

جزوه کامل برای یادگیری SQLAlchemy

مقدمه

در این جزوه، با تکیه بر مستندات رسمی SQLAlchemy و همچنین کمک گرفتن از هوش مصنوعی‌های مختلف، تلاش می‌کنم کاربردها و جزئیات مهم این کتابخانه را بررسی کنم. هدف این است که مفاهیم را نه تنها در بستر پایتون، بلکه در ارتباط با زبان‌ها و فریم‌ورک‌های مختلف هم مرور کنیم تا یادگیری جنبه‌ی کاربردی‌تر و گسترده‌تری داشته باشد.

فهرست مطالب

Basics and prerequisites

Introduce SQLAlchemy •

What is SQLAlchemy ○

Benefits and uses ○

Installation and setup ○

Core vs ORM ○

Connect to database •

Connection و Engine ○

Connection Strings ○

Connection Pooling ○

Database Events ○

SQLAlchemy Core

MetaData , Table Definition •

- Table objects ○

- Column types ○

- Constraints ○

- Indexes ○

- Schema operations ○

SQL Expression Language •

- Select statements ○

- Insert, Update, Delete ○

- Joins ○

- Operators , Functions ○

- Subqueries ○

Executing Statements •

- Connection execution ○

- Result objects ○

- Transactions ○

SQLAlchemy ORM

Declarative Base •

- Model definition ○

- Table mapping ○

- Primary keys ○

- Column options ○

Sessions •

- Session lifecycle ○

- configuring sessions , Creating ○

- Session states ○

- rollback , Committing ○

Basic Queries •

- Query objects ○

- Filtering ○

- Ordering ○

- Limiting ○

Advanced ORM , Relationships

Relationships •

One-to-Many ○

Many-to-One ○

One-to-One ○

Many-to-Many ○

Back references ○

Lazy Loading vs Eager Loading •

Lazy loading patterns ○

Eager loading (joinedload, selectinload) ○

N+1 problem ○

Loading strategies ○

Advanced Querying •

Complex joins ○

Subqueries ○

Union operations ○

Window functions ○

Raw SQL integration ○

Advanced Topics

Session Management Patterns •

Session per request ○

Contextual sessions ○

Thread-local sessions ○

Session events ○

Advanced Relationships •

Self-referential relationships ○

Polymorphic relationships ○

Hybrid properties ○

Extensions , Customization •

Custom types ○

Validators ○

listeners , Events ○

Mixins ○

Custom loading techniques ○

Optimization 9 Performance

Performance Tuning •

Query optimization ○

Index strategies ○

Connection pooling ○

Bulk operations ○

Caching •

Query result caching ○

Second-level cache ○

Dogpile.cache integration ○

Advanced Features

Alembic 4 Migrations •

Migration basics ○

Auto-generating migrations ○

Manual migrations ○

merging 9 Branching ○

Testing •

Testing patterns ○

Fixtures ○

Mock strategies ○

Database testing ○

Real-world Applications

Design Patterns •

Repository pattern ○

Unit of Work ○

Data Mapper ○

Active Record considerations ○

Integration •

Web framework integration (Flask, FastAPI) ○

Async SQLAlchemy ○

Multi-database setups ○

Sharding strategies ○

مبانی و مقدمات

معرفی SQLAlchemy

SQLAlchemy چیست

SQLAlchemy یکی از قدرتمندترین و محبوبترین کتابخانه‌های Python برای کار با پایگاه‌های داده است. این کتابخانه دو رویکرد اصلی ارائه می‌دهد:

- SQL Toolkit ← برای نوشتن و اجرای کوئری‌های SQL با امکانات بیشتر و سطح پایین‌تر.
- ORM (Object Relational Mapper) ← برای کار با پایگاه‌های داده از طریق Python objects به جای نوشتن مستقیم SQL خام.

مزایا و کاربردها

1. انتزاع بالا

```
# خام SQL به جای نوشتن:
cursor.execute("SELECT * FROM users WHERE age > 18")

# می‌توانید بنویسید:
users = session.query(User).filter(User.age > 18).all()
```

2. پشتیبانی از چندین پایگاه داده

- PostgreSQL
- MySQL
- SQLite
- Oracle
- Microsoft SQL Server
- و بسیاری دیگر

3. IntelliSense و Type Safety

- **Type Safety**: وقتی داری کد می‌نویسی، کامپیوتر کمک می‌کند که نوع داده‌ها درست باشه و جلوی اشتباهات رایج رو بگیره
- **IntelliSense**: این قابلیت IDE هاست که وقتی داری کد می‌نویسی، خودکار پیشنهادها، اتوماتیک تکمیل، و مستندات کوتاه نشون بده

4. Migration Management

مدیریت تغییرات دیتابیس وقتی مدل‌ها رو تغییر میدی.

نصب و راه‌اندازی

نصب پایه

برای نصب SQLAlchemy به صورت پایه:

```
pip install sqlalchemy
```

برای نصب با پشتیبانی از `async`:

```
pip install "sqlalchemy[asyncio]"
```

نصب درایورهای پایگاه داده

• PostgreSQL:

```
pip install psycopg2-binary
```

• MySQL:

```
pip install pymysql
```

• Oracle:

```
pip install cx_Oracle
```

نصب برای فریمورک‌های مختلف

• Flask:

```
pip install flask flask-sqlalchemy flask-migrate
```

• FastAPI:

```
pip install fastapi sqlalchemy alembic uvicorn
```

• Django: Django به صورت پیش‌فرض ORM داخلی دارد، اما در صورت نیاز می‌توانید SQLAlchemy را هم استفاده کنید:

```
pip install django sqlalchemy
```

تست نصب

برای اطمینان از نصب صحیح، این کد را اجرا کنید:

```
import sqlalchemy
print(sqlalchemy.__version__) # باید ورژن نصب شده را نمایش دهد
```

Core vs ORM

execute متدی است که یک دستور SQL (مثل select, insert, update, delete) را روی پایگاه داده اجرا می‌کند.

SQLAlchemy دو رویکرد اصلی ارائه می‌دهد:

SQLAlchemy Core (سطح پایین)

Core برای کسانی است که می‌خواهند کنترل بیشتری روی SQL داشته باشند.

```
from sqlalchemy import create_engine, MetaData, Table, Column, Integer, String, select

# تعریف جدول
metadata = MetaData()
users = Table('users', metadata,
              Column('id', Integer, primary_key=True),
              Column('name', String(50)),
              Column('email', String(100))
)

# اتصال و ایجاد جدول
engine = create_engine("sqlite:///example.db")
metadata.create_all(engine)

# کوئری
with engine.connect() as conn:
    # Insert
    conn.execute(users.insert().values(name='احمد', email='ahmad@example.com'))

    # Select
    result = conn.execute(select(users).where(users.c.name == 'احمد'))
    for row in result:
        print(f"ID: {row.id}, Name: {row.name}")
```

مزایای Core:

- سرعت بیشتر
- کنترل کامل روی SQL
- مناسب برای کوئری‌های پیچیده
- حجم حافظه کمتر

SQLAlchemy ORM (سطح بالا)

ORM برای توسعه‌دهندگانی است که می‌خواهند با Python objects کار کنند.

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



```

Base = declarative_base()

# تعریف مدل
class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    email = Column(String(100))

    def __repr__(self):
        return f"<User(name='{self.name}', email='{self.email}')>"

# راه اندازی
engine = create_engine("sqlite:///example.db")
Base.metadata.create_all(engine)
Session = sessionmaker(bind=engine)
session = Session()

# استفاده
# Insert
new_user = User(name='احمد', email='ahmad@example.com')
session.add(new_user)
session.commit()

# Select
users = session.query(User).filter(User.name == 'احمد').all()
for user in users:
    print(user)

```

مزایای ORM:

- کد قابل خوانش تر
- Type safety
- Relationship handling آسان
- کمتر SQL نوشتن

اتصال به پایگاه داده

Connection و Engine

Engine چیست؟ Engine در SQLAlchemy مانند مدیر ارتباط با پایگاه داده عمل می‌کند.

- یک بار ساخته می‌شود و در تمام برنامه استفاده می‌شود.
- مسئول مدیریت ارتباطات و اجرای دستورات SQL است.

ساخت Engine

```
from sqlalchemy import create_engine

# SQLite
engine = create_engine("sqlite:///myapp.db")

# PostgreSQL
engine = create_engine("postgresql://user:password@localhost/dbname")
```

استفاده از Connection

روش 1: با Context Manager (بهترین روش)

```
with engine.connect() as conn:
    result = conn.execute("SELECT 1")
    print(result.fetchone())
```

✓ مزیت: بعد از پایان بلوک، ارتباط به صورت خودکار بسته می‌شود.

روش 2: دستی (باید خودتان close کنید)

```
conn = engine.connect()
result = conn.execute("SELECT 1")
print(result.fetchone())
conn.close()  # حتماً ارتباط را ببندید
```

مثال در Flask

```
from flask import Flask
from sqlalchemy import create_engine

app = Flask(__name__)
engine = create_engine("sqlite:///app.db")

@app.route("/test")
```

```
def test_db():
    with engine.connect() as conn:
        result = conn.execute("SELECT COUNT(*) FROM users")
        return f"تعداد کاربران: {result.scalar()}"
```

Connection Strings

Connection String آدرس پایگاه داده شماست:

انواع مختلف:

```
# SQLite
"sqlite:///path/to/database.db"
"sqlite:///C:/path/to/database.db" # Windows
"sqlite:///memory:" # حافظه در حافظه

# PostgreSQL
"postgresql://username:password@localhost:5432/dbname"
"postgresql+psycopg2://user:pass@localhost/dbname"

# MySQL
"mysql://username:password@localhost:3306/dbname"
"mysql+pymysql://user:pass@localhost/dbname"

# SQL Server
"mssql+pyodbc://user:pass@server/dbname?driver=ODBC+Driver+17+for+SQL+Server"
```

Connection Pooling

Pool یعنی استخر اتصال - تعدادی connection آماده نگه می‌دارد.

```
# تنظیمات Pool
engine = create_engine(
    "postgresql://user:pass@localhost/db",
    pool_size=10, # connection تا 10
    max_overflow=20, # حداکثر 20 تا اضافی
    pool_timeout=30, # ثانیه صبر 30
    pool_recycle=3600 # reset کن ها را connection هر ساعت
)

# چک کردن وضعیت Pool
print(f"Pool size: {engine.pool.size()}")
print(f"Active connections: {engine.pool.checked_in()}")
```

Database Events

در SQLAlchemy، Event ها مکانیزمی هستند که به توسعه‌دهنده اجازه می‌دهد به رویدادها و مراحل مختلف عملیات دیتابیس گوش دهد و به آن‌ها واکنش نشان دهد. این قابلیت امکان مانیتورینگ، لاگینگ و اعمال تغییرات سفارشی در سطح اتصال، کوئری یا تراکنش را فراهم می‌کند.

```
# قبل از اتصال
@event.listens_for(engine, "connect")
def set_sqlite_pragma(dbapi_connection, connection_record):
    if "sqlite" in str(engine.url):
        cursor = dbapi_connection.cursor()
        cursor.execute("PRAGMA foreign_keys=ON")
        cursor.close()

# شدن کوئری execute قبل از
@event.listens_for(engine, "before_cursor_execute")
def receive_before_cursor_execute(conn, cursor, statement, parameters, context, execute)
    print(f"اجرا می‌شود: {statement}")
```

SQLAlchemy Core

MetaData و Table Definition

- **MetaData** یک شیء مرکزی است که تمام اطلاعات مربوط به جداول، ستون‌ها و ارتباطات پایگاه داده را نگهداری و مدیریت می‌کند.
- یعنی هر جدول (Table) که تعریف می‌کنی معمولاً به یک **MetaData** وصل می‌شود تا بعداً بتوان همه آن‌ها را با هم ایجاد، حذف یا مدیریت کرد.

