FastAPI 특강

3일차: CRUD + DB연동 및 인증 & 보안

목 차

- 2일차 복습
- 비동기 DB로 CRUD 구현하기
- 인증의 중요성 및 JWT 개념
- Depends를 사용한 인증 디펜던시

2일차 복습

Pydantic 기본 개념 및 모델 정의

중첩 데이터 모델 및 유효성 검사: 요청에 잘못된 데이터가 포함되었을 때

코드 정의

```python
class Address(BaseModel):
    city: str
    zip_code: str

class User(BaseModel):
    name: str
    age: int
    address: Address
```

```python
@app.post("/users/")
def create_user(user: User):
    return {"user": user}
```

요청 JSON

```json
{
  "name": "OZ",
  "age": 20,
  "address": {
    "city": "Seoul",
    "zip_code": 10001
  }
}
```

응답 JSON

```json
{
    "detail": [
        {
            "type": "string_type",
            "loc": [
                "body",
                "address",
                "zip_code"
            ],
            "msg": "Input should be a valid string",
            "input": 10001,
            "url": "https://errors.pydantic.dev/2.12/v/string_type"
        }
    ]
}
```

Pydantic 의 validator 개념 및 사용법

validator의 기본 사용법

- 구조 예시

```python
from pydantic import BaseModel, field_validator

class ExampleModel(BaseModel):
    number: int

    @field_validator("number")
    @classmethod
    def validate_number(cls, value):
        if value < 1:
            raise ValueError("Number must be at least 1")
        if value > 100:
            raise ValueError("Number must not exceed 100")
        return value
```

고급 Pydantic 검증: 여러 필드 검증

- mode = “before” 예시

```python
from pydantic import BaseModel, model_validator

class User(BaseModel):
    username: str

    @model_validator(mode="before")
    @classmethod
    def preprocess_username(cls, data):
        if isinstance(data, dict) and "username" in data:
            data["username"] = data["username"].lower()  # 소문자로 변환
        return data

# 테스트
user = User(username="JohnDoe")
print(user.username)  # 출력: "johndoe"
```

고급 Pydantic 검증: 여러 필드 검증

- mode = “after” 예시

```python
from pydantic import BaseModel, model_validator

class UserRegister(BaseModel):
    username: str
    password: str
    confirm_password: str

    @model_validator(mode="after")
    def check_password_match(self):
        if self.password != self.confirm_password:
            raise ValueError("Passwords do not match")
        return self

# 테스트
try:
    user = UserRegister(username="testuser", password="StrongPass1!", confirm_password="WrongPass!")
except ValueError as e:
    print("Error:", e)
```

고급 Pydantic 검증: 자동 계산 필드

```python
from pydantic import BaseModel, computed_field
from datetime import datetime

class User(BaseModel):
    name: str
    birth_year: int

    @computed_field  # 자동 계산 필드
    @property
    def age(self) -> int:
        return datetime.now().year - self.birth_year

# 테스트
user = User(name="Alice", birth_year=2000)
print(user.age)  # 현재 연도 - 2000 (예: 2025 - 2000 = 25)
```

고급 Pydantic 검증: 초기값 동적 설정

```python
from pydantic import BaseModel, Field
from datetime import datetime

class LogEntry(BaseModel):
    message: str
    created_at: datetime = Field(default_factory=datetime.utcnow)

# 테스트
log1 = LogEntry(message="System started")
log2 = LogEntry(message="User login")

print(log1.created_at)  # 자동 생성된 UTC 시간
print(log2.created_at)  # 다른 시간으로 자동 설정됨
```

비동기 DB로 CRUD 구현하기

비동기 Asynchronous

- 작업 요청과 결과 처리가 동시에 진행되지 않고, **요청이 처리되는 동안 다른 작업을 수행할 수 있도록** 하는 방식

- Event Loop를 기반으로 작업의 시작, 중간 처리, 완료를 관리

- Python에서는 **async**와 **await** 키워드를 사용하여 비동기 처리를 명시적으로 구현

비동기 DB 작업

- 데이터베이스 요청(ex. 데이터 삽입, 조회, 수정 등)을 처리할 때 I/O 작업(ex. 네트워크 또는 파일 시스템 접근)이 발생

- 이 I/O 작업 동안 애플리케이션이 멈추지 않고 다른 작업을 처리할 수 있도록 비차단 방식을 사용

=> **성능 향상, 동시성 지원 ,자원 절약**

비동기 DB 작업과 SQLAlchemy

- SQLAlchemy v1.4 이상부터 비동기 작업을 공식적으로 지원
  - create_async_engine
    - 비동기 데이터베이스 엔진 생성
  - AsyncSession 사용
    - 비동기 세션으로 데이터 베이스 작업 관리
    - I/O 작업마다 await 키워드로 비동기 호출 처리

