UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Hisashi It Yamaguti

Integração de Interface Conversacional (Chatbot) a uma Aplicação Web para Gestão de Consultas Médicas

Uberlândia, Brasil

2025

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Hisashi It Yamaguti

Integração de Interface Conversacional (Chatbot) a uma Aplicação Web para Gestão de Consultas Médicas

Trabalho de conclusão de curso apresentado à Faculdade de Engenharia Mecânica da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção do título de Bacharel em Engenharia Aeronáutica.

Orientador: Profº Drº Fabiano Azevedo Dorça

Universidade Federal de Uberlândia – UFU
Faculdade de Engenharia Mecânica – FEMEC
Bacharelado em Engenharia Aeronáutica

Uberlândia, Brasil
2025

Hisashi It Yamaguti

Integração de Interface Conversacional (Chatbot) a uma Aplicação Web para Gestão de Consultas Médicas

Trabalho de conclusão de curso apresentado à Faculdade de Engenharia Mecânica da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção do título de Bacharel em Engenharia Aeronáutica.

Trabalho aprovado. Uberlândia, Brasil, 29 de Setembro de 2025:

Profº Drº Fabiano Azevedo Dorça
Orientador

Profº Drº Higor Luis Silva
Professor

Profº Drº Maria Adriana Vidigal de Lima
Professor

Uberlândia, Brasil

2025

Resumo

Este trabalho apresenta o desenvolvimento de um ecossistema para agendamento de consultas composto por: (i) um website no qual profissionais (médicos) e pacientes visualizam compromissos, criam disponibilidades e realizam marcações; e (ii) um chatbot no WhatsApp que executa a autenticação baseada em telefone. Na interação do usuário com o chatbot, a WhatsApp Cloud API gera um payload JSON contendo o número de telefone, garantindo a identificação inequívoca do contato. A partir disso, o chatbot registra o número no sistema e envia um link com código de acesso único para o painel correspondente ao perfil do usuário (médico ou paciente), liberando o acesso ao site apenas após a verificação.

O website (https://medcoqueiral.com.br), desenvolvido em HTML, CSS, JavaScript e PHP e hospedado na LocaWeb, funciona como camada de interface, encaminhando requisições HTTP ao back-end em Python (Flask). Esse back-end, inicialmente exposto ao público por um túnel seguro via Ngrok para prototipação, foi posteriormente migrado para a infraestrutura do Google Cloud. O banco de dados PostgreSQL é compartilhado entre site e chatbot, armazenando contas, registros de consultas e assegurando a unicidade de e-mails e telefones. A solução automatiza o envio e o recebimento de mensagens via WhatsApp Cloud API (HTTP), gerenciando o fluxo conversacional e a persistência de dados.

O objetivo principal é simplificar o agendamento entre clientes e profissionais, bem como o cadastro e a entrada segura no sistema, além de facilitar consultas (queries) aos agendamentos para apoiar a organização do profissional. Embora concebida para o contexto médico, a solução é escalável e aplicável a outras áreas que demandam marcação de compromissos. O sistema foi projetado em conformidade com as diretrizes da Meta e com a legislação brasileira, preservando a integridade dos dados e evitando práticas sujeitas a sanções. A adoção de um banco de dados estruturado e da linguagem Python também prepara o terreno para trabalhos futuros em análise de dados e construção de sistemas de recomendação, aproveitando o ecossistema robusto de bibliotecas de IA em Python.

Palavras-chave: agendamento; chatbot; WhatsApp; Flask; PostgreSQL; autenticação; sistemas web; Google Cloud.

Sumário

1 Introdução

O WhatsApp Business é amplamente utilizado por profissionais para organizar atendimentos e manter contato com clientes. Quando esse processo é conduzido manualmente pelo próprio profissional ou por um funcionário, são comuns falhas operacionais que comprometem a qualidade do serviço, tais como:

1. esquecer de registrar um atendimento no sistema;
2. sobrepor um atendimento a outro já agendado;
3. registrar data e/ou horário incorretos;
4. deixar de enviar lembretes quando a data do atendimento se aproxima.

Este trabalho apresenta um ecossistema de agendamento que mitiga tais fragilidades por meio de dois componentes integrados: (i) um website em que profissionais e pacientes visualizam compromissos, criam disponibilidades e realizam marcações; e (ii) um *chatbot* no WhatsApp responsável pela autenticação baseada em telefone e pela automação de fluxos críticos. Na interação com o *chatbot*, a WhatsApp Cloud API fornece o número do usuário em um *payload* `JSON`, garantindo sua identificação inequívoca. A partir disso, o sistema registra com segurança o telefone, gera um link com código de acesso único e direciona o usuário ao painel apropriado (médico ou paciente), liberando o acesso ao site apenas após a verificação.

A consistência do agendamento e a prevenção de erros são asseguradas por um banco de dados relacional (`PostgreSQL`), que centraliza contas e registros de consultas, impõe regras de integridade (unicidade de e-mail e telefone) e validações de data/horário, impedindo conflitos de agenda. O *back-end*, implementado em `Python` com `Flask`, orquestra a lógica de negócios, processa mensagens recebidas via *webhook* e expõe *endpoints* `HTTP` consumidos pelo website. O site (`https://medcoqueiral.com.br`), desenvolvido em `HTML`, `CSS`, `JavaScript` e `PHP` e hospedado na LocaWeb, atua como camada de interface, emitindo requisições `HTTP` ao *back-end* para executar consultas de alto nível ao banco, sem exigir que o usuário conheça `SQL`. Na prototipação, a comunicação entre o ambiente local e a WhatsApp Cloud API foi viabilizada por um túnel seguro via `Ngrok`; posteriormente, a infraestrutura foi migrada para a nuvem (Google Cloud), preservando escalabilidade e disponibilidade.

O sistema foi concebido em conformidade com as diretrizes da Meta e com a legislação brasileira, preservando a integridade dos dados e evitando práticas sujeitas a sanções. Embora validada inicialmente no contexto médico, a solução é escalável e aplicável a diferentes áreas que dependem de marcação de compromissos. A adoção de `Python` e de um banco estruturado também abre caminho para trabalhos futuros em análise de dados e sistemas de recomendação, apoiados pelo ecossistema robusto de bibliotecas de IA.

1.1 Justificativa

A opção por construir o *chatbot* e o website com `Python`, `Flask` e `PostgreSQL` responde à crescente demanda por soluções sob medida que extrapolam o escopo oferecido por *Business Solution Providers* (BSPs), como fluxos de autenticação por telefone, lembretes automatizados, consultas de alto nível a dados transacionais e integrações específicas (por exemplo, com serviços de pagamento e de IA). Dominar o uso de APIs — como a WhatsApp Cloud API — viabiliza a construção de aplicações orientadas a APIs (*API-driven applications* (ZENKINS, 2025)), nas quais serviços externos são combinados de forma segura para compor funcionalidades de maior valor agregado.

Ressalta-se que ferramentas e *frameworks* evoluem rapidamente. Embora o `Flask` tenha cumprido papel central na orquestração de mensagens e no tratamento de requisições `HTTP`, alternativas como `FastAPI` ganham adoção. Ainda assim, o conhecimento sólido do protocolo `HTTP`, de *webhooks*, de *endpoints* REST e do ciclo de requisições/respostas permanece essencial e transferível a futuras evoluções tecnológicas. Essa base conceitual garante longevidade ao investimento em engenharia, independentemente da troca de ferramentas.

1.2 Objetivos

O objetivo principal deste projeto é desenvolver um ecossistema de gerenciamento de consultas entre médicos e pacientes, com foco em simplicidade de uso, segurança e integridade dos dados. Para alcançá-lo, estabelecem-se os seguintes objetivos específicos:

1. Criar e hospedar na nuvem um website operacional 24/7 que permita a médicos realizar consultas de alto nível ("queries") sobre suas agendas, sem exigir conhecimento de `SQL`.

2. Projetar e implementar um banco de dados relacional em `PostgreSQL` para armazenar dados de médicos, pacientes e consultas, com regras de integridade e unicidade (e-mail e telefone).

3. Desenvolver uma API que possibilite a visualização das consultas pelo médico, a partir do domínio `https://medcoqueiral.com.br`, mesmo quando o banco esteja hospedado em servidor distinto do website.

4. Implementar um *chatbot* no WhatsApp que registre consultas entre médicos e pacientes após confirmação de pagamento validado pela API da Stripe, persistindo esses registros no banco de dados.

5. Criar um fluxo de registro de usuários (médicos e pacientes), com autenticação via telefone iniciada no *chatbot*, que gera um link de acesso único direcionando ao website para conclusão do cadastro.

6. Criar o perfil *Admin*, com privilégios para criar códigos de autenticação para registro de médicos e para gerenciar (criar/editar/excluir) perfis de médicos e pacientes.

7. Definir políticas distintas de cadastro: pacientes podem se registrar sem código de autenticação; médicos somente mediante código de autenticação emitido pelo perfil *Admin*.

8. Iniciar o fluxo de registro no *chatbot* do WhatsApp, que gera um link de acesso único para o website `https://medcoqueiral.com.br`, onde o usuário conclui o cadastro e acessa seu painel.

Neste trabalho, "queries" referem-se a consultas ao banco de dados `PostgreSQL` por meio de operações expostas pela API, sem exigir que o usuário escreva `SQL`.

1.3 Metodologia

Esta seção descreve a abordagem metodológica adotada para o desenvolvimento do ecossistema e o cumprimento dos objetivos definidos. Dada a complexidade técnica, a implementação do *chatbot* foi fortemente apoiada pelo tutorial do cientista de dados Dave Ebbelaar (EBBELAAR, 2023a) e pelo código disponibilizado em seu repositório no GitHub (EBBELAAR, 2023b). Esse material serviu como base prática para o tratamento de *webhooks*, formatação de mensagens e manipulação de *payloads* `JSON` da WhatsApp Cloud API. Ressalta-se que a modelagem do banco de dados e o desenvolvimento do website foram conduzidos de forma independente.

Foram consultados os documentos oficiais da Meta para configuração do *webhook* da WhatsApp Cloud API e entendimento do formato das requisições `HTTP` (`JSON`) (META, 2025). Durante a fase de prototipação, empregou-se o `Ngrok` para expor com segurança o servidor local à internet, viabilizando testes de ponta a ponta; posteriormente, migrou-se a infraestrutura para nuvem (Google Cloud), mantendo disponibilidade e escalabilidade.

Para o projeto de dados, realizou-se estudo aprofundado do referencial clássico de bancos de dados (SILBERSCHATZ; KORTH; SUDARSHAN, 2010), com ênfase na construção do modelo entidade–relacionamento, mapeamento para o modelo relacional e fundamentos da programação web (`HTML`, `CSS` e protocolo `HTTP`). Complementarmente, foram estudados conceitos de REST, *endpoints*, *webhooks* e o envio de requisições `HTTP` (GET e POST) com `Flask`, com apoio de material introdutório a APIs (TECLADO, s.d.). Esse conhecimento embasou a criação da API consumida pelo website hospedado na LocaWeb, que executa operações sobre um banco de dados em servidor distinto.

Após a revisão teórica e técnica, o sistema foi desenvolvido conforme descrito a seguir.

1.3.1 Definição dos requisitos do sistema

Foram levantados requisitos funcionais e não funcionais:

- **Funcionais:** registro de médicos, pacientes e consultas; listagem de consultas futuras; agendamento automatizado via *website*; acesso diferenciado por perfil (paciente, médico, administrador); lembretes automáticos.
- **Não funcionais:** persistência dos dados em banco relacional; integração de website e chatbot via requisições `HTTP`; escalabilidade e alta disponibilidade; segurança de informações sensíveis; conformidade com diretrizes da Meta e legislação brasileira; integração com APIs externas (WhatsApp Cloud API e Stripe).

1.3.2 Arquitetura baseada em camadas

A arquitetura de software adotada para o desenvolvimento deste projeto foi o modelo de três camadas (*three-tier architecture*), conforme segue o livro de (RICHARDS; FORD, 2020). Essa abordagem organiza o sistema em níveis lógicos e físicos distintos, promovendo a separação de responsabilidades, o que resulta em um sistema mais organizado, escalável e de fácil manutenção. A estrutura é composta pelas seguintes camadas: Apresentação, Negócio e Dados.

Camada de Apresentação (Presentation Layer) Esta é a camada mais externa, responsável pela interface com o usuário (*front-end*). É por meio dela que o usuário final interage com o sistema, seja para inserir dados ou para visualizar informações, como o histórico e o agendamento de suas próximas consultas. Para este projeto, a camada de apresentação foi implementada em duas frentes: um website e uma interface conversacional (*chatbot*) integrada ao WhatsApp, visando ampliar a acessibilidade e a conveniência para o usuário.

Camada de Negócio (Business Layer) Atuando como o centro lógico ('o cérebro') da aplicação (*back-end*), esta camada intermediária é responsável por toda a lógica e pelas regras de negócio que governam o sistema. Ela processa as solicitações recebidas da Camada de Apresentação, executa as operações e validações necessárias — como verificar a disponibilidade de horários para uma consulta ou validar dados de um agendamento — e coordena o fluxo de informações entre as outras camadas. Essencialmente, é nesta camada que a inteligência do sistema reside.

Camada de Dados (Data Layer) Sendo a camada mais interna, sua função é gerenciar a persistência e o acesso aos dados. Ela abstrai a complexidade do banco de dados e é a única responsável por manipular as informações armazenadas, realizando operações como criação, leitura, atualização e exclusão de registros (CRUD). Ao centralizar o acesso aos dados, esta camada garante a integridade, a consistência e a segurança das informações do sistema, como cadastros de usuários e registros de consultas.

1.3.3 Projeto do banco de dados

O projeto do banco de dados ocorreu em três níveis:

1. **Modelo conceitual:** elaboração do Diagrama Entidade-Relacionamento (DER) com entidades Usuário, Médico, Paciente e Consulta, além de relacionamentos e cardinalidades.

2. **Modelo lógico:** mapeamento para o modelo relacional com definição de chaves primárias e estrangeiras, restrições de unicidade (e-mail e telefone) e regras de integridade referencial.

3. **Modelo físico:** mapeamento para o Database Markup Language (DBML) a qual envolve criação das tabelas e índices, escolha de tipos de dados apropriados e *constraints* para garantir desempenho e consistência, incluindo validações de datas/horários e prevenção de conflitos de agenda.

1.3.4 Desenvolvimento do back-end

O *back-end* foi implementado em `Python` com o *framework* `Flask`, responsável por:

- expor *endpoints* `HTTP` consumidos pelo website;

- processar *payloads* `JSON` recebidos por *webhooks* da WhatsApp Cloud API;

- implementar regras de negócio para cadastro, autenticação por telefone, agendamento e lembretes;

- orquestrar a persistência de dados no `PostgreSQL` com transações seguras.

1.3.5 Integração com a WhatsApp Cloud API

A integração foi realizada via *webhook* para eventos em tempo real. O fluxo adotado:

1. o usuário envia mensagem ao número do WhatsApp Business;
2. a API do WhatsApp encaminha um `JSON` estruturado para o *webhook*;
3. o servidor `Flask` interpreta o *payload* `JSON`, identifica a intenção do usuário e executa a lógica apropriada (por exemplo, horário de atendimento do consultório, autenticar acesso);
4. os dados são persistidos no `PostgreSQL` e uma resposta é enviada ao usuário dentro da janela de atendimento (24 horas), incluindo, quando aplicável, um link de acesso único para o painel no website.

Também foram implementados a verificação do *webhook* (token e *challenge*), tratamento de erros e políticas de idempotência para evitar registro duplicado de eventos em caso de reentrega pela API.

1.3.6 Desenvolvimento do front-end

Foi implementada uma interface web no domínio `https://medcoqueiral.com.br`, com foco na usabilidade e na clareza das informações:

- painéis específicos para médicos e pacientes, acessados por link com código de acesso único enviado pelo *chatbot*;
- visualização de consultas futuras em ordem cronológica, com filtros por data e paciente;
- execução de consultas de alto nível à agenda, onde o `PHP` no servidor web intermedia as requisições `HTTP` para o *back-end* em `Flask` (exposto publicamente via Ngrok na fase de desenvolvimento), abstraindo do usuário a necessidade de conhecimento em `SQL`;
- layout responsivo em `HTML`, `CSS` e `JavaScript`, com integração em `PHP` para orquestração das chamadas à API.

1.3.7 Programação Orientada a Objetos

A aplicação foi estruturada segundo princípios de POO para promover coesão e reduzir acoplamento:

- classe base `Usuario` e especializações `Paciente`, `Medico` e `Administrador`;
- aplicação de abstração, encapsulamento, herança e polimorfismo para facilitar manutenção e evolução.

1.3.8 Testes e validação

Foram realizados testes funcionais e de integração para assegurar a conformidade dos requisitos:

- verificação de *constraints* no `PostgreSQL` (unicidade de e-mail e telefone, integridade referencial, validações de data/horário e prevenção de conflitos de agenda);
- simulação de eventos da WhatsApp Cloud API com ferramentas como `curl`/Postman para validar o processamento de *payloads* e respostas do *webhook*;
- testes de acesso aos painéis via links de código único, garantindo autenticação por telefone e escopo de permissões por perfil;
- checagens de segurança (validação de entrada, tratamento de erros, ocultação de detalhes sensíveis em respostas).

1.3.9 Implantação e infraestrutura

Durante a prototipação, utilizou-se o `Ngrok` para expor com segurança o servidor local e validar o fluxo fim a fim com a WhatsApp Cloud API. Em produção, o *back-end* foi migrado para a nuvem (Google Cloud), preservando escalabilidade e disponibilidade, enquanto o website permaneceu hospedado na LocaWeb como camada de interface. Esse arranjo permite separar preocupações, simplificar o ciclo de implantação e manter a comunicação entre camadas via `HTTP`.

2 Referencial Teórico

O presente capítulo apresenta um levantamento bibliográfico dos fundamentos teóricos necessários para o desenvolvimento do presente trabalho.

2.1 Armazenamento de dados

Para guardar as informações de registro de médico, paciente e as consultas entre paciente e médico, precisou-se utilizar algum mecanismo que permitisse realizar a persistência dos dados em caso de eventual parada do programa ou algo do tipo.

Isso se deve ao fato de que ao se utilizar uma linguagem de programação, os dados são armazenados enquanto o programa estiver funcionando por meio de estruturas de dados, como listas ou dicionários (Python). Contudo, caso o programe pare de funcionar, todos os dados armazenados seja numa lista, seja num dicionário são perdidos ao se reinicializar o programa.

