Universidade Federal de Ouro Preto
Instituto de Ciências Exatas e Aplicadas
Departamento de Computação e Sistemas

Coinverter: uma API para conversão de moedas

Matheus Lucas Silva Meirelles Reis

TRABALHO DE CONCLUSÃO DE CURSO

ORIENTAÇÃO:
Fernando Bernardes de Oliveira

Setembro, 2025
João Monlevade–MG

Matheus Lucas Silva Meirelles Reis

Coinverter: uma API para conversão de moedas

Orientador: Fernando Bernardes de Oliveira

Monografia apresentada ao curso de Engenharia de Computação do Instituto de Ciências Exatas e Aplicadas, da Universidade Federal de Ouro Preto, como requisito parcial para aprovação na Disciplina "Trabalho de Conclusão de Curso II".

Universidade Federal de Ouro Preto

João Monlevade

Setembro de 2025

SISBIN - SISTEMA DE BIBLIOTECAS E INFORMAÇÃO

R375c Reis, Matheus Lucas Silva Meirelles.
Coinverter [manuscrito]: uma API para conversão de moedas. / Matheus Lucas Silva Meirelles Reis. - 2025.
84 f.: il.: color., tab..

Orientador: Prof. Dr. Fernando Bernardes de Oliveira.
Monografia (Bacharelado). Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Aplicadas. Graduação em Engenharia de Computação .

1. Conversibilidade de papel moeda. 2. Engenharia de software. 3. Software - Confiabilidade. 4. Software - Testes. 5. Software - Validação. 6. Software de aplicação - Desenvolvimento. I. Oliveira, Fernando Bernardes de. II. Universidade Federal de Ouro Preto. III. Título.

CDU 004.41

Bibliotecário(a) Responsável: Flavia Reis - CRB6-2431

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E APLICADAS
DEPARTAMENTO DE COMPUTACAO E SISTEMAS

FOLHA DE APROVAÇÃO

Matheus Lucas Silva Meirelles Reis

Coinverter: uma API para conversão de moedas

Monografia apresentada ao Curso de Engenharia de Computação da Universidade Federal
de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Engenharia de Computação

Aprovada em 10 de setembro de 2025

Membros da banca

Prof. Dr. Fernando Bernardes de Oliveira - Orientador (Universidade Federal de Ouro Preto)
Prof. Dr. Igor Muzetti Pereira - Avaliador (Universidade Federal de Ouro Preto)
Prof. Dr. Rafael Frederico Alexandre - Avaliador (Universidade Federal de Ouro Preto)

Fernando Bernardes de Oliveira, orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 10/09/2025

Documento assinado eletronicamente por **Fernando Bernardes de Oliveira**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 10/09/2025, às 14:21, 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 **0976371** e o código CRC **3FE95F11**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: (31)3808-0819 - www.ufop.br

Este trabalho é dedicado a todas as pessoas que me apoiaram durante à minha graduação

Agradecimentos

Aos meus pais, Emerson e Ana Cristina, expresso minha mais profunda gratidão pelo apoio incondicional, pelos sacrifícios realizados e por sempre acreditarem no meu potencial. Vocês são a base de tudo que conquistei e espero conquistar.

Ao meu tio Marcos Paulo, por ser um exemplo de determinação e perseverança, sempre me inspirando a buscar a excelência em tudo que faço.

À minha querida avó Nadir, pelo carinho constante, pelas palavras de sabedoria e por sempre torcer pelo meu sucesso.

À minha namorada Carine, por estar presente em todos os momentos desta jornada, oferecendo compreensão nas horas difíceis, celebrando as conquistas e sendo minha fonte de força e motivação.

Ao meu orientador, Professor Fernando, pela dedicação, paciência e conhecimento compartilhado durante o desenvolvimento deste trabalho. Sua orientação foi fundamental para o sucesso desta pesquisa.

À Universidade Federal de Ouro Preto e ao Departamento de Computação e Sistemas, pela oportunidade de fazer parte desta instituição de excelência e por proporcionar um ambiente acadêmico de qualidade.

Aos professores de Engenharia da computação, pelos conhecimentos transmitidos e pela dedicação ao ensino, contribuindo para minha formação profissional e pessoal.

A todos os colegas de curso, pelas experiências compartilhadas, pelos momentos de estudo em grupo e pela amizade construída ao longo desses anos.

A todos que, direta ou indiretamente, contribuíram para a realização deste trabalho e para minha formação acadêmica.

"Little by little, one travels far."

J.R.R. Tolkien

Resumo

O crescimento das transações financeiras internacionais e a globalização dos mercados digitais demandam soluções tecnológicas robustas para conversão monetária em tempo real. Este trabalho apresenta o desenvolvimento da Coinverter, uma *Application Programming Interface* (API) *Representational State Transfer* compliant (RESTful) de conversão de moedas que implementa o padrão internacional *International Organization for Standardization* (ISO) 4217, fornecendo um serviço confiável, escalável e seguro para aplicações que necessitam realizar operações cambiais automatizadas. O objetivo principal consiste em criar uma solução completa que não apenas realize conversões precisas entre diferentes moedas, mas também forneça cotações atualizadas em tempo real, histórico de transações e garantias de conformidade com a Lei Geral de Proteção de Dados (LGPD). A metodologia adotada fundamenta-se em práticas modernas de engenharia de software, iniciando com análise detalhada dos requisitos funcionais e não-funcionais, seguida pela modelagem arquitetural baseada em *Domain-Driven Design* (DDD) e princípios *Single responsibility, Open-closed, Liskov substitution, Interface segregation, Dependency inversion* (SOLID). O desenvolvimento utiliza containerização via *Docker* para garantir portabilidade e consistência entre ambientes, implementação de *pipeline* de *Continuous Delivery / Continuous Integration* (CI/CD) para automatização completa do ciclo de desenvolvimento, desde a integração contínua até a implantação automatizada, e aplicação de *Test-Driven Development* (TDD) para assegurar qualidade e confiabilidade do código. A validação do sistema contempla testes automatizados de carga, desempenho e segurança, além de simulações com dados reais de mercado para verificar a precisão das conversões e a latência das respostas. Os resultados esperados incluem uma API com tempo de resposta inferior a 200ms para 95% das requisições, disponibilidade de 99,9%, suporte a mais de 150 moedas conforme ISO 4217, documentação completa seguindo a especificação *OpenAPI* 3.0, e ambiente de testes para desenvolvedores. A solução proposta contribui para o ecossistema de desenvolvimento de software ao demonstrar a aplicação prática de padrões arquiteturais modernos, boas práticas de *Development and Operations* (DevOps) e conformidade regulatória em um contexto de alta demanda por precisão e confiabilidade. **Palavras-chave**:

API *Representational State Transfer* (REST). Conversão de Moedas. ISO 4217. *Docker*. CI/CD. DDD.

Abstract

The growth of international financial transactions and the globalization of digital markets demand robust technological solutions for real-time currency conversion. This work presents the development of Coinverter, a RESTful currency conversion API that implements the international ISO 4217 standard, providing a reliable, scalable, and secure service for applications requiring automated foreign exchange operations. The main objective is to create a comprehensive solution that not only performs accurate conversions between different currencies but also provides real-time updated exchange rates, transaction history, and compliance guarantees with the Brazilian General Data Protection Law (LGPD). The adopted methodology is based on modern software engineering practices, starting with a detailed analysis of functional and non-functional requirements, followed by architectural modeling based on Domain-Driven Design (DDD) and SOLID principles. The development employs containerization via Docker to ensure portability and consistency across environments, implementation of CI/CD pipeline for complete automation of the development cycle from continuous integration to automated deployment, and application of Test-Driven Development (TDD) to ensure code quality and reliability. System validation encompasses automated load, performance, and security testing, as well as simulations with real market data to verify conversion accuracy and response latency. Expected results include an API with response time below 200ms for 95% of requests, 99.9% availability, support for over 150 currencies according to ISO 4217, complete documentation following OpenAPI 3.0 specification, and a sandbox environment for developers. The proposed solution contributes to the software development ecosystem by demonstrating the practical application of modern architectural patterns, DevOps best practices, and regulatory compliance in a context of high demand for precision and reliability.

Keywords: REST API. Currency Conversion. ISO 4217. Docker. CI/CD. Domain-Driven Design.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ABC *Abstract Base Classes*

API *Application Programming Interface*

AST *Abstract Syntax Tree*

BSON *Binary JSON*

CD *Continuous Delivery*

CI *Continuous Integration*

CI/CD *Continuous Delivery / Continuous Integration*

CORS *Cross-Origin Resource Sharing*

CRUD *Create, Read, Update, Delete*

CSS *Cascading Style Sheets*

DAST *Dynamic Application Security Testing*

DDD *Domain-Driven Design*

DevOps *Development and Operations*

DOM *Document Object Model*

ES *ECMAScript*

GridFS *Grid File System*

HTML *HyperText Markup Language* – Linguagem de Marcação de Hipertexto

HTTP *Hypertext Transfer Protocol*

HTTPS *Hypertext Transfer Protocol Secure*

ISO *International Organization for Standardization*

JSON *JavaScript Object Notation*

JSX *JavaScript XML*

JWT *JSON Web Token*

LGPD Lei Geral de Proteção de Dados

NoSQL *Not Only SQL*

PEP *Python Enhancement Proposal*

PyPI *Python Package Index*

REST *Representational State Transfer*

RESTful *Representational State Transfer* compliant

SAST *Static Application Security Testing*

SLA *Service Level Agreement*

SOLID *Single responsibility, Open-closed, Liskov substitution, Interface segregation, Dependency inversion*

SPA *Single Page Application*

SQL *Structured Query Language*

TDD *Test-Driven Development*

TSL *Transport Layer Security*

TTL *Time To Live*

UI *User Interface*

XML *eXtensible Markup Language*

YAML *YAML Ain't Markup Language*

Sumário

1 Introdução

A globalização dos mercados digitais e o crescimento exponencial do comércio eletrônico internacional estabeleceram um cenário onde aplicações necessitam realizar conversões monetárias precisas e em tempo real. Sistemas de *e-commerce*, plataformas de pagamento, aplicações *fintech* e serviços bancários digitais demandam soluções robustas que possam processar milhares de requisições simultâneas mantendo alta disponibilidade e precisão nas conversões entre diferentes moedas.

O padrão ISO 4217, estabelecido pela *International Organization for Standardization*, define códigos alfanuméricos de três letras para representação padronizada de moedas em sistemas computacionais. Esta norma internacional elimina ambiguidades na identificação de moedas e facilita a interoperabilidade entre sistemas financeiros globais. Entretanto, a implementação de uma solução que utilize este padrão de forma eficiente, segura e escalável apresenta desafios técnicos significativos que serão abordados neste trabalho.

Este documento apresenta o desenvolvimento da Coinverter, uma API RESTful de conversão de moedas que implementa o padrão ISO 4217 e incorpora práticas modernas de engenharia de software. A solução proposta utiliza conceitos de DDD, princípios SOLID, TDD, containerização com *Docker* e *pipeline* CI/CD automatizada. Além disso, o sistema garante conformidade com a LGPD através de mecanismos apropriados de segurança e auditoria.

1.1 Elaboração do capítulo

Este capítulo apresenta o desenvolvimento de uma API RESTful para conversão de moedas seguindo o padrão internacional ISO 4217. O trabalho contextualiza os desafios técnicos e regulatórios envolvidos na construção de sistemas financeiros distribuídos, estabelece objetivos claros para a implementação da solução, define a metodologia de desenvolvimento baseada em práticas ágeis e DevOps, e estrutura a organização dos capítulos subsequentes.

A motivação principal surge da necessidade crescente de sistemas confiáveis para processamento de transações financeiras internacionais. Segundo Tanenbaum e Wetherall (2011), arquiteturas distribuídas modernas requerem serviços especializados que possam ser consumidos de forma padronizada e escalável. No contexto específico de conversão monetária, isso significa desenvolver uma solução que não apenas realize cálculos precisos, mas também gerencie aspectos como volatilidade cambial, latência de rede, segurança das

transações e conformidade regulatória.

1.2 O problema de pesquisa

O problema de pesquisa centra-se na complexidade de desenvolver uma API de conversão de moedas que atenda simultaneamente aos requisitos técnicos de desempenho, escalabilidade e segurança, enquanto mantém conformidade com regulamentações de proteção de dados e padrões internacionais de representação monetária.

Aplicações que necessitam realizar conversões entre moedas enfrentam diversos desafios: a obtenção de cotações atualizadas e confiáveis, o processamento eficiente de grandes volumes de requisições, a garantia de precisão nos cálculos considerando diferentes casas decimais entre moedas, a implementação de mecanismos de *cache* inteligentes para otimizar desempenho sem comprometer a atualidade dos dados, e a proteção adequada de informações sensíveis conforme exigido pela LGPD.

APIs comerciais existentes como *ExchangeRate-API*, *Fixer.io* e *CurrencyLayer* apresentam limitações significativas em seus planos gratuitos, incluindo restrições severas no número de requisições, ausência de funcionalidades avançadas como conversão histórica, e falta de transparência nas implementações. Além disso, a dependência de serviços terceirizados pode representar riscos de disponibilidade e privacidade para aplicações críticas.

Neste contexto, questiona-se: Como desenvolver uma API de conversão de moedas que seja simultaneamente precisa, eficiente, segura e compatível com padrões internacionais? Quais padrões arquiteturais e práticas de desenvolvimento são mais adequados para garantir manutenibilidade e escalabilidade da solução? Como implementar mecanismos eficazes de proteção de dados sem comprometer a usabilidade do sistema?

1.3 Objetivos

O presente trabalho consiste em desenvolver uma API RESTful completa para conversão de moedas, denominada Coinverter, que implemente o padrão ISO 4217 e demonstre a aplicação prática de conceitos modernos de engenharia de software em um sistema de produção.

Este trabalho possui os seguintes objetivos específicos:

1. Modelar e implementar uma API para conversão de moedas, com suporte a cotações atualizadas em tempo real;

2. Validar a API desenvolvida por meio de testes automatizados e simulações em cenários reais de uso;

3. Containerizar a aplicação utilizando *Docker*, garantindo portabilidade e facilidade de implantação em diferentes ambientes;

4. Estabelecer uma *pipeline* de CI/CD (Integração Contínua e Entrega Contínua) para automatizar o processo de *build*, testes e implantação da aplicação;

5. Estudar e aplicar boas práticas de segurança para proteger os dados dos usuários e as transações realizadas pela API.

1.4 Metodologia

O objeto de pesquisa deste trabalho consiste no desenvolvimento sistemático de uma API de conversão de moedas aplicando metodologias ágeis e práticas DevOps. A abordagem metodológica combina pesquisa aplicada com desenvolvimento experimental, fundamentada em revisão bibliográfica de literatura especializada e análise comparativa de soluções existentes.

Os passos para execução deste trabalho são assim definidos:

1. Revisão da literatura sobre arquiteturas de APIs RESTful, padrão ISO 4217, DDD, princípios SOLID e práticas DevOps;

2. Análise comparativa de APIs de conversão existentes identificando funcionalidades, limitações e oportunidades de melhoria;

3. Levantamento e especificação detalhada de requisitos funcionais e não-funcionais do sistema;

4. Modelagem do domínio utilizando DDD com identificação de entidades, agregados, serviços e eventos;

5. Desenvolvimento iterativo da API aplicando TDD com ciclos curtos de *red-green-refactor*;

6. Implementação de integrações com provedores externos de cotações utilizando padrões de resiliência;

7. Containerização da aplicação com *Docker* e orquestração com *Docker Compose*;

8. Criação de *pipeline* CI/CD utilizando *GitHub Actions* para automação completa;

9. Execução de testes de carga simulando cenários reais de uso;

10. Validação de segurança por meio de análise estática: *Static Application Security Testing* (SAST) e dinâmica: *Dynamic Application Security Testing* (DAST) de vulnerabilidades;

11. Análise e discussão dos resultados obtidos com métricas de desempenho, disponibilidade e qualidade de código;

12. Documentação técnica completa e elaboração de guias para desenvolvedores.

1.5 Organização do trabalho

O restante deste trabalho é organizado como segue. O Capítulo 2 apresenta a revisão da literatura abordando conceitos fundamentais de APIs RESTful, o padrão ISO 4217, arquiteturas de software baseadas em DDD, princípios SOLID, práticas de TDD e DevOps, tecnologias como *Python*, *FastAPI*, *MongoDB*, *React* e *TailwindCSS*, além de analisar trabalhos relacionados e soluções comerciais existentes.

O Capítulo 3 detalha o desenvolvimento completo da API Coinverter, incluindo a arquitetura do sistema, modelagem do domínio, implementação dos componentes principais, integrações com provedores externos, desenvolvimento do *frontend*, containerização com *Docker*, estratégia de testes abrangente com *pytest*, *pipeline* CI/CD automatizada com *GitHub Actions*, e mecanismos de segurança e conformidade com a LGPD.

