UNIVERSIDADE DE CAXIAS DO SUL
CENTRO DE CIÊNCIAS EXATAS E DA TECNOLOGIA

AUGUSTO ZANESCO BORTONCELLO

OPEN HEALTH SYNC: SERVIÇO OPEN SOURCE PARA INTEGRAÇÃO DE DADOS DA SAÚDE ENTRE SISTEMAS

BENTO GONÇALVES

2025

AUGUSTO ZANESCO BORTONCELLO

OPEN HEALTH SYNC: SERVIÇO OPEN SOURCE PARA INTEGRAÇÃO DE DADOS DA SAÚDE ENTRE SISTEMAS

Monografia apresentada como requisito para a obtenção do grau de Bacharel em Ciência da Computação da Universidade de Caxias do Sul.

Orientador: Prof. Dr. Leonardo Pellizzoni

BENTO GONÇALVES

2025

AUGUSTO ZANESCO BORTONCELLO

OPEN HEALTH SYNC: SERVIÇO OPEN SOURCE PARA INTEGRAÇÃO DE DADOS DA SAÚDE ENTRE SISTEMAS

Monografia apresentada como requisito para a obtenção do grau de Bacharel em Ciência da Computação da Universidade de Caxias do Sul.

Aprovado em 24/11/2025

BANCA EXAMINADORA

Prof. Dr. Leonardo Pellizzoni
Universidade de Caxias do Sul - UCS

Profa. Dra. Helena Graziottin Ribeiro
Universidade de Caxias do Sul - UCS

Prof. Dr. Daniel Luis Notari
Universidade de Caxias do Sul - UCS

RESUMO

O uso crescente de tecnologias no cotidiano, impulsionado pela disseminação de dispositivos móveis e vestíveis, como os *smartwatches*, tem transformado significativamente a forma como informações pessoais são coletadas, analisadas e utilizadas. Apesar do potencial desses dados para apoiar diagnósticos e decisões clínicas, sua descentralização e heterogeneidade representam desafios à consolidação e ao uso clínico efetivo. Este trabalho propõe e implementa uma arquitetura de *software open source* para a integração de dados da saúde, com o foco principal no armazenamento, organização, acesso e interoperabilidade à estes dados, de modo que, dispositivos externos possam gravar, acessar e gerenciar o conteúdo enviado. Uma implementação é desenvolvida como prova de conceito e funcionalidade, a fim de realizar testes locais que comprovem o funcionamento, a robustez, a segurança e a autenticidade da solução. O software resultante é disponibilizado publicamente sob a licença *MIT*, permitindo seu uso, modificação e distribuição por outros pesquisadores e desenvolvedores, reforçando o caráter colaborativo e aberto da proposta. Essa disponibilização visa fomentar novas aplicações e aprimoramentos futuros, bem como facilitar a integração da solução em sistemas de maior escala voltados à gestão de dados em saúde.

Palavras-chave: Arquitetura de Software. Integração de Dados. API. Open-Source. Segurança. Interoperabilidade em Saúde.

ABSTRACT

The increasing use of technology in everyday life, driven by the spread of mobile and wearable devices such as smartwatches, has significantly transformed the way personal information is collected, analyzed, and utilized. Despite the potential of these data to support diagnostics and clinical decision-making, their decentralization and heterogeneity pose challenges to effective consolidation and clinical use. This work proposes and implements an open-source software architecture for health data integration, focusing on the storage, organization, access, and interoperability of such data, enabling external devices to record, access, and manage the transmitted content. An implementation is developed as a proof of concept and functionality, aiming to conduct local tests that demonstrate the system's performance, robustness, security, and authenticity. The resulting software is publicly released under the MIT License, allowing its use, modification, and distribution by other researchers and developers, thereby reinforcing the collaborative and open nature of the proposal. This public availability aims to encourage new applications and future improvements, as well as to facilitate the integration of the solution into larger-scale systems focused on health data management.

Keywords: Software Architecture. Data Integration. API. Open Source. Security. Health Interoperability.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API *Application Programming Interface*
APK *Android Application Package*
AWS *Amazon Web Services*
BPMN *Business Process Model and Notation*
EHR *Electronic Health Record*
EHIS *Electronic Health Information Systems*
FHIR *Fast Healthcare Interoperability Resources*
GDPR *General Data Protection Regulation*
HIPAA *Health Insurance Portability and Accountability Act*
HL7 *Health Level Seven International*
HTTP *HyperText Transfer Protocol*
IaaS *Infrastructure as a Service*
IOC *Inversion of Control*
IOS *iPhone Operating System*
IPC *Inter-Process Communication*
JSON *JavaScript Object Notation*
JPA *Java Persistence API*
JWT *JSON Web Token*
LAI Lei de Acesso à Informação
LGPD Lei Geral de Proteção de Dados Pessoais
MDM *Master Data Management*
MOM *Message-Oriented Middleware*
MVC *Model View Controller*
ODM *Object Data Modeling*
PaaS *Platform as a Service*
PHD *Personal Health Data*
REST *Representational State Transfer*
RNDS Registro Nacional de Dados de Saúde
SaaS *Software as a Service*
SDK *Software Development Kit*
SGBD Sistema de Gerenciamento de Banco de Dados
SQL *Structured Query Language*

UML *Unified Modeling Language*

XML *Extensible Markup Language*

YAML *YAML Ain't Markup Language*

SUMÁRIO

1 INTRODUÇÃO

O mundo moderno está cada vez mais interconectado, com uma crescente dependência da produtividade promovida pelas ferramentas tecnológicas em diversas áreas do cotidiano. Na saúde, o uso dessas ferramentas permite melhorar a assertividade sobre diagnósticos e quadros clínicos (PELLIZZONI; FALAVIGNA, 2024). Uma tecnologia que se mantém em crescimento são os *smartwatches*, usados frequentemente em conjunto com os celulares, amplamente utilizados no monitoramento da saúde e de atividades físicas (HOSSEINI *et al.*, 2023).

No mercado de tecnologia móvel, os *smartwatches* estão crescendo à medida que dispositivos médicos convergem para monitorar a saúde e o bem estar das pessoas. Nas pesquisas, mostram-se muito promissores na detecção de doenças cardíacas, distúrbios do movimento e até mesmo sinais precoces de COVID-19. Esses dispositivos coletam dados contínuos sobre os usuários, como batimentos cardíacos, níveis de oxigênio no sangue, passos dados durante o dia e outras informações vitais (HOSSEINI *et al.*, 2023).

Na área da saúde, a falta de estruturação e acessibilidade dos dados clínicos pode levar à perda de informações valiosas, prejudicando a precisão da avaliação médica e criando riscos tanto para os pacientes, que podem receber tratamentos baseados em dados incompletos, quanto para o sistema, que perde eficiência e oportunidades de inovação. Para superar essa fragmentação, é essencial que os dados sejam integrados e disponibilizados de forma segura e eficiente, funcionando como uma base confiável para decisões clínicas e gestão em saúde (PRICE; NICHOLSON, 2017).

Dentre as vantagens dos softwares, destacam-se a agilidade, organização e segurança dos dados, especialmente na área da saúde, onde é fundamental lidar com informações críticas de pacientes. A utilização desses sistemas melhora a qualidade do atendimento, favorecendo um atendimento mais ágil e eficaz. Além disso, esses sistemas ajudam as organizações a se diferenciarem no mercado e fornecem os dados necessários para otimizar o trabalho dos profissionais de saúde (COSTA[1]; ORLOVSKI, 2011).

Moura e Pessôa (1999), Agha (2014) comentam que, a tecnologia da informação tem proporcionado aos profissionais da saúde a possibilidade de melhorar a qualidade do atendimento, reduzir erros frequentes, aumentar a receita da empresa e, principalmente, diminuir os custos decorrentes de erros, como desperdício de materiais e recursos. Além disso, os sistemas utilizados permitem que as informações coletadas sejam mais confiáveis para toda a equipe, tanto operacional quanto clínica. (BATES; GAWANDE, 2003) comenta também que, a tecnologia em setores fora o da saúde, proporcionou o que ele chamou de "*mass customization*", produção eficiente e confiável de serviços baseados nas altas necessidades personalizadas dos clientes individuais.

As ferramentas permitem que tarefas sejam executadas de maneira mais eficiente e otimizada. No contexto tecnológico, elas funcionam como facilitadores, simplificando processos para seus usuários. A engenharia de software, por sua vez, evoluiu significativamente, desempenhando um papel de grande importância, com aplicações que são amplamente projetadas e integradas ao cotidiano da sociedade (SOMMERVILLE, 2011).

O contexto de desenvolvimento de aplicações está amplamente ligado a este trabalho, com o foco principal na criação de uma arquitetura e uma interface de integração que seja capaz de recepcionar, tratar e armazenar dados de saúde, permitindo que terceiros, como médicos, hospitais ou aplicativos de saúde, possam acessar e utilizar essas informações para fins de diagnóstico e monitoramento. A interface de integração, em conjunto com a arquitetura proposta, visa oferecer uma solução robusta, segura e autêntica, garantindo que os dados sejam armazenados de forma organizada e que o acesso à eles seja controlado e eficiente.

Para validar a funcionalidade da interface, desenvolveu-se uma implementação inicial como prova de conceito, permitindo a realização de testes locais que comprovem a eficácia, a segurança e a escalabilidade da solução. Essa implementação servirá como base para futuras integrações em sistemas maiores, podendo ser utilizada como uma camada de abstração para o gerenciamento de dados de saúde.

Para auxiliar o desenvolvimento, escolheu-se a metodologia *ICONIX* na elaboração das provas de conceito da interface proposta. Essa metodologia foi aplicada na implementação de operações básicas na interface, como autenticação, obtenção e salvamento de dados. O desenvolvimento foi conduzido por meio de diagramas, exemplos práticos de implementação e documentação contínua ao longo do projeto.

1.1 QUESTÃO DE PESQUISA

É possível sincronizar e integrar dados de diferentes estruturas entre sistemas de múltiplas origens de forma autenticável e *open source*, em conformidade com a Lei Geral de Proteção de Dados Pessoais (LGPD)?

1.2 OBJETIVO

Projetar e desenvolver uma arquitetura para integrar dados da saúde entre sistemas que, viabilize a segurança, a escalabilidade e que seja *open source*, de modo que possa guardar dados da saúde provenientes de diferentes estruturas, e que seja possível obtê-los de forma íntegra de diferentes locais e dispositivos autorizados, contribuindo para a evolução do ecossistema de dados de saúde em conformidade com a LGPD.

O trabalho tem como objetivos específicos:

a) Projetar e implementar uma arquitetura *open source* de integrar dados da saúde provenientes de várias origens e em diferentes estruturas de forma escalável;

b) Testar a integração como prova de conceito, realizando os testes diretamente no nível de comunicação entre sistemas, sem a necessidade de um aplicativo conectado;

c) Adicionar mecanismos de segurança que assegurem o armazenamento e a integridade dos dados;

d) Elaborar e disponibilizar documentação detalhada para a implementação de diversos softwares e dispositivos;

e) Assegurar a conformidade da solução com os princípios da LGPD no tratamento de dados de saúde.

1.3 ESTRUTURA DO TRABALHO

O presente capítulo apresentou uma introdução sobre o trabalho, incluindo seus objetivos, motivações e o cenário base para a ideia. No Capítulo 2, será apresentado o referencial teórico, dando ênfase na arquitetura de software de sistemas distribuídos, sem dar muita ênfase nas tecnologias utilizadas, mas fornecendo uma base teórica fundamental para o entendimento da proposta e os motivos das escolhas das mesmas. Os trabalhos correlatos são apresentados no Capítulo 3, abordando os trabalhos e ferramentas com forte relação com o presente trabalho. No Capítulo 4, será apresentada uma proposta de solução para o problema apresentado, juntamente com as representações das arquiteturas, diagramas, tecnologias e *frameworks*, além da explicação de como o software será disponibilizado e licenciado. O Capítulo 5 aborda o processo de desenvolvimento da solução proposta, apresentando as etapas de implementação, e os desafios enfrentados durante a construção da proposta. No Capítulo 6, são descritos os testes realizados, separando os testes feitos na solução e os testes desenvolvidos para assegurar a confiabilidade do código. Por fim, no Capítulo 7 apresenta as considerações finais, discutindo as conclusões alcançadas no trabalho, alguns desafios e possíveis aprimoramentos e trabalhos futuros.

2 FUNDAMENTOS

Quando um software é produzido, e seu papel é cumprido com sucesso suprindo as necessidades das pessoas envolvidas, performando por um longo período de tempo e conseguindo-se modificá-lo de forma rápida e escalável, o mesmo consegue mudar o contexto das pessoas envolvidas para o melhor (ROZANSKI; WOODS, 2012). Porém, caso o oposto ocorra, pode-se transformar aquilo que seria uma solução em um completo fracasso, trazendo insatisfação aos usuários. Nesse contexto, uma abordagem de engenharia faz-se necessária, para que o design e o projeto sejam bem construídos (PRESSMAN, 2005).

A engenharia de software, pode ser definida como uma tecnologia em camadas para projetar e desenvolver sistemas, comprometendo-se organizadamente com a qualidade. Para isso, deve-se seguir uma filosofia que promova o aperfeiçoamento contínuo dos processos. A base fundamental da engenharia de software é o foco na qualidade, como mostra a Figura 1 abaixo (PRESSMAN, 2005).

Figura 1 – Camadas da Engenharia de Software

Fonte: Adaptado de (PRESSMAN, 2005)

Para que se mantenha o foco na qualidade, (PRESSMAN, 2005) comenta que a atividade metodológica por trás da engenharia de software consiste em um conjunto de ações, cada uma delas composta por tarefas que devem ser executadas e concluídas. Essas tarefas estão descritas na Tabela 1.

Tabela 1 – Ações genéricas em metodologias de engenharia de software

Atividade	**Função**
Comunicação	Etapa Inicial, entender os objetivos e os requisitos do software, conversando com os envolvidos.
Planejamento	Definição dos escopos, prazos, recursos e as estratégias envolvidas no projeto.
Modelagem	Criar representações visuais e estruturais do software (como diagramas e "esboços") para guias na construção.
Construção	Implementar o software com base nos requisitos e nos modelos e estratégias definidos.
Entrega	Liberar o software ou feature desenvolvida, garantindo as expectativas e os requisitos definidos

Os modelos de softwares, que implementam essas tarefas base para o desenvolvimento de aplicações e softwares, não necessariamente ficam dependentes dessas tarefas, já que a maioria dos modelos não são estáticos, e sim podem-se adaptar às situações que são apresentadas

durante o desenvolvimento e as necessidades dessas aplicações (PRESSMAN, 2005; SOMMERVILLE, 2011).

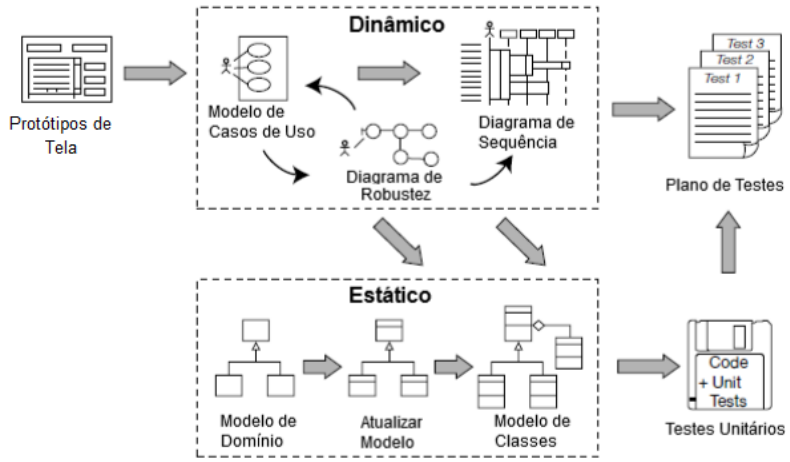
Para o desenvolvimento de um projeto, faz-se necessário uma metodologia de software que implementa essas tarefas, por isso escolheu-se a metodolgia ICONIX, elaborada por Rosenberg e Kendal Scott, a partir de um processo simples e unificado dos pesquisadores Booch, Rumbaugh e Jacobson (ROSENBERG; STEPHENS; COLLINS-COPE, 2005).

2.1 METODOLOGIA DE DESENVOLVIMENTO DE SOFTWARE

A *Unified Modeling Language* (UML) é, atualmente, a linguagem de modelagem padrão utilizada globalmente pela indústria de engenharia de software. Com tudo, a mesma possui um grande número de diagramas que dão ênfase tanto às características estruturais quanto comportamentais do software. A pessoa que for usar essa linguagem, não necessariamente precisará utilizar todos os artefatos disponíveis dela, poderá se adaptar conforme a demanda (GUEDES, 2018). Por esses motivos, outras soluções surgem, como o ICONIX por exemplo.

O ICONIX é um processo utilizado de desenvolvimento de software, emergido de uma abordagem minimalista e simplificada da UML, ficando entre os casos de uso e o código. Na prática, o tempo que se leva para fazer a modelagem e o design, nunca parece ser o suficiente. Este processo surge como solução para esse problema (ROSENBERG; SCOTT, 1999). A Figura 2, exemplifica o fluxo geral de funcionamento do processo ICONIX.

Figura 2 – Visão geral do processo ICONIX



Fonte: Adaptado de (ROSENBERG; SCOTT, 1999)

A Figura 2 apresenta uma visão geral dos artefatos e atividades do processo. As fases do processo - requisitos, análise (*design* preliminar), *design* detalhado e implementação - se relacionam com esses artefatos, conforme descrito a seguir (ROSENBERG; SCOTT, 1999):

1. **Requisitos:**

- Definir os requisitos funcionais com base no que o sistema será capaz de fazer.
- Definir a modelagem de domínio, entendendo o problema.
- Definir requisitos de comportamento, entendendo como o usuário irá interagir.
- Validar se os casos de uso condizem com as expectativas dos clientes.

2. **Análise (design preliminar):**

   - Desenhar o diagrama de robustez, uma interpretação dos objetos de casos de uso, redesenhando-o caso necessário.
   - Atualizar a modelagem de domínio enquanto desenvolve os casos de uso e desenha o diagrama de robustez. Possivelmente descobrirá ambiguidades, classes e atributos de domínio faltantes.
   - Nomear todas as funções lógicas do software necessárias para os casos de uso funcionar. Caso seja necessário, atualize os casos de uso.

3. **Design detalhado:**

   - Desenhar o diagrama de sequência, uma representação dos detalhes dos casos de uso. Desenha-se um diagrama de sequência para cada caso de uso.
   - Atualizar o modelo de domínio enquanto desenha o diagrama de sequência, adicionando operações para os objetos de domínio.

4. **Implementação:**

   - Escrever código e seus testes unitários. O oposto poderá ser feito também.
   - Realize testes com base nos casos de uso, testando tanto o cenário base quanto os alternativos.
   - Realize uma revisão de código e de modelo, para preparar o próximo passo do trabalho.

No âmbito do desenvolvimento de software, os sistemas de controle de versão desempenham um papel fundamental ao proporcionar a continuidade e a rastreabilidade dos ciclos incrementais e iterativos. O controle de versão é uma estrutura onde os registros de mudanças de arquivos ficam gravadas podendo no futuro visualizar essas mudanças. Para o desenvolvimento de software, torna-se uma ferramenta indispensável, pois permite que arquivos sejam revertidos para um estado anterior, seja feita comparações ao longo do tempo, ver quem modificou um arquivo pela última vez (útil nos casos de algum problema estar acontecendo) e reverter o projeto como um todo para o estado anterior (CHACON; STRAUB, 2014).

Existem alguns tipos de sistemas para o controle de versionamento, mas o mais conhecido é o Sistema de Controle de Versão Distribuído, como por exemplo, o *GIT*. Nesse tipo de

sistema, os clientes não apenas verificam a última versão dos arquivos, mas espelham todo o repositório, permitindo que qualquer cópia possa restaurar o servidor em caso de falha (CHACON; STRAUB, 2014).

Esses arquivos ficam armazenados em repositórios, onde várias pessoas possam acessar e clonar esses arquivos. A plataforma mais popular para o armazenamento desses arquivos é o *GitHub*, o maior repositório hospedado de projetos *GIT* e um dos principais centros de colaboração para milhões de desenvolvedores ao redor do mundo (CHACON; STRAUB, 2014).

2.2 PROTEÇÃO DOS DADOS

O uso de sistemas de informação para integração dados por meio de redes tem se tornado cada vez mais difundido. No entanto, esse ambiente está sujeito a diversas ameaças físicas e virtuais, que comprometem a segurança dos usuários e de suas informações. Um exemplo é a exposição de dados sensíveis a terceiros, colocando em risco a privacidade e a confidencialidade. É relevante adotar práticas de segurança eficientes para mitigar esses riscos, protegendo tanto a integridade dos sistemas quanto a privacidade dos usuários (MARCIANO; LIMA-MARQUES, 2006).

Dado o problema apresentado, as práticas e abordagens de seguranças devem estar alinhadas com a regulamentação de proteção de dados. As mesmas possuem um impacto direto de como o desenvolvimento e as operações de software são conduzidas. No Brasil, é a LGPD que possui a responsabilidade desse papel, a *General Data Protection Regulation* (GDPR) na União Europeia e a *Health Insurance Portability and Accountability Act* (HIPAA), nos Estados Unidos. Tanto a GDPR quanto a LGPD apresentam desafios significativos em sua implementação, pois foram redigidas em linguagem jurídica – e não técnica –, o que pode dificultar sua aplicação prática por profissionais de tecnologia e segurança da informação (FREITAS *et al.*, 2025).

