Universidade Federal de Ouro Preto
Escola de Minas
CECAU - Colegiado do Curso de
Engenharia de Controle e Automação

Vitor Tavares Gonçalves

Introdução à Criação de APIs: Conceitos Fundamentais e Implementação em Linguagem Python

Ouro Preto, 2025

Vitor Tavares Gonçalves

Introdução à Criação de APIs: Conceitos Fundamentais e Implementação em Linguagem Python

Trabalho apresentado ao Colegiado do Curso de Engenharia de Controle e Automação da Universidade Federal de Ouro Preto como parte dos requisitos para a obtenção do Grau de Engenheiro(a) de Controle e Automação.

Orientador: Me. João Carlos Vilela de Castro.

Ouro Preto
2025

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
ESCOLA DE MINAS
DEPARTAMENTO DE ENGENHARIA CONTROLE E AUTOMACAO

FOLHA DE APROVAÇÃO

Vitor Tavares Gonçalves

Introdução à Criação de APIs: Conceitos Fundamentais e Implementação em Linguagem Python

Monografia apresentada ao Curso de Engenharia de Controle e Automação da Universidade Federal de Ouro Preto como requisito parcial para obtenção do título de Engenheiro de Controle e Automação

Aprovada em 05 de setembro de 2025

Membros da banca

Me. João Carlos Vilela de Castro - Orientador (Universidade Federal de Outro Preto)
Dr. Pedro Henrique Lopes Silva - Convidado (Universidade Federal de Outro Preto)
Dr. Caio Fernando Teixeira Portela - Convidado (Universidade Federal de Outro Preto)

João Carlos Vilela de Castro, orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 30/09/2025

Documento assinado eletronicamente por **Joao Carlos Vilela de Castro**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 30/09/2025, às 16:27, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0974102** e o código CRC **C89C2182**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: 3135591533 - www.ufop.br

Agradecimentos

Gostaria de expressar minha mais profunda gratidão a todas as pessoas e instituições que tiveram um papel significativo na realização deste trabalho de conclusão de curso, em especial à UFOP, à Fundação Gorceix e à QI Tech. Essas instituições foram fundamentais para o meu ensino, oferecendo apoio, incentivo e oportunidades que tornaram este projeto possível.

À Universidade Federal de Ouro Preto, meu sincero agradecimento por proporcionar um ambiente acadêmico enriquecedor e por oferecer um ensino gratuito de excelência que me permitiu ampliar meus horizontes na área de Engenharia de Controle e Automação. Também gostaria de estender meus agradecimentos aos professores que me auxiliaram nessa jornada: João Carlos Vilela de Castro, Danny Augusto, Adrielle e Alan Kardek, que contribuíram de forma significativa para minha formação acadêmica.

À minha família e amigos, especialmente ao meu pai Roosevelt Jobert Gonçalves, à minha mãe Micheline Mercia Tavares Queiroz, à minha namorada Julia Muniz de Sousa e aos meus irmãos Julia, Joyce, Pedro, Aurora, Liz e Bento, expresso minha profunda gratidão. O apoio emocional e o encorajamento constantes de vocês foram fundamentais para que eu enfrentasse os desafios com determinação e entusiasmo.

Por fim, sou imensamente grato aos moradores da República Aquarius, onde vivi durante minha jornada universitária. A convivência com vocês proporcionou muito aprendizado e momentos memoráveis, jamais vou me esquecer de vocês, porque doutor, eu não me engano, meu coração é aquariano.

A todos que, direta ou indiretamente, contribuíram de alguma forma durante toda a caminhada, muito obrigado.

"É necessário sempre acreditar que o sonho é possível Que o céu é o limite e você, truta, é imbatível Que o tempo ruim vai passar, é só uma fase irmão..."

— Racionais MC's, "A Vida é um Desafio", álbum "Nada Como um Dia Após o Outro Dia", 2002.

Resumo

Este trabalho tem como objetivo fornecer uma base sólida e escalável sobre APIs (*Application Programming Interfaces*), explorando conceitos fundamentais, tecnologias associadas e práticas de desenvolvimento. A implementação prática é realizada em Python, utilizando o *framework* Falcon, destacando sua simplicidade e eficiência na construção de APIs modernas. Além disso, são abordados aspectos de integração entre APIs, versionamento de código com Git, uso de serviços em nuvem como a AWS e boas práticas de segurança e documentação. Como resultado, este projeto busca não apenas consolidar o entendimento sobre o ciclo de vida de uma API, mas também servir como referência para futuros projetos de desenvolvimento e integração de sistemas.

Palavras-chaves: APIs. Python. Falcon. Git. AWS. Rest.

Abstract

This work aims to provide a solid and scalable foundation on APIs (Application Programming Interfaces), exploring fundamental concepts, related technologies, and best development practices. The practical implementation is carried out in Python, using the Falcon *framework*, which stands out for its simplicity and efficiency in building modern APIs. Furthermore, the study addresses integration between APIs, code versioning with Git, the use of cloud services such as AWS, and essential aspects of security and documentation. As a result, this project not only consolidates the understanding of the API lifecycle but also serves as a reference for future development and system integration projects.

Key-words: APIs. Python. Falcon. Git. AWS. Rest.

Lista de ilustrações

Lista de abreviaturas e siglas

API	Application Programming Interface – Interface de Programação de Aplicações
AWS	Amazon Web Services – Serviço web da Amazon
HTTP	Hypertext Transfer Protocol – Protocolo de Transferência de Hipertexto
JSON	JavaScript Object Notation – Notação de Objetos JavaScript
REST	Representational State Transfer – Transferência de Estado Representacional
Terraform	Terraform – Ferramenta de infraestrutura como código (IaC)
GitHub	GitHub – Plataforma de hospedagem de código
SQL	Structured Query Language – Linguagem de Consulta Estruturada
VS Code	Visual Studio Code – Editor de código Visual Studio
IDE	Integrated Development Environment – Ambiente de Desenvolvimento Integrado
TDAH	Transtorno de Déficit de Atenção e Hiperatividade
IoT	Internet of Things – Internet das Coisas
WeatherAPI	API pública para consulta de dados meteorológicos
ORM	Object-Relational Mapping – Mapeamento Objeto-Relacional

Sumário

1 Introdução

A crescente digitalização de processos e a demanda por conectividade entre sistemas impulsionaram a adoção das APIs (*Application Programming Interfaces*, traduzido para Interface de Programação de Aplicações) como elementos essenciais no desenvolvimento de *software*. As APIs permitem a comunicação entre diferentes aplicações, viabilizando a integração eficiente de serviços e a automação de tarefas. No contexto atual, compreender a estrutura e o funcionamento dessas interfaces é fundamental para profissionais da área de tecnologia (MADDEN, 2020).

Este trabalho tem como objetivo apresentar os conceitos fundamentais relacionados às APIs, abordando sua implementação prática com a linguagem Python. Python se destaca por sua sintaxe simples e ampla adoção no mercado, além de contar com bibliotecas robustas que facilitam a construção de APIs eficientes. Além do desenvolvimento propriamente dito, serão exploradas práticas como o uso de padrões REST, arquitetura de APIs, *frameworks*, segurança da informação e estratégias de teste para garantir um projeto robusto e escalável.

Para demonstrar esses conceitos, foi desenvolvido um projeto prático utilizando a *Weather API*, que fornece dados meteorológicos de forma gratuita. Esse projeto serve como um esqueleto de aplicação, mostrando boas práticas de desenvolvimento e uso de tecnologias atuais. A ideia é que, a partir dele, o leitor possa compreender não apenas como uma API funciona, mas também como criar novas soluções a partir dessa base, aplicando o conhecimento em diferentes contextos.

Por fim, o código e a documentação do projeto serão disponibilizados de forma gratuita no GitHub, incentivando a reutilização e a adaptação do conteúdo. Assim, o estudo busca não apenas apresentar os fundamentos teóricos das APIs, mas também oferecer um guia prático que possa ser expandido por outros desenvolvedores, incentivando aplicações como por exemplo uma API que auxilia a comunicação entre um aplicativo móvel e o *backend* no tratamento de pacientes com TDAH como o trabalho de Pavaneli (2023).

1.1 Justificativas e Relevância

O uso de APIs tornou-se indispensável em praticamente todos os setores da tecnologia, permitindo que sistemas troquem informações de maneira eficiente e segura. Empresas utilizam APIs para integrar plataformas internas, conectar-se a serviços de terceiros e fornecer funcionalidades a clientes e parceiros. Dessa forma, a compreensão e o domínio do desenvolvimento de APIs são competências essenciais para engenheiros

de *software* e profissionais de automação. Além disso, à medida que se tornam pontos de integração crítica entre aplicações, também passam a demandar maior atenção em relação à segurança, como discute Madden (2020) em seu trabalho sobre práticas modernas de proteção em interfaces.

Um aspecto fundamental no contexto da automação e da Internet das Coisas é o papel das APIs como mecanismo de integração entre dispositivos, sensores e serviços digitais. Por meio delas, torna-se possível estabelecer comunicação padronizada e eficiente entre diferentes elementos da infraestrutura tecnológica, viabilizando a coleta, o processamento e o compartilhamento de dados em tempo real. Grønbæk (2008) destaca que em 2008 as arquiteturas baseadas em APIs permitiam o desacoplamento entre lógica de serviço e protocolos de rede, ampliando a portabilidade de sistemas. O fato de essa visão ter sido apresentada há mais de uma década evidencia a maturidade do conceito e sua relevância contínua para o desenvolvimento de soluções escaláveis e inteligentes na Internet das Coisas e em ambientes de automação.

No contexto deste trabalho, a relevância também se apoia no crescente uso de APIs para acesso a dados públicos. Pesquisas recentes mostram que o governo brasileiro tem disponibilizado informações em formato de API, o que favorece a interoperabilidade e o reuso por parte da sociedade (SILVA, 2023).

Um exemplo é a API do IBGE, que expõe dados agregados de censos e pesquisas nacionais, permitindo a automação na coleta e análise de informações (SILVA; SILVA, 2024).

Essa possibilidade abre caminho para soluções baseadas em dados, úteis tanto na formulação de políticas públicas quanto na criação de produtos e serviços digitais. Assim, dominar o desenvolvimento e a utilização de APIs não é apenas uma habilidade técnica, mas também uma ferramenta estratégica para transformar informação em inovação, o que reforça a contribuição prática deste estudo.

1.2 Objetivos

O objetivo deste trabalho é fornecer conhecimento prático e teórico para que outros usuários possam criar suas próprias APIs do zero, baseando-se no modelo fornecido. O foco principal é ensinar boas práticas de estruturação, autenticação, integração com APIs externas e persistência de dados, permitindo que os estudantes desenvolvam APIs funcionais, organizadas e seguras a partir de um guia educacional.

1.2.1 Objetivos Específicos

1 Explorar conceitos fundamentais de APIs: Estudar requisições HTTP, métodos RESTful e formato JSON, oferecendo base teórica sólida para a construção de

APIs.

2 Selecionar e integrar tecnologias adequadas: Utilizar Python, Falcon, SQL e GitHub de forma estratégica, demonstrando como essas ferramentas podem ser aplicadas no desenvolvimento de APIs didáticas e funcionais.

3 Criar uma API protótipo: Desenvolver *endpoints* RESTful que consumam dados da WeatherAPI, exemplificando a comunicação entre APIs externas e a manipulação de dados.

4 Garantir automação e integração: Implementar fluxos automatizados de requisição e processamento de dados, criando um modelo replicável de API funcional.

5 Documentação e controle de versões: Registrar todas as etapas do desenvolvimento, decisões de projeto e resultados no GitHub, facilitando o aprendizado e a replicação por outros estudantes.

6 Apresentar resultados práticos: Demonstrar o funcionamento da API, oferecendo aos interessados exemplos concretos de construção, execução e manutenção de uma aplicação web.

1.3 Organização e Estrutura da Execução

O desenvolvimento da API foi estruturado em etapas que abrangem desde a definição do escopo até a implementação e validação. Inicialmente, foram identificadas as tecnologias e conceitos essenciais para o projeto, como Python, Falcon e SQL. Em seguida, a arquitetura da API foi planejada, incluindo comunicação com a WeatherAPI, autenticação de usuários e estrutura interna. A implementação seguiu boas práticas de codificação, utilizando testes para garantir a confiabilidade do sistema. Por fim, todo o código e documentação foram versionados no GitHub, permitindo que outros usuários aprendam, repliquem e aprimorem o modelo apresentado.

2 Revisão Bibliográfica

A revisão bibliográfica deste trabalho tem como objetivo fundamentar a importância do uso de APIs, destacando sua relevância na integração de sistemas e na automação de processos. Além disso, serão explorados os benefícios de tecnologias para construção da API como Python, Falcon e GitHub entre outras.

2.1 Arquiteturas de APIs

No contexto do desenvolvimento de sistemas distribuídos, diferentes estilos arquiteturais foram propostos para a construção de *web services* e APIs. Cada abordagem apresenta vantagens e limitações, e a escolha depende das necessidades do projeto em termos de escalabilidade, flexibilidade, desempenho e segurança.

