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

Índice

| [Introdução | o](#intro | duçã | 0) | | | | | | | | | | | |
|-------------|-----------|---------|----------|-----------|---------|----------|--------|--------|----------|---------|------------|---------|------------|--------|
| [Capítulo 1 | l ? Ente | nden | do a Arq | juitetura | a do P | rojeto](| #capít | ulo-1 | enten | dendo | -a-arquite | etura-d | lo-projeto | o) |
| [Capítulo | 2 | | ? | Pre | parano | ob | 0 | | Ambie | nte: | do | | caos | ao |
| controle](# | capítulo | o-2p | reparan | do-o-ar | nbient | e-do-ca | aos-ao | -cont | role) | | | | | |
| [Capítulo | 3 | ? | Geran | ido | os | Mode | ls | autor | naticam | ente: | mend | os | tédio, | mais |
| produtivida | ade](#ca | apítulo | o-3gera | ando-o | s-mode | els-aut | omatic | amer | ite-men | os-téc | dio-mais-p | oroduti | vidade) | |
| [Capítulo | 4 ? | 0 | Poder | do C | CRUDI | Mixin: | crian | do, | lendo, | atua | lizando | e de | eletando | com |
| graça](#ca | pítulo-4 | о-р | oder-do- | crudmi | xin-cri | ando-le | endo-a | tualiz | ando-e | -deleta | ando-com | n-graça | a) | |
| [Capítulo | 5 | ? | Quer | yChair | n: a | a a | ırte | de | con | sultar | com | 0 | um ı | mestre |
| zen](#capí | tulo-5 | query | chain-a- | arte-de | e-consi | ultar-co | mo-ur | n-me | stre-zer | n) | | | | |
| [Capítulo | 6 ? | Cas | sos de | Uso | Rea | iis: q | uando | 0 | banco | de | dados | enco | ontra a | vida |
| real](#capi | ítulo-6 | casos | s-de-uso | -reais- | quando | o-o-bar | nco-de | -dadc | s-enco | ntra-a | -vida-rea | l) | | |
| [Capítulo | 7 | ? | Boas | Prática | as, | Armad | ilhas | Co | muns | е | Como | Evita | ar Tra | gédias |
| Anunciada | ıs](#cap | ítulo- | 7boas- | prática | s-arma | adilhas | -comu | ns-e- | como-e | vitar-t | ragédias- | anunc | iadas) | |
