FastAPI 특강

4일차: FastAPI에서의 고급 Pydantic 및 요청/응답 처리

목 차

- 3일차 복습
- Pydantic 고급 기능
- FastAPI와 Pydantic 고급 기능 결합

3일차 복습

좋은 API 설계법

- API 설계 원칙: **일관성**, 엔드포인트 설계, HTTP 상태 코드
  - API 사용자가 엔드포인트와 데이터 구조를 예측 가능하게 사용하도록 구성

```
GET /users              // 사용자 목록 조회
GET /users/123          // 특정 사용자 조회
POST /users             // 사용자 생성
DELETE /users/123       // 사용자 삭제
```



```
GET /getUsers           // 사용자를 조회
GET /fetchUser/123      // 특정 사용자 조회
POST /newUser           // 사용자 생성
DELETE /deleteUser/123 // 사용자 삭제
```

좋은 API 설계법

- API 설계 원칙: 일관성, **엔드포인트 설계**, HTTP 상태 코드
  - 리소스를 명확히 나타내야 하며, 동작이 아닌 데이터를 중심으로 설계하도록 구성
  - 설계 원칙
    - 필터링, 정렬, 페이징 지원 (= 성능 최적화)

```
GET /products?category=shirts          // 필터링
GET /products?sort=price&order=asc     // 정렬
GET /products?page=2&limit=20          // 페이징
```

좋은 API 설계법

- API 설계 원칙: 일관성, 엔드포인트 설계, **HTTP 상태 코드**
  - 상태 코드를 통해 요청의 결과를 명확히 전달

https://developer.mozilla.org/ko/docs/Web/HTTP/Status

클래스	설명	대표 상태 코드
2xx	요청이 성공적으로 처리됨	200 OK 201 Created 204 No Content
3xx	Redirection. 요청 완료를 위해 추가 작업이 필요함	301 Moved Permanently 302 Found 304 Not Modified
4xx	Client Error 클라이언트의 잘못된 요청으로 서버가 처리할 수 없음	400 Bad Request 401 Unauthorized 403 Forbidden 404 Not Found
5xx	Server Error 서버가 요청을 처리하던 도중 오류 발생	500 Internal Server Error 502 Bad Gateway 503 Service Unavailable 504 Gateway Timeout

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

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

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

- 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.
    """
```

3일차 실습 복습

예시: Postman.json 참고

[복습] 아래 3개의 API를 설계하고 구현하세요.

- **요구사항**
  - 2일차 실습 3번 코드에서 에서 계속 진행
  - 아래 구현할 모든 API를 사용하기 위해서는 사전에 register 및 login이 되어야 함
  - API 설계 원칙을 모두 지킬 필요는 없음. **중요한 것은 "FastAPI의 인증 및 인가 엔드포인트 설계"임**

- **엔드포인트**
  - 쉬움: 현재 로그인한 사용자의 토큰 만료 시간을 가져오는 `GET /token-expiry` API
  - 보통: 관리자만 접근 가능한 `GET /admin` API
  - 어려움: 사용자의 프로필을 수정하는 `POST /profile` API

3일차 실습 복습

[복습] 아래 3개의 API를 설계하고 구현하세요.

- **쉬움**: `GET /token-expiry` API
  - 유효한 토큰이면 만료 시간을 반환
  - 토큰이 만료되었으면 401 Unauthorized 반환
  - 힌트: token세팅 시 Depends(oauth2_scheme) 설정 필요

```
{
  "username": "testuser",
  "expires_at": "2025-01-30T03:54:56Z"
}
```

3일차 실습 복습

[복습] 아래 3개의 API를 설계하고 구현하세요.

- **보통**: `GET /admin` API
  - role = “admin”만 접근 가능하도록 설정
  - 일반 사용자가 접근하면 403 Forbidden 반환
  - role은 fake_users_db에 role필드를 추가하여 관리

```
fake_users_db = {
    "admin": {"username": "admin", "password": pwd_context.hash("adminpass"), "role": "admin"},
    "testuser": {"username": "testuser", "password": pwd_context.hash("testpass"), "role": "user"}
}
```

