Python Minimal API Project Structure

Minimal HTTP API using lightweight tools like Starlette or http.server, no full framework.

#python #api #http #rest #minimal #starlette

PNG PDF Copy Prompt

Project Directory

```
myapi/
    pyproject.toml  Project configu…
    README.md
    .gitignore
    .python-version
    .env.example  Environment tem…
  > src/  Source layout
    > myapi/
        __init__.py
        main.py  App entry point
        routes.py  Route definitio…
      > handlers/  Request handlers
            __init__.py
            health.py
            users.py
      > models/  Data models
            __init__.py
            user.py
        config.py  Environment set…
  > tests/
        __init__.py
        conftest.py
        test_routes.py
```

Why This Structure?

Sometimes FastAPI or Django is overkill. This structure uses Starlette (FastAPI's foundation) or similar lightweight ASGI tools for simple APIs. No magic, no decorators—just explicit request/response handling with minimal dependencies.

Key Directories

src/myapi/main.py - App instance and startup logic

routes.py - URL routing, no decorators

handlers/ - One module per resource

models/ - Pydantic or dataclass models

Starlette App Setup

```python
# src/myapi/main.py
from starlette.applications import Starlette
from starlette.routing import Route
from myapi.handlers import health, users

routes = [
    Route("/health", health.check),
    Route("/users", users.list_users),
    Route("/users/{id}", users.get_user),
]

app = Starlette(routes=routes)
```

Getting Started

1. `uv init myapi`
2. `uv add starlette uvicorn`
3. Define routes in `routes.py`
4. `uv run uvicorn myapi.main:app`

When To Use This

- Simple REST APIs with few endpoints
- Microservices with minimal overhead
- When FastAPI features are unnecessary
- Learning ASGI without abstraction
- Serverless functions

Trade-offs

No auto-docs - No built-in OpenAPI like FastAPI

Manual validation - No automatic request parsing

Less ecosystem - Fewer plugins and extensions

Best Practices

- Use Pydantic for request/response models
- Keep handlers thin, logic in services
- Use dependency injection manually
- Add health check endpoint
- Return proper HTTP status codes