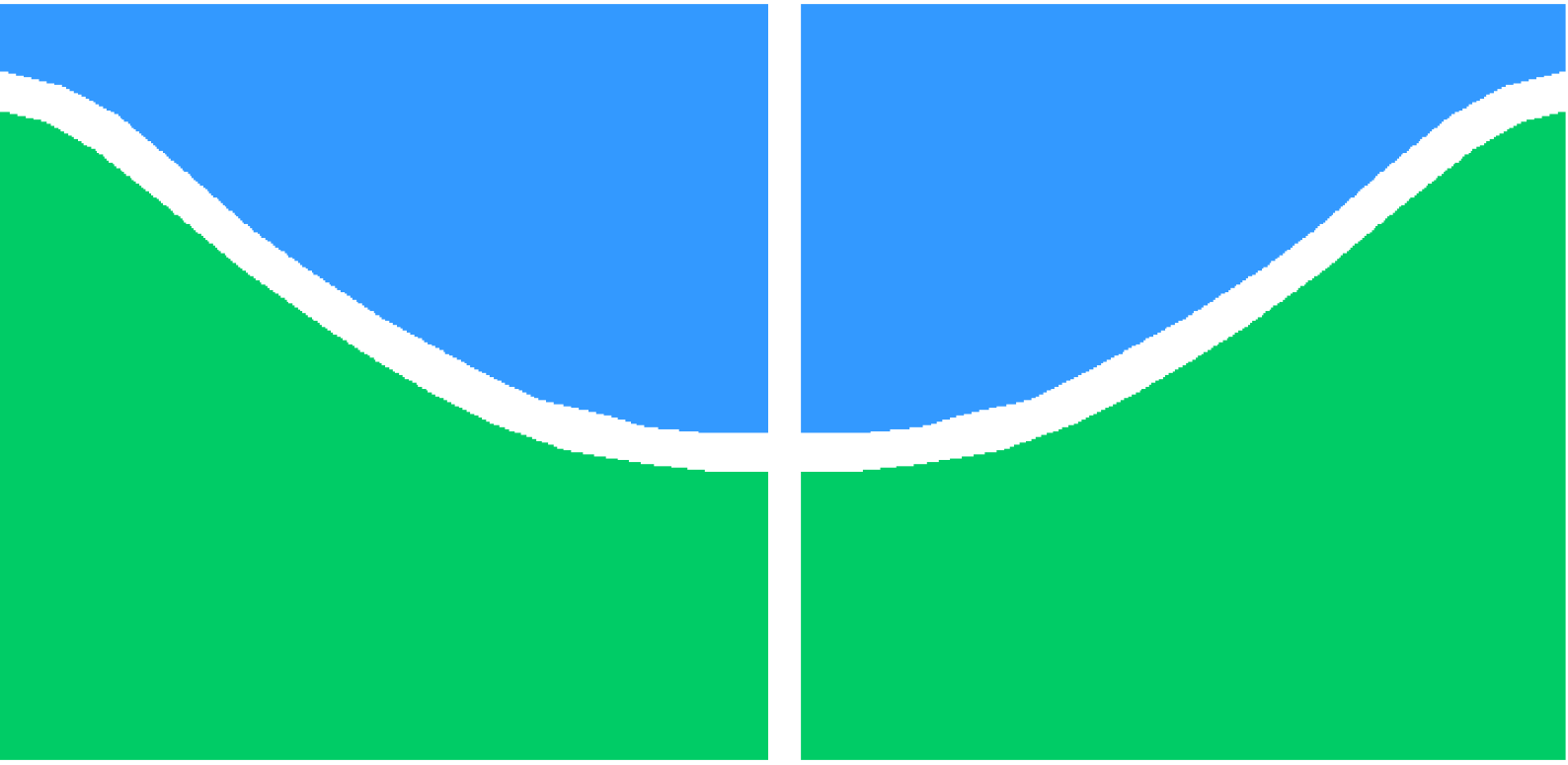


Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de Software

Arquitetura de microsserviços e consumo de APIs como pilares fundamentais em clubes de assinatura

Autor: Lucas Lima Ferraz e Nícalo Ribeiro

Orientador: Prof. Dr. Ricardo Matos Chaim

Brasília, DF

2024

Lucas Lima Ferraz e Nícalo Ribeiro

Arquitetura de microsserviços e consumo de APIs como pilares fundamentais em clubes de assinatura

Monografia submetida ao curso de graduação em Engenharia de Softwareda Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Ricardo Matos Chaim

Brasília, DF

2024

Lucas Lima Ferraz e Nícalo Ribeiro

Arquitetura de microsserviços e consumo de APIs como pilares fundamentais em clubes de assinatura/ Lucas Lima Ferraz e Nícalo Ribeiro. – Brasília, DF, 2024-
93 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Ricardo Matos Chaim

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2024.

1. Microsserviços. 2. API. I. Prof. Dr. Ricardo Matos Chaim. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Arquitetura de microsserviços e consumo de APIs como pilares fundamentais em clubes de assinatura

CDU 02:141:005.6

Lucas Lima Ferraz e Nícalo Ribeiro

Arquitetura de microsserviços e consumo de APIs como pilares fundamentais em clubes de assinatura

Monografia submetida ao curso de graduação em Engenharia de Softwareda Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 24 de setembro de 2024:

Prof. Dr. Ricardo Matos Chaim
Orientador

Prof. Dr. Vandor Roberto Vilardi Rissoli
Convidado 1

Prof. Dr. Giovanni Almeida Santos
Convidado 2

Brasília, DF
2024

Resumo

Este trabalho tem como objetivo desenvolver uma plataforma *web* que centralize soluções independentes de *software* de investimento, utilizando uma arquitetura com base em microsserviços e integração com APIs externas. A partir desse estudo, busca-se demonstrar como uma arquitetura de microsserviços pode aumentar a escalabilidade, a flexibilidade e a manutenibilidade de uma plataforma de clube de assinatura. O projeto visa evidenciar que empresas, desde *startups* até grandes corporações, podem adotar essas tecnologias para otimizar a entrega de serviços e melhorar a qualidade do código e da manutenção. O desenvolvimento envolve a criação de uma arquitetura de APIs robusta e escalável, além da implementação de microsserviços que funcionem de maneira independente e integrada. O potencial de escalabilidade da plataforma será avaliado com o intuito de incluir novas soluções de investimento e expandir a base de usuários. Portanto, este projeto propõe-se a explorar a aplicação prática dessas abordagens em um contexto de clube de assinatura, focando em benefícios como segurança, eficiência e qualidade de *software*.

Palavras-chaves: Microsserviços. API. Clube de assinatura.

Abstract

This project aims to develop a web platform that centralizes independent investment software solutions, utilizing a microservices architecture and external API integration. Through this study, the goal is to demonstrate how a microservices architecture can enhance the scalability, flexibility, and maintainability of a subscription club platform. The project seeks to show that companies, from startups to large corporations, can adopt these technologies to optimize service delivery and improve code quality and maintenance. The development includes creating a robust and scalable API architecture, as well as implementing microservices that function independently and in an integrated manner. The platform's scalability potential will be assessed to include new investment solutions and expand the user base. In this way, the project aims to explore the practical application of these approaches within a subscription club context, focusing on benefits such as security, efficiency, and software quality.

Key-words: Microservices. API. Subscription club.

Lista de Figuras

Figura 1 – API Rest. 23
Figura 2 – Caso de uso da InvestMinds. 49
Figura 3 – Estrutura de pastas do *frontend* 54
Figura 4 – Estrutura de pastas dos componentes 55
Figura 5 – Estrutura de pastas do roteamento 56
Figura 6 – Estrutura de pastas do *backend* 60
Figura 7 – Estrutura das pastas de domínio 60
Figura 8 – Fases do RUP. 66
Figura 9 – Arquitetura de microsserviços. 68
Figura 10 – Tela Home. 85
Figura 11 – Tela Sobre. 86
Figura 12 – Tela Cadastro. 86
Figura 13 – Tela Login. 87
Figura 14 – Navbar usuário. 87
Figura 15 – Navbar dono de solução. 88
Figura 16 – Component Chatbot. 88
Figura 17 – Tela criar publicação. 89
Figura 18 – Preview publicação. 89
Figura 19 – Planos disponíveis. 90
Figura 20 – Assinaturas disponíveis. 90
Figura 21 – Assinatura desenvolvida. 91
Figura 22 – Lista de softwares. 91
Figura 23 – Cadastro de produtos. 92
Figura 24 – Cadastro de usuários. 92
Figura 25 – Modelagem do Banco de Dados. 93

Lista de tabelas

Tabela 1 – Diferenças entre APIs SOAP, RPC, WebSocket e REST 22
Tabela 2 – Tabela resumo da metodologia de pesquisa adotada 43
Tabela 3 – Valores Ágeis 45
Tabela 4 – Vantagens e Desafios do modelo Incremental 46
Tabela 5 – Requisitos Funcionais da Plataforma InvestMinds 47
Tabela 6 – Requisitos não funcionais da plataforma InvestMinds. Fonte: Autores . 48

Lista de Códigos

1 Redux e React-Router 56

Lista de abreviaturas e siglas

ANPD	Autoridade Nacional de Proteção de Dados
API	Application Programming Interface
BFF	Backend for Frontend
BSON	Binary JSON
ChatGPT	Chat Generative Pre-trained Transformer
FGA	Faculdade do Gama
HMR	Hot Module Replacement
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IoT	Internet of Things
JSON	JavaScript Object Notation
LGPD	Lei Geral de Proteção de Dados
mTLS	Mutual Transport Layer Security
NoSQL	Not only SQL
REST	Representational State Transfer
RF	Requisito Funcional
RNF	Requisito Não Funcional
RPC	Remote Procedure Call
RUP	Rational Unified Process
SDK	Software Development Kit
SEBRAE	Serviço Brasileiro de Apoio às Micro e Pequenas Empresas
SOAP	Simple Object Access Protocol
SQL	Structured query language
TCC	Trabalho de Conclusão de Curso
UnB	Universidade de Brasília

Sumário

1 INTRODUÇÃO 17
1.1 Contexto 17
1.2 Questão de Pesquisa 18
1.3 Justificativa 18
1.4 Objetivos 19
1.4.1 Objetivo Geral 19
1.4.2 Objetivos Específicos 19
1.5 Organização do Trabalho 20

2 REFERENCIAL TEÓRICO 21
2.1 Fundamentos de API e Microsserviços 21
2.1.1 API 21
2.1.2 Consumo e criação de APIs 25
2.1.3 Microsserviços 28
2.1.4 Padrões e Arquiteturas de microsserviços 30
2.1.5 Arquitetura BFF 31
2.1.6 Desenvolvimento de microsserviços com Node.js e React 32
2.1.7 Bancos de dados relacionais e não-relacionais 33
2.1.8 MongoDB e Microsserviços 35
2.2 Evolução e Inovação 35
2.2.1 Impacto nos Clubes de Assinatura 35
2.2.2 Desafios e Soluções 36
2.2.3 Tendências Futuras e Inovações 37
2.2.4 Impacto na Maturidade de Empresas 38
2.3 Clubes de assinatura 39

3 METODOLOGIA 43
3.1 Metodologia de Pesquisa 43
3.2 Metodologia de Desenvolvimento de Software 44
3.2.1 Metodologia Ágil 44
3.2.2 Modelo incremental 45
3.3 Requisitos 45
3.3.1 Elicitação dos requisitos 46
3.3.2 Priorização dos Requisitos 49

4 DESENVOLVIMENTO 51

4.1 Desenvolvimento da Plataforma . . . 51
4.1.1 Frontend . . . 51
4.1.2 Arquitetura do Frontend . . . 53
4.1.3 Boas Práticas de Desenvolvimento . . . 54
4.1.4 Roteamento e Gerenciamento de Acesso . . . 55
4.1.5 Backend for Frontend (BFF) . . . 57
4.1.6 Backend . . . 57
4.1.7 Banco de Dados . . . 60
4.1.8 Segurança . . . 64
4.1.9 Infraestrutura e Deployment . . . 64
4.1.10 Testes e Qualidade de Código . . . 65
4.1.11 Integração com APIs Externas . . . 65
4.2 Gerenciamento . . . 65
4.2.1 Documentação . . . 66
4.3 Objetivos alcançados . . . 67
4.3.1 Arquitetura de APIs robusta . . . 67
4.3.2 Arquitetura de microsserviços eficiente . . . 67
4.3.3 Melhores práticas de desenvolvimento . . . 68
4.3.4 Potencial de escalabilidade . . . 68
4.3.5 Benefícios da arquitetura de microsserviços . . . 68

5 CONCLUSÃO . . . 71

6 TRABALHOS FUTUROS . . . 75

REFERÊNCIAS . . . 77

APÊNDICES 81

APÊNDICE A – CÓDIGO-FONTE . . . 83

APÊNDICE B – TELAS DO INVESTMINDS . . . 85

APÊNDICE C – BANCO DE DADOS . . . 93

1 Introdução

1.1 Contexto

De acordo com o Serviço de Apoio às Micro e Pequenas Empresas, Sebrae (2023), um clube de assinatura é um modelo de negócio em que o cliente realiza um pagamento recorrente para receber produtos ou serviços em intervalos regulares, como semanal, quinzenal, mensal ou trimestral. Serviços de *streaming* de vídeo e música são exemplos desse formato. Além disso, existem clubes de assinatura para diferentes produtos, como cervejas, cafés, frutas, verduras, livros, roupas e itens infantis. Serviços também podem ser oferecidos nesse modelo, como aulas *on-line* ou diversos cursos em uma única plataforma.

Segundo Ofoeda J. Boateng e Effah (2019), uma API, é um conjunto de regras e protocolos que permite a comunicação entre diferentes *softwares*. Em essência, uma API define como um sistema ou aplicativo pode interagir com outro, facilitando a troca de dados e funcionalidades de forma padronizada e segura. As APIs são essenciais para a integração de sistemas, pois permitem que diferentes plataformas, sejam elas *web*, móveis ou de *desktop*, possam trabalhar juntas de maneira eficiente, permitindo a criação de soluções mais robustas e interconectadas.

Um microsserviço é um processo coeso e independente que interage por meio de mensagens. Do ponto de vista técnico, microsserviços devem ser componentes independentes, concebidos para serem implantados de forma isolada e equipados com ferramentas de banco de dados dedicados. Como todos os componentes de uma arquitetura de microsserviços são microsserviços, seu comportamento distintivo deriva da composição e coordenação de seus componentes por meio de mensagens. Uma arquitetura de microsserviços é uma aplicação distribuída em que todos os seus módulos são microsserviços.

A manutenibilidade e a evolução de um *software* dependem da aplicação de boas práticas durante o desenvolvimento. A cobertura de testes, a análise estática da qualidade do código, a redação de *commits* significativos, a criação de documentação, o uso de arquitetura de microsserviços e o consumo de APIs externas garantem uma melhor compreensão do sistema desenvolvido. Essas práticas também facilitam a expansão do projeto e a evolução do software, assegurando a qualidade das entregas.

O trabalho de conclusão de curso desenvolvido explorará como os princípios e estruturas das melhores práticas de desenvolvimento de software podem ser integrados em uma plataforma de clube de assinatura por meio de microsserviços e consumo de APIs aumentando a qualidade e facilitando a manutenabilidade. Será mostrado como empresas em diferentes estágios de maturidade podem adotar essas tecnologias, desde *startups* até

grandes corporações, e quais são os passos necessários para uma implementação bem-sucedida.

1.2 Questão de Pesquisa

Para produção do TCC, é importante compreender como funciona um clube de assinaturas e como os microsserviços podem revolucionar no desenvolvimento de um software. Nesse sentido, a questão de pesquisa a ser discutida neste trabalho é:

Como a adoção de microsserviços e o consumo de APIs podem transcender as tendências atuais e se estabelecer como pilares fundamentais para a criação de soluções de *software* robustas e escaláveis em clubes de assinatura, independentemente do estágio de maturidade da empresa?

1.3 Justificativa

Uma pesquisa anual de desenvolvedores conduzida por Gino (2021) e publicada na Forbes revelou que não são apenas as empresas de tecnologia que estão aproveitando APIs para construir código, mas praticamente todas as empresas. As empresas de tecnologia lideram a pesquisa, com 96% dos desenvolvedores planejando usar APIs mais ou na mesma medida em 2021, em comparação com 2020. No entanto, a adoção em outros setores também é significativa, com 94% dos desenvolvedores em serviços financeiros, 89% em telecomunicações e 86% na área de saúde indicando planos para aumentar ou manter o uso de APIs.

De acordo com a Red Hat (2022), compartilhar APIs com parceiros selecionados ou com o mundo inteiro pode gerar efeitos positivos. Cada parceria aumenta o reconhecimento da marca, indo além dos esforços de marketing da empresa. Tornar a tecnologia acessível a todos, como por meio de uma API pública, incentiva os desenvolvedores a criar um ecossistema de aplicativos em torno da API. Quanto mais pessoas utilizarem a tecnologia, maior será a probabilidade de que façam negócios com eles.

Segundo a IBM (2024), APIs simplificam o design e desenvolvimento de novas aplicações e serviços, além de facilitar a integração e gestão de sistemas existentes, trazendo benefícios significativos para desenvolvedores e organizações. Elas melhoram a colaboração ao permitir a integração de plataformas e aplicativos, automatizando fluxos de trabalho e evitando silos de informação que prejudicam a produtividade. APIs também aceleram a inovação, oferecendo flexibilidade para conectar novos parceiros de negócios e explorar novos mercados. Além disso, APIs podem ser monetizadas e reforçam a segurança dos sistemas e a privacidade dos usuários, adicionando camadas de proteção durante a comunicação entre aplicações e infraestruturas.

De acordo com Dragoni et al. (2017), os microsserviços representam uma tendência recente na arquitetura de software, com foco no design e desenvolvimento de sistemas altamente escaláveis e de fácil manutenção. Essa abordagem gerencia a crescente complexidade ao decompor grandes sistemas em um conjunto de serviços independentes. Ao garantir que os serviços sejam totalmente independentes no desenvolvimento e na implantação, os microsserviços promovem baixo acoplamento e alta coesão, elevando o nível de modularidade. Isso resulta em diversos benefícios relacionados à manutenção e à escalabilidade.

Em concordância, a Amazon Web Services (2024) diz que os microsserviços oferecem diversos benefícios, como agilidade, permitindo que equipes independentes e pequenas acelerem ciclos de desenvolvimento. A escalabilidade flexível permite que cada serviço seja escalado individualmente, garantindo alta disponibilidade e otimização de recursos. Além disso, facilitam a implantação contínua e a experimentação, oferecendo liberdade tecnológica para escolher as melhores ferramentas para cada tarefa. A arquitetura também promove o reúso de código e aumenta a resiliência do sistema, evitando que falhas isoladas afetem o funcionamento completo do aplicativo.

Dessa forma, o estudo aprofundado sobre o impacto da utilização da arquitetura de microsserviços e do consumo de APIs em uma plataforma de clube de investimentos para um TCC não apenas é atual e relevante, mas também oferece uma visão estratégica sobre como a integração de sistemas pode impulsionar a inovação, a segurança e a expansão dos negócios no cenário tecnológico atual.

1.4 Objetivos

1.4.1 Objetivo Geral

Desenvolver uma plataforma *web* que centralize soluções independentes de *software* de investimento, utilizando uma arquitetura baseada em microsserviços e integração com APIs externas, visando demonstrar a eficácia da arquitetura e seus benefícios em clubes de assinatura.

1.4.2 Objetivos Específicos

Para alcançar o objetivo geral, definiram-se os seguintes objetivos específicos:

- Projetar e implementar uma arquitetura de APIs robusta: Desenvolver uma arquitetura de APIs que permita a integração com diferentes provedores, garantindo escalabilidade e flexibilidade.

- Projetar e implementar uma arquitetura de Microsserviços eficiente: Desenvolver uma arquitetura de Microsserviços que permita o funcionamento entre os serviços, escalabilidade e correção independentes.
- Utilizar as melhores práticas de desenvolvimento de forma a garantir a qualidade do código e manutenabilidade;
- Estudar o potencial de escalabilidade: Analisar como a plataforma pode ser expandida para incluir novas soluções de investimentos e clientes.

1.5 Organização do Trabalho

- Capítulo 1 - Introdução: Esta seção inicial apresenta o contexto do projeto, incluindo a questão de pesquisa, justificativa, objetivos e as metodologias utilizadas.
- Capítulo 2 - Referencial Teórico: Este capítulo explora os conceitos essenciais para a compreensão do contexto da pesquisa, abordando o tema principal e os recursos tecnológicos que podem embasar e esclarecer melhor a proposta.
- Capítulo 3 - Proposta do trabalho: Este capítulo é dedicado à explicação das metodologias adotadas na execução do trabalho, detalhando as atividades planejadas, as tecnologias empregadas, os requisitos estabelecidos e a arquitetura do *software*, juntamente com seus recursos.
- Capítulo 4 - Resultados: Esta seção demonstra os resultados obtidos após a realização prática do trabalho desenvolvido a partir do desenvolvimento da plataforma, gerenciamento do projeto e objetivos alcançados.
- Capítulo 5 - Considerações Finais: Este capítulo conclui a segunda fase do Trabalho de Conclusão de Curso (TCC 2), representando os pontos levantados ao longo do trabalho.

2 Referencial Teórico

O referencial teórico é fundamental para fornecer uma base sólida à pesquisa, integrando-a ao conhecimento já existente e validando os resultados alcançados. Neste capítulo, serão apresentados e discutidos os conceitos de APIs, microsserviços e clubes de assinatura, de forma a proporcionar uma compreensão clara e consistente desses temas ao longo do texto.

2.1 Fundamentos de API e Microsserviços

A arquitetura de microsserviços e o amplo uso de *Application Programming Interface* (API) revolucionaram o desenvolvimento de *software* ao permitir que sistemas sejam construídos de forma modular, escalável e resiliente. Diferente de abordagens monolíticas tradicionais, em que toda a funcionalidade de uma aplicação está encapsulada em uma única unidade, os microsserviços dividem uma aplicação em pequenos serviços independentes, cada um com sua própria responsabilidade. De acordo com Newman (2015b), esses serviços comunicam-se entre si por meio de APIs, frequentemente utilizando protocolos como HTTP, REST ou gRPC, promovendo um alto grau de desacoplamento e flexibilidade. Para Richardson, Amundsen e Ruby (2016), as APIs também desempenham um papel central na integração de sistemas, atuando como contratos que permitem a comunicação entre diferentes componentes de *software*, independentemente da linguagem de programação ou da plataforma em que foram desenvolvidos. Segundo Fowler (2014), esse padrão arquitetural tem sido adotado por muitas grandes corporações, como Netflix e Amazon, devido à sua capacidade de aumentar a escalabilidade, facilitar a manutenção e suportar a inovação contínua.

2.1.1 API

De acordo com a Amazon (2024), APIs são mecanismos que permitem que dois componentes de *software* se comuniquem entre si, utilizando um conjunto de definições e protocolos padronizados. A palavra *Application* refere-se a qualquer *software* com uma função específica, enquanto *Interface* pode ser entendida como um contrato de serviço entre duas aplicações, definindo como elas trocam dados por meio de requisições e respostas. Esse contrato, especificado na documentação da API, orienta os desenvolvedores sobre como estruturar adequadamente essas requisições e respostas. Na maioria dos casos, a arquitetura de API é descrita em termos de um cliente, que envia a requisição, e um servidor, que processa a requisição e retorna uma resposta.

