Universidade Católica do Salvador
Graduação Tecnóloga em Análise e Desenvolvimento de Sistemas

Alex Jordan
Arthur Vita de Souza Giovani
Ester Costa de Souza
Iury Xavier Assunção
Lucas Campos Torres

Aplicação de Clean Architecture no Desenvolvimento de um Sistema de voluntários para área tech: um estudo de caso

SALVADOR
2024

Alex Jordan
Arthur Vita de Souza Giovani
Ester Costa de Souza
Iury Xavier Assunção
Lucas Campos Torres

Aplicação de Clean Architecture no Desenvolvimento de um Sistema de voluntários para área tech: um estudo de caso

Trabalho de Conclusão de Curso apresentado à Universidade Católica do Salvador como parte dos requisitos necessários para a obtenção do Título de Tecnólogo em Análise e Desenvolvimento de Sistemas. Orientador: Profª. Angela Peixoto Santana

Universidade Católica do Salvador

SALVADOR
2024

Aplicação de Clean Architecture no Desenvolvimento de um Sistema de voluntários para área tech: um estudo de caso

Alex Jordan[1], **Arthur Vita de Souza Giovani**[1], **Ester Costa de Souza**[1] , **Iury Xavier Assunção**[1], **Lucas Campo Torres**[1], **Angela Peixoto Santana** [1]

[1]Instituto de Informática – Universidade Católica do Salvador (UCSAL)
CEP – 41740-090 – Salvador – BA – Brazil

{alex.jordan, arthur.giovani, ester.souza,

iury.assuncao, lucascampos.torres}@ucsal.edu.br,

{angela.santana}@pro.ucsal.br

Abstract. *This work explores the application of Clean Architecture in developing a volunteer management system for the tech sector, emphasizing the integration of IT professionals with non-governmental organizations (NGOs). Addressing the increasing demand for technological solutions in social causes, the study underscores the significance of adopting a scalable, flexible, and maintainable software architecture. By applying Clean Architecture principles, the research ensures a clear separation of concerns and independence between system layers, facilitating long-term adaptability and evolution. The resulting system, named Voluntaria Tech, is designed to bridge the gap between volunteers and NGOs, offering an effective solution to the shortage of skilled professionals in Brazil's tech sector. This work highlights the importance of employing best practices in software development to enhance collaboration, engagement, and the sustainability of social projects. It demonstrates the practical advantages of Clean Architecture in crafting systems that are both scalable and adaptable, showcasing its potential to meet the unique challenges of social and technological contexts.*
Key-words: Clean Architecture, Volunteer Management System, Social Impact, IT Professionals, Non-Governmental Organizations (NGOs), Scalable Software Architecture.

Resumo. *Este trabalho explora a aplicação da Clean Architecture no desenvolvimento de um sistema de gerenciamento de voluntários para o setor de tecnologia, com ênfase na integração de profissionais de TI com organizações não governamentais (ONGs). Abordando a crescente demanda por soluções tecnológicas em causas sociais, o estudo destaca a importância de adotar uma arquitetura de software escalável, flexível e de fácil manutenção. Por meio da aplicação dos princípios da Clean Architecture, a pesquisa garante uma separação clara de responsabilidades e a independência entre as camadas do sistema, facilitando sua adaptabilidade e evolução a longo prazo. O sistema desenvolvido, denominado Voluntaria Tech, foi projetado para reduzir a lacuna entre voluntários e ONGs, oferecendo uma solução eficaz para a escassez de profissionais qualificados no setor de tecnologia no Brasil. Este trabalho enfatiza a relevância de utilizar boas práticas de desenvolvimento de software para promover a colaboração, o engajamento e a sustentabilidade de projetos sociais. Além disso, demonstra as vantagens práticas da Clean Architecture na criação de sistemas escaláveis e adaptáveis, evidenciando seu potencial para atender aos desafios únicos de contextos sociais e tecnológicos.*

Palavras-chave: Clean architecture, Sistema de Gerenciamento de Voluntários, Impacto Social, Profissionais de TI, Organizações Não Governamentais (ONGs), Arquitetura de Software Escalável.

1. INTRODUÇÃO

Considerando o crescimento de aplicações e o aumento da complexidades dos sistemas de *software* modernos, exigem que os desenvolvedores adotem boas práticas e padrões arquiteturais. Essas práticas estão a cada dia mais comum. Lidar com desafios desses sistemas de *software* como manutenções, escalabilidade, usabilidade e acima de tudo independência de tecnologias não é uma tarefa fácil. Nesse contexto, a *Clean Architecture*, proposta por [Martin 2017], emergiu como uma abordagem estruturada para organizar e apoiar o desenvolvimento de *software*, de modo que o mesmo se torne robusto e adaptável às inúmeras mudanças do longo do tempo.

A *Clean Architecture* preconiza a separação clara de responsabilidades, priorizando a independência entre camadas e promovendo o isolamento dos casos de uso centrais em relação às decisões tecnológicas. Essa estrutura não apenas facilita a evolução do sistema, mas também reduz o acoplamento e favorece o reuso de código. De acordo com [Martin 2017], um dos objetivos principais dessa abordagem é garantir que ”os detalhes sejam completamente separados das regras de negócios”, permitindo que decisões sobre *frameworks*, bancos de dados e *interfaces* de usuário possam ser facilmente substituídas ou modificadas sem impactar o núcleo do sistema.

Além disso, estudos recentes reforçam a relevância de arquiteturas limpas no cenário atual. Alguns estudos apontam que uma má definição arquitetural é uma das principais causas de dívidas técnicas que afetam a qualidade do *software*. Já [Fowler 2018] destaca que boas práticas de codificação auxiliam no gerenciamento de complexidade e na entrega de *software* sustentável e de alta qualidade.

No decorrer desta pesquisa, identificou-se a necessidade de um desenvolvimento de *software* para conectar de forma eficiente os profissionais da área de tecnologia com organizações sociais, facilitando o engajamento e a colaboração em projetos de impacto social. A crescente demanda por soluções tecnológicas voltadas para causas sociais exige ferramentas que permitam a integração entre esses dois mundos, promovendo uma interação mais eficaz, escalável e sustentável ao longo do tempo.

Portanto, este trabalho tem por objetivo geral a especificação e o desenvolvimento de um *software* que possibilite a aplicação do *Clean Architecture*. Cabe ressaltar que, para o seu fim, este *software* será estruturado de maneira a garantir a independência entre suas camadas, promovendo uma arquitetura escalável, flexível e de fácil manutenção. O foco será criar uma solução que não apenas atenda aos requisitos técnicos, mas que também proporcione uma plataforma eficiente para o engajamento de profissionais de tecnologia com causas sociais, facilitando a colaboração e o impacto positivo na sociedade.

Este projeto contribuirá significativamente para a área da engenharia de *software*, servindo como um exemplo prático da aplicação de boas práticas arquiteturais, com destaque para a *Clean Architecture*. Ele demonstrará como essa abordagem pode aprimorar a organização, escalabilidade e manutenção de sistemas complexos. Além disso, ao direcionar seus esforços para o desenvolvimento de soluções voltadas ao voluntariado tecnológico, o projeto oferece uma perspectiva

inovadora sobre o uso da tecnologia para promover impacto social, incentivando a colaboração entre profissionais de TI e organizações sociais.

Além dessa introdução, o artigo está organizado da seguinte forma: a Seção 2 apresenta a revisão da literatura; a seção 3 apresenta o desenvolvimento do sistemas e a seção 4 aponta a conclusão e trabalhos futuros.

2. FUNDAMENTAÇÃO TEÓRICA

Esta seção apresenta uma breve fundamentação teórica sobre os aspectos que envolvem os conceitos de *clean architecture*, fornecendo subsídios para a proposta de desenvolvimento de um estudo de caso baseado nos princípios da *clean architecture*.

2.1. Arquitetura de *Software*

Segundo [Booch et al. 1994] determina que Arquitetura de *software* é uma estrutura fundamental para o desenvolvimento de sistemas. Essa estrutura é composta por elementos de *software*, propriedades externamente visíveis e o relacionamento entre eles.

A arquitetura de *software* é essencial para o desenvolvimento de sistemas de alta qualidade e complexidade. Sua principal função é garantir que o *software* seja organizado de forma eficiente, permitindo escalabilidade e facilitando a manutenção. Conforme [Bass and Paul Clements 2013], a arquitetura define a estrutura do sistema, especificando como seus componentes, módulos e serviços interagem, o que torna o sistema mais compreensível e fácil de definir.

Entre os principais benefícios de uma boa arquitetura de *software*, conforme elencados por [Norman 2013] e [Bass and Paul Clements 2013] destacam-se:

- **Organização**: Ela permite que os sistemas sejam organizados de forma clara, ajudando desenvolvedores a construir e manter o *software* de maneira eficiente.
- **Manutenção facilitada**: Uma arquitetura sólida possibilita que mudanças e expansões sejam realizadas com menos impacto em outras partes do sistema.
- **Escalabilidade**: Arquiteturas bem projetadas permitem que o sistema evolua em usuários e volume de dados sem comprometer o desempenho.
- **Atendimento aos requisitos de negócio**: Uma boa arquitetura garante que o *software* esteja alinhado com as necessidades de negócios, como segurança, desempenho e confiabilidade.

2.2. A origem da Arquitetura de *Software*

A noção de arquitetura de *software* começou a se desenvolver nas décadas de 1960 e 1970, quando os sistemas de *software* se tornaram mais complexos. À medida que os *softwares* cresciam em tamanho e complexidade, percebeu-se a necessidade de uma organização formal para garantir que sistemas distribuídos funcionassem de maneira eficiente e coordenada [Bass and Paul Clements 2013].

- **Décadas de 1960 e 1970**: Foi nesse período que os conceitos de engenharia de *software* começaram a ganhar força, à medida que os sistemas se tornavam maiores e mais complexos. Surgiu a necessidade de padronizar o desenvolvimento de *software*, facilitando sua criação e manutenção [Bass and Paul Clements 2013].

- **Motivo para o surgimento**: Conforme [Norman 2013], o crescimento da complexidade dos sistemas impulsionou o desenvolvimento da arquitetura de *software*, permitindo gerenciar grandes sistemas de forma organizada e eficiente, além de facilitar a reutilização de componentes e padrões de projetos.

2.2.1. Conexão com a Usabilidade

Quando se aborda a arquitetura de *software*, é essencial discutir usabilidade, que desempenha um papel crucial no sucesso de qualquer sistema. Segundo [Bass and Paul Clements 2013], a usabilidade está diretamente relacionada à forma como o usuário interage com o *software* e ao quão bem ele consegue realizar suas tarefas com facilidade e satisfação.

De acordo com [Norman 2013] e [Bass and Paul Clements 2013], a usabilidade impacta diretamente:

- **Eficiência**: Sistemas fáceis de usar permitem que os usuários realizem suas tarefas de maneira rápida e precisa.
- **Satisfação do usuário**: Uma experiência de uso positiva está intimamente ligada à satisfação, sendo um fator decisivo para a aceitação de qualquer sistema.
- **Redução de erros**: *Interfaces* bem desenhadas diminuem a ocorrência de erros, aumentando a confiabilidade do sistema.
- **Competitividade no mercado**: A usabilidade pode ser um diferencial competitivo, especialmente em mercados com múltiplas opções de *software*. Um sistema fácil de usar tem maiores chances de sucesso.

2.2.2. Arquitetura de *Software* e Usabilidade

A arquitetura de *software* e a usabilidade estão diretamente conectadas. A maneira como o sistema é arquitetado influencia diretamente a experiência do usuário. Conforme [Bass and Paul Clements 2013], uma boa arquitetura facilita o desenvolvimento de *interfaces* eficientes e responsivas, enquanto uma arquitetura deficiente pode prejudicar a usabilidade.

- **Desempenho**: A arquitetura afeta o desempenho do sistema, o que, por sua vez, impacta a experiência do usuário. Sistemas lentos ou com respostas inadequadas podem frustrar o usuário, mesmo que a *interface* seja intuitiva [Norman 2013].
- **Flexibilidade para mudanças**: Arquiteturas modulares permitem que a *interface* do sistema seja adaptada para melhorar continuamente a experiência do usuário, com base em *feedbacks* e novas necessidades [Bass and Paul Clements 2013].
- **Manutenção e evolução**: Segundo [Bass and Paul Clements 2013], quando a arquitetura é bem planejada, o sistema pode ser facilmente atualizado, permitindo a introdução de melhorias que otimizam a usabilidade ao longo do tempo.

