Manual ORM com SQLAlchemy

Manual Completo de Uso e Treinamento - Estrutura ORM com SQLAlchemy e PostgreSQL

Índice

Introdução

Parabéns. Se você está lendo este documento, é muito provável que tenha sobrevivido à primeira onda de

Manual ORM com SQLAlchemy

documentação técnica ? aquela mais resumida, prática, cheia de bullet points e exemplos secos como torradas esquecidas no forno. Mas agora é diferente. Você chegou ao **MANUAL**, o verdadeiro compêndio. Este é o seu mapa do tesouro, onde vamos destrinchar com minúcia o funcionamento interno desta arquitetura ORM baseada em SQLAlchemy. E não se preocupe, não será uma jornada solitária: eu estarei com você, passo a passo, sem pressa, e com algumas piadas sutis pelo caminho (prometo não exagerar).

Este manual é ideal para:

- Desenvolvedores iniciantes que querem aprender como estruturar suas aplicações com SQLAlchemy.
- Desenvolvedores experientes que precisam compreender os detalhes do projeto.
- Equipes técnicas que desejam padronizar o uso da camada de dados.

Antes de mergulharmos no código, precisamos responder a uma pergunta essencial:

Por que usar uma arquitetura ORM (Object Relational Mapping)?

Se você já teve que escrever dezenas de comandos SQL diretamente dentro do seu código ? e pior, repetir os mesmos SELECTs com pequenas variações ? sabe bem como isso pode se tornar um pesadelo. A abordagem ORM traz um modelo mais elegante: objetos Python representam suas tabelas, e os métodos que você chama nesses objetos geram automaticamente os SQLs necessários por baixo dos panos. Além disso, a separação entre lógica de negócio e persistência de dados se torna muito mais limpa e testável.

Manual ORM com SQLAlchemy

O projeto que você tem em mãos vai além do ORM básico. Ele implementa uma camada intermediária chamada `CRUDMixin`, com suporte a um poderoso encadeamento de consultas via `QueryChain`. Essa combinação permite escrever consultas com um nível de expressividade que beira a poesia ? ou quase isso.

Ao final deste manual, você será capaz de:

1. Entender a estrutura do projeto em profundidade.

2. Configurar corretamente o ambiente.

3. Gerar modelos automaticamente a partir do seu banco de dados.

4. Utilizar os métodos CRUD com segurança e clareza.

5. Encadear consultas complexas usando `QueryChain`.

6. Executar comandos SQL personalizados de forma segura.

Capítulo 1 ? Entendendo a Arquitetura do Projeto

Imagine o projeto como um prédio modular:

O **alicerce** é o SQLAlchemy. Ele define a base da comunicação com o banco e representa suas tabelas como classes.

O **térreo** é o `db.py`, que configura a conexão com o banco e cria a `Base` e o `SessionLocal`, fundamentais para qualquer operação.

O **primeiro andar** é o `crud.py`, que define `CRUDMixin` (a camada que fornece os métodos insert, update, delete, etc.) e `QueryChain`, a alma do encadeamento de consultas.

O **segundo andar** é a pasta `models/`, onde ficam os modelos Python que representam suas tabelas no banco.

No **telhado**, temos arquivos utilitários como `create_tables.py` e `generate_models.py`, que ajudam a construir e manter a estrutura de forma automática.

Nada aqui é aleatório. Cada peça tem sua razão de existir ? e todas funcionam em harmonia para oferecer uma interface robusta, escalável e elegante.

Capítulo 2 ? Preparando o Ambiente: do caos ao controle

Imagine que você acaba de baixar o projeto, animado para ver tudo funcionando. Você digita um `python

script.py` com entusiasmo e... erro. A tela te olha de volta com uma exceção digna de um filme de terror. Calma. Vamos evitar esse cenário.

2.1. Instalação dos pacotes necessários

Primeiro passo é garantir que você tenha o ambiente Python corretamente configurado. O projeto foi testado com Python 3.9+, então evite versões muito antigas (ou muito exóticas).

Instale os pacotes necessários:

pip install sqlalchemy psycopg2-binary

Esses dois pacotes são indispensáveis:

- `sqlalchemy`: o ORM principal que usamos.
- `psycopg2-binary`: driver para conectar com bancos PostgreSQL.

