Arquivo: api_consulta.py

```
import requests
import json
from pathlib import Path
from datetime import datetime, timedelta
from PyQt6.QtWidgets import *
from PyQt6.QtCore import QThread, pyqtSignal
from PyQt6.QtGui import *
import sqlite3
import re
from pathlib import Path
import time

COLUNAS_OBRIGATORIAS = [
    "criterioJulgamentoNome", "dataAtualizacao", "dataInclusao", "dataResultado",
    "descricao", "materialOuServico", "materialOuServicoNome", "niFornecedor",
    "nomeRazaoSocialFornecedor", "numeroControlePNCPCompra", "numeroItem",
    "percentualDesconto", "quantidade", "quantidadeHomologada",
    "situacaoCompraItemNome", "situacaoCompraItemResultadoNome",
    "temResultado", "tipoBeneficioNome", "unidadeMedida",
    "valorTotal", "valorTotalHomologado", "valorUnitarioEstimado", "valorUnitarioHomologado"
]

class ConsultaAPIDialog(QDialog):
    consulta_concluida = pyqtSignal(list, list)

    def __init__(self, numero, cnpj, sequencial, ano, uasg, parent=None):
        super().__init__(parent)
        self.setWindowTitle("Consulta API PNCP")
        self.resize(400, 200)
        self.numero = numero
        self.cnpj = cnpj
        self.sequencial = sequencial
        self.ano = ano
        self.uasg = uasg

        # Layout e elementos do diálogo
        layout = QVBoxLayout(self)
        self.progress_label = QLabel("Iniciando consulta...", self)
        self.progress_bar = QProgressBar(self)
        self.cancel_button = QPushButton("Cancelar", self)
        layout.addWidget(self.progress_label)
        layout.addWidget(self.progress_bar)
        layout.addWidget(self.cancel_button)

        self.cancel_button.clicked.connect(self.cancelar_consulta)

        # Inicializa a thread de consulta
             self.thread  =  PNCPConsultaThread(numero=self.numero,  cnpj=self.cnpj,  ano=self.ano,
sequencial=self.sequencial, uasg=self.uasg)
        self.thread.consulta_concluida.connect(self.exibir_resultado)
```

```
        self.thread.erro_consulta.connect(self.exibir_erro)
        self.thread.progresso_consulta.connect(self.atualizar_progresso)
        self.thread.start()

    def atualizar_progresso(self, mensagem, progresso_atual, progresso_total):
        self.progress_label.setText(mensagem)
        self.progress_bar.setMaximum(progresso_total)
        self.progress_bar.setValue(progresso_atual)

    def exibir_resultado(self, data_informacoes_lista, resultados_completos):
        print("Consulta concluída com sucesso.")
        print("Dados da consulta:", data_informacoes_lista)
        print("Resultados completos:", resultados_completos)

        # Emite o sinal `consulta_concluida` com os resultados
        self.consulta_concluida.emit(data_informacoes_lista, resultados_completos)
        self.accept()

    def exibir_erro(self, erro_mensagem):
        print("Erro na consulta:", erro_mensagem)
        self.reject()

    def cancelar_consulta(self):
        if self.thread.isRunning():
            self.thread.terminate()
        self.reject()

class PNCPConsultaThread(QThread):
    consulta_concluida = pyqtSignal(list, list)  # Sinal para retornar dados de consulta
    erro_consulta = pyqtSignal(str)  # Sinal para erros
    progresso_consulta = pyqtSignal(str, int, int)  # Atualizamos para incluir progresso com barra

    def __init__(self, numero, cnpj, ano, sequencial, uasg, parent=None):
        super().__init__(parent)
        self.numero = numero
        self.cnpj = cnpj
        self.ano = ano
        self.sequencial = sequencial
        self.uasg = uasg
        self.json_log_path = Path("consulta_pncp_log.json")  # Define o caminho do arquivo de log

        # # Prints de depuração
        # print(f"Inicializando PNCPConsultaThread com os valores:")
        # print(f"  Número: {self.numero}")
        # print(f"  Ano: {self.ano}")
        # print(f"  Sequencial: {self.sequencial}")
        # print(f"  CNPJ: {self.cnpj}")
        # print(f"  UASG: {self.uasg}")


    def run(self):
        try:
            # Realizar a consulta
```

```
            data_informacoes_lista, resultados_completos = self.consultar_por_sequencial()
            # Emitir o sinal com os dados de consulta
            self.consulta_concluida.emit(data_informacoes_lista, resultados_completos)
            # Salvar os dados para depuração
               self.salvar_json({"status": "sucesso", "dados_informacoes": data_informacoes_lista,
"resultados_completos": resultados_completos})
        except Exception as e:
            # Emitir o erro e salvar em log para depuração
            self.erro_consulta.emit(str(e))
            self.salvar_json({"status": "erro", "mensagem": str(e)})

    def consultar_por_sequencial(self):
        """Consulta a API do PNCP com o CNPJ, ano e sequencial fornecidos."""

