Manual Técnico do Sistema de Almoxarifado

Documentação Completa com Análise Linha por Linha

Desenvolvedor: Lourenço

Instituição: Fundação Liberato

Destinado a: Alunos e Funcionários

Versão: 1.0

Índice

---

1. Introdução

Este manual técnico fornece uma visão detalhada do Sistema de Almoxarifado, uma aplicação de desktop desenvolvida em Python com a biblioteca PyQt6 para a interface gráfica e MySQL como banco de dados. O objetivo deste documento é servir como um guia completo para o entendimento, manutenção e futuras expansões do sistema, sendo acessível para desenvolvedores e estudantes com conhecimentos básicos em programação.

O sistema foi projetado para gerenciar o empréstimo e a devolução de componentes eletrônicos aos alunos da Fundação Liberato, oferecendo uma interface intuitiva para pesquisa de alunos, consulta de componentes e registro de movimentações.

1.1. Componentes do Sistema

O sistema é composto por:

- **Aplicação Principal (`app.py`):** Interface gráfica e lógica de negócio
- **Módulo de Configuração (`Config.py`):** Gerenciamento de conexão com banco de dados
- **Script de Processamento de Alunos (`processar_alunos.py`):** Preparação de dados CSV
- **Script de Atualização de Fotos (`atualizar_fotos.py`):** Importação automática de fotos

---

2. Arquitetura do Sistema

O sistema é composto por três camadas principais:

Interface do Usuário (Frontend): Desenvolvida com a biblioteca PyQt6, responsável por toda a interação visual com o usuário. A interface é organizada em janelas, abas e componentes interativos que permitem realizar todas as operações do sistema.

Lógica de Negócio (Backend): Implementada em Python, esta camada contém as regras de negócio, o processamento das solicitações do usuário e a comunicação com o banco de dados. É o núcleo do sistema, onde as operações de empréstimo, devolução e consulta são executadas.

Banco de Dados (Persistência): Utiliza o MySQL para armazenar de forma persistente todos os dados da aplicação, incluindo informações de alunos, componentes e registros de empréstimos. A conexão é gerenciada por um módulo de configuração dedicado.

---

3. Estrutura do Banco de Dados

O banco de dados é o coração do sistema, armazenando todas as informações de forma organizada e relacional. Ele é composto por três tabelas principais: `alunos` , `componentes` e `emprestimos` .

3.1. Tabela `alunos`

Armazena as informações cadastrais dos alunos.

Coluna	Tipo de Dado	Descrição
`id`	INT	Identificador único do aluno (Chave Primária, Auto-Incremento)
`matricula`	INT	Número de matrícula do aluno
`nome`	VARCHAR	Nome completo do aluno
`email`	VARCHAR	Endereço de e-mail do aluno
`turma`	INT	Código da turma do aluno
`foto`	LONGBLOB	Foto do aluno em formato binário

Observação Importante: A coluna `id` deve ser configurada como **PRIMARY KEY** e **AUTO_INCREMENT** para garantir a unicidade e geração automática de identificadores.

3.2. Tabela `componentes`

Armazena o catálogo de componentes disponíveis para empréstimo.

Coluna	Tipo de Dado	Descrição
`id`	INT	Identificador único do componente (Chave Primária, Auto-Incremento)
`nome`	VARCHAR	Nome ou descrição do componente

Observação Importante: A coluna `id` deve ser configurada como **PRIMARY KEY** e **AUTO_INCREMENT**.

3.3. Tabela `emprestimos`

Registra todas as operações de empréstimo de componentes, vinculando um aluno a um componente.

Coluna	Tipo de Dado	Descrição
`id`	INT	Identificador único do empréstimo (Chave Primária, Auto-Incremento)
`aluno_id`	INT	Chave estrangeira que referencia o `id` da tabela `alunos`
`componente_id`	INT	Chave estrangeira que referencia o `id` da tabela `componentes`
`quantidade`	INT	Quantidade do componente emprestada
`data_emp`	DATETIME	Data e hora em que o empréstimo foi realizado
`status`	VARCHAR	Status atual do empréstimo ('Emprestado' ou 'Devolvido')
`data_dev`	DATETIME	Data e hora em que a devolução foi registrada (pode ser NULO)

Observação Importante: A coluna `id` deve ser configurada como **PRIMARY KEY** e **AUTO_INCREMENT**.

---

4. Guia de Instalação e Configuração

Para executar o sistema em um novo ambiente, siga os passos abaixo. Este guia assume que você está em um sistema operacional Windows, mas os passos são similares para macOS e Linux.

4.1. Instalação das Ferramentas

Passo 1: Instalar Python 3.11

- Baixe o instalador em python.org
- Durante a instalação, **marque a opção "Add Python to PATH"**
- Verifique a instalação abrindo o terminal e digitando: `python --version`

Passo 2: Instalar VS Code

- Baixe o editor de código em code.visualstudio.com
- Siga as instruções de instalação padrão
- Instale a extensão "Python" da Microsoft para melhor suporte

Passo 3: Instalar DBeaver

- Baixe o gerenciador de banco de dados em dbeaver.io
- Instale a edição "Community"
- Configure a conexão com o servidor MySQL da instituição

4.2. Configuração do Ambiente

Passo 1: Instalar Dependências Python

Abra um terminal ou prompt de comando (CMD) e execute os seguintes comandos:

Bash

```
pip install mysql-connector-python
pip install PyQt6
pip install pytz
pip install pandas
pip install requests
```

Passo 2: Configurar a Conexão no DBeaver

- Abra o DBeaver
- Clique em "Nova Conexão" e selecione "MySQL"
- Insira as credenciais fornecidas no arquivo `Config.py`:
  - Host: `10.233.43.215`
  - Usuário: `troadmin`
  - Senha: (conforme arquivo Config.py)
  - Banco de dados: `site_eletronicaX` ou `site_eletronicaX_teste`
- Teste a conexão antes de salvar

Passo 3: Criar e Configurar as Tabelas

Execute os seguintes comandos SQL no DBeaver para criar as tabelas:

SQL

```
-- Criar tabela alunos
CREATE TABLE alunos (
    id INT NOT NULL AUTO_INCREMENT,
    matricula INT,
    nome VARCHAR(255),
    email VARCHAR(255),
    turma INT,
    foto LONGBLOB,
    PRIMARY KEY (id)
```

```
);

-- Criar tabela componentes
CREATE TABLE componentes (
    id INT NOT NULL AUTO_INCREMENT,
    nome VARCHAR(255),
    PRIMARY KEY (id)
);

-- Criar tabela emprestimos
CREATE TABLE emprestimos (
    id INT NOT NULL AUTO_INCREMENT,
    aluno_id INT,
    componente_id INT,
    quantidade INT,
    data_emp DATETIME,
    status VARCHAR(50),
    data_dev DATETIME,
    PRIMARY KEY (id),
    FOREIGN KEY (aluno_id) REFERENCES alunos(id),
    FOREIGN KEY (componente_id) REFERENCES componentes(id)
);
```

Para um guia visual detalhado de como configurar as chaves primárias no DBeaver, consulte o **Apêndice A**.

4.3. Executar o Sistema

Passo 1: Abrir o Projeto no VS Code

- Inicie o VS Code
- Vá em "File" > "Open Folder"
- Selecione a pasta onde os arquivos `app.py` e `Config.py` estão localizados

Passo 2: Executar o Arquivo Principal

- Abra o arquivo `app.py`
- Pressione `F5` ou clique no ícone de "Run" (▶️)
- A janela do Sistema de Almoxarifado deverá aparecer na tela

---

5. Análise Detalhada do Código-Fonte

Esta seção apresenta uma análise linha por linha de todos os arquivos do sistema, explicando o propósito e funcionamento de cada trecho de código.

5.1. Arquivo `Config.py` - Análise Linha por Linha

O arquivo `Config.py` é responsável por gerenciar a conexão com o banco de dados MySQL, permitindo alternar entre ambientes de produção e teste.

```python
import mysql.connector
from mysql.connector import Error
```

Linha 1: Importa o módulo `mysql.connector`, que é a biblioteca oficial para conectar Python ao MySQL. Esta biblioteca fornece todas as funções necessárias para estabelecer conexões, executar queries e manipular resultados.

Linha 2: Importa especificamente a classe `Error` do módulo `mysql.connector`. Esta classe é usada para capturar e tratar exceções relacionadas ao banco de dados, como falhas de conexão, erros de sintaxe SQL ou problemas de autenticação.

```python
class ConfigBanco:
    _modo = "produção"
```

Linha 4: Define a classe `ConfigBanco`, que encapsula toda a lógica de configuração do banco de dados. O uso de uma classe permite organizar métodos relacionados e manter o estado da configuração.