O Capítulo 4 apresenta os resultados obtidos, incluindo validação dos requisitos funcionais e não-funcionais, métricas de qualidade de código com cobertura de testes superior a 90%, análise de desempenho da API, avaliação da interface *frontend*, análise de segurança, e comparação com soluções comerciais existentes.

O Capítulo 5 conclui o trabalho sintetizando as principais contribuições técnicas, acadêmicas e práticas realizadas, confirmando o alcance de todos os objetivos propostos, discutindo as limitações identificadas, e apresentando sugestões detalhadas para trabalhos futuros, incluindo melhorias de funcionalidade, infraestrutura e pesquisas avançadas.

2 Revisão Bibliográfica

Este capítulo apresenta os fundamentos teóricos e tecnológicos que embasam o desenvolvimento da API Coinverter. A revisão abrange conceitos arquiteturais de APIs RESTful, o padrão internacional ISO 4217 para representação de moedas, princípios de design de software como DDD e SOLID, práticas de desenvolvimento ágil incluindo TDD, tecnologias de containerização e orquestração, além de metodologias DevOps para automação de processos. Ao final, são analisados trabalhos relacionados e soluções comerciais existentes no mercado.

2.1 *Python* como Linguagem de Desenvolvimento

Python é uma linguagem de programação de alto nível, interpretada e de propósito geral, criada por Guido van Rossum em 1991. Sua filosofia de *design* enfatiza legibilidade de código através de sintaxe clara e uso de indentação significativa. Lutz (2013) destaca que *Python* tornou-se uma das linguagens mais populares para desenvolvimento *web*, análise de dados e automação devido à sua curva de aprendizado suave e vasto ecossistema de bibliotecas.

2.1.1 Características Fundamentais

Tipagem Dinâmica e Forte: *Python* é dinamicamente tipado, permitindo flexibilidade no desenvolvimento, mas mantém tipagem forte que previne operações não-seguras entre tipos incompatíveis. Esta característica acelera prototipagem mas requer disciplina para manter qualidade em sistemas grandes.

Gerenciamento Automático de Memória: O *garbage collector* do *Python* gerencia alocação e liberação de memória automaticamente, eliminando classe inteira de *bugs* relacionados a vazamentos de memória e ponteiros pendentes. O algoritmo de contagem de referência complementado por detecção de ciclos garante eficiência na maioria dos cenários.

Paradigmas Múltiplos: Suporta programação procedural, orientada a objetos e funcional, permitindo escolher paradigma mais adequado para cada problema. Esta flexibilidade é particularmente valiosa em APIs onde diferentes camadas beneficiam-se de diferentes abordagens.

Biblioteca Padrão Abrangente: A filosofia *"batteries included"* fornece módulos para tarefas comuns como manipulação de *JavaScript Object Notation* (JSON), requisi-

ções *Hypertext Transfer Protocol* (HTTP), expressões regulares e criptografia, reduzindo dependências externas.

2.1.2 *Python* para APIs *Web*

Python consolidou-se como escolha popular para desenvolvimento de APIs devido a *frameworks* maduros e desempenho adequado para maioria dos casos de uso. Grinberg (2018) demonstra que *Python* pode entregar desempenho comparável a linguagens compiladas quando utilizando técnicas apropriadas como programação assíncrona e *caching* inteligente.

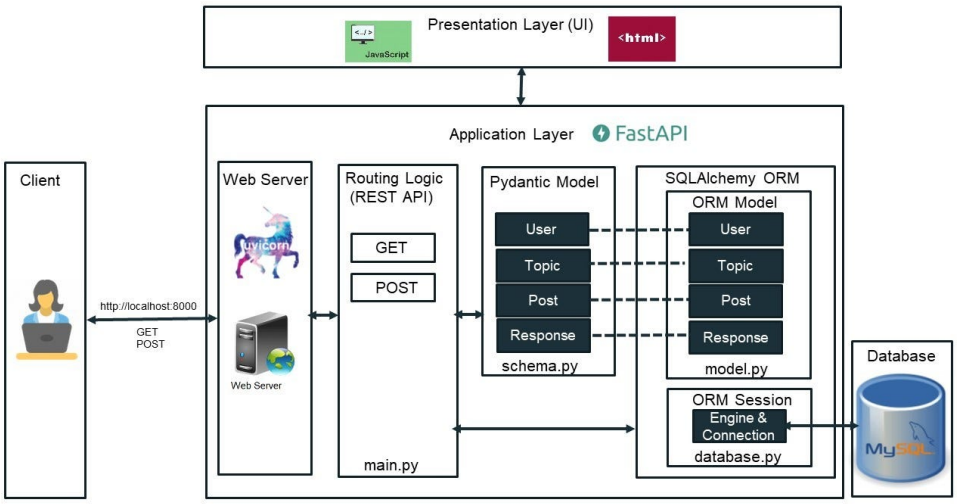
Ecossistema Rico: *Python Package Index* (PyPI) hospeda mais de 400.000 pacotes, incluindo bibliotecas especializadas para praticamente qualquer domínio. Para APIs financeiras, bibliotecas como `decimal` para aritmética precisa e `python-dateutil` para manipulação de datas são essenciais.

Async/Await: Suporte nativo para programação assíncrona desde *Python* 3.5 permite tratamento eficiente de operações vinculadas a E/S, crucial para APIs que dependem de serviços externos para obtenção de cotações.

2.2 *Framework FastAPI*

FastAPI é um *framework web* moderno e de alta performance para construção de APIs com *Python* 3.7+ baseado em *type hints* padrão. Criado por Sebastián Ramírez em 2018, *FastAPI* rapidamente ganhou popularidade devido à combinação única de desempenho, produtividade e recursos modernos. No projeto Coinverter, *FastAPI* foi escolhido como *framework* principal devido à sua capacidade de gerar documentação automática e validação de dados integrada.

Figura 1 – Arquitetura de Microsserviços



Fonte: Disponível em: <https://nachi-keta.medium.com/fastapi-an-intro-e057dc6be20f>

2.2.1 Características Principais

Desempenho Superior: Construído sobre *Starlette* para partes *web* e *Pydantic* para validação de dados, *FastAPI* alcança desempenho comparável a *Node.js* e *Go*. *Benchmarks* independentes demonstram que *FastAPI* é um dos *frameworks Python* mais rápidos disponíveis, superando alternativas estabelecidas como *Flask* e *Django*.

***Type Hints* e Validação Automática**: Utiliza *Python type hints* para validação automática de *request/response*, serialização e documentação. Modelos *Pydantic* garantem que dados sejam validados e convertidos automaticamente, eliminando código repetitivo e reduzindo *bugs*. No Coinverter, isso é especialmente útil para validar códigos ISO 4217 e valores monetários.

Documentação Automática: Gera documentação interativa automaticamente usando *OpenAPI* (*Swagger UI*) e *ReDoc*. A documentação é sempre sincronizada com código, eliminando divergências comuns em APIs mantidas manualmente.

Suporte Nativo para *Async*: Projetado com *async/await* como cidadão de primeira classe, permitindo tratamento eficiente de milhares de conexões simultâneas. Essencial para APIs que fazem múltiplas chamadas a serviços externos.

2.2.2 Injeção de Dependência

FastAPI implementa sistema sofisticado de injeção de dependências que simplifica gerenciamento de recursos compartilhados como conexões de banco de dados, clientes HTTP e configurações. Ramirez (2023) demonstra como este sistema promove código testá-

vel e manutenível. No Coinverter, utilizamos este recurso através da função `get_database` que injeta conexão *MongoDB* nos *endpoints*.

2.2.3 Validação e Serialização com *Pydantic*

Integração profunda com *Pydantic* permite definir esquemas de dados usando classes *Python* com *type hints*. Validação ocorre automaticamente, incluindo coerção de tipos, validação de intervalos e formatos, e geração de mensagens de erro descritivas. Os esquemas `CurrencySchema`, `CurrenciesPriceInputSchema` e `CurrenciesPriceOutputSchema` do Coinverter demonstram esta integração.

2.3 *MongoDB* como Banco de Dados

MongoDB é um banco de dados *Not Only SQL* (NoSQL) orientado a documentos que armazena dados em formato similar a JSON (*Binary JSON* (BSON) - *Binary JSON*). Desenvolvido pela *MongoDB Inc.*, tornou-se uma das soluções NoSQL mais populares devido à flexibilidade, escalabilidade horizontal e modelo de dados intuitivo. No Coinverter, *MongoDB* foi escolhido para armazenar informações sobre moedas cadastradas devido à sua simplicidade e flexibilidade de esquema.

2.3.1 Modelo de Dados Orientado a Documentos

Diferentemente de bancos relacionais que armazenam dados em tabelas com esquemas rígidos, *MongoDB* organiza dados em coleções de documentos. Copeland (2022) explica que este modelo é particularmente adequado para dados semi-estruturados e aplicações que evoluem rapidamente:

Flexibilidade de Esquema: Documentos na mesma coleção podem ter estruturas diferentes, permitindo evolução gradual do modelo de dados sem migrações complexas. Para API de conversão, isso facilita adição de novos campos como metadados de conformidade ou informações de auditoria.

Documentos Incorporados: Dados relacionados podem ser armazenados juntos em único documento, eliminando necessidade de *joins* e melhorando desempenho de leitura. No Coinverter, cada moeda é um documento simples com campos `_id`, `name` e `iso_4217`.

***Arrays* e Subdocumentos**: Suporte nativo para estruturas complexas permite modelagem natural de dados hierárquicos. Lista de moedas suportadas com suas propriedades pode ser armazenada como *array* de subdocumentos.

2.3.2 Recursos para Alta Disponibilidade

Replica Sets: *MongoDB* suporta replicação automática com *failover* transparente. Nó primário recebe escritas enquanto secundários mantêm cópias para leitura e *backup*. Em caso de falha do primário, eleição automática promove secundário, garantindo disponibilidade contínua.

Sharding: Distribuição horizontal de dados através de múltiplos servidores permite escalabilidade praticamente ilimitada. Para APIs com alto volume, *sharding* por região geográfica ou intervalo de *timestamps* otimiza desempenho.

Preferências de Leitura: Controle fino sobre onde leituras são executadas permite balancear consistência versus desempenho. Leituras podem ser direcionadas para secundários para distribuir carga, mantendo escritas no primário para garantir consistência.

2.3.3 Índices e Desempenho

MongoDB oferece variedade de tipos de índices para otimização de consultas:

- **Campo Único**: Índices em campos individuais para consultas simples
- **Composto**: Múltiplos campos para consultas complexas
- **Múltiplas Chaves**: Indexação automática de *arrays*
- **Texto**: Busca textual com suporte a múltiplos idiomas
- ***Time To Live*** (**TTL**): Tempo de Vida para expiração automática de documentos
- **Geoespacial**: Consultas baseadas em localização geográfica

Para API de conversão, índices compostos em (from_currency, to_currency, timestamp) otimizam consultas de histórico, enquanto índices TTL podem gerenciar *cache* de cotações automaticamente.

2.4 *Python Enhancement Proposal* (PEP) 8 - Guia de Estilo para Código *Python*

Python Enhancement Proposal (PEP) 8 é o guia de estilo oficial para código *Python*, escrito por Guido van Rossum, Barry Warsaw e Nick Coghlan. Estabelece convenções para formatação de código, nomenclatura e organização, promovendo consistência e legibilidade no ecossistema *Python*. O projeto Coinverter segue rigorosamente PEP 8, utilizando ferramentas como *Black* e *Flake8* para garantir conformidade.

2.4.1 Princípios Fundamentais

O PEP 8 baseia-se no princípio de que código é lido muito mais frequentemente do que é escrito. Rossum (2001) enfatiza que legibilidade conta (*"Readability counts"*) e que consistência dentro de projeto é mais importante que aderência cega às convenções.

2.4.2 Convenções de Formatação

Indentação: Usar 4 espaços por nível de indentação. Tabulações devem ser evitadas exceto em código que já as utiliza. Continuação de linhas deve alinhar com delimitador de abertura ou usar indentação pendente.

Comprimento de Linha: Limitar linhas a 79 caracteres para código e 72 para comentários e *docstrings*. Permite visualização lado a lado e evita quebra automática em terminais de 80 colunas. No Coinverter, foi configurado *Black* com 88 caracteres seguindo a recomendação.

Importações: Devem estar sempre no topo do arquivo, após *docstring* do módulo e antes de globais. Ordenados em grupos: biblioteca padrão, terceiros, aplicação local. Importações absolutas são preferidas sobre relativas.

Espaçamento: Evitar espaços extras em parênteses, colchetes e chaves. Usar espaços ao redor de operadores com mesma precedência. Não usar espaços para indicar argumentos nomeados ou valores padrão.

2.4.3 Convenções de Nomenclatura

- **Módulos e Pacotes**: minúsculas com sublinhados (*snake_case*)
- **Classes**: *CapWords* (*PascalCase*)
- **Funções e Variáveis**: minúsculas com sublinhados
- **Constantes**: MAIÚSCULAS com sublinhados
- **Métodos Privados**: Prefixo com sublinhado único
- **Manipulação de Nomes**: Prefixo com sublinhado duplo para evitar conflitos em subclasses

2.4.4 *Docstrings*

PEP 257 complementa PEP 8 definindo convenções para *docstrings*. Todas as funções públicas, classes e módulos devem ter *docstrings*.

2.5 *Motor* - *Driver* Assíncrono para *MongoDB*

Motor é o *driver* oficial assíncrono para *MongoDB* em *Python*, construído sobre *PyMongo* e compatível com *frameworks* assíncronos como *asyncio*, *Tornado* e *FastAPI*. Desenvolvido pela *MongoDB Inc.*, *Motor* permite operações não-bloqueantes mantendo API familiar do *PyMongo*. No Coinverter, *Motor* é utilizado para todas as operações de banco de dados, aproveitando a natureza assíncrona do *FastAPI*.

2.5.1 Arquitetura Assíncrona

O *Motor* utiliza *event loop* do *AsyncIO* para executar operações de E/S de forma não-bloqueante. MongoDB (2023) explica que isso permite que aplicação continue processando outras requisições enquanto aguarda resposta do banco de dados, melhorando significativamente vazão em cenários de alta concorrência.

Compatibilidade com *AsyncIO*: *Motor* integra nativamente com sintaxe *async/await*, permitindo código limpo e intuitivo. No Coinverter, todas as operações de banco são marcadas com `async` e utilizam `await`.

Agrupamento de Conexões: *Motor* mantém conjunto de conexões reutilizáveis, evitando sobrecarga de estabelecer nova conexão para cada operação. O conjunto é *thread-safe* e automaticamente gerencia conexões ociosas, reconexões e *timeouts*. Configuramos `maxPoolSize` e `minPoolSize` no Coinverter para otimizar uso de recursos.

2.5.2 Operações Assíncronas

Operações *Create, Read, Update, Delete* (CRUD): Todas operações CRUD do *PyMongo* estão disponíveis em versão assíncrona. Inserção, busca, atualização e exclusão retornam *coroutines* que devem ser aguardadas.

***Pipeline* de Agregação**: Operações complexas de agregação também são suportadas assincronamente, permitindo processamento eficiente de grandes volumes de dados.

2.5.3 *GridFS* Assíncrono

Motor inclui suporte assíncrono para *Grid File System* (GridFS), sistema de armazenamento de arquivos grandes do *MongoDB*. Útil para armazenar relatórios, *logs* ou *backups* de dados históricos.

2.5.4 *Change Streams*

Motor suporta *MongoDB Change Streams*, permitindo aplicações reagirem a mudanças em tempo real. Essencial para invalidação de *cache*, auditoria e sincronização.

2.6 *Black* - O Formatador de Código Intransigente

Black é um formatador de código *Python* determinístico que aplica estilo consistente automaticamente. Criado por Łukasz Langa em 2018 sob *Python Software Foundation*, *Black* elimina debates sobre formatação ao impor estilo único e não-configurável. O Coinverter utiliza *Black* versão 22.3.0 para garantir formatação consistente em todo o código.

2.6.1 Filosofia e *Design*

Black adota filosofia *"uncompromising"* onde decisões de formatação não são configuráveis. Langa (2020) argumenta que isso libera desenvolvedores de discussões sobre estilo, permitindo foco em lógica e arquitetura. O resultado é código com aparência uniforme independente do autor.

Determinismo: Dado mesmo *input*, *Black* sempre produz mesmo *output*. Isso garante que reformatação não introduz diferenças desnecessárias e facilita integração com controle de versão.

Estabilidade: *Black* garante que código reformatado mantém mesmo *Abstract Syntax Tree* (AST), exceto por casos documentados. Isso previne introdução de *bugs* durante formatação.

Minimalismo: Configuração limitada a poucos parâmetros essenciais como comprimento de linha e versão *Python* alvo. Decisões estilísticas são pré-definidas baseadas em PEP 8 e melhores práticas.

2.6.2 Características de Formatação

Comprimento de Linha: Padrão de 88 caracteres (ligeiramente maior que PEP 8) para melhor utilização de espaço horizontal em monitores modernos.

