

# PNU Mini Bootcamp 백엔드&클라우드 과정

## 2일차 - FastAPI 프로젝트 구조 설계

---

송준우

2025년 2월 4일

멋쟁이사자처럼

## 2. FastAPI 설계 실습(블로그)

### 2.1 블로그 어플리케이션 API 설계

#### 2.1.1 요구사항 분석

#### 2.1.2 API 설계

### 2.2 Dummy API 구현

#### 2.2.1 회원가입 API 구현

#### 2.2.3 게시물 목록 API 구현

#### 2.2.4 특정 아이디의 게시물 가져오기 API 구현

#### 2.2.5 게시물 작성 API 구현

#### 2.2.6 게시물 수정 API 구현

#### 2.2.7 게시물 삭제 API 구현

2.2.8 HTTP 응답코드 설정

## 2.3 리팩토링

2.3.1 Response Model

2.3.2 Parameter 검증

## 2.4 Router

2.4.1 경로별 파일 분리

## 2.5 Dependency

2.5.1 의존성 주입을 통한 공통로직 처리

## 2. FastAPI 설계 실습(블로그)

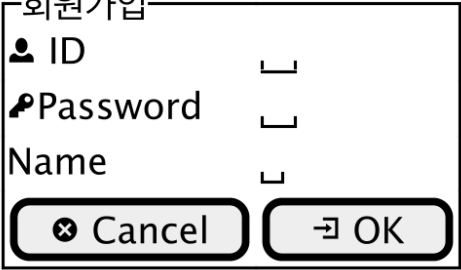
---

## 2.1.1 요구사항 분석

### 요구사항 분석

블로그 어플리케이션의 API를 설계해봅시다.

### 회원가입 화면



회원가입

👤 ID

🔑 Password

Name

⌫ Cancel

## 2.1.1 요구사항 분석

### 요구사항 분석

블로그 어플리케이션의 API를 설계해봅시다.

### 로그인 화면

A diagram of a login form. At the top left, the text "로그인" (Login) is followed by a horizontal line. Below this, there are two input fields. The first field is labeled "ID" with a person icon to its left and a small square placeholder to its right. The second field is labeled "Password" with a key icon to its left and a small square placeholder to its right. At the bottom of the form, there are two buttons: "Cancel" with a close icon (X) and "OK" with an enter icon (↵).

## 2.1.1 요구사항 분석

### 요구사항 분석

블로그 어플리케이션의 API를 설계해봅시다.

### 게시물 목록 화면

"키워드 " [검색]

첫번째 게시물입니다. 1일전

두번째 게시물입니다. 2일전

.....  
.....  
.....  
.....  
.....

이전페이지

다음페이지



## 2.1.1 요구사항 분석

### 요구사항 분석

블로그 어플리케이션의 API를 설계해봅시다.

### 게시물 보기 화면

Linux의 블로그

게시물 제목입니다. 작성자: Linux  
2025.02.03  
게시물 내용입니다.  
동해물과 백두산이  
마르고 닳도록  
.....  
.....  
.....

▲

▼

수정

공개/비공개

삭제

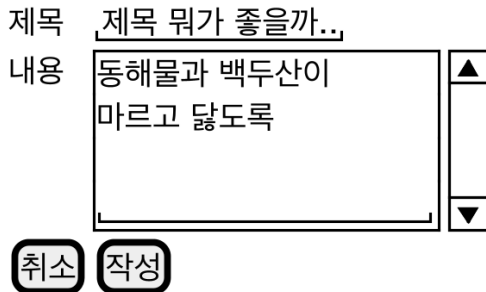


## 2.1.1 요구사항 분석

### 요구사항 분석

블로그 어플리케이션의 API를 설계해봅시다.

### 게시물 작성/수정 화면



제목 제목 뭐가 좋을까..

내용 동해물과 백두산이  
마르고 닳도록

취소 작성

## 2.1.1 요구사항 분석

### 요구사항 분석

블로그 어플리케이션에 필요한 API를 도출해봅시다.

- 회원가입
- 로그인
- 게시물 목록(검색)
- 게시물 가져오기
- 게시물 작성
- 게시물 수정
- 게시물 공개/비공개
- 게시물 삭제

## 2.1.2 API 설계

### 사용자 모델: User

- ID
- Password
- 사용자 이름

### 게시물 모델: Post

- 게시물 고유번호
- 제목
- 내용
- 작성일시
- 공개/비공개 여부

## 2.1.2 API 설계

dataclass를 이용하여 모델 클래스를 정의합니다.

```
from dataclasses import dataclass
@dataclass
class User:
    login_id: str
    password: str
    name: str

@dataclass
class Post:
    id: int
    title: str
    body: str
    created_at: int
    published: bool
```

## 2.1.2 API 설계

요구사항 분석 단계에서 도출한 내용을 기반으로 API를 정의합니다.

