PNU Mini Bootcamp 백엔드&클라우드 과정

3일차 - 프로젝트 설계와 초기 구현

송준우

2025년 2월 5일

멋쟁이사자처럼

목차

1. Database 연동

1.1 데이터베이스 연동

지난 시간에 만든 Blog API에 데이터베이스를 연결해봅시다.
SQLModel과 SQLite를 사용하여 로컬상에 데이터베이스를 구축합니다.
먼저 sqlmodel을 설치합니다.

```
(.venv) pip3 install sqlmodel
```

1.1 데이터베이스 연동

SQLModel은 1개의 모델 클래스가 1개의 DB Table에 대응시킬 수 있습니다.
dataclass로 만들었던 Post 클래스를 SQLModel 형태로 수정해봅시다.

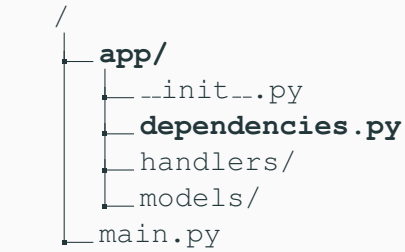
models/post_models.py

```python
from sqlmodel import Field, SQLModel

class Post(SQLModel, table=True):
    id: int | None = Field(default=None, primary_key=True)
    created_at: int | None = Field(index=True)
    published: bool = Field(index=True)
    title: str
    body: str
```

1.1 데이터베이스 연동

Database 연결관리 코드를 Dependency로 등록하면 핸들러 함수들에 적용이
용이해집니다.
아래와 같이 app/dependencies.py 파일을 생성합니다.



1.1 데이터베이스 연동

dependencies.py 파일에 데이터베이스 연결 코드를 작성합니다.

app/dependencies.py

```
from sqlmodel import Session, create_engine, select

db_file_name = "blog.db"
db_url = f"sqlite:///{db_file_name}"
db_conn_args = {"check_same_thread": False}
db_engine = create_engine(db_url, connect_args=db_conn_args)

def get_db_session():
    with Session(db_engine) as session:
        yield session
```

1.1 데이터베이스 연동

FastAPI가 처음 시작될 때 테이블을 생성하는 함수를 추가합니다.

app/dependencies.py

```
def create_db_and_tables():
    SQLModel.metadata.create_all(db_engine)
```

1.1 데이터베이스 연동

on_event라는 FastAPI 내장 함수를 이용하면 웹 서버의 라이프싸이클 이벤트 콜백을 사용할 수 있습니다.
FastAPI 서버가 처음 시작될때 호출되는 startup 이벤트 콜백을 정의하여 데이터베이스와 테이블을 생성하는 dependencies.create_db_and_tables() 함수를 호출합니다.

main.py

```
from app.dependencies import get_db_session, create_db_and_tables

@app.on_event("startup")
def on_startup():
    create_db_and_tables()
```

1.1 데이터베이스 연동

FastAPI 서버를 재시작한 뒤 프로젝트 폴더를 보면 blog.db 파일이 생성된것을 확인할 수 있습니다.

```
ctrl-C # Kill Server
fastapi dev main.py # Start Server
```

1.1 데이터베이스 연동

app/handlers/posts/posts_handler.py 파일의 create_post 함수에서 사용자의 입력 파라메터를 DB에 저장하도록 코드를 추가해봅시다.

```
from app.dependencies import get_db_session
@router.post("/", status_code=201)
def create_post(post: Post = Depends(validate_post_params),
                session = Depends(get_db_session)) -> RespPosts:
    post.created_at = int(time.time())
    session.add(post)
    session.commit()
    session.refresh(post)
    return RespPosts(
        posts=[
            post
        ]
    )
```

1.1 데이터베이스 연동

게시물 목록을 불러오는 get_posts, get_post 함수를 아래와 같이 수정하여 데이터베이스와 연동합니다.

```
from sqlmodel import select
@router.get("/")
def get_posts(page: int=1, limit: int=10,
              session=Depends(get_db_session)) -> RespPosts:
    if page < 1:
        page = 1
    if limit < 1 or limit > 10:
        limit  = 10
    nOffset = (page-1) * limit
    posts = session.exec(
        select(Post).offset(nOffset).limit(limit)
    ).all()
    return RespPosts(
        posts= posts
    )
```

1.1 데이터베이스 연동

```
from fastapi import APIRouter, Depends, HTTPException
@router.get("/{post_id}")
def get_post(post_id: int,
             session=Depends(get_db_session)) -> RespPosts:
    post = session.get(Post, post_id)
    if not post:
        raise HTTPException(status_code=404, detail="Post not found")
    return RespPosts(
        posts= [post]
    )
```

