

SQLAlchemy ORM for FastAPI

Version 2.1.9 (FastAPI 0.165) · verified 2026-11-18

SQLAlchemy 2.1 async ORM in FastAPI — typed models, a session per request, select(), writes, eager loading and the MissingGreenlet trap.

Contents

Mental model 3

What it solves 3

Models 4

Engine and session factory 6

A session per request 8

Reading rows 10

Writing rows 11

Relationships and eager loading 13

Responses with Pydantic 15

Further reading 17

Mental model

FastAPI gives you validated HTTP in and JSON out; it has no opinion about storage. SQLAlchemy fills that gap: you declare Python classes once, and the ORM turns them into tables, turns `select()` into SQL, and tracks which objects changed so a single `commit()` writes them. The unit that does the tracking is the **Session** — one per request, never shared — and under asyncio it may only touch the database when you `await` it. Most async SQLAlchemy bugs are an attribute trying to load data without an `await`.

What it solves

Without an ORM	With SQLAlchemy
Hand-written SQL strings	`select(User).where(...)`
Rows as tuples	Typed `User` objects
Tracking what changed	Unit of work: `commit()`
Opening a connection per query	Engine with a pool

Raw SQL scatters column names across string literals that no type checker sees. SQLAlchemy keeps the schema in one Python class, builds SQL from typed expressions, pools connections, and wraps each request in a transaction. You still write SQL-shaped code — the ORM does not hide joins or `WHERE` clauses, it composes them.

```
pip install "sqlalchemy[asyncio]" asyncpg
pip install fastapi uvicorn
```

> **Note:** 2.1 requires Python 3.11+ and no longer installs `greenlet` by default. Without the `[asyncio]` extra, `create_async_engine` fails at startup.

Models

Construct	Meaning
`class Base(DeclarativeBase)`	Registry for all models

Construct	Meaning
`Mapped[int]`	Column type, `NOT NULL`
`Mapped[str | None]`	Nullable column
`mapped_column(...)`	Keys, defaults, indexes
`String(50)`	Explicit SQL type / length

The `Mapped[...]` annotation sets both the Python type and nullability; `mapped_column()` adds what the annotation cannot say — keys, defaults, indexes. A plain `Mapped[str]` with no `mapped_column()` is a valid column.

```
from sqlalchemy import String
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)
  email: Mapped[str] = mapped_column(
    String(255), unique=True)
  bio: Mapped[str | None]
```

> **Gotcha:** `bio: Mapped[str]` without `| None` is `NOT NULL`. Optional in Pydantic does not mean nullable here — the annotation is the only signal.

Engine and session factory

Object	Lifetime
`create_async_engine(url)`	One per app (owns the pool)
`async_sessionmaker()`	One per app (a factory)
`AsyncSession`	One per request
`await engine.dispose()`	At shutdown, in `lifespan`

The engine holds the connection pool, so create it once and dispose it in the app's `lifespan` on shutdown. The session factory fixes the options every session shares. Set `expire_on_commit=False`: by default `commit()` expires every loaded attribute, and reading one afterwards needs a query you did not `await`. `pool_pre_ping=True` tests each connection on checkout, so a database restart does not surface as a burst of 500s.

```
from contextlib import asynccontextmanager
from fastapi import FastAPI
from sqlalchemy.ext.asyncio import (
  async_sessionmaker, create_async_engine)
engine = create_async_engine(
  "postgresql+asyncpg://app@db/app",
  pool_size=5, pool_pre_ping=True)
SessionLocal = async_sessionmaker(
  engine, expire_on_commit=False)

@asynccontextmanager
```

```python
async def lifespan(app: FastAPI):
  yield
  await engine.dispose()  # close pooled conns
app = FastAPI(lifespan=lifespan)
```

> **Warning:** `Base.metadata.create_all()` creates missing tables but never alters existing ones. Use it in tests; ship schema changes as Alembic migrations.

A session per request

Piece	Role
`async with SessionLocal()`	Opens, always closes
`yield session`	Hands it to the route
`Annotated[..., Depends()]`	Reusable `DB` type
`db: DB` in a route	Injects the session

A `yield` dependency gives each request its own session and closes it when the request ends, even on error.

Commit in the route, where the write happens. A single `AsyncSession` is not safe across concurrent tasks — never store one globally or pass it into `asyncio.gather()`.

```
from typing import Annotated
from fastapi import Depends
from sqlalchemy.ext.asyncio import AsyncSession

async def get_db():
  async with SessionLocal() as session:
    yield session  # closed after the response

DB = Annotated[AsyncSession, Depends(get_db)]

@app.get("/health")
async def health(db: DB):
  return {"in_tx": db.in_transaction()}
```

> **Gotcha:** Code after `yield` runs **after** the response is sent. A `commit()` placed there that fails still returns 200 to the client. Commit in the route, or use `Depends(get_db, scope="function")` (FastAPI 0.121+).