Linha 5: Declara uma variável de classe `_modo` com valor inicial `"produção"`. O underscore no início (`_`) é uma convenção Python que indica que esta variável é "privada" (embora não seja tecnicamente privada). Esta variável controla qual banco de dados será utilizado.

```python
    @staticmethod
    def usar_banco_producao():
        ConfigBanco._modo = "produção"
```

Linha 7: O decorador `@staticmethod` indica que este método não precisa de uma instância da classe para ser chamado. Pode ser invocado diretamente como `ConfigBanco.usar_banco_producao()`.

Linha 8: Define o método `usar_banco_producao()`, que não recebe parâmetros além do decorador.

Linha 9: Altera o valor da variável de classe `_modo` para `"produção"`. Quando este método é chamado, o sistema passa a utilizar o banco de dados de produção.

Python

```
    @staticmethod
    def usar_banco_teste():
        ConfigBanco._modo = "teste"
```

Linha 11: Outro método estático, seguindo o mesmo padrão do anterior.

Linha 12: Define o método `usar_banco_teste()`.

Linha 13: Altera o valor de `_modo` para `"teste"`, fazendo com que o sistema utilize o banco de dados de teste.

Python

```
    @staticmethod
    def banco_atual():
        return "Produção" if ConfigBanco._modo == "produção" else "Teste"
```

Linha 15: Mais um método estático.

Linha 16: Define o método `banco_atual()`, que retorna uma string indicando qual banco está em uso.

Linha 17: Utiliza um operador ternário (if-else em linha única) para retornar `"Produção"` se `_modo` for igual a `"produção"`, caso contrário retorna `"Teste"`. Este método é útil para exibir na interface qual banco está sendo utilizado.

Python

```
    @staticmethod
    def obter_conexao():
        try:
```

Linha 19: Define o método mais importante da classe, `obter_conexao()`, que retorna um objeto de conexão com o banco de dados.

Linha 20: Inicia um bloco `try` para capturar possíveis exceções durante a tentativa de conexão.

Python

```
            if ConfigBanco._modo == "produção":
                return mysql.connector.connect(
                    host='10.233.43.215',
                    user='troadmin',
                    password='eleTROn1c_flop3#W!w',
```

```
                database='site_eletronicaX',
                charset='utf8mb4',
                connection_timeout=30
            )
```

Linha 21: Verifica se o modo atual é `"produção"`.

Linha 22: Se for produção, chama o método `connect()` do `mysql.connector` para estabelecer uma conexão.

Linha 23: `host='10.233.43.215'` - Define o endereço IP do servidor MySQL na rede intranet da Fundação Liberato.

Linha 24: `user='troadmin'` - Define o nome de usuário para autenticação no banco de dados.

Linha 25: `password='eleTROn1c_flop3#W!w'` - Define a senha do usuário. **Nota de segurança:** Em produção real, esta senha deveria estar em variáveis de ambiente ou arquivo de configuração criptografado.

Linha 26: `database='site_eletronicaX'` - Define o nome do banco de dados de produção.

Linha 27: `charset='utf8mb4'` - Define a codificação de caracteres como UTF-8 com suporte completo a emojis e caracteres especiais.

Linha 28: `connection_timeout=30` - Define um tempo limite de 30 segundos para estabelecer a conexão. Se o servidor não responder neste período, uma exceção será lançada.

Python

```
        else:
            return mysql.connector.connect(
                host='10.233.43.215',
                user='troadmin',
                password='eleTROn1c_flop3#W!w',
                database='site_eletronicaX_teste',
                charset='utf8mb4',
                connection_timeout=30
            )
```

Linha 30: Se o modo não for `"produção"` (ou seja, for `"teste"`), executa este bloco.

Linhas 31-38: Estabelece conexão com o banco de teste. Os parâmetros são idênticos ao banco de produção, exceto o nome do banco de dados (`site_eletronicaX_teste`). Isso permite testar alterações sem afetar dados reais.

Python

```
    except Error as e:
        print(f"Erro ao conectar ao banco: {e}")
```

```
            return None
```

Linha 39: Captura qualquer exceção do tipo `Error` que ocorra durante a tentativa de conexão e a armazena na variável `e`.

Linha 40: Imprime uma mensagem de erro no console, incluindo a descrição do erro capturado. O `f` antes da string indica uma f-string, que permite interpolar variáveis diretamente no texto.

Linha 41: Retorna `None` para indicar que a conexão falhou. O código que chamar este método deve verificar se o retorno é `None` antes de tentar usar a conexão.

5.2. Arquivo `app.py` - Análise Linha por Linha

O arquivo `app.py` contém toda a lógica da aplicação principal, incluindo a interface gráfica e as operações de banco de dados.

5.2.1. Importações e Configurações Iniciais

Python

```python
import sys
import mysql.connector
from mysql.connector import Error
```

Linha 1: Importa o módulo `sys`, que fornece acesso a variáveis e funções relacionadas ao interpretador Python. É usado principalmente para `sys.exit()` ao final do programa.

Linha 2: Importa o conector MySQL para permitir operações de banco de dados diretamente do código da aplicação.

Linha 3: Importa a classe `Error` para tratamento de exceções de banco de dados.

Python

```python
from PyQt6.QtWidgets import (
    QApplication, QWidget, QGridLayout, QVBoxLayout, QHBoxLayout,
    QGroupBox, QLabel, QPushButton, QLineEdit, QListWidget,
    QMessageBox, QScrollArea, QFrame, QCheckBox,
    QSplitter, QSizePolicy, QDialog, QTabWidget
)
```

Linhas 4-10: Importam diversos widgets (componentes de interface) do módulo `QtWidgets` do PyQt6:

- `QApplication`: Gerencia o fluxo de controle e configurações principais da aplicação GUI

- `QWidget` : Classe base para todos os objetos de interface do usuário
- `QGridLayout` , `QVBoxLayout` , `QHBoxLayout` : Gerenciadores de layout para organizar widgets
- `QGroupBox` : Caixa de grupo com título para agrupar widgets relacionados
- `QLabel` : Exibe texto ou imagens não editáveis
- `QPushButton` : Botão clicável
- `QLineEdit` : Campo de entrada de texto de uma linha
- `QListWidget` : Lista de itens selecionáveis
- `QMessageBox` : Caixas de diálogo para mensagens ao usuário
- `QScrollArea` : Área rolável para conteúdo que excede o espaço disponível
- `QFrame` : Widget que pode conter outros widgets e ter uma borda
- `QCheckBox` : Caixa de seleção (checkbox)
- `QSplitter` : Divisor que permite redimensionar painéis
- `QSizePolicy` : Define como um widget deve crescer ou encolher
- `QDialog` : Janela de diálogo modal ou não-modal
- `QTabWidget` : Widget com abas (tabs)

```python
from PyQt6.QtGui import QPixmap, QFont
from PyQt6.QtCore import Qt
```

Linha 11: Importa classes relacionadas a gráficos e fontes:

- `QPixmap` : Representa uma imagem que pode ser exibida na tela
- `QFont` : Representa uma fonte de texto com tamanho, estilo e família

Linha 12: Importa a classe `Qt` , que contém enumerações e constantes usadas em toda a biblioteca Qt, como alinhamentos, orientações e flags.

```python
from datetime import datetime
from pytz import timezone
from Config import ConfigBanco
```

Linha 13: Importa a classe `datetime` para manipulação de datas e horas.

Linha 14: Importa a função `timezone` da biblioteca `pytz` para trabalhar com fusos horários. Isso garante que as datas sejam registradas no fuso horário correto (America/Sao_Paulo).

Linha 15: Importa a classe `ConfigBanco` do arquivo `Config.py` criado anteriormente, permitindo acesso às funções de conexão com o banco de dados.

5.2.2. Classe `ItemEmprestimo`

Esta classe cria um widget personalizado para exibir cada empréstimo na interface.

Python

```
class ItemEmprestimo(QFrame):
    def __init__(self, id_emprestimo, componente, quantidade, data_emp,
data_dev, status, parent=None):
        super().__init__(parent)
        self.id_emprestimo = id_emprestimo
        self.componente = componente
        self.quantidade = quantidade
        self.data_emp = data_emp
        self.data_dev = data_dev
        self.status = status
        self.configurar_interface()
```

Linha 17: Define a classe `ItemEmprestimo` que herda de `QFrame`. Cada instância desta classe representa visualmente um empréstimo.

Linha 18: Define o construtor `__init__` que recebe os dados de um empréstimo:

- `id_emprestimo`: ID único do empréstimo no banco de dados
- `componente`: Nome do componente emprestado
- `quantidade`: Quantidade emprestada
- `data_emp`: Data de empréstimo (formatada como string)
- `data_dev`: Data de devolução (formatada como string ou "---")
- `status`: Status atual ("Emprestado" ou "Devolvido")
- `parent`: Widget pai (opcional)

Linha 19: Chama o construtor da classe pai (`QFrame`) para inicializar o widget base.

Linhas 20-26: Armazenam os dados recebidos como atributos da instância para uso posterior.

Linha 27: Chama o método `configurar_interface()` para construir a aparência visual do item.