Existem diversos tipos de APIs, cada um com características específicas para atender a diferentes necessidades de comunicação entre sistemas. As quatro formas mais comuns de APIs, conforme categorizado pela Amazon (2024), são: SOAP, RPC, WebSocket e REST. Cada tipo de API oferece vantagens específicas de acordo com o cenário de uso e o nível de complexidade das interações entre cliente e servidor. A Tabela 1 resume as diferenças fundamentais entre essas APIs, detalhando os formatos de mensagens e os casos de uso mais frequentes.

Tipo de API	**Modelo de Comunicação**	**Formato de Mensagens**	**Casos de Uso Típicos**
REST	Comunicação baseada em HTTP, com operações CRUD (Create, Read, Update, Delete)	JSON, XML, HTML	Serviços web que seguem princípios de escalabilidade e simplicidade, como APIs públicas e sistemas distribuídos
SOAP	Protocolo baseado em XML, com mensagens padronizadas via HTTP/HTTPS	XML	Aplicações que requerem segurança e transações complexas, como serviços financeiros e de pagamento
RPC	Chamada de Procedimento Remoto (Remote Procedure Call)	JSON-RPC, XML-RPC	Aplicações de chamada direta a métodos remotos, em que a simplicidade e performance são cruciais
WebSocket	Comunicação bidirecional em tempo real	Texto ou Binário	Aplicações que necessitam de comunicação contínua e em tempo real, como chat, jogos *on-line* e *streaming*

Tabela 1 – Diferenças entre APIs SOAP, RPC, WebSocket e REST

As APIs SOAP utilizam o protocolo *Simple Object Access Protocol*, no qual o cliente e servidor trocam mensagens formatadas em XML. W3C (2007) diz que essa abordagem foi amplamente utilizada no passado devido à sua robustez e foco em segurança, especialmente em ambientes empresariais complexos. No entanto, para Newman (2015b), o SOAP tem perdido popularidade devido à sua relativa complexidade e à necessidade de maiores recursos computacionais, sendo substituído em muitos casos pelo REST, que oferece maior flexibilidade.

As APIs RPC, por sua vez, implementam o conceito de chamadas de procedimentos remotos. Nessa abordagem, Birrell e Nelson (1984) explica que o cliente executa uma função ou procedimento diretamente no servidor, recebendo o resultado como se fosse uma função local. Para Newman (2015b), embora o RPC tenha a vantagem de ser eficiente em

cenários que exigem baixa latência, ele enfrenta desafios em termos de acoplamento entre cliente e servidor, o que pode dificultar a manutenção e a escalabilidade do sistema.

Segundo Fette e Melnikov (2011), a API WebSocket é uma tecnologia mais moderna, projetada para permitir comunicação bidirecional em tempo real entre cliente e servidor . Essa abordagem é especialmente útil em aplicações que requerem interações contínuas, como plataformas de chat, jogos *on-line* e sistemas de *streaming*. Ao contrário de REST, que segue o padrão de requisição-resposta, o WebSocket mantém uma conexão aberta, permitindo que o servidor envie dados ao cliente sem a necessidade de novas requisições, tornando-o mais eficiente em cenários de alta interatividade.

Dessa forma, Fielding (2000) diz que as APIs REST são muito utilizadas no desenvolvimento de aplicações *web* modernas, sendo amplamente apreciadas por sua simplicidade e compatibilidade com o protocolo HTTP. REST permite que clientes façam requisições ao servidor utilizando métodos bem definidos (como *GET, POST, PUT e DELETE*) e o servidor responde com dados formatados, normalmente em JSON ou XML. Para Richardson, Amundsen e Ruby (2016), o REST segue o princípio de ser *stateless*, ou seja, o servidor não armazena informações sobre requisições anteriores, o que o torna mais escalável. A Figura 1 ilustra o funcionamento básico de uma API REST.

Figura 1 – API Rest.

Segundo Ofoeda J. Boateng e Effah (2019), apesar de existirem diversas razões pelas quais as empresas adotam uma estratégia baseada em APIs, três áreas principais se destacam como motivadoras desse uso: a melhoria na experiência do cliente, a ampliação dos ecossistemas e a busca por eficiência operacional.

Uma das principais vantagens do uso de APIs é a capacidade de oferecer uma experiência de cliente aprimorada. As APIs permitem que as empresas integrem rapidamente novos serviços e funcionalidades em suas plataformas, personalizando ofertas de acordo com as necessidades específicas dos usuários. Isso é especialmente relevante em um mercado onde a personalização e a conveniência são fatores críticos para a satisfação do cliente. Ao permitir a integração com plataformas de terceiros, como redes sociais, sistemas de pagamento ou serviços de mensagens, as APIs proporcionam uma experiência mais fluida e envolvente.

Além disso, as APIs permitem que empresas atualizem seus serviços de forma contínua, sem interrupções significativas para o usuário final. Isso não só garante que os

clientes tenham acesso às funcionalidades mais recentes, mas também melhora a confiabilidade do serviço. A experiência de cliente se torna mais ágil e conectada, resultando em maior satisfação e retenção.

A expansão de ecossistemas é outra área central no uso de APIs, pois elas facilitam a integração de diversas plataformas e serviços, permitindo que empresas criem redes colaborativas robustas. Ao adotar APIs, uma organização pode conectar seu sistema a fornecedores, parceiros e outros serviços complementares, criando um ecossistema tecnológico integrado. Esse tipo de interconexão permite que as empresas compartilhem dados de forma eficiente, melhorem a interoperabilidade e desenvolvam soluções inovadoras com base em colaborações.

Além disso, a expansão de ecossistemas através de APIs permite às empresas entrar em novos mercados e oferecer serviços a um público mais amplo. As APIs funcionam como portas de entrada para novas oportunidades de negócios e parcerias, permitindo que terceiros criem novas aplicações e produtos a partir das funcionalidades já existentes. Com isso, as empresas ampliam seu alcance, criando um ciclo de inovação onde novas funcionalidades podem ser desenvolvidas e integradas de maneira rápida, escalável e colaborativa.

A eficiência operacional é uma das áreas mais impactadas pela adoção de APIs, uma vez que elas permitem automatizar processos, reduzir a duplicação de esforços e otimizar o uso de recursos. Com APIs, empresas podem conectar sistemas internos, eliminando tarefas manuais e criando fluxos de trabalho mais dinâmicos e eficientes.

Além de simplificar processos, as APIs também ajudam a reduzir custos operacionais. A automação de tarefas repetitivas e a integração de sistemas eliminam a necessidade de desenvolver funcionalidades do zero, economizando tempo e recursos. Em vez de criar novos sistemas, empresas podem utilizar APIs para integrar soluções já existentes, otimizando o uso de tecnologias. Isso se traduz em uma operação mais enxuta, ágil e pronta para escalar de acordo com a demanda, aumentando a competitividade da empresa no mercado.

Ainda de acordo com Ofoeda J. Boateng e Effah (2019), programadores também são, em alguns casos, usuários de APIs e, por isso, precisam de bibliotecas que sejam fáceis de aprender. O valor de uma API é amplamente determinado por sua potência e, mais significativamente, pela sua usabilidade. Dessa forma, uma boa API deve ser de fácil aprendizado e utilização, contribuindo diretamente para a produtividade dos desenvolvedores. Para serem utilizáveis, as APIs precisam ser eficientes, satisfatórias, apresentar o mínimo de erros, serem fáceis de aprender e memorizáveis, permitindo que os desenvolvedores lembrem-se facilmente das chamadas da API. Além disso, é importante que as APIs sejam adequadas para o uso em tarefas específicas.

Além da facilidade de aprendizado e memorização, uma API eficaz também deve proporcionar clareza em sua documentação. Documentações completas e bem estruturadas permitem que desenvolvedores possam rapidamente encontrar as informações necessárias, tornando a interação com a API mais eficiente. Isso reduz a curva de aprendizado e diminui o número de erros ao implementar a API, uma vez que os desenvolvedores têm acesso claro às funcionalidades disponíveis e aos parâmetros esperados. Além disso, uma boa documentação deve incluir exemplos de uso, ajudando os programadores a visualizarem cenários reais de implementação.

Outro ponto relevante é a consistência no design da API. Quando uma API segue padrões consistentes em suas chamadas e estrutura de dados, os desenvolvedores têm mais facilidade em adaptar seu uso a diferentes contextos, aumentando a reutilização do código. APIs bem projetadas permitem que os programadores se concentrem mais na lógica de negócios de suas aplicações, sem se preocupar com a complexidade da integração. Essa consistência contribui para um ciclo de desenvolvimento mais ágil, já que diminui o tempo necessário para debug e implementação de funcionalidades novas, promovendo a produtividade e a qualidade do software final.

2.1.2 Consumo e criação de APIs

De acordo com Fette e Melnikov (2011), cada tipo de API possui suas vantagens e desvantagens que variam de acordo com o cenário de uso. WebSocket destaca-se em aplicações que necessitam de comunicação contínua e em tempo real, como plataformas de bate-papo e jogos *on-line*. Ao contrário de REST e SOAP, que seguem o padrão de requisição-resposta, WebSocket mantém uma conexão aberta e persistente entre cliente e servidor, permitindo a troca de mensagens bidirecionais e em tempo real. Um bom exemplo são em sistemas de corretoras de ações, nos quais os dados sobre cotações de mercado precisam ser atualizados em tempo real para os usuários. Esse modelo reduz a latência e o *overhead* de abertura de novas conexões, o que o torna ideal para casos de uso de alta interatividade. No entanto, WebSocket pode ser mais complexo de implementar e gerenciar, pois exige cuidados adicionais com a segurança e o gerenciamento de conexões abertas por longos períodos, especialmente em termos de autenticação e proteção contra ataques. Além disso, para Fowler (2018), o uso contínuo de WebSockets pode aumentar significativamente o consumo de recursos do servidor, exigindo um planejamento cuidadoso de escalabilidade da infraestrutura.

Para W3C (2007), o protocolo SOAP oferece uma abordagem mais robusta e segura, sendo ideal para ambientes empresariais que requerem altos níveis de segurança e integridade de dados, como bancos e sistemas de saúde. De acordo com OASIS (2006), ele inclui suporte nativo para transações distribuídas, controle de erros e WS-Security, um padrão que garante a proteção de mensagens SOAP através de autenticação, integridade

e confidencialidade. Em serviços financeiros, por exemplo, SOAP é amplamente utilizado para garantir a segurança na troca de informações sensíveis, como detalhes de transações bancárias. O uso de XML como formato de mensagem pode gerar um *overhead* maior e torná-lo lento em operações simples quando comparado a REST. Em concordância, Newman (2015b) diz que o SOAP é compatível com vários protocolos de transporte, como HTTPS e SMTP, o que o torna uma escolha flexível para redes corporativas complexas, mas com um custo de implementação mais elevado.

Birrell e Nelson (1984) dizem que o RPC é bastante utilizado em cenários que exigem baixa latência e execução direta de funções remotas, como em sistemas distribuídos em tempo real. Um exemplo de uso de RPC é no controle de sistemas de automação industrial, no qual dispositivos remotos, como sensores e controladores, devem se comunicar com um servidor central para executar tarefas específicas. Com RPC, é possível que um cliente invoque diretamente funções no servidor, como se essas funções estivessem sendo executadas localmente. Para Newman (2015b), essa abordagem pode gerar um alto acoplamento entre cliente e servidor, uma vez que ambos precisam estar sincronizados em relação às definições dos métodos e às estruturas de dados, o que pode dificultar a manutenção e evolução do sistema, especialmente em ambientes em que a flexibilidade é necessária, como quando os serviços precisam ser modificados ou escalados rapidamente.

De acordo com Fielding (2000), o REST é amplamente utilizada devido à sua simplicidade e flexibilidade, principalmente em aplicações web. Seguindo os princípios da arquitetura da *web*, REST utiliza verbos HTTPS para manipulação de recursos, sendo adequado para integração entre sistemas e plataformas. Um exemplo do uso de REST é em plataformas de redes sociais, como o Facebook, em que a API permite que desenvolvedores interajam com dados dos usuários e funcionalidades da plataforma. Por outro lado, para Richardson, Amundsen e Ruby (2016), o REST pode não ser a melhor escolha para sistemas que exigem comunicação em tempo real ou transações complexas, devido à sua natureza sem estado (*stateless*). Isso pode aumentar a complexidade em aplicações que exigem persistência de sessões, como sistemas de e-commerce que mantêm o estado do carrinho de compras do usuário.

Segundo Massé (2012), cada método HTTP tem semânticas específicas e bem definidas no contexto do modelo de recursos de uma API REST. O propósito do GET é recuperar uma representação do estado de um recurso. HEAD é usado para recuperar os metadados associados ao estado do recurso. PUT deve ser utilizado para adicionar um novo recurso a um repositório ou atualizar um recurso. DELETE remove um recurso de seu pai. POST deve ser usado para criar um novo recurso dentro de uma coleção e executar controladores.

Ainda de acordo com Massé (2012), os métodos HTTP possuem regras específicas que devem ser seguidas ao projetar e implementar uma API REST. Seguir essas regras

garante que a API funcione de acordo com os princípios do HTTP, preservando a integridade e a transparência da comunicação entre clientes e servidores. Essas normas são fundamentais para manter o comportamento correto das operações e evitar ambiguidades que possam comprometer a funcionalidade do sistema.

A primeira regra a ser considerada é que os métodos GET e POST não devem ser usados para realizar *tunneling* de outros métodos de requisição. *Tunneling* refere-se ao uso indevido do protocolo HTTP, mascarando ou deturpando a intenção de uma mensagem, o que compromete a transparência da comunicação. Uma API REST deve evitar esse tipo de abuso e fazer uso adequado dos métodos de requisição HTTP, conforme definido por suas respectivas funções.

No caso do método GET, ele deve ser utilizado exclusivamente para recuperar uma representação de um recurso. Ao enviar uma requisição GET, o cliente busca o estado de um recurso em uma forma representacional, podendo incluir cabeçalhos, mas sem conter um corpo na requisição. Além disso, o método GET é desenhado para ser seguro e sem efeitos colaterais, o que permite que as requisições sejam repetidas sem alterações no estado do servidor. Isso é essencial para o funcionamento correto dos sistemas de cache na Web, que dependem da repetição de GET sem causar mudanças nos dados.

O método HEAD segue uma abordagem semelhante ao GET, porém é usado para recuperar apenas os cabeçalhos da resposta, sem incluir o corpo. Dessa forma, um cliente pode verificar a existência de um recurso ou acessar seus metadados sem precisar transferir o conteúdo completo do recurso. Esse método é particularmente útil quando se deseja reduzir a carga de dados trafegados em consultas específicas.

O método PUT deve ser utilizado tanto para inserir quanto para atualizar um recurso já existente. Ele permite que o cliente envie a representação completa do recurso que deseja armazenar, especificando o URI. Quando usado, o PUT substitui o recurso no servidor com a nova versão fornecida ou cria um novo, caso o recurso ainda não exista. Essa abordagem é adequada para situações onde é necessário garantir que o recurso completo seja enviado ou atualizado em uma única operação.

Além disso, o método PUT também deve ser empregado para atualizar recursos mutáveis. Isso significa que, quando um recurso existente precisa ser alterado, a requisição PUT deve conter o estado desejado para a atualização. Essa atualização pode incluir um corpo na requisição que reflita as modificações necessárias.

Já o método POST é utilizado quando o objetivo é criar um novo recurso dentro de uma coleção gerenciada pelo servidor. Nesse caso, o cliente envia a representação sugerida do novo recurso no corpo da requisição, e o servidor é responsável por armazená-lo e atribuir um identificador único. Além disso, o POST também deve ser empregado para invocar controladores, que executam funções específicas dentro da API. Diferente dos

outros métodos, o POST é semânticamente mais aberto e pode ser usado para diversas finalidades, como realizar operações que não se encaixam em métodos como GET, PUT ou DELETE.

O método DELETE, por sua vez, deve ser utilizado para remover um recurso de seu pai, seja este uma coleção ou um repositório. Uma vez que uma requisição DELETE é processada com sucesso, o recurso não estará mais acessível, e futuras tentativas de acesso ao mesmo, através de GET ou HEAD, devem resultar em um código de status 404 ("Não encontrado").

Por fim, o método OPTIONS é utilizado para recuperar metadados de um recurso, como as interações permitidas e os métodos HTTP suportados. Um exemplo de resposta a uma requisição OPTIONS incluiria um cabeçalho *Allow* com os métodos permitidos, como GET, PUT e DELETE. Esse método fornece informações sobre as possíveis ações que podem ser executadas em um recurso, sem realizar qualquer modificação em seu estado.

2.1.3 Microsserviços

Segundo Newman (2015b), microsserviços são uma abordagem arquitetural que divide uma aplicação em serviços pequenos e independentes, cada um responsável por uma funcionalidade específica do sistema. Já em concordância com Fowler (2014), ao contrário da arquitetura monolítica, na qual todas as funcionalidades são implementadas em uma única aplicação, os microsserviços promovem o desacoplamento e a modularidade, fazendo com que cada serviço seja desenvolvido, implantado e escalado de forma isolada. Esse modelo oferece maior flexibilidade e escalabilidade, além de facilitar a manutenção e a atualização de partes específicas da aplicação sem afetar o sistema como um todo.

Em concordância, Alshuqayran, Ali e Evans (2016) diz que os microsserviços são um estilo de arquitetura que divide o sistema em pequenos e leves serviços, cada um projetado para desempenhar uma função de negócio coesa. É uma evolução da arquitetura orientada a serviços (SOA) e é caracterizado pela independência dos seus componentes. Interagem por meio de mensagens e são frequentemente distribuídos em uma aplicação onde todos os módulos são microsserviços.

De acordo com Richardson, Amundsen e Ruby (2016), cada microsserviço possui sua própria lógica de negócio e pode ser desenvolvido utilizando diferentes linguagens de programação ou tecnologias, desde que siga padrões de comunicação bem definidos, geralmente através de APIs REST, gRPC ou protocolos de mensagem como AMQP ou Kafka. Essa abordagem tem sido amplamente adotada por grandes empresas, como Netflix e Amazon, que precisam de sistemas escaláveis e resilientes para suportar uma grande base de usuários e operações em tempo real.

Alshuqayran, Ali e Evans (2016) afirma que que modelar microsserviços com as notações padrão do UML é comparável a criar outra notação de modelagem abrangente, além de ser comparável ao uso de desenhos informais com caixas livres e linhas acompanhadas de uma narrativa. No entanto, como um sistema típico baseado em microsserviços consiste em vários contêineres e cada contêiner, por sua vez, contém um ou mais componentes, ou seja, microsserviços, que, por sua vez, são implementados por uma ou mais classes, a notação padrão do UML pode fornecer um conjunto comum de abstrações e notações para descrever a arquitetura de microsserviços. Portanto, o uso de vários diagramas UML, como contexto, contêiner, componente, classe, caso de uso, sequência, cada um mostrando uma parte diferente de toda a arquitetura, será eficaz para comunicar os designs de software de maneira eficaz e eficiente.

O gerenciamento de microsserviços em um ambiente de produção exige uma série de ferramentas e práticas que garantem a escalabilidade, o monitoramento, a resiliência e a consistência dos serviços. Entre as ferramentas mais utilizadas, destacam-se o Docker e o Kubernetes.

O Docker (2013) desempenha um papel importante no gerenciamento de microsserviços. Ele permite que cada serviço seja encapsulado em contêineres isolados com tudo o que é necessário para sua execução: código, bibliotecas e dependências. Cada serviço pode ser iniciado, interrompido e escalado independentemente, conforme a demanda, além de permitir a replicação em sistemas distribuídos. Ao utilizar contêineres, os microsserviços podem ser executados de maneira consistente em diferentes ambientes, seja no desenvolvimento, nos testes ou na produção.

Em um ambiente de microsserviços, é comum que diversos contêineres sejam executados simultaneamente, e o Kubernetes (2014) automatiza o processo de implantação, balanceamento de carga, escalabilidade e recuperação de falhas desses contêineres. Ele permite o gerenciamento de forma eficiente, fornecendo suporte para a descoberta de serviços e atualizações sem tempo de inatividade. Quando um microsserviço falha ou precisa ser escalado, o Kubernetes automaticamente inicia novos contêineres ou redireciona o tráfego, garantindo a disponibilidade contínua da aplicação. Além disso, o Kubernetes permite a configuração de escalabilidade horizontal, onde novos contêineres são criados automaticamente quando a carga de trabalho aumenta.

Além de Docker e Kubernetes, outras ferramentas complementares são úteis para o gerenciamento de microsserviços:

- Istio (2018): Uma *service mesh* que controla a comunicação entre microsserviços, gerencia o tráfego de rede e implementa políticas de segurança, como a mTLS, além de fornecer métricas e *logs* detalhados para observabilidade.
- Prometheus e Grafana: Ferramentas de monitoramento e visualização de métricas.

O Prometheus coleta dados de desempenho, enquanto o Grafana exibe essas informações em *dashboards* interativos, auxiliando na detecção e resolução de problemas em sistemas distribuídos.

- **Elastic Stack (ELK)**: Composto por Elasticsearch, Logstash e Kibana, o ELK oferece uma solução robusta para o gerenciamento e análise de *logs*, agregando dados de diferentes serviços e facilitando a busca por falhas ou comportamentos anômalos.
- **Consul** e **Etcd**: Ferramentas de gerenciamento de configuração e descoberta de serviços, permitindo que os microsserviços armazenem suas configurações e se descubram dinamicamente, sem a necessidade de configuração manual.

De acordo com Resilience4j (2018), é fundamental lidar com falhas de maneira eficaz, já que cada serviço é uma peça crítica do sistema como um todo. Padrões como *Circuit Breaker*, implementado com ferramentas como Hystrix ou Resilience4j, permitem interromper automaticamente as chamadas a um serviço que está falhando, evitando que falhas em cascata comprometam todo o sistema. Esse tipo de abordagem ajuda a garantir a resiliência em sistemas distribuídos, oferecendo uma estratégia robusta para gerenciar falhas parciais e manter a operação de serviços essenciais.

2.1.4 Padrões e Arquiteturas de microsserviços

A arquitetura de microsserviços envolve a decomposição de aplicações em serviços independentes, mas isso também traz desafios relacionados à comunicação, gerenciamento de dados e resiliência. Para enfrentar esses desafios, surgiram diversos padrões de arquitetura que auxiliam no design e gerenciamento eficaz de microsserviços.

Seguindo os ideias de Newman (2015b), o princípio fundamental dos microsserviços é que cada serviço seja independente, ou seja, cada serviço pode ser desenvolvido, testado, implantado e escalado de maneira autônoma. Isso permite que equipes de desenvolvimento trabalhem em paralelo e adotem tecnologias diferentes para cada microsserviço. Um exemplo é quando uma empresa decide utilizar linguagens de programação diferentes para cada serviço, dependendo da natureza da tarefa (por exemplo, Python para processamento de dados e Java para serviços de autenticação).

Para Fowler (2014), a independência dos serviços também traz benefícios para a escalabilidade: serviços críticos que demandam mais recursos podem ser escalados individualmente sem afetar os outros serviços. No entanto, esse padrão também apresenta desafios, como a necessidade de gerenciar a comunicação e a consistência entre os serviços, especialmente quando há dependências de dados.

Richardson (2016) diz que um API *Gateway* é um padrão comum em arquiteturas de microsserviços, atuando como um ponto de entrada único para todas as requisições

externas. Ele recebe as solicitações do cliente e as encaminha para os serviços apropriados. O API *Gateway* também pode executar tarefas como autenticação, roteamento e controle de versões das APIs, simplificando a lógica de comunicação entre o cliente e os microsserviços.

Para Fowler (2014), além de facilitar a comunicação, o API *Gateway* melhora a segurança, pois centraliza o controle de acesso e pode aplicar políticas de segurança de forma consistente. No entanto, o uso do API *Gateway* também introduz um ponto único de falha e pode adicionar latência à comunicação, especialmente em sistemas distribuídos.