Se quiser brincar em modo local com SQLite, o SQLAlchemy também suporta, mas aqui focaremos na estrutura pensada para PostgreSQL.

2.2. Entendendo o `config.json`

O `config.json` é o cérebro das configurações de ambiente. Ele informa qual banco usar, credenciais, host e até mesmo o schema (isso mesmo, aquele que alguns esquecem que existe em bancos mais parrudos como o PostgreSQL).

Veja um exemplo de entrada de ambiente:

```
{
"ambiente": "dev",
"database": {
"dev": {
"database": "pgsql",
"dbname": "meubanco",
"user": "meuusuario",
"password": "minhasenha",
"host": "127.0.0.1",
```

"port": "5432",

"schema": "public"

}

}

}

Alguns pontos importantes:

- `ambiente`: define qual configuração será carregada.
- `database`: agrupa as conexões disponíveis por nome (dev, prod, staging, autokit... você escolhe).

Você pode definir múltiplos ambientes, e mudar entre eles apenas trocando a chave `ambiente`.

2.3. Conexão com o banco

A função `get_engine()` dentro de `db.py` vai ler o `config.json` e montar a string de conexão com base no ambiente ativo. Não é mágica negra ? é só leitura de JSON, concatenação e uso da função `create_engine()` do SQLAlchemy.

O `SessionLocal` também é criado ali, e será usado para abrir conexões de sessão de forma segura e isolada.

Capítulo 3 ? Gerando os Models automaticamente: menos tédio, mais produtividade

Se você já teve que escrever à mão todos os modelos de um banco com 30, 50 ou 200 tabelas... você sabe: é o tipo de tarefa que testa sua sanidade. Este projeto elimina essa tortura com um script que automatiza toda essa geração com base no schema do banco.

3.1. O que o script `generate_models.py` faz por você?

Esse script acessa seu banco de dados, reflete as tabelas e gera um arquivo `.py` para cada tabela, dentro da pasta `models/`. Ele gera:

- A declaração da classe com `Base` e `CRUDMixin`
- Colunas e tipos automaticamente
- Chaves primárias e estrangeiras
- Indexes e restrições únicas, se existirem
- Atribuição de schema

Tudo isso é extraído diretamente da estrutura do banco de dados.

3.2. Executando o gerador com um prefixo

A ideia é que você possa gerar apenas os models que começam com um determinado prefixo (por exemplo, `tbl_`, `app_`, `sys_`, etc.):

```
python generate_models.py tbl_
```

Esse comando gera apenas os arquivos dos modelos cujos nomes de tabela começam com `tbl_`.

O resultado será algo como:

```
models/
??? tbl_bot_registros_primeira_sentenca.py
??? tbl_bots_control.py
```

Cada arquivo contém uma classe declarada corretamente, pronta para uso.

3.3. Anatomia de um model gerado

Vamos analisar um exemplo gerado automaticamente:

```
from sqlalchemy import Column, Integer, String, Boolean

from .db import Base

from .crud import CRUDMixin


class TblBotsControle(Base, CRUDMixin):

__tablename__ = 'tbl_bots_control'

__table_args__ = {'schema': 'public'}


id = Column(Integer, primary_key=True)

nome = Column(String, nullable=False)

ativo = Column(Boolean, default=True)
```

Explicando:

`Base` é a base declarativa do SQLAlchemy, herdada por todos os models.

`CRUDMixin` traz todos os métodos de acesso ao banco (insert, update, delete, all, get, etc.).

`__tablename__` define o nome da tabela no banco.

`__table_args__` define o schema utilizado (importante em PostgreSQL).

As colunas usam os tipos corretos, extraídos do banco.

Você pode editar livremente os modelos após a geração, inclusive adicionando métodos próprios ou propriedades especiais.

No próximo capítulo, você aprenderá a criar as tabelas no banco com um único comando, e entenderá como isso se conecta com os modelos gerados.

