FastAPI Day 5 - Pydantic Models & Validations

1. Introduction to Pydantic

- Pydantic is used for **data validation and serialization** in FastAPI.
- Ensures **type safety** and provides **automatic error handling**.
- Generates **Swagger docs** automatically for defined models.

2. Basic Pydantic Models

```
from pydantic import BaseModel
from typing import Optional

class Product(BaseModel):
    name: str
    price: float
    phone: int
    age: Optional[int] = 18
```

- `Optional` fields can be skipped in request. - Default values can be assigned directly.

3. POST Endpoints with Models

```
from fastapi import FastAPI

app = FastAPI()

@app.post('/create')
def create_product(pro: Product):
    return {
        'message': 'Product created successfully',
        'product': pro
    }
```

- FastAPI automatically validates request body. - Returns 422 error if validation fails.

4. Nested Models & Lists

```
from typing import List

class Order(BaseModel):
    product_name: str
    quantity: int
```

```python
class Customer(BaseModel):
    name: str
    orders: List[Order]
```

- Allows validation of **nested objects and lists**. - Each object inside the list is validated separately.

5. Field Validations

```python
from pydantic import Field
from typing import Optional

class FieldExample(BaseModel):
    name: str = Field(..., min_length=2)
    phone: int = Field(..., gt=1000)
    age: Optional[int] = Field(18, ge=18, lt=100)
```

- `gt`, `ge`, `lt`, `le` → numeric validations. - `min_length`, `max_length` → string length validation. - `...` → marks a field as required.

6. Response Models

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

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

@app.post('/create-user/', response_model=UserOut)
def create_user(user: User):
    return user
```

- Controls **what is returned**. - Hides sensitive fields (like `password`). - Swagger UI shows only the response model fields.

7. Nested Validations

```python
class Order(BaseModel):
    product_name: str = Field(..., min_length=2)
    quantity: int = Field(..., gt=0)

class Customer(BaseModel):
    name: str
    orders: List[Order]
```

```
@app.post('/create-customer/')
def create_customer(customer: Customer):
    return {
        'message': 'Customer created successfully',
        'customer': customer
    }
```

- Each item in the nested list is validated individually.

8. Alias & Serialization

```
class User(BaseModel):
    full_name: str = Field(..., alias='fullName')
    age: int

@app.post('/create-user/')
def create_user(user: User):
    return user
```

- Maps **JSON keys** to Python variables. - Supports different naming conventions (camelCase vs snake_case).

9. Swagger & Extra Options

```
class Product(BaseModel):
    name: str = Field(..., example='Laptop')
    price: float = Field(..., example=1500.00)

app = FastAPI(title='My API', description='Manages products')

@app.get('/old-endpoint/', deprecated=True)
def old_endpoint():
    return {'message': 'Deprecated endpoint'}
```

- `example` → shows example input in Swagger. - `title` & `description` → customizes API metadata. - `deprecated=True` → marks endpoints as deprecated.

Key Takeaways

1. Pydantic models simplify **data validation**.
2. Field validations enforce **rules automatically**.
3. Nested models handle **complex data structures**.
4. Response models control **returned data**.
5. Aliases & Swagger options improve **API clarity and usability**.

Day 5 is fully covered with Pydantic, validations, nested models, aliases, response models, and Swagger options.