De acordo com Richardson (2016), o padrão Saga é utilizado para gerenciar transações distribuídas, uma vez que cada microsserviço geralmente possui seu próprio banco de dados e transações tradicionais (ACID) não são aplicáveis em uma arquitetura distribuída. Em uma Saga, uma série de operações distribuídas são executadas em sequência e, se uma dessas operações falha, um outro conjunto é acionado para desfazer as operações anteriores.

É necessário garantir a consistência de dados entre diferentes microsserviços sem bloquear o sistema. Um exemplo comum é em sistemas de reserva de passagens aéreas, onde várias operações (como reservar assentos e processar pagamentos) precisam ser realizadas de forma coordenada. Caso uma das operações falhe, as outras são revertidas para manter a consistência do sistema.

2.1.5 Arquitetura BFF

De acordo com Lukasz Plotnicki (2015), a arquitetura *Backend for Frontend* é uma variação da arquitetura de microsserviços. Seu conceito surgiu como uma solução arquitetural para lidar com a crescente complexidade no desenvolvimento de aplicações que consomem múltiplos microsserviços. O termo foi popularizado pela SoundCloud em 2015, em resposta aos desafios enfrentados na construção de APIs genéricas que atendessem igualmente a diferentes tipos de clientes, como aplicações *web*, *mobile* e *desktop*. Em uma arquitetura baseada em BFF, cada *frontend* interage exclusivamente com o seu respectivo BFF, que então coordena as interações com os microsserviços. Segundo Newman (2015a), esse modelo oferece diversas vantagens:

- **Customização**: Cada BFF pode ser adaptado para as necessidades específicas do seu *frontend*, agregando ou transformando dados antes de enviá-los ao cliente, o que reduz a complexidade no *frontend* e melhora o desempenho.
- **Isolamento de Mudanças**: Mudanças em um *frontend* específico não precisam afetar os demais. O BFF atua como uma camada de abstração, isolando o *frontend* das mudanças nos microsserviços ou em outros *frontends*.

- **Desempenho**: O BFF pode orquestrar chamadas a múltiplos microsserviços em paralelo e agregar os resultados antes de retorná-los ao *frontend*, o que pode reduzir a latência percebida pelo usuário.

- **Segurança**: Ao centralizar as interações do *frontend* com os microsserviços em um BFF, é possível aplicar políticas de segurança e controle de acesso de maneira mais granular e adaptada às necessidades de cada *frontend*.

No desenvolvimento de uma plataforma *web* que consome diversos microsserviços, o BFF se mostra particularmente útil para resolver problemas de complexidade e desempenho. Para Richardson (2018), o BFF pode unificar dados de diferentes microsserviços em um único *payload*, economizando chamadas HTTP no *frontend* e simplificando a lógica do cliente, além de realizar chamadas paralelas a vários microsserviços e processar os resultados antes de retorná-los ao *frontend*.

Então, a arquitetura BFF se apresenta como uma solução poderosa e flexível para o desenvolvimento de plataformas *web* que consomem múltiplos microsserviços. Ela permite o desenvolvimento de aplicações mais performáticas, escaláveis e de fácil manutenção. Ao reduzir a complexidade e melhorar a eficiência das interações entre o *frontend* e os microsserviços, o BFF torna-se um componente de destaque em arquiteturas modernas baseadas em microsserviços.

2.1.6 Desenvolvimento de microsserviços com Node.js e React

No desenvolvimento de microsserviços, adotar boas práticas é fundamental para garantir manutenibilidade, escalabilidade e a evolução contínua do software. A construção de microsserviços com Node.js e a interface com React permite a criação de sistemas escaláveis, resilientes e fáceis de manter. A adoção dos padrões de simplicidade e modularidade, conforme discutidos por Kondov (2023a), Kondov (2023b), resulta em um ciclo de desenvolvimento mais eficiente, pois as mudanças são localizadas e as dependências são minimizadas.

O Node.js é amplamente utilizado para o desenvolvimento de microsserviços devido à sua capacidade de lidar com I/O de forma assíncrona e não bloqueante, características que facilitam o desenvolvimento de aplicações escaláveis. Kondov (2023a) enfatiza a importância de manter o código modular e dividido em pequenas unidades de responsabilidade clara. Esse princípio é essencial para a arquitetura de microsserviços, no qual cada serviço deve ser independente e dedicado a uma tarefa específica. Manter uma arquitetura modular em Node.js facilita o desenvolvimento e a manutenção dos serviços, uma vez que as funcionalidades podem ser facilmente modificadas ou escaladas sem impactar outras partes do sistema. Essa modularidade também facilita a integração com ferramentas e

frameworks externos, promovendo a extensibilidade dos serviços. Um microsserviço modular pode ser evoluído e adaptado com novos recursos ou tecnologias sem necessidade de refatorar grandes partes da aplicação.

Um dos maiores benefícios do React é sua abordagem baseada em componentes, que permite a criação de UIs reutilizáveis e compostas de pequenas peças independentes. Em uma arquitetura de microsserviços, em que a independência de cada componente ou serviço é primordial, o React complementa bem esse estilo de arquitetura. De acordo com Kondov (2023b), componentes React podem ser usados de maneira similar a microsserviços: pequenos, modulares e responsáveis por uma única tarefa. Isso não só facilita a manutenibilidade, mas também permite que partes da interface evoluam independentemente das demais.

A combinação de Node.js e React também permite o compartilhamento de código e padrões entre o *frontend* e o *backend*, promovendo uma experiência de desenvolvimento mais unificada e eficiente. A capacidade de utilizar JavaScript em ambos os lados da aplicação simplifica o processo de desenvolvimento, uma vez que a equipe pode trabalhar com uma linguagem única em todo o sistema.

2.1.7 Bancos de dados relacionais e não-relacionais

Os bancos de dados relacionais são caracterizados por armazenar dados de forma estruturada em tabelas compostas por linhas e colunas, nas quais cada linha representa um registro e cada coluna, um atributo desse registro. Utilizando a linguagem SQL para manipulação e consulta de dados, o modelo relacional requer que as tabelas sigam um esquema rígido previamente definido a fim de garantir a integridade e consistência das informações. Bancos como MySQL, PostgreSQL e Oracle são exemplos notáveis dessa categoria e são amplamente utilizados em sistemas que lidam com transações complexas e onde a estrutura dos dados não sofre grandes alterações Kroenke e Auer (2013).

Com o crescimento das aplicações *web* modernas, como redes sociais e outros sistemas que precisam lidar com diferentes tipos de dados, os bancos de dados relacionais começaram a enfrentar desafios em termos de flexibilidade e escalabilidade. Além disso, os dados gerados por essas aplicações frequentemente não seguem a forma rigidamente estruturada das tabelas relacionais, o que leva à necessidade de lidar com três tipos principais de dados: estruturados, semi-estruturados e não estruturados.

- **Dados estruturados**: São os dados organizados de forma rígida em um esquema predefinido, geralmente em tabelas, como é o caso nos bancos de dados relacionais. Cada dado é armazenado em um formato tabular com campos bem definidos e tipos de dados consistentes, facilitando a consulta por meio de SQL. Exemplos incluem transações financeiras e registros de clientes.

- **Dados não estruturados**: Refere-se a qualquer informação que não segue um modelo de dados predefinido ou um esquema formal. Esse tipo de dado não é facilmente categorizado em tabelas ou bases de dados relacionais. Exemplos incluem arquivos de texto, imagens e vídeos.

- **Dados semi-estruturados**: São dados que não possuem uma estrutura rígida, mas ainda contêm algumas tags ou marcadores que os organizam de maneira parcial. Exemplos típicos incluem documentos XML, JSON e logs de sistemas.

Os bancos de dados não-relacionais, também conhecidos como NoSQL, surgem como alternativa aos bancos de dados relacionais tradicionais, atendendo às demandas de aplicações que requerem maior flexibilidade e escalabilidade. Segundo Cattell (2011), os bancos de dados NoSQL diferenciam-se dos bancos de dados relacionais, principalmente, por não dependerem de esquemas rígidos, o que permite a modelagem de dados de maneira mais flexível. Eles também são projetados para escalar horizontalmente, distribuindo dados por múltiplos servidores de maneira eficiente, o que é essencial para aplicações que lidam com grandes volumes de dados e acessos concorrentes.

A principal motivação para o uso de bancos de dados NoSQL está relacionada à sua capacidade de lidar com grandes quantidades de dados não estruturados ou semi-estruturados, como ocorre em aplicações como redes sociais. Segundo Pokorny (2011), enquanto os bancos de dados relacionais são baseados no modelo de dados tabular, os sistemas NoSQL podem ser baseados em diferentes modelos, como documentos, colunas, grafos ou chaves-valor, cada um adequado para diferentes tipos de aplicações e necessidades.

- **Bancos orientados a documentos**: Armazenam dados em documentos no formato JSON ou BSON, o que permite flexibilidade na estrutura dos dados. MongoDB é o exemplo mais comum dessa categoria.

- **Bancos orientados a colunas**: Organizam os dados em colunas em vez de linhas, sendo úteis para consultas analíticas em grandes volumes de dados. Um exemplo comum é o Apache Cassandra.

- **Bancos de chave-valor**: Armazenam pares de chave e valor, onde cada chave é única e aponta para um valor, que pode ser de qualquer tipo de dado. Redis e DynamoDB são exemplos desse tipo.

- **Bancos orientados a grafos**: São usados para representar e consultar dados com relações complexas, como em redes sociais ou sistemas de recomendação. Neo4j é um exemplo popular.

Esses diferentes modelos de bancos de dados NoSQL fornecem soluções versáteis para lidar com a necessidade de escalabilidade e com a diversidade de formatos de dados encontrados em aplicações. Cada tipo de banco NoSQL oferece vantagens específicas de acordo com o cenário de uso, seja ele armazenamento de grandes volumes de dados, flexibilidade de estrutura, ou a habilidade de lidar com relacionamentos complexos.

2.1.8 MongoDB e Microsserviços

O MongoDB é um banco de dados orientado a documentos que armazena dados em documentos JSON-*like* (no formato BSON, internamente), o que permite uma maior flexibilidade na estruturação dos dados Chodorow (2013). Essa característica é essencial para aplicações que exigem a modelagem de dados de maneira dinâmica, sem a necessidade de definir previamente um esquema rígido.

No contexto do desenvolvimento de microsserviços, os bancos de dados não-relacionais, como o MongoDB, desempenham um papel importante devido à sua flexibilidade, escalabilidade e resiliência. Newman (2015b) destaca como essas características dos bancos NoSQL são importantes para garantir que os serviços possam operar de forma autônoma e eficiente em um ambiente de microsserviços. Uma das maiores vantagens de utilizar o MongoDB em microsserviços é sua capacidade de escalar horizontalmente. Newman (2015b) discute como o MongoDB, através do particionamento de dados (*sharding*), permite que a carga de trabalho seja distribuída entre múltiplos nós, o que resulta em um sistema capaz de lidar com grandes volumes de dados sem comprometer o desempenho. Diferente dos bancos de dados relacionais, que muitas vezes exigem estratégias complexas para escalar, o MongoDB facilita a adição de novos nós, garantindo que a escalabilidade ocorra de forma natural e com mínimo impacto na operação.

Para Newman (2015b), por utilizar modelo de dados orientado a documentos, o MongoDB oferece maior flexibilidade ao desenvolvimento de microsserviços, o que permite que cada microsserviço tenha controle total sobre sua própria estrutura de dados, sem a necessidade de depender de um esquema rígido. Isso se alinha perfeitamente com a filosofia dos microsserviços, no qual cada serviço deve ser autônomo e, ao permitir que diferentes serviços ajustem suas estruturas de dados conforme necessário, o MongoDB facilita a independência e a evolução contínua de cada serviço, sem impactar os demais.

2.2 Evolução e Inovação

2.2.1 Impacto nos Clubes de Assinatura

Clubes de assinatura são influenciados pelo meio digital. Uma ação de *marketing* bem planejada pode aumentar significativamente o acesso de usuários e a quantidade de

pedidos em um curto período de tempo. Utilizando o consumo de APIs e a arquitetura de microsserviços, é possível aumentar a escalabilidade do projeto com pouco esforço da equipe de desenvolvimento.

A adoção de microsserviços e APIs em uma plataforma de clube de assinatura traz uma série de benefícios que aprimoram tanto a operação quanto a experiência do usuário. A flexibilidade e a escalabilidade dos microsserviços permitem que cada parte da plataforma cresça de forma independente, otimizando a infraestrutura de acordo com a demanda, enquanto a resiliência garante que falhas em um serviço não comprometam o funcionamento completo da plataforma. Com APIs, a integração com sistemas existentes e a expansão para diferentes plataformas são facilitadas, possibilitando uma rápida inovação e atualização de funcionalidades sem a necessidade de grandes reescritas de código. Além disso, a modularização proporcionada pelos microsserviços e a facilidade de manutenção das APIs permitem maior reutilização de código e uma gestão eficiente de mudanças, resultando em um sistema mais ágil, estável e preparado para crescer com o mercado.

2.2.2 Desafios e Soluções

A arquitetura de microsserviços apresenta desafios devido à sua natureza distribuída. Gerenciar um sistema com múltiplos serviços independentes mostra-se complexo, tendo em vista que cada microsserviço opera isoladamente e precisa se comunicar com os demais por meio de redes, o que pode resultar em latência e falhas de comunicação. A confiabilidade do sistema também é afetada, pois a integração de vários serviços aumenta a chance de erros, especialmente em cenários em que a falha de um serviço pode impactar o desempenho de todo o sistema. Além disso, a complexidade de monitorar e depurar um sistema distribuído exige ferramentas especializadas para auditoria e análise de segurança, o que pode elevar os custos operacionais.

Outro grande desafio está relacionado à segurança e à manutenção. Cada microsserviço expõe sua API, ampliando a superfície de ataque e tornando o sistema mais vulnerável a invasões externas. Para mitigar essas ameaças, é necessário implementar medidas de segurança adicionais, como criptografia, o que pode impactar negativamente o desempenho do sistema. A testabilidade também se torna mais complicada, pois, embora seja possível testar os serviços individualmente, garantir a integração correta entre todos eles requer esforços consideráveis. Com isso, os custos de manutenção aumentam, principalmente à medida que o número de microsserviços cresce e a interdependência entre eles se torna mais complexa.

Para contornar os desafios, o uso de um *API Manager* é imprescindível. Um *API Manager* é uma plataforma que facilita a gestão e o controle de APIs, oferecendo funcionalidades como criação, publicação, autenticação e autorização de acessos. Ele também permite o monitoramento do desempenho das APIs, fornecendo dados de uso e identifi-

cando possíveis problemas. Além disso, o *API Manager* implementa controle de tráfego, como *rate limiting*, para evitar sobrecargas, garantindo segurança, escalabilidade e eficiência no uso das APIs em sistemas distribuídos. Há opções *open source* que são confiáveis e utilizadas por grande empresas ao redor do mundo, como é o caso da WSO2 (2005).

2.2.3 Tendências Futuras e Inovações

O uso de APIs e microsserviços continua a evoluir rapidamente, com várias inovações emergindo para lidar com os desafios de escalabilidade, manutenção e desempenho. Com o aumento da adoção de arquiteturas baseadas em nuvem, o modelo *serverless* e as ofertas inovadoras de provedores como AWS , Microsoft Azure e Google Cloud estão moldando o futuro das arquiteturas de software.

Em uma arquitetura *serverless*, os desenvolvedores podem focar na lógica de negócios de suas aplicações sem se preocupar com o gerenciamento de servidores ou infraestrutura subjacente. Provedores de nuvem, como AWS, Azure e Google Cloud, oferecem plataformas que gerenciam automaticamente o provisionamento, escalabilidade e manutenção dos servidores. Para Services (2020b), Azure (2020a), a AWS Lambda, Azure Functions e Google Cloud Functions são exemplos de serviços *serverless* que permitem aos desenvolvedores executar código em resposta a eventos sem a necessidade de gerenciamento de infraestrutura.

Uma de suas principais vantagens é a escalabilidade automática, onde os recursos são alocados dinamicamente conforme a demanda, o que significa que não há necessidade de configurar manualmente a infraestrutura para lidar com picos de tráfego. Isso reduz custos e aumenta a eficiência operacional, especialmente para aplicações que têm cargas de trabalho variáveis. Além disso, para Richardson (2016), como o modelo *serverless* cobra apenas pelo uso real dos recursos (modelo *pay-as-you-go*), as empresas podem reduzir significativamente seus custos com infraestrutura.

Por outro lado, as arquiteturas *serverless* também apresentam desafios. O tempo de inicialização (*cold start*) de funções *serverless* pode impactar a latência em algumas situações. Fowler (2018) diz que o controle sobre a infraestrutura subjacente é limitado, o que pode ser uma desvantagem para aplicações que exigem otimização detalhada de desempenho ou requisitos específicos de rede.

Um *Service Mesh* é uma camada de infraestrutura dedicada que gerencia a comunicação entre microsserviços, controlando como diferentes partes de uma aplicação compartilham dados. Para Istio (2018), em uma arquitetura de *Service Mesh*, um *proxy* é implementado junto a cada serviço para interceptar todas as comunicações entre eles, o que permite monitorar, otimizar e proteger essas interações de forma centralizada.

Com uma *Service Mesh*, os desenvolvedores conseguem configurar políticas de co-

municação sem alterar o código dos serviços, facilitando a implantação e operação em ambientes dinâmicos e distribuídos. Além disso, as ferramentas de observabilidade integradas permitem que as equipes de operações identifiquem problemas de desempenho, gargalos e falhas rapidamente, o que melhora a resiliência geral do sistema.

Com o aumento do IoT, o conceito de *Edge Computing* está se tornando cada vez mais relevante. De acordo com Gubbi et al. (2013), o IoT é um modelo de computação distribuída que traz o processamento e o armazenamento de dados para mais próximo da origem desses dados, ou seja, "na borda"(*edge*) da rede, em vez de depender totalmente de servidores centralizados ou na nuvem. Isso reduz a latência e a utilização de largura de banda, permitindo que os dados sejam processados localmente, antes de serem enviados para a nuvem ou *data centers* centrais.

Esse modelo é vantajoso para aplicações em tempo real, como veículos autônomos. Ao processar dados localmente, o *edge computing* permite respostas mais rápidas e eficientes, mesmo com limitações de conectividade com a nuvem.

Provedores de nuvem como AWS (com AWS IoT Greengrass) e Azure (com Azure IoT Edge) estão facilitando a integração da *edge computing* com arquiteturas de APIs e microsserviços. Para Services (2020a), Azure (2020b), essas plataformas permitem que o processamento e a análise de dados ocorram no *edge*, fornecendo baixa latência e melhor desempenho em dispositivos IoT. As APIs desempenham um papel importante nesse modelo, permitindo que dados sejam coletados, processados e distribuídos em tempo real entre dispositivos e a nuvem, promovendo uma arquitetura estável e escalável.

2.2.4 Impacto na Maturidade de Empresas

De acordo com Dragoni et al. (2017), a adoção da arquitetura de microsserviços exige que a organização atinja um nível adequado de maturidade tecnológica e organizacional. Um dos requisitos fundamentais é a implementação de uma cultura *DevOps* e a formação de times multifuncionais. As equipes devem ser organizadas em torno de capacidades de negócios, com a responsabilidade tanto pelo desenvolvimento quanto pela operação de cada microsserviço, seguindo o princípio "você constrói, você opera", o que facilita a manutenção e a evolução contínua dos serviços.

A automação total também é imprescindível. Para aproveitar plenamente a arquitetura de microsserviços, é necessário contar com *pipelines* de integração e entrega contínua (CI/CD) automatizados, permitindo que novas versões de microsserviços sejam implantadas de forma rápida e independente, sem comprometer o restante do sistema. A automação reduz o risco de erros e acelera o desenvolvimento.

É essencial dispor da infraestrutura e das ferramentas adequadas de contêinerização, como Docker, e de orquestração, como Kubernetes. São necessárias para garantir que

os microsserviços sejam implantados e gerenciados de maneira eficiente. Esses recursos também são fundamentais para garantir o monitoramento e a escalabilidade dos serviços, fatores essenciais para lidar com a complexidade dos sistemas distribuídos.

Mediante o exposto, é importante que a organização esteja preparada para enfrentar os desafios de gerenciamento de sistemas distribuídos. A arquitetura de microsserviços envolve a comunicação entre diversos serviços, o que aumenta a complexidade em termos de segurança, gestão de falhas e confiabilidade do sistema. Portanto, é crucial ter processos robustos para enfrentar esses desafios antes de migrar para essa arquitetura.

2.3 Clubes de assinatura

Seguindo o conceito fornecido pelo Sebrae (2023) clubes de promoção são grupos ou programas criados por empresas para oferecer benefícios exclusivos, descontos e promoções especiais a seus membros. Esses clubes podem funcionar como programas de fidelidade ou de associação, onde os clientes se inscrevem para receber vantagens contínuas. Eles incentivam os clientes a voltarem repetidamente, oferecendo recompensas por sua lealdade, o que cria um relacionamento mais forte entre a marca e o consumidor, aumentando a retenção de clientes.

Com descontos e ofertas exclusivas, os membros do clube são incentivados a fazer mais compras, impulsionando o volume de vendas. Ofertas temporárias e promoções especiais podem criar um senso de urgência que motiva as compras impulsivas. Ao se inscreverem em um clube de promoção, os clientes geralmente fornecem informações valiosas, como dados demográficos e preferências de compra. Isso permite que a empresa personalize ofertas e campanhas de *marketing*, tornando-as mais eficazes.

Clientes satisfeitos com os benefícios e recompensas de um clube de promoção são mais propensos a recomendar a marca a amigos e familiares, ampliando o alcance da empresa e atraindo novos clientes. Um clube de promoção bem estruturado pode diferenciar uma marca de seus concorrentes, oferecendo valor adicional que outros não oferecem, o que pode ser um fator decisivo na escolha do consumidor.

Clubes de promoção mantêm os clientes engajados com a marca através de comunicações regulares, como *newsletters*, atualizações de ofertas e eventos exclusivos. Esse engajamento contínuo mantém a marca na mente dos consumidores. Oferecer benefícios exclusivos e personalização através de um clube de promoção melhora a experiência geral do cliente. Clientes que se sentem valorizados e bem tratados têm uma percepção mais positiva da marca.

Membros de clubes de promoção são mais propensos a fornecer *feedback* sobre produtos e serviços, permitindo que a empresa melhore continuamente sua oferta com base

nas opiniões dos clientes mais fiéis. Em resumo, clubes de promoção são uma estratégia de *marketing* eficaz que não só aumenta a fidelidade e as vendas, mas também fortalece o relacionamento com os clientes, melhora a coleta de dados e diferencia a marca no mercado.

De acordo com Braga (2023), para um clube obter sucesso ele deve focar em cinco estratégias chave: captação e retenção de clientes, gestão eficiente do ciclo de vida do assinante, oferecer valor contínuo, inovação e adaptação e gestão financeira. Nos próximos parágrafos cada estratégia será explicada em conformidade com os ideais de Braga.

A captação e retenção de clientes são essenciais para o sucesso de qualquer clube de assinaturas. Realizar campanhas de *marketing* eficazes é o primeiro passo para atrair novos assinantes. Essas campanhas precisam ser bem direcionadas, usando diversos canais como redes sociais, *e-mail marketing* e publicidade *on-line*, para alcançar um público mais amplo e interessado. Além disso, oferecer uma experiência de excelência ao cliente é essencial para garantir que eles permaneçam por mais tempo no clube. Isso pode incluir atendimento ao cliente de alta qualidade, entregas pontuais e um serviço personalizado que atenda às necessidades específicas de cada assinante.