Capítulo 4 ? CRUDMixin: o motor silencioso por trás da mágica

Manual ORM com SQLAlchemy

Chegamos a um dos pontos mais poderosos ? e muitas vezes subestimados ? dessa arquitetura: o `CRUDMixin`. Este mixin é responsável por fornecer todos os métodos essenciais para operar sobre os dados. A beleza disso? Você escreve pouquíssimo código e ganha uma capacidade enorme de controle sobre as operações de banco.

Vamos agora destrinchar cada método deste mixin. E não apenas dizer o que ele faz, mas mostrar como, quando, e por que utilizá-lo.

4.1. all(where=None, or_where=None)

Este método é o ponto de entrada para iniciar uma consulta complexa, retornando uma instância de `QueryChain`, que permite encadeamento fluente de filtros, ordenações, joins e outros modificadores.

Parâmetros:

`where`: tupla ou lista de tuplas com filtros a serem aplicados com `AND`
`or_where`: tupla ou lista de tuplas com filtros aplicados com `OR`

Retorno:

Um objeto `QueryChain`, que precisa ser finalizado com um método de execução, como `.toList()`, `.toDict()`, `.first()`, etc.

Exemplo:

usuarios = Usuario.all(where=("ativo", True)).orderBy("id", "desc").limit(10).toDict()

4.2. get(where=None, or_where=None)

Retorna o primeiro registro que satisfaz os critérios passados, convertido automaticamente para um dicionário.

Retorno:

Um `dict` com os campos do model ou `None` se nenhum registro for encontrado.

Exemplo:

admin = Usuario.get(where=("email", "admin@empresa.com"))

4.3. insert(**kwargs)

Cria e persiste um novo registro no banco.

Parâmetros:

Argumentos nomeados, correspondendo às colunas da tabela.

Retorno:

A instância do model recém-criada.

Exemplo:

novo_usuario = Usuario.insert(nome="Fernanda", email="fer@empresa.com", ativo=True)

4.4. create(records)

Cria múltiplos registros em uma única transação.

Parâmetros:

`records`: uma lista de dicionários, cada um representando um novo registro.

Retorno:

Uma lista de dicionários contendo os dados persistidos.

Exemplo:

```
dados = [
{"nome": "Ana"},
{"nome": "Bruno"}
]
usuarios = Usuario.create(dados)
```

4.5. update(data=None, **kwargs)

Atualiza os campos da instância atual. Pode receber os dados em múltiplos formatos.

Formas válidas:

- `data={"campo": valor}`
- `data=[("campo", valor)]`
- `campo=valor` diretamente

Exemplo:

usuario.update(nome="João da Silva", ativo=False)

4.6. delete()

Remove o registro atual do banco. Não há volta, então cuidado com esse botão nuclear.

Exemplo:

usuario.delete()

4.7. findById(id)

Busca um registro pela chave primária.

Retorno:

A instância do model, ou `None` se não encontrar

Exemplo:

usuario = Usuario.findById(42)

4.8. rawSql(sql_string, params=None, db_key=None)

Executa SQL puro diretamente no banco. Deve ser usada com cautela, mas é essencial quando você precisa de consultas mais complexas ou fora do escopo do ORM.

Parâmetros:

- `sql_string`: string SQL (use parâmetros nomeados para segurança)
- `params`: dicionário com os valores dos parâmetros
- `db_key`: opcional, permite selecionar outro banco configurado no `config.json`

Exemplo:

sql = "SELECT * FROM usuarios WHERE nome ILIKE :nome"

res = Usuario.rawSql(sql, {"nome": "%jo%"})

No próximo capítulo, vamos nos aprofundar no `QueryChain`, essa maravilha que permite escrever consultas que antes exigiriam várias linhas de SQL, agora com poucas e elegantes instruções Python.

Capítulo 5 ? QueryChain: a arte de consultar como um mestre zen

Chegamos ao coração do sistema de consultas desta arquitetura: a classe `QueryChain`. Ela é, para todos os efeitos práticos, o seu melhor amigo quando se trata de extrair dados do banco de maneira expressiva, legível e incrivelmente poderosa. Este capítulo é inteiramente dedicado a te transformar em um verdadeiro mestre zen do encadeamento de consultas.

Mas antes de ir direto ao ponto, vamos entender o **porquê** da existência dessa classe.

