AI 바이브/페어 개발 — CLAUDE CODE/CODEX 과정

개인 지식 대시보드 100분 안에 만드는 바이브코딩 사이클

Session 5 · 개인 지식 대시보드 · 바이브코딩 사이클

FastAPI + Postgres + React · Anthropic API · CLAUDE.md · subagent 3 · pytest · Playwright · Docker Compose

강사 박수현 · 🚀 젠아이랩스(GenAI Labs)

바이브코딩 2일 집중 · 프롬프트·컨텍스트·하네스 엔지니어링

PART 1

왜 이 실습 — *todo 가 아닌 이유·요구사항·스택*

> 세션 목표 · todo-app 이 최악의 예제인 이유 · 요구사항 6개와 3축 매핑 · FastAPI + Postgres + React + Anthropic 스택 선택 근거.

세션 목표 — *동작하는 대시보드가 뜬다*

이 세션이 끝나면 참여자 노트북에 **동작하는 개인 지식 대시보드** 가 뜬다. RSS 피드를 긁고, 노트를 인덱싱하고, Anthropic API 로 요약·태깅한 결과를 React UI 에서 확인할 수 있다.

🎯 진짜 목표

완성물이 아니라 **바이브코딩 사이클** 이다.

> 스펙 → 아키텍처 프롬프트 → CLAUDE.md 규약 → subagent 분업 → 반복 리팩토링 → 회귀 방지 테스트 → 배포

이 사이클을 100분 안에 한 번 완주하는 게 목표.

📌 코드가 많이 나오지만 강조점은 언제나 **"어떤 프롬프트로, 어떤 컨텍스트를 얹어, 어떤 하네스에 맡겼는지"** 다.

출처: FastAPI + Postgres + React 실전 | 개인 지식 대시보드

todo-app 이 아닌 이유 — *바이브코딩 관점에서 최악의 예제*

많은 AI 코딩 튜토리얼이 todo-app 을 쓴다. CRUD 4개로 프레임워크 개념이 다 나오기 때문이다. 그런데 **바이브코딩 관점에서 todo-app 은 최악의 예제** 다.

❌ todo-app 의 세 가지 결함

- **요구사항이 확장되지 않는다** — 마감일·라벨이 붙어도 CRUD 반복. 아키텍처 결정 지점 없음.
- **외부 시스템 통합이 없다** — RSS · GitHub · Slack 같은 실전 API 를 안 만진다.
- **LLM 활용 지점이 없다** — todo 를 요약·태깅할 이유가 없다. LLM 호출 자체가 안 나온다.

✅ 개인 지식 대시보드 는 정반대

- **요구사항이 자연히 확장** — RSS → GitHub → 노트 → 알림. 매 단계 아키텍처 결정.
- **외부 API 3개 이상** — RSS · GitHub · Anthropic · (선택) Slack.
- **LLM 이 본질** — 자동 요약·태깅이 핵심 기능. 프롬프트 카드가 실제 코드에 박힌다.

📌 각자 진행 중인 사이드 프로젝트가 이 대시보드와 얼마나 닮았는지 잠깐 확인해보자. 90% 는 비슷한 요구를 이미 갖고 있다. 이 겹침이 실습을 자기 리포로 옮기는 근거다.

출처: FastAPI + Postgres + React 실전 | 개인 지식 대시보드

요구사항 6개 · 3축의 실전 발생 지점

GOALS.md 의 여섯 개 기능 요구를 3축 (프롬프트·컨텍스트·하네스) 과 매핑한다.

#	기능	프롬프트 축	컨텍스트 축	하네스 축
1	RSS 피드 등록·수집·저장	스펙→라우트 프롬프트	`CLAUDE.md` 스타일 규약	pre-commit 훅 lint
2	GitHub 활동 동기화	외부 API 스키마 프롬프트	GitHub 인증 규약 문서화	`.env` 관리 훅
3	마크다운 노트 업로드·인덱싱	파일 파싱 프롬프트	노트 폴더 경로 규약	업로드 크기 제한 훅
4	Anthropic API 요약·태깅	요약 프롬프트 카드	태그 사전 컨텍스트	실패 재시도 정책
5	대시보드 UI (검색·필터)	React 컴포넌트 프롬프트	디자인 토큰 정의	Playwright 테스트 훅
6	(선택) Slack 알림	Webhook 설계 프롬프트	Slack 채널 규약	rate-limit 훅

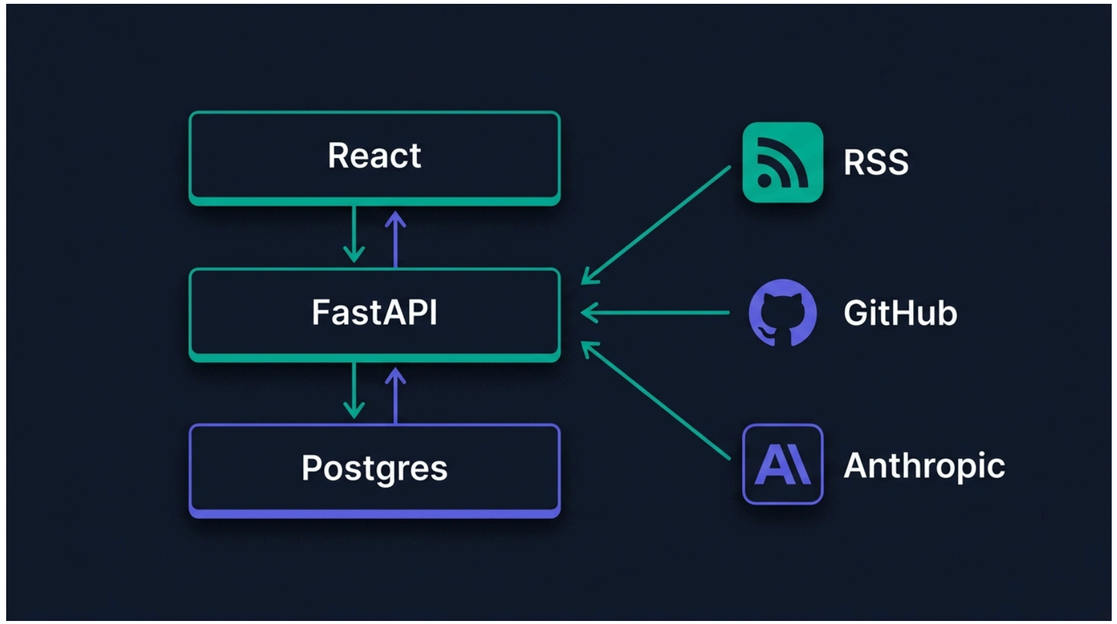
📌 **핵심 관찰.** 어느 한 요구도 세 축 중 하나만으로 안 풀린다. 프롬프트만 써도 안 되고, CLAUDE.md 만 써도 안 되고, hook 만 걸어도 안 된다. **세 축이 동시에 걸리는 지점이 실전이다.**