| [Capítulo | 8 ? | Teste | s, Exte | nsões | е о | Futur | o: ad | aptar | ndo su | a ard | quitetura | para | crescei | com |
| você](#cap | oítulo-8- | -teste | es-exten | sões-e | -o-futu | ro-ada | otando | -sua- | arquite | tura-p | ara-cresc | er-con | n-você) | |

Introdução

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

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.

| 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: |
| 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": {

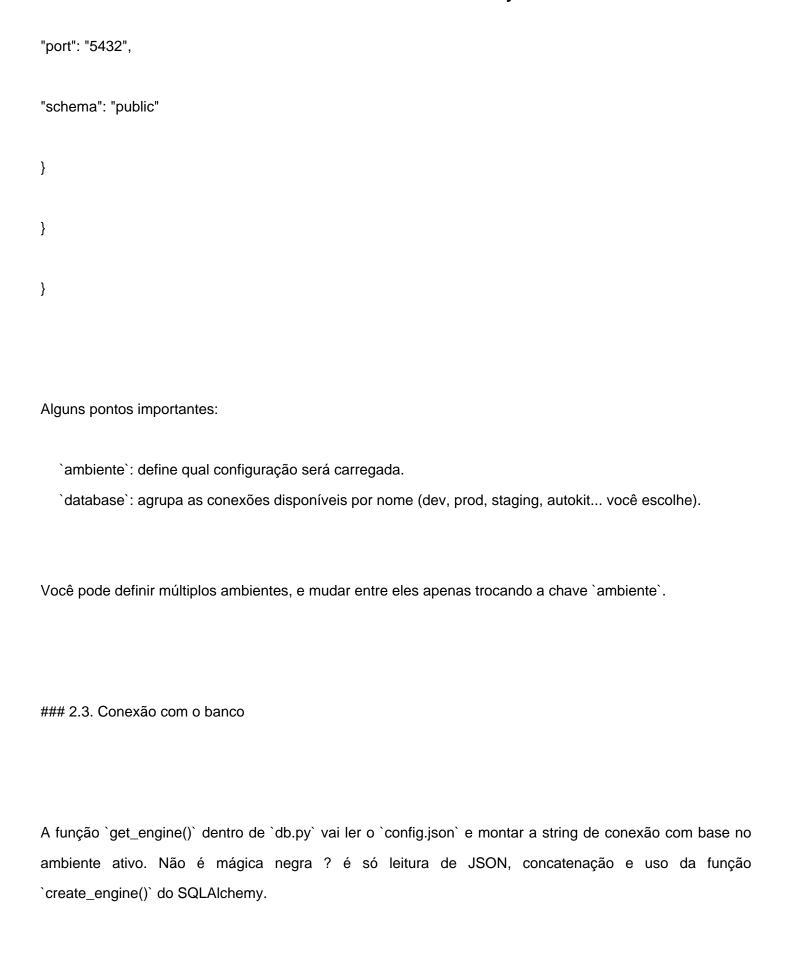
"database": "pgsql",

"dbname": "meubanco",

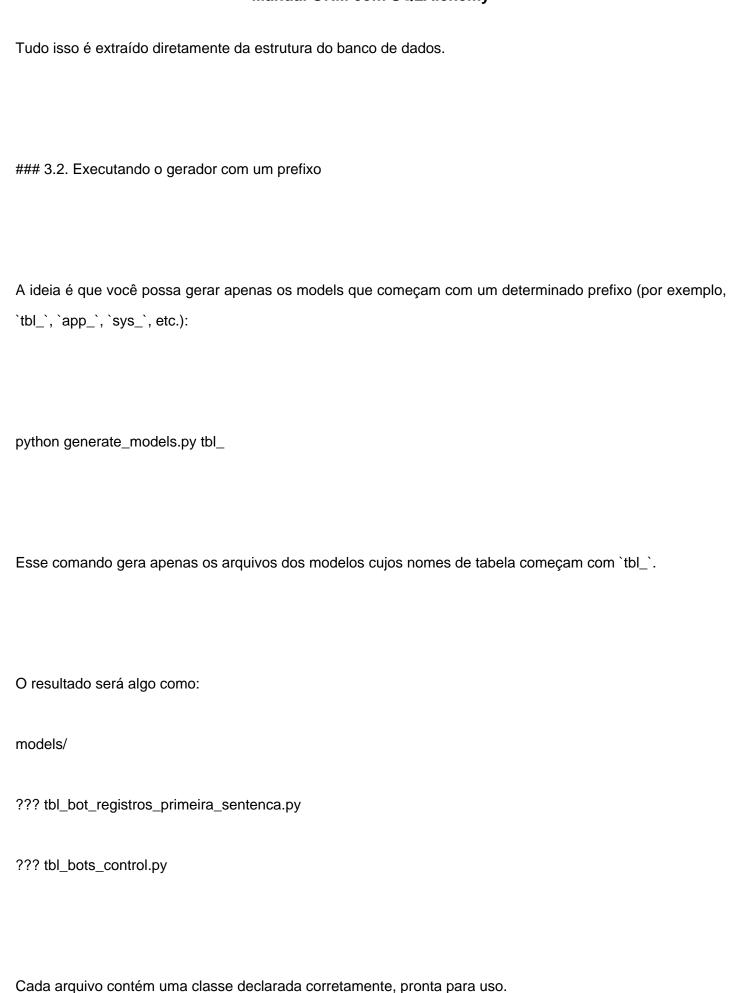
"user": "meuusuario",

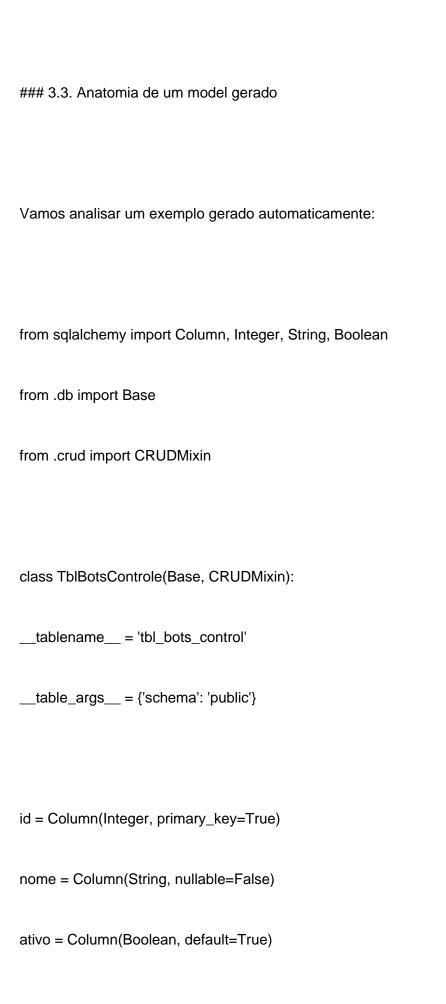
"password": "minhasenha",
```

"host": "127.0.0.1",



| O `SessionLocal` também é criado ali, e será usado para abrir conexões de sessão de forma segura isolada Capítulo 3 ? Gerando os Models automaticamente: menos tédio, mais produtividade |
|--|
| Capítulo 3 ? Gerando os Models automaticamente: menos tédio, mais produtividade |
| 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 sabé o tipo de tarefa que testa sua sanidade. Este projeto elimina essa tortura com um script que automati 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, den 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 |





| 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 právimo conítulo, você apronderá a crier ao tabolas no banco com um único comendo, a entenderá como |
| 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. |
| isso se conecta com os modelos gerados. |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| Capítulo 4 ? CRUDMixin: o motor silencioso por trás da mágica |

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.

