

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Gabriel Duro da Silva

**Desenvolvimento de uma interface web para
parametrização de modelos de otimização de
dietas**

Uberlândia, Brasil

2025

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Gabriel Duro da Silva

**Desenvolvimento de uma interface web para
parametrização de modelos de otimização de dietas**

Trabalho de conclusão de curso apresentado
à Faculdade de Computação da Universidade
Federal de Uberlândia, como parte dos requi-
sitos exigidos para a obtenção título de Ba-
charel em Ciência da Computação.

Orientador: Prof. Dr. Paulo Henrique Ribeiro Gabriel

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2025

**UNIVERSIDADE FEDERAL DE UBERLÂNDIA**

Faculdade de Computação

Av. João Naves de Ávila, nº 2121, Bloco 1A - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902

Telefone: (34) 3239-4144 - <http://www.portal.facom.ufu.br/> facom@ufu.br**ATA DE DEFESA - GRADUAÇÃO**

Curso de Graduação em:	Ciência da Computação: Bacharelado				
Defesa de:	GBC082: Projeto de Graduação 2				
Data:	09/05/2025	Hora de início:	14:00	Hora de encerramento:	14:55
Matrícula do Discente:	12111BCC047				
Nome do Discente:	Gabriel Duro da Silva				
Título do Trabalho:	Desenvolvimento de uma interface web para parametrização de modelos de otimização de dietas				
A carga horária curricular foi cumprida integralmente?		(X) Sim () Não			

Reuniu-se remotamente através da plataforma Microsoft Teams, a Banca Examinadora, designada pelo Colegiado do Curso de Graduação em Ciência da Computação, assim composta: Professores: Dr. Luís Fernando Faina - FACOM/UFU; Dra. Maria Adriana Vidigal de Lima - FACOM/UFU; e Dr. Paulo Henrique Ribeiro Gabriel - FACOM/UFU, orientador(a) do(a) candidato(a).

Iniciando os trabalhos, o presidente da mesa, Dr. Paulo Henrique Ribeiro Gabriel, apresentou a Comissão Examinadora e o candidato, agradeceu a presença do público, e concedeu ao discente a palavra, para a exposição do seu trabalho. A duração da apresentação do discente e o tempo de arguição e resposta foram conforme as normas do curso.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir o candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o candidato:

(X) Aprovado Nota 95

OU

() Aprovado(a) sem nota.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Documento assinado eletronicamente por **Paulo Henrique Ribeiro Gabriel, Professor(a) do**



Magistério Superior, em 09/05/2025, às 14:53, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



Documento assinado eletronicamente por **Maria Adriana Vidigal de Lima, Professor(a) do Magistério Superior**, em 09/05/2025, às 14:53, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



Documento assinado eletronicamente por **Luís Fernando Faina, Professor(a) do Magistério Superior**, em 09/05/2025, às 14:53, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **6323665** e o código CRC **3F798E61**.

Referência: Processo nº 23117.030614/2025-71

SEI nº 6323665

Agradecimentos

Primeiramente, gostaria de agradecer minha esposa, pelo apoio e paciência interminável durante toda a minha trajetória durante o curso e, em especial, durante o desenvolvimento do projeto de graduação. Com certeza eu não seria capaz de chegar até onde estou sem este apoio.

Estendo meus agradecimentos à minha família, que me criou e me preparou para as responsabilidades da vida.

Aos meus amigos, antigos e que fiz ao longo do curso, pelas diversas vezes que me ajudaram quando precisei de apoio.

Ao meu professor orientador Prof. Dr. Paulo Henrique Ribeiro Gabriel, pela paciência e disponibilidade ao longo de todo o processo, esclarecendo e contribuindo ao trabalho.

Agradeço também os meus professores, que contribuíram para minha formação ao longo dos anos.

Resumo

Este trabalho aborda o desenvolvimento de uma aplicação web progressiva (*Progressive Web App* - *PWA*) acessível e simples para planejamento de dietas personalizáveis para diferentes situações de indivíduos integrando um modelo de otimização multiobjetivo. Assim, espera-se que a ferramenta auxilie profissionais de saúde, em especial na área de nutrição e relacionados, na criação de planos alimentares que considerem as necessidades específicas de cada indivíduo. A aplicação desenvolvida permite o cadastro multiusuário e multipaciente por usuário, onde é possível parametrizar metas e refeições de cada paciente por períodos, realizando o processo de sugestão de dietas para um plano alimentar automaticamente através do modelo de otimização integrado. Após o desenvolvimento, foi realizado um estudo de caso utilizando um caso já testado na definição do modelo de otimização e, a partir dos testes realizados, observou-se que o sistema foi capaz de produzir resultados legíveis e utilizáveis por profissionais.

Palavras-chave: PWA, dietas, otimização, desenvolvimento de aplicação web.

Abstract

This work addresses the development of a simple and accessible progressive web application (PWA) that integrates a multiobjective optimization model into diet planning that is customizable for different types of individuals and situations. We expect that the tool will assist healthcare professionals, especially in nutrition and related fields, in creating dietary plans that consider the specific needs of each individual. The developed application allows multi-user and multi-patient registration per user, where it is possible to set goals and meals for each patient over periods, automatically suggesting diets through the integrated optimization model. After development, a case study was conducted using a previously tested case when the optimization model was defined. Based on the performed tests, the system may produce readable and usable results for professionals.

Keywords: PWA, diets, optimization, web application development.

Lista de ilustrações

Figura 1 – Exemplo de tarefa de implementação de interface de usuário	18
Figura 2 – Backlog de implementação de funcionalidades	19
Figura 3 – Estrutura do <i>Front-End</i>	21
Figura 4 – Estrutura do <i>Back-End</i>	22
Figura 5 – Diagrama de Entidades Relacionais	26
Figura 6 – Opções básicas do Anvil	27
Figura 7 – Sistema de Controle de Versão Imbutido	28
Figura 8 – Utilização de <i>Layouts</i> e conteúdos dinâmicos	30
Figura 9 – Layout utilizado ao acessar o site sem um usuário em sessão	31
Figura 10 – Layout utilizado ao acessar o site com um usuário em sessão	31
Figura 11 – Visualização da aplicação em dispositivos de tela pequena	32
Figura 12 – Esqueleto do <i>CRUD</i> de Paciente	37
Figura 13 – Interface de agendamento de geração de dietas	38
Figura 14 – Página de geração de relatórios	40
Figura 15 – Tela de Sessão de Usuário	45
Figura 16 – Campos de Cadastro de Usuário	46
Figura 17 – Dados Básicos do Paciente	47
Figura 18 – Edição de Plano Alimentar de Paciente	47
Figura 19 – Nova Refeição	48
Figura 20 – Refeições cadastradas	49
Figura 21 – Metas diárias	50
Figura 22 – Controle de Progresso de Geração de Dieta	51
Figura 23 – Dietas Geradas	52

Lista de tabelas

Tabela 1 – Comparação de Resultados	51
---	----

Lista de abreviaturas e siglas

AJAX	<i>Asynchronous JavaScript and XML</i>
API	<i>Application Programming Interface</i>
CRUD	<i>Create, Read, Update and Delete</i>
CSS	<i>Cascading Style Sheets</i>
HTML	<i>HyperText Markup Language</i>
MD3	<i>Material Design 3</i>
MVP	<i>Minimal Viable Product</i>
PWA	<i>Progressive Web App</i>
REST	<i>Representational State Transfer</i>
XML	<i>Extensible Markup Language</i>

Sumário

1	INTRODUÇÃO	12
2	REFERENCIAL TEÓRICO	13
2.1	Otimização e Programação Linear	13
2.2	Otimização multiobjetivo aplicada a dietas	13
2.3	<i>Minimal Viable Product (MVP)</i>	14
2.4	<i>SCRUM</i> e Métodos Ágeis	15
2.5	Anvil Works	15
2.6	<i>Progressive Web Apps</i> e desenvolvimento <i>web</i>	15
3	DESENVOLVIMENTO	17
3.1	Visão geral	17
3.2	Planejamento de funcionalidades	18
3.3	Arquitetura e estruturação da aplicação	19
3.3.1	Arquitetura do sistema	19
3.3.2	Linguagem de desenvolvimento	20
3.3.3	Estruturação de pastas	20
3.4	Enumerações e entidades	22
3.4.1	<i>BaseEnum</i> e <i>Entity</i>	22
3.4.2	Enumerações	24
3.4.3	Entidades	24
3.5	Banco de dados e tabelas	24
3.5.1	Sistema e estrutura de banco de dados	24
3.5.2	Tabelas	25
3.6	Configuração do projeto	26
3.6.1	Criação de projeto no Anvil	26
3.6.2	Controle de versão e repositório	27
3.6.3	Instalação de pacotes	28
3.6.4	Configuração de domínio e servidor	28
3.6.5	Configuração de dependências	28
3.7	Criação de <i>layouts</i> e rotas	30
3.7.1	<i>Layouts</i>	30
3.7.2	Rotas	32
4	RESULTADOS	37
4.1	Funcionalidades da ferramenta	37

4.1.1	Cadastro de paciente	37
4.1.2	Geração de dietas	39
4.1.3	Emissão de dietas geradas	40
4.1.4	Comunicação com servidor	41
4.1.5	Geração parametrizável de dietas	42
4.2	Estudo de caso	44
4.2.1	Cadastro de usuário	44
4.2.2	Cadastro de paciente	45
4.2.3	Geração de dietas	50
4.2.4	Comparação de resultados	51
5	CONCLUSÃO	53
	REFERÊNCIAS	54
	APÊNDICES	55
	APÊNDICE A – CLASSE BASEENUM	56
	APÊNDICE B – CLASSE ENTITY	58

1 INTRODUÇÃO

A personalização de dietas é um fator determinante para o sucesso de intervenções nutricionais, seja no contexto clínico, esportivo ou de promoção de saúde em geral. Diversas condições clínicas — como diabetes, obesidade, hipertensão e dislipidemias — exigem atenção cuidadosa à composição da dieta, enquanto atletas e praticantes de atividade física podem ter metas específicas de ganho de massa, perda de gordura ou melhora de desempenho. Em todos esses casos, o planejamento nutricional adequado pode contribuir diretamente para a qualidade de vida e a prevenção de agravos.

Entretanto, a criação de dietas personalizadas é uma tarefa que exige tempo, conhecimento técnico e a análise de múltiplos critérios nutricionais e práticos, como valor energético, distribuição de macronutrientes, custo dos alimentos, preferências alimentares e variedade. Nesse cenário, o uso de algoritmos de otimização surge como uma ferramenta poderosa para auxiliar profissionais da saúde na elaboração de planos alimentares mais eficientes, equilibrados e adaptados às necessidades individuais de cada paciente.

O presente trabalho tem como objetivo o desenvolvimento de uma aplicação web que viabilize a utilização prática de um algoritmo de otimização multiobjetivo proposto por Soares (2023), integrando esse modelo a uma interface gráfica que permita sua parametrização por parte do profissional de saúde. A aplicação possibilita que o profissional defina metas nutricionais específicas, organize refeições, determine penalidades por repetição alimentar e, a partir desses parâmetros, gere dietas personalizadas de forma automatizada.

A aplicação foi construída utilizando a plataforma Anvil, que oferece um ambiente completo para o desenvolvimento de aplicações web progressivas (*Progressive Web App - PWA*) em *Python*, incluindo ferramentas visuais para construção da interface, modelagem do banco de dados, publicação e integração com bibliotecas externas. Durante a implementação do projeto foram empregadas metodologias ágeis, baseadas no SCRUM, para melhor organização e planejamento das implementações e entregas.

Como estudo de caso, a ferramenta foi aplicada no mesmo plano utilizado no modelo de otimização usado por Soares (2023) de forma a promover comparações dos resultados.

2 REFERENCIAL TEÓRICO

Neste capítulo são apresentados os principais conceitos e tecnologias fundamentais para a compreensão e o desenvolvimento deste trabalho. Além disto, é apresentado o trabalho correlato que inspira a idealização deste projeto.

2.1 Otimização e Programação Linear

Otimização é um ramo da matemática que visa encontrar o melhor resultado possível (resultado ótimo) em um determinado problema. Normalmente, a solução para um problema de otimização envolve maximizar (ou seja, buscar o maior valor), ou minimizar (ou seja, buscar o menor valor) de uma instância variável do problema. Frequentemente a busca de maximização de uma variável também envolve a minimização de outra variável distinta, por exemplo o problema de buscar **o maior lucro** com venda de certos produtos manufaturados poderia envolver a busca do produto com **menor custo** de produção.

O modelo matemático desses problemas costuma ser representado através de expressões matemáticas, fórmulas e variáveis que descrevam a essência do problema, uma função objetivo a ser otimizada e um conjunto de restrições que delimitam o espaço de soluções viáveis (MARINS, 2011) (BERTSEKAS; PANG, 1999).

Um método frequentemente utilizado para encontrar soluções deste tipo é a programação linear (LUENBERGER, 1969). Marins (2011) define a Programação Linear (PL) como um conjunto de técnicas e métodos científicos com o objetivo de maximizar ou minimizar uma função linear que representa o modelo matemático do problema, chamada função objetivo, ainda considerando as restrições do problema, representados por um sistema linear de igualdades e desigualdades.

2.2 Otimização multiobjetivo aplicada a dietas

Para este trabalho, o modelo desenvolvido por Soares (2023) possui maior importância, pois foi utilizado na otimização e geração das dietas. Em seu trabalho, o autor estudou o Diabetes Mellitus, ou simplesmente diabetes, e tinha como principal objetivo o desenvolvimento de um modelo matemático de otimização multiobjetivo para o planejamento automático de dietas para indivíduos portadores, considerando as restrições nutricionais para tal.

O foco e objetivo deste modelo incluem a minimização da ingestão de carboidratos, nutriente que mais afeta os níveis de glicose em diabéticos, enquanto reduzia o consumo

de lipídeos e colesterol, evitando compensações energéticas prejudiciais. Também foi levado em consideração a diversidade alimentar, tentando evitar a repetição de alimentos e aumentando a adesão do paciente à dieta planejada.

O primeiro modelo proposto por Soares (2023) teve como objetivo determinar um plano alimentar para um único dia, focando em um objetivo único de minimizar a quantidade de carboidratos na dieta mantendo valores nutricionais adequados. Além da limitação de se concentrar exclusivamente na minimização dos carboidratos, este modelo também não leva em consideração a frequência com que um alimento específico pode aparecer em um menu diário, o que pode resultar em sugestões que repetiam o mesmo alimento, algo que pode não ser aconselhável. Adicionalmente, o modelo é orientado para o cardápio de um único dia e não fazia distinções entre diferentes refeições ao longo do dia.

O segundo modelo apresentou novas funções objetivos, incorporando objetivos relacionados à lipídios e colesterol junto com carboidratos, além de considerar o custo junto à minimização dos macronutrientes. O modelo também foi aprimorado com a introdução de *penalidades* para cada alimento, que visavam assegurar a diversidade alimentar e evitar a seleção frequente dos mesmos alimentos.

