FastAPI with Strawberry GraphQL Project Structure

GraphQL API with Strawberry. Code-first schema, dataloaders, and async resolvers.

#fastapi #python #graphql #strawberry #api #async

PNG PDF Copy Prompt

Project Directory

```
myproject/
    main.py  Mount GraphQL r…
  app/  Application code
    __init__.py
    config.py  Pydantic settin…
  core/
    __init__.py
    database.py  Async engine
    context.py  GraphQL context
  models/  SQLAlchemy mode…
    __init__.py
    base.py
    user.py
    post.py
  graphql/  GraphQL schema
    __init__.py
    schema.py  Root Query/Muta…
  types/  Strawberry types
      __init__.py
      user.py  UserType, UserI…
      post.py
      common.py  Pagination, err…
  resolvers/  Query/Mutation …
      __init__.py
      user.py
      post.py
  dataloaders/  Batch loading f…
      __init__.py
      user_loader.py
      post_loader.py
  services/
    __init__.py
    user_service.py
    post_service.py
tests/
    __init__.py
    conftest.py  Test client, fi…
  graphql/
    __init__.py
    test_queries.py
    test_mutations.py
requirements.txt
requirements-dev.txt
.env.example
.gitignore
pyproject.toml
```

Why This Structure?

Strawberry is a code-first GraphQL library that uses Python type hints. Your schema is defined with dataclasses and decorators—no SDL files to sync. It integrates natively with FastAPI's async support and dependency injection.

Key Directories

app/graphql/types/ - Strawberry types mirror your domain models

app/graphql/resolvers/ - Query/Mutation logic, calls services

app/graphql/dataloaders/ - Batch DB queries to prevent N+1

app/core/context.py - Request context with DB session, loaders

Getting Started

1. `python -m venv venv && source venv/bin/activate`
2. `pip install -r requirements.txt`
3. `cp .env.example .env`
4. `uvicorn main:app --reload`
5. Visit /graphql for GraphiQL playground

Strawberry Type

```python
# app/graphql/types/user.py
@strawberry.type
class UserType:
    id: strawberry.ID
    email: str
    posts: list["PostType"]

    @strawberry.field
    async def posts(self, info: Info) -> list["PostType"]:
        loader = info.context.post_loader
        return await loader.load(self.id)
```

Dataloader Pattern

Dataloaders batch multiple `loader.load(id)` calls into one query. Define in `dataloaders/`, attach to context in `context.py`. Each request gets fresh loader instances to avoid cache issues across requests.

When To Use This

- APIs with complex, nested data requirements
- Mobile apps needing flexible queries
- Teams preferring code-first over SDL-first
- Projects already using Pydantic/dataclasses
- When clients need to specify exact fields

Trade-offs

Learning curve - GraphQL concepts (resolvers, loaders) if new to it

Caching complexity - HTTP caching harder than REST endpoints

Tooling - Need GraphQL clients (Apollo, urql) on frontend

Testing Strategy

tests/graphql/ - Test queries/mutations via test client

conftest.py - GraphQL test client fixture with mocked context

Snapshot tests - Optional: compare query responses to snapshots