### 회원가입 Request

<b>URL</b>	/auth/signup			
<b>Method</b>	POST			
<b>Content-Type</b>	application/json; charset=utf-8			
<b>Request Body</b>	<b>Name</b>	<b>Type</b>	<b>Required</b>	<b>Default</b>
	login_id	str	True	
	pwd	str	True	
	name	str	True	

## 2.1.2 API 설계

### 회원가입 Response

<b>Content-Type</b>	application/json; charset=utf-8				
<b>Status Code</b>	201 Created				
<b>Response Body</b>	<b>Name</b>	<b>Type</b>	<b>Required</b>	<b>Default</b>	
	jwt_token	str	False	null	
	err_msg	str	False	null	

## 2.1.2 API 설계

### 로그인 Request

<b>URL</b>	/auth/signin				
<b>Method</b>	POST				
<b>Content-Type</b>	application/json; charset=utf-8				
<b>Request Body</b>	<b>Name</b>	<b>Type</b>	<b>Required</b>	<b>Default</b>	
	login_id	str	True		
	pwd	str	True		

### 로그인 Response

<b>Status Code</b>	200 OK				
<b>Response Body</b>	<b>Name</b>	<b>Type</b>	<b>Required</b>	<b>Default</b>	
	jwt_token	str	False	null	
	err_msg	str	False	null	

## 2.1.2 API 설계

### 게시물 목록 Request

<b>URL</b>	/posts			
<b>Method</b>	GET			
<b>Content-Type</b>	application/json; charset=utf-8			
<b>Query Params</b>	<b>Name</b>	<b>Type</b>	<b>Required</b>	<b>Default</b>
	page	int	False	1
	limit	int	False	10



## 2.1.2 API 설계

### 게시물 목록 Response

<b>Content-Type</b>	application/json; charset=utf-8			
<b>Status Code</b>	200 OK			
<b>Response Body</b>	<b>Name</b>	<b>Type</b>	<b>Required</b>	<b>Default</b>
	posts	list[Post]	True	[ ]

## 2.1.2 API 설계

### 게시물 1개 가져오기 Request

<b>URL</b>	/posts/{post_id}			
<b>Method</b>	GET			
<b>Content-Type</b>	application/json; charset=utf-8			
<b>Path Params</b>	<b>Name</b>	<b>Type</b>	<b>Required</b>	<b>Default</b>
	post_id	int	True	None

## 2.1.2 API 설계

### 게시물 1개 가져오기 Response

<b>Content-Type</b>	application/json; charset=utf-8			
<b>Status Code</b>	200 OK			
<b>Response Body</b>	Name	Type	Required	Default
	posts	list[Post]	True	[ ]

## 2.1.2 API 설계

### 게시물 작성 Request

<b>URL</b>	/posts				
<b>Method</b>	POST				
<b>Content-Type</b>	application/json; charset=utf-8				
<b>Request Body</b>	<b>Name</b>	<b>Type</b>	<b>Required</b>	<b>Default</b>	
	title	str	True	None	
	body	str	True	None	
	publish	bool	False	False	

## 2.1.2 API 설계

### 게시물 작성 Response

<b>Content-Type</b>	application/json; charset=utf-8			
<b>Status Code</b>	201 Created			
<b>Response Body</b>	<b>Name</b>	<b>Type</b>	<b>Required</b>	<b>Default</b>
	posts	list[Post]	True	[ ]

## 2.1.2 API 설계

### 게시물 수정 Request

<b>URL</b>	/posts/{post_id}				
<b>Method</b>	PUT				
<b>Content-Type</b>	application/json; charset=utf-8				
<b>Path Params</b>	<b>Name</b>	<b>Type</b>	<b>Required</b>	<b>Default</b>	
	post_id	int	True	None	
<b>Request Body</b>	<b>Name</b>	<b>Type</b>	<b>Required</b>	<b>Default</b>	
	title	str	False	None	
	body	str	False	None	
	publish	bool	False	False	

## 2.1.2 API 설계

### 게시물 수정 Response

<b>Content-Type</b>	application/json; charset=utf-8			
<b>Status Code</b>	200 OK			
<b>Response Body</b>	<b>Name</b>	<b>Type</b>	<b>Required</b>	<b>Default</b>
	posts	list[Post]	True	[ ]