Python

```
    def configurar_interface(self):
        self.setFrameShape(QFrame.Shape.StyledPanel)
        self.setFixedHeight(70)
        self.setSizePolicy(QSizePolicy.Policy.Expanding,
QSizePolicy.Policy.Fixed)
```

Linha 29: Define o método que constrói a interface visual do item.

Linha 30: Define o formato do frame como `StyledPanel`, que adiciona uma borda estilizada ao redor do widget.

Linha 31: Define a altura fixa do item como 70 pixels, garantindo que todos os itens tenham o mesmo tamanho vertical.

Linha 32: Define a política de tamanho do widget:

- `Expanding` horizontalmente: o widget pode crescer para preencher o espaço disponível
- `Fixed` verticalmente: a altura permanece fixa em 70 pixels

Python

```
        layout = QHBoxLayout()
        layout.setContentsMargins(10, 5, 10, 5)
```

Linha 34: Cria um layout horizontal (`QHBoxLayout`) que organizará os elementos do item da esquerda para a direita.

Linha 35: Define as margens internas do layout (esquerda, topo, direita, baixo) em pixels. Isso cria um espaçamento entre o conteúdo e a borda do frame.

Python

```
        self.checkbox = QCheckBox()
        self.checkbox.setEnabled(self.status == "Emprestado")
        layout.addWidget(self.checkbox)
```

Linha 37: Cria um checkbox (caixa de seleção) que será usado para marcar itens para devolução.

Linha 38: Habilita o checkbox apenas se o status for "Emprestado". Itens já devolvidos não podem ser selecionados novamente.

Linha 39: Adiciona o checkbox ao layout horizontal.

Python

```
```

```
        layout_info = QVBoxLayout()
        layout_info.setSpacing(2)
```

Linha 41: Cria um layout vertical para organizar as informações do empréstimo (nome, quantidade, datas) em linhas.

Linha 42: Define o espaçamento entre os elementos do layout vertical como 2 pixels, criando uma aparência compacta.

Python

```
        cabecalho = QHBoxLayout()
        label_nome = QLabel(f"<b>{self.componente}</b>")
        label_nome.setFont(QFont("Arial", 9))
        label_nome.setMinimumWidth(150)
        cabecalho.addWidget(label_nome)
```

Linha 44: Cria um layout horizontal para o cabeçalho do item (primeira linha).

Linha 45: Cria um label com o nome do componente em negrito. O uso de tags HTML (`<b>`) permite formatação rica do texto.

Linha 46: Define a fonte do label como Arial, tamanho 9.

Linha 47: Define a largura mínima do label como 150 pixels para garantir alinhamento consistente.

Linha 48: Adiciona o label do nome ao layout do cabeçalho.

Python

```
        label_quantidade = QLabel(f"<b>Quantidade:</b> {self.quantidade}")
        label_quantidade.setFont(QFont("Arial", 9))
        cabecalho.addWidget(label_quantidade)
        cabecalho.addStretch()
```

Linha 50: Cria um label exibindo a quantidade emprestada, com o rótulo "Quantidade:" em negrito.

Linha 51: Define a fonte como Arial, tamanho 9.

Linha 52: Adiciona o label de quantidade ao cabeçalho.

Linha 53: Adiciona um espaço elástico (`addStretch()`) que empurra os próximos elementos para a direita, criando um espaçamento flexível.

Python

```
        self.label_status = QLabel()
        if self.status == "Emprestado":
            self.label_status.setText("<b>EMPRESTADO</b>")
            self.label_status.setStyleSheet("color: red;")
        else:
            self.label_status.setText("<b>DEVOLVIDO</b>")
            self.label_status.setStyleSheet("color: green;")
        self.label_status.setFont(QFont("Arial", 9))
        cabecalho.addWidget(self.label_status)
```

Linha 55: Cria um label para exibir o status do empréstimo.

Linhas 56-61: Condicional que define o texto e a cor do status:

- Se "Emprestado": texto em vermelho
- Se "Devolvido": texto em verde
  Isso fornece feedback visual imediato sobre o estado do empréstimo.

Linha 62: Define a fonte do label de status.

Linha 63: Adiciona o label de status ao cabeçalho.

Python

```
        layout_info.addLayout(cabecalho)
```

Linha 65: Adiciona o layout do cabeçalho (que contém nome, quantidade e status) ao layout vertical de informações.

Python

```
        layout_datas = QHBoxLayout()

        layout_data_emp = QHBoxLayout()
        layout_data_emp.addWidget(QLabel("<b>Retirado:</b>"))
        self.label_data_emp = QLabel(self.data_emp)
        self.label_data_emp.setFont(QFont("Arial", 8))
        layout_data_emp.addWidget(self.label_data_emp)
        layout_datas.addLayout(layout_data_emp)
```

Linha 67: Cria um layout horizontal para as datas (segunda linha do item).

Linha 69: Cria um sub-layout horizontal para a data de empréstimo.

Linha 70: Adiciona um label com o texto "Retirado:" em negrito.

Linha 71: Cria um label com a data de empréstimo.

Linha 72: Define a fonte como Arial, tamanho 8 (menor que o cabeçalho).

Linha 73: Adiciona o label da data ao sub-layout.

Linha 74: Adiciona o sub-layout de data de empréstimo ao layout de datas.

Python

```
        layout_data_dev = QHBoxLayout()
        layout_data_dev.addWidget(QLabel("<b>Devolvido:</b>"))
        self.label_data_dev = QLabel(self.data_dev)
        self.label_data_dev.setFont(QFont("Arial", 8))
        layout_data_dev.addWidget(self.label_data_dev)
        layout_datas.addLayout(layout_data_dev)
```

Linhas 76-81: Seguem o mesmo padrão das linhas anteriores, mas para a data de devolução. Se o item ainda não foi devolvido, este label exibirá "---".

Python

```
        layout_info.addLayout(layout_datas)
        layout.addLayout(layout_info, 1)
        self.setLayout(layout)
```

Linha 83: Adiciona o layout de datas ao layout vertical de informações.

Linha 84: Adiciona o layout vertical de informações ao layout horizontal principal. O parâmetro `1` indica o fator de estiramento (stretch factor), fazendo com que este layout ocupe todo o espaço horizontal disponível.

Linha 85: Define o layout horizontal principal como o layout deste widget.

Python

```
        self.setStyleSheet("""
            QFrame {
                border: 1px solid #D6DBE0;
                border-radius: 4px;
                background-color: #FFFFFF;
            }
            QLabel { color: #333333; }
            QCheckBox { color: #333333; }
            QFrame:hover { background-color: #F5F5F5; }
        """)
```

Linhas 87-96: Define o estilo CSS do widget:

- `QFrame`: Borda cinza clara, cantos arredondados, fundo branco

- `QLabel` e `QCheckBox` : Texto em cinza escuro
- `QFrame:hover` : Quando o mouse passa sobre o item, o fundo fica cinza claro

Este estilo cria uma aparência moderna e profissional para cada item de empréstimo.

5.2.3. Classe Principal `AppAlmoxarifado`

Esta é a classe principal da aplicação, que gerencia toda a interface gráfica e a lógica de negócio.

Python

```
class AppAlmoxarifado(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Sistema de Almoxarifado - Liberato")
        self.setGeometry(100, 100, 1200, 700)
```

Linha 98: Define a classe principal que herda de `QWidget` , tornando-a uma janela de aplicação.

Linha 99: Construtor da classe, executado quando uma instância é criada.

Linha 100: Chama o construtor da classe pai (`QWidget`) para inicializar o widget base.

Linha 101: Define o título da janela que aparecerá na barra de título.

Linha 102: Define a geometria da janela:

- Posição X: 100 pixels da esquerda da tela
- Posição Y: 100 pixels do topo da tela
- Largura: 1200 pixels
- Altura: 700 pixels

Python

```
        self.modo_escuro = False
        self.matricula_selecionada = None
        self.itens_emprestimo = []
```

Linha 104: Inicializa a variável que controla o tema da interface (claro/escuro). Começa como `False` (tema claro).

Linha 105: Armazena a matrícula do aluno atualmente selecionado. Inicia como `None` (nenhum aluno selecionado).

Linha 106: Lista que armazenará os widgets `ItemEmprestimo` exibidos na interface.

Python

```
        layout_principal = QVBoxLayout()
        self.setLayout(layout_principal)

        self.configurar_interface()
        self.aplicar_estilo()
```

Linha 108: Cria um layout vertical que organizará os elementos principais da janela de cima para baixo.

Linha 109: Define este layout como o layout principal da janela.

Linha 111: Chama o método que constrói todos os componentes da interface.

Linha 112: Chama o método que aplica os estilos CSS à interface.

5.2.4. Métodos de Pesquisa e Exibição

Python

```
    def pesquisar_aluno(self):
        termo = self.campo_pesquisa.text().strip()
        if not termo:
            QMessageBox.warning(self, "Campo Vazio", "Digite um termo para
pesquisa")
            return
```

Linha 540: Define o método que realiza a pesquisa de alunos no banco de dados.

Linha 541: Obtém o texto digitado no campo de pesquisa e remove espaços em branco nas extremidades com `strip()`.