```
{
  "message": "Welcome, admin!"
}
```

3일차 실습 복습

[복습] 아래 3개의 API를 설계하고 구현하세요.

- **어려움**: `POST /profile` API
  - 사용자가 자신의 프로필**만** 수정할 수 있어야 함
    - username 변경 가능
    - password 변경 불가능

```
{
  "message": "Profile updated successfully",
  "updated_user": {
    "username": "newusername"
  }
}
```

Pydantic 고급 기능

Pydantic 기본 개념 및 모델 정의

기본 데이터 모델 정의: 성공 시 (200)

코드 정의

```python
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    price: float
    is_offer: bool = False
```

```python
@app.post("/items/")
def create_item(item: Item):
    return {"item": item}
```

요청 JSON

```json
{
  "name": "Book",
  "price": 12.99
}
```

응답 JSON

```json
{
    "item": {
        "name": "Book",
        "price": 12.99,
        "is_offer": false
    }
}
```

Pydantic 의 validator 개념 및 사용법

validator의 기본 사용법

- 구조
  - `@field_validator("<필드 이름>")`, `@classmethod` 데코레이터를 사용하여 특정 필드의 값을 검증
  - 검증 함수는 항상 클래스 메서드 형태로 정의되며, 첫 번째 인자로 필드값을 받음
  - 검증 실패 시 예외를 발생시켜야 함: ValueError, TypeError 등

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
```

Pydantic 의 validator 개념 및 사용법

validator의 기본 사용법

- 동작
  - 모델 초기화 시, 해당 필드의 값이 검증 로직을 통과해야 함
  - 통과하지 못하면 예외 메시지가 반환됨

```
example = ExampleModel(number=50)
print(example)
# Output: number=50
```

```
example = ExampleModel(number=150)
# ValueError: Number must not exceed 100
```

더 복잡한 데이터 검증이 필요한 이유

- 사용자가 입력하는 데이터가 항상 올바르지는 않음
- **보안 문제**: 잘못 입력된 SQL Injection, XSS 등 보안 취약점을 유발할 수 있음
- **비즈니스 로직 보장**: 이메일 형식. 전화번호 국가별 형식, 나이 제한 등

```
{
  "email": "wrong_format",
  "phone_number": "abcdefg"
}
```

고급 Pydantic 검증

- 단일 필드 검증을 넘어 필드 간 관계를 고려한 검증이 필요
- 실제 서비스에서 발생할 수 있는 논리적 오류를 방지
- 보안, 데이터 정합성을 보장하기 위해 더 정교한 검증이 필요

고급 Pydantic 검증: 비밀번호 고급 검증

[실습 1] 비밀번호 고급 검증을 할 수 있는 UserRegister 모델을 설계하세요.

- username: str
- password: str
- 비밀번호 조건
  - 길이가 8 이상일 것
  - 최소 1개 이상의 대문자, 숫자, 특수문자가 있을 것

고급 Pydantic 검증: 비밀번호 고급 검증

[실습 1] 비밀번호 고급 검증을 할 수 있는 UserRegister 모델을 설계하세요.

```
try:
    weak_user = UserRegister(username="weakuser", password="weakpass")
    strong_user = UserRegister(username="stronguser", password="Strong@pass1")
except ValidationError as e:
    print(e)
```

고급 Pydantic 검증: 여러 필드 검증

- 여러 필드를 확인할 때 사용
  - 회원가입 시 비밀번호 2번 입력
  - 비밀번호 변경시 비밀번호 2번 입력
  - 특정 필드가 존재하면 다른 필드의 유효성을 달리 적용하는 경우
    - email이 없을 경우, phone_number는 반드시 입력해야 함
- 여러 필드를 한 번에 검증해야 하므로, `@model_validator`을 사용해야함

고급 Pydantic 검증: 여러 필드 검증

```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
```

고급 Pydantic 검증: 여러 필드 검증

Mode	실행 시점	데이터 접근 방식	예제
before	데이터 유효성 검증 전에 실행	dict 형태로 원시 데이터 접근	입력 데이터 반환 (소문자 변환, 공백 제거 등) 타입 변환 (str -> int)
after	데이터 유효성 검증 후 실행	self 객체를 사용해 필드 값 접근	여러 필드 간 검증 값의 논리적 검증 비즈니스 로직 반영 (role 기반 접근 제한)

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: 여러 필드 검증

[실습 2] email 또는 phone_number 중 하나 이상은 필수로 입력해야 하는 모델을 설계하세요.