No terceiro e último modelo, Soares (2023) utilizou categorias de alimentos para organizar a distribuição equitativa de nutrientes e também uma organização de dieta em seis refeições diárias, que utilizavam conjuntos das categorias de alimentos. Ainda assim, mesmo com os ajustes, o modelo final ainda apresenta algumas limitações. Uma das limitações era o fato de que o modelo não levava em conta necessidades nutricionais individuais e preferências alimentares. Outra limitação mais abrangente do projeto é que o modelo foi preparado considerando um perfil fictício, hipotético, de portador de diabetes, podendo ser possível expandir sua aplicabilidade a uma diversidade maior de contextos dietéticos e populações com alterações nos parâmetros.

O presente trabalho, portanto, adapta esse modelo teórico para uso prático em ambiente web, oferecendo uma interface de parametrização acessível para profissionais da saúde.

2.3 *Minimal Viable Product (MVP)*

O conceito de Produto Mínimo Viável (*Minimal Viable Product - MVP*) se refere a uma versão simples e inicial de um produto que possui apenas as funcionalidades essenciais necessárias para ser lançada. Seu objetivo é, com o mínimo esforço e investimento possível, validar que o produto tem valor suficiente para ser lançado, demonstrar seus benefícios para reter usuários iniciais, e coletar resposta do cliente que guiará o desenvolvimento posterior. (RIES, 2011)

2.4 SCRUM e Métodos Ágeis

Os Métodos Ágeis surgiram como alternativa aos modelos tradicionais de planejamento, como o modelo em cascata, que priorizava etapas sequenciais, nas quais era necessário concluir uma fase antes de avançar para a seguinte. Nos métodos ágeis a prioridade está em entregas contínuas, em ciclos curtos com metas estabelecidas, iterando estas metas e adaptando os próximos períodos com base nas respostas dos períodos anteriores.

Dentre os métodos ágeis mais utilizados, destaca-se o SCRUM, que organiza o trabalho em ciclos curtos denominados *sprints*, normalmente com duração de duas a quatro semanas. Cada *sprint* busca entregar um incremento funcional do produto ([SCHWABER; SUTHERLAND, 2020](#)).

2.5 Anvil Works

A plataforma Anvil Works é uma ferramenta de desenvolvimento que oferece a habilidade de criar e disponibilizar rapidamente aplicações *web* usando principalmente *Python*.

A plataforma disponibiliza ferramentas completas e visuais de desenvolvimento, incluindo um editor de código integrado para *Python*, *JavaScript*, *HTML* e *CSS*, bem como o *designer* de interfaces visuais com recursos de *drag-and-drop* (arrastar e soltar) e uma biblioteca de componentes nativos pré-construídos. Também oferece um sistema de banco de dados próprio baseado em *PostgreSQL*, acessível tanto por meio de código quanto através de um editor visual.

Além dessas funcionalidades, também é oferecido integração nativa com serviços externos como Google Suite, publicação, hospedagem e versionamento das aplicações e outras ferramentas diversas ([Anvil Works, 2024](#)).

2.6 Progressive Web Apps e desenvolvimento web

A programação *web* refere-se ao processo de criação e desenvolvimento de aplicações e sites acessíveis através de navegadores, utilizando linguagens como *HyperText Markup Language (HTML)*, *Cascading Style Sheets (CSS)* e *JavaScript*. Inicialmente, a web era predominantemente estática, com *HTML* simples sendo a principal linguagem de marcação, mas com a introdução de linguagens de *script* como *JavaScript* nos anos 90, a interatividade começou a emergir. O surgimento de *frameworks* como Angular, React e Vue.js, bem como a introdução de tecnologias como o Asynchronous JavaScript and

XML¹ (AJAX) e RESTful², permitiram que desenvolvedores criassem experiências mais ricas e responsivas (BROWN, 2019). Recentemente, a ascensão de tecnologias como *Progressive Web Apps* (PWAs) e a integração de *Application Programming Interfaces* (APIs) em tempo real têm promovido experiências mais ricas e rápidas para os usuários, refletindo uma evolução contínua na maneira como interagimos com a web (FLANAGAN, 2020; W3C, 2025).

As PWAs são aplicações web que utilizam tecnologias modernas para proporcionar uma experiência similar à de aplicativos nativos em dispositivos móveis e *desktop*, sem perder a simplicidade e acessibilidade global de um site. Uma das principais características das PWAs é sua capacidade de funcionar *offline* ou em redes de baixa qualidade. Outra característica chave é a responsividade, que garante que a PWA se adapte a diferentes tamanhos de telas e dispositivos, proporcionando uma experiência adaptada ao usuário, esteja ele utilizando em um computador pela *browser* ou em um dispositivo móvel.

Um dos principais benefícios das PWAs são a redução de custos de desenvolvimento, já que uma única base de código pode ser usada em diferentes plataformas e sistemas operacionais, o que impactam diretamente na retenção de usuários (WEB.DEV, 2015).

¹ XML neste caso é a sigla para *Extensible Markup Language*.

² RESTful significa “Que aplica tecnologias REST”, onde REST é a sigla para *Representational State Transfer*.

3 DESENVOLVIMENTO

O desenvolvimento deste trabalho teve como objetivo principal a criação de uma aplicação *web* que implementa o algoritmo de otimização multiobjetivo proposto por Soares (2023), permitindo sua parametrização e utilização prática por profissionais da área da saúde. A plataforma desenvolvida disponibiliza ao usuário a possibilidade de definir metas nutricionais personalizadas e gerar automaticamente dietas otimizadas de acordo com os critérios estabelecidos.

Neste capítulo será detalhado o desenvolvimento da aplicação de parametrização de dietas e da utilização e armazenamento dos dados do cliente. O objetivo é fornecer uma visão do processo, das arquiteturas e linguagens empregadas ao longo deste trabalho.

3.1 Visão geral

A execução do trabalho foi feita seguindo as seguintes etapas:

- **Planejamento de funcionalidades:** planejamento inicial de quais funcionalidades são necessárias e suas prioridades;
- **Arquitetura e estruturação da aplicação:** escolha de plataforma, bem como estruturação de pastas e arquivos e organização dos componentes da aplicação;
- **Enumerações e entidades:** definição das enumerações bem como as entidades relacionais utilizadas na aplicação;
- **Banco de dados e tabelas:** criação das tabelas no banco de dados e definição de tipos e relacionamentos;
- **Configuração do projeto:** instalação, configuração e adaptação de bibliotecas e *frameworks* próprios e de terceiros necessários para execução da aplicação;
- **Criação de *layouts* e rotas:** desenvolvimento do *design* básico da aplicação bem como definição dos caminhos acessíveis;

O código e implementação específica do projeto está disponível no repositório <<https://github.com/GDuroS/ufu-tec>> para consulta.

3.2 Planejamento de funcionalidades

O desenvolvimento da aplicação teve um MVP, organizado com o levantamento de requisitos funcionais, relacionados às interfaces de cadastro, parametrização de metas nutricionais e geração de dietas, e também requisitos não funcionais, como responsividade da interface, compatibilidade com dispositivos móveis e persistência de dados. O levantamento desses requisitos orientou a modelagem das entidades, o fluxo de telas e a definição da arquitetura da aplicação.

O planejamento foi realizado através da plataforma de Gerenciamento de Projetos *ClickUp*. Foram elencadas todas as tarefas necessárias para realização do projeto, incluindo modelagem, configurações e funcionalidades que seriam necessárias para o projeto, criando o *backlog* da aplicação. Em cada tarefa a se realizar foram definidas prioridades em relação ao projeto e tempo estimado para conclusão. Nas tarefas relacionadas à funcionalidades, foram descritos os requisitos de sistema, seguidos de uma sugestão de implementação visual de forma a direcionar o desenvolvimento posterior. A Figura 1 ilustra uma tarefa planejada para interface de usuário.

Figura 1 – Exemplo de tarefa de implementação de interface de usuário

The screenshot shows a task card in ClickUp. At the top, there's a header with 'Task' and a unique ID '86b2au6xh', along with an 'Ask AI' button. The main title of the task is 'Parametrização de Refeições do Paciente'. Below the title, there's a bar with AI prompts: 'Ask Brain to create a summary · generate subtasks · find similar tasks · or ask about this task'. The task details are organized into two columns. The left column includes: Status (FECHADO / PRODUÇÃO), Dates (6 Mar → 7 Mar, 18:00), Time Estimate (1d), and Tags (Empty). The right column includes: Assignees (one user icon), Priority (High), Track Time (34h), and Relationships (1 Task). The main body of the task contains a description in Portuguese, a bulleted list of requirements, and three paragraphs of detailed implementation instructions.

Task ID: 86b2au6xh | Ask AI

Parametrização de Refeições do Paciente

Ask Brain to create a summary · generate subtasks · find similar tasks · or ask about this task

Status	FECHADO / PRODUÇÃO	Assignees	[User Icon]
Dates	6 Mar → 7 Mar, 18:00	Priority	High
Time Estimate	1d	Track Time	34h
Tags	Empty	Relationships	1 Task

Para cada Paciente, o profissional poderá criar uma lista de refeições necessárias para o paciente, informando dados como:

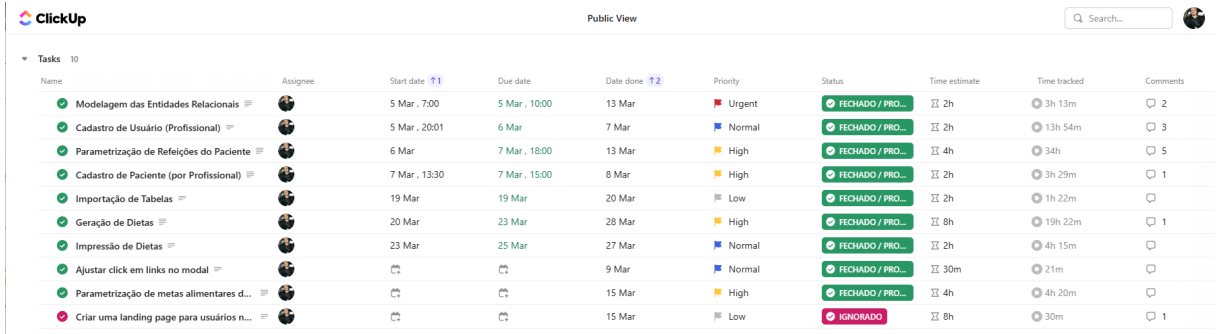
- Nome da Refeição (ex.: Almoço, Janta, Lanche, Café da Manhã);
- Horário recomendado;
- Quantidades necessárias por classificação.

A listagem poderá ser exibida na forma de uma *grid* de valores editável, adicionável e móvel.

No modo de visualização, cada linha da tabela exibe o nome da refeição e horário recomendado nas colunas apropriadas e uma lista de ícones e números na coluna de Quantidades, representando as quantidades por classificação.

No modo de edição, exibir um painel com campo de texto para nome da refeição e campo de seleção de Horário para horário recomendado. Exibir também uma *grid* contendo linhas fixas para cada uma das classificações de alimentos e apenas uma coluna, editável, para as quantidades. No final do painel, exibir o botão de aplicar ou cancelar.

Figura 2 – Backlog de implementação de funcionalidades



Name	Assignee	Start date	Due date	Date done	Priority	Status	Time estimate	Time tracked	Comments
Modelagem das Entidades Relacionais		5 Mar, 7:00	5 Mar, 10:00	13 Mar	Urgent	FECHADO / PRO...	2h	3h 13m	2
Cadastro de Usuário (Profissional)		5 Mar, 20:01	6 Mar	7 Mar	Normal	FECHADO / PRO...	2h	13h 54m	3
Parametrização de Refeições do Paciente		6 Mar	7 Mar, 18:00	13 Mar	High	FECHADO / PRO...	4h	34h	5
Cadastro de Paciente (por Profissional)		7 Mar, 13:30	7 Mar, 15:00	8 Mar	High	FECHADO / PRO...	2h	3h 29m	1
Importação de Tabelas		19 Mar	19 Mar	20 Mar	Low	FECHADO / PRO...	2h	1h 22m	
Geração de Dietas		20 Mar	23 Mar	28 Mar	High	FECHADO / PRO...	8h	19h 22m	1
Impressão de Dietas		23 Mar	25 Mar	27 Mar	Normal	FECHADO / PRO...	2h	4h 15m	
Ajustar click em links no modal				9 Mar	Normal	FECHADO / PRO...	30m	21m	
Parametrização de metas alimentares d...				15 Mar	High	FECHADO / PRO...	4h	4h 20m	
Criar uma landing page para usuários n...				15 Mar	Low	IGNORADO	8h	30m	1

Já com a listagem de todas as tarefas necessárias para o desenvolvimento da aplicação, foi necessário realizar o planejamento do cronograma de implementação. A metodologia escolhida para esta etapa foi o *SCRUM*, onde as tarefas seriam agrupadas por dependências entre si, importância, tempo e assunto em dois *sprints* de duas semanas, sendo o primeiro focado no desenvolvimento das interfaces de criação de usuário e cadastro de pacientes, enquanto o segundo foi focado na implementação da lógica de criação das dietas e exibição ao usuário. Foram definidas datas previstas de início e conclusão para cada tarefa, baseadas nas prioridades, dependências entre si e nas previsões tem tempo definidas para cada tarefa.

O *backlog* das funcionalidades, incluindo o que foi feito nos dois sprints está representado na Figura 2 e pode ser acessado através do link público <<https://sharing.clickup.com/3000536/1/h/2vj6r-6014/1a1646e1f787ddd>>.

3.3 Arquitetura e estruturação da aplicação

Nesta seção, será explicada a organização da arquitetura do sistema e a estruturação, bem como ferramentas, linguagens e bibliotecas utilizadas, proporcionando uma visão geral sobre sua criação e desenvolvimento.

3.3.1 Arquitetura do sistema

A criação do sistema se utiliza amplamente do Anvil Works, uma plataforma para construção full-stack de PWAs usando essencialmente *Python*, tanto no código servidor, quanto cliente. A plataforma oferece editor integrado com *designer*, distribuição imediata, controle de versão, hospedagem e banco de dados semelhante à Postgres. A ferramenta oferece planos gratuitos e pagos, mas o autor utilizou de um plano pago que oferece um melhor desempenho *server-side* além de remoção de *banner* da marca, possibilidade de domínios próprios além de outras funcionalidades.

3.3.2 Linguagem de desenvolvimento

O desenvolvimento foi realizado utilizando *Python* em praticamente sua totalidade, deste código do cliente, incluindo componentes visuais e lógica de eventos de componentes, até código de execução no servidor e comunicação com banco de dados.

Nas interfaces de usuário, foram utilizados componentes pré-prontos disponibilizados pela plataforma Anvil Works, alguns componentes de uma biblioteca pública criada para a plataforma, chamada Anvil Extras, assim como componentes próprios desenvolvidos pelo autor, alguns sendo de multiuso criados anteriormente para outras aplicações, outros sendo específicos da aplicação em questão. Nestes componentes próprios, eventualmente foram escritos alguns códigos em HTML, CSS e, muito eventualmente, JavaScript.

