IFGOIANO - CAMPUS MORRINHOS
CURSO SUPERIOR TECNOLOGIA EM SISTEMAS DE INTERNET

JESSICA FERREIRA SILVA

UMA SOLUÇÃO DE MICROSSERVIÇO PARA GESTÃO DE DADOS DE VIGILÂNCIA EPIDEMIOLÓGICA

Morrinhos - GO
2023

JÉSSICA FERREIRA SILVA

UMA SOLUÇÃO DE MICROSSERVIÇO PARA GESTÃO DE DADOS DE VIGILÂNCIA EPIDEMIOLÓGICA

Artigo Científico apresentado ao Curso Superior de Tecnologia em Sistemas de Internet do Instituto Federal Goiano – Campus Morrinhos.
Área de concentração: Ciência da Computação
Orientador: Prof. Dr. Alexandre Carvalho Silva
Coorientador: Prof. Dr. Fernando Silva Matos

Morrinhos - GO
2023

AGRADECIMENTOS

Recebi apoio significativo de várias pessoas, cujas contribuições merecem destaque:

Primeiramente, gostaria de expressar minha gratidão ao meu irmão, Bruno Ferreira Silva, que desempenhou um papel fundamental ao me orientar, ensinar e apoiar ao longo de todo o período acadêmico, não permitindo que eu desistisse.

Também gostaria de agradecer ao meu marido, Wanderson Araújo de Alcantara, por seu apoio contínuo ao longo de toda a realização deste trabalho.

Minha amiga de classe, Dione Brandão, merece especial reconhecimento. Juntas, enfrentamos dias difíceis e incertos, marcados por uma pandemia sem precedentes. Quando pensei em desistir, ela me incentivou a persistir.

Agradeço Ana Paula Oliveira, que participou ativamente da ideia de projeto, programando paralelamente a interface e aplicativo ao qual se comunica com a API.

Agradeço à Yasmin Cunha, amiga, professora, bióloga e profissional agente de endemias na cidade de Caldas Novas, por sua valiosa contribuição.

Agradeço o professor Dr. Alexandre Carvalho, cujos conselhos e orientações foram de extrema importância.

Deixo também meu agradecimento e minha gratidão ao Professor Fernando, que exemplificou constantemente a dedicação e o comprometimento como docente desta instituição, além de transmitir conhecimento e orientação de forma exemplar.

E por último, e talvez o mais sublime de todos, o agradecimento à minha mãe, que infelizmente não está mais entre nós, mas que sempre acreditou que poderíamos realizar esse sonho juntas. Dedico a ela mais essa conquista.

EPÍGRAFE

“Não importa o que aconteça, continue a nadar.”
(WALTERS, GRAHAM; PROCURANDO NEMO, 2003.)

UMA SOLUÇÃO DE MICROSSERVIÇO PARA GESTÃO DE DADOS DE VIGILÂNCIA EPIDEMIOLÓGICA

[1] **Jéssica Ferreira Silva** Programa de Graduação em Tecnologia em Sistemas De Internet (Morrinhos/IF Goiano)

E-mail:jheferreiramath@gmail.com

[2] **Alexandre Carvalho Silva** Programa de Graduação em Tecnologia em Sistemas De Internet (Morrinhos/IF Goiano)

E-mail: alexandre.silva@ifgoiano.edu.br

[3] **Fernando Barbosa Matos** do Programa de Graduação em Tecnologia em Sistemas De Internet (Morrinhos/IF Goiano)

E-mail: fernando.matos@ifgoiano.edu.br

RESUMO

Este artigo propõe a criação de uma Interface de Programação de Aplicações (API) para aprimorar a vigilância epidemiológica, um componente crucial na prevenção e controle de doenças. A API visa otimizar a monitorização e análise de dados relacionados a doenças, com o objetivo de aumentar a eficácia das iniciativas de saúde pública. A estrutura da API é projetada para facilitar o armazenamento e a recuperação eficiente de informações epidemiológicas. Ela disponibilizará endpoints e rotas para acessar estatísticas gerais, detalhes de casos individuais, notificações em tempo real e relatórios personalizados. Essa abordagem permitirá que profissionais de saúde, pesquisadores e autoridades governamentais acessem rapidamente dados pertinentes, o que auxiliará na tomada de decisões e na aplicação de medidas de controle. A pesquisa e implementação desta API na área de vigilância epidemiológica têm o potencial de contribuir para o progresso da saúde pública, fornecendo informações precisas e atualizadas que podem ser empregadas no controle de doenças, na formulação de políticas e no desenvolvimento de estratégias de prevenção eficazes.

Palavras-chave: API , Arquitetura de Software , Dados, Desenvolvimento de Software, Rest.

ABSTRACT

Epidemiological surveillance plays a key role in disease prevention and control. This article proposes the creation of an API (Application Programming Interface) to improve the monitoring and analysis of epidemiological data, aiming to improve the effectiveness of public health actions. The API's data structure will be designed to allow efficient storage and retrieval of epidemiological information. The API will offer endpoints and routes to query general statistics, individual case details, real-time notifications and custom reports. This approach allows healthcare professionals, researchers and government authorities to have quick and easy access to relevant data, assisting in decision-making and the implementation of control measures.

It is hoped that this API research and implementation in the area of epidemiological surveillance will contribute to the advancement of public health by providing accurate and up-to-date information that can be used in disease control, policy formulation and the development of effective prevention strategies.

Keywords: API, Software Architecture, Data, Software Development, Rest.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

API	Application Program Interface
CU	Caso de uso
HTTP	Hypertext Transfer Protocol
ID	Identificador Único
IP	Internet Protocol
JS	JavaScript
JSON	JavaScript Object notation
REST	Representational State Transfer
RF	Requisitos Funcionais
RNF	Requisitos Não Funcionais
TCP	Transmission Control Protocol
TLS	Transport Layer Security

SUMÁRIO

INTRODUÇÃO

Embora a Vigilânica Epidemiológica seja um campo consolidado desde meados da década de 60, sua relevância e debate ganharam destaque significativo por volta dos anos 2020, após discussões em torno da epidemia do Covid-19. Segundo OPAS (Organização Pan-Americana da Saúde), a pandemia global trouxe à tona a necessidade urgente de rastrear, monitorar e responder a doenças tranmisssíveis em uma escala nunca antes vista. De acordo com a Lei 8.080, de 19 de Setembro de 1990, refere-se à conceito de vigilância epidemiológica: "é o conjunto de ações que proporciona o conhecimento, a detecção ou prevenção de qualquer mudança nos fatores determinantes e condicionantes de saúde individual ou coletiva, com a finalidade de recomendar e adotar as medidas de prevenção e controle das doenças ou agravos."

