PNU Mini Bootcamp 백엔드&클라우드 과정

4일차 - 프로젝트 설계와 초기 구현

송준우

2025년 2월 6일

멋쟁이사자처럼

목차

1. Pydantic을 활용한 데이터 검증

1.1.1 Pydantic 기초

Pydantic은 데이터 검증에 유용한 Python 라이브러리 입니다.
지금까지 모델 클래스 코드를 작성할 때 dataclass를 이용했었습니다.

Pydantic 라이브러리는 다양한 기본 검증 메서드들을 지원하며
가장 많이 사용되는 데이터 교환 형식인 JSON과 Python의 dict로의
편리한 직렬화 기능들을 제공합니다.

새로운 프로젝트 폴더를 생성하고 가상환경을 구축해봅시다.

```
mkdir pydanticex
cd pydanticex
python3 -m venv .venv
source .venv/bin/activate
(.venv) pip3 install pydantic
```

1.1.2 데이터 검증 기초

Pydantic으로 모델 클래스를 작성할 때는 pydantic.BaseModel을 상속받아 구현합니다.

main1.py 파일을 생성하고 아래 예제를 실행해봅시다.

```
from pydantic import BaseModel

class User(BaseModel):
    id: int
    name: str
    email: str
    is_active: bool = True

user = User(id=1, name="Linux", email="linux@linux.com")
print(user)
```

1.1.2 데이터 검증 기초

BaseModel을 상속받은 클래스 인스턴스를 생성하면 Pydantic은 입력받은 데이터의 유형을 검증합니다.

만약 아래와 같이 잘못된 타입의 데이터를 입력하면 pydantic.ValidationError 예외를 발생시킵니다.

```
user2 = User(id="myId", name="iPhone", email="apple@apple.com")
print(user2)
```

1.1.2 데이터 검증 기초

Pydantic은 잘못된 타입의 데이터가 입력되어도 타입 변환이 가능하다면 자동으로 변환을 시도합니다.

```
user3 = User(id="12", name="Linux", email="linux@linux.com")
print(user3)
```

1.1.2 데이터 검증 기초

Pydantic의 BaseModel은 중첩하여 사용 가능합니다.

```
class Address(BaseModel):
    country: str
    city: str

class User(BaseModel):
    id: int
    name: str
    email: str
    is_active: bool = True
    address: Address
```

1.1.2 데이터 검증 기초

생략 가능한 항목에는 typing.Optional을 사용하여 정의해줍니다.

```
from typing import Optional

class User(BaseModel):
    id: int
    name: str
    email: str
    is_active: bool = True
    address: Optional[Address] = None
```

1.1.2 데이터 검증 기초

모델 클래스 직렬화

Model →JSON 문자열

```
user_json = user.model_dump_json()
```

Model →dict

```
user_json = user.model_dump()
```

1.1.2 데이터 검증 기초

dict →Model

```
userDict = {
    "id": 99,
    "name": "Linux",
    "email": "linux@linux.com",
    "is_active": False
}
user4 = User(**userDict)
print(user4)
```

JSON 문자열 →Model

```
str_user_json = '{"id":100,"name":"Linux","email":"email@mail.com"}'
user5 = User.model_validate_json(str_user_json)
print(user5)
```

1.2.1 Literal type

Python의 typing.Literal을 사용하면 선택 가능한 값 중 한가지를 담을 수 있는 데이터를 정의할 수 있습니다.

```
from pydantic import BaseModel, ValidationError
from typing import Literal

class Car(BaseModel):
    color: Literal['red', 'blue', 'green']

Car(color='red')
Car(color='green')
try:
    Car(color='yellow')
except ValidationError as e:
    print(e)
```

1.2.2 Boolean

Pydantic은 아래의 값들을 모두 bool 값으로 처리합니다.

