UNIVERSIDADE FEDERAL DO CEARÁ *CAMPUS* SOBRAL

DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO

CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

KLAYVER XIMENES CARMO

UM ESTUDO COMPARATIVO ENTRE TECNOLOGIAS DE BACK-END: NODE.JS, DJANGO REST FRAMEWORK E ASP.NET CORE

SOBRAL

2023

KLAYVER XIMENES CARMO

UM ESTUDO COMPARATIVO ENTRE TECNOLOGIAS DE BACK-END: NODE.JS, DJANGO REST FRAMEWORK E ASP.NET CORE

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação do *Campus* Sobral da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Fischer Jônatas Ferreira.

SOBRAL

2023

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

C285e Carmo, Klayver Ximenes.
Um estudo comparativo entre tecnologias de back-end: Node.js, Django REST Framework e ASP.NET Core / Klayver Ximenes Carmo. – 2023.
106 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, , Sobral, 2023.
Orientação: Prof. Dr. Fischer Jônatas Ferreira.

1. Back-end. 2. Node.js. 3. Django REST Framework. 4. ASP.NET Core. I. Título.
CDD

KLAYVER XIMENES CARMO

UM ESTUDO COMPARATIVO ENTRE TECNOLOGIAS DE BACK-END: NODE.JS, DJANGO REST FRAMEWORK E ASP.NET CORE

Aprovada em: 28 de Novembro de 2023

BANCA EXAMINADORA

Prof. Dr. Fischer Jônatas Ferreira (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Wendley Souza da Silva
Universidade Federal do Ceará (UFC)

Prof. Dr. Antonio Josefran de Oliveira Bastos
Universidade Federal do Ceará (UFC)

AGRADECIMENTOS

Agradeço primeiramente à minha família, que sempre me apoiou e incentivou cada um dos meus esforços.

Ao meu dedicado professor orientador, Dr. Fischer, expresso minha profunda gratidão pelo apoio constante e pela confiança depositada em mim. Sua orientação foi fundamental em cada etapa desta pesquisa, orientando-me com maestria.

Aos professores participantes da banca examinadora, meu sincero agradecimento pelo tempo dedicado, pelas colaborações e sugestões que contribuíram significativamente para aprimorar este trabalho.

À Universidade Federal do Ceará (UFC), expresso meu reconhecimento aos professores e servidores que, com seu conhecimento e incentivo, enriqueceram o meu percurso acadêmico.

Por último, mas não menos importante, meus agradecimentos à amigos que fizeram toda a diferença: Cícero, que esteve ao meu lado desde o início desta jornada; Jannielly, Lucas, Cléber, Júnior, Ítalo, Natanael, Vinícius, Ramiro, e a todos que contribuíram nos momentos de estudo e diversão; Cleani, Nícolas, Tafarel, Manuel e aos amigos ciclistas pelo conhecimento compartilhado.

Obrigado a todos pelo apoio constante, que tornou esta caminhada mais leve e enriquecedora.

"Viver é como andar de bicicleta. É preciso estar em constante movimento para manter o equilíbrio." (Albert Einstein)

RESUMO

CONTEXTO: No cenário tecnológico atual, o desenvolvimento de aplicações Web, com foco no *back-end*, tem se tornado uma área de grande importância e evolução constante. Esta camada desempenha um papel fundamental na arquitetura de um sistema. A escolha da tecnologia a ser utilizada para essa camada do projeto é essencial para garantir o desempenho, a segurança e a eficiência da aplicação como um todo. MOTIVAÇÃO: Nesse contexto, existem diversas tecnologias disponíveis que se destacam para atender a esses propósitos, cada uma com suas peculiaridades. É crucial compreender as diferenças e características da ferramenta mais adequada para determinado contexto de desenvolvimento. No entanto, até o momento, não se tem conhecimento de estudos recentes na literatura que abordem a identificação de requisitos e características específicas para orientar a seleção de uma tecnologia *back-end* em aplicações Web. OBJETIVO: Em vista disso, este trabalho realiza uma análise comparativa entre as tecnologias *back-end* mais utilizadas de acordo com a pesquisa mais recente realizada pelo Stack Overflow: Node.js, Django REST Framework e ASP.NET Core. Assim, objetiva-se fornecer informações relevantes para a escolha de cada uma destas ferramentas em diferentes sistemas e aplicações Web. METODOLOGIA: Para condução do presente estudo, foi realizado uma etapa de estudo bibliográfico e definição de critérios para comparação teórica. Em seguida, foi feito uma implementação de um servidor API em cada uma das tecnologias para a realização de testes e coleta de dados e métricas, como desempenho e performance em diferentes cenários. RESULTADOS: Como resultados deste estudo, foi possível observar que, na comparação teórica, embora alguns *frameworks* possuam funcionalidades e compatibilidades nativas, outras podem ser implementadas por meio de bibliotecas externas. No entanto, nos resultados práticos, observou-se que que o consumo de recursos pelo .Net foi significativamente elevado, enquanto o Node.js e o DRF demonstraram baixo consumo e desempenho semelhante. Com relação aos tempos de resposta, o .Net demonstrou o desempenho mais eficaz, seguido do Node.js e do DRF. BENEFICIADOS: Os resultados deste estudo beneficiam desenvolvedores, empresas e pesquisadores. Desta forma, eles poderão observar as características levantadas sobre as principais tecnologias de *back-end* e utilizar em seus projetos. Além disso, servirá como ponto de referência para pesquisadores que atuam na área, contribuindo com o avanço no desenvolvimento de módulos de *back-end* para aplicações Web.

Palavras-chave: Back-end, Node.js, Django REST Framework, ASP.NET Core.

ABSTRACT

CONTEXT: In the current technological scenario, the development of Web applications, with a focus on *back-end*, has become an area of great importance and constant evolution. This layer plays a fundamental role in the architecture of a system. The choice of technology to be used for this layer of the project is essential to guarantee the performance, security and efficiency of the application as a whole. MOTIVATION: In this context, there are several technologies available that stand out for these purposes, each with its own peculiarities. It is crucial to understand the differences and characteristics of the most suitable tool for a given development context. However, as far as is known, no recent studies in the literature identify requirements and specific characteristics to guide the selection of a back-end technology in Web applications. OBJECTIVE: Considering this, this work compares the most used back-end technologies according to the most recent survey conducted by Stack Overflow: Node.js, Django REST Framework, and ASNP.NET Core. Therefore, it aims to provide relevant information for the choice of each of these tools in different systems and Web applications. METHODOLOGY: To this end, a bibliographical study and definition of criteria for theoretical comparison were carried out. Next, an API server was implemented in each of the frameworks to carry out tests and collect data and metrics, such as performance and performance in different scenarios.

RESULTS: As a result of this study, it was possible to observe that, in the theoretical comparison, although some frameworks have native functionalities and compatibilities, others can be implemented through external libraries. However, in practical results, we observed that .Net's resource consumption was significantly high, while Node.js and DRF showed low consumption and similar performance. With regard to response times, .Net showed the most efficient performance, followed by Node.js and DRF. BENEFITS: The results of this study benefit developers, companies and researchers. In this way, they will be able to observe the characteristics raised about the main back-end technologies and use them in their projects. In addition, it will serve as a reference point for researchers working in the field, contributing to the advancement in the development of back-end modules for Web applications.

Keywords: Back-end, Node.js, Django REST Framework, ASP.NET Core.

LISTA DE FIGURAS

Figura 1 – Modelo arquitetural de uma *Application Programming Interface* (API) *Representational State Transfer* (REST) 21
Figura 2 – Página de *download* do *Workbench* MySQL 22
Figura 3 – Página de configuração do *Workbench* MySQL 23
Figura 4 – Página inicial de *download* da ferramenta Insomnia 24
Figura 5 – Painel inicial da ferramenta Insomnia 25
Figura 6 – Interface da ferramenta Insomnia 25
Figura 7 – Funcionamento do motor V8 27
Figura 8 – Diagrama de operações do *loop* de eventos 29
Figura 9 – Diagrama de operações do *loop* de eventos 29
Figura 10 – Funcionamento do motor V8 35
Figura 11 – Utilização do método GET (API Node.js) 36
Figura 12 – Utilização do método POST (API Node.js) 37
Figura 13 – Utilização do método PUT (API Node.js) 38
Figura 14 – Utilização do método DELETE (API Node.js) 39
Figura 15 – Representação modelo *Model View Template* (MVT) 41
Figura 16 – Estrutura de arquivos inicial 45
Figura 17 – Utilização do método GET (API Django REST) 50
Figura 18 – Utilização do método POST (API Django REST) 50
Figura 19 – Utilização do método PUT (API Django REST) 51
Figura 20 – Utilização do método DELETE (API Django REST) 51
Figura 21 – Painel administrativo do Django 52
Figura 22 – Disponibilização e consumo de pacotes por meio do NuGet 54
Figura 23 – Página inicial do instalador do *Software Development Kit* (SDK) 55
Figura 24 – Instalação do ambiente ASP.NET 56
Figura 25 – Árvore de arquivos inicial 57
Figura 26 – Geração dos controladores com *Scaffold* 61
Figura 27 – Configuração dos controladores 61
Figura 28 – Exemplo método GET (API ASP.NET Core) 63
Figura 29 – Exemplo método POST (API ASP.NET Core) 64
Figura 30 – Exemplo método PUT (API ASP.NET Core) 66

Figura 31 – Exemplo método DELETE (API ASP.NET Core) 67
Figura 32 – Etapas metodológicas 68
Figura 33 – Diagrama do *software* alvo 70
Figura 34 – Processos para a modelagem dos testes 72
Figura 35 – Fluxo de requisições dos testes 74
Figura 36 – Modelagem do teste de pico 75
Figura 37 – Modelagem do teste de carga crescente 76
Figura 38 – Modelagem do teste de resistência 77
Figura 39 – Consumo médio de recursos - teste de pico 83
Figura 40 – Tempo de resposta médio dos métodos HTTP - teste de pico 84
Figura 41 – Consumo médio recursos - teste de carga crescente 85
Figura 42 – Tempo de resposta médio dos métodos HTTP - teste de carga crescente . . . 86
Figura 43 – Consumo médio de recursos - teste de resistência 87
Figura 44 – Tempo de resposta médio dos métodos HTTP - teste de resistência 88
Figura 45 – Inicialização do .Net no teste de carga crescente 89
Figura 46 – Consumo médio de recursos - média total 90
Figura 47 – Tempo de resposta médio dos métodos HTTP - média total 90

LISTA DE TABELAS

Tabela 1 – Métodos *Hypertext Transfer Protocol* (HTTP) 21

Tabela 2 – Modelagem do teste de pico 75

Tabela 3 – Modelagem do teste de carga crescente 76

Tabela 4 – Modelagem do teste de resistência 77

Tabela 5 – Suporte das tecnologias aos principais Sistema Operacional (SO)s. 80

Tabela 6 – Suporte das tecnologias aos principais bancos de dados. 80

Tabela 7 – Suporte das tecnologias ao ORM nativo 81

Tabela 8 – Suporte das tecnologias à documentação nativa. 82

LISTA DE ABREVIATURAS E SIGLAS

AOT	*Ahead-Of-Time*
API	*Application Programming Interface*
BSD	*Berkeley Software Distribution*
CLI	*Common Language Infrastructure*
CLR	*Common Language Runtime*
CPU	Unidade Central de Processamento
CRUD	*Create, Read, Update e Delete*
CSS	*Cascading Style Sheets*
DOM	*Document Object Model*
DRF	*Django REST Framework*
DRY	*Don't Repeat Yourself*
EF	*Entity Framework*
EFC	*Entity Framework Core*
GQM	Goal Question Metric
HTML	*HyperText Markup Language*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*

IDE Ambiente de Desenvolvimento Integrado

IL Linguagem intermediária

JIT *Just in Time*

JS JavaScript

JSON *JavaScript Object Notation*

LINQ *Language Integrated Query*

MVC *Model View Controller*

MVT *Model View Template*

npm *Node Package Manager*

ORM *Object Relational Mapping*

PID *Process Identification Number*

POO *Object-oriented Programming*

RAM Memória de Acesso Aleatório

REST *Representational State Transfer*

SDK *Software Development Kit*

SO Sistema Operacional

SQL *Structured Query Language*

URL *Uniform Resource Locator*

VU	Usuário virtual
XML	*Extensible Markup Language*
YAML	*YAML Ain't Markup Language*

SUMÁRIO

1 **INTRODUÇÃO** 16
2 **OBJETIVOS** 18
2.1 **Objetivos** 18
2.1.1 *Objetivos Gerais* 18
2.1.2 *Objetivos Específicos* 18
2.2 **Questões de pesquisa** 19
3 **FUNDAMENTAÇÃO TEÓRICA** 20
3.1 **API e API RESTful** 20
3.1.1 *Persistência dos dados e configuração do banco de dados* 21
3.2 **Insomnia** 23
3.3 **Introdução ao Node.js** 26
3.3.1 *JavaScript engine V8* 27
3.3.2 *JavaScript Libuv (Eventloop)* 28
3.3.3 *Diferença entre JavaScript para front-end e back-end* 29
3.3.4 *Ambiente de desenvolvimento para o Node.js* 30
3.3.5 *Framework Express* 31
3.3.6 *Exemplo Create, Read, Update e Delete (CRUD) Node.js com* `Express` . . 31
3.4 **Introdução ao Django REST Framework** 39
3.4.1 *Padrão MVT* 40
3.4.2 *Princípio Don't Repeat Yourself (DRY)* 41
3.4.3 *Serialização e deserialização* 42
3.4.4 *Ambiente de desenvolvimento para o Django REST Framework* 42
3.4.5 *Exemplo CRUD Django REST Framework* 43
3.5 **Introdução ao .NET** 52
3.5.1 *Gerenciador de pacotes NuGet e Entity Framework Core* 53
3.5.2 *Ambiente de desenvolvimento para o ASP.NET Core* 54
3.5.3 *Exemplo CRUD ASP.NET Core* 55
4 **METODOLOGIA** 68
4.1 **Levantamento de características teóricas** 68
4.1.1 *Análise comparativa teórica* 69

4.2 **Levantamento de requisitos do *software* alvo** 70
4.3 **Implementação do *software* alvo** 70
4.4 **Métricas comparativas práticas** 71
4.5 **Execução dos testes para leitura das métricas** 72
5 **RESULTADOS E DISCUSSÕES** 79
5.1 **Análise teórica** 79
5.1.1 ***Sistemas Operacionais suportados*** 79
5.1.2 ***Compatibilidade com bancos de dados*** 80
5.1.3 ***Suporte ao ORM nativo*** 81
5.1.4 ***Documentação nativa da API*** 81
5.2 **Análise prática** 82
5.2.1 ***Teste de pico*** 82
5.2.2 ***Teste de carga crescente*** 84
5.2.3 ***Teste de resistência*** 86
5.3 **Outras observações** 88
6 **CONCLUSÕES E TRABALHOS FUTUROS** 91
REFERÊNCIAS 93
APÊNDICES 95
APÊNDICE A –JAVASCRIPT 96
A.0.1 ***Sintaxe do JavaScript*** 96
A.0.1.1 *Declaração de variáveis* *96*
A.0.1.2 *Condicionais* *96*
A.0.1.3 *Laços de Repetições* *97*
A.0.1.4 *Funções* *98*
APÊNDICE B –PYTHON 99
B.0.1 ***Sintaxe do Python*** 99
B.0.1.1 *Declaração de variáveis* *99*
B.0.1.2 *Condicionais* *99*
B.0.1.3 *Laços de Repetições* *100*
B.0.1.4 *Funções* *100*
APÊNDICE C –C# 102
C.0.1 ***Sintaxe do C#*** 102

C.0.1.1 Paradigma POO 102
C.0.1.2 Declaração de variáveis 102
C.0.1.3 Condicionais 103
C.0.1.4 Laços de Repetições 103
C.0.1.5 Funções e métodos 104

1 INTRODUÇÃO

O desenvolvimento Web teve seu início na década de 90 com a criação dos primeiros sites, cujo objetivo era o compartilhamento de informações na internet. Nessa época, os sites eram compostos principalmente por páginas estáticas, contendo elementos básicos como textos e imagens (PAGE, 2023; MDN, 2023). Com o passar do tempo, as tecnologias evoluíram permitindo um desenvolvimento Web mais dinâmico e interativo, com o surgimento de *frameworks* e bibliotecas, tornando o processo mais rápido e eficiente. A separação dos termos *front-end* e *back-end* começou a surgir por meados do ano de 2008, com o desenvolvimento acelerado da Web e popularização das novas tecnologias. O *back-end* surgiu como uma nova camada do desenvolvimento Web para garantir um bom funcionamento de um site ou aplicação com gerenciamento de conteúdo e infraestrutura relacionadas ao servidor (ROCKETSEAT, 2023). Com isso, é possível agora gerenciar servidores, bancos de dados, processar e armazenar dados, bem como outros recursos necessários para a execução adequada de uma aplicação Web.

A crescente e contínua demanda por desenvolvedores tem destacado a real importância desta função no cenário de desenvolvimento atual. Simultaneamente, o cenário em constante mudança da tecnologia trouxe muitas opções para atender a necessidade de aplicações Web escaláveis, eficientes e seguras. Cada uma dessas tecnologias, fornecendo recursos únicos, têm o propósitos de tornar o desenvolvimento *back-end*mais robusto. A escolha da tecnologia adequada para um projeto específico busca estar de acordo com as tendências do mercado, seguindo uma abordagem estratégica e planejada. (HACKERRANK, 2022).

Em 2022, o *StackOverflow* conduziu uma pesquisa com mais de 70 mil desenvolvedores com o objetivo de determinar as tecnologias mais desejadas, utilizadas e populares, bem como outras informações relevantes. De acordo com os resultados, pelo décimo ano consecutivo, o *JavaScript* foi a linguagem de programação mais utilizada pelos desenvolvedores, juntamente com *Python* e *C#* que estão entre as 10 primeiras posições. Os resultados da pesquisa também mostraram que os *frameworks* `Django`, `ASP.NET Core` e `Node.js` estão entre os 10 mais utilizados, sendo esta última a tecnologia *back-end* mais utilizada pelos profissionais. (STACKOVERFLOW, 2022)

Diante da grande quantidade de tecnologias existentes, este trabalho tem como objetivo fazer uma apresentação comparativa, analisando características, funcionalidades e elencando vantagens e desvantagens das tecnologias de *back-end* mais utilizadas segundo a pesquisa. As tecnologias escolhidas foram `Node.js`, `Django` e `ASP.NET Core`. Este estudo

visa oferecer uma fundamentação prática e teórica para auxiliar empresas e desenvolvedores de software na escolha das tecnologias de *back-end*, visando aprimorar a qualidade de seus produtos e projetos. Tendo como objetivo fornecer um recursos que permitam tomar decisões informadas ao escolher a tecnologia de *back-end* mais apropriada, fornecendo uma compreensão sólida das vantagens e desafios de cada plataforma.

Foi realizado um levantamento de informações sobre as tecnologias selecionadas, destacando aspectos como facilidade de desenvolvimento, segurança, compatibilidades e desempenho. Além disso, para uma compreensão mais tangível, serão elaborados casos de uso práticos, consistindo na implementação um sistema Web por meio da criação de APIs utilizando cada uma das tecnologias em questão. A abordagem prática permitiu a avaliação de como cada tecnologia se comporta em cenários reais, explorando os desafios e potencialidades em tempo de desenvolvimento. A coleta dos dados se deu por meio de estudos comparativos, análises de documentação oficial, trabalhos relacionados e análise de métricas de desempenho. Os estudos teóricos e práticos permitiram uma compreensão de cada tecnologia, facilitando em tomadas de decisões fundamentadas na escolha da tecnologia *back-end* mais adequada para determinado contexto.

Desta forma, este trabalho foi composto de algumas etapas para a obtenção dos resultados. Primeiramente, obteve-se uma análise teórica e comparativa das tecnologias selecionadas, com informações sobre características, vantagens e limitações. Isso possibilitou a compreensão para tomada de decisões com base na escolha da tecnologia mais adequada para uma determinada necessidade. Através do desenvolvimento das APIs em cada uma das tecnologias, realizou-se uma comparação dinâmica explorando aspectos como facilidade de desenvolvimento e análise de desempenho, realizando testes e medições em cenários específicos. Estes testes permitiram a compreensão de como as tecnologias se comportam com respeito ao consumo de recursos e tempos de respostas em diferentes casos.

Este trabalho está organizado seguindo a divisão dos Capítulos 2, 3, 4, 5 e 6. No Capítulo 2, são apresentados os objetivos gerais e específicos, além da motivação e das questões de pesquisa. O Capítulo 3 aborda a fundamentação teórica deste estudo. Descrevendo conceitos e tópicos relevantes no desenvolvimento *back-end*. Já o Capítulo 4 descreve a metodologia empregada na condução deste trabalho. No Capítulo 5, estão expostos os resultados encontrados e a discussão dos mesmos. Por fim, no Capítulo 6, são apresentadas as considerações finais e propostas para trabalhos futuros.

2 OBJETIVOS

Neste capítulo, são descritos os objetivos, motivação e questões de pesquisa deste estudo. A Seção 2.1 descreve os objetivos, incluindo gerais e específicos. A Seção 2.2 apresenta a motivação e as questões de pesquisa.

2.1 Objetivos

A definição dos objetivos foram feitos com base no modelo objetivo-questão-métrica Goal Question Metric (GQM) (BASILI; ROMBACH, 1988). Foram **comparados** *frameworks back-end* de maneira teórica e prática; **com a finalidade** de i) identificar funcionalidades nativas em fase de desenvolvimento e ii) analisar sua performance e desempenho; **no que diz respeito** a auxiliar profissionais e pesquisadores na escolha da abordagem mais adequada para suas estratégias **sob a perspectiva** de desenvolvedores e pesquisadores na área de desenvolvimento back-end no contexto de construção e estudo de aplicações Web.

2.1.1 Objetivos Gerais

Este trabalho tem como objetivo geral analisar e comparar os *frameworks back-end* `Node.js`, `Django REST Framework` e `ASP.NET Core`. Pontuando características/funcionalidades que são presenciadas em tempo de desenvolvimento e comparar resultados práticos em diferentes cenários. Assim como, relatar o aprendizado e resultados obtidos no desenvolvimento de um software alvo destinado ao controle de estoque de laboratório.

2.1.2 Objetivos Específicos

Os objetivos específicos deste trabalho são:

- Realizar estudos teóricos sobre os frameworks.
- Pontuar características de desenvolvimento entre os frameworks.
- Desenvolver um *software* alvo utilizando os frameworks em estudo.
- Aplicar e analisar métricas com base no software alvo desenvolvido.

2.2 Questões de pesquisa

Com base em pesquisas bibliográficas, até o momento, não foram encontrados estudos comparativos focados à avaliação das compatibilidades/funcionalidades teóricas e à análise de cenários práticos de testes nos três *frameworks* alvo. Motivado por isto, foram formuladas as seguintes questões de pesquisa:

QP_1: *Qual o framework com mais funcionalidades nativas seguindo as características escolhidas para estudo?*

- **Cenário 1 - Teste de pico**

  QP_2: *Qual o framework mais otimizado com relação ao consumo de recursos no cenário de teste de pico?*

  QP_3: *Qual o framework mais otimizado com relação ao tempo de resposta no cenário de teste de pico?*

- **Cenário 2 - Teste de carga crescente**

  QP_4: *Qual o framework mais otimizado com relação ao consumo de recursos no cenário de teste de carga crescente?*

  QP_5: *Qual o framework mais otimizado com relação ao tempo de resposta no cenário de teste de carga crescente?*

- **Cenário 3 - Teste de resistência**

  QP_6: *Qual o framework mais otimizado com relação ao consumo de recursos no cenário de teste de resistência?*

  RQ_7: *Qual o framework mais otimizado com relação ao tempo de resposta no cenário de teste de resistência?*

Estas questões de pesquisa exploram a otimização dos *frameworks* em termos de eficiência no consumo de recursos e tempo de resposta em diferentes cenários. A análise de dados práticos possibilita uma comparação que pode beneficiar empresas e desenvolvedores de *software*, oferecendo resultados em alguns cenários de teste que podem apoiar a escolha de uma estratégia de teste que melhor atenda às suas necessidades. Além disso, auxilia pesquisadores da área em projetos de pesquisa futuros.

3 FUNDAMENTAÇÃO TEÓRICA

Este capítulo tem como principal objetivo fornecer uma visão geral dos conceitos utilizados e relacionados ao presente trabalho. Assim, as seções apresentam, de forma didática, algumas tecnologias de *back-end* (`Node.js`, `Django DRF` e `ASP.NET`). A Seção 3.1 apresenta uma visão geral sobre API e API RESTful. A Seção 3.2 apresenta uma ferramenta chamada INSOMNIA para utilizar os exemplos de API demonstrados no texto. Finalmente, as Seções 3.3, 3.4 e 3.5 discutem fundamentos e apresentam exemplos práticos sobre o `Node.js`, `Django DRF` e `ASP.NET`, respectivamente.

3.1 API e API RESTful

A API (*Application Programming Interface*), ou Interface de Programação de Aplicação, é uma interface utilizada na programação que permite a comunicação entre sistemas e aplicações. Isto é feito por meio de um conjunto de protocolos e ferramentas para o desenvolvimento de *software* e aplicativos. No contexto do desenvolvimento Web, a API é exposta por um servidor que permite a comunicação e interação de clientes externos e em diferentes plataformas, como em *sites*, aplicativos *mobile* ou *softwares desktop*. Seu desenvolvimento permite a criação de uma camada de abstração entre o *front-end* e o *back-end*, facilitando a integração entre os serviços.

API RESTFul é um tipo específico de API que segue as restrições e princípios arquiteturais definidos pela arquitetura REST. A arquitetura REST é baseada nos protocolos HTTP e *Hypertext Transfer Protocol Secure* (HTTPS) que são utilizados na comunicação entre cliente e servidor na maioria dos serviços de aplicações (VALES; BRADA, 2020). A nomenclatura "REST" refere-se à independência das solicitações anteriores de um cliente ao servidor, ou seja, um cliente pode fazer solicitações de recursos em qualquer ordem, estando em ausência de estado e sendo isolada de outras requisições (AMAZON, 2023a).

A Tabela 1 apresenta os métodos HTTP mais utilizados para manipular os recursos disponíveis no servidor, tornando a arquitetura REST mais escalável e flexível devido à popularização e padronização dos verbos HTTP. Geralmente, as respostas retornadas de uma API são no formato *JavaScript Object Notation* (JSON), *Extensible Markup Language* (XML) ou *HyperText Markup Language* (HTML) e são transmitidas por meio dos métodos HTTP. Uma API RESTFul pode ser construída em diferentes tecnologias *back-end*, como `Node.js` com o

framework `Express`, `Django` REST e `ASP.NET` Web API, no qual cada um desses *frameworks* possui suas próprias vantagens e desvantagens.

