FastAPI 특강

2일차: Pydantic 기본 + 심화

목 차

- 1일차 복습
- Pydantic 기본 (2)
- Pydantic 심화

1일차 복습

FastAPI란?

- Python으로 RESTful API를 쉽고 빠르게 개발할 수 있는 **모던 웹 프레임워크**

- 비동기(Asynchronous) 기반으로 설계되어 높은 성능 제공

- 고성능: Starlette와 Pydantic을 기반으로 높은 처리 속도 제공

FastAPI 특징

자동화된 API 문서화

- Swagger UI와 ReDoc: API를 작성하면 OpenAPI 명세가 자동 생성됨
  - `/docs`: Swagger UI
  - `/redoc`: ReDoc 기반 API 문서

- 문서화 작업 없이 클라이언트와 개발자 간의 협업을 빠르게 지원

FastAPI 특징

비동기 처리 (Asynchronous Programming)

- async/await를 기본적으로 지원: 높은 요청 처리량 보장
- 동시 요청 처리에 유리: DB 쿼리, 외부 API 호출 같은 I/O 작업에서 효율적

데이터 검증 및 직렬화

- Pydantic 모델을 통해 데이터 검증과 직렬화를 자동 처리
- 클라이언트 요청(Request) 데이터를 Python 객체로 변환
  + 응답(Response) 데이터를 JSON으로 자동 직렬화

FastAPI 경로 매개변수 (Path Parameters)

- 경로 매개변수란?
  - URL 경로의 일부로 동적인 값을 전달
  - 예: `/items/9`에서 9는 동적인 값

```python
@app.get("/items/{item_id}")
def read_item(item_id: int):
    return {"item_id": item_id}
```

FastAPI 쿼리 매개변수 (Query Parameters)

- 쿼리 매개변수란?
  - URL에서 `?key=value` 형식으로 전달되는 값
  - 경로 매개변수와 달리 선택적으로 사용할 수 있음

```python
@app.get("/items/")
def read_items(skip: int = 0, limit: int = 10):
    return {"skip": skip, "limit": limit}
```

FastAPI 경로 매개변수 + 쿼리 매개변수 조합

- `user_id`는 경로 매개변수, `detailed`는 쿼리 매개변수

```python
@app.get("/users/{user_id}")
def get_user(user_id: int, detailed: bool = False):
    if detailed:
        return {"user_id": user_id, "info": "Detailed user information"}
    return {"user_id": user_id}
```

FastAPI 비동기 처리 (Asynchronous)

[실습 5] `/async-items` API 만들기

```python
import asyncio

@app.get("/async-items/")
async def get_async_items():
    await asyncio.sleep(2)  # 2초 지연
    return {"message": "This is an async response"}
```

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 기본 (2)

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 기본 개념 및 모델 정의

[실습 1] Order와 Item 모델 정의하기: 아래의 조건을 만족하도록 모델들을 정의해주세요.

- Item 모델
  - 필드: name (문자열), quantity (정수, 1 이상)
- Order 모델
  - 필드: id (정수), items (여러 개의 Item 객체), -> 힌트: List 사용하기
    total_price (소수, 0 이상)

`/orders/`로 POST 요청을 받아 주문 데이터를 생성하는 API 작성

- 최소 2개의 상품을 포함하는 요청 JSON 작성

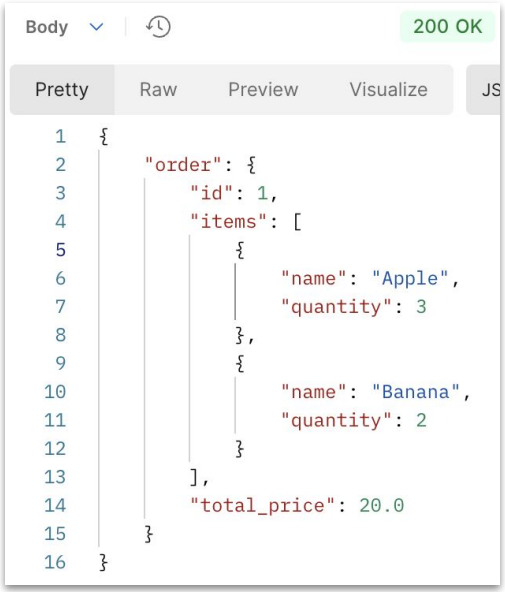
Pydantic 기본 개념 및 모델 정의

[실습 1] Order와 Item 모델 정의하기: 아래의 조건을 만족하도록 모델들을 정의해주세요.

요청 JSON

```json
{
    "id": 1,
    "items": [
        {"name": "Apple", "quantity": 3},
        {"name": "Banana", "quantity": 2}
    ],
    "total_price": 20.0
}
```

응답 JSON



Pydantic 의 validator 개념 및 사용법

validator란?