2.3. *Clean Architecture*

Criada por [Martin 2003], a Arquitetura Limpa é composta por dois elementos principais: as políticas e os detalhes. As políticas referem-se às regras de negócio, que estão localizadas nos círculos mais internos da arquitetura. Já os detalhes correspondem aos elementos necessários para

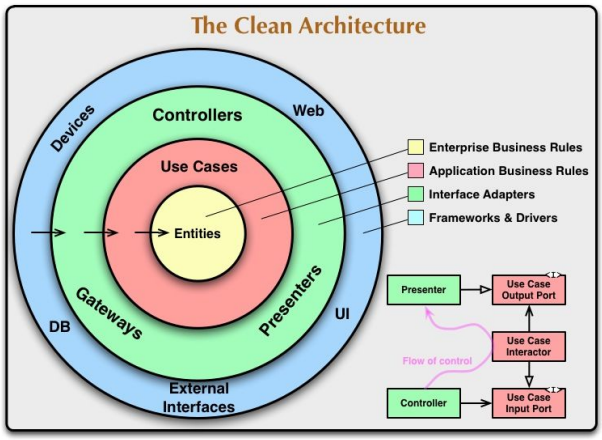
a implementação dessas políticas, mas encontram-se posicionados nos níveis mais externos do sistema.

De acordo com [Martin 2017] a arquitetura de *software* desempenha um papel fundamental no desenvolvimento de sistemas. Quando adequadamente estruturada, ela facilita e agiliza todo o processo de programação dentro da equipe de desenvolvimento. Independentemente das tecnologias ou linguagens de programação utilizadas, é possível estabelecer padrões que tornam o processo mais eficiente e organizado, promovendo a qualidade e a manutenibilidade do código [Ivanics 2016].

Com a prioridade na organização da arquitetura do *software*, focando em princípios como a reusabilidade e a regra da dependência. Embora sejam distintas umas das outras, todas as arquiteturas possuem um intuito: a separação das preocupações. De acordo com [Martin 2017] a segregação é feita por meio de camadas. Seguindo a estrutura da palavra *software* , "produto suave" indica em sua origem que deve ser de fácil mudança [Ivanics 2016].

A figura 1, representa o diagrama do *clean architecture*, conforme apresenta em [Martin 2017].

Figura 1. Estrutura Básica de uma aplicação *Clean Architecture*



Autor: [Martin 2017]

Conforme pode ser visualizado, os círculos indicam diferentes areas do software, quanto mais externo o círculo mais baixo nível, e quanto mais interno o círculo mais alto nível. As regras de negócio se dispõem nos círculos internos e precisam ser protegidos de alterações externas, com isso o mesmo pontua a regra de dependência [Ferreira et al. 2022].

Outra característica fundamental da Arquitetura Limpa é a aplicação dos princípios do SOLID, como o Princípio da Responsabilidade Única. Além disso, a regra da inversão de dependências é um conceito central dessa arquitetura. De acordo com essa regra, os elementos localizados nos círculos externos não devem ser referenciados ou dependentes do código presente nos círculos internos [Martin 2017]. Os princípios do SOLID estão detalhados na seção. 2.4.

[Martin 2003] exemplifica no livro Arquitetura Limpa esse processo em quatro camadas. Sendo a primeira para simbolizar as entidades, que descreve em seu exemplo como regra de

negócios da empresa. Na segunda camada mais externa vem o caso de uso. Nela estão contidas as regras de negócios da aplicação que orienta o fluxo de dados e posiciona as entidades nas aplicações das regras.

Uma camada mais externa é a de controladores. Os *softwares* da camada de adaptadores de *interface* basicamente converte os dados no formato mais apropriado para os casos de usos e entidades [Martin 2017].

Ademais a camada mais externa no exemplo fornecido por [Martin 2003] é a de *frameworks* e *drivers*, cujo os detalhes devem ficar localizados nessa camada para evitar que possam causar algum dano na sua aplicação.

2.4. Princípios SOLID

Os primeiros indícios dos princípios SOLID apareceram por volta de 1995, através do artigo "*The Principles of Object-Oriented Design"* de Robert C. Martin, conhecido popularmente como *"Uncle Bob"*. Com um crescente interesse em compreender esses temas, Martin publicou em 2002 o livro *"Agile Software Development, Principles, Patterns, and Practices"*. Contudo, a sigla SOLID foi introduzida posteriormente por Michael Feathers [Feathers 2004].

A *Clean Architecture* fundamenta-se nos princípios SOLID, que orientam a organização de funções e dados em classes, bem como a forma como essas classes devem interagir. De acordo com [Martin 2019], os principais objetivos dessa abordagem incluem a facilidade de adaptação a mudanças, a clareza para a equipe de desenvolvimento e a padronização de componentes para aplicação em diferentes sistemas.

Por outro lado, conforme argumenta [Érico Padilha Júnior 2012], é crucial ter conhecimento para aplicar os princípios do SOLID corretamente. Ele ressalta que esses princípios devem ser utilizados com julgamento, pois, caso contrário, sua aplicação pode ser tão prejudicial quanto a ausência de sua utilização. Os cinco princípios que compõem o SOLID são essenciais para o gerenciamento de dependências na programação orientada a objetos. Entretanto, quando esse gerenciamento é realizado de forma inadequada, pode resultar em código rígido, frágil e de difícil reuso. Portanto, a aplicação correta dos princípios é fundamental para o sucesso de um projeto de *software*.

O acrônimo SOLID representa cinco princípios desse conjunto, que são descritos a seguir:

1. **Single Responsibility Principle (SRP) - Princípio da Responsabilidade Única:**
   O Princípio da Responsabilidade Única determina que uma classe deve ser encarregada de apenas uma tarefa ou função, devendo conter métodos específicos e bem definidos que atendam a uma única necessidade funcional [Chebanyuk and Markov 2016]. Em Pressman [PRESSMAN 2011], enfatiza-se a importância da clareza nos objetivos das unidades de *software* para assegurar coesão e minimizar o esforço necessário para a manutenção e expansão do sistema.
2. **Open/Closed Principle (OCP) - Princípio Aberto/Fechado:**
   Este princípio sugere que as classes devem ser desenvolvidas de tal forma que novas funcionalidades possam ser adicionadas sem a necessidade de alterar o código existente. Ou seja, elas devem ser extensíveis sem modificação de suas implementações originais [Chebanyuk and Markov 2016] e [Martin 2020] . Essa abordagem facilita a

evolução do software, permitindo que ele se adapte a novas demandas sem comprometer a estabilidade do código existente, reduzindo o risco de introduzir novos erros [Fernández-Ropero et al. 2015]. De acordo com [PRESSMAN 2011], sistemas adaptáveis são mais capazes de responder a mudanças nos requisitos, um aspecto crucial em ambientes de desenvolvimento ágil.

3. **Princípio da Substituição de Liskov:**
   Conforme descrito por Robert C. Martin [Martin 2003], afirma que objetos de uma classe base devem ser substituíveis por objetos de suas subclasses sem alterar o comportamento esperado do programa. Isso significa que as subclasses devem manter as expectativas e o comportamento definidos pela classe base, garantindo que a lógica do sistema permaneça intacta. Se uma subclasse não respeitar essas condições, pode introduzir erros ou comportamentos inesperados, comprometendo a integridade do software. Em essência, o LSP assegura que as hierarquias de classes sejam coerentes e que a extensibilidade do código não resulte em falhas no funcionamento do sistema [Martin 2019].
4. **Interface Segregation Principle (ISP) - Princípio da Segregação de Interfaces:**
   De acordo com Martin [Martin 2019], esse princípio determina que, ao herdar de outra classe, os objetos da classe pai devem ser substituíveis por objetos da classe filha sem afetar o comportamento esperado do sistema. Em essência, uma subclasse deve poder substituir a classe pai sem provocar comportamentos inesperados ou falhas. Isso assegura que o sistema continue a operar corretamente, mesmo com a introdução de novas classes. Essa abordagem promove a consistência e previsibilidade no código, permitindo que as subclasses sejam tratadas de maneira intercambiável com a classe pai, sem comprometer a integridade do sistema.
5. **Dependency Inversion Principle (DIP) - Princípio da Inversão de Dependência:**
   O Princípio da Inversão de Dependência sugere que as dependências no código devem ser direcionadas a abstrações, e não a implementações concretas [Martin, 2019]. Isso facilita a realização de modificações e a expansão do código de maneira mais simples e eficiente. Segundo Pressman (2011), a redução do acoplamento é uma das metas fundamentais da engenharia de software, pois melhora a adaptabilidade e escalabilidade do sistema, tornando-o mais flexível e preparado para futuras mudanças.

2.4.1. Vantagens do SOLID

Os princípios SOLID são amplamente utilizados no desenvolvimento de *software* pelos seguintes motivos:

- Facilita a manutenção e os testes do código [Martin 2017].
- Ajuda a evitar duplicações e torna a extensão do sistema mais simples [Martin 2003].
- Promove o desacoplamento entre módulos, aumentando a flexibilidade [Martin 2017].
- Melhora a modularidade, permitindo que componentes sejam reutilizados em outros contextos [Fernández-Ropero et al. 2015].

Esses princípios são especialmente valiosos em ambientes de desenvolvimento ágil, onde os requisitos estão em constante mudança e o *software* precisa ser flexível e facilmente adaptável [Martin 2003].

2.5. Node.Js

Em [Dahl 2009] encontramos um problema ao rastrear tempos de carregamento em navegadores usando técnicas como solicitações AJAX que sobrecarregavam o servidor. Ao tentar implementar servidores em linguagens como *Ruby, Lua* e *Haskell, Dahl* percebeu que, por serem de natureza paralela, eram mais memoráveis e utilizáveis. Seu objetivo é encontrar uma linguagem que simplifique a programação sem sobrecarregar a CPU com operações de E/S (*input/output* demoradas.

É aqui que o *JavaScript* chama a atenção, proporcionando simplicidade, leveza e suporte para ações sem bloqueio. A funcionalidade, aprimorada pelo mecanismo V8 de código aberto, o convenceu a usar *JavaScript* como base para construir todo o seu servidor: *Node.js* [Dahl 2009]. *Node.js* é uma estrutura *JavaScript* de código aberto para a construção de grandes aplicativos da *web*. Em outras palavras, permite executar código *JavaScript* fora do navegador, no servidor. Esta capacidade foi estendida ao *JavaScript*, permitindo a criação de uma ampla gama de aplicações, desde servidores *web* até aplicações baseadas na *web*.

Segundo [Pereira 2016], *Node.js* é uma "plataforma web muito grande e de baixo nível" projetada para lidar com processamento paralelo usando uma arquitetura orientada a eventos e programação dinâmica. Ao contrário de linguagens como .NET, Java e PHP, que possuem uma arquitetura de bloqueio, o *Node.js* usa uma abordagem sem bloqueio, que permite realizar operações de E/S de forma eficiente, sem sobrecarregar o *mainframe*. A estrutura *Node.js* usa um único *thread* para processar todas as solicitações, o que significa que não cria um novo *thread* para cada solicitação recebida. A utilização do motor *Google Chrome*, conhecido pela velocidade de execução do código *JavaScript*, também foi destacada como um dos motivos pelos quais o *Node.js* é uma boa opção para aplicações em tempo real, como servidor de *chat* ou sistema de entrega [Pereira 2016].

Além disso, o ecossistema de código aberto do *Node.js* permite que ele seja usado em muitas áreas. De acordo com [Lewenhagen and Åkesson 2013], mais de 50% dos projetos revisados no *GitHub*[1] usam *Node.js* para construir suas aplicações. Porém, seus usos vão além disso e incluem servidores de jogos, proxies, ferramentas de automação e até sistemas operacionais. Outro ponto forte do *Node.js* é sua estrutura modular, que organiza o código em arquivos separados, fáceis de manter e usar. O gerenciador de pacotes npm (*Node Package Manager*) percorre um longo caminho, fornecendo um amplo ambiente pronto para uso, acelerando o desenvolvimento de aplicações e permitindo integração rápida e eficiente em diversas aplicações.

Resumindo, o *Node.js* contribuiu muito para o desenvolvimento do *JavaScript*, ampliando bastante sua funcionalidade. Antes de sua introdução, o *JavaScript* era usado principalmente internamente, limitado à criação de *interfaces* de usuário em navegadores. No entanto, graças ao *Node.js*, uma linguagem que ganhou a capacidade de trabalhar no lado do servidor, ele abre novas maneiras de criar aplicativos confiáveis e escaláveis. Essa mudança expandiu as capacidades do *JavaScript* e consolidou sua posição como uma das linguagens de programação mais importantes no desenvolvimento de *software* moderno.

2.6. *React Native*

Em 2015 o Facebook apresentou o *React Native*, um *framework* de código aberto projetado para o desenvolvimento de aplicativos móveis para as plataformas *iOS* e *Android*, utilizando uma única

[1] https://github.com/

base de código em *JavaScript*. O *React Native* possibilita a criação de *interfaces* de usuário nativas, combinando a flexibilidade e eficiência do *React* com a capacidade de acessar componentes nativos de cada plataforma, o que o torna uma escolha popular entre desenvolvedores [Alura].