출처: FastAPI + Postgres + React 실전 | 개인 지식 대시보드

스택 개요 — *FastAPI · Postgres · React · Anthropic*

스택 선택 자체가 이 실습의 첫 아키텍처 결정이다.

🧱 스택 다이어그램



📋 계층별 선택 · 근거

계층	선택	이유
백엔드	FastAPI 0.136+	파이썬 · async · Pydantic v2
DB	PostgreSQL 16	관계형 · JSONB · FTS
프론트	React 18 + Vite	표준 · 빌드 초고속
LLM	Anthropic Python SDK	Claude Code 와 동일 벤더
배포	Docker Compose	로컬 단일 명령

📌 🐍 **파이썬 한 언어로 통일** — 백엔드는 파이썬, 프론트만 JS. 컨텍스트 스위칭이 적다. 🖥️ **로컬 실습 한정** — 클라우드 배포·인증·CDN 은 이 세션 밖, 확장 아이디어에서 언급.

PART 2

요구사항 설계 · CLAUDE.md · subagent

> 프롬프트 카드 A (스펙→아키텍처) · 파일 트리 확정 · CLAUDE.md 세팅 · subagent 3 (backend-dev · frontend-dev · db-migrator) 분업.

프롬프트 카드 A — *스펙 → 아키텍처*

빈 폴더에서 시작. 참여자는 강사가 만든 뼈대 리포 `vibecoding-dashboard-skel` 을 clone 한다. 안에는 `README.md` 와 `SPEC.md` 두 개뿐이다. 여기서 첫 프롬프트를 던진다.

```
@SPEC.md

이 요구사항을 FastAPI + PostgreSQL + React(Vite) 로 구현할 계획을 세워줘.
다음 순서로 답변해줘.

1. 파일 트리 (backend/, frontend/, docker/, tests/)
2. 데이터 모델 (SQLAlchemy 클래스명·컬럼만, 코드 X)
3. API 엔드포인트 목록 (path · method · 요청/응답 요약만, 코드 X)
4. 3개의 잠재적 위험 (스코프 · 성능 · 보안)

코드는 이번 턴에 쓰지 마. 다음 턴에 파일 트리부터 만들자.
```

📌 **이 프롬프트의 세 가지 장치.** - `@SPEC.md` — 컨텍스트를 명시적으로 로드. Claude 가 "요구를 다시 물어보는" 왕복을 없앤다. - **"코드는 이번 턴에 쓰지 마"** — 설계와 구현을 강제로 분리. Plan 단계에서 코드가 쏟아지면 되돌리기가 어렵다. - **"3개의 잠재적 위험"** — 낙관적 계획을 강제로 견제. 이후 회귀 방지 카드로 이어진다.

파일 트리 확정 — *표준 뼈대*

프롬프트 A 응답을 검토해 파일 트리를 확정한다. 강사가 사전에 준비한 표준 트리는 아래와 같다.

```
vibecoding-dashboard/
├── backend/
│   ├── app/
│   │   ├── main.py              # FastAPI 앱 진입
│   │   ├── config.py            # 설정 (환경변수 로드)
│   │   ├── db.py                # SQLAlchemy 엔진·세션
│   │   ├── models.py            # ORM 모델
│   │   ├── schemas.py           # Pydantic 스키마
│   │   ├── routes/              # feeds · github · notes · digest
│   │   └── services/            # rss · github_sync · summarizer · tagger
│   ├── tests/                   # pytest · conftest.py
│   ├── pyproject.toml
│   └── alembic/                 # DB 마이그레이션
├── frontend/
│   ├── src/                     # App · api.ts · components/
│   ├── package.json
│   └── vite.config.ts
├── docker/                       # docker-compose.yml + Dockerfile 2종
├── CLAUDE.md
├── SPEC.md
└── README.md
```

📌 응답 검토 후 파일 트리에 이견이 있으면 이 시점에서 조정한다. 이 조정 대화가 뒤이은 구현 턴의 컨텍스트에 남는다.

뼈대 생성 — *acceptEdits 로 넘어가는 지점*

파일 트리가 확정되면 뼈대 파일을 한 번에 만든다.

```
확정된 파일 트리를 만들어줘. 각 파일은 빈 셸(주석 한 줄 + 최소 import) 로 두고,
pyproject.toml·package.json·docker-compose.yml 은 실제로 동작하는 최소 구성으로 채워줘.
```

📦 30~40 개 파일이 한 번에

Claude 가 파일을 다수 만든다. Default 모드라면 승인 요청이 수십 건씩 쌓인다.

⌨️ Shift+Tab 한 번

이 지점부터 acceptEdits 로 넘긴다. 편집은 자동으로, 셸 명령만 승인. 반복 승인 부담이 사라진다.

📌 뼈대 만들기 직전이 acceptEdits 전환의 정확한 타이밍이다. 여기서 넘기면 이후 반복 사이클이 매끄럽다. 셸 실행은 여전히 승인이 걸리니 위험도가 낮다.