5.1. O problema: consultas monolíticas, ilegíveis e inflexíveis

Quem já usou SQLAlchemy puro para montar consultas com múltiplos filtros, joins, ordenações e paginações sabe que o código começa a ficar verboso rapidamente. Pior ainda, quando você precisa aplicar condicionais dinâmicas ? dependendo de parâmetros recebidos por uma API ou lógica de negócio ? manter o código limpo se torna quase impossível sem muita abstração.

5.2. A solução: QueryChain

A `QueryChain` resolve exatamente isso. Ela encapsula uma `query` SQLAlchemy e fornece uma interface fluente (inspirada em bibliotecas como jQuery e LINQ) que permite encadear modificadores e só executar a query no final, com um dos métodos de execução.

Vamos olhar para isso em profundidade, analisando cada método que você pode usar e como ele altera o estado da consulta.

5.3. Métodos de Encadeamento

select(*columns)

Seleciona colunas específicas para retorno. Se não for usado, retorna todas as colunas do model.

usuarios = Usuario.all().select("id", "nome").toDict()

Você também pode passar colunas de outro model em joins:

from models.empresa import Empresa

res = Usuario.all().join(Empresa, Usuario.empresa_id == Empresa.id)

.select(Usuario.nome, Empresa.nome.label("empresa"))

.toDict()

where(...)

Aplica um filtro `AND`. Suporta dois formatos:

- `("coluna", valor)` => assume igualdade
- `("coluna", "operador", valor)` => operador explícito (`!=`, `>`, `like`, etc.)

```
Usuario.all().where("ativo", True).where("idade", ">=", 18)
```

whereIn(coluna, lista)

Filtra registros com valores dentro de uma lista:

```
Usuario.all().whereIn("id", [1,2,3])
```

whereNotIn(coluna, lista)

Oposto do anterior:

```
Usuario.all().whereNotIn("perfil", ["admin", "root"])
```

isTrue(coluna) e isFalse(coluna)

Aplica verificação booleana:

```
Usuario.all().isTrue("ativo")
```

```
Usuario.all().isFalse(["confirmado", "validado"])
```

empty(coluna) e notEmpty(coluna)

Verifica se os campos estão vazios (nulo ou string vazia):

Usuario.all().empty("data_desativacao")

Usuario.all().notEmpty("nome")

emptyOrNull(coluna)

Versão mais inteligente que trata strings e outros tipos:

Usuario.all().emptyOrNull("descricao")

join(model, onclause=None)

Faz inner join:

Usuario.all().join(Empresa, Usuario.empresa_id == Empresa.id)

leftJoin(model, onclause=None)

Faz left outer join:

Usuario.all().leftJoin(Empresa, Usuario.empresa_id == Empresa.id)

groupBy(...)

Agrupamento por colunas:

Usuario.all().groupBy(Usuario.perfil).toList()

orderBy(...)

Aceita duas formas:

- Strings em pares: `("nome", "asc")`
- Expressões SQLAlchemy: `Model.coluna.desc()`

Usuario.all().orderBy("nome", "asc", "id", "desc")

Usuario.all().orderBy(Usuario.id.desc())

limit(valor) e offset(valor)

Paginação:

Usuario.all().limit(10).offset(20)

5.4. Métodos de Execução

Estes são os métodos que de fato **executam** a query montada.

toList()

Retorna uma lista de objetos (instâncias do model). Ideal para lógica interna.

registros = Usuario.all().toList()

for r in registros:

print(r.nome)

toDict()

Retorna lista de dicionários. Ideal para APIs e serialização.

dados = Usuario.all().select("id", "nome").toDict()

first()

Retorna a primeira instância encontrada (ou `None`).

u = Usuario.all().where("email", "fulano@email.com").first()

firstToDict()

Mesmo que `first()`, mas converte para dicionário.

dados = Usuario.all().where("email", "fulano@email.com").firstToDict()

count()

Executa um `SELECT COUNT(*)` com os filtros aplicados:

ativos = Usuario.all().isTrue("ativo").count()

5.5. Exemplo prático completo

Vamos construir uma consulta complexa para ilustrar tudo isso:

```
usuarios = Usuario.all()\

.leftJoin(Empresa, Usuario.empresa_id == Empresa.id)\

.select(Usuario.id, Usuario.nome, Empresa.nome.label("empresa"))\

.where("ativo", True)\

.notEmpty("email")\

.orderBy("nome", "asc")\

.limit(10)\

.toDict()
```

Você acabou de construir o equivalente a uma query SQL com `JOIN`, `WHERE`, `ORDER BY`, `LIMIT`, projeção de colunas e tratamento de nulls ? tudo isso em um encadeamento legível e reaproveitável.