- **요구사항**
  - email 또는 phone_number 중 하나 이상 필수 입력
  - 이메일은 올바른 이메일 형식 사용 xxx@xxx.xxx
    - 만약 이메일 내에 대문자가 있을 시 소문자로 변환해야 함

```python
class ContactInfo(BaseModel):
    email: str | None = None
    phone_number: str | None = None
```

점심 & 쉬는 시간

고급 Pydantic 검증: 자동 계산 필드

- 입력값 기반으로 자동 계산되는 필드
  - 나이 대신 생년월일을 입력받고 자동으로 계산
  - 정가를와 할인률을 입력하면 할인가를 자동으로 계산

- `@computed_field`를 사용해야함

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: 자동 계산 필드

[실습 3] price(원가)와 discount(할인률)를 입력하면 자동으로 final_price(할인가)를 계산하도록 설계하세요.

- **요구사항**
  - 필드: name(str), price(float), discount(float)
  - 자동으로 final_price 필드 계산
  - discount가 입력되지 않으면 기본값 0%. 할인률은 0~100% 사이
  - final_price는 소수점 둘째자리에서 반올림 = 결과값이 xxx.x 로 나와야 함

고급 Pydantic 검증: 자동 계산 필드

[실습 3] price(원가)와 discount(할인률)를 입력하면 자동으로 final_price(할인가)를 계산하도록 설계하세요.

```
product1 = Product(name="Laptop", price=1000, discount=10)
print(product1.final_price)  # 900.0 (10% 할인 적용)

product2 = Product(name="Smartphone", price=500)  # 할인율 기본값 0%
print(product2.final_price)  # 500.0 (할인 없음)
```

고급 Pydantic 검증: 초기값 동적 설정

- 초기값을 동적으로 설정할 때 사용
  - 자동 증가 ID
  - created_at을 현재 시간으로 설정
  - verification code를 위해 6개의 랜덤숫자 설정

- `default_factory`를 사용해야함

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
```

고급 Pydantic 검증: 초기값 동적 설정

[실습 4] 회원가입 시 user_id를 입력하지 않으면 자동으로 UUID를 생성하도록 설계하세요.

- **요구사항**
  - 필드: user_id(str), name(str), role(str), created_at(str)
  - user_id를 입력하지 않으면 자동으로 UUID 생성
  - created_at(가입 날짜)는 현재 날짜(YYYY-MM-DD HH:MM:SS)로 자동 설정
  - role필드는 기본값 “user” 설정

고급 Pydantic 검증: 초기값 동적 설정

[실습 4] 회원가입 시 user_id를 입력하지 않으면 자동으로 UUID를 생성하도록 설계하세요.

```
user1 = User(name="Alice")
print(user1)

user2 = User(name="Bob", role="admin")  # role 변경 가능
print(user2)
```

```
user_id='4e3f528c-cc4a-4c91-bf58-768b87f7b7d6' name='Alice' role='user' created_at='2025-01-30 14:30'
user_id='b29a7b98-5c94-4a64-a5d8-983c5d65c5c3' name='Bob' role='admin' created_at='2025-01-30 14:30'
```

고급 Pydantic 검증: 초기값 동적 설정

[실습 5] OTP (일회용 비밀번호) 자동 생성하도록 설계하세요.

- **요구사항**
  - 필드: phone_number(str), otp(int), otp_expiry(str)
  - 사용자가 phone_number를 입력하면 OTP(6자리 랜덤 숫자) 자동 생성
  - otp_expiry 필드는 현재 시간 기준 5분 후로 자동 설정
    - YYYY-MM-DD HH:MM:SS
- **힌트**
  - 6자리 숫자 랜덤은 import random 이용

고급 Pydantic 검증: 초기값 동적 설정

[실습 5] OTP (일회용 비밀번호) 자동 생성하도록 설계하세요.