        # # Prints de depuração antes da consulta
        # print(f"Executando consulta com os parâmetros:")
        # print(f"  Número: {self.numero}")
        # print(f"  Ano: {self.ano}")
        # print(f"  Sequencial: {self.sequencial}")
        # print(f"  CNPJ: {self.cnpj}")
        # print(f"  UASG: {self.uasg}")

        """Consulta a API do PNCP com o CNPJ, ano e sequencial fornecidos."""
                                                                    url_informacoes         =
f"https://pncp.gov.br/api/pncp/v1/orgaos/{self.cnpj}/compras/{self.ano}/{self.sequencial}"
        tentativas_maximas = 10

        for tentativa in range(1, tentativas_maximas + 1):
            try:
                # Emitir progresso
                       self.progresso_consulta.emit(f"Tentativa {tentativa}/{tentativas_maximas} -
Consultando sequencial {self.sequencial} no PNCP\nUASG: {self.uasg}", 0, 0)

                # Fazer a requisição para obter as informações
                response_informacoes = requests.get(url_informacoes, timeout=20)
                response_informacoes.raise_for_status()
                data_informacoes = response_informacoes.json()

                # Salvar JSON retornado para depuração
                self.salvar_json(data_informacoes, tentativa)

                # Validar dados conforme esperado
                ano_compra = int(data_informacoes.get("anoCompra"))
                numero_compra = str(data_informacoes.get("numeroCompra")).strip()

                if ano_compra != int(self.ano):
                         raise Exception(f"Ano da compra não corresponde: {ano_compra} (esperado:
{self.ano})")

                if numero_compra != str(self.numero).strip():
                     raise Exception(f"Número da compra não corresponde: {numero_compra} (esperado:
{self.numero})")
```

```
                # Converter para lista e retornar
                data_informacoes_lista = self.converter_para_lista(data_informacoes)
                qnt_itens = self.consultar_quantidade_de_itens()
                            resultados_completos  =  self.consultar_detalhes_dos_itens(qnt_itens,
self.progresso_consulta)

                return data_informacoes_lista, resultados_completos

            except requests.exceptions.RequestException as e:
                 self.progresso_consulta.emit(f"Erro na tentativa {tentativa}/{tentativas_maximas}:
{str(e)}", 0, 0)
                if tentativa < tentativas_maximas:
                    time.sleep(2)
                else:
                    raise Exception(f"Falha após {tentativas_maximas} tentativas: {str(e)}")

    def salvar_json(self, data, tentativa=None):
        """Salva o JSON retornado da consulta em um arquivo para análise."""
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        if tentativa is not None:
            filename = f"consulta_pncp_result_tentativa_{tentativa}_{timestamp}.json"
        else:
            filename = f"consulta_pncp_result_erro_{timestamp}.json"

        path = self.json_log_path.parent / filename

        with open(path, "w", encoding="utf-8") as file:
            json.dump(data, file, ensure_ascii=False, indent=4)

    def consultar_quantidade_de_itens(self):
                                                                        url_quantidade         =
f"https://pncp.gov.br/api/pncp/v1/orgaos/{self.cnpj}/compras/{self.ano}/{self.sequencial}/itens/qu
antidade"
        response_quantidade = requests.get(url_quantidade)
        response_quantidade.raise_for_status()
        data_quantidade = response_quantidade.json()

        if isinstance(data_quantidade, int):
            return data_quantidade
        else:
            raise Exception("Resposta inesperada da API para quantidade.")