비동기 DB 작업과 SQLAlchemy

- **비동기 DB 설정**
  - create_async_engine
  - AsyncSession

```python
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from sqlalchemy import Column, Integer, String

# 비동기 SQLite 데이터베이스 URL 설정
DATABASE_URL = "sqlite+aiosqlite:///./test.db"

# 비동기 SQLAlchemy 엔진 생성
async_engine = create_async_engine(
    DATABASE_URL,
    echo=True,  # SQL 쿼리 로그 출력
)

# 비동기 세션 생성
AsyncSessionLocal = sessionmaker(
    bind=async_engine,
    class_=AsyncSession,
    expire_on_commit=False
)

# Base 클래스 생성
Base = declarative_base()
```

비동기 DB 작업과 SQLAlchemy

- **데이터베이스 모델 정의**: 동일하게 설정

```python
class User(Base):
    __tablename__ = "users"  # 테이블 이름 설정

    id = Column(Integer, primary_key=True, index=True)  # 기본 키
    name = Column(String, index=True)  # 이름 컬럼
    email = Column(String, unique=True, index=True)  # 이메일 컬럼 (유니크)
```

비동기 DB 작업과 SQLAlchemy

- **데이터베이스 초기화**
  - 테이블 생성: 동일하게 설정
  - 엔진 초기화: **async with** 사용

```python
async def init_db():
    async with async_engine.begin() as conn:
        await conn.run_sync(Base.metadata.create_all)

# 데이터베이스 초기화 실행
if __name__ == "__main__":
    import asyncio
    asyncio.run(init_db())
```

비동기 DB 작업과 SQLAlchemy

- CRUD 구현: AsyncSession + async + await + commit + refresh

```python
async def create_user(name: str, email: str):
    async with AsyncSessionLocal() as session:
        new_user = User(name=name, email=email)
        session.add(new_user)
        await session.commit()  # 비동기 커밋
        await session.refresh(new_user)  # 데이터 갱신
        return new_user
```

비동기 DB 작업과 SQLAlchemy

- CRUD 구현: AsyncSession + async + await + commit + refresh

```python
# 모든 사용자 조회
async def get_all_users():
    async with AsyncSessionLocal() as session:
        # SQL 쿼리를 text()로 감싸서 실행
        result = await session.execute(text("SELECT * FROM users"))
        users = result.fetchall()
        return users

# 특정 사용자 조회
async def get_user_by_email(email: str):
    async with AsyncSessionLocal() as session:
        # SQL 쿼리를 text()로 감싸서 실행
        result = await session.execute(
            text("SELECT * FROM users WHERE email = :email"),
            {"email": email}  # 파라미터 바인딩
        )
        user = result.fetchone()
        return user
```

비동기 DB 작업과 SQLAlchemy

- CRUD 구현: AsyncSession + async + await + commit + refresh

```python
async def update_user(user_id: int, name: str, email: str):
    async with AsyncSessionLocal() as session:
        user = await session.get(User, user_id)
        if not user:
            return None

        user.name = name
        user.email = email
        await session.commit()
        await session.refresh(user)  # 최신 상태로 동기화
        return user
```

비동기 DB 작업과 SQLAlchemy

- CRUD 구현: AsyncSession + async + await + commit + refresh

```python
async def delete_user(user_id: int):
    async with AsyncSessionLocal() as session:
        user = await session.get(User, user_id)
        if not user:
            return None
        await session.delete(user)
        await session.commit()
        return user_id
```

비동기 DB 작업과 SQLAlchemy

- **비동기 작업 실행**: async + await 사용

```python
import asyncio

async def main():
    # 사용자 생성
    new_user = await create_user(name="Alice", email="alice@example.com")
    print("Created User:", new_user)

    # 모든 사용자 조회
    users = await get_all_users()
    print("All Users:", users)

    # 특정 사용자 조회
    user = await get_user_by_email(email="alice@example.com")
    print("Found User:", user)

    # 사용자 수정
    updated_user = await update_user(user_id=new_user.id, name="Alice Smith", email="alice.smith@example.com")
    print("Updated User:", updated_user)

    # 사용자 삭제
    deleted_id = await delete_user(user_id=new_user.id)
    print("Deleted User ID:", deleted_id)

asyncio.run(main())
```