2.2.1 GDPR

A GDPR, vigente desde 25/05/2018, é uma legislação que regula o processamento e armazenamento de dados de cidadãos da União Europeia, fortalecendo a proteção da privacidade perante aos desafios digitais. Seu alcance é global, impactando qualquer organização que opere no mercado europeu ou trate dados pessoais de residentes da União Europeia. A norma confere aos usuários direitos como consentimento revogável (Art. 7) e direito ao esquecimento (Art. 17), enquanto exige dos controladores e processadores de dados medidas como proteção desde a concepção (Art. 25) e o registro de todas as atividades processadas (Art. 30). A GDPR diz que, as organizações devem obter consentimento explícito e adotar medidas técnicas robustas para proteger estes dados, ou seja, para alcançar o chamado *compliance*, caso contrário, essas organizações serão responsabilizadas pelo não comprimento da GDPR (LI; YU; HE, 2019).

> *Compliance* refere-se ao conjunto de práticas e processos destinados a assegurar o cumprimento de leis, regulamentos, políticas internas e normas institucionais. Seu objetivo é prevenir, identificar e corrigir possíveis desvios ou não conformidades dentro da organização, garantindo conformidade legal e operacional (SILVA; COVAC, 2018).

A GDPR responsabiliza organizações que processam dados de residentes da União Europeia, exigindo conformidade com regras rigorosas de segurança e privacidade. Empresas de tecnologia, provedores de nuvem e *marketers* devem adotar medidas mais rigorosas para proteger dados pessoais — como nomes, *e-mails*, IPs e informações genéticas — sob risco de multas elevadas. Empresas como *Google* e *Facebook* já adaptaram suas políticas para o *compliance* dessa legislação. A conformidade com a GDPR não evita apenas penalidades, mas também oferece uma vantagem competitiva contra os concorrentes que não se adequaram a ela (SILVA; COVAC, 2018).

2.2.2 HIPAA

A HIPAA é uma lei federal que tem como principal objetivo regular o uso e a divulgação de informações de saúde dos pacientes nos Estados Unidos. Aprovada em 1996, embora sua proposta inicial não estivesse diretamente focada em questões de privacidade, o avanço tecnológico no setor de saúde evidenciou a necessidade de proteger os dados médicos sensíveis. Diante desse cenário, os engenheiros de *software* tornaram-se peças fundamentais no desenvolvimento de aplicativos e sistemas de saúde, garantindo que essas soluções estejam em plena conformidade com os requisitos da HIPAA (ELKOURDI *et al.*, 2024).

A regulamentação exige que os softwares que lidam com informações de saúde dos pacientes, implementem controles de seguranças apropriados, como controle de acesso, autenticação e criptografia. Além desses controles de segurança, a HIPAA exige que os engenheiros de *softwares* responsáveis por esses softwares implementem técnicas específicas para preservar as informações, como *backup* de dados e procedimentos para recuperação dos mesmos (ELKOURDI *et al.*, 2024).

2.2.3 LGPD

Aprovada em 14 de agosto de 2018, a LGPD entrou em vigor em setembro de 2020, inspirada na norma europeia de proteção de dados, a GDPR, previamente apresentada (BNDES, 2025). A legislação da LGPD visa garantir os direitos básicos à privacidade, liberdade e desenvolvimento pessoal dos cidadãos brasileiros. Ela regulamenta o manuseio de dados pessoais – tanto em formatos físicos quanto digitais – por empresas ou órgãos públicos. As normas abrangem diversas operações de processamento, independentemente do meio utilizado, como por exemplo, manual ou eletrônico. Seu escopo é amplo, aplicando-se a qualquer entidade que realize tratamento dessas informações (SOCIAL, 2025).

A LGPD classifica em categorias específicas os dados pessoais, como seu nível de sensibilidade e aplicabilidade. Entender essas diferenças é crucial para organizações e indivíduos, com o objetivo de assegurar que, essas informações sejam tratadas de forma ética e justa. A seguir, serão apresentadas essas categorias:

a) **Dados Pessoais**: Informações que identificam uma pessoa, diretamente ou indiretamente, como por exemplo, CPF, nome , e-mail.

b) **Dados Sensíveis**: Uma subcategoria dos dados pessoais. São características íntimas, como por exemplo, origem racial, dados da saúde, orientação sexual e questões genéticas.

c) **Dados Públicos**: Informações acessíveis pelo próprio titular - redes sociais por exemplo - ou por interesse público. Podem ser usados sem um novo consentimento desde que respeitem a Lei de Acesso à Informação (LAI).

d) **Dados Anonimizados**: Informações que não permitem a identificação do titular, mesma utilizando técnicas avançadas. Não se aplicam diretamente a LGPD, porém se forem pseudonimizados, voltam a ser regulados pela lei.

Para a LGPD, existem diferentes entidades que podem exercer papeis diferentes na adaquação á proteção de dados pessoais, as mesmas estão descritas a baixo:

a) **Controlador**: Pessoa física ou jurídica (pública ou privada) que toma as decisões sobre como os dados pessoais serão tratados, definindo finalidades e métodos.

b) **Operador**: Pessoa física ou jurídica que processa os dados em nome do controlador, seguindo suas instruções, como empresas terceirizadas.

c) **Encarregado**: Representante indicado pelo controlador/operador para intermediar a comunicação entre a organização, os titulares dos dados e a ANPD, além de supervisionar o cumprimento da LGPD.

d) **ANPD**: Órgão público responsável por fiscalizar, orientar e aplicar a LGPD em todo o território nacional.

e) **Titular**: Pessoa física a quem pertencem os dados pessoais em tratamento, com direitos garantidos pela lei.

A LGPD estabelece 10 princípios fundamentais que devem guiar o tratamento de dados pessoais, garantindo transparência, segurança e respeito aos direitos dos titulares (BRASIL, 2018):

a. **Finalidade**: Os dados só podem ser usados para objetivos legítimos, específicos e informados ao titular, sem desvios posteriores.

b. **Adequação**: O tratamento deve ser compatível com as finalidades declaradas, considerando o contexto em que os dados são coletados.

c. **Necessidade**: Coletar apenas dados estritamente necessários para a finalidade, evitando excessos.

d. **Livre Acesso**: O titular tem direito a consultar, de forma fácil e gratuita, como seus dados são tratados e por quanto tempo.

e. **Qualidade dos Dados**: Garantir que as informações sejam precisas, claras e atualizadas, conforme a necessidade do tratamento.

f. **Transparência**: Disponibilizar informações claras sobre o tratamento, identificando os responsáveis.

g. **Segurança**: Adotar medidas técnicas e administrativas para proteger os dados contra vazamentos, acessos não autorizados ou perdas acidentais.

h. **Prevenção**: Agir proativamente para evitar danos aos titulares durante o tratamento.

i. **Não Discriminação**: Proíbe o uso de dados para fins discriminatórios ou abusivos.

j. **Responsabilização**: O controlador/operador deve comprovar que cumpre a LGPD, adotando medidas eficazes e documentando suas ações (*accountability*).

Com base nas legislações apresentadas anteriormente, elaborou-se a Tabela 2, que apresenta um comparativo entre os principais critérios abordados em cada uma delas.

2.3 ARQUITETURA DE SISTEMAS DISTRIBUÍDOS

Um sistema distribuído pode ser definido quando componentes em máquinas conectadas em rede interagem exclusivamente através da passagem de mensagens. Com tudo, essa abordagem possui alguns desafios, como a heterogeneidade dos componentes, segurança, ser um sistema aberto - o que permite que componentes adicionais sejam adicionados ou modificados com facilidade - e o fornecimento de um serviço de qualidade, sendo escalável (COULOURIS *et al.*, 2013).

2.3.1 ARQUITETURA EM CAMADAS

Esta arquitetura se baseia na decomposição do software em componentes modulares, uma abordagem amplamente adotada por engenheiros de software. O modelo pode ser comparado a uma estrutura em camadas, similar a um bolo, onde cada nível superior se apoia sobre

Tabela 2 – Comparativo entre GDPR, HIPAA e LGPD

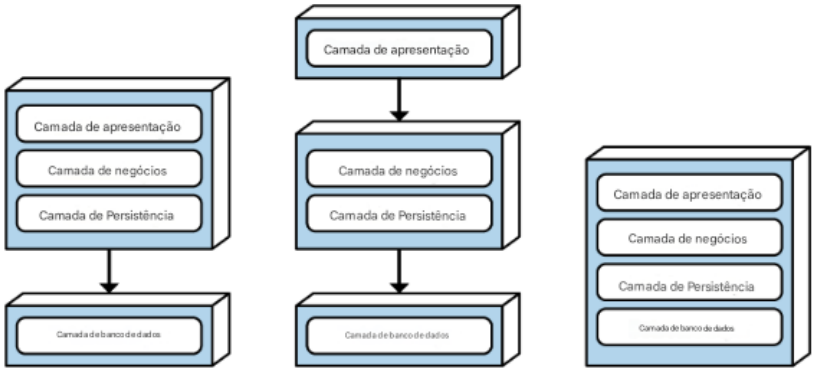
Critério	GDPR	HIPAA	LGPD
Origem	União Europeia, 2018	Estados Unidos, 1996	Brasil, 2018 (vigente desde 2020)
Abrangência	Dados pessoais de residentes da UE, mesmo tratados fora da UE	Informações de saúde protegidas (PHI) nos EUA	Dados pessoais tratados no território nacional ou com efeitos no Brasil
Foco	Proteção geral de dados pessoais	Privacidade e segurança de informações de saúde	Privacidade, proteção e uso adequado de dados pessoais
Exemplos de dados	Nome, e-mail, IP, dados genéticos, localização	Histórico médico, diagnósticos, resultados de exames, prontuários	CPF, nome, e-mail, dados de saúde, origem racial, orientação sexual
Direitos dos titulares	Acesso, retificação, exclusão (direito ao esquecimento), portabilidade, oposição, revogação do consentimento	Acesso e controle sobre seus próprios dados de saúde	Confirmação, acesso, correção, anonimização, portabilidade, exclusão, revogação do consentimento
Exigências técnicas	Consentimento explícito, registro de atividades, proteção desde a concepção, criptografia, anonimização	Controle de acesso, autenticação, criptografia, backup, plano de recuperação	Medidas técnicas e administrativas de segurança, prevenção contra vazamentos, acesso não autorizado ou perda

as camadas inferiores. Neste padrão arquitetural, as camadas superiores utilizam serviços das camadas subjacentes, enquanto estas últimas permanecem independentes e desconhecem a existência das camadas superiores (FOWLER, 2009).

A arquitetura em camadas oferece três benefícios principais, a abstração, permitindo a construção de software utilizando camadas sem a necessidade de conhecer seus detalhes internos de implementação, a independência, que reduz ao mínimo as dependências entre as diferentes camadas e a reutilização, possibilitando que uma camada já implementada seja aproveitada por diversos serviços distintos. Essas características combinadas promovem a criação de sistemas mais modulares, flexíveis e eficientes (FOWLER, 2009). A quantidade de camadas, no entanto, não é fixa, variando conforme as necessidades de cada aplicação (RICHARDS; FORD, 2020).

Dentre a organização frequentemente utilizada, a divisão é feita em três níveis lógicos, a de apresentação, domínio e fonte de dados, cada uma com sua regra e responsabilidades definidas. A Tabela 3 representa três formas diferentes de organizar as três camadas:

Figura 3 – Exemplos de aplicação da arquiteturas em camadas



Fonte: Adaptado de (RICHARDS; FORD, 2020)

Tabela 3 – Responsabilidades de cada camada na arquitetura

Camada	**Responsabilidades**
Apresentação	Fornecimento de serviços, exibição de informações (p. ex., em Windows ou HTML, tratamento de solicitações do usuário (cliques com o mouse, pressionamento de teclas) requisições HTTP, chamadas em linha de comando, API em lotes)
Domínio	Lógica que é o real propósito do sistema
Fonte de dados	Comunicação com o banco de dados, sistema de mensagens, gerenciadores de transações, outros pacotes

Fonte: (FOWLER, 2009)

2.3.2 PADRÕES DE PROJETO

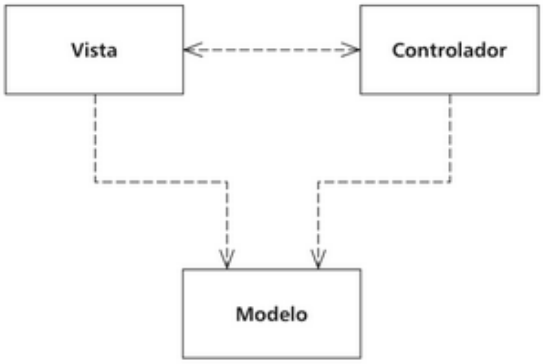
Na computação, um padrão descreve tanto um problema frequente em determinado contexto quanto sua solução essencial, que pode ser adaptada e reutilizada de múltiplas formas, sem exigir implementação idêntica em cada aplicação (ALEXANDER, 1977). Conforme discutido na Seção 2.3.1, a arquitetura em camadas promove a separação de responsabilidades, mas requer mecanismos eficientes de integração entre seus níveis para preservar seus benefícios de modularidade. Neste contexto, os padrões arquiteturais surgem como solução potencial, embora sua eficácia dependa da aplicação rigorosa de boas práticas de desenvolvimento - fator crítico para a construção de sistemas complexos e escaláveis (ALUR; CRUPI; MALKS, 2003).

O restante deste capítulo apresenta em detalhes os padrões de projeto relevantes para este trabalho, explorando especificamente aqueles que estabelecem relações diretas com os objetivos e abordagens propostos neste trabalho.

2.3.2.1 MVC

O padrão *Model View Controller* (MVC), é um dos padrões mais mencionados e usados no desenvolvimento de aplicações. Ele considera três papéis, como apresentado na Figura 4.

Figura 4 – Divisão da interação da interface com o usuário em três papeis distintos



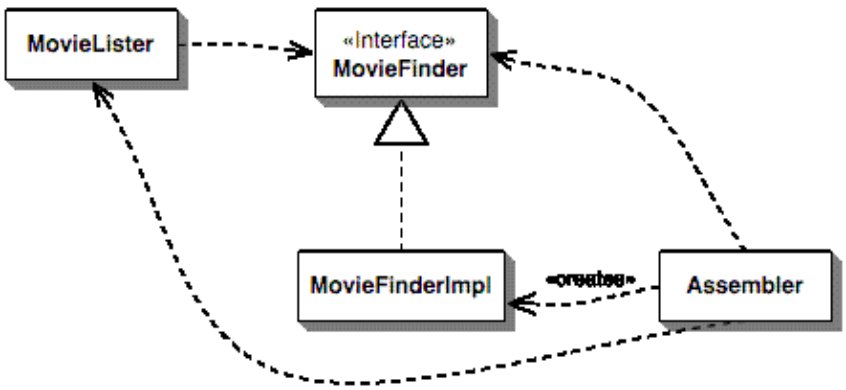
Fonte: (FOWLER, 2009)

O modelo corresponde a um objeto que encapsula dados e regras do domínio da aplicação, enquanto a vista é responsável por renderizar esses dados na interface do usuário. Por sua vez, o controlador atua como intermediário, processando as interações do usuário, modificando o modelo conforme necessário e garantindo que a visão seja atualizada de forma consistente (FOWLER, 2009).

2.3.2.2 INJEÇÃO DE DEPEDÊNCIA

Este padrão arquitetural proporciona a base para que estruturas de software e *frameworks* funcionem como arcabouços extensíveis, transferindo para as aplicações que os incorporam a definição dos aspectos específicos de implementação. Tal abordagem favorece tanto a reutilização de componentes quanto a redução de dependências entre módulos, conforme destacado por (JOHNSON; FOOTE, 1988).

Figura 5 – Dependências para um Injetor de Dependências



Fonte: (FOWLER, 2004)

A Figura 5 ilustra o princípio da injeção de dependência, em que um objeto externo —

o *montador* (*assembler*) — é responsável por fornecer à classe *Lister* uma implementação concreta da interface que ela necessita, reduzindo assim o acoplamento entre as classes (FOWLER, 2004).

2.3.2.3 REPOSITORY PATTERN

O *Repository Pattern* é um padrão de projeto que encapsula a lógica de acesso aos dados, isolando a camada de persistência da camada de domínio e atuando como um mediador entre ambas. Ele oferece uma interface simples e consistente para consulta e manipulação de objetos de domínio, permitindo que o código da aplicação permaneça focado nas regras de negócio. Segundo Fowler (2009), o repositório abstrai a complexidade de acesso ao banco de dados e pode operar com diferentes fontes de dados, inclusive em memória, o que facilita a execução de testes e melhora o desempenho. Além disso, o uso de injeção de dependência nesse padrão promove maior testabilidade, manutenção e flexibilidade, possibilitando a substituição do banco de dados ou da tecnologia de persistência sem impactar o restante do sistema. Com isso, o *Repository Pattern* contribui para um código mais limpo, desacoplado e de fácil manutenção e evolução (FOWLER, 2009).

2.3.3 COMUNICAÇÃO

A comunicação é essencial para os sistemas distribuídos, pois permite que processos em máquinas distintas troquem dados e atuem como um todo. Para possibilitar o desenvolvimento distribuído em grande escala, há diversas maneiras de realizar essa comunicação, cada uma com suas particularidades e abordagens (STEEN; TANENBAUM, 2023).

Quando dois ou mais processos necessitam se comunicar, um envia uma mensagem ao outro, e essa comunicação pode ocorrer de forma síncrona ou assíncrona. O primeiro aspecto a ser considerado nesse cenário é a persistência da mensagem. Na comunicação assíncrona, a mensagem é persistida, permitindo que os processos envolvidos a executem em momentos distintos. Dessa forma, o processo remetente pode prosseguir com suas tarefas sem interrupção. Já na comunicação síncrona, a mensagem exige execução imediata por ambos os processos, bloqueando o remetente até que a operação seja concluída. Há três situações em que o remetente pode ser liberado na comunicação síncrona (STEEN; TANENBAUM, 2023):

- Notificação de início da transmissão da mensagem;
- Confirmação de entrega da mensagem ao destinatário;
- Resposta direta do destinatário após o processamento;

Dentre alguns modelos que existem para a comunicação entre sistemas distribuídos, o autor destaca dois, sendo eles os modelos de *publish-subscribe* (publicador-assinante) e *request-reply* (requisição-resposta).

2.3.3.1 PUBLISH-SUBSCRIBE

Este modelo de comunicação é assíncrono, baseada em eventos, onde nele, várias filas podem ser criadas, e as aplicações que querem se comunicar devem assinar (*subscribe*) aquelas que possuem interesse. Para estabelecer a comunicação, as aplicações publicam mensagens (*publish*) nas filas criadas. Quando o processamento dentro dessas filas é concluído, as mensagens são automaticamente direcionadas às aplicações assinantes correspondentes (STEEN; TANENBAUM, 2023). Um exemplo que o autor comenta para este modelo de comunicação é o *Message-Oriented Middleware* (MOM).

Outro exemplo para este modelo são os *webhooks*, utilizando protocolo *HyperText Transfer Protocol* (HTTP). Eles podem ser vistos como uma forma especializada de manipuladores de eventos, sendo ativados automaticamente quando um evento ocorre, sem a necessidade contínua de verificação do cliente. Além disso, funcionam de forma semelhante as chamadas funções de *callbacks*, ou funções de retorno, com a exceção de que estes *webhooks* estão hospedados em outros lugares, e não na mesma aplicação. As responsabilidades envolvidas neste método são divididas entre os clientes, que implementam e registram o *webhook* de fato, e os provedores, que definem eventos, gerenciando as assinaturas e entregando os dados quando os eventos ocorrem (BIEHL, 2017).

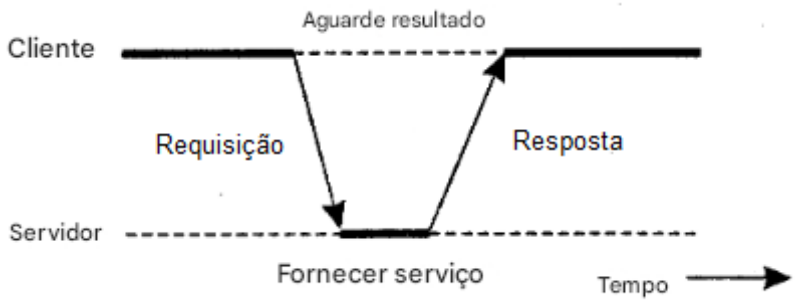
2.3.3.2 REQUEST-REPLY

Este modelo adota a arquitetura cliente-servidor, um paradigma de comunicação distribuída em que as entidades assumem papéis distintos: o cliente, que solicita serviços, e o servidor, que os disponibiliza. A interação ocorre com o cliente enviando uma solicitação (*request*) ao servidor, que por sua vez processa a demanda e retorna uma resposta (*reply*). Este modelo é síncrono, ou seja, o cliente permanece em estado de espera até a recepção da resposta do servidor. A dinâmica dessa comunicação é ilustrada na Figura 6 (STEEN; TANENBAUM, 2023).

Um protocolo que utiliza esse modelo de requisição resposta é o HTTP. Esse protocolo utiliza um modelo de arquitetura chamado *Representational State Transfer* (REST), no qual visa estabelecer uma interface padronizada para o fluxo de dados, com o objetivo de melhorar a escalabilidade e a precisão no tratamento de requisições. Dessa forma, cabe ao cliente incluir todos os dados necessários em cada requisição, permitindo que o servidor processe a solicitação de forma independente (FIELDING, 2000).

As APIs REST constituem uma implementação prática desse modelo de comunicação, alinhada aos princípios do protocolo REST. Funcionando como um tipo específico de *web ser-*

Figura 6 – Interação geral entre um cliente e um servidor



Fonte: Adaptado de (STEEN; TANENBAUM, 2023)

vice — isto é, um serviço disponibilizado na web para atender demandas de sistemas clientes —, elas permitem que aplicações interajam com sistemas externos, acessando dados e serviços de maneira estruturada (MASSE, 2011).

