Módulo 3: Validación de Datos con Pydantic

En el Módulo 2, aprendimos a recibir datos simples a través de la ruta (Path Parameters), como item_id: int. Pero, ¿qué sucede cuando necesitamos recibir datos complejos, como un JSON en el cuerpo (*body*) de una petición POST?

3.1. El Problema: "Garbage In, Garbage Out" (GIGO)

Imagina que queremos crear un nuevo "ítem" en nuestra base de datos. El cliente debería enviarnos un JSON como este:

```
{
  "name": "Laptop",
  "description": "Un portátil de alto rendimiento",
  "price": 1299.99,
  "tax": 129.99
}
```

Pero, ¿qué pasa si el cliente envía esto?

```
{
  "name": 12345,
  "prce": "un millon",
  "descripcion": null
}
```

Esto es un desastre.

1. name debería ser un *string*, no un *integer*.
2. price (escrito como prce) falta y, aunque estuviera, "un millon" no es un número.
3. description (escrito como descripcion) tiene un *typo*.
4. tax (que podría ser opcional) falta.

Si nuestro código de Python (main.py) recibe esto sin validar, fallará estrepitosamente. O peor, podría intentar guardarlo, corrompiendo nuestra base de datos con datos incompletos o inválidos.

Necesitamos un "guardián" en la puerta de nuestra API que valide rigurosamente la estructura y los tipos de *todos* los datos entrantes. Ese guardián es **Pydantic**.

3.2. Pydantic: Definiendo nuestros *Schemas*

Pydantic es una librería que te permite definir "schemas" (esquemas) de datos usando clases de Python y *type hints*.

La idea central es heredar de BaseModel:

```
# main.py
```

```
from pydantic import BaseModel
from typing import Optional # Necesario para Python < 3.10

# 1. Definimos nuestro "schema" como una clase
class Item(BaseModel):
    name: str
    description: str | None = None  # Python 3.10+ (o Optional[str] = None)
    price: float
    tax: float | None = None
```

Analicemos qué acabamos de definir:

- class Item(BaseModel): Creamos un modelo de datos llamado Item.
- name: str: Declaramos un campo name que **debe** ser un *string*. Es **obligatorio**.
- description: str | None = None: Declaramos un description que puede ser un *string* o None. Es **opcional** (gracias a = None).
- price: float: Un campo price **obligatorio** que debe ser un *float*.
- tax: float | None = None: Un campo tax **opcional** que, si existe, debe ser un *float*.

¿Qué hace Pydantic con esto (gratis)?

1. **Validación de Tipos:** Si price se recibe como "1299.99" (un string), Pydantic es lo suficientemente inteligente como para *coercionarlo* (convertirlo) a 1299.99 (float). Si recibe "mil", fallará.
2. **Validación de Requeridos:** Si name o price (que no tienen valor por defecto) faltan en el JSON, Pydantic rechazará la petición.
3. **Valores por Defecto:** Si description o tax faltan, Pydantic les asignará None.
4. **Generación de Errores:** Si la validación falla, Pydantic genera un error JSON detallado que FastAPI le enviará al cliente (el error 422 que ya vimos).
5. **Autocompletado en el Editor:** Tu editor (VSCode, PyCharm) ahora "entiende" la estructura de Item y te dará autocompletado.

3.3. Modelos de Entrada vs. Modelos de Salida

Este es un concepto crucial para una API bien diseñada.

- **Problema:** Cuando un cliente **Crea** un ítem (POST /items), no debería enviarnos el id del ítem. El id es algo que *nuestro servidor* debe generar (ej. id: 1, id: 2, ...).
- **Problema 2:** Cuando el cliente **Lee** un ítem (GET /items/1), nosotros *sí* queremos devolverle el id.

Esto implica que la "forma" (schema) de un ítem al *entrar* es diferente de la "forma" al *salir*.

- **Solución:** Creamos dos (o más) modelos Pydantic.

Paso 1: Crear un ItemBase

Creamos una clase base con los campos comunes para evitar repetir código.

```
# main.py
from pydantic import BaseModel
from typing import Optional
```

```
class ItemBase(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None
```

Paso 2: Crear el modelo de Entrada (ItemCreate)
Este modelo hereda de ItemBase y define lo que el cliente envía para crear.

```
class ItemCreate(ItemBase):
    # Por ahora, no tiene campos extra.
    # Es exactamente lo que el cliente debe proveer.
    pass
```

Paso 3: Crear el modelo de Salida/Almacenamiento (Item)
Este modelo hereda de ItemBase y añade los campos que genera el servidor (como el id).

```
class Item(ItemBase):
    id: int

    # Podríamos tener otros campos generados por el servidor,
    # como owner_id, created_at, etc.
```

3.4. Integrando Pydantic con FastAPI

Ahora, usemos estos modelos en nuestros *endpoints*.