- True, False
- 0, 1
- '0', '1'
- 'off', 'on'
- 'f', 't'
- 'false', 'true'
- 'n', 'y'
- 'no', 'yes'

'off', 'yes'와 같은 문자열 값들은 대소문자 구분이 없음

1.2.2 Boolean

```
class BoolModel(BaseModel):
    val: bool

print(BoolModel(val=False))
print(BoolModel(val='False'))
print(BoolModel(val=1))
print(BoolModel(val='yes'))
print(BoolModel(val='hello'))
```

1.2.3 Callable

Callable은 함수 타입을 검증하는데 사용합니다.

```
from typing import Callable

class Methods(BaseModel):
    callback: Callable[[int], str]

def int_to_str(val: int) -> str:
    return f'{val}'

m = Methods(callback=int_to_str)
print(type(m.callback(1)))
```

1.2.4 Datetimes

숫자, 문자열 등 다양한 형태로 datetime을 검증할 수 있습니다.

```
from datetime import date, datetime, time, timedelta
class DTModel(BaseModel):
    d: date = None
    dt: datetime = None
    t: time = None
    td: timedelta = None

dtModel = DTModel(
    d=1738195200.0,
    dt='2025-01-31T10:20:30.400+09:00',
    t=time(10,12,13),
    td='P3DT12H30M5S'
)
print(dtModel)
```

1.2.5 JSON

JSON 문자열을 위한 Json 타입

```
from typing import Any, List
from pydantic import BaseModel, ValidationError, Json

class AnyJsonModel(BaseModel):
    json: Json[Any]

class ConstrainedJsonModel(BaseModel):
    json: Json[List[int]]

print(AnyJsonModel(json='{"Hello": 3.14}'))
print(ConstrainedJsonModel(json='[3,1,4]'))
```

1.2.6 Number

Pydantic은 Python 표준 라이브러리의 숫자타입들에 더하여
제약조건을 추가할 수 있는 몇가지 숫자 타입을 제공합니다.

기본 숫자 자료형

- int: int(v)로 강제변환 가능한 값
- float: float(v)로 강제변환 가능한 값
- decimal.Decimal: Deciaml(v)로 강제변환 가능한 값
- enum.IntEnum

enum.IntEnum

```
from enum import IntEnum
class CarType(IntEnum):
    suv = 1
    bus = 2
```

1.2.6 Number

Constrained types

- conint: int형에 제약조건 설정
- confloat: float형에 제약조건 설정
- condecimal: decimal.Decimal형에 제약조건 설정

제약조건

- gt: greater than
- ge: greater than or equal to
- lt: less than
- le: less than or equal to
- multiple_of: multiple of
- strict: 강제변환 허용 여부

Constrained types 예시

```
from pydantic import BaseModel, conint

class TestModel(BaseModel):
    age: conint(gt=0, lt=150)

try:
    print(TestModel(age=10))
    print(TestModel(age=200))
except Exception as e:
    print(e)
```

1.2.6 Number

Constrained integers

- PositiveInt
- NegativeInt
- NonPositiveInt
- NonNegativeInt

```
from pydantic import BaseModel, PositiveInt

class PositiveModel(BaseModel):
    val: PositiveInt

try:
    print(PositiveModel(val=10))
    print(PositiveModel(val=0))
    print(PositiveModel(val=-10))
except Exception as e:
```

1.2.6 Number

Constrained floats

- PositiveFloat
- NegativeFloat
- NonPositiveFloat
- NonNegativeFloat

1.2.7 Secret Types

User 정보를 표현하는 모델 클래스에서 패스워드와 같은 민감한 정보를 담고있다면 개발자가 직접 패스워드 항목을 숨김처리 후 JSON으로 변환하여 응답해야합니다.
만약 실수로 비식별화를 누락시키면 심각한 보안 사고를 초래할 수 있습니다.
Pydantic은 이 경우를 위해 특수한 타입인 SecretStr을 지원합니다.
SecretStr은 JSON으로 변환할때 해당 문자열을 ******과 같이 숨김처리를 자동으로 처리해줍니다.

```
from pydantic import BaseModel, SecretStr
class SecretUserModel(BaseModel):
    login_id: str
    pwd: SecretStr

sm = SecretUserModel(login_id='linux', pwd='1234')
print(sm.pwd)
print(sm.model_dump())
```