CLAUDE.md — *프로젝트 규약 (전반부)*

뼈대가 만들어졌어도 구현 프롬프트를 바로 이어가지 않는다. **먼저 CLAUDE.md 를 채운다.** 이걸 채우지 않으면 Claude 는 매 세션 스타일·테스트·라우트 규약을 다시 물어본다.

```
# Vibecoding Dashboard — 프로젝트 규약

## 스택
- Backend: FastAPI 0.136+, SQLAlchemy 2.x, Alembic, Pydantic v2
- Frontend: React 18, Vite, TypeScript
- DB: PostgreSQL 16 (Docker)
- LLM: Anthropic Python SDK (`anthropic`)

## 스타일
- Python: black + ruff (line-length 100). `poetry run lint` 로 검증.
- TypeScript: prettier + eslint. `pnpm lint` 로 검증.
- 모든 함수는 타입 힌트 필수. `mypy --strict` 통과 목표.
- 커밋 메시지: `feat(feeds): add RSS ingest route` 형식.

## 테스트
- 백엔드: pytest, `poetry run pytest -x` 가 통과해야 커밋 가능.
- 프론트: Vitest + Playwright. `pnpm test` 로 통합.
- 새 라우트 추가 시 라우트별 테스트 1개는 필수.
```

📌 **CLAUDE.md 는 매 세션마다 반복되는 지시를 담아 두는 파일**이다. 참여자가 자기 팀 규약을 자기 CLAUDE.md 에 옮기는 감각을 여기서 익힌다.

CLAUDE.md — *프로젝트 규약 (후반부)*

```
## API 규약
- 모든 라우트는 `/api/v1/` prefix.
- 응답은 항상 `{"data": ..., "error": null}` 또는 `{"data": null, "error": {...}}` 봉투.
- 200/201 외 상태 코드는 `HTTPException` 으로만 발생.
- 시간은 UTC ISO 8601, DB 컬럼은 `TIMESTAMPTZ`.

## 시크릿
- `.env` 는 절대 커밋 금지. `.env.example` 만 유지.
- Anthropic API 키는 `ANTHROPIC_API_KEY` 환경변수로만.

## Claude 지시
- 새 파일 만들기 전에 항상 파일 트리와 대조.
- 모델은 Sonnet 4.6 기본, 어려운 리팩토링만 Opus 4.7.
- 대량 변경은 항상 커밋을 나눠서 (한 커밋 = 한 관심사).
```

📌 **응답 봉투 규약이 이 안에 박히면 프론트·백엔드가 자동으로 규약을 따른다.** 참여자에게 이 시점에서 자기 팀의 응답 스키마·시크릿 규칙을 대입해볼 시간을 준다.

subagent 분업 — *backend-dev 정의*

프로젝트가 크면 한 세션에서 백엔드·프론트·DB 를 모두 다루기 어렵다. Claude Code 의 **subagent** 를 세 개 정의해 역할을 나눈다. 세 subagent 는 리포의 `.claude/agents/` 안에 마크다운으로 정의한다.

`.claude/agents/backend-dev.md`

```
---
name: backend-dev
description: FastAPI 백엔드 라우트·서비스·모델 담당. DB 마이그레이션·프론트 변경은 금지.
tools: [Read, Edit, Write, Bash]
---

너는 이 리포의 FastAPI 백엔드 개발자다.
다음 원칙을 지킨다.

- `backend/` 디렉토리 안에서만 편집. `frontend/` · `alembic/` 은 만지지 않는다.
- 모든 라우트는 CLAUDE.md 의 API 규약을 따른다.
- 새 라우트에는 pytest 케이스 최소 1개.
- 의심스러운 스키마 변경은 db-migrator 에게 위임한다고 답한다.
```

- `name` = subagent 식별자 (`/agent backend-dev` 로 호출).
- `description` = 언제 이 subagent 로 넘길지 판단하는 근거.
- `tools` = subagent 가 쓸 수 있는 도구 목록 (여기서 벗어난 도구는 호출 불가).

📌 **금지 조항이 핵심이다.** `frontend/` 만지지 마라 · 스키마 변경은 db-migrator 에게 위임하라. subagent 정의는 "무엇을 하나" 보다 **"무엇을 하지 마라"** 가 더 중요하다.

출처: FastAPI + Postgres + React 실전 | 개인 지식 대시보드

subagent 3종 — *역할·금지·호출*

이름	담당	금지
backend-dev	FastAPI 라우트·서비스·pytest	`frontend/` · `alembic/` 편집
frontend-dev	React 컴포넌트·Playwright 테스트	`backend/` 편집
db-migrator	Alembic 마이그레이션·스키마	라우트·UI 편집

호출 방법.

```
> Task(subagent_type="backend-dev", prompt="feeds 라우트에 GET /api/v1/feeds/{id}/refresh 추가")
```

🧠 왜 나누는가

한 세션에서 백엔드·프론트를 왔다갔다 하면 컨텍스트 창이 금방 차고, Claude 가 서로의 파일을 잘못 만진다.

🎯 격리의 효과

역할별로 컨텍스트를 격리 하면 응답 품질도 오른다. 각 subagent 는 자기 담당 파일만 컨텍스트에 얹는다.

PART 3

반복 개발 사이클 — *FastAPI·DB·RSS·LLM·React*

> FastAPI 뼈대 · DB 모델 2 슬라이드 · RSS 수집기 · Anthropic 요약·태깅 · React 대시보드 · CLAUDE.md 갱신 ·hook.

FastAPI 뼈대 ① — *config.py*

첫 실제 코드는 FastAPI 진입 · DB 세션 · 헬스체크. 강사가 이 코드를 한 번에 만드는 프롬프트를 던진다.

```
backend/app/main.py · db.py · config.py 를 채워줘. 조건.
- FastAPI 앱은 CORS · lifespan 훅으로 DB 엔진 초기화.
- SQLAlchemy 2.x 비동기 세션.
- 환경변수는 pydantic-settings 로.
- GET /api/v1/health 헬스체크 라우트 하나.
- 코드만. 설명은 필요 없음.
```

`backend/app/config.py`

```
from pydantic_settings import BaseSettings, SettingsConfigDict


class Settings(BaseSettings):
    database_url: str = "postgresql+asyncpg://dash:dash@db:5432/dash"
    anthropic_api_key: str = ""
    cors_origins: list[str] = ["http://localhost:5173"]
    model_config = SettingsConfigDict(env_file=".env", env_file_encoding="utf-8")


settings = Settings()
```

📌 pydantic-settings 는 **환경변수 > `.env` > 기본값** 순으로 병합한다 — 컨테이너·CI 주입값이 최우선, `.env` 는 로컬용(커밋 금지), 기본값은 안전망. 배포·로컬·CI 가 같은 코드로 돈다.

