FastAPI Async SQLAlchemy Project Structure

Fully async FastAPI with SQLAlchemy 2.0 and Alembic migrations. Production-ready database setup.

#fastapi #python #api #rest #sqlalchemy #async #alembic

PNG PDF Copy Prompt

Project Directory

```
myproject/
    main.py App factory, li…
  > app/ Application code
      __init__.py
      config.py Pydantic settin…
      dependencies.py Shared DI depen…
    > core/ Core utilities
        __init__.py
        database.py Async engine & …
        security.py JWT, hashing
        exceptions.py Custom HTTP exc…
    > models/ SQLAlchemy mode…
        __init__.py Export all mode…
        base.py DeclarativeBase…
        user.py
        item.py Example domain …
    > schemas/ Pydantic schemas
        __init__.py
        user.py UserCreate, Use…
        item.py
        common.py Pagination, res…
    > api/ API routes
        __init__.py
        deps.py Route dependenc…
      > v1/ Version 1 endpo…
          __init__.py
          router.py Aggregates all …
          users.py
          items.py
          auth.py Login, register
    > services/ Business logic
        __init__.py
        user_service.py
        item_service.py
    > repositories/ Data access lay…
        __init__.py
        base.py Generic CRUD re…
        user_repository.py
        item_repository.py
  > alembic/ Database migrat…
      env.py Async migration…
      script.py.mako
    > versions/ Migration files
        .gitkeep
    alembic.ini Alembic configu…
  > tests/
      __init__.py
      conftest.py Fixtures, async…
    > api/
        __init__.py
        test_users.py
        test_items.py
    > services/
        __init__.py
        test_user_service.py
    requirements.txt
    requirements-dev.txt Test dependenci…
    .env.example
    .gitignore
    pyproject.toml Project metadat…
```

Why This Structure?

This structure leverages SQLAlchemy 2.0's native async support with a clean layered architecture. Routes → Services → Repositories pattern keeps business logic testable and database access centralized. Alembic handles schema migrations asynchronously.

Key Directories

app/models/ - SQLAlchemy ORM models (database tables)

app/schemas/ - Pydantic schemas (API contracts)

app/services/ - Business logic, orchestrates repositories

app/repositories/ - Database CRUD operations

app/api/v1/ - Versioned API endpoints

alembic/ - Database migration scripts

Getting Started

1. python -m venv venv && source venv/bin/activate
2. pip install -r requirements.txt -r requirements-dev.txt
3. cp .env.example .env (set DATABASE_URL)
4. alembic upgrade head
5. uvicorn main:app --reload
6. Visit /docs for Swagger UI

Async Database Pattern

Use `async with AsyncSession as session` for all database operations. The `database.py` file provides `get_async_session()` as a FastAPI dependency. Never use synchronous `Session`—SQLAlchemy 2.0 async is fully native, not wrapped.

Layer Responsibilities

Route - HTTP handling, validation, calls service

Service - Business logic, calls repository

Repository - Database queries only

Model - SQLAlchemy table definition

Schema - Pydantic request/response shapes

When To Use This

- Production APIs with PostgreSQL/MySQL
- Projects needing database migrations
- Teams wanting clear separation of concerns
- APIs with complex business logic
- Long-lived projects expecting growth

Alembic Commands

```
# Generate a new migration file
alembic revision --autogenerate -m "Add users table"

# Apply migrations to database
alembic upgrade head

# Rollback last migration
alembic downgrade -1

# View migration history
alembic history

# View current revision
alembic current

# Upgrade to specific revision
alembic upgrade <revision_id>

# Create migration without autogenerate
alembic revision -m "Manual migration"
```

Testing Strategy

conftest.py - Async test client, test database fixtures

tests/api/ - Integration tests against endpoints

tests/services/ - Unit tests with mocked repositories

Trade-offs

More boilerplate - Layers add files but improve maintainability

Learning curve - SQLAlchemy 2.0 async is different from 1.x

Complexity - Overkill for < 5 endpoints (use minimal)