SQLAlchemy

conceitos básicos

Live de Python #258

Roteiro

1. O básico

Uma introdução ao sqlalchemy

2. Core

Funcionalidades principais (engine, dialetos, pool transações, schemas, types e query builder)

3. ORM

Camada de mais alto nível do sqlalchemy

4. Alguns pontos adicionais

Coisas que eu gostaria de complementar, mas não tem um tópico próprio

apoia.se/livedepython

pix.dunossauro@gmail.com

patreon.com/dunossauro

Ajude o projeto <3

Ademar Peixoto, Adilson Herculano, Alemao, Alexandre Harano, Alexandre Lima, Alexandre Takahashi, Alexandre Villares, Alfredo Braga, Alisson Souza, Alysson Oliveira, Andre Azevedo, Andre Mesquita, Andre Paula, Aniltair Filho, Antônio Filho, Arnaldo Turque, Aslay Clevisson, Aurelio Costa, Bárbara Grillo, Ber_dev_2099, Bernardo Fontes, Bruno Almeida, Bruno Barcellos, Bruno Batista, Bruno Freitas, Bruno Ramos, Caio Nascimento, Carlos Ramos, Cristian Firmino, Daniel Bianchi, Daniel Freitas, Daniel Wojcickoski, Danilo Boas, Danilo Silva, David Couto, David Kwast, Davi Souza, Dead Milkman, Denis Bernardo, Dgeison Peixoto, Diego Guimarães, Dino, Edgar, Edilson Ribeiro, Emerson Rafael, Ennio Ferreira, Erick Andrade, Érico Andrei, Everton Silva, Fabio Barros, Fábio Barros, Fabio Valente, Fabricio Biazzotto, Felipe Augusto, Felipe Rodrigues, Fernanda Prado, Fernando Celmer, Flávio Meira, Francisco Silvério, Frederico Damian, Gabriel Espindola, Gabriel Mizuno, Gabriel Moreira, Gabriel Paiva, Gabriel Ramos, Gabriel Simonetto, Giovanna Teodoro, Giuliano Silva, Guilherme Beira, Guilherme Felitti, Guilherme Gall, Guilherme Ostrock, Guilherme Piccioni, Guilherme Silva, Gustavo Suto, Haelmo Almeida, Harold Gautschi, Heitor Fernandes, Helvio Rezende, Henrique Andrade, Henrique Sebastião, Hiago Couto, Higor Monteiro, Igor Taconi, Janael Pinheiro, Jean Victor, Jefferson Antunes, Joelson Sartori, Jonatas Leon, Jônatas Oliveira, Jônatas Silva, Jorge Silva, Jose Barroso, Jose Edmario, José Gomes, Joseíto Júnior, Jose Mazolini, Josir Gomes, Juan Felipe, Juliana Machado, Julio Batista-silva, Julio Franco, Júlio Gazeta, Júlio Sanchez, Kaio Peixoto, Leandro Silva, Leandro Vieira, Lengo, Leonan Ferreira, Leonardo Mello, Leonardo Nazareth, Lucas Carderelli, Lucas Lattari, Lucas Mello, Lucas Mendes, Lucas Nascimento, Lucas Schneider, Lucas Simon, Luciano Filho, Luciano Ratamero, Luciano Teixeira, Luiz Carlos, Luiz Duarte, Luiz Lima, Luiz Paula, Mackilem Laan, Marcelo Araujo, Marcio Silva, Marco Mello, Marcos Gomes, Marina Passos, Mateus Lisboa, Matheus Silva, Matheus Vian, Mírian Batista, Mlevi Lsantos, Murilo Carvalho, Murilo Viana, Nathan Branco, Ocimar Zolin, Otávio Carneiro, Pedro Henrique, Pedro Pereira, Peterson Santos, Philipe Vasconcellos, Pytonyc, Rafael Araújo, Rafael Faccio, Rafael Romão, Raimundo Ramos, Ramayana Menezes, Renato José, Renato Moraes, Rene Pessoto, Renne Rocha, Ricardo Combat, Ricardo Silva, Rinaldo Magalhaes, Riverfount, Rjribeiro, Rodrigo Barretos, Rodrigo Oliveira, Rodrigo Quiles, Rodrigo Santana, Rodrigo Vaccari, Rodrigo Vieira, Rogério Nogueira, Rui Jr, Samanta Cicilia, Santhiago Cristiano, Selmison Miranda, Téo Calvo, Thiago Araujo, Thiago Borges, Tiago Henrique, Tony Dias, Tyrone Damasceno, Valdir, Valdir Tegon, Varlei Menconi, Vcwild, Vinícius Costa, Vinicius Stein, Vladimir Lemos, Washington Teixeira, Willian Lopes, Willik Araujo, Wilson Duarte, Zeca Figueiredo

Obrigado você

SQL Alchemy

O básico!

SQLAlchemy

Um kit de ferramentas para trabalhar com bancos de dados SQL e Python. A ideia principal é "abstrair" as chamadas diretas para drivers específicos de bancos de dados.

O toolkit suporta:

- Async e Sync
- Anotações de tipos e forma dinâmica
- Implementa a DBAPI ([PEP 249)
- Suporta cPython e PyPy
- ORM
- Sistemas de plugins
- Eventos

SQLAlchemy

- Iniciado em **2005**, primeira release em **2006**
- Criado por Michael Bayer
  - SQLAlchemy
  - Mako
  - Alembic
  - ...
- Atualmente na versão **2.0.27**
- Licença **MIT**
- Cross Plataforma

https://github.com/zzzeek
https://techspot.zzzeek.org/

https://decisionstats.com/2015/12/29/interview-mike-bayer-sqlalchemy-pydata-python/

pip install sqlalchemy

Instalação

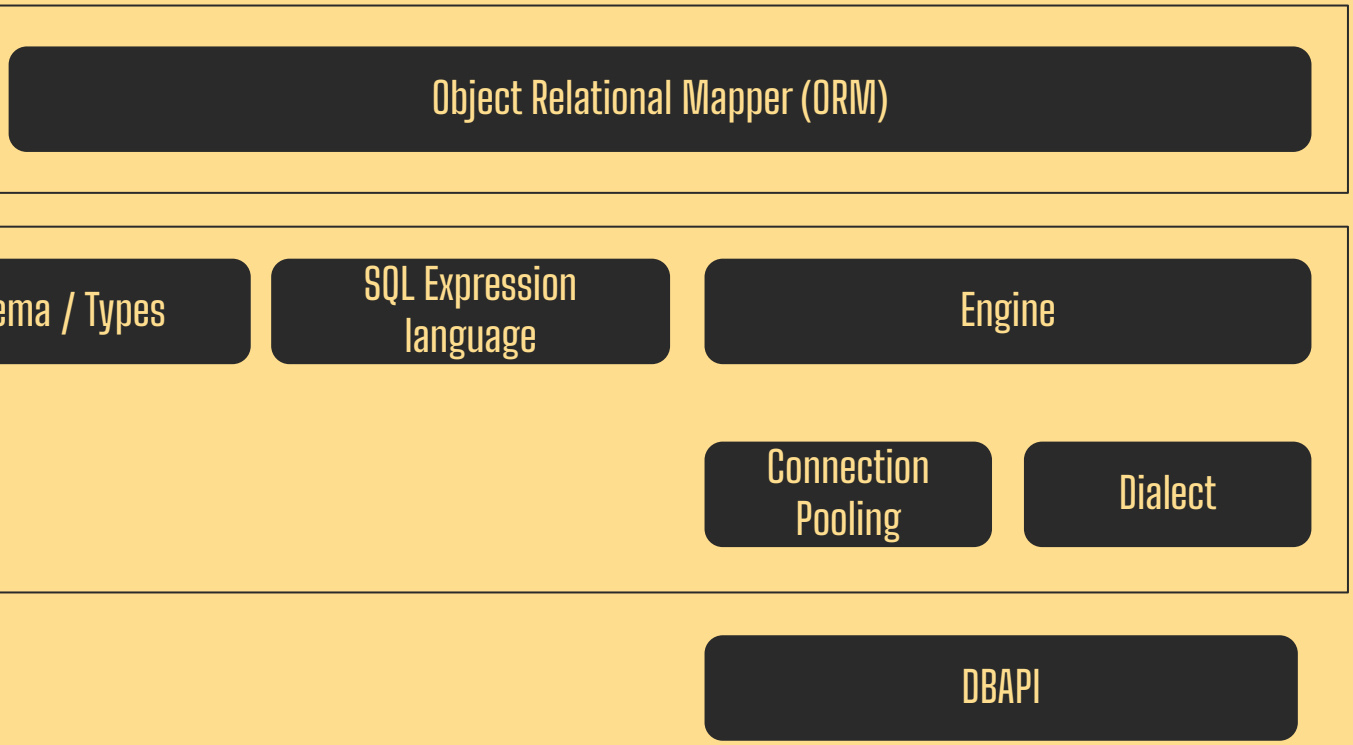
pip install sqlalchemy

pip install aiolite # vamos usar o sqlite async em algum momento

pip install psycopg-binary # para mostrar a diferença quando usamos postgres

Instalação

Um entendimento geral



https://docs.sqlalchemy.org/en/20/_images/sqla_arch_small.png

CORE

A base de tudo!