Tabela 1 – Métodos HTTP

Método	**Ação**	**Exemplo de uso**
GET	Ler	Capturar recursos de um sistema
POST	Criar	Inserir um recursos em um sistema
PUT	Atualizar	Atualizar dados de um recurso em um sistema
DELETE	Apagar	Apagar recursos em um sistema

A Figura 1 mostra como funciona o processo do modelo arquitetural de uma API REST, onde um usuário é capaz de realizar a comunicação com o servidor por meio de uma API fazendo requisições utilizando os métodos HTTP. O servidor então retorna uma resposta HTTP para o cliente, podendo, por exemplo, ser no formato JSON, XML ou HTML. O formato da resposta irá depender da requisição do cliente ou das restrições do servidor.

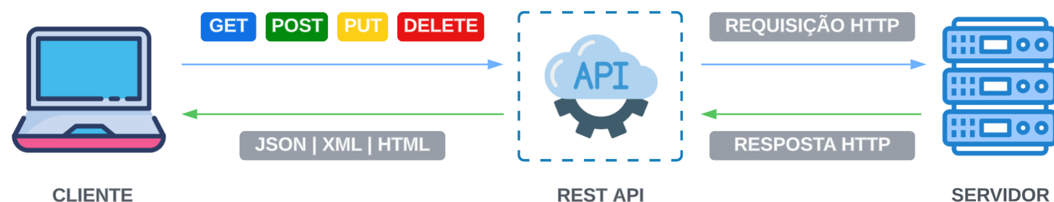

Figura 1 – Modelo arquitetural de uma API REST

3.1.1 *Persistência dos dados e configuração do banco de dados*

Na desenvolvimento de aplicações Web, é de extrema importância o armazenamento das informações que são compartilhadas e trafegadas. Nesse contexto, os bancos de dados desempenham um papel fundamental para o gerenciamento da mesma. Além da persistência, eles também garantem a integridade, segurança e disponibilidade dos dados. Existem diferentes tipos de bancos de dados, cada um projetado para atender necessidades específicas de armazenamento. No geral, podem ser divididos em duas categorias principais: banco de dados relacionais e banco de dados não relacionais. Neste trabalho será utilizado o banco de dados baseado no modelo relacional. Este modelo organiza dos dados em linhas e colunas, onde as tabelas e as relações

entre elas são definidas previamente.

O banco de dados relacional a ser utilizado será o `MySQL`[1], que utiliza a linguagem *Structured Query Language* (SQL) para consulta e interação com os dados. Os procedimentos de instalação do *Workbench* `MySQL` seguirão as etapas para a instalação em uma máquina com sistema operacional *Windows*. O primeiro passo é acessar o site oficial de *download*[2] e clicar na opção download, como destacado em vermelho na Figura 2. Os próximos passos para instalação devem seguir o recomendado pela documentação oficial[3].

Figura 2 – Página de *download* do *Workbench* `MySQL`

Com o *Workbench* já instalado, é necessário criar uma conexão para permitir a transferência de dados entre o aplicativo e o banco de dados. Na página inicial do programa instalado, é preciso clicar no botão de "+" na área de conexões `MySQL`, com isso abrirá uma janela para configuração da conexão, conforme apresenta a Figura 3. Nela, é possível definir o nome da conexão e outras configurações, como usuário, *host* e porta de conexão. Com a conexão já definida, é necessário criar o banco de dados/*schema*. O Listing 3.1.1 apresenta a *query* SQL que deve ser executada dentro do editor de texto da conexão.

[1] https://www.mysql.com/
[2] https://dev.mysql.com/downloads/workbench/
[3] https://dev.mysql.com/doc/workbench/en/wb-installing-windows.html#wb-windows-standalone

```
1 CREATE SCHEMA `nome_do_banco_de_dados`;
```

Listing 3.1.1 – *Query* SQL para criar um banco de dados

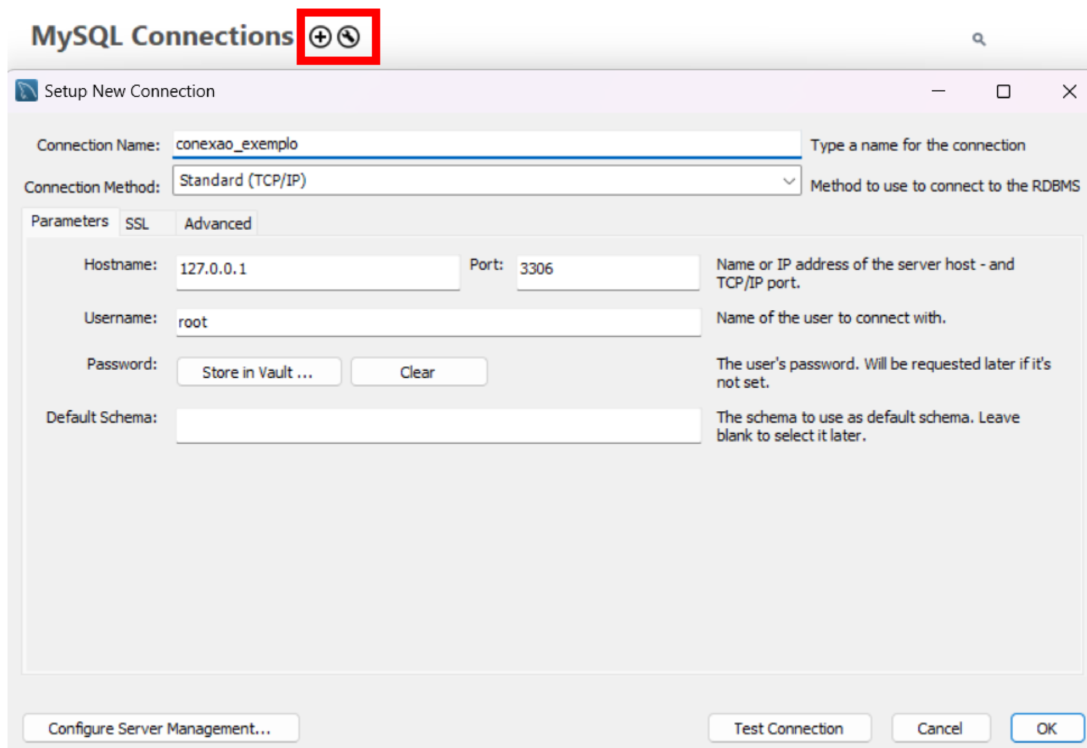

Figura 3 – Página de configuração do *Workbench* `MySQL`

Os bancos bancos a serem utilizados neste trabalho terão os seguintes padrões de nomenclatura:

– 'crud_nodejs': Banco relacionado ao exemplo na tecnologia `Node.js`.

– 'crud_drf': Banco relacionado ao exemplo na tecnologia *Django REST Framework* (DRF).

– 'crud_aspnet': Banco relacionado ao exemplo na tecnologia `.NET`.

3.2 Insomnia

No desenvolvimento de APIs é de suma importância o teste e a documentação do projeto. Existem algumas ferramentas que possibilitam a realização dessas atividades, como `Postman`[4] e `Insomnia`[5]. Neste trabalho será utilizado a ferramenta `Insomnia` para testes das rotas fazendo solicitações HTTP. Ela pode ser obtida para instalação por meio do seu site oficial.

[4] https://www.postman.com/
[5] https://insomnia.rest/

A ferramenta `Insomnia` contém várias funcionalidades que ajudam nas configurações para testes das requisições, como a possibilidade de criar ambientes de desenvolvimento, por meio da declaração de variáveis específicas para cada ambiente, como *Uniform Resource Locator* (URL)s que variam de acordo com o ambiente de trabalho (desenvolvimento, teste e produção). A mesma também possibilita a importação e exportação de uma coleção de requisições para uma equipe de desenvolvedores, não sendo necessário criar um novo ambiente de testes a cada utilização.

Nesta seção será apresentada uma série de passos à fim de demonstrar como a ferramenta `Insomnia` pode ser instalada. O procedimentos seguirão as etapas para a instalação em uma máquina com sistema operacional *Windows*. O primeiro passo é acessar o site oficial de *download*[6], clicar em "*Download Insomnia for Windows*" e esperar o mesmo ser iniciado. A Figura 4 apresenta a tela inicial da página de *download* da ferramenta para o consumo de APIs.

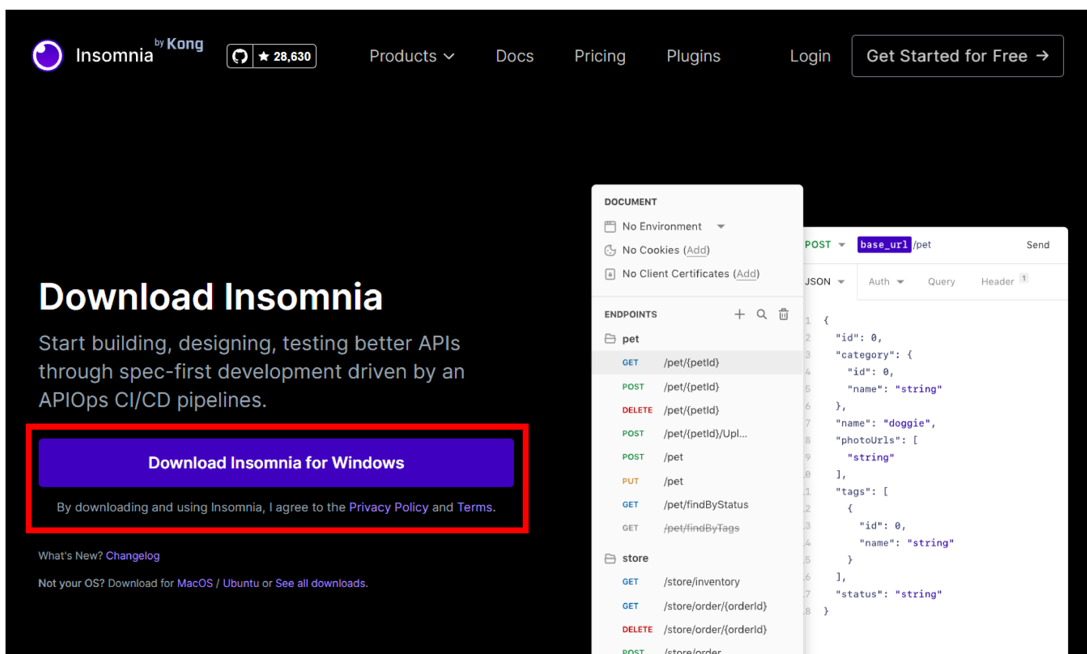

Figura 4 – Página inicial de *download* da ferramenta `Insomnia`

Com a finalização do *download*, o próximo passo é clicar no instalador e inicializar o processo de instalação. Uma tela para este procedimento será iniciada com a logo da ferramenta `Insomnia`. Após a finalização do processo, a janela inicial do programa irá abrir e nela terão as opções de ferramentas que podem ser utilizadas. A Figura 5 apresenta a estrutura da página inicial da ferramenta.

[6] https://insomnia.rest/download

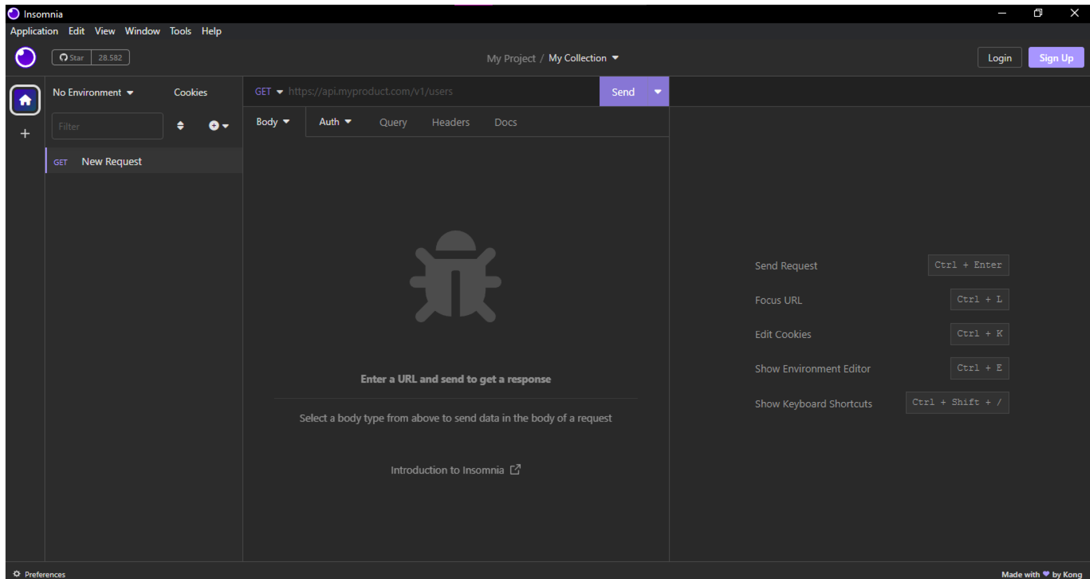

Figura 5 – Painel inicial da ferramenta `Insomnia`

A Figura 6, adaptado de (INSOMNIA, 2023), mostra um exemplo de utilização da ferramenta. Na parte superior é inserido a URL a ser testada, juntamente com o método HTTP desejado. Abaixo da URL é possível inserir algumas configurações conforme o teste a ser feito, como anexar um corpo à requisição, autenticação, cabeçalho entre outros. No lado direito da figura é apresentado um código de resposta 200, indicando o sucesso na requisição, assim como o tempo e o tamanho da mesma. Logo abaixo é possível verificar o conteúdo da resposta retornada da API, no exemplo foi utilizado uma requisição para a API do `GitHub` buscando os dados de um usuário, onde é possível ver as informações como *login*, id e outras.

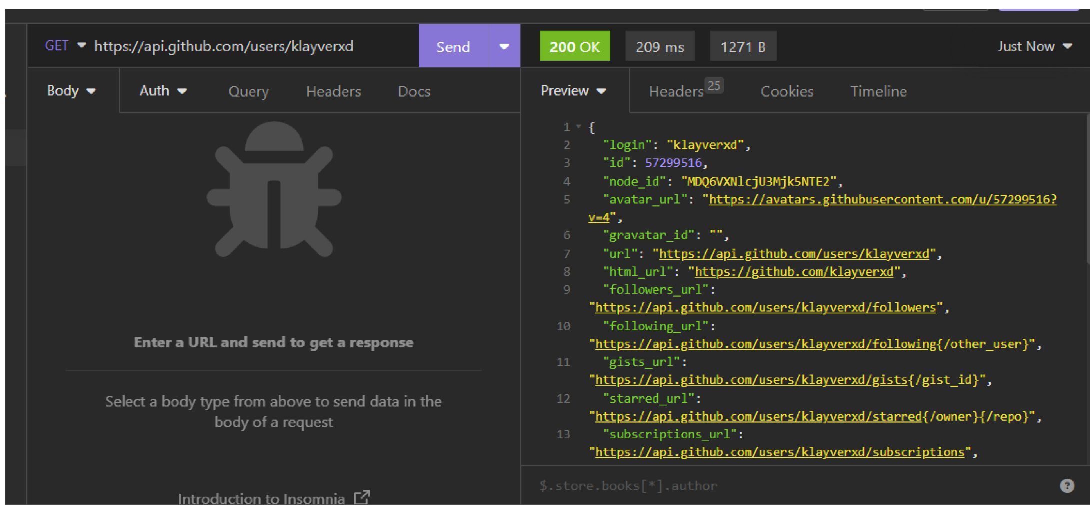

Figura 6 – Interface da ferramenta `Insomnia`

3.3 Introdução ao Node.js

O `Node.js`[7] é uma plataforma para desenvolvimento de *software back-end* criado em 2009 por Ryan Dahl. Sua criação teve como intuito ser uma alternativa para as tecnologias da época, já que as existentes até então, possuíam deficiência em aplicações Web em tempo real. Além disso, o `Node.js` é baseado em *JavaScript*, sendo construído sobre a *engine JavaScript* criado pelo Google para o Chrome, chamado de V8. Por ter *JavaScript* como linguagem de desenvolvimento, o `Node.js` permite o desenvolvimento de aplicações com a mesma linguagem, tanto no lado do servidor como no lado do cliente, tendo também uma grande quantidade de módulos e bibliotecas para um desenvolvimento mais rápido e facilitado.

Como um ambiente de execução *JavaScript* assíncrono orientado a eventos, o `Node.js` é projetado para desenvolvimento de aplicações escaláveis de rede. Sua versatilidade permite que seja utilizado em vários contextos de desenvolvimento, como em aplicações em tempo real, na construção de jogos, plataformas de *streaming* e ambientes escaláveis, como serviços de *back-end* e servidores.

Ao iniciar um projeto temos vários pontos positivos para a escolha inicial do `Node.js`, como sua arquitetura não bloqueante, que permite o processamento simultâneo de várias solicitações de entrada e saída, o que torna a plataforma altamente escalável e adequada para aplicações em tempo real. Por outro lado, também é encontrado algumas desvantagens, como o uso em aplicações que requerem um grande uso de Unidade Central de Processamento (CPU), como aplicativos de renderização 3D ou de processamento de imagens. Esses pontos negativos podem surgir devido ao fato de o `Node.js` ser construído em torno de uma abordagem assíncrona e orientada a eventos. Portanto, para casos que exigem um grande processamento síncrono, seria mais adequado a utilização de outra plataforma de desenvolvimento.

Grandes empresas de tecnologia utilizam o `Node.js` em seus serviços, como *LinkedIn*, *Netflix* e *Paypal*, além de várias *startups* e desenvolvedores de *software* independentes. Além de sua flexibilidade e escalabilidade, o `Node.js` oferece facilidade de uso para manipulação de vídeos, arquivos, criação de *Web services* e outros. Sua grande disponibilidade de módulos e pacotes de código aberto também tornam essa tecnologia uma opção viável para a maioria dos casos de uso, sendo uma das melhores escolhas para construção e solução de aplicações eficientes. Essas grandes possibilidades reforçam o grande uso da tecnologia e sua capacidade de suprir as demandas do mercado.

[7] https://nodejs.org

3.3.1 *JavaScript engine V8*

O V8[8] é um mecanismo de alto desempenho desenvolvido pela *Google*, com código aberto e escrito em *C++*. Ele é usado no Chrome, no `Node.js`, entre outros projetos para a interpretação e execução de código *JavaScript*(V8, 2022). O motor `V8` serve para que o `Node.js` execute o código *JavaScript* por meio do *Just in Time* (JIT), fazendo a compilação *JavaScript* em *bytecode* e convertendo para uma linguagem de baixo nível em tempo de execução, para que seja entendida pelos processadores, ao invés de interpretá-lo em tempo real (KRYLOV *et al.*, 2020). A Figura 8 mostra como funciona essa sequência de processos, desde a análise e conversão à execução do código na CPU.

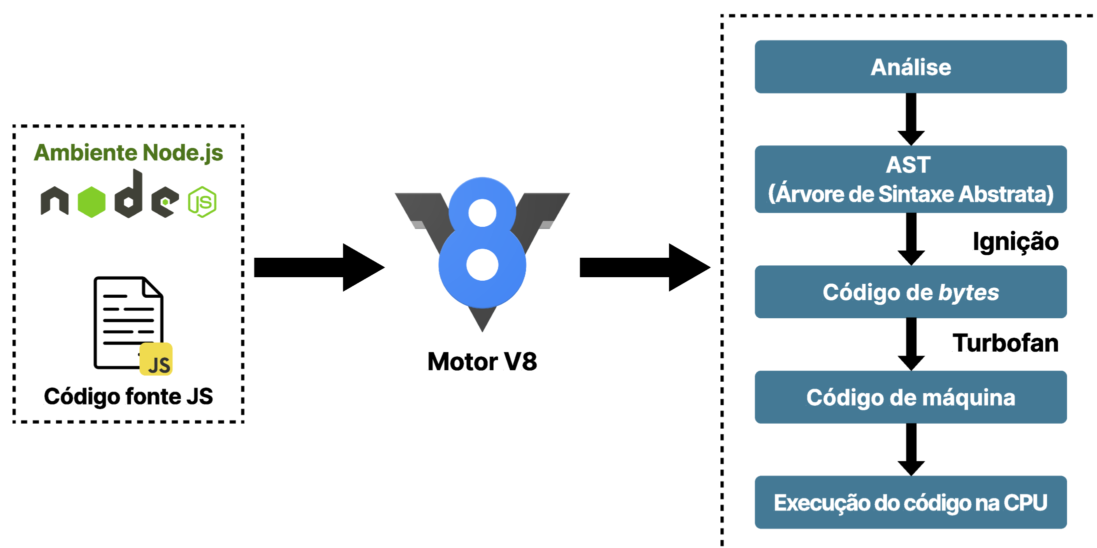

Figura 7 – Funcionamento do motor V8

Além da compilação e execução do código *JavaScript*, o `V8` é responsável pela manipulação, alocação de memória e coleta de lixo dos objetos que não são utilizados. Esse gerenciamento de coleta de lixo é uma das principais funcionalidades que contribui para o desempenho do `Node.js`. Ao liberar a memória ocupada por objetos desnecessários, o V8 evita a ocorrência de vazamentos de memória e ajuda a manter a aplicação em execução de forma eficiente. Assim, a coleta de lixo é fundamental para garantir um melhor desempenho e uma maior eficiência nos cenários de alta carga de trabalho e processamento intensivo (V8, 2022).

[8] https://v8.dev/

3.3.2 *JavaScript Libuv (Eventloop)*

A biblioteca `Libuv`[9] é responsável pela assincronicidade do modelo de entradas e saídas não bloqueantes. Seu desenvolvimento teve como objetivo o uso em aplicações `Node.js`, mas atualmente também é utilizada por `Luvit`[10] (sistema de execução assíncrono para a linguagem de programação Lua), `Julia`[11] (linguagem de programação que utiliza a biblioteca em seu sistema de execução, chamado *Julia's Event IO*), `uvloop`[12] (biblioteca assíncrona para Python), e outros. A `Libuv` é uma biblioteca de código aberto que fornece uma camada de abstração sobre o sistema operacional, para que os programas executem operações de entrada e saída de forma simultânea como o gerenciamento de eventos, processos e *threads*. Seu funcionamento é baseado no conceito de *loop* de eventos, que é inicializado quando o `Node.js` inicia (LIANG *et al.*, 2017).

A Figura 8, adaptado de (LIBUV, 2014), apresenta a ordem de operações centrada no *loop* de eventos. O ciclo inicia quando o `Node.js` recebe uma requisição, disparando um evento devido à sua arquitetura orientada a eventos. O conjunto de eventos gera uma fila de requisições, que são solicitações de diversos tipos feitas ao ciclo de eventos do `Node.js`. Uma condicional é inicialmente feita para saber se a requisição é do tipo bloqueante, caso não seja, ela é rapidamente processada e retornada ao usuário. Quando uma tarefa bloqueante é requisitada no ciclo de eventos, ela é enviada ao grupo de *threads* internamente por meio do `Libuv`. Este conjunto de *threads* compartilhadas são necessárias para que não seja preciso fazer o bloqueio do ciclo principal, se tornando eficiente no consumo de recursos. Quando o processamento interno é concluído, é retornado uma *callback* para o ciclo de eventos, que por sua vez envia a resposta para o usuário (ROBERTS, 2014).

---
[9] https://docs.libuv.org/en/v1.x/
[10] https://luvit.io/
[11] https://julialang.org/
[12] https://github.com/MagicStack/uvloop

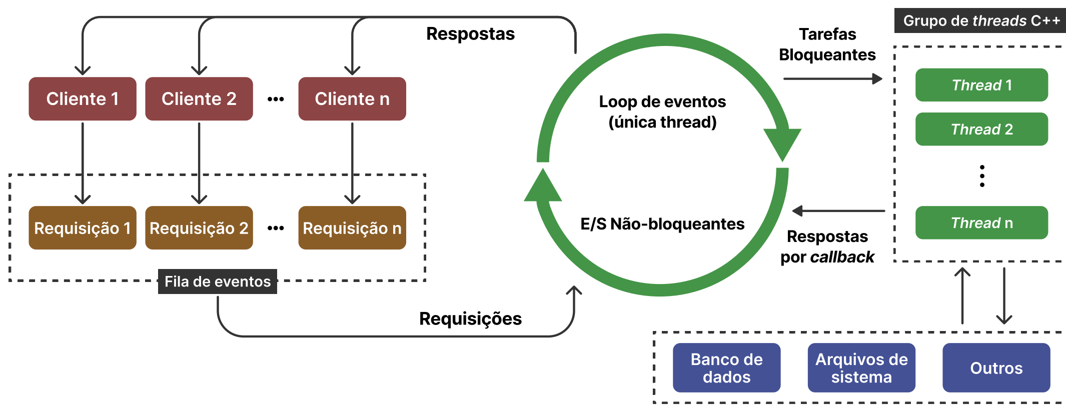

Figura 8 – Diagrama de operações do *loop* de eventos

3.3.3 *Diferença entre JavaScript para front-end e back-end*

Embora a linguagem *JavaScript* possa ser utilizada tanto no *front-end* quanto no *back-end*, existem diferenças importantes na implementação e nas funcionalidades ao desenvolver para a Web e na construção de APIs. A Figura 9 mostra algumas tecnologias que são utilizadas para o desenvolvimento Web, como *React*, *Vue* e *Angular*. No *back-end* temos o `Node.js`, que é a tecnologia em estudo utilizada para a construção de servidores e APIs.