Linhas 542-544: Valida se o campo não está vazio. Se estiver, exibe uma mensagem de aviso e retorna sem executar a pesquisa.

Python

```
        self.lista_alunos.clear()

        try:
            conexao = ConfigBanco.obter_conexao()
            if conexao is None:
                raise Error("Não foi possível conectar ao banco de dados")
```

Linha 546: Limpa a lista de alunos, removendo resultados de pesquisas anteriores.

Linha 548: Inicia um bloco `try` para capturar possíveis erros.

Linha 549: Obtém uma conexão com o banco de dados usando a classe `ConfigBanco`.

Linhas 550-551: Verifica se a conexão foi estabelecida com sucesso. Se retornar `None`, lança uma exceção.

Python

```
            cursor = conexao.cursor()

            cursor.execute(
                "SELECT matricula, nome FROM alunos "
                "WHERE matricula LIKE %s OR nome LIKE %s "
                "LIMIT 50",
                (f"%{termo}%", f"%{termo}%")
            )
            resultados = cursor.fetchall()
```

Linha 553: Cria um cursor, que é um objeto usado para executar comandos SQL e recuperar resultados.

Linhas 555-560: Executa uma query SQL que:

- Seleciona as colunas `matricula` e `nome` da tabela `alunos`
- Busca registros onde a matrícula OU o nome contenham o termo pesquisado (operador `LIKE`)
- Limita os resultados a 50 registros para evitar sobrecarregar a interface
- Os símbolos `%` ao redor do termo permitem busca parcial (ex: "Silva" encontra "João Silva" e "Silva Santos")
- Os `%s` são placeholders que serão substituídos pelos valores na tupla seguinte

Linha 561: Recupera todos os resultados da query em uma lista de tuplas.

Python

```
            if resultados:
                for matricula, nome in resultados:
                    self.lista_alunos.addItem(f"{matricula} - {nome}")
            else:
                self.lista_alunos.addItem("Nenhum aluno encontrado.")
```

Linha 563: Verifica se há resultados.

Linhas 564-565: Para cada resultado, adiciona um item à lista no formato "matrícula - nome".

Linhas 566-567: Se não houver resultados, exibe uma mensagem informativa.

```
Python
```

```
            cursor.close()
            conexao.close()
        except Error as erro:
            QMessageBox.critical(self, "Erro", f"Erro na pesquisa:
{str(erro)}")
```

Linha 569: Fecha o cursor para liberar recursos.

Linha 570: Fecha a conexão com o banco de dados.

Linhas 571-572: Captura qualquer erro de banco de dados e exibe uma mensagem crítica ao usuário.

5.2.5. Método de Adicionar Empréstimo

```
Python
```

```
    def adicionar_emprestimo(self):
        if not self.matricula_selecionada:
            QMessageBox.warning(self, "Aluno Não Selecionado", "Selecione um
aluno primeiro")
            return
```

Linha 820: Define o método que registra um novo empréstimo.

Linhas 821-823: Valida se um aluno foi selecionado. Se não, exibe aviso e retorna.

```
Python
```

```
        componente = self.campo_componente.text().strip()
        quantidade = self.campo_quantidade.text().strip()

        if not componente:
            QMessageBox.warning(self, "Componente Ausente", "Informe o
componente")
            return

        if not quantidade.isdigit() or int(quantidade) <= 0:
            QMessageBox.warning(self, "Quantidade Inválida", "Digite um
```

```
número positivo")
            return
```

Linhas 825-826: Obtém os valores dos campos de componente e quantidade.

Linhas 828-830: Valida se o campo componente não está vazio.

Linhas 832-834: Valida se a quantidade é um número inteiro positivo. O método `isdigit()` verifica se a string contém apenas dígitos.

Python

```
        conexao = None
        cursor = None
        try:
            conexao = ConfigBanco.obter_conexao()
            if conexao is None:
                raise Error("Não foi possível conectar ao banco de dados")
```

Linhas 836-837: Inicializa as variáveis como `None` para garantir que existam no escopo do `finally`.

Linhas 838-841: Obtém conexão e valida se foi bem-sucedida.

Python

```
            cursor = conexao.cursor()

            cursor.execute("SELECT id FROM alunos WHERE matricula = %s",
(self.matricula_selecionada,))
            aluno = cursor.fetchone()
            if not aluno:
                QMessageBox.warning(self, "Aluno Não Encontrado", "Matrícula
não existe")
                return
            aluno_id = aluno[0]
```

Linha 843: Cria o cursor.

Linhas 845-850: Busca o ID do aluno pela matrícula. O método `fetchone()` retorna uma tupla com um único resultado ou `None` se não encontrar. O ID é extraído da primeira posição da tupla (`aluno[0]`).

Python

```
            if componente.isdigit():
                cursor.execute("SELECT id, nome FROM componentes WHERE id =
%s", (int(componente),))
```

```
            else:
                cursor.execute("SELECT id, nome FROM componentes WHERE nome
LIKE %s", (f"%{componente}%",))

            componente_info = cursor.fetchone()
            if not componente_info:
                QMessageBox.warning(self, "Componente Não Encontrado",
"Nenhum componente encontrado")
                return

            componente_id, componente_nome = componente_info
```

Linhas 852-854: Verifica se o usuário digitou um ID numérico ou um nome. Se for número, busca por ID exato; se for texto, busca por nome parcial.

Linhas 856-859: Valida se o componente foi encontrado.

Linha 861: Desempacota a tupla retornada em duas variáveis: ID e nome do componente.

Python

```
            agora = datetime.now(timezone("America/Sao_Paulo"))
            cursor.execute(
                """INSERT INTO emprestimos
                   (aluno_id, componente_id, quantidade, data_emp, status)
                   VALUES (%s, %s, %s, %s, 'Emprestado')""",
                (aluno_id, componente_id, int(quantidade), agora)
            )

            conexao.commit()
            self.carregar_emprestimos(self.matricula_selecionada)
```

Linha 863: Obtém a data e hora atual no fuso horário de São Paulo.

Linhas 864-869: Executa um `INSERT` para adicionar o novo empréstimo ao banco de dados. O status é automaticamente definido como 'Emprestado'.

Linha 871: Confirma a transação no banco de dados (commit). Sem isso, as alterações seriam descartadas.

Linha 872: Recarrega a lista de empréstimos para exibir o novo registro.

Python

```
            self.campo_componente.clear()
            self.campo_quantidade.clear()

        except Error as erro:
            QMessageBox.critical(self, "Erro", f"Erro ao registrar
```

```
empréstimo: {str(erro)}")
        finally:
            if cursor:
                cursor.close()
            if conexao:
                conexao.close()
```

Linhas 874-875: Limpa os campos do formulário após o sucesso.

Linhas 877-878: Captura e exibe erros.

Linhas 879-883: O bloco `finally` garante que a conexão e o cursor sejam fechados mesmo se ocorrer um erro.

5.2.6. Método de Registrar Devolução

Python

```
    def registrar_devolucao(self):
        if not self.matricula_selecionada:
            QMessageBox.warning(self, "Aluno Não Selecionado", "Selecione um
aluno primeiro")
            return

        itens_selecionados = []
        for i in range(self.layout_emprestados.count()):
            widget = self.layout_emprestados.itemAt(i).widget()
            if isinstance(widget, ItemEmprestimo) and
widget.checkbox.isChecked():
                itens_selecionados.append(widget)
```

Linha 885: Define o método que registra devoluções.

Linhas 886-888: Valida se há um aluno selecionado.

Linhas 890-894: Itera sobre todos os widgets no layout de empréstimos ativos, verifica se são instâncias de `ItemEmprestimo` e se seus checkboxes estão marcados. Os itens marcados são adicionados à lista.

Python

```
        if not itens_selecionados:
            QMessageBox.warning(
                self,
                "Nenhum Item Selecionado",
                "Selecione pelo menos um empréstimo ativo"
            )
            return
```

```
        confirmacao = QMessageBox.question(
            self,
            "Confirmar Devolução",
            f"Deseja registrar a devolução de {len(itens_selecionados)}
item(ns)?",
            QMessageBox.StandardButton.Yes | QMessageBox.StandardButton.No
        )
        if confirmacao != QMessageBox.StandardButton.Yes:
            return
```

Linhas 896-902: Valida se pelo menos um item foi selecionado.

Linhas 904-909: Exibe uma caixa de diálogo de confirmação perguntando se o usuário realmente deseja registrar a devolução. Retorna se o usuário clicar em "No".

Python

```
        conexao = None
        cursor = None
        try:
            conexao = ConfigBanco.obter_conexao()
            if conexao is None:
                raise Error("Não foi possível conectar ao banco de dados")

            cursor = conexao.cursor()

            agora = datetime.now(timezone("America/Sao_Paulo"))
            for item in itens_selecionados:
                cursor.execute(
                    """UPDATE emprestimos SET data_dev = %s, status =
'Devolvido'
                       WHERE id = %s""",
                    (agora, item.id_emprestimo)
                )

            conexao.commit()
            self.carregar_emprestimos(self.matricula_selecionada)

        except Error as erro:
            QMessageBox.critical(self, "Erro", f"Erro ao registrar
devolução: {str(erro)}")
        finally:
            if cursor:
                cursor.close()
            if conexao:
                conexao.close()
```

Linhas 911-918: Estabelece conexão com o banco de dados.