Core

O core é o componente mais básico do SQLAlchemy. Responsável por criar a conexão com o banco de dados, fazer buscas e definir tipos.

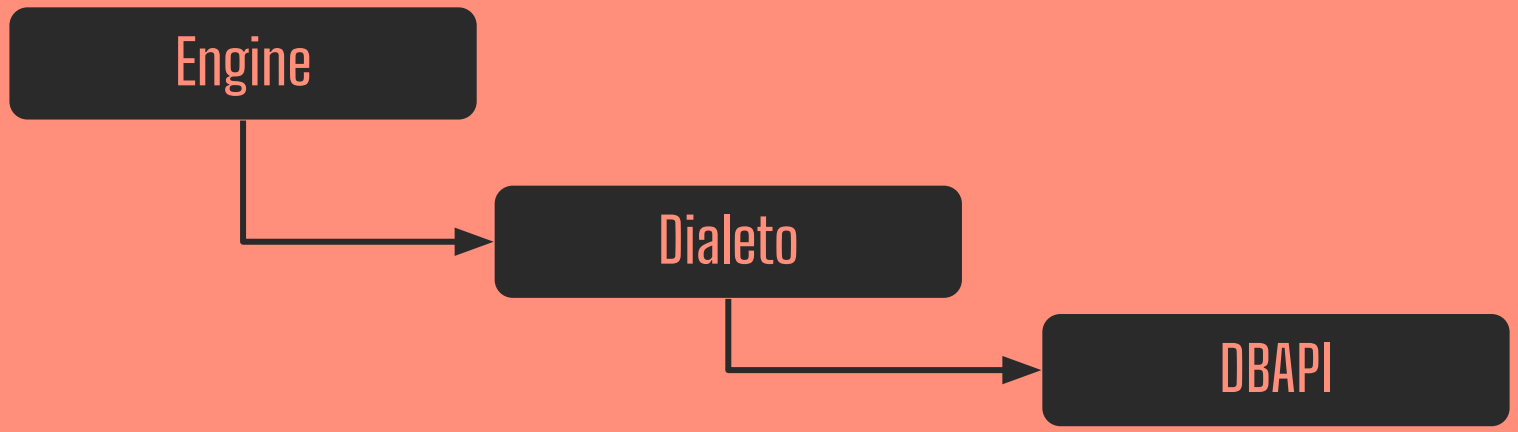
Alguns componentes importantes são:

- **Engine**
  - **Connection**: Interface para se comunicar com o banco
  - **Dialect**: Mecanismos específicos para cada banco de dados
  - **Pool**: Deixa conexões em memória para ser mais fácil reutilizar
- **SQL Expression Language**: Construções em Python para representar SQL
- **Schema/Types**: Construções em python que representam tabelas, colunas e tipos de dados

Engine

A engine, o coração do Alchemy, é uma fábrica de conexões com o banco de dados.

O objetivo dela é que de forma dinâmica podemos nos comunicar com diferentes drivers de banco de dados usando dialetos específicos para cada banco de dados.



Engine em código

```python
# exemplo_00.py
from sqlalchemy import create_engine


engine = create_engine('sqlite:///database.db')

print(engine) # Engine(sqlite:///database.db)
```

Dialetos

A engine fabrica uma conexão com a base de dados específica usando os dialetos. Dialetos são chamadas diretas para os drivers específicos para databases específicos.

Por exemplo, o SQLAlchemy suporta nativamente:

- SQLite
- PostgreSQL
- MySQL / MariaDB
- Oracle
- Microsoft SQL Server

Contando com diversas implementações via plugins como CockroachDB, Firebird, Amazon Redshift, etc..

No código

```python
from sqlalchemy import create_engine

engine_pg = create_engine('postgresql+psycopg://...')
print(engine_pg.dialect)
# <sqlalchemy.dialects.postgresql.psycopg.PGDialect_psycopg object at ...>

engine_lite = create_engine('sqlite:///database.db')
print(engine_lite.dialect)
# <sqlalchemy.dialects.sqlite.pysqlite.SQLiteDialect_pysqlite object at ...>
```

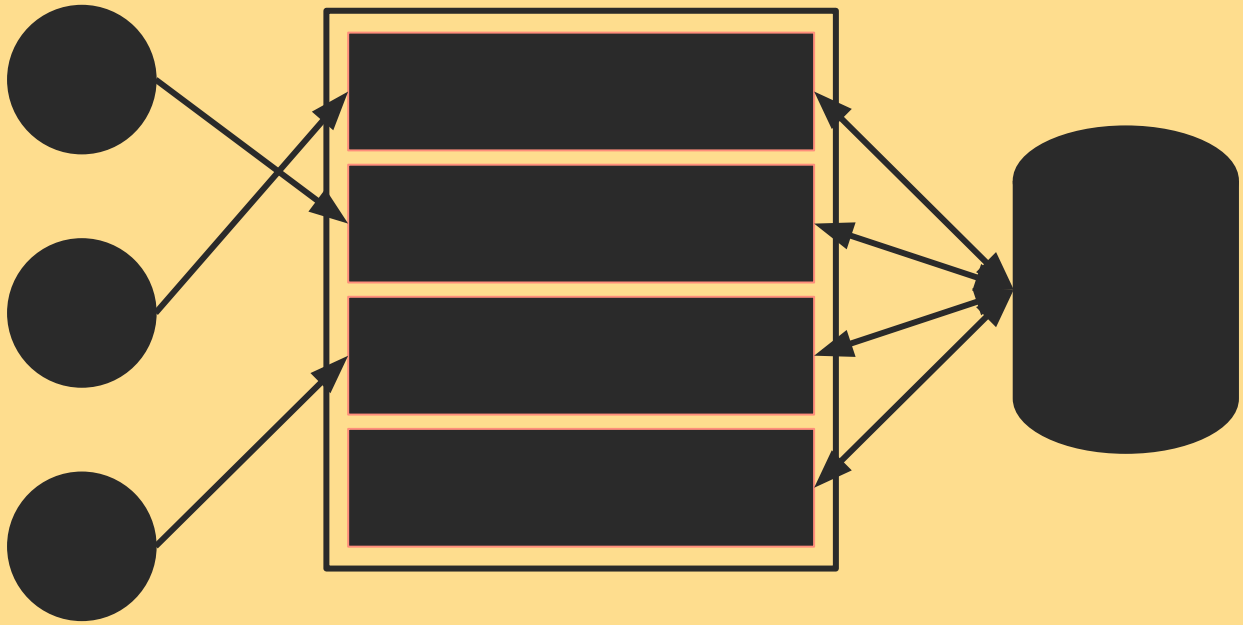
Conexão

Com a engine conhecendo o dialeto especificado para conexão, ela pode iniciar a comunicação com o banco:

```
1  from sqlalchemy import create_engine
2  engine = create_engine('sqlite:///database.db', echo=True)
3
4  connection = engine.connect()
5  print(connection.connection.dbapi_connection)
6  # <sqlite3.Connection object at 0x747eb38e8f40>
7  connection.close()
```

Pool

Uma instrução relativamente cara de IO é criação da conexão com o banco de dados. Por esse motivo, o sqlalchemy armazena as conexões em um `reservatório` de conexões. Chamamos de `Pool`.



Pool

Quando uma conexão é fechada, ela é fechada no contexto do sqlalchemy, mas se mantém aberta na pool. Dessa forma o escalonamento acontece de forma transparente. Quando novas conexões forem abertas, caso existam conexões no pool, elas serão atribuídas.

```python
# Exemplo_02.py
print(engine.pool)
# <sqlalchemy.pool.impl.QueuePool object at 0x73261435c770>

con_1 = engine.connect()
print(engine.pool.status())
# Pool size: 5  Connections in pool: 0 Current Overflow: -4 Current Checked out connections: 1

con_2 = engine.connect()
print(engine.pool.status())
# Pool size: 5  Connections in pool: 0 Current Overflow: -3 Current Checked out connections: 2
```