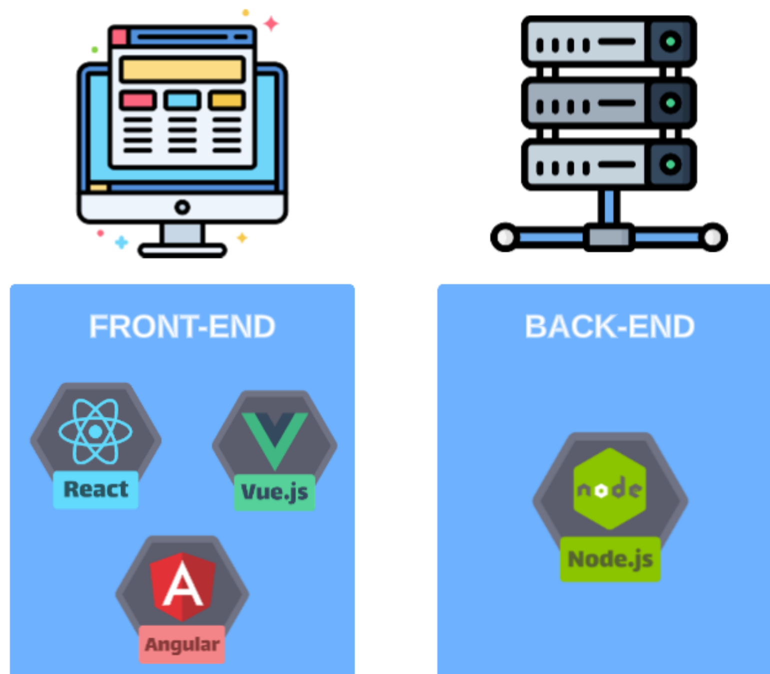

Figura 9 – Diagrama de operações do *loop* de eventos

No desenvolvimento de aplicações para navegador, o *JavaScript* é interpretado e executado em cima de uma máquina virtual. Ele é responsável pela interatividade, manipulação da estrutura de uma página Web, chamada de *Document Object Model* (DOM), envio de requisições e outras tarefas relacionadas ao *front-end*. Já no *back-end*, como no desenvolvimento em `Node.js`, o *JavaScript* é executado fora do navegador, permitindo o acesso a recursos do sistema, arquivos e outros recursos típicos de um servidor. Dessa forma, o *JavaScript* é utilizado para construir aplicações *back-end* completas, como APIs e serviços Web. No lado do cliente, onde o *JavaScript* é comumente utilizado, o navegador fornece a DOM para a manipulação dos objetos, o mesmo acontece para o `V8`, onde é fornecido pela *engine* todos os tipos de dados, operadores, objetos e funções especificados no padrão `ECMAScript` e `CommonJS`.

3.3.4 Ambiente de desenvolvimento para o Node.js

O `Node.js`[13] é uma plataforma de desenvolvimento *cross-plataform*, o que significa que ele pode ser executado em diferentes sistemas operacionais, como Windows, Linux e MacOS. Isso ocorre porque o `Node.js` é baseado em *JavaScript*, uma linguagem de programação que é interpretada pelo navegador ou pelo ambiente de execução do `Node.js`, e não depende de nenhuma plataforma específica.

Para ter um ambiente de desenvolvimento `Node.js`, é necessário fazer a instalação de acordo com o seu sistema operacional por meio do site oficial. Após a instalação, é possível verificar se tudo ocorreu de forma correta digitando "`node --version`" no terminal ou *prompt* de comando do sistema operacional. O comando exibirá a versão do `Node.js` instalada, confirmando se a instalação foi bem-sucedida.

Além disso, é possível utilizar o gerenciador de pacotes do `Node.js`, o *Node Package Manager* (npm), que já vem instalado com o *software*. Para verificar se o npm está instalado corretamente, basta digitar o comando "`npm --version`" no terminal ou *prompt* de comando. O comando exibirá a versão do npm instalada, confirmando se a instalação ocorreu de forma bem-sucedida. O npm é uma ferramenta muito útil para instalar e gerenciar pacotes e bibliotecas necessárias para o desenvolvimento de aplicações `Node.js`.

[13] https://nodejs.org

3.3.5 *Framework Express*

No desenvolvimento de APIs RESTful, o `Node.js` possui o *framework Express*[14], que facilita a criação de aplicações e é um dos mais populares devido à sua facilidade, flexibilidade e eficiência. Ele fornece uma camada de abstração sobre o protocolo HTTP, reduzindo a complexidade e fornecendo uma maneira mais fácil de lidar com rotas, *middlewares* para a manipulação no processo de requisições e respostas, além de permitir integração com outros módulos, como diversos bancos de dados e outros serviços.

O Listing 3.3.1 demonstra um servidor simples em `Node.js` que utiliza o *framework Express*. Nele, é criado uma instância do módulo *Express* e definido uma rota com o método GET na raiz da aplicação. Quando o servidor é iniciado com o método listen() na porta 3000, ele fica disponível para receber solicitações HTTP e uma mensagem é exibida no console indicando que o servidor está rodando na porta especificada. Ao receber uma solicitação por um cliente com o método GET para a rota raiz, a função de *callback* associada é executada, enviando como resposta uma mensagem "Ola, mundo!" para o usuário.

```
1  const express = require('express')
2  const app = express()
3
4  app.get('/', (req, res) => {
5    res.send('Ola, mundo!')
6  })
7
8  app.listen(3000, () => {
9    console.log('Servidor rodando na porta 3000!')
10 })
```

Listing 3.3.1 – Exemplo de um servidor Express

3.3.6 *Exemplo CRUD Node.js com* `Express`

Uma das aplicações mais básicas no desenvolvimento de um projeto com banco de dados é a implementação das operações do CRUD (*Create*, *Read*, *Update* e *Delete*). Nesta seção, será apresentado um projeto que abrange todas as operações básicas de uma API, permitindo a criação, leitura, atualização e exclusão de dados. Para implementar o CRUD em `Node.js`, é necessário criar rotas que possam aceitar solicitações HTTP, como GET, POST, PUT e DELETE,

[14] https://expressjs.com/

e realizar as operações correspondentes no banco de dados. O `Node.js` possui muitas bibliotecas e ferramentas que podem ser utilizadas para implementar operações CRUD, como o `Express` que foi mencionado na Seção 3.3.5.

Para iniciar um projeto em `Node.js` é necessário executar o comando "`npm init -y`" , com isso será criado um arquivo "package.json" na raiz do projeto com as configurações padrões, local onde foi executado o comando. Este arquivo contém as informações do projeto, tais como nome, versão, autor, dependências, *scripts*, entre outras, ele é importante para gerenciar as dependências do projeto e também para compartilhar o mesmo com outros desenvolvedores, facilitando sua instalação e execução.

Para estruturação do projeto, serão instaladas as bibliotecas que auxiliarão no desenvolvimento. A execução do comando "`npm install express mysql2 nodemon dotenv`" faz com que o gerenciador de pacotes instale as seguintes bibliotecas:

- `Express` (obrigatória): *Framework* que contém recursos para a criação de aplicações Web.
- `mysql2` (obrigatória): *Driver* do MySQL para `Node.js` para que seja possível se conectar com um banco de dados MySQL e executar consultas SQL.
- `nodemon` (opcional): Ferramenta que monitora as alterações no código-fonte e reinicia automaticamente o servidor sempre que uma alteração é detectada em tempo de desenvolvimento.
- `dotenv` (opcional): Um pacote que permite carregar variáveis de ambiente a partir de um arquivo .env.

Em um arquivo na raiz do projeto, nomeado de "index.js", será desenvolvido o Listing 3.3.2. Inicialmente, são importados os pacotes express e dotenv. Na constante "app", é instanciado o *framework Express*, assim como a configuração para o *parsing* de JSON que será recebido nas requisições. As rotas do servidor são definidas no arquivo "userRouters.js", apresentado em 3.3.4, onde é importada como "userRoutes" e definida para que todas as requisições feitas na raiz do servidor ("/") sejam tradadas neste módulo separadamente. Por fim, é feita a inicialização do servidor na porta definida nas variáveis de ambiente e é exibida uma mensagem no console para indicar que o servidor está em execução.

Para a comunicação com o banco de dados, será criado na raiz do projeto uma pasta para configurações do servidor nomeada de "config". Nesta pasta será criado um arquivo chamado "database.js", seguindo o Listing 3.3.3. Nele, será feita a instância (através do "mysql2.createConnection") e a conexão com o banco de dados usando as variáveis de ambiente,

```
import express from "express";
import dotenv from "dotenv";
dotenv.config();

import userRoutes from "./routes/userRouters.js";

const app = express();

const port = process.env.PORT || 5000;

app.use(express.json());

app.use("/", userRoutes);

app.listen(port, () => {
	console.log(`Server is listening on port ${port}`);
});
```

Listing 3.3.2 – Arquivo principal de um servidor CRUD

informando o endereço, nome do usuário, senha e nome do banco de dados. Os dados do banco de dados para a conexão seguem os passos apresentados na Seção 3.1.1. Em seguida, a conexão será exportada para que outros arquivos possam utilizá-la para realizar as operações CRUD no banco de dados.

```
import mysql2 from "mysql2";
import dotenv from "dotenv";
dotenv.config();

export const database = mysql2.createConnection({
	host: process.env.DB_HOST,
	user: process.env.DB_USER,
	password: process.env.DB_PASSWORD,
	database: process.env.DB_DATABASE,
});
```

Listing 3.3.3 – Arquivo de conexão com o banco de dados do CRUD

A listagem das rotas é feita no arquivo "userRouters.js" dentro da pasta *routes*, na raiz do projeto, seguindo o Listing 3.3.4, onde o roteador é instanciado usando o método "express.Router()". Em seguida, são definidos os *endpoints* para os métodos GET, POST, PUT e DELETE. Quando um determinado *endpoint* é requisitado, é executada uma função de controle correspondente disponível nos arquivos da pasta *controllers* para cada verbo HTTP.

```
import express from "express";

import { lerUsuarios } from "../controllers/lerUsuarios.js";
import { adicionaUsuario } from "../controllers/adicionaUsuario.js";
import { atualizaUsuario } from "../controllers/atualizaUsuario.js";
import { deletaUsuario } from "../controllers/deletaUsuario.js";

const router = express.Router();

router.get("/", lerUsuarios);
router.post("/", adicionaUsuario);
router.put("/:id", atualizaUsuario);
router.delete("/:id", deletaUsuario);

export default router;
```

Listing 3.3.4 – Arquivo de rotas do CRUD

As funções de controle são feitas em arquivos separados correspondentes a cada método HTTP seguindo o seguinte padrão:

– Arquivo "adicionaUsuario.js": corresponde ao método POST.
– Arquivo "atualizaUsuario.js": corresponde ao método PUT.
– Arquivo "deletaUsuario.js": corresponde ao método DELETE.
– Arquivo "lerUsuario.js": corresponde ao método GET.

A estrutura final do projeto deverá ficar como mostrado na Figura 10.

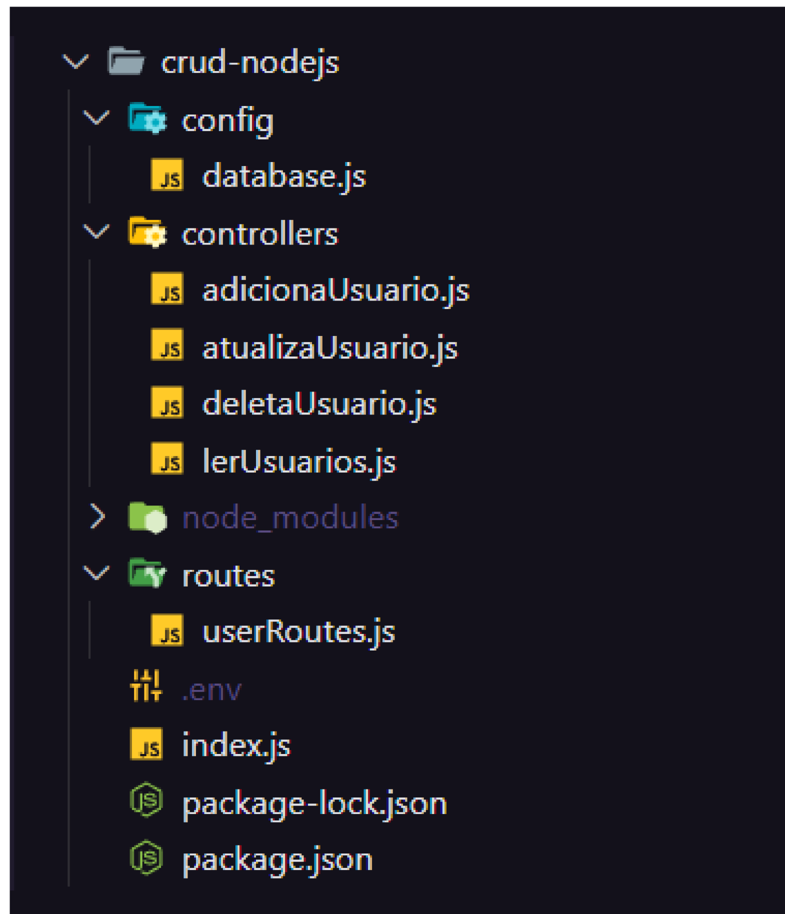

Figura 10 – Funcionamento do motor V8

Os controladores de rota para a implementação do método GET são apresentadas no Listing 3.3.5. Essa função é responsável por processar solicitações de leitura de dados feitas por meio do verbo HTTP, realizar operação de seleção dos registros no banco de dados com a consulta *getQuery* no banco de dados *database* e enviar as respostas apropriadas aos clientes. Um exemplo de utilização do método GET pode ser observado na Figura 11. Nesse exemplo, a requisição feita ao servidor retornará todos os usuários cadastrados no banco de dados em formato JSON, com as informações de id, nome, *email* e idade.

```javascript
import { database } from "../config/database.js";

export const lerUsuarios = (_, res) => {
        const getQuery = "SELECT * FROM usuarios";

        database.query(getQuery, (err, data) => {
                if (err) return res.json(err);

                return res.status(200).json(data);
        });
};
```

Listing 3.3.5 – Arquivo controlador do método GET

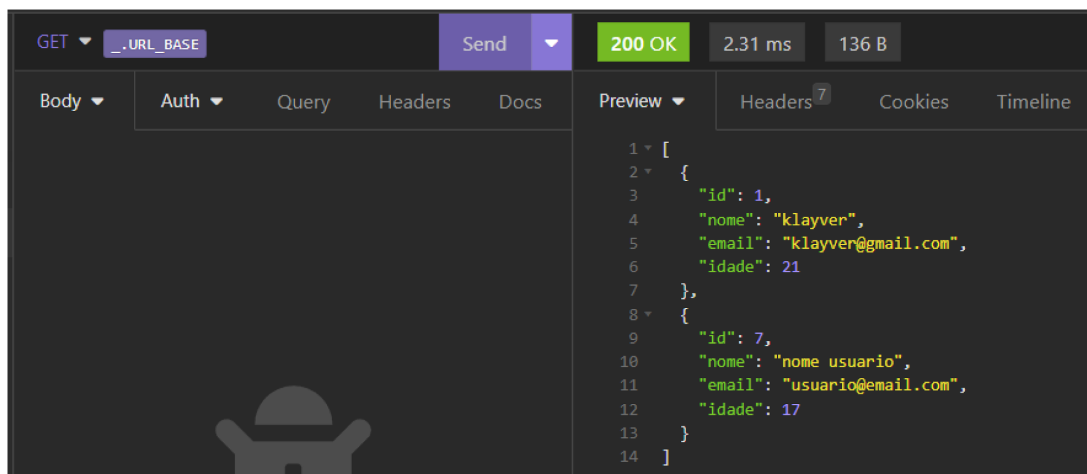

Figura 11 – Utilização do método GET (API Node.js)

No Listing 3.3.6 são apresentadas as funções de controle necessárias para processar solicitações de inserção de dados por meio do verbo HTTP PUT. Essa função realiza a inserção do registro no banco de dados com a consulta *insertQuery* e envia as respostas apropriadas aos clientes. Para tal operação também é possível fazer a validação dos dados enviados pelo cliente para ter uma maior consistência. Um exemplo de como consumir o método POST pode ser visto na Figura 12. Nesse exemplo, a requisição feita ao servidor enviará um JSON com as informações de um novo usuário no corpo da requisição, incluindo nome, *email* e idade. O servidor então fará a inserção do novo registro de usuário no banco de dados e retornará uma mensagem de sucesso ao cliente.

```
import { database } from "../config/database.js";

export const adicionaUsuario = (req, res) => {
        const insertQuery =
                "INSERT INTO usuarios(`nome`, `email`, `idade`)
                ↪  VALUES(?)";

        const valores = [req.body.nome, req.body.email, req.body.idade];

        database.query(insertQuery, [valores], err => {
                if (err) return res.json(err);

                return res.status(200).json("Usuário adicionado com
                ↪  sucesso!");
        });
};
```

Listing 3.3.6 – Arquivo controlador do método POST

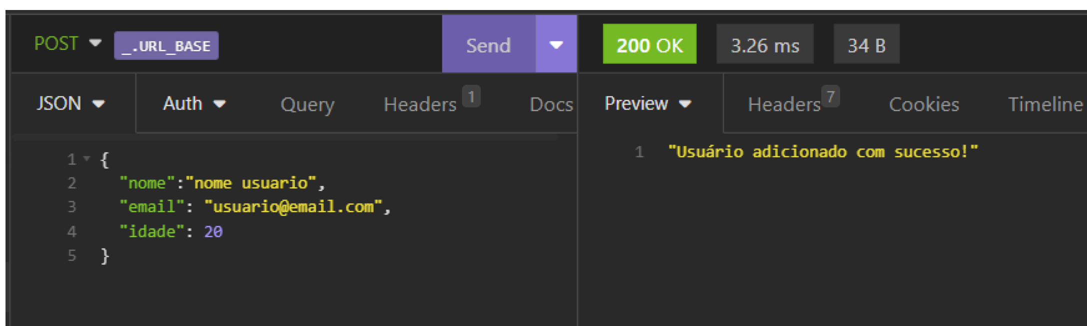

Figura 12 – Utilização do método POST (API Node.js)

No Listing 3.3.7, é apresentada a função de controle necessária para processar solicitações de atualização de dados por meio do verbo HTTP PUT. Essa função é responsável por realizar a atualização do registro no banco de dados por meio da consulta *updateQuery* e enviar uma resposta de sucesso na alteração para o clientes. É possível realizar a validação dos dados enviados pelo cliente para garantir a consistência dos mesmos. Um exemplo de como consumir o método PUT pode ser observado na Figura 13. Nesse exemplo, a requisição feita ao servidor enviará um JSON com as informações de um usuário existente a ser atualizado, incluindo nome, *email* e idade, no corpo da requisição em formato JSON. O servidor então atualizará o registro correspondente no banco de dados de acordo com o id repassado como parâmetro na URL e retornará uma mensagem de sucesso ao cliente caso a operação tenha acontecido de forma correta.

```
1  import { database } from "../config/database.js";
2
3  export const atualizaUsuario = (req, res) => {
4          const updateQuery =
5                  "UPDATE usuarios SET `nome` = ?, `email` = ?, `idade` =
                   ↪ ? WHERE `id` = ?";
6
7          const valores = [req.body.nome, req.body.email, req.body.idade];
8
9          database.query(updateQuery, [...valores, req.params.id], err =>
           ↪ {
10                 if (err) return res.json(err);
11
12                 return res.status(200).json("Usuário atualizado com
                   ↪ sucesso!");
13         });
14 };
```

Listing 3.3.7 – Arquivo controlador do método PUT

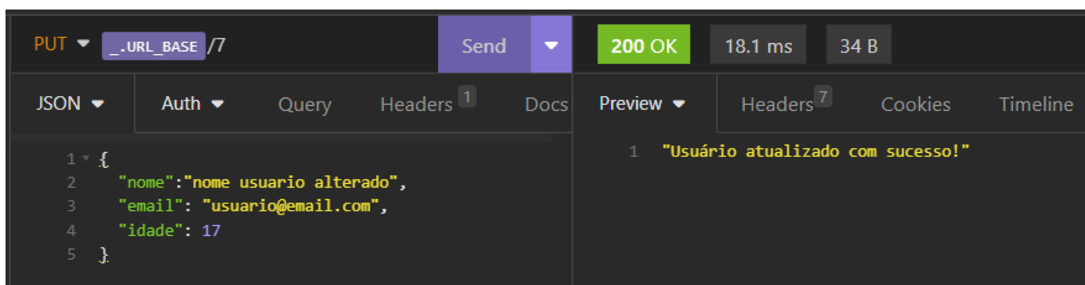

Figura 13 – Utilização do método PUT (API Node.js)

No Listing 3.3.8, é apresentada a função de controle responsável por processar solicitações de exclusão de dados por meio do verbo HTTP DELETE. Essa função é responsável por realizar a exclusão do registro correspondente no banco de dados por meio da consulta *deleteQuery* e enviar uma resposta de sucesso na remoção para o cliente. É possível realizar a validação dos dados enviados pelo cliente para garantir a consistência dos dados, como a verificação para saber se é um registro existente. Um exemplo de como consumir o método DELETE pode ser observado na Figura 14. Nesse exemplo, a requisição feita ao servidor enviará o "id" do usuário a ser excluído como parâmetro na URL. O servidor então excluirá o registro correspondente no banco de dados e retornará uma mensagem de sucesso ao cliente caso a operação tenha acontecido de forma correta.

```
import { database } from "../config/database.js";

export const deletaUsuario = (req, res) => {
        const deleteQuery = "DELETE FROM usuarios WHERE `id` = ?";

        database.query(deleteQuery, [req.params.id], err => {
                if (err) return res.json(err);

                return res.status(200).json("Usuário removido com
                 ↪  sucesso!");
        });
};
```

Listing 3.3.8 – Arquivo controlador do método DELETE

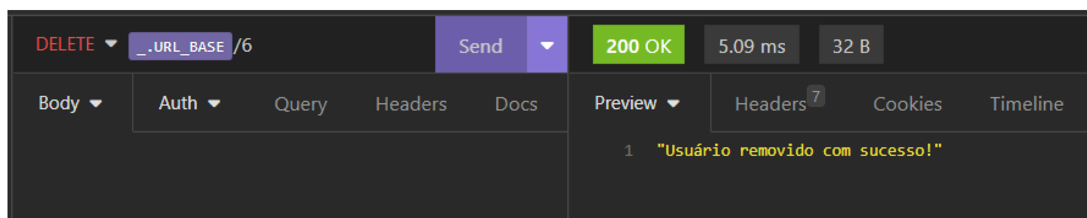

Figura 14 – Utilização do método DELETE (API Node.js)

3.4 Introdução ao Django REST Framework

O `Django`[15] é um *framework* Web *Python* criado em 2003 por Adrian Holovaty e Simon Willison que incentiva o desenvolvimento rápido e um *design* limpo e pragmático, com foco em escalabilidade rápida e flexível, velocidade de desenvolvimento e segurança. Inicialmente foi criado para gerenciar um site jornalístico, mas em 2005 se tornou *open source* com licença *Berkeley Software Distribution* (BSD) (DJANGO, 2022). Muitos desenvolvedores escolhem este *framework* devido à sua organização e simplicidade de aprendizado e instalação. O `Django` também possui outras características atrativas para quem escolhe utilizá-lo no desenvolvimento de uma aplicação. Por exemplo, a velocidade no desenvolvimento, uma vez que oferece estruturas prontas para uso em várias tarefas comuns de desenvolvimento Web, como autenticação de usuários e administração de conteúdo, além de ser um projeto gratuito de código aberto (AMAZON, 2023b).

Esta tecnologia possui vários conceitos no seu desenvolvimento, o que se destaca entre eles é o padrão MVT, que atrelado com a linguagem *Python*, permite a criação de aplicações

[15] https://www.djangoproject.com/

com poucas linhas de código. Devido à sua ampla gama de recursos de desenvolvimento, várias empresas optaram por utilizá-lo em suas aplicações, incluindo a *Mozilla*, *Instagram* e *NASA*. Sua utilização abrange vários cenários, como gerenciamento de servidores Web e bancos de dados, roteamento de URLs, processamento de dados e gerenciamento de eventos. Além disso, a comunidade de desenvolvedores em torno desta tecnologia é bastante ativa, o que contribui para a disponibilidade de recursos e suporte.

Para a construção de APIs RESTful, foi criado em 2013 por Tom Christie, o `Django` REST Framework[16], que fornece algumas funcionalidades e ferramentas adicionais para o desenvolvimento de forma rápida, fácil e eficiente. Está incluso em algumas de suas funcionalidades os pacotes de política de autenticação (OAuth1a e OAuth2) e suporte de serialização de dados, *Object Relational Mapping* (ORM) e não ORM, que serão explicados na Seção 3.4.3. O DRF funciona como um complemento ao `Django`, sendo necessária a sua instalação para a utilização do REST framework (ALURA, 2023). Muitas empresas internacionais utilizam o DRF em suas aplicações, como *Mozilla*, *Red Hat*, *Heroku*, e *Eventbrite* (DRF, 2023).

3.4.1 Padrão MVT

O *Model-View-Template* é um padrão de *design* para desenvolvimento de aplicações Web que separa as responsabilidades de cada componente da aplicação, facilitando o entendimento e manutenibilidade. A Figura 15 apresenta todas as camadas do padrão MVT e como elas estão relacionadas, tanto entre si como com as requisições de um usuário.

[16] https://www.django-rest-framework.org/

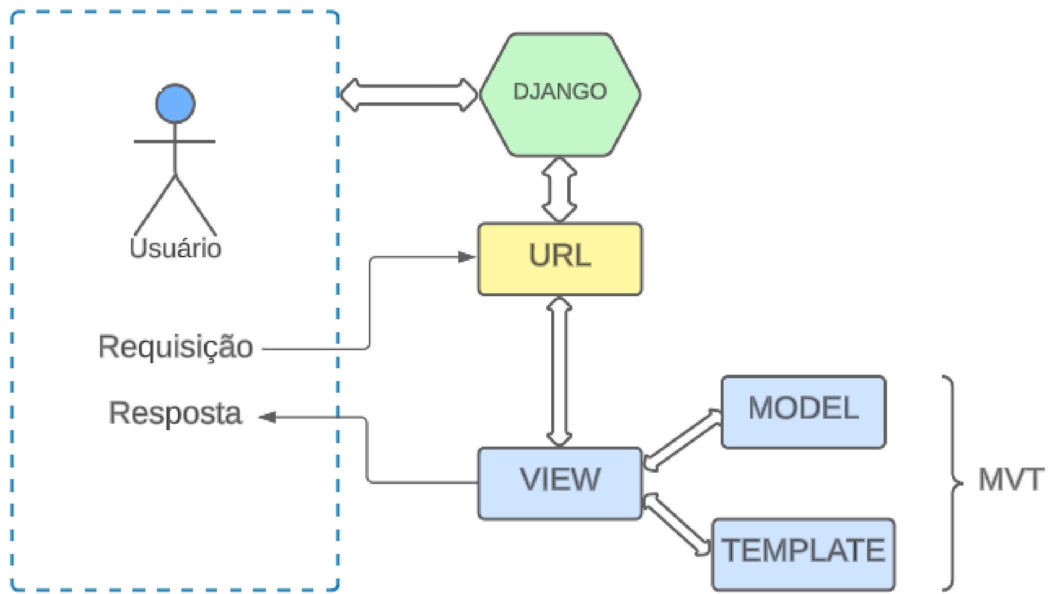

Figura 15 – Representação modelo MVT

O Modelo atua como a interface dos seus dados, sendo responsável pela manutenção e estrutura lógica por trás de toda a aplicação e é representada por um banco de dados (geralmente bancos de dados relacionais como `MySQL`, `PostgreSQL`), definindo como os dados serão armazenados, acessados e manipulados. A Visualização é a interface do usuário vista no navegador quando é renderizado um site. Ela é representada por arquivos HTML/*Cascading Style Sheets* (CSS)/JavaScript (JS), sendo responsável pela lógica do negócio da aplicação e por controlar a interação do usuário. A Visualização recebe as solicitações do usuário, consulta o modelo e renderiza uma resposta para o usuário. O *Template* consiste em partes estáticas da saída HTML desejada, bem como em alguma sintaxe especial que descreve como o conteúdo dinâmico será inserido. Nele será definindo a estrutura e o *layout* da página que é responsável por exibir os dados fornecidos pela visualização (GEEKSFORGEEKS, 2023).

