Banco de Dados – EP 2/2025 – Fase 2

Relatório do Protótipo - TikEvents

Átila Aroso Soares (14745546) | Kawe da Cruz Gomes (11838839) | Vitor Pazos (14611814)

---

1. Linguagem de Programação

Utilizamos **Python 3.10+** como linguagem para o desenvolvimento do protótipo. A escolha se justifica pela simplicidade sintática, curva de aprendizado suave e excelente ecossistema de bibliotecas para integração com bancos de dados. Python oferece suporte nativo para gerenciamento de conexões, tratamento de exceções e manipulação de dados, facilitando o desenvolvimento rápido de protótipos funcionais. A biblioteca `psycopg2` foi selecionada por ser o adaptador PostgreSQL mais maduro e confiável para Python, oferecendo suporte completo ao protocolo do PostgreSQL e excelente performance.

2. Sistema Gerenciador de Banco de Dados

Adotamos o **PostgreSQL 14** como SGBD por ser uma solução robusta, gratuita e de código aberto. PostgreSQL oferece conformidade com padrões SQL, suporte completo a transações ACID, integridade referencial avançada com múltiplas opções de CASCADE, tipos de dados apropriados para nosso domínio (NUMERIC para valores monetários, DATE/TIME para eventos) e constraints complexas (CHECK, UNIQUE compostas). Sua arquitetura extensível permite futuras otimizações através de índices especializados e particionamento de tabelas.

O PostgreSQL também se destaca pela excelente documentação, comunidade ativa e facilidade de migração para ambientes de produção em nuvem (AWS RDS, Google Cloud SQL, Azure Database), garantindo escalabilidade futura do projeto. A ferramenta pgAdmin fornece interface gráfica intuitiva para administração durante o desenvolvimento.

3. Procedimentos de Instalação do SGBD

Windows:

1. Baixar o instalador em `postgresql.org/download/windows`
2. Executar como administrador e seguir o wizard de instalação
3. Definir senha forte para o superusuário 'postgres'
4. Adicionar `C:\Program Files\PostgreSQL\14\bin` ao PATH do sistema
5. Verificar instalação com `psql --version` no terminal

Linux (Ubuntu/Debian):

```
sudo apt update && sudo apt install postgresql postgresql-contrib
sudo systemctl start postgresql
sudo systemctl enable postgresql
sudo -u postgres psql
CREATE DATABASE tikevents;
CREATE USER dev WITH PASSWORD 'devpass';
GRANT ALL PRIVILEGES ON DATABASE tikevents TO dev;
\q
```

macOS:

```
brew install postgresql@14
brew services start postgresql@14
psql postgres
CREATE DATABASE tikevents;
CREATE USER dev WITH PASSWORD 'devpass';
GRANT ALL PRIVILEGES ON DATABASE tikevents TO dev;
\q
```

4. Procedimentos de Conexão com o SGBD

A conexão é estabelecida através da string DSN (Data Source Name): `dbname=tikevents user=dev password=devpass host=localhost port=5432`. Utilizamos context managers do Python (`with`) para garantir o gerenciamento adequado de recursos, fechando automaticamente conexões e cursores mesmo em caso de exceções.

Para configurar o ambiente Python, instale as dependências necessárias:

```
python -m venv venv
source venv/bin/activate  # Linux/Mac
venv\Scripts\activate     # Windows
pip install psycopg2-binary
```

Em caso de erros comuns de conexão: *"password authentication failed"* indica credenciais incorretas; *"connection refused"* sugere que o serviço PostgreSQL não está em execução; *"relation does not exist"* significa que as tabelas ainda não foram criadas - execute o script SQL primeiro.

5. Programa Fonte Desenvolvido

```
# arquivo: prototipo_tikevents.py
# Requisitos: pip install psycopg2-binary

import psycopg2
from typing import List, Tuple, Optional

# Configuração da conexão
DSN = "dbname=tikevents user=dev password=devpass host=localhost port=5432"

# DDL para criar tabela se não existir
DDL = """
CREATE TABLE IF NOT EXISTS Artista (
    id_artista SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    genero VARCHAR(50)
);
"""

def get_conn():
    """Retorna uma nova conexão com o banco de dados."""
    return psycopg2.connect(DSN)

def init_schema():
    """Inicializa o esquema do banco."""
    with get_conn() as conn:
        with conn.cursor() as cur:
            cur.execute(DDL)
        conn.commit()

def create_artista(nome: str, genero: Optional[str] = None) -> int:
    """Insere um novo artista. Retorna o ID criado."""
    with get_conn() as conn:
        with conn.cursor() as cur:
            cur.execute(
                "INSERT INTO Artista (nome, genero) VALUES (%s, %s) RETURNING id_artista;",
                (nome, genero)
            )
            artista_id = cur.fetchone()[0]
        conn.commit()
    return artista_id

def read_artistas() -> List[Tuple]:
    """Retorna todos os artistas cadastrados."""
    with get_conn() as conn:
        with conn.cursor() as cur:
            cur.execute("SELECT id_artista, nome, genero FROM Artista ORDER BY nome;")
            return cur.fetchall()

def update_artista(artista_id: int, novo_nome: Optional[str] = None,
                   novo_genero: Optional[str] = None) -> int:
    """Atualiza dados de um artista. Retorna linhas afetadas."""
    updates = []
    params = []

    if novo_nome is not None:
        updates.append("nome = %s")
        params.append(novo_nome)

    if novo_genero is not None:
        updates.append("genero = %s")
        params.append(novo_genero)

    if not updates:
        return 0
```

```
    params.append(artista_id)

    with get_conn() as conn:
        with conn.cursor() as cur:
            cur.execute(
                f"UPDATE Artista SET {', '.join(updates)} WHERE id_artista = %s;",
                params
            )
            rows = cur.rowcount
        conn.commit()

    return rows

def delete_artista(artista_id: int) -> int:
    """Remove um artista. Retorna linhas afetadas."""
    with get_conn() as conn:
        with conn.cursor() as cur:
            cur.execute("DELETE FROM Artista WHERE id_artista = %s;", (artista_id,))
            rows = cur.rowcount
        conn.commit()
    return rows

# Programa de teste
if __name__ == '__main__':
    init_schema()

    # CREATE - Inserir artistas
    a1 = create_artista('Radiohead', 'Rock Alternativo')
    a2 = create_artista('Björk', 'Art Pop')
    print(f'Artistas criados: {a1}, {a2}')

    # READ - Listar todos
    artistas = read_artistas()
    print(f'Lista de artistas: {artistas}')

    # UPDATE - Atualizar gênero
    update_artista(a2, novo_genero='Experimental')
    print('Gênero atualizado')

    # DELETE - Remover artista
    delete_artista(a1)
    print(f'Artista {a1} removido')

    # Verificar resultado final
    artistas_final = read_artistas()
    print(f'Lista final: {artistas_final}')
```

Observação: O protótipo implementa com sucesso as quatro operações CRUD (Create, Read, Update, Delete) para a tabela Artista, validando a arquitetura proposta. O código está estruturado de forma modular, facilitando a expansão para as demais tabelas nas próximas fases do projeto.