Entre as principais vantagens do *React Native*, destaca-se a reutilização do código, o que tem um impacto significativo na redução de tempo e custo para o desenvolvimento de aplicativos móveis. Isso é possível porque o *framework* permite que o mesmo código seja usado para ambas as plataformas, *iOS* e *Android*, o que elimina a necessidade de reescrever o mesmo código para sistemas operacionais distintos. Além disso, o *React Native* oferece um desempenho muito próximo ao de aplicativos nativos, devido à sua arquitetura que possibilita uma comunicação direta e eficiente com as APIs nativas de cada sistema operacional, fazendo com que o aplicativo tenha a mesma aparência e comportamento de uma aplicação nativa, segundo site oficial da [UDS].

Uma característica fundamental do *React Native* é o uso de uma camada de abstração chamada *bridge* (ponte). A *bridge* facilita a comunicação entre o código *JavaScript* e os componentes nativos, permitindo que a aplicação mantenha uma experiência de usuário fluida e responsiva. Segundo o artigo do *GeeksforGeeks*, a bridge funciona como um intermediário assíncrono, permitindo que o código *JavaScript* envie comandos e receba dados do código nativo sem bloquear a *thread* principal. Esse processo assíncrono é essencial para garantir a performance do aplicativo, prevenindo que ele se torne lento ou trave durante a execução de tarefas. Graças à *bridge*, o *React Native* consegue acessar funcionalidades nativas do dispositivo, como a renderização da *interface* de usuário, o acesso a sensores e a manipulação de dados do sistema, tudo isso sem comprometer o desempenho. Além disso, essa abordagem possibilita que o mesmo código *JavaScript* seja reutilizado em ambas as plataformas móveis, *iOS* e *Android*, garantindo maior eficiência no desenvolvimento e manutenção dos aplicativos [GeeksforGeeks], segundo o artigo).

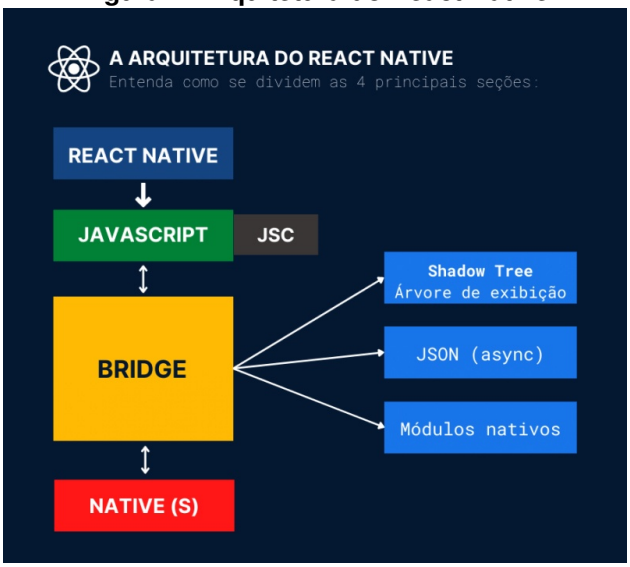
Segundo a análise do [Alura], a adoção do *React Native* em empresas trouxe benefícios de produtividade e eficiência, especialmente em termos de redução de custos e simplificação da manutenção. Como o *framework* permite a reutilização do código entre plataformas, os desenvolvedores podem se concentrar mais nas funcionalidades do aplicativo, ao invés de desenvolver e manter código duplicado para diferentes plataformas.

Empresas como *Tesla*, *Discord*, *Pinterest* e *Facebook*, entre outras, adotaram o R*eact Native* para o desenvolvimento de seus aplicativos móveis, destacando sua eficácia e confiabilidade no mercado. O *framework* se consolidou rapidamente como uma solução eficaz para o desenvolvimento de aplicativos móveis modernos, escaláveis e de alta performance [React 2024], segundo o site oficial).

O crescimento do *React Native* também pode ser observado em sua crescente adoção pela comunidade de desenvolvedores. Segundo o site oficial do [React 2024], o *framework* rapidamente se tornou uma das ferramentas mais populares no desenvolvimento móvel, conquistando uma posição de destaque em repositórios de código aberto como o *GitHub*, com um aumento significativo em contribuições ao longo dos anos.

No artigo disponibilizado pela Alura, a arquitetura do *React Native* é descrita como composta por quatro componentes principais: *React Native, JavaScript (JSC), Bridge* e Módulos Nativos. A figura 2 apresenta a arquitetura do *React Native*.

Figura 2. Arquitetura do React Native



[Alura 2024]

O *React Native* é responsável por gerenciar a aplicação e a *interface* do usuário, permitindo que os desenvolvedores escrevam código em *JavaScript* utilizando a biblioteca *React*. Esse código é posteriormente traduzido para código nativo, permitindo a criação de aplicativos para as plataformas *iOS* e *Android* com uma única base de código, o que torna o processo de desenvolvimento mais ágil.

A camada de *JavaScript* (JSC) executa o código, sendo responsável por gerenciar o ciclo de vida dos componentes, controlar a renderização da interface e realizar a lógica de negócios, tudo com um desempenho otimizado. Essa abordagem permite que a experiência do usuário seja semelhante à de aplicativos nativos, mesmo quando se utiliza *JavaScript*.

A *Bridge* desempenha a função de interligar o código *JavaScript* com os módulos nativos, viabilizando uma comunicação assíncrona entre as camadas e permitindo o acesso a funcionalidades específicas do dispositivo, como câmera e GPS, sem impactar a performance da aplicação.

No desenvolvimento nativo, os programadores utilizam Objective-C ou *Swift* para *iOS*, e *Kotlin* ou Java para Android. Essas linguagens são adaptadas às necessidades de cada plataforma, sendo usadas para acessar funcionalidades avançadas dos dispositivos. A *thread* principal da *interface* do usuário continua disponível, o que assegura uma renderização eficiente. Os Módulos Nativos, por sua vez, garantem o acesso a funcionalidades como sensores e armazenamento, sendo acessados através do Bridge. Isso possibilita que o *React Native* combine a flexibilidade do *JavaScript* com a potência das funcionalidades nativas.

2.7. *Framework*

Os *frameworks* são vistos por [Sommerville 2015] como surgindo para atender ao chamado por desenvolvimento de *software* mais ágil e padronizado. No início, era necessário criar códigos do zero, o que significa que os desenvolvedores tinham que realizar tarefas repetitivas para criar interfaces, manipulação de dados ou integração de banco de dados. O processo de codificação manual não era apenas demorado, mas também arriscado devido a possíveis erros e inconsistências.

Conforme discutido por [Gamma et al. 1994], as crescentes complexidades do sistema também exigiram *software* mais confiável ou reutilização de código. Consequentemente, foi assim e quando os primeiros *frameworks* nasceram, especialmente com a disseminação de linguagens orientadas a objetos, como C++ e Java. Esses primeiros *frameworks* forneceram os melhores meios para reutilizar código já testado e confiável, tornando o desenvolvimento rápido e eficiente.

Segundo [Buschmann et al. 2007] enfatizam que a adoção de padrões de design marcou um marco fundamental nessa evolução. De acordo com [Schmidt 1995], os *frameworks*, portanto, dão suporte ao desenvolvimento ágil de *software*, pois fornecem soluções prontas para problemas comuns, permitindo que os desenvolvedores se concentrem nas funcionalidades específicas do projeto, em vez de recodificar detalhes de nível básico ou lidar com aspectos de baixo nível da tecnologia. Em [Mezini and Ostermann 2002] argumentam que a reutilização de componentes padrão não apenas reduz o tempo necessário para o desenvolvimento, mas também facilita a necessidade de alterar os requisitos do sistema.

Portanto, os *frameworks* fornecem uma maneira organizada e coerente de construir um sistema, o que, por sua vez, ajuda a garantir a consistência e a qualidade da base de código. De acordo com [Rocha and Silva 2020], as seguintes convenções definidas tornam mais fácil para qualquer membro da equipe continuar de onde outro parou, permitindo que mesmo projetos complexos permaneçam escaláveis e sustentáveis no longo prazo.

2.8. MONGOdb

De acordo com [Hows et al. 2014], o *MongoDB* deriva da palavra inglesa *humongous*, que significa gigantesco. Por ser um banco de dados não relacional (*NoSQL*), ele não possui os conceitos tradicionais de tabelas, esquemas, SQL ou linhas. Além disso, não há suporte a transações com conformidade ACID, joins ou chaves estrangeiras. Em vez disso, o *MongoDB* trabalha com documentos no lugar de linhas, sendo extremamente rápido, amplamente escalável e de fácil uso. Essa abordagem é viável porque o sistema abandona algumas funcionalidades típicas dos bancos relacionais, tornando-se ideal para o armazenamento de dados complexos [Hows et al. 2014].

Em [Paniz 2016] explica que, no *MongoDB*, quando um documento é armazenado em uma coleção (equivalente a uma linha em uma tabela de um banco relacional), os dados são guardados em um formato semelhante ao JSON, conhecido como BSON (*Binary JSON*). Os campos do *MongoDB*, por sua vez, correspondem às colunas de uma tabela em bancos SQL.

Ainda segundo [Hows et al. 2014], o princípio filosófico básico do *MongoDB* é o entendimento de que um único modelo de banco de dados não é adequado para todas as situações. Bancos de dados relacionais (SQL) são frequentemente utilizados para armazenar dados de todos os tipos, independentemente de sua adequação ao modelo relacional. Isso ocorre porque é mais fácil e seguro ler e escrever dados em um banco do que diretamente em sistemas de arquivos. No entanto, esse uso pode levar os desenvolvedores a trabalhar contra o fluxo natural do design, caso os dados não estejam alinhados ao modelo para o qual o banco foi projetado.

No trabalho de [Sadalage and Fowler 2013] argumentam que, para os desenvolvedores que utilizam bancos de dados *NoSQL*, como *MongoDB* [2], *Cassandra*[3] ou *Riak* [4], a escolha é justificável

[2]https://www.mongodb.com/pt-br

[3]https://cassandra.apache.org/$_{index.html}$

[4]https://riak.com/

pelas vantagens oferecidas, como melhor desempenho, maior escalabilidade e simplificação na programação.

Em [Boaglio 2020] destaca que uma das principais vantagens do *MongoDB* está na organização de dados em função da aplicação. Diferentemente dos bancos relacionais, não há as limitações de uma tabela relacional. Por exemplo, é possível armazenar dentro de um campo estruturas como arrays ou listas de valores, algo inviável em tabelas convencionais. Essa flexibilidade ocorre porque o *MongoDB* é um banco de dados baseado em documentos, projetados para representar toda a informação necessária sem as restrições dos modelos relacionais. Esses documentos são agrupados em coleções (*collections*), que juntas formam o banco de dados.

Por fim, [Sadalage and Fowler 2013] afirmam que o *MongoDB* é uma escolha acertada para o desenvolvimento de aplicativos modernos. Isso se deve, em parte, ao tempo economizado no mapeamento de dados entre estruturas em memória e bancos relacionais. Além disso, bancos *NoSQL* são mais recomendados para cenários com grandes volumes de dados, nos quais a escalabilidade e o desempenho são cruciais.

2.9. Trabalhos relacionados

Esta seção apresenta os trabalhos relacionados encontrados na literatura, com foco nas principais abordagens, técnicas e resultados obtidos nesses estudos. Serão descritas as metodologias empregadas, as soluções propostas e as respectivas contribuições para o estado da arte, bem como as limitações e lacunas que motivaram a realização deste trabalho.

O grupo formado por Felipe de Oliveira Vianna Pinto, Gustavo Bartz Guedes, André Constantino Silva e Daniela Marques em sua pesquisa intitulada: Aplicação de uma solução REST para Filantopia com uso de geolocalização, pelo Instituto Federal de Educação, Ciência e Tecnologia de São Paulo, apresenta o desenvolvimento de uma arquitetura para um aplicativo chamado *Ants*. Cujo objetivo é conectar voluntários e Organizações Não Governamentais (ONGs) utilizando geolocalização, com a finalidade de ampliar o alcance e a quantidade de ações solidárias. O trabalho tem como foco principal o desenvolvimento do *Web Services* e a implementação de testes unitários automatizados, garantindo a eficiência e a confiança [Pinto et al.].

A dupla Leonardo O. Trinca e Fabiana PM Caravierin em sua pesquisa Kune: Aplicativo de Gerenciamento de trabalho voluntário da FATEC Jales, apresenta o aprimoramento do sistema Kune para uma versão web, utilizando o framework Laravel na linguagem PHP. Embora esse projeto seja mais abrangente, permitindo acesso em diversas plataformas, seu funcionamento guarda semelhanças com o presente trabalho de pesquisa, oferecido como uma referência relevante [TRINCA 2023].