Para elaboração deste artigo , incialmente foi adotado como cenário a cidade de Caldas Novas- GO, por ser um destino turístico de renome está intrinsecamente ligada à sua característica distintiva de possuir uma grande quantidade de piscinas termais, oferecendo aos visitantes uma experiência relaxante e agradável. No entanto, o clima quente e úmido da região também cria um ambiente propício para a proliferação do mosquito *Aedes aegypti*, vetor da dengue. A gestão da saúde pública em Caldas Novas enfrenta o desafio adicional de equilibrar o turismo e a proteção da saúde local, implementando medidas rigorosas de controle da dengue para garantir a segurança dos residentes e dos milhares de turistas que desfrutam da beleza natural da cidade.

De acordo com pesquisa feita na cidade com os agentes de saúde, a abordagem predominante tem sido a coleta manual de informações, registradas em planilhas impressas, um processo que consome uma quantidade substancial de tempo e recursos. Essas informações, uma vez coletadas, são posteriormente digitalizadas e armazenadas em arquivos físicos, o que cria barreiras para análises eficazes e a tomada de decisões.

Este artigo tem como objetivo a criação de um programa dedicado à inserção eficiente de informações epidemiológicas, o desenvolvimento de uma aplicação, uma API -Application Program Interface, *JavaScript Object Notation* sendo ela uma fonte provedora de dados. Essa solução será projetada com um foco na segurança, em síntese, incorporando medidas de criptografia de dados. A aplicação poderá ser acessível a

diversos interessados, independentemente da linguagem de programação que utilizem, promovendo uma comunicação direta com futuras aplicações desenvolvidas . Além disso, a solução suportará coleta, manipulação e mensuração de informações epidemiológicas, estabelecendo uma base para o planejamento e a prevenção de doenças, bem como a tomada de decisões futuras.

2 FUNDAMENTAÇÃO TEÓRICA

Segundo dissertação Fielding, (2000), "Uma API é baseada em bibliotecas e fornece um conjunto de pontos de entrada de código e símbolos/ conjuntos de parâmetros para que um programador possa usar o código de outra pessoa para fazer o "trabalho sujo" de manter a interface real entre sistemas semelhantes, desde que o programador obedeça às restrições de arquitetura e linguagem que acompanham esse código." A suposição é que todos os lados da comunicação usam a mesma API e, portanto, o interior da interface é importante apenas para o desenvolvedor da API e não para o desenvolvedor do aplicativo que também incorporam uma variedade de protocolos e formatos de dados, como por exemplo, o JSON. De acordo com a publicação do documento ECMA-262 3ª edição - dezembro de 1999 *"JavaScript Object Notation* (JSON) é um formato leve de intercâmbio de dados. JSON é fácil de ler e escrever para humanos. JSON é fácil para as máquinas analisarem e gerarem. JSON é baseado em um subconjunto da linguagem de programação JavaScript, padrão. JSON é um formato de texto completamente independente de linguagem, mas usa convenções que são familiares aos programadores da família C de linguagens, incluindo C , C++, C#, Java™, JavaScript, Perl, Python e muitos outros. Essas propriedades tornam o JSON uma linguagem ideal para intercâmbio de dados." Portanto, JSON se tornou um formato amplamente adotado para a comunicação entre sistemas.

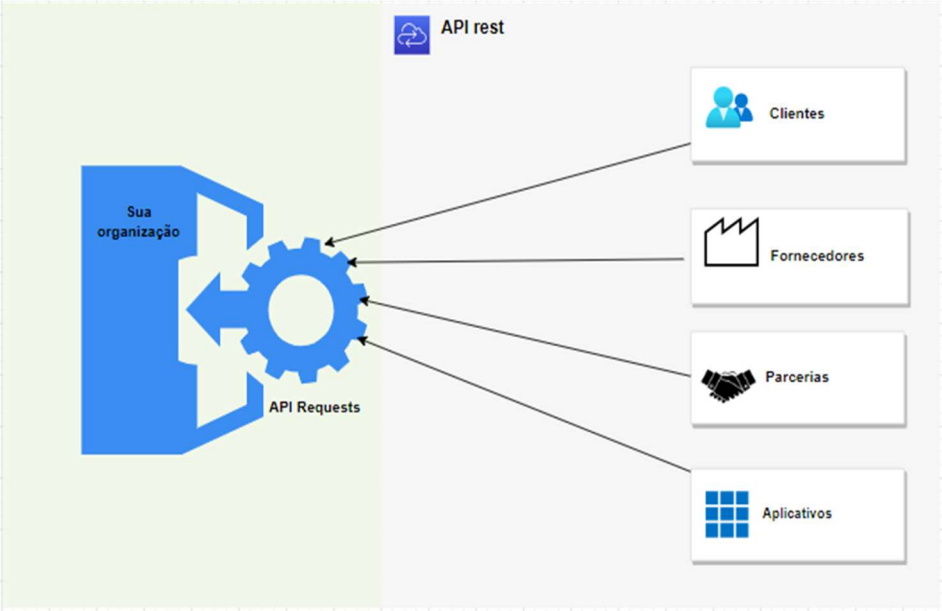
Conforme relatado por, J. Fowler (2017), "Embora os conceitos fundamentais por trás dos microsserviços não sejam novos, a aplicação contemporânea da arquitetura de microsserviços é atual, e sua adoção tem sido motivada em parte pelos desafios de escalabilidade, falta de eficiência, velocidade lenta de desenvolvimento e dificuldades em adotar novas tecnologias que surgem quando sistemas complexos de software estão contidos, e são implantados como uma grande aplicação monolítica". Assim, a

importância das APIs da Web cresceu exponencialmente com o advento da arquitetura orientada a microsserviços. Essas tendências incentivaram a modularização de sistemas em componentes independentes, cada um oferecendo uma API claramente definida para interações externas.