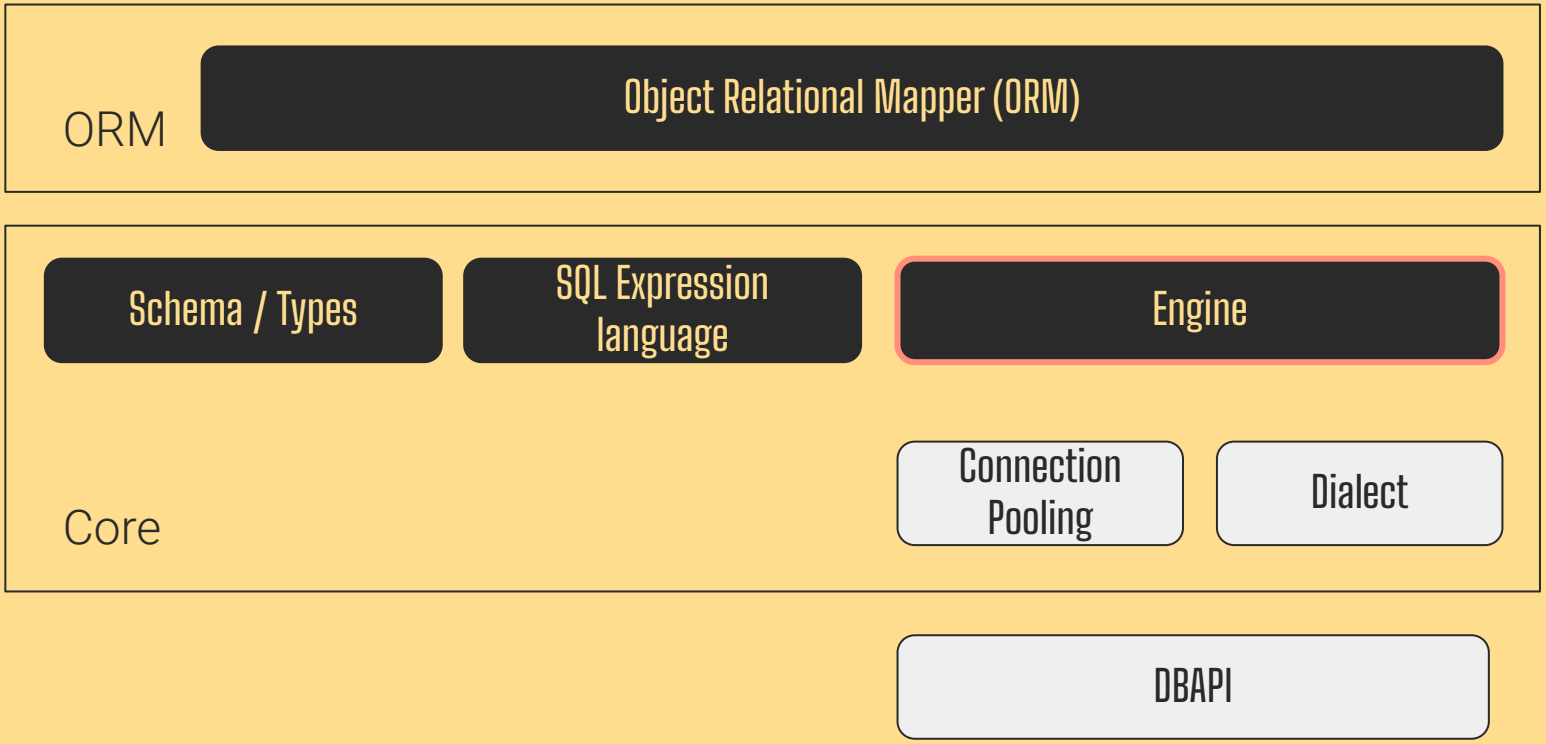
Pool

Quando uma conexão é fechada, ela é fechada no contexto do sqlalchemy, mas se mantém aberta na pool. Dessa forma o escalonamento acontece de forma transparente. Quando novas conexões forem abertas, caso

```
con_1.close()
print(engine.pool.status())
# Pool size: 5  Connections in pool: 1 Current Overflow: -3 Current Checked out connections: 1

con_3 = engine.connect()
print(engine.pool.status())
# Pool size: 5  Connections in pool: 0 Current Overflow: -3 Current Checked out connections: 2
```

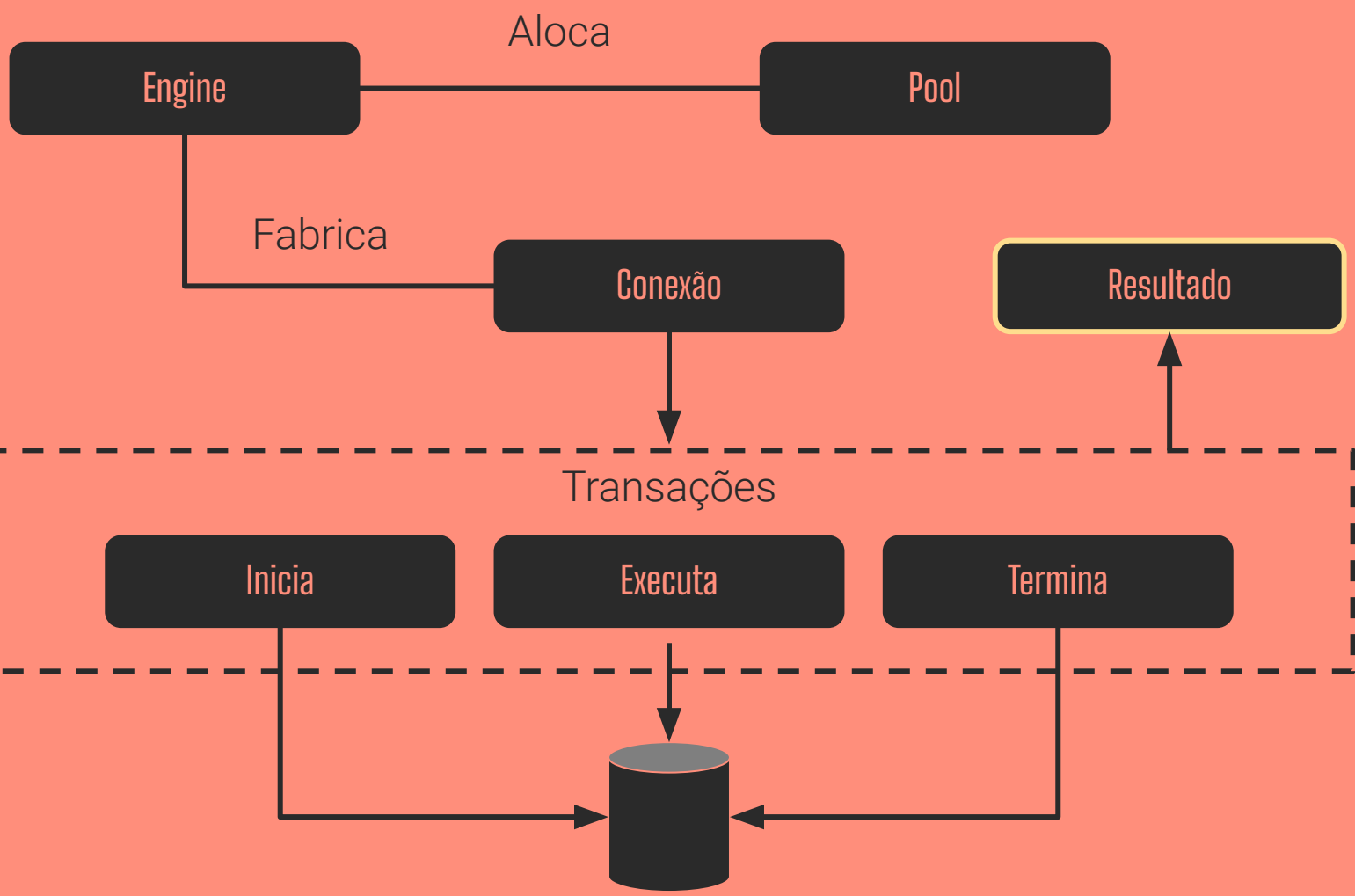
```
con_1 = engine.connect()
print(engine.pool.status())
# Pool size: 5  Connections in pool: 0 Current Overflow: -4 Current Checked out connections: 1

con_2 = engine.connect()
print(engine.pool.status())
# Pool size: 5  Connections in pool: 0 Current Overflow: -3 Current Checked out connections: 2
```

Um entendimento geral



https://docs.sqlalchemy.org/en/20/_images/sqla_arch_small.png

Engine


Aloca
Engine
Pool
Fabrica
Conexão
Resultado
Transações
Inicia
Executa
Termina

Transação

```python
# exemplo_03.py
from sqlalchemy import create_engine, text

engine = create_engine(<URL>, echo=True)

connection = engine.connect()

sql = text('select id, name, comment from comments')

result = connection.execute(sql)

connection.close()
```