O trabalho Sistema de Busca e Notificação de Trabalhos Voluntários, desenvolvido por João Paulo Cavalcante dos Anjos, Eduardo Chagas Oliveira e Luciene Chagas de Oliveira, da Universidade de Uberaba, este trabalho apresenta uma pesquisa que resultou no desenvolvimento de um aplicativo para a plataforma *Windows Phone*, utilizando o servidor *Azure*. O objetivo principal do aplicativo é comunicar vagas disponíveis para trabalhos voluntários, ampliando o número de participantes e incentivos a ações solidárias [dos Anjos et al.].

No trabalho Tecnologia Social: A Doação na Perspectiva do Aplicativo Solidárius, da Universidade Católica de Brasília é outro trabalho voltado para o estudo de aplicativos sociais. O foco do aplicativo é a facilitação de ações, ao invés de se concentrar em trabalhos voluntários.

Ele busca promover a triangulação da comunicação entre usuários que desejam realizar doações, instituições de caridade e organizações que possam intermediar o transporte das doações, transferindo a resolução de questões logísticas. O trabalho foi proposto por Eduardo Amadeu Dutra Moresi, Sibele Gasiela Guedes Godinho, Ricardo Spindola Mariz, Mácio de Oliveira Braga Filho, Jair Alves Barbosa, Michel Carmo Lopes, Waldemar Anton Osmala Júnior e Marcos Augusto Alves Tito de Morais, com o objetivo de apoiar as doações para o grupo de assistência social Sociedade São Vicente de Paulo – Vincentinos[Moresi et al. 2017].

Afonso Felipe de Jesus Lardosa, da Universidade Federal do Pará, apresenta em seu trabalho UNIVOL: Protótipo de uma Aplicação Móvel para Divulgação e Concentração de Vagas de Trabalho Voluntário no Campus da UFPA – Castanhal. Neste Trabalho de Conclusão de Curso, Afonso Felipe de Jesus Lardosa apresenta o UNIVOL, um protótipo de aplicativo desenvolvido para auxiliar os estudantes da UFPA Castanhal a encontrar vagas de estágio voluntário. Desta forma, os alunos têm a oportunidade de adquirir experiência profissional duradoura [LARDOSA et al. 2022].

O trabalho Uso de *Flutter, Node.js* e *Clean Architecture* no Desenvolvimento de um *Software Mobile* para Monitoria Acadêmica, de Wictor Oliveira de Souza. O estudo ressalta a importância da utilização de tecnologias modernas no desenvolvimento de soluções de software, com destaque para a adoção do *Clean Architecture* como base para uma estruturação eficiente e escalável. Essa abordagem permite melhorar tanto a eficiência do sistema quanto a sua usabilidade, contribuindo para a criação de aplicativos mais robustos e de fácil manutenção. A solução utiliza o *Flutter* para o desenvolvimento da interface do usuário e o *Node.js* como *backend*, promovendo uma interação mais intuitiva e eficiente entre monitores e estudantes.[Souza 2024].

Embora existam trabalhos semelhantes na proposta de aplicativos sociais, muitos se concentram em auxiliar pessoas interessadas em atividades como o voluntariado ou na pesquisa de fenômenos relacionados. Contudo, esses estudos diferem significativamente na atenção dada à arquitetura dos softwares desenvolvidos. Entre os trabalhos analisados, apenas um utiliza o *Clean Architecture*, embora seu foco seja exclusivamente voltado para monitoria acadêmica.

O diferencial da proposta apresentada reside na combinação de duas frentes: oferecer oportunidades para estudantes de cursos de tecnologia iniciarem seu contato com o mercado de trabalho, ao mesmo tempo em que prioriza uma arquitetura robusta e escalável para o aplicativo, utilizando os princípios do *Clean Architecture*. Essa abordagem garante tanto a funcionalidade quanto a sustentabilidade do sistema ao longo do tempo.

3. SISTEMAS DE VOLUNTÁRIOS

Esta seção apresenta o desenvolvimento da ferramenta resultante deste trabalho de pesquisa, denominada Voluntaria *Tech*. Serão detalhados os processos, metodologias e ferramentas empregados em sua análise e implementação.

Nos tópicos a seguir, são abordados os seguintes aspectos: as regras de negócio definidas para o sistema; os principais requisitos funcionais; o diagrama de casos de uso; a documentação da arquitetura baseada no modelo C4; o desenvolvimento do *front-end* e da *API REST*; e, por fim, a apresentação da ferramenta em funcionamento.

3.1. Voluntária *Tech*

A crescente demanda por profissionais de Tecnologia da Informação (TI) no Brasil evidencia a necessidade de estratégias eficazes para suprir a escassez de talentos na área. De acordo com uma pesquisa feita pelo [G1 2023], o Brasil enfrentará um *déficit* de aproximadamente 530 mil profissionais de TI até 2025, devido à insuficiência na formação de novos especialistas para atender à demanda do mercado. Esse *déficit* representa um grande desafio para as organizações, especialmente para as organizações não governamentais (ONGs), que frequentemente carecem de recursos financeiros para contratar profissionais qualificados. A escassez de mão de obra qualificada pode comprometer a capacidade dessas instituições em implementar soluções tecnológicas que potencializem suas operações e ampliam seu impacto social.

A iniciativa "Voluntária *Tech*"surge como uma resposta a essa problemática, oferecendo uma plataforma que conecta ONGs a profissionais de TI dispostos a contribuir voluntariamente. Essa abordagem não apenas auxilia as ONGs a suprir suas necessidades tecnológicas sem custos, mas também proporciona aos profissionais em início de carreira a oportunidade de adquirir experiência prática em projetos significativos, fortalecendo seu portfólio e ampliando suas perspectivas profissionais.

Um exemplo concreto é a ONG "Sou Servir", localizada em Salvador, que desde 2023 mobiliza voluntários na área de saúde para ações comunitárias. Através de uma conversa com Emanuele Vita, uma das voluntárias do programa, foi identificada a necessidade de um sistema para agendamento de ações comunitárias, integração de líderes comunitários, e gestão das principais demandas da comunidade. A ONG também destacou a importância de ter uma plataforma que auxilie no gerenciamento de seus principais projetos, garantindo uma melhor organização das ações e maior eficácia na implementação das atividades. A colaboração com a "Voluntária *Tech*"pode ajudar a otimizar esses processos e melhorar a gestão das iniciativas comunitárias, ampliando o impacto social da "Sou Servir".

Essa colaboração entre ONGs e profissionais de TI voluntários representa uma solução inovadora para mitigar o déficit de profissionais na área, ao mesmo tempo em que fortalece o impacto social das organizações envolvidas.

Portanto, será desenvolvido com um conjunto limitado e cuidadosamente selecionado de requisitos, refletindo uma abordagem focada na criação de um MVP (*Minimum Viable Product*) orientado pela necessidade de criar uma solução intuitiva e eficaz para conectar ONGs e voluntários da área *tech*, priorizando funcionalidades essenciais que promovem colaboração, engajamento e facilidade de utilização.

Neste trabalho, foram priorizadas funcionalidades essenciais que atendem diretamente às principais necessidades dos usuários, como *login*, cadastro e gestão dos projetos. Cada funcionalidade foi projetada para proporcionar um valor imediato e maximizar o impacto com um escopo reduzido, mas mantendo a sua alta qualidade.

A seguir, estão descritos os requisitos funcionais e não funcionais que compõem o sistema Voluntária *Tech*:

3.2. Requisitos funcionais

Os requisitos funcionais identificados para construção da aplicação são:

1. **RF001**: O sistema deve permitir que os voluntários realizem *login* com suas credenciais cadastradas.
2. **RF002**: O sistema deve permitir que as ONGs realizem *login* com suas credenciais cadastradas.
3. **RF003**: O sistema deve manter dados do cadastro para ONGs e voluntários, permitindo o preenchimento e armazenamento das informações necessárias.
4. **RF004**: O sistema deve oferecer um painel administrativo exclusivo para as ONGs, onde possam gerenciar projetos, visualizar detalhes e acompanhar a lista de voluntários associados a cada projeto.
5. **RF005**: O sistema deve possibilitar que os voluntários forneçam informações detalhadas sobre suas habilidades técnicas, disponibilidade de tempo e motivação para participar dos projetos.
6. **RF006**: O sistema deve incluir uma funcionalidade de busca para que os voluntários possam localizar projetos pelo título de forma rápida e eficiente.
7. **RF007**: O sistema deve disponibilizar um campo onde a ONG possa adicionar *links* de convite para plataformas de comunicação externas, facilitando o contato com os participantes.
8. **RF008**: O sistema deverá oferecer uma visão geral do projeto, mostrando os participantes envolvidos.
9. **RF009**: O sistema deverá permitir que a ONG aceite ou recuse um voluntário em um projeto.
10. **RF010**: O sistema deve restringir a edição de projetos pelas ONGs caso existam voluntários pendentes de aprovação, aceitos ou recusados, garantindo integridade nas informações.

3.3. Requisitos não funcionais:

Durante a execução do trabalho, foram encontrados alguns requisitos não funcionais, detalhados a seguir:

1. **RNF001**: O sistema deve garantir a segurança dos dados dos usuários utilizando criptografia robusta e aderindo às normas de proteção de dados, como a LGPD e o GDPR.
2. **RNF002**: O sistema deve oferecer configurações que permitam aos usuários controlar a visibilidade de suas informações e ajustar suas preferências de privacidade.
3. **RNF003**: O sistema deve ser escalável, permitindo a expansão de sua capacidade para suportar um aumento no número de usuários e dados sem degradação de desempenho.
4. **RNF004**: O sistema deve ser projetado para ser testável, garantindo que todas as funcionalidades possam ser verificadas por meio de testes automatizados e manuais.
5. **RNF005**: O sistema deve ser de fácil manutenção, com um código estruturado, documentado e modular, facilitando futuras atualizações e correções.
6. **RNF006**: O sistema deve estar integralmente em conformidade com a Lei Geral de Proteção de Dados (LGPD), implementando políticas claras de uso, coleta e armazenamento de dados.
7. **RNF007**: O banco de dados do sistema deve ser implementado utilizando *MongoDB*, garantindo flexibilidade no armazenamento de dados não estruturados.
8. **RNF008**: O backend deve ser desenvolvido em *Node.js*, aproveitando sua escalabilidade e suporte para desenvolvimento de aplicações modernas e robustas.
9. **RNF009**: O sistema deverá ser desenvolvido para aplicativos móveis.

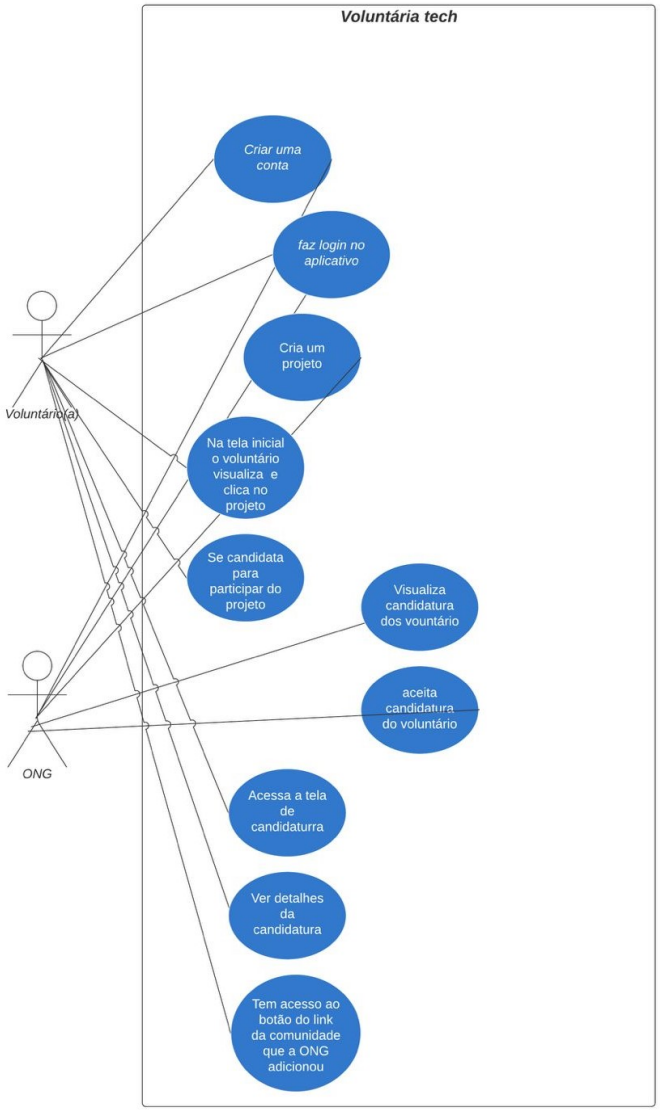
10. **RNF010**: O sistema deverá contar com uma interface manipulável e intuitiva.
11. **RNF011**: O *frontend* da aplicação deve ser desenvolvido utilizando *React Native*, garantindo desempenho eficiente e compatibilidade com múltiplas plataformas.