2.1.1 REST

O *Representational State Transfer* (REST) é um estilo arquitetural definido por Fielding (2000), amplamente utilizado por sua simplicidade e aderência ao protocolo HTTP. Nesse modelo, os recursos são representados por meio de URLs, enquanto métodos HTTP como GET, POST, PUT e DELETE são empregados para manipulação. Sua leveza e compatibilidade com sistemas heterogêneos o tornaram a principal escolha no desenvolvimento de APIs modernas, favorecendo escalabilidade e integração entre diferentes plataformas (FIELDING, 2000).

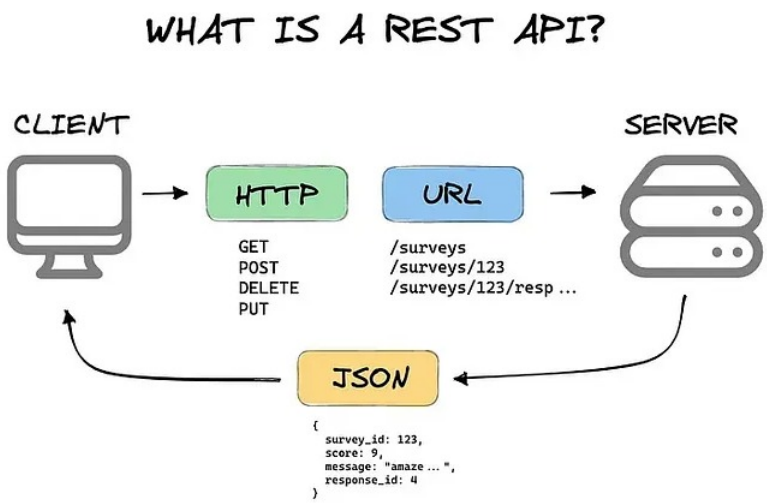


Figura 1 – Exemplo de arquitetura REST API. Fonte: https://medium.com/@shikha.ritu17/rest-api-architecture-6f1c3c99f0d3

2.1.2 SOAP

O *Simple Object Access Protocol* (SOAP) é um protocolo baseado em XML, padronizado pela W3C, que garante forte tipagem, suporte a transações complexas e alto nível de segurança. Apesar de mais rígido que REST, o SOAP é amplamente empregado em domínios que exigem confiabilidade e segurança, como sistemas financeiros e governamentais. Embora mais complexo, o SOAP continua sendo uma escolha relevante em contextos nos quais a padronização e a robustez das transações são fatores determinantes ressaltam (GOMES; JANDL JR, 2009).

2.1.3 GraphQL

O GraphQL, desenvolvido pelo Facebook, surgiu como uma alternativa para a construção de APIs, oferecendo maior flexibilidade em relação a tecnologias tradicionais como REST e SOAP. Essa tecnologia permite que o cliente especifique exatamente quais informações deseja receber em uma única requisição, tornando o processo de comunicação entre cliente e servidor mais eficiente. O GraphQL tem se mostrado especialmente útil em aplicações modernas, incluindo sistemas móveis e aplicações com interfaces ricas, nas quais a agilidade na obtenção de dados é um fator importante (RIBEIRO, 2019).

2.2 APIs e sua Relevância

As APIs (*Application Programming Interfaces*) tornaram-se elementos fundamentais no desenvolvimento de soluções tecnológicas, permitindo a integração eficiente entre sistemas e facilitando a interoperabilidade entre diferentes plataformas. As APIs não apenas otimizam processos internos das organizações, mas também desempenham um papel estratégico ao possibilitar novas formas de monetização e criação de valor. Elas funcionam como uma ponte invisível para o usuário final, conectando serviços e garantindo a comunicação segura e estruturada entre aplicações e o banco de dados, conforme demonstrado na Figura 2 (JUNIOR, 2018).

A adoção de APIs em diversos setores é motivada pela necessidade crescente de escalabilidade e automação. Segundo Sousa, Almeida e Silva e Bandeira (2017), a arquitetura RESTful é amplamente utilizada para estruturar servidores de forma organizada e eficiente, sendo um padrão essencial na implementação de serviços distribuídos. Esse modelo permite que sistemas distintos se comuniquem de maneira padronizada, otimizando a gestão de informações e melhorando a eficiência no desenvolvimento de aplicações.

Além do impacto técnico, a utilização de APIs está diretamente relacionada à segurança e ao gerenciamento de dados, proporcionando maior controle sobre acessos e auditorias. A capacidade de integrar diferentes ativos digitais dentro de uma organização

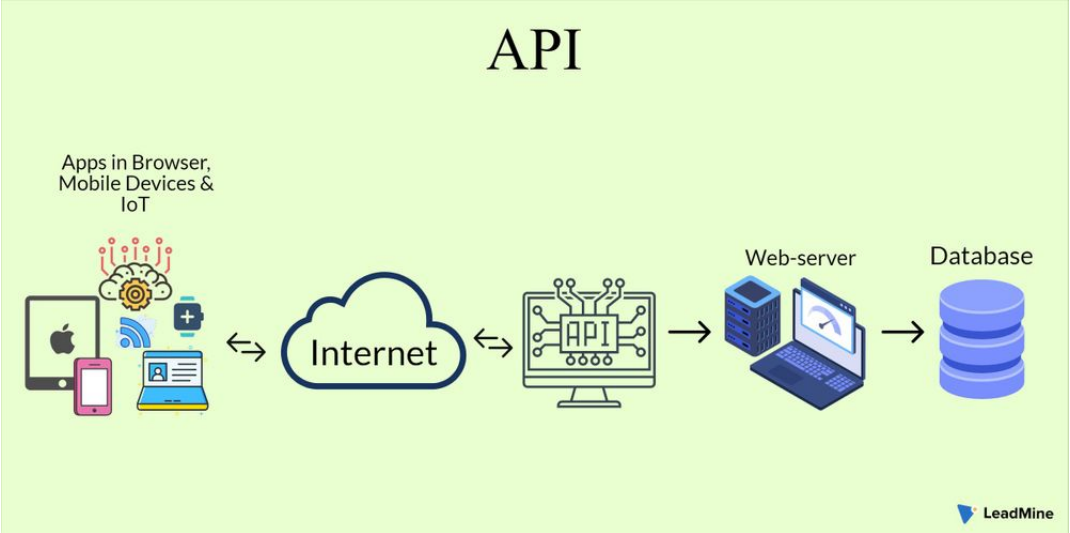


Figura 2 – Exemplo de funcionamento de uma API. Fonte: (LEADMINE, 2023).

amplia a gestão estratégica da informação e melhora a governança tecnológica. Dessa forma, as APIs consolidam-se como ferramentas indispensáveis para a inovação e a modernização dos sistemas computacionais, promovendo a automação, a escalabilidade e a conectividade no ambiente digital (JUNIOR, 2018).

Hoje, a discussão sobre a economia das APIs ganha cada vez mais relevância, pois a integração via API não apenas transforma processos internos das empresas, mas também cria novas oportunidades de negócios e valor estratégico no mercado digital. A utilização eficiente de APIs torna-se, portanto, um diferencial competitivo, reforçando sua importância crescente na economia moderna (JUNIOR, 2018).

2.3 Python como Ferramenta de Desenvolvimento

Python é uma das linguagens de programação mais utilizadas no desenvolvimento de soluções computacionais devido à sua simplicidade e vasta biblioteca de suporte. De acordo com Vasconcelos (2022), sua versatilidade permite aplicações que vão desde a resolução de problemas matemáticos e físicos até a criação de simulações computacionais, tornando-se uma ferramenta essencial para diversas áreas do conhecimento. No campo educacional, estudos como o de Pesente (2019) demonstram que o uso de Python pode auxiliar no ensino de matemática, favorecendo o desenvolvimento do raciocínio lógico e a compreensão de conceitos complexos. Além disso, a importância da linguagem no ensino da lógica de programação e algoritmos também é evidenciada em trabalhos como o de Santos *et al.* (2015), que destacam seu papel na motivação e no aprendizado de estudantes em disciplinas introdutórias de computação.

Essa ampla aplicabilidade faz com que a linguagem Python seja amplamente

utilizado em projetos que envolvem automação, análise de dados e inteligência artificial, consolidando-se como uma linguagem essencial para o desenvolvimento tecnológico em diferentes domínios. No contexto do desenvolvimento de APIs, a linguagem Python se destaca por meio de *framework*s como Falcon e Flask, que oferecem estruturas leves e eficientes para a criação de serviços RESTful. Tais ferramentas permitem a construção de aplicações escaláveis, seguras e de fácil manutenção, reforçando a importância da linguagem como um recurso indispensável tanto para projetos acadêmicos quanto para aplicações em larga escala no mercado tecnológico (FELTRIN, 2023).

2.4 Escolha do *framework* para Desenvolvimento da API

Um *framework* de desenvolvimento pode ser compreendido como uma estrutura de suporte composta por bibliotecas, ferramentas e padrões pré-definidos que facilitam a criação e a manutenção de aplicações de *software*. Seu objetivo é fornecer uma base reutilizável, reduzindo o esforço necessário para implementar funcionalidades comuns e permitindo que o desenvolvedor concentre-se nos aspectos específicos da solução em construção. Dessa forma, o *framework* atua como um alicerce que organiza a arquitetura da aplicação, promove boas práticas de programação e acelera o processo de desenvolvimento.

A escolha do *framework* é um dos pontos centrais no desenvolvimento de uma API, pois ele define não apenas a arquitetura e o alto desempenho da aplicação, mas também a produtividade do desenvolvedor e a facilidade de manutenção do código. No ecossistema Python, diversos *framework*s se consolidaram como soluções populares, cada um atendendo a diferentes demandas e contextos de aplicação (GRINBERG, 2018).

Entre os mais utilizados, destacam-se:

- **Flask**: conhecido por sua simplicidade e flexibilidade, o Flask é um micro*framework* que permite ao desenvolvedor construir aplicações de forma rápida, com grande liberdade de decisão sobre a arquitetura e as bibliotecas auxiliares. É amplamente utilizado em projetos de pequeno e médio porte (GRINBERG, 2018).

- **Django**: *framework* robusto e completo, o Django segue o padrão MVC (Model-View-Controller) e inclui diversas funcionalidades prontas, como autenticação, ORM (Mapeamento Objeto-Relacional) e administração. É indicado para aplicações de grande porte, mas pode ser considerado excessivo para projetos que buscam apenas expor serviços via API (HOLOVATY; KAPLAN-MOSS, 2009).

- **FastAPI**: considerado um dos *framework*s mais modernos para Python, o FastAPI se destaca pelo uso de tipagem estática, integração com OpenAPI e suporte nativo à assincronicidade. Oferece alto desempenho e documentação automática, sendo ampla-

mente utilizado em soluções que exigem rapidez de desenvolvimento e escalabilidade (LUBANOVIC, 2023).

- **Falcon**: projetado especificamente para a criação de APIs REST de alto desempenho, o Falcon adota uma abordagem minimalista e enxuta, privilegiando o desempenho e a utilização eficiente de recursos. É frequentemente escolhido em cenários que demandam baixo tempo de resposta e alta taxa de requisições por segundo (GRIFFITHS *et al.*, 2019).

Diante desse panorama, a escolha pelo Falcon neste trabalho se justifica por algumas razões estratégicas. Por ser um *framework* leve e voltado exclusivamente para o desenvolvimento de APIs, ele se alinha ao objetivo deste projeto de oferecer uma solução prática, clara e eficiente para a introdução ao tema. Além disso, o Falcon é reconhecido por sua escalabilidade e pela capacidade de lidar com alto volume de requisições com baixo consumo de recursos, atributos importantes em cenários reais de produção (GRIFFITHS *et al.*, 2019). Sua simplicidade e objetividade também favorecem o aprendizado, tornando-se uma ferramenta ideal para o caráter didático deste trabalho, que busca não apenas apresentar conceitos fundamentais, mas também incentivar a criação de novos projetos no campo do desenvolvimento de APIs.

2.5 O Papel do GitHub no Desenvolvimento de APIs

O Git, como sistema de versionamento distribuído, é a peça central que permite rastrear mudanças, gerenciar diferentes ramificações de desenvolvimento e possibilitar a colaboração eficiente entre equipes, independentemente da plataforma utilizada. Embora existam várias plataformas que suportam o Git, como GitLab, Bitbucket e Azure DevOps, o GitHub se destacou por criar um ecossistema colaborativo robusto em torno do desenvolvimento de *software*. O GitHub é uma das plataformas mais amplamente utilizadas para hospedagem e controle de versões de código, sendo baseado no sistema de versionamento Git (CHACON; STRAUB, 2014), sua popularidade se deve não apenas ao suporte técnico para versionamento distribuído, mas também à facilidade de colaboração, visualização de histórico e integração com ferramentas modernas (COHEN, 2017).