Gerenciar de forma eficiente o ciclo de vida do assinante é fundamental para manter o nível de satisfação e fidelidade dos clientes. Monitorar e analisar os dados dos assinantes permite entender melhor suas preferências e comportamentos, possibilitando a personalização da oferta. Programas de fidelidade e recompensas também são ferramentas poderosas para aumentar a retenção, incentivando os assinantes a permanecerem no clube e a recomendarem o serviço a outras pessoas. Estes programas podem incluir descontos exclusivos, brindes e acesso antecipado a novos produtos ou serviços.

Oferecer valor contínuo é outra estratégia crucial para o sucesso de um clube de assinaturas. Assegurar que os produtos ou serviços oferecidos sejam de alta qualidade e relevantes para os assinantes é essencial. Isso requer uma avaliação constante do mercado e uma comunicação aberta e transparente com os assinantes para entender suas necessidades e *feedback*. Manter uma comunicação constante e transparente ajuda a construir uma relação de confiança e a ajustar a oferta de acordo com as expectativas dos clientes, garantindo que eles vejam valor contínuo em sua assinatura.

Inovação e adaptação são componentes vitais para manter a competitividade e relevância no mercado. Estar sempre atualizado com as tendências do mercado e inovar na oferta de produtos ou serviços permite que o clube se diferencie da concorrência. Além disso, a capacidade de se adaptar rapidamente às mudanças nas preferências dos clientes e no ambiente de negócios é essencial. Isso pode incluir a introdução de novos produtos, a modificação de estratégias de *marketing* e a implementação de novas tecnologias para melhorar a experiência do assinante.

Uma gestão financeira rigorosa é indispensável para a viabilidade de um clube de assinaturas. Manter um controle rígido sobre os custos operacionais ajuda a garantir que o clube permaneça financeiramente saudável. Além disso, buscar parcerias estratégicas pode ser uma maneira eficaz de melhorar a oferta de valor aos assinantes e reduzir custos. Essas parcerias podem incluir acordos com fornecedores, colaborações com outras empresas e até mesmo programas de *co-branding*, que é a junção de duas ou mais marcas para entregar uma experiência inovadora, que podem oferecer benefícios adicionais aos assinantes e fortalecer a posição do clube no mercado.

3 Metodologia

Para a pesquisa, será adotado o modelo sugerido por PRODANOV e FREITAS (2013), que classifica o estudo com base na natureza, abordagem, procedimentos e objetivos. A Tabela 2 exibe as escolhas relacionadas aos critérios estabelecidos para cada categoria da metodologia de pesquisa deste trabalho.

Natureza	**Abordagem**	**Objetivo**	**Procedimentos**
Natureza aplicada	Abordagem qualitativa	Objetivo exploratório	Pesquisa bibliográfica

Tabela 2 – Tabela resumo da metodologia de pesquisa adotada

Serão adotados princípios da Metodologia Ágil para o desenvolvimento, com ajustes específicos em algumas fases devido ao tamanho reduzido da equipe e às particularidades do projeto.

3.1 Metodologia de Pesquisa

As metodologias empregadas neste trabalho foram separadas em duas categorias: pesquisa científica e desenvolvimento de *software*. Cada uma possui suas particularidades e requer uma abordagem metodológica distinta.

A seção de metodologia é essencial para a compreensão do trabalho, pois descreve, de maneira detalhada, o percurso metodológico seguido ao longo da pesquisa. Ela oferece uma visão clara dos procedimentos adotados, desde a coleta de dados até a análise, garantindo a transparência e a replicabilidade do estudo. A metodologia apresentada neste trabalho foi selecionada para atender aos objetivos propostos, garantindo que as técnicas e abordagens escolhidas fossem as mais adequadas para explorar e responder às questões de pesquisa levantadas. A seguir, são descritos os métodos utilizados, bem como as justificativas para suas escolhas.

Segundo Gil (2002), pesquisa é o procedimento racional e sistemático que tem como objetivo proporcionar respostas aos problemas que são propostos, podendo ela ser requerida quando não se dispõe de informação suficiente para responder ao problema. O objetivo desta seção é apresentar a caracterização da metodologia de pesquisa adotada no desenvolvimento do trabalho. Seu resumo pode ser observado na Tabela **??**.

Segundo PRODANOV e FREITAS (2013), uma pesquisa aplicada objetiva gerar conhecimentos para aplicação prática dirigidos à solução de problemas específicos. Este trabalho possui esta característica, já que tem como foco entender melhor o funcionamento das APIs e como utilizá-las no desenvolvimento de aplicações de *software*.

Para PRODANOV e FREITAS (2013), uma pesquisa de abordagem qualitativa não utiliza dados estatísticos como o centro do processo de análise de um problema, não tendo, portanto, a prioridade de numerar ou medir unidades, além de possuir uma maior preocupação com o processo do que com o produto. Sendo assim, este trabalho se caracteriza com esse tipo de abordagem.

De acordo com PRODANOV e FREITAS (2013), uma pesquisa é caracterizada como exploratória quando ela se encontra na fase preliminar e tem como finalidade proporcionar mais informações sobre o assunto alvo, além de contar com pesquisas bibliográficas. Este trabalho é caracterizado por tal objetivo, já que conta com levantamento bibliográfico acerca dos assuntos de interesse ao tema e visa levantar mais informações sobre o uso das APIs como ferramenta de desenvolvimento de *softwares*.

Ainda de acordo com PRODANOV e FREITAS (2013), os procedimentos técnicos são as maneiras pela qual obtemos os dados necessários para a elaboração da pesquisa. Portanto, este trabalho se caracteriza como pesquisa bibliográfica uma vez que possui como base material já publicado, como livros e publicações em periódicos, acerca dos conceitos e temas que envolvem o desenvolvimento de softwares utilizando APIs.

3.2 Metodologia de Desenvolvimento de Software

3.2.1 Metodologia Ágil

De acordo com Beck (2001), a metodologia ágil é um conjunto de princípios que guiam o desenvolvimento de *software* de maneira eficiente e flexível, priorizando a satisfação do cliente através da entrega contínua e antecipada de *software* valioso. Ele incentiva a adaptação às mudanças de requisitos, mesmo em estágios avançados do desenvolvimento, para oferecer vantagem competitiva ao cliente. O *software* funcional é entregue com frequência, em intervalos de semanas a meses, com preferência por prazos mais curtos.

A colaboração diária entre profissionais de negócios e desenvolvedores é fundamental, assim como a construção de projetos em torno de indivíduos motivados, fornecendo-lhes o ambiente e o suporte necessários e confiando em sua capacidade de realizar o trabalho. A comunicação face a face é considerada o método mais eficaz de transmitir informações dentro da equipe de desenvolvimento. O progresso é medido principalmente pelo *software* em funcionamento, e o desenvolvimento sustentável é promovido, permitindo que patrocinadores, desenvolvedores e usuários mantenham um ritmo constante de trabalho indefinidamente.

A atenção contínua à excelência técnica e ao bom design aprimora a agilidade, e a simplicidade, ou seja, a arte de maximizar a quantidade de trabalho não realizado, é

considerada essencial. As melhores arquiteturas, requisitos e designs emergem de equipes auto-organizadas, que, em intervalos regulares, refletem sobre como se tornar mais eficazes e ajustam seu comportamento de acordo.

A metodologia ágil propõe formas mais eficazes de desenvolver *software*. Como resultado, na tabela 3, os itens à esquerda são mais valorizados do que os itens à direita, embora estes também sejam importantes.

Mais valorizado	**Menos valorizado**
Indivíduos e interações	Processos e ferramentas
Software em funcionamento	Documentação abrangente
Colaboração com o cliente	Negociação de contratos
Responder a mudanças	Seguir um plano

Tabela 3 – Valores Ágeis

3.2.2 Modelo incremental

De acordo com Sommerville (2011), o desenvolvimento incremental consiste em criar uma versão inicial do sistema, disponibilizá-la para avaliação dos usuários, e continuar o processo através da criação de múltiplas versões, até que se chegue a um sistema final satisfatório. Nesse modelo, as atividades de especificação, desenvolvimento e validação ocorrem de forma integrada e contínua, permitindo um rápido retorno sobre todas as etapas do processo.

A entrega incremental de *software* envolve definir em detalhes os requisitos para o primeiro incremento do sistema e desenvolvê-lo sem permitir alterações durante esse processo. Após a conclusão de cada incremento, ele é entregue aos clientes, que podem colocá-lo em operação, experimentando o sistema e refinando suas necessidades para incrementos futuros. Essa abordagem permite a integração de novos incrementos aos anteriores, aprimorando a funcionalidade do sistema com cada entrega. A tabela 4 demonstra as vantagens e desafios do modelo de entrega incremental.

Para alguns sistemas, como aqueles muito grandes ou embutidos, e sistemas críticos que exigem análise detalhada de todos os requisitos, a entrega incremental pode não ser adequada. Nesses casos, um protótipo desenvolvido iterativamente pode ajudar a experimentar e ajustar os requisitos e o *design* do sistema antes de se comprometer com a versão final.

3.3 Requisitos

De acordo com Sommerville (2011), os requisitos de um sistema são as descrições do que o sistema deve fazer, os serviços que oferece e as restrições a seu funcionamento.

Vantagens	**Desafios**
Uso de incrementos iniciais como protótipos, ajudando a definir necessidades futuras.	Dificuldade em identificar recursos comuns devido à definição tardia dos requisitos.
Entrega antecipada de funcionalidades críticas, permitindo uso mais cedo.	Problemas em obter *feedback* útil de sistemas substitutos incompletos.
Facilidade de incorporar mudanças durante a etapa de desenvolvimento.	Conflitos com contratos que exigem especificação completa, incompatíveis com a abordagem incremental.
Foco nos testes das partes mais importantes, reduzindo falhas nessas áreas.	

Tabela 4 – Vantagens e Desafios do modelo Incremental

Esses requisitos refletem as necessidades dos clientes para um sistema que serve a uma finalidade determinada, como controlar um dispositivo, colocar um pedido ou encontrar informações.

Há dois tipos de requisitos: os requisitos de usuário, que são descrições em linguagem natural, acompanhadas de diagramas, que indicam os serviços que o sistema deve oferecer e as restrições sob as quais deve operar, e os requisitos de sistema, que são descrições mais detalhadas das funções, serviços e restrições do *software*, especificando exatamente o que deve ser implementado. O documento de requisitos do sistema, também conhecido como especificação funcional, serve como um acordo formal entre o comprador e os desenvolvedores do *software*, definindo as obrigações contratuais.

Neste trabalho abordaram-se dois tipos de requisitos: Requisitos Funcionais e Requisitos Não Funcionais. Requisitos funcionais especificam os serviços que o sistema deve oferecer, detalhando como ele deve reagir a entradas e se comportar em diferentes situações, incluindo o que não deve fazer e pode ser vista na tabela 5. Requisitos não funcionais são restrições que afetam esses serviços, como limitações de tempo, processos de desenvolvimento, e conformidade com normas, impactando o desempenho e a qualidade do sistema e consegue ser observado na tabela 6.

3.3.1 Elicitação dos requisitos

De acordo com Sommerville (2011), após a realização de um estudo inicial de viabilidade, o próximo passo na engenharia de requisitos é a elicitação e análise de requisitos. Nessa fase, engenheiros de *software* trabalham em conjunto com clientes e usuários finais para coletar informações essenciais sobre o domínio da aplicação, serviços esperados, desempenho e restrições do sistema.

ID	**Descrição**	**MoSCoW**
RF01	Permitir o cadastro de usuário para criação de conta e acesso à plataforma mediante pagamento.	Must
RF02	Permitir o cadastro de administrador para adicionar soluções de investimento na plataforma.	Must
RF03	Oferecer a opção de cadastro via Facebook (Meta).	Should
RF04	Incluir uma tela de pagamentos para a conclusão da assinatura dos planos.	Must
RF05	Implementar uma tela de *login* para acesso ao sistema.	Must
RF06	Implementar uma tela de cadastro para novos usuários e administradores.	Must
RF07	Implementar uma tela de recuperação de senha para usuários que esqueceram suas credenciais.	Must
RF08	Implementar uma tela de gerenciamento de produtos para administradores adicionarem, editarem ou removerem soluções de investimento.	Must
RF09	Implementar uma tela para visualização dos produtos cadastrados, acessível tanto por usuários quanto por administradores.	Must
RF10	Implementar uma tela de gerenciamento de usuários, permitindo que administradores gerenciem contas de usuário.	Should
RF11	Implementar uma tela Quem Somos para apresentar informações sobre a plataforma e sua missão.	Could
RF12	Implementar uma tela Home que serve como ponto de partida para os usuários navegarem na plataforma.	Must
RF13	Implementar uma tela de contato para que usuários possam entrar em contato com a equipe de suporte.	Would

Tabela 5 – Requisitos Funcionais da Plataforma InvestMinds

A elicitação de requisitos envolve a participação de diversos *stakeholders*, que são todas as pessoas com influência direta ou indireta sobre o sistema, como usuários finais, engenheiros de sistemas relacionados, gerentes de negócios e especialistas de domínio. A interação com esses *stakeholders* é crucial para descobrir e entender os requisitos do sistema de forma abrangente.

O processo de elicitação inclui atividades como a descoberta, onde os requisitos são

ID	Descrição	Classificação
RNF01	O sistema deve garantir que todas as transações financeiras sejam realizadas com segurança.	Segurança
RNF02	O site deve estar disponível 99,9% do tempo para garantir acessibilidade.	Confiabilidade
RNF03	O tempo de resposta para carregamento das páginas não deve exceder 2 segundos.	Usabilidade
RNF04	A plataforma deve estar em conformidade com as normas de proteção de dados (LGPD).	Segurança
RNF05	O sistema deve suportar múltiplos usuários simultaneamente sem perda de desempenho.	Usabilidade
RNF06	O sistema deve ser escalável para suportar um número crescente de usuários e dados.	Manutenibilidade
RNF07	O banco de dados deve ser otimizado para consultas rápidas, garantindo eficiência no acesso aos dados.	Usabilidade
RNF08	A interface do usuário deve ser intuitiva e fácil de navegar, seguindo as melhores práticas de design de UX/UI.	Usabilidade
RNF09	O sistema deve ser compatível com os principais navegadores, garantindo uma experiência consistente para todos os usuários.	Usabilidade

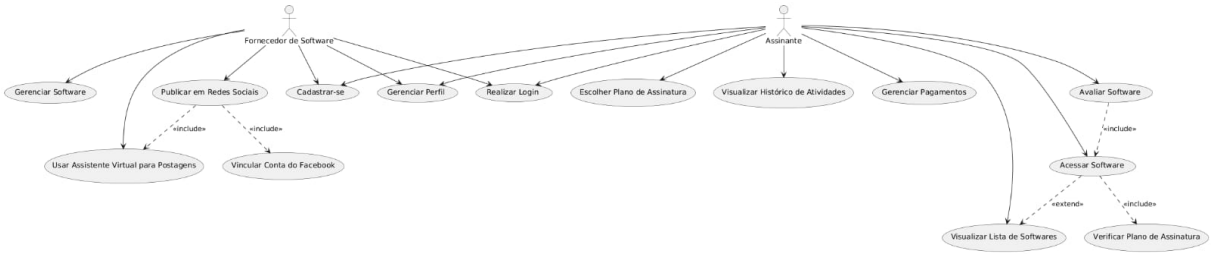
Tabela 6 – Requisitos não funcionais da plataforma InvestMinds. Fonte: Autores

identificados; classificação e organização, que agrupa requisitos relacionados; priorização e negociação, para resolver conflitos entre *stakeholders*; e especificação, onde os requisitos são documentados para o próximo ciclo de desenvolvimento.

De acordo com Sommerville (2011), entrevistas formais ou informais com *stakeholders* do sistema são uma parte fundamental na maioria dos processos de engenharia de requisitos. Durante as entrevistas, a equipe de engenharia de requisitos faz perguntas aos *stakeholders* sobre o sistema atual e o sistema a ser desenvolvido, permitindo que os requisitos sejam identificados a partir das respostas. As entrevistas podem ser classificadas em dois tipos principais: entrevistas fechadas, onde o *stakeholder* responde a um conjunto de perguntas predefinidas, e entrevistas abertas, nas quais não há uma agenda fixa, permitindo uma exploração mais ampla das necessidades dos *stakeholders*.

Neste trabalho, optou-se por conduzir entrevistas abertas. Esse formato permitiu uma abordagem mais flexível, onde foi possível explorar diversas questões com os *stakehol-*

Figura 2 – Caso de uso da InvestMinds.



ders, desenvolvendo uma compreensão mais profunda e detalhada de suas necessidades e expectativas em relação ao sistema.

De acordo com Cambridge (2021), o *brainstorming* é uma maneira de gerar novas ideias a partir de um grupo de pessoas por meio de uma discussão na qual elas fazem várias sugestões e as melhores são adotadas.

Durante uma sessão de *brainstorming*, os participantes são incentivados a compartilhar todas as ideias que vêm à mente, sem se preocupar com julgamentos ou críticas imediatas. A ideia é criar um ambiente aberto e livre, onde a quantidade de ideias é priorizada sobre a qualidade inicial. Isso permite que até mesmo as sugestões inusitadas possam desencadear outras ideias mais viáveis ou inovadoras.

Nesse trabalho a técnica de *brainstorming* foi utilizada tanto em conjunto com o *stakeholder* quanto entre a própria equipe de desenvolvimento. O *brainstorming* mostrou-se eficaz porque combina a diversidade de perspectivas dos participantes, aproveitando a inteligência coletiva para encontrar soluções. Ao final do processo, as melhores ideias foram selecionadas para serem desenvolvidas e implementadas.

Segundo Sommerville (2011), um caso de uso identifica os atores que participam de uma interação e nomeia o tipo de interação. Em seguida, essa identificação é complementada com informações adicionais que detalham como ocorre a interação com o sistema. Essas informações podem ser apresentadas como uma descrição em texto ou através de modelos gráficos, como diagramas de sequência ou de estados da UML. O diagrama de caso de uso pode ser visto na figura 2.

3.3.2 Priorização dos Requisitos

Em concordância com Oliveira (2014), o MoSCoW é uma maneira de classificar e priorizar requisitos para inclusão em um sistema de informação. A técnica consiste em categorizar os requisitos em *must*, *should*, *could* e *would*, que em português significa deve (imprescindível), deve (recomenda-se), poderia e gostaria. A tabela 5 demonstra a priorização dos requisitos funcionais elicitados.

4 Desenvolvimento

4.1 Desenvolvimento da Plataforma

O desenvolvimento da plataforma utilizou diversas tecnologias modernas tanto no *frontend* quanto no *backend*, além de integrações com serviços externos para garantir escalabilidade, segurança e funcionalidade. A seguir, apresentamos uma visão detalhada das principais tecnologias e práticas adotadas durante o desenvolvimento.

4.1.1 Frontend

O *frontend* da plataforma foi desenvolvido utilizando a biblioteca React, com suporte de outras bibliotecas para facilitar a construção da interface do usuário e a gestão de estado. Dentre elas, destacam-se:

- **TailwindCSS**: Utilizado para a estilização de componentes. TailwindCSS permite criar interfaces sem a necessidade de escrever CSS customizado extensivamente.
- **Ant Design (Antd)**: Fornece uma variedade de componentes, como tabelas, formulários e modais, acelerando o desenvolvimento da interface.
- **Redux**: Utilizado para o gerenciamento de estado global da aplicação, permitindo o compartilhamento de dados entre componentes de maneira centralizada e previsível. Isso foi particularmente importante para a manipulação de dados do usuário e a integração com APIs.
- **React Router**: Implementado para o roteamento da aplicação, permitindo a navegação entre diferentes páginas e a aplicação de regras específicas, como rotas baseadas em permissões de usuário (*role-based routing*).
- **Axios**: Utilizado para fazer requisições HTTPS assíncronas às APIs, facilitando a comunicação entre o *frontend* e os microsserviços *backend*.

O TailwindCSS foi empregado no frontend para facilitar a estilização dos componentes da interface de usuário. Por ser um framework de utilidades CSS, ele permitiu criar classes específicas que já possuem estilos predefinidos, eliminando a necessidade de escrever grandes quantidades de CSS customizado. Isso trouxe agilidade e eficiência ao processo de desenvolvimento, permitindo que o time focasse mais na lógica da aplicação e na construção de componentes reutilizáveis. Além disso, o uso de TailwindCSS facilitou

a consistência visual da interface, uma vez que as classes são modulares e garantem que o design siga um padrão sem que seja necessário definir estilos manualmente.

Outro ponto importante no uso do TailwindCSS foi a sua integração direta com o JavaScript e frameworks como o React. Como as classes são altamente configuráveis, isso permitiu personalizar rapidamente componentes sem recarregar a página, fazendo atualizações dinâmicas diretamente no código JSX. Isso também ajudou na manutenção do código, pois as classes utilitárias de Tailwind são declarativas e fáceis de modificar, o que acelerou a iteração durante o desenvolvimento, especialmente quando ajustes no layout ou nas cores eram necessários.

O Ant Design (Antd) foi importante na construção rápida e eficiente de interfaces de usuário modernas e responsivas. Ao oferecer uma biblioteca de componentes prontos, como botões, tabelas, modais e formulários, o Antd possibilitou uma redução significativa no tempo necessário para criar elementos visuais complexos. Esses componentes possuem um design elegante e são altamente configuráveis, permitindo que a equipe aplicasse customizações com facilidade quando necessário. Isso se mostrou particularmente vantajoso em áreas da aplicação que exigiam interação com grandes volumes de dados, como tabelas e dashboards, que já vinham otimizados para oferecer uma boa experiência de usuário.

Além disso, o Antd possui uma excelente documentação e uma comunidade ativa, o que facilitou a resolução de problemas e a implementação de novas funcionalidades. Com sua integração fácil com o React, o Antd se encaixou bem no ciclo de desenvolvimento da aplicação, proporcionando também compatibilidade com temas personalizados e suporte ao modo escuro. Dessa forma, o uso do Antd acelerou a implementação de interfaces sofisticadas e interativas, sem sacrificar o desempenho ou a flexibilidade.

O Redux foi utilizado como solução para o gerenciamento global de estados no frontend, permitindo centralizar os dados e suas atualizações. Em um contexto de múltiplos componentes interagindo com dados compartilhados, o Redux foi essencial para manter a previsibilidade do fluxo de informações. Cada atualização de estado disparada em qualquer ponto da aplicação foi gerenciada de forma consistente pelo Redux, o que facilitou a depuração e manteve o código mais organizado. O uso de *actions* e *reducers* permitiu que alterações no estado da aplicação fossem realizadas de maneira explícita e rastreável, o que foi fundamental, principalmente quando houve a necessidade de manipular dados sensíveis, como informações de usuários.

Outra grande vantagem do Redux foi sua capacidade de integrar a lógica de estado com APIs externas. Isso foi feito utilizando *middlewares* como o Redux Thunk, que permitiram gerenciar requisições assíncronas de maneira mais organizada e eficiente. Dessa forma, o Redux se mostrou uma ferramenta importante para lidar com o gerenciamento de dados em tempo real, proporcionando uma interface de usuário mais fluida e responsiva.

O React Router foi empregado para gerenciar o roteamento da aplicação, permitindo uma navegação fluida entre diferentes páginas. Ao dividir a interface em várias rotas, o React Router possibilitou que diferentes componentes fossem carregados dinamicamente à medida que o usuário navegava na aplicação, o que foi essencial para manter a aplicação modular e escalável. Além disso, o React Router oferece suporte para aninhamento de rotas, facilitando a criação de estruturas mais complexas, como páginas de perfil de usuário ou dashboards com múltiplas seções.

