UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE TELEINFORMÁTICA (DETI)
CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

ÁTILA NÓBREGA MAIA AIRES

AGENDAQUI: CASO DE USO PARA O ESTUDO E APLICAÇÃO DA ARQUITETURA LIMPA NO DESENVOLVIMENTO DE SOFTWARE

FORTALEZA
2025

ÁTILA NÓBREGA MAIA AIRES

AGENDAQUI: CASO DE USO PARA O ESTUDO E APLICAÇÃO DA ARQUITETURA LIMPA NO DESENVOLVIMENTO DE SOFTWARE

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

Orientador: Prof. Dr. José Marques Soares

FORTALEZA

2025

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

A255a Aires, Átila Nóbrega Maia.
Agendaqui : caso de uso para o estudo e aplicação da arquitetura limpa no desenvolvimento de software / Átila Nóbrega Maia Aires. – 2025.
55 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Centro de Tecnologia, Curso de Engenharia de Computação, Fortaleza, 2025.
Orientação: Prof. Dr. José Marques Soares.

1. Plataforma de agendamento. 2. Desenvolvimento de software. 3. Arquitetura limpa. 4. Arquitetura de software. I. Título.

CDD 621.39

ÁTILA NÓBREGA MAIA AIRES

AGENDAQUI: CASO DE USO PARA O ESTUDO E APLICAÇÃO DA ARQUITETURA LIMPA NO DESENVOLVIMENTO DE SOFTWARE.

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Centro de Tecnologia da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de Bacharelado em Engenharia de Computação.

Aprovada em: 06/03/2025.

BANCA EXAMINADORA

_____

Prof. Dr. José Marques Soares
Universidade Federal do Ceará (UFC)

_____

Prof. Allyson Bonetti França
Instituto Federal de Educação, Ciência e Tecnologia do Ceará (IFCE)

_____

Eng. Luiz Felipe Feitosa Leite
Especialista em Transformação Tecnológica (CI&T)

Aos meus pais, Adonai e Ana Cláudia, pelo incentivo à minha educação. Aos meus irmãos, André e Abner, por serem meu norte e inspiração.

AGRADECIMENTOS

Á meus familiares que por todo o apoio e direcionamento provido. Ao meu pai Adonai, por ser a fonte de força, sabedoria e inspiração em cada etapa da minha vida. À minha mãe, Ana Cláudia, por todo amor e dedicação, que continuam sendo um pilar fundamental na minha vida e nas minhas conquistas. Aos meus irmãos e mentores, Abner e André, cuja orientação, paciência e apoio foram essenciais durante essa jornada.

À minha parceira, Andresa, por sua compreensão, amor e por estar ao meu lado em todos os momentos, oferecendo suporte emocional e inspiração para que eu seguisse em frente.

Ao Prof. Dr. José Marques Soares, pela orientação acadêmica, dedicação e contribuições inestimáveis para minha formação e desenvolvimento deste trabalho. Sua expertise e disponibilidade foram indispensáveis para a concretização deste projeto.

A todos vocês, minha mais profunda gratidão.

"Software é mais que apenas um produto; é uma ferramenta que, quando bem feita, muda o mundo." (BECK, 2000, p. 14).

RESUMO

O objetivo deste projeto é aplicar princípios arquiteturais no desenvolvimento de software por meio do desenvolvimento do Agendaqui, ferramenta multifuncional de gestão de agendamentos em diferentes setores do mercado, que será utilizada como caso de uso, possibilitando analisar os desafios e limitações da implementação prática de um modelo arquitetural planejado. Para isso, a metodologia utilizada incluiu a análise de tecnologias adequadas para *backend*, *frontend*, banco de dados, autenticação, sistema de mensagens de notificação e a implementação de boas práticas de design de software. A plataforma Agendaqui será planejada e desenvolvida usando ideias de arquitetura limpa e busca atuar em diversas áreas diferentes do mercado profissional, incluindo consultas médicas, consultas odontológicas, serviços de beleza, entre outros.

Palavras-chave: Plataforma de Agendamento, Desenvolvimento de Software, Arquitetura Limpa, Arquitetura de Software.

ABSTRACT

The goal of this project is to apply architectural principles to software development through the development of Agendaqui, a multifunctional tool for scheduling management in different market sectors, which will be used as a use case, enabling the analysis of the challenges and limitations of the practical implementation of a planned architectural model. To this end, the methodology used included the analysis of appropriate technologies for *backend*, *frontend*, database, authentication, notification messaging system and the implementation of good software design practices. The Agendaqui platform will be planned and developed using clean architecture ideas and seeks to act in several different areas of the professional market, including medical appointments, dental appointments, beauty services, among others.

Keywords: Scheduling Platform, Software Development, Clean Architecture, Software Architecture.

LISTA DE ABREVIATURAS E SIGLAS

API	Interface de Programação de Aplicações
SMTP	Simple Mail Transfer Protocol
SMTPS	Simple Mail Transfer Protocol Secure
POP3	Post Office Protocol 3
IMAP	Internet Message Access Protocol
UML	Unified Modeling Language
SOLID	Conjunto de cinco princípios de design para programação
SRP	Single Responsibility Principle
OCP	Open/Closed Principle
LSP	Liskov Substitution Principle
ISP	Interface Segregation Principle
DIP	Dependency Inversion Principle
AES	Advanced Encryption Standard (Algoritmo de Criptografía simétrica)
RSA	Rivest-Shamir-Adleman (Algoritmo de Criptografía Asimétrica)
JWT	JSON Web Token
SQL	Structured Query Language
NoSQL	Not Only Structured Query Language
HTTP	Protocolo de Transferência de Hipertexto
REST	Representational State Transfer
SPA	Single Page Application
CCP	Common Closure Principle

SUMÁRIO

1 INTRODUÇÃO

A transformação digital tem intensificado a complexidade do desenvolvimento de software, exigindo que arquitetos e designers encontrem um equilíbrio entre atender às expectativas dos usuários e assegurar a sustentabilidade e manutenibilidade dos sistemas. Nesse contexto, a adoção de boas práticas arquiteturais, como modularização, separação de responsabilidades e a aplicação de padrões bem definidos, é essencial para mitigar os impactos da evolução contínua do software, reduzindo custos de manutenção e facilitando adaptações futuras (PARGAONKAR, 2023, p. 116).

A tomada de decisões arquiteturais representa um obstáculo significativo no desenvolvimento de sistemas, visto que desenvolvedores tomam decisões constantemente sobre estilos arquiteturais, design de Interface de Programação de Aplicações (APIs) e estruturação de classes, frequentemente sem plena consciência dos fatores que influenciam essas escolhas. Esse processo de design da arquitetura de software não se resume apenas à síntese de conhecimento e justificativas racionais, pois envolve múltiplos *stakeholders* e uma série de atividades que vão desde a definição de objetivos até a estruturação do código em diferentes níveis de abstração e dentro do tempo previsto (TANG, 2017).

Diante desse cenário, este trabalho propõe a implementação de um estudo prático sobre a aplicação de princípios arquitetônicos sólidos no desenvolvimento de software. Para isso, será desenvolvida a plataforma web Agendaqui como caso de uso, permitindo avaliar as vantagens e desafios de estruturar um sistema seguindo um método arquitetural bem definido. Essa pesquisa não se limita a validar boas práticas, mas também busca compreender as dificuldades reais da implementação manual de uma arquitetura planejada, considerando fatores como limitações de tempo, complexidade das decisões e cenários nos quais a aderência total a recomendações teóricas pode não ser viável ou, até mesmo, a melhor escolha. Assim, este estudo pretende fornecer uma análise crítica sobre os impactos da arquitetura no processo de desenvolvimento, destacando tanto seus benefícios quanto suas restrições na prática.

A plataforma será planejada com a aplicação dos conceitos arquiteturais estudados. Assim, seu desenvolvimento seguirá um modelo estruturado, com funcionalidades de agendamento e notificação de serviços conforme os padrões arquiteturais definidos, além de uma integração com a Interface de Programação de Aplicações (API) do Google[1].

[1] Link para as APIs do Google Cloud: https://cloud.google.com/apis?hl=pt-BR

O desenvolvimento da plataforma web será realizado utilizando as linguagens de programação Python[2] e JavaScript[3], juntamente com os *frameworks* FastAPI[4], React[5] e Vite[6] para a criação e implementação das funcionalidades.

1.1 Objetivos

O objetivo geral deste trabalho é estudar e aplicar princípios arquiteturais no desenvolvimento de software, projetando e implementando uma arquitetura baseada em boas práticas e padrões estabelecidos. Para isso, a plataforma Agendaqui será utilizada como caso de uso, permitindo analisar os desafios e limitações da implementação prática de um modelo arquitetural planejado. O estudo visa compreender os impactos dessas decisões na organização do código e na evolução do sistema, considerando as dificuldades encontradas ao seguir diretrizes teóricas em um ambiente de desenvolvimento real.

Os objetivos específicos são:

1. Projetar a arquitetura da solução da plataforma Agendaqui.
2. Desenvolver a plataforma Web baseado na arquitetura proposta, incluindo *frontend*, *backend*, banco de dados, integração com API externa da Google e sistema de notificações.
3. Analisar, ao longo do desenvolvimento, as vantagens e dificuldades encontradas na aplicação prática da arquitetura planejada.

1.2 Organização do documento

Este trabalho está dividido em 5 capítulos. No Capítulo 2, é contemplada a fundamentação teórica que engloba a importância de ferramentas de gerenciamento de tempo e notificações que são componentes da solução do nosso caso de uso, os princípios básicos para a construção de uma arquitetura limpa, os elementos importantes para arquitetar a solução, as tecnologias selecionadas para o desenvolvimento da plataforma, os mecanismos de autenticação aplicados à plataforma e os conceitos de mensageria utilizados na implementação do sistema de notificações.

---

[2] Link para a tecnologia Python: https://www.python.org/
[3] Link para a tecnologia Javascript: https://javascript.info/
[4] Link para o *framework* FastApi: https://fastapi.tiangolo.com/
[5] Link para o *framework* React: https://react.dev/
[6] Link para a ferramenta Vite: https://vite.dev/

No Capítulo 3 é apresentado o planejamento do Agendaqui, seguindo os conceitos vistos no capítulo anterior. É abordada a diagramação das entidades da Arquitetura, com destaque para suas funcionalidades, e as tecnologias e *frameworks* a serem utilizados.

No Capítulo 4, é descrito detalhadamente o desenvolvimento da plataforma com cada uma das suas funcionalidades, incluindo a integração com a API do Google e a realização de testes no serviço de mensageria responsável pelas notificações.

Por fim, no Capítulo 5, são apresentadas as considerações finais, observando se os objetivos estabelecidos inicialmente foram atingidos e se todas as funcionalidades desenvolvidas estarão funcionando como esperado.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, são discutidos os fundamentos teóricos relacionados às temáticas de gerenciamento de tempo, notificações, arquitetura limpa, tecnologias de desenvolvimento, autenticação e mensageria para notificações.

2.1 Ferramentas de Agendamento e Notificação

A organização eficiente de compromissos e a comunicação clara entre prestadores de serviços e clientes são pilares fundamentais para garantir a produtividade e o bom funcionamento de diversas atividades econômicas.

Nesta seção, são discutidas as ferramentas utilizadas para gerenciar agendas e notificações, abordando como essas tecnologias evoluíram para atender às demandas de setores variados. Além disso, é analisada a importância de agendas digitais como instrumentos de controle de compromissos e a aplicação de serviços de mensageria e notificações no aprimoramento do engajamento dos usuários e na redução de ausências em compromissos agendados.

2.1.1 Agendas Digitais

No mundo atual, a complexidade e a velocidade das atividades diárias demandam ferramentas tecnológicas que auxiliem na organização e gestão do tempo, como é destacado por Levitin (2015, p. 22):

> A necessidade de assumir o controle de nossos sistemas de atenção e memória nunca foi tão imperativa. Nossos cérebros estão mais ocupados que nunca. Somos bombardeados por fatos, factoides, besteiras e boatos, tudo se apresentando como informação. Tentar descobrir o que você precisa saber e o que pode ignorar é exaustivo, e ao mesmo tempo é o que mais fazemos. Assim, encontrar tempo para agendar nossas diversas atividades tornou-se um tremendo desafio.

Nesse cenário, as agendas virtuais emergem como ferramentas indispensáveis em diversos setores, oferecendo uma alternativa prática e acessível às agendas físicas tradicionais e possibilitando maior eficiência no gerenciamento de compromissos pessoais e profissionais. Existe uma crescente demanda por sistemas que atuem como filtros externos de atenção,

ajudando pessoas altamente bem-sucedidas a gerenciar seu tempo com eficiência e se manterem realmente presentes em suas tarefas de maior importância (LEVITIN, 2025, p. 35). Essa observação destaca o potencial de ganhos de oportunidade em contextos profissionais, onde a otimização do agendamento de compromissos entre clientes e prestadores de serviços pode trazer benefícios significativos ao gerenciamento de atividades.

Soluções como Doctoralia[7] e Calendly[8] foram desenvolvidas com o propósito de conectar profissionais e clientes, facilitando a marcação de compromissos e reduzindo os impactos de ausências e remarcações. Essas plataformas oferecem funcionalidades como integração com calendários externos, notificações automáticas e gerenciamento de disponibilidade em tempo real, características essenciais para a otimização do tempo e dos serviços prestados.

