# Mapeamento Objeto-Relacional: SQLAlchemy + CRUD Router e Performance e Logs

QXD0099 - Desenvolvimento de Software para Persistência

**Universidade Federal do Ceará - *Campus* Quixadá**

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

# Agenda

- SQLAlchemy - recapitulando
- CRUD Router
- Técnicas para Melhorar Performance em Bancos de Dados
- Logs
  - Configuração Básica de Logs
  - Adicionar Logs nos Endpoints
  - Logs Estruturados com JSON
- Monitoramento e Métricas
- Diagnóstico de Performance com Ferramentas

# SQLAlchemy - Recapitulando

- **Componentes**
  - **Engine:** Abstração da conexão com o banco de dados.
  - **Session:** Contexto para executar operações no banco.
  - **Declarative Base:** Classe base para definir modelos.

# Exemplo de Configuração com SQLite

```python
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

# Configuração do banco
engine = create_engine('sqlite:///example.db')
Base = declarative_base()
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

# Modelo de exemplo
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, index=True)
    email = Column(String, unique=True, index=True)

# Criar tabelas
Base.metadata.create_all(bind=engine)
```

# CRUD Router

- **O que é o CRUD Router?**
  - O CRUDRouter é uma extensão do FastAPI que facilita a criação de operações CRUD (Create, Read, Update, Delete) em APIs.

  - Ele permite configurar rotas com menos código, oferecendo funcionalidades prontas para manipular dados em bancos de dados.

- *pip install fastapi-crudrouter*

# Exemplo básico com CRUDRouter

```python
from fastapi import FastAPI
from fastapi_crudrouter import SQLAlchemyCRUDRouter
from sqlalchemy.orm import Session

# Modelo e Sessão importados da configuração anterior
app = FastAPI()

# Roteador CRUD
router = SQLAlchemyCRUDRouter(
    schema=User,  # Esquema de dados
    db_model=User,  # Modelo do banco
    db=get_db  # Função que retorna a sessão do banco
)

app.include_router(router)
```

- **app = FastAPI()**: Cria a aplicação FastAPI.
- **SQLAlchemyCRUDRouter**: Gera rotas CRUD (Create, Read, Update, Delete) automaticamente.
  - **schema=User**: Valida dados com Pydantic.
  - **db_model=User**: Representa a tabela no banco de dados.
  - **db=get_db**: Fornece conexões ao banco.
- **app.include_router(router)**: Adiciona as rotas CRUD à API.
- A API terá endpoints como POST /users/, GET /users/, PUT /users/{id}, etc.

# CRUD Router

- **Endpoints Gerados Automaticamente**
  - GET /users/ - Lista todos os usuários.
  - POST /users/ - Cria um novo usuário.
  - GET /users/{id} - Obtém um usuário pelo ID.
  - PUT /users/{id} - Atualiza um usuário pelo ID.
  - DELETE /users/{id} - Exclui um usuário pelo ID.

# CRUD com banco de dados usando SQLAlchemy

```python
from fastapi import FastAPI
from fastapi_crudrouter import SQLAlchemyCRUDRouter
from sqlalchemy import Column, Integer, String, create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

# Configuração do SQLAlchemy
DATABASE_URL = "sqlite:///./test.db"
engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

# Modelo de dados
class Item(Base):
    __tablename__ = "items"
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, index=True)

Base.metadata.create_all(bind=engine)

# Configuração do FastAPI com CRUDRouter
app = FastAPI()
router = SQLAlchemyCRUDRouter(
    schema=Item,
    db_model=Item,
    db=sessionmaker(bind=engine)(),
    prefix="items"
)

app.include_router(router)
```

- **Configuração do banco de dados**:
  - DATABASE_URL: Define o caminho do banco SQLite.
  - engine: Cria uma conexão com o banco.
  - SessionLocal: Gerencia as sessões de banco de dados.
  - Base: Base para definir modelos de dados.
- **Modelo de dados**:
  - Classe Item representa a tabela items com duas colunas: id e name.
- **CRUD automático**:
  - SQLAlchemyCRUDRouter: Cria rotas CRUD automaticamente para o modelo Item.
- **Inicialização do FastAPI**:
  - app.include_router(router): Inclui as rotas CRUD na aplicação.
- **Resultado**: A API terá rotas para criar, ler, atualizar e deletar itens em /items.