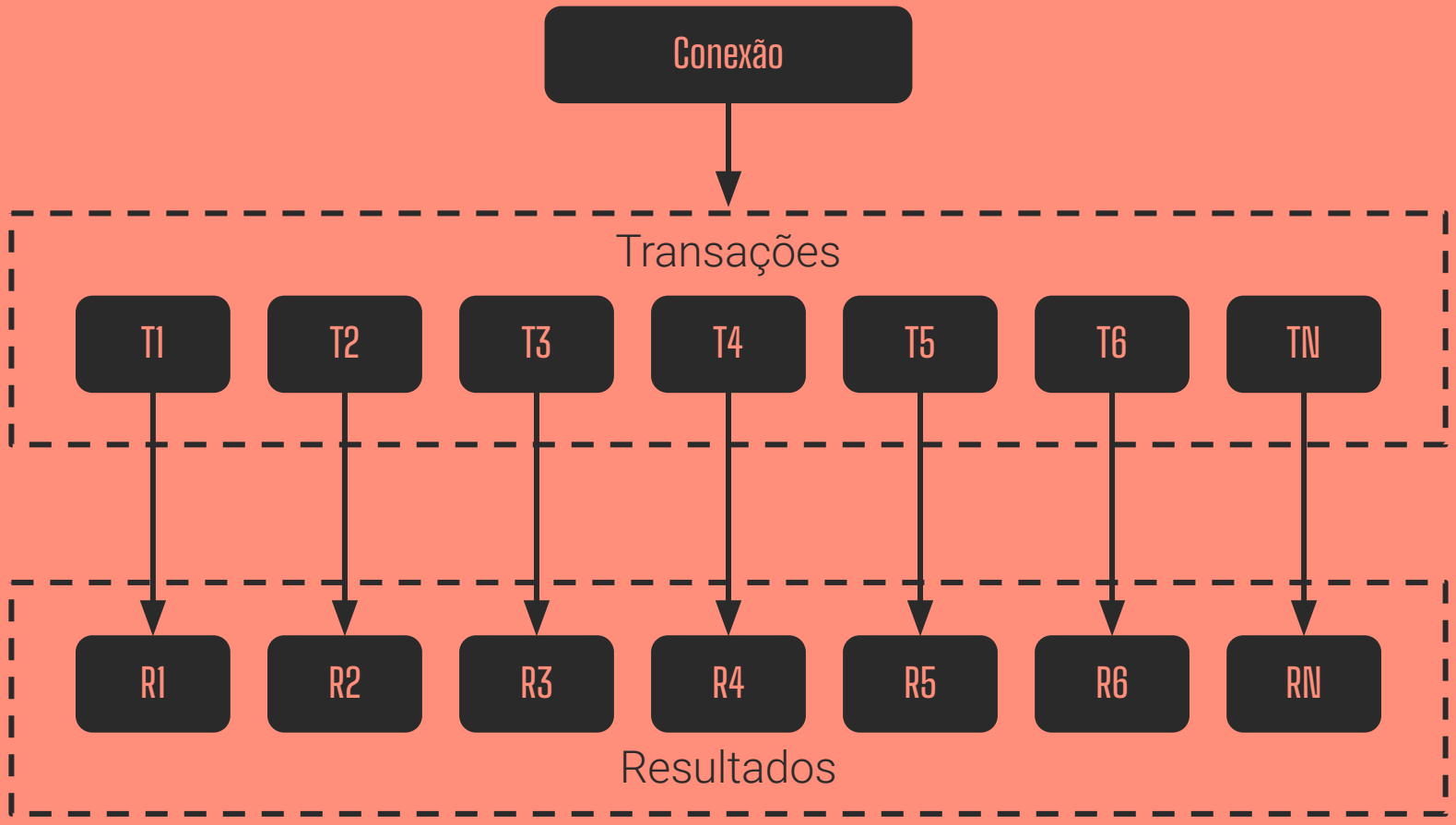
Transação com gerenciador de contexto

```python
# exemplo_04.py
from sqlalchemy import create_engine, text

engine = create_engine(<URL>, echo=True)

with engine.connect() as connection:
    sql = text('select id, name, comment from comments')
    result = connection.execute(sql)
```

Caso queira fazer isso com async

```python
# exemplo_05.py
from asyncio import run
from sqlalchemy import create_engine, text
from sqlalchemy.ext.asyncio import create_async_engine

engine = create_async_engine(<URL>)

async def main():
    async with engine.connect() as connection:
        sql = text('select id, name, comment from comments')
        result = await connection.execute(sql)

run(main())
```

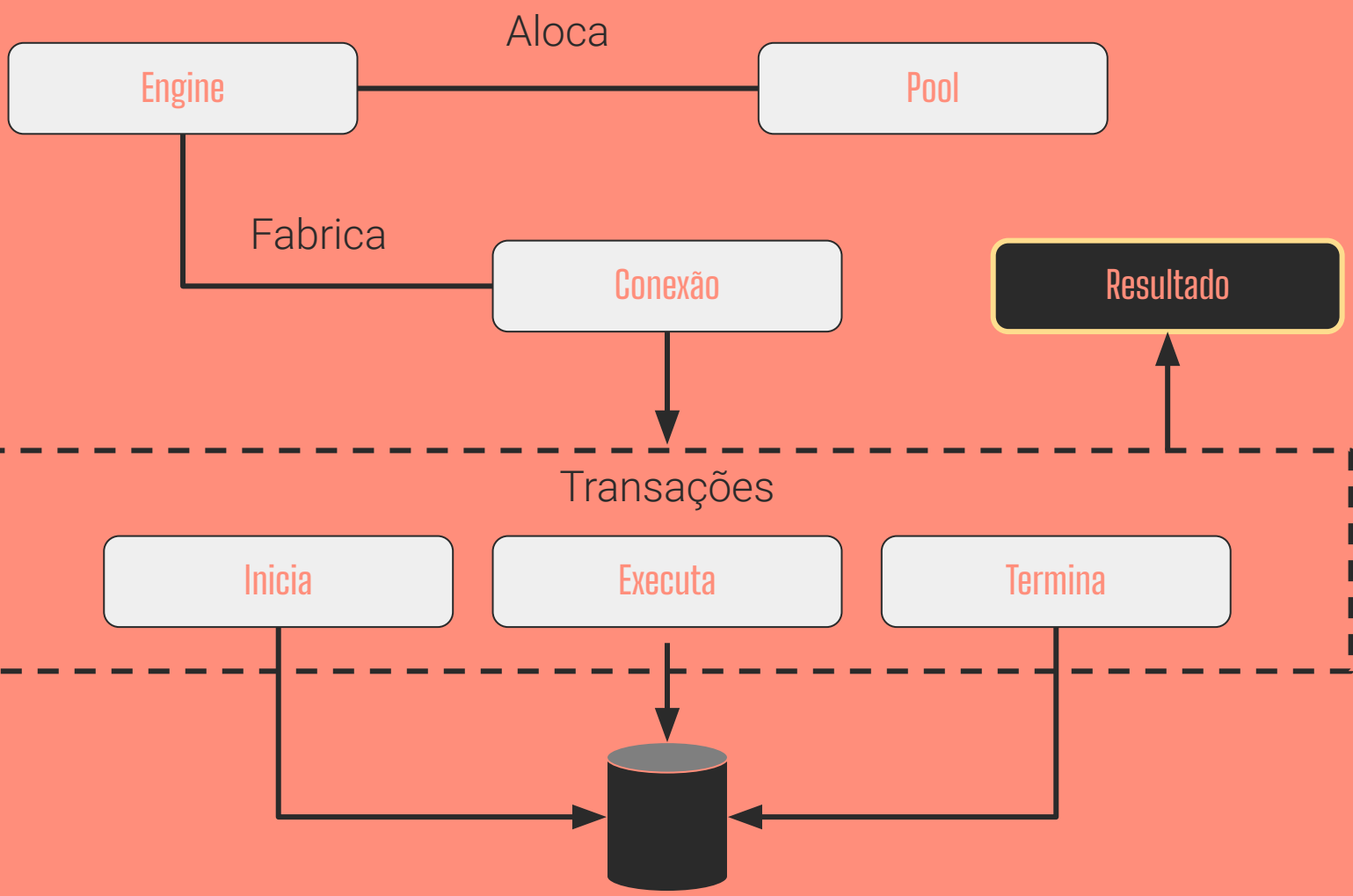
N transactions



Begin

```python
# exemplo_06.py
from sqlalchemy import create_engine, text

engine = create_engine(<URL>)
query = 'select id, name, comment from comments limit 10 offset {of}'

with engine.connect() as connection:
    with connection.begin():
        sql = text(query.format(of=0))
        result1 = connection.execute(sql)
    with connection.begin():
        sql = text(query.format(of=1))
        result2 = connection.execute(sql)
    with connection.begin():
        sql = text(query.format(of=2))
        result3 = connection.execute(sql)
```

Result


Aloca
Engine
Pool
Fabrica
Conexão
Resultado
Transações
Inicia
Executa
Termina

Result

O resultado obtido no execute é um objeto especial chamado **Result**. Ele implementa diversos métodos, além de ser um iterável.

