iFood de Serviços Domésticos

1. Introdução

Objetivo da Documentação

Este documento descreve a arquitetura, as tecnologias e a organização do código do sistema "iFood de Serviços Domésticos", uma plataforma *marketplace* que conecta Clientes que necessitam de serviços domésticos (limpeza, reparos) a Prestadores de Serviços qualificados. O objetivo é fornecer um guia para desenvolvedores, arquitetos e *DevOps*.

Visão Geral do Sistema

O sistema é uma plataforma de agendamento e transação de serviços, focada em estabelecer uma conexão direta e transparente entre usuários finais e profissionais. Ele gerencia o ciclo de vida completo do serviço, desde o cadastro dos usuários e a oferta de serviços até a criação do pedido, o acompanhamento de *status* e a avaliação do prestador.

Principais Funcionalidades

As funcionalidades baseiam-se nos atores **Cliente** e **Prestador**, conforme o Diagrama de Casos de Uso:

- **Registro e Perfil:** Cadastro de Cliente/Prestador e atualização de perfil.
- **Gestão de Serviços:** O Prestador pode adicionar serviços e o Cliente pode solicitá-los.
- **Fluxo de Pedido:** Criação de pedido, aceitação pelo Prestador e atualização de *status*.
- **Avaliação:** Cliente pode analisar o Prestador.
- **Transações:** Processamento simulado de pagamento e emissão de recibo.

2. Arquitetura do Sistema

Tipo de Arquitetura

Para a fase de protótipo e demonstração de POO (Produto Mínimo Viável - MVP), adotou-se uma **Arquitetura Monolítica** baseada em Python.

O sistema monolítico é contido em uma única aplicação que engloba a Interface de Usuário (GUI), a Lógica de Negócio e a Simulação de Persistência.

Descrição Geral

O sistema é estruturado em três camadas principais, rigidamente separadas pelos arquivos Python:

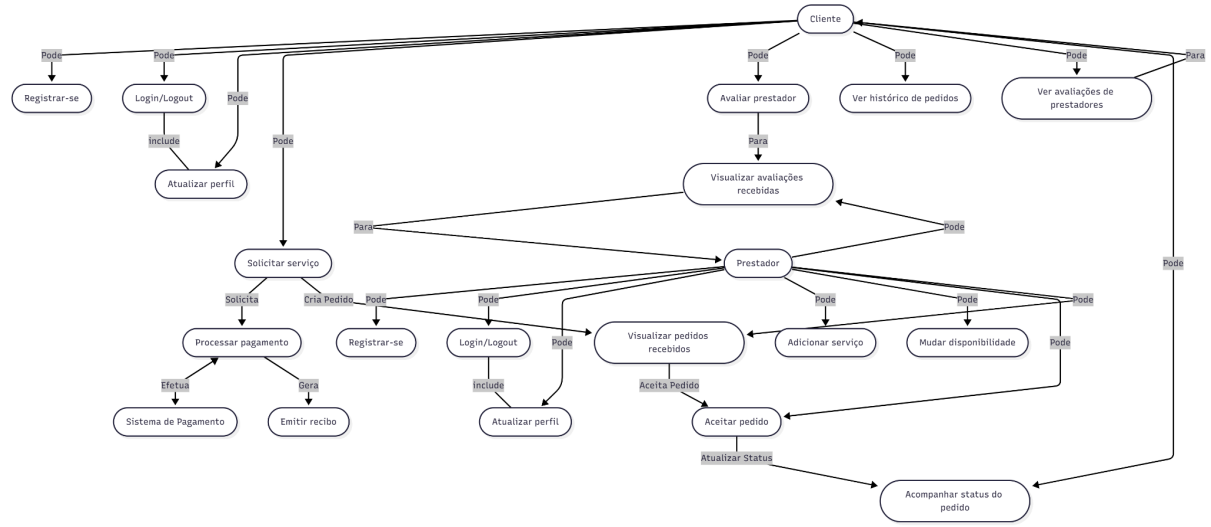
1. **Interface de Usuário (Apresentação):** Implementada via gui_app.py utilizando a biblioteca **Flet**.
2. **Lógica de Negócio:** Contida nas classes de POO (pessoa.py, servico.py, pedido.py) e orquestrada pela gui_app.py.

3. **Persistência (Simulada):** Gerenciada pela classe Repositorio, que armazena os objetos em memória (listas e dicionários).

Apesar de ser monolítica, a aplicação está preparada para uma transição futura para uma arquitetura de Microsserviços, como a sugerida no exemplo, bastando substituir a lógica de persistência e a chamada de métodos diretos por chamadas HTTP para serviços REST.

Diagrama de Casos de Uso

Abaixo está o diagrama que define os atores e as interações do sistema:



3. Tecnologias e Ferramentas Utilizadas

Categoria	**Tecnologia**	**Uso**
Linguagem Principal	Python 3.10+	Lógica de negócio e desenvolvimento da GUI.
Framework GUI	Flet	Criação da Interface Gráfica de Usuário (multi-plataforma).
Modelagem de Dados	Pydantic (Implícito)	Embora o Flet não exija, as classes de POO são baseadas na estrutura Pydantic definida em etapas anteriores, garantindo tipagem forte.

Princípios	Programação Orientada a Objetos (POO)	Encapsulamento, Herança, Polimorfismo e Abstração.
Banco de Dados	Memória (Classe Repositorio)	Persistência temporária para o MVP.