Dessa maneira, para se realizar a persistência, i.e., manutenção dos dados após eventual parada de funcionamento do programa, é necessária a utilização de um mecanismo de armazenamento desses dados, sendo ou arquivos ou Sistema Gerenciador de Banco de Dados (SGBD). Neste trabalho, utilizou-se o Sistema de Gerenciamento de Banco de Dados chamado PostgreSQL, pois ele é o melhor para esse tipo de problema nos dias atuais. Antes do advento desse sistema, usavam-se os Sistemas de Processamento de Arquivos, os quais possuíam várias desvantagens.

2.1.1 Sistema de Processamento de Arquivos

O sistema de processamento de arquivos, anterior ao advento do Sistema de Gerenciador de Banco de Dados (SGBD), era basicamente a utilização de arquivos que podiam ter formato texto, CSV, binário ou outro formato e eram usados para guardar os dados. Cada aplicação de leitura, escrita, retirada, atualização de dado era escrita em alguma linguagem, por exemplo C, e os dados eram armazenados em algum arquivo de formato csv, txt, binário.

2.1.2 Exemplo de Código em C para Leitura de Arquivos

No Apêndice A, apresenta-se um exemplo de código em C mostrando como o arquivo estudantes.txt seria lido e escrito, bem como era feita a restrição de ID único antes do advento do Sistema de Gerenciamento de Banco de Dados o qual se utiliza o conceito de chaves primárias.

Segue o arquivo estudantes.txt que foi lido:

```
Ana 1001
Bruno 1002
Carlos 1003
```

Basicamente, o estudante possuía dois atributos: nome e id, sendo o código na linguagem C capaz de ler o id e o nome, escrevê-los a partir de um menu e verificar se o id se repete no arquivo txt e, caso se repita, impossibilita esse novo dado de ser escrito no arquivo. Pelo tamanho do código em C, pela simplicidade da aplicação, que é apenas leitura, escrita de dois dados, e a restrição de unicidade de id, evidencia-se como era complicado trabalhar com este sistema: Sistema de Processamento de Arquivos.

2.1.3 Motivos pelos quais deve-se usar o SGBD

De acordo com (SILBERSCHATZ; KORTH; SUDARSHAN, 2006), os Sistemas de Gerenciamento de Banco de Dados (SGBD ou DBMS) apresentam vantagens significativas em relação aos sistemas de processamento de arquivos convencionais. A seguir, descrevem-se os principais motivos para a utilização de SGBDs.

Redundância e Inconsistência de Dados

Como diferentes programadores criam arquivos e programas de aplicação ao longo do tempo, os arquivos provavelmente terão estruturas diferentes e os programas podem ser escritos em diversas linguagens de programação. Além disso, as mesmas informações podem ser duplicadas em vários locais. Por exemplo, o endereço e o número de telefone de um cliente podem aparecer em um arquivo com registros de conta poupança e em outro com registros de conta corrente. Essa redundância gera a necessidade maior de alocação de memória para armazenamento e pode causar inconsistência de dados ao possuir várias cópias dos mesmos dados e estes não concordando entre si (SILBERSCHATZ; KORTH; SUDARSHAN, 2006).

Dificuldade de Acesso a Dados

Suponha que um gerente precise encontrar os nomes de todos os clientes que moram em uma determinada área de mesmo CEP. Caso o sistema original não preveja esse tipo de solicitação, não haverá um programa de aplicação pronto para gerar essa lista. O gerente terá duas opções: obter a lista completa e extrair manualmente as informações necessárias ou solicitar a criação de um programa específico a um programador (SILBERSCHATZ; KORTH; SUDARSHAN, 2006).

Isolamento de Dados

Como os dados estão dispersos em vários arquivos e podem estar em diferentes formatos, escrever novos programas de aplicação para recuperar os dados apropriados torna-se uma tarefa difícil (SILBERSCHATZ; KORTH; SUDARSHAN, 2006).

Problemas de Integridade

Os valores de dados armazenados no banco de dados devem satisfazer determinadas restrições de consistência. Por exemplo, o saldo de certos tipos de contas bancárias nunca pode ser inferior a uma quantia predeterminada. Nos sistemas de arquivos convencionais, essas restrições são impostas pelos desenvolvedores por meio de códigos em vários programas de aplicação. Quando novas restrições são acrescentadas, é difícil modificar os programas para implementá-las, especialmente se envolverem dados de diferentes arquivos (SILBERSCHATZ; KORTH; SUDARSHAN, 2006).

Problemas de Atomicidade

Sistemas de computador estão sujeitos a falhas. Em muitas aplicações, é essencial que, se ocorrer uma falha, os dados sejam restaurados ao estado consistente anterior. Por exemplo, em uma transferência de 50 dólares da conta A para a conta B, uma falha durante a execução pode remover o valor da conta A sem creditar a conta

B, gerando inconsistência. A transferência precisa ser atômica, i.e., ou ocorre a retirada de 50 dólares da conta A e a entrada de 50 dólares na conta B, ou não ocorre nenhuma das duas. Garantir atomicidade em sistemas de arquivos convencionais é difícil (SILBERSCHATZ; KORTH; SUDARSHAN, 2006).

Anomalias de Acesso Concorrente

Sistemas modernos permitem que múltiplos usuários atualizem dados simultaneamente. Em bancos de dados com grande volume de acessos, a interação das atualizações concorrentes pode gerar inconsistência. Por exemplo, se dois clientes retirarem fundos quase que simultaneamente de uma conta com saldo de 500 dólares, i.e., um retira 50 dólares e o outro 100 dólares, ambos os programas podem ler o saldo inicial e escrever o saldo inicial menos o saldo retirado no banco de dados. Isso pode deixar o saldo final inconsistente. Para resolver isso, é necessária uma supervisão, mas isso é complicado de ser fornecido caso os dados possam ser acessados por diferentes programas (SILBERSCHATZ; KORTH; SUDARSHAN, 2006).

Problemas de Segurança

Nem todos os usuários devem acessar todos os dados. Por exemplo, funcionários do departamento de folha de pagamento devem visualizar apenas informações sobre os funcionários, não sobre contas de clientes. Em sistemas de arquivos convencionais, aplicar essas restrições de forma consistente é difícil, especialmente quando novos programas de aplicação são adicionados de maneira provisória (SILBERSCHATZ; KORTH; SUDARSHAN, 2006).

Tabela 1: Comparação entre CRUD em arquivos TXT com C e PostgreSQL

Característica	**TXT em C**	**PostgreSQL**
Integridade dos dados	Verificação de duplicatas e validação devem ser implementadas manualmente (ex.: função `id_existe`)	Garantida pelo banco (chaves primárias, constraints, tipos de dados)
CRUD	Operações complexas: abrir/ler/escrever arquivos e tratar parsing manual	Operações simples via SQL (`INSERT`, `SELECT`, `UPDATE`, `DELETE`)
Consultas complexas	Loop manual, parsing e filtros implementados pelo programador	Filtros, ordenações, junções e agregações nativos via SQL
Concorrência	Difícil de gerenciar; múltiplos processos podem corromper o arquivo	Totalmente suportado; transações e locks garantem segurança
Escalabilidade	Funciona bem apenas para pequenas quantidades de registros	Escala para milhões/bilhões de registros com alta performance
Backup/restore	Feito manualmente copiando arquivos	Ferramentas nativas (`pg_dump`, `pg_restore`)
Validação de dados	Feita manualmente no código	Constraints (tipo de dados, UNIQUE, NOT NULL, FOREIGN KEY)
Velocidade de acesso	Lento para arquivos grandes (precisa ler linha a linha)	Índices e otimizações internas garantem alta velocidade

Fonte: Próprio autor.

2.2 Níveis de Abstração do Banco de Dados

Para que o sistema seja funcional, ele precisa recuperar dados de maneira eficiente. A necessidade de eficiência tem levado projetistas a usar estruturas de dados complexas para representar dados no banco de dados. Como muitos usuários de sistema de banco de dados não são treinados em computador, os desenvolvedores, no caso o autor desse texto, ocultam a complexidade dos usuários sob vários níveis de abstração, para simplificar as interações do usuário com o sistema (SILBERSCHATZ; KORTH; SUDARSHAN, 2006):

1. Nível físico - o nível de abstração mais baixo possível e o qual não se trabalhou neste projeto.

2. Nível lógico - o nível de abstração do programador, o qual foi trabalhado pelo autor deste trabalho.

3. Nível de view - o nível de abstração do usuário leigo, i.e., o médico neste trabalho.

O programador desenvolveu uma aplicação, i.e., a criação de uma API que é capaz de, ao ser acessado o medcoqueiral.com.br e feito o login, o médico é capaz de apertar o botão "Consultas". Após feito isso, aparecerá todas as consultas com

pacientes que ele fará no futuro ordenadas da mais próxima para a mais distante. O programador, que é o autor deste texto, se encontra no nível lógico.

Os usuários, que podem ser médico, paciente ou administrador, encontram-se no nível de view.

2.3 Projeto de banco de dados

Para o desenvolvimento do chatbot, é fundamental estruturar o banco de dados de forma eficiente. Para isso, existem etapas que devem ser seguidas para sua implementação da maneira correta.

> "O projeto de um banco de dados usualmente ocorre em três etapas. A primeira etapa, a modelagem conceitual, procura capturar formalmente os requisitos de informação de um banco de dados. A segunda etapa, o projeto lógico objetiva definir, a nível de SGBD, as estruturas de dados que implementarão os requisitos identificados na modelagem conceitual. A terceira etapa, o projeto físico, define parâmetros físicos de acesso ao Banco de Dados, procurando otimizar a performance do sistema como um todo."(HEUSER, 2009)

2.3.1 Modelo Conceitual

Inicia-se a construção do banco de dados a partir do modelo conceitual, uma representação de alto nível da estrutura de dados e das regras de negócio.

> "Um modelo conceitual é uma descrição do banco de dados de forma independente de implementação em um SGBD. O modelo conceitual registra que dados podem aparecer no banco de dados, mas não registra como estes dados estão armazenados a nível de SGBD. A técnica mais difundida de modelagem conceitual é a abordagem entidade-relacionamento (ER). Nesta técnica, um modelo conceitual é usualmente representado através de um diagrama, chamado diagrama entidade-relacionamento (DER)."(HEUSER, 2009)

Dessa maneira, seguindo tanto o livro do Heuser (2009) quanto o do Silberschatz, Korth e Sudarshan (2010), faz-se um exemplo para se compreender melhor a abordagem.

O Diagrama Entidade-Relacionamento (DER) representa graficamente as entidades, seus atributos e os relacionamentos existentes entre elas.

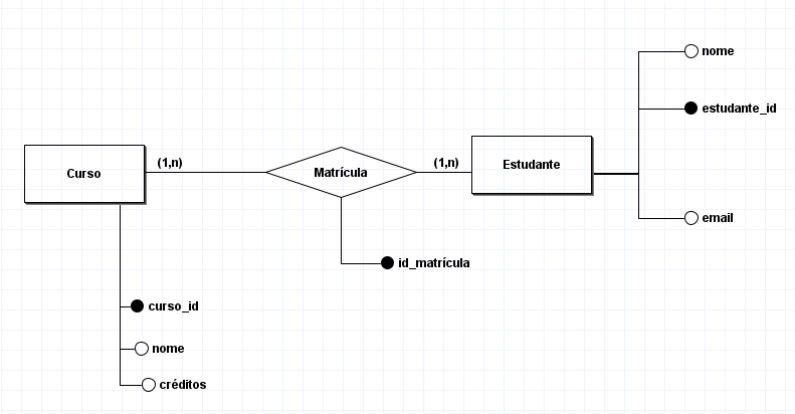

Figura 1: Exemplo de Modelo Conceitual com a técnica Entidade-Relacionamento.

O diagrama (Figura 1) demonstra um sistema acadêmico simples, cujos componentes principais são analisados a seguir.

Entidades: São os objetos centrais do modelo, representados por retângulos.

- **Estudante:** Representa os alunos da instituição.
- **Curso:** Representa as disciplinas ou cursos oferecidos.

Atributos: São as propriedades das entidades ou relacionamentos. No diagrama, os círculos preenchidos indicam atributos que são chaves primárias.

- Atributos de Estudante: `estudante_id` (PK), `nome`, `email`.
- Atributos de Curso: `curso_id` (PK), `nome`, `créditos`.
- Atributo de Matrícula: `id_matrícula` (PK).

Relacionamento: Descreve a interação entre as entidades, representado pelo losango.

- **Matrícula:** É o evento que conecta um `Estudante` a um `Curso`.

Relações de Cardinalidade (M:N): A cardinalidade, na notação (min, max), define a proporção numérica do relacionamento. A notação `(1,n)` em ambos os lados indica uma relação **Muitos-para-Muitos (N:M)** com participação total, significando:

- **Cardinalidade Máxima (n):** Um estudante pode se matricular em 'n' (muitos) cursos, e um curso pode ter 'n' (muitos) estudantes.
- **Cardinalidade Mínima (1):** A participação no relacionamento é obrigatória. Um curso deve ter no mínimo 1 estudante matriculado, e um estudante deve estar matriculado em no mínimo 1 curso.

2.3.2 Necessidades Funcionais

Além da estrutura dos dados, o modelo conceitual também deve abranger como os usuários interagem com o banco de dados.

> "Um esquema conceitual totalmente desenvolvido também indicará as necessidades funcionais da empresa. Em uma especificação das necessidades funcionais, os usuários descrevem os tipos de operações (ou transações) que serão realizadas nos dados. Exemplos de operações incluem modificar ou atualizar dados, pesquisar e recuperar dados específicos e excluir dados. Nessa fase do projeto conceitual, o projetista pode revisar o esquema para se certificar de que ele atende às necessidades funcionais."(SILBERSCHATZ; KORTH; SUDARSHAN, 2006)

2.3.3 Modelo Lógico

O Modelo Lógico, também chamado de Projeto Lógico, é o esquema do banco de dados que resulta do mapeamento do modelo conceitual. Esta é uma representação em termos de tabelas e relacionamentos que podem ser implementados em um sistema de gerenciamento de banco de dados relacional, como o PostgreSQL. A partir do modelo ER, são definidos:

- **Tabelas:** correspondentes às entidades, com seus atributos transformados em colunas.
- **Chaves primárias e estrangeiras:** que garantem a integridade referencial entre as tabelas.
- **Relacionamentos:** implementados por meio de chaves estrangeiras ou tabelas associativas.

2.3.4 Mapeamento para Tabelas

Ao traduzir o modelo conceitual para um modelo lógico, o relacionamento M:N **Matrícula** se torna uma **tabela associativa**. As tabelas resultantes são:

Tabela `Estudante`:

- `estudante_id` (Chave Primária - PK)
- `nome`
- `email`

Tabela `Curso`:

- `curso_id` (Chave Primária - PK)
- `nome`
- `créditos`

Tabela `Matricula`:

- `id_matrícula` (Chave Primária - PK)
- `estudante_id` (Chave Estrangeira - FK, referenciando `Estudante`)
- `curso_id` (Chave Estrangeira - FK, referenciando `Curso`)

Observações sobre Chaves na Tabela `Matricula`: A definição das chaves para a tabela associativa `Matricula` é uma decisão de projeto importante. Temos duas opções para identificar unicamente cada registro:

- **Chave Natural:** A combinação das chaves estrangeiras, `{estudante_id, curso_id}`.
- **Chave Substituta:** Um novo identificador sem significado de negócio, como `id_matrícula`, frequentemente utilizadas como substituição de uma chave primária composta.

A prática moderna favorece o uso de chaves substitutas como chaves primárias por simplificarem os relacionamentos. Para entender a escolha, aplicamos a teoria de chaves conforme segue (SILBERSCHATZ; KORTH; SUDARSHAN, 2010):

- **Superchave:** Qualquer conjunto de colunas que identifica unicamente uma linha. Para a `Matricula`, tanto `{id_matricula}` quanto `{estudante_id, curso_id}` são superchaves.
- **Chave Candidata:** Uma superchave mínima. Nesse caso, temos duas chaves candidatas: `{id_matricula}` e `{estudante_id, curso_id}`.
- **Chave Primária:** A chave candidata que o projetista **escolhe** para ser o identificador principal.

Neste projeto de exemplo, **escolhe-se `{id_matricula}` como a Chave Primária**. A outra chave candidata, `{estudante_id, curso_id}`, é implementada como uma **Chave Alternativa** (restrição 'UNIQUE') para garantir a regra de negócio.

2.3.5 Representação Textual do Esquema Lógico

O esquema lógico pode ser representado de forma textual e concisa. As tabelas são definidas pelo nome, seguido por seus atributos entre parênteses. A chave primária é sublinhada e as chaves estrangeiras são indicadas.

Estudante = (estudante_id, nome, email)

Curso = (curso_id, nome, créditos)

Matrícula = (id_matrícula, *estudante_id (FK)*, *curso_id (FK)*)

2.4 DBML: A Ponte Entre o Design e o Código

O **DBML (Database Markup Language)** é uma linguagem de marcação de código aberto, declarativa e de fácil leitura, projetada especificamente para definir e documentar esquemas de bancos de dados. Criada pela equipe por trás do `dbdiagram.io`, sua principal finalidade é permitir que desenvolvedores e analistas de dados projetem e mantenham a estrutura de um banco de dados usando uma sintaxe simples e versionável, tratando o esquema como *schema as code*, i.e., o "esquema é muito semelhante ao código".

2.4.1 Finalidade e Motivação

Tradicionalmente, os esquemas de banco de dados são projetados com ferramentas gráficas (GUI). Embora úteis para visualização, esses diagramas podem se tornar desatualizados, difíceis de compartilhar e quase impossíveis de versionar de forma eficaz em sistemas como o Git. O DBML resolve esses problemas ao fornecer uma fonte única da verdade em formato de texto, facilitando a colaboração e a manutenção.

2.4.2 Exemplo de Aplicação

A tradução do esquema lógico para a sintaxe DBML resulta em um código limpo e legível, que pode ser visualizado em ferramentas como o `dbdiagram.io`, conforme ilustrado na Figura 2.