## 2.1.2 API 설계

### 게시물 공개/비공개 Request

<b>URL</b>	/posts/{post_id}				
<b>Method</b>	PUT				
<b>Content-Type</b>	application/json; charset=utf-8				
<b>Path Params</b>	Name	Type	Required	Default	
	post_id	int	True	None	
<b>Request Body</b>	Name	Type	Required	Default	
	publish	bool	False	False	



## 2.1.2 API 설계

### 게시물 공개/비공개 Response

<b>Content-Type</b>	application/json; charset=utf-8			
<b>Status Code</b>	200 OK			
<b>Response Body</b>	Name	Type	Required	Default
	posts	list[Post]	True	[ ]

## 2.1.2 API 설계

### 게시물 삭제 Request

<b>URL</b>	/posts/{post_id}			
<b>Method</b>	DELETE			
<b>Content-Type</b>	application/json; charset=utf-8			
<b>Path Params</b>	<b>Name</b>	<b>Type</b>	<b>Required</b>	<b>Default</b>
	post_id	int	True	None

## 2.1.2 API 설계

### 게시물 삭제 Response

<b>Content-Type</b>	application/json; charset=utf-8			
<b>Status Code</b>	200 OK			
<b>Response Body</b>	Name	Type	Required	Default

## 2.2 Dummy API 구현

API 설계를 기반으로 Dummy API를 구현해봅시다.  
이 단계를 통해 FastAPI가 생성하는 Swagger UI로  
설계한 내용을 검증해볼 수 있습니다.

Dummy API 구현을 위해 새로운 프로젝트를 생성합니다.

```
mkdir fastapi-002
cd fastapi-002
python3 -m venv .venv
source .venv/activate
(venv) pip3 install "fastapi[standard]"
```

## 2.2.1 회원가입 API 구현

main.py 파일을 생성하고 앞에서 설계한 모델 클래스를 정의합니다.

```
from dataclasses import dataclass
@dataclass
class User:
    login_id: str
    password: str
    name: str

@dataclass
class Post:
    id: int
    title: str
    body: str
    created_at: int
    published: bool
```

## 2.2.1 회원가입 API 구현

fastapi 모듈을 import하고 회원가입 요청 핸들러 함수의 기본 코드를 작성합니다.

```
from fastapi import FastAPI
app = FastAPI()

@app.post("/auth/signup")
def auth_signup(user: User):
    return {
        "jwt_token": "Here is your token",
        "err_msg": None
    }
```

## 2.2.1 회원가입 API 구현

fastapi 서버를 실행 후 `http://127.0.0.1:8000/docs`에 접속하여 `/auth/signup` API를 테스트해보세요.

```
(venv) fastapi dev main.py
```

```
{  
  "jwt_token": "Here is your token",  
  "err_msg": null  
}
```

## 2.2.2 로그인 API 구현

아래와 같이 로그인 API의 기본 코드를 작성 후 /docs에 접속하여 API를 테스트해봅시다.

```
@app.post("/auth/signin")
def auth_signin(user: User):
    return {
        "jwt_token": "Here is your token",
        "err_msg": None
    }
```



## 2.2.2 로그인 API 구현

로그인API와 회원가입API의 Request Body 항목은 name외에는 모두 동일하여 User 모델 클래스를 활용하면 코드를 간소화 시킬 수 있습니다.  
회원가입API에는 login\_id, password, name이 모두 필요하지만 로그인API는 name이 필요없기에 User 클래스의 name 항목을 Optional로 지정해주어야합니다.

```
from typing import Union
@dataclass
class User:
    login_id: str
    password: str
    name: Union[str, None] = None
```

## 2.2.2 로그인 API 구현

User 클래스의 name 항목을 Optional로 변경 후에는 로그인API를 요청할때 name 매개변수를 입력하지 않아도 오류가 발생하지 않습니다.

하지만 Swagger API문서에는 여전히 남아있습니다. 이 문제는 4회차 수업에서 다시 살펴보겠습니다.