3.3.3 Estruturação de pastas

O código escrito para a aplicação fica naturalmente organizado em duas grandes seções: **Client Code** e **Server Code**. A Figura 3 apresenta a estruturação de pastas do sistema para códigos executados no lado cliente, enquanto a Figura 4 apresenta a estrutura de arquivos executados no servidor.

O código que é executado no cliente inclui a lista de funcionalidades do sistema (Changelog), um conjunto de variáveis globais, como usuário da sessão atual e lista de pacientes (Commons), os componentes visuais específicos da aplicação utilizados nas interfaces do sistema (Components), a lista de classes vinculadas a entidades relacionais do Banco de Dados (Entities), a lista de classes representativas de enumerações necessárias (Enums), as interfaces que serviam como base da navegação pública e privada da aplicação (Layouts), as interfaces específicas que seriam utilizadas dentro das interfaces base conforme o usuário navegasse (Pages), além da definição de rotas da aplicação (Routes) e das definições iniciais de inicialização do ambiente do sistema (Main). Maiores detalhes sobre esta estrutura serão tratados nas seções posteriores.

Já no servidor, todo o código anteriormente mencionado e definido como *Client Code* também estava disponível, juntamente com os arquivos de uso exclusivo do server, definidos como *Server Code*. Isso era necessário para que classes de modelo, como as que estavam em Entities e Enums estivessem disponíveis para uso do server. A estrutura do código exclusivo do server estava organizada em módulos responsáveis por definir a fachada de recebimento das requisições internas, realizadas pela própria aplicação rodando no cliente (Resource), um módulo que definia as classes de serviço que eram responsáveis por manipular a regra de negócio e persistência de dados (Service) e um módulo auxiliar cuja responsabilidade era tratar a navegação de usuários direto para uma parte específica da aplicação. Maiores detalhes sobre esta estrutura também serão tratados nas seções posteriores.

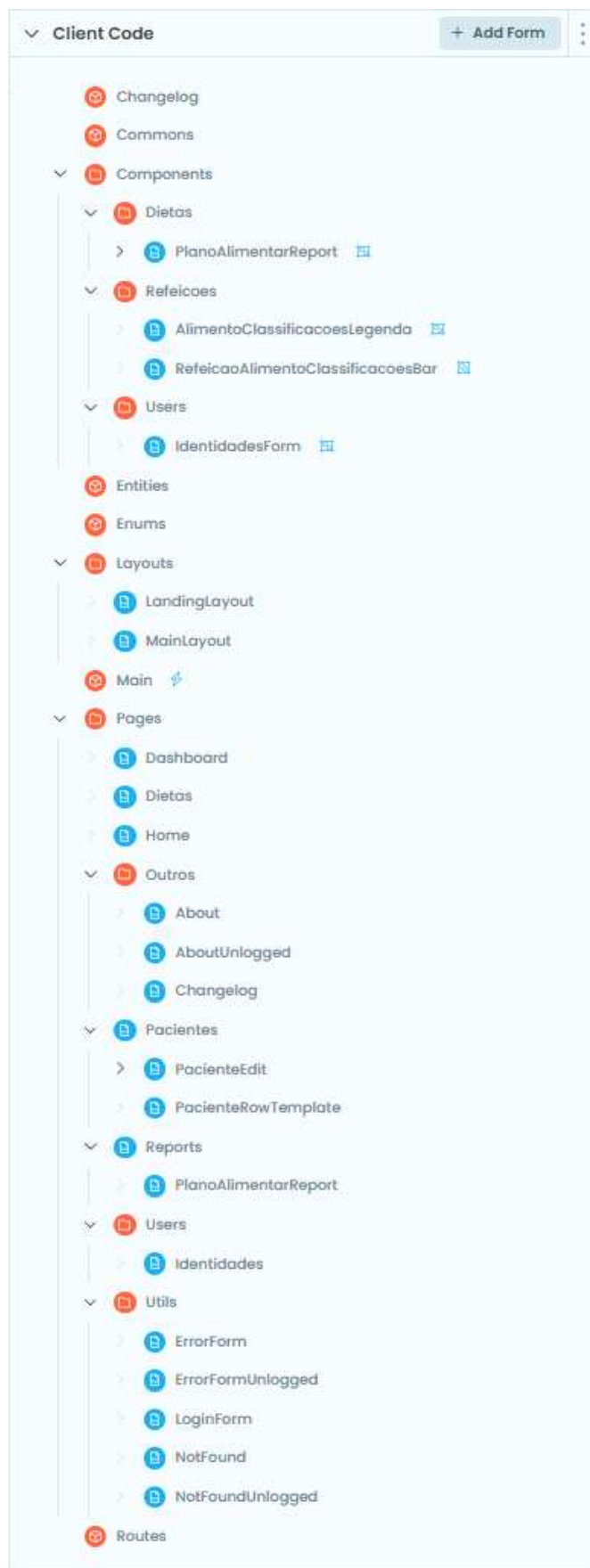
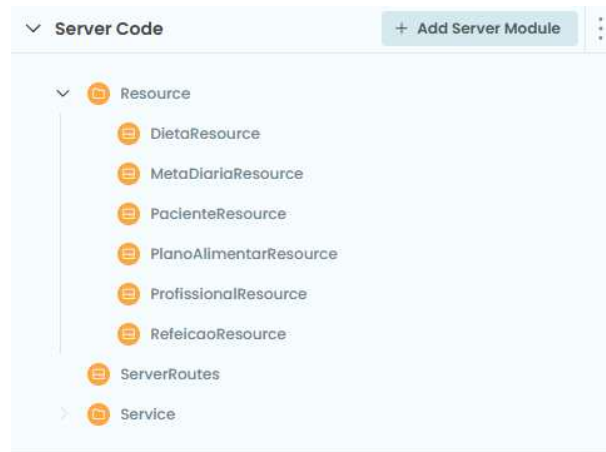
Figura 3 – Estruturação de arquivos *Client Code* (Fonte: o Autor)

Figura 4 – Estruturação de arquivos *Server Code* (Fonte: o Autor)

3.4 Enumerações e entidades

Nesta seção, serão listadas as classes criadas para representar o modelo de entidades relacionais e enumerações utilizadas ao longo da aplicação.

3.4.1 *BaseEnum* e *Entity*

Na definição do modelo, todas as classes que representavam Enums da aplicação eram classes que herdavam de uma classe base e efetivamente abstrata chamada *BaseEnum*. Da mesma forma, classes para entidades relacionais herdavam da classe base *Entity*. Ambas classes foram criadas pelo autor e estão definidas em uma biblioteca núcleo comum, que é usada também para outras aplicações.

A classe *BaseEnum* é uma simples classe *Python* que define métodos e propriedades padrão para suas subclasses. Esta classe foi criada, ao invés de usar a classe *Enum* do *Python* pois o *Skulpt*, a implementação de JavaScript para *Python* que é usada no browser pelo Anvil Works, não suporta a classe *Enum* padrão. Isso impediria que enums fossem utilizados no código que é executado pelo cliente, apenas no server, o que cria diversas dificuldades no desenvolvimento, pois a maior utilização de enumerações é justamente em interfaces visuais para o usuário. Sendo assim, para simplificar o processo, foi criada uma simples classe que implementava todos os métodos que seriam necessários para enumerações. O pseudo-código para *BaseEnum* pode ser encontrado no Apêndice A.

De forma similar, foi criada uma classe base abstrata chamada *Entity* para gerenciar as entidades que são persistidas no banco de dados, oferecendo métodos automáticos para criação de instância, *bufferização* de mudanças, persistência a gerenciamento de listas de relacionamentos. Uma classe que estende *Entity* é automaticamente vinculada à tabela do banco de dados que tiver o mesmo nome, criando uma forma genérica de implementação da persistência, assim como controle de sequenciais, quando necessário.

Ao criar uma nova instância de subclasse de `Entity`, o construtor recebe um objeto que representa a linha do banco de dados. Este objeto é salvo internamente como o dado original. Qualquer posterior alteração nos valores da instância é, na verdade, armazenado em um dicionário que mantém temporárias as alterações. Quando necessário, a entidade pode desfazer as alterações ou persistir elas usando métodos baseados no nome da classe.

O exemplo do código da classe `Entity` encontra-se disponível no Apêndice B.

Outras duas classes auxiliares à `Entity` permitem o gerenciamento de listas de filhos e auto instanciação de pais. Ao definir um atributo de classe como sendo um *EntityDescriptor* da classe pai, podemos fazer a instanciação automática também do pai.

Código 3.1 – Exemplo de utilização de `EntityDescriptor`

```
class Parent(Entity):
    pass

class Child(Entity):
    parent = EntityDescriptor(Parent)

row = {'foo': 42, 'bazz': 0, 'parent': parentRow}
child = Child(row)
type(child.parent)
# Outputs: <class Parent>
```

Na entidade pai, é possível gerenciar a lista dos seus filhos automaticamente usando uma instância de *ManagedRelationship*, informando o campo de mapeamento. Isso faz com que o carregamento das listas seja automático e também controla as alterações nas listas.

Código 3.2 – Exemplo de utilização de `ManagedRelationship`

```
class Child(Entity):
    # Tem um atributo 'pai' que e do tipo Parent

class Parent(Entity):
    childList = ManagedRelationship(Child, "pai")

row = {'foo': 42, 'bazz': 0}
pai = Parent(row)

"""Automaticamente faz uma query no banco pelas instancias
the Child que tem como pai a instancia Parent especifica."""
pai.childList

# Controla appends, removes e changes nos dados das listas
pai.childList.append(novoFilho)
pai.childList.remove(filhoRemovido)
pai.childList[0].foo = 24
```

```
# Possibilita o reset das alteracoes, voltando ao estado original  
pai.childList.reset_changes()
```

3.4.2 Enumerações

Duas enumerações foram criadas para facilitar serem utilizadas em interfaces e classes.

A primeira é *AlimentoClassificacaoEnum*, que define quais classificações os alimentos podem pertencer, tendo como opções **Bebidas**, **Sucos**, **Frutas**, **Leites e Derivados**, **Cereais Derivados**, **Cereais**, **Grãos**, **Vegetais** e **Proteínas**. Cada valor do enum define o nome, um ícone e uma cor que são utilizados nos componentes visuais da aplicação.

A segunda enumeração é *AlimentoComposicaoEnum*, que define os tipos de nutrientes que cada alimento tem, sendo as opções **Energia**, **Proteína**, **Carboidrato**, **Fibra**, **Cálcio**, **Magnésio**, **Ferro**, **Fósforo**, **Manganês**, **Sódio**, **Zinco** e **Lipídios**.

3.4.3 Entidades

As classes de entidade criadas refletem parcialmente as tabelas existentes da aplicação, sendo classes criadas nos casos onde eram utilizadas em interfaces de forma a facilitar o gerenciamento de alterações e persistencia. Foram criadas classes para **User**, **Profissional**, **Paciente**, **PlanoAlimentar**, **Refeicao**, **MetaDiaria** e **Dieta**. Maiores detalhes sobre a estrutura de cada entidade é tratado na seção seguinte.

3.5 Banco de dados e tabelas

Foram criadas treze tabelas relacionais que são utilizadas na aplicação, sendo algumas delas diretamente relacionadas ao funcionamento da regra de negócio e algumas tabelas auxiliares.

3.5.1 Sistema e estrutura de banco de dados

O sistema de banco de dados utilizado é o sistema *built-in* do Anvil Works, construído em cima de PostgreSQL. Tabelas neste banco são criadas através do editor visual do Anvil ficam disponíveis imediatamente no código.

As tabelas aceitam colunas do tipo **Text**, **Number**, **Boolean**, **Date**, **Datetime**, **Object**, **Media** ou *Links*, vínculos para outras tabelas. Colunas do tipo *Link* sempre devem apontar para uma tabela específica e podem ser vínculos para uma linha, funcionando como o relacionamento padrão de uma base SQL qualquer, ou links para múltiplas linhas, armazenando listas.

O acesso às tabelas e aos dados delas não é feito com SQL direto, mas sim usando a interface disponibilizada pelo Anvil, que disponibiliza formas de realizar consultas usando uma variedade de operadores. Estas consultas só podem ser feitas no lado do servidor e retornam instâncias ou *iterators* de Row, um objeto *dict-like* que contém os valores da base.

Código 3.3 – Pseudo-código exemplo de utilização do banco de dados

```
# Exemplo de consultas na tabela Paciente (executando em server-side)

# Retorna a linha exata, None se nenhuma ou erro se multiplas
paciente = app_tables.paciente.get(cpf="00000000000")

# Consultando valores da linha
print(paciente['nome'])

# Atualiza linha criada
paciente['nome'] = "Fulano"
paciente.update(nome="Ciclano", cpf="11111111111")

# Remove a linha
paciente.delete()

# Adiciona uma nova linha
app_tables.paciente.add_row(nome="Beltrano", cpf="22222222222")

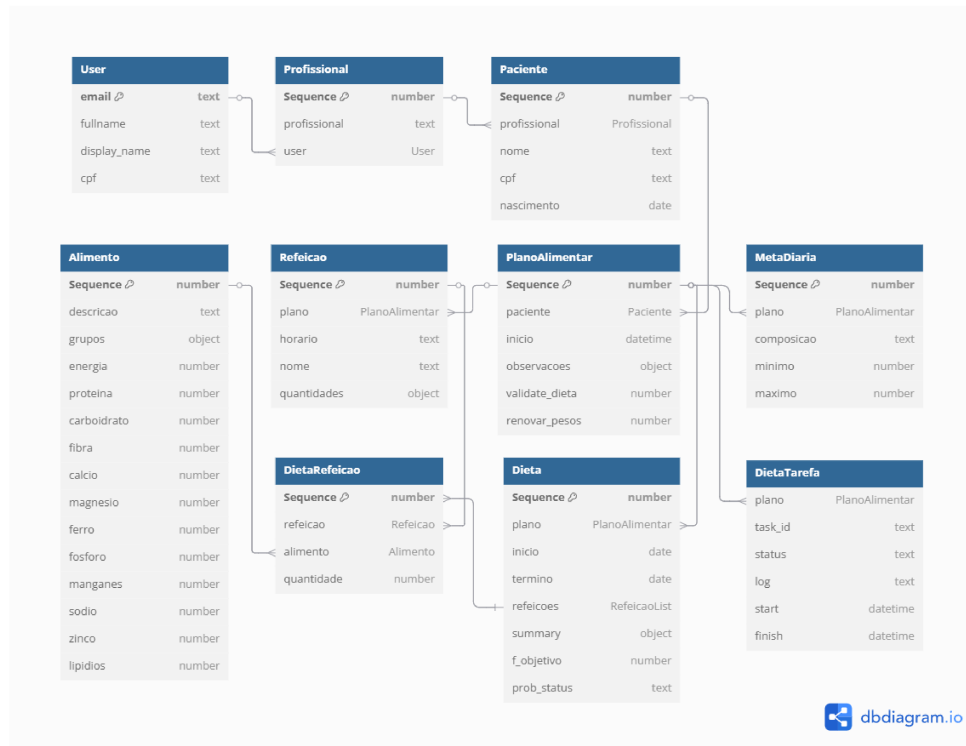
# Retorna o iterator de todos os resultados que correspondem
app_tables.paciente.search(nome=q.ilike("Fulano%"))
```

3.5.2 Tabelas

As tabelas *Users*, *UserSettings*, *Sequencer* e *FuncionalidadeVersao* foram criadas de forma a sustentar a estrutura das demais tabelas. *Users* e *UserSettings* persistem os dados básicos de identificação dos usuários e as configurações do usuário respectivamente. *Sequencer* é uma tabela especial que contém apenas uma linha, mas múltiplas colunas. A função desta tabela é controlar o sequenciador das outras tabelas que necessitam, assim como a versão atual do sistema. *FuncionalidadeVersao* é uma tabela multiaplicação que mantém todas as versões de aplicações, usada para gerenciar a versão e as notas de versão de aplicações.

