Documentação Técnica

Esta documentação detalha a arquitetura, o design e a implementação do Sistema de Gerenciamento de Estoque de Doações.

1. Arquitetura do Sistema

O sistema segue uma arquitetura em camadas para promover a modularidade, a manutenibilidade e a escalabilidade. As principais camadas são:

- **Camada de Dados (Data Layer):** Responsável pela persistência e recuperação de dados. Utiliza SQLite como banco de dados e o módulo `sqlite3` do Python para interagir com ele. O arquivo `database.py` gerencia a conexão e a criação das tabelas, enquanto as classes em `models.py` encapsulam as operações CRUD para cada entidade.
- **Camada de Lógica de Negócios (Business Logic Layer):** Contida nas classes de modelo em `models.py`, esta camada implementa as regras de negócio, como a lógica para buscar itens próximos do vencimento (`get_expiring_items`) e o agrupamento de estoque (`get_grouped_stock`). Ela interage diretamente com a camada de dados para realizar as operações necessárias.
- **Camada de Apresentação (Presentation Layer - GUI):** Implementada com a biblioteca `Tkinter` em `app.py`, esta camada é responsável pela interface do usuário. Ela exibe os dados recuperados da camada de lógica de negócios e captura as entradas do usuário, que são então passadas para a camada de lógica de negócios para processamento. Inclui validações de entrada e funcionalidades de pesquisa para melhorar a usabilidade.

2. Esquema do Banco de Dados

O banco de dados `estoque_doacoes.db` é um banco de dados SQLite e possui as seguintes tabelas:

- **`doadores`**
  - `id_doador` (INTEGER PRIMARY KEY AUTOINCREMENT)

    - `nome` (TEXT NOT NULL)
    - `telefone` (TEXT)
    - `email` (TEXT)
    - `endereco` (TEXT)
- **`beneficiarios`**
    - `id_beneficiario` (INTEGER PRIMARY KEY AUTOINCREMENT)
    - `nome` (TEXT NOT NULL)
    - `telefone` (TEXT)
    - `email` (TEXT)
    - `endereco` (TEXT)
    - `alimento_necessidade_1` (TEXT)
    - `alimento_necessidade_2` (TEXT)
    - `alimento_necessidade_3` (TEXT)
- **`itens`**
    - `id_item` (INTEGER PRIMARY KEY AUTOINCREMENT)
    - `nome_item` (TEXT NOT NULL)
    - `marca` (TEXT)
    - `unidade` (TEXT NOT NULL)
- **`doacoes_recebidas`**
    - `id_doacao_recebida` (INTEGER PRIMARY KEY AUTOINCREMENT)
    - `id_doador` (INTEGER NOT NULL, FOREIGN KEY para `doadores.id_doador`)
    - `id_item` (INTEGER NOT NULL, FOREIGN KEY para `itens.id_item`)
    - `quantidade` (REAL NOT NULL)
    - `data_recebimento` (TEXT NOT NULL, formato YYYY-MM-DD)
    - `data_validade` (TEXT NOT NULL, formato YYYY-MM-DD)
- **`doacoes_realizadas`**

- `id_doacao_realizada` (INTEGER PRIMARY KEY AUTOINCREMENT)
- `id_beneficiario` (INTEGER NOT NULL, FOREIGN KEY para `beneficiarios.id_beneficiario`)
- `id_item` (INTEGER NOT NULL, FOREIGN KEY para `itens.id_item`)
- `quantidade` (REAL NOT NULL)
- `data_doacao` (TEXT NOT NULL, formato YYYY-MM-DD)

3. Módulos e Classes

`database.py`

- `create_connection(db_file)`: Estabelece e retorna uma conexão com o banco de dados SQLite. Configura `row_factory` para `sqlite3.Row` para permitir acesso às colunas por nome.
- `create_tables(conn)`: Cria todas as tabelas necessárias no banco de dados, se elas ainda não existirem. Inclui a adição dos novos campos na tabela `beneficiarios`.

`models.py`

- **`BaseModel`**
  - Classe base para todas as entidades do banco de dados, fornecendo métodos CRUD genéricos.
  - `__init__(self, table_name, conn)`: Inicializa o modelo com o nome da tabela e a conexão do banco de dados.
  - `save(self, data)`: Insere um novo registro na tabela.
  - `get_all(self)`: Retorna todos os registros da tabela.
  - `get_by_id(self, id_value)`: Retorna um registro específico pelo seu ID.
  - `update(self, id_value, data)`: Atualiza um registro existente.
  - `delete(self, id_value)`: Exclui um registro.
- **`Doador(BaseModel)`**
  - Gerencia operações para a tabela `doadores`.

