

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

def salvar_consulta_api_no_db(self, data_informacoes):
    # Função salva os dados no banco de dados, verificando se já existe uma entrada
    with self.connect_to_database() as conn:
        cursor = conn.cursor()
        # Verificar e criar a tabela se necessário
        cursor.execute("""
CREATE TABLE IF NOT EXISTS pregoes_consultados (
    valorTotalEstimado REAL,
    valorTotalHomologado REAL,
    orcamentoSigilosoCodigo INTEGER,
    orcamentoSigilosoDescricao TEXT,
    numeroControlePNCp TEXT PRIMARY KEY,
    linkSistemaOrigem TEXT,
        """)

```

```

linkProcessoEletronico TEXT,
anoCompra INTEGER,
sequencialCompra INTEGER,
numeroCompra TEXT,
processo TEXT,
orgaoEntidadeCnpj TEXT,
orgaoEntidadeRazaoSocial TEXT,
orgaoEntidadeEsferaId TEXT,
orgaoEntidadePoderId TEXT,
unidadeOrgaoCodigoUnidade TEXT,
unidadeOrgaoNomeUnidade TEXT,
unidadeOrgaoMunicipioNome TEXT,
unidadeOrgaoCodigoIbge TEXT,
unidadeOrgaoUfSigla TEXT,
unidadeOrgaoUfNome TEXT,
modalidadeId INTEGER,
modalidadeNome TEXT,
justificativaPresencial TEXT,
modoDisputaId INTEGER,
modoDisputaNome TEXT,
tipoInstrumentoConvocatorioCodigo INTEGER,
tipoInstrumentoConvocatorioNome TEXT,
amparoLegalCodigo INTEGER,
amparoLegalNome TEXT,
amparoLegalDescricao TEXT,
objetoCompra TEXT,
informacaoComplementar TEXT,
srp BOOLEAN,
dataPublicacaoPncc TEXT,
dataAberturaProposta TEXT,
dataEncerramentoProposta TEXT,
situacaoCompraId INTEGER,
situacaoCompraNome TEXT,
existeResultado BOOLEAN,
dataInclusao TEXT,
dataAtualizacao TEXT,
usuarioNome TEXT
)
"""
# Inserir ou atualizar a entrada no banco de dados
cursor.execute("""
    INSERT INTO pregoes_consultados (
        valorTotalEstimado, valorTotalHomologado, orcamentoSigilosoCodigo,
        orcamentoSigilosoDescricao,
        numeroControlePNCP, linkSistemaOrigem, linkProcessoEletronico, anoCompra,
        sequencialCompra,
        numeroCompra, processo, orgaoEntidadeCnpj, orgaoEntidadeRazaoSocial,
        orgaoEntidadeEsferaId,
        orgaoEntidadePoderId, unidadeOrgaoCodigoUnidade, unidadeOrgaoNomeUnidade,
        unidadeOrgaoMunicipioNome,
        unidadeOrgaoCodigoIbge, unidadeOrgaoUfSigla, unidadeOrgaoUfNome, modalidadeId,
        modalidadeNome,
        justificativaPresencial, modoDisputaId, modoDisputaNome,

```



```

        usuarioNome = excluded.usuarioNome
        """, (
            data_informacoes.get("valorTotalEstimado"),
            data_informacoes.get("valorTotalHomologado"),
            data_informacoes.get("orcamentoSigilosoCodigo"),
            data_informacoes.get("orcamentoSigilosoDescricao"),
            data_informacoes.get("numeroControlePNCP"),
            data_informacoes.get("linkSistemaOrigem"),
            data_informacoes.get("linkProcessoEletronico"),
            data_informacoes.get("anoCompra"),
            data_informacoes.get("sequentialCompra"),
            data_informacoes.get("numeroCompra"),
            data_informacoes.get("processo"),
            data_informacoes.get("orgaoEntidade", {}).get("cnpj"),
            data_informacoes.get("orgaoEntidade", {}).get("razaoSocial"),
            data_informacoes.get("orgaoEntidade", {}).get("esferaId"),
            data_informacoes.get("orgaoEntidade", {}).get("poderId"),
            data_informacoes.get("unidadeOrgao", {}).get("codigoUnidade"),
            data_informacoes.get("unidadeOrgao", {}).get("nomeUnidade"),
            data_informacoes.get("unidadeOrgao", {}).get("municipioNome"),
            data_informacoes.get("unidadeOrgao", {}).get("codigoIbge"),
            data_informacoes.get("unidadeOrgao", {}).get("ufSigla"),
            data_informacoes.get("unidadeOrgao", {}).get("ufNome"),
            data_informacoes.get("modalidadeId"),
            data_informacoes.get("modalidadeNome"),
            data_informacoes.get("justificativaPresencial"),
            data_informacoes.get("modoDisputaId"),
            data_informacoes.get("modoDisputaNome"),
            data_informacoes.get("tipoInstrumentoConvocatorioCodigo"),
            data_informacoes.get("tipoInstrumentoConvocatorioNome"),
            data_informacoes.get("amparoLegal", {}).get("codigo"),
            data_informacoes.get("amparoLegal", {}).get("nome"),
            data_informacoes.get("amparoLegal", {}).get("descricao"),
            data_informacoes.get("objetoCompra"),
            data_informacoes.get("informacaoComplementar"),
            data_informacoes.get("srp"),
            data_informacoes.get("dataPublicacaoPnlp"),
            data_informacoes.get("dataAberturaProposta"),
            data_informacoes.get("dataEncerramentoProposta"),
            data_informacoes.get("situacaoCompraId"),
            data_informacoes.get("situacaoCompraNome"),
            data_informacoes.get("existeResultado"),
            data_informacoes.get("dataInclusao"),
            data_informacoes.get("dataAtualizacao"),
            data_informacoes.get("usuarioNome")
        ))
    conn.commit()