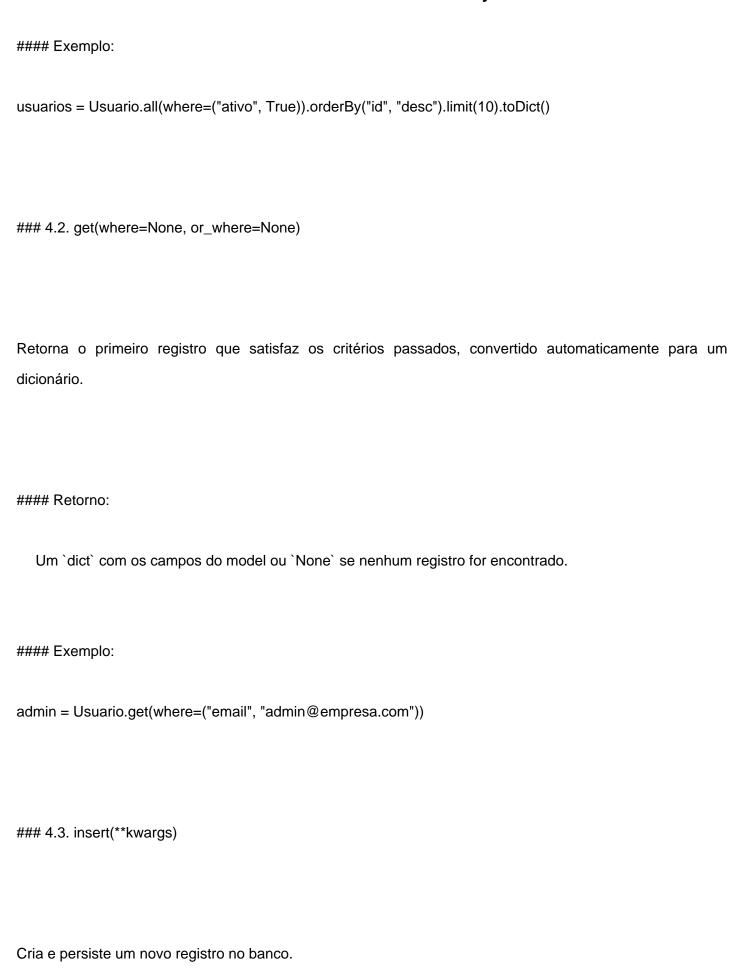
Parâmetros:

`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:

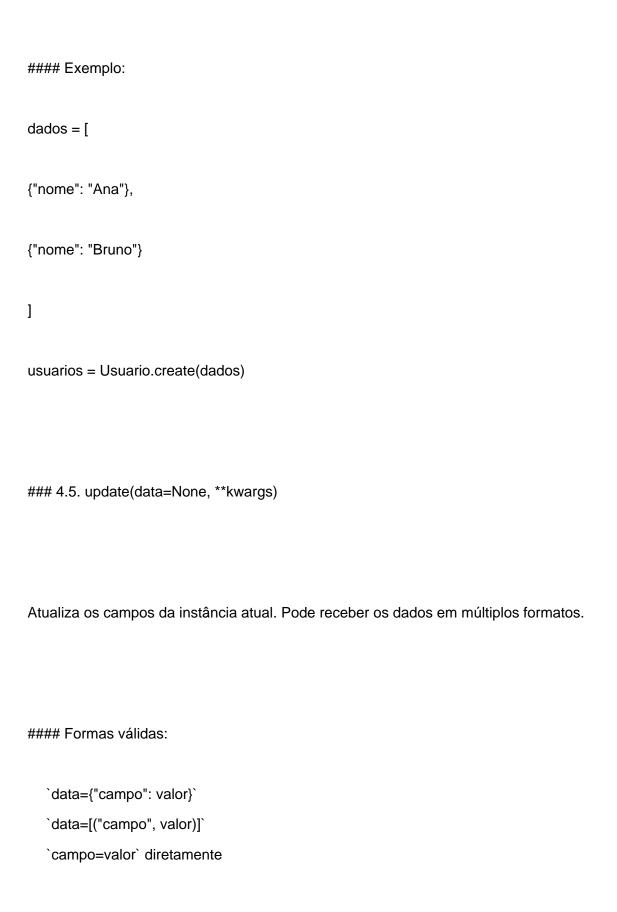
Um objeto `QueryChain`, que precisa ser finalizado com um método de execução, como `.toList()`, `.toDict()`, `.first()`, etc.