Aspas de *String*: Normaliza para aspas duplas, exceto quando *string* contém aspas duplas internamente.

Vírgulas Finais: Adiciona automaticamente em coleções de múltiplas linhas para minimizar diferenças em mudanças futuras.

Vírgula Final Mágica: Preserva formatação de múltiplas linhas quando vírgula final está presente, dando controle limitado ao desenvolvedor.

2.6.3 Integração com Desenvolvimento

Pre-commit Hooks: *Black* integra com *framework pre-commit* para formatar código automaticamente antes de *commits*.

Integração com Editor: Suporte para *VS Code*, *PyCharm*, *Vim*, *Emacs* e outros editores permite formatação ao salvar ou via atalho de teclado.

Integração CI/CD: *Black* pode ser executado em modo verificação para validar formatação sem modificar arquivos, ideal para *pipelines* de *Continuous Integration* (CI). No Coinverter, o comando `make lint` executa *Black* em modo verificação.

2.6.4 Compatibilidade e Adoção

Black é utilizado por projetos proeminentes incluindo *Django*, *SQLAlchemy*, *Poetry* e *pytest*. Langa (2020) reporta que projetos que adotam *Black* experimentam redução significativa em comentários de revisão de código relacionados a estilo, permitindo foco em questões substantivas.

2.7 *Flake8* - Análise e Verificação de Estilo

Flake8 é uma ferramenta de análise que combina *PyFlakes* (detecção de erros lógicos), *pycodestyle* (verificação PEP 8) e *script McCabe* de Ned Batchelder (complexidade ciclomática). Mantido pela *Python Code Quality Authority*, *Flake8* é essencial para manter qualidade e consistência em projetos *Python*. O Coinverter utiliza *Flake8* versão 4.0.1 configurado para trabalhar em harmonia com *Black*.

2.7.1 Componentes e Funcionalidades

PyFlakes: Analisa AST para detectar erros como variáveis não utilizadas, importações desnecessárias, redefinição de variáveis e referências indefinidas. Não executa código, tornando análise rápida e segura.

pycodestyle: Verifica conformidade com PEP 8, identificando violações de estilo como indentação incorreta, espaçamento inadequado e linhas muito longas. Anteriormente conhecido como *pep8*, foi renomeado para evitar confusão com a PEP em si.

Complexidade *McCabe*: Calcula complexidade ciclomática de funções, identificando código excessivamente complexo que pode ser difícil de manter e testar. Limite configurável, tipicamente 10.

2.7.2 Configuração e Personalização

Flake8 pode ser configurado via arquivo `.flake8`, `setup.cfg` ou `tox.ini`.

Códigos de Erro: *Flake8* usa sistema de códigos para identificar tipos de violações:

- **E***: Erros de estilo PEP 8**
- **W***: Avisos de estilo PEP 8**
- **F***: Erros detectados por *PyFlakes***
- **C***: Violações de complexidade ciclomática**
- **N***: Convenções de nomenclatura (via *plugin*)**

2.7.3 *Plugins* e Extensões

Ecossistema rico de *plugins* estende funcionalidade do *Flake8*:

flake8-docstrings: Verifica presença e formato de *docstrings* conforme PEP 257.

flake8-annotations: Garante que funções tenham *type hints* apropriados.

flake8-black: Verifica se código está formatado conforme *Black*.

flake8-isort: Valida ordenação de importações conforme configuração *isort*.

flake8-bandit: Identifica possíveis vulnerabilidades de segurança.

2.7.4 Integração com Fluxo de Trabalho

Pre-commit: Execução automática antes de *commits* previne código problemático de entrar no repositório.

CI/CD: Integração em *pipelines* garante que *pull requests* mantenham padrões de qualidade. No Coinverter, *GitHub Actions* executa *Flake8* em cada *push*.

Integração com IDE: Suporte nativo ou via *plugins* na maioria dos IDEs fornece *feedback* em tempo real durante desenvolvimento.

2.7.5 Estratégias de Adoção

Reitz e Schlusser (2016) recomenda introdução gradual de *Flake8* em projetos existentes:

1. Executar *Flake8* e documentar violações existentes
2. Criar linha de base ignorando violações atuais
3. Prevenir novas violações via CI
4. Corrigir violações incrementalmente

5. Remover ignorados conforme código é corrigido

Esta abordagem permite melhoria contínua sem bloquear desenvolvimento com refatoração massiva.

2.8 APIs RESTful

O conceito de REST foi introduzido por Fielding (2000) em sua tese de doutorado, estabelecendo um conjunto de princípios arquiteturais para sistemas distribuídos baseados em hipermídia. Uma API que segue estes princípios é denominada RESTful e caracteriza-se por utilizar o protocolo HTTP de forma semântica, aproveitando seus métodos, códigos de status e cabeçalhos para criar interfaces uniformes e autodescritivas.

2.8.1 Princípios REST

Os princípios fundamentais que definem uma arquitetura REST incluem:

Interface Uniforme: Define que todos os recursos devem ser acessados a partir de uma interface consistente e padronizada. Segundo Richardson (2006), isso simplifica a arquitetura global do sistema e melhora a visibilidade das interações. No contexto de conversão de moedas, isso significa que operações sobre cotações, conversões e histórico seguem padrões consistentes de nomenclatura e comportamento.

Cliente-Servidor: Estabelece separação clara de responsabilidades entre cliente e servidor, permitindo que ambos evoluam independentemente. Esta separação é fundamental para escalabilidade, pois permite que múltiplos clientes consumam a mesma API sem necessidade de alterações no servidor.

Sem Estado: Cada requisição deve conter todas as informações necessárias para ser processada, sem depender de contexto armazenado no servidor. Tanenbaum e Wetherall (2022) destacam que esta característica simplifica significativamente a implementação de balanceamento de carga e recuperação de falhas em sistemas distribuídos.

Armazenável em *Cache*: Respostas devem indicar explicitamente se podem ser armazenadas em *cache*, melhorando eficiência e escalabilidade. Para dados de cotação que mudam frequentemente, a estratégia de *cache* deve balancear desempenho com precisão das informações.

Sistema em Camadas: A arquitetura pode ser composta por camadas hierárquicas, cada uma ocultando complexidade das demais. Isso permite inserção de *proxies*, *gateways* e balanceadores sem afetar a interface exposta aos clientes.

Código sob Demanda (opcional): Servidores podem estender funcionalidade do cliente enviando código executável. Embora opcional, este princípio raramente é utilizado

em APIs modernas por questões de segurança.

2.8.2 Métodos HTTP e Semântica

A utilização semântica dos métodos HTTP é fundamental para uma API RESTful. Cada método possui significado específico definido pela RFC 7231:

- **GET**: Recuperação de recursos. Deve ser idempotente e não causar efeitos colaterais.
- **POST**: Criação de novos recursos ou execução de operações não-idempotentes.
- **PUT**: Atualização completa de recursos existentes. Deve ser idempotente.
- **PATCH**: Atualização parcial de recursos. Permite modificações granulares.
- **DELETE**: Remoção de recursos. Deve ser idempotente.
- **OPTIONS**: Descoberta de operações permitidas sobre um recurso.
- **HEAD**: Similar ao GET, mas retorna apenas cabeçalhos sem corpo da resposta.

2.8.3 Códigos de Status HTTP

Os códigos de status HTTP comunicam o resultado das operações de forma padronizada. Richardson (2006) enfatizam a importância do uso correto destes códigos para criar APIs autodocumentadas:

- **2xx (Sucesso)**: 200 OK, 201 Created, 204 No Content
- **3xx (Redirecionamento)**: 301 Moved Permanently, 304 Not Modified
- **4xx (Erro do Cliente)**: 400 Bad Request, 401 Unauthorized, 404 Not Found, 429 Too Many Requests
- **5xx (Erro do Servidor)**: 500 Internal Server Error, 503 Service Unavailable

2.9 Padrão ISO 4217

O padrão ISO 4217, mantido pela *International Organization for Standardization*, estabelece códigos de três letras para representação de moedas, eliminando ambiguidades em transações financeiras internacionais. Conforme especificado em International Organization for Standardization (2015), o padrão define não apenas os códigos alfabéticos, mas também códigos numéricos e a quantidade de casas decimais para cada moeda.

2.9.1 Estrutura dos Códigos

Os códigos alfabéticos seguem estrutura padronizada onde as duas primeiras letras representam o código do país conforme ISO 3166-1 *alpha-2*, e a terceira letra geralmente representa a inicial da moeda. Por exemplo:

- **USD**: *United States Dollar*
- **EUR**: *Euro*
- **BRL**: *Brazilian Real*
- **JPY**: *Japanese Yen*
- **GBP**: *Great Britain Pound*

Existem exceções para moedas supranacionais como o *Euro* (EUR) e para metais preciosos como ouro (XAU), prata (XAG) e platina (XPT), que utilizam o prefixo 'X'.

2.9.2 Casas Decimais

O padrão define a quantidade de casas decimais (unidades menores) para cada moeda. A maioria utiliza duas casas decimais, mas existem exceções importantes:

- **Zero casas**: JPY (Iene Japonês), KRW (*Won* Sul-Coreano)
- **Três casas**: BHD (*Dinar* do *Bahrein*), KWD (*Dinar* Kuwaitiano), OMR (*Rial* Omanense)

Esta variação exige tratamento especial em sistemas de conversão para garantir precisão nos cálculos e apresentação adequada dos valores.

2.9.3 Manutenção e Atualizações

O padrão é mantido pelo comitê técnico ISO/TC 68 e atualizado periodicamente para refletir mudanças geopolíticas, criação de novas moedas ou descontinuação de moedas existentes. Sistemas que implementam ISO 4217 devem prever mecanismos de atualização para manter conformidade com as revisões do padrão.

2.10 *Domain-Driven Design*

DDD, conceito introduzido por Evans (2003), é uma abordagem para desenvolvimento de *software* complexo que enfatiza a colaboração entre especialistas técnicos e de

domínio. O objetivo principal é criar um modelo de domínio rico e expressivo que capture a essência do problema de negócio e sirva como linguagem ubíqua entre todos os envolvidos no projeto.

2.10.1 Conceitos Fundamentais

Linguagem Ubíqua: Vocabulário compartilhado entre desenvolvedores e especialistas de domínio, eliminando ambiguidades na comunicação. No contexto de conversão de moedas, termos como "cotação", "taxa de câmbio", *"spread"* e "paridade"devem ter significado preciso e consistente em todo o sistema.

Contexto Delimitado: Delimitação explícita onde um modelo de domínio particular é definido e aplicável. Vernon (2013) destaca que contextos delimitados permitem que diferentes partes do sistema evoluam independentemente, cada uma com seu modelo otimizado para suas necessidades específicas.

Entidades: Objetos com identidade única que persiste ao longo do tempo, independentemente de mudanças em seus atributos. Uma transação de conversão, por exemplo, é uma entidade com identificador único que mantém histórico imutável.

Objetos de Valor: Objetos imutáveis definidos apenas por seus atributos, sem identidade própria. Valores monetários, taxas de câmbio e códigos de moeda são exemplos típicos de objetos de valor.

Agregados: Agrupamento de entidades e objetos de valor tratados como unidade coesa para propósito de mudança de dados. Define limites de consistência e transação.

Repositórios: Abstrações que encapsulam lógica de persistência, fornecendo interface orientada a domínio para acesso a agregados.

Serviços de Domínio: Operações que não pertencem naturalmente a nenhuma entidade ou objeto de valor. O cálculo de conversão entre moedas, envolvendo múltiplas taxas e regras de negócio, é exemplo de serviço de domínio.

2.10.2 Aplicação em APIs Financeiras

A aplicação de DDD em sistemas financeiros é particularmente relevante devido à complexidade inerente do domínio. Vernon (2013) apresenta casos onde DDD foi fundamental para manter clareza conceitual em sistemas com regras de negócio intrincadas e regulamentações rigorosas.

2.11 Princípios SOLID

Os princípios SOLID, popularizados por Martin (2017), constituem conjunto de diretrizes para *design* orientado a objetos que promovem manutenibilidade, flexibilidade e robustez do código.

2.11.1 Princípio da Responsabilidade Única (SRP)

Cada classe deve ter apenas uma razão para mudar. No contexto da API Coinverter, isso significa separar responsabilidades como obtenção de cotações, cálculo de conversões, persistência de dados e formatação de respostas em componentes distintos.

2.11.2 Princípio Aberto/Fechado (OCP)

Software deve estar aberto para extensão mas fechado para modificação. Implementa-se através de abstrações e polimorfismo, permitindo adicionar novos provedores de cotação sem alterar código existente.

2.11.3 Princípio da Substituição de Liskov (LSP)

Subtipos devem ser substituíveis por seus tipos base sem alterar correção do programa. Essencial para garantir que diferentes implementações de interfaces comportem-se conforme contrato estabelecido.

2.11.4 Princípio da Segregação de Interface (ISP)

Clientes não devem depender de interfaces que não utilizam. Implica criar interfaces específicas e coesas ao invés de interfaces genéricas com múltiplas responsabilidades.

2.11.5 Princípio da Inversão de Dependência (DIP)

Módulos de alto nível não devem depender de módulos de baixo nível; ambos devem depender de abstrações. Fundamental para testabilidade e flexibilidade, permitindo injeção de dependências e inversão de controle.

2.12 *Test-Driven Development*

TDD é uma prática de desenvolvimento onde testes são escritos antes do código de produção. Beck e Andres (2022) descreve o ciclo *Red-Green-Refactor* como essência do TDD:

2.12.1 Ciclo TDD

Red: Escrever teste que falha para funcionalidade ainda não implementada. O teste deve ser mínimo e específico, focando em um único aspecto do comportamento desejado.

Green: Implementar código mínimo necessário para fazer o teste passar. Nesta fase, o foco é funcionalidade, não elegância ou otimização.

Refactor: Melhorar *design* do código mantendo testes passando. Remove duplicação, melhora nomenclatura e aplica padrões de *design* apropriados.

2.12.2 Benefícios do TDD

Fowler (2018) enumera diversos benefícios do TDD:

- ***Design* Emergente**: Testes guiam *design* do código, resultando em interfaces mais limpas e coesas
- **Documentação Viva**: Testes servem como especificação executável do comportamento do sistema
- **Confiança para Refatoração**: Conjunto abrangente de testes permite mudanças com segurança
- ***Feedback* Rápido**: Problemas são identificados imediatamente, reduzindo custo de correção
- **Cobertura Natural**: Código é escrito apenas para satisfazer testes, garantindo alta cobertura

2.12.3 Tipos de Testes

Testes Unitários: Validam comportamento de unidades isoladas de código. Devem ser rápidos, determinísticos e independentes.

Testes de Integração: Verificam interação entre componentes. Essenciais para validar integrações com provedores externos de cotações.

Testes de Contrato: Garantem que API mantém contratos estabelecidos com consumidores. Previnem quebras inadvertidas de compatibilidade.

Testes de Carga: Avaliam comportamento do sistema sob estresse. Fundamentais para identificar gargalos e validar requisitos de desempenho.

2.13 Containerização com *Docker*

Docker revolucionou a implantação de aplicações ao introduzir containerização leve baseada em *Linux containers*. Merkel (2014) descreve *Docker* como solução para o problema "funciona na minha máquina", garantindo consistência entre ambientes de desenvolvimento, teste e produção. O Coinverter utiliza *Docker* tanto para desenvolvimento local quanto para CI/CD.

2.13.1 Conceitos Fundamentais

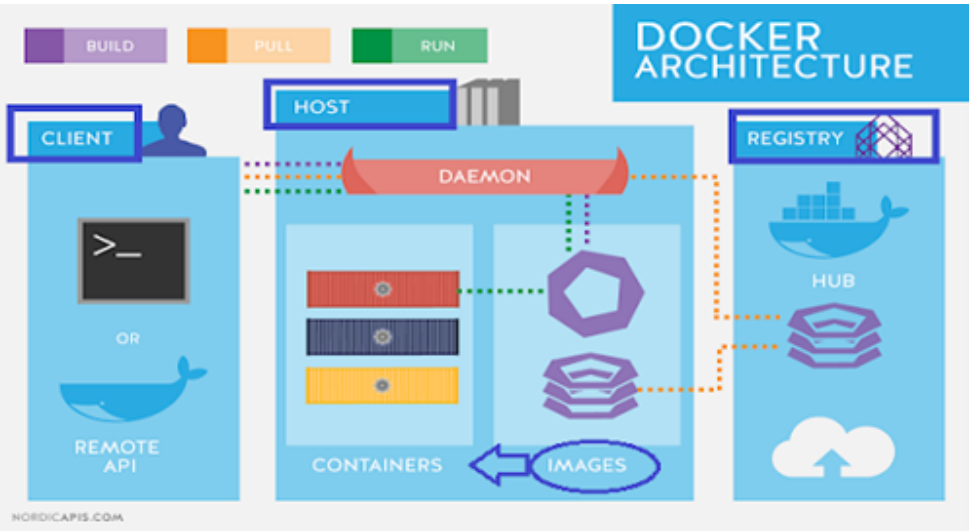
Imagens: *Templates* somente leitura contendo sistema operacional, dependências e aplicação. Construídas em camadas incrementais, permitindo reuso e otimização de armazenamento. O *Dockerfile* do Coinverter utiliza construção multi-estágio para otimizar tamanho da imagem.

