Module 9: Asynchronous Programming

Overview

- Understanding `async` and `await`
- Async database interactions with
  - `asyncpg` Postgres driver
  - `Databases` package
  - `SQLAlchemy ORM`
- Building async endpoints

Understanding Async and Await

- `async`: Declares a function as asynchronous → returns a coroutine
- `await`: Suspends execution until the awaited coroutine finishes
- **Benefits:**
  - Non-blocking I/O
  - Efficient for high-concurrency apps
- **When to use:**
  - Network calls (HTTP, DB queries)
  - File operations

Asynchronous function - Example

```
import asyncio

async def greet():
    await asyncio.sleep(1)
    print("Hello Async!")

asyncio.run(greet())
```

Asynchronous Programming in Python

Colab Notebook

Async in FastAPI

- FastAPI supports async def endpoints
- Allows handling multiple requests concurrently
- Blocking calls should be avoided in async endpoints

Async endpoints

```python
from fastapi import FastAPI
import asyncio

app = FastAPI()

@app.get("/async-example")
async def async_example():
    await asyncio.sleep(1)
    return {"message": "Async Response"}
```

Using databases in FastAPI asynchronously

- I. `asyncpg` Postgres driver
- II. `Databases` package
- III. SQLAlchemy ORM (with async support)

I. asyncpg

- A low-level, highly optimized PostgreSQL driver for Python (async I/O).
- Talks directly to PostgreSQL without extra layers
- **Pros:**
  - Fastest, Full async support, no blocking
- **Cons:**
  - No ORM, raw SQL, boilerplate code

II. Databases package

- A query builder + async database access layer
- Works with multiple databases (PostgreSQL, MySQL, SQLite) and supports SQLAlchemy core syntax
- **Pros:**
  - Fully async, easier connection management
- **Cons:**
  - slower than `asyncpg`
  - No ORM - you have to handle model <--> dict conversions manually

III. SQLAlchemy ORM with async support

- A full-featured ORM that maps Python classes to database tables.
- The oRM sits on top of SQLAlchemy Core.
- **Pros:**
  - Most feature-rich; maintainable for complex schemas
- **Cons:**
  - More overhead — generally slower than asyncpg or Databases for simple queries; Async support is newer

FastAPI & Databases

Feature	asyncpg	Databases	SQLAlchemy ORM
Performance	🔥 Fastest	⚡ Fast	🚗 Slower
Abstraction level	Low	Medium	High
Async Support	✅ Full	✅ Full	✅ (1.4+)
Write raw SQL	Yes	Optional	Rarely
Type safety	Manual	Partial	Strong
Learning curve	Medium	Low-Med	High
Maintainability	Low	Medium	High
Best for	Max speed	Async no ORM	Large apps, complex schemas

Code Examples

I. asyncpg Database Interactions

```
import asyncpg
import asyncio
DATABASE_URL = "postgresql://user:pass@localhost/db"

async def fetch_users():
    conn = await asyncpg.connect(DATABASE_URL)
    rows = await conn.fetch("SELECT * FROM users;")
    # Format each row as a dict for readability
    formatted_rows = [dict(row) for row in rows]
    # Print formatted records
    print("Fetched Users:")
    for record in formatted_rows:
        print(record)
    await conn.close()
    return formatted_rows
```

II. FastAPI + Databases Library(1)

```python
from fastapi import FastAPI
import asyncio
import databases
import os

from dotenv import load_dotenv

load_dotenv()

# Read DB_USER and DB_PASS from environment variables
DB_USER = os.getenv("DB_USER", "postgres")
DB_PASS = os.getenv("DB_PASS", "postgres")
print(f"DB_USER: {DB_USER}, DB_PASS: {DB_PASS}")

DATABASE_URL = f'postgresql://{DB_USER}:{DB_PASS}@localhost/fastapi_week6'
database = databases.Database(DATABASE_URL)

app = FastAPI()
```

II. FastAPI + Databases Library(2)

```python
@app.on_event("startup")
async def startup():
    await database.connect()

@app.on_event("shutdown")
async def shutdown():
    await database.disconnect()

@app.get("/users")
async def get_users():
    query = "SELECT * FROM users;"
    return await database.fetch_all(query)
```

III. FastAPI + SQLAlchemy 2.0 async ORM

Key differences from the previous **databases** version:

- Pure async with SQLAlchemy 2.0's async ORM.
- Uses AsyncSession and **create_async_engine** with `asyncpg`.
- Table creation (`Base.metadata.create_all`) runs in async context.
- No need for the `databases` library at all.

III. FastAPI + SQLAlchemy 2.0 async ORM (1)

Database Setup

```
# Load environment variables!!!
DATABASE_URL = f'postgresql+asyncpg://{DB_USER}:{DB_PASS}@localhost/fastapi_week9'

engine = create_async_engine(DATABASE_URL, echo=True)
AsyncSessionLocal = sessionmaker(
    bind=engine, class_=AsyncSession, expire_on_commit=False
)

Base = declarative_base()
```

III. FastAPI + SQLAlchemy 2.0 async ORM (2)

ORM model

```
class User(Base):
    __tablename__ = "users"

    id: Mapped[int] = mapped_column(Integer, primary_key=True, index=True)
    name: Mapped[str] = mapped_column(String, nullable=False)
    email: Mapped[str] = mapped_column(String, unique=True, index=True)
```

III. FastAPI + SQLAlchemy 2.0 async ORM (3)

FastAPI app

```
app = FastAPI()

# Dependency for async session
async def get_session() -> AsyncSession:
    async with AsyncSessionLocal() as session:
        yield session

@app.on_event("startup")
async def on_startup():
    async with engine.begin() as conn:
        await conn.run_sync(Base.metadata.create_all)
```

III. FastAPI + SQLAlchemy 2.0 async ORM (4)

Endpoint for creating a new user

```python
@app.post("/users", response_model=UserRead)
async def create_user(user: UserCreate, session: AsyncSession = Depends(get_session)):
    new_user = User(name=user.name, email=user.email)
    session.add(new_user)
    await session.commit()
    await session.refresh(new_user)
    return new_user
```

```python
class UserCreate(BaseModel):
    name: str
    email: str
```

III. FastAPI + SQLAlchemy 2.0 async ORM (5)

Endpoint for retrieving all users

```python
@app.get("/users", response_model=List[UserRead])
async def get_users(session: AsyncSession = Depends(get_session)):
    result = await session.execute(select(User))
    users = result.scalars().all()
    return users
```

```python
class UserRead(BaseModel):
    id: int
    name: str
    email: str
```

Remember

- `async` / `await` → enables non-blocking concurrency
- Use async DB drivers for scalability
- Avoid blocking code in async routes
- FastAPI makes async-first development easy