Reading rows

Call	Returns
`await db.get(User, 1)`	`User | None` by primary key
`await db.scalars(stmt)`	Iterable of `User`
`.all()` / `.first()`	`list` / first or `None`
`.one()`	Exactly one, else raises
`await db.scalar(stmt)`	First column of first row

Build a statement with `select()`, then execute it on the session. `scalars()` returns ORM objects; `execute()` returns `Row` tuples, which you want only when selecting

several columns or entities. `get()` checks the identity map first and may skip the query entirely.

```
from sqlalchemy import func, select

@app.get("/users")
async def list_users(db: DB, q: str = ""):
  stmt = select(User).where(
    User.email.contains(q)).limit(50)
  return (await db.scalars(stmt)).all()

@app.get("/users/count")
async def count_users(db: DB):
  stmt = select(func.count()).select_from(User)
  return await db.scalar(stmt)  # 3
```

Writing rows

Call	Effect
`db.add(obj)`	Stage an INSERT (no `await`)

Call	Effect
`await db.flush()`	Send SQL, get IDs, no commit
`await db.commit()`	Flush and commit
`await db.delete(obj)`	Stage a DELETE
`await db.refresh(obj)`	Reload server-side values

You never write `UPDATE` for a loaded object: change its attributes and the session notices. `add()` only stages the object; the SQL goes out at `flush()` or `commit()`. Use `flush()` when you need a generated primary key before the transaction ends.

```
from pydantic import BaseModel


class UserIn(BaseModel):
  email: str
  bio: str | None = None


@app.post("/users", status_code=201)
```

```python
async def create_user(body: UserIn, db: DB):
  user = User(**body.model_dump())
  db.add(user)
  await db.commit()
  return user  # user.id is set
```

> **Gotcha:** With autoflush on, any query — in 2.1 even a raw `text()` one — flushes pending changes first. An `IntegrityError` can surface on a `SELECT` far from the `add()` that caused it.

Relationships and eager loading

Option	Use for
`selectinload(User.posts)`	Collections — default choice
`joinedload(Post.author)`	Many-to-one; one JOIN
`lazy="raise"`	Forbid implicit loads
`awaitable_attrs.posts`	One-off load (`AsyncAttrs`)

A relationship attribute is loaded lazily by default: the first access runs a query. Under asyncio that access is not awaited, so it raises `MissingGreenlet` ("greenlet_spawn has not been called"). Load what the response needs up front with a loader option on the statement. Below, `User.posts` is attached after the fact; declaring `posts: Mapped[list[Post]]` inside `User` is equivalent.

```
from sqlalchemy import ForeignKey
from sqlalchemy.orm import (
  relationship, selectinload)
class Post(Base):
  __tablename__ = "posts"
  id: Mapped[int] = mapped_column(
    primary_key=True)
  user_id: Mapped[int] = mapped_column(
    ForeignKey("users.id"))
  author: Mapped[User] = relationship(
    back_populates="posts")
```

```
User.posts = relationship(
  Post, back_populates="author", lazy="raise")
stmt = select(User).options(
  selectinload(User.posts))
```

> **Gotcha:** `joinedload()` on a collection repeats the parent row once per child, so `scalars()` raises until you call `.unique()` on the result. `selectinload()` avoids this with a second `IN` query.

Responses with Pydantic

Setting	Effect
`from_attributes=True`	Read ORM object attributes
`response_model=UserOut`	Filter fields on the way out
`-> list[UserOut]`	Same, from the annotation

Returning a bare ORM object serializes every loaded column, including ones you meant to keep private. Declare an output model with `from_attributes=True` so

Pydantic reads attributes instead of dict keys, and the response contains exactly its fields. A nested model field reads the relationship, so the statement must eager-load it.

```
from fastapi import HTTPException
from pydantic import ConfigDict

class UserOut(BaseModel):
  model_config = ConfigDict(from_attributes=True)
  id: int
  email: str

@app.get("/users/{uid}")
async def read_user(uid: int, db: DB) -> UserOut:
  user = await db.get(User, uid)
  if user is None:
    raise HTTPException(404, "user not found")
  return user
```

Further reading

- SQLAlchemy 2.1 ORM Quick Start
- SQLAlchemy asyncio extension
- Unified tutorial (Core and ORM)
- What's New in SQLAlchemy 2.1
- FastAPI: Dependencies with yield
- Alembic tutorial