FastAPI 특강

4일차: 실시간 기능 (웹소켓)

목 차

- 3일차 복습
- WebSocket 웹소켓
  - 실습 1: WebSocket 기본 세팅
  - 실습 2: 실시간 시스템 리소스 모니터링
  - 실습 3: 실시간 숫자 업다운 게임
  - 실습 4: 개인 실시간 접속 환영 인사
  - 실습 5: 실시간 단체 채팅방 (Broadcasting)

3일차 복습

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

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
```

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

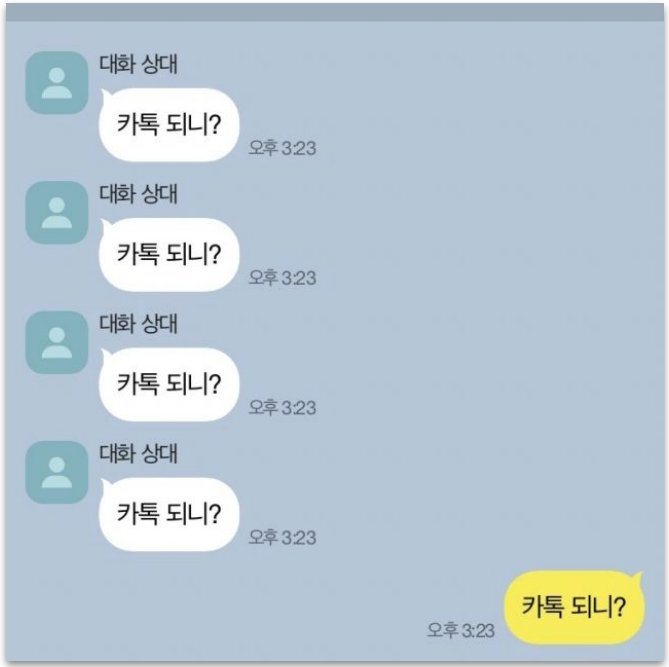
```python
@app.get("/profile")
async def profile(current_user: dict = Depends(get_current_user)):
    """
    1. JWT 토큰을 통해 인증된 사용자만 접근할 수 있는 API
    2. get_current_user() 디펜던시를 사용하여 현재 로그인된 사용자 정보 가져옴
    3. 사용자 정보가 유효하면 프로필 데이터 반환
    """
    return {
        "username": current_user["username"]
 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

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

웹 소켓
WebSocket

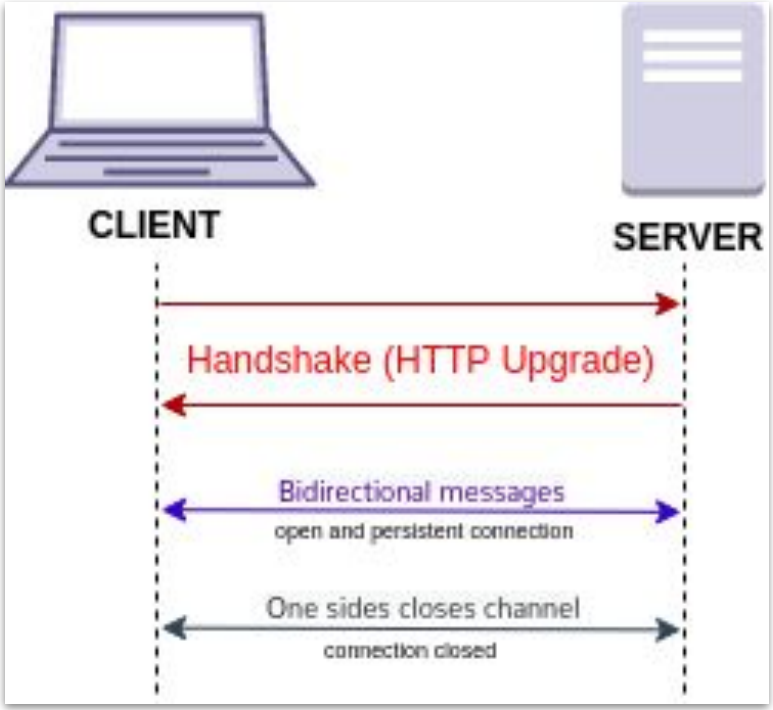
WebSocket이란



HTTP vs WebSocket

특징	HTTP	WebSocket
연결 방식	요청-응답 방식	지속적인 연결 유지
통신 방식	클라이언트가 요청하면 서버가 응답	서버와 클라이언트가 자유롭게 송수신 가능
활용 사례	REST API, 정적 페이지 요청	실시간 채팅, 알림 시스템, 주식 데이터 스트리밍