3.4. Caso de Uso

O diagrama de casos de uso apresentado a seguir foi elaborado com o objetivo de representar, de maneira clara e objetiva, as interações entre os atores e as funcionalidades principais do sistema proposto neste trabalho.

Este caso de uso representado na figura 3, reflete uma funcionalidade essencial para o cumprimento dos objetivos do sistema, destacando como os usuários Voluntário e ONG interagem com a aplicação desenvolvida.

Figura 3. Caso de Uso



Autoria própria

A modelagem foi desenvolvida conforme os princípios da UML (*Unified Modeling Language*), possibilitando uma visão macro do funcionamento do voluntária *Tech*. Este caso de uso descreve o fluxo de interação entrar uma ONG e um voluntário no aplicativo Voluntária *Tech*, desde o cadastro da conta da ONG e voluntário, criação de um projeto feita pela ONG até a aceitação de um voluntário para participar do projeto.

Por meio deste diagrama, é possível compreender de forma intuitiva as principais funcionalidades do sistema e as relações entre os atores e os casos de uso, proporcionando uma base sólida para o desenvolvimento das próximas fases do Voluntária *Tech*.

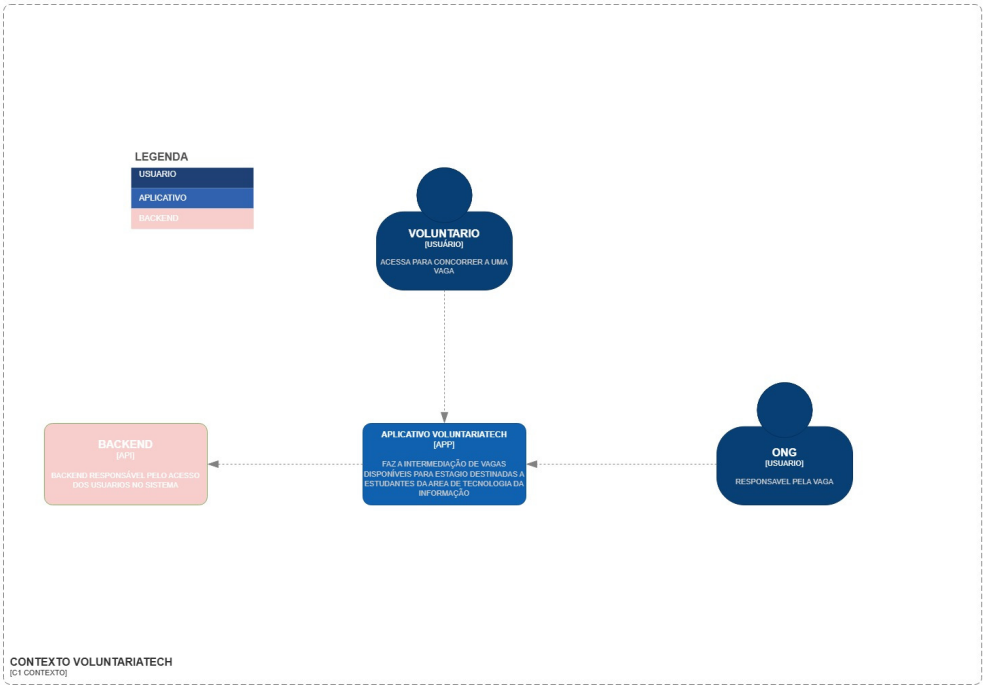
3.5. Documentação - MODELO C4

De acordo com [Brown 2024], O Modelo C4 foi criado para as equipes de desenvolvedores pudessem ver e comunicar com mais facilidade a arquitetura de *software*. Ela é dividida em 4 níveis. O primeiro Nível é o de Contexto, o segundo de Contêiner , o terceiro de componentes e o quarto é último nível é a de código.

Este é o primeiro nível , o Diagrama de Contexto do sistema . Ele inicia a visualização da arquitetura do aplicativo, uma demonstração de como o escopo se apresenta ao mundo ao seu redor.

A figura 4 representa o modelo C4 da arquitetura do aplicativo Voluntaria *Tech:*

Figura 4. Contexto - C1



Autoria própria

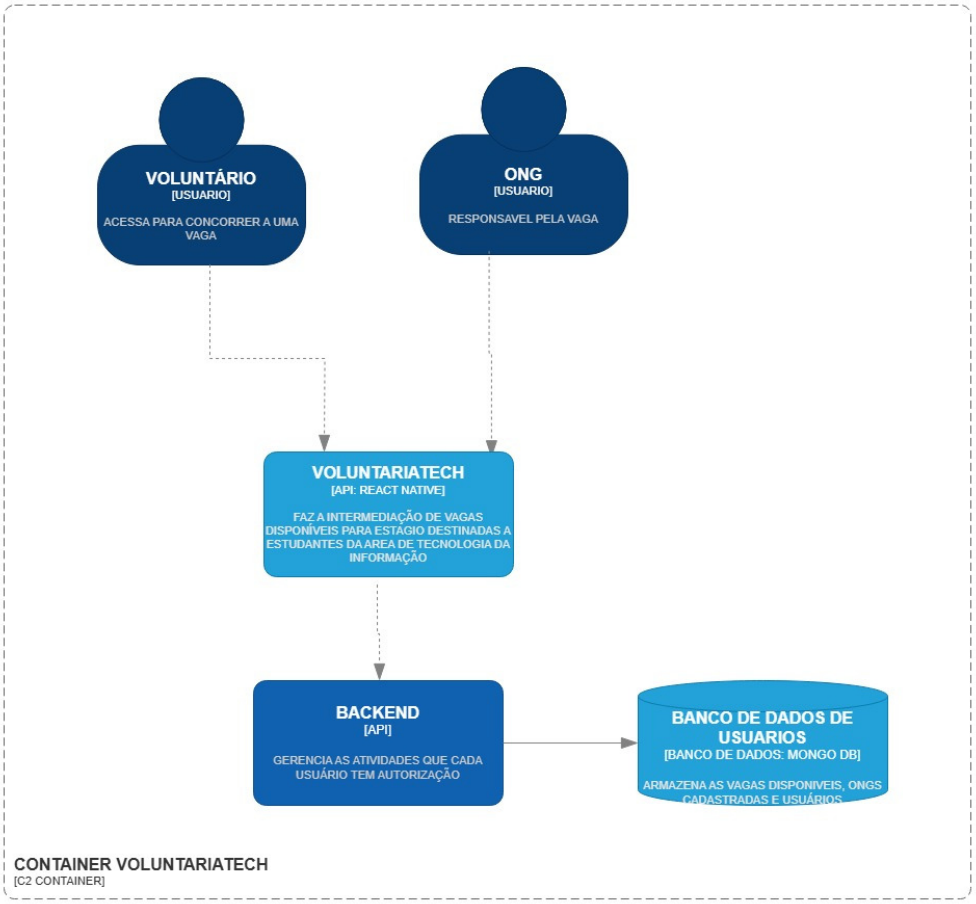
O Diagrama de Contexto (C1) oferece uma visão geral de alto nível do sistema Voluntaria *Tech*, destacando os principais atores e as suas interações com os componentes centrais da aplicação. Ele permite compreender, de forma simplificada, como o sistema se conecta com os atores externos e os fluxos de informação envolvidos.

Usuários Principais:

- **Voluntário:** Um usuário que acessa o aplicativo para buscar e aplicar a vagas de estágio disponíveis na plataforma.
- **ONG:** Outro usuário do sistema, responsável por criar e gerenciar as vagas de estágio no aplicativo.

Já no segundo nível, encontra-se o diagrama de *Contêiner*, conforme apresentado na figura 5, que amplia o escopo apresentado anteriormente. Nesse nível, é possível visualizar, de forma clara, como os dados são armazenados no sistema, bem como a relação entre os diferentes componentes, como aplicações, bancos de dados e serviços que compõem a arquitetura.

Figura 5. Diagrama de*Contêiner* - C2



Autoria própria

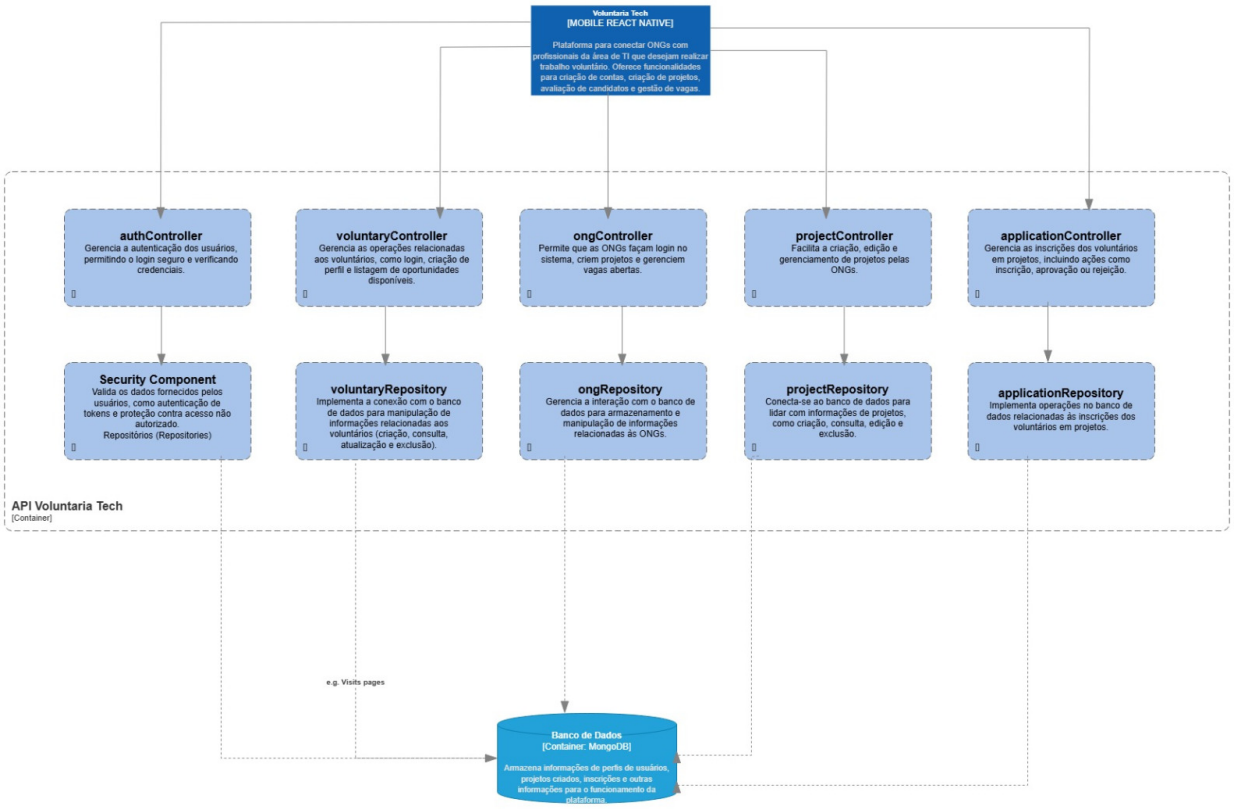
Este diagrama foca nos contêineres da aplicação e suas interações:
Usuários: Dois tipos principais de usuários:

- **Voluntário:** Busca e aplica para vagas.
- **ONG:** Gerencia vagas e projetos.

No terceiro nível, encontra-se o Diagrama de Componentes, conforme pode ser visualizado na figura 6, que aprofunda a visualização de um *contêiner* individual. Esse diagrama detalha os

componentes internos do contêiner, permitindo compreender como eles interagem e colaboram para o funcionamento do sistema.

Figura 6. Diagrama de Componentes - C3



Autoria própria

Este dagrama detalha os principais componentes do *backend* da aplicação. Apresentando os seguintes elementos:

Controllers: Responsáveis por gerenciar as funcionalidades principais do sistema:

- ***authController*:** Garante a autenticação segura dos usuários.
- ***voluntaryController*:** Gerencia ações relacionadas aos voluntários.
- ***ongController:*** Permite o *login* e a gestão de ONGs.
- ***projectController:*** Facilita a criação e gerenciamento de projetos.
- ***applicationController:*** Controla inscrições de voluntários em projetos.