Do ponto de vista técnico, o GitHub fornece recursos que vão além do simples armazenamento de código. Ele permite integração nativa com pipelines de *Continuous Integration/Continuous Deployment* (CI/CD), bem como de métricas que auxiliam na manutenção de qualidade e segurança do *software* (FOWLER, 2006). Esses recursos tornam a plataforma do GitHub uma peça-chave para a modernização do ciclo de vida de APIs, permitindo que processos de teste, integração e entrega contínua sejam facilmente incorporados ao fluxo de trabalho, conforme ilustrado na Figura 3.

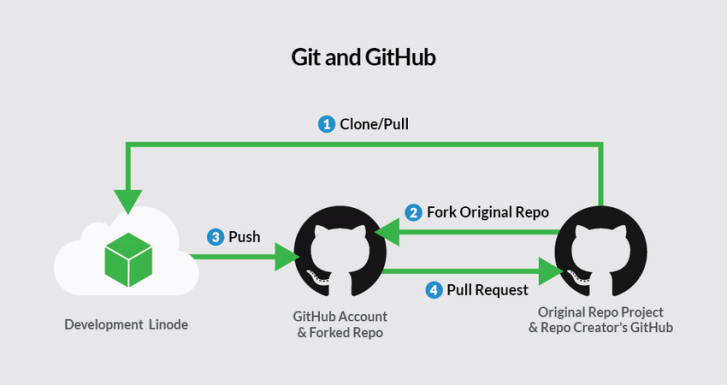


Figura 3 – Fluxos de trabalho do Git utilizando a plataforma GitHub. Fonte: Adaptado de Git with GitHub Workflows

Outro aspecto essencial do GitHub está relacionado à dimensão social do desenvolvimento de *software*. A plataforma introduziu mecanismos como *pull requests*, revisões de código, discussões em *issues* e visibilidade pública das contribuições de cada usuário, o que promove um ambiente de transparência e gerenciamento (DABBISH *et al.*, 2012). Esses mecanismos de interação ajudam a consolidar a reputação dos desenvolvedores e incentivam práticas colaborativas que aumentam a qualidade do código (TSAY; DABBISH; HERBSLEB, 2014). Fatores sociais como confiança e reputação dentro da comunidade exercem influência significativa na aceitação de contribuições em projetos hospedados na plataforma.

Além disso, o GitHub exerce um papel importante na disseminação de boas práticas de engenharia de software. A ampla adoção da plataforma em ambientes acadêmicos, corporativos e em projetos de código aberto contribui para a padronização de práticas de versionamento, documentação e integração contínua. Essa padronização não apenas facilita a entrada de novos desenvolvedores, como também promove maior consistência em projetos distribuídos globalmente.

Portanto, o GitHub deve ser entendido como muito mais do que uma ferramenta de versionamento. Ele se consolidou como uma infraestrutura essencial para a colaboração, inovação e difusão do conhecimento na engenharia de software contemporânea. Sua relevância transcende o aspecto técnico, atuando também como catalisador de comunidades, práticas colaborativas e inovação aberta, fatores que o tornam indispensável para projetos modernos de desenvolvimento de APIs.

2.6 Desenvolvimento de *software*: *frontend* e *backend*

O desenvolvimento de *software*, especialmente em aplicações web, é tradicionalmente dividido em *frontend* e *backend*, permitindo a separação de responsabilidades e facilitando a manutenção, escalabilidade e evolução dos sistemas (RAJABOV, 2023; FLANAGAN, 2020). O

frontend corresponde à parte visível pelo usuário, abrangendo interfaces gráficas, interações e navegação, utilizando tecnologias como HTML, CSS, JavaScript e *frameworks* modernos, como React, Angular e Vue.js. Seu objetivo é proporcionar uma experiência intuitiva, responsiva e acessível, garantindo que os usuários possam interagir de forma natural com o sistema.

O *backend* representa a lógica e o processamento da aplicação, incluindo autenticação, persistência de dados, controle de sessões e integração com sistemas externos (RAJABOV, 2023). Um componente central do *backend* moderno é a implementação de APIs, que funcionam como pontos de acesso a funcionalidades específicas do sistema e permitem a comunicação entre diferentes componentes e aplicações. O padrão REST (*Representational State Transfer*) é amplamente utilizado nesse contexto, definindo um conjunto de princípios arquiteturais para sistemas distribuídos. APIs RESTful utilizam métodos HTTP como GET, POST, PUT, DELETE e PATCH, permitindo operações padronizadas para leitura, criação, atualização e remoção de recursos. Esse modelo promove interoperabilidade, consistência e escalabilidade, sendo uma prática consolidada no desenvolvimento de sistemas web e distribuídos (FIELDING, 2000).

A Figura 4 ilustra a diferença entre Front-End e Back-End, mostrando como cada camada da aplicação possui responsabilidades distintas, mas complementares, no fluxo de dados e processamento.

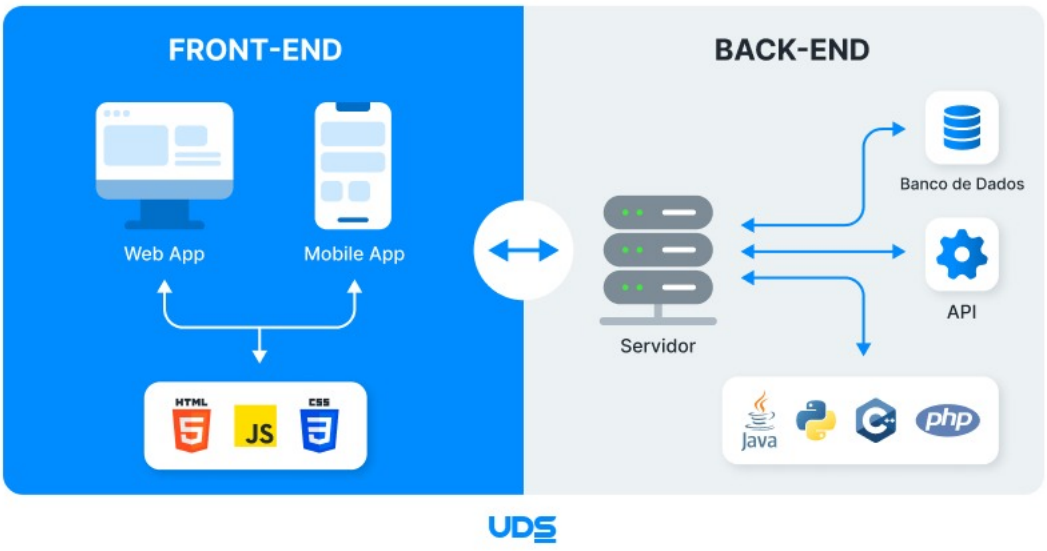


Figura 4 – Comparativo entre Front-End e Back-End. Fonte: Adaptado de https://uds.com.br/blog/front-end-o-que-e-linguagens-frameworks

2.7 SQL, Banco de Dados e ORM com SQLAlchemy

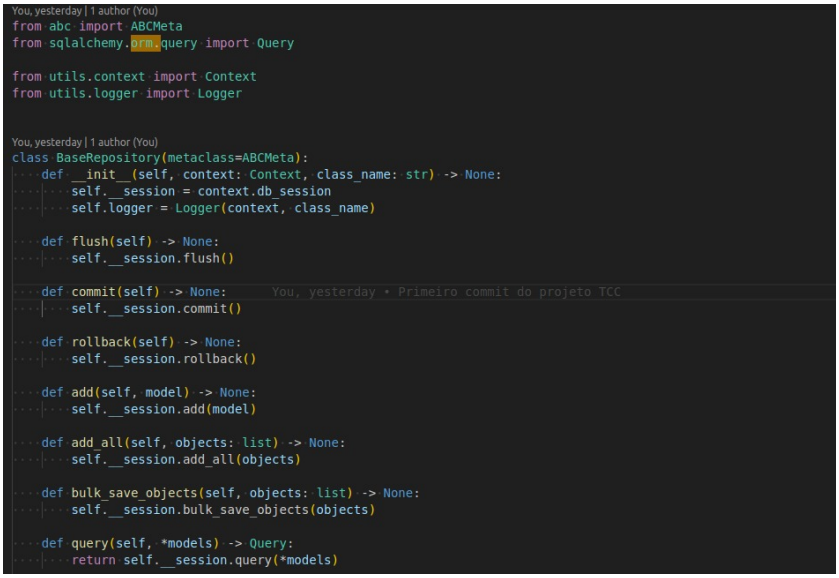
O armazenamento e a recuperação eficiente de dados são elementos fundamentais no desenvolvimento de sistemas modernos. Entre as principais tecnologias utilizadas para esse fim estão os bancos de dados relacionais, que fazem uso da linguagem SQL (*Structured Query Language* traduzido para Linguagem de Consulta Estruturada) para realizar operações como inserção, atualização, remoção e consulta de informações. O SQL consolidou-se desde a década de 1970 como o padrão para manipulação de dados estruturados, sendo reconhecido por sua robustez, simplicidade e ampla adoção em diferentes plataformas e motores de banco de dados (CODD, 1970). Sua padronização pela ANSI e ISO reforça seu papel como a linguagem universal para bancos de dados relacionais, o que garante portabilidade e interoperabilidade em sistemas de diferentes naturezas.

Apesar da robustez do SQL (Linguagem de Consulta Estruturada), o desenvolvimento direto sobre bancos de dados relacionais pode apresentar alguns desafios. Entre eles estão a complexidade do mapeamento entre estruturas de dados tabulares e objetos nas linguagens de programação, a necessidade de escrever grandes quantidades de instruções SQL manualmente e a dificuldade de manter a consistência entre o modelo de domínio da aplicação. Esses fatores podem aumentar a probabilidade de erros, reduzir a produtividade e dificultar a manutenção do sistema.

Para mitigar esses desafios, surgem os *Object Relational Mappers* (ORMs), que permitem que as tabelas de um banco de dados sejam representadas como objetos dentro de linguagens de programação. Essa abordagem proporciona uma camada de abstração entre a aplicação e o banco de dados, tornando o código mais expressivo e próximo ao modelo de domínio do problema. Além de melhorar a legibilidade, os ORMs reduzem a quantidade de instruções SQL escritas manualmente, o que minimiza erros, aumenta a produtividade e favorece a portabilidade entre diferentes sistemas de gerenciamento de bancos de dados. Contudo, é importante destacar que essa abstração pode, em alguns casos, introduzir sobrecarga de desempenho, exigindo um equilíbrio entre produtividade e eficiência (CARMO NICOLETTI; MARÇAL, s.d.).

No ecossistema Python, uma das bibliotecas ORM mais consolidadas é o *SQLAlchemy*. Diferentemente de outros ORMs que priorizam apenas a abstração, o SQLAlchemy adota uma filosofia de oferecer tanto um mapeamento de alto nível, baseado em objetos, quanto a flexibilidade de manipulação direta do SQL quando necessário (BAYER, 2012), o SQLAlchemy é usado para construir uma base de comunicação com o banco de dados. Essa característica permite que desenvolvedores combinem clareza e produtividade com controle refinado sobre consultas mais complexas e de alto desempenho. Além disso, este ORM fornece suporte robusto para gerenciamento de sessões, controle de transações, definição de relacionamentos entre entidades e compatibilidade com múltiplos sistemas de bancos de dados, como PostgreSQL, MySQL, SQLite e Oracle, podemos observar a comunicação

do SQLAlchemy com o banco através da Figura 5.



```python
You, yesterday | 1 author (You)
from abc import ABCMeta
from sqlalchemy.orm.query import Query

from utils.context import Context
from utils.logger import Logger


You, yesterday | 1 author (You)
class BaseRepository(metaclass=ABCMeta):
    def __init__(self, context: Context, class_name: str) -> None:
        self.__session = context.db_session
        self.logger = Logger(context, class_name)

    def flush(self) -> None:
        self.__session.flush()

    def commit(self) -> None:      You, yesterday • Primeiro commit do projeto TCC
        self.__session.commit()

    def rollback(self) -> None:
        self.__session.rollback()

    def add(self, model) -> None:
        self.__session.add(model)

    def add_all(self, objects: list) -> None:
        self.__session.add_all(objects)

    def bulk_save_objects(self, objects: list) -> None:
        self.__session.bulk_save_objects(objects)

    def query(self, *models) -> Query:
        return self.__session.query(*models)
```

Figura 5 – Exemplo de comunicação com banco de dados via ORM Fonte: código do autor

Dessa forma, o uso de SQL aliado a ORMs como o SQLAlchemy representa uma combinação eficiente entre robustez, clareza e produtividade no desenvolvimento de aplicações modernas. Essa abordagem garante maior consistência na manipulação de dados, reduz a complexidade do código e facilita a manutenção de APIs que necessitam de operações complexas de persistência e consulta em ambientes reais de produção.