As APIs da Web modernas representam a convergência de teorias arquitetônicas como REST *Representational State Transfe*. De acordo com Fielding (2000), "Não houve problemas significativos causados pela introdução das novas normas, embora estas tenham sido sujeitas a alterações graduais e implantação fragmentada junto com aplicativos Web legados. Além disso, tiveram um efeito positivo na robustez da Web e permitiram novas métodos para melhorar o desempenho percebido pelo usuário por meio de hierarquias de cache e redes de distribuição de conteúdo". A API RESTful é uma interface que dois sistemas de computador usam para trocar informações de forma segura pela internet. A maioria das aplicações de negócios precisa se comunicar com outras aplicações internas e de terceiros para executar várias tarefas. Elas suportam essa troca de informações porque seguem padrões de comunicação de software seguros, confiáveis e eficientes. Elas são um reflexo da necessidade de comunicação eficiente e padronizada entre sistemas, impulsionando a inovação tecnológica e a criação de ecossistemas interconectados. À medida que a tecnologia continua a avançar, as APIs da Web desempenharão um papel fundamental na forma como as aplicações são projetadas, desenvolvidas e implantadas, facilitando a colaboração entre diferentes plataformas e serviços.

Segue na Figura 1. Exemplo de uma API, uma breve representação esquemática referente a comunicação de sistemas por meio de APIs.

Figura 1. Exemplo de uma API



Fonte: Própria autoria

Na Figura 1 os termos Cliente, Fornecedor, Parcerias e Aplicativos apresentam um importante papel no processo, são eles:

Clientes: neste contexto, são os usuários finais ou consumidores dos serviços ou informações disponibilizados por meio da API. Os clientes podem ser indivíduos ou organizações que acessam os recursos ou dados fornecidos pela API por meio de aplicativos ou sistemas.

Fornecedores: se referem às empresas ou sistemas que disponibilizam a API para outros aplicativos, desenvolvedores ou parceiros utilizarem. No contexto de uma API, o fornecedor é responsável por criar, manter e documentar a API, além de garantir que ela seja segura e eficiente. Os fornecedores podem oferecer a API de forma gratuita ou paga, dependendo da estratégia de negócios. Eles também podem definir políticas de uso, limites de acesso e autenticação para proteger a API e seus recursos.

Parcerias: referem-se a outras empresas ou organizações com as quais a empresa que fornece a API colabora para oferecer serviços complementares ou integrados. No contexto de uma API, as parcerias podem utilizar a API para criar integrações entre seus

produtos e serviços e os da empresa que fornece a API. Isso permite que ambas as partes se beneficiem da cooperação, expandindo suas ofertas e alcançando novos públicos.

Aplicativos: são programas de software projetados para executar tarefas específicas em dispositivos eletrônicos, como smartphones, tablets e computadores. No contexto de uma API (Interface de Programação de Aplicativos), os aplicativos são os consumidores da API. Eles utilizam a API para acessar recursos ou serviços específicos oferecidos por outra aplicação ou sistema. Por exemplo, um aplicativo endereços pode utilizar uma API para obter dados de localizações de uma fonte externa e exibi-los para o usuário.

2.1 Comunicação

A API é uma interface entre diferentes programas de software ou serviço. Sua comunicação é dada por meio do Protocolo HTTP, onde clientes e servidores se comunicam transacionando mensagens individuais em oposição a um fluxo dados. As mensagens enviadas pelo cliente, são chamadas de ***requests*****,** e em contrapartida, as mensagens enviadas como resposta do servidor são chamadas de ***responses*****.**

2.2 Protocolo HTTP

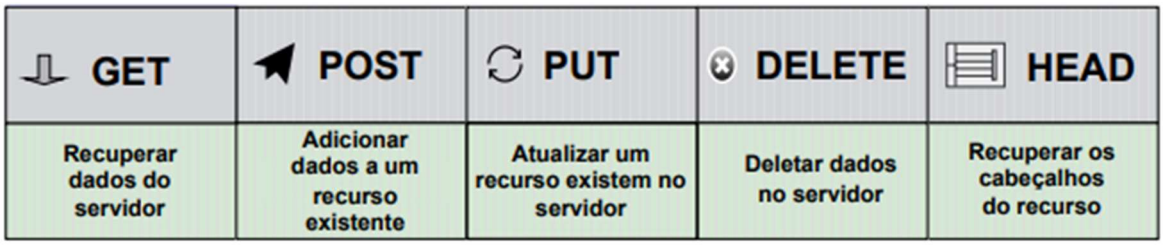
A primeira comunicação bem sucedida entre um cliente HTTP e o servidor através da internet surgiu ao final dos anos 1990, criado por Timothy John Berners-Lee, um físico britânico, cientista da computação e professor. Berners-Lee é o fundador da World Wide Web, e em Abril de 2009 foi eleito como membro da Academia Nacional de Ciências dos Estados Unidos. Em 2017, recebeu o Prêmio Turing, considerado o "Nobel da Computação". De acordo com Lee, o protocolo HTTP é a base que permite a troca de dados na Web por meio de um protoco cliente-servidor. São efetuadas requisições por um destinatário, que geralmente provém de um navegador web, ao mesmo tempo que um servidor envia respostas. A camada de aplicação que é enviado por TCP (*Transmission Control Protocol*) ou por uma conexão TCP criptografada por TLS (*Transport Layer Security*), entretanto, o HTTP não requer que o protocolo de transporte utilizado seja baseado em conexões, só requer que seja confiável ou não perca mensagens.

Uma conexão é controlada na camada de transporte, portanto, fora do controle do HTTP, embora qualquer protocolo de transporte confiável possa teoricamente ser usado. Considerando o seu caráter extenso, é utilizado não apenas para buscar documentos de hipertexto, mas também para imagens e vídeos ou para publicar conteúdo em servidores, como nos resultados de formulários HTML.