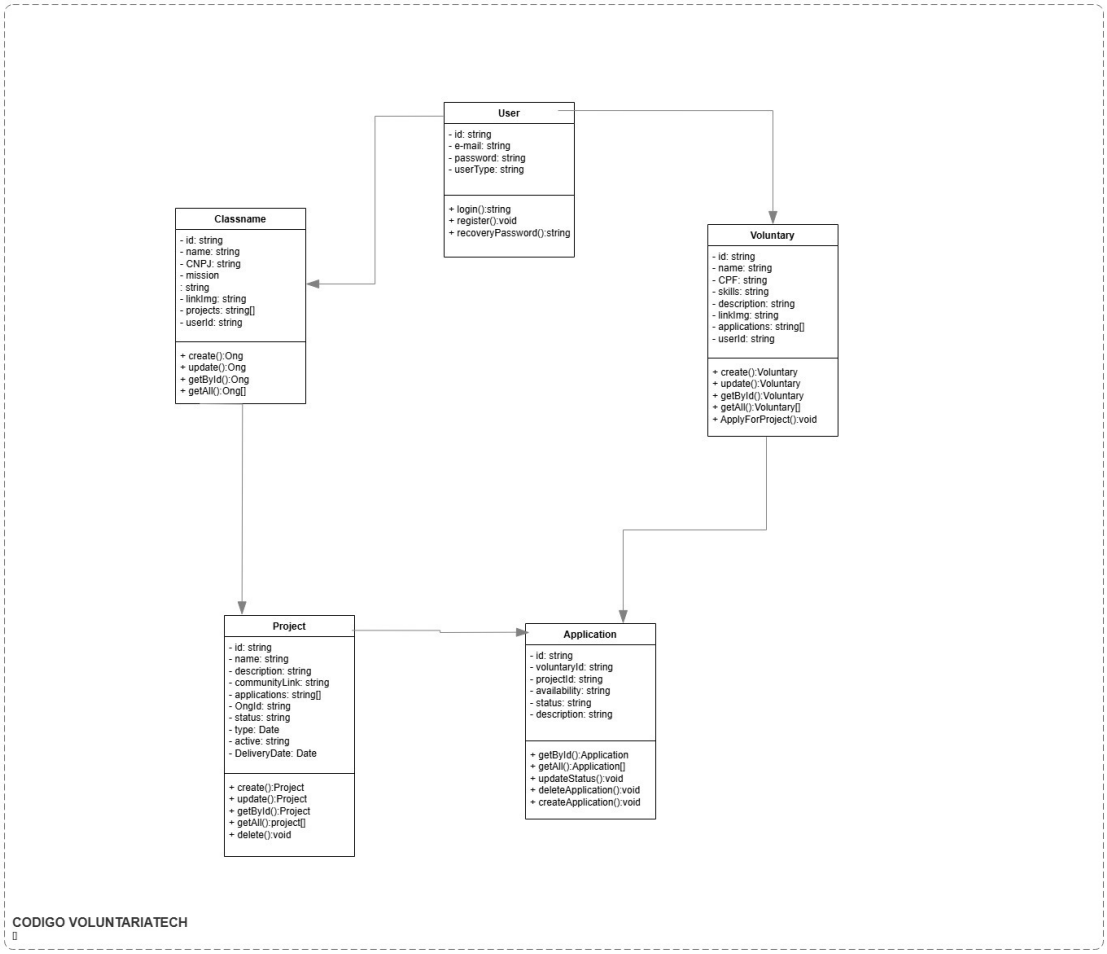
Componentes de Segurança e Repositórios: Responsáveis por garantir a proteção dos dados e funcionalidades do sistema contra acessos não autorizados, ataques ou violações.

- ***Security Component:*** Realiza validações e protege contra acessos não autorizados.
- **Repositórios (*voluntaryRepository, ongRepository, etc.*):** Integram o sistema ao banco de dados *MongoDB*, manipulando informações específicas (voluntários, ONGs, projetos e inscrições).

A base do diagrama conecta todos os componentes ao banco de dados central, que armazena os dados essenciais.

No quarto e último nível, denominado Diagrama de Código, um componente específico é selecionado e representado por meio de um diagrama UML. Conforme pode ser visualizado na figura 7, esse nível detalha a implementação do componente, permitindo uma compreensão clara de sua estrutura e funcionamento no nível do código-fonte.

Figura 7. Diagrama de Código - C4



Autoria própria

O nível mais detalhado do modelo C4, este diagrama apresenta as classes principais e seus relacionamentos.

Classes:

- ***User:*** Representa o usuário (voluntário ou ONG) e suas operações (login, registro, recuperação de senha).
- ***Voluntary*** **e Ong:** Extensões de User com dados e métodos específicos, como gerenciamento de perfis.
- ***Project:*** Gerencia informações relacionadas a projetos.
- ***Application:*** Registra inscrições e *status* de voluntários em projetos.

Métodos e Atributos:

Cada classe define os atributos essenciais, como ID, nome e *status*, além de métodos que permitem realizar as operações de criação, consulta, atualização e exclusão de dados (CRUD). O diagrama ilustra a lógica do sistema, destacando as interações e relações entre as classes, de forma a atender aos casos de uso especificados na aplicação.

3.6. DESENVOLVIMENTO

Nesta seção, será detalhado o desenvolvimento do projeto com foco no *backend*, abrangendo suas principais etapas e estruturas. Além disso, serão apresentadas e explicadas detalhadamente todas as tecnologias empregadas, destacando sua importância e aplicação no contexto do sistema.

3.6.1. *BACKEND*

Para o desenvolvimento do *backend*, foram utilizadas as tecnologias *Node.js* e o *framework Express.js*, que oferece uma estrutura de roteamento flexível, com uma vasta gama de métodos utilitários HTTP e *middlewares*. Essas ferramentas possibilitaram a criação de uma API robusta, rápida e fácil de manter.

Além disso, foi escolhido o *MongoDB* como banco de dados. Por ser um banco de dados não relacional, o *MongoDB* se adequa bem ao contexto do Voluntária *Tech*, que é um MVP. Ele permite esquemas flexíveis, baseados em documentos no formato JSON, facilitando mudanças rápidas no modelo de dados. O *MongoDB* também proporciona uma fácil integração com o *Node.js* por meio do *Mongoose*, que é uma ferramenta de modelagem de objetos *MongoDB* para *Node.js*, garantindo uma interação eficiente entre o banco de dados e a aplicação.

Partindo pro código em si, decidiu-se seguir com a estrutura abaixo representado na figura 8, com o intuito de organizar as pastas do projeto:

Figura 8. Estrutura do projeto

Autoria própria

É importante destacar que o conceito de *Clean Architecture* vai além da organização de pastas, sendo essa apenas uma parte visível da implementação. A verdadeira essência da *Clean Architecture* não está na divisão de diretórios, mas sim na garantia de que as camadas mais internas da aplicação não dependem das camadas externas, promovendo um design desacoplado e flexível.

Na implementação do Voluntaria *Tech*, foram utilizadas as entidades *Voluntary, Ong, Project, User* e *Application*, conforme pode ser visualizado na figura 9 na camada de domínio, que encapsula a lógica central da aplicação, mantendo a independência das camadas externas, como o banco de dados e a *interface* com o usuário.

Entidades:

Figura 9. Entidades: *Voluntary*, ONG, *Project* e *Application*

```
export class Voluntary {
  constructor(
    public name: string,
    public cpf: string,
    public skills: string,
    public linkImg: string,
    public description: string,
    public applications: string[],
    public readonly userId?: string,
    public readonly _id?: string,
  ) {}
}
```

```
export class Ong {
  constructor(
    public name: string,
    public cnpj: string,
    public mission: string,
    public linkImg: string,
    public projects: string[],
    public readonly userId?: string,
    public readonly _id?: string,
  ) {}
}
```

```
export class Project {
  constructor(
    public name: string,
    public description: string,
    public communityLink: string,
    public volunteers: string[],
    public deliveryDate: Date,
    public type: string,
    public active?: string,
    public readonly ongId?: string,
    public status?: string,
    public readonly _id?: string,
  ) {}
}
```

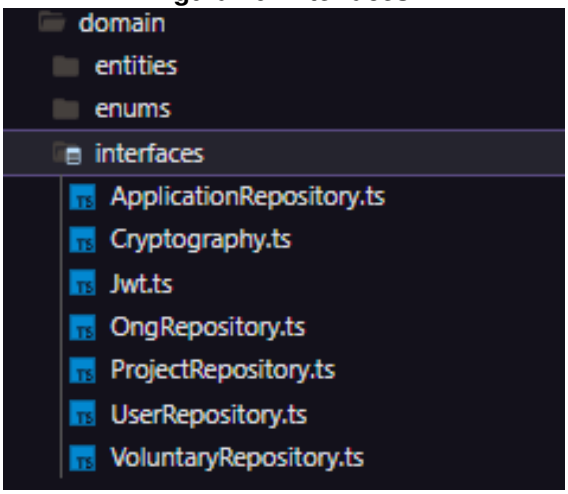
```
export class Application {
  constructor(
    public voluntaryId: string,
    public status: string,
    public readonly projectId: string,
    public availability: string,
    public description: string,
    public readonly _id?: string,
  ) {}
}
```

Autoria própria

Na camada de *use cases* e *controllers*, o código foi estruturado em módulos para proporcionar uma organização mais eficiente e aumentar a manutenibilidade do sistema. Essa abordagem modular facilita a identificação e correção de problemas, além de permitir uma evolução mais controlada da aplicação.

Ainda na camada de domínio, foram criadas *interfaces* para realizar a injeção de dependência, garantindo que a camada de *use case* dependa da abstração e não da implementação concreta. As *interfaces* utilizadas para esse propósito foram figura 10:

Figura 10. Interfaces



Autoria prórpia

Todas as *interfaces* seguem esse padrão, conforme exemplificado na *voluntaryRepository*. Essa abordagem garante consistência na estruturação do código e facilita a implementação e manutenção das dependências no sistema. Um exemplo desta implementação pode ser visualizado na figura 11.

Figura 11. *Voluntary Repository*

```
import { Voluntary } from '../entities/Voluntary';

export interface VoluntaryRepository {
  findAll(filters: Partial<Voluntary>): Promise<Voluntary[]>;
  findOne(filters: Partial<Voluntary>): Promise<Voluntary | null>;
  create(Voluntary: Voluntary): Promise<Voluntary>;
  update(id: string, Voluntary: Voluntary): Promise<void>;
  delete(id: string): Promise<void>;
}
```

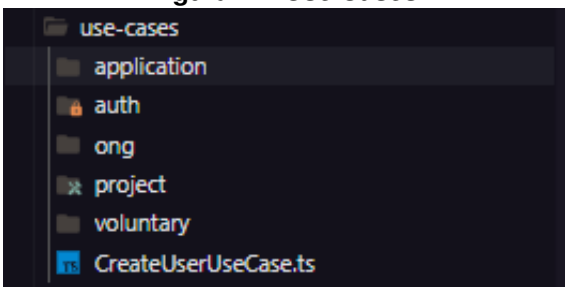
Autoria própria

Na camada de regras de negócio (*use cases*), a implementação foi organizada por módulos, e dentro de cada módulo, as responsabilidades foram distribuídas entre classes distintas. Em cada classe, a dependência foi passada por meio do construtor, garantindo que a injeção de dependência ocorra no momento da instanciação da classe.

Essa abordagem assegura que não haja dependência direta entre as classes, promovendo o desacoplamento do sistema. Como resultado, o design facilita a realização de testes unitários,

uma vez que cada componente pode ser testado isoladamente, sem depender de outros módulos ou classes. A figura 12 ilustra a implementação dessa camada.

Figura 12. *Use Cases*



Autoria própria

Para as entidades voluntários, ONGs, candidaturas e projetos, foi adotada a mesma estrutura, aplicando o princípio da responsabilidade única a cada classe. Dessa forma, cada classe foi projetada para gerenciar uma única responsabilidade, alinhando-se ao princípio da responsabilidade única do SOLID e garantindo um design limpo e coeso.

A Figura 13 ilustra esse exemplo.

Figura 13. Create Voluntary Case

```typescript
import { Voluntary } from '../../domain/entities/Voluntary';
import { VoluntaryRepository } from '../../domain/interfaces/VoluntaryRepository';

export class CreateVoluntaryUseCase {
  constructor(private readonly voluntaryRepository: VoluntaryRepository) {}

  async execute(voluntary: Voluntary) {
    const voluntaryAlreadyExists = await this.voluntaryRepository.findOne({
      cpf: voluntary.cpf,
    });

    if (voluntaryAlreadyExists) {
      throw new Error('Voluntário já cadastrado com esse CPF!');
    }

    return await this.voluntaryRepository.create(voluntary);
  }
}
```

Autoria própria

Na camada de infraestrutura, foi mantida a mesma estrutura adotada para os *controllers*, garantindo que cada classe tenha uma responsabilidade única.

Abaixo, na figura 14 segue um exemplo com o pacote de voluntários.