Table objects

Table در SQLAlchemy مثل نقشه جدول شماس است.

```
from sqlalchemy import MetaData, Table, Column, Integer, String, DateTime
from datetime import datetime

metadata = MetaData()

# تعریف جدول users
users_table = Table('users', metadata,
```

```

Column('id', Integer, primary_key=True),
Column('name', String(50), nullable=False),
Column('email', String(100), unique=True),
Column('created_at', DateTime, default=datetime.utcnow)
)

# ایجاد جدول
engine = create_engine("sqlite:///example.db")
metadata.create_all(engine)

```

Column types

انواع اصلی Column:

```

from sqlalchemy import Integer, String, Text, Boolean, DateTime, Float, Numeric

users = Table('users', metadata,
    # اعداد
    Column('id', Integer),
    Column('age', Integer),
    Column('salary', Float),
    Column('balance', Numeric(10, 2)), # رقم، 2 اعشار 10
    # متن
    Column('name', String(50)),        # حداکثر 50 کاراکتر
    Column('bio', Text),               # متن طولانی
    # تاریخ و زمان
    Column('created_at', DateTime),
    Column('birth_date', DateTime),
    # True/False
    Column('is_active', Boolean, default=True)
)

```

در FastAPI/Flask:

```

# Flask-SQLAlchemy
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), nullable=False)
    email = db.Column(db.String(100), unique=True)
    is_active = db.Column(db.Boolean, default=True)

```

```
# FastAPI
from sqlalchemy import Column, Integer, String, Boolean
class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    is_active = Column(Boolean, default=True)
```

Constraints

انواع Constraint:

```
from sqlalchemy import ForeignKey, CheckConstraint, UniqueConstraint

users = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String(50), nullable=False),
    Column('email', String(100), unique=True),
    Column('age', Integer),

    # Check constraint
    CheckConstraint('age >= 18', name='age_check'),

    # Unique constraint
    UniqueConstraint('email', 'name', name='unique_email_name')
)

# Foreign Key با orders جدول
orders = Table('orders', metadata,
    Column('id', Integer, primary_key=True),
    Column('user_id', Integer, ForeignKey('users.id')),
    Column('total', Float)
)
```

Indexes

Index برای سرعت بخشیدن جستجو:

```
from sqlalchemy import Index

users = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('email', String(100)),
    Column('city', String(50)),
```

```

Column('age', Integer),

# Index ساده
Index('idx_email', 'email'),

# Composite Index
Index('idx_city_age', 'city', 'age'),

# Unique Index
Index('idx_email_unique', 'email', unique=True)
)

```

Schema operations

ایجاد و حذف:

```

# ایجاد همه جداول
metadata.create_all(engine)

# ایجاد فقط یک جدول
users_table.create(engine)

# حذف همه جداول
metadata.drop_all(engine)

# حذف یک جدول
users_table.drop(engine)

```

SQL Expression Language

فرض می‌کنیم یک جدول به اسم users_table داریم با ستون‌های: id, name, email, age, is_active و جدول orders_table با ستون‌های: id, user_id, total

Select statements

دستورات SELECT در SQLAlchemy برای انتخاب و بازیابی داده‌ها از جداول استفاده می‌شوند.

انتخاب همه ردیف‌ها

```

from sqlalchemy import select

```

```
stmt = select(users_table) # انتخاب همه ردیف ها
with engine.connect() as conn:
    result = conn.execute(stmt)
    for row in result:
        print(f"نام: {row.name}، ایمیل: {row.email}")
```

انتخاب ستون خاص

```
stmt = select(users_table.c.name, users_table.c.email)
# یعنی ستون c
```

با شرط ساده

```
stmt = select(users_table).where(users_table.c.age > 25)
```

چند شرط با AND

```
stmt = select(users_table).where(
    (users_table.c.age > 18) & (users_table.c.is_active == True)
)
```

شرط LIKE

```
stmt = select(users_table).where(users_table.c.name.like('%احمد%'))
```

Insert, Update, Delete

Insert – اضافه کردن داده

```
# یکی Insert
stmt = users_table.insert().values(name='علی', email='ali@test.com')
with engine.connect() as conn:
    result = conn.execute(stmt)
    print(f"ID جدید: {result.inserted_primary_key}")

# چندتایی Insert
stmt = users_table.insert()
with engine.connect() as conn:
    conn.execute(stmt, [
```



```
{'name': 'احمد', 'email': 'ahmad@test.com'},
{'name': 'فاطمه', 'email': 'fateme@test.com'}
])
```

Update – به روزرسانی

```
stmt = users_table.update().where(users_table.c.id == 1).values(name='علی احمدی')

with engine.connect() as conn:
    result = conn.execute(stmt)
    print(f"{result.rowcount} رکورد به روزرسانی شد")
```

Delete – حذف

```
stmt = users_table.delete().where(users_table.c.age < 18)

with engine.connect() as conn:
    result = conn.execute(stmt)
    print(f"{result.rowcount} رکورد حذف شد")
```

Joins

Joinها برای ادغام داده‌ها از دو یا چند جدول بر اساس یک شرط مشترک کاربرد دارند.

1 Inner Join

داده‌ها را فقط زمانی برمی‌گرداند که ردیف‌های هر دو جدول با شرط join مطابقت داشته باشند. مثال کاربردی: گرفتن فقط کاربرانی که حداقل یک سفارش ثبت کرده‌اند.

```
# Inner Join
stmt = select(users_table.c.name, orders_table.c.total).join(
    orders_table, users_table.c.id == orders_table.c.user_id
)
```

مفهوم ساده:

می‌خواهیم اطلاعات کاربر و مجموع سفارشش رو بگیریم.

Inner Join یعنی فقط کاربرانی که حداقل یک سفارش دارند نمایش داده می‌شوند.

شرط `users_table.c.id == orders_table.c.user_id` مشخص می‌کند که کاربر به کدام سفارش وصل شود.

💡 نتیجه: کاربرانی بدون سفارش نمایش داده نمی‌شوند.

2 Outer Join (Left Join)

همه ردیف‌های جدول اصلی (چپ) را برمی‌گرداند، حتی اگر مطابقتی در جدول دوم وجود نداشته باشد. مثال کاربردی: گرفتن همه کاربران، حتی آن‌هایی که هنوز هیچ سفارشی ثبت نکرده‌اند؛ در این صورت ستون‌های جدول دوم (orders) مقدار NULL دارند.

```
# Left (Outer) Join
stmt = select(users_table.c.name, orders_table.c.total).select_from(
    users_table.outerjoin(orders_table)
)
```

مفهوم ساده:

باز هم می‌خواهیم اطلاعات کاربر و مجموع سفارشش رو بگیریم. Left Outer Join یعنی همه کاربران نمایش داده می‌شوند، حتی اگر هیچ سفارشی نداشته باشند. ستون‌های سفارش برای کاربرانی که سفارش ندارند، مقدار NULL خواهند داشت. 💡 نتیجه: هیچ کاربری حذف نمی‌شود، حتی بدون سفارش.

♦ مفهوم C.

C. یعنی ستون‌های جدول (columns).

وقتی می‌نویسیم users_table.c.name یعنی ستون name از جدول users_table.

Operators و Functions

Functions – توابع SQL

مثال‌های کاربردی:

```
from sqlalchemy import select, func

# تعداد کاربران
stmt = select(func.count(users_table.c.id))

# بیشینه، کمینه و میانگین سن کاربران
stmt = select(
    func.max(users_table.c.age),
    func.min(users_table.c.age),
    func.avg(users_table.c.age)
)
```

```
# گروه‌بندی بر اساس شهر و تعداد کاربران
stmt = select(
    users_table.c.city,
    func.count(users_table.c.id)
).group_by(users_table.c.city)
```

نکته: 💡

- `func` معادل توابع SQL است
- می‌توانی از توابع aggregation مثل COUNT, MAX, MIN, AVG, SUM استفاده کنی
- aggregation توابعی است که برای ترکیب یا تجزیه داده‌ها از جدول استفاده می‌شوند.

Operators – عملگرها

مقایسه:

```
users_table.c.age > 18
users_table.c.name == 'علی'
users_table.c.email.like('%gmail%')
users_table.c.id.in([1, 2, 3])
```

عملگرهای منطقی:

```
# AND
(users_table.c.age > 18) & (users_table.c.is_active == True)

# OR
(users_table.c.city == 'تهران') | (users_table.c.city == 'اصفهان')
```

نکته: 💡

- OR = `|`
- AND = `&`
- `like()` برای جستجوی شبیه‌سازی رشته استفاده می‌شود
- `in()` بررسی عضویت در یک لیست

Subqueries

Subquery ساده

```
# کاربرانی که حداقل یک سفارش دارند
subq = select(orders_table.c.user_id).distinct()
```

```
stmt = select(users_table).where(users_table.c.id.in_(subq))
```

scalar و alias با Subquery

```
subq = select(func.avg(users_table.c.age)).scalar_subquery()  
stmt = select(users_table).where(users_table.c.age > subq)
```

EXISTS

```
from sqlalchemy import exists  
  
stmt = select(users_table).where(  
    exists().where(orders_table.c.user_id == users_table.c.id)  
)
```

نکته: 💡

- Subquery = کوئری داخلی
- scalar_subquery() = بازگرداندن یک مقدار تکی
- exists() = بررسی وجود حداقل یک ردیف

Executing Statements

Connection execution

روش‌های اجرا

```
from sqlalchemy import create_engine, text, select  
  
engine = create_engine("sqlite:///example.db")  
  
# روش ۱: با Connection  
with engine.connect() as conn:  
    result = conn.execute(text("SELECT * FROM users"))  
    print(result.fetchall())  
  
# روش ۲: مستقیم از Engine  
result = engine.execute("SELECT COUNT(*) FROM users")  
print(result.scalar())
```

```
# (مدرن‌تر و امن‌تر) Statement Object روش ۳: با
stmt = select(users_table).where(users_table.c.age > 25)
with engine.connect() as conn:
    result = conn.execute(stmt)
    for row in result:
        print(f"نام: {row.name}")
```

پارامترگذاری (Parameterized Queries)

```
with engine.connect() as conn:
    # یک پارامتر
    result = conn.execute(
        text("SELECT * FROM users WHERE age > :min_age"),
        {"min_age": 18}
    )

    # چند پارامتر
    result = conn.execute(
        text("SELECT * FROM users WHERE age BETWEEN :min_age AND :max_age"),
        {"min_age": 18, "max_age": 65}
    )
```

💡 مزیت: امنیت بیشتر (SQL Injection جلوگیری می‌شود)

Result objects

گرفتن داده‌ها

```
stmt = select(users_table)
with engine.connect() as conn:
    result = conn.execute(stmt)

    # یک سطر
    first_row = result.fetchone()
    print(f"اولین کاربر: {first_row.name}")

    # چند سطر
    some_rows = result.fetchmany(5)

    # همه سطرها
    all_rows = result.fetchall()
```

Transactions

✦ تراکنش = مجموعه‌ای از عملیات که یا همه انجام میشه یا هیچ‌کدوم.