As tabelas específicas da aplicação são **Alimento**, **Dieta**, **DietaRefeicao**, **DietaTarefa**, **MetaDiaria**, **Paciente**, **PlanoAlimentar**, **Profissional** e **Refeicao**. A Figura 5 exemplifica os atributos de cada tabela e as suas relações.

A coluna *MetaDiaria.composicao* é do tipo text, mas os únicos valores esperados são as chaves de *AlimentoComposicaoEnum*.

Figura 5 – Diagrama de Entidades Relacionais. Acessível em dbdiagram.io

A coluna *Dieta.refeicoes* é um link múltiplo, sendo uma lista de tuplas de *Dieta-Refeicao*, representada pela relação em vermelho.

As colunas do tipo *object* variam em cada uma das suas utilizações. Em *Alimento.grupos* é uma lista de chaves de *AlimentoClassificacaoEnum*. Já em *Refeicao.quantidades* é um dicionário contendo as quantidades definidas para cada *AlimentoClassificacaoEnum* naquela refeição. Em *Dieta.summary* é um dicionário contendo um sumário dos resultados obtidos na geração da Dieta. Em *PlanoAlimentar.observacoes* é um o texto html das observações inseridas pelo profissional.

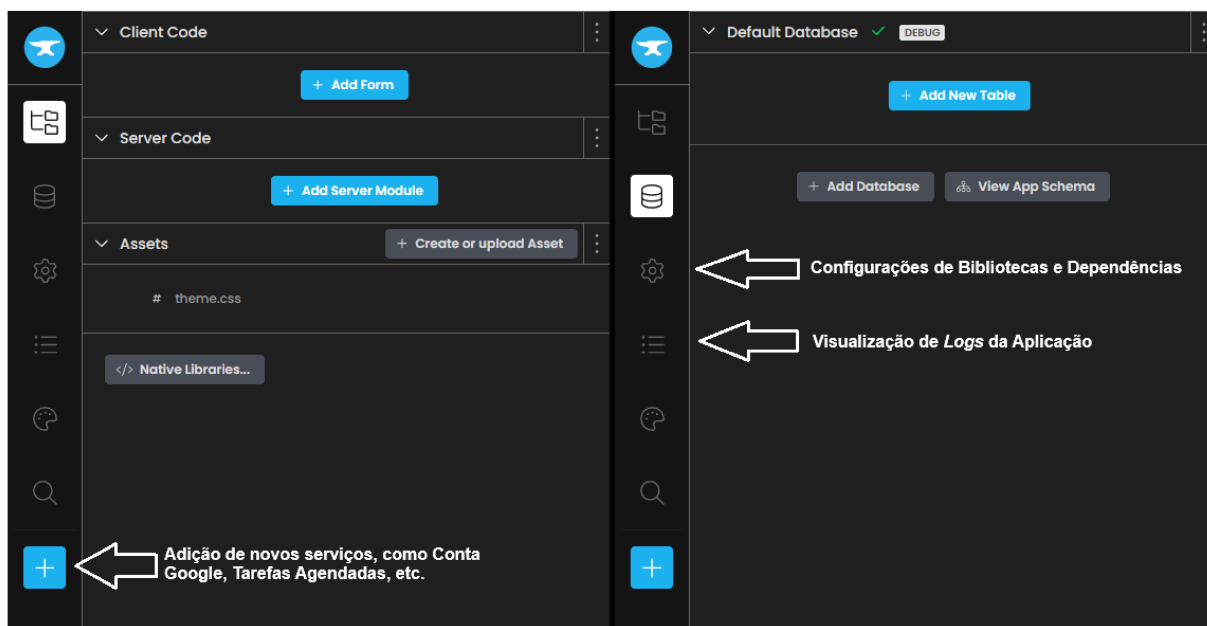
3.6 Configuração do projeto

Nesta seção, serão detalhadas as bibliotecas que foram utilizadas neste processo e sua importância no projeto instalação, configuração e definição de variáveis e componentes globais e inicialização do projeto.

3.6.1 Criação de projeto no Anvil

Projetos no Anvil são criados em questão de segundos através da interface visual e imediatamente é disponibilizado o próprio editor online da plataforma. A partir do editor é possível a criação e edição de arquivos de código *Python*, que são subdivididos em *Client Code* e *Server Code*, sendo o primeiro o código que é inerente do lado de execução no

Figura 6 – Opções básicas do Anvil



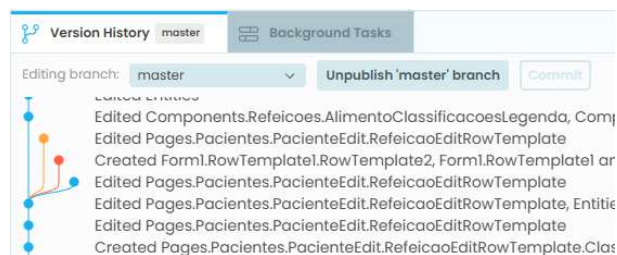
cliente, mas que também é acessível no servidor; e o segundo o código que só é acessível no servidor da aplicação. Em ambos casos é possível criar módulos, que são arquivos de extensão padrão do *Python* e que contém código puro em *Python*, ou pacotes, que funcionam como pastas para armazenar outros pacotes e módulos dentro, mas também permitem a criação de código direto no nível do pacote. No *Client Code* é também possível a criação de Formulários, que se referem a interface visual da aplicação.

Além de arquivos de código *Python* também é possível a criação das tabelas de banco de dados diretamente pela interface. A criação de projeto no Anvil é um processo simples, bastando a criação de uma conta que pode ser criada vinculada a uma conta Google. A plataforma oferece opções de plano gratuito e planos pagos. O autor já havia previamente adquirido um plano pago, porém os benefícios do plano para este desenvolvimento serão mínimos.

3.6.2 Controle de versão e repositório

A própria plataforma do Anvil oferece um Sistema de Controle de Versão imbutido. Nele é possível a criação de ramificações, criação de *commits*, mesclagens e resoluções de conflitos. A plataforma realiza *commits* automáticos a cada quinze minutos sem interação e oferece e é possível visitar *commits* e ramificações antigas a qualquer momento. A aplicação também pode ser automaticamente publicada em conta GitHub vinculada e clone local também é possível através do *Git*, embora não seja necessário.

Figura 7 – Sistema de Controle de Versão Imbutido



3.6.3 Instalação de pacotes

Código *Python* executado no navegador adere aos pacotes disponíveis pelo *Skulpt*, porém para o código que é executado no servidor o Anvil oferece configurações do ambiente. Dentre as opções disponíveis, foi escolhida a opção que utiliza a versão *Python* 3.10, que é a mais recente dentre as opções disponíveis e que permite maior controle de quais pacotes são instalados no ambiente do servidor. Algumas opções de pacotes pré-instalados são oferecidos ao usuário, sendo escolhida a opção *Standard*, que já disponibiliza um ambiente pré-instalado com alguns pacotes. Além dos pacotes iniciais, foi manualmente instalada o PuLP, utilizado na otimização das dietas.

3.6.4 Configuração de domínio e servidor

A aplicação foi desenvolvida como uma aplicação *PWA*, o que significa que é nativamente *web*, sendo possível ser instalada em dispositivos compatíveis, desde que a partir de navegadores também compatíveis.

Para publicação do aplicativo, foi utilizado o domínio privado da empresa do Autor, OruData Sistemas de Informática. A aplicação está publicamente acessível através da URL <<https://dietas.orudata.com.br/>>. A criação de subdomínio foi feita através da plataforma *GoDaddy*.

No processo de publicação da aplicação pelo Anvil, é possível informar um domínio personalizado (recurso disponível em planos pagos), informando qual ramificação de código representa a aplicação em ambiente de produção. O controle da hospedagem da aplicação no domínio informado é feito inteiramente pelo Anvil. O ambiente de execução do código do servidor é distribuído e é construído a cada chamada.

3.6.5 Configuração de dependências

O Anvil oferece a possibilidade de vinculação de outros projetos como “dependências” do projeto atual. Adicionar um outro projeto como dependência do projeto atual significa que todo código criado na dependência está também disponível no projeto atual. A utilização de dependências incentiva o compartilhamento de código público na comuni-

dade, mas também possibilita a reutilização de códigos que poderão ser compartilhados por múltiplas aplicações.

O projeto deste trabalho se utiliza de cinco dependências, sendo duas delas proprietárias do autor e 3 dependências de código aberto.

A dependência *m3* é uma dependência nativa do Anvil e que oferece opções de componentes visuais básicos utilizando o sistema de estilos *Material Design 3* (MD3). É utilizada nos *layouts* e componentes mais básicos como campos de texto e número, seletores e botões.

No mesmo escopo, a *core_m3* é uma dependência proprietária do autor, baseado na dependência anterior, construída para agrupar apenas componentes visuais que respeitem o sistema de estilos *MD3* que poderiam ser reaproveitados em múltiplas aplicações. É utilizada nos componentes visuais de uso mais específico, mas que ainda não diretamente vinculados à nenhuma regra de negócio específica, como formulários de Usuário, campos de CPF e campos de sugestão automática.

A principal dependência, entretanto, é a *OruData*, que contém todo o código de ambiente e de variáveis e classes globais e abstratas utilizados na aplicação. A dependência foi criada pelo autor e contruída ao longo de diversos anos de utilização, sendo amplamente ajustada durante este trabalho com implementação de padrões de classe, Services, Orientação à Objetos e gerenciadores de contexto. Nesta dependência estão as classes abstratas de *Entity*, e **BaseEnum**, além de estruturas que armazenam as variáveis de identificação e configuração do ambiente da aplicação, dados do dispositivo, usuário da sessão e outras variáveis globais relacionadas. Também está definido nela as implementações utilitárias para validações e criação de interfaces que permitam de forma genérica as operações de criação, leitura, alteração e remoção (*Create, Read, Update and Delete - CRUDs*). Incluída nesta dependência estão também as duas outras dependências restantes: *routing* e *anvil_extras*

A dependência *routing* é nativa do Anvil e experimental. Ela inclui as ferramentas necessárias para definir as rolas de carregamento das páginas da aplicação, com carregamento, navegação e *caching*.

Por fim, *anvil_extras* é uma dependência de código aberto mantida pelo grupo *Anvilistas*, que são usuários da plataforma, oferecendo uma lista de componentes visuais e funcionalidades extras não ainda nativas. Desta dependência são muito utilizadas as ferramentas de *Logger*, para registrar o funcionamento da aplicação, e *call_async*, para processamento de requisições assíncronas não-nativamente suportadas pela plataforma.

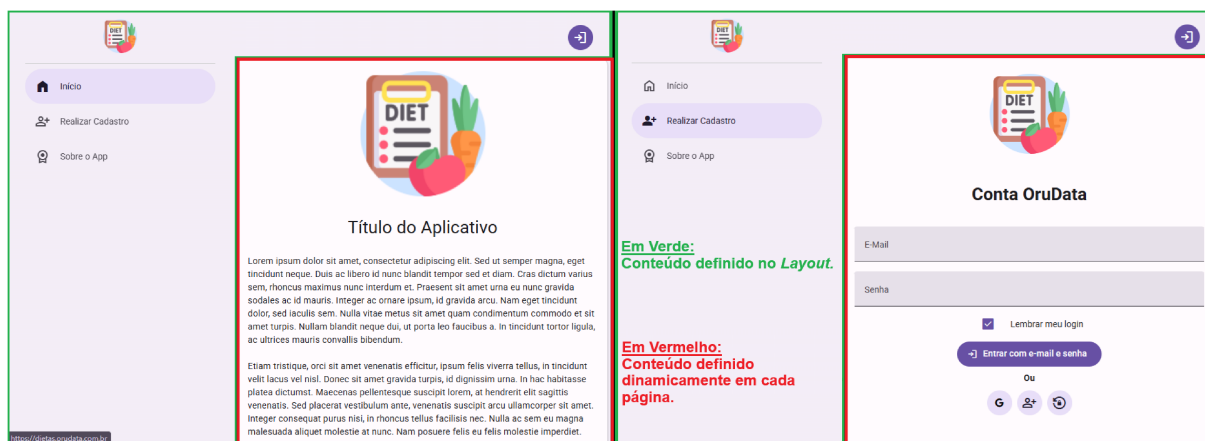
3.7 Criação de *layouts* e rotas

A navegação e exibição das páginas é feita usando *Layouts* e *Routing*, duas funcionalidades do Anvil, sendo *Layouts* relativo à reutilização e padronização visual das interfaces e *Routing* relativo a vinculação de caminhos de navegação à cada página.

3.7.1 *Layouts*

Com a utilização de *Layouts* é possível criar um único modelo básico de página, que todas as páginas iriam herdar, e apenas desenhar o conteúdo interno de cada página após isso. A Figura 8 exemplifica duas páginas distintas criadas usando *Layouts*.

Figura 8 – Utilização de *Layouts* e conteúdos dinâmicos.



A estrutura central de cada página, que contém a barra de títulos e botão de sessão de usuário, a barra de navegação lateral e um container de conteúdo, é definida apenas uma vez na criação do *Layout* (realçado em verde na Figura), enquanto os dados que são exibidos dentro do container de conteúdo são definidos em cada página que herda aquele *Layout* (realçado em vermelho na mesma figura).

Foram criados dois layouts, um para ser usado em páginas públicas, quando ainda não houvesse um usuário em sessão, e uma para ser usada na navegação das páginas que exigiam um usuário.

O *layout* público é usado numa página de entrada, cuja função é explicar sobre o que é a aplicação, além das páginas de realizar cadastro e créditos. O container de conteúdo é simples que pode receber qualquer tipo genérico de páginas.