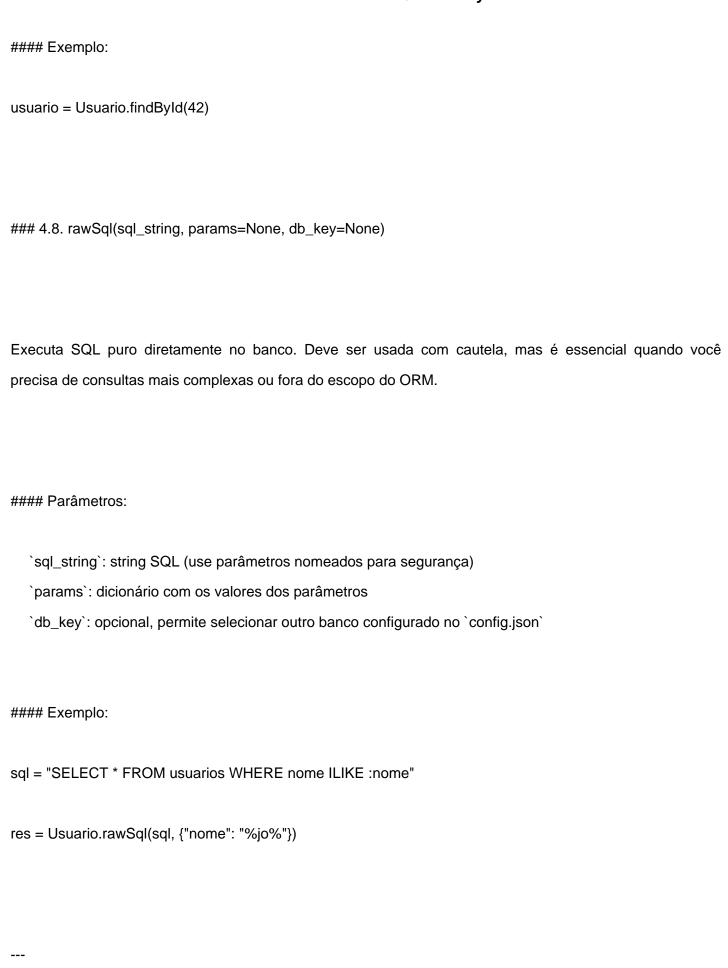
| #### Parâmetros: |
|---|
| Argumentos nomeados, correspondendo às colunas da tabela. |
| |
| #### Retorno: |
| A instância do model recém-criada. |
| |
| #### Exemplo: |
| novo_usuario = Usuario.insert(nome="Fernanda", email="fer@empresa.com", ativo=True) |
| |
| |
| ### 4.4. create(records) |
| |
| |
| 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: |
|---|
| 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 |



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

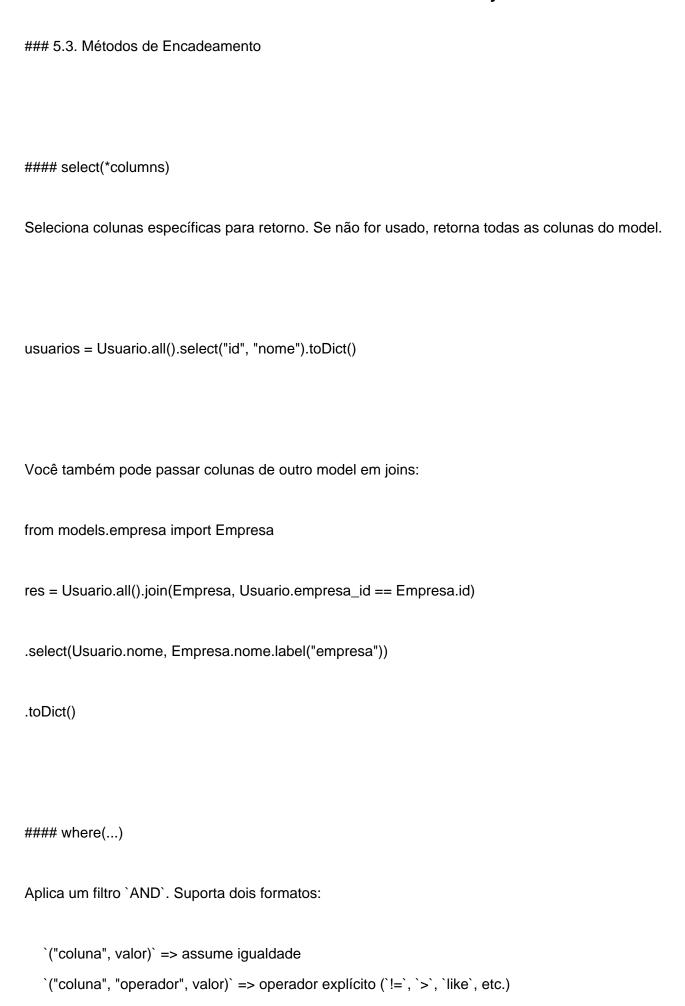
