Sistema de Agendamento da UBS

Este é um sistema simples de console (linha de comando) para gerenciar o cadastro de pacientes em uma Unidade Básica de Saúde (UBS). Ele permite criar, listar, modificar e excluir registros de pacientes, interagindo com um banco de dados MySQL.

Instruções de Execução

1. Configuração do Banco de Dados (MySQL)

Para inicializar o banco de dados localmente, siga estes passos:

1. Instale o MySQL Server (ou utilize XAMPP/Docker).
2. Abra seu cliente de banco de dados (DBeaver, MySQL Workbench, etc.).
3. **Execute o script SQL:** Carregue e execute o arquivo `database_schema.sql` completo. Este script irá criar o schema `ubs_agendamento` e todas as tabelas necessárias.

2. Configuração do Python

1. **Instale as dependências:** `pip install mysql-connector-python python-dotenv`
2. **Configure as variáveis de ambiente:** Crie um arquivo chamado `.env` na raiz do projeto e adicione suas credenciais do banco de dados, como no exemplo abaixo. Isso evita que informações sensíveis fiquem expostas no código.

```
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=sua_senha_aqui
DB_DATABASE=ubs_agendamento
```

3. **Execute a aplicação:** `python app.py`

Estrutura do Projeto

O projeto está dividido em arquivos com responsabilidades bem definidas para facilitar a manutenção e o entendimento do código:

- `README.md`: Este arquivo, com a documentação do projeto.
- `database_schema.sql`: Script de inicialização do banco de dados.
- `db_connector.py`: Gerencia os detalhes e as funções de conexão com o banco de dados.
- `data_manager.py`: Contém toda a lógica para interagir com o banco de dados (operações CRUD). É a camada de acesso a dados.
- `app.py`: Responsável pela interface do usuário (o menu de console) e pela coleta de dados. É a camada de apresentação.
- `models.py`: (Atualmente vazio) Arquivo destinado a futuras validações e modelos de dados mais complexos.

1. Conector do Banco de Dados (`db_connector.py`)

Este script isola a configuração e a lógica de conexão com o banco de dados MySQL. Para maior segurança, ele foi adaptado para ler as credenciais de um arquivo `.env`, evitando que dados sensíveis sejam expostos diretamente no código.

Código (`db_connector.py`)

```
# 2. Conexão MySQL
import mysql.connector
from mysql.connector import Error
import os
from dotenv import load_dotenv

# Carrega as variáveis de ambiente do arquivo .env
load_dotenv()

# --- Configurações do Banco de Dados ---
# As credenciais são lidas das variáveis de ambiente
DB_CONFIG = {
    'host': os.getenv('DB_HOST'),
    'user': os.getenv('DB_USER'),
    'password': os.getenv('DB_PASSWORD'),
    'database': os.getenv('DB_DATABASE')
}

def conectar_bd():
    """Tenta estabelecer a conexão com o banco de dados."""
    try:
        conn = mysql.connector.connect(**DB_CONFIG)
        if conn.is_connected():
            print("✅ Conexão com o MySQL estabelecida com sucesso!")
            return conn
        else:
            print("❌ Falha na conexão com o banco de dados.")
            return None
    except Error as e:
        print(f"❌ Erro ao conectar ao MySQL: {e}")
        return None

def fechar_conexao(conn):
    """Fecha a conexão com o banco de dados."""
    if conn and conn.is_connected():
        conn.close()
        print("🔌 Conexão com o MySQL fechada.")
```

Explicação

- `DB_CONFIG`: Um dicionário que armazena as credenciais de acesso. Manter isso separado do resto do código é uma boa prática.

- `conectar_bd()`: Tenta criar uma conexão usando as configurações do `DB_CONFIG`. Retorna o objeto de conexão em caso de sucesso ou `None` em caso de falha.
- `fechar_conexao()`: Verifica se a conexão existe e está ativa antes de fechá-la, evitando erros.

---

2. Gerenciador de Dados (`data_manager.py`)

Este arquivo é o coração da lógica de negócios que interage com o banco de dados. Ele é responsável por todas as operações de **CRUD** (Criar, Ler, Atualizar, Deletar) relacionadas aos pacientes, utilizando as funções do `db_connector.py`.