ساده (خودکار)

```
with engine.connect() as conn:
    conn.execute(users_table.insert().values(name="احمد"))
    conn.execute(users_table.insert().values(name="علی"))
# خودکار commit در پایان بلوک
```

توضیح:

اینجا وقتی with تمام میشه، SQLAlchemy به‌طور خودکار یک transaction باز کرده بود و اون رو commit می‌کنه.

اگر در وسط اجرای کوئری‌ها خطایی رخ بده، تراکنش به صورت خودکار rollback میشه. یعنی تمام کوئری‌هایی که داخل بلوک with اجرا شدن یا همه ذخیره میشن یا هیچ‌کدوم. پس این حالت اتمیک هست. ✓

دستی

```
conn = engine.connect()
trans = conn.begin()
try:
    conn.execute(users_table.insert().values(name="احمد"))
    conn.execute(users_table.insert().values(name="علی"))
    trans.commit() # ذخیره تغییرات
except:
    trans.rollback() # برگرداندن
finally:
    conn.close()
```

توضیح:

اینجا خودت به صورت دستی یک transaction باز می‌کنی (conn.begin()).

بعد از اجرای کوئری‌ها، باید خودت commit() یا rollback() بزنی.

این روش وقتی لازمه که کنترل کامل روی تراکنش داشته باشی (مثلاً وسط کار شرط خاصی بررسی کنی و تصمیم بگیری commit بشه یا rollback).

این حالت هم اتمیک هست، چون اگر commit نشه، همه تغییرات rollback میشن. ✓

♦ خلاصه:

Connection ← برای اجرای کوئری

Result ← برای گرفتن خروجی و کار با سطرها

Transaction ← برای اطمینان از یکپارچگی داده‌ها

SQLAlchemy ORM

Declarative Base

♦ Declarative Base چیست؟

یک روش شیء‌گرا برای تعریف جدول‌هاست.

به جای اینکه به صورت دستی Table بسازید، یک کلاس پایتون می‌نویسید و SQLAlchemy خودش آن را به جدول دیتابیس متصل (Map) می‌کند.

در SQLAlchemy ما دو راه برای تعریف جدول داریم:

روش دستی (Table-based): با استفاده از Table و ستون‌ها (Column) جدول می‌سازیم.

روش شیء‌گرا (Declarative Base): با استفاده از کلاس‌های پایتون جدول‌ها را تعریف می‌کنیم.

Model definition

روش قدیمی (SQLAlchemy < 2.0)

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

# پایه
Base = declarative_base()

# تعریف مدل
class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    email = Column(String(100))
```

➡ این کد یک جدول به اسم users در دیتابیس می‌سازه. هر شیء از کلاس User معادل یک رکورد (سطر) در جدول هست.

روش جدید (SQLAlchemy 2.0)

```
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column

class Base(DeclarativeBase):
    pass

class User(Base):
    __tablename__ = "users"

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(50))
    email: Mapped[str] = mapped_column(String(100))
```

✓ فرقی اینه که در نسخه ۲.۰، تایپ‌ها (Mapped[int]) واضح‌تر و سازگارتر با پایتون مدرن تعریف شدن.

Table mapping

Table Mapping = وصل کردن یک کلاس پایتون به یک جدول دیتابیس

♦ حالت ساده (اسم‌ها یکسان)

وقتی اسم جدول و ستون‌ها همون اسم کلاس و ویژگی‌ها باشه، کار خیلی ساده‌ست:

```
class User(Base):
    __tablename__ = 'users' # جدول دیتابیس

    id = Column(Integer, primary_key=True) # ستون id
    name = Column(String(50)) # ستون name
```

اینجا کلاس User به جدول users وصل شده.

ویژگی id در پایتون دقیقاً به ستون id در دیتابیس وصل میشه.

ویژگی name در پایتون دقیقاً به ستون name در دیتابیس وصل میشه.

✳ یعنی اسم‌ها یکی هستن ← نگاشت خودکار و مستقیم.

♦ حالت سفارشی (اسم‌ها فرق کنن)

گاهی توی دیتابیس اسم ستون‌ها یا جدول یه چیز دیگه‌ست، اما نمی‌خواهی توی پایتون از همون اسم سخت استفاده کنی. اینجا Mapping سفارشی کمک می‌کنه:

```
class User(Base):
    __tablename__ = 'user_accounts'  # اسمش user_accounts جدول دیتابیس اسمش

    user_id = Column("id", Integer, primary_key=True)  # ویژگی id → ستون
    full_name = Column("name", String(100))  # ویژگی name → ستون
```

جدول واقعی دیتابیس اسمش user_accounts هست.

در دیتابیس ستونی به اسم id داریم، ولی در پایتون اسمش رو user_id گذاشتیم.

در دیتابیس ستونی به اسم name داریم، ولی در پایتون از full_name استفاده می‌کنیم.

اینطوری کد پایتون خواناتر میشه، اما همچنان به دیتابیس وصل میشه. 📌

Primary keys

ساده

```
class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)  # Auto increment
```

با UUID

```
import uuid
from sqlalchemy.dialects.postgresql import UUID

class User(Base):
    __tablename__ = "users"
    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
```

ترکیبی (Composite Key)

```
class OrderItem(Base):
    __tablename__ = "order_items"
    order_id = Column(Integer, primary_key=True)
    product_id = Column(Integer, primary_key=True)
    quantity = Column(Integer)
```

Column options

```
from sqlalchemy import Boolean, DateTime
from datetime import datetime

class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True)
    name = Column(String(50), nullable=False) # اجباری
    email = Column(String(100), unique=True) # باید یکتا باشد
    is_active = Column(Boolean, default=True) # مقدار پیشفرض
    created_at = Column(DateTime, default=datetime.utcnow) # زمان ساخت
    updated_at = Column(DateTime, onupdate=datetime.utcnow) # زمان تغییر
    city = Column(String(50), index=True) # ایندکس
    status = Column(String(20), server_default="active") # پیشفرض سمت سرور
```

Sessions

● Session چیست؟

یک پل ارتباطی بین کد پایتون و دیتابیس.

یعنی همه کارهای ORM (افزودن، تغییر، حذف، خواندن) از طریق Session انجام میشه.

Session تغییرات رو نگه می‌داره تا وقتی که commit() بزنی در دیتابیس ثبت بشن.

📌 پس Session مثل یک دفترچه یادداشت موقت هست که در آخر می‌تونی تصمیم بگیری تغییرات رو ذخیره کنی یا پاک کنی.

این Session در SQLAlchemy را با Session احراز هویت اشتباه نگیرید؛ در ادامه توضیح داده می‌شود.

Session lifecycle

◆ چرخه (Session Lifecycle)

ایجاد Session Factory

```
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)
```

این یعنی "کارخانه"ی ساخت Session.

ایجاد Session

```
session = Session()
```

استفاده از Session

```
user = User(name="احمد")  
session.add(user)
```

Commit (ذخیره در دیتابیس)

```
session.commit()
```

بستن Session

```
session.close()
```

configuring sessions و Creating

```
Session = sessionmaker(  
    bind=engine,  
    autoflush=False,      # نکنه flush تغییرات رو query خودش قبل از  
    autocommit=False,     # نکنه commit خودش  
    expire_on_commit=False # داده ها باطل نشن commit بعد از  
)
```

✦ معمولاً در فریمورک‌ها این تنظیمات رو ست می‌کنن تا کنترل بیشتری داشته باشی.

Session states

```
session = Session()  
  
# 1. Transient (بی ارتباط)  
user = User(name="احمد")  
  
# 2. Pending (در انتظار)  
session.add(user)  
print(user in session.new) # True
```

```
# 3. Persistent (ثابت)
session.commit()
print(user in session) # True
print(user.id) # دارد ID حالا

# 4. Detached (جدا شده)
session.expunge(user) # حذف از session
print(user in session) # False

# 5. Deleted (حذف شده)
session.delete(user)
print(user in session.deleted) # True
```

rollback و Committing

Commit: ذخیره کردن تغییرات در دیتابیس.

Rollback: برگرداندن همه تغییرات (اگر خطا پیش آمده باشد).

مثال:

```
session = Session()
try:
    user1 = User(name="احمد")
    user2 = User(name="علی")
    session.add_all([user1, user2])
    session.commit()
except:
    session.rollback()
finally:
    session.close()
```

♦ استفاده از Context Manager (روش پیشنهادی)