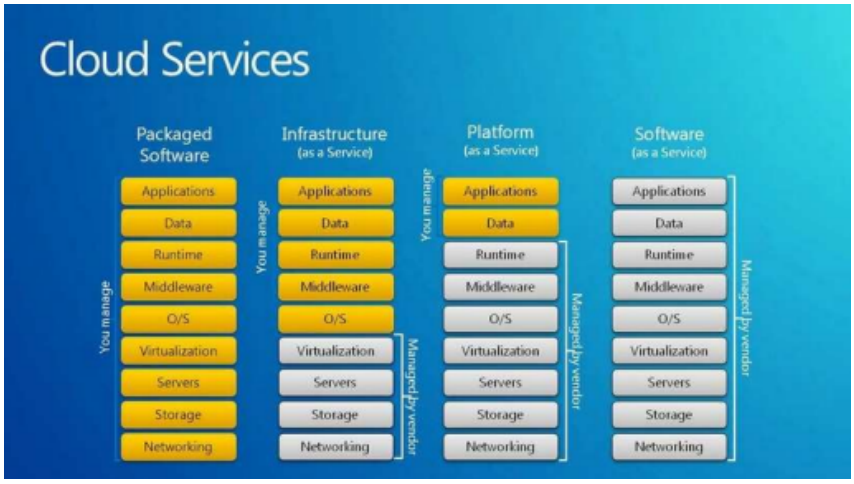
Esse padrão de APIs é discutido por (TORAB-MIANDOAB *et al.*, 2023), que destaca a adoção do *Health Level Seven International* (HL7) *Fast Healthcare Interoperability Resources* (FHIR) como padrão internacional para interoperabilidade em saúde. Desenvolvido pela organização HL7, o FHIR representa um modelo avançado de interoperabilidade que utiliza REST APIs como arquitetura, para padronizar e facilitar a troca de dados na área da saúde.

2.4 COMPUTAÇÃO EM NUVEM

A computação em nuvem consiste em um paradigma que oferece acesso onipresente, prático e sob demanda à um conjunto compartilhado de recursos computacionais configuráveis — incluindo redes, servidores, armazenamento, softwares e serviços — os quais podem ser rapidamente alocados ou liberados, exigindo mínimo esforço de administração ou interação com o provedor. Esse modelo é caracterizado por cinco propriedades essenciais (autoatendimento sob demanda, amplo acesso via rede, compartilhamento de recursos, rápida elasticidade e serviço mensurável), três modelos de serviço (*Software as a Service* (SaaS), *Platform as a Service* (PaaS) e *Infrastructure as a Service* (IaaS)) e quatro formas de implantação (nuvem privada, comunitária, pública e híbrida) (SATYANARAYANA, 2012). Diante dessas características, para este trabalho, a implementação inicial será desenvolvida no modelo SaaS, aproveitando os benefícios da computação em nuvem para oferecer à aplicação acessos escaláveis, acessíveis e de fácil manutenção.

O SaaS é um modelo em que aplicativos são disponibilizados online por um provedor, sendo acessados pelos usuários via *Web*, normalmente por meio de uma assinatura. Esse sistema remove a obrigação de instalar, manter e administrar o software localmente, já que as atualizações e otimizações ficam a cargo do fornecedor. Entre suas principais características estão o acesso por navegador, pouca customização, infraestrutura compartilhada e pagamento

Figura 7 – Modelos de serviços na computação em nuvem



Fonte: (SATYANARAYANA, 2012)

periódico baseado no uso. O SaaS traz uma transformação significativa no setor de software, transferindo a ênfase do produto para o serviço e alinhando o sucesso do fornecedor à satisfação e fidelização do cliente (SATYANARAYANA, 2012).

2.5 TECNOLOGIAS E FRAMEWORKS

Profissionais de desenvolvimento de software utilizam diversas ferramentas em seu cotidiano, as quais são essenciais para edição de documentos e gestão do extenso volume de dados produzidos em projetos complexos (SOMMERVILLE, 2011). Nesta seção, apresenta-se o conjunto de ferramentas e tecnologias que irão ser usadas no decorrer deste trabalho.

2.5.1 LINGUAGEM DE PROGRAMAÇÃO

Uma linguagem de programação, é uma forma padronizada que desenvolvedores utilizam para expressar instruções de um programa para um computador ou máquina, seguindo uma estrutura de regras sintáticas - que dizem respeito a forma como é realizada a escrita - e as semânticas - o conteúdo das instruções. Com ela, pode-se dizer quais dados vão ser trabalhados, quais as tratativas estes dados vão receber, quais ações irão ser tomadas dependendo do contexto, entre outras. Vale ressaltar que cada linguagem de programação possui o seu conjunto de regras para a escrita, mas todas dependem da mesma coisa para funcionar, a lógica de programação, uma habilidade para organizar comandos e informações de forma estruturada para resolver algum problema seguindo uma regra de negócio (GOTARDO, 2015).

2.5.1.1 JAVASCRIPT

O *JavaScript* é uma linguagem desenvolvida para aplicações *web* com o intuito de adicionar dinamismo nas páginas, sendo amplamente utilizada por sites e navegadores. Ela é uma

linguagem de alto nível, interpretada, orientada a objetos funcionais e derivada na linguagem Java, que, em meados da década de 1990, possuía a atenção dos desenvolvedores. Esta linguagem surgiu inicialmente para funcionar apenas no lado do cliente, neste caso, os navegadores, mas depois, a linguagem foi se desacoplando deles para ser usada no lado do servidor. Para essa finalidade, criaram-se alguns interpretadores da linguagem, entre eles o *Node* uma versão do interpretador *V8* da *Google* utilizada no navegador *Google Chrome* (FLANAGAN, 2012).

O *Node* surgiu para solucionar problemas de linguagens como *PHP*, *.NET* e *C#* que paralisavam um processamento no servidor, tecnologia conhecida como *Blocking-Thread*, consumindo uma maior quantidade de recursos de hardware destes servidores, e causando gargalos. A tecnologia é *open source* e utiliza o paradigma de *single-thread*, ou seja, uma única *thread* por processo, evitando *dead-locks*. Para suportar alta carga de requisições e processamento, a tecnologia conta uma arquitetura assíncrona, onde impede o bloqueio desta *thread* única, esperando a resposta da operação quando a mesma estiver pronta, evitando assim, desperdiçar ciclos do processador, tecnologia conhecida como *non-blocking thread*. Isso faz com que o *Node* consiga lidar com uma grande quantidade de conexões simultâneas em apenas um servidor, sem precisar aplicar a estrutura de múltiplas *threads* e seu gerenciamento (PEREIRA, 2014; OpenJS Foudation, 2025a).

Além dos pontos citados a cima, o *Node* possui um gerenciador de pacotes nativo, chamado *npm*. Com ele, pode-se gerenciar as dependências do projeto, isto é, trechos de código pré compilados como pacotes e bibliotecas necessárias para o projeto funcionar. O *npm* inicialmente foi criado para o *Node*, mas já usado como gerenciador para dependências no *JavaScript* do lado do cliente também (OpenJS Foudation, 2025b).

2.5.2 ARMAZENAMENTO

O armazenamento de dados em uma aplicação de software é realizado por meio de um banco de dados, que, segundo (DATE, 2004), pode ser comparado a um sistema de arquivamento digital, organizando conjuntos de informações de forma estruturada. Esses bancos de dados são gerenciados por um Sistema de Gerenciamento de Banco de Dados (SGBD), responsável pelo controle, acesso e manipulação dos registros. Os bancos de dados podem ser classificados em duas categorias principais: os relacionais e os não-relacionais.

Um banco de dados relacional consiste em um conjunto de tabelas inter-relacionadas, cada uma composta por uma estrutura definida. Nessa estrutura, destacam-se duas partes fundamentais: o cabeçalho, que contém a definição das colunas com seus respectivos nomes e tipos de dados; e o corpo, formado por linhas que seguem rigorosamente a estrutura definida no cabeçalho, armazenando os valores dos dados. Sistemas baseados em bancos de dados relacionais geralmente adotam a linguagem *Structured Query Language* (SQL) como padrão para definição, manipulação e controle de dados em um SGBD relacional (DATE, 2004).

Em contraste com os bancos de dados relacionais, os sistemas não relacionais utilizam estruturas de dados alternativas às tabelas tradicionais. Essa diferença fundamental exige operadores específicos para manipulação e consulta dos dados, adaptados a cada modelo particular (DATE, 2004). O termo *NoSQL* apareceu e 1998 para referenciar bancos de dados relacionais que omitiam o uso do SQL, porém esse termo ganhou força em 2009 para se referenciar a banco de dados não relacionais. Eles surgiram com o objetivo de resolver alguns problemas que os modelos relacionais traziam, como por exemplo, o excesso de complexidade necessária em alguns casos de armazenamento e aumentar o rendimento, a fim de diminuir o tempo de espera das consultas (STRAUCH; SITES; KRIHA, 2011).

Os bancos de dados não relacionais podem ser divididos em algumas classes ou categorias, dependo do tipo de estrutura de dados que armazenam, como por exemplo, chaves/valores, documentos e orientados a colunas. Os modelos que armazenam documentos são considerados uma evolução do modelos de chaves/valores, pois conseguem armazenar estruturas mais complexas, encapsulando pares de chaves e valores em documentos. Um dos sistemas que mais apresentam essa classe de armazenamento de dados é o *MongoDB*, amplamente utilizado em sistemas (STRAUCH; SITES; KRIHA, 2011).

O *MongoDB* é um projeto de código aberto desenvolvido em *C++*, no qual possui como principal objetivo fechar a lacuna da velocidade e escalabilidade de banco de dados relacionais. Nele, armazenamos os documentos nas chamadas *collections*, um agrupamento de documentos. Estes documentos pode ser de diferentes estruturas, como comentado anteriormente, e as linguagens mais usuais para representá-los são são *Extensible Markup Language* (XML) e *JavaScript Object Notation* (JSON) (STRAUCH; SITES; KRIHA, 2011).

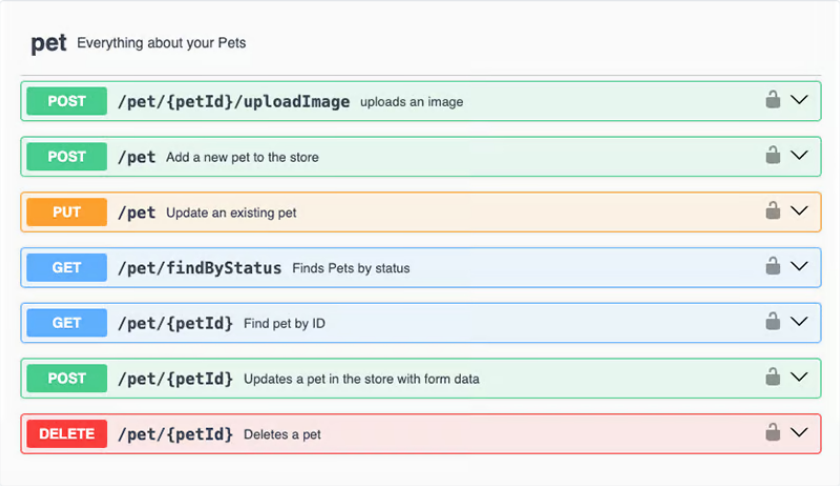
2.5.3 DOCUMENTAÇÃO

A documentação faz parte do software, sendo ela, uma diferença importante do software profissional do amador. A documentação pode ser tanto a nível de sistema, descrevendo a sua estrutura, ou a nível de usuário, explicando como usar de fato o sistema (SOMMERVILLE, 2011).

A produção e utilização de APIs Web enfrentam desafios significativos, particularmente devido à ausência de padrões universalmente adotados para especificação e documentação dessas interfaces. Essa lacuna frequentemente resulta em inconsistências na implementação e dificuldades de integração por parte dos consumidores das APIs. A partir desse cenário, um *framework* que vem se consolidado e possui a capacidade de resolver estes problemas chama-se *Swagger* (CASAS *et al.*, 2021).

O *Swagger* é uma das possibilidades de *frameworks* utilizados para a documentação de APIs Web. Surgido em 2011, sua especificação foi renomeada para *OpenAPI* em 2016, quando a empresa *SmartBear* a doou à *OpenAPI Initiative*. Por isso, os termos *Swagger* e *OpenAPI* são frequentemente usados como sinônimos. Essa especificação oferece um padrão aberto, portátil

Figura 8 – Exemplo de documentação via Swagger



Fonte: (SmartBear Software, 2025)

e independente de linguagem para descrever recursos e operações de APIs REST em JSON ou *YAML Ain't Markup Language* (YAML), permitindo que consumidores compreendam as funcionalidades do serviço sem necessidade de acessar o código-fonte ou analisar a rede. Atualmente, a especificação *OpenAPI/Swagger* tornou-se a principal referência para documentação de APIs REST (CASAS *et al.*, 2021). A Figura Figura 8 ilustra a interface de documentação da API conforme é disponibilizada aos usuários finais.

2.5.4 CONTEINERIZAÇÃO

A conteinerização é uma forma de se utilizar um mesmo hospedeiro físico por dois ou mais usuário ou aplicações, por meio da virtualização de recursos a nível de sistema. No âmbito do desenvolvimento de software, é comum existirem dificuldades como por exemplo, configurar ambientes de desenvolvimento em diferentes máquinas, com diferentes tecnologias. A conteinerização por meio dessa virtualização, simplifica este e outros processos, otimizando tempo e recursos computacionais. A principal ferramenta que utiliza esse princípio é o *Docker*, vem ganhando cada vez mais notoriedade (DRUMMOND, 2017).

O *Docker*, criado pela *Docker Inc.*, é uma plataforma de código aberto que facilita a criação, o envio e o funcionamento de aplicativos em contêineres. Essa plataforma inclui o *Docker Engine*, que cria e roda as imagens dos contêineres; os *registries*, que guardam e distribuem essas imagens (como o *Docker Hub*, um serviço na nuvem); e o *Swarm*, uma ferramenta que ajuda a organizar e controlar grupos de contêineres (Docker Inc, 2025). Grande parte da popularização da conteinerização se deve ao *Docker*.

3 TRABALHOS CORRELATOS

Esta seção do trabalho tem como objetivo apresentar soluções e trabalhos propostos por outros autores, relacionados à integração de dados entre sistemas, não necessariamente sendo dados da saúde. Embora possuam objetivos distintos em relação a este trabalho, eles forneceram resultados e metodologias para a resolução de problemas apresentados pelo presente trabalho.

Para a pesquisa, utilizou-se principalmente o Google Acadêmico com os termos-chave: "Interoperabilidade entre dados da saúde", "Integração entre dados da saúde"e "*Information System Integration*". Aplicou-se um filtro dos últimos dez anos para garantir a atualidade dos padrões analisados, priorizando trabalhos com histórico recente de atualizações. Os artigos encontrados foram ordenados por data e selecionados pela análise de seus resumos e *abstracts*, com foco nos objetivos deste trabalho.

Foram analisados estudos acadêmicos, como o de Moreno (2016), que aborda os desafios da interoperabilidade em saúde, destacando a diversidade de padrões de dados e as preocupações com segurança e confidencialidade das informações, o que compromete a harmonização em escala global. A problemática dos múltiplos padrões é aprofundada por (SALES; PINTO, 2019), que realiza uma pesquisa exploratória sobre padrões de metadados aplicados à interoperabilidade, identificando aqueles mais apropriados para esse contexto. No campo da segurança da informação, o estudo de (TRINDADE *et al.*, 2024) explora os caminhos para assegurar à proteção dos dados de saúde. O trabalho enfatiza o papel crucial do conhecimento digital e da disponibilidade de recursos tecnológicos apropriados para que essas medidas de proteção sejam postas em prática de forma eficaz.

Outro autor que explora a relação entre os sistemas de informação e o setor da saúde é Wu e Trigo (2021), destacando características, impactos, desafios e suas consequências, além de propor soluções para que o *Electronic Health Information Systems* (EHIS) possa fornecer dados valiosos ao setor, melhorando serviços, diagnósticos, armazenamento e análise de doenças transmissíveis e não transmissíveis.

O trabalho de COELHO e PEDRON (2024) empregou uma Revisão Sistemática da Literatura para delimitar o conceito de *Personal Health Data* (PHD), identificando as principais barreiras que impactam a adoção dessa tecnologia. Trata-se de uma solução digital em saúde projetada para otimizar o gerenciamento dos dados pessoais de saúde e facilitar a comunicação entre pacientes e profissionais da área.

No contexto de tecnologias de monitoramento e apoio à adesão terapêutica, o estudo de (ROCHA *et al.*, 2017) examina a produção científica existente sobre aplicativos móveis de saúde, chamados de *mHealth*, buscando compreender seu potencial para ampliar e melhorar os cuidados em saúde.

A seguir, serão explicados os trabalhos que possuem forte relação com o presente trabalho, junto com a Tabela 4, que resume os principais pontos de cada proposta.

Tabela 4 – Tabela comparativa das soluções e trabalhos analisados

Título	**Referência**	**Resumo**	**Relação**
Interoperabilidade de sistemas de informação em saúde.	(MORENO, 2016)	Destaca pontos que dificultam a interoperabilidade entre sistemas, como múltiplos padrões e preocupações com segurança e confidencialidade.	Trata dos desafios para interoperabilidade em saúde, tema central da interface desenvolvida.
Tecnologias digitais de informação para a saúde: revisando os padrões de metadados com foco na interoperabilidade.	(SALES; PINTO, 2019)	Pesquisa exploratória que identifica e analisa padrões de metadados adequados à interoperabilidade de dados em saúde.	Complementa o estudo de Moreno ao aprofundar a questão dos padrões utilizados na interoperabilidade.
Estratégias em segurança de dados na saúde: uma revisão rápida.	(TRINDADE *et al.*, 2024)	Discute estratégias e tecnologias necessárias para garantir a proteção e confidencialidade dos dados de saúde.	Reforça a importância da segurança no contexto da interoperabilidade proposta pela interface.
Impact of information system integration on the healthcare management and medical services.	(WU; TRIGO, 2021)	Analisa impactos, desafios e benefícios dos sistemas eletrônicos de informação em saúde, propondo soluções para otimizar diagnósticos e gestão de dados clínicos.	Relaciona-se à proposta do trabalho ao discutir melhorias na integração e utilização de dados de saúde.
Adoção de sistemas de gestão de dados de saúde pessoal: uma revisão sistemática da literatura.	(COELHO; PEDRON, 2024)	Define o conceito de PHD e identifica barreiras para sua adoção no gerenciamento de dados pessoais de saúde.	Apresenta desafios semelhantes aos enfrentados pela interface proposta, voltados à gestão e integração de dados pessoais.
Uso de apps para a promoção dos cuidados à saúde.	(ROCHA *et al.*, 2017)	Examina o potencial dos aplicativos móveis de saúde para melhorar o monitoramento e adesão terapêutica.	Relaciona-se ao uso de tecnologias móveis para ampliar o acesso e o compartilhamento de informações de saúde.
HealthSync	(HealthSync, s.d.a)	Aplicativo móvel voltado ao compartilhamento de informações de saúde entre usuários e terceiros.	Software simples para compartilhamento de dados da saúde.
Health Connect	(Google, 2024)	Plataforma *Android* que permite o armazenamento e compartilhamento de dados de saúde entre aplicativos via APIs padronizadas.	APIs para sincronização de dados da saúde, porém restrita ao ecossistema *Android*.
Hapi Fhir	(Hapi FHIR, 2025)	*Framework* em Java com ferramentas para criação de ecossistemas FHIR para troca de dados da saúde.	Ferramenta para criação de ecossistema interoperável baseado em padrão global.
RNDS	(RNDS, s.d.d)	Conjunto de APIs do governo brasileiro para envio de resultados de exames do SARS-CoV-2 usando o padrão FHIR.	Interface de integração baseada em protocolo global, mas restrita a apenas exames laboratoriais até o momento.

Fonte: Fonte: O Autor (2025).

3.1 HEALTHSYNC

O *HealthSync* é uma ferramenta mobile criada com o intuito de compartilhar informações importantes sobre a saúde entre familiares, médicos e amigos (HealthSync, s.d.a). Criado por Jerry Kestenbaum, com o objetivo de diminuir a burocracia e a taxa de abandono com as pessoas mais velhas (HealthSync, s.d.b), fornecendo as seguintes possibilidades para o usuário:

- Gravar sintomas, com a possibilidade de adicionar notas e datas de cada alteração desses eventos.
- Manter uma lista de remédios prescritos.
- Possibilidade de receber avisos, por exemplo, de próximos compromissos.
- Listar condições diagnosticadas, a fim dos familiares e terceiros estarem atualizados com as situações da pessoa.

Esse aplicativo possibilita de uma forma intuitiva e prática que informações da saúde de uma pessoa possam ser compartilhadas com terceiros, através do celular. As funções não se limitam apenas às citadas acima, embora a ideia central esteja em torno das ações citadas.

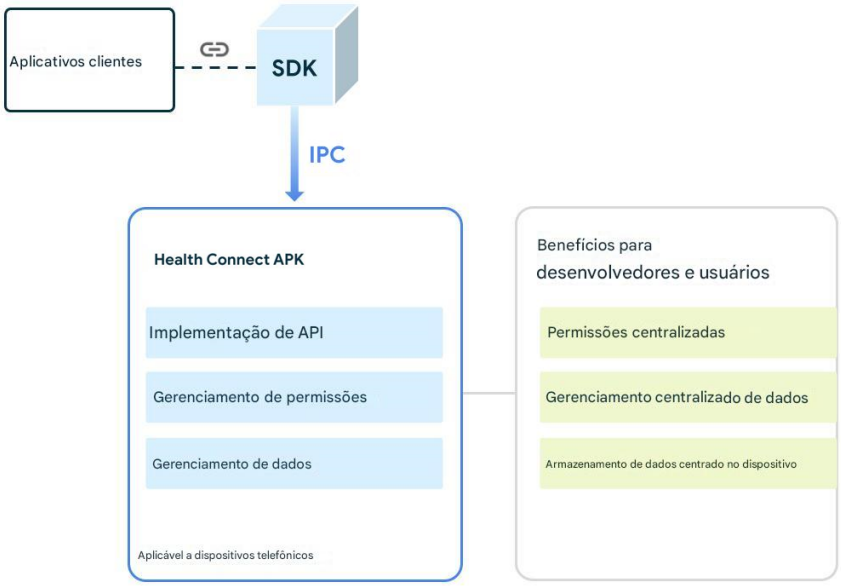
Para o contexto de engenharia, o autor dessa ferramenta não especifica quais tecnologias e arquiteturas foram utilizadas para o desenvolvimento, mas a ideia principal do aplicativo está fortemente alinhada com o presente trabalho. O principal ponto a desejar dessa ferramenta é a falta de automação: quase tudo é feito manualmente, e as informações cadastradas pelos usuários não podem ser integradas a outros sistemas, pelo menos até o momento desta pesquisa.