Linha 920: Obtém a data e hora atual.

Linhas 921-926: Para cada item selecionado, executa um `UPDATE` que:

- Define `data_dev` como a data/hora atual
- Altera o `status` para 'Devolvido'
- Usa o `id_emprestimo` do widget para identificar o registro correto

Linha 928: Confirma as alterações no banco.

Linha 929: Recarrega os empréstimos para atualizar a interface.

Linhas 931-937: Tratamento de erros e fechamento de recursos.

---

6. Scripts Auxiliares

Além da aplicação principal, o sistema inclui scripts auxiliares para manutenção e atualização de dados.

6.1. Script `processar_alunos.py` - Análise Linha por Linha

Este script processa arquivos CSV de alunos, eliminando duplicatas e preparando os dados para importação.

Python

```python
import pandas as pd
import os
import glob
```

Linha 1: Importa a biblioteca `pandas`, uma poderosa ferramenta para manipulação e análise de dados tabulares (como planilhas e CSVs).

Linha 2: Importa o módulo `os`, que fornece funções para interagir com o sistema operacional.

Linha 3: Importa o módulo `glob`, usado para buscar arquivos que correspondam a um padrão específico (como "alunos*.csv").

Python

```python
def processar_alunos():
    try:
        # 🔍 Busca o arquivo CSV mais recente com prefixo 'alunos'
        arquivos = glob.glob('alunos*.csv')
```

```
        if not arquivos:
            print("❌ Nenhum arquivo 'alunos*.csv' encontrado.")
            return False
```

Linha 5: Define a função principal que encapsula toda a lógica de processamento.

Linha 6: Inicia um bloco `try` para capturar exceções.

Linha 8: Usa `glob.glob()` para buscar todos os arquivos que começam com "alunos" e terminam com ".csv" no diretório atual. Retorna uma lista de nomes de arquivos.

Linhas 9-11: Verifica se a lista está vazia. Se não encontrar nenhum arquivo, exibe mensagem e retorna `False`.

Python

```
        arquivo_mais_recente = max(arquivos, key=os.path.getmtime)
        print(f"📂 Arquivo selecionado: {arquivo_mais_recente}")
```

Linha 13: Seleciona o arquivo mais recente da lista. A função `max()` usa `os.path.getmtime` como critério, que retorna o timestamp da última modificação do arquivo.

Linha 14: Exibe o nome do arquivo selecionado.

Python

```
        # 📥 Tenta diferentes encodings
        try:
            df = pd.read_csv(arquivo_mais_recente, encoding='utf-8')
        except UnicodeDecodeError:
            df = pd.read_csv(arquivo_mais_recente, encoding='latin-1')
            print("⚠️ Usando encoding latin-1")
```

Linha 17: Tenta ler o CSV com encoding UTF-8 (padrão moderno).

Linhas 18-20: Se ocorrer um erro de decodificação (arquivo não está em UTF-8), tenta novamente com encoding Latin-1 (comum em sistemas Windows antigos). Isso garante compatibilidade com diferentes fontes de dados.

Python

```
        # ✅ Verifica e cria campos obrigatórios
        campos_necessarios = ['id', 'matricula', 'nome', 'turma']
        for campo in campos_necessarios:
            if campo not in df.columns:
                if campo == 'id':
                    df['id'] = range(1, len(df) + 1)
```

```
                else:
                    df[campo] = ''
```

Linha 23: Define uma lista dos campos obrigatórios que o sistema espera.

Linhas 24-29: Itera sobre cada campo necessário:

- Se o campo não existir no DataFrame, ele é criado
- Se for o campo 'id', gera uma sequência numérica de 1 até o número total de linhas
- Para outros campos, cria uma coluna vazia (string vazia)

Python

```
        df_alunos = df[campos_necessarios].copy()
```

Linha 31: Cria um novo DataFrame contendo apenas as colunas necessárias. O método `.copy()` cria uma cópia independente para evitar modificações no DataFrame original.

Python

```
        # 📧 Cria campo de e-mail institucional
        df_alunos['email'] = df_alunos['matricula'].astype(str) +
'@liberato.com.br'
```

Linha 34: Cria a coluna 'email' concatenando a matrícula (convertida para string) com o domínio institucional. Exemplo: matrícula 12345 vira "12345@liberato.com.br".

Python

```
        # 🔢 Converte turma para numérico para comparação
        df_alunos['turma_num'] = pd.to_numeric(df_alunos['turma'],
errors='coerce')
```

Linha 37: Cria uma coluna auxiliar 'turma_num' convertendo os valores de 'turma' para números. O parâmetro `errors='coerce'` faz com que valores não numéricos sejam convertidos para `NaN` (Not a Number) em vez de gerar erro.

Python

```
        # 🧠 Agrupa por matrícula e mantém a turma mais alta
        df_alunos.sort_values(by=['matricula', 'turma_num'], ascending=
[True, False], inplace=True)
        df_alunos = df_alunos.drop_duplicates(subset='matricula',
keep='first')
```

Linha 40: Ordena o DataFrame por matrícula (ordem crescente) e por turma_num (ordem decrescente). Isso garante que, para cada matrícula, o registro com a turma mais alta apareça primeiro.

Linha 41: Remove duplicatas baseadas na coluna 'matricula', mantendo apenas a primeira ocorrência (`keep='first'`). Como ordenamos pela turma mais alta, isso garante que mantemos o registro mais recente de cada aluno.

Python

```
# 🧹 Remove campo auxiliar
df_alunos.drop(columns=['turma_num'], inplace=True)
```

Linha 44: Remove a coluna auxiliar 'turma_num', pois não é mais necessária.

Python

```
# 🔍 Verifica duplicatas
duplicatas_matricula = df_alunos['matricula'].duplicated().sum()
duplicatas_nome = df_alunos['nome'].duplicated().sum()

if duplicatas_matricula > 0 or duplicatas_nome > 0:
    print(f"⚠️ Atenção: {duplicatas_matricula} matrículas e
{duplicatas_nome} nomes duplicados.")
else:
    print("✅ Nenhuma duplicata de matrícula ou nome encontrada.")
```

Linhas 47-48: Conta quantas duplicatas ainda existem nas colunas 'matricula' e 'nome'. O método `.duplicated()` retorna uma série booleana, e `.sum()` conta os valores `True` .

Linhas 50-53: Exibe mensagem apropriada dependendo se foram encontradas duplicatas.

Python

```
# 💾 Exporta para CSV compatível com Banco 2
df_alunos.to_csv('alunos_banco2.csv', index=False, encoding='utf-8')
print("✅ 'alunos_banco2.csv' gerado com sucesso!")
```

Linha 56: Exporta o DataFrame processado para um novo arquivo CSV:

- `index=False` : Não inclui o índice do DataFrame como coluna
- `encoding='utf-8'` : Garante compatibilidade com caracteres especiais

Linha 57: Exibe mensagem de sucesso.

Python

```
        # 👀 Preview
        print("\n📋 Primeiras 5 linhas:")
        print(df_alunos.head())

        return True

    except Exception as e:
        print(f"❌ Erro inesperado: {e}")
        return False
```

Linhas 60-61: Exibe as primeiras 5 linhas do DataFrame processado para que o usuário possa verificar visualmente o resultado.

Linha 63: Retorna `True` indicando sucesso.

Linhas 65-67: Captura qualquer exceção não prevista, exibe a mensagem de erro e retorna `False`.

Python

```
if __name__ == "__main__":
    processar_alunos()
```

Linhas 69-70: Este bloco só é executado se o script for executado diretamente (não importado como módulo). Chama a função principal.

6.2. Script `atualizar_fotos.py` - Análise Linha por Linha

Este script baixa fotos da intranet e atualiza automaticamente o banco de dados.

Python

```
import mysql.connector
import requests
from mysql.connector import Error
import logging
from pathlib import Path
import sys
```

Linha 1: Importa o conector MySQL para operações de banco de dados.

Linha 2: Importa a biblioteca `requests`, usada para fazer requisições HTTP e baixar as imagens da intranet.

Linha 3: Importa a classe `Error` para tratamento de exceções de banco de dados.

Linha 4: Importa o módulo `logging`, que fornece um sistema profissional de registro de eventos e erros.

Linha 5: Importa `Path` da biblioteca `pathlib`, uma forma moderna de manipular caminhos de arquivos.

Linha 6: Importa `sys` para controle do fluxo de execução e códigos de saída.