3.4.2 Princípio DRY

Citado no livro "Código Limpo" de Uncle Bob, o princípio DRY é direcionado a não duplicação de código, prezando pela reutilização e facilitando, principalmente, a manutenção e atualização (MARTIN, 2008). Durante o processo de desenvolvimento de uma aplicação, é comum encontrar dificuldades na escrita de um código abstrato. Quando se chega ao ponto de repetir determinados trechos de código, é recomendado verificar se é possível substituir o

código repetido escrevendo e utilizando um código mais abstrato. Nesse sentido, a generalização envolve identificar os aspectos repetidos e, a partir disso, passar a escrever um código mais abstrato (HAOYU; HAILI, 2012).

A aplicação correta do princípio implica diretamente na modularização e reutilização, ou seja, uma modificação em um módulo não implica uma mudança em outro elemento logicamente não relacionado. O objetivo do reuso de pedaços de código permite introduzir relações entre classes no paradigma *Object-oriented Programming* (POO), bem como melhorar a coesão e o acoplamento. Na POO, o DRY é implementado através da criação de classes e objetos que encapsulam comportamentos e propriedades comuns, evitando a repetição desnecessária de código (CABEZAS *et al.*, 2020).

3.4.3 Serialização e deserialização

A serialização é o processo de conversão de dados para facilitar o consumo pelas aplicações. No contexto do Django REST *Framework*, a serialização se refere à conversão de objetos do *Python* em formatos de dados comuns, como XML, *YAML Ain't Markup Language* (YAML) ou o mais utilizado, JSON. O `Django` fornece um módulo de serialização padrão que pode ser usado para serializar dados em vários formatos, mas o DRF oferece recursos avançados de serialização que simplificam e agilizam o processo de conversão de objetos *Python* em dados com formato compatível com a Web.

Várias classes de serialização são oferecidas pelo DRF para personalizar a serialização dos dados, como para modelos, listas e *hyperlinked*, de acordo com a necessidade da aplicação. Além de incluir um conjunto completo de campos de serialização personalizados para tipos de dados comuns, como datas, horas e horas de data. O DRF também inclui a funcionalidade de validação de dados. O Serializador pode ser configurado para validar dados de entrada e fornecer mensagens de erro detalhadas caso ocorra algum problema. Isso ajuda a garantir que a entrada do usuário seja sempre válida e consistente. O Listing 3.4.1 apresenta um exemplo de serialização nos campos definidos tomando como exemplo os dados de um comentário, como o *email*, conteúdo e data de criação.

3.4.4 Ambiente de desenvolvimento para o Django REST Framework

No desenvolvimento em Django REST *Framework*, a escolha do sistema operacional é uma questão de preferência pessoal do desenvolvedor, pois o DRF pode ser executado em

```python
from rest_framework import serializers

class CommentSerializer(serializers.Serializer):
    email = serializers.EmailField()
    content = serializers.CharField(max_length=200)
    created = serializers.DateTimeField()

serializer = CommentSerializer(comment)
serializer.data

>> {'email': 'leila@example.com', 'content': 'foo bar', 'created':
↪  '2022-05-27T15:17:10.375877'}
```

Listing 3.4.1 – Exemplo serialização de dados

vários sistemas operacionais, desde que seja compatível com a versão do *Python* utilizada para o desenvolvimento. Para criar um ambiente de desenvolvimento adequado, é necessário instalar o *Python*, que pode ser baixado por meio do site oficial[17]. O site disponibiliza tanto a versão mais atualizada quanto versões específicas anteriores. Após a instalação, é possível verificar a versão do *Python* instalada por meio do comando "`python --version`" no console do terminal de comando.

Após a instalação do *Python*, é recomendável criar um ambiente virtual para o desenvolvimento, o que ajuda a evitar conflitos entre as dependências de projetos e bibliotecas instaladas no sistema, além de permitir ter várias versões do *Python* instaladas para projetos específicos. A biblioteca 'venv' já vem instalada com o *Python*, e para criar um ambiente virtual, basta executar o comando "`python -m venv myenv`" no terminal. Para ativar o ambiente virtual no sistema operacional Windows, por exemplo, é necessário executar o comando "`myenv\Scripts\activate`" no terminal.

A instalação do `Django` pode ser feita por meio do gerenciador de pacotes nativo do *Python*, o 'pip', que já é incluído na instalação padrão do *Python*. Para instalar o *framework*, basta executar o comando 'pip install django' no terminal. Para criar um novo projeto no `Django`, basta executar o comando "`django-admin startproject nome_do_projeto`" no terminal.

3.4.5 Exemplo CRUD Django REST Framework

Contextualmente, a criação de uma API RESTful com o DRF envolve primeiramente a definição dos modelos de dados, que representam os objetos de uma aplicação que serão

[17] https://www.python.org/downloads/

expostos pela API. Após a criação dos modelos é preciso definir as operações que podem ser realizadas nesses objetos com base na arquitetura REST, contendo os métodos para leitura, criação, atualização e exclusão. Esta definição é feita utilizando as visualizações do DRF, que definem como os dados serão manipulados e retornados em resposta às solicitações HTTP. Para tornar os dados acessíveis pela API, é necessário converter os objetos do modelo em um formato serializado, que pode ser JSON ou outro formato compatível com a Web. O DRF fornece uma camada de serialização, que permite definir como os objetos do modelo serão serializados e desserializados. Este processo é feito usando os serializadores do DRF, que transformam os objetos em dados serializados e vice-versa.

Para iniciar um projeto CRUD no DRF é necessário instalar, por meio do gerenciador de pacotes pip no terminal, os pacotes que serão utilizadas com o comando "`pip install django djangorestframework mysqlclient`", onde será instalado os seguintes pacotes:

- `django` (obrigatório): *Framework* que contém recursos para a criação de aplicações Web.
- `djangorestframework` (obrigatório): Extensão do DRF que simplifica o desenvolvimento de APIs RESTful.
- `mysqlclient` (obrigatório): *Driver* do MySQL para o Python, permite a conexão com um banco de dados MySQL e a execução de consultas SQL.

Para um projeto API RESTFul de controle de usuários, a inicialização é feita com o comando "`django-admin startproject crud_drf_cbv`". Com este comando será criada uma pasta como o nome do projeto dentro do diretório atual, isto evita o conflito com módulos externos. Além disso, um arquivo principal com nome "`manage.py`" também será gerado, contendo as informações primárias do projeto. Dentro do projeto `Django` criado anteriormente, será utilizado o comando "`python manage.py startapp usuarios`" para iniciar um aplicativo com nome "usuarios", que ajudará na modularização do projeto. A estrutura do projeto e aplicação deverá seguir o modelo da Figura 16, onde a aplicação gerada fica dentro do projeto principal.

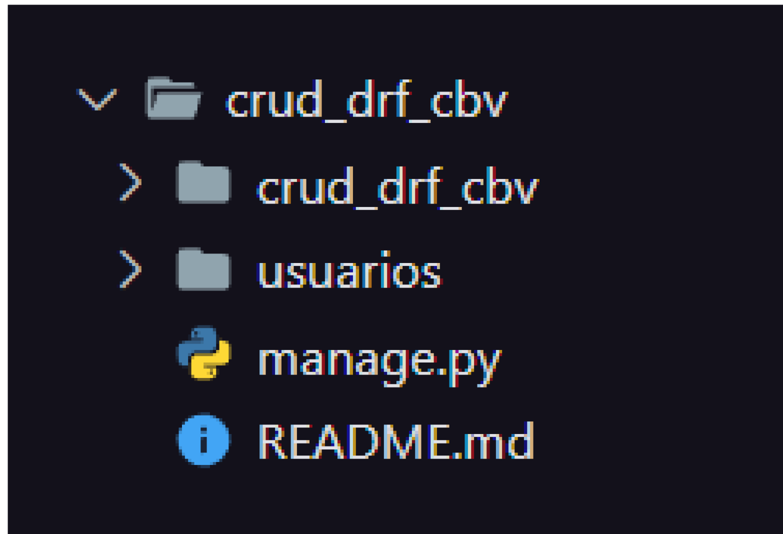

Figura 16 – Estrutura de arquivos inicial

Com a inicialização do projeto e aplicação feitas, é preciso fazer as configurações iniciais. Na raiz do projeto, no arquivo "`settings.py`", será inserido o "rest_framework" e a aplicação (neste exemplo o nome "usuarios") na variável "INSTALLED_APPS". Este processo é necessário para a indicação ao `Django` que a aplicação faz parte do projeto e que deve ser carregada quando o mesmo for iniciado. O "rest_framework" define a dependência do projeto em relação ao DRF, onde o `Django` verificará se o DRF está instalado no ambiente *Python* antes de iniciar o projeto, exibindo um erro caso o DRF não esteja instalado. Além disso, o `Django` carregará automaticamente os arquivos de configuração do DRF, como as definições de roteamento de URL. A estrutura da variável "INSTALLED_APPS" deve ser semelhante ao Listing 3.4.2.

```
INSTALLED_APPS = [
    # outras importações
    "rest_framework",
    "usuarios",
]
```

Listing 3.4.2 – Estrutura da variável "INSTALLED_APPS"

No mesmo arquivo, na variável "DATABASES", será feita as configurações de conexão com o banco de dados. Nela será definido os parâmetros de conexão, como nome, usuário, senha, *host*, porta e *engine*. Os dados do banco de dados para a conexão seguem os passos apresentados na Seção 3.1.1. Esta configuração de conexão deve seguir o modelo do

Listing 3.4.3.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'mydatabase',
        'USER': 'mydatabaseuser',
        'PASSWORD': 'mypassword',
        'HOST': 'localhost',
        'PORT': '3306',
    }
}
```

Listing 3.4.3 – Configuração do banco de dados

Após a configuração do projeto e conexão com o banco de dados, é feito a criação do modelo que a API utilizará para executar as operações CRUD. No `Django`, um modelo é utilizado para gerar as tabelas de banco de dados, colunas, relacionamentos entre as tabelas e várias restrições, fornecendo também modelos já prontos, como usuários de autenticação. Dentro do arquivo "models.py", localizado na raiz do diretório da aplicação criada nos passos anteriores, será criado uma classe "Usuario". O Listing 3.4.4 apresenta a estrutura da mesma com a definição dos campos de nome, *email* e idade. A classe Meta é usada para definir metadados adicionais sobre o modelo, em que pode ser especificado, por exemplo, o nome da tabela do banco de dados a ser criada, que terá o nome "usuarios_cbv". O método "__str__" é definido para representar o objeto do modelo como uma *string*. Neste caso, é retornado o nome do usuário.

```
from django.db import models

class Usuario(models.Model):
    nome = models.CharField(max_length=255)
    email = models.EmailField(max_length=45)
    idade = models.IntegerField()

    class Meta:
        db_table = 'usuarios_cbv'

    def __str__(self):
        return self.nome
```

Listing 3.4.4 – Configuração modelo Usuario

O `Django` inclui um painel administrativo que permite gerenciar o conteúdo de

uma aplicação. Esse painel é criado automaticamente na criação de um projeto. Dentro do arquivo "admin.py", na raiz da aplicação "usuarios", é feita a configuração de registro dos modelos seguindo o Listing 3.4.5. Ele é responsável por registrar o modelo "Usuario", criado no Listing 3.4.4 para que seja gerenciado por meio do painel administrativo. Isso significa que, ao acessar o painel, haverá uma seção para gerenciamento de usuários, onde será possível listar, criar, atualizar e deletar registros da tabela "usuarios", apresentado na Figura 21.

```
from django.contrib import admin
from .models import Usuario

admin.site.register(Usuario)
```

Listing 3.4.5 – Configuração modelo Usuário na página admin

Como explicado na Seção 3.4.3, é necessário um processo de serialização para fazer a conversão de objetos ou lista de objetos retornados pelo banco de dados em tipos de dados compatíveis com a estrutura Web, como o JSON. Para estas configurações será feito um arquivo na raiz da aplicação nomeado de "serializers.py". Neste projeto foi utilizado o "ModelSerializer" para fazer a serialização, como apresenta o Listing 3.4.6. Na classe 'Meta' é definido o modelo com o qual o serializador deve trabalhar, e são especificados quais campos do modelo serão serializados. Neste caso, o serializador irá retornar um JSON contendo o id, nome, *email* e idade de um usuário, correspondendo aos campos do modelo criado no Listing 3.4.4.

```
from rest_framework import serializers
from .models import Usuario

class UsuarioSerializer(serializers.ModelSerializer):
    class Meta:
        model = Usuario
        fields = ['id', 'nome', 'email', 'idade']
```

Listing 3.4.6 – Configuração serializador Usuário

Os processamentos das solicitações HTTP no DRF são processadas na camada de visualização, que recebem as requisições. Após o processamento, a API retorna as respostas correspondentes. No DRF existem dois tipos principais de *views*, são as baseadas em classes, que oferecem recursos de *mixins* e as *views* baseadas em funções, que são mais simples e flexíveis. Neste projeto será utilizado o modelo baseado em classes, dentro do arquivo "views.py" locali-

zado na raiz da aplicação. O equivalente também pode ser feito para visualizações baseadas em funções.

O Listing 3.4.7 implementa a classe "ListaUsuario", que é uma subclasse da classe APIView. As duas operações de CRUD, "listar" e "criar", são definidas por meio dos métodos "get" e "post", referenciando os métodos HTTP GET e POST. No método "get", a *view* busca todos os objetos do tipo "Usuario" do banco de dados utilizando a classe "Usuario.objects.all()" e, em seguida, serializa esses objetos utilizando a classe "UsuarioSerializer", criado no Listing 3.4.6. O argumento "many=True" passado ao serializador indica que há vários objetos a serem serializados, em vez de um único objeto. Já no método "post", a *view* recebe uma requisição com os dados do novo usuário a ser adicionado, que são validados pelo serializador "UsuarioSerializer". Caso os dados sejam válidos, o novo usuário é salvo no banco de dados com o método "save()", e uma mensagem de sucesso é retornada como resposta com o código de status 201. Caso contrário, uma mensagem de erro com o código de status 400 é retornada.

```
1  class ListaUsuario(APIView):
2      def get(self, request, format=None):
3          usuario = Usuario.objects.all()
4          serializer = UsuarioSerializer(usuario, many=True)
5          return Response(serializer.data)
6
7      def post(self, request, format=None):
8          serializer = UsuarioSerializer(data=request.data)
9
10         if serializer.is_valid():
11             serializer.save()
12             return Response({'mensagem': 'Usuário adicionado com
                ↪  sucesso!'}, status=status.HTTP_201_CREATED)
13
14         return Response(serializer.errors,
            ↪  status=status.HTTP_400_BAD_REQUEST)
```

Listing 3.4.7 – Configuração da *views* dos métodos GET e POST

A implementação dos verbos HTTP PUT e DELETE são implementados na "DetalhesUsuario", que também é uma subclasse da APIView, nela são feitas as definições dos métodos para atualizar e excluir um usuário, como apresentado no Listing 3.4.8. O método "get_object(pk)" é uma função auxiliar para recuperar o usuário com o id correspondente a pk (*primary key*). Se não houver usuário com o id fornecido, ele retornará um erro 404. O método "put" recebe a requisição HTTP PUT com dados JSON atualizados para um usuário

e, em seguida, valida os dados usando o serializador "UsuarioSerializer". Se os dados forem válidos, o serializador salva as atualizações no banco de dados e retorna uma mensagem de sucesso. Se os dados não forem válidos, o método retornará uma mensagem de erro 400. O método "delete" recebe uma requisição HTTP DELETE e remove o usuário correspondente ao id pk. Caso o usuário exista, é retornado um status 204.

```
class DetalhesUsuario(APIView):
    def get_object(self, pk):
        try:
            return Usuario.objects.get(pk=pk)
        except Usuario.DoesNotExist:
            raise Http404

    def put(self, request, pk, format=None):
        usuario = self.get_object(pk)
        serializer = UsuarioSerializer(usuario, data=request.data)

        if serializer.is_valid():
            serializer.save()
            return Response({'mensagem': 'Usuário atualizado com
            ↪  sucesso!'})

        return Response(serializer.errors,
        ↪  status=status.HTTP_400_BAD_REQUEST)

    def delete(self, request, pk, format=None):
        usuario = self.get_object(pk)
        usuario.delete()

        return Response(status=status.HTTP_204_NO_CONTENT)
```

Listing 3.4.8 – Configuração das *views* dos métodos GET e POST

A listagem das rotas da aplicação é feita seguindo o Listing 3.4.9, que são implementadas em um arquivo "`urls.py`" dentro da raiz aplicação. Para a configuração das rotas globais do projeto, as mesmas são configuradas dentro do arquivo "`urls.py`" no diretório raiz do projeto, seguindo o Listing 3.4.10. As rotas criadas da aplicação são:

1. "usuarios/": esta rota aponta para a classe de visualização "ListaUsuario" definida no arquivo "`views.py`" do aplicativo usuarios. Quando um usuário acessa essa rota usando o método GET, todos os usuários cadastrados no banco de dados são retornados como resposta em formato JSON, como apresenta a Figura 17. Quando um usuário usa o método

POST, um novo usuário é adicionado ao banco de dados de acordo com as informações passadas no corpo da requisição, como mostra a Figura 18.

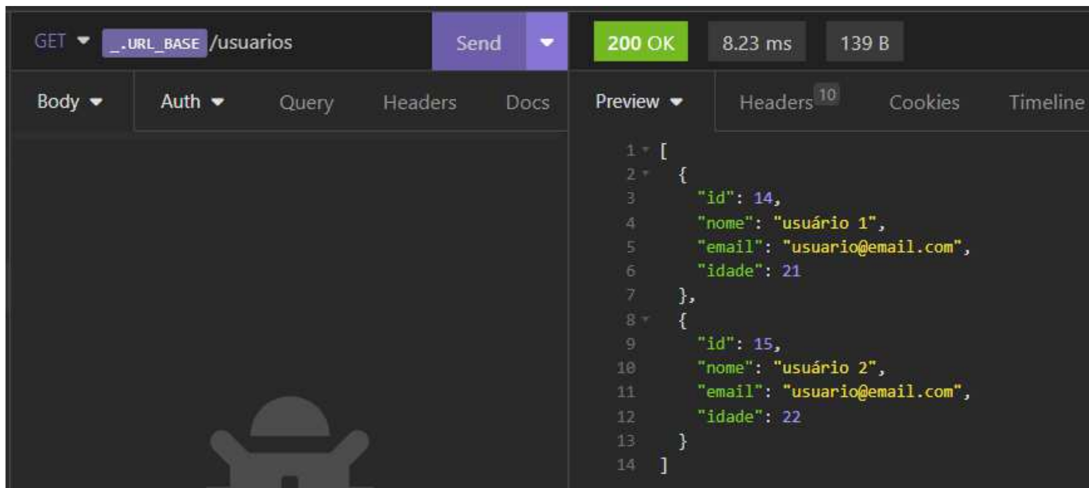

Figura 17 – Utilização do método GET (API Django REST)

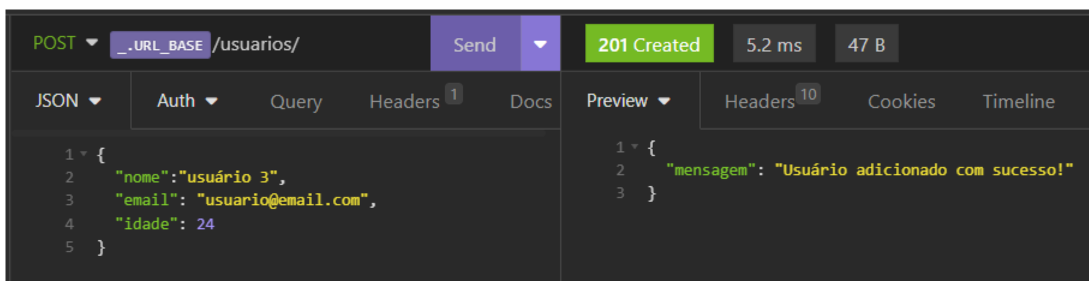

Figura 18 – Utilização do método POST (API Django REST)

2. "usuarios/<int:pk>/": esta rota aponta para a classe de visualização "DetalhesUsuario" definida no arquivo "`views.py`" do aplicativo usuarios. Quando um usuário acessa essa rota usando o método PUT, os dados do usuário com o ID fornecido são atualizados com os novos dados fornecidos no corpo da solicitação, como mostra a Figura 19. Quando um usuário usa o método DELETE, o usuário com o ID fornecido é excluído do banco de dados, apresentado na Figura 20.

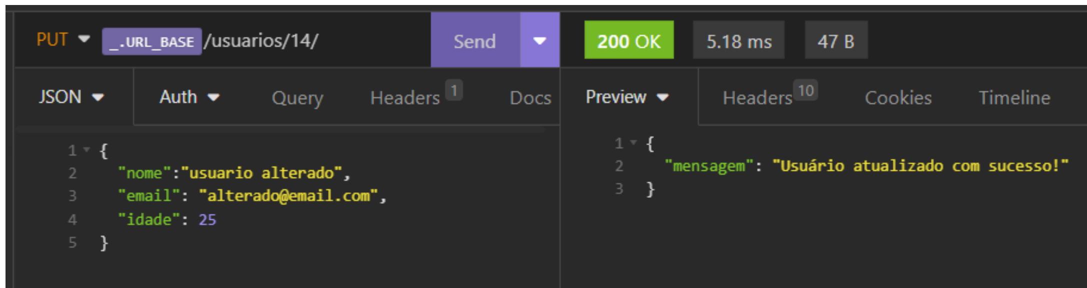

Figura 19 – Utilização do método PUT (API Django REST)

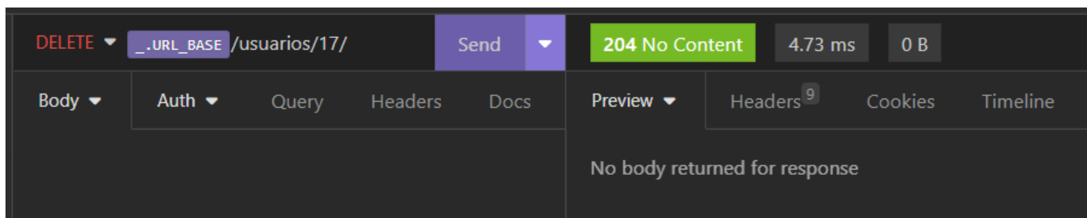

Figura 20 – Utilização do método DELETE (API Django REST)

```
from django.urls import path
from usuarios import views

urlpatterns = [
    path('usuarios/', views.ListaUsuario.as_view()),
    path('usuarios/<int:pk>/', views.DetalhesUsuario.as_view()),
]
```

Listing 3.4.9 – Configuração das URLs da aplicação

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('usuarios.urls')),
]
```

Listing 3.4.10 – Configuração das URLs do projeto

Para fazer a sincronização do banco de dados com as alterações feitas nos modelos do aplicativo, é utilizado o comando "`python manage.py makemigrations`". Esse comando cria arquivos de migração de acordo com os modelos no diretório "*migrations*" do aplicativo.

Em seguida, os arquivos de migração criados precisam ser aplicados ao banco de dados para atualizá-lo, esta migração é feita executando o comando "`python manage.py migrate`". Por padrão, também são criadas as migrações de autenticação (*auth*) que incluem as tabelas do banco de dados para gerenciar usuários, grupos e permissões. Além disso, as migrações de *admin* são criadas, incluindo as tabelas de banco de dados para suportar o painel de administração do `Django`. Ao executar o comando "`python manage.py runserver`", é possível acessar a API tanto por meio do cliente Web quanto pelo painel administrativo do `Django`. Isso permite realizar as operações necessárias, conforme mostrado na Figura 21.

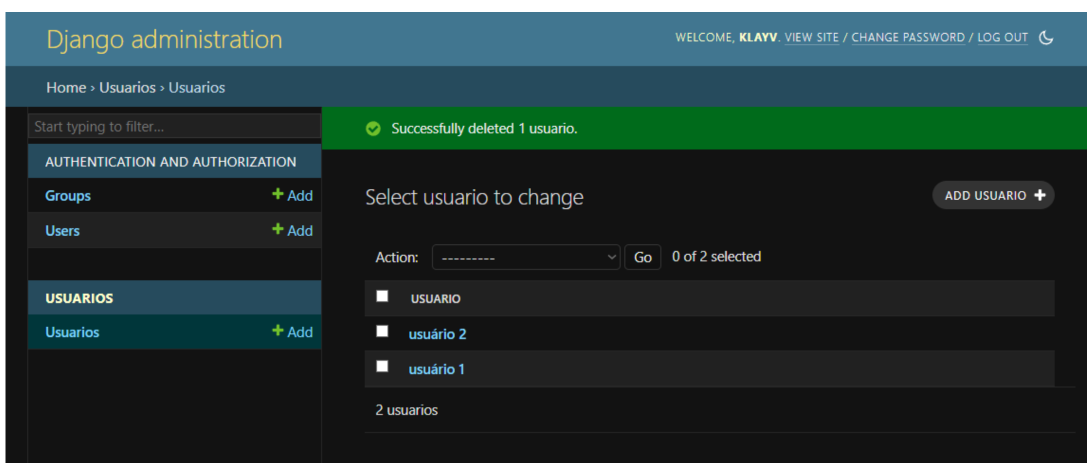

Figura 21 – Painel administrativo do Django

3.5 Introdução ao .NET

O ecossistema do .NET[18] começou em 2002 com o lançamento da *Microsoft* de uma plataforma desenvolvimento para criar aplicativos do *Windows*, chamado de `.NET Framework`. Em 2014 foi introduzido o `.NET Core`, um sucessor multiplataforma e de software livre para o `.NET Framework`, utilizado para aplicativos *Linux*, *macOS* e *Windows*. A plataforma `.NET` além dos sistemas operacionais citados, também dá suporte em ambientes emulados, além da possibilidade de ser utilizado em arquiteturas Arm64, x64 e x32. Os aplicativos e as bibliotecas do .NET podem ser criados utilizando tanto a *Common Language Infrastructure* (CLI) do `.NET` ou um Ambiente de Desenvolvimento Integrado (IDE) como o Visual Studio.