```
otp1 = OTPVerification(phone_number="010-1234-5678")
print(otp1)

otp2 = OTPVerification(phone_number="010-5678-1234")
print(otp2)
```

```
phone_number='010-1234-5678' otp=562918 otp_expiry='2024-02-01 14:35:30'
phone_number='010-5678-1234' otp=923715 otp_expiry='2024-02-01 14:35:30'
```

고급 Pydantic 검증: 여러 타입 지원

- 특정 필드에 여러 타입 지원
  - product_id가 int또는 str일 경우
  - score가 int또는 float일 경우
- `Union`을 사용해야함

고급 Pydantic 검증: 여러 타입 지원

```python
from pydantic import BaseModel
from typing import Union

class Product(BaseModel):
    product_id: Union[int, str]  # 숫자 또는 문자열 가능
    name: str

# 테스트
p1 = Product(product_id=123, name="Laptop")
p2 = Product(product_id="XYZ-456", name="Phone")

print(p1.product_id)  # 123
print(p2.product_id)  # "XYZ-456"
```

고급 Pydantic 검증: 여러 타입 지원

[실습 6] User모델이 다음의 요구사항을 만족할 수 있도록 설계하세요.

- **요구사항**
  - 필드: username(str), phone_number(str, None), score(int, float)
  - phone_number가 None이면 “No phone number”로 자동 설정

```
user1 = User(username="Alice", phone_number="010-1234-5678", score=90)
print(user1)

user2 = User(username="Bob", score=89.5)  # phone_number 미입력
print(user2)
```

```
username='Alice' phone_number='010-1234-5678' score=90
username='Bob' phone_number='No phone number' score=89.5
```

고급 Pydantic 검증: 직렬화

- **직렬화 Serialization**: 다양한 종류의 데이터를 기계가 쓰고 읽기 편리하게 나타낸 형식
  - 기본적으로 datetime, Decimal 등은 JSON으로 직렬화할 수 없음
  - API응답에서 날짜 형식을 통일하거나 소수점 자리수 제한할 때 필요
- `json_encoders`를 사용해야함

고급 Pydantic 검증: 직렬화

```python
from pydantic import BaseModel
from datetime import datetime

class Order(BaseModel):
    order_id: int
    total_price: float
    created_at: datetime

    class Config:
        json_encoders = {
            datetime: lambda v: v.strftime("%Y-%m-%d %H:%M:%S")
        }

# 테스트
order = Order(order_id=1, total_price=100.5, created_at=datetime.now())
print(order.model_dump_json())
```

고급 Pydantic 검증: 여러 타입 지원

[실습 7] Boolean 값을 Yes 또는 No로 변환하도록 설계하세요.

- 요구사항
  - 필드: username(str), is_active(bool)

```python
user1 = User(username="Alice", is_active=True)
print(user1.model_dump_json())  # JSON 변환

user2 = User(username="Bob", is_active=False)
print(user2.model_dump_json())  # JSON 변환
```

```
{
  "username": "Alice",
  "is_active": "Yes"
}
{
  "username": "Bob",
  "is_active": "No"
}
```

FastAPI와 Pydantic 고급 기능 결합

응답 필터링 (response_model_exclude)

- API 응답에서 **민감한 정보(비밀번호, API Key 등)를 숨기고 싶을 때**
- 유저 권한에 따라 **보여줄 데이터 조절**
  - admin은 모든 필드
  - 일반 유저는 일부 필드
- `response_model_exclude`를 사용해야함

응답 필터링 (response_model_exclude)

```python
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class User(BaseModel):
    username: str
    email: str
    password: str


@app.get("/users/me", response_model=User, response_model_exclude={"password"})
async def get_user():
    user_data = {"username": "testuser", "email": "test@example.com", "password": "mypassword"}
    return User(**user_data)  # `response_model_exclude`로 비밀번호 자동 제거
```

GET `/users/me`

```json
{
  "username": "testuser",
  "email": "test@example.com"
}
```

JSON & XML 응답 포맷 변환

- API가 JSON 뿐만 아니라 XML도 지원해야할 때
- 사용자가 원하는 응답 포맷을 동적으로 변경하도록 설정
- `Response` 객체 및 `xml.etree.ElementTree`를 사용해야함

JSON & XML 응답 포맷 변환