3.2 HEALTH CONNECT

Lançado pelo Google, o *Health Connect* ou Conexão saúde em português, é uma plataforma Android que permite armazenar *offline* e compartilhar dados da saúde, fitness e bem estar entre aplicativos. Para isso, essa solução disponibiliza um conjunto de API's para acessar os dados de um usuário, além de possuir um esquema padronizado de dados com um controle e gerenciamento de privacidade em um nível altamente detalhado, visando criar experiências melhores entre os usuários (Google, 2024).

A Figura 9 representa a arquitetura projetada para o *Health Connect*. O *Software Development Kit* (SDK) permite que aplicativos clientes se comuniquem com o *Android Application Package* (APK) da Conexão Saúde por meio de *Inter-Process Communication* (IPC), facilitando a integração com a plataforma. Os apps clientes vinculam o SDK ao aplicativo de saúde e fitness, fornecendo uma superfície de API que simplifica a interação com a API Health Connect. O APK da Conexão Saúde, disponível diretamente no dispositivo do usuário, é a base dessa

Figura 9 – Arquitetura do Health Connect



Fonte: Adaptado de (Google, 2023)

API, contendo componentes essenciais para o gerenciamento de permissões e dados (Google, 2023).

A plataforma inclui uma interface para solicitação e controle de permissões, permitindo que os usuários gerenciem o acesso aos seus dados por diferentes aplicativos. Além disso, oferece uma visão consolidada dos dados registrados, como contagem de passos, velocidade de ciclismo, frequência cardíaca e outros tipos de informações compatíveis, garantindo transparência e controle ao usuário. Essa estrutura centralizada simplifica a interoperabilidade entre apps de saúde e *fitness*, promovendo uma experiência mais segura e integrada (Google, 2025).

O ecossistema que essa solução entrega é bastante promissor, especialmente por facilitar a comunicação interna com APIs do *Android*. No entanto, essa abordagem apresenta limitações relevantes para este trabalho, pois se restringe ao ambiente *Android* e não permite integração com outros sistemas, como o ecossistema *iPhone Operating System* (IOS) da *Apple*, nem com plataformas externas, como *sites web*, para disponibilização de informações a terceiros.

3.3 HAPI FHIR

O *Hapi FHIR* é um *framework open source* completo desenvolvido em Java do padrão HL7 FHIR, previamente apresentado neste trabalho. O software permite a criação, armazenamento, recuperação e troca de dados da saúde em um formato padronizado (Hapi FHIR, 2025). Desenvolvido por uma organização chamada *Smile Digital Health*, focada em soluções para a área da saúde, com o objetivo de reduzir as barreiras entre informações e cuidados médicos, utilizando padrões abertos e uma plataforma interoperável baseada no padrão FHIR (Smile Digital Health, s.d.).

Esta solução fornece uma série de funcionalidades essenciais para a interoperabilidade de sistemas de saúde, destacando-se como uma solução robusta e flexível para implementação do padrão FHIR. Sua documentação detalha as principais funcionalidades e arquiteturas disponíveis, que serão analisadas a seguir.

O *framework* provê um mecanismo integrado para realizar a conexão com servidores FHIR, utilizando dois tipos de *Rest Client* para essa finalidade, sendo eles (HAPI FHIR, 2025a):

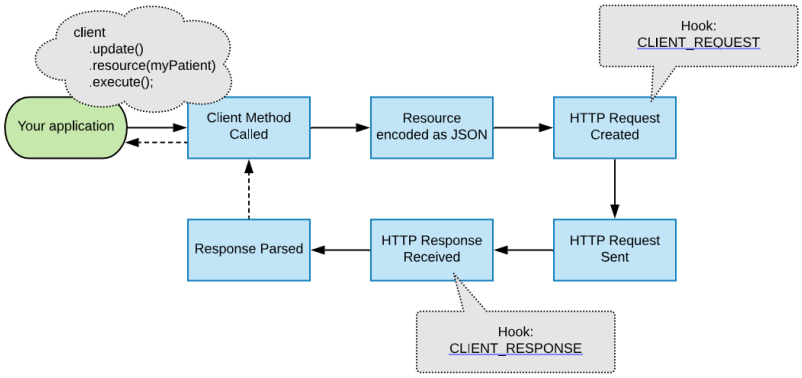
a) *Generic Client*: Projetado para ser flexível e de fácil uso, ideal para iniciantes no *framework*. Este *Client* é projetado usando uma interface *Fluent* , o que facilita a legibilidade e o uso dela.

> Uma interface *Fluent* na engenharia de software, é uma forma de compor o código dependendo do encadeamento de métodos, com a finalidade de aumentar a legibilidade do código (FOWLER, 2005).

b) *Annotaion Client*: Projetado para ser mais seguro, onde uma configuração inicial de maior complexidade deve ser feita. Ideal para separação de responsabilidades, quando quem define os métodos não é quem os utiliza. Usa vinculação estática, o que garante maior segurança de tipos.

A Figura 10 mostra a *pipeline* de processamento das requisições para processar uma *request do Client*:

Figura 10 – Estrutura do *Client* do HAPI FHIR



Fonte: (HAPI FHIR, 2025b)

O *framework* provê vários mecanismos para a construção de servidores FHIR, esta escolha deve se dar pelas especificações do projeto que será implementado. A documentação comenta dois principais mecanismos (HAPI FHIR, 2025c):

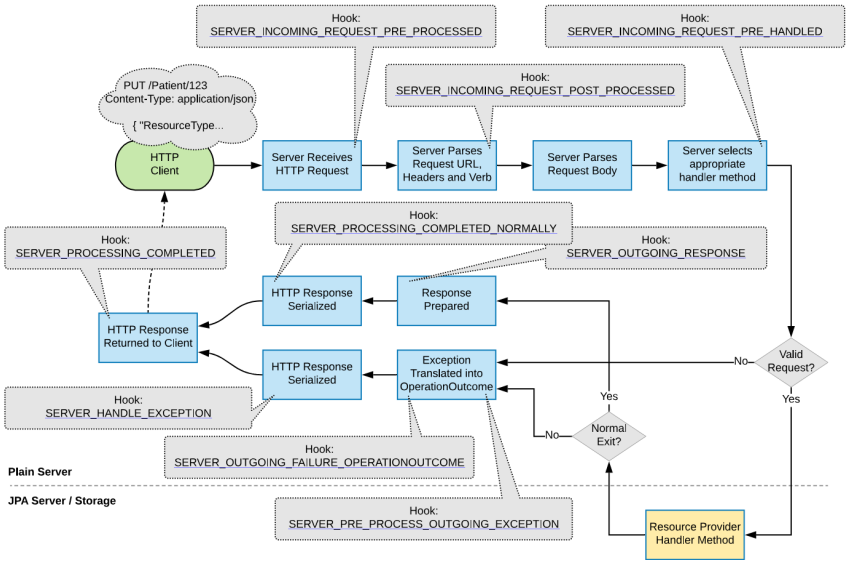
a) *Plain Server / Facade*: Implementa um servidor FHIR sobre um sistema já existente, tendo as funcionalidades de serialização e *parsing* dos recursos, processamento HTTP e semântica REST para o padrão FHIR. Fica a cargo do desenvolvedor implementar a lógica de armazenamento e recuperação dos dados.

b) JPA *Server*: Implementa um servidor FHIR completo com um banco de dados relacional próprio, sem a necessidade de uma customização própria. Esta solução é ideal para o desenvolvimento rápido de aplicações, por ser mais escalável e ter o esquema de banco de dados pronto. A documentação comenta que esse mecanismo foi um sucesso para aplicativos mobile e para sistemas de governos e empresas de grande porte, onde a escalada de pacientes é muito grande.

O mecanismo de JPA *Server* pode ser combinado também com o *Master Data Management* (MDM), um módulo dentro do *framework*, permitindo que *links* sejam criados e compartilhados entre os recursos FHIR. O MDM é a prática e/ou tecnologia que oferece uma visão precisa dos dados de uma organização e os torna facilmente acessíveis para outras áreas de negócios (ORACLE, 2025).

A arquitetura do mecanismo de *Plain Server*, pode ser visualizada na Figura 11 e a do JPA *Server* na Figura 12.

Figura 11 – Componentes do *Plain Server*

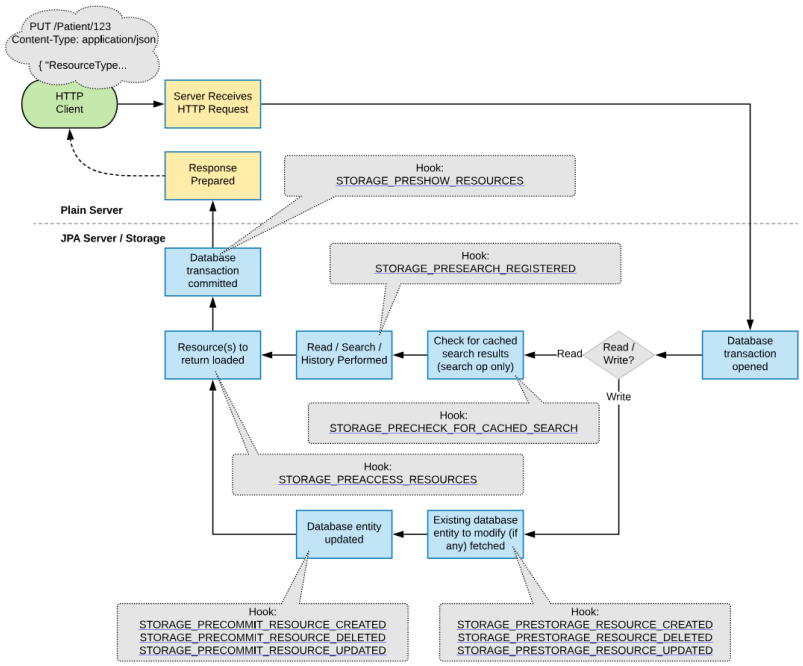


Fonte: (HAPI FHIR, 2025d)

Ambas as arquiteturas possuem abordagens distintas para armazenamento e processamento de dados. A documentação comenta que, dependendo da aplicação submetida, o fluxo pode mudar drasticamente.

A documentação comenta que a segurança vai além do *framework* e que cada sistema tem regras e requisitos diferentes. A arquitetura desse projeto fornece uma camada de segurança

Figura 12 – Organização do JPA *Server*



Fonte: (HAPI FHIR, 2025d)

única para todos os casos, oferecendo ferramentas e blocos de construção úteis para personalizar toda a arquitetura de segurança, dividida em três tópicos (HAPI FHIR, 2025e):

a) Autenticação (*AuthN*): Verificação se o usuário é quem ele diz ser, testando um usuário e senha um um *bearer token* na solicitação.

b) Autorização (*AuthZ*): Verificação se o usuário possui permissão para realizar a ação solicitada, como uma leitura por exemplo. Na autenticação, valida-se se o usuário possui acesso aos servidores, e na autorização se o mesmo pode realizar algumas ações dentro deste servidor.

c) Consentimento e Auditoria: Verificação se o usuário tem direito de ver ou modificar os recursos específicos solicitados. Um registro da ocorrência dessa solicitação é gravado para a auditoria futura.

O projeto fornece três principais abordagens para a realização das validações dos recursos (HAPI FHIR, 2025f):

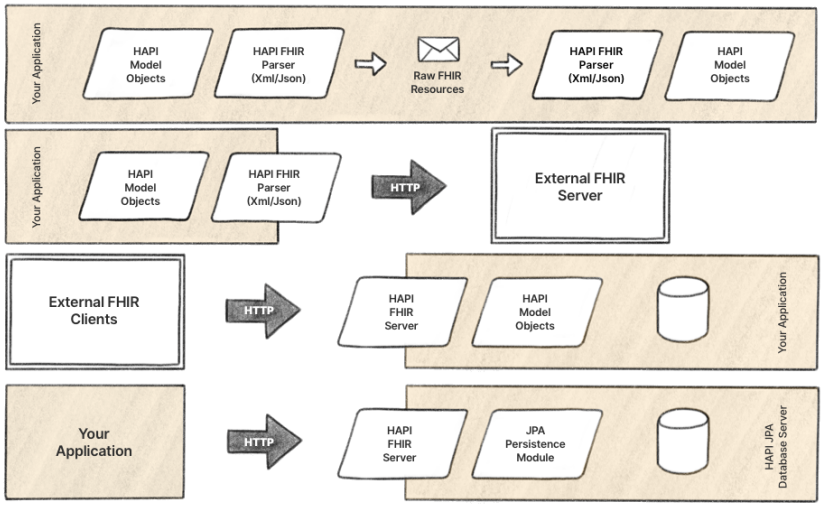
a) *Parse Error Handler*: Validação em tempo real durante o *parsing* de um recurso, com o objetivo de garantir que os dados de entrada possam ser mapeados para um modelo correto de dados do *framework*. Esta abordagem é menos abrangente que as demais, mas é rápida, leve e garante a perda de dados durante o *parsing*.

b) *Instance Validator*: Validação de recursos contra as regras oficiais do padrão FHIR, como a *StructureDefinition*, estrutura dos dados enviados com as suas regras, e o *ValueSet*, conjunto de códigos extraídos destinados a um contexto específico.

c) *ShemaValidation*: Validação usando arquivos *XSD/Schematron* fornecidos pelo padrão FHIR. Uma estrutura de validação depreciada, possui mensagens de erros menos claras.

O criador do projeto HAPI FHIR explica que, ao arquitetar e desenvolver a solução, o principal objetivo era garantir flexibilidade - oferecer uma maneira combinável e adaptável de incorporar recursos FHIR em aplicativos. A Figura 13 ilustra algumas das formas sugeridas pelo autor para utilizar o *framework*. Por exemplo, a primeira forma apresentada na figura ilustra o uso do HAPI FHIR apenas como biblioteca dentro da aplicação, onde os *model objects* são criados internamente e, em seguida, convertidos para formatos padrão FHIR (XML ou JSON) por meio do *HAPI FHIR Parser*. Esse fluxo permite que a aplicação gere e interprete recursos FHIR localmente, trocando apenas os dados brutos com outros sistemas quando necessário, sem atuar como servidor FHIR completo. Esse é um cenário comum quando a aplicação precisa apenas produzir ou consumir recursos FHIR, mas não expor uma API FHIR própria.

Figura 13 – Formas sugeridas pelo autor para o uso do *framework* HAPI FHIR



Fonte: (Hapi FHIR, 2025)

Tendo essa visão geral do *framework Hapi FHIR* apresentada, torna-se notório sua aplicação para garantir que todos os protocolos do padrão FHIR sejam respeitados, além de uma implementação completa e flexibilizada, onde a construção de algum sistema que envolve a interoperabilidade entre dados da saúde, se tornará extremamente escalável com essa solução. Contudo, para o contexto do presente trabalho, há um problema que essa solução não resolve, a complexidade. Ao mesmo tempo que fornece uma base rica de informações e recursos para o uso do protocolo FHIR na integração de dados da saúde, ainda existe uma curva de aprendizado

muito grande para se implementar tudo. Além disso, o servidor, hospedagem e afins, ficam a mercê de quem vai desenvolver e buscar soluções para garantir uma configuração segura e eficaz.

3.4 RNDS

A RNDS é uma plataforma brasileira desenvolvida para promover a interoperabilidade de dados de saúde em âmbito nacional, utilizando o padrão FHIR. Criada durante a pandemia de COVID-19, sua primeira aplicação prática foi a notificação centralizada de resultados de exames para detecção do SARS-CoV-2, vírus causador da doença. Até o momento desta pesquisa, a solução ainda se limita a esse escopo específico, porém os autores destacam planos de expansão para outras funcionalidades. Antes da implementação da RNDS, os laboratórios armazenavam os laudos exclusivamente em seus bancos de dados locais. Com a adoção obrigatória da plataforma, houve uma significativa centralização desses registros, otimizando o gerenciamento de dados durante a crise sanitária (RNDS, s.d.d).

Esta solução tem como público alvo os estabelecimentos de saúde de todo o país (RNDS, s.d.f). A ideia é conectar o gestor do estabelecimento junto com um desenvolvedor que irá realizar a integração com a API do RNDS. O gestor precisará de um certificado digital e uma conta no sistema brasileiro, o *gov.br*, só assim conseguirá solicitar o acesso ao ambiente de homologação e produção para enviar os laudos. O desenvolver por sua vez, deverá integrar com o RNDS, primeiramente em ambiente de homologação, para se gerar evidências que comprovem o envio e, essas evidências deverão ser enviadas pelo gestor para a solicitação do ambiente de produção. A Figura 14 representa o diagrama BPMN do fluxo de credenciamento do RNDS (RNDS, s.d.b; RNDS, s.d.c).

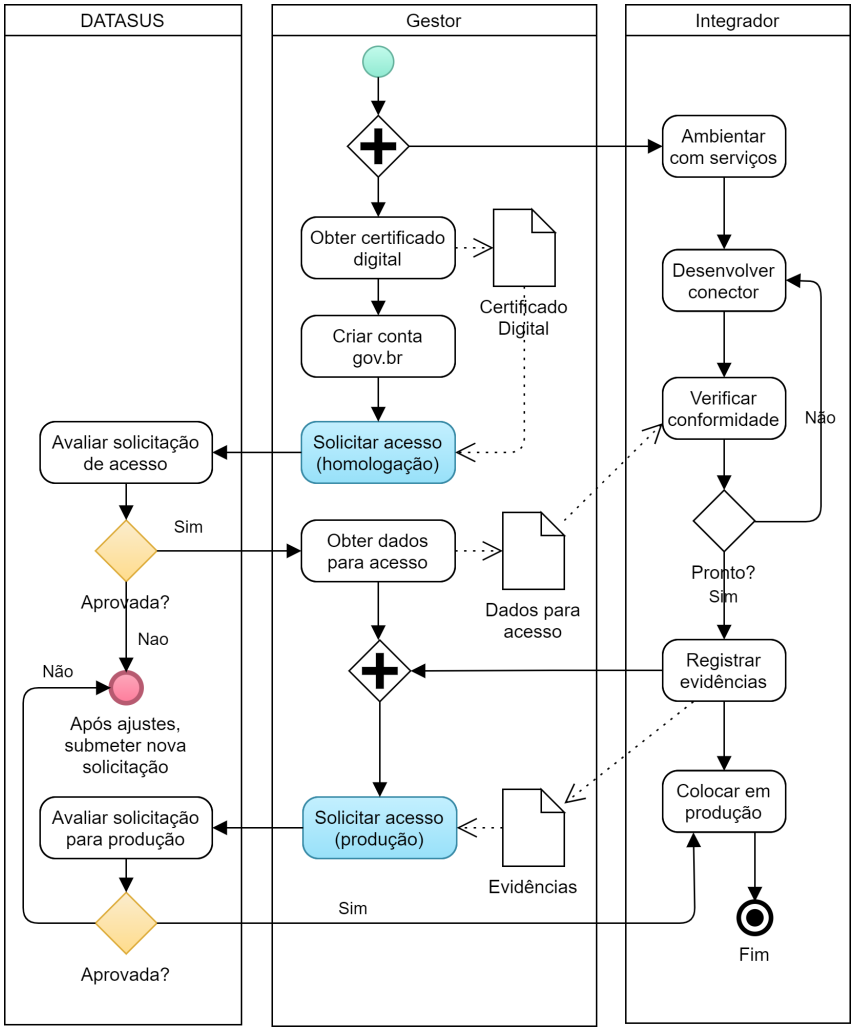
O RNDS não possui um *Client* para enviar as requisições, ele é uma API credenciada do governo brasileiro para a troca de informações autorizadas. A troca de informações se dá por meio de dois endereços (RNDS, s.d.a):

a) O primeiro é o endereço de autenticação, chamado pelos autores da documentação de *Auth*. Nele deve-se enviar o certificado previamente configurado para obter um *token*, este será enviado em todas as requisições por meio de um *header*. Todos os estabelecimentos devem usar o mesmo endereço para obter o *token*.

b) O segundo, denominado *Electronic Health Record* (EHR), é para o acesso aos serviços de troca de informações em saúde. Cada um dos estados do Brasil possui um link dedicado, e é de responsabilidade do desenvolvedor utilizar o link adequado dependendo do estado do estabelecimento que estará enviando as requisições.

A API do RNDS disponibiliza uma lista de serviços, divididos em duas categorias principais: segurança e envio de dados. Os serviços de segurança são apresentados na Tabela 5,

Figura 14 – Diagrama BPMN do fluxo de credencimaneto do RNDS



Fonte: (RNDS, s.d.e)

enquanto os de envio de dados estão detalhados na Tabela 6 (RNDS, s.d.g).