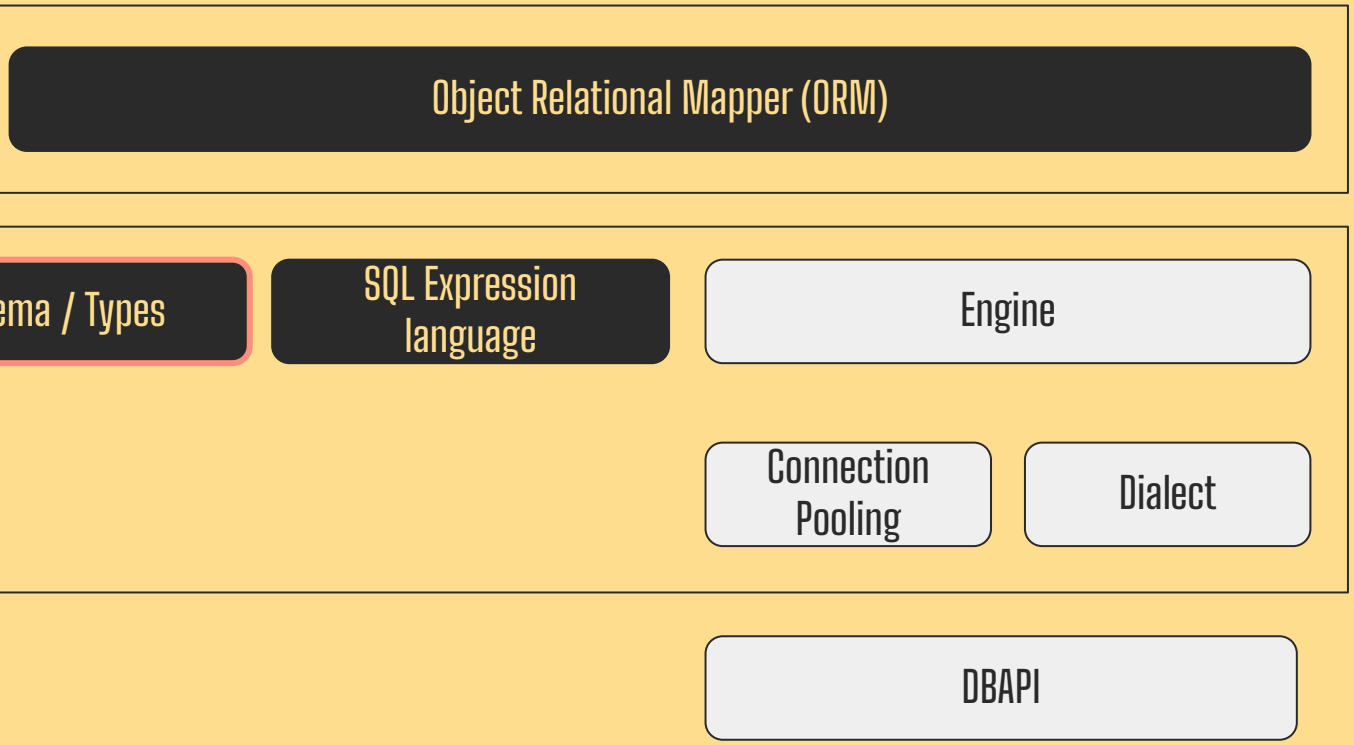
Alguns dos métodos que gostaria de mostrar do result:

- .**fetchone**(): pega o primeiro
- .**fetchmany**(3) / .**partitions**(3): pega alguns valores
- .**fetchall**() / .**all**(): pega todos os valores
- .**first**(): Pega 1, mas não dá erro se não conseguir

Result

```python
# exemplo_07.py
with engine.connect() as con:
    sql = text('select * from comments limit 10 offset 10')
    result = con.execute(sql)

primeito_valor = result.fetchone()
todos_os_valores = result.fetchall()

print(primeito_valor)
print(todos_os_valores)
```

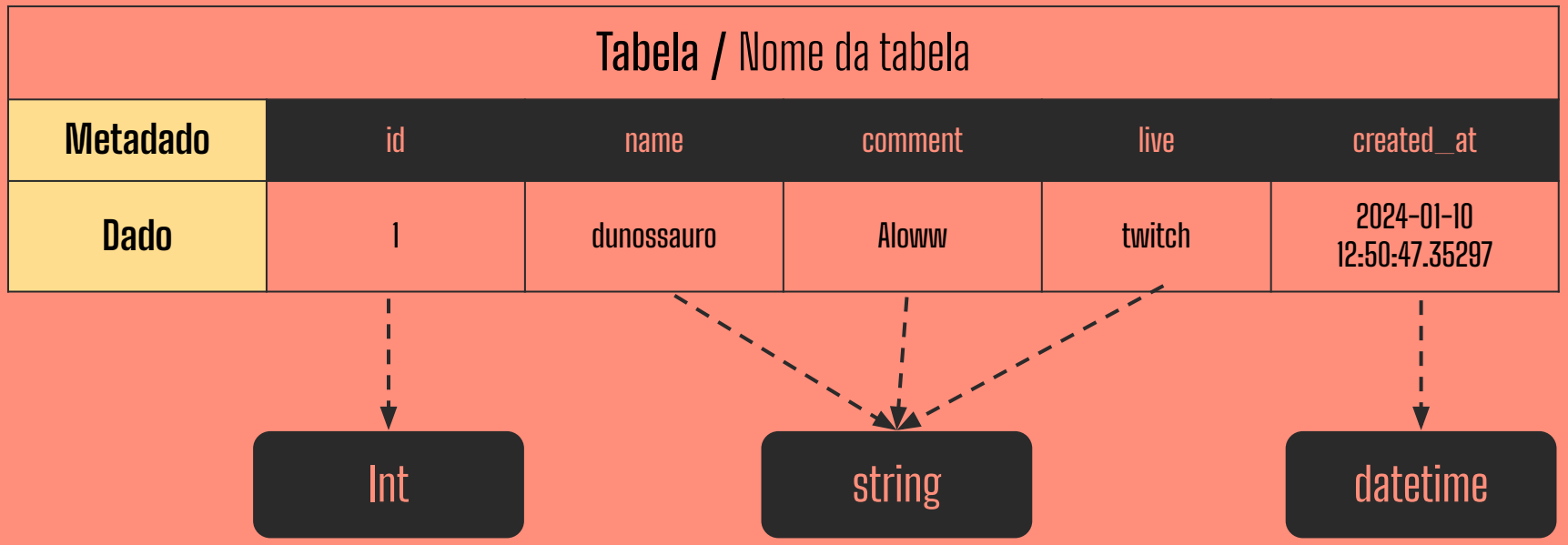
Um entendimento geral



https://docs.sqlalchemy.org/en/20/_images/sqla_arch_small.png

Schemas / Types

Os metadados das tabelas podem ser descritos por Schemas e seus determinados Tipos.



Schemas / Types

```python
# exemplo_08.py
import sqlalchemy as sa

metadata = sa.MetaData()

t = sa.Table('comments', metadata,
    sa.Column('id', sa.Integer(), nullable=False),
    sa.Column('name', sa.String(), nullable=False),
    sa.Column('comment', sa.String(), nullable=False),
    sa.Column('live', sa.String(), nullable=False),
    sa.Column('created_at', sa.DateTime(), nullable=True),
    sa.PrimaryKeyConstraint('id'),
)
```

Schemas / Types

```python
# exemplo_08.py
import sqlalchemy as sa

metadata = sa.MetaData()

t = sa.Table('comments', metadata,
    sa.Column('id', sa.Integer(), nullable=False),
    sa.Column('name', sa.String(), nullable=False),
    sa.Column('comment', sa.String(), nullable=False),
    sa.Column('live', sa.String(), nullable=False),
    sa.Column('created_at', sa.DateTime(), nullable=True),
    sa.PrimaryKeyConstraint('id'),
)
```

```python
# exemplo_08.py
engine = sa.create_engine('sqlite://')

metadata.create_all(engine)
```

Schemas / Types

```python
# exemplo_08.py
engine = sa.create_engine('sqlite://')

metadata.create_all(engine)
```

```python
# exemplo_08.py
with engine.connect() as con:
    with con.begin():
        con.execute(
            sa.text(
                f"""
                insert into comments (name, comment, live, created_at)
                values ('dunossauro', 'alow', 'youtube', '2024-01-10 12:50:47.35297');
                """
            )
        )

    with con.begin():
        result = con.execute(sa.text('select * from comments'))
        print(result.fetchall())
```

```
    )
```

Inspeção de metadados

```python
# exemplo_09.py
from sqlalchemy import create_engine, inspect

engine = create_engine('<URL>')

inspected = inspect(engine)

tables = inspect.get_table_names()
columns = inspected.get_columns('comments')
```

Reflection

As funções de inspeções são agregadas a construção de schemas, para evitar a criação dos metadados em um banco que já existe:

```python
# exemplo_10.py
from sqlalchemy import create_engine, Table, MetaData

engine = create_engine(<URL>)
metadata = MetaData()

comments = Table('comments', metadata, autoload_with=engine)

print(comments.c)
print(comments.columns)
print(comments.c.id)
```