    def consultar_detalhes_dos_itens(self, qnt_itens, progresso_callback):
        resultados_completos = []

        for i in range(1, qnt_itens + 1):
                                                                            url_item_info      =
f"https://pncp.gov.br/api/pncp/v1/orgaos/{self.cnpj}/compras/{self.ano}/{self.sequencial}/itens/{i
}"
            response_item_info = requests.get(url_item_info)
            response_item_info.raise_for_status()
            data_item_info = response_item_info.json()
```

```
                progresso_callback.emit(f"Verificando item {i}/{qnt_itens}", i, qnt_itens)  # Atualização de progresso

            if data_item_info.get('temResultado', False):
                # Se tem resultado, faz a consulta adicional
                                                                    url_item_resultados = f"https://pncp.gov.br/api/pncp/v1/orgaos/{self.cnpj}/compras/{self.ano}/{self.sequencial}/itens/{i}/resultados"
                response_item_resultados = requests.get(url_item_resultados)
                response_item_resultados.raise_for_status()
                data_item_resultados = response_item_resultados.json()

                if isinstance(data_item_resultados, list):
                    for resultado in data_item_resultados:
                        for key, value in resultado.items():
                            data_item_info[key] = value
            else:
                # Se não há resultado, adicionar 'None' para as chaves esperadas
                    expected_keys = ['dataResultado', 'niFornecedor', 'nomeRazaoSocialFornecedor', 'numeroControlePNCPCompra', 'tipoBeneficioNome']
                for key in expected_keys:
                    data_item_info[key] = None

            # Adiciona o item, seja com resultado ou com valores 'None'
            resultados_completos.append(data_item_info)

        return resultados_completos

    def converter_para_lista(self, dados):
        """
        Converte um dicionário aninhado em uma lista de pares chave: valor.
        Substitui os subdicionários por pares chave: valor com a chave concatenada.
        """
        lista_resultado = []

        def _achatar(sub_dados, chave_pai=""):
            if isinstance(sub_dados, dict):
                for chave, valor in sub_dados.items():
                    nova_chave = f"{chave_pai}.{chave}" if chave_pai else chave
                    if isinstance(valor, (dict, list)):
                        _achatar(valor, nova_chave)
                    else:
                        lista_resultado.append((nova_chave, valor))
            elif isinstance(sub_dados, list):
                for index, item in enumerate(sub_dados):
                    nova_chave = f"{chave_pai}[{index}]" if chave_pai else f"[{index}]"
                    _achatar(item, nova_chave)

        _achatar(dados)
        return lista_resultado
```

```
class PNCPConsulta(object):  # Herdando de QObject
    dados_integrados = pyqtSignal()  # Sinal emitido ao finalizar a integração de dados

    def __init__(self, numero, ano, sequencial, uasg, parent=None):
        super().__init__(parent)  # Inicializa o QObject corretamente
        self.numero = numero
        self.ano = ano
        self.sequencial = sequencial
        self.uasg = uasg
        self.parent = parent
        self.db_path = CONTROLE_DADOS_PNCP

    def consultar(self):
          # Simulação da consulta que retornaria dados JSON (essa parte deve ser adaptada ao seu
contexto real)
        dados_json = [
            {"ano": self.ano, "sequencial": self.sequencial},
            # outros possíveis dados retornados...
        ]
        # Retorna apenas ano e link_pncp
        return [{"ano": item["ano"], "sequencial": item["sequencial"]} for item in dados_json]

    def salvar_json_na_area_de_trabalho(self, json_data, filename):
        # Salva os resultados filtrados no arquivo JSON
        with open(Path.home() / f"{filename}.json", 'w') as file:
            json.dump(json_data, file, indent=4)

    def integrar_dados(self, data_informacoes_lista, resultados_completos):
        # Verificação dos dados recebidos
        print(f"Dados recebidos para salvar no banco:")
        print(f"data_informacoes: {json.dumps(data_informacoes_lista, indent=2)}")
        print(f"resultados_completos: {json.dumps(resultados_completos, indent=2)}")

        # Nomes dinâmicos das tabelas
        table_name_info = f"INFO_DE{self.numero}{self.ano}{self.sequencial}{self.uasg}"
        table_name_resultados = f"DE{self.numero}{self.ano}{self.sequencial}{self.uasg}"

        # Remover caracteres especiais dos nomes das tabelas
        table_name_info = re.sub(r'[^\w]', '_', table_name_info)
        table_name_resultados = re.sub(r'[^\w]', '_', table_name_resultados)

        # Salvar os dados de 'data_informacoes' no banco (dicionário)
        self.salvar_dados_no_banco_lista_tupla(data_informacoes_lista, table_name_info)

        # Salvar os dados de 'resultados_completos' no banco (lista)
        self.salvar_dados_no_banco_lista(resultados_completos, table_name_resultados)

        # Confirmação de sucesso
        QMessageBox.information(self.parent, "Integrar Dados",
                                            f"Os dados foram integrados com sucesso nas tabelas
'{table_name_info}' e '{table_name_resultados}'.")