A `QueryChain` resolve exatamente isso. Ela encapsula uma `query` SQLAlchemy e fornece uma interface

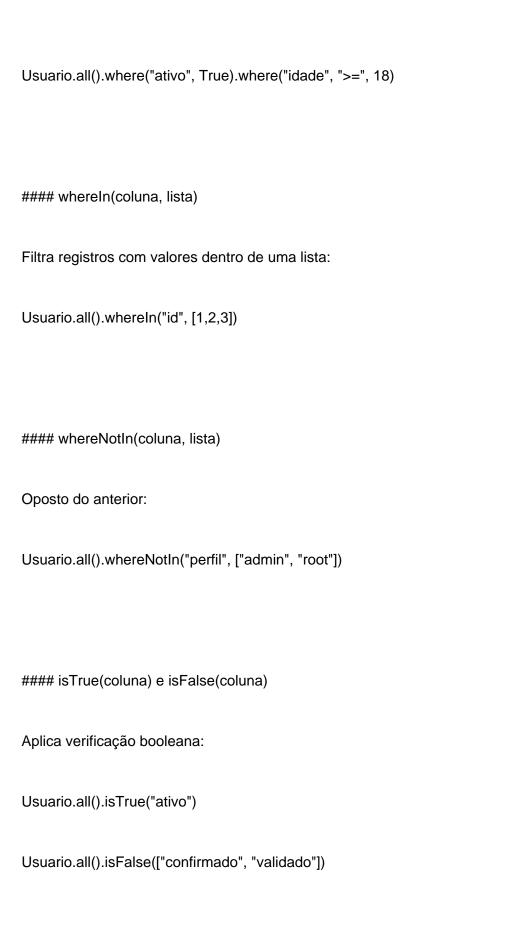
fluente (inspirada em bibliotecas como ¡Query e LINQ) que permite encadear modificadores e só executar a

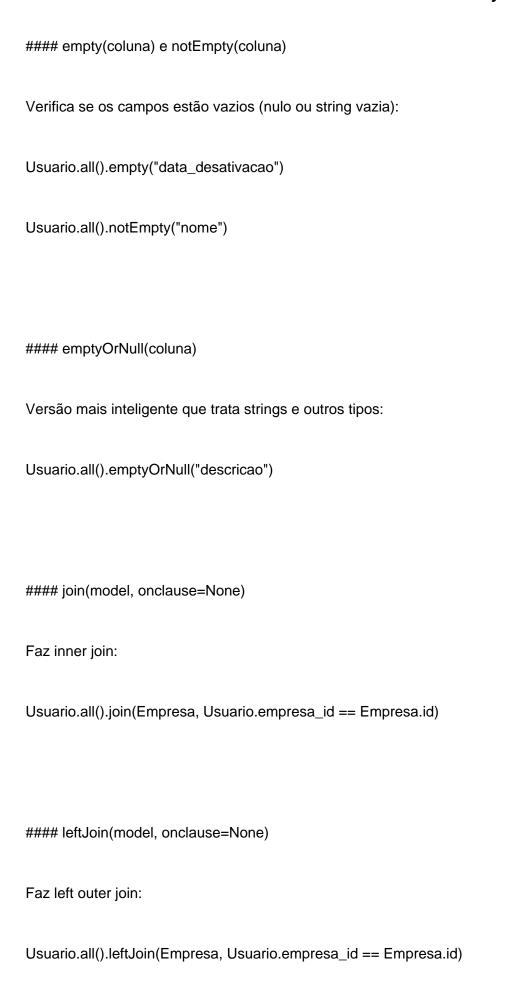
query no final, com um dos métodos de execução.

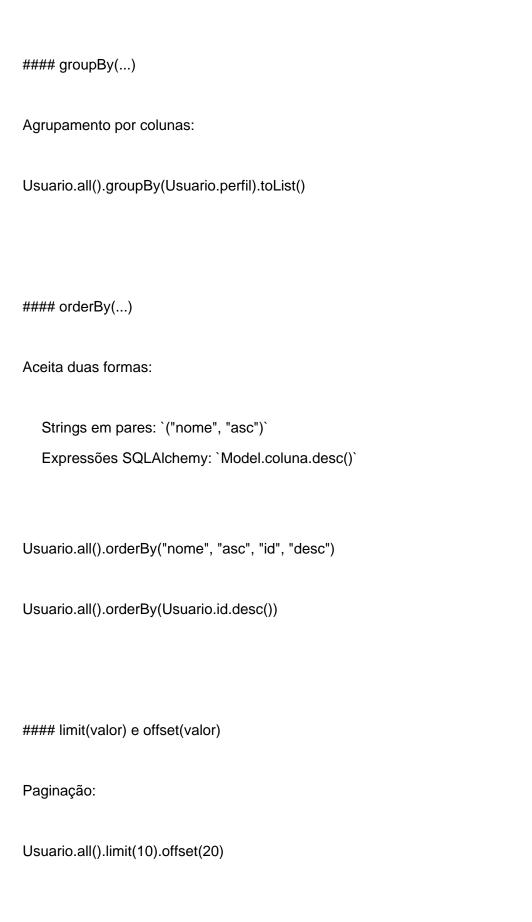
Vamos olhar para isso em profundidade, analisando cada método que você pode usar e como ele altera o

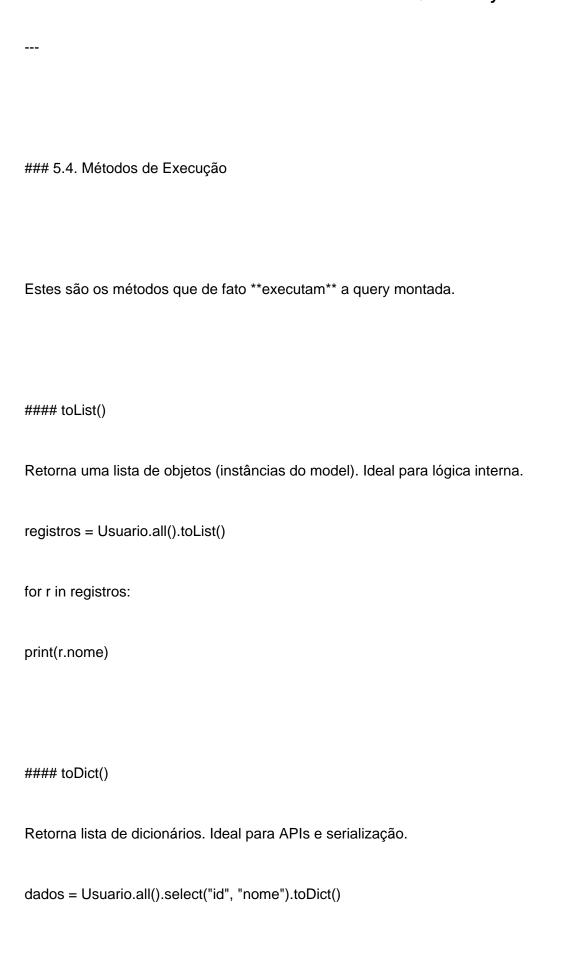
estado da consulta.