2.3 Requests Methods:

O método *GET* (buscar), recupera um recurso especificado (uma lista ou um único recurso). Se não houver erros, o método retorna uma representação do recurso em JSON. O método *POST* (publicar) modifica o estado do servidor, pois cria novos recursos. Por exemplo, quando se faz login em um site, o login envia as credenciais para o servidor usando uma solicitação POST. O método *DELETE* é usado para excluir (ou remover) um recurso existente, como exemplo, excluir um determinado item em um aplicativo da Web. O método *HEAD* (cabeça) recupera os cabeçalhos do recurso , sem o próprio recurso. É usado, por exemplo , para verificar modificações no lado do servidor. Segue abaixo como exemplo a Figura 2. HTTP Requests methods.

Figura 2. HTTP Requests Methods:



Fonte: Própria autoria

Os principais métodos HTTP correspondentes às operações CRUD. Esse é o nome dado a um acrônimo para *Create*, *Read*, *Update* e *Delete*, que são as quatro operações

básicas que podem ser realizadas em um sistema de gerenciamento de banco de dados DBMS - "*Database Management System"*. **Create (Criação):** permite a criação de novos registros no banco de dados. **Read (Leitura)**: permite a leitura e visualização de registros existentes no banco de dados. **Update (Atualização)**: permite a atualização de registros existentes no banco de dados. **Delete (Exclusão)**: permite a exclusão de registros existentes no banco de dados. O CRUD é comumente utilizado em sistemas de informação e é fundamental para a gestão e manutenção de dados em um banco de dados.

3 ARQUITETURA E DETALHES DE IMPLEMENTAÇÃO

3.1 Definição dos Requisitos

A definição dos requisitos é uma etapa fundamental em qualquer projeto de desenvolvimento de software, incluindo a implementação de um sistema de API.

O primeiro passo para definir a implementação e arquitetura do sistema foi definir seus requisitos. Isso inclui identificar as funcionalidades que a API deveria oferecer, bem como os formatos de entrada e saída dos dados. Essas definições foram feitas por meio de entrevistas, questionários e análise de dados. Envolveu-se todas as partes interessadas no processo de definição dos requisitos, para garantir que todas as necessidades fossem consideradas e para evitar retrabalho ou problemas de comunicação.

3.2 Requisitos funcionais do usuário

Requisitos funcionais, são declarações de serviços que o sistema deve fornecer, de como o sistema deve reagira a entradas específicas e de como o sistema deve se comportar em determinadas situações. Em alguns casos, os requisitos funcionais também podem explicitar o que o sistema não deve fazer. (SOMMERVILLE, 2020).

RF1: O sistema deve permitir cadastrar um usuário.
RF2: O sistema deve permitir procurar todos os usuários cadastrados.

RF3: O sistema deve permitir procurar usuários por meio de um identificador;

RF4: O sistema deve permitir alterar os dados de um usuário por meio de um identificador;

RF5: O sistema deve permitir deletar um usuário por meio de um identificador;

RF6: O sistema deve permitir autenticação de usuários autorizados;

RF7: O sistema deve permitir usuários cadastrarem um morador;

RF8: O sistema deve permitir pesquisar todos os moradores cadastrados.

RF9: O sistema deve permitir pesquisar moradores por meio de identificador;

RF10: O sistema deve permitir alterar dados de um morador.

RF11: O sistema deve permitir deletar o cadastro de um morador por meio de um identificador.

RF12: O sistema deve permitir registrar amostras encontradas;

RF13: O sistema deve permitir registrar a quantidade de amostras encontradas no local

RF14: O sistema deve permitir o registro de depósitos onde encontrou-se a amostra.

RF15: O sistema deve permitir cadastrar data de retorno de visitas.

RF16: O sistema deve permitir procurar todos os endereços cadastrados.

RF17: O sistema deve permitir alterar os dados de endereços cadastrados.

RF18: O sistema deve permitir deve permitir deletar um endereço cadastrado.

3.3 Requisitos não funcionais

Requisitos não funcionais. São restrições aos serviços ou funções oferecidos pelo sistema. Incluem restrições de *timing,* restrições no processo de desenvolvimento e restrições impostas pelas normas. Ao contrário das características individuais ou serviços do sistema, os requisitos não funcionais, muitas vezes, aplicam-se ao sistema como um todo. (SOMMERVILLE, 2020).

Os requisitos não funcionais são requisitos que não estão diretamente relacionados com os serviços específicos oferecidos pelo sistema a seus usuários, eles são compostos de duas partes: função e comportamento. Em detalhes, temos como esperado o

desempenho: O sistema deve demonstrar a capacidade de atender ao número requerido de usuários sem sofrer qualquer deterioração em seu desempenho. A **segurança**: sistema deve ser protegido contra acesso não autorizado e manter seguro os dados de qualquer usuário e também clientes cadastrados em sua base. A **manutenibilidade**: seu código é de fácil análise, a reutilização do mesmo para expansão é uma característica. Sua manutenção é simples, devido suas classes estarem muito bem definidas, evitando que uma só classe tenha diversas funções. Os testes são de fácil acesso, e intuitivos. A **portabilidade**: é facilmente adaptável em diferentes plataformas. Sua instalação é fácil e intuitiva. E a **usabilidade**: sistema deve ser protegido contra erros, os métodos muito bem definidos, a fim de que um usuário acidentalmente não exclua um cadastro indevido, por exemplo. A aprendizagem deve ser de fácil entendimento. A manipulação de recursos deverá ser feita por meio de representação JSON.

3.4 Tecnologia

O próximo passo foi escolher a tecnologia NodeJs, que foi uma escolha popular para muitas aplicações web porque é rápido, escalável, usa uma linguagem comum, tem um ecossistema de pacotes robusto, uma comunidade ativa e é multiplataforma. “Como um tempo de execução JavaScript assíncrono orientado a eventos, o Node.js foi projetado para construir aplicativos de rede escalonáveis. No exemplo "olá mundo" a seguir, muitas conexões podem ser tratadas simultaneamente. A cada conexão, o retorno de chamada é acionado, mas se não houver trabalho a ser feito, o Node.js irá dormir”.(Node.js, 2023).

