# Programowanie Back-End

## Walidacja danych, ORM

mgr inż. Jakub Gogola

# Walidacja danych

## Dlaczego walidacja jest ważna?

Walidacja danych jest kluczowym elementem bezpieczeństwa i integralności aplikacji backendowej. Poprawna walidacja requestów:

- Chroni przed błędami i atakami, np. SQL Injection czy XSS.
- Umożliwia lepsze wykrywanie błędów użytkownika, co poprawia doświadczenie użytkownika.
- Zapewnia spójność danych w systemie (np. poprawne formaty emaili, liczby w dopuszczalnym zakresie).

FastAPI wykorzystuje bibliotekę **Pydantic**, który pozwala na łatwe definiowanie schematów danych i automatyczną walidację.

# Walidacja danych

## Przykład

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

app = FastAPI()

class User(BaseModel):
    name: str
    age: int
    email: str

@app.post("/users/")
async def create_user(user: User):
    return {"message": f"User {user.name} created!"}
```

# Walidacja danych

## Query string

```python
from fastapi import Query

@app.get("/users/")
async def read_users(age: int = Query(..., gt=18, lt=100)):
    return {"message": f"Users with age {age}"}
```

# Walidacja danych

## Path

```python
from fastapi import Path

@app.get("/users/{user_id}")
async def read_user(user_id: int = Path(..., gt=0)):
    return {"message": f"User ID: {user_id}"}
```

# Walidacja danych

## Body

```python
from pydantic import EmailStr

class User(BaseModel):
    name: str
    email: EmailStr

@app.post("/users/")
async def create_user(user: User):
    return {"message": f"User {user.name} with email {user.email} created!"}
```

# Walidacja danych

## Path - UUID

```python
from fastapi import FastAPI
from uuid import UUID

app = FastAPI()

@app.get("/users/uuid/{user_uuid}")
async def read_user_by_uuid(user_uuid: UUID):
    return {"message": f"User UUID: {user_uuid}"}
```

# Walidacja danych

## Body - złożony przykład

```python
from fastapi import FastAPI
from pydantic import BaseModel, EmailStr, Field
from uuid import UUID

app = FastAPI()

class AdvancedUser(BaseModel):
    id: UUID
    name: str = Field(..., min_length=3, max_length=50)
    age: int = Field(..., ge=18, le=100)
    email: EmailStr
    is_active: bool = True
    roles: list[str] = ["user"]

@app.post("/users/advanced/")
async def create_advanced_user(user: AdvancedUser):
    return {"message": f"Advanced user {user.name} with roles {user.roles} created!"}
```

# Walidacja danych

## Zaawansowane techniki

**Pydantic** to biblioteka, która stanowi rdzeń walidacji danych w FastAPI. Pozwala na precyzyjne określanie typów danych i reguł walidacji, które są stosowane podczas przetwarzania danych wejściowych. W tej sekcji omówimy bardziej zaawansowane techniki walidacji, takie jak:

- Walidacja z użyciem `@validator` i `@field_validator`
- Tworzenie własnych niestandardowych walidatorów
- Dziedziczenie modeli Pydantic
- Walidacja na poziomie parametrów request'u

# Walidacja danych

## Typy danych i ich ograniczenia

Pydantic pozwala na stosowanie zaawansowanych typów danych i ich ograniczeń. Typy danych, takie jak `EmailStr`, `UUID`, `constr` (dla łańcuchów znaków) czy `condecimal` (dla liczb zmiennoprzecinkowych) umożliwiają bardziej precyzyjne walidowanie danych.

```
class AdvancedUser(BaseModel):
    id: UUID
    name: str = Field(..., min_length=3, max_length=50)
    age: int = Field(..., ge=18, le=100)
    email: EmailStr
    is_active: bool = True
    roles: list[str] = ["user"]
```

# Walidacja danych
## @validator

```
from pydantic import BaseModel, validator
from datetime import datetime

class Event(BaseModel):
    name: str
    date: datetime

    @validator('date')
    def validate_date(cls, v):
        if v < datetime.now():
            raise ValueError('Date must be in the future!')
        return v
```

# Walidacja danych
## @field_validator

```python
class User(BaseModel):
    name: str
    age: int
    email: EmailStr

    # Walidacja pola 'age' za pomocą @field_validator
    @field_validator('age')
    def validate_age(cls, v):
        if v < 18:
            raise ValueError('Age must be greater than or equal to 18')
        return v

    # Walidacja pola 'email' za pomocą @field_validator
    @field_validator('email')
    def validate_email(cls, v):
        blocked_domains = ["blocked.com", "spam.com"]
        domain = v.split('@')[-1]
        if domain in blocked_domains:
            raise ValueError(f"Email domain '{domain}' is blocked")
        return v