Outro recurso relevante utilizado do React Router foi a implementação de rotas condicionais com base em permissões de usuário, conhecido como *role-based routing*. Isso permitiu que diferentes tipos de usuários, como administradores e usuários comuns, acessassem diferentes partes da aplicação com base em suas permissões. O React Router também facilitou a integração com componentes que controlam a autenticação, garantindo que as rotas protegidas fossem acessadas apenas por usuários autenticados, tornando a navegação mais segura e eficiente.

O Axios foi utilizado como ferramenta principal para realizar requisições HTTP assíncronas entre o frontend e os microsserviços backend. Com ele, foi possível enviar e receber dados de forma eficiente, fazendo chamadas GET, POST, PUT e DELETE para interagir com APIs RESTful. A facilidade de uso do Axios permitiu que as requisições fossem estruturadas com interceptadores para gerenciar tokens de autenticação e capturar erros de forma padronizada, o que foi essencial para manter a segurança e a integridade das informações trocadas entre o frontend e o backend.

Além disso, o Axios possui suporte para Promises, o que facilitou o controle de fluxos assíncronos dentro da aplicação. Dessa forma, foi possível exibir carregamentos dinâmicos ou mensagens de erro ao usuário de maneira eficaz, aprimorando a experiência de uso. Outra vantagem do Axios foi sua flexibilidade para configurar as requisições de forma global, como a definição de URLs base ou cabeçalhos personalizados, o que otimizou a comunicação com as diferentes APIs e microsserviços utilizados na aplicação.

4.1.2 Arquitetura do Frontend

As principais funcionalidades do *frontend* incluíram a criação de componentes reutilizáveis e a integração com APIs usando o Axios. O Redux foi fundamental para gerenciar o estado da aplicação, enquanto o React Router possibilitou o roteamento dinâmico com base nas permissões dos usuários.

A implementação do *frontend* utilizou uma estrutura de pastas que visa organizar o projeto de maneira modular e escalável, garantindo a separação de responsabilidades e a manutenibilidade do código.

Como observável na Figura 3, no diretório `src/`, várias subpastas são usadas para

Figura 3 – Estrutura de pastas do *frontend*

```
src/
├── assets/
│   └── <-- Imagens, arquivo auxiliares etc -->
├── components/
│   └── <-- Outros componentes reutilizáveis -->
├── layouts/
│   └── <-- Layouts do sistema -->
├── pages/
│   └── <-- Páginas do sistema -->
├── routes/
│   └── <-- Rotas do sistema -->
├── services/
│   └── <-- Configurações de services do sistema -->
├── store/
│   └── <-- Configuração do Redux -->
├── App.jsx
└── main.jsx
```

agrupar diferentes aspectos da aplicação. A pasta `assets/` contém arquivos auxiliares como imagens, enquanto `components/` é dedicada a componentes reutilizáveis. O diretório `layouts/` organiza os *layouts* principais do sistema, e `pages/` armazena as páginas da aplicação. O roteamento é gerenciado em `routes/`, e as lógicas de negócios e integrações externas estão centralizadas em `services/`. A configuração do estado global da aplicação, gerenciada pelo Redux, é feita dentro da pasta `store/`. Os arquivos principais do sistema, como `App.jsx` e `main.jsx`, servem como ponto de entrada da aplicação. Essa estrutura modular e organizada facilita o desenvolvimento de novas funcionalidades e a manutenção contínua do sistema.

4.1.3 Boas Práticas de Desenvolvimento

No desenvolvimento da plataforma, várias boas práticas foram adotadas para garantir que o código fosse legível, reutilizável e preparado para novas implementações. O React tem como um de seus pilares a componentização. Através da componentização, é possível construir blocos atômicos de código, que são facilmente identificáveis, reutilizáveis e mantidos de forma independente, garantindo que o sistema seja escalável e modular. No projeto, houve uma preocupação central em organizar os componentes de maneira clara, objetiva e isolada, de modo que cada componente tivesse suas dependências organizadas e seu escopo bem definido.

A Figura 4 exemplifica a estrutura do componente AssistantChat, responsável por exibir um *chat* de texto integrado à API da OpenAI. Esse componente foi projetado de maneira reutilizável, permitindo sua integração em diferentes partes da plataforma sem a

necessidade de reescrita. Além disso, a lógica de negócios, como a comunicação com a API da OpenAI, foi separada da camada de apresentação, garantindo que a interface gráfica permaneça simples e fácil de ajustar. Isso segue o princípio da separação de responsabilidades, que facilita a manutenabilidade e garante que as mudanças possam ser feitas em um único lugar, refletindo-se em todos os pontos onde o componente é utilizado.

Figura 4 – Estrutura de pastas dos componentes

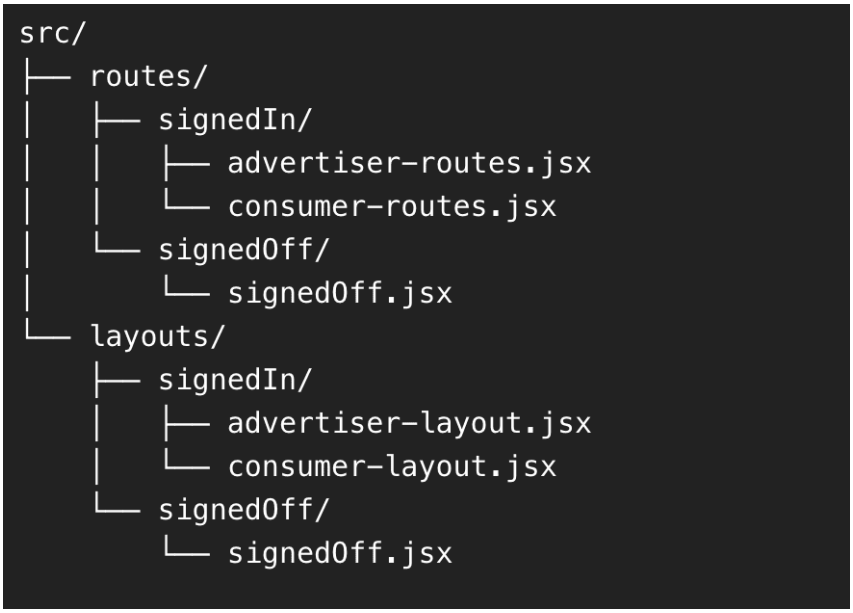
4.1.4 Roteamento e Gerenciamento de Acesso

Outro ponto essencial no desenvolvimento da plataforma é o roteamento. A Figura 5 ilustra como as rotas são organizadas de acordo com o estado de autenticação do usuário e seu papel no sistema. Para usuários não autenticados, há um conjunto de rotas separadas (SignedOff), enquanto usuários autenticados têm rotas específicas conforme seu papel — consumidor (Consumer) ou anunciante (Advertiser). Essa segmentação garante que cada usuário acesse apenas os recursos adequados à sua função, promovendo segurança e personalização na experiência do usuário. Além disso, a utilização de *layouts* específicos para cada papel facilita a manutenção e a atualização de funcionalidades sem impactar outras áreas da plataforma.

Como observável na Figura 5, as rotas da plataforma são separadas para os usuários não autenticados no sistema (SignedOff), para os autenticados (SignedIn) e de acordo também com o papel do usuário autenticado, seja ele consumidor (Consumer) ou anunciante (Advertiser), o que garante o acesso independente para cada categoria aos recursos do sistema e com base em suas respectivas permissões. Além disso, o uso de *layouts* para cada tipo de usuário garante uma experiência única e personalizável para cada categoria, e com manutenção independente. No trecho de código 1, é possível observar como o Redux e react-router atuam juntos para determinar o que estará disponível para o usuário da plataforma:

O código 1 exemplifica como o Redux e o react-router trabalham em conjunto para definir quais rotas estarão disponíveis para o usuário com base no seu papel. A função RoutesMapper é responsável por mapear o papel do usuário para o conjunto apropriado de rotas, enquanto o Redux, através do *hook* useSelector, recupera as informações do

Figura 5 – Estrutura de pastas do roteamento




```jsx
const RoutesMapper = (role) => {
  switch (role) {
    case "consumer":
      return consumerRouter;
    case "advertiser":
      return advertiserRouter;
    default:
      return signedOffRouter;
  }
};

const App = () => {
  const loggedUser = useSelector((state) => state.user);
  const dispatch = useDispatch();

  useLayoutEffect(() => {
    const { jwt } = loggedUser;
    const token = localStorage.getItem("access_token");
    if (!token && !jwt) return;
    const userInfo = jwtDecode(token);
    dispatch(setUser({ user: { ...userInfo, jwt: token } }));
  }, []);

  return <RouterProvider router={RoutesMapper(loggedUser.role)}/>;
};
```

Código 1 – Redux e React-Router

usuário armazenadas globalmente na aplicação. Essa arquitetura permite que o sistema seja facilmente extensível para novos papéis de usuário e garante que as rotas existentes sejam mantidas de forma organizada e segura.

4.1.5 Backend for Frontend (BFF)

Mediante o exposto, com a adoção do padrão *Backend for Frontend*, o sistema torna-se mais modular e eficiente. O BFF atua como uma camada intermediária entre o *frontend* e os microsserviços *backend*, garantindo que cada interface de usuário tenha acesso a uma API otimizada para suas necessidades específicas. Isso resulta em uma arquitetura onde cada cliente (seja *web*, *mobile* ou outro) recebe uma interface de API personalizada, simplificando a comunicação e aumentando a performance da aplicação.

4.1.6 Backend

O *backend* foi desenvolvido utilizando Node.js com o *framework* Express, adotando um modelo de microsserviços para garantir escalabilidade e flexibilidade. Cada microsserviço foi responsável por uma função específica da plataforma, como autenticação, gerenciamento do catálogo de *softwares*, pagamentos e comunicação com o *frontend*. As bibliotecas e SDKs principais utilizados no *backend* incluem:

- **JWT (JSON Web Token)**: Utilizado para a autenticação de usuários. Tokens JWT foram criados a partir de um hash seguro para garantir a validade e a integridade das sessões.
- **Mongoose**: Utilizado como uma biblioteca de modelagem de dados para o **MongoDB**, permitindo a definição de esquemas de dados e a interação com o banco de dados de maneira eficiente.
- **Stripe SDK**: Integrado para gerenciar os pagamentos e assinaturas dos usuários, além da implementação de *webhooks* que monitoram eventos do Stripe, como falhas de pagamento e atualizações de assinaturas.
- **OpenAI SDK**: Utilizado para integração com o serviço de *chat*, fornecendo uma interface de conversação dinâmica para os usuários da plataforma.

O JWT foi empregado no *backend* como a principal solução para autenticação e autorização de usuários. Cada vez que um usuário fazia login, um *token* JWT era gerado, contendo as informações essenciais da sessão, como o ID do usuário e suas permissões. Esse *token* foi assinado com um segredo ou chave privada, o que garantiu sua validade e

integridade. Ao ser enviado junto com cada requisição subsequente, o *token* JWT permitiu que a autenticação fosse feita de maneira segura e sem a necessidade de armazenar informações de sessão no servidor, promovendo uma arquitetura mais escalável.

Além disso, o JWT trouxe flexibilidade na implementação de permissões específicas, permitindo que diferentes níveis de acesso fossem atribuídos a diferentes tipos de usuários, como administradores ou assinantes comuns. O *backend* podia facilmente verificar as permissões contidas no token para conceder ou negar acesso a certas rotas ou funcionalidades da aplicação, como a administração de produtos ou gerenciamento de pagamentos. Esse mecanismo de segurança aumentou a eficiência do sistema, garantindo que apenas usuários autenticados e autorizados pudessem realizar ações críticas.

O Mongoose foi utilizado no *backend* para interagir com o banco de dados MongoDB, fornecendo uma maneira robusta de modelar os dados da aplicação. Através de seus esquemas, foi possível definir de maneira clara a estrutura dos documentos armazenados no banco de dados, incluindo as validações e relacionamentos necessários. Isso facilitou a consistência dos dados ao longo da aplicação, já que cada interação com o banco passava pelas regras definidas nos esquemas Mongoose, evitando erros e garantindo que as informações armazenadas seguissem um formato padronizado.

O Mongoose também ofereceu diversas funcionalidades que tornaram o gerenciamento de dados mais eficiente, como métodos para consultas avançadas, atualizações e a criação de *middlewares*. Por exemplo, foi possível implementar ganchos (*hooks*) que executavam funções antes ou depois de certas operações no banco de dados, como ao salvar ou atualizar um documento. Isso permitiu a automatização de processos, como a hash de senhas antes de armazená-las no banco, contribuindo para a segurança e integridade dos dados dos usuários.

O Stripe SDK foi integrado ao *backend* para gerenciar os pagamentos e assinaturas dos usuários da plataforma. Com essa ferramenta, foi possível implementar de forma simples e segura o processamento de pagamentos recorrentes e pontuais, oferecendo uma experiência de pagamento fluida para os assinantes. Além de possibilitar a criação de planos de assinatura personalizados, o Stripe também forneceu uma série de APIs e ferramentas que facilitaram o gerenciamento de dados financeiros e relatórios sobre as transações realizadas, além da aplicação de tarifas e impostos automaticamente.

Outra funcionalidade importante do Stripe SDK foi a integração de *webhooks*, que permitiu monitorar eventos em tempo real, como atualizações em assinaturas, pagamentos bem-sucedidos ou falhas de pagamento. Isso possibilitou a automação de respostas dentro da plataforma, como a suspensão de contas em caso de falhas de pagamento ou a emissão de recibos de pagamento quando a transação era concluída. Esse nível de integração com o Stripe trouxe uma camada extra de confiabilidade e automação para a plataforma, permitindo o gerenciamento eficiente das assinaturas sem a necessidade de intervenção

manual.

O OpenAI SDK foi utilizado no *backend* para integrar funcionalidades de IA conversacional na plataforma. Com essa solução, foi possível criar uma interface de *chat* dinâmica, permitindo que os usuários interagissem com um assistente virtual em tempo real. O OpenAI SDK facilitou a comunicação com os modelos de linguagem da OpenAI, fornecendo respostas em linguagem natural a partir das interações dos usuários. Isso trouxe uma camada de interatividade à plataforma, enriquecendo a experiência do usuário ao oferecer suporte personalizado ou respostas automatizadas para dúvidas sobre produtos e serviços.

Além das funcionalidades de conversação, o OpenAI SDK também foi utilizado para realizar tarefas mais complexas, como a geração de conteúdo dinâmico, sugestões automatizadas ou até a criação de resumos e análises de textos longos. Isso ampliou as possibilidades de aplicação da inteligência artificial na plataforma, não apenas para o suporte ao cliente, mas também para outras funcionalidades relacionadas à personalização de conteúdo e otimização de processos internos. A integração com o OpenAI SDK ofereceu uma interface intuitiva para acessar e utilizar esses serviços de IA no backend, facilitando o desenvolvimento dessas funcionalidades avançadas.

O *backend* foi organizado seguindo um modelo baseado em domínios, onde cada entidade (como user) possuía sua própria estrutura de rotas, *controllers*, serviços e *models*, facilitando a organização e a escalabilidade do projeto.

A implementação do *backend* teve como objetivo principal garantir a simplicidade e modularidade dos microsserviços. Foram desenvolvidos quatro microsserviços independentes: Autenticação, Catálogo de *Softwares*, Pagamentos e o BFF. Cada microsserviço foi projetado para executar uma responsabilidade específica e isolada, o que facilita a manutenção e a escalabilidade do sistema. A base do desenvolvimento girou em torno da biblioteca `Express.js`, um framework leve para Node.js que fornece recursos para a construção de APIs RESTful.

A estrutura geral do *backend* seguiu o princípio da separação de responsabilidades, onde cada microsserviço possui seu próprio conjunto de arquivos organizados de maneira modular, como apresentado na Figura 6.

Nesta estrutura, o diretório `config/` contém as configurações globais, como integração com SDKs e configurações de ambiente. A pasta `middlewares/` concentra os *middlewares* utilizados nas rotas para tratamento de autenticação, por exemplo. A pasta genérica `domínios/` organiza cada domínio da aplicação de forma independente, separando a lógica de negócios em controladores, modelos, rotas e serviços.

Cada domínio segue a organização mostrada na Figura 7, onde os arquivos essenciais para o funcionamento do microsserviço estão dispostos de forma clara e modular.

Figura 6 – Estrutura de pastas do *backend*

```
src/
├── config/
│   └── <-- Configuração de SDKs -->
├── middlewares/
│   └── <-- Middlewares para as rotas -->
└── <domínios>/
    └── <-- Domínios com seus contextos  -->
index.js
```

O arquivo `user-controller.js` lida com as requisições HTTPS e interage com os outros componentes. O `user-model.js` define a modelagem de dados para o MongoDB, o `user-routes.js` configura as rotas da API, enquanto o `user-service.js` contém a lógica de negócios.

Figura 7 – Estrutura das pastas de domínio

```
src/
└── user/
    ├── user-controller.js
    ├── user-model.js
    ├── user-routes.js
    └── user-service.js
index.js
```

Com essa organização, o *backend* é capaz de evoluir de forma independente em cada microsserviço, garantindo que novos recursos sejam adicionados sem impacto nas demais partes do sistema. Essa modularidade também facilita a aplicação de testes unitários em cada componente, além de melhorar a capacidade de escalar horizontalmente, adicionando novos serviços conforme a demanda aumenta.

4.1.7 Banco de Dados

A plataforma foi construída utilizando o banco de dados **MongoDB**, hospedado no serviço **MongoDB Atlas**, para garantir alta disponibilidade e escalabilidade. O banco de dados foi modelado utilizando a biblioteca **Mongoose**, permitindo a definição de

esquemas de dados flexíveis. A seguir, é apresentado o dicionário de dados das coleções implementadas.

A modelagem de usuário possui uma coleção *User* que armazena as informações dos usuários da plataforma, incluindo tanto consumidores quanto anunciantes. A modelagem foi feita de forma a suportar diferentes formas de autenticação (e-mail e Facebook) e perfis de usuário. A seguir, o dicionário de dados da coleção *User*:

- **email**: *String*, obrigatório, único. E-mail do usuário.
- **password**: *String*, obrigatório se o usuário não estiver autenticado via Facebook. Senha de acesso.
- **name**: *String*, obrigatório. Nome completo do usuário.
- **cpf**: *String*, obrigatório e único se o usuário não estiver autenticado via Facebook. Cadastro de Pessoa Física (CPF) do usuário.
- **facebookId**: *String*, opcional, único. Identificador do Facebook para login via Facebook.
- **role**: *String*, obrigatório, valores possíveis: *"consumer"* ou *"advertiser"*. Define o papel do usuário na plataforma. Por padrão, o valor é *"consumer"*.i
- **facebookPages**: *Array de objetos*. Lista de páginas do Facebook gerenciadas pelo usuário. Cada objeto contém:
  - **pageId**: *String*, obrigatório. Identificador da página no Facebook.
  - **pageName**: *String*, obrigatório. Nome da página no Facebook.
  - **pageToken**: *String*, obrigatório. *Token* de acesso à página no Facebook.
  - **category**: *String*, opcional. Categoria da página no Facebook.
- **facebookToken**: *String*, opcional. *Token* geral de autenticação no Facebook.
- **picture**: *String*, opcional. *Link* da foto do perfil do usuário.
- **subscriptions**: *Array de ObjectId*, referência para a coleção *Subscription*. Lista de assinaturas ativas do usuário.
- **threadId**: *String*, opcional. Identificador do usuário em conversas no sistema de chat no OpeanAI.
- **passwordResetToken**: *String*, opcional. *Token* utilizado para redefinição de senha.
- **passwordResetExpires**: *Date*, opcional. Data de expiração do *token* de redefinição de senha.

- **timestamps**: *Date*. Armazena as datas de criação e atualização dos registros.

A modelagem de assinatura possui uma coleção *Subscription* que gerencia as informações das assinaturas de usuários, permitindo que a plataforma ofereça diferentes níveis de serviços, como planos gratuitos ou pagos. A seguir, o dicionário de dados da coleção *Subscription*:

- **subscriptionId**: *String*, opcional. Identificador da assinatura no Stripe.
- **price**: *Number*, obrigatório, valor padrão: `0`. Valor da assinatura.
- **category**: *String*, obrigatório, valores possíveis: *"Free"*, *"Basic"*, *"Premium"*, *"VIP"*. Categoria do plano de assinatura.
- **endDate**: *Date*, opcional. Data de término da assinatura.
- **isActive**: *Boolean*, valor padrão: *true*. Indica se a assinatura está ativa.
- **invoiceUrl**: *String*, opcional. URL da fatura gerada para a assinatura.
- **owner**: *ObjectId*, referência para a coleção *User*. Dono da assinatura.
- **timestamps**: *Date*. Armazena as datas de criação e atualização dos registros.

A modelagem de catálogo de software possui uma coleção *SoftwareCatalog* que armazena as informações dos softwares cadastrados na plataforma. Cada software pode ser associado a diferentes categorias e níveis de assinatura, além de conter dados relacionados ao proprietário e sua classificação. A seguir, o dicionário de dados da coleção *SoftwareCatalog*:

- **name**: *String*, obrigatório. Nome do software.
- **description**: *String*, opcional, valor padrão: . Descrição do software.
- **external_link**: *String*, obrigatório. Link externo para o software.
- **categories**: *Array de Strings*, obrigatório. Lista de categorias nas quais o software se encaixa.
- **subscription_level**: *String*, obrigatório, valores possíveis: *"Free"*, *"Basic"*, *"Premium"*, *"VIP"*. Nível de assinatura necessário para acessar o software.
- **owner**: *ObjectId*, obrigatório, referência para a coleção `User`. Dono do software.
- **images**: *Array de Strings*, opcional. URLs das imagens associadas ao software.

- **tags**: *Array de Strings*, opcional. Palavras-chave associadas ao software.
- **price**: *Number*, opcional, valor padrão: `0`. Preço do software.
- **rating**: *Number*, opcional, valor mínimo: 0, valor máximo: 5. Classificação média do software com base nas avaliações dos usuários.
- **isArchived**: *Boolean*, opcional, valor padrão: *false*. Indica se o software está arquivado.
- **timestamps**: *Date*. Armazena as datas de criação e atualização dos registros.

A modelagem de avaliações possui uma coleção *Review* que armazena avaliações feitas por usuários sobre os *softwares* cadastrados na plataforma. Cada avaliação inclui uma nota, um comentário e a opção de ser anônima. A seguir, o dicionário de dados da coleção *Review*:

- **software**: *ObjectId*, obrigatório, referência para a coleção *SoftwareCatalog*. Identifica o software avaliado.
- **user**: *ObjectId*, obrigatório, referência para a coleção *User*. Identifica o usuário que fez a avaliação.
- **rating**: *Number*, obrigatório, valor mínimo: 1, valor máximo: 5. Avaliação numérica atribuída ao *software*.
- **comment**: *String*, opcional, valor padrão: . Comentário adicional deixado pelo usuário sobre o *software*.
- **isAnonymous**: *Boolean*, valor padrão: `false`. Indica se a avaliação foi feita de forma anônima.
- **timestamps**: *Date*. Armazena as datas de criação e atualização dos registros.

Os relacionamentos entre as coleções *User*, *Subscription*, *Review* e *SoftwareCatalog* são fundamentais para o funcionamento da plataforma. A coleção *Review* faz referência direta ao *software* avaliado e ao usuário que fez a avaliação, enquanto a coleção *SoftwareCatalog* faz referência ao proprietário do *software* (usuário). A modelagem com referências do tipo *ObjectId* permite uma organização modular e escalável dos dados, o que possibilita a expansão da plataforma com novas funcionalidades sem comprometer a performance ou a integridade dos dados existentes.