Node.js é uma tecnologia de desenvolvimento web que permite executar JavaScript no servidor. Uma das principais vantagens do Node.js é que ele é muito eficiente em lidar com operações de entrada/saída de dados, como requisições HTTP, conexões com banco de dados e acesso a sistemas de arquivos. Isso porque o Node.js utiliza um modelo de programação assíncrona e não bloqueante, o que permite que ele realize múltiplas tarefas simultaneamente sem precisar esperar a conclusão de uma operação para iniciar outra. Além de possuir uma grande quantidade de pacotes (bibliotecas) disponíveis através do gerenciador de pacotes NPM *(Node Package Manager*), o que facilita muito o desenvolvimento de aplicações web robustas e escaláveis. ”.(Node.js, 2023)

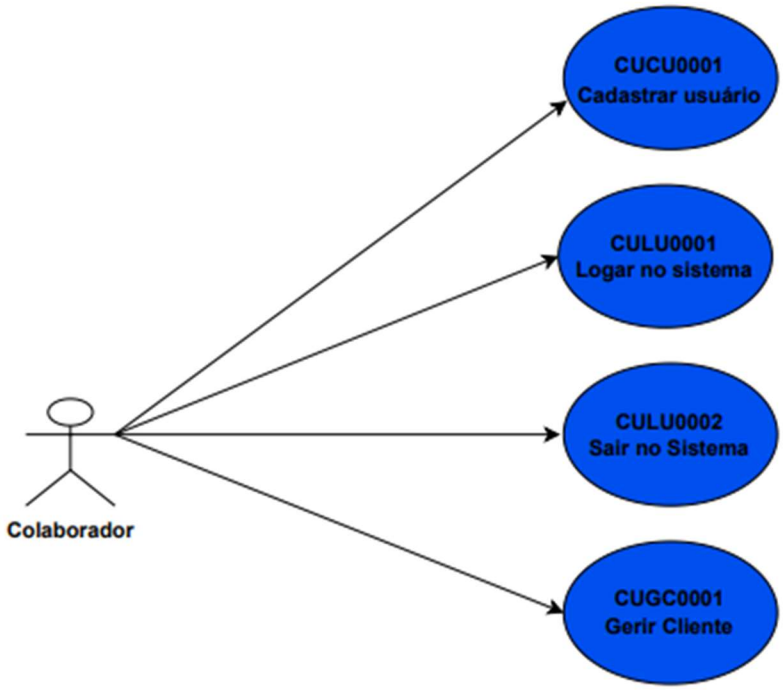
3.5 Banco de dados

Um dos problemas levantados desde o início do projeto seria onde armazenar as informações. Por se tratar de um trabalho acadêmico, decidiu-se por vincular as informações local, contudo, a aplicação pode ser armazenada em servidores, de acordo com a escolha do consumidor do aplicativo, e demanda do cliente a ser contratado. O banco de dados escolhido para armazenar as informações MongoDB, um banco de dados NoSQL orientado a documentos, desenvolvido pela empresa MongoDB Inc. Entre as principais vantagens do MongoDB, estão a capacidade de escalar horizontalmente, ou seja, adicionar mais servidores para aumentar a capacidade de armazenamento e processamento de dados; a alta disponibilidade e tolerância a falhas, que garantem que o banco de dados continue funcionando mesmo em caso de falhas de hardware ou rede; e a flexibilidade para alterar o esquema de dados sem precisar interromper o serviço ou fazer migrações complexas. (Mongo,2023)

3.6 Diagrama de Caso de Uso

De acordo com Ivar Jacobson, (2004) podemos dizer que um caso de uso é um "documento narrativo que descreve a sequência de eventos de um ator que usa um sistema para completar um processo". Um diagrama de caso de uso é uma ferramenta de modelagem visual usada em engenharia de software para representar as interações entre os usuários (atores) e um sistema em um determinado contexto. É composto por atores, casos de uso e relacionamentos entre eles. É uma ferramenta importante para a definição de requisitos de um sistema. Ele pode ajudar a identificar e esclarecer as necessidades dos usuários e a entender como o sistema deve funcionar em diferentes situações, como no exemplo abaixo, a Figura 3 Diagrama de Caso de Uso do Sistema.

Figura 3. Diagrama de Caso de Uso do Sistema



Fonte: Própria autoria

A seguir serão detalhados os casos de uso da Figura 3:

CUCU0001: Caso de Uso Cadastrar Usuário: este caso de uso representa a funcionalidade que permite aos usuários registrarem-se no sistema. Os usuários podem inserir informações pessoais, como nome, endereço de e-mail e senha, para criar uma conta no sistema.

CULU0001: Caso de Uso Logar Usuário: esse caso de uso descreve o processo em que os usuários existentes fazem login no sistema. Os usuários inserem suas credenciais (nome de usuário e senha) para acessar suas contas.

CULU0002: Caso de Uso Logout Usuário: esse caso de uso representa a funcionalidade de sair do sistema. Quando os usuários optam por fazer logout, eles encerram sua sessão e não podem mais acessar as funcionalidades restritas.

CUGC0001: Caso de Uso Gerir Cliente: este caso de uso abrange a gestão de

clientes no sistema. Ele permite que os usuários administrem informações relacionadas aos clientes. As funcionalidades incluem a inserção, alteração, busca e exclusão de registros de clientes no sistema. Essa funcionalidade é essencial para a gestão de informações de clientes de forma eficaz.

3.7 Documentar a API

Após a implementação, realizou-se a padronização da documentação da API no programa Postman, que é uma ferramenta que ajuda a padronizar a documentação de APIs, facilitando a compreensão e uso por outros desenvolvedores. Com uma documentação bem estruturada e completa, é possível reduzir o tempo gasto em comunicação e aumentar a eficiência do processo de desenvolvimento. A documentação no Postman permitiu definir uma estrutura padronizada para a API. O que garante consistência nas solicitações, nos cabeçalhos, nos parâmetros e nas respostas, e facilita o desenvolvimento e a manutenção de aplicativos que se integram à API.(Postman, 2023).

4 IMPLEMENTAÇÃO E TESTES

Após a elaboração da arquitetura do sistema, o programa foi codificado conforme requisitos funcionais e modelos de casos de uso, iniciando assim a fase de testes. Essa etapa marca um ponto crítico no ciclo de desenvolvimento de software, onde a qualidade e a segurança do sistema começam a ser rigorosamente avaliadas.