POST /auth/signin Auth Signin	
Parameters	
No parameters	
Request body <small>required</small>	
Example Value   Schema	
<pre>{   "login_id": "string",   "password": "string",   "name": "string" }</pre>	

## 2.2.3 게시물 목록 API 구현

```
import time
@app.get("/posts")
def get_posts(page: int=1, limit: int=10):
    return {
        "posts": [
            Post(id=1,
                title="Post1",
                body="Hello\nWorld",
                created_at=int(time.time()),
                published=True),
            Post(id=2,
                title="Post2",
                body="Hello\nNew World",
                created_at=int(time.time()),
                published=True)
        ]
    }
```

## 2.2.4 특정 게시물 가져오기 API 구현

```
@app.get("/posts/{post_id}")
def get_post(post_id: int):
    return {
        "posts": [
            Post(id=post_id,
                title=f"Post{post_id}",
                body="Hello\nWorld",
                created_at=int(time.time()),
                published=True)
        ]
    }
```

## 2.2.5 게시물 작성 API 구현

```
@app.post("/posts")
def create_post(post: Post):
    return {
        "posts": [
            Post(id=100,
                title=post.title,
                body=post.body,
                created_at=int(time.time()),
                published=post.published)
        ]
    }
```

## 2.2.6 게시물 수정 API 구현

```
@app.put("/posts/{post_id}")
def update_post(post_id: int, post: Post):
    return {
        "posts": [
            Post(id=post_id,
                title=post.title,
                body=post.body,
                created_at=int(time.time()),
                published=post.published)
        ]
    }
```

## 2.2.7 게시물 삭제 API 구현

```
@app.delete("/posts/{post_id}")  
def delete_post(post_id: int):  
    return {}
```

## 2.2.8 HTTP 응답코드 설정

FastAPI는 별도의 상태코드를 지정하지 않으면 성공시 기본 응답코드를 200 OK를 사용합니다. 아래와 같이 Method 데코레이터에 `status_code`를 추가하여 특정한 코드를 지정할 수 있습니다.

```
@app.post("/auth/signup", status_code=201)
def auth_signup(user: User):
    ...

@app.post("/posts", status_code=201)
def create_post(post: Post):
    ...
```



## 2.3.1 Response Model i

FastAPI 핸들링 함수에 리턴 타입을 지정하면 API응답 형식을 쉽게 검증할 수 있습니다.

회원가입API와 로그인API는 `jwt_token`과 `err_msg`를 응답합니다.

`dataclass`로 응답형식을 정의하여 핸들러 함수의 리턴타입으로 사용해봅시다.

```
class RespAuth:
    jwt_token: str | None = None
    err_msg: str | None = None

@app.post("/auth/signup", status_code=201)
def auth_signup(user: User) -> RespAuth:
    return RespAuth(
        jwt_token="This is your token",
        err_msg=None
    )
```

```
@app.post("/auth/signin")
```

## 2.3.1 Response Model ii

```
def auth_signin(user: User) -> RespAuth:  
    return RespAuth(  
        jwt_token="This is your token",  
        err_msg=None  
    )
```

## 2.3.1 Response Model i

게시물 모델인 Post 배열을 응답하는 API들은 아래와 같이 응답형식을 지정할 수 있습니다.

```
class RespPosts:
    posts: list[Post] | None = None
    err_msg: str | None = None

@app.get("/posts")
def get_posts(page: int=1, limit: int=10) -> RespPosts:
    return RespPosts(
        posts= [
            Post(id=1,
                title="Post1",
                body="Hello\nWorld",
                created_at=int(time.time()),
                published=True),
```

## 2.3.1 Response Model ii

```
        Post(id=2,
              title="Post2",
              body="Hello\nNew World",
              created_at=int(time.time()),
              published=True)
    ]
)

@app.get("/posts/{post_id}")
def get_post(post_id: int) -> RespPosts:
    return RespPosts(
        posts= [
            Post(id=post_id,
                  title=f"Post{post_id}",
                  body="Hello\nWorld",
                  created_at=int(time.time()),
```

## 2.3.1 Response Model iii

```
                published=True)
        ]
    )

@app.post("/posts", status_code=201)
def create_post(post: Post) -> RespPosts:
    return RespPosts(
        posts=[
            Post(id=100,
                title=post.title,
                body=post.body,
                created_at=int(time.time()),
                published=post.published)
        ]
    )
```

## 2.3.1 Response Model iv

```
@app.put("/posts/{post_id}")
def update_post(post_id: int, post: Post) -> RespPosts:
    return RespPosts(
        posts= [
            Post(id=post_id,
                title=post.title,
                body=post.body,
                created_at=int(time.time()),
                published=post.published)
        ]
    )
```

## 2.3.2 Parameter 검증

외부로부터 들어오는 데이터는 반드시 검증이 필요합니다.

해킹 공격 또는 프론트 코드 구현의 오류로 인해 서버 리소스에 예상치 못한 잘못된 영향을 줄 수 있습니다.