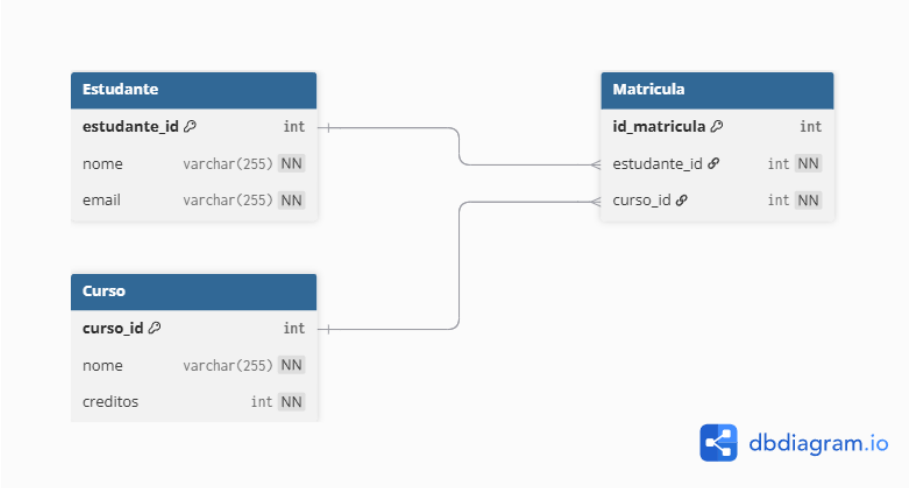

Figura 2: Tradução do esquema lógico para a sintaxe DBML no `dbdiagram.io`.

2.4.3 Principais Vantagens

A adoção do DBML traz benefícios significativos para o ciclo de vida de um banco de dados:

- **Sintaxe Legível:** A sintaxe é limpa e intuitiva, facilitando a leitura e a escrita tanto por humanos quanto por máquinas.
- **Independente de SGBD:** O DBML é independente em relação ao banco de dados. Um mesmo arquivo `.dbml` pode ser usado para gerar o código

SQL DDL (`CREATE TABLE`, etc.) para diferentes sistemas, como PostgreSQL, MySQL e SQL Server.

- **Facilita o Versionamento:** Por ser um arquivo de texto simples, as alterações no esquema do banco de dados podem ser facilmente rastreadas, revisadas e mescladas usando sistemas de controle de versão como o `Git`.
- **Ecossistema de Ferramentas:** A linguagem é suportada por um conjunto de ferramentas poderosas, incluindo o visualizador online `dbdiagram.io` e a ferramenta de linha de comando `dbml-cli`, que permite a conversão entre DBML e SQL.
- **Documentação Integrada:** Permite adicionar notas e comentários diretamente nas tabelas e colunas, mantendo a documentação sempre sincronizada com a estrutura.

O DBML, portanto, atua como uma ponte essencial entre o modelo lógico e o modelo físico, fornecendo um formato padronizado que agiliza a colaboração, a documentação e a implementação de bancos de dados de forma moderna e eficiente.

2.5 Geradores de Relatório

Para que o médico ou paciente conseguisse ver as consultas no website `medcoqueiral.com.br` a partir de requisições `HTTP` ao endpoint que se comunicava com o servidor em Python (localhost), foi necessário seguir o seguinte referencial teórico.

No capítulo 8 do livro (SILBERSCHATZ; KORTH; SUDARSHAN, 2010), fala-se sobre o desenvolvimento de aplicações web as quais o usuário interage com banco de dados sem uma linguagem de consulta. A maioria interage com um sistema de banco de dados por uma destas maneiras:

1. Formulários e interfaces gráficas
2. Geradores de relatório
3. Ferramentas de análise de dados

O caso deste trabalho foi o de geradores de relatório que é:

> "Geradores de relatório permitem que relatórios predefinidos sejam gerados no conteúdo atual do banco de dados." (SILBERSCHATZ; KORTH; SUDARSHAN, 2006)

É exatamente isso que ocorre nessa aplicação, a qual são gerados relatórios de consultas entre médico e paciente a partir do banco de dados atual.

Nesta aplicação, o médico pode gerar um relatório com as suas consultas agendadas, realizadas e que serão realizadas.

2.6 Integração do Flask e do HTTP

Para que o front-end do website `medcoqueiral.com.br` pudesse exibir as consultas médicas, foi necessário estabelecer uma comunicação entre o navegador do usuário e o servidor que hospeda a aplicação em Python. Essa comunicação ocorre por meio do protocolo `HTTP` (Hypertext Transfer Protocol), que define como mensagens são formatadas e transmitidas entre cliente e servidor.

O Flask, um microframework web em Python, foi utilizado para criar os endpoints `HTTP` que recebem requisições do front-end e retornam os dados do banco de dados. Cada requisição `HTTP` realizada pelo navegador é interpretada pelo Flask, que executa a lógica correspondente — no caso, consultas ao banco de dados para gerar relatórios médicos — e retorna uma resposta contendo os dados solicitados em formato apropriado, como `JSON`.

Dessa forma, o médico ou paciente consegue visualizar as consultas sem precisar interagir diretamente com o banco de dados, garantindo uma experiência mais simples e segura. O Flask abstrai a complexidade do acesso ao banco de dados, permitindo que o front-end apenas faça requisições `HTTP` e receba os relatórios desejados.

2.7 Métodos HTTP: GET e POST na Prática com Flask

A comunicação entre o cliente (navegador) e o servidor, regida pelo protocolo `HTTP`, não se limita a apenas solicitar e enviar dados de forma genérica. O `HTTP` define *"métodos"* que especificam a intenção da requisição. No desenvolvimento de APIs com Flask, como a presente aplicação, os métodos mais fundamentais são `GET` e `POST`. A escolha correta entre eles é crucial para a segurança, clareza e funcionalidade do sistema.

2.7.1 O Método GET: Buscando Dados

O método **`GET`** é utilizado para **solicitar e recuperar dados** de um recurso específico no servidor. Sua principal característica é a **idempotência**, o que significa que realizar a mesma requisição `GET` múltiplas vezes deve produzir o mesmo resultado, sem causar efeitos colaterais ou alterações no estado do servidor. Os dados enviados em uma requisição `GET` são anexados diretamente à URL, na forma de uma *query string* (ex: `/endpoint?chave=valor`).

No código da aplicação, o uso do `GET` é evidente em rotas cuja finalidade é a consulta de informações. Por exemplo, o endpoint de login via token:

Listagem 1: Exemplo de endpoint GET no Flask.

```
@app.route("/login_api", methods=["GET"])
def login_api():
    telefone = request.args.get("telefone", "")
    token = request.args.get("token", "")
```

Neste caso, o cliente envia o telefone e o token como parâmetros na URL. O servidor, através do objeto `request.args` do Flask, lê esses parâmetros para buscar e validar a sessão do usuário. Outros exemplos claros são os endpoints `/listar_medicos_dashboard`, `/listar_consultas_paciente` e `/listar_horarios`,

todos definidos com `methods=["GET"]` e cuja única função é retornar listas de dados existentes no banco de dados.

2.7.2 O Método POST: Enviando Dados para Modificação

Em contrapartida, o método `POST` é utilizado para **enviar dados ao servidor para criar um novo recurso ou executar uma ação que altera seu estado**. Diferentemente do `GET`, os dados em uma requisição `POST` são enviados no corpo (*body*) da requisição, geralmente em formato `JSON` ou como dados de formulário. Isso torna o método mais seguro para o envio de informações sensíveis (como senhas ou dados pessoais) e permite o envio de volumes de dados muito maiores. Requisições `POST` não são idempotentes; por exemplo, enviar os mesmos dados de cadastro duas vezes resultaria na tentativa de criar dois usuários distintos.

A aplicação utiliza o método `POST` para todas as operações que modificam o banco de dados. O endpoint de cadastro de usuários é um exemplo canônico:

Listagem 2: Exemplo de endpoint POST no Flask.

```
@app.route("/cadastro", methods=["POST"])
def cadastrar_usuario():
    data = request.get_json()
```

Aqui, o Flask utiliza `request.get_json()` para extrair os dados do corpo da requisição e, com eles, criar um novo registro de `Pessoa` e `Paciente` ou `Medico`. Outras rotas como `/create-checkout-session` (cria uma sessão de pagamento), `/adicionar_horario` (insere um novo horário), `/cancelar_consulta` (altera o status de uma consulta) e o crucial `/stripe-webhook` (que recebe uma notificação externa para criar uma consulta) também utilizam `POST` para indicar que estão realizando uma ação de escrita ou modificação no servidor.

Tabela Comparativa

A tabela a seguir resume as principais diferenças entre `GET` e `POST` com base no seu uso na aplicação.

Tabela 2: Tabela Comparativa: GET vs POST

Característica	Método GET	Método POST
Finalidade	Recuperar dados do servidor.	Enviar dados para criar ou modificar recursos no servidor.
Envio de Dados	Pela *query string* da URL (`request.args`).	No corpo (body) da requisição (`request.get_json()`, `request.data`).
Visibilidade	Dados visíveis na URL, no histórico do navegador e em logs.	Dados não são visíveis na URL, oferecendo maior privacidade.
Idempotência	Sim. Múltiplas chamadas não alteram o estado do servidor.	Não. Múltiplas chamadas podem criar múltiplos recursos ou gerar erros.
Exemplos no Código	`/login_api`, `/listar_consultas`, `/listar_horarios`.	`/cadastro`, `/adicionar_horario`, `/create-checkout-session`.

2.8 Webhooks

No projeto, a integração com a API do WhatsApp Cloud foi realizada por meio de um *webhook*, que permite a comunicação assíncrona baseada em eventos, também chamada de *event-driven API architecture* (EBBELAAR, 2023a). Diferentemente de sistemas síncronos, nos quais o servidor precisa consultar continuamente o serviço externo para verificar se há novas informações, um webhook possibilita que o servidor local seja notificado automaticamente sempre que um evento relevante ocorre, como o recebimento de uma mensagem de um usuário. Uma analogia comum é o "Hollywood casting": assim como um ator não precisa ficar ligando para o diretor perguntando se foi contratado, é o diretor quem entra em contato informando o resultado; de forma similar, o servidor do WhatsApp envia os dados ao webhook assim que a mensagem chega, sem necessidade de requisições constantes.

Essa abordagem é assíncrona porque o servidor local não bloqueia sua execução aguardando novas mensagens; ele apenas reage aos eventos enviados pelo serviço externo. Além disso, é praticamente em tempo real, permitindo que ações como o agendamento de consultas ou envio de respostas automáticas sejam executadas rapidamente, garantindo maior eficiência na comunicação entre sistemas.

2.9 WhatsApp Cloud API

A compreensão do funcionamento da WhatsApp Cloud API fundamenta-se na documentação técnica provida pela Meta for Developers. Esta documentação detalha os requisitos, a arquitetura e os procedimentos necessários para a integração de sistemas com a plataforma de mensagens (META FOR DEVELOPERS, 2025b).

2.9.1 Registro de Telefone na WhatsApp Business API

O processo de registro de um número de telefone na WhatsApp Business API exige a verificação de um CNPJ, restringindo o seu uso a entidades empresariais. Para a implementação deste projeto, foi necessário vincular o domínio web `medcoqueiral.com.br` ao CNPJ da empresa e desenvolver uma página institucional. Essa página institucional detalha o ramo de atuação da organização e apresenta uma política de privacidade explícita, que informa como os dados dos usuários são tratados e armazenados, cumprindo os pré-requisitos da plataforma (META FOR DEVELOPERS, 2025b).

2.9.2 Consentimento do Usuário para Recebimento de Mensagens

Para que uma empresa possa iniciar o contato com um usuário através do chatbot, é imprescindível a obtenção prévia de seu consentimento explícito (opt-in). A Meta estabelece diversas formas para que essa autorização seja concedida, entre elas (META FOR DEVELOPERS, 2025a):

1. SMS
2. Website
3. Fluxo de Resposta de Voz Interativa (URA)
4. Autorização presencial
5. Assinatura em documento físico

No contexto deste trabalho, os números de telefone foram obtidos através da própria interação dos usuários com o chatbot. A iniciativa do contato por parte do usuário demonstra o interesse em estabelecer comunicação com a empresa, configurando, assim, um consentimento implícito para o tratamento de dados relacionados ao agendamento de consultas médicas.

2.9.3 Conversas Iniciadas pelo Usuário

De acordo com a documentação oficial da Meta para desenvolvedores, as conversas na plataforma WhatsApp iniciadas pelo usuário não geram custos para a empresa. Ou seja, quando um usuário envia a primeira mensagem para o chatbot, essa interação inicial é isenta de tarifas. A isenção se estende à resposta do chatbot e a todas as mensagens trocadas dentro de uma janela de 24 horas, denominada "Janela de Atendimento ao Cliente" (Customer Service Window). Durante esse período, a empresa pode se comunicar livremente com o usuário sem incorrer em custos adicionais (META FOR DEVELOPERS, 2025c).

2.9.4 Estrutura de Preços para Mensagens no WhatsApp

A política de preços para o uso da API do WhatsApp é complexa e sujeita a alterações frequentes. Portanto, as informações aqui apresentadas podem estar desatualizadas no futuro (META FOR DEVELOPERS, 2025c). O WhatsApp não cobra por mensagens enviadas pela empresa que se enquadrem nas seguintes condições:

1. Sejam respostas a uma conversa iniciada pelo usuário (dentro da janela de 24 horas) e não utilizem um modelo de mensagem (template).

Os "templates" são mensagens pré-aprovadas pelo WhatsApp, utilizadas para iniciar conversas com os usuários.

			Is it billable?		
Message type	**Message category**	**When can it be sent?**	**Delivered outside CSW***	**Delivered inside CSW***	**Delivered inside FEP** window**
Template messages	Marketing	Anytime	Yes	Yes	**No**
	Authentication	Anytime	Yes	Yes	**No**
	Utility	Anytime	Yes	**No**	**No**
Non-template (free-form) messages	Service	Only inside an open CSW*	N/A	**No**	**No**

* Customer service window

** Free entry point window

Figura 3: Tabela de preços retirada do Meta Developers (META FOR DEVELOPERS, 2025c)

A "Janela de Atendimento ao Cliente" (CSW - Customer Service Window) é o período de 24 horas, iniciado a partir da última mensagem do usuário, dentro do qual o chatbot pode responder sem custos. O chatbot em questão operará exclusivamente dentro da CSW, uma vez que as respostas são enviadas imediatamente após a interação do usuário.

2.9.5 JSON Payload da WhatsApp Cloud API

As mensagens enviadas ao chatbot são transformadas em *JSON payloads* ao transitarem pela WhatsApp Cloud API e são entregues a um *webhook*, o qual é processado pelo servidor local. Esses *JSON payloads* seguem um formato padronizado que contém informações sobre a mensagem, o remetente e metadados da conta de WhatsApp Business.

Segue um exemplo de de *JSON payload* que o WhatsApp Cloud API envia para o *webhook* quando alguém envia uma mensagem ao chatbot.

Listagem 3: Exemplo de JSON payload da WhatsApp Cloud API.

```
{
  "object": "whatsapp_business_account",
  "entry": [
    {
      "id": "WHATSAPP_BUSINESS_ACCOUNT_ID",
      "changes": [
        {
          "value": {
            "messaging_product": "whatsapp",
            "metadata": {
              "display_phone_number": "5511999999999",
              "phone_number_id": "PHONE_NUMBER_ID"
            },
            "messages": [
              {
                "from": "5511888888888",
                "id": "ABCD1234",
                "timestamp": "1693905600",
                "type": "text",
                "text": {
                  "body": "Ol  , gostaria de agendar uma consulta"
                }
              }
            ]
          },
          "field": "messages"
        }
      ]
    }
  ]
}
```

No *backend* do chatbot, implementado em Python com Flask, o *webhook* recebe o *JSON payload* e extrai informações relevantes, como o número do remetente, conteúdo da mensagem e o tipo da interação. Existem interações recebidas pelo número de WhatsApp do usuário que são filtradas de tal maneira que não são respondidas pelo chatbot, como uma atualização da foto do WhatsApp ou uma atualização do nome. Em suma, apenas interações do tipo mensagem são respondidas, i.e., as mensagens enviadas do usuário ao chatbot.

O fluxo de processamento do *JSON payload* pode ser resumido da seguinte forma:

1. O usuário envia uma mensagem para o número do WhatsApp Business.
2. A WhatsApp Cloud API transforma a mensagem em um *JSON payload* e envia para o *webhook* público do servidor.
3. O servidor back-end recebe o *JSON payload* e processa os dados, identificando remetente, tipo e conteúdo da mensagem.
4. Os dados extraídos entram em fluxos conversacionais pré-definidos em estruturas (em geral condicionais de if-else), permitindo marcar consulta com médico, saber o dia e horário da consulta e outros métodos disponíveis para aquele usuário.

Dessa forma, o uso de *JSON payloads* padronizados permite que o sistema do chatbot mantenha comunicação estruturada e assíncrona com a WhatsApp Cloud API.

2.10 Programação Orientada a Objetos

Para a implementação das classes de usuários que interagem com o banco de dados, adotou-se o paradigma de programação orientada a objetos (POO). Este paradigma permite modelar sistemas complexos de forma mais intuitiva, aproximando a estrutura do código das entidades do mundo real. Na POO, o foco está na criação de *objetos*, que encapsulam tanto dados (*atributos*) quanto comportamentos (*métodos*), promovendo maior modularidade, reutilização e manutenção do código.

O paradigma de programação orientada a objetos foi fundamental para este projeto, principalmente pelos seguintes motivos:

1. **Modularização do código:** Cada classe representa uma entidade específica do sistema, tornando o desenvolvimento mais organizado e facilitando futuras alterações ou expansões.

2. **Redução de redundâncias:** A herança e a criação de métodos comuns em classes abstratas permitem evitar a repetição de código, garantindo consistência e eficiência.

3. **Organização de métodos e atributos:** A definição clara dos atributos e métodos de cada classe possibilita melhor gerenciamento das responsabilidades de cada tipo de usuário.

Princípios de Programação Orientada a Objetos

O desenvolvimento do sistema utilizou os principais conceitos de POO:

Abstração Permite representar entidades reais de forma simplificada, destacando apenas os atributos e comportamentos relevantes para o sistema. No projeto, a classe *Usuário* é uma classe abstrata que define propriedades e métodos comuns a todos os usuários, mas não será instanciada diretamente.

Encapsulamento Refere-se à proteção dos dados e à definição de métodos de acesso controlados (*getters* e *setters*), evitando alterações indevidas nos atributos internos dos objetos. Por exemplo, informações sensíveis como senha e CPF são encapsuladas, permitindo acesso apenas por métodos específicos.