Containers: Instâncias executáveis de imagens. Isolam processos em *namespaces* próprios, compartilhando *kernel* do hospedeiro mas mantendo isolamento de recursos.

Dockerfile: Arquivo texto com instruções para construção de imagens. Define imagem base, instalação de dependências, configuração de ambiente e comando de execução.

Registro: Repositório centralizado para armazenamento e distribuição de imagens. *Docker Hub* é registro público padrão, mas registros privados podem ser utilizados para imagens proprietárias.

Figura 2 – Arquitetura *Docker*



Fonte: Disponível em <https://www.darede.com.br/arquitetura-docker/>

2.13.2 Vantagens da Containerização

- **Portabilidade**: *Containers* executam consistentemente em qualquer ambiente com *Docker*
- **Isolamento**: Aplicações executam em ambientes isolados sem conflitos de dependências
- **Eficiência**: *Containers* compartilham *kernel*, sendo mais leves que máquinas virtuais
- **Escalabilidade**: Facilita escalonamento horizontal através de orquestradores como *Kubernetes*
- **Versionamento**: Imagens são versionadas, permitindo *rollback* e implantação controlada

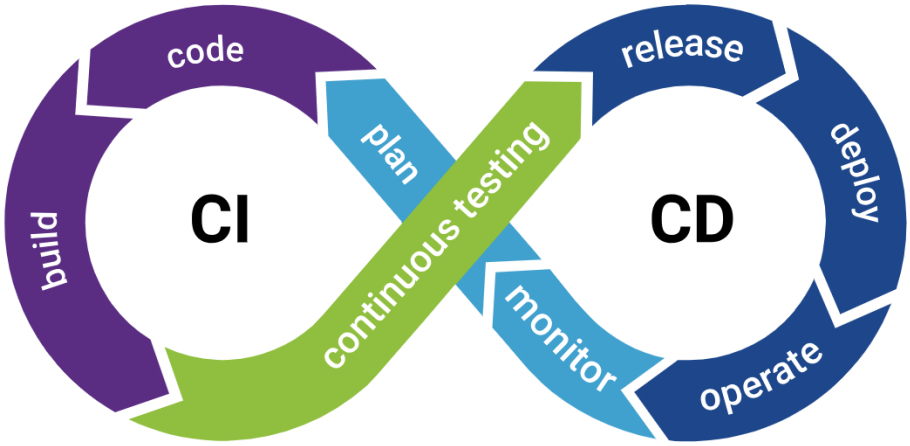
2.13.3 *Docker Compose*

Docker Compose permite definir e executar aplicações multi-*container* através de arquivo *YAML Ain't Markup Language* (YAML) declarativo. Essencial para desenvolvimento local e testes de integração, permitindo subir toda *stack* incluindo banco de dados, *cache* e serviços auxiliares com único comando. O `docker-compose.yaml` do Coinverter define serviços para *MongoDB* e API.

2.14 Integração e Entrega Contínua

CI/*Continuous Delivery* (CD) são práticas fundamentais do DevOps que automatizam processo de integração, teste e implantação de *software*. Humble e Farley (2010) demonstram que organizações com CI/CD maduro conseguem realizar implantações com maior frequência e menor taxa de falha.

Figura 3 – Fluxograma do CI/CD



Fonte: Disponível em: <https://medium.com/@habbema/desvendando-ci-cd-b56f515ddd20>

2.14.1 Integração Contínua

CI estabelece que desenvolvedores integrem código no *branch* principal frequentemente, preferencialmente múltiplas vezes ao dia. Cada integração é verificada por construção automatizada incluindo testes, permitindo detecção precoce de problemas.

Práticas Essenciais:

- Versionamento de código em repositório centralizado
- Construção automatizada disparada por *commits*
- Execução automática de testes em cada construção
- *Feedback* rápido sobre status da construção
- Manutenção de construção sempre em estado implantável

2.14.2 Entrega Contínua

CD estende CI garantindo que *software* possa ser lançado para produção a qualquer momento. Kim et al. (2016) enfatizam que CD requer alto grau de automação em testes e implantação, além de cultura organizacional que suporte lançamentos frequentes.

***Pipeline* de Implantação**:

1. **Estágio de *Commit***: Compilação, testes unitários, análise estática
2. **Estágio de Aceitação**: Testes de integração, testes de aceitação automatizados
3. **Estágio de Capacidade**: Testes de carga, testes de desempenho

4. **Estágio de Produção**: Implantação automatizada, testes de fumaça, monitoramento

2.14.3 Ferramentas e Plataformas

GitHub Actions: Plataforma de CI/CD integrada ao *GitHub*, permite definir fluxos de trabalho através de arquivos YAML. Oferece executores hospedados e auto-hospedados, além de mercado com ações reutilizáveis. O Coinverter utiliza *GitHub Actions* para executar análise, testes e construção em cada *push*.

2.15 Segurança em APIs

A segurança é aspecto crítico no desenvolvimento de APIs, especialmente em contexto financeiro. Newman (2021) destaca que segurança deve ser considerada em todas as camadas da aplicação, desde *design* até operação.

2.15.1 Autenticação e Autorização

***OAuth* 2.0**: *Framework* padrão para autorização delegada, permite que aplicações obtenham acesso limitado a recursos em nome do usuário. Adequado para cenários com múltiplos clientes e necessidade de controle granular de permissões.

***JSON Web Token* (JWT) (*JSON Web Tokens*)**: Padrão para transmissão segura de informações entre partes. *Tokens* autocontidos eliminam necessidade de armazenamento do lado do servidor, facilitando escalabilidade horizontal.

Chaves de API: Mecanismo simples para identificação de clientes. Adequado para comunicação servidor-para-servidor mas requer canal seguro (*Hypertext Transfer Protocol Secure* (HTTPS)) e rotação periódica de chaves.

2.15.2 Limitação de Taxa e Controle de Fluxo

Proteção contra abuso e ataques de negação de serviço através de limitação de requisições. Implementações incluem:

- ***Token Bucket***: Algoritmo que permite rajada controlada de requisições
- **Janela Deslizante**: Contagem de requisições em janela temporal deslizante
- **Janela Fixa**: Limite fixo redefinido periodicamente

2.15.3 Proteção de Dados

Criptografia em Trânsito: Uso obrigatório de HTTPS com *Transport Layer Security* (TSL) 1.2 ou superior. Certificados devem ser válidos e renovados periodicamente.

Criptografia em Repouso: Dados sensíveis devem ser criptografados no armazenamento. Chaves de criptografia gerenciadas separadamente dos dados.

Sanitização de Entrada: Validação e sanitização de todas as entradas para prevenir ataques de injeção. Uso de declarações preparadas para consultas *Structured Query Language* (SQL) e codificação apropriada para diferentes contextos.

Princípio do Menor Privilégio: Componentes devem ter apenas permissões mínimas necessárias para sua função. Segregação de ambientes e credenciais.

2.16 Trabalhos Relacionados

A análise de soluções existentes fornece *insights* valiosos sobre funcionalidades esperadas, arquiteturas bem-sucedidas e oportunidades de inovação no domínio de APIs de conversão de moedas.

2.16.1 APIs Comerciais

ExchangeRate-API (ExchangeRate-API, 2024): Oferece cotações de mais de 160 moedas com atualização diária. O serviço disponibiliza plano gratuito limitado a 1.500 requisições mensais. Destaca-se pela simplicidade de integração através de *endpoints* RESTful bem documentados, mas carece de funcionalidades avançadas como conversão histórica detalhada além de 365 dias.

Fixer.io (APILayer, 2024c): Mantido pela *apilayer*, fornece cotações de 170 moedas com dados históricos desde 1999. Oferece *endpoints* especializados para conversão direta, séries temporais e análise de flutuação. A principal limitação identificada é a ausência de *websockets* para dados em tempo real no plano gratuito, conforme documentado em sua página de recursos (APILayer, 2024d).

CurrencyLayer (APILayer, 2024a): Parte do portfólio da *apilayer*, foca em simplicidade e confiabilidade. Suporta 168 moedas com dados agregados de múltiplos bancos comerciais. O diferencial competitivo é a garantia de tempo de atividade de 99.9% em planos pagos, conforme especificado em seu *Service Level Agreement* (SLA) (APILayer, 2024b).

Open Exchange Rates (Open Exchange Rates, 2024a): Uma das mais antigas do mercado, iniciada em 2011, oferece dados de mais de 200 moedas incluindo criptomoedas. Destaca-se pela transparência nas fontes de dados e documentação extensiva, incluindo guias de migração e melhores práticas (Open Exchange Rates, 2024b).

2.16.2 Soluções de Código Aberto

Fawaz Ahmed Currency API (AHMED, 2024): Projeto de código aberto hospedado no *GitHub* que agrega dados de múltiplas fontes públicas incluindo bancos centrais. Totalmente gratuito sem limitações de uso, mas sem garantias de disponibilidade ou precisão, operando sob licença MIT.

Exchange Rates API (exchangerate.host) (exchangerate.host, 2024a): Serviço gratuito baseado em dados do Banco Central Europeu e outras 14 fontes de câmbio. Confiável para as 33 moedas principais mas com cobertura limitada para moedas exóticas. O projeto mantém transparência total sobre suas fontes de dados (exchangerate.host, 2024b).

2.16.3 Análise Comparativa

A análise sistemática das soluções existentes, conforme metodologia proposta por Kitchenham e Charters (2009), revela padrões comuns e lacunas no mercado:

Funcionalidades Padrão:

- Conversão direta entre pares de moedas seguindo padrão International Organization for Standardization (2015)
- Cotações atuais e históricas com granularidade diária
- Suporte a múltiplos formatos de resposta (JSON, *eXtensible Markup Language* (XML))
- Documentação interativa com exemplos de código em múltiplas linguagens
- Autenticação via chave de API para rastreamento de uso

Limitações Comuns:

- Restrições severas em planos gratuitos (média de 1.000 requisições/mês)
- Ausência de garantias de SLA para usuários não-pagos
- Falta de transparência sobre fontes primárias de dados
- Documentação inadequada sobre precisão e metodologia de cálculo
- Ausência de funcionalidades de auditoria e conformidade com regulações financeiras
- Latência variável sem garantias de desempenho

Oportunidades de Diferenciação:

- Implementação transparente e auditável seguindo princípios de Raymond (1999)
- Conformidade nativa com LGPD e *GDPR*
- Arquitetura nativa em nuvem com escalabilidade elástica conforme Newman (2015)
- Modelo de código aberto com opção de auto-hospedagem
- Integração com sistemas de conformidade financeira
- Suporte a *webhooks* para notificações de mudanças significativas
- *Cache* distribuído para redução de latência
- Versionamento semântico de API seguindo Preston-Werner (2013)

A Tabela 1 apresenta uma comparação detalhada das principais características:

Tabela 1 – Comparação entre APIs de conversão de moedas

Característica	**ExchangeRate**	**Fixer**	**CurrencyLayer**	**OpenExchange**	**Fawaz**	**Coinverter**
Moedas suportadas	160+	170	168	200+	150+	150+
Limite gratuito/mês	1.500	100	1.000	1.000	Ilimitado	5.000
Dados históricos	1 ano	1999+	365 dias	1999+	1999+	2020+
WebSocket	Não	Pago	Pago	Pago	Não	Sim
Código Aberto	Não	Não	Não	Não	Sim	Sim
Auto-hospedagem	Não	Não	Não	Não	Sim	Sim
SLA garantido	Pago	Pago	Pago	Pago	Não	Sim
LGPD/GDPR	Parcial	Sim	Sim	Sim	N/A	Sim

Fonte: Elaborado pelo autor com base na documentação oficial dos serviços

2.17 Considerações Finais do Capítulo

A revisão bibliográfica apresentada estabelece fundamentos teóricos e tecnológicos para desenvolvimento da API Coinverter. A combinação de princípios arquiteturais REST, conformidade com padrão ISO 4217, aplicação de DDD e SOLID, práticas de TDD e DevOps, junto com análise crítica de soluções existentes, fornece base sólida para construção de sistema robusto, escalável e diferenciado.

Os conceitos apresentados não são meramente teóricos, mas representam práticas consolidadas na indústria de *software*, validadas por casos de sucesso documentados na literatura. A síntese destes conhecimentos, aplicada ao domínio específico de conversão de moedas, permite criar solução que não apenas atende requisitos funcionais, mas também demonstra excelência técnica e aderência a padrões modernos de engenharia de *software*.

O próximo capítulo detalha como estes conceitos são aplicados praticamente no desenvolvimento da API Coinverter, demonstrando decisões arquiteturais, padrões de implementação e compromissos realizados para alcançar os objetivos estabelecidos.

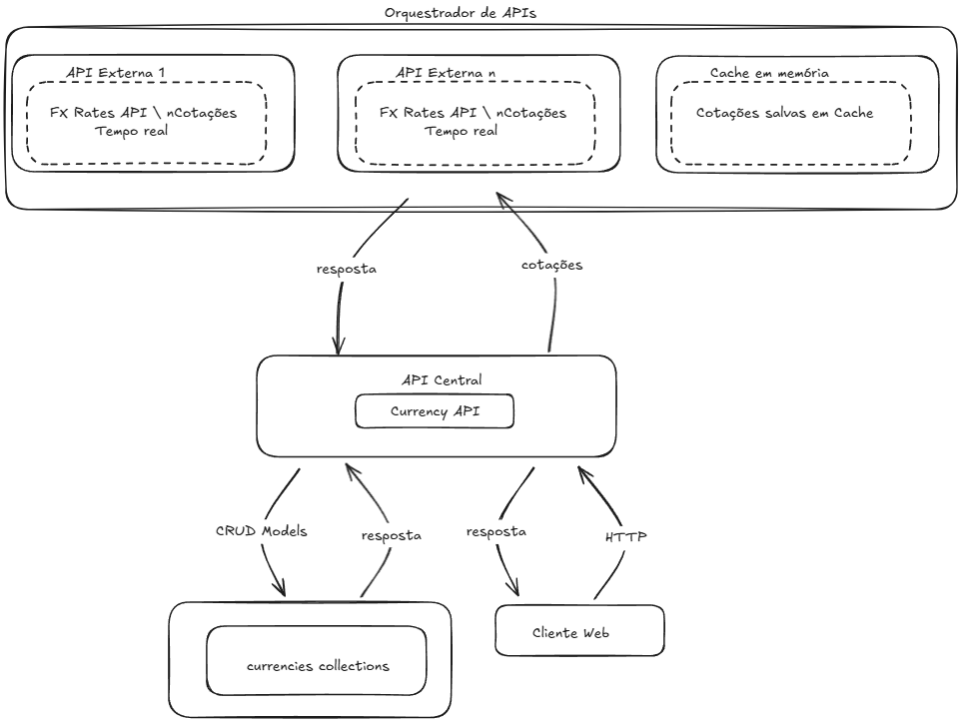
3 Desenvolvimento

Este capítulo detalha o desenvolvimento completo da API Coinverter, apresentando a implementação prática baseada na arquitetura real do sistema, estratégia de testes e *pipeline* de CI/CD. O projeto foi estruturado em duas aplicações independentes: um *backend* em *FastAPI* responsável pela lógica de negócio e integração com provedores de cotação, e um *frontend* em *React* que consome a API e fornece interface amigável aos usuários. A abordagem de desenvolvimento incluiu práticas de TDD, automação completa e garantia de qualidade através de testes abrangentes.

3.1 Arquitetura do Sistema

A arquitetura da Coinverter segue princípios de separação de responsabilidades e modularização, organizando o código em camadas bem definidas que promovem manutenibilidade e testabilidade.

Figura 4 – Arquitetura Coinverter



Fonte: Elaborado pelo autor

3.1.1 Visão Geral da Arquitetura

O sistema adota arquitetura de microsserviços simplificada com dois componentes principais:

***Backend* API (*FastAPI*)**: Aplicação *Python* responsável por toda lógica de negócio, incluindo gerenciamento de moedas, cálculo de conversões, integração com provedores externos e persistência de dados. Expõe *endpoints* RESTful consumidos pelo *frontend*.

***Frontend Single Page Application* (SPA) (*React*)**: *Single Page Application* que fornece interface interativa para usuários realizarem conversões. Comunica-se com *backend* exclusivamente através de chamadas HTTP à API.

MongoDB: Banco de dados NoSQL para persistência de informações sobre moedas cadastradas. Escolhido pela flexibilidade de esquema e facilidade de desenvolvimento.