1.2.8 String Types

URLs

- AnyUrl
- AnyHttpUrl
- HttpUrl
- FileUrl

Email

email-validator를 설치하면 email 주소형식 검증을 자동화할 수 있습니다.

```
pip3 install email-validator
```

- EmailStr: E-mail 주소 문자열
- NameEmail: '.'으로 구분된 주소형식(song.linux@example.com)

1.2.8 String Types

Constrained String

constr을 이용하여 대소문자, 길이, 정규표현식으로 문자열 형식을 지정할 수 있습니다.

- strip_whitespace: bool = False
- to_upper: bool = False
- to_lower: bool = False
- min_length: int = None
- max_length: int = None
- pattern: str = None

```
from pydantic import BaseModel, constr
class StringModel(BaseModel):
    title: constr(strip_whitespace=True, min_length=1, max_length=20)
print(StringModel(title=" Hello All! "))
```

@field_validator를 이용하여 검증 로직을 직접 추가할 수 있습니다.

```
from pydantic_core.core_schema import FieldValidationInfo
from pydantic import BaseModel, ValidationError, field_validator,
    EmailStr, SecretStr

class UserModel(BaseModel):
    login_id: str
    email: EmailStr
    password1: SecretStr
    password2: SecretStr

    @field_validator('login_id')
    def login_id_alphanumeric(cls, v):
        assert v.isalnum(), 'must be alphanumeric'
        return v
```

1.3.1 field_validator ii

```
    @field_validator('password2')
    def passwords_match(cls, v, info: FieldValidationInfo):
        if 'password1' in info.data and v != info.data['password1']:
            raise ValueError('passwords do not match')
        return v

user = UserModel(
    login_id='linux',
    email='linux@example.com',
    password1='1234',
    password2='1234'
)
print(user)
```

1.3.2 Validator의 재사용 i

여러 BaseModel 클래스에 동일한 검증 조건이 필요하다면 검증용 함수를 미리 작성하여 여러 클래스에 재사용할 수 있습니다.

```
from pydantic import BaseModel, field_validator

def validate_alpha_numeric(v: str) -> str:
    assert v.isalnum(), 'must be alphanumeric'
    return v

class Reader(BaseModel):
    login_id: str

    validate_login_id =
        field_validator('login_id')(validate_alpha_numeric)

class Writer(BaseModel):
```

1.3.2 Validator의 재사용 ii

```
    login_id: str

    validate_login_id =
        field_validator('login_id')(validate_alpha_numeric)

print(Reader(login_id='aaa'))
print(Writer(login_id='b@'))
```

1.4 Model Config

Model Config를 이용하면 모델 클래스 내의 모든 속성들에 동일한 제약 조건을 설정할 수 있고 또는 클래스에 없는 속성의 추가를 허용하거나 금지시킬 수도 있습니다.

ConfigDict

```
from pydantic import BaseModel, ConfigDict

class ConfigModel(BaseModel):
    model_config = ConfigDict(str_max_length=10)

    id: int
    name: str
    phone: str

try:
    print(ConfigModel(id=1, name='abc', phone='1234'))
    print(ConfigModel(id=12345678900, name='cdf', phone='0' * 20))
except Exception as e:
    print(e)
```

Alias

모델 클래스의 속성에 별칭을 지정할 수 있습니다.

```
from pydantic import BaseModel, Field

class BookModel(BaseModel):
    name: str
    language_code: str = Field(alias='lang')

book1 = BookModel(name='Linux', lang='KO')
print(book1)
print(book1.model_dump())
print(book1.model_dump(by_alias=True))
```

1.4 Model Config

Extra Attributes

ConfigDict의 extra 설정을 통해 모델 클래스에 선언되지 않은 속성이 추라되는것을 허용하거나 비허용시킬 수 있습니다.