Herança Permite que classes derivadas (*Paciente* e *Médico*) reutilizem atributos e métodos da classe base (*Usuário*), evitando duplicação de código e facilitando manutenção. O *Administrador*, por sua vez, é implementado como uma classe isolada, pois suas funcionalidades não se enquadram no perfil de *Usuário*.

Polimorfismo Permite que objetos de diferentes classes sejam tratados de maneira uniforme, desde que compartilhem a mesma interface ou classe base. No projeto, métodos definidos na classe abstrata *Usuário* podem ser implementados de formas diferentes em *Paciente* e *Médico*, garantindo flexibilidade na execução de operações específicas.

Estrutura das Classes

O sistema foi estruturado com quatro classes principais:

1. **Pessoa:** Classe abstrata que define atributos, como nome, CPF, celular e senha. Nenhum objeto será criado diretamente desta classe.

2. **Paciente:** Herda de *Pessoa* e inclui métodos específicos para agendamento de consultas e visualização de histórico médico. Como atributos específicos, possui um histórico de comorbidades os quais recebem os valores booleanos de verdadeiro ou falso.

3. **Médico:** Herda de *Pessoa* e possui métodos para gerenciamento de agenda, registro de consultas e acesso a informações dos pacientes. Como atributos específicos, possui CRM.

4. **Administrador:** Classe isolada, responsável por funcionalidades de gerenciamento do sistema, como exclusão de usuários, alteração de dados de usuários e geração da chave de criação do usuário do tipo Médico.

Essa estrutura baseada em POO garante um sistema modular, organizado, seguro e escalável, atendendo aos requisitos de manutenção e extensibilidade do projeto.

2.11 Tecnologias da Camada de Apresentação (Front-end)

A construção da interface do usuário (front-end) do sistema, que compreende tanto o site institucional quanto os painéis dinâmicos de médicos e pacientes, fundamentou-se em um conjunto de tecnologias web consolidadas. O conhecimento aprofundado de `HTML`, `CSS`, `JavaScript` e `PHP` foi crucial para traduzir os requisitos do projeto em uma experiência de usuário funcional, segura e interativa.

2.11.1 HTML (HyperText Markup Language)

O `HTML` foi empregado como a linguagem de marcação fundamental para a estruturação de todo o conteúdo das páginas web. Sua importância reside na capacidade de definir a semântica e a hierarquia dos elementos, como cabeçalhos, parágrafos, listas, tabelas e formulários. No desenvolvimento do site `medcoqueiral.com.br`, a utilização de tags semânticas como `<header>`, `<nav>` e `<section>` não apenas organizou o conteúdo de forma lógica, mas também otimizou a acessibilidade e a indexação por mecanismos de busca. O `HTML` serviu como o esqueleto sobre o qual os estilos visuais e as funcionalidades dinâmicas foram aplicados.

2.11.2 CSS (Cascading Style Sheets)

Enquanto o HTML estrutura o conteúdo, o CSS é responsável por toda a sua apresentação visual. O conhecimento em CSS foi essencial para criar a identidade visual do site, definindo cores, fontes, espaçamentos e layouts. No projeto, foi utilizado tanto um arquivo de estilos customizado (`styles.css`) para o site público quanto o framework *utility-first* **Tailwind CSS** nos painéis de usuário. O conhecimento de boas práticas de Engenharia de Software foram aplicadas de tal maneira que toda a parte de CSS foi mantida em um único aquivo (`styles.css`) e importada em cada página na web. A aplicação de CSS permitiu a criação de um design responsivo, garantindo que a interface se adaptasse de forma fluida a diferentes tamanhos de tela, como desktops e dispositivos móveis, um requisito indispensável para a usabilidade do sistema.

2.11.3 JavaScript

O JavaScript foi a tecnologia central para adicionar interatividade e dinamismo às páginas, atuando como o *"cérebro"* do lado do cliente (navegador). Seu domínio foi vital para transformar páginas estáticas em aplicações ricas e reativas. No projeto, o JavaScript desempenhou múltiplos papéis:

- **Manipulação do DOM (Document Object Model):** Foi utilizado para criar e modificar elementos HTML dinamicamente, como na página de especialidades, onde a lista é gerada e inserida na tabela em tempo de execução. Nos painéis, toda a exibição de consultas e horários é renderizada dinamicamente com base nos dados recebidos da API.
- **Comunicação Assíncrona (AJAX):** Através da API Fetch, o JavaScript realiza requisições HTTP assíncronas ao servidor. Essa capacidade foi crucial para os painéis de médico e paciente, permitindo buscar, enviar e atualizar dados (como a lista de consultas) sem a necessidade de recarregar a página inteira, proporcionando uma experiência de usuário mais rápida e fluida.
- **Gerenciamento de Eventos:** Captura interações do usuário, como cliques em botões para agendar uma consulta ou adicionar um horário, disparando as funções correspondentes para interagir com o back-end.

2.11.4 PHP (Hypertext Preprocessor)

Nesta arquitetura, o PHP foi empregado como a principal linguagem do lado do servidor web (*server-side*), atuando como um orquestrador e uma camada de segurança entre o front-end e o back-end em Python. Seu conhecimento foi fundamental para duas funções críticas:

1. **Gerenciamento de Sessão:** O PHP foi responsável por iniciar (`session_start()`) e manter a sessão do usuário após o login. Ao armazenar o identificador único e seguro do usuário (`id_pessoa`) na superglobal `$_SESSION`, o sistema pôde verificar a autenticidade do usuário em todas as requisições subsequentes, protegendo o acesso aos painéis.

2. **Proxy Seguro para a API:** Foi implementado um script de *proxy* em `PHP` (`proxy.php`) que atua como um intermediário seguro. Todas as requisições `JavaScript` dos painéis são enviadas a este script. O proxy, então, valida a sessão `PHP` ativa, anexa o `id_pessoa` do usuário à requisição e a encaminha para o back-end em Python. Esta abordagem protege o back-end, garantindo que ele apenas receba requisições de usuários já autenticados pela camada do `PHP`.

2.11.5 Fluxo de Autenticação e Gerenciamento de Sessão

O processo de login foi desenhado para ser seguro e desacoplado, combinando a autenticação via chatbot com o gerenciamento de sessão no servidor web. O fluxo teórico pode ser resumido da seguinte forma:

1. **Início no Chatbot:** O usuário inicia a autenticação no WhatsApp, que, após validação, fornece um link de acesso único contendo um token de uso único e o número de telefone.

2. **Validação do Token:** O usuário clica no link e é direcionado a um script `PHP` (`login_api.php`) no servidor web. Este script atua como um proxy, repassando o token para um endpoint específico no back-end em Python.

3. **Criação da Sessão:** O back-end valida o token. Se for válido, retorna os dados do usuário, como seu ID interno e seguro (`id_pessoa`), nome e tipo de perfil. O `PHP`, ao receber essa confirmação, inicia uma sessão segura (`session_start()`) e armazena esses dados na variável `$_SESSION`.

4. **Acesso Persistente:** O usuário é redirecionado para o seu painel respectivo (`painel_paciente.php` ou `painel_medico.php`). A partir deste momento, todas as futuras requisições feitas pelo `JavaScript` para obter ou modificar dados são autenticadas pela sessão `PHP` ativa, que é verificada pelo `proxy.php` antes de qualquer comunicação com o back-end.

Este modelo híbrido aproveita a conveniência do login via chatbot e a robustez do gerenciamento de sessão tradicional baseado em servidor, garantindo que, uma vez autenticado, o acesso do usuário permaneça seguro e persistente durante sua navegação.

3 Desenvolvimento

Esta seção apresenta como o projeto de software foi desenvolvido. Para garantir que o processo fosse conduzido corretamente, foi necessário seguir uma metodologia adequada, isto é, os passos da engenharia de software.

O primeiro passo consistiu no levantamento de requisitos, que incluíam: modelagem do banco de dados e sua implementação em PostgreSQL na máquina pessoal do autor, a criação de um chatbot no WhatsApp para receber mensagens dos pacientes e capacidade de realizar o agendamento de consultas médicas, além de uma interface no site medcoqueiral.com.br que permitisse a médicos e pacientes acompanhar suas consultas agendadas. Assim como uma função que terá como ação o envio de notificações aos pacientes quando eles estiverem próximos à data da consulta.

O chatbot, que representa o back-end, utiliza a WhatsApp Cloud API e está hospedado em ambiente local (localhost). Como não é possível acessá-lo diretamente pela internet, foi necessário expô-lo por meio de uma URL pública, o que foi feito utilizando o software Ngrok. Já a interface gráfica do front-end encontra-se em um hosting compartilhado da Locaweb.

Como os dois servidores estão armazenados em locais distintos — um na máquina pessoal do autor e outro no servidor compartilhado —, tornou-se necessário estabelecer comunicação entre eles. Essa comunicação foi viabilizada por meio de uma API em Flask, permitindo a troca de informações via requisições HTTP.

3.1 Visão pelo website

Caso o usuário, seja ele médico, seja ele paciente, interagisse pelo website, ele faria, na realidade, o seguinte fluxograma para ter acesso aos dados do seu agendamento.

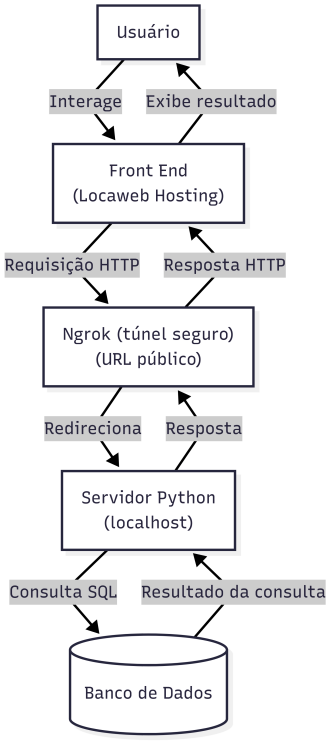

Figura 4: Visão do usuário pelo website

Na Figura 4, usa-se o termo Front End (Locaweb Hosting), o qual significa o website (medcoqueiral.com.br) e a maneira como ele está sendo mantido online, por meio de um host compartilhado da locaweb.com.br. O usuário, sendo ele médico ou paciente, faz requisições HTTP do tipo GET e estas são redirecionadas (por meio do URL público) ao servidor Python que faz consulta SQL no banco de dados e recebe os dados referente àquela requisição HTTP do usuário. Em seguida, o servidor Python envia a resposta para o URL público e ela é redirecionada para o website medcoqueiral.com.br a qual aparece ao usuário. Essas requisições do tipo GET são feitas a partir de um seletor de menu e o clique em enter em um dashboard que aparece ao usuário do tipo médico.

3.2 Visão pelo chatbot

Caso o usuário interagisse com o chatbot, i.e., enviasse mensagens ao chatbot, o fluxograma muda um pouco, pois agora ele está interagindo com a WhatsApp Cloud API que manda requisições HTTP ao URL público que redireciona essas requisições HTTP ao localhost.

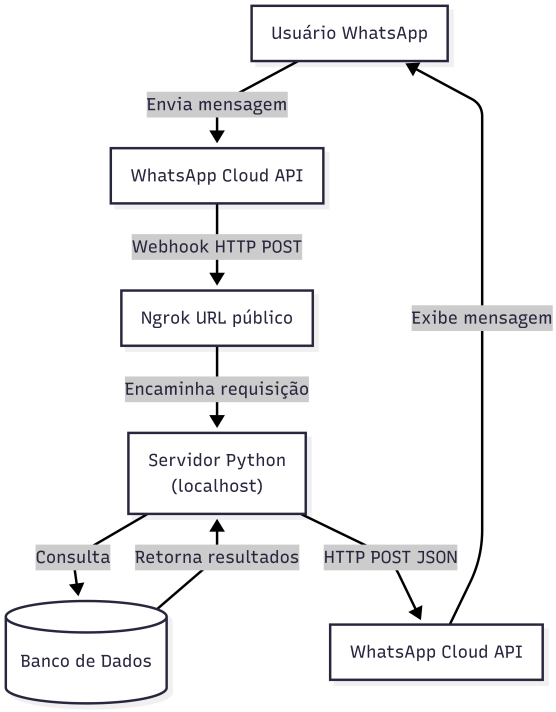

Figura 5: Visão do usuário pelo chatbot

3.3 Configuração do Ambiente na Plataforma Meta for Developers

O desenvolvimento de uma solução de chatbot integrada ao WhatsApp exige a utilização da infraestrutura fornecida pela Meta através da sua API de Nuvem (Cloud API). Esta subseção descreve o processo fundamental de configuração do ambiente de desenvolvimento, desde a criação da conta de desenvolvedor até a configuração inicial de um número para testes. Esta etapa é um pré-requisito indispensável para estabelecer a comunicação entre a aplicação e a API do WhatsApp.

3.4 Configuração Inicial da Plataforma Meta e WhatsApp

O acesso à API do WhatsApp é gerenciado pela plataforma Meta for Developers. A configuração inicial envolve a criação de uma conta, uma aplicação empresarial e a verificação de um número para testes.

3.4.1 Criação da Conta e da Aplicação

O primeiro passo é o cadastro na plataforma `developers.facebook.com`, vinculando uma conta do Facebook ao perfil de desenvolvedor. Dentro do painel de controle (*dashboard*), cria-se um novo aplicativo do tipo **"Empresarial"** (*Business*). A este aplicativo, adiciona-se o produto **"WhatsApp"**, o que direciona para a página de configuração da API.

3.4.2 Adição de um Número de Telefone para Testes

A Meta fornece um número de teste para facilitar o desenvolvimento. Para validar um destinatário, na seção **WhatsApp > API Setup**, adiciona-se um número de telefone pessoal no campo **"To"**. Apenas mantenha o número de teste que se encontra no **"From"** ou selecione esse número de testes gratuito que já se encontra no seletor. Um código de verificação é enviado via WhatsApp para este número adicionado no **"To"** e deve ser inserido na plataforma para autorizar o recebimento de mensagens. Ao ser finalizada essa etapa, já se pode enviar mensagens pré-aprovadas e fixas (templates), mas ainda não é possível a interação entre aplicação e a Cloud API. Para isso, caso a aplicação esteja em um computador pessoal, precisa-se de fazer um túnel seguro o qual a aplicação apontará para o localhost e será exposta para um domínio público o qual, por meio deste, a Cloud API mandará requisições HTTP e o domínio público enviará requisições HTTP para o localhost.

3.5 Configuração do Webhook para Recebimento de Mensagens

Para que a Cloud API do WhatsApp possa notificar a aplicação sobre novas mensagens recebidas, é necessário configurar um *endpoint* de Webhook. Durante o desenvolvimento local, a aplicação em execução na máquina do desenvolvedor não é acessível publicamente pela internet. Para resolver essa questão, utiliza-se a ferramenta **Ngrok**, que cria um túnel seguro entre um endereço público na internet e a porta local onde a aplicação está sendo executada (e.g., `localhost:8000`), servindo como uma ponte para a comunicação.

3.5.1 Estabelecimento do Túnel com Ngrok

A Meta exige um domínio estático para a validação do webhook. O Ngrok oferece um domínio estático gratuito, cujo processo de configuração é detalhado a seguir:

1. **Cadastro e Autenticação:** Baixe o Ngrok, realiza-se o cadastro na plataforma do Ngrok e autentica-se o agente localmente, i.e., no terminal do Ngrok, utilizando o *authtoken* fornecido no painel de controle.

2. **Criação de um Domínio Estático:** No painel do Ngrok, na seção *Cloud Edge > Domains*, cria-se um novo domínio estático gratuito.

3. **Inicialização do Túnel:** Com a aplicação local em execução (e.g., uma aplicação Flask na porta 8000), o túnel é iniciado no terminal do Ngrok (o que foi baixado anteriormente) com o seguinte comando, que direciona o tráfego do domínio público para a porta local:

   ```
   ngrok http 8000 -domain seu-dominio.ngrok-free.app
   ```

   O Ngrok exibirá a URL pública (e.g., https://weevil-cuddly-personally.ngrok-free.app), que será usada na próxima etapa.

3.6 Visão Geral da Arquitetura e Fluxo de Mensagens

O chatbot opera em um modelo de requisição-resposta, onde a API da Meta notifica a aplicação sobre eventos (como novas mensagens) através de *Webhooks*. A aplicação, por sua vez, processa esses eventos e, se for uma mensagem de usuário, gera uma resposta e a envia de volta à API da Meta.

O fluxo de uma mensagem de entrada é o seguinte:

1. Um usuário envia uma mensagem para o número do WhatsApp configurado.
2. A API da Meta recebe essa mensagem e a encaminha para o *endpoint* de Webhook da aplicação (exposto via Ngrok durante o desenvolvimento).
3. A aplicação (Flask) recebe a requisição POST do Webhook.
4. A requisição é validada e o corpo da mensagem é processado para extrair o conteúdo e os dados do remetente.
5. Uma resposta é gerada (inicialmente, apenas o texto em maiúsculas).
6. A resposta é formatada e enviada de volta à API da Meta, que a entrega ao usuário no WhatsApp.

3.7 Configuração de Variáveis de Ambiente (.env)

Um aspecto fundamental da implementação é a gestão de credenciais e configurações sensíveis através de variáveis de ambiente, carregadas a partir de um arquivo `.env`. Isso garante que informações como tokens de acesso não sejam diretamente expostas no código-fonte (EBBELAAR, 2023b).

As principais variáveis de ambiente configuradas são:

- **ACCESS_TOKEN:** Token de acesso à API da Meta para autenticar as requisições.
- **APP_ID** e **APP_SECRET:** Identificador e segredo da aplicação na Meta.
- **RECIPIENT_WAID:** O ID do WhatsApp do destinatário para testes.
- **VERSION:** A versão da API Graph da Meta utilizada (e.g., `v18.0`).
- **PHONE_NUMBER_ID:** O ID do número de telefone do bot na Meta Business Platform.
- **VERIFY_TOKEN:** Token secreto, definido pelo desenvolvedor, usado para autenticar a configuração inicial do Webhook.

3.8 Implementação das Funções Chave

O código-fonte do projeto é modularizado para separar responsabilidades, principalmente entre a definição de rotas e o processamento de mensagens, conforme a estrutura apresentada por (EBBELAAR, 2023b).