No próximo capítulo, vamos entrar em cenários de uso do mundo real, com casos que exigem lógica condicional, múltiplas tabelas e decisões em tempo de execução.

Capítulo 6 ? Casos de Uso Reais: quando o banco de dados encontra a vida real

Até aqui, exploramos ferramentas. Agora é hora de ver essas ferramentas em ação ? como um chef de cozinha que conhece suas facas e panelas, mas quer mesmo é saber como preparar um belo risoto de cogumelos.

Este capítulo apresenta cenários reais que vão desde a consulta simples até a manipulação condicional de dados, passando por joins, paginação dinâmica, construção de filtros em tempo de execução, e integração com endpoints REST.

6.1. Cenário 1 ? Consulta condicional com parâmetros dinâmicos

Imagine que você tem um endpoint de API que aceita múltiplos parâmetros de filtro:

Parâmetros vindos de uma requisição

params = {

"ativo": True,

"perfil": "editor",

"idade_min": 25,

"busca": "silva"

}

Queremos montar a consulta com base apenas nos parâmetros fornecidos (e ignorar os ausentes). Com QueryChain, isso é simples:

query = Usuario.all()

```
if "ativo" in params:

query = query.isTrue("ativo") if params["ativo"] else query.isFalse("ativo")



if "perfil" in params:

query = query.where("perfil", params["perfil"])



if "idade_min" in params:

query = query.where("idade", ">=", params["idade_min"])



if "busca" in params:

query = query.where("nome", "like", f"%{params['busca']}%")



usuarios = query.toDict()
```

Resultado: você construiu dinamicamente uma consulta altamente flexível sem precisar escrever ifs aninhados com SQL literal.

6.2. Cenário 2 ? Gerando relatório com agregações

Vamos supor que você precise de um relatório que conte o número de usuários por perfil. Aqui entra o `groupBy()`:

```
from sqlalchemy import func

resumo = Usuario.all()\
.select(Usuario.perfil, func.count(Usuario.id).label("total"))\
.groupBy(Usuario.perfil)\
.orderBy("total", "desc")\
.toDict()
```

O resultado será uma lista como:

```
[
{"perfil": "admin", "total": 12},
{"perfil": "editor", "total": 7},
```

{"perfil": "leitor", "total": 4}

]

6.3. Cenário 3 ? API paginada com ordenação dinâmica

Suponha que você esteja implementando um endpoint REST com suporte a ordenação e paginação:

Parâmetros de paginação

page = 3

per_page = 20

sort_field = "nome"

sort_order = "asc"

usuarios = Usuario.all()\

.orderBy(sort_field, sort_order)\

.limit(per_page)\

```
.offset((page - 1) * per_page)\

.toDict()
```

Resultado: apenas os usuários daquela página serão retornados, ordenados conforme desejado.

6.4. Cenário 4 ? Join com múltiplas tabelas e múltiplas colunas

Agora imagine que você precisa montar um relatório completo com dados da tabela de usuários, empresas e registros:

```
from models.empresa import Empresa

from models.registro import Registro


dados = Usuario.all()\

.join(Empresa, Usuario.empresa_id == Empresa.id)\

.join(Registro, Registro.usuario_id == Usuario.id)\

.select(
```

```
Usuario.id, Usuario.nome,

Empresa.nome.label("empresa"),

Registro.tipo, Registro.criado_em

)\

.where("ativo", True)\

.orderBy("criado_em", "desc")\

.limit(50)\

.toDict()
```

Você acabou de fazer um join triplo com projeção personalizada. E tudo isso com encadeamento limpo.