def criar_tabela_itens_pregao(self, numeroCompra, anoCompra, unidadeOrgaoCodigoUnidade):
    table_name = f'{numeroCompra}-{anoCompra}-{unidadeOrgaoCodigoUnidade}-HomologAPI'
    column_order = [
        'grupo', 'item PRIMARY KEY', 'catalogo', 'descricao', 'unidade', 'quantidade',
        'valor_estimado',

```

```

        'valor_homologado_item_unitario',      'percentual_desconto',
'valor_estimado_total_do_item',
        'valor_homologado_total_item', 'marca_fabricante', 'modelo_versao', 'situacao',
        'descricao_detalhada', 'uasg', 'orgao_responsavel', 'num_pregao', 'ano_pregao',
        'srp', 'objeto', 'melhor_lance', 'valor_negociado', 'ordenador_despesa', 'empresa',
        'cnpj', 'endereco', 'cep', 'municipio', 'telefone', 'email', 'responsavel_legal'
]

with self.connect_to_database() as conn:
    cursor = conn.cursor()
    columns_definition = ", ".join(column_order)
    create_table_query = f"CREATE TABLE IF NOT EXISTS '{table_name}'"
    ({columns_definition})
    cursor.execute(create_table_query)
    conn.commit()

def popular_db_consulta_itens_api(self, resultados_completos, data_informacoes, numeroCompra,
anoCompra, unidadeOrgaoCodigoUnidade):
    table_name = f"{numeroCompra}-{anoCompra}-{unidadeOrgaoCodigoUnidade}-HomologAPI"

    # Informações gerais de data_informacoes que serão inseridas com cada item
    data_informacoes_to_insert = {
        "ano_pregao": data_informacoes.get("anoCompra"),
        "num_pregao": data_informacoes.get("numeroCompra"),
        "uasg": data_informacoes.get("unidadeOrgao", {}).get("codigoUnidade"),
        "orgao_responsavel": data_informacoes.get("unidadeOrgao", {}).get("nomeUnidade"),
        "objeto": data_informacoes.get("objetoCompra"),
        "srp": data_informacoes.get("srp")
    }

    # Para cada item em resultados_completos, insere ou atualiza os dados no banco de dados
    for item in resultados_completos:
        quantidade = item.get("quantidadeHomologada", 0) or 0
        valor_estimado = item.get("valorUnitarioEstimado", 0) or 0
        valor_homologado_item_unitario = item.get("valorUnitarioHomologado", 0) or 0

        # Cálculos necessários
        percentual_desconto = (
            (valor_estimado - valor_homologado_item_unitario) / valor_estimado * 100
            if valor_estimado else 0
        )
        valor_homologado_total_item = quantidade * valor_homologado_item_unitario
        valor_estimado_total_do_item = quantidade * valor_estimado

        # Determina a situação com base no valor booleano
        situacao = 'Adjudicado e Homologado' if item.get("temResultado") == 1 else
        'Fracassado/Deserto/Cancelado ou Anulado'

        # Combina dados específicos do item com as informações gerais e os cálculos
        data_to_insert = {
            "item": item.get("numeroItem"),
            "descricao": item.get("descricao"),
            "descricao_detalhada": item.get("descricao"),

```

```
"unidade": item.get("unidadeMedida"),
"quantidade": quantidade,
"valor_estimado": valor_estimado,
"valor_homologado_item_unitario": valor_homologado_item_unitario,
"percentual_desconto": percentual_desconto,
"valor_homologado_total_item": valor_homologado_total_item,
"valor_estimado_total_do_item": valor_estimado_total_do_item,
"situacao": situacao, # Adiciona o valor convertido de situacao
"cnpj": item.get("niFornecedor"),
"empresa": item.get("nomeRazaoSocialFornecedor"),
**data_informacoes_to_insert # Adiciona as informaçoes de data_informacoes
}

# Comando SQL de INSERT OR REPLACE para atualizar caso item já exista
placeholders = ", ".join(["?"] * len(data_to_insert))
columns = ", ".join(data_to_insert.keys())
insert_query = f"INSERT OR REPLACE INTO '{table_name}' ({columns}) VALUES
({placeholders})"

# Executa a inserção ou atualização no banco de dados
self.execute_update(insert_query, tuple(data_to_insert.values()))
```