2.8 Docker

Docker é uma plataforma de **conteinerização** que permite empacotar aplicações e suas dependências em *containers* isolados, garantindo execução consistente em qualquer ambiente. Introduzido em 2013, Docker trouxe uma abordagem mais leve que as máquinas virtuais tradicionais, promovendo maior agilidade, portabilidade e simplicidade no desenvolvimento e entrega de *software* (BOETTIGER, 2015).

Um *container* é uma unidade leve e independente que inclui tudo o que uma aplicação precisa para ser executada: código, bibliotecas, dependências e configurações. Diferente de uma máquina virtual, que simula um sistema operacional completo, o container compartilha o núcleo do sistema operacional do *host* — ou seja, do computador ou servidor onde o Docker está instalado — mantendo o isolamento entre processos. Isso garante que a aplicação rode de forma previsível, sem conflitos de versão ou dependência, seja no ambiente de desenvolvimento, em testes ou em produção (GOMES, 2019).

As *imagens* Docker são templates imutáveis utilizados para criar containers. Elas podem ser automatizadas via *Dockerfiles*, permitindo ambientes reproduzíveis e versionáveis. Essa estratégia resolve o problema clássico do "funciona na minha máquina", garantindo

consistência entre desenvolvimento, teste e produção (MERKEL, 2014). Dessa forma, Docker combina a portabilidade dos containers com o controle preciso das dependências, tornando o ciclo de vida do software mais eficiente e previsível.

Entre as principais vantagens do Docker está a portabilidade, pois os containers podem ser executados em qualquer sistema compatível, garantindo ambientes consistentes entre desenvolvimento e produção. O isolamento também é um ponto forte, permitindo que múltiplas aplicações rodem no mesmo servidor sem conflitos (BOETTIGER, 2015).

Além disso, o Docker assegura reprodutibilidade, facilitando testes e integração contínua, e contribui para a escalabilidade e eficiência, já que os containers são leves, iniciam rapidamente e podem ser replicados com facilidade (PAHL, 2015). Por fim, sua integração com ferramentas como o Docker Compose simplifica a configuração e o gerenciamento de ambientes complexos (TURNBULL, 2014).

2.9 Relevância para a Automação

A automação de processos através do uso de APIs tem se tornado cada vez mais relevante no cenário atual de inovação tecnológica, principalmente no contexto industrial e de controle de sistemas. O uso de APIs permite a integração de sistemas de forma ágil, flexível e escalável, essencial para garantir a eficiência de processos em setores que demandam alta disponibilidade e controle, como a Indústria 4.0.

Segundo Bigheti (2020), a integração de tecnologias como a IIoT(*Industrial Internet of Things* traduzido para Internet das Coisas Industrial), sistemas de controle via redes e computação em nuvem são fundamentais para o avanço da automação industrial. A automação colaborativa, impulsionada pelo uso de APIs e microserviços, promove uma arquitetura flexível e distribuída que pode ser facilmente adaptada às necessidades de diferentes sistemas. Essa integração entre serviços de diferentes plataformas, como sistemas embarcados e ambientes de nuvem, é crucial para a criação de soluções inovadoras, permitindo uma comunicação transparente e a automação de processos de forma eficiente.

No contexto de controle e supervisão de sistemas, as APIs também desempenham um papel essencial. De acordo com (SANTOS, 2022), o desenvolvimento de sistemas de controle que utilizam computação em nuvem permite a supervisão remota e o acesso a dados de qualquer lugar, sem a necessidade de *softwares* específicos instalados localmente. Esse avanço torna os sistemas de controle mais acessíveis, modulares e expansíveis, além de permitir a manipulação e armazenamento de dados de maneira eficiente, sem restrições de tempo real.

A tendência é que o uso de APIs continue a crescer, especialmente com o aumento da adoção de tecnologias como a computação em nuvem e a IoT, que são impulsionadas

pela necessidade de integração e automação em tempo real. As APIs, ao promoverem a interoperabilidade entre diferentes sistemas e serviços, possibilitam a criação de soluções mais inteligentes e automatizadas, transformando a maneira como as indústrias operam e se adaptam às mudanças do mercado, como por exemplo Parizotto (2024) que criou um modelo de inteligência artificial para predição dos movimentos no mercado financeiro, para isso utilizou API's para consulta de dados e treino do modelo.

2.10 Definição do Escopo da API

O objetivo central da API é demonstrar o uso real de uma interface que consome dados de outra API externa. Para isso, será utilizada a WeatherAPI, que fornece informações meteorológicas em tempo real, como temperatura, umidade e condições climáticas de diferentes localidades. A integração com esta API será apenas para fins de demonstração, permitindo que os usuários compreendam o fluxo de requisição e resposta entre APIs, bem como a manipulação de dados em formato JSON. Além disso, a prática reforça a relevância das APIs no desenvolvimento moderno de *software*, destacando seu papel na interconexão de sistemas, na reutilização de funcionalidades pré-existentes e na criação de soluções mais ágeis e escaláveis (OFOEDA; BOATENG; EFFAH, 2019).

3 Desenvolvimento

3.1 Metodologia

O desenvolvimento deste projeto seguirá uma metodologia estruturada para garantir organização, clareza e reprodutibilidade. Inicialmente, será definida a estrutura da API, incluindo a separação entre módulos, *endpoints* e banco de dados, de modo a criar um modelo que possa servir como referência para outros estudantes. Em paralelo, será realizado o escopo do projeto, determinando funcionalidades essenciais, fluxos de dados e processos de autenticação. A construção do código é feita utilizando Python e o *framework* Falcon, priorizando boas práticas de programação. O código base está disponibilizado publicamente, permitindo que outros usuários possam utilizar e aprender a criar uma API funcional a partir do modelo desenvolvido (LUTZ, 2013).

- **Python:** Linguagem principal, escolhida por sua clareza, versatilidade e ampla base de bibliotecas.
- **Falcon:** *Framework* leve e de alto desempenho para criação de APIs REST escaláveis.
- **APIs Externas:** Integração com a WeatherAPI para consulta de dados meteorológicos em tempo real.
- **VS Code:** IDE utilizada para desenvolvimento, com suporte a depuração, Git e gerenciamento de ambientes Python.
- **SQL:** Linguagem para manipulação de bancos de dados relacionais e integração com a API.
- **GitHub:** Plataforma de versionamento e colaboração para controle de código e integração contínua.

Adicionalmente, foi utilizado um assistente de inteligência artificial generativa como ferramenta de apoio à redação e organização do texto. Seu uso se concentrou em revisão gramatical, estilística e estruturação de informações complexas, sempre com avaliação e edição pelo autor para manter fidelidade ao conteúdo técnico. A IA teve caráter assistivo, não substituindo o processo intelectual do pesquisador, que manteve responsabilidade sobre decisões metodológicas, análise e interpretação dos resultados (SAMPAIO; SABBATINI; LIMONGI, 2025).

3.2 Arquitetura da API

A API desenvolvida neste projeto segue a arquitetura REST com uma organização modular e escalável, facilitando manutenção, testes e evolução futura. Abaixo é detalhada a função de cada diretório e arquivo dentro da estrutura do projeto.

3.2.1 Database

- **database.sql**: Contém os *scripts* SQL responsáveis pela criação e configuração do banco de dados utilizado pela API. Inclui a definição de tabelas, relacionamentos, índices e dados iniciais que garantem que a aplicação tenha um ambiente consistente desde a primeira execução.
- **Dockerfile**: Arquivo de configuração para construção da imagem Docker do banco de dados, permitindo que ele seja executado isoladamente em um container. Isso garante que o ambiente de banco seja reproduzível e padronizado, independente do sistema operacional do desenvolvedor.

3.2.2 Src (Source)

O diretório `src` contém todo o código-fonte da API e é organizado em módulos lógicos:

- **connectors**: Responsável por gerenciar conexões externas, como bancos de dados ou serviços de terceiros. Facilita a abstração do acesso a recursos externos, permitindo que a lógica da aplicação não dependa diretamente de detalhes de conexão.
- ***controllers***: Contém a lógica principal de controle da aplicação, processando requisições recebidas, chamando serviços ou repositórios, e formatando respostas adequadas para o cliente. Eles funcionam como intermediários entre as rotas da API e a lógica de negócios.
- **dtos (Data Transfer Objects)**: Define os objetos que serão utilizados para transportar dados entre camadas da aplicação. Eles ajudam a garantir que apenas as informações necessárias sejam expostas, aumentando a segurança e a clareza do código.
- **errors**: Centraliza a definição e o tratamento de erros da API, incluindo exceções customizadas e mensagens padronizadas. Essa abordagem facilita a manutenção e proporciona respostas consistentes ao cliente em casos de falhas como exemplo na figura 6.

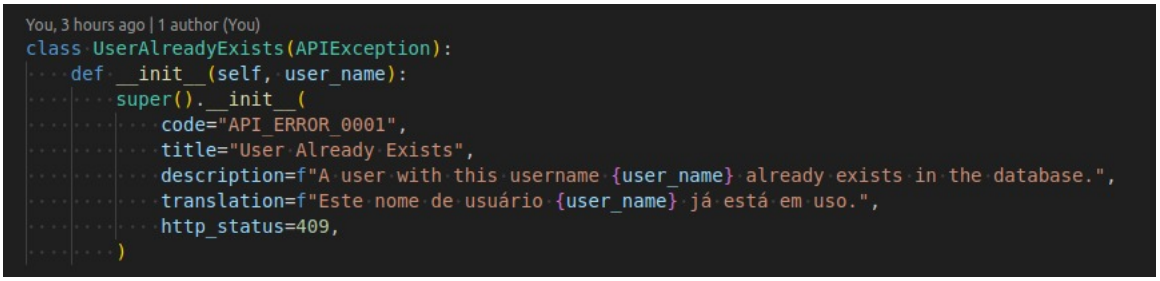


Figura 6 – Modelo de erro usado no base errors.

- **middlewares**: Contém funções que interceptam requisições e respostas, permitindo adicionar funcionalidades como autenticação, autorização, logging ou tratamento global de erros, antes que a requisição atinja os *controllers*.
- ***models***: Define os modelos de dados utilizados pela aplicação, incluindo mapeamentos para o banco de dados. Geralmente utiliza ORMs (Object Relational Mappers) como SQLAlchemy, permitindo abstrair a camada de persistência e trabalhar com objetos Python.
- ***repositories***: Implementa a lógica de acesso aos dados, encapsulando consultas e operações no banco. Essa separação promove o princípio de responsabilidade única e facilita testes unitários.
- **resources**: Contém as rotas da API e suas associações com *controllers*, definindo como os *endpoints* serão expostos e quais métodos HTTP serão suportados.
- ***schemas***: Define a validação e serialização de dados de entrada e saída da API, garantindo que apenas dados corretos e no formato esperado sejam processados, geralmente utilizando bibliotecas como Marshmallow ou Pydantic.
- **utils**: Armazena funções utilitárias e helpers que são usadas em diversas partes da aplicação, evitando duplicação de código e melhorando a legibilidade.

3.2.3 Tests

- **integrations**: Contém os testes de integração da API, que verificam o funcionamento correto do sistema como um todo, incluindo interações entre múltiplos módulos, banco de dados e *endpoints* HTTP.
- **utils**: Inclui funções auxiliares para facilitar a escrita de testes, como inicialização do ambiente para teste.

3.2.4 Arquivos de configuração e documentação

- **local.env**: Arquivo de variáveis de ambiente utilizadas durante o desenvolvimento local, como credenciais de banco, chaves de API, portas e URLs de serviços externos. Permite que a configuração sensível não fique hardcoded no código-fonte.
- **README.md**: Contém a documentação principal do projeto, incluindo instruções de instalação, configuração, execução da API e exemplos de uso.
- **requirements.txt**: Lista todas as dependências Python necessárias para execução da API, permitindo que o ambiente seja reproduzido facilmente via `pip install -r requirements.txt`.

3.2.5 Considerações sobre a Estrutura

A organização segue boas práticas de desenvolvimento de APIs RESTful em Python, priorizando modularidade, clareza e manutenibilidade. A separação em camadas *controllers*, *repositories*, *models* e *schemas* facilita testes, integração com bancos de dados, e evolução da aplicação sem comprometer a estabilidade de partes existentes do sistema. Além disso, a utilização de containers *docker* garante portabilidade, isolamento de ambiente e consistência entre desenvolvimento, teste e produção.

3.3 Segurança em APIs

A segurança em APIs é um aspecto crítico no desenvolvimento de aplicações modernas, pois APIs geralmente expõem dados sensíveis e serviços essenciais a múltiplos clientes, incluindo aplicações web, dispositivos móveis e sistemas de terceiros (FOUNDATION, 2023). A ausência de medidas de segurança adequadas pode resultar em acesso não autorizado, vazamento de dados, ataques de injeção, exploração de vulnerabilidades e comprometimento de todo o sistema.