Tabela 5 – Serviços de segurança do RNDS

Método	**Path**	**Descrição**
GET	/api/token	Obtém token de acesso
POST	/api/contexto-atendimento	Obtém token para contexto de atendimento

Fonte: Reproduzido de (RNDS, s.d.g)

Tabela 6 – Serviços de envio de dados do RNDS

Método	**Path**	**Descrição**
GET	/api/fhir/r4/Patient	Obter informações sobre paciente.
GET	/api/fhir/r4/Organization	Obter informações sobre um estabelecimento de saúde ou outra organização.
GET	/api/fhir/r4/Practitioner	Obter informações sobre profissional de saúde
GET	/api/fhir/r4/PractitionerRole	Obter informações sobre papéis desempenhados por profissionais de saúde.
POST	/api/fhir/r4/Bundle	Enviar resultado de exame
POST	/api/fhir/r4/Bundle	Substituir resultado de exame

Fonte: Reproduzido de (RNDS, s.d.g)

A solução desenvolvida pelo governo brasileiro alinha-se significativamente aos objetivos deste trabalho, sendo que, dentre os outros trabalhos e soluções apresentadas, a que mais se aproxima com este trabalho. No entanto, conforme mencionado anteriormente, sua aplicação ainda está restrita ao envio de resultados de exames relacionados ao SARS-CoV-2, representando uma limitação.

Tendo em vista todo o repertório de trabalhos e soluções discutidas, retoma-se a Tabela 4, anteriormente apresentada, a qual reúne os principais pontos de conclusão e destaca as relações de cada estudo com o presente trabalho.

4 PROPOSTA DE SOLUÇÃO

O objetivo deste trabalho é propor uma arquitetura de *software* e desenvolver uma solução para uma REST API, com a finalidade de sincronizar dados de saúde de usuários, isto é, armazenar e disponibilizar posteriormente de forma eficiente. O projeto foi desenvolvido de maneira *open source*, com o código-fonte disponível publicamente. No entanto, uma versão hospedada em nuvem foi disponibilizada, pronta para uso imediato, com uma cobrança pelo acesso, como um SaaS. Caso os usuários prefiram não utilizar o serviço contratado, o código fonte do projeto está documentado de forma detalhada, permitindo que realizem a hospedagem por conta própria. Além disso, a solução proposta foi elaborada em conformidade com a LGPD, sendo portanto um software nacional.

Não é escopo deste trabalho criar a lógica para a cobrança da hospedagem do serviço, e sim a API para integrar os dados. Além disso, foram desenvolvidos os principais *endpoints* da aplicação, mas dependendo do contexto de quem irá usar, novos recursos deverão ser implementados para o correto uso do *software*.

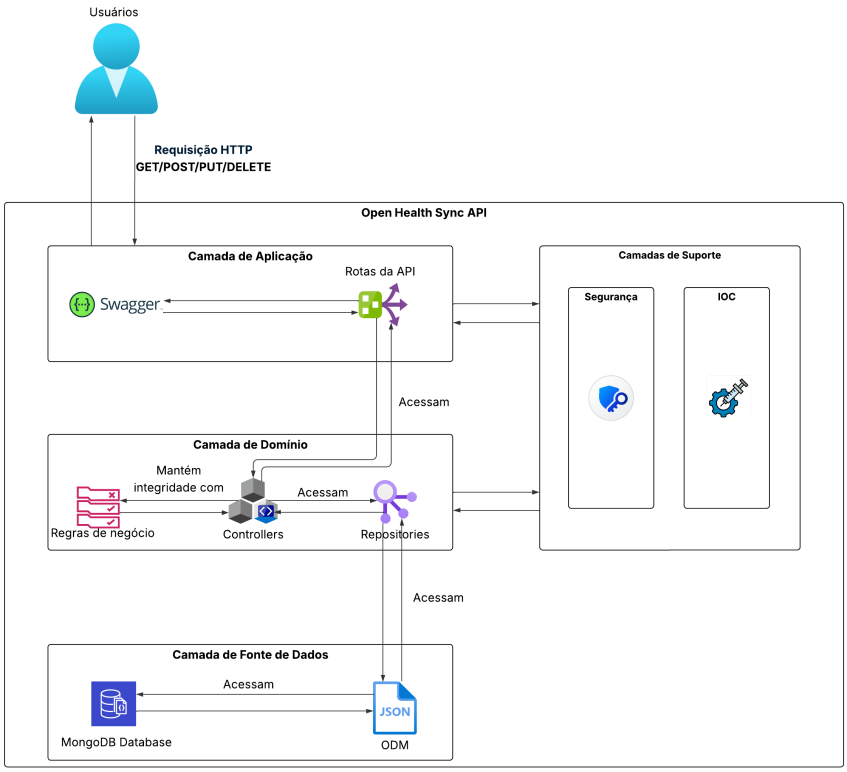
No decorrer desta seção, serão apresentadas os diagramas de casos de uso e de sequência planejados para a aplicação, a arquitetura candidata e a explicação das tecnologias utilizadas para validar a ideia geral do projeto.

4.1 ARQUITETURA DA API

Para atender aos objetivos propostos, adotou-se a arquitetura ilustrada na Figura 15, que apresenta a estrutura em camadas da API *Open Health Sync*.

Como discutido anteriormente sobre arquitetura em camadas, este projeto adota uma estrutura dividida em três principais camadas. A primeira, a camada de apresentação, inclui as rotas da API e o *Swagger* como interface para documentação e visualização dos *endpoints*. Em segundo, a camada de domínio, representada pelos *controllers*, concentra a lógica de processamento das requisições e validações, mantendo a integridade com as regras de negócio da API e acessando os *repositories*, utilizados para intermediar o acesso e a manipulação dos dados no banco de dados. Por fim, a terceira camada, a de fonte de dados, utilizando o *Object Data Modeling* (ODM) juntamente com o *MongoDB* para o armazenamento e recuperação de informações. Além delas, temos outras camadas auxiliares para a arquitetura, uma de segurança, representando a autenticação da API, e a outra de *Inversion of Control* (IOC), para promover o desacoplamento e a usabilidade das camadas. Essa estrutura amplia a modularidade da API, viabilizando atualizações e escalonamento individualizado de cada componente.

Figura 15 – Arquitetura da API Open Health Sync



Fonte: O Autor (2025).

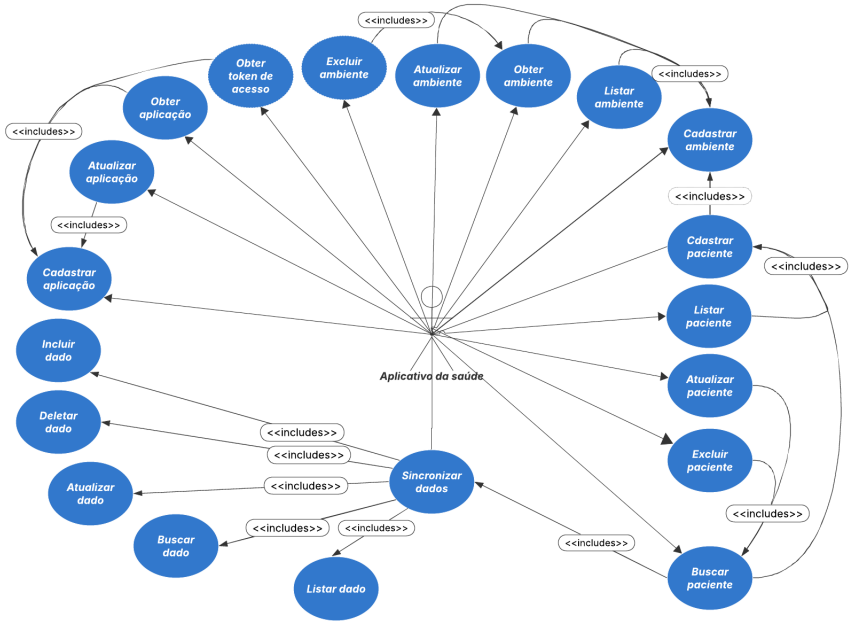
4.1.1 CASOS DE USO

Os casos de uso oferecem uma abordagem estruturada para capturar os requisitos comportamentais de um sistema, permitindo a criação de um projeto bem fundamentado. Esses casos de uso auxiliam na resposta a questões fundamentais, como: o que os usuários do sistema pretendem realizar? Qual é a experiência do usuário? Grande parte da funcionalidade do software é definida pela forma como os usuários interagem com ele. Além disso, os casos de uso fornecem uma base sólida para o design do sistema, além de permitirem estimativas mais confiáveis de tempo e esforço necessários para o desenvolvimento (ROSENBERG; SCOTT, 1999).

Para este projeto, o autor principal não se concentra em apenas uma entidade, e sim várias, como por exemplo aplicativos, hospitais, serviços de saúde que precisarão de algum lugar para armazenar os dados de saúde dos seus pacientes. A Figura 16 representa o diagrama de casos de uso planejado para aplicação.

- **Cadastrar aplicação:** A aplicação cliente deve registrar-se na API, fornecendo credenciais (usuário e senha) para autenticação futura.
- **Atualizar aplicação:** A aplicação cliente pode atualizar seus dados, previamente cadastrados.
- **Obter aplicação:** A aplicação cliente pode buscar seus dados, previamente cadastrados.

Figura 16 – Diagrama de casos de uso para a API



Fonte: O Autor (2025).

- **Obter *token* de acesso:** a aplicação deverá enviar o usuário e a senha cadastrados anteriormente para obter o *token* de acesso para acessar as outras funcionalidades de API.
- **Cadastrar ambiente:** a aplicação pode cadastrar um ambiente para o envio posterior dos dados.
- **Atualizar ambiente:** a aplicação pode atualizar o ambiente previamente cadastrado, isso incluiu sua desativação.
- **Obter ambiente:** a aplicação pode obter um ambiente para o envio posterior dos dados.
- **Excluir ambiente:** a aplicação pode excluir um ambiente não mais desejado.
- **Listar ambiente:** a aplicação pode listar os ambientes disponíveis.
- **Cadastrar paciente:** a aplicação pode cadastrar um paciente dentro de um ambiente específico para o envio posterior de seus dados.
- **Atualizar paciente:** a aplicação pode atualizar os dados de um paciente previamente cadastrado em um ambiente.
- **Excluir paciente:** a aplicação pode excluir um paciente, incluindo seus dados sincronizados.
- **Buscar paciente:** a aplicação pode buscar os dados cadastrados para um paciente.
- **Excluir paciente:** a aplicação pode excluir um paciente desejado.

- **Sincronizar dados:** a aplicação pode buscar sincronizar os dados de um paciente, isso incluiu as ações de inclusão, remoção, atualização, busca e a listagem destes dados.

4.1.2 DIAGRAMAS DE SEQUÊNCIA

De acordo com (ROSENBERG; SCOTT, 1999), os diagramas de sequências são utilizados para representar a interação temporal entre objetos no contexto de um caso de uso específico. Ele ilustra como os objetos se comunicam por meio de mensagens organizadas em uma linha do tempo vertical, permitindo descrever de forma precisa o fluxo de execução do sistema. Cada diagrama de sequência é derivado de um caso de uso e tem como objetivo detalhar o comportamento do sistema, alocando responsabilidades às classes envolvidas.

Nesta seção, apresenta-se dois diagramas de sequência, um para o caso de uso da autenticação e outro para o cadastro do paciente. Os outros não serão representados pois seguem em linha com os apresentados.

A Figura 17 representa o diagrama do processo de autenticação da aplicação cliente. Nesse fluxo, o cliente envia suas credenciais de acesso (e-mail e senha) por meio de uma requisição HTTP à API. Após a validação correta das informações enviadas, a API gera e retorna um *JSON Web Token* (JWT), que será utilizado para autenticar as próximas requisições aos outros *endpoints* protegidos do sistema. Sem esse *token*, o cliente não terá acesso às funcionalidades seguras da aplicação.

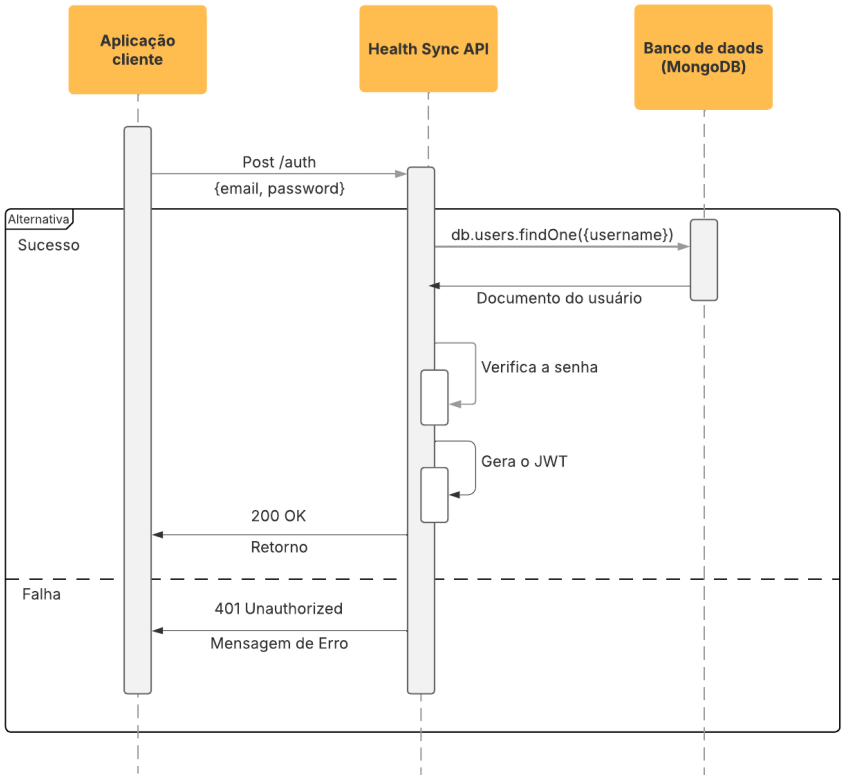
A Figura 18 ilustra o fluxo de cadastro de um novo paciente por meio da API. Para realizar essa operação, o cliente deve enviar uma requisição HTTP contendo os dados do paciente, acompanhados de um *token* JWT válido e do id do ambiente autorizado. Após o recebimento, a API realiza as validações de segurança necessárias. Caso o *token* e o ambiente sejam considerados válidos, os dados são persistidos no banco de dados *MongoDB*, e a API retorna a como resposta o identificador único do paciente cadastrado. Em caso de erro na autenticação ou validação, uma mensagem adequada contendo o motivo do erro é retornada para o cliente, impedindo o acesso não autorizado à funcionalidade.

4.1.3 DIAGRAMAS DE CLASSES

O diagrama de classes, tem como objetivo mostrar as principais classes do sistema, com seus atributos, métodos e relacionamentos (ROSENBERG; SCOTT, 1999). A Figura 19, representa o diagrama de classes que foi utilizado na implementação deste trabalho.

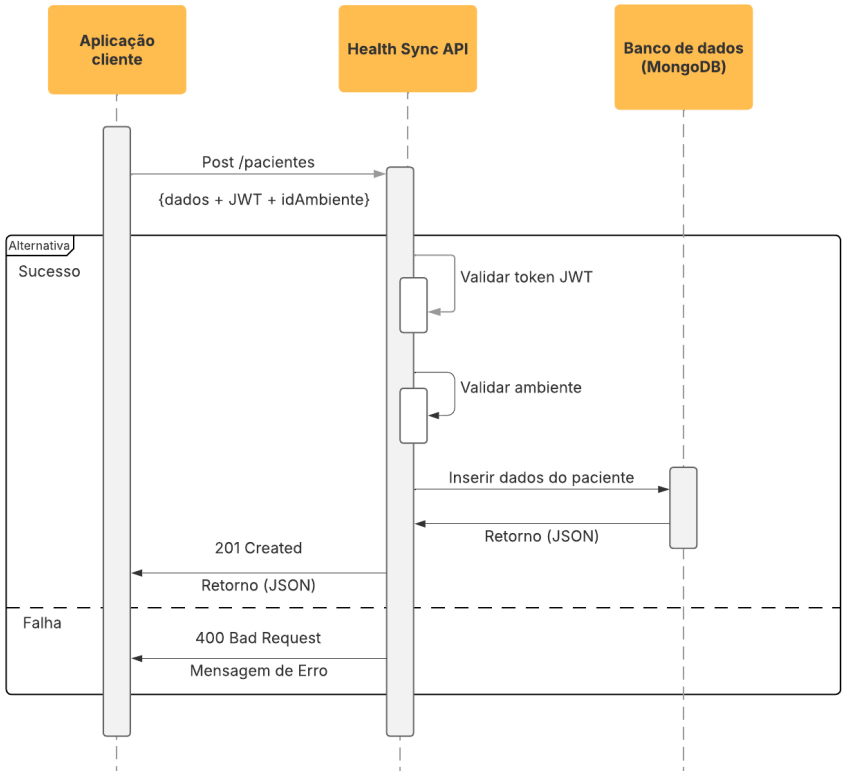
Como se pode observar na Figura 19, o projeto gira em torno da figura da *"Aplicacao"*. Esta representa quem efetivamente contrata e administra o uso da API projetada. Essa entidade está diretamente ligada ao *"Ambiente"*, que estabelece o contexto de operação, como os ambientes de produção e testes, para onde os dados serão sincronizados. Dando seguimento, o

Figura 17 – Diagrama de sequência para a autenticação na API



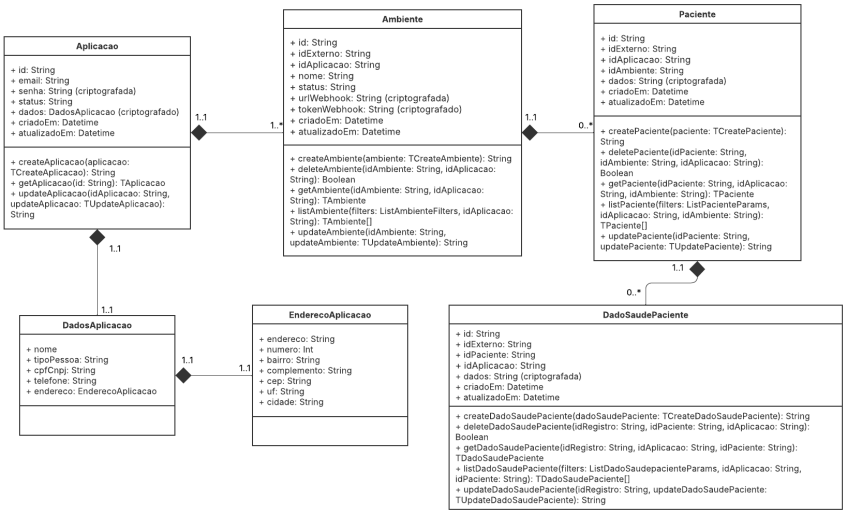
Fonte: O Autor (2025).

Figura 18 – Diagrama de sequência para o cadastro de um paciente



Fonte: O Autor (2025).

Figura 19 – Diagrama de classes para a solução proposta



Fonte: O Autor (2025).

"Paciente" é quem recebe os dados de cadastro, e o *"DadoSaudePaciente"* guarda as informações, sejam elas clínicas ou transacionais, referentes a cada paciente.

4.2 TECNOLOGIAS UTILIZADAS

Na seção Seção 2.5 comenta-se sobre as principais tecnologias que foram utilizadas no decorrer deste trabalho, porém, não é comentado a forma como essas tecnologias irão se interconectar. Está seção visa explicar melhor como essa conexão será aplicada, visto que, as tecnologias citadas possuem formas nativas de realizar essa conexão, porém existem bibliotecas e *frameworks* que facilitam e abstraem melhor essa parte, gerando mais escalabilidade e segurança.

Vale destacar que, para o presente projeto, foi utilizada a versão 22.18 do *node.js*, e os seguintes *frameworks* e bibliotecas aqui apresentados foram incorporadas no projeto utilizando o próprio gerenciador de pacotes nativo do *Node*, o *npm* na sua versão 10.9, ambos previamente apresentados.

4.2.1 FASTIFY FRAMEWORK

Para a criação do servidor HTTP e a manipulação das rotas, escolheu-se o *framework fastify* 5.3. O *Node* já possui uma forma nativa pra realizar essas ações, porém dessa forma, acarreta baixa modularidade, dificuldade de escalar, e o desenvolvedor precisa implementar as funcionalidades de forma manual, como tratamento de requisições e respostas, além do *parsing* dos dados. Este *framework* visa resolver esse problema ao fornecer alto desempenho - suportando até trinta mil requisições por segundo -, ser extensível por meio de *hooks* e *plugins* com funcionalidades prontas, e ser expressivo, facilitando seu uso e suprindo limitações

da abordagem nativa do *Node.js*. Quando cria-se um servidor dessa forma, pode-se ganhar no custo menor de infraestrutura sem afetar a carga do servidor. Além desses pontos comentados, o *fastify* é compatível com o *TypeScript*, que será apresentado adiante. Com isso, tem-se um *framework* robusto, extensível, de fácil uso e que suporta uma grande quantidade de requisições, tornando-o ideal para este trabalho (OpenJS Foundation and The Fastify Team, 2025).

4.2.2 MONGOOSE

Para o banco de dados, foi utilizado o *MongoDB*, previamente apresentado. Este banco, possui seu *driver* nativo para realizar as operações, isso permite uma maior flexibilidade sobre quais dados irão ser inseridos. No entanto, essa flexibilidade excessiva pode resultar em inconsistências ao longo do tempo, comprometendo a integridade dos dados e, eventualmente, a performance da aplicação. Considerando esse cenário, optou-se por utilizar a biblioteca *Mongoose* 8.15 como ODM, que atua como intermediador nas operações com o banco, oferecendo uma camada de abstração que equilibra flexibilidade com validação e estruturação dos dados (MongoDB Inc., 2024).