### SQL Injection

' OR '1'='1

```
SELECT id FROM users WHERE pwd='{param}';
```

```
SELECT id FROM users WHERE pwd='' OR '1'='1';
```

## 2.3.2 Parameter 검증

### Cross-Site Scripting

```
<script>alert("Hello I'm a hacker");</script>
```



## 2.3.2 Parameter 검증

### Directory Traversal

```
../../../../etc/passwd
```

## 2.3.2 Parameter 검증

### File Upload

- .jsp
- .php
- .sh

## 2.3.2 Parameter 검증

### 다른 소유자의 리소스 수정

```
@app.put('/posts/{id}')
def update_post(body: str):
    db.UpdatePostBody(id, body)
    return {
        'success': True
    }
```

## 2.3.2 Parameter 검증

### User 클래스에 데이터 검증함수 추가하기

```
@dataclass
class User:
    login_id: str
    password: str
    name: Union[str, None] = None

    def validate(self):
        if len(self.login_id) < 4 or len(self.login_id) > 12:
            return 'Length of login_id: 4~12'

        if len(self.password) < 4 or len(self.password) > 12:
            return 'Length of password: 4~12'

        if (self.name is not None and
            (len(self.name)<2 or len(self.name)>20)):
            return 'Length of name: 2~20'

        return None
```

## 2.3.2 Parameter 검증

### signup 핸들러에 데이터 검증로직 추가하기

```
@app.post("/auth/signup", status_code=201)
def auth_signup(user: User) -> RespAuth:
    if (err := user.validate()) is not None:
        return RespAuth(
            err_msg=err
        )
    return RespAuth(
        jwt_token="This is your token",
        err_msg=None
    )
```

## 2.3.2 Parameter 검증

### signin 핸들러에 데이터 검증로직 추가하기

```
@app.post("/auth/signin")
def auth_signin(user: User) -> RespAuth:
    if (err := user.validate()) is not None:
        return RespAuth(
            err_msg=err
        )
    return RespAuth(
        jwt_token="This is your token",
        err_msg=None
    )
```

## 2.3.2 Parameter 검증

### get\_posts 함수의 파라미터 검증

```
@app.get("/posts")
def get_posts(page: int=1, limit: int=10) -> list[Post]:
    if page < 1:
        page = 1
    if limit < 1 or limit > 10:
        limit = 10
    return {
        "posts": [
            ...
        ]
    }
```

## 2.3.2 Parameter 검증

### get\_post 함수의 파라미터 검증

```
@app.get("/posts/{post_id}")
def get_post(post_id: int) -> RespPosts:
    if post_id < 1:
        return RespPosts(
            err_msg="post_id should be greater than 0"
        )
    return RespPosts(
        posts= [
            Post(id=post_id,
                title=f"Post{post_id}",
                body="Hello\nWorld",
                created_at=int(time.time()),
                published=True)
        ]
    )
```



## 2.3.2 Parameter 검증

### Post 클래스에 데이터 검증함수 추가하기

```
@dataclass
class Post:
    title: str
    body: str
    created_at: Union[int, None] = None
    id: Union[int, None] = None
    published: bool = False

    def validate(self):
        if len(self.title) < 4 or len(self.title) > 30:
            return 'Length of title: 4~30'
        if len(self.body) < 10 or len(self.body) > 100:
            return 'Length of content: 10~100'
```

## 2.3.2 Parameter 검증

### 게시물 작성/수정 시 입력 데이터 검증하기

```
@app.post("/posts", status_code=201)
def create_post(post: Post) -> RespPosts:
    if (err := post.validate()) is not None:
        return RespPosts(
            err_msg=err
        )
    return RespPosts(
        posts=[
            Post(id=100,
                title=post.title,
                body=post.body,
                created_at=int(time.time()),
                published=post.published)
        ]
    )
```

## 2.3.2 Parameter 검증

### 게시물 작성/수정 시 입력 데이터 검증하기

```
@app.put("/posts/{post_id}")
def update_post(post_id: int, post: Post) -> RespPosts:
    if (err := post.validate()) is not None:
        return RespPosts(
            err_msg=err
        )
    return RespPosts(
        posts= [
            Post(id=post_id,
                title=post.title,
                body=post.body,
                created_at=int(time.time()),
                published=post.published)
        ]
    )
```

## 2.4.1 경로별 파일 분리

블로그 API는 크게 2가지로 분류할 수 있습니다.

- 사용자 인증 처리 `/auth/...`
- 게시물 처리 `/posts/...`

Path의 최상위 항목을 기준으로 API들을 각각 별도의 파일로 나누고 Router를 사용하여 `main.py` 파일에서 분리된 파일의 API들을 FastAPI의 핸들러로 추가할 수 있습니다.