WebSocket 프로토콜 구조



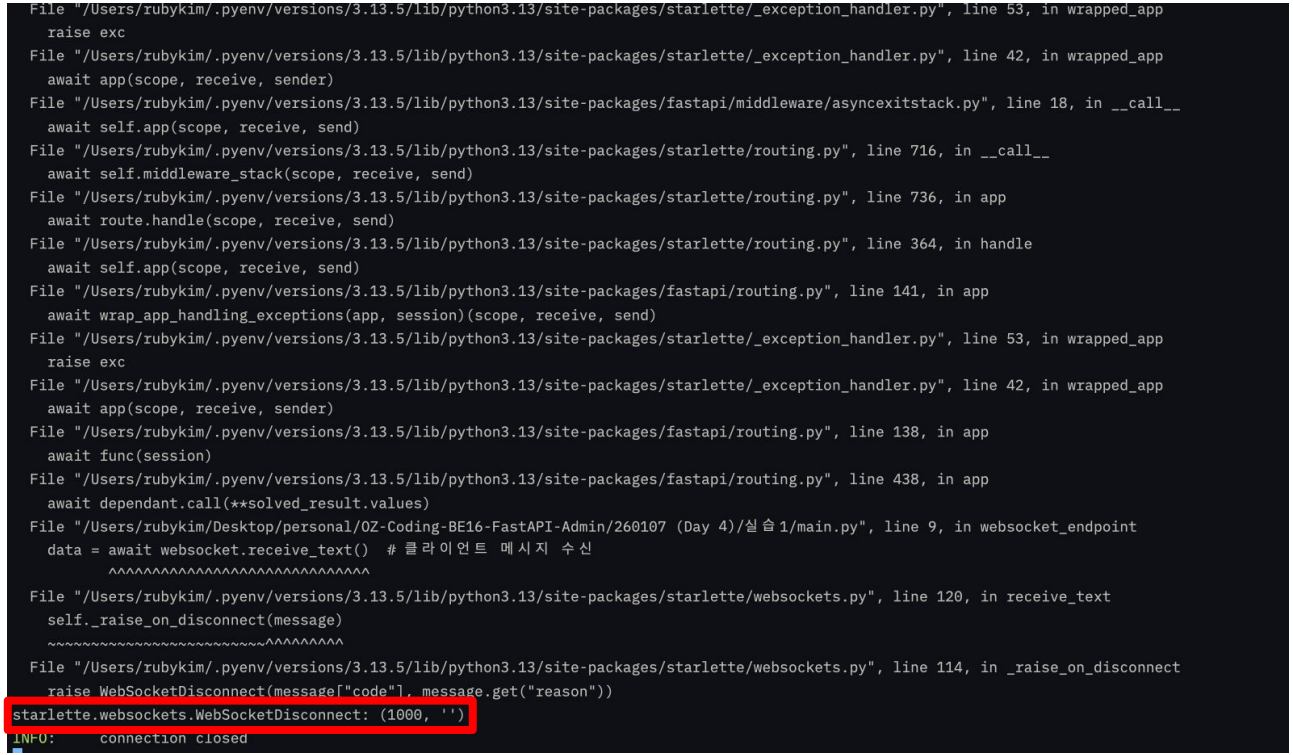
WebSocket 서버 구현

```python
from fastapi import FastAPI, WebSocket

app = FastAPI()

@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()  # WebSocket 연결 수락
    while True:
        data = await websocket.receive_text()  # 클라이언트 메시지 수신
        await websocket.send_text(f"서버 응답: {data}")  # 클라이언트에게 응답
```

WebSocket 서버 구현 - Disconnect 1000



WebSocket 서버 구현 - Disconnect 1000

- Starlette.websockets.WebSocketDisconnect: (1000, ‘’)
  - 에러코드 1000: WebSocket이 정상적으로 종료됨
  - 클라이언트(예: Postman, 브라우저)가 WebSocket 연결을 닫으면 서버에서 발생하는 오류
- 해결 방법: try-except을 추가하여 예외 처리
  - fastapi에서 WebSocketDisconnect을 import 후, except 처리

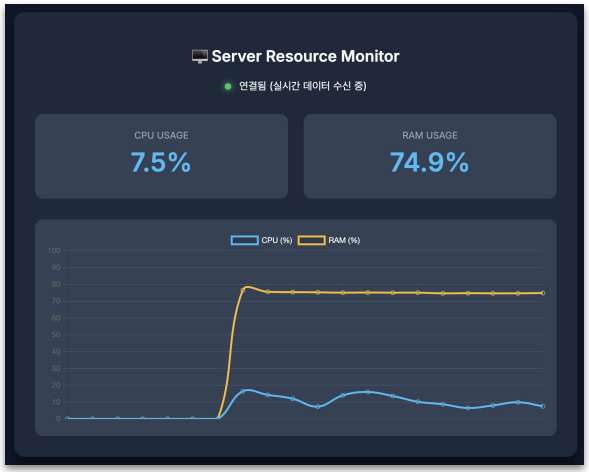
[실습 1] WebSocket 서버 구현

이미지를 참고하여 웹소켓을 구현 후 Postman에서 테스트 해보세요. 이 때 try-except으로 WebSocketDisconnect 에러를 깨끗하게 처리해주세요.