Figura 14. Update Voluntary By Id Controller

```
import { Request, Response } from 'express';
import { UpdateVoluntaryByIdUseCase } from '../../../use-cases/voluntary';
import { plainToClass } from 'class-transformer';
import { validateOrReject, ValidationError } from 'class-validator';
import { CreateVoluntaryDTO } from '../../dtos';

export class UpdateVoluntaryByIdController {
  constructor(
    private readonly updateVoluntaryByIdUseCase: UpdateVoluntaryByIdUseCase,
  ) {}

  public async handle(request: Request, response: Response): Promise<any> {
    try {
      const createVoluntaryDTO = plainToClass(CreateVoluntaryDTO, request.body);
      await validateOrReject(createVoluntaryDTO);

      const voluntary = await this.updateVoluntaryByIdUseCase.execute(
        request.params.id,
        createVoluntaryDTO,
      );

      return response
        .status(200)
        .json({ message: 'Voluntário editado com sucesso!' });
    } catch (error: any) {
      if (
        Array.isArray(error) &&
        error.every((e) => e instanceof ValidationError)
      ) {
        const formattedErrors = error.map((err: ValidationError) => ({
          field: err.property,
          errors: Object.values(err.constraints || {}),
        }));
        return response.status(400).json({ errors: formattedErrors });
      }

      console.error(error);
      return response.status(400).json({ message: error.message });
    }
  }
}
```

Autoria própria

Na camada de infraestrutura, foram implementadas as abstrações dos repositories e a integração com o banco de dados.

A seguir, na figura 15, é apresentada a implementação do *repository* de voluntário.

Figura 15. Mongo Voluntary Repository

```
import { Voluntary } from '../../domain/entities/Voluntary';
import { VoluntaryRepository } from '../../domain/interfaces/VoluntaryRepository';
import { VoluntaryModel } from '../models/VoluntaryModel';

export class MongoVoluntaryRepository implements VoluntaryRepository {
  findOne(filters: Partial<Voluntary>): Promise<Voluntary | null> {
    return VoluntaryModel.findOne(filters)
      .populate('userId')
      .populate('applications')
  }

  async findAll(filters: Partial<Voluntary>): Promise<Voluntary[]> {
    return (await VoluntaryModel.find(filters)
      .populate('userId')
      .populate('applications'));
  }

  async findById(id: string): Promise<Voluntary | null> {
    return await VoluntaryModel.findById(id);
  }

  async create(Voluntary: Voluntary): Promise<any> {
    const newVoluntary = new VoluntaryModel(Voluntary);
    await newVoluntary.save();
    return newVoluntary;
  }

  async update(id: string, Voluntary: Voluntary): Promise<void> {
    await VoluntaryModel.findByIdAndUpdate(id, Voluntary);
  }

  async delete(id: string): Promise<void> {
    await VoluntaryModel.findByIdAndDelete(id);
  }
}
```

Autoria prórpia

Abaixo, na figura 16, segue também o exemplo da *model* de voluntário, implementada com o *Mongoose*.

O *Mongoose* permite definir a estrutura dos dados de forma robusta e facilita a interação com o banco de dados, fornecendo funcionalidades como validação, criação, atualização e remoção de documentos. Com ele, é possível garantir que os dados armazenados atendam a certos requisitos e facilitar o gerenciamento das operações relacionadas ao banco de dados.

Figura 16. Mongoose IVoluntary

```
import mongoose, { Schema, model, Document, ObjectId } from 'mongoose';

interface IVoluntary extends Document {
  email: string;
  name: string;
  cpf: string;
  skills: string;
  linkImg: string;
  applications: ObjectId[];
  description: string;
  userId?: mongoose.Types.ObjectId;
}

const VoluntarySchema = new Schema<IVoluntary>(
  {
    name: { type: String, required: true },
    cpf: { type: String, required: true, unique: true },
    skills: { type: String, required: true },
    description: { type: String, required: false },
    linkImg: { type: String, required: false },
    applications: [{ type: Schema.Types.ObjectId, ref: 'application' }],
    userId: { type: Schema.Types.ObjectId, ref: 'user', required: true },
  },
  { timestamps: true },
);

const VoluntaryModel = model<IVoluntary>('voluntary', VoluntarySchema);

export { VoluntaryModel, IVoluntary };
```

Autoria própria

A *model* de voluntário foi criada para refletir os dados necessários para o funcionamento da aplicação, incluindo atributos como nome, habilidades, disponibilidade e outros detalhes relevantes para o processo de cadastro e gerenciamento de voluntários.

3.7. Apresentação do Sistema Voluntaria *Tech*

Neste tópico, serão apresentadas as principais telas do sistema, com o objetivo de ilustrar as funcionalidades chave e a experiência do usuário em cada uma delas. Cada tela foi projetada para garantir uma navegação intuitiva e eficiente, atendendo às necessidades dos usuários, seja para voluntários ou para as ONGs.

A seguir, serão apresentadas as telas mais relevantes, destacando como cada uma contribui para o fluxo geral da aplicação, facilitando o acesso às informações e o gerenciamento das funcionalidades do sistema. Além disso, será detalhado o design e as interações que permitem uma experiência fluida e sem dificuldades.

A tela representada na Figura 17 é onde o voluntário ou a ONG realizará o login para acessar o aplicativo. Nessa tela, os usuários podem inserir suas credenciais de forma segura, garantindo que apenas usuários autorizados tenham acesso às funcionalidades do sistema.

Figura 17. Tela de Login

Autoria própria

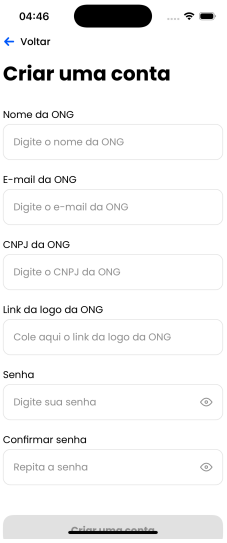
A figura 18 apresenta a tela responsável pela criação de conta, onde o usuário deve escolher o tipo de conta que deseja criar, com as opções disponíveis de ONG ou Voluntário(a). Essa tela permite que o sistema direcione o usuário para o fluxo adequado, dependendo do tipo de perfil selecionado.

Figura 18. Escolher Tipo de Conta

Autoria própria

Como mostrado na figura 19, caso o usuário opte pelo tipo de conta ONG, será necessário preencher os seguintes campos: esses dados são essenciais para a criação de um perfil completo da organização no sistema.

Figura 19. Cadastro de ONG



Autoria própria

Através dessa tela, a ONG poderá fornecer informações como nome, descrição, área de atuação, entre outros detalhes relevantes para sua identificação e funcionamento dentro da plataforma. Além disso, o preenchimento adequado desses campos garantirá que a ONG tenha acesso às funcionalidades necessárias para gerenciar seus projetos, interagir com voluntários e administrar suas atividades na plataforma.

Caso o usuário escolha o tipo de conta ONG, ele vai ter que preencher os seguintes campos:

- Nome da ONG.
- E-mail da ONG.
- CNPJ da ONG.
- Um link da logo da ONG.
- Senha desejada.
- Confirmar a senha desejada.

Como mostrado na tela da figura 20, caso o usuário escolha o tipo de conta Voluntário(a), será necessário preencher os seguintes campos:

- Nome completo.
- E-mail.
- CPF.
- Um link de uma foto o voluntário.
- Habilidades.
- Senha desejada.
- Confirmar a senha desejada.

Figura 20. Cadastro de Voluntário

Autoria própria

Visão do Voluntário

A tela inicial do voluntário, apresentada na figura 21, exibirá uma lista de todos os projetos disponíveis, provenientes de diversas ONGs. Nessa tela, o voluntário poderá visualizar e explorar as opções de projetos, facilitando a busca por oportunidades que correspondam aos seus interesses e habilidades.

Figura 21. Tela Inicial do Voluntário

Autoria própria

A tela de detalhes do projeto, ilustrada na figura 22, exibe as informações completas do projeto selecionado pelo usuário, incluindo o nome, a descrição e um botão que permite ao voluntário

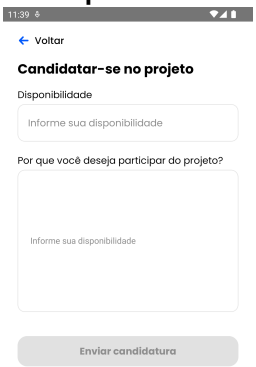
se candidatar a participar daquele projeto. Essa tela oferece uma visão detalhada, facilitando a decisão do voluntário sobre sua participação.

Figura 22. Detalhes do Projeto

Autoria própria

A figura 23 representa a tela para criação de candidatura. Quando o voluntário clica no botão ”Me candidatar”, ele é redirecionado para esta tela, onde deve fornecer informações sobre sua disponibilidade para trabalhar no projeto, bem como o motivo pelo qual deseja participar dele. Esse processo permite à ONG avaliar melhor o interesse e a aptidão do voluntário para o projeto.

Figura 23. Tela para criar candidatura

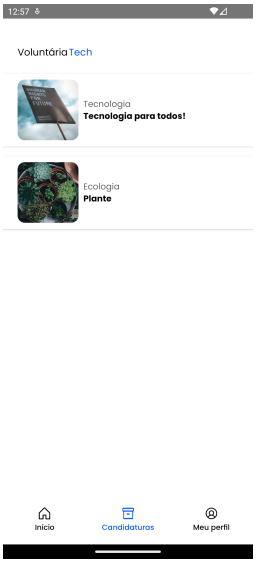


Autoria própria

A figura 24 ilustra a tela de candidaturas, que tem como função exibir as candidaturas

feitas pelo voluntário para os projetos. Nessa tela, o voluntário pode visualizar o *status* de suas candidaturas e acompanhar o progresso de sua participação nos projetos selecionados.

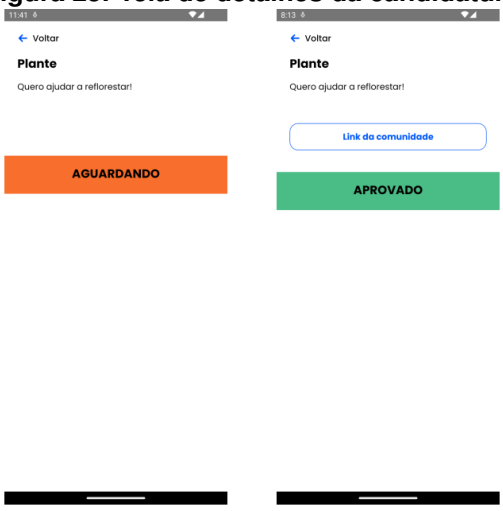
Figura 24. Tela de candidaturas



Autoria própria

A figura 25 ilustra as telas quando o projeto que o voluntário se candidatou é clicado, aparecendo os detalhes do projeto e o *status* de sua candidatura. Sendo possível aparecer: AGUARDANDO, APROVADO, RESCUSADO.

Figura 25. Tela de detalhes da candidatura



Autoria própria

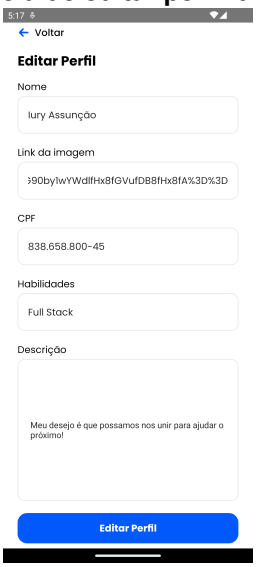
A tela de perfil do voluntário, representada na figura 26, exibe os detalhes completos do perfil do voluntário, incluindo informações como nome, habilidades, foto de perfil e outras informações relevantes. Essa tela proporciona uma visão clara e organizada dos dados do voluntário, facilitando a interação com o sistema e o gerenciamento de suas informações pessoais.

Figura 26. Tela de perfil do Voluntário

Autoria própria

A tela na figura 27 permite ao voluntário editar seu perfil, oferecendo a opção de atualizar os seguintes itens: nome, foto de perfil, habilidades e descrição. Essa funcionalidade proporciona ao voluntário uma maneira fácil de manter suas informações atualizadas e personalizadas, refletindo com precisão suas qualificações e interesses.

Figura 27. Tela de editar perfil do voluntário



Autoria própria

Visão da ONG:

A figura 28 apresenta a tela inicial do usuário do tipo ONG, onde serão listados todos os projetos cadastrados pela organização. Nessa tela, a ONG pode visualizar rapidamente o *status* de seus

projetos, acessar detalhes de cada um e gerenciar informações relacionadas.

Além disso, a interface foi projetada para facilitar a navegação e o acesso às funcionalidades essenciais, proporcionando uma experiência intuitiva para o gerenciamento eficiente dos projetos.

Figura 28. Tela Inicial

Autoria própria

A figura 29 ilustra a tela de detalhes do projeto. Ao clicar em um projeto específico, o usuário é redirecionado para uma visão detalhada, onde são exibidas informações essenciais como o nome do projeto, sua descrição completa, um botão que permite à ONG editar as informações do projeto, o link para a comunidade e uma lista de candidatos interessados em participar.