3.8.1 Gerenciamento de Rotas e Webhook (views.py)

O arquivo `views.py` é responsável por definir os *endpoints* da aplicação. Utilizando um **Blueprint** do Flask nomeado de `webhook_blueprint`, este módulo organiza as duas principais interações com o Webhook.

@webhook_blueprint.route("/webhook", methods=["GET"]) e verify() Esta rota é utilizada pela Meta para validar a URL do Webhook. A função `verify()` confere se o `hub.verify_token` recebido corresponde ao configurado, respondendo com o `hub.challenge` para confirmar a autenticidade do *endpoint* (EBBELAAR, 2023a).

@webhook_blueprint.route("/webhook", methods=["POST"]) e handle_message() Esta rota recebe as notificações de eventos em tempo real. A função `handle_message()` filtra eventos de status e, se for uma mensagem de usuário válida (verificada por `is_valid_whatsapp_message()`), invoca a função `process_whatsapp_message()` para orquestrar a resposta.

3.8.2 Utilitários e Processamento de Mensagens (whatsapp_utils.py)

Este módulo contém as funções auxiliares para processar e enviar mensagens.

process_whatsapp_message(body) Função central que extrai os dados do remetente e o conteúdo da mensagem. Chama `generate_response()`, que na implementação base apenas converte o texto para maiúsculas, servindo como um eco de teste. Por fim, formata e envia a resposta através de outras funções utilitárias.

send_message(data) Responsável por fazer a requisição HTTP POST para a API Graph da Meta, construindo os cabeçalhos de autorização com o `ACCESS_TOKEN` e enviando o payload da mensagem.

3.9 Funcionamento do Chatbot em Resposta a uma Mensagem

Quando um usuário envia uma mensagem, o ciclo de resposta, conforme implementado (EBBELAAR, 2023b,a), é o seguinte:

1. **Recebimento e Validação:** A rota POST em `views.py` recebe a requisição.
2. **Processamento:** A função `handle_message` direciona o corpo da requisição para `process_whatsapp_message`.
3. **Extração e Geração de Resposta:** Os dados do usuário e a mensagem são extraídos e a resposta é gerada.

4. **Envio:** A resposta é formatada e enviada de volta à API da Meta, que a entrega ao usuário final.

Este ciclo estabelece a base para a construção de interações mais complexas o qual foi completamente implementado seguindo a referência (EBBELAAR, 2023b). Contudo, ao decorrer do projeto, foi aperfeiçoado, uma vez que o chatbot serviu como um intermediador para se fazer o acesso (login) ao painel do website.

3.10 Banco de Dados

A fundação do sistema é um banco de dados relacional, projetado a partir do mapeamento de um modelo entidade-relacionamento para o modelo lógico. A implementação foi realizada no Sistema de Gerenciamento de Banco de Dados (SGBD) PostgreSQL, escolhido por sua robustez, extensibilidade e suporte a tipos de dados avançados, como ENUM. A estrutura relacional foi adotada para garantir a atomicidade, consistência, isolamento e durabilidade (ACID) das transações, essenciais para a integridade dos dados de agendamento.

3.10.1 Modelo Lógico e Estrutura das Tabelas

O modelo lógico foi materializado em um conjunto de tabelas inter-relacionadas, definidas utilizando a sintaxe da Database Markup Language (DBML) para clareza e melhor manutenção. A seguir, detalha-se a estrutura e o propósito de cada entidade.

Pessoa Representa a entidade base para todos os usuários do sistema, aplicando-se a especialização para médico e paciente. Esta abordagem centraliza atributos comuns a médicos e pacientes, como informações de contato e identificação, evitando a redundância de dados e simplificando futuras extensões do sistema.

- `id`: Chave primária do tipo `SERIAL`, garantindo um identificador único e autoincremental.
- `telefone_celular`: `VARCHAR`, com restrição de unicidade (`UNIQUE`) e não nulidade (`NOT NULL`), servindo como principal meio de contato e, potencialmente, como credencial de login.
- `cpf`, `email`: `VARCHAR`, com restrição de unicidade para garantir a identificação inequívoca dos usuários.
- `nome`: `VARCHAR`, obrigatório.
- `data_nascimento`: `DATE`.
- `tipo_usuario`: Utiliza o tipo `ENUM` `tipo_usuario_enum`, restrito aos valores "M" (Médico) ou "P" (Paciente), atuando como um "discriminador" para as tabelas especializadas.

Médico Entidade especializada que herda os atributos de `Pessoa`. Contém informações exclusivas do profissional de saúde. A relação um-para-um com a tabela `Pessoa` é garantida ao definir `id_pessoa` como chave primária e estrangeira simultaneamente.

- `id_pessoa`: Chave primária e estrangeira (`INTEGER`) que referencia `pessoa(id)`.
- `crm`: `VARCHAR`, obrigatório, armazena o registro do Conselho Regional de Medicina.
- `especialidade`: `VARCHAR`, obrigatória.

Paciente Entidade especializada que também herda de `Pessoa`. Seu principal propósito é armazenar o histórico de comorbidades, dados essenciais para a anamnese e análise médica.

- `id_pessoa`: Chave primária e estrangeira (`INTEGER`) referenciando `pessoa(id)`.
- `anemia_falciforme`, `hepatites`, etc.: Campos do tipo `BOOLEAN` com valor padrão `FALSE`. Esta abordagem é eficiente para um conjunto predefinido e limitado de comorbidades, facilitando consultas e filtros.

Consulta Entidade associativa que representa o agendamento, estabelecendo a relação entre um `Medico` e um `Paciente` em uma data e hora específicas.

- `id_consulta`: Chave primária do tipo `SERIAL`.
- `id_medico`: Chave estrangeira (`INTEGER`) referenciando `medico(id_pessoa)`.
- `id_paciente`: Chave estrangeira (`INTEGER`) referenciando `paciente(id_pessoa)`.
- `data_hora`: `TIMESTAMP`, tipo de dado que armazena data e hora em um único campo, otimizando o armazenamento e a consulta de agendamentos.
- `status`: Utiliza o tipo `ENUM` `status_consulta_enum`, com os valores 'agendada', 'realizada' ou 'cancelada', permitindo um controle de estado robusto para o ciclo de vida da consulta.

Neste caso, utilizou-se `id_consulta` como chave primária, sendo ela uma chave substituta conforme enunciado no capítulo 2.3.4.

HorarioDisponivel Armazena os blocos de tempo que um médico disponibiliza para atendimento. Esta tabela é fundamental para a lógica de negócio de agendamento.

- `id_medico` e `data_hora`: Compõem uma chave primária composta. Esta restrição garante, no nível do banco de dados, que é impossível para um médico cadastrar o mesmo horário como disponível mais de uma vez, prevenindo inconsistências de agendamento.

WebSession Responsável pela gestão de sessões de autenticação de usuários, como as geradas para login via sistemas externos (e.g., chatbot de WhatsApp) ou para manter o usuário logado na aplicação web.

- `id`: Chave primária do tipo `SERIAL`.
- `id_pessoa`: Chave estrangeira que referencia a tabela genérica `pessoa(id)`, permitindo que o mesmo mecanismo de sessão seja usado por médicos e pacientes.
- `login_token`: `VARCHAR`, armazena o token de sessão único e seguro.
- `expires_at`: `TIMESTAMP`, define o tempo de expiração do token, um requisito de segurança fundamental para invalidar sessões automaticamente.

Admin e ChaveMedico Estas tabelas independentes formam um subsistema de administração e segurança.

- `Admin`: Entidade para o administrador do sistema, que possui credenciais de acesso para gerenciar o sistema. O atributo `senha` deve armazenar uma representação criptográfica da senha (hash), e não o texto plano.
- `ChaveMedico`: Implementa um mecanismo de controle para o cadastro de novos médicos. Um administrador gera chaves de uso único (`valor`) que são fornecidas aos médicos. O campo `usada` controla o ciclo de vida da chave, garantindo que cada uma seja utilizada apenas uma vez.

O projeto lógico, detalhado acima, serve de base para o diagrama DBML apresentado na Figura 6, que ilustra visualmente as entidades, seus relacionamentos, chaves e cardinalidades. Por exemplo, a relação de um-para-muitos (*one-to-many*) entre `Medico` e `Consulta` indica que um médico pode ter várias consultas, mas cada consulta pertence a um único médico.

Dessa maneira, o diagrama que representa o modelo DBML, conforme segue, expressa-se assim:

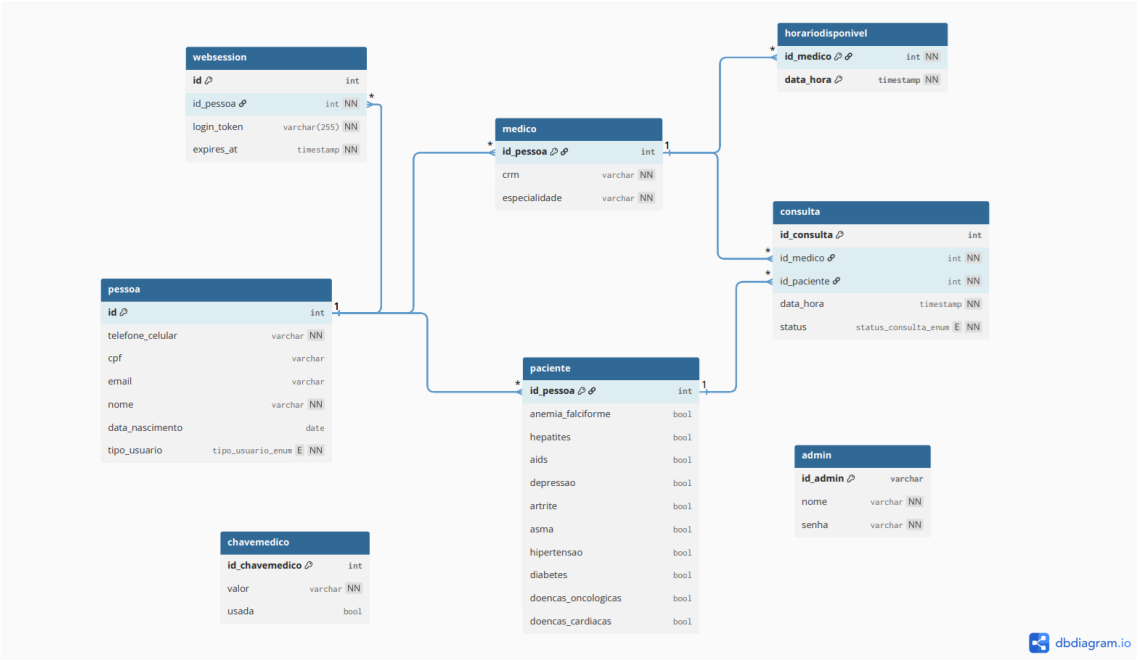

Figura 6: Diagrama Lógico do Banco de Dados em Notação DBML

Após o mapeamento para DBML, o esquema do banco de dados foi definido. A etapa subsequente foi a sua implementação física em PostgreSQL, através da tradução do modelo lógico para comandos SQL (Data Definition Language - DDL), cujo código completo pode ser encontrado no Apêndice B.

3.11 Classes na Programação Orientada a Objetos

Optou-se pelo modelo de programação orientada a objetos devido ao fato de que cada tipo de usuário — médico, paciente e administrador — possui responsabilidades e permissões distintas. Cada classe encapsula atributos e métodos específicos, garantindo que o acesso e a manipulação do banco de dados sejam controlados de acordo com o tipo de usuário.

Para representar essas relações de forma clara, foi elaborado um diagrama UML que evidencia os atributos e métodos de cada classe, bem como a interação entre elas.

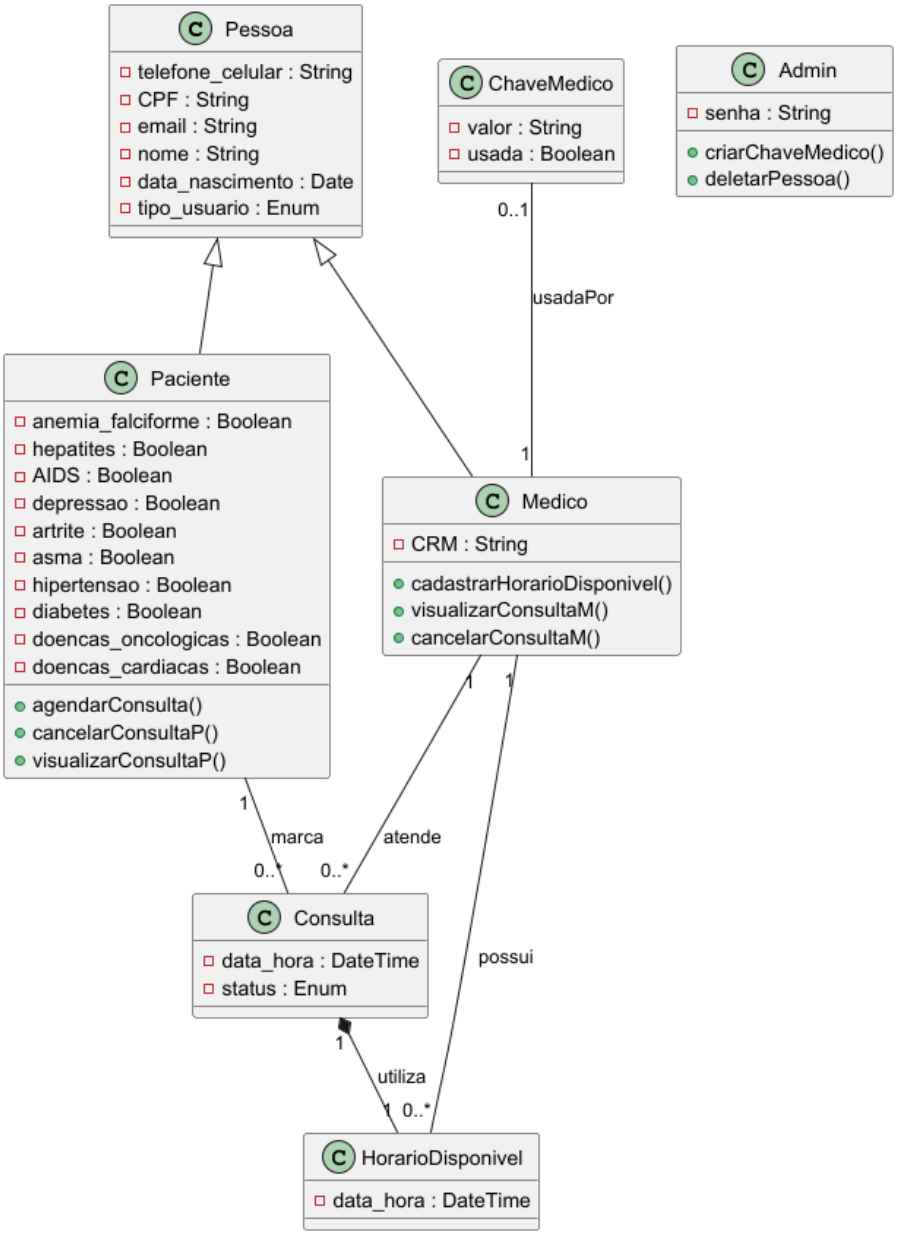

Figura 7: Diagrama UML das classes e suas interações entre si

3.11.1 Comentários sobre o diagrama UML

Evidencia-se que os métodos do Paciente `cancelarConsultaP()` e `visualizarConsultaP()` possuem a escrita semelhante aos métodos do Médico `cancelarConsultaM()` e `visualizarConsultaM()`. Isso é devido à diferença que será aplicada ao projeto para um agendamento e uma visualização de consultas feitas pelo paciente e as mesmas ações pelo médico.

3.12 Implementação das Classes em SQLAlchemy

O modelo conceitual, definido no diagrama UML da Figura 7, é traduzido para uma implementação concreta utilizando a linguagem Python e o framework SQLAlchemy. Essa abordagem de Mapeamento Objeto-Relacional (ORM) permite representar as

entidades do sistema, como `Pessoa`, `Medico` e `Paciente`, como classes Python que se correspondem diretamente a tabelas no banco de dados, garantindo modularidade e um controle de acesso a dados estruturado.

Esta seção detalha a arquitetura do modelo ORM implementado.

3.12.1 Estratégia de Chaves e Identificadores

Para garantir a integridade referencial, o desempenho das consultas e a privacidade dos dados, o modelo adota chaves substitutas como padrão.

3.12.2 O Padrão Implementado: id numérico

Todas as entidades principais do sistema são identificadas por um id numérico, i.e. chaves substitutas, sequencial e não sensível. A tabela `Pessoa`, por exemplo, utiliza a coluna `id` como sua chave primária.

Essa abordagem traz vantagens críticas para a arquitetura:

- **Privacidade:** Dados sensíveis, como o telefone ou CPF, não são utilizados como identificadores em chaves estrangeiras ou expostos em APIs, mitigando riscos de segurança.
- **Desempenho:** Operações de junção (*join*) em colunas de tipo `Integer` são significativamente mais rápidas e eficientes do que em colunas de texto (`String`).
- **Estabilidade:** O id de um usuário é imutável. Mesmo que um usuário altere seu número de telefone, a chave primária permanece a mesma, eliminando a necessidade de atualizações em cascata e prevenindo inconsistências nos dados.

Dados que precisam ser únicos, como `telefone_celular`, `cpf` e `email`, são mantidos com uma restrição de unicidade (*unique constraint*), garantindo a consistência sem sobrecarregá-los com a função de chave primária.

```
class Pessoa(Base):
    __tablename__ = "pessoa"
    id = Column(Integer, primary_key=True)
    telefone_celular = Column(String, unique=True, nullable=
   False)

    cpf = Column(String, unique=True)
    email = Column(String, unique=True)
    nome = Column(String, nullable=False)
    data_nascimento = Column(Date)
    tipo_usuario = Column(Enum(TipoUsuarioEnum), nullable=
   False)
```

Listagem 4: Definição da classe `Pessoa` com chave primária numérica.

3.12.3 Estrutura de Herança: Pessoa → Médico/Paciente

O modelo implementa um padrão de herança por tabela associada. As classes `Medico` e `Paciente` são tratadas como especializações da classe `Pessoa`, compartilhando seus

atributos de base. Não só isso, pode-se dizer que Pessoa é uma classe abstrata, pois nenhum usuário é definido como Pessoa.