Além disso, as coleções foram implementadas para suportar diferentes formas de interação entre os usuários e os *softwares* cadastrados na plataforma. O sistema permite o gerenciamento de assinaturas, a avaliação de *softwares*, assim como sua categorização e classificação, o que mantém a flexibilidade para futuras evoluções e novas funcionalidades.

4.1.8 Segurança

A segurança foi uma prioridade durante o desenvolvimento da plataforma. Algumas das práticas implementadas incluem:

- **HTTPS**: O *frontend* foi configurado para funcionar em um ambiente seguro utilizando HTTPS, garantindo que as comunicações entre o cliente e o servidor sejam criptografadas e protegidas contra ataques de interceptação.
- **bcrypt**: Utilizado para a criptografia de senhas dos usuários, garantindo que mesmo em casos de vazamento de dados, as senhas estariam protegidas de ataques de força bruta.
- **JWT**: Os tokens JWT criados são baseados em um *hash* seguro, o que garante a autenticidade e a validade das sessões dos usuários. Esse método de autenticação é altamente seguro e escalável.

Essas medidas garantiram que tanto os dados dos usuários quanto as transações financeiras realizadas na plataforma fossem protegidos contra acessos não autorizados.

4.1.9 Infraestrutura e Deployment

O *deploy* do *frontend* foi realizado na Vercel, uma plataforma que permite a automação do processo de deploy e assegura as boas práticas de CI/CD. Caso os testes não sejam aprovados, o deploy é automaticamente interrompido como medida de segurança, e um aviso é emitido detalhando o motivo da falha.

O *deploy* do *backend* da solução InvestMinds foi realizado no Render, uma plataforma que simplifica o processo de implantação de aplicações *web* e APIs. O Render oferece *builds* automatizados, garantindo a integração e entrega contínuas, além de proporcionar um ambiente escalável e confiável para hospedar os serviços de *backend*.

O *deploy* da documentação foi realizado com HTML e CSS e hospedado no Github Pages, que garante a facilidade de publicação de sites estáticos diretamente de repositórios GitHub, sem a necessidade de servidores adicionais. Ele permite atualizações rápidas e automáticas sempre que mudanças são feitas no repositório, além de ser gratuito e de integrar-se perfeitamente com controle de versão, sendo ideal para hospedar documentações, portfólios e projetos simples, garantindo visibilidade e acessibilidade imediatas.

O *deploy* do banco de dados foi realizado no MongoDB Atlas, uma solução disponibilizada pela própria MondoDB que permite criar e gerenciar o banco de dados pela cloud. A segurança e integridade dos dados é garantida pelo MongoDB Atlas, estando dentro das políticas da LGPD.

4.1.10 Testes e Qualidade de Código

Embora os testes tenham sido realizados principalmente no *backend* utilizando Jest para testes unitários, práticas adicionais de qualidade de código foram implementadas. Cada microsserviço foi armazenado em repositórios Git separados, o que permitiu uma gestão organizada do código e maior controle sobre as versões e o ciclo de vida de cada serviço. Além disso, o Codacy, é uma ferramenta automatizada de revisão de código, foi utilizado para garantir padrões de qualidade de código.

4.1.11 Integração com APIs Externas

A plataforma foi integrada com várias APIs externas para fornecer funcionalidades avançadas aos usuários. As integrações incluem:

- **API do Facebook**: Permitiu que os usuários realizassem *login* na plataforma via Facebook, além de gerenciar páginas e fazer postagens diretamente.
- **Stripe API**: Gerenciou todo o fluxo de pagamento e assinaturas, automatizando a cobrança dos usuários e atualizando seus planos de assinatura na plataforma.
- **OpenAI API**: Forneceu um serviço de *chat* inteligente, permitindo uma interação dinâmica com os usuários.
- **ImgBB API**: Facilitou o *upload* e o armazenamento de imagens para serem utilizadas pelos usuários na plataforma.

Essas integrações permitiram que a plataforma oferecesse uma experiência rica e variada, aproveitando os serviços de terceiros para expandir suas funcionalidades sem a necessidade de desenvolver soluções do zero.

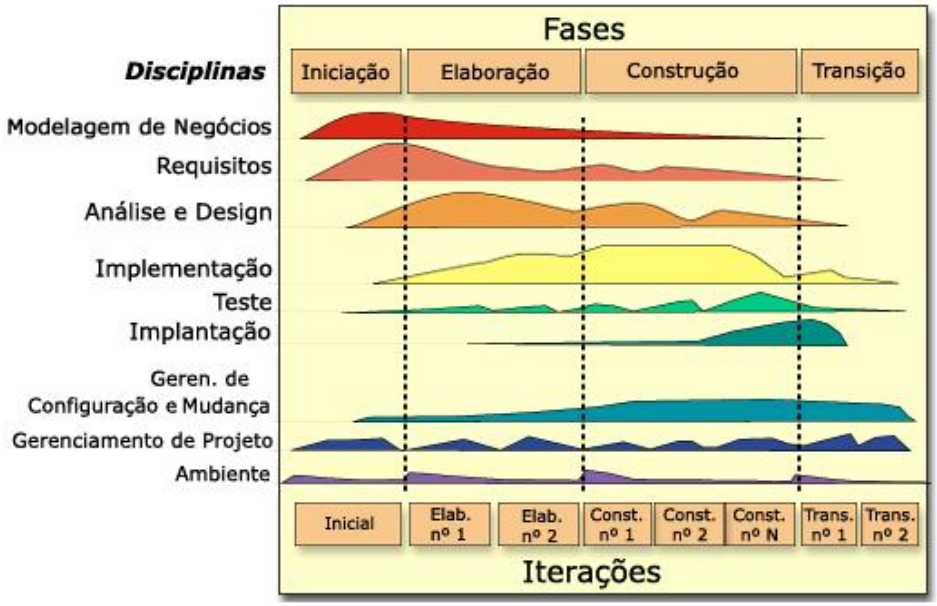
4.2 Gerenciamento

O RUP é um *framework* de desenvolvimento de *software* iterativo e incremental, baseado nas melhores práticas de engenharia de *software*. Ele organiza o trabalho em ciclos de desenvolvimento chamados iterações, onde a cada ciclo são realizadas atividades de análise de requisitos, *design*, implementação e testes. O objetivo do RUP é fornecer um processo estruturado que permita gerenciar a complexidade dos projetos e garantir a qualidade do produto final. A abordagem iterativa permite a adaptação e refinamento do sistema ao longo do tempo, conforme novas informações são descobertas e *feedback* é obtido.

O RUP oferece uma série de ferramentas, diretrizes e templates que auxiliam as equipes no desenvolvimento de *software*. Ele divide o processo de desenvolvimento em

quatro fases principais: Iniciação, Elaboração, Construção e Transição. Cada fase tem um foco específico, como entender os requisitos e mitigar riscos durante a Elaboração ou realizar testes finais na fase de Transição. O RUP não segue uma abordagem rígida, sendo flexível para se adaptar a diferentes tipos de projetos e equipes. A figura 8 demonstra as fases do RUP.

Figura 8 – Fases do RUP.



4.2.1 Documentação

A documentação do InvestMinds foi desenvolvida em HTML e CSS e disponibilizada no GitHub Pages. Entre os documentos criados, estão a explicação do projeto, guias de contribuição e conduta, análise dos riscos envolvidos durante o desenvolvimento, práticas de *DevOps*, qualidade do projeto com foco em testes e análise estática de código, termo de abertura do projeto, arquitetura do InvestMinds, tecnologias utilizadas e arquitetura dos microsserviços.

Os documentos criados são essenciais para garantir uma execução eficiente e controlada oferecendo clareza sobre os objetivos, riscos, processos e tecnologias, além de estabelecer padrões de qualidade e colaboração. Através de uma documentação sólida, é possível mitigar riscos, assegurar a entrega contínua de valor, facilitar a comunicação entre as partes interessadas e promover a escalabilidade e manutenção do projeto, resultando em uma gestão mais organizada e eficiente.

4.3 Objetivos alcançados

O desenvolvimento da plataforma utilizou uma abordagem baseada na arquitetura de microsserviços e na integração com APIs externas para criar uma solução moderna e escalável para clubes de assinatura de *softwares*. Com a divisão da aplicação em quatro microsserviços principais (Autenticação, Catálogo de *Softwares*, Pagamento e BFF), foi possível garantir independência entre as partes do sistema, facilitando a manutenção, escalabilidade e a integração de novas funcionalidades. A arquitetura de APIs permitiu a comunicação eficiente entre a plataforma e serviços externos essenciais, como o Facebook, Stripe, OpenAI e ImgBB. Além disso, o uso de práticas de desenvolvimento, como testes automatizados e contêiner Docker, assegurou a qualidade do código e a consistência dos ambientes de execução.

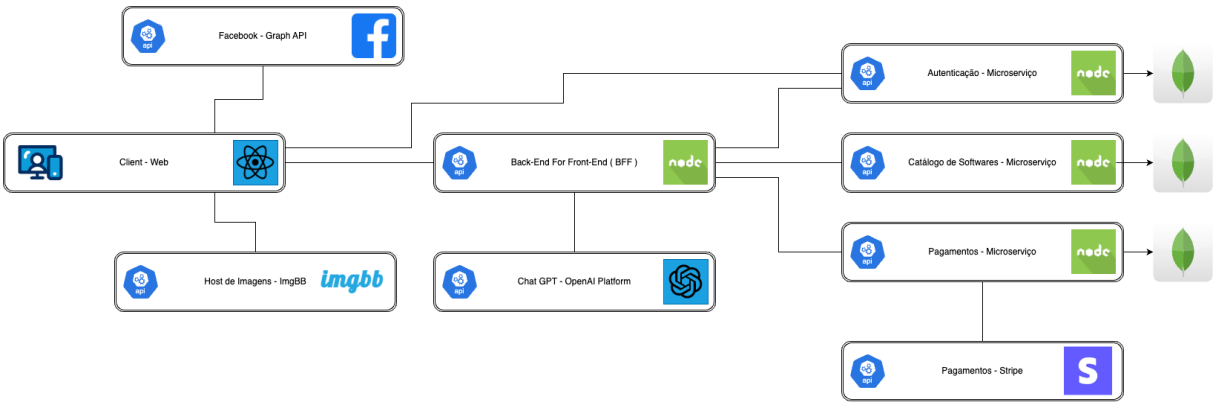
4.3.1 Arquitetura de APIs robusta

A plataforma foi projetada para se comunicar de forma eficiente com APIs externas, permitindo a integração com provedores essenciais para o funcionamento da solução. A API do Facebook foi utilizada para realizar o *login* social dos usuários e permitir a gestão de páginas e postagens. O Stripe foi integrado para gerenciar pagamentos e assinaturas, garantindo uma automação segura e eficiente. O uso da API do OpenAI possibilitou a criação de um *chat* com inteligência artificial, enquanto o ImgBB foi utilizado para o armazenamento de imagens. A escolha de Node.js e Express para o desenvolvimento das APIs internas garantiu alta performance e uma comunicação fluida com esses serviços externos. Essa arquitetura de APIs robusta permitiu que a plataforma fosse facilmente expansível para novos serviços no futuro, mantendo a flexibilidade e modularidade necessárias para a evolução da solução.

4.3.2 Arquitetura de microsserviços eficiente

A arquitetura de microsserviços foi um fator chave para garantir a escalabilidade e a independência entre os componentes da plataforma. Os quatro microsserviços principais — Autenticação, Catálogo de *Softwares*, Pagamento e *Backend for Frontend* — foram desenvolvidos de forma isolada, utilizando Docker para garantir sua execução independente. Essa separação de responsabilidades tornou a plataforma mais resiliente, já que falhas em um serviço não afetariam o funcionamento dos outros. Além disso, a modularidade dos microsserviços facilitou a manutenção do sistema e permitiu o desenvolvimento de novas funcionalidades de forma mais eficiente. Cada microsserviço foi concebido para operar dentro de um contexto bem definido, permitindo escalabilidade e facilidade de ajuste conforme o crescimento da plataforma. A figura 9 demonstra a arquitetura dos microsserviços criada.

Figura 9 – Arquitetura de microsserviços.



4.3.3 Melhores práticas de desenvolvimento

Para assegurar a qualidade do código e facilitar a manutenabilidade, foram adotadas boas práticas de desenvolvimento. O uso de testes unitários com Jest permitiu que as funcionalidades dos microsserviços fossem validadas, minimizando a ocorrência de bugs e erros. Cada microsserviço foi armazenado em repositórios Git separados, o que garantiu maior controle sobre as versões e permitiu que cada serviço fosse trabalhado de forma independente. Além disso, o uso de Docker proporcionou um ambiente consistente, eliminando problemas relacionados à configuração do ambiente e garantindo a confiabilidade das implantações.

4.3.4 Potencial de escalabilidade

A plataforma demonstrou grande potencial de escalabilidade, tanto pela arquitetura de microsserviços quanto pela integração com APIs externas. Cada microsserviço pode ser replicado ou expandido de forma independente, sem a necessidade de reestruturar toda a aplicação. Esse fator é essencial para atender ao crescimento da base de usuários e para a inclusão de novas soluções de software no futuro. A integração com serviços como o Stripe para pagamentos e o Facebook para login também mostrou que a plataforma pode expandir seu escopo sem grandes reestruturações. O modelo proposto garante que novos provedores de serviços possam ser adicionados conforme a demanda sem comprometer o desempenho do sistema.

4.3.5 Benefícios da arquitetura de microsserviços

A arquitetura de microsserviços trouxe diversos benefícios claros para a plataforma. A independência entre os serviços proporcionou maior resiliência, já que falhas isoladas não afetaram o funcionamento completo do sistema. Essa separação também facilitou a manutenção, já que cada serviço pode ser atualizado e modificado sem interferir

nos demais. Outro benefício importante foi a escalabilidade. Cada microsserviço pode ser expandido conforme necessário, o que permite que a plataforma cresça de forma sustentável, mantendo seu desempenho e eficiência. Além disso, a modularidade facilitou a adição de novos recursos e provedores de serviços, garantindo flexibilidade para atender às demandas futuras da plataforma.

Portanto, o uso da arquitetura de microsserviços e a integração com APIs externas possibilitaram a criação de uma plataforma escalável, modular e de fácil manutenção, demonstrando a eficácia desses pilares no desenvolvimento de uma solução robusta para clubes de assinatura.

5 Conclusão

Neste trabalho, foi concluída a segunda etapa do TCC da UnB, no curso de Engenharia de Software. Nessa fase, foram alcançados resultados importantes que demonstram que uma empresa recém-criada deve iniciar um clube de assinatura utilizando a arquitetura de microsserviços e consumo de APIs. Já uma empresa mais consolidada no mercado precisa, primeiramente, realizar adaptações em seus processos antes de adotar essa abordagem. O estudo também evidencia que, embora existam desafios na adoção da arquitetura de microsserviços, o uso de uma plataforma de gerenciamento de APIs (APIM) pode facilitar essa transição, proporcionando benefícios significativos.

O clube de assinatura Investminds, desenvolvido com uma arquitetura baseada em microsserviços e consumo de APIs, oferece robustez, escalabilidade e elasticidade. Ao adotar as melhores práticas de mercado no design desses microsserviços, o Investminds garante uma alta manutenabilidade, facilitando a contribuição de qualquer pessoa interessada em colaborar.

O desenvolvimento da plataforma InvestMinds pode trazer uma série de desafios técnicos e operacionais, principalmente relacionados à gestão de APIs e à segurança em um ambiente de microsserviços. A necessidade de integrar diversas APIs externas, muitas vezes instáveis, pode exigir soluções robustas para garantir a confiabilidade e a performance da plataforma. Além disso, a ampliação da superfície de ataque por meio dessas integrações pode aumentar a complexidade de garantir a segurança dos dados e transações. A comunicação eficiente entre os microsserviços também pode se mostrar como um ponto crítico, dado o risco de latência e falhas de conexão, o que demandou um esforço contínuo para manter a estabilidade e a qualidade do sistema.

A gestão de múltiplas APIs pode se tornar um desafio considerável em uma arquitetura de microsserviços, especialmente em plataformas que integram diversas soluções externas, como clubes de assinatura. À medida que a quantidade de APIs cresce, a manutenção de cada uma delas, o monitoramento de seu desempenho e a garantia de uma integração eficiente tornam-se tarefas complexas. Além disso, a necessidade de gerenciar diferentes versões e garantir que todas estejam funcionando corretamente pode sobrecarregar as equipes de desenvolvimento e operações.

A instabilidade das APIs externas é uma das principais preocupações ao construir plataformas que dependem fortemente de serviços de terceiros. Em sistemas que integram várias APIs externas, como em clubes de assinatura, interrupções em um serviço podem afetar diretamente a experiência do usuário. A dependência dessas APIs também aumenta a complexidade na detecção de falhas e na implementação de soluções temporárias para

mitigar o impacto da indisponibilidade.

A segurança das APIs é um aspecto crítico em arquiteturas baseadas em microsserviços. Como cada microsserviço se comunica com outros serviços por meio de APIs, há um aumento significativo da superfície de ataque. É necessário implementar estratégias robustas de autenticação e criptografia para garantir a proteção de dados sensíveis e a integridade das transações. No entanto, garantir a segurança sem comprometer a performance pode ser desafiador, principalmente em sistemas com alta demanda.

A integração de múltiplos microsserviços e APIs externas envolve uma série de desafios, incluindo a compatibilidade entre diferentes sistemas e protocolos, o gerenciamento de dependências e a garantia de comunicação eficiente entre os serviços. Além disso, é necessário lidar com a complexidade de garantir que todas as APIs integrem-se corretamente ao sistema, sem comprometer o desempenho ou a escalabilidade da plataforma.

Baseando-se nos resultados do estudo, ficou claro que uma plataforma de clube de assinaturas deve adotar uma arquitetura de APIs que facilite a integração com diversos provedores externos. Isso garante não apenas a escalabilidade do sistema, mas também a flexibilidade necessária para acomodar novos serviços e expandir a oferta de soluções de investimento. O uso de uma plataforma de gerenciamento de APIs (APIM) surge como uma ferramenta relevante nesse processo, ajudando a organizar e controlar as conexões de forma eficiente.

A adoção da arquitetura de microsserviços, como demonstrado no desenvolvimento do clube Investminds, proporciona um ambiente robusto, escalável e flexível. Além de permitir que os serviços funcionem de maneira independente, essa abordagem possibilita correções e melhorias sem afetar o funcionamento da plataforma como um todo. No entanto, conforme mostrado no estudo, empresas já estabelecidas devem ter cautela ao migrar para essa arquitetura, ajustando seus processos de forma gradual para evitar interrupções.

A aplicação das melhores práticas de desenvolvimento, como observado no projeto Investminds, é pertinente para garantir a manutenabilidade a longo prazo. Ao seguir padrões de código consistentes e adotar ferramentas modernas de automação e testes, é possível facilitar a contribuição de desenvolvedores e garantir a continuidade do projeto com qualidade. Isso é especialmente relevante para empresas que buscam construir uma base sólida desde o início, evitando retrabalho e problemas futuros.

O estudo reforça a importância de planejar a escalabilidade desde o início, especialmente para empresas em estágio inicial. A arquitetura de microsserviços, aliada a uma infraestrutura de APIs bem projetada, permite que a plataforma seja expandida com novos serviços e soluções de investimento sem comprometer o desempenho. Além disso, empresas que desejam crescer devem considerar o uso de ferramentas que possibilitem essa expansão de forma fluida, tornando o sistema capaz de suportar uma base crescente

de clientes e soluções.

Espera-se que o sistema desenvolvido sirva como base para que empresas iniciantes adotem a abordagem correta desde o início, enquanto empresas em processo de migração possam realizar essa transição de forma cuidadosa, evitando problemas decorrentes de uma mudança precipitada. Embora seja um processo potencialmente longo e custoso, a adoção adequada da arquitetura de microsserviços se mostra necessário para a saúde de uma empresa a longo prazo. No caso de empresas novas, é importante destacar que, ao começar de maneira adequada, não enfrentarão os desafios de migração de arquitetura.

Este trabalho representa uma valiosa contribuição para a comunidade interessada em utilizar a arquitetura de microsserviços e consumo de APIs na criação de plataformas de clubes de assinatura. A transformação proposta comprova que o uso dessa arquitetura será um pilar na criação e manutenção dessas plataformas.

6 Trabalhos Futuros

O futuro da plataforma InvestMinds apresenta oportunidades promissoras para seu crescimento e aprimoramento. A integração com um API Manager trará maior controle e segurança sobre as APIs consumidas, enquanto a adição de novas soluções de investimento permitirá que a plataforma expanda suas funcionalidades de forma modular e flexível. Elementos de gamificação prometem aumentar o engajamento dos usuários, tornando a experiência mais interativa e educativa. Além disso, a integração com APIs de plataformas financeiras oferecerá dados em tempo real, enriquecendo a experiência de investimento dos usuários. A contínua evolução em disciplinas como Engenharia de Produto de Software (EPS) também garantirá que a plataforma adote as melhores práticas de desenvolvimento, facilitando sua escalabilidade e manutenção.

A integração com um API Manager permitirá maior controle e gerenciamento sobre as APIs utilizadas pela plataforma, oferecendo funcionalidades como autenticação, monitoramento de tráfego, controle de acessos e escalabilidade. Com essa evolução, será possível otimizar o consumo de APIs externas e assegurar uma gestão centralizada de todas as integrações, facilitando o monitoramento e a resolução de problemas. Além disso, o uso de um API Manager permite implementar regras de segurança mais rígidas e evitar sobrecargas de tráfego.

Com a adição de novas soluções de investimento de forma independente, a plataforma se tornará mais flexível e escalável, oferecendo diversas opções aos usuários sem sobrecarregar o sistema central. Essas soluções poderão ser integradas de forma modular, o que permitirá que novos serviços sejam adicionados ou removidos sem a necessidade de reestruturar a arquitetura principal, proporcionando mais agilidade para atender às demandas do mercado.

A implementação de gamificação na plataforma pode melhorar a experiência do usuário, incentivando maior engajamento e fidelização. Ao introduzir elementos como recompensas, rankings e desafios, a plataforma pode transformar o processo de investimento em algo mais dinâmico e interativo. A gamificação também contribui para educar os usuários sobre diferentes produtos e estratégias de investimento, criando uma experiência de aprendizado mais imersiva.

A integração com APIs de plataformas financeiras permitirá expandir as funcionalidades da plataforma, oferecendo dados em tempo real sobre o mercado financeiro, como cotações de ações, índices e outros produtos de investimento. Essa integração fornecerá informações valiosas para os usuários e ajudará a criar uma plataforma mais robusta e conectada com o cenário financeiro global, facilitando a tomada de decisões de investi-

mento.

Evoluir em disciplinas como Engenharia de Produto de Software (EPS) permitirá uma aplicação mais sólida das melhores práticas de desenvolvimento, gerando um produto com maior qualidade, escalabilidade e flexibilidade. Ao aprimorar o conhecimento em EPS, será possível melhorar processos de planejamento, execução e manutenção da plataforma, garantindo que o projeto esteja sempre alinhado às demandas do mercado e às necessidades dos usuários.

Referências

ALSHUQAYRAN, N.; ALI, N.; EVANS, R. A systematic mapping study in microservice architecture. In: COMPUTING ENGINEERING AND MATHEMATICS, UNIVERSITY OF BRIGHTON. *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. Brighton, UK, 2016. p. 44–51. Citado 2 vezes nas páginas 28 e 29.

AMAZON. *What is an API (Application Programming Interface)?* 2024. Acesso em: 18 ago. 2024. Disponível em: <https://aws.amazon.com/what-is/api/>. Citado 2 vezes nas páginas 21 e 22.