Além dos mecanismos de autenticação e autorização, a validação de entrada por meio de *schemas* contribui significativamente para a segurança. O uso de definições formais, como o JSON *Schema*, assegura que os dados recebidos estejam em conformidade com a estrutura esperada, restringindo tipos, formatos e valores permitidos, neste trabalho ele foi utilizado apenas para limitar a quantidade de caracteres para usuário e senha. Isso garante que a entrada do banco de dados está segura, podemos ver um exemplo na Figura 7 (DROETTBOOM *et al.*, 2015).

Figura 7 – Estrutura de JSON shcema. Fonte: código do autor

3.3.1 Autenticação e Autorização

Para proteger o acesso a recursos sensíveis, a API deve implementar mecanismos robustos de **autenticação** e **autorização**. Autenticação refere-se à verificação da identidade do usuário ou sistema que está realizando a requisição, enquanto autorização determina quais recursos ou operações aquele usuário tem permissão para acessar (MADDEN, 2020).

Na API desenvolvida neste projeto, a autenticação dos usuários é realizada através de um banco de dados que armazena credenciais de forma segura. Para isso, aplicamos técnicas de **hash** com **salt**, garantindo que as senhas não sejam armazenadas em texto simples. O *hash* transforma a senha em um valor fixo, tornando praticamente impossível recuperar a senha original, enquanto o *salt* adiciona um valor aleatório à senha antes do hashing, aumentando a proteção contra ataques de força bruta e *rainbow tables* (SRIRAMYA; KARTHIKA, 2015).

3.3.2 Hash e Salt: Estratégia Implementada

O processo adotado na API é o seguinte:

1. Cada usuário recebe um *salt* único gerado aleatoriamente.
2. A senha fornecida pelo usuário é concatenada com o *salt*.
3. O resultado é processado por uma função de hash segura, como SHA-256 ou bcrypt.
4. O hash resultante e o *salt* são armazenados no banco de dados.

Quando um usuário tenta se autenticar, a senha informada é concatenada com o *salt* armazenado. O valor resultante é comparado com o hash previamente armazenado,

garantindo que apenas senhas corretas sejam aceitas, sem que a senha original precise ser conhecida pelo sistema (SRIRAMYA; KARTHIKA, 2015).

3.3.3 Boas Práticas de Segurança em APIs

Além do uso de hash e salt, a API segue diversas boas práticas recomendadas para proteger dados e serviços:

- **Uso de HTTPS**: Todas as comunicações entre cliente e servidor são criptografadas, prevenindo ataques de interceptação (*man-in-the-middle*) (FOUNDATION, 2023).
- **Rate Limiting**: Limitação do número de requisições por usuário ou IP, prevenindo ataques de negação de serviço (*DoS*).
- **Validação de entrada**: Todas as entradas de usuários são validadas e sanitizadas para evitar injeção de SQL ou comandos maliciosos.
- **Tokens de sessão seguros**: Autenticação baseada em tokens JWT (JSON Web Tokens) que expiram após determinado período, minimizando o risco de sessões comprometidas.
- **Armazenamento seguro de credenciais**: Variáveis sensíveis, como chaves de API e senhas, são armazenadas em arquivos de configuração ou variáveis de ambiente, fora do código-fonte.

3.3.4 Implementação

A estratégia implementada demonstra uma abordagem equilibrada entre segurança e usabilidade. O uso de hash e salt protege as credenciais mesmo em caso de vazamento do banco de dados, enquanto a adoção de boas práticas de segurança em toda a API reduz significativamente a superfície de ataque. Em um cenário de produção, é recomendável combinar essas técnicas com auditorias periódicas de segurança e monitoramento de acessos, garantindo que a API permaneça segura frente a novas vulnerabilidades (MADDEN, 2020).

Neste projeto, o JWT (JSON Web Token) é utilizado como mecanismo de autenticação de usuários. Após a validação das credenciais armazenadas de forma segura com hash e salt a API emite um token assinado, que passa a ser utilizado nas requisições subsequentes. Esse token possui tempo de expiração e substitui a necessidade de sessões tradicionais baseadas em cookies, alinhando-se às práticas modernas de autenticação (FOUNDATION, 2023).

É importante destacar que, embora o JWT seja amplamente usado também juntamente com uma *API Key* para comunicação entre serviços externos Almeida e Canedo

(2022), essa abordagem não foi adotada neste trabalho. O objetivo central foi autenticar usuários finais e garantir a proteção de suas credenciais, e não integrar múltiplos serviços distribuídos. Assim, a escolha por restringir o uso do JWT apenas à autenticação local simplifica a implementação sem comprometer a segurança dos dados dos usuários.

Essa decisão é coerente com o caráter acadêmico e experimental do projeto: prioriza-se a clareza no fluxo de autenticação e o fortalecimento da segurança das credenciais, sem adicionar camadas de complexidade que seriam mais relevantes em cenários de grande escala ou ambientes corporativos, para se aprofundar um pouco mais com relação ao JWT leia o artigo (JONES; BRADLEY; SAKIMURA, 2015).

O sistema foi projetado para oferecer:

- Registro de usuários com hashing de senhas e salt único por usuário.
- Autenticação via requisição inicial, onde usuário e senha são verificados e uma sessão temporária é estabelecida.
- Armazenamento seguro de credenciais da WeatherAPI, isolando-as do acesso direto dos usuários.
- Embora simplificado, esse modelo reflete boas práticas e proporciona uma base segura para futuras evoluções.

3.4 Integração com WeatherAPI

A construção de uma API de previsão do tempo requer serviços externos capazes de fornecer dados meteorológicos atualizados e confiáveis. Esses provedores disponibilizam informações como temperatura, umidade, vento, pressão atmosférica e condições gerais em tempo real, possibilitando que aplicações ofereçam previsões precisas sem a necessidade de infraestrutura própria para coleta e processamento.

Neste projeto, optou-se pela WeatherAPI, plataforma global que disponibiliza condições atuais, previsões horárias e estendidas. Sua escolha foi motivada pela boa documentação, suporte a múltiplos idiomas, planos gratuitos e pagos, além do retorno dos dados em formato JSON, o que facilita a integração em Python e a compatibilidade com diversas bibliotecas (WEATHER API, 2025). O acesso é realizado por meio de chaves de autenticação (*API Keys*), garantindo consumo seguro e controlado dos dados.

Como os dados são fornecidos em inglês e no Sistema Internacional (SI), foi necessário aplicar um processo de tradução e padronização, convertendo campos (`temperature` para *temperatura*, `humidity` para *umidade relativa*, etc.) e ajustando unidades (Kelvin para Celsius, m/s para km/h). Esse procedimento assegura a compreensão dos resultados

em análises acadêmicas e sua adequação ao padrão brasileiro (DA COSTA *et al.*, 2022; PRAÇA, 2015).

Embora alternativas como OpenWeatherMap e WeatherStack também sejam amplamente utilizadas, apresentam limitações em volume de requisições ou custos de acesso em tempo real. A WeatherAPI, por sua vez, mostrou-se mais equilibrada em custo, abrangência e disponibilidade de dados, justificando sua adoção neste trabalho (DA COSTA *et al.*, 2022).

A integração foi realizada via requisições HTTP (método GET), passando parâmetros como localização e chave de autenticação. O *framework* Falcon foi utilizado para estruturar e validar os dados recebidos, seguindo princípios RESTful e garantindo escalabilidade, interoperabilidade e fácil manutenção do sistema. Além disso, foram aplicadas verificações adicionais, como padronização de datas, tratamento de valores ausentes e categorização de condições climáticas (*ensolarado*, *nublado*, *chuva leve*), assegurando consistência e confiabilidade (PRAÇA, 2015).

Portanto, a utilização da WeatherAPI evidencia como a integração de serviços externos é prática consolidada e eficiente, ao mesmo tempo em que reforça a importância de processos de tradução, conversão e validação para garantir que os dados atendam tanto às necessidades técnicas quanto aos objetivos acadêmicos.

3.5 Testes e Validação

A fase de testes e validação da API meteorológica foi conduzida em múltiplas camadas, garantindo o correto funcionamento da aplicação e a confiabilidade e consistência dos dados fornecidos pela WeatherAPI. O objetivo principal foi assegurar que todas as funcionalidades estivessem operando conforme especificado, que os dados retornados fossem precisos e que a API fosse robusta frente a diferentes cenários de uso (PRESSMAN, 2014).

3.5.1 Testes unitários

Os **testes unitários** foram realizados em funções e módulos isolados da aplicação. Essa camada de testes permitiu verificar a correta implementação de funcionalidades individuais, como:

- Autenticação e autorização de usuários;
- Processamento e normalização dos dados recebidos da WeatherAPI;
- Conversão de unidades de medida e tradução dos campos de resposta;
- Validação de parâmetros de entrada das requisições.

Ferramentas como *pytest* e *unittest* foram utilizadas para automatizar esses testes, possibilitando execução rápida e integração com pipelines de integração contínua (CI/CD) (OKKEN, 2022).

3.6 Documentação e Registro dos Resultados

O conjunto da documentação e o registro sistemático dos resultados representa uma etapa essencial no desenvolvimento de sistemas de *software*, especialmente em projetos acadêmicos que envolvem integração com serviços externos, como a WeatherAPI (PRESSMAN, 2014; GITLAB INC., 2023). Este conjunto não apenas orienta o uso da API, mas também asseguram a reprodutibilidade científica, transparência e rastreabilidade das decisões de projeto.

3.6.1 Documentação técnica

A documentação técnica teve como objetivo fornecer instruções claras e detalhadas para desenvolvedores e usuários finais da API. Entre os elementos incluídos:

- **README detalhado:** descrevendo pré-requisitos, instalação, configuração e execução da API, bem como exemplos de chamadas HTTP e interpretação dos resultados, o arquivo se torna uma página no repositório do GitHub mas pode também ser usado para documentação das API's, foi utilizado linguagem Markdown para o mesmo (CONE, 2020).
- **Exemplos de uso:** *scripts* de teste, exemplos de requisições com diferentes parâmetros de localização e formatos de resposta, incluindo a tradução de campos meteorológicos para o português.
- **Especificações técnicas:** diagramas de arquitetura, fluxos de dados e descrição dos *endpoints* da API, facilitando manutenção futura e integração com outros sistemas.
- **Repositório no GitHub:** o código-fonte, documentação e exemplos encontram-se disponíveis em repositório público, assegurando transparência, versionamento e colaboração aberta (GITLAB INC., 2023; TAVARES, 2025).

A padronização da documentação seguiu boas práticas de engenharia de software e princípios de *DevOps*, visando a facilidade de compreensão, manutenção e escalabilidade (PRESSMAN, 2014).

3.6.2 Documentação acadêmica

Paralelamente à documentação técnica, o próprio trabalho acadêmico exerce o papel de documentação científica, registrando e comunicando o desenvolvimento do projeto:

- **Relatório de desenvolvimento:** descreve a fundamentação teórica, escolha de tecnologias, metodologia de integração com a WeatherAPI, testes realizados e resultados obtidos.
- **Registro de decisões de projeto:** apresenta as alternativas avaliadas, justificativas de escolha de ferramentas e estratégias de implementação, permitindo que futuras pesquisas compreendam o raciocínio por trás das escolhas técnicas.
- **Disponibilidade online:** além deste documento, os códigos-fonte, registros e resultados estão vinculados ao repositório GitHub do projeto, promovendo acesso aberto e facilitando a reprodutibilidade.

A integração entre a documentação técnica e a documentação acadêmica aqui apresentada garante transparência e confiabilidade, além de possibilitar que o trabalho sirva como referência para pesquisadores, desenvolvedores e equipes de manutenção, promovendo a continuidade e evolução do projeto (PRESSMAN, 2014; GITLAB INC., 2023).

3.6.3 Boas práticas e padronização

Para assegurar qualidade e consistência na documentação, foram adotadas práticas reconhecidas na literatura e no mercado:

- Uso de linguagem Markdown para documentação legível e versionável;
- Controle de versão com Git, permitindo rastrear alterações e reverter para versões anteriores se necessário;
- Comentários claros no código-fonte e documentação online, explicando funções críticas e transformações de dados;
- Manutenção de um registro centralizado de resultados e métricas, garantindo integridade e acessibilidade das informações.

Essas práticas fortalecem a confiabilidade do projeto, promovem a colaboração entre membros da equipe e asseguram que os resultados possam ser replicados e validados em contextos acadêmicos ou industriais.

4 Resultados

Neste capítulo, são apresentados e analisados os resultados obtidos a partir do desenvolvimento e implementação da API meteorológica. São detalhados a arquitetura da aplicação, os mecanismos de integração com a WeatherAPI e os testes realizados, de forma a demonstrar a robustez, a confiabilidade e o potencial de expansão do sistema. Oferecendo uma visão clara sobre o funcionamento do projeto e evidenciar a aplicabilidade prática das soluções propostas.