Essa estrutura é estabelecida utilizando a chave primária `pessoa.id` como chave primária e estrangeira nas tabelas `medico` e `paciente`. Isso cria uma relação de um-para-um, garantindo que todo `Medico` ou `Paciente` seja, antes de tudo, uma `Pessoa` no sistema.

```
class Medico(Base):
    __tablename__ = "medico"
    id_pessoa = Column(Integer, ForeignKey("pessoa.id"), primary_key=True)

    crm = Column(String, nullable=False)
    especialidade = Column(String, nullable=False)

    pessoa = relationship("Pessoa", back_populates="medico")

class Paciente(Base):
    __tablename__ = "paciente"
    id_pessoa = Column(Integer, ForeignKey("pessoa.id"), primary_key=True)

    pessoa = relationship("Pessoa", back_populates="paciente")
```

Listagem 5: Estrutura de herança para `Medico` e `Paciente`.

Esta abordagem normaliza a estrutura do banco de dados, evita a duplicação de informações (como nome e CPF) e organiza o modelo de forma lógica e escalável.

3.12.4 Camada de Acesso a Dados e Lógica de Negócio

Todas as interações com o banco de dados são encapsuladas em funções de CRUD (Create, Read, Update, Delete), que operam sobre os ids numéricos das entidades. Essa camada de serviço garante que a lógica de negócio seja aplicada de forma consistente.

Por exemplo, a criação de um `Medico` é associada a uma `Pessoa` existente através do `id_pessoa`, tornando a relação explícita e segura.

```
def create_medico(session: Session, id_pessoa: int, crm: str, especialidade: str):
    medico = models.Medico(id_pessoa=id_pessoa, crm=crm, especialidade=especialidade)
    session.add(medico)
    session.commit()
    session.refresh(medico)
    return medico
```

Listagem 6: Exemplo de função CRUD que utiliza o id para criar uma associação.

Além disso, operações críticas como o agendamento de consultas são projetadas para serem **atômicas**. A função `create_consulta` verifica a existência de um horário, remove-o da tabela de disponibilidade e cria o registro da consulta dentro da

mesma transação. Isso previne condições de corrida (*race conditions*) e garante a integridade do agendamento.

3.12.5 Otimização de Respostas para a API

As funções responsáveis por fornecer dados para o front-end, i.e., website foram projetadas para retornar dicionários Python em vez de objetos SQLAlchemy.

```
def listar_consultas_paciente(session: Session, id_paciente: int):
    consultas_query = session.query(models.Consulta)\
        .join(models.Medico, models.Consulta.id_medico == models.Medico.id_pessoa)\
        .join(models.Pessoa, models.Medico.id_pessoa == models.Pessoa.id)\
        .filter(models.Consulta.id_paciente == id_paciente)\
        .all()

    resultado = []
    for c in consultas_query:
        resultado.append({
            "id_consulta": c.id_consulta,
            "nome_medico": c.medico.pessoa.nome,
            "data_hora": c.data_hora.isoformat(),
            "status": c.status.value
        })

    return resultado
```

Listagem 7: Função otimizada para listar consultas, retornando um dicionário JSON-serializável.

Esta prática oferece dois benefícios principais:

1. **Desempenho:** Evita o problema de consultas N+1, pois todos os dados necessários (como o nome do médico) são carregados em uma única consulta com `join`.
2. **Simplicidade:** Garante que os dados sejam facilmente serializáveis para JSON e desacopla a representação da API da estrutura interna do modelo ORM.

3.12.6 Conclusão da Implementação

O modelo de dados implementado, baseado em chaves substitutas numéricas e uma clara estrutura de herança, resulta em um sistema seguro, eficiente e de fácil manutenção. As otimizações na camada de acesso a dados garantem a consistência e a performance da aplicação, estabelecendo uma base sólida para futuras expansões.

3.13 Implementação do Sistema Web para cadastros

O objetivo deste projeto foi criar um sistema web para cadastro de usuários (médicos e pacientes) no banco de dados, garantindo que o frontend (`cadastro.html`) se

comunique corretamente com o backend em Flask (`cadastro.py`) e que o backend aceite requisições do frontend sem problemas de CORS ou `Failed to fetch`.

3.14 Implementação do Módulo de Cadastro

O módulo de cadastro é um componente central do sistema, sendo a porta de entrada para novos usuários, sejam eles pacientes ou médicos. A sua implementação foi projetada com uma arquitetura desacoplada, dividida em três camadas principais: a interface de conversação (chatbot), a interface web (frontend) e o serviço de aplicação (backend). Essa separação garante manutenibilidade, escalabilidade e uma experiência de usuário fluida.

3.14.1 Estrutura do Backend e Endpoints da API

O núcleo do sistema é uma API RESTful desenvolvida em Python com o framework Flask, orquestrada no arquivo `run.py`. Este servidor centraliza toda a lógica de negócio, interações com o banco de dados e comunicação com outras interfaces.

3.14.2 Endpoint de Cadastro (`/cadastro`)

O principal endpoint para o registro de novos usuários é o `/cadastro`, que aceita requisições do tipo `POST`. Sua implementação segue um fluxo rigoroso para garantir a integridade e segurança dos dados:

1. **Recebimento e Validação:** A rota recebe os dados do usuário em formato JSON. A primeira etapa é validar a presença de todos os campos obrigatórios (telefone, CPF, email, nome, data de nascimento e tipo de usuário). Caso algum campo esteja ausente, a API retorna um erro `400 (Bad Request)`.

2. **Verificação de Duplicidade:** Para evitar registros duplicados, o sistema consulta o banco de dados através da função `crud.get_pessoa_by_telefone()` para verificar se o número de telefone informado já existe. Se positivo, um erro `409 (Conflict)` é retornado ao cliente.

3. **Criação da Entidade Base:** Se a validação for bem-sucedida, uma nova entrada é criada na tabela `pessoa` utilizando `crud.create_pessoa()`. Esta entidade armazena as informações comuns a todos os usuários.

4. **Criação da Entidade Especializada:** Com base no campo `tipo_usuario` (`'P'` para Paciente ou `'M'` para Médico), o sistema cria um registro na tabela correspondente (`paciente` ou `medico`).

    - Para **Pacientes**, são registrados os dados de saúde opcionais (anemia falciforme, diabetes, etc.), obtidos do formulário.
    - Para **Médicos**, é validada a presença obrigatória dos campos `crm` e `especialidade`, essenciais para o perfil profissional.

5. **Gerenciamento de Transação:** Todas as operações de banco de dados são executadas dentro de um bloco `try/except`. Em caso de qualquer falha durante o processo, a transação é revertida (`db.rollback()`) para garantir que

o banco de dados não fique em um estado inconsistente. A sessão com o banco é sempre fechada no bloco `finally`.

6. **Resposta ao Cliente:** Ao final do processo, uma resposta JSON é enviada ao frontend, indicando sucesso (`HTTP 201 Created`) ou falha (`HTTP 500 Internal Server Error`), permitindo que a interface do usuário forneça um feedback claro.

Para permitir a comunicação entre o domínio do frontend (`medcoqueiral.com.br`) e a API, foi utilizado o pacote `Flask-CORS`, configurado para aceitar requisições de origem cruzada (CORS) de forma segura.

3.14.3 Estrutura da Interface de Cadastro (Frontend)

A interface de cadastro (`cadastro.html`) foi construída com HTML5 e estilizada com a biblioteca `Tailwind CSS` para criar um formulário moderno, responsivo e intuitivo. A interatividade é gerenciada por JavaScript, que executa as seguintes funções:

1. **Preenchimento dinâmico:** ao carregar a página, o script extrai o número de telefone do usuário a partir dos parâmetros da URL (`?telefone=...`) e o insere automaticamente em um campo oculto no formulário. Isso conecta a interação inicial do chatbot diretamente ao formulário web.

2. **Visibilidade condicional:** O formulário se adapta dinamicamente ao tipo de usuário selecionado. Se o usuário seleciona "Médico", os campos para CRM e especialidade se tornam visíveis e obrigatórios. Se escolhe "Paciente", a seção com *checkboxes* para condições de saúde é exibida.

3. **Processamento e Envio de Dados:** No momento da submissão:

   - O JavaScript intercepta o evento, prevenindo o recarregamento da página.
   - Ele coleta todos os dados do formulário, convertendo os valores das *checkboxes* de condições de saúde para `true` ou `false`.
   - Os dados são empacotados em um objeto JSON.
   - Uma requisição `fetch` do tipo `POST` é enviada para o backend através de um script `proxy.php`. Este proxy serve como uma camada de comunicação segura entre a interface do usuário (frontend) e a API, sendo responsável por direcionar a chamada para o endpoint `/cadastro` no servidor da aplicação.

4. **Feedback ao Usuário:** O script aguarda a resposta do backend. Com base no status e no conteúdo da resposta, uma mensagem de sucesso ou erro é exibida dinamicamente na tela, informando o usuário sobre o resultado de sua solicitação.

3.15 Fluxo de Cadastro do Usuário: Uma Visão Integrada

O processo de cadastro foi projetado para ser uma jornada contínua e intuitiva, começando no WhatsApp e terminando na página web.

1. **Início da Interação (Chatbot):** O usuário inicia uma conversa com o chatbot no WhatsApp. Ao selecionar a opção de "Login/Registro", a lógica em `whatsapp_utils.py` é acionada. O sistema verifica no banco de dados, através da função `crud.get_pessoa_by_telefone()`, se o número de telefone (`wa_id`) do usuário já está registrado.

2. **Direcionamento Inteligente:**

   - **Usuário Existente:** Se o telefone já está cadastrado, o chatbot gera um token de acesso único e temporário através da função `crud.create_web_session()`. Em seguida, envia ao usuário um link de login seguro, como: `https://medcoqueiral.com.br/login.php?telefone={numero}&token={token}` conforme a Figura 15.
   - **Novo Usuário:** Se o telefone não for encontrado, o chatbot identifica a necessidade de um novo cadastro e envia ao usuário um link para a página de registro, passando o número do celular como parâmetro na URL: `https://medcoqueiral.com.br/cadastro.html?telefone={numero}` conforme a Figura 11.

3. **Preenchimento do Formulário:** O usuário clica no link e acessa a página `cadastro.html`. O campo de telefone já está preenchido, e ele completa os demais dados (nome, CPF, tipo de usuário, etc.) conforme a Figura 12.

4. **Submissão para a API:** Ao clicar em "Cadastrar", o JavaScript da página envia os dados em formato JSON para o backend via `fetch`. A requisição é direcionada ao endpoint `/cadastro` da API Flask.

5. **Processamento e Persistência:** O backend executa todas as validações e lógicas de negócio descritas anteriormente, criando os registros `Pessoa` e `Paciente`/`Medico` no banco de dados PostgreSQL.

6. **Confirmação Final:** O backend retorna uma mensagem de sucesso. O frontend a exibe para o usuário, informando que o cadastro foi concluído e que ele pode retornar ao chatbot para efetuar o login conforme a Figura 13.

Este fluxo integrado assegura que os dados sejam coletados de forma estruturada, validados de maneira robusta e persistidos com segurança, proporcionando uma base sólida para todas as outras funcionalidades do sistema.

3.16 Consultas e Interações do Usuário Autenticado

Uma vez que o usuário está cadastrado e autenticado no sistema, ele pode realizar consultas e interações com os dados que lhe são pertinentes. O fluxo de comunicação para essas operações segue um padrão seguro e rastreável, que se inicia na interface do usuário e percorre todas as camadas até o banco de dados.

A seguir, detalha-se o passo a passo de uma requisição de dados típica: a visualização de médicos disponíveis por um paciente logado.

3.16.1 Fluxo de Criação de Horários Disponíveis

Este fluxo descreve a ação de um médico ao adicionar novos horários de atendimento em sua agenda, tornando-os visíveis para os pacientes.

1. **Ação do Usuário (Médico):** No seu painel de controle (`painel_medico.php`), o médico seleciona uma data e hora no campo de calendário e clica no botão "Adicionar". A ação dispara a função JavaScript `addHorario()`.

Listagem 8: Chamada para adicionar um novo horário.

```
function addHorario() {
    const input = document.getElementById("inputHorario")
        .value;
    if (!input) { return; }

    makeApiCall('adicionar_horario', 'POST', { data_hora:
        input });
}
```

2. **Proxy e Autenticação:** A requisição `POST` é enviada ao `proxy.php`. O script valida a sessão do médico, identifica seu `id_medico` e encaminha a requisição, junto com o corpo JSON contendo a data e hora, para o backend.

3. **API Backend:** A rota Flask correspondente recebe a chamada, extrai o `id_medico` da URL e a `data_hora` do corpo da requisição.

Listagem 9: Rota Flask para adicionar horário.

```
@app.route("/adicionar_horario", methods=["POST"])
def adicionar_horario():
    id_medico = request.args.get('id_medico', type=int)
    data = request.get_json()
    data_hora_str = data.get("data_hora")

    # Converte a string para datetime e chama a camada
        CRUD
    data_hora = datetime.fromisoformat(data_hora_str)
    crud.adicionar_horario_medico(db, id_medico,
        data_hora)

    return jsonify({"success": True})
```

4. **Persistência no Banco de Dados:** A função `crud.adicionar_horario_medico` cria uma nova instância do modelo `HorarioDisponivel`, associando-a ao `id_medico`, e a salva no banco de dados.

5. **Confirmação e Atualização da UI:** O backend retorna uma mensagem de sucesso. O frontend a exibe para o médico e, em seguida, invoca a função `loadHorarios()` para recarregar e exibir a lista de horários atualizada, incluindo o que acabou de ser adicionado.

3.16.2 Fluxo de Agendamento de Consulta (Paciente)

Este é um dos fluxos mais críticos, pois envolve a seleção de um horário, um processo de pagamento e a garantia de que o agendamento seja único e à prova de falhas. Dessa maneira, ele deve ser atômico, i.e., ou ele acontece em sua totalidade, ou ele não acontece. O processo é dividido em duas fases principais: a iniciação do pagamento e a confirmação assíncrona.

Fase 1: Iniciação do Pagamento

1. **Ação do Usuário (Paciente):** No `painel_paciente.php`, após visualizar os horários disponíveis de um médico, o paciente clica em um dos botões de horário. Esta ação aciona a função JavaScript `iniciarPagamento()`, passando os dados da consulta desejada.

Listagem 10: Iniciação do fluxo de pagamento no frontend.

```
function iniciarPagamento(idMedico, nomeMedico, dataHora) {
    makeApiCall('create-checkout-session', 'POST', {
        id_medico: idMedico,
        nome_medico: nomeMedico,
        data_hora: dataHora
    });
}
```

2. **Proxy e API Backend:** A requisição `POST` é enviada ao `proxy.php`, que valida a sessão do paciente e anexa seu `id_paciente` à URL. A API Flask, na rota `/create-checkout-session`, recebe os dados.

3. **Criação da Sessão de Pagamento:** O backend **não agenda a consulta ainda**. Em vez disso, ele se comunica com a API da Stripe, criando uma sessão de checkout segura. Crucialmente, os dados da consulta (`id_paciente`, `id_medico`, `data_hora`) são armazenados no campo `metadata` da sessão da Stripe. O backend então retorna o ID dessa sessão de checkout para o frontend.

4. **Redirecionamento para Pagamento:** O JavaScript do frontend recebe o ID da sessão da Stripe e utiliza a biblioteca `Stripe.js` para redirecionar o usuário para a página de pagamento segura da Stripe, finalizando a interação com o sistema temporariamente.

Fase 2: Confirmação Assíncrona via Webhook

5. **Confirmação da Stripe:** Após o paciente concluir o pagamento com sucesso na página da Stripe, a Stripe envia uma notificação automática e assíncrona (um *webhook*) para um endpoint específico no backend: `/stripe-webhook`. Esta comunicação ocorre de servidor para servidor, independentemente do navegador do usuário.

6. **Processamento do Webhook:** A rota `/stripe-webhook` no backend:

    - Verifica a assinatura da requisição para garantir que ela veio da Stripe.

- Extrai o `metadata` que foi salvo na Fase 1, recuperando o `id_paciente`, `id_medico` e `data_hora`.
- Agora, com a confirmação do pagamento, invoca a função `crud.create_consulta()`.

Listagem 11: Lógica do webhook para confirmar o agendamento.

```
@app.route('/stripe-webhook', methods=['POST'])
def stripe_webhook():

    if event['type'] == 'checkout.session.completed':
        metadata = event['data']['object']['metadata']
        id_paciente = int(metadata.get('id_paciente'))
        id_medico = int(metadata.get('id_medico'))
        data_hora = datetime.fromisoformat(metadata.get('
           data_hora'))
        crud.create_consulta(db, id_medico, id_paciente,
           data_hora)

    return 'Success', 200
```

7. **Agendamento Atômico no Banco de Dados:** A função `crud.create_consulta` executa uma transação atômica: ela primeiro localiza e **exclui** o registro correspondente da tabela `HorarioDisponivel` e, em seguida, **cria** um novo registro na tabela `Consulta`. Este processo garante que um mesmo horário não possa ser agendado duas vezes.

8. **Feedback ao Usuário:** Independentemente do webhook, a Stripe redireciona o navegador do paciente de volta para o sistema, para a URL de sucesso (`.../painel_paciente.php?payment=success`). O JavaScript nesta página detecta o parâmetro na URL, exibe uma mensagem de "Pagamento aprovado! Sua consulta foi agendada." e atualiza a lista de "Minhas Consultas" do paciente.

3.16.3 Fluxo de Requisição de Dados: Paciente Visualizando Médicos

Este processo descreve como as informações de médicos e seus horários são solicitadas, processadas e exibidas no painel do paciente.

1. **Iniciação no Frontend:** Ao carregar a página `painel_paciente.php`, o script do lado do cliente executa a função `loadMedicosDashboard()` para iniciar a busca dos dados a serem exibidos.

2. **Chamada Assíncrona via *fetch*:** A função JavaScript invoca a chamada à API, que cria e envia uma requisição `GET` para o script intermediário, através do caminho `/proxy.php/listar_medicos_dashboard`. Neste momento, a requisição ainda não contém a identidade do paciente.

3. **Autenticação e Encaminhamento no Proxy:** O servidor web recebe a chamada no `proxy.php`. Este script executa duas ações críticas:

- **Validação de Sessão:** Verifica a sessão PHP ativa para garantir que o usuário está autenticado, lendo o valor de `$_SESSION['id_pessoa']`. Se a sessão for inválida, o acesso é negado com status `401 Unauthorized`.
- **Enriquecimento da Requisição:** Sendo a sessão válida, o proxy anexa o `id` do usuário como parâmetro de consulta (*query parameter*) na URL de destino. Exemplo: `.../listar_medicos_dashboard?id_paciente=123`.