Os programas em *C#* são executados no `.NET` em um sistema de execução virtual chamado *Common Language Runtime* (CLR). O CLR é o *runtime* presente em todos os aplicativos `.NET`, tendo como recursos fundamentais a coleta de lixo, segurança de memória, suporte de alto

[18] https://learn.microsoft.com/pt-br/dotnet/fundamentals/

nível para linguagens de programação e design multiplataforma. Este *runtime* é compatível com várias linguagens de programação com suporte da *Microsoft*, como *C#*, *F#* e *Visual Basic*, além de ser multiplataforma. A característica de ser comportado para diferentes sistemas operacionais e arquiteturas dá a possibilidade de não precisar ser recompilado para ser executado em novos ambientes, necessitando apenas da instalação de um *runtime* diferente para executar o aplicativo.

Por ser uma linguagem orientada a objetos, o *C#* segue o paradigma de programação orientada a objetos POO. Sua construção e desenvolvimento são baseados em objetos, que são divididos em classes, e se comunicam entre si por meio de métodos. É possível a utilização de vários conceitos deste paradigma, como a herança, encapsulamento e polimorfismo, no desenvolvimento em como *C#*. O POO permite a modularização, reutilização e facilidade de manutenção do código. Além disso, o *C#* também suporta outros paradigmas, como o paradigma funcional e a programação assíncrona (MICROSOFT, 2023).

A execução das aplicações em como *C#* do `.NET` acontecem por meio de uma compilação em uma Linguagem intermediária (IL), que é um formato de código compactado com suporte em qualquer SO e arquitetura. O `.NET` dá suporte a modelos de compilação *Ahead-Of-Time* (AOT) e JIT, como por exemplo na compilação padrão JIT nas plataformas *Android*, *macOS* e *Linux*. O compilador JIT tem a capacidade de adaptar o código para funcionar melhor em um ambiente específico, levando em consideração o sistema operacional e o *hardware* do dispositivo em que o aplicativo está sendo executado. O AOT proporciona uma inicialização mais rápida de um aplicativo, visto que faz a compilação antes da execução. Este compilador possui como desvantagem a necessidade de compilar o aplicativo em diferentes sistemas operacionais e arquiteturas de *hardware* (MICROSOFT, 2023).

3.5.1 Gerenciador de pacotes NuGet e Entity Framework Core

NuGet[19] é o mecanismo com suporte da *Microsoft* para compartilhamento de código que possui as definições de como os pacotes do `.NET` são criados, hospedados e consumidos, além de fornecer as ferramentas para cada uma dessas funções. Por meio da ferramenta `NuGet` os desenvolvedores podem compartilhar pacotes em *hosts* públicos ou privado e consumi-los em seus projetos.

[19] https://learn.microsoft.com/pt-br/nuget

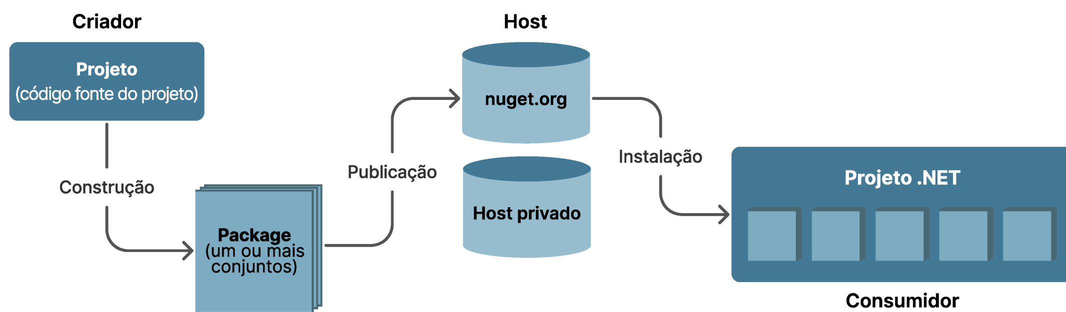

Figura 22 – Disponibilização e consumo de pacotes por meio do NuGet

O *Entity Framework* (EF)[20] é um *framework* de mapeamento objeto-relacional ORM. O *Entity Framework* Core[21] é uma versão mais leve e flexível do *Entity Framework*. Ambos foram criados pela *Microsoft* e são utilizados para o acesso e a manipulação de dados em um banco. O *Entity Framework Core* (EFC) é compatível com vários tipos de banco de dados, como `SqlServer`, `PostgreSQL` e `MySql`. No processo de desenvolvimento, o EFC também é utilizado como um *framework* ORM, onde é possível fazer o mapeamento automático das classes e propriedades dos modelos de dados para as tabelas e colunas do banco de dados, sem a necessidade de escrever código SQL manualmente. O *framework* não apenas oferece suporte à linguagem de consulta SQL, mas também permite o uso da linguagem de consulta *Language Integrated Query* (LINQ). Além disso, ele simplifica o processo de alteração no banco de dados através das migrações, proporcionando atualizações mais eficientes e organizadas.

3.5.2 Ambiente de desenvolvimento para o ASP.NET Core

Assim como nos outros *frameworks* mencionados, o ambiente para desenvolvimento em *C#* pode ser configurado em vários sistemas operacionais, incluindo *Windows*, *MacOS* e *Linux*. Há vários IDEs disponíveis para o desenvolvimento em *C#*, tais como Visual Studio Code, JetBrains Rider e Visual Studio, sendo este último o mais utilizado, pois permite a implantação e depuração de aplicações. Para desenvolver em *C#*, é necessário ter instalado o SDK do `.NET`, que inclui o compilador *C#* e as bibliotecas necessárias para criar aplicativos `.NET`.

O SDK é um conjunto de bibliotecas e ferramentas que permite aos desenvolvedores criar e desenvolver aplicações e bibliotecas. Ele pode ser baixado e instalado a partir do site oficial da *Microsoft*[22], de acordo com o sistema operacional. A Figura 23 mostra a página inicial

[20] https://learn.microsoft.com/pt-br/ef/
[21] https://learn.microsoft.com/pt-br/ef/core/
[22] https://dotnet.microsoft.com/pt-br/download

do instalador, onde ao clicar em "*Install*", será instalado todas as dependências necessárias. O SDK contém alguns componentes que são utilizados para a criação e execução das aplicações, como o `.NET` CLI, *runtime*, as bibliotecas .NET e o *driver* 'dotnet'. Após o *download* e instalação, é possível verificar se ela foi realizada com sucesso executando o comando "`dotnet -version`" no console do terminal, no qual deverá aparecer a versão do SDK instalado.

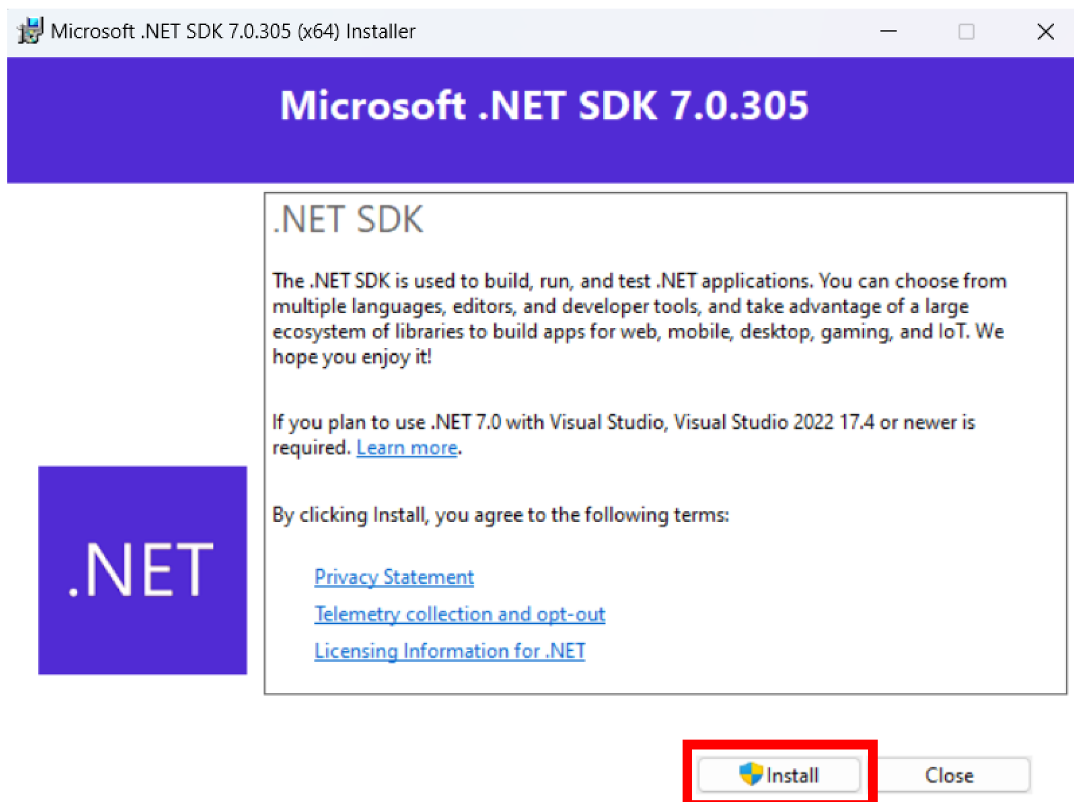

Figura 23 – Página inicial do instalador do SDK

3.5.3 Exemplo CRUD ASP.NET Core

A criação de um CRUD em `ASP.NET Core` pode ser feito de duas maneiras, nas quais podemos ter um maior controle sobre a implementação, tendo maior personalização na criação manual de todas as partes do CRUD, como nos modelos, controladores e rotas, suprindo as necessidades específicas de cada projeto. Contudo, é maior a demanda de tempo para o desenvolvimento e esforço na criação de todas as partes de uma forma individual. Existem também geradores de códigos para auxiliar o processo de desenvolvimento, que tem uma geração de código automático baseado nos modelos de dados e nas definições do banco de dados. A utilização do Visual Studio requer a instalação do ambiente para desenvolvimento `ASP.NET Core` que é fornecido na inicialização do editor, como mostra a Figura 24 com a opção destacada

em vermelho.

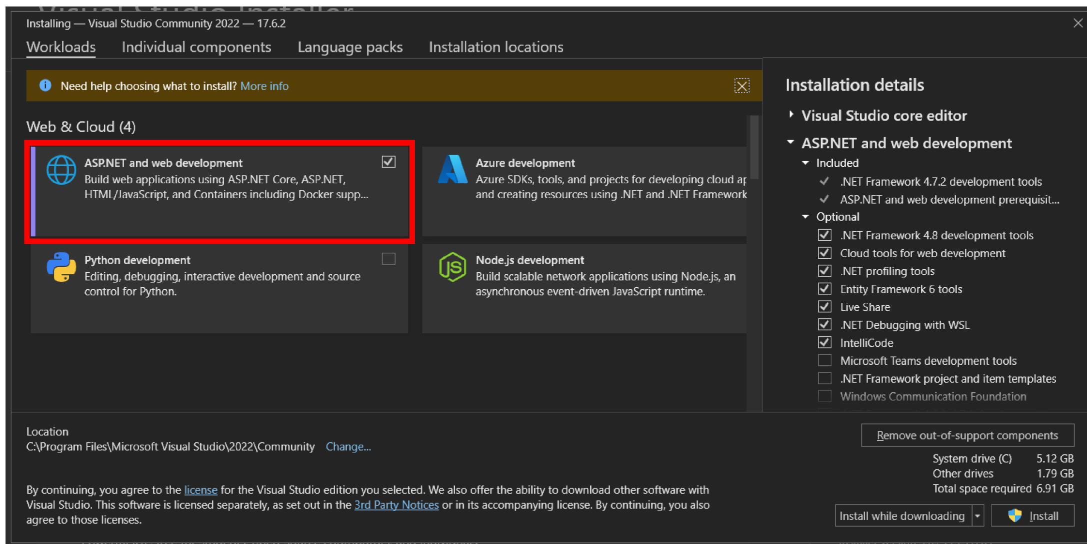

Figura 24 – Instalação do ambiente ASP.NET

A inicialização do projeto é feita após a instalação do ambiente ASP.NET, no qual é possível criar um projeto baseado em vários *templates*. O CRUD será feito baseado no *template* ASP.NET Core Web API, que vem com um exemplo já pronto de um serviço RESTful HTTP, além de também ter a possibilidade de ser utilizado para ASP.NET Core *Model View Controller* (MVC). Após a escolha do *template* é necessário fazer as configurações iniciais do projeto, como escolha do nome, diretório e versão do .NET, que foi escolhida a 7.0, além de configuração de autenticação, Docker e outros. Quando o projeto é criado, uma estrutura já é apresentada com um exemplo inicial, como é apresentado na Figura 25.

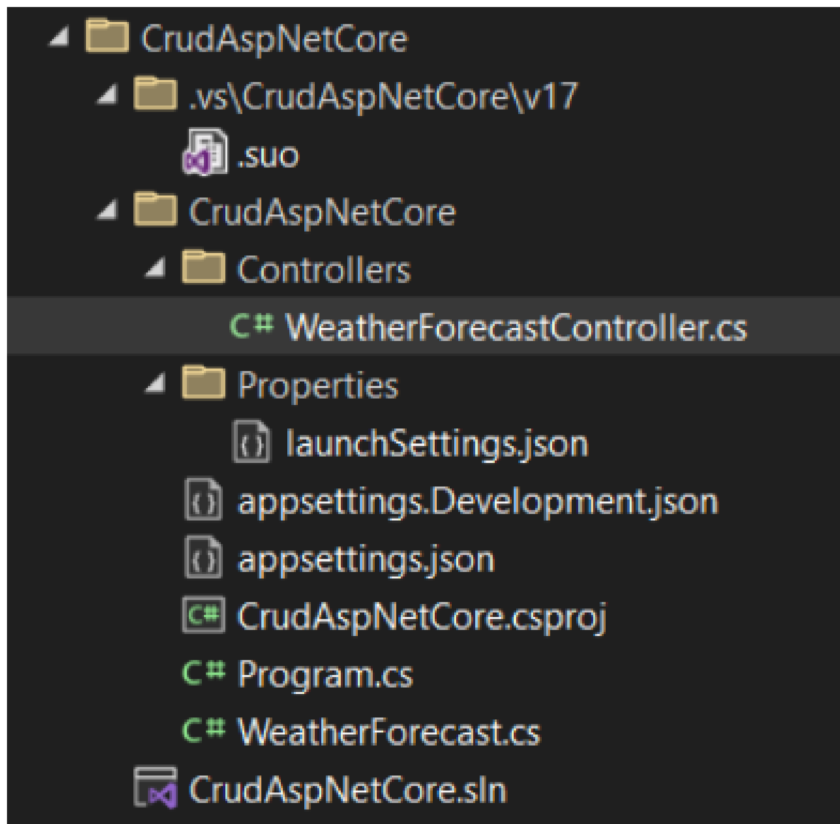

Figura 25 – Árvore de arquivos inicial

Utilizando o gerenciador de pacotes `NuGet`, citado na Seção 3.5.1, será instalado os seguintes pacotes obrigatórios:

- "Microsoft.EntityFrameworkCore": fornece as funcionalidades básicas do *Entity Framework Core*, como a definição de classes de entidade, contexto de banco de dados, configurações de relacionamento e recursos para consultas e atualizações no banco de dados. Ele é necessário para usar o EFC no projeto.
- "Microsoft.EntityFrameworkCore.Design": contém as ferramentas de *design* do *Entity Framework Core*, que são usadas para criar e modificar o modelo de entidade. Ele fornece funcionalidades, como *scaffolding* (geração de código a partir do banco de dados), *Migrations* (gerenciamento de esquema do banco de dados) e outras ferramentas úteis para o desenvolvimento com o EF Core.
- "Microsoft.EntityFrameworkCore.Tools": contém comandos da CLI do EF Core, permitindo a execução de tarefas relacionadas ao banco de dados, como a criação de migrações, aplicação de migrações, *scaffolding*, entre outras operações. Essas ferramentas serão úteis durante o desenvolvimento e manutenção do banco de dados, como no processo de

migrations.

– "Pomelo.EntityFrameworkCore.MySql": é uma extensão específica para suportar o MySQL no *Entity Framework Core*, o mesmo banco de dados utilizado nas aplicações em `Node.js` e `Django` REST. Ele adiciona o provedor de banco de dados do `MySql` ao EF Core, permitindo a conexão a bancos de dados `MySQL`. Além de fornecer suporte completo para consultas, atualizações e migrações no `MySQL` usando o EF Core.

Como foi feito nos projetos em `Node.js` e `Django`, será criado o modelo Usuário para a estruturação do banco de dados. Este processo é feito no arquivo "UsuarioModel.cs" dentro da pasta *Models*, criada na raiz do projeto. O Listing 3.5.1 apresenta a estrutura do modelo de dados do usuário com as propriedades 'Id', 'Nome', 'Email' e 'Idade'. O atributo [*Key*] na propriedade 'Id' indica que é a chave primária da entidade. Os atributos [*MaxLength*] em 'Nome' e 'Idade' definem os limites máximos de caracteres. Para a criação do banco de dados, foi utilizado o método *Code First*, no qual as classes de entidade e o contexto do banco de dados foram definidos diretamente no código.

```
namespace CrudUsuarios.Models
{
    public class UsuarioModel
    {
        [Key]
        public long Id { get; set; }

        [MaxLength(255)]
        public string Nome { get; set; }

        [MaxLength(45)]
        public string Email { get; set; }

        public int Idade { get; set; }
    }
}
```

Listing 3.5.1 – Modelo Usuário

Outra etapa necessária é a criação de um contexto de dados (*data context*). Ela é responsável por mapear as classes de entidade (criadas no passo anterior) para tabelas do banco de dados e gerenciar as operações de acesso aos dados. Este processo é feito no arquivo "UsuarioDbCotext.cs", localizado dentro da pasta "Data", criada na raiz do projeto. O Listing 3.5.2 mostra como é a estrutura, na qual possui a classe UsuarioDbContext, que herda de DbContext,

fornecida pelo *Entity Framework Core*. Essa classe representará o contexto de dados e fornecerá acesso às entidades do modelo. O 'DbSet' permite o acesso e manipulação dos dados da tabela "Usuarios" por meio do contexto de banco de dados. Sendo assim, para realizar operações na tabela "Usuarios" no banco de dados, será utilizada a entidade "UsuarioModel" da aplicação.

```
namespace CrudUsuarios.Data
{
    public class UsuarioDbContext : DbContext
    {
        public UsuarioDbContext(DbContextOptions<UsuarioDbContext>
    options) : base(options)
        {

        }

        public DbSet<UsuarioModel> Usuarios { get; set; }
    }
}
```

Listing 3.5.2 – Contexto de dados Usuário

Para estabelecer a conexão entre a aplicação e o banco de dados é necessário definir a *string* de conexão no arquivo "`appsettings.json`", localizado na raiz da aplicação, seguindo o modelo do Listing 3.5.3. Essa configuração contém informações essenciais para identificar o servidor, a porta, credenciais de autenticação e outros detalhes relacionados ao banco de dados. Os dados do banco de dados para a conexão seguem os passos apresentados na Seção 3.1.1. A configuração da *string* de conexão pode variar dependendo do ambiente de execução da aplicação, como ambiente de desenvolvimento e produção. Existem diferentes maneiras de configurá-la, utilizando extensões de arquivos específicas para cada ambiente.

```
"ConnectionStrings": {
  "DefaultConnection": "Server=localhost; Port=3306; initial
    catalog=nome_tabela; uid=root ;pwd=senha;"
}
```

Listing 3.5.3 – *String* de conexão do banco de dados

Após a configuração da *string* de conexão, é necessário configurar o contexto de dados no arquivo "`Program.cs`", situado na raiz do projeto, conforme demonstrado no Listing 3.5.4. Neste arquivo, a *string* de conexão do arquivo de configuração será lida e, em seguida,

o contexto de dados será configurado para utilizar essa *string* de conexão com o banco de dados MySQL.

```
using CrudUsuarios.Data;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

var connectionStringMysql =
 ↪  builder.Configuration.GetConnectionString("ConnectionMysql");
builder.Services.AddDbContext<UsuarioDbContext>(options =>
 ↪  options.UseMySql(
        connectionStringMysql,
        ServerVersion.AutoDetect(connectionStringMysql)
        )
);
```

Listing 3.5.4 – Configuração do contexto de dados

Com todas as configurações da conexão feitas, é possível executar o comando "`add-migration init`" no console gerenciador de pacotes. Esse comando criará uma migração inicial com base no modelo de dados atual, após a execução será gerado um arquivo com nome 'init' contendo os contextos de migração. O *Entity Framework* criará uma pasta 'Migrations' na raiz da aplicação com dois arquivos, um de *Snapshot* do modelo de dados e outro com uma classe de migração contendo o código necessário para criar o esquema do banco de dados inicial. Após a criação da migração, o comando "`update-dataset`" aplicará a mesma ao banco de dados `MySQL`, criando as tabelas, colunas e restrições de acordo com o modelo de dados previamente definido.

O processo de criação dos controladores RESTful foi feito utilizando os próprios recursos fornecidos pela plataforma do Visual Studio e pelo EF. Nele é possível fazer a geração de código por meio do *scaffold*, permitindo realizar as operações CRUD (*Create*, *Read*, *Update*, *Delete*) nas entidades do modelo de dados. As Figuras 26 e 27 apresentam os passos para a criação e configuração do mesmo. Com isso, serão escolhidos os tipos de geração de código e quais modelos e classes do contexto de dados serão baseados, respectivamente. Neste caso foi escolhido a geração de controladores com ações utilizando o EF, além de se basear no modelo "UsuarioModel" e contexto "UsuarioDbContext", que foram anteriormente criados. O arquivo a ser gerado para os controladores terá o nome "UsuarioController".

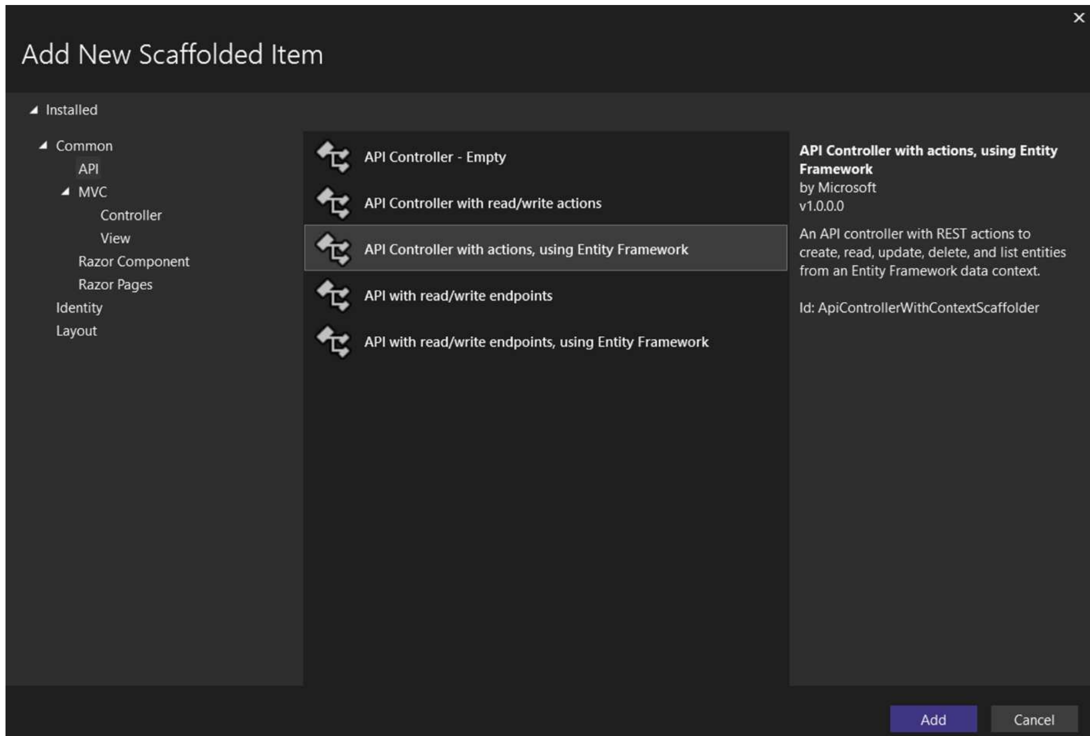

Figura 26 – Geração dos controladores com *Scaffold*

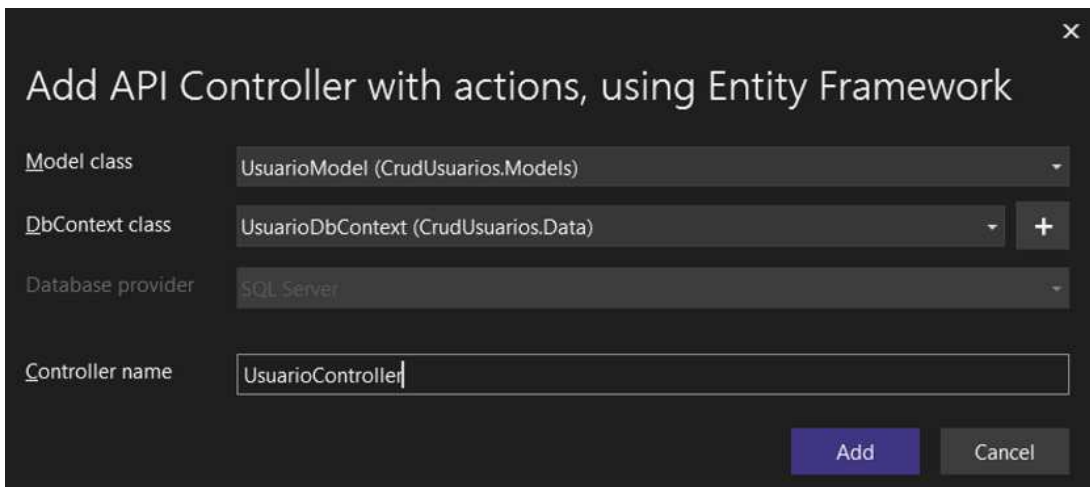

Figura 27 – Configuração dos controladores

Após a configuração e criação de todos os arquivos, a aplicação pode ser executada pelo próprio Visual Studio, utilizando a interface gráfica, ou por meio do comando "`dotnet run`", que pode ser executado em um terminal dentro do diretório raiz do projeto. O servidor ficará então disponível para acesso, com a geração automática de uma documentação padrão em *Swagger*, que permite a visualização e interação de forma gráfica com a API. Nele será exibido informações sobre os *endpoints* disponíveis, parâmetros aceitos, modelos de dados, entre outros detalhes que serão apresentados em seguida. O controlador do método GET gerado automaticamente é apresentado no Listing 3.5.5, contendo dois *endpoints*, servindo tanto para leitura de todos usuários, quando para um usuário com 'Id' específico. O controlador retornará uma lista de objetos 'UsuarioModel' ou um objeto específico, dependendo da rota. Um exemplo de uso da rota que retorna todos os usuário pode ser visto na Figura 28.

```
1  // GET: api/Usuario
2  [HttpGet]
3  public async Task<ActionResult<IEnumerable<UsuarioModel>>> GetUsuarios()
4  {
5    if (_context.Usuarios == null)
6    {
7        return NotFound();
8    }
9      return await _context.Usuarios.ToListAsync();
10 }
11
12 // GET: api/Usuario/5
13 [HttpGet("{id}")]
14 public async Task<ActionResult<UsuarioModel>> GetUsuarioModel(long id)
15 {
16   if (_context.Usuarios == null)
17   {
18       return NotFound();
19   }
20     var usuarioModel = await _context.Usuarios.FindAsync(id);
21
22     if (usuarioModel == null)
23     {
24         return NotFound();
25     }
26
27     return usuarioModel;
28 }
```