Com o *Mongoose*, pode-se forçar a esquemas rígidos para os documentos salvos no banco de dados, mantendo um padrão. Para a aplicação desenvolvida neste trabalho, de forma geral, o banco de dados precisava ser flexível. Entretanto, havia informações obrigatórias para os documentos. Com a utilização desta biblioteca, foi possível alcançar essa flexibilidade sem abrir mão de esquemas rígidos, garantindo maior segurança e consistência dos dados. Na seção Seção 4.1.3, comentou-se sobre o diagrama de classes do projeto, onde essa relação entre flexibilidade e rigidez do esquema se torna mais evidente (MongoDB Inc., 2024). De forma geral, entidades que possuem a propriedade genérica "*dados*"não utilizam um esquema rígido, permitindo armazenar informações variadas conforme cada integração. É o caso da entidade "*Paciente*", cuja estrutura depende principalmente desse campo. Em contraste, entidades com propriedades bem definidas, como "*Aplicacao*", adotam esquemas rígidos para garantir consistência e validação estruturada.

4.2.3 TYPESCRIPT

Como linguagem de programação, foi apresentada o *JavaScript*, mais especificadamente o *Node* 22.16, sua versão interpretada para servidor. Quando essa linguagem surgiu, não se esperava que ela fosse crescer tanto, logo ela possui algumas peculiaridades que podem muitas das vezes, atrapalhar o desenvolvedor, como por exemplo, a maioria das linguagens geram erros quando algum tipo ou propriedade acessada em alguma variável é inválida, o *JavaScript* não. Dado este problema e outros aqui não citados, surgiu o *TypeScript*, uma linguagem que é um superconjunto do *JavaScript*, adicionando um verificador de tipos estáticos e regras sobre como diferentes tipos de valores podem ser usados. Com ela, o desenvolvimento pode-se tornar mais estruturado, visto que, agora tem-se regras mais definidas que garantem a lógica de negócio,

adicionando uma maior segurança no projeto, além de claro, todos os benefícios de uma linguagem fortemente tipada (Microsoft Corporation, 2025). Foi utilizado o *TypeScript* na versão 5.8 para este trabalho.

As bibliotecas e *frameworks* citadas nesta seção, são as principais que foram utilizadas neste trabalho, porém vale destacar outras bibliotecas menores, que, podem possuir uma única função no projeto, mas são de suma importância para garantir seu funcionamento. São elas:

- ***jsonwebtoken 9.0:*** Biblioteca responsável por gerenciar e gerar os *tokens* JWT para a autenticação na API.
- ***bcrypt 6.0:*** Biblioteca responsável por criptografar senhas de forma segura, utilizando um algoritmo de *hashing* numa sequência irreversível de caracteres.
- ***mongoose-encryption 2.1:*** Biblioteca responsável por criptografar os campos escolhidos dos esquemas criados com o *Mongoose*.
- ***eslint 9.33:*** Biblioteca utilizada para padronizar o código e identificar erros ou más práticas em tempo de desenvolvimento.
- ***prettier 3.6:*** Ferramenta de formatação automática de código, garantindo consistência no estilo e na identação do projeto.
- ***eslint-config-prettier 10.1:*** Configuração que integra o *Prettier* ao *ESLint*, evitando conflitos entre regras de formatação. ***globals 16.3:*** Biblioteca que fornece uma lista padronizada de variáveis globais de diferentes ambientes (*Node.js*, *browser*, etc.), auxiliando ferramentas de análise de código.
- ***pino-pretty 13.1:*** Extensão do *Pino* que formata os registros de log, tornando-os mais legíveis durante o desenvolvimento.
- ***tsx 4.20:*** Executa arquivos *TypeScript* diretamente, sem necessidade de compilação prévia, facilitando o ambiente de desenvolvimento.
- ***typescript-eslint 8.39:*** Conjunto de ferramentas que permite usar o *ESLint* em projetos escritos com *TypeScript*.
- ***vitest 3.2:*** Framework de testes rápido e leve, inspirado no *Jest*, usado para validar o comportamento da aplicação.
- ***cpf-cnpj-validator 1.0:*** Biblioteca responsável por validar e formatar números de CPF e CNPJ conforme os padrões brasileiros.
- ***@fastify/cors 11.1:*** Plugin do *Fastify* que habilita o uso do CORS, permitindo o controle de acesso entre diferentes domínios.

- ***@fastify/swagger 9.5:*** Plugin responsável por gerar automaticamente a documentação da API no formato OpenAPI.
- ***@fastify/swagger-ui 5.2:*** Extensão que fornece uma interface visual interativa para explorar e testar os endpoints da API.
- ***@vercel/node 5.3:*** Biblioteca que permite a execução e implantação de funções *serverless* em ambientes da *Vercel*.
- ***dotenv 17.2:*** Gerencia variáveis de ambiente a partir de um arquivo `.env`, mantendo dados sensíveis fora do código-fonte.
- ***fastify-type-provider-zod 5.0:*** Integra o *Zod* ao *Fastify*, permitindo validação de tipos e esquemas nas rotas.
- ***zod 4.0:*** Biblioteca usada para validação e definição de esquemas de dados de forma segura e tipada em *TypeScript*.
- ***ts-node 10.9:*** Ferramenta que permite executar arquivos *TypeScript* diretamente no ambiente *Node.js*, sem necessidade de compilação manual.
- ***globals 16.3:*** Biblioteca que fornece uma lista padronizada de variáveis globais de diferentes ambientes, auxiliando ferramentas como o *ESLint*.

Além dessas bibliotecas, foram utilizadas também as bibliotecas correspondentes com os *types* de cada uma. Por exemplo, o *node*, possui um conjunto de tipos específicos dele, onde pode-se incorporar a biblioteca *@types/node* para ter-se acesso à esses tipos. Um exemplo é o *type ReadDirOptions*, utilizado para as opções de leitura de arquivos com o *node*. Não se listou todos os pacotes de *types* utilizados no projeto. Para visualiza-los, pode-se acessar o arquivo *package.json* na raiz do projeto, lá terá todas as dependências externas utilizadas.

4.3 CRIPTOGRAFIA

Como este trabalho trata diretamente com dados sensíveis das pessoas e empresas, definiu-se um esquema de criptografia, visando a segurança da solução e para estar em conformidade com as regras descritas pela LGPD. Para isso, utilizou-se a biblioteca *mongoose-encryption*, previamente apresentada. Com ela, pode-se definir as chaves secretas para a criptografia, salvando campos específicos dos esquemas definidos. As chaves estão configuradas nas variáveis de ambiente do projeto, ou seja, fora do repositório onde o mesmo estará configurado. Com tudo, esse planejamento gera um problema, a validação dessas chaves secretas para o encriptação. Caso seja definida uma chave secreta inválida, ou seja, que não segue os padrões definidos pela biblioteca, poderá ocorrer erros no salvamento dos dados. Para resolver essa questão, adicionou-se uma verificação quando o servidor for inicializado, confirmando se

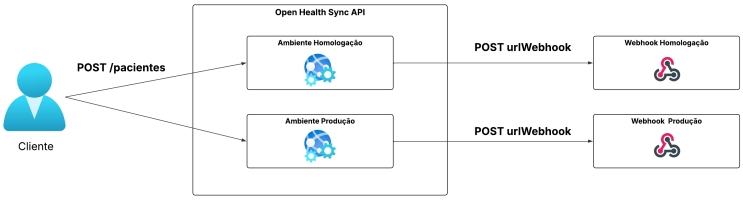
as variáveis de ambiente configuradas estão definidas e nos padrões previamente definidos na documentação do projeto. Desse forma, tem-se uma estrutura completa e segura para o gerenciamento da criptografia a nível de banco. Caso alguma configuração estiver incorreta, o servidor não irá ser inicializado, e uma mensagem de erro será apresentado ao desenvolvedor.

4.4 WEBHOOKS

Como previamente definido na Seção 2.3.3.1, os *webhooks* são um modelo de comunicação assíncrona, baseada em eventos. Com eles, pode-se distribuir o consumo de recursos, evitando um *pooling* constante na aplicação para buscar dados novos, além de facilitar a integração entre sistemas, melhorando a usabilidade e a escalabilidade. Para este trabalho, a API foi implementada como provedor; por isso, foram criadas duas propriedades no cadastro de um ambiente da aplicação, chamadas de *urlWebhook* e *tokenWebhook*, onde a aplicação que usará a API projetada poderá configurar uma *url* por ambiente, assim quando algum dado for alterado, inserido ou deletado, a API envia uma notificação para este *webhook* cadastrado, contendo o tipo do evento correspondente, os dados desse evento e um *header* fixo chamado de *X-Webhook-Token* contento o *token* configurado junto com a *url* do *webhook*. Esse mecanismo de configurar um *token* junto com a *url*, permite que o usuário consiga validar se a *request* é de origem da API, e não por um terceiro desconhecido, garantindo uma maior segurança. Caso nenhuma *url* for inserida, nenhum evento será disparado. Não foi escopo deste projeto implementar uma esquema de *retries* caso o envio para o *webhook* cadastro resultar em falha.

O fluxo descrito pode ser visualizado na Figura 20, onde é representado o envio de dados após cadastro de um paciente:

Figura 20 – Fluxograma de disparo para o *webhook*



Fonte: O Autor (2025).

Pensando na escalabilidade da aplicação, o modelo arquitetural adotado para o *webhook* utiliza injeção de dependência, um dos princípios da camada de *IOC*, assim irá se conseguir um baixo acoplamento e com a possibilidade de modificar a implementação sem mexer no código base. Inicialmente, o envio foi feito de forma síncrona, mas com esse princípio, no futuro, caso alguém queira implementar uma forma de diferente, como por exemplo, o envio para uma fila externa, irá precisar apenas passar um objeto diferente respeitando um interface definida.

4.5 DISPONIBILIDADE

O *software* desenvolvido neste trabalho pode ser disponibilizado de duas formas. A primeira é mediante à uma assinatura, onde o usuário irá pagar um valor em reais pré estabelecido e ganhará o acesso ao mesmo. A segunda é, caso o usuário não optar por assinar o *software* já hospedado, ele pode utilizar o código fonte disponibilizado no *GitHub* e hospeda-ló por conta própria, visto que o software é *open source*, ou seja, de código aberto. Para este segundo caso, no próprio projeto hospedado, existe um arquivo em *markdown* chamado "*README*", onde constam todas as instruções de como configurar as variáveis do projeto, bem como executa-lo localmente. A parte de configuração e *deploy* para uma plataforma de hospedagem não foi documentado, visto que existem diversas soluções atualmente no mercado, ficando a parte do usuário escolher e entender o que precisará ser feito em cada caso.

Para o primeiro caso, tem-se uma questão importante a ser definida, o valor que será de fato cobrada de um usuário para a utilização do software. Contudo, isso dependerá de qual provedor será escolhido para de fato hospedar, visto que cada um possui planos e valores diferentes, além do volume que o cliente irá gerar. A seguir, mapeou-se em três principais plataformas de computação em nuvem, que disponibilizam essa solução de hospedagem para realizar o estudo da cobrança, sendo elas, a *Vercel*, *Amazon Web Services* (AWS) e *Azure*.

- ***Vercel:*** Plataforma de hospedagem e infraestrutura para aplicações modernas, com foco em *JavaScript*. Ela fornece um produto que promete deixar o *deploy* de aplicações o mais fácil possível. Possui um plano grátis, chamado de *Hobby*, com algumas funcionalidades. O próximo plano se chama *Pro*, e custa vinte dólares mensais. Por fim, o último plano se chama *Enterprise*, e não possui um custo fixo, dependerá do quanto o cliente de fato usará.

- ***AWS:*** Plataforma de nuvem da empresa *Amazon*, fornecendo serviços de tecnologia de infraestrutura, como computação, armazenamento e banco de dados, além de tecnologias emergentes, como inteligência artificial e *data lakes*. Esta plataforma não possui um custo fixo mensal, e sim um custo conforme o uso de seus serviços, podendo variar. Possui um gratuidade por doze meses, porém, caso o uso dos serviços exceda a cota da gratuidade, ou se o período dessa gratuidade acabar, serão cobradas as taxas pela utilização dos serviços. A plataforma disponibiliza uma calculadora para o custo aproximado de uma determinada aplicação.

- ***Azure:*** Plataforma de nuvem muito parecida com a AWS, porém da empresa *Microsoft*. Ela fornece serviços de hospedagem, de computação e inteligência artificial. Possui um gratuidade nos doze primeiros meses após a criação da conta, mas com recursos limitados, após isso, o valor é conforme o uso. Assim como a AWS, possui uma calculadora para determinar o custo de uma aplicação.

Dada a pesquisa a cima, criou-se a Tabela 7 com a comparação dos principais pontos de cada serviço:

Tabela 7 – Comparativo entre plataformas de hospedagem em nuvem

Plataforma	**Foco principal**	**Nível de implementação**	**Gratuidade**	**Custo (R$)**	**Indicação**
Vercel	Frontend e Backend moderno (React, Next.js, Node.js)	Fácil	Sim – com recursos suficientes	0 a 30	Sites e apps web com backend leve
AWS	Infraestrutura completa e escalável	Complexa	Sim – 12 meses com limites	30 a 200+	Aplicações robustas e personalizadas
Azure	Soluções completas e escaláveis integradas com Microsoft	Complexa	Sim – 12 meses com créditos e recursos limitados	30 a 200+	Ambientes corporativos com tecnologias da Microsoft

Fonte: O Autor (2025).

A partir dos dados, informações e da análise da Tabela 7, pode-se definir melhor a cobrança da aplicação proposta. Inicialmente, a API poderia ser hospeda na própria *Vercel*, com o plano gratuito deles, assim o cliente poderá testar o serviço sem precisar pagar nada. Caso ele opte por continuar, será cobrado uma mensalidade de 30 reais[1], isso para custear o plano PRO da própria *Vercel*. Este valor já engloba a possível escalada de clientes na aplicação, porém poderá mudar para determinados clientes que utilizarão a API de uma maneira mais frequente. O ideal aqui seria criar planos para o uso da API, com *rate limits* e checagem de armazenamento por aplicação, e assim calcular o valor a ser cobrando do cliente, com tudo, este valor proposto irá englobar a maioria do clientes pequenos e médios que irão utilizar do serviço criado.

Vale destacar que, para o presente trabalho, uma versão de testes foi disponibilizada na plataforma *Vercel* para a validação da estrutura do projeto. Não é de responsabilidade deste trabalho criar o sistema de cobrança de fato, e sim planejar, arquitetar e criar o software para posteriormente ser hospedado.

4.6 LICENCIAMENTO

O *software* deste trabalho está disponível publicamente no *GitHub*, de forma *open source*, para que qualquer pessoa possa utilizar e modificar, mas isso não quer dizer que não

[1] O valor proposto levou em consideração a cotação atual do dólar, cerca R$ 5,60, podendo variar no futuro.

possui uma licença. Uma licença de software é um documento legal que diz quais direitos e deveres a entidade que irá usar, modificar e copiar deverá seguir, funcionando como um contrato, dizendo o que é e não é permitido fazer com aquele código. Para o software desenvolvido neste trabalho, adotou-se a licença *MIT* (Opensource.org, 2025), uma licença permissiva, que impõe poucas restrições, deixando a pessoa ou empresa com interesse no software desenvolvido fazer o que desejar com o mesmo, porém, precisará manter o nome do autor original do projeto nos créditos. A licença está disponível em um arquivo chamado *LICENSE.md* na raiz do projeto disponibilizado no *GitHub*.

4.7 INTERFACES DA API

Na Tabela 8, apresenta-se as principais interfaces disponibilizadas pela API projetada, também conhecidas como *endpoints*, que permitem a comunicação de fato com outros sistemas. Cada interface representa um recurso acessível, detalhando a ação que aquela interface irá realizar.

Tabela 8 – Tabela com os *endpoints* da API

Recurso	**Ação**	**Endpoint**	**Descrição**
Autenticação	POST	auth	Autenticar
Aplicação	POST	aplicacao	Criar aplicação
	GET	aplicacao/id	Consultar aplicação
	PUT	aplicacao/id	Atualizar aplicação
Ambiente	POST	ambiente	Criar ambiente
	GET	ambiente	Listar ambientes
	GET	ambiente/id	Consultar ambiente
	PUT	ambiente/id	Atualizar ambiente
	DELETE	ambiente/id	Excluir ambiente
Paciente	POST	ambiente/idAmbiente/paciente	Criar paciente
	GET	ambiente/idAmbiente/paciente	Listar pacientes
	GET	ambiente/idAmbiente/paciente/idPaciente	Consultar paciente
	PUT	ambiente/idAmbiente/paciente/idPaciente	Atualizar paciente
	DELETE	ambiente/idAmbiente/paciente/idPaciente	Excluir paciente
Registros de Saúde	POST	paciente/idPaciente/registro	Criar registro
	GET	paciente/idPaciente/registro	Listar registros
	GET	paciente/idPaciente/registro/idRegistro	Consultar registro
	PUT	paciente/idPaciente/registro/idRegistro	Atualizar registro
	DELETE	paciente/idPaciente/registro/idRegistro	Excluir registro

5 DESENVOLVIMENTO

Com base nos tópicos abordados nos capítulos anteriores e, sobretudo, na proposta de solução apresentada, o desenvolvimento da API foi realizado integralmente do início, sem a utilização de modelos ou projetos pré-existentes. Durante o desenvolvimento, foram necessários alguns ajustes pontuais na estrutura inicialmente proposta, sendo o maior deles a incorporação do *Repository Pattern* na abstração para o acesso e uso dos dados por meio do ODM. Essa abordagem visou facilitar a criação de testes e a manutenção do código.

Este capítulo tem como objetivo apresentar as etapas e processos do desenvolvimento da API, iniciando pela configuração do ambiente até o *deploy* e a finalização da implementação. Vale destacar que, o detalhamento da linha temporal do desenvolvimento, está no projeto no repositório no *GitHub*, onde estão disponíveis os códigos e resultados desse trabalho.

5.1 CONFIGURAÇÃO DO AMBIENTE

A configuração do ambiente da API, realizada antes do início do desenvolvimento dos *endpoints*, foi dividida em duas etapas principais: a configuração do ambiente *Node* com *TypeScript* e a instalação dos primeiros *frameworks* responsáveis pela estruturação e organização do código-fonte.

5.1.1 AMBIENTE NODE COM TYPESCRIPT

A primeira etapa consistiu na instalação e uso do *Node* versão 22.18, enquanto o *npm* foi utilizado para o gerenciamento das dependências e para a configuração inicial do ambiente de desenvolvimento da API.

Para a configuração do ambiente *Node* com *Typescript*, foi realizado a inicialização do arquivo de configuração do node, o *package.json* via *npm init -y*. Este arquivo guarda todas as informações principais do projeto, como dependências externas e suas versões, *scripts* que podem ser configurados para auxiliar o desenvolvimento e informações descritivas do projeto, como nome, versão e o autor.

Em seguida, instalou-se o *Typescript* juntamente com os tipos nativos do *Node* para o próprio *Typescript*, com o comando *npm install typescript @types/node ts-node -D*. O parâmetro "*-D*"no final do comando indica que estas dependências são destinadas exclusivamente para o ambiente de desenvolvimento, não sendo utilizadas em produção. O *Typescript* exige compilação prévia para *JavaScript* nativo a fim de ser executado, essa compilação é realizada por meio de uma ferramenta nativa do próprio *Typescript*, denominada *tsc*. Contudo, essa ferramenta é manual, toda vez se altera algum arquivo, é preciso compilar e executar o código novamente.

Para otimizar esse processo, foi instalado como dependência de desenvolvimento outra ferramenta, chamada de *tsx* (*npm install tsx -D*). Essa ferramenta permite a execução em tempo real dos arquivos modificados, realizando a compilação automática e reduzindo o tempo gasto durante o desenvolvimento.

Após a configuração inicial do ambiente com o *Node.js* e o *TypeScript*, realizou-se o ajuste do arquivo de configuração *tsconfig.json*, responsável por definir as diretrizes de compilação, como os diretórios de saída, os arquivos incluídos e a versão de *JavaScript* de destino. Em seguida, foi criado o diretório *src*, contendo o arquivo principal *index.ts*, utilizado para validar o funcionamento do ambiente configurado e servir como base inicial da API.

Com o ambiente validado, o projeto foi versionado em um repositório remoto no *GitHub*, incluindo a criação do arquivo *.gitignore*, destinado a excluir do controle de versão pastas e arquivos sensíveis, como o *.env*, que armazena as variáveis de ambiente. Essa etapa marcou a conclusão da configuração inicial do ambiente de desenvolvimento.

Por fim, as etapas seguintes de instalação de dependências seguiram a mesma estrutura de comando, mudando-se apenas entre dependências de desenvolvimento e de produção, assim omitindo os comandos completos nesta descrição.

5.1.2 PRIMEIROS FRAMEWORKS E ORGANIZAÇÃO DE CÓDIGO

Após a configuração inicial do projeto, iniciou-se a configuração dos primeiros *frameworks* e ferramentas para a organização e padronização do código da API. Foram utilizados o *Prettier*, responsável pela formatação automática do código e manutenção de um padrão consistente durante o desenvolvimento, e o *ESLint*, empregado para a análise estática e identificação de possíveis inconsistências ou más práticas.

Em seguida, foram adicionados os principais *frameworks* para a construção da API: o *Fastify*, utilizado como base para a criação dos *endpoints*; o *Zod*, voltado à validação de dados; e o *Swagger*, empregado na documentação da aplicação. Também foi utilizado o pacote *dotenv* para o gerenciamento das variáveis de ambiente. Além disso, configurou-se o *Docker* para conteinerizar o banco de dados, por meio do arquivo *docker-compose.yaml*.

Com essas configurações concluídas, estabeleceu-se a estrutura inicial do projeto e a base necessária para o desenvolvimento dos primeiros *endpoints* da API.

5.2 DESENVOLVIMENTO DOS ENDPOINTS DA API

O desenvolvimento dos *endpoints* deu-se com base na arquitetura proposta na Seção 4.1 e pelos casos de uso da Seção 4.1.1. Durante essa seção, serão apresentadas os detalhes de cada *endpoint*, bem como pacotes adicionais incorporados ao projeto para o desenvolvimento de uma determinada funcionalidade daquele *endpoint* específico.