Amazon Web Services. *O que são microsserviços?* 2024. Acesso em: 04 ago. 2024. Disponível em: <https://aws.amazon.com/pt/microservices/>. Citado na página 19.

AZURE, M. *Azure Functions*. 2020. <https://azure.microsoft.com/en-us/services/functions/>. Citado na página 37.

AZURE, M. *Azure IoT Edge*. 2020. <https://azure.microsoft.com/en-us/services/iot-edge/>. Citado na página 38.

BECK, K. *Manifesto for Agile Software Development*. 2001. Acesso em: 04 ago. 2024. Disponível em: <https://agilemanifesto.org/>. Citado na página 44.

BIRRELL, A. D.; NELSON, B. J. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 2, n. 1, p. 39–59, 1984. Citado 2 vezes nas páginas 22 e 26.

BRAGA, V. R. *Fatores críticos de sucesso para os clubes de assinatura no mercado brasileiro*. Dissertação (Mestrado Profissionalizante) — Universidade de São Paulo, São Paulo, Brasil, 2023. Acesso em: 04 ago. 2024. Disponível em: <https://www.teses.usp.br/teses/disponiveis/12/12142/tde-31082023-202024/>. Citado na página 40.

Cambridge. *Definition of brainstorming*. 2021. Acesso em: 04 ago. 2024. Disponível em: <https://dictionary.cambridge.org/dictionary/english-portuguese/brainstorming>. Citado na página 49.

CATTELL, R. *Scalable SQL and NoSQL Data Stores*. [S.l.]: ACM SIGMOD Record, 2011. Citado na página 34.

CHODOROW, K. *MongoDB: The Definitive Guide*. [S.l.]: O'Reilly Media, 2013. Citado na página 35.

DOCKER. *What is Docker?* 2013. <https://www.docker.com/what-docker>. Citado na página 29.

DRAGONI, N. et al. Microservices: yesterday, today, and tomorrow. In: _____. [S.l.: s.n.], 2017. Citado 2 vezes nas páginas 19 e 38.

FETTE, I.; MELNIKOV, A. The websocket protocol. *IETF*, 2011. Citado 2 vezes nas páginas 23 e 25.

FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. [S.l.]: University of California, Irvine, 2000. Citado 2 vezes nas páginas 23 e 26.

FOWLER, M. *Microservices: a definition of this new architectural term*. [S.l.]: martinfowler.com, 2014. Citado 4 vezes nas páginas 21, 28, 30 e 31.

FOWLER, P. S. *Serverless Architectures on AWS: With examples using AWS Lambda*. [S.l.]: Manning Publications, 2018. Citado 2 vezes nas páginas 25 e 37.

GIL, A. C. *Como elaborar projetos de pesquisa*. 4. ed. São Paulo: Atlas, 2002. Citado na página 43.

GINO, I. *Why Almost Every Company Is Now An API Company*. 2021. Acesso em: 04 ago. 2024. Disponível em: <https://www.forbes.com/councils/forbestechcouncil/2021/09/16/why-almost-every-company-is-now-an-api-company/>. Citado na página 18.

GUBBI, J. et al. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, Elsevier, v. 29, n. 7, p. 1645–1660, 2013. Citado na página 38.

IBM. *What is an API (application programming interface)?* 2024. Acesso em: 04 ago. 2024. Disponível em: <https://www.ibm.com/topics/api>. Citado na página 18.

ISTIO. *What is Istio?* 2018. <https://istio.io/latest/docs/concepts/what-is-istio/>. Citado 2 vezes nas páginas 29 e 37.

KONDOV, A. *Tao of Node: The universal guide to building better Node.js applications (The Tao Programming Books)*. [S.l.]: Self-published, 2023. Citado na página 32.

KONDOV, A. *Tao of React: The universal guide to building better React.js applications (The Tao Programming Books)*. [S.l.]: Self-published, 2023. Citado 2 vezes nas páginas 32 e 33.

KROENKE, D. M.; AUER, D. J. *Database Concepts*. [S.l.]: Pearson, 2013. Citado na página 33.

KUBERNETES. *What is Kubernetes?* 2014. <https://kubernetes.io/docs/what-is-kubernetes/>. Citado na página 29.

Lukasz Plotnicki. *BFF @ SoundCloud*. [S.l.], 2015. Disponível em: <https://www.thoughtworks.com/insights/blog/bff-soundcloud>. Citado na página 31.

MASSé, M. *REST API Design Rulebook*. USA: O'Reilly Media, 2012. ISBN 978-1-449-31195-6. Citado na página 26.

NEWMAN, S. *Backends For Frontends*. 2015. <https://samnewman.io/patterns/architectural/bff/>. Citado na página 31.

NEWMAN, S. *Building Microservices*. [S.l.]: O'Reilly Media, Inc., 2015. Citado 6 vezes nas páginas 21, 22, 26, 28, 30 e 35.

OASIS. *Web Services Security: SOAP Message Security 1.1 (WS-Security 2004)*. 2006. <https://docs.oasis-open.org/wss-m/wss/v1.1.1/os/wss-SOAPMessageSecurity-v1.1.1-os.html>. Citado na página 25.

OFOEDA J. BOATENG, R.; EFFAH, J. Application programming interface (api) research: A review of the past to inform the future. *International Journal of Enterprise Information Systems*, IGI Global, v. 15, n. 3, p. 76–89, 2019. Citado 3 vezes nas páginas 17, 23 e 24.

OLIVEIRA, R. *PRINCE2: A Técnica de Priorização MoSCoW*. 2014. Citado na página 49.

POKORNY, J. *NoSQL Databases: A Step to Database Scalability in Web Environment*. [S.l.]: International Journal of Web Information Systems, 2011. Citado na página 34.

PRODANOV, C. C.; FREITAS, E. C. D. *Metodologia do trabalho científico*: métodos e técnicas da pesquisa e do trabalho acadêmico-2ª edição. 2. ed. Novo Hamburgo, RS: Editora Feevale, 2013. Citado 2 vezes nas páginas 43 e 44.

Red Hat. *What is an API?* 2022. Acesso em: 04 ago. 2024. Disponível em: <https://www.redhat.com/en/topics/api>. Citado na página 18.

RESILIENCE4J. *Circuit Breaker*. 2018. <https://resilience4j.readme.io/docs/circuitbreaker>. Citado na página 30.

RICHARDSON, C. *Microservices Patterns: With examples in Java*. [S.l.]: Manning Publications, 2016. Citado 3 vezes nas páginas 30, 31 e 37.

RICHARDSON, C. *Microservices Patterns: With examples in Java*. [S.l.]: Manning Publications, 2018. Citado na página 32.

RICHARDSON, L.; AMUNDSEN, M.; RUBY, S. *RESTful Web APIs*. [S.l.]: O'Reilly Media, Inc., 2016. Citado 4 vezes nas páginas 21, 23, 26 e 28.

SEBRAE. *Como montar o seu clube de assinatura*. 2023. Acesso em: 04 ago. 2024. Disponível em: <https://sebrae.com.br/sites/PortalSebrae/artigos/como-montar-o-seu-clube-de-assinatura,c0b025f143db6810VgnVCM1000001b00320aRCRD>. Citado 2 vezes nas páginas 17 e 39.

SERVICES, A. W. *AWS IoT Greengrass*. 2020. <https://aws.amazon.com/greengrass/>. Citado na página 38.

SERVICES, A. W. *AWS Lambda*. 2020. <https://aws.amazon.com/lambda/>. Citado na página 37.

SOMMERVILLE, I. *Engenharia de Software*. 9ª. ed. Brasil: Pearson Education do Brasil, 2011. ISBN 978-85-7936-108-1. Citado 4 vezes nas páginas 45, 46, 48 e 49.

W3C. *SOAP Specifications*. 2007. <https://www.w3.org/TR/soap/>. Citado 2 vezes nas páginas 22 e 25.

WSO2. *API Manager*. 2005. Acesso em: 04 ago. 2024. Disponível em: <https://wso2.com/api-manager/>. Citado na página 37.

Apêndices

APÊNDICE A – Código-fonte

O repositório com os códigos desenvolvidos no decorrer do trabalho pode ser acessado no seguinte *link*: <https://github.com/investminds>.

APÊNDICE B – Telas do Investminds

As figuras apresentadas mostram diversas telas e componentes desenvolvidos para a plataforma de clube de assinaturas InvestMinds. A Figura 10 exibe a tela inicial da aplicação, enquanto a Figura 11 detalha a página Sobre, que contém informações sobre o serviço. A Figura 12 mostra a interface de cadastro de usuários, seguida pela Figura 13, que ilustra a tela de login. As Figuras 14 e 15 destacam as barras de navegação personalizadas para os usuários e proprietários de soluções, respectivamente. Já a Figura 16 apresenta o componente de chatbot integrado à plataforma, enquanto as Figuras 17 e 18 exibem a criação e visualização de publicações. As telas de planos e assinaturas estão representadas nas Figuras 19, 20, e 21. As Figuras 22, 23, e 24 mostram funcionalidades relacionadas à gestão de *softwares*, produtos, e usuários dentro da plataforma.

Figura 10 – Tela Home.

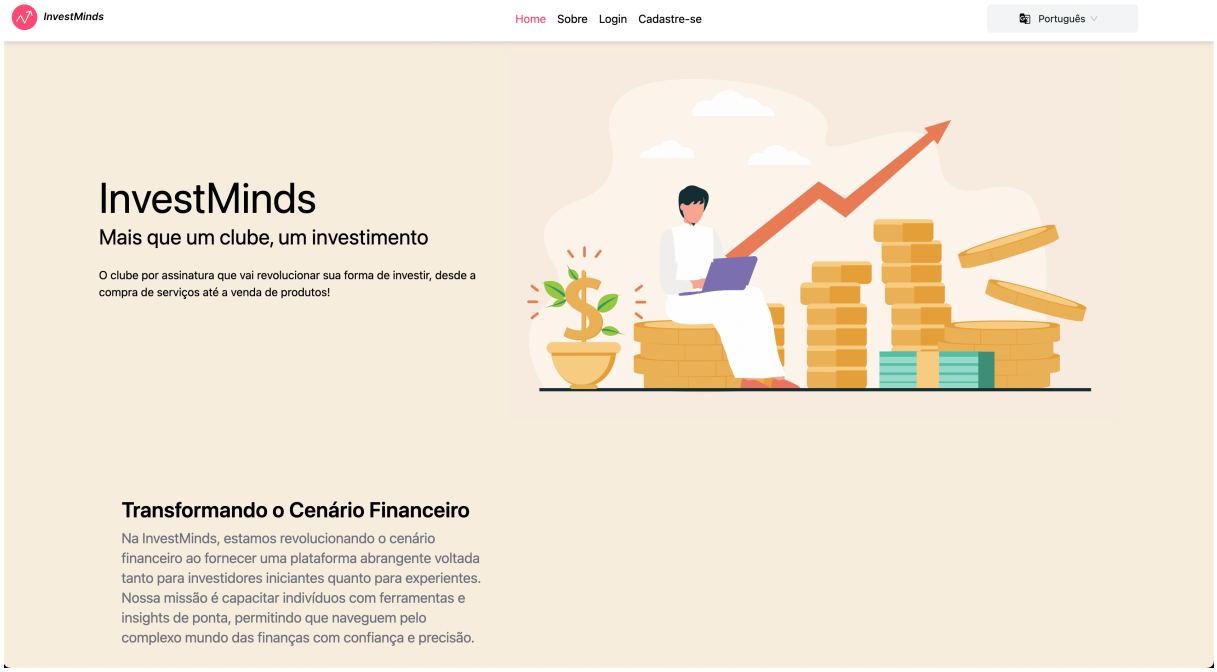


Figura 11 – Tela Sobre.

Figura 12 – Tela Cadastro.

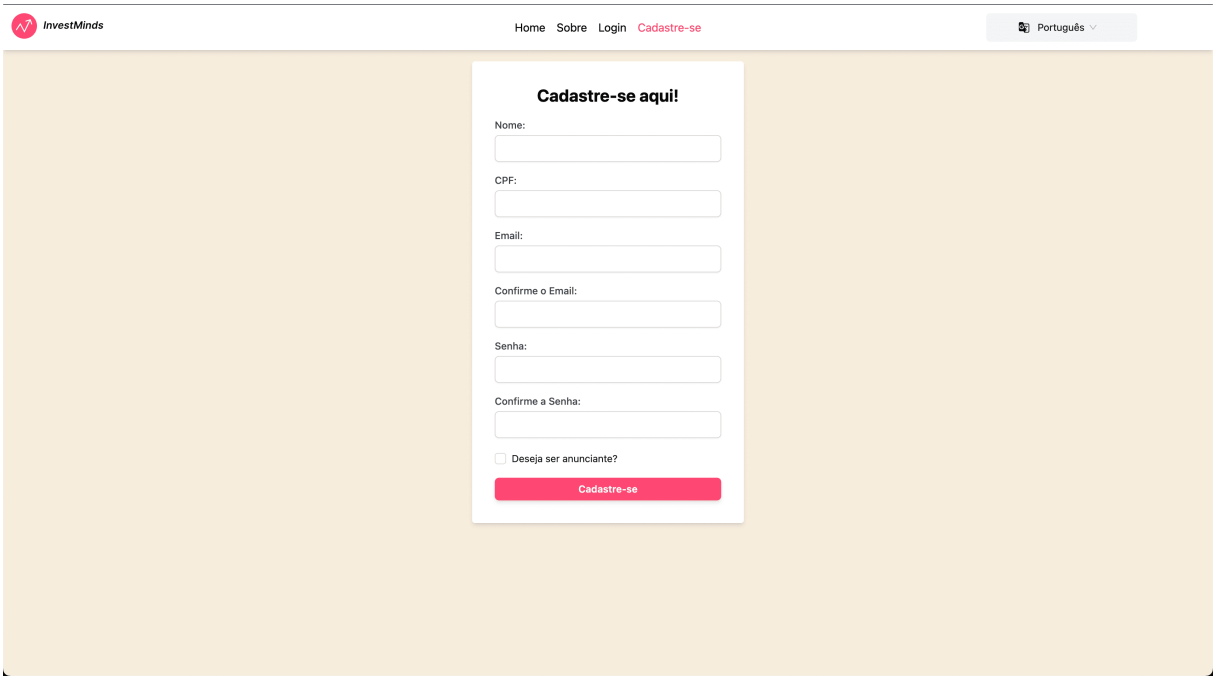


Figura 13 – Tela Login.

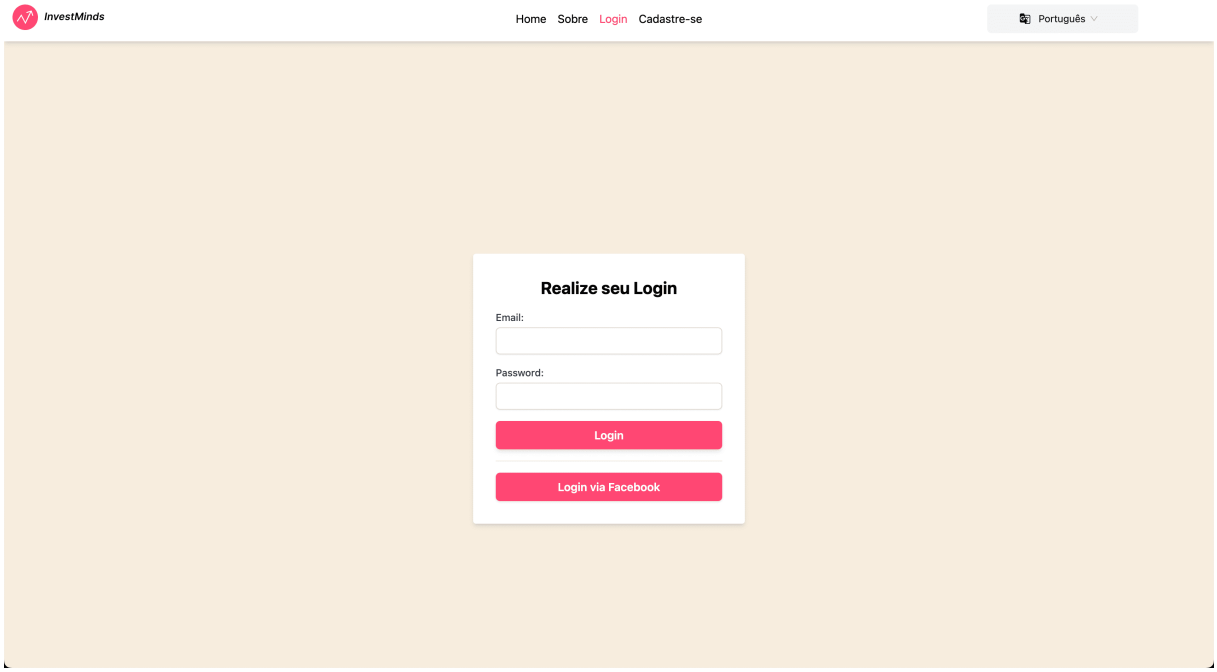


Figura 14 – Navbar usuário.

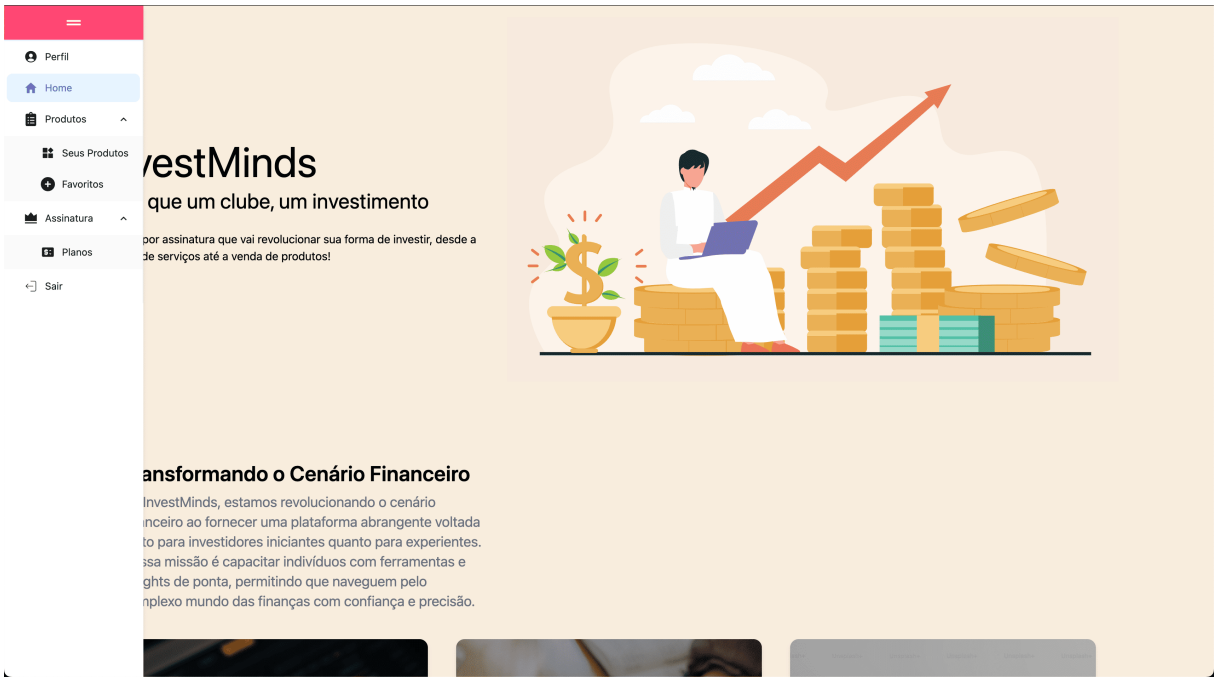


Figura 15 – Navbar dono de solução.

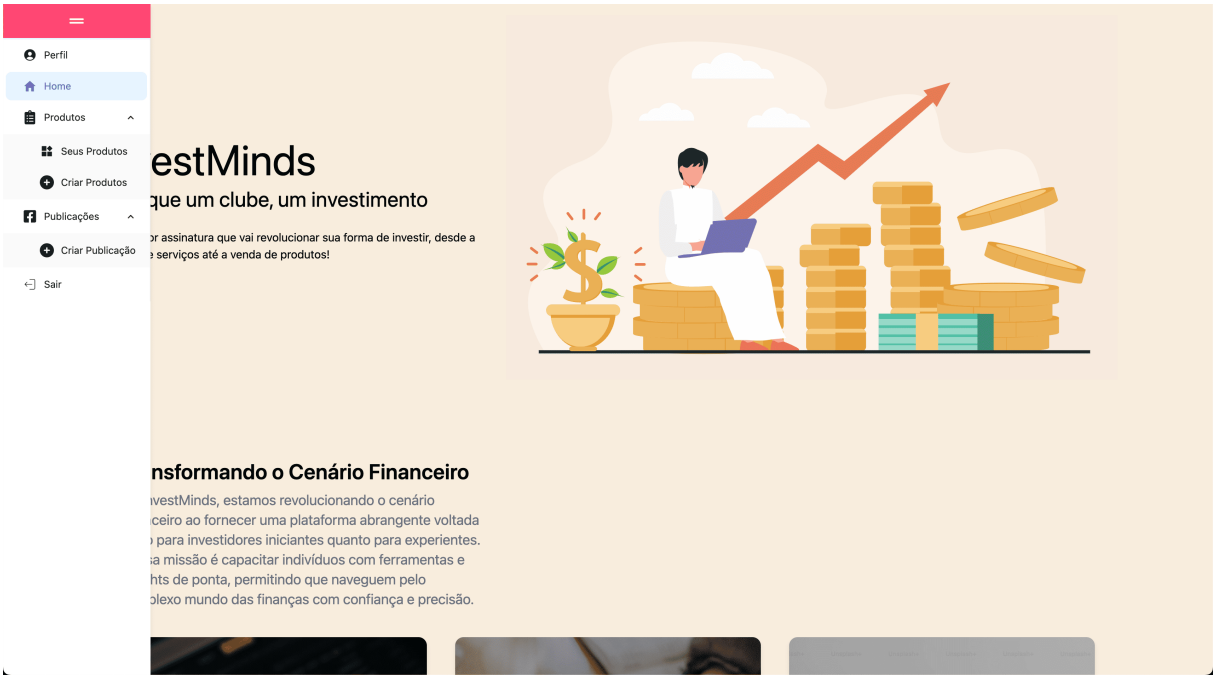


Figura 16 – Component Chatbot.