Listing 3.5.5 – Controlador do método GET

Figura 28 – Exemplo método GET (API ASP.NET Core)

O método POST gerado é apresentado no Listing 3.5.6. Possui o decorador ’[HttpPost]’ indicando que é acionado na rota ’api/Usuario’ por meio do método POST recebendo um objeto UsuarioModel no corpo da requisição, que serão os dados do usuário a serem criados. Caso a tabela exista no contexto de dados, o usuário será adicionado utilizando o método ’Add()’, salvando as alterações assincronamente por meio do método ’SavechangesAsync()’. O ’CreatedAtAction’ será o retorno que indicará o sucesso na requisição. Um exemplo de uso do método POST pode ser visto na Figura 29.

```
// POST: api/Usuario
[HttpPost]
public async Task<ActionResult<UsuarioModel>> PostUsuarioModel(UsuarioModel usuarioModel)
{
  if (_context.Usuarios == null)
  {
      return Problem("Entity set 'UsuarioDbContext.Usuarios'  is null.");
  }
    _context.Usuarios.Add(usuarioModel);
    await _context.SaveChangesAsync();

    return CreatedAtAction("GetUsuarioModel", new { id = usuarioModel.Id }, usuarioModel);
}
```

Listing 3.5.6 – Controlador do método POST

Figura 29 – Exemplo método POST (API ASP.NET Core)

O Listing 3.5.7 apresenta a geração do método PUT. O decorador '[HttpPut("id")]' faz a indicação de que o controlador lida com requisições do tipo PUT na rota 'api/Usuario/id', no qual 'id' é um parâmetro de rota que identifica o usuário que será alterado. O controlador inicialmente fará uma verificação para saber se o 'Id' do usuário repassado na rota é o mesmo do corpo da requisição, caso não seja, será retornado um 'BadRequest()'. Uma verificação também é feita posteriormente para saber se o usuário contém no banco de dados. Caso exista, a operação será concluída e terá um retorno do código 204, indicando sucesso. Um exemplo da utilização da rota para alterar os dados de um usuário pode ser visto na Figura 30.

```
// PUT: api/Usuario/{id}
[HttpPut("{id}")]
public async Task<IActionResult> PutUsuarioModel(long id, UsuarioModel
    usuarioModel)
{
    if (id != usuarioModel.Id)
    {
        return BadRequest();
    }

    _context.Entry(usuarioModel).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!UsuarioModelExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}
```

Listing 3.5.7 – Controlador do método PUT

PUT /api/Usuario/{id}

Parameters

Try it out

Name Description

id * required

integer($int64)

(path)

id

Request body

application/json

Example Value | Schema

```
{
  "id": 0,
  "nome": "string",
  "email": "string",
  "idade": 0
}
```

Responses

Code	Description	Links
200	Success	No links

Figura 30 – Exemplo método PUT (API ASP.NET Core)

Finalizando os controladores para um CRUD RESTful, é apresentado o controlador para o método POST no Listing 3.5.8. Semelhante ao decorador do método PUT, o ’[HttpDelete("id")]’ indicará que o controlador será utilizado na rota ’api/Usuario/id’ referentes ao método DELETE. O mesmo fará a verificação para saber se o ’Id’ repassado no parâmetro de rota existe no banco de dados, caso exista, o usuário será deletado e retornará ’NoContent()’ com o método 204, indicando sucesso na operação. Um exemplo de uso pode ser visto na Figura 31.

```csharp
// DELETE: api/Usuario/{id}
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteUsuarioModel(long id)
{
    if (_context.Usuarios == null)
    {
        return NotFound();
    }

    var usuarioModel = await _context.Usuarios.FindAsync(id);

    if (usuarioModel == null)
    {
        return NotFound();
    }

    _context.Usuarios.Remove(usuarioModel);
    await _context.SaveChangesAsync();

    return NoContent();
}
```

Listing 3.5.8 – Controlador do método DELETE

Figura 31 – Exemplo método DELETE (API ASP.NET Core)

4 METODOLOGIA

Neste capítulo, será abordada a metodologia utilizada para realizar a comparação entre os *frameworks back-end* `Node.js`, `Django REST Framework` e `ASP.NET Core`. Serão definidas as etapas necessárias para alcançar os objetivos deste estudo. A metodologia proposta inclui as seguintes etapas: levantamento de características teóricas, levantamento dos requisitos do *software* alvo, implementação dos CRUDs (*Create, Read, Update, Delete*) em cada tecnologia, escolha das métricas para comparação, execução das métricas, análise comparativa das métricas, pesquisa envolvendo desenvolvedores e usuários do software alvo e, por fim, apresentação dos resultados e sugestões para trabalhos futuros. A Figura 32 apresenta um diagrama de atividades com todas as etapas que serão descritas nas próximas seções.

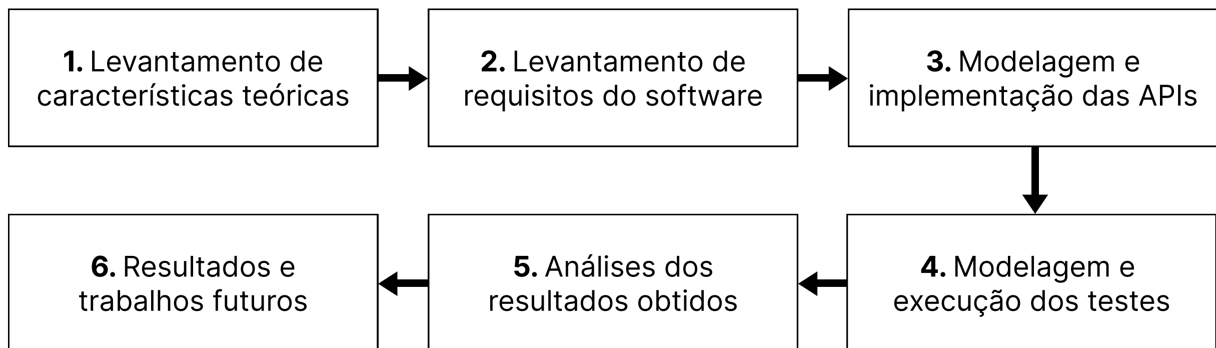

Figura 32 – Etapas metodológicas

A Seção 4.1 apresenta os primeiros passos nas etapas metodológicas, iniciando com a revisão bibliográfica, onde foram realizados estudos e definições das métricas comparativas teóricas. A definição dos requisitos do *software* alvo é apresentada na Seção 4.2. A implementação das APIs em cada *framework* é detalhada na Seção 4.3. A Seção 4.4 discute as métricas escolhidas para a comparação prática a partir dos testes. Por fim, a modelagem e execução dos testes para a captura dos dados utilizados na comparação prática são abordadas na Seção 4.5.

4.1 Levantamento de características teóricas

A primeira etapa do estudo se dará com uma revisão bibliográfica. A revisão bibliográfica tem como objetivo principal explorar e analisar as fontes existentes na literatura acadêmica e técnica relacionadas às tecnologias *back-end* `Node.js`, `Django REST Framework` e ASP.NET Core, no contexto do desenvolvimento de um software.

Durante essa etapa, foram pesquisados artigos científicos, livros, documentações oficiais e outros materiais relevantes que abordassem essas tecnologias e sua aplicação no desenvolvimento de sistemas e aplicações. A revisão bibliográfica permite uma compreensão aprofundada das características, recursos, benefícios e desafios associados a cada uma das tecnologias analisadas. Como resultado deste estudo, será fornecida uma análise comparativa teórica entre as tecnologias em estudo. A pesquisa é baseada em estudos bibliográficos e visa contribuir com a documentação disponível para futuros trabalhos e projetos que utilizem essas tecnologias.

4.1.1 Análise comparativa teórica

A partir dos dados coletados nas etapas revisão bibliográfica, nesta etapa foi feito uma síntese com base em critérios qualitativos das tecnologias em estudo. Contudo, a lista a seguir apresenta as as métricas que foram comparadas:

1. Multiplataforma: Foram verificados os Sistemas Operacionais suportados por cada tecnologia, com foco especial em Windows, Linux e MacOS. Este aspecto é importante para garantir a portabilidade e a flexibilidade do sistema, possibilitando o desenvolvimento em diversos ambientes distintos.
2. Suporte a banco de dados: É importante verificar o suporte aos bancos de dados mais utilizados, conforme indicado pelo Survey do Stack Overflow[1], tais como `MySQL`, `PostgreSQL`, `SQLite`, `MongoDB` e `Microsoft SQL Server`. A flexibilidade na escolha do banco de dados permite a seleção da opção mais adequada às necessidades específicas de sua aplicação. Essa variedade de suporte oferece a flexibilidade necessária para escolhera que melhor se adequa aos requisitos de armazenamento e gerenciamento de dados de um projeto.
3. Suporte a ORM Nativo: Foi verificado o suporte a ORM nativo como método de desenvolvimento para otimização no processo de criação de aplicações. A utilização de ORM nativo simplifica o desenvolvimento e a manutenção do código, resultando em um aumento na produtividade e na redução de erros durante o ciclo de desenvolvimento. Essa abordagem oferece uma forma mais eficaz e eficiente de interagir com bancos de dados e modelar dados em objetos, facilitando o trabalho dos desenvolvedores.
4. Documentação nativa da API: Foi considerado a disponibilidade da documentação nativa da API em tempo de desenvolvimento. A documentação automática torna mais fácil

[1] https://survey.stackoverflow.co/2022

para os desenvolvedores entenderem como usar a API. Isso melhora a comunicação entre equipes de desenvolvimento, permitindo que os desenvolvedores tenham conhecimento com relação aos *endpoints*, parâmetros, respostas e formatos de dados.

4.2 Levantamento de requisitos do *software* alvo

Para a aplicação prática e integrativa do estudo foi desenvolvido um *software* alvo para a realização das métricas a serem comparadas assim como medidas qualitativas relevantes. A aplicação consiste em uma API para um *software* de controle de estoque de componentes eletrônicos para o laboratório da UFC, onde será controlado por um professor tendo as seguintes funções e requisitos:

- **Inserção**: Inserir um novo componente desejado informando nome, descrição, referência e quantidade.
- **Atualização**: Retirar ou adicionar uma certa quantidade de um componente já existente.
- **Deleção**: Remover um componente quando o mesmo já não possui mais em estoque.
- **Pesquisa**: Buscar informações de um componente específico ou geral.

A leitura das informações do estoque de componentes pode ser feita tanto pelo professor quanto por aluno, onde ambos poderão fazer pesquisa de componentes existentes ou buscar por informações de um componente específico. A Figura 33 apresenta um diagrama da aplicação do CRUD na aplicação citada.

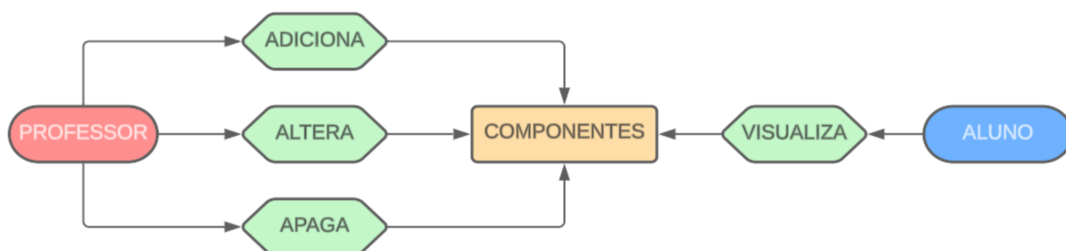

Figura 33 – Diagrama do *software* alvo

4.3 Implementação do *software* alvo

O desenvolvimento do *software* alvo teve como base os projetos implementados nas Seções 3.3.6, 3.4.5 e 3.5.3. Foram desenvolvidas três APIs, uma para cada *framework*. Cada APIs foi implementada de forma independente, utilizando a mesma tecnologia de banco de dados

(`MySQL`), mas com estruturas diferentes devido às peculiaridades de cada *framework*. Os CRUDs implementados foram feitos seguindo os padrões apresentados na Seção 3.1. Cada método HTTP foi responsável por fazer uma operação no sistema, sendo elas:

- **GET**: Buscar informações dos componentes disponíveis na base de dados.
- **POST**: Inserir um componente na base de dados.
- **PUT**: Alterar informações de um componente já existente.
- **DELETE**: Remover um componente da base de dados.

4.4 Métricas comparativas práticas

Após o desenvolvimento das APIs em cada uma das tecnologia em estudo, `Node.js`, `Django REST Framework` e `.NET`, uma etapa fundamental nesta pesquisa envolveu a condução de testes práticos. O objetivo principal foi avaliar o desempenho e a eficiência de cada tecnologia em diferentes contextos. Para isso, foram implementados testes de carga nos servidores desenvolvidos. Esses testes simulam situações do mundo real, submetendo as APIs a diferentes níveis tráfego e demanda, permitindo avaliar como cada tecnologia se comporta em determinados cenário.

Nos testes de carga, destacam-se duas categorias de métricas essenciais que podem ser monitoradas e coletadas: métricas de rendimento e métricas de desempenho. Essas métricas são fundamentais para entender como as tecnologias lidam quando são submetidas a diferentes cenários de cargas e fornecer *insights* sobre sua capacidade de resposta, eficiência e escalabilidade em ambientes variados (JIANG; HASSAN, 2015).

Para este trabalho, com base no estudo anterior, foram escolhidas as seguintes métricas:

- Taxa de erros de requisição: Como uma métrica de rendimento, a taxa de erros de requisição é essencial para analisar a proporção de solicitações que resultam em erros em relação ao número total de solicitações feitas durante os testes. Estes erros podem incluir falhas de requisição ou erros internos do servidor.
- Tempo de requisição: Dentro do contexto de performance, esta métrica é importante para medir o tempo que uma API leva para responder ou completar uma solicitação de um cliente. Um menor tempo de resposta geralmente indica um melhor desempenho da API.
- Consumo de Memória de Acesso Aleatório (RAM): Dentro do contexto de gerenciamento de recursos, a avaliação do consumo de memória permite entender o quanto de memória a

tecnologia consome durante a execução das solicitações. Isso é essencial para dimensionar e otimizar o uso de recursos do servidor.

– Consumo de CPU: Similar ao consumo de memória, a análise do consumo de CPU também é importante para o gerenciamento de recursos de uma aplicação. A alta utilização da CPU pode levar a custos operacionais elevados e, em última instância, impactar o desempenho e escalabilidade.

4.5 Execução dos testes para leitura das métricas

Após o desenvolvimento das APIs nas tecnologias em estudo, foram realizados testes de carga para a obtenção das métricas detalhadas na Seção 4.4. O processo de testes envolve alguns passos importantes para a melhor otimização dos processos e validação dos resultados. A Figura 34 apresenta os processos dos testes que entregaram como resultado observações práticas e *insights* sobre o comportamento e desempenho das tecnologias submetidas a diferentes cenários (JIANG; HASSAN, 2015; HOODA; CHHILLAR, 2015).

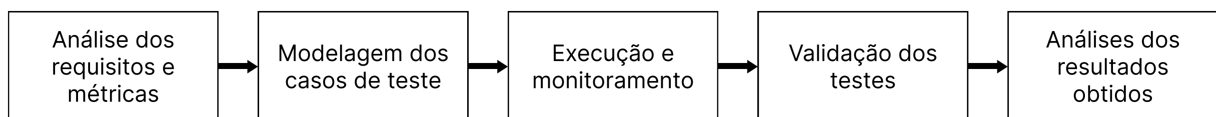

Figura 34 – Processos para a modelagem dos testes

Dentro do cenário de testes, os métodos se resumem à dois grupos principais: Testes de pico (também conhecidos como *flash test*) e testes de imersão (ARTILLERY, 2023). Os testes de pico são cenários para avaliar como um sistema se comporta quando é submetido a picos repentinos de carga, ou seja, um aumento significativo no tráfego ou na demanda do sistema em um curto período de tempo. Os testes de imersão, também conhecidos como testes de estresse contínuo, são projetados para verificar o comportamento do sistema sob carga constante contínua por um período de tempo prolongado. Existem também outros tipos de teste, como teste de estresse, teste de carga zero e outros, sendo variações dos grupos principais citados ou para casos específicos (MUSTAFA *et al.*, 2009). Com isso, para garantir que servidores possam lidar com a demanda de diferentes cenários e oferecer um desempenho satisfatório aos usuários, são realizados testes de carga nas aplicações simulando os casos reais.

Para comparação prática foram modelados três tipos de cenários para a coleta das

métricas de performance e rendimento de cada API, são eles:

- Teste de pico: a aplicação será submetida à uma grande quantidade de requisições durante um curto intervalo de tempo e será observado o comportamento da mesma com relação ao tempo de resposta e consumo de recurso.
- Teste de carga crescente: a aplicação será submetida à uma carga crescente de requisições e monitorada para analisar o comportamento sob esta situação.
- Teste de resistência: como uma variação de *soak test*[2], neste teste será feito um cenário com requisições a uma carga constante durante um longo período de tempo, buscando identificar o comportamento geral do sistema.

Para que os servidores fossem monitorados e comparados em sua melhor capacidade, foram feitos alguns procedimentos específicos. A quantidade de conexões do banco de dados foi alterada para que não interferisse nos resultados. O banco de dados utilizados nas APIs foi o `MySQL`, que tem capacidade máxima padrão de 151 conexões. Através da documentação foi possível ter um suporte para fazer a alteração deste valor, tendo como valor máximo a quantidade de 10.000 conexões[3]. A cada cenário de teste os servidores foram reiniciados para não ter influências de resultados passados ou requisições pendentes. A máquina utilizada contém as seguintes especificações:

- Processador: Ryzen 5 5600x;
- Memória RAM: 16GB, 3200MHz;
- Placa de vídeo: GTX 1660;

A modelagem e execução dos testes foi na ferramenta de testes `JMeter`[4] (ABBAS *et al.*, 2017). O `JMeter` é uma ferramenta de código aberto em Java utilizada para testes estáticos e dinâmicos em aplicações Web, servidores e outros tipos de sistema. Ele oferece recursos para modelagem, execução, geração de relatórios e análise de testes. Sua instalação é de fácil entendimento, podendo ser feita pelo site oficial[5].

Os testes feitos no `JMeter` consistem em requisições que simulam usuários, chamados de Usuário virtual (VU)s, usuários virtuais. Estes VUs simulam o comportamento de um usuário real utilizando a aplicação a ser testada. Conforme apresenta a Figura 35, em todos os cenários o fluxo de requisições escolhido foi:

[2] Teste utilizado para identificar problemas de um sistema quando exposto a uma carga constante por um grande período de tempo
[3] https://dev.mysql.com/doc/refman/8.0/en/too-many-connections.html
[4] https://jmeter.apache.org/
[5] https://jmeter.apache.org/download_jmeter.cgi

1. Criar um componente no banco de dados;
2. Atualizar os dados do componente criado no passo anterior;
3. Fazer a leitura do componente criado;
4. Apagar o componente criado no passo 1;

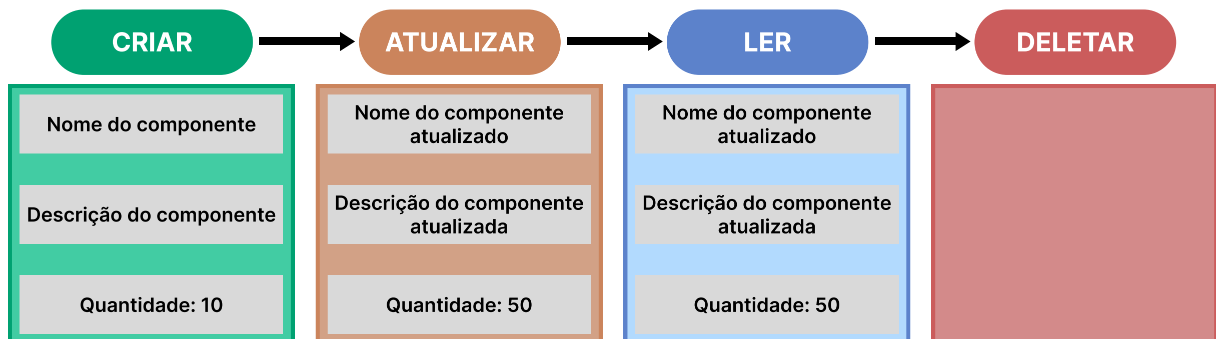

Figura 35 – Fluxo de requisições dos testes

A modelagem dos testes foram feitas utilizando o *plugin Stepping Thread Group* e *Ultimate Thread Group*[6]. Com ele é possível definir vários parâmetros de configuração para testes de carga. Os principais parâmetros utilizados foram a quantidade de grupos de *threads*, atraso de iniciação de cada grupo e o tempo de duração dos mesmos. A seguir são apresentados como foram feitas as modelagem dos testes com relação à quantidade de cargas utilizadas durante a execução de cada cenário.

– **Teste de pico**: A Tabela 2 apresenta os dados de como foi modelado o teste de pico. Nele, o servidor foi inicializado com uma carga de 25 a 50 *threads* e repentinamente elevada para um pico de 500 *threads* durante 30 segundos. Após isto a quantidade de *threads* decaiu para os mesmos valores do início. A Figura 36 mostra graficamente como se comporta a quantidade de *threads* ao longo do tempo.

[6] http://jmeterplugins.com/wiki/Start/index.html

Quantidade de *threads*	**Atraso inicial (segundos)**	**Duração da carga (segundos)**
25	0	20
50	20	20
500	40	30
50	70	20
25	90	20

Tabela 2 – Modelagem do teste de pico

Figura 36 – Modelagem do teste de pico

– **Teste de carga crescente**: A Tabela 3 apresenta como se deu a modelagem do teste de carga crescente. Neste cenário o servidor iniciou com 50 usuários tendo um aumento de 20 usuários gradualmente a cada 20 segundos, chegando ao limite de 200 *threads*. A Figura 37 mostra graficamente como ocorre este processo de aumento de *threads ao longo do tempo*.

Quantidade de *threads*	**Atraso inicial (segundos)**	**Duração da carga (segundos)**
50	0	20
75	20	20
100	40	20
125	60	20
150	80	20
175	100	20
200	120	20

Tabela 3 – Modelagem do teste de carga crescente

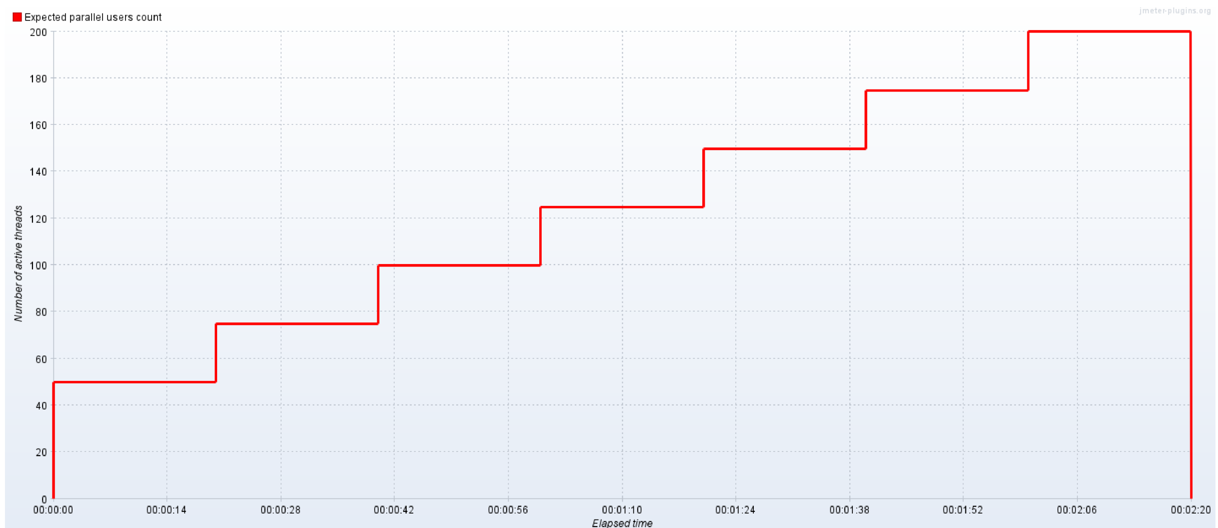

Figura 37 – Modelagem do teste de carga crescente

– **Teste de resistência**: Diferente dos outros cenários, no teste de resistência não tem variação de carga ao longo do tempo. A Tabela 4 apresenta como se deu a modelagem deste teste. Os servidores foram submetidos à 50 *threads* durante um período de 3 horas. A Figura 38 mostra graficamente como se comporta o grupo de *threads* ao longo do tempo, não ocorrendo variações.

Quantidade de *threads*	**Atraso inicial (segundos)**	**Duração da carga (segundos)**
50	0	10800

Tabela 4 – Modelagem do teste de resistência

Figura 38 – Modelagem do teste de resistência

Na execução de cada teste o `Jmeter` faz a leitura e monitoramento dos dados e recursos. Ao final da execução é gerado um arquivo com os dados do teste executado. Com estes dados foram feitos relatórios onde foi possível fazer a verificação do tempo de execução, sucesso das requisições, quantidade de dados trafegados e outros. Os testes foram feitos no sistema operacional Linux, auxiliando no monitoramento do consumo de recursos das APIs. Primeiramente foi utilizado o comando "lsof"[7]. Com ele é possível identificar o processo de execução do servidor através da porta aberta na execução da API. O Listing 4.5.1 apresenta o comando utilizado para identificação do processo.