O *layout* das páginas que exigem um usuário em sessão exibe mais opções de navegação, como o cadastro de pacientes, dietas e emissão de relatórios. As páginas construídas utilizando este *Layout* tem disponíveis três espaços chamados de *slots* ou encaixes em seu container de conteúdo, onde é possível posicionar de forma padrão certos componentes. No canto superior esquerdo do container de conteúdo um encaixe é reservado

Figura 9 – Layout utilizado ao acessar o site sem um usuário em sessão

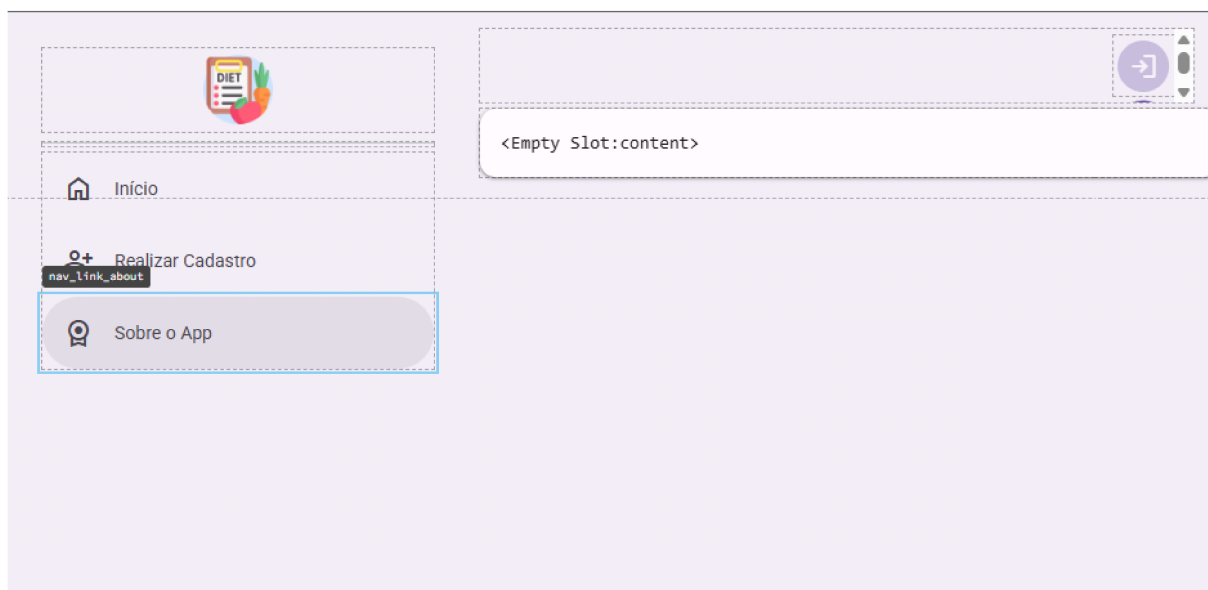


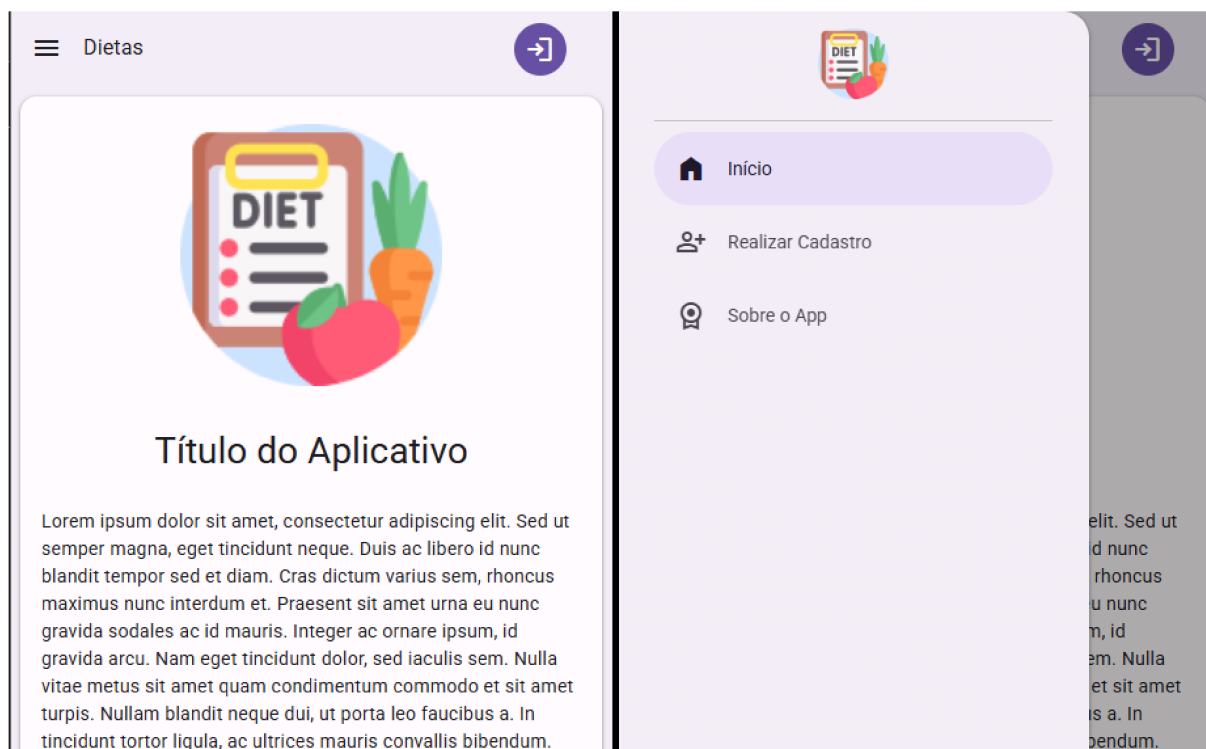
Figura 10 – Layout utilizado ao acessar o site com um usuário em sessão



para o título da página, enquanto um outro encaixe, na mesma linha, mas alinhado à direita, é reservado para botões de ações, como salvar ou cancelar. Abaixo desta linha é disponibilizado um encaixe genérico de conteúdo, como o utilizado no container do *layout* público.

Os layouts também se adaptam automaticamente conforme a tela do dispositivo, oferecendo uma visão moderna tanto em telas maiores como de um computador de mesa ou notebook, quanto em dispositivos móveis como tablets e celulares. Em telas maiores, a barra de navegação é fixada na lateral esquerda da aplicação, sempre visível. Já em telas menores, ela é retraída e escondida dentro de um “menu de hamburguer” mobile, que

Figura 11 – Visualização da aplicação em dispositivos de tela pequena. À direita, exibição do menu de navegação expandido



pode ser expandido com o toque, deixando o espaço disponível inteiro para o conteúdo da página.

Esta adaptação do menu lateral é possível através de *media-queries* definidas no escopo geral e herdadas da dependência nativa *m3*. Outras adaptações são feitas no escopo local da aplicação para a exibição de um título do aplicativo na mesma linha do botão de iniciar sessão quando em modo mobile, também utilizando de *media-queries* feitas pelo autor.

3.7.2 Rotas

A navegação é inteiramente realizada utilizando o conceito de rotas e a dependência *routing*. Com ela, é possível definir quais interfaces devem ser exibidas ao navegar para cada rota interna da aplicação, além dos requisitos necessários para acessar tais páginas e o que fazer nos casos de falha.

Por exemplo, o formulário de cadastro e acesso de usuário só deve ser exibido usando o *layout* público e somente quando não houver usuário em sessão. Em contrapartida, o formulário de cadastro de pacientes só pode ser exibido quando o usuário já ter sido informado.

Utilizando a dependência *routing*, cada rota pode ser definida através da criação de

uma classe específica da rota que estenda a classe *Route*, inclusa na dependência. A classe criada deve definir no nível de classe valores para os atributos *path* e *form*, duas *strings* que informavam o caminho relativo e o formulário que seria exibido na determinada rota respectivamente.

Código 3.4 – Exemplo de utilização de Rotas

```
class SomeRoute(Route):
    path = '/some'
    form = 'Pages.SomeForm'

# A rota SomeRoute ira exibir o form encontrado no caminho Pages/
# SomeForm quando a rota myexemple-app.com/some for acessado
```

Além dos atributos *path* e *form*, outras funcionalidades facilitaram a implementação da complexidade de rotas da aplicação. As mais importantes destas funcionalidades são os métodos *before_load*, *load_data* e *cache_deps*, tratadas em mais detalhes nos próximos parágrafos.

O formato de definição das rotas permite a implementação de múltiplas heranças em sua criação, o que permitiu a criação de três classes auxiliares para a definição de rotas:

Código 3.5 – Classes *SecureRoute* e *UnloggedRoute*

```
# mock code
class SecureRoute:
    def before_load(self, **loader_args):
        from .Globals import CommonData
        logged_user = CommonData().get_logged_user_forced()
        if not logged_user:
            raise Redirect(path="/login")
        return logged_user

class UnloggedRoute:
    def before_load(self, **loader_args):
        from .Globals import CommonData
        if CommonData().get_logged_user_forced():
            raise Redirect(path="/dashboard")
```

Rotas que exigiam que o usuário estivesse em sessão estendem *SecureRoute*, enquanto rotas que exigissem que não houvesse usuário informado estendem *UnloggedRoute*, com ambos os casos e os casos genéricos também estendendo *Route*. Ao encontrar uma rota definida para o caminho de navegação, o método *before_load* é executado. Caso ele não lance erros, o processo é concluído com sucesso, porém se uma exceção do tipo *Redirect* for lançada com um caminho como argumento do construtor, a nova rota informada é carregada no lugar. Desta forma, a verificação é feita sempre antes de carregar cada rota.

Se uma rota segura é carregada sem um usuário carregado, a aplicação automaticamente redireciona para a página de início de sessão e, caso uma rota pública fosse acessada enquanto em sessão, o usuário é redirecionado ao *dashboard*.

Desta forma, é possível definir de forma simples rotas públicas e privadas como o código de exemplo abaixo:

```
class HomeRoute(UnloggedRoute, Route):
    path = '/'
    form = 'Pages.Home'

class DashboardRoute(SecureRoute, Route):
    path = '/dashboard'
    form = 'Pages.Dashboard'
```

Um caso particular é o formulário de pacientes, pois precisa ter rotas definidas para criar, visualizar e editar um paciente no mesmo formulário. Além disso, a navegação entre editar e visualizar precisa ser feita sem um recarregamento da página, utilizando os dados já carregados para no modo anterior. Também é importante que seja possível a navegação direta a um paciente através de uma *URL* específica.

Para atender estes requisitos foi primeiro implementada uma classe genérica para telas que implementam funcionalidades *CRUD*.

Código 3.6 – Classe CrudRoute

```
class CrudRoute:
    entity = None
    create_route = False

    def load_data(self, **loader_args):
        if self.create_route:
            return None
        row = loader_args["nav_context"].get("row")
        if row is None:
            seq = loader_args["params"]["id"]
            row = self.entity.load_by_id(seq)
        return row
```

Rotas que estenderem *CrudRoute* ganham o requisito de implementarem os atributos de classe *entity* e *create_route*, sendo este último falso por padrão. Ao acessar um caminho, mas depois de identificar a rota e passar com sucesso pela validação de *before_load*, o método *load_data* é invocado se definido. A responsabilidade deste método é retornar o objeto item do formulário que será carregado, a instância a ser editada nos *CRUDs*.

No caso desta implementação genérica, no caso de rota de criação, é retornado *None* que representa que nenhum dado persistido será carregado. Em seguida, é verificado

se a navegação foi feita por dentro da aplicação. Quando a tela de busca de pacientes navega para a tela de *CRUD* de paciente, por exemplo, o objeto referente ao paciente buscado já foi carregado na própria tela de busca, sendo assim não seria necessário realizar uma nova busca, uma tela simplesmente repassa o objeto para a próxima usando o contexto de navegação, economizando uma viagem ao servidor. Isso é representado pela linha `loader_args["nav_context"].get("row")`.

A implementação, entretanto, serve para os casos onde o usuário está navegando diretamente à *URL* da informação. Caso o usuário esteja navegando diretamente para o caminho `"/pacientes/1"` salvo em seus favoritos. Ao iniciar a página, não haverá valores no contexto de navegação, pois a aplicação inteira será carregada diretamente na página específica. Neste caso, o `"id"` é identificado nos parâmetros de caminho e usado para carregar a instância específica durante o carregamento da página, exibindo imediatamente sem erros.

Para resolver os problemas de navegação interna e as rotas de criação, visualização e edição, as rotas de paciente são então definidas da seguinte forma:

```
class PacienteCRUD(SecureRoute, CrudRoute, Route):
    cache_form = True
    form = "Pages.Pacientes.PacienteEdit"
    entity = Paciente

    def cache_deps(self, **loader_args):
        return None

class PacienteCreateRoute(PacienteCRUD):
    create_route = True
    cache_form = False
    path = '/pacientes/create'

class PacienteEditRoute(PacienteCRUD):
    path = '/pacientes/:id'
```

A classe `PacienteCRUD` é definida como uma classe de rota, que exige sessão de usuário e de *CRUD*. Ela será a base para as três rotas mencionadas anteriormente e por isso nela é definido o formulário, mas não a rota. O atributo `cache_form` e o método `cache_deps` são os responsáveis pela navegação entre modos sem recarregamento da página, onde o primeiro habilita o armazenamento *cache* do formulário para rotas já acessadas anteriormente e o segundo define a regra de quando utilizar o *cache*. O retorno de `cache_deps`, usualmente um dicionário, informa quais parâmetros de *queries* e *hash* de rota foram usados para carregar a página, o que cria a “chave” na qual o formulário será salvo em *cache*. Estaticamente retornar `None` diz que para todas os *query params* e *hash routes* o mesmo formulário deverá ser usado, ou seja, ao navegar de `/pacientes/1?mode="view"` para

`/pacientes/1?mode="edit"`, ambos os casos o mesmo formulário é utilizado e a mudança é apenas visual.

Por fim, as rota são definida como extensões da classe `PacienteCRUD`, tendo a rota de criação o atributo relacionado a isso definido como verdadeiro e o armazenamento em *cache* desabilitado, enquanto a rota de edição e visualização recebe um parâmetro no caminho (usado em `load_data`).

4 RESULTADOS

Ao final do desenvolvimento, foi construída uma plataforma web. Nas seções a seguir, serão apresentadas as diferentes funcionalidades da aplicação e o estudo de caso utilizando o modelo desenvolvido por (SOARES, 2023).

4.1 Funcionalidades da ferramenta

A aplicação é essencialmente composta de três interfaces de usuário: o *Cadastro de Pacientes*, a *Geração de Dietas* e a *Emissão de Dietas Geradas*. Nesta seção serão apresentadas as interfaces listadas, assim como o formato de comunicação com a fachada de requisições para execução de métodos *CRUD* e o processo assíncrono de geração de dietas com base nos parâmetros informados e cadastrados.