- allow: 속성추가 허용
- forbid: 속성이 추가되면 예외 발생
- ignore(기본값): 추가된 속성을 무시함

Extra Attributes

```
from pydantic import BaseModel, ConfigDict
class ExtModel1(BaseModel):
    name: str
class ExtModel2(BaseModel):
    model_config = ConfigDict(extra='allow')
    name: str
class ExtModel3(BaseModel):
    model_config = ConfigDict(extra='forbid')
    name: str

try:
    print(ExtModel1(name='Linux', age=20))
    print(ExtModel2(name='Linux', age=20))
    print(ExtModel3(name='Linux', age=20))
except Exception as e:
    print(e)
```

Validate Assignment

Pydantic은 모델이 처음 생성될 때만 검증을 처리합니다.
아래 예제에서 name 속성은 str 형식이지만
모델을 생성 후 값을 변경시킬때는 타입 검증이 무시되는 것을 확인할 수 있습니다.

```
from pydantic import BaseModel

class VAUser(BaseModel):
    name: str
vaUser = VAUser(name='Linux')
vaUser.name = 123
print(vaUser)
```

Validate Assignment

validate_assignment를 True로 설정하면
모델 생성 후 값을 변경시킬 때도 타입 검증이 수행됩니다.

```
from pydantic import BaseModel

class VAUser2(BaseModel, validate_assignment=True):
    name: str
vaUser = VAUser2(name='Linux')
vaUser.name = 123
```

1.5 Field

Field를 이용하면 모델의 속성에 메타데이터를 추가하거나 커스터마이징 할 수 있습니다.

기본값 지정하기

```
from pydantic import (
    BaseModel,
    Field
)

class User(BaseModel):
    name: str = Field(default='Linux')

print(User())
```

1.5 Field

Field의 default_factory 속성을 사용하면 기본값을 동적으로 생성할 수 있습니다.

```
from uuid import uuid4
from pydantic import (
    BaseModel,
    Field
)

def generate_uuid4() -> int:
    return uuid4().hex

class User(BaseModel):
    id: int = Field(default_factory=generate_uuid4)
    name: str = Field(default='Linux')

print(User())
```

alias

```python
from pydantic import BaseModel,Field

class User(BaseModel):
    name: str = Field(..., alias='username')

user = User(username='Linux')
print(user)

user.name='hello'
print(user)
# name='hello'
print(user.model_dump(by_alias=True))
# {'username': 'hello'}
```

validation_alias: 검증시에만 사용

```
from pydantic import BaseModel,Field

class User(BaseModel):
    name: str = Field(..., validation_alias='username')

user = User(username='Linux')
print(user)

user.name='hello'
print(user)
# name='hello'
print(user.model_dump())
# {'name': 'hello'}
```

serialization_alias: 직렬화시에만 사용(JSON변환 등)

```python
from pydantic import BaseModel,Field

class User(BaseModel):
    name: str = Field(..., serialization_alias='username')

user = User(name='Linux')
print(user)

user.name='hello'
print(user)
# name='hello'
print(user.model_dump(by_alias=True))
# {'username': 'hello'}
```

AliasPath

dict 값의 특정 키에 해당하는 배열값들을 모델 클래스의 특정 속성에 대응시킬 수 있습니다.