- ***/aplicacao***: gerencia as operações relacionadas às aplicações registradas no sistema, oferecendo os seguintes métodos:
  - ***POST***: cria uma nova aplicação.
  - ***GET***: recupera uma aplicação específica, quando informado o *id*.
  - ***PUT***: atualiza os dados de uma aplicação existente, identificada por seu *id*.

  Durante o desenvolvimento destes *endpoints*, foram instalados alguns pacotes, entre eles estão o *bcrypt*, *mongoose*, *mongoose-encryption*, *cpf-cnpj-validator* como dependências globais e, por fim, o *pino-pretty* como dependência de desenvolvimento.

- ***/auth***: responsável pelo processo de autenticação da API.
  - ***POST***: realiza o processo de autenticação do usuário e gera o *token* de acesso. Precisa-se enviar o e-mail e a senha cadastrados no *endpoint* anterior da aplicação.

  Para este *endpoint*, foi instalado o pacote *jsonwebtoken*.

- ***/ambiente***: gerencia as informações relacionadas aos ambientes de uso da aplicação, oferecendo os seguintes métodos:
  - ***POST***: cria um novo ambiente.
  - ***GET***: lista todos os ambientes ou retorna um ambiente específico, quando informado o *id*.
  - ***PUT***: atualiza os dados de um ambiente existente, identificado por seu *id*.
  - ***DELETE***: remove um ambiente do sistema, conforme o *id* informado.

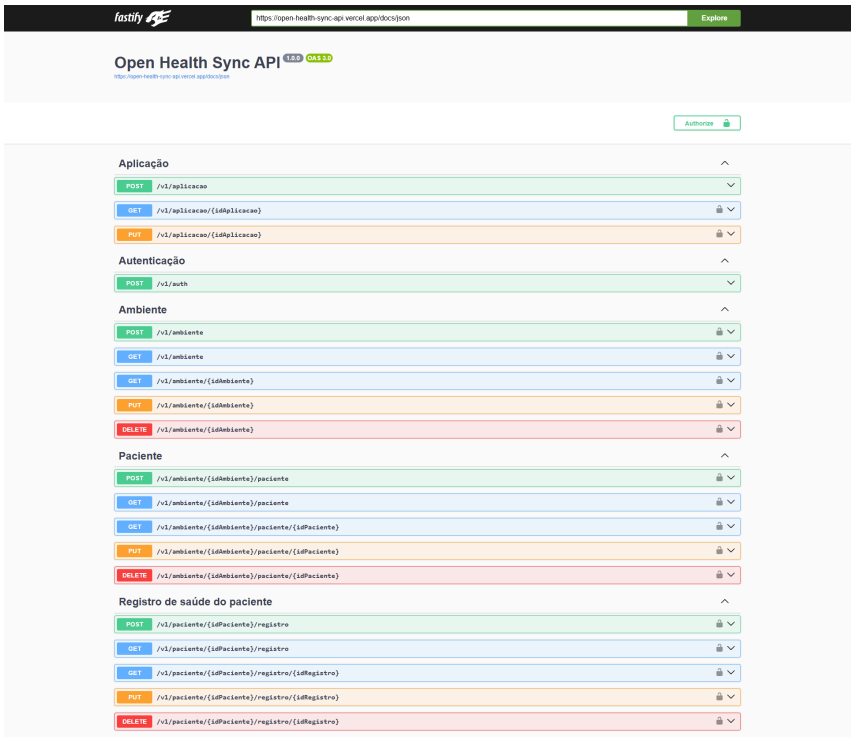
- ***/ambiente/{idAmbiente}/paciente***: gerencia os pacientes vinculados a um ambiente específico, oferecendo as seguintes operações:
  - ***POST***: cadastra um novo paciente no ambiente informado.
  - ***GET***: retorna a lista de pacientes de um ambiente ou um paciente específico, conforme o *idPaciente*.
  - ***PUT***: atualiza os dados de um paciente existente, identificado por seu *idPaciente*.
  - ***DELETE***: exclui o registro de um paciente do ambiente indicado.

- ***/paciente/{idPaciente}/registro***: responsável pelo gerenciamento dos registros de saúde de cada paciente, oferecendo os métodos:
  - ***POST***: cria um novo registro de saúde vinculado ao paciente especificado.
  - ***GET***: recupera todos os registros de saúde de um paciente ou um registro específico, quando informado o *idRegistro*.

– ***PUT***: atualiza os dados de um registro existente, identificado pelo *idRegistro*.
– ***DELETE***: exclui um registro de saúde do paciente informado.

Vale destacar que os *endpoints* das entidades de "*Ambiente*", "*Paciente*"e "*Registro de Saúde do Paciente*"apresentam dois comportamentos distintos para o método *GET*. Quando um *id* é informado na URL, a requisição retorna uma entidade específica correspondente daquele *endpoint*. Caso contrário, é retornada uma lista completa, podendo ser aplicados filtros específicos conforme os campos disponíveis em cada entidade. Contudo, são dois *endpoints* separados, apenas contém a mesma ação HTTP. Os detalhes técnicos de cada *endpoint* — como parâmetros de entrada, tipos de resposta e códigos de status — não foram descritos neste trabalho, pois há um *endpoint* dedicado à documentação da API, gerada automaticamente pelo *Swagger*, apresentando um nível de detalhamento mais completo. Essa documentação pode ser acessada pela rota */docs*. A Figura 23 mostra essa documentação dos *endpoints* da API, gerada pelo *swagger*.

Figura 21 – Documentação via *Swagger* da API



Fonte: O Autor (2025).

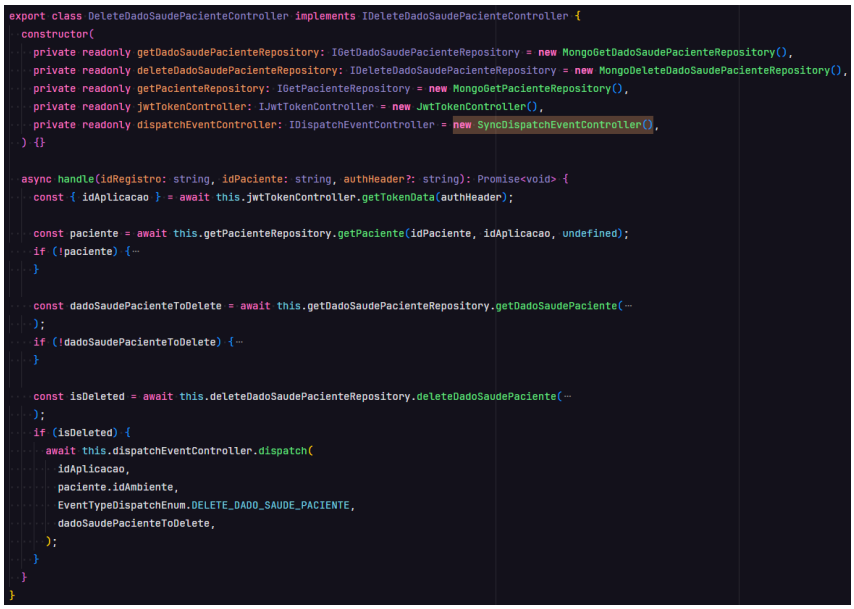
Vale destacar que os *endpoints* exibidos com o cadeado fechado na Figura 23 exigem autenticação prévia pelo *endpoint* de autenticação. Caso ela não seja realizada, não será possível acessar os recursos disponibilizados por esses *endpoints*.

5.3 WEBHOOK

Para o *webhook*, foi desenvolvida uma interface chamada de *IDispatchEventController* e, em cada *controller* que tem sua ação mapeada para disparar o evento, foi criado uma

propriedade na classe para guardar a instância desse *controller* responsável por esse disparo. Em cada classe com essa instância, criou-se um *controller* chamado de *SyncDispatchEvent-Controller* responsável por fazer o disparo de maneira síncrona, ao final do processamento da rota específica. Caso no futuro se desejar criar um *controller* para com uma lógica de disparo e controle mais complexo, basta respeitar a interface e mudar de *controller* nos locais desejados. Os eventos mapeados inicialmente foram os de criação, atualização e remoção de pacientes e de seus dados, os mesmo podem ser visualizados no arquivo *sync-dispatch-event-controller.ts*. A Figura 22 demonstra como essa arquitetura foi implementada na rota de exclusão de um paciente.

Figura 22 – Exemplo de implementação do *webhook* da rota de exclusão de paciente



Fonte: O Autor (2025).

5.4 DEPLOY

Após desenvolver os *endpoints* da aplicação, de autenticação e dos ambientes, iniciou-se o processo de *deploy*. Neste ponto do desenvolvimento do projeto, já possuía-se 9 dos 19 casos de uso e todas as tecnologias instaladas no projeto, tornando-se um bom momento para avançar à disponibilização da API em um ambiente online, a fim de validar seu funcionamento fora do ambiente local. Conforme comentado nas seções anteriores, optou-se pela *Vercel*, já que a mesma possui um plano grátis com recursos generosos. O *deploy* foi dividido em duas etapas principais, a de configuração do banco de dados no Atlas, e o *deploy* de fato para a *Vercel*.

Embora a *Vercel* tenha sido escolhida para este trabalho, o processo de *deploy* e hospedagem pode ser realizado em outros provedores de nuvem, uma vez que a arquitetura da aplicação foi desenvolvida de forma independente do ambiente de hospedagem. A escolha da *Vercel* se deu principalmente por sua facilidade de configuração e pela sua ausência de custo.

5.4.1 CONFIGURAÇÃO DO ATLAS

O Atlas é a versão SaaS do *MongoDB* disponível na nuvem . Para realizar essa configuração, foi utilizado o site oficial do *MongoDB Atlas*, configurando-se um *Cluster* do banco para ser utilizado. O armazenamento e o número de conexões é limitado, por conta do plano ser gratuito, porém não configura um limitante para este trabalho pois as demais funcionalidades se mantêm. Após criado o *Cluster*, utilizou-se a URL de conexão para aquele *Cluster*. Essa URL de conexão é um dado sensível, então colocou-se a mesma numa variável de ambiente chamada de *MONGODB_URI*, utilizando a prática de utilização de arquivos de configuração *ENV*.

5.4.2 DEPLOY NA VERCEL

Primeiro, foi criada uma conta na *Vercel* por meio da conta do *GitHub* onde o repositório da *API* está hospedado. Com isso, quando cria-se um novo projeto na *Vercel*, automaticamente detecta-se os repositórios listados no *GitHub*. Após selecionado o projeto da API, precisou-se configurar algumas coisas antes de fato subir o código:

Primeiro, criou-se uma conta na *Vercel* utilizando o *GitHub* onde o repositório da API está hospedado. A plataforma reconhece automaticamente os repositórios vinculados, permitindo selecionar o projeto desejado e prosseguir com as configurações necessárias antes do *deploy*. Essas configurações estão descritas a seguir:

Tabela 9 – Configurações de *Deploy* na Vercel

Configuração	**Descrição**
Framework Preset	Qual framework o projeto está usando. Foi deixado como "*Other*", já que não existe a opção *NodeJs*.
Root Directory	Diretório principal do projeto. Foi deixado em branco, para pegar o configurado no *package.json*.
Build Command	Comando para realizar a compilação do projeto. Foi configurado o comando *npm run build*. Esse comando foi definido no *package.json*.
Output Directory	Qual diretório o código compilado será colocado. Foi configurada a pasta */dist* a partir da raiz do projeto.
Install Command	Comando para instalar as dependências do projeto. Foi configurado o comando *npm install*, que executa e instala todos os pacotes configurados no *package.json* do projeto.
Environment Variables	Variáveis de ambiente para o projeto. Todas as variáveis de ambiente documentadas no repositório do projeto foram configuradas nesta parte, incluindo a *MONGODB_URI* comentada anteriormente com a URL do *Cluster Mongo* criado na nuvem.

Após a configuração inicial, a primeira tentativa de *deploy* na *Vercel* não teve sucesso, pois a plataforma não reconhecia a aplicação como uma API. Após análise, identificou-se que

a *Vercel* utiliza o modelo de *Serverless Functions*, no qual o código é executado sob demanda, sem necessidade de servidores dedicados. Para adequar o projeto a esse modelo, foi criada uma pasta */api* na raiz contendo o arquivo *index.ts*, responsável por exportar a instância da API como função padrão do *JavaScript*, além do arquivo *vercel.json*, configurado para redirecionar todas as requisições para essa função. Com esses ajustes, a API passou a operar corretamente no ambiente da *Vercel*.

Além do problema de configuração inicial, outro obstáculo enfrentado durante o *deploy* foi relacionado às importações de arquivos na *Vercel*. Durante o desenvolvimento, algumas pastas foram renomeadas de letras maiúsculas para minúsculas, mas o *Git*, em ambiente *Windows* (que não diferencia maiúsculas e minúsculas), não registrou essa alteração. Como a *Vercel* executa o código em ambiente *Linux*, que faz essa distinção, ocorreram erros de importação até que os nomes dos diretórios fossem corrigidos e sincronizados adequadamente com o repositório remoto.

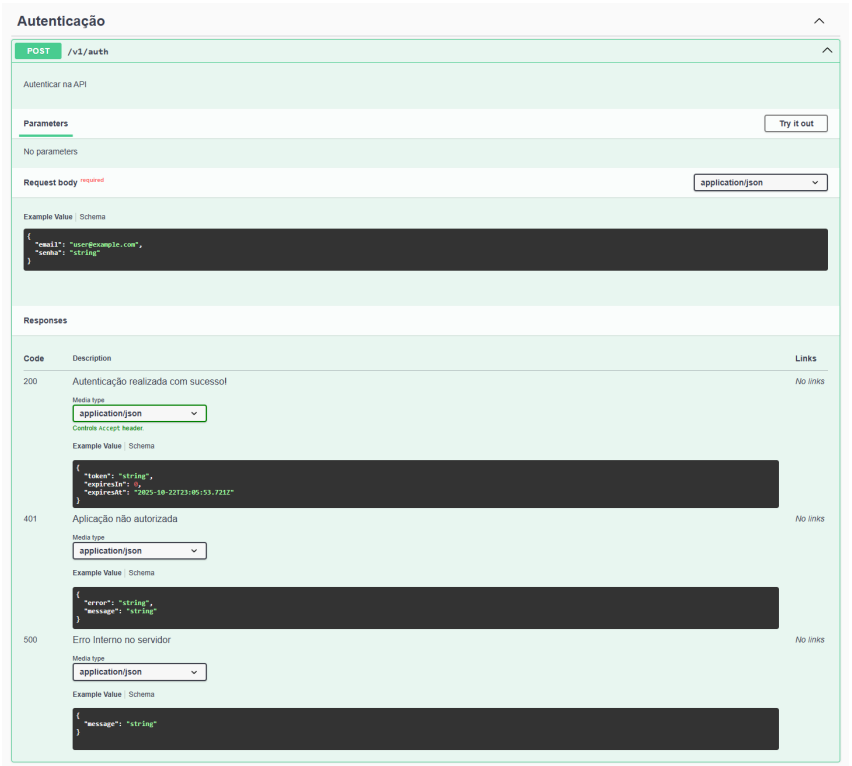
Passado estes problemas, continuou-se o desenvolvimento dos outros *endpoints* da API. Toda vez que era feito um *commit* para a *branch* principal do projeto, a *master*, o *deploy* já era feito automaticamente na *Vercel*, facilitando os testes e a validação do código previamente desenvolvido.

5.5 DOCUMENTAÇÃO

A documentação do projeto foi desenvolvida em duas etapas e de duas maneiras. A primeira ocorreu durante o desenvolvimento, por meio do uso do *Swagger*, em conjunto com os *frameworks Fastify* e *Zod*, permitindo a definição e descrição automática das rotas da aplicação. Para cada rota, foram definidos os parâmetros, o formato do corpo da requisição e os possíveis retornos. Esses *frameworks*, em conjunto, transformaram o processo de documentação em uma atividade integrada ao próprio desenvolvimento, garantindo maior consistência e atualização automática das informações da API. Com isso, gerou-se a documentação visualizada na Figura 23, onde é demonstrado um exemplo de detalhamento da rota de autenticação da API. A documentação completa da API pode ser acessada no seguinte endereço: <https://open-health-sync-api.vercel.app/docs>.

A segunda etapa foi a documentação do projeto em um nível mais técnico, detalhando a estrutura, *frameworks*, funções e como realizar a configuração do ambiente para executar a API. Para isso, criou-se um arquivo na raiz do projeto, chamado de *readme.md*, detalhando essa parte mais técnica. Junto com isso, incorporou-se também ao projeto a *collection* com todas as chamadas para a API, criadas durante o desenvolvimento para testes. Essa *collection* poderá ser importada em softwares externos utilizados para fazer requisições HTTP e testar APIs, como o caso do *postman*, software utilizado para fazer essas chamadas para o presente trabalho.

Figura 23 – Exemplo de documentação da rota de autenticação da API



Fonte: O Autor (2025).

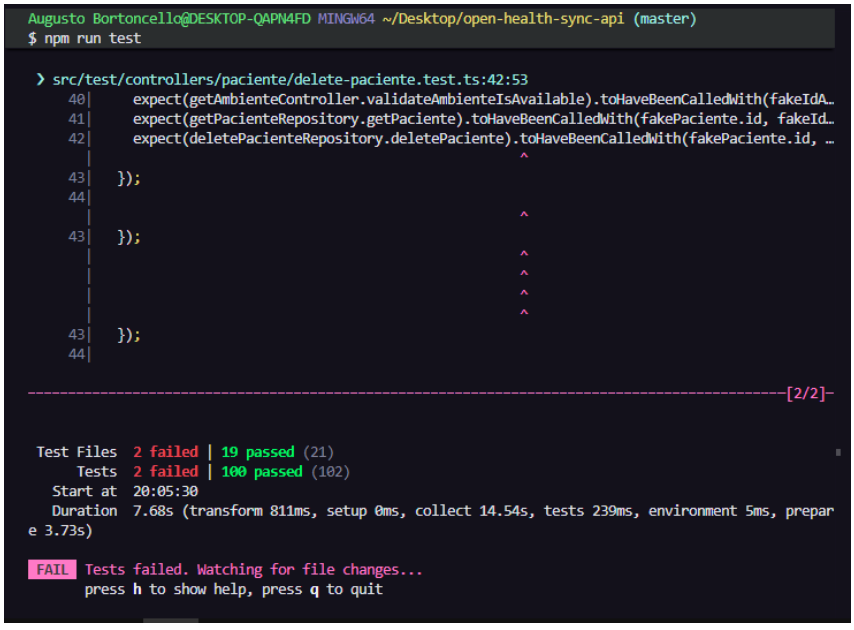
6 TESTES

Para assegurar as funcionalidades da API desenvolvida, foram realizados testes. Estes testes visam corrigir falhas encontradas durante o processo de desenvolvimento, além de garantir a confiabilidade do código. O processo de testes foi divido em duas etapas principais, uma de código, envolvendo testes unitários, e a outra envolvendo o teste dos *endpoints* da API, utilizando um software, simulando o consumo de um aplicativo ou sistema.

6.1 TESTES UNITÁRIOS

Testes unitários garantem uma maior confiabilidade e resiliência do código, além de ajudar na detecção precoce de problemas. Na prática, um teste unitário engloba forçar entradas em uma parte isolada do código, e comparar com saídas pré-definidas. Para o projeto, utilizou-se como dependência de desenvolvimento uma biblioteca responsável à fazer esse gerenciamento de entradas e saídas das funções, chamada de *vitest* e, junto com isso, criou-se um *script* no *package.json* chamado de "*test*"para executar estes testes unitários.

Figura 24 – Exemplo de execução dos testes unitários com sucessos e falhas



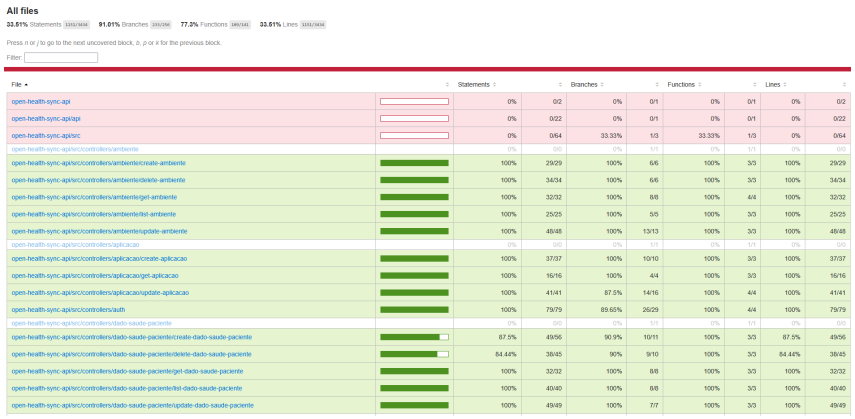
Fonte: O Autor (2025).

Foram feitos testes unitários para cada função de cada *controller* da API, cobrindo 100% dos comportamentos de cada função. A Figura 25 mostra um exemplo de quando algum teste unitário falha, executando o comando previamente citado.

Outra opção incorporada ao projeto, envolvendo os testes unitários é a execução de testes no modo *coverage*, uma forma de medir a cobertura de código, ou seja, qual a porcentagem

do código-fonte foi executado e testado pelos testes. Pra isso, utilizou-se outra dependência externa, chama de *@vitest/coverage-v8*, para a realização dessa cobertura. A Figura 25 mostra uma parte do relatório dos testes em mode de *coverage*.

Figura 25 – Exemplo de execução dos testes unitários em modo de *coverage*



Fonte: O Autor (2025).