4.1.1 Cadastro de paciente

No *Cadastro de paciente* o profissional em sessão deve informar os dados do paciente em questão. A tela implementa os funcionamentos de *CRUD*, podendo criar pacientes e tendo por padrão a exibição do paciente selecionado e já criado em formato de leitura,

Figura 12 – Esqueleto do *CRUD* de Paciente

The image shows a wireframe of a web application for patient registration. On the left is a sidebar with a logo at the top and navigation links: Dashboard, Pacientes, Dietas, Relatórios, and Sobre o App. The main content area is titled 'Cadastro de Paciente' and includes a toggle switch and two icons (refresh and save). Below the title is a section 'Dados Básicos' with a 'Nome Completo' field. Underneath are two fields: 'CPF' and 'Nascimento' (with a calendar icon). The next section is 'Planos Alimentares' with a tooltip icon and a heading 'planos_title_tooltip_heading'. Below this is a 'Plano Vigente' section with 'Inicio' and 'Termino' date pickers (the latter is 'Não definido'). There is a rich text editor with a toolbar (Normal, Bold, Italic, Underline, Link, Unlink, List, Unlist, Indent, Outdent) and a text area for 'Observações adicionais sobre o plano'. At the bottom, there is a 'Definições' section with a plus icon.

Figura 13 – Interface de agendamento de geração de dietas

Timers: task_check_timer

Geração de Dietas

Paciente

Plano Alimentar

Gerar nova dieta a cada: 7

Renovar pesos a cada: Nunca

Quantidade de dias que cada dieta deve ser reutilizada.

Quantidade de dias para renovar os pesos.

Gerar dieta para o Plano definido

percentage_text

careta_status_text

Ver Log

Ver Receita Gerada

mas podendo habilitar o modo de edição ao interagir com o *switch* apropriado, ao lado do título da página. Para gravar as alterações, são obrigatórios os dados de Data de Nascimento e CPF, havendo uma validação do CPF para impedir que o mesmo paciente seja cadastrado múltiplas vezes para o mesmo profissional.

Ainda nesta interface, é possível visualizar e implementar um Plano Alimentar do paciente, definindo uma data de início e fim para o plano, informando quais refeições serão necessária e quais as metas de nutrientes que devem ser atingidas em cada dia do plano, considerando todas as refeições de cada dia. É também disponibilizado um campo de texto livre para o profissional realizar anotações sobre plano, caso seja necessário.

Ao cadastrar refeições, é necessário informar um nome, horário recomentado e quantidades de alimentos que precisam ser incluídos naquela refeição. Não há limites para a quantidade de refeições que podem ser cadastradas, mas cada refeição precisa pelo menos uma quantidade diferente de zero.

Para as metas diárias, uma tabela é criada e pré-preenchida com cada nutriente que pode ser analisado, com sugestões de valores mínimos e máximos que podem ser alterados pelo usuário livremente.

4.1.2 Geração de dietas

Concluído o cadastro de pacientes e do seu plano alimentar, o profissional tem a sua disposição uma interface para a geração das dietas utilizadas para o período informado do plano. A tela automaticamente exibe um componente de sugestão de pacientes, listando todos os pacientes cadastrados para o profissional em sessão. Ao selecionar um paciente, uma lista dos seus planos alimentares é exibida, automaticamente selecionando o plano alimentar mais recente. Dois campos são exibidos para preenchimento de dados que serão necessários durante a geração das dietas usando o algoritmo de otimização:

1. **Gerar nova dieta a cada:** campo numérico, informa por quantos dias uma dieta gerada deverá ser reutilizada, gerando uma nova dieta após esta quantidade de dias.
2. **Renovar pesos a cada:** campo numérico. A cada criação de dieta, cada alimento utilizado recebe uma atualização nos seus pesos de preferência de forma a ser menos provável de ser escolhido novamente. Este campo define a cada quantos dias estas medidas devem ser completamente descartadas e reiniciadas para seus valores iniciais.

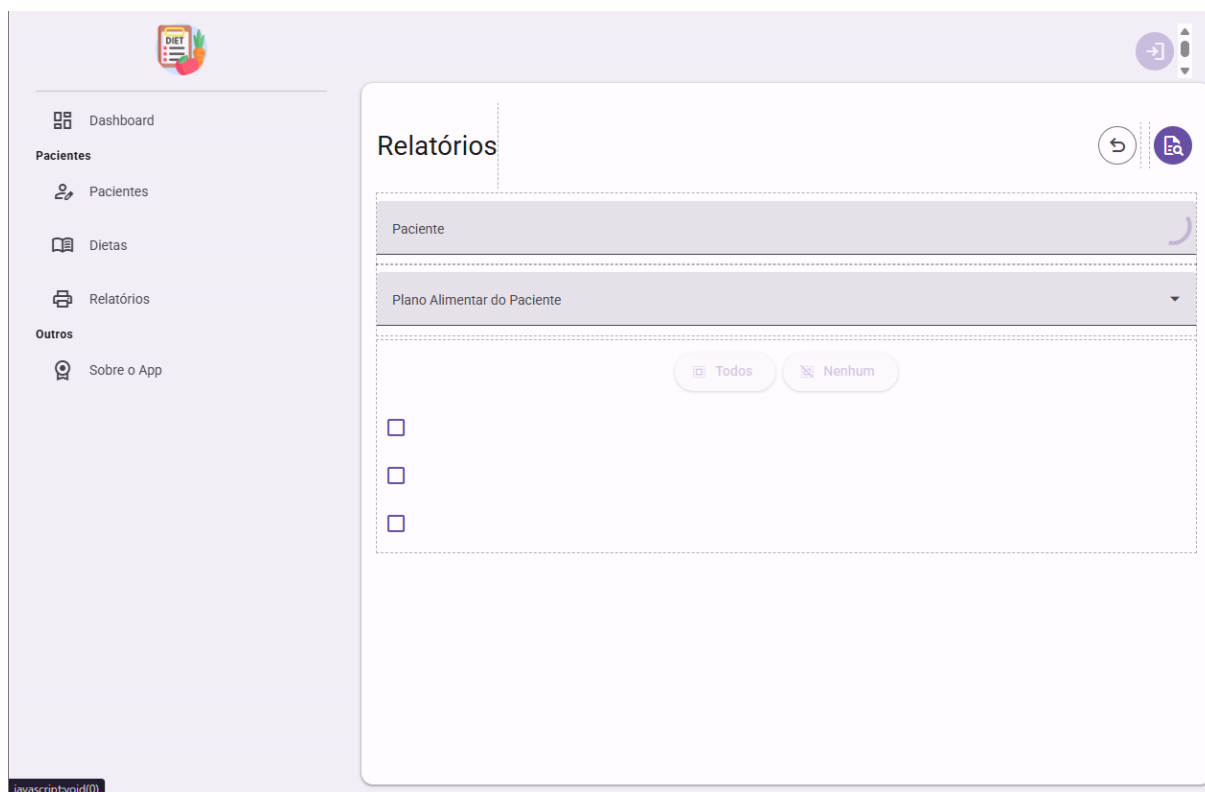
Após preenchidos os valores o usuário deve então clicar no botão para gerar as dietas. O tempo de processamento e geração das dietas irá variar dependendo do tamanho do período do plano, da quantidade de refeições e restrições criadas, dos parâmetros informados para renovação e criação das dietas e da base de alimentos utilizada. Sendo assim, para tratar todos os casos, a geração da dieta é feita através de uma tarefa assíncrona executada no servidor.

Ao iniciar a geração da tarefa, a chamada realizada entrega do servidor um objeto de tarefa em execução em segundo plano. Um temporizador faz verificações no estado da tarefa a cada segundo, atualizando uma barra de progresso na tela para que o usuário possa realizar um acompanhamento do processo e ter uma previsão de conclusão.

Caso o processo resulte em falha, um botão é exibido com a opção de visualizar os registros da execução, de forma a identificar qual o problema encontrado durante o processo. No caso de sucesso, um outro botão é exibido para realizar a navegação para a página de *Emissão de Dietas Geradas*.

É importante salientar que não é exigido ao usuário que mantenha a aplicação carregada durante o processo de geração das dietas. Uma vez iniciado o processo, o usuário que não desejar acompanhar o processo, poderá fechar o aplicativo ou a aba e janela na qual a aplicação está carregada tranquilamente. Ao acessar a tela de geração para aquele paciente e plano escolhidos anteriormente, a aplicação irá automaticamente verificar o processo que estava em execução e exibir a situação de falha ou sucesso que foi encontrada

Figura 14 – Página de geração de relatórios



durante a execução, ou exibir a barra de progresso atualizada, caso este processo ainda esteja em execução.

4.1.3 Emissão de dietas geradas

Uma página de emissão de relatórios é disponibilizada ao profissional que desejar reemitir uma dieta já gerada anteriormente. Os campos iniciais são semelhante ao de geração, sendo possível selecionar um entre os diversos pacientes cadastrados para o profissional e então selecionar um plano alimentar do paciente em questão. A interface disponibiliza a opção de visualização em tela da dieta ou emissão em arquivo no formato PDF para *download*.

O relatório emitido exibe os dados de início e fim do Plano Alimentar do Paciente, as observações informadas como cabeçalho da página, seguido pela lista de Dietas geradas, identificadas pelo período de validade da dieta que foi automaticamente criado usando os parâmetros informados na geração. Para cada período é exibido uma lista das refeições, os alimentos automaticamente escolhidos para a refeição e a quantidade. Ao final do período, uma pequena tabela de resumo mostra quais serão os valores atingidos para cada nutriente em cada dia ao seguir a dieta gerada.

4.1.4 Comunicação com servidor

Classes e métodos definidos em arquivos do tipo *Server Code* só são acessíveis quando executados no servidor. Nestes arquivos estão definidas as regras de persistência e consultas em banco de dados, assim como o código de otimização de dietas, por utilizar a biblioteca PuLP, não disponível no lado cliente.

Os arquivos de execução no *back-end* são organizados em *Resource*, que é a fachada, onde as requisições e chamadas de servidor são recebidas; e *Service*, onde ficam todos os métodos de regra de negócio, incluindo manipulação de tabelas.

Na fachada, cada conjunto de métodos de mesmo contexto na regra de negócios é agrupado em um mesmo arquivo. Por padrão, são definidos métodos para buscar por identificador, uma busca por parâmetros, um método para salvar, atualizar e remover são definidos para cada entidade, resultando em algo semelhante ao conjunto exemplo abaixo.

Código 4.1 – Exemplo de Resource: PacienteResource

```
import anvil.server
from ..Service import paciente_service

@anvil.server.callable
def getPacienteById(sequence:int):
    return paciente_service.get_by_sequence(sequence)

@anvil.server.callable
def getPacienteFind(*args, **kwargs):
    return paciente_service.find(*args, **kwargs)

@anvil.server.callable(require_user=True)
def getPacienteByUser(user=None):
    return paciente_service.get_by_user(user)

@anvil.server.callable(require_user=True)
def postPaciente(paciente, plano_vigente=None):
    return paciente_service.save(paciente, plano_vigente)

@anvil.server.callable(require_user=True)
def putPaciente(paciente, changes, plano_vigente=None):
    return paciente_service.update(paciente, changes, plano_vigente)

@anvil.server.callable(require_user=True)
def deletePaciente(paciente):
    paciente_service.delete(paciente)
```

Os métodos na fachada não contém regra de negócio: eles servem apenas para definir quais métodos do *service* estão sendo expostos para a aplicação. A anotação `anvil.server.callable` é essencial para tal funcionamento. Todo método definido em arquivos

do tipo *server* que for anotado desta maneira passa a ser acessível através da chamada do método `anvil.server.call` no lado cliente. Sendo assim, é possível encontrar o paciente de sequencial 1 a partir da interface de pacientes ao executar o código `anvil.server.call('getPacienteById', sequence = 1)`. Foi convencionado para métodos do tipo a utilização de *camel-case* e um prefixo equivalente a um método HTTP equivalente da operação.

Como mencionado anteriormente, uma das formas de realizar uma operação com o servidor a partir do código executado no cliente é através `anvil.server.call`. A execução neste caso é feita de forma síncrona, pausando a aplicação e exibindo um *spinner* indicando o carregamento. Nativamente, não existe formas de realizar chamadas assíncronas no Anvil, porém o módulo *non_blocking* da dependência *anvil_extras* implementa o método *call_async*, que permite chamadas de servidor e outras execuções de forma assíncrona usando *Promises* do JavaScript. Isto é utilizado para agilizar o acesso de certas telas realizando alguns pré-carregamentos no momento do início de sessão.

Já nos arquivos categorizados como serviços são definidas as classes de serviço de cada entidade da aplicação. Elas herdam *AbstractCrudServiceClass*, uma classe genérica de serviços definida na dependência *OruData* e que oferece por padrão métodos para buscar, recuperar, salvar, atualizar e remover entidades. No caso de remoção, a classe também oferece operações de cascadeamento automático configurável de relacionamentos órfãos, algo não nativo do banco de dados embarcado do Anvil. Cada entidade tem sua instância desta classe ou de uma classe herdada que sobrescreve ou implementa novos métodos específicos da regra de negócio da entidade específica.

4.1.5 Geração parametrizável de dietas

Como mencionado anteriormente, o processo de geração das dietas pode levar muito tempo, o que poderia ocasionar *timeouts* e uma má experiência de usuário. Para contornar esta situação, quando o usuário solicita a geração de das dietas, a fachada de serviços agenda uma tarefa em segundo plano.

Código 4.2 – Agendamento de Dietas

```
@anvil.server.callable(require_user=True)
def postDietaGerar(plano_seq, vigencia_dieta, renovacao_pesos):
    task = anvil.server.launch_background_task('gerar_dieta',          None
        , plano_seq, vigencia_dieta, renovacao_pesos)
    dieta_service.registrar_tarefa(plano_seq, task)
    return task
```

A chamada `anvil.server.launch_background_task('gerar_dieta', ...)` cria uma tarefa que é executada em segundo plano, retornando um objeto de referência da tarefa. Em seguida, é registrado o vínculo entre a tarefa em execução e o plano em questão. Isto é realizado para ser possível identificar se já existe uma tarefa para o plano e se ela já

foi concluída e seu resultado, por exemplo no caso de reabrir a tela de geração de dietas enquanto o processo ainda não foi concluído.

A tarefa é retornada para o cliente, que passa a fazer consultas de verificação de conclusão a cada segundo:

Código 4.3 – Pseudo-código de checagem da tarefa

```
def task_check_timer_tick(self, **event_args):
    """Codigo automaticamente executado a cada 1 segundo"""
    # self.task: definido como o resultado da chamada de agendamento
    if self.task.get_state()['progress']:
        # Exibe e atualiza a barra de progresso
        # ...
    if self.task.get_state()['status'] == 'COMPLETED':
        # Exibe o botao de navegar para o report
        # ...
    if self.task.get_state()['status'] == 'ABORTED':
        # Exibe o botao de logs de erro
        # ...
```

Do lado do servidor, o processo é iniciado logo após a chamada de inicialização da tarefa em segundo plano. O processo inicia com as consultas necessárias para o processo, carregando todos os alimentos disponíveis no banco de dados e o plano alimentar específico do planejamento, com as suas refeições e metas diárias vinculadas. É feito um cálculo da quantidade de períodos diferentes, com dietas diferentes com base nas datas de início e fim do plano escolhido, além dos parâmetros de vigência de dieta e renovação de pesos. Então, o processo real de geração de dietas é iniciado, iterando para cada período e gerando uma nova dieta.