Entretanto, a construção desse tipo de sistema envolve desafios na organização do software e na implementação das funcionalidades, especialmente no que diz respeito às decisões arquiteturais tomadas durante o desenvolvimento. Embora as soluções mencionadas sejam amplamente utilizadas, seu desenvolvimento envolve desafios estruturais que nem sempre são transparentes ao usuário final, assim, o caso de uso da Agendaqui é uma excelente maneira de compreender as dificuldades enfrentadas na aplicação de um modelo arquitetural bem definido, avaliando suas vantagens e limitações na prática.

2.1.2 Serviços de Mensageria e Notificação

A presença de sistemas de notificação é fundamental para garantir a efetividade das plataformas de agendamento, especialmente quando se trata de reduzir o não comparecimento a compromissos agendados.

Segundo o Panorama das Clínicas e Hospitais 2024, elaborado pela Doctoralia e pela Feegow (2024), a taxa de no-show é superior a 11% em 31% das instituições de saúde. Outro levantamento sobre absenteísmo ambulatorial aqui no Brasil apontou que o prejuízo desse não-comparecimento pode chegar a uma média de R$ 396 mil reais em três anos (OLIVEIRA; FILHO; VIEIRA, 2020).

Com a crescente digitalização de serviços e atividades, os sistemas de notificação desempenham um papel crucial na melhoria da experiência do usuário, especialmente em plataformas de agendamento. As notificações são uma maneira eficiente de manter os

[7] Link para o site: https://www.doctoralia.com.br/
[8] Link para o site: https://calendly.com/

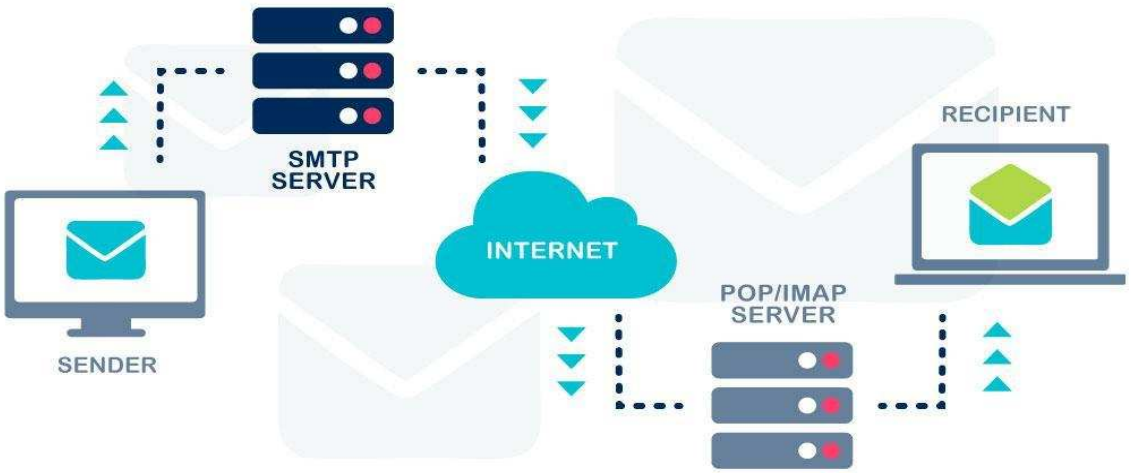
usuários informados sobre eventos importantes, como compromissos, mudanças no agendamento ou lembretes sobre tarefas a serem realizadas. Elas não apenas asseguram que os usuários não se esqueçam de suas responsabilidades, mas também promovem o engajamento contínuo com a plataforma, melhorando a comunicação entre prestadores de serviços e clientes.

Os serviços de mensageria, quando bem implementados, permitem a comunicação automática e em tempo real, garantindo que as mensagens sejam entregues de maneira eficiente e personalizada. Além disso, as notificações podem ser configuradas para diferentes canais, como e-mail ou até mesmo integração com aplicativos de mensagens como WhatsApp.

Para compreender a implementação prática dessas tecnologias, discute-se o funcionamento técnico de protocolos de envio, como o Simple Mail Transfer Protocol (SMTP). O SMTP é um protocolo padrão da Internet utilizado para o envio de e-mails, definido pelo RFC 5321 (KLENSIN, 2008). Sua principal função é possibilitar a comunicação entre servidores de e-mail, enviando mensagens de um remetente para um ou mais destinatários. O protocolo SMTP é responsável apenas pelo envio de e-mails, enquanto o recebimento das mensagens é gerenciado por protocolos específicos, como o Post Office Protocol 3 (POP3), descrito no RFC 1939 (MYERS, 1996), e o Internet Message Access Protocol (IMAP), padronizado no RFC 3501 (CRISPIN, 2003). O POP3 permite que o cliente baixe os e-mails do servidor para acesso offline, enquanto o IMAP possibilita a sincronização dos e-mails entre dispositivos, garantindo maior flexibilidade no gerenciamento de mensagens.

O SMTP opera no modelo cliente-servidor, em que o cliente (por exemplo, uma aplicação de *backend*) se comunica com o servidor SMTP (por exemplo, do Google) para entregar as mensagens aos destinatários. Para isso, o cliente envia comandos para o servidor SMTP, que, por sua vez, executa as ações de encaminhamento do e-mail até o servidor de destino. Uma vez que o servidor SMTP do destinatário recebe o e-mail, ele pode armazená-lo temporariamente até que o destinatário o recupere, sendo os protocolos, POP3 ou IMAP, responsáveis por essa tarefa. O POP3 permite que o destinatário baixe o e-mail para o dispositivo, removendo-o do servidor, enquanto o IMAP mantém a mensagem no servidor e permite a sincronização de múltiplos dispositivos. Esse fluxo completo de envio e recebimento de e-mails garante que as mensagens sejam entregues ao destinatário de forma eficiente e segura (RIABOV, 2007, p. 2 e 14). Uma versão simplificada desse processo pode ser visto na figura 1:

Figura 1 – Representação de uma comunicação via SMTP



Fonte: TURBOSMTP (s.d.).

A segurança no envio dos e-mails pode ser garantida por meio do SMTPS, uma variação do SMTP que faz uso do Transport Layer Security (TLS) para estabelecer uma conexão segura, protegendo as informações transmitidas de interceptações durante o envio. (AWS – AMAZON WEB SERVICES, 2025)

Essas funcionalidades de envio seguro estão implementadas em servidores de e-mail modernos, como o do Gmail. Com o uso de bibliotecas como smtplib (PYTHON SOFTWARE FOUNDATION), é possível implementar um sistema automático de envio de e-mails seguro e eficiente. Basta configurar a conexão com o servidor SMTP, autenticar as credenciais de envio e utilizar a camada de segurança com TLS para garantir a confidencialidade das mensagens.

2.2 Princípios de Arquitetura de Software

A arquitetura de um sistema de software é a forma que este possui, está na divisão desse sistema em componentes, na organização de suas diferentes partes e nos modos como elas se comunicam entre si (MARTIN, 2019, p. 133). Um software de alta qualidade deve levar sempre em conta que a sua arquitetura entrega um valor maior que sua funcionalidade, pois assim ele será capaz de se adaptar facilmente a mudanças de requisitos, garantindo uma

manutenção eficaz e assegurando que sua funcionalidade continue existindo em diferentes cenários adversos (MARTIN, 2019, p.15-16).

A Matriz de Eisenhower, figura 2, é uma ferramenta que ajuda a diferenciar entre o que é urgente e o que é importante, um conceito que também se aplica ao desenvolvimento de software. Sobre essas matriz Eisenhower afirmou (informação verbal)[9]: "Eu tenho dois tipos de problemas, os urgentes e os importantes. Os urgentes não são importantes e os importantes nunca são urgentes".

Figura 2 – Matriz de Eisenhower sobre Arquitetura e Código

Fonte: Adaptado de MARTIN 2019 p. 17.

Um software possui dois valores principais: o comportamento, que é urgente, mas nem sempre importante; e a arquitetura, que é importante, mas nem sempre urgente. Um erro comum que muitos dos gerentes de negócios e desenvolvedores cometem é elevar a importância de itens devido ao fato de serem urgentes, inviabilizando o devido foco naquilo que é realmente importante, como a arquitetura. Portanto, é responsabilidade da equipe de desenvolvimento de um software garantir a importância da arquitetura sobre a urgência dos recursos (MARTIN, 2019, p. 17-18).

Robert C. Martin (2019, p. 17-18) defende que a adoção de princípios arquiteturais bem estabelecidos, como a independência de frameworks, a definição clara de

[9] Palestra na Universidade Northwestern, 1954, em Evanston, Illinois, Estados Unidos.

entidades e casos de uso, a separação em camadas, a aplicação da regra de dependência e a abstração de detalhes técnicos, é essencial para um software mais sustentável e de fácil evolução. No entanto, a aplicabilidade desses princípios pode variar conforme o contexto do projeto, sendo um tema debatido entre arquitetos de software, especialmente quando há restrições de tempo, custo e requisitos específicos que exigem abordagens mais pragmáticas. Nos próximos tópicos, cada um desses princípios será explorado em profundidade.

2.2.1 Independência

Uma boa arquitetura deve suportar de maneira independente os casos de uso e operação do sistema, o seu desenvolvimento e sua implantação, como é citado por Martin (2019, p. 148-150):

- **Casos de uso:** A arquitetura do sistema deve ser projetada para dar suporte à sua intenção principal. Um dos aspectos mais importantes de uma boa arquitetura é tornar a lógica de negócio visível e clara, garantindo que os casos de uso sejam evidentes no nível arquitetural e não fiquem ocultos entre detalhes de implementação.
- **Operação:** A arquitetura tem um papel fundamental na capacidade do sistema de operar de forma consistente e eficiente. Um sistema bem arquitetado deve ser capaz de suportar todas as funcionalidades do software, mesmo que alterações ocorram em componentes como bancos de dados, infraestrutura ou mecanismos de comunicação.
- **Desenvolvimento:** O ambiente de desenvolvimento deve ser sempre apoiado pela arquitetura. É necessário particionar adequadamente o sistema em componentes independentemente desenvolvíveis e bem isolados para que arquitetura facilite a ação entre membro de uma equipe sem interferências.
- **Implantação:** A arquitetura tem um papel relevante em determinar a facilidade com a qual será implantado um sistema. Uma boa arquitetura ajuda o sistema a ser imediatamente implantável depois da construção, sendo viabilizado pelo particionamento e isolamento adequado dos componentes do sistema, incluindo componentes principais responsáveis pela inicialização e integração dos demais.

2.2.2 Entidades e Casos de Uso

A arquitetura de software deve ser construída com foco na regra de negócio, garantindo que a estrutura do sistema proteja e preserve sua lógica essencial. Assim, a arquitetura deve ser projetada de forma que as entidades e casos de uso sejam os elementos centrais do sistema.

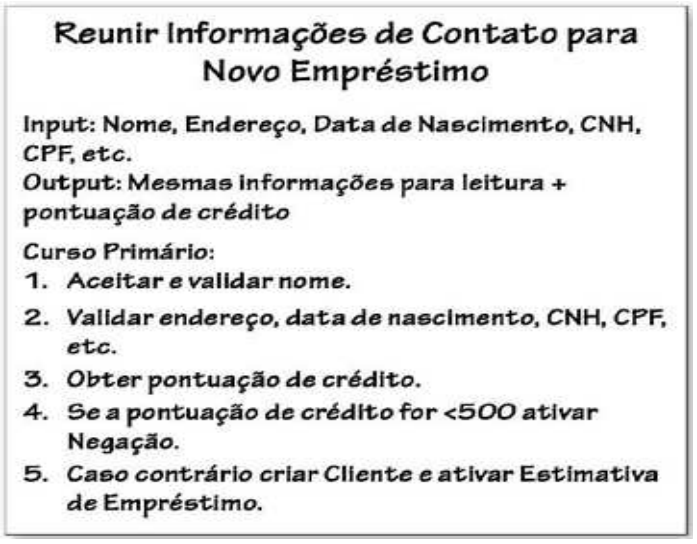
A entidade é um objeto contido no sistema de computador. Ela incorpora um pequeno conjunto de regras de negócio e possui dados cruciais ou acesso facilitado a estes (MARTIN, 2019, p. 190). Por exemplo, a figura 3 representa como uma entidade de empréstimo seria se fosse uma classe Unified Modeling Language (UML). Ela contém três elementos dos dados cruciais de negócio e estabelece três regras na sua interface. Assim, essa entidade reúne o software que implementa um conceito crucial para o negócio e é separado de todas as outras questões do sistema.

Figura 3 – Exemplo de Entidade Empréstimo em UML

Fonte: MARTIN 2019 p. 191.