# Configurações adicionais

- **Prefixo das rotas:**
  - Você pode adicionar um prefixo às rotas para melhor organização

```
app.include_router(MemoryCRUDRouter(schema=Item, prefix="/items"))
```

- **Customização das operações CRUD:**
  - Você pode sobrescrever qualquer operação padrão do CRUDRouter:

```
@router.get("/custom-route")
async def custom_route():
    return {"message": "Esta é uma rota customizada!"}
```

# Vantagens do CRUD Router

- Redução de código boilerplate.
- Fácil integração com ORMs (SQLAlchemy, Tortoise ORM, etc.).
- Configurações flexíveis: permite ajustar prefixos, rotas e métodos.
- Compatível com bancos de dados relacionais e NoSQL,

# Técnicas para Melhorar Performance em Bancos de Dados

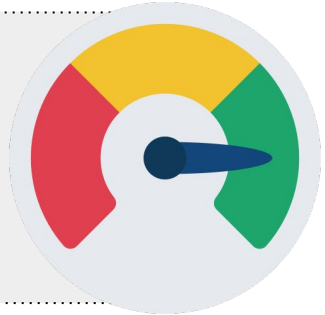
Performance

# Lazy Loading (Carregamento Preguiçoso):

- Os dados relacionados são carregados sob demanda, ou seja, apenas quando você acessa a propriedade.
- **Vantagem:** Economiza memória e reduz o número de dados transferidos se nem todos os dados relacionados forem necessários.
- **Desvantagem:** Pode causar múltiplas consultas ao banco (problema de N+1 queries).

```python
from sqlalchemy.orm import relationship

class User(Base):
    id = Column(Integer, primary_key=True)
    posts = relationship("Post", lazy="select")  # Lazy Loading
```



# Eager Loading (Carregamento Antecipado):

- Os dados relacionados são carregados imediatamente, junto com a consulta principal.

- **Vantagem:** Reduz o número de consultas ao banco (útil quando você sabe que precisará dos dados relacionados).

- **Desvantagem:** Pode carregar mais dados do que o necessário, aumentando o tempo da consulta inicial.

```
session.query(User).options(joinedload(User.posts)).all()  # Eager Loading
```

# Uso de Índices

- Um índice é uma estrutura que acelera as buscas em uma tabela, reduzindo o tempo de leitura de dados.

- **Tipos de índices:**
  - **Índices simples:** Baseados em uma única coluna.
  - **Índices compostos:** Baseados em múltiplas colunas.
  - **Índices únicos:** Garantem que os valores em uma coluna ou conjunto de colunas sejam únicos.

# Uso de Índices

- **Vantagem:**
  - Acelera consultas que usam cláusulas WHERE, JOIN, ORDER BY, e GROUP BY.

- **Desvantagem:**
  - Aumenta o uso de armazenamento.
  - Pode impactar negativamente o desempenho de operações de escrita (inserções, atualizações e exclusões).

```
CREATE INDEX idx_coluna ON tabela(coluna);
```

# Cache

- Cache armazena dados frequentemente acessados em memória para evitar consultas repetidas ao banco de dados.

- **Tipos de cache:**
  - **Cache em memória:** Usa ferramentas como Redis, Memcached ou a memória local da aplicação.
  - **Query caching:** Armazena os resultados das consultas.
  - **Cache de objeto:** Armazena objetos serializados para evitar a reexecução de consultas complexas.

- **Vantagem:**
  - Reduz o tempo de resposta.
  - Diminui a carga no banco de dados.

- **Desvantagem:**
  - Pode levar a inconsistências se o cache não for invalidado corretamente.

```python
import redis

r = redis.Redis(host='localhost', port=6379, db=0)

# Definindo um valor no cache
r.set('key', 'value')

# Recuperando o valor do cache
print(r.get('key'))
```

# Quando aplicar cada técnica?

- Use Lazy Loading quando trabalhar com grandes relações e não precisar de todos os dados relacionados imediatamente.
- Use Eager Loading para reduzir consultas ao banco em casos onde você sabe que os dados relacionados serão necessários.
- Índices são essenciais para melhorar a velocidade de leitura em consultas frequentes, mas use-os estrategicamente para não comprometer a performance de escrita.
- Implemente cache para dados que não mudam frequentemente, como resultados de consultas pesadas ou dados acessados repetidamente.