Para cada iteração, os pesos dos alimentos são atualizados, começando com 0 e sendo atualizados conforme suas utilizações dentro das dietas. Após definição ou atualização dos pesos, são definidos os objetivos padrão do problema de otimização, seguidos pelas definições de quantidades de cada refeição que foram parametrizados na tela de Pacientes. Neste momento, para cada meta diária definida, um novo elemento é adicionado no função de otimização, conforme parâmetros definidos pelo profissional. Na última etapa de processamento é iterada cada refeição para verificar quais alimentos são aceitáveis de acordo com o que foi definido em tela, executando o método de solucionar da função objetivo.

O processo final que ocorre após o resultado de cada dieta na iteração dos períodos é um armazenamento de instâncias dos dados das dietas em uma lista temporária em memória, que vão ser persistidas apenas no final do processo como todo. Após o último período ser iterado, uma transação é aberta no banco de dados, persistindo cada entidade produzida no processo.

Durante o processo por inteiro uma variável de estado da tarefa é atualizada com

dados de situação, percentual de progresso e registro de erros e execução, que serão consultados em tempo real pela aplicação cliente e exibidos ao usuário.

4.2 Estudo de caso

Após o desenvolvimento completo do sistema, foi realizado um estudo de caso para validar seu funcionamento e verificar se ele atende aos requisitos definidos no início do desenvolvimento e aos objetivos propostos inicialmente. A seguir são apresentados os resultados observados neste estudo de caso. Os testes realizados nesta etapa do trabalho visam:

- Verificar a possibilidade de criação de novo usuário para profissional na plataforma de forma totalmente autônoma;
- Verificar a funcionalidade de cadastro de paciente para o profissional, bem como o cadastro de um plano alimentar para este;
- Verificar a funcionalidade de criação automática de dietas para um paciente e a emissão da dieta gerada.

Os testes foram realizados usando uma máquina comum acessando a plataforma através do URL <<https://dietas.orudata.com.br/>>. O equipamento utilizado neste estudo de caso tem a seguinte configuração:

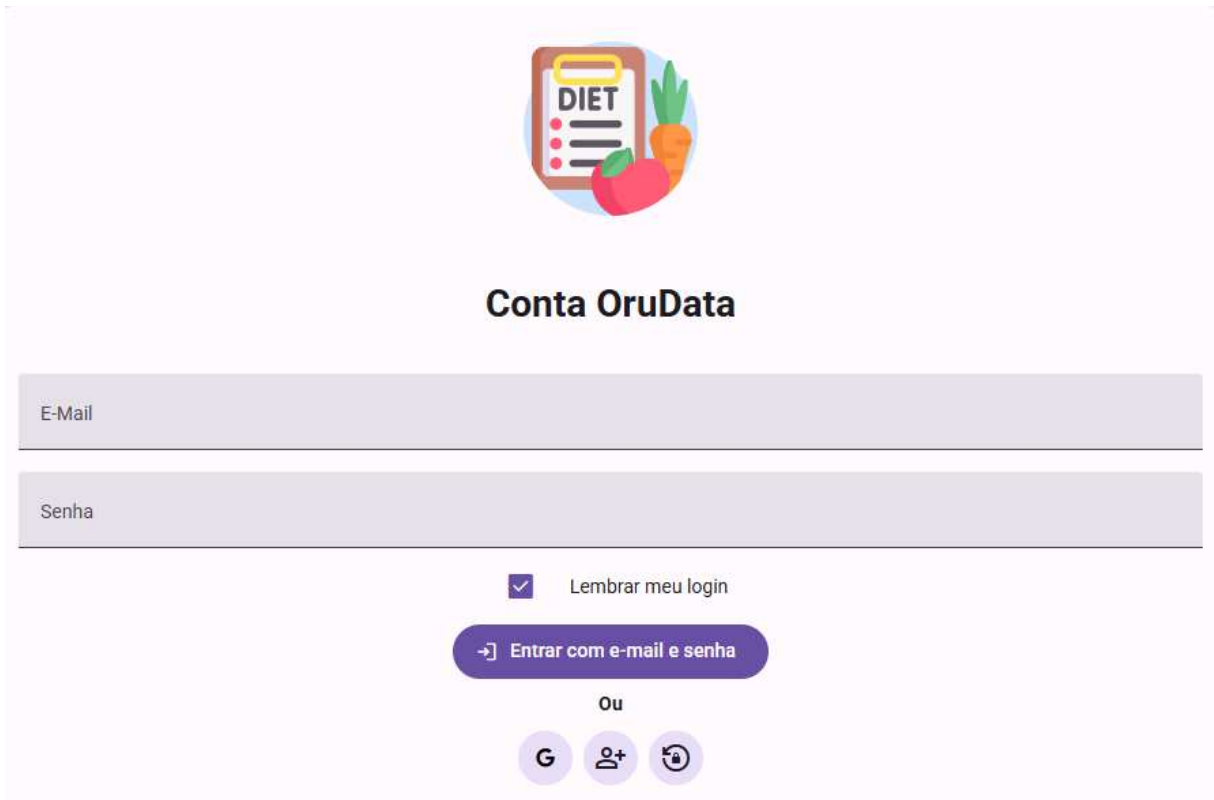
- **Processador:** Intel(R) Core(TM) i7-8750H @2.20GHz
- **RAM:** 32GB DDR4 **Sistema Operacional:** Windows 11 **Navegador:** Google Chrome 135.0.7049.96

4.2.1 Cadastro de usuário

Ao acessar a aplicação pela primeira vez, o usuário é recebido pela página inicial, que exibirá informações da aplicação. A partir daí, o usuário pode realizar um novo cadastro ou iniciar uma sessão de usuário já cadastrado clicando no botão de acesso no canto superior direito ou no botão de navegação à esquerda intitulado “Realizar Cadastro”. Em ambos os casos uma interface de informação de usuário é exibida. Nesta interface, o usuário poderá realizar início de sessão, caso já tenha usuário cadastrado, criar um novo usuário através de e-mail e senha, cadastrar e/ou iniciar sessão através de integração com conta do Google ou solicitar a redefinição de senha.

Uma vez iniciada a sessão, o usuário será redirecionado para a página inicial do sistema, caso já seja um usuário cadastrado, ou para a página de concluir cadastro, caso seja um usuário novo.

Figura 15 – Tela de Sessão de Usuário



A imagem mostra a interface de login do sistema OruData. No topo, há um ícone circular com um calendário rotulado 'DIET' e uma maçã vermelha. Abaixo, o título 'Conta OruData' é exibido em negrito. O formulário de login contém dois campos de entrada: 'E-Mail' e 'Senha'. Abaixo dos campos, há uma opção 'Lembrar meu login' com uma caixa de seleção marcada. Um botão arredondado com o texto 'Entrar com e-mail e senha' e uma seta à esquerda está centralizado. Abaixo do botão, a palavra 'Ou' é exibida, seguida por três ícones circulares: o 'G' do Google, um ícone de perfil de usuário e um ícone de chave. O fundo da tela é um tom claro de roxo.

Um usuário novo é impedido de navegar para qualquer outra página diferente da página de conclusão de cadastro, sendo automaticamente redirecionado de volta sempre que tentar navegar para outra parte da aplicação.

A interface de conclusão de cadastro de funcionário é simples e funciona em duas etapas, sendo a primeira solicitada os dados básicos e pessoais do usuário, como nome completo, um nome simples para exibição e o CPF. No campo de CPF é utilizado para verificar se já existe outro usuário cadastrado com mesmo CPF e evitar duplicação de cadastro. Também foi implementada uma validação de CPF para impedir um CPF inválido de ser usado em cadastro.

Após preencher estes dados, o usuário pressiona um botão de “Avançar” para preencher a próxima seção, que se refere aos dados do profissional, atualmente somente exibindo um campo para definição de qual a profissão do usuário. Com este campo informado, o usuário pode pressionar no botão de “Concluir” e será redirecionado à página inicial.

4.2.2 Cadastro de paciente

A partir de qualquer tela, o usuário em sessão pode acessar o Cadastro de Paciente através do botão de navegação intitulado “Pacientes”. No primeiro momento, a interface

Figura 16 – Campos de Cadastro de Usuário

Primeiro, vamos completar seu Cadastro:

Dados Pessoais

Nome Completo

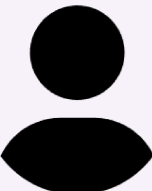
Fulano de Tal

Nome de exibição

Fulano

CPF

123.456.789-01



Dados Profissionais

Profissão

Nutricionista

exibida lista os pacientes cadastrados pelo profissional em sessão, oferecendo a opção de atualizar a lista, buscar por nome ou data de nascimento, navegar para um registro de paciente já cadastrado ou cadastrar um novo paciente. A escolha da primeira interface do cadastro ser a busca é deliberada: o profissional de saúde provavelmente irá muito mais vezes acessar um cadastro já realizado para atualizar um plano alimentar ou verificar alguma informação do que ele fará novos cadastros de pacientes. Ao pressionar o botão de “Novo Paciente” (ícone de pessoa e +), uma nova interface para o cadastro do paciente é exibida. A nova interface apresentada exibe campos de cadastro para o nome completo, CPF e Data de Nascimento do paciente, conforme visto na Figura 17. O campo de CPF é utilizado para validação de cadastro duplicado para o paciente, impedindo que o mesmo paciente seja cadastrado múltiplas vezes para o mesmo profissional.

A interface exibe um botão no canto superior direito para salvar o registro quando os campos forem preenchidos. Quando paciente sendo exibido já foi previamente salvo e está sendo editado, um segundo botão é exibido ao lado para desfazer as alterações e retornar ao estado original.

Após preenchidos os campos obrigatórios, o usuário pode salvar o paciente ou continuar e criar um plano alimentar imediatamente.

Um cartão para os Planos Alimentares é exibido abaixo dos dados básicos e contém um botão para a criação de novo plano. Caso esteja visualizando um paciente já cadastrado e com plano alimentar, o plano cadastrado é exibido ao invés disto.

Clicar no botão de criar novo plano expande o cartão para incluir novos campos. No início, são exibidos os campos de Início e Término do plano, além de um campo livre para observações adicionais que o profissional pode achar interessante incluir.

Figura 17 – Dados Básicos do Paciente

Cadastro de Paciente

Dados Básicos

Nome Completo
Ciclano da Silva

CPF
[REDACTED]

Nascimento
23/03/1992

Planos Alimentares ⓘ
Não existem planos cadastrados para o paciente.
[Criar novo Plano Alimentar](#)

Figura 18 – Edição de Plano Alimentar de Paciente

Plano Vigente

Início
07/04/2025 08:00

Término
04/05/2025 08:00

Normal

B

I

U

Observações adicionais sobre o plano.

Refeições

Nome	Horário	Alimentos	
			Adicionar refeição

Legenda

Bebidas

Sucos

Frutas

Leites e Derivados

Cereais Derivados

Cereais

Grãos

Vegetais

Proteínas

Na sequência, duas tabelas são exibidas: uma para cadastros das Refeições do paciente e outra para as Metas diárias a serem atingidas.

Figura 19 – Nova Refeição

Refeições

Nome	Horário	Alimentos
Nome	Horário Recomendado	
Café da Manhã	07:00	
Classificação	Qty.	
Bebidas	1	
Sucos	0	
Frutas	1	

Na primeira tabela é possível realizar o cadastro das refeições que serão utilizadas como base na geração das dietas do paciente. A tabela de refeições inicia vazia, mas o usuário pode cadastrar quantas refeições forem necessárias.

Ao clicar no botão de “Adicionar refeição”, novos campos são exibidos dentro da tabela, solicitando um nome e horário recomendado para a refeição nova. Abaixo, uma tabela é exibida com a lista de classificações de alimentos onde o usuário pode informar a quantidade de alimentos de cada classificação que são obrigatórias naquela refeição. Esta interface é exemplificada na Figura 19.

Uma vez adicionadas as refeições, elas são listadas de forma resumida utilizando ícones representativos de cada classificação de alimento seguido da quantidade para aquela refeição. Uma legenda é apresentada na parte inferior da tabela para auxiliar o entendimento do usuário.

Abaixo das refeições é exibida uma tabela contendo as metas diárias. As metas diárias são metas de máximo e mínimo que deverão ser respeitadas em cada dieta para cada nutriente. A tabela inicia pré-preenchida com todos os nutrientes e com valores pré-definidos, mas podem ser livremente pelo usuário.

Figura 20 – Refeições cadastradas

Refeições			
Nome	Horário	Alimentos	
Café da Manhã	07:00	 x1  x1  x1	 
Primeiro Lanche	10:00	 x1	 
Almoço	12:00	 x1  x1  x1  x1  x1	 
Segundo Lanche	15:00	 x1  x1	 
Jantar	18:30	 x1  x1  x1  x1	 
Ceia	20:30	 x1	 
 Adicionar refeição			
Legenda			
 Bebidas  Sucos  Frutas  Leites e Derivados  Cereais Derivados  Cereais  Grãos  Vegetais  Proteínas			

Figura 21 – Metas diárias

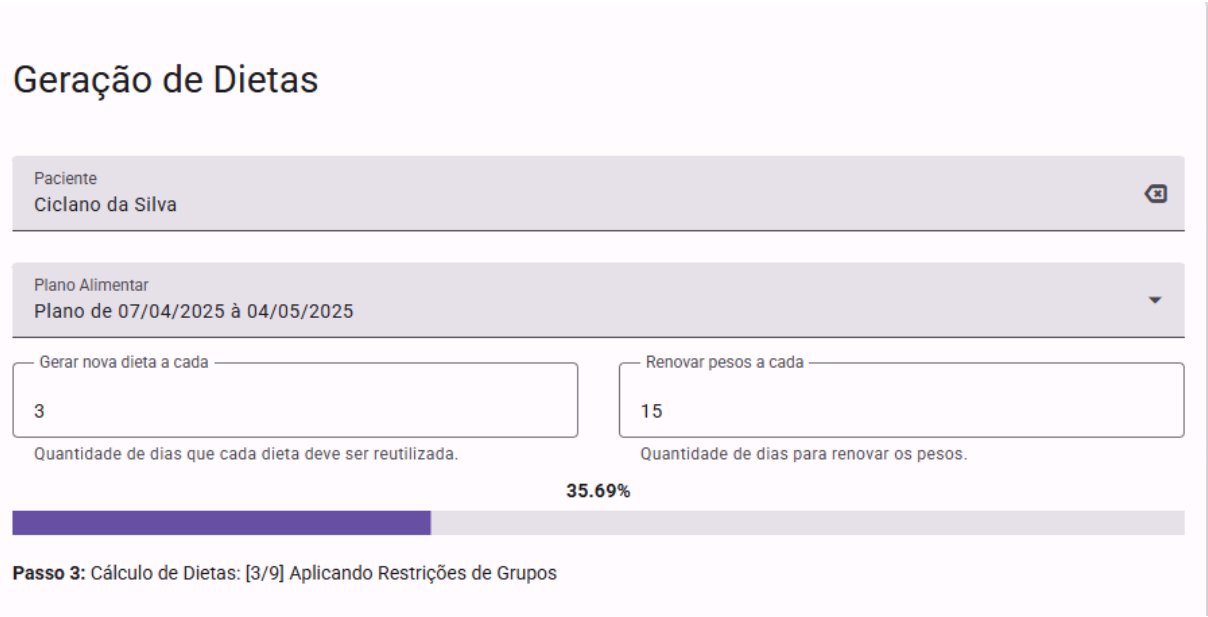


	Mínimo	Máximo	
Energia	1800.0		 
Proteína	75.0		 
Carboidrato	202.5		 
Fibra	30.0		 
Cálcio	850.0		 
Magnésio	400.0		 
Ferro	18.0		 
Fósforo	700.0		 
Sódio		2000.0	 
Zinco	15		 
Lipídios		60.0	 

4.2.3 Geração de dietas

Com o paciente cadastrado, o próximo passo é acessar a interface de geração de dietas através do botão de navegação intitulado “Dietas”. A interface exibida inicia com apenas um simples componente de seleção de paciente. É selecionado o paciente e plano alimentar cadastrado anteriormente. Como o modelo original do trabalho Soares (2023) gera apenas uma dieta, não há parâmetros base para os campos de validade da dieta ou renovação de pesos. Para o estudo, foi utilizada uma mesma dieta a cada 3 dias, renovando os pesos e penalidades a cada 15 dias. Após preenchidos os campos, é pressionado o botão de gerar dieta. Uma barra de progresso é exibida, atualizando automaticamente com base nos passos de geração realizados.