4. Estrutura de Pastas e Organização do Código

O projeto é organizado seguindo o padrão de modularidade de POO, onde cada arquivo Python representa um módulo ou uma camada específica do sistema.

```
/projeto-ifood-servicos
├── gui_app.py        # Interface de Usuário (Flet) - Camada de Apresentação
├── pessoa.py         # Classes Cliente e Prestador - Modelos de Usuário
├── servico.py        # Classes Servico e ServicoEspecial - Modelos de Serviço
├── pedido.py         # Classe Pedido - Lógica de Agendamento e Cálculo
├── repositorio.py    # Classe Repositorio - Simulação de Persistência
└── README.md
```

5. Padrões e Convenções

Padrões de Código

- **Nomenclatura:** Variáveis internas de classes são prefixadas com _ (ex: self._nome), aplicando o princípio de encapsulamento.
- **Convenção de Classes:** Classes e métodos seguem as boas práticas de POO, com uso de herança (Cliente e Prestador de Pessoa) e polimorfismo (calcular_preco em ServicoEspecial).

6. Integrações Externas (Sugestões para o Futuro)

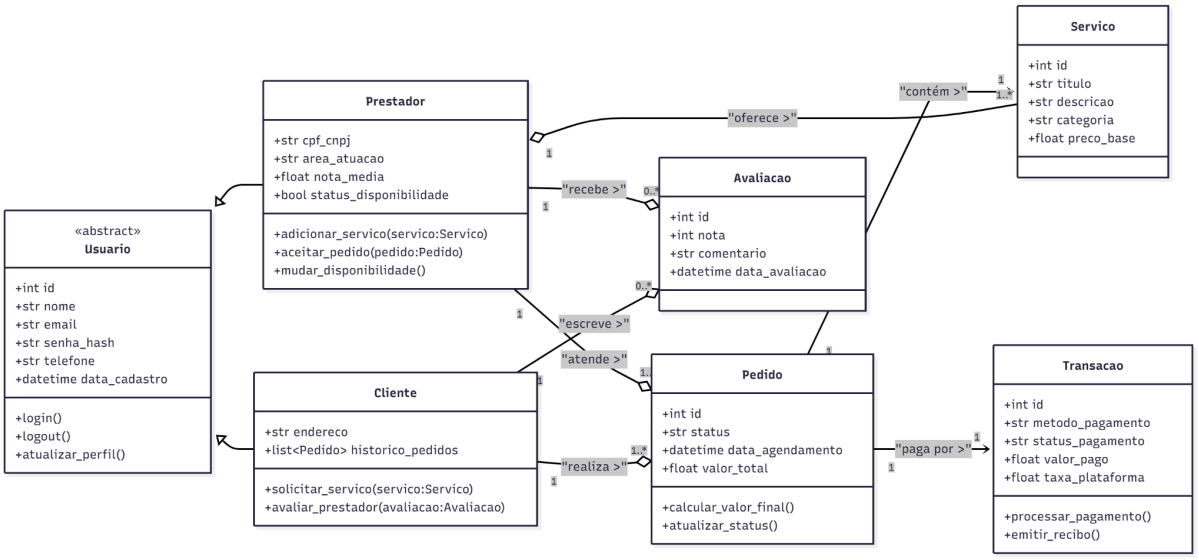
Embora a versão atual seja um MVP em Python puro, o Diagrama de Casos de Uso e o Diagrama de Classes preveem integrações futuras cruciais:

- **Sistema de Pagamento:** O caso de uso **Processar pagamento** e a classe **Transacao** (no Diagrama de Classes) sugerem a integração com APIs de pagamento (Ex: Stripe, PagSeguro).
- **Geolocalização:** A presença do atributo endereco na classe Cliente implica a necessidade futura de uma API de Geolocalização (Ex: Google Maps API) para pareamento Cliente-Prestador por proximidade.

7. Banco de Dados

Modelagem de Dados

A modelagem de dados é definida pelo Diagrama de Classes a seguir, que serve como base para a criação de tabelas em um SGBD relacional (como PostgreSQL) ou coleções em um NoSQL (como MongoDB)[10101010].



Conexões e Configurações

- **MVP Atual:** O objeto Repositorio mantém todos os dados em memória RAM. Não há *string* de conexão externa.
- **Próxima Etapa:** É recomendada a substituição do Repositorio por uma camada de acesso a dados que use um ORM (ex: SQLAlchemy) para se conectar a um banco de dados persistente, com a *string* de conexão armazenada em variáveis de ambiente.

8. Processo de Build e Deploy

Build

Como o projeto usa Python e Flet, a "construção" (build) é simplificada:

- **Geração de Executável (Desktop):** O Flet permite o *build* em *binários* nativos para Desktop (Windows, macOS, Linux) usando ferramentas como flet build.
- **Web/Mobile:** O Flet também permite a compilação para Web e Mobile (iOS/Android).

Deploy

A aplicação é executada diretamente pelo interpretador Python:

Bash
python gui_app.py

Para produção web, o aplicativo Flet seria hospedado em um contêiner Docker (ou serviço *serverless*), expondo a aplicação via navegador.

9. Monitoramento e Logs

Loggers

No MVP, o *feedback* do sistema é dado na GUI através do componente feedback_message em gui_app.py. Para um sistema em produção, é necessário implementar um *logger* como logging (nativo do Python) ou Winston para microsserviços Node.js.

10. Considerações de Segurança

As seguintes medidas de segurança devem ser implementadas em futuras versões:

- **Controle de Acesso (Autenticação e Autorização):** O fluxo **Login/Logout** deve ser implementado usando *tokens* JWT, garantindo que apenas usuários autenticados possam realizar ações (ex: Prestadores adicionarem serviços).
- **Criptografia de Dados Sensíveis:** Senhas (senha_hash em Usuario) devem ser armazenadas de forma irreversível (Ex: bcrypt).
- **HTTPS:** Uso obrigatório de HTTPS para comunicação segura em ambiente web.