Os testes desempenham um papel multifacetado nesse processo, abordando diversos aspectos essenciais. Primeiramente, eles visam garantir que o sistema atenda aos requisitos funcionais definidos, assegurando que todas as funcionalidades se comportem conforme o esperado. Além disso, os testes têm como objetivo identificar quaisquer problemas ou inconsistências que possam surgir durante a execução do programa. Eles remetem a redução de custos, à resolução de problemas, ao não retrabalho e aos custos associados, uma vez que problemas encontrados em estágios iniciais de desenvolvimento são mais fáceis e baratos de corrigir. Também foi validado a questão de vulnerabilidades afim de garantir que a API esteja protegida contra ataques maliciosos, minimizando o

risco de violação de dados e protegendo a privacidade dos usuários e dados de clientes. Além dos testes de unidade realizados em todas as classes, os testes de integração, validação de comunicação entre uma classe e outra e também os testes de sistema, que trata de unificar todo o processo anterior;

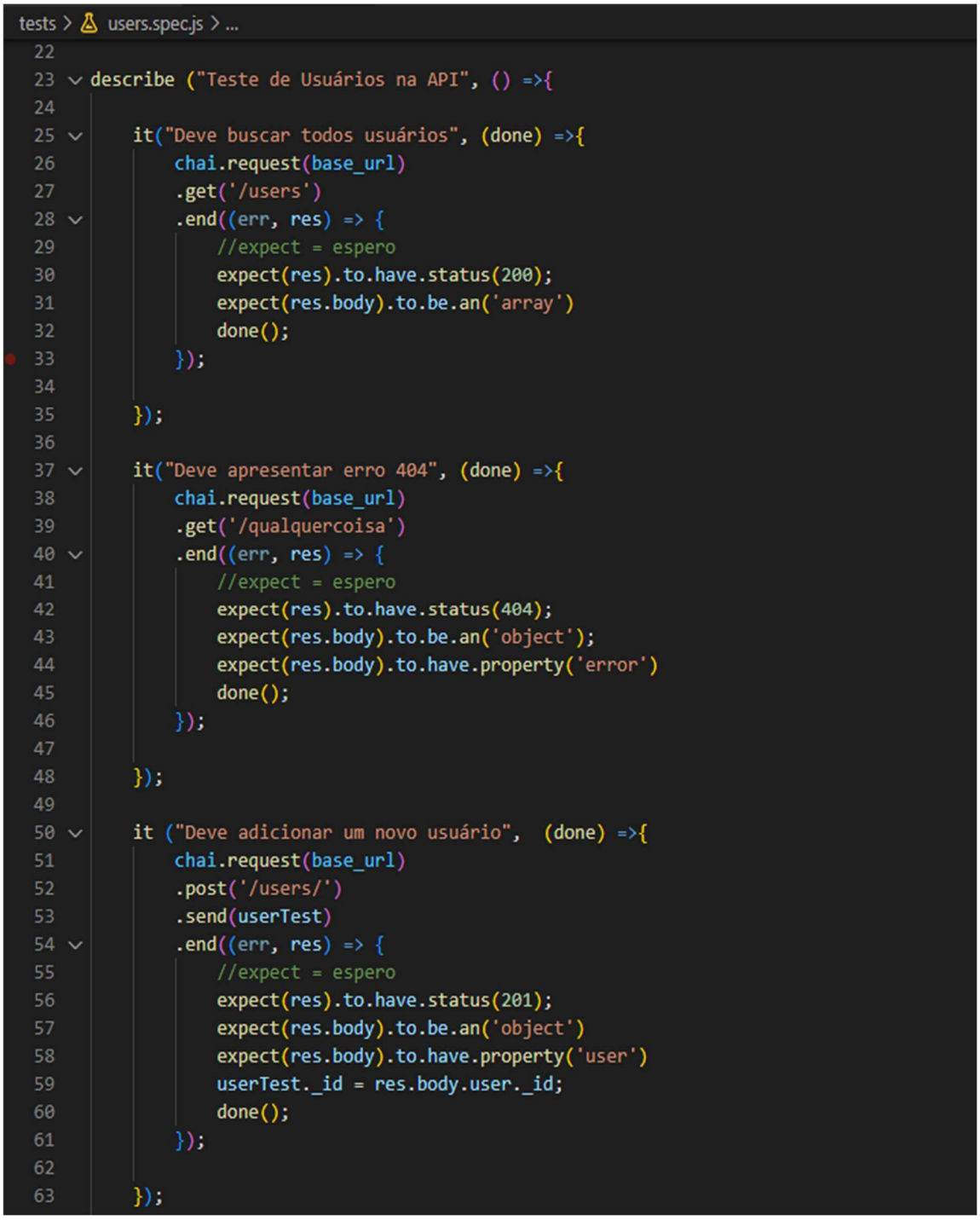
Por fim, os testes de aceitação, que consistem em validar se as solicitações do usuário estão de acordo com os requisitos funcionais apresentados no item Arquitetura e Detalhes de Implementação, (esse teste geralmente é realizado pelo usuário).

Nos primeiros testes realizados, os testes de unidade foram realizados dentro da própria estrutura do programa, no arquivo, com nome de users.spec.js. O arquivo "spec.js" é um arquivo de teste escrito em JavaScript, que segue a convenção de nomenclatura utilizada pelo framework de testes de JavaScript, o Jasmine. Nesse tipo de arquivo, o desenvolvedor pode escrever testes para verificar o comportamento de funções, módulos, componentes ou outras partes de um aplicativo ou sistema. Esses testes são executados automaticamente pelo framework de teste e podem ajudar a detectar erros ou problemas de lógica no código. O arquivo "spec.js" geralmente contém uma ou mais suítes de teste, que são grupos lógicos de testes relacionados, e dentro de cada suíte pode haver várias especificações (specs) que são os testes individuais que serão executados. Cada especificação geralmente contém uma ou mais expectativas, que são as asserções que verificam se o comportamento do código está correto. Esse arquivo disponibiliza importar bibliotecas disponíveis na linguagem de programação estruturando testes na API sem necessariamente popular o banco de dados.

A biblioteca utilizada, chaiHttp é uma extensão da biblioteca de asserções Chai para testes de integração de APIs HTTP. Ela fornece uma API fácil de usar para testar solicitações HTTP e respostas, permitindo que os desenvolvedores escrevam testes de integração robustos e confiáveis para suas APIs. No arquivo foram automatizados testes de unidade no cadastro de usuários, prevendo possíveis erros de usuários ao tentar realizar um determinado cadastro.