Código (`data_manager.py`)

```
# 3. Lógica CRUD/SQL
import mysql.connector
from db_connector import conectar_bd, fechar_conexao
from mysql.connector import Error

def criar_paciente(nome, cpf, data_nascimento, telefone, endereco=None):
    """Insere um novo registro de paciente na tabela Pacientes."""
    conexao = conectar_bd()
    if conexao is None:
        print("❌ Falha na operação: Sem conexão com o banco de dados.")
        return False

    cursor = conexao.cursor()
    sql = """
        INSERT INTO Pacientes (nome, cpf, data_nascimento, telefone, endereco)
        VALUES (%s, %s, %s, %s, %s)
    """
    valores = (nome, cpf, data_nascimento, telefone, endereco)

    try:
        cursor.execute(sql, valores)
        conexao.commit()
        novo_id = cursor.lastrowid
        print(f"✅ Paciente '{nome}' cadastrado com sucesso! ID: {novo_id}")
        return novo_id
    except mysql.connector.Error as e:
        print(f"❌ Falha ao cadastrar Paciente: {e}")
        conexao.rollback()
        return False
    finally:
        cursor.close()
        fechar_conexao(conexao)

def listar_pacientes():
    """Busca todos os registros de pacientes no banco de dados."""
    conexao = conectar_bd()
    if conexao is None:
        return []
```

```
    cursor = conexao.cursor(dictionary=True)
    sql = "SELECT paciente_id, nome, cpf, data_nascimento, telefone FROM
Pacientes"

    try:
        cursor.execute(sql)
        pacientes = cursor.fetchall()
        return pacientes
    except mysql.connector.Error as e:
        print(f"❌ Falha ao listar Pacientes: {e}")
        return []
    finally:
        cursor.close()
        fechar_conexao(conexao)

def buscar_paciente_por_id(paciente_id):
    """Busca um único paciente pelo ID."""
    conexao = conectar_bd()
    if conexao is None:
        return None

    cursor = conexao.cursor(dictionary=True)
    sql = "SELECT paciente_id, nome, cpf, data_nascimento, telefone, endereco FROM
Pacientes WHERE paciente_id = %s"

    try:
        cursor.execute(sql, (paciente_id,))
        paciente = cursor.fetchone()
        return paciente
    except mysql.connector.Error as e:
        print(f"❌ Falha ao buscar Paciente por ID: {e}")
        return None
    finally:
        cursor.close()
        fechar_conexao(conexao)

def atualizar_paciente(paciente_id, nome, cpf, data_nascimento, telefone,
endereco=None):
    """Executa o comando SQL UPDATE para modificar um paciente existente."""
    conexao = conectar_bd()
    if conexao is None:
        return False

    cursor = conexao.cursor()
    sql = """
        UPDATE Pacientes SET nome = %s, cpf = %s, data_nascimento = %s, telefone =
%s, endereco = %s
        WHERE paciente_id = %s
    """
    valores = (nome, cpf, data_nascimento, telefone, endereco, paciente_id)

    try:
        cursor.execute(sql, valores)
        conexao.commit()
```

```
        if cursor.rowcount > 0:
            print(f"✅ Paciente ID {paciente_id} atualizado com sucesso!")
            return True
        else:
            print(f"⚠ Paciente ID {paciente_id} não encontrado para
atualização.")
            return False
    except mysql.connector.Error as e:
        print(f"❌ Falha ao atualizar Paciente: {e}")
        conexao.rollback()
        return False
    finally:
        cursor.close()
        fechar_conexao(conexao)

def excluir_paciente(paciente_id):
    """Executa o comando SQL DELETE para remover um paciente existente."""
    conexao = conectar_bd()
    if conexao is None:
        return False

    cursor = conexao.cursor()
    sql = "DELETE FROM Pacientes WHERE paciente_id = %s"

    try:
        cursor.execute(sql, (paciente_id,))
        conexao.commit()
        if cursor.rowcount > 0:
            print(f"✅ Paciente ID {paciente_id} excluído com sucesso!")
            return True
        else:
            print(f"⚠ Paciente ID {paciente_id} não encontrado.")
            return False
    except mysql.connector.Error as e:
        print(f"❌ Falha ao excluir Paciente: {e}")
        conexao.rollback()
        return False
    finally:
        cursor.close()
        fechar_conexao(conexao)
```