1.1 데이터베이스 연동

게시물을 삭제하는 delete_post 함수를 아래와 같이 수정합니다.

```
@router.delete("/{post_id}")
def delete_post(post_id: int,
                session=Depends(get_db_session)):
    post = session.get(Post, post_id)
    if not post:
        raise HTTPException(status_code=404, detail="Post not found")
    session.delete(post)
    session.commit()
    return {}
```

1.1 데이터베이스 연동

게시물을 수정할때는 기존 게시물을 먼저 불러온 후
파라메터로 전달받은 데이터를 dict로 변환한 값을 sqlmodel_update 함수로 병합시킵니다.
세션의 add 함수로 병합된 데이터를 DB에 적용합니다.

```
@router.put("/{post_id}")
def update_post(post_id: int,
                post: Post = Depends(validate_post_params),
                session=Depends(get_db_session)) -> RespPosts:
    oldPost = session.get(Post, post_id)
    if not oldPost:
        raise HTTPException(status_code=404, detail="Post not found")

    postData = post.model_dump(exclude_unset=True)
    oldPost.sqlmodel_update(postData)
    session.add(oldPost)
    session.commit()
    session.refresh(oldPost)

    return RespPosts(
        posts= [oldPost]
    )
```

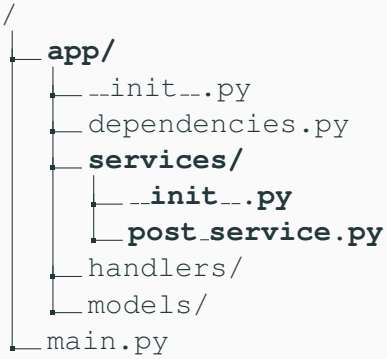
1.1.2 서비스 계층 분리

데이터를 추가, 수정하는것과 같은 비즈니스 로직은 별도의 계층으로 분리하는 것이 코드 관리에 용이합니다.
일반적으로 서비스 계층이라고 부르며 아래와 같이 services라는 폴더로 구분합니다.

```
/
├── app/
│   ├── __init__.py
│   ├── dependencies.py
│   ├── services/
│   ├── handlers/
│   └── models/
└── main.py
```

1.1.2 서비스 계층 분리

app 폴더에 services 폴더를 생성하고 게시물과 관련된 비즈니스 로직을 처리할 post_service.py 파일을 생성합니다.



1.1.2 서비스 계층 분리 i

post_handlers.py의 Post데이터 처리 로직을 post_service.py로 옮깁니다.
먼저 post_service.py 파일에 비즈니스 로직을 담당하는 PostService 클래스를 만듭니다.
이 클래스는 핸들러 함수에 의존성 주입을 통해 사용됩니다.

app/services/post_service.py

```
from fastapi import HTTPException
from app.models.post_models import Post
from sqlmodel import Session, select
import time

class PostService:
    def get_post(self, db: Session, post_id: int) -> Post:
        post = db.get(Post, post_id)
        if not post:
            raise HTTPException(status_code=404, detail="Post not found")
```

1.1.2 서비스 계층 분리 ii

```
        return post

    def get_posts(self, db: Session, page: int=1, limit: int=10) ->
        list[Post]:
        nOffset = (page-1) * limit
        posts = db.exec(
            select(Post).offset(nOffset).limit(limit)
        ).all()
        return posts

    def create_post(self, db: Session, post: Post) -> Post:
        post.created_at = int(time.time())
        db.add(post)
        db.commit()
        db.refresh(post)
        return post
```

1.1.2 서비스 계층 분리 iii

```
    def update_post(self, db:Session, post_id: int, post: Post) -> Post:
        oldPost = self.get_post(db, post_id)
        postData = post.model_dump(exclude_unset=True)
        oldPost.sqlmodel_update(postData)
        db.add(oldPost)
        db.commit()
        db.refresh(oldPost)
        return oldPost

    def delete_post(self, db:Session, post_id: int):
        post = self.get_post(db, post_id)
        db.delete(post)
        db.commit()
```

1.1.2 서비스 계층 분리 i

post_handlers.py의 핸들러 함수들의 매개변수에 PostService 클래스를 의존성 주입을 통해 가져옵니다.

app/handlers/posts/posts_handler.py

```
from app.services.post_service import PostService

@router.get("/")
def get_posts(page: int=1, limit: int=10,
              session=Depends(get_db_session),
              service: PostService = Depends()) -> RespPosts:
    if page < 1:
        page = 1
    if limit < 1 or limit > 10:
        limit  = 10
    posts = service.get_posts(session, page, limit)
```