FastAPI 뼈대 ② — *db.py · 비동기 세션*

`backend/app/db.py`

```
from sqlalchemy.ext.asyncio import (
    AsyncSession,
    async_sessionmaker,
    create_async_engine,
)

from .config import settings

engine = create_async_engine(settings.database_url, echo=False, pool_pre_ping=True)
SessionLocal = async_sessionmaker(engine, class_=AsyncSession, expire_on_commit=False)
```

🔌 async 엔진

`create_async_engine` + `asyncpg` 드라이버.
`pool_pre_ping=True` 는 커넥션을 꺼내기 직전 가벼운 `SELECT 1` 로 살아 있는지 확인 — DB 재시작·네트워크 순단 후 첫 요청이 죽는 사고를 막는다.

🧵 FastAPI 의존성

`get_session` 이 `async def` 안에서 `yield` 하는 **async generator** 다. `yield` 앞에서 세션을 열고, 라우트 응답이 끝난 뒤 `async with` 가 자동으로 세션을 닫는다 — 요청 단위 트랜잭션이 하네스로 관리된다.

📌 `expire_on_commit=False` 는 SQLAlchemy 2.x async 세션의 표준. commit 후에도 객체 속성 접근이 가능해진다.

FastAPI 뼈대 ③ — *main.py · 헬스체크*

`backend/app/main.py`

```
from contextlib import asynccontextmanager
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from .config import settings
from .db import engine

@asynccontextmanager
async def lifespan(app: FastAPI):
    yield
    await engine.dispose()  # shutdown: 커넥션 풀 정리

app = FastAPI(title="Vibecoding Dashboard", lifespan=lifespan)
app.add_middleware(
    CORSMiddleware, allow_origins=settings.cors_origins,
    allow_credentials=True, allow_methods=["*"], allow_headers=["*"],
)

@app.get("/api/v1/health")
async def health():
```

- `lifespan` — `yield` **앞** = startup(엔진은 import 시점에 이미 만들어져 비움), **뒤** = shutdown(`engine.dispose()`).
- `allow_credentials=True` — 쿠키· `Authorization` 을 크로스오리진에 허용. 이 조합엔 `["*"]` 가 금지라 `settings.cors_origins` 화이트리스트 명시.

📌 **관찰.** 세 파일이 CLAUDE.md 의 API 봉투 규약 (`{"data": ..., "error": null}`) 을 지켰다. 규약을 미리 넣어둔 효과다.

데이터 모델 ① — *FeedSource · db-migrator 위임*

헬스체크가 뜨면 데이터 모델을 얹는다. **db-migrator subagent 에게 위임.**

```
Task(subagent_type="db-migrator", prompt="""
SPEC.md 의 4개 엔티티(FeedSource · FeedItem · Note · GithubEvent) 를 SQLAlchemy 모델 + alembic 초기 마이그레이션으로. 조건.
- 모든 테이블 공통: id(uuid) · created_at · updated_at (TIMESTAMPTZ).
- FeedItem 은 summary · tags(jsonb) 컬럼, title+summary 에 GIN 인덱스.
""")
```

`backend/app/models.py` — **FeedSource**

```
import uuid
from datetime import datetime
from sqlalchemy import String, func
from sqlalchemy.dialects.postgresql import UUID
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column, relationship


class Base(DeclarativeBase):
    pass


class FeedSource(Base):
    __tablename__ = "feed_sources"
    id: Mapped[uuid.UUID] = mapped_column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    url: Mapped[str] = mapped_column(String(500), unique=True)
    title: Mapped[str] = mapped_column(String(200))
    created_at: Mapped[datetime] = mapped_column(server_default=func.now())
```

데이터 모델 ② — *FeedItem · 요약·태그 컬럼*

`backend/app/models.py` — **FeedItem**

```
from sqlalchemy import JSON, ForeignKey, String, Text, func


class FeedItem(Base):
    __tablename__ = "feed_items"
    id: Mapped[uuid.UUID] = mapped_column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    source_id: Mapped[uuid.UUID] = mapped_column(ForeignKey("feed_sources.id"))
    title: Mapped[str] = mapped_column(String(300))
    url: Mapped[str] = mapped_column(String(500))
    raw_content: Mapped[str] = mapped_column(Text)
    summary: Mapped[str | None] = mapped_column(Text)
    tags: Mapped[dict] = mapped_column(JSON, default=dict)
    published_at: Mapped[datetime] = mapped_column()
    created_at: Mapped[datetime] = mapped_column(server_default=func.now())
    updated_at: Mapped[datetime] = mapped_column(server_default=func.now(), onupdate=func.now())
    source: Mapped[FeedSource] = relationship(back_populates="items")
```

마이그레이션 실행.

```
docker compose exec backend alembic upgrade head
```

📌 SQL 을 직접 쓰지 않는다. 스키마 결정은 프롬프트로, 실행은 하네스로. **Alembic 이 만드는 파일을 통제 밖에 두면 안 된다** — 참여자가 마이그레이션 파일을 반드시 열어본다.

RSS 수집기 — *feedparser · async 래퍼*

데이터 모델이 준비되면 RSS 수집기부터 붙인다. **backend-dev 호출.**

```
Task(subagent_type="backend-dev", prompt="""
services/rss.py: async fetch_feed(url)→list[dict] (feedparser).
routes/feeds.py: POST /feeds(등록) · POST /feeds/{id}/refresh(fetch→FeedItem) · pytest 1개.
""")
```

`backend/app/services/rss.py` — 핵심부

```
import asyncio, feedparser

async def fetch_feed(url: str) → list[dict]:
    loop = asyncio.get_running_loop()
    parsed = await loop.run_in_executor(None, feedparser.parse, url)  # sync 파서를 스레드로
    return [
        {"title": e.get("title", "")[:300], "url": e.get("link", ""),
         "raw_content": e.get("summary", ""), "published_at": _published_at(e)}
        for e in parsed.entries
    ]
```

📎 `_published_at()` (published_parsed·updated_parsed → UTC datetime, 없으면 now) 전체 구현은 뼈대 리포 `backend/app/services/rss.py` 참고.

RSS 라우트 ① — *등록 엔드포인트*

`backend/app/routes/feeds.py` — imports · router · `POST /api/v1/feeds`

```
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy import select
from sqlalchemy.ext.asyncio import AsyncSession

from ..db import get_session
from ..models import FeedItem, FeedSource
from ..schemas import FeedSourceIn, FeedSourceOut
from ..services.rss import fetch_feed

router = APIRouter(prefix="/api/v1/feeds", tags=["feeds"])


@router.post("", response_model=FeedSourceOut, status_code=201)
async def register(payload: FeedSourceIn, session: AsyncSession = Depends(get_session)):
    src = FeedSource(url=str(payload.url), title=payload.title)
    session.add(src)
    await session.commit()
    await session.refresh(src)
    return {"data": src, "error": None}
```

📌 **관찰.** `response_model` 이 스키마를 강제하는 동시에, 반환 dict 에 `{data, error}` 봉투를 손으로 씌웠다. **이 불일치가 다음 리팩토링의 표적** — Part 4.3 프롬프트 카드 C 에서 처리한다.

RSS 라우트 ② — *refresh 엔드포인트*

`backend/app/routes/feeds.py` (이어서) — `POST /api/v1/feeds/{id}/refresh`

```
@router.post("/{source_id}/refresh")
async def refresh(source_id: str, session: AsyncSession = Depends(get_session)):
    src = await session.get(FeedSource, source_id)
    if not src:
        raise HTTPException(status_code=404, detail="feed not found")
    fetched = await fetch_feed(src.url)
    for item in fetched:
        session.add(FeedItem(source_id=src.id, **item))
    await session.commit()
    return {"data": {"ingested": len(fetched)}, "error": None}
```

📌 **관찰.** refresh 는 (a) source_id 검증 → (b) `fetch_feed` 호출 → (c) 결과를 벌크 insert → (d) 개수 반환. 네 단계가 명확하게 순서대로 드러나 있어 나중에 트랜잭션·retry 를 얹기 좋다.

Anthropic API 요약·태깅 ① — *SDK·프롬프트 상수*

LLM 호출이 처음 들어오는 지점. `anthropic` SDK 를 쓴다. 프롬프트 자체가 **모듈 상수로 코드에 박힌다.**

`backend/app/services/summarizer.py` — imports · client · SUMMARY_PROMPT

```
import asyncio
import json
import logging

from anthropic import AsyncAnthropic, APIError

from ..config import settings

logger = logging.getLogger(__name__)

client = AsyncAnthropic(api_key=settings.anthropic_api_key)

SUMMARY_PROMPT = """너는 개인 지식 큐레이터다.
주어진 글의 제목과 본문을 읽고 다음 JSON 을 반환한다.

{
  "summary": "3문장 이내 한국어 요약",
  "tags": ["소문자 영어 슬러그", "최대 5개"]
}

JSON 외 어떤 텍스트도 반환하지 마."""
```

📌 **관찰.** `SUMMARY_PROMPT` 가 모듈 상단 상수로 자리 잡았다 — **프롬프트 카드의 코드화**. 팀원 누구든 이 상수만 고치면 요약 스타일이 바뀐다.

Anthropic API 요약·태깅 ② — *호출·재시도·실패 처리*

`backend/app/services/summarizer.py` (이어서) — `summarize_and_tag`

```
async def summarize_and_tag(title: str, body: str) → dict:
    user_msg = f"제목: {title}\n\n본문:\n{body[:4000]}"

    for attempt in range(3):
        try:
            resp = await client.messages.create(
                model="claude-haiku-4-5",
                max_tokens=512,
                system=SUMMARY_PROMPT,
                messages=[{"role": "user", "content": user_msg}],
            )
            text = resp.content[0].text.strip()
            return json.loads(text)
        except (APIError, json.JSONDecodeError) as exc:
            logger.warning("summarize attempt %d failed: %s", attempt + 1, exc)
            await asyncio.sleep(2**attempt)

    return {"summary": "", "tags": []}
```

📌 **관찰.** JSON 강제 · 3회 재시도 · exponential backoff · 실패 시 빈 값 반환 — 네 장치가 LLM 호출의 표준. `body[:4000]` 슬라이싱으로 토큰 초과도 사전 차단.

요약·태깅 — *pytest 케이스*

`backend/tests/test_summarizer.py`

```
import pytest

from app.services.summarizer import summarize_and_tag


@pytest.mark.asyncio
@pytest.mark.integration
async def test_summarize_and_tag_shape():
    result = await summarize_and_tag(
        title="Anthropic releases Claude Opus 4.7",
        body="Anthropic announced ...",
    )
    assert isinstance(result, dict)
    assert "summary" in result
    assert "tags" in result and isinstance(result["tags"], list)
```

📌 `@pytest.mark.integration` 이 붙어 있어 기본 실행에서는 스킵된다. 실제 API 호출 검증을 원할 때만 마커를 켠다.

React 대시보드 ① — *api.ts · fetch 래퍼*

백엔드가 요약을 반환할 수 있게 되면 프론트를 붙인다. **frontend-dev 호출.**

```
Task(subagent_type="frontend-dev", prompt="""
frontend/src 컴포넌트.
- App.tsx: 상단 SearchBar + 하단 DigestList
- DigestList.tsx: /api/v1/digest/today 를 카드 그리드로
- SearchBar.tsx: debounced 300ms 로 /api/v1/digest/search?q=... 호출
- api.ts: fetch 래퍼 (base = .env VITE_API_BASE)
스타일 최소, Tailwind 대신 모듈 CSS.
```

`frontend/src/api.ts`

```
const BASE = import.meta.env.VITE_API_BASE ?? "http://localhost:8000";

export type DigestItem = {
  id: string; title: string; url: string;
  summary: string; tags: string[]; published_at: string;
};

async function get<T>(path: string): Promise<T> {
  const res = await fetch(`${BASE}${path}`);
  if (!res.ok) throw new Error(`${res.status}`);
  return (await res.json()).data as T;
}

export const api = {
  today: () => get<DigestItem[]>("/api/v1/digest/today"),
```

출처: FastAPI + Postgres + React 실전 | 개인 지식 대시보드

React 대시보드 ② — *DigestList (props·state·hooks)*

`frontend/src/components/DigestList.tsx` — imports · props · state · effect

```
import { useEffect, useState } from "react";
import { api, DigestItem } from "../api";
import styles from "./DigestList.module.css";

export function DigestList({ query }: { query: string }) {
  const [items, setItems] = useState<DigestItem[]>([]);

  useEffect(() => {
    const load = query ? api.search(query) : api.today();
    load.then(setItems).catch(() => setItems([]));
  }, [query]);

  // return ( ... ) — 이어짐
}
```

📌 **관찰.** `query` prop 이 바뀌면 effect 가 재실행되어 검색·today API 를 갈아탄다. 실패 시 빈 배열로 되돌리는 `catch` 로 UI 가 죽지 않는다.

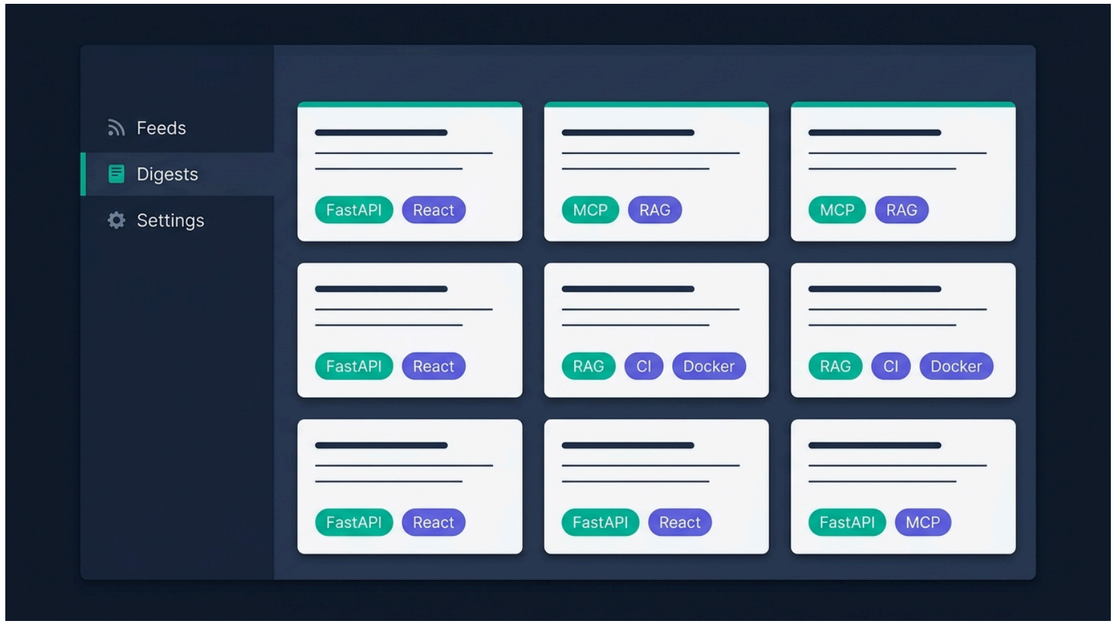
React 대시보드 ③ — *DigestList (JSX 렌더)*

`frontend/src/components/DigestList.tsx` (이어서) — return JSX 와 렌더 결과 대조

💻 return JSX

```
  return (
    <div className={styles.grid}>
      {items.map((item) => (
        <article key={item.id} className={styles.card}>
          <h3>
            <a href={item.url} target="_blank" rel="noreferrer">
              {item.title}
            </a>
          </h3>
          <p>{item.summary}</p>
          <div className={styles.tags}>
            {item.tags.map((t) => (
              <span key={t}>{t}</span>
            ))}
          </div>
        </article>
      ))}
    </div>
  );
}
```

🖥️ 렌더된 화면



출처: FastAPI + Postgres + React 실전 | 개인 지식 대시보드

CLAUDE.md 갱신 — *실습 진행 중 특이사항*

지금까지 만든 것이 세 번의 반복을 거쳤다. FastAPI 뼈대 → RSS 라우트 → 요약 서비스 → React 대시보드. **매 반복마다 결정이 CLAUDE.md 에 추가돼야 한다.**

CLAUDE.md 에 추가할 세 줄.

```
## 이 리포 특이사항 (실습 진행 중 갱신)

- 요약 프롬프트는 `backend/app/services/summarizer.py` 의 SUMMARY_PROMPT 상수만 편집.
  프롬프트 변경은 별도 커밋으로.
- RSS refresh 는 동기. 다음 라운드에 background task 로 옮길 예정.
- 프론트 컴포넌트는 함수형만. class 컴포넌트 금지.
```

📌 CLAUDE.md 는 처음 세팅한 뒤 그대로 두는 게 아니다. **반복이 결정을 낳고, 결정이 CLAUDE.md 를 갱신한다** — 결정 하나에 한 줄, 세션이 끝나면 5~10 줄. 다시 세션을 열었을 때 Claude 가 "왜 이렇게 되어 있나" 물어볼 필요가 없어진다. 이 흐름이 컨텍스트 축의 실체다.

hook 3개 — `.claude/settings.local.json`

hook 은 리포 `.claude/settings.local.json` 에 얹는다. 규약을 문서에서 코드로 옮기는 지점.

```
{
  "hooks": {
    "PreToolUse": [
      { "matcher": "Write|Edit", "hooks": [
        { "type": "command", "command": "test ! -f .env || echo 'WARNING: .env exists'" }
      ]}
    ],
    "PostToolUse": [
      { "matcher": "Edit", "hooks": [
        { "type": "command", "command": "cd backend && poetry run ruff check --fix . 2>&1 | tail -20" }
      ]}
    ]
  }
}
```

🔐 PreToolUse

`.env` 존재 경고. 시크릿 커밋을 사전에 잡는다.

🧹 PostToolUse

편집 후 ruff --fix 자동 실행. 규약이 코드에 강제된다.

🚀 SessionStart (참고)

`git status` 를 세션 시작 때 자동으로 찍어 컨텍스트에 얹으면 유용.

PART 4

회귀 방지 · 리팩토링 — *프롬프트 카드 B·C*

> pytest 첫 초록불 · 프롬프트 카드 B (회귀 방지) · 프롬프트 카드 C (반복 리팩토링) · Playwright E2E · Docker Compose 배포.

pytest 도입 — *초록불 만들기*

코드가 쌓였다. 리팩토링에 들어가려면 **테스트가 초록불** 이어야 한다.

`backend/pyproject.toml` — pytest 설정

```
[tool.pytest.ini_options]
asyncio_mode = "auto"
markers = ["integration: 외부 API 호출 필요 (기본 skip)"]
addopts = "-x -ra --strict-markers"
```

`backend/tests/conftest.py` — 인메모리 DB fixture (핵심)

```
import pytest_asyncio
from sqlalchemy.ext.asyncio import AsyncSession, create_async_engine, async_sessionmaker
from app.models import Base

@pytest_asyncio.fixture
async def db_session() → AsyncSession:
    engine = create_async_engine("sqlite+aiosqlite:///:memory:")
    async with engine.begin() as conn:
        await conn.run_sync(Base.metadata.create_all)
    async with async_sessionmaker(engine, expire_on_commit=False)() as session:
        yield session
```

- `sqlite+aiosqlite:///:memory:` — 인메모리 async SQLite. fixture 마다 새로 만들고 폐기해 케이스 간 오염 차단.
- `client` fixture 는 httpx `ASGITransport(app=app)` 로 uvicorn 없이 ASGI 앱을 직접 호출 — 포트 충돌·기동 대기 없음.

프롬프트 카드 B — *회귀 방지*

초록불이 잡히면 **회귀 방지 카드** 를 던진다. 이 카드는 리팩토링 프롬프트 앞에 항상 붙는다.

```
지금부터 리팩토링을 진행할 텐데, 다음 원칙을 지킨다.

1. 변경 전 `poetry run pytest -x` 결과를 먼저 실행해 초록불 확인.
2. 변경 후 같은 명령을 실행해 여전히 초록불인지 확인.
3. 만약 실패한다면, 실패한 테스트 이름 · 실패 원인 · 수정 diff 를 표로 보고.
4. 이 원칙은 이 세션 내 모든 리팩토링에 적용.
```

📌 **이 카드의 세 장치.** - **테스트 명령을 명시적으로 지정** — Claude 가 자기 재량으로 다른 명령을 쓰지 않게. - **초록불 → 리팩토링 → 초록불** 의 3 스텝 강제. - **실패 시 표 보고** — 자유 서술로 두면 원인을 감춘다.

🔄 세션 안에서 반복 사용

이 카드는 매 리팩토링 프롬프트 앞에 한 번씩 붙인다.

📌 CLAUDE.md 에 옮김

이 원칙 자체를 CLAUDE.md 에 옮기면 매 세션 던지지 않아도 된다.

프롬프트 카드 C — *반복 리팩토링*

이제 실제 리팩토링을 시작한다. 첫 표적은 **응답 봉투 반복 코드**. `{"data": ..., "error": None}` 을 라우트마다 손으로 쓰는 게 냄새다.

```
backend/app/routes/feeds.py 의 응답 봉투(`{"data": ..., "error": None}`) 가
라우트마다 반복된다. 이 반복을 제거하는 3가지 접근을 표로 비교해줘.

각 접근에 다음 열.
- 접근명
- 구현 위치 (미들웨어 · 의존성 · 유틸 함수)
- 장점 2줄
- 단점 2줄
- 이 리포에 적합한 정도 (상/중/하)

표만 먼저. 코드는 다음 턴에.
```

📌 **이 카드의 핵심.** 곧바로 "고쳐줘" 하지 않고 **3가지 접근 비교표** 를 먼저 받는다. 아키텍처 결정을 강사·참여자가 직접 내리고, Claude 는 결정된 접근만 구현.

프롬프트 카드 C — *응답 표와 선택*

접근	위치	장점	단점	적합
Response Model + Envelope 클래스	schemas.py	Pydantic 검증 유지 · OpenAPI 스키마 명확	매 라우트 return 를 감싸야 함	중
ASGI 미들웨어 (response 재작성)	main.py	라우트 코드 순수해짐	이미 봉투 씌운 응답과 이중 감싸짐 위험	하
FastAPI 의존성 (Response 훅)	커스텀 dependency	명시적 · 예외 처리 통합	파일 여러 개 수정 필요	상

강사가 이 표를 읽고 적합=상 인 3번 접근을 고른다. 다음 턴에 코드를 요청.

```
3번 접근으로 구현해줘. 다음 파일 편집.
- backend/app/envelope.py (새 파일)
- backend/app/routes/feeds.py (기존 라우트 수정)

변경 후 pytest -x 결과 확인 (카드 B 원칙 적용).
```

🎯 **결과물 미리보기.** 다음 턴에 나오는 `envelope.wrap(...)` 은 5줄짜리 유틸이다. `feeds.py` 의 라우트가 `return {"data": src, "error": None}` 을 반복하던 곳마다 `return wrap(src)` 한 줄로 축약된다 — 봉투 규약이 유틸 한 곳에만 남고 라우트 코드는 다시 순수해진다.

📌 리팩토링이 **카드 두 장 (B 회귀 방지 + C 반복 리팩토링) 의 조합** 으로 안전하게 진행된다. **초록불 유지 여부** 가 매 단계 관문.

Playwright — *프론트 회귀 시나리오 하나*

백엔드는 pytest 로 회귀를 막았다. 프론트는 Playwright 로 **한 개** 만 붙인다. 검색 → 결과 표시.