```python
from fastapi import FastAPI
from fastapi.responses import JSONResponse, Response
import xml.etree.ElementTree as ET

app = FastAPI()

@app.get("/data/")
async def get_data(format: str = "json"):
    data = {"message": "Hello, FastAPI!"}

    if format == "xml":
        root = ET.Element("response")
        message = ET.SubElement(root, "message")
        message.text = data["message"]
        xml_str = ET.tostring(root, encoding="utf-8", method="xml")
        return Response(content=xml_str, media_type="application/xml")

    return JSONResponse(content=data)
```

GET `/data/?format=xml`

```xml
<response>
    <message>Hello, FastAPI!</message>
</response>
```

다국어 응답 처리 (Accept-Language)

- 글로벌 API에서는 사용자의 언어에 따라 다른 메시지 반환 필요
- 브라우저 요청 헤더 `Accept-Language`를 사용하여 자동 감지
- 지원하는 언어가 없으면 기본값을 설정해야 함

다국어 응답 처리 (Accept-Language)

```python
from fastapi import FastAPI, Header

app = FastAPI()

responses = {
    "en": {"message": "Hello, welcome!"},
    "ko": {"message": "안녕하세요, 환영합니다!"},
    "fr": {"message": "Bonjour, bienvenue!"}
}

@app.get("/greet/")
async def greet(accept_language: str = Header("en")):
    return responses.get(accept_language, responses["en"])  # 기본값 en
```

```
GET /greet/
GET /greet/ (Accept-Language=ko)
GET /greet/ (Accept-Language=fr)
```

```
{
  "message": "안녕하세요, 환영합니다!"
}
```

고급 Pydantic + FastAPI 실습

[실습 8] 가이드에 맞춰 지금까지의 내용들을 모두 구현해보세요.

- **프로그램: 온라인 쇼핑몰 API**
  - 사용자 정보 조회
  - 주문 조회
  - 상품 조회
  - 사용자 리스트 반환 (json, xml)
  - 다국어 응답(인사) 지원

고급 Pydantic + FastAPI 실습

[실습 8] 가이드에 맞춰 지금까지의 내용들을 모두 구현해보세요.

- **파일 구조**

  📂 **project/**

  ├── **`main.py`** FastAPI 서버: API 라우트 정의

  ├── **`models.py`** Pydantic 데이터 모델

  ├── **`utils.py`** 다국어 처리 및 응답 변환 함수

  └── **`requirements.txt`** (선택) 프로젝트에서 필요한 패키지 목록

고급 Pydantic + FastAPI 실습

[실습 8] 가이드에 맞춰 지금까지의 내용들을 모두 구현해보세요.

- **`models.py`** - User 모델 (사용자 정보)

필드명	타입	설명	추가 조건
user_id	str	사용자 고유 ID	UUID 자동 생성
username	str	사용자 이름	필수 입력
email	str	이메일 주소	필수 입력
password	str	비밀번호	응답에서 숨김 처리
is_active	bool	활성화 여부	Yes / No로 변환

고급 Pydantic + FastAPI 실습

[실습 8] 가이드에 맞춰 지금까지의 내용들을 모두 구현해보세요.

- **`models.py`** - Order 모델 (주문 정보)

필드명	타입	설명	추가 조건
order_id	int	주문 ID	필수 입력
username	str	주문한 사용자 이름	필수 입력
total_price	Decimal	주문 총 금액	소수점 2자리 제한
is_paid	bool	결제 여부	Yes / No 변환
created_at	datetime	주문 생성 날짜	YYYY-MM-DD HH:MM:SS 형식으로 변환

고급 Pydantic + FastAPI 실습

[실습 8] 가이드에 맞춰 지금까지의 내용들을 모두 구현해보세요.

- **`models.py`** - Product 모델 (상품 정보)

필드명	타입	설명	추가 조건
product_id	str	상품 ID	필수 입력
name	str	상품명	필수 입력
price	float	원가	필수 입력
discount	float	할인율 (%)	기본값 0%
final_price	float	할인 적용가	자동 계산

고급 Pydantic + FastAPI 실습

[실습 8] 가이드에 맞춰 지금까지의 내용들을 모두 구현해보세요.

- **`utils.py`**
  - Accept-Language 헤더를 읽어 다국어 응답 반환
  - JSON 데이터를 XML 형식으로 변환하는 함수 추가
  - 다국어 지원: en, ko, fr, de, es

고급 Pydantic + FastAPI 실습

예시: Postman.json 참고

[실습 8] 가이드에 맞춰 지금까지의 내용들을 모두 구현해보세요.

- **utils.py**

