src/api/routes/reconciliation.py """ 🚀 API ASSÍNCRONA PARA CONCILIAÇÃO Endpoint: POST /api/reconciliation/analyze Processa em background (não bloqueia) Retorna ID de job para monitorar progresso """ from fastapi import APIRouter, UploadFile, File, BackgroundTasks from fastapi.responses import JSONResponse import uuid import os from pathlib import Path from ..scripts.reconciliation_engine import ConciliationEngine import logging logger = logging.getLogger(__name__) router = APIRouter(prefix="/api/reconciliation", tags=["reconciliation"]) # Armazenar status de processamento PROCESSING_JOBS = {} UPLOAD_DIR = Path("./uploads") OUTPUT_DIR = Path("./output") UPLOAD_DIR.mkdir(exist_ok=True) OUTPUT_DIR.mkdir(exist_ok=True) def process_background(job_id: str, banco_path: str, financeiro_path: str, razao_path: str = None): """Função que roda em background""" try: PROCESSING_JOBS[job_id] ['status'] = 'processing' engine = ConciliationEngine() # Parse files banco_df, financeiro_df = engine.parse_files(banco_path, financeiro_path) # Reconcile resultado = engine.reconcile(banco_df, financeiro_df) # Export arquivos = engine.export(resultado, str(OUTPUT_DIR)) # Update job status PROCESSING_JOBS[job_id] = { 'status': 'completed', 'taxa_conciliacao': resultado['taxa_conciliacao'], 'banco_conciliado': resultado['banco_conciliado'], 'banco_total': resultado['banco_total'], 'files': arquivos, 'timestamp': resultado } logger.info(f"✅ Job {job_id} concluído: {resultado['taxa_conciliacao']:.1f}%") except Exception as e: PROCESSING_JOBS[job_id] = { 'status': 'error', 'error': str(e) } logger.error(f"❌ Job {job_id} falhou: {str(e)}") @router.post("/analyze") async def analyze_reconciliation(background_tasks: BackgroundTasks, banco_file: UploadFile = File(...), financeiro_file: UploadFile = File(...), razao_file: UploadFile = None): """ 📤 ENDPOINT: Enviar arquivos para conciliação Retorna imediatamente com job_id para monitorar progresso """ # Gerar ID único para o job job_id = str(uuid.uuid4()) # Salvar uploads banco_path = UPLOAD_DIR / f"{job_id}_banco.csv" financeiro_path = UPLOAD_DIR / f"{job_id}_financeiro.csv" # Escrever arquivos content = await banco_file.read() with open(banco_path, 'wb') as f: f.write(content) content = await financeiro_file.read() with open(financeiro_path, 'wb') as f: f.write(content) # Registrar job PROCESSING_JOBS[job_id] = { 'status': 'queued', 'created_at': str(__import__('datetime').datetime.now()) } # Disparar processamento em background background_tasks.add_task(process_background, job_id, str(banco_path), str(financeiro_path), None) logger.info(f"📨 Job {job_id} iniciado") return JSONResponse({ 'status': 'processing', 'job_id': job_id, 'message': 'Análise iniciada. Use /status/{job_id} para monitorar' }) @router.get("/status/{job_id}") async def get_status(job_id: str): """ 📊 ENDPOINT: Monitorar progresso """ if job_id not in PROCESSING_JOBS: return JSONResponse({'error': 'Job not found'}, status_code=404) job = PROCESSING_JOBS[job_id] return JSONResponse({ 'job_id': job_id, 'status': job['status'], 'taxa_conciliacao': job.get('taxa_conciliacao'), 'banco_conciliado': job.get('banco_conciliado'), 'banco_total': job.get('banco_total'), }) @router.get("/download/{job_id}/{file_type}") async def download_file(job_id: str, file_type: str): """ 💾 ENDPOINT: Baixar arquivo conciliado file_type: 'banco' ou 'financeiro' """ if job_id not in PROCESSING_JOBS: return JSONResponse({'error': 'Job not found'}, status_code=404) job = PROCESSING_JOBS[job_id] if job['status'] != 'completed': return JSONResponse({'error': f"Job still {job['status']}"}, status_code=400) file_map = { 'banco': job['files']['banco_file'], 'financeiro': job['files'] ['financeiro_file'] } if file_type not in file_map: return JSONResponse({'error': 'Invalid file type'}, status_code=400) from fastapi.responses import FileResponse return FileResponse(file_map[file_type], media_type='text/csv', filename=f" {file_type.upper()}_CONCILIADO.csv")