Postman에서 Disconnect를 하면 WebSocketDisconnect에러가 나오지 않아야 합니다.

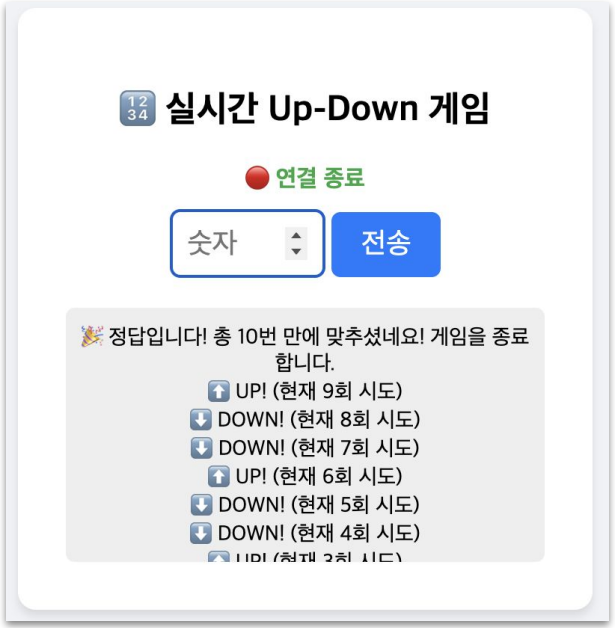
[실습 2] 실시간 시스템 리소스 모니터링

- 개념
  - 클라이언트: 웹소켓 연결 후 서버가 보내주는 리소스 데이터 기다림
  - 서버: 연결된 클라이언트에게 1초마다 서버의 CPU + RAM 사용량 데이터를 자동 전송 -> psutil 라이브러리를 통해 시스템 자원 정보 취득 -> JSON형식으로 가공하여 송신
  - 클라이언트: 수신된 JSON 데이터 해석 및 화면 표시



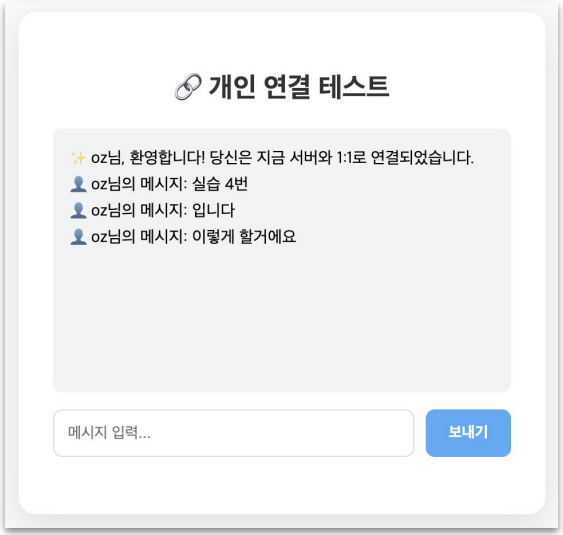
[실습 3] 실시간 숫자 업다운 게임

- 개념
  - 클라이언트: 서버에 웹소켓 연결 후 정답이라는 숫자를 전송
  - 서버: 연결 시점에서 각 클라이언트마다 독립적인 숫자와 시도 횟수를 생성 및 초기화 -> 실시간으로 판별하여 메시지 피드백
  - 클라이언트: 서버의 피드백에 따라 숫자를 다시 입력 + 정답을 맞춰감



[실습 4] 개인 실시간 접속 환영 인사

- 개념
  - 클라이언트: URL 경로에 본인 닉네임을 포함하여 서버 연결 요청
  - 서버: 경로 매개변수로 클라이언트의
    이름 인식 및 연결 수락
    -> 개인 맞춤형 환영 메시지 전송
  - 클라이언트: 본인의 이름이 포함된 메시지를
    실시간으로 주고받으며 서버와 소통



[실습 5] 실시간 단체 채팅방 (Broadcasting)

- 개념
  - 클라이언트: 닉네임을 설정한 후 Websocket 연결 시도 및 메시지 전송
  - 서버: ConnectionManager로 소켓 세션 관리 + 모든 접속자에게 메시지 전송
  - 클라이언트: 서버로부터 받은 JSON 데이터 분석 후 표시

QnA