4.1 Arquitetura da API

A arquitetura da API foi planejada para ser modular, escalável e segura, adotando o *framework* Falcon como base para o desenvolvimento dos serviços. O ponto de entrada é o arquivo `app.py`, responsável por gerenciar as rotas e inicializar os recursos e *middlewares*, mostrado na Figura 8. Esse planejamento segue os princípios fundamentais da arquitetura REST, conforme estabelecido por Fielding (FIELDING, 2000).

```
app.py > create
You, 28 seconds ago | 1 author (You)
import falcon
from resources import (WeatherResource, HealthCheckResource, UserResource)

from middlewares import (SessionManager, ContextCreator, InputOutputMiddleware, SecureHeaders, PublicValidation)
from errors import APIErrorHandler

def create():
    api = falcon.App(middleware=[
        ContextCreator(), SessionManager(),
        InputOutputMiddleware(), SecureHeaders(),
        PublicValidation()
        ])

    health_check_resource = HealthCheckResource()
    api.add_route("/ufop_api/health_check", health_check_resource)

    user_resource = UserResource()
    api.add_route("/ufop_api/public/create_user", user_resource, suffix="create_user")
    api.add_route("/ufop_api/public/validate_user", user_resource, suffix="validate_user")
      You, last week • Primeiro commit do projeto TCC …
    weather_resource = WeatherResource()
    api.add_route("/ufop_api/public/weather/current", weather_resource, suffix="current")
    api.add_route("/ufop_api/public/weather/forecast", weather_resource, suffix="forecast")

    return api
```

Figura 8 – Estrutura inicial da aplicação. Fonte: código do autor

Um dos aspectos centrais dessa arquitetura é a distinção entre rotas públicas e rotas privadas, permitindo controlar o acesso de forma granular. Rotas públicas são utilizadas para interações básicas e monitoramento, enquanto as privadas exigem autenticação via token, garantindo maior segurança e proteção dos dados. Essa prática está alinhada com

recomendações encontradas em estudos mais recentes sobre o desenvolvimento e a validação de APIs REST em contextos aplicados (LIMA, 2022).

4.1.1 Middlewares

Os *middlewares* atuam como camadas transversais responsáveis por funcionalidades essenciais, tais como:

- **SessionManager**: Criação e gerenciamento de sessões de banco de dados.
- **ContextCreator**: Inicialização de contexto para cada requisição.
- **InputOutputMiddleware**: Registro de logs e serialização dos dados de entrada e saída.
- **SecureHeaders**: Adição de cabeçalhos de segurança às respostas HTTP.
- **PublicValidation**: Validação de tokens de autorização para *endpoints* públicos, como mostrado na Figura 9.

```python
class PublicValidation:
    def process_request(self, req: Request, resp: Response):

        auth_header = req.get_header('Authorization')
        if not auth_header:
            raise HTTPUnauthorized(description="Authorization header missing")

        parts = auth_header.split()
        if parts[0].lower() != 'bearer':
            raise HTTPUnauthorized(description="Authorization header must start with Bearer")
        if len(parts) != 2:
            raise HTTPUnauthorized(description="Malformed authorization header")

        token = parts[1]

        try:
            payload = jwt.decode(token, JWT_SECRET, algorithms=[JWT_ALGORITHM])
        except jwt.ExpiredSignatureError:
            raise HTTPUnauthorized(description="Token expired")
        except jwt.InvalidTokenError:
            raise HTTPUnauthorized(description="Invalid token")
```

Figura 9 – Fluxo simplificado de validação de token no middleware *PublicValidation*.

De acordo com Grinberg, a utilização de *middlewares* em aplicações web contribui diretamente para a modularidade, manutenção e clareza da arquitetura do sistema, permitindo que responsabilidades sejam distribuídas em camadas bem definidas (GRINBERG, 2018).

Complementando, Almeida ressalta o papel dos *middlewares* na segurança de APIs, destacando sua relevância para a implementação de mecanismos como autenticação, autorização e adição de cabeçalhos de proteção às requisições e respostas (ALMEIDA, 2021).

4.1.2 Recursos (Resources)

A API disponibiliza recursos principais que concentram sua lógica de interação:

- **HealthCheckResource**: Verificação de funcionamento e disponibilidade do serviço.
- **UserResource**: Gerenciamento de usuários e autenticação.
- **WeatherResource**: Comunicação com a API externa (WeatherAPI) para fornecimento de dados meteorológicos.

No desenvolvimento com o framework Falcon, cada *resource* é implementado como uma classe que define métodos correspondentes aos verbos HTTP, como `on_get`, `on_post`, `on_put` e `on_delete`. Esses métodos são mapeados para *endpoints* através das rotas registradas na aplicação, podendo incluir sufixos opcionais para parametrização de recursos, como IDs ou filtros. Essa abordagem modularizada permite que cada recurso seja responsável por sua própria lógica de entrada, facilitando a manutenção e a expansão da API (GRIFFITHS *et al.*, 2019).

O retorno das requisições é padronizado por meio de *Data Transfer Objects* (DTOs), que encapsulam os dados a serem enviados ao cliente sem expor diretamente a lógica interna do sistema. O uso de DTOs garante consistência, validação de tipos e a possibilidade de aplicar transformações nos dados antes do envio, promovendo separação de responsabilidades e maior confiabilidade no contrato da API (SABBAG FILHO, 2024). Na Figura 10 é possível observar a chamada do recurso, a utilização do validador de *schema* e o encapsulamento do retorno em DTO.

```
You, last week | 1 author (You)
import falcon
from falcon import Request, Response

from controllers import UserController
from dtos import UserDTO
from utils.schema_handler import SchemaHandler

💡
You, last week | 1 author (You)
class UserResource:      You, last week • Primeiro commit do projet
    @SchemaHandler.validate("incoming_user.json")
    def on_post_create_user(self, req: Request, resp: Response):
        user_controller = UserController(req.context.instance)
        incoming_schema = req.context.instance.media

        user = user_controller.create_user(incoming_schema)

        resp.media = UserDTO.only_obj_key(user)
        resp.status = falcon.code_to_http_status(201)
```

Figura 10 – Exemplo de chamada de recurso. Fonte: código do autor

4.1.3 Controladores e Repositórios

Os controladores concentram a regra de negócio da aplicação, enquanto os repositórios são responsáveis por gerenciar a comunicação com o banco de dados por meio do SQLAlchemy.

Durante o processo de criação de usuários, o controlador valida os dados recebidos, aplica funções de *hash* e *salt* para proteger as credenciais e, em seguida, aciona o repositório para realizar o armazenamento no banco de dados. Caso o usuário já exista, uma resposta padronizada de erro é retornada como mostrado na Figura 11.

Embora a figura represente apenas a atuação do controlador, o papel do repositório é igualmente essencial: ele abstrai as operações de acesso e persistência dos dados, garantindo que informações como registros de usuários sejam corretamente salvas, atualizadas ou recuperadas do banco de dados de forma consistente e segura. Este desacoplamento da lógica de acesso a dados da lógica de negócios segue os princípios do *Repository Pattern*, promovendo uma arquitetura mais modular, organizada e testável (ALI, 2022).

```
You, last week | 1 author (You)
class UserController(BaseController):
    def __init__(self, context: Context) -> None:
        super().__init__(context, __name__)
        self.user_repository = UserRepository(context)

    def create_user(self, user_schema):
        user_name = user_schema["user_name"]
        password = user_schema["password"]

        already_exist_user = self.user_repository.get_by_user_name(user_name)

        if already_exist_user is not None:
            raise UserAlreadyExists(user_name)

        salt = self.__generate_salt()
        password_hash = self.__hash_password(password, salt)
        user = self.user_repository.create(user_name, salt, password_hash)

        self.user_repository.commit()

        return user      You, last week • Primeiro commit do projeto TCC
```

Figura 11 – Exemplo de chamada do controlador. Fonte: código do autor

Esse fluxo garante segurança e rastreabilidade, além de reforçar a separação de responsabilidades entre as camadas da aplicação.

4.2 Integração com WeatherAPI

A integração com a WeatherAPI representa um dos pontos mais relevantes do sistema. Por meio do conector `WeatherAPIConnector`, a API realiza requisições HTTP,

trata as respostas e transforma os dados recebidos em DTOs que podem ser consumidos pelos clientes da aplicação.

O tratamento de respostas incluiu a simplificação de informações, com o objetivo de fornecer dados mais claros e relevantes ao usuário final. Esse processo de curadoria torna o sistema mais acessível para diferentes cenários de uso.

Entre as possibilidades práticas de aplicação dos dados obtidos pela WeatherAPI, destacam-se:

- **Agronegócio:** decisões sobre irrigação e colheita podem ser tomadas com base em informações como umidade, precipitação e temperatura.
- **Logística:** previsão de condições meteorológicas em rotas de transporte auxilia na redução de riscos e otimização de trajetos.
- **Sistemas urbanos:** integração a sistemas de mobilidade urbana ou monitoramento de enchentes pode ajudar a prevenir danos e melhorar a segurança da população.
- **Aplicações pessoais:** aplicativos de previsão do tempo personalizados podem ser desenvolvidos a partir do consumo dos dados.

Esse tipo de integração demonstra como a comunicação entre APIs é um dos pilares fundamentais em arquiteturas modernas, especialmente em ecossistemas baseados em microsserviços, foi realizado um exemplo prático de como funciona essa comunicação entre serviços, para melhor entendimento do padrão utilizado recomendo ler a documentação do WeatherApi (WEATHER API, 2025), exemplo na Figura 12.

4.3 Testes e Validação

A fase de testes teve como objetivo assegurar a confiabilidade da API e validar o funcionamento de ponta a ponta. Os testes foram conduzidos em duas frentes: automatizados, utilizando a biblioteca `pytest`, e manuais, em ambiente local com Docker, envolvendo cadastro de usuários, autenticação e consultas à WeatherAPI.

4.3.1 Testes Automatizados com Pytest

Os testes unitários implementados com `pytest` permitiram validar de forma isolada funções críticas da aplicação, como a geração de *hash* de senhas, autenticação de tokens e manipulação de dados recebidos da WeatherAPI. Cada teste verifica um comportamento específico, garantindo que alterações futuras não quebrem funcionalidades já implementadas.

```
You, 7 days ago | 1 author (You)
class WeatherAPIConnector(RestConnector):
    def __init__(self, context):
        super().__init__(
            context=context,
            class_name="WeatherAPIConnector",
            base_url="http://api.weatherapi.com/v1",
            timeout=5
        )

    def get_current_weather(self, city: str):
        endpoint = "/current.json"
        params = {
            "key": WEATHER_API_KEY,
            "q": city,
            "aqi": "no"
        }
        response = self.send(endpoint=endpoint, method="GET", params=params)
        return response.response_json

    def get_forecast_weather(self, city: str, days: int = 3):      You, 7 days ago • fix
        endpoint = "/forecast.json"
        params = {
            "key": WEATHER_API_KEY,
            "q": city,
            "days": days,
            "aqi": "no",
            "alerts": "no"
        }
        response = self.send(endpoint=endpoint, method="GET", params=params)
        return response.response_json
```

Figura 12 – Exemplo de conector para integração com a WeatherAPI. Fonte: código do autor

O `pytest` se destaca por sua simplicidade, rapidez e escalabilidade, oferecendo recursos como *fixtures* para configuração de cenários de teste, marcação de testes (*markers*) e parametrização de entradas. Essas funcionalidades permitem a execução de múltiplos cenários de forma automatizada, assegurando maior confiabilidade do sistema, facilitando a detecção precoce de falhas e contribuindo para a manutenção e evolução contínua da aplicação (OKKEN, 2022).

4.3.2 Testes Manuais em Ambiente Local

Após os testes automazidaso com pytest, a API foi testada localmente para verificar o correto funcionamento dos *endpoints*, a integração com a WeatherAPI e a persistência de dados no banco SQL. Serão realizados testes unitários e de integração, garantindo a confiabilidade do sistema. Estudos indicam que testes unitários bem elaborados podem servir como exemplos de uso da API, auxiliando desenvolvedores a entenderem funcionalidades complexas e fluxos de dados (NASEHI; MAURER, 2010).

Podemos observar através das figuras 13 e 14 os testes em ambiente local, utilizando Postman para envio de requisições e o DBeaver para consulta de dados, ambos os softwares foram escolhidos pela familiaridade e facilidade de uso. Na primeira imagem temos o envio da requisição para criação de usuário com senha e na segunda podemos observar como foi salvo no banco através do uso de *salt* e *hash* conforme mencionado anteriormente.

