Modelo Relacional: Implementação com DBAPI + PostgreSQL

QXD0099 - Desenvolvimento de Software para Persistência

Universidade Federal do Ceará - *Campus* Quixadá

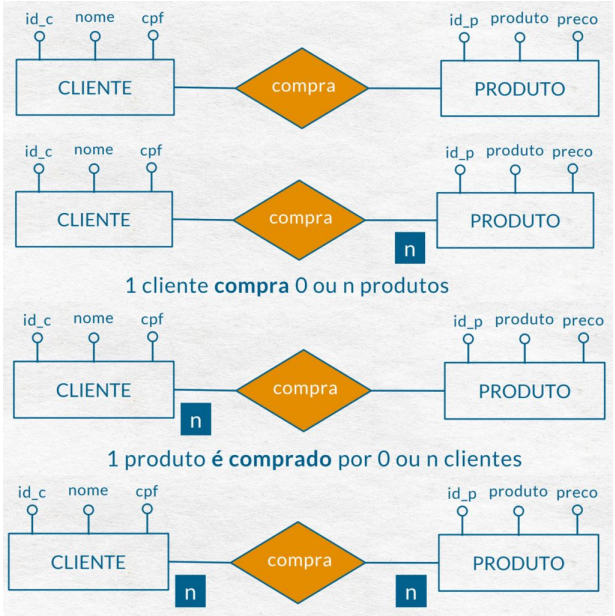
Prof. Francisco Victor da Silva Pinheiro
victorpinheiro@ufc.br

Agenda

- Entidades e Relacionamentos
- Tipos de Relacionamentos
- Introdução ao FastAPI sem ORM
  - Quando e por que usar sem ORM?
  - Boas práticas
- Uso de parâmetros em consultas SQL
- Joins e Consultas Complexas
- Implementação

Entidades e Relacionamentos

- **Entidades e Relacionamentos:**
  - Usuário: Representa os clientes do sistema.
  - Pedido: Representa transações realizadas por usuários.
  - Produto: Itens disponíveis para compra.
  - PedidoProduto: Representa a relação N:M entre Pedido e Produto.



Tipos de Relacionamentos

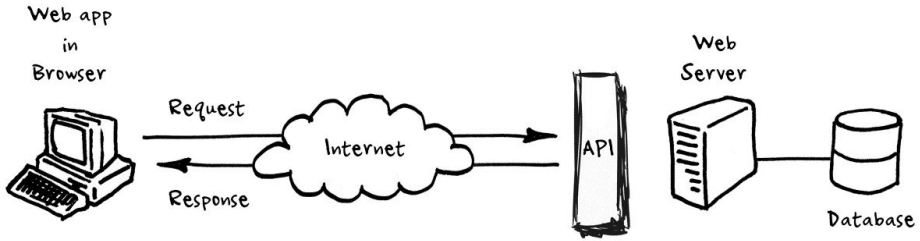
- **1:1 (Um-para-Um):**
  - Exemplo: Cada funcionário tem uma única estação de trabalho.
  - Implementação: FK com UNIQUE.

- **1:N (Um-para-Muitos):**
  - Exemplo: Um cliente pode fazer muitos pedidos.
  - Implementação: FK na tabela do lado "muitos".

- **N:M (Muitos-para-Muitos):**
  - Exemplo: Pedidos podem conter vários produtos, e produtos podem estar em vários pedidos.
  - Implementação: Tabela intermediária com FKs (PedidoProduto).

Introdução ao FastAPI sem ORM

- **Quando e por que usar sem ORM?**
  - Cenários onde não é necessário um banco de dados relacional ou onde o acesso ao banco é realizado por bibliotecas específicas.
  - Melhor controle sobre as consultas SQL.
  - Mais leve e direto, útil para microserviços.