6.5. Cenário 5 ? Atualização condicional de registros

Vamos imaginar que você deseja desativar todos os usuários inativos há mais de um ano:

```
from datetime import datetime, timedelta
```

```
limite = datetime.now() - timedelta(days=365)



usuarios = Usuario.all()\

.where("ativo", True)\

.where("ultimo_login", "<", limite)\

.toList()



for usuario in usuarios:

usuario.update(ativo=False)
```

Esses são apenas alguns exemplos reais que mostram o poder da arquitetura quando aplicada com criatividade e clareza.

No próximo capítulo, vamos tratar de boas práticas, erros comuns e como evitar surpresas desagradáveis em produção.

Capítulo 7 ? Boas Práticas, Armadilhas Comuns e Como Evitar Tragédias Anunciadas

Chegamos à parte em que a teoria encontra o campo de batalha. Não importa o quão poderosa seja sua arquitetura: se mal utilizada, ela pode se transformar em um festival de bugs, lentidão e frustração. Este capítulo é um guia de sobrevivência: ele não só revela as boas práticas que você deve seguir, mas também expõe os erros comuns com uma lanterna potente ? e, é claro, mostra como evitá-los.

7.1. Fechando sessões corretamente

Problema: você abriu uma sessão, executou uma query... e nunca a fechou. Isso, em ambientes com

muitas requisições simultâneas, é como deixar a torneira da pia aberta durante um racionamento de água.

Solução: todos os métodos que executam query no `QueryChain` (como `.toDict()`, `.first()`, `.count()`) já fecham a sessão automaticamente. Mas, se você escrever consultas personalizadas, lembre-se de fechar a sessão manualmente:

```
session = SessionLocal()

try:

resultado = session.query(Usuario).all()

finally:

session.close()
```

7.2. Use `.select()` com sabedoria

Problema: você precisa retornar só duas colunas, mas esquece de usar `.select()` e retorna todos os campos ? incluindo colunas com blobs, JSONs enormes ou logs antigos.

Solução: sempre que sua consulta for para API ou relatório, use `.select()` com as colunas exatas. Isso

reduz o tráfego, melhora a performance e clareia o código:

Usuario.all().select("id", "nome", "email").toDict()

7.3. Evite `.toList()` se você precisa de dicionários

Problema: você usa `.toList()` e depois tenta serializar as instâncias do model em JSON... e se vê preso escrevendo `vars()` ou `.__dict__`.

Solução: se você quer um resultado pronto para serialização, use `.toDict()`. O método já converte internamente os objetos com base nas colunas definidas.

7.4. Cuidado com operadores mal utilizados

Problema: você tenta usar `>=` diretamente como string no `.where()` e se esquece de passar os três argumentos:

ERRADO

Usuario.all().where("idade >=", 18)

CERTO

Usuario.all().where("idade", ">=", 18)

Dica bônus: se você tiver múltiplos filtros, prefira aplicá-los encadeando `.where()` ou usando listas para clareza.

7.5. Joins sem `label()` podem quebrar sua API

Problema: você faz um join e usa `.select()` com duas colunas `nome` ? uma da tabela `Usuario` e outra da tabela `Empresa`. Resultado: a última sobrescreve a anterior no dicionário.

Solução: sempre use `.label("alias")` em colunas de outras tabelas:

.select(Usuario.nome, Empresa.nome.label("empresa"))

7.6. Atualizações em massa exigem cuidado

Problema: você busca múltiplos registros com `.toList()` e faz update em loop, mas esquece que cada `update()` abre e fecha uma sessão separada. Em alguns bancos, isso pode gerar deadlocks.

Solução: em updates críticos, use `.rawSql()` ou agrupe as atualizações em uma única transação personalizada com `session.begin()`.

7.7. Documente os models gerados automaticamente

Problema: o gerador de models cria tudo corretamente, mas você se esquece de adicionar docstrings, comentários ou validações personalizadas depois.

Solução: após gerar os arquivos com `generate_models.py`, revise cada um, documente, adicione métodos específicos do domínio e centralize regras de negócio simples ali mesmo.

7.8. Use `rawSql()` com moderação (mas sem medo)

Problema: você evita `rawSql()` por medo de perder o controle ? ou usa demais, burlando toda a abstração ORM.