5.5. Exemplo prático completo



| Manual ORM com SQLAIchemy | |
|---|--|
| | |
| | |
| | |
| | |
| | |
| | |
| 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

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

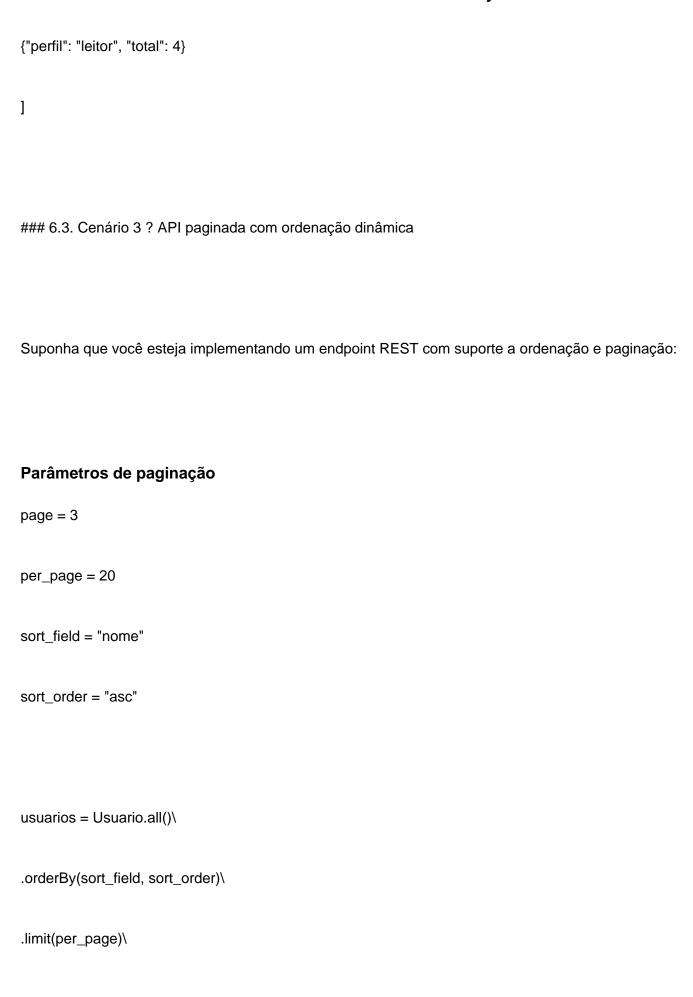
cogumelos.

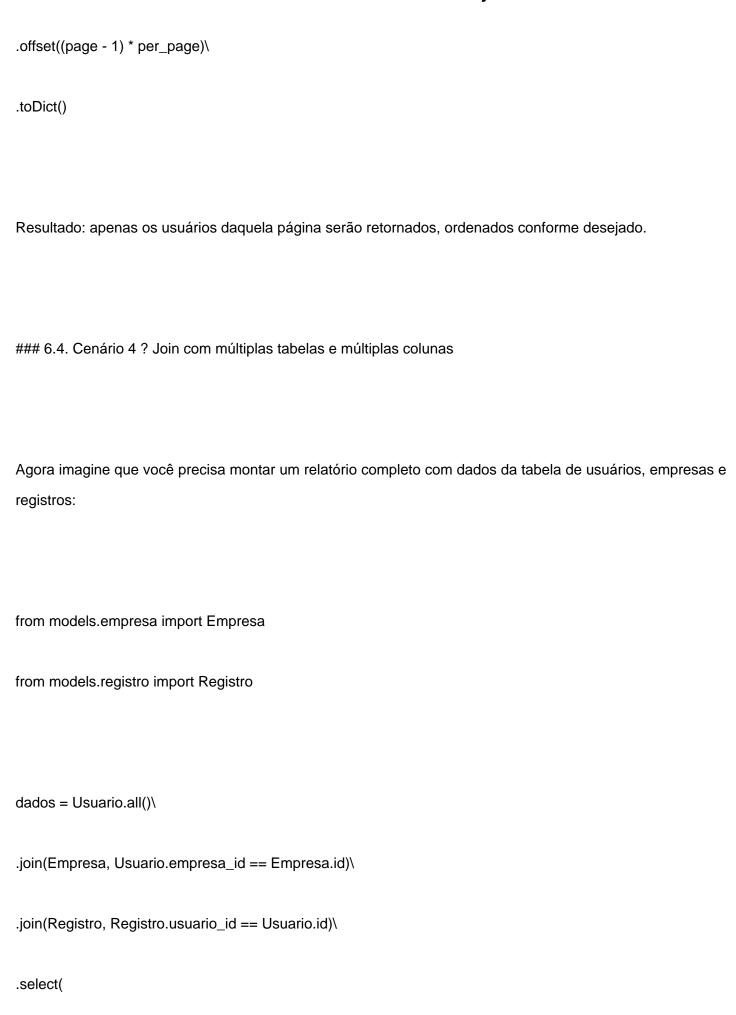
com endpoints REST.