Os casos de uso, por sua vez, são uma descrição da maneira que um sistema automatizado é usado. Ele especifica a entrada, retorno e passos do processamento, descrevendo regras de negócio específicas da aplicação. Um exemplo disso pode ser visto na figura 4:

Figura 4 – Exemplo de caso de uso



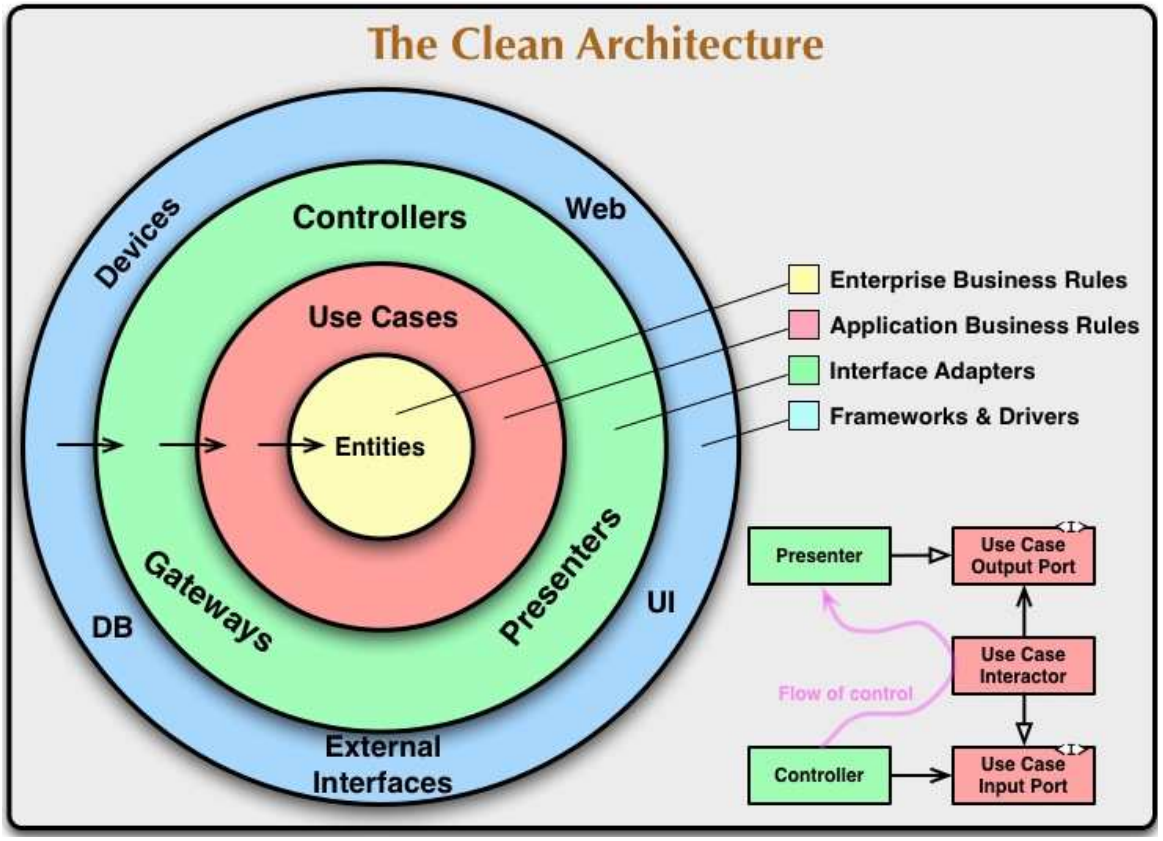
Fonte: MARTIN 2019 p. 192.

2.2.3 Arquitetura limpa em Camadas:

Nas últimas décadas, surgiram várias ideias relacionadas a arquiteturas de sistemas, como Arquitetura hexagonal de Alistair Cockburn; a Data, Context, and Interaction (DCI) de Coplien e Reenskaug e a Entity Control Boundary (BCE) de Ivar Jacobson que, apesar de variarem de alguma forma, são similares – principalmente em seus objetivos: a separação de preocupações por meio de uma divisão do software em camadas. (MARTIN, 2019, p. 202)

A figura 5 é uma tentativa de Martin de integrar todas as características dessas arquiteturas em única ideia acionável:

Figura 5 – Arquitetura Limpa em camadas



Fonte: MARTIN 2012.

As camadas segundo Martin, são (MARTIN, 2012):

- **Entidades:** Representam as regras de negócio mais fundamentais e estáveis do sistema. Essas regras não devem ser impactadas por mudanças na interface, no banco de dados ou em *frameworks*.
- **Casos de Uso:** Definem as regras de negócio específicas da aplicação e coordenam a interação entre entidades e outros componentes. Essa camada deve ser independente do banco de dados, User Interface (UI) ou *frameworks* externos.
- **Adaptadores de Interface**: São responsáveis por converter dados entre as camadas internas (casos de uso e entidades) e os sistemas externos, como banco de dados, APIs e interface gráfica. Aqui encontramos padrões como Model-View-Controller (MVC).
- ***Frameworks* e *Drivers*:** São as ferramentas externas usadas pelo sistema, como bancos de dados, *frameworks* web e serviços de terceiros. Esses componentes devem estar na camada mais externa, sendo tratados apenas como detalhes técnicos, sem impactar a lógica central do software.

A Regra da Dependência determina que as dependências no código devem sempre apontar para dentro. Ou seja, os elementos das camadas internas não podem depender dos

elementos das camadas externas e nada do código interno deve conhecer funções, classes ou variáveis de fora, assim evitando que mudanças nos detalhes externos afetem o núcleo do software (MARTIN, 2019, p. 203). A arquitetura em camadas segue um princípio claro: à medida que avançamos para o centro, a abstração aumenta. O código externo deve servir apenas como um suporte para a lógica do sistema, garantindo modularidade, fácil manutenção e flexibilidade.

2.2.4 Abstrair a Detalhes

Na arquitetura de software, os detalhes de implementação, como bancos de dados, *frameworks* e interfaces externas, devem ser tratados como módulos periféricos, sem impactar a lógica central do sistema. Isso significa que a aplicação deve depender de abstrações, e não de implementações concretas. Os componentes centrais, como regras de negócio e casos de uso, devem ser projetados sem conhecimento direto das tecnologias externas. Dessa forma, é possível trocar ferramentas sem comprometer o núcleo do sistema, garantindo maior flexibilidade e manutenção simplificada. (MARTIN, 2019, p. 277-295)

Esse princípio reforça a importância de manter os detalhes técnicos na camada mais externa da arquitetura, reduzindo o acoplamento e aumentando a estabilidade do software.

2.3 Princípios de Design do Software

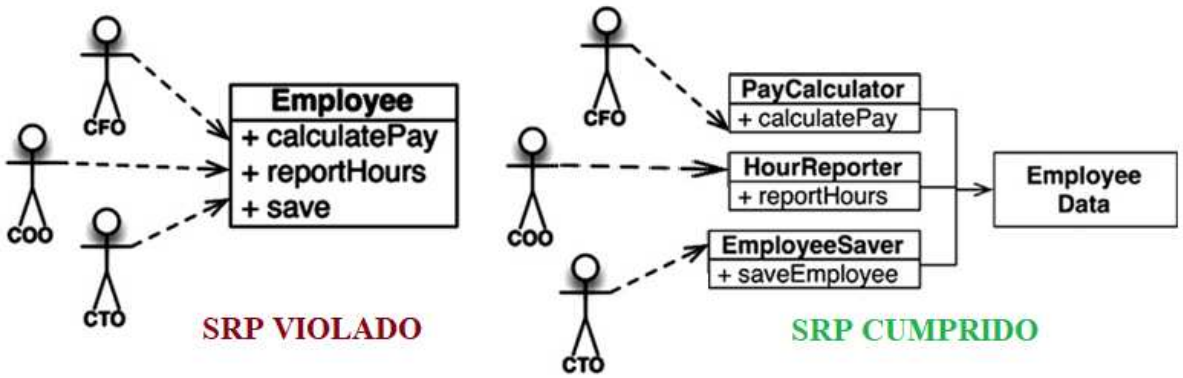
Os princípios de design de software ajudam a criar sistemas modulares, flexíveis e fáceis de manter. Eles orientam a organização do código para minimizar o acoplamento, aumentar a reutilização e facilitar futuras mudanças. Abaixo, exploramos o SOLID, que é um acrônimo para cinco postulados de design, destinados a facilitar a compreensão, o desenvolvimento e a manutenção de software.

2.3.1 O princípio da Responsabilidade Única

O princípio da responsabilidade única, ou Single Responsibility Principle (SRP), é definido por Martin (MARTIN, 2019, p. 62) como: “Um módulo deve ser responsável por um, e apenas um, ator”.

Assim, esse módulo, que pode ser definido como um arquivo fonte ou um conjunto coeso de funções e estruturas de dados, é responsável por um único ator. Na figura 6, podemos ver exemplos onde o SRP é violado ou seguido:

Figura 6 – SRP



Fonte: Adaptado de MARTIN, 2019, p. 63 .

2.3.2 O princípio Aberto/Fechado

O princípio Aberto/Fechado, ou Open/Closed Principle (OCP), possui a seguinte definição (MEYER, 1988, p. 23): “Um artefato de software deve ser aberto para extensão, mas fechado para modificação”.

Assim, o comportamento de um artefato de software deve ser extensível sem que isso modifique o artefato em si. Em nível arquitetural, modificações significam que as funcionalidades devem ser separadas em como, por que e quando da mudança. Em seguida, devem ser organizadas em uma hierarquia de componentes, onde os componentes mais altos da hierarquia são protegidos das mudanças feitas em níveis mais baixos. (MARTIN, 2019, p. 74).

2.3.3 O princípio de Substituição de Liskov

O princípio de Substituição de Liskov, ou Liskov Substitution Principle (LSP), foi escrito por Liskov (1988) da seguinte forma:

> O que queremos aqui é algo como a seguinte propriedade de substituição: se, para cada objeto o1 de tipo S, houver um objeto o2 de tipo T, de modo que, para todos os programas P definidos em termos de T, o comportamento de P não seja modificado quando o1 for substituído por o2, então S é um subtipo de T

O LSP estabelece que uma subclasse deve ser usada no lugar de sua classe base sem afetar a correção do programa. Isso significa que uma subclasse não deve modificar o comportamento esperado da classe pai de maneira que surpreenda ou quebre o sistema. Na prática, esse princípio garante que a hierarquia de classes respeite um comportamento previsível, evitando que subclasses introduzam restrições inesperadas ou alterem funcionalidades essenciais.

Um exemplo comum de violação do LSP ocorre quando uma subclasse sobrescreve um método da classe pai, de forma que este deixe de funcionar conforme o esperado. Quando o princípio é respeitado, a substituição de uma classe por sua subclasse ocorre de forma transparente, sem exigir modificações no código que a utiliza.

2.3.4 O princípio da Segregação de Interface

O princípio da Segregação de Interface, ou Interface Segregation Principle (ISP) é explicado por Martin (2020) com a seguinte afirmação: “Mantenha as interfaces pequenas para que os usuários não dependam de coisas que não precisam.”

Isso significa que as interfaces devem ser específicas e coesas, evitando que classes implementem funcionalidades desnecessárias. Quando esse princípio não é seguido, surgem "interfaces gordas", que obrigam classes a implementar métodos irrelevantes, aumentando o acoplamento e a complexidade do código. Aplicar o ISP corretamente melhora a modularidade, flexibilidade e manutenção do sistema, garantindo que cada classe dependa apenas das funcionalidades essenciais para sua operação (MARTIN, 2019, p. 84-86).

2.3.5 O princípio da Inversão de Dependência

O princípio da Inversão de Dependência, ou Dependency Inversion Principle (DIP) é definido por Martin (2019, p. 87) como: “Segundo DIP, os sistemas mais flexíveis são aqueles em que as dependências de código-fonte se referem apenas a abstrações e não a itens concretos.”

Em outras palavras, módulos de alto nível não devem depender de módulos de baixo nível, ambos devem depender de abstrações. Isso significa que a lógica central do sistema deve ser desacoplada de implementações concretas, garantindo maior flexibilidade e

facilidade de manutenção. Quando o DIP é aplicado corretamente, as dependências são invertidas por meio de interfaces ou classes abstratas, permitindo que os detalhes de implementação sejam modificados sem impactar o núcleo da aplicação. Dessa forma, o sistema se torna mais modular, testável e adaptável a mudanças tecnológicas sem exigir grandes refatorações.

Esse design está diretamente relacionado com o capítulo 2.2.3, pois na separação de camadas, o DIP assegura que a camada de entidades e casos de uso não dependa diretamente da camada de infraestrutura e *frameworks*. Isso permite que o software seja mais modular, testável e flexível, pois mudanças nos detalhes externos (como troca de banco de dados ou de tecnologia de interface) não afetam a lógica principal do sistema.

2.4 Tecnologias de Desenvolvimento Web

O desenvolvimento web moderno é baseado em uma arquitetura cliente-servidor, onde o *frontend* é responsável pela interface com o usuário, enquanto o *backend* gerencia a lógica de negócios, o armazenamento de dados e a comunicação com serviços externos.

As tecnologias de desenvolvimento web evoluíram significativamente nos últimos anos, impulsionadas por demandas como experiência do usuário aprimorada, maior escalabilidade e integração com serviços de terceiros. Dessa forma, *frameworks* e bibliotecas especializadas foram desenvolvidos tanto para o *frontend*, visando interfaces interativas e responsivas, quanto para o *backend*, com foco em desempenho, segurança e modularidade.

2.4.1 Frontend

A arquitetura do *Frontend* moderno evoluiu para um modelo baseado em componentização, onde a interface é dividida em pequenas partes reutilizáveis que interagem entre si. Sobre isso, Martin afirma (2019, p. 106): “Reúna em componentes as classes que mudam pelas mesmas razões e nos mesmos momentos. Separe em componentes diferentes as classes que mudam em momentos diferentes e por razões diferentes”. Esse princípio, também conhecido como Common Closure Principle (CCP), melhora a modularidade, manutenção e escalabilidade do sistema. Além disso, a separação entre camadas de apresentação e lógica de aplicação permite que diferentes partes da interface sejam desenvolvidas de maneira isolada, facilitando o trabalho em equipe e a reutilização de código (KASENDA et. al., 2024, p. 5193-5196).