Nas figuras 4 e 5 são apresentados os testes de unidade:

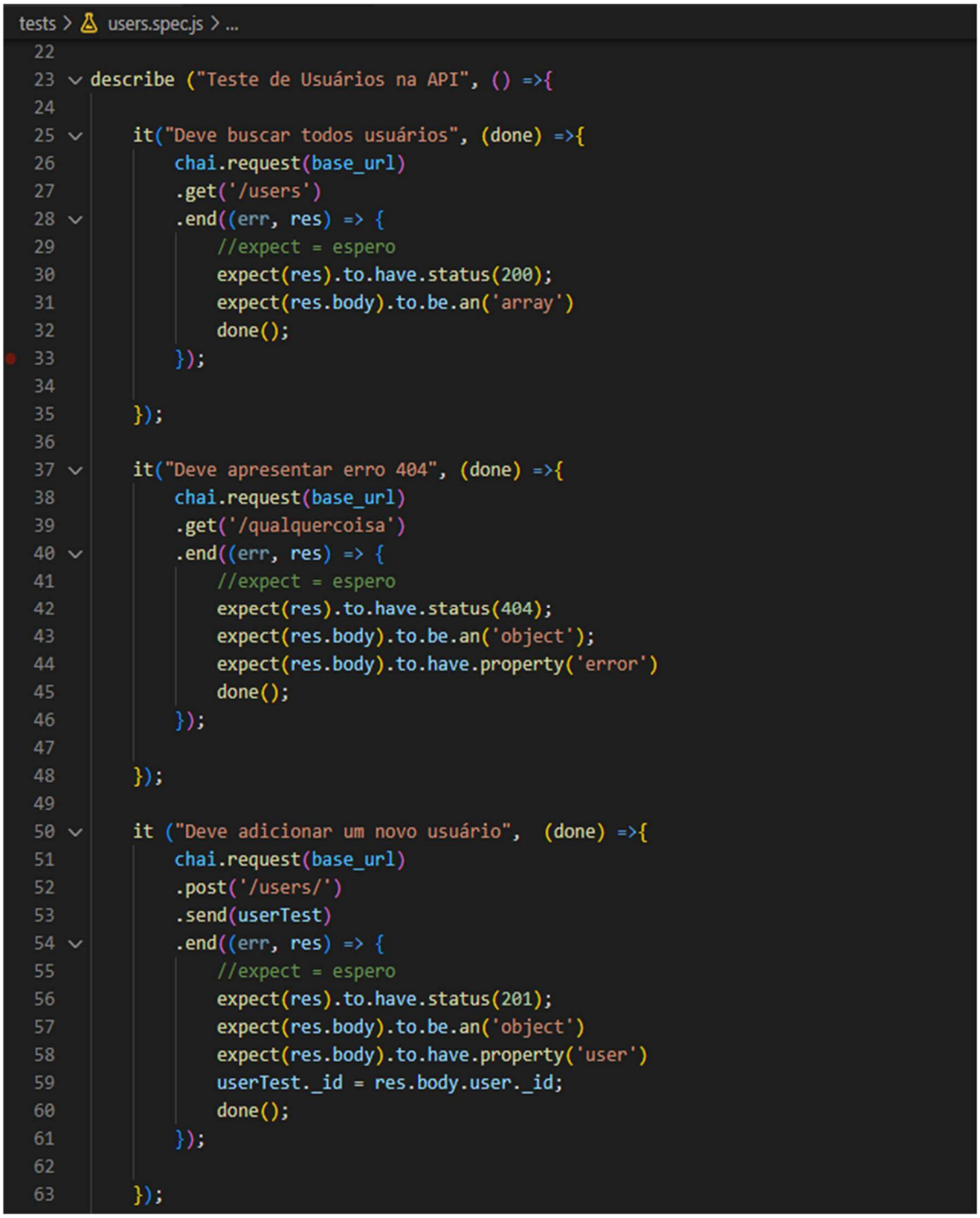
Figura 4: Arquivo users.spec.js A



```
tests > users.spec.js > ...
22
23  describe ("Teste de Usuários na API", () =>{
24
25      it("Deve buscar todos usuários", (done) =>{
26          chai.request(base_url)
27          .get('/users')
28          .end((err, res) => {
29              //expect = espero
30              expect(res).to.have.status(200);
31              expect(res.body).to.be.an('array')
32              done();
33          });
34
35      });
36
37      it("Deve apresentar erro 404", (done) =>{
38          chai.request(base_url)
39          .get('/qualquercoisa')
40          .end((err, res) => {
41              //expect = espero
42              expect(res).to.have.status(404);
43              expect(res.body).to.be.an('object');
44              expect(res.body).to.have.property('error')
45              done();
46          });
47
48      });
49
50      it ("Deve adicionar um novo usuário",  (done) =>{
51          chai.request(base_url)
52          .post('/users/')
53          .send(userTest)
54          .end((err, res) => {
55              //expect = espero
56              expect(res).to.have.status(201);
57              expect(res.body).to.be.an('object')
58              expect(res.body).to.have.property('user')
59              userTest._id = res.body.user._id;
60              done();
61          });
62
63      });
```

Fonte: Própria autoria

Figura 5: Arquivo users.spec.js B



```
tests > users.spec.js > ...
22
23  describe ("Teste de Usuários na API", () =>{
24
25      it("Deve buscar todos usuários", (done) =>{
26          chai.request(base_url)
27          .get('/users')
28          .end((err, res) => {
29              //expect = espero
30              expect(res).to.have.status(200);
31              expect(res.body).to.be.an('array')
32              done();
33          });
34
35      });
36
37      it("Deve apresentar erro 404", (done) =>{
38          chai.request(base_url)
39          .get('/qualquercoisa')
40          .end((err, res) => {
41              //expect = espero
42              expect(res).to.have.status(404);
43              expect(res.body).to.be.an('object');
44              expect(res.body).to.have.property('error')
45              done();
46          });
47
48      });
49
50      it ("Deve adicionar um novo usuário",  (done) =>{
51          chai.request(base_url)
52          .post('/users/')
53          .send(userTest)
54          .end((err, res) => {
55              //expect = espero
56              expect(res).to.have.status(201);
57              expect(res.body).to.be.an('object')
58              expect(res.body).to.have.property('user')
59              userTest._id = res.body.user._id;
60              done();
61          });
62
63      });
```

Fonte: Própria autoria

5 CONCLUSÕES E TRABALHOS FUTUROS

Após identificar a necessidade de automatizar processos na área da saúde, a API BackVigilance, projetada para adiministrar e armazenar dados relacionados à Vigilância Epidemiológica, visa oferecer soluções que otimizem o fluxo de informações entre os profissionais de saúde e os cidadãos. Por meio da automação de tarefas, a API busca agilizar a coleta, o armazenamento e a análise de dados epidemiológicos, fornecendo um panorama abrangente e em tempo real das condições de saúde da população.