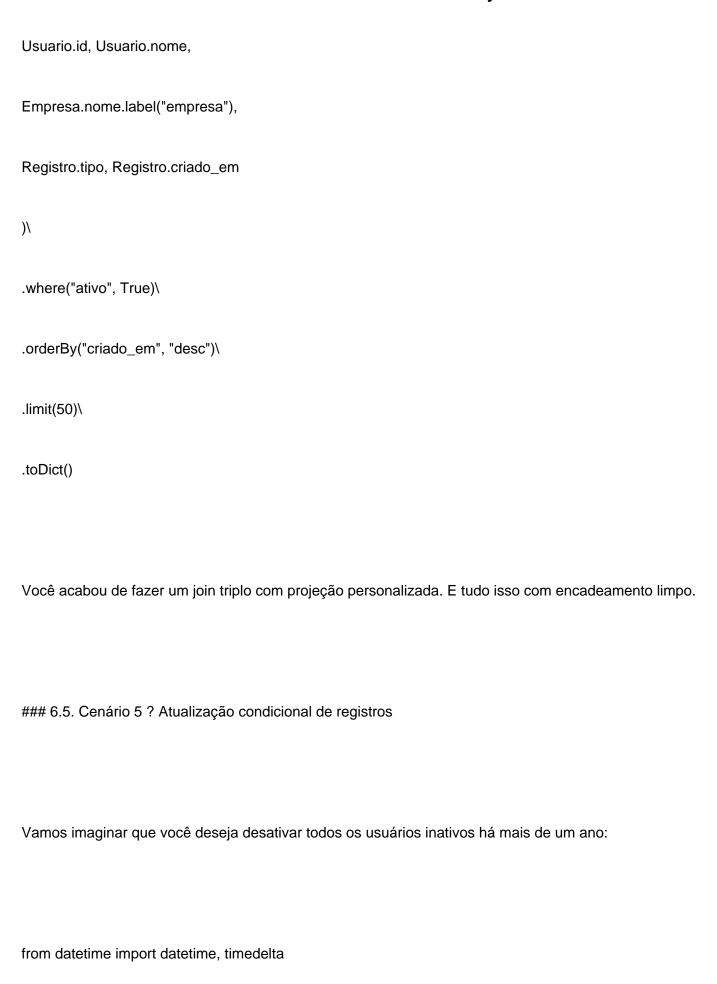
| ### 6.1. Cenário 1 ? Consulta condicional com parâmetros dinâmicos |
|---|
| Imagine que você tem um endpoint de API que aceita múltiplos parâmetros de filtro: |
| Parâmetros vindos de uma requisição |
| params = { |
| "ativo": True, |
| "perfil": "editor", |
| "idade_min": 25, |
| "busca": "silva" |
| } |
| Queremos montar a consulta com base apenas nos parâmetros fornecidos (e ignorar os ausentes). Com QueryChain, isso é simples: |
| query = Usuario.all() |



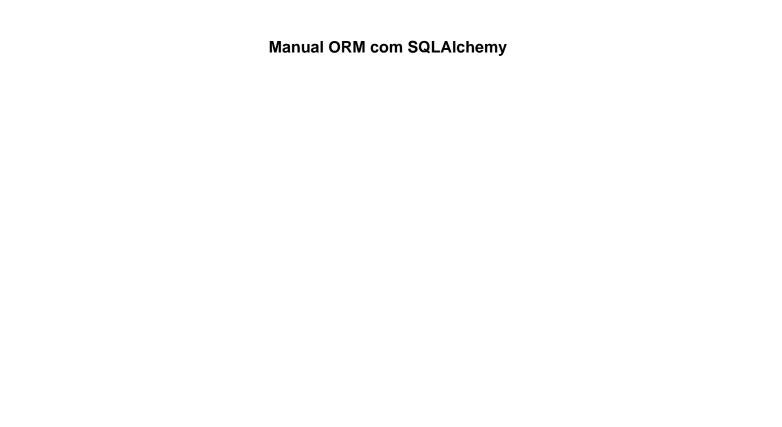
| ### 6.2. Cenário 2 ? Gerando relatório com agregações |
|---|
| Vamos supor que você precise de um relatório que conte o número de usuários por perfil. Aqui entra o `groupBy()`: |
| from sqlalchemy import func |
| resumo = Usuario.all()\ |
| .select(Usuario.perfil, func.count(Usuario.id).label("total"))\ |
| .groupBy(Usuario.perfil)\ |
| .orderBy("total", "desc")\ |
| .toDict() |
| |
| O resultado será uma lista como: |
| [|
| {"perfil": "admin", "total": 12}, |
| {"perfil": "editor", "total": 7}, |











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

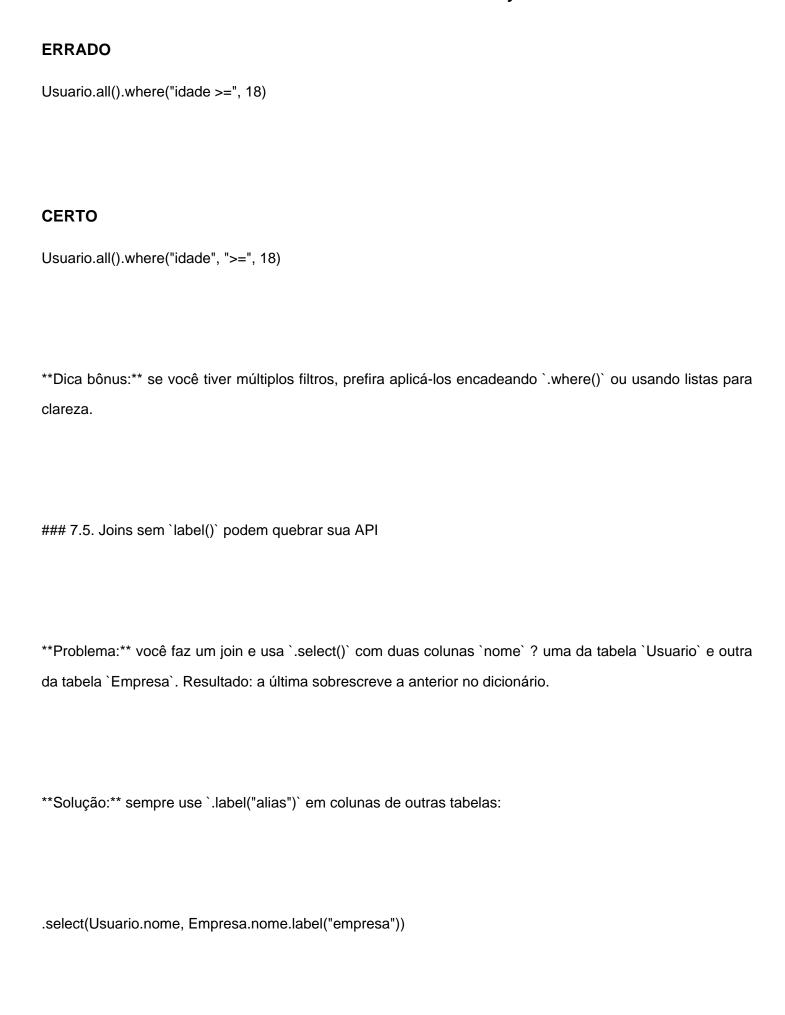