Python

```python
# 📋 CONFIGURAÇÃO DO SISTEMA DE LOGS
def setup_logging():
    """
    Configura o sistema de logging para registrar em arquivo e console
    """
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
        handlers=[
            logging.FileHandler("atualizacao_fotos.log", encoding='utf-8'),
            logging.StreamHandler(sys.stdout)
        ]
    )
    return logging.getLogger(__name__)
```

Linha 9: Define uma função para configurar o sistema de logs.

Linhas 10-13: Docstring explicando o propósito da função.

Linhas 14-22: Configura o logging:

- `level=logging.INFO`: Registra mensagens de nível INFO ou superior (INFO, WARNING, ERROR, CRITICAL)
- `format`: Define o formato das mensagens incluindo timestamp, nome do logger, nível e mensagem
- `handlers`: Define dois destinos para os logs:
  - `FileHandler`: Salva em arquivo "atualizacao_fotos.log"
  - `StreamHandler`: Exibe no console (stdout)

Linha 23: Retorna um objeto logger configurado.

Python

```python
# 🛡️ FUNÇÃO DE VALIDAÇÃO DE IMAGEM
def is_valid_image(content):
    """
    Verifica se o conteúdo é uma imagem válida:
```

```
    - Tamanho mínimo de 1KB
    - Assinatura mágica de formatos conhecidos
    - Não começa com HTML ou mensagens de erro
    """
    if len(content) < 1024:
        return False
```

Linha 26: Define uma função que valida se o conteúdo baixado é realmente uma imagem.

Linhas 27-32: Docstring explicando os critérios de validação.

Linhas 33-34: Verifica se o tamanho do conteúdo é menor que 1KB (1024 bytes). Imagens muito pequenas provavelmente são inválidas ou corrompidas.

Python

```
    image_signatures = {
        b'\xff\xd8\xff': 'JPEG',
        b'\x89PNG\r\n\x1a\n': 'PNG',
        b'GIF8': 'GIF',
        b'BM': 'BMP'
    }

    for signature in image_signatures:
        if content.startswith(signature):
            return True
```

Linhas 36-41: Define um dicionário com as "assinaturas mágicas" (magic numbers) dos formatos de imagem mais comuns. Estes são os bytes iniciais que identificam o tipo de arquivo:

- JPEG começa com `FF D8 FF`
- PNG começa com `89 50 4E 47 0D 0A 1A 0A`
- GIF começa com `GIF8`
- BMP começa com `BM`

Linhas 43-45: Itera sobre as assinaturas e verifica se o conteúdo começa com alguma delas. Se sim, é uma imagem válida.

Python

```
    invalid_starts = [b'<html', b'<!DOCTYPE', b'HTTP', b'Error', b'Not
Found']
    for invalid in invalid_starts:
        if content.startswith(invalid):
            return False
```

```
    return True
```

Linha 47: Define uma lista de inícios de conteúdo que indicam que o download não foi uma imagem (páginas de erro HTML, mensagens HTTP, etc.).

Linhas 48-50: Verifica se o conteúdo começa com algum desses padrões inválidos. Se sim, retorna `False`.

Linha 52: Se passou por todas as verificações sem encontrar problemas, retorna `True`.

Python

```
# 🚀 FUNÇÃO PRINCIPAL
def atualizar_fotos_alunos():
    """
    Atualiza a coluna 'fotos' da tabela 'alunos' com imagens da intranet
    usando o campo 'id' como chave única
    """
    try:
        # 🔌 Conexão com o banco de dados MySQL
        logger.info("🔌 Conectando ao banco de dados MySQL...")
        conn = mysql.connector.connect(
            host='10.233.43.215',
            user='troadmin',
            password='eleTROn1c_flop3#W!w',
            database='site_eletronicaX',
            charset='utf8mb4',
            connection_timeout=30
        )
```

Linha 55: Define a função principal que realiza a atualização das fotos.

Linhas 56-60: Docstring explicando o propósito da função.

Linha 61: Inicia bloco `try` para tratamento de exceções.

Linha 63: Registra no log que está iniciando a conexão.

Linhas 64-71: Estabelece conexão com o banco de dados MySQL usando as mesmas credenciais do sistema principal.

Python

```
        if not conn.is_connected():
            logger.error("❌ Falha na conexão com o MySQL")
            return False
```

```
        logger.info("✅ Conexão com MySQL estabelecida com sucesso!")
```

Linhas 73-75: Verifica se a conexão foi estabelecida. Se não, registra erro e retorna `False`.

Linha 77: Registra sucesso na conexão.

Python

```
        # 🌐 URL base da intranet onde estão as imagens
        url_base = 'https://intranet.liberato.com.br/tro/data/fotos/'

        # 📝 Cria arquivo de falhas se não existir
        Path('falhas.txt').touch(exist_ok=True)
        logger.info("📁 Arquivo de falhas preparado: falhas.txt")
```

Linha 80: Define a URL base onde as fotos estão hospedadas na intranet.

Linha 83: Cria o arquivo `falhas.txt` se não existir. O método `touch()` cria um arquivo vazio.

Linha 84: Registra que o arquivo de falhas está pronto.

Python

```
        with conn.cursor(buffered=True) as cursor:
            # 🔍 Busca todos os alunos com id e matrícula
            logger.info("🔍 Buscando registros de alunos...")
            cursor.execute('SELECT id, matricula FROM alunos')
            registros = cursor.fetchall()

            total = len(registros)
            sucessos = 0
            falhas = 0
            pulados = 0
```

Linha 86: Cria um cursor com buffer. O `with` garante que o cursor será fechado automaticamente ao final do bloco.

Linhas 88-90: Busca todos os IDs e matrículas da tabela alunos.

Linhas 92-95: Inicializa contadores para estatísticas.

Python

```
            logger.info(f"🎯 Iniciando processamento de {total} alunos...")

            # 🔁 Loop por cada aluno
            for i, (id_aluno, matricula) in enumerate(registros, 1):
```

```
                extensoes = ['.jpeg', '.jpg', '.png', '.gif']
                imagem_encontrada = False
```

Linha 97: Registra o início do processamento.

Linha 100: Itera sobre os registros. `enumerate(registros, 1)` retorna o índice (começando em 1) e o conteúdo de cada registro.

Linha 101: Define as extensões de arquivo que serão tentadas para cada aluno.

Linha 102: Flag para controlar se uma imagem foi encontrada para este aluno.

Python

```
                # ⏭️ Verifica se já existe imagem cadastrada
                cursor.execute('SELECT fotos FROM alunos WHERE id = %s',
(id_aluno,))
                resultado = cursor.fetchone()

                if resultado and resultado[0] is not None:
                    pulados += 1
                    if i % 100 == 0:
                        logger.info(f'📊 [{i}/{total}] ⏭️ Pulados={pulados},
✅ Sucessos={sucessos}, ❌ Falhas={falhas}')
                    continue
```

Linhas 105-106: Verifica se o aluno já possui uma foto no banco.

Linhas 108-111: Se já possui foto, incrementa o contador de pulados e continua para o próximo aluno. A cada 100 registros, exibe estatísticas.

Python

```
                # 🔎 Tenta baixar imagem com diferentes extensões
                for extensao in extensoes:
                    nome_arquivo = f'C{matricula}{extensao}'
                    url_imagem = f'{url_base}/{nome_arquivo}'

                    try:
                        response = requests.get(url_imagem, timeout=10,
stream=True)
                        if response.status_code == 200:
                            content = response.content
                            if is_valid_image(content):
                                cursor.execute(
                                    'UPDATE alunos SET fotos = %s WHERE id =
%s',
                                    (content, id_aluno)
```

```
                                )
                                imagem_encontrada = True
                                sucessos += 1
                                logger.info(f'✅ [{i}/{total}] ID {id_aluno}
atualizado - {len(content):,} bytes')
                                break
```

Linhas 114-115: Para cada extensão, constrói o nome do arquivo e a URL completa. O padrão é `C{matricula}.extensao` (ex: C12345.jpeg).

Linha 118: Faz uma requisição HTTP GET para baixar a imagem. `timeout=10` define 10 segundos como limite, e `stream=True` permite baixar em partes.

Linhas 119-120: Verifica se a resposta foi bem-sucedida (código 200) e obtém o conteúdo.

Linhas 121-129: Se a imagem for válida, executa um UPDATE no banco de dados, armazenando o conteúdo binário na coluna `fotos`. Incrementa o contador de sucessos e registra no log. O `break` interrompe o loop de extensões, pois já encontrou uma imagem.

Python

```
                            else:
                                logger.debug(f'🖼️ [{i}/{total}] Imagem
inválida: {url_imagem}')
                        else:
                            logger.debug(f'🌐 [{i}/{total}] HTTP
{response.status_code}: {url_imagem}')
                    except requests.exceptions.Timeout:
                        logger.warning(f'⏱️ [{i}/{total}] Timeout acessando:
{url_imagem}')
                        continue
                    except requests.exceptions.ConnectionError:
                        logger.warning(f'🔌 [{i}/{total}] Erro de conexão:
{url_imagem}')
                        continue
                    except requests.exceptions.RequestException as e:
                        logger.debug(f'🌐 [{i}/{total}] Erro de rede:
{url_imagem} - {e}')
                        continue
```

Linhas 130-132: Se a imagem não for válida ou o código HTTP não for 200, registra no log em nível DEBUG.