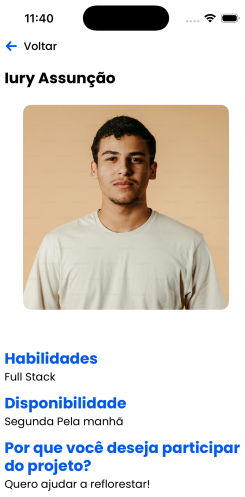
Figura 29. Tela de detalhes do projeto

Autoria própria

Além disso, a ONG tem a capacidade de gerenciar os candidatos, podendo aceitar ou recusar a participação de cada voluntário. Essa funcionalidade oferece um controle total sobre a seleção de membros para o projeto, permitindo que a ONG tome decisões informadas sobre quem se juntará à equipe, com base nas necessidades do projeto e nas qualificações dos voluntários. Dessa forma, a plataforma garante um processo de recrutamento organizado e eficiente, alinhado às expectativas da ONG.

Na tela apresentada na figura 30, a ONG tem a opção de selecionar um dos candidatos e visualizar os detalhes completos de sua candidatura. Nessa tela, são exibidas informações como o motivo da candidatura, a disponibilidade do voluntário, suas habilidades e outros dados relevantes que podem ajudar na avaliação de sua adequação ao projeto. Essa funcionalidade permite que a ONG tome decisões mais informadas ao escolher quais voluntários participarão do projeto, promovendo um processo de seleção mais eficiente e alinhado com as necessidades da organização.

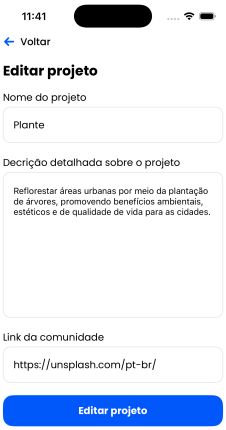
Figura 30. Detalhes da candidatura do voluntário



Autoria própria

A figura 31 representa a tela onde a ONG pode editar os detalhes de um projeto. Nessa tela, a ONG tem a possibilidade de atualizar informações essenciais do projeto, como nome, descrição, prazos e outros parâmetros importantes.

Figura 31. Tela para a ONG editar o projeto



Autoria própria

No entanto, a edição do projeto só é permitida quando não há candidatos associados a ele, garantindo que alterações importantes não interfiram no processo de seleção de voluntários. Esse controle foi implementado para evitar que modificações indesejadas impactem candidatos em processo de inscrição ou envolvimento no projeto, garantindo a integridade e a transparência no gerenciamento dos projetos.

Esta tela representada na figura 32 é responsável por permitir a criação de novos projetos pela ONG. Nela, a organização pode inserir todas as informações essenciais sobre o projeto, como título, descrição, objetivos, prazos e outras especificações importantes para o seu planejamento e execução.

Figura 32. Tela de criar projeto

Autoria própria

A interface foi projetada para ser intuitiva, facilitando o processo de cadastramento do projeto, e garantindo que todos os dados necessários sejam fornecidos de forma clara e organizada. Além disso, a tela inclui validações para assegurar que as informações inseridas atendam aos requisitos do sistema, evitando erros durante o processo de criação do projeto.

A figura 33 ilustra a tela de perfil da ONG, onde são exibidos de forma clara e organizada os detalhes essenciais da organização. Nessa tela, a ONG pode visualizar informações como nome, descrição, área de atuação, logo e outros dados importantes que caracterizam sua identidade e missão.

Figura 33. Tela de perfil ONG

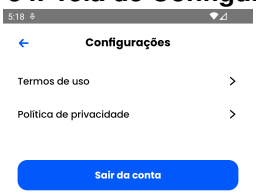
Autoria própria

Além disso, a interface permite que a ONG edite e atualize suas informações de forma prática e ágil, garantindo que o perfil reflita com precisão suas atividades, missão e identidade organizacional. Essa funcionalidade assegura que os dados estejam sempre atualizados, facilitando a comunicação com voluntários e outras partes interessadas.

Também oferece uma navegação intuitiva e bem estruturada, permitindo acesso rápido a outras funcionalidades do sistema, como o gerenciamento de projetos, acompanhamento de candidaturas e interação direta com voluntários, otimizando a experiência do usuário.

A tela de configurações, paresentada na figura 34 do aplicativo oferece ao usuário a possibilidade de personalizar e ajustar preferências de acordo com suas necessidades. Nela, é possível configurar diversas opções, como Termos de uso e preferências de privacidade.

Figura 34. Tela de Configurações



Autoria própria

4. CONSIDERAÇÕES FINAIS

Este trabalho apresentou uma análise detalhada da proposta de uma plataforma que conecta profissionais da área de tecnologia com organizações sociais, visando gerar impactos sociais positivos. A utilização de boas práticas de engenharia de *software*, com foco na *Clean Architecture*, foi fundamental para garantir a criação de uma estrutura flexível, escalável e de fácil manutenção, assegurando a eficiência do sistema ao longo do tempo.

A relevância desse estudo reside na capacidade de utilizar a tecnologia como um agente de transformação social, oferecendo uma solução inovadora para o fortalecimento do voluntariado na área tecnológica. A plataforma proposta não só facilita a interação entre voluntários e ONGs, mas também oferece uma maneira eficaz de impulsionar a colaboração em causas sociais, impactando positivamente as comunidades envolvidas.

Em termos de implicações práticas, o trabalho ressalta a importância de construir soluções tecnológicas que considerem a escalabilidade e a sustentabilidade, adotando arquiteturas que permitam a evolução contínua sem comprometer a qualidade. No campo teórico, a pesquisa contribuiu para a compreensão de como tecnologias bem estruturadas podem ser aplicadas de forma prática em contextos sociais, promovendo mudanças significativas na sociedade.

Propostas futuras incluem a contínua melhoria da plataforma, explorando novos recursos e formas de engajamento, como gamificação e a integração de outras ferramentas de comunicação, para potencializar a interação entre os voluntários e as ONGs. Adicionalmente, a implementação de um sistema de recuperação de senha será fundamental para garantir a segurança e a facilidade de acesso dos usuários à plataforma, além de proporcionar maior autonomia para os voluntários e ONGs.

Outro aspecto importante para a melhoria contínua da plataforma será a implementação de um sistema de feedback para voluntários e ONGs. A coleta de avaliações e sugestões permitirá

que a plataforma seja aprimorada de acordo com as necessidades reais dos usuários, criando um ambiente mais dinâmico e colaborativo. Esse feedback pode incluir tanto aspectos técnicos (como a interface e a usabilidade) quanto a experiência geral de interação entre as partes.

O desenvolvimento contínuo e a adaptação às necessidades reais dos usuários serão essenciais para garantir que a plataforma continue gerando benefícios para a sociedade. Além disso, o projeto pode ser expandido para integrar novas tecnologias, como inteligência artificial, que pode otimizar a distribuição de voluntários e personalizar as experiências. A plataforma também pode ser conectada a sistemas de gerenciamento de ONGs, criando uma rede mais eficiente de apoio às causas sociais.

Em conclusão, a proposta demonstra o potencial de transformação social ao unir tecnologia e solidariedade, contribuindo para a criação de um ambiente mais colaborativo e impactante. Com os próximos passos, a continuidade do projeto tem o poder de ampliar as oportunidades de envolvimento e desenvolvimento para profissionais da área de tecnologia e ONGs, resultando em mudanças positivas duradouras na sociedade. Além disso, a replicação do modelo em diferentes contextos e regiões poderá ampliar ainda mais o impacto social, estabelecendo uma rede global de colaboração.

Referências

Alura. React native: Introdução ao desenvolvimento móvel. https://www.alura.com.br/artigos/react-native. Acessado em: 08 dez. 2024.

Alura (2024). React native: Introdução ao desenvolvimento móvel. *Alura - Cursos Online de Tecnologia*. Acessado em: 08 dez. 2024.

Bass, L. and Paul Clements, R. K. (2013). *Software Architecture in Practice*. Addison-Wesley, 3rd edition.

Boaglio, F. (2020). *MongoDB: Construa Novas Aplicações com Novas Tecnologias*. Editora Específica (se disponível).

Booch, G., Bryan, D. L., and Petersen, C. G. (1994). *Software engineering with Ada*, volume 30608. Addison-Wesley Professional.

Brown, S. (2024). C4 model for visualising software architecture. Acessado em: 08 dez. 2024.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (2007). *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley.

Chebanyuk, A. and Markov, V. (2016). Title of the article. *Journal Name*.

Dahl, R. (2009). Node. js foundation. *Node. js*.

dos Anjos, J. P. C., de Oliveira, E. C., and de Oliveira, L. C. Sistema de busca e notificação de trabalhos voluntários.

Feathers, M. (2004). *Working Effectively with Legacy Code*. Prentice Hall.

Fernández-Ropero, M., Medina-Bulo, I., and Pérez-Chacón, R. (2015). Design principles in object-oriented programming. *Journal of Software Engineering and Applications*, 8(11):557–570.

Ferreira, V. B. S., Ferreira, C. A., and Grande, E. T. G. (2022). Estado da arte da pesquisa em: Clean architecture e princípios de solid. *Research, Society and Development*, 11(16):e335111637198–e335111637198.

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

G1 (2023). Brasil terá déficit de 530 mil profissionais de tecnologia até 2025, mostra estudo do google. Acesso em: 07 dez. 2024.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

GeeksforGeeks. What is a bridge in react native? https://www.geeksforgeeks.org/what-is-a-bridge-in-react-native/. Acessado em: 08 dez. 2024.

Hows, D., Membrey, P., and Plugge, E. (2014). *Introdução ao MongoDB*. Editora Específica (se houver).

Ivanics, P. (2016). An introduction to clean software architecture. *Department of Computer Science, University of Helsinki: Helsinki, Finland*.

LARDOSA, A. F. d. J. et al. (2022). Univol: protótipo de uma aplicação móvel para divulgação e concentração de vagas de trabalho voluntário no campus da ufpa-castanhal.

Lewenhagen, K. and Åkesson, A. (2013). Node.js in open source projects on github. *Open Source Review*. Disponível em: `https://github.com/`. Acesso em: 8 dez. 2024.

Martin, R. C. (2003). *Agile Software Development, Principles, Patterns, and Practices*. Pearson Education.

Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, Upper Saddle River, NJ. 1st edition.

Martin, R. C. (2019). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.

Martin, R. C. (2020). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.

Mezini, M. and Ostermann, K. (2002). Integrating independent components with on-demand remodularization. *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, page 52–67.

Moresi, E. A. D., Godinho, S. G. G., Mariz, R. S., de Oliveira Braga Filho, M., Barbosa, J. A., Lopes, M. C., Júnior, W. A. O., and de Morais, M. A. A. T. (2017). Tecnologia social: a doação na perspectiva do aplicativo solidarius. *Revista Ibérica de Sistemas e Tecnologias de Informação*, (23):1–16.

Norman, D. A. (2013). *The Design of Everyday Things*. Basic Books.

Paniz, D. (2016). *NoSQL: Como Armazenar os Dados de uma Aplicação Moderna*. Editora Específica (se disponível).

Pereira, C. R. (2016). *Aplicações Web em Tempo Real com Node.js*. Casa do Código. ISBN: 978-85-5519-021-6.

Pinto, F. d. O. V., Guedes, G. B., Silva, A. C., and Marques, D. Aplicação de uma solução rest para filantropia com uso de geolocalização.

PRESSMAN, R. S. (2011). Uma abordagem profissional.

React, N. (2024). React native: The framework that brings modern web techniques to mobile. `https://reactnative.dev/`. Acessado em: 08 dez. 2024.

Rocha, C. A. and Silva, D. L. (2020). *Frameworks de Software: Estruturas e Aplicações*. Editora Ciência e Tecnologia.

Sadalage, P. J. and Fowler, M. (2013). *NoSQL Essencial: Um Guia Conciso para o Mundo Emergente da Persistência Poliglota*. Editora Específica (se disponível).

Schmidt, R. (1995). Consciousness and foreign language learning: A tutorial on the role of attention and awareness in learning. *Attention and awareness in foreign language learning*, 9:1–63.

Sommerville, I. (2015). *Software Engineering*. Pearson, 10th edition.

Souza, W. O. d. (2024). Uso de flutter, node. js e clean architecture no desenvolvimento de um software mobile para monitoria acadêmica.

TRINCA, L. d. O. (2023). Kune: aplicativo de gerenciamento de trabalho voluntário.

UDS. React native no desenvolvimento de aplicativos móveis. https://uds.com.br/blog/react-native-no-desenvolvimento-de-apps/. Acessado em: 08 dez. 2024.

Érico Padilha Júnior (2012). Princípios solid e boas práticas de desenvolvimento de software.