Chegamos à parte em que a teoria encontra o campo de batalha. Não importa o quão poderosa seja sua arquitetura: se mal utilizada, ela pode se transformar em um festival de bugs, lentidão e frustração. Este capítulo é um guia de sobrevivência: ele não só revela as boas práticas que você deve seguir, mas 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



| 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: |



| ### 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 `rawSqI()` 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. |
| |
| |

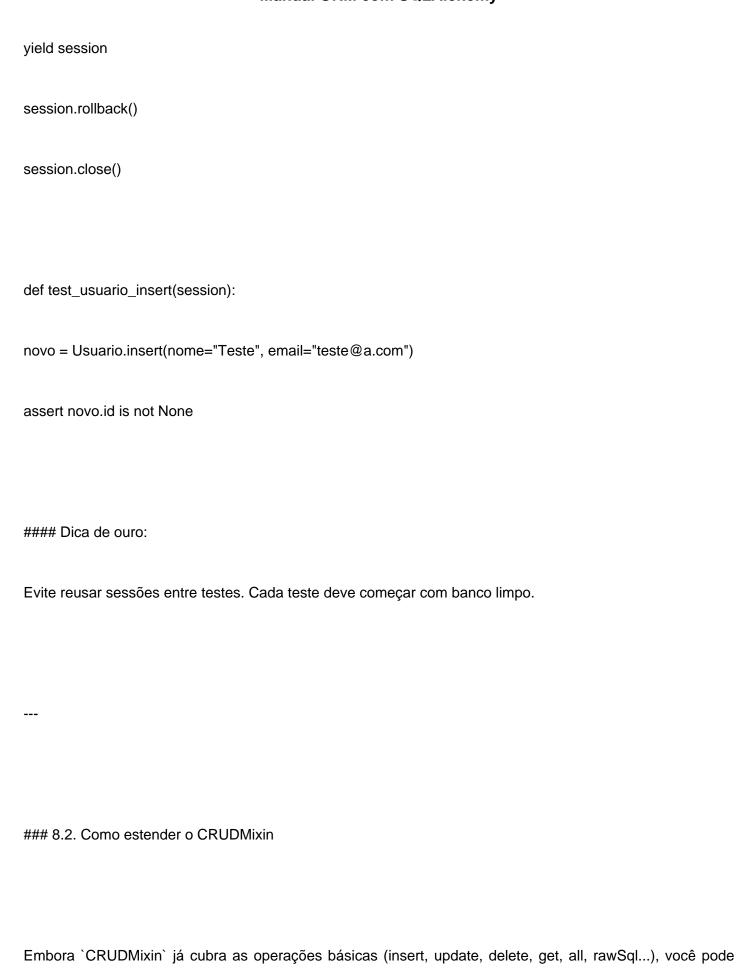


Testar é a única forma confiável de dormir tranquilo enquanto o deploy roda na sexta-feira. E o melhor: a

8.1. Testes automatizados com SQLAlchemy

arquitetura deste projeto já está preparada para isso.

| #### Estrategia 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(): |
| doi occion(). |
| engine = get_engine() |
| |
| Base.metadata.create_all(bind=engine) |
| |
| session = SessionLocal() |



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. |
| |
| |

| 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.

| Mais do que saber usar, você sabe adaptar, refatorar e evoluir. |
|---|
| |
| |
| Missão cumprida. |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |