Day 3: FastAPI + 실무 Python - 완전 가이드

> **GitHub Repository**: RAG-on-Local-CPU-minimal

> 스크립트 → API 서비스 전환

🎯 학습 목표

Day 3의 핵심은 다음과 같습니다:

> **"RAG 파이프라인을 API 형태로 실행·검증한다"**

이 Day를 마치면:

- ☑ FastAPI로 API 서버 구축
- ☑ dataclass / Pydantic 모델 이해
- ☑ `/query` 엔드포인트 구현
- ☑ Swagger UI 활용
- ☑ 왜 API 서버로 전환했는지 이해

📂 프로젝트 구조

```
Rag_minimal_day3/
├─ data/
│  └─ docs.txt       # RAG 입력 문서
├─ config.py         # 설정 값 분리
├─ models.py         # dataclass + Pydantic 모델
├─ loader.py         # 문서 로딩
├─ pipeline.py       # RAG 파이프라인
└─ main.py           # FastAPI 진입점
```

🤔 왜 이런 구조인가?

1. 왜 FastAPI로 바뀌었나?

Day 2 (스크립트):

```
python app/main.py
# 실행 → 결과 출력 → 종료
```

Day 3 (API 서버):

```
uvicorn main:app --reload
# 서버 계속 실행 → HTTP 요청 대기
```

이유:

1. **실무는 API 형태**
   - 웹/앱에서 호출
   - 다른 서비스와 통합
   - 24/7 실행
2. **테스트 용이**
   - Swagger UI로 즉시 테스트
   - Postman, curl 등 도구 사용
   - 자동화 테스트 가능
3. **확장 가능**
   - 여러 엔드포인트 추가
   - 인증/인가 추가
   - 로깅, 모니터링 추가

2. 왜 dataclass와 Pydantic을 같이 쓰나?

dataclass (내부 로직):

```
@dataclass
class Document:
    id: int
    text: str
```

- Python 기본 기능
- 간단하고 빠름
- 내부 데이터 구조

Pydantic (API 경계):

```
class QueryRequest(BaseModel):
    query: str
```

- FastAPI 필수
- 자동 검증
- JSON ↔ Python 변환

왜 구분?

- API 경계와 내부 로직 분리
- 각자 최적의 도구 사용
- 실무 표준 패턴

3. 왜 models.py를 분리했나?

Before (models.py 없음):

```
# main.py에 모두 정의
class QueryRequest(BaseModel):
    query: str

class QueryResponse(BaseModel):
    answer: str

@dataclass
class Document:
    id: int
    text: str
```

After (models.py 분리):

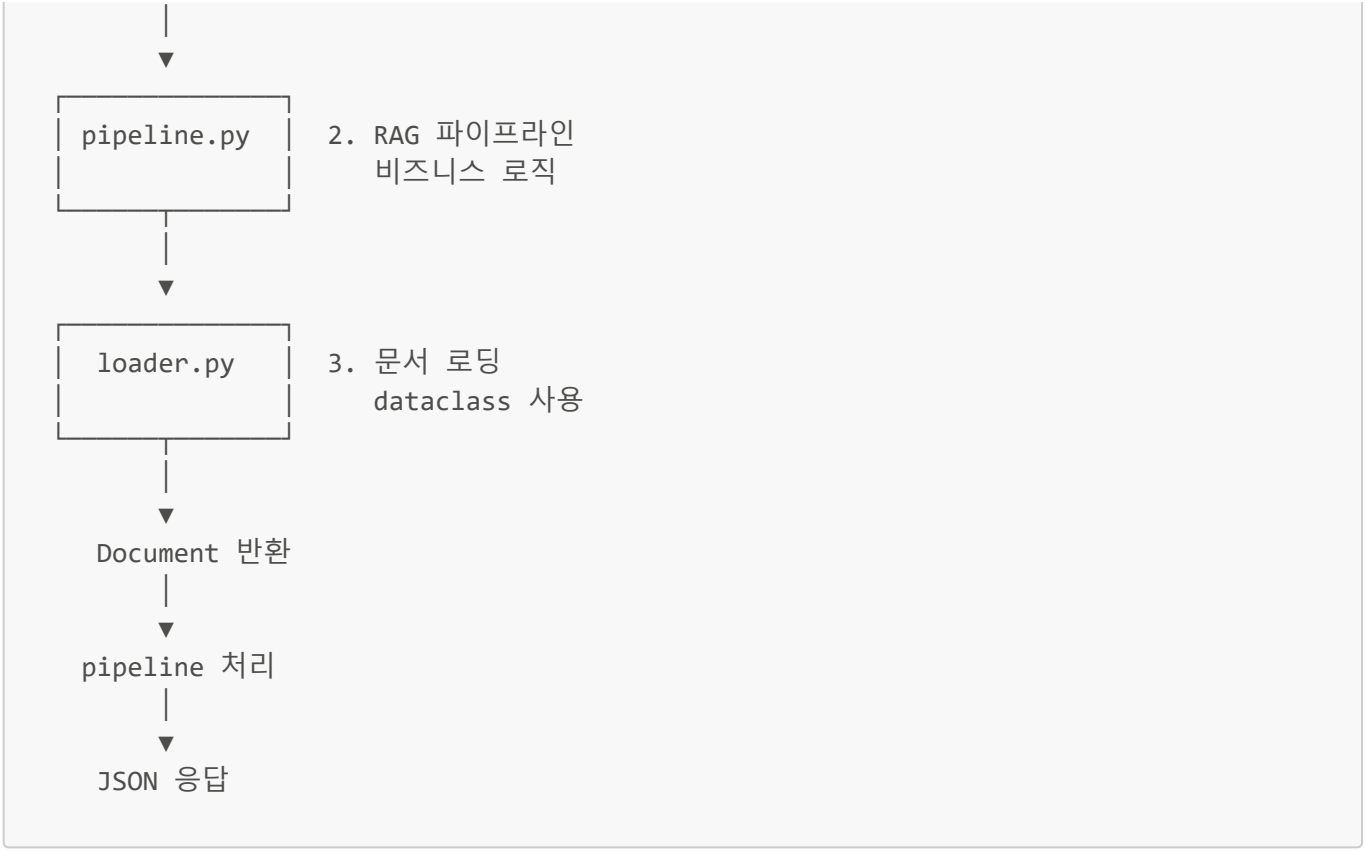
```
# models.py
# 모든 데이터 구조를 한곳에

# main.py
from models import QueryRequest, QueryResponse
```

장점:

- ☑ 데이터 구조 한눈에 파악
- ☑ 재사용 용이
- ☑ 타입 일관성 유지

🔄 실행 흐름도

```
┌─────────────────┐
│ HTTP Request │  POST /query
│   (JSON)       │  {"query": "..."}
└─────────────────┘
        │
        ▼
┌─────────────────┐
│    main.py     │  1. FastAPI 엔드포인트
│                │     Pydantic 검증
└─────────────────┘
        │
```



⚙ 설치 및 실행

1. 패키지 설치

Windows (PowerShell):

```
cd Rag_minimal_day3
pip install fastapi uvicorn
```

macOS/Linux (Bash):

```
cd Rag_minimal_day3
pip install fastapi uvicorn
# 또는
pip3 install fastapi uvicorn
```

2. 서버 실행

```
uvicorn main:app --reload
```

옵션 설명:

- `main:app`: main.py의 app 객체

- `--reload`: 코드 수정 시 자동 재시작

예상 출력:

```
INFO:     Will watch for changes in these directories: [...]
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [...]
INFO:     Started server process [...]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

🔍 API 테스트

1. Health Check

브라우저 접속:

```
http://127.0.0.1:8000/health
```

응답:

```
{"status": "ok"}
```

2. Swagger UI

브라우저 접속:

```
http://127.0.0.1:8000/docs
```

화면 구성:

- GET /health
- POST /query

기능:

- API 문서 자동 생성
- 브라우저에서 직접 테스트
- 요청/응답 스키마 확인

3. Query API 테스트

Swagger UI에서:

1. `/query` POST 클릭
2. "Try it out" 클릭
3. Request body 입력:

```
{
  "query": "RAG란 무엇인가?"
}
```

4. "Execute" 클릭

응답:

```
{
  "answer": "입력 질문: RAG란 무엇인가?, 문서 수: 1"
}
```

curl로 테스트:

```
curl -X POST "http://127.0.0.1:8000/query" \
  -H "Content-Type: application/json" \
  -d '{"query":"RAG란 무엇인가?"}'
```

🐛 디버깅 방법

방법 1: print() 디버깅

main.py에 추가:

```
@app.post("/query", response_model=QueryResponse)
def query_api(req: QueryRequest):
    print("=" * 50)
    print("DEBUG: /query 호출")
    print("=" * 50)
    print(f"1. 받은 요청: {req.query}")
    print(f"2. 요청 타입: {type(req)}")

    answer = pipeline(req.query)

    print(f"3. 파이프라인 결과: {answer}")
    print(f"4. 응답 타입: {type(answer)}")
    print("=" * 50)

    return QueryResponse(answer=answer)
```

터미널 출력:

```
==================================================
DEBUG: /query 호출
==================================================
1. 받은 요청: RAG란 무엇인가?
2. 요청 타입: <class 'models.QueryRequest'>
3. 파이프라인 결과: 입력 질문: RAG란 무엇인가?, 문서 수: 1
4. 응답 타입: <class 'str'>
==================================================
INFO:     127.0.0.1:52341 - "POST /query HTTP/1.1" 200 OK
```

방법 2: VS Code 디버거

1. launch.json 설정 (선택)

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "FastAPI",
      "type": "python",
      "request": "launch",
      "module": "uvicorn",
      "args": ["main:app", "--reload"],
      "jinja": true
    }
  ]
}
```

2. 디버깅:

- `main.py` 브레이크포인트 설정
- **F5** 실행
- Swagger UI에서 API 호출
- VS Code에서 변수 확인

📖 코드 Deep Dive

main.py 한 줄씩

```
# Line 1: FastAPI 임포트
from fastapi import FastAPI

# Line 2-3: 모델, 파이프라인 임포트
from models import QueryRequest, QueryResponse
```

```python
from pipeline import pipeline

# Line 5: FastAPI 앱 생성
app = FastAPI()

# Line 7-9: Health Check 엔드포인트
@app.get("/health")  # GET 요청, /health 경로
def health():
    return {"status": "ok"}  # JSON 반환

# Line 11-13: Query 엔드포인트
@app.post("/query", response_model=QueryResponse)
# POST 요청, /query 경로
# response_model: 응답 형식 지정
def query_api(req: QueryRequest):
    # req: Pydantic이 자동으로 JSON → Python 변환
    answer = pipeline(req.query)
    # QueryResponse로 감싸서 반환
    return QueryResponse(answer=answer)
```

핵심 포인트:

- `@app.post`: 데코레이터로 엔드포인트 등록
- `response_model`: 응답 검증 및 문서화
- `req: QueryRequest`: 자동 JSON 파싱

models.py 한 줄씩

```python
# Line 1-2: 필요한 모듈
from dataclasses import dataclass
from pydantic import BaseModel

# Line 4-7: 내부 데이터 구조 (dataclass)
@dataclass
class Document:
    id: int      # 문서 ID
    text: str    # 문서 내용

# Line 9-11: API 요청 모델 (Pydantic)
class QueryRequest(BaseModel):
    query: str   # 사용자 질문
    # Pydantic이 자동으로:
    # - JSON에서 query 필드 추출
    # - 타입 검증 (str이 아니면 에러)
    # - 누락 시 에러

# Line 13-15: API 응답 모델 (Pydantic)
class QueryResponse(BaseModel):
    answer: str  # RAG 응답
    # FastAPI가 자동으로:
    # - Python → JSON 변환
```

```
    # - 응답 문서화 (Swagger)
    # - 타입 검증
```

핵심 차이:

```
# dataclass: Python 객체
doc = Document(id=1, text="...")
print(doc.text)  # Python에서만 사용

# Pydantic: JSON ⇔ Python
req = QueryRequest(query="...")
req.json()  # JSON으로 변환 가능
```

config.py 한 줄씩

```
# Line 1-3: 설정 상수
MODEL_NAME = "local-llm"  # 사용할 모델 이름
CHUNK_SIZE = 500          # 청크 크기
TOP_K = 3                 # 검색할 문서 수

# 나중에 사용:
# from config import MODEL_NAME, TOP_K
```

왜 config.py?