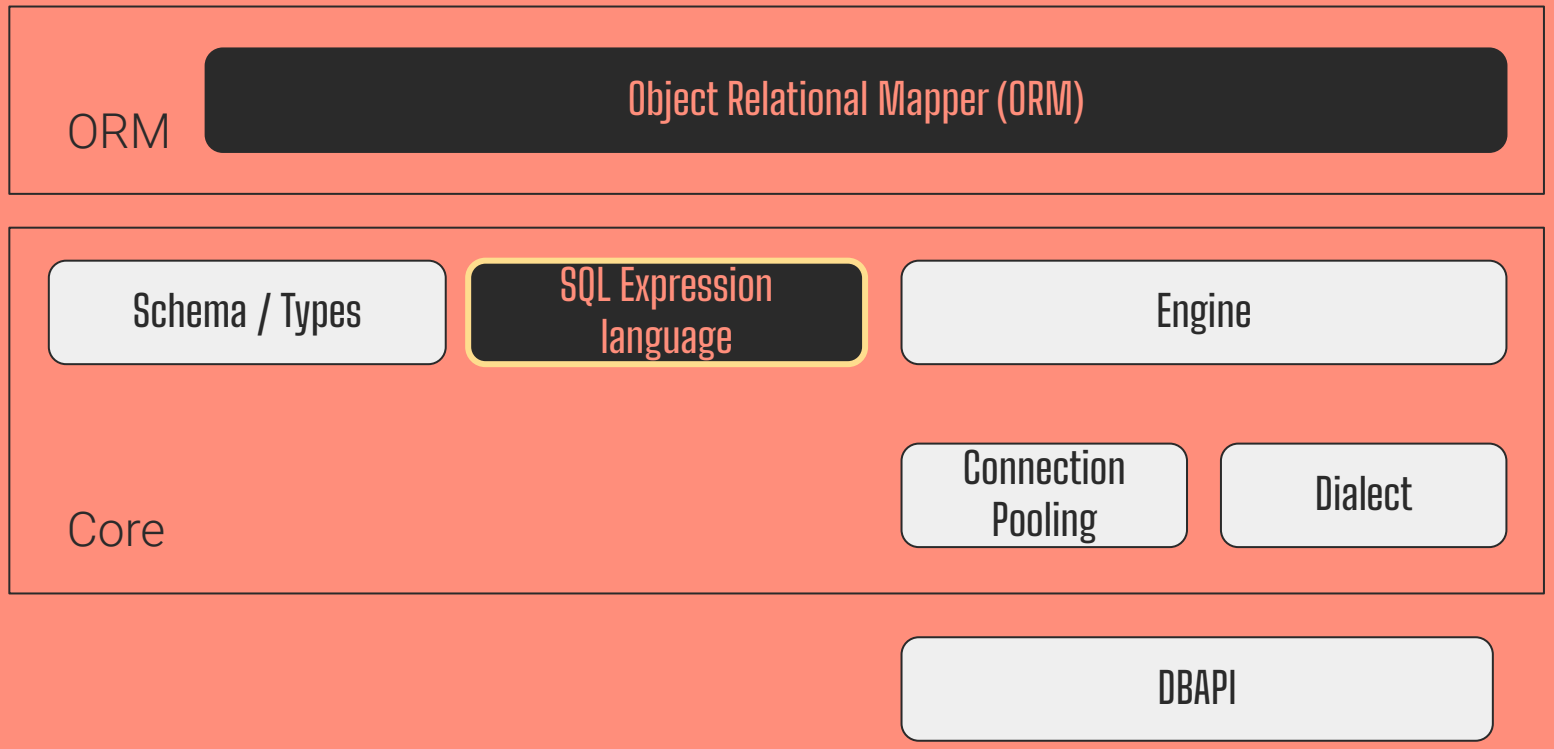
Reflection

As funções de inspeções são agre
evitar a criação dos metadados e

```
# exemplo_10.py
from sqlalchemy import crea

engine = create_engine(<URL
metadata = MetaData()

comments = Table('comments',
                 load_with=engine)

print(comments.c)
print(comments.columns)
print(comments.c.id)
```

```
ReadOnlyColumnCollection(
    comments.id,
    comments.name,
    comments.comment,
    comments.live,
    comments.created_at
)
```

Um entendimento geral



https://docs.sqlalchemy.org/en/20/_images/sqla_arch_small.png

SQL Expression Language

Até esse momento, todas as operações que fizemos com o banco, fizemos com a função **text**() e escrevemos SQL na mão.

O Core tem um grupo de funções e objetos que podem nos ajudar a montar SQL:

- **DQL**: Data Query Language
- **DML**: Data Manipulation Language

Usado em conjunto com os Schemas.

DQL

Uma das partes mais importantes dentro dos bancos de dados é a busca pelos dados. O que chamamos de Query. O Alchemy tem um sistema completo e extenso sobre a criação de queries.

Começaremos belo básico, a função **select**()

```
stmt = select(comments)
print(stmt)
# SELECT comments.id, comments.name, comments.comment, comments.live, comments.created_at
# FROM comments
```

CompoundSelect

O resultado do select é um **builder**, com ele você pode encadear comandos e fazer uma busca mais complexa

```
stmt = (
    select(comments.c.name, comments.c.comment)
    .where(comments.c.name == 'Eduardo Mendes')
    .limit(3)
    .offset(0)
    .order_by(comments.c.id)
)
```

CompoundSelect

O resultado do select é um **builder**, com ele você pode encadear comandos e fazer uma busca mais complexa

```
stmt = (
    select(comments.c.name, comments.c.comment)
    .where(comments.c.name == 'Eduardo Mendes')
    .limit(3)
    .offset(0)
    .order_by(co
)
```

```
"""
SELECT comments.name, comments.comment AS name__1
FROM comments
WHERE comments.name = :name_1 ORDER BY comments.id
LIMIT :param_1
OFFSET :param_2
"""
```

Conectivos lógicos

```
stmt = (
    select(comments.c.name, comments.c.comment)
    .where(
        (comments.c.name == 'Eduardo Mendes')
        | (comments.c.name == 'dunossauro')
        & (comments.c.live == 'twitch')
    )
)
print(stmt)
"""
SELECT comments.name, comments.comment
FROM comments
WHERE comments.name = :name_1 OR comments.name = :name_2
  AND comments.live = :live_1
"""
```

Existe MUITOS mais coisas

Das quais não vou me aprofundar aqui.

https://docs.sqlalchemy.org/en/20/core/sqlelement.html

DML

Quando precisamos manipular dados no SQL, usamos algumas das instruções listadas:

- delete: remove registros
- insert: insere registros
- update: atualiza registos

O Alchemy tem uma função específica para cada uma dessas operações.

Insert

```
stmt = insert(comments).values(
    name='dunossauro',
    comment='LLL',
    live='youtube',
    created_at=datetime.now(),
)
# INSERT INTO comments (name, comment, live, created_at)
# VALUES (:name, :comment, :live, :created_at)
```

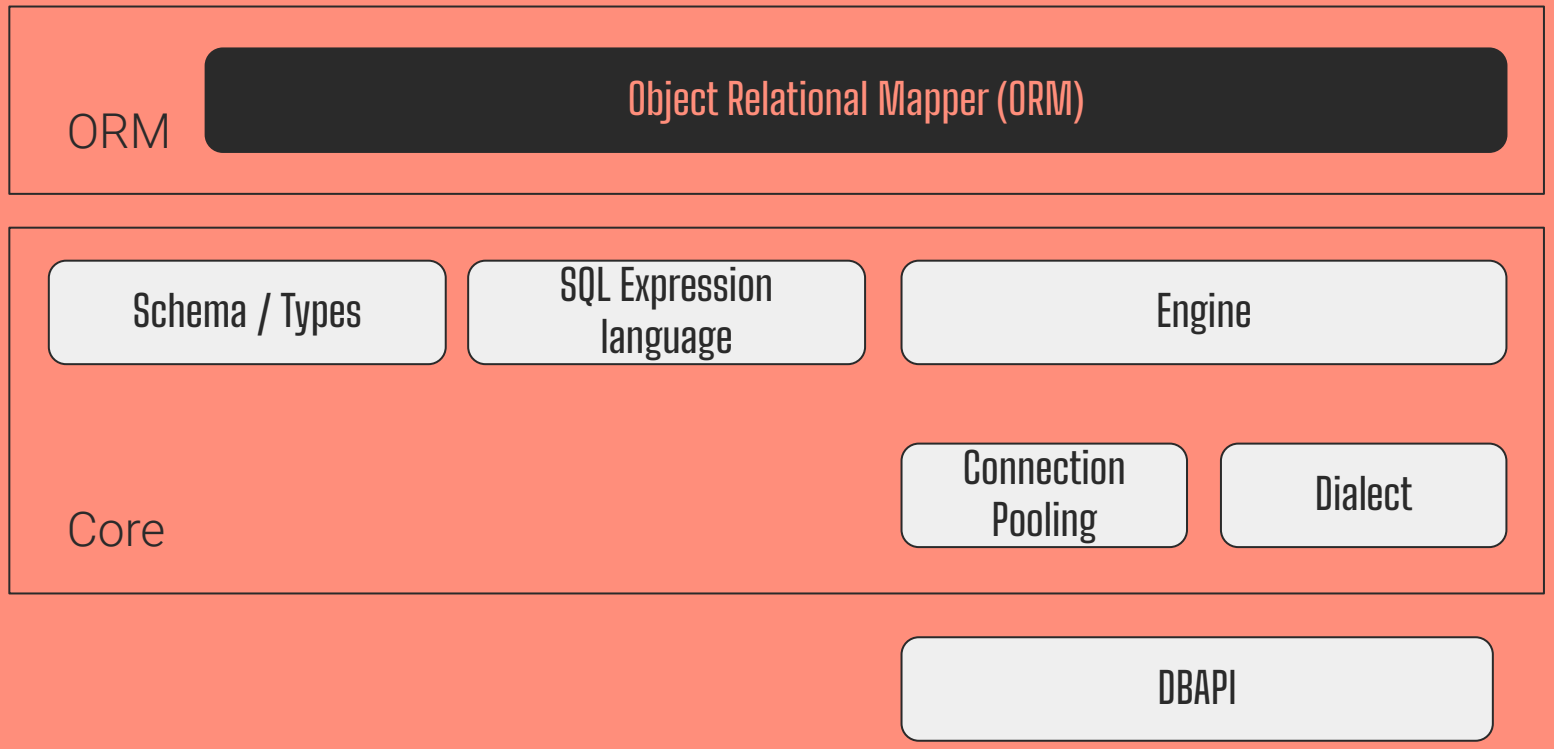
Update

```
stmt = (
    update(comments)
    .where(
        comments.c.name == 'dunossauro',
        comments.c.comment == 'LLL',
        comments.c.live == 'youtube',
    )
    .values(comment='Pei')
)
# UPDATE comments SET comment=:comment
# WHERE comments.name = :name_1
#   AND comments.comment = :comment_1 AND comments.live = :live_1
```