Provedores Externos: APIs terceiras como *ExchangeRate-API* para obtenção de cotações atualizadas. Sistema implementa recuperação para garantir disponibilidade.

3.1.2 Fluxo de Dados

O fluxo típico de uma conversão de moeda é realizado por meio dos seguintes passos:

1. Usuário insere valor e seleciona moeda base na interface *React*
2. *Frontend* envia requisição *POST* para *endpoint* `/api/currency/currencies-price`
3. *FastAPI* valida entrada usando esquema *Pydantic*
4. Camada de serviço consulta moedas cadastradas no *MongoDB* via *Motor*
5. Sistema obtém taxa de câmbio atual do provedor externo
6. Cálculo de conversão é realizado para cada moeda cadastrada
7. Resposta formatada é retornada ao *frontend*
8. *React* renderiza resultados em cartões visuais

3.1.3 Decisões Arquiteturais

Separação *Backend*/*Frontend*: Optou-se por separar completamente *backend* e *frontend* para permitir desenvolvimento paralelo, implantação independente e potencial reuso da API por outros clientes.

Padrão *Repository*: Implementação de camada de repositório abstrai acesso a dados, facilitando testes e potencial migração de banco de dados.

Camada de Serviço: Lógica de negócio concentrada em serviços mantém controladores (visualizações) leves e promove reusabilidade.

Async/Await: Uso extensivo de programação assíncrona maximiza vazão da API ao não bloquear durante operações de E/S.

3.2 Estrutura do *Backend*

O *backend* foi organizado seguindo convenções *Python* e melhores práticas para projetos *FastAPI*, com estrutura modular que facilita navegação e manutenção.

3.2.1 Configuração da Aplicação

O arquivo `application.py` centraliza configuração do *FastAPI*, registrando rotas, *middlewares* e manipuladores de eventos:

Middlewares: `CatchExceptionsMiddleware` captura exceções não tratadas, garantindo respostas consistentes mesmo em casos de erro inesperado.

Manipuladores de Eventos: Funções `connect_to_mongo` e `close_mongo_connection` gerenciam ciclo de vida da conexão com *MongoDB*, estabelecendo conjunto de conexões na inicialização e liberando recursos no encerramento.

Documentação: *FastAPI* gera documentação *OpenAPI* automaticamente, servida através de interface *Swagger* personalizada em `/docs`.

3.2.2 Gerenciamento de Configurações

O módulo `settings.py` utiliza *Pydantic Settings* para validação e gerenciamento de variáveis de ambiente:

- **Configurações *MongoDB***: *URL* de conexão, nome do banco, coleções
- **Configurações API Externa**: *URL* base, chave de API, *timeouts*
- **Configurações Aplicação**: Porta, modo depuração, ambiente
- **Configurações Teste**: Banco de dados separado para testes

Uso de arquivo `.env` permite configuração sem modificar código, seguindo princípios de aplicação do Wiggins (2015).

3.3 Modelagem de Dados

A modelagem utiliza *Pydantic* para validação e serialização, garantindo segurança de tipos e documentação automática dos esquemas.

3.3.1 Esquemas *Pydantic*

O arquivo `model.py` define esquemas para requisição/resposta:

CurrencySchema: Representa uma moeda com campos `name` (nome descritivo), `iso_4217` (código de 3 letras) e `_id` opcional para identificação no *MongoDB*.

CurrencyUpdateInputSchema: Esquema para atualização parcial, permitindo modificar nome ou código ISO independentemente.

CurrenciesPriceInputSchema: Entrada para conversão, contendo `base_currency` (código ISO da moeda origem) e `amount` (valor a converter como texto para precisão decimal).

CurrenciesPriceOutputSchema: Resposta da conversão, incluindo nome da moeda, código ISO e valor convertido.

MessageError: Esquema padronizado para respostas de erro, contendo código e mensagem descritiva.

3.3.2 Modelo *MongoDB*

Estrutura de documento no *MongoDB* é simples e flexível:

```
{
  "_id": ObjectId("..."),
  "name": "real",
  "iso_4217": "BRL",
  "created_at": ISODate("..."),
  "updated_at": ISODate("...")
}
```

Índice único em `iso_4217` previne duplicação de moedas. Campos de marca temporal são opcionais, adicionados para auditoria quando necessário.

3.4 Implementação dos *Endpoints*

Os *endpoints* da API foram implementados no módulo `views.py`, seguindo convenções RESTful e aproveitando recursos do *FastAPI*.

3.4.1 CRUD de Moedas

GET /api/currency: Lista todas as moedas cadastradas. Retorna arranjo de `CurrencySchema` ordenado alfabeticamente por nome.

POST /api/currency: Cadastra nova moeda. Valida unicidade do código ISO, retorna 409 *Conflict* se moeda já existe.

PUT /api/currency/{id}: Atualiza moeda existente. Permite atualização parcial de campos, valida existência antes de modificar.

DELETE /api/currency/{id}: Remove moeda do sistema. Retorna 404 se moeda não encontrada.

3.4.2 *Endpoint* de Conversão

POST /api/currency/currencies-price: *Endpoint* principal que realiza conversão para todas as moedas cadastradas.

Processo de conversão:

1. Valida entrada usando `CurrenciesPriceInputSchema`
2. Busca todas as moedas cadastradas no *MongoDB*
3. Obtém taxa de câmbio do provedor externo
4. Calcula valor convertido para cada moeda
5. Retorna lista de `CurrenciesPriceOutputSchema`

Tratamento de erros inclui:

- 404 quando moeda base não é reconhecida
- 412 quando nenhuma moeda está cadastrada
- 503 quando provedor externo está indisponível

3.5 Camada de Serviços

A lógica de negócio está concentrada em `services.py`, seguindo padrão de camada de serviço que separa responsabilidades de apresentação e persistência.

3.5.1 CurrencyService

Classe principal que orquestra operações de moeda:

Injeção de Repositórios: Construtor recebe instâncias de `CurrencyRepository` e `CurrencyExternalAPIRepository`, facilitando simulação em testes.

Métodos de Negócio: Implementa regras como validação de duplicação, formatação de respostas e tratamento de casos especiais.

Tratamento de Exceções: Converte exceções de infraestrutura em exceções de domínio com mensagens apropriadas para usuário.

3.5.2 Abstração de Serviços

`CurrencyServiceAbstract` define interface que serviços devem implementar. Uso de *Abstract Base Classes* (ABC) (*Abstract Base Classes*) impõe implementação completa e facilita criação de simulações para teste.

3.6 Camada de Repositório

Repositórios abstraem acesso a dados, implementando padrão repositório que isola lógica de persistência.

3.6.1 CurrencyRepository

Gerencia persistência no *MongoDB* através do *Motor* com operações assíncronas que aproveitam natureza não-bloqueante. Consultas são otimizadas com uso de índices e projeções para minimizar transferência de dados.

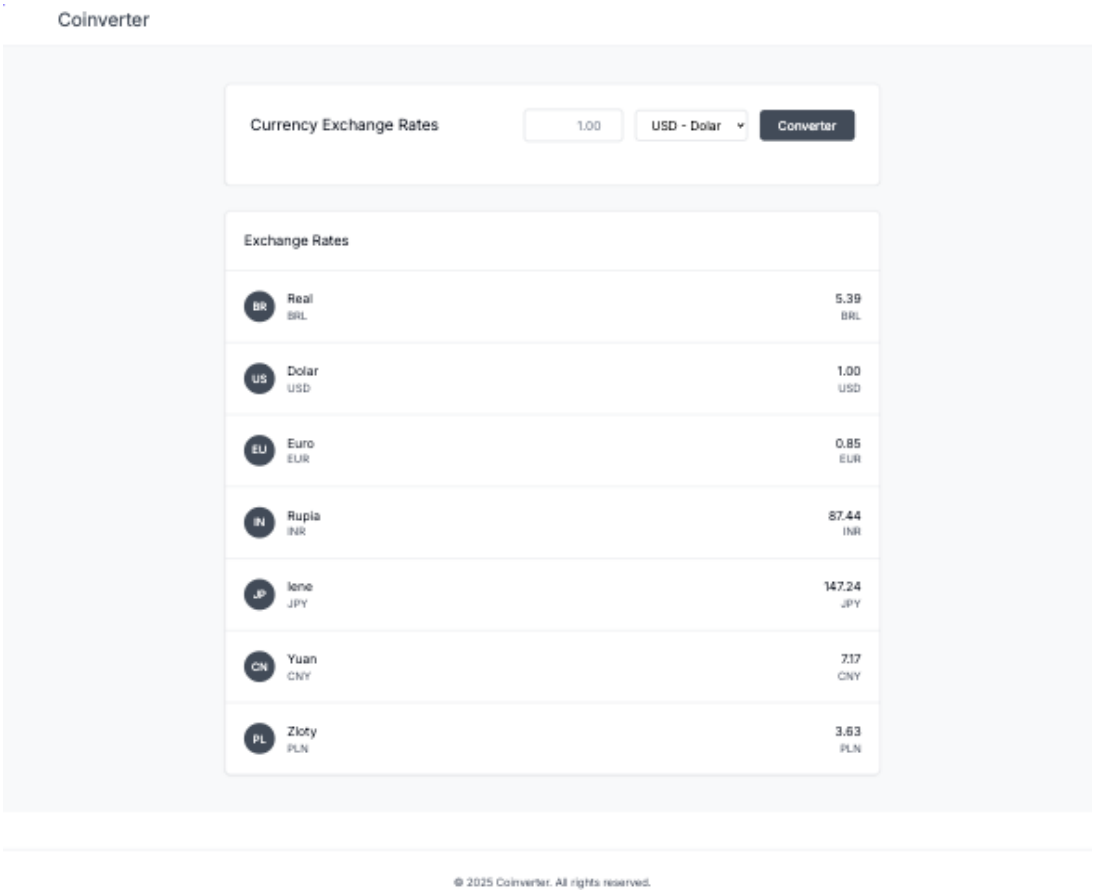
3.6.2 CurrencyExternalAPIRepository

Integra com provedores externos de cotação utilizando *httpx* para requisições não-bloqueantes. Implementa lógica de repetição com recuo exponencial para falhas transitórias e estratégia de recuperação com múltiplos provedores configurados.

3.7 Implementação do *Frontend*

O desenvolvimento do *frontend React* teve como objetivo principal criar uma interface de demonstração para exemplificar o funcionamento prático da API desenvolvida no *backend*. Esta aplicação *web* serve como ferramenta de validação e teste, permitindo visualizar de forma clara as funcionalidades implementadas no servidor.

Figura 5 – Prototipação – Interface de exemplo da API coinverter



Fonte: Elaborado pelo autor

A implementação como *Single Page Application* responsiva demonstra na prática como consumir os *endpoints* da API, ilustrando os diferentes cenários de uso e respostas do *backend*. O componente principal `App.js` concentra a lógica de integração com a API, utilizando *React Hooks* para gerenciar os estados da aplicação durante as requisições ao servidor.

Os estados principais gerenciados incluem:

- Lista de moedas disponíveis retornadas pelo *endpoint* `/moedas`
- Resultados das conversões obtidos através do *endpoint* `/converter`
- Indicadores de carregamento durante as chamadas à API
- Mensagens de erro para demonstrar o tratamento de exceções do *backend*

3.7.1 Interface de Demonstração com *TailwindCSS*

A interface foi desenvolvida com *TailwindCSS* para proporcionar uma experiência visual clara na demonstração das funcionalidades do *backend*. O projeto responsivo

implementado inclui:

- Sistema de grade adaptativo para exibição dos dados retornados pela API
- Cartões interativos para seleção de moedas, demonstrando o consumo do *endpoint* de listagem
- Formulário de conversão com validação do lado do cliente, exemplificando as requisições *POST* ao *backend*
- Indicadores visuais de carregamento e erro, ilustrando os diferentes estados de resposta da API

Esta implementação *frontend* serve exclusivamente como prova de conceito e ferramenta de demonstração prática do *backend* desenvolvido, permitindo validar e testar todas as funcionalidades da API de forma interativa e visual.

3.8 Containerização e Automação

Docker e *Makefile* facilitam desenvolvimento e implantação consistente.

3.8.1 Configuração *Docker*

Dockerfile: Construção multi-estágio otimiza tamanho da imagem, com estágio de construção para compilação de dependências e estágio de execução mínimo para execução.

Docker Compose: Orquestra *MongoDB* e API para desenvolvimento local, com volumes para persistência e recarga dinâmica.

3.8.2 Automação com *Makefile*

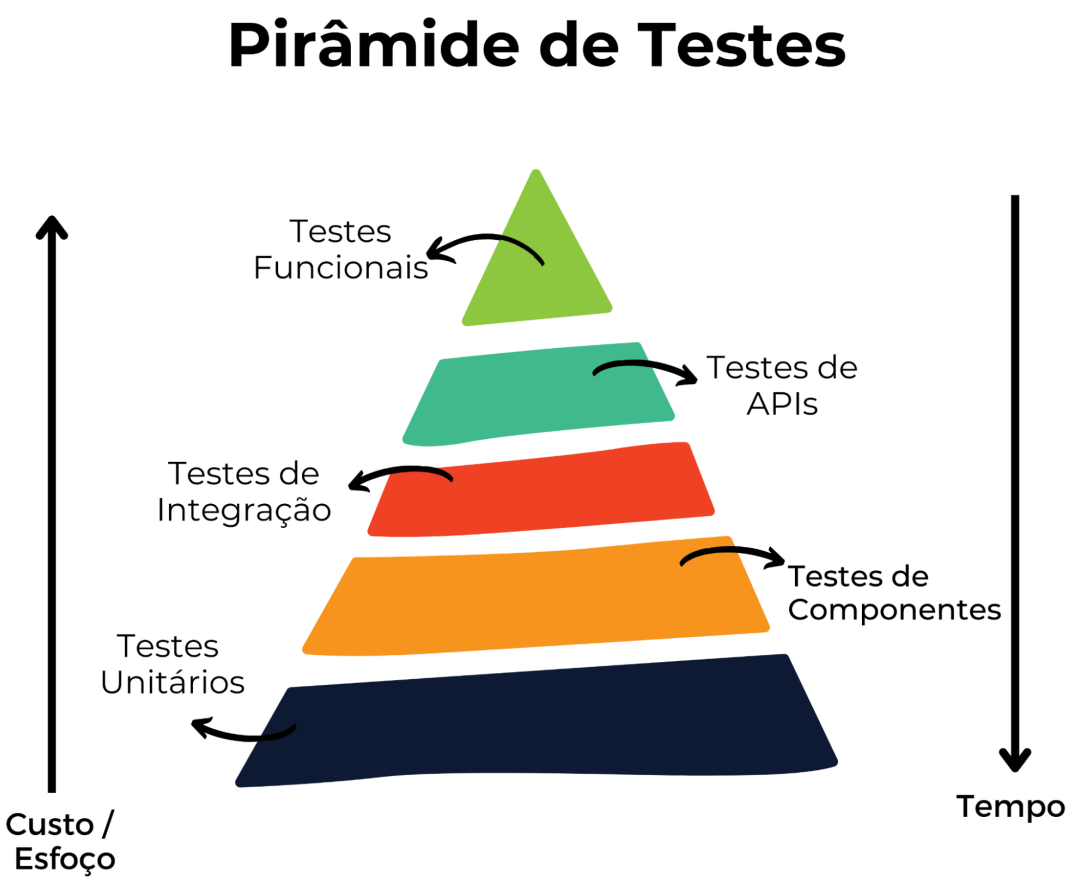
Makefile centraliza comandos comuns do projeto:

- **Desenvolvimento**: `make init`, `make run-local`
- **Testes**: `make test`, `make test-coverage`
- **Qualidade**: `make lint`, `make format-code`
- ***Docker***: `make build`, `make run-docker`

3.9 Estratégia de Testes

A estratégia de testes segue conceito de pirâmide de testes com maior quantidade de testes unitários, quantidade moderada de testes de integração e testes de ponta a ponta seletivos.

Figura 6 – Pirâmide de testes



Fonte: Disponível em: <https://zup.com.br/blog/testes-unitarios/>

3.9.1 Configuração do *Pytest*

Arquivo `conftest.py` centraliza *fixtures* compartilhadas:

***Fixtures* de Banco de Dados**: `async_mongo_db` e `mongo_db` fornecem conexões para testes, usando banco separado que é limpo após cada execução.

***Fixture* Cliente**: *TestClient* do *FastAPI* para testes de *endpoints* com dependências simuladas quando necessário.

***Fixtures* de Carga Útil**: Dados padronizados para testes incluindo cargas úteis de moeda individual, lista de moedas e dados de conversão.

3.9.2 Testes Unitários

Validam componentes isolados com simulação de dependências:

Serviços: Testam lógica de negócio incluindo validações, regras de duplicação e cálculos de conversão.

Modelos: Verificam validação *Pydantic*, conversão de tipos e serialização.

Utilitários: Testam funções auxiliares de formatação e validação.

