SQLAlchemy

FastAPI Users provides the necessary tools to work with SQL databases thanks to SQLAlchemy ORM with asyncio.

Asynchronous driver

To work with your DBMS, you'll need to install the corresponding asyncio driver. The common choices are:

- For PostgreSQL: `pip install asyncpg`
- For SQLite: `pip install aiosqlite`

Examples of `DB_URL`s are:

- PostgreSQL: `postgresql+asyncpg://user:password@host:port/name`
- SQLite: `sqlite+aiosqlite:///name.db`

For the sake of this tutorial from now on, we'll use a simple SQLite database.

> **Warning**
>
> When using asynchronous sessions, ensure `Session.expire_on_commit` is set to `False` as recommended by the SQLAlchemy docs on asyncio. The examples on this documentation already have this setting correctly defined to `False` when using the `async_sessionmaker` factory.

Create the User model

As for any SQLAlchemy ORM model, we'll create a `User` model.

```
from collections.abc import AsyncGenerator

from fastapi import Depends
from fastapi_users.db import SQLAlchemyBaseUserTableUUID, SQLAlchemyUserDatabase
from sqlalchemy.ext.asyncio import AsyncSession, async_sessionmaker,
create_async_engine
from sqlalchemy.orm import DeclarativeBase
```

```
DATABASE_URL = "sqlite+aiosqlite:///./test.db"


class Base(DeclarativeBase):
    pass


class User(SQLAlchemyBaseUserTableUUID, Base):
    pass


engine = create_async_engine(DATABASE_URL)
async_session_maker = async_sessionmaker(engine, expire_on_commit=False)


async def create_db_and_tables():
    async with engine.begin() as conn:
        await conn.run_sync(Base.metadata.create_all)


async def get_async_session() -> AsyncGenerator[AsyncSession, None]:
    async with async_session_maker() as session:
        yield session


async def get_user_db(session: AsyncSession = Depends(get_async_session)):
    yield SQLAlchemyUserDatabase(session, User)
```

As you can see, **FastAPI Users** provides a base class that will include base fields for our `User` table. You can of course add you own fields there to fit to your needs!

Primary key is defined as UUID

By default, we use UUID as a primary key ID for your user. If you want to use another type, like an auto-incremented integer, you can use `SQLAlchemyBaseUserTable` as base class and define your own `id` column.

```
class User(SQLAlchemyBaseUserTable[int], Base):
    id: Mapped[int] = mapped_column(Integer, primary_key=True)
```

Notice that `SQLAlchemyBaseUserTable` expects a generic type to define the actual type of ID you use.

Implement a function to create the tables

We'll now create an utility function to create all the defined tables.

```
from collections.abc import AsyncGenerator

from fastapi import Depends
from fastapi_users.db import SQLAlchemyBaseUserTableUUID, SQLAlchemyUserDatabase
from sqlalchemy.ext.asyncio import AsyncSession, async_sessionmaker,
create_async_engine
from sqlalchemy.orm import DeclarativeBase

DATABASE_URL = "sqlite+aiosqlite:///./test.db"


class Base(DeclarativeBase):
    pass


class User(SQLAlchemyBaseUserTableUUID, Base):
    pass


engine = create_async_engine(DATABASE_URL)
async_session_maker = async_sessionmaker(engine, expire_on_commit=False)


async def create_db_and_tables():
    async with engine.begin() as conn:
        await conn.run_sync(Base.metadata.create_all)


async def get_async_session() -> AsyncGenerator[AsyncSession, None]:
    async with async_session_maker() as session:
        yield session


async def get_user_db(session: AsyncSession = Depends(get_async_session)):
    yield SQLAlchemyUserDatabase(session, User)
```

This function can be called, for example, during the initialization of your FastAPI app.

> **Warning**
>
> In production, it's strongly recommended to setup a migration system to update your SQL schemas. See Alembic.

Create the database adapter dependency

The database adapter of **FastAPI Users** makes the link between your database configuration and the users logic. It should be generated by a FastAPI dependency.

```
from collections.abc import AsyncGenerator

from fastapi import Depends
from fastapi_users.db import SQLAlchemyBaseUserTableUUID, SQLAlchemyUserDatabase
from sqlalchemy.ext.asyncio import AsyncSession, async_sessionmaker,
create_async_engine
from sqlalchemy.orm import DeclarativeBase

DATABASE_URL = "sqlite+aiosqlite:///./test.db"


class Base(DeclarativeBase):
    pass


class User(SQLAlchemyBaseUserTableUUID, Base):
    pass


engine = create_async_engine(DATABASE_URL)
async_session_maker = async_sessionmaker(engine, expire_on_commit=False)


async def create_db_and_tables():
    async with engine.begin() as conn:
        await conn.run_sync(Base.metadata.create_all)


async def get_async_session() -> AsyncGenerator[AsyncSession, None]:
    async with async_session_maker() as session:
        yield session


async def get_user_db(session: AsyncSession = Depends(get_async_session)):
    yield SQLAlchemyUserDatabase(session, User)
```

Notice that we define first a `get_async_session` dependency returning us a fresh SQLAlchemy session to interact with the database.

It's then used inside the `get_user_db` dependency to generate our adapter. Notice that we pass it two things:

- The `session` instance we just injected.
- The `User` class, which is the actual SQLAlchemy model.