Delete

```
stmt = delete(comments).where(
    comments.c.name == 'dunossauro',
    comments.c.live == 'youtube',
    comments.c.comment == 'Pei',
)
# DELETE FROM comments
# WHERE comments.name = :name_1
#   AND comments.live = :live_1 AND comments.comment = :comment_1
```

Um entendimento geral



https://docs.sqlalchemy.org/en/20/_images/sqla_arch_small.png

Object Relational Mapper

ORM

Object-Relational Mapper

Antes de entrarmos a fundo nisso, acho que cabe uma explicação do termo ORM:

- **Object**: quer dizer um **objeto python**, como uma classe que construímos
- **Relational**: Relacional é em relação aos bancos relacionais.
- **Mapper**: quer dizer que é feito um **mapeamento entre os metadados** das tabelas em uma classe e cada row é relacionada a uma instância

O/R Mapper é uma forma possível de escrever também

Objetos?

A parte interessante de um ORM é poder definir os schemas usando objetos do python

```python
from sqlalchemy import Column, DateTime, Integer, String, func
from sqlalchemy.orm import DeclarativeBase

class Base(DeclarativeBase):
    ...

class Comment(Base):
    __tablename__ = 'comments'

    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    comment = Column(String, nullable=False)
    live = Column(String, nullable=False)
    created_at = Column(DateTime, server_default=func.now())
```

Typing?

```python
from datetime import datetime
from sqlalchemy import func
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column

class Base(DeclarativeBase):
    ...
class Comment(Base):
    __tablename__ = 'comments'

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str]
    comment: Mapped[str]
    live: Mapped[str]
    created_at: Mapped[datetime] = mapped_column(server_default=func.now())
```

Dataclasses! (minha forma preferida)

```python
from datetime import datetime

from sqlalchemy import func
from sqlalchemy.orm import Mapped, mapped_column, registry

reg = registry()


@reg.mapped_as_dataclass
class Comment:
    __tablename__ = 'comments'

    id: Mapped[int] = mapped_column(init=False, primary_key=True)
    name: Mapped[str]
    comment: Mapped[str]
    live: Mapped[str]
    created_at: Mapped[datetime] = mapped_column(
        init=False, server_default=func.now()
    )
```

Imperativo

Os objetos que definimos aqui são **declarativos**, declaramos como ele é. Podemos fazer de forma imperativa.

```python
from sqlalchemy import Column, DateTime, Integer, PrimaryKeyConstraint, String, Table
from sqlalchemy.orm import registry

mapper_registry = registry()

t = Table('comments', mapper_registry.metadata,
    Column('id', Integer(), nullable=False),
    Column('name', String(), nullable=False),
    Column('comment', String(), nullable=False),
    Column('live', String(), nullable=False),
    Column('created_at', DateTime(), nullable=True),
    PrimaryKeyConstraint('id'),
)

class Comment:
    pass

mapper_registry.map_imperatively(Comment, t)
```

Automap

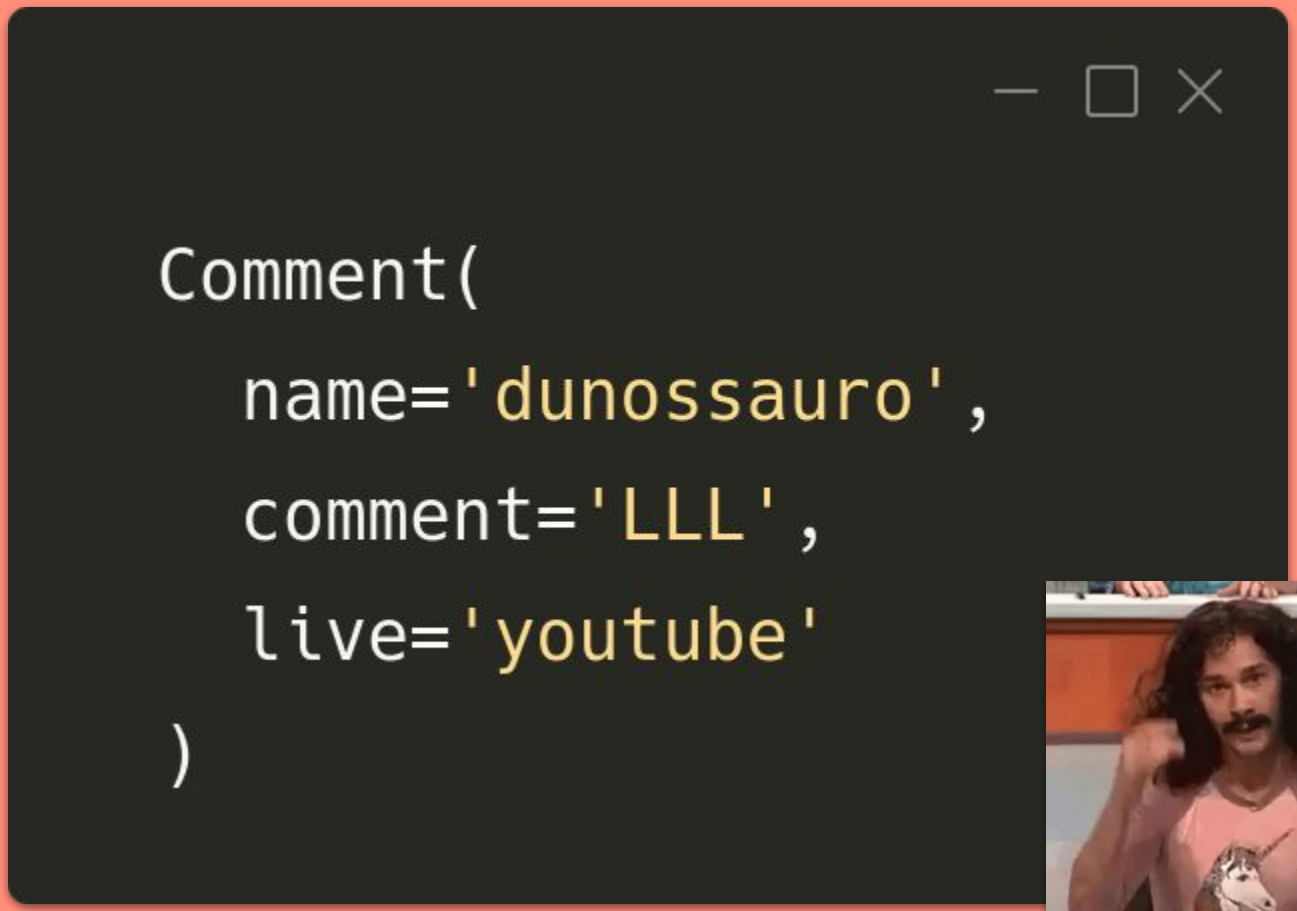
Não vamos falar sobre eventos hoje, mas é possível

https://docs.sqlalchemy.org/en/20/orm/declarative_tables.html#automating-column-naming-schemes-from-reflected-tables