Solução: `rawSql()` é ótimo para relatórios complexos, views materializadas ou queries com CTEs. Use com moderação, mas sem preconceito.

Dominar a arquitetura não é apenas aprender como usá-la, mas também como **não** usá-la. Uma arquitetura elegante precisa de disciplina para se manter limpa, segura e eficiente.

No próximo capítulo, fecharemos com orientações sobre testes, extensões futuras e como essa arquitetura pode evoluir junto com seu sistema.

Capítulo 8 ? Testes, Extensões e o Futuro: adaptando sua arquitetura para crescer com você

Neste último capítulo, vamos dar um passo além da implementação: vamos falar sobre **manutenção**, **evolução** e, principalmente, como garantir que sua arquitetura continue funcionando à medida que sua aplicação cresce, muda, escala e ? inevitavelmente ? quebra.

Um código funcional é ótimo. Um código testável, extensível e resiliente é o que diferencia um desenvolvedor pragmático de um desenvolvedor verdadeiramente profissional.

8.1. Testes automatizados com SQLAlchemy

Testar é a única forma confiável de dormir tranquilo enquanto o deploy roda na sexta-feira. E o melhor: a arquitetura deste projeto já está preparada para isso.

Estratégia recomendada

Use `pytest` como framework principal de testes.

Crie um banco de testes separado (pode até ser SQLite em memória para velocidade).

Use `Base.metadata.create_all()` para criar as tabelas temporariamente antes dos testes.

Use `session.begin()` ou `session.rollback()` para garantir isolamento dos testes.

Exemplo de teste simples

```
import pytest

from models.usuario import Usuario

from models.db import SessionLocal, Base, get_engine



@pytest.fixture(scope="function")

def session():

engine = get_engine()

Base.metadata.create_all(bind=engine)

session = SessionLocal()
```

```
yield session

session.rollback()

session.close()



def test_usuario_insert(session):

novo = Usuario.insert(nome="Teste", email="teste@a.com")

assert novo.id is not None
```

Dica de ouro:

Evite reusar sessões entre testes. Cada teste deve começar com banco limpo.

8.2. Como estender o CRUDMixin

Embora `CRUDMixin` já cubra as operações básicas (insert, update, delete, get, all, rawSql...), você pode perfeitamente personalizá-lo para as regras do seu sistema.

Exemplo: adicionando um método soft delete

class CustomMixin(CRUDMixin):

def soft_delete(self):

self.update(ativo=False)

Depois, use isso em seus modelos:

class Usuario(Base, CustomMixin):

...

Você também pode sobrescrever `insert()` ou `update()` para aplicar validações específicas antes do commit.

8.3. Integração com frameworks web

Essa arquitetura se encaixa perfeitamente com frameworks como Flask, FastAPI e até Django (em projetos com arquitetura hexagonal).

Exemplo com FastAPI

@app.get("/usuarios")

def listar():

return Usuario.all().isTrue("ativo").orderBy("nome").toDict()

Cuidado:

Use sempre sessões curtas e encapsuladas em rotas para evitar problemas de concorrência.

8.4. Arquitetura futura: sugestões de evolução

A estrutura atual funciona muito bem em projetos médios. Mas, conforme o sistema cresce, é saudável planejar algumas melhorias:

Sugestões práticas:

- Separar domínio em módulos (ex: `usuario/`, `produto/`, `financeiro/`)
- Implementar validação de dados com Pydantic (ou Marshmallow)
- Adicionar versionamento de migrations com Alembic
- Criar uma interface de repositório desacoplada do ORM
- Configurar logs SQL de performance em produção

E por que não?

- Incluir suporte nativo a cache Redis no `QueryChain`
- Criar filtros automáticos a partir de query params
- Integrar com fila de eventos para registrar alterações

Ao longo deste manual, você viu como uma estrutura bem desenhada com SQLAlchemy pode ir muito além do básico. Agora você tem em mãos não apenas uma biblioteca ? mas uma base sólida, expressiva e expansível para qualquer aplicação profissional.

Manual ORM com SQLAlchemy

Mais do que saber usar, você sabe adaptar, refatorar e evoluir.

Missão cumprida.