- 하드코딩 제거
- 환경별 설정 관리
- 한곳에서 수정

loader.py 한 줄씩

```
# Line 1: 모델 임포트
from models import Document

# Line 3: 문서 로드 함수
def load_documents():
    # Line 4-5: 파일 읽기
    with open("data/docs.txt", "r", encoding="utf-8") as f:
        text = f.read()

    # Line 6: Document 객체로 반환
    # dataclass 사용
    return [Document(id=1, text=text)]
```

핵심:

- 파일 읽기 책임만
- Document 객체로 감싸서 반환

pipeline.py 한 줄씩

```
# Line 1: 로더 임포트
from loader import load_documents

# Line 3: 파이프라인 함수
def pipeline(query: str) -> str:
    # Line 4: 문서 로딩
    docs = load_documents()

    # Line 5: 간단한 응답 생성
    # (Day 5부터 실제 검색 추가)
    return f"입력 질문: {query}, 문서 수: {len(docs)}"
```

핵심:

- 아직 LLM 없음
- 흐름과 구조만 구축
- 나중에 확장 예정

💡 학습 효율을 높이는 팁

1. FastAPI 핵심 패턴

엔드포인트 정의:

```
@app.get("/path")       # GET 요청
@app.post("/path")      # POST 요청
@app.put("/path")       # PUT 요청
@app.delete("/path")    # DELETE 요청

def function_name():
    return {"key": "value"}  # JSON 자동 변환
```

경로 파라미터:

```
@app.get("/items/{item_id}")
def read_item(item_id: int):
    return {"item_id": item_id}
```

쿼리 파라미터:

```python
@app.get("/items")
def read_items(skip: int = 0, limit: int = 10):
    return {"skip": skip, "limit": limit}
```

2. Pydantic 검증

자동 검증:

```python
class QueryRequest(BaseModel):
    query: str
    max_results: int = 10  # 기본값

# 요청: {"query": "test"}
# → max_results = 10 (자동)

# 요청: {"max_results": 5}
# → 에러! query 필수

# 요청: {"query": 123}
# → 에러! query는 str
```

Optional 필드:

```python
from typing import Optional

class QueryRequest(BaseModel):
    query: str
    user_id: Optional[int] = None  # 선택적
```

3. 서버 실행 옵션

```bash
# 기본 실행
uvicorn main:app

# 자동 재시작 (개발용)
uvicorn main:app --reload

# 포트 변경
uvicorn main:app --port 8080

# 외부 접속 허용
uvicorn main:app --host 0.0.0.0
```

4. 에러 처리

Pydantic 검증 에러:

```
{
  "detail": [
    {
      "loc": ["body", "query"],
      "msg": "field required",
      "type": "value_error.missing"
    }
  ]
}
```

해결:

- 필수 필드 확인
- 타입 확인
- JSON 형식 확인

🔧 직접 해보기: 실습

실습 1: 새 엔드포인트 추가

main.py에 추가:

```
@app.get("/info")
def info():
    return {
        "version": "1.0.0",
        "description": "RAG Minimal Day 3",
        "endpoints": ["/health", "/query", "/info"]
    }
```

테스트:

```
http://127.0.0.1:8000/info
```

실습 2: 응답 포맷 변경

models.py 수정:

```
class QueryResponse(BaseModel):
    answer: str
    doc_count: int  # 추가
    timestamp: str  # 추가
```

pipeline.py 수정:

```
from datetime import datetime

def pipeline(query: str) -> dict:
    docs = load_documents()
    return {
        "answer": f"질문: {query}",
        "doc_count": len(docs),
        "timestamp": datetime.now().isoformat()
    }
```

main.py 수정:

```
@app.post("/query", response_model=QueryResponse)
def query_api(req: QueryRequest):
    result = pipeline(req.query)
    return QueryResponse(**result)  # 딕셔너리 언패킹
```