Trabalhando com o objeto

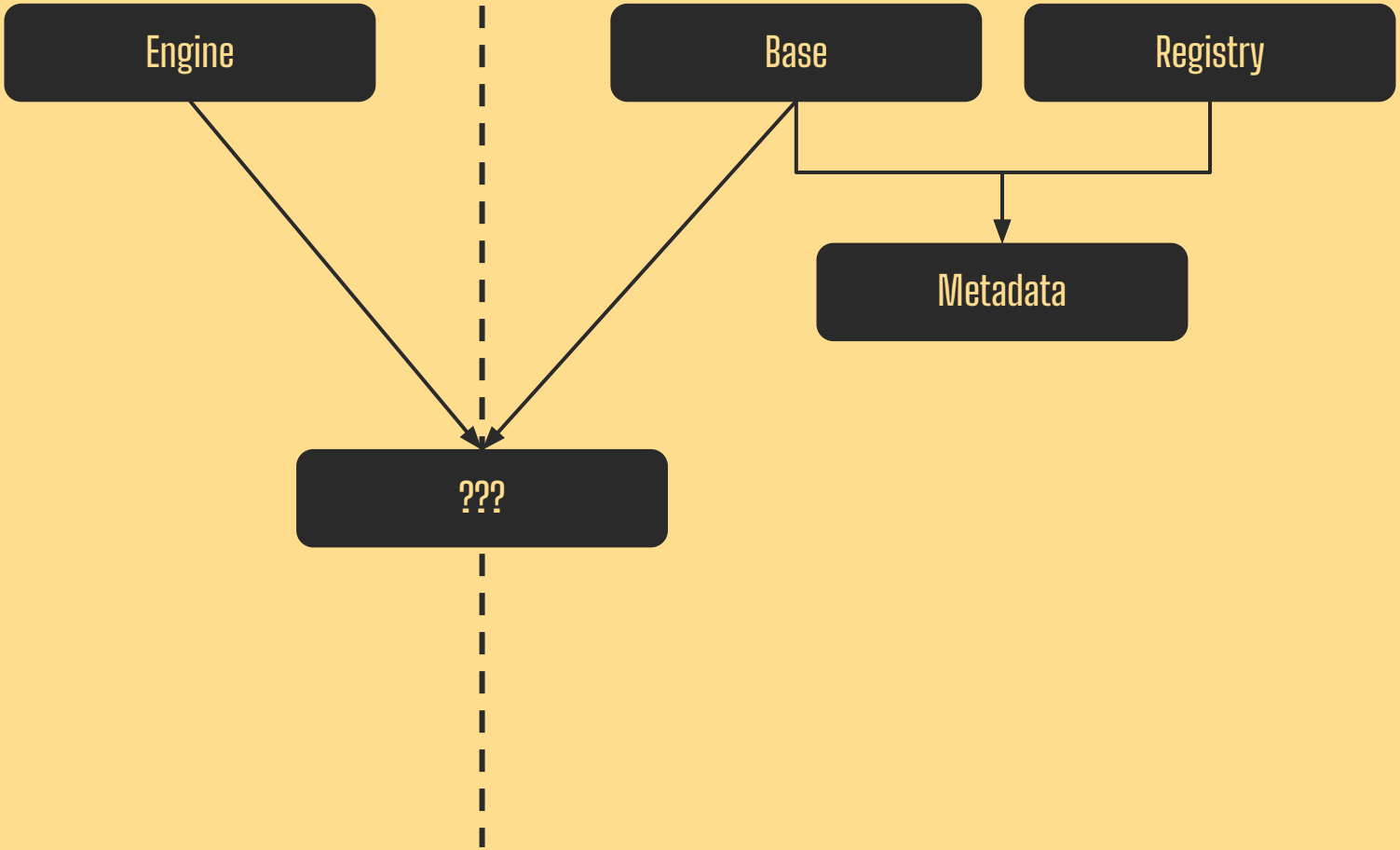
```
Comment(
    name='dunossauro',
    comment='LLL',
    live='youtube'
)
```

Trabalhando com o objeto

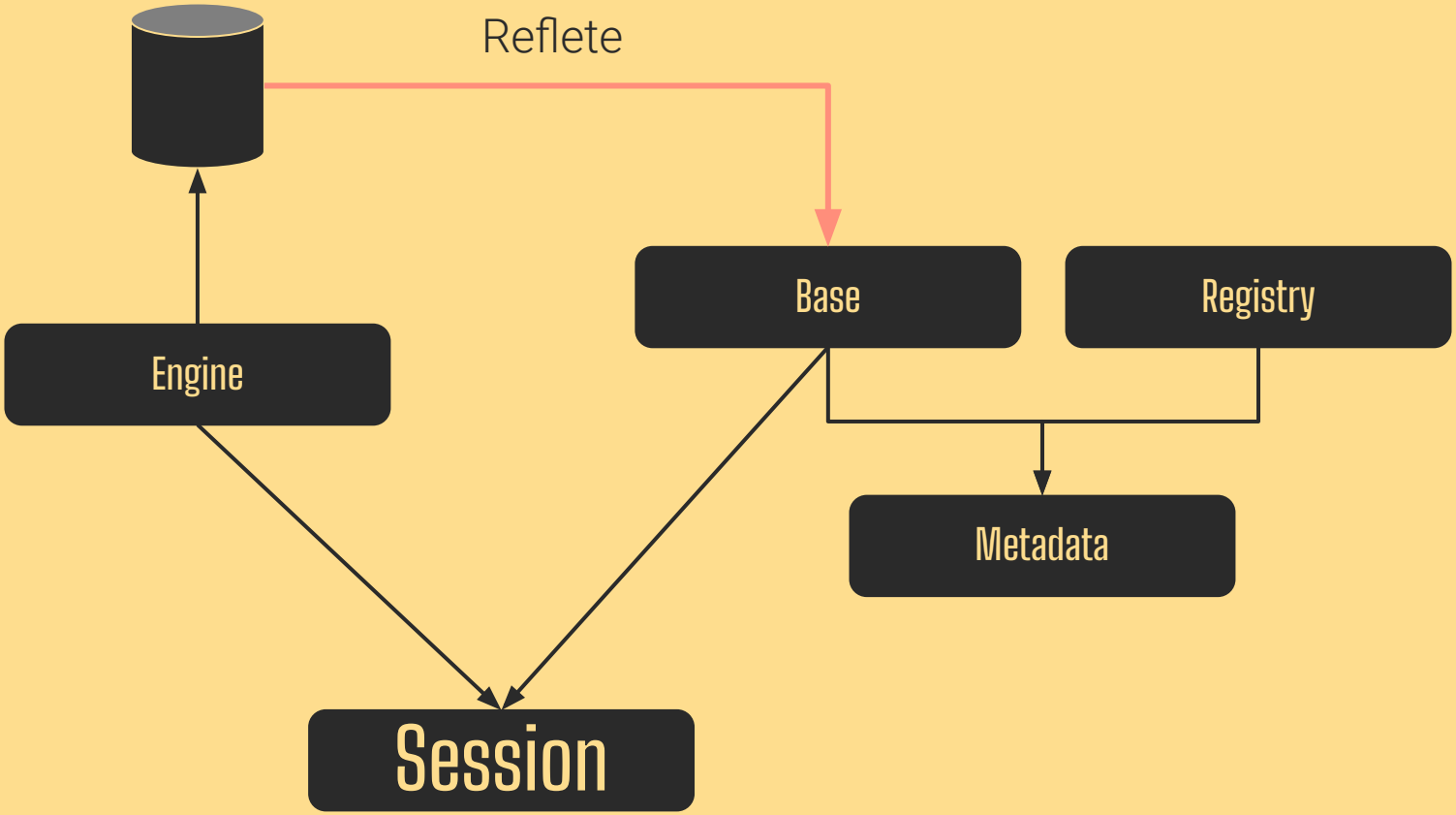


```
Comment(
    name='dunossauro',
    comment='LLL',
    live='youtube'
)
```

Engine / Metadata



Session



Session

A session faz o papel da "connection" do core, mas retorna objetos do ORM na query.

```python
from sqlalchemy.orm import Session

with Session(engine) as s:
    result = s.scalars(select(<Objeto Tabela>))
    print(result.fetchmany(3))
```

Updates

```
with Session(engine) as s:
    result = s.scalar(select(Comment).where(Comment.id == 1))
    result.comment = 'PEI'
    s.commit()
```

Delete

```
with Session(engine) as s:
    result = s.scalar(select(Comment).where(Comment.id == 1))
    s.delete(result)
    s.commit()
```

SIIIMMM!

Tem

+?

Coisas que não vimos

Mais que são legais de mais:

- eventos
- Cache com dogpile
- Migrações com Alembic
- ...

apoia.se/livedepython

pix.dunossauro@gmail.com

patreon.com/dunossauro

Ajude o projeto <3