- **`Beneficiario(BaseModel)`**
    - Gerencia operações para a tabela `beneficiarios`. Inclui a manipulação dos campos `alimento_necessidade_1`, `alimento_necessidade_2`, `alimento_necessidade_3`.
- **`Item(BaseModel)`**
    - Gerencia operações para a tabela `itens`.
- **`DoacaoRecebida(BaseModel)`**
    - Gerencia operações para a tabela `doacoes_recebidas`.
    - `get_all_with_details(self)`: Retorna todas as doações recebidas com detalhes do doador e do item (usando JOINs).
    - `get_expiring_items(self, days_threshold)`: Retorna itens de doações recebidas que vencem dentro de um número especificado de dias a partir da data atual.
    - `get_grouped_stock(self)`: Retorna o estoque atual agrupado por tipo de alimento e unidade, somando as quantidades.
    - `get_stock_by_item(self, item_id)`: Retorna a quantidade total em estoque para um item específico.
    - `get_all_stock_items(self)`: Retorna todos os itens que já foram recebidos e ainda têm estoque positivo.
- **`DoacaoRealizada(BaseModel)`**
    - Gerencia operações para a tabela `doacoes_realizadas`.
    - `get_all_with_details(self)`: Retorna todas as doações realizadas com detalhes do beneficiário e do item (usando JOINs).

`app.py`

- **`EstoqueApp(tk.Tk)`**
    - Classe principal da aplicação Tkinter.
    - `__init__(self)`: Configura a janela principal, as abas (Notebook) e inicializa os modelos de dados.

- `validate_phone(self, phone)`: Função de validação para o formato de telefone (DDD+número).
- `validate_email(self, email)`: Função de validação para o formato de e-mail.
- Métodos para cada aba (ex: `create_donor_tab`, `create_beneficiary_tab`, `create_received_donation_tab`, `create_distributed_donation_tab`, `create_stock_tab`, `create_entries_tab`, `create_exits_tab`, `create_alerts_tab`).
- Funções de callback para os botões e eventos da GUI, que interagem com os métodos das classes de modelo para realizar as operações no banco de dados.
- Implementação de `Combobox` com funcionalidade de pesquisa para seleção de doadores, beneficiários, tipos de alimento e marcas.
- Lógica para preencher Treeviews e listas suspensas com dados do banco de dados.
- Implementação do sistema de alerta de vencimento, atualizando a aba de alertas periodicamente para itens com 30 dias ou menos para vencer.
- Validação de estoque antes de registrar doações realizadas.

4. Fluxo de Dados

1. **Inicialização:** `app.py` cria a conexão com o banco de dados (`database.py`) e inicializa as classes de modelo (`models.py`), passando a conexão para elas.
2. **Interação do Usuário:** O usuário interage com a GUI (`app.py`), preenchendo formulários e clicando em botões.
3. **Processamento da GUI:** As funções de callback em `app.py` coletam os dados da interface, aplicam validações (telefone, e-mail, data, quantidade, estoque) e chamam os métodos apropriados nas classes de modelo (`models.py`).
4. **Lógica de Negócios e Persistência:** As classes de modelo processam os dados, aplicam regras de negócio (como a verificação de estoque) e interagem com o banco de dados (via `self.conn` que é a conexão passada para elas) para salvar, atualizar, deletar ou buscar informações.

5. **Atualização da GUI:** Após a operação do banco de dados, a GUI é atualizada para refletir as mudanças (ex: recarregar listas, exibir mensagens de sucesso/erro, atualizar comboboxes).

5. Considerações de Segurança e Robustez

- **SQLite:** Por ser um banco de dados baseado em arquivo, é simples de usar e ideal para aplicações de pequeno porte. No entanto, não é adequado para ambientes multiusuário concorrentes sem um mecanismo de bloqueio de arquivo adequado.
- **Tratamento de Erros:** As operações de banco de dados em `models.py` incluem blocos `try-except` para capturar `sqlite3.Error` e imprimir mensagens de erro, tornando o sistema mais robusto contra falhas inesperadas no banco de dados.
- **Validação de Entrada:** A validação de entrada de dados é feita na camada da GUI e nos modelos para garantir a integridade dos dados antes da persistência, incluindo validações de formato para telefone, e-mail e datas, além de validação de estoque.
- **Gerenciamento de Conexões:** A conexão com o banco de dados é gerenciada de forma a ser aberta e fechada para cada operação na `BaseModel` (quando não é uma conexão passada externamente), ou mantida aberta para o ciclo de vida do teste, garantindo que os recursos sejam liberados corretamente.