```
LANGUAGES = {
    "en": {"message": "Hello, welcome!"},
    "ko": {"message": "안녕하세요, 환영합니다!"},
    "fr": {"message": "Bonjour, bienvenue!"},
    "de": {"message": "Hallo, willkommen!"},
    "es": {"message": "¡Hola, bienvenido!"}
}
```

고급 Pydantic + FastAPI 실습

[실습 8] 가이드에 맞춰 지금까지의 내용들을 모두 구현해보세요.

- **`main.py`**
  - FastAPI 인스턴스 생성
  - DB는 fake_db를 사용 (sqlite같은 DB사용 x)
    - fake_users_db
    - fake_orders_db
    - fake_products_db

고급 Pydantic + FastAPI 실습

[실습 8] 가이드에 맞춰 지금까지의 내용들을 모두 구현해보세요.

- `main.py`

```python
# 가짜 사용자 데이터
fake_users_db = {
    "alice": {"username": "Alice", "email": "alice@example.com", "password": "secret123"},
    "bob": {"username": "Bob", "email": "bob@example.com", "password": "password456"},
}

# 가짜 주문 데이터
fake_orders_db = {
    1: {"order_id": 1, "username": "Alice", "total_price": 49.99, "is_paid": True},
    2: {"order_id": 2, "username": "Bob", "total_price": 99.99, "is_paid": False},
}

# 가짜 상품 데이터
fake_products_db = {
    "p1": {"product_id": "p1", "name": "Laptop", "price": 1000.0, "discount": 10},
    "p2": {"product_id": "p2", "name": "Phone", "price": 500.0, "discount": 5},
}
```

고급 Pydantic + FastAPI 실습

[실습 8] 가이드에 맞춰 지금까지의 내용들을 모두 구현해보세요.

- **`main.py`**
  - 사용자 정보 조회 API: `GET /users/{username}`
    - 보안상의 이유로 비밀번호는 응답에서 숨김 처리
    - response_model_exclude를 활용한 필드 제외 기능 적용

고급 Pydantic + FastAPI 실습

[실습 8] 가이드에 맞춰 지금까지의 내용들을 모두 구현해보세요.

- **`main.py`**
  - 주문관리 API: `GET /orders/{order_id}`
    - 사용자가 주문한 내역을 확인하는 API
    - total_price는 소수점 2자리까지 반환
    - is_paid는 Boolean값을 Yes 또는 No로 변환
    - created_at은 YYYY-MM-DD HH:MM:SS 형식으로 변환

고급 Pydantic + FastAPI 실습

[실습 8] 가이드에 맞춰 지금까지의 내용들을 모두 구현해보세요.

- **`main.py`**
  - 상품 조회 API: `GET /products/{product_id}`
    - 상품을 조회하는 API
    - 할인율을 적용한 final_price를 자동 계산
    - 할인율이 100%일 경우 FREE라는 문자열 반환

고급 Pydantic + FastAPI 실습

[실습 8] 가이드에 맞춰 지금까지의 내용들을 모두 구현해보세요.

- **`main.py`**
  - 사용자 리스트 반환 API: `GET /users/`
    - JSON, XML 응답 형식을 지원하는 API
    - query_params(?format=json, ?format=xml)를 통해 응답 포맷 변경 가능
    - XML 변환을 위해 xml.etree.ElementTree 활용

고급 Pydantic + FastAPI 실습

[실습 8] 가이드에 맞춰 지금까지의 내용들을 모두 구현해보세요.

- **`main.py`**
  - 다국어 지원 API: `GET /greets/`
    - API 응답을 사용자의 언어에 맞게 제공
    - Accept-Language 헤더를 기반으로 다국어(en, ko, fr, de, es) 지원

QnA