Explicação

- **Padrão `try...except...finally`**: Todas as funções seguem este padrão para garantir que a conexão com o banco de dados seja sempre fechada, mesmo que ocorram erros durante a transação.
- **`commit()` e `rollback()`**: `commit()` salva as alterações (INSERT, UPDATE, DELETE), enquanto `rollback()` as desfaz em caso de erro, mantendo a integridade dos dados.
- **`cursor(dictionary=True)`**: Usado nas funções de leitura (`listar`, `buscar`) para retornar os dados como dicionários Python, facilitando o acesso por nome de coluna (ex: `paciente['nome']`).
- **Cláusula `WHERE`**: Essencial nas funções `atualizar_paciente` e `excluir_paciente` para garantir que a operação afete apenas o registro correto.

3. Interface do Usuário (`app.py`)

Este arquivo é a camada de apresentação do sistema. Ele exibe o menu principal, interage com o usuário, coleta as informações e chama as funções apropriadas do `data_manager.py` para executar as ações.

Código (`app.py`)

```
# 1. Interface do Usuário (Menu)

from data_manager import criar_paciente, listar_pacientes, buscar_paciente_por_id,
atualizar_paciente, excluir_paciente
from datetime import datetime

def validar_e_formatar_data(data_str):
    """Tenta converter a data de DD/MM/AAAA para o formato MySQL AAAA-MM-DD."""
    try:
        data_obj = datetime.strptime(data_str, '%d/%m/%Y')
        data_sql = data_obj.strftime('%Y-%m-%d')
        return data_sql
    except ValueError:
        print("\n❌ Formato de Data de Nascimento inválido. Use o formato
DD/MM/AAAA.")
        return None

def cadastrar_paciente():
    """Lógica para coletar e cadastrar um novo paciente."""
    # ... (coleta de dados do usuário) ...
    if nome and cpf and telefone: # Checagem básica
        criar_paciente(nome, cpf, data_nasc_sql, telefone, endereco)
    # ...

def exibir_pacientes():
    """Busca e exibe a lista de pacientes de forma formatada."""
    # ... (formatação da tabela de exibição) ...

def modificar_paciente():
    """Lógica para buscar, coletar novos dados e atualizar um paciente."""
    # ... (busca paciente por ID, coleta novos dados, chama a atualização) ...

def remover_paciente():
    """Lógica para solicitar o ID, confirmar e excluir um paciente."""
    # ... (busca paciente por ID, pede confirmação, chama a exclusão) ...

def menu_principal():
    """Exibe o menu principal do sistema e gerencia a navegação."""
    while True:
        print("\n=== Sistema UBS Agendamento ===")
        print("1. Cadastrar Novo Paciente")
        print("2. Listar Pacientes")
        print("3. Modificar Paciente")
        print("4. Remover Paciente")
```

```
        print("5. Gerenciar Profissionais")
        print("6. Agendar Consulta")
        print("7. Sair")

        opcao = input("Escolha uma opção: ")

        if opcao == '1':
            cadastrar_paciente()
        elif opcao == '2':
            exibir_pacientes()
        elif opcao == '3':
            modificar_paciente()
        elif opcao == '4':
            remover_paciente()
        elif opcao == '7':
            print("Encerrando o sistema. Até logo!")
            break
        else:
            print("Opção inválida. Tente novamente.")

# Execução do programa
if __name__ == "__main__":
    menu_principal()
```

Explicação

- `menu_principal()`: É o loop principal da aplicação. Ele exibe as opções e, com base na escolha do usuário, chama a função correspondente.
- **Funções de Ação (`cadastrar_paciente`, `exibir_pacientes`, etc.)**: Cada uma dessas funções tem uma responsabilidade clara: interagir com o usuário para uma tarefa específica e depois delegar a lógica de banco de dados para o `data_manager`.
- `validar_e_formatar_data()`: Uma função utilitária que mostra como a camada de interface é responsável por tratar e validar a entrada do usuário antes de enviá-la para as camadas inferiores.