6.2 TESTES NOS ENDPOINTS

Para realizar os testes nos *endpoints*, foi utilizado um software externo para realizar as requisições HTTP à API, chamado de *Postman*[1]. Foram configuradas todas as rotas desenvolvidas, realizando testes individuais em cada rota, para garantir que a funcionalidade principal estivesse funcionando corretamente. Além disso, foram realizados alguns testes combinando diferentes aplicações, com diferentes pacientes com diferentes dados, a fim de garantir a consistência e a segurança da API. A lista abaixo detalha todos os testes com diferentes dados realizados nos *endpoints*.

- **Autenticação**
  - Tentativas com e-mail inválido, inexistente ou associado a outra aplicação.
  - Senha incorreta.
  - *Token* inválido, expirado, gerado por e-mail anterior ou de aplicação com status inválido.
- **Paciente**
  - Operações de busca, cadastro, atualização e remoção de pacientes:
    - Em diferentes ambientes da mesma aplicação.
    - Em diferentes aplicações.

[1] <https://www.postman.com>

    - Em ambientes de outras aplicações (para validação de erros).

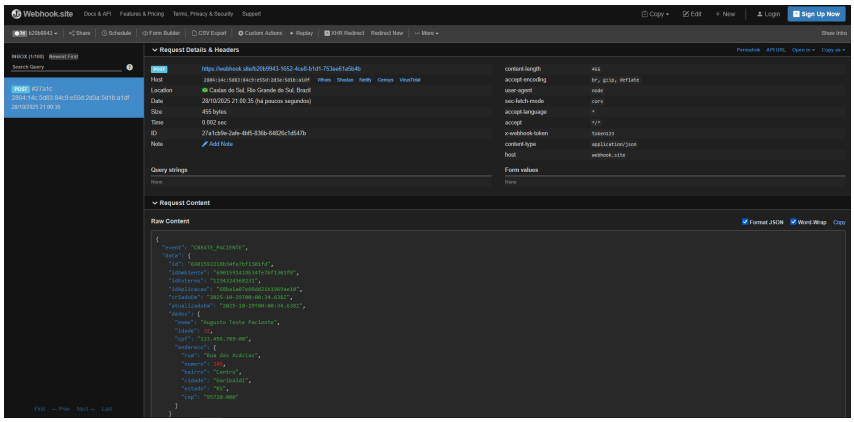
- **Registro de saúde do paciente**
  - Operações de busca, cadastro, atualização e remoção de registros:
    - Em diferentes pacientes da mesma aplicação.
    - Em diferentes aplicações.
    - Em pacientes de outras aplicações (para validação de erros).

Vale destacar que, os testes de entrada de dados inválidos nos *endpoints*, conforme a documentação, também foram realizados durante o desenvolvimento de cada *endpoint*.

6.2.1 WEBHOOK

Para a realização dos testes dos *webhooks* cadastrados em cada ambiente, foi utilizado a ferramenta *Webhook.site* [2]. Esse serviço, trabalha com uma URL exclusiva que permite a recepção e a visualização dos dados enviados pelas requisições, permitindo a verificação do funcionamento e da estrutura das mensagens transmitidas pela API. Essa URL foi utilizada nos ambientes cadastrados para a realização dos testes. A Figura 26 exemplifica como é exibido os dados enviados da API para o *webhook* cadastrado, utilizando o serviço deste site, para um evento de castro de um novo paciente.

Figura 26 – Exemplo de recepção de dados enviado da API para o *webhook*



Fonte: O Autor (2025).

[2] <https://webhook.site>

7 CONSIDERAÇÕES FINAIS

Com base nos estudos realizados e na solução desenvolvida, pode-se concluir que é viável e possível realizar a interoperabilidade de dados da saúde de diferentes estruturas, entre diferentes origens, de maneira *open source* e em conformidade com a LGPD. O código-fonte do projeto está disponível publicamente em: <https://github.com/AugustoBortoncello3547/open-health-sync-api>.

Este trabalho teve como principal objetivo, projetar e desenvolver uma API *open source* para realizar a interoperabilidade dos dados saúde, sem impor um formato específico para eles, assegurando, ao mesmo tempo, segurança e escalabilidade. Para o desenvolvimento da solução, foi necessário fundamentar conceitos que envolvem o tema de integração entre sistemas, arquitetura e engenharia de software, além de um estudo sobre principais tecnologias utilizados no momento deste trabalho. A partir dessa fundamentação, foi proposta uma solução, definindo a arquitetura e as tecnologias utilizadas, além de definir os principais requisitos e diagramas necessários para a implementação.

Com a solução proposta, junto com seu planejamento, iniciou-se a etapa de desenvolvimento, começando pela configuração do ambiente de desenvolvimento, configurando os primeiros *frameworks* para organização de código e para iniciar o desenvolvimento da API. Em seguida, começou-se o desenvolvimentos dos primeiros *endpoints* da interface, como o de aplicação e autenticação, para a segurar e testar o comportamento as tecnologias escolhidas, visto que, esses primeiros *endpoints* já englobavam todos os *frameworks* e tecnologias escolhidos. Após isso, foi feito o primeiro *deploy* da aplicação para a nuvem, já disponibilizando estes primeiros *endpoints* para testes. Por fim, finalizou-se o restante das rotas, criando a documentação do projeto e os testes unitários.

Contudo, a alta flexibilidade dos dados, desempenhada pela ausência de um padrão, e a adoção de criptografia em nível de campo no banco de dados também possuem algumas desvantagens, exigindo um equilíbrio entre segurança, desempenho e capacidade de consulta.

A ausência de um padrão dificulta a manipulação dos dados pela API, pois diferentes sistemas podem enviar informações em estruturas heterogêneas. Com isso, a API acaba realizando operações amplas, como substituir ou remover todo o conjunto de dados recebidos, em vez de executar manipulações mais específicas e granulares. A criptografia a nível de campo no banco de dados proporciona um nível mais elevado de segurança aos dados. Contudo, essa abordagem impõe limitações importantes, especialmente em consultas, ordenações e indexações, já que os valores criptografados não podem ser interpretados diretamente pelo banco. Por esse motivo, as buscas passam a depender de informações não criptografadas, como as datas de criação ou atualização dos registros, ou ainda o "*idExterno*", que permitem identificar e recuperar

os dados de forma eficiente mesmo com a criptografia aplicada.

Ademais, o conjunto da arquitetura proposta, em conjunto com as tecnologias escolhidas, se mostraram adequadas e eficientes para o contexto do projeto, oferecendo usabilidade, flexibilidade e capacidade de adaptação às necessidades da solução desenvolvida. Esse conjunto de componentes permite que a API seja facilmente estendida para outras necessidades dentro do contexto de integração de dados em saúde. Isso inclui, por exemplo, a adição de novas entidades, a inclusão de formatos diferentes de entrada de dados, a integração com outros padrões de interoperabilidade ou a implementação de novas regras de transformação e validação conforme exigências específicas de cada sistema de origem ou destino. Além disso, a plataforma utilizada para disponibilizar a API para testes, a *Vercel*, atendeu aos requisitos de disponibilidade e desempenho, embora tenha demandado ajustes adicionais durante a configuração inicial e o primeiro processo de implantação.

7.1 TRABALHOS FUTUROS

No decorrer deste trabalho, ficaram evidenciados alguns pontos que poderiam ser implementados no futuro, como melhoria e continuidade do projeto, sendo eles:

- **Cobrança da API:** o modelo proposto para a aplicação, é de um SaaS, então para isso, precisa-se de uma parte que faça a cobrança e gerencie os pagamentos das aplicações que irão utilizar a API. Não foi o objetivo deste trabalho realizar esta parte, mas sua estrutura foi pensada para facilitar esta etapa também, visto que uma parte da fundamentação do trabalho, foi realizar um estudo prévio sobre plataformas que poderiam hospedar a API e uma sugestão de valores a serem cobrados para cada uma.

- **Lógica para retentivas no *webhook*:** atualmente, quando irá se disparar um evento para o *webhook* cadastrado no ambiente, não é feito nenhuma tratativa caso dê algum erro no envio, bem como nenhuma retentiva. Dessa forma, caso o destino estiver inacessível ou ocorra algum erro de comunicação, o evento será perdido. A implementação dessa política reforçaria a confiabilidade do sistema, além de garantir que as notificações sejam entregues diante de instabilidades temporárias.

- **Mecanismos avançados de busca nos *endpoints* de listagem:** atualmente, as operações de consulta são limitadas a filtros básicos, o que restringe a eficiência na recuperação de informações em cenários com grandes volumes de dados. Uma melhoria relevante seria a implementação de mecanismos de busca mais robustos, como filtragem por múltiplos parâmetros, categorização por *tags*, buscas textuais e paginação otimizada. Esses recursos ampliariam a flexibilidade da API e ofereceriam melhor desempenho e precisão nas consultas realizadas pelos sistemas integrados.

REFERÊNCIAS

AGHA, L. The effects of health information technology on the costs and quality of medical care. **Journal of health economics**, Elsevier, v. 34, p. 19–30, 2014.

ALEXANDER, C. **A pattern language: towns, buildings, construction**. [S.l.]: Oxford university press, 1977.

ALUR, D.; CRUPI, J.; MALKS, D. **Core J2EE patterns: best practices and design strategies**. [S.l.]: Gulf Professional Publishing, 2003.

BATES, D. W.; GAWANDE, A. A. Improving safety with information technology. **New England journal of medicine**, Mass Medical Soc, v. 348, n. 25, p. 2526–2534, 2003.

BIEHL, M. **Webhooks–Events for RESTful APIs**. [S.l.]: API-University Press, 2017. v. 4.

BNDES. **Lei Geral de Proteção de Dados (LGPD)**. 2025. <https://www.bndes.gov.br/wps/portal/site/home/transparencia/lgpd> [Acesso em: 18 de abr. de 2025].

BRASIL, C. N. do. **LEI Nº 13.709, DE 14 DE AGOSTO DE 2018**. 2018. <https://www.planalto.gov.br/ccivil_03/_ato2015-2018/2018/lei/l13709.htm> [Acesso em: 18 de abr. de 2025].

CASAS, S. *et al.* Uses and applications of the openapi/swagger specification: a systematic mapping of the literature. In: IEEE. **2021 40th International Conference of the Chilean Computer Science Society (SCCC)**. [S.l.], 2021. p. 1–8.

CHACON, S.; STRAUB, B. **Pro git**. [S.l.]: Springer Nature, 2014.

COELHO, F. B.; PEDRON, C. D. Adoção de sistemas de gestão de dados de saúde pessoal: Uma revisão sistemática da literatura. 2024.

COSTA[1], K. C.; ORLOVSKI, R. A importância da utilização do software na área da saúde. 2011.

COULOURIS, G. *et al.* **Sistemas Distribuídos-: Conceitos e Projeto**. [S.l.]: Bookman Editora, 2013.

DATE, C. J. **Introdução a sistemas de bancos de dados**. [S.l.]: Elsevier Brasil, 2004.

Docker Inc. **Use containers to Build, Share and Run your applications**. 2025. Acesso em: 07 de out. de 2025. Disponível em: <https://www.docker.com/resources/what-container/>.

DRUMMOND, A. C. **Avaliação de Desempenho de Contêineres Docker para Aplicações do Supremo Tribunal Federal**. Tese (Doutorado) — Universidade de Brasília, 2017.

ELKOURDI, F. *et al.* Exploring current practices and challenges of hipaa compliance in software engineering: Scoping review. **IEEE Open Journal of Systems Engineering**, IEEE, 2024.

FIELDING, R. T. **Architectural styles and the design of network-based software architectures**. [S.l.]: University of California, Irvine, 2000.

FLANAGAN, D. **JavaScript: o guia definitivo**. [S.l.]: Bookman Editora, 2012.

FOWLER, M. **Inversion of Control Containers and the Dependency Injection pattern**. 2004. Acesso em: 27 de abr. de 2025. Disponível em: <https://martinfowler.com/articles/injection.html>.

____. **Fluent Interface**. 2005. <https://martinfowler.com/bliki/FluentInterface.html> [Acesso em: 31 de mar. de 2025.].

____. **Padrões de arquitetura de aplicações corporativas**. [S.l.]: Bookman, 2009.

FREITAS, D. S. de *et al.* Devsecops practices for gdpr, hipaa or lgpd compliance in software development: A systematic review. **Simpósio Brasileiro de Sistemas de Informação (SBSI)**, SBC, p. 145–153, 2025.

Google. **Review the platform architecture**. 2023. <https://developer.android.com/health-and-fitness/guides/health-connect/plan/architecture?hl=pt-br>. Acesso em: 30 mar. 2025.

____. **Health Connect: simplify connectivity between apps**. 2024. <https://developer.android.com/health-and-fitness/guides/health-connect?hl=pt-br>. Acesso em: 30 mar. 2025.

____. **Review Health Connect functionality**. 2025. <https://developer.android.com/health-and-fitness/guides/health-connect/plan/developer-functionality?hl=pt-br>. Acesso em: 30 mar. 2025.

GOTARDO, R. A. Linguagem de programação. **Rio de Janeiro: Seses**, v. 60, 2015.

GUEDES, G. T. **UML 2-Uma abordagem prática**. [S.l.]: Novatec Editora, 2018.

Hapi FHIR. **A Free and Open Source Global Good: Powering Interoperability Around the World for 23 Years.** 2025. <https://hapifhir.io/>. Acesso em: 30 mar. 2025.

HAPI FHIR. **Client Introduction**. 2025. <https://hapifhir.io/hapi-fhir/docs/client/introduction.html>. Acesso em: 30 mar. 2025.

____. **Client Pointcuts**. 2025. <https://hapifhir.io/hapi-fhir/docs/interceptors/client_pointcuts.html>. Acesso em: 30 mar. 2025.

____. **HAPI FHIR Server Introduction**. 2025. <https://hapifhir.io/hapi-fhir/docs/server_plain/server_types.html>. Acesso em: 30 mar. 2025.

____. **Plain Server Pointcuts - Storage / JPA Server Pointcuts**. 2025. <https://hapifhir.io/hapi-fhir/docs/interceptors/server_pointcuts.html>. Acesso em: 30 mar. 2025.

____. **REST Server Security: Introduction**. 2025. <https://hapifhir.io/hapi-fhir/docs/security/introduction.html>. Acesso em: 30 mar. 2025.

____. **Validation Introduction**. 2025. <https://hapifhir.io/hapi-fhir/docs/validation/introduction.html>. Acesso em: 30 mar. 2025.

HealthSync. **HealthSync**. s.d. <https://www.healthsync.com/>. Acesso em: 30 mar. 2025.

____. **Why did I create this app, and why does the world need another app?** s.d. <https://www.healthsync.com/about>. Acesso em: 30 mar. 2025.

HOSSEINI, M. M. *et al.* Smartwatches in healthcare medicine: assistance and monitoring; a scoping review. **BMC Medical Informatics and Decision Making**, Springer, v. 23, n. 1, p. 248, 2023.

JOHNSON, R. E.; FOOTE, B. Designing reusable classes. **Journal of object-oriented programming**, v. 1, n. 2, p. 22–35, 1988.

LI, H.; YU, L.; HE, W. **The impact of GDPR on global technology development**. [S.l.]: Taylor & Francis, 2019. 1–6 p.

MARCIANO, J. L.; LIMA-MARQUES, M. O enfoque social da segurança da informação. **Ciência da Informação**, SciELO Brasil, v. 35, p. 89–98, 2006.

MASSE, M. **REST API design rulebook: designing consistent RESTful web service interfaces**. [S.l.]: "O'Reilly Media, Inc.", 2011.

Microsoft Corporation. **TypeScript para o Novo Programador**. 2025. <https://www.typescriptlang.org/pt/docs/handbook/typescript-from-scratch.html>. Acessado em: 25 maio 2025.

MongoDB Inc. **Introdução ao MongoDB e Mongoose**. 2024. <https://www.mongodb.com/pt-br/developer/languages/javascript/getting-started-with-mongodb-and-mongoose/>. Publicado em: 07 abr. 2022. Atualizado em: 05 ago. 2024. Acessado em: 25 maio 2025.

MORENO, R. A. Interoperabilidade de sistemas de informação em saúde. **Journal of Health Informatics**, v. 8, n. 3, 2016.

MOURA, L. R.; PESSÔA, M. Gestão integrada da informação: proposição de um modelo de organização baseado no uso da informação como recurso da gestão empresarial. 1999.

OpenJS Foudation. **Introduction to Node.js**. 2025. Acesso em: 22 maio 2025. Disponível em: <https://nodejs.org/pt/learn/getting-started/introduction-to-nodejs>.

____. **An introduction to the npm package manager**. 2025. Acesso em: 22 maio 2025. Disponível em: <https://nodejs.org/pt/learn/getting-started/an-introduction-to-the-npm-package-manager>.

OpenJS Foundation and The Fastify Team. **Fastify: Fast and Low Overhead Web Framework**. 2025. <https://fastify.dev/>. Acessado em: 25 maio 2025.

Opensource.org. **MIT License**. 2025. <https://opensource.org/licenses/MIT>. Acessado em: 11 junho 2025.

ORACLE. **O que é master data management (MDM, Gerenciamento de Dados Mestres)?** 2025. <https://www.oracle.com/br/scm/product-lifecycle-management/master-data-management/> [Acesso em: 31 de mar. de 2025.].

PELLIZZONI, L.; FALAVIGNA, A. Connecting verified databases with clinical practice and the patient's experience through omnichannel communication. **International Journal of Medical Informatics**, Elsevier, v. 192, p. 105639, 2024.

PEREIRA, C. R. **Aplicações web real-time com Node. js**. [S.l.]: Editora Casa do Código, 2014.

PRESSMAN, R. S. **Software engineering: a practitioner's approach**. [S.l.]: Palgrave macmillan, 2005.

PRICE, W.; NICHOLSON, I. Risk and resilience in health data infrastructure. **Colo. Tech. LJ**, HeinOnline, v. 16, p. 65, 2017.

RICHARDS, M.; FORD, N. **Fundamentals of Software Architecture: An Engineering Approach**. 1. ed. Estados Unidos: O'Reilly Media, 2020. ISBN 1492043451, 9781492043454.

RNDS. **Ambientes**. s.d. <https://rnds-guia.saude.gov.br/docs/rnds/ambientes>. Acesso em: 30 mar. 2025.

____. **Gestor**. s.d. <https://rnds-guia.saude.gov.br/docs/publico-alvo/gestor/gestor>. Acesso em: 30 mar. 2025.

____. **Integrador**. s.d. <https://rnds-guia.saude.gov.br/docs/publico-alvo/ti/ti>. Acesso em: 30 mar. 2025.

____. **Introdução**. s.d. <https://rnds-guia.saude.gov.br/docs/introducao/>. Acesso em: 30 mar. 2025.

____. **Passo a passo**. s.d. <https://rnds-guia.saude.gov.br/docs/passo-a-passo>. Acesso em: 30 mar. 2025.

____. **Público-alvo**. s.d. <https://rnds-guia.saude.gov.br/docs/publico-alvo/publico-alvo>. Acesso em: 30 mar. 2025.

____. **Serviços**. s.d. <https://rnds-guia.saude.gov.br/docs/rnds/servicos>. Acesso em: 30 mar. 2025.

ROCHA, F. S. da *et al.* Uso de apps para a promoção dos cuidados à saúde. **Anais do Seminário Tecnologias Aplicadas a Educação e Saúde**, 2017.

ROSENBERG, D.; SCOTT, K. **Use case driven object modeling with UML**. [S.l.]: Springer, 1999.

ROSENBERG, D.; STEPHENS, M.; COLLINS-COPE, M. Agile development with iconix process. **New York, Editorial Apress**, Springer, 2005.

ROZANSKI, N.; WOODS, E. **Software systems architecture: working with stakeholders using viewpoints and perspectives**. [S.l.]: Addison-Wesley, 2012.

SALES, O. M. M.; PINTO, V. B. Tecnologias digitais de informação para a saúde: revisando os padrões de metadados com foco na interoperabilidade. **Revista Eletrônica de Comunicação, Informação & Inovação em Saúde**, v. 13, n. 1, 2019.

SATYANARAYANA, S. Cloud computing: Saas. **Computer Sciences and Telecommunications**, - , n. 4, p. 76–79, 2012.

SILVA, R. B.; COVAC, J. R. O que é compliance. **Conceitos & Ferramentas na Visão de um Auditor Interno. Editora: Albatroz, Rio de Janeiro**, 2018.

SmartBear Software. **Swagger**. 2025. Acesso em: 27 de abr. de 2025. Disponível em: <https://swagger.io/>.

Smile Digital Health. **About Smile Digital Health**. s.d. <https://www.smiledigitalhealth.com/about-us>. Acesso em: 30 mar. 2025.

SOCIAL, F. e. C. F. M. Ministério do Desenvolvimento e A. **Lei Geral de Proteção de Dados Pessoais (LGPD)**. 2025. <https://www.gov.br/mds/pt-br/acesso-a-informacao/governanca/integridade/campanhas/lgpd> [Acesso em: 18 de abr. de 2025].

SOMMERVILLE, I. Engenharia de software, 9a. **São Palo, SP, Brasil**, p. 63, 2011.

STEEN, M. van; TANENBAUM, A. S. **Distributed Systems: Principles and Paradigms**. 4. ed. [S.l.]: Maarten van Steen, 2023. ISBN 9081540637, 9789081540636.

STRAUCH, C.; SITES, U.-L. S.; KRIHA, W. Nosql databases. **Lecture Notes, Stuttgart Media University**, v. 20, n. 24, p. 79, 2011.

TORAB-MIANDOAB, A. *et al.* Interoperability of heterogeneous health information systems: a systematic literature review. **BMC medical informatics and decision making**, Springer, v. 23, n. 1, p. 18, 2023.

TRINDADE, I. *et al.* Estratégias em segurança de dados na saúde: uma revisão rápida. **Journal of Health Informatics**, v. 16, n. Especial, 2024.

WU, Z.; TRIGO, V. Impact of information system integration on the healthcare management and medical services. **International Journal of Healthcare Management**, Taylor & Francis, v. 14, n. 4, p. 1348–1356, 2021.