LangChain with FastAPI Project Structure

Production API service with FastAPI, streaming, and async chains.

#langchain #python #fastapi #api #streaming #production

PNG PDF Copy Prompt

Project Directory

```
myproject/
    main.py  FastAPI app ent…
  > app/
      __init__.py
      config.py  Pydantic Settin…
    > api/  API endpoints
        __init__.py
        routes.py  Route definitio…
        deps.py  Dependencies
    > chains/
        __init__.py
        chat.py  Chat chain
        rag.py  RAG chain
    > models/  Pydantic models
        __init__.py
        requests.py
        responses.py
    > services/
        __init__.py
        llm.py  LLM client sing…
        vectorstore.py
  Dockerfile
  requirements.txt
  .env.example
  .gitignore
```

Why This Structure?

Deploy LangChain as a production API with FastAPI. Streaming responses via SSE, async chains for performance, and Pydantic models for request/response validation. The `services/` layer manages LLM and vector store singletons.

Key Directories

app/api/routes.py - FastAPI endpoints for chat/RAG

app/chains/ - Async LangChain chains

app/models/ - Pydantic request/response schemas

app/services/ - Singleton LLM and DB clients

Dockerfile - Production container build

Streaming Endpoint

```python
@router.post("/chat")
async def chat(request: ChatRequest):
    chain = create_chat_chain()
    return StreamingResponse(
        chain.astream(request.message),
        media_type="text/event-stream"
    )
```

When To Use This

- Production LLM API services
- Chat applications with streaming
- RAG systems with HTTP interface
- Microservice architecture

Trade-offs

More boilerplate - API layer adds code

Async required - Must use async chains for perf

Deployment ops - Need container orchestration