criados para os *endpoints* da API.

Concluimos assim que os objetivos propostos no presente trabalho foram alcançados e considerando as limitações desta pesquisa, é apresentado algumas recomendações de desenvolvimento futuro em que estas sugestões contribuirão para o contínuo avanço da eficiência desta API.

5.1 RECOMENDAÇÕES DE ATUALIZAÇÕES FUTURAS

Como recomendações futuras para a melhoria deste projeto, propomos a implementação de novas funcionalidades e processos que aumentarão a capacidade do sistema e atenderá as demais necessidades da academia. Um exemplo notável seria o desenvolvimento de um módulo financeiro, que permitiria aos gestores da academia realizarem o controle completo do fluxo de caixa diretamente no sistema. Tudo isso centralizado em um único sistema, facilitando a administração e deixando a aplicação com maior eficiência.

Além disso, sugerimos a realização de melhorias contínuas nos processos já implementados, refinando e otimizando funcionalidades de acordo com os feedbacks dos usuários e necessidades da própria academia. Esses processos, que foram desenvolvidos com base na estrutura e nas necessidades específicas de academia, podem ser ajustados para atender às demandas de outros contextos que também trabalham na área de esporte como natação, crossfit, quadras de esportes e outros.

Como etapa estratégica, recomendamos testar o projeto na própria academia da UNIDAVI, permitindo assim, avaliar o sistema em um ambiente real, identificando possíveis ajustes e coletando insights dos usuários. A partir disso, é possível estruturar e consolidar o projeto, preparando-o para ser comercializável e utilizado no nicho de mercado de academias.

Por fim, é essencial investir em estratégias de divulgação e suporte técnico para garantir que o sistema seja bem recebido no mercado. O objetivo é transformar essa solução em uma ferramenta indispensável para academias, oferecendo inovação, praticidade e confiabilidade para os gestores e usuários.

REFERÊNCIAS

ALVES, William Pereira. **Banco de dados**. São Paulo: Erica, 2014. *Ebook*.

CARDOSO, Leandro da Conceição. **Frameworks back end**. São Paulo: Platos Soluções Educacionais, 2021. *Ebook*.

CARVALHO, Vinicius. **PostgreSQL**: banco de dados para aplicações web modernas. São Paulo: Casa do Código, 2017.

FERREIRA, Arthur Gonçalves. **Interface de programação de aplicações (API) e web services**. São Paulo: Platos Soluções Educacionais, 2021. *Ebook*.

GOTARDO, Reginaldo Aparecido. **Linguagens de Programação**. Rio de Janeiro: Seses, 2015.

INSOMMIA. **Documentação Insomnia**. Disponível em: <<https://insomnia.rest/>> Acesso em: 28 de setembro de 2024.

MDN, Web Docs. **Métodos de Requisição HTTP**. Disponível em: <<https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Methods>> Acesso em: 17 jun. 2024.

NODEJS. **NodeJs com TypeScript**. Disponível em: <<https://nodejs.org/pt/learn/getting-started/nodejs-with-typescript>> Acesso em: 17 jun. 2024.

OLIVEIRA, Cláudio Luís Vieira; ZANETTI, Humberto Augusto Piovesana. **NodeJs**: programe de forma rápida e prática. São Paulo: SRV Editora LTDA, 2021. *Ebook*.

PAULA FILHO, Wilson de Pádua. **Engenharia de Software**. Rio de Janeiro: Grupo GEN, 2019. *Ebook*.

PEREIRA, Caio Ribeiro. **Construindo APIs Rest com NodeJs**. São Paulo: Casa do Código, 2016.

POSTMAN. **Documentação Postman**. Disponível em: <<https://learning.postman.com/docs/introduction/overview/>> Acesso em: 23 de outubro de 2024.

PRISMA. **Documentação Prisma**. Disponível em: <<https://www.prisma.io/docs/orm/overview/introduction>> Acesso em: 17 jun. 2024.

SOMMERVILLE, Ian. **Engenharia de software**. São Paulo: Pearson, 2011.

SWAGGER. Disponível em: <<https://swagger.io/docs/>> Acesso em: 28 de setembro de 2024.

TIOBE, **The Software Quality Company**. Tiobe Index for June 2024. Disponível em: <<https://www.tiobe.com/tiobe-index/>> Acesso em: 17 jun. 2024.