[7] https://www.geeksforgeeks.org/lsof-command-in-linux-with-examples/

```
1 lsof -i :<porta>
```

Listing 4.5.1 – Comando para identificação de processo no Linux

Após a identificação do *Process Identification Number* (PID) do processo é feito o monitoramento do mesmo. Para isto foi utilizado o comando "top"[8]. Com este comando foi possível gerar um arquivo de logs apresentando os dados de consumo de recursos em relação ao tempo. O comando para este processo é apresentado no Listing 4.5.2.

```
1 top -b -d 1 -p <PID> >> <caminho/do/arquivo/de/logs.txt>
```

Listing 4.5.2 – Comando para gerar o arquivo de gerenciamento de recursos no Linux

Para otimização dos processos de testes foram utilizados (opcional) recursos como variáveis ambientes do `Jmeter` para facilitar a transição de cenários de testes e tecnologias a serem testadas. Além disso também foi feito um *script* para a execução do arquivo de teste do `Jmeter` para execução via linha de comando, assim como para os comandos nativos do Linux para identificação do PID dos processos e geração de arquivos de logs.

[8] https://www.geeksforgeeks.org/top-command-in-linux-with-examples/

5 RESULTADOS E DISCUSSÕES

Neste capítulo, serão apresentados e discutidos os resultados da comparação entre as tecnologias back-end. Este trabalho envolve dois momentos principais: a análise das características e o estudo empírico dessas tecnologias. O levantamento e análise das características envolveram a abordagem exploratória para realizar uma análise documental dessas tecnologias. Para isso, foi identificado características essenciais para avaliar tecnologias de desenvolvimento, destacando suas funcionalidades, recursos, contextos de aplicação e outros fatores relevantes. No estudo empírico, foi desenvolvido um *software* alvo de maneira uniforme em todas as tecnologias selecionadas, garantindo que os aplicativos fossem equivalentes em termos de funcionalidade e recursos. Isso foi feito com o propósito de permitir uma comparação precisa das métricas de desempenho, como tempo de resposta e consumo de recursos, entre `Node.js`, `Django REST Framework` (DRF) e `.NET Core`. Essa abordagem garantiu que as diferenças observadas nas métricas refletissem o desempenho característico de cada tecnologia, independentemente das funcionalidades específicas de cada uma.

5.1 Análise teórica

Os resultados da análise das características das tecnologias `Node.js`, `Django REST Framework` (DRF) e `.NET` são apresentados nos tópicos a seguir. Tais informações foram obtidas por meio de estudos bibliográficos relacionados e documentações oficiais de cada uma das tecnologias. A discussão das comparações abordam os atributos citados na Seção 4.1.1. Os estudos realizados respondem a ***QP*$_1$*:*** *Qual o framework com mais funcionalidades nativas seguindo as características escolhidas para estudo?*.

5.1.1 Sistemas Operacionais suportados

A Tabela 5 apresenta o suporte para o desenvolvimento utilizando as tecnologias em estudos em diferentes sistemas operacionais. Todas as tecnologias são compatíveis com os sistemas operacionais citados. Apesar da semelhança, o `.NET Core` tem um melhor desempenho, sendo uma nova estrutura reescrita do `.NET Framework`. Para casos de migração de estrutura é fornecido na própria documentação um suporte para tal atualização[1].

[1] https://learn.microsoft.com/pt-br/aspnet/core/migration/proper-to-2x/?view=aspnetcore-7.0

CARACTERÍSTICAS		Node.js	DRF	ASP.NET Core
Sistemas operacionais	**Windows**	Sim	Sim	Sim
	Linux	Sim	Sim	Sim*
	MacOS	Sim	Sim	Sim*

Tabela 5 – Suporte das tecnologias aos principais SOs.
** É importante ressaltar que apenas a estrutura do ASP.NET Core é multiplataforma, se diferenciando da estrutura .NET Framework que tem suporte apenas para o sistema operacional Windows.*

5.1.2 Compatibilidade com bancos de dados

A Tabela 6 apresenta a compatibilidade das tecnologias com diferentes bancos de dados, relacionais e não relacionais. As três tecnologias em estudos tem suporte aos bancos de dados citados, com exceção do `MongoDB` no DRF. Os banco de dados não relacionais não são suportados oficialmente pelo `Django`, porém, há alguns projetos para implementação[2]. Para o suporte para banco de dados não relacionais, em específico para o `MongoDB`, é citado na wiki do `Django` a utilização da biblioteca `djongo`[3], que faz a tradução de uma *query* SQL para uma *query* do `MongoDB`[4]. É importante citar que a plataforma `.NET` é altamente compatível com o `SQL Server`, banco de dados desenvolvido pela Microsoft. Esses dois componentes funcionam bem juntos, facilitando o desenvolvimento de aplicativos corporativos.

CARACTERÍSTICAS		Node.js	DRF	ASP.NET Core
Suporte à Banco de dados	**MySQL**	Sim	Sim	Sim
	PostgreSQL	Sim	Sim	Sim
	SQLite	Sim	Sim	Sim
	MongoDB	Sim	Não*	Sim
	SQL Server	Sim	Sim	Sim

Tabela 6 – Suporte das tecnologias aos principais bancos de dados.
** Bancos de dados não relacionais não são suportados nativamente pelo Django.*

[2] https://docs.djangoproject.com/pt-br/4.2/faq/models/#does-django-support-nosql-databases
[3] https://github.com/doableware/djongo
[4] https://code.djangoproject.com/wiki/NoSqlSupport

5.1.3 Suporte ao ORM nativo

A Tabela 7 faz a relação das tecnologias que oferecem suporte ao ORM nativo. Tanto o `Node.js` quanto o `.NET Core` não possuem esta funcionalidade nativamente. No `Node.js`, é possível utilizar bibliotecas de terceiros em projetos `Node.js` para a implementação de ORM, como `Prisma`[5], `Sequelize`[6] e `TypeORM`[7]. O `.NET Core`, por sua vez, possui alguns ORMs, sendo o `Entity Framework Core`[8] o mais utilizado, tendo o Dapper[9] como uma alternativa de software livre popular. O DRF é o único que possui esta funcionalidade nativamente, não precisando fazer consultas SQL manualmente, podendo ser abstraído esta interação com o banco de dados.

CARACTERÍSTICAS	Node.js	DRF	ASP.NET Core
ORM nativo	Não	Sim	Não

Tabela 7 – Suporte das tecnologias ao ORM nativo

5.1.4 Documentação nativa da API

A Tabela 8 apresenta o suporte aos sistemas operacionais mais utilizados das tecnologias em estudo. Dentre as três, o `.Net` é o único *framework* que possui suporte a geração automática de documentação nativamente. Ele utiliza o `Swagger` como ferramenta para tal, gerando uma interface `SwaggerUI` apresentando os serviços e os métodos de ações públicas[10]. Esta documentação permite fazer o teste da API através da interface e verificar informações específicas de cada método.

No DRF, em tempo de construção do projeto, é gerado um painel administrativo no formato de interface Web que permite a configuração e o gerenciamento de conteúdos. A geração de documentação da API não é feita de forma automática, no entanto existe suporte integrado para a geração de esquemas OpenAPI, que é utilizado para geração de documentação[11].

[5] https://www.prisma.io/
[6] https://sequelize.org/
[7] https://typeorm.io/
[8] https://learn.microsoft.com/pt-br/ef/
[9] https://github.com/StackExchange/Dapper
[10] https://learn.microsoft.com/pt-br/aspnet/core/tutorials/web-api-help-pages-using-swagger?view=aspnetcore-7.0
[11] https://www.django-rest-framework.org/topics/documenting-your-api/

Como biblioteca de terceiros é recomendado a utilização do `drf-yasg`[12] e `drf-spectacular`[13]. No `Node.js` também é possível fazer esta configuração utilizando bibliotecas externas, como o `Swagger`. Com isto também pode ser gerado uma interface com `SwaggerUI` assim como no `.Net`.

CARACTERÍSTICAS	Node.js	DRF	ASP.NET Core
Documentação nativa	Não	Não*	Sim

Tabela 8 – Suporte das tecnologias à documentação nativa.
** A geração de documentação pelo DRF não é feita de forma automática, porém existe suporte integrado.*

QP$_1$. De acordo com os estudos bibliográficos realizados, o DRF e o .Net apresentaram mais funcionalidades nativas. No entanto, os três *frameworks* possuem compatibilidade para diversas funcionalidades, principalmente utilizando bibliotecas de terceiros. Portanto, os três são flexíveis para adaptação dependendo do contexto que serão utilizados.

5.2 Análise prática

Os resultados da análise das métricas práticas das tecnologias `Node.js`, `Django Rest Framework` (DRF) e `.Net` são apresentados nos tópicos a seguir. Tais informações foram obtidas por meio de testes realizados nos servidores feitos em cada uma das tecnologias, descritos na Seção 4.5. A discussão das comparações abordam os atributos citados na Seção 4.4. A Seção 5.2.1 apresenta os resultados obtidos no teste de pico. A Seção 5.2.2 mostra os dados resultantes do teste de carga crescente. A Seção 5.2.3 demonstra os resultados do teste de resistência. E por fim, a Seção 5.3 descreve algumas outras observações notadas durante a realização dos testes feitos.

5.2.1 Teste de pico

Neste tópico será apresentado os resultados comparativos no cenário de teste de pico. Este caso simula um servidor com execução padrão e que em um determinado momento acontece um aumento repentino na quantidade de requisições. Isto simula, por exemplo, uma loja que está com algum evento de promoção e em determinado momento o acesso ao site cresce

[12] https://github.com/axnsan12/drf-yasg/
[13] https://github.com/tfranzel/drf-spectacular/

repentinamente.

Respondendo a QP$_2$: *"Qual o framework mais otimizado com relação ao consumo de recursos no cenário de teste de pico?"*. A Figura 39 apresenta os resultados médios do consumo de CPU e memória em porcentagem das tecnologias durante os testes de pico. As tecnologias `Node.js` e DRF tiveram o consumo relativamente aproximados e baixo, comparados ao `.Net`. O `.Net`, por sua vez, teve um consumo mais elevado.

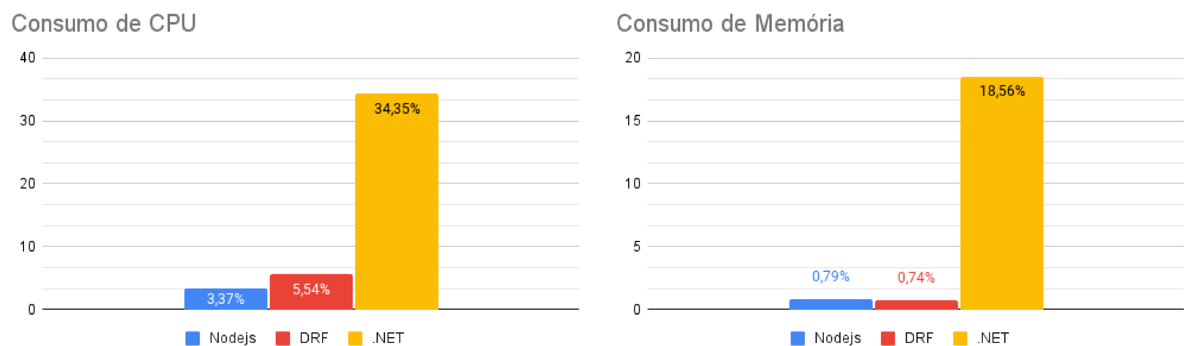

Figura 39 – Consumo médio de recursos - teste de pico

QP$_2$. O Node.js apresentou menor consumo de CPU enquanto o DRF mostrou melhor resultado no consumo de RAM.

Respondendo a QP$_3$. *"Qual o framework mais otimizado com relação ao tempo de resposta no cenário de teste de pico?"*. A Figura 40 apresenta os resultados médios do tempo de resposta agrupados por métodos HTTP das tecnologias durante os testes de pico. Como discrepância foi registrado o método POST do DRF neste cenário. Os demais métodos não tiveram o mesmo comportamento, tendo uma maior proximidade nos tempos. Dentre eles se destacam os tempos do `.Net`, sendo inferior a todos neste caso de teste.

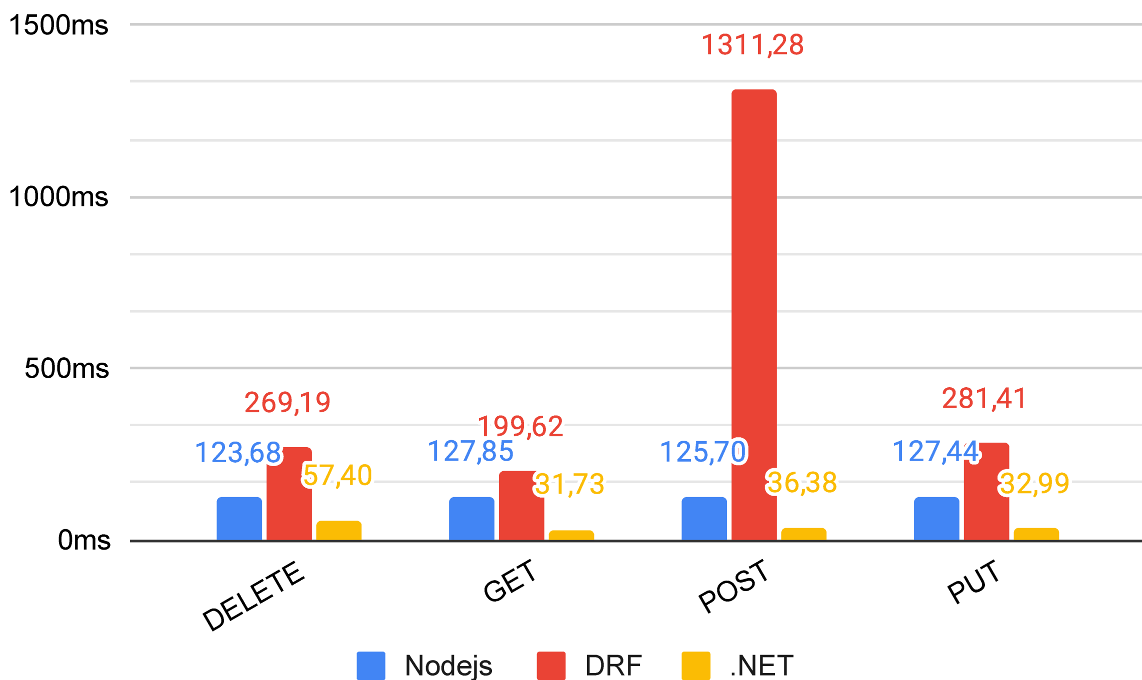

Figura 40 – Tempo de resposta médio dos métodos HTTP - teste de pico

QP$_3$. O .Net apresentou os melhores resultados em tempo de resposta em todos os métodos.

5.2.2 *Teste de carga crescente*

Neste tópico será apresentado os resultados comparativos no cenário de teste de carga crescente. Este caso simula um servidor com execução padrão que tem sua carga acrescida gradativamente ao longo do tempo. Isto simula, por exemplo, um servidor de *streaming* de vídeo que possui um evento programado, como um jogo de futebol, e ao longo do tempo o tráfego dos usuários aumenta gradualmente a medida que chega perto do início do evento.

Respondendo a QP$_4$: *"Qual o framework mais otimizado com relação ao consumo de recursos no cenário de teste de carga crescente?"*. A Figura 41 apresenta os resultados médios do consumo de CPU e Memória em porcentagem das tecnologias durante os testes de carga crescente. As tecnologias `Node.js` e DRF tiveram o consumo relativamente aproximados e baixo, comparados ao `.Net`, sendo o `Node.js` com os menores valores. O `.Net`, por sua vez, teve um consumo mais elevado.

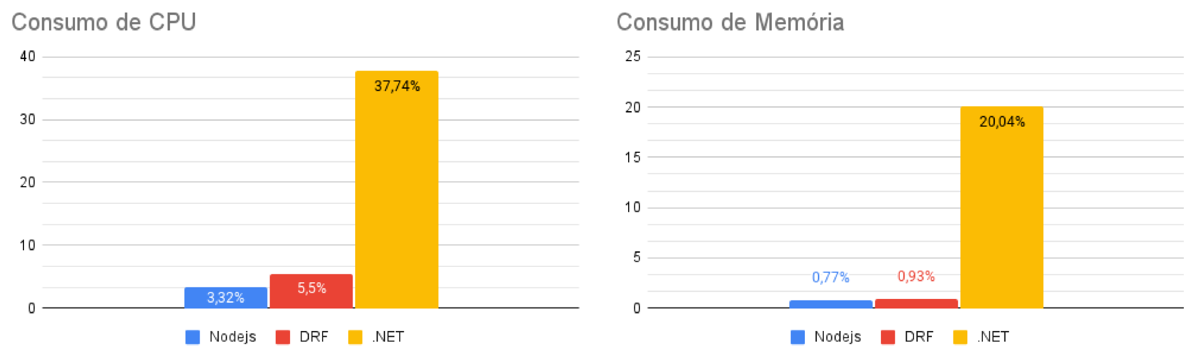

Figura 41 – Consumo médio recursos - teste de carga crescente

QP$_4$. O Node.js apresentou os melhores resultados, tanto para consumo de CPU quando RAM.

Respondendo a QP$_5$: *"Qual o framework mais otimizado com relação ao tempo de resposta no cenário de teste de carga crescente?"*. A Figura 42 apresenta os resultados médios do tempo de resposta agrupados por métodos HTTP das tecnologias durante os testes de carga crescente. Como discrepância foi registrado o método POST do DRF neste cenário, tendo aproximadamente o dobro do tempo comparado com seus outros métodos (DELETE, GET e PUT). Os demais métodos não tiveram o mesmo comportamento, tendo uma maior proximidade nos tempos. Dentre eles se destacam os tempos do `Node.js` e `.Net` com pouca variação de tempo entre os métodos, sendo o `.Net` com menores tempos neste caso de teste.

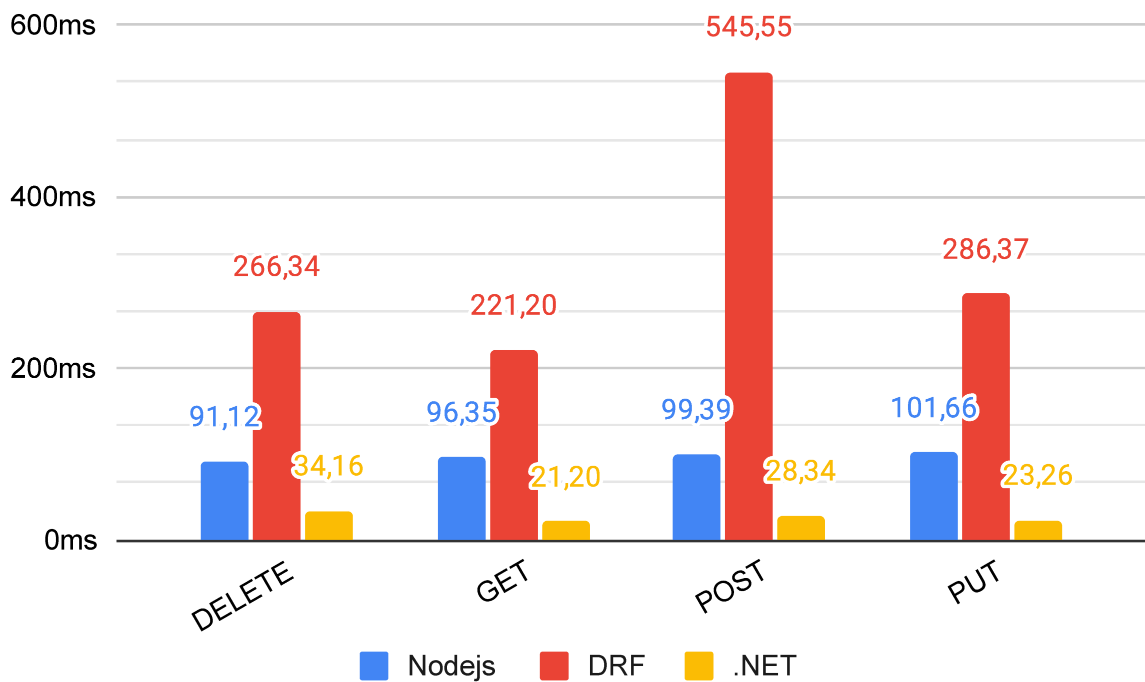

Figura 42 – Tempo de resposta médio dos métodos HTTP - teste de carga crescente

QP$_5$. O .Net apresentou os melhores resultados em tempo de resposta em todos os métodos no cenário de carga crescente.

5.2.3 Teste de resistência

Neste tópico será apresentado os resultados comparativos no cenário de teste de resistência. Este caso simula um servidor com execução padrão que tem sua carga mantida durante um longo período de tempo. Isto simula, por exemplo, um sistema de monitoramento de sensores em uma fábrica, onde tem-se a leitura dos sensores e controle dos dados durante um logo período de tempo se aproximando de uma carga constante em um cenário sem distúrbios.

Respondendo a QP$_6$: *"Qual o framework mais otimizado com relação ao consumo de recursos no cenário de teste de resistência?"*. A Figura 43 apresenta os resultados médios do consumo de CPU e Memória em porcentagem das tecnologias durante os testes de resistência. Neste cenário as tecnologias `Node.js` e DRF tiveram baixos consumos de recursos quando comparados ao `.Net`. O `Node.js` obteve o menor consumo de CPU, enquanto o DRF teve um menor consumo de memória, sendo levemente melhor que o `Node.js`.

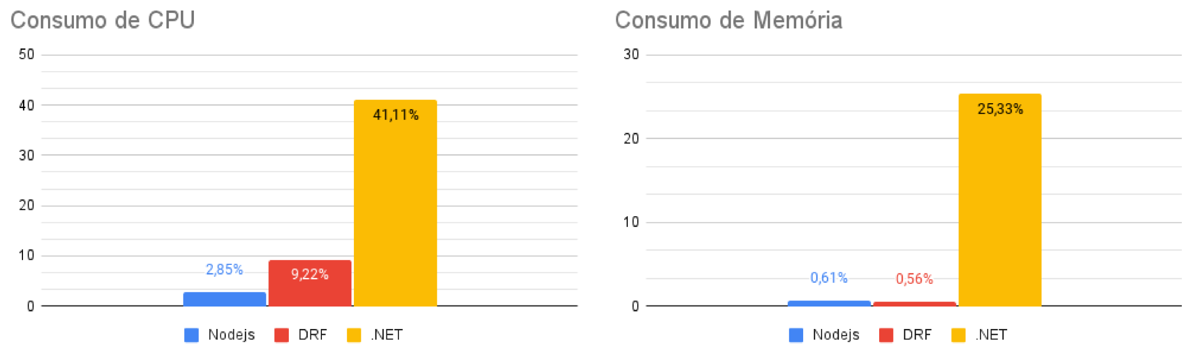

Figura 43 – Consumo médio de recursos - teste de resistência

QP_6. Assim como na QP_2, o Node.js apresentou menor consumo de CPU enquanto o DRF mostrou melhor resultado no consumo de RAM.

Respondendo a QP_7: *"Qual o framework mais otimizado com relação ao tempo de resposta no cenário de teste de resistência?"*. A Figura 44 apresenta os resultados médios do tempo de resposta agrupados por métodos HTTP das tecnologias durante os testes de resistência. Foi observado que todos os tempos ficaram abaixo dos 100ms, exceto o método PUT do DRF, ficando com média de 108,80ms. O `Node.js` e o `.Net` obtiveram os melhores tempos no geral, destacando o `.Net` que teve seu maior tempo no método POST com 12,30ms. O DRF, por sua vez, teve melhor resultado no método POST do que o `Node.js`.

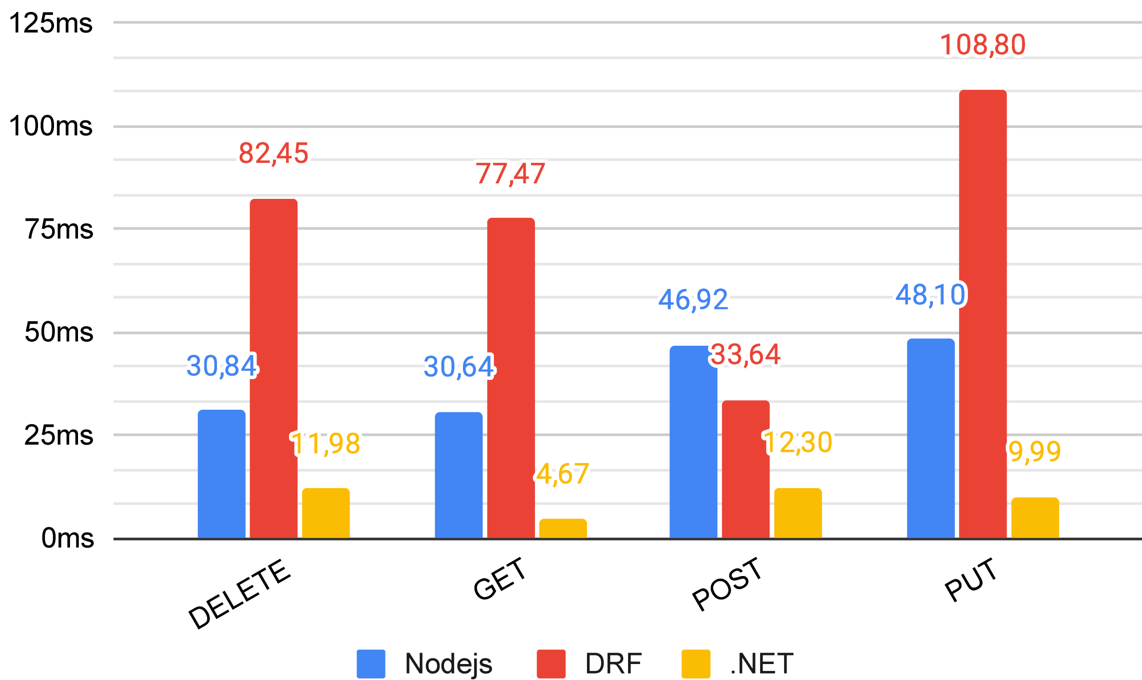

Figura 44 – Tempo de resposta médio dos métodos HTTP - teste de resistência

QP7. O .Net apresentou os melhores resultados em tempo de resposta em todos os métodos no cenário de resistência.

5.3 Outras observações

Em relação à comparação teórica, observou-se que todas as tecnologias ofereciam suporte adequado para a construção das APIs, conforme documentações oficiais disponíveis. No entanto, surgiram algumas dificuldades durante o desenvolvimento em .Net. Isso se deveu à natureza orientada a objetos do C#, linguagem utilizada para a criação das APIs, que demanda conhecimento em conceitos relativamente mais avançados. Isso contrastou com o `Node.js`, conhecido por sua abordagem funcional, e o DRF, que oferece flexibilidade no desenvolvimento, permitindo o uso de funções e classes.