아래 예시는 names라는 키의 값인 문자열 배열의 값들 중 특정 위치의 값을 모델의 특정 속성에 대입시키는 예시입니다.

```
from pydantic import BaseModel,Field,AliasPath
class User(BaseModel):
    first_name: str = Field(validation_alias=AliasPath('names', 0))
    last_name: str = Field(validation_alias=AliasPath('names', 1))

dictVal = {'names': ['Linux', 'Song']}
user = User.model_validate(dictVal)
print(user)
# first_name='Linux' last_name='Song'
```

AliasChoices

dict 값의 특정 키에 해당하는 단일값을 모델 클래스의 특정 속성에 대응시킬 수 있습니다.

```
from pydantic import BaseModel,Field,AliasChoices

class User(BaseModel):
    first_name: str = Field(validation_alias=AliasChoices('first_name',
        'fname'))
    last_name: str = Field(validation_alias=AliasChoices('last_name',
        'lname'))

dictVal = { 'fname': 'Linux', 'lname': 'Song'}
user = User.model_validate(dictVal)
print(user)
# first_name='Linux' last_name='Song'
```

AliasPath와 AliasChoices의 조합

```
from pydantic import BaseModel,Field,AliasChoices,AliasPath
class User(BaseModel):
    first_name: str = Field(
        validation_alias=AliasChoices('first_name', AliasPath('names',
            0))
    )
    last_name: str = Field(
        validation_alias=AliasChoices('last_name', AliasPath('names', 1))
    )

dictVal = {'names': ['Linux', 'Song']}
user = User.model_validate(dictVal)
print(user)
# first_name='Linux' last_name='Song'
```

Numeric Constraints

Field를 이용해 숫자 제약조건을 정의할 수 있습니다.

- gt: greater than
- lt: less than
- ge: greater than or equal to
- le: less than or equal to
- muliple_of: 배수
- allow_inf_nan: 무한대 또는 NaN 허용

Numeric Constraints

```python
from pydantic import BaseModel,Field
class NumModel(BaseModel):
    n1: int = Field(gt=0)
    n2: int = Field(ge=0)

    n3: int = Field(lt=0)
    n4: int = Field(le=0)

    even: int = Field(multiple_of=2)

num = NumModel(
    n1=1, n2=0,
    n3=-1, n4=0,
    even=4
)
print(num)
#n1=1 n2=0 n3=-1 n4=0 even=4
```

1.5 Field

String Constraints로 문자열의 길이나 정규표현식을 통한 패턴 제약을 정의할 수 있습니다.

- min_length
- max_length
- pattern

```
from pydantic import BaseModel, Field
class StrModel(BaseModel):
    short: str = Field(min_length=3)
    regex: str = Field(pattern=r'^\d*$')
```

https://docs.python.org/ko/3.13/howto/regex.html

1.5 Field

Decimal Constraints를 사용하여 Decimal 숫자의 최대 자릿수와 소수점 이하의 자릿수를 고정할 수 있습니다.

- max_digits: 최대 자릿 수
- decimal_places 소수점 이하 부분의 자릿 수

```
from decimal import Decimal
from pydantic import BaseModel, Field
class DecModel(BaseModel):
    precise: Decimal = Field(max_digits=5, decimal_places=2)

val = DecModel(precise=Decimal('123.45'))
print(val)
# precise=Decimal('123.45')
```

Immutability

frozen 파라메터로 한번 대입된 속성을 변경할 수 없도록 제약을 설정할 수 있습니다.

```
from pydantic import BaseModel,Field,ConfigDict

class User(BaseModel):
    model_config = ConfigDict(validate_assignment=True)
    name: str = Field(frozen=True)

user = User(name='Linux')
try:
    user.name = 'Windows'
except Exception as e:
    print(e)
print(user)
# name='Linux'
```

Include and Exclude

include, exclude 파라메터로 모델을 JSON이나 dict로 변환할 때
해당 속성을 포함시킬지 여부를 설정할 수 있습니다.

```
fromm pydantic import BaseModel,Field
class Guest(BaseModel):
    name: str
    age: int = Field(exclude=True)

guest = Guest(name='Song', age=41)
print(guest.model_dump())
# {'name': 'Song'}
```

Computed fields

모델을 생성할 때 속성값이 입력되면 그 값들을 계산한 결과로 새로운 속성을 만들 수 있습니다.

```
from pydantic import BaseModel,computed_field
class Rect(BaseModel):
    width: int
    height: int

    @computed_field
    @property
    def area(self) -> int:
        return self.width * self.height

print(Rect(width=3, height=2))
# width=3 height=2 area=6
```