# Logs

- **Importância dos Logs:**
  - Monitorar operações e erros em tempo real.
  - Análise de comportamento e depuração.

```python
import logging

# Configurar logs
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(message)s')

logger = logging.getLogger(__name__)
logger.info("Aplicação iniciada")
```

**Configuração básica de logs com Python**

# Adicionar Logs nos Endpoints

```python
from fastapi import FastAPI, Request
import logging

app = FastAPI()
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger("fastapi")

@app.middleware("http")
async def log_requests(request: Request, call_next):
    logger.info(f"Recebendo requisição: {request.method} {request.url}")
    response = await call_next(request)
    logger.info(f"Resposta enviada com status: {response.status_code}")
    return response
```

# Logs Estruturados com JSON

```
import json_log_formatter

formatter = json_log_formatter.JSONFormatter()
json_handler = logging.FileHandler(filename='app.log')
json_handler.setFormatter(formatter)
logger.addHandler(json_handler)

logger.info("Início da aplicação", extra={"event": "startup"})
```

# Monitoramento e Métricas

- **Middleware para Monitoramento de Desempenho**
  - Medir o tempo de execução das requisições.

```python
import time

@app.middleware("http")
async def measure_execution_time(request: Request, call_next):
    start_time = time.time()
    response = await call_next(request)
    process_time = time.time() - start_time
    logger.info(f"Tempo de execução: {process_time} segundos")
    return response
```

- **Integrando com Ferramentas de Observabilidade**
  - Ferramentas como Prometheus e Grafana para monitoramento

# Diagnóstico de Performance com Ferramentas

- **Ferramenta EXPLAIN no SQL:**
  - Utilize para diagnosticar consultas lentas e identificar gargalos.

```
EXPLAIN SELECT * FROM clientes WHERE idade > 30;
```

- Quando executa uma consulta com o comando **EXPLAIN** (ou variações como EXPLAIN ANALYZE), o banco de dados retorna informações detalhadas sobre:
  - A ordem de leitura das tabelas.
  - Os índices que estão sendo usados.
  - O método de junção (join) entre tabelas.
  - O custo estimado de cada etapa do plano de execução.

# Diagnóstico de Performance com Ferramentas

- **Profiling com pyinstrument:**
  - Ferramenta para analisar o desempenho de funções.
    - *pip install pyinstrument*

- **Por que usar o PyInstrument?**
  - **Leveza:** É fácil de usar e fornece informações claras.
  - **Foco em alto nível:** Mostra o tempo gasto em chamadas de função e operações, permitindo entender onde o tempo é realmente consumido.
  - **Relatórios detalhados:** Gera relatórios fáceis de interpretar, tanto no terminal quanto em formato HTML.

# Diagnóstico de Performance com Ferramentas

- **Usando como um decorador: Adicione o PyInstrument como um decorador para medir uma função específica:**

```
from pyinstrument import Profiler

def funcao_demorada():
    sum(range(10000000))

profiler = Profiler()
profiler.start()

funcao_demorada()

profiler.stop()
print(profiler.output_text(unicode=True, color=True))
```

# Diagnóstico de Performance com Ferramentas

- **Gerando relatórios em HTML: Para obter um relatório detalhado em HTML:**

```
profiler = Profiler()
profiler.start()

# Código a ser analisado
funcao_demorada()

profiler.stop()
with open("relatorio.html", "w") as f:
    f.write(profiler.output_html())
```


LET'S
CODE

# Referências

- Curso completo de FastAPI por Eduardo Mendes
  - https://fastapidozero.dunossauro.com/
  - https://github.com/dunossauro/fastapi-do-zero
  - Playlist no YouTube

- FastAPI - https://fastapi.tiangolo.com/
- Pydantic - https://pydantic.dev/
- SQLAlchemy - https://www.sqlalchemy.org/
- SQLModel - https://sqlmodel.tiangolo.com

- https://docs.github.com/pt/rest/using-the-rest-api/using-pagination-in-the-rest-api?apiVersion=2022-11-28

# Obrigado!

Dúvidas?

**Universidade Federal do Ceará - *Campus* Quixadá**

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