Arquivo: database.py

```
import json
import sqlite3
import pandas as pd
import os
import re
from datetime import datetime
from pathlib import Path
from PyQt6.QtWidgets import QFileDialog
from PyQt6.QtWidgets import QMessageBox
import logging
import num2words
import locale
import pandas as pd
import re
locale.setlocale(locale.LC_ALL, 'pt_BR.UTF-8')

def formatar_valor_monetario(valor):
    if pd.isna(valor):  # Verifica se o valor é NaN e ajusta para string vazia
        valor = ''
    # Limpa a string e converte para float
    valor = re.sub(r'[^\d,]', '', str(valor)).replace(',', '.')
    valor_float = float(valor) if valor else 0
    # Formata para a moeda local sem usar locale
      valor_monetario = f"R$ {valor_float:,.2f}".replace(",", "X").replace(".", ",").replace("X",
".")
    # Converte para extenso
    valor_extenso = num2words.num2words(valor_float, lang='pt_BR', to='currency')
    return valor_monetario, valor_extenso

def remover_caracteres_especiais(texto):
    mapa_acentos = {
        'á': 'a', 'à': 'a', 'ã': 'a', 'â': 'a', 'ä': 'a',
        'Á': 'A', 'À': 'A', 'Ã': 'A', 'Â': 'A', 'Ä': 'A',
        'é': 'e', 'è': 'e', 'ê': 'e', 'ë': 'e',
        'É': 'E', 'È': 'E', 'Ê': 'E', 'Ë': 'E',
        'í': 'i', 'ì': 'i', 'î': 'i', 'ï': 'i',
        'Í': 'I', 'Ì': 'I', 'Î': 'I', 'Ï': 'I',
        'ó': 'o', 'ò': 'o', 'õ': 'o', 'ô': 'o', 'ö': 'o',
        'Ó': 'O', 'Ò': 'O', 'Õ': 'O', 'Ô': 'O', 'Ö': 'O',
        'ú': 'u', 'ù': 'u', 'û': 'u', 'ü': 'u',
        'Ú': 'U', 'Ù': 'U', 'Û': 'U', 'Ü': 'U',
        'ç': 'c', 'Ç': 'C', 'ñ': 'n', 'Ñ': 'N'
    }
    for caractere_original, caractere_novo in mapa_acentos.items():
        texto = texto.replace(caractere_original, caractere_novo)

    # Adicionando substituição para caracteres impeditivos em nomes de arquivos e pastas
    caracteres_impeditivos = r'\\/:*?"<>|'
    for caractere in caracteres_impeditivos:
        texto = texto.replace(caractere, '-')
```

```
    return texto

class DatabaseManager:
    def __init__(self, db_path):
        self.db_path = db_path
        self.connection = None
        logging.basicConfig(level=logging.INFO, filename='app.log', filemode='a',
                            format='%(name)s - %(levelname)s - %(message)s')

    def set_database_path(self, db_path):
        self.db_path = db_path  # Atualiza o caminho do banco dinamicamente

    def __enter__(self):
        self.connection = self.connect_to_database()
        return self.connection

    def connect_to_database(self):
        try:
            connection = sqlite3.connect(self.db_path)
            return connection
        except sqlite3.Error as e:
            logging.error(f"Failed to connect to database at {self.db_path}: {e}")
            raise

    def execute_query(self, query, params=None):
        with self.connect_to_database() as conn:
            try:
                cursor = conn.cursor()
                if params:
                    cursor.execute(query, params)
                else:
                    cursor.execute(query)
                return cursor.fetchall()
            except sqlite3.Error as e:
                logging.error(f"Error executing query: {query}, Error: {e}")
                return None

    def execute_update(self, query, params=None):
        with self.connect_to_database() as conn:
            try:
                cursor = conn.cursor()
                if params:
                    cursor.execute(query, params)
                else:
                    cursor.execute(query)
                conn.commit()
            except sqlite3.Error as e:
                logging.error(f"Error executing update: {query}, Error: {e}")
                return False
            return True

    def close_connection(self):
```

```
        if self.connection:
            self.connection.close()

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.close_connection()

    def consultar_registro(self, tabela, campo, valor):
        """
        Consulta um registro na tabela especificada com base no campo e valor fornecidos.