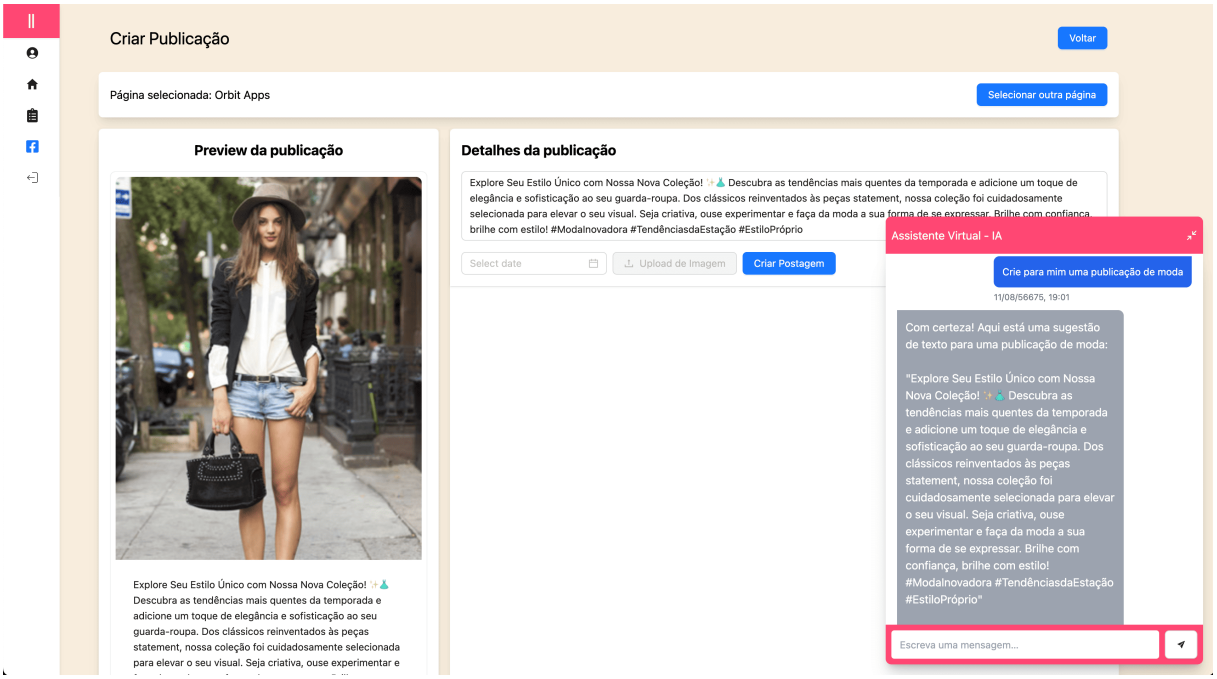


Figura 17 – Tela criar publicação.

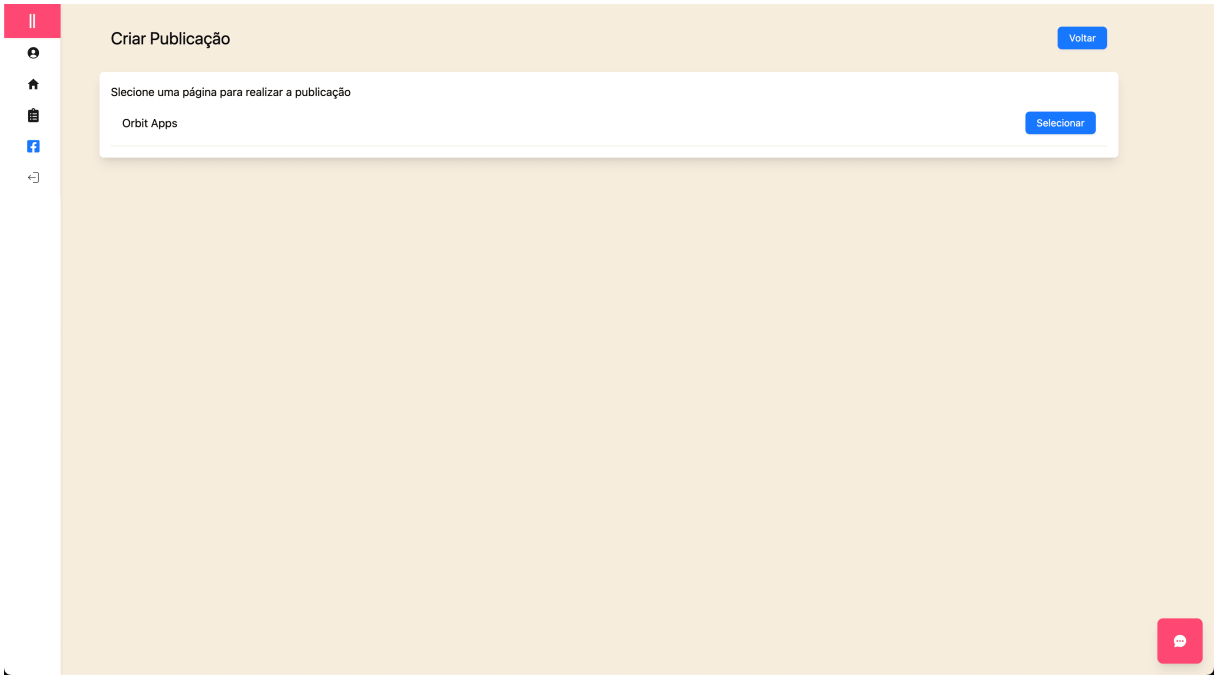


Figura 18 – Preview publicação.

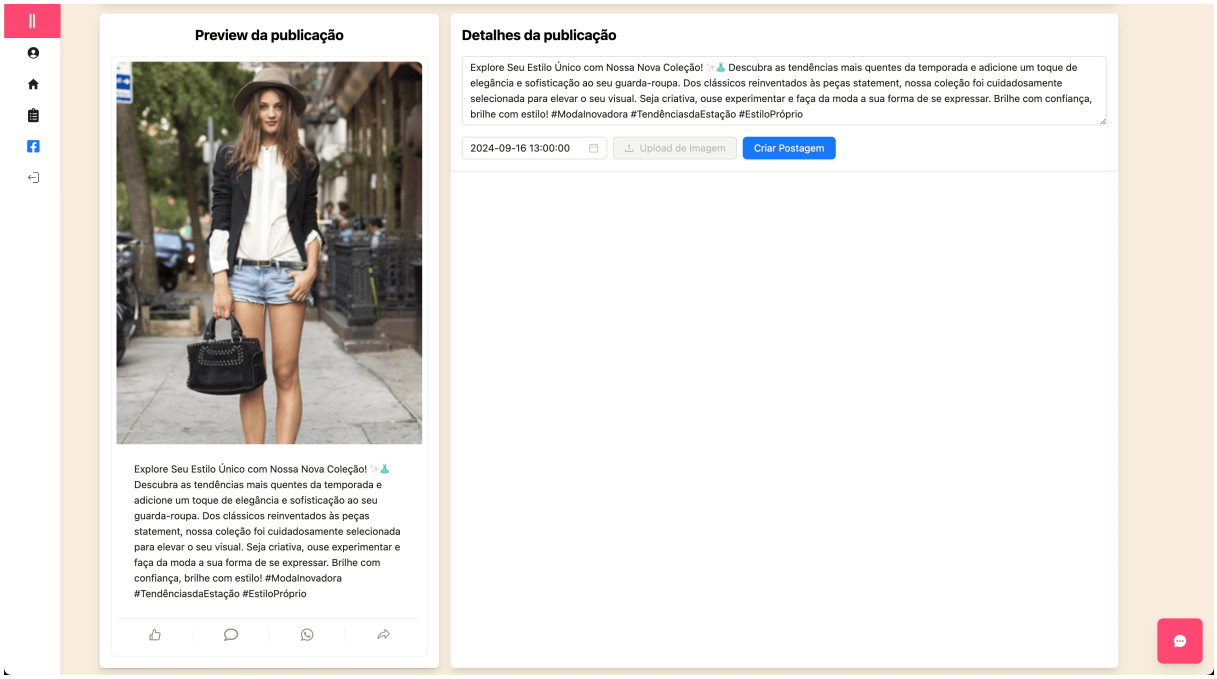


Figura 19 – Planos disponíveis.

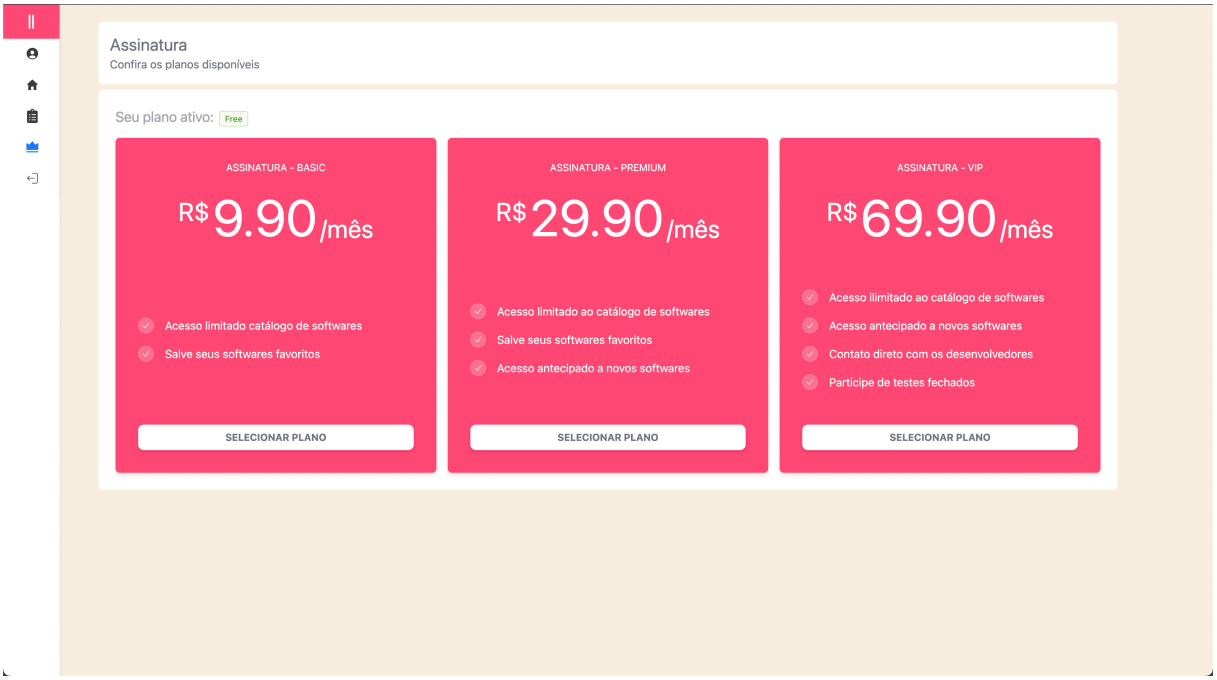


Figura 20 – Assinaturas disponíveis.

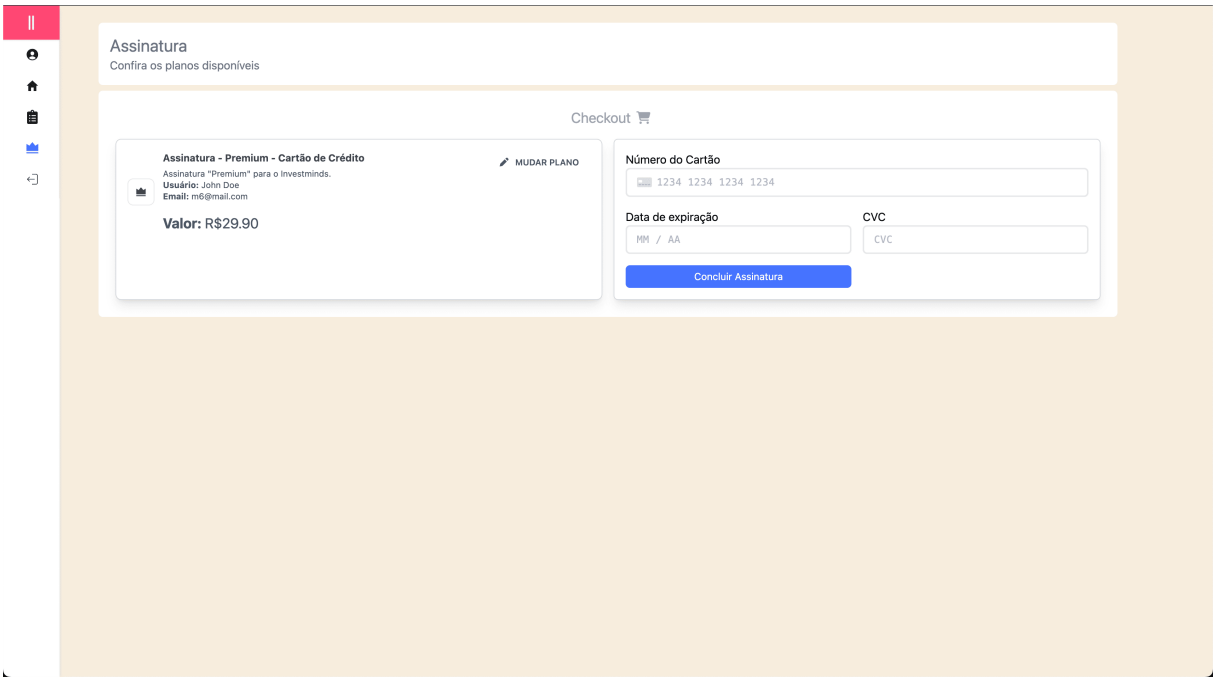


Figura 21 – Assinatura desenvolvida.

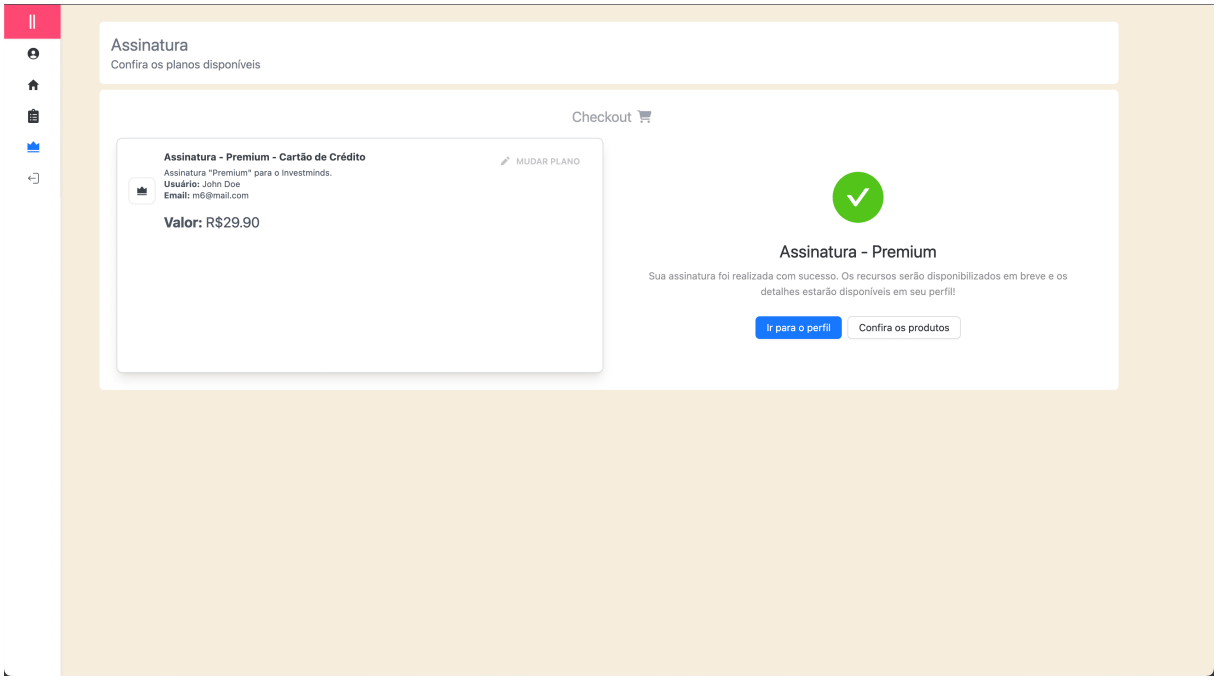


Figura 22 – Lista de softwares.

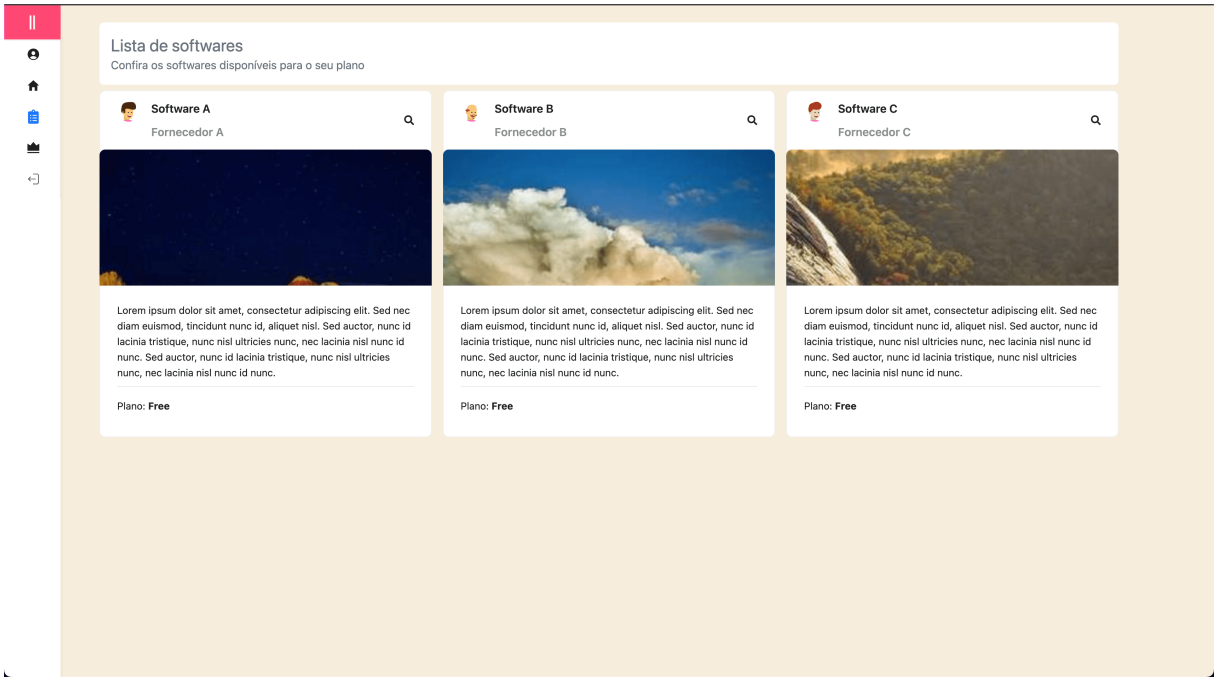


Figura 23 – Cadastro de produtos.

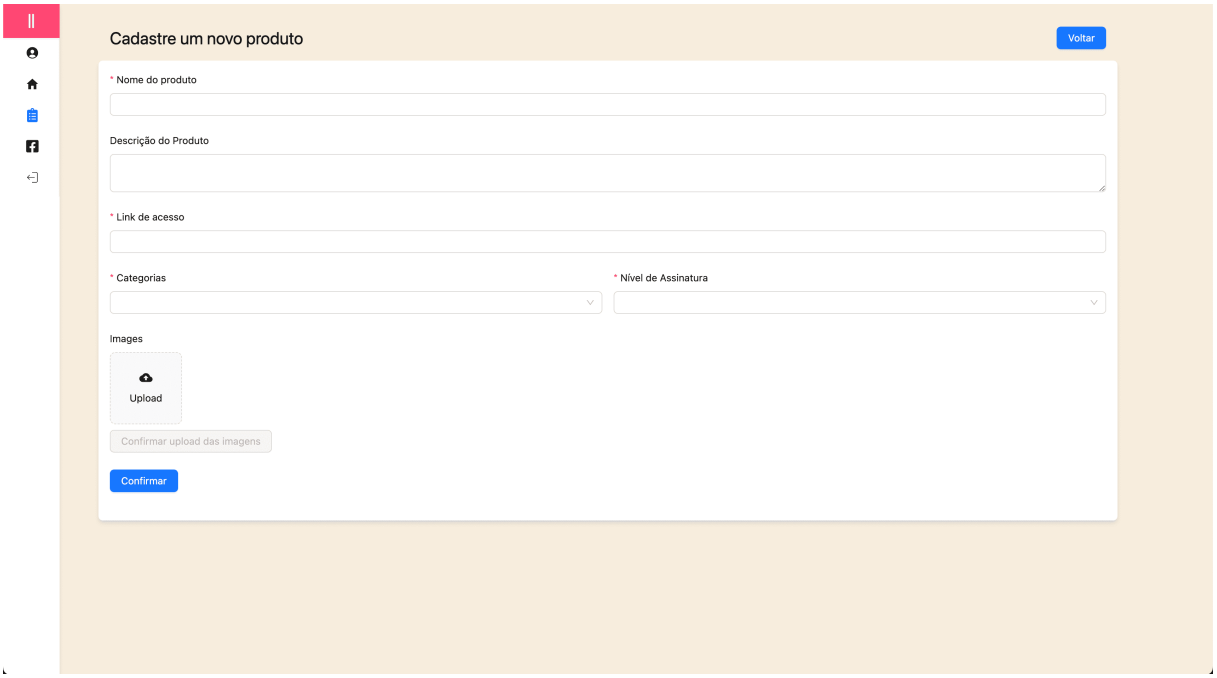
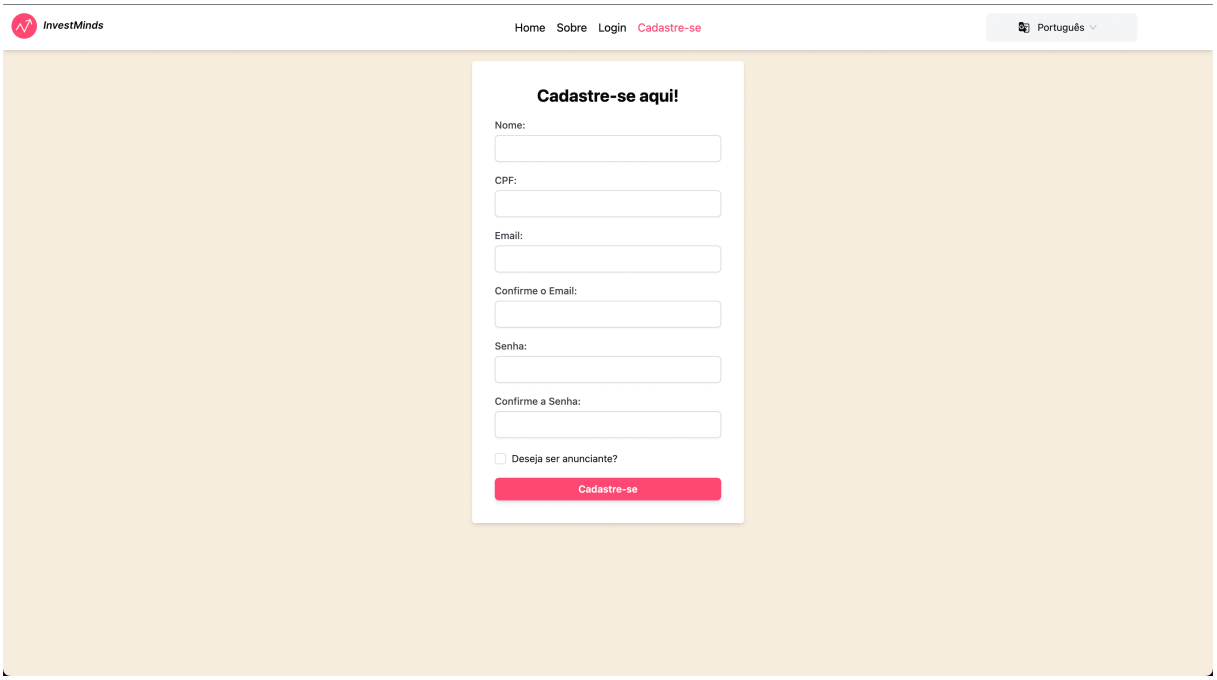


Figura 24 – Cadastro de usuários.



APÊNDICE C – Banco de dados

A Figura 25 apresenta a modelagem do banco de dados da aplicação InvestMinds. Nela, são representadas as entidades e seus relacionamentos, detalhando a estrutura que suporta o armazenamento e a organização das informações essenciais para o funcionamento da plataforma. Essa modelagem inclui tabelas relacionadas a usuários, produtos, planos de assinatura, publicações e outras funcionalidades chave, garantindo a integridade e a escalabilidade da aplicação.

Figura 25 – Modelagem do Banco de Dados.