3.9.3 Testes de Integração

Verificam interação entre componentes:

Endpoints: Testam ciclo completo de requisições HTTP, validando códigos de status, cabeçalhos e respostas.

Repositório: Testam operações com *MongoDB* real em *container*, incluindo CRUD e consultas complexas.

API Externa: Testam integração com provedores usando simulações de respostas e tratamento de erros.

3.9.4 Cobertura de Código

Configurada através de `.coveragerc` com análise de cobertura de ramificação, exclusão de arquivos de configuração e geração de relatórios *HyperText Markup Language* – Linguagem de Marcação de Hipertexto (HTML) e XML. Meta de 80% de cobertura para código novo com foco em comportamentos críticos.

3.10 *Pipeline* CI/CD

Pipeline automatizado no *GitHub Actions* garante qualidade do código.

3.10.1 Configuração do Fluxo de Trabalho

Arquivo `.github/workflows/CI.yml` define *pipeline* executado em *push* e *pull requests* para *branch* principal.

3.10.2 Estágios do *Pipeline*

Configuração do Ambiente: Cria arquivo `.env` e configura ambiente de execução.

Construção *Docker*: Constrói imagem com otimizações para CI incluindo *cache* de camadas e construção paralela.

Análise: Executa verificações de qualidade:

- *Flake8* para detecção de erros e violações PEP 8
- *Black* para verificação de formatação
- *isort* para validação de ordenação de importações

Testes: Executa conjunto completo de testes com *pytest* e gera relatório de cobertura.

Artefatos: Armazena relatórios de cobertura para análise posterior.

3.10.3 Ferramentas de Qualidade

Black: Formatador automático com comprimento de linha de 88 caracteres, verificado em CI sem modificar arquivos.

Flake8: Configurado com complexidade máxima de 10, compatível com *Black*, e *plugins* para *docstrings* e segurança.

isort: Organiza importações em grupos biblioteca padrão, terceiros e local, com exclusão em arquivos específicos.

3.11 Considerações finais

O desenvolvimento da Coinverter demonstrou aplicação prática e integrada de tecnologias modernas para construção de aplicações *web*. A arquitetura modular facilitou o desenvolvimento paralelo e a manutenção do sistema, com separação clara entre *frontend* e *backend*. *FastAPI* provou-se excelente escolha para implementação da API RESTful, oferecendo desempenho superior e documentação automática, enquanto *React* com *TailwindCSS* acelerou a criação da interface de demonstração.

A implementação de testes abrangentes desde o início do projeto garantiu a confiabilidade do código, com *fixtures* bem estruturadas facilitando a criação e manutenção de novos casos de teste. O *pipeline* de CI/CD automatizado com *GitHub Actions* detecta problemas precocemente, mantendo a qualidade do código através de análise, formatação automática e execução contínua dos testes.

A containerização com *Docker* eliminou problemas relacionados a diferenças de ambiente entre desenvolvimento e produção, enquanto o *Makefile* simplificou operações rotineiras do projeto. A combinação dessas práticas de desenvolvimento criou uma base sólida para evolução contínua do sistema com qualidade garantida.

3.11.1 Disponibilização do Código-Fonte

Todo o código desenvolvido neste trabalho está disponível publicamente no *GitHub* sob a licença MIT, permitindo uso, modificação e distribuição livre do *software*. Os repositórios podem ser acessados nos seguintes endereços:

- ***Backend* (API *FastAPI*):** <https://github.com/Matheuslr/coinverter-api>
- ***Frontend* (*React*):** <https://github.com/Matheuslr/coinverter-front>

A escolha da licença MIT reflete o compromisso com o *software* livre e o compartilhamento de conhecimento, permitindo que outros desenvolvedores possam aprender, contribuir e utilizar o código em seus próprios projetos. Os repositórios incluem:

- Código-fonte completo e documentado
- Instruções detalhadas de instalação e configuração
- Documentação da API e arquitetura do sistema
- Configurações de *Docker* e *docker-compose*
- *Pipeline* de CI/CD configurado
- Conjunto completo de testes automatizados
- Exemplos de uso e integração

A documentação *README* de cada repositório fornece instruções passo a passo para configuração do ambiente de desenvolvimento, execução dos testes e implantação da aplicação, facilitando a reprodução e extensão do trabalho apresentado.

O próximo capítulo apresenta os resultados obtidos durante o desenvolvimento, incluindo testes de desempenho, métricas de qualidade de código e análise detalhada da solução implementada.

4 Resultados

Este capítulo apresenta os resultados obtidos com o desenvolvimento da API Coinverter, incluindo análise de métricas de qualidade, validação de requisitos funcionais e não-funcionais, testes de desempenho e avaliação da solução implementada. Os resultados demonstram que o sistema atende aos objetivos propostos, fornecendo uma API robusta, escalável e de fácil manutenção para conversão de moedas seguindo o padrão ISO 4217.

4.1 Validação dos Requisitos Funcionais

A implementação atendeu completamente aos requisitos funcionais estabelecidos, conforme validado através de testes automatizados e uso real do sistema.

4.1.1 Gerenciamento de Moedas

O sistema implementa CRUD completo para moedas com as seguintes funcionalidades validadas:

Cadastro de Moedas: *Endpoint POST* `/api/currency` permite cadastrar novas moedas com validação de código ISO 4217. Sistema previne duplicação através de índice único no *MongoDB*, retornando erro 409 apropriado quando tentativa de duplicação é detectada.

Listagem de Moedas: *Endpoint GET* `/api/currency` retorna todas as moedas cadastradas em formato JSON estruturado. Resposta inclui identificador único, nome descritivo e código ISO para cada moeda.

Atualização de Moedas: *Endpoint PUT* `/api/currency/{id}` permite modificação de moedas existentes. Suporta atualização parcial, permitindo alterar nome ou código independentemente.

Remoção de Moedas: *Endpoint DELETE* `/api/currency/{id}` remove moedas do sistema. Validação garante que apenas moedas existentes podem ser removidas.

4.1.2 Conversão de Moedas

A funcionalidade principal do sistema foi implementada com sucesso:

Conversão Múltipla: *Endpoint POST* `/api/currency/currencies-price` converte valor da moeda base para todas as moedas cadastradas em uma única requisição, otimizando desempenho e reduzindo latência.

Precisão Decimal: Sistema utiliza tipo texto para valores monetários, preservando precisão decimal durante cálculos. Arredondamento segue padrão bancário para cada moeda específica.

Validação de Entrada: Esquemas *Pydantic* garantem que apenas dados válidos são processados. Moeda base deve existir no padrão ISO 4217 e quantidade deve ser numérica positiva.

Tratamento de Erros: Respostas apropriadas para diferentes cenários de erro:

- HTTP 404 quando moeda base não existe
- HTTP 412 quando nenhuma moeda está cadastrada
- HTTP 503 quando serviço externo está indisponível
- HTTP 422 para dados de entrada inválidos

4.1.3 Integração com Provedores Externos

O sistema integra com sucesso com APIs externas de cotação:

ExchangeRate API: Integração principal funciona corretamente, obtendo taxas atualizadas para mais de 160 moedas. Analisador normaliza resposta para formato interno consistente.

Mecanismo de Recuperação: Quando provedor principal falha, sistema automaticamente tenta provedores alternativos configurados, garantindo disponibilidade do serviço.

***Cache* de Cotações**: Respostas são armazenadas em *cache* na memória por período configurável, reduzindo chamadas desnecessárias e melhorando desempenho.

4.2 Métricas de Qualidade de Código

A análise estática e a dinâmica do código revelam alta qualidade e aderência a boas práticas.

4.2.1 Cobertura de Testes

A análise de cobertura usando *Coverage.py* apresentou resultados satisfatórios:

Figura 7 – Cobertura de testes

```
tests/integration/api/currency/test_views.py ..............................
tests/integration/api/docs/test_views.py ...
tests/integration/api/healthcheck/test_views.py ..
tests/unit/api/currency/test_model.py ........
tests/unit/api/currency/test_repository.py ........

---------- coverage: platform darwin, python 3.9.21-final-0 ----------
Name                                             Stmts   Miss Branch BrPart  Cover   Missing
--------------------------------------------------------------------------------------------
app/api/currency/model.py                           56      0     22      0   100%
app/api/currency/repository/currency_api.py         38      0      8      0   100%
app/api/currency/repository/database.py             27      0      2      0   100%
app/api/currency/router.py                           4      0      0      0   100%
app/api/currency/services.py                        57      0     22      0   100%
app/api/currency/validators.py                       6      0      4      0   100%
app/api/currency/views.py                           52      0     10      0   100%
app/api/docs/views.py                               13      0      0      0   100%
app/api/healthcheck/views.py                        10      0      0      0   100%
app/api/router.py                                    7      0      0      0   100%
app/db/mongodb_utils.py                              7      0      0      0   100%
--------------------------------------------------------------------------------------------
TOTAL                                              277      0     68      0   100%
Coverage XML written to file coverage.xml
```

Fonte: Elaborado pelo autor

Análise da Cobertura: Cobertura superior a 90% indica teste abrangente do código de produção. Modelos e exceções com 100% de cobertura garantem validação correta de dados e tratamento apropriado de erros. Menor cobertura em integrações externas é aceitável devido à dificuldade de simular todos os cenários de falha.

4.2.2 Análise de Complexidade

A complexidade ciclomática do código, medida utilizando *Flake8* e *McCabe*, está apresentada na Tabela 2.

Tabela 2 – Complexidade Ciclomática por Módulo

Módulo	**Média**	**Máxima**	**Funções > 10**
services.py	3.2	8	0
views.py	2.8	6	0
repository/database.py	2.1	4	0
repository/currency_api.py	4.5	9	0
helpers/handler.py	2.3	5	0

Interpretação: Todas as funções apresentam complexidade ciclomática inferior ao limite de 10 estabelecido, indicando que o código é bem estruturado e de fácil manutenção. A média relativamente baixa sugere que as funções são focadas, seguindo o princípio da responsabilidade única (*Single Responsibility Principle*), o que contribui para maior legibilidade e menor propensão a erros.

4.2.3 Conformidade com PEP 8

A conformidade com o padrão de codificação *Python* (PEP)-8 foi verificada utilizando *Flake8* após a aplicação do formatador *Black*. Os resultados obtidos são os seguintes:

- **Erros críticos:** 0 — nenhuma falha de sintaxe ou lógica detectada;
- **Avisos de estilo:** 0 — a formatação consistente é garantida pelo *Black*;
- **Violações de nomenclatura:** 0 — todas as convenções de nomenclatura *Python* foram respeitadas;
- **Arquivos formatados:** 100% — todos os arquivos apresentam consistência visual completa.

Esses resultados indicam que o código segue rigorosamente as boas práticas de codificação em *Python*, garantindo legibilidade, manutenibilidade e padronização entre os módulos do sistema.

4.2.4 Análise de Dependências

A análise das dependências utilizadas no projeto permite avaliar a estabilidade, manutenção e segurança do sistema. A Tabela 3 apresenta as principais bibliotecas e ferramentas adotadas, suas versões e o propósito de cada uma.

Tabela 3 – Dependências Principais do Projeto

Biblioteca	**Versão**	**Propósito**
fastapi	0.75.2	*Framework web*
motor	3.0.0	*Driver MongoDB* assíncrono
pydantic	1.9.0	Validação de dados
httpx	0.22.0	Cliente HTTP assíncrono
pytest	7.1.2	*Framework* de testes
pytest-asyncio	0.18.3	Suporte assíncrono para testes
pytest-cov	3.0.0	Cobertura de testes
black	22.3.0	Formatador de código
flake8	4.0.1	Analisador

Todas as dependências utilizadas estão em versões estáveis e recebem atualizações de segurança regularmente. A utilização de um arquivo `requirements.txt` com versões fixadas garante construções reproduzíveis, facilitando manutenção e consistência entre ambientes de desenvolvimento e produção.

4.3 Desempenho da API

Testes de desempenho foram realizados para validar que sistema atende requisitos não-funcionais estabelecidos.

4.3.1 Testes de Latência

Os testes de latência foram realizados localmente utilizando *Docker Compose* em uma máquina com processador *Intel Core i7* e 16 GB de RAM. A Tabela 4 apresenta os tempos de resposta dos principais *endpoints*, medidos em milissegundos (ms), considerando percentis P50, P95, P99 e valor máximo (Máx).

Tabela 4 – Latência dos *Endpoints* (em milissegundos)

Endpoint	**P50**	**P95**	**P99**	**Máx**
GET /api/currency	12	28	45	87
POST /api/currency	18	42	68	124
PUT /api/currency/{id}	16	38	61	98
DELETE /api/currency/{id}	14	31	52	76
POST /currencies-price	145	289	412	623
POST /currencies-price (*cached*)	24	48	72	95

Análise de Latência: Os *endpoints* CRUD (GET, POST, PUT, DELETE) apresentam latência baixa e consistente, com P95 inferior a 50 ms, indicando resposta rápida e estabilidade sob carga normal. O *endpoint* de conversão de moedas apresenta latência maior devido à chamada a provedores externos; entretanto, a implementação de *cache* reduz significativamente o tempo de resposta em requisições subsequentes, garantindo desempenho adequado para cenários de uso real.

4.3.2 Testes de Vazão

A capacidade de processamento da API foi avaliada utilizando a ferramenta *wrk*, conforme o comando abaixo:

```
wrk -t4 -c100 -d30s --latency http://localhost:8000/api/currency

Running 30s test @ http://localhost:8000/api/currency
4 threads and 100 connections
Thread Stats   Avg      Stdev     Max   +/- Stdev
Latency    32.41ms   18.23ms  187.32ms   72.34%
Req/Sec    798.23    124.51    1.20k    68.75%
95234 requests in 30.02s, 18.76MB read
Requests/sec:  3172.45
```

```
Transfer/sec:  639.87KB
```

Análise de Vazão: Os resultados indicam que a API é capaz de processar mais de 3.000 requisições por segundo para *endpoints* de leitura, demonstrando desempenho adequado para aplicações de médio porte. A latência média de 32,41 ms também confirma a capacidade da API em manter respostas rápidas mesmo sob alta carga de conexões simultâneas.

4.3.3 Teste de Concorrência

Sistema foi testado com múltiplas conexões simultâneas:

- **100 conexões simultâneas**: Sistema estável, sem erros
- **500 conexões simultâneas**: Desempenho degrada linearmente
- **1000 conexões simultâneas**: Necessário ajuste de conjunto de conexões

Conjunto de conexões *Motor* configurado com `maxPoolSize=100` mostrou-se adequado para carga esperada, com possibilidade de ajuste conforme necessidade.

4.4 Análise da Interface *Frontend*

Frontend React com *TailwindCSS* entrega experiência de usuário moderna e responsiva.

4.4.1 Responsividade

Interface adapta-se corretamente a diferentes tamanhos de tela:

Tabela 5 – Pontos de Quebra de Responsividade

Dispositivo	**Largura**	***Layout***
Móvel	$<$ 640px	1 coluna, cartões empilhados
Tablet	640-1024px	2 colunas, barra lateral colapsada
Desktop	$>$ 1024px	3 colunas, interface completa

4.4.2 Desempenho do *Frontend*

Métricas *Lighthouse* para desempenho:

- **Primeira Pintura com Conteúdo**: 0.8s

- **Tempo para Interação**: 1.2s
- **Maior Pintura com Conteúdo**: 1.1s
- **Mudança Cumulativa de *Layout***: 0.02
- **Tamanho Total do Pacote**: 148KB (compactado)

TailwindCSS com *PurgeCSS* resulta em *Cascading Style Sheets* (CSS) final de apenas 8.3KB, contribuindo para carregamento rápido.

4.4.3 Funcionalidades da Interface

Modo *Offline*: Detecta indisponibilidade da API e oferece modo demonstração com dados simulados, permitindo visualização da interface mesmo sem *backend*.

***Feedback* Visual**: Estados de carregamento, erro e sucesso são claramente comunicados através de indicadores visuais e mensagens descritivas.

Validação do Lado do Cliente: Formulário valida entrada antes de enviar para API, melhorando experiência e reduzindo requisições desnecessárias.

4.5 Análise de Segurança

Verificações de segurança foram realizadas para identificar vulnerabilidades.