        # Emitir o sinal após integrar os dados
```

```
        self.dados_integrados.emit()

    def salvar_dados_no_banco_lista(self, dados, nome_tabela):
        try:
            conn = sqlite3.connect(self.db_path)
            cursor = conn.cursor()

            if not dados or len(dados) == 0:
                raise ValueError("Lista de dados está vazia ou inválida.")

            colunas = COLUNAS_OBRIGATORIAS
            colunas_str = ", ".join(colunas)
            valores_placeholder = ", ".join("?" for _ in colunas)

            colunas_definicao = ", ".join([f"{coluna} TEXT" for coluna in colunas])
            cursor.execute(f"CREATE TABLE IF NOT EXISTS {nome_tabela} ({colunas_definicao})")

            for item in dados:
                numero_item = item.get("numeroItem")
                if not numero_item:
                    raise ValueError("O número do item não foi encontrado em alguns dados.")

                valores = [item.get(coluna, None) for coluna in colunas]

                # Verifica se o `numeroItem` já existe
                           cursor.execute(f"SELECT 1 FROM {nome_tabela} WHERE numeroItem = ?",
(numero_item,))
                exists = cursor.fetchone()

                if exists:
                    # Atualiza o registro existente
                    print(f"numeroItem {numero_item} já existe. Sobrescrevendo informações.")
                        update_query = f"UPDATE {nome_tabela} SET {', '.join([f'{coluna} = ?' for
coluna in colunas])} WHERE numeroItem = ?"
                    cursor.execute(update_query, valores + [numero_item])
                else:
                    # Insere um novo registro
                    print(f"numeroItem {numero_item} não existe. Criando novo registro.")
                                cursor.execute(f"INSERT INTO {nome_tabela} ({colunas_str}) VALUES
({valores_placeholder})", valores)

            conn.commit()
            conn.close()

            print(f"Dados salvos com sucesso na tabela: {nome_tabela}")

        except Exception as e:
            print(f"Erro ao salvar os dados (lista): {str(e)}")
                  QMessageBox.critical(self.parent, "Erro", f"Erro ao salvar os dados (lista):
{str(e)}")


    def salvar_dados_no_banco_lista_tupla(self, dados, nome_tabela):
```

```
        try:
            conn = sqlite3.connect(self.db_path)
            cursor = conn.cursor()

            if not dados or len(dados) == 0:
                raise ValueError("Lista de dados está vazia ou inválida.")

            colunas = COLUNAS_OBRIGATORIAS
            colunas_str = ", ".join(colunas)
            valores_placeholder = ", ".join("?" for _ in colunas)

            colunas_definicao = ", ".join([f"{coluna} TEXT" for coluna in colunas])
            cursor.execute(f"CREATE TABLE IF NOT EXISTS {nome_tabela} ({colunas_definicao})")

            # Remove todos os registros antigos antes de inserir os novos (sobrescrever)
            cursor.execute(f"DELETE FROM {nome_tabela}")

            # Organizar os valores como tupla para corresponder às colunas obrigatórias
            valores = [dict(dados).get(coluna, None) for coluna in colunas]
                        insert_query = f"INSERT INTO {nome_tabela} ({colunas_str}) VALUES
({valores_placeholder})"
            cursor.execute(insert_query, valores)

            conn.commit()
            conn.close()

            print(f"Dados salvos com sucesso na tabela: {nome_tabela}")

        except Exception as e:
            print(f"Erro ao salvar os dados (tupla): {str(e)}")
                  QMessageBox.critical(self.parent, "Erro", f"Erro ao salvar os dados (tupla):
{str(e)}")

    # Método para exibir os dados obtidos no QDialog
    def exibir_dados_em_dialog(self, data_informacoes_lista, resultados_completos):
        # Cria o QDialog para exibir os dados
        dialog = QDialog(self.parent)
        dialog.setWindowTitle("Dados do PNCP")

        # Define o tamanho fixo do QDialog
        dialog.setFixedSize(800, 400)  # Aumenta o tamanho para acomodar os dois layouts

        # Cria um layout horizontal principal
        layout_horizontal = QHBoxLayout()

        # Cria os layouts verticais para os dois conjuntos de dados