Durante os testes práticos, identificou-se em alguns cenários que o `.Net`, apesar de apresentar os melhores tempos de resposta, exibiu um tempo de inicialização mais longo em comparação com as outras tecnologias. A Figura 45 ilustra um dos casos nos quais essa particularidade foi observada. Nela, é observado que a primeira requisição do conjunto, requisição POST, chega a ter um tempo de resposta acima de 200ms.

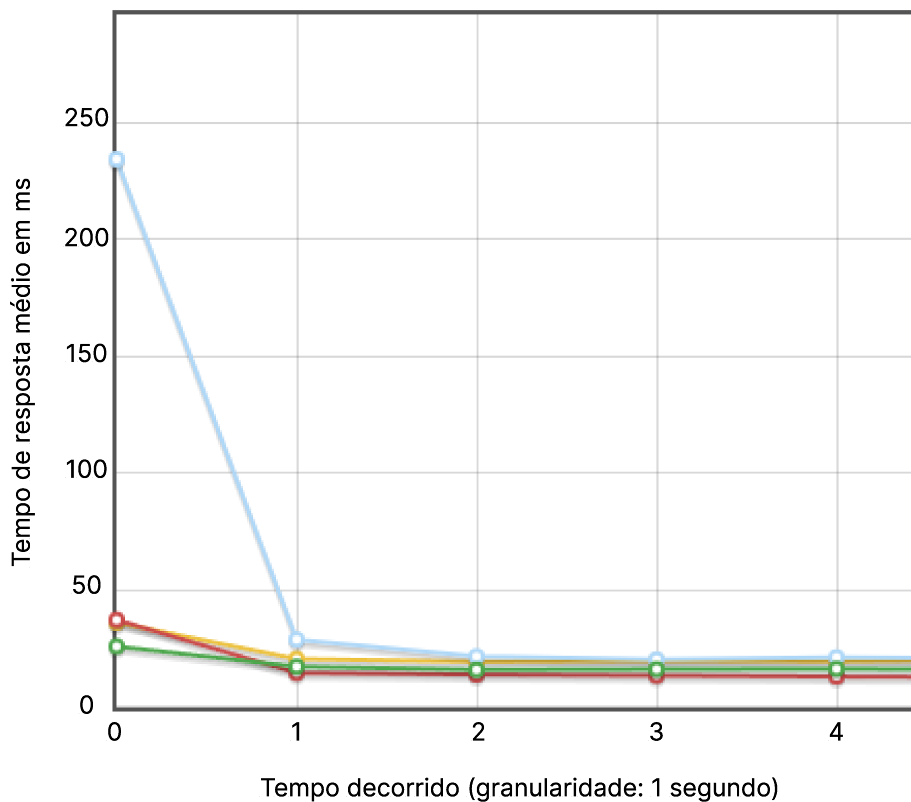

Figura 45 – Inicialização do .Net no teste de carga crescente

De uma maneira geral, as Figuras 46 e 47 exibem as médias dos resultados obtidos nos três testes realizados nas APIs. Observou-se que, em termos de consumo de recursos, o `Node.js` e o DRF demonstraram desempenho superior, destacando-se por sua eficiência tanto no consumo de CPU quando em consumo de RAM. No que diz respeito ao tempo de resposta, o `.Net` se destacou, apresentando os valores mais otimizados em todos os métodos e em todos os testes realizados. Esses resultados indicam que o `Node.js` e o DRF são mais eficientes no gerenciamento de recursos, enquanto o `.Net` se sobressai na agilidade de resposta.

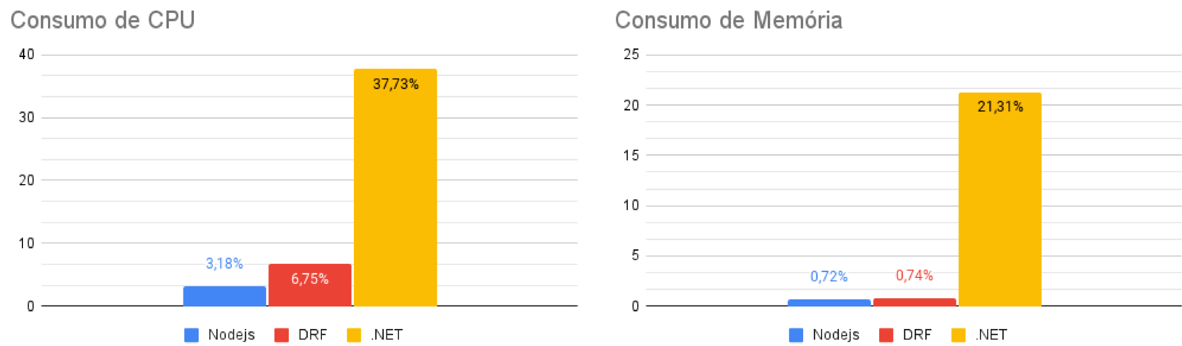

Figura 46 – Consumo médio de recursos - média total

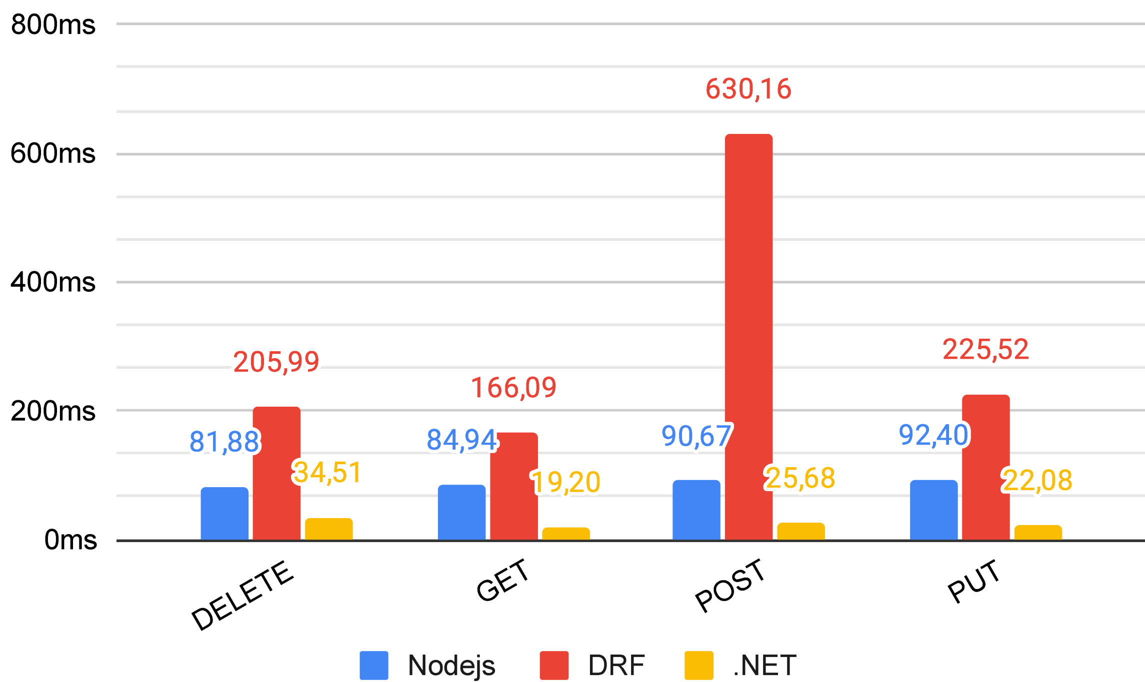

Figura 47 – Tempo de resposta médio dos métodos HTTP - média total

Foi observado também, durante os testes práticos, que nenhum dos *frameworks* demonstrou falhas nas requisições. Em todos os cenários avaliados, a taxa de erros de requisição foi igual a 0. Com estes resultados conclui-se que, para os cenários modelados, os *frameworks* possuem bom rendimento.

6 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho, realizou-se uma análise comparativa entre os *frameworks* de *back-end* `Node.js`, `Django REST Framework` e `.NET Core`, abordando tanto aspectos teóricos quanto práticos. A análise teórica baseou-se em uma série de critérios e atributos. Nesta comparação foram utilizados materiais teóricos baseados em publicações acadêmicas e documentação oficial dos próprios *frameworks*. Os pontos de comparação incluíram:

- Suporte a Sistemas Operacionais;
- Suporte a bancos de dados;
- Suporte a ORM nativo;
- Suporte a documentação nativa da API;

A análise prática envolveu a criação de três APIs nos *frameworks* em estudo. A comparação foi realizada com base em métricas de performance e desempenho, sendo elas consumo de recursos (CPU e Memória) e tempo de reposta. Essas métricas foram obtidas por meio da execução de três cenários de testes:

- Teste de pico;
- Teste de carga crescente;
- Teste de resistência;

Com relação aos dados teóricos, foi identificado que cada um dos *frameworks* apresenta diferentes funcionalidades, tendo suas próprias peculiaridades e especificidades. No entanto, é importante notar que, apesar das diferenças entre eles, todos os *frameworks* podem ser configuradas de acordo com as necessidades do projeto, seja através de suas próprias extensões e recursos internos, seja recorrendo a ferramentas externas. Isso destaca a flexibilidade desses *frameworks* e a capacidade de adaptação para atender a uma variedade de requisitos de desenvolvimento.

Durante a análise dos testes práticos, observou-se que tanto o `Node.js` e o `Django REST Framework` (DRF) apresentaram melhores resultados em termos de eficiência no consumo de recursos, demonstrando uma utilização mais econômica em relação ao uso de CPU e memória em comparação com o `.NET Core`. Por outro lado, o `.NET Core` se destacou ao demonstrar os menores tempos de resposta em todos os cenários de teste. Esses resultados destacam as diferentes forças e características de cada *framework*, com o `Node.js` e o DRF oferecendo uma eficiência de recursos e o `.NET Core` otimizando o desempenho e a latência nas respostas.

Como trabalhos futuros é sugerido a replicação deste estudo em outros cenários

de testes, para verificar como é o comportamento dos *frameworks* em diferentes casos. Em outra configuração também é possível a avaliação utilizando outros bancos de dados, como por exemplo algum banco não relacional. Por fim, é sugerida a comparação dos *Frameworks* em outros fluxos de requisições, como por exemplo a funcionalidade de login.

Em resumo, este estudo fornece uma visão acerca das diferenças entre os *frameworks* de *back-end* populares. Cada *framework* tem suas próprias vantagens e desvantagens, e a escolha ideal dependerá dos requisitos específicos do projeto e das restrições de recursos. Este estudo contribui para o entendimento das implicações teóricas e práticas das escolhas de *frameworks* de *back-end* e espera-se que ele ajude os desenvolvedores a tomar decisões informadas em seus projetos futuros.

REFERÊNCIAS

ABBAS, R.; SULTAN, Z.; BHATTI, S. N. Comparative analysis of automated load testing tools: Apache jmeter, microsoft visual studio (tfs), loadrunner, siege. In: **2017 International Conference on Communication Technologies (ComTech)**. [*S. l.*: *s. n.*], 2017. p. 39–44.

ALURA. **Django e Django Rest: Diferenças e semelhanças**. 2023. Acessado em 09-junho-2023, disponível em https://www.alura.com.br/artigos/django-django-rest-diferencas.

AMAZON. **O que é API RESTful?** 2023. Acessado em 09-junho-2023, disponível em https://aws.amazon.com/pt/what-is/restful-api/.

AMAZON. **O que é o Django?** 2023. Acessado em 09-junho-2023, disponível em https://aws.amazon.com/pt/what-is/django/.

ARTILLERY. **Artillery Best Practices**. 2023. Acessado em 10-setembro-2023, disponível em https://www.artillery.io/docs/get-started/best-practices.

BASILI, V.; ROMBACH, H. The tame project: towards improvement-oriented software environments. **IEEE Transactions on Software Engineering**, v. 14, n. 6, p. 758–773, 1988.

CABEZAS, I.; SEGOVIA, R.; CARATOZZOLO, P.; WEBB, E. Using software engineering design principles as tools for freshman students learning. In: **2020 IEEE Frontiers in Education Conference (FIE)**. [*S. l.*: *s. n.*], 2020. p. 1–5.

DJANGO. **Documentação Django**. 2022. Acessado em 20-abril-2023, disponível em https://www.djangoproject.com.

GEEKSFORGEEKS. **Django Project MVT Structure**. 2023. Acessado em 20-abril-2023, disponível em https://www.geeksforgeeks.org/django-project-mvt-structure/.

DRF. **Documentação do Django REST Framewok**. 2023. Acessado em 09-junho-2023, disponível em https://www.django-rest-framework.org/.

HACKERRANK. **Top Back-End Development Trends for 2023**. 2022. Acessado em 07-agosto-2023, disponível em https://www.hackerrank.com/blog/back-end-development-trends/.

HAOYU, W.; HAILI, Z. Basic design principles in software engineering. In: **2012 Fourth International Conference on Computational and Information Sciences**. [*S. l.*: *s. n.*], 2012. p. 1251–1254.

HOODA, I.; CHHILLAR, R. S. Article: Software test process, testing types and techniques. **International Journal of Computer Applications**, v. 111, n. 13, p. 10–14, February 2015. Full text available.

INSOMNIA. **Insomnia**. 2023. Acessado em 05 de julho de 2023: https://insomnia.rest/.

JIANG, Z. M.; HASSAN, A. E. A survey on load testing of large-scale software systems. **IEEE Transactions on Software Engineering**, v. 41, n. 11, p. 1091–1118, 2015.

KRYLOV, G.; PATROU, M.; DUECK, G. W.; SIU, J. The evolution of garbage collection in v8: Google's javascript engine. In: **2020 9th Mediterranean Conference on Embedded Computing (MECO)**. [*S. l.*: *s. n.*], 2020. p. 1–6.

LIANG, L.; ZHU, L.; SHANG, W.; FENG, D.; XIAO, Z. Express supervision system based on nodejs and mongodb. In: **2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS)**. [*S. l.*: *s. n.*], 2017. p. 607–612.

LIBUV. **Documentação Libuv**. 2014. Acessado em 11-junho-2023, disponível em https://docs.libuv.org/en/v1.x/design.html#the-i-o-loop.

MARTIN, R. C. **Código Limpo**. São Paulo: Alta Books, 2008.

MDN. **A web e seus padrões**. 2023. Acessado em 09-junho-2023, disponível em https://developer.mozilla.org/pt-BR/docs/Learn/Getting_started_with_the_web/The_web_and_web_standards.

MICROSOFT. **Arquitetura do .NET**. 2023. Acessado em 27-abril-2023, disponível em https://learn.microsoft.com/pt-br/dotnet/csharp/tour-of-csharp/#net-architecture.

MUSTAFA, K. M.; AL-QUTAISH, R. E.; MUHAIRAT, M. I. Classification of software testing tools based on the software testing methods. In: **2009 Second International Conference on Computer and Electrical Engineering**. [*S. l.*: *s. n.*], 2009. v. 1, p. 229–233.

PAGE, T. A. **Uma breve história do desenvolvimento Web**. 2023. Acessado em 09-junho-2023, disponível em https://pt.theastrologypage.com/brief-history-web-development#:~:text=Em%201989%2C%20Tim%20Berners-Lee,Language%20(HTML)%20em%201990.

ROBERTS, P. **What the heck is the event loop anyway?** 2014. Acessado em 11-junho-2023, disponível em https://2014.jsconf.eu/speakers/philip-roberts-what-the-heck-is-the-event-loop-anyway.html.

ROCKETSEAT. **Do back ao mobile: de onde surgiu a programação fullstack**. 2023. Acessado em 09-junho-2023, disponível em https://blog.rocketseat.com.br/do-back-ao-mobile-de-onde-surgiu-a-programacao-fullstack/#:~:text=Essa%20foi%20a%20norma%20atÃl’,como%20redes%20sociais%20e%20smartphones.

STACKOVERFLOW. **Stack Overflow Annual Developer Survey**. 2022. Acessado em 14-abril-2023, disponível em https://survey.stackoverflow.co/2022/#technology-most-popular-technologies.

V8. **Documentação V8 engine**. 2022. Acessado em 20-abril-2023, disponível em https://v8.dev/.

VALES, Z.; BRADA, P. Service api modeling and comparison: A technology-independent approach. In: **2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)**. [*S. l.*: *s. n.*], 2020. p. 158–161.

APÊNDICES

APÊNDICE A – JAVASCRIPT

A.0.1 Sintaxe do JavaScript

JavaScript é uma linguagem leve, interpretada e baseada em objetos com funções de primeira classe, mais conhecida como a linguagem de *script* para páginas Web, mas usada também em vários outros ambientes sem *browser*, como no Node.js. O JavaScript é uma linguagem baseada em protótipos, multi-paradigma e dinâmica, suportando estilos de orientação a objetos, imperativos e declarativos (como por exemplo a programação funcional).

No JavaScript temos, como principais sintaxe, a declaração de variáveis, condicionais, laços de repetições e funções.

A.0.1.1 Declaração de variáveis

Temos 3 tipos de declarações de variáveis em JavaScript, utilizamos as palavras-chave 'var', 'let' e 'const' para diferenciá-las, além da nomenclatura também se diferenciam no seu escopo e reatribuição de valores.

Nas variáveis do tipo 'var', é possível acessá-las em qualquer lugar do código, inclusive antes de sua declaração devido ao *hoisting*. Além disso, é possível reatribuir um novo valor a uma variável declarada com 'var'. Já no caso das variáveis do tipo 'let', elas são acessíveis apenas após a sua declaração e não permitem a redeclaração no mesmo escopo. Por fim, as variáveis do tipo 'const' são utilizadas para valores constantes que não podem ser reatribuídos após a sua inicialização, como o próprio nome sugere.

```
1 var myvar = "um valor para var";
2 let mylet = 5;
3 const PI = 3.14;
```

Listing A.0.1 – Exemplo de declaração de variável em *JavaScript*

A.0.1.2 Condicionais

As tomadas de decisões em JavaScript são feitas através de estruturas condicionais, que permitem diferentes fluxos de execução do código dependendo de uma condição especificada.

Podemos utilizar tanto a estrutura 'if' com a cláusula 'else', que permite executar um bloco de código se a condição for verdadeira e outro se for falsa, quanto a estrutura 'switch', que

permite avaliar uma expressão e executar diferentes blocos de código dependendo do seu valor.

```
/* Exemplo de codigo com a estrutura 'if' */
if (condicao) {
    declaracao_1;
} else if (condicao_2) {
    declaracao_2;
} else {
    declaracao_final;
}

/* Exemplo de codigo com a estrutura 'switch' */
switch (expressao) {
    case rotulo_1:
        declaracoes_1
        [break;]
    case rotulo_2:
        declaracoes_2
        [break;]
       ...
    default:
        declaracoes_padrao
        [break;]
}
```

Listing A.0.2 – Exemplo das estruturas condicionais *if* e *switch* em *JavaScript*

A.0.1.3 Laços de Repetições

Quando queremos replicar tarefas repetitivas utilizamos os laços de repetições, onde possuem, na maioria das vezes, um contador, condição de saída e um iterador.

No laço do tipo 'for', temos um inicializador, uma condição de saída e uma expressão final, onde o código no seu escopo se repete enquanto a condição de saída não for satisfeita. Já no laço 'while', é repetido um trecho de código enquanto sua condição é satisfeita. Tem-se também a variação 'do-while', onde é executado um trecho do código antes da verificação da expressão condicional.

```
var passo;
for (passo = 0; passo < 5; passo++) {
  // Executa 5 vezes, com os valores de passos de 0 a 4.
  console.log('Ande um passo para o leste');
}
```

Listing A.0.3 – Exemplo da do laço de repetição *for* em *JavaScript*

A.0.1.4 Funções

Para tarefas reutilizáveis, que são utilizadas em mais de um local ou situação, é possível transformá-las em funções, um bloco de código que não precisa ser reescrito toda vez que precisar ser executado. Podemos utilizar tanto funções já prontas de bibliotecas e módulos quanto criar nossas próprias funções.

```
function square(numero) {
  return numero * numero;
}

square("O quadrado de 2 eh: ", square(2));

>> O quadrado de 2 eh: 4
```

Listing A.0.4 – Exemplo do uso de funções em *JavaScript*

APÊNDICE B – PYTHON

B.0.1 Sintaxe do Python

Python é uma linguagem de programação poderosa e fácil de aprender. Possui estruturas de dados eficientes de alto nível e uma abordagem simples, mas eficaz, para programação orientada a objetos. A sintaxe elegante e a digitação dinâmica do Python, juntamente com sua natureza interpretada, o tornam uma linguagem ideal para *scripts* e desenvolvimento rápido de aplicativos em muitas áreas na maioria das plataformas.

No Python cada bloco de código é baseado em identação, onde cada escopo é definido através de espaços ou tabulações, diferente do JavaScript que utiliza as chaves ’’ para fazer essa delimitação.

B.0.1.1 Declaração de variáveis

Em Python, não é necessário declarar o tipo de uma variável antes de utilizá-la, pois o tipo é inferido automaticamente, sendo característico de linguagem fracamente tipada. Isso torna o processo de declaração de variáveis mais simples e direto. Ou seja, podemos declarar uma variável atribuindo um valor a ela, seja inteiro, *string*, *float*, lista ou outro tipo de dado.

B.0.1.2 Condicionais

Para a execução de um bloco de código baseado em decisões lógicas, o Python fornece uma estrutura condicional semelhante ao JavaScript, diferindo na sua sintaxe, onde a expressão condicional não precisa estar em volta de parênteses, necessitando apenas do caractere dois pontos ’:’ após a expressão.

```
x = int(input("Please enter an integer: "))
>> Please enter an integer: 42
if x < 0:
    x = 0
    print('Negative changed to zero')
elif x == 0:
    print('Zero')
elif x == 1:
    print('Single')
else:
    print('More')
>> More
```

Listing B.0.1 – Exemplo do uso da condicional *if elif* em Python

B.0.1.3 Laços de Repetições

Como no JavaScript, o Python também possui estrutura para a repetição de blocos de código, podendo ser tanto baseado em expressões lógicas ou para percorrer listas e tuplas, por exemplo.

```
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))

>> cat 3
>> window 6
>> defenestrate 12
```

Listing B.0.2 – Exemplo do uso do laço de repetição *for* em Python

B.0.1.4 Funções

A sintaxe da construção de uma função em Python é especificada pelo prefixo 'def', acompanhado do nome da função e finalizando com os dois pontos ':'. Todo o seu escopo é identado, como de padrão na linguagem.

```
def hello_people(name):
    print("Ola " + name)

hello_people(Claudia)

>>Ola Claudia
```

Listing B.0.3 – Exemplo do uso de funções em Python

APÊNDICE C – C#

C.0.1 Sintaxe do C#

C# é uma linguagem de programação fortemente tipada orientada a objetos e orientada a componentes. É utilizada por desenvolvedores para a criação de aplicativos seguros e robustos que são executados no .NET. Sua semântica e sintaxe tem raízes do C, outra linguagem de programação fortemente tipada.

Vários recursos do C# permitem a construção de aplicativos robustos e duráveis, entre as principais estão os tratamentos de exceções utilizado para a detecção e recuperação de erros, expressões lambda para uma programação com paradigma funcional e um sistema de tipos unificados, onde todos os tipos, incluindo os tipos primitivos, como int e double, herdam de um único tipo de objeto raiz, compartilhando um conjunto de operações em comum.

C.0.1.1 Paradigma POO

Por ser uma linguagem orientada a objetos, o C# segue o paradigma de programação orientada a objetos (POO). Sua construção e desenvolvimento são baseados em objetos, que são divididos em classes, e se comunicam entre si através de métodos.

É possível a utilização de vários conceitos deste paradigma, como a herança, encapsulamento e polimorfismo, no desenvolvimento em C#. O POO permite a modularização, reutilização e facilidade de manutenção do código. Além disso, o C# também suporta outros paradigmas, como o paradigma funcional e a programação assíncrona.

C.0.1.2 Declaração de variáveis

Por ser uma linguagem fortemente tipada, todas as variáveis, constantes e expressões têm um tipo definido. A definição de cada tipo em específico inclui vários conceitos, como as operações permitidas, seu espaço para armazenamento e seus valores máximos e mínimos. Em sua declaração a variável pode ter um tipo explícito, como int, double ou char, ou implícito com tipo var, neste caso o compilador determina automaticamente seu tipo com base na sua atribuição inicial.

```
// Tipado explicitamente.
int idade = 22;

// Tipado implicitamente.
var a = 10;
```

Listing C.0.1 – Exemplo de declaração de variável em C#

C.0.1.3 Condicionais

As instruções de seleção em C# são semelhantes às do JavaScript, permitindo ao programador definir blocos de instruções a serem executados dependendo de expressões condicionais. Elas podem ser combinadas para criar lógica mais complexa.

```
DisplayWeatherReport(15.0);  // Output: Cold.
DisplayWeatherReport(24.0);  // Output: Perfect!

void DisplayWeatherReport(double tempInCelsius) {
    if (tempInCelsius < 20.0) {
        Console.WriteLine("Cold.");
    }
    else {
        Console.WriteLine("Perfect!");
    }
}
```

Listing C.0.2 – Exemplo de uma estrutura condicional em C#

C.0.1.4 Laços de Repetições

Para a repetição de blocos de código o C# possui instruções de iteração que executam determinado pedaço de código repetidas vezes. Dentre elas estão o ’for’, executando um bloco de código enquanto a expressão especificada for verdadeira, ’foreach’, fazendo a execução de determinado trecho de código para cada elemento de uma coleção, e o ’do while’, que podem ser utilizados de forma conjunta ou independente, repetindo nenhuma ou mais vezes um bloco de código.

```
for (int i = 0; i < 3; i++) {
    Console.Write(i);
}
>> 012

//Exemplo da instrução de iteração 'do while'
int n = 0;
do {
    Console.Write(n);
    n++;
} while (n < 5);
```

Listing C.0.3 – Exemplo da instrução de iteração 'for' em C#

C.0.1.5 Funções e métodos

Quando desejamos criar um bloco de código e não ficar repetindo o mesmo dentro da aplicação, criamos uma função, ou método, devido ao paradigma 'POO' do C#. Ao ser invocado, o método executa uma série de instruções com os argumentos especificados, podendo retornar ou não um valor.

```
public class ExemploQuadradoNumero {
    static int Quadrado(int numero) {
      return numero * numero;
   }

   public static void Main() {
      int numero = 4;
      int numeroQuadrado = Quadrado(numero);
      Console.WriteLine("Quadrado: ", numeroQuadrado);
   }

   >> Quadrado: 16
}
```

Listing C.0.4 – Exemplo do uso de funções em C#