인증의 중요성 및 JWT 개념

인증의 중요성

- **인증 Authentication 이란?**
  - 클라이언트가 자신이 누구인지 서버에 증명하는 과정
  - 인증 없이는 사용자가 제한된 자원에 접근하거나 민감한 데이터를 보호할 수 없음

인증의 중요성

- **인증이 중요한 이유?**
  - 보안 강화: 사용자 데이터 및 시스템 자원을 보호
  - 권한 관리: 인증을 통해 사용자의 권한을 검증
  - 사용자 경험: 안전한 환경에서 맞춤형 서비스 제공

인증의 중요성

- **인증 vs 인가**
  - 인증 (Who are you?): 사용자가 누구인지 확인
  - 인가 (What you can do?): 사용자가 무엇을 할 수 있는지 확인

JWT 개념

- **JWT(JSON Web Token)란?**
  - 인증 정보를 JSON 형식으로 저장하고, 서명을 통해 위변조를 방지한 토큰
  - 주로 클라이언트와 서버 간의 인증을 위해 사용

JWT 구조

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6ImY1ODg5MGQxOSJ9.eyJhdWQiOiI
4NWEwMzg2Ny1kY2NmLTQ4ODItYWRkZS0xYTc5YWVlYzUwZGYiLCJleHAiOjE2N
DQ4ODQxODUsImlhdCI6MTY0NDg4MDU4NSwiaXNzIjoiYWNtZS5jb20iLCJzdWIiOiI
wMDAwMDAwMC0wMDAwLTAwMDAtMDAwMC0wMDAwMDAwMDAwMDEiL
CJqdGkiOiIzZGQ2NDM0ZC03OWE5LTRkMTUtOThiNS03YjUxZGJiMmNkMzEiLCJhd
XRoZW50aWNhdGlvblR5cGUiOiJQQVNTV09SRCIsImVtYWlsIjoiYWRtaW5AZnVza
W9uYXV0aC5pbyIsImVtYWlsX3ZlcmlmaWVkIjp0cnVlLCJhcHBsaWNhdGlvbklkIjoiO
DVhMDM4NjctZGNjZi00ODgyLWFkZGUtMWE3OWFlZWM1MGRmIiwicm9sZXMiOl
siY2VvIl19.dee-Ke6RzR0G9avaLNRZf1GUCDfe8Zbk9L2c7yaqKME

JWT 구조

출처: https://fusionauth.io/articles/tokens/jwt-components-explained

JWT 장단점

- **장점**
  - 무상태 인증: 서버가 세션을 유지할 필요 없음
  - 확장성: 다양한 서비스에서 쉽게 통합 가능
- **단점**
  - 크기가 크므로 반복적으로 보내는 경우 성능 이슈 발생
  - 만료되지 않은 토큰이 탈취되면 보안 문제가 발생

JWT를 이용하여 /login, /register 구현하기

[실습 2] 전체적인 코드 작성해보기

```python
@app.post("/register")
async def register(
    user: UserRegister,
    db: AsyncSession = Depends(get_db)
):
    result = await db.execute(
        select(User).where(User.username == user.username)
    )
    existing_user = result.scalar_one_or_none()

    if existing_user:
        raise HTTPException(status_code=400, detail="Username already registered")

    new_user = User(
        username=user.username,
        password=hash_password(user.password),
    )

    db.add(new_user)
    await db.commit()

    return {"message": "User registered successfully"}
```

```python
@app.post("/login", response_model=Token)
async def login(
    user: UserLogin,
    db: AsyncSession = Depends(get_db)
):
    result = await db.execute(
        select(User).where(User.username == user.username)
    )
    db_user = result.scalar_one_or_none()

    if not db_user or not verify_password(user.password, db_user.password):
        raise HTTPException(status_code=401, detail="Invalid username or password")

    access_token = create_access_token({"sub": db_user.username})
    return {"access_token": access_token, "token_type": "bearer"}
```

```
project/
├── main.py
├── database.py
├── models.py
├── init_db.py
└── test.db
```

Depends를 사용한 인증 디펜던시

인증 디펜던시 (Depends) 개념

- **인증 디펜던시란?**
  - FastAPI의 Depends를 사용하면 특정 기능 (예: DB 연결, 인증 등)을 경로 함수에서 **반복적으로 재사용**할 수 있음

- **인증 디펜던시가 필요한 이유**
  - 인증 및 사용자 검증 로직을 **코드 중복 없이 모듈화**할 수 있음
  - 인증이 필요한 API 엔드포인트에서 간편하게 적용 가능
  - JWT 토큰을 자동으로 검증하고 사용자 정보를 **API 경로 함수에 전달**할 수 있음

인증 디펜던시의 동작 방식

- **기본적인 인증 흐름**
  1. 클라이언트가 로그인하여 **JWT 토큰**을 발급받음
  2. 클라이언트는 모든 요청의 **Authorization** 헤더에 JWT 토큰을 포함하여 보냄
  3. FastAPI는 **Depends(get_current_user)**를 통해 인증 디펜던시를 실행
  4. JWT 토큰 검증 과정
     a. 토큰이 유효한지 확인: **jwt.decode**
     b. 만료 여부 확인
     c. 사용자가 존재하는지 데이터베이스에서 확인
  5. 인증 성공 시, 해당 사용자 정보를 경로 함수에 전달

인증 디펜던시 구현 방법

- **JWT 설정: Oauth2PasswordBearer(tokenUrl="login")**
  - 클라이언트가 Authorization: Bearer <jWT_TOKEN> 형태로 요청을 보낼 때 추출하는 역할
  - tokenUrl="login"은 로그인 API 엔드포인트를 지정하는 용도

```python
from fastapi.security import OAuth2PasswordBearer
from jose import jwt, JWTError
from fastapi import Depends, HTTPException
from sqlalchemy.ext.asyncio import AsyncSession

# OAuth2 Bearer 토큰 설정
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="login")

# JWT 설정
SECRET_KEY = "your_secret_key"
ALGORITHM = "HS256"
```

인증 디펜던시 구현 방법

- 인증 디펜던시 함수: get_current_user

```python
async def get_current_user(
    token: str = Depends(oauth2_scheme),
    db: AsyncSession = Depends(get_db),
):
    """
    1. 전달된 JWT 토큰을 검증하여 사용자 정보를 가져옴.
    2. JWT가 유효한지 확인하고, 만료 여부를 체크해야 함.
    3. 토큰에서 사용자 정보를 추출한 후, 데이터베이스에서 해당 사용자가 존재하는지 검증해야 함.
    4. 인증 실패 시, 401 Unauthorized 응답을 반환해야 함.
    """
```

인증 디펜던시 구현 방법

- **인증이 필요한 엔드포인트 적용**
  - ex) 프로필 조회 API

```python
@app.get("/profile")
async def profile(current_user: dict = Depends(get_current_user)):
    """
    1. JWT 토큰을 통해 인증된 사용자만 접근할 수 있는 API
    2. get_current_user() 디펜던시를 사용하여 현재 로그인된 사용자 정보 가져옴
    3. 사용자 정보가 유효하면 프로필 데이터 반환
    """
    return {
        "username": current_user["username"]
    }
```

인증 디펜던시 장점

- 코드 재사용성 증가
  - Depends(get_current_user)를 활용하면 API마다 중복된 인증 코드 작성이 불필요
  - 여러 API에서 일관된 방식으로 인증 적용 가능
- 보안성 강화
  - JWT 토큰을 중앙에서 검증: 일괄적인 보안 적용 가능
  - 모든 요청마다 get_current_user를 호출하여 인증을 검증
- 유지보수 용이
  - 인증 방식이 변경되더라도, get_current_user만 수정하면 전체 API에 반영됨
  - 관리가 쉽고, 확장성이 높음

Depends를 사용한 인증 디펜던시

[실습 3] get_current_user를 완성하여 profile API가 잘 작동하도록 구성해주세요.

```python
async def get_current_user(
    token: str = Depends(oauth2_scheme),
    db: AsyncSession = Depends(get_db),
):
    """
    1. 전달된 JWT 토큰을 검증하여 사용자 정보를 가져옴.
    2. JWT가 유효한지 확인하고, 만료 여부를 체크해야 함.
    3. 토큰에서 사용자 정보를 추출한 후, 데이터베이스에서 해당 사용자가 존재하는지 검증해야 함.
    4. 인증 실패 시, 401 Unauthorized 응답을 반환해야 함.
    """
```

Depends를 사용한 인증 디펜던시

[실습 3] get_current_user를 완성하여 profile API가 잘 작동하도록 구성해주세요.