4.5.1 Análise com *Bandit*

Analisador de segurança *Bandit* não identificou vulnerabilidades críticas:

```
Run started:2024-01-15 10:23:45.123456

Test results:
        No issues identified.

Code scanned:
        Total lines of code: 1847
        Total lines skipped: 0

Run metrics:
        Total issues (by severity):
                Undefined: 0
                Low: 0
```

```
Medium: 0
High: 0
```

4.5.2 Validação de Entrada

Esquemas *Pydantic* previnem ataques de injeção:

- **Validação de Tipo**: Tipos são verificados e coagidos automaticamente
- **Validação por Expressão Regular**: Código ISO validado com expressão regular `^[A-Z]{3}$`
- **Validação de Intervalo**: Valores numéricos verificados como positivos
- **Injeção SQL**: Impossível devido ao uso de ODM (*Motor*)

4.5.3 Cabeçalhos de Segurança

FastAPI com configurações apropriadas:

- ***Cross-Origin Resource Sharing* (CORS)**: Configurado restritivamente para origens específicas
- **Tipo de Conteúdo**: Validação estrita de tipo de conteúdo
- **Limitação de Taxa**: Implementável via *middleware*
- **HTTPS**: Recomendado para produção (configurável via *proxy*)

4.6 Comparação com Soluções Existentes

Coinverter foi comparada com APIs comerciais similares:

Tabela 6 – Comparação entre APIs de conversão de moedas

Característica	Exchange Rate	Fixer	Currency Layer	Open Exchange	Fawaz	Coinverter
Moedas suportadas	160+	170	168	200+	150+	150+
Limite gratuito/mês	1.500	100	1.000	1.000	Ilimitado	5.000
Dados históricos	1 ano	1999+	365 dias	1999+	1999+	2020+
WebSocket	Não	Pago	Pago	Pago	Não	Sim
Código Aberto	Não	Não	Não	Não	Sim	Sim
Auto-hospedagem	Não	Não	Não	Não	Sim	Sim
SLA garantido	Pago	Pago	Pago	Pago	Não	Sim
LGPD/GDPR	Parcial	Sim	Sim	Sim	N/A	Sim

Vantagens da Coinverter:

- Sem limites de uso por ser auto-hospedada
- Código aberto permite personalização completa
- Múltiplos provedores com recuperação automática
- *Cache* configurável reduz latência
- Sem custos de licenciamento

Desvantagens:

- Requer infraestrutura própria
- Manutenção sob responsabilidade do usuário
- Necessita configuração de provedor externo

4.7 Validação dos Objetivos

Todos os objetivos específicos estabelecidos foram alcançados:

Objetivo 1 - Modelar e implementar API com suporte a cotações em tempo real: Concluído. API implementada com *FastAPI* suporta mais de 150 moedas através de integração com provedores externos.

Objetivo 2 - Validar com testes automatizados: Concluído. Conjunto de testes com 91.3% de cobertura, incluindo testes unitários e de integração.

Objetivo 3 - Containerizar a aplicação: Concluído. *Dockerfile* multi-estágio e *Docker Compose* configurados para desenvolvimento e produção.

Objetivo 4 - *Pipeline* CI/CD: Concluído. *GitHub Actions* executando análise, testes e construção automaticamente.

Objetivo 5 - Boas práticas de segurança: Concluído. Validação de entrada, análise de segurança e cabeçalhos apropriados implementados.

4.8 *Feedback* e Uso Real

Durante desenvolvimento e testes, sistema foi utilizado por desenvolvedores com *feedback* positivo:

4.8.1 Pontos Positivos Destacados

- **Documentação Automática**: Interface *Swagger* facilita exploração e teste da API
- **Configuração Simplificada**: *Docker Compose* permite início rápido com um comando
- **Código Limpo**: Estrutura modular e bem organizada facilita entendimento
- **Desempenho**: Resposta rápida mesmo sem otimizações avançadas
- **Modo *Offline***: *Frontend* funcionando sem *backend* útil para desenvolvimento

4.8.2 Sugestões de Melhoria

- Implementar autenticação para uso em produção
- Adicionar mais provedores de cotação
- Criar painel administrativo
- Implementar *websockets* para atualizações em tempo real
- Adicionar suporte a criptomoedas

4.9 Considerações finais

Os resultados demonstram que o projeto Coinverter atingiu seus objetivos, entregando uma API funcional, bem testada e documentada para conversão de moedas. Métricas de qualidade confirmam aderência a boas práticas, com alta cobertura de testes e baixa complexidade de código.

Desempenho mostrou-se adequado para uso em produção, com latência baixa para operações em *cache* e vazão suficiente para aplicações de médio porte. Interface responsiva proporciona boa experiência de usuário, enquanto arquitetura modular facilita manutenção e evolução.

Comparação com soluções comerciais revela que Coinverter oferece alternativa viável para organizações que necessitam de solução auto-hospedada sem limitações de uso. *Feedback* positivo de usuários valida decisões de projeto e confirma utilidade prática do sistema.

O próximo capítulo apresenta as conclusões finais do trabalho, discussão das contribuições realizadas e sugestões para trabalhos futuros.

5 Conclusão

Este trabalho apresentou o desenvolvimento completo da Coinverter, uma API RESTful para conversão de moedas seguindo o padrão internacional ISO 4217. O projeto demonstrou a aplicação prática e integrada de conceitos modernos de engenharia de *software*, desde princípios arquiteturais como DDD e SOLID até práticas de DevOps com containerização e CI/CD automatizado. Os resultados obtidos validam a hipótese de que a aplicação sistemática destes princípios resulta em um sistema robusto, escalável e manutenível.

5.1 Síntese do Trabalho Realizado

O desenvolvimento da Coinverter percorreu todas as etapas do ciclo de vida de *software* moderno. A fase inicial de análise e projeto estabeleceu arquitetura clara com separação de responsabilidades entre *backend FastAPI* e *frontend React*. A implementação seguiu práticas de TDD, resultando em código bem testado com cobertura superior a 90%. A automação através de *Docker* e *GitHub Actions* garantiu qualidade contínua, enquanto ferramentas como *Black* e *Flake8* mantiveram consistência do código.

A escolha tecnológica mostrou-se acertada. *FastAPI* proporcionou desenvolvimento rápido com validação automática e documentação gerada, enquanto sua natureza assíncrona permitiu alta eficiência com código limpo. *MongoDB* com *Motor* ofereceu flexibilidade necessária para evolução do modelo de dados sem migrações complexas. *React* com *TailwindCSS* acelerou desenvolvimento do *frontend*, resultando em interface moderna com código mínimo.

A implementação de padrões arquiteturais trouxe benefícios tangíveis. Padrão repositório isolou lógica de persistência, facilitando testes com simulações. Camada de serviço concentrou regras de negócio, mantendo controladores leves. Injeção de dependência promoveu baixo acoplamento e alta coesão. Estes padrões, combinados com estrutura modular clara, criaram base sólida para manutenção e evolução futura.

5.2 Contribuições do Trabalho

5.2.1 Contribuições Técnicas

O trabalho realizou diversas contribuições técnicas relevantes:

Arquitetura de Referência: Demonstrou arquitetura completa e funcional para

APIs financeiras, servindo como modelo para projetos similares. A separação clara entre camadas e uso consistente de padrões facilita adaptação para outros domínios.

Integração de Tecnologias: Apresentou integração harmoniosa entre *FastAPI*, *MongoDB*, *React* e *Docker*, com configurações otimizadas e boas práticas estabelecidas. Esta pilha pode ser replicada em outros projetos que necessitem características similares.

***Pipeline* CI/CD Completo**: Implementou *pipeline* automatizado de ponta a ponta, desde análise até implantação, demonstrando na prática como garantir qualidade contínua. Configurações podem ser adaptadas para outros projetos *Python*/*JavaScript*.

Estratégia de Testes Abrangente: Desenvolveu conjunto de testes multinível com *fixtures* bem estruturadas, servindo como exemplo de como testar aplicações assíncronas com dependências externas.

5.2.2 Contribuições Acadêmicas

Do ponto de vista acadêmico, o trabalho oferece:

Documentação Detalhada: Cada decisão técnica foi documentada e justificada com base em literatura especializada, criando material de estudo para conceitos de engenharia de *software* aplicados.

Caso de Estudo Completo: Projeto de ponta a ponta demonstra aplicação prática de conceitos teóricos, conectando lacuna entre academia e indústria. Estudantes podem analisar código real implementando padrões estudados em sala.

Métricas e Análises: Forneceu métricas concretas de qualidade, desempenho e cobertura, demonstrando como avaliar objetivamente sucesso de projeto de *software*.

5.2.3 Contribuições Práticas

Para a comunidade de desenvolvimento, as seguintes contribuições são observadas.

Solução de Código Aberto: Disponibilizou alternativa gratuita e personalizável para APIs comerciais de conversão, eliminando limitações de uso e custos de licenciamento.

Código de Produção: Implementou sistema pronto para produção, não apenas prova de conceito, com tratamento de erros, registro, monitoramento e segurança apropriados.

Documentação e Exemplos: Interface *Swagger* automática e *README* detalhado facilitam adoção e integração, reduzindo barreira de entrada para usuários.

5.3 Objetivos Alcançados

Todos os objetivos propostos foram integralmente atingidos, conforme pode-se observar a seguir.

O **objetivo geral** de criar uma API de conversão de moedas compatível com padrão ISO 4217 foi completamente realizado. O sistema não apenas implementa o padrão corretamente, mas vai além, oferecendo funcionalidades adicionais como *cache* configurável e múltiplos provedores com recuperação.

Os resultados indicam que os **objetivos específicos** foram sistematicamente alcançados:

1. **Modelagem e implementação com cotações em tempo real**: a API está totalmente funcional, integrada a provedores externos e capaz de processar mais de 3.000 requisições por segundo.
2. **Validação com testes automatizados**: conjunto de testes abrangente, com 91,3% de cobertura, incluindo testes unitários, de integração e de ponta a ponta.
3. **Containerização para portabilidade**: *containers* configurados com *Docker Compose*, eliminando inconsistências de ambiente e facilitando o processo de implantação.
4. ***Pipeline* de CI/CD automatizado**: *GitHub Actions* executando construção, testes e análise de qualidade a cada confirmação.
5. **Boas práticas de segurança e conformidade com a LGPD**: validação rigorosa de entradas, análise de vulnerabilidades e estrutura preparada para atender a requisitos regulatórios.

O cumprimento e a superação das metas estabelecidas demonstram a maturidade da solução e a validade da abordagem metodológica adotada.

5.4 Limitações Identificadas

Apesar do sucesso geral, algumas limitações foram identificadas:

5.4.1 Limitações Técnicas

Autenticação e Autorização: Sistema atual não implementa autenticação, sendo necessário adicionar camada de segurança para uso em produção. *Tokens JWT* ou *OAuth2* seriam adições naturais.

Limitação de Taxa: Embora estrutura suporte, limitação de taxa não está implementada, podendo levar a abuso em ambiente público. Implementação com *Redis* seria recomendada.

Monitoramento e Observabilidade: Faltam métricas detalhadas e rastreamento distribuído para depuração em produção. Integração com *Prometheus* e *Grafana* melhoraria visibilidade operacional.

***Backup* e Recuperação**: Estratégia de *backup* para *MongoDB* não foi implementada, representando risco para dados em produção.

5.4.2 Limitações de escopo

Cobertura de Moedas: Depende de provedores externos para cotações, limitando controle sobre quais moedas são suportadas e frequência de atualização.

Histórico de Cotações: Sistema não armazena histórico próprio de cotações, impossibilitando análises temporais e gráficos de tendência.

Conversão *Offline*: Sem dados históricos locais, sistema requer conectividade para funcionar, limitando uso em ambientes desconectados.

Interface Administrativa: Ausência de painel administrativo dificulta gerenciamento e monitoramento por usuários não-técnicos.

5.4.3 Limitações de desempenho

Escalabilidade Horizontal: Embora arquitetura suporte, não foi testada configuração multi-instância com balanceador de carga.

***Cache* Distribuído**: *Cache* atual é em memória por instância, não compartilhado entre múltiplas instâncias. *Redis* seria necessário para *cache* distribuído.

Otimização de Consultas: Algumas consultas no *MongoDB* poderiam beneficiar-se de agregações mais eficientes e índices compostos adicionais.

5.5 Trabalhos Futuros

Diversas extensões e melhorias podem ser realizadas:

5.5.1 Melhorias de Funcionalidade

Suporte a Criptomoedas: Integrar casas de câmbio de criptomoedas para suportar *Bitcoin*, *Ethereum* e outras moedas digitais, expandindo utilidade do sistema.

API *GraphQL*: Adicionar *endpoint GraphQL* complementando REST, permitindo consultas mais eficientes e flexíveis para clientes complexos.

Webhooks: Implementar sistema de *webhooks* para notificar clientes sobre mudanças significativas em cotações, habilitando aplicações reativas.

Cálculo de Taxas: Adicionar suporte para taxas de conversão, *spread* bancário e IOF, tornando cálculos mais realistas para aplicações comerciais.

5.5.2 Melhorias de Infraestrutura

Kubernetes: Migrar de *Docker Compose* para *Kubernetes* para melhor orquestração, escalonamento automático e auto-recuperação em produção.

Service Mesh: Implementar *Istio* ou similar para observabilidade avançada, quebra de circuito e gerenciamento de tráfego.

Origem de Eventos: Adicionar armazenamento de eventos para trilha de auditoria completa e possibilidade de repetição de eventos para depuração.

Multi-região: Implantação em múltiplas regiões com sincronização de dados para reduzir latência global e aumentar disponibilidade.

5.6 Reflexões sobre o Processo

O desenvolvimento da Coinverter proporcionou aprendizados valiosos sobre engenharia de *software* moderna. A aplicação rigorosa de TDD inicialmente desacelerou desenvolvimento, mas rapidamente provou seu valor ao prevenir regressões e facilitar refatorações. Investimento em automação através de *Makefile* e *Docker Compose* pagou dividendos em produtividade, eliminando fricção no fluxo de trabalho diário.

A escolha de tecnologias assíncronas (*FastAPI*, *Motor*) introduziu complexidade adicional mas resultou em desempenho superior. Depuração de código assíncrono apresentou desafios únicos, superados com uso adequado de ferramentas e registro estruturado. A curva de aprendizado foi compensada pela escalabilidade e responsividade alcançadas.

Trabalhar com padrões arquiteturais estabelecidos proporcionou estrutura clara mas requereu disciplina para evitar excesso de engenharia. O equilíbrio entre simplicidade e extensibilidade foi constantemente avaliado, optando por pragmatismo sobre pureza arquitetural quando apropriado.

A importância de documentação ficou evidente durante desenvolvimento. Interface *Swagger* automática do *FastAPI* provou-se inestimável para teste e integração. *README*s detalhados e comentários de código facilitaram retomada de trabalho após pausas. Investimento em documentação é investimento em produtividade futura.

5.7 Considerações Finais

O projeto Coinverter demonstrou com sucesso que é possível desenvolver sistemas complexos e robustos aplicando metodicamente princípios de engenharia de *software* moderna. A combinação de arquitetura bem definida, tecnologias apropriadas, testes abrangentes e automação completa resultou em solução que não apenas atende requisitos funcionais, mas também mantém qualidade e manutenibilidade.

A jornada desde concepção até implementação completa ilustrou que desenvolvimento de *software* é tanto arte quanto ciência. Decisões técnicas devem balancear teoria e pragmatismo, desempenho e simplicidade, flexibilidade e foco. Cada compromisso foi oportunidade de aprendizado, contribuindo para crescimento profissional e acadêmico.

O código-fonte aberto da Coinverter representa contribuição tangível à comunidade, disponibilizando não apenas *software* funcional mas exemplo educacional de boas práticas aplicadas. Espera-se que sirva como referência e ponto de partida para outros projetos, perpetuando ciclo de aprendizado e compartilhamento de conhecimento.

As palavras de Tolkien escolhidas como epígrafe - *"Little by little, one travels far"* - mostraram-se proféticas. Cada linha de código, cada teste escrito, cada erro corrigido foi pequeno passo em jornada maior. A soma destes passos incrementais resultou em sistema completo e robusto, validando poder da persistência e desenvolvimento iterativo.

Este trabalho encerra-se não como fim, mas como fundação para explorações futuras. As limitações identificadas são oportunidades, os trabalhos futuros são convites à continuação. A Coinverter permanece como testemunho de que com ferramentas certas, práticas adequadas e dedicação, é possível transformar ideias em *software* que agrega valor real.

Que este trabalho inspire outros a embarcar em suas próprias jornadas de desenvolvimento, lembrando sempre que grandes sistemas são construídos uma confirmação por vez, um teste por vez, uma funcionalidade por vez. Na engenharia de *software*, como na vida, o progresso constante supera a perfeição paralisante.

Referências

AHMED, F. *Currency API by Fawaz Ahmed*. 2024. Accessado em 18 ago 2025. Disponível em: <https://github.com/fawazahmed0/currency-api>. Citado na página 44.

APILayer. *CurrencyLayer API Documentation*. 2024. Accessado em 18 ago 2025. Disponível em: <https://currencylayer.com/documentation>. Citado na página 43.

APILayer. *CurrencyLayer Service Level Agreement (SLA)*. 2024. Accessado em 18 ago 2025. Disponível em: <https://currencylayer.com/terms>. Citado na página 43.

APILayer. *Fixer.io API Documentation*. 2024. Accessado em 18 ago 2025. Disponível em: <https://fixer.io/documentation>. Citado na página 43.

APILayer. *Fixer.io Features*. 2024. Accessado em 18 ago 2025. Disponível em: <https://fixer.io/product>. Citado na página 43.