```
with Session() as session:
    user = User(name="احمد")
    session.add(user)
    session.commit()
```

اینطوری Session خودش در پایان بسته میشه. 📌

نکات پایانی: 1 Session در SQLAlchemy / ORM

کاربرد: مدیریت ارتباط با دیتابیس و تراکنش‌ها

مسئول ذخیره، ویرایش، حذف، و خواندن داده‌هاست.

کنترل می‌کند که تغییرات به دیتابیس commit بشن یا rollback.

وضعیت آبجکت‌ها رو نگه می‌داره: Transient, Pending, Persistent, Detached, Deleted

✅ نتیجه: همه عملیات ORM و کوئری‌ها از طریق همین session انجام میشه.

2 Session در فریم‌ورک‌ها (Flask, FastAPI, Django)

کاربرد: مدیریت کاربر لاگین‌شده و احراز هویت

معمولاً وقتی کاربر وارد میشه، اطلاعاتی مثل user_id یا token توی session ذخیره میشه.

این Session به دیتابیس وصل نیست، بلکه یک Context ذخیره اطلاعات بین درخواست‌ها هست.

در Flask/FastAPI معمولاً با cookie یا server-side session store پیاده میشه.

در session، Django، مدیریت شده و اطلاعات داخل دیتابیس یا cache ذخیره میشه.

نوع Session	SQLAlchemy ORM	فریم‌ورک (Flask/FastAPI/Django)
هدف	مدیریت ارتباط با دیتابیس	مدیریت احراز هویت / اطلاعات کاربر
شامل	Object states, تراکنش‌ها	user_id, token, login state
طول عمر	کوتاه‌مدت، معمولاً برای هر request	طولانی‌تر، بین request ها
عملیات مهم	add, commit, rollback, query	set/get/remove, check login
محل ذخیره	حافظه پایتون، دیتابیس	cookie, دیتابیس، cache

♦ نکته کلیدی:

SQLAlchemy Session و Session فریم‌ورک کاملاً مستقل هستند.

می‌تونن همزمان از هر دو استفاده کنی: SQLAlchemy Session برای کار با داده‌ها، و فریم‌ورک Session برای مدیریت کاربر.

Basic Queries

وقتی با Session کار می‌کنی، عملاً داری روی دیتابیس کوئری می‌زنی. Session بهت امکاناتی میده تا داده‌ها رو بخونی، فیلتر کنی، مرتب کنی و محدود کنی.

Query objects

Query Objects (شیء کوئری)

♦ وقتی می‌خوای داده‌ها رو بگیری، اول یه Query Object می‌سازی:

```

session = Session()

# برای جدول Query ساخت یک User
query = session.query(User)

print(type(query))
# خروجی: <class 'sqlalchemy.orm.query.Query'>

```

این Query Object مثل یه قالب آماده‌ست. بعد می‌تونن روی اون اجرا کنی:

```

users = query.all() # همه User ها
first_user = query.first() # اولین User
user_count = query.count() # تعداد کل
one_user = query.one() # باید دقیقاً یکی باشه

```

Filtering

می‌خواهی فقط یه بخشی از داده‌ها رو بگیری؟ از filter یا filter_by استفاده کن.

```

# همه افراد بالای 18 سال
adult_users = session.query(User).filter(User.age >= 18).all()

# همه کاربران فعال
active_users = session.query(User).filter_by(is_active=True).all()

```

♦ چند شرط هم می‌تونن بزنی:

```

query = session.query(User).filter(
    User.age >= 18,
    User.is_active == True,
    User.city == 'تهران'
)

```

📌 اپراتورها:

```

session.query(User).filter(User.name.like('%احمد%')) # شروع با احمد
session.query(User).filter(User.email.contains('@gmail')) # شامل @gmail
session.query(User).filter(User.id.in_([1, 2, 3])) # داخل لیست

```

```
session.query(User).filter(User.age.between(18, 65)) # بین 18 و 65
session.query(User).filter(User.phone.is_(None))      # NULL
```

♦ شرط‌های ترکیبی:

```
from sqlalchemy import and_, or_, not_

# AND
session.query(User).filter(and_(User.age >= 18, User.is_active == True))

# OR
session.query(User).filter(or_(User.city == 'تهران', User.city == 'اصفهان'))

# NOT
session.query(User).filter(not_(User.is_deleted))
```

Ordering

مرتب‌سازی (Ordering)

```
# صعودی
users = session.query(User).order_by(User.name).all()

# نزولی
users = session.query(User).order_by(User.created_at.desc()).all()

# چند ستون
users = session.query(User).order_by(
    User.city,
    User.name.desc()
).all()
```

پیشرفته‌تر: 📌

```
from sqlalchemy import func

# بر اساس طول نام
session.query(User).order_by(func.length(User.name))

# تصادفی
session.query(User).order_by(func.random()).limit(5)
```

```
# NULL بیان  
session.query(User).order_by(User.phone.nullslast())
```

Limiting

خیلی وقتا همه داده‌ها رو نمی‌خوای. برای این Pagination داریم:

```
# فقط 10 رکورد  
users = session.query(User).limit(10).all()  
  
# از رکورد 20 تا 30  
users = session.query(User).offset(20).limit(10).all()  
  
# آخرین 5 نفر  
latest_users = session.query(User)\  
    .order_by(User.created_at.desc())\  
    .limit(5)\  
    .all()
```

Advanced ORM و Relationships

Relationships

وقتی چند جدول داریم، باید بگیم که رابطه بینشون چیه:

- یک کاربر می‌تونه چند سفارش داشته باشه (One-to-Many)
- یک سفارش فقط به یک کاربر تعلق داره (Many-to-One)
- هر کاربر یک پروفایل داره (One-to-One)
- کاربرها می‌تونن نقش‌های مختلف داشته باشن و برعکس (Many-to-Many)
- ForeignKey ← تو دیتابیس می‌گه این رکورد به کدوم جدول وصله. ستونی که به رکورد جدول دیگه اشاره می‌کنه
- relationship ← تو پایتون بهت اجازه میده راحت به رکوردهای مرتبط دسترسی داشته باشی. این دیگه مخصوص ORM (مثل SQLAlchemy یا Django ORM) هست. به جای اینکه مدام JOIN بزنی، می‌تونی مستقیماً از آبجکت‌ها استفاده کنی.

One-to-Many

یک کاربر می‌تونه چند سفارش داشته باشه.


```

class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)
    name = Column(String(50))

    orders = relationship("Order", back_populates="user") # لیست سفارش‌ها

class Order(Base):
    __tablename__ = "orders"
    id = Column(Integer, primary_key=True)
    total = Column(Float)
    user_id = Column(Integer, ForeignKey("users.id")) # سفارش مال کدوم کاربره؟

    user = relationship("User", back_populates="orders")

```

استفاده:

```

user = session.query(User).first()
print(user.orders) # لیست سفارش‌ها

```

در جدول Order یک ForeignKey داریم (user_id) ← این اتصال اصلی توی دیتابیس.

اما ORM (SQLAlchemy) می‌خواد رابطه‌ی دوطرفه بسازه:

از سمت User: دسترسی به لیست سفارش‌ها ← user.orders

از سمت Order: دسترسی به صاحب سفارش ← order.user

اگر فقط یک طرف relationship رو بذاری، از همون سمت کار می‌کنه، ولی معمولاً دوطرفه راحت‌تره (یعنی بتونی هم user.orders بزنی هم order.user).

👉 پس: دوتا relationship برای راحتی در کدنویسیه، نه برای تعیین نوع رابطه. نوع رابطه رو در واقع همون ForeignKey مشخص می‌کنه.

Many-to-One

همون مثال بالاست، فقط از سمت سفارش نگاه می‌کنیم:

```

order = session.query(Order).first()
print(f"این سفارش برای {order.user.name} است")

```

One-to-One

هر کاربر یک پروفایل دارد.

```
class Profile(Base):
    __tablename__ = "profiles"
    id = Column(Integer, primary_key=True)
    bio = Column(String(200))
    user_id = Column(Integer, ForeignKey("users.id"), unique=True)

    user = relationship("User", back_populates="profile")

class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)
    name = Column(String(50))

    profile = relationship("Profile", back_populates="user", uselist=False)
```

کلید خارجی (FK) یعنی «این ستون به جدول دیگر وصل میشه».

وقتی `unique=True` هم بزنی یعنی هر `user_id` فقط یک بار می‌تونه تو جدول بیاد.

✅ نتیجه: اینطوری رابطه تبدیل میشه به One-to-One (هر کاربر یک پروفایل می‌تونه داشته باشه).

بدون `unique=True` میشه One-to-Many (هر کاربر می‌تونه چند پروفایل داشته باشه).

به طور پیش‌فرض `relationship` فکر می‌کنه چقدر آجکت برگردونه (مثلاً `user.orders` → یک لیست).

ولی وقتی رابطه یک‌به‌یک (One-to-One) باشه، می‌خواهی یه آجکت تکی برگردونه نه لیست.

اینجاست که `uselist=False` استفاده می‌شود.

Many-to-Many

کاربرها می‌تونن چند نقش داشته باشن (ادمین، کاربر ساده).

```
user_roles = Table(
    "user_roles", Base.metadata,
    Column("user_id", Integer, ForeignKey("users.id")),
    Column("role_id", Integer, ForeignKey("roles.id"))
)

class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)
```

```

name = Column(String(50))

roles = relationship("Role", secondary=user_roles, back_populates="users")

class Role(Base):
    __tablename__ = "roles"
    id = Column(Integer, primary_key=True)
    name = Column(String(50))

    users = relationship("User", secondary=user_roles, back_populates="roles")

```

این `secondary=user_roles` در `Many-to-Many` چیه؟ در رابطه‌ی `Many-to-Many` باید یک جدول میانی داشته باشیم در اینجا جدول `user_roles` مثل پل بین `users` و `roles` عمل می‌کنه.

Back references

back_populates ♦

باید دو طرف رابطه رو خودت تعریف کنی.

اسم attribute (ویژگی) رو دستی مشخص می‌کنی.

مثال `One-to-Many` (کاربر ↔ سفارش‌ها):

```

class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)
    name = Column(String(50))

    orders = relationship("Order", back_populates="user") # سمت User

class Order(Base):
    __tablename__ = "orders"
    id = Column(Integer, primary_key=True)
    total = Column(Float)
    user_id = Column(Integer, ForeignKey("users.id"))

    user = relationship("User", back_populates="orders") # سمت Order

```

یعنی: 📌

از سمت کاربر: `user.orders` ← لیست سفارش‌ها

از سمت سفارش: `order.user` ← کاربر مربوط به اون سفارش

اینجا دو خط `relationship` داری و باید دقیقاً همدیگه رو اشاره بدن.

backref ♦

به جای اینکه دو بار تعریف کنی، یک بار تعریف می‌کنی، و sqlalchemy خودش طرف مقابل رو می‌سازه. یعنی shortcut هست.

مثال همون بالا، اما کوتاه‌تر:

```
class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)
    name = Column(String(50))

class Order(Base):
    __tablename__ = "orders"
    id = Column(Integer, primary_key=True)
    total = Column(Float)
    user_id = Column(Integer, ForeignKey("users.id"))

    user = relationship("User", backref="orders")
```

حالا هم داری: 📌

```
order.user
user.orders
```

ولی فقط یک بار relationship نوشتی.

Lazy Loading vs Eager Loading

Lazy loading patterns

داده‌ها وقت نیاز load می‌شوند:

```
user = session.query(User).first() # کوئری میشه User فقط جدول
print(user.name)