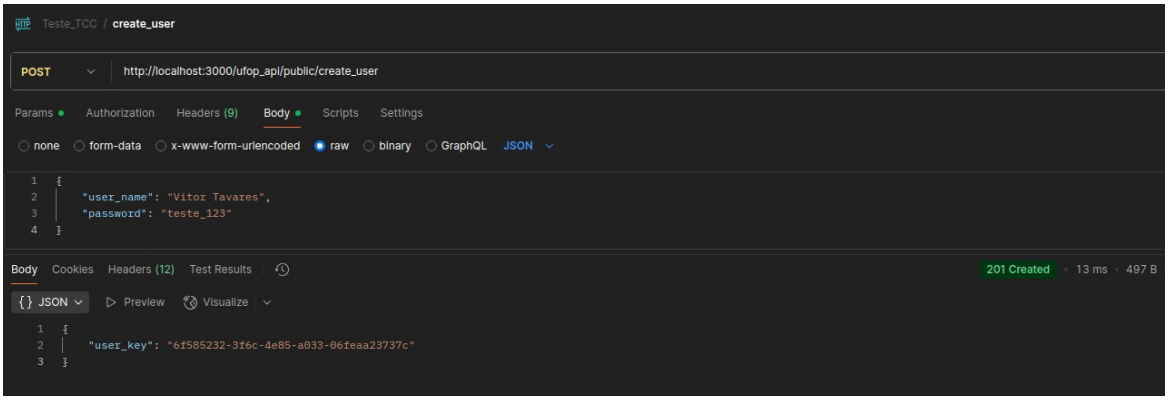


Figura 13 – Envio de requisição para criação de usuário via Postman.

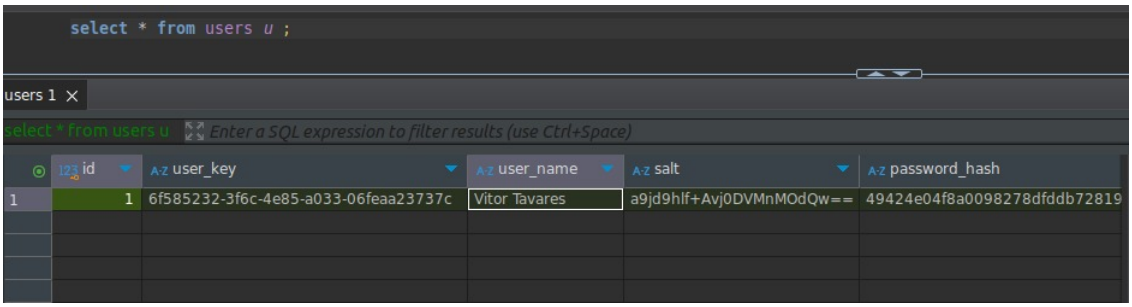


Figura 14 – Registro do usuário no banco de dados PostgreSQL.

Em complemento, testes manuais foram conduzidos para simular cenários de uso real em ambiente Docker. Esse processo incluiu:

- Criação de usuários e login para obtenção de token de acesso.
- Validação da persistência dos dados no banco de dados PostgreSQL.
- Consulta ao *endpoint* de clima, utilizando o token gerado.

O fluxo de teste envolveu três etapas principais para verificação do funcionamento da API:

1. **Requisição via Postman:** envio de um `POST` com credenciais para criação de usuário e autenticação.
2. **Registro no banco de dados:** verificação do cadastro do usuário no Banco de Dados local PostgreSQL.
3. **Consulta autenticada:** utilização do token retornado para acessar de forma autenticada o recurso de previsão meteorológica.

Os resultados mostraram que a API respondeu adequadamente em todos os cenários, apresentando mensagens claras de erro em casos inválidos, onde foi utilizado um base_error.py para configurar o retorno dos erros padrão. A api retorna dados meteorológicos confiáveis quando a autenticação é realizada com sucesso como no exemplo da Figura 15, onde foi inserido o token obtido após validar o usuário para enviar a requisição.

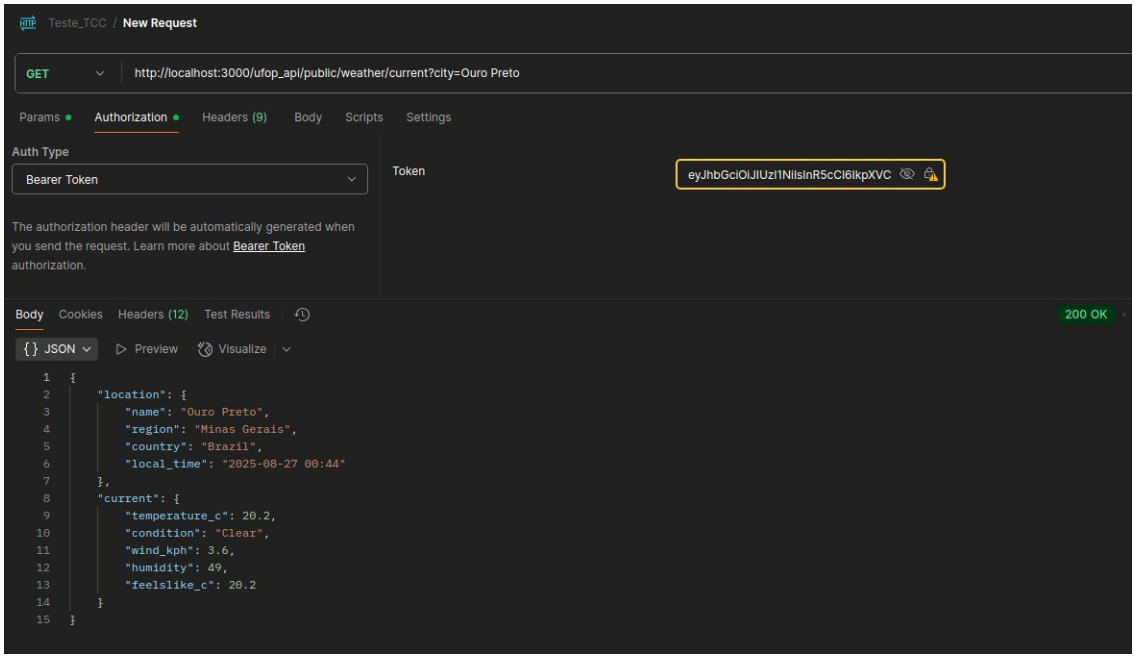


Figura 15 – Consulta de dados meteorológicos com token válido.

5 Conclusão

Este trabalho apresentou de forma abrangente os conceitos fundamentais de APIs, destacando sua implementação prática utilizando a linguagem Python. Ao longo do desenvolvimento, foi possível explorar não apenas a criação de uma API funcional de dados meteorológicos, mas também aspectos essenciais para sua robustez e escalabilidade, como padrões RESTful, arquitetura de APIs, *framework*s, segurança e estratégias de teste.

Mais do que a entrega de um protótipo, este projeto proporcionou um entendimento profundo das etapas envolvidas na concepção e manutenção de uma API moderna. A disponibilização do código-fonte online reforça o caráter colaborativo e didático do trabalho, permitindo que outros desenvolvedores possam aprender, reutilizar ou evoluir a solução.

5.1 Considerações Finais

O presente trabalho teve como objetivo a construção de uma API para fins educacionais e didáticos, disponibilizando dados meteorológicos a partir da integração com a *WeatherAPI*, demonstrando, de forma prática, a importância da utilização de serviços externos em soluções modernas bem como oferecendo um modelo para futuros projetos. A implementação foi conduzida com foco na clareza, modularidade e manutenibilidade, incorporando boas práticas de desenvolvimento, automação com *containers* (Docker), testes em múltiplas camadas bem como documentação técnica e acadêmica.

Através do desenvolvimento, pode-se evidenciar como a integração de APIs de terceiros potencializa a reutilização de dados e reduz a necessidade de construir infraestruturas próprias de coleta e análise, garantindo confiabilidade e eficiência, desde que os fornecedores ofereçam uma base para tal. A aplicação, mesmo em ambiente local, demonstrou a viabilidade do modelo proposto, permitindo consultas em tempo real a dados meteorológicos, validados e tratados para facilitar a leitura.

Além disso, os testes realizados asseguraram que a API está apta a lidar com diferentes cenários de uso, garantindo robustez, consistência e confiabilidade. O registro documental dos processos e decisões, por sua vez, possibilitou a reprodutibilidade científica e fornece uma base sólida para trabalhos futuros.

5.2 Trabalhos Futuros

Embora os resultados obtidos tenham atingido os objetivos inicialmente propostos, este trabalho também abre caminho para uma série de evoluções futuras. Como principal

avanço recomendado, destaca-se o deploy (publicação) em ambiente de nuvem, permitindo que a API deixe de operar apenas em ambiente local e passe a estar acessível de forma pública e escalável.

A realização do deploy em nuvem representa uma etapa natural no ciclo de desenvolvimento e disponibilização de aplicações modernas, uma vez que garante maior escalabilidade, disponibilidade e facilidade de gerenciamento. Nesse contexto, plataformas como **AWS (Amazon Web Services)** e **Google Cloud** surgem como opções relevantes, dado o conjunto abrangente de serviços, documentação consolidada e suporte robusto. No caso da AWS, por exemplo, recursos como EC2 (instâncias de máquinas virtuais), RDS (banco de dados gerenciado), S3 (armazenamento de objetos) e Secrets Manager (gestão segura de credenciais) permitem estruturar uma infraestrutura completa, segura e altamente disponível para hospedar a API meteorológica. De modo semelhante, a Google Cloud oferece serviços integrados com objetivos semelhantes, facilitando a automação e o monitoramento do ambiente. Essa abordagem, que alia flexibilidade a boas práticas de orquestração de contêineres, é discutida por Sampaio e Picoli (2024), ao tratarem da automação de deploy de aplicações em contêineres Docker na nuvem (SAMPAIO; GUILHERME, 2024).

Para tornar essa implantação escalável e reprodutível, recomenda-se o uso do **Terraform** como ferramenta de *Infrastructure as Code* (IaC). A definição da infraestrutura em arquivos de configuração declarativos permitirá a criação, replicação e versionamento dos ambientes de forma automatizada, reduzindo a complexidade operacional e alinhando o projeto às práticas contemporâneas de *DevOps* (HASHICORP, 2023). Isso proporcionaria vantagens como:

- Criação de ambientes de desenvolvimento, homologação e produção idênticos;
- Controle de versão da infraestrutura via Git, possibilitando auditoria e rastreabilidade;
- Automação de processos de provisionamento, reduzindo falhas humanas;
- Aplicação de políticas de segurança de forma centralizada e padronizada.

Além da publicação em nuvem, outras melhorias técnicas podem ser consideradas como trabalhos futuros:

- **Documentação da API**
  Documentação com Swagger: automatizar a documentação de forma interativa com a API, facilitando a compreensão dos *endpoints*, atualização e a integração por terceiros (PAULA STACHISSINI, s.d.).

- **Observabilidade e monitoramento**
  Logs estruturados: implementar mecanismos de registro de eventos em formato estruturado (JSON), permitindo rastreabilidade, auditoria e análise por ferramentas de observabilidade (DROETTBOOM *et al.*, 2015).

- **Qualidade e desempenho**
  Testes de carga: aplicar ferramentas como Apache JMeter ou Locust para avaliar o desempenho e a escalabilidade da API sob diferentes níveis de estresse (CHIMUCO; BARBOSA, 2024).

- **Eficiência no consumo da API**
  Mecanismos de cache: introduzir cache de respostas, seja em memória (Redis) ou distribuído, com o objetivo de reduzir a latência e melhorar a experiência de consumo da API (PEREIRA, 2024).

- **Segurança**
  Prevenção contra SQL Injection: reforçar práticas de segurança no acesso ao banco de dados, utilizando consultas parametrizadas e ORMs robustos, evitando vulnerabilidades comuns em aplicações web (BASILIO; OLIVEIRA, 2022).

Glossário

backend Camada responsável pelo processamento, regras de negócio e comunicação com bancos de dados e serviços externos em uma aplicação.

endpoint Ponto final de acesso em uma API, geralmente representado por uma URL específica que permite consultar, enviar ou manipular dados.

Interoperabilidade Capacidade de diferentes sistemas de hardware e *software* se comunicarem, trocarem dados e utilizarem essas informações de forma eficaz, independentemente da plataforma ou tecnologia utilizada.

Marketplace Plataforma digital que conecta vendedores e compradores, permitindo a comercialização de produtos ou serviços de forma centralizada.

Pull Request Proposta de integração de alterações em um projeto de *software* versionado, geralmente usada em plataformas como GitHub para revisão e colaboração no código.

Issue Registro de problema, melhoria ou sugestão em um repositório de *software*, usado para organizar o desenvolvimento colaborativo.

Referências

ALI, AzraJabeen Mohamed. Optimizing Software Architecture: Using the Repository Pattern in Decoupling Data Access Logic, 2022. Citado 1 vez na página 38.

ALMEIDA, Murilo Góes de; CANEDO, Edna Dias. Authentication and authorization in microservices architecture: A systematic literature review. *Applied sciences*, MDPI, v. 12, n. 6, p. 3023, 2022. Citado 1 vez na página 30.

ALMEIDA, Rafael Marques de. Análise e desenvolvimento de um middleware de segurança para apis. 251, 2021. Citado 1 vez na página 36.