---

tests/test_reconciliation.py

""" ✅ TESTES DE CARGA Gera 10.000 transações sintéticas Valida performance em < 10 segundos """ import pytest import pandas as pd import time from datetime import datetime, timedelta import random import numpy as np from src.scripts.reconciliation_engine import ConciliationEngine @pytest.fixture def synthetic_data(): """Gera 10.000 transações fictícias para teste de carga""" n_records = 10000 start_date = datetime(2024, 1, 1) # BANCO: 500 linhas com múltiplos itens financeiros banco_data = [] for i in range(500): data = start_date + timedelta(days=random.randint(0, 365)) empresa = random.choice(['PPSI', 'PRIME', 'TEST']) valor = round(random.uniform(100, 50000), 2) banco_data.append({ 'Data': data, 'Empresa': empresa, 'Valor': -valor, 'Descricao': f'Transacao BANCO {i}', 'Banco/Caixa': 'ITAU' }) banco_df = pd.DataFrame(banco_data) # FINANCEIRO: 9.500 linhas (múltiplas para cada BANCO) fin_data = [] for i in range(9500): data = start_date + timedelta(days=random.randint(0, 365)) empresa = random.choice(['PPSI', 'PRIME', 'TEST']) valor = round(random.uniform(10, 5000), 2) fin_data.append({ 'Data movimento': data, 'Empresa': empresa, 'Valor (R$)': -valor, 'Descrição': f'Lançamento FIN {i}', 'CÓDIGO Conciliação': '' }) financeiro_df = pd.DataFrame(fin_data) return banco_df, financeiro_df def test_performance_10k_records(synthetic_data): """Teste: Processar 10.000 registros em menos de 10 segundos""" banco_df, financeiro_df = synthetic_data # Salvar em CSVs temporários banco_df.to_csv('/tmp/test_banco.csv', index=False) financeiro_df.to_csv('/tmp/test_fin.csv', index=False) # Executar motor engine = ConciliationEngine() start = time.time() banco_df, financeiro_df = engine.parse_files('/tmp/test_banco.csv', '/tmp/test_fin.csv') resultado = engine.reconcile(banco_df, financeiro_df) elapsed = time.time() - start print(f"\n⏱️ Tempo de processamento: {elapsed:.2f}s") print(f"📊 Taxa de conciliação: {resultado['taxa_conciliacao']:.1f}%") print(f"✅ Banco conciliado: {resultado['banco_conciliado']}/{resultado['banco_total']}") # Asserts assert elapsed < 10, f"Processamento levou {elapsed:.2f}s (máximo: 10s)" assert resultado['taxa_conciliacao'] > 0, "Taxa de conciliação deve ser > 0" def test_memory_optimization(): """Teste: Otimização de memória reduz uso em 50%""" # Criar DataFrame grande com tipos não otimizados n = 100000 df = pd.DataFrame({ 'Nome': ['Test'] * n, 'Empresa': ['PPSI'] * (n//2) + ['PRIME'] * (n//2), 'Valor': np.random.rand(n) * 1000, 'Data': pd.date_range('2024-01-01', periods=n, freq='H') }) mem_before = df.memory_usage(deep=True).sum() / 1024**2 engine = ConciliationEngine() df_optimized = engine.optimize_dtypes(df) mem_after = df_optimized.memory_usage(deep=True).sum() / 1024**2 savings = ((mem_before - mem_after) / mem_before) * 100 print(f"\n💾 Memória antes: {mem_before:.2f} MB") print(f"💾 Memória depois: {mem_after:.2f} MB") print(f"💾 Economia: {savings:.1f}%") assert savings > 30, f"Economia deve ser > 30% (foi {savings:.1f}%)" if __name__ == "__main__": pytest.main([__file__, "-v", "-s"])