# لود میشه orders وقتی اینجا دسترسی می‌کنی، تازه
print(len(user.orders))

# حالا به کوئری جدید زده میشه =>
```

Eager loading (joinedload, selectinload)

همه داده‌ها یکجا load می‌شوند:

♦ فرق joinedload و selectinload

- هر دو برای Eager Loading هستند (یعنی داده‌ها رو همون اول میارن)، ولی روش کارشون فرق می‌کنه:

joinedload 📌

از JOIN در همون کوئری اصلی استفاده می‌کنه.

همه‌چیز با یک کوئری میاد.

سریع‌تره وقتی داده کم باشه.

مشکل: اگه جدول فرزند (مثل orders) خیلی رکورد داشته باشه، داده تکراری زیادی کشیده میشه.

مثال: 📌

```
users = session.query(User).options(joinedload(User.orders)).all()
# SQL: SELECT users.*, orders.* FROM users LEFT JOIN orders ...
```

selectinload 📌

اول همه Userها رو میاره، بعد یه کوئری جدا برای همه ordersها میزنه.

نتیجه: دو کوئری (ولی بهینه).

بهره‌تره وقتی رکوردهای فرزند زیادن.

مثال: 📌

```
users = session.query(User).options(selectinload(User.orders)).all()
# SQL 1: SELECT * FROM users
# SQL 2: SELECT * FROM orders WHERE user_id IN (list of all ids)
```

پس: ✅

داده کم ← joinedload خوبه.

داده زیاد ← selectinload بهینه‌تره.

N+1 problem


وقتی برای گرفتن یک مجموعه داده، ۱ کوئری اولیه + N کوئری اضافی برای هر رکورد زده بشه. 📌 مثال
:(N+1)

```
users = session.query(User).all() # کوئری 1
for user in users:
    print(len(user.orders))      # N اضافه (جدا User برای هر) کوئری اضافه
```

مثال (Eager Loading): 

```
users = session.query(User).options(joinedload(User.orders)).all()
for user in users:
    print(len(user.orders)) # کوئری اضافه نمی‌خوره
```

Loading strategies

Loading Strategies (روش‌های بارگذاری رابطه‌ها) 

وقتی relationship تعریف می‌کنی، می‌تونی تعیین کنی چطور داده لود بشه:

1. Lazy (پیش‌فرض: "select")

کوئری فقط وقتی لازم باشه زده میشه.  مثال:

```
orders = relationship("Order", lazy="select")
# اجرا میشه orders صدا بزنی، کوئری user.orders فقط وقتی
```

2. Eager همیشه ("joined")

همراه موجودیت اصلی (User) با JOIN میاره.  مثال:

```
orders = relationship("Order", lazy="joined")
# هم همزمان میاد orders، ها رو بگیری User وقتی
```

3. فقط وقت نیاز ("dynamic")

به جای لیست، یه Query object برمی‌گردونه.

می‌تونی روی همون رابطه فیلتر و limit بزنی.

 مثال:

```
orders = relationship("Order", lazy="dynamic")
```

```
user = session.query(User).first()
big_orders = user.orders.filter(Order.price > 100).all()
```

4. دستی ("noload") Load

هیچوقت خودکار لود نمیشه.

اگه بخوای باید خودت دستی کوئری بزنی.

مثال: 📌

```
orders = relationship("Order", lazy="noload")
user = session.query(User).first()
print(user.orders) # خالی برمی‌گرده
```

Advanced Querying

گاهی کوئری‌های ساده کافی نیستن و باید سراغ ابزارهای پیشرفته‌تر بریم: Joins، Subquery، Union، Window Functions، Raw SQL.

Complex joins

Inner Join (اتصال مستقیم) 📌

فقط داده‌هایی میاره که تو هر دو جدول وجود دارن.

```
result = session.query(User, Order)\
    .join(Order)\
    .filter(Order.total > 100)\
    .all()
# همه کاربرانی که سفارششون بیشتر از 100 باشه
```

Outer Join (اتصال بیرونی) 📌

حتی اگه داده در جدول دوم نباشه، رکورد جدول اول رو میاره.

```
result = session.query(User)\
    .outerjoin(Order)\
    .filter(Order.id.is_(None))\
    .all()
# همه کاربرانی که هیچ سفارشی ندارن
```

Subqueries

کوئری‌ای که داخل یه کوئری دیگه استفاده میشه.

Subquery عادی

```
subq = session.query(Order.user_id)\
    .filter(Order.total > 1000)\
    .subquery()

rich_users = session.query(User)\
    .filter(User.id.in_(subq))\
    .all()

# کاربران که سفارشی بیشتر از 1000 دارن
```

Scalar Subquery

یه مقدار تکی (مثل میانگین یا ماکس) برمی‌گردونه.

```
avg_total = session.query(func.avg(Order.total)).scalar_subquery()

above_avg_orders = session.query(Order)\
    .filter(Order.total > avg_total)\
    .all()

# سفارش‌هایی که بیشتر از میانگین کل باشن
```

توضیح scalar_subquery:

```
session.query(func.avg(Order.total)).scalar()
```

👉 این واقعاً یک عدد پایتونی برمی‌گردونه (مثلاً 125.3). یعنی کوئری اجرا میشه الان همین لحظه و مقدار میانگین از دیتابیس میاد. (دیگه نمی‌تونی توی کوئری‌های بعدی ازش استفاده کنی.)

```
session.query(func.avg(Order.total)).scalar_subquery()
```

👉 این یه زیرکوئری می‌سازه که هنوز داخل دیتابیس اجرا نشده. می‌تونی این زیرکوئری رو به‌عنوان بخشی از شرط کوئری بزرگ‌تر استفاده کنی. `scalar_subquery()` خودش یک آبجکت زیرکوئری (SQL expression object) برمی‌گردونه

Union operations

برای ترکیب خروجی چند کوئری مختلف.

♦ Union نتیجه‌ی دو کوئری رو ترکیب می‌کنه بدون تکراری‌ها.

📌 Union (بدون تکراری‌ها)

```
young_users = session.query(User).filter(User.age < 25)
old_users = session.query(User).filter(User.age > 65)

all_users = young_users.union(old_users).all()
# همه کاربران، یا خیلی جوان یا خیلی پیر
```

♦ Union All نتیجه‌ی دو کوئری رو ترکیب می‌کنه با تکراری‌ها.

📌 Union All (با تکراری‌ها)

```
all_users = young_users.union_all(old_users).all()
```

Window functions

برای تحلیل داده‌ها روی ردیف‌ها (مثل رتبه‌بندی یا شماره‌گذاری).

📌 Row Number به هر ردیف یک شماره‌ی یکتا میده، صرفاً برای ترتیب.

```
result = session.query(
    User.name,
    func.row_number().over(order_by=User.created_at).label('row_num')
).all()
# شماره ردیف برای هر کاربر بر اساس زمان ثبت‌نام
```

📌 Rank رتبه میده، ولی اگه دو تا رکورد مساوی باشن رتبه‌ی یکسان می‌گیرن.

مثال ساده:

- User 1 ← سفارش 500 ← رتبه 1
- User 1 ← سفارش 500 ← رتبه 1
- User 1 ← سفارش 300 ← رتبه 3

```
result = session.query(
    Order.user_id,
```

```
Order.total,
func.rank().over(
    partition_by=Order.user_id,
    order_by=Order.total.desc()
).label('rank')
).all()
# رتبه سفارش هر کاربر بر اساس مبلغ
```

Raw SQL integration

وقتی کوئری خیلی پیچیده باشد یا از فانکشن خاص دیتابیس بخوای استفاده کنی.

- text() برای نوشتن کوئری خام.
- pattern به placeholder هست که مقدارش رو به صورت امن تزریق می‌کنیم.
- fetchall() ← همه ردیف‌ها.
- fetchone() ← فقط یک ردیف.
- fetchmany(5) ← تعداد مشخصی رکورد.
- from_statement() وقتی می‌خواهی کوئری خام رو به ORM وصل کنی (خروجی ORM بشه)

اجرای مستقیم SQL

```
result = session.execute(
    text("SELECT * FROM users WHERE name LIKE :pattern"),
    {"pattern": "احمد%"}
).fetchall()
# همه کاربرانی که اسمشون با احمد شروع بشه
```

ترکیب با ORM

```
users = session.query(User)\
    .from_statement(
        text("SELECT * FROM users WHERE complex_condition()")
    ).all()
```

Advanced Topics

Session Management Patterns

در Session، SQLAlchemy مسئول مدیریت ارتباط با دیتابیس و اجرای کوئری‌هاست. روش‌های مختلفی برای مدیریت Session داریم که در فریمورک‌ها استفاده می‌شوند.

Session per request

برای هر درخواست HTTP یک Session جدید ساخته می‌شود و بعد از پایان درخواست بسته می‌شود. این روش رایج و امن است چون Session ها بین درخواست‌ها به اشتراک گذاشته نمی‌شوند.

مثال FastAPI

```
def get_db():
    db = SessionLocal() # جدید Session ایجاد یک
    try:
        yield db # endpoint استفاده در
    finally:
        db.close() # بعد از پایان درخواست Session بسته شدن

@app.get("/users")
def get_users(db: Session = Depends(get_db)):
    return db.query(User).all()
```

Contextual sessions

گاهی چند تابع یا چند Thread باید از یک Session استفاده کنند. scoped_session این کار را انجام می‌دهد و هر Thread یک Session اختصاصی دارد.

```
from sqlalchemy.orm import scoped_session, sessionmaker

Session = scoped_session(sessionmaker(bind=engine))

def some_function():
    user = Session.query(User).first() # فعلی Thread برای Session همان
    return user

# بعد از اتمام کار Session پاک کردن
Session.remove()
```

- sessionmaker یک کارخانه (Factory) برای ساخت Session است.
- engine مسئول ارتباط با دیتابیس است (مانند TCP connection یا SQLite file).
- وقتی bind=engine می‌گذاری، یعنی این Session بداند که با کدام دیتابیس کار کند.

مثال:

```
engine = create_engine("sqlite:///app.db")
Session = sessionmaker(bind=engine)
session = Session() # متصل است "app.db" به Session این
```

Thread-local sessions

هر Thread خودش یک Session جداگانه دارد. مشابه scoped_session ولی دستی مدیریت می‌شود.

```
import threading

session_registry = threading.local()

def get_session():
    if not hasattr(session_registry, 'session'):
        session_registry.session = SessionLocal()
    return session_registry.session
```

- session_registry یک فضای ذخیره‌سازی مخصوص هر Thread است (threading.local()).
- وقتی می‌نویسیم session_registry.session = SessionLocal()، داریم یک Session اختصاصی برای Thread فعلی می‌سازیم و ذخیره می‌کنیم.

Session events

می‌توانیم قبل یا بعد از commit/rollback تغییرات دیتابیس عملیات دلخواه انجام دهیم، مثل logging یا validation.