        layout_informacoes = QVBoxLayout()
        layout_resultados = QVBoxLayout()

        # Campo de texto para exibir 'data_informacoes'
        text_edit_informacoes = QTextEdit()
        text_edit_informacoes.setReadOnly(True)
```

```
        # Campo de texto para exibir 'resultados_completos'
        text_edit_resultados = QTextEdit()
        text_edit_resultados.setReadOnly(True)

        # Função para formatar a lista de pares (chave, valor)
        def formatar_lista_pares(lista_pares):
            texto = ""
            for chave, valor in lista_pares:
                texto += f"{chave}: {valor}\n"
            return texto

        # Exibir 'data_informacoes_lista' como uma lista de pares chave-valor
        texto_informacoes = "Informações:\n"
        texto_informacoes += formatar_lista_pares(data_informacoes_lista)

        # Exibir 'resultados_completos'
        texto_resultados = "Resultados:\n"
        if isinstance(resultados_completos, list):
            for i, resultado in enumerate(resultados_completos, 1):
                texto_resultados += f"\nItem {i}:\n"
                    texto_resultados += json.dumps(resultado, indent=2)  # Formatação básica para
exibir o JSON dos resultados
        else:
            texto_resultados += json.dumps(resultados_completos, indent=2)

        # Adiciona os textos formatados aos QTextEdits
        text_edit_informacoes.setText(texto_informacoes)
        text_edit_resultados.setText(texto_resultados)

        # Adiciona os QTextEdits aos layouts verticais
        layout_informacoes.addWidget(QLabel("Informações"))
        layout_informacoes.addWidget(text_edit_informacoes)

        layout_resultados.addWidget(QLabel("Resultados"))
        layout_resultados.addWidget(text_edit_resultados)

        # Adiciona os dois layouts verticais ao layout horizontal
        layout_horizontal.addLayout(layout_informacoes)
        layout_horizontal.addLayout(layout_resultados)

        # Adiciona o layout principal ao QDialog
        layout = QVBoxLayout()
        layout.addLayout(layout_horizontal)

        # Botão para integrar dados
        button_integrar = QPushButton("Integrar Dados")
               button_integrar.clicked.connect(lambda:  self.integrar_dados(data_informacoes_lista,
resultados_completos))
        layout.addWidget(button_integrar)

        # Botão de fechar o diálogo
        button_close = QPushButton("Fechar")
        button_close.clicked.connect(dialog.accept)
```

```
    layout.addWidget(button_close)

    # Define o layout no diálogo
    dialog.setLayout(layout)

    # Exibe o diálogo
    dialog.exec()

def limpar_dados(self, json_data):
    campos_para_remover = [
        "orcamentoSigiloso",
        "itemCategoriaId",
        "itemCategoriaNome",
        "patrimonio",
        "codigoRegistroImobiliario",
        "criterioJulgamentoId",
        "situacaoCompraItem",
        "tipoBeneficio",
        "incentivoProdutivoBasico",
        "imagem",
        "aplicabilidadeMargemPreferenciaNormal",
        "aplicabilidadeMargemPreferenciaAdicional",
        "percentualMargemPreferenciaNormal",
        "percentualMargemPreferenciaAdicional",
        "ncmNbsCodigo",
        "ncmNbsDescricao",
        "tipoPessoa",
        "timezoneCotacaoMoedaEstrangeira",
        "moedaEstrangeira",
        "valorNominalMoedaEstrangeira",
        "dataCotacaoMoedaEstrangeira",
        "codigoPais",
        "porteFornecedorId",
        "amparoLegalMargemPreferencia",
        "amparoLegalCriterioDesempate",
        "paisOrigemProdutoServico",
        "indicadorSubcontratacao",
        "ordemClassificacaoSrp",
        "motivoCancelamento",
        "situacaoCompraItemResultadoId",
        "sequencialResultado",
        "naturezaJuridicaNome",
        "porteFornecedorNome",
        "naturezaJuridicaId",
        "dataCancelamento",
        "aplicacaoMargemPreferencia",
        "aplicacaoBeneficioMeEpp",
        "aplicacaoCriterioDesempate"
    ]

    # Remover os campos de cada item no json_data
    for item in json_data:
        for campo in campos_para_remover:
```

```
            item.pop(campo, None)

    return json_data
```