A construção de um *Frontend* eficiente segue padrões e práticas consolidadas, tais como:

- ➢ **Design Responsivo:** Permite que a interface se adapte a diferentes tamanhos de tela, garantindo usabilidade em dispositivos móveis e desktops.
- ➢ **Single Page Applications (SPAs):** Aplicações que carregam uma única página e atualizam o conteúdo dinamicamente, melhorando a fluidez da navegação.
- ➢ **Componentização:** Organização do código em pequenas unidades reutilizáveis, facilitando a manutenção e a escalabilidade do sistema.

Tais práticas podem ser facilmente implementadas com o auxílio de *frameworks* e bibliotecas, sendo um exemplo disso o React, uma biblioteca declarativa para construção de interfaces baseadas em componentes reutilizáveis. Sua adoção facilita a organização do código, melhora a escalabilidade da aplicação e promove o reuso eficiente de elementos da interface. Além do React, *frameworks* como Vue.js e Angular seguem a mesma abordagem baseada em componentes, proporcionando flexibilidade no planejamento da arquitetura do *frontend* e permitindo que aplicações sejam projetadas sem dependência exclusiva de uma única ferramenta.

A componentização, aliada a uma abordagem declarativa, possibilita a construção de aplicações mais dinâmicas e de fácil manutenção, reduzindo o impacto de mudanças na interface sem comprometer a lógica central do sistema.

2.4.2 Backend

O *backend* é a parte da aplicação responsável pelo processamento de dados, regras de negócio e comunicação com o banco de dados e serviços externos. Diferente do *frontend*, que lida diretamente com a UI, o *backend* atua nos bastidores, garantindo que as operações sejam executadas de maneira eficiente e segura.

Um *backend* bem estruturado deve ser modular, escalável e seguro, permitindo que o sistema gerencie grandes volumes de dados e requisições sem comprometer o desempenho. Além disso, sua arquitetura precisa facilitar a comunicação entre diferentes componentes, garantindo a integração fluida entre *frontend*, banco de dados e serviços externos, como APIs de terceiros.

2.4.2.1 Arquiteturas

O *backend* pode ser estruturado de diferentes formas, dependendo da complexidade e dos objetivos da aplicação. Os principais modelos arquiteturais incluem (PIKKUMÄKI, 2023, p. 12-20):

- **Arquitetura Monolítica:** A arquitetura monolítica é um modelo tradicional onde toda a aplicação é construída como uma única unidade coesa, integrando a lógica de negócio, banco de dados e interface de usuário dentro de um único código-fonte. Código centralizado e de fácil desenvolvimento inicial. Comunicação interna simplificada, pois todos os componentes compartilham o mesmo ambiente.
- **Arquitetura de Microserviços:** A arquitetura de microserviços é uma abordagem moderna que divide a aplicação em múltiplos serviços independentes, cada um responsável por uma funcionalidade específica. Ela proporciona maior escalabilidade pois cada serviço pode ser escalado independentemente conforme a demanda, além de facilitar a manutenção do software, pois alterações em um microserviço não afetam diretamente os outros.
- **Arquitetura Serverless:** A arquitetura Serverless (ou "Computação sem Servidor") permite que as aplicações sejam executadas sem a necessidade de gerenciar infraestrutura, utilizando funções sob demanda que são ativadas apenas quando necessário e o seu custo é baseado em uso. Essas funções funcionam em provedores de Cloud Computing, como AWS Lambda, Google Cloud Functions e Azure Functions. Possui escalabilidade automática e dispensa a configuração manual dos servidores. Além disso, o provedor de Cloud gerencia sua infraestrutura.

2.4.2.2 APIs e RESTful

A comunicação entre sistemas distribuídos no ambiente web ocorre, em grande parte, por meio de APIs, que possibilitam a interação entre diferentes componentes de software, permitindo o intercâmbio de dados e a execução de operações de forma estruturada. De acordo com Fielding (2000), APIs seguem um conjunto de regras e padrões que definem como sistemas podem se comunicar, são úteis em integrar aplicações, serviços e dispositivos diversos.

Para garantir a interoperabilidade entre sistemas, diversos padrões de APIs foram desenvolvidos ao longo do tempo, sendo um dos mais utilizados no contexto web o modelo RESTful. O termo REST (Representational State Transfer) foi introduzido por Roy Fielding MYERS(2000) em sua tese de doutorado, propondo um estilo arquitetural para comunicação entre sistemas distribuídos. Portanto, uma API que segue os princípios do REST é chamada de RESTful API e se baseia na comunicação por meio do Protocolo de Transferência de Hipertexto (HTTP) utilizando um modelo padronizado.

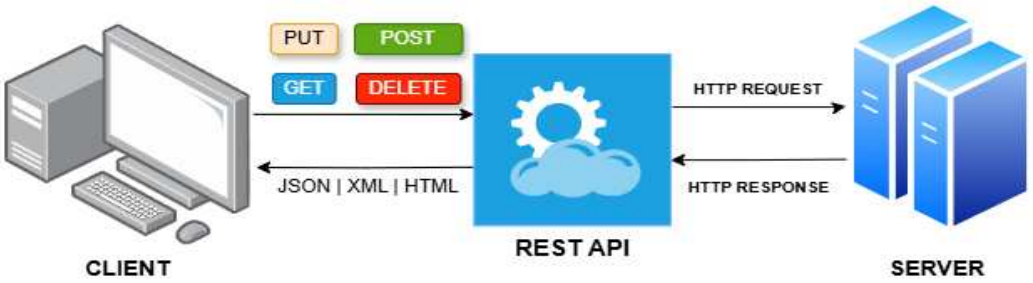
A definição oficial do protocolo HTTP, utilizado nas APIs RESTful, está documentada no RFC 2616, que especifica a estrutura das requisições e respostas, os métodos HTTP (GET, POST, PUT, DELETE) e as diretrizes para o controle de cache e autenticação (FIELDING et al., 1999). Além disso, outros RFCs complementares, como o RFC 7231, aprimoram as diretrizes sobre códigos de status e cabeçalhos HTTP, garantindo a padronização e interoperabilidade entre aplicações (FIELDING et al., 2014). O uso dessas especificações permite que APIs RESTful sejam compatíveis com diferentes clientes e servidores, assegurando a confiabilidade na comunicação entre sistemas distribuídos.

Os principais princípios de uma API RESTful garantem uma comunicação eficiente, escalável e padronizada entre sistemas. O primeiro princípio é o uso de verbos HTTP para definir operações sobre os recursos, como GET para recuperação de dados, POST para criação, PUT/PATCH para atualização e DELETE para remoção, como representado na figura 7.

Outro aspecto essencial é a arquitetura stateless, onde cada requisição ao servidor deve conter todas as informações necessárias para seu processamento, sem depender de estados armazenados entre chamadas. Além disso, a estrutura de uma API RESTful deve ser baseada em recursos acessíveis por meio de URLs intuitivas que facilitem a identificação e manipulação de dados sem a necessidade de parâmetros complexos.

Por fim, nota-se flexibilidade no suporte a diferentes formatos de resposta, sendo o JSON o mais comum. Isso garante compatibilidade entre diversos sistemas e permite que aplicações web, mobile e até mesmo dispositivos IoT consumam a API de maneira eficiente (FIELDING, 2000).

Figura 7 – Representação de uma API RESTful



Fonte: Fornecido pelo autor.

2.4.2.3 Bancos de dados

Os bancos de dados são componentes essenciais no *backend* e são responsáveis pelo armazenamento e recuperação de informações, podendo ser classificados em relacionais (SQL, do inglês Structured Query Language) e não relacionais (NoSQL, do inglês Not Only SQL).

Os bancos de dados SQL organizam os dados em tabelas estruturadas com relações entre si, utilizando linguagem SQL para manipulação de informações. Exemplos populares incluem PostgreSQL, MySQL e SQL Server, sendo amplamente utilizados em aplicações que exigem consistência e integridade dos dados, como sistemas financeiros e corporativos.

Já os bancos de dados NoSQL armazenam os dados de forma mais flexível, sem um esquema fixo, permitindo modelos como documentos (MongoDB), chave-valor (Redis), colunas amplas (Cassandra) e grafos (Neo4j). Essa abordagem é mais indicada para grandes volumes de dados e aplicações escaláveis, como redes sociais e serviços de streaming, onde a velocidade e a distribuição dos dados são mais importantes do que a rigidez das relações.

A escolha entre SQL e NoSQL deve considerar fatores como estrutura dos dados, necessidade de escalabilidade e requisitos de consistência, garantindo a melhor performance e organização da aplicação.

2.5 Métodos de Autenticação e segurança de dados

A segurança da informação é um aspecto fundamental no desenvolvimento de aplicações web, especialmente no que diz respeito à autenticação de usuários e proteção de

dados sensíveis. Métodos eficazes de autenticação e criptografia garantem a integridade, confidencialidade e proteção contra acessos não autorizados.

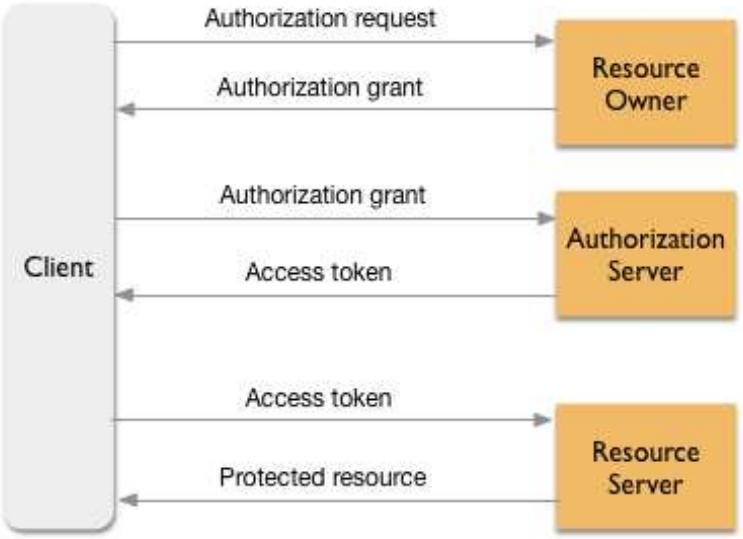
2.5.1 OAuth 2.0:

O OAuth 2.0 é um protocolo amplamente adotado para autenticação segura, permitindo que usuários concedam acesso a suas informações sem expor credenciais sensíveis. Esse modelo é utilizado por grandes plataformas, como Google e Facebook, para permitir login por meio de terceiros, sem que as credenciais do usuário sejam compartilhadas diretamente com o serviço autenticador.

O protocolo opera com tokens de acesso, que são gerados e validados por um servidor de autenticação, garantindo maior segurança e controle sobre as permissões. Além disso, pode ser combinado com JSON Web Tokens (JWT) para uma autenticação eficiente e escalável em aplicações distribuídas.

A Figura 8 ilustra o fluxo de autenticação utilizando JWT em um modelo baseado no OAuth 2.0. O processo inicia quando o cliente envia uma requisição de autorização ao dono do recurso. Após a aprovação, um grant de autorização é enviado ao servidor de autorização, que valida a solicitação e retorna um token de acesso para o cliente. Com esse token, o cliente pode fazer requisições autenticadas ao servidor de recursos, acessando os dados protegidos sem necessidade de re-autenticação, assim garantindo a segurança e controle de acesso de maneira eficiente na plataforma.

Figura 8 – Fluxo OAuth 2.0



Fonte: GOOGLE CLOUD (s.d.).

2.6 Considerações Finais

Os conceitos abordados neste capítulo forneceram a base teórica essencial para o desenvolvimento da plataforma Agendaqui. A definição de princípios arquiteturais, como a Clean Architecture, orientou a organização modular do sistema, garantindo separação de responsabilidades, da mesma forma, a compreensão sobre APIs RESTful e protocolos HTTP, regulamentados pelos RFCs 2616 e 7231, possibilitou a construção de um *backend* padronizado, capaz de interagir de maneira eficiente com o *frontend*.

Além disso, a análise de protocolos de comunicação, como OAuth 2.0 e JWT, foi fundamental para a implementação de um mecanismo seguro de autenticação e autorização, permitindo a integração com provedores externos, como o Google, sem a necessidade de armazenamento de senhas, além de que o estudo de padrões de envio e recebimento de e-mails por SMTP, POP3 e IMAP, descritos nos RFCs 5321, 1939 e 3501, possibilitou a compreensão e criação de um sistema de notificações automatizadas para usuários da plataforma.

A fundamentação teórica apresentada neste capítulo orientou diversas decisões de implementação ao longo do desenvolvimento do Agendaqui, buscando construir um sistema para testar práticas e padrões reconhecidos na indústria de software. No próximo capítulo, será detalhado o processo de concepção da plataforma, incluindo os requisitos do sistema e a estratégia adotada para sua implementação.

3 AGENDAQUI - UMA PLATAFORMA DE AGENDAMENTO DE SERVIÇOS

Neste capítulo é apresentada a plataforma Agendaqui como prova de conceito para arquitetura de software. São abordados a sua conceituação, os objetivos e os diferenciais, além do planejamento para seu desenvolvimento, incluindo as tecnologias utilizadas e a estratégia de implementação.