실습 3: 디버깅 메시지 추가

전체 파이프라인에 print 추가:

```
# main.py
@app.post("/query", response_model=QueryResponse)
def query_api(req: QueryRequest):
    print(f"[API] 요청 받음: {req.query}")
    answer = pipeline(req.query)
    print(f"[API] 응답 전송: {answer}")
    return QueryResponse(answer=answer)

# pipeline.py
def pipeline(query: str) -> str:
    print(f"[PIPELINE] 시작: {query}")
    docs = load_documents()
    print(f"[PIPELINE] 문서 {len(docs)}개 로딩")
    result = f"입력 질문: {query}, 문서 수: {len(docs)}"
    print(f"[PIPELINE] 완료: {result}")
    return result

# loader.py
def load_documents():
    print(f"[LOADER] 파일 읽기 시작")
    with open("data/docs.txt", "r", encoding="utf-8") as f:
        text = f.read()
    print(f"[LOADER] {len(text)} chars 읽음")
    return [Document(id=1, text=text)]
```

실행 결과:

```
[API] 요청 받음: RAG란 무엇인가?
[PIPELINE] 시작: RAG란 무엇인가?
[LOADER] 파일 읽기 시작
[LOADER] 123 chars 읽음
[PIPELINE] 문서 1개 로딩
[PIPELINE] 완료: 입력 질문: RAG란 무엇인가?, 문서 수: 1
[API] 응답 전송: 입력 질문: RAG란 무엇인가?, 문서 수: 1
```

☑ Day 3 완료 기준

다음을 이해하고 실습했다면 완료입니다:

이해

- ✔ 왜 스크립트에서 API로 전환했는지
- ✔ dataclass vs Pydantic 차이와 용도
- ✔ FastAPI 엔드포인트 작동 방식
- ✔ Swagger UI의 역할

코드

- ✔ main.py의 역할 (API 진입점)
- ✔ models.py의 역할 (데이터 구조)
- ✔ pipeline.py의 역할 (비즈니스 로직)
- ✔ Pydantic 자동 검증 이해

실습

- ✔ uvicorn으로 서버 실행
- ✔ /health 접속 확인
- ✔ Swagger UI에서 API 테스트
- ✔ /query POST 요청/응답 확인
- ✔ 새 엔드포인트 추가해봄

➡ SOON 다음 단계

Day 4는 전자책 전용

Day 5에서는:

```
# Embedding 생성
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('all-MiniLM-L6-v2')
```

```
embeddings = model.encode(docs)

# Vector DB 저장
import faiss
index = faiss.IndexFlatL2(embedding_dim)
index.add(embeddings)

# 검색
query_embedding = model.encode([query])
distances, indices = index.search(query_embedding, k=3)
```

실제 RAG 검색 시작!

💬 학습 후 자가 점검

다음 질문에 답할 수 있나요?

1. **FastAPI를 쓰는 이유는?**
    - 실무 API 표준, Swagger 자동 생성, 타입 체크
2. **dataclass와 Pydantic의 차이는?**
    - dataclass: 내부 로직 / Pydantic: API 경계
3. **@app.post의 역할은?**
    - POST 엔드포인트 등록, 함수를 API로 노출
4. **Swagger UI는 어떻게 접속?**
    - http://127.0.0.1:8000/docs
5. **pipeline.py의 역할은?**
    - 비즈니스 로직, RAG 파이프라인 구현

모두 답할 수 있다면 Day 3 완료! 🎉

📜 라이선스

허용

- ☑ 개인 학습 목적 사용
- ☑ 출처 표시 후 비영리 공유

금지

- ❌ 상업적 이용
- ❌ 내용 수정 및 2차 저작
- ❌ 저작자 허락 없는 재배포

상업적 이용 문의: j4angguiop@gmail.com

⭐ **이 프로젝트가 도움이 되었다면 Star를 눌러주세요!**