## 2.4.1 경로별 파일 분리

아래와 같이 서브 폴더를 추가하고 각 폴더내에 `--init--.py` 파일을 생성합니다.

```
/
├── app/
│   ├── --init--.py
│   ├── handlers/
│   │   ├── --init--.py
│   │   ├── auth/
│   │   │   ├── --init--.py
│   │   ├── posts/
│   │   │   ├── --init--.py
│   └── main.py
```

## 2.4.1 경로별 파일 분리 i

app/handlers/auth 폴더에 auth\_handlers.py 파일을 추가하고 main.py의 아래 항목들을 옮겨봅시다.

### app/handlers/auth/auth\_handlers.py

```
from dataclasses import dataclass
from typing import Union

@dataclass
class User:
    login_id: str
    password: str
    name: Union[str, None] = None

    def validate(self):
        if len(self.login_id) < 4 or len(self.login_id) > 12:
            return 'Length of login_id: 4~12'
```

## 2.4.1 경로별 파일 분리 ii

```
    if len(self.password) < 4 or len(self.password) > 12:
        return 'Length of password: 4~12'

    if (self.name is not None and
        (len(self.name)<2 or len(self.name)>20)):
        return 'Length of name: 2~20'
    return None

@dataclass
class RespAuth:
    jwt_token: str | None = None
    err_msg: str | None = None

@app.post("/auth/signup", status_code=201)
def auth_signup(user: User) -> RespAuth:
    if (err := user.validate()) is not None:
        return RespAuth(
            err_msg=err
```

## 2.4.1 경로별 파일 분리 iii

```
    )
    return RespAuth(
        jwt_token="This is your token",
        err_msg=None
    )

@app.post("/auth/signin")
def auth_signin(user: User) -> RespAuth:
    if (err := user.validate()) is not None:
        return RespAuth(
            err_msg=err
        )
    return RespAuth(
        jwt_token="This is your token",
        err_msg=None
    )
```



## 2.4.1 경로별 파일 분리 i

app/handlers/posts 폴더에 posts\_handlers.py 파일을 추가하고 main.py의 아래 항목들을 옮겨봅시다.

### app/handlers/posts/posts\_handlers.py

```
from dataclasses import dataclass
from typing import Union
import time

@dataclass
class Post:
    title: str
    body: str
    created_at: Union[int, None] = None
    id: Union[int, None] = None
    published: bool = False

    def validate(self):
```

## 2.4.1 경로별 파일 분리 ii

```
        if len(self.title) < 4 or len(self.title) > 30:
            return 'Length of title: 4~30'
        if len(self.body) < 10 or len(self.body) > 100:
            return 'Length of content: 10~100'

@dataclass
class RespPosts:
    posts: list[Post] | None = None
    err_msg: str | None = None

@app.get("/posts")
def get_posts(page: int=1, limit: int=10) -> RespPosts:
    if page < 1:
        page = 1
    if limit < 1 or limit > 10:
        limit = 10
    return RespPosts(
        posts= [
```

## 2.4.1 경로별 파일 분리 iii

```
Post(id=1,
     title="Post1",
     body="Hello\nWorld",
     created_at=int(time.time()),
     published=True),
Post(id=2,
     title="Post2",
     body="Hello\nNew World",
     created_at=int(time.time()),
     published=True)
]
```

```
@app.get("/posts/{post_id}")
def get_post(post_id: int) -> RespPosts:
    if post_id < 1:
        return RespPosts(
            err_msg="post_id should be greater than 0"
```

## 2.4.1 경로별 파일 분리 iv

```
        )
    return RespPosts(
        posts= [
            Post(id=post_id,
                title=f"Post{post_id}",
                body="Hello\nWorld",
                created_at=int(time.time()),
                published=True)
        ]
    )

@app.post("/posts", status_code=201)
def create_post(post: Post) -> RespPosts:
    if (err := post.validate()) is not None:
        return RespPosts(
            err_msg=err
        )
    return RespPosts(
```

## 2.4.1 경로별 파일 분리 v

```
        posts=[
            Post(id=100,
                title=post.title,
                body=post.body,
                created_at=int(time.time()),
                published=post.published)
        ]
    )

@app.put("/posts/{post_id}")
def update_post(post_id: int, post: Post) -> RespPosts:
    if (err := post.validate()) is not None:
        return RespPosts(
            err_msg=err
        )
    return RespPosts(
        posts= [
            Post(id=post_id,
```