3.1 Conceituação

A Agendaqui é uma plataforma de agendamento de serviços desenvolvida para otimizar a gestão de compromissos entre prestadores de serviço e clientes. Seu objetivo principal é fornecer uma solução digital intuitiva e eficiente, permitindo que empresas e profissionais organizem seus horários de forma prática, reduzindo falhas no processo de agendamento, como esquecimentos, sobreposições de compromissos e falta de controle sobre a disponibilidade.

A plataforma se propõe a oferecer uma experiência integrada, permitindo que os usuários visualizem horários disponíveis, realizem agendamentos de maneira simplificada e recebam notificações automatizadas. Além disso, o Agendaqui possibilita que profissionais cadastrem sua localização, personalizem seus horários e categorizem seus serviços, atendendo diferentes setores, como clínicas médicas, academias, salões de beleza e consultorias profissionais. Sua principal vantagem está na combinação entre usabilidade, automação de processos e organização dos compromissos, tornando-se uma solução para profissionais que buscam maior controle e produtividade em sua rotina.

Com um sistema baseado em princípios de arquitetura limpa, a plataforma é projetada para ser escalável, modular e flexível, garantindo um ambiente seguro e de fácil manutenção.

3.2 Plano de Desenvolvimento

Nessa seção, é apresentado o plano de desenvolvimento da plataforma, detalhando sua arquitetura e tecnologias utilizadas.

3.2.1 Arquitetura do Sistema

A arquitetura do sistema da Agendaqui segue os princípios da Clean Architecture, proposta por Robert C. Martin e discutidos no capítulo 2.2, visando modularidade, escalabilidade e facilidade de manutenção. Essa abordagem permite que os componentes do sistema sejam bem organizados, reduzindo o acoplamento entre as camadas e facilitando a evolução do software ao longo do tempo.

Para a implementação do sistema, foi adotada a **arquitetura monolítica**, onde toda a aplicação (*frontend*, *backend* e banco de dados) é gerenciada dentro de um único projeto e ambiente. Essa escolha se justifica pela simplicidade de desenvolvimento, manutenção e implantação inicial, além de facilitar a coerência transacional e a integração entre os componentes. Apesar do modelo de arquitetura monolítica, o design do sistema foi estruturado de forma modular, permitindo futuras transições para um modelo distribuído, como microserviços, caso haja necessidade de escalabilidade e crescimento da plataforma.

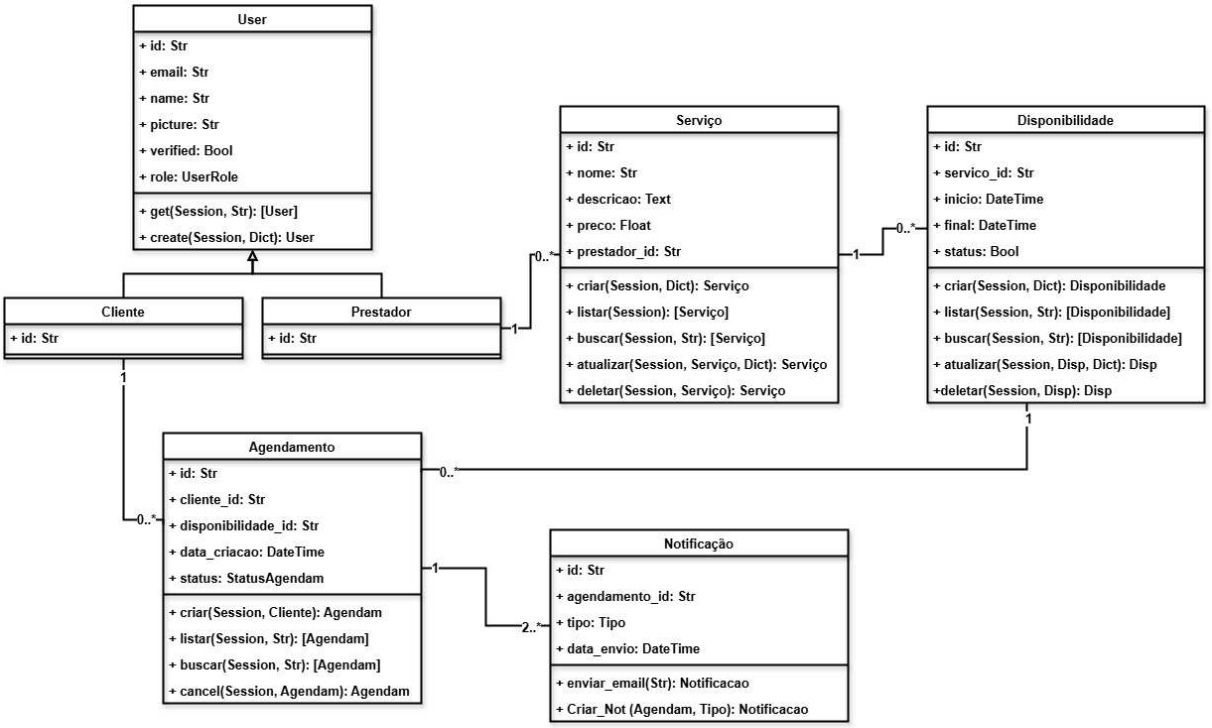
A estrutura do sistema está dividida em quatro camadas principais, assim como visto na seção 2.2.3, garantindo uma separação clara entre regras de negócio e detalhes de implementação. A aplicação foi construída sobre um modelo arquitetural adequado ao contexto do projeto, escolhendo entre abordagens discutidas em 2.4.2.1, de acordo com os requisitos de escalabilidade e manutenção da plataforma.

3.2.1.1 Camada de Entidades

A Camada de Entidades representa o núcleo da aplicação, contendo as regras de negócio mais fundamentais e independentes de qualquer tecnologia externa. No contexto da Agendaqui, as entidades são responsáveis por modelar os conceitos centrais da plataforma, como usuários, Agendamento, disponibilidade, Serviço e notificações. Esses objetos encapsulam as informações essenciais e definem comportamentos que não devem ser alterados pela implementação de interfaces ou detalhes tecnológicos.

Na figura 9, é apresentado um diagrama de Classes UML para representar as entidades do Agendaqui, juntamente com suas relações, atributos e funcionalidades. Esse diagrama reflete a modelagem lógica do sistema, estruturada para seguir os princípios de Clean Architecture, garantindo modularidade e escalabilidade.

Figura 9 – Diagrama de Classes UML das entidades do Agendaqui



Fonte: Fornecido pelo autor.

A entidade **Usuário** é a classe base, responsável por representar qualquer pessoa cadastrada na plataforma. Ela contém os dados genéricos, como nome, e-mail e credenciais de autenticação, e fornece funcionalidades comuns a todos os usuários, como cadastro, login e autenticação.

As entidades **Cliente** e **Prestador** herdam da classe Usuário, assim:

- **Cliente:** Possui funcionalidades específicas, como agendar ou cancelar serviços, e mantém um histórico de agendamentos realizados.
- **Prestador:** É responsável por cadastrar e remover serviços, definir horários disponíveis e gerenciar sua lista de serviços.

Um **Serviço** é o núcleo de um agendamento e está vinculado exclusivamente a um Prestador. Ele contém dados como nome, descrição e características do serviço oferecido.

A **Disponibilidade** representa os horários em que um Serviço pode ser agendado. Ela está diretamente associada ao Serviço, garantindo que cada horário seja exclusivo para aquele serviço. O status da Disponibilidade controla se o horário está livre ou ocupado, podendo ser alterado dinamicamente.

A entidade **Agendamento** conecta Cliente e Disponibilidade. Ela registra o compromisso em um horário específico e mantém atributos como data de criação e status (pendente, confirmado ou cancelado). Essa entidade é essencial para o funcionamento do

sistema de gestão de compromissos. Já a **Notificação** é vinculada diretamente ao Agendamento e permite enviar mensagens para o Cliente e/ou Prestador sobre compromissos agendados. Ela registra informações como a data de envio e o método de notificação, ajudando a reduzir ausências e a manter os usuários informados.

O design apresentado segue os princípios de SOLID, promovendo um sistema coeso e manutenível. O SRP é garantido pela separação clara das responsabilidades entre as entidades: por exemplo, a classe Agendamento lida exclusivamente com a gestão de compromissos, enquanto Notificação gerencia mensagens específicas, evitando sobrecarga em qualquer uma delas. O OCP é atendido com a possibilidade de adicionar novos tipos de notificação ou funcionalidades sem modificar o comportamento existente, graças à modularidade e à especialização. Além disso, o design respeita o LSP, permitindo que Cliente e Prestador substituam Usuário sem causar inconsistências, já que herdam apenas funcionalidades genéricas.

3.2.1.2 Camada de Casos de Uso

A Camada de Casos de Uso é responsável por implementar as regras de negócio específicas da aplicação, atuando como um elo entre a Camada de Entidades e as demais partes do sistema, como os adaptadores de interface e infraestrutura. Cada caso de uso representa uma operação central que o sistema realiza para atender às solicitações dos atores, garantindo que a lógica de negócio seja aplicada de maneira consistente e isolada de detalhes técnicos. A seguir, serão apresentados os casos de uso do Agendaqui:

O caso de uso "Gerenciar Horários Disponíveis" é uma das funcionalidades principais do Agendaqui. Ele permite que o Prestador defina ou atualize os horários nos quais seus serviços estarão disponíveis para agendamento. Essa funcionalidade é essencial para possibilitar ajustes na agenda em função da disponibilidade, demanda ou imprevistos. O sistema valida possíveis conflitos com horários previamente cadastrados e atualiza a disponibilidade com o status inicial como "Disponível", assegurando que somente horários válidos sejam exibidos aos Clientes.

No caso de uso "Agendar Serviço", o Cliente pode selecionar um Serviço oferecido por um Prestador, escolher um Horário Disponível e confirmar o compromisso. O processo inclui a verificação da disponibilidade do horário selecionado, o registro do agendamento vinculando Cliente, Prestador e Serviço, e a atualização do status do horário para "Ocupado". Além disso, o sistema envia notificações automatizadas para confirmar o

compromisso ao Cliente e informar o Prestador, promovendo uma comunicação eficiente e reduzindo a possibilidade de esquecimentos.

O caso de uso "Cancelar Agendamento" possibilita que tanto Clientes quanto Prestadores cancelem compromissos previamente agendados. Após o cancelamento, o sistema atualiza automaticamente o status do Horário para "Disponível", permitindo que ele seja agendado novamente por outro Cliente. O processo inclui a validação de prazos para cancelamento, o registro da ação no sistema e o envio de notificações para ambas as partes envolvidas, garantindo transparência e organização no gerenciamento dos compromissos.

Por fim, o caso de uso "Enviar Notificação" é responsável por assegurar a comunicação eficiente entre o sistema e seus usuários, sejam eles Clientes ou Prestadores. Sempre que ocorre um evento relevante, como um novo agendamento, cancelamento ou alteração de status, o sistema gera notificações personalizadas para informar as partes envolvidas. Essas notificações, enviadas via e-mail ou outro canal configurado, ajudam a reduzir ausências e mantém os usuários informados sobre compromissos e mudanças na agenda.

3.2.1.3 Adaptadores de Interface

A camada de Adaptadores de Interface no Agendaqui é responsável por intermediar a comunicação entre os casos de uso e os elementos externos, como o *frontend*, APIs externas e outros sistemas de integração. Essa camada encapsula as entradas e saídas do sistema, garantindo que as regras de negócio permaneçam protegidas contra alterações no formato dos dados de entrada ou nos requisitos específicos de integração.

Os adaptadores de interface desempenham um papel fundamental no isolamento da lógica de negócio, transformando as solicitações feitas pelos usuários em um formato compreensível para os casos de uso. No contexto do Agendaqui, o *frontend*, projetado para ser flexível quanto ao framework utilizado, interage com os casos de uso por meio de uma API RESTful, que segue os princípios descritos na seção 2.4.2.2. Essa API atua como o principal meio de comunicação entre a aplicação cliente e o *backend*, garantindo que os dados sejam transmitidos de maneira consistente e segura.

Além disso, os adaptadores são responsáveis pela integração com APIs externas, caso o sistema seja expandido para incluir funcionalidades adicionais, como serviços de autenticação via OAuth 2.0 ou sistemas de envio de e-mails. Ao manter as integrações encapsuladas nessa camada, a arquitetura do sistema assegura que futuras alterações em

serviços externos não impactem a lógica central, promovendo modularidade e facilidade de manutenção.

3.2.1.4 Frameworks & Drivers

A camada de *Frameworks* e Drivers no Agendaqui é responsável por gerenciar a infraestrutura técnica e as interações com os componentes externos do sistema. Ela deve atuar como a camada mais externa da arquitetura, encapsulando detalhes técnicos de baixo nível, como *frameworks* de desenvolvimento, bibliotecas, bancos de dados e serviços externos. Essa camada é projetada para ser substituível, garantindo que mudanças em ferramentas ou tecnologias não impactem as camadas internas de lógica de negócio e casos de uso.

3.2.2 Planejamento do Frontend

O planejamento do *frontend* no Agendaqui tem como objetivo criar uma interface de usuário (UI) intuitiva, responsiva e eficiente, garantindo uma experiência fluida tanto para Clientes quanto para Prestadores de Serviço. A arquitetura do *frontend* está baseada no modelo de componentização, permitindo a construção de elementos reutilizáveis e modulares, que promovem a escalabilidade e a manutenção do sistema.