# Przykład użycia
try:
    user = User(name="John Doe", age=17, email="john@blocked.com")
except ValueError as e:
    print(f"Error: {e}")

# Prawidłowy przypadek
user = User(name="Jane Doe", age=25, email="jane@valid.com")
print(user)
```

# Walidacja danych

## Czym różnią się od siebie `@validator` i `@field_validator`?

| | `@validator` | `@field_validator` |
|---|---|---|
| **Zakres walidacji** | Walidator oznaczony tym dekoratorem działa na poziomie całego modelu i umożliwia walidację kilku pól jednocześnie. Możesz zdefiniować logikę walidacji, która sprawdza nie tylko jedno pole, ale także zależności między różnymi polami w modelu. | Walidator stosowany do pojedynczego pola w modelu. Używany w przypadku, gdy chcesz walidować konkretne pole w modelu, bez konieczności sprawdzania innych pól. Jest to bardziej precyzyjne narzędzie do walidacji pojedynczych pól. |
| **Kiedy używać?** | Używaj go, gdy musisz przeprowadzić walidację na kilku polach lub zależności między polami w modelu. Jest idealny, gdy masz logiczne zależności między wartościami różnych pól. | Używaj go, gdy chcesz przeprowadzić walidację tylko dla jednego konkretnego pola. Idealny, gdy masz jedno pole, które wymaga specyficznych zasad walidacji (np. długość ciągu znaków, zakres liczb). |

# Walidacja danych

## Własne walidatory

```python
from pydantic import BaseModel, validator

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

    @validator('name')
    def name_must_be_capitalized(cls, v):
        if not v.istitle():
            raise ValueError('Name must be capitalized')
        return v
```

# Walidacja danych

## Dziedziczenie modeli

```python
class BaseUser(BaseModel):
    name: str
    email: EmailStr

class AdvancedUser(BaseUser):
    age: int
    is_active: bool = True

    @validator('age')
    def age_must_be_valid(cls, v):
        if v < 18:
            raise ValueError('Age must be greater than or equal to 18')
        return v
```

# Walidacja danych

## *Cross validation*

```python
class Purchase(BaseModel):
    product_price: float
    discount: float

    @root_validator
    def check_discount(cls, values):
        product_price = values.get('product_price')
        discount = values.get('discount')

        if discount > product_price:
            raise ValueError('Discount cannot be greater than the product price')

        return values
```

# ORM

## Co to jest?

**ORM** (ang. *Object-Relational Mapping*) to technika, która umożliwia mapowanie obiektów w aplikacji na dane w relacyjnej bazie danych. ORM pozwala na manipulowanie danymi bazy danych przy pomocy obiektów, co ułatwia pracę z bazą danych, eliminując konieczność pisania skomplikowanych zapytań SQL.

- **Definiowanie modeli danych**: Modele, które reprezentują tabele w bazie danych.
- **Zarządzanie sesjami**: Sesja w ORM służy do interakcji z bazą danych – dodawania, edytowania, usuwania i pobierania danych.
- **Migracje**: Zmiany w strukturze bazy danych, które muszą być synchronizowane z kodem aplikacji.

# ORM
## SQLAlchemy

**SQLAlchemy** to jeden z najpopularniejszych ORM-ów w Pythonie, oferujący:

- **Mocne wsparcie dla baz danych**: Obsługuje wiele baz danych, takich jak PostgreSQL, MySQL, SQLite.
- **Elastyczność**: SQLAlchemy oferuje dwa główne podejścia do ORM:
  - **ORM**: Abstrakcja na poziomie obiektów (np. definiowanie klas, które mapują tabele w bazie).
  - **Core**: Niższy poziom abstrakcji, który pozwala na ręczne pisanie zapytań SQL w Pythonie, ale z zachowaniem pewnej struktury.
- **Wsparcie dla migracji**: SQLAlchemy współpracuje z Alembic (narzędzie do zarządzania migracjami bazy danych).

# SQLAlchemy

## Model

Modele w SQLAlchemy definiują strukturę tabel w bazie danych. W modelu musisz zdefiniować kolumny tabeli oraz ich typy.

```
from sqlalchemy import Column, Integer, String, Boolean
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, index=True)
    email = Column(String, unique=True, index=True)
    is_active = Column(Boolean, default=True)
```

# SQLAlchemy

## Sesja

Sesje w SQLAlchemy służą do interakcji z bazą danych. Sesja przechowuje wszystkie operacje, które mają być wykonane na bazie danych (np. dodawanie, usuwanie danych).

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

DATABASE_URL = "sqlite:///./test.db"
engine = create_engine(DATABASE_URL, connect_args={"check_same_thread": False})
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

# Tworzenie sesji
db = SessionLocal()
```

# Przykłady z wykładu

- https://github.com/JakubGogola/dsw-backend-programming/tree/main/lectures/lecture-2

# Dziękuję za uwagę!