`frontend/tests/e2e/search.spec.ts`

```
import { test, expect } from "@playwright/test";

test("search shows digest items", async ({ page }) => {
  await page.goto("http://localhost:5173");

  await page.getByPlaceholder("검색").fill("anthropic");

  await expect(page.locator("article")).toHaveCount(1, { timeout: 3000 });
  await expect(page.locator("article h3")).toContainText(/./);
});
```

실행.

```
pnpm exec playwright test
```

📌 **한 시나리오만 붙이는 이유 — 투입 대비 회귀 방지 효과.** 프론트에 한 개라도 있으면 대시보드 전체 로드가 깨지는 것을 잡는다. 없으면 조용히 깨진다.

Docker Compose ① — *services.db*

회귀 방지가 갖춰졌으면 마지막 조각은 **한 명령으로 뜨는 배포**. Docker Compose 로 백엔드·DB·프론트를 한꺼번에.

`docker/docker-compose.yml` — `db` 블록

```
services:
  db:
    image: postgres:16-alpine
    environment:
      POSTGRES_USER: dash
      POSTGRES_PASSWORD: dash
      POSTGRES_DB: dash
    volumes:
      - db-data:/var/lib/postgresql/data
    ports:
      - "5432:5432"
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U dash"]
      interval: 5s
```

🐘 postgres:16-alpine

alpine 기반 경량 이미지. 로컬 실습이라 사이즈 우선.

🩺 healthcheck

`pg_isready` 로 준비 상태 확인. backend 는 `depends_on.condition: service_healthy` 로 이 healthcheck 를 기다린다.

Docker Compose ② — *services.backend*

`docker/docker-compose.yml` — `backend` 블록

```
backend:
  build:
    context: ..
    dockerfile: docker/Dockerfile.backend
  environment:
    DATABASE_URL: postgresql+asyncpg://dash:dash@db:5432/dash
    ANTHROPIC_API_KEY: ${ANTHROPIC_API_KEY}
    CORS_ORIGINS: '["http://localhost:5173"]'
  depends_on:
    db:
      condition: service_healthy
  ports:
    - "8000:8000"
  command: >
    sh -c "alembic upgrade head &&
           uvicorn app.main:app --host 0.0.0.0 --port 8000"
```

> 📎 `docker/Dockerfile.backend` (python:3.12-slim + poetry `--only main` 설치, `uvicorn` CMD — compose `command` 가 오버라이드) 전체는 뼈대 리포 참고.

- `CORS_ORIGINS: '["http://localhost:5173"]'` — `list[str]` 필드라 pydantic-settings 가 **JSON 파싱**. 리스트는 JSON 배열 문자열로 넣어야 한다.
- `ANTHROPIC_API_KEY: ${ANTHROPIC_API_KEY}` — Compose 가 **호스트 셸 환경변수를 물려받음**. 시크릿이 이미지·git 에 안 남고, 없으면 빈 문자열로 확장.

Docker Compose ③ — *services.frontend*

`docker/docker-compose.yml` — `frontend` 블록

```
frontend:
  build:
    context: ..
    dockerfile: docker/Dockerfile.frontend
  environment:
    VITE_API_BASE: http://localhost:8000
  ports:
    - "5173:5173"
  depends_on:
    - backend
```

한 명령 배포.

```
docker compose -f docker/docker-compose.yml up --build
```

🌐 접속 URL

- 프론트 — `http://localhost:5173`
- 백엔드 문서 — `http://localhost:8000/docs`

✅ 최종 초록불

`docker compose up --build` 한 줄로 3개 컨테이너가 뜨고 대시보드가 로드된다. **이 세션의 최종 초록불.**

PART 5

정리 · 확장 — *한 문장·확장 아이디어 5*

> 확장 아이디어 5개 (Slack · 모바일 · 오디오 · 백업 · 벡터 검색) · 마무리 한 문장.

확장 아이디어 5개 — *하나만 골라 가져간다*

이 대시보드는 초안이다. 각자 자기 리포로 가져가 확장할 다섯 방향을 짧게 짚는다.

#	방향	요지
1	**Slack 챗봇 인터페이스**	`/digest today` 슬래시 명령으로 요약을 Slack 채널에 게시. Slack Bolt SDK.
2	**모바일 웹 최적화**	React 컴포넌트에 responsive · PWA 매니페스트. 오프라인 캐시.
3	**오디오 노트 인덱싱**	mp3 업로드 → Whisper 로 전사 → 요약 파이프라인에 편입.
4	**일일 자동 백업**	pg_dump 를 cron 으로 · 오브젝트 스토리지에 저장.
5	**의미 검색 (벡터)**	요약본 임베딩 → pgvector 로 저장 → 자연어 검색 강화.

📌 **다섯 개를 다 하지 말 것.** 하나만 골라 자기 리포에 옮긴다. 옮기는 과정에서 오늘 배운 사이클을 자기 것으로 만든다.

마무리 — *사이클을 한 번 완주했다*

100분의 실습이 끝났다. 스펙 → 아키텍처 프롬프트 → CLAUDE.md → subagent → 반복 → 회귀 방지 → Docker Compose 까지 한 사이클을 통과했다.

🎯 오늘 실습의 뼈대 한 줄

"프롬프트 카드는 코드에 박히고, CLAUDE.md 는 매 세션 갱신되고, subagent 는 관심사를 나눈다."

🎯 오늘 확실히 남은 것

- **프롬프트 카드 A·B·C** — 설계·회귀·리팩토링
- **CLAUDE.md 갱신 감각** — 결정마다 한 줄
- **subagent 3 분업** — 관심사 격리
- **LLM 통합 표준** — JSON 강제·재시도·실패 처리
- **회귀 방지 흐름** — pytest·Playwright·Docker Compose

🚀 자기 리포로 가져갈 것

- 확장 아이디어 5 중 **하나만** 선택
- 사이드 프로젝트로 3주 정도 이어가기
- 스택이 달라도 사이클은 그대로 적용
- FastAPI 대신 Django · React 대신 Vue 여도 동일

마무리 — 사이클을 한 번 완주했다