A escolha do React como biblioteca principal justifica-se pela sua popularidade, desempenho e foco em componentes reutilizáveis. Com sua abordagem declarativa, o React facilita o desenvolvimento de interfaces dinâmicas e atualizações eficientes do estado da aplicação. Além disso, ferramentas como o React Router foram utilizadas para gerenciar a navegação entre as páginas, garantindo uma experiência de Single Page Application (SPA), onde o carregamento de dados acontece de forma dinâmica, sem a necessidade de recarregar a página inteira.

O design do *frontend* segue princípios de design responsivo, assegurando que a aplicação seja acessível em diferentes dispositivos, como desktops, tablets e smartphones, para isso, usamos o framework Tailwind CSS para estilizar e padronizar, assim proporcionando um sistema de layout flexível e moderno. Adicionalmente, foi implementada uma integração com a API RESTful do *backend*, permitindo que dados de agendamentos, horários disponíveis e notificações sejam exibidos em tempo real para os usuários.

3.2.3 Planejamento do Backend

O planejamento do *backend* no Agendaqui foca na construção de uma API robusta, segura e escalável, que serve como núcleo para a comunicação entre o *frontend* e o sistema de gerenciamento de dados. A arquitetura do *backend* foi desenvolvida com FastAPI, um framework conhecido por sua performance e simplicidade no desenvolvimento de APIs RESTful, permitindo integrações eficientes e uma gestão clara das rotas e dados.

Para autenticação, foi utilizado o padrão OAuth 2.0, assegurando uma experiência de login segura e flexível, com suporte para token de acesso (JWT). Esse método permite que os usuários se autentiquem e acessem recursos de forma controlada, garantindo a proteção de dados sensíveis. Foi feito o uso da API de autorização do Google para que os usuário possam utilizar a plataforma fazendo um login e autenticação com a conta do Google, assim foi necessário integrar e configurar nossa aplicação para usar essa API e obter dados do usuário. Por fim, o envio de notificações foi realizado por meio da biblioteca smtplib, utilizando o protocolo SMTP para envio de e-mails automáticos.

3.2.4 Planejamento do Banco de Dados

O planejamento do banco de dados no Agendaqui visa garantir a organização, integridade e eficiência no armazenamento das informações essenciais para o funcionamento da plataforma. O banco de dados escolhido para a aplicação foi o PostgreSQL, uma solução relacional amplamente utilizada e reconhecida por sua robustez, flexibilidade e suporte a operações complexas. A estrutura do banco foi projetada para refletir as entidades e suas relações, como Cliente, Prestador, Serviço, Agendamento e Notificação, alinhando-se aos princípios de modelagem relacional.

Para a construção da base de dados, foi utilizada uma diagramagem UML que representa as entidades, atributos e relações. Esse diagrama visa facilitar a visualização da estrutura lógica do sistema, garantindo que a modelagem atenda aos requisitos de negócio e suporte às operações necessárias, além de que a sua criação possibilita identificar relações entre tabelas e definir restrições, como chaves primárias e estrangeiras, assegurando consistência e integridade referencial.

4 DESENVOLVIMENTO

O presente capítulo descreve a implementação prática da plataforma Agendaqui, abordando as tecnologias utilizadas, os processos de desenvolvimento e a integração entre os componentes do sistema, tudo isso seguindo os princípios de arquitetura limpa.

4.1 Configuração do Ambiente de Desenvolvimento

Para garantir um desenvolvimento eficiente e estruturado, a configuração inicial do ambiente do Agendaqui foi planejada com base nas ferramentas e tecnologias definidas no capítulo 3.2. O processo de preparação do ambiente envolveu a escolha de tecnologias adequadas para cada camada do sistema, a organização do repositório de código e a instalação das dependências essenciais para o desenvolvimento da plataforma.

Após a definição das tecnologias, o repositório Git foi configurado para versionamento de código, seguindo boas práticas de controle de versão. O repositório foi organizado em duas pastas principais: "*backend*" e "*frontend*", separando claramente as responsabilidades da aplicação. Além disso, foi definida uma estratégia de branches baseada no fluxo Git Flow, onde a branch principal (main) contém versões estáveis, enquanto novas funcionalidades são desenvolvidas em branches separadas antes de serem integradas.

Por fim, a instalação das bibliotecas e dependências essenciais foi realizada para cada camada da aplicação. No *backend*, foram adicionadas o Framework do **FastAPI** com suas respectivas dependências, também foi adicionado bibliotecas como **SQLAlchemy** para manipulação do banco de dados, **pydantic** para validação de dados das entidades propostas e **smtplib** para envio de e-mails via SMTP e a biblioteca da API do Google, **google-api-python-client**. No *frontend*, foram instaladas dependências como **React Router** para gerenciamento de rotas e **Axios** para comunicação com a API. O uso de um gerenciador de pacotes, como pip para o *backend* e npm para o *frontend*, facilitou a instalação e atualização dos pacotes.

Com essa configuração inicial do ambiente, há uma base sólida para o desenvolvimento das diferentes camadas do Agendaqui.

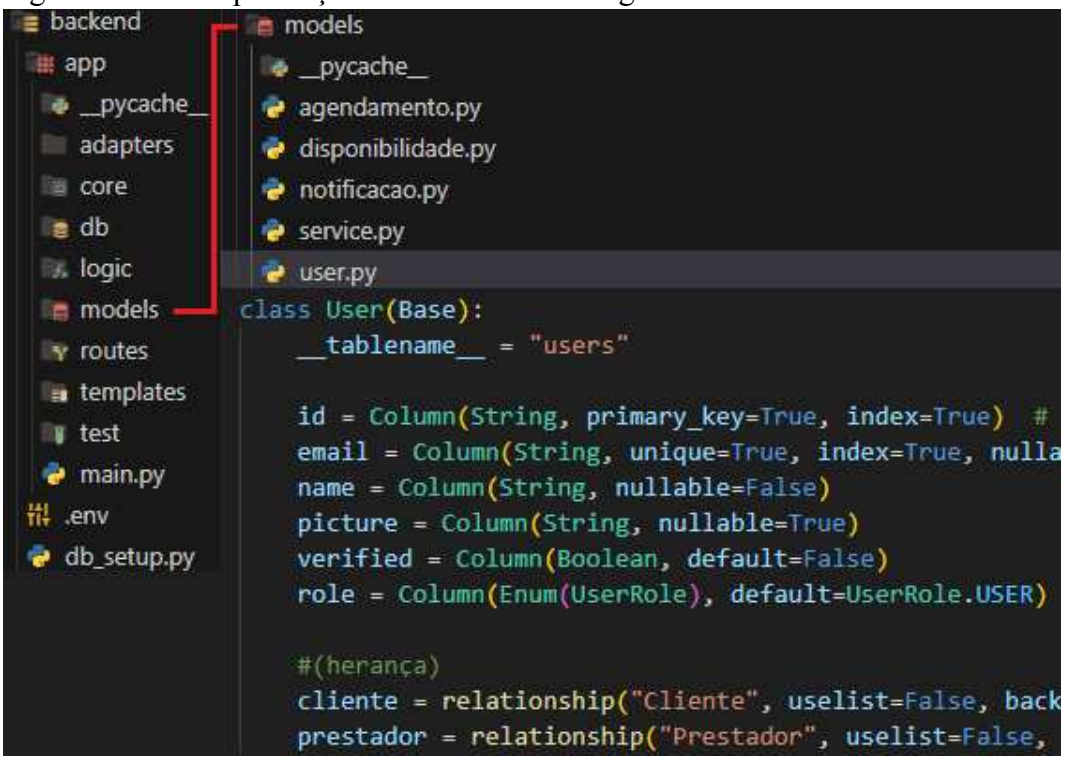
4.2 Implementação do Backend

A implementação do *backend* do Agendaqui seguiu a estrutura planejada, garantindo uma API escalável e bem organizada. O desenvolvimento foi feito utilizando FastAPI, um framework eficiente para a criação de APIs RESTful, escolhido pela sua performance e suporte a tipagem estática. A estrutura do código foi organizada seguindo os princípios da Clean Architecture, separando as responsabilidades em camadas bem definidas para facilitar a manutenção e futuras expansões.

A organização do *backend* seguiu uma estrutura modular, dividida em pastas conforme as camadas da arquitetura. Foram criadas pastas para modelos (entidades), logica (casos de uso), adaptadores de interface, rotas e infraestrutura, assegurando um isolamento adequado entre as partes do sistema. As entidades, responsáveis por representar os objetos de negócio, foram definidas como classes separadas, refletindo os principais conceitos do sistema, como Usuário, Prestador, Cliente, Serviço, Agendamento e Notificação.

A Figura 10 exemplifica essa estrutura aplicada na implementação do *backend* da plataforma Agendaqui. A pasta models contém as definições das entidades, incluindo user.py, que representa o modelo de usuário com seus atributos e relacionamentos.

Figura 10 – Exemplificação da estrutura de código do *Backend*



Fonte: Fornecido pelo autor.

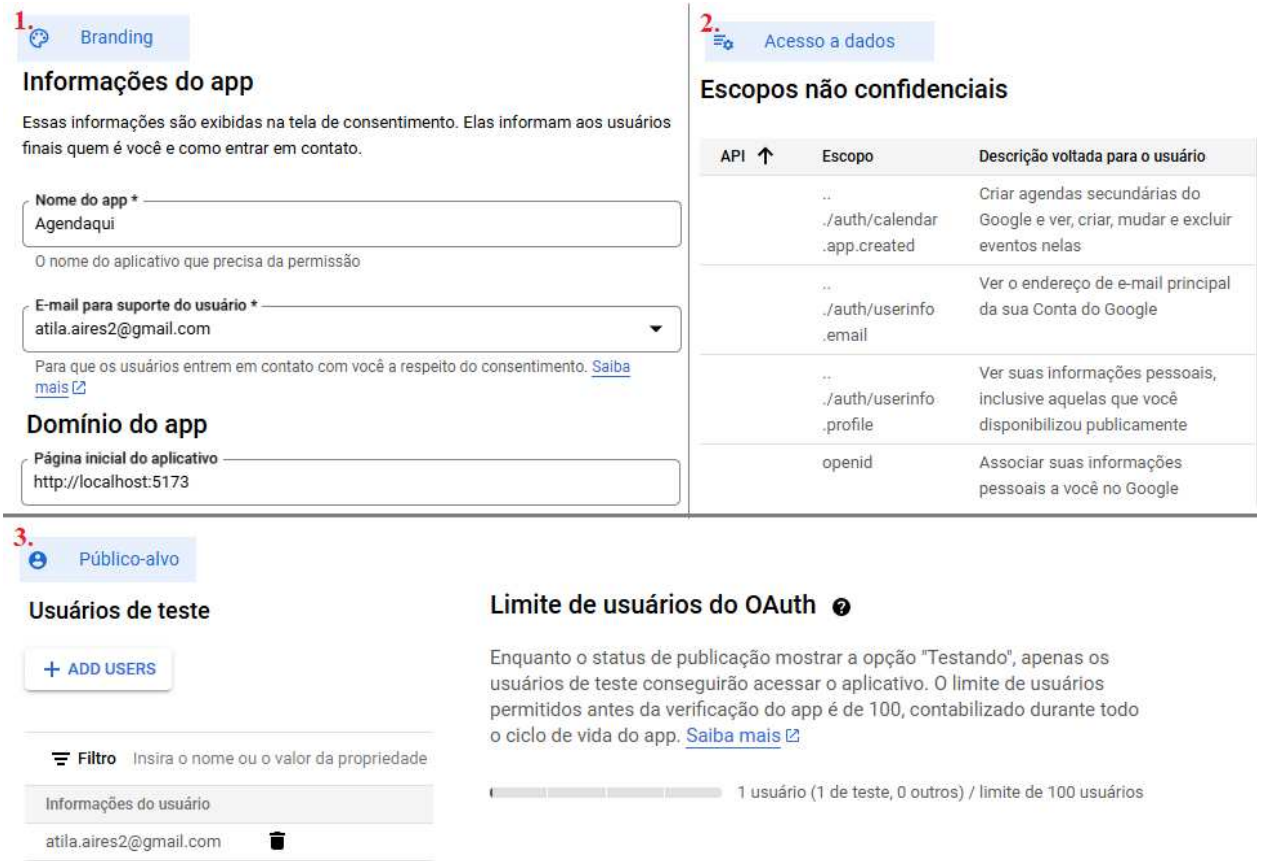
Algo importante a se notar, é que devido ao curto tempo de desenvolvimento, foi optado por inserir elementos de frameworks e bibliotecas externas dentro dos modelos internos, ao invés de criar adaptadores intermediários para estabelecer as comunicação entre esses componentes. Isso fere o princípio de independência da arquitetura, porém foi necessário para garantir o foco do trabalho nos casos de uso e que as funcionalidades fossem entregues no tempo previsto.

A camada de casos de uso foi implementada dentro das pastas logic e db para encapsular a lógica de negócio, garantindo que as regras do sistema estivessem bem definidas e desacopladas de detalhes técnicos. Essa abordagem permite que a lógica do Agendaqui seja reutilizável e independente de *frameworks* ou bibliotecas externas. Já os adaptadores de interface foram responsáveis por lidar com a comunicação entre a API e as camadas externas, como o *frontend* e serviços de mensageria, ela é responsável por traduzir a mensagem de um contexto interno para um externo, e vice-versa, sendo utilizado pelos Endpoints.