- Pydantic에서 데이터 모델의 특정 필드에 대해 **추가적인 검증 로직**을 구현할 때 사용하는 데코레이터

- 단순한 데이터 타입 검증 이상으로, 조건을 커스터마이징하거나 값 변환을 수행할 수 있음

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
```

Pydantic 기본 개념 및 모델 정의

[실습 2] Reservation 모델 정의하기: 아래의 조건을 만족하도록 모델들을 정의해주세요.

- 필수 필드
  - name: 예약자 이름 (최대 50자)
  - email: 예약자 이메일
  - date: 예약 날짜 (미래 날짜만 허용) -> 힌트: `datetime.now()`
- 선택적 필드
  - special_requests: 문자열 (기본값: 빈 문자열)

Pydantic 기본 개념 및 모델 정의

[실습 2] Reservation 모델 정의하기: 아래의 조건을 만족하도록 모델들을 정의해주세요.

- 요구사항
  - `/reservations/` 로 POST 요청을 받아 예약 데이터를 생성하는 API 작성
  - 아래 데이터를 검증해주세요:
    - 날짜가 과거일 경우 에러 반환
    - 이름이 50자를 초과할 경우 에러 반환

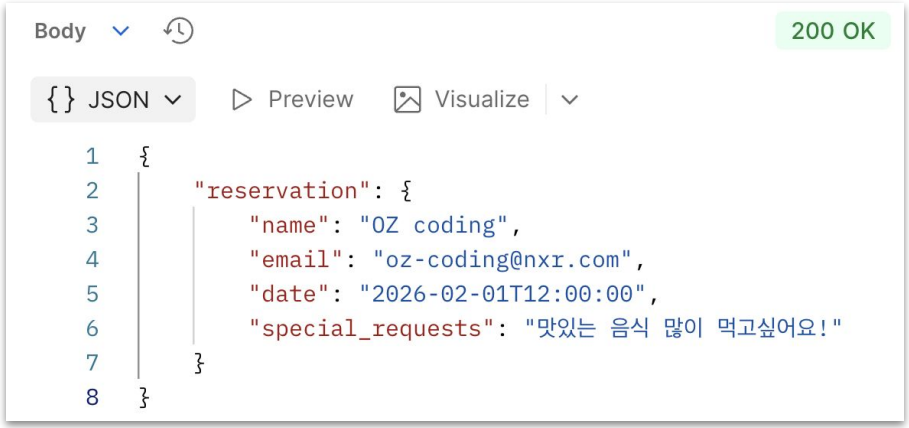
Pydantic 기본 개념 및 모델 정의

[실습 2] Reservation 모델 정의하기: 아래의 조건을 만족하도록 모델들을 정의해주세요.

요청 JSON (26년 1월 1일 오전 12시 실행)

```json
{
  "name": "OZ coding",
  "email": "oz-coding@nxr.com",
  "date": "2026-02-01T12:00:00",
  "special_requests": "맛있는 음식 많이 먹고싶어요!"
}
```

응답 JSON



Pydantic 기본 개념 및 모델 정의

[실습 2] Reservation 모델 정의하기: 아래의 조건을 만족하도록 모델들을 정의해주세요.

요청 JSON

```json
{
  "name": "OZ coding",
  "email": "oz-coding@nxr.com",
  "date": "2026-01-01T12:00:00"
}
```

응답 JSON

Body 422 Unprocessable Content 5 ms 366 B

{} JSON ▷ Preview Debug with AI


```json
{
    "detail": [
        {
            "type": "value_error",
            "loc": [
                "body",
                "date"
            ],
            "msg": "Value error, Reservation date must be in the future",
            "input": "2026-01-01T12:00:00",
            "ctx": {
                "error": {}
            },
            "url": "https://errors.pydantic.dev/2.12/v/value_error"
        }
    ]
}
```

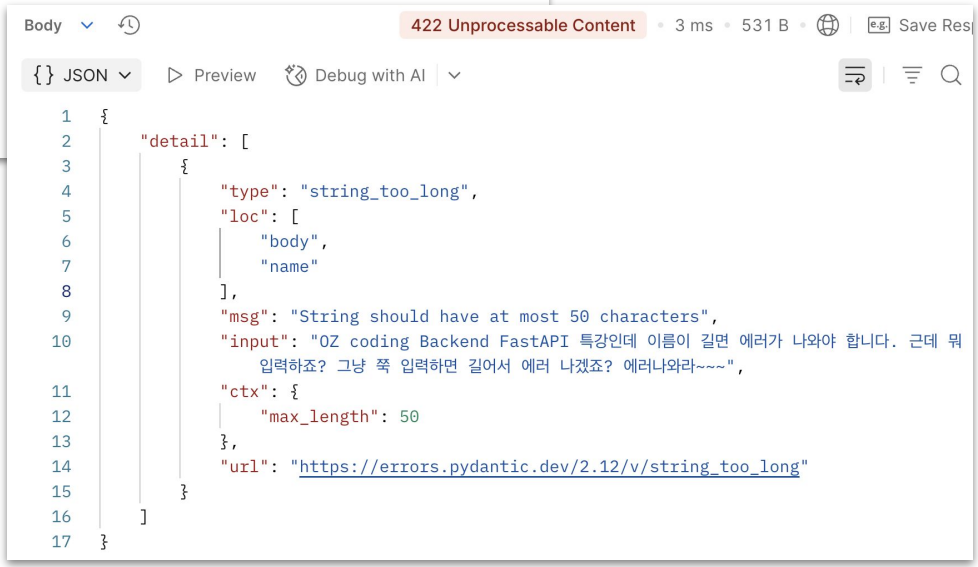
Pydantic 기본 개념 및 모델 정의

[실습 2] Reservation 모델 정의하기: 아래의 조건을 만족하도록 모델들을 정의해주세요.

요청 JSON

```
{
  "name": "OZ coding Backend FastAPI 특강인데 이름이 길면 에러가 나와야 합니다. 근데 뭐 입력하죠? 그냥 쭉
    입력하면 길어서 에러 나겠죠? 에러나와라~~~",
  "email": "oz-coding@nxr.com",
  "date": "2026-02-01T12:00:00"
}
```

응답 JSON



Pydantic 심화

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: 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 3] 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
```

```python
@app.post("/contact")
def create_contact(contact: ContactInfo):
    return {
        "message": "Contact info accepted",
        "data": contact
    }
```

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 4] price(원가)와 discount(할인율)를 입력하면 자동으로 final_price(할인가)를 계산하도록 설계하세요.

```python
@app.post("/products")
def create_product(product: Product):
    return product
```

- **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: 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 5] 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

QnA