```
from sqlalchemy import event

@event.listens_for(Session, 'before_commit')
def before_commit(session):
    print("Commit قبل از")

@event.listens_for(Session, 'after_commit')
```

```
def after_commit(session):
    print("Commit بعد از")
```

before_commit ← قبل از ذخیره نهایی تغییرات

after_commit ← بعد از ذخیره نهایی تغییرات

Advanced Relationships

SQLAlchemy امکانات پیشرفته‌ای برای مدل‌سازی روابط بین جدول‌ها دارد. در این بخش به مواردی مهم می‌پردازیم

Self-referential relationships

یک جدول می‌تواند به خودش ارجاع داشته باشد

```
class Employee(Base):
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    manager_id = Column(Integer, ForeignKey('employees.id'))

    manager = relationship("Employee", remote_side=[id], back_populates="subordinates")
    subordinates = relationship("Employee", back_populates="manager")
```

remote_side در SQLAlchemy:

- وقتی یک جدول به خودش ارجاع می‌دهد (Self-referential)، SQLAlchemy باید بداند کدام ستون «سمت دیگر رابطه» است.

Polymorphic relationships

- می‌توان کلاس‌های فرزند را در یک ساختار مشترک ذخیره کرد و SQLAlchemy می‌فهمد هر ردیف از کدام نوع است
- وقتی داری چند نوع موجودیت داری که بعضی ستون‌هاشون مشترکه، می‌تونی از Polymorphic استفاده کنی.

```
class Person(Base):
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    type = Column(String(20)) # مشخص می‌کند نوعش چیه
    __mapper_args__ = {'polymorphic_on': type, 'polymorphic_identity': 'person'}

class Employee(Person):
```

```

salary = Column(Float)
__mapper_args__ = {'polymorphic_identity': 'employee'}

class Customer(Person):
    credit_limit = Column(Float)
    __mapper_args__ = {'polymorphic_identity': 'customer'}

```

۱. نقش mapper_args

این یک دیکشنری مخصوص SQLAlchemy ORM است که به کلاس می‌گوید چگونه با جدول و وراثت رفتار کند.

مخصوصاً وقتی از Polymorphic Inheritance (وراثت جدول‌ها) استفاده می‌کنیم، اینجا مشخص می‌کنیم که SQLAlchemy چطور نوع رکوردها را تشخیص دهد.

۲. polymorphic_on

polymorphic_on: type

مشخص می‌کند کدام ستون جدول نوع (Type) رکورد را نگه می‌دارد.

در مثال ما، ستون type مشخص می‌کند که این رکورد یک Person است یا Employee یا Customer. یعنی هر رکورد وقتی در جدول ذخیره می‌شود، این ستون مقدارش تعیین می‌کند که SQLAlchemy کدام کلاس را به آن اختصاص دهد.

۳. polymorphic_identity

polymorphic_identity مقداری است که در ستون type جدول ذخیره می‌شود و مشخص می‌کند رکورد مربوط به کدام کلاس است.

این مقدار می‌تواند هر رشته‌ای باشد، ولی معمولاً خوانا و مرتبط با کلاس انتخاب می‌شود.

Hybrid properties

می‌توان متدی در کلاس تعریف کرد که هم در Python قابل دسترسی باشد و هم در کوئری SQL استفاده شود.

```

from sqlalchemy.ext.hybrid import hybrid_property
from sqlalchemy import func

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    first_name = Column(String(30))
    last_name = Column(String(30))

    @hybrid_property # for python

```

```

def full_name(self):
    return self.first_name + ' ' + self.last_name

@full_name.expression # for SQL
def full_name(cls):
    return func.concat(cls.first_name, ' ', cls.last_name)

# استفاده
user = session.query(User).first()
print(user.full_name) # "احمد علی"

```

وقتی بخواهیم از `full_name` در `filter()` یا `order_by()` استفاده کنیم، SQLAlchemy خودش اون رو به SQL تبدیل می‌کنه:

```

users = session.query(User).filter(User.full_name == 'احمد علی').all()
# استفاده میشه @full_name.expression اینجا از

```

Extensions و Customization

Custom types

برای زمانی که می‌خواهیم نوع داده‌ای غیر استاندارد در دیتابیس ذخیره کنیم، مثل JSON:

```

from sqlalchemy.types import TypeDecorator, String
import json

class JSONType(TypeDecorator):
    impl = String

    def process_bind_param(self, value, dialect):
        return json.dumps(value) if value else value

    def process_result_value(self, value, dialect):
        return json.loads(value) if value else value

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    preferences = Column(JSONType())

# استفاده
user = User(preferences={'theme': 'dark', 'lang': 'fa'})

```

TypeDecorator در SQLAlchemy یعنی:

- من می‌خواهم یک نوع دیتابیس سفارشی بسازم که پشت‌صحنه روی یک نوع استاندارد پیاده بشه.
- فرض کن دیتابیس فقط String و Integer و... رو می‌فهمه.
- ولی من می‌خواهم یک ستون داشته باشم که بتونه دیکشنری/JSON ذخیره کنه.

:impl

- impl یعنی نوع اصلی دیتابیس که این نوع سفارشی روی اون سوار شده.
- اینجا می‌گیم: این JSONType در اصل همون String هست
- پس داخل دیتابیس یه VARCHAR/TEXT ذخیره میشه، ولی توی Python یه دیکشنری/JSON می‌بینیم.

:process_bind_param

- وقتی می‌خواهی داده رو بفروستی توی دیتابیس (insert/update)، این تابع اجرا میشه.

:process_result_value

- وقتی از دیتابیس داده رو بگیری (select)، این تابع اجرا میشه.

Validators

با @validates می‌توان قبل از ذخیره داده، اعتبارسنجی انجام داد:

```
from sqlalchemy.orm import validates

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    email = Column(String(100))
    age = Column(Integer)

    @validates('email')
    def validate_email(self, key, email):
        if '@' not in email:
            raise ValueError("ایمیل نامعتبر")
        return email

    @validates('age')
    def validate_age(self, key, age):
        if age < 0 or age > 150:
            raise ValueError("سن نامعتبر")
        return age
```

✓ جلوی داده‌های اشتباه قبل از Commit گرفته می‌شود.

دکوراتور @validates:

- قبل از اینکه داده در این فیلد ذخیره بشه، اول از این تابع ردش کن
- key: اسم فیلدی که validate میشه (مثلاً 'email').
- داده اگه درست باشه، return میشه و ذخیره میشه.
- اگه اشتباه باشه، Exception میندازه.

اعتبارسنجی چند فیلد در یک تابع

اگه بخوای می‌تونی یک تابع برای چند فیلد بذاری:

```
@validates('email', 'age')
def validate_fields(self, key, value):
    if key == 'email':
        if '@' not in value:
            raise ValueError("ایمیل نامعتبر")
    elif key == 'age':
        if value < 0 or value > 120:
            raise ValueError("سن نامعتبر")
    return value
```

listeners و Events

می‌توانیم به Insert، Update، Delete واکنش نشان دهیم:

```
from sqlalchemy import event
from datetime import datetime

@event.listens_for(User, 'before_insert')
def set_created_at(mapper, connection, target):
    target.created_at = datetime.utcnow()

@event.listens_for(User, 'after_update')
def log_update(mapper, connection, target):
    print(f"User {target.id} updated")
```

Mixins

می‌توانیم ستون‌ها و ویژگی‌های مشترک را در یک کلاس بنویسیم و به مدل‌ها اضافه کنیم:

```
class TimestampMixin:
    created_at = Column(DateTime, default=datetime.utcnow)
    updated_at = Column(DateTime, default=datetime.utcnow, onupdate=datetime.utcnow)
```

```
class User(Base, TimestampMixin):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
```

Custom loading techniques

برای کنترل اینکه چه داده‌هایی هنگام Query بارگذاری شوند

مشکل رایج: N+1 Query Problem

وقتی لیست کاربران رو می‌گیری، و بعد برای هر کاربر جداگانه Orders یا Profile رو query می‌کنی.

این میشه ده‌ها کوئری جدا (خیلی کند).

راه‌حل: Loader یا joinedload / selectinload سفارشی.

```
from sqlalchemy.orm import joinedload

# loader سفارشی
class UserLoader:
    @staticmethod
    def full_load():
        return [joinedload(User.orders), joinedload(User.profile)]

users = session.query(User).options(*UserLoader.full_load()).all()
```

نکته: هم می‌توانم یک loader نوشتت وهم می‌توان و هم می‌توان دستی به این صورت نوشت

مثال کامل FastAPI

```
from fastapi import FastAPI, Depends
from sqlalchemy.orm import Session, joinedload

app = FastAPI()

@app.get("/users/{user_id}/full")
def get_user_full(user_id: int, db: Session = Depends(get_db)):
    user = db.query(User)\
        .options(joinedload(User.orders), joinedload(User.profile))\
        .filter(User.id == user_id)\
        .first()
    return user
```

Optimization و Performance

Performance Tuning

Query optimization

اشتباه رایج: همه داده‌ها رو بیاریم، بعد در پایتون فیلتر کنیم.
روش درست: از خود SQL برای فیلتر و محدود کردن داده‌ها استفاده کنیم.

```
# بد: همه کاربرها، بعد در پایتون فیلتر
all_users = session.query(User).all()
active_users = [u for u in all_users if u.is_active]

# خوب: مستقیم در دیتابیس فیلتر
active_users = session.query(User).filter(User.is_active == True).all()
```

Index strategies

ایندکس مثل فهرست کتاب عمل می‌کند ← به جای ورق زدن کل جدول، سریع رکورد مورد نظر پیدا میشه.
برای ستون‌هایی که زیاد سرچ/فیلتر میشن، ایندکس بزن.

♦ مثال:

```
class User(Base):
    __tablename__ = 'users'
    email = Column(String(100), index=True)  # ایندکس روی ایمیل
```

♦ ایندکس ترکیبی (چند ستون با هم):

```
Index('idx_status_date', Order.status, Order.created_at)
```

Connection pooling

هر بار وصل شدن به دیتابیس هزینه‌بره.
Connection Pool باعث میشه اتصال‌ها کش بشن و دوباره استفاده بشن.

♦ مثال:

```
engine = create_engine(
    "postgresql://user:pass@localhost/db",
    pool_size=20,          # اتصال آماده 20
    max_overflow=10,       # اتصال اضافی در اوج 10
    pool_timeout=30,       # ثانیه منتظر بمونه 30
    pool_recycle=3600,     # هر 1 ساعت تازه سازی
    pool_pre_ping=True     # قبل استفاده تست کن
)
```

Bulk operations

به جای یکی یکی insert/update/delete → همه رو یکجا بزن.

:Insert ♦

```
# بد
for i in range(1000):
    session.add(User(name=f"User{i}"))
session.commit()