A separação das responsabilidades dentro dessa estrutura permitiu que cada parte do sistema fosse desenvolvida e mantida de maneira independente, promovendo baixo acoplamento e alta coesão. Por fim, a API foi documentada utilizando Swagger, ferramenta já integrada ao FastAPI que facilita a visualização das rotas e a execução de requisições para validação dos endpoints. Com essa abordagem, o *backend* ficou preparado para integração com o *frontend* e outras camadas da aplicação.

A **autenticação da plataforma** Agendaqui foi implementada utilizando **OAuth 2.0 do Google**, garantindo maior segurança e eliminando a necessidade de armazenar senhas e realização de cadastros. Como ilustrado na Figura 11, a configuração envolveu a definição da tela de consentimento, a escolha dos escopos necessários e o cadastro de usuários de teste. O processo segue um fluxo onde o usuário é redirecionado para a página de login do Google e concede permissão ao aplicativo, um exemplo disso é visto na figura 12, em seguida, recebe um token JWT gerado pelo *backend*. Esse token é utilizado para armazenar dados importantes do usuário e proteger rotas privadas, assegurando que apenas usuários autenticados possam interagir com os serviços da aplicação.

Figura 11 – Configuração do OAuth da Google



Fonte: Fornecido pelo autor.

Figura 12 – Configuração do JWT da Google

Fonte: Fornecido pelo autor.

Além do login seguro, foi realizada uma integração simplificada com a API do Google Calendar, permitindo a sincronização automatizada dos agendamentos relacionados aos usuários com o calendário Google, assim, sempre que o usuário desejar, ele pode espelhar a agenda do Agendaqui com a do Google Agendas.

O sistema de notificações foi implementado para garantir que clientes e prestadores sejam informados sobre a criação e o cancelamento de agendamentos. Sempre que um agendamento é registrado ou removido, uma notificação é gerada no banco de dados e um e-mail automático é enviado aos usuários envolvidos, contendo os detalhes do serviço, data e horário do compromisso. Como ilustrado na Figura 13, o e-mail é formatado para apresentar de forma clara as informações do agendamento, proporcionando uma comunicação eficiente e reduzindo a chance de esquecimentos.

Figura 13 – Exemplo de e-mail de notificação

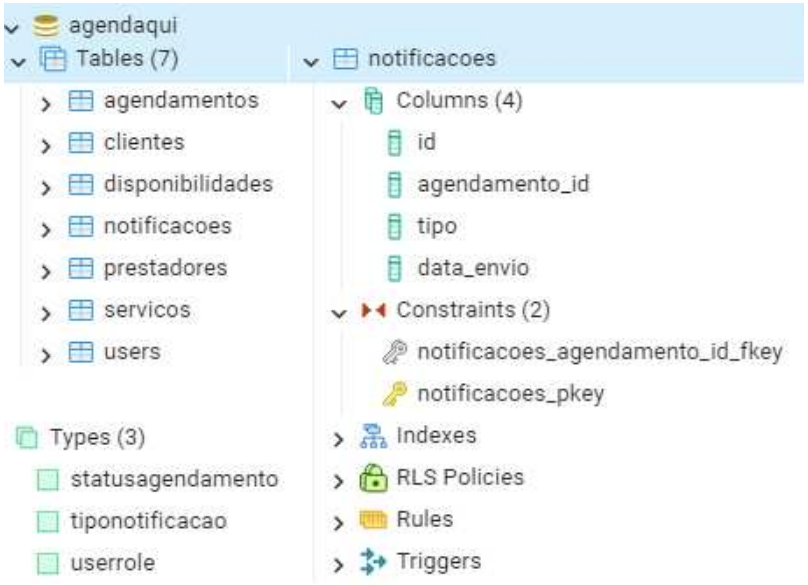
Fonte: Fornecido pelo autor.

4.3 Implementação do Banco de Dados

A implementação do banco de dados no Agendaqui foi planejada para garantir a persistência eficiente das informações, seguindo um modelo relacional baseado em PostgreSQL. A estrutura do banco foi definida conforme a modelagem proposta na seção 3.2.4, garantindo que as entidades e suas relações refletissem corretamente as regras de negócio da aplicação.

O desenvolvimento do banco seguiu um processo estruturado, iniciando com a modelagem das tabelas com base no diagrama de classes da UML definido na figura 9. Cada entidade do sistema, como Usuário, Prestador, Cliente, Serviço, Agendamento e Notificação, foi convertida em uma tabela com seus respectivos atributos e chaves primárias. Para garantir a integridade referencial, foram estabelecidas chaves estrangeiras conectando os relacionamentos entre as tabelas, todos esses detalhes foram implementados no banco do PostgresSQL 16, os resultados podem ser vistos na figura 14.

Figura 14 – Configuração dos Banco pgAdmin



Fonte: Fornecido pelo autor.

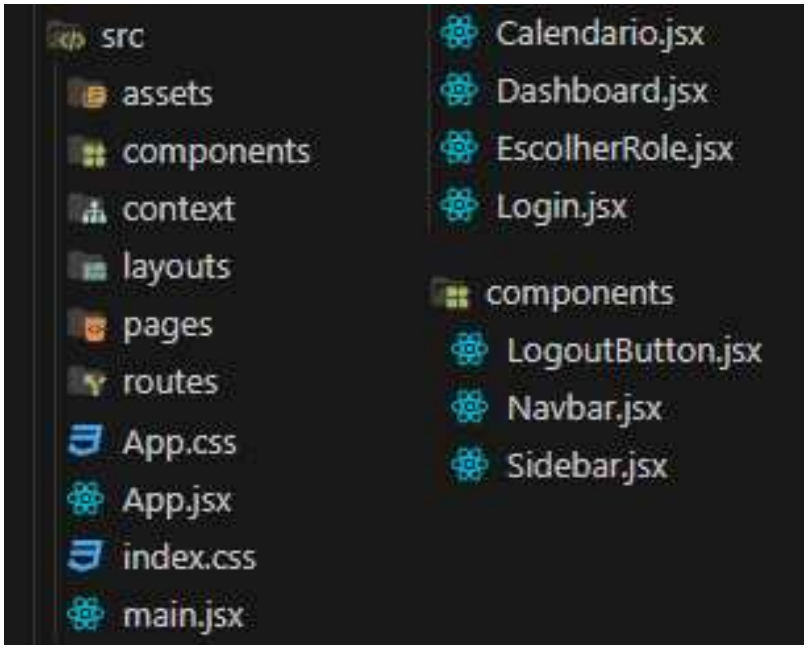
Para manipulação dos dados no *backend*, foi utilizado o SQLAlchemy, uma biblioteca ORM (Object-Relational Mapping) que permitiu a integração entre o banco de dados e a aplicação. Com essa abordagem, foi possível realizar operações como inserção, atualização e consulta de registros sem a necessidade de escrever SQL puro, garantindo maior padronização, flexibilidade e segurança no acesso aos dados.

4.4 Desenvolvimento do Frontend

O desenvolvimento do *frontend* do Agendaqui seguiu as diretrizes estabelecidas na fase de planejamento, garantindo uma interface intuitiva e responsiva para os usuários. A primeira etapa consistiu na configuração do ambiente, utilizando o Vite como ferramenta principal para otimizar a performance do projeto.

A organização do código foi estruturada para manter a separação de responsabilidades, garantindo um código limpo e de fácil manutenção. A estrutura das pastas foi planejada para dividir os módulos principais da aplicação, conforme ilustrado na Figura 15.

Figura 15 – Exemplificação da estrutura de código do *Frontend*



Fonte: Fornecido pelo autor.

Essa organização permitiu um desenvolvimento modular e escalável, facilitando a adição de novas funcionalidades e garantindo a manutenção eficiente do código.

Para o gerenciamento de rotas, foi utilizado o React Router, permitindo uma navegação fluida entre as páginas da plataforma. As principais telas desenvolvidas foram:

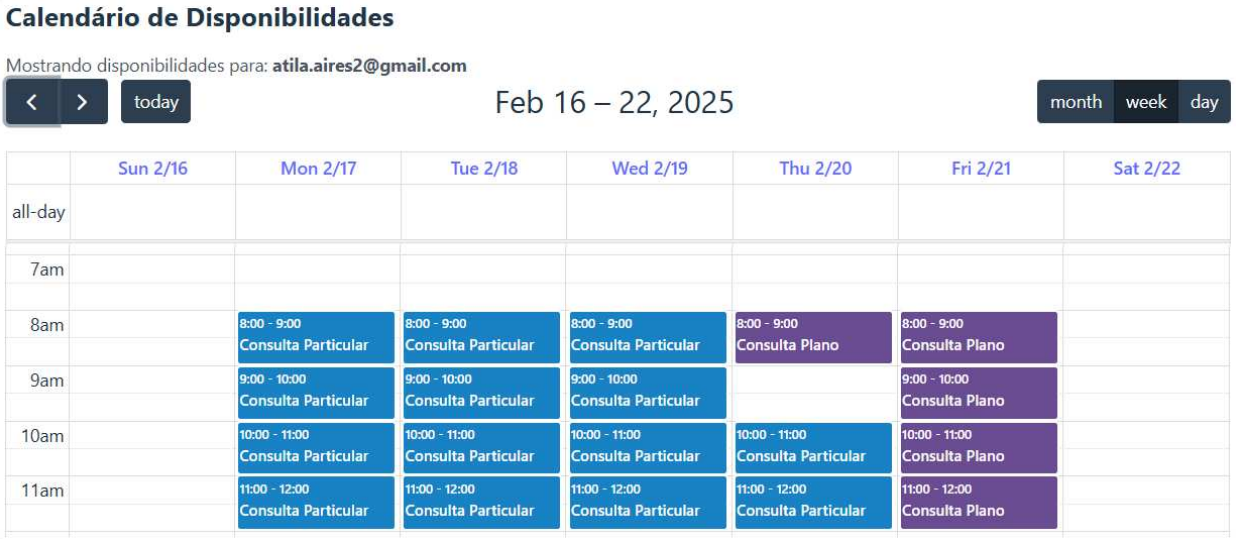
- **Tela de Login:** Responsável pela autenticação dos usuários, integrada ao *backend*, utilizando OAuth 2.0 e JWT da Google, figura 16.
- **Dashboard:** Apresenta um resumo dos dados do usuário e as opções de navegação da plataforma.

- **Calendário (Cliente):** Permite que os clientes visualizem os horários disponíveis em um calendário interativo e realizem agendamentos de forma intuitiva, exemplo na figura 17.
- **Calendário (Prestador):** Permite que os prestadores visualizem os horários disponíveis em um calendário interativo e os apague.
- **Tela de Servicos:** Interface para que prestadores de serviço possam cadastrar e deletar serviços que eles estão ofertando, exemplo na figura 18.
- **Tela de Agendamento (Prestador):** Interface para que prestadores de serviço possam visualizar e configurar horários disponíveis, exemplo na figura 19.
- **Tela de Agendamento (Cliente):** Interface para que os clientes possam visualizar e deletar seus horários agendados.

Figura 16 – Página de Login

Fonte: Fornecido pelo autor.

Figura 17 – Página de Calendário Interativo

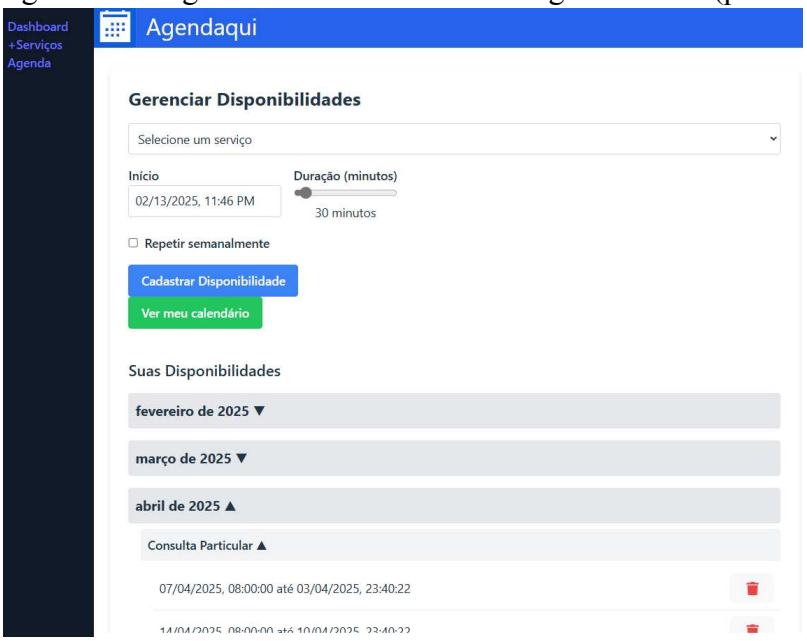


Fonte: Fornecido pelo autor.

Figura 18 – Página de cadastro de Serviço

Fonte: Fornecido pelo autor.