BASILIO, Guilherme Manfrim; OLIVEIRA, Wdson de. Ferramentas de segurança para banco de dados: focando em SQL Injection. *Revista Brasileira em Tecnologia da Informação*, v. 4, n. 2, p. 10–19, 2022. Citado 1 vez na página 45.

BAYER, Michael. *SQLAlchemy Documentation*. 2012. Disponível em: https://docs.sqlalchemy.org/. Citado 1 vez na página 21.

BIGHETI, Jeferson André. Arquitetura de automação e controle orientada a microserviços para a indústria 4.0. Universidade Estadual Paulista (Unesp), 2020. Citado 1 vez na página 23.

BOETTIGER, Carl. An Introduction to Docker for Reproducible Research. *ACM SIGOPS Operating Systems Review*, ACM, v. 49, n. 1, p. 71–79, 2015. Citado 2 vezes nas páginas 22, 23.

CARMO NICOLETTI, Maria do; MARÇAL, Denilson Silva. Mapeamento Objeto-Relacional–Considerações sobre a Ferramenta Ponte entre Programação Orientada a Objetos e Base de Dados Relacional. Citado 1 vez na página 21.

CHACON, Scott; STRAUB, Ben. *Pro Git*. 2. ed.: Apress, 2014. Acesso em: 02 ago. 2025. Disponível em: https://git-scm.com/book/en/v2. Citado 1 vez na página 18.

CHIMUCO, Pedro Ventura Lucunde; BARBOSA, Ana Claudia Garcia. Validação de carga em testes de desempenho de APIs de cadastro de usuários: garantindo qualidade e eficiência com Docker e K6, 2024. Citado 1 vez na página 45.

CODD, Edgar F. A relational model of data for large shared data banks. *Communications of the ACM*, ACM, v. 13, n. 6, p. 377–387, 1970. Citado 1 vez na página 21.

COHEN, Robert. *How Software is Transforming the U.S. Economy*. Jul. 2017. DOI: 10.13140/RG.2.2.13507.91686. Citado 1 vez na página 18.

CONE, Matt. *Markdown guide*. Independently Published, 2020. Citado 1 vez na página 33.

DA COSTA, Walingson da Silva *et al.* Sistema Web para pré-processamento e análise de dados meteorológicos. *Revista Brasileira de Climatologia*, v. 30, p. 591–610, 2022. Citado 2 vezes na página 32.

DABBISH, Laura *et al.* Social coding in GitHub: transparency and collaboration in an open software repository. *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, ACM, p. 1277–1286, 2012. Citado 1 vez na página 19.

DROETTBOOM, Michael *et al.* Understanding json schema. *Available on: http://spacetelescope. github. io/understanding-jsonschema/UnderstandingJSONSchema. pdf (accessed on 14 April 2014)*, 2015. Citado 2 vezes nas páginas 28, 45.

FELTRIN, Larissa. APIs REST: o impacto das APIs REST na evolução digital das empresas. Americana, SP, nov. 2023. Disponível em: https://ric.cps.sp.gov.br/handle/123456789/15349. Citado 1 vez na página 17.

FIELDING, Roy Thomas. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000. Citado 3 vezes nas páginas 14, 20, 35.

FLANAGAN, David. *JavaScript: The Definitive Guide*. 7. ed.: O'Reilly Media, 2020. Citado 1 vez na página 19.

FOUNDATION, OWASP. *OWASP API Security Top 10*. 2023. https://owasp.org/www-project-api-security/. Citado 3 vezes nas páginas 28, 30.

FOWLER, Martin. *Continuous Integration: Improving Software Quality and Reducing Risk*. 1st. Boston, MA: Addison-Wesley Professional, 2006. Citado 1 vez na página 18.

GITLAB INC. *Documentation Guide: Best Practices*. 2023. Acesso em: 17 ago. 2025. Disponível em: https://docs.gitlab.com/ee/development/documentation/. Citado 3 vezes nas páginas 33, 34.

GOMES, Daniel Adorno; JANDL JR, Peter. WEB SERVICES SOAP E REST. *Revista Intellectus*, v. 6, n. 1, p. 88–114, 2009. Citado 1 vez na página 15.

GOMES, Rafael. Docker para desenvolvedores. *Leanpub, Salvador, Bahia,* 2019. Citado 1 vez na página 22.

GRIFFITHS, Kurt *et al. Falcon Documentation*. Release, 2019. Citado 3 vezes nas páginas 18, 37.

GRINBERG, Miguel. *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media, Inc., 2018. Citado 3 vezes nas páginas 17, 36.

GRØNBÆK, Inge. Architecture for the Internet of Things (IoT): API and interconnect. *In:* IEEE. 2008 Second International Conference on Sensor Technologies and Applications (sensorcomm 2008). 2008. p. 802–807. Citado 1 vez na página 12.

HASHICORP. *Terraform: Infrastructure as Code*. 2023. Acesso em: 17 ago. 2025. Disponível em: https://www.terraform.io/. Citado 1 vez na página 44.

HOLOVATY, Adrian; KAPLAN-MOSS, Jacob. *The definitive guide to Django: Web development done right*. Springer, 2009. Citado 1 vez na página 17.

JONES, Michael; BRADLEY, John; SAKIMURA, Nat. *Json web token (jwt)*. 2015. Citado 1 vez na página 31.

JUNIOR, Eudis Fernandes Gomes. APPLICATION PROGRAMMING INTERFACE, 2018. Citado 3 vezes nas páginas 15, 16.

LEADMINE. *Application Programming Interface (API)*. 2023. Disponível em: https://www.leadmine.net/glossary/application-programming-interface/. Acesso em: 16 ago. 2025. Citado 0 vez na página 16.

LIMA, Washington Luiz da Silva. Aplicando BDD em testes de REST API: uma experiência prática. Universidade Federal do Rio Grande do Norte, 2022. Citado 1 vez na página 36.

LUBANOVIC, Bill. *FastAPI*. "O'Reilly Media, Inc.", 2023. Citado 1 vez na página 18.

LUTZ, Mark. *Learning python: Powerful object-oriented programming*. "O'Reilly Media, Inc.", 2013. Citado 1 vez na página 25.

MADDEN, Neil. *API security in action*. Simon e Schuster, 2020. Citado 4 vezes nas páginas 11, 12, 29, 30.

MERKEL, Dirk. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014. Citado 1 vez na página 23.

NASEHI, Seyed Mehdi; MAURER, Frank. Unit tests as API usage examples. *In:* IEEE. 2010 IEEE International Conference on Software Maintenance. 2010. p. 1–10. Citado 1 vez na página 40.

OFOEDA, Joshua; BOATENG, Richard; EFFAH, John. Application programming interface (API) research: A review of the past to inform the future. *International Journal of Enterprise Information Systems (IJEIS)*, IGI Global Scientific Publishing, v. 15, n. 3, p. 76–95, 2019. Citado 1 vez na página 24.

OKKEN, Brian. *Python testing with Pytest: simple, rapid, effective, and scalable*. The Pragmatic Programmers LLC, 2022. Citado 2 vezes nas páginas 33, 40.

PAHL, Claus. Containerization and the PaaS Cloud. *IEEE Cloud Computing*, IEEE, v. 2, n. 3, p. 24–31, 2015. Citado 1 vez na página 23.

PARIZOTTO, Gabriel Zanforlin. Predição de tendências no mercado americano com Random Forest. Universidade Estadual Paulista (Unesp), 2024. Citado 1 vez na página 24.

PAULA STACHISSINI, Carlos Eduardo de. *APLICANDO AS MELHORES PRÁTICAS PARA PUBLICAÇÃO DE DADOS NA WEB ENVOLVENDO A DOCUMENTAÇÃO DE APIS DE ACESSO A DADOS*. Tese (Doutorado) – Universidade Estadual do Centro-Oeste. Citado 1 vez na página 44.

PAVANELI, Tiago Henrique. Desenvolvimento de uma API e uma Aplicação Web para auxiliar na análise de dados providos por Aplicação Móvel. Universidade Federal de Uberlândia, 2023. Citado 1 vez na página 11.

PEREIRA, Maria Eduarda Eloi. *Otimizando a autorização em múltiplos sistemas por meio de cache nas permissões*. 2024. B.S. thesis – Universidade Federal do Rio Grande do Norte. Citado 1 vez na página 45.

PESENTE, Guilherme Moraes. *O ensino de matemática por meio da linguagem de programação Python*. 2019. Diss. (Mestrado) – Universidade Tecnológica Federal do Paraná. Citado 1 vez na página 16.

PRAÇA, Fabíola Silva Garcia. Metodologia da pesquisa científica: organização estrutural e os desafios para redigir o trabalho de conclusão. *Revista Eletrônica Diálogos Acadêmicos*, v. 8, n. 1, p. 72–87, 2015. Citado 2 vezes na página 32.

PRESSMAN, Roger S. *Engenharia de Software: Uma Abordagem Profissional*. São Paulo, Brasil: McGraw-Hill, 2014. Citado 4 vezes nas páginas 32–34.

RAJABOV, Sherzod Baxtiyorovich. The role of backend and frontend information systems infrastructure. *Science and Education*, v. 4, n. 3, p. 212–216, mar. 2023. Disponível em: https://openscience.uz/index.php/sciedu/article/view/5338. Citado 2 vezes nas páginas 19, 20.

RIBEIRO, Leonardo. Graphql: uma alternativa para webservices. Universidade Estadual de Goiás, 2019. Citado 1 vez na página 15.

SABBAG FILHO, Nagib. Boas práticas para o uso de DTOs (Data Transfer Objects) em uma arquitetura limpa. *Leaders Tec*, Leaders Tec, v. 1, n. 26, 2024. Citado 1 vez na página 37.

SAMPAIO, Caio; GUILHERME, Picoli. Automatização de deploy de aplicações em contêineres docker em nuvem. Universidade Presbiteriana Mackenzie, 2024. Citado 1 vez na página 44.

SAMPAIO, Rafael Cardoso; SABBATINI, Marcelo; LIMONGI, Ricardo. Diretrizes para o uso ético e responsável da Inteligência Artificial Generativa: um guia prático para pesquisadores.

Boletim Técnico do PPEC, v. 10, n. 00, e025003, fev. 2025. Disponível em: https://econtents.bc.unicamp.br/boletins/index.php/ppec/article/view/9509. Citado 1 vez na página 25.

SANTOS, Antunes *et al.* A importância do fator motivacional no processo ensino-aprendizagem de algoritmos e lógica de programação para alunos repetentes. *In:* SBC. WORKSHOP sobre Educação em Computação (WEI). 2015. p. 168–177. Citado 1 vez na página 16.

SANTOS, João Vitor Mendes Pinto dos. IMPLEMENTAÇÃO DE UMA API PARA INTEGRAÇÃO DIGITAL ENTRE UMA PLANTA DE MANUFATURA AVANÇADA E UMA FÁBRICA MODELO 4.0. *Revista Brasileira de Automação e Controle*, 2022. Citado 1 vez na página 23.

SILVA, Patrícia Nascimento. Observatório de dados governamentais abertos: acesso às APIs brasileiras. *Revista ACB*, Universidade Federal de Minas Gerais, 2023. Citado 1 vez na página 12.

SILVA, Patrícia Nascimento; SILVA, Gabriel Vieira Pereira da. DADOS ABERTOS DO IBGE: RECUPERAÇÃO NA API DE DADOS AGREGADOS. *Encontros Bibli*, SciELO Brasil, v. 29, e96185, 2024. Citado 1 vez na página 12.

SOUSA, Hitalo Cunha de; ALMEIDA E SILVA, Bruno Ramon de; BANDEIRA, Júnior Marcos. USO DE API RESTFUL PARA GESTÃO DE EVENTOS. *Anais do Congresso de Iniciação Científica*, 2017. Disponível em: https://www.unibalsas.edu.br/wp-content/uploads/2017/01/Hitalo.pdf. Citado 1 vez na página 15.

SRIRAMYA, P.; KARTHIKA, R. A. Providing password security by salted password hashing using bcrypt algorithm. *ARPN Journal of Engineering and Applied Sciences*, v. 10, n. 13, p. 5551–5556, 2015. Citado 2 vezes nas páginas 29, 30.

TAVARES, Vitor. *Repositório do Projeto de API TCC em Python*. 2025. https://github.com/VitorTG/api_tcc. Acessado em: 17 fev. 2025. Citado 1 vez na página 33.

TSAY, Jason; DABBISH, Laura; HERBSLEB, James. Influence of social and technical factors for evaluating contribution in GitHub. *In:* ACM. PROCEEDINGS of the 36th International Conference on Software Engineering. 2014. p. 356–366. Citado 1 vez na página 19.

TURNBULL, James. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014. Citado 1 vez na página 23.

VASCONCELOS, Constantino Francisco. Utilizando a linguagem Python para resolução de problemas de física, 2022. Citado 1 vez na página 16.

WEATHER API. *Weather API Documentation*. 2025. Acesso em: 07 ago. 2025. Disponível em: https://www.weatherapi.com/docs/. Citado 2 vezes nas páginas 31, 39.