# خوب
users_data = [{"name": f"User{i}"} for i in range(1000)]
session.bulk_insert_mappings(User, users_data)
session.commit()
```

:Update ♦

```
# خوب
session.query(User)\
    .filter(User.last_login < old_date)\
    .update({"is_active": False})
session.commit()
```

:Delete ♦

```
session.query(LogEntry)\
    .filter(LogEntry.created_at < old_date)\
    .delete()
session.commit()
```

Caching

کش به معنی ذخیره موقت داده‌ها در حافظه است تا دفعه بعدی که به همان داده‌ها نیاز داریم، دیگر به دیتابیس مراجعه نکنیم و پاسخ سریع‌تر باشد.

انواع کش و کاربردها

۱. کش ساده درون‌پردازشی (in-process cache)

- محل ذخیره: حافظه همان پروسس Python.
- محدودیت: فقط برای همان پروسس است و در چند سرور یا چند پروسس به اشتراک گذاشته نمی‌شود.
- مثال مفهومی: یک دیکشنری ساده که نتایج کوئری را نگه می‌دارد.
- استفاده رایج: `functools.lru_cache` برای ذخیره نتایج تابع.

۲. کش پراسس مشترک (distributed cache)

- محل ذخیره: سیستم بیرونی مثل Redis یا Memcached.
- همه پروسس‌ها و سرورها می‌توانند از آن استفاده کنند.
- می‌توان TTL (زمان انقضا) برای داده‌ها تعریف کرد.
- نیازمند سریال‌سازی داده‌ها است (JSON یا pickle).

۳. Second-level cache (SQLAlchemy)

- SQLAlchemy خودش کشی در سطح مدل و session دارد.
- مفهوم: یک رکورد از دیتابیس که بار اول خوانده شده، در cache سطح دوم نگه داشته می‌شود تا Session بعدی بتواند بدون مراجعه به دیتابیس آن را استفاده کند.
- مثال مفهومی: کاربر با `id=1` بار اول از دیتابیس خوانده شد، بار دوم از کش داخلی SQLAlchemy برمی‌گردد.

۴. Dogpile.cache (Cache حرفه‌ای)

- یک ابزار قدرتمند برای مدیریت کش.
- منطقه‌های کش (Region) قابل تنظیم.
- TTL (زمان انقضا) و invalidation خودکار.
- پشتیبانی از Redis، Memcached، و حافظه داخلی.
- مفهوم: توابع یا کوئری‌ها را wrap می‌کند و نتایج را به صورت امن در کش می‌گذارد.
- پاک‌سازی کش (invalidation) بعد از تغییر داده‌ها ضروری است تا stale data برنگردد.

Query result caching

کش ساده با `functools.lru_cache`

ایده: نتایج یک تابع را در حافظه نگه می‌داریم تا دفعه بعدی بدون رفتن به دیتابیس، همان نتیجه را برگرداند.

```
from functools import lru_cache

# تابعی که کاربر را از دیتابیس می‌خواند
@lru_cache(maxsize=100) # حداکثر 100 نتیجه در حافظه نگه داشته شود
def get_user_by_email(email):
    print("Querying DB for", email)
    return session.query(User).filter(User.email == email).first()

# استفاده
user1 = get_user_by_email("test@example.com") # این بار از دیتابیس خوانده می‌شود
user2 = get_user_by_email("test@example.com") # این بار از کش خوانده می‌شود
```

Second-level cache

هدف: داده‌هایی که بین چند Session یا Thread مشترک هستند، در یک کش مرکزی نگه داشته شود. برخلاف کش ساده‌ی Session (که فقط در همان Session فعال است)، این کش بین Session‌های مختلف قابل استفاده است.

```
from sqlalchemy_utils import CacheManager

# راه‌اندازی Cache Manager
cache_manager = CacheManager()

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(String(50))

    # برای این مدل Cache
    cache = cache_manager.cache

# استفاده
user = session.query(User).filter(User.id == 1).first() # از DB
user = session.query(User).filter(User.id == 1).first() # از Cache
```

Dogpile.cache integration

یک کتابخانه مخصوص کش که امکانات پیشرفته دارد، مثل invalidation خودکار و TTL.

```
pip install dogpile.cache
```

```
from dogpile.cache import make_region

# تنظیم Region
region = make_region().configure(
    'dogpile.cache.memory', # هم استفاده کرد Redis حافظه داخلی؛ می‌توان
    expiration_time=600     # دقیقه 10
)

@region.cache_on_arguments()
def get_user_profile(user_id):
    return session.query(User).filter(User.id == user_id).first()

# استفاده
user = get_user_profile(1) # بار اول از DB
user = get_user_profile(1) # بار دوم از کش

# پاک کردن کش بعد از تغییر دیتا
def update_user(user_id, **kwargs):
    session.query(User).filter(User.id == user_id).update(kwargs)
    session.commit()

    region.delete(f"get_user_profile|{user_id}") # پاک کردن کش
```

نکته: ✓

`cache_on_arguments()` به طور خودکار کلید کش می‌سازد.
بعد از آپدیت دیتا، حتماً کش را پاک کنیم تا داده تازه برگردد.

Advanced Features

Alembic با Migrations

وقتی مدل‌ها (`models.py`) تغییر می‌کنن (مثلاً به ستون یا جدول اضافه می‌کنی)، دیتابیس به صورت خودکار تغییر نمی‌کنه.

Migration ابزاریه که تغییرات رو قدم به قدم ثبت و روی دیتابیس اعمال می‌کنه، مثل نسخه‌بندی برای دیتابیس.

Migration basics

نصب و راه‌اندازی:

```
pip install alembic

# شروع پروژه
alembic init alembic
```

یک پوشه‌ی /alembic ساخته میشه.

داخلش env.py هست که باید Base.metadata رو بهش بدی (تا بفهمه مدل‌ها چیه).
داخل alembic.ini آدرس دیتابیس رو می‌ذاری:

```
# تنظیم alembic.ini
sqlalchemy.url = postgresql://user:pass@localhost/dbname
```

Auto-generating migrations

وقتی مدل تغییر کرد (مثلاً User اضافه شد):

```
alembic revision --autogenerate -m "Add user table"
alembic upgrade head
```

دستور اول یک فایل migration در alembic/versions می‌سازه.

دستور دوم اون migration رو روی دیتابیس اجرا می‌کنه.

برای برگشت:

```
alembic downgrade -1 # برگرد عقب migration یک
```

یعنی upgrade اعمال می‌کنه، downgrade برعکسش رو.

مثال ساده Migration تولیدشده


```
def upgrade():
    op.create_table(
        'users',
        sa.Column('id', sa.Integer, primary_key=True),
        sa.Column('name', sa.String(50)),
        sa.Column('email', sa.String(100))
    )

def downgrade():
    op.drop_table('users')
```

Manual migrations

Manual Migration یعنی وقتی خودکار (autogenerate) کافی نیست – باید دستی Schema و/یا داده‌ها را تغییر دهی.

Manual Migration یعنی شما یک revision خالی می‌سازید (alembic revision -m "...") و داخل upgrade() و downgrade() خودتان کدهای *.op و SQL لازم را می‌نویسید.

♦ قالب کلی یک فایل migration

```
# versions/xxxx_custom_migration.py
from alembic import op
import sqlalchemy as sa

revision = 'xxxx'
down_revision = 'prev_rev'
branch_labels = None
depends_on = None

def upgrade():
    # دستوراتی که هنگام بالا بردن باید اجرا شود
    pass

def downgrade():
    # برای بازگشت upgrade معکوس
    pass
```

نکته: همیشه یک downgrade() بنویس حتی اگر ساده باشد تا در صورت نیاز بتوانی برگردی.
این یخش کمی تخصصی تر است و از گفتن در اینجا اجتناب میشود

merging و Branching

وقتی چند نفر روی پروژه کار می‌کنن، هرکسی ممکنه migration خودش رو بسازه → migration شاخه‌ای میشن.

ساخت :branch:

```
alembic revision -m "Feature A" --branch-label=feature_a
```

ادغام چند migration:

```
alembic merge -m "merge features" head1 head2
```

دیدن تاریخچه:

```
alembic history --verbose
```

دیدن وضعیت فعلی دیتابیس:

```
alembic current
```

Migration مثل git commit برای دیتابیس.

Testing

Testing patterns

@pytest.fixture چیست؟

در Fixture، pytest یک راهکار برای آماده‌سازی داده‌ها، منابع، یا محیط تست است.

به جای اینکه در هر تست دوباره چیزهایی مثل Session یا داده بسازیم، Fixture این کار را یک بار انجام می‌دهد و به تست‌ها تزریق می‌شود.

"scope="session" یعنی چه؟

Fixture می‌تواند طول عمر مختلف داشته باشد:

- function (پیش فرض): هر تست یک نمونه جدید دریافت می‌کند.
- class: هر کلاس تست یک نمونه دارد.
- module: هر ماژول یک نمونه.
- session: کل تست‌ها در یک اجرا یک نمونه مشترک دارند.

Setup تست

از SQLite in-memory برای تست سریع استفاده می‌کنیم:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from myapp.models import Base
import pytest

@pytest.fixture(scope="session")
def test_engine():
    engine = create_engine("sqlite:///memory:") # پایگاه داده موقت در حافظه
    Base.metadata.create_all(engine)
    return engine

@pytest.fixture
def session(test_engine):
    Session = sessionmaker(bind=test_engine)
    session = Session()
    yield session
    session.rollback() # بعد از هر تست همه تغییرات برگردانده شود
    session.close()
```

هر تست یک Session تازه دارد، پس تغییرات بین تست‌ها تداخل ندارد.
scope session یعنی ایجاد engine فقط یک بار برای کل تست‌ها.

تست ساده مدل‌ها

```
def test_create_user(session):
    user = User(name="احمد", email="ahmad@test.com")
    session.add(user)
    session.commit()

    assert user.id is not None # کاربر ساخته شد
    assert user.name == "احمد"
```

تست رابطه‌ها

```
def test_user_relationships(session):
    user = User(name="علی")
    order = Order(total=100.0)
    user.orders.append(order)

    session.add(user)
    session.commit()

    assert len(user.orders) == 1
    assert user.orders[0].total == 100.0
```

Fixtures

می‌توانیم نمونه‌های آماده بسازیم و در چندین تست استفاده کنیم:

```
@pytest.fixture
def sample_user(session):
    user = User(name="کاربر تست", email="test@example.com")
    session.add(user)
    session.commit()
    return user
```

Mock strategies

فرض کنید تابعی به API خارجی یا سرویس دیگر متصل می‌شود، ما نمی‌خواهیم هر بار تست این سرویس واقعی را صدا بزنند.

Mock یعنی جایگزین کردن واقعی با نسخه‌ی شبیه‌سازی شده که نتیجه دلخواه برگرداند.