Figura 19 – Página de Gerenciamento de Agendamento (prestador)



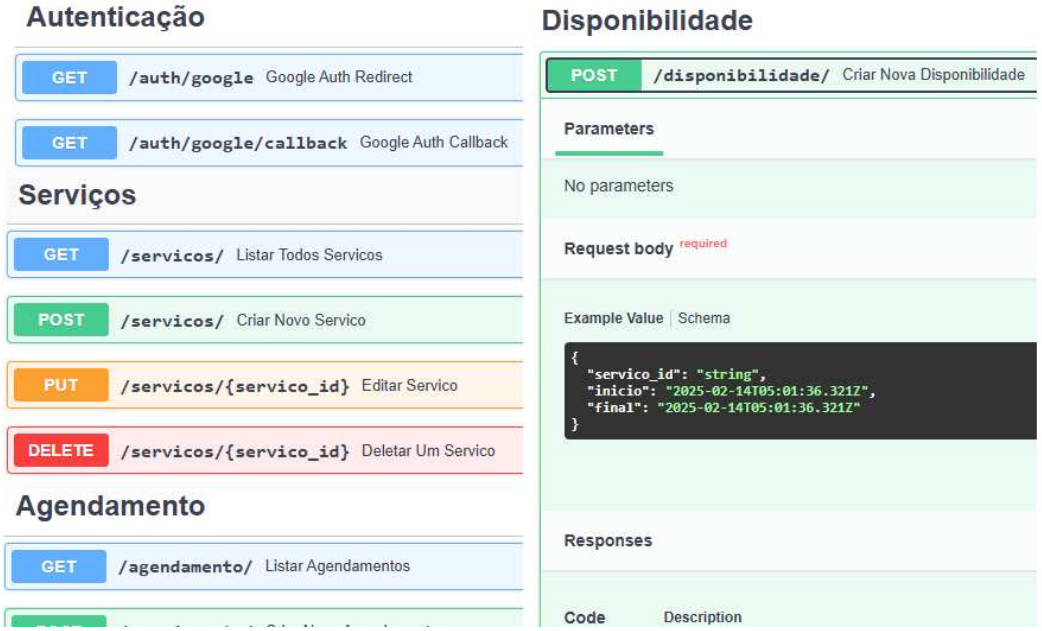
Fonte: Fornecido pelo autor.

A comunicação entre o *frontend* e o *backend* foi realizada por meio de chamadas à API RESTful, garantindo uma interação eficiente e dinâmica entre as camadas da aplicação. Para isso, foi utilizado o Axios, uma biblioteca que facilita o envio de requisições HTTP assíncronas, permitindo que o *frontend* obtenha e manipule dados do servidor de maneira otimizada. O uso do Axios também permitiu a configuração centralizada de headers de autenticação e tratamento de erros, tornando a integração entre *frontend* e *backend* mais segura e confiável.

4.5 Integração e testes

A fase de integração e testes foi conduzida por meio de verificações manuais, utilizando ferramentas como Postman e Swagger para validar o funcionamento dos endpoints da API. Durante o desenvolvimento, requisições HTTP foram testadas diretamente no Postman, permitindo verificar a resposta do servidor, analisar os dados retornados e garantir que as operações de criação, leitura, atualização e exclusão (CRUD) funcionassem corretamente. Além disso, o Swagger, integrado à API, foi utilizado para visualizar e testar as rotas disponíveis, facilitando a inspeção das requisições e parâmetros esperados. Na Figura 20, é possível observar a documentação gerada pelo Swagger, que fornece uma interface interativa para explorar os endpoints, visualizar as estruturas de entrada e saída e executar testes diretamente na aplicação, agilizando o processo de validação do *backend*.

Figura 20 - Endpoints na interface do Swagger



Fonte: Fornecido pelo autor.

Para validar a experiência de uso da plataforma na prática, foi conduzida uma simulação do fluxo completo de um profissional da área médica utilizando o sistema. Durante essa simulação, foram realizados cadastros de serviços e disponibilidades, além da realização de agendamentos por parte de clientes fictícios, com o objetivo de avaliar a fluidez da navegação, a precisão das permissões de acesso e o funcionamento do sistema de notificações. A validação foi estruturada com base nos seguintes critérios:

- **Correção da navegação entre as telas:** Teste das interações entre as interfaces da aplicação, garantindo que todas as funcionalidades estivessem acessíveis e operando corretamente.
- **Validação da lógica de permissões:** Garantia de que apenas prestadores de serviço poderiam registrar serviços e disponibilidades, enquanto apenas clientes poderiam efetuar agendamentos, impedindo acessos indevidos a funcionalidades restritas.
- **Análise do sistema de notificações:** Verificação da emissão automática de e-mails de confirmação e cancelamento de agendamentos, assegurando que tanto o prestador quanto o cliente recebessem as informações corretamente.

Assim, foram identificadas e aplicadas melhorias para corrigir falhas pontuais e otimizar a experiência do usuário, tornando o sistema mais robusto e confiável. Embora a validação tenha sido conduzida de forma manual, com simulações práticas, futuros trabalhos podem incluir testes formais com usuários reais, possibilitando a coleta de métricas quantitativas e feedbacks qualitativos sobre a experiência de uso da plataforma.

Com o intuito de disponibilizar a base do projeto para futuras melhorias e estudos, o código-fonte da aplicação, incluindo a implementação dos testes e a documentação dos endpoints gerada pelo Swagger, encontra-se disponível no **repositório do GitHub**: https://github.com/Atila-Nobrega/Agendaqui.

A construção da plataforma Agendaqui foi conduzida com ênfase na aplicação de princípios de arquitetura limpa e boas práticas de desenvolvimento, garantindo que sua estrutura fosse organizada, modular e de fácil manutenção. A adoção da Clean Architecture permitiu a separação clara entre as camadas do sistema, promovendo maior flexibilidade para futuras expansões sem comprometer a integridade do software.

5 CONCLUSÃO

O desenvolvimento da plataforma Agendaqui buscou a construção de um sistema eficiente e funcional para o gerenciamento de agendamentos, permitindo que usuários organizem compromissos de maneira intuitiva e automatizada. A adoção de uma arquitetura modular garantiu a separação clara entre as diferentes camadas do sistema, facilitando sua manutenção e possibilitando futuras expansões sem comprometer a integridade do código.

Além disso, a implementação baseada nos princípios da Clean Architecture contribuiu para um design de software bem estruturado, reduzindo o acoplamento entre os componentes, porém, seguir todas as regras de arquitetura limpa provou ser uma dificuldade por conta do curto tempo de desenvolvimento. A Agendaqui como prova de conceito permitiu que a arquitetura fosse projetada alinhada a boas práticas de desenvolvimento de software, porém ao longo do seu desenvolvimento notamos que seguir o as práticas a risca seria custoso demais e comprometimento foram necessário para que todos os casos de uso fossem implementados.

Durante o desenvolvimento, testes manuais foram realizados utilizando ferramentas como Postman e Swagger, permitindo validar a comunicação entre *frontend* e *backend*, além disso, foi simulada a experiência de um profissional médico oferecendo consultas para avaliar o fluxo de agendamentos e o funcionamento das notificações. Apesar desses testes terem sido muito valiosos na implementação das funcionalidades, a falta de testes unitários e de integração podem prejudicar a manutenção do software, além de que não foram realizadas validações com usuários reais, o que poderia fornecer informações adicionais sobre a usabilidade e possíveis melhorias na experiência do usuário.

Por fim, algumas funcionalidades inicialmente planejadas, como notificações automáticas para lembrete de compromissos e integração com o Google Calendar, foram simplificadas ou deixadas para trabalhos futuros.

De acordo com os resultado obtidos para a plataforma, seguem algumas sugestões para trabalhos futuros:

- ➢ Separação de frameworks dos modelos internos e criação de adaptadores.
- ➢ Inclusão de testes unitário e de integração.
- ➢ A adoção de ferramentas como Docker e Kubernetes pode otimizar a implantação e gerenciamento da plataforma.

REFERÊNCIAS

BECK, K. **Extreme Programming Explained: Embrace Change.** Addison-Wesley, 2000.

CAYIRLI, T.; VERAL, E. Outpatient scheduling in health care: A review of literature. **Production and Operations Management**, v.12, p. 519-549, 2009. Disponível em:https://www.researchgate.net/publication/229881171_Outpatient_scheduling_in_health_care_A_review_of_literature. Acesso em: 3 fevereiro 2025.

CRISPIN, Mark. Internet Message Access Protocol - Version 4rev1. RFC 3501, Março 2003. Disponível em: https://datatracker.ietf.org/doc/html/rfc3501. Acesso em: 2 março 2025.

DOCTORALIA; FEEGOW. Panorama das Clínicas e Hospitais 2024. Disponível em: https://pro.doctoralia.com.br/recursos-gratuitos/pesquisa/panorama-das-clinicas-e-hospitais-2024/form. Acesso em: 1 mar. 2025.

FIELDING. **Architectural Styles and the Design of Network-based Software Architectures**. 2000. Dissertação. University Of California, Irvine. Disponível em: https://ics.uci.edu/~fielding/pubs/dissertation/top.htm. Acesso em: 3 fevereiro 2025.

FIELDING et al. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, 1999. Disponível em: https://www.rfc-editor.org/rfc/rfc2616. Acesso em: 3 fevereiro 2025.

FIELDING, et al. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, 2014. Disponível em: https://www.rfc-editor.org/rfc/rfc7231. Acesso em: 3 fevereiro 2025.

GOOGLE CLOUD. Introdução ao OAuth. Disponível em: https://cloud.google.com/apigee/docs/api-platform/security/oauth/oauth-introduction?hl=pt-br. Acesso em: 16 de Janeiro de 2025.

KASENDA, R. et al. The Role and Evolution of Frontend Developers in the Software Development Industry. **Jurnal Syntax Admiration**, v. 5, n.11, 2024. Disponível em: https://www.researchgate.net/publication/386369600_The_Role_and_Evolution_of_Frontend_Developers_in_the_Software_Development_Industry. Acesso em: 3 fevereiro 2025.

KLENSIN, John. Simple Mail Transfer Protocol. RFC 5321, Outubro 2008. Disponível em: https://datatracker.ietf.org/doc/html/rfc5321 Acesso em: 2 março 2025.

LEVITIN, Daniel J. **A Mente Organizada: Como Pensar com Clareza na Era da Sobrecarga de Informação**. Rio de Janeiro: Objetiva, 2015.

LISKOV, B. Data abstraction and hierarchy. **ACM SIGPLAN Notices**, v.23, n.5, 1988.

MARTIN, R. **Arquitetura Limpa: o Guia do Artesão para Estrutura e Design de Software**. Rio de Janeiro: Alta Books, 2019.

MARTIN, R. Solid Relevance. **The Clean Code Blog**, 2012. Disponível em: https://blog.cleancoder.com/uncle-bob/2020/10/18/Solid-Relevance.html. Acesso em: 3 fevereiro 2025.

MARTIN, R. The Clean Architecture. **The Clean Code Blog**, 2012. Disponível em:https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html. Acesso em: 3 fevereiro 2025.

MEYER, B. **Object-Oriented Software Construction**. Nova Iorque: Prentice Hall, 1988, p.23.

MYERS, J.; ROSE, M. Post Office Protocol - Version 3. RFC 1939, Maio 1996. Disponível em: https://www.ietf.org/rfc/rfc1939.txt. Acesso em: 2 março 2025.

OLIVEIRA, Fabricio V..; FILHO, Reisoli B.; VIEIRA, Kelmara M. Custo econômico do absenteísmo de consultas: estudo de caso em um ambulatório de atenção psicossocial. **Research, Society and Development**, v.9, n.7, 2020. Disponível em: https://rsdjournal.org/index.php/rsd/article/view/4066/3568. Acesso em: 11 março 2025.

O QUE É STMP? **AWS**. Disponível em: https://aws.amazon.com/what-is/smtp/. Acesso em: 3 fevereiro 2025.

ORIGAMI. **Github**. Origami is a pure Ruby library to parse, modify and generate PDF documents. Disponível em: https://github.com/gdelugre/origami. Acesso em: 3 fevereiro 2025.

PARGAONKAR, Shravan. Enhancing software quality in architecture design: a survey-based approach. *International Journal of Scientific and Research Publications*, v. 13, n. 08, 2023. ISSN 2250-3153. Disponível em: http://dx.doi.org/10.29322/IJSRP.13.08.2023.p14014. Acesso em: 09 março 2025.

PIKKUMÄKI, T. **Comparison of Monolithic, Micro-service, and Cloud Development**. 2023. Tese. JAMK University of Applied Sciences, Jyväskylä, Finlândia, 2023. Disponível em: https://www.theseus.fi/bitstream/handle/10024/795180/MastersThesis_Pikkum%c3%a4ki_Toni.pdf?sequence=2&isAllowed=y. Acesso em: 3 fevereiro 2025.

RIABOV, V. SMTP (Simple Mail Transfer Protocol). In: **The Handbook of Computer Networks**. John Wiley & Sons, p.338-406, 2007.

SMTPLIB — SMTP protocol client**. Python**. Disponível em: https://docs.python.org/3/library/smtplib.html. Acesso em: 3 fevereiro 2025.

TANG, ANTONY, et al. Human Aspects in Software Architecture Decision Making - A Literature Review. 2017. Disponível em: www.researchgate.net/publication/316266811_Human_Aspects_in_Software_Architecture_Decision_Making_-_A_Literature_Review

TOKER, K. et al. A Solution to Reduce the Impact of Patients' No-Show Behavior on Hospital Operating Costs: Artificial Intelligence-Based Appointment System. **Healthcare (Basel)**, v. 12, n.21, p.10, 2024. Disponível em: https://pmc.ncbi.nlm.nih.gov/articles/PMC11545362/. Acesso em: 3 fevereiro 2025.

TURBOSMTP. What is SMTP and why it is fundamental in email sending. s. d. Disponível em: https://serversmtp.com/what-is-smtp/. Acesso em: 12 jan. 2025.