Introdução ao FastAPI sem ORM

- **Configurando o FastAPI**
  - app.py:

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"message": "Hello, FastAPI without ORM"}
```

Introdução ao FastAPI sem ORM

- **Configurando o banco de dados**
  - Use um cliente como sqlite3 ou asyncpg: database.py:

```
import sqlite3

def get_connection():
    conn = sqlite3.connect("database.db")
    return conn
```

Introdução ao FastAPI sem ORM

- **Criando serviços**
  - services.py:

```python
from database import get_connection

def get_all_users():
    conn = get_connection()
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM users")
    rows = cursor.fetchall()
    conn.close()
    return rows
```

Introdução ao FastAPI sem ORM

- **Criando rotas**
  - app.py:

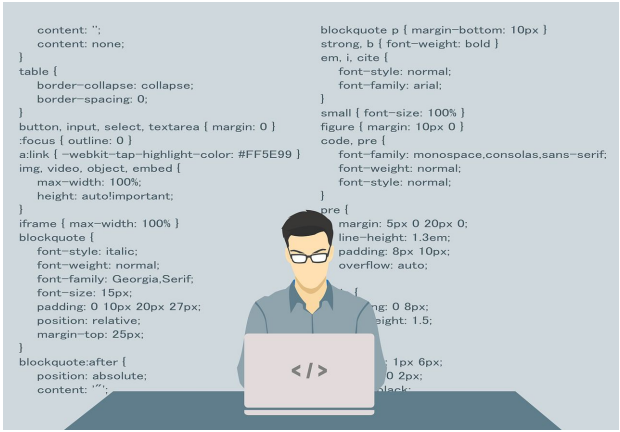
```python
from fastapi import FastAPI
from services import get_all_users

app = FastAPI()

@app.get("/users")
def read_users():
    return {"users": get_all_users()}
```

Introdução ao FastAPI sem ORM

- **Boas práticas**
  - Use conexões de banco de dados gerenciadas (e.g., context managers) para evitar vazamentos.
  - Valide e sanitize os dados usando o Pydantic.
  - Documente sua API automaticamente com o FastAPI (Swagger).



Uso de parâmetros em consultas SQL

- Prevenção de Injeção de SQL
  - Ao usar parâmetros ?, a DB-API automaticamente escapa e sanitiza os valores, protegendo o banco de dados contra entradas maliciosas.

- Reutilização e Manutenção de Código
  - Consultas mais fáceis de ler, manter e reutilizar, pois o SQL é separado dos dados.
  - Pode-se usar a mesma consulta com diferentes valores de parâmetros sem reescrevê-la.

Uso de parâmetros em consultas SQL

- Desempenho Aprimorado
  - Algumas implementações de banco de dados otimizam consultas parametrizadas através de pré-compilação ou caching do plano de execução, reduzindo o tempo de execução em consultas repetidas.

- Separação entre lógica (query) e dados (valores)

- Compatibilidade com Tipos de Dados
  - A DB-API trata automaticamente da formatação correta para os diferentes tipos de dados do banco, como strings, números, datas e binários.

Exemplo

```python
import sqlite3

# Usando gerenciador de contexto para garantir fechamento automático da conexão
with sqlite3.connect("exemplo.db") as conexao:
    # Criando um cursor explicitamente
    cursor = conexao.cursor()

    # Consulta parametrizada
    consulta = "SELECT * FROM usuarios WHERE idade > ? AND cidade = ?"
    parametros = (25, "Nova York")

    # Executando a consulta com parâmetros
    cursor.execute(consulta, parametros)

    # Recuperando os resultados
    resultados = cursor.fetchall()

# Imprimindo os resultados
print(resultados)
```

Joins e Consultas Complexas

- **INNER JOIN:** Retorna registros que têm correspondência em ambas as tabelas.
- **LEFT JOIN:** Retorna todos os registros da tabela à esquerda, com correspondências da tabela à direita.
- **RIGHT JOIN:** Retorna todos os registros da tabela à direita, com correspondências da tabela à esquerda.
- **FULL OUTER JOIN:** Retorna todos os registros de ambas as tabelas, com ou sem correspondência.

Implementação

- Configuração do Ambiente Instale as bibliotecas necessárias:
  - pip install fastapi uvicorn psycopg2 psycopg2-binary

- Estrutura do Projeto
  - main.py
  - db.py
  - models.py
  - crud.py

Passo a passo

- Criação das Tabelas no PostgreSQL com DBAPI
  - Arquivo: db.py
- Modelo Relacional
  - Arquivo: models.py
- CRUD Completo
  - Arquivo: crud.py
- Endpoints
  - Arquivo: main.py

Criação das Tabelas no PostgreSQ