Database testing

تست‌های پایگاه داده واقعی

می‌توانیم PostgreSQL واقعی یا MySQL استفاده کنیم:

```
@pytest.fixture(scope="session")
def postgres_engine():
    engine = create_engine("postgresql://test_user:test_pass@localhost/test_db")
    Base.metadata.create_all(engine)
    yield engine
    Base.metadata.drop_all(engine)

@pytest.fixture
```

```
def postgres_session(postgres_engine):  
    Session = sessionmaker(bind=postgres_engine)  
    session = Session()  
    yield session  
    session.rollback()  
    session.close()
```

مزیت: تست روی پایگاه داده واقعی انجام می‌شود، اشکالات واقعی SQL پیدا می‌شوند.

Real-world Applications

Design Patterns

Repository pattern

چیست؟ یک لایه بین کدهای اپلیکیشن و دیتابیس ایجاد می‌کند تا دسترسی به داده‌ها مرتب و استاندارد شود. یعنی به جای اینکه مستقیم `session.query(User)` بنویسید، از یک Repository استفاده می‌کنید.

مزیت:

جداسازی منطق دیتابیس از بیزینس لایه

راحت‌تر کردن تست‌ها

تغییر دیتابیس بدون تغییر کل کد

مثال ساده:

```
# repository.py  
class UserRepository:  
    def __init__(self, session):  
        self.session = session  
  
    def get_by_id(self, user_id):  
        return self.session.query(User).filter(User.id == user_id).first()  
  
    def add(self, user):  
        self.session.add(user)  
        self.session.commit()  
  
# استفاده  
repo = UserRepository(session)
```

```
user = repo.get_by_id(1)
repo.add(User(name="Ali"))
```

نتیجه: کل اپلیکیشن به Repository وابسته است، نه مستقیم به Session یا SQLAlchemy. ✓

Unit of Work

چیست؟ یک الگو برای مدیریت تراکنش‌ها و Session‌ها. اطمینان می‌دهد که همه تغییرات به صورت یکجا commit یا rollback شوند.

مثال ساده:

```
class UnitOfWork:
    def __init__(self, session_factory):
        self.session_factory = session_factory

    def __enter__(self):
        self.session = self.session_factory()
        return self.session

    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type:
            self.session.rollback()
        else:
            self.session.commit()
        self.session.close()

# استفاده
with UnitOfWork(Session) as session:
    user = User(name="Ahmad")
    session.add(user)  # می‌شود rollback اگر خطایی باشد
```

نکته مهم: متدهای **enter** و **exit** خودکار اجرا می‌شوند وقتی که از **with** استفاده می‌کنید.

وقتی `with UnitOfWork(Session) as session:` می‌نویسید، Python خودش **enter** را صدا می‌زند و نتیجه (session) را به شما می‌دهد.

- وقتی بلاک **with** تمام می‌شود (حتی اگر خطا رخ داده باشد)، Python خودش **exit** را صدا می‌زند.
- داخل **exit** ما مشخص کرده‌ایم:
- اگر خطا بود `rollback()`
- اگر خطا نبود `commit()`
- بعد هم Session بسته می‌شود با `close()`

Data Mapper

چیست؟ SQLAlchemy خودش از این الگو استفاده می‌کند. ایده اصلی: کلاس‌های Python به جدول‌های دیتابیس Map شوند و کلاس‌ها به صورت مستقل از دیتابیس عمل کنند.

- کلاس‌ها فقط داده نگه می‌دارند و منطق بیزینسی.
- Session / Mapper مسئول ذخیره‌سازی و بارگذاری داده‌هاست.

مثال ساده:

```
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))

# Mapper جدا از کلاس User
user = User(name="Sara")
session.add(user)
session.commit()
```

Mapper همین Base و SQLAlchemy ORM است که پشت صحنه کلاس User را به جدول users نگاشت می‌کند.

شما اصلاً SQL ننوشته‌اید.

Mapper این کار را انجام می‌دهد: تبدیل User به یک INSERT INTO users ... و ارسال به دیتابیس.

✓ یعنی Mapper همان چیزی است که بین کلاس پایتون و جدول دیتابیس ارتباط برقرار می‌کند.

✓ نتیجه: کلاس‌های شما مستقل هستند و دیتابیس فقط با Mapper و Session کار می‌کند.

Active Record considerations

چیست؟ یک الگو که در آن کلاس هم داده و هم منطق دیتابیس را در خود دارد. مثلاً کلاس خودش Delete، Save، Update را انجام می‌دهد.

این الگو در Django ORM یا Rails زیاد دیده می‌شود.

SQLAlchemy می‌تواند شبیه Active Record شود اما Data Mapper بیشتر توصیه می‌شود چون انعطاف بیشتری دارد.

مثال ساده Active Record-style:

```
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
```

```

name = Column(String(50))

def save(self, session):
    session.add(self)
    session.commit()

# استفاده
user = User(name="Neda")
user.save(session) # کلاس خودش داده را ذخیره کرد

```

جمع بندی:

1 Session

کار اصلی: مدیریت همه تعاملات با دیتابیس.

وظایفش:

- گرفتن داده‌ها (query)
- اضافه کردن یا حذف کردن رکوردها (add, delete)
- commit و rollback تراکنش‌ها
- نگه داشتن وضعیت Object ها (Transient, Pending, Persistent, Detached)

می‌توانی فکر کنی: Session مثل یک دفتر کار است که تمام تغییرات روی دیتابیس را ثبت و مدیریت می‌کند.

2 Mapper (یا ORM)

کار اصلی: نگاشت کلاس‌های Python به جدول‌های دیتابیس.

وظایفش:

- مشخص کردن چه کلاس Python ای مربوط به کدام جدول است (Declarative Base)
 - تبدیل Object ها به SQL و برعکس
 - مدیریت روابط بین کلاس‌ها (One-to-Many, Many-to-Many, Polymorphic)
- می‌توانی فکر کنی: Mapper همان پلی بین کلاس‌های Python و جدول‌های دیتابیس است.

Integration

Web framework integration (Flask, FastAPI)

اینجا هدف این است که SQLAlchemy را در کنار یک فریمورک وب استفاده کنیم تا Request ها بتوانند به راحتی با دیتابیس کار کنند.

FastAPI مثال:


```

from fastapi import FastAPI, Depends
from sqlalchemy.orm import Session
from database import SessionLocal, User

app = FastAPI()

# Session per Request
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

@app.get("/users/{user_id}")
def get_user(user_id: int, db: Session = Depends(get_db)):
    return db.query(User).filter(User.id == user_id).first()

```

Async SQLAlchemy

نسخه async برای کار با دیتابیس به صورت غیرهمزمان در فریمورک‌هایی مثل FastAPI یا Starlette استفاده می‌شود.

```

from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession
from sqlalchemy.orm import sessionmaker
from models import User

engine = create_async_engine("postgresql+asyncpg://user:pass@localhost/db")
AsyncSessionLocal = sessionmaker(engine, class_=AsyncSession, expire_on_commit=False)

async def get_user(user_id: int):
    async with AsyncSessionLocal() as session:
        result = await session.get(User, user_id)
        return result

```

✅ نکته: Async برای اپلیکیشن‌هایی با I/O بالا بسیار مناسب است و جلوی بلاک شدن event loop را می‌گیرد.

Multi-database setups

گاهی پروژه چند دیتابیس دارد (مثلاً خواندن و نوشتن جدا یا دیتابیس‌های مختلف برای ماژول‌ها). SQLAlchemy اجازه تعریف چند engine و Session می‌دهد.

```
# برای دو دیتابیس مختلف Engine دو
engine_main = create_engine("postgresql://user:pass@main_db")
engine_logs = create_engine("postgresql://user:pass@logs_db")

SessionMain = sessionmaker(bind=engine_main)
SessionLogs = sessionmaker(bind=engine_logs)

# استفاده
with SessionMain() as main_session, SessionLogs() as log_session:
    user = main_session.query(User).first()
    log_session.add(LogEntry(message="Fetched user"))
    log_session.commit()
```

Sharding strategies

Sharding یعنی تقسیم داده‌ها روی چند دیتابیس برای مقیاس‌پذیری. SQLAlchemy خودش Sharding را مدیریت می‌کند.

```
from sqlalchemy.ext.horizontal_shard import ShardedSession

# فرض کنید دو دیتابیس داریم
shards = {
    'shard_1': create_engine('postgresql://user:pass@db1'),
    'shard_2': create_engine('postgresql://user:pass@db2')
}

def shard_chooser(mapper, instance, clause=None):
    # تصمیم‌گیری برای اینکه رکورد کجا برود
    return 'shard_1' if instance.id % 2 == 0 else 'shard_2'

session = ShardedSession(shards=shards, shard_chooser=shard_chooser)

user = User(id=5, name="Ali")
session.add(user) # مناسب می‌رود shard خودکار روی
session.commit()
```

✓ نکته: Sharding بیشتر برای دیتابیس‌های بزرگ و مقیاس‌پذیر استفاده می‌شود.

🎉 پایان مرجع جامع SQLAlchemy

تبریک! 🎉

شما با موفقیت یکی از کامل‌ترین و جامع‌ترین مراجع فارسی SQLAlchemy را مطالعه کردید. این مرجع شامل:

- ✓ مفاهیم پایه از صفر تا صد
- ✓ تکنیک‌های پیشرفته برای پروژه‌های واقعی
- ✓ بهترین روش‌ها (Best Practices) در صنعت
- ✓ الگوهای طراحی حرفه‌ای
- ✓ بهینه‌سازی عملکرد و کش
- ✓ تست و Migration با Alembic
- ✓ ادغام با فریمورک‌های وب مدرن

🚀 پیشنهادات بعدی

حالا که SQLAlchemy را به خوبی یاد گرفته‌اید:

1. پروژه عملی بسازید - بهترین راه یادگیری، تمرین است
2. با FastAPI ترکیب کنید - برای ساخت API های مدرن
3. Async SQLAlchemy امتحان کنید - برای اپلیکیشن‌های پرتراфик
4. PostgreSQL پیشرفته یاد بگیرید - برای استفاده بهینه از قابلیت‌ها

🙏 تشکر ویژه

از صبر و همراهی شما در این سفر یادگیری تشکر می‌کنیم. امیدواریم این مرجع برای شما مفید بوده و در پروژه‌های آینده‌تان کمک‌تان کند.

📌 در تماس باشیم

اگر سوال، پیشنهاد، یا نیاز به راهنمایی بیشتر داشتید، خوشحال می‌شویم کمک‌تان کنیم.

موفق و پیروز باشید! 🍀

🌟 ساخته شده با ❤️ توسط امیرحسین بابایی برای جامعه توسعه‌دهندگان فارسی‌زبان 🌟