        :param tabela: Nome da tabela no banco de dados.
        :param campo: Nome do campo a ser filtrado (ex: 'cnpj').
        :param valor: Valor a ser buscado no campo.
        :return: Dicionário com os dados do registro, ou None se não encontrado.
        """
        query = f"SELECT * FROM {tabela} WHERE {campo} = ?"
        print(f"Executando consulta: {query} com valor: {valor}")

        resultado = self.execute_query(query, (valor,))

        # Verifica se o resultado da consulta está vazio
        if resultado:
            print(f"Resultado da consulta encontrado: {resultado}")
            try:
                # Obtenção dos nomes das colunas para transformar o resultado em dicionário
                with self.connect_to_database() as conn:
                    cursor = conn.cursor()
                    cursor.execute(query, (valor,))
                    colunas = [desc[0] for desc in cursor.description]

                print(f"Colunas da tabela: {colunas}")
                registro_dict = dict(zip(colunas, resultado[0]))
                print(f"Registro encontrado: {registro_dict}")
                return registro_dict

            except sqlite3.Error as e:
                print(f"Erro ao transformar o resultado em dicionário: {e}")
                logging.error(f"Erro ao transformar o resultado em dicionário: {e}")
                return None

        print(f"Nenhum registro encontrado para {campo} = {valor} na tabela {tabela}.")
        logging.info(f"Nenhum registro encontrado para {campo} = {valor} na tabela {tabela}.")
        return None


    @staticmethod
    def verify_and_create_columns(conn, table_name, required_columns):
        cursor = conn.cursor()
        cursor.execute(f"PRAGMA table_info({table_name})")
         existing_columns = {row[1]: row[2] for row in cursor.fetchall()}  # Storing column names
and types

        # Criar uma lista das colunas na ordem correta e criar as colunas que faltam
```

```
        for column, column_type in required_columns.items():
            if column not in existing_columns:
                # Assume a default type if not specified, e.g., TEXT
                cursor.execute(f"ALTER TABLE {table_name} ADD COLUMN {column} {column_type}")
                logging.info(f"Column {column} added to {table_name} with type {column_type}")
            else:
                # Check if the type matches, if not, you might handle or log this situation
                if existing_columns[column] != column_type:
                       logging.warning(f"Type mismatch for {column}: expected {column_type}, found
{existing_columns[column]}")

        conn.commit()
        logging.info(f"All required columns are verified/added in {table_name}")

    def get_tables_with_keyword(self, keyword, db_path=None):
        """Retorna uma lista de tabelas que contêm o 'keyword' no nome."""
        db_path = db_path or self.db_path  # Usa o caminho padrão se `db_path` não for fornecido
        try:
            with sqlite3.connect(db_path) as conn:
                cursor = conn.cursor()
                  cursor.execute("SELECT name FROM sqlite_master WHERE type='table' AND name LIKE
?", ('%' + keyword + '%',))
                tables = [row[0] for row in cursor.fetchall()]
            return tables
        except Exception as e:
            QMessageBox.warning(None, "Erro", f"Erro ao obter tabelas do banco de dados: {e}")
            return []


    def load_table_to_dataframe(self, table_name, db_path=None):
        """Carrega a tabela especificada em um DataFrame."""
        db_path = db_path or self.db_path  # Usa o caminho padrão se `db_path` não for fornecido
        try:
            with sqlite3.connect(db_path) as conn:
                df = pd.read_sql_query(f"SELECT * FROM [{table_name}]", conn)
            return df
        except Exception as e:
            QMessageBox.warning(None, "Erro", f"Erro ao carregar a tabela '{table_name}': {e}")
            return None
```