- **힌트**
  - Step 1: JWT 토큰 검증
    - jwt.decode()를 사용하여 토큰이 유효한지 확인
  - Step 2: 사용자 정보 추출
    - JWT의 sub 필드에서 사용자 명을 가져옴
  - Step 3: 데이터베이스에서 사용자 검증
    - 데이터베이스에서 해당 사용자명이 존재하는지 확인
  - Step 4: 인증된 사용자 정보 반환

Depends를 사용한 인증 디펜던시

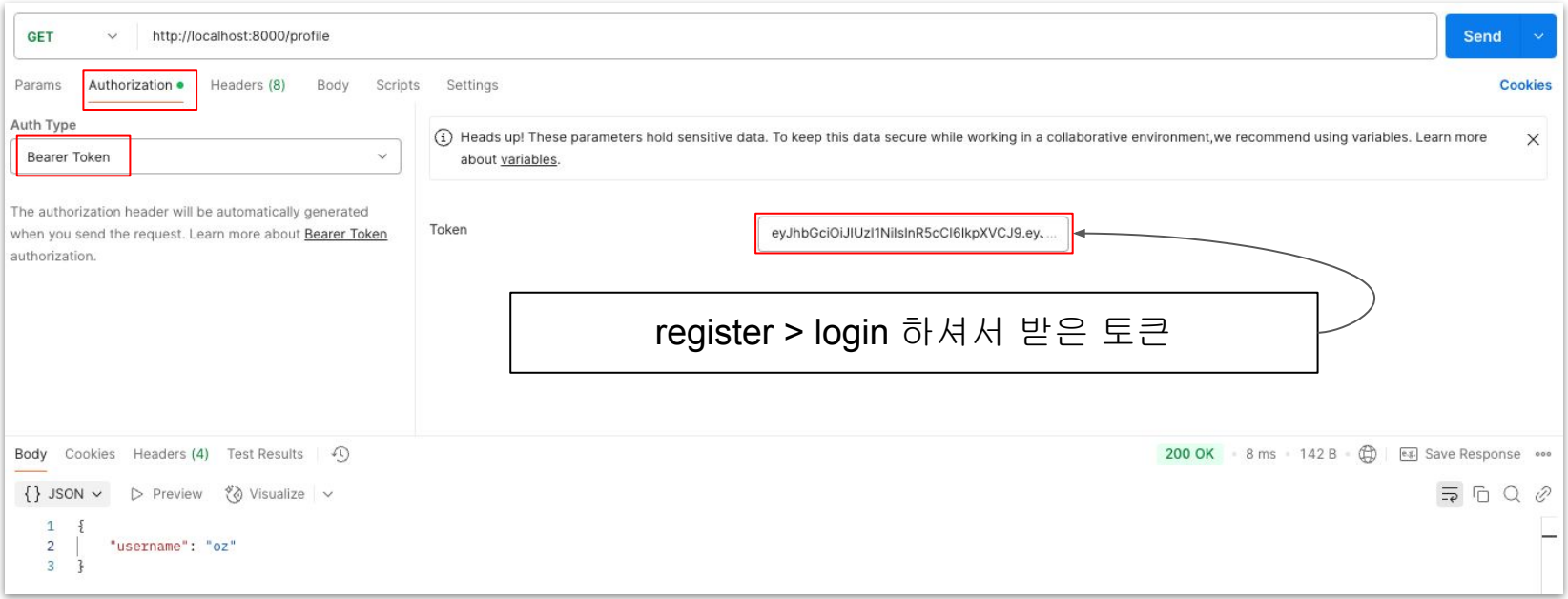
[실습 3] get_current_user를 완성하여 profile API가 잘 작동하도록 구성해주세요.

- /profile API는 아래의 코드 그대로 사용 가능

```python
@app.get("/profile")
async def profile(current_user: dict = Depends(get_current_user)):
    """
    1. JWT 토큰을 통해 인증된 사용자만 접근할 수 있는 API
    2. get_current_user() 디펜던시를 사용하여 현재 로그인된 사용자 정보 가져옴
    3. 사용자 정보가 유효하면 프로필 데이터 반환
    """
    return {
        "username": current_user["username"]
    }
```

Depends를 사용한 인증 디펜던시

[실습 3] get_current_user를 완성하여 profile API가 잘 작동하도록 구성해주세요.



QnA