Ao implantar essa tecnologia, espera-se que ocorra uma melhoria substancial na qualidade do atendimento, na tomada de decisões e na prevenção de doenças. A API BackVigilance possibilitará uma maior integração entre os diferentes setores da saúde, permitindo a troca rápida e precisa de informações entre hospitais, clínicas, laboratórios e autoridades de saúde. Além disso, a API visa facilitar o acesso dos agentes aos serviços de saúde, proporcionando um canal de comunicação direto e simplificado. No entanto, é importante ressaltar que este trabalho representa apenas um ponto de partida, abrindo caminho para a contínua pesquisa e aprimoramento no campo da arquitetura de software. Ao término desse trabalho, já é previsto sugestões e melhorias à implementar:

- ✓ Tabelas com informações específicas para o cenário, como por exemplo, possibilitar ao usuário selecionar um campo de checkbox, onde haja a possibilidade do usuário selecionar um ou múltiplos itens simultanemanete com as informações de larvas quando encontradas e também os depósitos(locais onde foram encontradas amostras), passado préviamente, com seus respectivos identificadores.
- ✓ Outras possibilidades de pesquisas, criar buscas além das opções de nome e identificador do morador, como cpf, endereço e data de registro.
- ✓ Nas listas de FormOpenHouse e FormClosedHouse, criar os campos de pesquisa por data, setor e tipo de amostrass encontradas.

Por fim, é esperado que este estudo possa contribuir para a consolidação da área de arquitetura de software, fornecendo subsídios teóricos e práticos para o desenvolvimento de sistemas mais eficientes, escaláveis e seguros, capazes de acompanhar as demandas e transformações da era digital.

6 APÊNDICES

O intuito desse material de apêndice e a inclusão de links para os testes realizados permite que os leitores interessados em explorar detalhes específicos dos testes desejam entender os métodos utilizados ou verificar os resultados por si mesmos.

6.1 Teste A

Consultas aos testes salvos em banco de dados local, disponível visualização nos links abaixo:

Classe Cadastro do Usuário, diponível em: https://youtu.be/4hpfsbZiSkE

6.1 Teste B

Classe Login, disponível em: https://youtu.be/Av71e8NhzWc

6.1 Teste C

Após efetuado o login do usuário cadastrado no sistema, o mesmo agora como agente poderá fazer o cadastro de toda visita efetuada, inserindo no sistema os dados de cadastro da visita. Quando a visita é cadastrada com sucesso, se encontrado alguma espécie de praga, o agente insere a quantidade e o tipo de depósito em que foi encontrado, persistindo as informações no banco de dados.

Classe Form Open House, disponível em: https://youtu.be/0IMKn9QebbY

6.1 Teste D

Caso o agente de saúde faça uma visita não conclusiva, ou seja, o morador não encontrava-se na casa, ou não pode atendê-lo por algum motivo, o agente poderá registrar a visita, informando uma possível data de retorno.

Classe Form Closed House, disponível em: https://youtu.be/Jx55JW_aEuA

6.2 Link Documentação da API:

Disponível em**:** https://documenter.getpostman.com/view/20304674/2s93m7VgEB

7 REFERÊNCIAS BIBLIOGRÁFICAS

[BARBOSA, Jarbas], **Organização Pan-Americana da Saúde.** Disponível em: https://www.paho.org/pt/noticias/1-2-2021-relatorio-global-da-oms-destaca-necessidade-urgente-dados-melhores-para . Acesso em: (17/09/2023).

[BRASIL, Fundação Nacional da Saúde], **Guia de Vigilância Epidemiológica.** Fundação Nacional de saúde. 5. ed. Brasília: Funasa, (2002). Disponível em: https://bvsms.saude.gov.br/bvs/publicacoes/funasa/guia_vig_epi_vol_l.pdf . Acessado em: (17/092023).

[ECMA], Standard ECMA -262 **Standardizing Information and Communication Systems.** 3. ed.(1999). Disponível em: https://www.ecma-international.org/wp-content/uploads/ECMA-262_3rd_edition_december_1999.pdf . Acesso em: (17/09/2023).

[FIELDING, Roy Thomas], **Architectural Styles and the Design of Network-based Software Architectures**. Doctoral dissertation. University of California, Irvine, (2000). Disponível em: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf . Acesso em: (17/09/2023).

[FOWLER, Susan J], **Microsserviços prontos para a produção: Construindo Sistemas Padronizados em uma Organização de Engenharia de Software.** Novatec Editora, (2017).

[JACOBSON, Ivar], **Aspect-Oriented Software Development with Use Cases** 1.ed. Addison Wesley, (2004).

[MERRIMAN, Dwight]; [HOROWITZ, Eliot]; [RYAN, Kevin], **MongoDB (2007)**. Disponível em: https://www.mongodb.com/pt-br/company. Acesso em: (17/09/2023).

[NODE.Js], Disponível em: https://nodejs.org/en/about. Acesso em: (17/09/2023).

[POSTMAN], Disponível em: https://www.postman.com/. Acesso em (17/09/2023).

[SOMMERVILLE, Ian], **Engenharia de Software** 9. ed. São Paulo: Pearson Prentice Hall, (2011).

[LEE, Tim Berners], **Biografia**. Disponível em: https://www.w3.org/People/Berners-Lee/Overview.html. Acesso em: (19/07/2023).