## 2.4.1 경로별 파일 분리 vi

```
        title=post.title,  
        body=post.body,  
        created_at=int(time.time()),  
        published=post.published)  
    ]  
)  
  
@app.delete("/posts/{post_id}")  
def delete_post(post_id: int):  
    return {}
```

## 2.4.1 경로별 파일 분리 i

이제 main.py 파일에는 아래 내용만 남게 됩니다. **main.py**

```
from fastapi import FastAPI

app = FastAPI()
```

## 2.4.1 경로별 파일 분리

auth\_handlers.py 파일에 라우터 모듈을 추가하고  
아래와 같이 각 핸들러 함수의 @app 데코레이터를 @router로 수정합니다.  
Router 모듈을 생성할때 prefix로 Path의 최상위 단계인 auth를 추가하면  
각 핸들러함수의 Path 앞에 자동으로 auth가 붙습니다.

### app/handlers/auth/auth\_handlers.py

```
from fastapi import APIRouter

router = APIRouter(
    prefix="/auth"
)

@router.post("/signup", status_code=201)
...

@router.post("/signin")
...
```



## 2.4.1 경로별 파일 분리

posts\_handlers.py 파일에도 라우터 모듈을 추가하고 핸들러 함수들의 데코레이터와 Path를 수정해봅시다.

### app/handlers/posts/posts\_handlers.py

```
from fastapi import APIRouter
router = APIRouter(
    prefix="/posts"
)
@router.get("/")
...
```

## 2.4.1 경로별 파일 분리

main.py 파일에 posts\_handlers 모듈과 auth\_handlers 모듈을 불러와 FastAPI의 라우터로 등록합니다.

### main.py

```
from fastapi import FastAPI
from app.handlers.auth import auth_handlers
from app.handlers.posts import posts_handlers

app = FastAPI()

app.include_router(auth_handlers.router)
app.include_router(posts_handlers.router)
```

## 2.4.1 경로별 파일 분리

입력과 출력에 사용되는 모델 클래스들도 별도의 파일로 분리해봅시다.

아래처럼 app/ 아래에 models 폴더를 추가하고

\_\_init\_\_.py, user\_models.py, post\_models.py 파일을 생성합니다.

```
/
├── app/
│   ├── __init__.py
│   └── models/
│       ├── __init__.py
│       ├── user_models.py
│       └── post_models.py
└── handlers/
main.py
```

## 2.4.1 경로별 파일 분리 i

auth\_handlers.py의 User, RespData 클래스를 user\_models.py 파일로 옮깁니다.

### app/models/user\_models.py

```
from dataclasses import dataclass
from typing import Union

@dataclass
class User:
    login_id: str
    password: str
    name: Union[str, None] = None

    def validate(self):
        if len(self.login_id) < 4 or len(self.login_id) > 12:
            return 'Length of login_id: 4~12'

        if len(self.password) < 4 or len(self.password) > 12:
            return 'Length of password: 4~12'
```

## 2.4.1 경로별 파일 분리 ii

```
    if (self.name is not None and
        (len(self.name)<2 or len(self.name)>20)):
        return 'Length of name: 2~20'
    return None
```

```
@dataclass
class RespAuth:
    jwt_token: str | None = None
    err_msg: str | None = None
```

## 2.4.1 경로별 파일 분리 i

posts\_handlers.py의 Post, RespPost 클래스를 app/models/post\_models.py 파일로 옮깁니다. **app/models/post\_models.py**

```
from dataclasses import dataclass
from typing import Union

@dataclass
class Post:
    title: str
    body: str
    created_at: Union[int, None] = None
    id: Union[int, None] = None
    published: bool = False

    def validate(self):
        if len(self.title) < 4 or len(self.title) > 30:
            return 'Length of title: 4~30'
        if len(self.body) < 10 or len(self.body) > 100:
```

## 2.4.1 경로별 파일 분리 ii

```
        return 'Length of content: 10~100'
```

```
@dataclass
```

```
class RespPosts:
```

```
    posts: list[Post] | None = None
```

```
    err_msg: str | None = None
```

## 2.4.1 경로별 파일 분리

auth\_handlers.py, posts\_handlers.py 파일에서 models 모듈을 불러오도록 수정합니다.

**app/handlers/auth/auth\_handlers.py**

```
from app.models.user_models import *
```

**app/handlers/posts/posts\_handlers.py**

```
from app.models.post_models import *
```



## 2.5.1 의존성 주입을 통한 공통로직 처리

`auth_signup`, `auth_signin` 함수는 `User.validate()` 함수를 이용해 클라이언트의 입력을 검증하는 공통된 로직을 처리하고 있고

`create_post`, `update_post` 함수는 `Post.validate()` 함수를 이용해 클라이언트의 입력을 검증하고 있습니다. FastAPI의 Dependency Injection 기능을 사용하면 중복되는 로직을 분리하여 코드를 단순화 시킬 수 있습니다.