Figura 22 – Controle de Progresso de Geração de Dieta



Quando o processo é concluído com sucesso, um novo botão é exibido para visualização dos resultados. Pressionar o botão navega para a página de relatório. Os resultados encontrados são exibidos sequencialmente, seguidos pela tabela de resumo diário por nutrientes, conforme exemplificado na Figura 23.

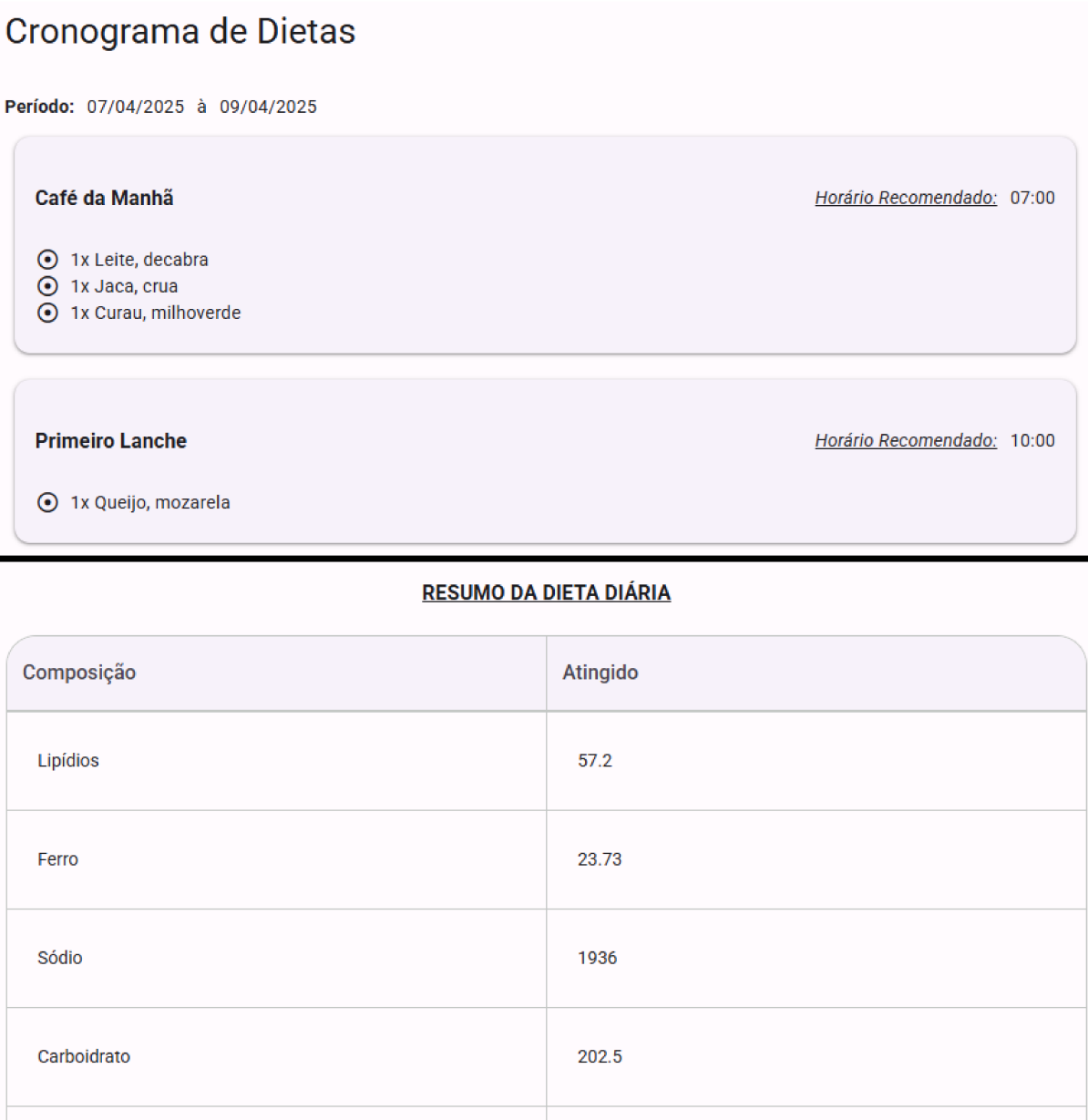
4.2.4 Comparação de resultados

A tabela a seguir faz uma comparação de resultados entre o que foi gerado no algoritmo puro e o que foi gerado através da aplicação.

Tabela 1 – Comparação de Resultados

Nutriente	Modelo	Aplicação
Lipídios	58.6	57,2
Ferro	19,33	23,73
Sódio	1419	1936
Carboidrato	202,5	202,5
Fósforo	1767	1868
Energia	1800	1800
Zinco	20.7	19,8
Proteína	130	126,3
Magnésio	511	451
Cálcio	1341	1436
Fibra	38,1	34,8

Figura 23 – Dietas Geradas



5 CONCLUSÃO

O presente trabalho teve como objetivo principal o desenvolvimento de uma plataforma *web* que integrasse um modelo de otimização multiobjetivo em dietas com interfaces visuais simples que permitissem uma configuração de objetivos individuais parametrizáveis com base nas necessidades individuais de cada paciente. O sistema foi projetado com o objetivo de facilitar a utilização destes modelos por profissionais de saúde, visando oferecer interfaces amigáveis para parametrização e utilização sem necessitar de conhecimento prévio de programação ou otimização. Através da plataforma os usuários tem acesso a criação de dietas individuais para períodos definíveis para cada caso e a visualização e impressão destes resultados.

A plataforma foi desenvolvida considerando boas práticas de organização de código, planejamento iterativo por meio de sprints e reutilização de componentes e bibliotecas previamente estruturadas. O uso de dependências próprias, em conjunto com dependências públicas, possibilitou uma evolução rápida da aplicação, mantendo uma padronização visual e funcional.

No estudo de caso realizado, foi demonstrado que a solução proposta alcançou os objetivos inicialmente propostos com sucesso. Os testes realizados confirmaram que o sistema é funcional e estável, validando sua aplicação prática utilizando contextos de uso baseados em trabalhos anteriores. Além disso, a abordagem permitiu observar o potencial do modelo teórico de [Soares \(2023\)](#) quando integrado a uma interface interativa.

No entanto, é importante ressaltar que ainda existem pontos de limitações a serem aprimorados na aplicação. Um destes pontos seria a disponibilização de uma forma de penalização inicial de alimentos de acordo com preferências do paciente, de forma a incluir desde o início situações de preferências ou mesmo limitações de saúde como impossibilidade de consumo de certas substâncias como glúten e a lactose. O algoritmo também ainda mantém como função objetivo a otimização energética e de carboidratos que foi utilizada no modelo para pacientes com diabetes, que é algo que poderia ser facilmente parametrizado no futuro. Ainda, existe uma limitação em identificar a refeição principal quando existem poucas refeições cadastradas no plano alimentar. Uma outra questão seria a substituição da base de dados de alimentos por uma maior, provavelmente baseada na TBCA, ou mesmo implementação de bases regionais ou por usuário.

De forma geral, todas as limitações apresentadas são de simples melhorias. Assim, esta aplicação se beneficiaria com um contínuo desenvolvimento, mas principalmente com um caso de uso concreto e participação de um profissional de saúde com críticas e sugestões concretas baseadas em seu trabalho diário.

Referências

- Anvil Works. **Anvil: Build Web Apps with Nothing but Python**. 2024. Acesso em: 11 outubro 2024. Disponível em: <<https://anvil.works/>>. Citado na página 15.
- BERTSEKAS, D. P.; PANG, J. S. **Linear Programming and Network Flows**. 2nd. ed. [S.l.]: Athena Scientific, 1999. Citado na página 13.
- BROWN, E. **Web Development with Node and Express: Leveraging the JavaScript stack**. 2. ed. Sebastopol, CA: O'Reilly Media, 2019. Citado na página 16.
- FLANAGAN, D. **JavaScript: The definitive guide**. 7. ed. Sebastopol, CA: O'Reilly Media, 2020. 1093 p. Citado na página 16.
- LUENBERGER, D. G. **Optimization by Vector Space Methods**. [S.l.]: Wiley, 1969. Citado na página 13.
- MARINS, F. A. S. Introdução à pesquisa operacional. **São Paulo: Cultura Acadêmica: Universidade Estadual Paulista**, 2011. Citado na página 13.
- RIES, E. **The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses**. [S.l.]: Crown Business, 2011. Citado na página 14.
- SCHWABER, K.; SUTHERLAND, J. **The Scrum Guide: The Definitive Guide to Scrum: The Rules of the Game**. [S.l.]: Scrum.org, 2020. Citado na página 15.
- SOARES, T. F. **Otimização Multiobjetivo Aplicada ao Planejamento de Dietas para Portadores de Diabetes**. 103 p. Dissertação (Mestrado) — Universidade Federal de Uberlândia, jan. 2023. Disponível em: <<https://repositorio.ufu.br/handle/123456789/41255>>. Citado 7 vezes nas páginas 12, 13, 14, 17, 37, 50 e 53.
- W3C. **Web Application Manifest**. 2025. Disponível em: <<https://www.w3.org/TR/appmanifest/>>. Acesso em: 22 abr 2025. Citado na página 16.
- WEB.DEV. **Progressive Web Apps**. 2015. Disponível em: <<https://web.dev/explore/progressive-web-apps>>. Acesso em: 31 out 2024. Citado na página 16.

Apêndices

APÊNDICE A – Classe BaseEnum

A seguir, é apresentada a pseudo-implementação da classe BaseEnum, utilizada como base para todas enumerações da aplicação.

Código A.1 – Pseudo-código para BaseEnum

```
class BaseEnum():
    """Classe base, efetivamente abstrata
    para Enums estenderem"""
    _key = None

    @property
    def key(self):
        return self._key

    @property
    def nome(self):
        return self._nome

    def get(self, attr, default=None):
        return getattr(self, attr, getattr(self, f'_{attr}', default))

    def __getitem__(self, attr):
        return getattr(self, attr, getattr(self, f'_{attr}'))

    def __repr__(self):
        rep = f"<{self.__class__.__name__}: {self.key}>"
        return rep

    @classmethod
    def by_key(cls, key:str):
        """Retorna a instancia baseada na chave informada."""

    @classmethod
    def key_list(cls):
        """Lista de chaves de cada valor do Enum"""

    @classmethod
    def items(cls):
        """Lista de tuplas(key, inst) de cada valor do Enum"""

    @classmethod
    def list(cls):
        """Lista de instancias dos valores do Enum"""
```

```
@classmethod
def map(cls, attr):
    """Lista de valores mapeados pelo atributo informado"""

@classmethod
def attr_key_tuple(cls, attr):
    """Lista de tuplas(attr, key) de cada valor do Enum"""
```

APÊNDICE B – Classe Entity

A seguir, é apresentada a implementação da classe `Entity`, utilizada como base para todas as entidades persistentes da aplicação.

Código B.1 – Classe Entity

```
class Entity:
    # Tem uma coluna Sequence que funciona como um "id"
    has_sequence = True
    # Entidade do banco ou um objeto Virtual
    persisted = True

    def __init__(self, row=None, display_value=None):

        if row is None:
            row = {}
        type_of = type(row)
        if type_of is dict:
            self._object_row = None
        else:
            self._object_row = row
        self._original_key_list = []
        self._changes = row if type_of is dict else {}
        if type_of is not dict:
            for k in row:
                self._original_key_list.append(k[0])
        if display_value:
            self.display_value = display_value

    @property
    def original_row(self):
        return self._object_row

    @property
    def row_changes(self):
        return self._changes

    @property
    def is_new(self):
        return self._object_row is None

    @property
    def is_empty(self):
        return self.is_new and not self.was_changed
```

```
@property
def was_changed(self):
    return len(self._changes) > 0

def reset_changes(self):
    self._changes.clear()

def merge(self):
    """Executa save ou update do AbstractService com base em ser uma
        nova entidade"""
    if not self.persisted:
        raise TypeError()
    if type(self) is Entity:
        raise TypeError()
    if not self.was_changed:
        raise ValueError()
    if self.is_new:
        return type(self)(anvil.server.call(f'post{self.cls_name()}',
            , self.row_changes), self.display_value)
    else:
        return type(self)(anvil.server.call(f'put{self.cls_name()}',
            *self.get_tuple()), self.display_value)

def get(self, key, default=None):
    try:
        return self._changes[key]
    except KeyError:
        try:
            return self._object_row[key]
        except (TypeError, NoSuchColumnError):
            return default

def __getitem__(self, attr):
    """Primeiro, tenta buscar um atributo personalizado.
    Se nao existir, busca de changes.
    Se nao foi alterado, busca do objeto original
    Se tudo falhar, raise KeyError"""
    try:
        # Checo se eh um atributo personalizado de uma subclasse
        return getattr(self, attr)
    except AttributeError:
        # Retorno o valor nos changes, ou se nao tiver, do original
        try:
            return self._changes[attr]
        except KeyError:
            try:
```

```
        return self._object_row[attr]
    except (TypeError, NoSuchColumnError):
        return None

def __setitem__(self, attr, value):
    self._changes.pop(attr, None)
    value_instance = None
    if isinstance(value, Entity):
        value_instance = value
        value = value_instance.original_row or value_instance.
            row_changes
    try:
        old_value = self._object_row[attr]
        if value != old_value:
            self._changes[attr] = value
            if value_instance:
                setattr(self, attr, value_instance)
    except (TypeError, NoSuchColumnError):
        self._changes[attr] = value
        if value_instance:
            setattr(self, attr, value_instance)
```