A. Modelos de Entrada (Request Body)

Para que FastAPI valide el cuerpo (body) JSON de una petición, simplemente declaramos el modelo como un *type hint* en un parámetro de la función.

```
# main.py
from fastapi import FastAPI
from pydantic import BaseModel
from typing import Optional

# ... (definiciones de ItemBase, ItemCreate, Item) ...

app = FastAPI()

# Simulación de nuestra BBDD por ahora
db_items = {}

@app.post("/items/")
```

```
def create_item(item: ItemCreate):
    # 1. FastAPI recibe el JSON del body.
    # 2. Intenta validarlo contra el modelo 'ItemCreate'.
    # 3. Si falla -> Devuelve un error 422 automático.
    # 4. Si tiene éxito -> 'item' es una instancia de Pydantic 'ItemCreate'.

    # Ahora podemos trabajar con 'item' como un objeto de Python
    print(f"Creando ítem: {item.name}")
    print(f"Precio: {item.price}")

    # Lógica para "guardarlo" (simulación)
    new_id = len(db_items) + 1

    # Convertimos el modelo Pydantic a un dict para guardarlo
    # .model_dump() es el reemplazo moderno de .dict()
    db_item_data = item.model_dump()
    db_item_data["id"] = new_id # Le asignamos el ID

    db_items[new_id] = db_item_data

    return db_item_data
```

Si ejecutas esto y vas a http://127.0.0.1:8000/docs, verás que Swagger UI ahora te da una caja de texto para que escribas el JSON de un ItemCreate.

B. Modelos de Salida (response_model)

En el ejemplo anterior, create_item devuelve db_item_data (un dict). Esto funciona, pero no es ideal. ¿Qué pasa si nuestro db_item_data tiene campos sensibles que no queremos devolver (ej. internal_cost)?
Podemos *forzar* el schema de la *respuesta* usando el argumento response_model en el decorador.

```
# ... (mismas importaciones) ...

# Usamos nuestro modelo 'Item' (el que tiene 'id') como modelo de respuesta
@app.post("/items/", response_model=Item)
def create_item(item: ItemCreate):

    # ... (misma lógica para crear el new_id y db_item_data) ...

    new_id = len(db_items) + 1
    db_item_data = item.model_dump()
    db_item_data["id"] = new_id
```

```
    db_items[new_id] = db_item_data

    # AHORA, FastAPI hace lo siguiente:
    # 1. Coge el objeto que devolvemos (db_item_data).
    # 2. Lo valida contra el 'response_model' (Item).
    # 3. Si le falta el 'id', ¡falla! (Error interno del servidor).
    # 4. Si tiene campos extra, ¡los filtra!
    # 5. Serializa el resultado validado y filtrado como JSON al cliente.
    return db_item_data
```

Beneficios del response_model:

1. **Validación de Salida:** Garantiza que *nuestra* API cumple el contrato que promete.
2. **Filtrado de Datos:** Automáticamente oculta campos que no están definidos en el response_model (¡genial para seguridad!).
3. **Documentación Clara:** Swagger UI ahora sabe *exactamente* cuál es el schema de la respuesta (el modelo Item) y lo documenta.

Código Completo del Módulo 3

Tu main.py ahora debería tener esta estructura.

```
# main.py
from fastapi import FastAPI
from pydantic import BaseModel
from typing import Optional

# --- Modelos Pydantic (Schemas) ---

class ItemBase(BaseModel):
    """Modelo base con campos comunes."""
    name: str
    description: str | None = None
    price: float
    tax: float | None = None

class ItemCreate(ItemBase):
    """Modelo para la creación (entrada)."""
    pass

class Item(ItemBase):
    """Modelo para la lectura (salida/almacenamiento)."""
    id: int

# --- Instancia de la App y BBDD (simulada) ---
```

```
app = FastAPI(
    title="API Módulo 3",
    description="API con validación de Pydantic",
    version="0.3.0",
)

# Simulación de base de datos
db_items = {}

# --- Endpoints ---

@app.get("/")
def read_root():
    return {"status": "API funcionando"}

@app.post("/items/", response_model=Item)
def create_item(item: ItemCreate):
    """
    Crea un nuevo ítem en la base de datos.

    - Recibe un `ItemCreate` en el body.
    - Devuelve un `Item` (con el ID asignado).
    """
    # Lógica de creación
    new_id = len(db_items) + 1

    # Convertimos el modelo de entrada a un dict
    db_item_data = item.model_dump()

    # Añadimos los campos generados por el servidor
    db_item_data["id"] = new_id

    # "Guardamos" en la BBDD
    db_items[new_id] = db_item_data

    # FastAPI validará este dict contra 'response_model=Item'
    return db_item_data

@app.get("/items/{item_id}", response_model=Item)
def read_item(item_id: int):
    """
    Lee un ítem por su ID.
    (Aún no maneja errores 404, eso es Módulo 4)
```

```
    """
    if item_id in db_items:
        return db_items[item_id]

    # Temporal (en Módulo 4 usaremos HTTPException)
    return {"error": "Item not found"}
```