O proxy então utiliza a biblioteca `cURL` do PHP para encaminhar a requisição, agora autenticada, para o servidor da aplicação Flask.

4. **Processamento na API Backend:** A API Flask, no arquivo `run.py`, recebe a requisição no endpoint correspondente. A rota extrai o `id_paciente` e delega a tarefa à camada de acesso a dados.

Listagem 12: Rota Flask para listar médicos disponíveis.

```
@app.route('/listar_medicos_dashboard')
def listar_medicos_dashboard():
    id_paciente = request.args.get("id_paciente")
    return crud.listar_medicos_com_horarios(id_paciente)
```

5. **Execução da Consulta no Banco de Dados:** A função na camada CRUD executa a consulta otimizada (vide Seção 7), realizando junções entre as tabelas `pessoa`, `medico` e `horario_disponivel`. O resultado é formatado como uma lista de dicionários.

6. **Retorno da Resposta:** A API Flask serializa os dados em JSON e os envia como resposta HTTP. Enquanto o `proxy.php` retransmite integralmente essa resposta ao navegador do cliente.

7. **Renderização Dinâmica na Interface:** A promessa (promise) da função *fetch* no JavaScript do `painel_paciente.php` é resolvida, recebendo o JSON e construindo dinamicamente os elementos HTML para exibir as especialidades, nomes dos médicos e seus horários disponíveis.

3.16.4 Fluxo de Manipulação de Dados: Médico Atualizando Consulta

O fluxo para manipulação de dados (criação, atualização ou exclusão) é semelhante, mas utiliza métodos HTTP diferentes, como o `POST`. Um exemplo é quando um médico marca uma consulta como *Realizada*.

1. **Ação do Usuário:** No painel (`painel_medico.php`), o médico clica no botão "Realizada". Uma função JavaScript é acionada para enviar a requisição de atualização.

Listagem 13: Chamada para marcar consulta como realizada.

```
function marcarRealizada(idConsulta) {
    makeApiCall('marcar_realizada', 'POST', { id_consulta
        : idConsulta });
}
```

2. **Proxy:** O `proxy.php` valida a sessão, anexa o `id_medico` à URL e encaminha a requisição `POST`, incluindo o corpo JSON que identifica a consulta a ser alterada.

3. **API Backend:** A rota correspondente em Python é ativada, extraindo os dados da requisição para processamento.

Listagem 14: Rota Flask para atualizar o status da consulta.

```
@app.route('/marcar_realizada', methods=['POST'])
def marcar_realizada():
    id_medico = request.args.get("id_medico")
    id_consulta = request.json.get("id_consulta")
    return crud.marcar_consulta_realizada(id_consulta,
       id_medico)
```

4. **Execução da Lógica de Negócio:** A função na camada CRUD verifica se o médico tem permissão para alterar a consulta e, em caso afirmativo, atualiza o status da mesma no banco de dados para "realizada".

5. **Confirmação e Atualização da UI:** O backend retorna uma mensagem de sucesso. O frontend a recebe, exibe uma notificação para o usuário e atualiza a interface, movendo a consulta da lista de "Agendadas" para o "Histórico".

Esses fluxos demonstram como o sistema opera de forma segura e eficiente, garantindo que cada usuário interaja apenas com os dados aos quais tem permissão, mantendo a integridade e a consistência em toda a aplicação.

3.17 Procedimentos de testes

Para a realização dos testes do sistema de chatbot e do ecossistema de atendimento, foi configurado um ambiente de desenvolvimento que permitisse a comunicação em tempo real com a API de pagamentos da Stripe. Os procedimentos para a configuração deste ambiente foram os seguintes:

1. **Inicialização do Servidor Local:** A aplicação foi iniciada executando o script `run.py`, que disponibiliza o serviço na porta `8000`.

2. **Criação de um Túnel com Ngrok:** Para expor a aplicação local à internet e permitir o recebimento de webhooks, foi utilizado o `ngrok` com o seguinte comando, que cria um URL público e o redireciona para a porta `8000` local:

```
ngrok http 8000 --domain weevil-cuddly-personally.ngrok-free.app
```

3. **Encaminhamento de Eventos da Stripe:** Utilizando a CLI da Stripe, foi estabelecido um monitoramento de eventos que os redirecionava para o servidor local. Após a autenticação com `stripe login`, o comando abaixo foi executado para encaminhar os webhooks para a rota `/stripe-webhook` da aplicação:

```
stripe listen --forward-to localhost:8000/stripe-webhook
```

Este fluxo permitiu testar de forma eficaz a integração com o sistema de pagamentos, simulando se quando o pagamento fosse aprovado, as consultas seriam marcadas automaticamente.

4 Conclusão

O trabalho foi muito extenso, muitas habilidades foram desenvolvidas desde o design de banco de dados como modelo entidade relacionamento (DER) até sua implementação em SQL e a utilização de programação orientada a objetos para gerir funções específicas de cada tipo de usuário. Assim como o aprendizado de diferentes linguagens utilizadas e a utilização do framework Flask como uma ponte de conexão (por meio dos endpoints) entre um servidor que está em hosting remoto (Locaweb) e um servidor local (localhost) por meio de um domínio estático público exposto via Ngrok.

Apesar de o trabalho ter sido desenvolvido primariamente para agendamento de consultas entre médicos e pacientes, ele pode ser utilizado para qualquer tipo de agendamento sem muito esforço, desde agendamentos de viagens entre passageiro e companhia aérea a agendamentos entre qualquer tipo de profissional e um cliente. Apenas seria necessário criar uma nova entidade (por consequência uma nova classe na programação orientada a objetos) no caso das passagens aéreas, isto é, a entidade avião seria criada e relacionar-se-ia a HorariosDisponiveis. Além disso, aviões relacionar-se-iam a consultas e pacientes (passageiros). Poucas mudanças seriam necessárias no banco de dados, nas classes de programação orientada a objetos e no CRUD.

O projeto, de certa maneira, não explorou tecnologias novas e ficou mais focado na tentiva de replicar as melhores práticas daquilo que já estava muito bem consolidado e utilizado extensivamente no mercado. Entretanto, a base do projeto pode ser muito bem explorada em projetos futuros e os conhecimentos adquiridos de como ele funciona serão muito bem utilizados para desenvolver aplicações mais inteligentes.

Ao finalizar o trabalho, alguns objetivos não foram cumpridos:

1. Painel de administrador
2. Chave de acesso para se criar um usuário do tipo médico (os médicos são criados da mesma maneira que pacientes, logo qualquer pessoa pode se registrar como médico)
3. Escalabilidade
4. Chatbot integrado à OpenAI API
5. Criptografia dos dados sensíveis

O painel de administrador não foi criado e a chave de acesso que seria criada pelo administrador para um novo médico ser registrado também não. Contudo, os médicos podem se registrar.

Médicos se registram da mesma maneira que pacientes, o que é um problema sério, pois os pacientes não sabem quem são os médicos de verdade.

A escalabilidade não foi alcançada pelo fato de não ter sido realizado nem a dockerização nem a colocação dos arquivos na Google Cloud (GCP).

A conexão da OpenAI API ao chatbot para que ele fosse possível recomendar médicos também não foi implementada.

Dados sensíveis não foram criptografados no banco de dados de tal maneira que se um agente malicioso conseguisse acesso ao banco de dados, este teria acesso a

todos os dados dos usuários registrados. Contudo, não teria acesso à conta (painel médico ou painel paciente), pois o acesso do usuário é realizado por meio de uma chave de acesso único que é gerada ao se comunicar com o chatbot do WhatsApp.

Os resultados principais foram cumpridos:

1. Website completamente funcional com todas as páginas responsivas para dar caráter sério aos visitantes.
2. Ecossistema de Login e Cadastro completamente implementados
3. Agendamento de horários entre médico e paciente implementados
4. Protótipo de verificação de pagamento utilizando a Stripe API no modo teste completamente testado e funcional

4.1 Trabalhos Futuros

Espera-se, no futuro, um estudo aprofundado de certos temas para que o chatbot fique mais inteligente, sendo eles:

1. Estudo a respeito das tecnologias de Docker e Cloud (GCP ou AWS) para se escalar o projeto, i.e., deixá-lo funcionando 24 horas por dia ininterruptamente.
2. Estudo aprofundado de *colaborative filtering* (CV) para implementar um sistema recomendacional o qual recomendará médicos para os pacientes registrados a partir dos médicos que esses pacientes já se consultaram anteriormente.
3. Estudo aprofundado de inteligência artificial aplicada a NLP para conseguir dar respostas inteligentes ou recomendações aos usuários a partir de texto livre, por exemplo: "um usuário fala que está com dor de cabeça, febre alta e dor de barriga", o chatbot recomendá-lo-á que se consulte com um infectologista ou gastroentologista. Espera-se que se estude *Large Language Models* extensivamente e aprenda a realizar *fine-tuning* e *evaluation* do modelo.
4. Implementação da criptografia nos dados armazenados no banco de dados.
5. Estudo de como funciona a OpenAI API e possibilidade de implementá-la a um custo que corresponda ao orçamento do projeto.

Referências

EBBELAAR, Dave. **How To Connect OpenAI To WhatsApp (Python Tutorial) [vídeo]**. [S.l.: s.n.], 2023. https://www.youtube.com/watch?v=3YPeh-3AFmM&t=1152s. Acesso em: 12 ago. 2025.

_____. **Python-whatsapp-bot [repositório]**. [S.l.: s.n.], 2023. https://github.com/daveebbelaar/python-whatsapp-bot/tree/main. Acesso em: 12 ago. 2025.

HEUSER, Carlos Alberto. **Projeto de Banco de Dados**. 4. ed. Porto Alegre: Sagra Luzzatto, 2009.

META. **WhatsApp Business API**. [S.l.: s.n.], 2025. https://developers.facebook.com/docs/whatsapp. Acesso em: 12 ago. 2025.

META FOR DEVELOPERS. **Get Opt-In for WhatsApp**. Acesso em: 24 set. 2025. 2025. Disponível em: <https://developers.facebook.com/docs/whatsapp/overview/getting-opt-in>. Acesso em: 24 set. 2025.

_____. **Get Started, On-Premises API**. Acesso em: 24 set. 2025. 2025. Disponível em: <https://developers.facebook.com/docs/whatsapp/on-premises/get-started/>. Acesso em: 24 set. 2025.

_____. **Pricing**. Acesso em: 24 set. 2025. 2025. Disponível em: <https://developers.facebook.com/docs/whatsapp/pricing/>. Acesso em: 24 set. 2025.

RICHARDS, Mark; FORD, Neal. **Fundamentals of Software Architecture: An Engineering Approach**. [S.l.]: O'Reilly Media, Inc., 2020. ISBN 978-1492043454.

SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN, S. **Database System Concepts**. 6. ed. New York: McGraw-Hill, 2010.

_____. **Sistemas de Banco de Dados**. 5. ed. Porto Alegre: Bookman, 2006. Tradução da 5ª edição.

TECLADO. **What is REST API?** [S.l.: s.n.]. https://rest-apis-flask.teclado.com/docs/course_intro/what_is_rest_api/. Acesso em: 12 ago. 2025.

ZENKINS. **.NET vs GraphQL for API-Driven Applications**. 2025. Disponível em: <https://zenkins.com/insights/net-vs-graphql-for-api-driven-applications/>. Acesso em: 12 ago. 2025.

5 Apêndices

A Código C para gerenciamento de estudantes em TXT

O Apêndice A apresenta o código completo em C que implementa um CRUD simples utilizando arquivos TXT, incluindo verificação de ID como chave primária e menu interativo.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Funcao para verificar se o ID ja existe
int id_existe(int id) {
    FILE *f = fopen("estudantes.txt", "r");
    if (!f) return 0; // arquivo nao existe ainda

    char line[100];
    char name[80];
    int id_existente;

    while (fgets(line, sizeof(line), f)) {
        if (sscanf(line, "%79[^0-9] %d", name, &id_existente)
            == 2) {
            if (id_existente == id) {
                fclose(f);
                return 1; // ID ja existe
            }
        }
    }

    fclose(f);
    return 0; // ID nao encontrado
}

int main() {
    FILE *f;
    char line[100];
    char name[80];
    int id;
    int option;
    char buffer[100];

    while (1) {
        printf("\nMenu:\n");
        printf("1 - Listar estudantes\n");
        printf("2 - Adicionar estudante\n");
        printf("3 - Sair\n");
        printf("Escolha uma opcao: ");

```

```
42          fgets(buffer, sizeof(buffer), stdin);
43          option = atoi(buffer); // converte para inteiro
44
45          if (option == 1) {
46              f = fopen("estudantes.txt", "r");
47              if (!f) {
48                  printf("Nenhum estudante cadastrado ainda.\n"
                        );
49                  continue;
50              }
51
52              printf("\nLista de estudantes:\n");
53              while (fgets(line, sizeof(line), f)) {
54                  if (sscanf(line, "%79[^0-9] %d", name, &id)
                        == 2) {
55                      printf("Nome: %s, ID: %d\n", name, id);
56                  }
57              }
58
59              fclose(f);
60
61          } else if (option == 2) {
62              // Ler nome
63              printf("Digite o nome: ");
64              fgets(name, sizeof(name), stdin);
65              name[strcspn(name, "\n")] = 0; // remove \n
66
67              // Ler ID
68              printf("Digite o ID: ");
69              fgets(buffer, sizeof(buffer), stdin);
70              id = atoi(buffer);
71
72              // Verifica se o ID ja existe
73              if (id_existe(id)) {
74                  printf("Erro: ja existe um estudante com o ID
                        %d.\n", id);
75                  continue;
76              }
77
78              // Adicionar no arquivo
79              f = fopen("estudantes.txt", "a");
80              if (!f) {
81                  printf("Erro ao abrir o arquivo!\n");
82                  continue;
83              }
84
85              fprintf(f, "%s %d\n", name, id);
86              fclose(f);
87
88              printf("Estudante adicionado com sucesso!\n");
89
```

```
90          } else if (option == 3) {
91              printf("Saindo...\n");
92              break;
93
94          } else {
95              printf("Opcao invalida!\n");
96          }
97      }
98
99      return 0;
100 }
```

B Código SQL para criar o banco de dados

O Apêndice B apresenta o código completo em PostgreSQL que implementa o SQL para se criarem as tabelas necessárias para armazenar os dados necessários para se fazer o login no ecossistema de atendimentos, i.e., tanto no chatbot quanto no website:

Código SQL em Anexo

```
-- ========================
-- CREATE ENUMS (No Changes)
-- ========================
CREATE TYPE tipo_usuario_enum AS ENUM ('M', 'P');
CREATE TYPE status_consulta_enum AS ENUM ('agendada', 'realizada',
↪  'cancelada');

-- ========================
-- CREATE TABLES (With Updates)
-- ========================

-- PESSOA
CREATE TABLE pessoa (
    id SERIAL PRIMARY KEY,                          -- NEW: The non-sensitive
    ↪  primary key.
    telefone_celular VARCHAR UNIQUE NOT NULL,    -- CHANGED: Now a unique
    ↪  attribute, not the PK.
    cpf VARCHAR UNIQUE,
    email VARCHAR UNIQUE,
    nome VARCHAR NOT NULL,
    data_nascimento DATE,
    tipo_usuario tipo_usuario_enum NOT NULL
);

-- MÉDICO
CREATE TABLE medico (
    id_pessoa INTEGER PRIMARY KEY REFERENCES pessoa(id) ON DELETE CASCADE,
    ↪  -- CHANGED: Now links to pessoa.id.
```

```
    crm VARCHAR NOT NULL,
    especialidade VARCHAR NOT NULL
);

-- PACIENTE
CREATE TABLE paciente (
    id_pessoa INTEGER PRIMARY KEY REFERENCES pessoa(id) ON DELETE CASCADE,
    ↪  -- CHANGED: Now links to pessoa.id.
    anemia_falciforme BOOLEAN DEFAULT FALSE,
    hepatites BOOLEAN DEFAULT FALSE,
    aids BOOLEAN DEFAULT FALSE,
    depressao BOOLEAN DEFAULT FALSE,
    artrite BOOLEAN DEFAULT FALSE,
    asma BOOLEAN DEFAULT FALSE,
    hipertensao BOOLEAN DEFAULT FALSE,
    diabetes BOOLEAN DEFAULT FALSE,
    doencas_oncologicas BOOLEAN DEFAULT FALSE,
    doencas_cardiacas BOOLEAN DEFAULT FALSE
);

-- ADMIN (No Changes)
CREATE TABLE admin (
    id_admin VARCHAR PRIMARY KEY,
    nome VARCHAR NOT NULL,
    senha VARCHAR NOT NULL
);

-- CHAVE DE CADASTRO DE MÉDICO (No Changes)
CREATE TABLE chavemedico (
    id_chavemedico SERIAL PRIMARY KEY,
    valor VARCHAR UNIQUE NOT NULL,
    usada BOOLEAN DEFAULT FALSE
);

-- HORÁRIO DISPONÍVEL
CREATE TABLE horariodisponivel (
    id_medico INTEGER NOT NULL REFERENCES medico(id_pessoa) ON DELETE
    ↪  CASCADE, -- CHANGED: References the new key.
    data_hora TIMESTAMP NOT NULL,
    PRIMARY KEY (id_medico, data_hora) -- CHANGED: The composite key now
    ↪  uses the new ID.
);

-- CONSULTA
CREATE TABLE consulta (
    id_consulta SERIAL PRIMARY KEY,
    id_medico INTEGER NOT NULL REFERENCES medico(id_pessoa),     --
    ↪  CHANGED: Foreign key uses the new ID.
    id_paciente INTEGER NOT NULL REFERENCES paciente(id_pessoa), --
    ↪  CHANGED: Foreign key uses the new ID.
    data_hora TIMESTAMP NOT NULL,
```

```
    status status_consulta_enum NOT NULL
);