User모델과 Post모델의 파라미터를 검증하는 별도의 함수로 분리한 후 FastAPI의 의존성 주입 기능으로 중복되는 파라미터 검증 코드를 줄여봅시다.

## 2.5.1 의존성 주입을 통한 공통로직 처리

### User 파라미터 검증함수 분리 `app/models/user_models.py`

```
@dataclass
class User:
    login_id: str
    password: str
    name: Union[str, None] = None

def validate_user_params(user: User) -> str | None:
    if len(self.login_id) < 4 or len(self.login_id) > 12:
        return 'Length of login_id: 4~12'

    if len(self.password) < 4 or len(self.password) > 12:
        return 'Length of password: 4~12'

    if (self.name is not None and
        (len(self.name)<2 or len(self.name)>20)):
        return 'Length of name: 2~20'

    return None
```

## 2.5.1 의존성 주입을 통한 공통로직 처리

### Post 파라미터 검증함수 분리 `app/models/post_models.py`

```
@dataclass
class Post:
    title: str
    body: str
    created_at: Union[int, None] = None
    id: Union[int, None] = None
    published: bool = False

def validate_post_params(post: Post) -> str | None:
    if len(self.title) < 4 or len(self.title) > 30:
        return 'Length of title: 4~30'
    if len(self.body) < 10 or len(self.body) > 100:
        return 'Length of content: 10~100'
    return None
```

## 2.5.1 의존성 주입을 통한 공통로직 처리 i

validate\_user\_params() 함수가 User 파라미터 검증에 실패하면 400 Bad Request 예외를 발생시키고 검증이 완료되면 파라미터를 그대로 리턴하도록 수정해봅시다.

**models/user\_models.py**

```
from fastapi import HTTPException, status

def validate_user_params(user: User) -> User:
    if len(user.login_id) < 4 or len(user.login_id) > 12:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail='Length of login_id: 4~12'
        )

    if len(user.password) < 4 or len(user.password) > 12:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail='Length of password: 4~12'
        )
```

## 2.5.1 의존성 주입을 통한 공통로직 처리 ii

```
if (user.name is not None and
    (len(user.name)<2 or len(user.name)>20)):
    raise HTTPException(
        status_code=status.HTTP_400_BAD_REQUEST,
        detail='Length of name: 2~20'
    )
return user
```

## 2.5.1 의존성 주입을 통한 공통로직 처리

validate\_post\_params() 함수가 User 파라미터 검증에 실패하면 400 Bad Request 예외를 발생시키고 검증이 완료되면 파라미터를 그대로 리턴하도록 수정해봅시다.

### models/post\_models.py

```
from fastapi import HTTPException, status

def validate_post_params(post: Post) -> Post:
    if len(post.title) < 4 or len(post.title) > 30:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail='Length of title: 4~30'
        )
    if len(post.body) < 10 or len(post.body) > 100:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail='Length of content: 10~100'
        )
    return post
```

## 2.5.1 의존성 주입을 통한 공통로직 처리

auth\_handlers.py 파일의 User 파라미터 검증 코드를 삭제하고 Depends 모듈로 검증로직을 자동화시킵니다. **handlers/auth/auth\_handlers.py**

```
from fastapi import APIRouter, Depends
@router.post("/signup", status_code=201)
def auth_signup(user: User = Depends(validate_user_params)) -> RespAuth:
    return RespAuth(
        jwt_token="This is your token",
        err_msg=None
    )

@router.post("/signin")
def auth_signin(user: User = Depends(validate_user_params)) -> RespAuth:
    return RespAuth(
        jwt_token="This is your token",
        err_msg=None
    )
```

## 2.5.1 의존성 주입을 통한 공통로직 처리 i

posts\_handlers.py 파일에서 Post 파라미터 검증 코드를 삭제하고 Depends 모듈로 검증로직을 자동화시킵니다. **handlers/posts/posts\_handlers.py**

```
from fastapi import APIRouter, Depends
@router.post("/", status_code=201)
def create_post(post: Post = Depends(validate_post_params)) -> RespPosts:
    return RespPosts(
    )

@router.put("/{post_id}")
def update_post(post_id: int, post: Post = Depends(validate_post_params)) ->
    RespPosts:
    return RespPosts(
    )
```