Beyond Data Modeling with SQLAlchemy



Douglas Starnes
Author

linktr.ee/douglasstarnes

Database Transactions



A unit of work containing one or more database operations



Ensures that all tasks in the transaction succeed



If one task fails, the entire transaction is rolled back



Transactions are ACID

SQLAlchemy Transactions

Must occur in a `Session`

The `begin()` method starts a transaction

Errors will automatically roll back the transaction

Explicitly call the `rollback()` method to cancel a transaction

Nested transactions

SQLAlchemy Transactions

```
with Session(bind=engine) as session:
  with session.begin():
    calendar = Calendar("Personal Calendar")
    session.add(calendar)
    session.commit()
```



Nested Transactions

```
with Session(bind=engine) as session:
  with session.begin():
    calendar = Calendar("Personal Calendar")
    with session.begin_nested():
      event = CalendarEvent("First Personal Event")
      session.add(event)
    session.add(calendar)
    session.commit()
```



Explict Rollback

```
with Session(bind=engine) as session:
  try:
    calendar = Calendar("Personal Calendar")
    session.add(calendar)
    session.commit()
  except Exception:
    session.rollback()
```



Async SQLAlchemy

Everything depends on the `AsyncSession`

Methods like `scalars()`, `commit()`, **and** `execute()` **are awaitable**

Works great with FastAPI

Async SQLAlchemy

```python
from sqlalchemy.ext.asyncio import (
  create_async_engine,
  AsyncSession
)

DATABASE_URL = “sqlite+aiosqlite:///countdown.db”

engine = create_async_engine(DATABASE_URL)

async def create_tables():
  async with engine.begin() as cn:
    await cn.run_sync(Base.metadata.create_all)
```



Async SQLAlchemy

```python
async def add_calendar_event(
  session: AsyncSession,
  name: str,
  date: str):
  new_event = CalendarEvent(
    event_name=name,
    event_date=date
  )
  session.add(new_event)
  await session.commit()

async def list_calendar_events(
  session: AsyncSession
):
  query = select(CalendarEvent)
  events = await session.scalars(query)
  return events.all()
```



FastAPI

```python
from fastapi import FastAPI, Depends

app = FastAPI()

async def get_db():
  async_session = AsyncSession(
    engine, expire_on_commit=False
  )
  try:
    yield async_session
  finally:
    await async_session.close()

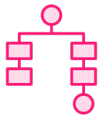
@app.get("/events")
async def list_events(
  db: AsyncSession = Depends(get_db)
):
  events = await list_calendar_events(db)
  return events
```



SQLModel



SQLAlchemy ORM objects are not JSON-serializable



FastAPI pattern is to create a Pydantic schema



The data from SQLAlchemy objects populates the Pydantic schema



SQLModel combines the two

SQLModel

```python
from sqlmodel import SQLModel, Field, select

class CalendarEventBase(SQLModel):
  event_name: str = Field(...)
  event_date: str = Field(...)

class CalendarEvent(CalendarEventBase, table=True):
  __tablename__ = “calendar_event”

  id: int = Field(
    default=None, primary_key=True
  )

class CalendarEventRead(CalendarEventBase):
  id: int
```



SQLModel

```python
async def list_calendar_events(
  session: AsyncSession,
):
  query = select(CalendarEvent)
  result = await session.scalars(query)
  return result.all()

@app.get(
  "/events",
  response_model=List[CalendarEventRead]
)
async def list_events(
  db: AsyncSession = Depends(get_db)
):
  events = await list_calendar_events(db)
  return events
```



Switching to a Different Database

Often developers use a local database (SQLite) during development but deploy to a database server (PostgreSQL) in production

Switching databases in SQLAlchemy is often as simple as changing the connection string

Migrations



Scripts to manage changes to the structure of a relational database



Migrations are generated based on SQLAlchemy model classes



Apply migrations to make changes to the database



Revert to previous migrations and return to earlier versions



Alembic is a Python package implementing migrations

SQLite

A local, file-based database

Python supports SQLite with the `sqlite3` package

Query SQLite with SQL statements and get lists of tuples

Use row factories to transform the tuples in Python objects

SQLAlchemy ORM

Mapped columns add type annotations to the data model

One-to-many relationships

Many-to-many relationships

Association proxies

Querying SQLAlchemy

Connect to the database with an engine

Interact with the database through a session

Write queries using the SQLAlchemy Core API

Work with ORM objects

Beyond Data Modeling

Transition the database by changing the connection string

Manage data model changes with Alembic migrations

Transactions

Async support and SQLModel

Thank you!