-- SESSÕES WEB
CREATE TABLE websession (
    id SERIAL PRIMARY KEY,
    id_pessoa INTEGER NOT NULL REFERENCES pessoa(id) ON DELETE CASCADE, --
    ↪  CHANGED: Foreign key uses the new ID.
    login_token VARCHAR(255) NOT NULL,
    expires_at TIMESTAMP NOT NULL
);
```

C Fluxos de Uso do Sistema

Este apêndice documenta e ilustra, passo a passo, os principais fluxos de interação do usuário com o sistema, desde o registro inicial até o agendamento de consultas.

C.1 Fluxo de Registro de Paciente

O processo de registro foi projetado para ser iniciado a partir da plataforma principal, guiando o usuário de forma intuitiva até a conclusão do seu cadastro.

1. O fluxo inicia na página principal do site `medcoqueiral.com.br`.

Figura 8: Página inicial do sistema.

2. Ao clicar em "Login", o usuário é redirecionado para o chatbot no WhatsApp para iniciar uma interação segura.

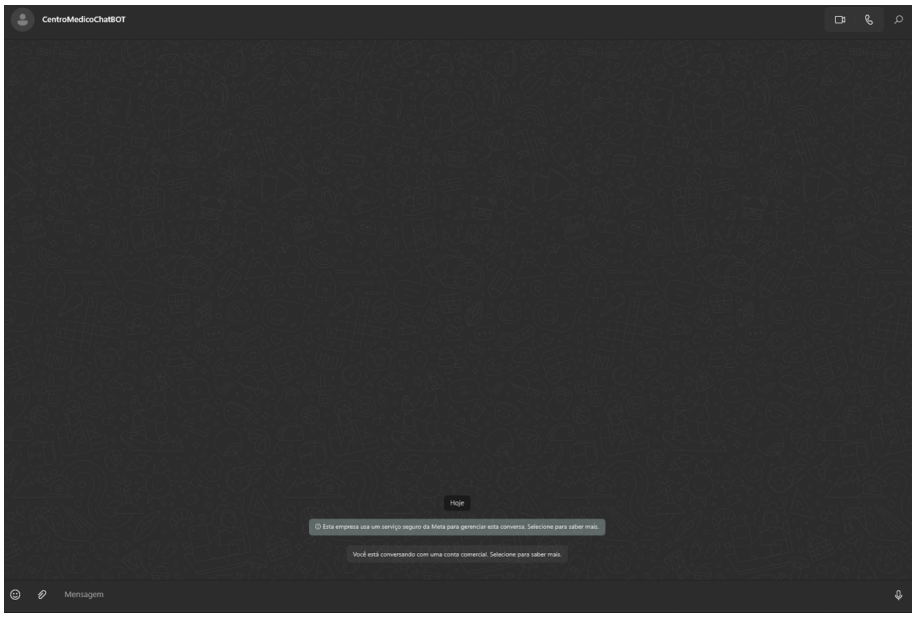

Figura 9: Primeira interação com o chatbot.

3. O usuário envia uma mensagem inicial (e.g., "oi") para ativar o menu principal do chatbot.

Figura 10: Menu de opções do chatbot.

4. O usuário escreve "1"para "Registrar/Login". Como o número ainda não está cadastrado, o sistema fornece um link único para a página de registro.

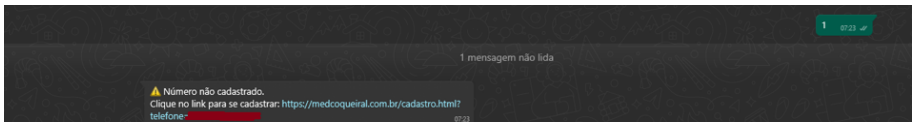

Figura 11: URL de registro fornecida pelo chatbot.

5. O usuário acessa o link, preenche seus dados no formulário e clica em "Cadastrar".

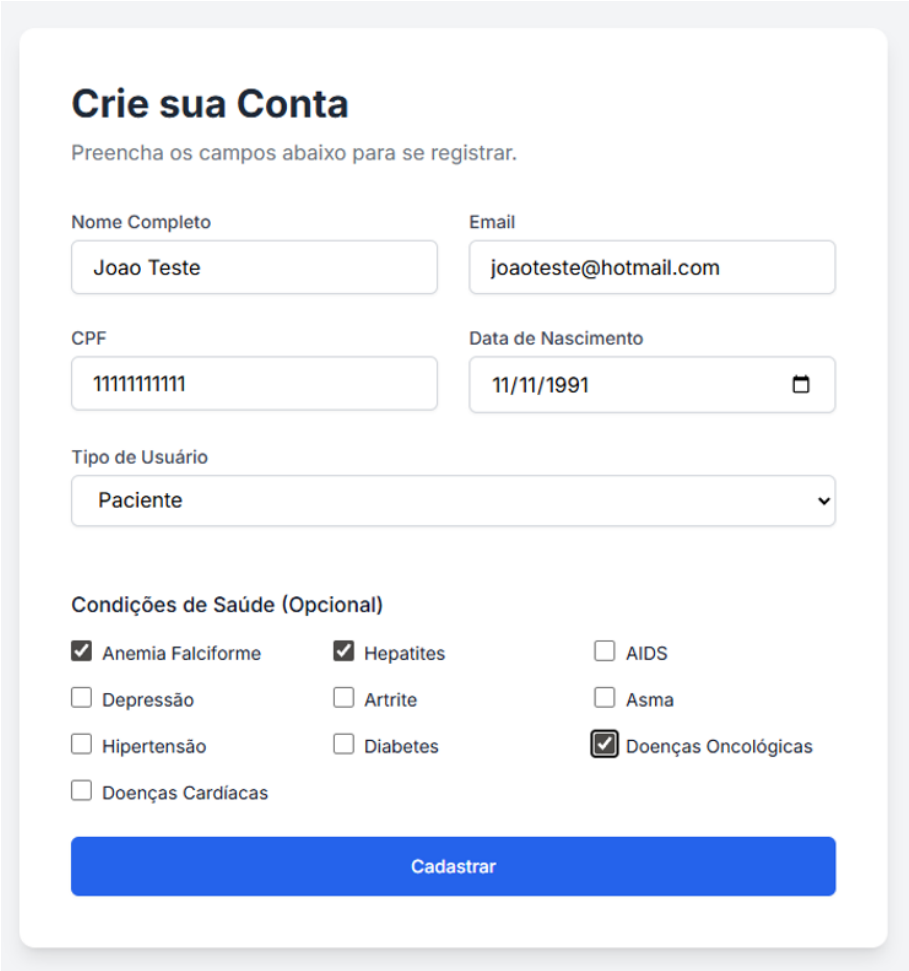

Figura 12: Página de cadastro do sistema.

6. Após a submissão bem-sucedida, o usuário é direcionado para uma página de confirmação.

Figura 13: Confirmação de registro concluído.

C.2 Fluxo de Login de Paciente

Com o cadastro concluído, o usuário retorna ao chatbot para realizar o login e acessar seu painel.

1. O usuário retorna ao WhatsApp e digita "menu"para reativar as opções.

Figura 14: Retorno ao menu do chatbot para login.

2. Ele novamente seleciona a opção "1". Desta vez, o sistema reconhece o número como cadastrado e gera um link de acesso autenticado e de uso único.

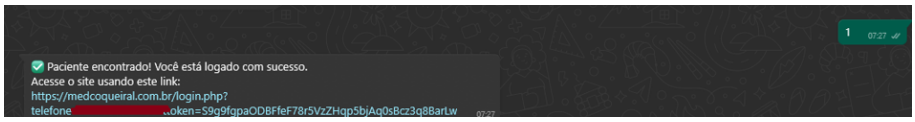

Figura 15: URL de acesso ao painel fornecida para o usuário.

3. Ao acessar o link, o usuário é direcionado diretamente para o seu painel de paciente, completando o ciclo de login.

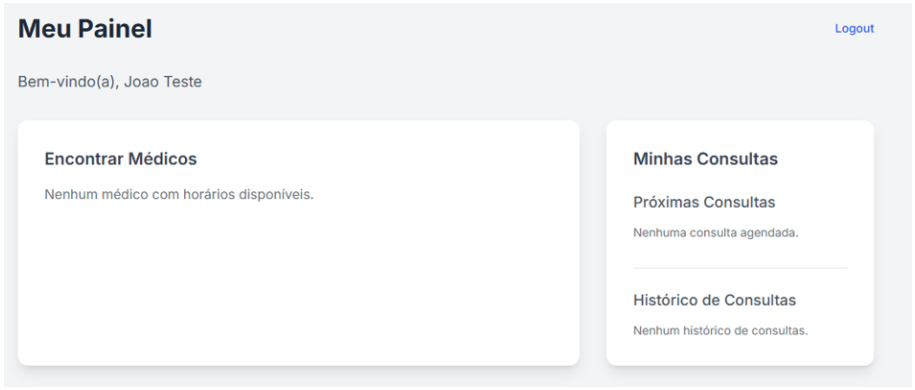

Figura 16: Painel do Paciente após login bem-sucedido.

C.3 Fluxo de Criação de Horários (Médico)

O fluxo de registro e login para o médico é idêntico ao do paciente. A diferenciação ocorre no painel, que oferece funcionalidades específicas para o profissional.

1. O médico realiza o login através do chatbot, recebendo seu link de acesso exclusivo.

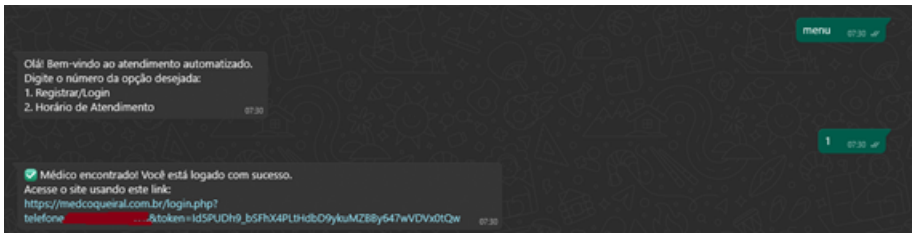

Figura 17: Médico recebendo o link de acesso ao painel.

2. Ao acessar o link, ele é direcionado ao Painel do Médico.

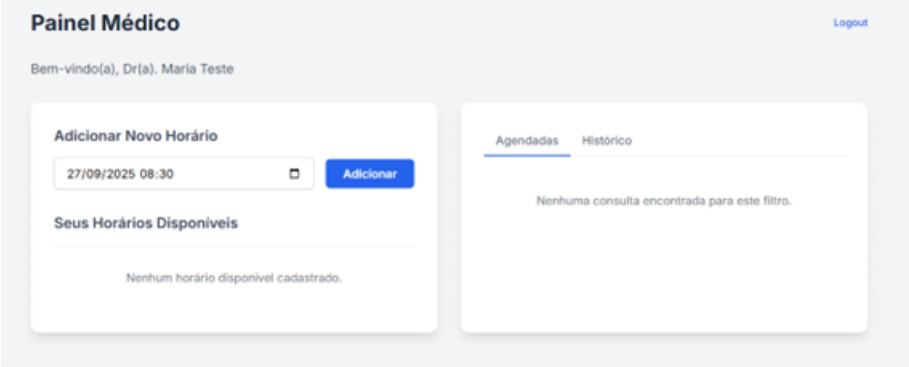

Figura 18: Visão geral do Painel do Médico.

3. O médico utiliza a interface para selecionar uma data e um horário em que deseja disponibilizar para consultas.

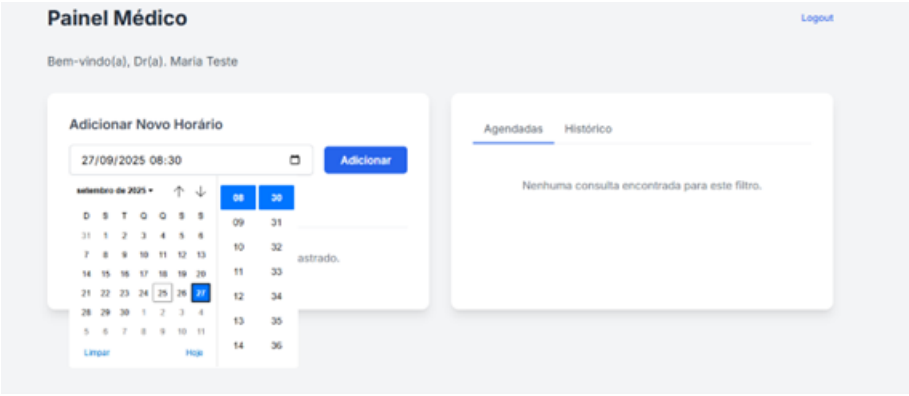

Figura 19: Interface para adição de novos horários disponíveis.

4. Após clicar em "Adicionar", o novo horário é listado em seu painel e se torna imediatamente visível para os pacientes que buscam agendamento.

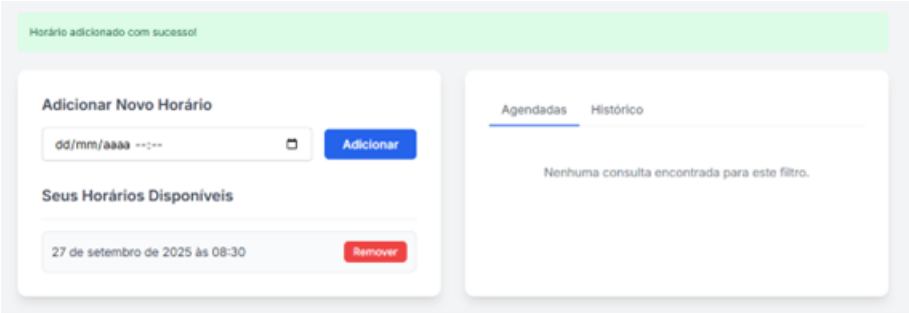

Figura 20: Painel do médico exibindo o horário recém-adicionado.

C.4 Fluxo de Agendamento de Consulta (Paciente)

Este fluxo descreve como um paciente, já autenticado no sistema, realiza o agendamento de uma consulta.

1. No painel do paciente, são exibidos os médicos e seus respectivos horários disponíveis.

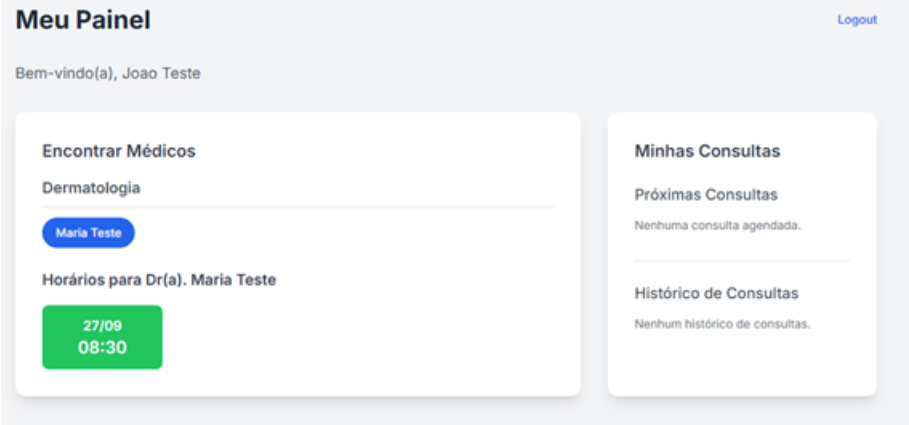

Figura 21: Painel do paciente com horários disponíveis para agendamento.

2. O paciente escolhe um horário desejado (no exemplo, 27/09 às 08:30) e clica para agendar. A ação o redireciona para a página de pagamento seguro, processada pela API da Stripe.

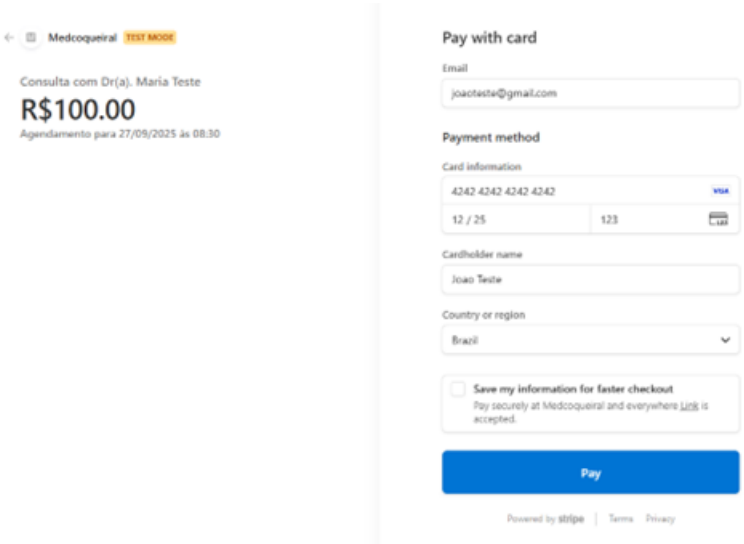

Figura 22: Página de checkout seguro da Stripe.

3. Após a confirmação do pagamento, o sistema processa o agendamento. O horário selecionado é removido da lista de disponibilidades e a consulta é adicionada à seção "Minhas Consultas"do painel do paciente.

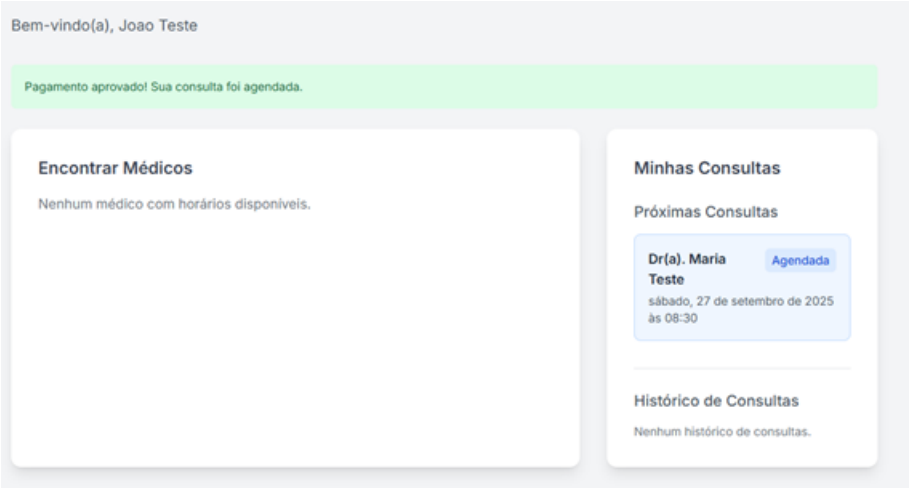

Figura 23: Painel do paciente exibindo a consulta agendada com sucesso.