1.1.2 서비스 계층 분리 ii

```
    return RespPosts(
        posts= posts
    )

@router.get("/{post_id}")
def get_post(post_id: int,
             session=Depends(get_db_session),
             service: PostService = Depends()) -> RespPosts:
    post = service.get_post(session, post_id)
    return RespPosts(
        posts= [post]
    )

@router.post("/", status_code=201)
def create_post(post: Post = Depends(validate_post_params),
                session = Depends(get_db_session),
```

1.1.2 서비스 계층 분리 iii

```
                service: PostService = Depends()) -> RespPosts:
    post = service.create_post(session, post)
    return RespPosts(
        posts=[
            post
        ]
    )

@router.put("/{post_id}")
def update_post(post_id: int,
                post: Post = Depends(validate_post_params),
                session=Depends(get_db_session),
                service: PostService = Depends()) -> RespPosts:
    post = service.update_post(session, post_id, post)
    return RespPosts(
        posts= [post]
```

1.1.2 서비스 계층 분리 iv

```
    )

@router.delete("/{post_id}")
def delete_post(post_id: int,
                session=Depends(get_db_session),
                service: PostService = Depends()):
    service.delete_post(session, post_id)
    return {}
```

1.2.1 환경변수

소스코드 관리를 위해 git과 같은 형상관리 도구를 사용하다보면 개발팀의 내부 또는 외부 인원과 코드를 공유하게됩니다.
만약 데이터베이스 패스워드와 같은 민감한 값들을 소스코드내에 하드코딩하게되면 보안사고의 위험이 있기도하고 개발환경에서 작업했던 코드를 프로덕션 환경으로 이전하였을 때 배포를 실행할 때마다 소스코드의 해당 부분들을 검색해서 수정해줘야 합니다.
이런 문제점들을 피하기 위해 데이터베이스 패스워드, 데이터베이스 연결 주소 등 프로그램의 작동환경에 따라 달라져야되는 값들은 환경변수나 Parameter Store(AWS)에 기록하고 소스코드 레벨에서는 이 값들을 불러와 사용하는 형태로 구현해야합니다.
Blog API의 경우 DB패스워드를 지정하지 않았지만 DB파일 경로값을 환경변수로 관리하도록 수정해봅시다.

1.2.1 환경변수

Python의 환경변수 관리 모듈인 dotenv를 설치합니다.

```
(.venv) pip3 install python-dotenv
```

main.py 파일과 동일한 위치에 .env 파일을 생성하고 아래 내용을 작성합니다.

.env

```
DB_URL=sqlite:///blog.db
```

1.2.1 환경변수

Database 연결 코드가 있는 app/dependencies.py 파일의 DB 연결 코드를 아래와 같이 수정합니다.

```
from sqlmodel import SQLModel, Session, create_engine, select
import os
from dotenv import load_dotenv

load_dotenv()

db_url = os.getenv("DB_URL")
db_conn_args = {"check_same_thread": False}
db_engine = create_engine(db_url, connect_args=db_conn_args)

...
```

2. 개인 프로젝트

2.1.1 개인 프로젝트 주제

API 프로젝트 예시

- SNS 서비스: 블로그, 트위터, ...
- 데이터 제공 서비스: 날씨, 통계, 주소검색, ...
- 유틸리티: Todo, 일정관리, 가계부, ...

프로젝트 공통 요구사항

- 회원가입, 로그인
- 데이터 가져오기, 기록하기, 삭제하기
- GET, POST, PUT/PATCH, DELETE 각 메서드 1개 이상 사용
- SQLite 데이터베이스 연동하기

2.1.2 프로젝트 폴더 구조

```
/
├── main.py: FastAPI 인스턴스
└── app/
    ├── dependencies/: 의존성
    ├── routers/: Path별 핸들러
    ├── models/: Model 클래스
    └── services/: 비즈니스 로직
```

app 폴더를 포함한 모든 하위폴더 내에는 __init__.py 파일이 있어야합니다.

2.2.1 프로젝트 설계

프로젝트 설계 절차

1. API 수요자가 개발할 프론트 어플리케이션을 스케치합니다.
2. API가 제공할 기능들을 도출합니다.
3. 필요한 데이터를 도출하고 데이터를 구조화하여 모델 클래스를 설계합니다.
4. API Routing을 정의하고 Request, Response 형식을 설계합니다.
5. 공통적인 요청,응답 Parameter를 도출하여 요청과 응답에 사용할 모델 클래스를 설계합니다.