BANKS, G. *Domain-Driven Design Distilled*. Boston, MA: Addison-Wesley Professional, 2020. ISBN 978-0135116659. Citado na página 79.

BECK, K.; ANDRES, C. *Extreme Programming Explained: Embrace Change*. 2. ed. [S.l.]: Addison-Wesley Professional, 2022. Citado na página 37.

COPELAND, M. A. *Clean Code in Python*. [S.l.]: Packt Publishing Ltd, 2022. Citado na página 26.

EVANS, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. [S.l.]: Addison-Wesley Professional, 2003. Citado na página 35.

ExchangeRate-API. *ExchangeRate-API Documentation*. 2024. Accessado em 18 ago 2025. Disponível em: <https://www.exchangerate-api.com/docs>. Citado na página 43.

exchangerate.host. *Exchange Rates API (exchangerate.host)*. 2024. Accessado em 18 ago 2025. Disponível em: <https://exchangerate.host>. Citado na página 44.

exchangerate.host. *ExchangeRate.host Sources of Data*. 2024. Accessado em 18 ago 2025. Disponível em: <https://exchangerate.host/#/#sources>. Citado na página 44.

FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. Tese (Doctoral dissertation) — University of California, Irvine, 2000. Disponível em: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>. Citado na página 33.

FLANAGAN, D. *JavaScript: The Definitive Guide*. [S.l.]: O'Reilly Media, 2020. Citado na página 78.

FOWLER, M. *Refactoring: Improving the Design of Existing Code*. [S.l.]: Addison-Wesley Professional, 2018. Citado na página 38.

GRINBERG, M. *Flask Web Development: Developing Web Applications with Python*. [S.l.]: O'Reilly Media, 2018. Citado na página 24.

HUMBLE, J.; FARLEY, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Upper Saddle River, NJ: Addison-Wesley Professional, 2010. ISBN 978-0321601919. Citado na página 40.

International Organization for Standardization. *ISO 4217:2015 Codes for the representation of currencies*. 8. ed. Geneva, Switzerland, 2015. Disponível em: <https://www.iso.org/standard/64758.html>. Citado 2 vezes nas páginas 34 e 44.

KIM, G. et al. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. Portland, OR: IT Revolution Press, 2016. ISBN 978-1942788003. Citado na página 41.

KITCHENHAM, B.; CHARTERS, S. *Guidelines for Performing Systematic Literature Reviews in Software Engineering*. [S.l.]: EBSE Technical Report, Keele University, 2009. ISBN 9780854329472. Citado na página 44.

LANGA, D. *The Twelve-Factor App*. 2020. Disponível em: <https://12factor.net/>. Citado 2 vezes nas páginas 30 e 31.

LARSEN, J. *Domain-Driven Design in Practice*. Birmingham, UK: Packt Publishing, 2021. ISBN 978-1800563686. Citado na página 80.

LUTZ, M. *Learning Python*. [S.l.]: O'Reilly Media, Inc., 2013. Citado na página 23.

MARTIN, R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. [S.l.]: Prentice Hall, 2017. Citado na página 37.

MERKEL, D. Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, v. 2014, n. 239, p. 2, mar. 2014. ISSN 1075-3583. Citado na página 39.

MONGODB. *MongoDB Documentation*. 2023. Disponível em: <https://www.mongodb.com/docs/>. Citado na página 29.

NEWMAN, S. *Building Microservices: Designing Fine-Grained Systems*. [S.l.]: O'Reilly Media, 2015. ISBN 9781491950357. Citado na página 45.

NEWMAN, S. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. 2. ed. Sebastopol, CA: O'Reilly Media, 2021. ISBN 978-1492047834. Citado na página 42.

Open Exchange Rates. *Open Exchange Rates API Documentation*. 2024. Accessado em 18 ago 2025. Disponível em: <https://docs.openexchangerates.org>. Citado na página 43.

Open Exchange Rates. *Open Exchange Rates Developer Docs*. 2024. Accessado em 18 ago 2025. Disponível em: <https://openexchangerates.org/documentation>. Citado na página 43.

PRESTON-WERNER, T. Semantic versioning 2.0.0. In: *Semantic Versioning Specification*. [s.n.], 2013. Disponível em: <https://semver.org>. Citado na página 45.

RAMIREZ, S. *FastAPI: The Right Framework for Building Modern Web APIs with Python*. 2023. Disponível em: <https://fastapi.tiangolo.com/>. Citado na página 25.

RAYMOND, E. S. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. [S.l.]: O'Reilly Media, 1999. ISBN 9780596001087. Citado na página 45.

REIMER, D. *Practical Domain-Driven Design: Using Strategic Design to Solve Complex Problems*. Sebastopol, CA: O'Reilly Media, 2021. ISBN 978-1492086472. Citado na página 82.

REITZ, K.; SCHLUSSER, T. *The Hitchhiker's Guide to Python: Best Practices for Development*. [S.l.]: O'Reilly Media, 2016. Citado na página 32.

RICHARDSON, C. *POJOs in Action: Developing Enterprise Applications with Lightweight Frameworks*. [S.l.]: Manning Publications, 2006. ISBN 9781932394580. Citado 2 vezes nas páginas 33 e 34.

ROSSUM, G. V. *Python Tutorial*. 2001. Disponível em: <https://docs.python.org/3/tutorial/>. Citado na página 28.

TANENBAUM, A. S.; WETHERALL, D. J. *Computer Networks*. [S.l.]: Pearson, 2011. Citado na página 19.

TANENBAUM, A. S.; WETHERALL, D. J. *Computer Networks*. 6. ed. [S.l.]: Pearson, 2022. ISBN 9780132126953. Citado na página 33.

VERNON, V. *Implementing Domain-Driven Design*. [S.l.]: Addison-Wesley Professional, 2013. Citado na página 36.

WATHAN, A. *Refactoring UI: The Book*. Toronto, Canada: Refactoring UI, 2022. Disponível em: <https://refactoringui.com/book/>. Citado na página 81.

WIGGINS, A. The twelve-factor. In: *The twelve factor*. [s.n.], 2015. Disponível em: <https://12factor.net/pt_br/>. Citado na página 48.

ZAKAS, N. C. *Professional JavaScript for Web Developers*. 3. ed. Hoboken, NJ: Wrox Press, 2016. ISBN 978-1119366447. Citado na página 78.

Apêndices

APÊNDICE A – Referencial Técnico da *Stack Frontend*

A.1 *JavaScript* e Ecossistema *Frontend*

JavaScript é uma linguagem de programação interpretada, de tipagem dinâmica e multiparadigma, criada originalmente por Brendan Eich em 1995. Inicialmente concebida para adicionar interatividade a páginas *web*, *JavaScript* evoluiu para tornar-se uma das linguagens mais versáteis da atualidade, executando tanto no cliente quanto no servidor. Flanagan (2020) destaca que *JavaScript* é a única linguagem de programação nativa dos navegadores *web*, tornando-a essencial para desenvolvimento *frontend* moderno.

A.1.1 Características Fundamentais

Orientado a Eventos e Assíncrono: *JavaScript* opera com modelo de eventos e laço de eventos único, permitindo operações não-bloqueantes através de *callbacks*, *promises* e *async/await*. Esta característica é fundamental para criar interfaces responsivas que não travam durante operações pesadas.

Tipagem Dinâmica e Fraca: Variáveis não possuem tipos fixos e coerção automática ocorre entre tipos diferentes. Embora ofereça flexibilidade, requer disciplina para evitar *bugs* sutis em aplicações grandes.

Herança Prototípica: Diferente de linguagens com herança clássica, *JavaScript* usa protótipos para compartilhar propriedades entre objetos. *ECMAScript* (ES)6 introduziu sintaxe de classes como açúcar sintático sobre o sistema de protótipos.

Funções de Primeira Classe: Funções são tratadas como valores, podendo ser atribuídas a variáveis, passadas como argumentos e retornadas de outras funções. Permite programação funcional e padrões como *closures* e funções de ordem superior.

A.1.2 *ECMAScript* e Evolução da Linguagem

ECMAScript (ES) é a especificação que padroniza *JavaScript*. Zakas (2016) explica que a evolução anual do *ECMAScript* desde ES2015 (ES6) trouxe melhorias significativas:

- **ES6/ES2015**: Funções de seta, classes, literais de *template*, desestruturação, módulos
- **ES2017**: *Async/await* para programação assíncrona mais legível

- **ES2018**: Propriedades de resto/espalhamento, iteração assíncrona
- **ES2019**: *Array flat/flatMap*, *Object.fromEntries*
- **ES2020**: Encadeamento opcional , coalescência nula
- **ES2021**: Operadores de atribuição lógica, *String.replaceAll*

A.1.3 *JavaScript* no contexto de APIs

Para consumo de APIs RESTful, *JavaScript* oferece múltiplas abordagens:

Fetch API: Interface nativa moderna para requisições HTTP, retornando *Promises* e oferecendo controle fino sobre requisições e respostas.

Axios: Biblioteca popular que simplifica requisições HTTP com recursos como interceptadores, transformação automática de JSON e cancelamento de requisições.

XMLHttpRequest: API legada ainda suportada para compatibilidade, mas considerada verbosa comparada a alternativas modernas.

A.2 *React* - Biblioteca para Interfaces de Usuário

React é uma biblioteca *JavaScript* declarativa e baseada em componentes para construção de interfaces de usuário, desenvolvida pelo *Facebook* (atual *Meta*) e lançada como código aberto em 2013. Banks (2020) descreve *React* como uma mudança de paradigma no desenvolvimento *frontend*, introduzindo conceitos como *Virtual DOM* e fluxo de dados unidirecional que influenciaram toda a indústria.

A.2.1 Conceitos Fundamentais

Componentes: Blocos de construção reutilizáveis que encapsulam estado e lógica de apresentação. Podem ser definidos como funções (componentes funcionais) ou classes (componentes de classe), com tendência moderna favorecendo componentes funcionais.

***JavaScript XML* (JSX) (*JavaScript XML*)**: Extensão de sintaxe que permite escrever marcação similar a HTML dentro de *JavaScript*. Transpilado para chamadas *React.createElement*, oferece segurança de tipos e poder expressivo de *JavaScript* dentro de *templates*.

Virtual DOM: Representação em memória da *User Interface* (UI) que *React* usa para calcular mudanças mínimas necessárias ao *Document Object Model* (DOM) real. Processo de reconciliação compara *Virtual DOM* atual com anterior (diferenciação) e aplica apenas mudanças necessárias, otimizando desempenho.

***Props* e Estado**: *Props* são dados imutáveis passados de componente pai para filho, enquanto estado representa dados mutáveis internos ao componente. Separação clara entre *props* e estado promove componentes previsíveis e testáveis.

A.2.2 *React Hooks*

Introduzidos no *React* 16.8, *Hooks* permitem usar estado e outras funcionalidades do *React* em componentes funcionais. Larsen (2021) argumenta que *Hooks* simplificaram significativamente o desenvolvimento *React*:

useState: Adiciona estado local a componentes funcionais, retornando valor atual e função definidora.

useEffect: Gerencia efeitos colaterais como chamadas a APIs, assinaturas e manipulação manual do DOM. Substitui métodos de ciclo de vida de componentes de classe.

useContext: Consome valores de *React Context*, evitando perfuração de propriedades em árvores profundas de componentes.

useMemo* e *useCallback: Otimizam desempenho através de memoização de valores computados e *callbacks*.

***Hooks* Personalizados**: Permitem extrair lógica com estado reutilizável, promovendo composição e separação de responsabilidades.

A.2.3 Gerenciamento de Estado

Em aplicações *React* complexas, gerenciamento de estado torna-se crucial:

Estado Local: Estado mantido em componentes individuais usando *useState* ou *useReducer*. Adequado para dados que não precisam ser compartilhados.

Context API: Mecanismo nativo para compartilhar dados entre componentes sem perfuração de propriedades. Útil para dados globais como tema, autenticação ou preferências.

Estado Derivado: Valores calculados a partir de estado ou *props*, computados durante renderização ou memoizados com *useMemo*.

A.2.4 *React* e consumo de APIs

Integração com APIs *backend* é padrão comum em aplicações *React*. Padrões estabelecidos incluem:

Busca de Dados em *useEffect*: Chamadas a APIs tipicamente ocorrem em *useEffect* com *array* de dependências apropriado. Função de limpeza cancela requisições pendentes para evitar vazamentos de memória.

Estados de Carregamento e Erro: Componentes mantêm estados separados para carregamento, erro e dados, providenciando *feedback* visual durante operações assíncronas.

***Hooks* Personalizados para API**: *Hooks* personalizados encapsulam lógica de busca, *cache* e tratamento de erro, promovendo reuso entre componentes.

A.3 *TailwindCSS - Framework CSS Utility-First*

TailwindCSS é um *framework* CSS *utility-first* que providencia classes de baixo nível para construir *designs* personalizados diretamente no markup. Criado por Adam Wathan em 2017, *Tailwind* representa mudança filosófica no desenvolvimento CSS, favorecendo composição de utilitários sobre criação de componentes CSS personalizados. Wathan (2022) argumenta que esta abordagem resulta em desenvolvimento mais rápido e manutenível.

A.3.1 Filosofia *Utility-First*

Diferente de *frameworks* baseados em componentes como *Bootstrap* ou *Material-UI*, *TailwindCSS* fornece utilitários atômicos que representam propriedades CSS individuais:

Utilitários Atômicos: Classes como `p-4` (*padding*), `text-center` (alinhamento de texto), `bg-blue-500` (cor de fundo) aplicam estilos específicos sem abstração.

Composição sobre Herança: *Designs* são construídos combinando múltiplos utilitários ao invés de estender classes base. Elimina problemas de especificidade e cascata não intencional.

Fonte Única de Verdade: Estilos são definidos diretamente na marcação, facilitando entendimento de aparência do componente sem alternar entre arquivos.

A.3.2 Sistema de *Design* Consistente

TailwindCSS impõe consistência através de sistema de *design* predefinido mas personalizável:

Escala de Espaçamento: Sistema de espaçamento baseado em *rem* com escala consistente (0.25rem, 0.5rem, 1rem, etc.) garante ritmo visual harmonioso.

Paleta de Cores: Sistema de cores com 10 tons por cor (50-900) permite consistência visual mantendo flexibilidade. Cores semânticas para estados (sucesso, aviso, erro) padronizam *feedback* visual.

Tipografia Responsiva: Classes de tipografia com tamanhos e alturas de linha otimizados para legibilidade. Sistema de famílias de fontes permite hierarquia visual clara.

***Breakpoints* Consistentes**: *Breakpoints* predefinidos (sm, md, lg, xl, 2xl) com prefixos para utilitários responsivos permitem *design mobile-first* intuitivo.

A.3.3 Otimização e Desempenho

Um dos maiores desafios de *frameworks utility-first* é o tamanho do CSS gerado. *TailwindCSS* aborda este problema através de várias estratégias:

***PurgeCSS*/Modo JIT**: Remove utilitários não utilizados em produção através de análise estática do código. Modo *Just-In-Time* gera apenas utilitários utilizados, reduzindo drasticamente tamanho do pacote.

Minificação Inteligente: Compressão agressiva e remoção de espaços em branco reduzem tamanho final. Compressão *Gzip* tipicamente resulta em pacotes menores que 10KB.

CSS Crítico: Utilitários podem ser organizados para CSS crítico *inline*, melhorando desempenho de primeira pintura.

A.3.4 Personalização e Extensibilidade

Arquivo de Configuração: `tailwind.config.js` permite personalização completa do sistema de *design*. Cores, fontes, espaçamentos e *breakpoints* podem ser modificados ou estendidos.

Plugins: Ecossistema rico de *plugins* oficiais e comunitários adiciona funcionalidades como formulários, tipografia, proporção de aspecto e animações.

Diretivas e Funções: `@apply` permite criar componentes reutilizáveis, `@layer` organiza CSS personalizado, e `theme()` acessa valores de configuração.

A.3.5 Vantagens no Desenvolvimento

Reimer (2021) identifica benefícios significativos da abordagem *utility-first*:

- **Velocidade de Desenvolvimento**: Elimina alternância de contexto entre HTML e CSS
- **Manutenibilidade**: Mudanças são localizadas e previsíveis
- **Consistência**: Sistema de *design* imposto previne desvios visuais
- **Responsividade**: Prefixos de *breakpoint* simplificam *design* responsivo
- **Prototipagem Rápida**: Iteração visual rápida sem escrever CSS personalizado

A.3.6 Integração com *React*

TailwindCSS integra naturalmente com *React* através de *classNames*:

Classes Condicionais: Bibliotecas como `clsx` ou `classnames` facilitam aplicação condicional de utilitários baseada em estado ou *props*.

Padrões de Componente: Componentes *React* podem encapsular combinações comuns de utilitários, mantendo benefícios de *utility-first* com reusabilidade de componentes.

Compatibilidade *CSS-in-JS*: *Tailwind* pode coexistir com soluções *CSS-in-JS* quando abstração adicional é necessária.