Linhas 133-141: Captura diferentes tipos de exceções de rede:

- `Timeout`: Servidor não respondeu a tempo
- `ConnectionError`: Falha na conexão de rede

- `RequestException`: Qualquer outro erro de requisição

Python

```
                # ❌ Registra falha se nenhuma imagem foi encontrada
                if not imagem_encontrada:
                    falhas += 1
                    with open('falhas.txt', 'a', encoding='utf-8') as f:
                        f.write(f'{id_aluno},{matricula}\n')
                    logger.warning(f'❌ [{i}/{total}] Sem imagem para ID
{id_aluno} (Mat: {matricula})')

                # 📈 Log de progresso a cada 50 registros
                if i % 50 == 0:
                    progresso = (i / total) * 100
                    logger.info(f'📈 Progresso: {i}/{total}
({progresso:.1f}%)')
```

Linhas 144-148: Se nenhuma imagem foi encontrada após tentar todas as extensões, incrementa o contador de falhas e registra o ID e matrícula no arquivo `falhas.txt`.

Linhas 151-153: A cada 50 registros, calcula e exibe o percentual de progresso.

Python

```
            # 💾 Confirma todas as alterações no banco
            conn.commit()

            # 📊 Relatório final
            taxa_sucesso = (sucessos / (total - pulados)) * 100 if (total -
pulados) > 0 else 0
            logger.info(f"""
📊 RESUMO FINAL:
   👥 Total de alunos: {total}
   ✅ Fotos atualizadas: {sucessos}
   ⏭️ Registros pulados: {pulados}
   ❌ Falhas: {falhas}
   📈 Taxa de sucesso: {taxa_sucesso:.1f}%
            """)

    except Error as e:
        logger.error(f"❌ Erro de banco de dados: {e}")
        return False
    except Exception as e:
        logger.error(f"❌ Erro inesperado: {e}")
        return False
    finally:
        if 'conn' in locals() and conn.is_connected():
```

```
            conn.close()
            logger.info("🔌 Conexão com MySQL encerrada com sucesso")

    return True
```

Linha 156: Confirma todas as alterações no banco de dados (commit).

Linhas 159-169: Calcula a taxa de sucesso e exibe um relatório final completo com todas as estatísticas.

Linhas 171-176: Captura e registra erros de banco de dados e outros erros inesperados.

Linhas 177-180: O bloco `finally` garante que a conexão seja fechada, mesmo se ocorrer um erro.

Linha 182: Retorna `True` indicando sucesso.

Python

```python
# 🎯 Execução principal
if __name__ == "__main__":
    logger = setup_logging()
    logger.info("🚀 INICIANDO PROCESSO DE ATUALIZAÇÃO DE FOTOS")
    logger.info("=" * 50)

    try:
        if atualizar_fotos_alunos():
            logger.info("🎉 PROCESSAMENTO CONCLUÍDO COM SUCESSO!")
            print("✅ Processamento concluído com sucesso!")
            sys.exit(0)
        else:
            logger.error("💥 OCORRERAM ERROS DURANTE O PROCESSAMENTO")
            print("❌ Ocorreram erros durante o processamento.")
            sys.exit(1)
    except KeyboardInterrupt:
        logger.info("⏹️ Processo interrompido pelo usuário")
        print("⏹️ Processo interrompido pelo usuário")
        sys.exit(1)
    except Exception as e:
        logger.error(f"💥 Erro crítico: {e}")
        print(f"❌ Erro crítico: {e}")
        sys.exit(1)
```

Linhas 185-187: Configura o logger e registra o início do processo.

Linhas 189-195: Executa a função principal e, dependendo do resultado, exibe mensagens e define o código de saída:

- `sys.exit(0)` : Sucesso

- `sys.exit(1)` : Erro

Linhas 196-203: Captura interrupção manual (Ctrl+C) e erros críticos, registrando apropriadamente e encerrando com código de erro.

---

7. Conclusão

Este manual técnico forneceu uma análise abrangente e detalhada do Sistema de Almoxarifado da Fundação Liberato, documentando linha por linha todos os componentes do sistema. Através desta documentação, buscamos criar um recurso completo que serve múltiplos propósitos:

Para Desenvolvedores: O manual oferece uma compreensão profunda da arquitetura, lógica e implementação do sistema, facilitando a manutenção, correção de bugs e implementação de novas funcionalidades.

Para Alunos: A documentação linha por linha serve como material didático, permitindo que estudantes de programação compreendam como um sistema real é estruturado, desde a interface gráfica até a persistência de dados.

Para Funcionários: O guia de instalação e os tutoriais de configuração capacitam a equipe técnica a implantar e gerenciar o sistema de forma independente.

7.1. Pontos Fortes do Sistema

O Sistema de Almoxarifado apresenta várias características positivas que merecem destaque:

- **Arquitetura Modular:** A separação entre configuração (`Config.py`) e aplicação (`app.py`) facilita a manutenção e permite alternar entre ambientes de produção e teste.
- **Interface Intuitiva:** O uso do PyQt6 proporciona uma experiência de usuário moderna e responsiva, com feedback visual claro (cores indicando status de empréstimos).
- **Scripts Auxiliares:** Os scripts de processamento de dados e atualização de fotos automatizam tarefas administrativas, reduzindo trabalho manual e erros humanos.
- **Tratamento de Erros:** O sistema implementa captura de exceções em pontos críticos, exibindo mensagens amigáveis ao usuário.
- **Logging Profissional:** O script de atualização de fotos utiliza o módulo `logging` para manter registros detalhados de todas as operações.

7.2. Recomendações para Melhorias Futuras

Embora o sistema seja funcional e bem estruturado, algumas melhorias poderiam ser implementadas:

Segurança:

- Migrar credenciais do banco de dados para variáveis de ambiente ou arquivos de configuração criptografados
- Implementar autenticação de usuários com diferentes níveis de permissão
- Adicionar logs de auditoria para rastrear quem realizou cada operação

Funcionalidades:

- Implementar relatórios gerenciais (alunos com mais empréstimos, componentes mais emprestados, etc.)
- Adicionar notificações por e-mail para empréstimos pendentes
- Criar um sistema de reserva de componentes
- Implementar controle de estoque com alertas de quantidade baixa

Usabilidade:

- Adicionar atalhos de teclado para operações frequentes
- Implementar busca avançada com filtros múltiplos
- Criar um modo de visualização em tela cheia para apresentações

Manutenibilidade:

- Separar a lógica de negócio da interface gráfica em módulos distintos
- Implementar testes automatizados (unit tests)
- Adicionar documentação inline (docstrings) em todos os métodos

7.3. Palavras Finais

O Sistema de Almoxarifado representa um exemplo prático de como Python e suas bibliotecas podem ser utilizados para criar aplicações desktop robustas e funcionais. A documentação detalhada apresentada neste manual não apenas explica o código existente, mas também serve como base para futuras expansões e melhorias.

Esperamos que este documento seja útil para todos os envolvidos no projeto, desde desenvolvedores experientes até estudantes que estão dando seus primeiros passos na programação. A clareza na documentação é fundamental para a longevidade e evolução de qualquer sistema de software.

---

8. Apêndices

Apêndice A: Tutorial de Configuração de Chaves Primárias no DBeaver

Este tutorial detalha o processo para configurar corretamente as colunas `id` como chaves primárias com auto-incremento nas tabelas do sistema, utilizando o DBeaver.

Plain Text

```
📋 TUTORIAL COMPLETO - CONFIGURAR CHAVES PRIMÁRIAS NO DBEAVER
================================================================

🎯 OBJETIVO
Transformar a coluna 'id' em chave primária AUTO_INCREMENT
nas tabelas: alunos, componentes e emprestimos

📍 PRÉ-REQUISITOS
✔ DBeaver instalado e conectado ao MySQL
✔ Banco de dados criado
✔ Tabelas alunos, componentes e emprestimos existem
✔ Coluna 'id' existe em todas as tabelas

📋 PARTE 1: VERIFICAÇÃO INICIAL
--------------------------------

1. ✅ ABRA UMA NOVA ABA SQL
   - Clique em "SQL Editor" → "New SQL Script"

2. 🔍 EXECUTE ESTAS CONSULTAS DE VERIFICAÇÃO:

-- Verifica estrutura atual
DESC alunos;
DESC componentes;
DESC emprestimos;

-- Verifica se já tem primary key
SELECT TABLE_NAME, COLUMN_KEY
FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_SCHEMA = DATABASE()
AND TABLE_NAME IN ('alunos', 'componentes', 'emprestimos')
AND COLUMN_NAME = 'id';

3. 📊 INTERPRETE OS RESULTADOS:
   - Se COLUMN_KEY = 'PRI' → Já é chave primária
   - Se COLUMN_KEY = '' → Precisa converter
   - Se mostrar erro → Tabela não existe

📋 PARTE 2: EXECUÇÃO SEGURA (UM POR VEZ)
-----------------------------------------
```