```
CREATE TABLE IF NOT EXISTS usuario (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL
)

CREATE TABLE IF NOT EXISTS pedido (
    id SERIAL PRIMARY KEY,
    usuario_id INT REFERENCES usuario(id),
    data_pedido TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    status VARCHAR(20) NOT NULL
)
```

Criação das Tabelas no PostgreSQ

```
CREATE TABLE IF NOT EXISTS produto (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    preco DECIMAL(10, 2) NOT NULL
)

CREATE TABLE IF NOT EXISTS pedido_produto (
    pedido_id INT REFERENCES pedido(id),
    produto_id INT REFERENCES produto(id),
    quantidade INT NOT NULL,
    PRIMARY KEY (pedido_id, produto_id)
)
```

Consultas Avançadas com Joins

```
SELECT p.id AS pedido_id, u.nome AS usuario, p.data_pedido, p.status
       FROM pedido p
       INNER JOIN usuario u ON p.usuario_id = u.id
```

- Seleciona o ID do pedido, nome do usuário, data e status.
- Usa a tabela pedido e faz um INNER JOIN com usuario para combinar dados onde pedido.usuario_id = usuario.id.
- Retorna uma tabela unindo essas informações, como o nome do usuário para cada pedido.

Consultas Avançadas com Joins

```
SELECT p.id AS pedido_id, p.data_pedido, p.status, u.nome AS usuario
       FROM pedido p
       RIGHT JOIN usuario u ON p.usuario_id = u.id
```

- Utiliza RIGHT JOIN para incluir todos os registros da tabela usuario.
- Campos Retornados:
  - ID, data e status do pedido (pedido).
  - Nome do usuário (usuario).
- Registros sem Correspondência:
  - Usuários sem pedidos terão os campos do pedido como NULL.

Consultas Avançadas com Joins

```
SELECT u.nome AS usuario, p.id AS pedido_id, p.data_pedido, p.status
        FROM usuario u
        FULL OUTER JOIN pedido p ON u.id = p.usuario_id
```

- FULL OUTER JOIN inclui todos os registros de ambas as tabelas.
- Combina os dados quando há correspondência com base na condição ON u.id = p.usuario_id.
- Quando não há correspondência:
    - Campos sem dados correspondentes são preenchidos com NULL.

Consultas Avançadas com Joins

```
SELECT
    u.nome AS usuario,
    COUNT(p.id) AS total_pedidos,
    SUM(pp.quantidade * pr.preco) AS total_gasto,
    AVG(pp.quantidade * pr.preco) AS gasto_medio_por_pedido,
    MAX(pp.quantidade * pr.preco) AS maior_pedido,
    MIN(pp.quantidade * pr.preco) AS menor_pedido
FROM
    usuario u
LEFT JOIN
    pedido p ON u.id = p.usuario_id
LEFT JOIN
    pedido_produto pp ON p.id = pp.pedido_id
LEFT JOIN
    produto pr ON pp.produto_id = pr.id
GROUP BY
    u.id, u.nome
ORDER BY
    total_gasto DESC, total_pedidos DESC;
```

- Listar usuários com informações sobre seus pedidos e gastos.
- JOINs:
  - Relaciona usuários com pedidos, itens nos pedidos e produtos.
- Campos Calculados:
  - Total de pedidos (COUNT).
  - Total gasto (SUM).
  - Gasto médio por pedido (AVG).
  - Valor do maior pedido (MAX).
  - Valor do menor pedido (MIN).
- Agrupamento: Por ID e nome do usuário (GROUP BY).
- Ordenação: Prioriza maior gasto e número de pedidos.

Implementação

Referências

- PEP 249 – Python Database API Specification v2.0 | peps.python.org
- Python Database API Specification 2.0 — pyfirebirdsql 1.0.0 documentation
- The Novice’s Guide to the Python 3 DB-API | Phil Varner

Obrigado!

Dúvidas?

Universidade Federal do Ceará - *Campus* Quixadá

Prof. Francisco Victor da Silva Pinheiro
victorpinheiro@ufc.br