TypeAdapter

BaseModel로 구현하지 않은 데이터의 검증이 필요한 경우가 있습니다.
예를 들어 TypedDict형의 User클래스의 배열인 값을 검증하고 싶을 때
아래 예와 같이 TypeAdapter를 사용하여 Validator를 구현할 수 있습니다.

```
from typing import List
from typing_extensions import TypedDict
from pydantic import TypeAdapter

class User(TypedDict):
    name: str
    id: int

UserListValidator = TypeAdapter(List[User])
```

1.5 Field ii

```
user1 = {'name': 'Linux', 'id': 1}
userList1 = [user1]

user2 = {'name': 'Windows', 'id': 'hello', 'age': 10}
userList2 = [user2]

try:
    print(UserListValidator.validate_python(userList1))
    print(UserListValidator.validate_python(userList2))
except Exception as e:
    print(e)
```

1.6 환경변수 관리

Blog API 예제에서 데이터베이스 연결 정보를 .env 파일에 저장하고
Python Dotenv 라이브러리로 저장된 환경변수를 불러와 사용하였습니다.
데이터베이스 설정 외에도 현재 프로그램의 실행환경이 개발용 컴퓨터인지
운영용 서버인지 구분할때도 운영체제의 환경변수를 읽어와 판단하기도 합니다.
Pydantic은 환경변수의 값을 불러오고 그 값을 검증하는 자동화 기능을 제공합니다.
Pydantic에서 환경변수를 관리하기 위해서는 아래의 모듈을 설치해야합니다.

```
(.vnev) pip3 install pydantic-settings
```

1.6 환경변수 관리

아래와 같이 BaseSettings를 상속받아 클래스를 생성하면
클래스의 속성명과 동일한 환경변수값을 자동으로 불러옵니다.
먼저 현재 실행중인 컴퓨터에 아래와 같이 2개의 환경변수를 설정해봅시다.

- my_prefix_pass_key=1234
- my_prefix_env=DEBUG

1.6 환경변수 관리

Windows cmd에서 사용자 환경변수 등록방법

```
setx my_prefix_pass_key 1234
setx my_prefix_env DEBUG
```

Linux, macOS 터미널에서 사용자 환경변수 등록방법

```
export my_prefix_pass_key=1234
export my_prefix_env=DEBUG
```

1.6 환경변수 관리

아래 코드를 작성 후 환경변수를 추가했던 cmd 또는 터미널 창에서 실행해봅시다.

main.py

```
from pydantic_settings import BaseSettings, SettingsConfigDict

class Settings(BaseSettings):
    model_config = SettingsConfigDict(env_prefix='my_prefix_')

    pass_key: str
    env: str

print(Settings())
```

```
(.venv) python3 main.py
```

Dotenv support

Blog API 예제에서 작성했던 .env 파일의 변수들은 아래와 같이 불러올 수 있습니다.

.env

```
PASS_KEY=1234
ENV=DEBUG
```

main.py

```
from pydantic_settings import BaseSettings, SettingsConfigDict
class Settings(BaseSettings):
    model_config = SettingsConfigDict(env_file=".env", env_file_encoding="utf-8")
    pass_key: str
    env: str

print(Settings())
```

1.7 FastAPI 적용

새로운 FastAPI 프로젝트를 시작해봅시다.

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

1.7 FastAPI 적용

main.py 파일을 생성하고 아래 코드를 작성 후 FastAPI 서버를 실행해봅시다.

```python
from fastapi import FastAPI
from pydantic import BaseModel, Field, SecretStr

class User(BaseModel):
    login_id: str = Field(min_length=8, max_length=16)
    password: SecretStr = Field(exclude=True)
    name: str = Field(min_length=1, max_length=50)
    age: int = Field(gt=10, lt=150)

app = FastAPI()

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

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