⚠️ IMPORTANTE: Execute CADA COMANDO em ABAS SEPARADAS

1. 🚀 ABA 1 - TABELA ALUNOS:
   ```
   ----------------------------------------
   ALTER TABLE alunos
   MODIFY COLUMN id INT NOT NULL AUTO_INCREMENT,
   ADD PRIMARY KEY (id);
   ----------------------------------------
   ```
   ✅ Clique em "Execute SQL Statement" (Ctrl+Enter)

2. 🚀 ABA 2 - TABELA COMPONENTES:
   ```
   ----------------------------------------
   ALTER TABLE componentes
   MODIFY COLUMN id INT NOT NULL AUTO_INCREMENT,
   ADD PRIMARY KEY (id);
   ----------------------------------------
   ```
   ✅ Clique em "Execute SQL Statement" (Ctrl+Enter)

3. 🚀 ABA 3 - TABELA EMPRÉSTIMOS:
   ```
   ----------------------------------------
   ALTER TABLE emprestimos
   MODIFY COLUMN id INT NOT NULL AUTO_INCREMENT,
   ADD PRIMARY KEY (id);
   ----------------------------------------
   ```
   ✅ Clique em "Execute SQL Statement" (Ctrl+Enter)

📋 PARTE 3: MÉTODO VISUAL (ALTERNATIVO)
```
----------------------------------------
```

1. 🗂️ NA ÁRVORE DE TABELAS:
   - Expanda seu banco de dados
   - Expanda "Tables"
   - Clique com botão direito em "alunos"
   - Selecione "Alter Table"

2. ⚙️ CONFIGURAR COLUNA ID:
   - Aba "Columns"
   - Selecione a linha da coluna "id"
   - Marque as opções:
     ✓ [x] Primary Key
     ✓ [x] Not Null
     ✓ [x] Auto Increment

3. 💾 SALVAR:
   - Clique em "Save"
   - REPITA para as outras tabelas
   - Depois clique em "Save & Apply" em TODAS

✅ VERIFICAÇÃO FINAL
----

1. 🔍 EXECUTE NOVA CONSULTA:

```
SELECT
    TABLE_NAME,
    COLUMN_NAME,
    COLUMN_KEY,
    EXTRA
FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_SCHEMA = DATABASE()
AND TABLE_NAME IN ('alunos', 'componentes', 'emprestimos')
AND COLUMN_NAME = 'id';
```

2. 🎯 RESULTADO ESPERADO:

```
alunos       id    PRI    auto_increment
componentes  id    PRI    auto_increment
emprestimos  id    PRI    auto_increment
```

3. 🧪 TESTE PRÁTICO:

```
-- Inserir registro (deve gerar ID automaticamente)
INSERT INTO alunos (nome, matricula) VALUES ('Teste Silva', '999999');
SELECT * FROM alunos WHERE matricula = '999999';

-- Verificar se o ID foi gerado automaticamente
```

🆘 SOLUÇÃO DE PROBLEMAS

❌ ERRO: "Duplicate entry"
   → Há IDs duplicados na tabela
   → Execute primeiro:

```
  CREATE TABLE backup_alunos SELECT * FROM alunos;
  DELETE FROM alunos WHERE id IN (
    SELECT id FROM (
      SELECT id, COUNT(*) as count
      FROM alunos
      GROUP BY id
      HAVING count > 1
    ) as dup
  );
```

❌ ERRO: "Incorrect table definition"
   → Coluna id não existe ou tem tipo diferente

```
   → Verifique com: DESC nome_da_tabela;

❌ ERRO: "Table doesn't exist"
   → Nome da tabela está errado
   → Verifique os nomes exatos com: SHOW TABLES;

📞 BACKUP E SEGURANÇA
----------------------

1. 📋 DBeaver faz backup automático das queries
2. 🔄 Para recuperar: "Window" → "History"
3. 💾 Backup manual rápido:
   -- Backup das tabelas
   CREATE TABLE alunos_backup SELECT * FROM alunos;
   CREATE TABLE componentes_backup SELECT * FROM componentes;
   CREATE TABLE emprestimos_backup SELECT * FROM emprestimos;

🎉 SUCESSO!
------------

Se todas as verificações mostrarem:
✅ COLUMN_KEY = 'PRI'
✅ EXTRA = 'auto_increment'

Seu sistema está 100% configurado com chaves primárias!
As tabelas agora têm IDs únicos e auto-gerados.
```

Apêndice B: Comandos SQL para Criação das Tabelas

Para referência rápida, seguem os comandos SQL completos para criar todas as tabelas do sistema:

SQL

```sql
-- Criar banco de dados (se necessário)
CREATE DATABASE IF NOT EXISTS site_eletronicaX CHARACTER SET utf8mb4 COLLATE
utf8mb4_unicode_ci;
USE site_eletronicaX;

-- Tabela de alunos
CREATE TABLE IF NOT EXISTS alunos (
    id INT NOT NULL AUTO_INCREMENT,
    matricula INT NOT NULL,
    nome VARCHAR(255) NOT NULL,
    email VARCHAR(255),
    turma INT,
    foto LONGBLOB,
```

```
    PRIMARY KEY (id),
    UNIQUE KEY (matricula)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

-- Tabela de componentes
CREATE TABLE IF NOT EXISTS componentes (
    id INT NOT NULL AUTO_INCREMENT,
    nome VARCHAR(255) NOT NULL,
    PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

-- Tabela de empréstimos
CREATE TABLE IF NOT EXISTS emprestimos (
    id INT NOT NULL AUTO_INCREMENT,
    aluno_id INT NOT NULL,
    componente_id INT NOT NULL,
    quantidade INT NOT NULL DEFAULT 1,
    data_emp DATETIME NOT NULL,
    status VARCHAR(50) NOT NULL DEFAULT 'Emprestado',
    data_dev DATETIME NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (aluno_id) REFERENCES alunos(id) ON DELETE CASCADE,
    FOREIGN KEY (componente_id) REFERENCES componentes(id) ON DELETE CASCADE,
    INDEX idx_aluno (aluno_id),
    INDEX idx_componente (componente_id),
    INDEX idx_status (status)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

Apêndice C: Requisitos do Sistema

Requisitos Mínimos:

- Sistema Operacional: Windows 7 ou superior, macOS 10.12+, ou Linux (Ubuntu 18.04+)
- Processador: Intel Core i3 ou equivalente
- Memória RAM: 4 GB
- Espaço em Disco: 500 MB para a aplicação + espaço para fotos
- Conexão de Rede: Acesso à rede intranet da instituição

Software Necessário:

- Python 3.11 ou superior
- MySQL Server 5.7 ou superior
- DBeaver Community Edition (última versão)
- VS Code (opcional, mas recomendado)

Bibliotecas Python:

- mysql-connector-python >= 8.0.0
- PyQt6 >= 6.0.0
- pytz >= 2021.1
- pandas >= 1.3.0
- requests >= 2.26.0

Apêndice D: Glossário de Termos Técnicos

API (Application Programming Interface): Interface de Programação de Aplicações, conjunto de definições e protocolos para construir e integrar software.

AUTO_INCREMENT: Propriedade de uma coluna de banco de dados que gera automaticamente um valor único e crescente para cada novo registro.

BLOB (Binary Large Object): Tipo de dado usado para armazenar grandes quantidades de dados binários, como imagens ou arquivos.

Charset: Conjunto de caracteres usado para codificar texto. UTF-8 é o padrão moderno que suporta todos os idiomas.

Commit: Operação que confirma e torna permanentes as alterações feitas em uma transação de banco de dados.

Cursor: Objeto que permite executar comandos SQL e recuperar resultados de um banco de dados.

DataFrame: Estrutura de dados tabular da biblioteca pandas, similar a uma planilha.

Foreign Key (Chave Estrangeira): Coluna que estabelece uma relação entre duas tabelas, referenciando a chave primária de outra tabela.

GUI (Graphical User Interface): Interface Gráfica do Usuário, permite interação através de elementos visuais.

Logging: Sistema de registro de eventos e erros que ocorrem durante a execução de um programa.

Magic Number (Número Mágico): Sequência de bytes no início de um arquivo que identifica seu formato.

ORM (Object-Relational Mapping): Técnica de programação que converte dados entre sistemas incompatíveis usando programação orientada a objetos.

Primary Key (Chave Primária): Coluna ou conjunto de colunas que identifica unicamente cada registro em uma tabela.

Query: Consulta ou comando SQL usado para recuperar ou manipular dados em um banco de dados.

Widget: Elemento de interface gráfica, como botões, campos de texto, listas, etc.

Fim do Manual Técnico

Versão: 1.0

Data: 2025

Autor: Documentação preparada para Lourenço - Fundação Liberato

Contato: Para dúvidas ou sugestões sobre este manual, entre em contato com a equipe de TI da instituição.